CSC258 - Lab 3

Splitters, Adders and ALUs

1 Learning Objectives

In this lab you will design (a) multiplexers using the splitter component, (b) a simple ripple-carry adder, and (c) an Arithmetic Logic Unit (ALU), a fundamental component of each processor. You will also gain more practice with hierarchical design in Logisim and with using binary and hexadecimal numbers.

2 Marking Scheme

This lab is also worth 6% of your final grade, but you will be graded out of 8 marks for this lab, as follows.

- Prelab + Test Vectors: 3 marks
- Part I (in-lab): 1 mark
- Part II (in-lab): 1 mark
- Part III (in-lab): 3 marks

3 Preparation Before the Lab

Carefully review the "Preparation Before the Lab" instructions in the Lab 2 handout.

You are required to complete Parts I to III of the lab by building and testing your circuit in Logisim. Include your schematics and circuit in Logisim for Parts I to III in the prelab. You must test your circuit in Logisim using reasonable test vectors written in the format described in Lab 1 and Lab 2 handouts. The term *test vector* simply refers to one combination of inputs that you will use to test your design. Each simulation should consist of multiple test vectors, sufficient to demonstrate that your design functions as intended. Make sure to submit these test vector files as well.

You are also recommended to find information about *Wiring* in logisim_reference.pdf. Wiring components will be useful for this lab.

In-lab Work

You are required to implement and test all of Parts I to III of the lab. You need to show your designs and demonstrate the operation of each part to the teaching assistants, including your test vectors.

4 Part I

For this part of the lab, you will learn how to use *Splitter*( ) to divide a multi-bit value into individual bits. These bits in turn will become the input signals to a 7-to-1 multiplexer that you create.

Generally, multi-bit values are used as a single unit to store integers or other data values. At times though, these multi-bit values are just a grouping of related individual bits. When this is the case, we

often need to split up these bits and send them in different directions. We can accomplish this with the *Splitter* component. The *Splitter* can be found under *Wiring* in components. You can find more information about this component in *Help > User's Guide > Wiring bundles > Splitters*.

In addition to splitting multi-bit values into individual bits, splitters can also be used to concatenate individual bits into a single multi-bit value. You can adjust the multi-bit width, the "fan out" number and the direction of the splitter in its properties.

For this part, the splitter will be used to provide inputs to a 7-to-1 multiplexer that you will create in Logisim. In the components list, find and select *Plexers > Multiplexer* and then add one to the canvas. Clicking on the multiplexer you just created, go to *Properties* at the bottom left of the screen, change *Select Bits* to a value that allows 7 data inputs for your multiplexer.

1. Before implementing this on Logisim, draw a schematic that shows how a multi-bit input would be split up to provide inputs to a 7-to-1 multiplexer. Draw your schematic (how your overall design breaks down into interconnected modules) and label all inputs, outputs and wires between modules. Be prepared to explain your design approach to the TA as part of your preparation. **(PRELAB)**

   (a) Assume that at the highest level, the multi-bit input is coming from the DE1-SOC switches (with the first data bit coming from *SW[0]*) and the output of the multiplexer is displayed on *LEDR0*. This is for labeling purposes only, not because we expect you to upload your design onto the DE1-SOC board.

   (b) How big does the multi-bit input need to be to provide all the inputs to the 7-to-1 multiplexer? Provide your answer in your report. **(PRELAB)**

2. Build your circuit in Logisim, using a 7-to-1 multiplexer and the splitter components. Make sure your implementation reflects your design in the previous step, including the labels. **(PRELAB)**

3. Simulate your circuit with test vectors in *Simulate > Test Vector....* Include a screenshot of the simulation output as part of your prelab report. **(PRELAB)**

5   Part II

Figure 1(a) shows a circuit for a *full adder*, which has the inputs a, b, and c_i, and produces the outputs s and c_o. Parts b and c of the figure show a circuit symbol and truth table for the full adder, which produces the two-bit binary sum $c_o s = a + b + c_i$. Please note that the + operator here means addition and not logic OR. Figure 1(d) shows how four instances of this full adder module can be used to design a circuit that adds two four-bit numbers. This type of circuit is called a *ripple-carry* adder, because of the way that the carry signals are passed from one full adder to the next. Build this circuit in Logisim, as described below. Be sure to use what you learned about hierarchy in Lab 2.
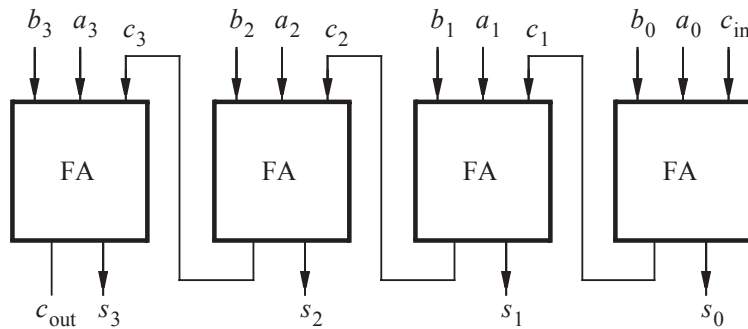
$CoS = b + a + ci$
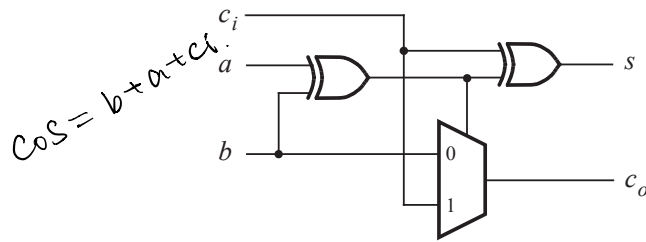
a) Full adder circuit
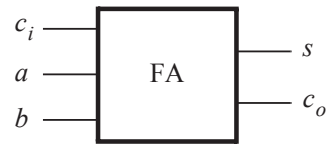
b) Full adder symbol

b	a	c_i	c_o	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
~~1~~	~~0~~	~~1~~	~~1~~	~~0~~
1	1	0	1	0
~~1~~	~~1~~	~~1~~	~~1~~	~~1~~

c) Full adder truth table

d) Four-bit ripple-carry adder circuit

Figure 1: A ripple-carry adder circuit.

Perform the following steps:

1. Draw a schematic showing your code structure with all wires, inputs and outputs labeled. Your schematic should resemble Figure 1(d), though it should also contain module and signal labels, and shows external connections to switches and LEDs (use *SW3-0* for A, *SW7-4* for B and *SW8* for C_{in}. Use *LEDR4-0* for the outputs). Be prepared to explain your approach to the TA as part of your preparation. **(PRELAB)**

2. Build the module for the full adder subcircuit and build a larger module that instantiates the four instances of this full adder. Name the input ports A, B and c_{in}, and the output ports S and c_{out}. Note: You should **NOT** use Logisim's built-in arithmetic addition component in your full-adder. Doing so will earn you 0 marks for this part. **(PRELAB)**

3. Simulate your 4-bit ripple-carry adder with test vectors for intelligently chosen values of A, B and c_{in}. You should include a screenshot of it in the prelab. Note that as circuits get more complicated, you will not be able to simulate or test all possible cases. This means that you can test only a subset. Here *intelligently chosen* means to find particular *corner cases* that exercise key aspects of the circuit. An example would be a pattern that shows that the carry signals are working. Be prepared to explain why your test cases are good enough. **(PRELAB)**

6   Part III

Using Part II from this lab and the HEX decoder from Lab 2 Part III (import your earlier modules from *File > Merge* and make sure all the modules have different names), you will implement a simple Arithmetic Logic Unit (ALU). The ALU generally has two data inputs and can perform multiple operations on these data inputs such as addition, subtraction, logical operations, etc. The operation performed by

the ALU is determined by another multi-bit input (called the *function* input) that specifies the output operation for the ALU.

The easiest way to build this ALU is to:

1. Create modules that implement all of the required operations and then connect the outputs of these modules to the inputs of a multiplexer.

2. Choose the operation that appears on the ALU output by using the ALU *function* inputs (connected to the multiplexer select lines).

3. Display the unsigned inputs A and B on two seven-segments displays.

4. Display the output of the ALU on eight LEDs (binary value) and two more seven-segment displays.

The following table shows the operations that you should implement in the ALU, given the specified *function* value.

function values	logic
0	Make the output equal to A+1, using the adder circuit from Part II of this Lab.
1	A + B using the adder from Part II of this lab
2	A + B using the *Adder* component found in *Arithmetic*
3	Bitwise A XOR B in the lower four bits and A OR B in the upper four bits
4	Output 1 (8'b00000001) if any of the 8 bits in either A or B are high, and 0 (8'b00000000) if all the bits are low (also known as a reduction OR operation)
5	Make both inputs appear at the output, with A in the most significant (left-most) four bits and B in the least significant (right-most) bits.

Note that the ALU takes in two **unsigned** 4-bit inputs, A and B and outputs an unsigned 8-bit value called *ALUout[7:0]*. Note that in some cases, the output will not require the full 8 bits so you will need to do something reasonable with the extra bits, such as making them 0 so that the value is still correct. Adding zeroes in front of any positive number is called *sign-extension*, and does not change the value of the number. In Logisim this is achieved by using *Wiring > Bit Extender* (details can be found `http://www.cburch.com/logisim/docs/2.6.0/en/libs/base/extender.html`).

Perform the following steps to complete the lab:

1. Draw a schematic showing your module structure as a block diagram with all wires, inputs and outputs labeled. Feel free to include multiple schematics if you wish to illustrate multiple levels in your design hierarchy. In particular, clearly highlight the multiplexer for your ALU as well as all inputs to this multiplexer. Be prepared to explain your design choices to the TA. **(PRELAB)**

2. Build the Logisim module for the ALU including all high-level inputs and outputs. **(PRELAB)**

3. Simulate your circuit with test vectors for a variety of input settings, ensuring the all the tests are passing. Include screenshots of your successful test output as part of your prelab. **(PRELAB)**

4. Prepare your design and implementation for your in-lab demo. **(PRELAB)**