Chapter 3

SVA FOR FINITE STATE MACHINES

FSM is the main control block in any design. It helps the design progress from state to state in an orderly manner by generating the respective control signals. Another way to generate control signals is by combining counters and glue logic. But it lacks good design structure and is also difficult to debug. An FSM provides great hardware infrastructure for control signals and also debugging capabilities since each state of the design is usually well defined.

There are two types of FSMs:

Moore State machine – The Moore FSM outputs are the function of the present state only.

Mealy State machine – One or more of the Mealy FSM outputs are a function of the present state and one or more of the inputs.

Different types of coding styles are used to describe the states of an FSM. The most popular coding style is the one-hot coding, wherein a one-bit register represents each state. This proves to be the fastest architecture. If the FSM has too many states, then one-hot coding will produce a rather big hardware. In these cases binary encoding is preferred. Another kind of encoding used commonly to describe an FSM is the gray coding.

An FSM controls the functionality of the entire design and hence should be verified thoroughly. The most common type of check is to make sure that the state transitions are occurring correctly without violating any timing requirement. SVA can be used effectively to do such checks.

3.1 Sample Design – FSM1

In this section, we analyze a simple linear FSM, which is more like a shift counter. The FSM produces control signals for the design in a linear sequential fashion and hence can be verified easily with SVA checks.

3.1.1 Functional description of FSM1

There are 16 states in the FSM. They are coded as follows:

IDLE = 16'd1
GEN_BLK_ADDR = 16'd2
WAIT6 = 16'd4
NEXT_BLK = 16'd8
WAIT0 = 16'd16
CNT1 = 16'd32
WAIT1 = 16'd64
CNT2 = 16'd128,
WAIT2 = 16'd256
CNT3 = 16'd512
WAIT3 = 16'd1024
CNT4 = 16'd2048
WAIT4 = 16'd4096
CNT5 = 16'd8192
WAIT5 = 16'd16384
CNT6 = 16'd32768

The FSM is coded with a one-hot coding style. Figure 3-1 shows the bubble diagram of FSM1:

- The FSM moves to the IDLE state upon reset and waits there for a valid "get_data" signal.
- Once a valid "get_data" signal is obtained, the FSM moves to the GEN_BLK_ADDR state. The FSM stays in this state until it finishes generating 64 read addresses (an internal counter keeps track of 64 clock cycles).
- After 64 clock cycles, it moves to the WAIT0 state. From this point, the FSM keeps moving to the next state on every clock cycle.
- The CNT* states are the ones where the output control signals for the rest of the design are generated.

- The WAIT* states are used to create a 2 cycle gap between the generation of the control signals (latch_en, dp1_en, dp2_en, dp3_en, dp4_en, wr).
- Once the FSM moves to the NEXT_BLK state it has to decide which way to go.

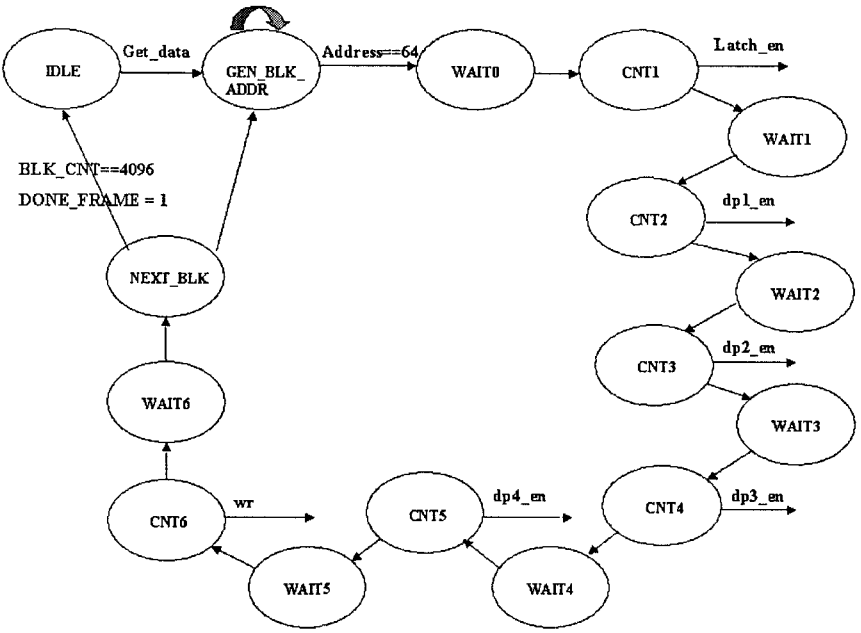

Figure 3-1. Bubble diagram for FSM1

- If the internal register "blk_cnt" has reached the value of 4096, the FSM goes to the IDLE state. This indicates that the entire data frame has been processed and the design is waiting for a new frame. When a new "get_data" signal arrives, the FSM goes through the same state transitions again starting from GEN_BLK_ADDR.
- While in state NEXT_BLK, if the internal register "blk_cnt" has not reached the value of 4096, the FSM will go back to the GEN_BLK_ADDR state. It waits for the generation of 64 new addresses and then moves over to the CNT* states to generate the control signals again.

Example 3.1 FSM1 sample code

```
module fsm (get_data, reset_, clk, rd, rd_addr,
data, done_frame, latch_en, sipo_en, dp1_en,
dp2_en, dp3_en, dp4_en, wr);

  input get_data;
  input reset_;
  input clk;
  input [7:0] data;

  output rd;
  output logic sipo_en, latch_en;
  output logic dp1_en, dp2_en, dp3_en, dp4_en, wr;
  output logic done_frame;
  output [17:0] rd_addr;

  logic [5:0] addr_cnt;
  logic [11:0] blk_cnt;
  logic [3:0]  pipeline_cnt;
  logic rd;
  logic [17:0] rd_addr;
 logic enable_cnt, enable_dly_cnt, enable_blk_cnt;

  assign done_frame = (blk_cnt == 4095);
  assign sipo_en = rd;

   enum bit[15:0] {IDLE = 16'd1,
       GEN_BLK_ADDR = 16'd2,
       DLY = 16'd4,
       NEXT_BLK = 16'd8,
       WAIT0 = 16'd16,
       CNT1 = 16'd32,
       WAIT1 = 16'd64,
       CNT2 = 16'd128,
       WAIT2 = 16'd256,
       CNT3 = 16'd512,
       WAIT3 = 16'd1024,
       CNT4 = 16'd2048,
       WAIT4 = 16'd4096,
       CNT5 = 16'd8192,
       WAIT5 = 16'd16384,
       CNT6 = 16'd32768} n_state, c_state;
```

```
// assign the different control signals

assign latch_en = (c_state == CNT1);
assign dp1_en = (c_state == CNT2);
assign dp2_en = (c_state == CNT3);
assign dp3_en = (c_state == CNT4);
assign dp4_en = (c_state == CNT5);
assign wr = (c_state == CNT6);

// 64bit counter to generate read address

always_ff @(posedge clk)
if (!reset_ || !enable_cnt)
  addr_cnt <= 0;
else if (enable_cnt)
  addr_cnt <= addr_cnt + 1;
else
  addr_cnt <= addr_cnt;

// 4096 bit counter

always_ff @(posedge clk)
if (!reset_)
  blk_cnt <= 0;
else if ((c_state == NEXT_BLK) && enable_blk_cnt)
  blk_cnt <= blk_cnt + 1;
else
  blk_cnt <= blk_cnt;

always_ff @(posedge clk)
  if (!reset_)
    c_state <= IDLE;
     else
    c_state <= n_state;

always @(*)
  begin
  rd <= 0;
  enable_cnt <= 0;
  //enable_dly_cnt <= 0;
  case (c_state)
    IDLE:  begin
        enable_blk_cnt <= 0;
```

```
        if(get_data)
            n_state <= GEN_BLK_ADDR;
        else
            n_state <= IDLE;
          end

    GEN_BLK_ADDR: begin
          enable_cnt <= 1;
          rd <= 1;
          rd_addr <= {blk_cnt, addr_cnt};
          if (addr_cnt == 63) begin
              //enable_dly_cnt <= 1;
              n_state <= WAIT0;
            end
          else begin
              n_state <= GEN_BLK_ADDR;
              //pipeline_cnt <= 0;
            end
          end

    WAIT0: n_state <= CNT1;
    CNT1: n_state <= WAIT1;
    WAIT1: n_state <= CNT2;
    CNT2: n_state <= WAIT2;
    WAIT2: n_state <= CNT3;
    CNT3: n_state <= WAIT3;
    WAIT3: n_state <= CNT4;
    CNT4: n_state <= WAIT4;
    WAIT4: n_state <= CNT5;
    CNT5: n_state <= WAIT5;
    WAIT5: n_state <= CNT6;
    CNT6: n_state <= DLY;

    DLY: begin
        enable_blk_cnt <= 1;
        n_state <= NEXT_BLK;
      end


NEXT_BLK: begin
          enable_blk_cnt<=1;
          if (blk_cnt == 4095)
              n_state <= IDLE;
```

```
            else
                n_state <= GEN_BLK_ADDR;
              end
    endcase
  end

  endmodule
```

The state transition from GEN_BLK_ADDR to the CNT*/WAIT* states is shown in Figure 3-2.

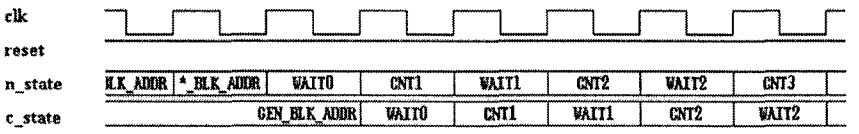

Figure 3-2. Waveform A for FSM1

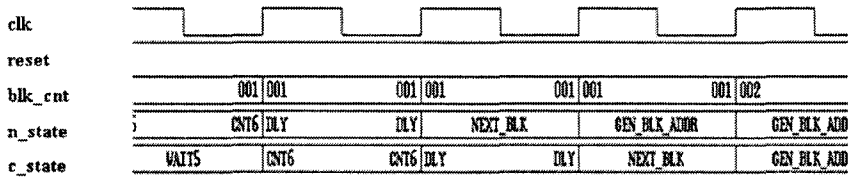

Figure 3-3. Waveform B for FSM1

Figure 3-3 shows that the FSM will loop back to GEN_BLK_ADDR state from the NEXT_BLK state if the register "blk_cnt" has not reached the value of 4096. When "blk_cnt" reaches the value of 4096, the FSM will break the loop from NEXT_BLK state and go to IDLE state.

3.1.2 SVA Checkers for FSM1

To verify FSM1 thoroughly, the following checks need to be done.

FSM1_chk1: FSM1 will always stay one-hot irrespective of the input conditions.

FSM1 is based on one-hot coding and hence should always have only one state bit asserted. If not, the FSM is not truly one-hot and the control signals might not be generated as expected. This can be tested by using any one of the built-in tasks, namely, **$countones** or **$onehot** defined in the SVA language.

```
property p_onehot;
  @(posedge clk) (reset_) |->
          ($countones(n_state) == 1);
endproperty

a_onehot: assert property(p_onehot);
c_onehot: cover property(p_onehot);
```

FSM1_chk2: If the current state is "IDLE" and if "get_data" is asserted, then the next state is "GEN_BLK_ADDR," and 64 cycles later the next state should be "WAIT0."

The FSM starts the transition, based on the IDLE state and the "get_data" signal. Once the FSM reaches GEN_BLK_ADDR, it has to stay there for 64 clock cycles.

```
sequence s_trans1;
     (c_state == IDLE) ##1
     ((c_state   ==   GEN_BLK_ADDR)   [*64])   ##1
     (c_state == WAIT0);
endsequence

property p_trans;
  @(posedge clk)
(reset_ && $rose(get_data)) |->
           (reset_) throughout (s_trans1);
endproperty

a_trans: assert property (p_trans);
c_trans: assert property (p_trans)
```

The sequence "s_trans1" verifies that, if the current state of FSM1 is IDLE, then one cycle later it will transition to the GEN_BLK_ADDR. The FSM will stay in the state GEN_BLK_ADDR for 64 cycles (verified by

using the repeat (*) operator) and one cycle later will move to the WAIT0 state. It is required that the reset is inactive throughout this property.

Figure 3-4 shows the results of "a_trans" property. The checker becomes active when there is a rising edge on the "get_data" signal and a match on the success is shown at the same point in the waveform. Though the checker stays active until reaching the WAIT0 state, the success is shown only at the starting point of the checker. The checker looks for a rising edge of "get_data" signal on every positive edge of the clock. If there isn't one, then the checker is assumed to succeed by default. This is a ***vacuous success*** as discussed in Chapter 1.

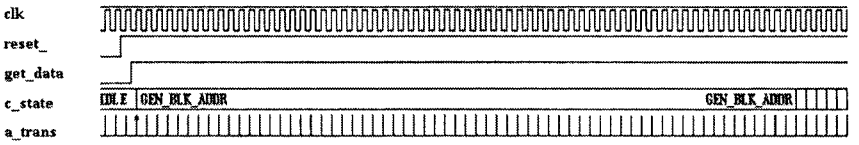

Figure 3-4. Waveform for FSM1_chk2

FSM1_chk3: If the current state is "WAIT0," then the FSM will transition states in a sequential manner from one state to another after one clock cycle each, unless the FSM is reset.

There is a wait state between every CNT* state. Hence, the FSM takes 2 clock cycles to move from one CNT* state to another CNT* state. The FSM moves in a linear fashion from CNT1 state to CNT6 state. The only possible path is as follows:

CNT1 -> CNT2 -> CNT3 -> CNT4 -> CNT5 -> CNT6

```
sequence s_trans3;
  ##1 (c_state == CNT1) ##2 (c_state == CNT2)
  ##2 (c_state == CNT3) ##2 (c_state == CNT4) ##2
  (c_state == CNT5) ##2 (c_state == CNT6);
endsequence

property p_linear_trans;
@(posedge clk)
((reset_) && (c_state == WAIT0)
&& ($past(c_state)== GEN_BLK_ADDR)) |->
```

```
s_trans3;
endproperty

a_linear_trans: assert property (p_linear_trans);
c_linear_trans: cover property (p_linear_trans);
```

Sequence "s_trans3" verifies that, if the FSM is currently in WAIT0 state and if it was in GEN_BLK_ADDR state in the previous cycle, then the FSM moves to CNT1 state after one cycle and WAIT1 state one cycle after that and so on up to reaching the state CNT6. Figure 3-5 shows the simulation results of the property p_linear_trans.

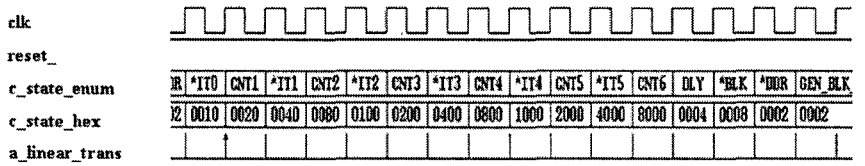

Figure 3-5. Waveform for FSM1_chk3

FSM1_chk4: Make sure that FSM1 is exercised such that it goes to both IDLE and GEN_BLK_ADDR at least once from the NEXT_BLK state.

This check acts as a functional coverage piece making sure that all paths of the FSM transitions are exercised once by the input test vectors.

```
sequence s_trans2;
  ##63 (c_state == GEN_BLK_ADDR) ##1
  (c_state == WAIT0);
endsequence

property p_frame;
  @(posedge clk)
((reset_) && (c_state == GEN_BLK_ADDR) &&
(($past (c_state)== IDLE) ||
($past(c_state == NEXT_BLK)))) |->
                s_trans2 ##0 s_trans3;
endproperty

a_frame: assert property(p_frame) cnt++;
```

```
c_frame: cover property(p_frame);

property p_complete_frame;
  @(posedge clk)
((cnt == 16'd4095)&&reset_&&
(c_state==CNT6)) |->
                      done_frame;
endproperty

a_complete_frame:
    assert property(p_complete_frame)
    $display   ("A   complete   frame   has   been
    transferred \n");
```

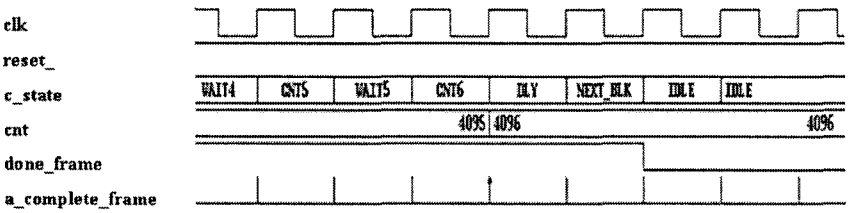

Figure 3-6. Waveform for FSM1_chk4

The property "p_frame" verifies the complete state transition of FSM1 starting from IDLE state. While this check is performed, a local variable "cnt" is incremented in the action block every time the property "a_frame" succeeds. When the value of the variable "cnt" reaches 4095, all blocks of data have been processed and the control signal "done_frame" is asserted. At the end of a complete frame testing, the action block of the check "a_complete_frame" can be used to display the results. Figure 3-6 shows the simulation results of the property p_complete_frame.

Two separate properties "p_frame_path1" and "p_frame_path2" are written to make sure that all possible paths of the FSM are covered during simulation.

```
property p_frame_path1;
  @(posedge clk)
  ((reset_) && (c_state == GEN_BLK_ADDR) &&
  ($past(c_state == NEXT_BLK))) |->
  s_trans2 ##0 s_trans3;
```

```
endproperty

c_frame_path1: cover property (p_frame_path1);

property p_frame_path2;
  @(posedge clk)
  ((reset_) && (c_state == GEN_BLK_ADDR) &&
  ($past(c_state == IDLE))) |->
      s_trans2 ##0 s_trans3;
endproperty

c_frame_path2: cover property (p_frame_path2);
```

3.2 Sample Design – FSM2

A slightly more complicated FSM is discussed in this section. The FSM discussed in Section 3.1 was linear and did not have many ways of getting to a particular state. FSM2 will have fewer states but there will be more ways of getting to a particular state. This presents a minor challenge in extracting the checks that need to be done.

3.2.1 Functional description of FSM2

FSM2 performs the role of an arbiter. At any given time, FSM2 can arbitrate between 3 master devices. Any or all of the master devices can request for the grant of the bus and the arbiter will decide who gets the bus based on a round robin fashion. Once the master acquires a grant, it uses the bus to do certain transactions. At the end of the transaction, the master lets the arbiter know and the bus is freed. Once the bus is free, all the masters can once again make a request for the bus if they have any pending transactions. The key concept is to make sure that the arbiter is not starving any of the masters.

The FSM has 7 possible states shown as follows:

IDLE = 7'b0000001
MASTER1 = 7'b0000010
IDLE1 = 7'b0000100
MASTER2 = 7'b0001000
IDLE2 = 7'b0010000
MASTER3 = 7'b0100000
IDLE3 = 7'b1000000

The FSM is coded with a one-hot coding style. Figure 3-7 shows the bubble diagram of FSM2.

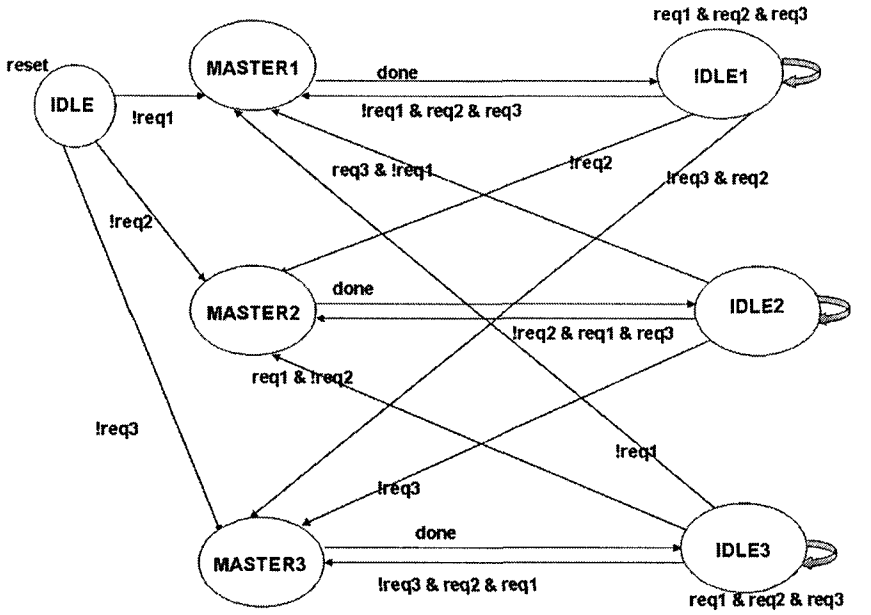

Figure 3-7. Bubble diagram for FSM2

- Upon reset the state machine moves to the IDLE state.
- While the state machine is in the IDLE state, it looks for a "req" from any of the master devices wanting to use the bus. The grant is given in a priority-encoded fashion. For example, when the FSM is in IDLE state, if all three master devices make a request then "master1" gets the bus.
- While the state machine is in the MASTER* state, it asserts the "gnt" signal of the respective master device.
- Once the master device is done with using the bus, it indicates this to the arbiter by asserting the "done" signal and this moves the FSM to the respective IDLE* state of that master device.
- When the FSM is in the IDLE1 state, it will look for "req" from the masters in the order of master2, master3 and then master1.
- When the FSM is in the IDLE2 state, it will look for "req" from the masters in the order of master3, master1 and then master2.

- When the FSM is in the IDLE3 state, it will look for "req" from the masters in the order of master1, master2 and then master3.

Example 3.2 FSM2 Sample code

```
module bus_arbiter(clk, reset, frame, irdy,
req1, req2, req3, gnt1, gnt2, gnt3);

input logic clk, reset, frame, irdy;
input logic req1, req2, req3;

output gnt1, gnt2, gnt3;

enum bit [6:0] {IDLE = 7'b0000001,
   MASTER1 = 7'b0000010,
   IDLE1 = 7'b0000100,
   MASTER2 = 7'b0001000,
   IDLE2 = 7'b0010000,
   MASTER3 = 7'b0100000,
   IDLE3 = 7'b1000000} next, state;

logic done, gnt1, gnt2, gnt3;

/* define glue signals */

assign done = frame && irdy;

/* state register code */

always@(posedge clk or negedge reset)
begin
  if(!reset)
    state <= IDLE;
  else
    state <= next;
end

/* next state combinational logic */

always@(*)
begin
  next = IDLE;
  case(state)
```

```
IDLE:
  if (req1 == 1'b0)
    next <= MASTER1;
  else if (req1 == 1'b1 & req2 == 1'b0)
    next <= MASTER2;
  else if (req3 == 1'b0 & req1 == 1'b1)
    next <= MASTER3;
  else
    next <= IDLE;

MASTER1:
  if(!done)
    next <= MASTER1;
  else
    next <= IDLE1;

IDLE1:

  if(req2 == 1'b0 )
    next <= MASTER2;
  else if (req3 == 1'b0 & req2 == 1'b1)
    next <= MASTER3;
  else if (req3 == 1'b1 & req1 == 1'b0 & req2 ==
  1'b1)
    next <= MASTER1;
  else
    next <= IDLE1;

MASTER2:
  if(!done)
    next <= MASTER2;
  else
    next <= IDLE2;

IDLE2:
  if (req3 == 1'b0)
    next <= MASTER3;
  else if (req3 == 1'b1 & req1 == 1'b0)
    next <= MASTER1;
  else if (req1 == 1'b1 & req2 == 1'b0)
    next <= MASTER2;
  else
```

```
      next <= IDLE2;
    MASTER3:
      if (!done)
        next <= MASTER3;
      else
        next <= IDLE3;

    IDLE3:
      if (req1 == 1'b0)
        next <= MASTER1;
      else if (req1 == 1'b1 & req2 == 1'b0)
        next <= MASTER2;
      else if (req2 == 1'b1 & req3 == 1'b0)
        next <= MASTER3;
      else
        next <= IDLE3;
    endcase

  end

  /* output generating statements */

  assign gnt1 = ((state == MASTER1)) ? 0 : 1;
  assign gnt2 = ((state == MASTER2)) ? 0 : 1;
  assign gnt3 = ((state == MASTER3)) ? 0 : 1;

  endmodule
```

Figure 3-8 shows a sample waveform for FSM2. For convenience, the state encoding is shown both in the enumerated value and hexadecimal value. The state value *1 means that the state is MASTER1, similarly, *2 for MASTER2 and *3 for MASTER3. At marker 1, both master2 and master3 make a request for the bus. The FSM is in IDLE3 state at this point and hence provides the grant to master2. At marker 2, the FSM is in IDLE2 state and both master1 and master3 request the bus. This time master3 gets the grant. The grant provided always depends on which IDLE* state the FSM is currently in.

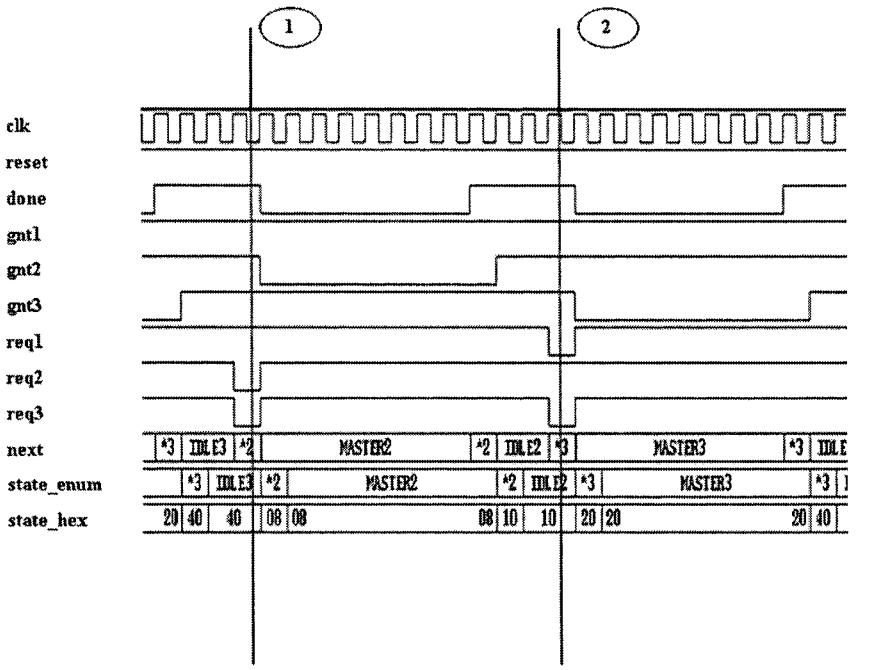

Figure 3-8. Waveform for FSM2

3.2.2 SVA Checkers for FSM2

For a state machine like this, since there are many ways to get to a specific state, the first thing to understand is what are the possible legal paths? This can be a very difficult process depending on the complexity of the state machine. As a first step, a matrix should be created with all states represented on both the x and y axis. Then the matrix should be filled with a "Yes" or "No" indicating whether it is possible to transition from that state in the x axis to a respective state in the y axis. Once we have a representation as described above, we can start categorizing the SVA checks.

- Based on the matrix, if a state to state transition is forbidden, then it should be verified using a SVA check.
- All possible legal state transitions should be covered by the testbench. This metric can be measured by using the "cover" statements on SVA properties. The same information can also be obtained from code coverage tools.

Based on the matrix analysis, as shown in Table 3-1, the following checks need to be written to verify FSM2 thoroughly.

Table 3-1. Matrix diagram for FSM2 state transition

	IDLE	M1	I1	M2	I2	M3	I3
IDLE	Y	Y	N	Y	N	Y	N
M1	Y	Y	Y	N	N	N	N
I1	Y	Y	Y	Y	N	Y	N
M2	Y	N	N	Y	Y	N	N
I2	Y	Y	N	Y	Y	Y	N
M3	Y	N	N	N	N	Y	Y
I3	Y	Y	N	Y	N	Y	Y

FSM2_chk1: FSM2 should always behave as a one-hot state machine.

```
property p_fsm2_encoding;
  @(posedge clk) $onehot(state);
endproperty

a_fsm2_encoding:
            assert property (p_fsm2_encoding);
c_fsm2_encoding:
            cover property (p_fsm2_encoding);
```

The built-in function **$onehot** can be used to make sure that only one bit of the state register is high at any given time, thus proving that the FSM always stays one-hot.

FSM2_chk2: From IDLE state the FSM cannot go to IDLE1, IDLE2 or IDLE3 states.

```
property p_forbid_trans1;
@(posedge clk)
(((state == IDLE1) || (state == IDLE2) ||
(state == IDLE3)) && reset) |->
         $past ((state == IDLE) == 0);
endproperty

a_forbid_trans1:assert property(p_forbid_trans1);
```

The property "p_forbid_trans1" verifies that, if the current state is IDLE1, IDLE2 or IDLE3, then the state of the FSM in the previous cycle cannot be IDLE.

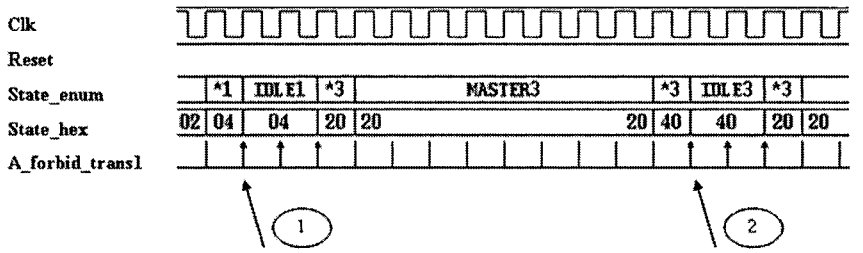

Figure 3-9. Waveform for FSM2_chk2

Figure 3-9 shows the results of the check "a_forbid_trans1." Marker 1 shows a point where the FSM is currently in IDLE1 state. The property passes here since the FSM was in MASTER1 state in the previous cycle and not in IDLE state. Similarly, marker 2 shows a point where the FSM is in IDLE3 state. The property passes here also since the FSM was in MASTER3 state in the previous clock cycle and not IDLE state.

FSM2_chk3:

From MASTER1 state the FSM cannot go to other MASTER states or IDLE2 or IDLE3.

From MASTER2 state the FSM cannot go to other MASTER states or IDLE1 or IDLE2.

From MASTER3 state the FSM cannot go to other MASTER states or IDLE1 or IDLE2.

This check makes sure that, if the FSM is in a certain MASTER* state, then the next transition will always be to the IDLE* state specific to the master state. In other words, if the FSM is currently in MASTER1 state, then it has to transition to only IDLE1 state next assuming the FSM is not reset. If it transitions to any other state, it is a violation. Similarly, MASTER2 should transition to IDLE2 and MASTER3 to IDLE3 state respectively. Figure 3-10 shows the result of the check "a_forbid_trans2a."

```
property p_forbid_trans2a;
  @(posedge clk)
  (((state == IDLE2) || (state == IDLE3) ||
  (state == MASTER2) || (state == MASTER3))
  && reset) |->
             $past ((state == MASTER1) == 0);
endproperty

a_forbid_trans2a:
          assert property(p_forbid_trans2a);
c_forbid_trans2a:
          cover property(p_forbid_trans2a);

property p_forbid_trans2b;
  @(posedge clk)
  (((state == IDLE1) || (state == IDLE3) ||
  (state == MASTER1) || (state == MASTER3))
  && reset) |->
             $past ((state == MASTER2) == 0);
endproperty

a_forbid_trans2b:
           assert property(p_forbid_trans2b);
c_forbid_trans2b:
           cover property(p_forbid_trans2b);

property p_forbid_trans2c;
  @(posedge clk)
  (((state == IDLE2) || (state == IDLE1) ||
  (state == MASTER2) || (state == MASTER1))
  && reset) |->
          $past (state == MASTER3) == 0);
endproperty

a_forbid_trans2c:
          assert property(p_forbid_trans2c);
c_forbid_trans2c:
          cover property(p_forbid_trans2c);
```

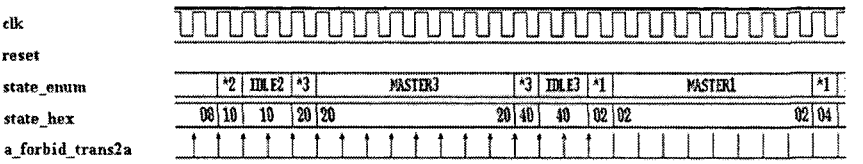

Figure 3-10. Waveform for FSM2_chk3

FSM2_chk4:
From IDLE1 state the FSM cannot go to IDLE2 or IDLE3 states.
From IDLE2 state the FSM cannot go to IDLE3 or IDLE1 states.
From IDLE3 state the FSM cannot go to IDLE2 or IDLE1 states.

This check makes sure that the FSM will always transition to a MASTER* state from an IDLE* state assuming that the FSM is not reset in between. If the FSM transitions from one IDLE* state to another IDLE* state, it is a violation. Figure 3-11 shows the result of the check "p_forbid_trans3a."

```
property p_forbid_trans3a;
  @(posedge clk)
  (((state == IDLE2) || (state == IDLE3))
  && reset) |->
        $past (state== IDLE1) == 0);
endproperty

a_forbid_trans3a:
            assert property(p_forbid_trans3a);
c_forbid_trans3a:
            cover property(p_forbid_trans3a);

property p_forbid_trans3b;
  @(posedge clk)
  (((state == IDLE1) || (state == IDLE3))
  && reset) |->
          $past (state== IDLE2) == 0);
endproperty

a_forbid_trans3b:
            assert property(p_forbid_trans3b);
c_forbid_trans3b:
            cover property(p_forbid_trans3b);
```

```
property p_forbid_trans3c;
  @(posedge clk)
  (((state == IDLE1) || (state == IDLE2))
  && reset) |->
            $past (state== IDLE3) == 0);
endproperty

a_forbid_trans3c:
            assert property(p_forbid_trans3c);
c_forbid_trans3c:
            cover property(p_forbid_trans3c);
```

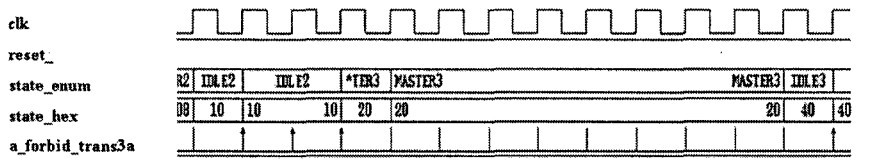

Figure 3-11. Waveform for FSM2_chk4

FSM2_chk5: There should be a grant for every request sent to the arbiter.

```
property p_req_gnt;
  @(posedge clk)
((!req1 || !req2 || !req3) && reset) |->
           ##1 (!gnt1 || !gnt2 || !gnt3);
endproperty

a_req_gnt: assert property(p_req_gnt);
c_req_gnt: cover property(p_req_gnt);
```

The property "p_req_gnt" verifies that, if any of the masters make a request for the bus, then within one cycle, any one of the "gnt" signal should be asserted. If the grant does not arrive in one cycle, it is a fatal error.

Figure 3-12 shows the result of the check 'a_req_gnt." Marker 1 shows the point in the waveform where master3 is requesting the bus and within one cycle the signal "gnt3" is asserted and hence the check passes. Similarly, marker 2 is pointing to a place where both master2

and master3 are requesting the bus and "gnt2" is asserted within one clock cycle and hence the check passes.

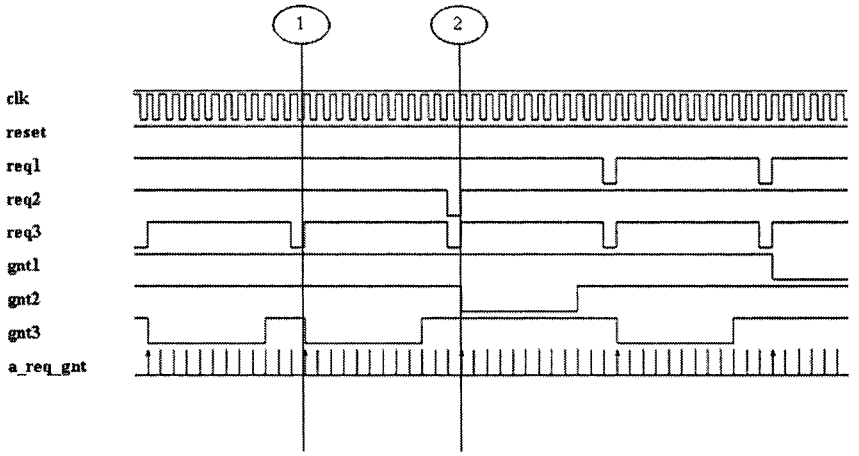

Figure 3-12. Waveform for FSM2_chk5

FSM2_chk6: Check for the fairness of the arbiter. Make sure that all the masters are getting equal number of grants.

```
property p_req_gnt_1;
  @(posedge clk) ((!req1 && reset)) |->
            ##1 !gnt1;
endproperty

c_req_gnt_1: cover property(p_req_gnt_1);

property p_req_gnt_2;
  @(posedge clk) ((!req2 && reset)) |->
            ##1 !gnt2;
endproperty

c_req_gnt_2: cover property(p_req_gnt_2);

property p_req_gnt_3;
  @(posedge clk) ((!req3 && reset)) |->
            ##1 !gnt3;
endproperty
```

```
c_req_gnt_3: cover property(p_req_gnt_3);

property p_req1;
  @(posedge clk)   ($fell(req1) && reset);
endproperty

c_req1: cover property(p_req1);

property p_req2;
  @(posedge clk)   ($fell(req2) && reset);
endproperty

c_req2: cover property(p_req2);

property p_req3;
  @(posedge clk)   ($fell(req3) && reset);
endproperty

c_req3: cover property(p_req3);
```

This check is performed to get the functional coverage information and also to validate the fairness of the arbiter. Three properties p_req_gnt1, p_req_gnt2 and p_req_gnt_3 are written to calculate how many times a master was actually able to get a grant. The next three properties, p_req1, p_req2 and p_req3 are written to calculate how many requests each master actually made. By using the cover statements on these properties, the simulation results are printed based on the number of matches. In a sample random test environment, the following results were produced

```
c_req_gnt_1, 10433 attempts, 288 match
c_req_gnt_2, 10433 attempts, 290 match
c_req_gnt_3, 10433 attempts, 291 match
c_req1, 10433 attempts, 481 match
c_req2, 10433 attempts, 474 match
c_req3, 10433 attempts, 505 match
```

Note that, each master requested the bus approximately 475 times and each one of the masters was granted the bus approximately 290 times. This shows that the arbiter is being very fair and is not starving any one master device.

3.2.3 FSM2 with a timing window protocol

In the previous section, FSM2 asserted the "gnt" signal one clock cycle after a request was made. In this section, the arbiter functionality is assumed such that it can take anywhere between 2 to 5 clock cycles to produce a grant. While most of the protocol extraction process still remains the same for the new arbiter, the timing needs to be adjusted in some of the checks.

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

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

a_req_gnt_w : assert property(p_req_gnt_w);
```

The property p_req_gnt_w looks for a rising edge on the "req" signal. The "req" signal is the OR output of all the three requests req1, req2 and req3 respectively. Once the pre-condition is true, it verifies that a rising edge occurs on the "gnt" signal within 2 to 5 clock cycles. The "gnt" signal is the OR output of all the three "gnt" signals gnt1, gnt2 and gnt3 respectively. Functional coverage statements similar to the ones shown in check FSM2_chk6 can be easily written for the new protocol based on the window of time. Figure 3-13 shows the results of the check "a_req_gnt_w."

The marker "1s" indicates the first valid request made to the arbiter. A valid "gnt" comes at marker "1e" after 5 clock cycles and hence the checker passes. The marker "2s" indicates the second valid request made to the arbiter. A valid "gnt" comes at the marker "2e" after 2 clock cycles and hence the checker passes.

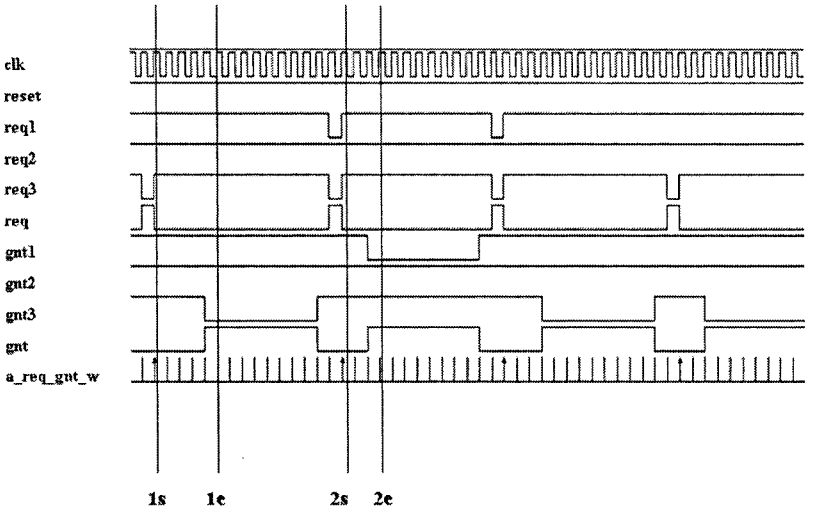

Figure 3-13. Waveform for window check

A key functional coverage data that a user might be interested in is the arbiter latency. The arbiter can take 2 to 5 cycles to respond to each one of the master devices. It is important to know if the average latency of the arbiter is the same for all three masters. We can use SVA cover statements to calculate the response time of the arbiter for each master.

```
genvar s;
generate

for (s=2; s<6; s++)
begin: generic
c_gnt_generic :
cover property(@(posedge clk) $rose(gnt) |->
        ($past(req,s) == 1'b1));
end
endgenerate
```

A generate statement can be used to create an array of cover statements. The objective is to find out the average response time of the arbiter. The built-in function **$past** is used to define a valid request and grant sequence. The variable "s" is used to loop around and create 4 separate cover properties for each possible latency values (2, 3, 4 and 5 clock cycles

respectively). The cover statement will increment the appropriate latency bin. Sample simulation results produced are shown below.

```
  tb.u4.u1.generic[2].c_gnt_generic, 5793 attempts,
112 match, 0 vacuous match
  tb.u4.u1.generic[3].c_gnt_generic, 5793 attempts,
113 match, 0 vacuous match
  tb.u4.u1.generic[4].c_gnt_generic, 5793 attempts,
101 match, 0 vacuous match
  tb.u4.u1.generic[5].c_gnt_generic, 5793 attempts,
104 match, 0 vacuous match
```

From these results, it is clear that the arbiter has an even distribution of latency. The previous example can be slightly modified to get latency information specific to each master. This way we will know if it takes longer to provide a grant to any specific master.

```
assign req_local[3:1] = ({req3, req2, req1});
assign gnt_local[3:1] = ({gnt3, gnt2, gnt1});

genvar j, k;
generate

for (j=2; j<6; j++)
begin: latency
for (k=1; k<4; k++)
begin: Master
c_gnt_o :
cover property(@(posedge clk) $fell(gnt_local[k])
        |-> ($past(req_local[k],j) == 1'b0));
end
end
endgenerate
```

Note that, a vector of the "gnt" signals called "gnt_local" and a vector of the "req" signals called "req_local" are defined. This allows one to loop through each master one at a time. Two loops are used, the outer loop "latency" defines the latency bins and the inner loop "Master" defines the master's identity. The property is active once a valid "gnt" signal is detected. A valid "req" for this specific 'gnt" is searched in the past anywhere between 2 and 5 clock cycles. Sample simulation results for such a coverage statement is shown below.

```
  tb.u4.u1.latency[2].Master[1].c_gnt_o,          5793
attempts, 41 match, 0 vacuous match
  tb.u4.u1.latency[2].Master[2].c_gnt_o,          5793
attempts, 37 match, 0 vacuous match
  tb.u4.u1.latency[2].Master[3].c_gnt_o,          5793
attempts, 34 match, 0 vacuous match
  tb.u4.u1.latency[3].Master[1].c_gnt_o,          5793
attempts, 39 match, 0 vacuous match
  tb.u4.u1.latency[3].Master[2].c_gnt_o,          5793
attempts, 34 match, 0 vacuous match
  tb.u4.u1.latency[3].Master[3].c_gnt_o,          5793
attempts, 40 match, 0 vacuous match
  tb.u4.u1.latency[4].Master[1].c_gnt_o,          5793
attempts, 27 match, 0 vacuous match
  tb.u4.u1.latency[4].Master[2].c_gnt_o,          5793
attempts, 36 match, 0 vacuous match
  tb.u4.u1.latency[4].Master[3].c_gnt_o,          5793
attempts, 38 match, 0 vacuous match
  tb.u4.u1.latency[5].Master[1].c_gnt_o,          5793
attempts, 34 match, 0 vacuous match
  tb.u4.u1.latency[5].Master[2].c_gnt_o,          5793
attempts, 29 match, 0 vacuous match
  tb.u4.u1.latency[5].Master[3].c_gnt_o,          5793
attempts, 41 match, 0 vacuous match
```

3.3 Summary on SVA for FSM

- FSMs are an integral part of any design and they need to be verified thoroughly.
- Every forbidden transition should be checked using SVA. If a forbidden transition occurs, it should be flagged as a fatal error.
- The testbench must cover all possible legal transitions. Functional coverage information should be used wisely to build a reactive simulation environment.