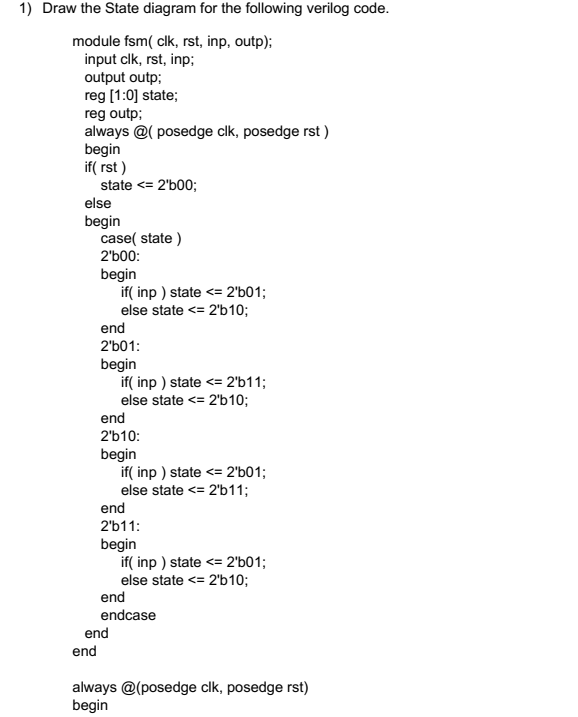
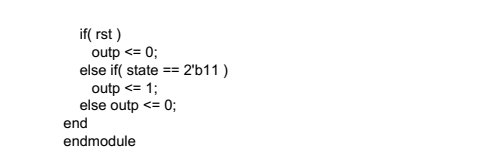
**FSM Question**

Phạm Quang Vinh ITITIU21347

A diagram of a state

Description automatically generated



Write the testbench to test the operation of this FSM circuit.

module fsm\_tb();

// Inputs

reg clk;

reg rst;

reg inp;

// Outputs

wire outp;

// Instantiate the FSM module

fsm uut (

.clk(clk),

.rst(rst),

.inp(inp),

.outp(outp)

);

// Clock generation

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

// Initial values

initial begin

clk = 0;

rst = 1; // Reset active high

inp = 0;

// Reset sequence

#10 rst = 0;

#10 rst = 1;

// Test sequence

// Scenario 1: inp = 0, expect outp = 0

#10 inp = 0;

#10;

// Scenario 2: inp = 1, expect outp = 0

#10 inp = 1;

#10;

// Scenario 3: inp = 1, expect outp = 0

#10 inp = 1;

#10;

// Scenario 4: inp = 0, expect outp = 0

#10 inp = 0;

#10;

// Scenario 5: inp = 1, expect outp = 1

#10 inp = 1;

#10;

// Scenario 6: inp = 0, expect outp = 0

#10 inp = 0;

#10;

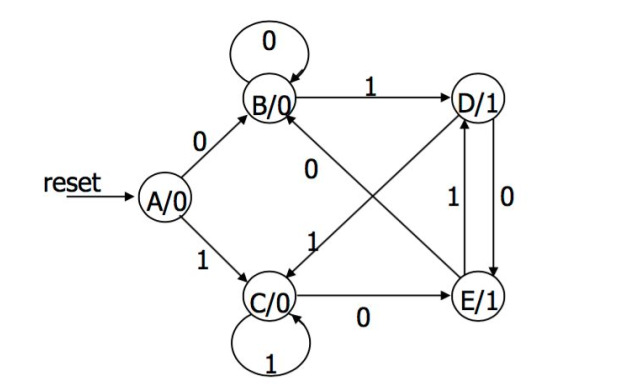
// End simulation

#10 $finish;

end

endmodule

2. A FSM has a 1 ­bit input, a 1­bit output and 5 states. The behavior is described by the state  
diagram below. Implement this FSM in Verilog, Write the testbench to test the operation of this FSM circuit.



**Verilog Implementation**

module FSM (

input wire clk,

input wire reset,

input wire inp,

output reg out

);

// Define states

typedef enum logic [2:0] {

A, B, C, D, E, End

} state\_t;

// State and next state variables

state\_t state, next\_state;

// State register

always\_ff @(posedge clk or posedge reset)

begin

if (reset)

state <= A; // Initial state

else

state <= next\_state;

end

// State transition and output logic

always\_comb

begin

case (state)

A: begin

if (inp)

next\_state = C;

else

next\_state = B;

out = 0;

end

B: begin

if (inp)

next\_state = D;

else

next\_state = B;

out = 0;

end

C: begin

if (inp)

next\_state = C;

else

next\_state = E;

out = 0;

end

D: begin

if (inp)

next\_state = C;

else

next\_state = E;

out = 1;

end

E: begin

if (inp)

next\_state = D;

else

next\_state = End;

out = 1;

end

End: begin

next\_state = End;

out = 1;

end

default: begin

next\_state = A;

out = 0;

end

endcase

end

endmodule

**Testbench**

module FSM\_tb;

// Parameters

parameter CLK\_PERIOD = 10; // Clock period in ns

// Signals

reg clk;

reg reset;

reg inp;

wire out;

// Instantiate FSM

FSM uut (

.clk(clk),

.reset(reset),

.inp(inp),

.out(out)

);

// Clock generation

always #((CLK\_PERIOD)/2) clk = ~clk;

// Initial stimulus

initial begin

clk = 0;

reset = 1;

inp = 0;

// Reset FSM

#20 reset = 0;

// Test case 1

#10 inp = 1; // Transition A -> C

#10 inp = 0; // Stay in C

#10 inp = 1; // Transition C -> C

#10 inp = 0; // Transition C -> E

#10 inp = 1; // Transition E -> D

#10 inp = 0; // Transition D -> End

// Add more test cases as needed

// End simulation

#10 $finish;

end

endmodule

3. Develop the FSM state diagram for the Sequence Recognizer to Identify the sequence 1001, regardless of where it occurs in a longer sequence. Implement this circuit using Verilog. Develop the testbench to test this circuit.

**1. FSM State Diagram**

The FSM for recognizing the sequence "1001" can be designed as follows:

* **States:**
  + **S0:** Initial state, waiting for the first '1'.
  + **S1:** Detected '1'.
  + **S2:** Detected '10'.
  + **S3:** Detected '100'.
  + **S4:** Detected '1001' (sequence detected).
* **Transitions:**
  + From **S0**, transition to **S1** on input '1'.
  + From **S1**, transition to **S2** on input '0'.
  + From **S2**, transition to **S3** on input '0'.
  + From **S3**, transition to **S4** on input '1'.
  + From **S4**, transition back to **S1** on input '1' to detect subsequent occurrences.
* **Reset Condition:**
  + Reset to **S0** on any other input or if no further '1' is detected after '1001'.

**2. Verilog Implementation**

module SequenceRecognizer(

input wire clk, // Clock input

input wire reset, // Reset input

input wire data\_in, // Input data stream (1-bit)

output reg sequence\_detected // Output indicating sequence detected

);

// Define FSM states

typedef enum logic [2:0] {

S0, S1, S2, S3, S4

} state\_type;

// State and next state variables

reg [2:0] state, next\_state;

// Sequential logic: state transition and output logic

always @(posedge clk or posedge reset) begin

if (reset) begin

state <= S0; // Reset to initial state

sequence\_detected <= 0; // Reset output

end else begin

state <= next\_state;

sequence\_detected <= (state == S4); // Output '1' when sequence detected

end

end

// Combinational logic: next state and state transition conditions

always @(\*) begin

case (state)

S0: next\_state = (data\_in == 1) ? S1 : S0;

S1: next\_state = (data\_in == 0) ? S2 : S1;

S2: next\_state = (data\_in == 0) ? S3 : S0;

S3: next\_state = (data\_in == 1) ? S4 : S0;

S4: next\_state = (data\_in == 1) ? S1 : S0;

default: next\_state = S0;

endcase

end

endmodule

**3. Testbench for Verification**

module testbench;

// Parameters

parameter CLK\_PERIOD = 10; // Clock period in ns

// Signals

reg clk;

reg reset;

reg data\_in;

wire sequence\_detected;

// Instantiate the Sequence Recognizer

SequenceRecognizer dut (

.clk(clk),

.reset(reset),

.data\_in(data\_in),

.sequence\_detected(sequence\_detected)

);

// Clock generation

always #((CLK\_PERIOD)/2) clk = ~clk;

// Initial reset

initial begin

clk = 0;

reset = 1;

data\_in = 0;

#20 reset = 0;

end

// Test case

initial begin

// Sequence "1001" at different positions

#30 data\_in = 1; // S0 -> S1

#10 data\_in = 0; // S1 -> S2

#10 data\_in = 0; // S2 -> S3

#10 data\_in = 1; // S3 -> S4, sequence detected

#100 data\_in = 1; // S4 -> S1 (new occurrence)

// Add more test cases as needed

// Ensure to reset and test for sequences not containing "1001"

// and edge cases (e.g., sequence split across clock cycles)

#100 $finish; // End simulation after sufficient time

end

endmodule