Finite state machine Design

1. If input ‘in’ is asserted, the state machine moves IDLE->START & remains in START until rst is asserted.
2. If input ‘in’ is de-asserted while FSM is in START state, the state machine moves to FINISH.
3. Output ‘out’ is asserted when FSM is in FINISH state.

1. Create a Verification play to verify the FSM.
2. Create an SV testbench with Random stimulus to verify all possible state transitions.
3. Simulate the design.
4. Ensure the code is compile clean & there is no simulation error.
5. Verify that the waveform covers all possible scenarios

FSM design

module fsm (

input logic clk, // Clock signal

input logic rst, // Reset signal

input logic in, // Input signal

output logic out // Output signal

);

// Define FSM states

typedef enum logic [1:0] {

IDLE = 2'b00,

START = 2'b01,

FINISH = 2'b10

} state\_t;

state\_t current\_state, next\_state;

// State transition logic

always\_ff @(posedge clk or posedge rst) begin

if (rst) begin

current\_state <= IDLE; // Reset to IDLE state

end else begin

current\_state <= next\_state;

end

end

// Next state logic based on inputs

always\_comb begin

case (current\_state)

IDLE: begin

if (in) next\_state = START; // Transition to START if in is asserted

else next\_state = IDLE; // Stay in IDLE if in is de-asserted

end

START: begin

if (in) next\_state = START; // Stay in START if in is still asserted

else next\_state = FINISH; // Transition to FINISH if in is de-asserted

end

FINISH: begin

next\_state = FINISH; // Stay in FINISH until reset occurs

end

default: next\_state = IDLE; // Default case to avoid latches

endcase

end

// Output logic

always\_ff @(posedge clk or posedge rst) begin

if (rst) begin

out <= 0;

end else if (current\_state == FINISH) begin

out <= 1; // Output is asserted when in FINISH state

end else begin

out <= 0;

end

end

endmodule

Design Plan: It includes checking for:

* **State Transitions**: Ensure all valid state transitions (IDLE -> START, START -> FINISH) happen correctly.
* **Input Handling**: Verify behavior when in is asserted or de-asserted.
* **Reset Behavior**: Ensure FSM resets correctly.
* **Output Behavior**: Assert out only when FSM is in FINISH state.
* **Edge Cases**: Verify FSM under boundary conditions, including when in is de-asserted at unexpected moments.

SV testbench:

module tb\_fsm;

// Declare inputs and outputs

logic clk;

logic rst;

logic in;

logic out;

// Instantiate the FSM design

fsm fsm\_inst (

.clk(clk),

.rst(rst),

.in(in),

.out(out)

);

// Clock generation

always begin

#5 clk = ~clk; // 10ns clock period

end

// Random stimulus generation

initial begin

// Initialize signals

clk = 0;

rst = 0;

in = 0;

// Apply reset

rst = 1;

#10 rst = 0; // Assert reset for 10ns

// Random stimulus: Generate random values for 'in'

#10 in = $random % 2; // Random 'in' value

#20 in = $random % 2; // Random 'in' value

#30 in = $random % 2; // Random 'in' value

#40 in = $random % 2; // Random 'in' value

// Finish the simulation after sufficient time

#100 $finish;

end

// Check all possible FSM state transitions and output behavior

initial begin

// Watch for transitions and output behavior

$monitor("Time = %t | rst = %b | in = %b | out = %b | Current State = %b",

$time, rst, in, out, fsm\_inst.current\_state);

end

// Assertions: Check that output 'out' is high only in FINISH state

property p\_out\_high;

@(posedge clk) disable iff (rst) (fsm\_inst.current\_state == fsm\_inst.FINISH) |-> (out == 1);

endproperty

assert property (p\_out\_high) else $fatal("Output 'out' should only be high in FINISH state!");

// Coverage: Ensure all transitions are covered

covergroup cg\_state\_transitions;

coverpoint fsm\_inst.current\_state;

endgroup

cg\_state\_transitions cg\_transitions = new;

// Collect coverage for state transitions

initial begin

#10 cg\_transitions.sample();

end

endmodule