8-BIT COMPUTER

ECE251 Computer Architecture

Prof. Marano

Spring 2022

Nicholas Singh, Ahmad Malik

Abstract

The purpose of this project is to design either a virtual or hardware implementation of an 8-bit computer. The computer should be able to perform arithmetic operations such as adding or subtracting, be able to store and load data from memory, and have the ability to process R/I/J-type instructions. The computer should operate under a single clock and all data traffic should be passed through a single bus that connects the CPU to the memory, hence satisfying Von Neumann architecture. The computer should have an IO interface that can be used to supply instructions/data and also clearly display the computed results.

Design and Overview

We decided to build an 8-bit computer entirely out of hardware, using mostly the 74LS series ICs (and some from the CD4000 series). We used LEDs as outputs, and dip switches as inputs. Although it was not necessary, we demonstrated the outputs of each individual module using LEDS so that the current state of each part of the computer can be better understood; it also helps with debugging the computer to determine whether data is being sent or received correctly.

The computer has many forms of memory, the simplest being the registers that temporarily hold data in between operations. In total, our computer uses four 8-bit registers that each has the ability to access data from the main data bus. The five registers are: the A-Register (Accumulator), the B-Register, Memory Address Register, Instruction Register, and the Output Register. Each Register will be discussed later in depth. Our computer also has 16 bytes of RAM that is byte addressable by 4 bits (the 4-bit addresses are read from the Memory Address Register). The RAM can store 1-byte long instructions or 1-bye long data. The last form of memory is the 16K-Bit EEPROMs that we used to program the CPU control logic. We used programmable memory because it replaces the redundant combinational logic we would have needed to use to process the instructions from memory into micro instructions the computer can execute. Furthermore, programmable memory has the benefit of being programmable as the name indicates, so we can change instructions or add more instructions to our ISA.

The computer also has two counters: the Program Counter and the Micro-Instructions Counter. The Program Counter is a simple 4-bit binary counter that keeps track of which byte of memory to point to assuming the bytes in memory are instructions of a program. The Micro-Instructions Counter is also a 4-bit binary counter used to keep track of which step the computer is on regarding a certain instruction since instructions like “Load” and “Add” , though simple, require multiple minute steps for the computer to execute.

The ALU of our computer can only perform addition and subtraction of binary inputs. The inputs come from whatever is present in the A-Register and the B-Register, and the result is computed instantaneously for whatever inputs are present. The result can then be outputted to the bus when a signal is sent from the control logic.

Finally, the heart beat of this computer is the clock which drives all of the sequential logic in the computer necessary for computation. It oscillates between 0 and 5 volts, spending an equal amount of time high and low. The clock can be pulsed manually by push button or can run automatically through the use of a 555 timer.

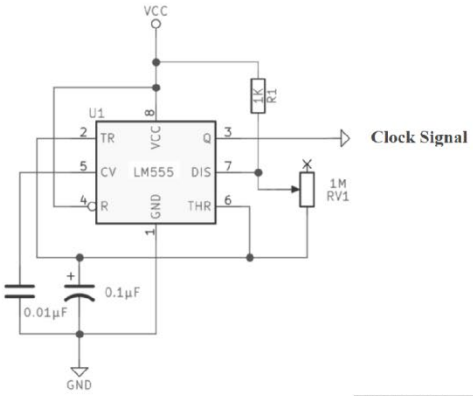
The Modules

In the following sections, we will discuss each of the briefly mentioned modules in detail.

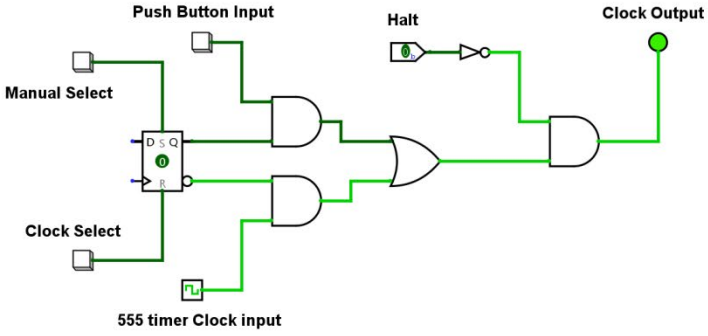
Clock

The clocks only function is to supply the computer with a consistent high/low oscillation. In our design, we used push buttons to select if the computer is being pulsed manually or by a 555timer of variable frequency. We needed a clock that would allow a manual input because it would allow us to step by step check if the instructions/micro-instructions are being sent/ received correctly. For the 555 timer (astable), we chose R1 and C1 values of 1kΩ and 0.1µF, and the R2 value was determined by a 1MΩ potentiometer, thus giving an output frequency that can vary between 7Hz to 4.8 kHz. We also added the ability to stop the clock given a "Halt" signal that will come out from the CPU control logic; it will stop the computer once the Halt instruction is received. Below is the schematic of 555 timer, the logic diagram that selects which mode the clock runs on, a block diagram for the clock, and an image of the actual hardware.

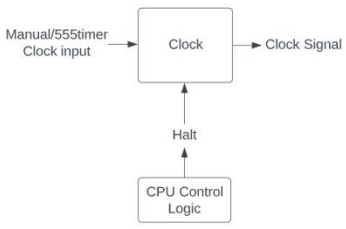
Variable 555 timer

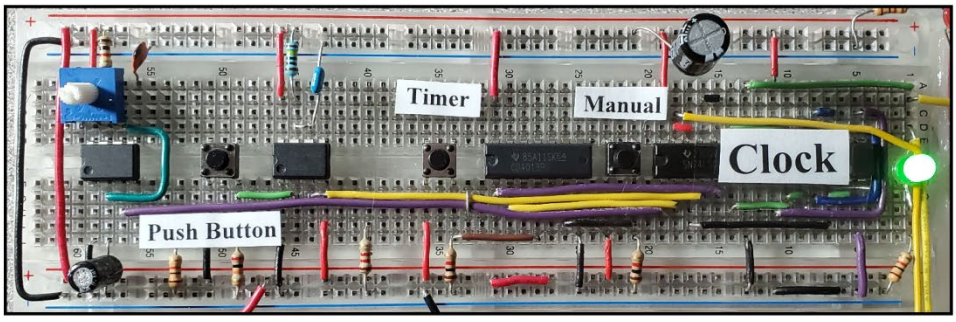


Clock Mode Select



Block Diagram





Registers

All of the registers used in this project are almost exactly identical to each other, with the exception of the Memory Address Register. They all have the ability to load and store 8 bits of data present on the bus given they receive a positive clock edge and if their "load" signals are set to LOW (the 74LS173 load signals are active LOW). The 74LS173 is a 4-bit D-type register, so we will need to attach two of them in series to create an 8 bit register for each register module.

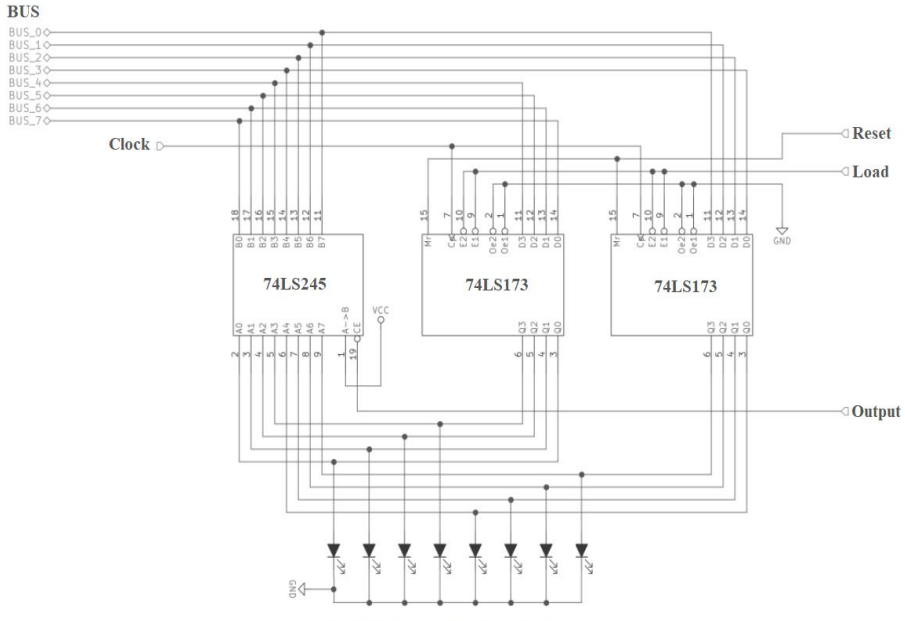
Once a register loads data from the bus, it is immediately stored in its memory and left floating on its output pins. Outputting data from the registers on to the bus is essential, but since the registers are always outputting data onto the bus, we had to use some kind of tri-state logic that can control when the registers are outputting on to bus given an "output" signal, otherwise the registers would interfere with important data on that may be on the bus. We used the 74LS245, Octal Bus Transceivers, to accomplish this and the resulting "output" signal that controls the outputting on to the bus were also active LOW.

The differences between the registers are minute. The A-Register can load and output all 8-bits of content to and from the bus given its load and output signals are set appropriately. The B-register on the other hand is identical, however, we don't need to output its contents on to the bus. This is because the B-register is used primarily for storing data that will be used for arithmetic operations, so its outputs go straight to the ALU. The Output Register is the same in that it loads 8 bits of data from the bus but does not output it back on to the bus, instead it goes directly to eight LED's that can be used to represent the output results of a program. The Instruction Register also loads all 8-bits from the bus, but when the output signal is called, it sends its 4 LSB bits to the bus and the 4 MSB bits to the CPU Control Logic.

Finally, the Memory Address Register is unlike the other registers in that it only loads the 4 LSB bits from the bus (so it uses a single 74LS173). We attached the outputs of this register straight to the RAM memory address inputs so the moment an address is loaded into the register, the RAM retrieves the data at that memory location.

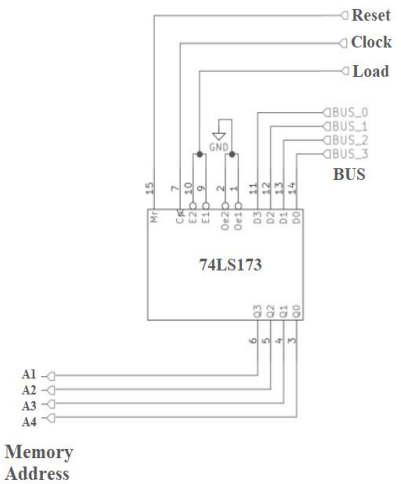
Below is the schematic for the general purpose 8-bit register we are using and 4 bit register used for memory.

General Purpose 8-Bit Register

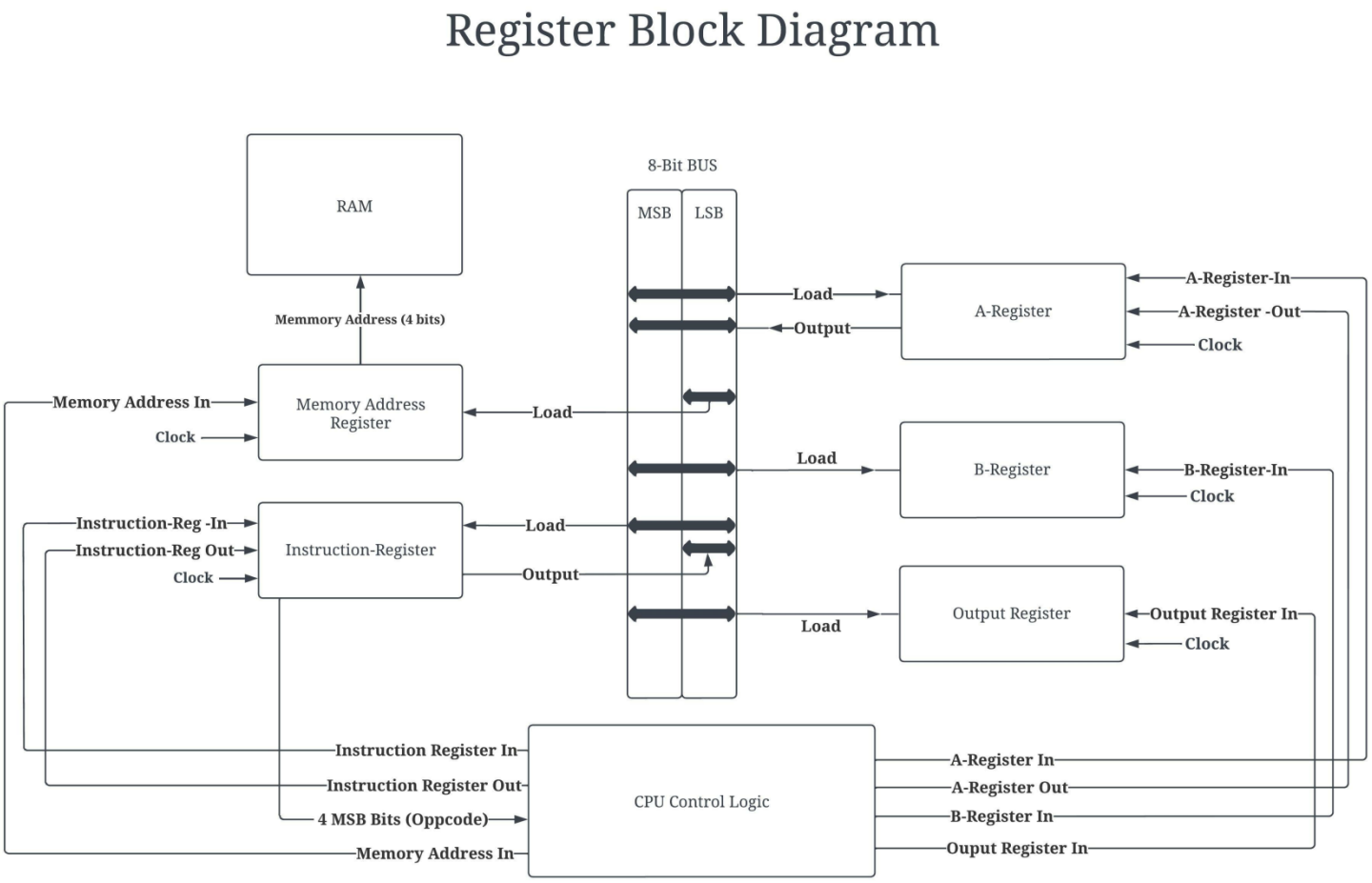


LED Indicators to see Data in Register

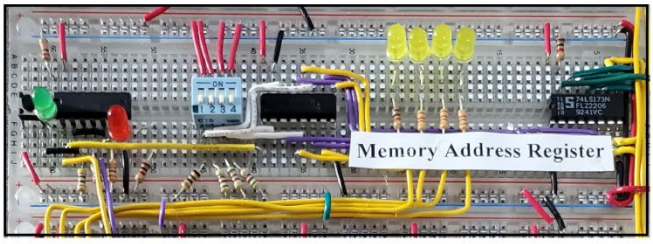
4-Bit Memory Address Register

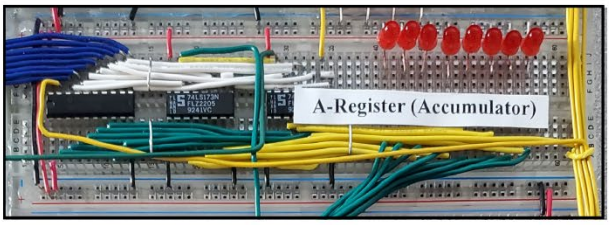


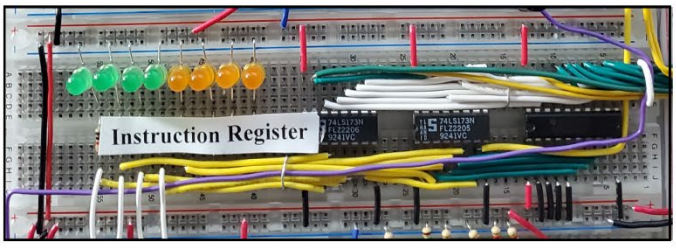
Below is a block diagram illustrating the data path of these five registers. The “load” and “output” signals are transmitted from the CPU Control Logic. To differentiate the different loads and output signals, the signals have been named accordingly.

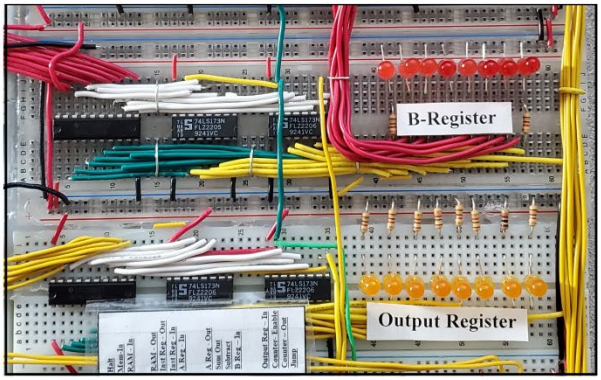


Below is the design implemented on breadboards.







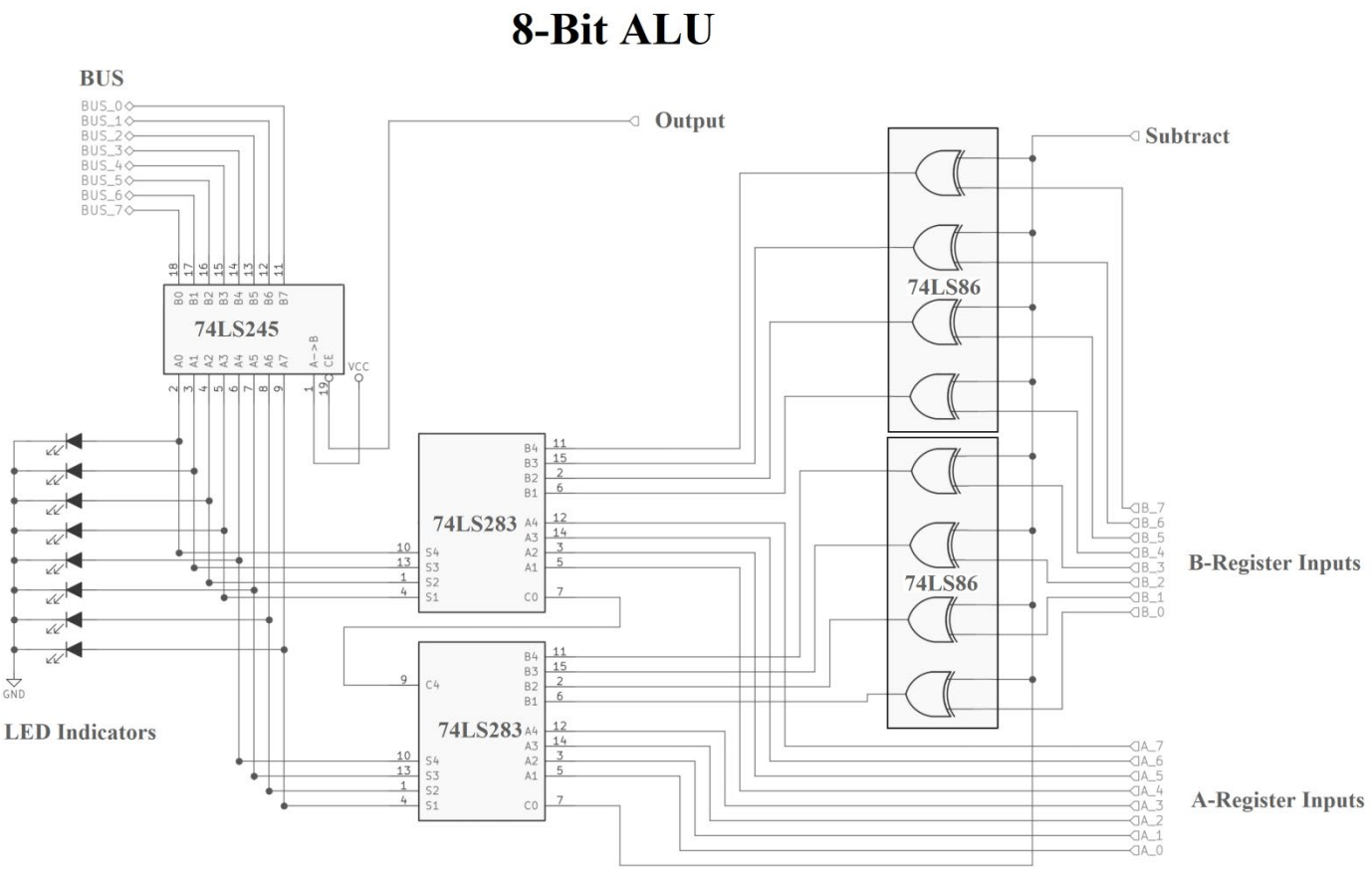


ALU

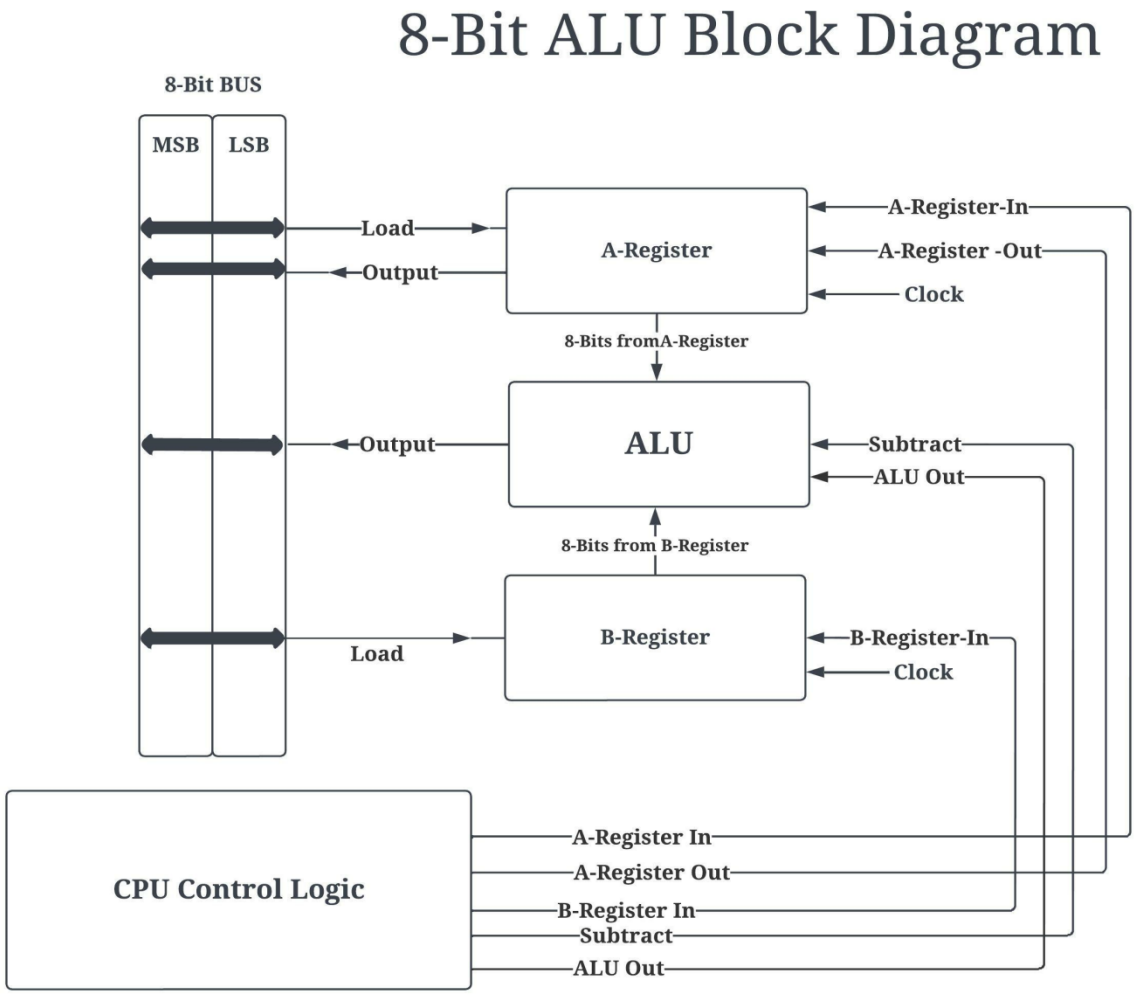
The Arithmetic Logic Unit of our computer will add and subtract 8-bit binary values. It will receive its inputs from the contents of the A-Register and from the contents of the B register. To add the two inputs from each register, we used a 74LS283 which is a 4bit binary adder with a carry in and carry out. Placing two of them in series such that the “carry out” of the first adder is and routed into the “carry in” of the second will allow us to add two binary numbers together. The outputs of the 8-bit adder are then sent to a 74LS245, Octal Bus Transceivers, which outputs the contents of the ALU on to the bus when its enable is set to Active low; this will be the output signal.

In order to subtract the A and B registers, we need to use 2’s complement. This would require inverting the bits of the B-Register and adding 1, then adding this value to the A-register value to get the results of the operation. We wanted the ALU to subtract and add the values depending on whether a single “subtract signal” was set high or low. To do this, we used a 74LS86 XOR chip that can invert the bits of the B register given the subtract signal is HIGH, otherwise when the subtract signal is LOW, the bits would flow through the XOR unaffected, thus performing normal binary addition. Inverting the B-Register bits is only 1’s complement. 2’s complement requires 1 to be added to those inverted bits as well. This can easily be done by wiring the subtract signal to the “carry in” of the first adder.

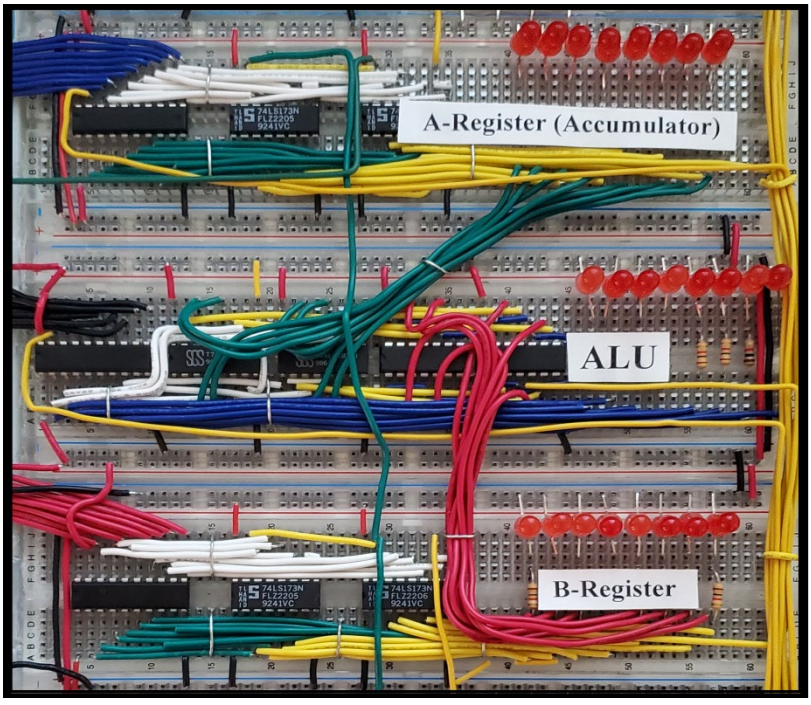
Thus we can subtract and add bits from the A/B registers simply by setting the “subtract” signal to HIGH or LOW.



Below is the Bock Diagram of the ALU



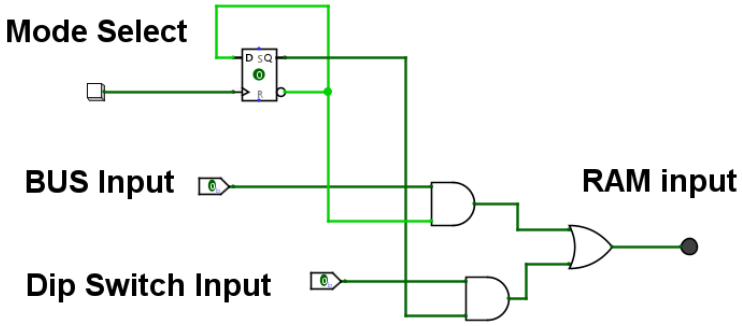
Below is the Hardware Implementation of the ALU



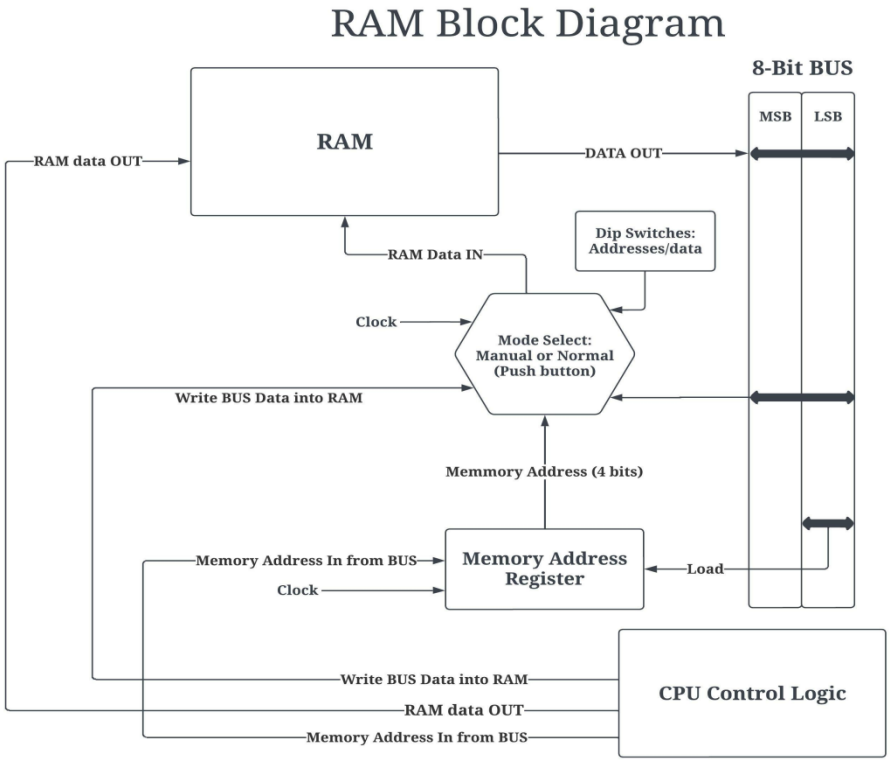
RAM

In this computer, we are using two 74LS189s to store all of the instructions and data. Each chip can store 64 bits of data and requires a 4 bit input that would represent the 16 possible addresses. Placing two of these RAM chips in parallel will give us 16 possible addresses, where each address can store a byte of data or instruction.

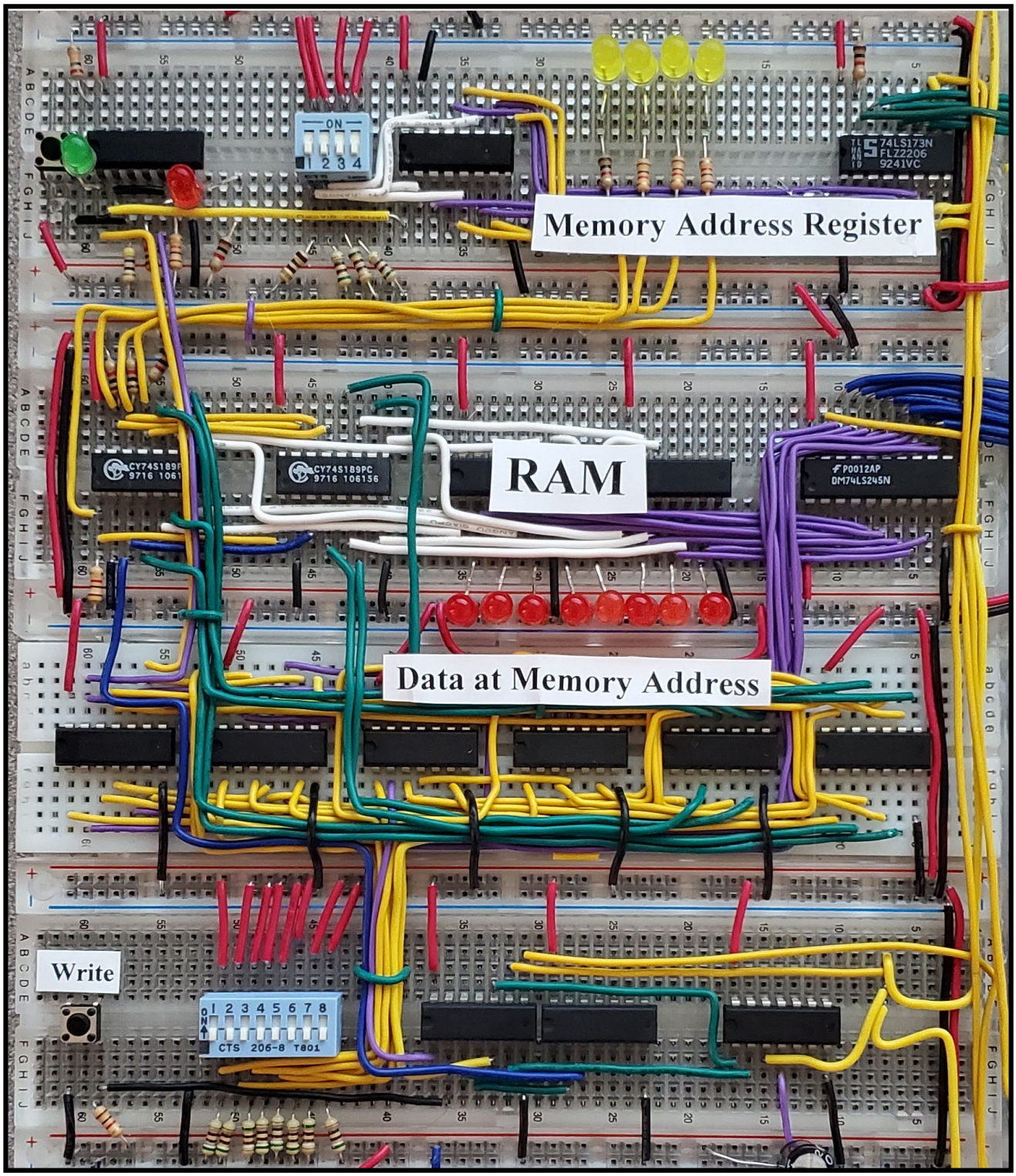
We want to be able to select between two modes of the memory, manual mode and normal mode. Under manual mode, we want to be able to program the RAM using dip switches. This means that we need a 4bit dip switch for the RAM addresses and an 8bit dip switch to program the data at those addresses. Under normal operation, the input to the RAM addresses would come from the Memory Address Register which receives data from the bus, and the data being stored Into the RAM would also come from the bus. Thus, we needed a way to select between these two operation modes, so we designed a simple circuit that used a D-flip-flop, a push button, a bunch of redundant logic to select between the modes. The logic circuit on the right was repeated eight times to compensate for inputs that might come either from the Bus or from the dip switches depending on the selected mode. The same circuit was implemented for the RAM address inputs, allowing inputs either from the dip switches or from the Memory Address Register.



To write data into the RAM, the chips need a signal either from the CPU control logic (during normal mode) or from a pushbutton (during manual mode). During normal mode, we only want to write data into RAM alongside the clock signal, thus we AND'ed the CPU control logic signal for "write data into ram" with the clock signal. Below shows the block diagram that characterizes the ram module.

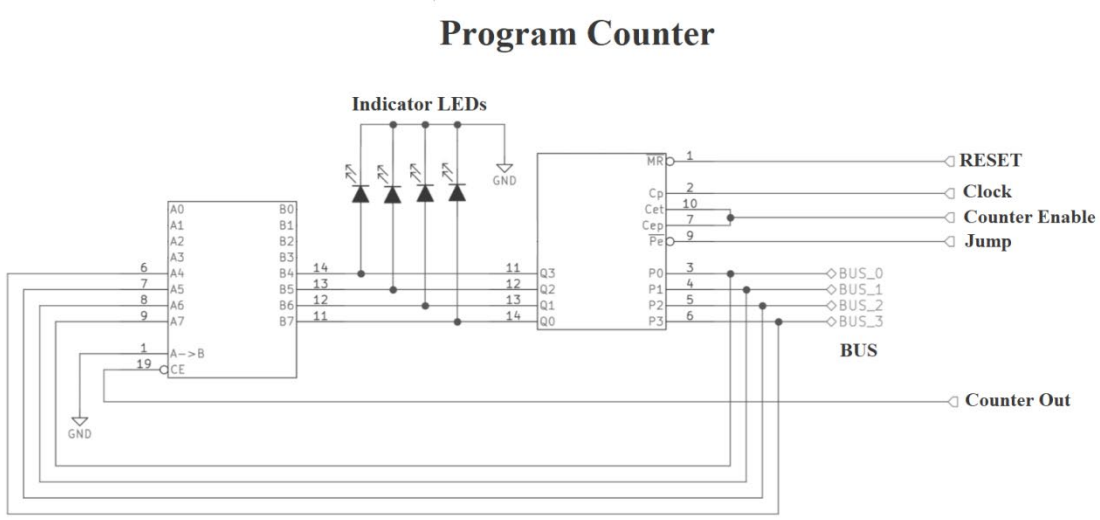


Below is the Hardware implementation of the RAM. On the top left, you can see a green and red LED which signifies the normal and manual mode, respectively. The 4bit dip switch can change the memory address under manual mode and the 8bit dip switch can write data into those addresses

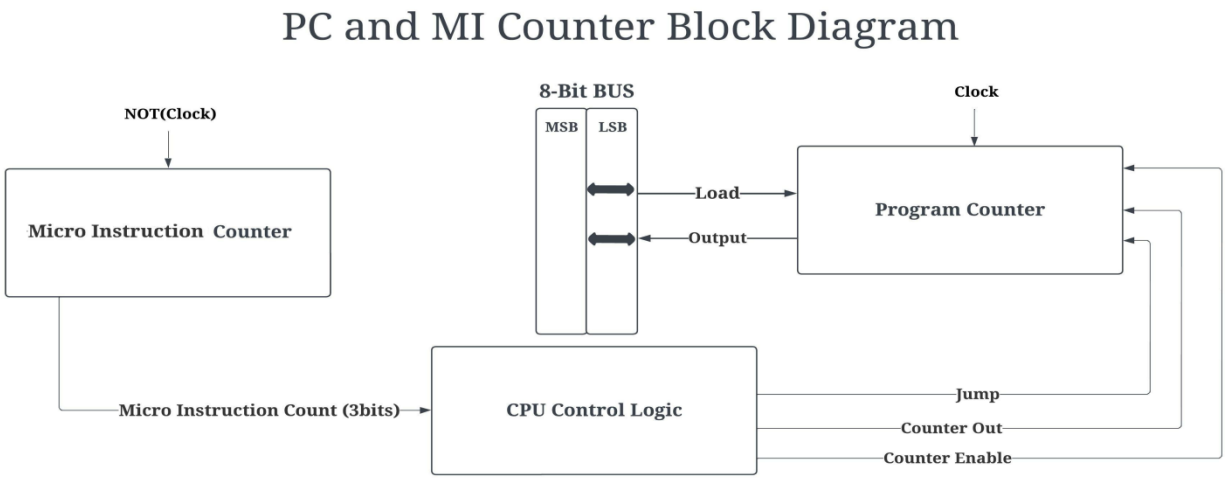


Counters

In this computer, there are two counters: the Program Counter and the Micro Instructions Counter. The program counter we are implementing is a simple binary counter that changes states every clock cycle. We are using the 74LS163, a 4 bit counter that also has a load function that will set the counter to the loaded bits and count from there. This is necessary because it will allow our computer to use jump instructions. In our computer the Jump instruction will load the 4LSB of the bus into the counter. The chip can also be enabled and disabled by input signal which will freeze the program counter so it doesn't keep counting every clock cycle. To output the counters data on to the bus, we are again going to use the 74LS245 Octal bus transceiver, thus giving us a third input that we can use to control when the counter will output onto the bus. Below is the Program Counter's schematic.



The Micro Instruction Counter is very similar to the Program Counter in that it uses the same 74LS163 chip. However, it operates on the negative edge of the clock ($\overline{\text{clock}}$). This is important because we want to load the microinstructions into the CPU control logic before the computer tries to execute it on the positive edge of the clock. The output of the Micro Instructions Counter will be a simply be a binary output of 3bits that feeds straight into the control logic. The reason why we only need 3bits of the 4 the counter outputs is because micro instructions for the typical instruction set are not very lengthy typically around only 5 steps, so 4bits (16 micro instructions) is overkill. We tie the 4th bit output to its own reset so the counter resets after Step 7. See the block diagram below which describe the two counters.



The ISA and CPU Control Logic

The only two inputs that are going into the Control Logic are the 3bits from the Micro Instruction Counter and the 4 MSB of the Instruction Register. Appending the 3bits to the end of the 4MSB will be the machine code that the Control Logic will understand. The 4 MSB will represent the Opcode or the instructions. The 3LSB of the machine code (7bits) will represent the micro instructions. This machine code is what the CPU Control Logic is going to have to interpret and use to run programs. Refer to the Table below which outlines the Opcode and the different types of instructions the Control Logic will understand.

Instruction	Type	Opcode	Format	Description
NOP	N/A	0000	0000,xxxx	No Operation
LOADA	R	0001	0001,MemAdd	Load from memory and store in the A - Register
STRA	R	0010	0010,MemAdd	Takes what's in A- Register and stores it in memory
ADD	R	0011	0011,MemAdd	Adds the value at a particular memory address to A-Register
SUB	R	0100	0100,MemAdd	Subtracts the value at a particular memory address to whatever is in A-register
LOADI	I	0101	0101,data	Loads the data value to A-Register
ADDI	I	0110	0110,data	Adds that data value to A- Register
SUBI	I	0111	0111,data	Subtracts the data value
JMP	J	1000	1000,data	Jump the Program Counter to data
OUTA	R	1001	1010,xxxx	Output Whatever is in Register A to the Output Register
HALT	R	1111	1111,xxxx	Stop the Computer (stop the clock)

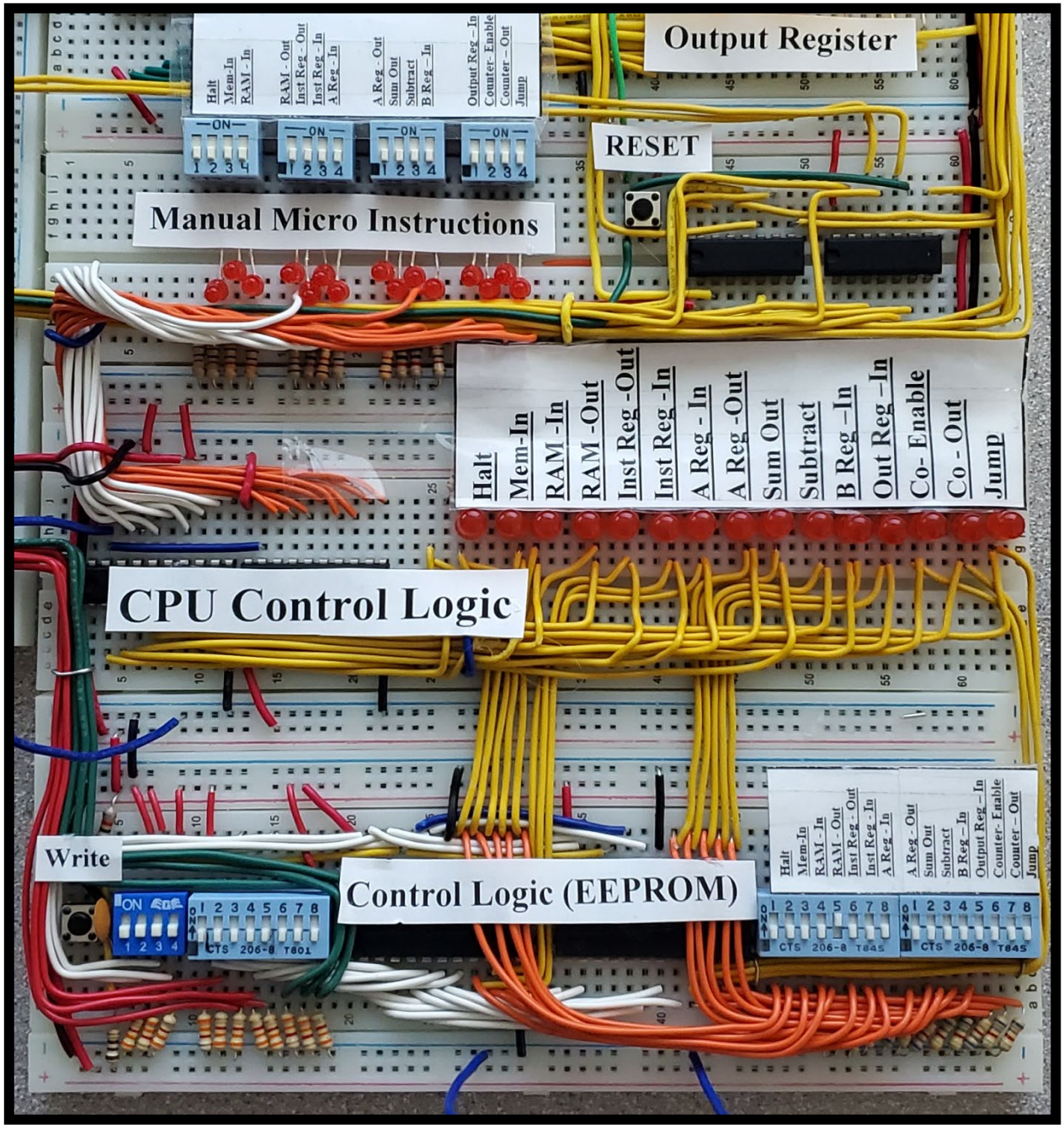
Machine Code Truth Table

The table demonstrates the Micro Instructions that the CPU will execute given a binary Machine code made up by the Opcode and the Micro Instruction Step Counter.

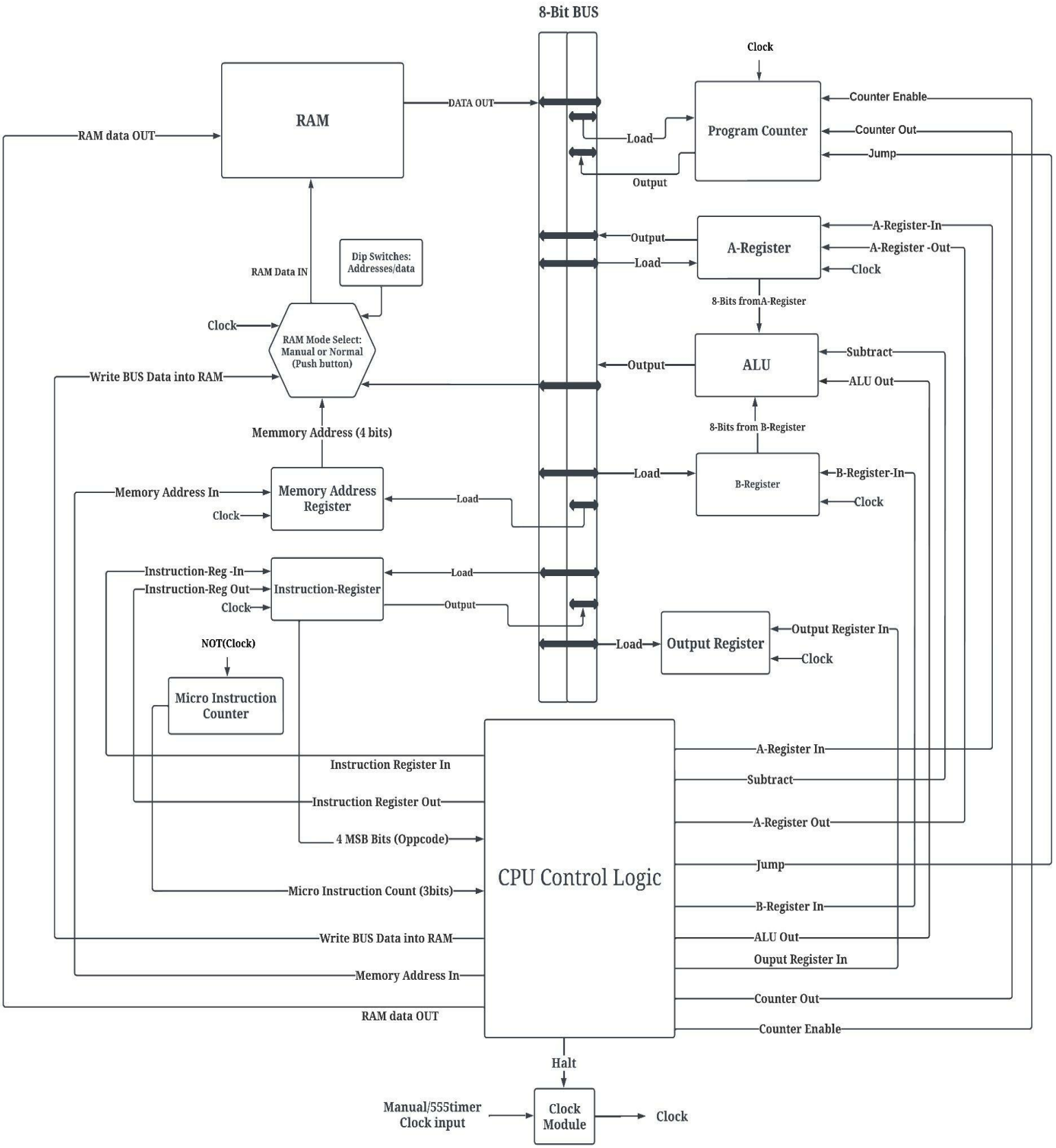
Instruction	Opcode	MI Step	Halt	Mem-In	RAM-In	RAM-Out	I-Reg-Out	I-Reg-In	A-Reg-In	A-Reg-Out	ALU-Out	Sub	B-Reg-In	Out-Reg-In	Co-Enbl	Co-Out	Jmp
NOP	**0000**	**000**	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
NOP	0000	001	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0
NOP	0000	010	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
NOP	0000	011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
NOP	0000	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
NOP	0000	101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LOADA	**0001**	**000**	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
LOADA	0001	001	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0
LOADA	0001	010	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
LOADA	0001	011	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0
LOADA	0001	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LOADA	0001	101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
STRA	**0010**	**000**	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
STRA	0010	001	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0
STRA	0010	010	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
STRA	0010	011	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0
STRA	0010	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
STRA	0010	101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ADD	**0011**	**000**	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
ADD	0011	001	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0
ADD	0011	010	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
ADD	0011	011	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0
ADD	0011	100	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0
ADD	0011	101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SUB	**0100**	**000**	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
SUB	0100	001	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0
SUB	0100	010	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
SUB	0100	011	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0
SUB	0100	100	0	0	0	0	0	0	1	0	1	1	0	0	0	0	0
SUB	0100	101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LOADI	**0101**	**000**	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
LOADI	0101	001	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0
LOADI	0101	010	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
LOADI	0101	011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LOADI	0101	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LOADI	0101	101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ADDI	**0110**	**000**	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
ADDI	0110	001	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0
ADDI	0110	010	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0
ADDI	0110	011	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0
ADDI	0110	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ADDI	0110	101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SUBI	**0111**	**000**	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
SUBI	0111	001	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0
SUBI	0111	010	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0
SUBI	0111	011	0	0	0	0	0	0	1	0	1	1	0	0	0	0	0
SUBI	0111	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SUBI	0111	101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
JMP	**1000**	**000**	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
JMP	1000	001	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0
JMP	1000	010	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
JMP	1000	011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
JMP	1000	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
JMP	1000	101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
OUTA	**1001**	**000**	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
OUTA	1001	001	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0
OUTA	1001	010	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0
OUTA	1001	011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
OUTA	1001	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
OUTA	1001	101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
HALT	**1111**	**000**	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
HALT	1111	001	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0
HALT	1111	010	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
HALT	1111	011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
HALT	1111	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
HALT	1111	101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Programming the Control Logic

To store the Truth Table from the previous page, we needed some kind of programmable memory module where the Machine Code would act as the address of the memory module, and the data bit located in the memory would represent the Micro Instructions. We accomplished by using two 28C16A-25 EEPROMs. They have 11 bit long addresses and can store a byte in each address. Tying two of them in parallel would give use 16 bits of data storage per address, which is enough to span the length of our micro-instruction set. Below is our hardware implementation of this. The dip switches are used to program the EEPROM. On the top, you can see 4 sets of 4bit dip switches that can manually input micro instruction. This was essentially for debugging the comput



The Complete Data Path



Finished Hardware Implementation