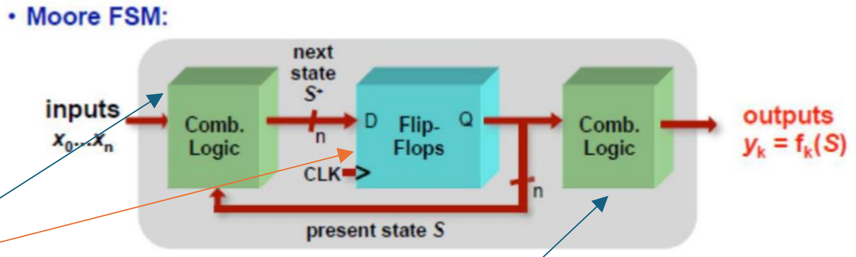
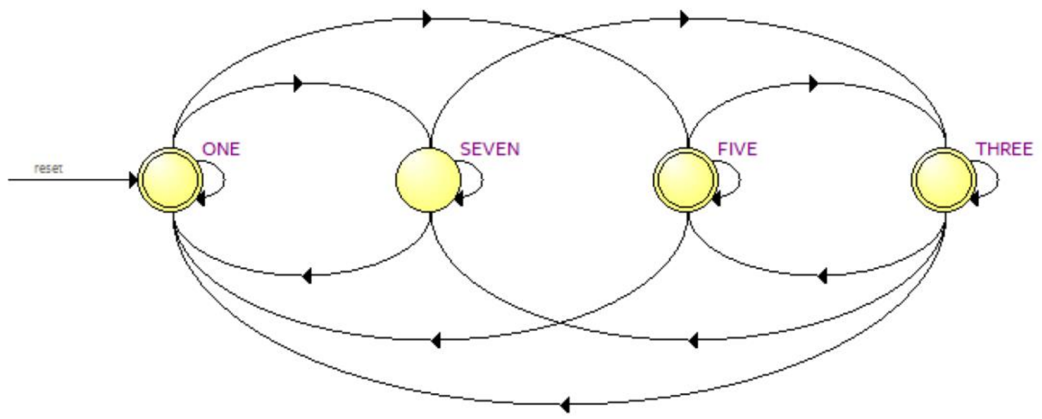
## Example Design: Random Counter using a Moore FSM:

```
/*----------------------------------------------------------------------------
  Random Counter:
  ---------------
  The counter if reset should display a count of 3'b001. If the reset is inactive
  and the counter is enabled and the mode is 0, the counting sequence should be:
  1 7 3 5 1 7 3 5 ... and so on.
  If the counter is enabled and the mode is 1, the counting squence should be:
  1 5 3 7 1 5 3 7 ... and so on.
  If the counter is not reset but the enable is not high, then the counter should
  hold the last value it has reached.
  This counter is synchronized by a clock. The reset and the enable are also
  synchronized, meaning that their effect will only be applied at a positive clock edge.
  The counter should be designed using a Moore FSM.
   */


module RandomCounter (count, clk, reset, enable, mode);

output reg [2:0] count;
input            clk, reset, enable, mode;

reg [1:0] state, next_state;

parameter ONE   = 2'b00;
parameter SEVEN = 2'b01;
parameter THREE = 2'b10;
parameter FIVE  = 2'b11;


// Sequential logic:

always @ (posedge clk)
if (reset) state <= ONE;
    else   state <= next_state;

// Combinational logic to find the next_state based on the inputs and the current state:

always @ *
case (state)

   ONE :       if (enable & ~mode)  next_state = SEVEN;   // 1 7 3 5 1 7 3 5
          else if (enable &  mode)  next_state = FIVE;    // 1 5 3 7 1 5 3 7
          else                      next_state = ONE;     // hold

 SEVEN :       if (enable & ~mode)  next_state = THREE;   // 1 7 3 5 1 7 3 5
          else if (enable &  mode)  next_state = ONE;     // 1 5 3 7 1 5 3 7
          else                      next_state = SEVEN;   // hold

 THREE :       if (enable & ~mode)  next_state = FIVE ;   // 1 7 3 5 1 7 3 5
          else if (enable &  mode)  next_state = SEVEN;   // 1 5 3 7 1 5 3 7
          else                      next_state = THREE;   // hold

 FIVE :        if (enable & ~mode)  next_state = ONE;     // 1 7 3 5 1 7 3 5
          else if (enable &  mode)  next_state = THREE;   // 1 5 3 7 1 5 3 7
          else                      next_state = FIVE;    // hold

 default: next_state = ONE;     // assumed to be the starting and reset case.

endcase


// Combinational logic to determine the output based on the current state:
 always @ (state)
 case (state)

 ONE   : count = 3'b001;
 SEVEN : count = 3'b111;
 THREE : count = 3'b011;
 FIVE  : count = 3'b101;
 default: count = 3'b000;    // invalid output

 endcase

 endmodule
```

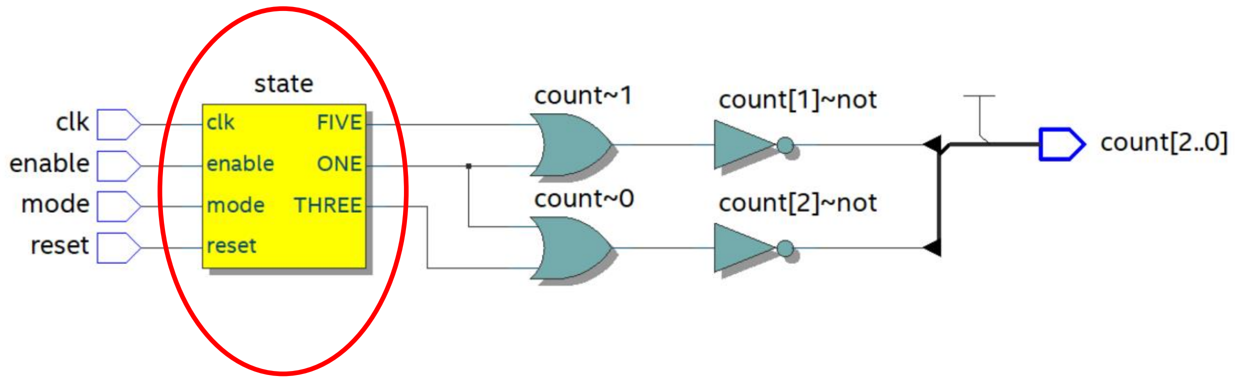


We can observe and verify the design used for the FSM by using the State Machine Viewer in Quartus:

**Tools -> Netlist viewer -> State Machine Viewer**



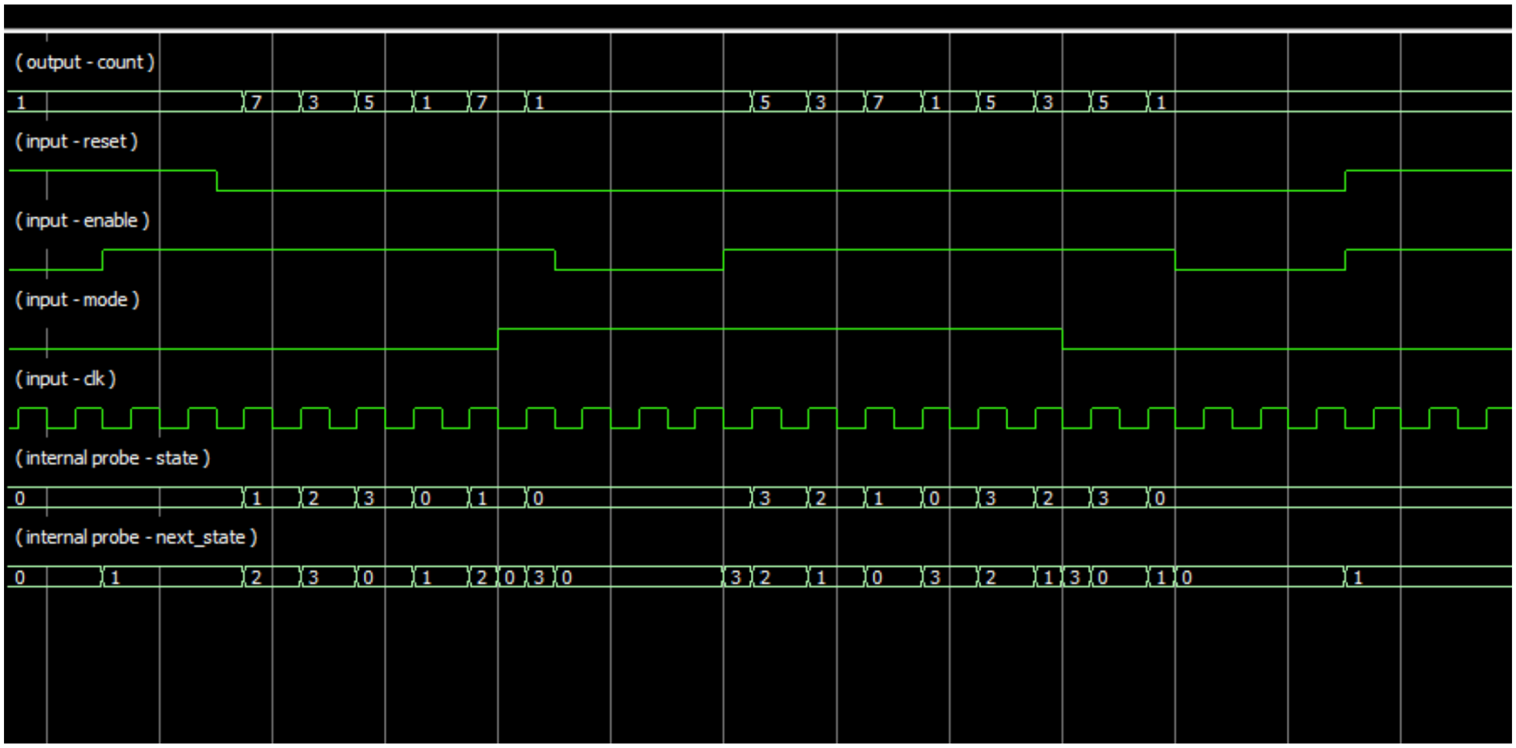
You can also access it by going to the **RTL viewer** and double clicking on the sequential logic finding the value of the next state and updating the state value.

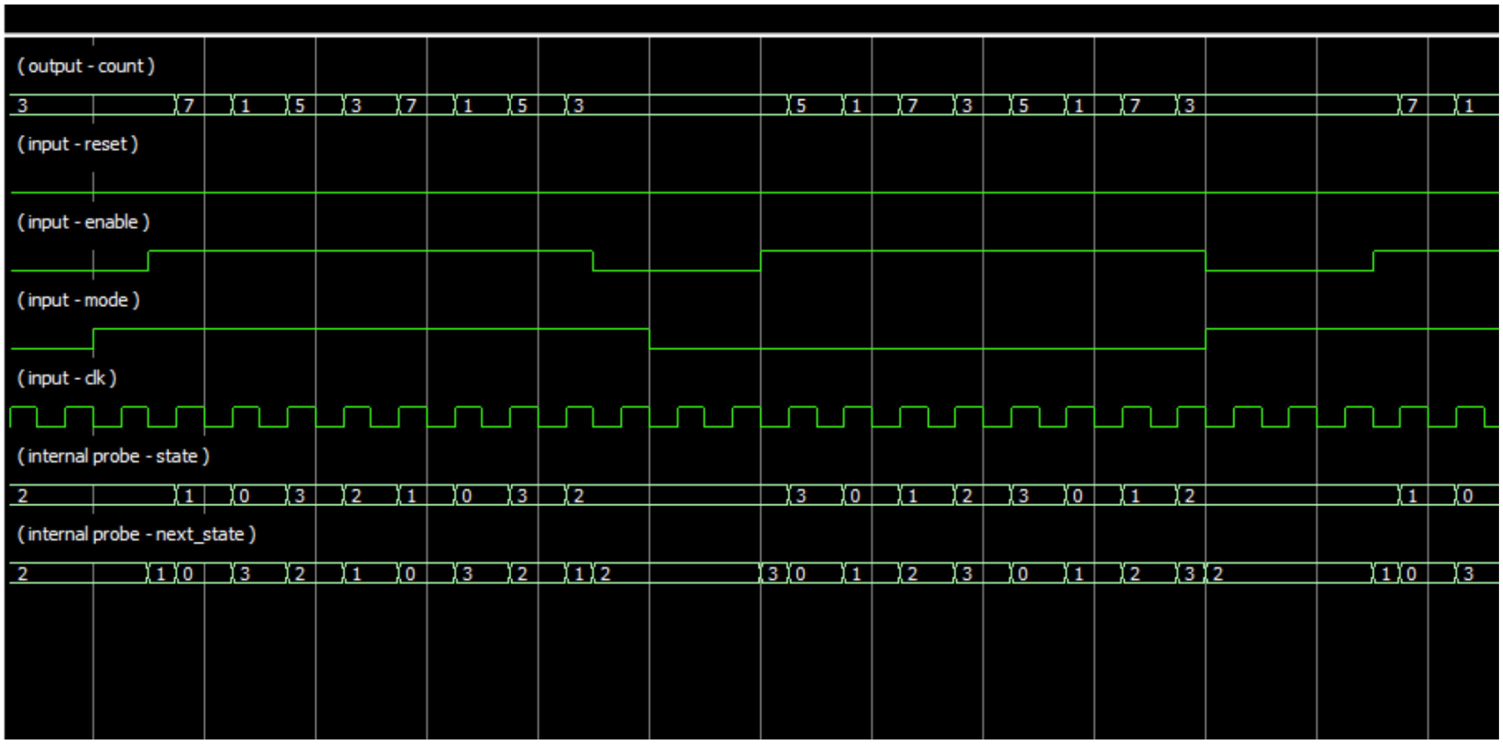


```
module RandomCounter_TB ();

wire [2:0] count;
reg         reset, enable, mode;
reg         clk;

// internal probes to observe how the state is changing:
wire [1:0] state, next_state;

RandomCounter UUT (count, clk, reset, enable, mode);

// assigning the internal probes to test the internal registers of the UUT
assign state      = UUT.state;
assign next_state = UUT.next_state;

initial
   begin
   clk = 1'b0;
   forever
   #5    clk = ~clk;
   end

 initial
    begin
    reset = 1'b1;
    # 50 reset = 1'b0;
    #200 reset = 1'b1;
    #40  reset = 1'b0;
    end


initial
   begin
   mode = 1'b0;
   forever
   # 100 mode = ~mode;
   end

initial
   begin
   enable = 1'b0;
   forever
      begin
      # 30 enable =  1'b1;
      # 80 enable =  1'b0;
      end
   end

endmodule
```

Observing internal signals within the UUT is sometimes very helpful in trouble shooting problems in the code and tracking any functional mistakes. You don't have to take the signal as an output, but rather assign an internal testing probe.

The simulation results:


( output - count )
( input - reset )
( input - enable )
( input - mode )
( input - clk )
( internal probe - state )
( internal probe - next_state )


( output - count )
( input - reset )
( input - enable )
( input - mode )
( input - clk )
( internal probe - state )
( internal probe - next_state )