DSD F18
Lab 5 Signal 7-Segment Display and Test Bench

Name: Andrew Quick Lab Session M T W Date_____FALL 2024_____

1. Explain why both the anode and the cathode need to be driven to a 0 to turn on a segment in a particular display?

Driving both the anode and cathode ensures proper current flow through the selected LED segment in displays that share anodes and cathodes across multiple segments, preventing unwanted segments from lighting up.

2. What would you have to do to convert this code into a module that could display 8 different digits all at the same time using the 8 seven segment displays. Remember you do not have separate line connecting each display, but all the segments are connected to the line for "a" and all the "b" segments are connected together and so on for a, b ,c, d, e, f, and g. You do have a unique anode line for each of the 8 digits, but if you turn these all on at the same time then the same thing will be displayed in each digit. So, what technique must you use to display different information on each digit of the 8 digit display? Describe in general terms what you have to do.

In this setup, since all the segment lines are shared, you cannot drive all 8 displays at the same time with different data because the same signal would be sent to all of the displays simultaneously. However, by turning on only one display at a time and cycling through the displays fast enough, you can create the illusion that all the displays are on simultaneously, each showing a different digit.You could make a simple state machine that changes the cathodes to increment from 0 to 9.

3. How would you modify this code to only display the digits 0-9?

To modify the code so that the seven-segment display only shows the digits 0-9, you will need to ensure that any value beyond 9 wraps around to 0. This behavior can be implemented in the counting_buttons.sv file where the increment logic resides. Specifically, in the part of the code where the encoded value is incremented, you can add logic to reset the count after it reaches 9.

4. Paste in your code and test bench, not the configuration file.

```
`timescale 1ns/10ps

module counting_buttons

 #

 (

  parameter MODE        = "HEX", // or "DEC"

  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, // Add one
  input wire                BTNU, // Load SW
//  input wire                BTNR, // Multiply by SW
//  input wire                BTND, // Divide by SW
  input wire                CPU_RESETN,
  input wire [15:0]          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;
 (* ASYNC_REG = "TRUE" *) logic [1:0]reset_sync = '1;
 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;
  logic                     button_down_SW;
//  logic                       button_down_R;
//  logic                       button_down_D;

initial begin
    last_button = 1'b0;       // Use 1'b0 for a single-bit signal
    button      = 1'b0;       // Correct initialization for single-bit signal
    button_down = 1'b0;       // Correct initialization
    button_down_SW = 1'b0;    // Correct initialization for BTNU debounce signal
//    button_down_R = 1'b0;
//    button_down_D = 1'b0;
end

  generate
    if (ASYNC_BUTTON == "SAFE") begin : g_CLOCK
```

```
    (* ASYNC_REG = "TRUE", mark_debug = "true" *) logic [2:0] button_sync;
    logic [2:0] button_sync_SW;  // New shift register for BTNU
    logic [2:0] button_sync_R;  // New shift register for BTNR
    logic [2:0] button_sync_D;  // New shift register for BTND

    always @(posedge clk) begin
     // BTNC handling
     button_down <= '0;
     button_sync <= button_sync << 1 | BTNC;
     if (button_sync[2:1] == 2'b01) button_down <= '1;
     else button_down <= '0;

     // BTNU handling
     button_down_SW <= '0;
     button_sync_SW <= button_sync_SW << 1 | BTNU;
     if (button_sync_SW[2:1] == 2'b01) button_down_SW <= '1;
     else button_down_SW <= '0;

//     // BTNR handling
//     button_down_R <= '0;
//     button_sync_R <= button_sync_R << 1 | BTNR;
//     if (button_sync_R[2:1] == 2'b01) button_down_R <= '1;
//     else button_down_R <= '0;

//     // BTND handling
//     button_down_D <= '0;
//     button_sync_D <= button_sync_D << 1 | BTND;
//     if (button_sync_D[2:1] == 2'b01) button_down_D <= '1;
```

```
//       else button_down_D <= '0;
      end


    end else if (ASYNC_BUTTON == "DEBOUNCE") begin : g_CLOCK
      (* ASYNC_REG = "TRUE", mark_debug = "true" *) logic [2:0] button_sync;
      (* mark_debug = "true" *) logic      counter_en;
      (* mark_debug = "true" *) logic [7:0] counter;

      always @(posedge clk) begin
        button_down <= '0;
        button_sync <= button_sync << 1 | BTNC;
        if (button_sync[2:1] == 2'b01) counter_en <= '1;
        else if (~button_sync[1]) begin
          counter_en <= '0;
          counter    <= '0;
        end
        if (counter_en) begin
          counter <= counter + 1'b1;
          if (&counter) begin
            counter_en <= '0;
            counter    <= '0;
            button_down <= '1;
          end
        end
        if (reset) begin
          counter_en  <= '0;
          counter     <= '0;
          button_down <= '0;
```

```
      end
    end
  end else begin : g_NOCLOCK
    always @(posedge clk) begin
      last_button                       <= button;
      button                           <= BTNC;
      if (BTNC & ~button) button_down <= '1;
      else button_down <= '0;
    end
  end
endgenerate

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

// Reset Synchronizer
always_ff @(posedge clk) reset_sync <= {reset_sync[0], ~CPU_RESETN};
assign reset = reset_sync[1];

  always @(posedge clk) begin
    if (button_down) encoded <= (MODE == "HEX") ? encoded + 1'b1 : dec_inc(encoded);
    if (button_down_SW) encoded <= SW; // Load value from switches
//      if (button_down_R) encoded <= (encoded * SW); // Multiply
//      if (button_down_D && SW != 0) encoded <= (encoded / SW); // Divide, only if SW is not zero
    if (reset) begin
      encoded     <= '0;
```

```
        digit_point <= '1;
      end
    end



  // Decimal increment function
  function [NUM_SEGMENTS-1:0][3:0] dec_inc;
    input [NUM_SEGMENTS-1:0][3:0] din;
    bit [3:0]                 next_val;
    bit                       carry_in;
    carry_in = '1;
    for (int i = 0; i < NUM_SEGMENTS; i++) begin
      next_val = din[i] + carry_in;
      if (next_val > 9) begin
        dec_inc[i] = '0;
        carry_in   = '1;
      end else begin
        dec_inc[i] = next_val;
        carry_in   = '0;
      end
    end // for (int i = 0; i < NUM_SEGMENTS; i++)
  endfunction // dec_inc

endmodule

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

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


// counting_buttons_tb.sv
// Testbench for counting_buttons.sv
```

```
`timescale 1ns/10ps

module counting_buttons_tb;

 // Testbench parameters
 parameter NUM_SEGMENTS = 8;
 parameter CLK_PERIOD = 10; // 100 MHz clock
 parameter TEST_DURATION = 100000; // Test for 100,000 ns

 // Testbench signals
 logic clk;
 logic BTNC;    // Simulate center button
 logic BTNU;    // Simulate upper button (load SW)
 logic CPU_RESETN;
 logic [15:0] SW;
 logic [NUM_SEGMENTS-1:0] anode;
 logic [7:0] cathode;

 // Instantiate the DUT (Device Under Test)
 counting_buttons #(
  .MODE("HEX"),
  .NUM_SEGMENTS(NUM_SEGMENTS),
  .CLK_PER(CLK_PERIOD),
  .REFR_RATE(1000),
  .ASYNC_BUTTON("SAFE")
 ) dut (
  .clk(clk),
  .BTNC(BTNC),
```

```
  .BTNU(BTNU),
  .CPU_RESETN(CPU_RESETN),
  .SW(SW),
  .anode(anode),
  .cathode(cathode)
);

// Generate clock signal
always # (CLK_PERIOD / 2) clk = ~clk;

// Testbench procedure
initial begin
  // Initialize inputs
  clk = 0;
  BTNC = 0;
  BTNU = 0;
  CPU_RESETN = 0;
  SW = 16'h0;

  // Apply reset
  #100;
  CPU_RESETN = 1;

  // Simulate button presses for counting
  #200;
  BTNC = 1; // Simulate button press (increment count)
  #20;
  BTNC = 0; // Button release
```

```
    #500;
    BTNC = 1; // Press again
    #20;
    BTNC = 0; // Release

    #500;
    BTNU = 1; // Simulate BTNU to load SW
    SW = 16'h3A; // Set switches to some value (hex)
    #20;
    BTNU = 0; // Release

    // Add more test scenarios if needed

    // End simulation after test duration
    #TEST_DURATION;
    $stop;
  end

  // Monitor the output values (for debugging purposes)
  initial begin
    $monitor("Time: %t, anode: %b, cathode: %b", $time, anode, cathode);
  end

endmodule
```