CSC 258 - Lab 6

Finite State Machines

1 Learning Objectives

The purpose of this lab is to learn how to create Finite State Machines and use them to perform operations on a datapath.

2 Marking Scheme

This lab is worth 6% 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 (bonus): 2 marks (1 for prelab, 1 for in-lab demo)

3 Preparation Before the Lab

You are required to complete Parts I and II of the lab by building and testing your circuits in Logisim (using reasonable tests that you can justify). Part III is optional, but has a prelab component, should you choose to do it. You must submit your prelab preparations (schematics, Logisim design, and simulation outputs) for Parts I to II (and Part III, if you choose to do it).

In-lab Work

You are required to implement and test all of Parts I and II of the lab (and Part III, if you choose to do it). Your demonstration to the teaching assistants should illustrate the rigorous testing procedure you used to verify your design.

4 Part I

Your task for this part is to implement a finite state machine (FSM) that implements a sequence recognizer. This will turn an output signal on if it recognizes two specific sequences of input signals: the sequence 1111 **or** the sequence 1101.

Given an input w and an output z, the output z is set to 1 Whenever $w = 1$ for four consecutive clock pulses, or when the most recent sequence on w was 1101 for the last four clock pulses. In all other cases, $z = 0$. Overlapping sequences are allowed, so that if $w = 1$ for five consecutive clock pulses the output z will be equal to 1 after both the fourth and fifth pulse. Figure 1 illustrates the required relationship between w and z. A state diagram for this FSM is shown in Figure 2.

In the starter Logisim file, the two modules *part1_FSM* and *part1_state_table* provide a starting point for implementing the required state machine:
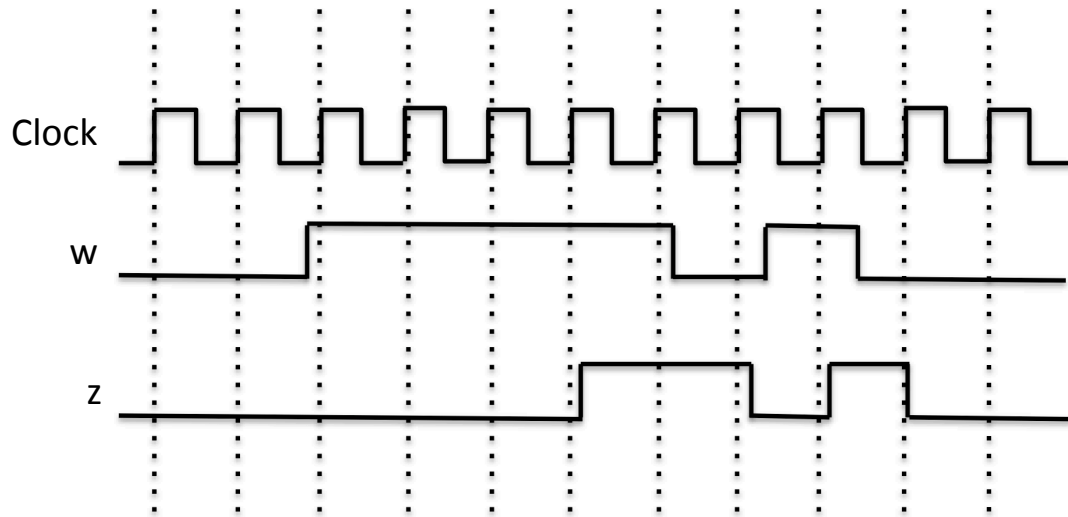
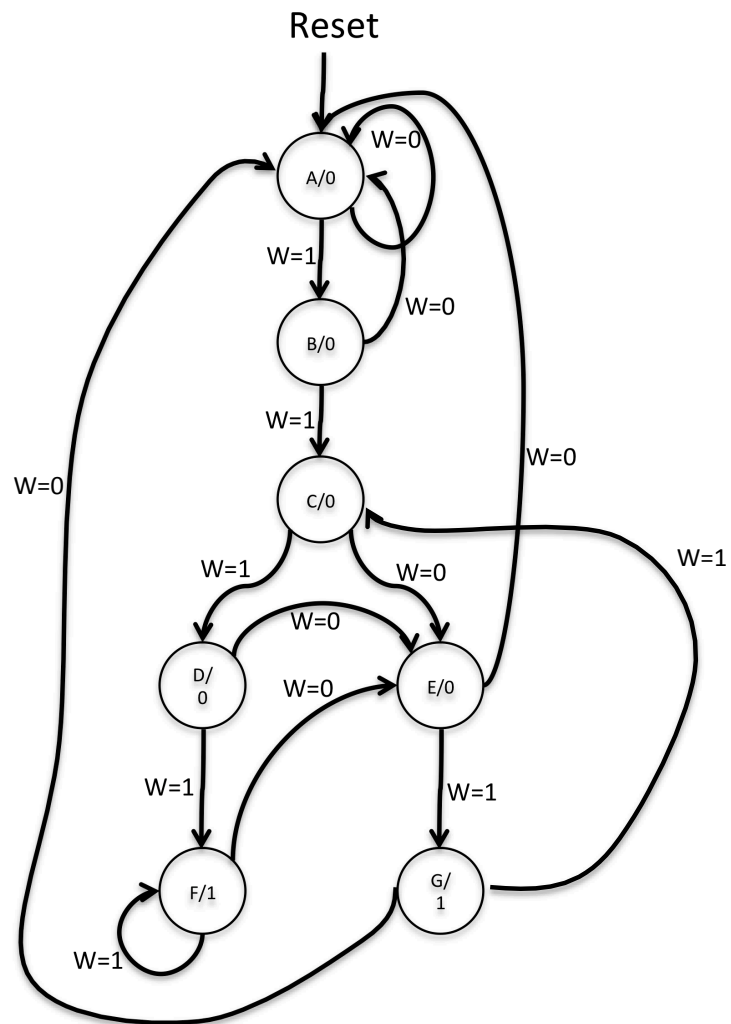Figure 1: Required timing for the output z.

Figure 2: A state diagram for the FSM.

- *part1_FSM* is the high-level FSM circuit that has the flip-flops needed to implement the states and a module for the state logic. A module for the output logic is omitted and must be filled in by you.

- *part1_state_table* is the module for the state logic that you must complete.

Study and understand these modules as they provide a model for how to clearly describe a finite state machine that will both simulate and synthesize properly.

To complete this part of the lab, you must follow the following steps:

1. Begin with the starter circuit provided in `lab6_starter.circ`. Answer the following questions in your prelab: **(PRELAB)**

   - Given the starter circuit, is the `Reset` signal a synchronous or asynchronous reset? *synchronous*

   - Is it active high, or active low signal? *active low.*

   - How should the `Reset` signal feature in the tests that you run on your FSM?

   Hint: if you're not sure of some of the answers to the first two questions, try experimenting with the circuit to confirm the behaviour you suspect.

2. Before modifying the Logisim starter circuit, assign flip-flop values to each of the states in Figure 2 and create a state table that illustrates the state transitions in response to the input signal w.

3. Fill in the rest of the circuit in the *part1_state_table* module to implement the state table you derived in Step 2. This module will implement the *state logic* (the combinational circuit that determines what the new flip-flop values should be, based on the previous flip-flop values and the input w).

4. Implement the output value circuit for z in *part1_FSM*. **(PRELAB)**

5. Outline the test plan for your circuit in your prelab report and why these test cases verify the correctness of your circuit. Use this plan to test your modules with *Poke*(👆) to confirm its expected behaviour. Include screenshots of your simulation output that illustrates key test cases. **(PRELAB)**

Note: There are several ways to implement the state logic and output logic. There are no restrictions on the approach that you use, but some useful suggestions will be offered during the tutorial session.

5 Part II

Processor circuits can be separated into two main components:

1. The *datapath* that connects data storage structures (registers) to processing units (the ALU).

2. The *control path* that operates the datapath signals to determine what data values flow through the datapath and what operations are performed on this data (a finite state machine).

In previous labs, you constructed a simple ALU. In Part I of this lab you constructed a simple *finite state machine* (FSM), which is the most common component used to implement the control path. To complete this part of the lab, you will implement an FSM to control a datapath that performs a common calculation. This is an important step towards building a microprocessor as well as any other computing circuit.

The datapath you will be controlling is provided as part of the starter circuit. A FSM is provided as well, which operates on this datapath to compute $2A^2 + C$. Your task is to create a **different** FSM that controls the datapath provided to perform the following computation:

$$Cx^2 + Bx + A$$

The values of x, A, B and C will be preloaded by the user on the switches before the computation begins.

design FSM, to output centrols signals to operate the datapath to set (

3

Figure 3 shows the block diagram of the datapath you will be using. There are a few things to note about this diagram:

- Reset signals are not shown to reduce clutter, but make sure to include them when building your circuit in Logisim.

- 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 only needs to perform addition and multiplication, but you're welcome to use a variation of the ALU you built in previous labs in order to have more operations available for solving other equations (in case you wish to try some things on your own).

- There are four registers R_x, R_A, R_B and R_C that will be loaded at the start with 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 its value both 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 load new values. They also have an active low asynchronous reset to clear their contents to zero.
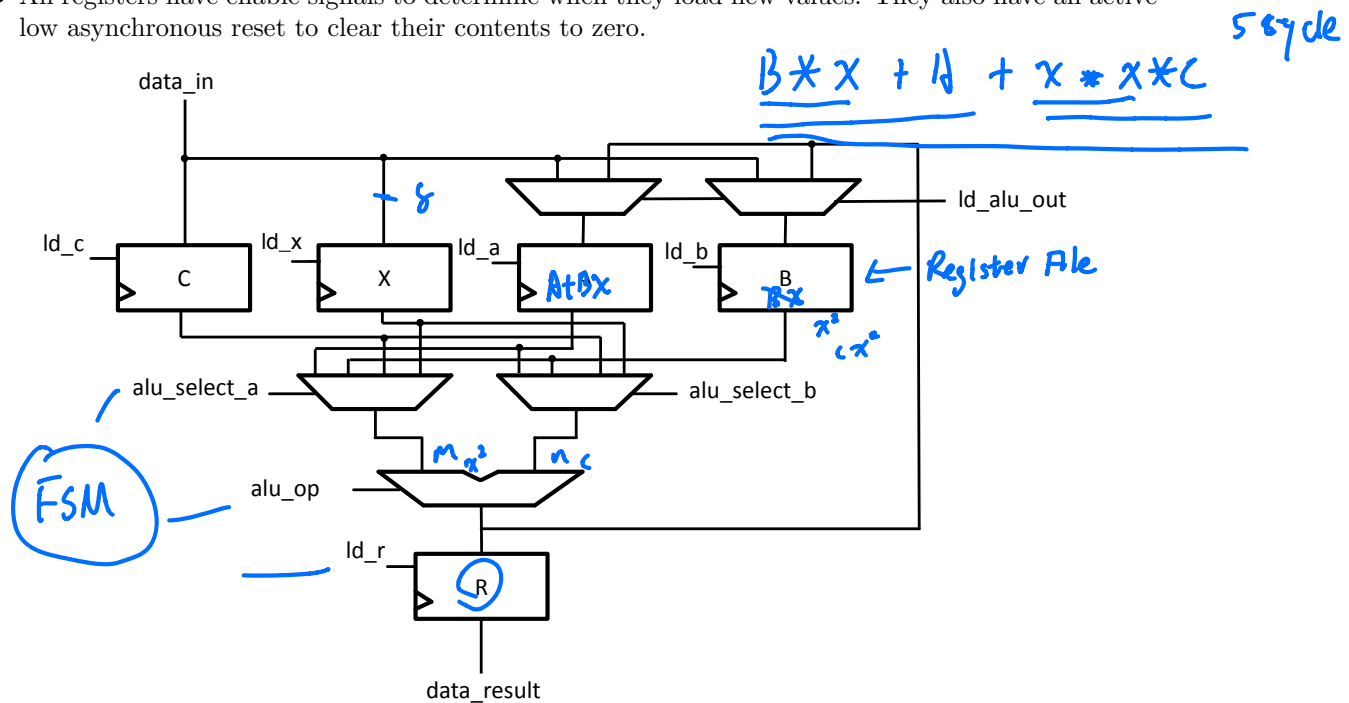

Figure 3: Block diagram of datapath.

The provided circuit should operate in the following manner:

- After an active low asynchronous reset, you will use the $data\_in$ bits to load the following values on the first four clock cycles:

  - On the first clock cycle, load the value for R_A.

  - On the second clock cycle, load the value for R_B.

  - On the third clock cycle, load the value for R_C.

  - On the fourth clock cycle, load the value for R_X.

- Computation will start after all values are loaded.

4

- When computation is finished, the final result will be loaded into R_R.

- This final result should be displayed on LEDs in binary and HEX displays in hexadecimal.

Perform the following steps for this part:

1. Examine the provided starter circuits (all the modules for this part start with `part_2`), which is available on Quercus. This is a major step in this part of the lab. You will not need to build the datapath yourself, but you will need to fully understand the datapath embodied by the starter circuit to be able to make your modifications. **(PRELAB)**

2. Determine a sequence of steps similar to the datapath example shown in lecture to control the datapath to perform the required computation. You should draw a table that shows the contents of the registers and the control signal values for each cycle of your computation. Include this table in your prelab. **(PRELAB)**

3. Draw a state diagram for your controller starting with the register load states provided in the example FSM. Include the state diagram in your prelab. **(PRELAB)**

4. Modify the provided FSM to implement your controller and synthesize it. You should only modify the control module, not the datapath. Submit your modified circuit in the prelab. **(PRELAB)**

5. Test your modules with *Poke*(👆) to verify its correctness. Include a few screenshots that shows the simulation output. **(PRELAB)**

Again, note that there are multiple ways to implement a FSM in Logisim. Explore these and find the approach that works best for you.

It is fine if your implementation does not use all of the modules provided, but that doesn't mean you should have your entire circuit in one giant module! We will start evaluating the elegance and readability of your design in these later labs, so find ways to divide your solution into smaller modules when possible/sensible.

6 Part III (Optional)

Note: Only start working on this part if you already completed other parts. This is an optional part that provides a more challenging exercise for you to further test your knowledge.

Addition, subtraction and multiplication are much easier to build in hardware than division. So division is the most complex operation in hardware. For this part, you will design a 4-bit restoring divider using a finite state machine.

Figure 4 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 5.

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 4) 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*.

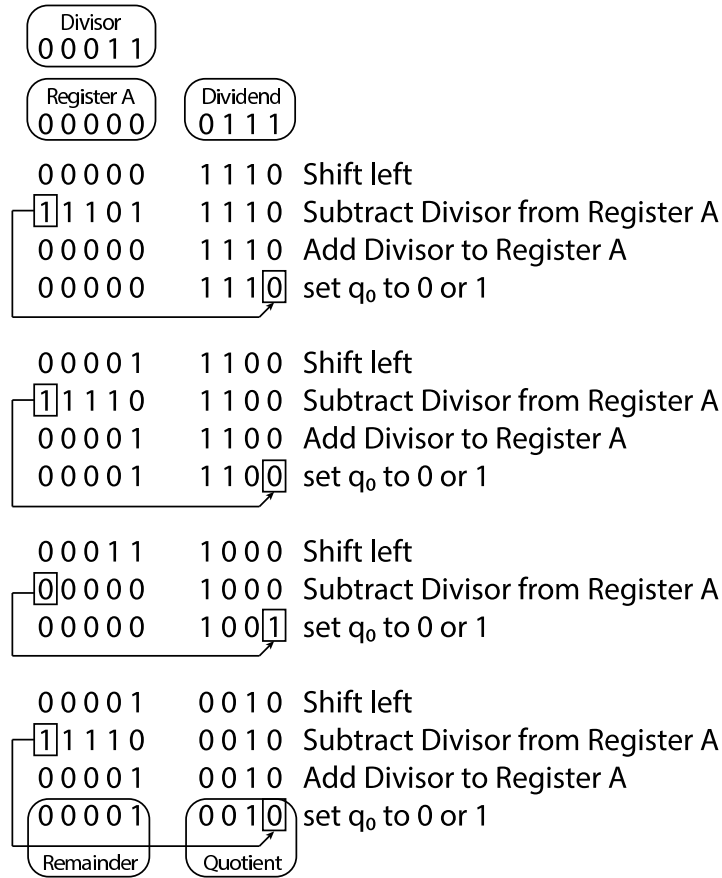Structure your code in the same way as you were shown in Part II and follow these steps for this part:

Figure 4: An example of how a restoring divider works.
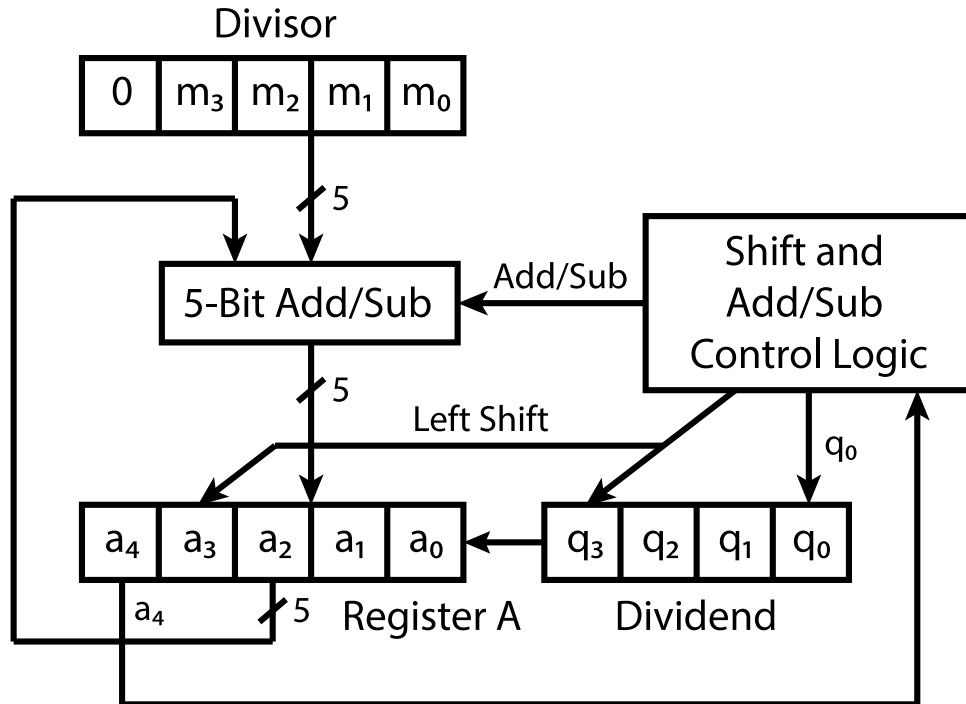
Figure 5: Block diagram of restoring divider.

1. Draw a schematic for the datapath of your circuit. It will be similar to Figure 5. You should show how you will initialize the registers, where the outputs are connected to, and include all the control signals that you require.

2. Draw the state diagram that controls 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. Build your circuit in Logisim.

6. Simulate your circuit using a variety of input settings. **(PRELAB)**

	state	w	next state	output
000	A	0	A	0
	A	1	B	0
001	B	0	A	0
	B	1	C	0
011	C	0	E	0
	C	1	D	0
111	D	0	E	0
	D	1	F	0
110	E	0	A	0
	E	1	G	0
101	F	0	E	1
	F	1	F	1
010	G	0	A	1
	G	1	C	1

cycle		m	n	OP	ldA	ldB	ldC	ldX	ld_Ahort	Sa	Sb	ld-r
1	load RA				1	0	0	0	0	00	00	0
2	$\vdots$ RB				0	1	0	0	0	00	00	0
3	$\vdots$ RC				0	0	1	0	0	00	00	0
4	$\vdots$ RX				0	0	0	1	0	00	00	0
5	$B \times x$ → store in R_B	x	B	1	0	1	0	0	1	00	01	0
6	$A+Bx$ → store in R_A	A	Bx	0	1	0	0	0	1	00	01	0
7	x^2 → store in R_B	x	x	1	0	1	0	0	1	11	11	0
8	$x^2 C$ → store in R_B	C	x^2	1	0	1	0	0	1	10	01	0
9	Cx^2+Bx+A $(A+Bx)(x^2)$ → store in R_R			0	0	0	0	0	0	00	01	1