Laboratory Exercise 6

Finite State Machines

The purpose of this exercise is to learn how to create and use finite state machines.

Preparation Before the Lab

You are required to complete Parts I to III of the lab by writing and testing Verilog code and compiling it with Quartus II. Show your schematic, Verilog, and simulations for Parts I to III and state diagrams for Parts II and III to the teaching assistants. You must simulate your circuit with ModelSim using reasonable test vectors.

In-lab Work

You are required to implement and test all of Parts I to III of the lab. You need to demonstrate all parts to the teaching assistants.

Part I

We wish to implement a finite state machine (FSM) that recognizes two specific sequences of applied input symbols, namely 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.

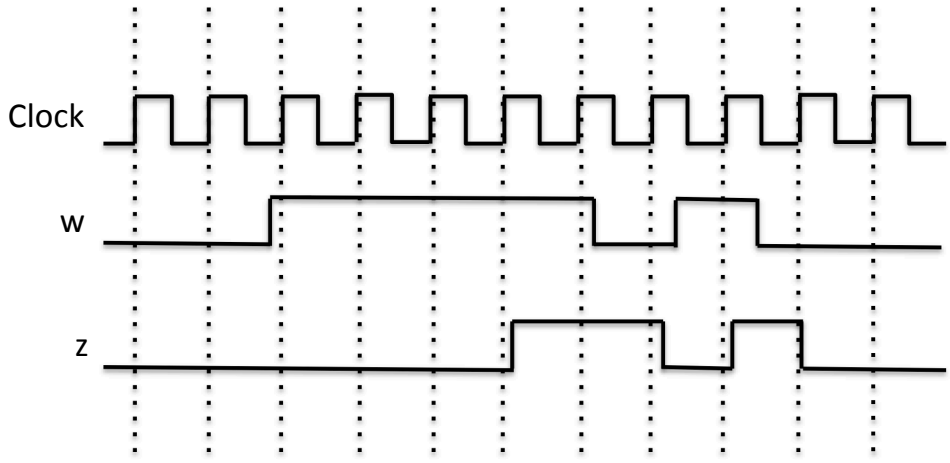

Figure 1: Required timing for the output z.

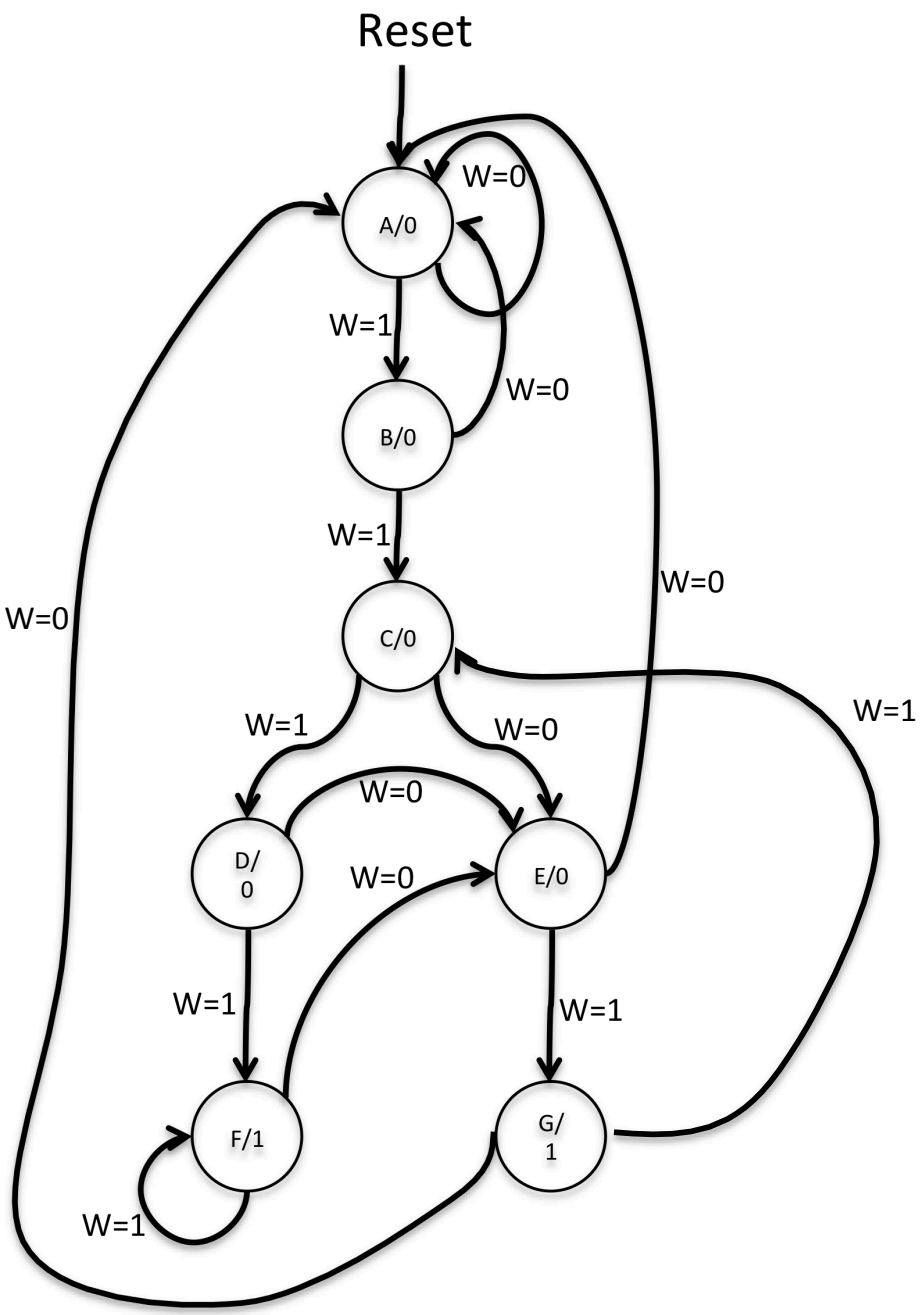

Figure 2: A state diagram for the FSM.

Figure 3 shows a partial Verilog file 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.

The toggle switch SW_0 on the DE1-SoC board is an active-low synchronous reset input for the FSM, SW_1 is the w input, and the pushbutton KEY_0 is the clock input that is applied manually. The red LED $LEDR_9$ is the output z, and the state flip-flop outputs are assigned to $LEDR_{3-0}$.

```
//SW[0] reset when 0
//SW[1] input signal

//KEY[0] clock signal

//LEDR[3:0] displays current state
//LEDR[9] displays output

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

    wire w, clock, reset_b;

    reg [3:0] y_Q, Y_D; // y_Q represents current state, Y_D represents next state
    wire out_light;

    parameter A = 4'b0000, B = 4'b0001, C = 4'b0010, D = 4'b0011, E = 4'b0100, F = 4'b0101, G = 4'b0110;

    assign w = SW[1];
    assign clock = ~KEY[0];
    assign reset_b = SW[0];

   // 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: 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: ???
             D: ???
             E: ???
             F: ???
             G: ???
             default: Y_D = A;
         endcase
    end // state_table


   // State Registers

    always @(posedge clock)
    begin: state_FFs
        if(reset_b == 1'b0)
            y_Q <= 4'b0000;
        else
            y_Q <= Y_D;
    end // state_FFS


   // Output logic
   // Set out_light to 1 to turn on LED when in relevant states

   assign out_light = ((y_Q == ???) | (y_Q == ???));

   // Connect to I/O

    assign LEDR[9] = out_light;
    assign LEDR[3:0] = y_Q;
endmodule
```

Figure 3: Verilog code for the FSM.

Perform the following steps:

1. Copy the code into a file and name it `sequence_detector.v`.

2. Complete the state table and the output logic.

3. Draw a schematic describing the circuit and explain it to the TA as part of your prelab.

4. Simulate your circuit with ModelSim for a variety of input settings, ensuring the output waveforms are correct. You must show this to the TA as part of your prelab.

5. Compile the project.

6. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit.

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 have already 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 that a useful operation is performed. This is an important step towards building a microprocessor as well as any other computing circuit.

In this part, you will be given a block diagram of a datapath. You are to implement the datapath and an FSM that performs the quadratic function:

$$Ax^2 + Bx + c$$

using the given datapath. The values of x, A, B and C will be preloaded before the computation begins.

Figure 4 shows the block diagram of the datapath you will build. Resets are not shown. The datapath will carry 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 only to perform 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_{OUT}, that captures the output of the ALU and displays the value in binary on the LEDs and in hex on the Hex displays. Two 4 to 1 multiplexers at the inputs to the ALU, *MuxA* and *MuxB*, 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 high synchronous reset.

The circuit operates in the following manner. After an active high synchronous *Reset* on KEY_0, you will preload registers R_x and R_A when KEY_1 is pushed and then registers R_B and R_C are loaded when KEY_2 is pushed. After KEY_2 is released, the computation is performed and the circuit returns to wait for new values to be loaded. The final result should be loaded into R_{OUT} for display.

Use SW_{3-0} for the initial values of x and B and SW_{7-4} for the initial values of A and C. Note that the registers are 8-bits and you only have four switches to specify the value. You should load the upper nybble (upper four bits) with 0's, i.e., you can only initialize with values in the range 0 to 15.

The final result is displayed on $LEDR_{7-0}$ in binary and *HEX0* and *HEX1* in hex.

You will use *CLOCK_50* as your clock.

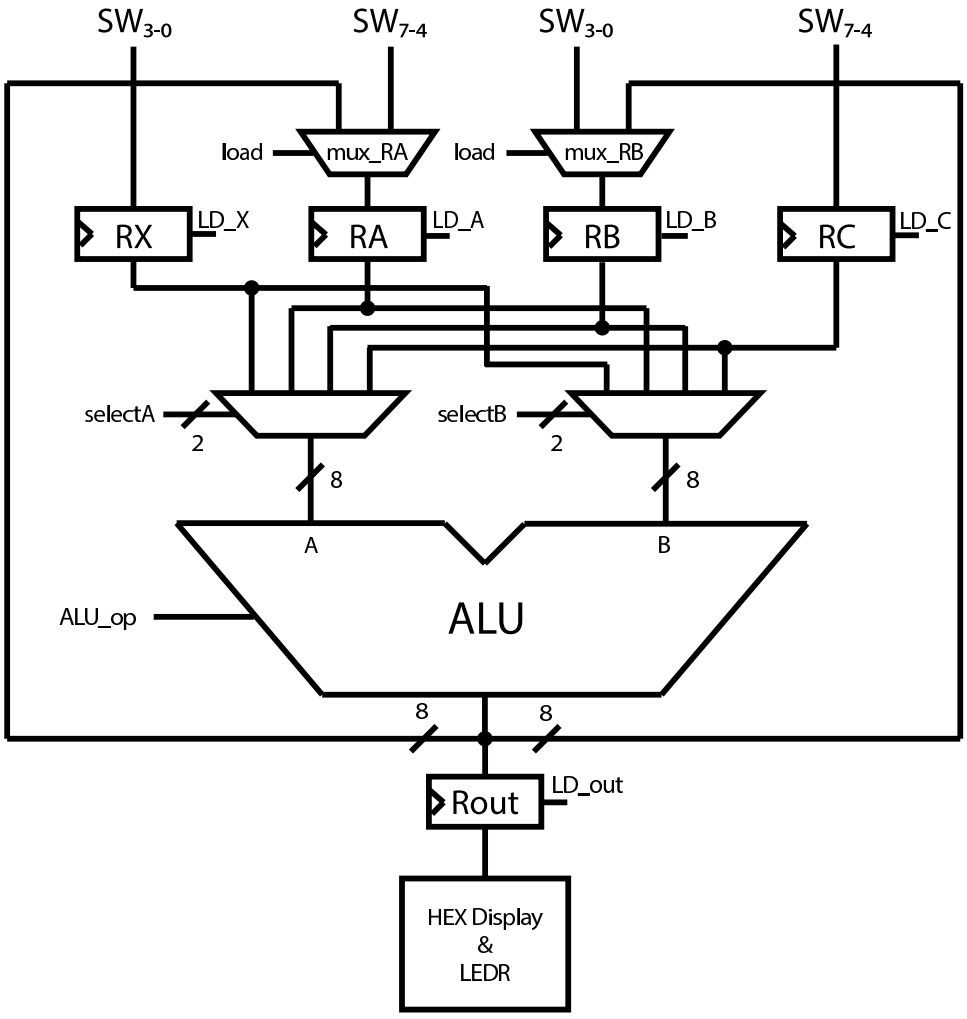

Figure 4: Block diagram of datapath.

Structuring Your Code

At the top level you will have two separate modules for the datapath and the control path. The main connections between the datapath and control path modules will be the control signals coming from the control path and entering the datapath. You will also instantiate your hex decoders at this level.

From Figure 4 you can see that there are five registers that are similar in structure. Write a generic register module that you can instantiate with the appropriate inputs and outputs for the five registers. Also write modules for the 2 to 1 and 4 to 1 multiplexers since you need more than one of them as well.

Build your datapath module by instantiating and connecting the registers and multiplexers according to Figure 4. You can write your ALU as an always block, or also instantiate a module, such as the one you built in a previous lab. You may use the add and multiply operator symbols, i.e., you do not have to build your own operator logic.

The control path module should use a structure similar to what you were provided in Figure 3 for Part I. The main difference will be for the *Output logic* because there are many more output signals. You can use a similar style as in Figure 3 by creating a number of `assign` statements. A better approach in this design is to use a second case statement that has the same states as the case statement for the *State table*, but assign the outputs instead of the next states. The example code fragment shown in Figure 5 computes Ax^2 in two cycles of the state machine. In this style, it is easy to see what the control signals are set to in each state.

You could be lazy and combine the state table and the output logic case statements. However, this makes the code more difficult to read, and worse, it makes it much more difficult for the synthesis tool to figure out what you want to do. This may lead to unexpected interpretation of your code by the tool and bugs that can be hard to find. The result is that being lazy often results in a lot more work and frustration as you try to debug your code, thus defeating the goal of being lazy in the first place! The choice is yours...

```
//  Some of the states:
//  RESET_S : reset state
//  AX_S : A * X
//  AXX_S : (A * X) * X
//  BX_S : B * X
//
//  Some of the control signals as labeled on Figure 4
//  LD_A : load RA
//  LD_B : load RB
//  load : select for Mux_RA and Mux_RB
//  selectA : MuxA select
//  selectB : MuxB select
//  ALU_OP : select ALU operation


//  Part of the State table

    always @(*)
    begin: state_table
        case (PresentState)
          ...
          AX_S: // Compute A * X
            NextState = AXX_S;
          AXX_S: // Compute (A * X) * X
            NextState = BX_S;
          ...
          default:     NextState = RESET_S;
        endcase
    end // state_table


   // Part of the Output logic

    always @(*)
    begin: output_logic
        case (PresentState)
          // In each state assign a value to all control signals

          ...

          AX_S: // Compute A * X and store in A
            begin
               LD_A = 1;  // Store result in RA
               LD_B = 0;
               load = 0;  // Select ALU output
               selectA = RA;  // ALU A input gets RA
               selectB = RX;  // ALU B input gets RX
               ALU_OP = MULT;
               // Set values for rest of control signals
            end
          AXX_S: // Compute (A * X) * X and store in A
            begin
               LD_A = 1;  // Store result in RA
               LD_B = 0;
               load = 0;  // Select ALU output
               selectA = RA;  // RA now has A * X
               selectB = RX;
               ALU_OP = MULT;
               // Set values for rest of control signals
            end

          ...

          default:
            begin
               LD_A = 0;
               LD_B = 0;
               load = 0;
               selectA = RX;
               selectB = RX;
               ALU_OP = ADD;
               // Set values for rest of control signals
            end
        endcase
    end // output_logic
```

Figure 5: Verilog code fragments for the controller

Figure 6 shows a code fragment for the controller output logic using `assign` statements.

```
// Alternate code for output logic
// It's ugly, so won't do too much here.

assign LD_A = ((PresentState == AX_S) | (PresentState == AXX_S) | ...);
assign selectA = (PresentState == AX_S) ? RA :
                 ((PresentState == AXX_S) ? RA :
                  ...
                  RX);
```

Figure 6: Verilog code fragments for the controller using `assign` statements

Perform the following steps.

1. Draw a state diagram for your controller.
2. Draw a schematic outlining the hierarchies you will use and explain them to the TA as part of your prelab.
3. Write a Verilog file that realizes the required circuit.
4. To examine the circuit produced by Quartus II 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.

   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.
5. Simulate your circuit with ModelSim for a variety of input settings, ensuring the output waveforms are correct. You must show this to the TA as part of your prelab. 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.)
6. Compile the project.
7. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit.

Part III

Division in hardware is the most complex of the four basic operations. Add, subtract and multiply are much easier to build in hardware.For this part, you will be designing a 4-bit restoring divider using a finite state machine.

Figure 7 shows an example of how the restoring divider works. 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 8.

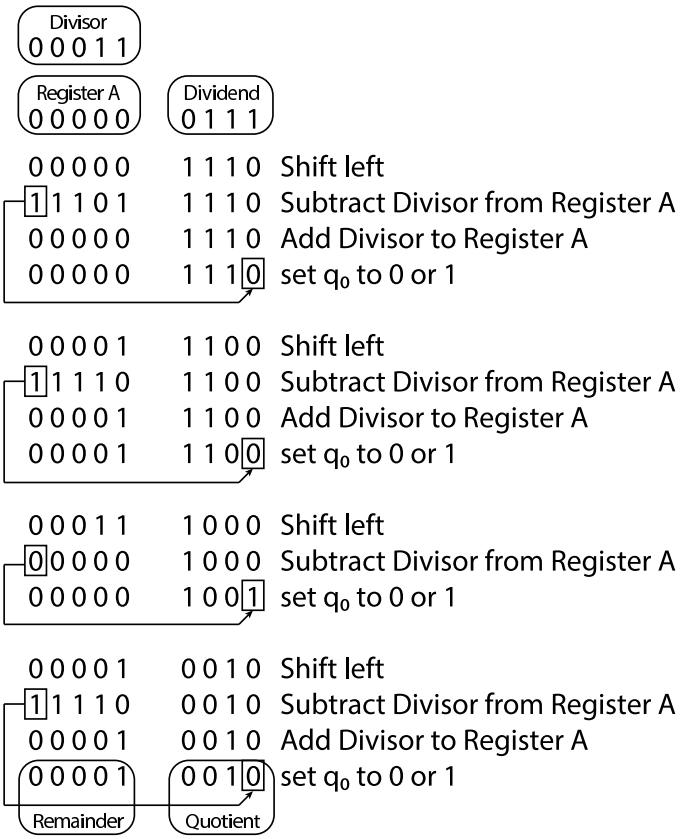

Figure 7: An example of functionality of restoring divider.

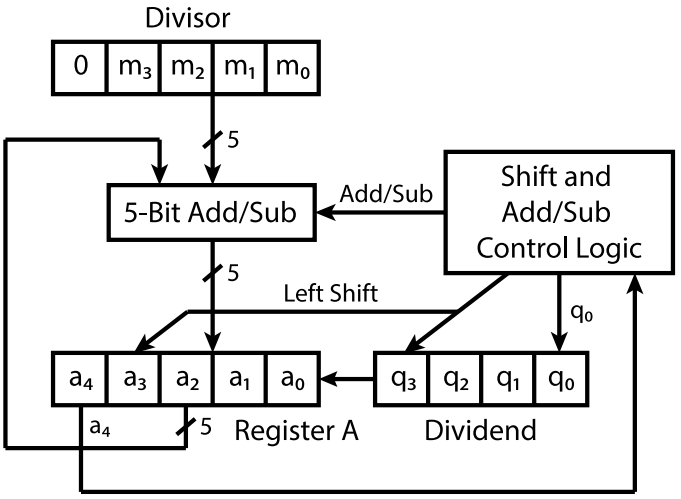

Figure 8: Block diagram of restoring divider.

The *Divisor* is then subtracted from *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 in 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* to for 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*.

Structure your code in the same way as you were shown in Part II.

Perform the following steps.

1. Draw a schematic for the datapath of your circuit. It will be similar to Figure 8. You should show how you will initialize the registers, where the outputs are taken, and include all the control signals that you require.
2. Draw the state diagram to control your datapath.
3. Draw a schematic that describes the hierarchies you will use.
4. Write a Verilog file that realizes your circuit.
5. Simulate your circuit with ModelSim for a variety of input settings, ensuring the output waveforms are correct. You must show this to the TA as part of your prelab.
6. Compile the project.
7. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit.