CSC 258 - Lab 6

Finite State Machines

Fall 2016

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

Each lab 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 (optional)

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 you can justify). Show your relevant preparation (schematics, Verilog, and simulations) for Parts I to II to the teaching assistants.

In-lab Work

You are required to implement and test all of Parts I and II of the lab. You need to demonstrate them 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.

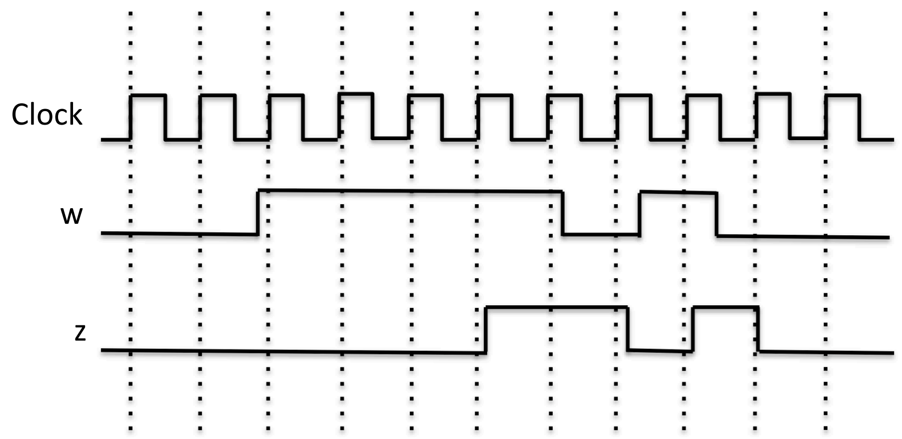


Figure 1: Required timing for the output z.

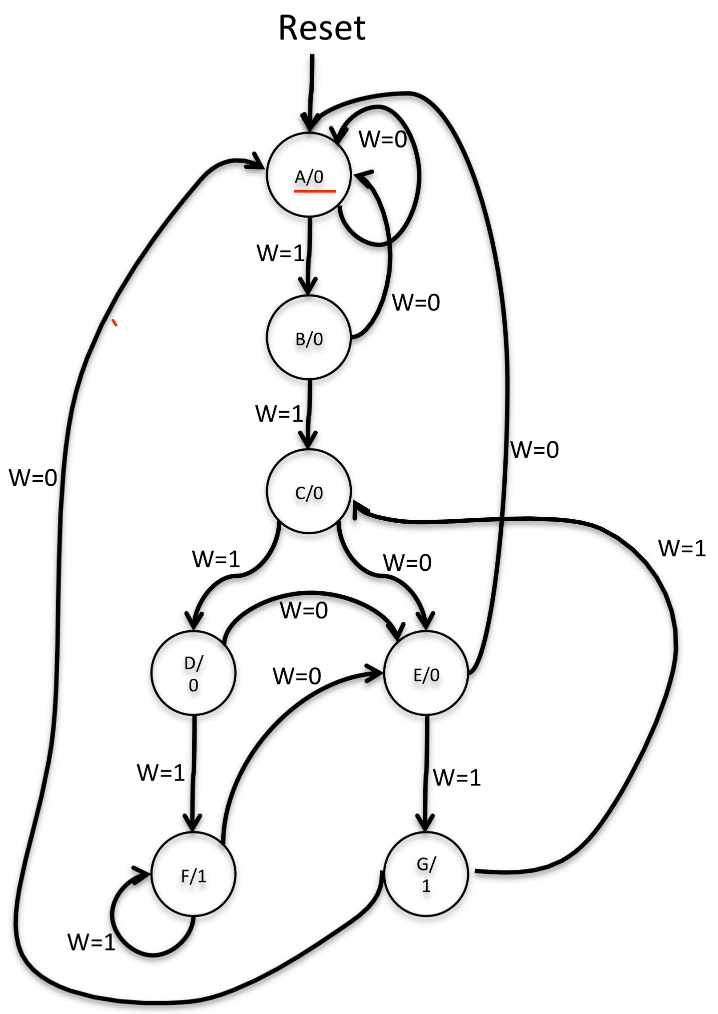


Figure 2: A state diagram for the FSM.

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 a 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 LED $LEDR_9$ is the output z, and the state flip-flop outputs are assigned to $LEDR_{2-0}$.

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

// KEY[0] clock signal

// LEDR[2:0] 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, resetn, out_light;

    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];

    // 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 Register (i.e., FFs)

    always @(posedge clock)
    begin: state_FFs
        if(resetn == 1'b0)
            y_Q <= A;
        else
            y_Q <= Y_D;
    end // State Register


    // 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[2:0] = y_Q;
endmodule
```

Figure 3: Verilog code for the FSM.

Perform the following steps:

1. Begin with the template code `sequence_detector.v` we provided to you. **(PRELAB)**
sync active low reset, have to let reset be low for one clock cycle
2. Answer the following questions: given the template code, is the `resetn` signal used as 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. **(PRELAB)**
4. Simulate your circuit using ModelSim for a variety of input settings, ensuring the output waveforms are correct. **(PRELAB)**
5. Create a new Quartus Prime project for your circuit. Make sure it is stored in your `W:\` drive. Make sure to select the correct FPGA device (5CSEMA5F31C6) and import the pin assignments. Compile the project.
6. Download the compiled circuit into the FPGA. Test the functionality of the circuit on your board.

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 are given a datapath and also an FSM that controls this datapath to perform computation $A^2 + C$. Using the given datapath, **you are required to implement a different FSM that controls the datapath to perform a different 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. 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.

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 and then you will input the next value on the switches that will be loaded into R_B. Likewise for 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.

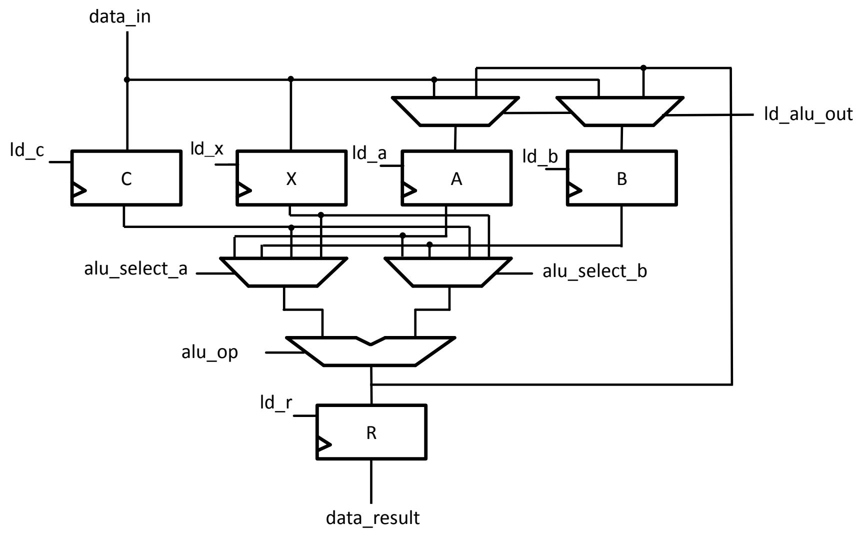


Figure 4: Block diagram of datapath.

Perform the following steps:

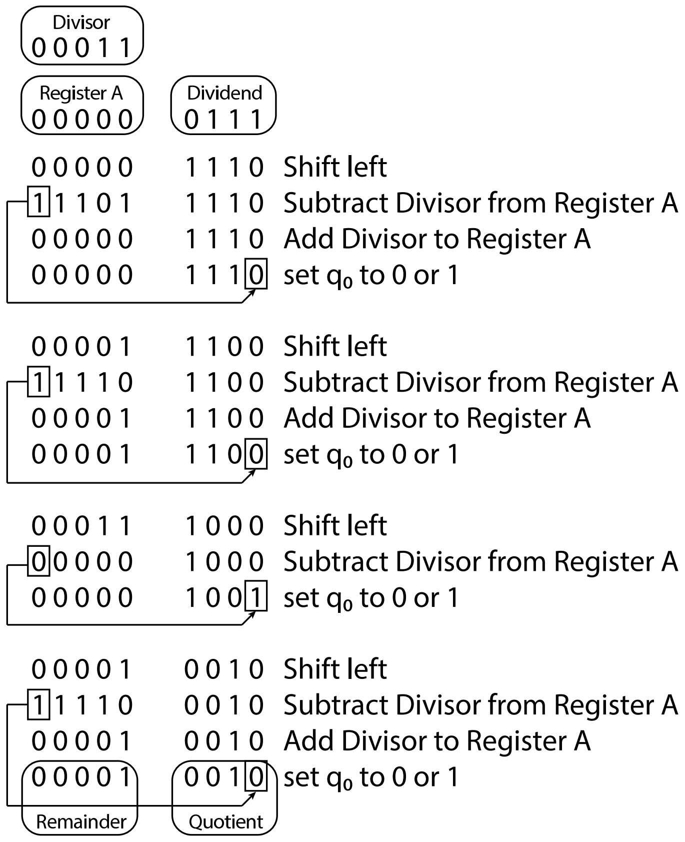
1. Examine the provided Verilog code. This is a major step in this part of the lab. You will not need to write much Verilog yourself, but you will need to fully understand the circuitry described by the provided Verilog to be able to make your modifications. **(PRELAB)**
2. Determine a sequence of steps 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. Show this to your TA. **(PRELAB)**
3. Draw a state diagram for your controller starting with the register load states provided in the example FSM. **(PRELAB)**
4. Modify the provided FSM to implement your controller and synthesize it. You should only modify the control module. **(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.

   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. **(PRELAB)**
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.) **(PRELAB)**
7. After you are satisfied with your simulations, download and test the functionality of the circuit on the FPGA board.

Part III (Optional)

Division in hardware is the most complex of the four basic operations. Addition, subtraction and multiplication are much easier to build in hardware than division. For this part, you will be designing 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.



Divisor 00011
Register A 00000
Dividend 0111
00000 1110 Shift left
1 1101 1110 Subtract Divisor from Register A
00000 1110 Add Divisor to Register A
00000 111 0 set q₀ to 0 or 1
00001 1100 Shift left
1 1110 1100 Subtract Divisor from Register A
00001 1100 Add Divisor to Register A
00001 110 0 set q₀ to 0 or 1
00011 1000 Shift left
0 0000 1000 Subtract Divisor from Register A
00000 100 1 set q₀ to 0 or 1
00001 0010 Shift left
1 1110 0010 Subtract Divisor from Register A
00001 0010 Add Divisor to Register A
00001 001 0 set q₀ to 0 or 1
Remainder
Quotient

Figure 5: An example showing how the restoring divider works.

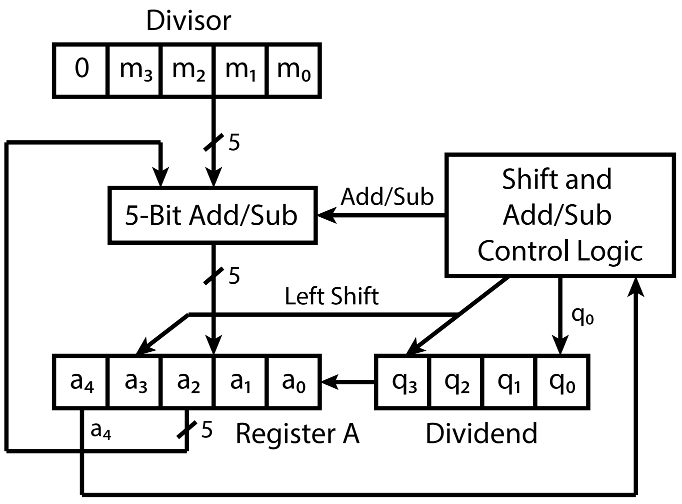


Figure 6: Block diagram of restoring divider.

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*.

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 6. 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 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 realizes your circuit.
6. Simulate your circuit with ModelSim for a variety of input settings, ensuring the output waveforms are correct.
7. After you are satisfied with your simulations, download and test the functionality of the circuit on the FPGA board.