

May 18<sup>th</sup>, 2024

---

**Overview**

---

**Design Goal:**

Create from scratch <sup>1</sup> a Turing complete 8 bit computer.

**Timeline:**

May: Design decisions	Logic family Proof of concept logic gates What are the macro components I need?
June: Logisim macro components, Breadboard macro components	
July: Full Logisim computer, Breadboard macro components	
August: Complete design, Manufacturing	

---

**Logic Families**

---

	Pros	Cons
<b>Resistor Transistor Logic (RTL)</b>	Incredibly simple BJT transistors Inexpensive components Static electricity resistant	Power inefficient Lots of components needed Slow switching speed Susceptible to noise Bad fan out
<b>Diode Transistor Logic (DTL)</b>	BJT transistors Inexpensive components Static electricity resistant Handles noise well	Power inefficient More complicated than RTL Lots of components needed Slow switching speed
<b>Transistor Transistor Logic (TTL)</b>	Power efficient BJT transistors Inexpensive components Static electricity resistant	Ideally uses multiple emitter transistors More complicated than RTL Lots of components needed

---

<sup>1</sup> I will not be manufacturing resistors and transistors.

<b>Complementary Metal Oxide Semiconductor Logic (CMOS)</b>	Incredibly power efficient Very fast switching speed	Static electricity susceptible Expensive components Complicated logic gates
---	---	---

There are of course more logic families, these are simply the most common and have differences that will be relevant to the project. Due to the low clock speed of the final computer, I don't have to worry too much about transistor saturation<sup>2</sup>.

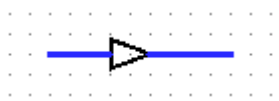
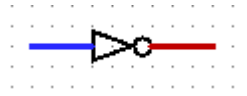
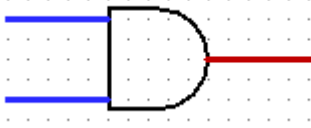
After screening<sup>3</sup> I cut DTL for the more complicated gate design and CMOS for the price and complexity again. Comparing TTL and RTL, RTL's simplicity won out for me. I do not have access to an oscilloscope currently and I want to minimize the amount of high-speed troubleshooting I need to do.

## Digital Logic

### Logic Gates

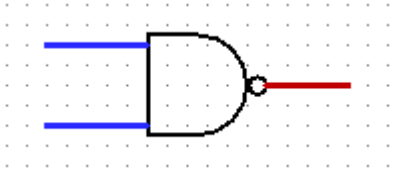
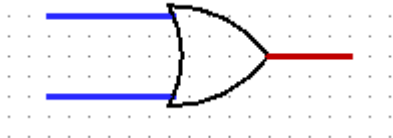
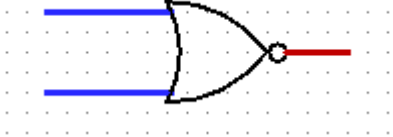


The fundamental building blocks of digital logic, these simply take in one or more input and give back one output.

Logic gates are commonly shown as block diagrams, which are summarized below:

	Block Diagram	Truth Table															
<b>BUFFER</b>		<table><tr><th>IN</th><th>OUT</th></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr></table>	IN	OUT	0	0	1	1									
IN	OUT																
0	0																
1	1																
<b>NOT</b>		<table><tr><th>IN</th><th>OUT</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	IN	OUT	0	1	1	0									
IN	OUT																
0	1																
1	0																
<b>AND</b>		<table><tr><th>A</th><th>B</th><th>OUT</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	OUT	0	0	0	1	0	0	0	1	0	1	1	1
A	B	OUT															
0	0	0															
1	0	0															
0	1	0															
1	1	1															

<sup>2</sup> If your project is going to run at a very high clock speed, or you care about transistor saturation you should look at Schottky TTL.

<sup>3</sup> Yes, we're using APSC 100-101 terminology.

NAND		<table> <tr><th>A</th><th>B</th><th>OUT</th></tr> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </table>	A	B	OUT	0	0	1	1	0	1	0	1	1	1	1	0
A	B	OUT															
0	0	1															
1	0	1															
0	1	1															
1	1	0															
OR		<table> <tr><th>A</th><th>B</th><th>OUT</th></tr> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </table>	A	B	OUT	0	0	0	1	0	1	0	1	1	1	1	1
A	B	OUT															
0	0	0															
1	0	1															
0	1	1															
1	1	1															
NOR		<table> <tr><th>A</th><th>B</th><th>OUT</th></tr> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </table>	A	B	OUT	0	0	1	1	0	0	0	1	0	1	1	0
A	B	OUT															
0	0	1															
1	0	0															
0	1	0															
1	1	0															
XOR		<table> <tr><th>A</th><th>B</th><th>OUT</th></tr> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </table>	A	B	OUT	0	0	0	1	0	1	0	1	1	1	1	0
A	B	OUT															
0	0	0															
1	0	1															
0	1	1															
1	1	0															
XNOR		<table> <tr><th>A</th><th>B</th><th>OUT</th></tr> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </table>	A	B	OUT	0	0	1	1	0	0	0	1	0	1	1	1
A	B	OUT															
0	0	1															
1	0	0															
0	1	0															
1	1	1															

### Boolean Algebra

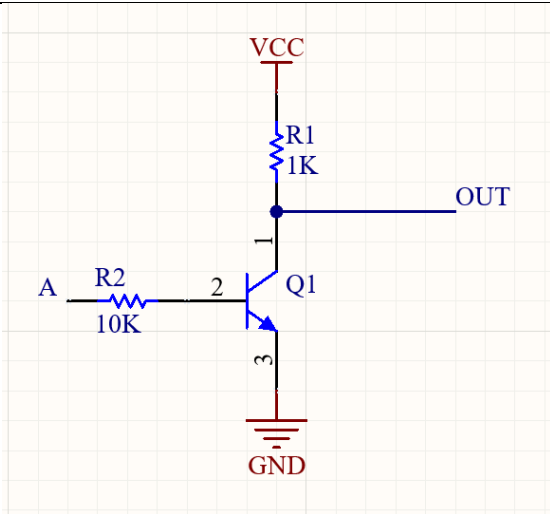
Boolean algebra allows us to compute what will happen in a more complex circuit. All the logic gates above can be represented through 4 mathematical operations:

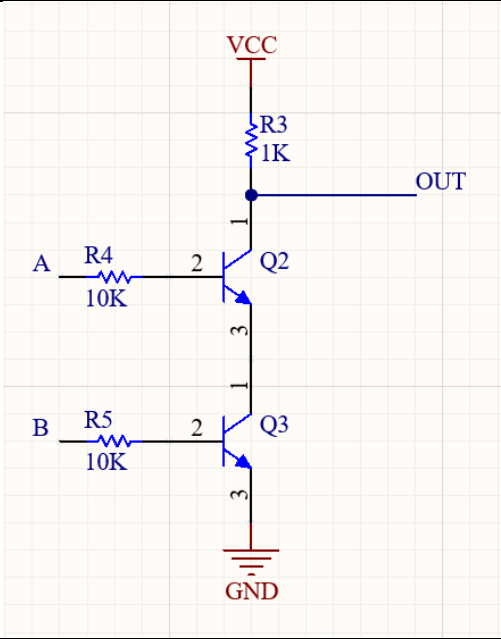
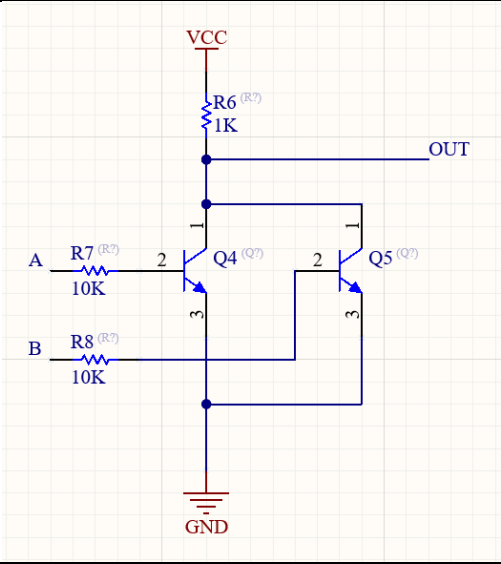
NOT	$\bar{A}$	Inverts the value of A	IN	OUT
			0	1
			1	0

<b>OR</b>	$A + B$	Returns 1 if either A or B is 1	A	B	OUT
			0	0	0
			1	0	1
			0	1	1
			1	1	1
<b>AND</b>	$AB$	Returns 1 if both A and B are 1	A	B	OUT
			0	0	0
			1	0	0
			0	1	0
			1	1	1
<b>XOR</b>	$A \oplus B$	Returns 1 if only A or B is 1	A	B	OUT
			0	0	0
			1	0	1
			0	1	1
			1	1	0

We can combine these operations to get any of the logic gates above.

RTL Logic Gates

NOT		<table><tr><th>IN</th><th>OUT</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	IN	OUT	0	1	1	0
			IN	OUT				
0	1							
1	0							

<div>NAND</div>	<div></div>	<table><thead><tr><th>A</th><th>B</th><th>OUT</th></tr></thead><tbody><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></tbody></table>	A	B	OUT	0	0	1	1	0	1	0	1	1	1	1	0
A	B	OUT															
0	0	1															
1	0	1															
0	1	1															
1	1	0															
<div>NOR</div>	<div></div>	<table><thead><tr><th>A</th><th>B</th><th>OUT</th></tr></thead><tbody><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></tbody></table>	A	B	OUT	0	0	1	1	0	0	0	1	0	1	1	0
A	B	OUT															
0	0	1															
1	0	0															
0	1	0															
1	1	0															

XOR

(Notice that this is just four NAND gates)

A	B	OUT
0	0	0
1	0	1
0	1	1
1	1	0

<https://www.youtube.com/watch?v=nB6724G3b3E>

May 25<sup>th</sup>, 2024

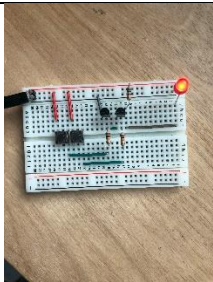
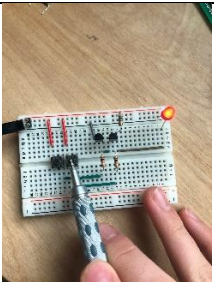
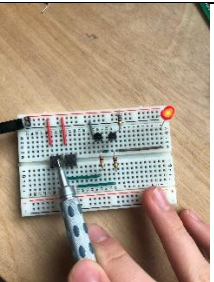
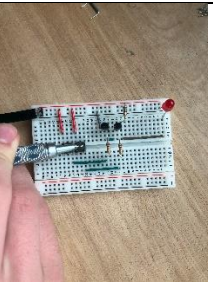
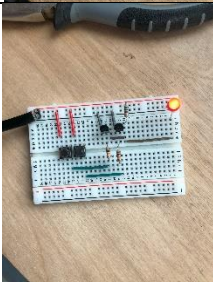
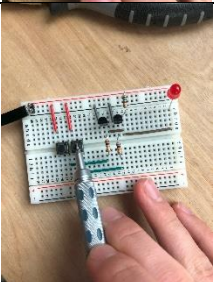
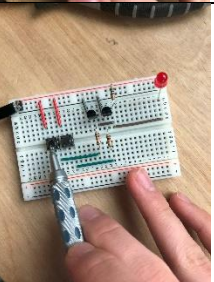
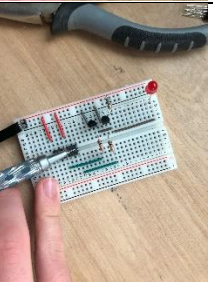
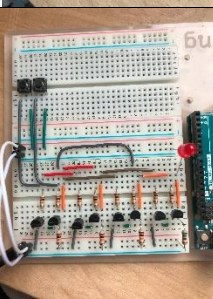
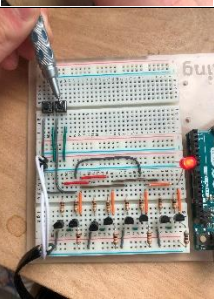
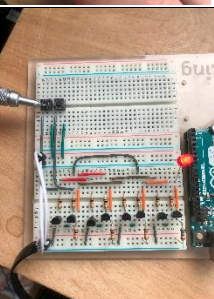
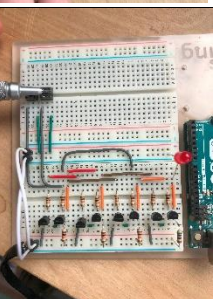
Physical Logic Gates

After creating the schematics for physical RTL logic gates last week, I built all of them up on breadboards:

NOT

0

1

	00	01	10	11
NAND				
NOR				
XOR				

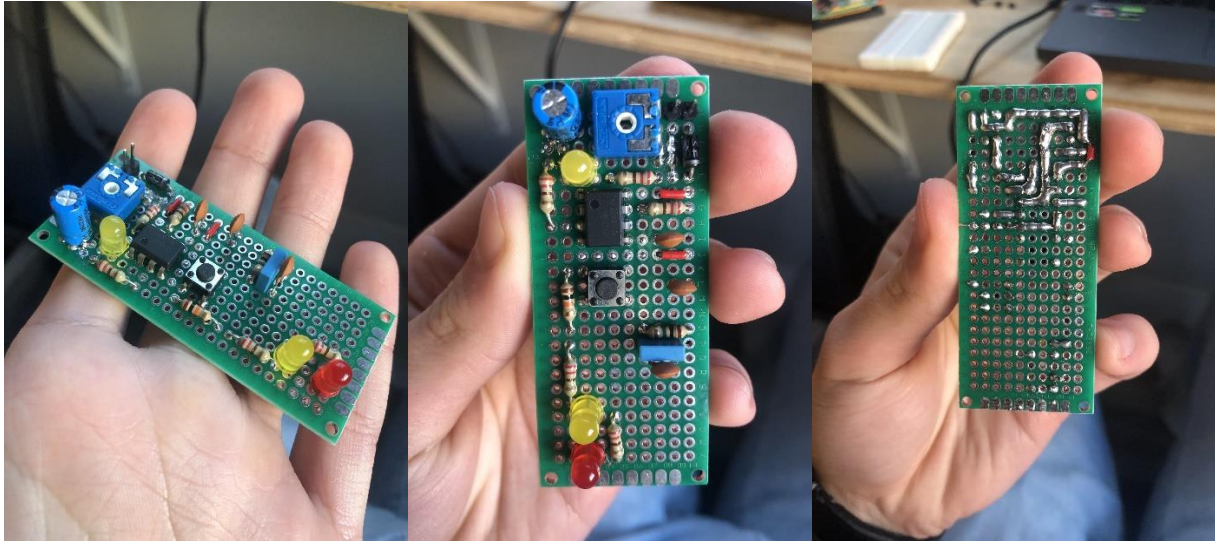
## 74LS Series Chips

Making a computer out of entirely transistors is... completely pointless. As I've built up logic gates from transistors, I will now be using the 74LS series for all my logic. It's TTL with Schottky diodes for ultra fast switching speed.

## Half Complete Clock Module

Ben eater has an amazing video on the 555 timer (among other things) which I'm using for my clock module. It has both an automatic and manual step mode with variable timings and a debounced button.

Here is what I could make this week with the components I had on hand:



Video to come next week.

---

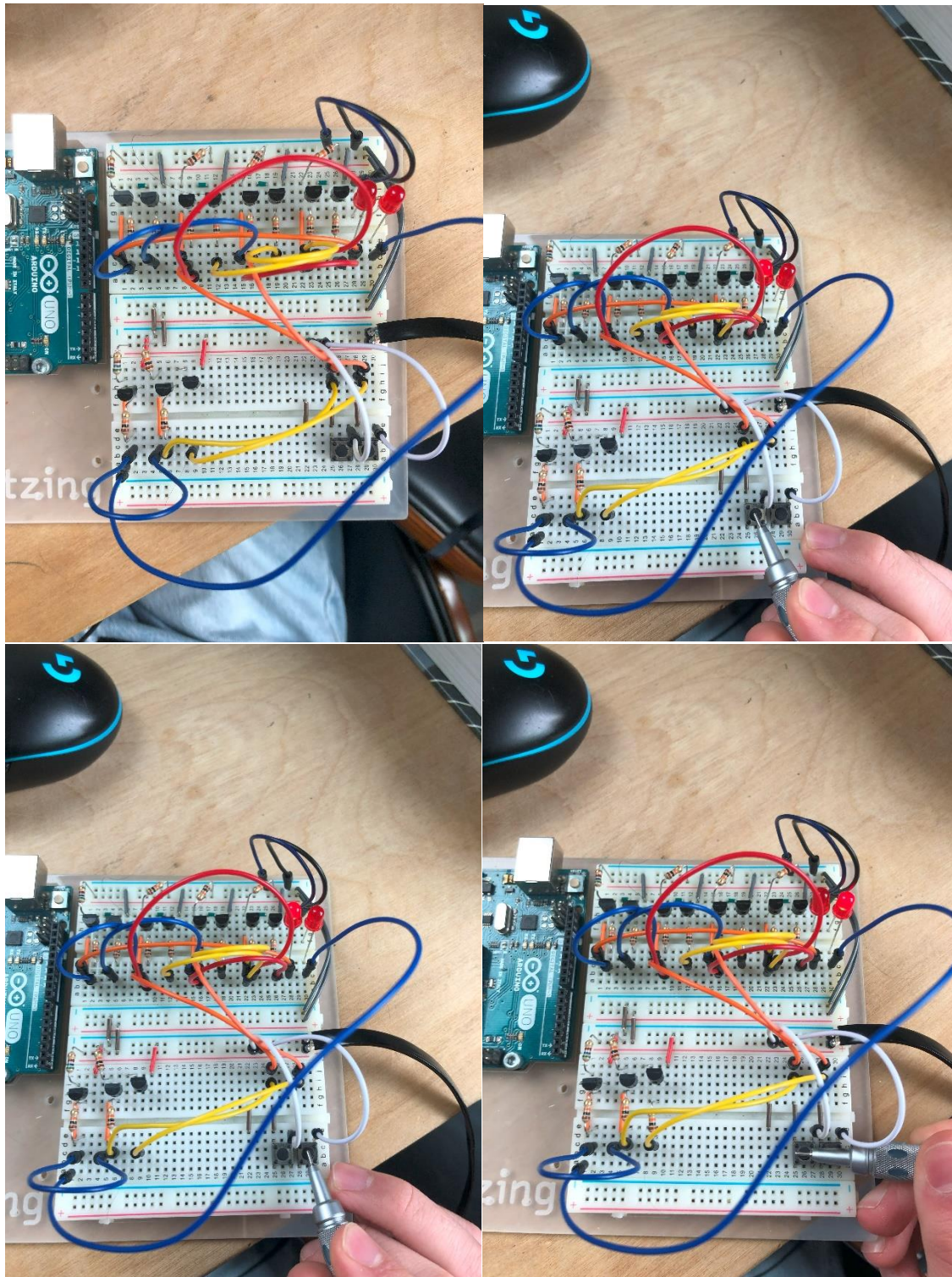
### Half Adder

One of the big parts I'll need for the final computer is an arithmetic and logic unit (ALU). The final design will be able to add, subtract, AND, OR, NAND, NOR, and XOR the numbers in registers A and B, then output them to register C.

The “core” of the ALU is the adder, and all that it does is add two 8-bit numbers together.

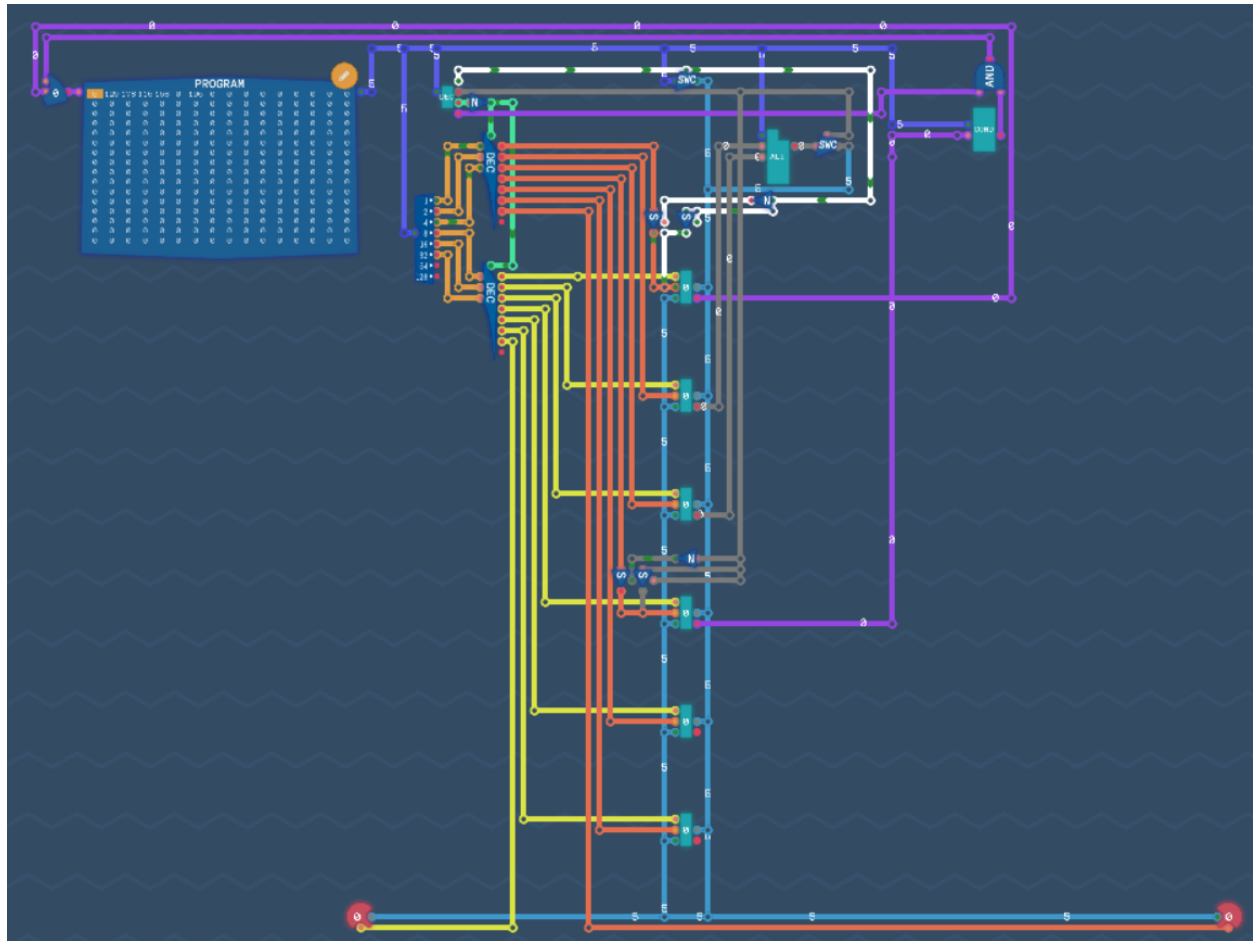
Next week will include what a full adder entails, and the algorithm for why a half adder doesn't work.





Almost Complete Simulated Computer

Before committing to buying chips and manufactured PCB's I went through exercises and built a Turing complete computer<sup>4</sup> in a digital logic simulator



There are still improvements to be made, for example there's no circuitry to XOR and no stack implementation yet, however this is a great starting point for building my computer and lets me know exactly what I'll need for the final design.

## Macro Components

---

The final design of this project will be relatively tame, and as such doesn't require and exorbitant number of parts.

## Core part list.

---

<sup>4</sup> A Turing complete computer, given enough time and memory can compute any computable problem.

Part	Purpose	Number Required
Arithmetic and Logic Unit (ALU)	<p>Do the following to 8-bit numbers in registers 1 and 2, then output to register 3:</p> <ul style="list-style-type: none"> <li>- Add</li> <li>- Subtract</li> <li>- AND</li> <li>- OR</li> <li>- NAND</li> <li>- NOR</li> <li>- XOR</li> <li>- XNOR</li> </ul>	1
Registers	Receive, store, and send 8-bit numbers on the bus.	5
Condition Unit	<p>Checks the 8-bit number in register 3 if it meets one of the following conditions, if it does change the program counter's value to what is in register 0:</p> <ul style="list-style-type: none"> <li>- Never</li> <li>- Always</li> <li>- Reg3 = 0</li> <li>- Reg3 ≠ 0</li> <li>- Reg3 ≥ 0</li> <li>- Reg 3 &gt; 0</li> <li>- Reg 3 &lt; 0</li> <li>- Reg 3 ≤ 0</li> </ul>	1
Program Counter	Counts up by 1 each clock cycle, then outputs that number to memory. If it receives a signal from the condition unit then it will change its value to the value in register 0.	1
Memory	Stores the program of the computer in 8-bit numbers called opcodes. The opcode	1

	output will correspond to the program counter's output.	
Instruction Decoder	<p>Does one of the following with the current opcode:</p> <ul style="list-style-type: none"> <li>- Put the opcode value into reg0 (immediate)</li> <li>- Copy from one register to another (copy)</li> <li>- Do a mathematical operation with the ALU (compute)</li> <li>- Check if a condition is met with reg3, if it is change the program counter value to the value in reg0 (condition)</li> </ul>	1
Clock	<p>Outputs a square wave to all synchronous components. Can run automatically or manually.</p>	1

These are the main bits, but I'll need "glue logic" as well.