Lab 2: Multiplexers, Decoders, and the Arithmetic Logic Unit (ALU)

Prerequisites: Before beginning this lab, you must:

- Understand how to use the tool flow (See the installation guide and Lab 0)
- Understand the operation of mulitplexers and decoders
- Understand 1's complement and 2's complement
- Have the components available from Lab 1

Equipment: Personal computer with the required software installed.

Files to copy from Lab 1: (all the .dig files)

half_adder.dig
incrementer.dig
full_adder.dig
four_bit_adder.dig

Files to download:

four_bit_mux_top.v
four_bit_mux_stim.txt
alu_top.v
alu_add.v

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

- Build, simulate, and debug a 2-to-1 multiplexer circuit.
- Build, simulate, and debug a 4-bit, 2-to-1 multiplexer circuit.
- Use seven-segment displays.
- Build, test, and debug an elementary arithmetic and logic unit (ALU).
- Describe the arithmetic and logical operations of which the ALU is capable.
- Describe the input control line values that correspond to each operation of the ALU.
- Generate tests to verify the operation of the ALU in simulation.

Introduction

In this lab you will continue constructing modules that will be used in assembling the microprocessor. Our concern in this lab is with circuits that control the flow of data through our system. You will use the data-flow-control circuit you create in this lab, a 4-bit 2-to-1 multiplexer, to make the microprocessor capable of routing data to appropriate locations. You will also use seven-segment displays. This will enable you to display numeric values in an easy-to-read format, which is much nicer than looking at binary values.

In Lab 1 you built combinational logic circuits of increasing complexity. In this laboratory we will build upon these modules and combine them to form a more complex combinational logic circuit: the arithmetic logic unit (ALU). The ALU in a microprocessor is the unit that performs all

of the arithmetic operations (such as add, subtract, negate, etc.) and all of the logical operations (such as 1's complement, AND, OR, etc.) Because this is an introductory class, we will limit the number of operations our ALU will perform so that we can limit the complexity of the design. Conceptually, however, your ALU will be no different from the ALU in the personal computer you use for performing these digital-logic simulations.

Our ALU, like all ALU's, will be a combinational logic circuit. It will have two data input ports (each of which will accept 4-bit binary data values[1]) and it will have a carry input. These data values are known as operands. Our ALU will be capable of operating on either one operand (e.g., performing a 1's complement operation), or two operands (e.g., performing the sum, A+B) and will produce a 4-bit result (plus a possible carry). The ALU will be controlled by a set of input control signals. These input control signals will determine which operation the ALU will perform. It will be left as an exercise at the end of this lab to create a table that lists the operations the ALU performs for each set of input control signals.

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

Create a folder named Lab2. Into that folder, copy the files listed above from Lab1. Be sure to only copy the required files. While it is tempting to do Lab 2 in the same directory where you did Lab 1, it is advisable to start in a fresh directory in case something goes wrong. That way, you still have the pristine Lab 1 results to copy again. In addition, this means that the Lab 2 folder will only have the files necessary for Lab 2.

Once you've copied the files from your Lab1 folder, download the files provided for Lab 2 and place them in the Lab2 folder. Now, you're ready to start!

NOTE: You are required to design the circuits as presented in this document. Even though Digital supplies many of the functions we will design, you are not to use them. For example, we will design our multiplexer so you are NOT to use the multiplexer available in Digital.

Task 2-1: Build and Test a 1-bit 2-to-1 Multiplexer

When we discuss the architecture of the microprocessor in Labs 3 and 4, we will find that hardware is needed that allows the executing program to select the route along which data flows. The component that we will need to perform this function is the **multiplexer**, or **mux**. A mux is a device that can be controlled to route one of its input signals to its sole output. Therefore, a mux is characterized by its number of signal inputs. In our case, we have two inputs and one output, thus we are dealing with a 2-to-1 or 2:1 mux. The schematic diagram for the 1-bit 2-to-1 mux you will build is shown in Figure 1. The control/select input, **s**, indicates that the output is identical to the **a** input when the select signal is low (0) and identical to the **b** input when the select signal is high (1). Build the multiplexer in Digital as shown in Figure 1.

[1] We choose 4-bit (rather than 32-bit) operands because the 4-bit circuitry is much less tedious to construct and debug.

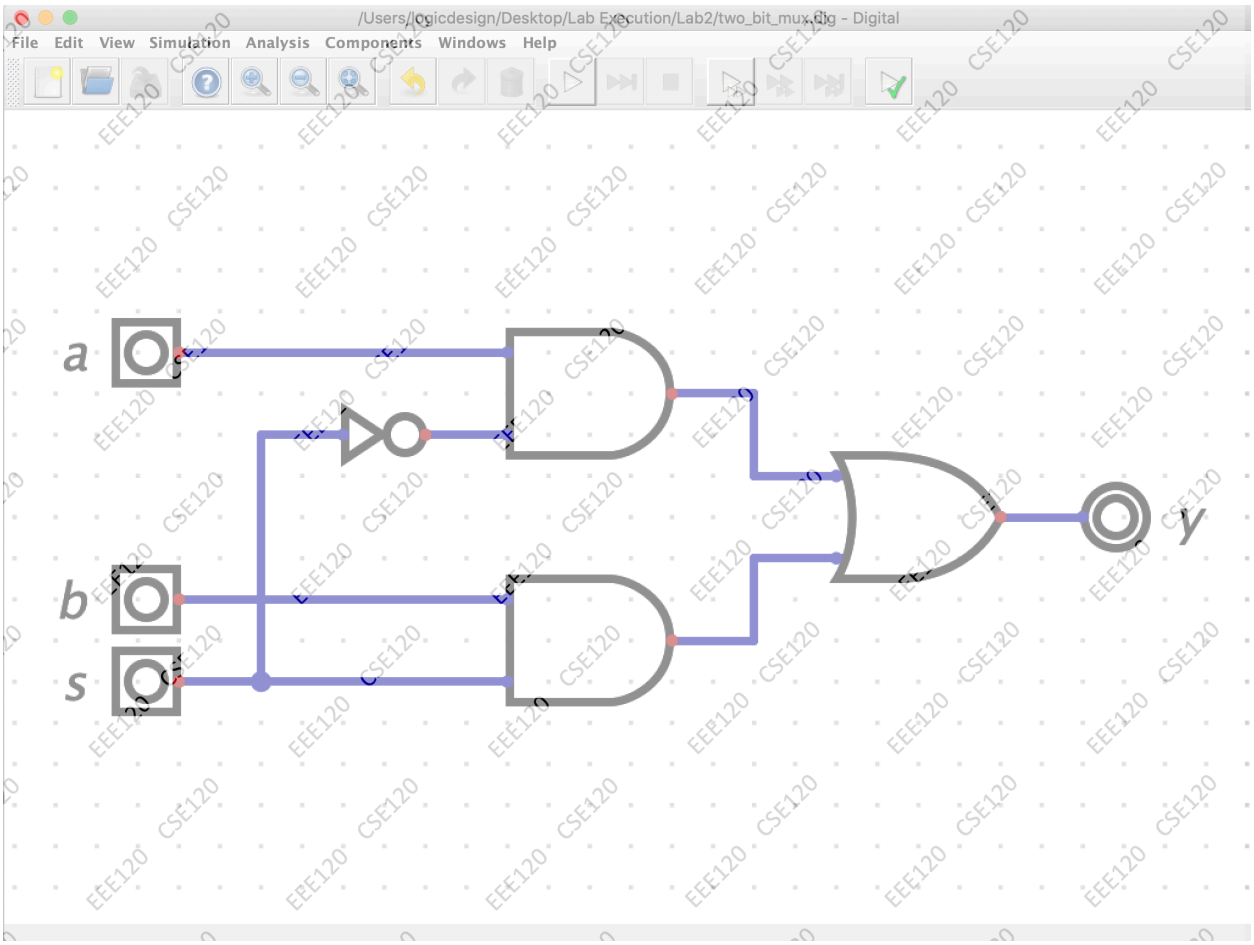

Figure 1. A 2-to-1 mulitplexer.

When done building your schematic, save it with the name two_bit_mux in the Lab2 folder. (Make sure it is saved in the Lab2 folder so you don't have to hunt for it later!) Digital will automatically append the .dig extension.

S	A	B	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Table 1: Truth table for a 1-bit 2-to-1 mux.

The truth table of 2-to-1 mux is shown in Table 1. 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. Satisfy yourself that your circuit correctly implements the truth table in Table 1. Once you are satisfied,

end the simulation and take a screenshot of your circuit and paste it in the template. Also comment on any issues that you encountered.

Task 2-2: Build a 4-Bit 2-to-1 Multiplexer

Since our microprocessor operates on 4-bit numbers, it is necessary to expand the 2-to-1 mux you built in the previous task into a 4-bit, 2-to-1 mux. The 4-bit mux should use a single control/select line to select one of two 4-bit numbers and the selected 4-bit number should appear on the output of the mux. You can accomplish this behavior by placing four instances of the 2-to-1 mux you designed in Task 2-1 in a new schematic diagram. You now need to create new inputs **a** and **b** as 4-bit buses, similar to those you used for the 4-bit full adder in Task 1-5. Consult your Lab 1 manual to review how to properly instantiate the required splitter/merger components. Note that once you configure the splitter/merger for **a**, you can copy and paste it to obtain the one your need for **b**. You also need to create a 4-bit output, **y**, and its splitter/merger. The select input, **sel**, however, remains a 1-bit input!

In Digital, select File->New. Then select File->Save As and save the new file as four_bit_mux, again making sure it is saved in the Lab2 folder. You should then be able to see the two_bit_mux under Components->Custom. Go ahead and build the circuit shown in Figure 2.

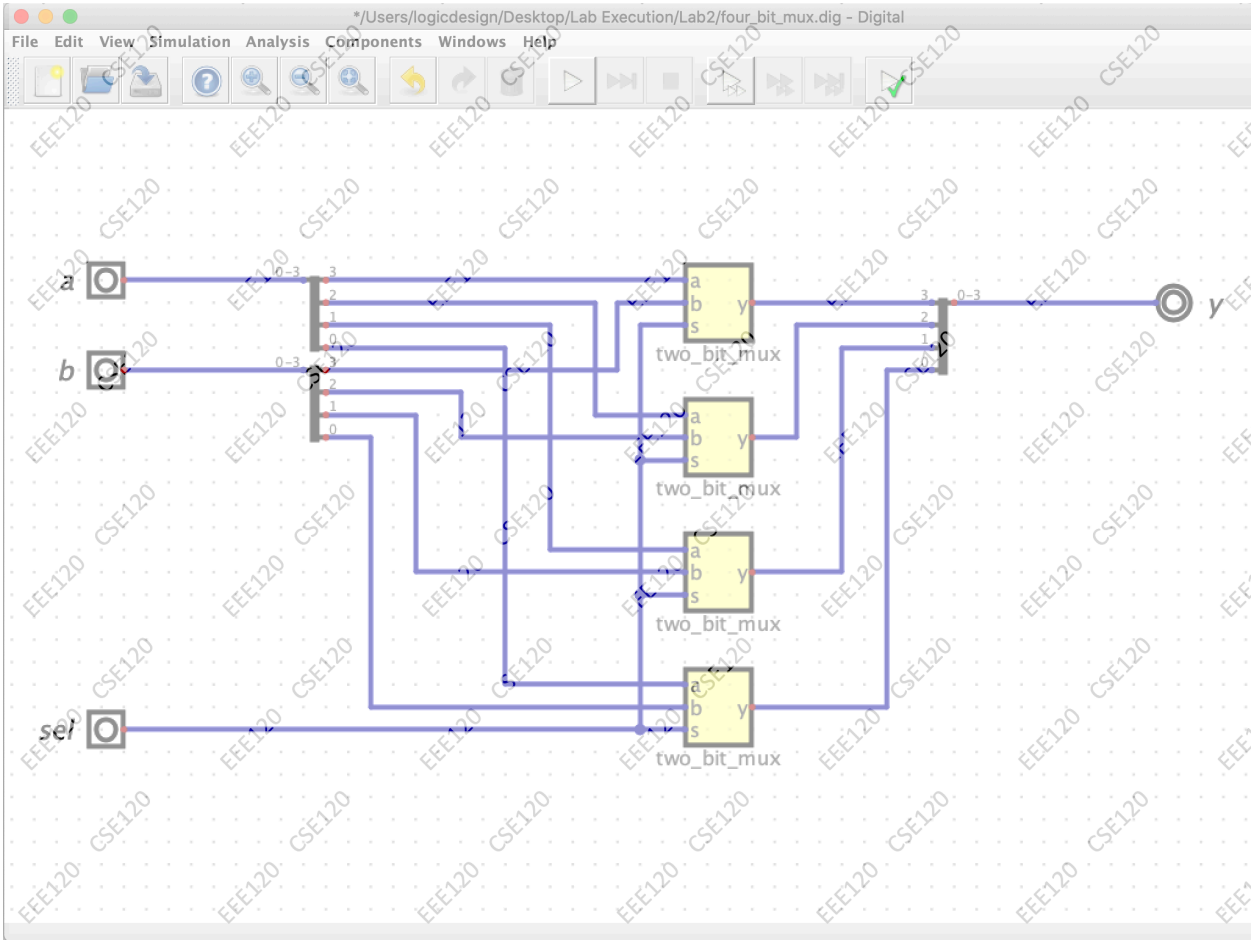

Figure 2. Circuit schematic for the 4-bit 2-to-1 multiplexer.

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

Verify that the circuit is working by simulating it Digital. When **sel**=0, the 4-bit value on **a** should appear on **y**; when **sel**=1, the 4-bit value on **b** should appear on **y**. 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 top banner with the path to your design file in the screenshot.

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

Now, let's simulate the design in iVerilog. You've been given two files to use with the four bit multiplexer: four_bit_mux_top.v and four_bit_mux_stim.txt. Make sure the files are in your Lab2 folder. Execute these commands:

iverilog -o four_bit_mux.exe four_bit_mux.v four_bit_mux_top.v
On Mac: ./four_bit_mux.exe
On Windows: vvp four_bit_mux.exe

Then open four_bit_mux_waves.vcd in GTKWave and click next to four_bit_mux_top to show the four_bit_mux and select four_bit_mux. Double click on the **a**, **b**, **sel**, and **y** signals. The waves should show the circuit selecting either **a** or **b** depending on the value of **sel**.

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.

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.

The stimulus file you've been provided is not a very good test of the circuit. It is up to you to modify the stimulus to test your circuit properly. Modify four_bit_mux_stim.txt to do this. Note that you must have 8 tests – do not change the number tests although you do not need to use them all if you are able to adequately test your circuit with fewer tests. On the template, you will show the tests you used and explain the goal of each test. That is, why did you need that test to verify the functionality of your design?

NOTE: The structure of four_bit_mux_stim.txt is similar to that of four_bit_adder_stim.txt from Lab 1. Review the instructions for modifying that file if you are unsure how to proceed. For reference, the bits in the file are defined as shown in Table 2.

Table 2: Bit definitions for four_bit_mux_stim.txt.

Bit #	15:12	11:9	8	7:4	3:0
Meaning	exp_y	unused	sel	a	b

The 4-bit 2-to-1 mux is another example of how a "smart choice" of test conditions can significantly cut down on testing time. You can approach the problem similar to that of the 4-bit full adder. The most important tests should verify that

- all signals pass through from **a** to **y** (zeros and ones).
- all signals pass through from **b** to **y** (zeros and ones).
- the order of the individual bits does not change.
- there is no cross-talk between inputs (a change on **b** does not affect the output if **a** is selected).

Open the waves in GTKWave and click on the symbol next to four_bit_mux_top so you can see four_bit_mux. Double click on each of **a**, **b**, **sel**, and **y**. Do the waves look correct?

When looking at the waves in GTKWave, changing the format to hexadecimal can make things easier to see. In the Signals column (the one just to the left of where your waves are displayed), select the signals you'd like to display in hexadecimal. (On Mac, press the CMD key to select multiple signals. On Windows, press the CTRL key to select multiple signals.) Then, from the menu, select Edit->Data Format->Hex.

Looking at submodules is easy in GTKWave. To view the contents of one of the 2-to-1 muxes, click on the symbol next to four_bit_mux and you'll see the four instances of two_bit_mux. Click on one of them and you'll see its signals. Add them to the waves. If they aren't in the order you'd like, simply drag them up or down.

Notice that it might be hard to tell which signal is from which module. You can easily separate groups of signals and even name the groups! To add separation, select the signal above the spot where you'd like to insert a gap and select Edit->Insert Blank. If it gets placed in the wrong spot, that's ok – just select it and move it where you want it to be.

To add a label, select the signal above the spot where you want the label to appear and select Edit->Insert Comment. A window will appear. Enter the text you'd like to see. For example, two bit mux 0, and then hit OK. Again, if it's not where you want it, select it and move it.

The ability to view signals at different levels of the hierarchy and to label them will be extremely useful when debugging future labs. Figure 3 shows an example. If you click in the waveform window, a vertical cursor will appear. This will cause the values at that spot to be

displayed in the Signals column. Note that the Signals column can be made wider by selecting the division between it and the waves and dragging to the right.

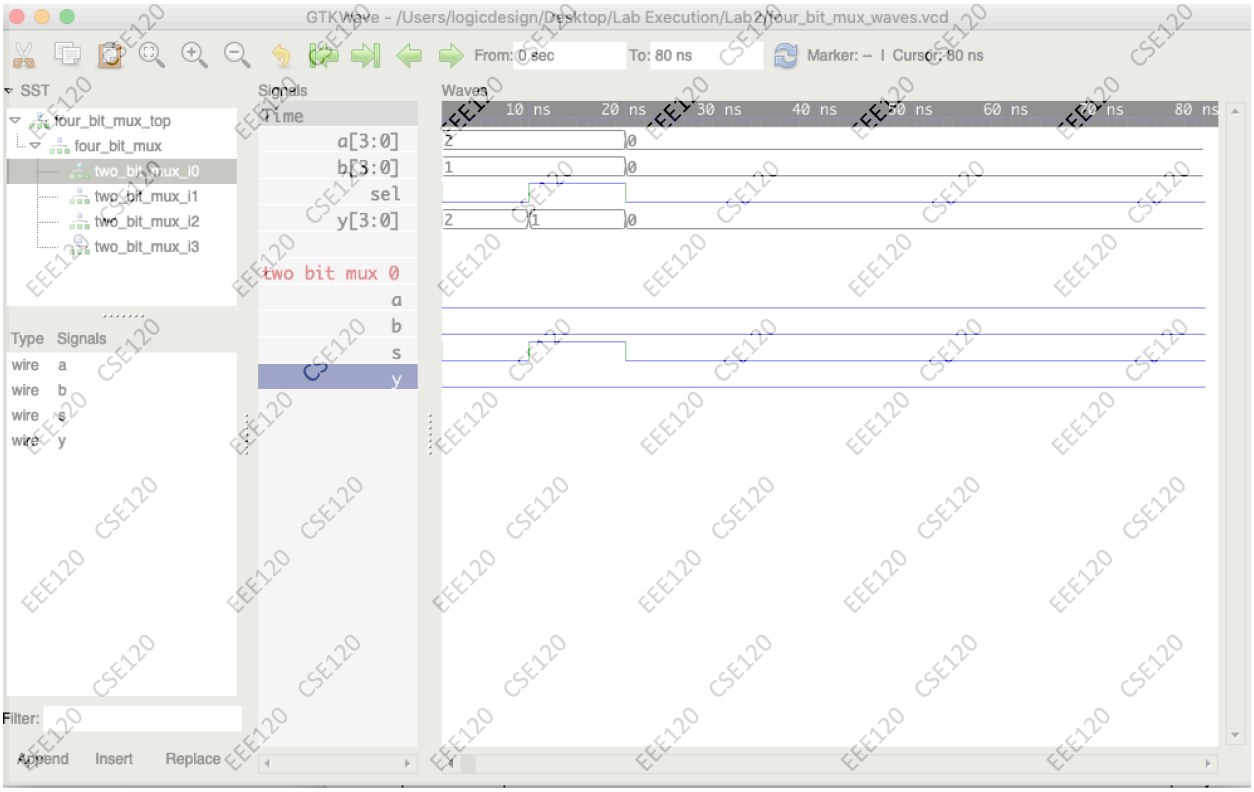

Figure 3. Example waves showing blanks and labels.

Once you are satisfied that your tests are verifying that your circuit works properly, take a screen shot of the waveforms making sure to include the banner at the top of the window which shows the path to your waves. Paste the screen shot into your template. (You do not need to include the 2-to-1 mux signals.)

NOTE: Once you are happy with how your GTKWave window is set up, you can save it. Select File->Write Save File to save your set up. If you come back later and want to look at the same design, you can then select File->Read Save File and your set up will be restored. That way, you don't need to recreate it every time. This will be very handy in future labs.

NOTE: Another nice feature in GTKWave is the ability to reload waves. If you modify four_bit_mux_stim.txt and rerun the simulation, you don't need to exit GTKWave and start over. You can just go to File->Reload Waveform and GTKWave will reload the waves from the same VCD file previously loaded.

Task 2-3: Add 7-Segment Displays to Your Circuit

It would be great to be able to display a hexadecimal value in hardware. For this particular purpose, so-called seven-segment displays were developed. They consist of seven individual LEDs, which are arranged in two squares that sit on top of each other and share one side. They can display the numbers 0-9 and can also display the letters A-F (A, b, C, d, E, F) as shown in Figure 4.

Figure 4: Numbers and letters (hexadecimal coefficients) as shown on a seven-segment display.

That, however, requires turning on particular LEDs depending on the binary input signal. The way to do this is to build a decoder which receives four binary signals as inputs and has seven individual outputs for the individual LEDs, often called a-g (see Figure 5). You could design such a seven-segment decoder by creating a truth table that tells us which LED needs to be turned on based on the binary signal received. Using this truth table, you could then generate minimized logic expressions for each LED. That, however, would require using seven 4-variable K-maps to generate the logic equations. Then, you would have to build the logic. You are welcome to look the equations up in the Marcovitz book (Chapter 5.8.2), but it would take you quite some time to put the schematic together.

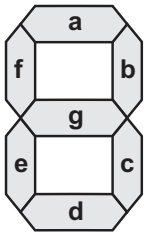

Figure 5: Labels of the individual LEDs of a seven-segment display.

Instead, we will use a component available in Digital which incorporates this logic. Open your four-bit 2-to-1 mux design, four_bit_mux.dig in Digital. Now click File->Save As and save it as four_bit_mux_display. (Digital will automatically add the .dig.) **We will need four_bit_mux.dig later, so be sure not to skip this step!**

Now select Components->IO->Displays->Seven Segment Hex Display and place one by each of **a**, **b**, and **y** as shown in Figure 6. There are several displays – make sure you pick the one described as “Seven Segment Display with a 4 bit hex input”. The two pins on this display are located on the bottom. The four-bit bus is connected to the left pin and causes the hex value to be displayed. (All the decoding logic is inside the component.) The right pin causes the decimal point to be lit or not. We don’t need it, so we’ll connect it to ground. The ground pin can be found under Components->Wires->Ground.

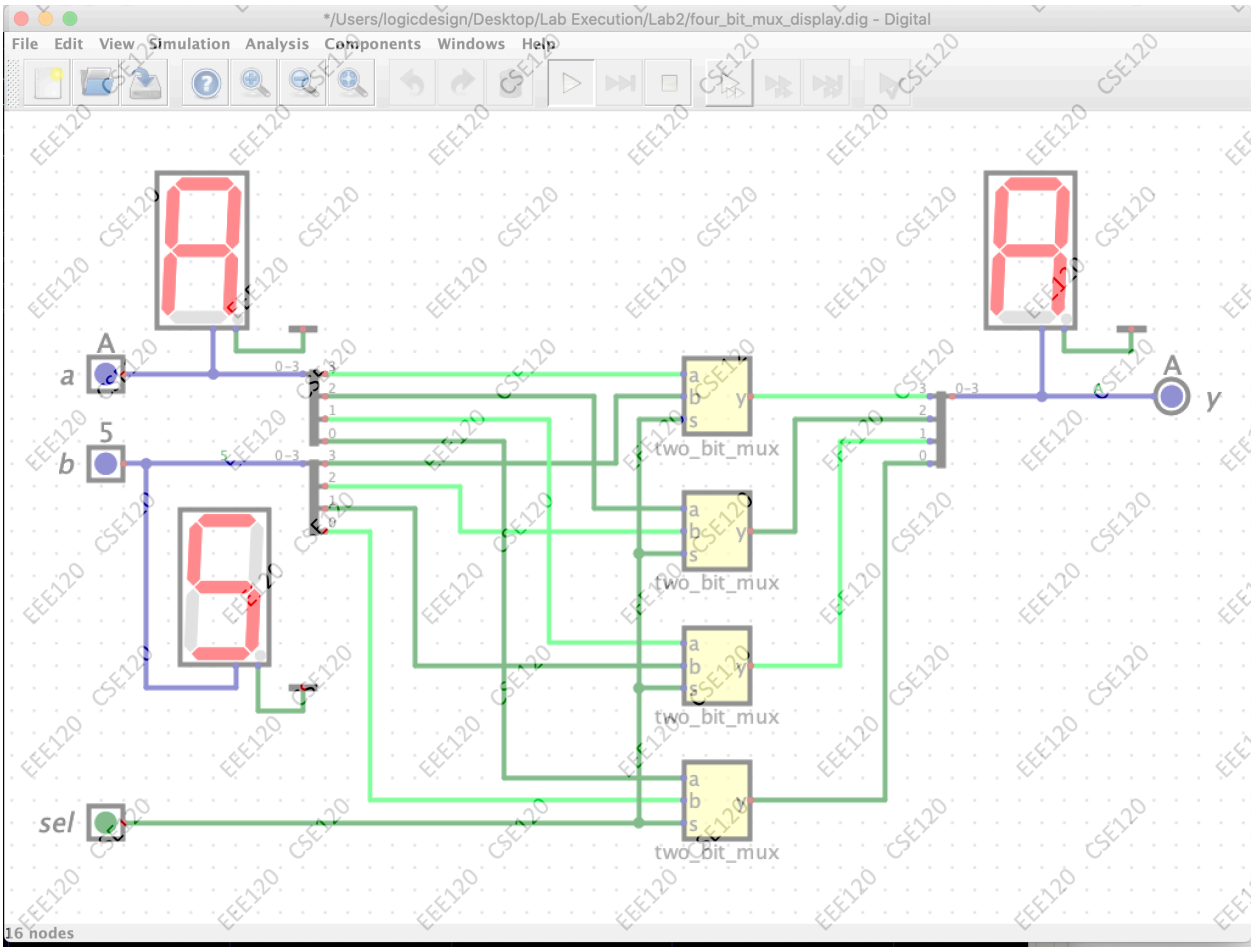

Figure 6. The 4-bit 2-to-1 mux with 7-segment displays.

Simulate the circuit in Digital to make sure it works. Once you are satisfied it is working properly, save the circuit. Place values on **a**, **b**, and **sel** and take a screen shot to prove that your circuit is working. (Choose different values from those shown, and make sure **a** and **b** have different values.) Paste the screen shot in your template.

Task 2-4: Build the NOT/NEG Circuit

In this task we will switch gears and build a module that will perform an operation on four binary digits. The operation it performs is determined by the control signals applied to the module. In Lab 1, you designed a 4-bit increment circuit. You can use this circuit to build another module that can do three things: it can perform the **2's complement operation** (i.e., negate arithmetically), perform the **1's complement** (i.e., logically complement each bit, also known as performing the NOT of each bit) or allow the input argument to **pass through** unscathed.

Select File->New and then File->Save and name the new design not_neg. Make sure you have copied the .dig files from Lab 1 and that they are in the Lab 2 directory. We will build the design

shown in Figure 7. Select Components->Custom->incrementer to place the incrementer in the design. You'll notice that the names of the input INC and the output CRY overlap. Let's fix that. Open the properties for the incrementer (right click on Mac, CTRL click on PC) and then click on Open Circuit. A new window will open showing the incrementer subcircuit. In that window, select Edit->Circuit specific settings. In this case, setting the Width to 4 should be sufficient. Don't change anything else. Click OK. If you save the change by clicking on the save icon or selecting File->Save, the symbol on the not_neg circuit will immediately change size so you can verify that the change was sufficient. Once satisfied, close the window. If you didn't save, you'll be asked if you want to save your changes, say yes. Figure 8 shows the Circuit specific settings pop up window.

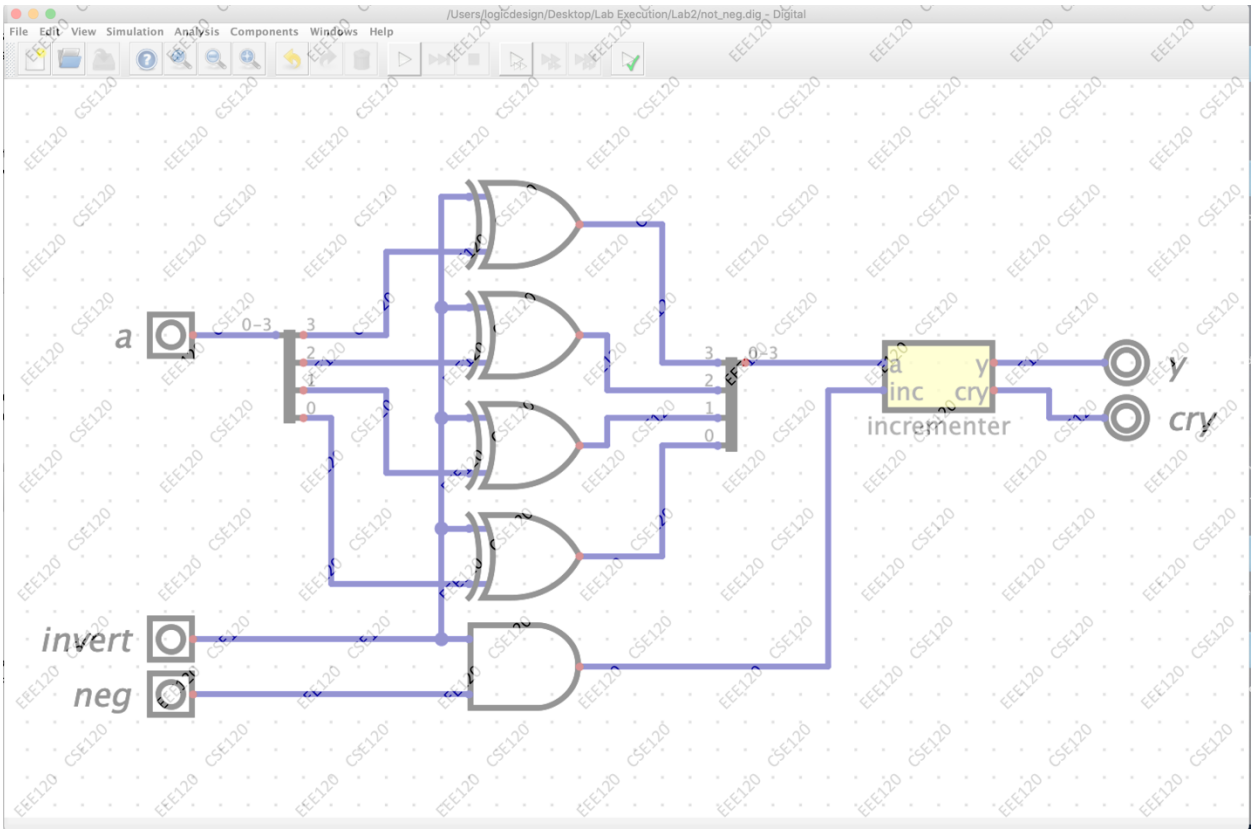

Figure 7. The not_neg circuit.

In Figure 7, two control signals have been added to accomplish the task of performing the three different operations described previously. The control signals are called **invert** and **neg**. Justify to yourself that the **invert** and **neg** signals work together to produce the operations shown in Table 3.

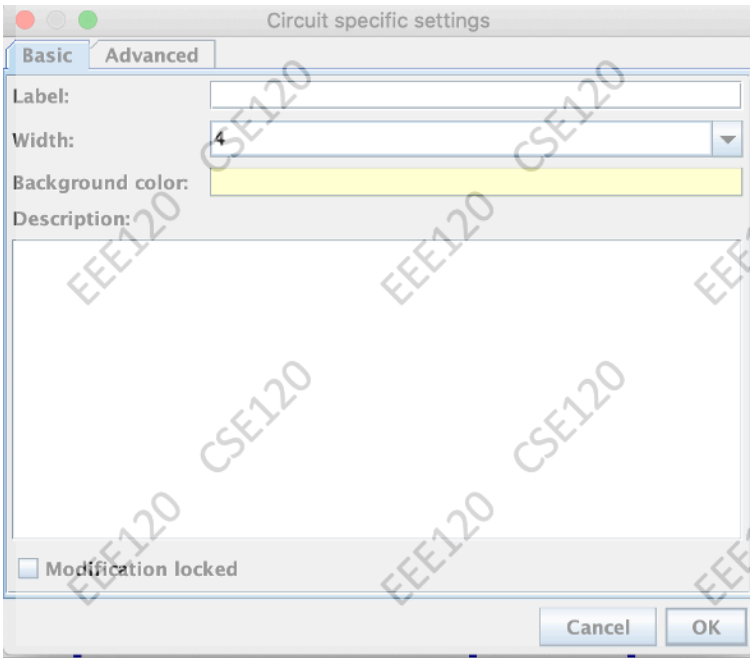

Figure 8. The Circuit specific settings window.

Table 3: NOT/NEG function definition table

invert	neg	Function
0	0	Pass through
0	1	Pass through
1	0	One's complement
1	1	Two's complement (Arithmetic negate)

Simulate the circuit in Digital to verify it is working correctly. Usually, you would have to simulate the circuit behavior in Verilog to make sure that you did not accidentally make a mistake in wiring the circuit diagram. However, we are trying to be bold and go ahead with building the next circuit. You will test the complete ALU circuit at the end of the lab.
Once you are convinced that your circuit diagram is complete, save it and then take a screengrab of your schematic and copy it to your lab template. Also, comment on any issues that you encountered.

Task 2-5: Build the AND/ADD Circuit.

The NOT/NEG circuit is a very simple version of an ALU, so simple that few would call it an ALU. Yet, it can do either an arithmetic operation, the NEGATE, or a logical operation, the NOT of 4 bits. It can also pass data through without operating on it. To justify calling our circuit an ALU, we will want to add more functions than just the three listed in Table 3. In particular, we would like to be able to perform an ADD and an AND of two 4-bit numbers, and we would like to retain the pass-through capability. You have already constructed all of the subcircuits we will

need to build this circuit. Let's build it in two stages; first let's build the AND/ADD circuit; then let's modify the circuit to gain the pass through capability.

Select File->New and then File->Save and use the name and_add. Then build the circuit shown in Figure 9. This circuit accepts two 4-bit operands, **a** and **b**, as inputs, which are fed to the 4-bit full-adder subcircuit (four_bit_adder found in Components->Custom if you copied it from Lab 1) and to the four 2-input AND gates. The 4-bit outputs of the AND gates and adder are fed into a 2-to-1 4-bit mux. This mux is used to select which result (ADD or AND) is routed to the output. (This is a typical example of how multiplexers are used in digital systems.) To control which result is routed to the output, the mux selector input is controlled by the **add** signal. The selector signal notation is consistent with the notation described earlier: when the **add** signal is low, the (logic) AND result is routed to the. When the **add** signal is high, the output (arithmetic) addition result is routed to the output.

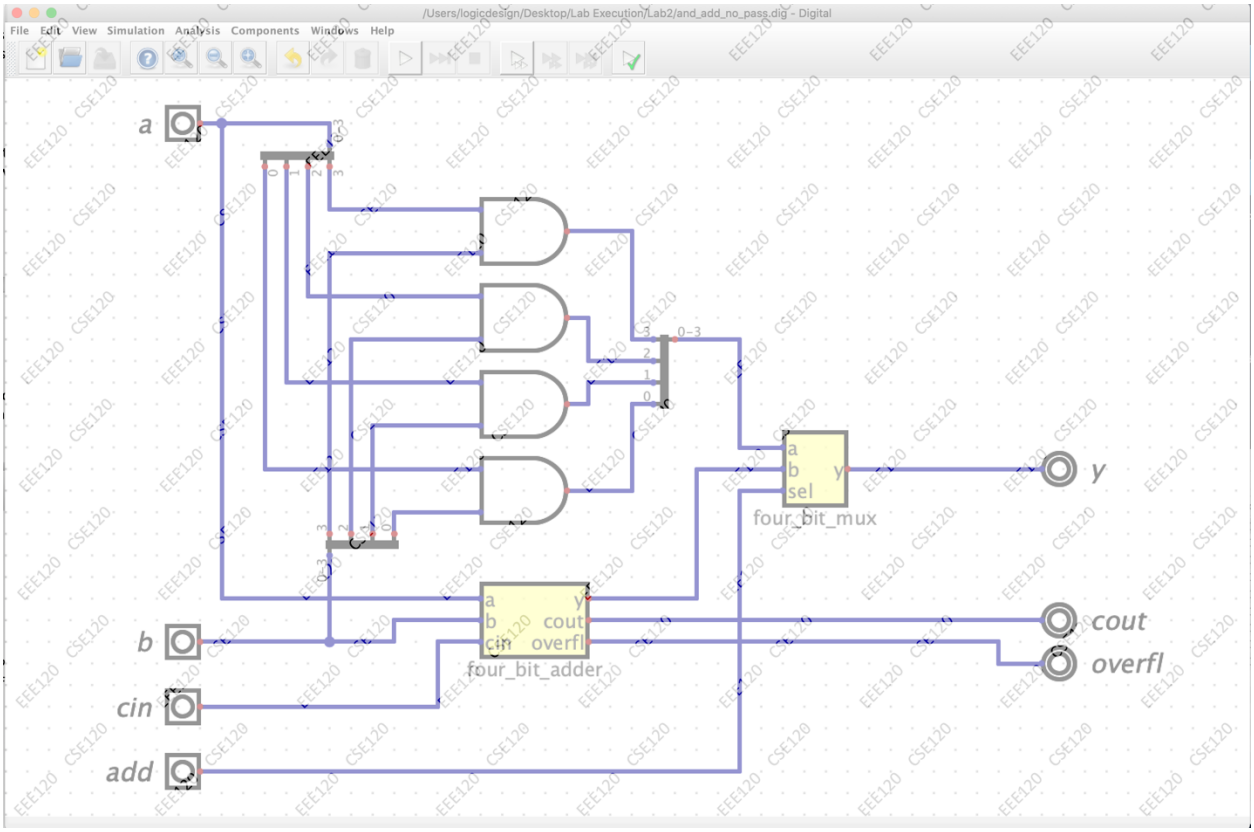

Figure 9. The AND/ADD circuit.

When you place the 4-bit adder, you'll again notice that it isn't wide enough to separate the input and output pin names. Open its properties and use Edit->Circuit specific settings to change its width to 4. But there's more we can do here. The outputs are not in the order we'd like them to be. Click on Edit->Order Outputs and the window in Figure 10 will appear. Select an output and use the up and down arrows to get the outputs in the order shown. Then click OK. Close the subcircuit window and save it.

Figure 10. The Order Outputs window.

Notice that the splitter/merger components connected to **a** and **b** are rotated. To do this, open their properties and click on the Advanced tab. The splitter/merger connected to **a** has been rotated 270 degrees. The one connected to **b** has been rotated 90 degrees. It is not required that you rotate these components, but it does make the schematic a bit neater.

To add the data pass-through capability to the circuit of Figure 9, we add a second 4-bit 2-to-1 mux as the output stage as shown in Figure 11. One input to the output-stage mux is the AND/ADD-circuit output, the other is the **a** input. The selector input of the output stage mux is controlled by the **pass** signal. When **pass** = 1, the **a** operand appears at the output unscathed. When **pass** = 0, the output function, either ADD or AND, depends on the value of the **add** signal appears at the output. The operations performed by the AND/ADD circuit is summarized in Table 4.

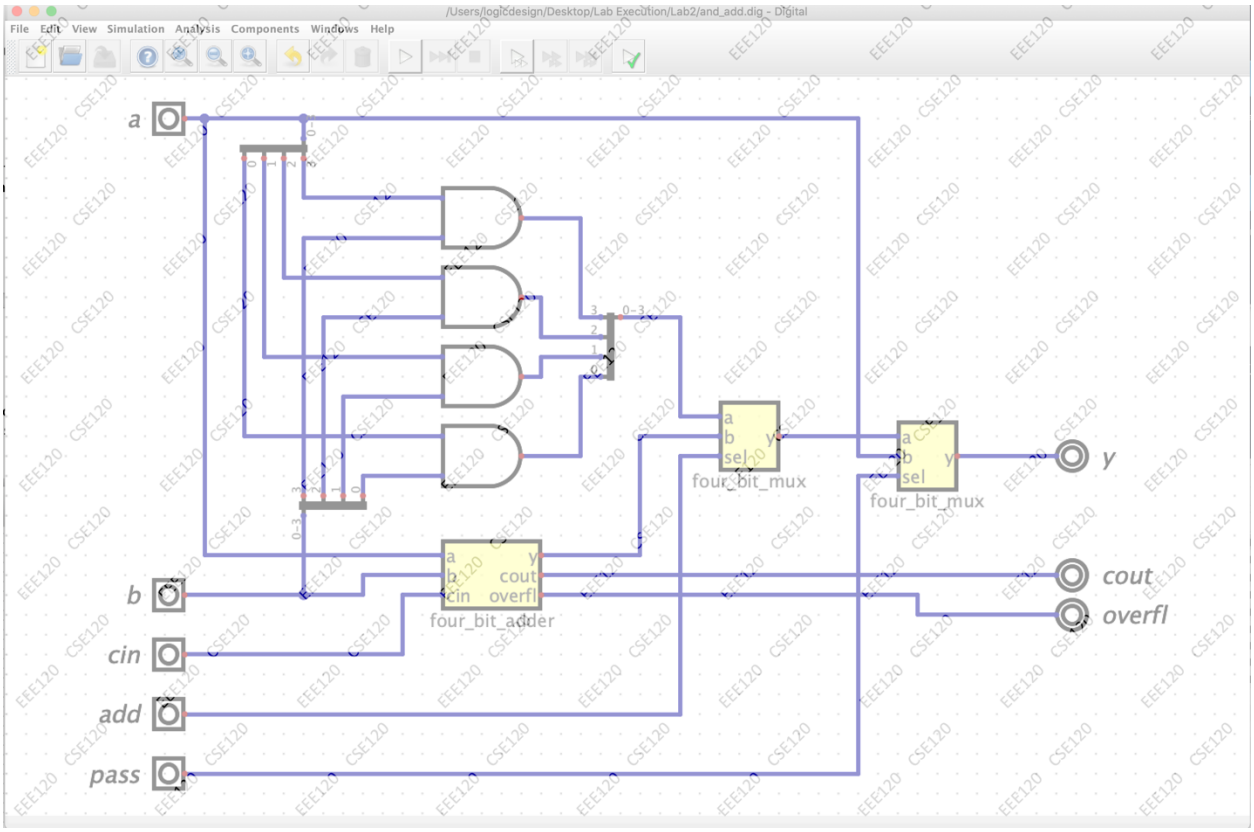

Figure 11. The AND/ADD circuit with pass through.

Table 4: AND/ADD (with pass-through) function definition table

pass	add	Function
0	0	AND
0	1	ADD
1	0	Pass through
1	1	Pass through

Feel free to simulate your design in Digital to verify your circuit is operating correctly. Then save the design and take a screen shot to paste in your template. Be sure to comment on any issues that you encountered.

Task 2-6: Build and Test the ALU Circuit.

The last stage in building the ALU is to combine the NOT/NEG and the AND/ADD subcircuits to create the complete ALU. Select File->New and File->Save As with the name alu. You are now ready to design the ALU circuit shown in Figure 12. Change the widths of the not_neg subcircuit to 4 and the and_add subcircuit to 5. Note that the **cry** output of the not_neg subcircuit is not connected.

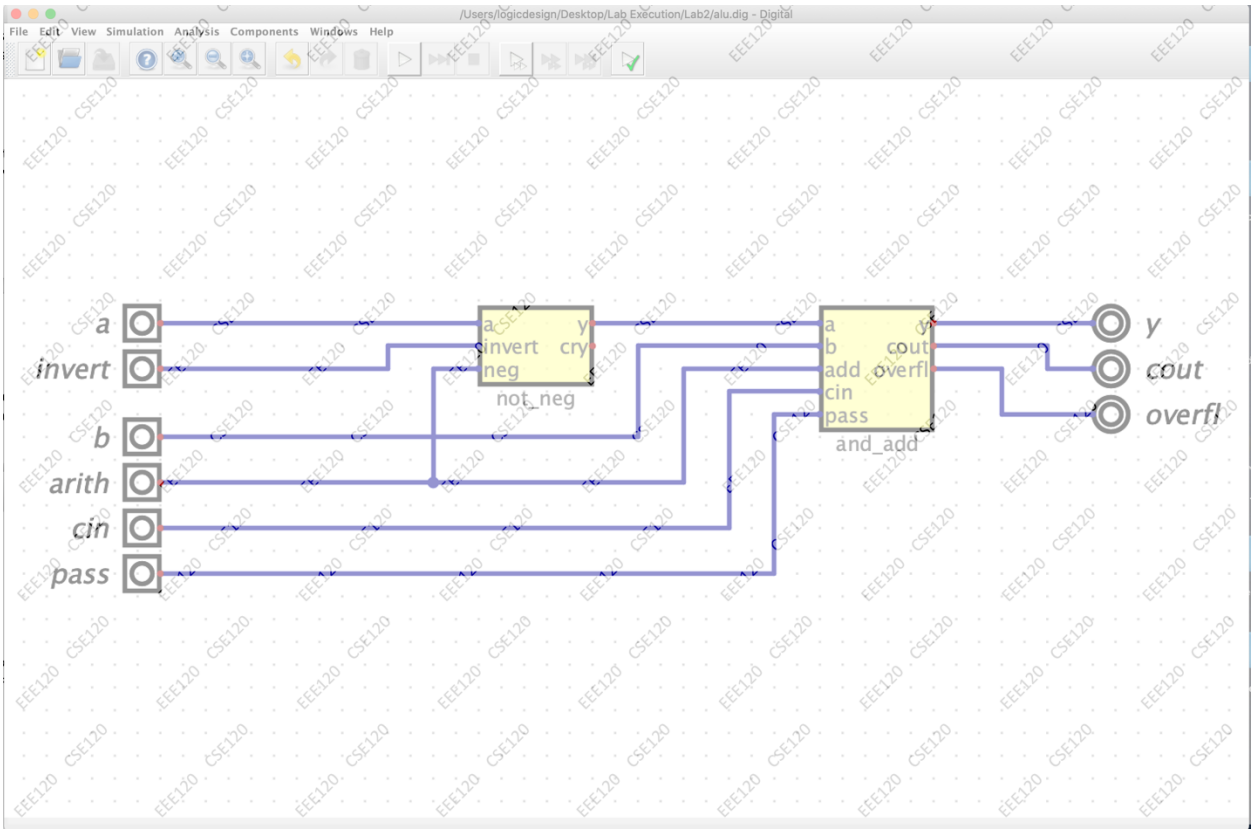

Figure 12. The complete ALU.

This ALU as shown in Figure 12 contains three input control signals:

- The **arith** signal operates so that **arith** = 1, an arithmetic operation is performed. When **arith** = 0, a logic operation is performed by the ALU.
- The **invert** signal, when active, indicates that some type of inversion, either a one's or two's complement is taking place.
- The **pass** signal, when active, indicates that the AND/ADD circuit is passing its **a** input to the ALU output.

Once this notation is understood, you may be able to fill out the function definition table in Table 5 for this circuit even without reference to Figure 12. Consider the example in Table 5:

- **arith** = 1 implies the operation is arithmetic.
- **invert** = 0 implies that A is not inverted (neither a one's nor a two's complement is performed.)
- **pass** = 0 implies that the A and B operands are combined in some fashion.

The only two-operand, noninverting arithmetic operation of which our ALU is capable is the sum, A + B.

Table 5. ALU function definition table

arith	invert	pass	Function
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	Arithmetic Sum, A+B
1	0	1	
1	1	0	
1	1	1	

Complete the ALU function definition table (Table 5) and include it in your lab template.

The next step is to test the function of the complete ALU circuit in a simulation. If you are still not sure how to complete Table 5, you can use the simulation to figure it out. However, that assumes that everything is working as it is supposed to. You can do some simulations in Digital. Once you think your circuit is correct, export it in Verilog as alu.v.

You have been provided two files to simulate your ALU: alu_top.v and alu_add.v. Let's have a look at alu_add.v as you will complete it and then copy and modify it to create tests for the other ALU functions.

IMPORTANT: Do not modify anything above the line in alu_add.v that reads:

// IMPORTANT: ONLY MODIFY BELOW THIS LINE ***

You will be modifying the following items. First, you can change the number of tests you want to run. You can run up to 8 tests. You will determine the number of tests you need to run for each of the 8 functions; the number of tests does not have to be the same for all 8 functions.

For each function, name the VCD file to match the function name. You've been given the ADD function, so the VCD in the file you were provided is alu_add.vcd.

Finally, you need to update the values assigned to test_vals. The test_vals variable is a memory with 8 words where each word is 18 bits wide. It is organized in a manner similar to that seen in Lab 1. Table 6 shows how the bits are defined.

Table 6: Bit definitions for alu_add.v.

Bit #	17	16	15:12	11	10	9	8	7:4	3:0
Meaning	exp_overfl	exp_cout	exp_y	cin	arith	invert	pass	a	b

The first test adds 1+2 and expects the answer of 3 with no overflow or carry out. The 4 indicates this is addition with no carry in. The remaining tests add 0+0. Feel free to change as

many tests as you need to in order to verify the addition function. Don't forget to change the value of **cin** for some tests! To run the test:

iverilog -o alu_add.exe alu.v alu_top.v alu_add.v
On Mac: ./alu_add.exe
On Windows: vvp alu_add.exe

Once you are satisfied, create a test file for each of the other functions of the ALU so that you have 8 files in total. Name them alu_func.v, where you replace "func" with something which corresponds to the function being tested similar to how alu_add.v was named. And name the executables alu_func.exe to match, as alu_add.exe was named. (You change the executable name on the iverilog command line.) You'll have to execute the iverilog command for each test and then execute the exe file. Make sure you take a screenshot of the waves for each of the functions and paste them into the template.

Save the ALU schematic and take a screenshot of your ALU schematic and paste that into your template as well.

Task 2-7: 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. Upload your video to your google drive. Be sure to set permissions so that everybody can see your video and paste the link into your template.

Make sure all of your files are in the Lab2 directory. Create a zip file of the Lab2 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.)

Congratulations! You've completed Lab 2! You will be using the ALU and the 4-bit multiplexer in future labs so be sure to hang onto them.