Chapter 6

SVA FOR PROTOCOL INTERFACE

SVA checkers for a sample PCI system

Compliance testing has become one of the major challenges in SOC designs. It is very common for designs to support certain standard protocols. For example, graphics applications might support a standard bus interface such as PCI/PCIX, USB or IEEE 1394 Firewire. These bus interfaces help the designs achieve higher bandwidth of data transmission and also provide a standard method to connect multiple devices. Bus protocols are complex and every device sitting on the bus should be compliant with a list of rules specific to that protocol.

The verification environment built for testing these standard protocol interfaces are often re-usable since the same set of rules applies to any device that supports the specific interface. Verification engineers often develop bus interface models (BIM) of the devices that support a specific interface. The BIM need not replicate the detailed internal functionality of the device. It just has to support the basic handshaking process that is compliant with the specific interface. This helps the verification engineer to create a sample system with the BIM and the Design Under Test (DUT). Tests can be written to create transactions between the BIM and the DUT. While running these tests, specific monitors are written to make sure that the DUT is being absolutely compliant with the standard protocol. Most verification environments create logs of the transactions as seen by the bus. SVA can be used very effectively to create these bus protocol monitors. In this chapter, a sample PCI system is used to demonstrate how SVA checkers are created for a PCI compliant device.

6.1 PCI – A Brief Introduction

The PCI local bus is a high performance, 32-bit or 64-bit bus with multiplexed address and data lines. The bus is intended for use as an interconnect mechanism between highly integrated peripheral controller components, peripheral add-in boards and processor memory systems. A sample PCI compliant device is shown in Figure 6-1.

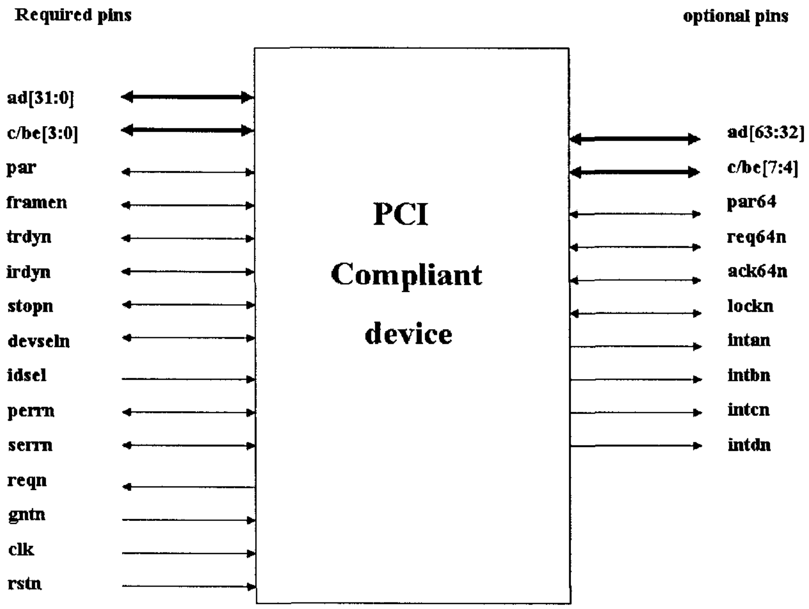

Figure 6-1. PCI compliant device

A brief description of each pin is listed below.

ad[31:0] – the address bus, this has the information on the location to which data is to be transferred or the location from which data should be obtained. This also acts as the data bus.

c/be[3:0] – the command bus, contains one of the twelve commands shown in Table 6-1. It also acts as the byte enable bus that defines which bytes in the data bus are to be transferred.

par – a parity bit, even number of 1's should appear on the ad, c/be and par bits. The required value of the par bit is driven by the device one clock cycle after the device drives the "ad" bus.

framen – frame signal, this is asserted by the master that wants to perform a data transaction. When the frame is asserted, the master also indicates the nature of the transaction by setting the appropriate command on the "c/be" bus. The frame signal is de-asserted when the master is ready to complete the final data transfer.

Table 6-1. PCI Bus commands

C/BE[3:0]	**Command type**
0000	Interrupt Acknowledge
0001	Special cycle
0010	I/O Read
0011	I/O write
0100	Reserved
0101	Reserved
0110	Memory Read
0111	Memory Write
1000	Reserved
1001	Reserved
1010	Configuration Read
1011	Configuration Write
1100	Memory Read Multiple
1101	Dual Address cycle
1110	Memory Read Line
1111	Memory Write and Invalidate

trdyn – target ready signal, this is asserted by the target device that is currently addressed by the master. By asserting this signal the target device lets the master know that it is ready for a data transaction.

irdyn – master ready signal, this is asserted by the master that wants to perform a data transaction.

stopn – stop signal, this is asserted by the target device if it wants to terminate the current transaction. If the target asserts the stop signal without performing any data phases, it is called a retry. If the target asserts the stop signal after performing one or more data phases, it is called a disconnect.

devseln – device select signal, this is asserted by the target device if it is selected. The target ready signal is asserted only after asserting this signal.

idsel – initialization device select signal, is used as a chip select during PCI configuration read and write transactions.

perrn – parity error signal, asserted one clock after a parity error is identified either by the master or a target.

serrn – system error signal, it is an output of both master and target devices. This is asserted only when something fatal occurs.

reqn – request signal, this is used by the master device to request the use of the PCI bus.

gntn – grant signal, this indicates that the PCI device has got the permission to use the PCI bus.

ad[63:32] – upper 32 bits of the data bus, used for 64-bit transactions.

c/be[7:4] – acts as the byte enable bus that defines which of the bytes in the upper data bus is to be transferred in a 64-bit transaction.

par64 – a parity bit, even number of 1's should appear on the ad[63:32], c/be[7:4] and par64 bits. The required value of the par64 bit is driven by the device one clock cycle after the device drives the "ad" bus.

req64n – request signal, this is used by the master device to request the use of the PCI bus for a 64-bit transaction.

ack64n – acknowledge pin, PCI target device acknowledges the 64-bit transaction requested by the master device.

6.1.1 A sample PCI Read transaction

A read transaction is initiated by the master device. The master asserts the "framen" signal and drives an address onto the "ad" bus. It also places a read command on the "c/be" bus. The target device decodes the address and identifies itself. Once it identifies itself, it asserts the "devseln" signal. The master device continues asserting the "framen" signal, but stops driving the address bus. It asserts the signal "irdyn" and also places the byte enable command on the "c/be" bus. In response to this, the target device places the first data on the data bus (ad) and also asserts the signal "trdyn" to acknowledge that the data on the bus is valid. In a multiple data transaction, it is the responsibility of the addressed target to increment the initial address to point to the subsequent data locations.

During a transaction if the target device is not ready to place the next data on the bus, it creates a wait state by de-asserting the signal "trdyn." The signal "devseln" will stay asserted and the data placed on the bus in the previous transaction will stay. The master will read the data only if both "trdyn" and "irdyn" are asserted. When the target is ready to transmit again it will assert the signal "trdyn." The master device indicates that the next data read will be the last one in the current transaction by de-asserting the signal "framen." Once the last data is read, the master de-asserts the signal "irdyn" and the target device de-asserts the "trdyn" and the "devseln" signals respectively. If both the signals "irdyn" and "framen" are de-asserted, the bus is said to be in an idle state. A waveform for a sample PCI read transaction is shown in Figure 6-2.

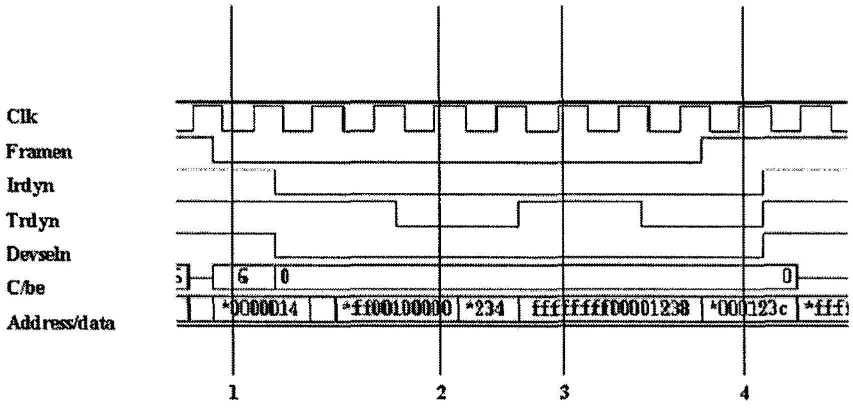

Figure 6-2. Sample PCI read transaction

Marker 1 shows the point where the master issues the read command (0110) on the "c/be" bus. On the same cycle, the address bus also carries the address for the target device. Marker 2 shows the point when the master reads a valid data. Marker 3 shows that the target device de-asserts the "trdy" signal indicating that it is not ready for the read transaction. Marker 4 indicates the last data phase since the signal "framen" is de-asserted. In the next clock cycle, the signals "irdyn," "trdyn" and "devseln" are all de-asserted, indicating the completion of the transaction.

6.1.2 A sample PCI Write transaction

The master device initiates a write transaction. It asserts the signal "framen" and drives an address onto the "ad" bus. It also places the write command on the "c/be" bus. The target device identifies itself and asserts the signals "devseln" and "trdyn." The master continues to assert the "framen" signal. The master places the data on the "ad" bus and also asserts the signal "irdy" to let the target know that the data on the bus is valid. The master also issues the command byte that identifies which bytes are to be written on the same clock cycle. If the master is not ready to place the next data on the bus, it can create a wait state by de-asserting the signal "irdyn." The master will drive the same data from the previous cycle during the wait state. The master will de-assert the "framen" signal just before the last data is ready to be written. Once all the data is written, the master de-asserts the "irdyn" signal and then the target de-asserts the signals "trdyn" and "devseln." A waveform for a sample PCI write transaction is shown in Figure 6-3.

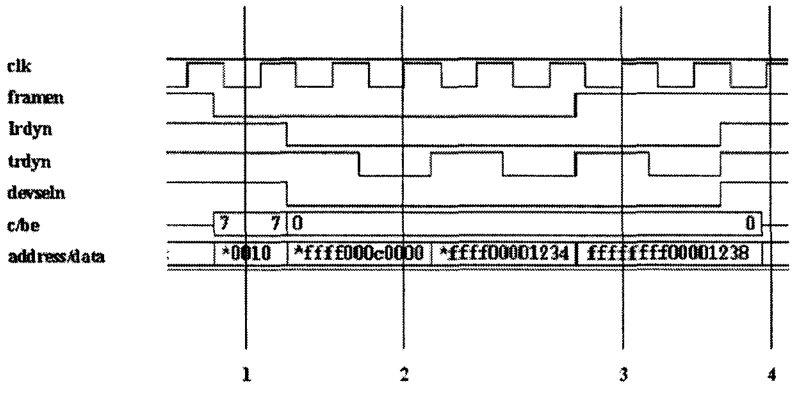

Figure 6-3. Sample PCI write transaction

Marker 1 shows the point where the master issues the write command (0111) on the "c/be" bus. On the same cycle, the address bus also carries the address for the target device. Marker 2 shows the point when the master writes a valid data. Marker 3 shows that the target device de-asserts the "trdy" signal indicating that it is not ready for the write transaction. In the same clock cycle, the master device de-asserts the signal "framen" indicating that this is the last data phase. Marker 4 shows that the signals "irdyn," "trdyn" and "devseln" are all de-asserted, indicating the completion of the write transaction.

6.2 A sample PCI System

A sample PCI system used for illustration purpose is shown in Figure 6-4. The figure shows that there are 2 PCI master devices and 2 PCI target devices. A user could be designing a device that is expected to act as a PCI master or a PCI target or both. One could use bus interface models for the other three devices in the sample system to verify the DUT. There are three specific scenarios for which SVA checkers could be written as part of the verification plan. These three scenarios are discussed in the upcoming sections.

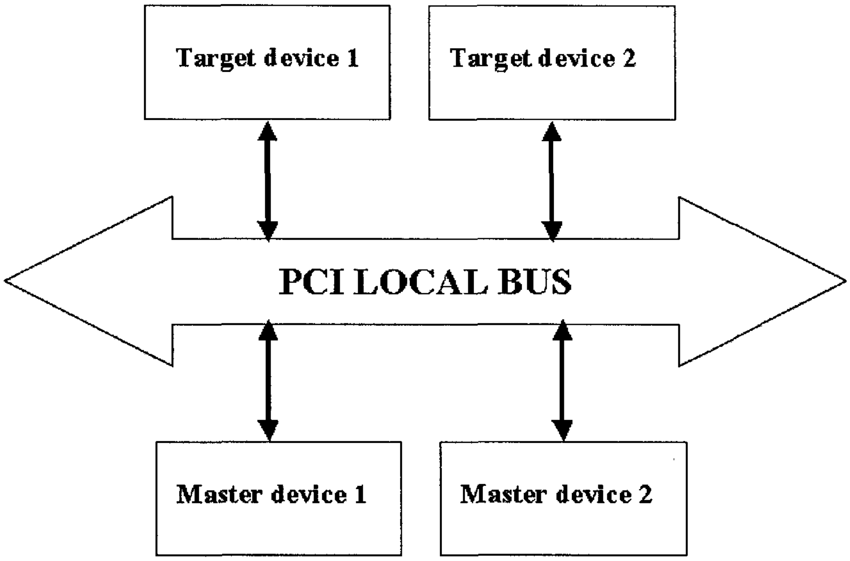

Figure 6-4. Sample PCI system

6.3 Scenario 1 – Master DUT Device

In this section, we assume that the design under test is a PCI master. Based on the PCI local bus specification, the PCI master has to follow certain protocol to be fully compliant. It is very common to write monitors as part of the verification environment. These monitors make sure that the DUT is not violating any of the protocol specifications.

The monitors can also produce detailed log files of all master transactions for post-processing purpose. SVA can be used to define a generic set of checkers that can be attached to any PCI master device. Since PCI is a standard protocol, the checkers developed should be written in such a way that it can be re-used with any PCI compliant master device. Figure 6-5 shows a sample configuration of the system.

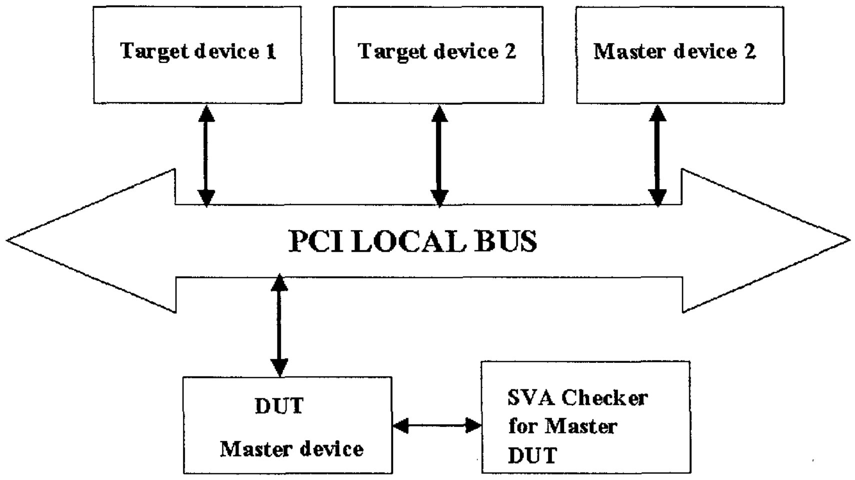

Figure 6-5. Sample configuration for PCI Master device as the DUT

6.3.1 PCI Master assertions

In this section, we show a few sample SVA checkers that can be written to verify the PCI master functionality. Some of the commonly used design conditions are defined as follows to enable re-use.

```
`define s_IO_READ
   ($fell (framen) && (cxben[3:0] == 4'b0010))
`define s_IO_WRITE
   ($fell (framen) && (cxben[3:0] == 4'b0011))
`define s_MEM_READ
   ($fell (framen) && (cxben[3:0] == 4'b0110))
`define s_MEM_WRITE
   ($fell (framen) && (cxben[3:0] == 4'b0111))
`define s_CONFIG_READ
   ($fell (framen) && (cxben[3:0] == 4'b1010))
`define s_CONFIG_WRITE
   ($fell (framen) && (cxben[3:0] == 4'b1011))
`define s_DUAL_ADDR_CYCLE
   ($fell (framen) && (cxben[3:0] == 4'b1101))
`define s_MEM_READ_LINE
   ($fell (framen) && (cxben[3:0] == 4'b1110))
`define s_MEM_WRITE_INV
   ($fell (framen) && (cxben[3:0] == 4'b1111))
```

```
`define s_BUS_IDLE
   (framen && irdyn)
```

Master_chk1: On a given clock cycle, "framen" cannot be de-asserted unless "irdyn" stays asserted on the same clock cycle.

```
property p_mchk1;
@(posedge clk)
   $rose (framen) |-> (irdyn == 0);
endproperty

a_mchk1: assert property(p_mchk1);
c_mchk1: cover property(p_mchk1);
```

The master device asserts the signal "framen" during the last data phase. Hence, the signal "irdyn" should stay asserted at this point. If not, this is a violation. Figure 6-6 shows a sample waveform of this check in a simulation. Markers 1, 2, 3, 4 and 5 show instances where there is a rising edge on the "framen" signal and in all those clock edges, the signal "irdyn" was always asserted. Hence, the checker succeeds.

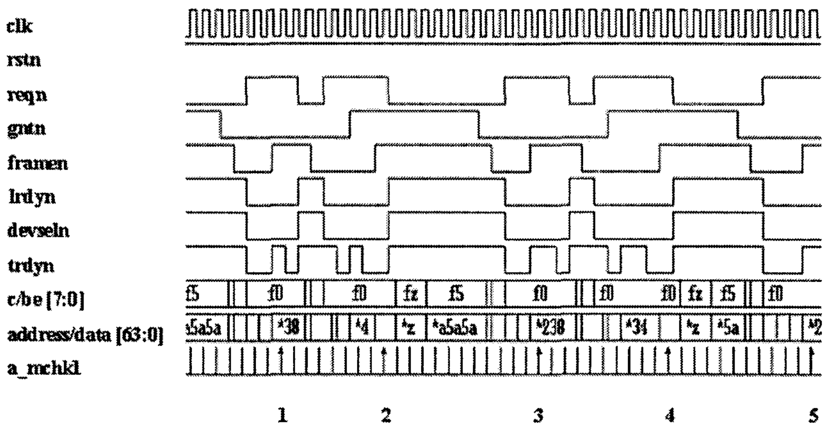

Figure 6-6. PCI Master check1

Master_chk2: Once "framen" is de-asserted, it cannot be asserted during the same transaction.

```
property p_mchk2;
@(posedge clk) $rose (framen) |->
     framen[*1:8] ##0 $rose (irdyn && trdyn);
endproperty

a_mchk2: assert property(p_mchk2);
c_mchk2: cover property(p_mchk2);
```

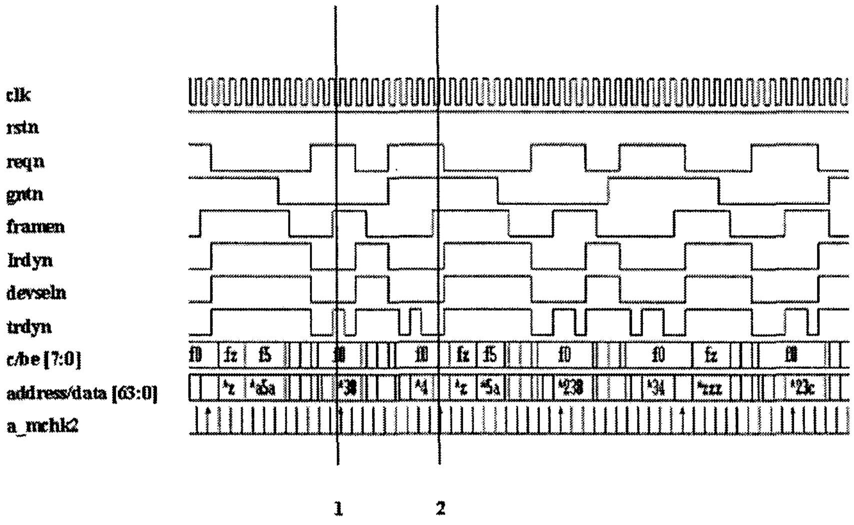

Figure 6-7. PCI Master check2

Once the signal "framen" is de-asserted, the master device has only one more data phase left. But it can take more than just one cycle to complete the last data phase. For example, if the target is not ready to accept the data, then the master waits to finish the last data phase. Before the master completes the last data phase, the frame cannot be asserted again. In other words, the signals "irdyn" and "trdyn" have to be de-asserted first before asserting "framen" again. Figure 6-7 shows a sample waveform of this check in a simulation.

Marker 1 shows a success of the assertion. At this point, there is a rising edge on the signal "frame" and hence the check becomes active. Note that in the same clock cycle, the signal "trdyn" is de-asserted indicating that the target is not ready to accept data. In the next clock cycle, both the signals "trdyn" and "irdyn" are asserted and hence the last data phase is complete.

One clock cycle, later the signals "irdyn" and "trdyn" are de-asserted. Marker 2 also shows a success but in this case, when the rising edge of the frame occurs, both the signals "irdyn" and "trdyn" are asserted and hence the last data phase is completed. In the next clock cycle, the signals "irdyn" and "trdyn" are de-asserted.

Master_chk3: Once "irdyn" is asserted, the master cannot change "irdyn" or "framen" until the current data phase begins.

```
property p_mchk3;
@(posedge clk)
$fell (irdyn) ##[0:5]
!(devseln) ##0 stopn    |->
      (!irdyn ) [*0:16] ##0 !trdyn;
endproperty

a_mchk3: assert property(p_mchk3);
c_mchk3: cover property(p_mchk3);
```

Once a master asserts the signal "irdyn," it is expected that a valid data phase begin within 16 clock cycles assuming there are no stop conditions issued by the target device. The data phase begins when the target device asserts the signal "trdyn." From the point when signal "irdyn" is asserted, assuming there are no stop conditions, the signal "irdyn" should be kept asserted until the signal "trdyn" is asserted by the target device.

Figure 6-8 shows a sample waveform of this check in a simulation. Marker 1 shows a success of the checker. The signal "irdy" and "devseln" are asserted at this point. One cycle later "trdyn" is asserted and hence the checker succeeds.

Marker 2 shows a condition wherein both signals "irdyn" and "devseln" are asserted and 2 clock cycles later, the signal "trdyn" is asserted. The signal "irdyn" stays asserted until the arrival of the "trdyn" signal and hence the checker succeeds.

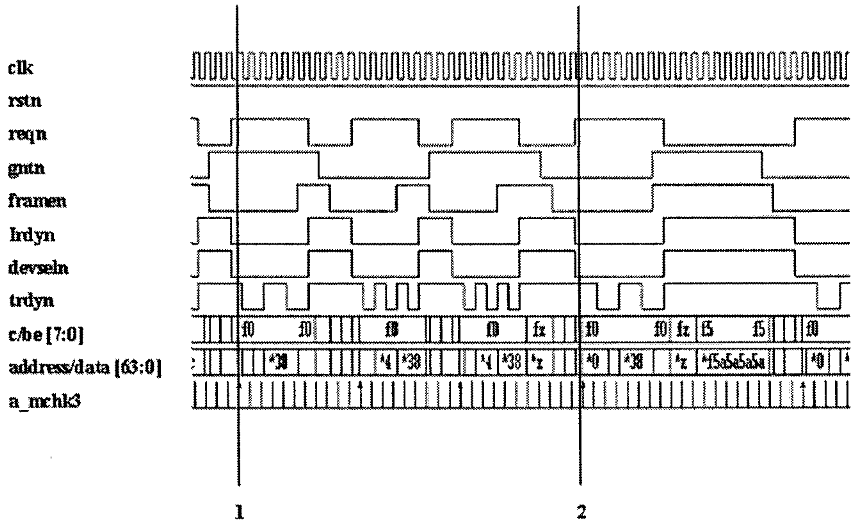

Figure 6-8. PCI Master check3

Master_chk4: The master is required to assert "irdyn" within 8 cycles from when the "framen" is asserted.

An **intersect** construct is used to control the length of the entire property. If the consequent of the property does not succeed within 1 to 8 clock cycles, the assertion will fail.

```
property p_mchk4;
@(posedge clk)
$fell (framen) |->
  1[*1:8] intersect
  ($fell (framen) ##[1:$] $fell(irdyn));
endproperty

a_mchk4: assert property(p_mchk4);
c_mchk4: cover property(p_mchk4);
```

Master_chk5: Normal Termination, once "framen" is de-asserted, the last data phase is completed within 8 clock cycles.

```
property p_mchk5;
@(posedge clk)
$rose (framen) |->
```

```
    (##[1:8] ($rose (irdyn && trdyn && devseln)));
endproperty

a_mchk5: assert property(p_mchk5);
c_mchk5: cover property(p_mchk5);
```

Master_chk6: Master Abort, "devseln" should be asserted within 5 cycles of "framen" being asserted. If "devseln" is not asserted within 5 cycles, then the "framen" should be de-asserted and one cycle later "irdyn" should be de-asserted.

```
sequence s_mchk6;
@(posedge clk)
  $fell (framen) ##1 (devseln)[*5] ##0 framen;
endsequence

property p_mchk6;
@(posedge clk)
  s_mchk6.ended |-> ##1 $rose (irdyn);
endproperty

a_mchk6: assert property(p_mchk6);
c_mchk6: cover property(p_mchk6);
```

Figure 6-9 shows a sample waveform of this check in a simulation. Marker 1 shows the clock edge in which the signal "framen" is detected as asserted. If the signal "devseln" does not arrive in the next 5 clock cycles, then the master should abort this transaction. Marker 2 shows the clock edge on which "devseln" failed to arrive and Marker 3 shows the next clock edge wherein the master device de-asserts the signal "irdyn." Since the property starts when the sequence s_mchk6 ends successfully, marker 2 is the point where the success is shown.

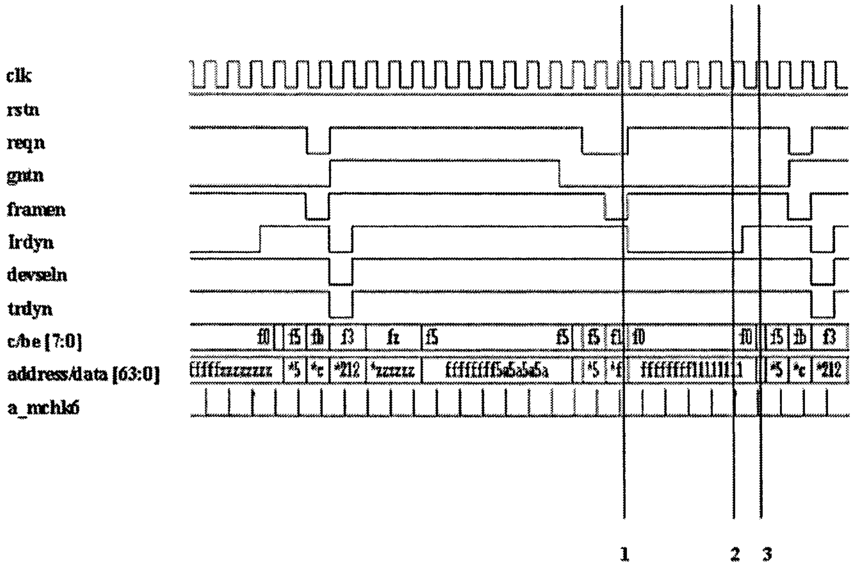

Figure 6-9. PCI Master check6

Master_chk7: When master is aborted by a target either by retry or disconnect, the master must de-assert its request before repeating the transaction. The request should be de-asserted on the clock cycle when the bus goes to the idle state and one clock cycle before or after the idle state.

```
sequence s_mchk7_before;
@(posedge clk)
   (!devseln && $fell (stopn) && trdyn)
   ##1 reqn ##1 `s_BUS_IDLE;
endsequence

sequence s_mchk7_after;
@(posedge clk)
   (!devseln && $fell (stopn) && trdyn)
   ##1 !reqn ##1 `s_BUS_IDLE;
endsequence

property p_mchk7_before;
@(posedge clk)
    s_mchk7_before.ended |->
                          reqn;
```

```
endproperty

property p_mchk7_after;
@(posedge clk)
   s_mchk7_after.ended |->
                        reqn [*2];
endproperty

a_mchk7_before: assert property(p_mchk7_before);
a_mchk7_after: assert property(p_mchk7_after);

c_mchk7_before: cover property(p_mchk7_before);
c_mchk7_after: cover property(p_mchk7_after);
```

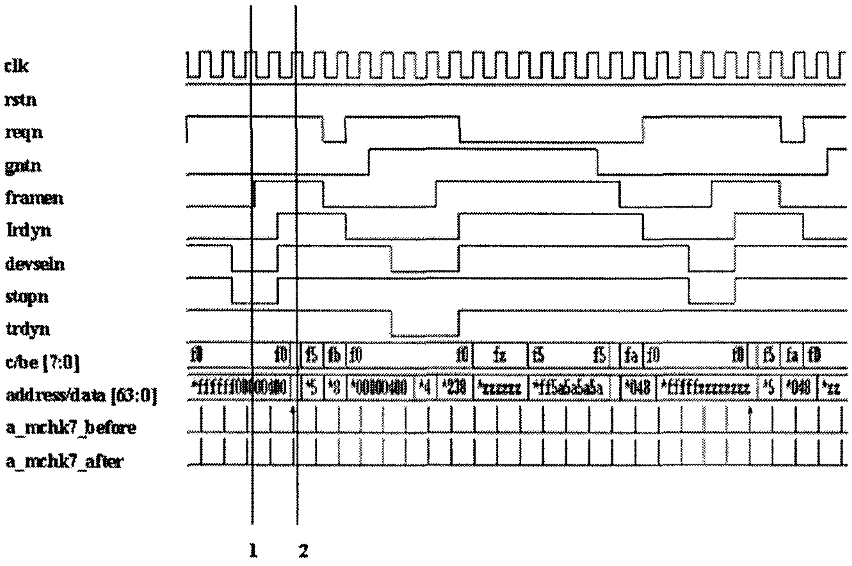

Figure 6-10. PCI Master check7

This check needs two separate properties. The main requirement is that, if the target device issues a stop condition, the master will have the "reqn" signal de-asserted before requesting the bus again. The master device could have de-asserted the signal "reqn" before the stop condition actually arrived. In this case, it is verified that the signal "reqn" is de-asserted during the clock cycle when the bus is idle and also that the signal "reqn" was de-asserted in the previous clock cycle (p_mchk7_before).

If the master device did not de-assert the signal "reqn" before the arrival of the stop condition, then it is verified that the signal "reqn" is de-asserted during the bus idle cycle and also the next clock cycle (p_mchk7_after). Note that the bus becomes idle 2 cycles after the target device asserts the signal "stopn." Figure 6-10 shows a sample waveform of this check in a simulation.

Marker 1 shows the clock edge when the target device asserts the "stopn" signal. Marker 2 shows the point when the bus becomes idle. Note that the signal "reqn" is de-asserted at this clock cycle and also in the previous clock cycle. Hence, the checker a_mchk7_before succeeds.

Master_chk8: When the target device terminates a transaction with a retry command, the master must repeat the same transaction until it is completed.

```
sequence s_mchk8a(temp1);
@(posedge clk)
(((!gntn || $rose (gntn))
&& $fell framen)),temp1=cxben[3:0])
##[1:2] $fell(irdyn) ##[0:5] $fell(stopn)
&& $fell (devseln) && trdyn;
endsequence

sequence s_mchk8b(temp2);
@(posedge clk)
$fell (reqn) ##[0:100] !gntn
##[0:5] $fell (framen)
##0 ((cxben[3:0] == temp2));
endsequence

property p_mchk8;
int temp;
@(posedge clk)
s_mchk8a(temp) |->
    ##[2:20] s_mchk8b(temp);
endproperty

a_mchk8: assert property(p_mchk8);
c_mchk8: cover property(p_mchk8);
```

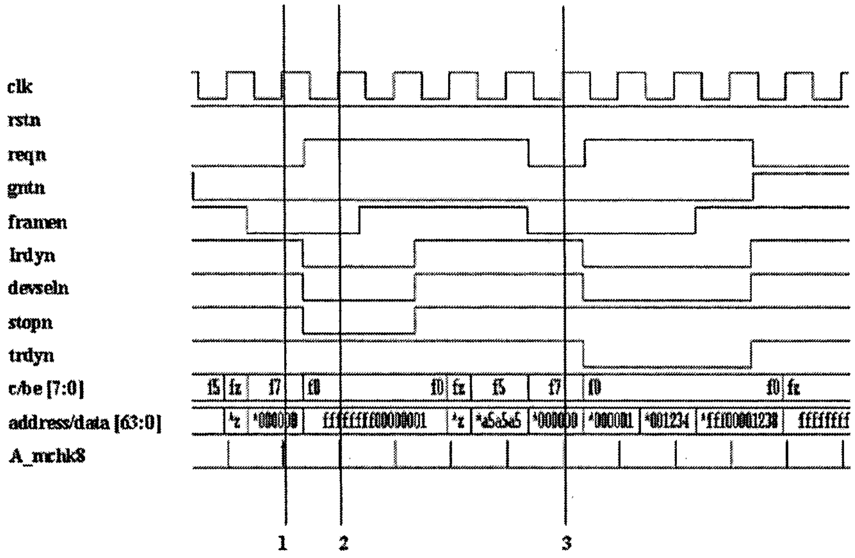

Figure 6-11. PCI Master check8

Two separate sequences are written to check this property. The first sequence starts at the point when the master device asserts the "framen" signal. When the master asserts the frame, it also issues the command. A temporary variable called "temp1" is used to store the command that was issued by the master. The variable is updated upon a successful match on a falling edge of the signal "framen." In the next few cycles, if the target device terminates the transaction by asserting the signal "stopn," then the sequence s_mchk8a will match. The property p_mchk8 has the sequence s_mchk8a as the antecedent. If the antecedent is true, then we wait for the next command to be issued by the master. If and when the master issues a new command, we compare the command value stored in the local variable "temp" to the actual command issued by the master on the bus, to verify that both the commands are the same. If the commands are not the same, it is a violation. Figure 6-11 shows a sample waveform of this check in a simulation.

Marker 1 shows the point when a falling edge of the signal "framen" is detected. At this point a command "f7" is placed on the command bus. Marker 2 shows the point when the target device terminated the transaction by asserting the signal "stopn." The master makes another request and gets the grant for the bus. Marker 3 shows the point when the master asserts the signal “framen” again. At this point, a command of “f7” is placed on the bus once again and hence the check succeeds.

Master_chk9: Bus parity check errors for address phase (SERR), this can be checked for all the different types of transactions like memory read, memory write, I/O read, I/O write, etc.

```
property p_mchk9;
@(posedge clk)
 $fell (framen)  ##1
 (par ^ $past (^(ad[31:0]^cxben[3:0])) == 1) |->
   ##[1:5] $fell (serrn);
endproperty

a_mchk9: assert property(p_mchk9);
c_mchk9: cover property(p_mchk9);
```

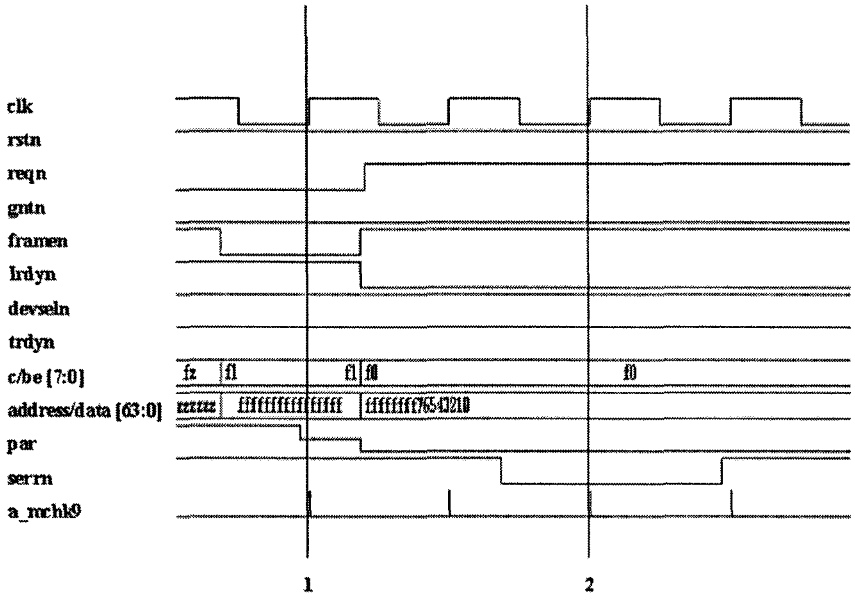

Figure 6-12. PCI Master check9

A parity check is performed during the address phase of every transaction. The parity should always be even for the signal “par” and the vectors “ad” and “c/be.” This can be achieved by XOR’ing these three signals. Usually, the parity error is issued on the next clock cycle. If a parity error occurs during the address phase of a transaction, it indicates a system error and the signal "serrn" should be asserted. Figure 6-12 shows a sample waveform of this check in a simulation.

Marker 1 shows the point when the address phase is sampled. The value of the bus "c/be," value of the bus "address" and the signal "par" are sampled at this point and XOR'ed to find if even parity exists. Marker 2 shows that, even parity does not exist and hence the signal "serrn" is asserted. Note that the signal "serrn" is kept asserted for 2 clock cycles.

Master_chk10: Parity error in data phase (PERR).

A parity check is performed during the data phase of every transaction. The parity should always be even for the signal "par" and the vectors "ad" and "c/be." This can be achieved by XOR'ing these three signals. Usually, the parity error is issued on the next clock cycle. If a parity error occurs during the data phase of a transaction, the signal "perrn" should be asserted. Figure 6-13 shows a sample waveform of this check in a simulation.

```
property p_mchk10;
@(posedge clk)
(!irdyn && !trdyn) ##1
(par ^ $past (^(ad[31:0]^cxben[3:0])) == 1) |->
                              ##[1:5] !perrn;
endproperty

a_mchk10: assert property(p_mchk10);
c_mchk10: cover property(p_mchk10);
```

Marker 1 shows the point when the first data phase occurs out of the multiple data phases of this particular transaction. In the next clock cycle, the required parity value is set. This value is XOR'ed along with the value of the data and the command byte enable sampled from the previous clock cycle. If the parity bit is set incorrectly, the signal "perrn" should be asserted. For marker 1, the XOR'ed value of the data and command is 1 and the par bit is set to 1. Hence, there is no parity error. Marker 2 shows the second data phase. The XOR'ed value of the data (1234) and the command (0000) is 1 and the parity bit is set to 0. This does not provide even parity and hence the signal "perrn" is asserted in the next clock cycle, as shown by marker 3.

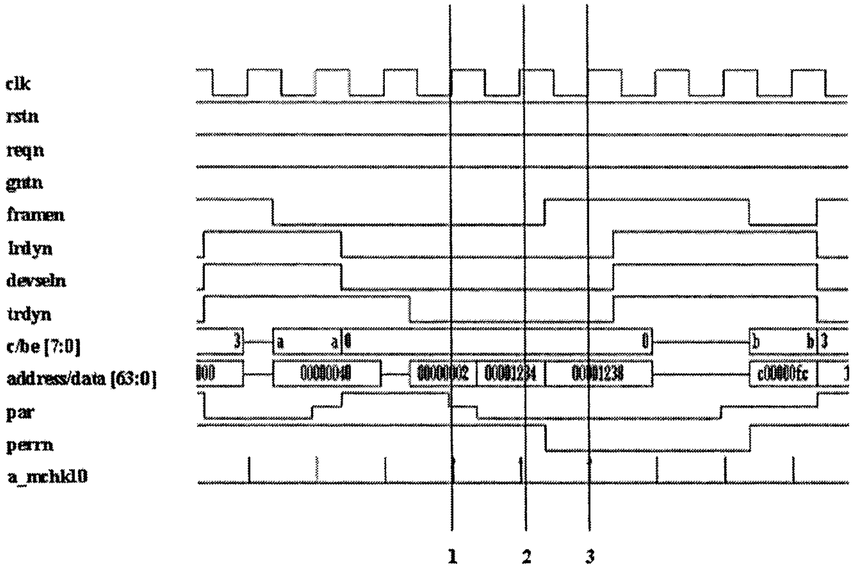

Figure 6-13. PCI Master check10

Master_chk11: PERR should not be asserted for special cycles.

The master can issue a command for special cycle. The command bus carries the value of "0001" during a special cycle. The parity error cannot be asserted during a special cycle irrespective of what data is driven into the data bus.

```
property p_mchk11;
@(posedge clk)
($fell (framen)&&(cxben[3:0] == (4'b0001))) |->
     (perrn [*1:$]
     ##0 ($rose (irdyn && trdyn))
     ##1 perrn[*2]);
endproperty

a_mchk11: assert property(p_mchk11);
c_mchk11: cover property(p_mchk11);
```

If the master asserts the signal "framen" during a special cycle, the signal "perrn" cannot be asserted until the bus becomes idle. Note that, it is necessary to make sure that the signal "perrn" is not asserted for 2 cycles even after the signals “trdyn” and “irdyn” are de-asserted. Since the parity error is issued in the next clock cycle of a data phase and the parity error is normally asserted for 2 clock cycles, this extension is necessary for the checker. Figure 6-14 shows a sample waveform of this check in a simulation. Marker 1 shows that the master has asserted the signal “framen” and issued the command “0001” on the c/be bus indicating that it is a special cycle. Note that the signal “perrn,” which indicates a parity error, remains de-asserted irrespective of what the “par” bit value is. Hence, the checker succeeds. Marker 2 shows a similar special cycle.

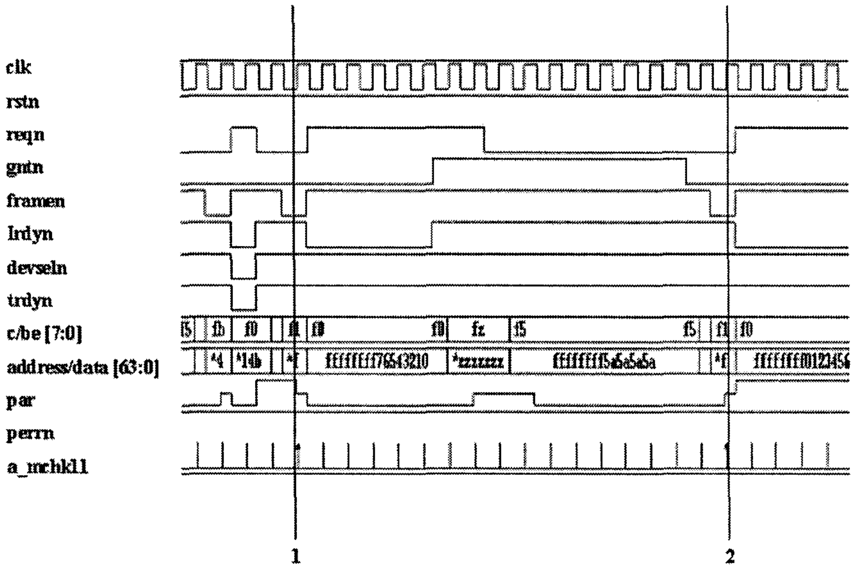

Figure 6-14. PCI Master check11

Master_Chk12: Dual Address Cycle.

It takes two cycles for the master to assert “irdyn” if it addresses a 64-bit target device. When the master asserts the signal “framen,” it also issues the command for the dual address cycle. Along with the command, it also asserts the signal “req64n” to let the target know that the master wishes to perform a 64-bit transaction.

```
property p_mchk12;
@(posedge clk)
`s_DUAL_ADDR_CYCLE && req64n |=>
```

```
        not  $fell (irdyn);
endproperty

a_mchk12: assert property(p_mchk12);
c_mchk12: cover property(p_mchk12);
```

Master_chk13: Full 64-bit Transactions.

The master device asserts the signal "req64n" along with the "framen" signal to let the target device know that it wants to perform a 64-bit transaction. The target device responds by asserting the signals "devseln" and "ack64n" within 1 to 5 clock cycles.

```
property p_mchk13;
@(posedge clk)
$fell (gntn) ##[1:8]
$fell (framen) && $fell(req64n) |->
     ##[1:5] $fell (ack64n) && $fell(devseln);
endproperty

a_mchk13: assert property(p_mchk13);
c_mchk13: cover property(p_mchk13);
```

Figure 6-15 shows a sample waveform of this check in a simulation. Marker 1 shows the point when the signal "gntn" is asserted. In the next clock cycle, the master asserts the "framen" signal and the "req64n" signal, as shown by marker 2. This alerts the target device that the master wants to perform a 64-bit transaction. The target acknowledges the request of the master by asserting the signal "ack64n" along with the signal "devseln" in the next clock cycle. The master asserts the signal "irdyn" in the next clock cycle. Note that the master takes 2 clock cycles to assert the signal "irdyn" after asserting the "framen" signal in a 64-bit transaction.

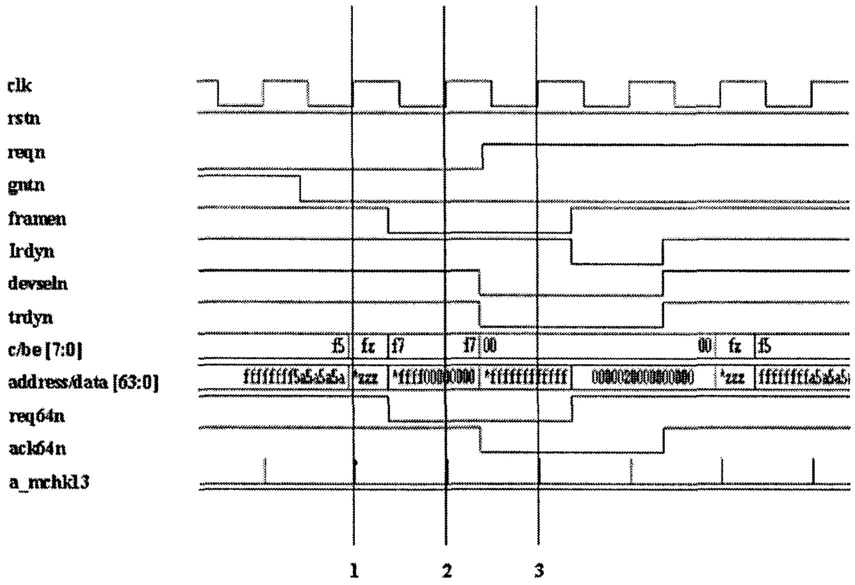

Figure 6-15. PCI Master check13

Master_Chk14: Check par64 signal validity.

Similar to the 32-bit transactions, a parity bit is used for 64-bit transactions. An even parity is maintained on the XOR'ed value of the most significant 32 bits of the data bus and the 4 most significant bits of the command byte enable, using the signal "par64."

```
property p_mchk14;
@(posedge clk)
(!ack64n && !irdyn && !trdyn && !devseln) &&
(^(ad[63:32]^cxben[7:4]) == 1) |=>
                                  par64;
endproperty

a_mchk14: assert property(p_mchk14);
c_mchk14: cover property(p_mchk14);
```

Figure 6-16 shows a sample waveform of this check in a simulation. Marker 1 shows the point when a valid 64-bit data phase occurs. The XOR value of the 32 MSB of data (00000200) and the 4 MSB of c/be (0000) is 1. Hence, to maintain the even parity, the value of the signal "par64" should be a 1 in the next clock cycle. As seen in marker 2, the value of "par64" is detected as 1 and hence the check succeeds.

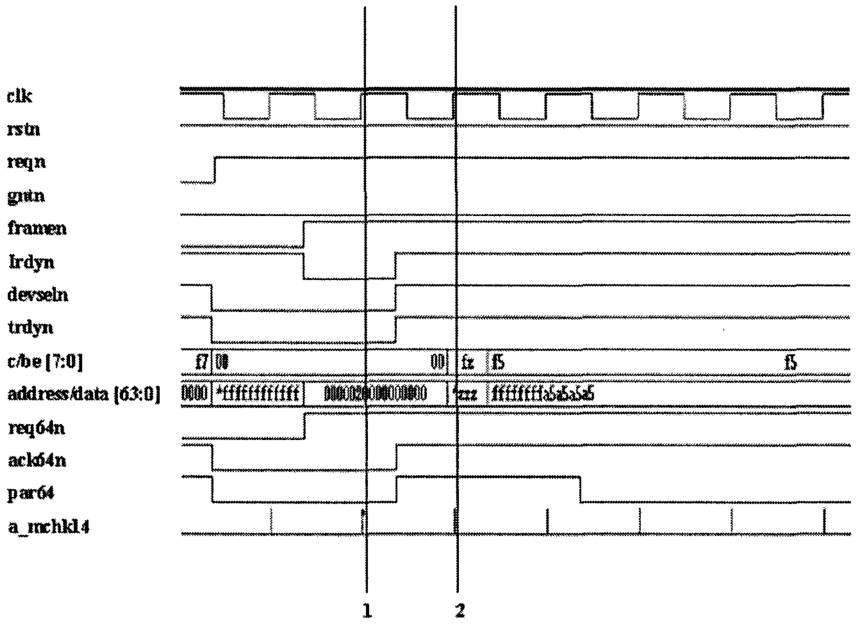

Figure 6-16. PCI Master check14

Master_chk15: Bus Parking

Bus parking happens when the "reqn" signal of a master is de-asserted but it still has the grant for the bus. The master goes to the idle state and then drives a stable value into the data bus and the command bus to indicate that it has parked the bus. When the master goes to the idle state, "reqn" should not be asserted. If "reqn" is asserted, it is considered as a back to back transaction.

```
sequence s_mchk15;
@(posedge clk)
first_match($fell (framen) ##[1:$]
     (framen && irdyn && !gntn && reqn));
endsequence

property p_mchk15;
```

```
@(posedge clk)
  s_mchk15 |->
  ##[1:8] (($stable(ad[31:0]))
 && ($stable (cxben[3:0])))
  ##1 (par ^ $past (^(ad[31:0]^cxben[3:0])) == 0);
endproperty

a_mchk15: assert property(p_mchk15);
c_mchk15: cover property(p_mchk15);
```

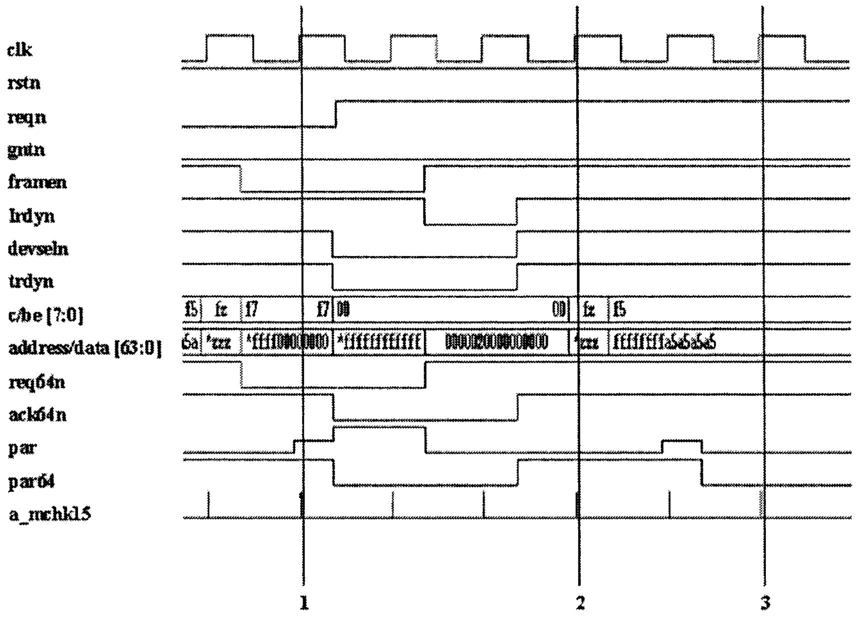

Figure 6-17. PCI Master check15

A simple sequence s_mchk15 is written to identify a valid completion of a master transaction. At the completion of the transaction, if the master still has the grant, then it is expected that it will drive a stable value into the data bus and the command bus, hence parking the PCI bus. The **$stable** function is used to detect if the bus values have stabilized. It is expected that one cycle later, the correct parity bit is set for the stable values. The XOR technique is used once again to detect the validity of the parity bit.

Figure 6-17 shows a sample waveform of this check in a simulation. Marker 1 shows a valid start of a transaction and marker 2 shows the completion of the transaction. Note that at this point, the signal "reqn" is de-asserted but the signal "gntn" is still asserted. Hence, the master is expected to drive a stable value into the data bus and command bus within 1 to 8 cycles. Marker 3 shows the point when the master has parked the bus.

Master_chk16: Fast back to back transactions.

A master device can perform fast back to back transactions wherein the signal "framen" is asserted in the immediate next clock cycle after the completion of a transaction. This can happen both at the completion of a single data phase or a multiple data phase sequence. Property p_mchk16 will capture only the single data phase transactions whereas property p_mchk17 will capture both single data phase and multiple data phase transactions.

```
property p_mchk16;
@(posedge clk)
($rose (framen) && $fell (irdyn))
##1 $fell (framen) |->
              $rose (irdyn);
endproperty

a_mchk16: assert property(p_mchk16);
c_mchk16: cover property(p_mchk16);


property p_mchk17;
@(posedge clk)
(!irdyn && framen)
##1 $fell (framen) |->
               $rose (irdyn);
endproperty

a_mchk17: assert property(p_mchk17);
c_mchk17: cover property(p_mchk17);
```

Note that the main difference between the two properties is the sampling mechanism. If it is a single data phase back to back transaction, the signals "framen" and "irdyn" are sampled for their edges (falling edge of "framen" and rising edge of "irdyn"). Figure 6-18 shows a sample waveform of this check in a simulation.

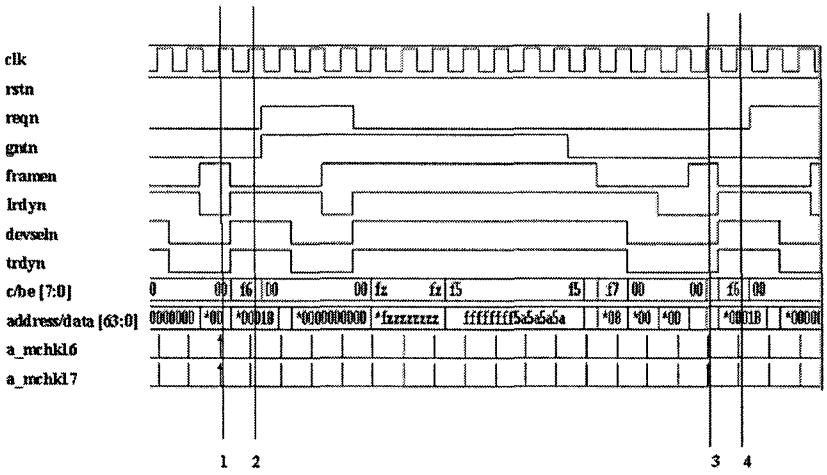

Figure 6-18. PCI Master check 16/17

Markers 1 and 2 indicate a single data phase back to back transaction. Markers 3 and 4 indicate a multiple data phase back to back transaction. Note that property p_mchk17 is sufficient to capture both scenarios.

Sample functional coverage point for PCI Master:

Master Abort - The master abort can happen in any of the following conditions - I/O read, I/O write, Configuration read, Configuration write, Memory read, Memory write. A cover statement can be written to make sure that the testbench executed all of these possible abort conditions at least once. After the master asserts the "irdyn" signal, if a target device does not respond within 5 clock cycles by asserting the "devseln" signal, the master will abort the transaction by de-asserting the "irdyn" signal. Note that the commands are specified in the properties by using the `define code.

```
property p_mcov1;
@(posedge clk)
`s_IO_READ ##1 (devseln)[*5] |=>
                        $rose (irdyn);
endproperty

property p_mcov2;
@(posedge clk)
`s_IO_WRITE ##1 (devseln)[*5] |=>
```

```
                         $rose (irdyn);
endproperty

property p_mcov3;
@(posedge clk)
`s_MEM_READ ##1 (devseln)[*5] |=>
                         $rose (irdyn);
endproperty

property p_mcov4;
@(posedge clk)
`s_MEM_WRITE ##1 (devseln)[*5] |=>
                         $rose (irdyn);
endproperty

property p_mcov5;
@(posedge clk)
`s_CONFIG_READ ##1 (devseln)[*5] |=>
                         $rose (irdyn);
endproperty


property p_mcov6;
@(posedge clk)
`s_CONFIG_WRITE  ##1 (devseln)[*5] |=>
                       $rose (irdyn);
endproperty

c_mcov1: cover property(p_mcov1);
c_mcov2: cover property(p_mcov2);
c_mcov3: cover property(p_mcov3);
c_mcov4: cover property(p_mcov4);
c_mcov5: cover property(p_mcov5);
c_mcov6: cover property(p_mcov6);
```

6.4 Scenario 2 – Target DUT Device

In this section, we assume that the design under test is a PCI target device. The rest of the system remains exactly the same. Figure 6-19 shows the sample system.

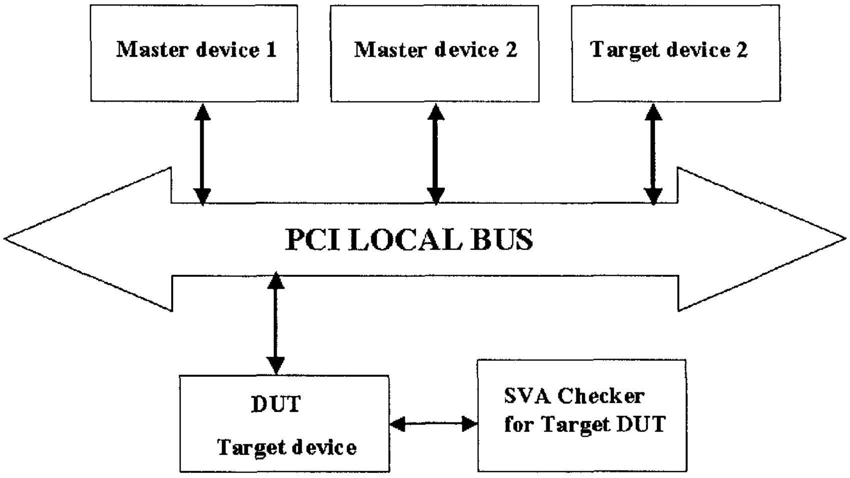

Figure 6-19. Sample Configuration for PCI Target device as DUT

6.4.1 PCI Target assertions

Target_chk1: Once target asserts the signal "stopn," it should keep "stopn" asserted until the signal "framen" is de-asserted, one clock cycle later "stopn" is de-asserted.

```
property p_tchk1;
@(posedge clk)
($fell (stopn) && !framen) |->
!stopn [*1:$]
##0 $rose (framen) ##1 $rose(stopn);
endproperty

a_tchk1: assert property(p_tchk1);
c_tchk1: cover property(p_tchk1);
```

Note that the property uses the "repeat until" construct to make sure that the signal "stopn" is kept asserted until the signal "framen" is de-asserted. Figure 6-20 shows a sample waveform of this check in a simulation.

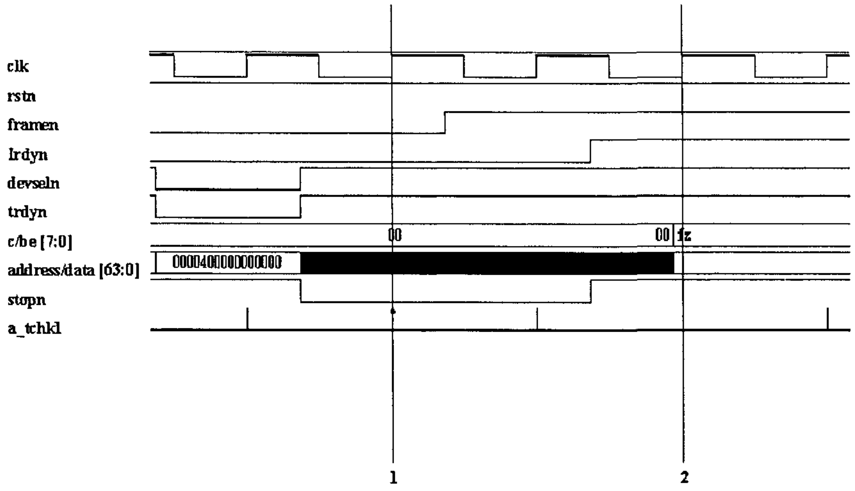

Figure 6-20. PCI Target check1

Marker 1 shows the point when the signal "stopn" is asserted. In the next clock cycle, the signal "framen" is de-asserted and one cycle later, the signal "stopn" is also de-asserted, as shown by marker 2.

Target_chk2: Once target has asserted the signal "trdyn," it cannot change "devseln" and "trdyn" until the current data phase completes.

When the target device asserts the signal "trdyn," it has acknowledged that it is ready to either accept data or send data. Hence, it cannot de-assert the signal "trdyn" without completing a data phase.

```
property p_tchk2;
@(posedge clk)
$fell (trdyn) |->
   (!trdyn && !devseln) [*0:16] ##0 !irdyn;
endproperty

a_tchk2: assert property(p_tchk2);
c_tchk2: cover property(p_tchk2);
```

The property p_tchk2 becomes active on the falling edge of the signal "trdyn." The consequent of the property makes sure that the signals "trdyn" and "devseln" stay asserted until the signal "irdyn" is asserted. The latency on the "trdyn" signal is 16 cycles.

Target_chk3: The target device cannot assert the signal "trdyn" until "devseln" is asserted.

```
property p_tchk3;
@(posedge clk)
     $fell (trdyn) |->!devseln;
endproperty

a_tchk3: assert property(p_tchk3);
c_tchk3: cover property(p_tchk3);
```

Target_chk5: Disconnect with data.

The target device indicates that it cannot continue a transaction by asserting the "stopn" signal and the "trdyn" signal at the same time. When this happens, the target is required to de-assert the "trdyn" signal in the next clock cycle but keep the signal "stopn" asserted. Hence, the last data phase completes without transferring any data since the signal "trdyn" is de-asserted. This is classified as "Disconnect – B" by the PCI local bus specification.

```
property p_tchk5b;
@(posedge clk)
($fell (stopn) && !framen && !trdyn && !irdyn)
  |=>
    (framen && trdyn)
    ##1 (stopn && devseln && irdyn);
endproperty

a_tchk5b: assert property(p_tchk5b);
c_tchk5b: cover property(p_tchk5b);
```

Figure 6-21 shows a sample waveform of this check in a simulation. Marker 1 shows the point when the signal "stopn" is asserted. In the next clock cycle, signal "trdyn" is de-asserted as expected. Marker 2 shows that, one clock cycle later, the signal "stopn," "devseln" and "irdyn" are all de-asserted hence completing the transaction.

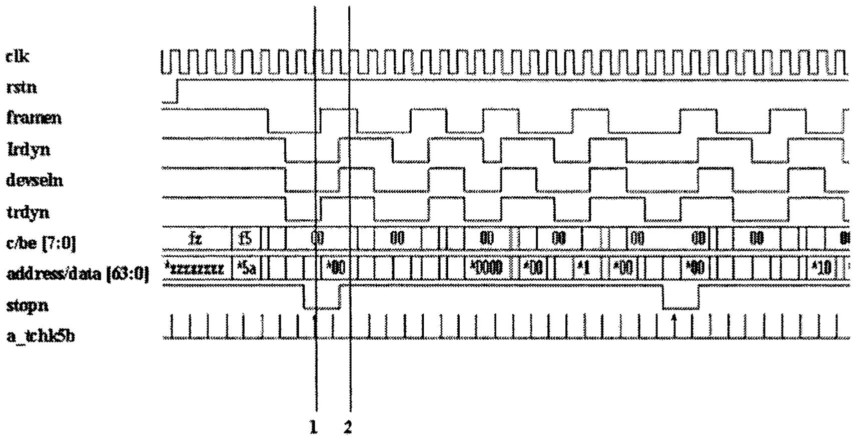

Figure 6-21. PCI Target check 5b

Target_chk6: Disconnect without data termination.

When the target device cannot complete any more data phases, it asserts the signal "stopn" and de-asserts the signal "trdyn." The target keeps the signal "stopn" asserted, until the final data phase is complete.

Note that a complex property like this should be split into smaller sequences as follows.

```
sequence s_tchk6a;
@(posedge clk)
(!irdyn && !trdyn && !devseln && !framen);
endsequence

sequence s_tchk6b;
@(posedge clk)
($fell (stopn) && $rose (trdyn) && !framen);
endsequence

sequence s_tchk6c;
@(posedge clk)
$rose (framen) ##[0:8] (!irdyn && !stopn);
endsequence

sequence s_tchk6;
```

```
@(posedge clk)
s_tchk6a.ended ##[1:8] s_tchk6b;
endsequence

property p_tchk6;
@(posedge clk)
 s_tchk6.ended |=> s_tchk6c;
endproperty

a_tchk6: assert property(p_tchk6);
c_tchk6: cover property(p_tchk6);
```

The sequence s_tchk6a identifies a valid data phase. The sequence s_tchkb identifies the point when the target device asserts the "stopn" signal. The sequence s_tchk6 is a concatenation of the two sequences s_tchk6a and s_tchk6b. The sequence s_tchk6 takes the check to the point, wherein a "stopn" has been issued. The sequence s_tchk6c looks for the point when both the signals "irdyn" and "stopn" are asserted. This is required because the master can get into a wait state before completing the last data phase, which therefore could have de-asserted the signal "irdyn."

Target_chk6_1: Master naturally terminating and target issuing an abort at the same time.

This is a case when the target is asserting the "stopn" signal to stop the transaction and at the same time the master device is also aborting the transaction naturally. This means that on the same clock cycle that the target asserts the "stopn" signal; the master de-asserts the "framen" signal.

```
sequence s_tchk6_1;
@(posedge clk)
(!irdyn && !trdyn && !devseln && !framen)
##[1:8] ($fell (stopn) && trdyn && framen);
endsequence

property p_tchk6_1;
@(posedge clk)
 s_tchk6_1.ended |=> (irdyn && stopn);
endproperty

a_tchk6_1: assert property(p_tchk6_1);
c_tchk6_1: cover property(p_tchk6_1);
```

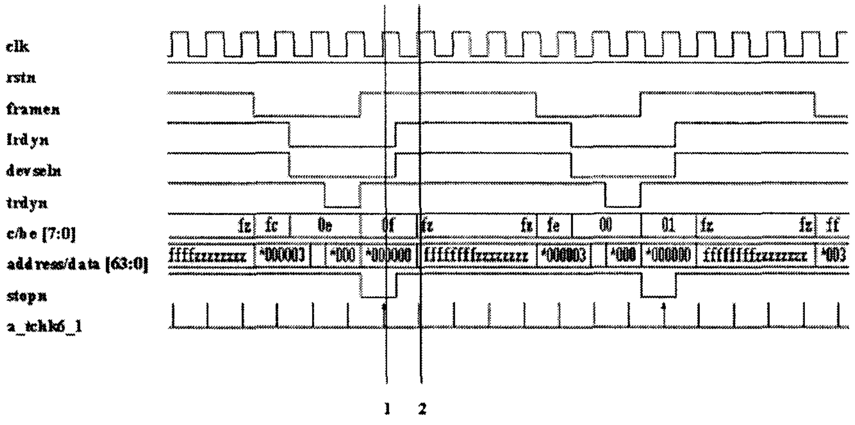

Figure 6-22. PCI Target check6_1

Note that a sequence is written to identify the point wherein both target and master are trying to stop the transaction simultaneously. The property checks that, if the antecedent matches, then the signal "irdyn" should be asserted in the next clock cycle along with the signal "stopn." Figure 6-22 shows a sample waveform of this check in a simulation. Marker 1 shows the point when signal "framen" is de-asserted and the signal "stopn" is asserted. Marker 2 shows the point when the signals "irdyn" and "stopn" are de-asserted.

Target_chk7: Retry

If the target is not ready for a transaction, it has to ask the master to retry the transaction at a later point. This has to be done before the occurrence of the first data phase. The target device will assert the signal "stopn" before asserting the "trdyn" signal for the first time.

```
sequence s_tchk7a;
@(posedge clk)
$fell (framen) ##[1:8] $fell(irdyn);
endsequence

sequence s_tchk7b;
@(posedge clk)
$fell (framen) ##[1:5]
$fell(devseln) && $fell(stopn) && trdyn;
endsequence
```

```
sequence s_tchk7;
@(posedge clk)
  first_match(s_tchk7a and s_tchk7b);
endsequence

property p_tchk7;
  @(posedge clk) s_tchk7.ended |=> framen;
endproperty

a_tchk7: assert property(p_tchk7);
c_tchk7: cover property(p_tchk7);
```

The sequence s_tchk7 becomes active when the signal "framen" is asserted. Once the signal "framen" is asserted, it is expected that the target device identify itself by asserting the signal "devseln." It can happen anywhere between 1 to 5 clock cycles depending on the speed of the target device. If the target device wants to issue a retry, it will assert the signal "stopn" along with the signal "devseln." At this point, the signal "trdyn" should stay de-asserted. One cycle after the retry is issued, the master de-asserts the "framen" signal to acknowledge the retry issued by the target device.

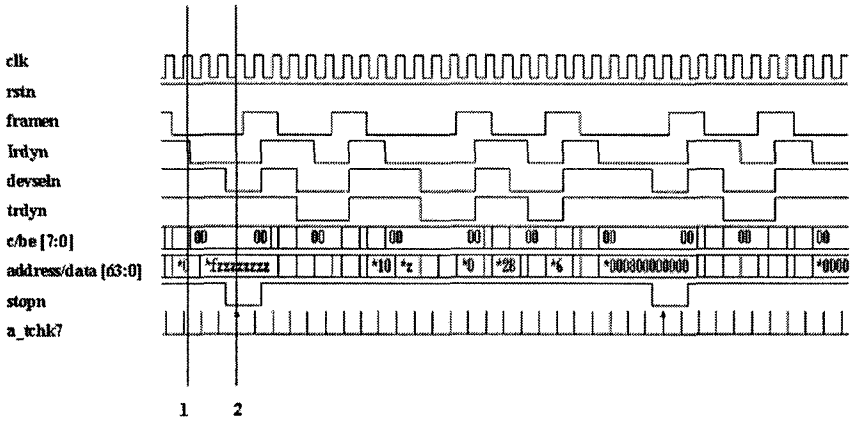

Figure 6-23. PCI Target check7

Figure 6-23 shows a sample waveform of this check in a simulation. Marker 1 shows the point when the master asserts the "framen" signal. Marker 2 shows the point when the target device asks the master to retry.

One cycle after marker 2, the master de-asserts the "framen" signal and ends the transaction.

Target_chk8: The signal "devseln" should not be asserted for a special cycle.

```
property p_tchk8;
@(posedge clk)
$fell (framen) && (cxben[3:0] == 4'b0001) |->
     devseln [*1:$] ##0 $rose (framen);
endproperty

a_tchk8: assert property(p_tchk8);
c_tchk8: cover property(p_tchk8);
```

During a special cycle, the signal "devseln" should not be asserted. The property becomes active when the "framen" signal is asserted and the master device places a special cycle command on the command bus. The consequent of the property makes sure that the signal "devseln" stays de-asserted until the master completes the transaction (by de-asserting the "framen" signal).

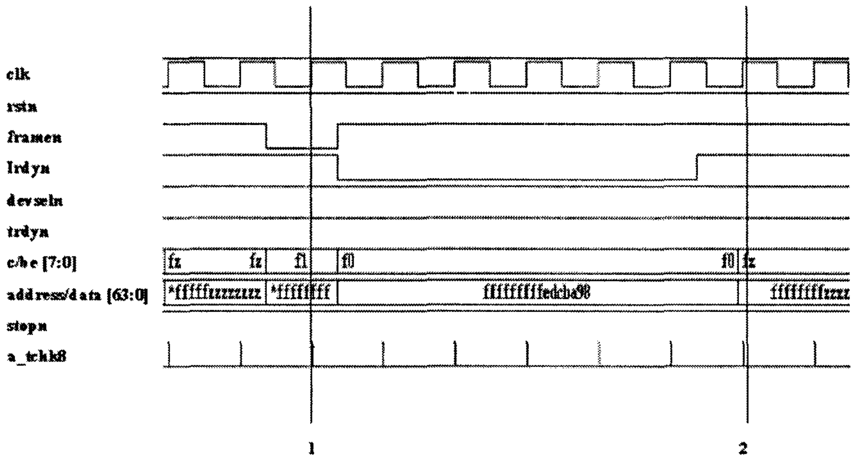

Figure 6-24. PCI Target check 8

Figure 6-24 shows a sample waveform of this check in a simulation. Marker 1 shows the point when a special cycle command is detected. Marker

2 shows the completion of the transaction. Note that from marker 1 to marker 2, the signal "devseln" stays de-asserted.

Target_chk9: Target latency for the completion of the first data phase is 16 cycles from the assertion of the signal "framen."

Once the master asserts the signal "framen," the target device identifies itself by asserting the signal "devseln" first. Depending on the nature of the target device, it can take anywhere from 1 to 5 cycles for the "devseln" signal to be asserted. For example, a fast target device takes only one clock cycle to respond, a medium target device takes 2 clock cycles to respond. After asserting the "devseln" signal, the target device will assert the "trdyn" signal if it is ready for a transaction. The total latency allowed by the PCI local bus specification, from the point the "framen" signal is asserted by the master to the point when an actual data phase happens (both "trdyn" and "irdyn" are asserted) is 16 clock cycles. Depending on the nature of the device the latency split can be summarized as shown in Table 6-2.

Two basic sequences are written to identify a valid data phase or a retry condition. A separate sequence is defined for each type of the target device. Note that the timing delay for the assertion of the "devseln" signal is the only difference between these sequences.

Table 6-2. Target latency table

Device type	Frame -> devsel	Devsel -> (irdy && trdy)
FAST	1	0:15
MEDIUM	2	0:14
SLOW	3	0:13
SUBTRACTIVE	4	0:12

```
sequence s_tchk9a;
@(posedge clk)
(!irdyn && !trdyn);
endsequence

sequence s_tchk9b;
@(posedge clk)
(!irdyn && !stopn);
endsequence

sequence s_tchk9_fast;
```

```
@(posedge clk)
$fell (framen) ##1 $fell(devseln);
endsequence
sequence s_tchk9_medium;
@(posedge clk)
$fell (framen) ##2 $fell(devseln);
endsequence

sequence s_tchk9_slow;
@(posedge clk)
$fell (framen) ##3 $fell(devseln);
endsequence

sequence s_tchk9_subtractive;
@(posedge clk)
$fell (framen) ##4 $fell(devseln);
endsequence

property p_tchk9_fast;
@(posedge clk)
s_tchk9_fast |-> ##[0:15]
   (!devseln) throughout
   (s_tchk9a.ended || s_tchk9b.ended);
endproperty

a_tchk9_fast: assert property(p_tchk9_fast);
c_tchk9_fast: cover property(p_tchk9_fast);

property p_tchk9_medium;
@(posedge clk)
s_tchk9_medium |-> ##[0:14]
   (!devseln) throughout
     (s_tchk9a.ended || s_tchk9b.ended);
endproperty

a_tchk9_medium: assert property(p_tchk9_medium);
c_tchk9_medium: cover property(p_tchk9_medium);

property p_tchk9_slow;
@(posedge clk)
s_tchk9_slow |-> ##[0:13]
   (!devseln) throughout
     (s_tchk9a.ended || s_tchk9b.ended);
```

```
endproperty

a_tchk9_slow: assert property(p_tchk9_slow);
c_tchk9_slow: cover property(p_tchk9_slow);

property p_tchk9_subtractive;
@(posedge clk)
s_tchk9_subtractive |-> ##[0:12]
   (!devseln) throughout
     (s_tchk9a.ended || s_tchk9b.ended);
endproperty

a_tchk9_subtractive:
     assert property(p_tchk9_subtractive);
c_tchk9_subtractive:
     cover property(p_tchk9_subtractive);
```

A separate property is written for each type of device. If the sequence mentioned in the antecedent of the property identifies a specific type of device, the consequent is allowed to take certain number of cycles to match, as specified in Table 6-2. For example, if it is a slow device, then the target device can take anywhere between 0 and 13 clock cycles to complete a valid data phase or issue a retry.

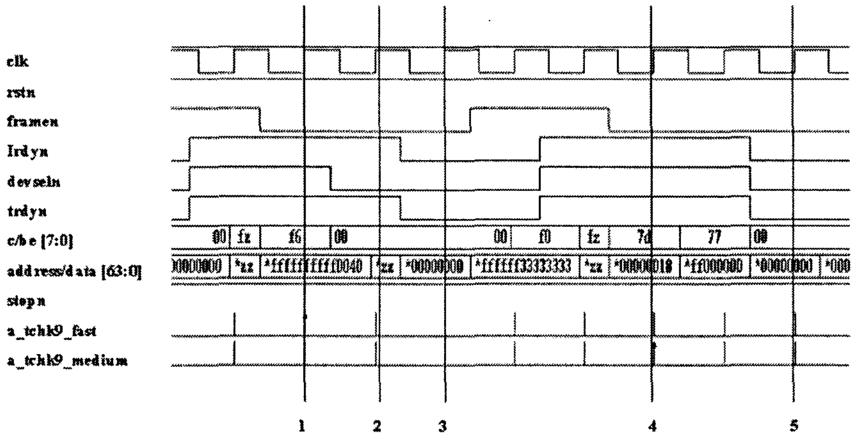

Figure 6-25. PCI Target check9

Figure 6-25 shows a sample waveform of this check in a simulation. Marker 1 shows the beginning of the property p_tchk9_fast. The master asserts the signal "framen" at this point. One cycle later, the signal "devseln" is asserted by the target device as shown by marker 2. One cycle after that, the signals "trdyn" and "irdyn" are asserted as shown by marker 3 and hence the check a_tchk9_fast succeeds.

Marker 4 shows the beginning of the property p_tchk9_medium. The master asserts the signal "framen" at this point. Two cycles later, the signal "devseln" is asserted by the target device as shown by marker 5. Note that the signals "trdyn" and "irdyn" are asserted on the same clock cycle and hence the check a_tchk9_medium succeeds.

Target_chk10: Latency for the subsequent data phase is 8 cycles from the previous data phase.

Both the master and target can issue wait states in between a transaction if they are not ready. If in a given clock edge, a data phase has just completed in a burst transaction, then the next data phase should occur within 8 clock cycles.

```
property p_tchk10;
@(posedge clk)
(!irdyn && !trdyn && !devseln && !framen) |->
    ##[1:8] (!irdyn && (!trdyn || !stopn));
endproperty

a_tchk10: assert property(p_tchk10);
c_tchk10: cover property(p_tchk10);
```

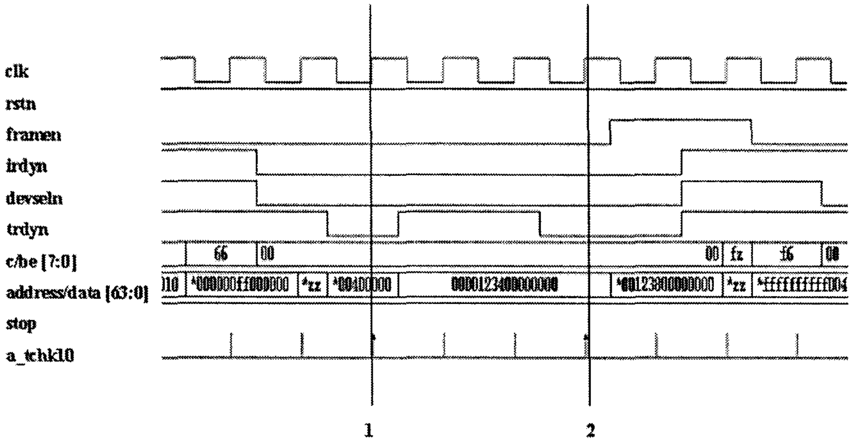

Figure 6-26. PCI Target check10

Figure 6-26 shows a sample waveform of this check in a simulation. Marker 1 shows a valid data phase. In the next clock cycle, the target device de-asserts the signal "trdyn" and hence issues a wait state. The wait state extends for one more cycle. One clock cycle after that, the signal "trdyn" is asserted again and hence a valid data phase occurs as shown by marker 2. In this case, the latency of the subsequent data phase is only 3 clock cycles and hence the check succeeds.

Target_chk11: The first data phase on a read command requires a turnaround cycle enforced by the signal "trdyn."

There are 4 possible read commands as shown in Table 6-1 and all read commands have a value of "10" in the 2 least significant bits. Whenever there is a read command, the master has to allow the target to drive the data into the bus and hence there is a turnaround cycle. The value of the data bus one clock cycle before the first data phase of a read cycle should be unknown.

```
sequence s_tchk11a;
@(posedge clk)
  ($fell (framen) && (cxben[1:0] == 2'b10));
endsequence

sequence s_tchk11b;
@(posedge clk)
```

```
first_match($fell (devseln) ##[1:16]
          $fell (trdyn));
endsequence

sequence s_tchk11;
@(posedge clk)
 s_tchk11a.ended ##[1:5] s_tchk11b;
endsequence

property p_tchk11;
@(posedge clk)
s_tchk11.ended |->
($isunknown (par)
&& $past ($isunknown(ad[31:0])));
endproperty

a_tchk11: assert property(p_tchk11);
c_tchk11: cover property(p_tchk11);
```

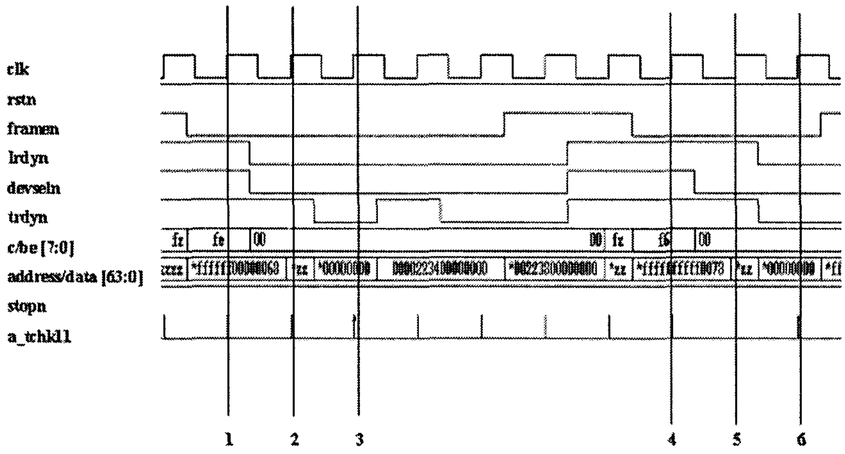

Figure 6-27. PCI Target check11

The sequence s_tchk11a detects a read command. The sequence s_tchk11b detects the first valid data phase after the read command was issued. The property p_tchk11 waits for the completion of the first data phase and then checks for the value of the par bit and the data bus in the previous cycle using the **$past** construct. If the values are not driven to "z" in the previous cycle, it is a violation.

Figure 6-27 shows a sample waveform of this check in a simulation. Marker 1 shows the point when a read command is detected (1110 – memory read line command). Marker 3 shows the point when the first valid data phase happens. One cycle before this, the data bus value should be a "z." Marker 2 shows that the value on the data bus is unknown and hence the check succeeds.

Marker 4 shows the point when a read command is detected (0110 – memory read command). Marker 6 shows the point when the first valid data phase happens. One cycle before this, the data bus value should be a "z." Marker 5 shows that the value on the data bus is unknown and hence the check succeeds.

Target_chk12: Configuration cycle (1).

During a valid configuration cycle, the 2 least significant bits of the address bus are set to either "00" or "01." When the configuration command is issued, the chip select signal "idsel" is asserted. The target device has to respond by asserting the signal "devseln" and eventually the configuration is completed when the signal "trdy" is asserted.

```
sequence s_tchk12a;
@(posedge clk)
(`s_CONFIG_READ || `s_CONFIG_WRITE) &&
((ad[1:0] == 2'b00) || (ad[1:0] == 2'b01)) &&
idsel;
endsequence

sequence s_tchk12b;
@(posedge clk)
!devseln && stopn;
endsequence

sequence s_tchk12;
@(posedge clk)
s_tchk12a ##[1:5] s_tchk12b;
endsequence

property p_tchk12;
@(posedge clk)
```

```
first_match(s_tchk12) |->
        ##[0:5] $fell (trdyn);
endproperty

a_tchk12: assert property(p_tchk12);
c_tchk12: cover property(p_tchk12);
```

Figure 6-28 shows a sample waveform of this check in a simulation. Marker 1 shows the point when a configuration command was issues by the system. Note that the signal "idsel" is asserted. Marker 2 shows the point when the target device asserts the signal "trdyn."

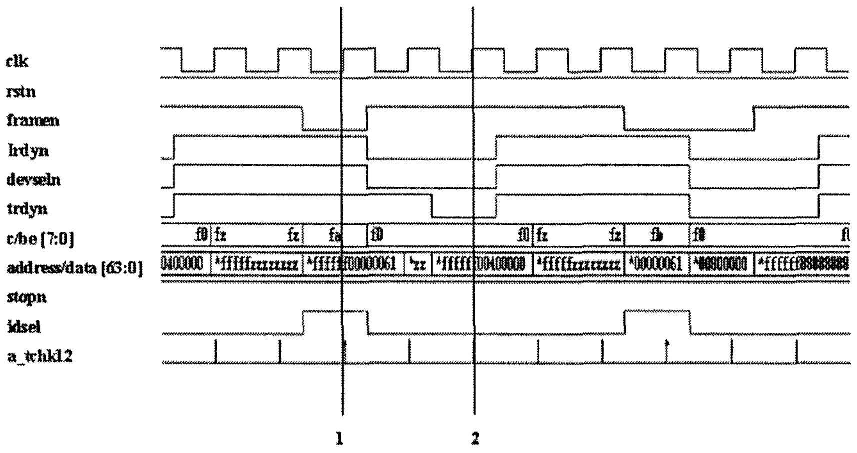

Figure 6-28. PCI Target check12

Target_chk13: Configuration cycle (2).

If the configuration command is issued and if the address bits are not set correctly ("10" or "11"), then the master should abort by de-asserting the "framen" signal.

```
sequence s_tchk13a;
@(posedge clk)
(`s_CONFIG_READ || `s_CONFIG_WRITE)
&& ((ad[1:0] == 2'b10) || (ad[1:0] == 2'b11))
&& idsel;
endsequence
```

```
sequence s_tchk13b;
@(posedge clk)
(devseln && stopn && trdyn) throughout
      (##[1:5] $rose (framen));
endsequence

property p_tchk13;
@(posedge clk)
   s_tchk13a |-> s_tchk13b;
endproperty

a_tchk13: assert property(p_tchk13);
c_tchk13: cover property(p_tchk13);
```

The sequence s_tchk13a detects an invalid configuration command. The sequence s_tchk13b makes sure that the signals "devseln," "trdyn" and "stopn" stay de-asserted until the signal "framen" is asserted. The signal "framen" should be de-asserted within 5 clock cycles.

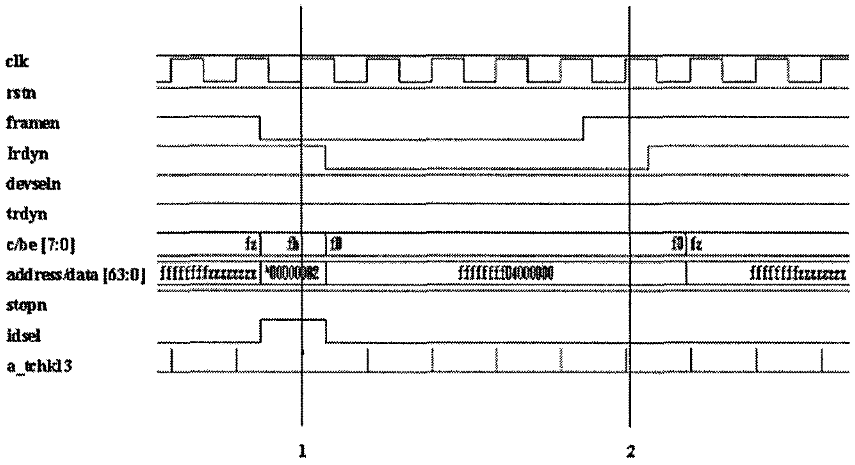

Figure 6-29. PCI Target check13

Figure 6-29 shows a sample waveform of this check in a simulation. Marker 1 shows the point when the invalid configuration command is detected. Marker 2 shows the point when the master aborts the transaction by de-asserting the "framen" signal. Note that the signals "trdyn," "devseln" and "stopn" stay de-asserted from marker1 to marker 2.

Sample functional coverage point for PCI Target:

Reserved commands – The signal "devseln" should not be asserted for any PCI reserved commands. The antecedent of the cover property looks for the reserved commands upon the assertion of the "framen" signal. The consequent makes sure that the signal "devseln" was kept de-asserted until the "framen" signal was de-asserted.

```
property p_tcov1;
@(posedge clk)
$fell (framen) && (cxben[3:0] == 4'b0100) |->
      devseln [*1:5] ##0 $rose (framen);
endproperty

c_tcov1: cover property(p_tcov1);

property p_tcov2;
@(posedge clk)
$fell (framen) && (cxben[3:0] == 4'b0101) |->
      devseln [*1:5] ##0 $rose (framen);
endproperty

c_tcov2: cover property(p_tcov2);

property p_tcov3;
@(posedge clk)
$fell (framen) && (cxben[3:0] == 4'b1000) |->
      devseln [*1:5] ##0 $rose (framen);
endproperty

c_tcov3: cover property(p_tcov3);

property p_tcov4;
@(posedge clk)
$fell (framen) && (cxben[3:0] == 4'b1001) |->
      devseln [*1:5] ##1 $rose (framen);
endproperty

c_tcov4: cover property(p_tcov4);
```

6.5 Scenario 3 – System level assertions

In this section, a few sample checks are shown for the PCI arbiter. The arbiter is usually part of the PCI bus. Figure 6-30 shows a sample system.

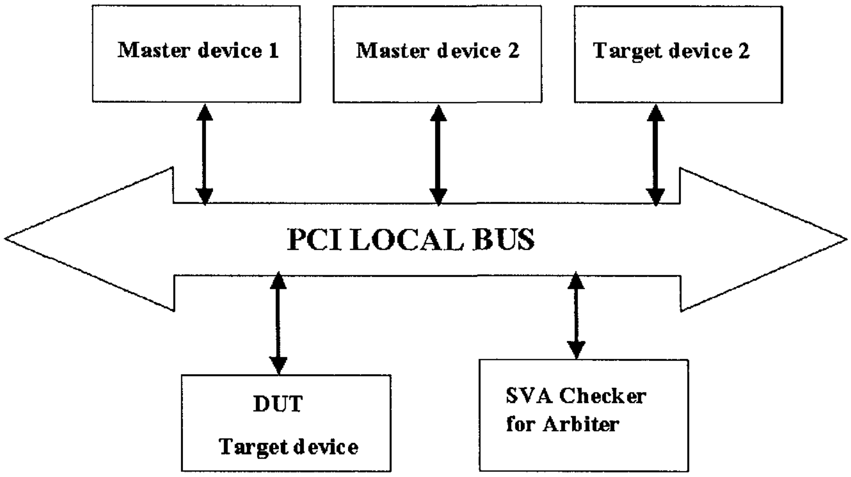

Figure 6-30. Sample PCI System for Arbiter checks

6.5.1 PCI Arbiter assertions

Arbiter_chk1: The signal "gntn" should be asserted when "framen" is asserted.

If the signal "gntn" is de-asserted and the signal "framen" is asserted in the same cycle, it is still valid.

```
property p_schk1;
@(posedge clk)
$fell (framen) |->
   !gntn[2] || $rose (gntn[2]);
endproperty

a_schk1: assert property(p_schk1);
c_schk1: cover property(p_schk1);
```

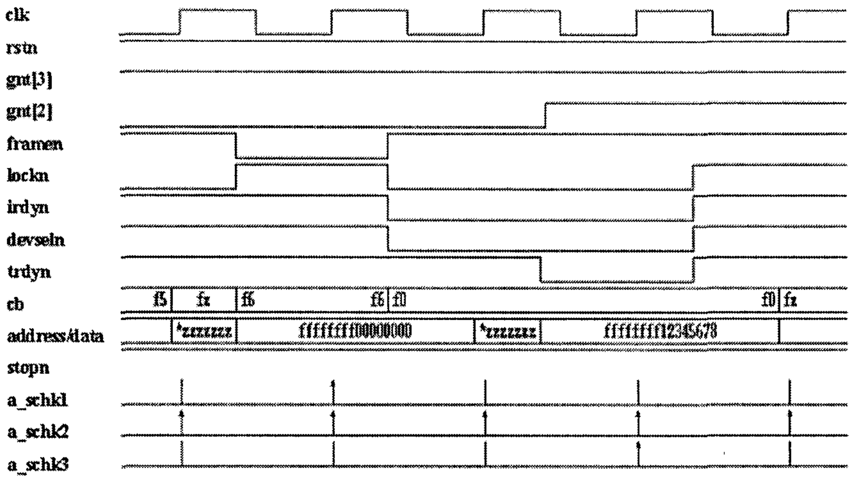

Figure 6-31. PCI Arbiter checks 1,2,3

Arbiter_chk2: Only one "gntn" signal can be asserted on a given clock cycle.

In the current sample system, there are two masters and hence, the arbiter uses two "gntn" signals.

```
property p_schk2;
@(posedge clk)
   $onehot0 ({!gntn[3], !gntn[2]});
endproperty

a_schk2: assert property(p_schk2);
c_schk2: cover property(p_schk2);
```

Since the "gntn" signals are active low signals, they are inverted and checked with a **zero one-hot** construct.

Arbiter_chk3: One "gntn" signal cannot be de-asserted and another asserted in the same cycle unless it is in idle cycle.

```
property p_schk3;
@(posedge clk)
$rose (gntn[2]) &&
(!framen || !irdyn) |->
```

```
        not $fell (gntn[3]);
endproperty

a_schk3: assert property(p_schk3);
c_schk3: cover property(p_schk3);
```

Figure 6-31 shows a sample waveform of the checks a_schk1, a_schk2 and a_schk3 in a simulation.

Arbiter_chk4: The signal "lockn" should be asserted for the whole data phase.

```
sequence s_schk4a;
@(posedge clk)
first_match($fell (lockn) ##[0:5] !devseln);
endsequence

sequence s_schk4b;
@(posedge clk)
framen && !irdyn && (!trdyn || !stopn);
endsequence

property p_schk4;
@(posedge clk)
s_schk4a |-> !lockn [*1:$] ##0 s_schk4b;
endproperty

a_schk4: assert property(p_schk4);
c_schk4: cover property(p_schk4);
```

Arbiter_chk5: The signal "lockn" should be de-asserted during address phase.

```
property p_schk5;
@(posedge clk)
$fell (lockn) |->
      (($past (framen) == 0)
      && ($past(framen,2) == 1));
endproperty

a_schk5: assert property(p_schk5);
c_schk5: cover property(p_schk5);
```

The antecedent of the property looks for the assertion of the "lockn" signal. The address phase occurs when the "framen" signal is asserted. By checking for the falling edge of the "framen" signal, we can confirm that an address phase just occurred. The **$past** operator is used to get the value of the "framen" signal in the past two cycles.

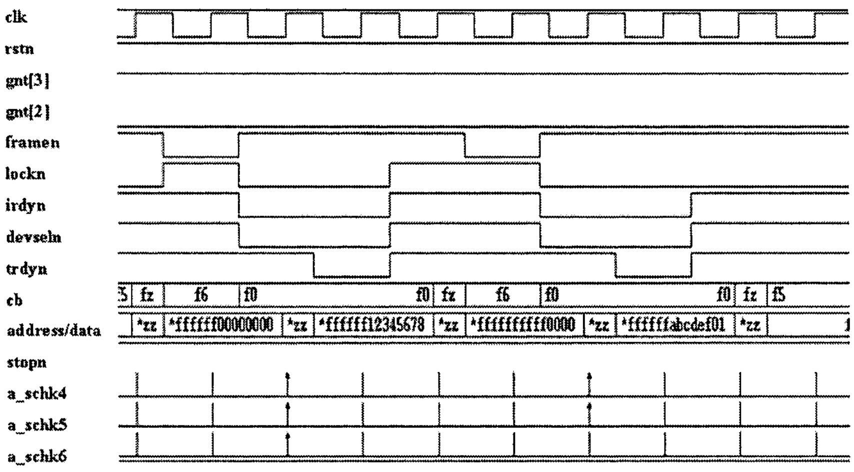

Figure 6-32. PCI Arbiter checks 4,5,6

Arbiter_chk6: The first transaction of a lock mechanism should be a read operation.

```
sequence s_schk6;
@(posedge clk)
first_match($fell (gntn[2]) ##[1:8]
      $fell (framen) ##1 $fell(lockn));
endsequence

property p_schk6;
@(posedge clk)
s_schk6.ended |->
   ($past(cxben[1:0]) == 2'b10);
endproperty

a_schk6: assert property(p_schk6);
c_schk6: cover property(p_schk6);
```

The sequence s_schk6 uses the "first_match" construct to identify the first assertion of the lock mechanism. The end of this is used as the antecedent of the property. The consequent part checks the 2 least significant bits of the command to make sure a read command was issued.

Figure 6-32 shows a sample waveform of the checks a_schk4, a_schk5 and a_schk6 in a simulation.

6.6 Summary on SVA for standard protocol

- Standard protocols are very complex and require a huge list of checkers to verify compliance.
- Timing rules are strict and these need to be achieved by the devices claiming to support these protocols.
- A common set of checkers can be developed for a particular interface and the same checkers can be re-used with other devices supporting similar interfaces.
- The complex nature of the protocol leads to multiple pre-conditions for most properties. Only SVA provides a variety of constructs and in-built mechanisms that can be used to define these complex pre-conditions.
- SVA also provides the capabilities to capture bus conditions using local variables. These local variables can be used effectively along with the pre-conditions to write complex temporal checks.
- SVA can be used effectively to create excellent functional coverage reports for a complex protocol.