Laboratory Exercise 6

Finite State Machines

Revision of November 4, 2022

The purpose of this lab is to:

1. learn how to write Finite State Machines (FSMs) in Verilog
2. learn how to use an FSM to control the sequencing of logical operations.

1 Workflow

For each part of the lab you should begin by writing and testing Verilog code and compiling it with Quartus. You should be prepared to show schematics, Verilog, and simulations to your TA, if requested. You must simulate your circuit with ModelSim using reasonable test vectors written in the format used in Lab 2 for the simulation files.

For Parts I and II, you are given a lot of code in the form of templates. This provides you with some good examples to use as models for your own code, but they also allow you to focus on the key elements relevant to building the required control sequences without having to write the supporting infrastructure. The amount of code you need to write in these parts is relatively small. Most likely, more of your time will be spent understanding all of the code presented to you and how to modify it. Learning to read other code is also a good thing to learn as you will do that a lot in industry. A good way to start is to reverse engineer a schematic from the code.

2 Test Plans: Getting More from Your Simulations

As the complexity of the circuits you build increases, the importance of simulation also increases. Thinking about how you are going to test your circuit is an integral part of doing a design. Without thinking about testing as you do your design, it is possible that you will build a circuit that you cannot properly test, or it will be extremely difficult to test. While the circuits you build for these lab exercises are still basic and straightforward, the goal at this time is for you to start thinking more about how you will do your testing.

Here are the steps you should do. Be prepared to discuss your *Test Plans* with your TA if requested.

1. Write out a *Test Plan*. The *Test Plan* lists everything you want to test to show your circuit is working, how you are going to do the test, and the correct result to expect. For example, if you are going to test a 4-bit adder, your test plan might look like:

   Test carry in = 0 Add 0000 + 1111 with carry in of 0. Expected output = 1111, carry out = 0;

   Test carry in = 1 Add 0000 + 1111 with carry in of 1. Expected output = 0000, carry out = 1;

   And so on Other tests ...

2. As you execute each test, you can document the behaviour of a test by annotating a screen capture of the waveforms to show the inputs and the outputs.

3. If something is not working, indicate the inputs, but then show in the simulation where it is not doing what you expect. This is important if you want to discuss the issue with a TA, or your manager if you are working.

The goal of doing the above is to make you think more about the purpose of simulation and how to get the most out of doing simulation.

3 Part I

In this part you will implement a basic finite state machine (FSM) in Verilog. All FSMs you write in Verilog should follow this structure or you can get into lots of trouble.

We wish to implement a 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 for an example input sequence. A state diagram for this FSM is shown in Figure 2.

Figure 3 shows a partial Verilog file for the required state machine. It is the template code in `part1_template.v` that you will need to complete for this part. 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.

Note that the top-level module of the template has the following signature description:

```
module part1(Clock, Resetn, w, z, CurState);
```

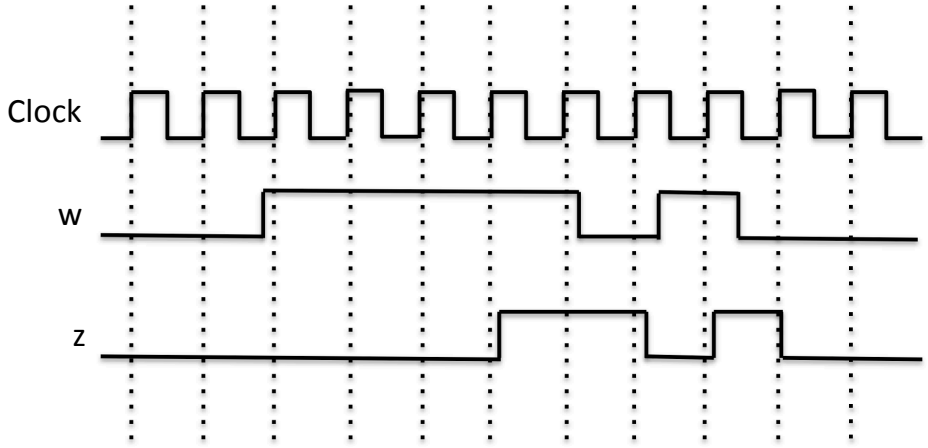

Figure 1: Required timing for the output z.

This circuit uses an active-low synchronous reset called `Resetn`. The input of the sequence detector is `w` and the output is `z`. The current state of the FSM is output with `CurState`.

3.1 What to Do

Perform the following steps:

1. Begin with the template code provided online in `part1_template.v`.
2. Complete the state table and the output logic. Look for the question marks (?).
3. Simulate your `part1` module with ModelSim to satisfy yourself that your circuit is working. Be prepared to justify that your test cases are enough to give confidence that your circuit is working. When you are satisfied with your simulations, you can submit to the Automarker.
4. Create a new Quartus project for your circuit. You will need a top-level module to make connections from the instantiation of your `part1` module to the switches, pushbutton and LEDs of the DE1-SoC board. The connections used are shown in Table 1.

   By using the pushbutton `KEY[0]` as the `Clock` input, you will be able to manually create clock pulses so that you can observe the transitions through the states.

   Simulate your new circuit with the DE1-SoC connections with ModelSim to ensure that you have done the connections correctly.
5. Compile the project to generate a bitstream to make sure your code can be synthesized.
6. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the various inputs and observing the outputs.

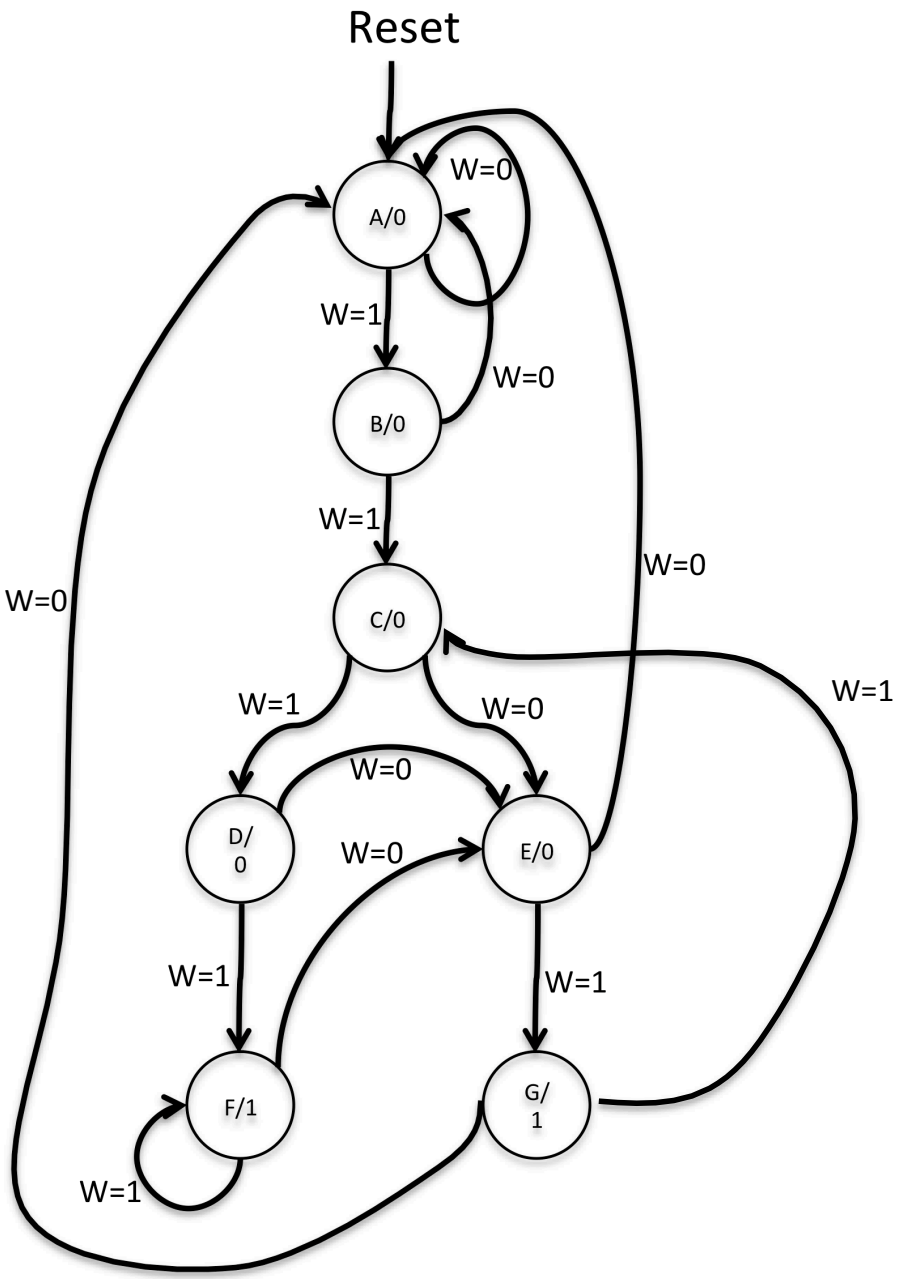

Figure 2: A state diagram for the FSM.

```
// Resetn synchronous reset when 0
// w input to detector
// z is output from detector
// Clock clock signal
// CurState outputs current state

module part1(Clock, Resetn, w, z, CurState);
  input Clock;
  input Resetn;
  input w;
  output z;
  output [3:0] CurState;

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

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

  //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(Resetn == 1'b0)
      y_Q <=  A; // Should set reset state to state A
    else
      y_Q <= Y_D;
  end // state_FFS

  // Output logic
  // Set z to 1 when in relevant states
  assign z = ((y_Q == F) | (y_Q == G));
  assign CurState = y_Q;
endmodule
```

Figure 3: Verilog code for the FSM.

part1 Port Name	Direction	DE1-SoC Pin Name
Clock	Input	KEY[0]
Resetn	Input	SW[0]
w	Input	SW[1]
z	Output	LEDR[9]
CurState	Output	LEDR[3:0]

Table 1: Module part1 mapping to DE1-SoC pin names

4 Part II

Please note that there is a lot written here. Most of it is explanation and guidance, so please read carefully.

A *finite state machine* (FSM) on its own, like the one built in Part I, cannot do much and is not what you usually do with an FSM except to teach how to build an FSM. The primary use of FSMs is to act as the main control for digital systems that require functions like sequencing or responding in different ways to some stimuli. This part will show you how to use an FSM to do something more interesting than recognizing a pattern of bits. To help you focus on the FSM design, you are provided an entire datapath and example FSM that performs a computation. Your task will be to change the FSM to do a different computation.

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 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 an FSM that controls the datapath so that it computes $A^2 + C$. Observe that the example code loads four values, despite using only two of the values in the computed result. This will give you a head start because you will need all four values loaded for the task you will do. Also, often when you inherit code, there might be parts of it that may not be used or be relevant because of how the code evolved. It is up to you to figure out what is relevant for the task you are given. Feel free to modify the code you are given, if you feel it is necessary to achieve the final result required. Note that the provided template code has an additional ResultValid port. While it is not used in the sample design, it is required for compatibility with the auto-marker. It's expected functionality will be described later in the handout.

Using the given datapath, you are required to implement an FSM that controls the datapath

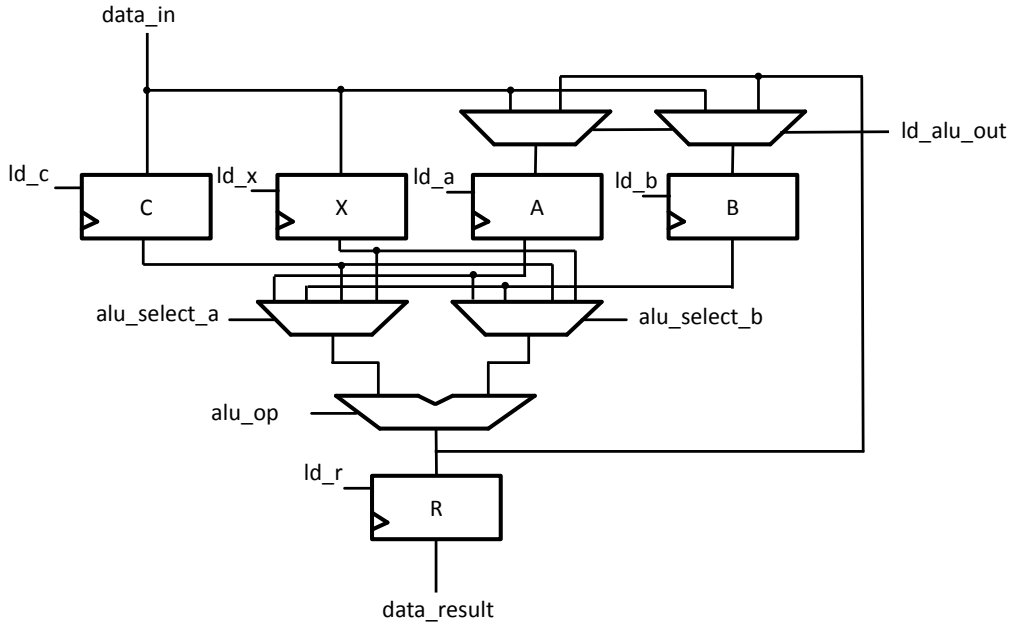

Figure 4: Block diagram of datapath.

so that it performs the 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. Resets are not shown, but do not forget them. 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_R, that captures the output of the ALU and displays the value in binary on the LEDs and in hex 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 you must modify the provided circuit to operate in the following manner. After an active low synchronous reset using `Resetn`, you will input the value for R_A on switches at

Table 2: Register contents and control signals for computing $A^2 + C$

	Reset	1	2	3	4	5	6	7	
data_in		5	4	3	2	2	2	2	
RA		0	5	5	5	5	25	25	
RB		0	0	4	4	4	4	4	
RC		0	0	0	3	3	3	3	
Rx		0	0	0	0	2	2	2	
RR		0	0	0	0	0	0	28	
ld_a		1	0	0	0	1	0	1	
ld_b		0	1	0	0	0	0	0	
ld_c		0	0	1	0	0	0	0	
ld_x		0	0	0	1	0	0	0	
ld_alu_out		0	0	0	0	1	0	0	1 = select alu output
alu_select_a		0	0	0	0	0	0	0	0 = select A
alu_select_b		0	0	0	0	0	2	0	2 = select C
alu_op		0	0	0	0	1	0	0	0 = add, 1 = multiply
ld_r		0	0	0	0	0	1	0	

the port `DataIn`. When `Go` is set and released, R_A will be loaded and then you will input the next value at `DataIn` that will be loaded into R_B. Set and release `Go` again. Repeat these steps to load R_C and R_X. Computation will start after `Go` is set and released for loading R_X. When the computation is finished, the final result will be loaded into R_R. This final result should be output on port `DataResult` in binary.

A naive implementation would require, at most, 5 operations to calculate the result. Again we provide some margin for error: hence, once the input data is loaded into the appropriate registers, the computation **should not** take more than 7 cycles which. When the computation is completed, the `ResultValid` port should be set to high at the same cycle that the new valid result is output to `DataResult`. This will indicate to the Automarker (or any downstream block) that the result on the `DataResult` port is ready and valid. This timing is strictly enforced. `ResultValid` should remain high and `DataResult` should maintain its valid value from the previous computation, indefinitely, until new input is provided, i.e., the `Go` is set to 1. The clock for the circuit is input at port `Clock`.

4.1 What to Do

Perform the following steps:

1. Examine the Verilog code provided online in `part2_template.v`. This is a major

step in this part of the lab. You will not need to write much Verilog, but you will need to fully understand the provided Verilog to make your modifications and build a simulation script. Here's how to read the code and what to look for:

(a) The `part2` module has two modules called `datapath` and `control`, which is the explicit partitioning of the control logic and the datapath logic. The code has been organized this way to make it very clear where the control logic is written versus the datapath logic. Most important is that you can clearly see what control signals coming from the control logic connect to the datapath.

(b) Read the datapath code and identify all the components shown in Figure 4 when looking at the datapath module.

(c) Identify all the control signals shown in Figure 4 and where they are generated in the control module. You should see that all the control signals are generated by the FSM in the control module.

(d) Find the case statement that defines the state table for the FSM. Also, find the case statement that defines the outputs that are set in each state. It is important that the state transitions and the output logic be in separate case statements. This makes it explicitly clear what logic is being generated from your Verilog. If you start mixing the state transitions with the outputs, the code becomes very challenging to understand, with the risk that the Verilog compilation will also do something you did not intend.

By looking at the state transitions and what outputs are set in each state, observe how the loading of the four registers is done and how the sequence of the loading is done according the specification. How does the FSM handle the setting and releasing of the input `Go` signal? Ultimately, the `Go` signal will be connected to a pushbutton on the DE1-SoC board, which is a mechanical device that runs in human time. If the clock is running at 50 MHz how many clock pulses might there be during the time that the `Go` signal is set high by a human? How does the FSM account for human time? Take this into account when doing your simulations.

You may find it helpful to draw a state diagram for the FSM because you can then just modify it for Step 5.

(e) Which states do the actual computation? Identify those states and find the sequence of control signals. Observe how the computation is done by using Figure 4 and seeing how that sequence of control signals does the required operation. Can you figure out what is being computed by looking at the FSM?

2. Table 2 shows a table of the sequence of register contents and control signals to do the computation of $A^2 + C$. You should be able to see the control signals changing the same way in `part2_template.v`. in the controller state machine output logic. See the supplemental notes *NotesForControlContentSignalsLab6.pdf* for a further explanation of Table 2.

3. Simulate `part2_template.v` using ModelSim starting with the values in Table 5 and some other values of your choosing. Remember that the registers are only eight bits wide, so make sure all values will fit in the correct range.

   Observe the state register of the FSM so that you can watch the state transitions. This is usually helpful when things are not working as getting the control sequences correct is usually the hardest part of a design.

   You should be able to use the same simulation script after you modify the circuit to do the new computation, except you may need to account for there being more operations being performed, so more simulation time will be needed.

   Before making changes to any design, it is always good to start from a known state, which in this case is a working example and a working simulation script. You are provided `part2_template.v`, which is working code and you will build the corresponding simulation script. After this step, you will have both before you start making changes to the code. If something goes wrong, go back to when the design last worked and figure out what change caused the error.

4. Following the model of `part2_template.v` and Table 5, build the table for computing $Ax^2 + Bx + C$.

5. Draw a state diagram for your controller starting with the register load states provided in `part2_template.v`.

6. Modify `part2_template.v` to implement your controller. You should only need to modify the control module.

7. Simulate your circuit with ModelSim for a variety of inputs. Since you are modifying the controller it is recommended that you simulate the controller module by itself first. Only when you are satisfied that it is working individually should you add it into the full design for a full system simulation. Why is this approach better? (Hint: Consider the case when your design has 20 different modules.) When you are satisfied with your simulations, you can submit to the Automarker,

8. Create a new Quartus project for your circuit. You will need a top-level module to make connections from the instantiation of your `part2` module to the switches, pushbuttons, LEDs and HEX displays of the DE1-SoC board. The connections used are shown in Table 3.

   Remember that the pushbuttons on the DE1-SoC board are active low.

   Simulate your new circuit with the DE1-SoC connections with ModelSim to ensure that you have done the connections correctly.

9. To examine the circuit produced by Quartus 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

part2 Port Name	Direction	DE1-SoC Pin Name
Resetn	Input	KEY[0]
DataIn	Input	SW[7:0]
Go	Input	KEY[1]
DataResult[7:0]	Output	LEDR[7:0]
ResultValid	Output	LEDR[8]
DataResult[3:0]	Output	HEX0
DataResult[7:4]	Output	HEX1

Table 3: Module part2 mapping to DE1-SoC pin names

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.

10. Compile the project to generate a bitstream to make sure your code can be synthesized.

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

5 Part III (Optional - For Bonus Marks)

In this part, you will have to build a complete datapath and corresponding state machine. We recommend that you still have two separate modules for the datapath and the control just to help you understand the separation of what is in a datapath and what is in a controller. In Part II, the control logic just generated control signals that were inputs to the datapath. In this part, you will see that there is a signal that needs to go from the datapath into the controller so make sure you account for it accordingly.

Division in hardware is the most complex of the four basic arithmetic 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 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*. 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*. You may use the subtract (−) and addition (+) operators in Verilog to perform the subtraction and addition. The *1* in the MSB of *Register A* means that the value in *Register A* after the subtraction is a negative number, meaning that the *Divisor* is larger than the original value in *Register A*. That is why *Register A* is *restored* by adding back the *Divisor*.

This sequence of steps 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*.

5.1 What to Do

The top-level module of your design should have the following declaration:

```
module part3(Clock, Resetn, Go, Divisor, Dividend, Quotient, Remainder,
                                ResultValid);
```

`Divisor` and `Dividend` are the input values for the division and `Quotient` and `Remainder` are the outputs of the division with the `ResultValid` port being used to indicate that the output is valid, similar to Part II. When `Divisor` and `Dividend` are both valid, `Go` is set and released in order to simultaneously load both input values into registers.

When `Go` is released, input values should be loaded into registers in 1 clock cycle. Your circuit should then perform the division in exactly 4 cycles i.e. 1 cycle per bit of the *Dividend*. `ResultValid` should remain high and `Quotient` and `Remainder` should maintain their final values until new input is provided, i.e., `Go` is set to 1. `Resetn` is a synchronous active low reset.

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. Do not forget the clock and resets.

2. Draw the state diagram to control your datapath. Check it by hand simulating the example shown in Figure 5. Hand simulation just means to work through the steps

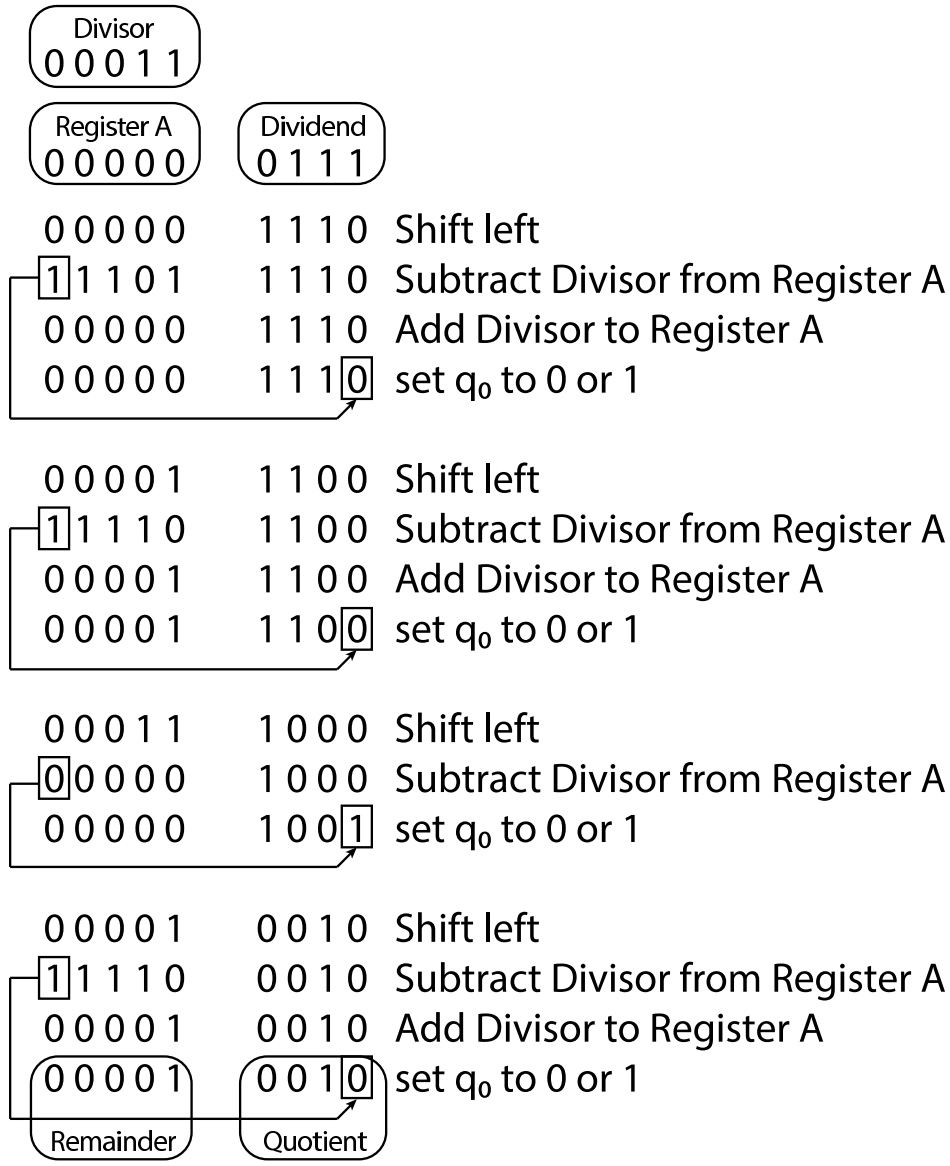

Figure 5: An example showing how the restoring divider works.

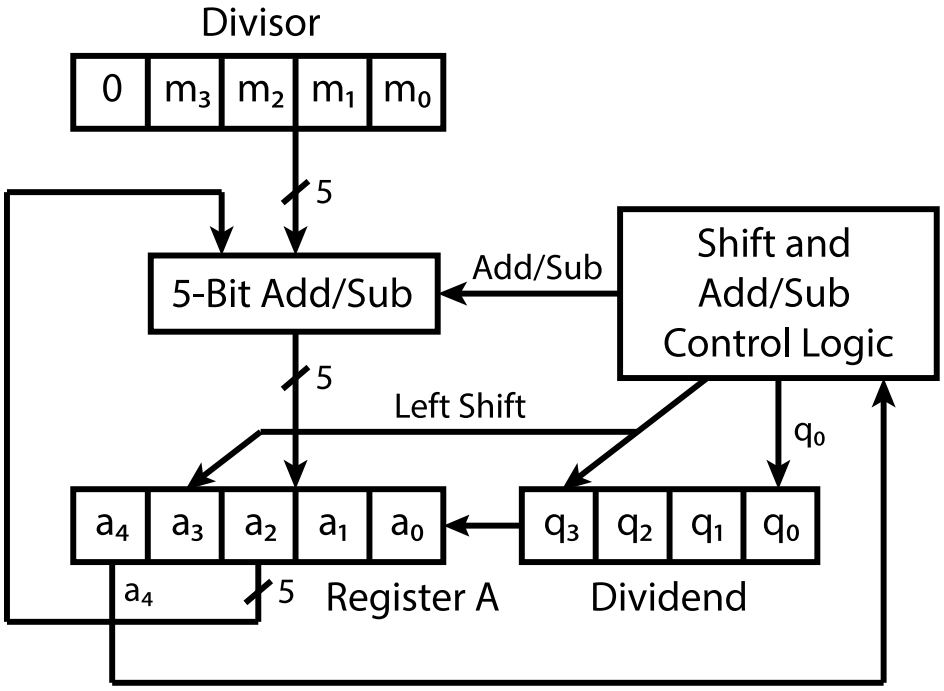

Figure 6: Block diagram of restoring divider.

using your schematic and state diagram to check whether you can do the required operations before going through the effort of setting up the simulator. This may not catch all bugs, but it is a good step to make sure you have a design that has a chance of working.

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

6. Simulate your circuit with ModelSim for a variety of input settings, ensuring the output waveforms are correct. Start with Figure 5 as an example because it shows you all the steps with the values that should be in the registers at each step.

7. Create a new Quartus project for your circuit. You will need a top-level module to make connections from the instantiation of your `part3` module to the switches, pushbuttons, LEDs and HEX displays of the DE1-SoC board. The connections used are shown in Table 4.

   Remember that the pushbuttons on the DE1-SoC board are active low.

part3 Port Name	Direction	DE1-SoC Pin Name
Clock	Input	Clock_50
Resetn	Input	KEY[0]
Divisor	Input	SW[3:0]
Dividend	Input	SW[7:4]
Go	Input	KEY[1]
Divisor	Output	HEX0
Dividend	Output	HEX2
Quotient	Output	HEX4
Remainder	Output	HEX5
Quotient	Output	LEDR[3:0]
ResultValid	Output	LEDR[4]

Table 4: Module part3 mapping to DE1-SoC pin names

Simulate your new circuit with the DE1-SoC connections with ModelSim to ensure that you have done the connections correctly.

8. Compile the project to generate a bitstream to make sure your code can be synthesized.

9. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the various inputs and observing the outputs.

6 Submission

6.1 Part I and Part II

Please submit part1.v and part2.v. For module names and port names, please follow the provided templates.

6.2 Part III

You may optionally submit part3.v for bonus marks. For Part III, you need to submit a file named part3.v with the following module in it:

```
module part3(Clock, Resetn, Go, Divisor, Dividend, Quotient, Remainder,
                                ResultValid);
```