Laboratory Exercise 5

Finite State Machines

Revision of October 23, 2024

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 building and debugging your code.

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.

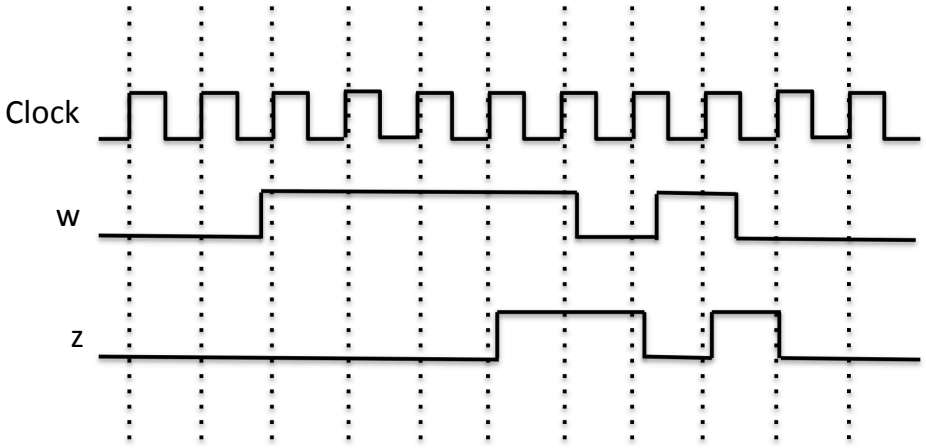

Figure 1: Required timing for the output z.

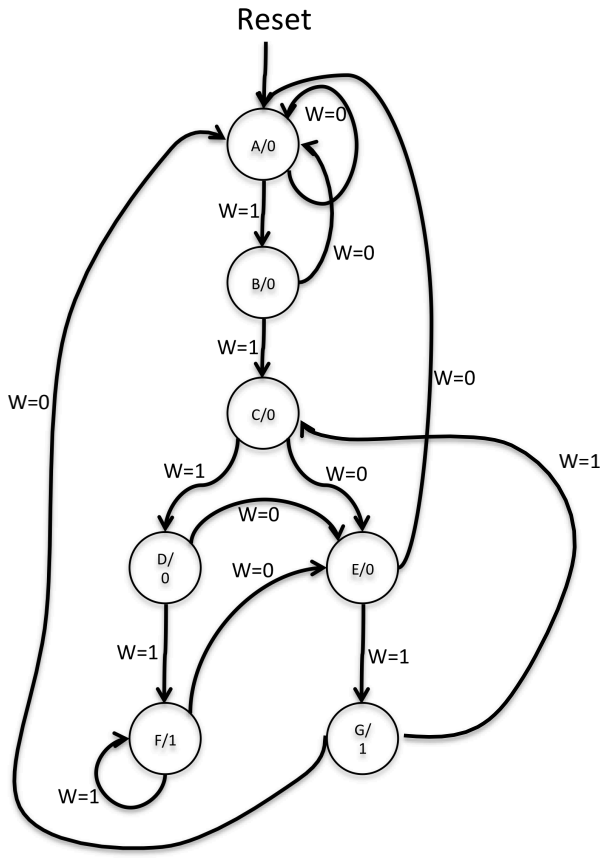

Figure 2: A state diagram for the FSM.

```
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 (all three steps should be completed prior to coming to lab):

Pre-Lab Work:

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 Optional: Running on FPGA

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

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

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

2 Part 2

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 main application of FSMs is to serve as the primary control for digital systems, managing tasks like sequencing or responding to various stimuli.

In this section, you'll learn how to leverage an FSM to achieve more complex functions beyond simple pattern recognition. Specifically, you will apply the knowledge gained from building an FSM in Part I and implementing a sequential ALU in Lab 3 to *build your own processor*. You'll explore the use of control paths and datapaths, essential for implementing complex hardware designs such as processors.

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, multiplexers, and registers which are common datapath components. 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.

We will apply this knowledge to implement the processor shown in Figure 3. In the figure, the blue sections represent the processor's datapath, while the red sections represent the control path.

2.1 Datapath

The processor's datapath can be thought of as an extension of the sequential ALU from Lab 3. This datapath consists of a couple of 16-bit registers, an instruction register (IR), a multiplexer, and an ALU. As seen in Figure 3, the multiplexer selects what inputs to feed into the ALU among the values from registers R0, R1, IR, and R. This data is either fed into Register A or directly into the ALU for computation, after which the ALU output is stored in register R. The output of the multiplexer can also be saved to either R0 or R1; this path is primarily used to store output values from R into register R0 or R1.

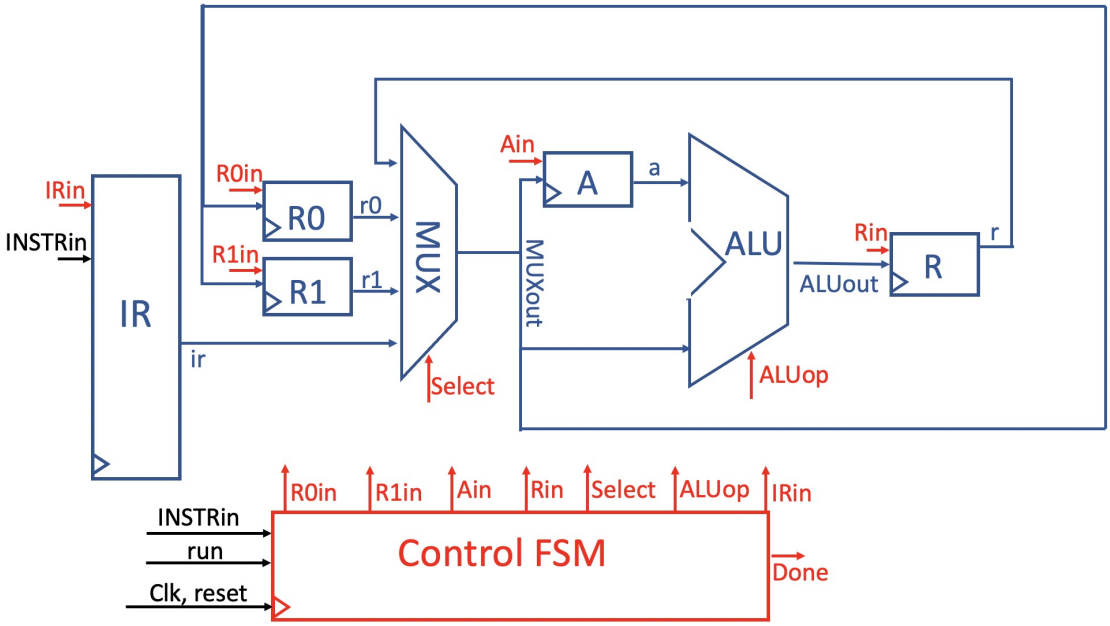

Figure 3: A Simple Processor

2.2 Control Path

The main building block of the processor's control path is the control FSM. This FSM manages the select lines of the multiplexer, determining which registers among R0, R1, IR, and R will drive MUXout. It also controls data transfer across registers via the enable signals A_{in}, R_{in}, $R0_{in}$, and $R1_{in}$ which control when registers `A`, `R`, `R0` and `R1` updated their stored values with new inputs.

The processor can perform different operations in each clock cycle, governed by the FSM. For example, if the FSM selects R0 as the output of the multiplexer (MUXout) and asserts A_{in} (i.e., A_{in}=1), the contents of register R0 will be loaded into register A on the next positive clock edge. In the next cycle, if the FSM selects R1 as the output of MUXout, deasserts A_{in}, chooses add as the ALUOP, and asserts R_{in}, the value in A (which matches the value in R0) will be summed with the value in R1 and stored in register R (R=R0+R1).

Thus, addition, subtraction, and multiplication of numbers are performed by using the multiplexer to first place the 16-bit number onto MUXout and then loading this number into register A. On the next cycle, a second 16-bit number is placed onto MUXout; the ALU performs the required operation, and the result is loaded into register R by asserting R_{in}. The data in R can then be transferred via the multiplexer to either R0 or R1 by selecting the appropriate signals and asserting either $R0_{in}$ or $R1_{in}$.

Table 2 lists the instructions supported by this processor. The left column shows the name

Instruction		Operation
mv	rX, rY	rX ← rY
	rX, #D	rX ← #D*
add	rX, rY	rX ← rX+rY
	rX, #D	rX ← rX+#D*
sub	rX, rY	rX ← rX–rY
	rX, #D	rX ← rX–#D*
mult	rX, rY	rX ← rX×rY
	rX, #D	rX ← rX×#D*
* (sign extended)		

Table 2: Instructions performed by processor

of an instruction and its operands. The syntax rX ← rY means that the value in the source register rY is loaded into the destination register rX. #D indicates that immediate data is used instead of a value from a register. Immediate data refers to data that is directly part of the instruction rather than data from a register. You have begun to learn about instruction operands in lecture.

Each instruction is encoded using a 16-bit format. When the source operand is a register, the instruction encoding is II0X00000000000Y, where II specifies what instruction is carried out, X specifies the destination register rX, and Y specifies the source register rY. When the source operand is an immediate value #D, the encoding is II1XDDDDDDDDDDDD, where the field DDDDDDDDDDDD represents a 12-bit signed value.

The instruction codes are as follows: II = 00 for the `mv` instruction, 01 for `add`, 10 for `sub`, and 11 for `mult`. The third bit of the instruction encoding is used by the control FSM to determine whether the source data is a register or an immediate value. We refer to this bit as M. Figure 4 provides a summary of the instruction opcode format.

The `mv` instruction (move) copies the content of one register into another using the syntax `mv rX, rY`. It can also be used to initialize a register with immediate data, as in `mv rX, #D`. Since the data D is represented inside the encoded instruction using only 12 bits, the processor must sign-extend the data before loading it into the destination register. This means that for an immediate value represented as $D_{11}D_{10}D_9D_8D_7D_6D_5D_4D_3D_2D_1D_0$, the most significant bit (D_{11}) will be replicated four times to form a 16-bit value: $D_{11}D_{11}D_{11}D_{11}D_{11}D_{10}D_9D_8D_7D_6D_5D_4D_3D_2D_1D_0$. The `add` instruction (`add rX, rY`) produces the sum of rX+rY and stores the result in rX. The `add rX, #D` instruction produces the sum of rX+D, where D is sign-extended to 16 bits, and saves the result in rX. The `sub` and `mult` instructions perform similar operations with the appropriate operands.

The `mv` instruction takes 2 clock cycles to complete; one for moving the instruction data ($\texttt{INSTR}_{in}$) into the instruction register, and another for performing the move operation. `Add`, `sub`, and `mult` take more clock cycles to complete because multiple transfers have to be

```
/* OPCODE format: II M X DDDDDDDDDDDD, where
    *     II = instruction, M = Immediate, X = rX; X = (rX==0) ? r0:r1
    *     If M = 0, DDDDDDDDDDDD = 00000000000Y = rY; Y = (rY==0) r0:r1
    *     If M = 1, DDDDDDDDDDDD = #D is the immediate operand
    *
    *  II M  Instruction   Description
    *  -- -  -----------   -----------
    *  00 0: mv    rX,rY    rX <- rY
    *  00 1: mv    rX,#D    rX <- D (sign extended)
    *  01 0: add   rX,rY    rX <- rX + rY
    *  01 1: add   rX,#D    rX <- rX + D
    *  10 0: sub   rX,rY    rX <- rX - rY
    *  10 1: sub   rX,#D    rX <- rX - D
    *  11 0: mult  rX,rY    rX <- rX * rY
    *  11 1: mult  rX,#D    rX <- rX * D
*/
```

Figure 4: Summary of Instruction Opcode Format

performed across the MUXout bus. The finite state machine in the processor steps through the cycles of each instruction, asserting and deasserting the necessary control signals in successive clock cycles until the instruction is completed. The processor starts executing the instruction when the Run signal is asserted.

Table 3 lists the control signals from Figure 3 that need to be asserted in each time step to implement the instructions in Table 2. Only the signals from Figure 3 that need to be asserted in each time step are listed in Table 3; all other signals are not asserted. The ALUOP signal in step C2 is set to 00 for addition, 01 for subtraction, and 10 for multiplication.

To illustrate, consider the instruction `Add r0, r1` (`r0` $\leftarrow$ `r0 + r1`):

- At cycle C0, `IRin` is set to 1 to accept the instruction data ($\texttt{INSTR}_{in}$) into the instruction register
- At cycle C1, `Select = 01` (for `r0`) and $\texttt{A}_{in}$ `= 1`. This loads the value from register `R0` into register `A`.
- At cycle C2, `Select = 10` (for `r1`), $\texttt{R}_{in}$ `= 1`, and `ALUOP = 00`. This feeds the value from `r1` into the ALU, adds it to the value in `A`, and stores the result in register `R`.
- At cycle C3, `Select = 00` (for `R`), $\texttt{R0}_{in}$ `= 1`, and `done = 1`. This stores the value from register `R` into `R0` and indicates the end of this instruction's execution.

This example assumes that any signal not mentioned is deasserted (i.e., set to zero).

	C0	C1	C2	C3
mv	IRin=1	Set Select based on M and rY Set R0in or R1in based on rX Done=1		
add	IRin=1	Set Select based on rX Ain = 1	Rin = 1 Set Select based on M and rY ALUop= 2'b00	Select = 0 Set R0in or R1in based on rX Done=1
sub	IRin=1	Set Select based on rX Ain = 1	Rin = 1 Set Select based on M and rY ALUop=2'b01	Select = 0 Set R0in or R1in based on rX Done=1
mult	IRin=1	Set Select based on rX Ain = 1	Rin = 1 Set Select based on M and rY ALUop=2'b10	Select = 0 Set R0in or R1in based on rX Done=1

Table 3: Control signals asserted for each instruction's cycle

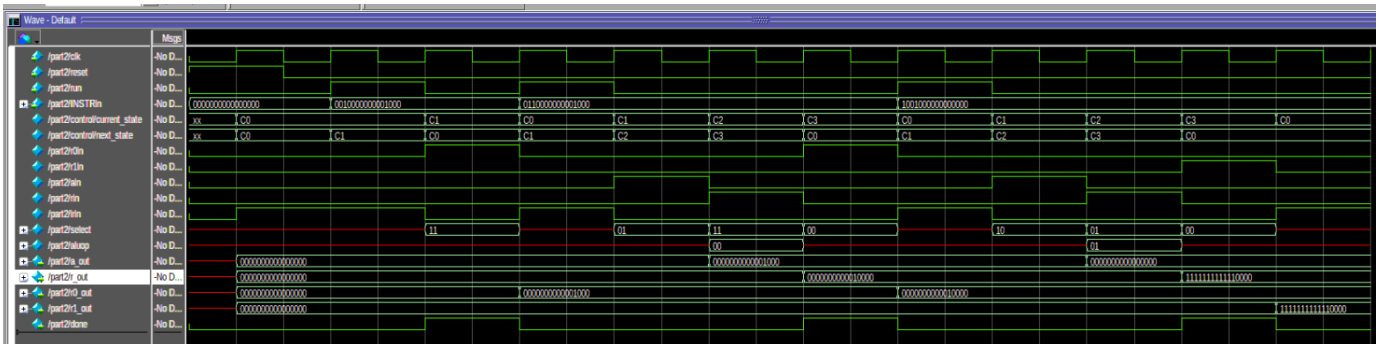

Figure 5: Expected Output from the provided do file

What to do

Pre-Lab work:

1. You are given System Verilog skeleton code for the processor in Fig. 3 on Quercus (`part2_template.sv`). Read the code and make sure you understand what is going on.
2. Create multiple copies of the processor schematic, and trace the path each instruction follows through the datapath. This will help ensure you understand how the processor works.

In Lab:

3. Copy the provided template to a file called `part2.sv`
4. Fill in the missing parts of the control path module using Table 3 as a reference.

5. Fill in the code for the datapath module using the Datapath schematic provided in Fig. 3 as a reference.
6. You are given an initial do file to test your design. The output signals should resemble the waveforms in Figure 5. Review the do file and waveforms, make sure you understand the signal behavior in the figure, and verify that your outputs match those shown using modelsim.
7. Create your own do files for more comprehensive testing.
8. After testing your design, transfer your file as `part2.sv` to the UG machines. Run the lab's autotester to verify that your module ports match the expected declarations. Once you're satisfied with the results, submit your file.

2.3 Optional: Running on FPGA

To run your design on an FPGA, use the mapping shown in Table 4. To display the register outputs on the HEX display, use the least significant 4 bits of each register. This limits the testable values for each register to a range from 0 to F.

`part2` Port Name	Direction	DE1-SoC Pin Name
`clk`	Input	KEY[0]
`reset`	Input	SW[0]
`run`	Input	SW[1]
`INSTRin`	Input	Specify in top module
`Done`	Output	LEDR[0]
`r0_out`	Output	HEX0
`r1_out`	Output	HEX1
`a_out`	Output	HEX2
`r_out`	Output	HEX3

Table 4: Module `part1` mapping to DE1-SoC pin names

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 divison is performed in hardware. Figure 6 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 7.

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 SystemVerilog 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
);
```

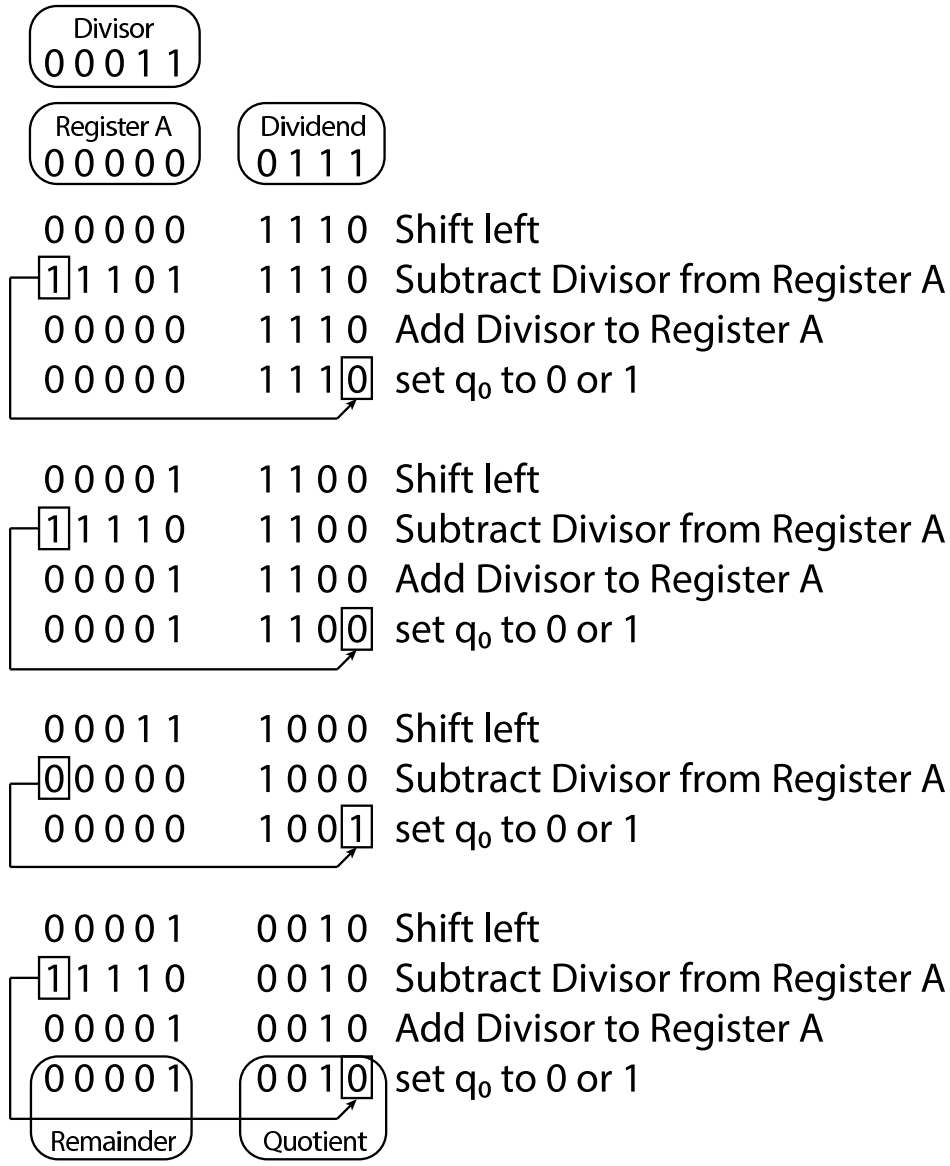

Figure 6: An example showing how the restoring divider works.

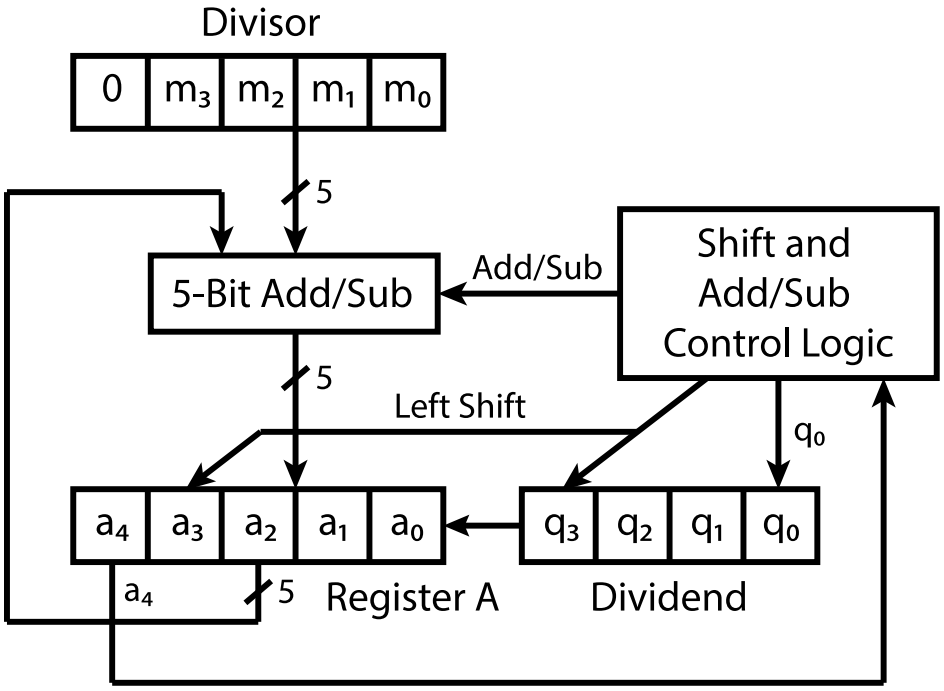

Figure 7: Block diagram of restoring divider.

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 7. 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 6. 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 6 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 5.

`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 5: 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);
```