```
 1: module seven_seg (
 2:     input       [3:0]  bcd,
 3:     output reg [6:0]  segments // Must be reg to set in always block!!
 4: );
 5:
 6:     // Your `always @(*)` block and case block here!
 7:     always @(bcd) begin
 8:         case(bcd)
 9:             4'b0000: segments = 7'b1000000;//0
10:             4'b0001: segments = 7'b1111001;//1
11:             4'b0010: segments = 7'b0100100;//2
12:             4'b0011: segments = 7'b0110000;//3
13:             4'b0100: segments = 7'b0011001;//4
14:             4'b0101: segments = 7'b0010010;//5
15:             4'b0110: segments = 7'b0000010;//6
16:             4'b0111: segments = 7'b1111000;//7
17:             4'b1000: segments = 7'b0000000;//8
18:             4'b1001: segments = 7'b0010000;//9
19:             default: segments = 7'b1111111;//anything else
20:         endcase
21:     end
22:
23: endmodule
```

```
    1: `timescale 1ns/1ns /* This directive (`) specifies simulation <time unit>/<time precision>. */
    2:
    3: module timer #(
    4:     parameter MAX_MS = 2047,            // Maximum millisecond value
    5:     parameter CLKS_PER_MS = 20 // What is the number of clock cycles in a millisecond?
    6: ) (
    7:     input                         clk,
    8:     input                         reset,
    9:     input                         up,
   10:     input  [$clog2(MAX_MS)-1:0] start_value, // What does the $clog2() function do here?
   11:     input                         enable,
   12:     output [$clog2(MAX_MS)-1:0] timer_value
   13: );
   14:
   15:     // Your code here!
   16:
   17:     // Counter for clock cycles (needs enough bits to count to CLKS_PER_MS)
   18:     reg [31:0] clk_counter;
   19:
   20:     // Counter for milliseconds
   21:     reg [$clog2(MAX_MS)-1:0] ms_counter;
   22:
   23:     // Direction flag
   24:     reg count_up;
   25:
   26:     always @(posedge clk) begin
   27:         if (reset == 1) begin
   28:             clk_counter <= 0;
   29:             if (up == 1) begin
   30:                 ms_counter <= 0;
   31:                 count_up <= 1;
   32:             end else begin
   33:                 ms_counter <= start_value;
   34:                 count_up <= 0;
   35:             end
   36:         end else if (enable == 1) begin
   37:             if (clk_counter >= CLKS_PER_MS - 1) begin
   38:                 clk_counter <= 0;
   39:                 if (count_up == 1)
   40:                     ms_counter <= ms_counter + 1;
   41:                 else
   42:                     ms_counter <= ms_counter - 1;
   43:             end else begin
   44:                 clk_counter <= clk_counter + 1;
   45:             end
   46:         end
   47:     end
   48:
   49:     assign timer_value = ms_counter;
   50:
   51:
   52: endmodule
   53:
   54:     /*** Hints (Challenge: delete these hints): ***/
   55:         /*
   56:          * Define 2 count bit vectors, one for counting clock cycles and the other for counting millisec
onds.
   57:          * Make sure that these vectors have an appropriate size given their respective maximum values!
   58:          *
   59:          * Define a register `count_up` to remember whether we should be counting up or down.
   60:          *
   61:          * Make a sequential logic always procedure:
   62:          *  If reset then:
   63:          *    Set the clock-cycle counter to zero.
   64:          *    If up is high:
   65:          *      Set the millisecond counter to 0,
   66:          *      Set count_up to high.
   67:          *    Else:
   68:          *      Set the millisecond counter to start_value,
   69:          *      Set count_up to low.
   70:          *  Else if enable then:
   71:          *    If the clock cycle counter is `CLKS_PER_MS - 1` or greater:
   72:          *      Set clock cycle counter back to 0,
   73:          *      If count_up is high:
   74:          *        Increment the millisecond counter.
   75:          *      Else:
   76:          *        Decrement the millisecond counter.
   77:          *    Else:
   78:          *      Increment the clock cycle counter by 1.
   79:          *
   80:          * Continuously assign timer_value to your milliseconds timer.
   81:          *
   82:          * Note: `CLKS_PER_MS` is the number of clock cycles in a millisecond - calculate this number.
   83:          */
   84:
```

```
    1: module reaction_time_fsm #(
    2:     parameter MAX_MS=2047
    3: )(
    4:     input                               clk,
    5:     input                               button_pressed,
    6:     input        [$clog2(MAX_MS)-1:0] timer_value,
    7:     output logic                        reset,
    8:     output logic                        up,
    9:     output logic                        enable,
   10:     output logic                        led_on
   11: );
   12:
   13:     // Edge detection block here!
   14:     logic button_q0, button_edge;/* FILL-IN VARIABLES */
   15:     always_ff @(posedge clk) begin : edge_detect
   16:         button_q0 <= button_pressed;
   17:     end : edge_detect
   18:     assign button_edge = (button_pressed > button_q0);
   19:     /* Complete remaining block here */
   20:
   21:     // State typedef enum here! (See 3.1 code snippets)
   22:     typedef enum logic [1:0] {S0,S1,S2,S3} state_type;
   23:     state_type current_state, next_state;
   24:
   25:     // always_comb for next_state_logic here! (See 3.1 code snippets)
   26:     // Set the default next state as the current state
   27:     always_comb begin
   28:         case(current_state)
   29:             S0: next_state = (button_edge == 1) ? S1:S0;
   30:             S1: next_state = (timer_value == 0) ? S2:S1;
   31:             S2: next_state = (button_edge == 1) ? S3:S2;
   32:             S3: next_state = (button_edge == 1) ? S0:S3;
   33:             default: next_state = current_state;
   34:         endcase
   35:     end
   36:
   37:
   38:     /* Complete code block here */
   39:
   40:     // always_ff for FSM state variable flip-flops here! (See 3.1 code snippets)
   41:     // Set the current state as the next state (Think about whether a blocking or non-blocking assignmen
t should be used here)
   42:     always_ff @(posedge clk) begin
   43:         current_state <= next_state;
   44:     end
   45:
   46:     /* Complete code block here */
   47:
   48:     // Continuously assign outputs of reset, up, enable and led_on based on the current state here! (See
 3.1 code snippets)
   49:
   50:     /* Complete code block here */
   51:     assign reset = (current_state == S0) ? 1:0 || (current_state == S1)?(timer_value==0):0;
   52:
   53:     assign up = (current_state == S1) ? 1:0;
   54:     assign enable = (current_state == S1 || current_state == S2) ? 1:0;
   55:     assign led_on = (current_state == S2) ? 1:0;
   56:
   57: endmodule
```

```
 1: /*
 2:  * Synchronizers: Double-Register Incoming Data!
 3:  *   When travelling through different clock-domains,
 4:  *   it is good practice to always "double-register" ("double-flop")
 5:  *   the signals once they have crossed over to the new domain
 6:  *   (using the new domain's clock signal). This is called a "Synchronizer".
 7:  *
 8:  *   This prevents metastability, which can cause serious failures.
 9: */
10: module synchroniser (input clk, x, output y);
11:     reg x_q0, x_q1;
12:     always @(posedge clk)
13:     begin
14:         x_q0 <= x;     // Flip-flop #1
15:         x_q1 <= x_q0; // Flip-flop #2
16:     end
17:     assign y = x_q1;
18: endmodule
19:
```

```
    1: module top_level (
    2:     input         CLOCK_50,              // DE2-115's 50MHz clock signal
    3:     input  [3:0]  KEY,                   // The 4 push buttons on the board
    4:     output [17:0] LEDR,                  // 18 red LEDs
    5:     output [6:0]  HEX0, HEX1, HEX2, HEX3 // Four 7-segment displays
    6: );
    7:
    8:     // Intermediate wires: (DO NOT EDIT WIRE NAMES!)
    9:     wire        timer_reset, timer_up, timer_enable, button_pressed;
   10:     wire [10:0] timer_value, random_value;
   11:
   12:     // First module instantiated for you as an example:
   13:     timer           u_timer         (// Inputs:
   14:                                      .clk(CLOCK_50),
   15:                                      .reset(timer_reset),
   16:                                      .up(timer_up),
   17:                                      .enable(timer_enable),
   18:                                      .start_value(random_value),
   19:                                      // Outputs:
   20:                                      .timer_value(timer_value));
   21:
   22:     // Add remaining module instantiations here!
   23:     rng u_rng (
   24:         .clk(CLOCK_50),
   25:         //Outputs
   26:         .random_value(random_value)
   27:     );
   28:
   29:     //Debounce
   30:     debounce #(.DELAY_COUNTS(1)) u_debounce (
   31:         //Inputs
   32:         .clk(CLOCK_50),
   33:         .button(~KEY[0]),
   34:         //outputs
   35:         .button_pressed(button_pressed)
   36:     );
   37:
   38:     //reaction time fsm
   39:     reaction_time_fsm u_reaction_time_fsm (
   40:
   41:         //Inputs
   42:         .clk(CLOCK_50),
   43:         .button_pressed(button_pressed),
   44:         .timer_value(timer_value),
   45:         //outputs
   46:         .reset(timer_reset),
   47:         .up(timer_up),
   48:         .enable(timer_enable),
   49:         .led_on(LEDR[0])
   50:     );
   51:
   52:     //displays
   53:     display u_display (
   54:         //Inputs
   55:         .clk(CLOCK_50),
   56:         .value(timer_value),
   57:         //outputs
   58:         .display0(HEX0),
   59:         .display1(HEX1),
   60:         .display2(HEX2),
   61:         .display3(HEX3)
   62:     );
   63:
   64: endmodule
```

```
    1: module display (
    2:     input         clk,
    3:     input  [10:0] value,
    4:     output [6:0]  display0,
    5:     output [6:0]  display1,
    6:     output [6:0]  display2,
    7:     output [6:0]  display3
    8: );
    9:     /*** FSM Controller Code: ***/
   10:     enum { Initialise, Add3, Shift, Result } next_state, current_state = Initialise; // FSM states.
   11:
   12:     logic init, add, done; // FSM outputs.
   13:
   14:     logic [3:0] count = 0; // Use this to count the 11 loop iterations.
   15:
   16:     /*** DO NOT MODIFY THE CODE ABOVE ***/
   17:
   18:     //TODO
   19:     always_comb begin : double_dabble_fsm_next_state_logic
   20:         case (current_state)
   21:             Initialise: next_state = Add3;
   22:             Add3: next_state = Shift;
   23:             Shift:      next_state = count == 10 ? Result : Add3; // After 11 iterations, exit out of th
e loop
   24:             Result: next_state = Initialise;
   25:             default:    next_state = Initialise;
   26:         endcase
   27:     end
   28:
   29:     //TODO
   30:     always_ff @(posedge clk) begin : double_dabble_fsm_ff
   31:         // Set state to next state.
   32:         current_state <= next_state;
   33:         // To implement the loop, we use count to count the iterations.
   34:         // Increment count if in the Shift state.
   35:         if (current_state == Shift) begin
   36:             count <= count + 1;
   37:         end
   38:
   39:         // Set count to zero if in the Result state.
   40:         if (current_state == Result) begin
   41:             count <= 0;
   42:         end
   43:
   44:     end
   45:     //TODO
   46:     always_comb begin : double_dabble_fsm_output
   47:         // Assign init, add and done based on the current state.
   48:         init = (current_state == Initialise);
   49:         add  = (current_state == Add3);
   50:         done = (current_state == Result);
   51:     end
   52:
   53:
   54:
   55:
   56:
   57:     /*** DO NOT MODIFY THE CODE BELOW ***/
   58:
   59:     logic [3:0] digit0, digit1, digit2, digit3;
   60:
   61:     //// Seven-Segment Displays
   62:     seven_seg u_digit0 (.bcd(digit0), .segments(display0));
   63:     seven_seg u_digit1 (.bcd(digit1), .segments(display1));
   64:     seven_seg u_digit2 (.bcd(digit2), .segments(display2));
   65:     seven_seg u_digit3 (.bcd(digit3), .segments(display3));
   66:
   67:     // Algorithm RTL:  (completed no changes required - see dd_rtl.png for a representation of the code
below but for 2 BCD digits.)
   68:     // essentially a 27-bit long, 1-bit wide shift-register, starting from the 11 input bits through to
the 4 bits of each BCD digit (4*4=16, 16+11=27).
   69:     // We shift in the Shift state, add 3 to BCD digits greater than 4 in the Add3 state, and initialise
 the shift-register values in the Initialise state.
   70:     logic [3:0]  bcd0, bcd1, bcd2, bcd3; // Do NOT change.
   71:     logic [10:0] temp_value; // Do NOT change.
   72:
   73:     always_ff @(posedge clk) begin : double_dabble_shiftreg
   74:         if (init) begin // Initialise: set bcd values to 0 and temp_value to value.
   75:             {bcd3, bcd2, bcd1, bcd0, temp_value} <= {16'b0, value}; // A nice usage of the concat operat
or on both LHS and RHS!
   76:         end
   77:         else begin
   78:             if (add) begin // Add3: 3 is added to each bcd value greater than 4.
   79:                 bcd0 <= bcd0 > 4 ? bcd0 + 3 : bcd0;  // Conditional operator.
   80:                 bcd1 <= bcd1 > 4 ? bcd1 + 3 : bcd1;
   81:                 bcd2 <= bcd2 > 4 ? bcd2 + 3 : bcd2;
```

```
   82:                 bcd3 <= bcd3 > 4 ? bcd3 + 3 : bcd3;
   83:             end
   84:             else begin // Shift: essentially everything becomes a shift-register
   85:                 {bcd3, bcd2, bcd1, bcd0, temp_value} <= {bcd3, bcd2, bcd1, bcd0, temp_value} << 1; // Co
ncat operator makes this easy!
   86:             end
   87:         end
   88:     end
   89:
   90:     always_ff @(posedge clk) begin : double_dabble_ff_output
   91:         // Need to 'flop' bcd values at the output so that intermediate calculations are not seen at the
 output.
   92:         if (done) begin  // Only take bcd values when the algorithm is done!
   93:             digit0 <= bcd0;
   94:             digit1 <= bcd1;
   95:             digit2 <= bcd2;
   96:             digit3 <= bcd3;
   97:         end
   98:     end
   99:
  100: endmodule
```

```
 1: module debounce #(
 2:   parameter DELAY_COUNTS = 2500 // 50us with clk period 20ns totals in ____ counts
 3: ) (
 4:     input clk, button,
 5:     output reg button_pressed
 6: );
 7:
 8:   // Use a synchronizer to synchronize `button`.
 9:   wire button_sync; // Output of the synchronizer. Input to your debounce logic.
10:   synchroniser button_synchroniser (.clk(clk), .x(button), .y(button_sync));
11:
12:   // Note: Use the synchronized `button_sync` wire as the input signal to the debounce logic.
13:
14:
15:   /*** Fill in the following scaffold: ***/
16:   reg prev_button;
17:   reg [31:0] count;
18:   // Set the count flip-flop:
19:   always @(posedge clk) begin
20:       if (button_sync != prev_button) begin
21:         count <= 0;
22:       end
23:       else if (count == DELAY_COUNTS) begin
24:         count <= count ;
25:       end
26:       else begin
27:         count <= count + 1;
28:       end
29:   end
30:
31:   // Set the prev_button flip-flop:
32:   always @(posedge clk) begin
33:     if (button_sync != prev_button) begin
34:       prev_button <= button_sync;
35:     end
36:     else begin
37:       prev_button <= prev_button;
38:     end
39:   end
40:
41:   //reg button_pressed;
42:   // Set the button_pressed flip-flop:
43:   always @(posedge clk) begin
44:     if (button_sync == prev_button && count == DELAY_COUNTS) begin
45:       button_pressed <= prev_button;
46:     end
47:     else begin
48:       button_pressed <= button_pressed;
49:     end
50:   end
51:
52: endmodule
53:
54:
```

```
    1: module rng #(
    2:     parameter OFFSET=200,
    3:     parameter MAX_VALUE=1223,
    4:     parameter SEED= 156/*Fill-In*/ // Choose a random number seed here!
    5: ) (
    6:     input clk,
    7:     output [$clog2(MAX_VALUE)-1:0] random_value // 11-bits for values 200 to 1223.
    8: );
    9:     reg [10:1] lfsr; // The 10-bit Linear Feedback Shift Register. Note the 10 down-to 1. (No bit-0, we
count from 1 in this case!)
   10:
   11:     // Initialise the shift reg to SEED, which should be a non-zero value:
   12:     initial lfsr = SEED;
   13:
   14:     // Set the feedback:
   15:     wire feedback;
   16:     assign feedback = lfsr[7] ^ lfsr[10];/* FILL-IN */
   17:
   18:     // Put shift register logic here (use an always @(posedge clk) block):
   19:     //    Make sure to shift left from bit 1 (LSB) towards bit 10 (MSB).
   20:     always @(posedge clk) begin
   21:         lfsr <= {lfsr[9:1], feedback};
   22:     end
   23:
   24:     // Your code here!
   25:
   26:     // Assign random_value to your LSFR output + OFFSET to acheive the range 200 to 1223. Use continuous
 assign!
   27:     assign random_value = OFFSET + lfsr;
   28: endmodule
   29:
```