CE Department Project Report

Art of Engineering

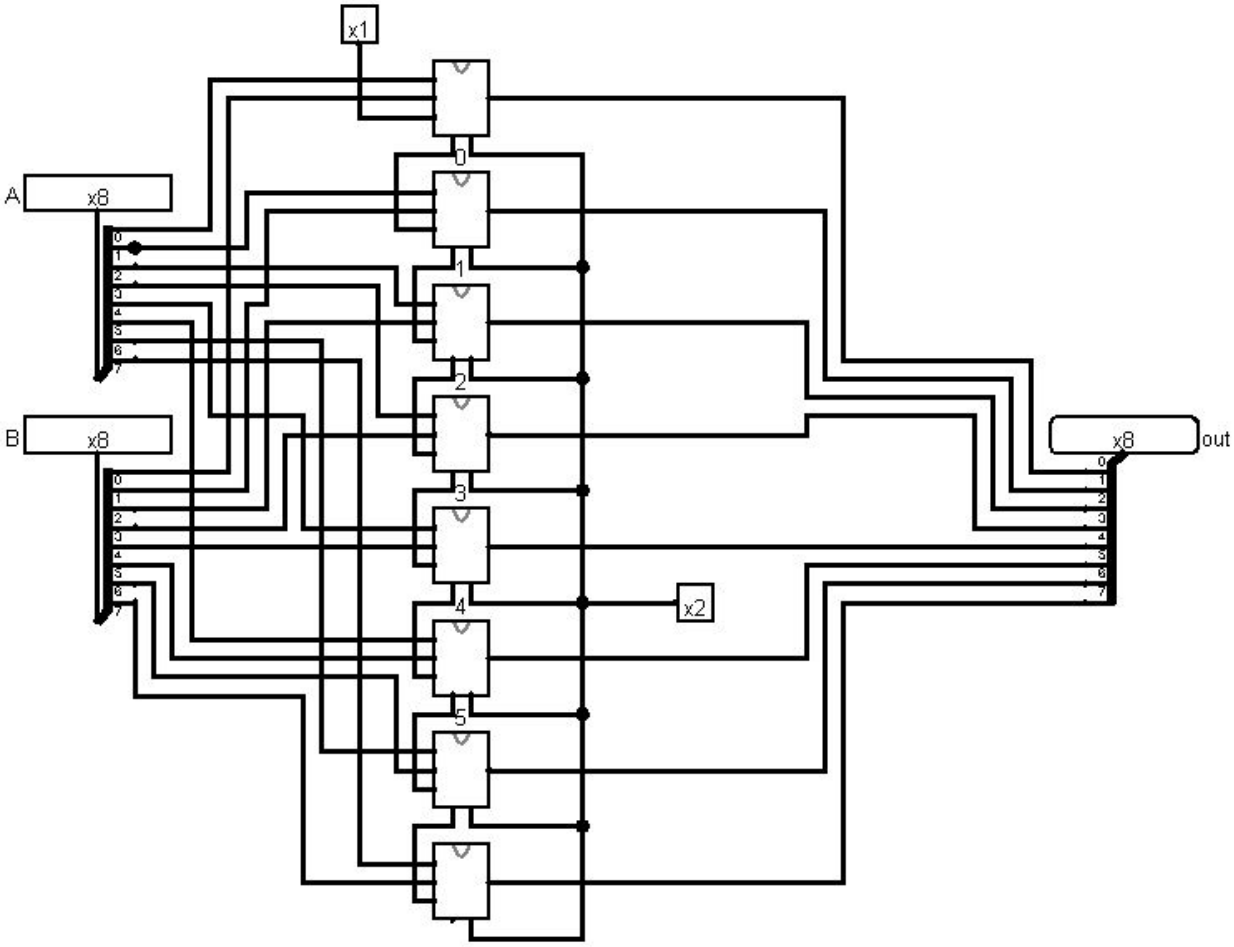
Jenny Cha

jc5329

Fall 2019

1. Top-Level ALU Design

8-bit ALU Circuit Diagram



Design Process For 8-bit ALU

To design the 8-bit ALU, I began by identifying the necessary components. I noted that I would need eight 1-bit ALUs, as well as two inputs and an output, each of these being 8-bits in width. To have a properly functioning 8-bit ALU, I first built up the individual components as much as I could to their simplest forms.

a) 1-bit ALU

Similar to the 8-bit ALU, for the 1-bit ALU I began by identifying necessary components by asking myself: what does my ALU need to accomplish? The answer was that it needed to take in two inputs and based off a control bit to denote the logic performed, return the output. The logic performed would either be an AND, OR, ADD. The AND and OR operations can be represented with logic gates, but the ADD operation requires its own circuit, a 1-bit adder.

i) 1-bit adder

The 1-bit adder was designed by first writing out truth tables for numerous gates, and then comparing the truth tables with the outputs of binary addition. I noted that for 1-bit addition without any carrying, the output matched the output for an XOR truth table (aka adding for the 2^0 place value). However, if input A =1 and input 2 =1, then A+B returns 10, which is now 2-bits. To account for this, there needed to be a carry value, which would be XOR'd with the 2^1 place value. For the carry value output, I noted that the output matched the output for a AND truth table. This is depicted in the tables below.

Adding 2 Binary Numbers Table

A	B	Output
0	0	00
1	0	01
0	1	01
1	1	10

AND Truth Table

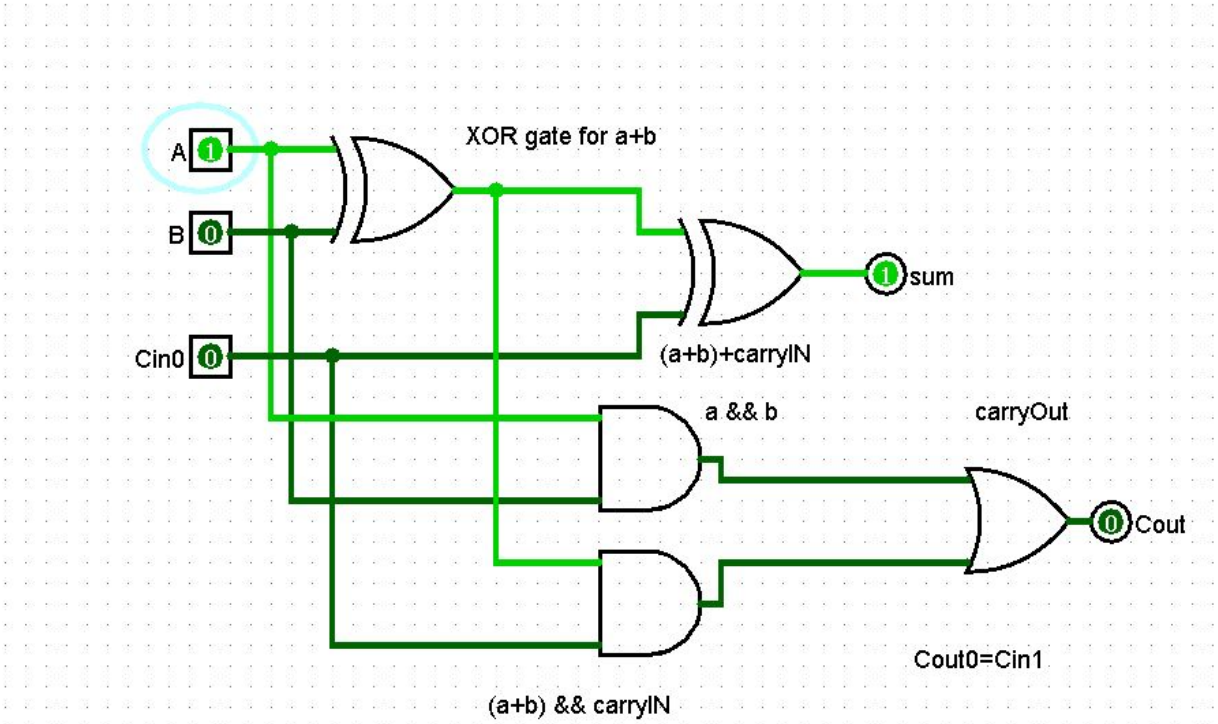
A	B	Output
0	0	0
1	0	0
0	1	0
1	1	1

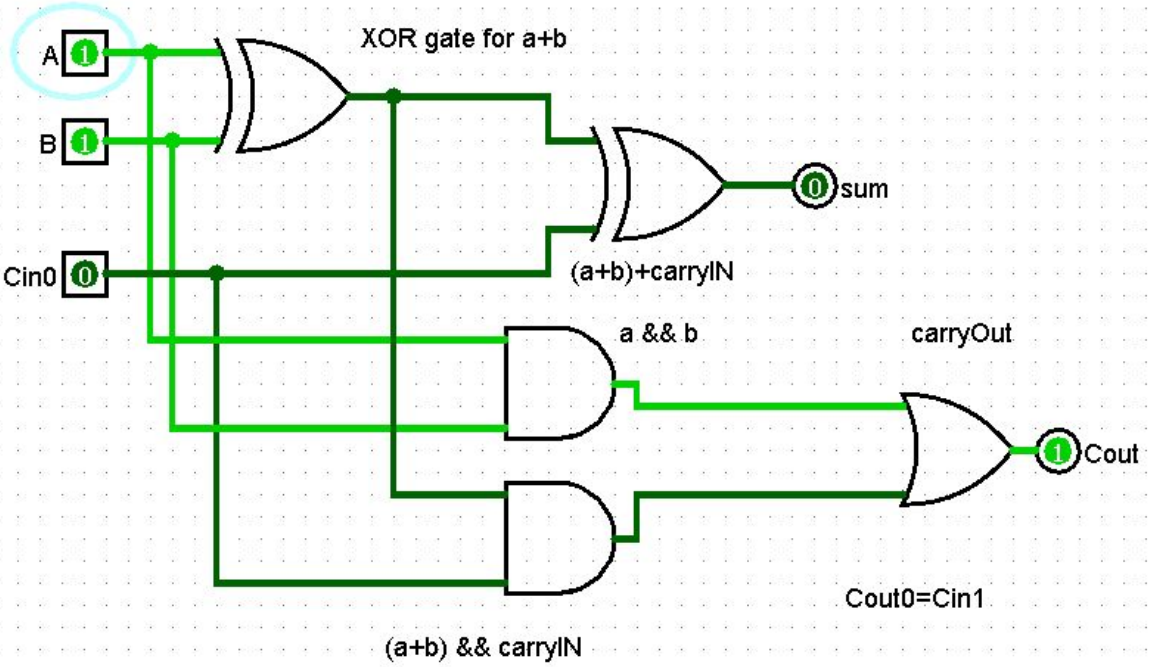
XOR Truth Table

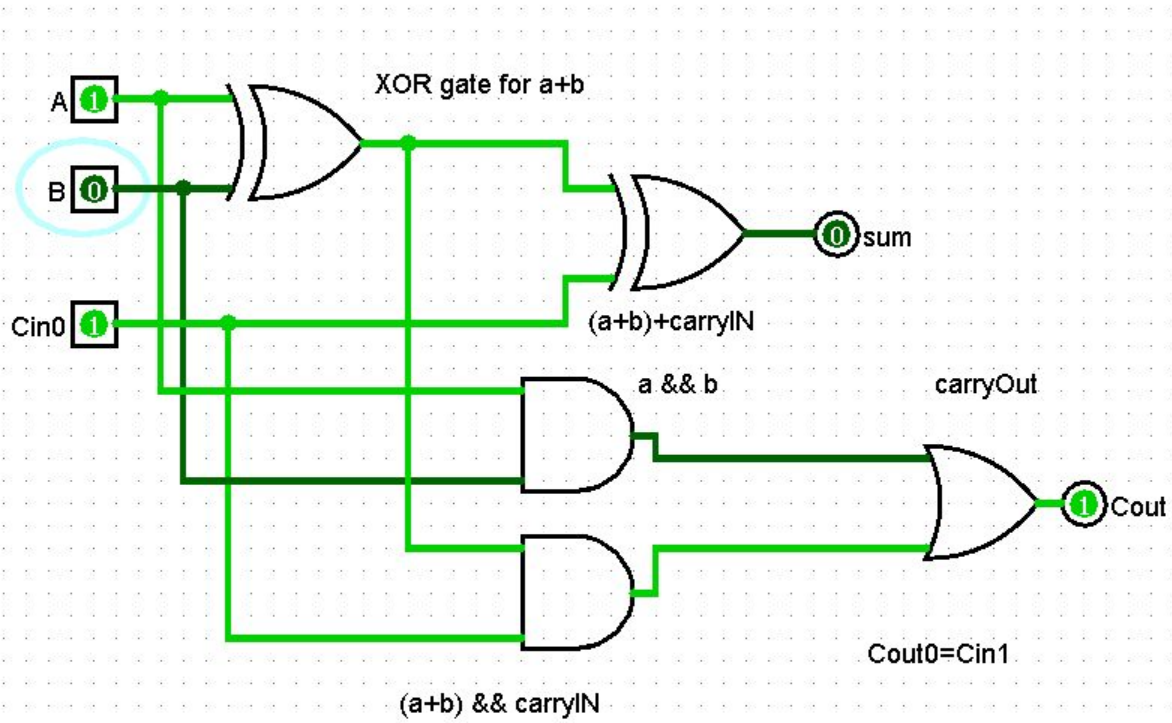
A	B	Output
0	0	0
1	0	1
0	1	1
1	1	0

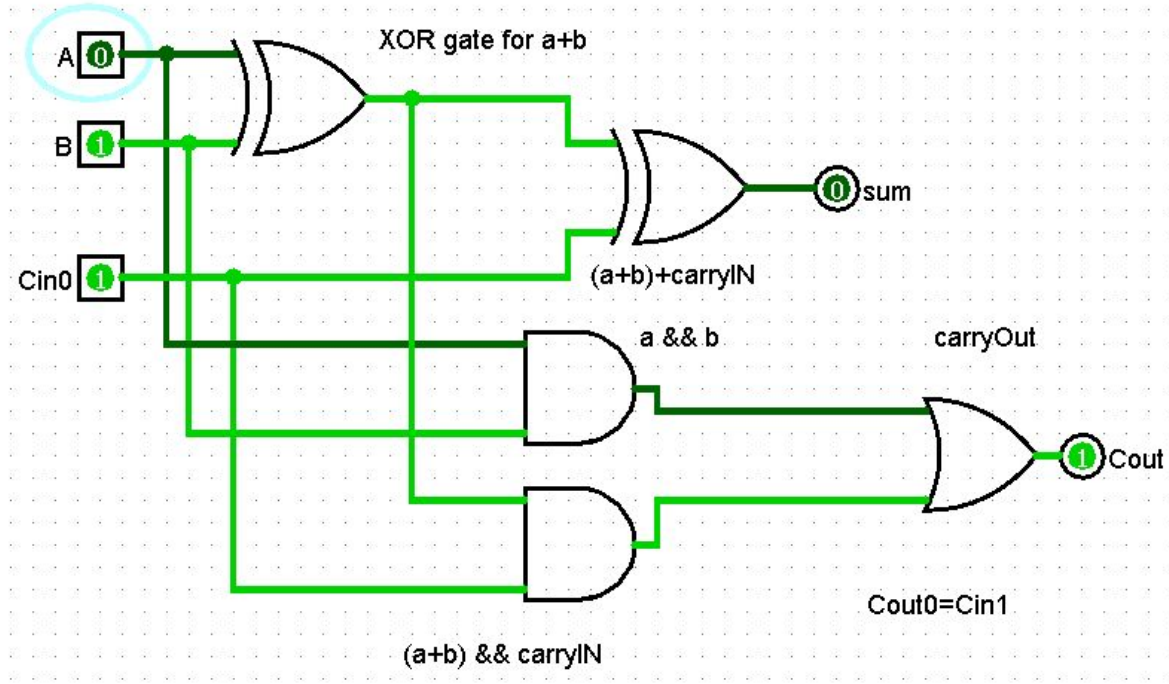
Since I need to add the carry to the other values in the same place value as it, I used another XOR gate (which can be seen as the addition gate), and I also checked if there was a carry for the next place value created by ANDing the sum (A+B) with the carry. Since the carry value is dependent on two AND gates (one that takes in A+B, carry, and another that takes in A,B), if one of them is true, then the carry exists. To do this, I used a OR gate, which gives the final carry output.

1-Bit Adder Circuit Design with Simulation





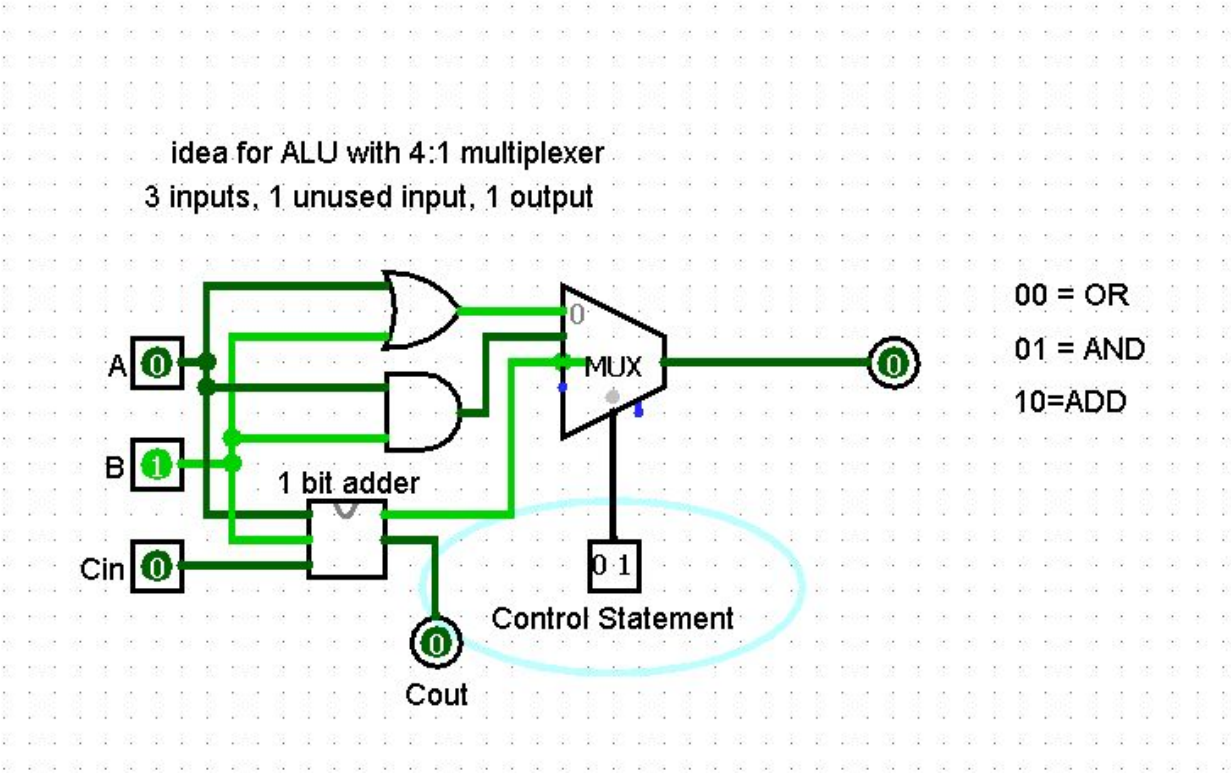

A
1
B
0
Cin0
1
XOR gate for a+b
0
sum
(a+b)+carryIN
a && b
carryOut
1
Cout
Cout0=Cin1
(a+b) && carryIN


A
0
B
1
Cin0
1
XOR gate for a+b
0
sum
(a+b)+carryIN
a && b
carryOut
1
Cout
Cout0=Cin1
(a+b) && carryIN

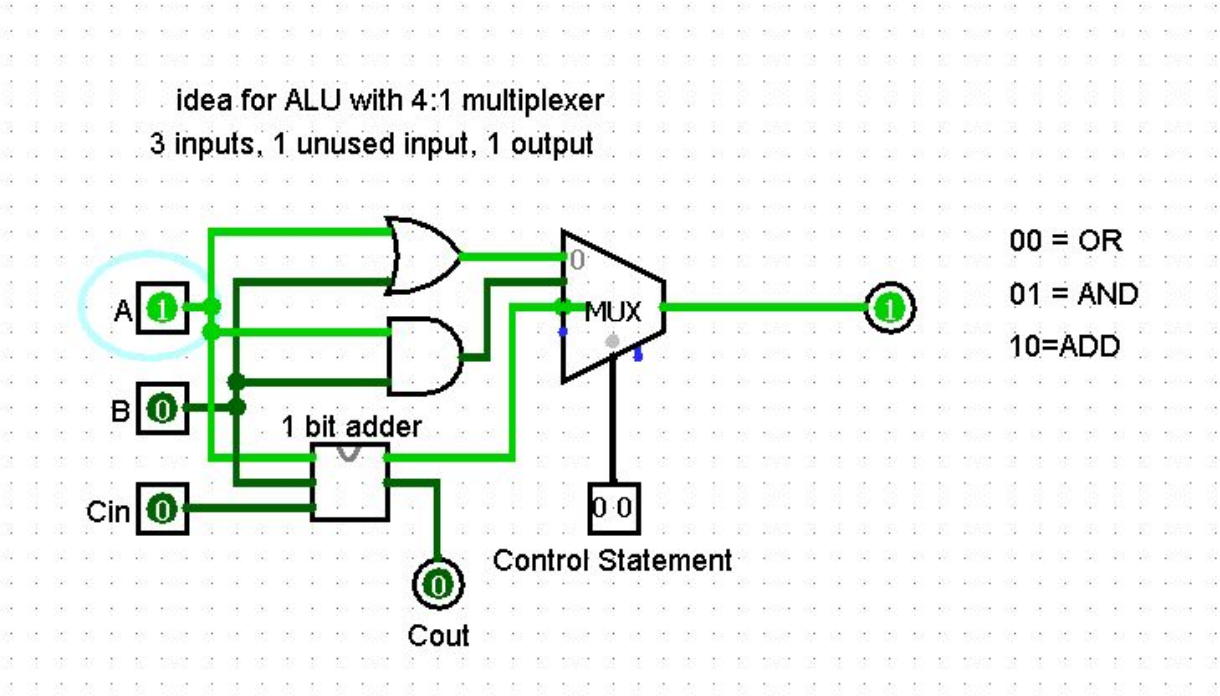
Once I ensured that the 1-bit adder had full functionality, I began on the 1-bit ALU. Based off the fact that the ALU can perform multiple operations, there needs to be a component that can take one input and based off a control input, return an output accordingly. To accomplish this, a multiplexer was implemented. The properties for the multiplexer were chosen as the following:

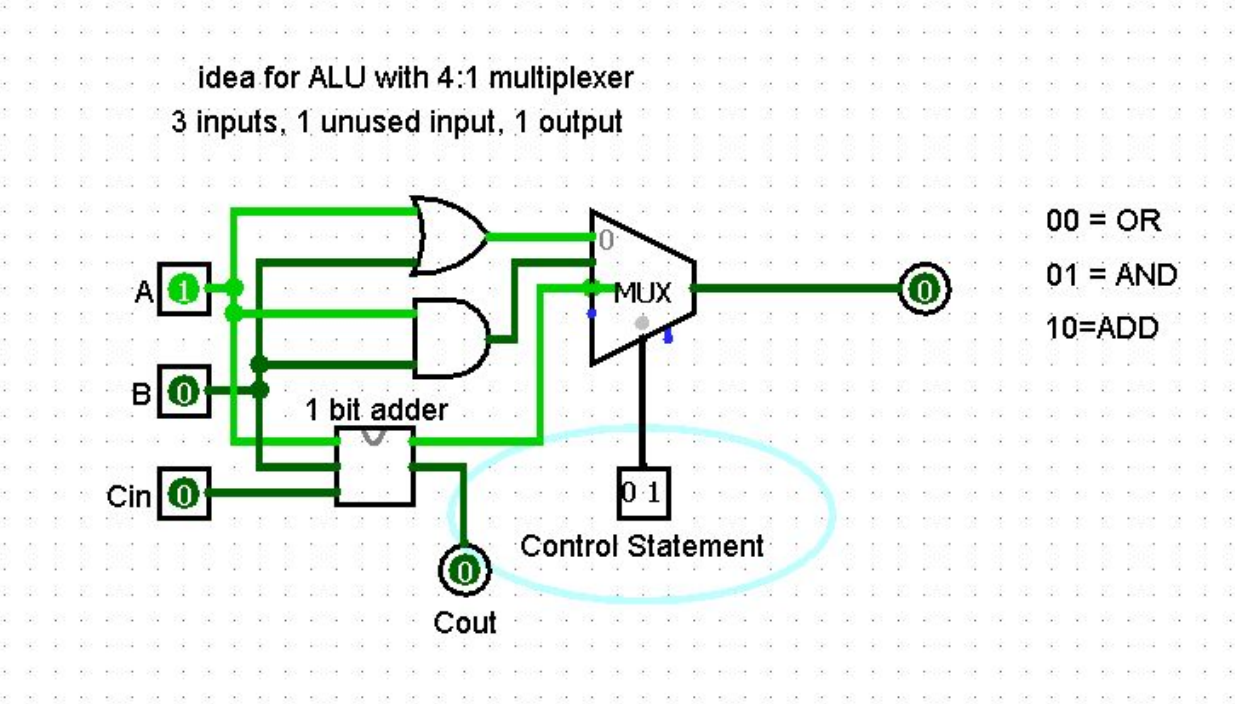
1. **Select Bits =2**
   This gives a total of 2^2=4 total operations that can be used, which is sufficient considering that the ALU is selecting between AND, OR , and ADD
2. **Data Bits = 1**
   This is because after taking in two 1-bit inputs, the multiplexer should return one 1-bit output.

The way the circuit was drawn, the OR gate's output was mapped to input 0 of multiplexer, the AND gate's output was mapped to input 1 of the multiplexer, and the ADD circuit's output was mapped to input 3 of the multiplexer. Input 4 is null because there are only 3 operations being performed; if subtract had been implemented though, its output likely would have been mapped to input 4 of the multiplexer. Thus, OR is selected by the multiplexer when the control input is 00, AND is selected by the multiplexer when the control input is 01, and ADD is selected by the multiplexer when the control input is 10. If the control input is 11, then the multiplexer will not return anything. These are shown by the screenshots below.



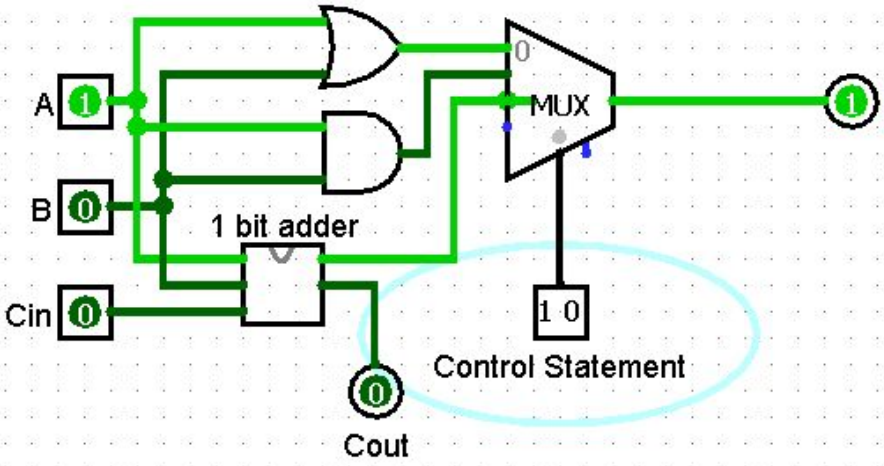
1-bit ALU Circuit Design with Simulations





idea for ALU with 4:1 multiplexer

3 inputs, 1 unused input, 1 output



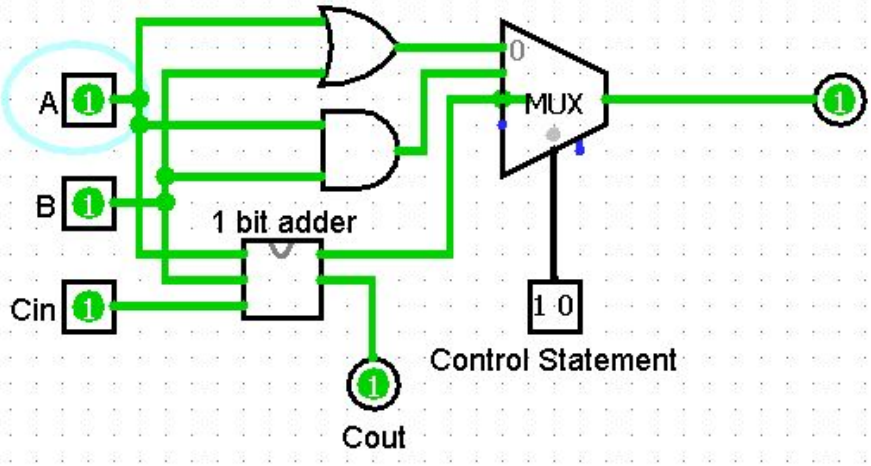
00 = OR

01 = AND

10=ADD

idea for ALU with 4:1 multiplexer

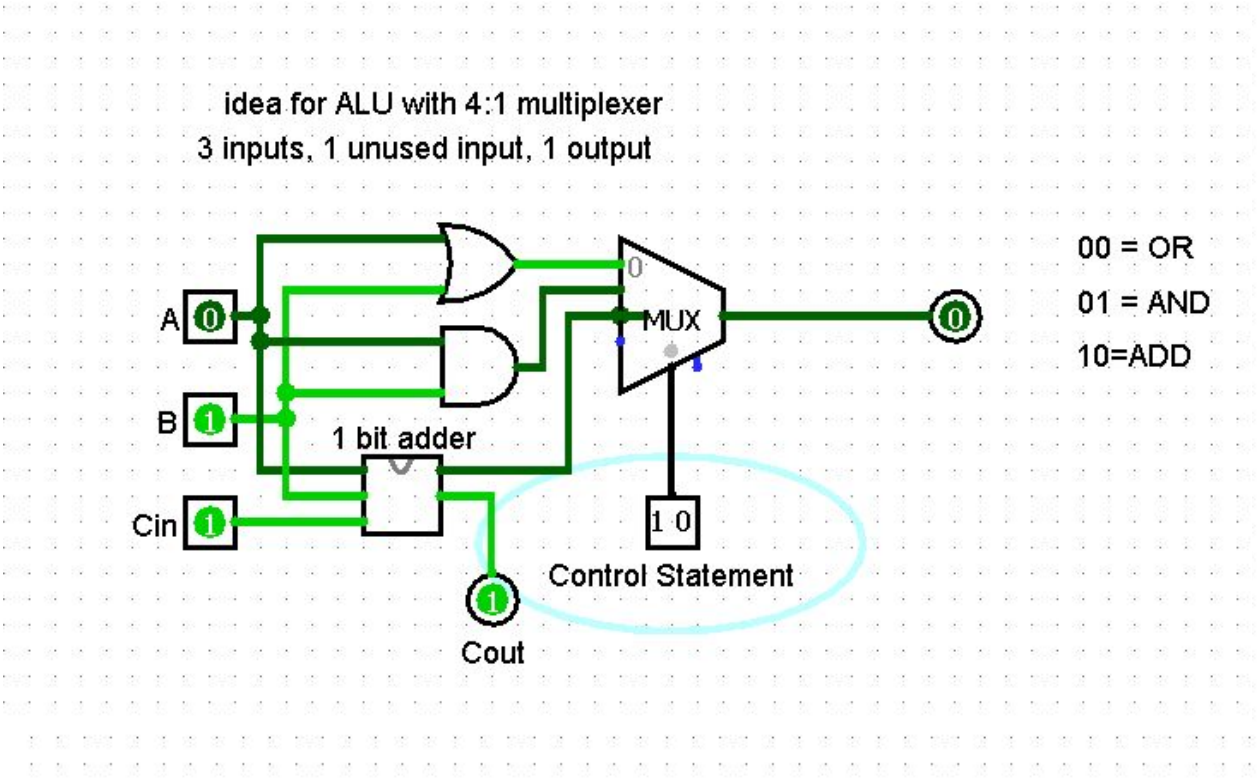
3 inputs, 1 unused input, 1 output



00 = OR

01 = AND

10=ADD



After assembling the 1-bit ALU, it was wrapped as a submodule and then implemented in the 8-bit ALU.

b) Input A and Input B

The two inputs were passed through a splitter due to the design of the 8-bit ALU. Since the 8-bit ALU is composed of individual 1-bit ALUs, the inputs that get put into these 1-bit ALUs needs to match its bit width. Thus, a splitter component was utilized to divide the 8-bit inputs into 1-bit ones, which were then passed into the respective 1-bit ALUs.

c) Output

Since the 8-bit ALUs function is to perform operations on 8-bit inputs, it would also be expected to return the output in 8-bits. Thus, I included another splitter (more like a "joiner") component which took in the 1-bit outputs calculated by each 1-bit ALU and placed them into the 8-bit output box.

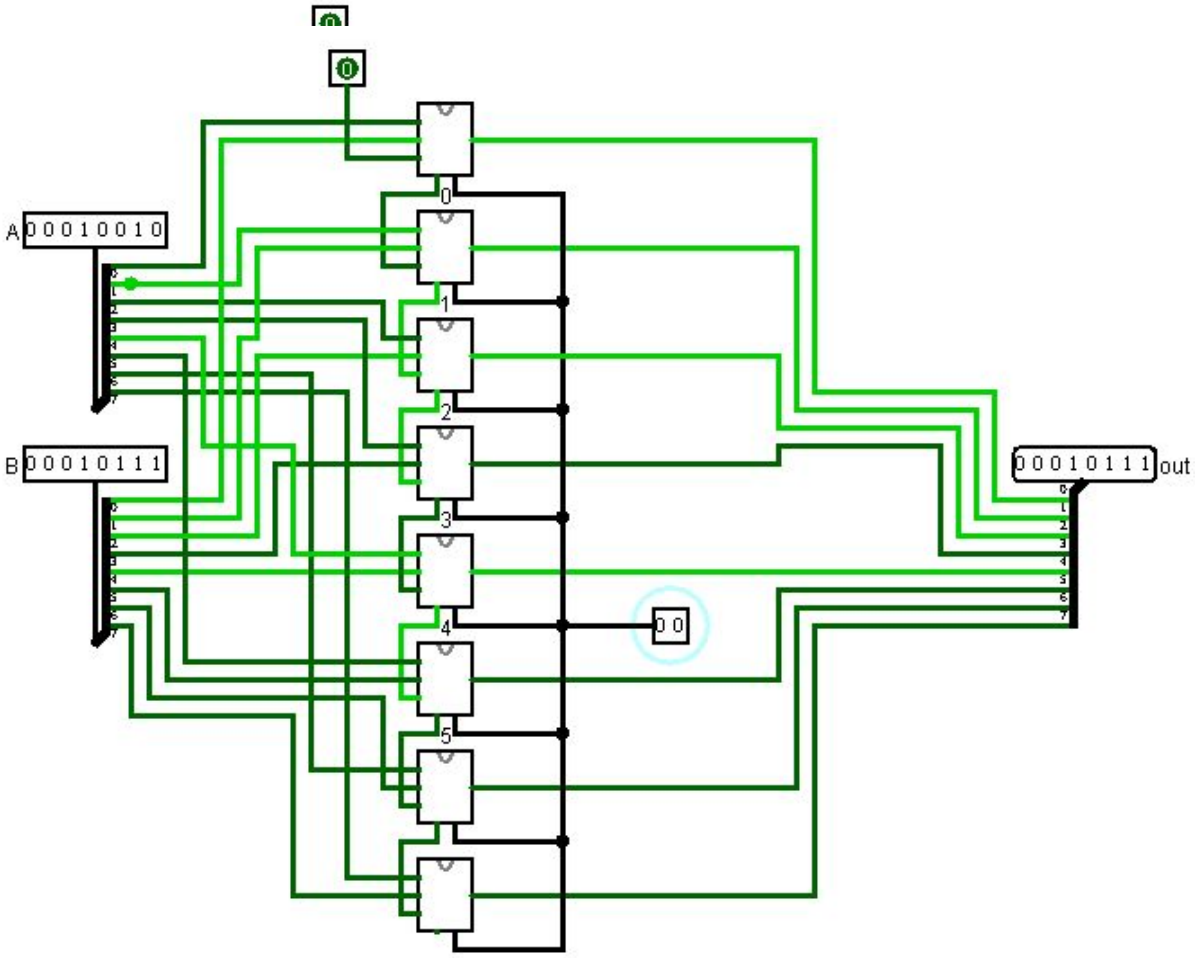
d) Carry for the 2^0 Place, Overflow, and Control Input

A 1-bit input was added to the ALU file, which represents the carry value at the start of calculations. This was left as 0, as before any calculations are done, the carry in is always equivalent to 0. A 1-bit output called overflow was added to receive the last 1-bit ALU's carry out value. A 2-bit input was passed in as the control statement, which is used by the ALU to determine what operation to perform.

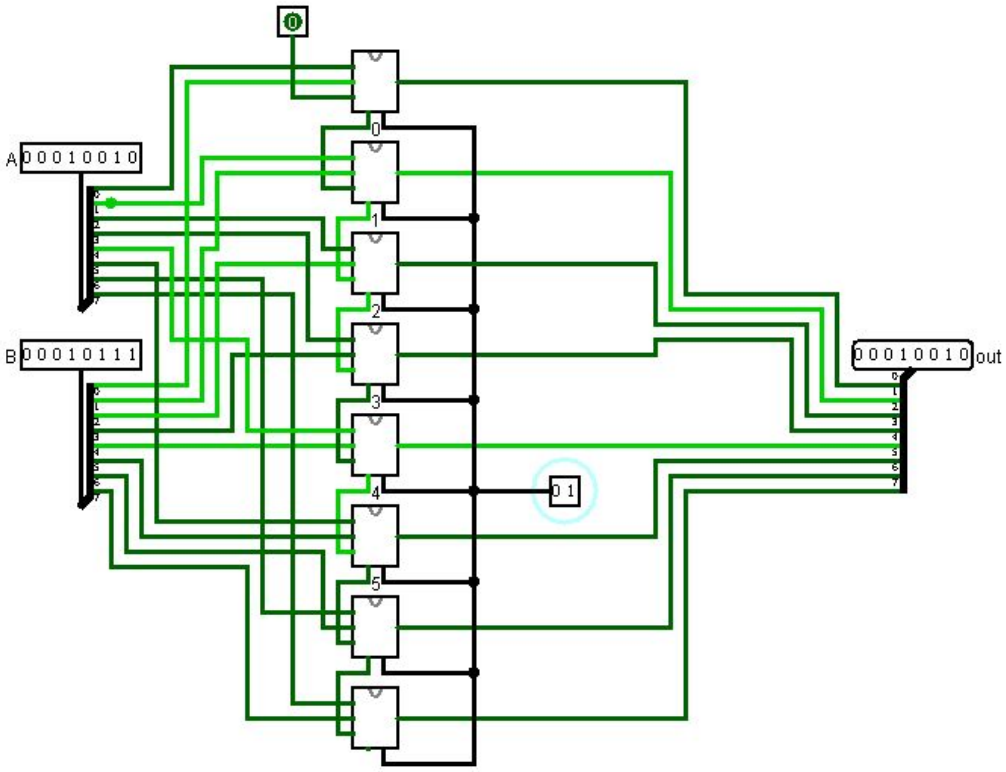
Though the 8-bit ALU is composed of the aforementioned components, the way the components were connected was crucial to ensuring that the correct output was returned, especially when adding. This led to a design where the carry output of the previous 1-bit ALU was connected to the carry input of the next 1-bit ALU to ensure that the binary addition was calculated with the carry each time. Furthermore, each 1-bit ALU was connected to the same control statement because all the individual components should perform that same operation that is defined for the overall 8-bit ALU.

8-Bit ALU Circuit Design with Simulations

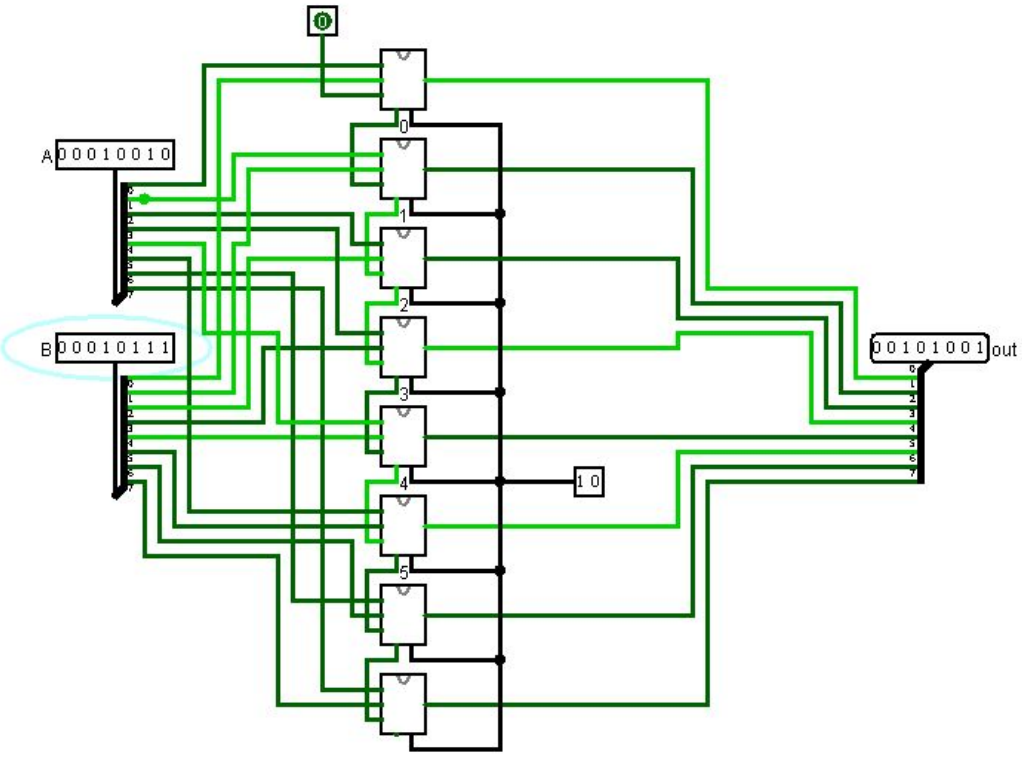
OR Simulation



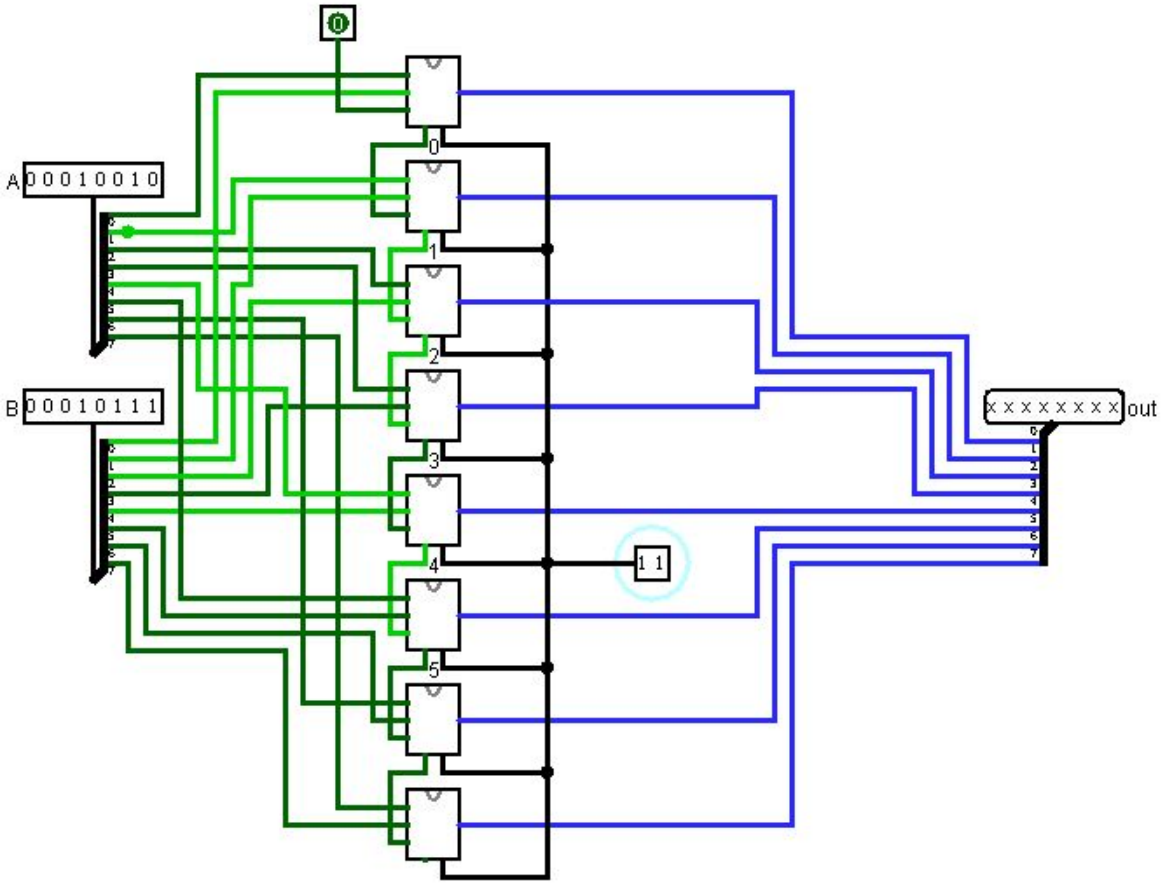
AND Simulation



ADD Simulation



Invalid Control Input Simulation



2. Memory Settings

The memory settings could be broken down into two kinds: instruction memory and data memory.

a) **Instruction Memory**

The instruction memory was created using the built-in ROM component in LogiSim. This design choice was made considering that the instructions are predetermined by users and that there is no need to write data into the instruction memory, The address bit width was selected as 8-bits, since that would make it easier for the 8-bit ALU to perform calculations on the memory address and ensure that other components wouldn't need to be added to manipulate another bit size into 8 bits.

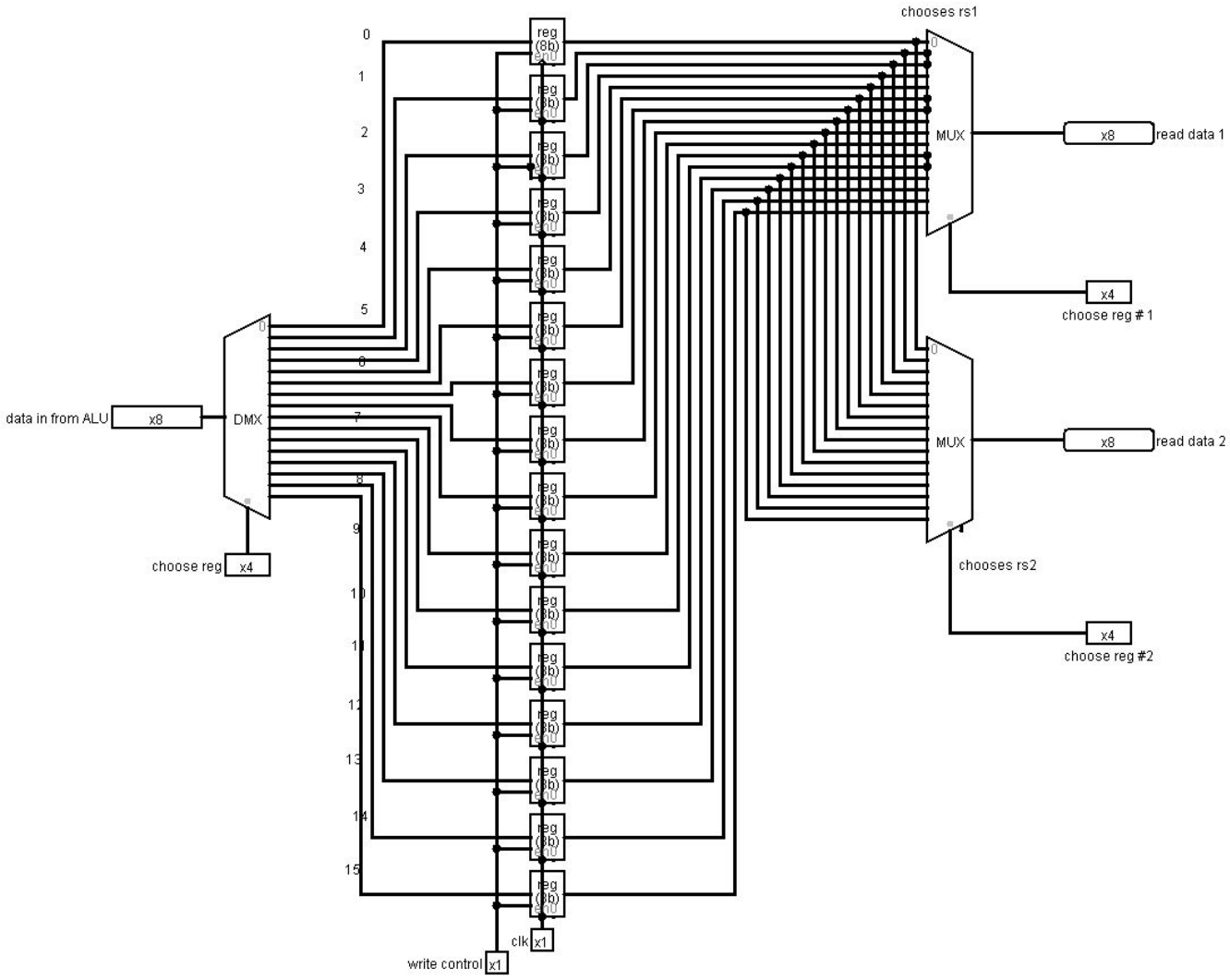
b) **Data Memory**

The data memory was created using the built-in RAM component in LogiSim based off the fact that the data memory needs to be read from and written to. The address bit width and data size bit width were set as 8 bits because the data from the register is 8 bits. Thus, to pass an address or data into the register, the data memory also needs to produce an

output that is 8-bits. The data memory was also selected to have separate load and store ports, since that would make it easier to determine whether or not a read or write will be performed, as well as separates the input address from the data input.

3. Register File

Register File Circuit Diagram



Design Process for Register File

The logic behind the register file was broken down into the following:

a) Number of Registers

The number of registers needed was chosen to be 16; since 4-bits denote the register used, there are 2^4 possible combinations = 16 registers necessary. These registers are shown in the picture above and are connected to two multiplexers and a demultiplexer. The clock signals of the registers are connected together to one input (which will be a block in the main circuit) to ensure that the registers all run at the same clock cycles.

Furthermore, the enable pins of all the registers are connected to one input for a similar reason as the clock input: to ensure that all registers write when they are given the proper input.

b) Recognizing rs1 and rs2

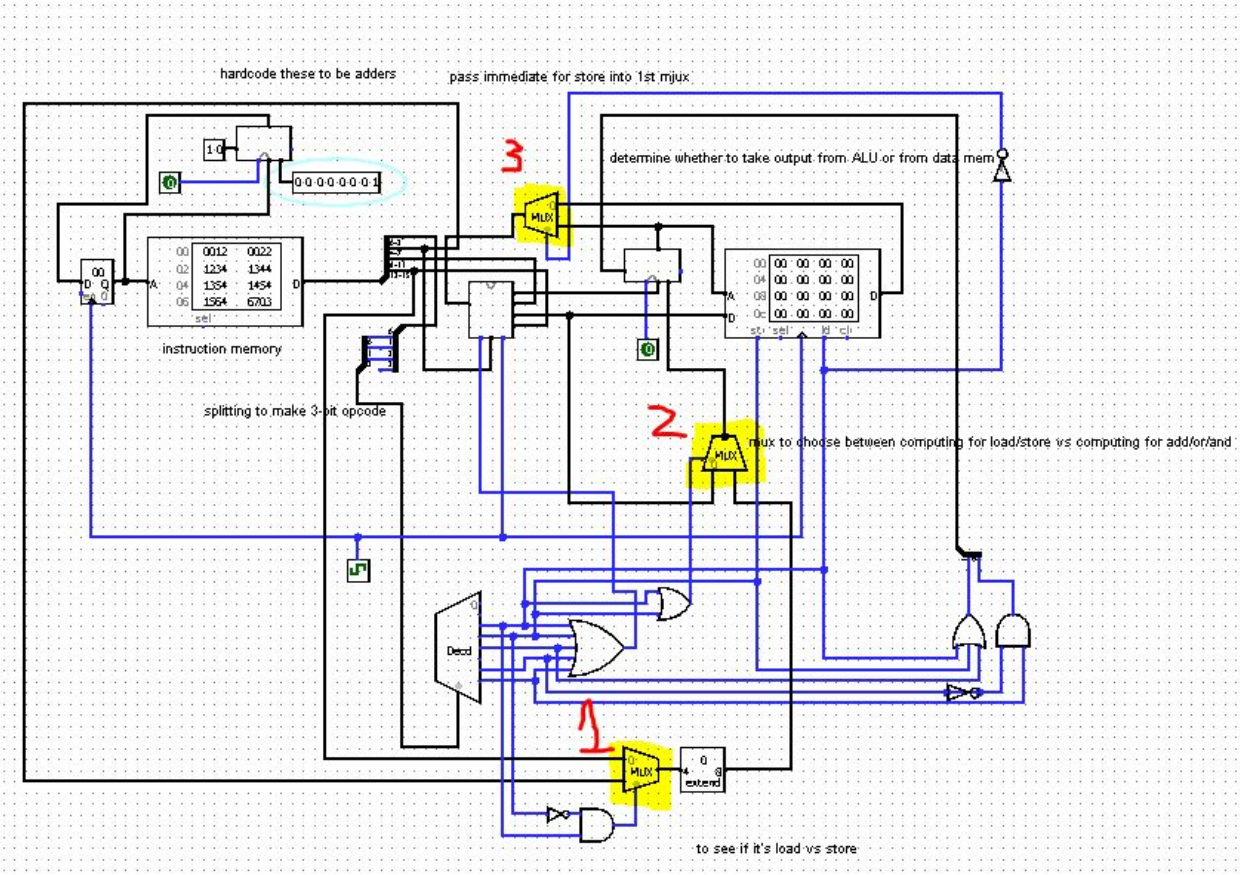
To determine which register to pull data from, two multiplexers were used. One outputs the data given rs1(instructions [11:8]) as the control input, and the other which outputs the data given rs2 (instructions [15:12]) as the control input.

c) Choosing Registers for Writing

In addition to outputting data, the register file also needs to be able to write data into its registers. To interpret which register it should write to, a demultiplexer is utilized which takes in the output from the ALU or data memory and then, using a control input, determines which register the data should output to. This control input is connected to rd, which is denoted by the instructions as [4:7].

4. Datapath

The datapath was composed of the instruction memory, the register file, the ALU, and the data memory, along with the interface between these components. The datapath can be broken down to examine the interaction between certain components.



a) **Instruction Memory and Register File**

To feed between the instruction memory and register file, it was necessary to use a slitter. This is due to the fact that the instructions come in 16-bit pieces, but the register addresses are denoted by 4-bits, so a splitter is used to split the instruction address into 4 pieces [0:3], [4:7], [8:11], [12:15], which correspond to different inputs in the register file. Since rs1 is always fed into the register, [8:11] is directly connected to the control input labelled rs1 (choose reg #1).

b) **Multiplexer #1**

This multiplexer is used to determine whether the opcode is a ld or a sd, which denotes the position of the immediate as [4:7] (store) or as [12:15] (load). The control input for multiplexer #1 is set by ANDing output 2 and output 3 from the decoder (explained in Section 5. Control Logic), which correspond to the ld and sd opcodes. To ensure that only one of the outputs can be true at one time, output 3 is fed through a NOT gate. Thus, when output 3 is 1, then it is equal to 0, which is ANDed with output 2's default value 0. When output 2 is equal to 1, it is ANDed with output 3's default value. Thus, the output for multiplexer #1 is either the immediate for a store operation, or the immediate for a load operation.

c) **Multiplexer #2**

This multiplexer is to determine whether the ALU's second input should come from rs2 or from the immediate, which is taken from the output of multiplexer #1. The control input for multiplexer #2 is based on whether or not there is a load or store operation taking place. The logic is that if the operation is not a store or load, then the operation will use rs2, not an immediate. Since rs1 corresponds to input 0 of the multiplexer, then the multiplexer will default to outputting the value from rs2. An OR gate is connected to outputs 2 and 3, which correspond to a load or store operation. If either none of these is true, then the control input is 1, which corresponds to the immediate from multiplexer #1.

d) **Multiplexer #3**

This multiplexer is used to determine whether the register file is taking in data from the ALU or from the data memory. The control input for multiplexer #3 uses a NOT gate running from output 2 because as long as the operation is not a load, the register file reads the output from the ALU. Thus, the inputs of the multiplexer are the output from the ALU and the output from the data memory.

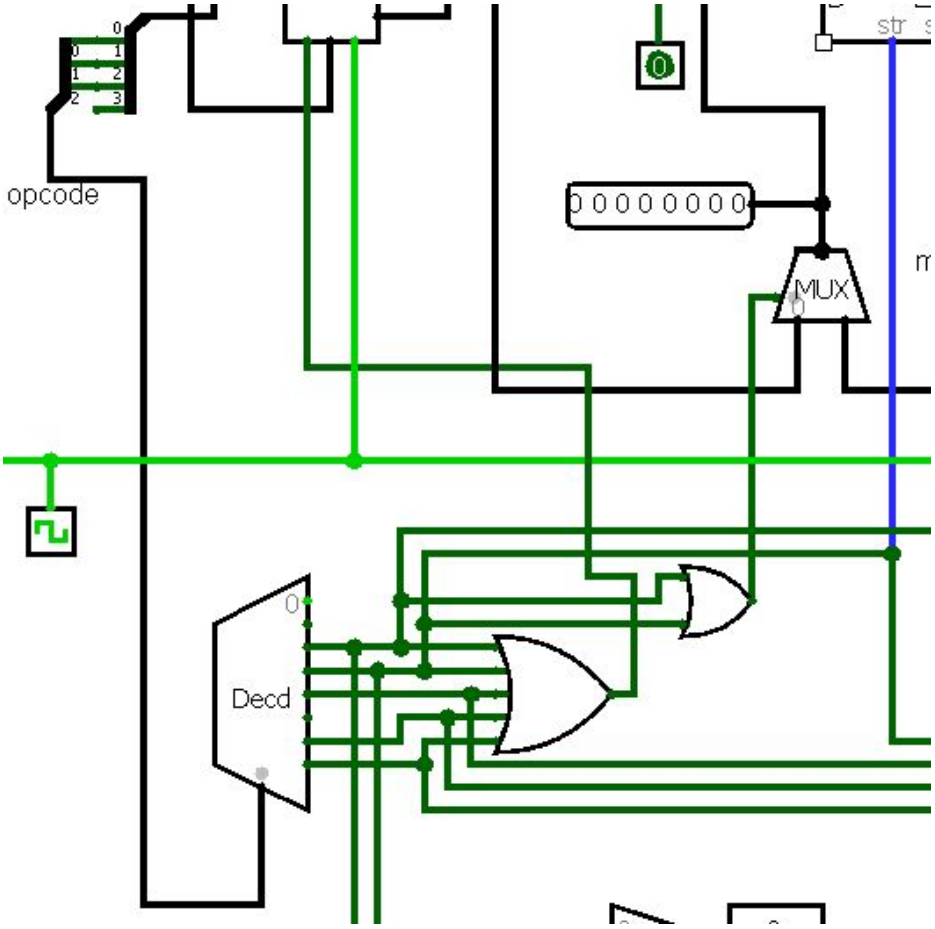
e) **Proposed Datapath for beq Operation**

Though I ran out of time to implement the beq operation, I did consider how I would perform the beq operation. I decided that this could be accomplished by using another multiplexer, which would have two inputs: 00000001 (to go to the next instruction memory address) and the value of the immediate, which is represented by [7:4]. The control input for the multiplexer would have been the beq operation result, which could

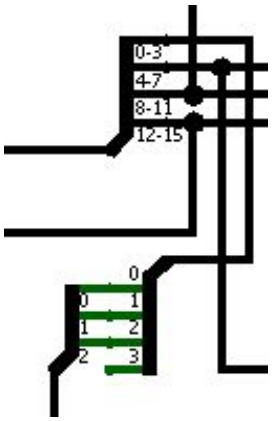
be accomplished by subtracting the two values that are being compared. If the two values are equal to 0 when subtracted, then the values are considered equal, and beq becomes 1, which causes the multiplexer to output the immediate.

5. Control Logic

Image of Decoder Taking in 3-bit opcode and splitting it into 8 outputs.



Decoding OpCode



In order to interpret the opcode, a decoder was utilized. First, the leading 0 of the op-code was dropped by using a splitter element into a splitter. I split the 4-bit op-code into individual 1-bits and then, using another splitter but as a joiner, collected the 1-bits into a 3-bit opcode. This opcode was fed into a decoder, taking advantage of the fact that each command corresponds to a decimal number from 0-7.This can be seen in the image to the right. The op-code is denoted by [0:3], which is

connected to the splitter element. The split up code is passed into another splitter, which is inverted to act as a joiner, except output 3, which holds the leading 0.

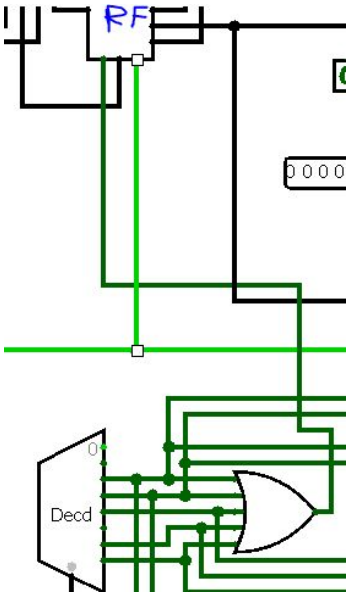
Table Mapping Opcode and Instruction Type to Decoder Output

Op Code	Corresponding Output of Decoder	Operation
0000	0	nop
0001	1	beq
0010	2	ld
0011	3	sd
0100	4	add
0101	5	sub
0110	6	and
0111	7	or

The outputs from the decoder were used to determine the signals for loading, storing, the operation performed by the ALU, and which part of the opcode is considered the immediate.

a) **Loading & Storing**
Given that the opcodes for loading (0010) and storing (0011) output to 1 only when their opcode is selected, then these outputs can be connected directly to the respective pins in the data memory.

b) **Write Control**
The logic for the write control was driven by the question: which operations needs to write into the register? The operations that need to do so are add, subtract, and, or, and load, which corresponds to output pins 4, 5, 6, 7, 2. As long as one of these are true, then the write control should be turned on. Thus, these operations were bundled up with an OR gate, whose output was fed to the write control input in the register. This is depicted in the picture to the right. However, since I did not include a subtract operation in my ALU, I simply chose not to connect output pin 5 to anything.



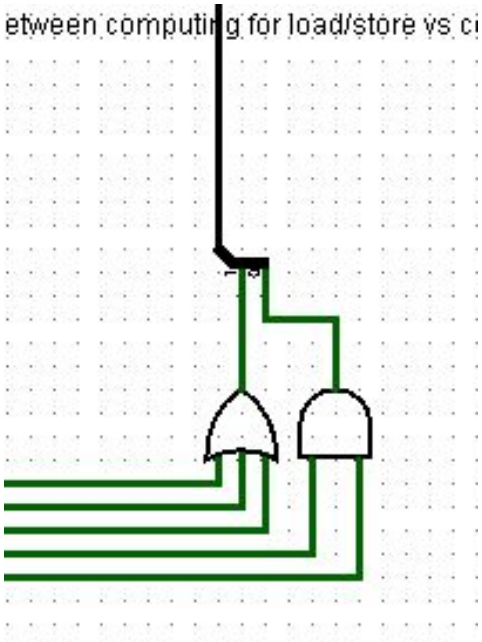
c) Converting Decoder Output into 2-bit Control Input for ALU

Since the decoder provided 1-bit outputs, I needed to find a way to make my outputs into 2-inputs, which matches the control input for the ALU. I began by creating a table of what the ALU state should be set to for each command, as well as listing out the decoder lines for easy interpretation.

ALU Control	Opcode Instruction	Decoder Output
00	OR	7
01	AND	6
10	ADD	4
10	LD	2
10	SD	3

Logic using AND gate Logic using OR gate

From the table, I realized that I could group up the add, load, and store decoder outputs. The reason that the load and store operations both need the ALU to be set as an adder is that to perform the operations, an immediate is added to the data memory address.If any of these are true, then for the 2^1 place of the ALU control input should be equal to 1. On the other hand, the OR and AND opcodes utilize the 2^0 place of the ALU control input. To design the logic for the OR and AND opcodes, I thought about how I could ensure that the input would only be true in one case. To do this, I decided to use an AND gate, which only holds true when both inputs are the same. However, since both inputs are not going to be the same, I manipulated output 7, the output for OR by using a NOT gate (not shown in image). This makes it so that if only AND is true, then output 6 will produce a 1, which is ANDed with output 7's default value, 1. The same is true vice versa: if OR is true, then output 7 will produce a 0, which is ANDed with out 6's default value, 0. However, these gates each only produce 1-bit outputs. Thus, I used an inverted splitter (joiner) to collect the gates two outputs and then combine them into a 2-bit input, which serves as the ALU's control input.

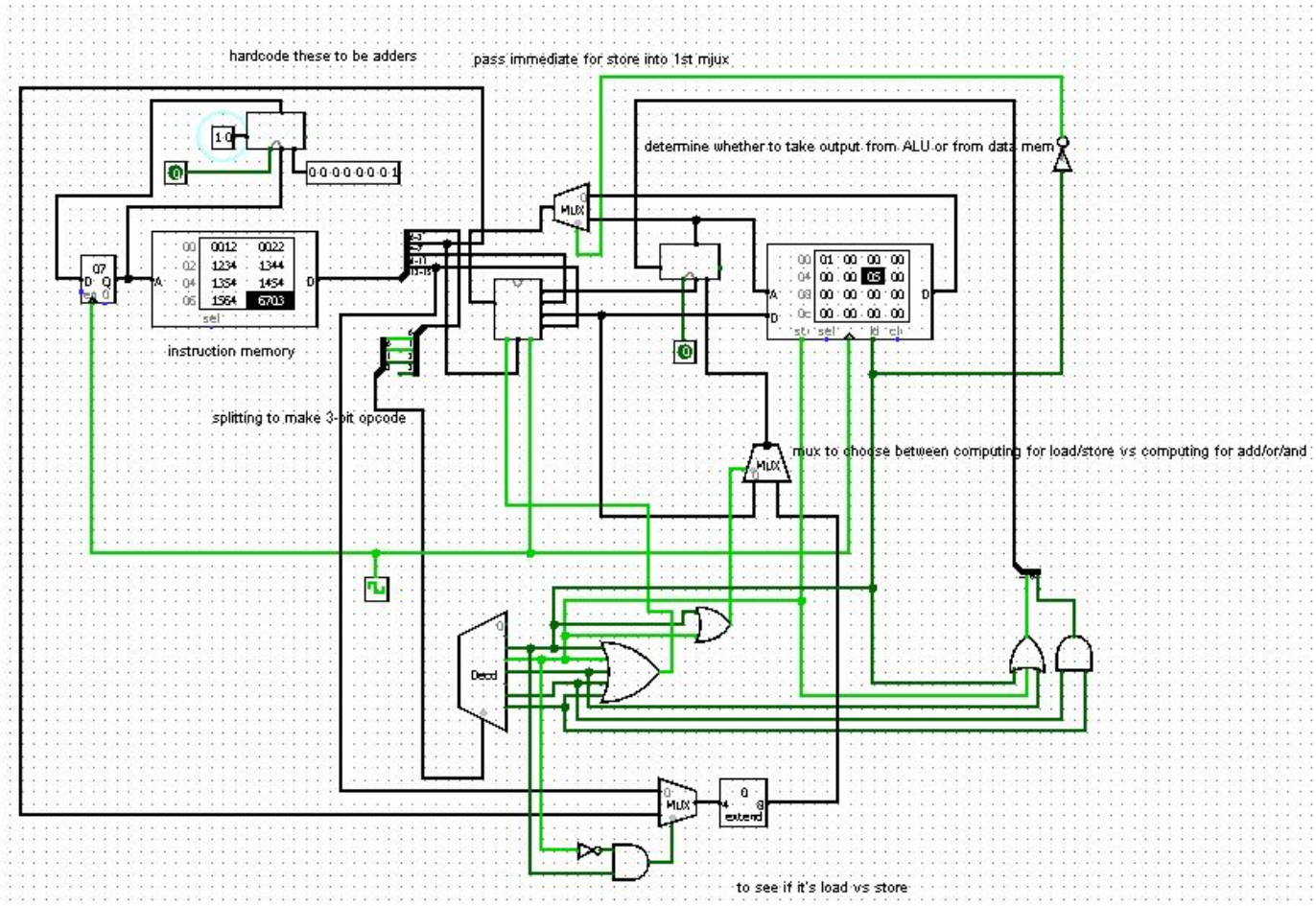


6. Programming

Instruction (in binary)	Hex Code	Description
0000 0000 0001 0010	0012	ld x1, 0(x0)
0000 0000 0010 0010	0022	ld x2, 0(x0)
0001 0010 0011 0100	1234	add x3, x2, x1
0001 0011 0100 0100	1344	add x4, x3, x1
0001 0010 0101 0100	1454	add x5, x4, x1
0001 0101 0110 0100	1564	add x6, x5, x1
0110 0111 0000 0011	6603	store x6, 0(x6)

This program counts to 5 and then stores the value into data address memory 6. To program, I began first by writing the descriptions for the instructions, as shown in the description column of the table. Since there are no values in the register, I begin by loading two registers with a value I preset in the data memory. Once I do that, I use one of the registers as a constant to add to my other registers, which I picked as register 1 containing the value 1. Once I decided that my logic in terms of the descriptions was sound, I translated the instructions into their binary representation based off the 16-bit instruction format provided. Finally, I converted the binary code into its hex code representation.

7. Simulation



This is a simulation of the code written in section 6, Programming. The instructions were preloaded into the instruction memory, and the value, 1, was preselected as address 0 of the data memory. As shown in the screenshot above, the instruction memory is incremented by a counter, and once it reaches the instruction for storing (hex code 6703), it stores the value of register 6, 05, into data memory address 6.

8. Reflection

The CS/CE Project was extremely challenging, which was what made it so rewarding. The data path is not very intuitive which required a lot of writing components out on paper to understand how they connected. I personally found the data path the hardest part because each component by itself is somewhat intuitive to understand due to its seemingly isolated function, but connecting each component definitely requires a lot more logic than I expected. In addition, I had a hard time distinguishing between the address bits and data bits in terms of the register file, because I didn't realize at first how the register indiscriminately handles both addresses and values, since both are simply represented by data. From this, I realized that I still was not

approaching the components with the right level of abstraction; I was assuming that my register was preprogrammed to determine whether the data it received was considered an address or data, when in reality, all data is just data. This gave me a newfound sense of respect towards those who work to design the computers that we use, especially considering the amount of information that our computer process.

I personally found it difficult to keep track of what was what, or what data was meant to be where, and I found myself writing out a page of notes during the office hours I attended. Furthermore, it was also intriguing to see that there are multiple ways to implement the microprocessor datapath. For example, during one of the office hours I attended, it was explained that the load operation could be implemented with a 3-input AND gate and 2 NOT gates, which was different from how I used a decoder. This made me consider my design in a more flexible way and also taught me to be more aware of gates and how I can manipulate them to generate a certain result. In high-level programming languages, there is no need to do so, because all the gates can be called by a few keystrokes. This also ties into the first part of the CE project, where a 1-bit adder was assembled. Similar to using logic statements, addition is generated in programming languages simply by one keystroke. However, designing the 1-bit adder showed me the importance of gates, and honestly was extremely fascinating and humbling (in the sense that I should be grateful that other people have already taken care of this level of abstraction).

Designing a simple counter code required a lot of effort due to conversions between assembly language into its binary representation into its hex code. I imagined doing the assembly language code on a much larger scale, which again, gave me a deeper appreciation for high-level programming languages, where I could write a counter program in a couple lines, utilizing built-in functions. This project has definitely opened my eyes to the beauty and grit that goes into computer engineering, as well as how interconnected the discipline is with computer science. Before, I simply assumed that computer engineering was just computer science with some hardware added in, but now I realize that there is definitely a distinction. Though I am not sure whether I would consider switching into computer engineering, this project has definitely challenged my preconceptions of the discipline and opened my eyes to its appeal.