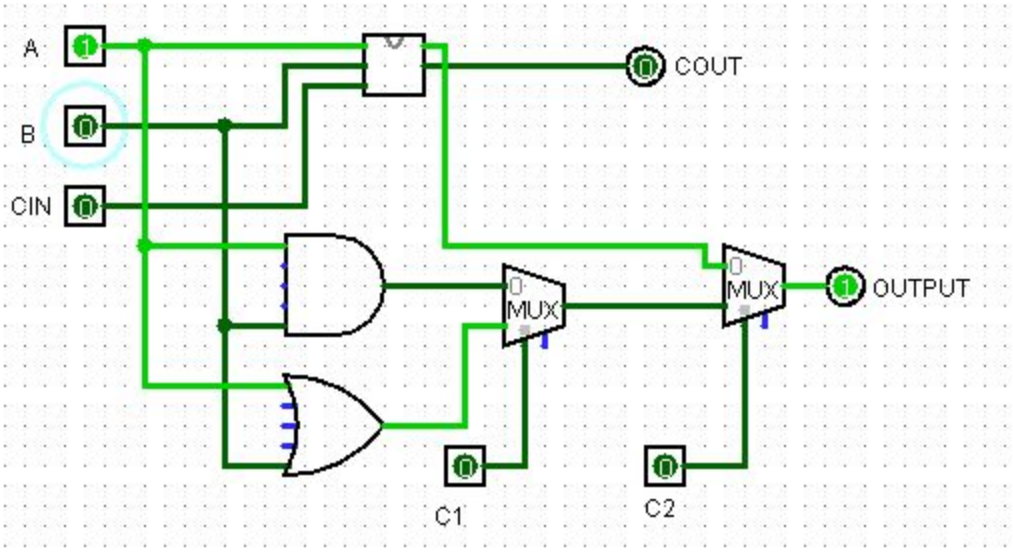
Daniel Lee
dsl2167
CS/CE Department Project

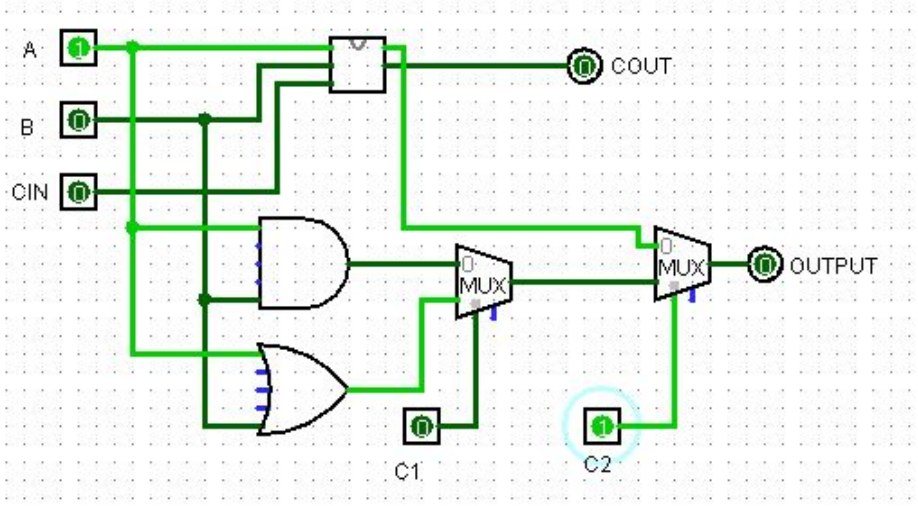
4.1. Top Level ALU Design
1-bit ALU:

C1	C2	OUTPUT
0	0	Adder
0	1	AND
1	0	Adder
1	1	OR

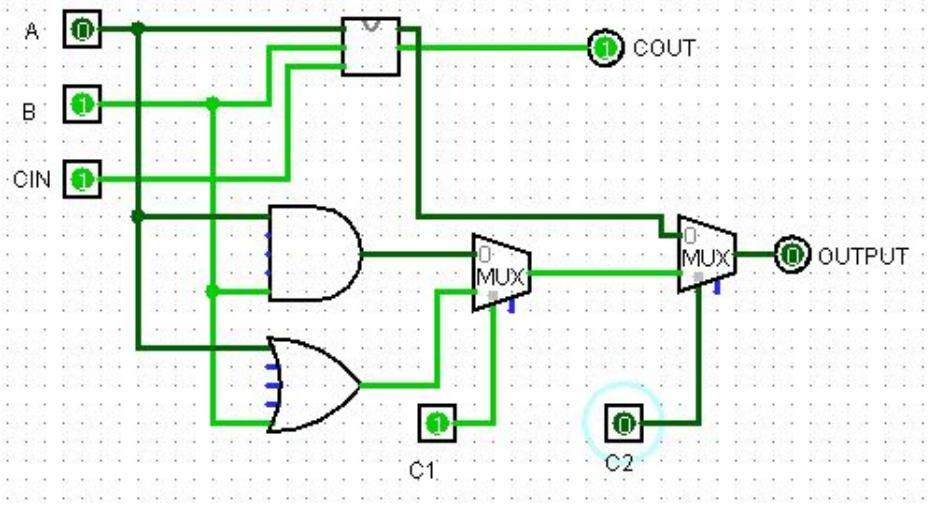
C1 = 0 / C2 = 0: Adder



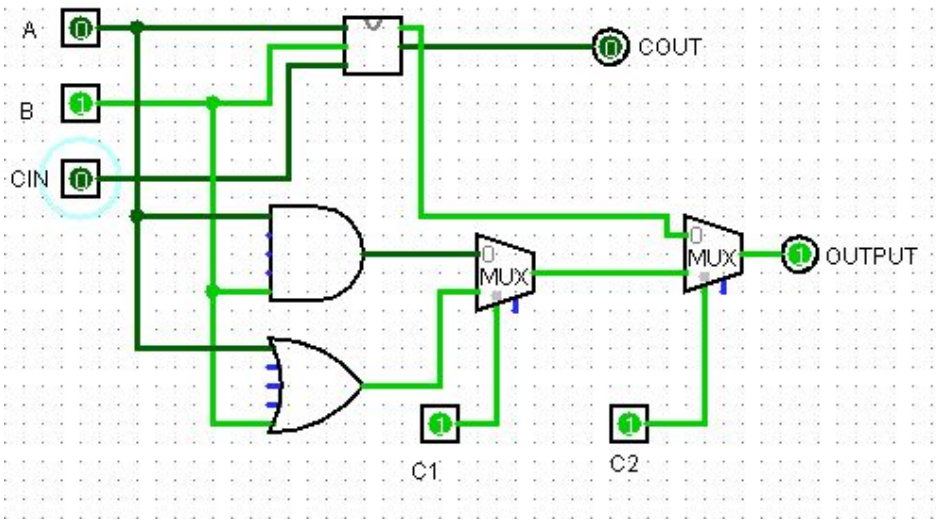
C1 = 0 / C2 = 1: AND



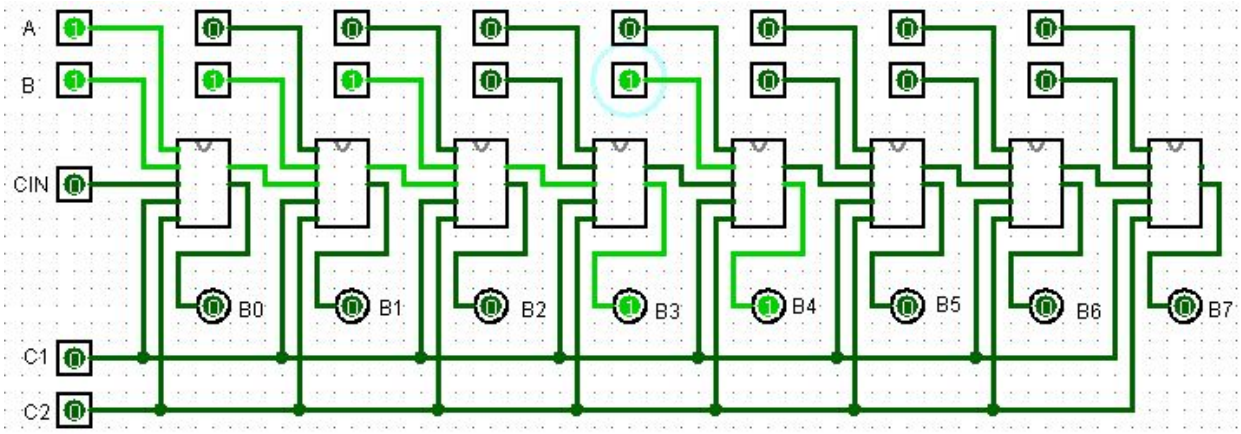
C1 = 1 / C2 = 0: Adder



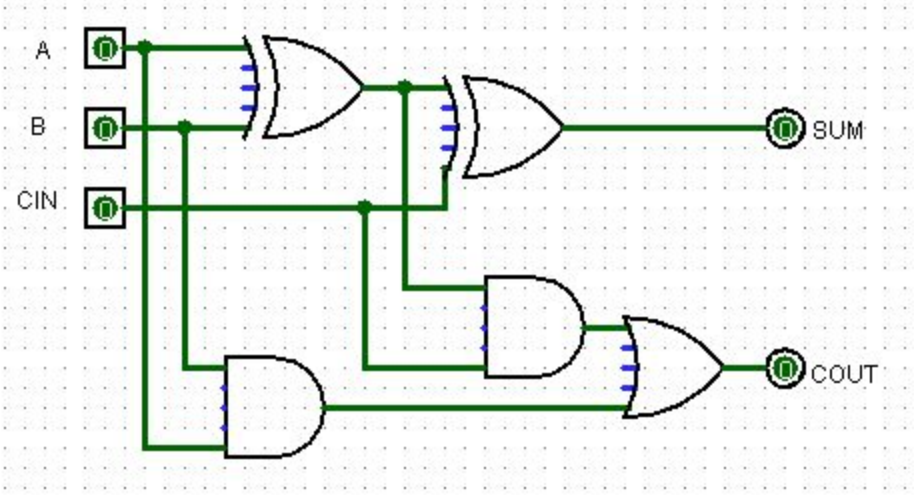
C1 = 1 / C2 = 1: OR



8-bit ALU Example:



Adder Used in ALU:



An ALU must have operations to AND, OR, and add. The first two operations can be done through gates, but adding must be done by an adder. The adder was built by first making a table of possible outcomes:

A	B	CIN	SUM	COUT
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1

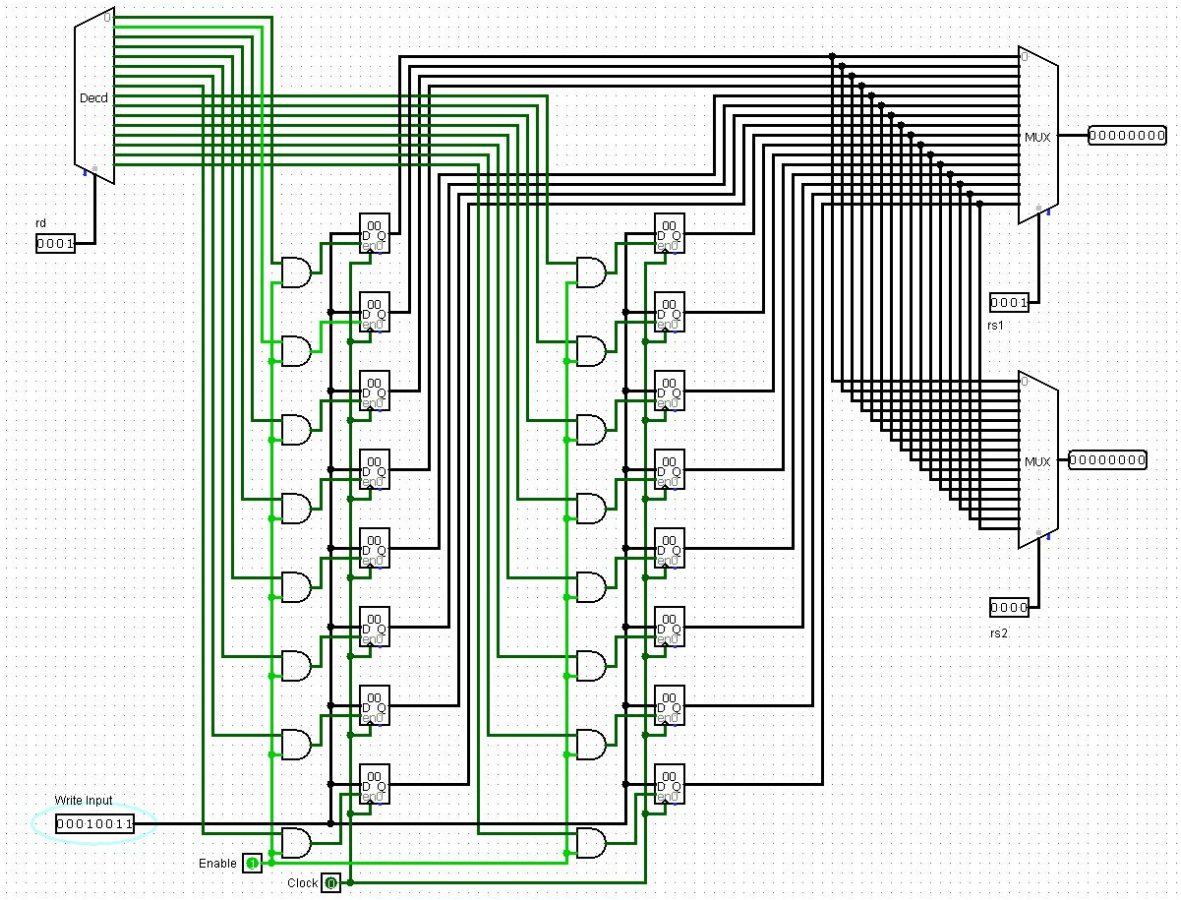
1	0	1	0	1
1	1	1	1	1

Using these outcomes, boolean logic can be applied to make a set of rules for which operations should be used. The corresponding gates are then incorporated into the circuit. Two AND gates, two OR gates, and one XOR gate were used.

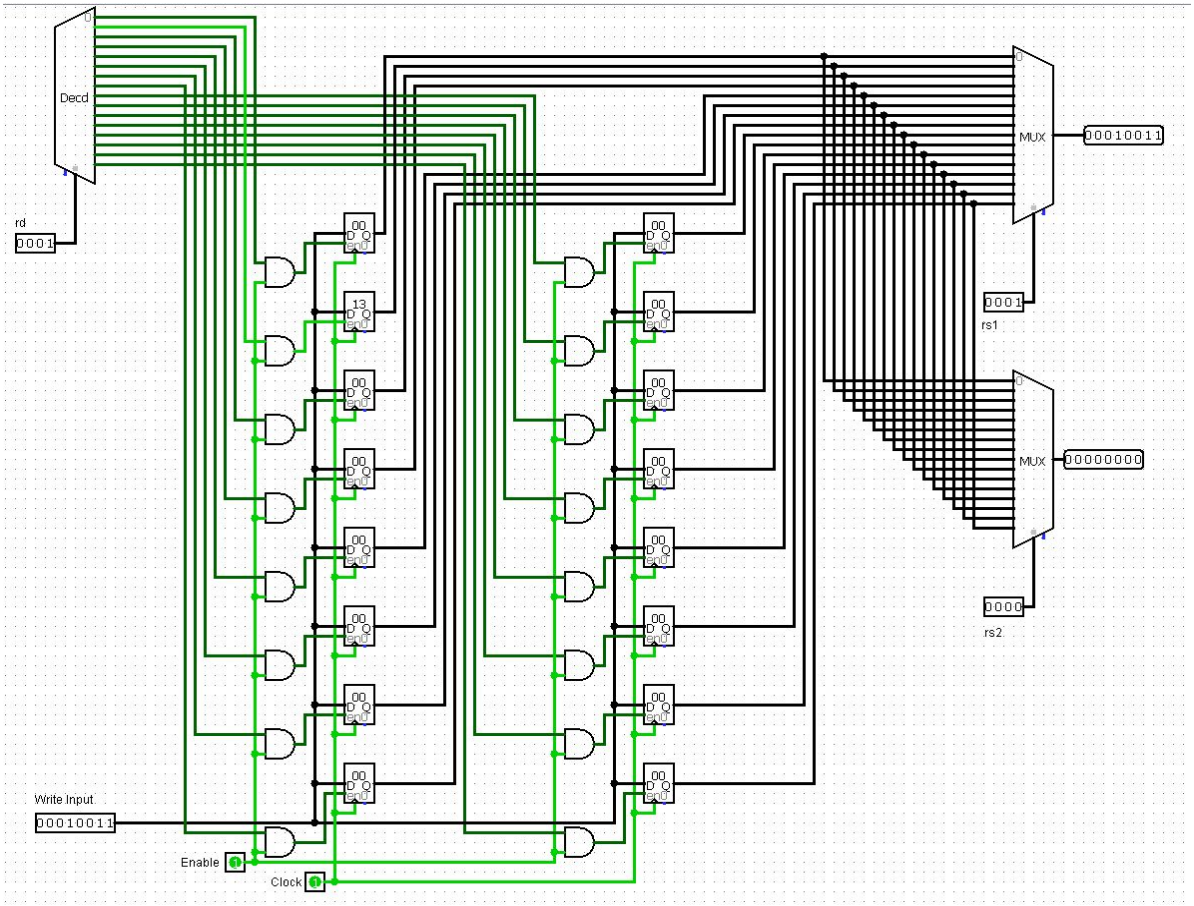
The ALU must be able to AND, OR, and add depending on input settings C1 and C2. Using multiplexers makes this possible. Both AND and OR operations take only inputs A and B and use one correponding gate each. The add operation takes all 3 inputs and also determines the carry output, so these four are connected with the adder. Two multiplexers then decide what the output will be based on the inputs. The first decides between AND and OR, the second decides between the output of the first multiplexer and add.

When making an n-bit ALU, there are n 1-bit ALUs combined with the carry out of the previous connected to the carry in of the next. C1 and C2 are all changed together for each 1-bit ALU so that one operation is performed by the entire ALU.

4.2. Top Level Register File Design



The 16 x 8 register has 16 8-bit registers, each of which can be given and hold 8-bits of data. The rd has 4 bits and decides where the 8-bit write input goes using a decoder. 16 AND gates are used, taking the decoder outputs and an enable switch, that turns on the register file. The output of this AND gate is connected to the enable of the registers. The write input is connected to every register's D since the write input is the only thing that can be inputed into the registers. A clock switch is also connected to every register at the clock input. A clock must turn from 0 to 1 for the write input to be inputed into the register. There are two multiplexers that each take all 16 of the register file outputs. The two demultiplexers have 4-bit rs1 and rs2 which choose which register to read. This does not depend on clock signals but only which register is being read and what the register holds.



This shows the second register (x1) being read after the write input has been sent into it after the clock signal turned to 1.

4.3. Detail of your memory settings

Data Memory:

Selection: RAM	
Address Bit Width	8
Data Bit Width	8
Data Interface	Separate load and store ports

My data memory has 8 bit addresses as the address will be a sum of an output from the register file (8-bits long) and the immediate (4-bits long but with 4 extra zeros becomes 8-bits). The data bit width is 8 bits as the data must be compatable with the register file, which deals with 8-bit long pieces of data. I chose separate load and store ports with the plan to potentially do both load and store functions but I opted to merely load, so I do not use the store port.

Instruction Memory:

Selection: RAM	
Address Bit Width	3
Data Bit Width	16
Data Interface	One asynchronous load/store port

My instruction memory has 3 bit long addresses because I only needed 6 instructions so having 8 slots of data is the smallest power of 2 I could have, hence a address bit width of 3. The data bit width is 16 because machine language instructions are 16 bits long. I had one load/store port because I knew I would only have to load instructions and never store.

4.4. Detail of your datapath

I had my instructions split into 4 divisions of 4-bit data by a splitter. My program would have to load, add, and branch based on the opcodes. The other three sets of 4-bit data would be the manner in which the operations were done. I made a 3-bit ALU that only adds, which gives it's output to a single 8-bit register (the 8-bit register output also loops back to ALU input). This was done so the instructions keep incrementing by 1 (or 5 when branching). The register file takes 4-bit inputs for rd, rs1, and rs2, which are outputed by the instruction memory, so the corresponding wiring was done. Both sets 8-bit data outputed by the register file (rs1 and rs2 outputs) would be inputed into an 8-bit ALU becuase these components would have to be added for my program. The 8-bit output of rs1 would also be connected to the input of a second 8-bit ALU. The second input of this ALU would be the immediate when loading (since the immediate is 4-bits long, four zeros take the other 4 input slots). The output of this 8-bit ALU is connected to the address of the data memory. Either the output of the data memory or the output of the first 8-bit ALU becomes the 8-bit write input (based on the operation being done).

4.5. Detail of your control logic

Instructions leave the instruction memory and are split into 4 divisions of 4-bits each.

Bits 0-3 make up the opcode and activates one of the three instructions: load, add, or branch. Being that the instructions are either 0001 (beq), 0010 (ld), or 0100 (add), by splting bits 0-3 into 4 1-bit pieces, it can be clearly seen which operation to do. Bits 0-2 each can be either 0 or 1 and perform a function when 1, so they each connect to a multiplexer and determine between two inputs.

For deciding whether to branch or increment by 1, the multiplexer normally outputs 001 until the branch operation tells it to output the immediate (though normally 4 bits, it is splitted and bits

0-2 are reconnected). For determining whether to add or not (with the first 8-bit ALU, which requires C1 = 0 and C2 = 0), the multiplexer will output nothing or a 0 when adding, which connects to both C1 and C2. For deciding to load or not, the multiplexer does the same thing as determining whether to add or not but with the second ALU (which is designated for choosing a memory address). Additionally, the load switch on the data memory is turned on.

Bits 4-7 make up either the rd for the register file (when adding or loading) or the immediate (when branching), so the data goes to both of these places but only does something when the opcode enables the necessary components.

Bits 8-11 always make up rs1 so it only goes there.

Bits 12-15 makes up either the immediate (when loading) or rs2 (when adding or branching), so once again, the data goes to both placces buy only does the function when the opcode enables the necessary components.

The write input can either be the output of the data memory or the first 8-bit ALU, which is determined by a multiplexer. The multiplexer is controlled by the load function from the opcode, so the multiplexer outputs the data memory output when loading and the ALU output when not loading.

4.6. Programming

My program computes the Fibonacci Sequence. When determing the machine language necessary, first I wrote the algorithm for this to work:

1. a = 1
2. b = 1
3. c = a + b
4. a = b
5. b = c
6. Go back to step 3

I then began turning this algorithm into assembly language. I assigned each variable to a register so:

- a to x0
- b to x1
- c to x2

When assigning original values a = 1 and b = 1, these values must come from the data memory. By going to memory address 0(x0), a 1 can be loaded from address 0. Now that x0 holds a 1, by going to 0(x0), another 1 can be loaded from address 1.

Additionally, the number zero would be used often as the operations a = b and b = c can be addition operations as: a = b + 0 and b = c + 0. So, having a register for the value of 0 would be useful. Thus: 0 to x3 (this doesn't need a step as registers hold 0 by default). With all of this, the assembly language can be written:

ld x0, 0(x0) - Load register x0 with 1
ld x1, 0(x0) - Load register x1 with 1
add x2, x0, x1 - Add contents of registers x0 and x1 to register x2
add x0, x1, x3 - Add contents of registers x1 and x3 to register x0
add x1, x2, x3 - Add contents of registers x2 and x3 to register x1
beq x2, x1, x5 - If the contents of registers x1 and x2 are equal, execute the instruction 5 lines down (this makes it go back to the third instruction)

This can then be translated into machine code:
0000 0000 0000 0010
0000 0000 0001 0010
0001 0000 0010 0100
0011 0001 0000 0100
0011 0010 0001 0100
0101 0010 0101 0001

When inputed into the intruction memory, it becomes:
0000
0002
0012
1024
3104
3214
1251

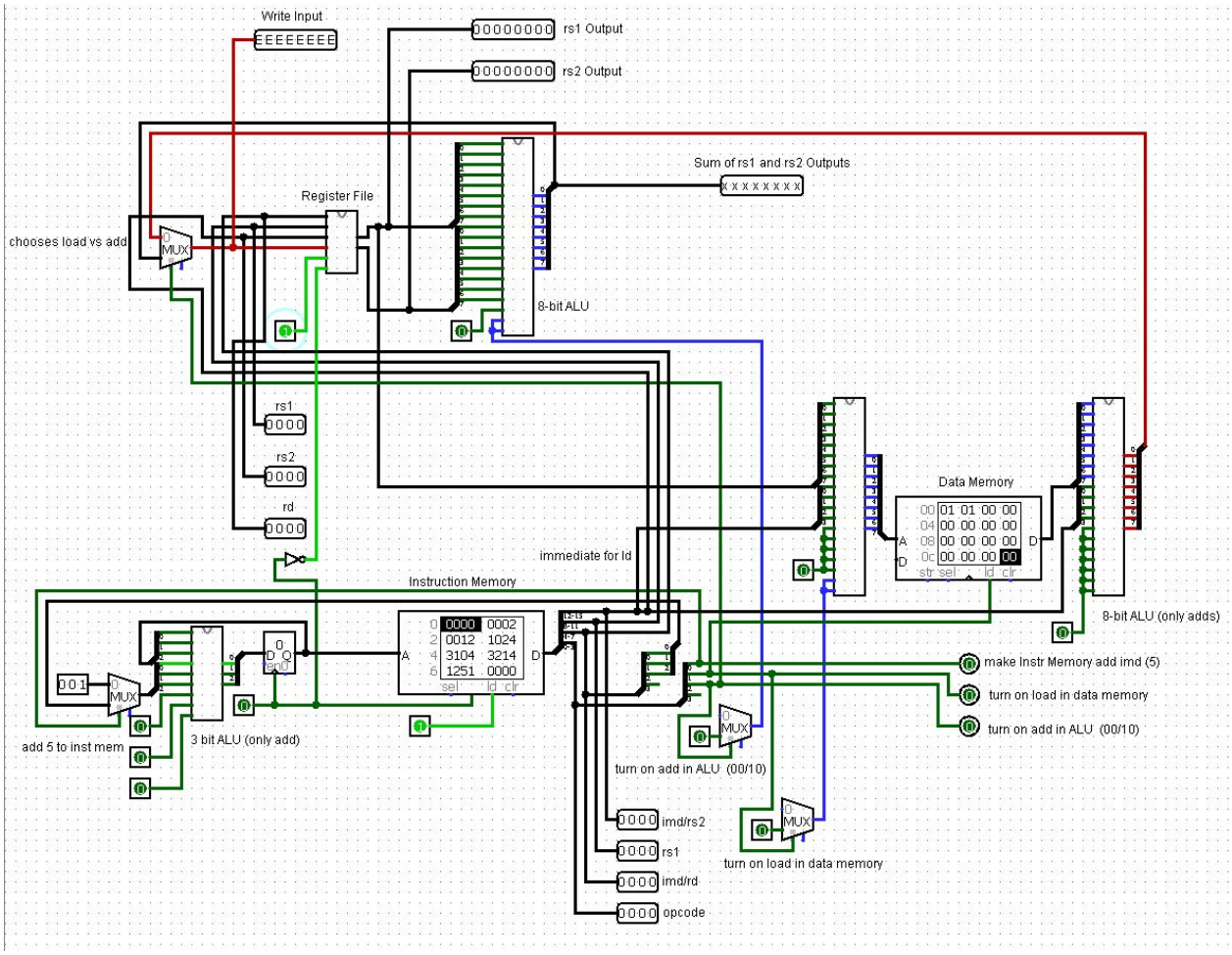
(I put a 0000 before the instructions to act as a filler instruction, as I put a NOT gate for the clock signal of the register file, which was done to resolve an issue with the register file performing functions with a delay)

4.7. Simulation

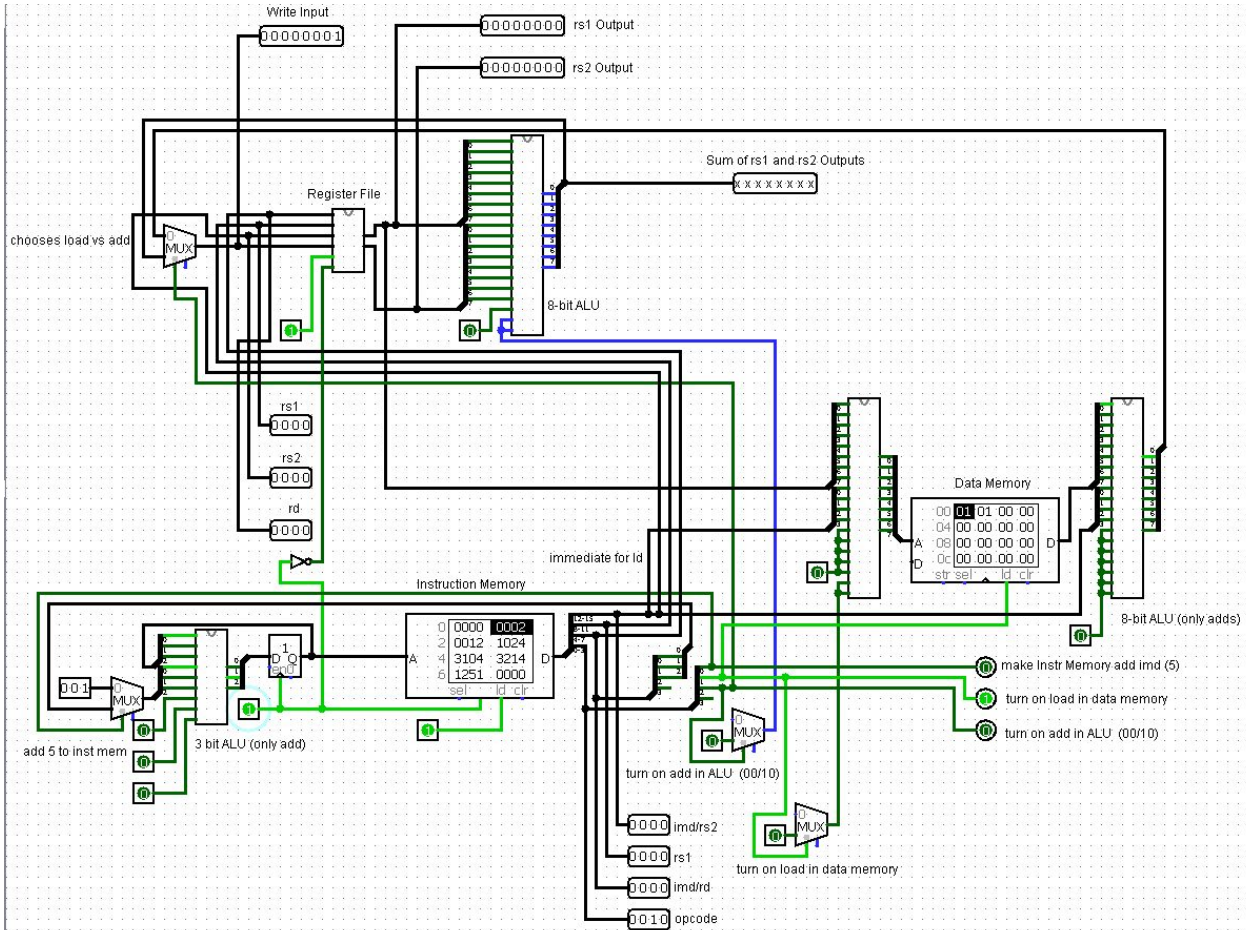
Setup:

1. Turn on load feature for instruction memory.
2. Turn the 000 into 001 (for consecutive instructions).
3. Enable the register file.
4. Fill in the data memory (1 for both addresses 0 and 1).
5. Add instructions to instruction memory.

After Setup:



After one click:



After two clicks:

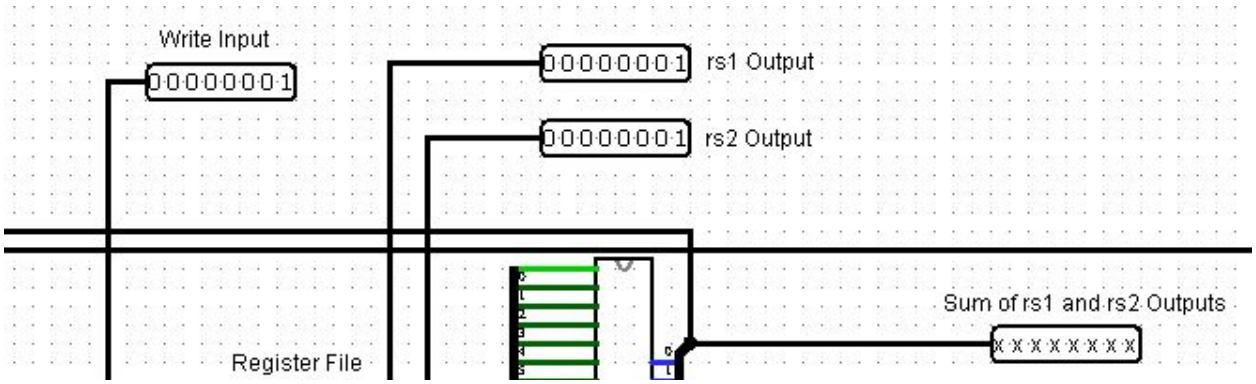


Chart of readings:

Clock Signal	Write Input	rs1 output	rs2 output	Sum	Significance
Original	X	0	0	X	Original

1 click (1)	00000001	00000000	00000000	X	Loading 1
2 clicks (0)	00000001	00000001	00000001	X	x0 holds 1
3 clicks (1)	00000001	00000001	00000001	X	x1 holds 1
4 clicks (0)	00000001	00000001	00000001	X	
5 clicks (1)	00000010	00000001	00000001	00000010	c set to 2 (1, 1, 2)
6 clicks (0)	00000010	00000001	00000001	00000010	
7 clicks (1)	00000001	00000001	00000000	00000001	a set to 1
9 clicks	00000010	00000010	00000000	00000010	b set to 2
11 clicks	X	00000010	00000010	X	Branching
13 clicks	00000011	00000001	00000010	00000011	c set to 3 (1, 1, 2, 3)
21 clicks	00000101	00000010	00000011	00000101	c set to 5 (...2, 3, 5)
29 clicks	00001000	00000011	00000101	00001000	c set to 8 (...3, 5, 8)
37 clicks	00001101	00000101	00001000	00001101	c set to 13 (...5, 8, 13)
45 clicks	00010101	00001000	00001101	00010101	c set to 21 (...8, 13, 21)

4.8. Reflection

This project has overall been a very good experience in learning, struggling, and having fun. I had learned about gates and transistors and programming in java beforehand but I had not really learned much about the area in between, such as microarchitecture and instruction set architecture. I didn't know how a bucnh of 0s and 1s become formal instructions in making a program run.

From this project, I learned what an ALU and a register file are, how to build them, and their significance in a computer. I learned what assembly language is, how to use it, and how to translate it into machine code to be used in a computer. I learned about memory and how to use it in a computer through machine code.

Coming into this project, I didn't know much about computer engineering. I just knew it was like a mix of electrical engineering and computer science. However, this project has shown me it is more than a sum of those two fields and that it is a really fun area to study. I planned on pursuing computer engineering during my undergraduate years and this project has further confirmed my goal to do so.