Lab 1: Half Adder, Full Adder, 4-bit Incrementer and Adder

Prerequisites: Before beginning this lab, you must:

- Understand how to use the tool flow (See the installation guide and Lab 0)
- Understand Boolean algebraic expressions and the use of logic gates

Equipment: Personal computer with the required software installed.

Files to download:

half_adder_stim.txt
half_adder_top.v
incrementer_top.v
full_adder_stim.txt
full_adder_top.v
four_bit_adder_stim.txt
four_bit_adder_top.v

Objectives: When you have completed this lab, you will be able to:

- Understand the truth tables for the addition of two and three single-bit numbers.
- Build and debug a simulation of a circuit that will perform the half-adder operation.
- Build and debug a simulation of a circuit that will perform the 4-bit increment operation.
- Build and debug a simulation of a circuit that will perform the full adder operation.
- Build and debug a simulation of a circuit that will perform a 4-bit adder operation.

Introduction

In this lab you will build two circuits from primitive logic gates that will provide part of the functionality of a complete microprocessor. These circuits will allow you to add either two 1-bit numbers (i.e., the half adder function) or three 1-bit numbers (i.e., the full adder function). You will then use these library components to build a 4-bit incrementer and a 4-bit adder. The 4-bit incrementer and the 4-bit adder will be used in future labs as subcircuits. Combining subcircuits to create more complex circuits is a powerful strategy that will allow you to create circuits whose modularized diagrams can easily be interpreted.

Warning: Use the signal and circuit names provided! Verilog does not allow names to start with a number or names that have dashes!

Task 1-1: Build and Test the 1-Bit Half Adder

Create a folder named Lab1. It will be important to save all the circuits you design during this lab in this folder.

In lecture, you learned how to build a half adder. Now, use Digital to build and test the 1-bit half adder circuit as shown in Figure 1. (Do NOT use the adder primitives available in Digital! You must build your circuits as specified here.)

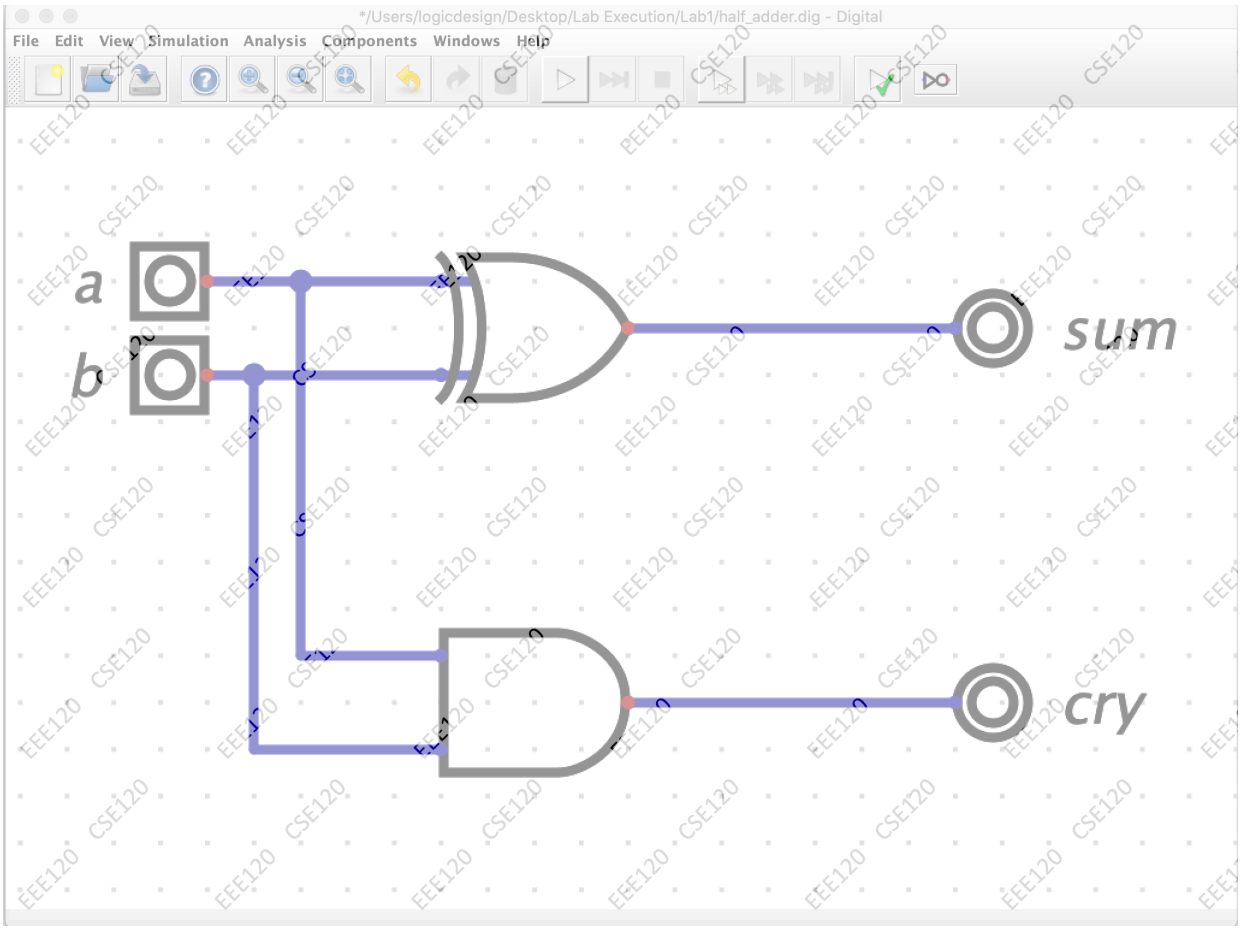

Figure 1: Circuit diagram of a 1-bit half adder.

When done building your schematic, save it with the name half_adder in the Lab1 folder. Digital will automatically append the .dig extension.

Click on the Simulation menu and select Start of Simulation. A shortcut is to click on the triangle icon to the right of the trash icon. (If you hover over the icons below the menus at the top of the screen, text pops up to tell you what pressing that icon will do.) Press the one that says, “Starts the simulation of the circuit”.

When the simulation starts, the wires will change from blue to green. Dark green means the wire is at logic 0. Light green means it is at logic one. (If you are color blind or would like to change the color scheme, select Settings under the Edit menu. There is a dropdown menu for Color-Scheme. Select the scheme you would like or create a custom scheme.)

Click on the inputs to change them. By watching the colors, you can see if the circuit is working as expected. When you are satisfied that the circuit is working, move on to the next task. End the simulation by clicking on the square icon two spots to the right of the triangle icon used to start the simulation or select Simulation->Stop Simulation.

The truth table of a 1-bit half adder is shown in Table 1. We now need to test this circuit. To do that, we need to generate a test waveform file that toggles the input pins through every (**a**, **b**) combination and compare the **sum** and **cry** outputs observed with those prescribed by the truth table of Figure 1. If the outputs your circuit generates agree with the outputs specified by the truth table, then your circuit is working correctly. If they do not agree, you have a problem with your circuit and you need to check that the connections have been made correctly.

a	b	cry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Table 1: Truth table of a 1-bit half adder.

Once you are satisfied that the circuit is working correctly, export your design to Verilog with the name half_adder.v by selecting File->Export->Export to Verilog. Make sure it is being saved in the Lab1 folder! The half_adder_top.v file will instantiate your design, apply the test stimulus found in half_adder_stim.txt, and check the results against the expected responses which are also in half_adder_stim.txt. Table 2 shows how the bits from half_adder_stim.txt are used.

Table 2: Bit definitions for half_adder_stim.txt.

Bit #	7:6	5	4	3:2	1	0
Meaning	unused	cry	sum	unused	a	b

To run the simulation, use the following commands:

iverilog -o half_adder.exe half_adder.v half_adder_top.v

On Mac: ./half_adder.exe
On Windows: vvp half_adder.exe

Use GTKWave to view the results produced in half_adder_waves.vcd. Are they what you expect? Did the simulation generate errors? If the answer to either question is no, then go back and make sure you entered the components correctly.

Reminder: On the Mac, launch the GTKWave application. Under File, select “Open New Tab”. Alternatively, on Mac press CMD-T or on Windows CTRL-T. Select the *waves.vcd, where *waves.vcd is the wave file you want to open. You may have to navigate to the Lab0 folder.

On Windows, type “gtkwave *_waves.vcd”, where *_waves is the file you want to open.

Once you are convinced that your circuit is working properly, take a screengrab of your schematic and the timing diagram and copy it to your lab template. Be sure to include the

entire window as shown in Figure 1 when you take your screenshot. Use View->Fit to window to maximize the size of your circuit in the window. Also, comment on any issues that you encountered.

Task 1-2: Build and Test a 4-Bit Increment Circuit

An arithmetic operation that our microprocessor will use in its program-counter circuitry is the increment operation. The increment circuit we will need must add 1 to a 4-bit number. For example, 0001 + 1 => 0010. To accomplish this task, we can use four 1-bit half adders to implement a 4-bit increment circuit. A 4-bit increment circuit will accept a 4-bit binary number as its operand and be controlled by the increment control, **inc**. Your circuit will produce a 4-bit binary number and a carry as output. If the input control, **inc**, is low, the output number will be the same as the input number. If the input control is high, the output will be one more than the input.

The task is easier to accomplish if you use multiple copies of the 1-bit half adder circuit you built in Task 1-1. Select File->New and you should get a brand new, empty schematic sheet. Now select File->Save As and type in the name incrementer as shown in Figure 2. Do not skip this save as it compensates for a feature of Digital where the subcircuits like half_adder are not visible unless they are in the folder, or a subfolder, of where the current design is saved.

NOTE FOR MAC USERS: If the half_adder.dig is still not visible, exit Digital and, in Terminal, go to the Lab1 directory and start Digital there using the command:

java -jar path_to_digital/Digital.jar &

where path_to_digital is the path to where you have installed Digital. For example, if the Digital folder is on your Desktop, replace path_to_digital with ~/Desktop/Digital/Digital.jar.

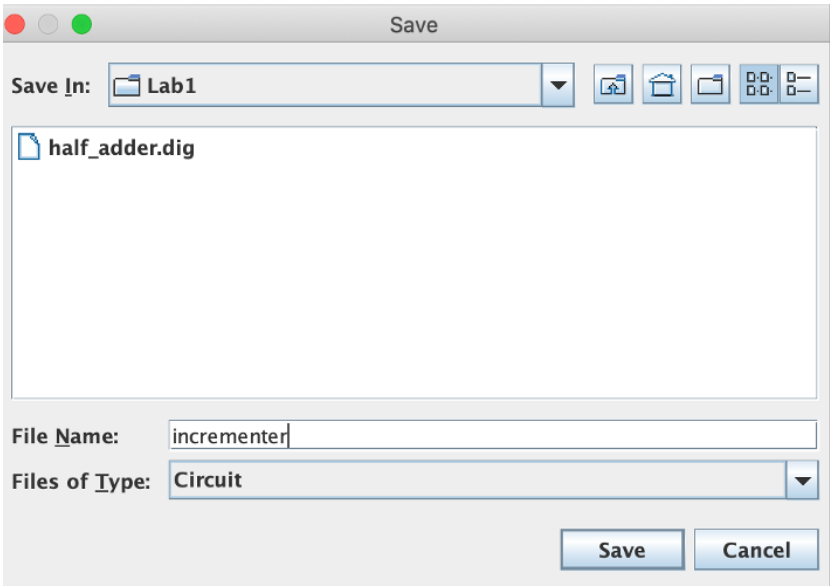

Figure 2. Saving the new incrementer design.

Now build the 4-bit incrementer circuit shown in Figure 3. You will find the half_adder circuit under Components->Custom. The new component needed for this design is the Splitter/Merger, which is found under Components->Wires. We will need to modify the Splitter/Merger so that it works with our design and details on how are below.

Note the extra space that separates the symbols for the input and the output from the splitters. When we simulate in Digital, that extra space will allow the value on the bus to be seen.

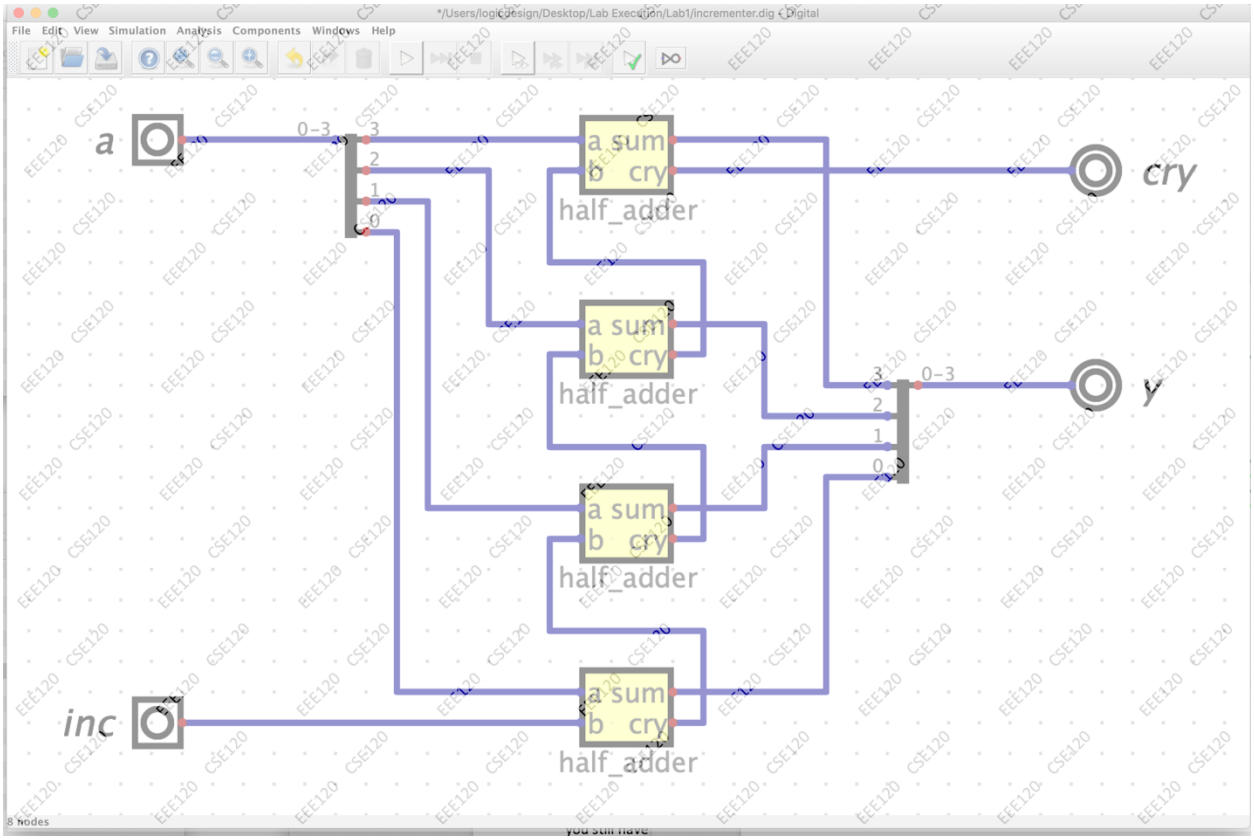

Figure 3. The incrementer design.

Once you have all the components in place, we need to set up our buses. A bus is a bundle of wires. CTRL click on Mac, or right click Windows, on the **a** input to open its properties. Name it **a** and set it to have 4 bits as shown in Figure 4. Then do the same thing for the output, **y**. The **inc** input and **cry** output should also be named, but leave them at just 1 bit.

NOTE: We will no longer detail how to open the "properties" window for a component. You'll need to remember that it is CTRL click on Mac and right click on Windows.

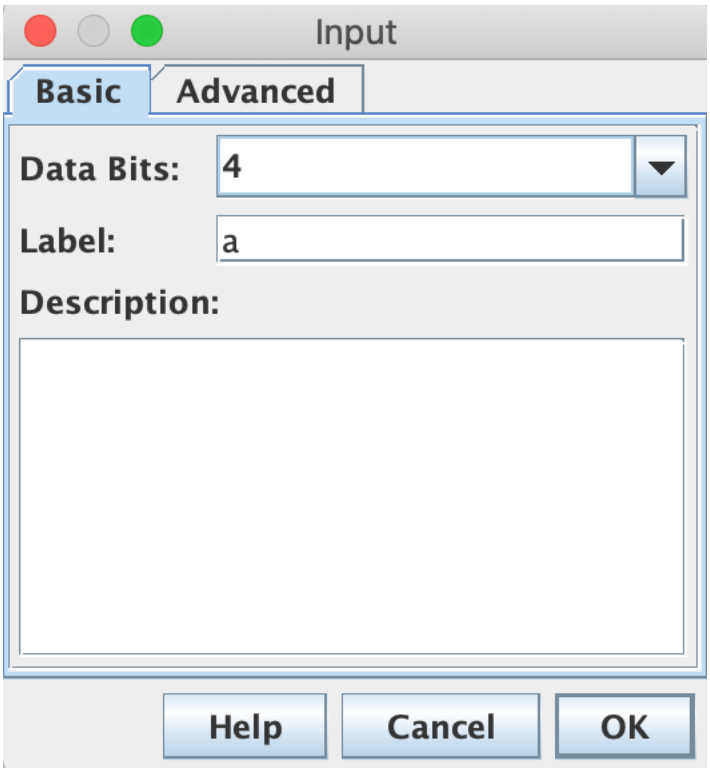

Figure 4. Setting an input to be 4 bits wide.

Now we need to modify the Splitter/Merger components. Open the properties window for the Splitter/Merger connected to **a** and set it up as shown in Figure 5. The "Input Splitting" field indicates that we'll just keep all the wires together. The "Output splitting" field shows which bits, and in what order, we want them coming out. So, bit 3 will be on top and bit 0 on the bottom. Since bit 3 is the most significant bit, this is how we want it arranged. Using the splitter allows us to have a single input for **a** rather than having to bring in 4 separate signals.

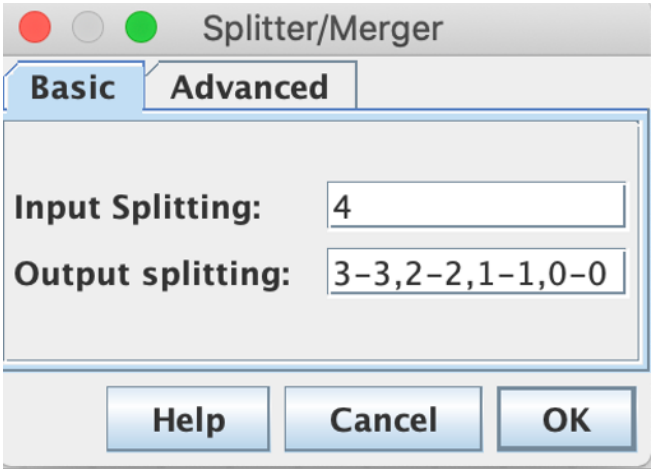

Figure 5. Set up for splitting a 4-bit bus into 4 individual wires.

Next, we need to modify the Splitter/Merger connected to **y**. Open its properties window and configure it as shown in Figure 6. This will gather the wires together so we can have a single output signal. Finally, connect the wires as shown in the figure.

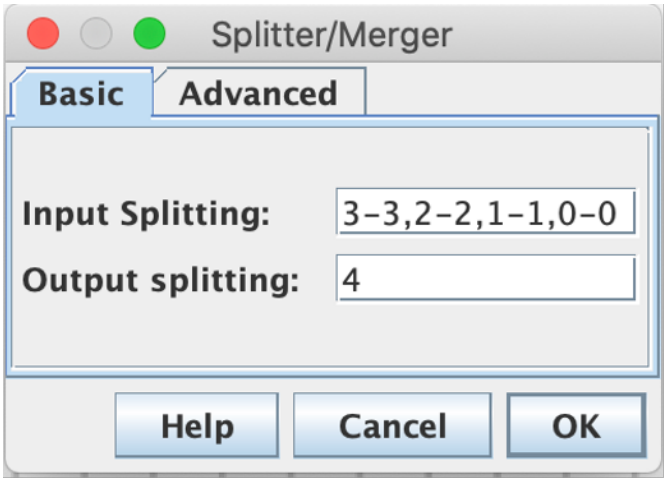

Figure 6. Set up for combining four wires into a bus.

Before proceeding to simulation, check out the functionality in Digital. Start the simulation. You'll notice that the buses are colored blue and that the bus's value appears in hexadecimal. Click on the input **a** and the circuit will appear like that in Figure 7. The window that popped up for input **a** has a field which shows the current value on the bus. In this case, I've set it to 5. (The 0x indicates that this is a hexadecimal number.) You can change the base using the dropdown menu on the left but I suggest you leave it as hexadecimal.) You can also click on the small boxes at the bottom to change individual bits and then hit Apply. Or, you can use the up and down arrows to change the value. When using the arrows, the changes happen in real time. In the figure, the value of **inc** is still at its default value of 0. Click on the arrows for **a** to see that the output **y** is always the same value as **a**. Then, click on **inc** and see that **y** is always 1 more than **a**. And, when **a** is equal to 15 (or 0xF in hexadecimal), **y** returns to 0 and **cry** is 1.

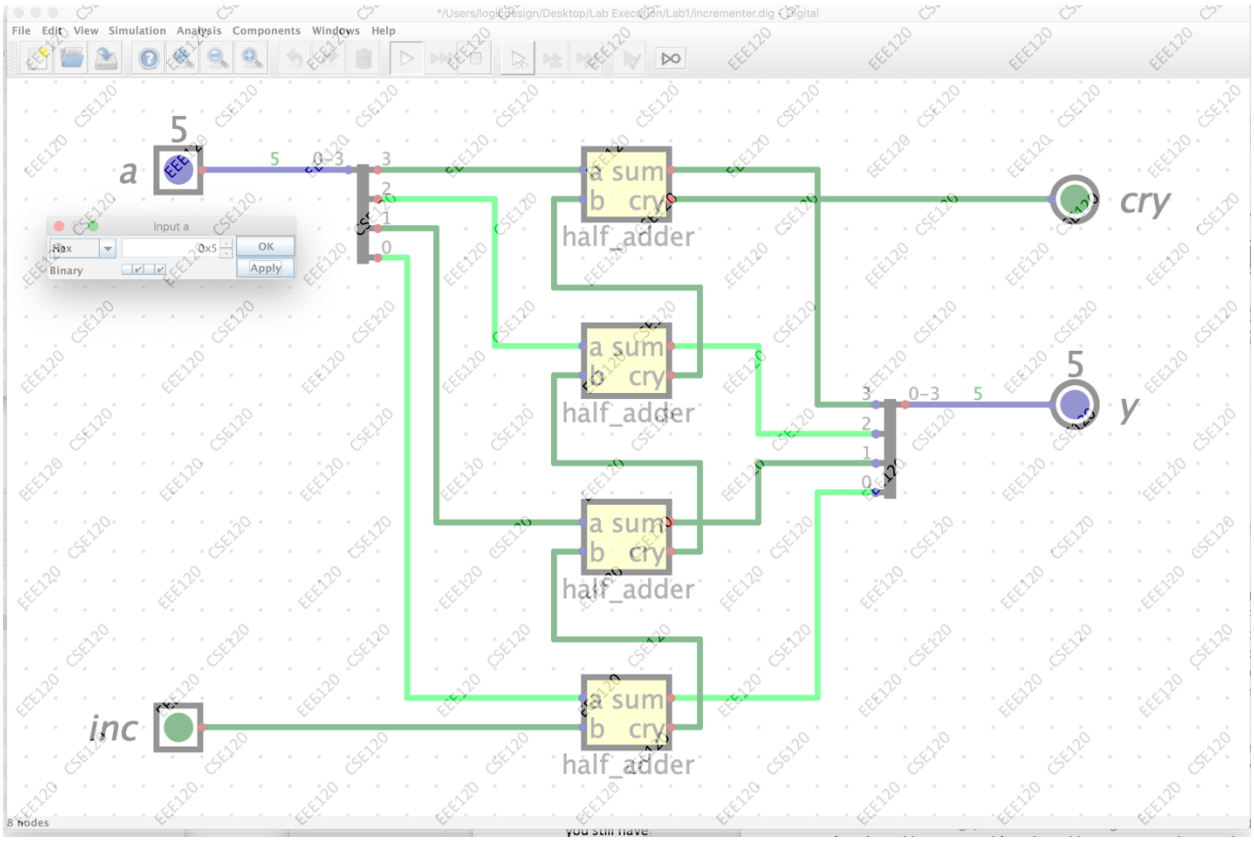

Figure 7. Simulating in Digital.

When the circuit is working properly, take a screenshot of the schematic. (Be sure to stop the simulation first!) Then paste the screenshot into your template. Be sure to include the entire window.

Make sure you save the design, and then choose File->Export->Export to Verilog, leaving the name incrementer.v.

Now, let's simulate the design in iVerilog. You've been given one file to use with the incrementer: incrementer_top.v. Make sure that file is in your Lab1 folder. Execute these commands:

iverilog -o incrementer.exe incrementer.v incrementer_top.v
On Mac: ./incrementer.exe
On Windows: vvp incrementer.exe

Then open incrementer_waves.vcd in GTKWave and click next to incrementer_top to show the incrementer and select incrementer. Double click on the **a**, **inc**, **y**, and **cry** signals. The waves should show the circuit incrementing when **inc**=1 and passing the **a** through to **y** unchanged when **inc**=0.

You'll note that there are many more wires present that are named s#, where # is some number. These are all the wires in the design connecting various components. You can ignore these. Just look at the inputs and outputs of your circuit.

Note that you can click on the symbol to the left of incrementer and you'll see the four half adder circuits. These are named half_adder_i0 through half_adder_i3. The addition of the suffix allows each to be handled separately. If you click on one of them, you'll see its inputs and outputs. This might be helpful if your circuit isn't working properly. Seeing signals at a lower level can be very helpful when debugging.

Take a screen shot of the waveforms making sure to include the entire window. Paste the screen shot into your template.

Task 1-3: Build and Test a 1-bit Full Adder

Build the 1-bit full adder shown in Figure 8. To create a 3-input XOR gate, place the XOR gate and then open its properties. This will bring up the customization window as shown in Figure 9. Set the number of inputs to 3 by typing 3 in the Number of Inputs field or by using the pulldown menu. You can do the same thing to the OR gate so that it has 3 inputs.

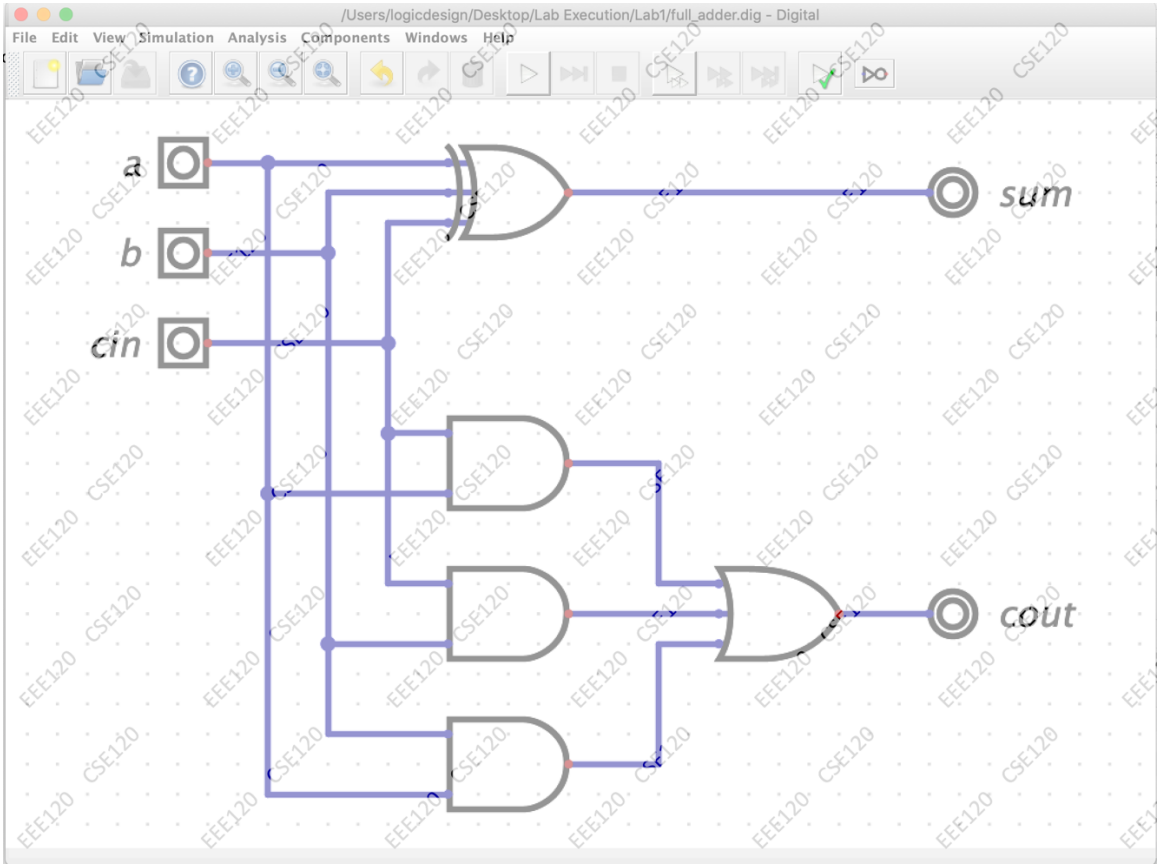

Figure 8. The full adder.

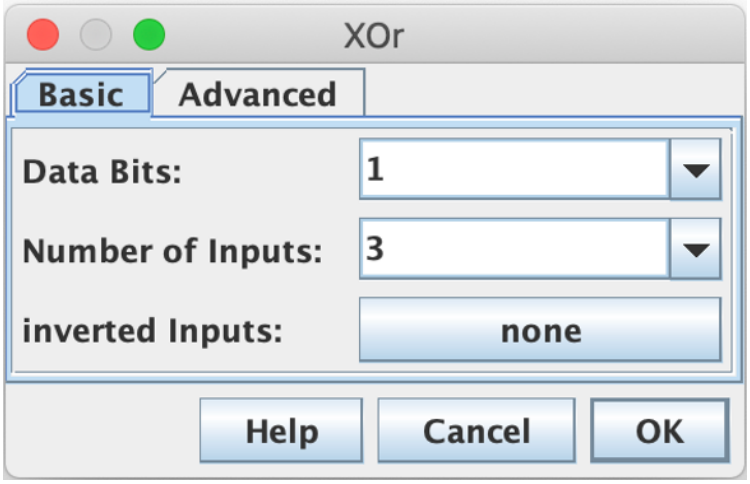

Figure 9. Changing the number of inputs to a gate.

Simulate the circuit in Digital to make sure it works. Once you are satisfied it is working properly, save the circuit as full_adder and export it to Verilog.

To simulate the full adder in Verilog, execute the following command using the files you were provided: full_adder_top.v and full_adder_stim.txt. Make sure those files are in your Lab1 folder. The full_adder_top.v file will instantiate your design, apply the test stimulus found in full_adder_stim.txt, and check the results against the expected responses which are also found in full_adder_stim.txt. Table 3 shows the bit definitions used in full_adder_stim.txt.

Table 3: Bit definitions for full_adder_stim.txt.

Bit #	7:6	5	4	3	2	1	0
Meaning	unused	cout	sum	unused	cin	a	b

iverilog -o full_adder.exe full_adder.v full_adder_top.v
On Mac: ./full_adder.exe
On Windows: vvp full_adder.exe

Next, open the waves file full_adder_waves.vcd using GTKWave. If everything is working properly, take screen shots of the design in Digital and the waves and paste them into your template. Be sure to include the entire window!

Task 1-4: Build and Test a 4-Bit Full Adder

Follow a similar approach to that taken in Task 1-2 to build a 4-bit version of the full adder. Click on File->New and then immediately do File->Save as to save the empty design as four_bit_adder. Then build the circuit as shown in Figure. 10. Set up the Splitter/Merger components as you did for the incrementer.

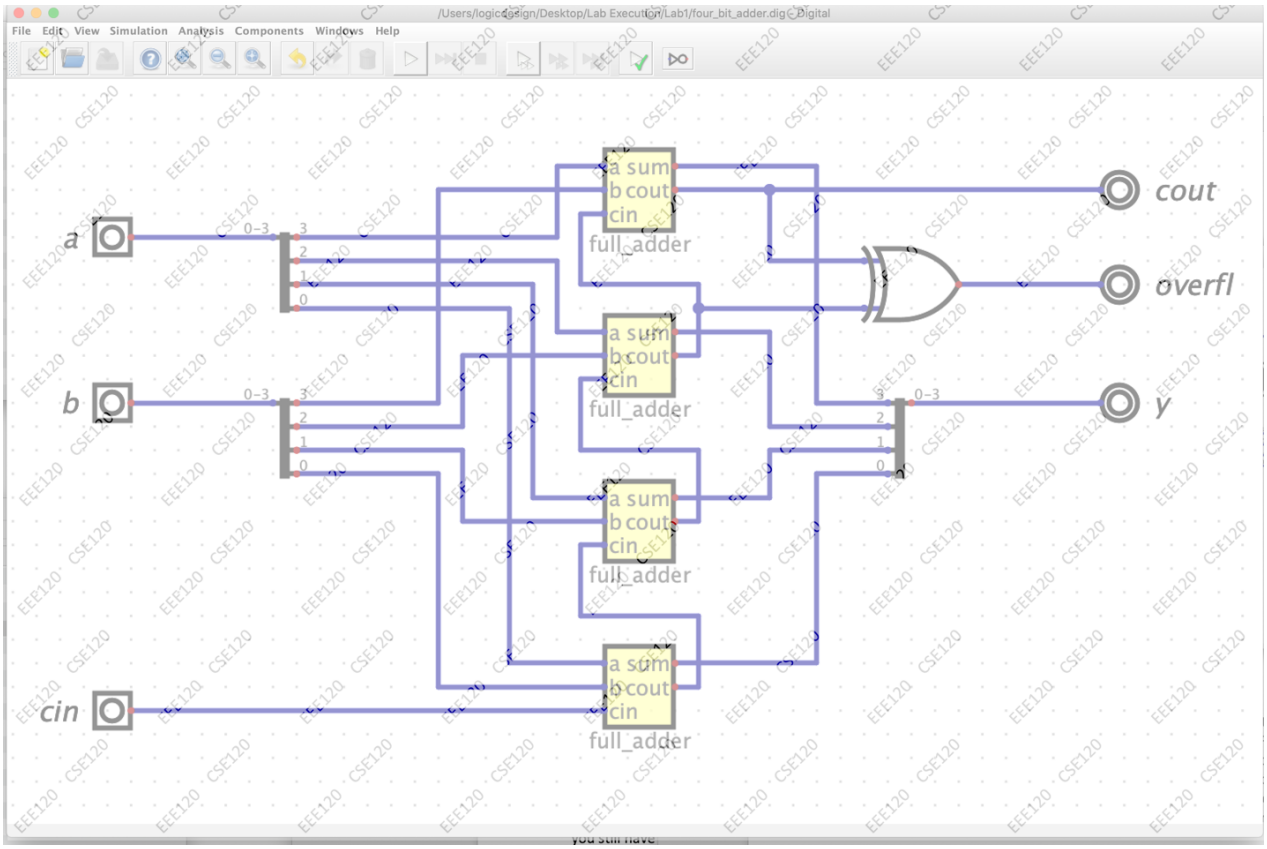

Figure 10. The 4-bit full adder.

Based on what you've learned in class, make sure you understand the difference between **cout**, which will indicate overflow when doing unsigned addition, and **overfl**, which will indicate overflow when doing signed, two's complement addition. When doing unsigned addition, **a** and **b** can represent the numbers 0-15. When doing signed, two's complement addition, **a** and **b** can represent the numbers from -8 to +7. Note that the hardware doesn't care what kind of numbers it is adding – it is up to the user to know whether the numbers are signed or unsigned.

Test out some combinations in the simulator inside Digital. For the adder, as there are 9 inputs (4 bits in **a**, 4 in **b**, and 1 in **cin**), there are 512 possible combinations. Far too many to try in Digital. Once you've tried a few combinations and are satisfied it works, save the design and then export it to Verilog using the filename four_bit_adder.v.

To simulate the design, you'll need to download these files: four_bit_adder_top.v and four_bit_adder_stim.txt. Make sure they are in the Lab1 folder. The four_bit_adder _top.v will instantiate your design, apply the test stimulus found in four_bit_adder _stim.txt, and check the results against the expected responses which are also in four_bit_adder _stim.txt.

The four_bit_adder_stim.txt file is read by four_bit_adder_top.v. Each line that starts with // is a comment and is ignored by the simulator. The rows with numbers, starting with the line

0_0_0_0_0, are where the action is. Let's unpack what each row means so you can implement your tests.

First, each of these digits is a hexadecimal number representing 4 bits. Therefore, those five zeros actually represent 20 bits! The underscores are ignored and are there to make reading the numbers easier. Table 3 shows how each bit is used during the simulation. In order to make the alignment of the various signals easy to see, some of the bits are unused. The heavier vertical lines separate the 4-bit hex digits.

Table 3: Bit definitions for four_bit_adder_stim.txt.

Bit #	19:18	17	16	15:12	11:9	8	7:4	3:0
Meaning	unused	exp_overfl	exp_cout	exp_y	unused	cin	a	b

Unused means that the bits aren't used in the simulation. It is recommended that those bits remain 0.

a, **b**, and **cin** are the inputs to your circuit. Remember that **a** and **b** are 4-bit values.

exp_overfl, **exp_cout**, and **exp_y** are the values you would expect your circuit to produce if it is working properly. The four_bit_adder_stim.v file will automatically check the expected values you have provided against the outputs your circuit produces. An error message will be printed if there is a mismatch. The index of the mismatch tells you which of the 16 tests didn't match. Of course, if there's a mismatch, you'll have to figure out if your circuit isn't working properly or if you have incorrect expected values!

Your task is to modify four_bit_adder_stim.txt with up to 16 tests to verify your design. (The version of four_bit_adder_stim.txt you received has the same 4 patterns repeated 4 times. You'll need to modify these to create your tests.) You will be graded on whether or not functionality is sufficiently verified. You'll need to show the contents of your version of four_bit_adder_stim.txt in the table in your template along with a description of what each test verified. Note that the file **must** have 16 tests, even if you leave some of them unchanged.

Examples: To add 4+4 with carry in = 1, you would replace one of the tests with 2_9_1_4_4 since 4+4+1 is overflow for signed addends but is just fine for unsigned. On the other hand, to do F+F with carry in = 0, you would replace a test with 1_E_0_F_F since F+F is overflow for unsigned numbers but is the same as -1+(-1)=-2 for signed.

Before modifying four_bit_adder_stim.txt, run a quick simulation to verify things are working:

iverilog -o four_bit_adder.exe four_bit_adder.v four_bit_adder_top.v
On Mac: ./four_bit_adder.exe
On Windows: vvp four_bit_adder.exe

Then open four_bit_adder_waves.vcd in GTKWave. It should look like Figure 11. You can ignore the signals that are not your inputs and outputs. You can also click on the symbol next to four_bit_adder to see the signals in individual full adders. You'll notice these waves are not very extensive and don't test very much. It will be up to you to find up to 16 different input combinations that will verify your design. Think about what needs to be tested for proper operation. Then, using the information below, modify the four_bit_adder_stim.txt file to create your tests.

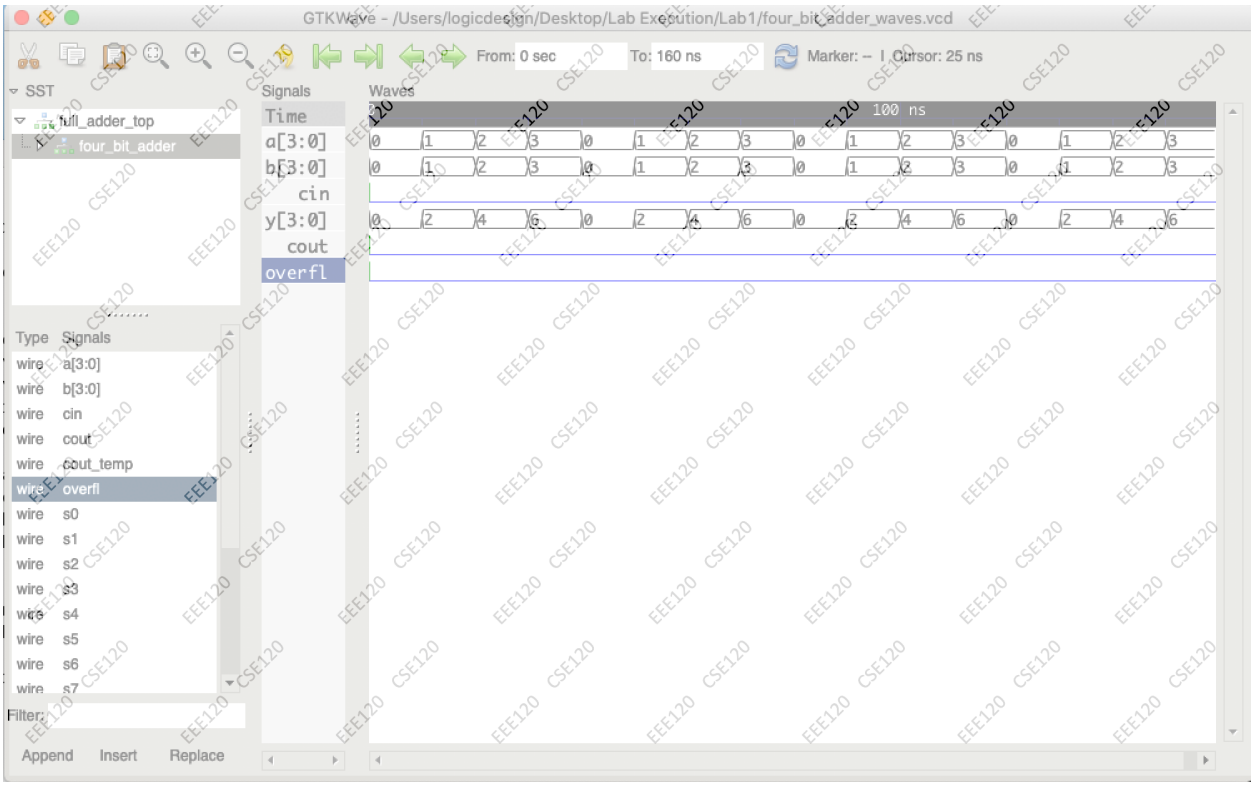

Figure 11. The original stimulus supplied for the 4-bit full adder.

When you are satisfied that your circuit is working and properly tested, take screen shots of your circuit and the wave window. Be sure to include the entire window. Paste these screenshots into your template.

Task 1-5: Create a video and submit your report.

Create a video showing your schematic in Digital, and your waveforms and explain how your design works. Be sure to show yourself in the video and show your screen. Place a link to the video in your template. Be sure to include any required password or to set permissions so that everybody can see your video.

Make sure all of your files are in the Lab1 directory. Create a zip file of the Lab1 directory. Turn in the zip file and your completed template. (Double check that you turned in the completed

template and NOT the blank one you downloaded. Unfortunately, turning in the wrong template file is a common mistake.)

Turn in your template as a PDF. If you turn in any other format, it might not be graded!

Congratulations! You've completed Lab 1! You will be using both the 4-bit incrementer and the 4-bit adder in future labs so be sure to hang onto them.