# WAWA Final Report: ROMULUS I

## I. Statement of Work

### A. August Bresnaider

Lead software design. Directed instruction set architecture and finite state machine design. Proved Turing completeness of architecture. Helped design PCBs and pick parts to use. Played role in debugging and determining the capabilities of the hardware through testing. Designed simulator, assembler and compiler, and wrote code to specify FSM and ALU flash, as well as Arduino code to flash the chips.

### B. Wolfgang Ploch

Lead hardware design. Participated in instruction set architecture design and high level architecture design. Designed circuitry, designed PCB, and wrote hardware description sections for boards 1, 2, 3, 5, and 6. Assisted in circuitry design for board 4. Soldered components on all 6 boards. Lead case and control panel design. Assembled entire device. Troubleshot hardware and implemented hardware solutions during testing. Designed and provided teletype peripheral and wrote driver programs for it. Wrote cost and physical constraint sections of the final report.

### C. Will Rimicci

Participated in instruction set architecture design and high level architecture design. Designed circuitry, designed PCB, and wrote hardware description section for board 4. Wrote code to generate control signals FSM flash data. Rigorously tested machine code simulator and assembler. Wrote several test programs to be run on simulator and computer. Troubleshot hardware and helped design hardware solutions during testing. Recorded tests for debugging purposes. Designed circuitry, designed PCB, and soldered general input/output peripheral. Wrote background, societal impact, intellectual property, full system test, and final results sections of the final report. Helped keep team focused on deliverables when due over the course of the semester.

### D. Austin Chappell

Assisted in the creation of ISA, software tools, and testing processes. Wrote the comprehensive test program for the whole project. Ensured timely and quality completion of project deliverables. Wrote standards, timeline, insights, and compiled references in this report. Helped keep team focused on deliverables when due over the course of the semester.

## II. Table of Contents

### Contents

***Abstract*—Our project revolves around the design and fabrication of a 16-bit CPU and supporting memory using Flash memory, 74XX series logic chips, and passive components. Aimed to be a educational device for students to learn computer architecture, digital logic, machine code, and electronics, the Romulus I is equipped with LEDs on the internal registers, 7-segment displays, debugging tools, and multiple clock speeds. The scope of this project also includes a compiled language and an assembler providing increasing layers of abstraction, allowing for either a top-down or bottom-up approach to teaching about computing systems and organization.**

## III. Background

We chose this project for a number of reasons. First, we were inspired by our experiences in other classes like Computer Systems and Organization and Digital Logic Design. After using the Digital open-source logic simulator [1] and the ToyISA simulator [2], both of which are virtual tools, we felt that we could create a physical, hands-on teaching tool that would help students understand computers in a new way. Second, we wanted to see if we could use what we had learned in those classes through lectures and homeworks to make something new. Finally, we just thought it would be a fun and interesting challenge.

Throughout this project, we used many techniques we learned in our classes. The general structure of the computer, having a control signals FSM, an ALU, a program counter, and memory, was material from Digital Logic Design. In that class, we created a programmable computer with those blocks in Digital [1] (though most of the designs of the blocks themselves were different from the approach we were taught in that class). The design of the instruction set architecture and assembly were from Computer Systems and Organization 1. The knowledge on how to create an assembler and compiler were from Computer Systems and Organization 2. Finally, the knowledge on how to create a PCB layout, order parts, and solder circuits were from the ECE Fundamentals series.

Though this project was based on and inspired by a number of past works, it is a new and distinct product. As mentioned above, we took inspiration from the virtual tools we used in our classes, like Digital [1] and ToyISA [2]. While these are fantastic tools for learning, they are only virtual programs, and do not provide a physical, interactable model from which to learn. Another similar past work is Ben Eater's 8-bit breadboard computer [3]. In his video series, Ben Eater creates a full-functional, programmable 8-bit computer on a set of breadboards using only simple logic gates. It has a clock, registers, ALU, RAM, program counter, and control logic. While this project is similar to our own, it is built on a set of breadboards, meaning the circuits are not as permanent as ours made from soldered PCBs, and it is much less versatile than ours: it is only eight bits and there are only two general purpose registers, compared to our 16 bits and 16 general purpose registers. Our work is different from any projects before it.

## IV. Project Description - Specifications

### A. Instruction Set Architecture

*1) Overview:* The instruction set architecture (or ISA, for short) acts as a specification that defines the operations the CPU is capable of, as well as their binary representations. By giving an operation a mapping, we include it in the capabilities of the CPU. Designing an ISA is a balancing act: If an ISA is too simple, common operations can become arduous (i.e., not including A & B requires the programmer to do $(\overline{A}|\overline{B})$ any time a bitwise AND is needed), or in the worst case, the CPU may not be Turing complete. On the contrary, if the ISA is overly complicated, it may require an overly complicated hardware implementation. To avoid the negative effects of these two poles, our ISA should be as follows:

- Turing complete

If our ISA is not Turing-complete, there is a strict and provable upper bound on the computational power of our hardware, which means our CPU will not be able to compute any given algorithm.

- Logically comprehensive

While the bare minimum number of operations needed to make a Turing-complete ISA is remarkably small (by our count, it is roughly 5 or 6), it's to include more instructions for ease of use purposes, much like the bitwise AND example above, to avoid frustration and code repetition on the programmer's part.

- Easy to code / represent in hexadecimal

As one of the revisions to ToyISA that we aim to implement, the bitcode should be intuitive and require minimal bit-fiddling on the programmer's part. ToyISA had 8-bit instructions, with the most significant 4 bits representing opcode, the next 2 representing rA (parameter 1), and the least significant 2 representing rB (parameter 2) or an optional set of selector bits for unary operations. When programming using bitcode, it is often a hassle having to compute the lower hexadecimal digit when changing rA or rB, so our ISA should split each 16-bit value into groups of 4 or 8 wherever possible to avoid this.

- Able to be utilized in higher-level abstraction

The ISA should be fully functional when programming using bitcode, but there should also be instructions geared towards higher-level concepts. For example, while a stack pointer and instructions incrementing and decrementing the stack is not strictly required, it gives way to creating a call stack and allowing for the writing of functions, loops, and recursion: features only truly available in romASM and / or Hawk. The ISA we designed for Romulus I in order to solve these problems to the best of our ability is given by Table I.

As referenced in the table, we settled on using 16 instructions, which allows our opcode to fit within the most significant 4 bits of the instruction. Bits 8-11 generally act as placeholder for $R_A$, the first parameter in many operations. From there, bits 4-7 are reserved for $R_B$, the second parameter for many operations, and finally, bits 0-3 are usually reserved

for $R_Y$, the register in which to store the result. Each group of 4 instructions are similar to each other, and so the next 4 sections will discuss the "Move", "Bitwise", "Arithmetic / Logical", and "Control Flow" instruction groups.

*2) Move Instructions (Opcodes 0-3):* The first 4 opcodes deal with moving values into and between registers, as well as interacting with memory. Opcode 0 deals with register-register moves, which we intentionally mapped as opcode 0, so that the bitcode "0000" refers to moving $R_0 \rightarrow R_0$, or essentially a NOP instruction. Thus, if the instruction RAM is initialized to all 0's, it will default to NOP instructions, instead of a random arithmetic or control flow operation. Instruction 1 moves an immediate value into a register, this being our only I-type instruction (as opposed to the 15 R-type instructions), in which the opcode and registers are in the first 16-bit value, and the second 16-bit value specifies the immediate value to load. We originally wanted to restrict the size of each instruction to 16 bits, and limit the size of immediates to 12 bits, but after discussion regarding the simplicity of multi-address instructions within our control signals logic, as well as the fact that 12-bit immediates would limit our addressing space to 4096 addresses, we decided to go forward with a true immediate-type instruction. This instruction is the only way to load immediate values, a decision we made to keep the total number of instructions within 16 to fit into a 4-bit opcode. The next two instructions (2 and 3) involve interfacing with memory, or more specifically, our data RAM. If we were to simulate a Turing machine, these instructions would be akin to reading and writing from the tape.

*3) Bitwise Instructions (Opcodes 4-7):* The next four instructions represent the four most commonly used bitwise logical operations in programming. While NAND is expressively complete (meaning, every logic gate can be derived from repeated applications of NAND), the NAND is not often used in programming, and operations like OR and XOR would require lots of register moves and instructions if only NAND were available. By selecting AND, OR, NOT and XOR, we only leave out NAND, XNOR and NOR, all three of which are fairly uncommon, and can be computed with a maximum of two operations (and no register shifting) if needed.

*4) Arithmetic / Logical Instructions (Opcodes 8-B):* When designing our ISA, we knew that we would be using parallel flash memory to implement our ALU. While this provides its benefits, namely the smaller ALU footprint and the ability to reconfigure, the drawback of this is that, due to the small number of address pins in parallel flash, each bit of an arithmetic operation may only rely on the opcode, the bit above it, and the bit below it. This works for an operation like addition or subtraction, where the operation happens in a rippling fashion (that is, carry and borrow bits are always adjacent to the bit in question) but this does not work for an operation like multiplication, where each bit of parameter A needs bits from parameter B that are not necessarily adjacent. Thus, we decided for our main arithmetic / logical operations to be addition, subtraction, logical NOT, and a logical shift right. Addition and subtraction are obvious choices to include, and we made a conscious decision to include both even though subtraction can be represented by an addition and a negation. This was another decision we made to improve the ease of use of our ISA. We the implemented logical NOT, which sets the least significant bit to 1 if the entire value is zero, and zero if the entire value is non-zero. While this may at first seem like a non-inline instruction, we planned to implement this instruction in a rippling fashion, where we begin with the most significant bit of the parameter, and use borrow-out inputs to signify if any of the numbers are non-zero. This ripples down to the least significant bit, which becomes zero if the borrow out is non-zero, and vice versa. Our last instruction of this section is a logical shift right, which again takes advantage of the borrow-out behavior to set subsequent bits. We discussed implementing an arithmetic right shift (i.e. sign-extending), but it is our belief that a generic operation that shifts the bits down is much more useful than one that acts on two's complement numbers. In order to abide by the in-lining rule though, it was necessary to require that right-shift only shifts 1 over. While we would have liked to implement a logical shift left, we decided that it would be strictly less valuable than any of the instructions currently in our ISA. Further, moving all digits to the left 1 place in binary is equivalent to multiplying that number by 2, or adding it to itself. Thus, using a little bit of creative bit-fiddling, the programmer can implement left-shift using addition if they so please. In fact, in our assembly, there is a left-shift instruction that takes advantage of this trick, abstracting this shortcoming away from the user.

*5) Control Flow Instructions (Opcodes C-F):* Instructions 12 and 13 are meant solely to interact with stack RAM. This stack can be utilized as a data structure, loading and storing values from registers, or to load and store values from the program counter, or in other words, jump program execution to another location. This is especially useful in higher levels of abstraction for calling and returning from functions, where a call can be represented as pushing the value to return to, and then unconditionally jumping to an arbitrary location, and a return can be represented as popping the value at the top of the stack into the program counter (or PC). It should be worth noting that this is why pushing PC actually pushes PC + 2, since a call takes 2 instructions, and the user would want to return to the next instruction after the call. In order to differentiate between pushing / popping with the PC or with registers, instructions 12 and 13 have an $Op_2$ bit in bit 7, where setting it interacts with PC, and resetting it interacts with the register designated in $R_A$ or $R_Y$, depending on the instruction. Instruction 14 handles unconditional jumps, which sets the value in PC to that of the defined register. This similar format is followed in instruction 15, where a jump is only initiated if a predicate is true, as specified by the flags in the lowest 4 bits of the instruction, and the mapping in Table II(note: X refers to don't-care bits; their values can be 1 or 0). Most assembly languages have something akin to jumping based on an arbitrary predicate (i.e. a ¿ b), amd while we would have liked to have a similar instruction, it was unable to make it within our 16 instructions.

TABLE I
INSTRUCTION SET ARCHITECTURE MAPPING

| Opcode (Op) | Instruction | Registers | Bits (MSB ... LSB) | | | |
|---|---|---|---|---|---|---|
| 0 (0000) | Register to Register | $R_A \rightarrow R_Y$ | Op (4) | $R_A$ (4) | X (4) | $R_Y$ (4) |
| 1 (0001) | Immediate to Register | I $\rightarrow R_Y$ | Op (4) | X (8) | | $R_Y$ (4) |
| | | | Immediate (16) | | | |
| 2 (0010) | Register to Memory | $R_A \rightarrow M_{RB}$ | Op (4) | $R_A$ (4) | $R_B$ (4) | X (4) |
| 3 (0011) | Memory to Register | $M_{RA} \rightarrow R_Y$ | Op (4) | $R_A$ (4) | X (4) | $R_Y$ (4) |
| 4 (0100) | Bitwise OR | $R_A$ —— $R_B \rightarrow R_Y$ | Op (4) | $R_A$ (4) | $R_B$ (4) | $R_Y$ (4) |
| 5 (0101) | Bitwise NOT | $\sim R_A \rightarrow R_Y$ | Op (4) | $R_A$ (4) | X (4) | $R_Y$ (4) |
| 6 (0110) | Bitwise AND | $R_A$ && $R_B \rightarrow R_Y$ | Op (4) | $R_A$ (4) | $R_B$ (4) | $R_Y$ (4) |
| 7 (0111) | Bitwise XOR | $R_A \hat{} R_B \rightarrow R_Y$ | Op (4) | $R_A$ (4) | $R_B$ (4) | $R_Y$ (4) |
| 8 (1000) | Add | $R_A$ + $R_B \rightarrow R_Y$ | Op (4) | $R_A$ (4) | $R_B$ (4) | $R_Y$ (4) |
| 9 (1001) | Subtract | $R_A$ - $R_B \rightarrow R_Y$ | Op (4) | $R_A$ (4) | $R_B$ (4) | $R_Y$ (4) |
| A (1010) | Logical NOT | $!R_A \rightarrow R_Y$ | Op (4) | $R_A$ (4) | X (4) | $R_Y$ (4) |
| B (1011) | Logical Shift Right | $R_A >> 1 \rightarrow R_Y$ | Op (4) | $R_A$ (4) | X (4) | $R_Y$ (4) |
| C (1100) | Push | ($R_A$ or PC+2) $\rightarrow$ Top stack | Op (4) | $R_A$ (4) | Op2 (1) | X (7) |
| D (1101) | Pop | Top stack $\rightarrow$ ($R_Y$ or PC) | Op (4) | X (4) | Op2 (1) | X (3) $R_Y$ (4) |
| E (1110) | Unconditional Jump | $R_A \rightarrow$ PC | Op (4) | $R_A$ (4) | X (8) | |
| F (1111) | Conditional Jump | If ($R_B$ ?? 0):$R_A \rightarrow$ PC | Op (4) | $R_A$ (4) | $R_B$ (4) | Flags (4) |

TABLE II
CONDITIONAL JUMP FLAGS

| Flags | Comparison |
|---|---|
| 0000 | N/A (Instruction functions as a NOP) |
| 0001 | $R_B$ == 0 |
| 001X | $R_B$ != 0 |
| 01XX | $R_B < 0$ |
| 1XXX | $R_B > 0$ |

### *B. Finite State Machine*

We implemented the control signals finite state machine as a flash chip lookup table because we though it would be easier and take up less board space than making it out of logic chips. We needed three flash chips because we have 24 FSM outputs, and each chip only had 8 data output bits.

A flowchart of the FSM is in figure 1.

Every instruction starts with a fetch step: Einstr activates, which allows the instruction in the instruction memory at the address of the current program counter onto the bus, and Cir activates, which takes the value off of the bus and puts it into the instruction register.

For opcode 0, register-to-register assignment, we first take the value from the chosen read register (Erx) and temporarily store it in the ALU (Cp0). We then take it from the ALU without operating on it (Ealu), then store it in the chosen write register (Crx, sel1). Crx opens the register file, and sel1 chooses which part of the instruction is being used as the address. Finally, we increment the PC to go to the next instruction (pcinc), and we reset the sub-instruction counter (setsub) to reset the FSM and start the next instruction.

For opcode 1, immediate-to-register assignment, after fetching, we just increment the PC (pcinc) and read the next value in instruction memory (Einstr) into the chosen register (Crx, sel1). Then, we increment the PC (pcinc) and reset the sub-counter (setsub), same as before.

For opcode 2, register-to-memory assignment (storing), we first take the memory address from the register file (Erx, sel0) and put it into the memory address register (Cmar). With the address set, we take the data from the desired register (Erx) and put it into memory (Cdata). Finally, we increment the PC and reset the subcounter (pcinc, setsub).

Opcode 3, memory-to-register assignment (loading), is similar to opcode 2. We take the memory address from the register file (Erx) and put it into the MAR (Cmar). Then, we take the data from memory (Edata) and put it into the correct register (Crx, sel1). Last, we pcinc ans setsub as usual.

Opcodes 4 through B (11), the ALU opcodes, operate nearly identically. First, we load the data in the desired register (Erx) into the ALU's first parameter register (Cp0). If the operation being performed is binary, that is, it operates on two operands, then the second value is taken from its register (Erx, sel0) and put into the ALU's second parameter register (Cp1). (This step is skipped for unary operations, like both NOTs and shift right.) The ALU does the proper calculation according to the opcode, and the result is taken (Ealu) and placed into the desired output register (Crx, sel1). Finally, we pcinc and setsub.

Opcode C (12), push, has two different cases: pushing a register value, and pushing the PC. Pushing a register is similar to register-to-memory assignment: We take the data from the desired register (Erx) and put it into the stack at the address given by the stack pointer (Cstack). We then increment the stack pointer (Csp++) and finish the instruction with a pcinc and a setsub.

To push the PC to the stack, we actually push PC+2. This is because the only reason you would be pushing the PC to the stack is to call a function, and there needs to be room after pushing to jump to the function with a jump instruction before continuing the program. To do this, first we fetch, then we increment PC twice to make it PC+2 (pcinc x2). We then take the value from the PC (Epc) and write it to the stack (Cstack). We then decrement the PC twice to return it to its original state (pcdec x2), then we add one to the stack pointer (csp++) and pcinc, setsub as usual.

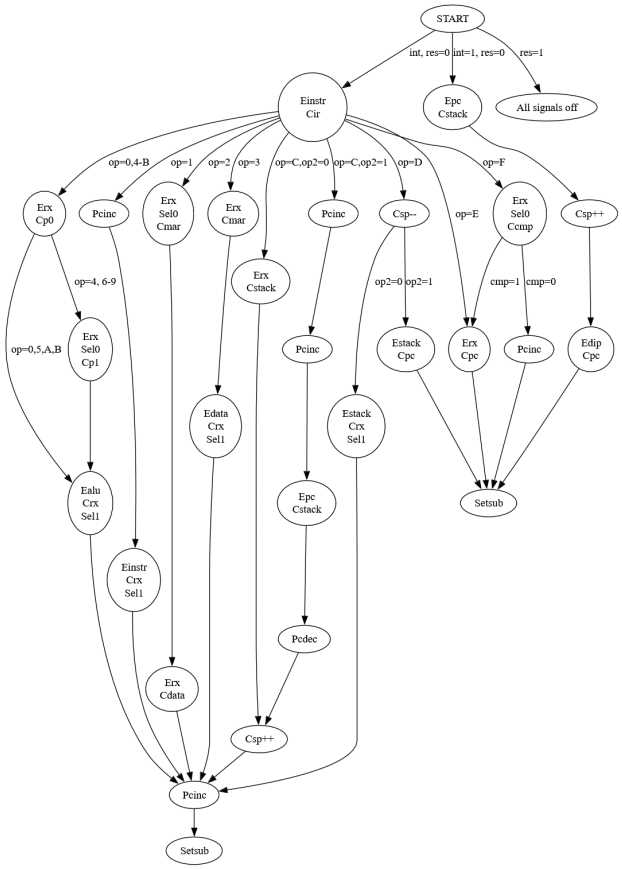


Fig. 1. A flowchart of the control signals FSM

Opcode D (13), pop, is similar to push in that it has two versions: popping to a register and popping to the PC. Popping to a register is like memory-to-register assignment: First we decrement the stack pointer (csp–), then we take what's in the stack (Estack) and move it to the desired register (Crx, sel1), then pcinc and setsub.

To pop to the PC, we first decrement the stack pointer (csp–), same as before, but after taking it from the stack (Estack), instead of putting it into a register, we just put it in the PC (Cpc). We then skip the pcinc step, since we just updated the PC by putting a new value in it instead, and setsub as normal.

Opcode E (14), unconditional jump, is very straight forward. After fetching, we take the value from the desired register (Erx) and put it into the PC (Cpc). We again do not increment the PC, and instead just skip straight to setsub.

Finally, opcode F (15) is a conditional jump. We first take the value we are comparing to from the register and put it onto the bus (Erx, sel0). The data bus is connected directly to the compare logic, which makes the comparison immediately and stores the result in a D-latch (Ccmp), using the last four bits of the instruction as flags to determine which comparison, ==0, !=0, ¿0, or ¡0, to make. Then, if the comparison is true, the FSM jumps like it would in opcode 14 by moving the value from the register (Erx) to the PC (Cpc), the setting the subcounter (setsub) as before. If the comparison is false, the FSM simply performs the usual pcinc and setsub.

Finally, the FSM can also handle interrupts from the peripheral slots and the reset button. If an interrupt is received from a peripheral slot, instead of fetching, the computer immediately takes the PC (Epc) and puts it onto the stack (Cstack) to be jumped to once the interrupt service routine is finished, then increments the stack pointer (csp++). It then takes the interrupt function location from the dip switches (Edip) and puts it into the PC (Cpc). Since the PC was updated manually, there is no need to pcinc, but the FSM does setsub as always.

If the reset button is pushed, the FSM stops all the other devices from putting anything onto the bus. Because the bus lines each have a pulldown resistor (i.e., a resistor on them connected to ground), the value of the bus is 0x0000 by default, when nothing is being actively put on it. Then, the FSM activates the write enable on every register and counter, which takes the value 0x0000 on the bus and stores it in all of them, resetting them. It does not, however, reset the instruction RAM, which means that you can run the program again without having to reupload it, and it also leaves the stack and data RAM alone (though because they are volatile memory, they are undefined when the computer starts up anyway).

## V. Project Description - Software

### A. Assembler ("romASM")

The assembler, which I will refer to as romASM is designed to act as a direct mapping to the bitcode, while also allowing for the implementation of higher level concepts like functions and loops that are arduous to implement with bitcode alone. This also allows us to fix up some of the shortcomings of our ISA. For example, as previously mentioned, there is no instruction for a left shift, whereas using the addition trick, the assembler has a built in instruction for it. Lastly, in any scenario where instructions have flags or bits specifying other nuances (push and pop have $Op_2$, conditional jump has flags), those were split into different assembly instructions. Following this, the capabilities of romASM are specified in full in Table III.

## VI. Project Description - Hardware

The hardware was designed using 74XX series logic chips for the most part. The general AND, NAND, OR, NOT, and XOR gate IC data sheets can be found in references [4], [5], [6], [7], and [8]. These chips contain 4 logic gates (6 for the NOT gate chip) in a single chip and are powered with 5 volts. The more specialized chips are discussed later in their sections.

### A. Board 1

*1) Overview:* Board 1 contains the register file and supporting circuitry The register file was chosen to have 16 registers because in order to address 16 registers, four bits are needed. Using 4 bits to select the register allows three registers to be chosen and the four-bit opcode to be specified all in one 16-bit instruction. Each register stores 16 bits. Each bit of each register was designed to be indicated with an LED. The SN74HC573 8-bit register [9] was chosen to store the contents of each register. Because each are 16-bit, two 74HC573s are needed per register. Unfortunately, the tri-state buffer built into the 74HC573 was not able to be used as we needed access to the data even when it is not active. The OE# of all the buffers were grounded so the output always reflected the data. The full Schematic of the register file board can be seen in Figure 2. It is referred to as board 1 in other sections.

*2) LED Displays:* The LEDs sourced were tested with various resistor values. A pleasant brightness was found with 820 Ω resistors. One LED draws 4.2 mA in this arrangement. This means the board could draw up to 1.075 amps from just the LEDs if 0xFFFF is stored in all the registers. To save space on the board, instead of using one discrete resistor for each LED, 8-resistor resistor arrays were sourced. These are SIP-9 components that have 8 resistors with one common pin. The anodes of each LED connect to the output of the register and the cathodes are grouped into 8s and connected to the resistor array, 2 resistor arrays per register. The common pin of each resistor array is connected to ground through a 2N3904 with a 1 KΩ resistor on the base. The other ends of the resistors are all connected to a common point. The transistor allows all the LEDs to be turned off with a single signal. This will be useful when running the computer at the highest speed.

*3) Register Design:* The outputs of the registers are connected to a 74HC541 buffer IC [10]. It is an 8-bit device so two of them are needed per register. The OE# are controlled by the decoder. All outputs of these buffers are connected by bit and then feed into two more 74HC541s to act as the final output enable. The schematic of a single register can be seen below in Figure 3.

*4) Addressing the Registers:* Two 74LS154s [?] were used to make the decoder. One is connected to the 4-bit address and its outputs connect directly to the OE# on the buffer pairs. The outputs of the 74LS154 and OE# inputs on the 74HC541 are both active low. The enable pins of the decoder are connected to ground to constantly enable the decoder. This way, one of the registers is constantly on the internal bus. The final buffer mentioned above connects the selected one to the bus using the OE# inputs on the board. The other decoder is used to write to the registers. The two WE# s connect to its two enable pins. Each output passes through a NAND gate whose other input is wired to MWE# . Pulling MWE# low causes all the registers to be written simultaneously. This is used during an asynchronous reset. Its address pins are also connected to the 4-bit address input. Each chip on the circuit has a 1 nF capacitor across its power pins to decouple the IC. The boards inputs are OE1# , OE2# , WE1# , WE2# , 4-bit address, LED CTRL. It has a 16-bit IO connection for the data bus. A trace width calculator was used to design the widths of the traces on the PCB to ensure they could supply the proper power. A simplified schematic of the control circuitry can be seen below in Figure 4.

*5) Modifications:* The PCB needed two modifications to work properly after its design and order. The first modification was the addition of the second OE# input to the board. The overall low-level architecture was not solidified before the PCB was ordered and it was not yet known that a second OE# would be needed. It was created by adding a daughter board on the PCB with a single 74HC32. The second modification was adding parallel RC networks on each of the four-bit address inputs for addressing the register. Transient issues were observed when multiple registers started to write at the same time due to errant pulses on these lines. The Parallel RC networks suppressed these and stopped the transient voltages. The resistor and capacitor values are 1 MΩ and 1 nF respectively. One of these networks can be seen in Figure 5.

### B. Board 2

*1) Overview:* The purpose of the FSM Clock and Reset Board (Board 2) is to generate the clock signals, the logic to switch between them, the reset signal, the control signals for the registers and other items on the other boards, and the timing signals used by the RAM chips. The control signals for the rest of the boards are generated using the outputs of flash memory chips, the SST39SF010A [11] to be exact. The reason that this design approach was taken was to allow for the finite state machine that runs the computer to be customizable. This proved to be a smart choice, as the finite state machine logic stored in these chips was rewritten

TABLE III
ROMASM SPECIFICATION

| Instruction | Registers | Format |
|---|---|---|
| Register to Register | $R_A \rightarrow R_Y$ | MOV rA rY |
| Label to Register | LABEL $\rightarrow R_Y$ | MOV LABEL rY |
| Immediate to Register | imm $\rightarrow R_Y$ | MOV imm rY |
| Register to Memory | $R_A \rightarrow$ M[$R_B$] | STR rB rA |
| Memory to Register | M[$R_A$] $\rightarrow R_Y$ | LDA rA rY |
| Bitwise OR | $R_A$ — $R_B \rightarrow R_Y$ | OR rA rB rY |
| Bitwise NOT | $\sim R_A \rightarrow R_Y$ | INV rA rY |
| Bitwise AND | $R_A$ & $R_B \rightarrow R_Y$ | AND rA rB rY |
| Bitwise XOR | $R_A$ ^$R_B \rightarrow R_Y$ | XOR rA rB rY |
| Add | $R_A$ + $R_B \rightarrow R_Y$ | ADD rA rB rY |
| Subtract | $R_A$ - $R_B \rightarrow R_Y$ | SUB rA rB rY |
| Logical NOT | !$R_A \rightarrow R_Y$ | NOT rA rY |
| Logical Shift Right | $R_A$ >>1 $\rightarrow R_Y$ | SHR rA rY |
| Logical Shift Left | $R_A$ <<1 $\rightarrow R_Y$ | SHL rA rY |
| Push | $R_A \rightarrow$ Top stack | PUSH rA |
| Call | PC+2 $\rightarrow$ Top stack<br>$R_A \rightarrow$ PC | CALL rA |
| Pop | Top stack $\rightarrow R_Y$ | POP rY |
| Return | Top stack $\rightarrow$ PC | RET |
| Unconditional Jump | $R_A \rightarrow$ PC | JMP rA |
| Jump If Equal | If ($R_B$ == 0): $R_A \rightarrow$ PC | JEZ rA rB |
| Jump If Not Equal | If ($R_B$ != 0): $R_A \rightarrow$ PC | JNZ rA rB |
| Jump If Greater | If ($R_B$ >0): $R_A \rightarrow$ PC | JGZ rA rB |
| Jump If Less Than | If ($R_B$ <0): $R_A \rightarrow$ PC | JLZ rA rB |

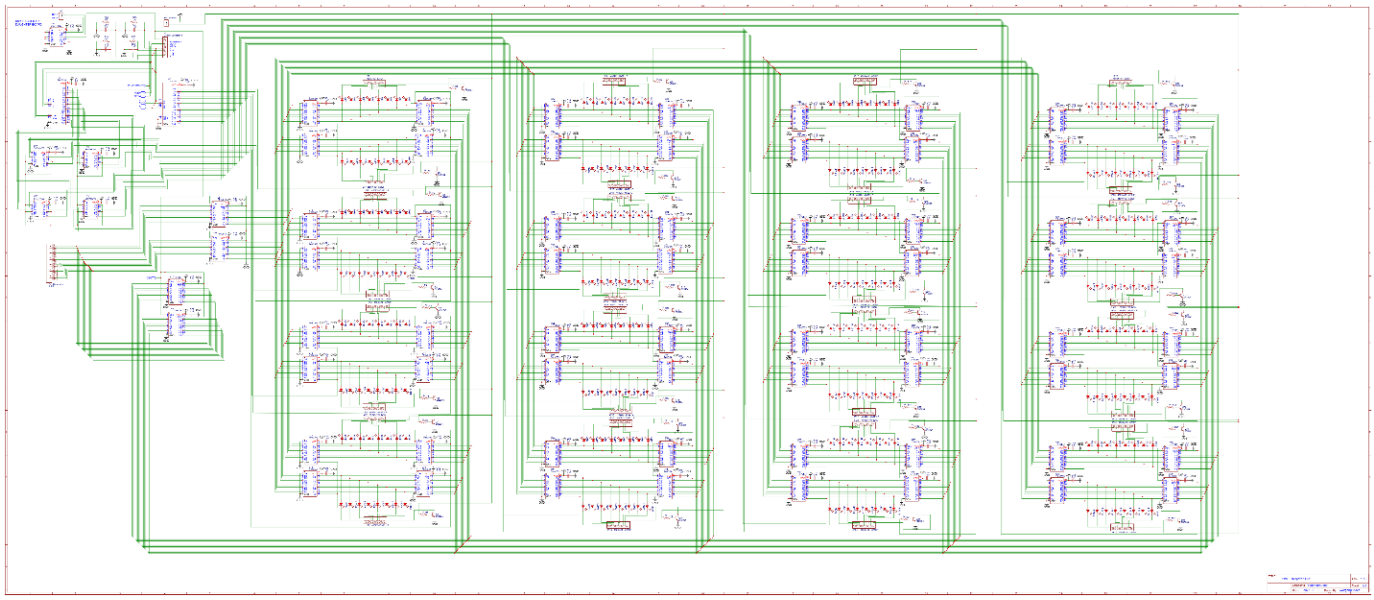

Fig. 2. Full Schematic of Register File Board

multiple times to compensate for problems discovered during troubleshooting. The full circuit can be seen below in Figure 6. It is important to note that this circuit was made using discrete gates rather than full chips. This was done to make the schematic more intelligible. The schematic used to create the PCB looks different but represents the same circuit. 1 nF bypass capacitors are placed on the power rails of every chip

*2) Clock Signals:* The board creates five different clock signals that can be selected using the rotary switch mounted on the control panel. The speed options are 4 MHz, 400 Hz, 8 Hz, manually increment one full instruction, and manually increment one sub-instruction. The 4 MHz and 400 Hz clock signals were generated using a single SN74S124 oscillator chip [12]. It contains 2 independent oscillators. The chip requires a voltage present at the FC pin, a current through the RNG pin and a capacitance across the two CX pins. Unfortunately, the datasheet graphs for the voltage, current and capacitance values did not match up completely with the oscillation frequency. Different values were used until the desired frequencies were obtained. The oscillator circuit can

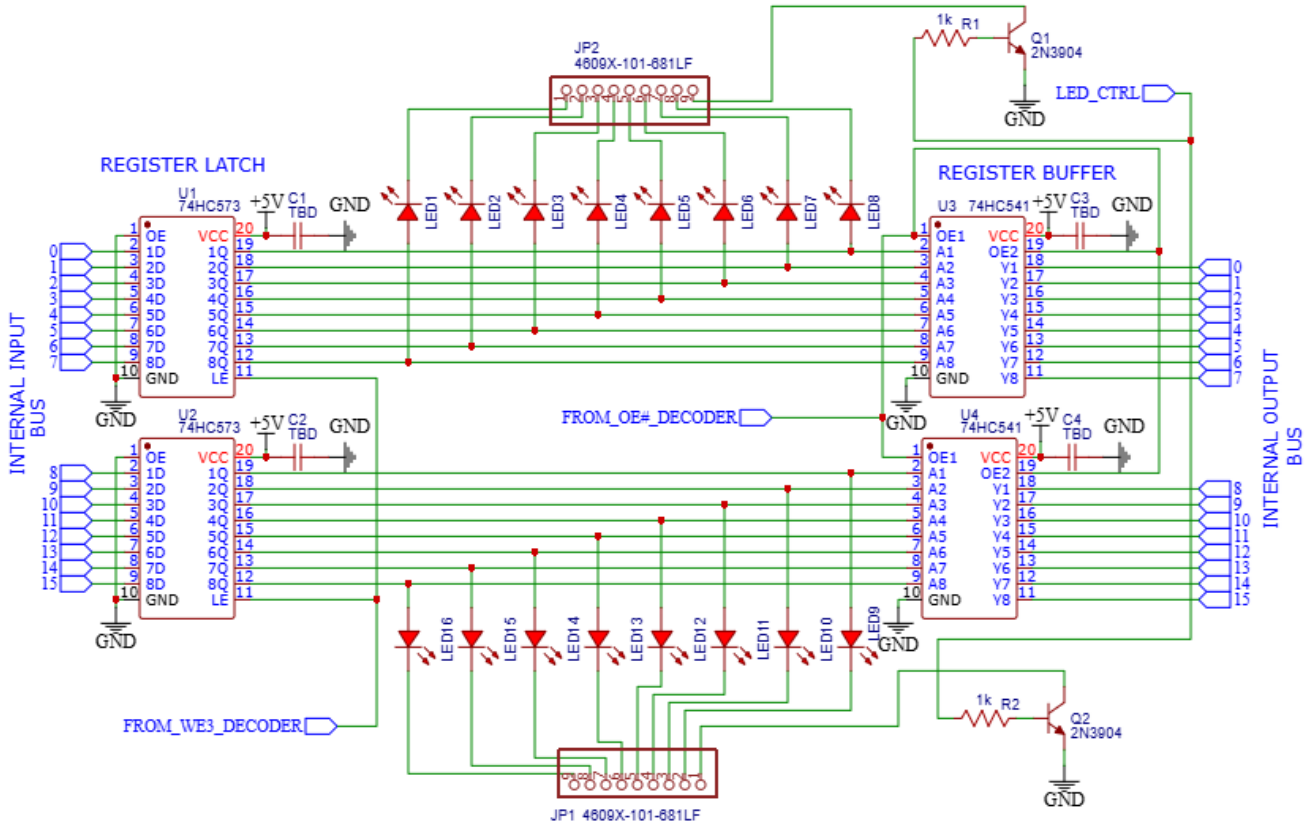


Fig. 3. Simplified Schematic of a Single Register in the Register File

be seen below in Figure 7.

The 8 Hz oscillator was created using an NE555 timer chip [13]. It was wired in a generic astable mode with the addition of a 1N4148 diode to bypass the second resistor. The purpose of this diode is to make the capacitor charge through one of the resistors and discard through the other. Setting the values of both resistors to the same value creates an oscillator with a perfect 50% duty cycle. This signal is more pleasing to the eye of the user. With this modification, the equation for the oscillating frequency becomes

$$f_{osc} = \frac{1}{1.34RC}$$

10 $K\Omega$ resistors a 10 uF capacitor were chosen which yield an oscillation frequency of 7.46 Hz. The schematic for this oscillator can be seen below in Figure 8.

The oscillator sections for the two manually increment models were a little more complicated to design as they required their own finite state machine. The sub-FSM and sub-sub-FSM counter needed to be used in this FSM. They will be described in more detail later. The sub-sub-FSM counter is a two-bit counter used to time the operations that happen in a sub instruction. The sub-FSM counter is a four-bit counter that is directly driven by the sub-sub-FSM counter. It is used to keep track of the current sub instruction. In order to increment through a sub-instruction, the FSM needs to start oscillating when a button is pressed and stop oscillating when the sub-sub-counter is 0b00. In order to increment though a full instruction, the FSM needs to start oscillating when the button is pressed and stop when both the sub-counter and sub-sub counter are 0b0000 and 0b00 respectively. Instead of designing two FSMs for each of these options, a single one was designed with an input that is high when either the sub-sub counter is 0b00 and increment one sub instruction is selected on the rotary switch or both the sub-sub counter and the sub counter are 0b00 and 0b0000 respectively. The logic for this part can be seen below in Figure 9. “POS0” is the position on the rotary encoder that corresponds to manually increment one sub instruction, it is active low.

The finite state machine was then developed using the one zero signal. The finite state machine is a mealy machine because it uses both current and stored values to determine the next output state. It has one output. The output stays low if the button is not pressed. Once the button is pressed, the output will toggle until the zero input is high, signaling that it has either reached the end of the instruction or sub instruction. An artifact of this FSM is that if the button is held down, the computer will increase forever until the button is released, it will then finish the instruction that it is on. The frequency that

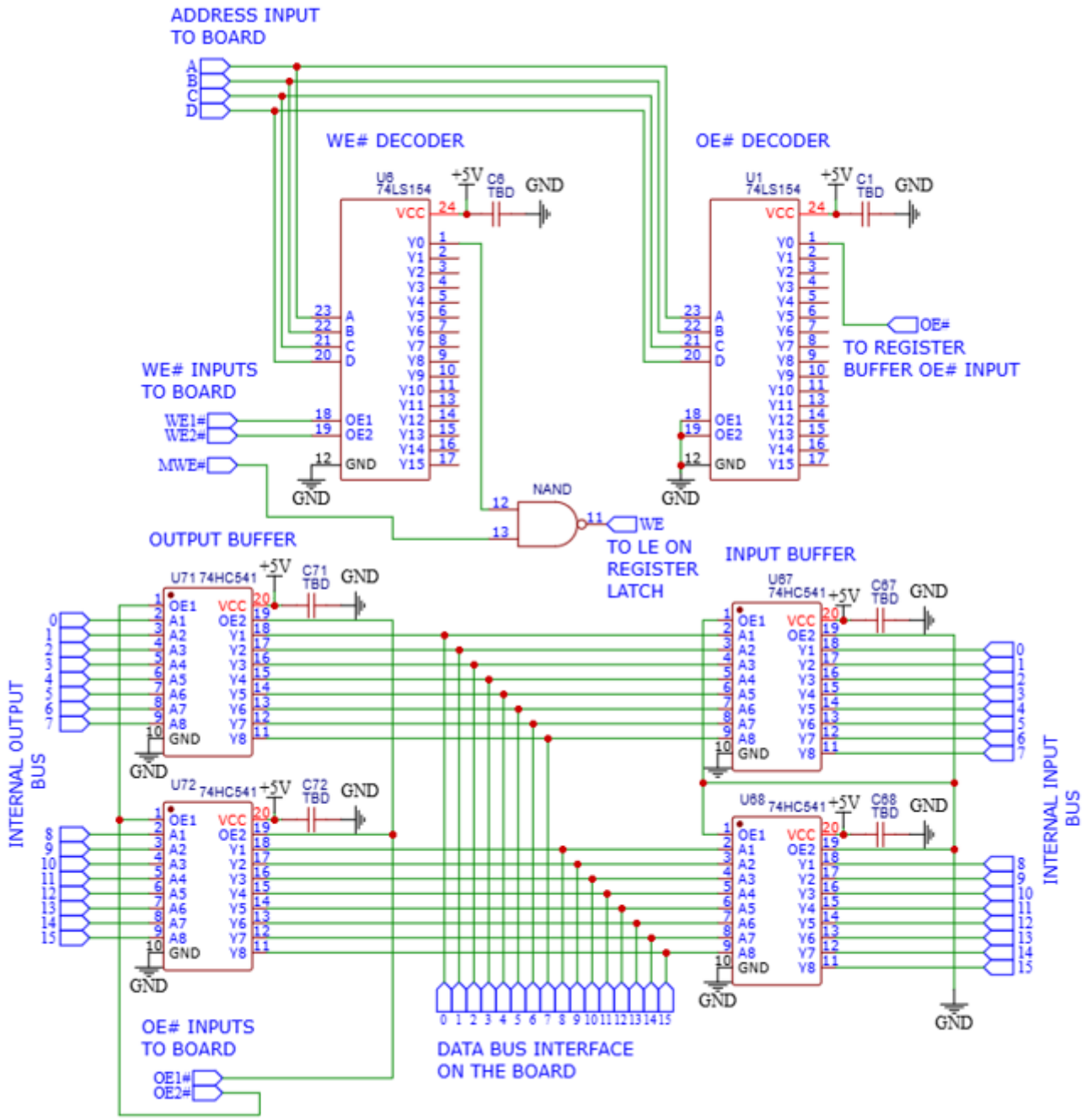


Fig. 4. Simplified Register Control Circuitry Schematic

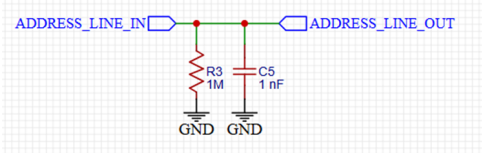


Fig. 5. RC Network Used to Filter Address Lines

this oscillation occurs at is 4 Hz. This was chosen so there is enough to see everything that is happening as this is mostly a feature for debugging. Another artifact is that the button needs to be held down for at most a 4th of a second so the FSM can see it and begin its cycle. The last artifact is that the finite state machine will trigger if the sub counters are not zero even if it is not on the manual setting. For instance, if the computer is run at a different speed, paused, and switched to manual, it

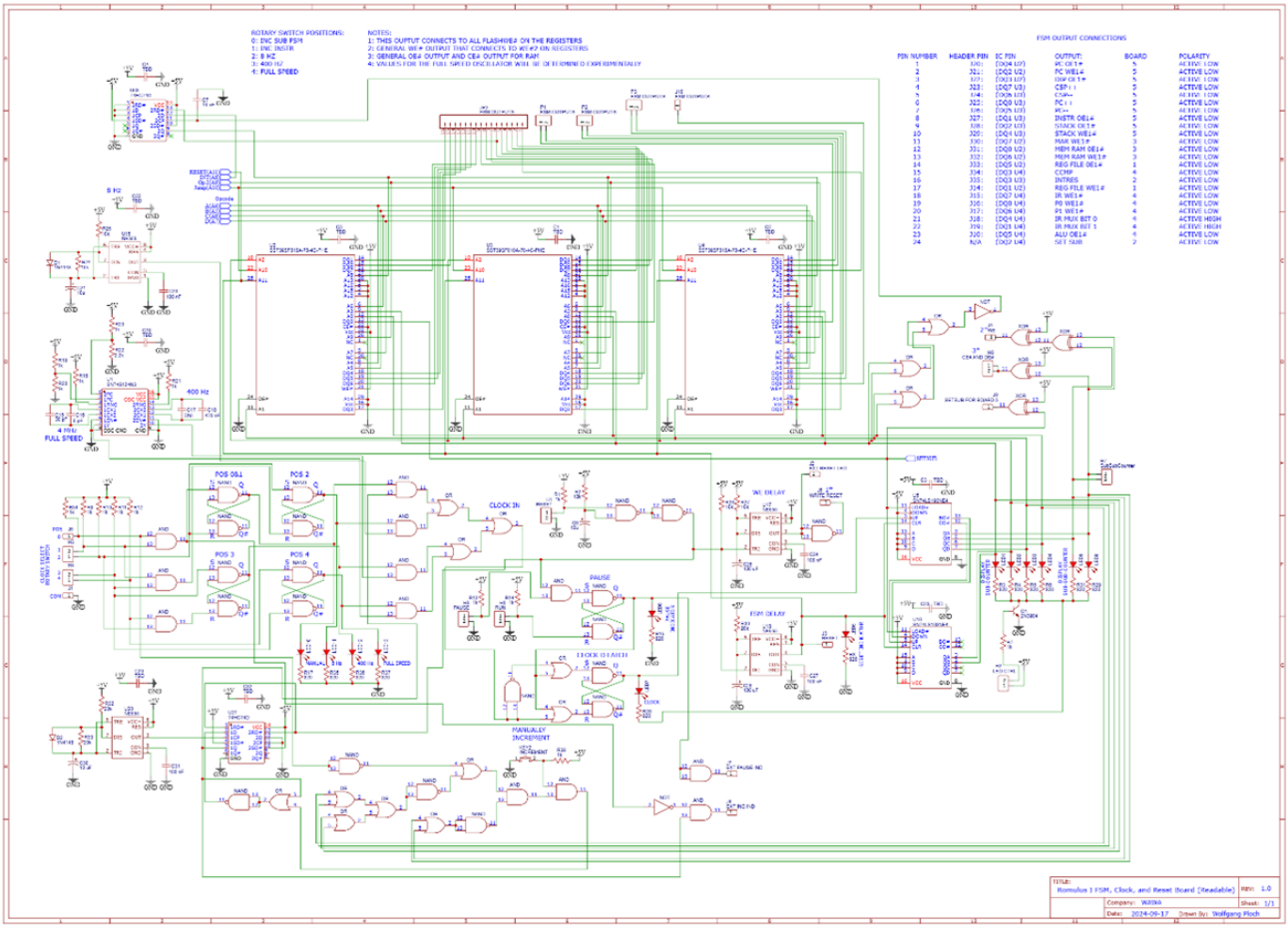

Fig. 6. Full Finite State Machine, Clock and Reset Board Schematic

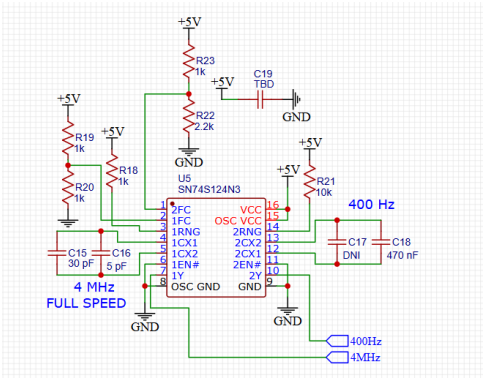


Fig. 7. 4 MHz and 400 Hz Oscillators

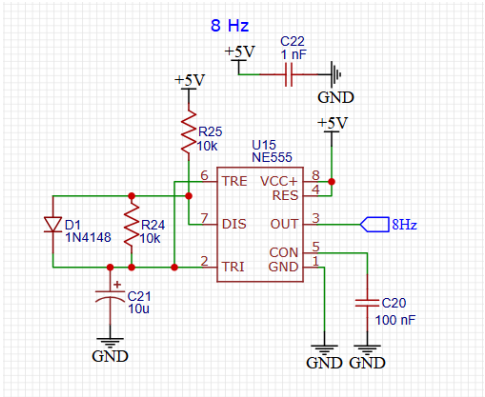


Fig. 8. 8 Hz Oscillator Schematic

will automatically start incrementing if the sub counter were not at zero at the time the computer is paused.

To make the finite state machine, a 74HC74 Dual D-Latch [14] was used to store the state and logic was used to realize the transfer table. A 4 Hz oscillator was designed using an NE555 [13] and the same equation used to design the 8 Hz oscillator. This circuit can be seen in Figure 10. “POS0” and “POS1” are the two positions on the rotary encoder that correspond to the two manual modes. The logic designed using them flashes the LED indicator on the control panel when clocking through manually.

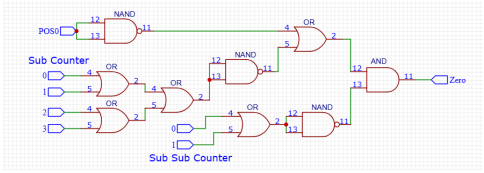


Fig. 9. Zero Determination Logic Schematic

To switch between all of the clock signals, a 5-position rotary switch is used. The common pin is grounded which each of the position pins are pulled to 5 volts using a 1 $k\Omega$ resistor. To generate the signals used to select the different clocks, a sophisticated SR latch circuit was designed. Physical switches often introduce "bouncing" in a circuit. The metal contact vibrates at the instant it is closing causing an undesired, short series of pulses upon a transition. To prevent this, the circuit was designed to latch the positions the instant that they are reached. The basic principle is that the desired position on the rotary switch sets its corresponding latch, this latch is then reset by the positions on either side of it. For instance, the latch for position 2 is set by position 2 and reset by either position 1 or 3. Positions on the edges require one reset (When at position 0, it can only possibly turn to position 1 from there). The positions 0 and 1 both correspond to the manual increment modes. Since both connect the same clock source to the clock input, they are combined and treated as one position as far as this debouncing circuit is concerned. The SR latches used in this circuit and the rest of the board are made from 2 NAND gates in an arrangement shown in Figure 11. The main output (Q) is active high (Q# ) is active low and the reset and set inputs are active low as well.

These were used to make the rotary switch selection circuit. The outputs of the position latches than feed into AND gates. The other input of the AND gates are connected to the corresponding clock. These are all ORed together to create a single clock output. This portion of the circuit can be seen below in Figure 12.

*3) Pause Handling:* This "main clock" signal created in the circuit above is connected to the input of a D latch. This D latch is used to pause and un-pause the computer by latching the clock when the pause button is pressed. The pause and un-pause buttons are located on the control panel. There is an indictor for pause on the control panel and the circuit board. To make it easy to keep the computer in its reset state upon startup, reset automatically pauses the computer. Pressing pause sets the pause SR latch while pressing un-pause, resets the latch. These SR latches are again designed using the NAND gate SR latch in Figure 6. The Q of the Pause SR latch is then connected to the LE input of the D latch. The "Conditioned Clock" is the final clock signal that will be used to drive the counter ICs. This portion of the circuit can be seen below in Figure 13.

*4) Reset Timing:* The computer resets itself when it is turned on and when the user presses the reset button on the control panel. To create this reset signal when the computer is powered on, an RC network is used to create an active low signal. The capacitor voltage starts at 0 volts and charges through the resistor until it reaches the high input level of the logic gate. With a resistor value of 10 $k\Omega$ and 10 uF, this reset signal takes 120 milliseconds to reach the high threshold level of the logic gate family used. This pulse is then ANDed with an active low button press to create a combined signal. This active low reset signal "RESET# " then connects to a few latches on this board. It can be seen in Figures 5 and 8. This "RESET# " is used to trigger two NE555 timers [13] in monostable mode. The first one is the "WE DELAY" which creates a one second pulse and the second is the "FSM DELAY" which creates a two second pulse. The reason these two timing signals are needed comes down to the latches used in all of the registers. They level triggered, not edge triggered. This means that the data that is latched into them needs to be held at the inputs after the latch enable input is deactivated. The data bus is pulled to ground using 4.7 $k\Omega$ resistors on board 4. The FSM DELAY timer is an input to the FSM flashes. When this input is high, the all of the output and write enables of all of the registers are deactivated, meaning that nothing is on the bus. The resistors pull the bus to ground. The output of WE DELAY is inverted, then wired off the board to the master writes on all registers in the computer. The delay for the FSM is longer than the delay used to latch the registers to ensure they all latch 0x0000 properly during a reset. The reset and timing circuitry can be seen in Figure 14.

*5) Timing Signals and Counters:* The sub counter and the sub-sub counter described in previous sections are generated using two 4-bit binary counter ICs, the 74LS193 [15]. This IC has a borrow and carry outputs that makes it easy to string them together and create counters higher than 4 bits. The carry output is connected to increment of the next chip and the borrow output is connected to its decrement. This is how the two 74LS193s were connected in the circuit. When the increment or decrement inputs on the chip are being used, the other one needs to be pulled to Vcc. Because this counter is only incrementing, the decrement input is permanently tied to 5 volts. The counter has an active high clear, both are connected to the FSM DELAY. The counter IC also offers 4 inputs for each bit. There is an active low load input as well. When this load input is pulled low, the data on these inputs is latched to the outputs. All of the data inputs on both chips are pulled high to 5 volts. The load input is connected to one of the FSM outputs, called SETSUB. The lowest order 2 bits of this 8 bit counter are the sub-sub counter, the next 4 bits are the sub-counter. The highest order 2 bits are unused. Since not all instructions use the same amount of sub instructions, there is a sub instruction called SETSUB. As a side note, this output is inverted and wired to an output on the board, an active high version is needed by board 5 to deal with hardware interrupts. When it is activated, it loads 0b111111 to the combined counter. This jumps the counter to the end, causing the next clock pulse to take it to 0b000000, and therefore, the next instruction. This is a way of terminating the instruction after all of the sub instructions are done instead of having to

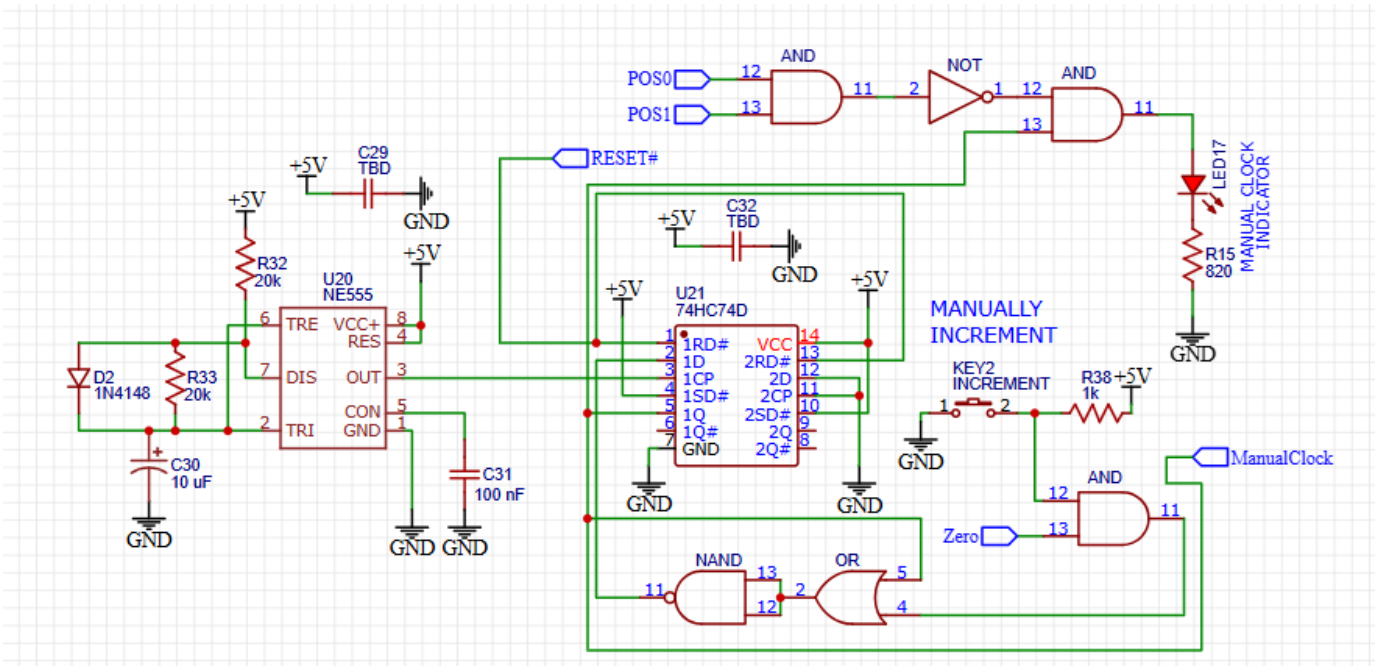


Fig. 10. Manual Increment Finite State Machine Circuit

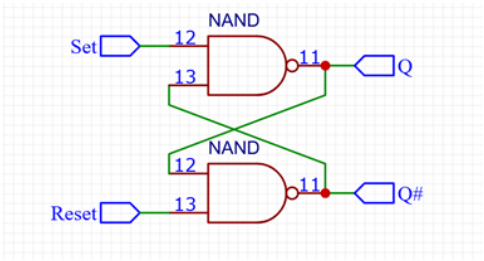


Fig. 11. SR Latch Circuit Using NAND Gates

wait for the counter to count past the unused sub-instruction slots.

Each of the 6 bits are connected to LED indicators on the board. These indicators (along with the main clock indicator) have their cathodes tied to the collector of a 2N3904 transistor. The base of this transistor is then controlled by the main LED control switch on the control panel. This connects to all of the other cathode transistors controlling all of the other LEDs on the other boards.

Special timing routines need to be used when writing and reading information to the selected SRAM chips, the AS6C1008s [16]. Their wiring is exploring in more detail when discussing their circuits, but the timing will be described here. The AS6C1008 does not have separate input and output signals, instead, IO pins that switch. The chip requires the WE# pin be pulled low and a period of time elapse before the CE# input be pulled low and data be present at the pins. Since the IO pins of the chip are connected directly to the data bus, this means the data cannot be present on the bus until a time after the WE# input is pulled low. The period of time is much shorter than the maximum clock frequency of the computer, but it still needs to be accounted for. This timing issue is the reason behind the two-bit sub-sub clock. This creates four different slots that be used to create this timing sequence. The timing sequence can be seen below in Figure 15.

WE is write enable, OE is output enable, and CE is chip enable. This timing sequence works for the write and output enable inputs of the registers as well. The reason that they are labeled with a 2 after the name is due to the way the FSM selected the register of SRAM to use. Every device has two WE# inputs and two OE# inputs. Both inputs must be low for the desired operation to take place. The first ones (OE1# and WE1# ) are connected to signals from the FSM flash chips. This is way the FSM selects the devices needed for the sub-instruction. The second set of inputs (OE2# and WE2# ) are controlled by the timing sequence seen in Figure 10. This allows the timing sequence to control the devices selected by the FSM. Each of the SRAM chips has two extra inputs (CE1# and CE2# ). CE1# is created on the board by ANDing its OE1# and WE1# . This reduces the number of outputs needed from the FSM flash chips. This allows the chip to be enabled if it is being written to or read from. Logic was used to generate these timing signals from the two bits of the sub-sub counter. The counter circuit and timing signal generating circuit can be seen below in Figure 16.

*6) Interrupt Handling Correction:* The interrupt signal generated on board 5 was initially a direct input to one of the addresses of the flash chips. This caused errors when an interrupt was triggered half way through an instruction. The solution was to latch the state of the interrupt input at the beginning of an instruction and use a control signal from the

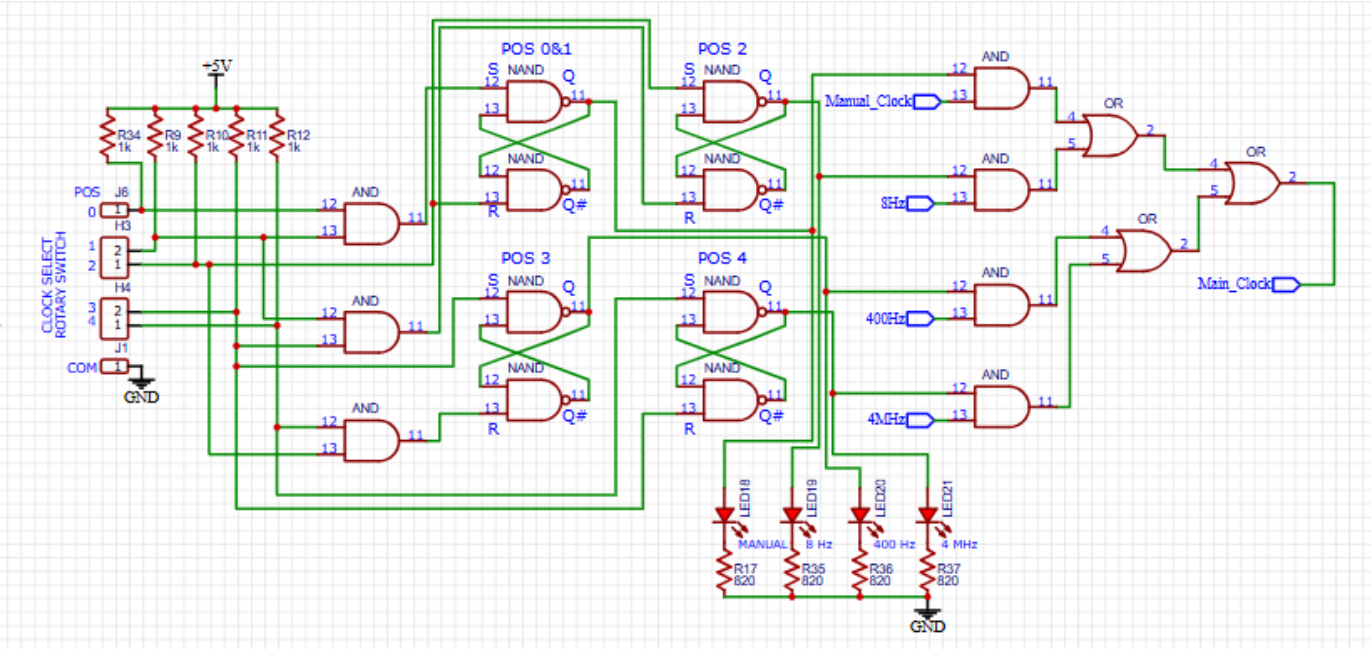


Fig. 12. Rotary Switch De-bounce and Clock Selection Circuitry

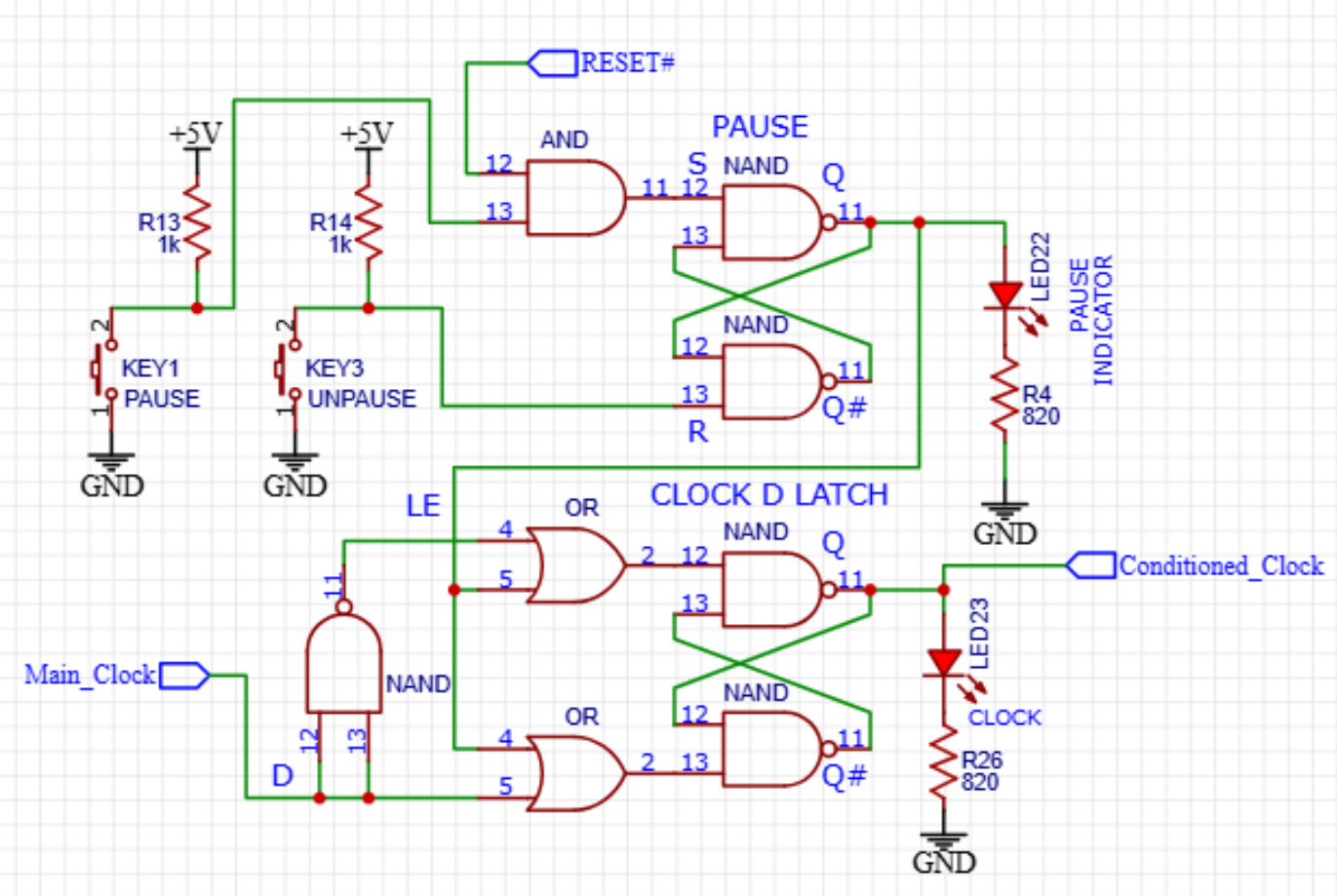


Fig. 13. Clock Pause Handling Circuit

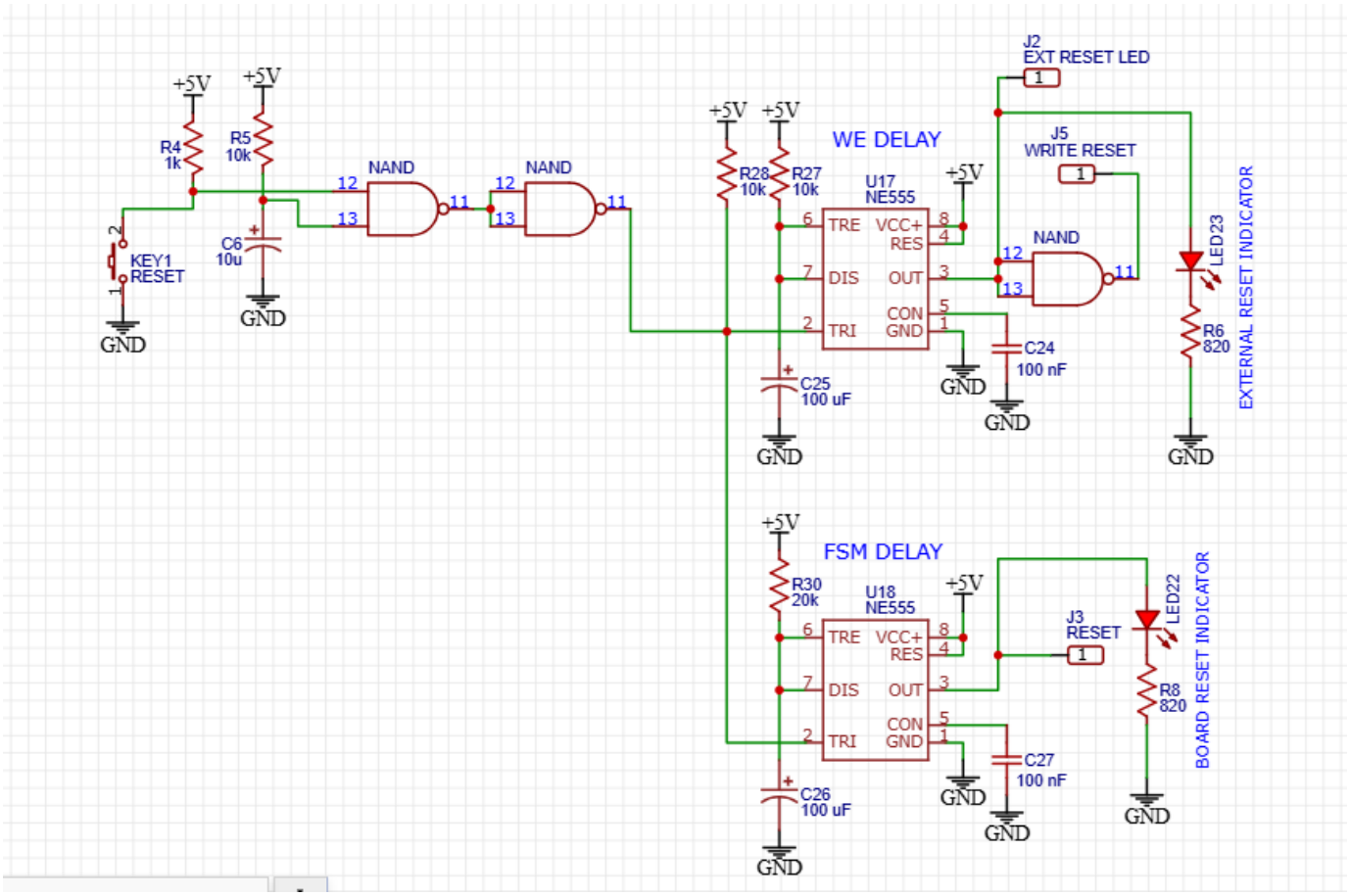


Fig. 14. Reset Timing Circuitry

| Sub-Sub Counter | 0b00 | 0b01 | 0b10 | 0b11 |
|---|---|---|---|---|
| WE2# | High | Low | Low | High |
| CE2# | High | High | Low | Low |
| OE2# | High | High | Low | Low |

Fig. 15. Sub-Sub Counter Control Signals Timing Diagram

FSM to reset it at the end of the interrupt path. To accomplish this, a 74HC74 D latch [14] was used. Its data bit is connected to the incoming interrupt signal, its Q pin is connected to the address bit of the flash. The clock pin is connected to logic that triggers it when the sub-counter is 0b0000. The reset pin is connected to INTRES, an active low signal created during the troubleshooting process that is used to control timing associated with the interrupt handling. This circuit can be seen below in Figure 17.

*7) Finite State Machine Flash Chip Lookup Table:* The finite state machines outputs are data locations store across three flash chips. These chips are the SST39SF010As [11]. They have 17 address inputs (A0-A16), and 8 outputs (DQ0-DQ7). They are programmed externally using a microcontroller and inserted into the circuit using 32 PDIP sockets. This way, their WE, CE, OE can be wired to constant voltage sources to make the chip always output data, making it act as simple combinational logic. Only 12 address inputs are used, the unused five (A12-A16) are wired to ground. The 12 used addresses of the flash chips are all wired together so each chip gets the same address. Each of the outputs of the three chips (24 of them total) is assigned a specific control signal that are sent to other parts of the computer. The programming of the flash chips and the flow of the FSM is explored further in its section. A summary of the inputs to the FSM flash chips can be seen in Table IV.

TABLE IV
INPUTS TO THE FSM FLASH CHIPS

| Signal | Polarity | Flash Chip Address |
|---|---|---|
| Sub Clock Bit 0 | Active High | A0 |
| Sub Clock Bit 1 | Active High | A1 |
| Sub Clock Bit 2 | Active High | A2 |
| Sub Clock Bit 3 | Active High | A3 |
| Opcode Bit 0 | Active High | A4 |
| Opcode Bit 1 | Active High | A5 |
| Opcode Bit 2 | Active High | A6 |
| Opcode Bit 3 | Active High | A7 |
| Interrupt | Active High | A8 |
| Op2.0 | Active High | A9 |
| Jump | Active High | A10 |
| Reset | Active High | A11 |

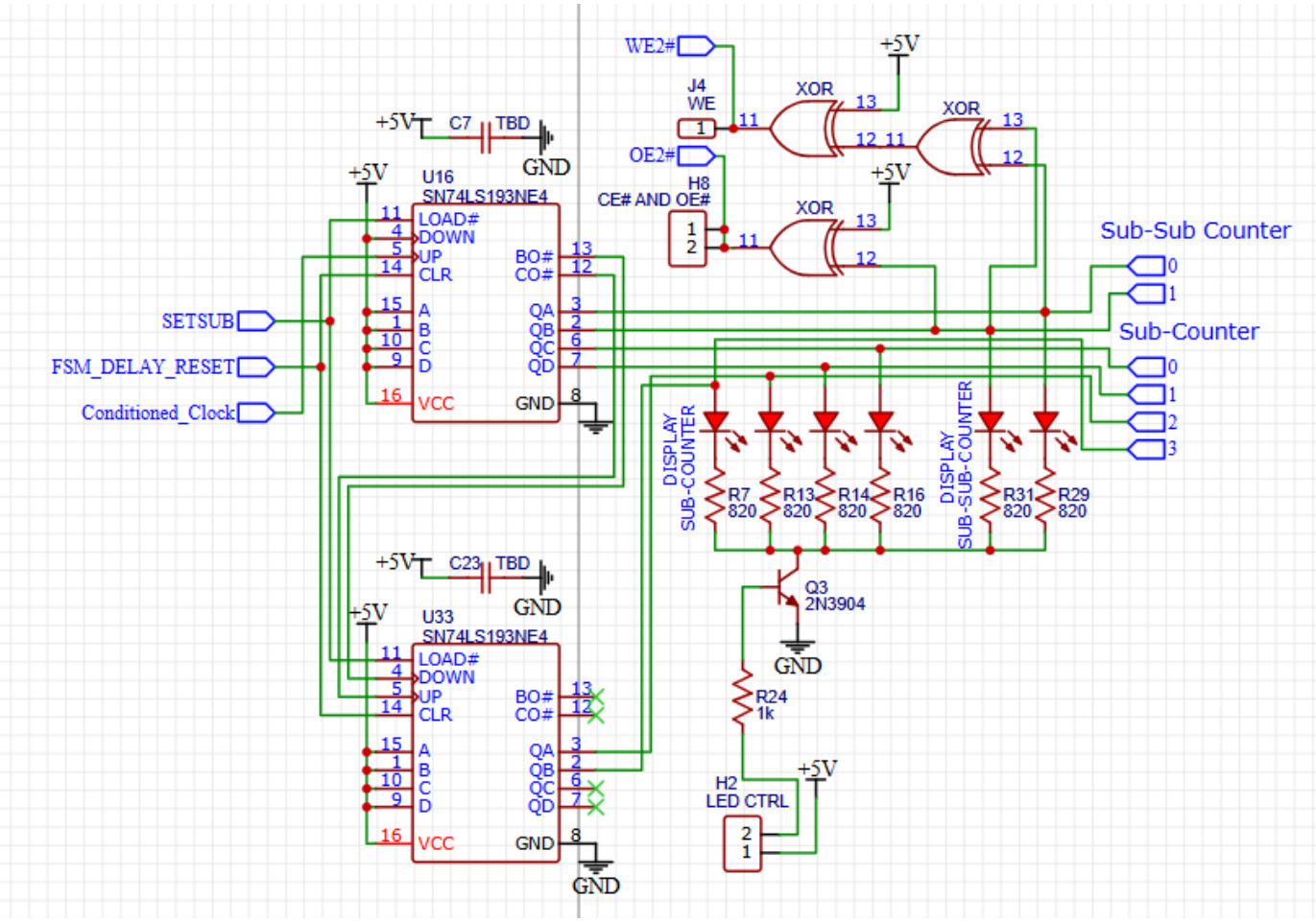


Fig. 16. Sub Counter, Sub-Sub Counter, and Control Signal Timing Circuit

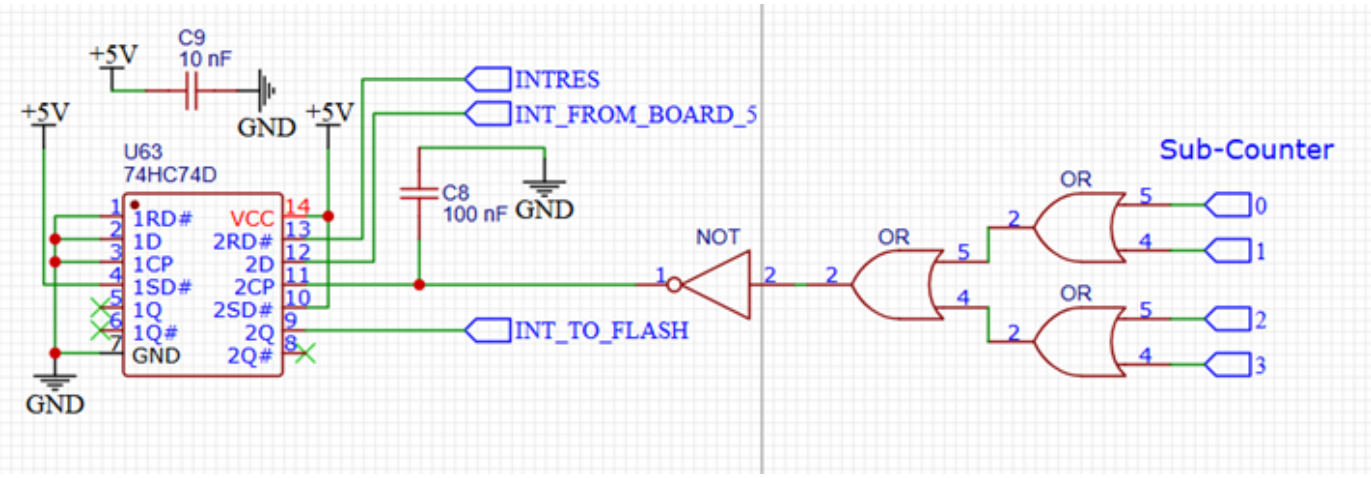


Fig. 17. Interrupt Latching Circuit

The four bits of the opcode come from the instruction register on board 4. The signal “Op2.0” is described in more detail in the section describing the instruction set architecture. Jump comes from board 4 as well and is described in the section on board 4. These are the 12 signals that the FSM needs in order to determine what its outputs should be. The outputs of the FSM flash chips are summarized below in Table V.

The signals outlined in Table V, not already mentioned in this section are explored in the section corresponding to their

TABLE V
FSM FLASH CHIP OUTPUTS

| Signal | Polarity | Destination Board | Flash IC | Pin |
|---|---|---|---|---|
| PC OE1# | Active Low | 5 | U2 | DQ4 |
| PC WE1# | Active Low | 5 | U2 | DQ2 |
| DIP OE1# | Active Low | 5 | U2 | DQ3 |
| CSP++ | Active Low | 5 | U3 | DQ7 |
| CSP– | Active Low | 5 | U3 | DQ6 |
| PC++ | Active Low | 5 | U3 | DQ0 |
| PC– | Active Low | 5 | U3 | DQ5 |
| INSTR OE1# | Active Low | 5 | U3 | DQ1 |
| STACK OE1# | Active Low | 5 | U3 | DQ2 |
| STACK WE1# | Active Low | 5 | U3 | DQ4 |
| MAR WE1# | Active Low | 3 | U2 | DQ7 |
| MEM RAM OE1# | Active Low | 3 | U2 | DQ0 |
| MEM RAM WE1# | Active Low | 3 | U2 | DQ6 |
| REG FILE OE1# | Active Low | 1 | U2 | DQ5 |
| CCMP | Active Low | 4 | U4 | DQ3 |
| INTRES | Active Low | 2,4 | U3 | DQ3 |
| REG File WE1# | Active Low | 1 | U2 | DQ1 |
| IR WE1# | Active Low | 4 | U4 | DQ7 |
| P0 WE1# | Active Low | 4 | U4 | DQ0 |
| P1 WE1# | Active Low | 4 | U4 | DQ6 |
| IR MUX BIT 0 | Active High | 4 | U4 | DQ4 |
| IR MUX BIT 1 | Active High | 4 | U4 | DQ1 |
| ALU OE1# | Active Low | 4 | U4 | DQ5 |
| SETSUB | Active Low | 2 | U4 | DQ2 |

board. These outputs connect directly to header pins on the edge of the board for ease of wiring to other boards.

### C. Board 3

*1) Overview:* The memory and mapped IO board contains the memory SRAM, the mapped IO slots, and the memory address register (MAR). It is referred to as board 3 in other design sections. It uses the same SRAM chips as board 5 (the AS6C1008s) [16]. The IO ports have 16 inputs and 16 outputs each. They are addressed as the last four memory addresses; 0xFFFC, 0xFFFD, 0xFFFE, and 0xFFFF. The full schematic of this board can be seen in Figure 18.

*2) Memory Address Register (MAR):* The MAR stores the memory address used by the RAM to find the correct value. It is identical in operation to the registers on board 1. Its inputs connect directly to the bus and its outputs connect directly to the address inputs of the SRAM chips. Since it is always outputting to the memory RAM, it does not have OE1# and OE2# signals. It's circuitry can be seen in Figure 19. In addition to what is seen in Figure 2, it has a MWE# which writes the register regardless of WE1# or WE2# .

*3) Address Decoding:* The circuit needs to be able to be able to detect if the addresses being written to or read from are in the SRAM or in the memory mapped IO slots (last 4 addresses). This is accomplished using combinational logic. Since the last 4 addresses correspond to the lowest order 14 addresses being 1, (0b11111111111111xx), if they are all 1 then the address has to be in the memory mapped IO section. Two signals are created, SELECT RAM and SELECT MAPPED IO. These signals are both active low. The logic used to create these signals can be seen below in Figure 20. The schematic shows stand 2-input AND gates for simpicity but 4-input AND gates were used in the real schematic and the PCB, the 74LS21 [17].

*4) Mapped IO Port:* The four mapped IO ports are identical. They resemble the other registers used in the computer, the difference being they have an IO port instead of LEDs. The output of the two 74HC573s [9] connect directly to the IO port and function as outputs. The inputs of the two 74HC541s [10] are pulled to ground through 4.7 $k\Omega$ resistor arrays. This ensures that the pins do not float when a device is not plugged into them, These function as the inputs to the computer. "Writing" to this memory address causes the value to appear on the 16 output pins while "reading" from this address stores the value of the 16 input pins. Because of this, information "stored" in this address cannot be read from it. Each of the 2x20 pin connectors is wired to the input and output bits. Each connector also has a power and ground pin as well as an interrupt pin. This pin is pulled to ground through a 1 $k\Omega$ resistor. The four interrupts from the four IO slots are ORed together and this creates the INT IN signal used by board 5. The schematic of the 0xFFFF IO slot can be seen below in Figure 21, the other three are identical.

*5) Mapped IO Port Control Signal Logic:* Logic is required to control the different write enables and read enables of the different mapped IO slots. Coming into the board, there are five signals which match the convent seen elsewhere in the computer: MEM WE1# , MEM WE2# , MEM OE1# , MEM OE2# , and FSM CE# . When the MAR holds an address value that belongs to one of the IO slots, logic further decodes that and the input signals to determine which mapped IO port is being selected and whether to write to it or read from it. This portion of the circuit can be seen in Figure 22. The MWE# input is used to write all of the IO ports at once during a reset.

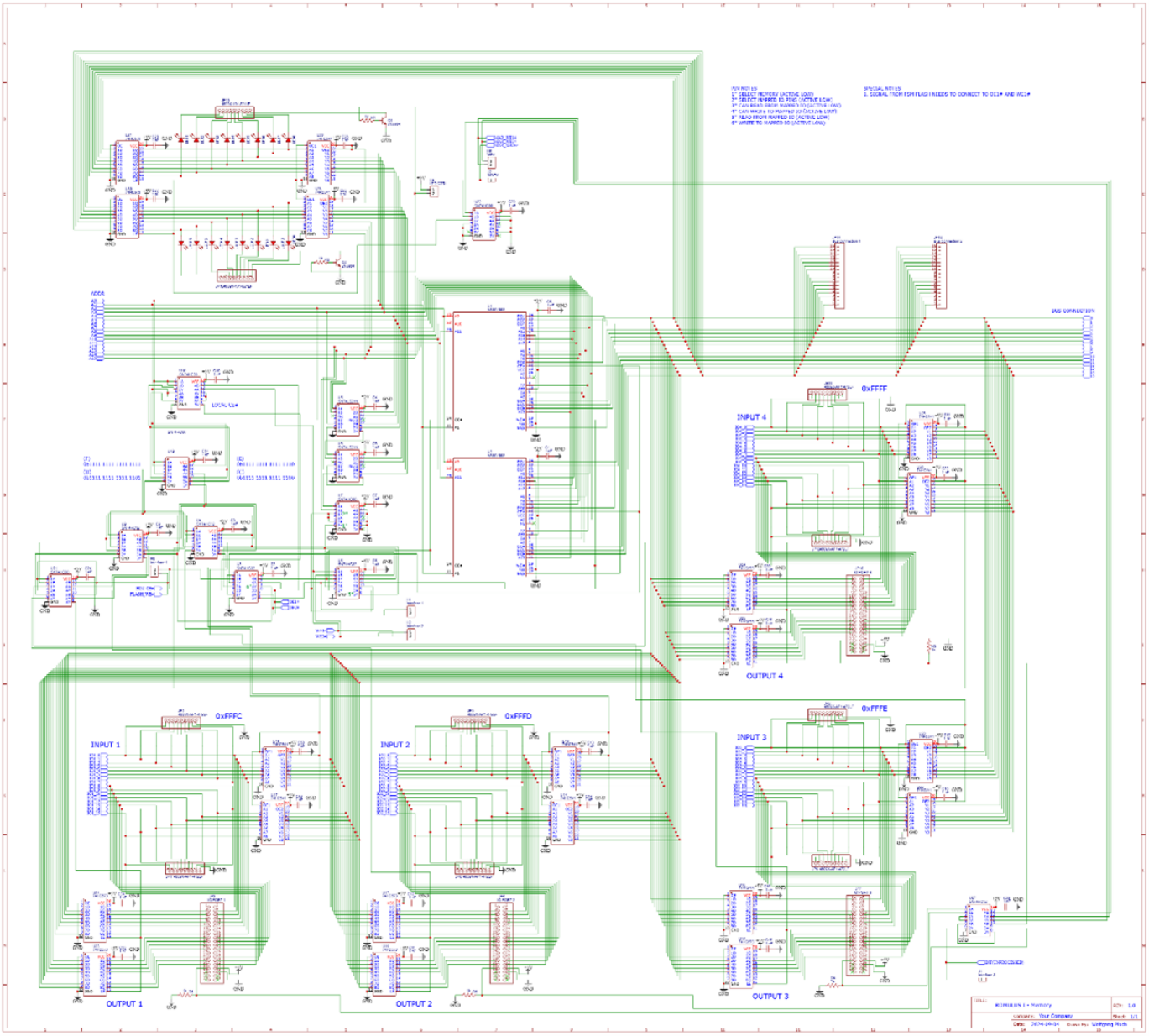


Fig. 18. Full Schematic of the Memory and Mapped IO Board

*6) Memory SRAM and Supporting Circuitry:* The SRAM chips chosen are the AS6C1008s [16]. They are 128k chips with 17 address pins. Two are used as each only stores one byte (8 bits). Because the computer is only capable of addressing 16 bits, the highest order bit on each chip (A16) is grounded. The IO pins of the chips are connected to the corresponding bits on the data bus. The address bits of the two chips are connected to each other (i.e. chip 1 A0 connected to chip 2 A0 etc.). These pairs are then attached directly to the MAR. The same timing procedure with WE, OE, and CE described in the description of board 2 is used here. On top of the regular timing logic, some needed to be designed to block the signals from the SRAM when the memory mapped IO ports are being used. The SRAM chips and their supporting circuitry can be seen in Figure 23.

### D. Board 4

*1) Arithmetic and Logic Unit (ALU):* The ALU is the part of the computer that does all of the arithmetic and logic operations (hence the name “arithmetic and logic unit”). It handles bitwise AND, NOT, OR, and XOR, as well as logical NOT, logical shift right, addition, and subtraction. When one or two parameters are loaded into the “Parameter 0” and “Parameter 1” registers at the bottom of the board and the opcode is sent to the ALU, it immediately calculates the correct answer and, when its output is enabled, places it onto the data bus. The schematic for the ALU in Figure 24.

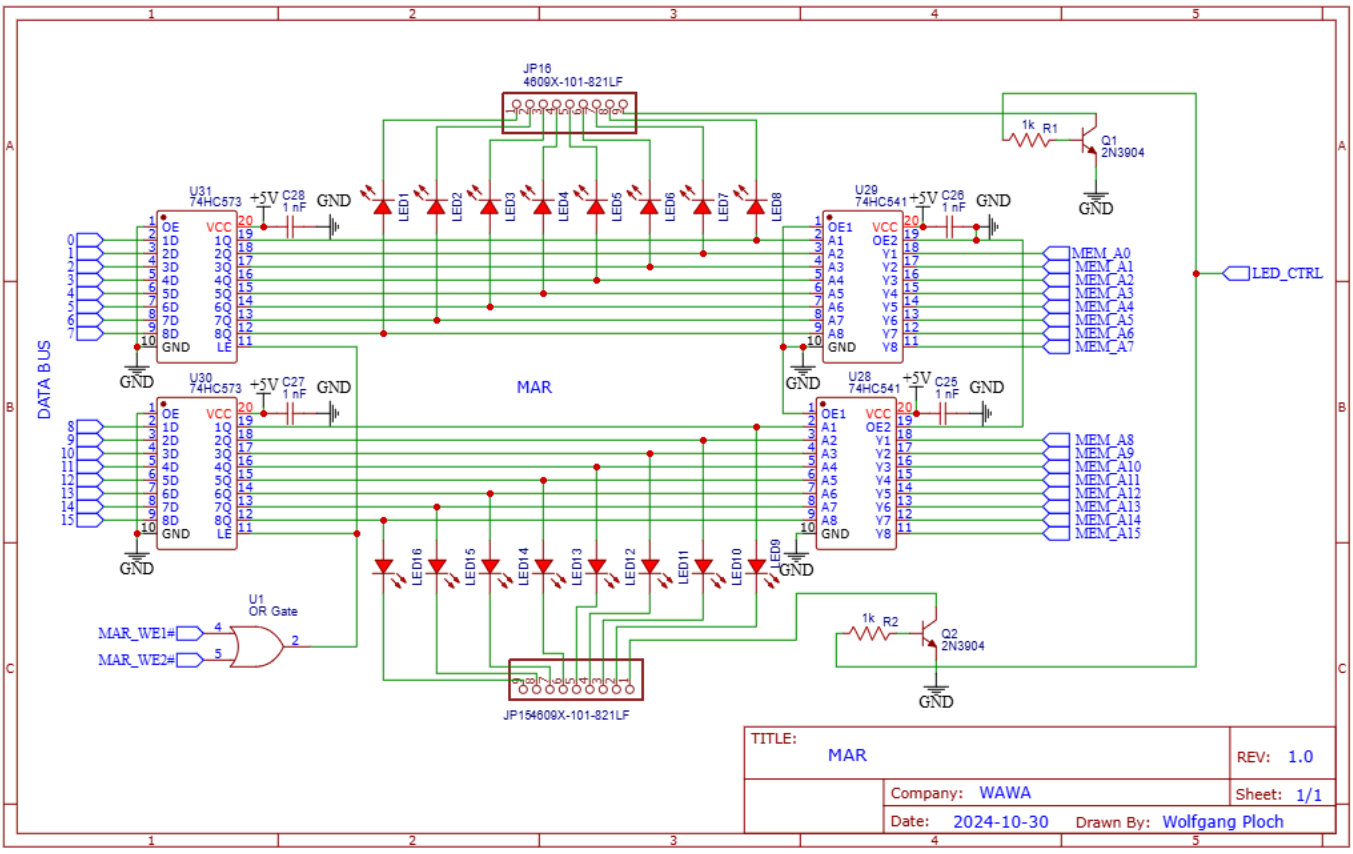


Fig. 19. Memory Address Register (MAR) Circuitry

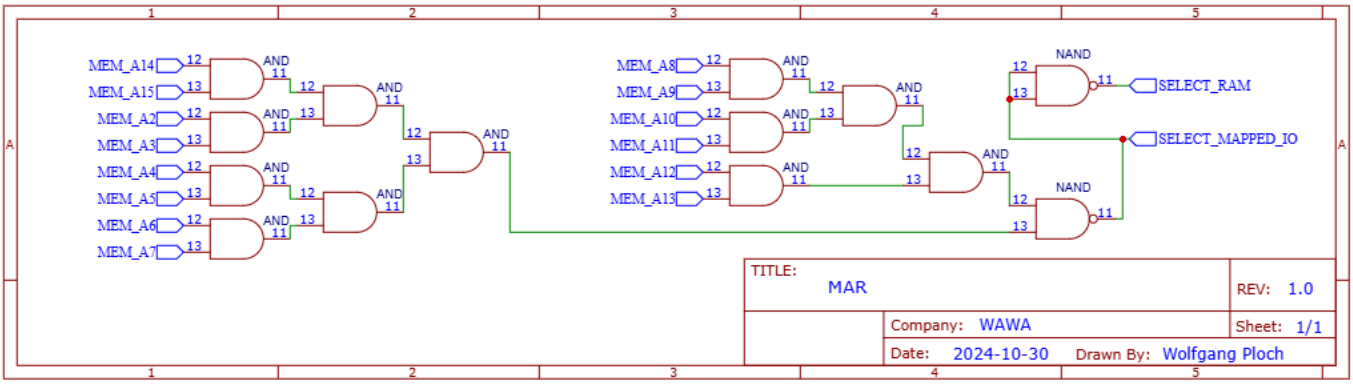


Fig. 20. Logic Used to Create SELECT Outputs

We decided to make the ALU out of flash chips (SST39SF010A) [11] instead of with logic gates because it takes up less physical space on the printed circuit board. It works like a lookup table, with the address bits as the inputs and the data bits as the outputs. We had to use three chips because we needed enough address (input) bits for two 16-bit parameters and a 4-bit opcode, and since each SST39SF010A chip only has 17 address bits, we needed three chips to have enough. Table VI shows which pins are used for what values on each chip.

*2) Instruction Register (IR):* The instruction register stores, in bitcode, the current instruction being executed by the computer. Many things are taken from the instruction: the four most significant bits of the instruction (bits 15-12) are the opcode, bit 7 is the sub-opcode used for push and pop, and the four least significant bits (bits 3-0) are used as flags to determine which comparison to make in the jump-compare instruction. Additionally, bits 11-8, 7-4, and 3-0 are used as addresses to the register file to determine which registers to

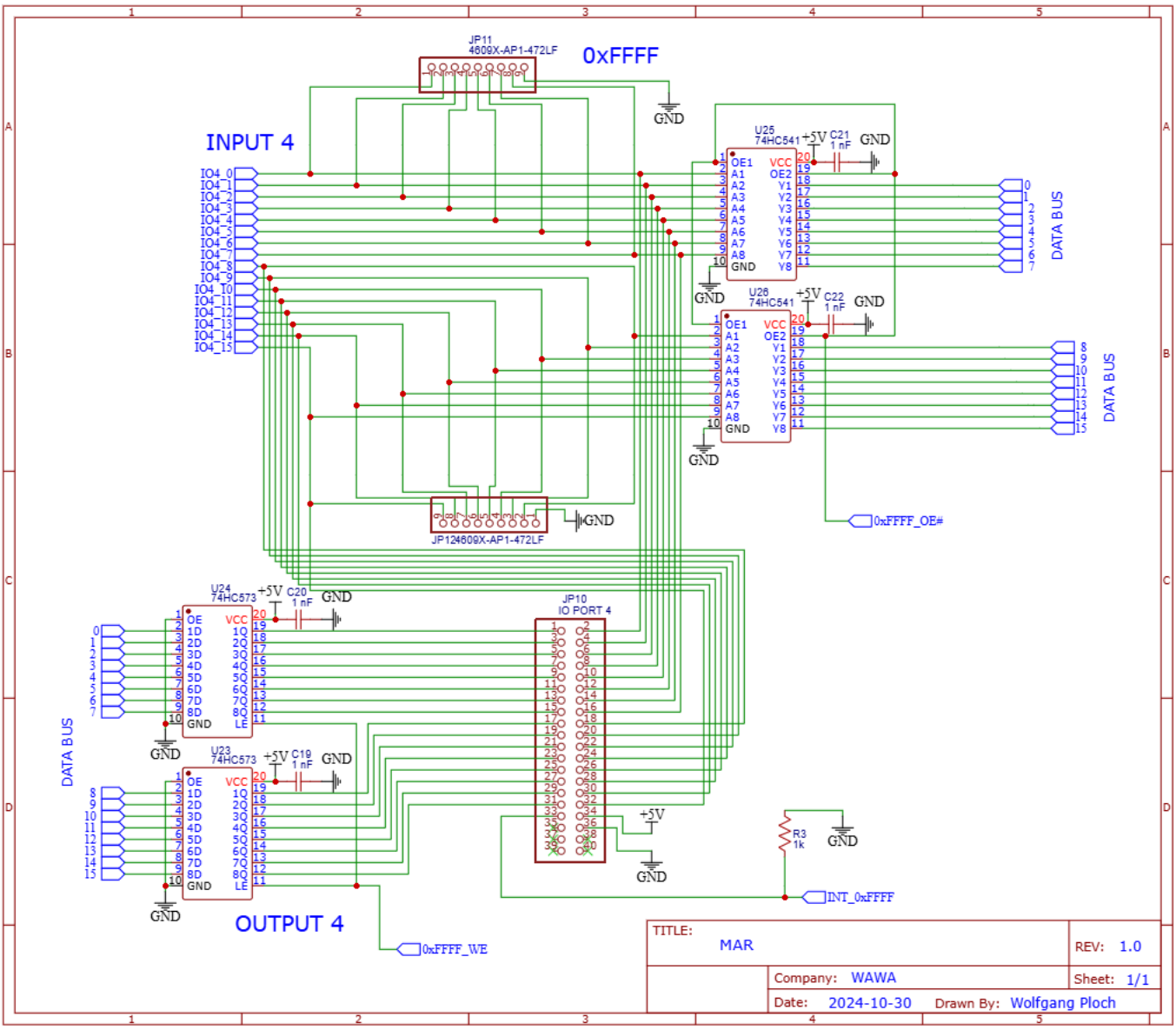


Fig. 21. Sample Memory Mapped IO Port

TABLE VI
PIN TO VALUE MAPPING FOR THE ALU. GND MEANS THE PIN IS CONNECTED TO GROUND, NC MEANS THE PIN IS NOT CONNECTED.

| U1: Most significant | | U2: Middle | | U3: Least significant | |
|---|---|---|---|---|---|
| Pins | Value | Pins | Value | Pins | Value |
| A16-A13 | Opcode | A16-A13 | Opcode | A16-A13 | Opcode |
| A12-A7 | P0[15:10] | A12-A9 | P0[9:6] | A12-A7 | P0[5:0] |
| A6-A1 | P1[15:10] | A8-A5 | P1[9:6] | A6-A1 | P1[5:0] |
| A0 | Carry in | A4-A2 | N/A (GND) | A0 | Borrow in |
| | | A1 | Borrow in | | |
| | | A0 | Carry in | | |
| DQ7-DQ2 | result[15:10] | DQ7-DQ4 | result[9:6] | DQ7-DQ2 | result[5:0] |
| DQ1 | N/A (NC) | DQ3-DQ2 | N/A (NC) | DQ1 | N/A (NC) |
| DQ0 | Borrow out | DQ1 | Borrow out | DQ0 | Carry out |
| | | DQ0 | Carry out | | |

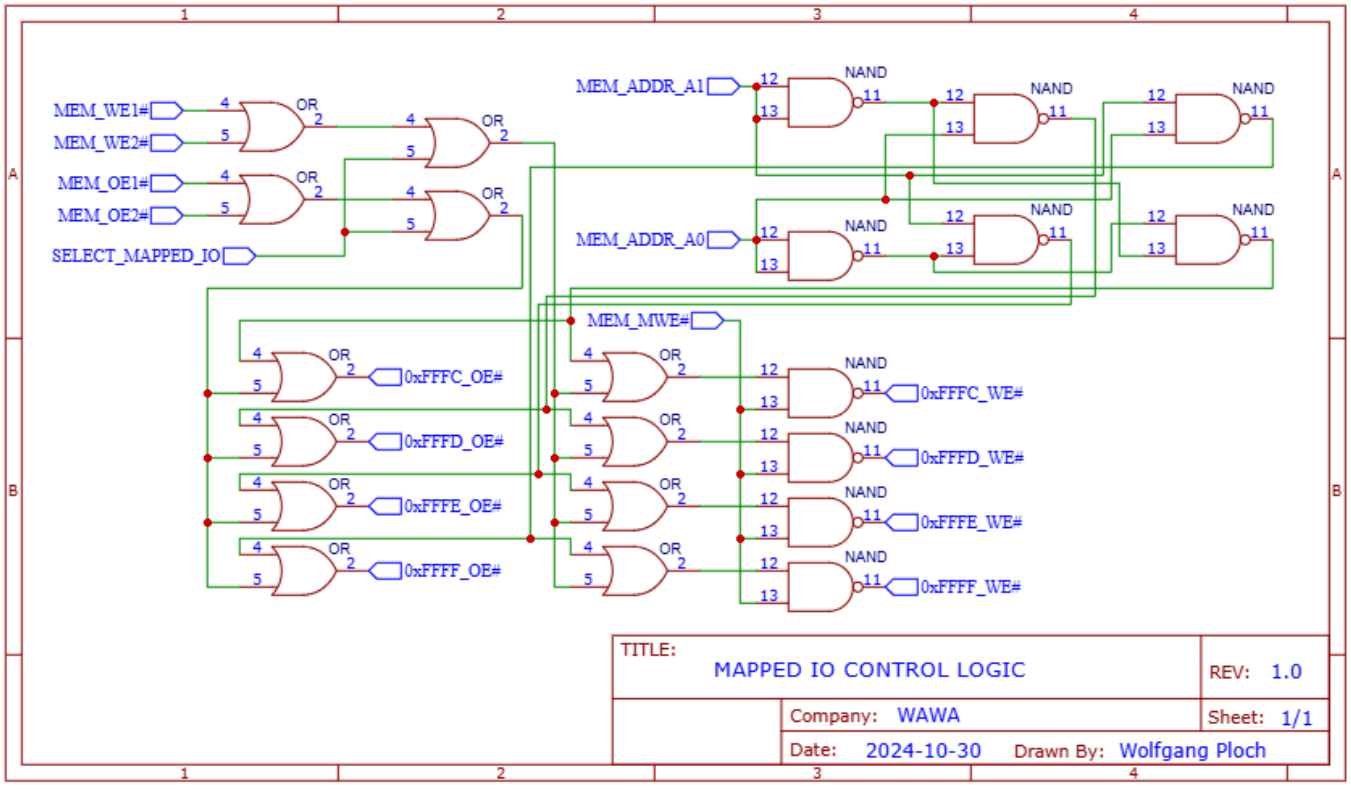


Fig. 22. Mapped IO Port Control Signal Logic

read and write from. Because these signals and values are so important, the IR always outputs its stored value, that is, its output enable is always active. The opcode, sub-opcode, and flags go directly to the control signals board as inputs to the finite state machine. The three groupings of four bits used for register addressing go through a multiplexer to the register file. The control signals outputs "sel1" and "sel0" are used to determine which portion of the value stored in the IR is used for the register file address: both low sends the first set of four bits, 11-8 (labeled as "Ra" in the instruction set), sel0 high sends the second set, 7-4 (labeled as "Rb"), and sel1 high sends the third set, 3-0 (labeled as "Ry"). Both sel0 and sel1 being high should never occur with our current control signals design, but should this happen erroneously, or should the FSM be changed to allow this to occur, the value 0 will be sent as the address. This multiplexer allows the correct register to be read or written to at the proper time.

*3) Compare Logic:* The compare logic uses the value on the bus as well as the flags from the IR to tell the control signals FSM whether or not to jump from a jump-compare instruction. It can do four different comparisons: bus == 0, bus != 0, bus ¡ 0, and bus ¿ 0. It computes all four at once, then selects which result (1 for true, 0 for false) to store in the D-latch at the end using the flags. Then the signal from the D-latch goes straight to the FSM. A simplified schematic for the compare logic is in Figure 26.

*4) 7-Segment Hex Displays:* The two sets of 7-segment displays on the board show what is in the instruction register and on the data bus. In a similar way to the ALU, they use flash chips (SST39SF020A) [18] as lookup tables, this time to convert the binary value on the data bus or in the IR to the signals corresponding to the correct segments on each display. The circuit also has a high-speed clock, a counter, and a decoder to quickly cycle between the four digits. Only one digit is on at a time, which lets us save on power and flash chips (since we only have to send data to one digit at a time as well, meaning we need fewer output pins), but because the clock cycles between them so quickly, it is impossible for your eyes to see the flickering, and it appears as though all four are on at once. The schematic for the two displays is in figure **??**.

### *E. Board 5*

*1) Overview:* This board will be referred to as board 5. It houses the stack and instruction RAM, the program counter (PC), the stack pointer counter (CSP), the dip switches used to set the interrupt handler location, the 40-pin programmer interface, and interrupt handling logic. It receives it control signals from board 2. The entire schematic can be seen in Figure 27.

*2) Program Counter:* The PC is a 16-bit value that needs to be able to be incremented, decremented, and loaded to. It stores the current address of the instruction RAM. The ideal

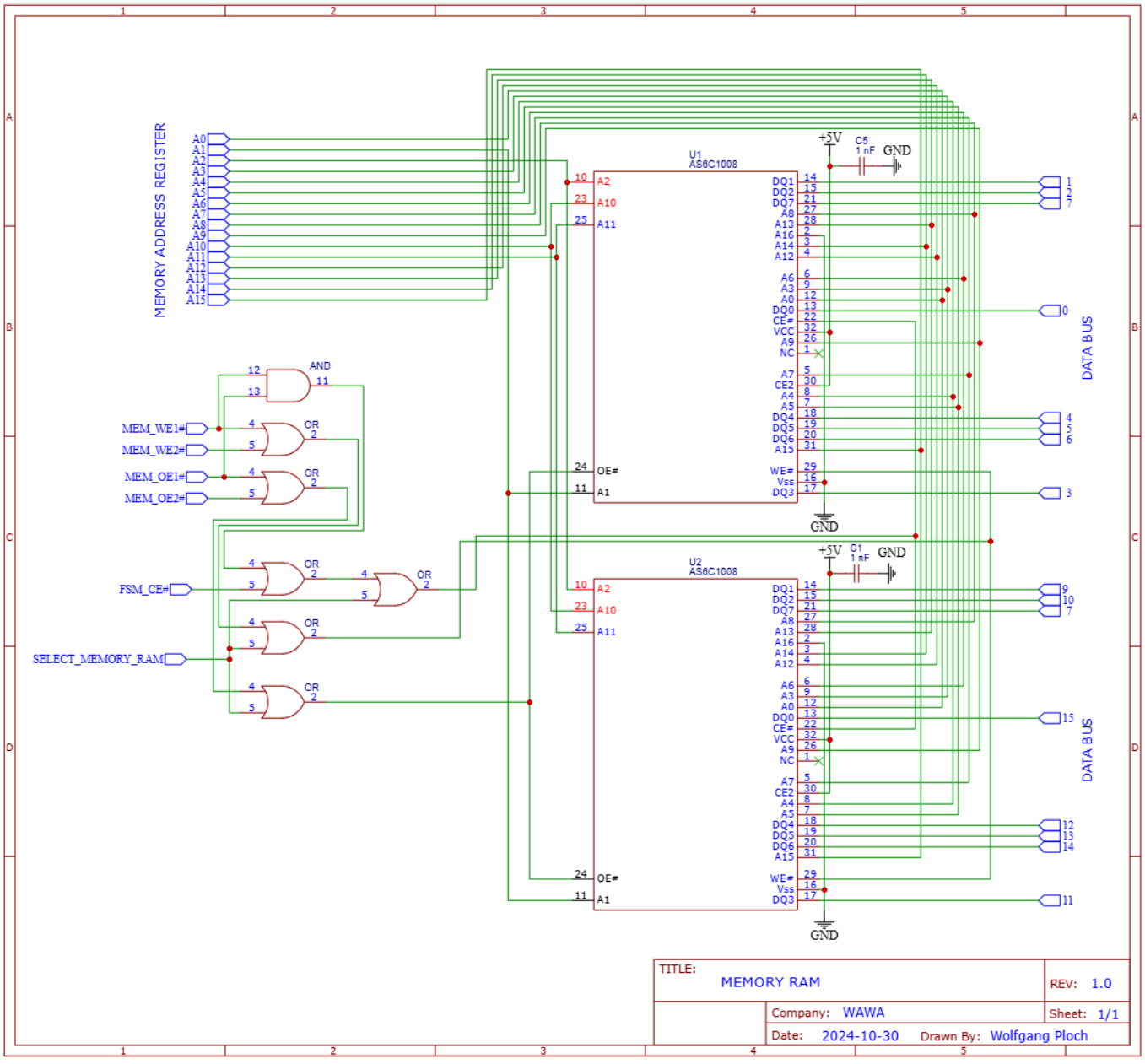


Fig. 23. Memory RAM and Supporting Circuitry

chip to allow for this operation is the 74LS193 counter IC [15]. A more detailed description of its operation can be found in the counter section of the description of board 2. Four of them are connected using the borrow and carry outputs to create a 16-bit counter that can be incremented, decremented and loaded to. The four data inputs on each chip are wired to their respective line on the data bus. The clear inputs of all the 74LS193 are wired to an active high version of the WE DELAY reset signal from board 2. The outputs of all the counters are wired to the inputs of two 74HC541 8-bit buffers [10]. These serve to buffer the outputs of the counter chips so that they do not need to drive the input addresses of the SRAM chips and the LEDs directly. Because the SRAM chips used to store the instructions need the value of PC constantly, these two buffers are constantly enabled. In addition, there are two more 74HC541s whose inputs connect to the buffered PC and whose outputs connect directly to the data bus. This is to allow the value of PC to be placed on the data bus. When pulling increment low to increment the counter, the decrement input needs to be held high and vice versa. Because of this, the PC++ and PC—control signals are treated as active low WE inputs. The control signals associated with the PC are PC++, PC–, PC OE, and PC WE. These signals conform to the stand outlined in the timing section of the board 2 description. Each

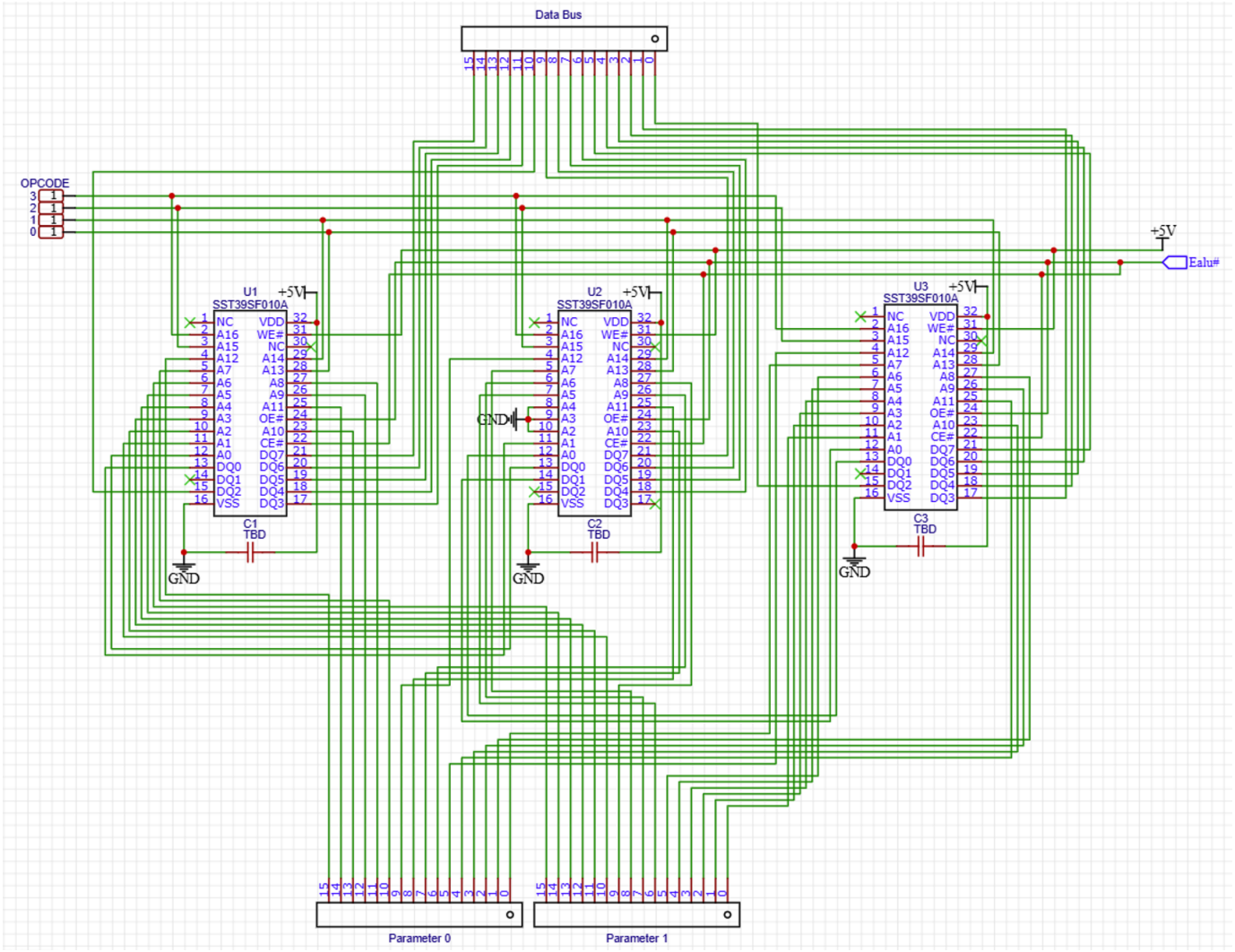


Fig. 24. ALU schematic. A table of connections is provided below, as this schematic is difficult to read.

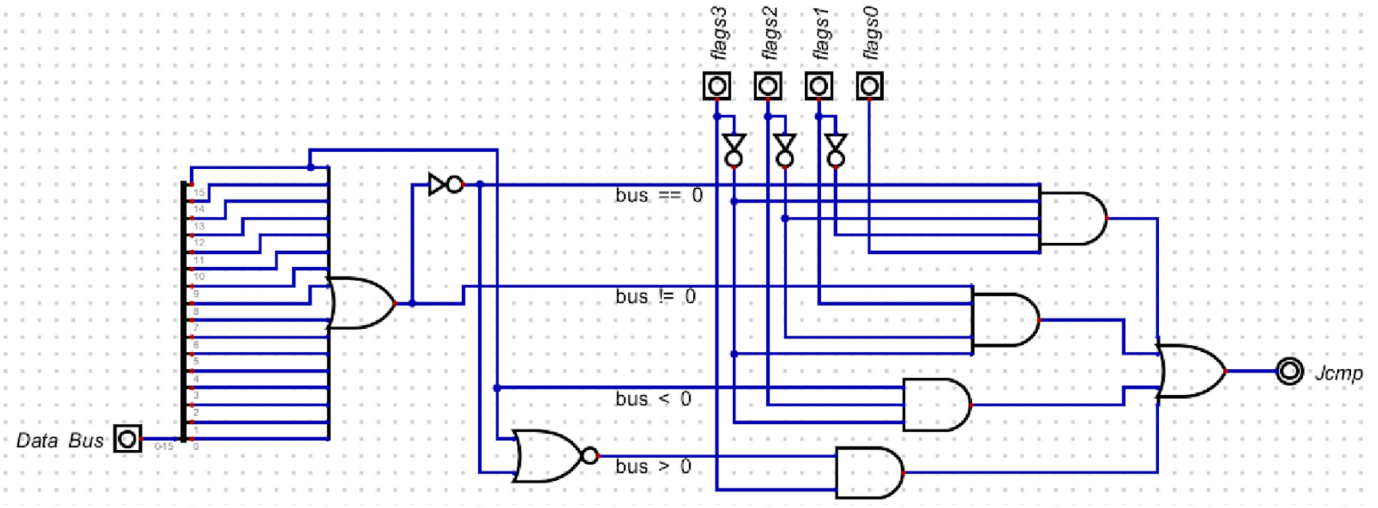


Fig. 25. A simplified schematic for the compare logic. The left side does the comparison, and the right side handles priority: flag3 has priority over flag2, which has priority over 1, which has priority over 0. The D-latch is not pictured.

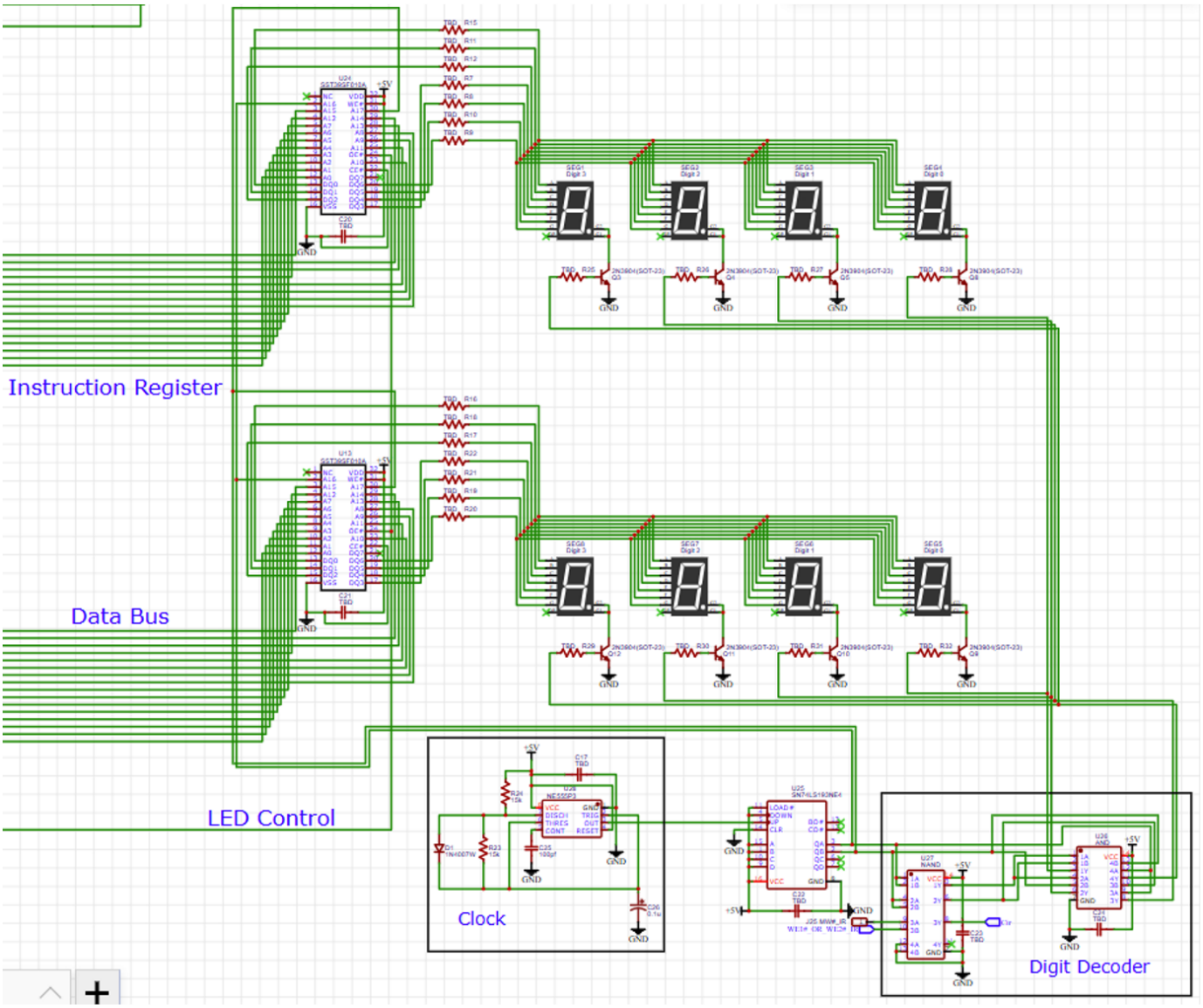


Fig. 26. The schematic for the hex displays. The circuitry at the bottom handles the digit cycling.

of these signals have to inputs (PC OE1# , PC OE2# , etc). The only one that is slightly different is the PC WE2# , it is different from all of the other WE2# as it is only high when the sub-sub counter is 0b10, not 0b01 and 0b10. This is because the 74LS193s [15] had a weird output state when they were being written to and nothing was on the bus. The PC and supporting circuitry can be seen in Figure 28.

*3) Stack Pointer Counter:* The stack pointer counter is used to store the current address of where the stack is pointing to. Its output connects to the address of the stack SRAM chips and nothing else. There is no need to jump to a specific location inside the stack, so the CSP only needs to be able to decrement and increment. Because of these requirements, it is exactly the same as the PC except it does not have the second set of buffers that connects it to the bus, and it does not have a WE input. The CSP and associated circuitry can be seen in Figure 29.

*4) Program Counter and Stack Pointer Counter LED Displays:* The 16 bits of the PC and 16 bits of the CSP are displayed by LEDs on the PCB arranged in the same way as the registers on board 1. The LEDs are grouped into 8s, each group connecting to an 8-820 Ω resistor array. The common pin of this array is connected to ground through a 2N3904 NPN transistor to turn on and off the LEDs. All of these are linked to the main LED control switch on the control panel. This portion of the circuit can be seen in Figure 30.

*5) Dip-switches for Interrupt Jump Location:* The computer used the address stored in a set of dip-switches to determine where to jump to when an interrupt occurs. This is accomplished by connecting on side of the dip-switches to 5 volts and pulling the other side of each one down with a 4.7 $k\Omega$ (two 8-resistor arrays were used for 16 dip-switches). The outputs of the switches are buffered by two 74HC541s [10].

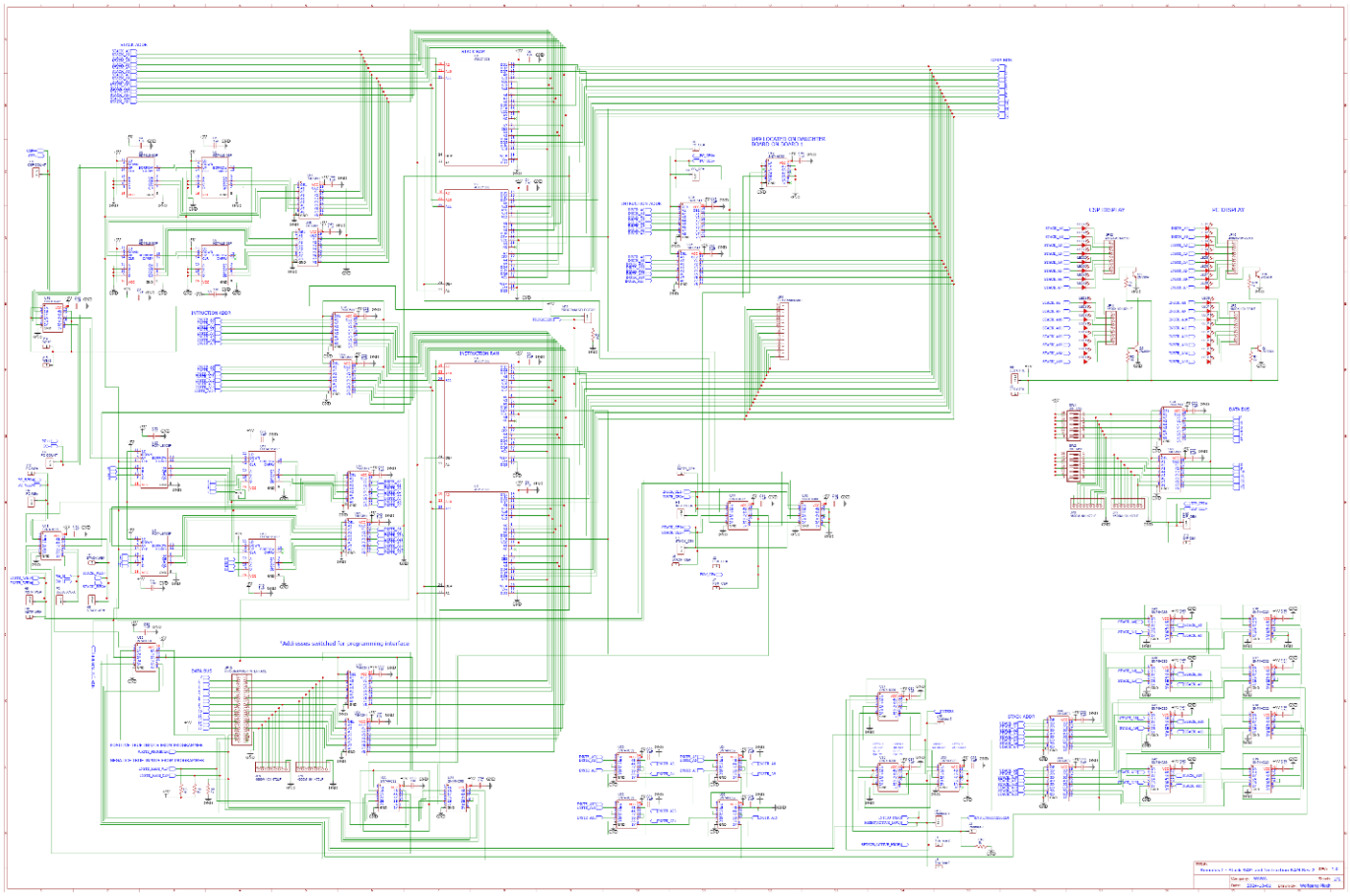

Fig. 27. Full Schematic of the Stack and Instruction RAM Board (Board 5)

Their OE connect to DIP OE1# and DIP OE2# . The outputs of the buffers connect to the bus. This circuit can be seen in Figure 31.

*6) Instruction SRAM:* The SRAM chips used for the instruction and stack RAM in the computer are the AS6C1008s [16]. The timing signals they require are discussed in the section on board 2. The computer technically does not need to have write access to the instruction RAM, the program is written by an external programmer and the computer simply reads it during operation. Write inputs to the instruction were added and the mistake was not realized until the PCB was designed. This was not a fatal error as these inputs were simply pulled high and not used. They still appear in the schematic. Two AS6C1008s had to be used as each only has 8 bits of storage per location. The IO pins on both chips connect directly to the data bus. 16 bits allows for the addressing of 64k memory. Only 32k and 128k versions of the chip were able to be found. The 128k version was selected and MSB of the address for both are connected to a switch on the control panel. This allows the user to store two programs in the computer at once and switch between them with the program switch. The PC does not connect directly to the address pins of the SRAM. It connects through a 1:2, 16-bit multiplexer that switches between the PC and pins of the 2x20 female header on the board. This multiplexer is made using four 74HC541 buffers [10] and some logic gates. There is a bit wired to the connector that changes control of the address pins between the PC and the programmer. The 2x20 female header also has 16 pins connected to the data bus. When the programmer is plugged in and it sets the WRITE PROGRAM bit high, it has control of the address pins of the SRAM as well as its WE and CE inputs. It then sequentially writes instructions by putting the instructions on the bus and incrementing the address bits. There are protections against conflicts where two devices are fighting for control of the bus. The WRITE PROGRAM input will only allow the programmer access if the PC is 0x0000 and the sub-sub-counter is 0b00. This ensures that there is nothing on the bus. The circuitry behind of this can be seen in Figure 32.

*7) Stack SRAM:* The stack SRAM uses two of the AS6C1008 chips [16] in the same arrangement as the instruction SRAM. Its circuitry is less complicated due to the fact that the CSP is the only thing connected to the address pins of the SRAM. The only supporting circuitry the stack SRAM needs are the logic gates to handle the multiple output and write enables and the logic to generate the local chip enable signal. The stack SRAM circuitry can be seen in Figure 33.

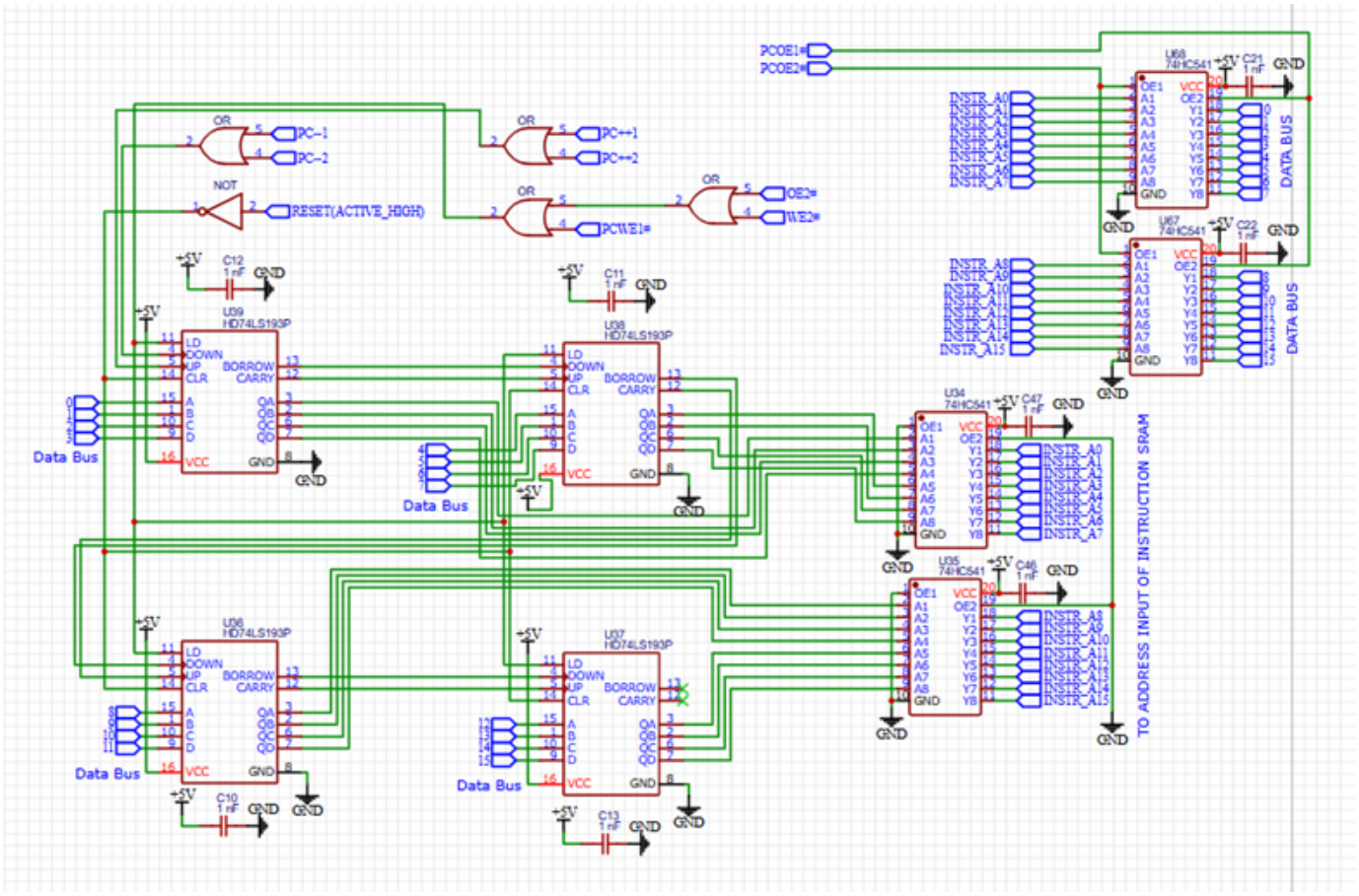


Fig. 28. Program Counter and Associated Circuitry

*8) Interrupt Handling:* The creation of the initial interrupt signal occurs on board 3. The interrupt inputs from the four IO slots are ORed together, this is the INT IN signal on board 5. The interrupt switch on the control panel disrupts this connection from board 3 to board 5 to enable or disable interrupts. To handle the interrupt correctly, the circuit needs to latch the incoming pulse long enough for board 2 to detect it. It also has to save the current value of CSP and disable any further interrupts until CSP returns to the value saved. Basically, it needs to disable interrupts from occurring until the current interrupt has been dealt with. The program will exit the function is pushed into the stack and it will return to its value before the interrupt was triggered. The SR latch that holds the initial triggering of the interrupt is called the primary interrupt latch. This primary latch will reset when the FSM sees the interrupt. The primary latch triggers a secondary latch that stays latched until the computer is done dealing with the interrupt function. This second latch blocks the first latch from triggering again. To determine when the CSP has returned to the starting value, two 74HC573s [9] are used. When there is not interrupt, they pass the CSP from the inputs to the outputs. The secondary latch latches the value of CSP on the inputs when an interrupt is triggered. This stored value is compared to the current value of CSP using logic and when there is a match, this resets the secondary latch. There is a delay in this comparison being able to reset the secondary latch as it would constantly reset itself before the computer jumps to the interrupt location. This is accomplished with logic before the reset input of the SR latch. The primary and secondary SR latches are also reset by the main computer reset signal. The SR latches used in this circuit are composed of NAND gates. The description of board 2 dives into these in more detail. This portion of the circuit can be seen in Figure 34.

The comparison of the stack and the saved stack location is done using XOR and OR gates. The SAME signal is blocked from reseting the secondary latch until the primary latch has been reset and the SETSUB signal is high. Once the SETSUB signal is high and the primary latch has been reset, the computer has already advanced PC to the interrupt function and incremented CSP.

## *F. Board 6*

*1) Overview:* The entire Romulus I computer is powered by 5 volts. The maximum current draw is around 4 amps. A large linear power supply with protection was designed to power the computer. The power supply PCB was designed to be able to handle 8 amps as a precaution. The entire Schematic of the

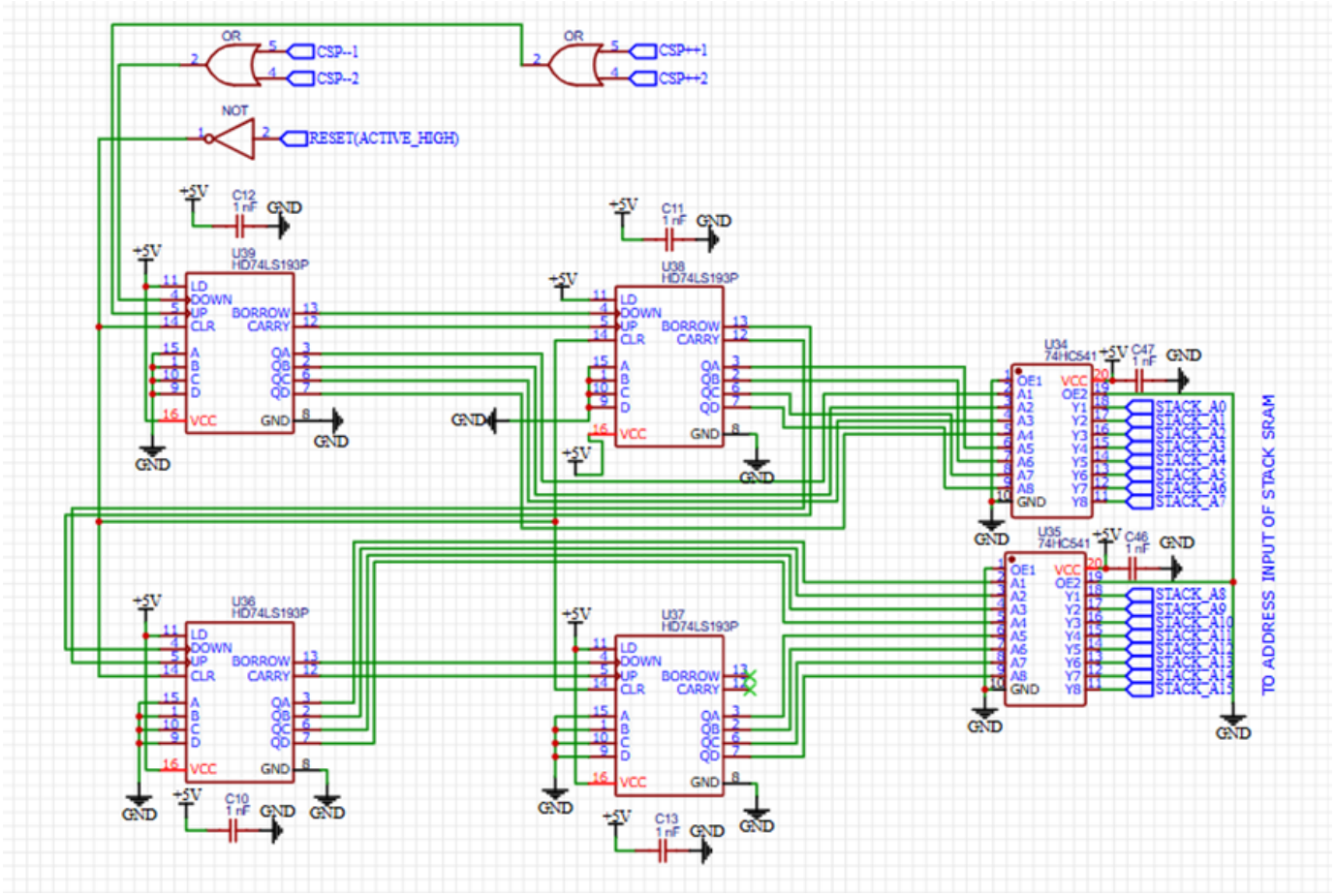


Fig. 29. CSP and Associated Circuitry

Power Supply Board can be seen in Figure 35. This board will be referred to as board 6 in other sections.

*2) Power Regulation Design:* A 24 VAC transformer with a current capacity of 4 amps was sourced and selected to step down the mains 120 VAC to a level that could be rectified. This transformer is center tapped with two 24 VAC windings. This means that two diodes can be used to rectify instead of a full-bridge rectifier. The diode used were the 6A05 which can handle a current flow of 6 amps. The rectified voltage is then smoothed out with 10000 uF of capacitance. A small 4.7 $k\Omega$ bleeder resistor was added to dissipate the capacitors in an absence of load. The rectified, unregulated voltage is referred to as "VCC" in the schematic. A simplified schematic of this section can be seen in Figure 36.

The rectified VCC is then used to create three different supplies, a 5 volt rail used to power the SR latch (SUB 5V+), a 12 volt rail used to power the op amp and comparator (+12V) and the main 5 volt rail used to power the computer (+5V). To create SUB 5V+, a 7805 linear voltage regulator was used to drop VCC down to 5 volts. This rail is not anticipated to have a large current draw but a small heat sink is attached to the regulator anyway for added protection. A 0.1 uF ceramic capacitor and a 10 uF electrolytic capacitor were connected to the input and output rails respectively. These protect against AC voltage present on the lines. The +12V supply is identical except for the replacement of the 7805 with a 7812. The main +5V supply had to be designed slightly differently because the 7805 can only supply a maximum current of 1 amp. To solve this problem, high-power 5 Ω resistor is placed between VCC and the input pin of the register. A 2N5684 (50 Amp PNP transistor) is then connected as follows: emitter to VCC, base to junction of 5 Ω resistor and voltage regulator and collector to output of voltage regulator. The circuit behaves like this, under low current draw, there is a low voltage drop across the resistor and the regulator provides most of the current. When a large current is drawn, the voltage across the resistor increases, pulling the base voltage of the 2N5684 lower causing it to conduct and supply the needed current to the output. These three supplies can be seen in Figure 37.

*3) Over Current and Over Voltage Protection:* Over voltage and over current protection are implemented to protect the computer. The over-voltage projection disconnections the +5V rail from the output if the +5V rail exceeds 6.2 volts. To do this, an LM339 [19] compares the +5V rail to a 6.2 reference voltage created with a 6.2 volt Zener diode and a 4.7 $k\Omega$ resistor connection in series on the +12V rail. The comparator

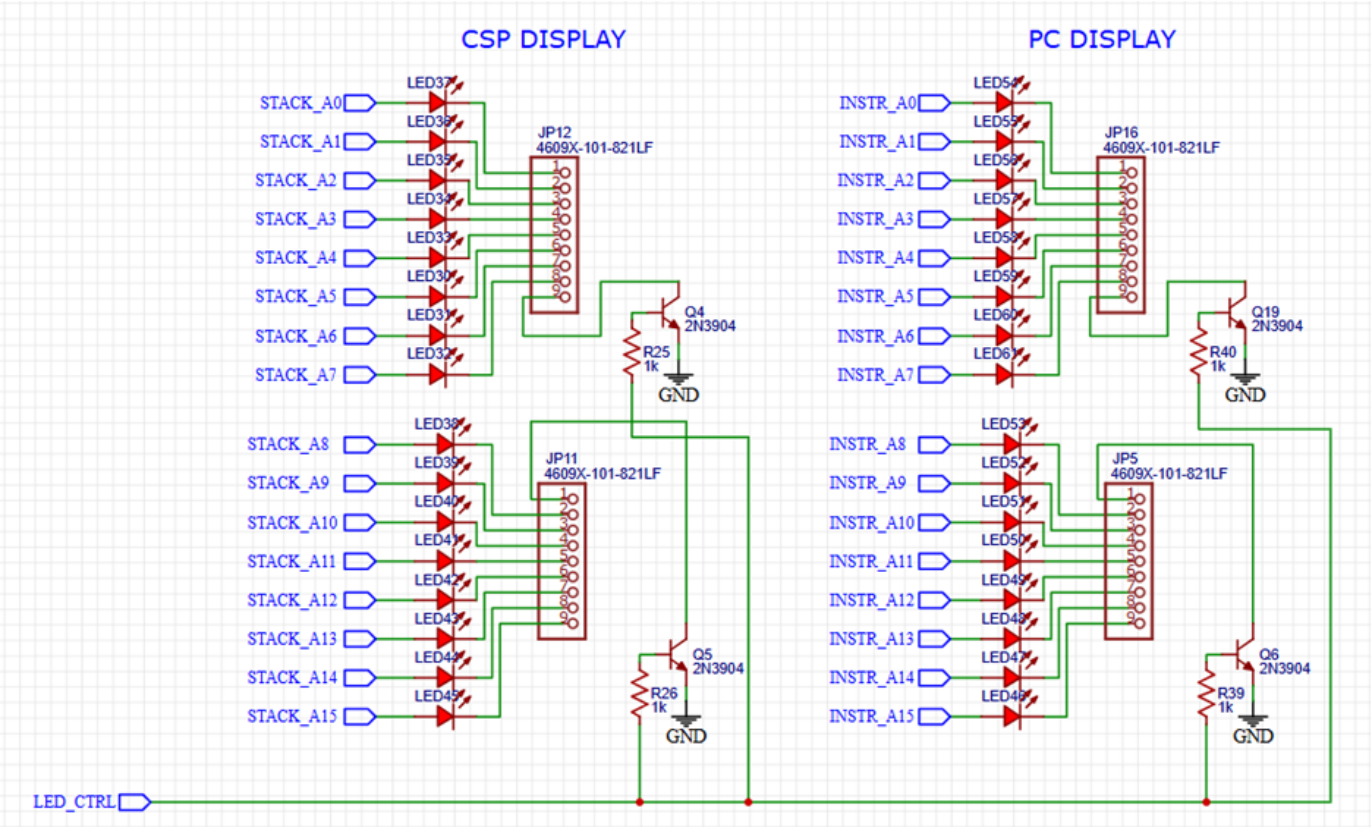


Fig. 30. PC and CSP LED Display Circuitry

pulls in the output low if the over-voltage condition is reached. To detect an over-current condition, three 0.1 Ω resistors were connected in parallel and then in series with the +5V rail and the output to the computer. As current is draw from the supply, a small voltage drop is created across these resistors. This drop is then amplified with a differential amplifier with a gain of 6.8. This differential amplifier is designed using the LM358 dual op amp chip [20]. This signal is then put through an RC low pass filter with a cutoff frequency of 3.39 Hz to remove noise amplified from the supply line. This signal is then compared to a reference voltage set using a 10 $k\Omega$ potentiometer mounted to the board. The output of the LM339 is pulled low if the over-current condition is met. The outputs of the LM339 are open collector meaning that they require a pull up resistor but also that they can be connected together to create an active low OR. This is done with the over-current and over-voltage to produce one active low output that indicates a fault has been reached. This combined output is pulled to +5V with a 4.7 $k\Omega$ resistor. This sets an SR latch built using NAND gates. The latch is reset by an RC network when the power supply is turned on. An LED turns on when a fault is detected and a relay is turned on with a 2N3904 to connect +5V to the output when no fault is detected. This way, if a fault is detected, the power supply has to be turned off and back on again to reset itself. The schematic of the protection circuitry described can be seen in Figure 38.

## VII. Test Plan

While our test plan was perhaps not as rigorous as it could have (or should have) been, it worked for our purposes. First, we tested the register file board (board 1) on its own.

Wires were soldered to the data bus and the control signals of the register. A circuit was built on a breadboard using two 74HC541 buffers [10] and 16 LEDs. The circuit allowed us to simulate a bus to connect the bus of the register file to. The control signals were then used to write 16-bit values to various registers and read them using the LEDs on the breadboard. It was confirmed that values could be written to and read from the register file.

The power supply was tested next. The board was wired to the external parts (the transformer, power resistor, ECG121 transistor etc.) and mounted on the case. The power wiring to the wall outlet was also completed. The transistor was originally soldering to the wrong pins creating an output voltage of 7 volts. This triggered the over voltage protection correctly and cut off the supply. This was not intended but ended up being a good way of testing the over-voltage protect. Once that problem was solved, the output showed a nice, clean 5 volts. The output was then shorted with a piece of wire to test the over-current protection. The supply shut off correctly which indicated that this feature was functional.

Board 2 was then tested. It was wired to power and the switches and indicators on the control panel. The oscillators

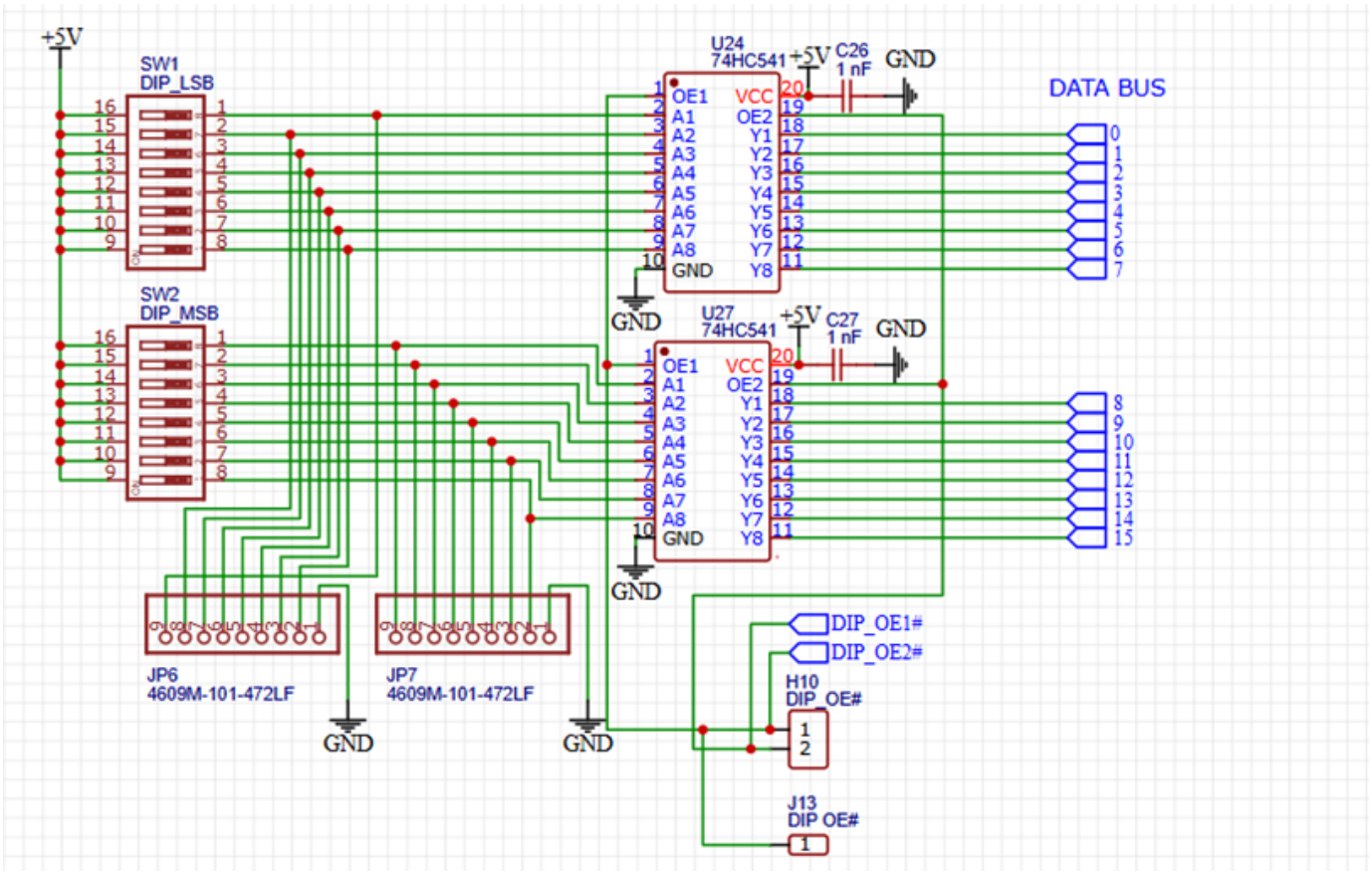


Fig. 31. Dipswitch Circuit Used to Set Interrupt Jump Location

were tested as well as the rotary switch circuit and the timing signals the board created. There was initially a problem with the rotary switch position connections due to an error in laying out the silkscreen but this was solved. There was also an issue with the FSM CE signal created due to it using the wrong bit of the sub sub counter. This was solved by cutting the trace and using a jumper wire. The finite state machine used to drive the manually increment modes was also tested and worked correctly.

After this, we decided it would be easiest to just put the computer together and troubleshoot each board as issues came up. The primary reason for this decision was that, if we wanted to test any other board individually, we would have to have used the Arduino Mega to simulate a data bus, something that we felt was unnecessary. We felt safe with this course of action because we were very confident in our designs, as we had checked them all several times between several people, and because each board was fairly independent, so if something went wrong on one, it would be easy to pinpoint where the issue was. These assumptions turned out to be true.

Our overall approach was to write small test programs of only a few instructions to test each opcode individually. We first tested to make sure we could write programs successfully. We ran into two issues here. The first was that this display for the instruction register had been wired incorrectly, so it was not displaying the value that had been stored. This we fixed in software by changing the data on the flash chips for the 7-segment displays. The second was that bits 4 and 5 of the instruction were always being set to 1. After some more testing, we found that this was an issue with the Arduino Mega we were using to write to instruction memory. We turned off serial communication to the laptop on the Mega (since that uses pins 4 and 5), which fixed the issue.

After those tests were completed, we moved on to testing individual instructions. We first tested register-to-register movement and immediate-to-register movement. We had to fix a small software issue where the PC was not incrementing twice during the load immediate instruction, but it otherwise went smoothly.

We then checked all of the ALU operations: bitwise AND, NOT, OR, and XOR, logical shift right and logical not, add, and subtract. The only issues with these were (1) a faulty latch on one of the parameter registers, and (2) a software fault in the ALU flash chip data, both of which were simple fixes: we replaced the latch and spent some time debugging the incorrect ALU.

At this point, we also tested register-to-memory movement, and memory-to-register movement, both of which worked just fine. During our testing, however, we uncovered a wiring issue with the program counter (since this was the first time we

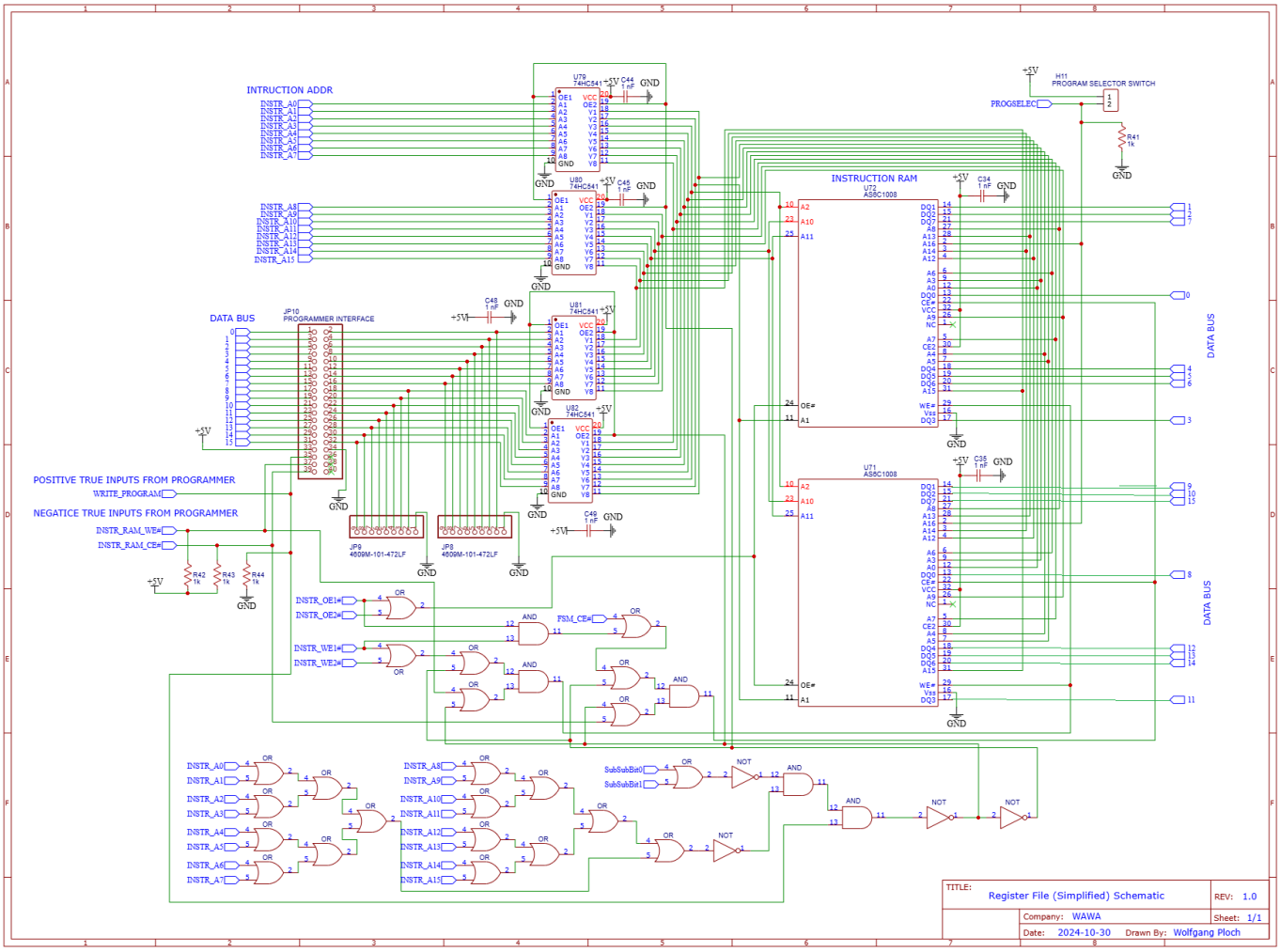


Fig. 32. Instruction SRAM, Programmer Interface, and Associated Circuitry

wrote a program longer than 16 instructions, and the issue was on bits 4-7) that we fixed with a few extra wires.

The last four operations we had to test were push, pop, unconditional jump, and conditional jump. There were once again two issues here. The first problem was that we had forgotten to put a latch on the output of the compare logic, so we added a daughter board with a few chips to create a latch matching the others on the board, with three write-enable signals: one active-low master write NANDed with two ORed active-low write signals, one from the control signals FSM and one from the sub-sub-clock (used for timing signals to prevent race conditions). The second problem was an FSM design issue: on the ”push PC” instruction, we should ave been pushing PC+2, not the PC+1 that we were pushing. This was easily fixed in software by changing the data on the FSM flash chip.

Finally, we tested the peripheral slots. We created a little makeshift input/output peripheral to test each slot. These tests revealed many small issues, most of which were not recorded because we were iterating so fast. After several hours of debugging, designing, testing, and iterating, we were able to get the peripheral slots working.

Our very last step was to run a comprehensive test program. It tested every instruction and stored a value in the last register to indicate which instructions had worked and which hadn’t. We had run the program throughout the testing process on several occasions, but up until this point, it had not worked properly. After fixing all of the bugs, however, the test program worked perfectly on every clock speed setting.

The remainder of the figures from this section are below.

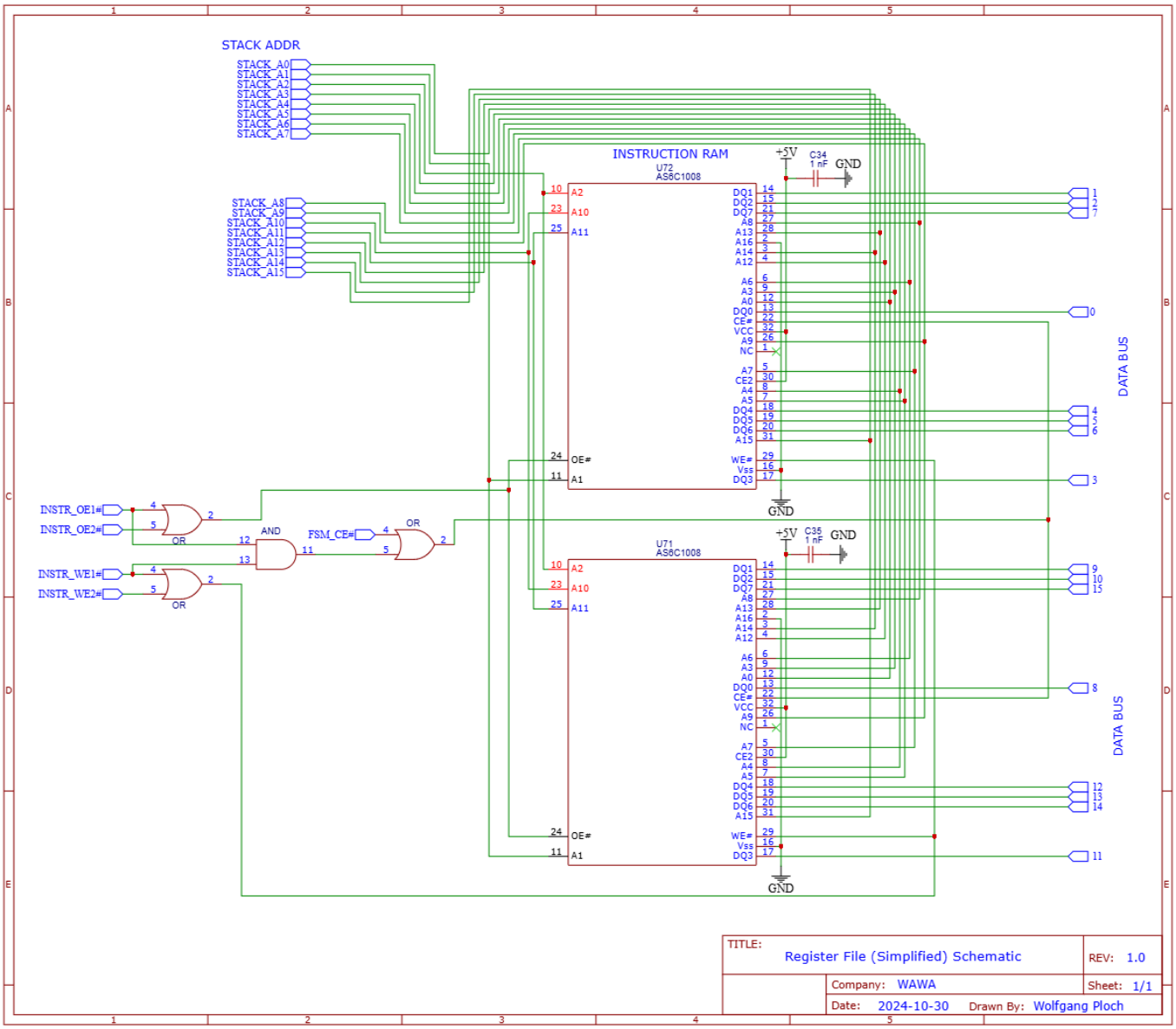


Fig. 33. Stack SRAM and Associated Circuitry

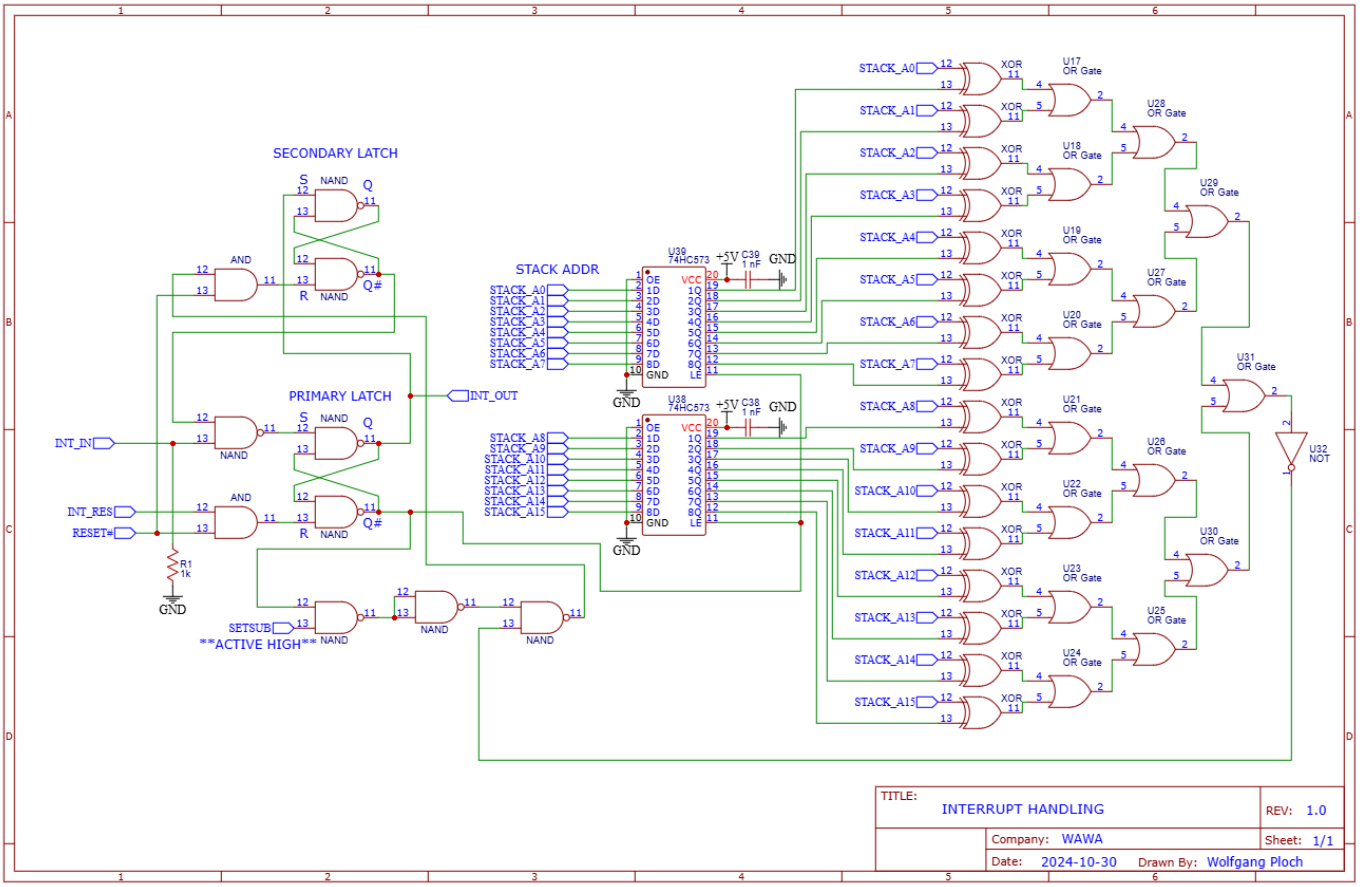


Fig. 34. Interrupt Handling Circuitry

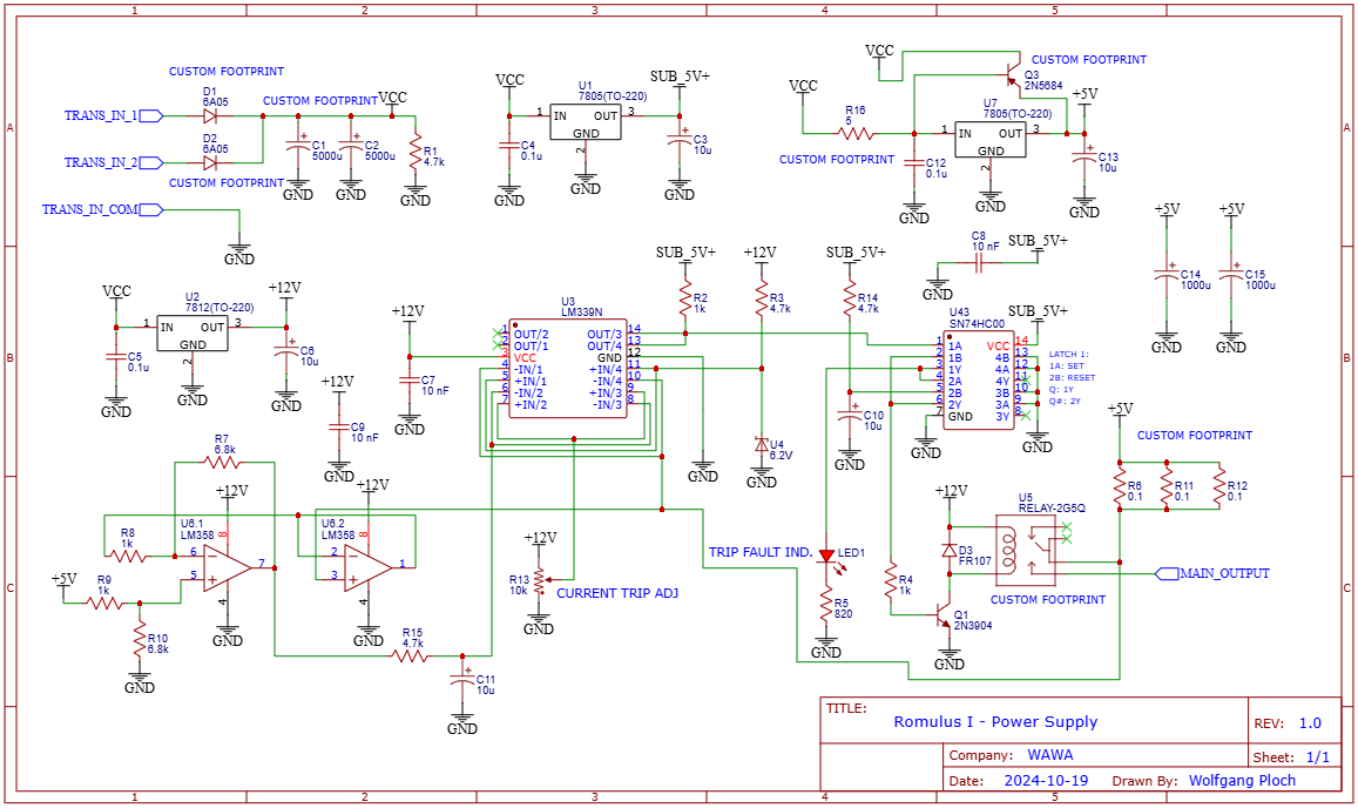


Fig. 35. Full Schematic of Power Supply Board

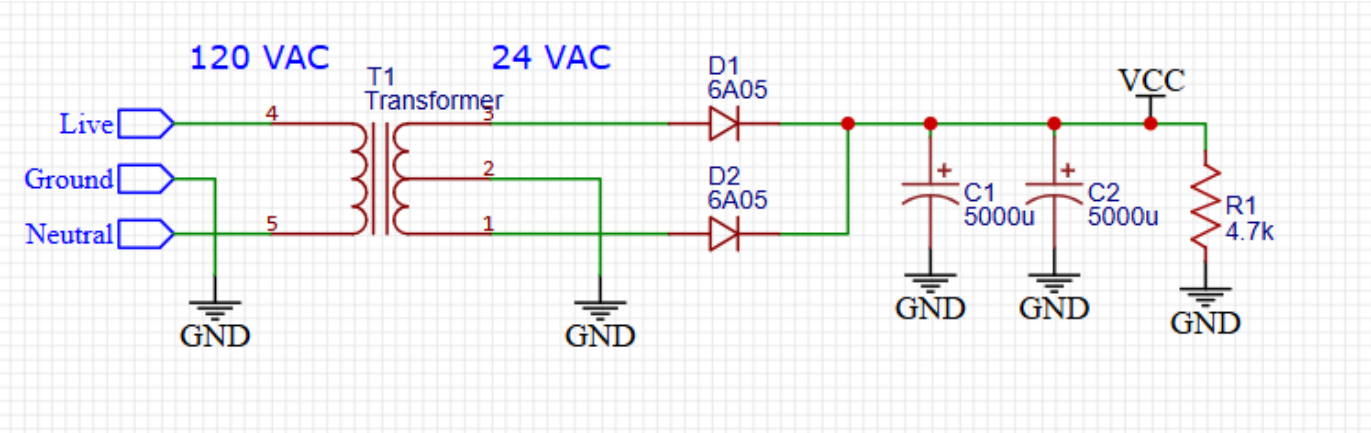


Fig. 36. Transformer and Rectifier Portion of Power Supply Schematic

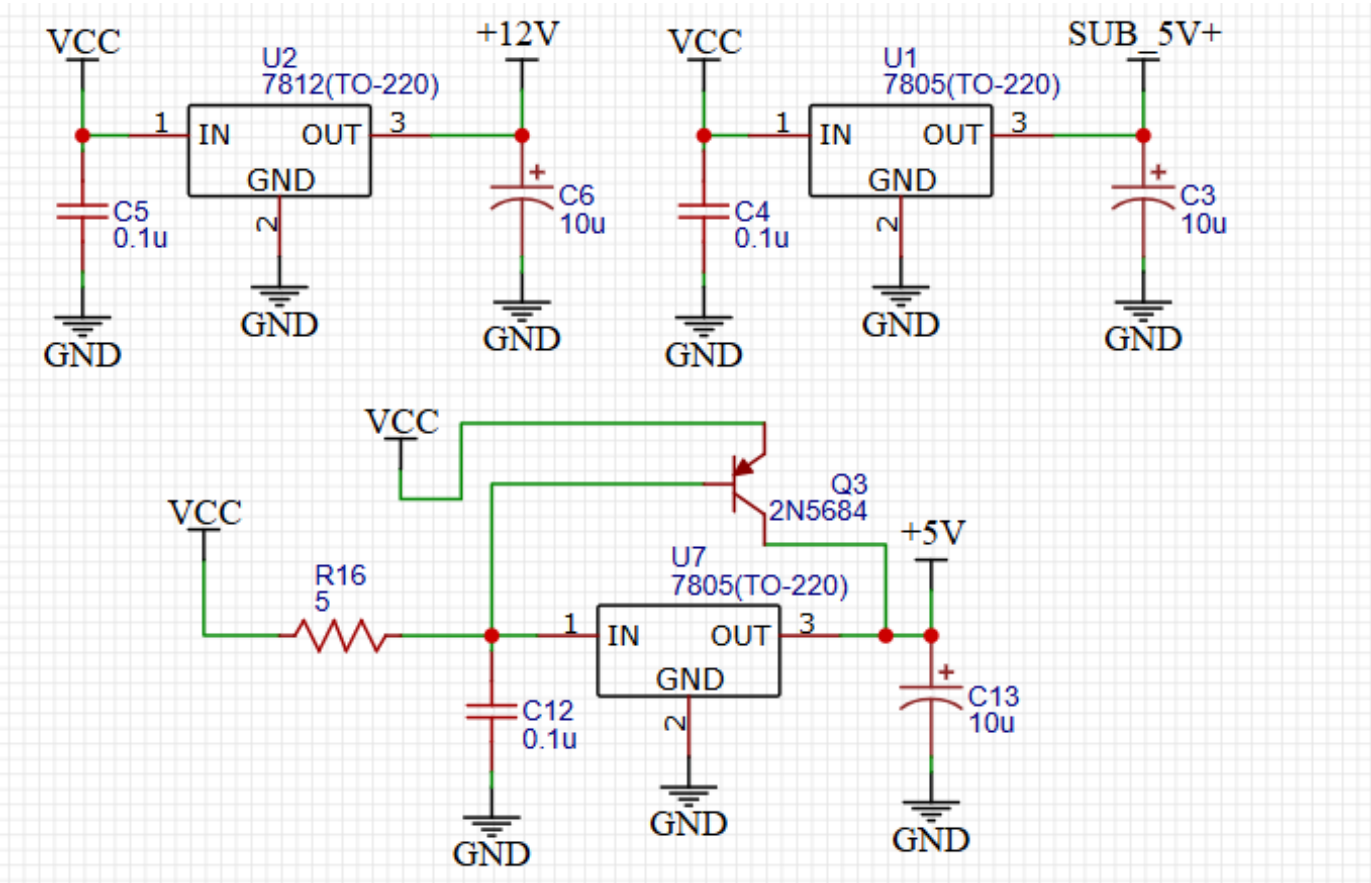


Fig. 37. Voltage Regulator Portion of the Power Supply Schematic

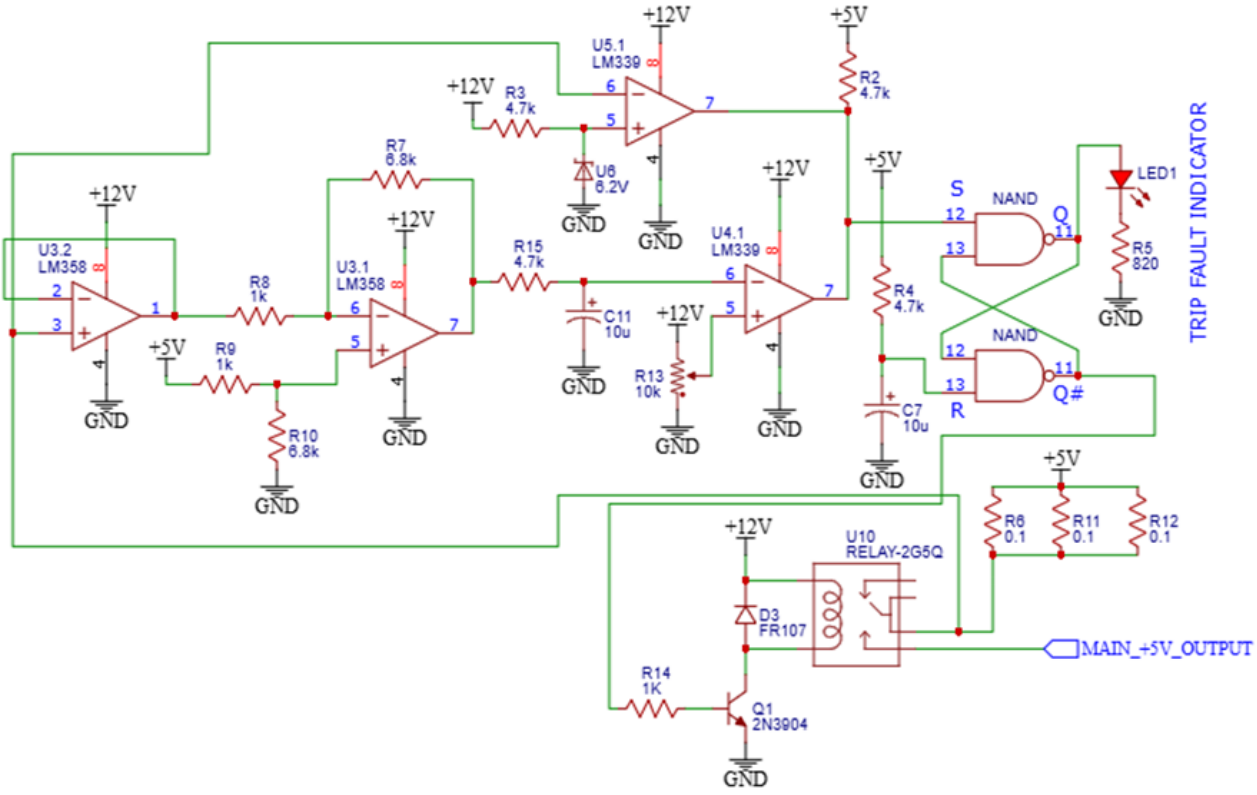


Fig. 38. Protection Circuitry Portion of Power Supply Schematic

## VIII. Physical Constraints

### A. Design and Manufacturing Constraints

Because the project is intended for the educational market, there were not many physical restraints placed on the project. Size was not important as this is not meant to be a pocket sized device. It just needs to be small enough for one person to reasonably carry it which it is. There were also no restraints on power consumption. This device did not need to be low power as it is not portable and does not need a battery. There were no CPU limitations as we designed the CPU ourselves so we could do whatever we needed. For instance, we chose for the stack RAM to be separate from the data RAM. This gives the advantage of having 64k stack locations and the user not being able over-right the stack by accident. This is usually not a commercially available feature in CPUs but we could implement it because we designed the CPU itself.

One limitation was PCB size. We ordered our PCBs from JLCPCB and their maximum size they will manufacture is 15x19 inches. Our 6 PCBs exceed that in total combined area, necessitating the circuit to be split among multiple PCBs

Part availability did constrain the project slightly. There are a lot of 74XX series chips that would have suited our circuits well that have unfortunately been discontinued. These parts are simply obsolete today. We were only able to use the 74LS154 decoder [21] because a group member had some in his possession, they have been discontinued on all major electronic part websites. Another part that was hard to find was 64k SRAM chips, only 32k and 128k chips could be found. This was used as a positive with the instruction RAM. The extra memory space was leveraged to allow the computer to store two programs simultaneously.

The time frame also constrained the project. While the computer itself functions perfectly as intended, we ran out of time to explore the peripherals we wanted to. For instance, there was no time to design a driver for the 64x64 RGB LED matrix intended to be used. The computer can function well with the Ploch Teletype peripheral, it acts as a terminal.

### B. Tools Used in the Project

*1) Visual Studio 2022:* Visual Studio 2022 was used to write the C++ programs associated with the computer. The simulator, assembler, and compiler were written in C++ using VS. Programs to generate the hex files needed to program the flash chips for the hexidecimal displays, finite state machine, and arithmetic and logic unit were also written in C++ using VS.

*2) Digital:* This software allows uses to drag and drop logic circuit components and create circuits. These circuits can then be tested with specific test cases or in an interactive mode. This software was used to validate parts of the circuits that were complex and in need of verification before manufacture (finite state machines, counter, etc.).

*3) Multisim:* Multisim was used briefly in the design of the power supply to verify the function of the differential amplifier and a filter used in the circuit.

*4) ANTLR:* ANTLR is a toolchain used for generating abstract syntax trees given a program, and a parser / lexer in the form of a context-free grammar. This was used extensively in the creation of romASM.

*5) Arduino IDE and Arduino Mega:* The Arduino Mega was used to program the flash chips that contain the encoding for the hexidecimal displays, the ALU and the FSM. This micro-controller was chosen as it had the required amount of digital pins to properly control the flash chips. It was also used in addition to a custom made shield in order to program the computer itself. Programs are uploaded to a micro-SD card and then the serial monitor of the Arduino IDE is used to transfer those programs to the computer via a 40-pin parallel cable. The Arduino Mega controls the data bus and address pins of the SRAM chip directly. This requires 35 digital pins which is again why the Arduino Mega was chosen for this task.

*6) Notepad:* The assembly or machine programs for the computer are written in a text editor (Notepad in this case). The machine code is directly uploaded to the computer while the assembly files are compiled to the bit code using an executable file and then uploaded to the computer.

*7) Physical Tools:* Tools used during the construction of the device included, soldering iron, de-soldering iron, electric drill, bandsaw, jigsaw, and a Dremel tool.

### C. Cost Constraints

Extra funding from the ECE department was received early in the design process prompting cost not to be an issue. There was no opulent spending although money wasn't paid much attention to. Some factors in the price come from the scaling of the device. Purchasing boards and components for the project would be much cheaper as more units are produced in during a run.

### D. Producing a Production Version

To produce a production version, a supplier for the transformer used in the power supply as well as the current meter would need to be found. These parts were found for free to use for the project. The power supply in general would probably be changed to a commercial supply due to the cost of the parts involved and the inefficiency of the current design. A simpler method would need to be established for connecting the boards together. the current method involves a lot of labor and could not be streamlined. The size of the device would also need to be reduced as it is pretty unwieldy in the current state.

## IX. Societal Impact

Our project could have impacts on a few different areas of society. The first impact, and the one we mainly intend, is on the academic field. Our hope is that this project will be used to teach college students how computers work in an physical, visual, easy-to-understand way. We hope that this, in turn, will help create better electrical and computer engineers who make more effective designs in their fields of work.

It is also possible that is has a small impact on the environment– if it ever becomes a mass-produced product (which we do not have plans for at the moment, but will not rule out), the computer does draw a considerable amount of current, which consumes more power and, if enough devices were to be sold, could have an impact on the environment. We find this outcome incredibly unlikely, however, and while we have not done any rigorous calculation, we believe that the environmental impact of a large collection of Romulus Is would be negligible compared to most major sources of pollution.

## X. External Standards

There are many standards that we needed to comply with when completing this assignment. One of the standards we adhered to were the Federal Communication Commission's standards for unintentional radiators. Specifically, code 15.101 states that and that "if a CPU board, power supply, or peripheral device will always be marketed with a specific personal computer, it is not necessary to obtain a separate authorization for that product provided the specific combination of personal computer" [22]. This gave us a lot of flexibility when it came to designing shielding and power circuitry, because our goal is not to make independent components to market, but rather to create the CPU as a whole.

Other standards we had to meet were the standards regarding safely isolating the power supply [23]. To meet this standard, all the wall power we get is confined to an electrical box, and the rest of our circuit is able to be displayed out in the open, because it falls under the category of "Safety Extra-Low Voltage" [23], and therefore do not require additional protection. This related to the Institute of Electrical and Electronics Engineers' (IEEE) Recommended Practice for Powering and Grounding Electronic Equipment [24]. We addressed this by implementing the over-voltage and over-current protection circuitry as discussed in the description of Board 6 above.

Another standard we followed when designing the boards was Institute of Printed Circuits (IPC) Standard 2221 for Printed Circuit Boards [25]. This gave us equations and plots to use to calculate correct trace widths and separations for the consistent results.

## XI. Intellectual Property Issues

We believe that Romulus I is patentable. We looked at a few patents potentially similar to our project to get a sense of what inventions were in the space already.

The first patent we looked at was the Kenbak-1 personal computer. It was created as a teaching tool, designed to help students learn how to write simple programs [26]. The patent has only one independent claim describing the specifications of the device, and no dependent claims [27].

While this sounds similar to our own product, ours is significantly different in a number of ways. First, the Kenbak-1 is not at all transparent– it's just a box with some buttons and a few lights. Romulus I has displays and lights for many different parts, making it more transparent. Our device also has multiple clock speeds and the manual step option, neither of which were in the Kenbak-1. It is also generally more powerful, with a higher maximum clock speed and instructions per second, more registers, more storage space, and more operations [26].

The second patent we looked at was the Apple I computer by Steve Wozniak. This was another, slightly more recent personal computer with support for a screen and keyboard [28]. The patent has two independent claims, both of which focus on the technology used to connect to a video display, and six dependent claims, which discuss individual parts of the independent claims [29].

Our product is once again different for many of the same reasons. Although the Apple 1 is just a PCB with chips on it, so the user can see the entire circuit laid out, it is not designed for readability and education, like ours is. It also again lacks the adjustable clock speed and manual step option [28].

The last patent we looked at was a digital logic simulator by Yifatch Tzori. There are four independent claims: One that discusses the overall system, one that focuses on the hardware with the logic chips in it, one involving specific algorithm the software uses, and one that describes another, faster-performing system. The 36 dependent claims again discuss smaller systems contained within the full product [30].

This product is also significantly different from ours. It cannot run actual programs, as it is a logic simulator, and it is not specifically designed for education, like ours is.

## XII. Timeline

Our original plan for our GANTT chart was generally split into $\frac{1}{3}$s, as shown in Figures 39, 40, and 41 in the appendix.

The first $\frac{1}{3}$ of the project involved much of the theoretical work: designing the ISA, confirming control signals logic, testing the feasibility of the design, and creating lower level schematics for the different modules. Once the ISA has been specified, we are able to parallelize operations. While some of us worked on the hardware and schematics, others worked on a simulator to test programs written for the CPU, as well as the flash programming. As the core CPU PCB design reaches its end, we planned to shift gears during the lead time. The software team would begin work on an assembler and/or compiler, and the hardware team would work on design of peripherals. Towards the end of the project, we all planned to work together on system-wide integration, testing, and tying up any final loose ends. As a general guide, August and Austin were given software projects, while Wolfgang and Will were given hardware projects. We purposefully gave ourselves extra time at the end, so that we could add additional time when we inevitably took longer to complete a section than we planned. The GANTT chart at the end of our project is shown in Figures 42, 43, and 44 in the appendix.

In the beginning, we generally followed the GANTT chart closely, and did a good job staying on schedule. When we reached the designs of the PCBs, however, we realized that we did not allot enough time for all the complicated design

work. Therefore, we extended the time we took in this process, however, since we decided to split up the boards, we were able to work on different boards in different processes, so we never had any dead time on the hardware side waiting for all the boards to come in. Our test plan had slightly changed since our original GANTT chart, so the testing process was more spread out, and then concentrated at the end for system testing. Another change we made, was the design for the power circuitry. We originally thought it made the most since to design it at the end when we knew how much current we needed, however it made more sense to make it earlier, so we could test it and use it to power our boards.

## XIII. Costs

This project required the use of many different components. It also utilized parts that are not available anymore and parts that were received from surplus locations for free. A detailed spreadsheet of the part, source, quantity, individual cost when ordering 1, 10, 25, 50, 100, 1000, 2500, 5000, 10000, and 25000 can be found in the Appendix in tables VII, VIII, IX and X in the Appendix. Parts that have an asterisk (*) were received for free and the costs are an estimate. Parts that are marked with a double asterisk (**) are slightly different compatible models than the ones used in the project as some parts were obsolete or could not be located. The ”NA” locations in the tables indicate that pricing data at the quantity was not necessary for calculations. The dashes ”-” in the part number are there to format the table correctly and do not actually appear in the model number. The 6 boards are also included in the spread sheet. The entry simply called ”hardware” refers to the wood, screws, junction box etc. that were purchased from Lowes. None of these items could be scaled down with quantity so they were grouped into one entry for convenience. The table also includes the necessary components to make the programmer for the computer, because one would likely need to be included with every unit. The shield for programming the flash chips is not included because this would likely be done in the factory while assembling. The costs in the sheet neglect the shipping costs of the items. The cost to manufacture one device (Including shipping) was roughly $ 1000 excluding the previously mentioned parts received for free. This cost also includes the cost of five PCBs rather than an individual one as the minimum order quantity was five. The estimated costs for manufacturing one unit in a run of 100 and 10,000 neglect the price of shipping. These values ended up being large due to the amount of parts that did not need to be purchased in building the one device. The price for producing one in a run of 100 was $ 668.41. The price for producing one in a run of 10,000 was $ 599.12. As expected, the price decreases when the number of units in a run increases. The price of labor could not be accounted for in this price which would likely be considerate compared to the cost of the components. The manufacturing of the boards would not be able to be streamlined using a pick and place machine and reflow oven as all of the components on the boards are through hole devices and those tools only work for SMD components.

## XIV. Final Result

The computer functions completely as we initially intended. Programs can be uploaded in the bitcode we designed via the programmer device, and they can be run without errors. All sixteen opcodes work correctly: register-to-register, immediate-to-register, register-to-memory, memory-to-register, bitwise AND, NOT, OR, XOR, logical not, logical shift right, add, subtract, push, pop, unconditional jump, and conditional jump. All sixteen general purpose registers, the memory address register, the two ALU parameter registers, the program counter, and the stack counter all properly display their values in binary through their LEDs. The data bus and instruction register properly display their values in hex through their 7-segment displays [31]. The clock can run at 8 Hz, 400 Hz, 4 MHz, or it can be stepped one instruction or FSM state at a time with a button. The program can be paused and resumed at any time, and the computer can be reset to the beginning of the program with another button. Finally, each of the four peripheral slots can send and receive data, and the control signals can handle an interrupt request from any of the four slots.

The software also works as intended. The instruction set architecture is Turing-complete. The simulator accurately predicts how the physical computer will act and displays the data and stack RAMs for debugging purposes. The assembler correctly implements a few extra instructions (logical shift left, call, and return), labels, and correctly assembles a program written in our assembly language to our bitcode.

We believe, according to the expectations outlines in our project proposal, that our project deserves an A. We were able to perfectly implement the CPU design we proposed, including an assembler and working memory-mapped I/O peripheral slots and some working peripherals.

## XV. Engineering Insight

There were many new tools that we had to learn to use for this project. For most of them, at least one team member had used in it the past, so having them as a resource for learning was very helpful. Some of these include ANTLR, Digital, and Arduino just to name a few.

Working on this project really showed how having a clear plan and outline, as well as good and open communication with group members is very important to team success. In projects for previous courses, they are usually small enough or have enough assistance from the professor, that you do not necessarily have to create a great plan to get it done, and you did not even have to have the greatest communication between group mates. However, for this project, us taking the time to meet and discuss our plans and thought processes for the project early in the semester helped get us on the same page and kept us on track throughout the semester. Since we had these discussions early on, we generally knew what to expect and already divided up what we were going to plan on working on for the project, so we did not end up facing many issues with teamwork, communication, and morale. This, however, I

believe proves how important setting a good foundation early in the project was.

Advice I would pass onto a future capstone student would first be to choose a project that your group will actually enjoy working on. You will have a much easier time working on a project you want to see come to completion rather than one that is simply an assignment. Secondly, I would get started on the project early on in the semester. We were able to work on the bells and whistles of our project, because we started meeting up early in the semester. A semester is a honestly a short time to complete an entire capstone project, so make sure to use all the time you are given. Lastly, as mentioned earlier, meet with your group early to set expectations and discuss how each member can best help the team, one another, and how you plan on completing the project. The sooner everyone is on the same page, the smoother the process will be.

## XVI. Future Work

In designing, building and testing Romulus, there are of course things that we wish we did, things we wish we didn't do, and tasks that fell through the cracks. In hindsight, we would definitely rework how we handle interrupts. Currently, the address of our ISR is specified by a DIP switch on the board. If future iterations give access to the ISR address programmatically, for example by binding it to a memory location, there would be large benefits in terms of the adaptability of the assembler. We also have no way to determine which port triggered an interrupt, and must poll through each to find out which. It would be more ideal to use another mapped location to set bits to determine which port triggered the interrupt. There is also lots of value in designing a compiler that rests on top of the assembler, and allows the user to abstract away the concepts of registers, jumps and memory management altogether, giving a feasible top-down approach to learning about computer architecture. We designed the syntax and most of the operations for such a compiler, but were unable to realize the full program due to time constraints. Lastly, we think there is lots of value to implementing some form of graphics, either using an LED matrix of sorts, or creating a VGA driver peripheral, as this will expand the capabilities of the computer dramatically.

## References


[1] H. Neemann, "hneemann/digital," original-date: 2016-06-28T17:40:16Z. [Online]. Available: https://github.com/hneemann/Digital

[2] Toy ISA simulator. [Online]. Available: https://researcher111.github.io/uva-cso1-F23-DG/homework/files/toy-isa-sim.html

[3] B. Eater. Breadboard computer kits. [Online]. Available: https://eater.net

[4] Texas Instruments, "Quadruple 2-input and gates," 74HC08 datasheet, [Revised Sept. 2021].

[5] ——, "Quadruple 2-input nand gates," sNx4HC00 datasheet, Dec. 1982 [Revised Aug. 2021].

[6] ——, "Quadruple 2-input or gates," 74HC32 datasheet, [Revised Apr. 2021].

[7] ——, "Hex inverters," 74HC04 datasheet, [Revised Apr. 2021].

[8] ——, "Quadruple 2-input xor gates," 74HC86 datasheet, [Revised Apr. 2021].

[9] ——, "Octal transparent d-type latches with 3-state outputs," sNx4HC573A datasheet, Dec. 1982 [Revised Apr. 2022].

[10] ——, "Octal buffers and line drivers with 3-state outputs," sNx4HC541 datasheet, Jan. 1996 [Revised May 2022].

[11] Microchip Technology, "Ic flash 1mbit parallel 32dip," sST39SF010A-70-4C-PHE datasheet, May 2022 [Revised Apr. 2016].

[12] Texas Instruments, "Dual voltage controlled oscillators," 74S124 datasheet, [Revised Apr. 2004].

[13] ——, "555 timer," xx555 datasheet, [Revised Sept. 2014].

[14] ——, "Dual d-type positive-edge-triggered flip-flops with clear and preset," sNx4HC74 datasheet, [Revised June 2021].

[15] ——, "Synchronous 4-bit up/down counters (dual clock with clear)," 74LS193 datasheet, [Revised Mar. 1988].

[16] Alliance Memory Inc., "128k x 8 bit low power cmos sram," aS6C1008 datasheet, [Revised Mar. 2000].

[17] Texas Instruments, "Dual 4-input positive-and gates," 74LS21 datasheet, [Revised Mar. 1988].

[18] Microchip, "Multi-purpose flash," sST39SF010A / SST39SF020A datasheet, [Revised Apr. 2016].

[19] Texas Instruments, "Comparator," lM339 datasheet, [Revised Oct. 2023].

[20] ——, "Dual op amp," lM358 datasheet, [Revised Mar. 2022].

[21] Fairchild Semiconductor, "4-line to 16-line decoder/demultiplexer," 74LS154 datasheet, [Revised Mar. 2000].

[22] *Radio Frequency Devices*. Federal Communications Commission Title 47, Chapter 1, Subchapter A, Part 15, 2024.

[23] Power supply safety standards, agencies, and marks. [Online]. Available: https://www.cui.com/catalog/resource/power-supply-safety-standards-agencies-and-marks

[24] R. Holleman, "IEEE recommended practice for powering and grounding electronic equipment," pp. 1–408, conference Name: IEEE Std 1100-1999. [Online]. Available: https://ieeexplore.ieee.org/document/798784/?arnumber=798784&tag=1

[25] "IPC-2221a(l).pdf." [Online]. Available: https://www-eng.lbl.gov/~shuman/NEXT/CURRENT_DESIGN/TP/MATERIALS/IPC-2221A(L).pdf

[26] Kenbak-1 computer. [Online]. Available: https://www.kenbak.com/kenbak-1-home

[27] L. D. Amdahl, G. M. Amdahl, H. L. Engel, E. J. Schneberger, and J. V. Blankenbaker, "Stored logic computer," patentus 3 246 303A. [Online]. Available: https://patents.google.com/patent/US3246303A/en

[28] Apple i microcomputer. [Online]. Available: https://americanhistory.si.edu/collections/object/nmah_1692121

[29] S. G. Wozniak, "Microcomputer for use with video display," patentus 4 136 359A. [Online]. Available: https://patents.google.com/patent/US4136359A/en

[30] Y. Tzori, "Digital logic simulation/emulation system," patentus 5 748 875A. [Online]. Available: https://patents.google.com/patent/US5748875A/en

[31] Lite-On Electronics, "Display 7seg 0.39" sgl red 10dip," lTS-4802BJR-H1 datasheet.


## XVII. Appendix

TABLE VII
COMPONENT, SOURCE, AND COSTS AT VARIOUS QUANTITIES PART 1

| Part | Source | Quantity | Cost per 1 (USD) | Cost per 10 (USD) | Cost per 25 (USD) | Cost per 50 (USD) | Cost per 100 (USD) | Cost per 500 (USD) | Cost per 1,000 (USD) | Cost per 2,500 (USD) | Cost per 5,000 (USD) | Cost per 10,000 (USD) | Cost per 25,000 (USD) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Board 1 PCB | JLCPCB | 1 | 27.48 | NA | NA | NA | 12.55 | NA | NA | NA | NA | 10.84 | NA |
| Board 2 PCB | JLCPCB | 1 | 8.04 | NA | NA | NA | 2.31 | NA | NA | NA | NA | 1.80 | NA |
| Board 3 PCB | JLCPCB | 1 | 8.38 | NA | NA | NA | 2.38 | NA | NA | NA | NA | 1.87 | NA |
| Board 4 PCB | JLCPCB | 1 | 11.60 | NA | NA | NA | 5.15 | NA | NA | NA | NA | 4.29 | NA |
| Board 5 PCB | JLCPCB | 1 | 10.56 | NA | NA | NA | 4.13 | NA | NA | NA | NA | 3.38 | NA |
| Board 1 PCB | JLCPCB | 1 | 2.10 | NA | NA | NA | 0.79 | NA | NA | NA | NA | 0.55 | NA |
| 74HC08 | Mouser | 15 | 0.73 | 0.63 | 0.63 | 0.63 | 0.482 | 0.381 | 0.305 | 0.276 | 0.257 | 0.248 | 0.238 |
| 74HC00 | Mouser | 19 | 0.60 | 0.382 | 0.382 | 0.382 | 0.309 | 0.295 | 0.263 | 0.255 | 0.249 | NA | 0.248 |
| 74HC04 | Mouser | 4 | 0.73 | 0.63 | 0.63 | 0.63 | 0.482 | 0.381 | 0.305 | 0.276 | 0.257 | 0.248 | 0.238 |
| 74HC32 | Mouser | 31 | 0.65 | 0.567 | 0.567 | 0.567 | 0.434 | 0.343 | 0.274 | 0.248 | 0.231 | 0.223 | 0.214 |
| 74HC86 | Mouser | 5 | 0.97 | 0.861 | 0.861 | 0.861 | 0.671 | 0.554 | 0.504 | 0.347 | 0.347 | 0.347 | 0.347 |
| 74LS21 | Mouser | 2 | 0.86 | 0.766 | 0.766 | 0.766 | 0.596 | 0.493 | 0.389 | 0.363 | 0.345 | 0.332 | 0.321 |
| 74LS193 | Mouser | 11 | 1.38 | 1.24 | 1.24 | 1.24 | 0.964 | 0.796 | 0.628 | 0.586 | 0.557 | 0.536 | 0.536 |
| 74HC541 | Mouser | 62 | 1.41 | 1.27 | 1.27 | 1.27 | 0.991 | 0.819 | 0.646 | 0.603 | 0.603 | 0.603 | 0.603 |
| 74HC573 | Mouser | 51 | 1.05 | 0.993 | 0.993 | 0.993 | 0.727 | 0.600 | 0.474 | 0.442 | 0.420 | 0.404 | 0.391 |
| AS6C-1008 | Mouser | 6 | 5.11 | 4.34 | 4.34 | 4.34 | 4.34 | 3.99 | 3.84 | 3.71 | 3.70 | 3.70 | 3.70 |
| 74LS154 | Jameco | 2 | 7.95 | 7.25 | 7.25 | 7.25 | 7.25 | 7.25 | 7.25 | 7.25 | 7.25 | 7.25 | 7.25 |
| SST39-SF010A | Mouser | 6 | 2.89 | 2.89 | 2.79 | 2.79 | 2.79 | 2.69 | 2.69 | 2.69 | 2.69 | 2.69 | 2.69 |
| SST39-SF020A | Mouser | 2 | 3.36 | 3.36 | 3.26 | 3.26 | 2.26 | 3.09 | 3.09 | 3.09 | 3.09 | 3.09 | 3.09 |
| NE555 | Mouser | 5 | 0.32 | 0.193 | 0.193 | 0.193 | 0.169 | 1.62 | 0.160 | 0.148 | 0.148 | 0.148 | 0.148 |
| 74S124 | Mouser | 1 | 7.66 | 7.66 | 6.51 | 6.48 | 6.35 | 6.18 | 5.33 | 5.33 | 5.33 | 5.33 | 5.33 |
| 74HC74 | Mouser | 2 | 0.96 | 0.86 | 0.86 | 0.86 | 0.671 | 0.554 | 0.437 | 0.408 | 0.388 | 0.373 | 0.361 |
| LM358 | Mouser | 1 | 0.800 | 0.481 | 0.481 | 0.481 | 0.481 | 0.410 | 0.381 | 0.381 | 0.369 | 0.369 | 0.369 |
| LM339 | Mouser | 1 | 0.88 | 0.533 | 0.533 | 0.533 | 0.335 | 0.251 | 0.224 | 0.190 | 0.190 | 0.149 | 0.143 |
| 1 nF Capacitor | Digikey | 228 | 0.24 | 0.139 | 0.139 | 0.099 | 0.086 | 0.065 | 0.058 | 0.051 | 0.047 | 0.047 | 0.047 |
| 2N3904 | Digikey | 52 | 0.100 | 0.056 | 0.056 | 0.056 | 0.038 | 0.029 | 0.025 | 0.022 | 0.020 | 0.019 | 0.016 |
| 1 $k\Omega$ Resistor | Digikey | 79 | 0.160 | 0.078 | 0.078 | 0.051 | 0.044 | 0.031 | 0.027 | 0.027 | 0.021 | 0.019 | 0.017 |
| 4.7 $k\Omega$ Resistor Array | Digikey | 14 | 0.480 | 0.270 | 0.218 | 0.186 | 0.160 | 0.114 | 0.100 | 0.100 | 0.075 | 0.075 | 0.075 |
| 820 $\Omega$ Resistor Array | Mouser | 42 | 0.430 | 0.316 | 0.254 | 0.188 | 0.188 | 0.118 | 0.085 | 0.085 | 0.085 | 0.085 | 0.085 |
| Red LEDs | Digikey | 354 | 0.300 | 0.167 | 0.167 | 0.167 | 0.102 | 0.077 | 0.069 | 0.063 | 0.063 | 0.052 | 0.052 |
| 820 $\Omega$ Resistor | Digikey | 18 | 0.250 | 0.128 | 0.128 | 0.086 | 0.074 | 0.054 | 0.048 | 0.048 | 0.038 | 0.035 | 0.035 |
| 10 $k\Omega$ Resistor | Digikey | 4 | 0.100 | 0.033 | 0.024 | 0.014 | 0.014 | 0.014 | 0.014 | 0.014 | 0.010 | 0.009 | 0.008 |
| 20 $k\Omega$ Resistor | Digikey | 3 | 0.100 | 0.078 | 0.054 | 0.032 | 0.022 | 0.013 | 0.013 | 0.013 | 0.011 | 0.009 | 0.008 |
| 2.2 $k\Omega$ Resistor | Digikey | 1 | 0.100 | 0.079 | 0.054 | 0.031 | 0.022 | 0.010 | 0.010 | 0.010 | 0.010 | 0.010 | 0.010 |

TABLE VIII
COMPONENT, SOURCE, AND COSTS AT VARIOUS QUANTITIES PART 2

| Part | Source | Quantity | Cost per 1 (USD) | Cost per 10 (USD) | Cost per 25 (USD) | Cost per 50 (USD) | Cost per 100 (USD) | Cost per 500 (USD) | Cost per 1,000 (USD) | Cost per 2,500 (USD) | Cost per 5,000 (USD) | Cost per 10,000 (USD) | Cost per 25,000 (USD) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 100 nF Capacitor | Digikey | 10 | 0.240 | 0.136 | 0.136 | 0.096 | 0.084 | 0.063 | 0.057 | 0.050 | 0.046 | 0.046 | 0.046 |
| 10 uF Capacitor | Digikey | 8 | 0.100 | 0.100 | 0.100 | 0.076 | 0.067 | 0.052 | 0.050 | 0.050 | 0.050 | 0.050 | 0.050 |
| 100 uF Capacitor | Digikey | 2 | 0.310 | 0.310 | 0.310 | 0.267 | 0.232 | 0.207 | 0.181 | 0.172 | 0.172 | 0.172 | 0.172 |
| 470 nF Capacitor | Digikey | 1 | 0.350 | 0.232 | 0.232 | 0.185 | 0.153 | 0.119 | 0.105 | 0.105 | 0.105 | 0.105 | 0.105 |
| 30 pF Capacitor | Digikey | 1 | 0.720 | 0.437 | 0.437 | 0.323 | 0.288 | 0.228 | 0.209 | 0.189 | 0.177 | 0.177 | 0.177 |
| 5 pF Capacitor | Digikey | 1 | 0.560 | 0.338 | 0.338 | 0.247 | 0.219 | 0.171 | 0.157 | 0.144 | 0.134 | 0.123 | 0.117 |
| 1N4148 | Digikey | 3 | 0.100 | 0.055 | 0.055 | 0.055 | 0.034 | 0.024 | 0.021 | 0.018 | 0.016 | 0.014 | 0.014 |
| 15 $k\Omega$ Resistor | Digikey | 2 | 0.100 | 0.078 | 0.054 | 0.32 | 0.022 | 0.013 | 0.013 | 0.013 | 0.011 | 0.009 | 0.008 |
| 150 $\Omega$ Resistor | Digikey | 14 | 0.100 | 0.045 | 0.031 | 0.020 | 0.017 | 0.013 | 0.013 | 0.013 | 0.010 | 0.008 | 0.008 |
| 4.7 $k\Omega$ Resistor | Digikey | 3 | 0.100 | 0.045 | 0.033 | 0.020 | 0.020 | 0.020 | 0.017 | 0.017 | 0.011 | 0.009 | 0.008 |
| 6.8 $k\Omega$ Resistor | Digikey | 2 | 0.100 | 0.078 | 0.054 | 0.032 | 0.022 | 0.013 | 0.013 | 0.011 |  | 0.009 | 0.008 |
| 6.2V Zener diode | Digikey | 1 | 0.100 | 0.057 | 0.057 | 0.057 | 0.029 | 0.026 | 0.023 | 0.023 | 0.022 | 0.022 | 0.022 |
| 1000 uF Capacitor | Digikey | 1 | 0.530 | 0.327 | 0.327 | 0.242 | 0.215 | 0.169 | 0.154 | 0.193 | 0.129 | 0.129 | 0.113 |
| LM7805 | Digikey | 2 | 1.74 | 1.279 | 1.279 | 1.103 | 1.103 | 0.970 | 0.934 | 0.906 | 0.876 | 0.856 | 0.856 |
| LM7812 | Digikey | 1 | 1.74 | 1.279 | 1.279 | 1.103 | 1.103 | 0.970 | 0.934 | 0.906 | 0.876 | 0.856 | 0.856 |
| 6A04-G** Diode | Digikey | 2 | 0.890 | 0.553 | 0.553 | 0.553 | 0.359 | 0.275 | 0.248 | 0.219 | 0.201 | 0.201 | 0.170 |
| 0.1 $\Omega$ Chassis Resistor** | Digikey | 3 | 3.38 | 2.298 | 1.905 | 1.664 | 1.664 | 1.419 | 1.419 | 1.419 | 1.419 | 1.419 | 1.419 |
| 5000 uF Capacitor** | Digikey | 2 | 7.690 | 5.315 | 5.315 | 4.354 | 4.354 | 3.680 | 3.495 | 3.342 | 3.196 | 3.196 | 3.196 |
| 10 $k\Omega$ Potentiometer* | Digikey | 1 | 1.050 | 1.050 | 1.050 | 1.050 | 1.050 | 1.050 | 1.050 | 1.050 | 1.050 | 1.050 | 1.050 |
| Transformer | Generic | 1 | 40.000 | 40.000 | 40.000 | 40.000 | 40.000 | 40.000 | 40.000 | 40.000 | 40.000 | 40.000 | 40.000 |
| 40 Pin Connector | Digikey | 6 | 1.090 | 1.044 | 0.918 | 0.918 | 0.918 | 0.666 | 0.567 | 0.567 | 0.567 | 0.567 | 0.567 |
| 12V Relay** | Digikey | 1 | 1.220 | 1.067 | 1.012 | 0.971 | 0.932 | 0.848 | 0.814 | 0.814 | 0.741 | 0.741 | 0.741 |

TABLE IX
COMPONENT, SOURCE, AND COSTS AT VARIOUS QUANTITIES PART 3

| Part | Source | Quantity | Cost per 1 (USD) | Cost per 10 (USD) | Cost per 25 (USD) | Cost per 50 (USD) | Cost per 100 (USD) | Cost per 500 (USD) | Cost per 1,000 (USD) | Cost per 2,500 (USD) | Cost per 5,000 (USD) | Cost per 10,000 (USD) | Cost per 25,000 (USD) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| KSA101-0YTU** | Digikey | 1 | 1.760 | 1.760 | 1.760 | 0.840 | 0.750 | 0.593 | 0.542 | 0.499 | 0.453 | 0.440 | 0.440 |
| 5 Ω Chassis Resistor | Digikey | 1 | 5.820 | 3.497 | 2.897 | 2.529 | 2.221 | 1.813 | 1.813 | 1.813 | 1.813 | 1.813 | 1.813 |
| 1 $M\Omega$ Resistor | Digikey | 4 | 0.100 | 0.035 | 0.028 | 0.020 | 0.017 | 0.013 | 0.013 | 0.013 | 0.010 | 0.009 | 0.009 |
| 7 Segment Display | Digikey | 8 | 1.310 | 0.817 | 0.817 | 0.817 | 0.565 | 0.466 | 0.436 | 0.405 | 0.405 | 0.376 | 0.365 |
| 32 PDIP Socket** | Amazon | 8 | 0.344 | 0.344 | 0.344 | 0.344 | 0.344 | 0.344 | 0.344 | 0.344 | 0.344 | 0.344 | 0.344 |
| 8 Dip Switch Package | Digikey | 2 | 2.960 | 2.960 | 2.227 | 2.089 | 2.089 | 1.780 | 1.665 | 1.557 | 1.424 | 1.328 | 1.328 |
| DC Current Gauge** | Electronix Express | 1 | 10.95 | 10.95 | 10.95 | 10.95 | 10.95 | 10.95 | 10.95 | 10.95 | 10.95 | 10.95 | 10.95 |
| Rotary Switch** | Amazon | 1 | 9.09 | 9.09 | 9.09 | 9.09 | 9.09 | 9.09 | 9.09 | 9.09 | 9.09 | 9.09 | 9.09 |
| Panel Mount LED Holders | Amazon | 4 | 1 | 0.315 | 0.315 | 0.315 | 0.315 | 0.315 | 0.315 | 0.315 | 0.315 | 0.315 | 0.315 |
| Panel Mount Push Buttons** | Parts Express | 4 | 1.250 | 1.250 | 1.250 | 1.250 | 1.250 | 1.250 | 1.250 | 1.250 | 1.250 | 1.250 | 1.250 |
| Panel Mount Toggle Switches** | Parts Express | 3 | 1.340 | 1.340 | 1.340 | 1.340 | 1.340 | 1.340 | 1.340 | 1.340 | 1.340 | 1.340 | 1.340 |
| Male Header Pins | Amazon | 248 | 0.020 | 0.020 | 0.020 | 0.020 | 0.020 | 0.020 | 0.020 | 0.020 | 0.020 | 0.020 | 0.020 |
| Hardware | Various | 1 | 110.630 | 110.630 | 110.630 | 110.630 | 110.630 | 110.630 | 110.630 | 110.630 | 110.630 | 110.630 | 110.630 |
| Heat Sink** | West Florida Components | 1 | 16.000 | 16.000 | 16.000 | 16.000 | 16.000 | 16.000 | 16.000 | 16.000 | 16.000 | 16.000 | 16.000 |
| PC Fan** | Star Tech | 1 | 16.990 | 16.990 | 16.990 | 16.990 | 16.990 | 16.990 | 16.990 | 16.990 | 16.990 | 16.990 | 16.990 |
| 22 AWG Wire Pack | Amazon | 2 | 19.990 | 19.990 | 19.990 | 19.990 | 19.990 | 19.990 | 19.990 | 19.990 | 19.990 | 19.990 | 19.990 |
| EMI Filter** | Digikey | 1 | 9.540 | 7.783 | 7.783 | 6.951 | 6.539 | 5.854 | 5.506 | 5.313 | 5.313 | 5.313 | 5.313 |
| Solder Reel | Amazom | 2 | 12.990 | 12.990 | 12.990 | 12.990 | 12.990 | 12.990 | 12.990 | 12.990 | 12.990 | 12.990 | 12.990 |
| PCB Standoff Pack | Amazon | 1 | 9.990 | 9.990 | 9.990 | 9.990 | 9.990 | 9.990 | 9.990 | 9.990 | 9.990 | 9.990 | 9.990 |
| Arduino Mega Shield | Digikey | 1 | 6.800 | 6.800 | 6.800 | 6.800 | 6.800 | 6.800 | 6.800 | 6.800 | 6.800 | 6.800 | 6.800 |

TABLE X
COMPONENT, SOURCE, AND COSTS AT VARIOUS QUANTITIES PART 4

| Part | Source | Quantity | Cost per 1 (USD) | Cost per 10 (USD) | Cost per 25 (USD) | Cost per 50 (USD) | Cost per 100 (USD) | Cost per 500 (USD) | Cost per 1,000 (USD) | Cost per 2,500 (USD) | Cost per 5,000 (USD) | Cost per 10,000 (USD) | Cost per 25,000 (USD) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Arduino Mega Compatible | Amazon | 1 | 22.990 | 22.990 | 22.990 | 22.990 | 22.990 | 22.990 | 22.990 | 22.990 | 22.990 | 22.990 | 22.990 |
| SD Card Break Out Board | Amazon | 1 | 3.000 | 3.000 | 3.000 | 3.000 | 3.000 | 3.000 | 3.000 | 3.000 | 3.000 | 3.000 | 3.000 |
| 16 GB Micro SD Card** | Digikey | 1 | 25.980 | 23.054 | 21.982 | 21.204 | 20.452 | 19.498 | 19.498 | 19.498 | 19.498 | 19.498 | 19.498 |
| USB Type B Cable** | Amazon | 1 | 9.890 | 9.890 | 9.890 | 9.890 | 9.890 | 9.890 | 9.890 | 9.890 | 9.890 | 9.890 | 9.890 |

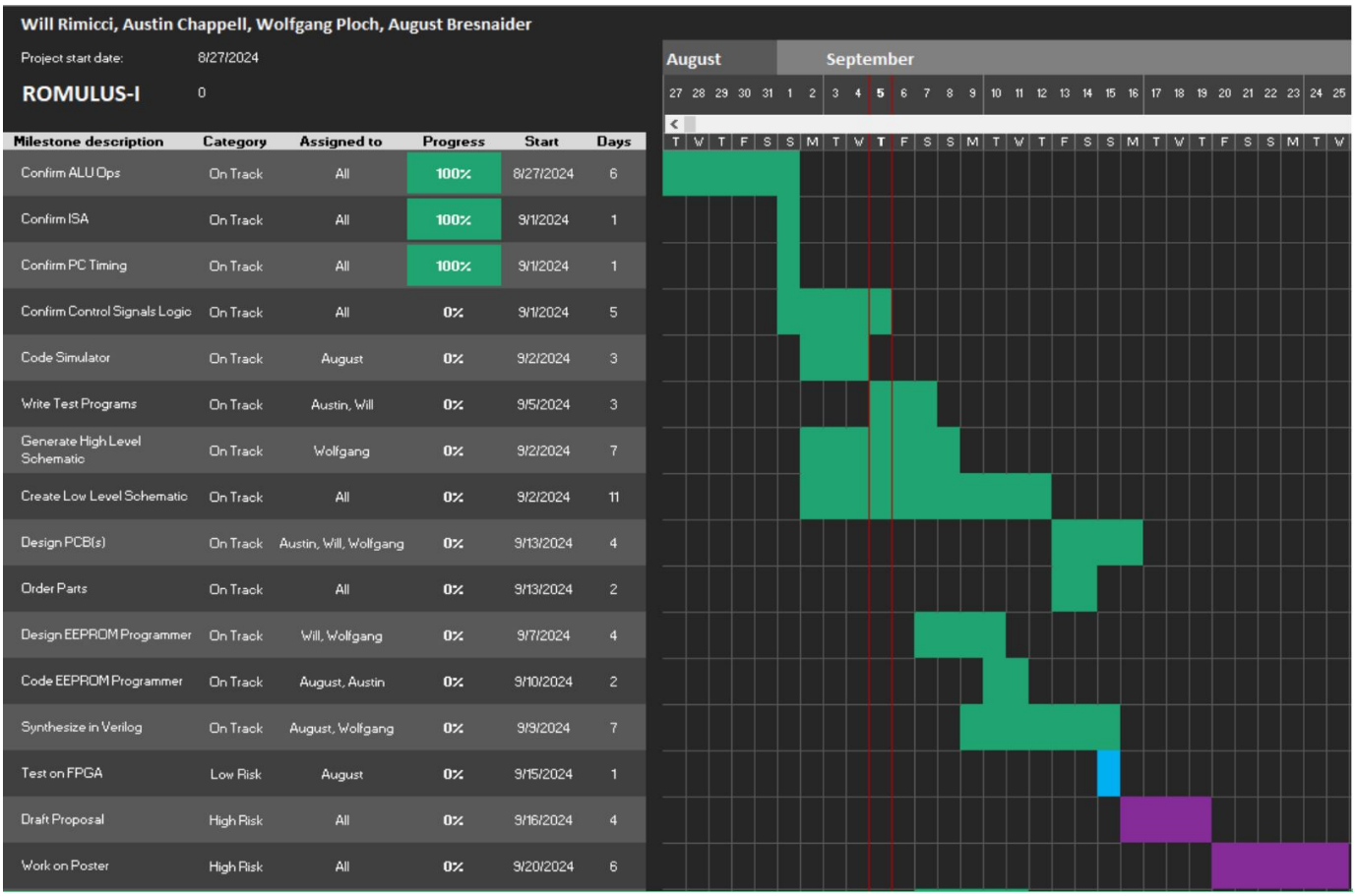


Fig. 39. The first section of our original GANTT chart

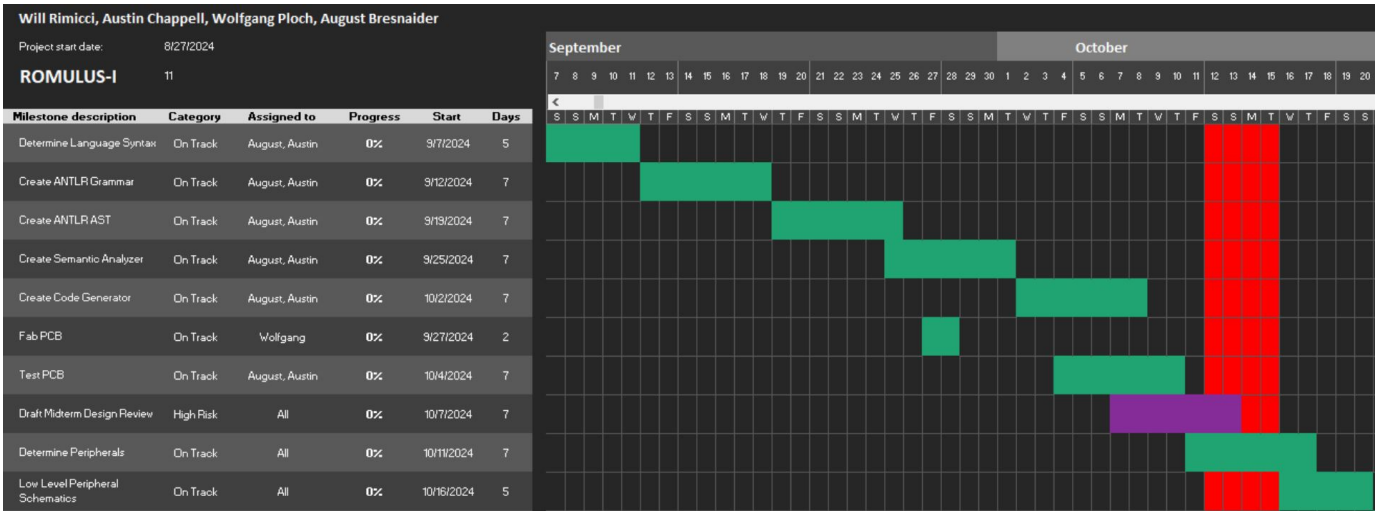


Fig. 40. The middle section of our original GANTT chart

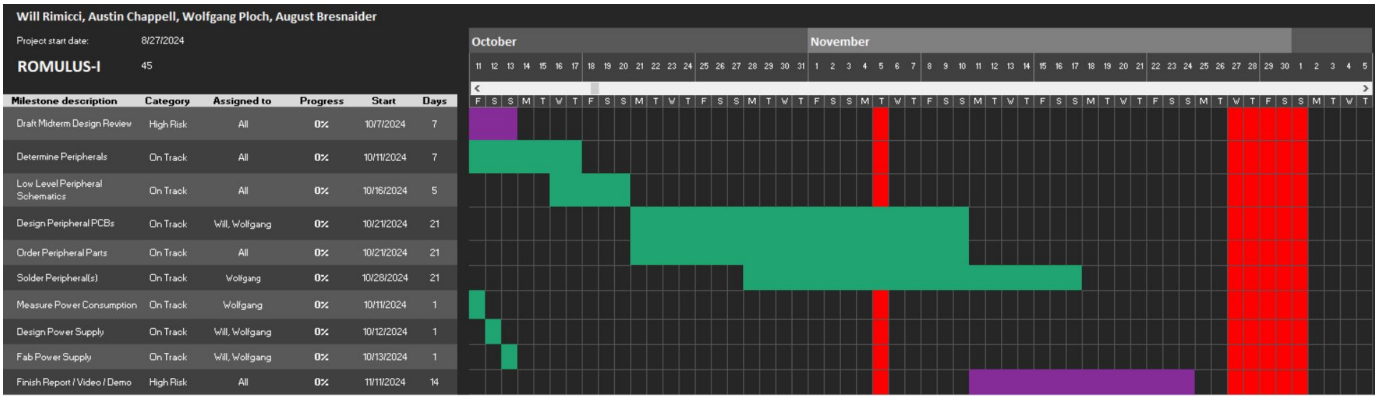


Fig. 41. The final section of our original GANTT chart

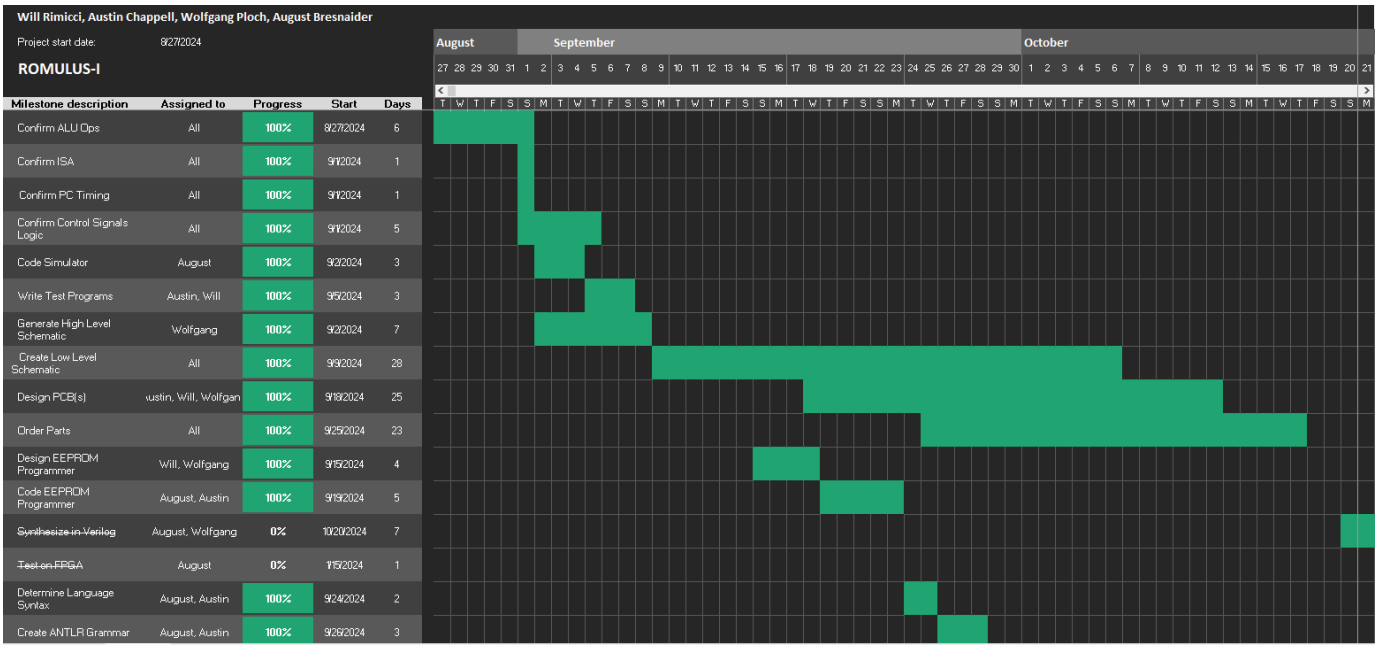


Fig. 42. The first section of our final GANTT chart

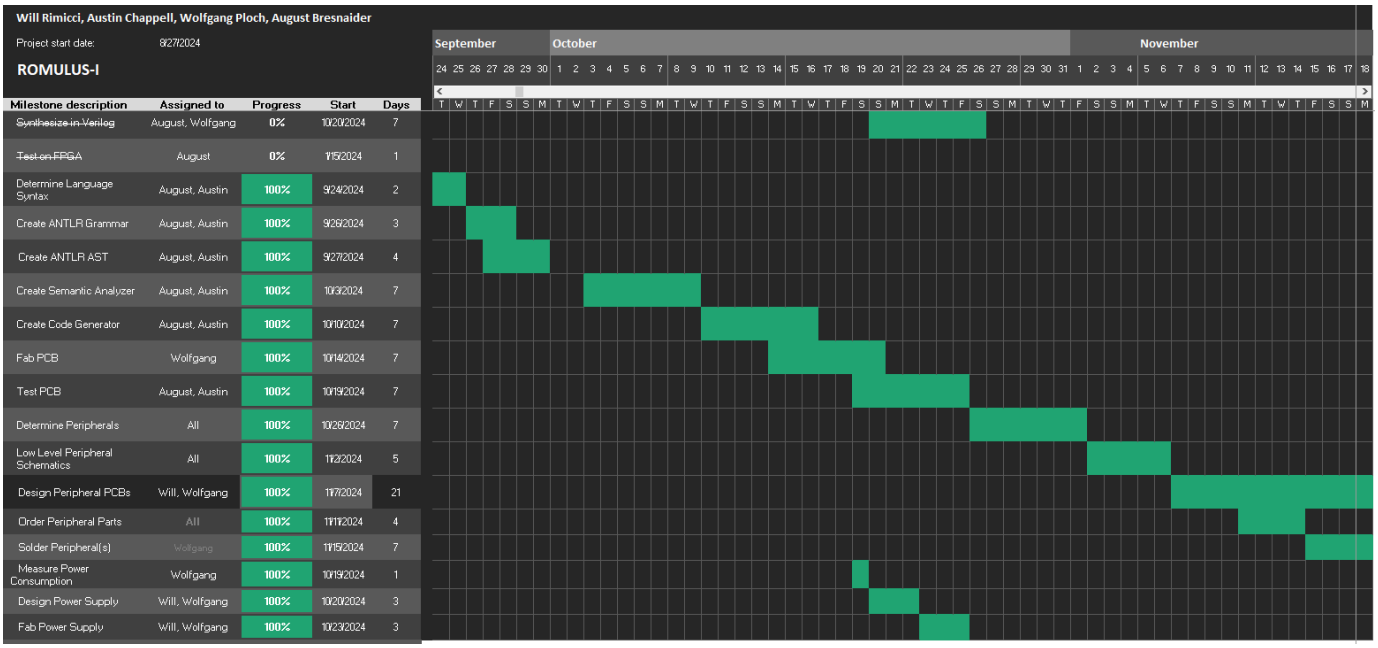


Fig. 43. The middle section of our final GANTT chart

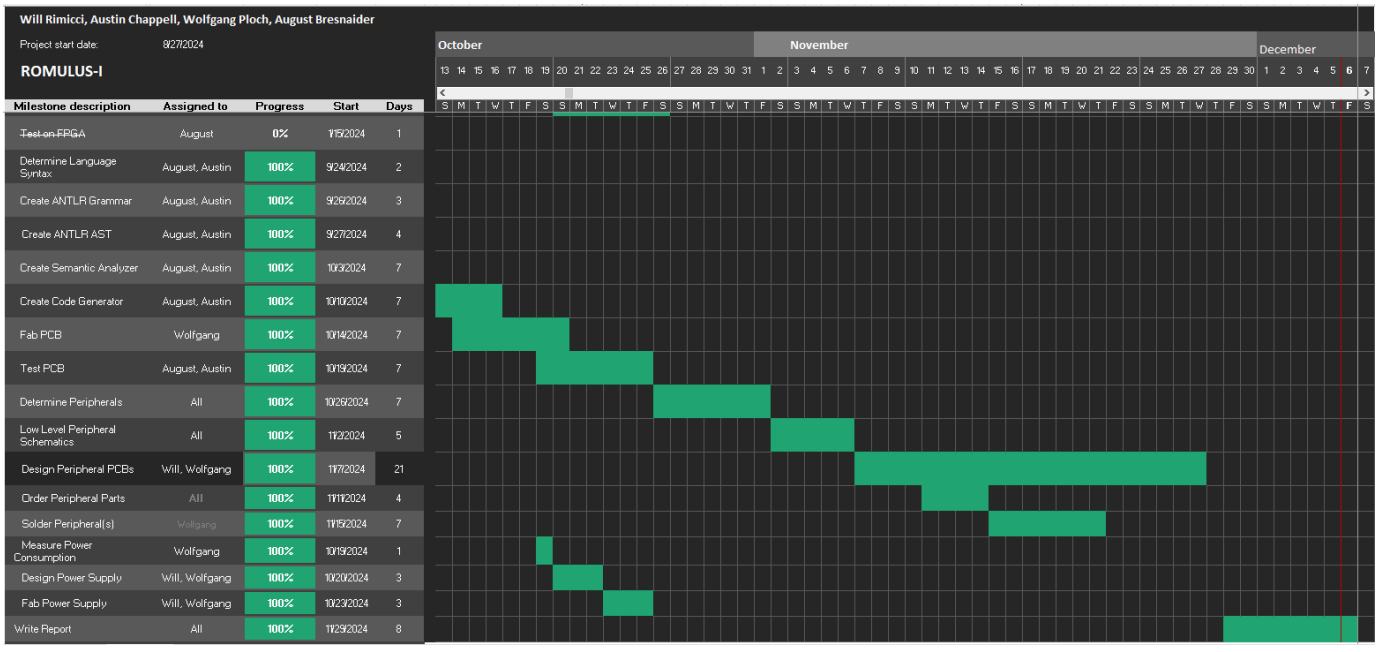


Fig. 44. The final section of our final GANTT chart