EE405(B), Fall 2022

Electronics Design Lab. <Advanced Digital System Design>

- Student Names: Hyemin Lee
- Student IDs: 20190533
- LAB Project Number: Lab 2

1. Goal / Specification

A. The ultimate goal of this lab is to design mental math master using (1) buttons for input, (2) LEDs for output, (3) ALU for arithmetic and bitwise operation, (4) LFSR for pseudo-random generation, and (5) FSM for signal control.

B. Buttons & LEDs

i. Buttons, BT0, BT1, BT2, and BT3, were used provide input to the mental math master. Each button was named as *rst, sel, ge,* and *lt*, respectively.

ii. LEDs, LD0 (LSB), LD1, LD2, and LD3 (MSB), were used to show the result of the system as output.

C. ALU (Arithmetic-Logic Unit)

i. ALU is a combinational digital circuit that computes arithmetic and bitwise operations on binary numbers. It is one of the fundamental building blocks of many types of computing devices.

ii. It receives two operands – op1 and op2 – and one opcode as an input. According to the opcode, it calculates the proper result and return it as an output. The proper calculation for each opcode is stated in below table. If the previous result does not exist, 0 was applied instead.

Opcode	Operation	Equation
00_2 (0)	Bitwise AND	result = (op1 & op2) + previous_result
01_2 (1)	Multiplication	result = (op1 * op2) + previous_result
10_2 (2)	Addition	result = (op1 + op2) + previous_result
11_2 (3)	Subtraction	result = (op1 - op2) + previous_result

D. LFSR (Linear-Feedback Shift Register)

i. LFSR is a shift register where its input bit is a linear function of its previous state. In this lab, we used a simple LFSR consists of a 4-bit shift register whose input bit is driven by the XOR gate of the two LSB bits of the shift register values.

ii. The initial value, seed, was inserted to initialize the LFSR. It returns shifted register value with valid sign when shift input is inserted.

E. FSM (Finite State Machine)

i. FSM is a type of computation model that can be used to simulate sequential logic and some computer programs. FSMs are made of one or more states,

and transfers in between states directed by the inputs. In moore machine, output is placed on states, while that of mealy machine is placed on transitions.

ii. FSM should control the entire mental math master system to operate as followings. Basically, there are three states in the system: reset, initialization, and accumulation.

1. The system starts with the reset state. In this state, the system resets all the parameters and does nothing, waiting for the first input of *sel*. During operation, the system turns back to this state whenever *rst* is pressed.

2. In the initialization state, the series of three *sel* input generates op1, opcode, and op2 sequentially and shows them by LEDs. Additional *sel* input enables ALU to calculate the result. Here, the user also calculates the result by mental math and press *ge* or *lt* to check the answer.

3. If the user presses additional *sel*, the system goes into the accumulation state. The series input of *sel* generates opcode and op2, and calculated result was accumulated to the previous result. This state repeats until the user presses *rst*. Additionally, if unwanted *ge* or *lt* input was made, exception sign was showed by LEDs and returns to original state with the input of *sel*.

2. Architecture / Design

A. Description

Our mental math master consists of a big module(mental_math_master), and the module 'mental_math_master' makes connections between four modules(ALU_inst, LSFR_opcode, LSFR_data, controller_inst). However, 'LSFR_opcode' and 'LSFR_data' have same function, so we only need to implement ALU, LSFR, and controller.

(A-1) ALU
Depending on the value of 'opcode_in', ALU does corresponding calculations which was explained in goal/specification. One thing different from traditional ALU is that it calculates like 'res_out = res_out + (operand1_in + operand2_in)'(when opcode_in = 00).

(A-2) LSFR
LSFR is used to make both opcode(BIT_WIDTH = 2) and operand(BIT_WIDTH = 4), so depending on the BIT_WIDTH, LSFR acts differently. If the number of bits of seed_in is less than BIT_WIDTH, LSFR will give undefined value.

(A-3) controller
controller itself controls all input and output going to and coming from LSFR and ALU modules. Controller gives led output and does some actions according to FSM we have designed. The most important role of controller is to act only when button is on or off. In other words, controller does not get all button inputs every clock cycles. This is because when the button is pressed, thousands of clock cycles

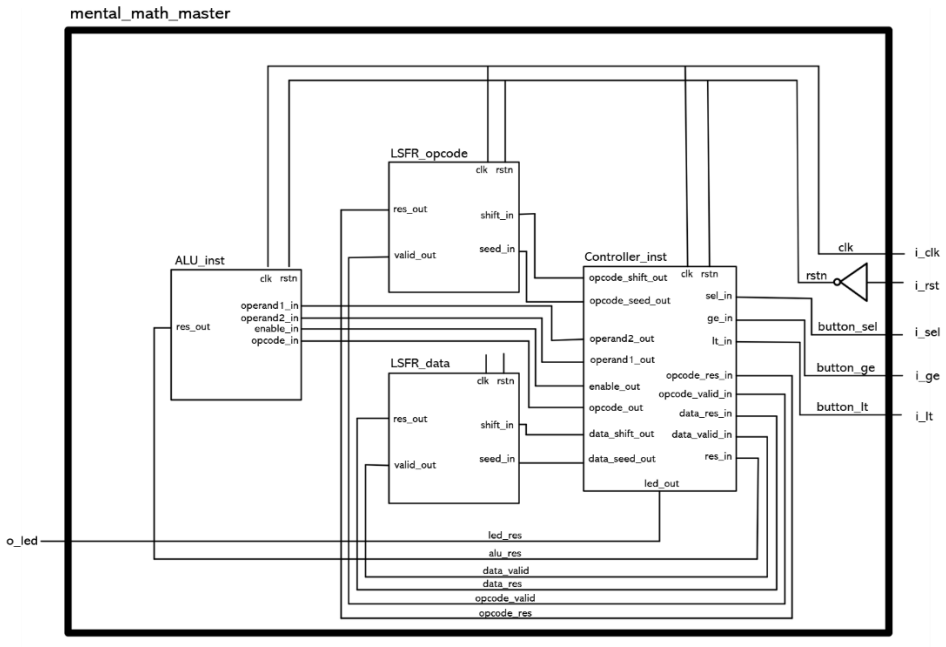
are passed. Therefore, in addition to FSM which describes the action and led outputs of controller, we designed FSM to get one input during thousands of clock cycles and added switch debouncing function.

(A-3-1) general control of LSFR and ALU.
The action of LSFR and ALU is controlled by data_shift_out, opcode_shift_out, and enable_out. If shift_out or enable_out is turned on, they are turned off at next clock cycle in order to LSFR or ALU is activated only once.

(A-3-2) switch debouncing
The idea of switch debouncing is to make our clock cycles longer so that buttons have time to stabilize their state(on/off). We defined register 'timer', which increase by one every clock cycles. When timer = 1, 'real_buttons_buffer' updates their values(sel_in, ge_in, lt_in), and contains it until our timer reaches some values(it can be 100,000). After that, 'real_buttons_buffer' gives their values to 'real_buttons_nxt'.

(A-3-3) button FSM
After switch debouncing, the button value is stabilized to on or off. However, even though 'real_buttons_nxt' is updated every thousand times of clock cycles, it is still short. Therefore, even though we pressed the button once, 'real_buttons_nxt' can be updated more than one time. This causes mental math mater to get several inputs from the button we pressed. To prevent this, button FSM is introduced. By applying button FSM, button input is one only once after button is pressed.

B. Hardware Block Diagram (Use Visio, at least Power point, etc., Don't draw by hand)



C. FSM

i. FSM consists of 17 states(RESET, RESET_SEL1, RESET_SEL1_WAIT, RESET_SEL2, RESET_SEL2_WAIT, RESET_SEL3, RESET_SEL3_WAIT, COMPARE, COMPARE_WAIT,

COMPARE_GE, COMPARE_LT, COMPARE_SEL1, COMPARE_SEL1_WAIT, COMPARE_SEL2, COMPARE_SEL2_WAIT, EXCEPTION_SEL1, and EXCEPTION_SEL2)

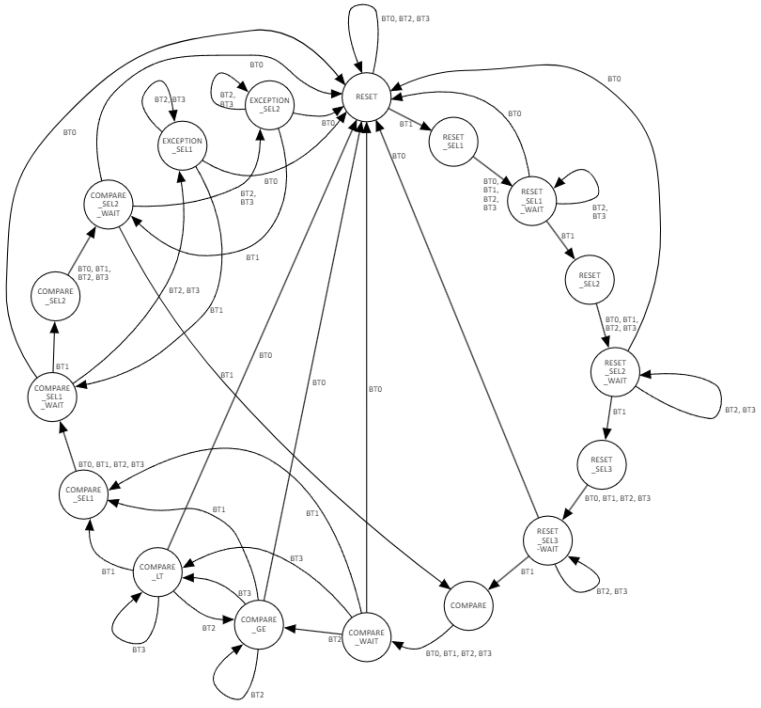
ii. There are four inputs: BT0(reset), BT1(sel), BT2(ge), BT3(lt).

iii. Output is placed on states. That is, FSM is Moore type.

iv. Output(LED) and actions corresponding to each state

State	Output (Action)
RESET	Turn off LED
RESET_SEL1	Make random number from LFSR
RESET_SEL1_WAIT	Save the number as operand1 Show the number with blue LED
RESET_SEL2	Make random number from LFSR
RESET_SEL2_WAIT	Save the lower 2 bits as operation Show the number with green LED
RESET_SEL3	Make random number from LFSR
RESET_SEL3_WAIT	Save the number as operand2 Show the number with blue LED
COMPARE	Calculate the result
COMPARE_WAIT	Turn off the LED
COMPARE_GE	If correct green LEDs, otherwise red LEDs
COMPARE_LT	If correct green LEDs, otherwise red LEDs
COMPARE_SEL1	Insert prev_res to operand1_ff Make random number from LFSR
COMPARE_SEL1_WAIT	Save the lower 2 bits as operation Show the number with green LED
COMPARE_SEL2	Make random number from LFSR Show the number with blue LED
COMPARE_SEL2_WAIT	Save the number as operand2 Show the number with blue LED
EXCEPTION_SEL1	Show G-R-G-R LED
EXCEPTION_SEL2	

v. FSM diagram



D. Include additional information you want to describe.

- Additional description of button FSM

i. Button FSM consists of two states: BT_OFF and BT_ON.

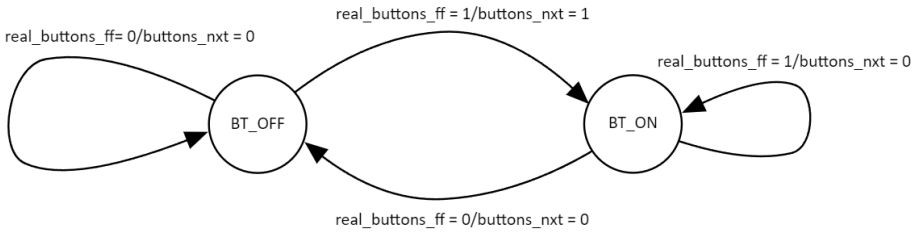
ii. inputs are the value of ‘real_buttons_ff’

iii. outputs are final inputs which is used in main FSM described in previous. Final inputs are stored in ‘buttons_nxt’.

iv. note that ‘state’ represents the previous updated value and ‘input’ represents the new updated value.

v. Output is placed on transitions. That is, FSM is Mealy type.

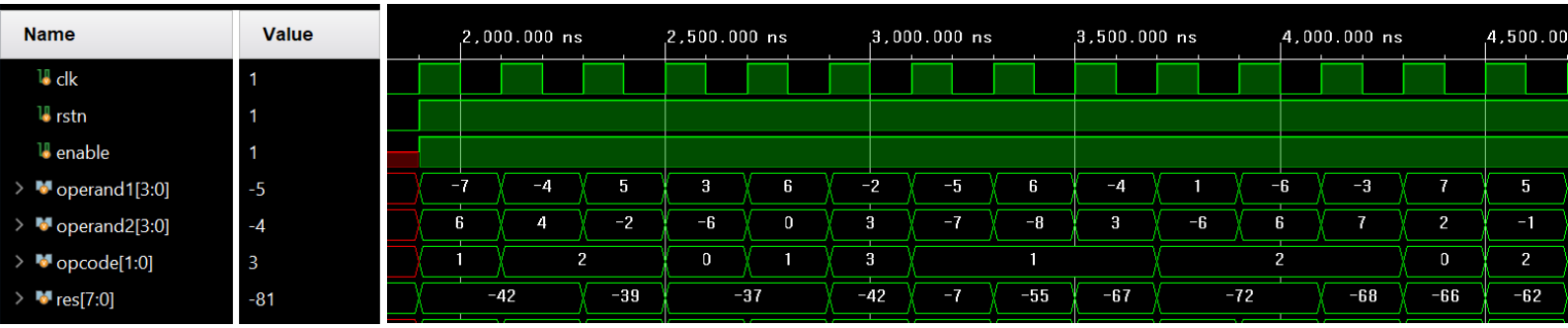
vi. FSM diagram



3. Experiment Results

A. Wave form

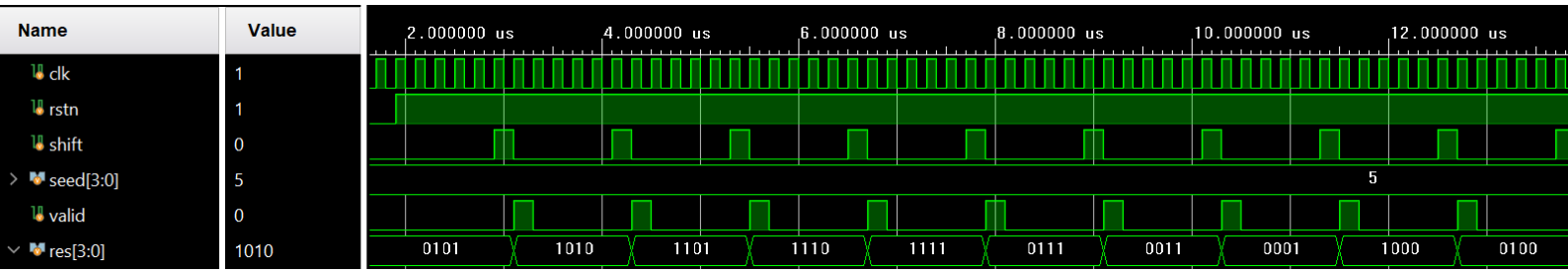
i. Problem 2A) ALU



1. As you can see in the above figure, our ALU successfully executes computation of two operands.

2. Since the above result is captured from the first week, ALU returns the result in the same cycle as combinational circuit and saves the previous result in register internally. However, in the second week (the implementation of entire system), it was modified to return the value at the next cycle without using the internal register.

ii. Problem 2B) LFSR

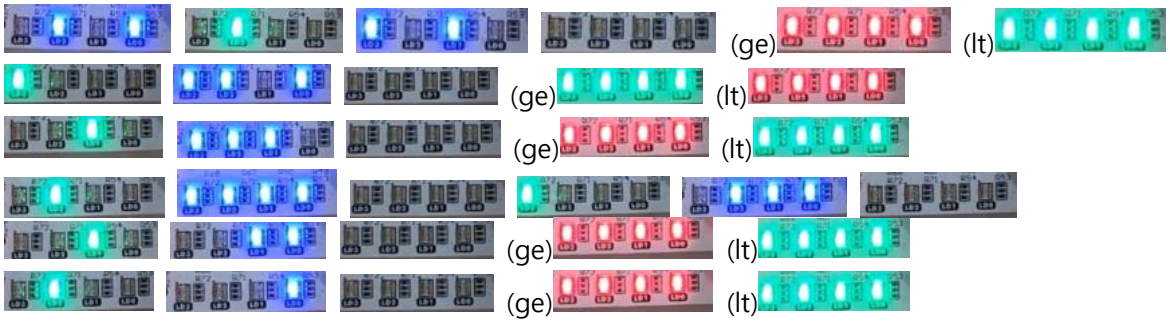


1. As you can see in the above figure, our LFST successfully generates pseudo-random number. The seed was 0101_2.

B. Demo Picture

i. Problem 2C) Mental Math Master

1. With the successful implementation, if we press the *sel* continuously, the result shows like below sequentially.



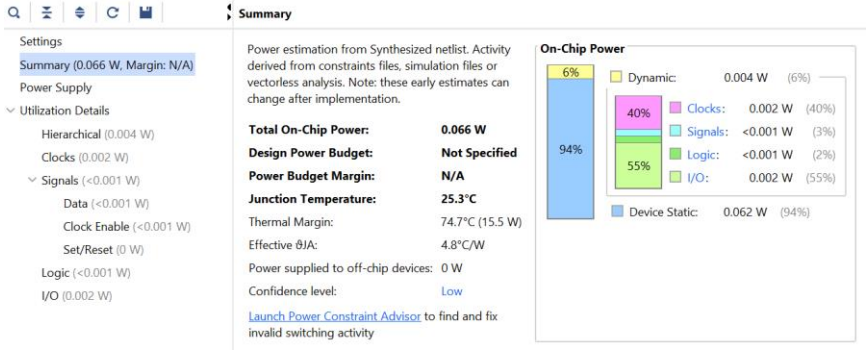
2. Handling exceptional *ge* and *lt* is also properly implemented.

C. FPGA report

i. Estimate Utilization & Power Analysis

1. Utilization and power reports are showed like below. The utilization report analyzed the number of components used in the synthesis model, and the power report analyzed that of power consumption.

Name	Slice LUTs (20800)	Slice Registers (41600)	Bonded IOB (210)	BUFGCTRL (32)
mental_math_master	104	97	17	1
ALU_inst (ALU)	4	8	0	0
controller_inst (controller)	96	81	0	0
LSFR_data (LSFR__parameterized0)	2	5	0	0
LSFR_opcode (LSFR)	2	3	0	0



4. Discussion

A. Explain the two major improvements of your design that makes your circuit design efficient.

i. I used individual FSM for each button to implement the system simply. Button FSMs provided only a single clock input to the main system when the button is pressed. If I didn't used button FSMs, the main FSM would be extremely complicated resulting large amount of LUTs and registers.

ii. The same two LFSR was used to create operands and opcode. Since I used identical LFSRs, time and effort needed for synthesizing and implementing multiple modules can be reduced.

B. Explain the effect of the bit length of an accumulation register with the proper experimental result.

i. Bit length of an accumulation register should be long enough to return exact result. This is because, if the bit length is short, overflow may occur during addition of results. Overflow will change the result to wrong value whose MSB is omitted.

C. Describe the throughput of your design, including the average throughput and utilization (avg. throughput / theoretical peak throughput) of the ALU. Discuss how you can increase the utilization of the ALU.

i. Throughput is defined as the number of processes executed in a given amount of time. We can calculate the throughput by dividing the processor speed – 10MHz in our case – by the number of cycles consumed.

ii. Theoretical peak throughput of ALU is $\frac{10\text{MHz}}{2\text{ cycles}} = 5MHz/cycle$ since 1 cycle is needed for inserting inputs and 1 cycle for resulting output. In our system, however, the inputs of ALU - op1, op2, and opcode - are made from LFSR which cause additional cycles to consume. LFSR requires a single cycle to make single pseudo-random number. Therefore, the average throughput of the ALU is $\frac{10\text{MHz}}{5\text{ cycles}} = 2\text{MHz/cycle}$ and the utilization is $\frac{2MHz/cycle}{5MHz/cycle} = 0.4$.

iii. The utilization can be increased by using multiple LFSR to generate input. If we use three LFSR and make all the random numbers in a single cycle, the average throughput and utilization can be increased up to $\frac{10\text{MHz}}{3\ cycles} = 3.33MHz/cycle$ and $\frac{3.33MHz/cycle}{5MHz/cycle} = 0.67$, respectively.

5. Appendix: Include your code here

A. ALU.sv

```
////////////////////////////////////////////////////////////////////////
//  EE405(B)
//
//  Name: ALU.sv
//  Description:
//    This module describes arithmetic logic unit that contains accumulation
//    registers.
//
////////////////////////////////////////////////////////////////////////

`timescale 1ns / 1ps

module ALU
#(
   parameter BIT_WIDTH                                          = 4
)
(
   input  logic                                                clk,
   input  logic                                                rstn,
   input  logic                                                enable_in,     //
enable signal that enables the ALU module
   input  logic signed [2*BIT_WIDTH-1:0]                            operand1_in,
   input  logic signed [BIT_WIDTH-1:0]                           operand2_in,
   input  logic [1:0]                                          opcode_in,
   output logic signed [2*BIT_WIDTH-1:0]                                 res_out
// result
);
   reg signed [2*BIT_WIDTH-1:0] res_out_ff, res_out_nxt;

   assign res_out = res_out_ff;

   always_ff @ (posedge clk) begin
      res_out_ff <= res_out_nxt;
   end

   always_comb begin
      if (~rstn) begin
         res_out_nxt = 0;
      end else begin
         if (enable_in) begin
            case (opcode_in)
               2'b00: begin // AND
                  res_out_nxt = (operand1_in & operand2_in) + res_out_ff;
               end
               2'b01: begin // multiplication
                  res_out_nxt = (operand1_in * operand2_in) + res_out_ff;
               end
               2'b10: begin // addition
                  res_out_nxt = (operand1_in + operand2_in) + res_out_ff;
               end
               2'b11: begin // subtraction
                  res_out_nxt = (operand1_in - operand2_in) + res_out_ff;
               end
            endcase
```

```
            end else begin
                // Default output
                res_out_nxt = res_out_ff;
            end
        end
    end
endmodule
```

B. LSFR.sv

```
//////////////////////////////////////////////////////////////////////////
//  EE405(B)
//
//  Name: LSFR.sv
//  Description:
//     This module describes linear-feedback shift register.
//
//////////////////////////////////////////////////////////////////////////

`timescale 1ns / 1ps

module LSFR
#(
    parameter BIT_WIDTH                                     = 4
)
(
    input  logic                                          clk,
    input  logic                                          rstn,
    input  logic                                          shift_in,
    input  logic [BIT_WIDTH-1:0]                            seed_in,
    output logic                                          valid_out,
    output logic [BIT_WIDTH-1:0]                            res_out
);

    // write your code below:
    reg [BIT_WIDTH-1:0] res_out_ff, res_out_nxt;
    reg valid_out_ff, valid_out_nxt;

    assign res_out = res_out_ff;
    assign valid_out = valid_out_ff;

    always_ff @ (posedge clk) begin
        res_out_ff <= res_out_nxt;
        valid_out_ff <= valid_out_nxt;
    end

    always_comb begin
        if (~rstn) begin
            res_out_nxt = seed_in;
            valid_out_nxt = 0;
        end else begin
            if (shift_in) begin
                if   (BIT_WIDTH   ==   4)   res_out_nxt   =   {(res_out_ff[1]   ^
res_out_ff[0]),res_out_ff[3],res_out_ff[2],res_out_ff[1]};
                else        res_out_nxt        =        {(res_out_ff[1]        ^
res_out_ff[0]),res_out_ff[1]};
                valid_out_nxt = 1;
            end else begin
```

```
            // Default values
            res_out_nxt = res_out_ff;
            valid_out_nxt = 0;
         end
      end
   end
endmodule
```

C. controller.sv

```
////////////////////////////////////////////////////////////////////////////
//  EE405(B)
//
//  Name: controller.sv
//  Description:
//    This module describes controller of mental_math_master.
//
////////////////////////////////////////////////////////////////////////////

`timescale 1ns / 1ps

module controller
#(
   parameter BIT_WIDTH                                          = 4
)
(
   input logic                                                 clk,
   input logic                                                 rstn,
   input logic                                                 sel_in,
   input logic                                                 ge_in,
   input logic                                                 lt_in,

   output logic                                                opcode_shift_out,
   output                       logic                         [(BIT_WIDTH-2)-1:0]
opcode_seed_out,
   input  logic                                                opcode_valid_in,
   input  logic [(BIT_WIDTH-2)-1:0]                               opcode_res_in,

   output logic                                                data_shift_out,
   output logic [BIT_WIDTH-1:0]                                  data_seed_out,
   input  logic                                                data_valid_in,
   input  logic [BIT_WIDTH-1:0]                                  data_res_in,

   output logic                                                enable_out,
   output logic [2*BIT_WIDTH-1:0]                                   operand1_out,
   output logic [BIT_WIDTH-1:0]                                  operand2_out,
   output logic [1:0]                                          opcode_out,
   input  logic [2*BIT_WIDTH-1:0]                                res_in,

   output logic [3:0][2:0]                                      led_out
);

   // Buttons
   logic [16:0] timer, timer_nxt;

   enum logic [1:0] {
      BT_OFF,
      BT_ON
```

```
   } sel_state_ff, sel_state_nxt, ge_state_ff, ge_state_nxt, lt_state_ff,
lt_state_nxt;

   struct packed{
      logic sel_in;
      logic ge_in;
      logic lt_in;
   } real_buttons_ff, real_buttons_nxt, real_buttons_buffer, buttons_ff,
buttons_nxt;

   // Controller FSM
   enum logic [4:0] {
      RESET,
      RESET_SEL1,
      RESET_SEL1_WAIT,
      RESET_SEL2,
      RESET_SEL2_WAIT,
      RESET_SEL3,
      RESET_SEL3_WAIT,
      COMPARE,
      COMPARE_WAIT,
      COMPARE_GE,
      COMPARE_LT,
      COMPARE_SEL1,
      COMPARE_SEL1_WAIT,
      COMPARE_SEL2,
      COMPARE_SEL2_WAIT,
      EXCEPTION_SEL1,
      EXCEPTION_SEL2
   } state_ff, state_nxt;

   struct packed{
      logic data_shift_out;
      logic opcode_shift_out;
      logic enable_out;
      logic [2*BIT_WIDTH-1:0] operand1;
      logic [BIT_WIDTH-1:0] operand2;
      logic [1:0] opcode_out;
      logic [3:0] led_red, led_green, led_blue;
   } regs_ff, regs_nxt;

   // output assignment
   assign data_shift_out = regs_ff.data_shift_out;
   assign opcode_shift_out = regs_ff.opcode_shift_out;
   assign enable_out = regs_ff.enable_out;
   assign operand1_out = regs_ff.operand1;
   assign operand2_out = regs_ff.operand2;
   assign opcode_out = regs_ff.opcode_out;
   assign data_seed_out = 4'b0101;
   assign opcode_seed_out = 2'b10;
   assign    led_out[0]    =    {regs_ff.led_red[0],    regs_ff.led_green[0],
regs_ff.led_blue[0]};
   assign    led_out[1]    =    {regs_ff.led_red[1],    regs_ff.led_green[1],
regs_ff.led_blue[1]};
   assign    led_out[2]    =    {regs_ff.led_red[2],    regs_ff.led_green[2],
regs_ff.led_blue[2]};
   assign    led_out[3]    =    {regs_ff.led_red[3],    regs_ff.led_green[3],
regs_ff.led_blue[3]};
```

```
    // Button Debouncing
    always_ff @ (posedge clk) begin
        timer <= timer_nxt;
        real_buttons_ff <= real_buttons_nxt;
    end
    always_comb begin
        if (~rstn) begin
            timer_nxt = 0;
            real_buttons_nxt = {$bits(regs_ff){1'b0}};
        end else begin
            if (timer == 1) begin
                real_buttons_buffer.sel_in = sel_in;
                real_buttons_buffer.ge_in = ge_in;
                real_buttons_buffer.lt_in = lt_in;
                timer_nxt = timer + 1;
            end else if (timer > 1) begin                          // CHANGE THE
TIMER VALUE
                timer_nxt = 0;
            end else timer_nxt = timer + 1;
            real_buttons_nxt = real_buttons_buffer;
        end
    end

    // Button FSM
    always_ff @ (posedge clk) begin
        sel_state_ff <= rstn ? sel_state_nxt : BT_OFF;
        ge_state_ff <= rstn ? ge_state_nxt : BT_OFF;
        lt_state_ff <= rstn ? lt_state_nxt : BT_OFF;
        buttons_ff <= buttons_nxt;
    end
    // Button FSM - sel
    always_comb begin
        sel_state_nxt = sel_state_ff;
        case (sel_state_ff)
            BT_OFF: begin
                if (real_buttons_ff.sel_in == 1) begin
                    sel_state_nxt = BT_ON;
                    buttons_nxt.sel_in = 1;
                end else begin
                    buttons_nxt.sel_in = 0;
                end
            end
            BT_ON: begin
                if (real_buttons_ff.sel_in == 0) begin
                    sel_state_nxt = BT_OFF;
                end
                buttons_nxt.sel_in = 0;
            end
        endcase
    end
    // Button FSM - ge
    always_comb begin
        ge_state_nxt = ge_state_ff;
        case (ge_state_ff)
            BT_OFF: begin
                if (real_buttons_ff.ge_in == 1) begin
                    ge_state_nxt = BT_ON;
```

```
                buttons_nxt.ge_in = 1;
            end else begin
                buttons_nxt.ge_in = 0;
            end
        end
        BT_ON: begin
            if (real_buttons_ff.ge_in == 0) begin
                ge_state_nxt = BT_OFF;
            end
            buttons_nxt.ge_in = 0;
        end
    endcase
end
// Button FSM - lt
always_comb begin
    lt_state_nxt = lt_state_ff;
    case (lt_state_ff)
        BT_OFF: begin
            if (real_buttons_ff.lt_in == 1) begin
                lt_state_nxt = BT_ON;
                buttons_nxt.lt_in = 1;
            end else begin
                buttons_nxt.lt_in = 0;
            end
        end
        BT_ON: begin
            if (real_buttons_ff.lt_in == 0) begin
                lt_state_nxt = BT_OFF;
            end
            buttons_nxt.lt_in = 0;
        end
    endcase
end

// Controller FSM
always_ff @ (posedge clk) begin
    state_ff <= state_nxt;
    regs_ff <= regs_nxt;
end

always_comb begin
    if (~rstn) begin
        state_nxt = RESET;
        regs_nxt = {$bits(regs_ff){1'b0}};
    end else begin
    state_nxt = state_ff;

    regs_nxt.data_shift_out = 0;
    regs_nxt.opcode_shift_out = 0;
    regs_nxt.enable_out = 0;
    regs_nxt.led_red = 0;
    regs_nxt.led_green = 0;
    regs_nxt.led_blue = 0;


    case (state_ff)
        RESET: begin
            regs_nxt.led_red = 4'b0;
```

```
            regs_nxt.led_green = 4'b0;
            regs_nxt.led_blue = 4'b0;

            if (buttons_ff.sel_in) state_nxt = RESET_SEL1;
            else state_nxt = RESET;
        end
        RESET_SEL1: begin
            state_nxt = RESET_SEL1_WAIT;
        end

        RESET_SEL1_WAIT: begin
            regs_nxt.operand1 = data_res_in;

            regs_nxt.led_blue = regs_nxt.operand1;

            if (buttons_ff.sel_in) state_nxt = RESET_SEL2;
            else if (buttons_ff.ge_in) state_nxt = RESET_SEL1_WAIT;
            else if (buttons_ff.lt_in) state_nxt = RESET_SEL1_WAIT;
        end

        RESET_SEL2: begin
            state_nxt = RESET_SEL2_WAIT;
        end

        RESET_SEL2_WAIT: begin
            regs_nxt.opcode_out = opcode_res_in;

            if (regs_nxt.opcode_out == 2'b00) regs_nxt.led_green = 4'b0000;
            else  if  (regs_nxt.opcode_out  ==  2'b01)  regs_nxt.led_green  =
4'b0010;
            else  if  (regs_nxt.opcode_out  ==  2'b10)  regs_nxt.led_green  =
4'b0100;
            else  if  (regs_nxt.opcode_out  ==  2'b11)  regs_nxt.led_green  =
4'b1000;

            if (buttons_ff.sel_in) state_nxt = RESET_SEL3;
            else if(buttons_ff.ge_in) state_nxt = RESET_SEL2_WAIT;
            else if(buttons_ff.lt_in) state_nxt = RESET_SEL2_WAIT;
        end

        RESET_SEL3: begin
            regs_nxt.data_shift_out = 1;

            state_nxt = RESET_SEL3_WAIT;
        end

        RESET_SEL3_WAIT: begin
            if (data_valid_in) regs_nxt.operand2 = data_res_in;

            regs_nxt.led_blue = regs_nxt.operand2;

            if (buttons_ff.sel_in) state_nxt = COMPARE;
            else if(buttons_ff.ge_in) state_nxt = RESET_SEL3_WAIT;
            else if(buttons_ff.lt_in) state_nxt = RESET_SEL3_WAIT;
        end

        COMPARE: begin
            regs_nxt.led_red = 4'b0;
```

```
            regs_nxt.led_green = 4'b0;
            regs_nxt.led_blue = 4'b0;

            regs_nxt.enable_out = 1;

            state_nxt = COMPARE_WAIT;
        end

        COMPARE_WAIT: begin
            regs_nxt.led_red = 4'b0;
            regs_nxt.led_green = 4'b0;
            regs_nxt.led_blue = 4'b0;

            if (buttons_ff.sel_in) state_nxt = COMPARE_SEL1;
            else if(buttons_ff.ge_in) state_nxt = COMPARE_GE;
            else if(buttons_ff.lt_in) state_nxt = COMPARE_LT;
        end

        COMPARE_GE: begin
            // res_in maintains
            regs_nxt.led_red = ($signed(res_in) >= 0) ? 4'b0 : 4'b1111;
            regs_nxt.led_green = ($signed(res_in) >= 0) ? 4'b1111 : 4'b0;
            regs_nxt.led_blue = 4'b0;

            if (buttons_ff.sel_in) state_nxt = COMPARE_SEL1;
            else if(buttons_ff.ge_in) state_nxt = COMPARE_GE;
            else if(buttons_ff.lt_in) state_nxt = COMPARE_LT;
        end

        COMPARE_LT: begin
            // res_in maintains
            regs_nxt.led_red = ($signed(res_in) < 0) ? 4'b0 : 4'b1111;
            regs_nxt.led_green = ($signed(res_in) < 0) ? 4'b1111 : 4'b0;
            regs_nxt.led_blue = 4'b0;

            if (buttons_ff.sel_in) state_nxt = COMPARE_SEL1;
            else if(buttons_ff.ge_in) state_nxt = COMPARE_GE;
            else if(buttons_ff.lt_in) state_nxt = COMPARE_LT;
        end

        COMPARE_SEL1: begin
            regs_nxt.operand1 = res_in;
            regs_nxt.opcode_shift_out = 1;

            state_nxt = COMPARE_SEL1_WAIT;
        end

        COMPARE_SEL1_WAIT: begin
            if (opcode_valid_in) regs_nxt.opcode_out = opcode_res_in;

            if (regs_nxt.opcode_out == 2'b00) regs_nxt.led_green = 4'b0000;
            else if (regs_nxt.opcode_out == 2'b01) regs_nxt.led_green =
4'b0010;
            else if (regs_nxt.opcode_out == 2'b10) regs_nxt.led_green =
4'b0100;
            else if (regs_nxt.opcode_out == 2'b11) regs_nxt.led_green =
4'b1000;
```

```
            if (buttons_ff.sel_in) state_nxt = COMPARE_SEL2;
            else if(buttons_ff.ge_in) state_nxt = EXCEPTION_SEL1;
            else if(buttons_ff.lt_in) state_nxt = EXCEPTION_SEL1;
        end

        COMPARE_SEL2: begin
            regs_nxt.data_shift_out = 1;

            state_nxt = COMPARE_SEL2_WAIT;
        end

        COMPARE_SEL2_WAIT: begin
            if (data_valid_in) regs_nxt.operand2 = data_res_in;

            regs_nxt.led_red = 4'b0;
            regs_nxt.led_green = 4'b0;
            regs_nxt.led_blue = regs_nxt.operand2;

            if (buttons_ff.sel_in) state_nxt = COMPARE;
            else if(buttons_ff.ge_in) state_nxt = EXCEPTION_SEL2;
            else if(buttons_ff.lt_in) state_nxt = EXCEPTION_SEL2;
        end

        EXCEPTION_SEL1: begin
            regs_nxt.led_red = 4'b0101;
            regs_nxt.led_green = 4'b1010;
            regs_nxt.led_blue = 4'b0;

            if (buttons_ff.sel_in) state_nxt = COMPARE_SEL1_WAIT;
            else if(buttons_ff.ge_in) state_nxt = EXCEPTION_SEL1;
            else if(buttons_ff.lt_in) state_nxt = EXCEPTION_SEL1;
        end

        EXCEPTION_SEL2: begin
            regs_nxt.led_red = 4'b0101;
            regs_nxt.led_green = 4'b1010;
            regs_nxt.led_blue = 4'b0;

            if (buttons_ff.sel_in) state_nxt = COMPARE_SEL2_WAIT;
            else if(buttons_ff.ge_in) state_nxt = EXCEPTION_SEL2;
            else if(buttons_ff.lt_in) state_nxt = EXCEPTION_SEL2;
        end

    endcase
    end
  end


endmodule
```