Building an 8-bit Turing-complete CPU on an array of breadboards

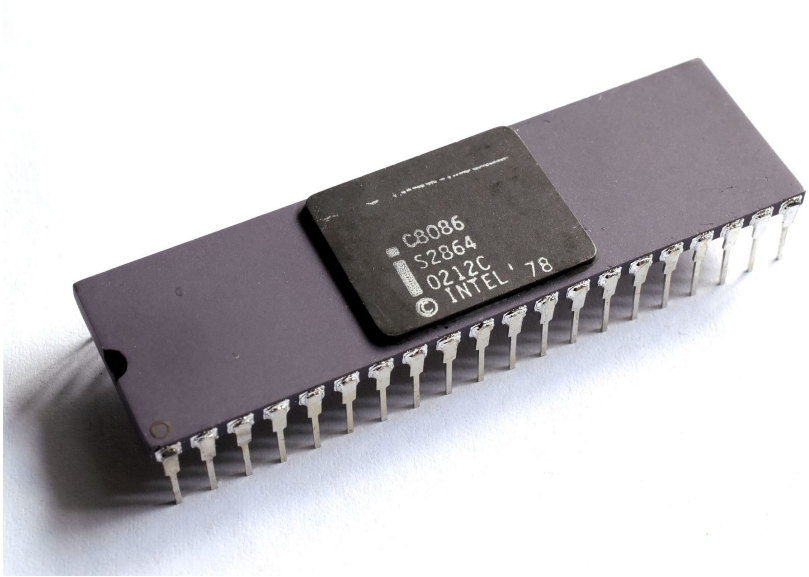
COMP 400 Independent Project, 2021
McGill University

MAREK BORIK
Supervised by Prof. Joseph Vybihal

TABLE OF CONTENTS

1. INTRODUCTION

Computer processors have gotten very sophisticated over the years. One of the very first “consumer-grade” processors was the 16-bit Intel 8086, released in 1978, that ran on 5MHz clock frequency. With 29,000 transistors it set the standard for future processors to come with its x86 architecture. The understanding of the individual operations in the processor was much simpler back then, as current-gen processors run on clocks literally 3 orders of magnitude higher, have several billions of transistors and also benefit from architectural improvements and new features such as speculative execution.



Intel 8086. Picture credit: https://en.wikipedia.org/wiki/Intel_8086#/media/File:Intel_C8086.jpg

2. GOAL

The goal of this project to build an 8-bit Turing-complete CPU on an array of breadboards. The schematics and tutorials are available on the internet. More about those in the Resources section. The processor has several basic instructions, 16 bytes of RAM (meaning each program can consist of at most 16 8-bit instructions + data) and 8-bit bus. First four bits of each instruction will generally be used as an opcode and the last four bits can be used for an immediate, address or other necessary information, according to the opcode. The clock will allow for a single-step and a normal mode, where the frequency will be adjustable by the user, and it will be able to reach up to 450 Hz.

Apart from simply building the processor, I am going to challenge myself and will try to improve 1 thing about the design. My plan is to improve the power supply because the original design uses a simple USB phone charger to provide 5V. However, I am estimating that the build may draw up to 1.5A, so a proper buck or DC-DC converter should be used for increased stability and thermal performance.

3. RESOURCES

The biggest resource for this project is the website https://eater.net/8bit. The creator of this design, Ben Eater, provides the schematics and tutorials, however I am imagining I won't be spared from a lot of troubleshooting and debugging on my side. He also sells the parts kits, some of which I purchased. Other tools I will be using:

- Multimeter
- Oscilloscope
- Arduino Software
- Wire strippers
- Tweezers

4. BUILD PROCESS

4.1. CLOCK MODULE

4.1.1. Automatic mode

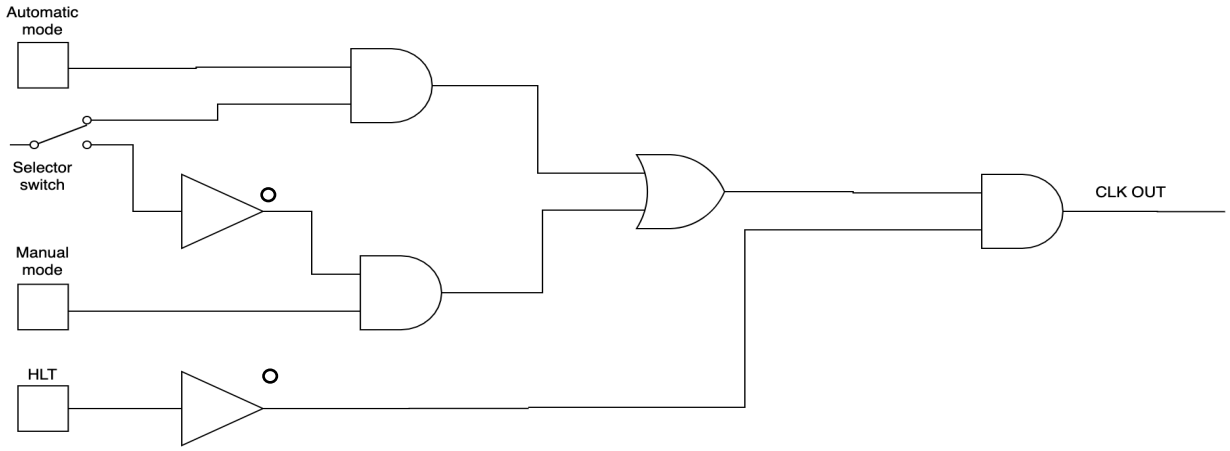
The heartbeat of the processor is a clock signal. Here we turn to the humble LM555, otherwise known as the 555-timer. We are using it in the astable configuration with a 1kΩ resistor, 1MΩ potentiometer (so that we can adjust the frequency) in series with another 1kΩ resistor and finally a 1μF capacitor. We also added a filtering 0.01μF capacitor on pin 5 to ground to help reduce noise and voltage overshooting, per datasheet recommendations. This allows us to run the clock at anywhere from 0.25 Hz – 450 Hz.

4.1.2. Manual mode

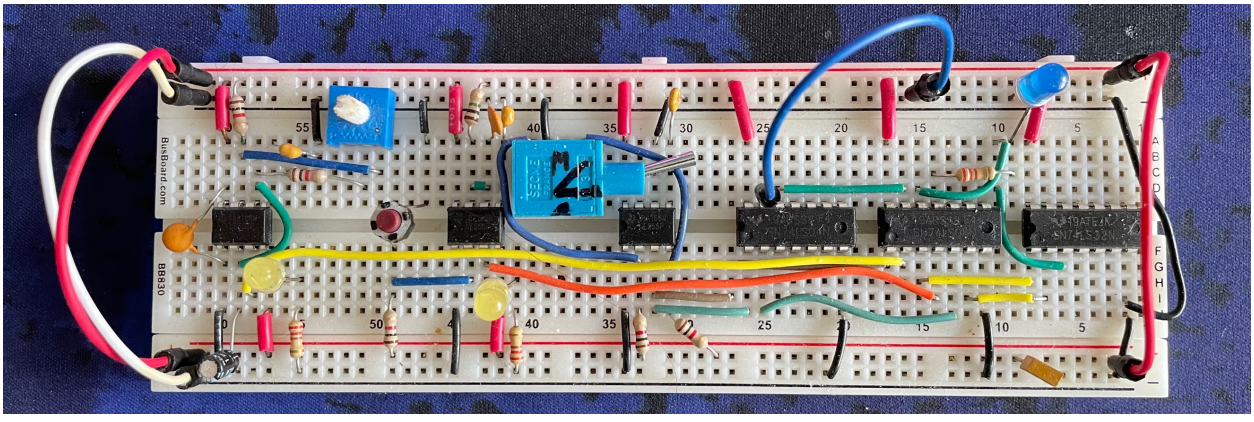
It is really handy to have the option to single step the execution of our program. We will achieve this by pushing a button, which will act as one clock pulse. The finished clock module will have a selector switch and we will be able to select whether we want the Automatic mode or the Manual mode. Very quickly we find a big drawback of physical buttons – they bounce. Imagine a situation where we are trying to single step our program and our 1 push of the button makes it bounce 3 times and we actually send 3 clock cycles. That is terrible and unacceptable. To solve this, we are going to use the LM555 again, this time in its “debouncing” (monostable) configuration.

4.1.3. Clock logic

We are ready to tie everything together. We can add a selector switch for the 2 modes and debounce it with another LM555. Lastly, we are going to incorporate a halt (*HLT*) signal. We would like to disconnect the clock when we are done with our computation. When *HLT* is low, the clock propagates to the output and when it is brought up, the clock is disconnected. Here are the logic gates to achieve the desired outcome:



We will use a 74LS08 chip for our 3 AND gates, 74LS32 for our OR gate and 74LS04 for our inverter. Since we are only using 1 OR gate and 2 inverters, we could save a chip and implement the same logic by only using NOR / NAND gates. Here is the finished module:



4.2. REGISTER MODULE

4.2.1. Motivation & interfacing with the bus

Our architecture incorporates an 8-bit wide bus. It is crucial that only certain modules read from and output to the bus at the same time, otherwise the transistors in the chips can behave like current sources / sinks and the bus can have an unpredictable behavior. To overcome this problem, we would like to have a 3-state functionality, so we need our output to be either zero, one or disconnected (high impedance). Our registers should also have a Load option, where we latch the values into the register only on the rising edge of the clock signal and when Load is set.

This logic can be done discretely, but we will use two 74LS173 4-bit D registers to accomplish this same functionality. Since they have a 3-state output, they could be used on their own, but they pose one big disadvantage for this particular project: you cannot see what is latched inside them. We would like to have 8 LEDs that depict the stored 8-bit value. We are going to solve this by always having the output of the registers always enabled and going to a 74LS245 8-bit transceiver that does have 3-state outputs.

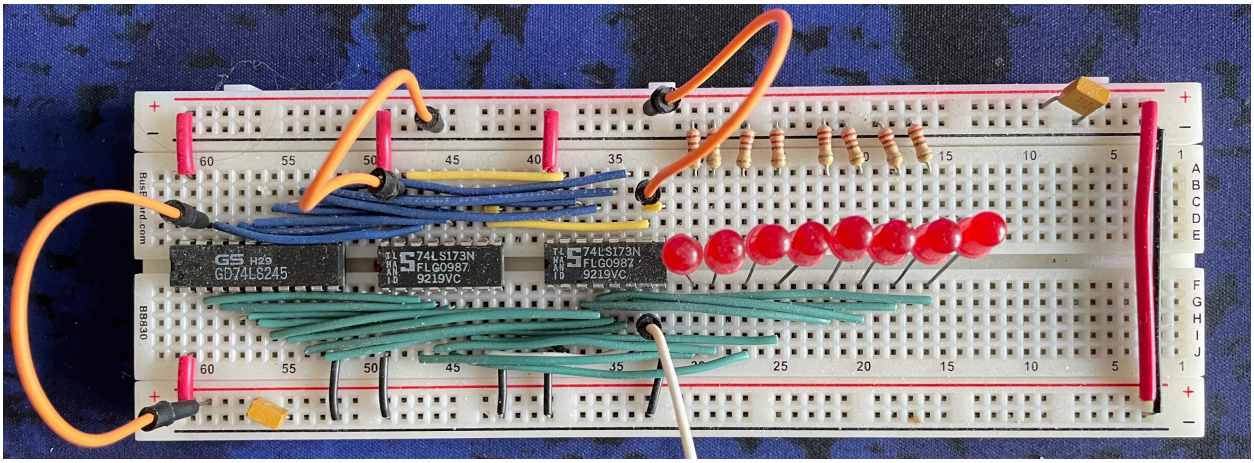
4.2.2. Building the A register

We start by connecting the outputs of our two 4-bit D registers with the inputs of our 8-bit transceiver. It is easy to tap to this connection with 8 red LEDs. Note that most of the 74LS type chips, including the 74LS173 4-bit D registers, have internal current limiting resistors, so we can connect the diodes directly to ground. With their presence we can easily see the stored 8-bit number without outputting onto the bus. Next, we tie the output of the 8-bit transceiver to the input of the D registers. This connection will later also be used to interface with the bus. It may seem strange that we are essentially connecting our output back to the input, but remember that if we do things correctly, we will either be reading from the input and having the output disconnected or writing to the output ourselves, but never both at the same time.

4.2.3. Other signals

Both 4-bit registers have 2 separate load signals $\overline{G1}$ and $\overline{G2}$ on pins 9 and 10. We can tie all four together and temporarily to 5V. This will be later used as the *AI* (A in) signal. They also both have a *CLR* option on pin 15, so we tie both together and for now we hook it to ground.

Since the transceiver is bidirectional, we set the DIR pin high to choose our direction, because we will be using it only in one direction. We will also set the transceiver's $\overline{OE}$ pin to 5V to disconnect the output (left most orange wire). This will later be used as *AO* (A Output) signal. Here is the finished A register:



4.2.4. Other registers

In the previous part we completed the A register. We will need to build a B register, which is identical to the A register and an instruction register, which has a few differences. There will be 4 blue and 4 yellow LEDs instead of the 8 red ones. The top four bits (blue LEDs) will not go back to the bus, but instead they will go to an instruction decoder, which we will build later. The bottom 4 bits (yellow LEDs) can output back to the bus and will contain the remainder of the instruction, either a memory address or an immediate. We can see that since we can only address 4 bits of memory, our program can have at most 16 memory addresses for instructions or data. That should still be fine for simple programs and adding a JUMP instruction will increase the capability.

4.3. ARITHMETIC LOGIC UNIT (ALU)

4.3.1. Motivation and interfacing

ALU is the core of every processor's core, pardon the pun. It allows is to perform fundamental computations of two binary numbers such as addition and subtraction, among other things. Registers A and B are the inputs to the ALU in our design and the output will use the beloved and previously mentioned 74LS245 8-bit transceiver for the same reasons.

The main operational principle of an ALU stems from the idea of a full binary adder, which performs the addition of 2 bits (and a potential carry bit). Even though they can be easily chained together, building 8 full binary adders would require a lot of chips. We can instead turn to two 74LS283 4-bit binary full adders and cascade them together.

4.3.2. Addition vs subtraction

It seems like having both the options for adding and subtracting would be very helpful. But how are we going to get subtraction from adders? The answer is by

representing all our 8-bit numbers in the two's complement format. If we are going to add 2 positive numbers, the *SU* (Subtract) signal will not be set.

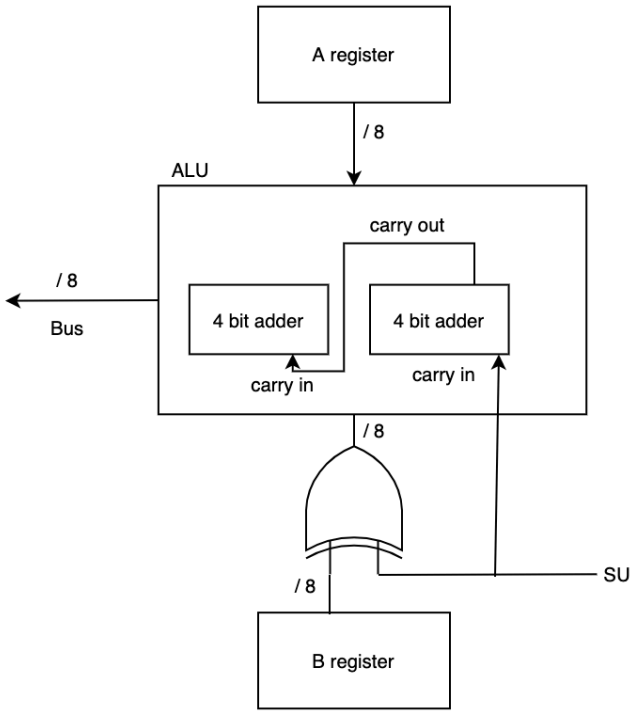
We can then load two numbers into our A and B registers and the addition will be performed. If we set the *SU* signal, the number in the B register will be converted into its two's complement and we can use the same addition hardware to perform the effective subtraction. That means that we will have to design a logic that converts a regular positive number into a negative number. The algorithm for converting is to invert all bits and add 1. For example, $+3_{10} = 00000011_2$. To get -3, we first invert all bits to 11111100_2 and then add one to get 11111101_2, which equals -3 in two's complement.

4.3.3. Subtraction hardware

Let's remind ourselves with a truth table for an XOR gate:

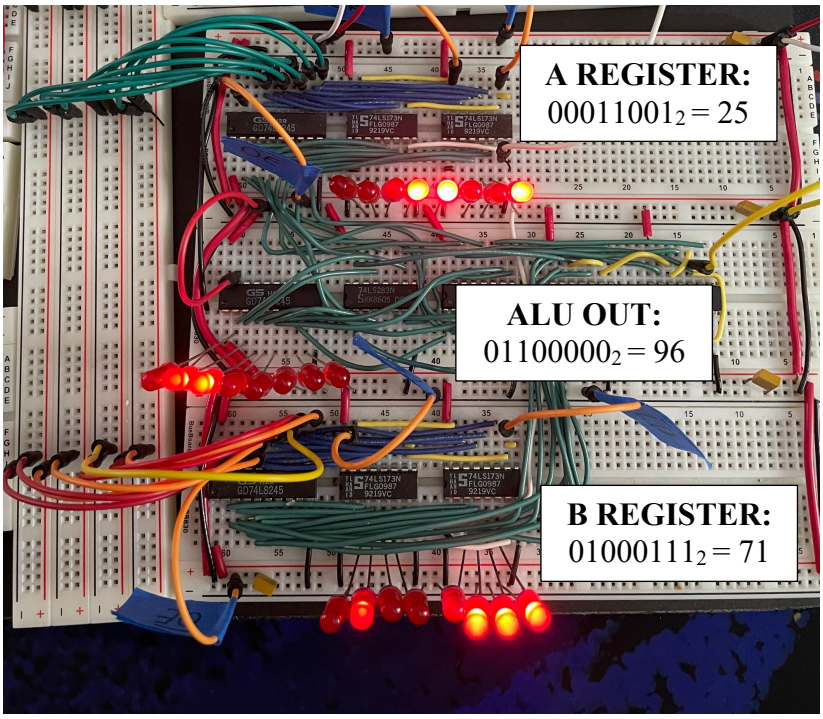
A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

Thinking of A as our *SU* signal gives us exactly what we need. When A is 0, the output is the same as B and when A is 1, the output is $\overline{B}$. But remember the to fully invert we have to also add one at the end. That is not a problem, we can have branch of *SU* going into a carry in pin of our adder. Here is a block diagram of the logic:



4.3.4. Building the ALU

We have outlined all components already; we just have to put it together. We connect the output of the B register into two 74LS86 Quad XOR Gate and we tie all the other inputs into the XOR gate together to form our *SU* signal. We also branch into the carry in pin of our lower 4-bit adder, as described above. The output of the XOR gates goes into the B1-B4 inputs of both of our adders. The A1-A4 inputs will be connected directly to the output of the A register. We can connect the Σ1-Σ4 outputs of the adders into the octagonal transceiver for our 3-state functionality and have some LEDs present to see the intermediate result. As we did with the other registers, we will mark the output enable of the transceiver, which will later be used as our *ΣO* (SUM out) signal. Here is the finished part:



4.4. RANDOM ACCESS MEMORY (RAM)

4.4.1. Design overview

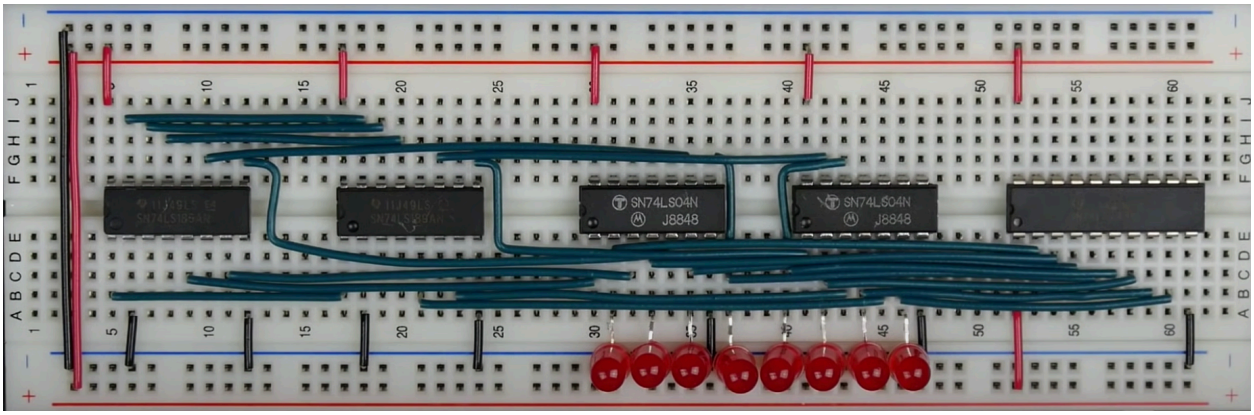
We already built two general purpose registers A and B, but we would like to have some storage as well. As described before, our instruction will consist of a 4-bit opcode and a 4-bit value. This value can for example be an address that we have to retrieve data from. Since we can use only four bits for our address, it gives us a maximum of 16 addressable bytes. We could build 16 copies of the general-purpose registers, but to save on parts, complexity, and repetitiveness and power draw, we are going to use a chip, the 74LS189 64-bit random access memory organized as a 16-word 4-bit array. To be precise, we need two of them to store 16 bytes, which will be our static RAM. It also already contains a 4-bit address decoder and some useful signals.

For a weird historical reason, the 74LS189 inverts its contents. It is an interesting design fluke of how TTL-logic transistors used to be manufactured in the 80s and 90s, and how it was sometimes practical for driving LEDs directly without current limiting resistors, but for our purpose we will have to add two 74LS04 hex inverting gates to get back our original data.

There actually exists a similar version to our RAM chip, the 74LS219 that doesn't invert the outputs, but is more expensive and much harder to find, so we will not use that one in this build.

4.4.2. Build process

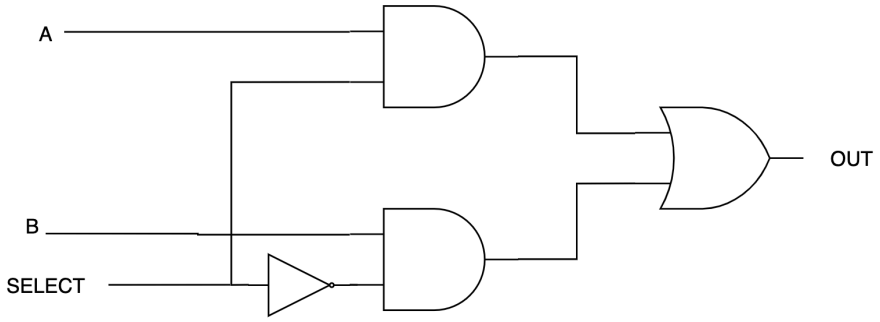
We start by connecting the outputs of the RAM chips to the inverters. We will use four out of six inverters on each 74LS04 for equal load balance and leave 2 unused. This should help with wear and tear and with heat dissipation. After connecting power and ground to everything we will also tie the $\overline{CS}$ on pin 2 low for both RAM chips, because we want to have our chip always enabled. Similarly, since we would like to write to both chips at the same time, we join pins 3 together, which will form our write enable ($\overline{WE}$) signal. For the 3-state functionality we are going to add another 74LS245, as we have done for all modules that need to connect to the bus. Then we connect the output from the inverters to the input of the 74LS245. Next, we tie our four address lines A1-A4 together on both chips, because they will both accept the same address, but one RAM chip will store the lower 4 bits and the other the upper 4 bits of our byte. Here is the finished RAM module:



4.4.3. Program mode & Run mode

It might be a good time to explain how we are going to insert data into our memory. We would like to have 2 modes of operation: program mode and run mode. When we are in program mode, we will use a 4-way dip switch to manually select an address and an 8-way dip switch for our instruction. We will have to manually insert instruction at address 0, then at address 1 and so on. When we flip a switch to enter run mode, the CPU starts executing instructions starting at address 0 and disregards any input from the switches. That means that we will have to build some selection logic that allows us to choose

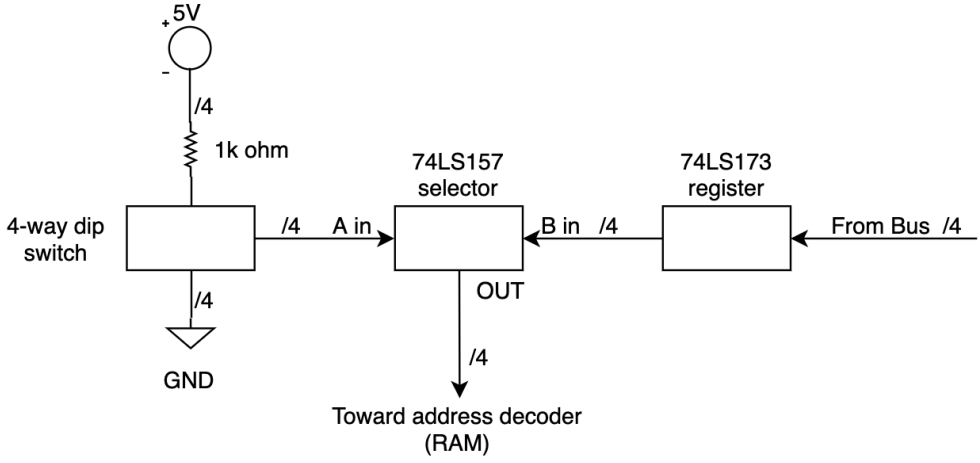
between the input from the dip switches or from the bus. Basic logic for selecting one out of 2 bits can be described with the following logic:



When SELECT is high, the OUT ≡ A and similarly when SELECT is low, OUT ≡ B. This logic gives us what we want, but we would need 12 copies (4 address lines + 8 data lines), each having 4 gates, so 48 gates in total, which would be about 15 chips. We are going to choose a more practical approach and use the 74LS157 Quad 2-Line to 1-Line Data Selectors/Multiplexers. The logic diagram in the datasheet of this chip has basically four identical copies of the diagram above, so we will need 3 of these chips in total, which is much less than 15.

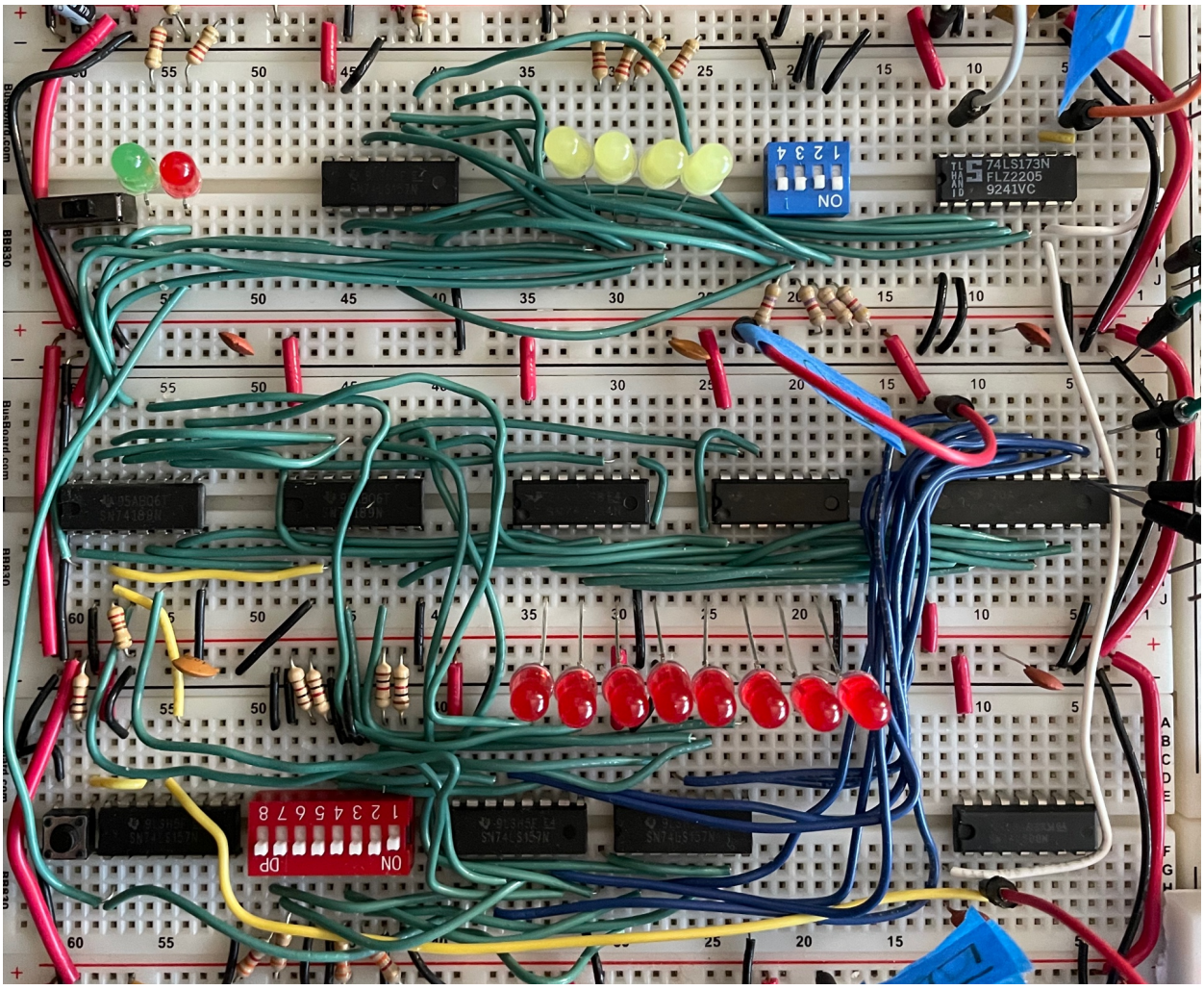
4.4.4. Adding selection logic

We start with a previously used 74LS173 4-bit D-type register. This will store the address from the bus that can be fed to the address lines of our RAM upon request. Its 4-line output will be connected to the B input of our 74LS157 selector. The A input of the selector will go to the 4-way dip switch. Here is an alternation to Ben Eater's design. He explains that the selector has internal pull up resistors, so we can just selectively connect the dip switches to ground. If the switch is on, it connects to ground and if the switch is off, the selector pulls the voltage up for logical high. Well, this didn't work for me at all. I spent several hours troubleshooting and I realized that when the switch was in its off position, the voltage was floating, and I was getting random values that were changing when I was waving my hands close to the circuit. I had to add my own 1kΩ pull up resistors, because the ones in the selector just weren't enough. Here is a block diagram:



Now we can add a single pole double throw switch that will toggle between the A and B inputs of the selector. We are going to add a green LED to denote Run mode (B input active) and red LED to denote Program mode (A input active).

This principle is then repeated for our 8 data lines with a few minor differences. We need to use an 8-way dip switch and 8 1kΩ pull up resistors in the same configuration to avoid floating voltage. Instead of the 74LS173 register we will use the 74LS245 8-bit transceiver. We didn't have to use it for the address because the address was only read from the bus, but here we will need to also write on the bus. Next addition is to add a button that will toggle our memory write. After we program the address and our instruction on the dip switches, we press a button, and the byte long instruction saves to our RAM. In the Run mode, this write enable signal will be supplied from our signal logic. However, we want to write only on the rising edge of the clock, so we will use a 74LS00 NAND gate to combine those two. The reason for using a NAND gate is that our write enable signal is an active low, so we won't have to invert it afterwards. Here is the finished RAM module with the selector logic:



4.5. PROGRAM COUNTER

We have to have a notion of which instruction we are about to execute. This is the job for a program counter, which we can alternatively name *Instruction Pointer*, as their function is identical. Our program will start executing instructions from address 0 and after we are done, we will increment our counter and then fetch the next instruction from address 1. During a jump instruction, we will set the program counter to a specific address, so that we fetch our next instruction from that memory place.

Binary counters are relatively easy to implement in hardware, all we need is to cascade some JK flip flops. In our design we are going to use a 74LS161 Synchronous 4-bit counter alongside the 74LS245. Similarly to previous register designs (the program counter is sort of a register itself), we will put 4 LEDs in between the chips so we can see the value of the counter at all times. We are interested in a few signals that these chips provide us: we need to load the value from the bus into the counter (for jumping) and the 74LS161 has a Load signal. We will also need to isolate the Counter Enable signal, because we do not want to increment on every clock signal, only sometimes. Finally, we will use the Counter Out signal to output the counter value onto the bus, which will later become a part of our instruction fetching.

4.6. OUTPUT MODULE

4.6.1. Design overview & motivation

Our 8-bit computer needs to display 8-bit numbers, ideally in decimal form, not binary. Here we are going to implement 2 display modes – signed and unsigned numbers. In the signed range we are going to output numbers from the closed interval [-128; 127] and in the unsigned range from [0; 255]. We should also have an easy way to switch from one mode to the other. We can see that we will need 4 seven segment displays – one for the optional sign and three for the three digits. One way to do this is to have one EEPROM for each segment. This naïve approach is the simplest, but also the most wasteful and we can do better. Instead, we are going to use only one EEPROM and multiplex through it. That means we will need a 2-bit counter. Our EEPROM input will then consist of this multiplexed counter and the number to display and output the corresponding signals for each seven-segment display. Here is an example for number 123 (01111011_2):

A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0		D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	1	1	1	1	0	1	1		0	1	0	0	1	1	1	1
0	0	1	0	1	1	1	1	0	1	1		0	1	0	1	1	0	1	1
0	1	0	0	1	1	1	1	0	1	1		0	0	0	0	0	0	1	1
0	1	1	0	1	1	1	1	0	1	1		0	0	0	0	0	0	0	0

We can see that the EEPROM replaces a gigantic combination logic table that in principle could be represented by other logic gates. The first line of the table corresponds to digit 3, the second to digit 2, the third to digit 1 and the last one to sign, which we don't need in this case.

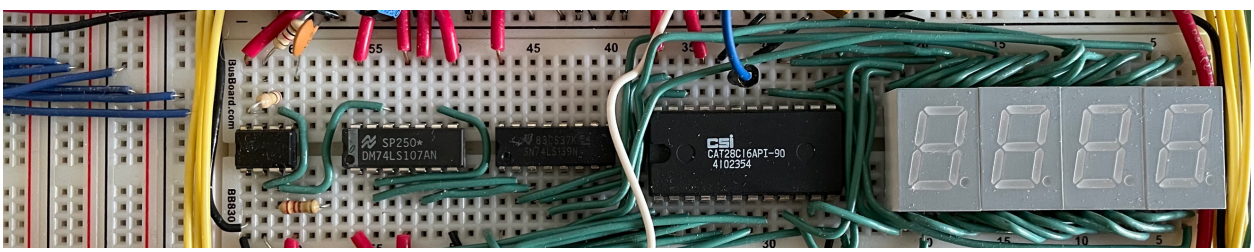
The 28C16AN EEPROM supports 11 address lines, and we will use all of them. The first 8 lines (A0-A7) will represent the binary number to output, in our case 123. The address lines A8 and A9 represent the multiplexed 2-bit counter and the address line A10 will signal whether we want to treat the output as signed or unsigned. In the case of 123 it will make no difference, because it would be displayed the same in both cases. The outputs D0-D7 correspond to which segments need to be turned on to represent the digit. To make it all work we need one more addition – we need to select which display we want to output to. If we only have what I described before, our 4 displays would all display 3 – 2 – 1 – (blank) in a cycle, from the least significant digit to the most significant one. That is not quite what we want, we want only one display to be active at the time. We will fix this problem by decoding the multiplexed counter with a 74LS139 decoder and use it to drive the cathode of the seven segment displays. This way only one digit will be shown at a time and the other 3 displays will be off. Even though we are showing one digit at a time, if we make our output module clock fast enough, the persistence of vision will ensure that we will see a 3-digit number without any flicker. Don't forget that this clock will be independent of the actual CPU clock and will be in the order of a few kilohertz.

4.6.2. Building the output module

We are going to start with the 555-timer for our clock in the astable configuration. To set our clock frequency we are going to put 0.01 μF capacitors on pins 2 and 5 to ground. We will connect pins 6 and 2 together and put a 100k resistor across pins 6 and 7. Finally we are going to put a 1k resistor between pin 7 and 5V to obtain the desired timing. Pin 3 of the timer will drive our 2-bit counter, which we decided to implement by using a 74LS109 Dual Master-Slave JK Flip-Flops. The output of the chip will split; one part goes to the EEPROM and connects to our A8 and A9 datelines and the other part will go into the 74LS139 decoder. The 4 decoded outputs will respectively drive the cathodes (pin 3) on each seven-segment display. The rest of pins are connected to the EEPROM outputs D0-D7.

Next addition is to make our output module behave like a register, since we would like to remember the value and keep displaying it as long as we are requested to display some other value. To change things up a little bit, instead of using two 74LS173 and a 74LS245, we are going to use one 74LS273 Octal D Flip-Flop chip. The biggest reason, apart from saving on parts, is that we will only be reading the value from the bus and storing it in the output register, we will not need to output back onto the bus. The purpose is exactly the same, to latch the value from the bus.

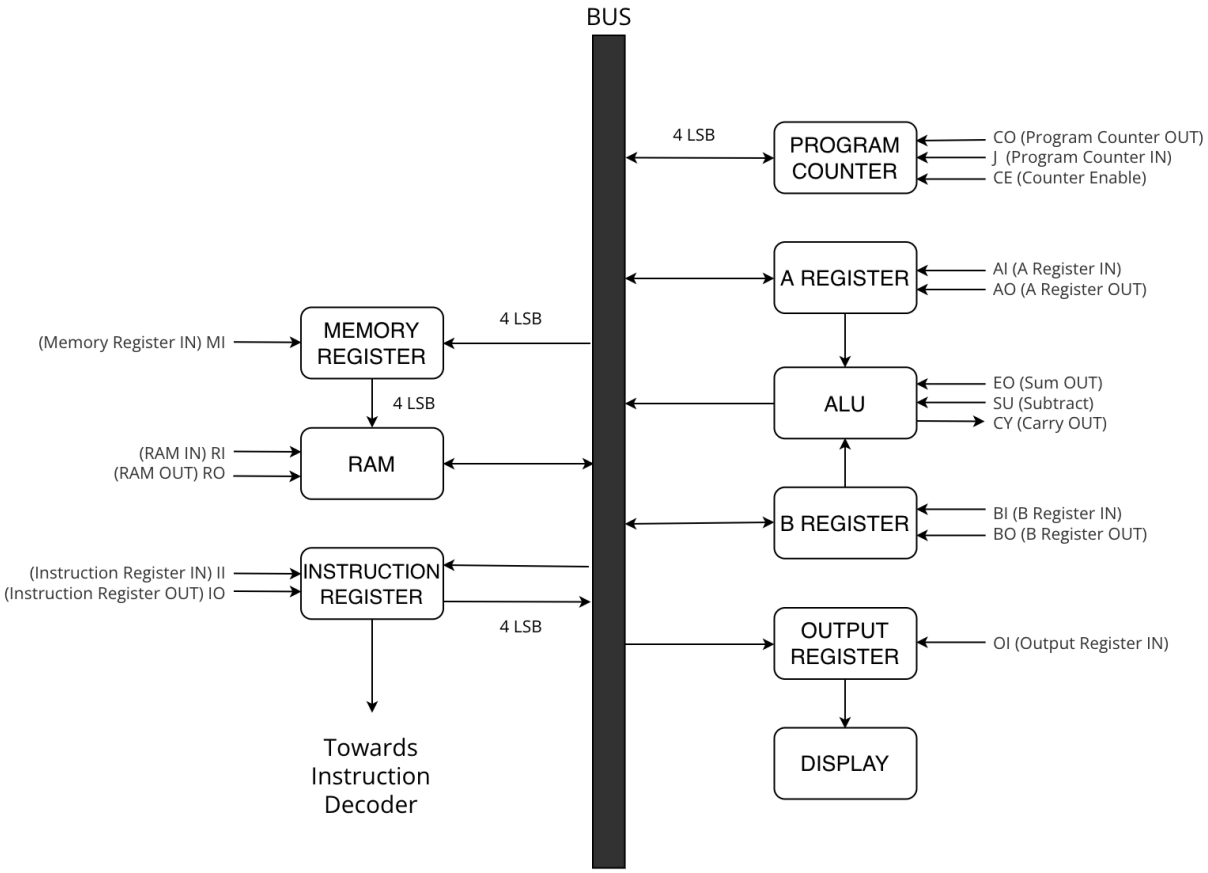
Finally, we need to tie the $\overline{WE}$ of our EEPROM to 5V since we never want to write to it. We also must set the output enable and chip enable pins to ground since we want to always output and always have the EEPROM enabled. Here is the finished module:



4.7. CONTROL LOGIC

4.7.1. Design Overview

Let's have a look at the block diagram of what we have built so far (note that some arrows are unidirectional, and some are bidirectional – they represent the possible data flow in our design):

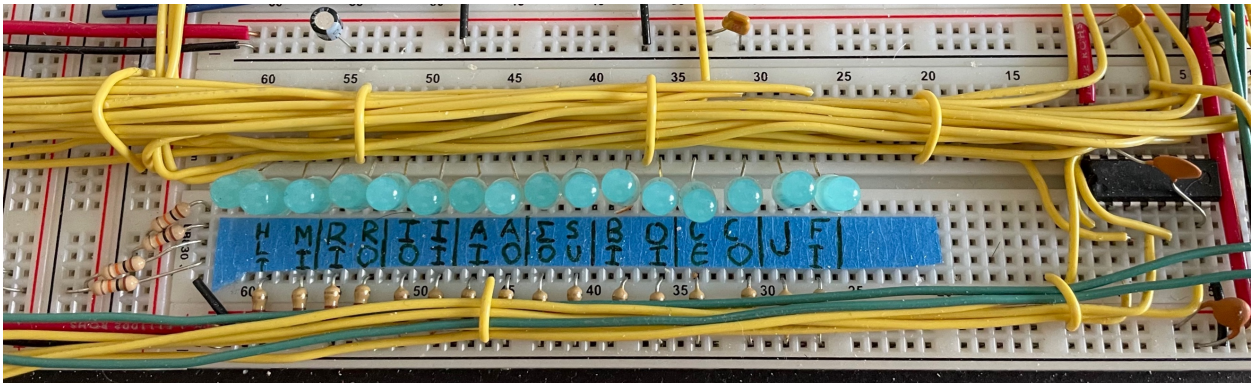


We have isolated the necessary signals from each module. We can manually set them high and low and manually pulse the clock to get an idea of how we can execute simple programs. Ideally, we would like to automate this process. This is where the Control Logic comes into play. It will consist of an instruction decoder, combination logic and a stepping counter for microinstructions. For each instruction (like LDA, SUB and so on) we will execute several microinstructions. In this design we will have 5 microinstruction (steps) per instruction, but we could accommodate up to 8 if we decide to enhance the capabilities. One step corresponds to one clock pulse.
The first two microinstructions will always be the same and will perform fetching from RAM into the Instruction Register. From there we can determine the Execute part (last 3 steps), which will depend on the actual instruction. Each microinstruction just sets different combination of our signals high and low, so we will be reading to and from different modules, depending on the instruction. Here is a table of all supported instructions. We have a space to add 2 more if we desire.

Instruction Opcode	Instruction Symbol	Instruction Name	Instruction Microcode					
			Fetch		Execute			
			Step 1	Step 2	Step 3	Step 4	Step 5	
0000	NOP	No Operation	MI | CO	RO | II | CE				
0001	LDA	Load A	MI | CO	RO | II | CE	IO | MI	RO | AI		
0010	ADD	Add from Address	MI | CO	RO | II | CE	IO | MI	RO | BI	EO | AI | FI	
0011	SUB	Sub from Address	MI | CO	RO | II | CE	IO | MI	RO | BI	EO | AI | SU | FI	
0100	STA	Store A to Address	MI | CO	RO | II | CE	IO | MI	AO | RI		
0101	LDI	Load Immediate	MI | CO	RO | II | CE	IO | AI			
0110	JMP	Jump	MI | CO	RO | II | CE	IO | J			
0111	JC	Jump on Carry	MI | CO	RO | II | CE				
1000	JZ	Jump on Zero	MI | CO	RO | II | CE				
1001	ADI	Add Immediate	MI | CO	RO | II | CE	IO | BI	EO | AI | FI		
1010	SUI	Subtract Immediate	MI | CO	RO | II | CE	IO | BI	EO | AI | SU | FI		
1011	OAH	Out - Address - Halt	MI | CO	RO | II | CE	IO | MI	RO | OI	HLT	
1100								
1101								
1110	OUT	Out	MI | CO	RO | II | CE	AO | OI			
1111	HTL	Halt	MI | CO	RO | II | CE	HLT			

4.7.2. Normalizing Signals

Some of our signals are active high and some of them are active low. This design includes a blue LED that indicates whether the signal is active. It also makes our code to program the EEPROMs easier, since we can then assume that high means active and low means not active. That's why we will need two 74LS04 Hex inverters, because out of our 16 signals, 11 of them are active low. We are going to label each signal, have a corresponding diode for it and if it is an active low, we will run it through the inverter.
The finished signals look like this:

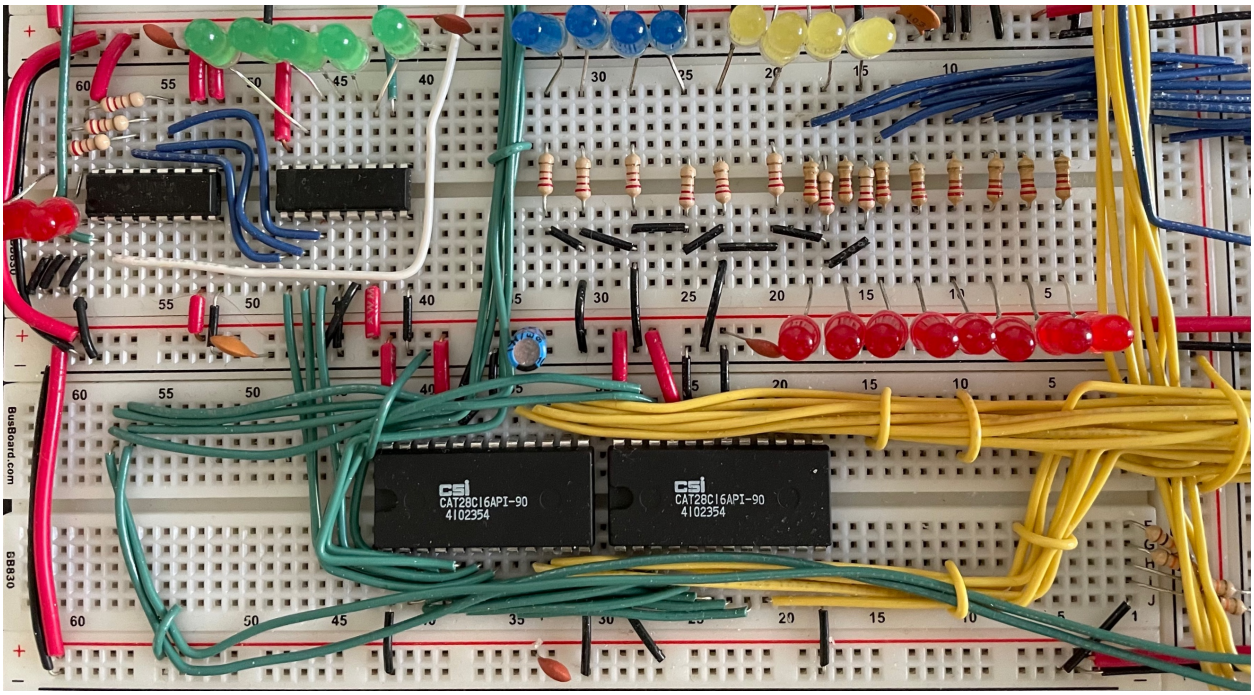


4.7.3. Step counter and EEPROMs

As mentioned in the Output Module part, the 2816AN EEPROM has 11 input address lines and 8 output lines. That means it can store 2^{11} 8-bit words. Since we have 16 signals, we will need two EEPROMs for our Control Logic. One of them will drive the first 8 signals and the other one the remaining 8. We will not need all 11 inputs – in this design we only need 9: 4 bits that correspond to the instruction opcode, 3 bits that correspond to our multiplexed step and 2 bits for 2 flags from the flags register that we are going to build in the future. That still leaves us with 2 more unused inputs that we can

take advantage of in the future. The EEPROMs again just replace a gigantic combinational logic table and they directly translate the input state into an output state. After connecting 5V and ground we need to tie the $\overline{WE}$ of our EEPROMs to 5V since we never want to write to them. We also must set the output enable and chip enable pins to ground since we want to always output and always have the EEPROM enabled, exactly as we did in the Output Module.

For the counter we are using 74LS161 Synchronous 4-bit counter alongside a 74LS139 3-to-8-line demultiplexer. We are using the decoder to have some green LEDs that display the current step, otherwise it is not needed. When we count to 5, we reset the counter back to zero. The reason for doing so is that none of our instructions needs more than 5 steps. If we design more complex instructions, we can have more steps per instruction. Here is the finished counter, decoder and EEPROMs:

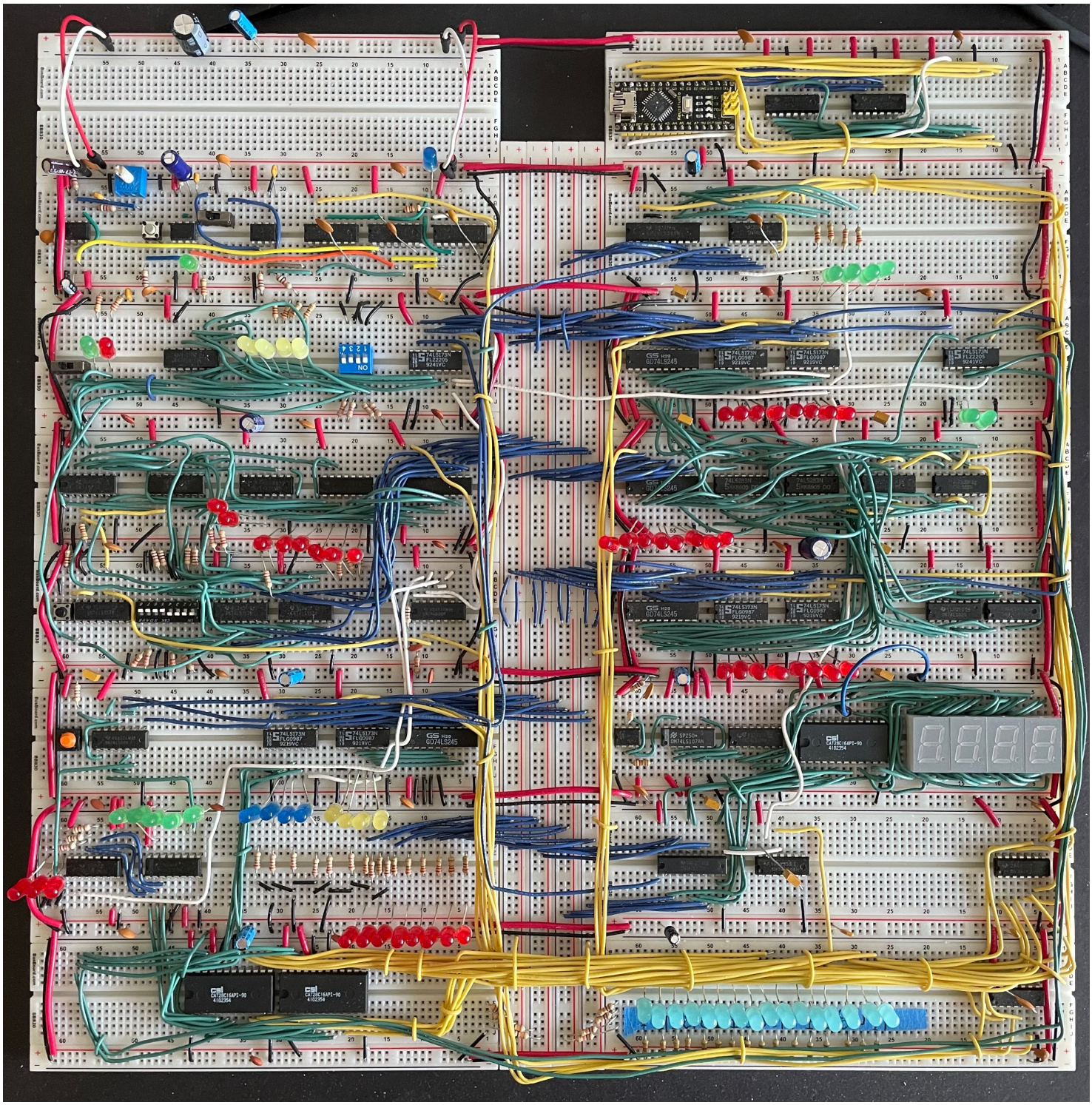


(Note that this picture has more things in it than just the counter, decoder and EEPROMs. The 4 blue LEDs correspond to the instruction opcode, 4 yellow LEDs correspond to the instruction payload and together they form the instruction register. They all have a 220-ohm current limiting resistor to ground. There is also a row of 8 red LEDs that correspond to the current value on the bus, again with 220-ohm current limiting resistors. The 3 red LEDs in the top left denote the multiplexed counter and the 5 green LEDs represent the demultiplexed counter. The 2 EEPROMs are in the middle.)

4.7.4. Flags Register

To make our CPU fully Turing Complete, we only need to add one more thing – conditional jump. We only need to implement one type, but we are going to implement two anyways: jump on carry and jump on zero, since they are both really easy. We already have the carry flag *CY* from the ALU. For the zero flag we could use an 8-input NAND gate, but they are hard to find. Instead, we are going to use two 74LS02 Quad NOR gates and one 74LS08 Quad AND gate to represent the same logic – we want the flag to be active only when all 8 bits from the A register are zero. To store these flags, we put them into 74LS173 and since we are only using two bits, we have room for expansion. This flag register with added LEDs is then part of the input for the EEPROM, as described above.

5. FINISHED BUILD



5.1. IMPROVEMENTS DONE

My original goal was to improve the power supply. I tried many different approaches. At first, I tried using the LM7805 Voltage regulator with a heatsink, since I had a 9V power supply at home. However, during the build I discovered that it couldn't maintain 5V when all components were connected. It should be noted that this design draws about 12W of power (so about than 2.2A of current), which the regulator couldn't handle. I also tried using a USB phone charger with a USB breakout board, but the charger I had at home again couldn't supply the amperage needed. In the end I decided to simply buy a 5V 3A barrel jack power supply and I had no power issues whatsoever. I added many wires to distribute the voltage to all parts of the build. Since I didn't improve the power supply, I decided to improve many small things:

- I added a lot of filtering bypass capacitors across power rails and across chips to help smooth out the current draw, since this many bread boards have a non-negligible resistance
- I added several ceramic capacitors to help with ringing, especially on the chips in the clock module and other critical parts
- I added many pull resistors to help with floating voltages, which the original design omits
- I added 3 instructions that the original design doesn't have
- I wrote an assembler in C that converts the instructions into the machine code: https://github.com/TheTask/8Bit-Assembler
- I wrote several programs of my own that fit into the 16B of memory
- I slightly improved the power distribution module

5.2. SAMPLE PROGRAMS

Computing and displaying the Fibonacci's sequence:

```
LDI 1
STA 14
LDI 0
STA 15
OUT
LDA 14
ADD 15
STA 14
OUT
LDA 15
ADD 14
JC 0
JMP 3
HLT
```

Here is a video of this program: https://www.youtube.com/watch?v=TjeFRSD93Ko

Computing and displaying powers of 2:

```
LDI 1
STA 15
LDA 15
OUT
ADD 15
JC 0
JMP 1
```

Multiplying X and Y and displaying the result (as long as the product is less than 256):

```
LDA 14
SUB 12
JC 6
LDA 13
OUT
HLT
STA 14
LDA 13
ADD 15
STA 13
JMP 0
-
ONE
ZERO
X
Y
```

6. CONCLUSION

This project taught me a lot. I improved my knowledge of electronics, learnt how to troubleshoot with a multimeter and solidified my understanding of low-level assembly. It also taught me a lot of patience with wire cutting, stripping, measuring, and connecting everything. It took about 150 hours to complete.

I would like to thank my supervisor Prof. Joseph Vybihal for the support and for the supervision. Even though this project has limited memory, it performs exactly as a modern CPU within its constraints. It is a great introduction into low-level architectures and a great learning resource with an active community.