Chapter 2

SVA SIMULATION METHODOLOGY

In Chapter 1, SVA language constructs were discussed in detail with examples. All examples were illustrated as relationships between two or more generic signals without any design details. In Chapter 2, a dummy system is used to present a real situation. The process of protocol extraction and assertion development will be discussed step by step. Various simulation methodologies that can significantly increase the productivity of assertion based verification will be discussed. Functional coverage and reactive testbench development will be discussed in detail.

2.1 A sample system under verification

The sample system under consideration is shown in Figure 2-1. The system has 3 master devices and 2 target devices. A link is established between the master and the target devices by the mediator. At a given time, only one master can conduct a transaction and with only one target device. Any master device can conduct a transaction with any target device. The transaction can be a read or a write. The mediator contains arbiter logic that decides which master will be allowed to conduct a transaction. The arbiter uses a simple round robin technique. The mediator also contains glue logic that actually decodes the master information for the target device and vice versa. The glue logic helps establish the link between a specific master device and target device to conduct the transaction successfully.

2.1.1 The Master device

The block diagram of the master device along with input and output ports is shown in Figure 2-2. The master device can perform a read and a write

transaction. It can support 2 target devices in a single system. When the master device gets the instruction "ask_for_it," it is ready to perform a transaction. It sends an active low pulse on the "req" signal and waits for a "gnt." The "gnt" signal is an active low signal. If the "gnt" signal does not come within 2 to 5 clock cycles, then the master will retry to get access at a later time. If the "gnt" is acquired, then the master will immediately assert the "frame" and "irdy" signals acknowledging the arrival of the "gnt" signal ("frame" and "irdy" are active low signals). In the same clock cycle it also selects the target device it will have the transaction with. The master uses the output signal "rsel" to indicate this. If signal "rsel" is set to 1, then the master will to have a transaction with target device 1. If the signal "rsel" is set to 0, then the master will have a transaction with target device 0.

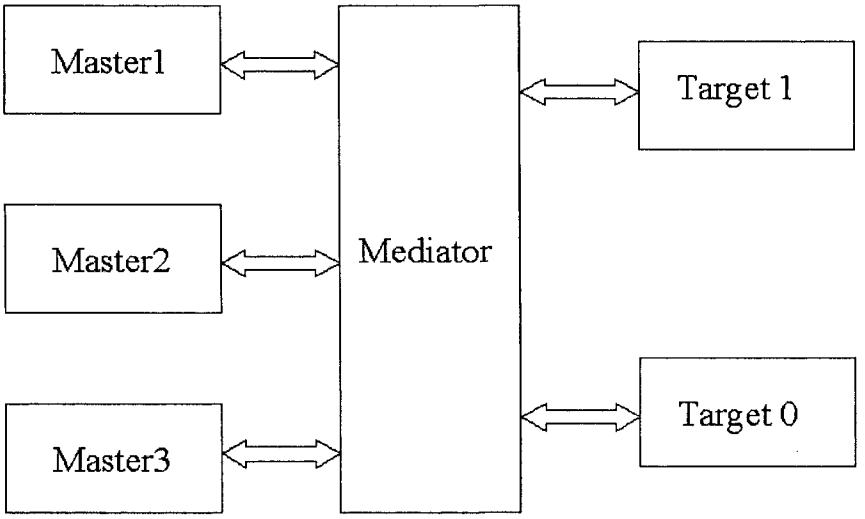

Figure 2-1. A sample system

Once the signal "rsel" is updated, the target device is expected to identify itself to the master. The target device uses the signal "trdy" to acknowledge its readiness. If the target does not acknowledge itself within 3 clock cycles from the point when "rsel" is assigned, it is an error condition. If the target does acknowledge itself, then the master decides whether to read or write. The master sends the data and the instruction whether to read or write through the "datac" bus.

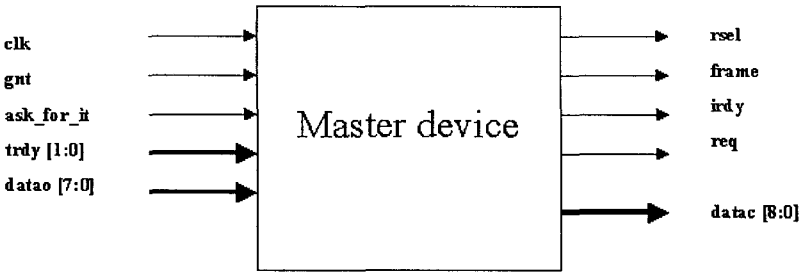

Figure 2-2. Sample master device

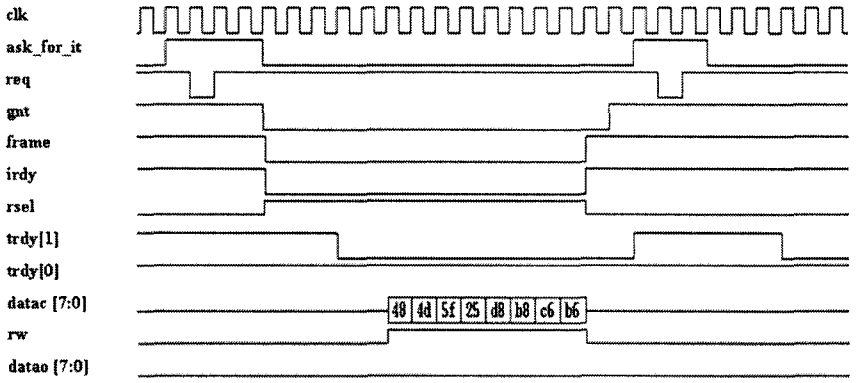

Figure 2-3. Write transaction of a master device

The most significant bit is the instruction bit (shown as signal "rw" in waveforms). If it is 1, the master will write and if it is a 0 then the master will read. If it is a write transaction, the least significant 8 bits consist of the data that needs to be written to the target device. If it is a read transaction, then the data read from the target device appears on the "datao" input bus. Each transaction of the master will last exactly 8 clock cycles. In other words, a master can either read 8 bytes in a transaction or write 8 bytes in a transaction. There is no specific address generation scheme. The master will write to the most updated write pointer address existing within the target device. Similarly, the master will read from the most updated read pointer address within the target device. The sample waveform for a master write transaction is shown in Figure 2-3. The sample waveform for a master read transaction is shown in Figure 2-4.

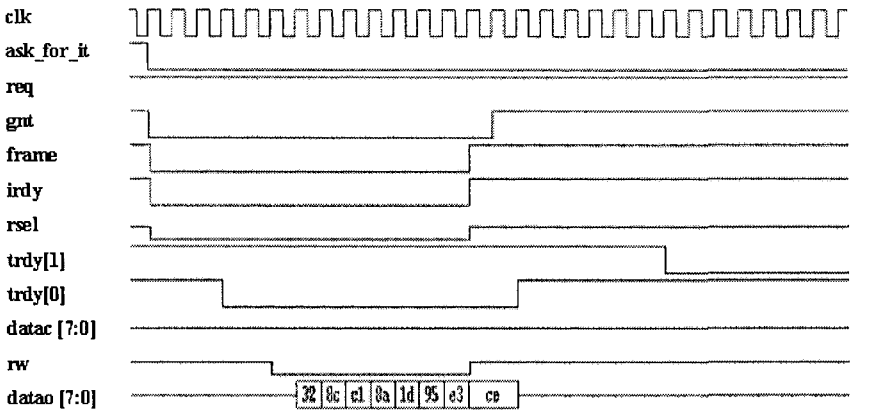

Figure 2-4. Sample read transaction of a master device

Once the read or write transaction is complete, the master indicates completion by de-asserting the signals "frame" and "irdy" in the next clock cycle. It also sets the "rsel" signal to tri-state. The arbiter acknowledges this and de-asserts the "gnt" signal in the next clock cycle. Once the arbiter removes the "gnt" signal, the target device acknowledges completion of the transaction by de-asserting the "trdy" signal.

2.1.2 The Mediator

The block diagram of the mediator along with input and output ports is shown in Figure 2-5. The mediator performs two important tasks:

1. Provide arbitration logic that decides which master will get access to conduct a transaction with a target device.

2. Establish the link between a specific master device and a target device. At a given time any number of masters can ask for access by asserting their respective "req" signal.

The arbiter uses a round robin algorithm and decides which master will get access. When the arbiter makes a decision, it will assert the "gnt" signal of the respective master device. The arbiter can take anywhere between 2 to 5 clock cycles to make a decision. The internal logic for the arbiter is described with a simple zero one-hot state machine.

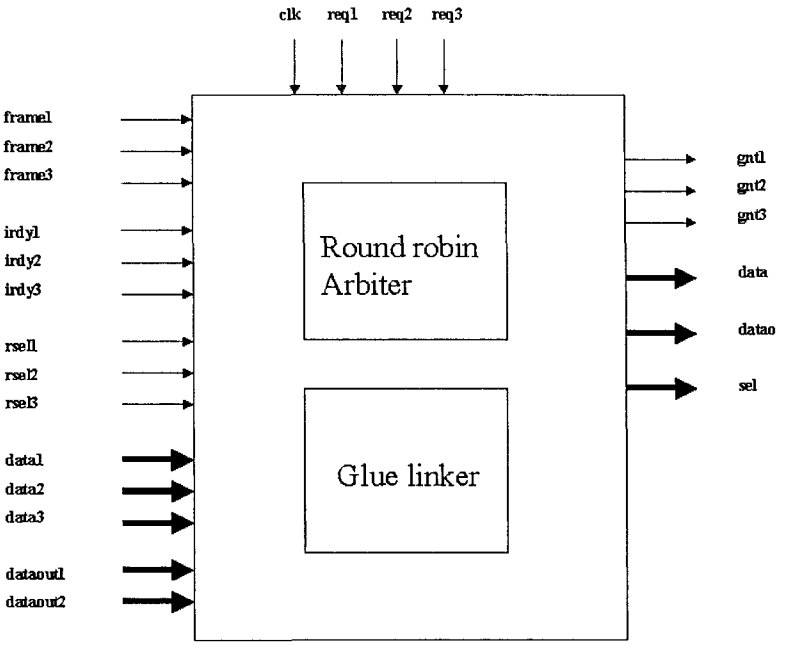

Figure 2-5. Sample mediator device

After the master selects the target it will have a transaction with, the mediator will provide that information to the specific target device. Since three masters are capable of having a transaction with any of the target devices, the mediator has to monitor the "rsel" signals from all three masters. At any given time, either all the three "rsel" signals are tri-stated or definitely two of them are tri-stated. If all three "rsel" signals are tri-stated, then there is no transaction request at that point. If there is a transaction, then one of the "rsel" signals will have a value of 0 or 1, depending on which target device will be used. If signal "rsel" is 1 then, the MSB of signal "sel" is set high indicating that target device 1 is selected. If signal "rsel" is 0 then, the LSB of signal "sel" is set high indicating that target device 0 is selected.

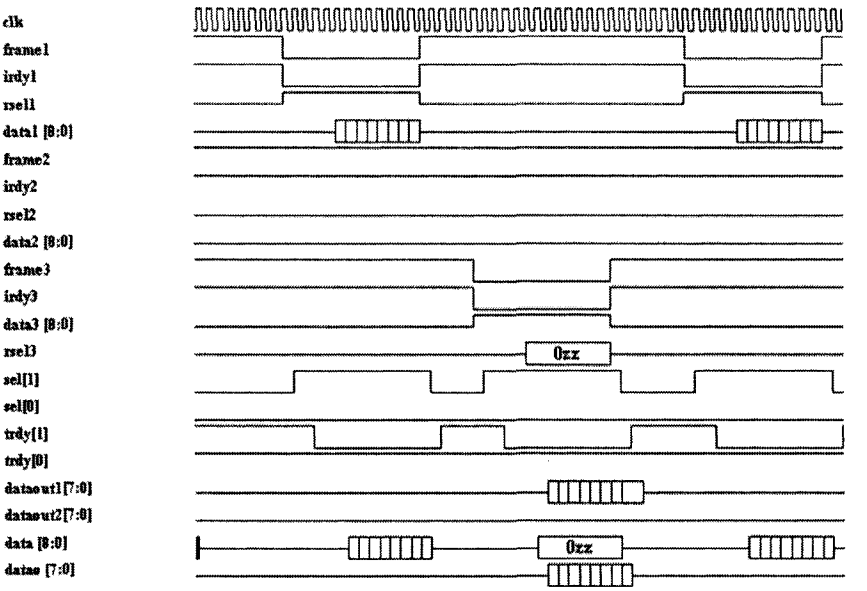

Figure 2-6. Waveform for mediator functionality

The mediator also selects the correct data signals for both write and read transactions. If it is a write transaction, then the mediator monitors which master's "rsel" signal is active and assigns the data value relevant to that master to the selected target device input. For example, if master 1 is asking for a write transaction with target device zero, then the signal "rsel1" will be set to low and the bus "data1" will be assigned to the mediator output bus "data." This output is fed to the input of the selected target device. The mediator also assigns the correct output data from the target device back to the master device in a read transaction. For example, if target 1 is involved in the read transaction, then the bus "dataout1" will be assigned to the bus "datao." The sample waveform for the mediator is shown in Figure 2-6.

2.1.3 The Target device

The block diagram of the target device along with input and output ports is shown in Figure 2-7. The target device has a first-in-first-out type memory that can store up to 64 bytes of data.

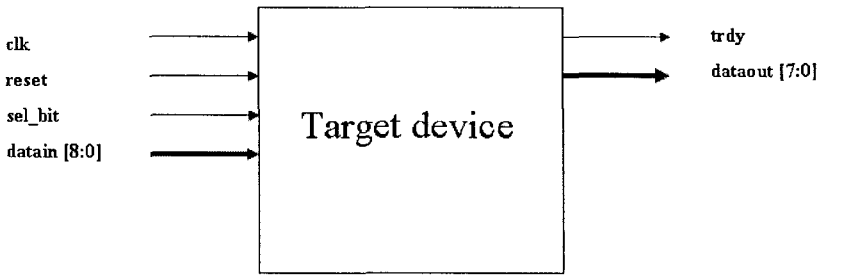

Figure 2-7. Sample target device

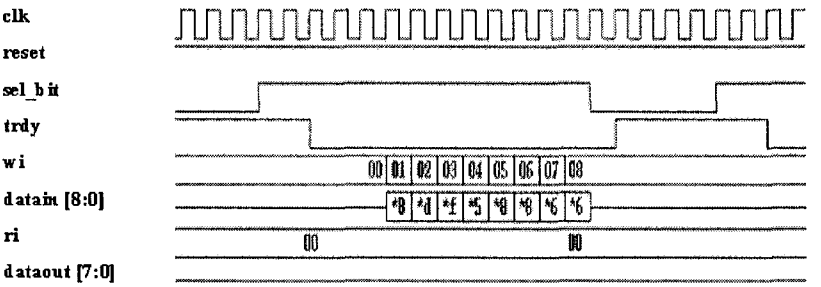

Figure 2-8. Target write transaction

The target device waits for the signal "sel_bit" to be asserted. Once signal "sel_bit" is asserted, the target has to acknowledge by asserting the signal "trdy" after 2 clock cycles. After asserting signal "trdy" the target device waits for a valid data and a valid write signal if it is a write transaction. Once a valid write signal is detected, the incoming data is stored in the target device in locations starting from the most updated value of the write pointer (wi) register. If it is a read transaction, then the target device reads out 8 data points from its memory using the current read pointer location (ri) as the starting address.

The type of transaction is indicated by the MSB of the bus "datain." In a read transaction, the data read appears on the bus vector "dataout." When the transaction is complete, the signal "sel_bit" is de-asserted and one clock cycle after that the signal "trdy" is de-asserted. The sample waveform for a target write transaction is shown in Figure 2-8. The sample waveform for a target read transaction is shown in Figure 2-9.

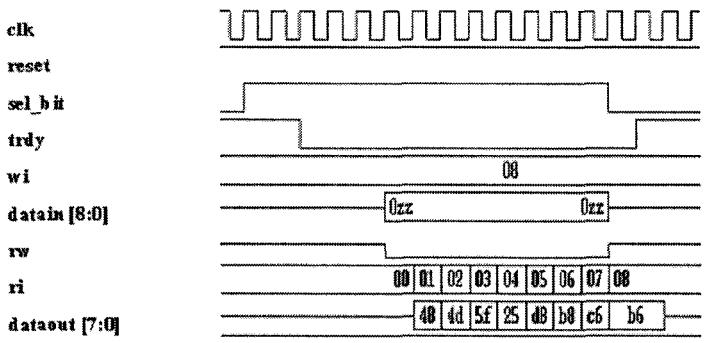

Figure 2-9. Target read transaction

2.2 Block level verification

As the individual design blocks get ready they should be tested thoroughly. Exhaustive verification of the blocks will uncover the corner case bugs ahead of time. Finding these bugs before integrating the system is a must. Finding these bugs at the system level will be very difficult. Also, system level failures provide a greater challenge for identifying and debugging corner case bugs. SVA can be used efficiently to test the individual blocks effectively. ***At the block level, the simulations are smaller and hence the bugs can be traced easily and fixed promptly.*** There are 4 individual design blocks in the sample system that need to be verified:

1. Master
2. Target
3. Arbiter
4. Glue

There are also 2 block level interfaces that need to be tested thoroughly:

1. Master and Mediator
2. Target and Mediator

2.2.1 SVA in design blocks

The following tips are recommended for doing block level verification with SVA:

- All SVA checks written for a block level design should be in-lined. Block level assertions often involve accessing internal registers of a design and hence, in-lining the checks within the design module is more efficient.

- The inclusion of SVA checks written at the block level should be controlled by a parameter defined within the design module. This gives the freedom to turn the checks on and off on a per simulation basis.

- The severity level of the SVA checks written at the block level should be controlled by a parameter defined within the design module. The default severity in SVA is to print an error message and continue simulating.

- Every block level SVA check written should be asserted and covered. It is a must that all the block level checks must have at least one real success.

2.2.2 Arbiter verification

Based on the protocol description of the arbiter from Section 2.1.2, the following SVA checks can be extracted. Some of the common expressions used repeatedly in the arbiter checks can be defined with "assign" statements as shown below:

```
assign frame = frame1 && frame2 && frame3;
assign irdy = irdy1 && irdy2 && irdy3;
assign gnt = !gnt1 || !gnt2 || !gnt3;
assign req = !req1 || !req2 || !req3;
```

The "frame" and "irdy" signals are all active low signals. Each master has a unique "frame" and "irdy" signal and these are inputs to the arbiter module. If a master is active, it sets both the "frame" and "irdy" low. Hence, by AND'ing the "frame" signals, we know that the bus is active if the AND'ed value is low. Similarly, by AND'ing the "irdy" signals, we know that the bus is active if the AND'ed value is low. If the AND'ed values of "frame" and "irdy" signals are high, then none of the masters are active.

Each master has a unique "req" signal that requests the bus and the arbiter provides a unique "gnt" signal. By OR'ing all the "req" signals we know that even if one master has a valid request, the arbiter considers the

request. Similarly, by OR'ing the "gnt" signals, we know that one master has acquired the grant. ***Creating such intermediate expressions make the SVA checkers more readable.***

Arb_chk1: On any given clock edge, the internal state of the arbiter should behave as a zero one-hot state machine.

```
property p_arb_onehot0;
  @(posedge clk) $onehot0(state);
endproperty
```

Arb_chk2: Upon a valid request by a master, the arbiter should provide a grant within 2 to 5 clock cycles.

```
property p_req_gnt;
  @(posedge clk) $rose (req) |->
              ##[2:5] $rose (gnt);
endproperty
```

Arb_chk3: Once the grant is awarded, the master should acknowledge acceptance in the same clock cycle by asserting the "frame" and "irdy" signals.

```
property p_gnt_frame;
  @(posedge clk) $rose (gnt) |->
                   $fell (frame && irdy);
endproperty
```

Arb_chk4: Once the master completes the transaction it de-asserts the "frame" and "irdy" signals, followed by that, the arbiter should de-assert the "gnt" signal on the next clock cycle.

```
property p_frame_gnt;
  @(posedge clk) $rose(frame && irdy)
                            |=> $fell(gnt);
Endproperty
```

2.2.3 SVA Checks for arbiter in simulation

The four checks shown in Section 2.2.2 should be in-lined within the arbiter module. There should be a provision to assert these properties on a need basis. The following code shows how this can be achieved.

```
module arbiter(…..);

// port declarations

parameter arb_sva = 1'b1;
parameter arb_sva_severity = 1'b1;

// Arbiter design description
// SVA property description

// SVA Checks

always@(posedge clk)
begin
if(arb_sva)
begin

a_arb_onehot0:
     assert property(p_arb_onehot0)
     else if(arb_sva_severity) $fatal;

a_req_gnt:
     assert property(p_req_gnt)
     else if(arb_sva_severity) $fatal;

a_gnt_frame :
     assert property(p_gnt_frame)
     else if(arb_sva_severity) $fatal;

a_frame_gnt:
     assert property(p_frame_gnt)
     else if(arb_sva_severity) $fatal;

c_arb_onehot0: cover property(p_arb_onehot0);
c_req_gnt: cover property(p_req_gnt);
c_gnt_frame: cover property(p_gnt_frame);
c_frame_gnt: cover property(p_frame_gnt);

end
end

endmodule
```

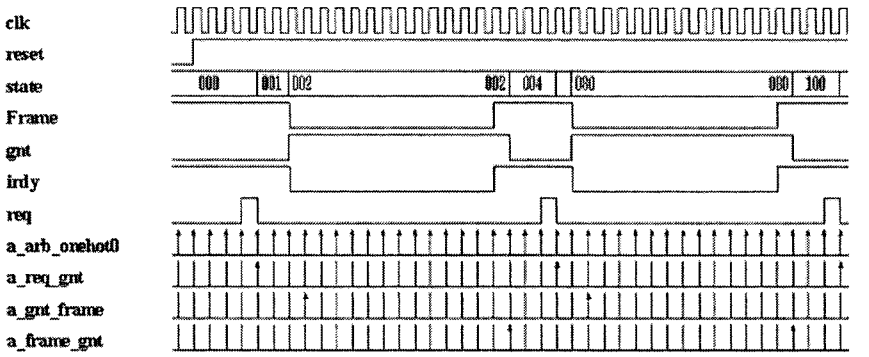

Figure 2-10. Arbiter checks in simulation

The parameter "arb_sva" will have to be set to 1 for the checks to be included in a simulation. The parameter "arb_sva_severity" controls the action to be taken during simulation. In this case, if the parameter is set to 1, then the severity is set to $fatal. This means that upon a failure of any of these checks, the simulation will exit. By setting the parameter to 0, the checks will use the default condition, which is to print an error message on a failure and continue simulating. A waveform from a sample simulation is shown in Figure 2-10.

2.2.4 Master verification

Based on the protocol description of the master from Section 2.1.1, the following SVA checks can be extracted. Note that each master has only one "req," "gnt," "frame" and "irdy" signals. The mention of these signals in the master checkers does not represent the expressions defined in the arbiter checkers. They are just individual signals present in each master device.

Master_chk1: Upon a valid request from a master, the grant shall come within 2 to 5 clock cycles. If so and if the signal "r_sel" is high, then on the same clock cycle, the master should assert the signals "frame" and "irdy." Three cycles later the target device one should acknowledge its selection by asserting the signal "trdy."

```
property p_master_start1;
@(posedge clk)
```

```
($fell (req) ##[2:5] ($fell(gnt) && r_sel)) |->
              (!frame && !irdy) ##3 !trdy[1];
endproperty
```

Master_chk2: Upon a valid request from a master, the grant shall come within 2 to 5 clock cycles. If so and if the signal "r_sel" is low, then on the same clock cycle, the master should assert the signals "frame" and "irdy." Three cycles later the target device zero should acknowledge its selection by asserting the signal "trdy."

```
property p_master_start2;
 @(posedge clk)
 ($fell (req) ##[2:5] ($fell(gnt) && !r_sel))|->
              (!frame && !irdy) ##3 !trdy[0];
endproperty
```

Master_chk3: Once the target acknowledges its selection, the master should complete its transaction within 10 clock cycles. It should indicate the transaction completion by de-asserting the signals "frame" and "irdy." One cycle later the signal "gnt" should be de-asserted.

```
property p_master_stop1;
 @(posedge clk)
$fell (trdy[1]) |-> ##10 (frame && irdy) ##1 gnt;
endproperty

property p_master_stop2;
 @(posedge clk)
$fell (trdy[0]) |-> ##10 (frame && irdy) ##1 gnt;
endproperty
```

Note that two separate properties are written to check the transaction completion, one for each target device.

Master_chk4: If the master is in a write transaction, then the bus data (data_c) should not be tri-stated and should have valid data.

```
property p_master_data1;
 @(posedge clk)
    ($fell (trdy[1]) ##2 rw) |->
        ($isunknown(data) == 0) [*7];
endproperty
```

```
property p_master_data2;
 @(posedge clk)
    ($fell (trdy[0]) ##2 rw) |->
       ($isunknown(data) == 0) [*7];
endproperty
```

- Note that two separate properties are written to check the validity of data during write transaction, one for each target device.
- Note that if the signal "rw" is high, then the master is conducting a write transaction.

Master_chk5: If the master is in a read transaction, then the bus data (data_o) should not be tri-stated and should have valid data.

```
property p_master_datao1;
 @(posedge clk)
      ($fell (trdy[1]) ##3 !rw) |=>
           ($isunknown(data_o) == 0) [*7];
endproperty

property p_master_datao2;
 @(posedge clk)
      ($fell (trdy[0]) ##3 !rw) |=>
         ($isunknown(data_o) == 0) [*7];
endproperty
```

- Note that two separate properties are written to check the validity of data during read transaction, one for each target device.
- Note that if the signal "rw" is low, then the master is conducting a read transaction.

2.2.5 SVA Checks for the master in simulation

The five checks shown in Section 2.2.4 should be in-lined within the master module. There should be a provision to assert these properties on a need basis. The following code shows how this can be achieved.

```
module master(.....);

// port declarations

parameter master_sva = 1'b1;
parameter master_sva_severity = 1'b1;
```

```
// Master design description

// SVA property description

// SVA Checks

always@(posedge clk)

begin

if(master_sva)

begin

a_master_start1:
    assert property(p_master_start1)
    else if(master_sva_severity) $fatal;

a_master_start2:
    assert property(p_master_start2)
    else if(master_sva_severity) $fatal;

a_master_stop1:
    assert property(p_master_stop1)
    else if(master_sva_severity) $fatal;

a_master_stop2:
    assert property(p_master_stop2)
    else if(master_sva_severity) $fatal;

a_master_data1:
    assert property(p_master_data1)
    else if(master_sva_severity) $fatal;

a_master_data2:
    assert property(p_master_data2)
    else if(master_sva_severity) $fatal;

a_master_datao1:
    assert property(p_master_datao1)
    else if(master_sva_severity) $fatal;
```

```
a_master_datao2:
    assert property(p_master_datao2)
    else if(master_sva_severity) $fatal;

c_master_start1: cover property(p_master_start1);
c_master_start2: cover property(p_master_start2);
c_master_stop1: cover property(p_master_stop1);
c_master_stop2: cover property(p_master_stop2);
c_master_data1: cover property(p_master_data1);
c_master_data2: cover property(p_master_data2);
c_master_datao1: cover property(p_master_datao1);
c_master_datao2: cover property(p_master_datao2);

end

end

endmodule
```

A waveform from a sample simulation of these master checks is shown in Figure 2-11.

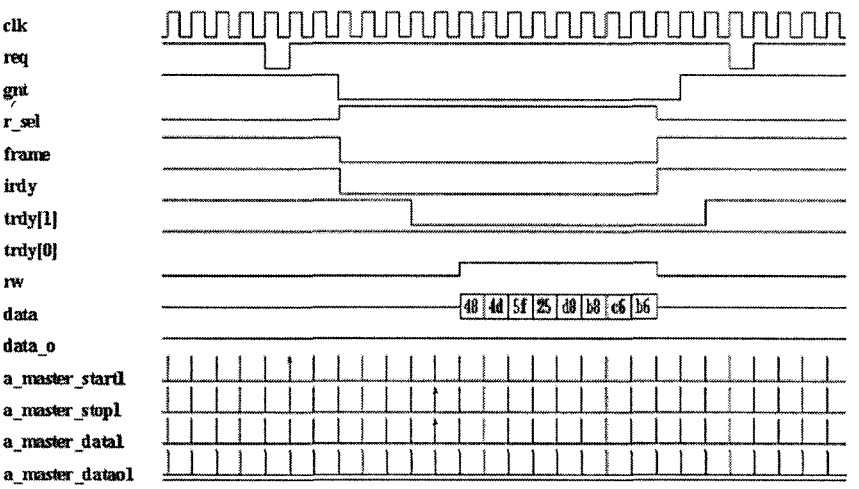

Figure 2-11. Master checks in simulation for target 1

2.2.6 Glue verification

Based on the protocol description of the glue logic from Section 2.1.2, the following SVA checks can be extracted.

Glue_chk1: If any one of the master select signals "sel1," "sel2" or "sel3" is high, then target device one should be selected.

```
property p_sel_1;
  @(posedge clk)
    (rsel1 || rsel2 || rsel3) |=> sel == 2'b10;
endproperty
```

Glue_chk2: If any one of the master select signals "sel1," "sel2" or "sel3" is low, then target device zero should be selected.

```
property p_sel_0;
  @(posedge clk)
    (!rsel1 || !rsel2 || !rsel3) |=> sel == 2'b01;
endproperty
```

Glue_chk3: During a write transaction, if the signal "rsel1" is not tri-stated, then the data from master device one should be written to the respective target device.

```
property p_rsel1_write;
  @(posedge clk)
((rsel1 || !rsel1) ##3 ($fell (trdy[1]) ||
$fell(trdy[0])) ##3 data1[8]) |->
       (data == $past(data1)) [*7];
endproperty
```

- Note that we determine the nature of the transaction (read/write) by using the most significant bit of the bus "data."
- If the MSB of the bus "data" is high, then it is a write transaction.
- If the MSB of the bus "data" is low, then it is a read transaction.
- Within the master device, the nature of the transaction is determined by the signal "rw." This signal is a copy of the MSB of the bus "data." The signal "rw" is local to the master device. The external interface should infer the nature of the transaction by using the MSB of the bus "data."

Glue_chk4: During a write transaction, if the signal "rsel2" is not tri-stated, then the data from master device two should be written to the respective target device.

```
property p_rsel2_write;
 @(posedge clk)
((rsel2  ||  !rsel2)  ##3  ($fell  (trdy[1])  ||
$fell(trdy[0])) ##3 data2[8]) |->
       (data == $past(data2)) [*7];
endproperty
```

Glue_chk5: During a write transaction, if the signal "rsel3" is not tri-stated, then the data from master device three should be written to the respective target device.

```
property p_rsel3_write;
 @(posedge clk)
((rsel3  ||  !rsel3)  ##3  ($fell  (trdy[1])  ||
$fell(trdy[0])) ##3 data3[8]) |->
         (data == $past(data3)) [*7];
Endproperty
```

Glue_chk6: During a read transaction, if target device one is selected, then data read from target one (dataout1) should be fed back to the respective master.

```
property p_read1;
 @(posedge clk)
($fell (trdy[1]) ##4 !data[8]) |->
              (dataout1 == datao) [*7];
endproperty
```

Glue_chk7: During a read transaction, if target device zero is selected, then data read from target zero (dataout2) should be fed back to the respective master.

```
property p_read0;
 @(posedge clk)
($fell (trdy[0]) ##4 !data[8]) |->
              (dataout2 == datao) [*7];
endproperty
```

2.2.7 SVA Checks for the glue logic in simulation

The seven checks shown in Section 2.2.6 should be in-lined within the glue module. There should be a provision to assert these properties on a need basis. The following code shows how this can be achieved.

```
module glue(.....);

// port declarations

parameter glue_sva = 1'b1;
parameter glue_sva_severity = 1'b1;

// glue design description
// glue SVA property description

// SVA Checks

always@(posedge clk)
begin
if(glue_sva)
begin

a_sel_1:
     assert property(p_sel_1)
     else if(glue_sva_severity) $fatal;

a_sel_0:
     assert property(p_sel_0)
     else if(glue_sva_severity) $fatal;

a_rsel1_write:
     assert property(p_rsel1_write)
     else if(glue_sva_severity) $fatal;

a_rsel2_write:
     assert property(p_rsel2_write)
     else if(glue_sva_severity) $fatal;

a_rsel3_write:
     assert property(p_rsel3_write)
     else if(glue_sva_severity) $fatal;
```

```
a_read1:
     assert property(p_read1)
     else if(glue_sva_severity) $fatal;

a_read0:
     assert property(p_read0)
     else if(glue_sva_severity) $fatal;

c_sel_1: cover property(p_sel_1);
c_sel_0: cover property(p_sel_0);
c_rsel1_write: cover property(p_rsel1_write);
c_rsel2_write: cover property(p_rsel2_write);
c_rsel3_write: cover property(p_rsel3_write);
c_read1: cover property(p_read1);
c_read0: cover property(p_read0);

end
end

endmodule
```

A waveform from a sample simulation of the glue checks is shown in Figure 2-12.

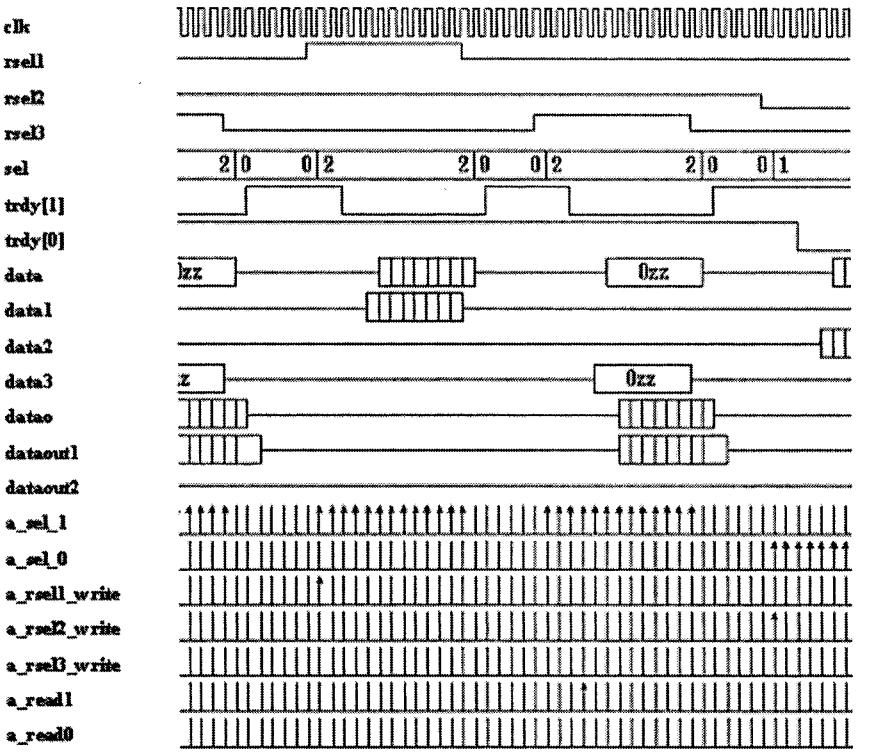

Figure 2-12. Glue checks in simulation

2.2.8 Target verification

Based on the protocol description of the target device from Section 2.1.3, the following SVA checks can be extracted.

Target_chk1: If a target is selected, then it should assert the signal "trdy" after 2 clock cycles.

```
property p_sel_trdy_start;
  @(posedge clk) $rose (sel_bit) |->
                   ##1 trdy ##1 !trdy;
endproperty
```

Target_chk2: At the end of a transaction, the "sel_bit" signal is de-asserted. One clock cycle after that, the signal "trdy" should be de-asserted.

```
property p_sel_trdy_stop;
  @(posedge clk) $fell (sel_bit) |=> trdy;
endproperty
```

Target_chk3: In a write transaction, the write pointers should be incremented by one after each clock cycle to complete a valid "write" to a unique address every time.

```
property p_write;
  @(posedge clk)
(datain[8] && sel_bit && (wi != 0)) |->
                    (wi == ($past(wi) + 1));
endproperty
```

- Note that the address pointer will roll over from 63 to 0. Hence, this check cannot be applied if on a given clock edge the write pointer is at 0.
- A different check can be written to verify that the pointer always rolls over correctly from 63 to 0.

Target_chk4: In a read transaction, the read pointers should be incremented by one after each clock cycle to complete a valid "read" from a unique address every time.

```
property p_read;
  @(posedge clk)
(!datain[8] && sel_bit && (ri != 63)) |=>
                      (ri == ($past(ri) + 1));
endproperty
```

- Note that in the case of read pointer, when the pointer is at 63 this check cannot be applied.
- The read operation has a latency of one clock cycle and hence we use the Non-overlapping implication operator.
- Since a non-overlapping operator is used, the check moves forward to one cycle and compares the address in the previous cycle.
- For example, on a given clock edge, if the antecedent of the implication is true, the check moves to the next clock cycle. If the pointer is at 63, then the check moves to pointer 0 and compares 63 and 0 for an increment of one. This is incorrect. Hence, the check should not be performed if the value of the read pointer is 63 on a given clock edge.

- A separate check can be written to make sure that the pointer rolls over from 63 to 0 accurately.

Target_chk5: During a valid read or write transaction, the data read from or written to the target should be valid.

```
property p_target_datain;
  @(posedge clk)
   ($fell (trdy) ##3 (datain[8])) |->
            not ($isunknown (datain)) [*7];
endproperty

property p_target_dataout;
  @(posedge clk)
  ($fell (trdy) ##3 (!datain[8])) |=>
          not ($isunknown(dataout)) [*7];
endproperty
```

2.2.9 SVA Checks for the Target in simulation

The five checks shown in Section 2.2.8 should be in-lined within the target module. There should be a provision to assert these properties on a need basis. The following code shows how this can be achieved.

```
module target(.....);

// port declarations

parameter target_sva = 1'b1;
parameter target_sva_severity = 1'b1;

// target design description
// target SVA property description
// SVA Checks

always@(posedge clk)
begin
if(target_sva)
begin

a_sel_trdy_start:
   assert property(p_sel_trdy_start)
```

```
    else if(target_sva_severity) $fatal;
a_sel_trdy_stop:
    assert property(p_sel_trdy_stop)
    else if(target_sva_severity) $fatal;

a_write:
     assert property(p_write)
     else if(target_sva_severity) $fatal;

a_read:
     assert property(p_read)
     else if(target_sva_severity) $fatal;

a_target_datain:
     assert property(p_target_datain)
     else if(target_sva_severity) $fatal;

a_target_dataout:
     assert property(p_target_dataout)
     else if(target_sva_severity) $fatal;

c_sel_trdy_start:
              cover property(p_sel_trdy_start);
c_sel_trdy_stop: cover property(p_sel_trdy_stop);
c_write: cover property(p_write);
c_read: cover property(p_read);
c_target_datain: cover property(p_target_datain);
c_target_dataout:
              cover property(p_target_dataout);

end
end
endmodule
```

A waveform from a sample simulation of the target checks is shown in Figure 2-13.

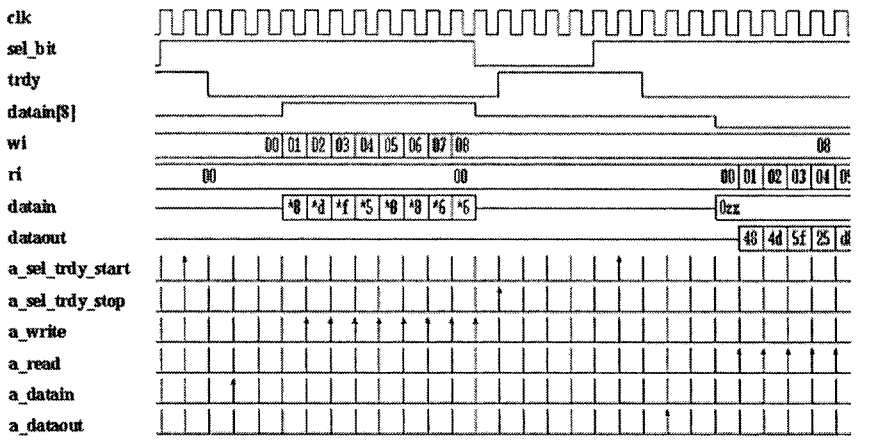

Figure 2-13. Target checks in simulation

2.3 System level verification

There are 3 masters and 2 targets in the system along with an instance of the mediator. The top-level connection of the system is shown below.

```
Module top(..,..,     );

// port declarations

master u1 (ask[2], clk, req1, gnt1, frame1,
irdy1, trdy, data1, rsel1, datao);

master u2 (ask[1], clk, req2, gnt2, frame2,
irdy2, trdy, data2, rsel2, datao);

master u3 (ask[0], clk, req3, gnt3, frame3,
irdy3, trdy, data3, rsel3, datao);

arbiter u4 (clk, reset, frame, irdy, req1, req2,
req3, gnt1, gnt2, gnt3);

glue u5 (clk, frame1, irdy1, frame2, irdy2,
frame3, irdy3, trdy, rsel1, rsel2, rsel3, data1,
```

```
data2, data3, sel, data, dataout1, dataout2,
datao);

target u6 (clk, reset, sel[1], trdy[1], data,
dataout1);

target u7 (clk, reset, sel[0], trdy[0], data,
dataout2);

endmodule
```

The following tips are recommended for doing system level verification with SVA:

- Since the internal functionality of the individual blocks was verified thoroughly, the block level assertions don't have to be included during the system level verification by default. The main motive behind this is performance.

- If performance is not a bottleneck, the block level assertions shall be included in the system level verification by default. The system interfaces provide a more realistic and unexpected set of input conditions and block level assertions must be able to react to them correctly.

- The verification environment should provide the facility to turn on block level assertions if there are any failures. For example, in our sample system, if a failure occurs during a transaction between master 1 and target 0, then the system level simulation should be re-run by including the block level SVA checks written for master 1 and target 0.

- At the system level, a new set of assertions should be written that verifies the connectivity of the system. More focus should be on the interface rules rather than the internal block details.

2.3.1 SVA Checks for system level verification

The following set of checks can be written for the system level verification based on the connectivity and protocol of the system.

Ss_shk1: Only one "trdy" signal can be asserted at any given point. In other words, only one target device can participate in a transaction at any given time.

```
property p_target;
 @(posedge clk) not (!trdy[0] && !trdy[1]);
endproperty
```

Ss_chk2: Only one set of "frame" and "irdy" signals can be asserted at any given clock cycle. In other words, only one master device can participate in a transaction at any give time.

```
property p_frame;
 @(posedge clk)
     $countones({frame1, frame2, frame3}) >1;
endproperty

property p_irdy;
 @(posedge clk)
     $countones({irdy1, irdy2, irdy3}) >1;
endproperty
```

Ss_chk3: Only one "gnt" signal shall be asserted at any given time. In other words, the arbiter can provide access for only one master at a time to pursue a transaction.

```
property p_gnt;
 @(posedge clk)
       $countones({gnt1, gnt2, gnt3}) > 1;
endproperty
```

Ss_chk4: Only one "rw" signal shall be active at any given clock cycle, the other "rw" signals should be tri-stated ("rw" signal is the MSB of the masters data output bus).

```
property p_rw;
 @(posedge clk)
 ($isunknown(rw1)     &&     $isunknown(rw2)     &&
 $isunknown(rw3) ) ||
 ((rw1==1'b1  ||  rw1==1'b0)  &&  $isunknown  (rw2)
 && $isunknown(rw3)) ||
 ((rw2==1'b1  ||  rw2==1'b0)  &&  $isunknown  (rw1)
 && $isunknown(rw3)) ||
```

```
  ((rw3==1'b1 || rw3==1'b0) && $isunknown (rw2)
  && $isunknown(rw2));
endproperty
```

Ss_chk5: Only one "rsel" signal shall be active at any given clock cycle, the other "rsel" signals should be tri-stated.

```
property p_rsel;
  @(posedge clk)
$isunknown(rsel1)    &&    $isunknown(rsel2)    &&
$isunknown(rsel3) ) ||
  ((rsel1==1'b1  ||  rsel1==1'b0)  &&  $isunknown
(rsel2) && $isunknown(rsel3)) ||
  ((rsel2==1'b1  ||  rsel2==1'b0)  &&  $isunknown
(rsel1) && $isunknown(rsel3)) ||
  ((rsel3==1'b1  ||  rsel3==1'b0)  &&  $isunknown
(rsel2) && $isunknown(rsel1));
endproperty
```

Ss_chk6: Upon a valid request by a master, a valid "gnt" should arrive within 2 to 5 clock cycles.

```
assign req = !req1 || !req2 || !req3;
assign gnt = !gnt1 || !gnt2 || !gnt3;

property p_req_gnt_w;
  @(posedge clk)
       $rose (req) |-> ##[2:5] $rose(gnt);
endproperty
```

Ss_chk7: At any given clock, if the "frame" and "irdy" signal of a master are asserted, then the relevant "trdy" signal should be asserted after 3 clock cycles.

```
assign frame_ = !frame1 || !frame2 || !frame3;
assign irdy_ = !irdy1 || !irdy2 || !irdy3;

property p_start_frame;
  @(posedge clk)
     $rose (frame_ && irdy_) |->##3 $rose(trdy_);
endproperty
```

Ss_chk8: At any given clock, if the "frame" and "irdy" signals of the master are de-asserted, then the relevant "trdy" signal should be de-asserted after 2 clock cycles.

```
assign trdyp = trdy[1] && trdy[0];

property p_end_frame;
  @(posedge clk)
        $rose (frame && irdy) |->##2 $rose(trdyp);
endproperty
```

Ss_chk9: If there is no valid transaction at any given clock, then the bus "data" and "datao" should be tri-stated.

```
property p_bus_not_in_use;
  @(posedge clk)
   trdyp |->
        ($isunknown(data) && $isunknown(datao));
endproperty

a_target : assert property(p_target);
a_frame: assert property(p_frame);
a_irdy: assert property(p_irdy);
a_rsel: assert property(p_rsel);
a_rw: assert property(p_rw);
a_gnt: assert property(p_gnt);
a_req_gnt_w : assert property(p_req_gnt_w);
a_start_frame: assert property(p_start_frame);
a_end_frame: assert property(p_end_frame);
a_bus_in_use: assert property(p_bus_not_in_use);

c_target : cover property(p_target);
c_frame: cover property(p_frame);
c_irdy: cover property(p_irdy);
c_rsel: cover property(p_rsel);
c_rw: cover property(p_rw);
c_gnt: cover property(p_gnt);
c_req_gnt_w : cover property(p_req_gnt_w);
c_start_frame: cover property(p_start_frame);
c_end_frame: cover property(p_end_frame);
c_bus_in_use: cover property(p_bus_not_in_use);
```

During the system level simulation, the top-level module should be configured with the parameter settings such that all block level assertions are turned off. In our sample system, since each design block has a parameter that allows including its relevant SVA checks on a need basis, we can configure the top module for system level run easily as shown below.

```
Module top(..,..,     );

// port declarations

master
#(.master_sva(1'b0),   .master_sva_severity(1'b0))
u1 (ask[2], clk, req1, gnt1, frame1, irdy1, trdy,
data1, rsel1, datao);

master
#(.master_sva(1'b0),   .master_sva_severity(1'b0))
u2 (ask[1], clk, req2, gnt2, frame2, irdy2, trdy,
data2, rsel2, datao);

master
#(.master_sva(1'b0),   .master_sva_severity(1'b0))
u3 (ask[0], clk, req3, gnt3, frame3, irdy3, trdy,
data3, rsel3, datao);

arbiter
#(.arb_sva(1'b0), .arb_sva_severity(1'b0))
u4 (clk, reset, frame, irdy, req1, req2, req3,
gnt1, gnt2, gnt3);

glue
#(.glue_sva(1'b0), .glue_sva_severity(1'b0))
u5 (clk, frame1, irdy1, frame2, irdy2, frame3,
irdy3, trdy, rsel1, rsel2, rsel3, data1, data2,
data3, sel, data, dataout1, dataout2, datao);

target
#(.target_sva(1'b0),   .target_sva_severity(1'b0))
u6 (clk, reset, sel[1], trdy[1], data, dataout1);

target
#(.target_sva(1'b0),   .target_sva_severity(1'b0))
u7 (clk, reset, sel[0], trdy[0], data, dataout2);
```

```
endmodule
```

Note that when each design block is instantiated, the parameter values are passed. The first parameter "*_sva" is set to 0 in all the individual instantiations, which indicates that the block level assertions will not be included. Now, the system level simulations can be run only with the system level checks.

Let us assume that there are failures on "Ss_chk6" during the system level simulation. This check looks for interface failures between the masters and the arbiter module. To debug the errors, the simulation can be re-run by including the block level checks relevant to the masters and the arbiter. The top modules configuration for such a run is shown below:

```
Module top(..,..,    );

// port declarations

master
#(.master_sva(1'b1),  .master_sva_severity(1'b0))
u1 (ask[2], clk, req1, gnt1, frame1, irdy1, trdy,
data1, rsel1, datao);

master
#(.master_sva(1'b1),  .master_sva_severity(1'b0))
u2 (ask[1], clk, req2, gnt2, frame2, irdy2, trdy,
data2, rsel2, datao);

master
#(.master_sva(1'b1),  .master_sva_severity(1'b0))
u3 (ask[0], clk, req3, gnt3, frame3, irdy3, trdy,
data3, rsel3, datao);

arbiter
#(.arb_sva(1'b1), .arb_sva_severity(1'b0))
u4 (clk, reset, frame, irdy, req1, req2, req3,
gnt1, gnt2, gnt3);

glue
#(.glue_sva(1'b0), .glue_sva_severity(1'b0))
u5 (clk, frame1, irdy1, frame2, irdy2, frame3,
irdy3, trdy, rsel1, rsel2, rsel3, data1, data2,
data3, sel, data, dataout1, dataout2, datao);
```

```
target
 #(.target_sva(1'b0), .target_sva_severity(1'b0))
u6 (clk, reset, sel[1], trdy[1], data, dataout1);

target
#(.target_sva(1'b0),  .target_sva_severity(1'b0))
u7 (clk, reset, sel[0], trdy[0], data, dataout2);

endmodule
```

Note that the parameter "master_sva" and "arb_sva" are set to 1 in this configuration. In the basic design blocks, SVA checks could also be included conditionally using the "`ifdef - `endif" construct. By conditionally compiling the SVA code, the user can either have the checks on all instances of the module or on none of the instances of the module. The disadvantage with this methodology is that, it is a global control mechanism. By using parameters, this disadvantage can be overcome and the user gets more flexibility in choosing the block level checks needed for a particular simulation run.

2.4 Functional coverage

The system level checks written so far look for specific protocol violations, if any. By making sure that these checks executed at least once in the simulation, the confidence level on the functionality of the system increases tremendously. The other aspect of functional coverage is covering all possible scenarios of system functionality during simulation from the testbench perspective. ***The scenarios to be covered during a simulation should be part of the test plan.***

The SVA checks written for dynamic simulation are only as good as the input stimulus. If the input vectors do not force the system to execute certain scenarios, then those remain untested. A lot of testbenches use random techniques to generate input stimulus vectors. A very common approach is to run a pre-determined number of transactions and measure coverage on certain scenarios. By constraining the random generation of input stimulus, the scenarios can be covered more efficiently. ***The key is to get the maximum functional coverage in a minimum number of cycles.*** The coverage information collected from SVA can be used effectively to create reactive verification environments.

2.4.1 Coverage plan for the sample system

The sample system discussed in this chapter has a lot of key functionality that should be covered as part of the functional verification.

2.4.1.1 Request Scenario

"All possible request scenarios should be covered"

There are three masters that can ask for access at any given time. This means that there are 7 possible combinations of the master "req" signals as shown in Table 2-1.

Table 2-1. Master request scenarios

Req1	Req2	Req3
0	1	1
1	0	1
1	1	0
0	0	1
1	0	0
0	1	0
0	0	0

A 0 in the table indicates that the master is requesting for the bus. The testbench should create all these possible input combinations during simulation.

The following code example shows how functional coverage data can be used to control the simulation environment. Property definitions for all 7 possible request combinations should be created as follows.

```
property p_req1; // master 1 requesting
  @(posedge clk) $fell (req1) && req2 && req3;
endproperty

property p_req2; // master 2 requesting
  @(posedge clk) $fell (req2) && req1 && req3;
endproperty

property p_req3; // master 3 requesting
  @(posedge clk) $fell (req3) && req1 && req2;
endproperty
```

```
property p_req12; // master 1&2 requesting
 @(posedge clk)
 $fell (req1) && $fell(req2)&& req3;
endproperty

property p_req23; // master 2&3 rtequesting
 @(posedge clk)
 $fell (req2) && $fell(req3) && req1;
endproperty

property p_req31; // master 1&3 requesting
 @(posedge clk)
 $fell (req3) && $fell(req1) && req2;
endproperty

property p_req123; // master 1&2&3 requesting
 @(posedge clk)
 $fell (req1) && $fell(req2) && $fell(req3);
endproperty
```

Each property should have a cover statement associated with it as shown below. The action block of the cover statement can be used to update register flags. In this case, every time the property is covered, a local register count is incremented. In the same clock, we check if the counter has reached a value of 3. If so, then the flag associated to that property is asserted. In other words, it is expected that each request combination occurs three times during simulation and if and when it happens, a flag associated with that specific request combination will be asserted.

```
c_req1: cover property(p_req1)
   begin
     creq1++;
     if(creq1 == 3) creq1_flag = 1'b1;
   end

c_req2: cover property(p_req2)
   begin
     creq2++;
     if(creq2 == 3) creq2_flag = 1'b1;
   end

c_req3: cover property(p_req3)
   begin
```

```
      creq3++;
      if(creq3 == 3) creq3_flag = 1'b1;
    end
c_req12: cover property(p_req12)
    begin
      creq12++;
      if(creq12 == 3) creq12_flag = 1'b1;
    end

c_req23: cover property(p_req23)
    begin
      creq23++;
      if(creq23 == 3) creq23_flag = 1'b1;
    end

c_req31: cover property(p_req31)
    begin
      creq31++;
      if(creq31 == 3) creq31_flag = 1'b1;
    end

c_req123: cover property(p_req123)
    begin
      creq123++;
      if(creq123 == 3) creq123_flag = 1'b1;
    end
```

This coverage information can be used effectively to control the simulation environment. In a random testbench for the sample system, a predetermined number of transactions could be performed one after the other. The simulation will finish when all transactions are completed. The following code shows how the functional coverage information can be used to terminate the simulation.

```
always@(posedge clk)
begin

if(creq1_flag && creq2_flag && creq3_flag &&
creq12_flag && creq23_flag && creq31_flag &&
creq123_flag)

begin
```

```
    $display("FC: All possible request scenarios
    covered 3 times each\n");
    $finish();

  end
  end
```

With this piece of code, there are two ways to terminate a simulation:

1. Run the pre-determined number of transactions randomly and exit.
2. Exit if all possible request scenarios are covered three times each.

Whichever occurs first will terminate the simulation.

2.4.1.2 Master to Target transactions

"Every master device should perform both a read and a write transaction with every target device"

There are 3 master devices and 2 target devices in the system. This creates 12 possible scenarios as shown in Table 2-2. Property definitions for all 12 possible transaction combinations should be created as follows.

Table 2-2. Master to target transactions

Master	Target	Transaction
M1	T1	Read
M1	T1	Write
M1	T0	Read
M1	T0	Write
M2	T1	Read
M2	T1	Write
M2	T0	Read
M2	T0	Write
M3	T1	Read
M3	T1	Write
M3	T0	Read
M3	T0	Write

```
property p_m1t1r;
// master1 reading from target 1
  @(posedge clk)
  $fell (frame1 && irdy1) |->
```

```
        ##3 ($fell (trdy[1])) ##3 !data[8];
endproperty

property p_m1t1w;
// master 1 writing to target 1
  @(posedge clk)
  $fell (frame1 && irdy1) |->
        ##3 ($fell (trdy[1])) ##3 data[8];
endproperty

property p_m1t0r;
// master 1 reading from target 0
  @(posedge clk)
  $fell (frame1 && irdy1) |->
        ##3 ($fell (trdy[0])) ##3 !data[8];
endproperty

property p_m1t0w;
// master 1 writing to target 0
  @(posedge clk)
  $fell(frame1 && irdy1) |->
        ##3 ($fell(trdy[0])) ##3 data[8];
endproperty

property p_m2t1r;
// master 2 reading from target 1
  @(posedge clk)
  $fell (frame2 && irdy2) |->
        ##3 ($fell(trdy[1])) ##3 !data[8];
endproperty

property p_m2t1w;
// master 2 writing to target 1
  @(posedge clk)
  $fell (frame2 && irdy2) |->
        ##3 ($fell (trdy[1])) ##3 data[8];
endproperty

property p_m2t0r;
// master 2 reading from target 0
  @(posedge clk)
  $fell (frame2 && irdy2) |->
        ##3 ($felltrdy[0])) ##3 !data[8];
```

```
endproperty

property p_m2t0w;
// master 2 writing to target 0
 @(posedge clk)
 $fell (frame2 && irdy2) |->
       ##3 ($fell (trdy[0])) ##3 data[8];
endproperty

property p_m3t1r;
// master 3 reading from target 1
 @(posedge clk)
 $fell (frame3 && irdy3) |->
       ##3 ($fell (trdy[1])) ##3 !data[8];
endproperty

property p_m3t1w;
// master 3 writing to target 1
 @(posedge clk)
 $fell (frame3 && irdy3) |->
       ##3 ($fell (trdy[1])) ##3 data[8];
endproperty

property p_m3t0r;
// master 3 reading from target 0
 @(posedge clk)
 $fell (frame3 && irdy3) |->
       ##3 ($fell (trdy[0])) ##3 !data[8];
endproperty

property p_m3t0w;
// master 3 writing to target 0
 @(posedge clk)
 $fell (frame3 && irdy3) |->
       ##3 ($fell (trdy[0])) ##3 data[8];
endproperty
```

Each property should have a cover statement associated with it as shown below. The same technique used in Section 2.4.1.1 is used to keep count of the number of occurrences of the scenario.

```
c_m1t1r: cover property(p_m1t1r)
   begin
```

```
      m1_t1_r++;
      if(m1_t1_r == 3) m1_t1_r_flag = 1'b1;
    end

  c_m1t1w: cover property(p_m1t1w)
    begin
      m1_t1_w++;
      if(m1_t1_w == 3) m1_t1_w_flag = 1'b1;
    end

  c_m1t0r: cover property(p_m1t0r)
    begin
      m1_t0_r++;
      if(m1_t0_r == 3) m1_t0_r_flag = 1'b1;
    end

  c_m1t0w: cover property(p_m1t0w)
    begin
      m1_t0_w++;
      if(m1_t0_w == 3) m1_t0_w_flag = 1'b1;
    end

  c_m2t1r: cover property(p_m2t1r)
    begin
      m2_t1_r++;
      if(m2_t1_r == 3) m2_t1_r_flag = 1'b1;
    end

  c_m2t1w: cover property(p_m2t1w)
    begin
      m2_t1_w++;
      if(m2_t1_w == 3) m2_t1_w_flag = 1'b1;
    end

  c_m2t0r: cover property(p_m2t0r)
    begin
      m2_t0_r++;
      if(m2_t0_r == 3) m2_t0_r_flag = 1'b1;
    end

  c_m2t0w: cover property(p_m2t0w)
    begin
      m2_t0_w++;
```

```
      if(m2_t0_w == 3) m2_t0_w_flag = 1'b1;
    end

  c_m3t1r: cover property(p_m3t1r)
    begin
      m3_t1_r++;
      if(m3_t1_r == 3) m3_t1_r_flag = 1'b1;
    end

  c_m3t1w: cover property(p_m3t1w)
    begin
      m3_t1_w++;
      if(m3_t1_w == 3) m3_t1_w_flag = 1'b1;
    end

  c_m3t0r: cover property(p_m3t0r)
    begin
      m3_t0_r++;
      if(m3_t0_r == 3) m3_t0_r_flag = 1'b1;
    end

  c_m3t0w: cover property(p_m3t0w)
    begin
      m3_t0_w++;
      if(m3_t0_w == 3) m3_t0_w_flag = 1'b1;
    end
```

This coverage information from both Sections 2.4.1.1 and 2.4.1.2 can be used effectively to control the simulation environment. With the piece of code shown below, there are two ways to terminate a simulation:

1. Run a pre-determined number of transactions randomly and exit.
2. If all possible request scenarios are covered three times and if all possible "master to target" transactions are covered three times, then exit the simulation.

Whichever occurs first will terminate the simulation.

```
always@(posedge clk)
begin

if(creq1_flag && creq2_flag && creq3_flag &&
creq12_flag && creq23_flag && creq31_flag &&
```

```
creq123_flag && m1_t1_r_flag && m1_t1_w_flag &&
m1_t0_r_flag && m1_t0_w_flag && m2_t1_r_flag &&
m2_t1_w_flag && m2_t0_r_flag && m2_t0_w_flag &&
m3_t1_r_flag && m3_t1_w_flag && m3_t0_r_flag &&
m3_t0_w_flag)

begin

  $display("FC: All possible request scenarios
  covered 3 times\n");

  $display("FC: All possible transactions covered
  3 times\n");

  $finish();

end
end
```

2.4.1.3 Advanced coverage options

There is another data point that can be used to measure the functional coverage of the system.

"Every target memory location should be written to and read from at least once by each master"

This information requires exhaustive testing. Every address space in the target device should be monitored for usage by each master device. SVA is not always the choice for performing functional coverage. ***Functional coverage that involves exhaustive test plan coverage points can be done more efficiently with a testbench language that supports object oriented programming constructs.*** Such exhaustive functional coverage points should be used while running long regression runs.

2.4.2 Functional coverage summary

Functional coverage measurement guarantees testing of all required scenarios. The measure can be used effectively for controlling simulation environments. One method is to terminate simulation upon achieving the functional coverage goals. In the sample system, the following results were observed:

- Default number of random transactions set in the testbench was 500.
- Terminating the simulation based on the request scenarios shown in Section 2.4.1.1 took 46 transactions.
- Terminating the simulation based on the request scenarios shown in Section 2.4.1.1 and the "master to target" transactions shown in Section 2.4.1.2 took 63 transactions.

The functional coverage data obtained can also be used to re-direct the testbench dynamically. In random testbenches, constraints are used to control the type of transactions generated. These constraints are assigned certain weights for the random distribution in the beginning of a simulation. Based on the functional coverage information obtained during the simulation, these weights can be adjusted dynamically to achieve the functional coverage goal quickly.

2.5 SVA for transaction log creation

SVA can be used to create excellent log files. The SVA checkers snoop for any design property violation during simulation. The same checkers can be called monitors if they log the information that they are snooping. In a complex system, it really helps to create a chronological log of the transactions. In our sample system, creating a log of all the read and write transactions, between whom these happened and at what time will be a great debugging asset.

SVA has the option to use a lot of the Verilog like capabilities within the scope of the checker. The action block of each checker or cover statement can be used efficiently to create log files. While displaying information upon the success of an assert or a cover statement is one way to create log files, another way is to call a task or a function. The calling of a task or a function expands the capabilities of the SVA checker. Apart from displaying information within the task, data checking can also be done effectively. The following code shows how a chronological transaction log is created for the sample system.

```
// open a file to document transactions

integer h_mt;
initial
begin
```

```
  h_mt = $fopen("mt.dat");
end

// calling task for documentation

`ifdef slv_doc

c_m1t1w_doc:
   cover property(p_m1t1w) master_xaction(1,1);
c_m1t1r_doc:
   cover property(p_m1t1r) master_xaction(1,1);
c_m1t2w_doc:
   cover property(p_m1t0w) master_xaction(1,0);
c_m1t2r_doc:
   cover property(p_m1t0r) master_xaction(1,0);
c_m2t1w_doc:
   cover property(p_m2t1w) master_xaction(2,1);
c_m2t1r_doc:
   cover property(p_m2t1r) master_xaction(2,1);
c_m2t2w_doc:
   cover property(p_m2t0w) master_xaction(2,0);
c_m2t2r_doc:
   cover property(p_m2t0r) master_xaction(2,0);
c_m3t1w_doc:
   cover property(p_m3t1w) master_xaction(3,1);
c_m3t1r_doc:
   cover property(p_m3t1r) master_xaction(3,1);
c_m3t2w_doc:
   cover property(p_m3t0w) master_xaction(3,0);
c_m3t2r_doc:
   cover property(p_m3t0r) master_xaction(3,0);

`endif

task master_xaction(
  input int m_identity, input int t_identity);

integer i;

begin

if(data[8])
begin
```

```
for(i=0; i<8; i++)
begin

$fwrite(h_mt,"WRITE:
Master %0d writing to Target %0d = %0d at
%0t\n",m_identity, t_identity, data[7:0],
$time);

@(posedge clk);
end
end

if(!data[8])
begin
@(posedge clk);
for(i=0; i<8; i++)
begin
$fwrite(h_mt,"READ:
Master %0d reading from Target %0d = %0d at
%0t\n", m_identity, t_identity, datao, $time);

@(posedge clk);
end
end

end

endtask
```

The properties defined for functional coverage in Section 2.4.1.2 are reused for creating transaction logs. If the cover statement succeeds, a task called "master_xaction" is called. The task expects two input arguments, one identifying the master and the other identifying the target device. By sending these arguments, a generic task can be written to log the transactions accurately.

The transactions are logged into a separate file called "mt.dat." A $fopen statement is used to open this file at the beginning of the simulation. Once the task is called, the task executes either the read block of the code or the write block of the code. Since our sample system does burst read or write in sets of 8 bytes, a "for" loop is used within the task. The loop goes around eight times and each time the relevant read or write data is logged into the

file "mt.dat" using a $fwrite statement. A part of the log created for the sample system using this code is shown below.

```
WRITE: Master 1 writing to Target 1 = 72 at 775
WRITE: Master 1 writing to Target 1 = 77 at 825
WRITE: Master 1 writing to Target 1 = 95 at 875
WRITE: Master 1 writing to Target 1 = 37 at 925
WRITE: Master 1 writing to Target 1 = 216 at 975
WRITE: Master 1 writing to Target 1 = 184 at 1025
WRITE: Master 1 writing to Target 1 = 198 at 1075
WRITE: Master 1 writing to Target 1 = 182 at 1125
READ: Master 3 reading from Target 1 = 72 at 1725
READ: Master 3 reading from Target 1 = 77 at 1775
READ: Master 3 reading from Target 1 = 95 at 1825
READ: Master 3 reading from Target 1 = 37 at 1875
READ:Master 3 reading from Target 1 = 216 at 1925
READ:Master 3 reading from Target 1 = 184 at 1975
READ:Master 3 reading from Target 1 = 198 at 2025
READ:Master 3 reading from Target 1 = 182 at 2075
```

The transaction logs can be made a lot more fancy and debug friendly depending on the user's application. Note that this code is included within the `ifdef - `endif block. This kind of a detailed transaction log might not be needed during long regressions and hence should have the provision to be included conditionally.

2.6 SVA for FPGA Prototyping

A variety of advanced verification methodologies exist today that can help find bugs quickly. Constrained random testbenches and assertions are an important piece in these methodologies. It is very common to write thousands of tests to make sure that all possible functionality has been tested correctly. While most of the bugs are found in the RTL verification, it is still very common to find functional bugs during the verification of implemented gates. Simulating gates has always been a performance bottleneck and will always be. Running all the tests developed during RTL verification on gates is not very practical. Gate level simulation is extremely slow and more and more verification teams are depending on other verification methodologies such as formal verification, FPGA prototyping, etc. as shown in Figure 2-14. By running the verification on the actual silicon, the verification process can be accelerated significantly. This allows running the regression suites developed for RTL exhaustively on actual silicon.

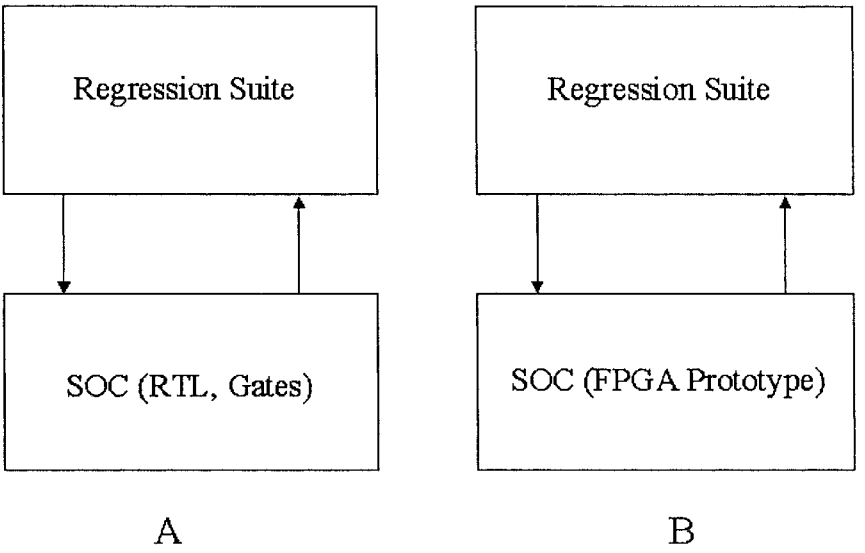

Figure 2-14. FPGA Prototyping

One major challenge in running tests on actual silicon prototype is debugging. SVA can help in this area significantly. By synthesizing the checkers along with the design, the debug process can be made a little easier. The checkers are written against the functional specification and having them monitor the design in real silicon adds great value. The design needs to be altered slightly to accommodate these assertions. If an assertion fails, it has to be notified to the external world using an output port. The output ports can be updated with the results, using the action block of the assertions. In most real-time testing, breakpoints can be set on these output ports and upon a failure on one of these debug ports, the verification can be stopped for further analysis. The master device used in the sample system is shown in Figure 2-2. This contains only the default ports relevant to the design. The sample Verilog code for the master device is shown below.

```
module master (ask_for_it, clk, req, gnt, frame,
irdy, trdy, data_c, r_sel, data_o);

input clk, gnt, ask_for_it;
input [1:0] trdy;
output req, frame, irdy, r_sel;
```

```
output [8:0] data_c;
input [7:0] data_o;

parameter master_sva = 1'b1;
parameter master_sva_severity = 1'b1;

// functional description of master

// Block level SVA checks

endmodule
```

The block level assertions should be made part of the design to help in FPGA prototyping. Each block level assertion should be associated with a debug output port. The debug output port should be asserted if the assertion fails. The following code description shows how this can be achieved.

```
module master (ask_for_it, clk, req, gnt, frame,
irdy,     trdy,     data_c,     r_sel,     data_o,
a_master_start1_flag, a_master_start2_flag,
a_master_stop1_flag, a_master_stop2_flag,
a_master_data1_flag, a_master_data2_flag,
a_master_datao1_flag, a_master_datao2_flag);

input clk, gnt, ask_for_it;
input [1:0] trdy;
output req, frame, irdy, r_sel;
output [8:0] data_c;
input [7:0] data_o;

// debug pins for FPGA prototyping
output a_master_start1_flag;
output a_master_start2_flag;
output a_master_stop1_flag;
output a_master_stop2_flag;
output a_master_data1_flag;
output a_master_data2_flag;
output a_master_datao1_flag;
output a_master_datao2_flag;

parameter master_sva = 1'b1;
parameter master_sva_severity = 1'b1;
```

```
// functional description of master

// Block level checks for prototype debugging

`ifdef master_debug

d_a_master_start1:
   assert property(p_master_start1)
     else
       a_master_start1_flag = 1'b1;
d_a_master_start2:
     assert property(p_master_start2)
     else
       a_master_start2_flag = 1'b1;
d_a_master_stop1:
     assert property(p_master_stop1)
     else
       a_master_stop1_flag = 1'b1;
d_a_master_stop2:
     assert property(p_master_stop2)
     else
       a_master_stop2_flag = 1'b1;
d_a_master_data1:
     assert property(p_master_data1)
     else
       a_master_data1_flag = 1'b1;
d_a_master_data2:
     assert property(p_master_data2)
     else
       a_master_data2_flag = 1'b1;
d_a_master_datao1:
     assert property(p_master_datao1)
     else
       a_master_datao1_flag = 1'b1;
d_a_master_datao2:
     assert property(p_master_datao2)
     else
       a_master_datao2_flag = 1'b1;

`endif

endmodule
```

Note that the respective output port flags will be asserted upon a failure. Since these assertions are concurrent, they will look for a valid start on every clock edge. If the silicon testing mechanism does not provide a way to set breakpoints on an assertion failure, then it is required that the failure be latched. Otherwise, the failure notification can be lost if the assertion succeeds in future clock cycles.

2.7 Summary on SVA simulation methodologies

- The addition of SVA to testbench environment makes dynamic simulation more productive.
- The designers are very familiar with the internal functionality of the design and hence, they should in-line SVA checkers in their respective design blocks.
- The verification engineer, who integrates and verifies the system, should add system level assertions that thoroughly verify the interface protocol.
- The verification engineer should be able to control/configure the block level assertions from his verification environment (He should be able to turn the assertions on and off on a need basis).
- Functional coverage metrics can be collected with little effort using SVA. This information should be used effectively to create reactive testbenches.
- SVA can be used to create informative log files since they are monitoring the design protocols throughout the simulation.
- By writing SVA checkers that follow synthesis coding guidelines, they can be made part of the net-list and used to debug prototyping/emulation failures.