CMSC389E Project 3: Finishing the ALU

Assigned **Friday February 25**

Due **Thursday March 17**

1 Looking More like a Computer Every Day

Now, we want to look at finishing up our ALU and making sure that all the components are connected properly. Also, we want to add the key functionality that'll allow us to request the operation we want from the ALU and have it spit out the correct result.

Our main focus will be in the implementing the operation selection, which we will do using a **encoder/decoder** setup.

This is a great way to introduce us to the practical uses of the encoders and decoders that we covered more theoretically in class, as we are now **selecting an operation** based on a given opcode.

Obviously, the implementation here is going to be a little divergent from the exact implementation that we covered in class, but by applying the concepts that we covered in class, you should be all set for success on this project.

At the end of it, you'll have implemented a perfectly functional opcode-based setup where you can input the operation you want, and the input parameters, and then receive the output!

2 Conceptual Overview: Encoding/Decoding

You will need a way to specify for your ALU which operation that you'd actually like to perform. There are a few ways to do this reliably, but the best way would most probably be to go forward with a concept from class. Luckily, the Encoder/Decoder family of circuits is incredibly fit for the task.

In this instance, you'll simply be taking in an input in the form of a 4-bit binary opcode number, and translating it to a decimal output (choosing one of the 9 operations that we have specified). Hopefully, this language should remind you of the encoder and decoder lesson from class.

3 Setup

Our first order of business will be to add a few more functionalities to the ALU. Specifically, now that we've done the tough stuff and built out the architecture for those tougher operations, we want to add a few more easier operations as well. We selected these operations specifically because they are very common in most assembly languages, and we think you'll find them useful for emulating computing circuits in the following projects.

We will start by loading the appropriate blocks for the first part of this project.

```
/test load 4
```

You will see that in doing this, you have the same input blocks as provided in the previous projects. Additionally, you will be given `oCntrl0`, `oCntrl1`, `oCntrl2` and `oCntrl3`. You will use these as the inputs for your encoder for your opcodes. In terms of outputs, you will have only one clean set of outputs now: `oOut0`, `oOut1`, and `oOut2`. You will connect these to the end of your output bus so that they will capture the appropriate 3-bit output of any of your input functions.

4 Adding a Few More Functions

First and foremost, we want to add just a few more functions to our ALU, as follows. These are all pretty simple, but one of these steps involves taking our adder that we created previously and expanding upon it so that it can also function as a full subtractor. This will be explained in the section below.

4.1 Bitshift Left

This one is pretty simple. Take the `a` input bus and return the logical shift left of it. The low bit (0) should remain off.

4.2 Bitshift Right

This one is the same. take the `a` input bus and return the logical shift right of it. The high bit (2) should remain off.

4.3 Subtraction

Here's the cool part- you can turn the adder that you have into a functional subtractor just by changing it slightly. We'll touch more upon this in the section below on converting

your adder into a functional subtractor.

For now, all you need to know is this- just as you have implemented the addition operation, you now have to implement the subtraction operation as well.

You will want to return the result of `a - b`.

4.4 Set if Less Than (SLT)

Recognize this from assembly code? You'll find this neat little conditional useful as well. We want an easy way to implement this. I would recommend that you first work out the details of your subtractor, after which this should be a piece of cake.

Run the inputs through the subtractor circuit as you would for a subtraction operation, but then at the end, have a flag that catches if the result of the subtraction was negative or not.

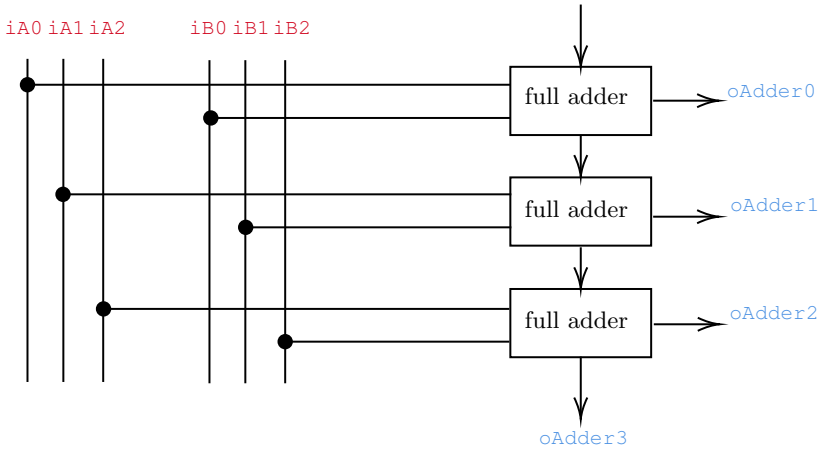
Essentially, we will want to return `1` if `a - b < 0`. Else, we will return `0`.

5 We Giveth (Adder) and We Taketh Away (Subtractor)

The bitshift implementations should take you a few minutes at most. They should not be more complicated than just you shifting some of the input wires to each side (appropriately). Now, we will want to work on turning out adder circuit into an adder/subtractor.

Luckily, we can take care of this pretty easily. That is, we can take our circuit that **only** does adding right now in three bits, and make some adjustments to it so that it can add **or** subtract, based on an input flag (which we will affectionately refer to as the *control wire*).

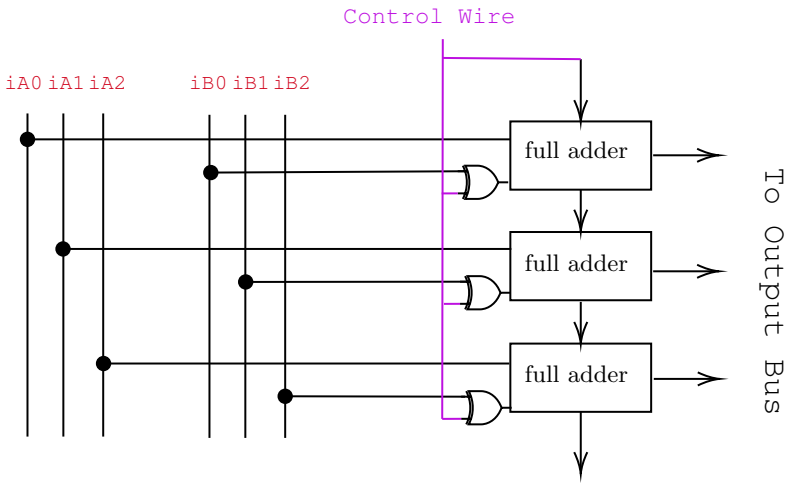
First things first, let's explain how this works. Currently, your circuit for adding should look a little something like this:



I have taken the liberty to include your old output blocks in the above diagram, just to remain consistent with what your design should look like right now. You now have two tasks. First, retrofit your adder setup with some changes so that it can successfully subtract as well. Second, instead of connecting your outputs to specific output blocks, send them all to an output bus.

In order to allow us to intelligently invert inputs and outputs when we need to facilitate complement operations, we will be augmenting our circuit as such. This will give us the ability to subtract numbers, as well as add numbers- all based on the control wire.

The diagram below shows you how your adder/subtractor circuit should be now- after we have made the improvements discussed above.



You will see in the diagram above that we have made a few changes. First of all, we've added a 'control wire' to the diagram. When the control wire is set to `0`, the circuit remains an adder, as it has been before. We leave the truth table for this as an exercise for the reader, so that they may see that they are the same. When the control wire is set to `1`, the circuit becomes a subtractor. We recommend that the reader give this a few tries on paper first, before attempting to implement this in Minecraft.

This is our recommended solution, but there is more than one way to turn an adder into a subtractor. If you come up with a solution that you think is more convenient to implement, may work more efficiently, or is just plain cooler looking, please feel free to implement it! As long as it passes the tests, it's good by us.

This setup is valuable to us because we can quite easily just determine when to activate and deactivate the control wire in order to get the outputs that we want. This also makes implementing SLT quite simple. We recommend that you test all of your individual functions before moving onto the next portion of the project, just to ensure that you have no issues that are caused by errors in operations.

6 The Opcode Controller

Now, we're going to use our magical decoder powers to turn this thing into a real arithmetic logic unit. That is, we're going to accept a **4 bit binary number** as an opcode, and appropriately determine the right ALU function to perform upon `iA0`, `iA1`, `iA2` and `iB0`, `iB1`, and `iB2`.

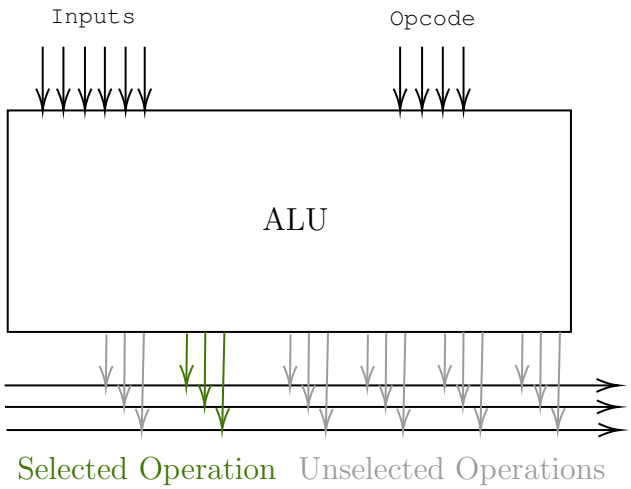
We will accomplish this using a decoder. Following are the inputs that we are expecting, and the functions that they select for. Note that the inputs on the left are provided in decimal, but you will be expecting them as `4-bit binary` for this circuit. E.g. 15 will be provided as the input `1111`, 7 will be provided as the input `0111`, and so on and so forth.

0	XOR(A, B)
1	AND(A, B)
2	OR(A, B)
3	ADD(A, B)
5	MULT(A, B)
6	BITSHIFTL(A)
7	BITSHIFTR(A)
9	NOT(A)
11	SUB(A,R3)
15	SLT(A, B) (is $A < B$? 0 if yes, 1 if false)

The entire idea is to ensure that, with the help of some `AND` gates, we only allow input to reach the logical circuit that we are selecting for. As such, the only thing that will be sent down the output buses at the end of this will be the outputs from the selected operation.

As you've seen many times in this class, learning about a logical circuit, then seeing its truth tables and diagrams, is usually different from the implementation in Minecraft. This is usually because the features that Minecraft affords us can help us in unique ways to make design more convenient, and this case is no different.

Here's a general look at what your finished ALU circuit should look like. Notice how we facilitate the 4 opcode control wires as input, as well as how we have redirected all the ALU outputs to the output bus.



Doing the latter is perfectly fine now, as we can ensure that only one operation is expected to be selected at a time.

6.1 Video Resources

We think that these pieces in particular are best described with video tutorials, which we have linked below. They are brought to you courtesy of Ashwath Krishnan, a former instructor for this class with a very cool gaming and streaming setup (hence the great video and mic quality).

- Part 1: Decoder in Minecraft
- Part 2: Output Bus Configuration

We recommend watching both of these videos in order, as they'll provide a lot of excellent insight into building out these components in the most efficient way possible. As a side note, we suggest familiarizing yourself with decoders, as they will be a key part of the next lecture and project as we finally start writing our own 'code'.

7 Testing & Submission

You will test and submit this project in the same way that you have done for every other project in this class. However, we are now getting to the point where you are attemping tests with a pretty large circuit, that will most probably have some settling time.

It is at this time that you should **ensure that none of your repeaters are set past the first position**. Our mod makes it so that the delay is pretty much nonexistent if the repeater is in first position, just to ensure that the settling time for your circuit is minimal. That is, our mod makes sure that redstone signals are not held up at repeaters for very

long at all, and as such, they won't take long to propagate as long as you don't change the repeaters from their default setting.

After having ensured this, go ahead and run your tests using the following:

```
/test start
```

Despite this, your signals may still take a little bit to propogate through the circuit. If you are finding this to be an issue, simply perform your tests with a delay by using the seconds parameter in the `start` command. I prefer using a delay of 2 seconds, as follows, to properly ensure that the signals are traveling through my circuit correctly.

```
/test start 2
```

Fun fact: This is not just a problem in Minecraft! In real life, electricity also takes a finite amount of time to travel through circuits- it doesn't just happen instantaneously. Based on the complexity of the circuit, this can be a significant amount of time, which led to the creation of the term ***settling time*** *for CPUs. This is the amount of time that it takes for a clock cycle to complete (for all the signals to propagate correctly). More on this when we do our next project!*

After this, you can submit the project as you usually would- remember to attach the:

- `.zip` file containing your world
- screenshot of the tests passing

Don't forget to pat yourself on the back for building a way cool ALU from start to finish! Now you can go teach CMSC411 like it's nobody's business.