

# Laboratory Exercise 6

## Finite State Machines

Revision of October 27, 2022

In this lab, you will learn how to write Finite State Machines (FSMs) in System Verilog and how to use an FSM to control the sequencing of logical operations.

For Parts I and II, you are provided code 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.

## 1 Part I

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

You must implement a FSM with an input  $w$  and an output  $z$ , that recognizes two specific sequences of inputs. When  $w = 1$  for four consecutive clock pulses, or when the sequence 1101 appears on  $w$  across four consecutive clock cycles, the value of  $z$  has to be 1; otherwise,  $z = 0$ . Overlapping sequences are allowed, so that if  $w = 1$  for five consecutive clock cycles the output  $z$  will be equal to 1 after the fourth and fifth cycles. 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.

Listing 1 shows a partial System Verilog file for the FSM. It is the template code in `part1_template.sv` 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 run on the FPGA properly.

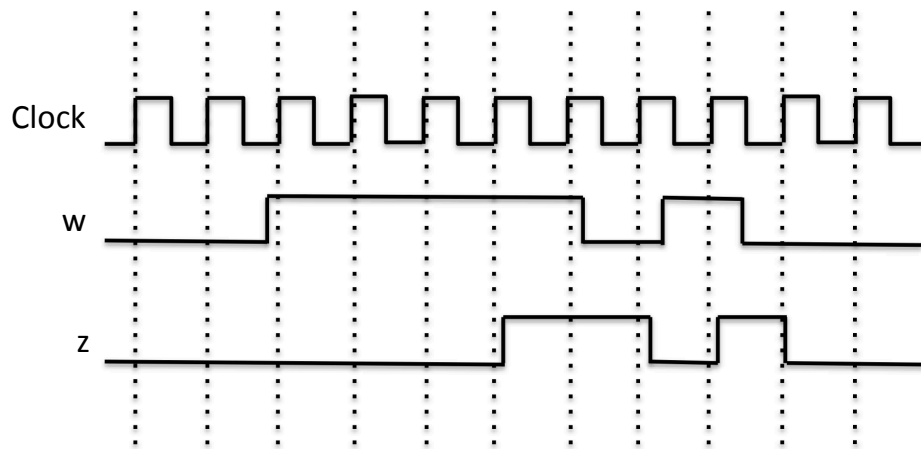


Figure 1: Required timing for the output  $z$ .

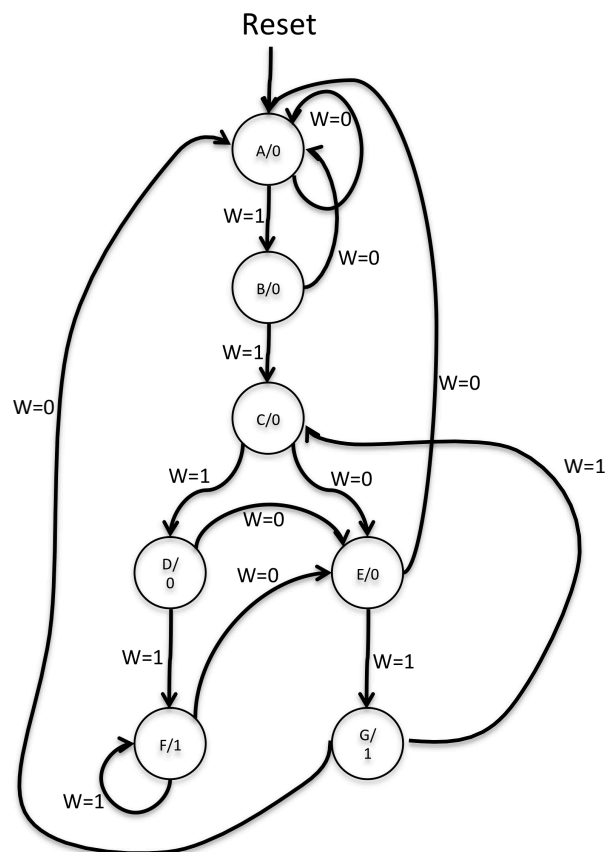


Figure 2: A state diagram for the FSM.

```

module part1(
    input logic Clock,
    input logic Reset,
    input logic w,
    output logic z,
    output logic [3:0] CurState
);
    typedef enum logic [3:0] {A = 4'd0, B = 4'd1, C = 4'd2, D =
        4'd3, E = 4'd4, F = 4'd5, G = 4'd6} statetype;

    statetype y_Q, Y_D;

    //State table
    always_comb begin
        case (y_Q)
            A: begin
                if (!w) Y_D = A;
                else Y_D = B;
            end
            B: // Complete
            C: // Complete
            D: // Complete
            E: // Complete
            F: // Complete
            G: // Complete
            default: // Complete
        endcase
    end // state_table

    // State Registers
    always_ff @(posedge Clock) begin
        if(Reset == 1'b1)
            // Should set reset state to state A
        else
            y_Q <= Y_D;
    end // state flip flops

    assign z = ((y_Q == F) | (y_Q == G)); // Output logic

    assign CurState = y_Q;
endmodule

```

Listing 1: Code template for FSM in part 1

## 1.1 What to Do

Perform the following steps:

1. Begin with the template code provided online in `part1_template.sv`.
2. Complete the state table and the output logic.
3. Simulate your `part1` module with ModelSim to satisfy yourself that your circuit is working. When you are satisfied with your simulations, you can submit to the Automarker.

## 1.2 Running on FPGA

To run your design on an FPGA, use the mapping shown in Table 1.

<code>part1</code> Port Name	Direction	DE1-SoC Pin Name
<code>Clock</code>	Input	KEY[0]
<code>Reset</code>	Input	SW[0]
<code>w</code>	Input	SW[1]
<code>z</code>	Output	LEDR[9]
<code>CurState</code>	Output	LEDR[3:0] & HEX[0]

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

## 2 Part II

**Warning:** Please note that this part has a lot of text. Most of it is explanation and guidance, so please **read carefully**.

In this part, you must design an ALU which can be used to calculate equations that involve several additions and multiplications. To do this, you will learn about using **control paths** and **datapaths**, which are used to implement more complex hardware designs.

### 2.1 Using FSMs in larger designs.

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. Part 2 will show you how to use an FSM to do something more interesting than recognizing a pattern of bits.

**Control path and Datapath:** 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. Using an FSM for the control path and an ALU for the datapath is fundamental for how CPUs work, which you have started to learn about in class. To show you how to write code to implement a control path and a datapath for an ALU, you are provided with code for a sample ALU.

### 2.2 Sample ALU

For Part 2, you are given a sample ALU that implements an FSM control path and a datapath to compute  $A^2 + B$ . The same code is provided on the Lab6 page as `part2_template.sv`.

#### 2.2.1 Datapath

The datapath implemented in the sample code is shown in Figure 3. Study this datapath and the provided code **carefully** to make sure you understand how the code implements the control path and datapath. You must modify this template to implement an expanded

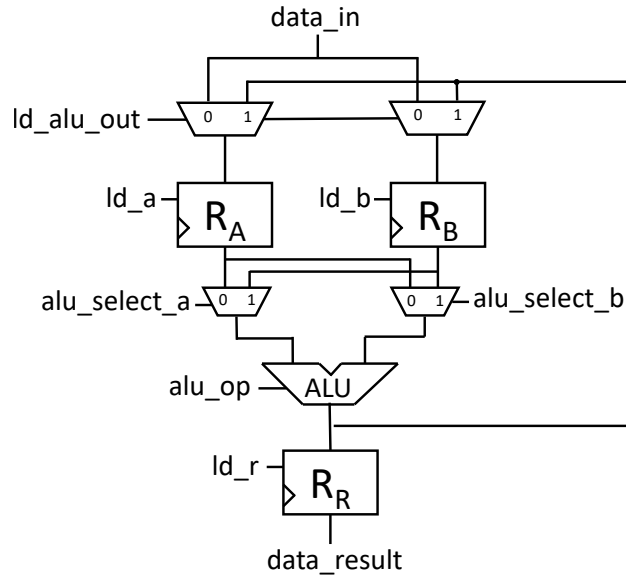


Figure 3: Block diagram of provided datapath.

datapath for Part 2. As is normal, the clock and reset signals are not shown in Figure 3 to keep the diagram neat, but it is important that you **do not** forget about them. **Reset** is a synchronous active-high reset. Both the provided datapath and the expanded datapath use 8-bit unsigned values. Assume for both cases that the input values are small enough to not cause any overflows at any point in the computation, i.e., no results will exceed  $2^8 - 1 = 255$ .

There are two registers  $R_A$ ,  $R_B$  used at the start to store the values of  $A$  and  $B$ , respectively. In a real CPU, these registers may store the result of some previous computation. However, for our datapaths, we load these registers before we start computation. There is one output register,  $R_R$ , that captures the output of the ALU. Each of these registers has an enable signal, shown with the prefix `ld_` in the Figure. Additionally, the  $R_A$  and  $R_B$  registers can be overwritten during the computation. This is supported using the 2-to-1 multiplexers controlled by the `ld_alu_out` signal. To select the inputs to the ALU, we use two 8-bit wide, 2-to-1 multiplexers, controlled by the `alu_select_a` and `alu_select_b` signals.

### 2.2.2 Control path

The provided sample code also implements an FSM for the control path of the ALU. You are **strongly encouraged** to draw the state diagram of this FSM, showing all the inputs and outputs. This will help you to better understand the code and will also serve as a starting point for the FSM you have to write for the expanded ALU.

The FSM first goes through a set of states to load the input registers  $A$  and  $B$ . We load each register in turn from the `data_in` input when the `Go` signal is asserted. This is implemented

in the `S_LOAD` states. However, we cannot be sure that `Go` will only be 1 for a single cycle. This is the case when you run this code on the FPGA and `Go` is connected to a `KEY`. Since the FPGA uses a 50 MHz clock, when you press the `KEY`, many millions of clock cycles will elapse. So if we load  $A$  and then  $B$  when `Go` is 1, we would load both in the same key press. To avoid this, we need to add a `S_LOAD_WAIT` state for each input. When `Go` becomes 1, we load the input and then go to the `S_LOAD_WAIT` state. There, we wait for `Go` to become 0 before loading the next input. This mimics the user pressing and then releasing the `KEY` input. The computation of  $A^2 + B$  occurs in two stages in states `S_CYCLE_0` and `S_CYCLE_1`. When computation is finished, the final result will be loaded into  $R_R$ . This final result should be output on port `DataResult` in binary.

The `always_comb` block labeled `enable_signals` sets the appropriate signals to the datapath for each state in the FSM. Table 2 shows the signals in the control path for each cycle of operation.

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

	Reset	1	2	3	4	5	6	7	
<code>Go</code>		1	0	1	0	0	0	0	
<code>data_in</code>		5	5	4	4	-	-	-	- = don't care.
<code>State</code>		0	2	3	4	5	6	1	
$R_A$		0	5	5	5	5	25	25	
$R_B$		0	0	0	4	4	4	4	
$R_R$		0	0	0	0	0	0	29	
<code>ld_a</code>		1	0	0	0	1	0	1	
<code>ld_b</code>		0	0	1	0	0	0	0	
<code>ld_r</code>		0	0	0	0	0	1	0	
<code>ld_alu_out</code>		0	0	0	0	1	0	0	1 = select alu output
<code>alu_select_a</code>		0	0	0	0	0	0	0	0 = select A
<code>alu_select_b</code>		0	0	0	0	0	1	0	1 = select B
<code>alu_op</code>		0	0	0	0	1	0	0	0 = add, 1 = multiply
<code>result_valid</code>		0	0	0	0	0	0	1	

You must write a `.do` file to simulate the provided ALU code to understand how it works. Once you are done this step, you can move on to expanding this ALU to implement a more complex function.

solve line new notes  
 expand  
 select mus

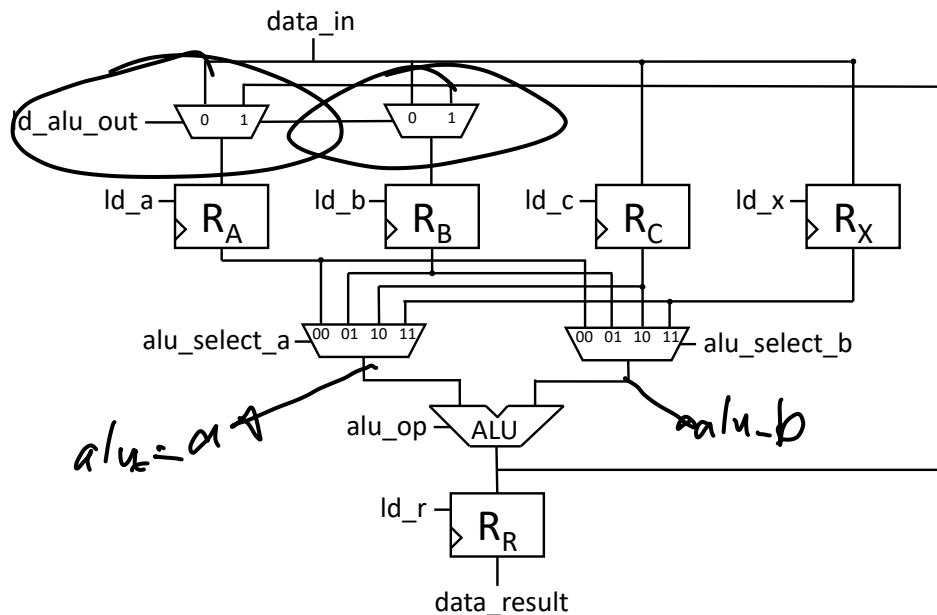


Figure 4: Block diagram of expanded datapath.

## 2.3 Expanded ALU

You must modify the provided ALU to implement the equation:  $Ax^2 + Bx + C$

Figure 4 shows the block diagram of the datapath you must build. You must change the control and datapath for the provided code to support this expanded ALU. You must modify the code to support four registers:  $R_A$ ,  $R_B$ ,  $R_C$  and  $R_X$ .  $R_C$  and  $R_X$  are loaded just like  $R_A$  and  $R_B$ . Data.in values should be loaded into registers in the order:  $R_A$ ,  $R_B$ ,  $R_C$ ,  $R_X$ . Computation will start after Go is set and released for loading  $R_X$ . When computation is finished, the final result should be loaded into  $R_R$ . This final result should be output on port DataResult in binary.

Once the input data is loaded into the appropriate registers, the computation **should not** take more than 7 cycles. When the computation is completed, the ResultValid port should be set to 1 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. 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 provided sample ALU already meets the above requirements but you must make sure your modified ALU does not violate them.

Finally, the ALU only needs 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.



## 2.4 What to Do

Perform the following steps:

1. Examine the Verilog code provided online in `part2_template.sv`. Be sure to remain this file as `part2.sv`.
2. Prior to coming to lab, create a new table, similar to Table 2 to implement  $Ax^2 + Bx + C$ . This will inform the new signals you will need in your datapath to support more registers.
3. Prior to coming to lab, add additional states to your FSM State diagram to support loading more registers and performing the additional operations.
4. Modify the control path module to implement the changes to your FSM.
5. Simulate your design by providing 4 values for the registers and checking that your design works.
6. When you are satisfied, you can submit to the automarker.

## 2.5 Running on the FPGA

To run on the FPGA, use the connections shown in Table 3.

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

Table 3: Module `part2` mapping to FPGA pin names

A ↗ A

### 3 Part III (Optional - For Bonus Marks)

In this bonus part, you must design a control and datapath to implement division in hardware. 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.

First, you must understand how division is performed in hardware. 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*. Once *Go* is asserted, your circuit must register the input values in 1 clock cycle and then perform the division in exactly 4 cycles i.e., 1 cycle per bit of the *Dividend*.

#### 3.1 What to Do

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

```
module part3(
    input logic Clock,
    input logic Reset,
    input logic Go,
    input logic [3:0] Divisor,
    input logic [3:0] Dividend,
    output logic [3:0] Quotient,
    output logic [3:0] Remainder,
    output logic ResultValid
);
```

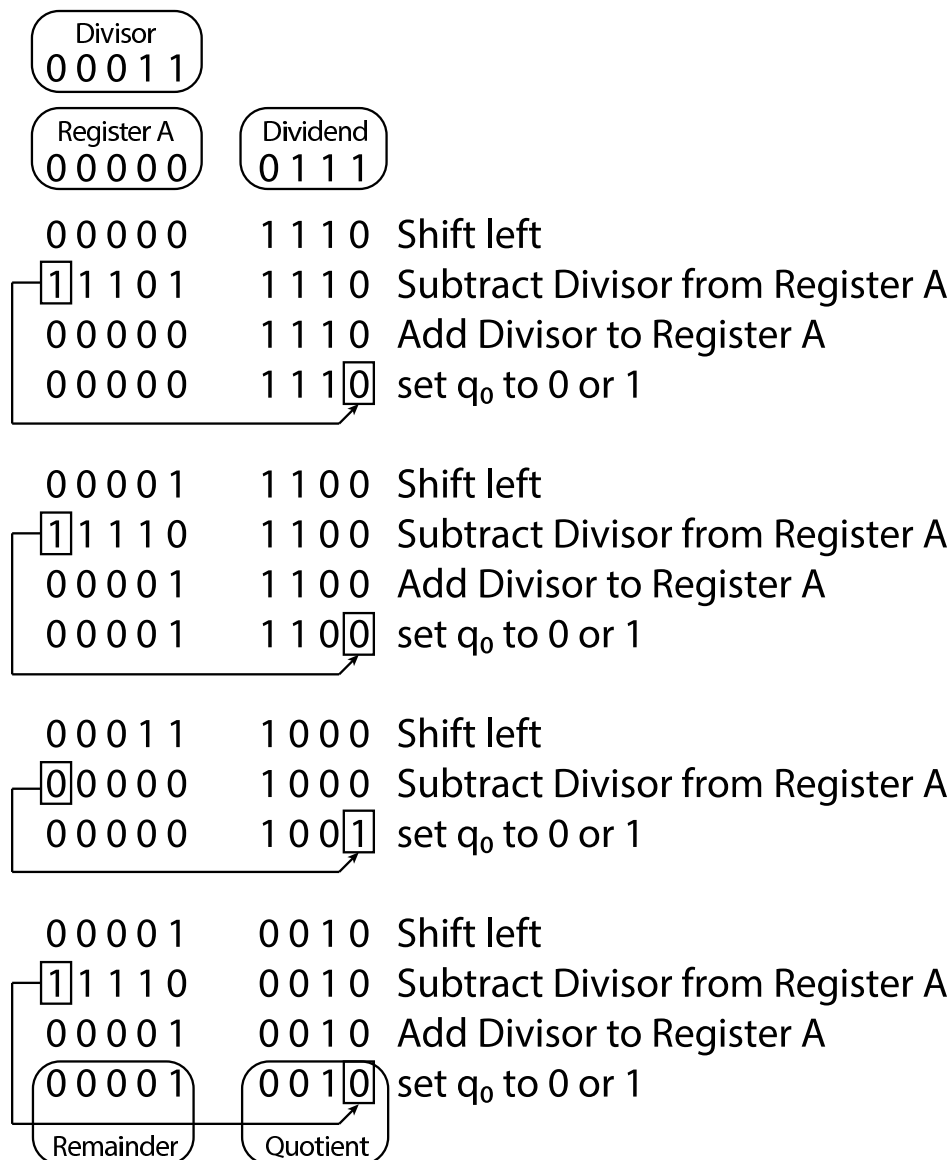


Figure 5: An example showing how the restoring divider works.

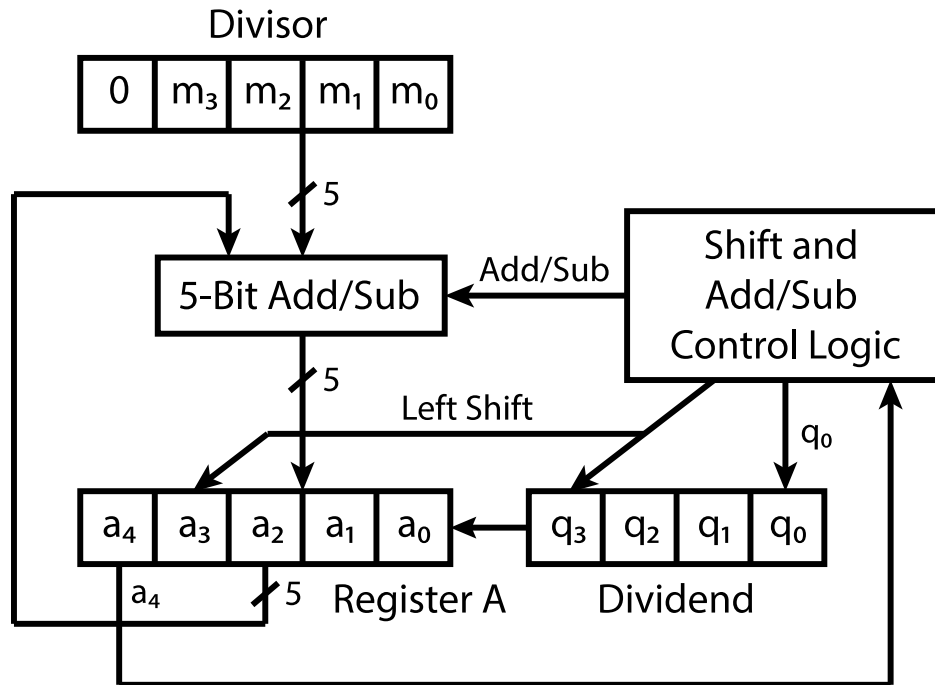


Figure 6: Block diagram of restoring divider.

1. **Divisor** and **Dividend** are the input values for the division.
2. **Go** is used to load the **Divisor** and **Dividend**, similar to Part 2.
3. **Quotient** and **Remainder** are the outputs of the division.
4. **ResultValid** indicates a valid output, similar to Part 2. **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. **Reset** is a synchronous active high 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 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 code that realizes your circuit. Structure your code in the same way as you were shown in Part 2.
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. Once you are satisfied with your simulations, you can submit your code for marking.

## 3.2 Running on the FPGA

The mapping to run your code on the FPGA is shown in Table 4.

part3 Port Name	Direction	DE1-SoC Pin Name
Clock	Input	Clock_50
Reset	Input	KEY[0]
Divisor	Input	SW[3:0] & HEX0
Dividend	Input	SW[7:4] & HEX2
Go	Input	KEY[1]
Quotient	Output	LEDR[3:0] & HEX4
Remainder	Output	LEDR[7:4] & HEX5
ResultValid	Output	LEDR[9]

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

## 4 Submission

### 4.1 Part I and Part II

Please submit files named `part1.sv` and `part2.sv`. For module names and port names, please use the ports in the provided templates.

### 4.2 Part III

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

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