CSC258- Lab 6

Finite State Machines

Learning Objectives

The purpose of this lab is to learn how to create FSMs as well as use them to control a datapath over multiple clock cycles.

Marking Scheme

This lab, like other labs, is worth 4% of your final grade, but you will be graded out of 8 marks for this lab, as follows:

• Prelab + Simulations: 3 marks

• Part I (in-lab): 2 marks

• Part II (in-lab): 3 marks

• Part III (bonus): 2 marks (1 for prelab, 1 for in-lab demo)

Preparation Before the Lab

You are required to complete Parts I and II of the lab by writing and testing Verilog code with Modelsim (using reasonable test vectors that you can justify). Part III is optional, but has a prelab component, should you choose to do it. You should hand-in your prelab preparations (schematics, Verilog, and simulation outputs) for Parts I to II to the TAs (and Part III, if you choose to do it).

In-Lab Work

You are required to implement and test all of Parts I and II of the lab (and Part III, if you choose to do it). You should demonstrate them to the teaching assistants when you finished testing them.

Part I

We aim to implement a finite state machine (FSM) that recognizes two specific sequences of applied input symbols: four consecutive 1s or the sequence 1101. There is an input w and an output z. Whenever w = 1 for four consecutive clock pulses, or when the sequence 1101 appears on w across four consecutive clock pulses, the value of z has to be 1; otherwise, z = 0. Overlapping sequences are allowed, so that if w = 1 for five consecutive clock pulses the output z will be equal to 1 after the fourth and fifth pulses. Figure 1 illustrates the required relationship between w and z. A state diagram for this FSM is shown in Figure 2.

Figure 3 shows a starter Verilog code for the required state machine. Study and understand this code as it provides a model for how to clearly describe a finite state machine that will both simulate and synthesize properly. It has missing parts that you should complete yourself.

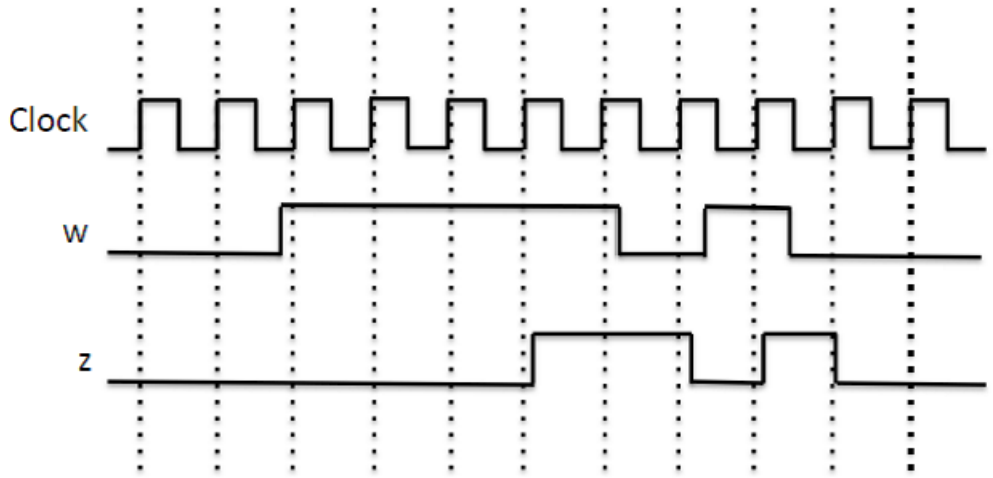


Figure 1: Required timing for the output z.

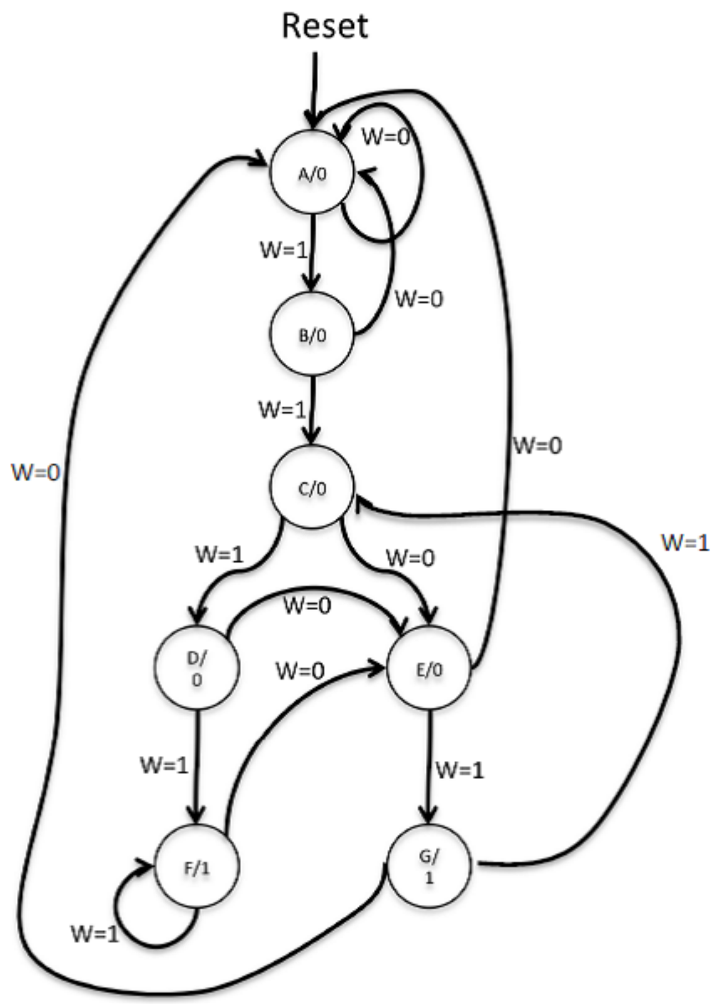


Figure 2: A state diagram for the FSM.

```
// SW[0]:        reset signal
// SW[1]:        input signal (w)

// KEY[0]:       clock

// LEDR[2:0]:    current state
// LEDR[9]:      output (z)

module sequence_detector(SW, KEY, LEDR);
    input [9:0] SW;
    input [3:0] KEY;
    output [9:0] LEDR;

    wire w, clock, resetn, z;

    reg [2:0] y_Q, Y_D; // y_Q represents current state, Y_D represents next state

    localparam A = 3'b000, B = 3'b001, C = 3'b010, D = 3'b011, E = 3'b100, F = 3'b101, G = 3'b110;

    // Connect inputs and outputs to internal wires
    assign w = SW[1];
    assign clock = ~KEY[0];
    assign resetn = SW[0];
    assign LEDR[9] = z;
    assign LEDR[2:0] = y_Q;

    // State table
    // The state table should only contain the logic for state transitions
    // Do not mix in any output logic.  The output logic should be handled separately.
    // This will make it easier to read, modify and debug the code.
    always @(*)
    begin   // Start of state_table
        case (y_Q)
            A: begin
                   if (!w) Y_D = A;
                   else Y_D = B;
               end
            B: begin
                   if(!w) Y_D = A;
                   else Y_D = C;
               end
            C: ...  // To be completed by you!
            D: ...  // To be completed by you!
            E: ...  // To be completed by you!
            F: ...  // To be completed by you!
            G: ...  // To be completed by you!
            default: Y_D = A;
        endcase
    end     // End of state_table

    // State Register (i.e., FFs)
    always @(posedge clock)
    begin   // Start of state_FFs (state register)
        if(resetn == 1'b0)
            y_Q <= A;
        else
            y_Q <= Y_D;
    end     // End of state_FFs (state register)

    // Output logic
    // Set z to 1 to turn on LED when in relevant states
    assign z = ((y_Q == ???) || (y_Q == ???));  // To be completed by you!
endmodule
```

Figure 3: Starter Verilog code for the FSM (sequence_detector.v)

The toggle switch *SW_0* on the DE1-SoC board is the reset signal input for the FSM, *SW_1* is the w input, and the pushbutton *KEY_0* is the clock input that is applied manually. The LED *$LEDR_9$* shows the output *z*, and the state of system is shown on *$LEDR_{2\text{-}0}$*.

For this part, perform the following steps:

1. Begin with the starter code `sequence_detector.v` (Figure 3) that we provided to you (on Blackboard, as well as in the appendices of this handout).

2. Answer the following questions in your prelab: given the starter code, is the `resetn` signal, an synchronous or asynchronous reset? Is it active high, or active low? Given this, what do you have to do in simulation to reset the FSM to the starting state? **(PRELAB)**

3. Complete the state table showing how the present state and input value determine the next state and the output value. Fill in all the missing parts of the template code to implement the FSM based on the state table you derived. Include completed code in your prelab. **(PRELAB)**

4. Simulate your circuit using ModelSim for a variety of input settings, ensuring the output waveforms are correct. Include a few screenshots that shows the simulation output. **(PRELAB)**

5. Create a new Quartus Prime project for your circuit. Make sure to store it in your W:\ drive; select the correct FPGA device (5CSEMA5F31C6); and import the pin assignments. Compile the project. **(IN-LAB)**

6. Download the compiled circuit into the FPGA. Test the functionality of the circuit on your board. When you are sure that it is working correctly, show it to TAs. **(IN-LAB)**

Part II

Most non-trivial digital circuits can be separated into two main functions. One is the datapath where the data flows and the other is the control path that manipulates the signals in the datapath to control the operations performed and how the data flows through the datapath. In previous labs, you learned how to construct a simple ALU, which is a common datapath component. In Part I of this lab you constructed a simple *finite state machine* (FSM), which is the most common component used to implement a control path. Now you will see how to implement an FSM to control a datapath so to perform a useful operation. This is an important step towards building a microprocessor as well as any other computing circuit.

In this part, you are given a datapath and a FSM that controls it to compute $A^2 + C$. Using the given datapath, you should implement a **different** FSM that controls the datapath to perform the following computation:

$$Ax^2 + Bx + C$$

The values of x, A, B and C will be preloaded by the user on the switches before the computation begins.

Figure 4 shows the block diagram of the datapath you will build. There are a few things to note about this diagram:

- Reset signals are not shown to reduce clutter, but do not forget to include them when writing your Verilog code.
- The datapath will operate on 8-bit unsigned values. Assume that the input values are small enough to not cause any overflows at any point in the computation, i.e., no results will exceed $2^8 - 1 = 255$.
- The ALU needs to perform only addition and multiplication, but you could use a variation of the ALU you built previously to have more operations available for solving other equations if you wish to try some things on your own.
- There are four registers R_x, R_A, R_B and R_C used at the start to store the values of x, A, B and C, respectively. The registers R_A and R_B can be overwritten during the computation.
- There is one output register, R_R, that captures the output of the ALU and displays the value in binary on the LEDs and in hexadecimal on the HEX displays.
- Two 8-bit-wide, 4-to-1 multiplexers at the inputs to the ALU are used to select which register values are input to the ALU.
- All registers have enable signals to determine when they are to load new values and an active low synchronous reset.

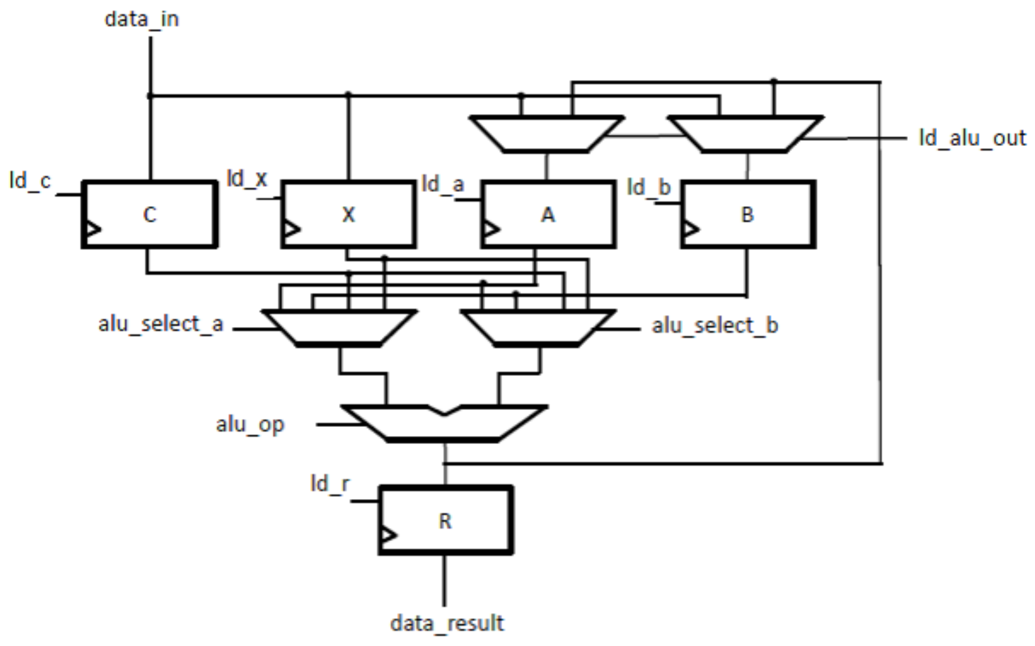


Figure 4: Block diagram of datapath.

The provided circuit should operate in the following manner:

- After an active low synchronous reset on KEY_0, you will input the value for R_A on switches $SW[7:0]$.
- When KEY_1 is pushed and released, R_A will be loaded. Then you will input the next value on the switches that will be loaded into R_B, and press and release KEY_1. You repeat this to load R_C and R_X.
- Computation will start after KEY_1 is pressed and released for loading R_X.
- When computation is finished, the final result will be loaded into R_R.
- This final result should be displayed on $LEDR_{7-0}$ in binary and *HEX0* and *HEX1* in hex.

Note that KEY_1 is NOT your clock! You will use the 50 MHz clock available through input CLOCK 50 as your clock.

For this part, you should perform the following steps:

1. Examine the provided starter Verilog code for this part (`poly_function.v`, which is available on Blackboard, as well as in the appendices of this handout). This is a major step in this part of the lab. You will not need to write much Verilog yourself, but you will need to fully understand the circuitry described by the provided Verilog to be able to make your modifications. **(PRELAB)**
2. Determine a sequence of steps similar to the datapath example shown in lecture that controls your datapath to perform the required computation. You should draw a table that shows the state of the Registers and control signals for each cycle of your computation. Include this table in your prelab. **(PRELAB)**
3. Draw a state diagram for your controller starting with the register load states provided in the example FSM. Include the state diagram in your prelab. **(PRELAB)**
4. Modify the provided FSM code to implement your controller and synthesize it. You should only modify the control module. Include your modified code in the prelab. **(PRELAB)**
5. To examine the circuit produced by Quartus Prime open the RTL Viewer tool (Tools > Netlist Viewers > RTL Viewer). Find (on the left panel) and double-click on the box shown in the circuit that represents the finite state machine, and determine whether the state diagram that it shows properly corresponds to the one you have drawn. To see the state codes used for your FSM, open the Compilation Report, select the Analysis and Synthesis section of the report, and click on State Machines. Include a screenshot of the generated FSM in your prelab. **(PRELAB)**

   The state codes after synthesis may be different from what you originally specified. This is because the tool may have found a way to optimize the logic better by choosing a different state assignment. If you really need to use your original state assignment, there is a setting to keep it (which we do not investigate here).
6. Simulate your circuit with ModelSim for a variety of input settings, ensuring the output waveforms are correct. It is recommended that you start by simulating the datapath and

controller modules separately. Only when you are satisfied that they are working individually should you combine them into the full design. Why is this approach better? (Hint: Consider the case when your design has 20 different modules.) Include few screenshots of simulation output in your prelab. **(PRELAB)**

7. Compile your project in Quartus, program the FPGA in DE1 board, and test it with various inputs. After you are satisfied that your circuit is working properly, show it to TAs. **(IN-LAB)**

Part III (Optional)

Note: Only start working on this part if you already completed other parts. This is an optional part that provides a more challenging exercise for you to further test your knowledge.

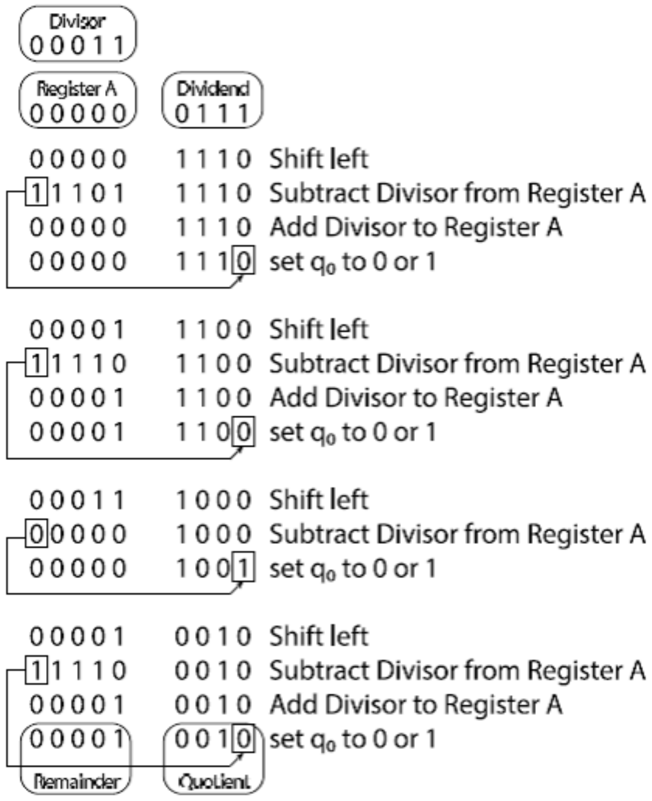
Addition, subtraction and multiplication are much easier to build in hardware than division. So division is the most complex operation in hardware. For this part, you will design a 4-bit restoring divider using a finite state machine.

Figure 5 shows an example of how the restoring divider works. This mimics what you do when you do long division by hand. In this specific example, number 7 (*Dividend*) is divided by number 3 (*Divisor*). The restoring divider starts with *Register A* set to *0*. The *Dividend* is shifted left and the bit shifted out of the left most bit of the *Dividend* (called the most significant bit or MSB) is shifted into the least significant bit (LSB) of *Register A* as shown in Figure 6.

The *Divisor* is then subtracted from *Register A*. This is equivalent to adding the 2's complement of the *Divisor* (11101 for the example in Figure 5) to *Register A*. If the MSB of *Register A* is a *1*, then we restore *Register A* back to its original value by adding the *Divisor* back to *Register A*, and set the LSB of the *Dividend* to *0*. Else, we do not perform the restoring addition and immediately set the LSB of the *Dividend* to *1*.

This cycle is performed until all the bits of the *Dividend* have been shifted out. Once the process is complete, the new value of the *Dividend* register is the *Quotient*, and *Register A* will hold the value of the *Remainder*.

To implement this part, you will use SW_{3-0} for the divisor value and SW_{7-4} for the dividend value. Use *CLOCK 50* as the clock signal, KEY_0 as a synchronous active high reset, and KEY_1 as the *Go* signal to start computation. The output of the *Divisor* will be displayed on *HEX0*, the *Dividend* will be displayed on *HEX2*, the *Quotient* on *HEX4*, and the *Remainder* on *HEX5*. Set the remaining HEX displays to 0. Also display the *Quotient* on *LEDR*.



```
Divisor
00011

Register A    Dividend
00000         0111

00000   1110  Shift left
11101   1110  Subtract Divisor from Register A
00000   1110  Add Divisor to Register A
00000   1110  set q0 to 0 or 1

00001   1100  Shift left
11110   1100  Subtract Divisor from Register A
00001   1100  Add Divisor to Register A
00001   1100  set q0 to 0 or 1

00011   1000  Shift left
00000   1000  Subtract Divisor from Register A
00000   1001  set q0 to 0 or 1

00001   0010  Shift left
11110   0010  Subtract Divisor from Register A
00001   0010  Add Divisor to Register A
00001   0010  set q0 to 0 or 1
Remainder  Quotient
```

Figure 5: An example of how a restoring divider works.

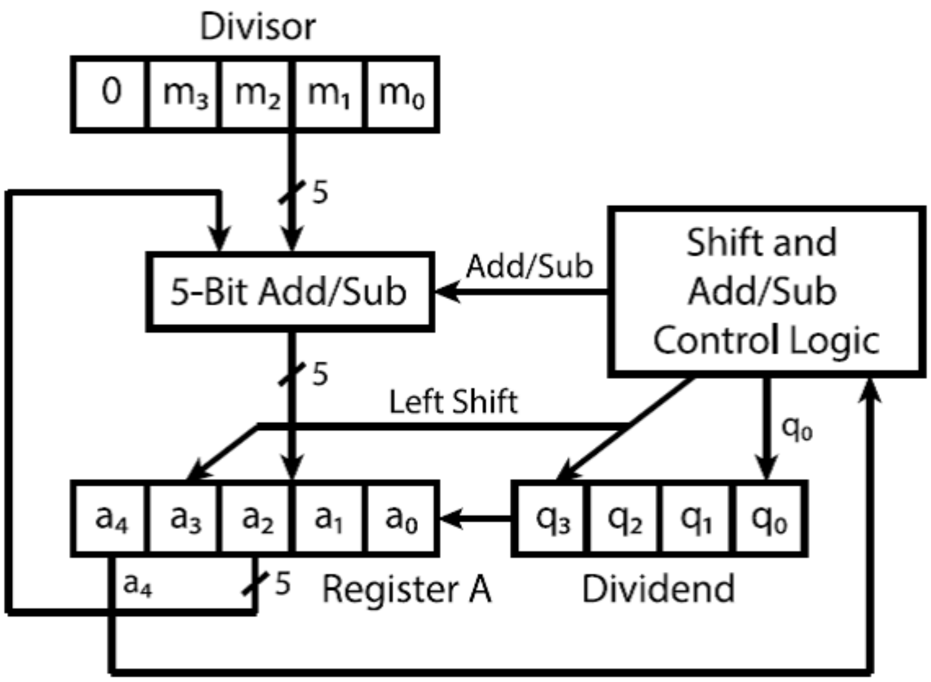


Figure 6: Block diagram of restoring divider.

Structure your code in the same way as you were shown in Part II and follow these steps for this part:

1. Draw a schematic for the datapath of your circuit. It will be similar to Figure 6. You should show how you will initialize the registers, where the outputs are connected to, and include all the control signals that you require.
2. Draw the state diagram that controls your datapath.
3. Draw the schematic for your controller module.
4. Draw the top-level schematic showing how the datapath and controller are connected as well as the inputs and outputs to your top-level circuit.
5. Write the Verilog code that implements your circuit.
6. Simulate your circuit in ModelSim for a variety of input settings.
7. After you are satisfied with your simulations, download and test it on the FPGA board.

Appendices: Source Codes

sequence_detector.v

```
// SW[0]:          reset signal
// SW[1]:          input signal (w)
// KEY[0]:         clock
// LEDR[2:0]:      current state
// LEDR[9]:        output (z)

module sequence_detector(SW, KEY, LEDR);
   input [9:0] SW;
   input [3:0] KEY;
   output [9:0] LEDR;

   wire w, clock, resetn, z;
   reg [2:0] y_Q, Y_D; // y_Q represents current state, Y_D represents next state
   localparam A = 3’b000, B = 3’b001, C = 3’b010, D = 3’b011, E = 3’b100,
            F = 3’b101, G = 3'b110;
   assign w = SW[1];
   assign clock = ~KEY[0];
   assign resetn = SW[0];
   assign LEDR[9] = z;
   assign LEDR[2:0] = y_Q;
   // State table
   // The state table should only contain the logic for state transitions
   // Do not mix in any output logic. The output logic should be handled
   // separately. This will make it easier to read, modify and debug the code.
   always @(*)
   begin    // Start of state_table
      case (y_Q)
         A: begin
               if (!w) Y_D = A;
               else Y_D = B;
            end
         B: begin
               if(!w) Y_D = A;
               else Y_D = C;
            end
         C: ???   // To be completed by you!
         D: ???   // To be completed by you!
         E: ???   // To be completed by you!
         F: ???   // To be completed by you!
         G: ???   // To be completed by you!
         default: Y_D = A;
      endcase
   end   // End of state_table
   // State Register (i.e., FFs)
   always @(posedge clock)
      begin:   // Start of state_FFs (state register)
         if(resetn == 1'b0)
            y_Q <=  A;
         else
            y_Q <= Y_D;
      end   // End of state_FFs (state register)

      // Output logic
      // Set z to 1 to turn on LED when in relevant states
      assign z = ((y_Q == ???) || (y_Q == ???));    // To be completed by you!
endmodule
```

poly_function.v

```
// SW[7:0]  data in
// KEY[0]   synchronous reset when pressed
// KEY[1]   go signal
// LEDR     displays result
// HEX0 & HEX1 also displays result

module fpga_top(SW, KEY, CLOCK50, LEDR, HEX0, HEX1);
   input [9:0] SW;
   input [3:0] KEY;
   input CLOCK 50;
   output [9:0] LEDR;
   output [6:0] HEX0, HEX1;

   wire resetn;
   wire go;

   wire [7:0] data result;
   assign go = ~KEY[1];
   assign resetn = KEY[0];

   part2 u0(
      .clk(CLOCK 50),
      .resetn(resetn),
      .go(go);
      .data in(SW[7:0]),
      .data result(data result)
      );

   assign LEDR[9:0] = {2’b00, data result};
   hex_decoder H0(
      .hex_digit(data_result[3:0]),
      .segments(HEX0)
      );
   hex_decoder H1(
      .hex_digit(data_result[7:4]),
      .segments(HEX1)
      );

endmodule

module part2(
   input clk,
   input resetn,
   input go,
   input [7:0] data_in,
   output [7:0] data_result
   );

   // lots of wires to connect our datapath and control
   wire ld_a, ld_b, ld_c, ld_x, ld_r;
   wire ld_alu_out;
   wire [1:0] alu_select_a, alu_select_b;
   wire alu_op;

   control C0(
      .clk(clk) ,
      .resetn(resetn),

      .go(go),

      .ld_alu_out(ld_alu_out),
      .ld_x(ld_x),
      .ld_a(ld_a),
      .ld_b(ld_b),
      .ld_c(ld_c),
      .ld_r(ld_r),

      .alu_select_a(alu_select_a),
      .alu_select_b(alu_select_b),
      .alu_op(alu_op)
      );
   datapath D0(
      .clk(clk) ,
      .resetn(resetn),

      .go(go),

      .ld_alu_out(ld_alu_out),
      .ld_x(ld_x),
      .ld_a(ld_a),
      .ld_b(ld_b),
      .ld_c(ld_c),
      .ld_r(ld_r),

      .alu_select_a(alu_select_a),
      .alu_select_b(alu_select_b),
      .alu_op(alu_op)

      .data_in(data_in),
      .data_result(data_result)
      );

endmodule

module control(
   input clk,
   input resetn,
   input go,

   output reg ld_a, ld_b, ld_c, ld_x, ld_r,
   output reg ld_alu_out,
   output reg [1:0] alu_select_a , alu_select_b,
   output reg alu_op
   );

   reg [3:0] current_state, next_state;

   localparam S_LOAD_A          = 4’d0,
              S_LOAD_A_WAIT     = 4’d1,
              S_LOAD_B          = 4’d2,
              S_LOAD_B_WAIT     = 4’d3,
              S_LOAD_C          = 4’d4,
              S_LOAD_C_WAIT     = 4’d5,
              S_LOAD_X          = 4’d6,
              S_LOAD_X_WAIT     = 4’d7,
              S_CYCLE_0          = 4’d8,
              S_CYCLE_1          = 4’d9,
              S_CYCLE_2          = 4’d10;

   // Next state logic aka our state table
   always @(*)
   begin: state table
      case (current state)
         // Loop in current state until value is input
         S_LOAD_A: next_state = go ? S_LOAD_A_WAIT : S_LOAD_A;
         // Loop in current state until go signal goes low
         S_LOAD_A_WAIT: next_state = go ? S_LOAD_A_WAIT : S_LOAD_B;
         // Loop in current state until value is input
         S_LOAD_B: next_state = go ? S_LOAD_B_WAIT : S_LOAD_B;
         // Loop in current state until go signal goes low
         S_LOAD_B_WAIT: next_state = go ? S_LOAD_B_WAIT : S_LOAD_C;
         // Loop in current state until value is input
         S_LOAD_C: next_state = go ? S_LOAD_C_WAIT : S_LOAD_C;
         // Loop in current state until go signal goes low
         S_LOAD_C_WAIT: next_state = go ? S_LOAD_C_WAIT : S_LOAD_X;
         // Loop in current state until value is input
         S_LOAD_X: next_state = go ? S_LOAD_X_WAIT : S_LOAD_X;
         // Loop in current state until go signal goes low
         S_LOAD_X_WAIT: next_state = go ? S_LOAD_X_WAIT : S_CYCLE_0;
         S_CYCLE_0: next_state = S_CYCLE_1;
         // we will be done our two operations, start over after
         S_CYCLE_1: next_state = S_LOAD_A;
      default: next_state = S_LOAD_A;
      endcase
   end // state_table

   // Output logic aka all of our datapath control signals
   always @(*)
      begin: enable_signals
         // By default make all our signals 0
         ld_alu_out = 1'b0;
         ld_a = 1'b0;
         ld_b = 1'b0;
         ld_c = 1'b0;
         ld_x = 1'b0;
         ld_r = 1'b0;
         alu_select_a = 2'b00;
         alu_select_b = 2'b00;
         alu_op       = 1'b0;

         case (current_state)
            S_LOAD_A: begin
              ld_a = 1'b1;
              end
            S_LOAD_B: begin
              ld_b = 1'b1;
              end
            S_LOAD_C: begin
                ld_c = 1'b1;
                end
            S_LOAD_X: begin
                ld_x = 1'b1;
                end
            S_CYCLE_0: begin                          // Do A <- A * A
                ld_alu_out = 1'b1; ld_a = 1'b1; // store result back into A
                alu_select_a = 2'b00;           // Select register A
                alu_select_b = 2'b00;           // Also select register A
                alu_op = 1'b1;                  // Do multiply operation
                end
            S_CYCLE_1: begin
                ld_r = 1'b1;          // store result in result register
                alu_select_a = 2'b00; // Select register A
                alu_select_b = 2'b10; // Select register C
                alu_op = 1'b0;        // Do Add operation
                end
        // default:
        // don't need default since we already made sure all of our outputs were
        // assigned a value at the start of the always block
        endcase
    end   // enable_signals

    // current_state registers
    always @(posedge clk)
    begin: state_FFs
        if(!resetn)
            current_state <= S_LOAD_A;
        else
            current_state <= next_state;
    end   // state_FFS
endmodule

module datapath(
    input clk,
    input resetn,
    input [7:0] data_in,
    input ld_alu_out,
    input ld_x, ld_a, ld_b, ld_c,
    input ld_r,
    input alu_op,
    input [1:0] alu_select_a, alu_select_b,
    output reg [7:0] data_result
    );

    // input registers
    reg [7:0] a, b, c, x;

    // output of the alu
    reg [7:0] alu_out;
    // alu input muxes
    reg [7:0] alu_a, alu_b;
```

```
// Registers a, b, c, x with respective input logic
always @(posedge clk) begin
   if (!resetn) begin
      a <= 8'd0;
      b <= 8'd0;
      c <= 8'd0;
      x <= 8'd0;
   end
   else begin
      if (ld_a)
         // load alu_out if load_alu_out signal is high,
         // otherwise load from data_in
         a <= ld_alu_out ? alu_out : data_in;
      if (ld_b)
         // load alu_out if load_alu_out signal is high,
         // otherwise load from data_in
         b <= ld_alu_out ? alu_out : data_in;
      if (ld_x)
         x <= data_in;

      if (ld_c)
         c <= data_in;
   end
end

// Output result register
always @(posedge clk) begin
   if (!resetn) begin
      data_result <= 8'd0;
   end
   else
      if(ld_r)
         data_result <= alu_out;
end

// The ALU input multiplexers
always @(*)
begin
   case (alu_select_a)
      2'd0: alu_a = a;
      2'd1: alu_a = b;
      2'd2: alu_a = c;
      2'd3: alu_a = x;
      default: alu_a = 8'd0;
   endcase

   case (alu_select_b)
      2'd0: alu_b = a;
      2'd1: alu_b = b;
      2'd2: alu_b = c;
      2'd3: alu_b = x;
      default: alu_b = 8'd0;
   endcase
end
```

```
   // The ALU
   always @(*)
   begin: ALU
      // alu
      case (alu_op)
         0: begin
            alu_out = alu_a + alu_b; //performs addition
            end
         1: begin
            alu_out = alu_a * alu_b; //performs multiplication
            end
         default: alu_out = 8'd0;
      endcase
   end

endmodule

module hex_decoder(hex_digit, segments);
   input [3:0] hex_digit;
   output reg [6:0] segments;

   always @(*)
      case (hex_digit)
         4'h0: segments = 7'b100_0000;
         4'h1: segments = 7'b111_1001;
         4'h2: segments = 7'b010_0100;
         4'h3: segments = 7'b011_0000;
         4'h4: segments = 7'b001_1001;
         4'h5: segments = 7'b001_0010;
         4'h6: segments = 7'b000_0010;
         4'h7: segments = 7'b111_1000;
         4'h8: segments = 7'b000_0000;
         4'h9: segments = 7'b001_1000;
         4'hA: segments = 7'b000_1000;
         4'hB: segments = 7'b000_0011;
         4'hC: segments = 7'b100_0110;
         4'hD: segments = 7'b010_0001;
         4'hE: segments = 7'b000_0110;
         4'hF: segments = 7'b000_1110;
         default: segments = 7'h7f;
      endcase
endmodule
```