SC1005

Tut 10

CE/CZ1105 Digital Logic
Tutorial 10

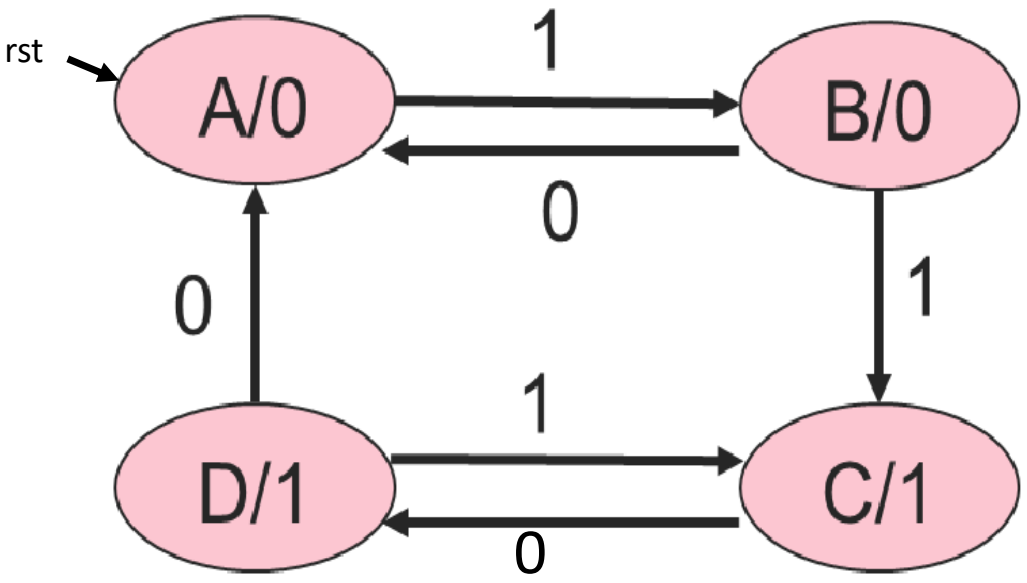
Q1. Design a finite state machine that has a single input and single output. It outputs a 1 from the second consecutive high input, and only then outputs a zero after the second consecutive low input. Hence, two consecutive 1 inputs, get a high output, that stays until two consecutive low inputs are received.

(a) Implement the finite state machine in Verilog using a combinational always block for the state transition logic.

(b) Redo the implementation using only assign statements for the state transition logic.

(c) Show how the state machine would respond to the following sequence of inputs: 0,1,0,1,1,0,1,1,0,0,0,1,0,1,1

First get the FSM transition (flow) diagram



State Transition Diagram

Also called a State Flow Diagram

In state A (with output 0), a 1 will take us to state B (with output 0). We remain in state A if the input is 0.

In state B, a 1 will take us to state C (with output 1), while a 0 will take us back to state A.

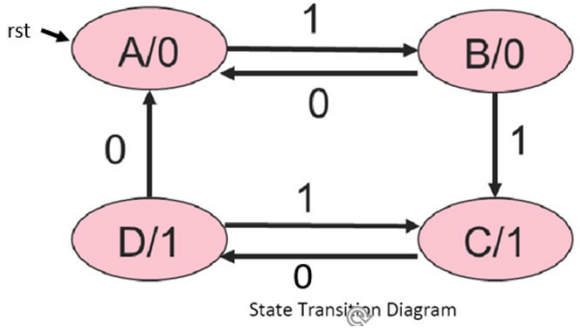
In state C, a 0 will take us to state D (output = 1). We remain in state C if the input is 1.

In state D, a 0 will take us to state A, while a 1 will take us back to state C.

Note: reset (rst) causes the FSM to go to state A (the reset state). Also, rst is not included in the input signals.

```
module fsma (input a, clk, rst, output reg x);
    parameter sta=2'b00, stb=2'b01, stc=2'b10, std=2'b11;
    reg [1:0] n, s;
    always @ * begin
          n = s; x = 1'b0;
          case(s)
                sta: if (a) n = stb;
                stb: if (a) n = stc; else n = sta;
                stc: begin
                     if (!a) n = std;   x = 1'b1;
                end
                std: begin
                     if (!a) n = sta;
                     else n = stc;
                     x = 1'b1;
                end
          endcase
    end
```

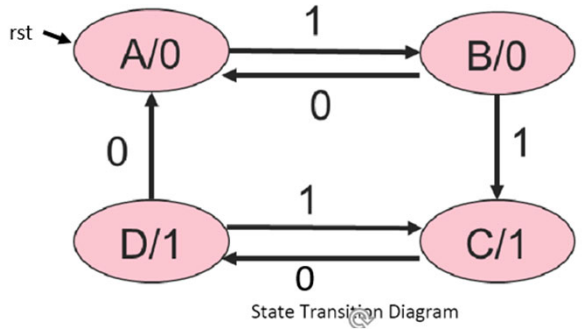
(a) Then we can implement the FSM in Verilog



State Transition Diagram

```
always @ (posedge clk)
begin
      if(rst)
            s <= sta;
      else
      s <= n;
end
endmodule
```

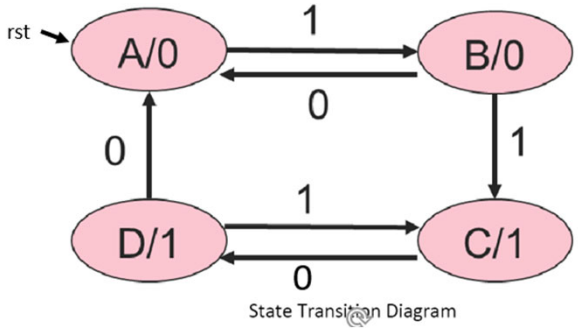
```
module fsma (input a, clk, rst, output x);
    parameter sta=2'b00, stb=2'b01, stc=2'b10, std=2'b11;
    reg [1:0] n, s;
    assign x = s[1];
    always @ *
    begin
          n = s;
          case(s)
                sta: if (a) n = stb;
                stb: if (a) n = stc; else n = sta;
                stc: if (!a) n = std;
                std: if (!a) n = sta; else n = stc;
          endcase
    end
    always @ (posedge clk)
    begin
            if(rst)  s <= sta;
            else     s <= n;
    end
endmodule
```



Note: The question asked to implement the finite state machine in Verilog using a combinational always block for the state transition logic.

So, this answer where we have used a continuous assignment for x is also correct.

(b) First, generate the transition table from the FSM flow diagram



State Transition Diagram

We can get the state transition table from the flow diagram, as:

Then by allocating binary values to the states we get the excitation tables as:

We can modify the state transition table so that the states (and possibly the inputs) are in grey code format. That makes it easier in the next part (getting the K-maps).

State (S_1S_0)	Next state (n_1n_0) a = 0	1	Output (x)
A	A	B	0
B	A	C	0
C	A	D	1
D	A	C	1

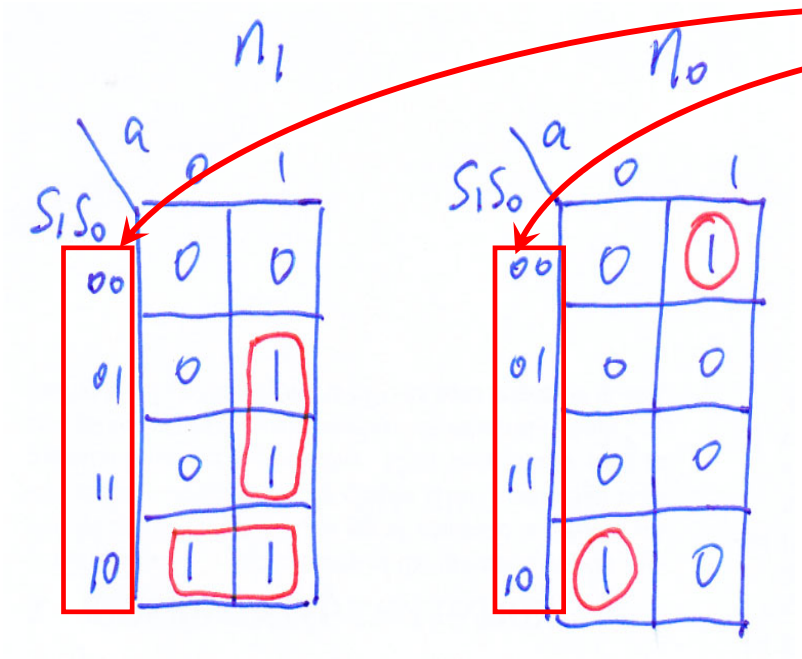
State Transition Table

State (S_1S_0)	Next state (n_1n_0) a = 0	1	Output (x)
A (00)	A (00)	B (01)	0
B (01)	A (00)	C (10)	0
D (11)	A (00)	C (10)	1
C (10)	D (11)	C (10)	1

Modified State Transition Table

(States C and D are swapped to make K-map easy)

(b) Then, determine the logic for the assign statements



State (S_1S_0)	Next state (n_1n_0) a = 0	1	Output (x)
A (00)	A (00)	B (01)	0
B (01)	A (00)	C (10)	0
D (11)	A (00)	C (10)	1
C (10)	D (11)	C (10)	1

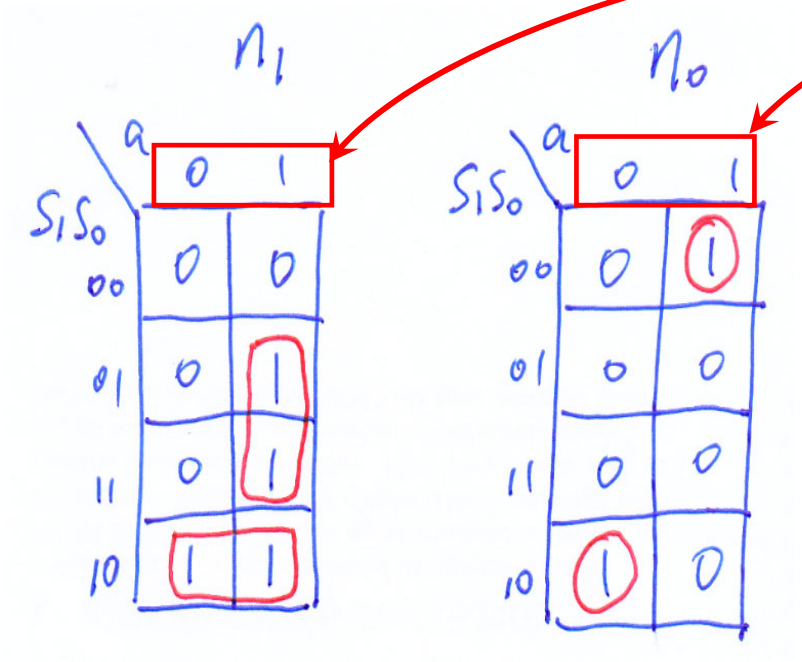
Modified State Transition Table

(States C and D are swapped to make K-map easy)

The current state (S_1 S_0)

Note: The K-maps come directly from the modified state transition table.

(b) Then, determine the logic for the assign statements



State (S_1S_0)	Next state (n_1n_0) a = 0	1	Output (x)
A (00)	A (00)	B (01)	0
B (01)	A (00)	C (10)	0
D (11)	A (00)	C (10)	1
C (10)	D (11)	C (10)	1

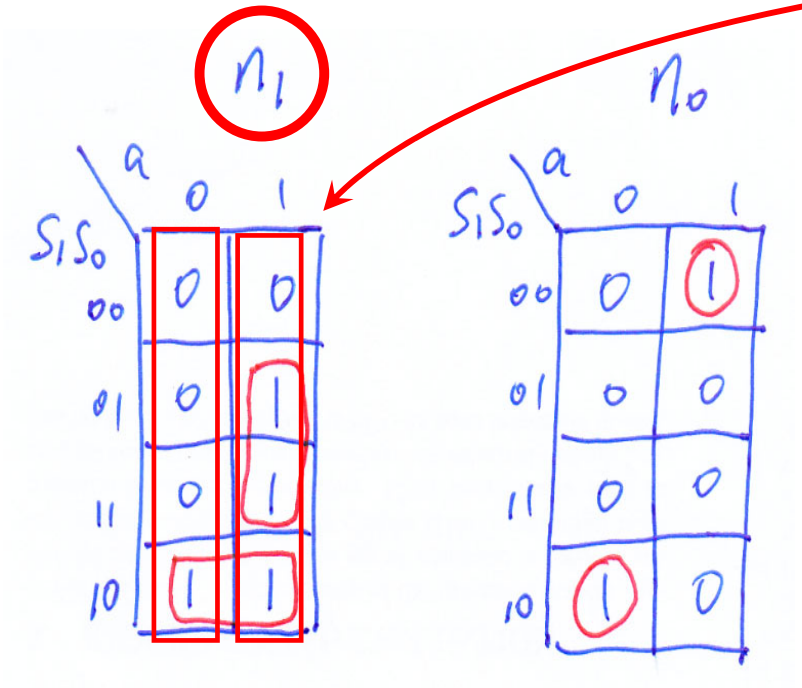
Modified State Transition Table

(States C and D are swapped to make K-map easy)

The current state (S_1 S_0) and the inputs (a) form the input conditions of the K-map.

Note: The K-maps come directly from the modified state transition table.

(b) Then, determine the logic for the assign statements



State (S_1S_0)	Next state (n_1n_0) a = 0	1	Output (x)
A (00)	A (00)	B (01)	0
B (01)	A (00)	C (10)	0
D (11)	A (00)	C (10)	1
C (10)	D (11)	C (10)	1

Modified State Transition Table

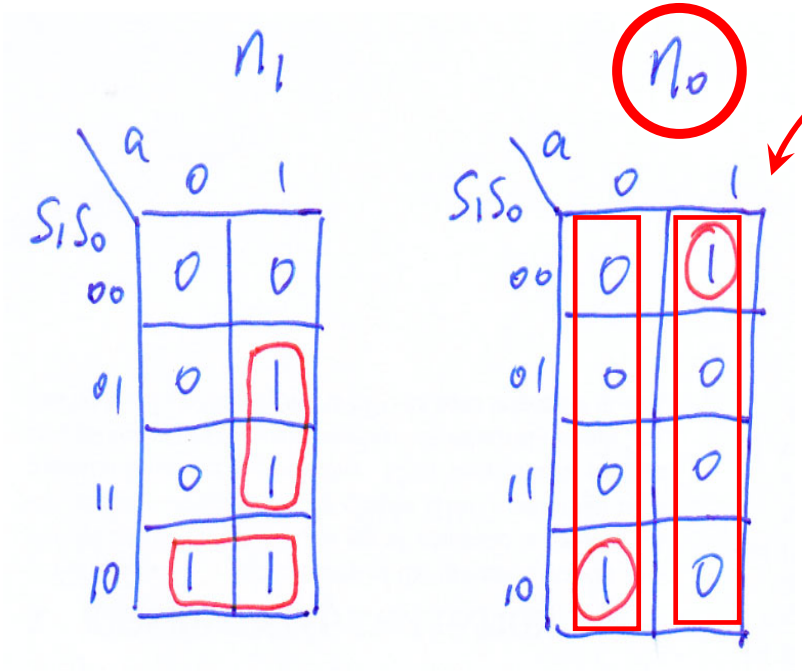
(States C and D are swapped to make K-map easy)

The current state (S_1 S_0) and the inputs (a) form the input conditions of the K-map.

The left-hand columns represent next state n_1

Note: The K-maps come directly from the modified state transition table.

(b) Then, determine the logic for the assign statements



State (S_1S_0)	Next state (n_1n_0) a = 0	1	Output (x)
A (00)	A (00)	B (01)	0
B (01)	A (00)	C (10)	0
D (11)	A (00)	C (10)	1
C (10)	D (11)	C (10)	1

Modified State Transition Table

(States C and D are swapped to make K-map easy)

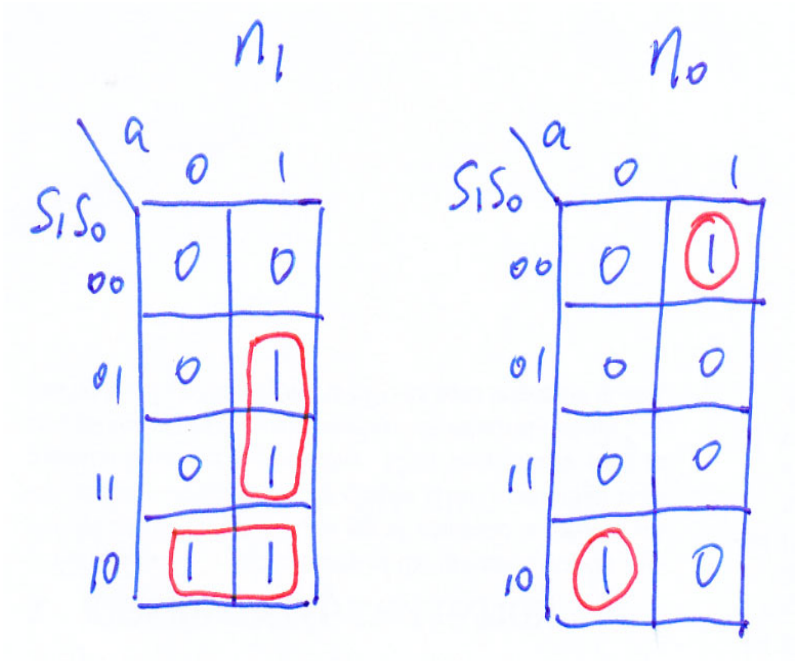
Note: The K-maps come directly from the modified state transition table.

The current state (S_1 S_0) and the inputs (a) form the input conditions of the K-map.

The left-hand columns represent next state n_1

The right-hand columns represent next state n_0

(b) Then, determine the logic for the assign statements



State (S_1S_0)	Next state (n_1n_0) a = 0	1	Output (x)
A (00)	A (00)	B (01)	0
B (01)	A (00)	C (10)	0
D (11)	A (00)	C (10)	1
C (10)	D (11)	C (10)	1

Modified State Transition Table

(States C and D are swapped to make K-map easy)

Then the next state and output expressions are:

$$n[1] = s[0]a + s[1]s[0]'$$

$$n[0] = s[1]'s[0]'a + s[1]s[0]'a'$$

$$x = s[1]$$

(b) Then, the Verilog code is

```
module fsma (input a, clk, rst, output x);
    reg [1:0] s;
    wire [1:0] n;       // n is declared as a wire

    assign n[1] = (s[0]&a) | (s[1]&~s[0]);
    assign n[0] = (~s[1]&~s[0]&a) | (s[1]&~s[0]&~a);
    assign x = s[1];

    always @ (posedge clk)
    begin
            if(rst)
                    s <= 2'b0;
            else
                    s <= n;
    end
endmodule
```

$$n[1] = s[0]a + s[1]s[0]'$$

$$n[0] = s[1]'s[0]'a + s[1]s[0]'a'$$

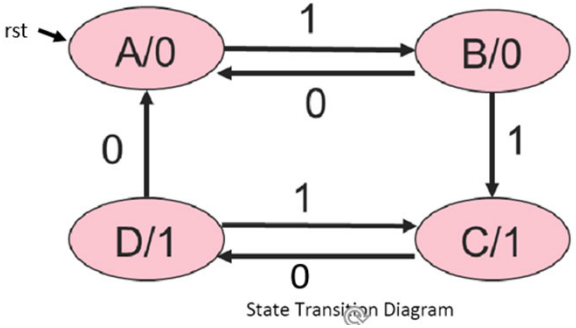
$$x = s[1]$$

The module description and declarations now must account for the next state & output being assigned from a continuous assign statement (eg NOT **reg**)

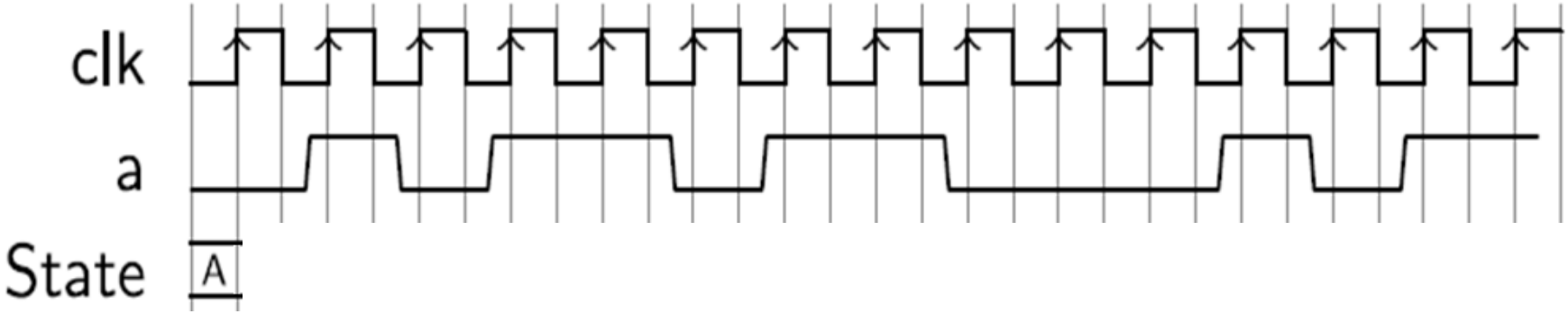
The synchronous block is identical to before.

The next states and output come from the K-map logic expressions.

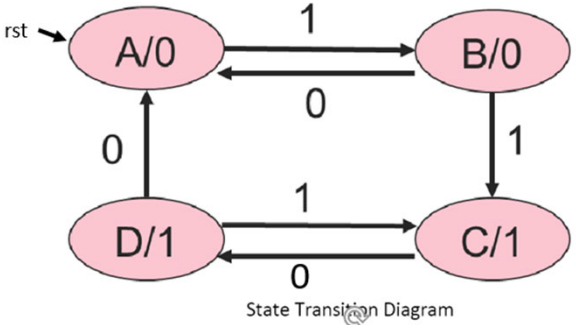
(c) Determine the FSM output for the input sequence: 0,1,0,1,1,0,1,1,0,0,0,1,0,1,1



We assume we start in the idle/initial state (state A). We then consider what state transitions we will get based on the given input sequence. Once we have the state sequence, we can determine the outputs.



(c) Determine the FSM output for the input sequence: 0,1,0,1,1,0,1,1,0,0,0,1,0,1,1



We assume we start in the idle/initial state (state A). We then consider what state transitions we will get based on the given input sequence. Once we have the state sequence, we can determine the outputs.