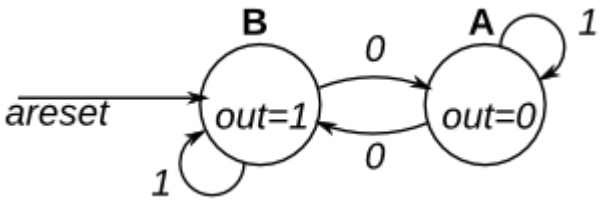
Fsm1

This is a Moore state machine with two states, one input, and one output. Implement this state machine. Notice that the reset state is B.

This exercise is the same as fsm1s, but using asynchronous reset.

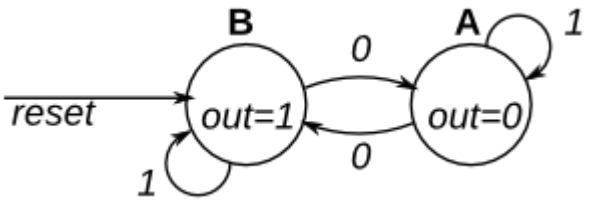



```
module top_module(
    input clk,
    input areset,    // Asynchronous reset to state B
    input in,
    output out);//

    parameter A=0, B=1;
    reg state, next_state;

    always @(*) begin    // This is a combinational always block
        // State transition logic
        case(state)
            A:next_state=(in)?A:B;
            B:next_state=(in)?B:A;
                    endcase

    end

    always @(posedge clk, posedge areset) begin    // This is a sequential always block
        // State flip-flops with asynchronous reset
                    if (areset) begin
                            state <= 1'b1;
                    end
                    else begin
                            state <= next_state;
                    end
    end

            assign out = state;

endmodule
```

Fsm1s

This is a Moore state machine with two states, one input, and one output. Implement this state machine. Notice that the reset state is B.

This exercise is the same as fsm1, but using synchronous reset.

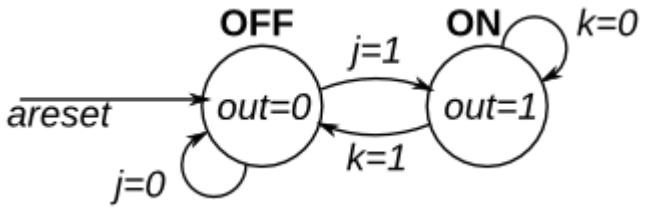



```
module top_module(clk, reset, in, out);
    input clk;
    input reset;    // Synchronous reset to state B
    input in;
    output out;//
    reg out;

    // Fill in state name declarations
    parameter STATE_A= 1'b0;
    parameter STATE_B= 1'b1;

    reg present_state, next_state;

    always @(posedge clk) begin
        if (reset) begin
            // Fill in reset logic
            present_state <= STATE_B;
        end
        else begin
            present_state <= next_state;
        end
    end

    always @ (*) begin
        case(present_state)
            STATE_A:next_state = (in ==1'b0)?STATE_B:STATE_A;
            STATE_B:next_state = (in ==1'b0)?STATE_A:STATE_B;
        endcase
    end

    assign out = (present_state==STATE_B)? 1'b1: 1'b0;
endmodule
```

Fsm2

This is a Moore state machine with two states, two inputs, and one output. Implement this state machine.

This exercise is the same as fsm2s, but using asynchronous reset.

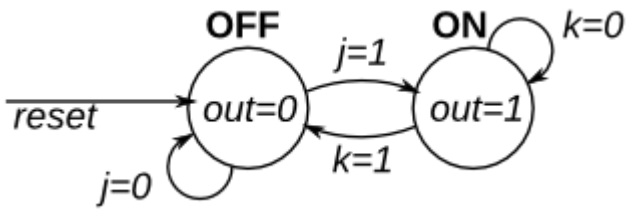



```
module top_module(
    input clk,
    input areset,    // Asynchronous reset to OFF
    input j,
    input k,
    output out); //

    parameter OFF=0, ON=1;
    reg state, next_state;

    always @(*) begin
        // State transition logic
        case(state)
            ON : next_state=(k==1'b0)?ON :OFF;
            OFF: next_state=(j==1'b0)?OFF:ON;
        endcase
    end

    always @(posedge clk, posedge areset) begin
        // State flip-flops with asynchronous reset
        if(areset)begin
            state <= OFF;
        end
        else begin
            state <= next_state;
        end
    end

    // Output logic
    assign out = (state == OFF)? 1'b0:1'b1;

endmodule
```

Fsm2s

This is a Moore state machine with two states, two inputs, and one output. Implement this state machine.

This exercise is the same as fsm2, but using synchronous reset.




```
module top_module(
    input clk,
    input reset,    // Synchronous reset to OFF
    input j,
    input k,
    output out); //

    parameter OFF=0, ON=1;
    reg state, next_state;

    always @(*) begin
        case(state)
            OFF: next_state=(j==1'b1)?ON:OFF;
            ON : next_state=(k==1'b1)?OFF:ON;
        endcase
        // State transition logic
    end

    always @(posedge clk ) begin
        if(reset)begin
            state <= OFF;
        end// State flip-flops with synchronous reset
        else begin
            state <= next_state;
        end
    end

    // Output logic
    assign out = (state == OFF)?1'b0:1'b1;

endmodule
```

Fsm3comb

The following is the state transition table for a Moore state machine with one input, one output, and four states. Use the following state encoding: A=2'b00, B=2'b01, C=2'b10, D=2'b11.

Implement only the state transition logic and output logic (the combinational logic portion) for this state machine. Given the current state (`state`), compute the `next_state` and output (`out`) based on the state transition table.

State	Next state		Output
	in=0	in=1	
A	A	B	0
B	C	B	0
C	A	D	0
D	C	B	1

```
module top_module(
    input in,
    input [1:0] state,
    output [1:0] next_state,
    output out); //

    parameter A=0, B=1, C=2, D=3;

    always @(*)begin
        case(state)
            A:next_state=(in==1'b0)?A:B;
            B:next_state=(in==1'b0)?C:B;
            C:next_state=(in==1'b0)?A:D;
            D:next_state=(in==1'b0)?C:B;
        endcase
    end
    // State transition logic: next_state = f(state, in)
```

```
    // Output logic:  out = f(state) for a Moore state machine
    always @(*)begin
        case(state)
            A:out=1'b0;
            B:out=1'b0;
            C:out=1'b0;
            D:out=1'b1;

        endcase
    end

endmodule
```

Fsm3onehot

The following is the state transition table for a Moore state machine with one input, one output, and four states. Use the following one-hot state encoding: A=4'b0001, B=4'b0010, C=4'b0100, D=4'b1000.

Derive state transition and output logic equations by inspection assuming a one-hot encoding. Implement only the state transition logic and output logic (the combinational logic portion) for this state machine. (The testbench will test with non-one hot inputs to make sure you're not trying to do something more complicated)

State	Next state		Output
	in=0	in=1	
A	A	B	0
B	C	B	0
C	A	D	0
D	C	B	1

What does "derive equations by inspection" mean?

One-hot state machine encoding guarantees that exactly one state bit is 1. This means that it is possible to determine whether the state machine is in a particular state by examining only *one* state bit, not *all* state bits. This leads to simple logic equations for the state transitions by examining the incoming edges for each state in the state transition diagram.

For example, in the above state machine, how can the state machine can reach state A? It must use one of the two incoming edges: "Currently in state A and in=0" or "Currently in state C and in = 0". Due to the one-hot encoding, the logic equation to test for "currently in state A" is simply the state bit for state A. This leads to the final logic equation for the next state of state bit A: `next_state[0] = state[0]&(~in) | state[2]&(~in)`. The one-hot encoding guarantees that at most one clause (product term) will be "active" at a time, so the clauses can just be ORed together.

When an exercise asks for state transition equations "by inspection", use this particular method. The judge will test with non-one-hot inputs to ensure your logic equations follow this method, rather that doing something else (such as resetting the FSM) for illegal (non-one-hot) combinations of the state bits.

Although knowing this algorithm isn't necessary for RTL-level design (the logic synthesizer handles this), it is illustrative of why one-hot FSMs often have simpler logic (at the expense of more state bit storage), and this topic frequently shows up on exams in digital logic courses.

```
module top_module(
    input in,
    input [3:0] state,
    output [3:0] next_state,
```

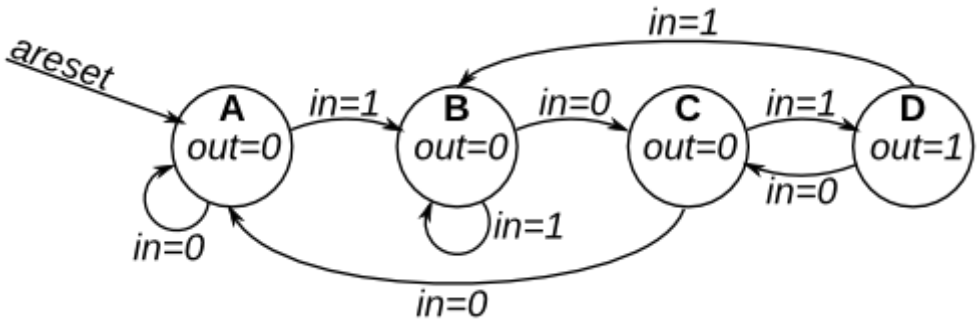
```
    output out); //

    parameter A=0, B=1, C=2, D=3;
//A=4'b0001, B=4'b0010, C=4'b0100, D=4'b1000.
    // State transition logic: Derive an equation for each state flip-flop.
    assign next_state[A] = ~in& (state[0]|state[2]);
    assign next_state[B] =  in& (state[0]|state[1]|state[3]);
    assign next_state[C] = ~in& (state[1]|state[3]);
    assign next_state[D] =  in&  state[2];

    // Output logic:
    assign out = state[3];

endmodule
```

Fsm3

See also: State transition logic for this FSM

The following is the state transition table for a Moore state machine with one input, one output, and four states. Implement this state machine. Include an asynchronous reset that resets the FSM to state A.

State	Next state		Output
	in=0	in=1	
A	A	B	0
B	C	B	0
C	A	D	0
D	C	B	1




```
module top_module(
    input clk,
    input in,
    input areset,
    output out); //

    reg [1:0] curr_state, next_state;
    parameter A=2'b00;
    parameter B=2'b01;
    parameter C=2'b10;
    parameter D=2'b11;

    // State transition logic
    always @ (*)begin
        case(curr_state)
            A:next_state = in? B:A;
            B:next_state = in? B:C;
            C:next_state = in? D:A;
            D:next_state = in? B:C;
        endcase
    end

    // State flip-flops with asynchronous reset
```

```
    always @ (posedge clk or posedge areset)begin
        if(areset) begin
            curr_state <=A;
        end
        else begin
            curr_state <= next_state;
        end
    end

    // Output logic
    assign out = (curr_state==D)? 1'b1:1'b0;

endmodule
```

Fsm3s

See also: State transition logic for this FSM

The following is the state transition table for a Moore state machine with one input, one output, and four states. Implement this state machine. Include a synchronous reset that resets the FSM to state A. (This is the same problem as Fsm3 but with a synchronous reset.)

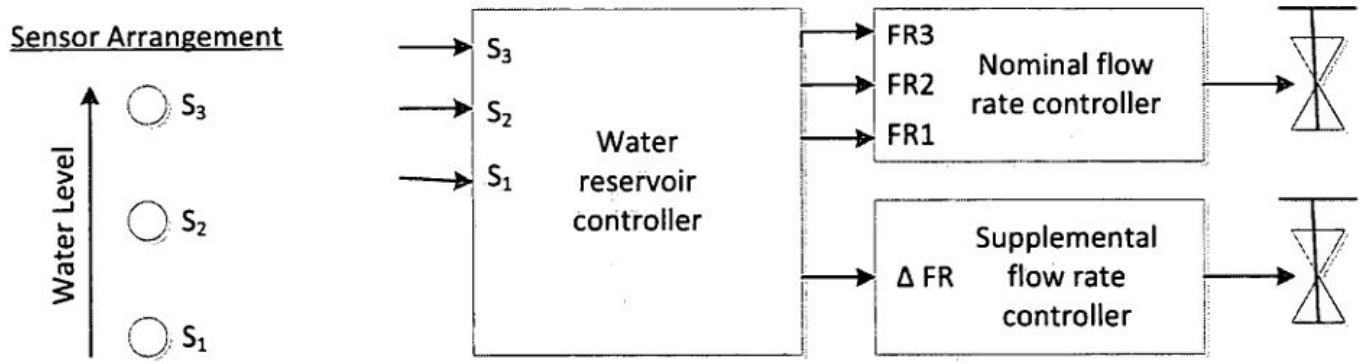
State	Next state		Output
	in=0	in=1	
A	A	B	0
B	C	B	0
C	A	D	0
D	C	B	1

```
module top_module(
    input clk,
    input in,
    input reset,
    output out); //

    reg [1:0] curr_state, next_state;
    parameter A=2'b00;
    parameter B=2'b01;
    parameter C=2'b10;
    parameter D=2'b11;

    // State transition logic
    always @ (*)begin
        case(curr_state)
            A:next_state = in? B:A;
            B:next_state = in? B:C;
            C:next_state = in? D:A;
            D:next_state = in? B:C;
        endcase
    end

    // State flip-flops with asynchronous reset
    always @ (posedge clk )begin
        if(reset) begin
            curr_state <=A;
        end
        else begin
            curr_state <= next_state;
        end
    end
```

```
    // Output logic
    assign out = (curr_state==D)? 1'b1:1'b0;

endmodule
```

Exams/ece241 2013 q4

Q4. [10] A large reservoir of water serves several users. In order to keep the level of water sufficiently high, three sensors are placed vertically at 5-inch intervals. When the water level is above the highest sensor (S_3), the input flow rate should be zero. When the level is below the lowest sensor (S_1), the flow rate should be at maximum (both Nominal flow valve and Supplemental flow valve opened). The flow rate when the level is between the upper and lower sensors is determined by two factors: the water level and the level previous to the last sensor change. Each water level has a nominal flow rate associated with it, as shown in the table below. If the sensor change indicates that the previous level was lower than the current level, the nominal flow rate should take place. If the previous level was higher than the current level, the flow rate should be increased by opening the Supplemental flow valve (controlled by ΔFR). Draw the Moore model state diagram for the water reservoir controller. Clearly indicate all state transitions and outputs for each state. The inputs to your FSM are S_1, S_2 and S_3; the outputs are FR1, FR2, FR3 and ΔFR.



Water Level	Sensors Asserted	Nominal Flow Rate Inputs to Be Asserted
Above S_3	S_1, S_2, S_3	None
Between S_3 and S_2	S_1, S_2	FR1
Between S_2 and S_1	S_1	FR1, FR2
Below S_1	None	FR1, FR2, FR3

Also include an active-high synchronous reset that resets the state machine to a state equivalent to if the water level had been low for a long time (no sensors asserted, and all four outputs asserted).

```
module top_module (
    input clk,
    input reset,
    input [3:1] s,
    output fr3,
    output fr2,
    output fr1,
    output dfr
);

    parameter S0=2'b00;
    parameter S1=2'b01;
    parameter S2=2'b10;
    parameter S3=2'b11;

    reg [1:0] curr_state, next_state;
    wire high_to_low, low_to_high;
```

```
always @ (posedge clk)begin
    //sync reset
    if (reset) begin
        curr_state <= S0;
    end
    else begin
        curr_state <= next_state;
    end
end


always @ (*)begin
    case(curr_state)
        S0: begin
            if(s[1])begin
                    next_state = S1;
                    low_to_high = 1'b1;
                    high_to_low = 1'b0;
            end
            else begin
                next_state = S0;
                    low_to_high = 1'b0;
                    high_to_low = 1'b0;
            end
        end
        S1: begin
            if (~s[1]) begin
                    next_state = S0;
                    low_to_high = 1'b0;
                    high_to_low = 1'b1;
            end
                    else if (s[2])begin
                    next_state = S2;
                low_to_high = 1'b1;
                    high_to_low = 1'b0;
                    end
            else begin
                next_state = S1;
                low_to_high = 1'b0;
                    high_to_low = 1'b0;
            end
        end
        S2:if(~s[2])begin
                    next_state= S1;
                    low_to_high = 1'b0;
                    high_to_low = 1'b1;
                    end
        else if (s[3])begin
            next_state = S3;
            low_to_high = 1'b1;
            high_to_low = 1'b0;
                    end
        else begin
            next_state = S2;
            low_to_high = 1'b0;
            high_to_low = 1'b0;
        end
```

```
            S3:if(~s[3]) begin
                next_state = S2;
                low_to_high = 1'b0;
                high_to_low = 1'b1;
            end
            else begin
                next_state = S3;
                low_to_high = 1'b0;
                high_to_low = 1'b0;
            end
        endcase
    end
     //output logic : fr1 fr2 fr3
    always @ (*)begin
        case(curr_state)
            S0: {fr1,fr2, fr3} = 3'b111;
            S1: {fr1,fr2, fr3} = 3'b110;
            S2: {fr1,fr2, fr3} = 3'b100;
            S3: {fr1,fr2, fr3} = 3'b000;
        endcase
    end

    //output delta(fr) change when state_change
    always @ (posedge clk)begin
        if(reset)begin
            dfr <= 1'b1;
        end
        else if (high_to_low)begin
            dfr<= 1'b1;
        end
        else if(low_to_high)begin
            dfr <= 1'b0;
        end
        else begin
            dfr <= dfr;
        end
    end

endmodule
```