DSD F18
Lab 6 Complement and Hex Display

Name: Andrew Quick Date_____10/2/24_____

1. Explain why you might have unwanted display of segments that have been turned off?

We're not using them, don't want to confuse the user or add unnecessary code.

2. As an extra feature you can make you digits travel across the display. Explain how you would do that. Would you modify the sub module or the top module?

To make digits travel across a 7-segment display, I would modify the top module by adding a counter or shift register to cycle through the digits and control which segment is active at each time step. The submodule for the 7-segment decoder would remain unchanged, as its function is purely to convert binary values to segment patterns. Timing control in the top module would handle the digit movement across the display.

3. What would limit the number of digits that you could display by this time division multiplex method?

The number of digits you can display using time-division multiplexing is limited by the refresh rate, which depends on how fast the multiplexing can switch between digits without visible flickering. If the switching speed is too slow, the display will appear to flicker. Additionally, hardware constraints like the clock speed and number of available output pins could also limit the number of digits.

4. What method did you use to take the complement of the input byte?

I used the two's complement method to take the complement of the input byte, which involves inverting the bits (~in) and then adding 1.

5. There are high and low limits on the clock speed for multiplexing these displays. Give at least one consideration that provides a limit for the high side and at least one for the low side.

For the high clock speed limit, the consideration is the response time of the 7-segment display; if the clock is too fast, the display won't have enough time to properly light up before switching to the next digit. On the low side, if the clock is too slow, human eyes will perceive flickering because each digit will not refresh fast enough, breaking the persistence of vision effect.

6. Paste in your code and test bench

```
// counting_buttons.sv


`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];
logic signed [15:0] twos_complement;
assign twos_complement = ~SW;

  always @(posedge clk) begin
```

```
      if (button_down) encoded <= (MODE == "HEX") ? encoded + 1'b1 : dec_inc(encoded);

      if (button_down_SW) encoded <= {twos_complement, SW}; // Concatenate SW (upper 4 bits) and
twos_complement (lower 4 bits)


//       if (button_down_SW) encoded <= SW; // Load value from switches

//       if (button_down_R) encoded <= twos_complement;

//       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 // counting_buttons



// 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
//
// 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




////// Test Bench//////////
`timescale 1ns/10ps
module tb_counting_buttons;

  // Testbench signals
  reg clk;
  reg BTNC;
```

```
reg BTNU;
reg BTNR;
reg CPU_RESETN;
reg [15:0] SW;
wire [7:0] anode;
wire [7:0] cathode;

// Clock generation
initial begin
  clk = 0;
  forever #5 clk = ~clk; // 100MHz clock (10ns period)
end

// DUT instantiation
counting_buttons #(
  .MODE("HEX"),
  .NUM_SEGMENTS(8),
  .CLK_PER(10),
  .REFR_RATE(1000),
  .ASYNC_BUTTON("SAFE")
) uut (
  .clk(clk),
  .BTNC(BTNC),
  .BTNU(BTNU),
  .BTNR(BTNR),
  .CPU_RESETN(CPU_RESETN),
  .SW(SW),
  .anode(anode),
```

```
  .cathode(cathode)
);

// Test scenario
initial begin
  // Initialize inputs
  CPU_RESETN = 0;
  BTNC = 0;
  BTNU = 0;
  BTNR = 0;
  SW = 16'b0;

  // Release reset
  #20;
  CPU_RESETN = 1;

  // Set input to 8-bit value (example: 8'b1111_1010 = FA)
  SW[7:0] = 8'b1111_1010; // FA in hex
  #20;
  BTNU = 1; // Load the value into the display
  #20;
  BTNU = 0;

  // Simulate pressing BTNR to display the 2's complement
  #50;
  BTNR = 1;
  #20;
  BTNR = 0;
```

```
    // Observe the results on the 7-segment display
    #200;
    $finish;
  end

endmodule
```