Notes for Assignment #4

I have created two videos, “Logisim and Circuits” and “8-Bit ALU”, that you should watch as part of this assignment. The material in the last part of Chapter 1 is straightforward; however, you will use this material when completing the first part of Assignment #4. We will now take a break from our discussion of MIPS so we can work with combinational logic. In order to complete your assigned work, you will need to install a logic simulator called Logisim. To do so, please follow the instructions below (I have actually placed the **logisim-win-2.7.1.exe** file under Course Materials on the course website as well):

1) Go to the following URL: sourceforge.net/projects/circuit
2) Save the Logisim executable to your desktop
3) You may now close the sourceforge.net window

We will learn how to use Logisim a little later in this assignment (when we start to learn about gates), but first we need to look at truth tables. The following comments deal with your assigned reading in Appendix B (The Basics of Logic Design) of the Patterson & Hennessy text.

Page B-4: Notice the distinction between combinational and sequential logic; we will focus on combinational logic at first.

Page B-5: You have likely been introduced to truth tables in some other course (possibly in a programming course or a discrete mathematics course). The number of entries (rows) in a truth table is 2^n, where n is the number of inputs.

Page B-6: Boolean algebra offers a convenient way of describing logic functions. The laws that are given in the middle of this page can easily be verified through the use of truth tables. There are similarities to regular algebra, but be careful! For example, these two laws in Boolean algebra definitely do not apply in regular algebra (assuming that you equate $\cdot$ with *):

$$A + 1 = 1$$
$$A + (B \cdot C) = (A + B) \cdot (A + C)$$

A similarity that does hold true is the comparison of Boolean algebra with set theory. Here, + is replaced union (U), $\cdot$ with intersection (∩) , 0 with the empty set (Φ), and 1 with the universal set (U). For example, the distributive law above becomes

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$

in set theory.

Page B-7: Technically, since + and $\cdot$ are binary operators, we should define what we mean by something like $A + B + C$ or $A \cdot B \cdot C$. The obvious definitions will do just fine:

$$A + B + C = (A + B) + C \quad \text{or} \quad A + B + C = A + (B + C)$$
$$A \cdot B \cdot C = (A \cdot B) \cdot C \quad \text{or} \quad A \cdot B \cdot C = A \cdot (B \cdot C)$$

Using truth tables, we can show that it makes no difference if the pairing is on the left or on the right (i.e., the truth tables are the same).

We will use the symbol “~” as the not operator. Traditionally, unary operators have higher precedence than binary operators. Therefore, to do a “not” on something like $A \cdot B \cdot C$, we must use parentheses: $\sim (A \cdot B \cdot C)$.

We will not be using Verilog this term, so we will skip over discussion related to that product. However, the comment at the bottom of this page about the definition of E is just not right! Here is a definition of E that will do (note that the symbol “^” is used for “exclusive-or”, the symbol “&” is used for “and” and the symbol “|” is used for “or”):

E = ~ (A ^ B ^ C) & (A | B | C) & ~ (A & B & C)

This says: E is not exactly one (of A, B and C), and it is at least one, and it is not all three!

Here is another that will do:

E = (A & B) ^ (A & C) ^ (B & C)

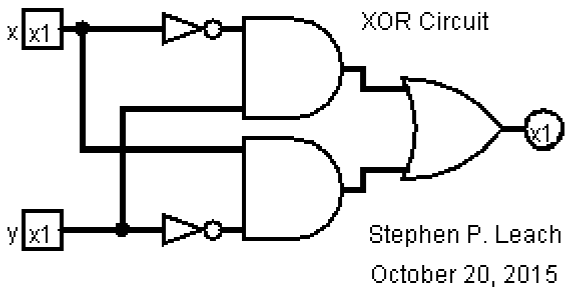
This says: Exactly one of these is true: A and B; A and C; B and C.

Page B-8: We now turn to the topic of gates. Let’s get back into the Logisim software. This product has a wonderful User’s Guide (under Help on the menu bar). Carefully work through the first three sections: Beginner’s tutorial, Libraries and attributes, and Subcircuits (skip the topic “Editing subcircuits appearance”). I am asking you to submit some work related to your going through this User’s Guide, so pay attention to these details:

1) When working through the User’s Guide, it is helpful to have the product visible at the same time (so you can try things as they are discussed). This will be easy if you don’t maximize either of these windows and have them slightly staggered (so that you can easily click on one or the other).
2) The User’s Guide has you construct an XOR circuit from scratch. Save this as **XOR.circ**. You will be asked to submit this as part of your assignment. Likewise, you will be asked to do something with 2:1 and 4:1

MUX circuits. [Brief explanation: A 2:1 MUX has two data inputs and a select input. The select input determines which of the data inputs is allowed to pass though. A 4:1 MUX has four data inputs and two select inputs. The select inputs (treated as a 2-digit binary number) determine which of the four data inputs is allowed to pass through.] When you start your work on the MUX circuits, use the New option under the File tab (or simply exit from Logisim and execute the program again). Save this work as **MUX.circ**.

3) The File tab on the menu bar has a Save option (this option saves your work as a .circ file) and an "Export Image …" option (this option saves a picture of your work). You will want to use both of these options. For example, I produced a .gif file of my XOR circuit, and here it is!



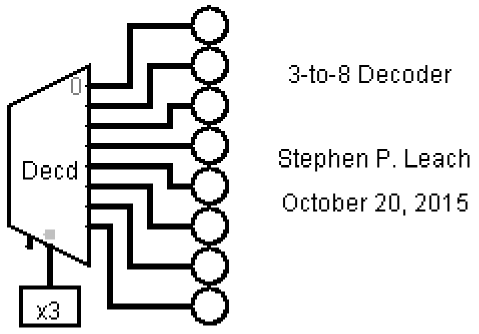
The "x1" that appears in the input and output pins indicates that they represent 1-bit values; later we will see that you can have multiple-bit pins. A four-bit pin would have "x4" inside of it.

4) The User's Guide shows how to make a copy of an individual component (using Ctrl-D). To make a copy of all or a portion of a circuit, select the desired area using the pointer tool (point at the top left and drag to the bottom right so the circuit of interest is surrounded with a box). Right-click the selected area and choose the "Copy Selection" option. At this point, a Ctrl-D will place a copy of that circuit on the canvas (as a "ghost") which you may drag to the desired location.

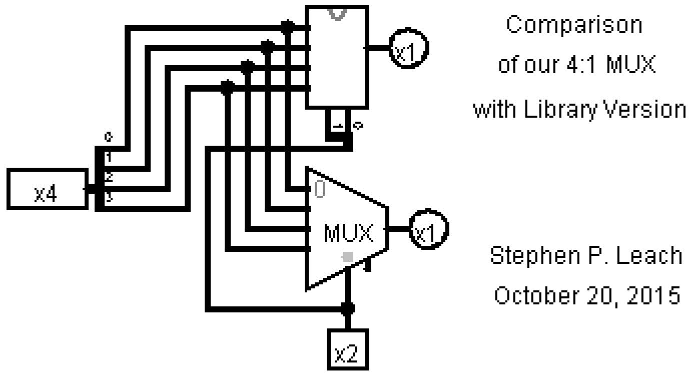
5) The "facing shortcut" is really useful when you start constructing circuits that have input pins, gates and output pins pointing all different directions. Remember that this shortcut exists!

If you don't think this is cool stuff … well, I don't know what to think!

Page B-9: A decoder lets you send it an n-bit binary number and it causes exactly one of its 2^n output lines to be activated. We see a 3-to-8 decoder on page B-9 (the inputs should be labeled "i2", "i1" and "i0", not 12, 11 and 10!). Let's use Logisim to demonstrate! After getting into Logisim, click on Decoder (inside the Plexers library), change the Select Bits to 3, and add a 3-to-8 decoder to your canvas. Click on the input pin tool, change the Bit Width to 3, press the up arrow (since the selector pin for this particular decoder is on the bottom, rather than on the left), and place a 3-bit input pin on the canvas below the decoder. Place 8 LEDs (inside the Input/Output library) on the canvas to the right of the decoder (you will need to move them a little bit to the right so that you can have all of the lines bend around in a reasonable way!). Connect everything up with wires. You can test the resulting circuit with the poke tool, by clicking on one or more of the bits in the input pin. A value of 000 should light up the top LED, 001 should light up the next, and so on. Label your result and save it as **decoder.circ** (I'll ask for this in the assignment). Here is mine!



Page B-10: The discussion on multiplexers should sound familiar; you created a 2:1 MUX and a 4:1 MUX when working through the first part of the Logisim User's Guide. Notice that there is a predefined multiplexer in the Plexers library. I put together a little test to see if both versions of the 4:1 MUX perform the same [the built-in version expects one select pin that has 2-bits, so I used a splitter (inside the Wiring library) to get the individual bits for our 4:1 MUX; likewise, I used a 4-bit data input pin with a splitter … just to show you what can be done). Here is my result (you might want to try constructing this test yourself, but it is <u>not</u> required in the assignment that you will be submitting):



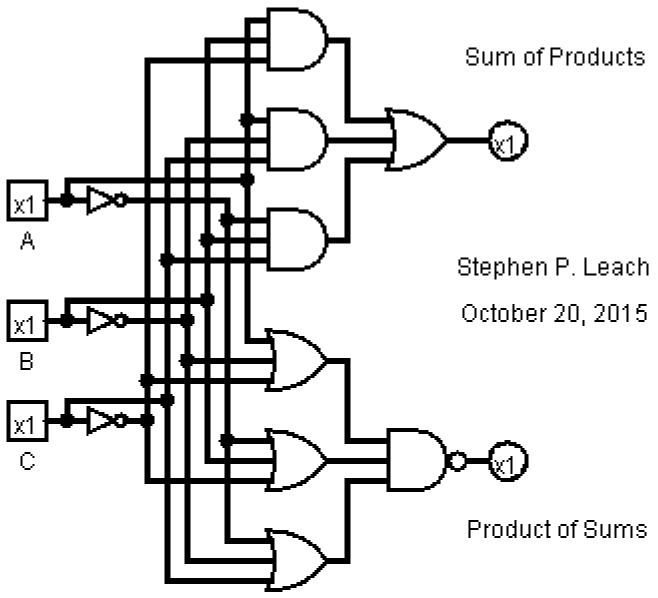
Pages B-11 through B-12: This material on "sum of products" and "product of sums" is pretty important! The following is a sum of products (from page B-7, except using & for "and", | for "or" and ~ for "not" … I have actually rearranged the terms to make them more natural):

E = (A & B & ~C) | (A & ~B & C) | (~A & B & C)

The elaboration on page B-11 says that the following product of sums (I have, once again, rearranged the terms to make them more natural) is an equivalent expression for E (the expression in the elaboration is actually missing a "not" on the last C):

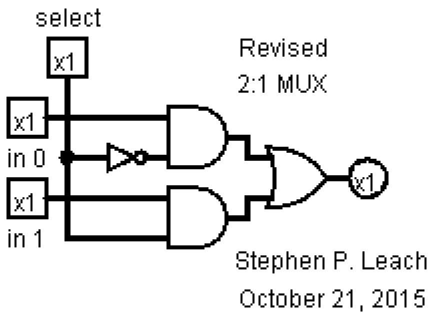
E = ~ [(A | ~B | ~C) & (~A | B | ~C) & (~A | ~B | C)]

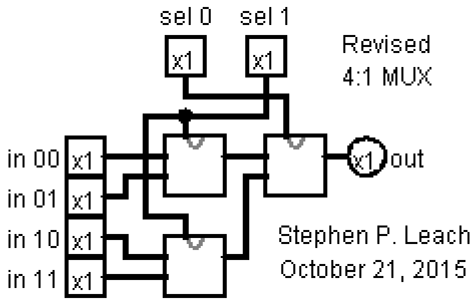
Here is a circuit that I put together to verify the equivalence of these two expressions. The top portion is the sum of products; the bottom is the product of sums. Notice that I changed the gates so that they only have 3 pins and are narrow, rather than wide. I also used a "nand" gate at the bottom right, rather than an "and" gate followed by an inverter. It is a worthwhile learning experience to construct this circuit, so I'll have you do it as part of the assignment. Save it on a file called **ProductSumTest.circ**.



We are now going to work slowly through pages B-26 to B-37 in Appendix B, creating a fairly sophisticated 8-bit ALU from scratch. The text actually describes the construction of a 32-bit ALU, but it will be obvious that no loss of generality will be lost in limiting our efforts to 8 bits.

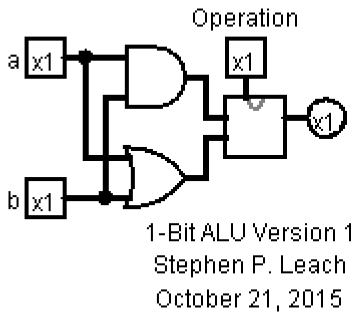
The first thing we need to do is to make a small change to the MUX.circ file that we created earlier. You may recall that the 2:1 MUX and 4:1 MUX that we created had the select bits coming *up* from the bottom of the circuits. It will be helpful if they come *down* from above the circuit so that these circuits will match the multiplexers that appear in the circuits of Appendix B. This is a fairly simple fix. Save your modified file as “Revised MUX.circ”. Here are pictures of my two revised MUX circuits:





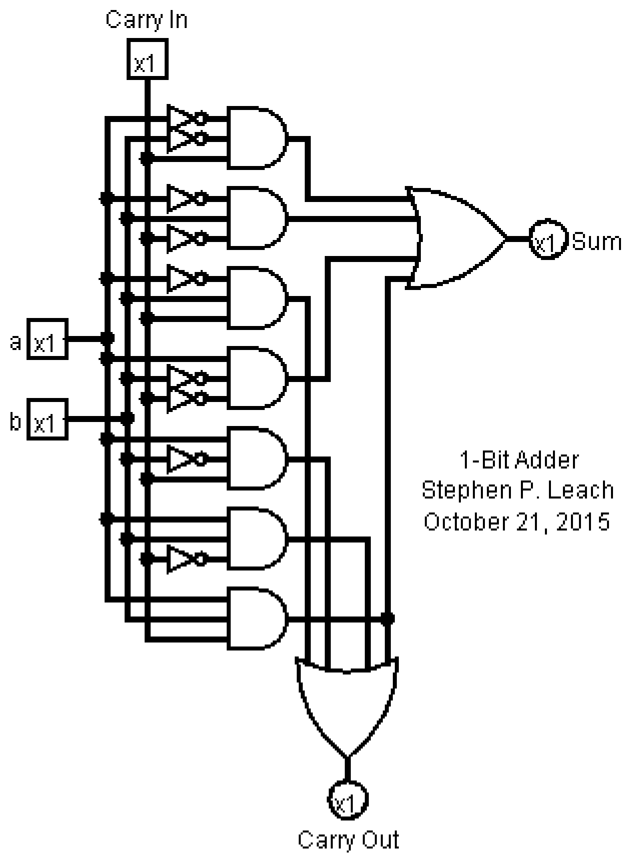
When you create circuits of your own, please remember to include labels on your inputs and outputs. You should also include some form of identification for the circuit itself (including your name and the date that you created the circuit). It may take you a while to complete all of the circuits that are discussed in these notes; be sure to save your file from time to time!

Start a new file inside of Logisim, calling it “ALU.circ”. Load “Revised MUX.circ” as a Logisim library into your “ALU.circ” file. Then create a circuit called “1-Bit ALU Version 1” that corresponds to the circuit shown at the top of page B-27. This is what mine looks like (the box that appears in the circuit is an instance of my 2:1 MUX that is in the “Revised MUX.circ” library):



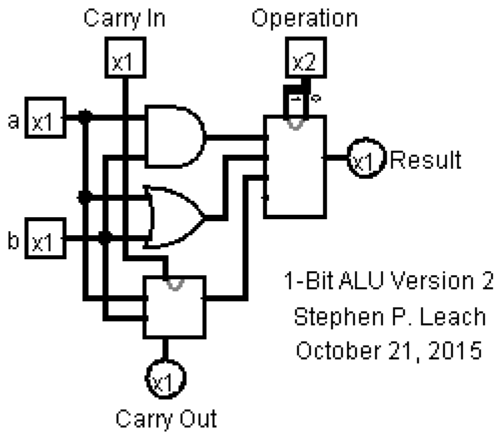
Once you have created your circuit, you should verify that it behaves as “AND” when Operation is zero and as “OR” when Operation is one.

Next, create a new circuit called “1-Bit Adder” (remember that all of these circuits will be part of your “ALU.circ” file). The circuit should conform to the truth table on page B-27 (using “Sum of Products”). Make sure that inputs a and b come in from the left, that the input CarryIn comes in from above, that the output Sum comes out from the right, and that the output CarryOut comes out from below. Here is mine:



You should verify that the circuit produces the right values of Sum and CarryOut for all eight combinations of the three inputs.

Now create a circuit called “1-Bit ALU Version 2” that corresponds to the circuit on the top of page B-29. I have placed a picture of mine following this paragraph. The long rectangle is an instance of the 4:1 MUX from the “Revised MUX” library. Notice that the Operation input is 2 bits in size (this is an attribute that you can change for a pin) and a splitter is attached to it that fans out those two bits. The smaller box is our “1-Bit Adder”. You should carefully verify that your circuit either does an “AND”, an “OR”, or an “ADD”, depending on whether the Operation value is 00, 01, or 10, respectively (11 is not used at this point).



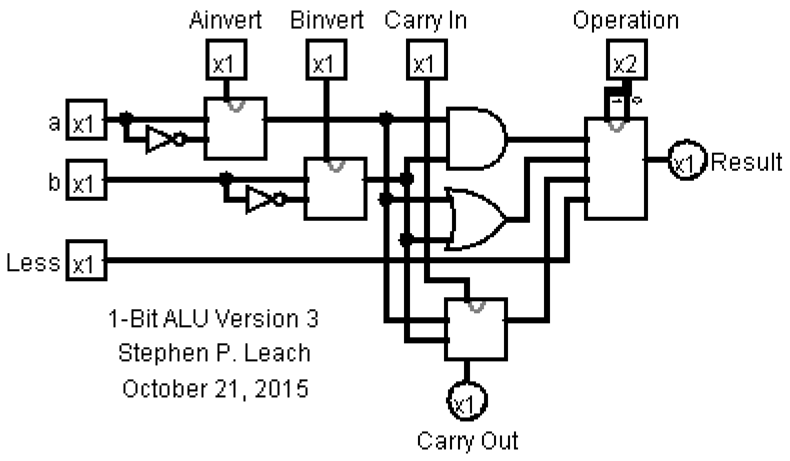
On pages B-29 through B-33, the authors add some additional features to the already powerful 1-Bit ALU. Remembering that you can subtract b by simply adding the negative of b, we avoid having a separate subtraction operation. Instead, we allow b to be complemented (along with changing the Carry In on the low order bit to a one, this accomplishes the subtraction).

The authors also allow the complementing of a as well, so that the NOR operation can be accomplished by ANDing the complements of a and b. This works because DeMorgan's Theorem tells us that

a NOR b = ~ (a OR b) = (~a) AND (~b).

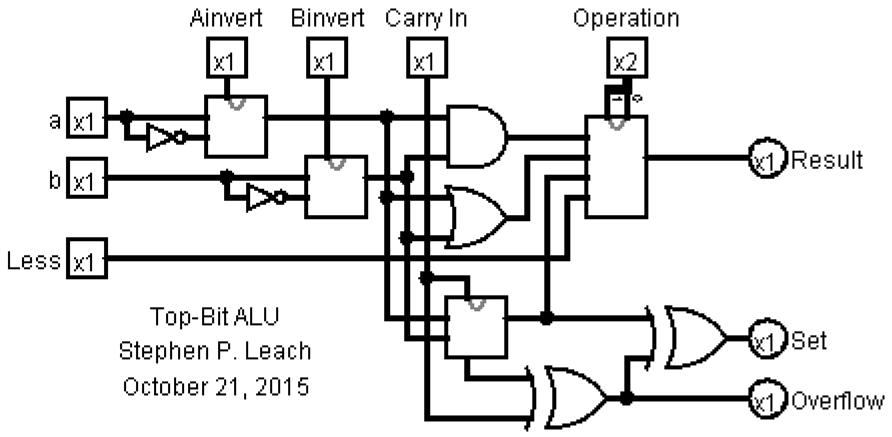
Finally, the authors add the capability of the slt operation. As you recall, this instruction compares a and b. If a is less than b, the result should be the number 1; otherwise, the result should be zero. We are using 2's complement numbers at this point, so we need to remember that there are negative and positive numbers to worry about. There is also the issue of overflow. You should read the discussion on pages B-32 through B-37 carefully to see how these items are handled. In particular, you will see that we end up needing a special 1-Bit ALU for our most significant bit.

You should create a Version 3 of our 1-Bit ALU that incorporates the negation of a and b, as well as implementing the 4th operation, slt (which is operation 11). This version of the 1-Bit ALU will be used for bit positions 0 through 6 of our final 8-Bit ALU. Such a circuit appears at the top of page B-33 in Appendix B. Here is my version of that circuit (call yours "1-Bit ALU Version 3"):

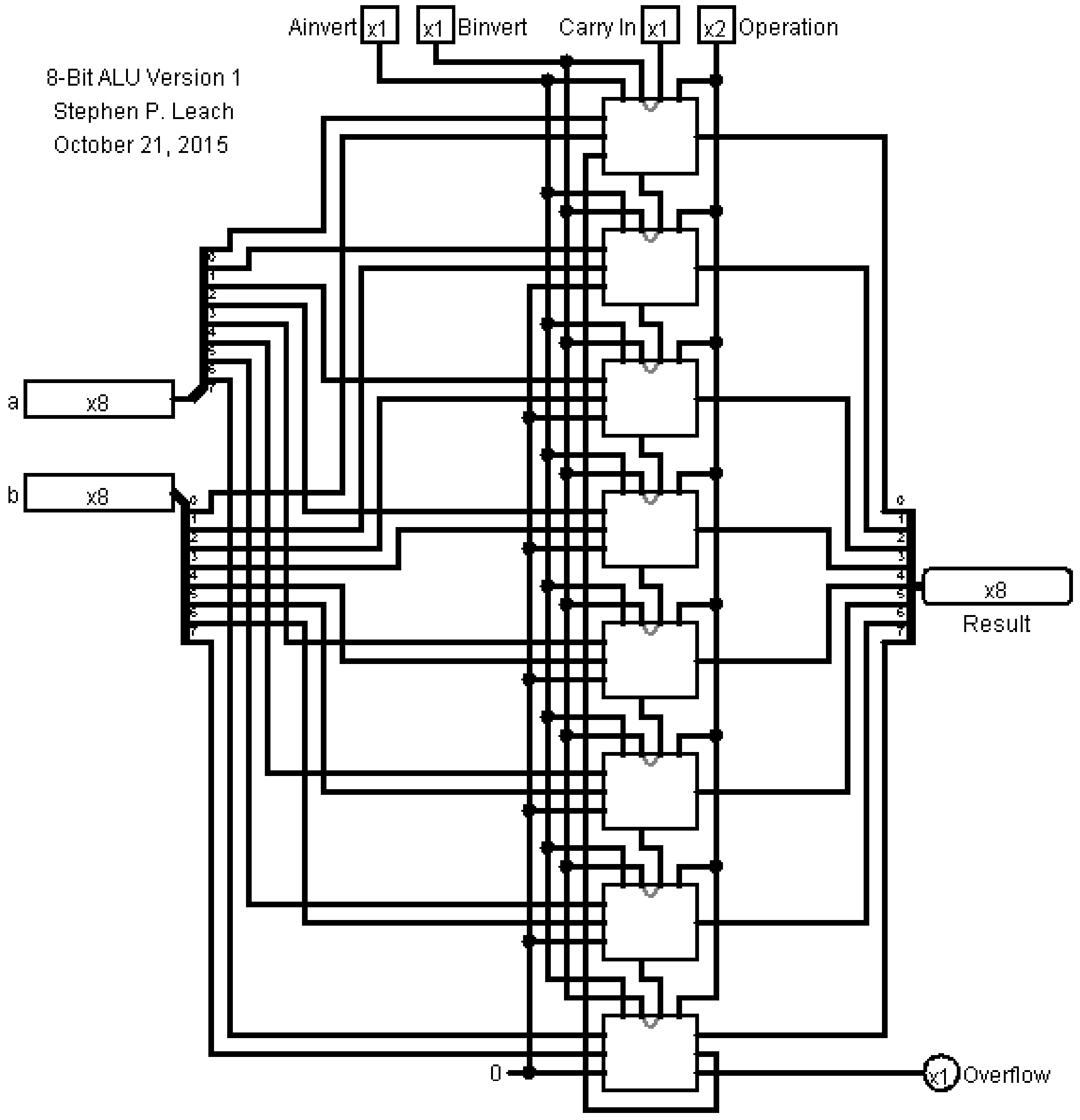


Aivert and Binvert are feeding into one of our Revised 2:1 MUXs. Carry Out is coming out of one of our 1-Bit Adders. Operation is feeding into one of our Revised 4:1 MUXs. You should verify that the various 1-Bit operations are working properly to ensure that your circuit is correct.

The ALU for the top bit (in our case, bit 7) is a little more complicated, since it needs to handle the setting of bit 0 (if a is less than b) and the detection of overflow. The text actually leaves the details of these two items partially as exercises, but I will explicitly show you how to handle them by showing you my solution. Notice that Overflow is simply the exclusive OR (XOR) of the Carry In (from Bit 6) and the Carry Out (from Bit 7). [The XOR is located in the Gates folder.] The value of Set ends up being the exclusive OR of the Overflow value and the result bit (Bit 7) coming out of this ALUs Adder (the circuit on the bottom of page B-33 shows it simply as the result bit from the Adder, but the special circumstances involving Overflow are explained in an exercise at the end of Appendix B). Here is my solution (you will need to reconstruct it yourself, since it will be used in constructing our final 8-Bit ALU … call yours “Top-Bit ALU”):



We are finally ready to put together an 8-Bit ALU similar to the one that appears on page B-34 of Appendix B. It looks a little scary, but with a little explanation you should see how it works. Eventually, you need to reconstruct this circuit yourself, calling it “8-Bit ALU Version 1”. First, here is my solution (comments will follow):



There are 8 subcircuits in the circuit above. Starting from the top, there are seven instances of “1-Bit ALU Version 3” and then one instance of “Top-Bit ALU”. There are 8 bits in a, b, and Result. In the circuit, notice that a, b, and Result have a splitter that fans out with eight branches.

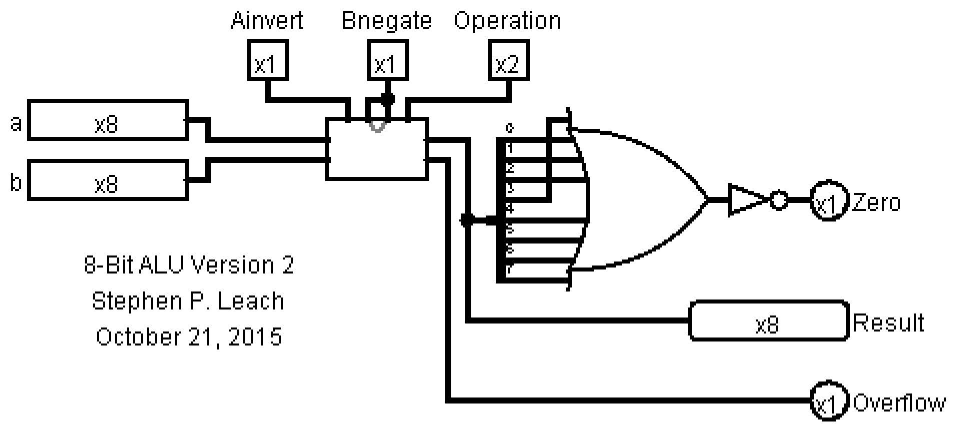
You can see that the Set output of “Top-Bit ALU” is connected to the Less input of the top “1-Bit ALU Version 3”. The other seven subcircuits have the constant 0 connected to their Less input (Constant appears under Wiring; you will need to change the value attribute to 0x0).

It is time to seriously test your result. Remember that with only eight bits, the largest positive value is 0111111_2. Numbers with a top bit of 1 are negative (two's complement). Here are some tests that I ran (you'll want to do others):

1) Verify that 1 + 1 is 2! (Operation 10). The "AND" of these two values (Operation 00) should be 1, as should be the "OR" (operation 01). The "NOR" should be -2 (Operation 00, with Ainvert and Binvert both set to 1). You might as well verify that 1 -1 is 0 (Operation 10, Ainvert back to zero, Binvert stays 1, and CarryIn set to 1). You can verify that 1 is not less than 1 (Operation 11, leave Binvert and Carryin as 1). If you change b to be 3, you can verify that 1 is less than 3).
2) Let a be the biggest positive number (one zero, followed by all 1's) and let b be 1. Verify that you get an overflow when you add. Instead let b be -1 and subtract (remember, to subtract you set Binvert and CarryIn); you should again get an overflow.
3) Verify that -1 is less than zero.
4) Verify that if you add largest negative value (1 followed by all 0's) to itself, you get overflow (big time!).

Get the idea? Experiment with it; share it with family and friends!

The textbook suggests one more enhancement to the 8-Bit ALU (see the picture on page B-36 in Appendix B). Create a new circuit, "8-Bit ALU Version 2," that incorporates this change. You should use "8-Bit ALU Version 1" as a subcircuit in this new circuit, incorporating the new features "Bnegate" and "Zero". The picture on Page B-36 is unclear on the order of the outputs on the right. I would like you to put "Zero" on top, then "Result" and then "Overflow". Here is mine:


Ainvert
Bnegate
Operation
x1
x1
x2
a
x8
b
x8
Zero
x8
Result
Overflow
8-Bit ALU Version 2
Stephen P. Leach
October 21, 2015