Lab 8 Oct Fall 2019
DSD EE3563

Name: Andrew Quick

1. Draw a SM Chart of your design

Current State	Next State (D3 D2 D1 D0)
0000 (S0 - C)	0001 (S1 - A)
0001 (S1 - A)	0010 (S2 - 4)
0010 (S2 - 4)	0011 (S3 - 5)
0011 (S3 - 5)	0100 (S4 - 7)
0100 (S4 - 7)	0101 (S5 - 3)
0101 (S5 - 3)	0110 (S6 - F)
0110 (S6 - F)	0111 (S7 - 6)
0111 (S7 - 6)	1000 (S8 - 9)
1000 (S8 - 9)	0000 (S0 - C)

2. Type in your FF input equations

D3 = Q2 & Q1 & Q0

D2 = Q3 & ~Q2 & ~Q1 & ~Q0 + ~Q3 & Q2 & ~Q1 & ~Q0

D1 = ~Q2 & Q1 & ~Q0 + Q2 & ~Q1 & Q0

D0 = ~Q0

3. Paste you code in here:

```
`timescale 1ns/10ps
module counting_state_machine
 #
 (
  parameter NUM_SEGMENTS = 8,
  parameter CLK_PER      = 10,   // Clock period in ns
  parameter REFR_RATE    = 1000, // Refresh rate in Hz
  parameter ASYNC_BUTTON = "SAFE" // "CLOCK", "NOCLOCK", "SAFE", "DEBOUNCE"
  )
 (
  input wire                  clk,
  input wire                  BTNC,    // Change state
  input wire                  CPU_RESETN,
  input wire [2:0]              SW,      // Switch inputs: SW[15:11] for address, SW[7:0] for data
  output logic [NUM_SEGMENTS-1:0] anode,
  output logic [7:0]            cathode
  );

 logic [NUM_SEGMENTS-1:0][3:0]      encoded;
 logic [NUM_SEGMENTS-1:0]           digit_point;
 logic                          reset;

 seven_segment
  #
```

```
  (
   .NUM_SEGMENTS (NUM_SEGMENTS),
   .CLK_PER     (CLK_PER),   // Clock period in ns
   .REFR_RATE    (REFR_RATE)  // Refresh rate in Hz
   )
u_7seg
  (
   .clk         (clk),
   .reset       (reset),
   .encoded     (encoded),
   .digit_point  (digit_point),
   .anode       (anode),
   .cathode     (cathode)
   );



// Capture the rising edge of button press
logic                       last_button;
logic                       button;
(* mark_debug = "true" *) logic     button_down;

initial begin
  last_button = 1'b0;       // Initialize single-bit signals
  button      = 1'b0;
  button_down = 1'b0;
```

```
end

generate
  if (ASYNC_BUTTON == "SAFE") begin : g_CLOCK
    (* ASYNC_REG = "TRUE", mark_debug = "true" *) logic [2:0] button_sync;

    always @(posedge clk) begin
      // BTNC handling (Move to the next state)
      button_down <= 1'b0;  // Reset button_down
      button_sync <= {button_sync[1:0], BTNC};  // Shift in BTNC state
      if (button_sync[2:1] == 2'b01)
        button_down <= 1'b1;  // Button pressed
      else
        button_down <= 1'b0;  // No button press detected
    end

  end else begin : g_NOCLOCK
    always @(posedge clk) begin
      last_button <= button;
      button      <= BTNC;
      if (BTNC & ~last_button)
        button_down <= 1'b1;  // Button pressed (detect rising edge)
      else
        button_down <= 1'b0;
    end
```

```
   end
  endgenerate


   // Define states
parameter S0 = 4'b0000, // C
          S1 = 4'b0001, // A
          S2 = 4'b0010, // 4
          S3 = 4'b0011, // 5
          S4 = 4'b0100, // 7
          S5 = 4'b0101, // 3
          S6 = 4'b0110, // F
          S7 = 4'b0111, // 6
          S8 = 4'b1000; // 9 (loop back to C)



  // State variables
  reg [3:0] state, next_state;
  logic [3:0] Z;  // Output from state machine to encoded

  // Use SW[2:0] as input for state transitions
  wire [2:0] sw_input = SW[2:0];  // Use the lowest 3 bits of SW for state transitions

  // Button handling logic
  (* ASYNC_REG = "TRUE" *) logic [2:0] button_sync;
```

```
// State machine next state logic
always @(posedge clk or negedge CPU_RESETN) begin
  if (!CPU_RESETN) begin
    state <= S0;  // Start in state S0 on reset
  end else if (button_down) begin
    state <= next_state;
  end
end

initial begin
  encoded     = '0;
  digit_point = '1;
end


  // State transition logic
  always @(*) begin
    case (state)
      S0: begin  // C
        next_state = S1;  // Move to A
        Z = 4'hC;        // Output the hex value for C
      end
      S1: begin  // A
        next_state = S2;  // Move to 4
        Z = 4'hA;        // Output the hex value for A
```

```
end
S2: begin  // 4
  next_state = S3;  // Move to 5
  Z = 4'h4;        // Output the hex value for 4
end
S3: begin  // 5
  next_state = S4;  // Move to 7
  Z = 4'h5;        // Output the hex value for 5
end
S4: begin  // 7
  next_state = S5;  // Move to 3
  Z = 4'h7;        // Output the hex value for 7
end
S5: begin  // 3
  next_state = S6;  // Move to F
  Z = 4'h3;        // Output the hex value for 3
end
S6: begin  // F
  next_state = S7;  // Move to 6
  Z = 4'hF;        // Output the hex value for F
end
S7: begin  // 6
  next_state = S8;  // Move to 9
  Z = 4'h6;        // Output the hex value for 6
end
```

```
      S8: begin  // 9
        next_state = S0;  // Loop back to C
        Z = 4'h9;        // Output the hex value for 9
      end
      default: begin
        next_state = S0;  // Default to state C
        Z = 4'h0;        // Default output (could be C as well)
      end
    endcase
  end


 // Seven-segment display encoding
 always @(posedge clk) begin
  if (button_down) begin
    encoded[4] <= Z;
    encoded[0] <= next_state;
  end
  if (!CPU_RESETN) begin
    encoded <= 4'b0000;  // Reset encoded display
  end
 end

endmodule
```

```
// seven_segment.sv
//
// Encapsulate multiple seven segment displays using the cathode driver plus an
// anode driver.
`timescale 1ns/10ps
module seven_segment
 #
 (
  parameter NUM_SEGMENTS = 8,
  parameter CLK_PER      = 10,  // Clock period in ns
  parameter REFR_RATE    = 1000 // Refresh rate in Hz
  )
 (
  input wire                    clk,
  input wire                    reset, // active high reset
  input wire [NUM_SEGMENTS-1:0][3:0] encoded,
  input wire [NUM_SEGMENTS-1:0]      digit_point,
  output logic [NUM_SEGMENTS-1:0]    anode,
  output logic [7:0]                 cathode
  );

 localparam INTERVAL = int'(100000000 / (CLK_PER * REFR_RATE));

 logic [$clog2(INTERVAL)-1:0]       refresh_count;
 logic [$clog2(NUM_SEGMENTS)-1:0]    anode_count;
```

```
logic [NUM_SEGMENTS-1:0][7:0]       segments;


cathode_top ct[NUM_SEGMENTS]
 (
  .clk       (clk),
  .encoded    (encoded),
  .digit_point(digit_point),
  .cathode    (segments)
  );


initial begin
 refresh_count = '0;
 anode_count   = '0;
end


always @(posedge clk) begin
 if (refresh_count == INTERVAL) begin
   refresh_count         <= '0;
   anode_count           <= anode_count + 1'b1;
 end else refresh_count <= refresh_count + 1'b1;
 anode                 <= '1;
 anode[anode_count]      <= '0;
 cathode               <= segments[anode_count];
 if (reset) begin
   refresh_count         <= '0;
```

```
    anode_count          <= '0;
   end
  end

endmodule


// cathode_top.sv
// -------------------------------------
// Drive the cathodes of 7 segment display
// -------------------------------------
//
// input the encoded value from 0-F and generate the cathode signals
`timescale 1ns/10ps
module cathode_top
  (
   input wire        clk,
   input wire [3:0]   encoded,
   input wire        digit_point,
   output logic [7:0] cathode
   );

  always_ff @(posedge clk) begin
    cathode[7] <= digit_point;
    case (encoded)
      4'h0: cathode[6:0] <= 7'b1000000;
```

```
    4'h1: cathode[6:0] <= 7'b1111001;
    4'h2: cathode[6:0] <= 7'b0100100;
    4'h3: cathode[6:0] <= 7'b0110000;
    4'h4: cathode[6:0] <= 7'b0011001;
    4'h5: cathode[6:0] <= 7'b0010010;
    4'h6: cathode[6:0] <= 7'b0000010;
    4'h7: cathode[6:0] <= 7'b1111000;
    4'h8: cathode[6:0] <= 7'b0000000;
    4'h9: cathode[6:0] <= 7'b0010000;
    4'hA: cathode[6:0] <= 7'b0001000;
    4'hB: cathode[6:0] <= 7'b0000011;
    4'hC: cathode[6:0] <= 7'b1000110;
    4'hD: cathode[6:0] <= 7'b0100001;
    4'hE: cathode[6:0] <= 7'b0000110;
    4'hF: cathode[6:0] <= 7'b0001110;
   endcase
  end

endmodule
```