Core i382

The User Manual
Dec 2022

Created and Edited by: Ayushi Gupta, Annabelle Crescenzo

Pledge: I pledge my honor that I have abided by the Stevens Honor System.

**Extra Credit Attempted:

- LED Display
- Assembler generates two image files

1 Job Descriptions

Ayushi Gupta: I designed the CPU in Logisim for our machine code that performs the LDR, ADD, and SUB instructions. Our assembly code uses 4 registers and 2 RAMs to load values from memory and perform mathematical calculations with addition and subtraction. I also worked on adding additional LED elements to our CPU for improved comprehensibility of the data flow.

Annabelle Crescenzo: I completed the assembler in python that translates LDR, ADD, and SUB instructions into machine code and writes their corresponding hexadecimals into an image file. I also worked on the extra credit for the assembler to distinguish between .text and .data segments to create two image files. The format of the image files is specific to loading into the memory of a Logisim circ file. We discussed the details of the machine code together, such as number of bits and instruction format.

2 Assembler Program

Set Up Instruction File: To use the assembler, download the assembly.py file attached in this user manual. Create a text file named "assembly.txt" in the same location as the python script on your computer. This text file holds the instructions and data that you want to run in the CPU. Be sure to distinguish between instructions and data by using the ".text" and ".data" titles. For organizational purposes and efficiency, it is recommended to put .text before .data. Make sure each of your instructions are typed on their own line in order of execution. These instructions must be written in the format specified in section 4 of this document. The data must be integers each written on its own line in order of how you want them to be loaded into memory. The starting address for the instruction memory and data memory is 00.

An example of an assembly.txt file is shown below:

```
.text
LDR X1 X1
LDR X2 X1
ADD X3 X2 X1
SUB X0 X3 X1
ADD X1 X0 X2
SUB X0 X1 X0
SUB X2 X3 X0
ADD X1 X2 X2

.data
2
3
5
7
```

Run Assembler: To run the python script in terminal, navigate to the location of the python file and assembly.txt file and use the command:

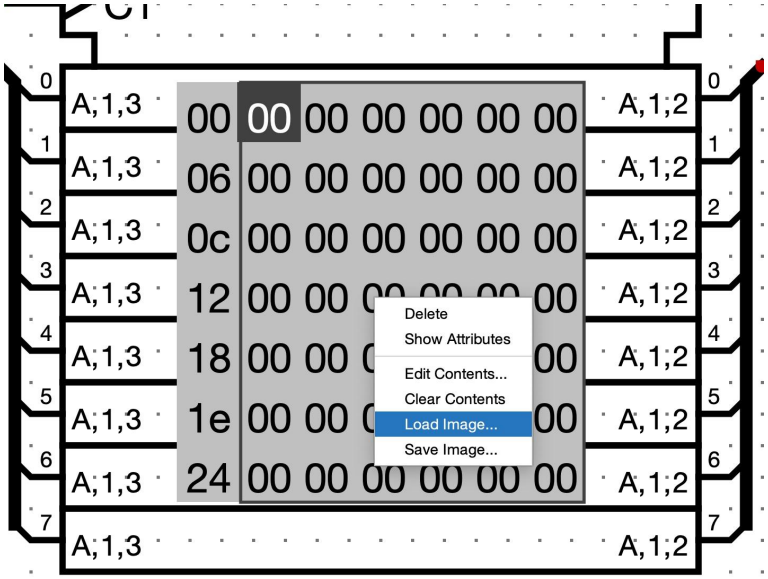
username@ubuntu python3 assembler.py

After the assembler is run, two new text files, called “instructionImage.txt” and “dataImage.txt” , are created. Both files are in the same location as the assembler python script and assembly.txt instruction file. The instructionImage.txt is the image file that contains the hexadecimal values for the instructions. The dataImage.txt is the image file that contains the hexadecimal values for the data. Both image files can be directly loaded into the instruction and data memory of the CPU in Logisim.

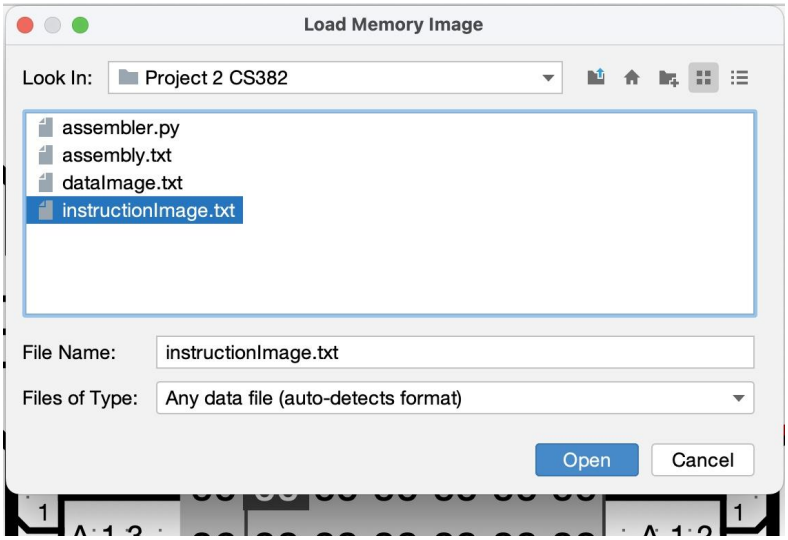
To open an image file through terminal to view its contents, make sure you are in the location of the file and use the command:

username@ubuntu open <filename.txt>

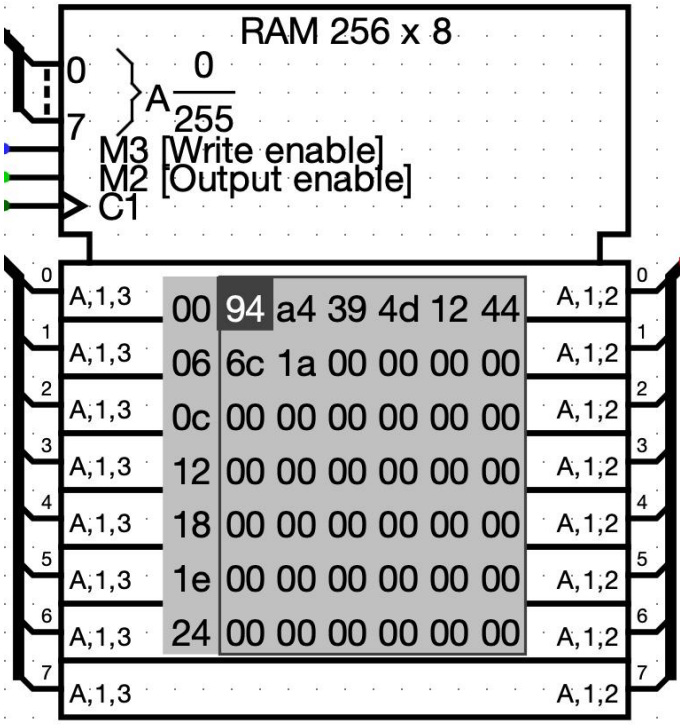
Load Image File: To load an image file into the instruction memory in Logisim, right click on the RAM architecture and select “Load Image”.



Then select the image file from its location on your computer. The data type is specified in the image file, so the “Files of Type:” can remain at auto detect format.



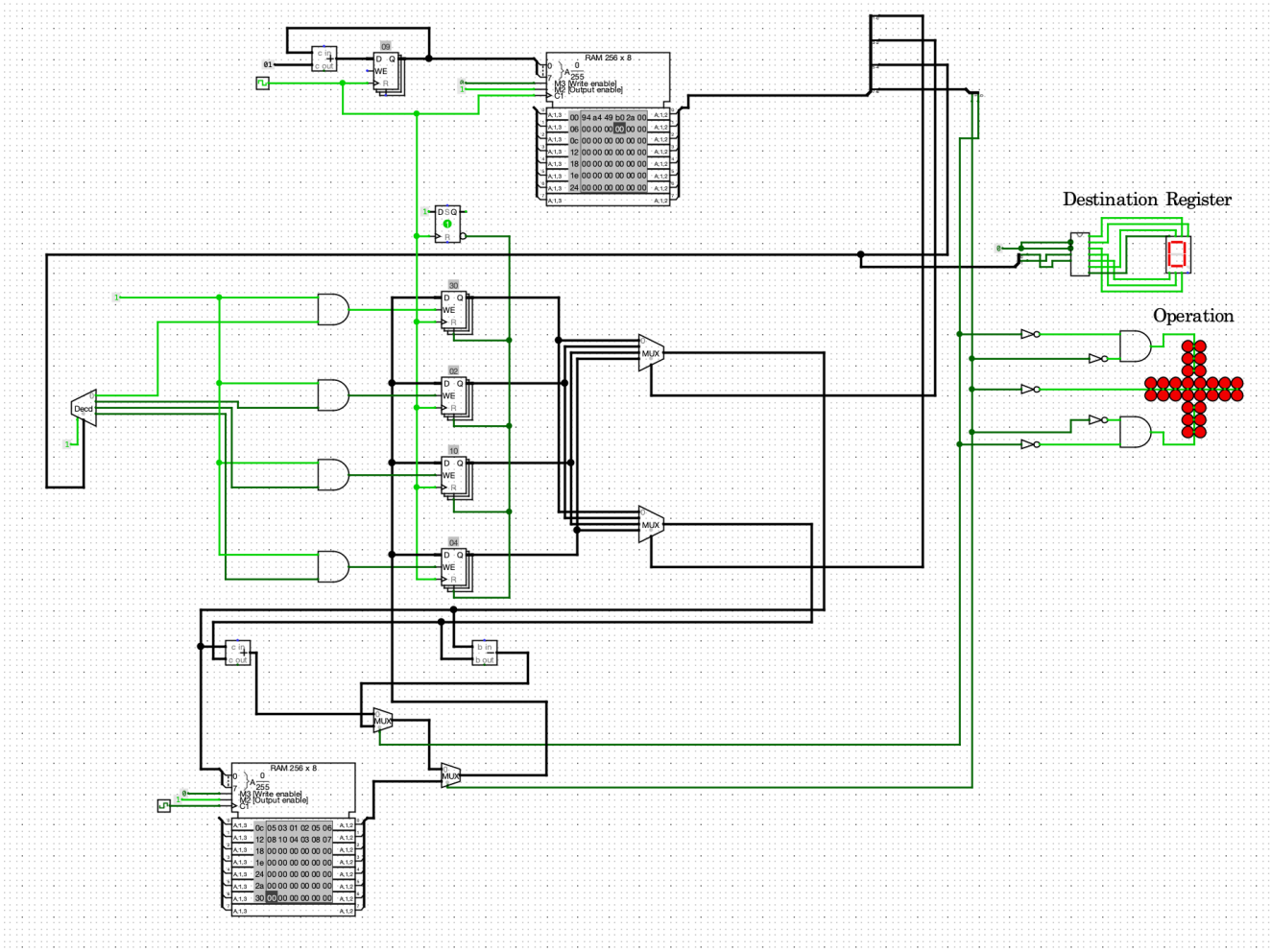
After loading the image file, the hexadecimal values appear in the instruction or data memory of the CPU starting at address 00.



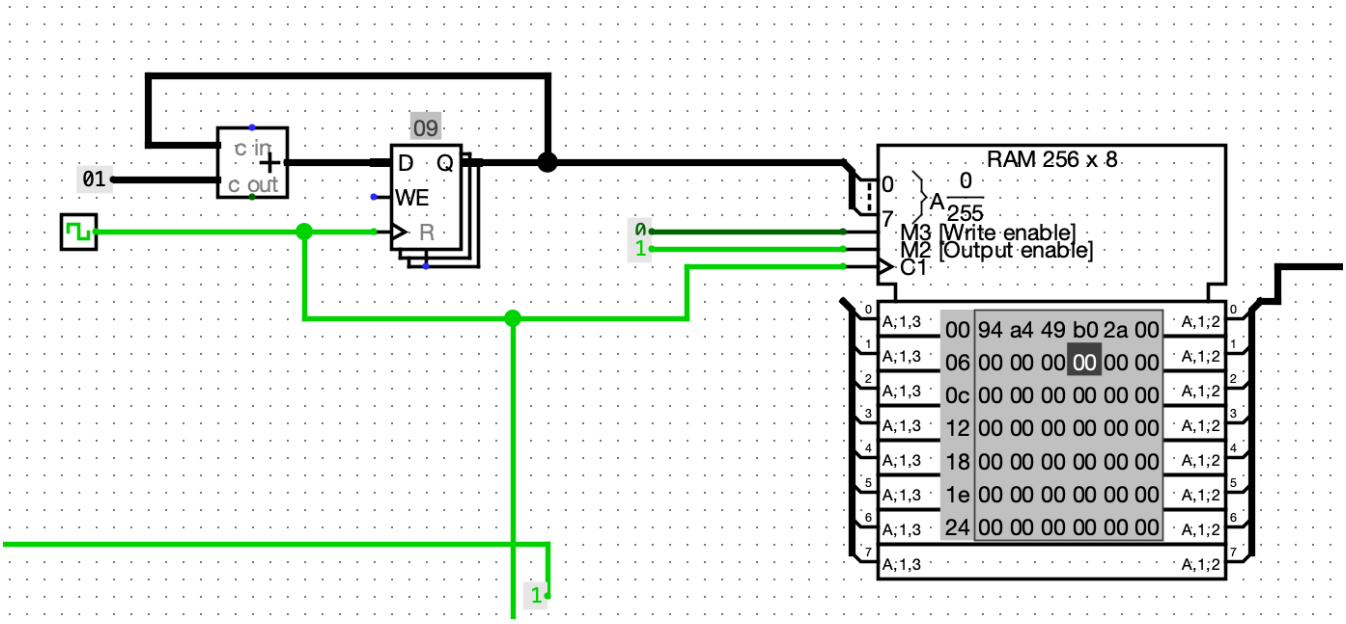
Enjoy: The CPU is now ready to run your instructions and data!

3 Architecture Description

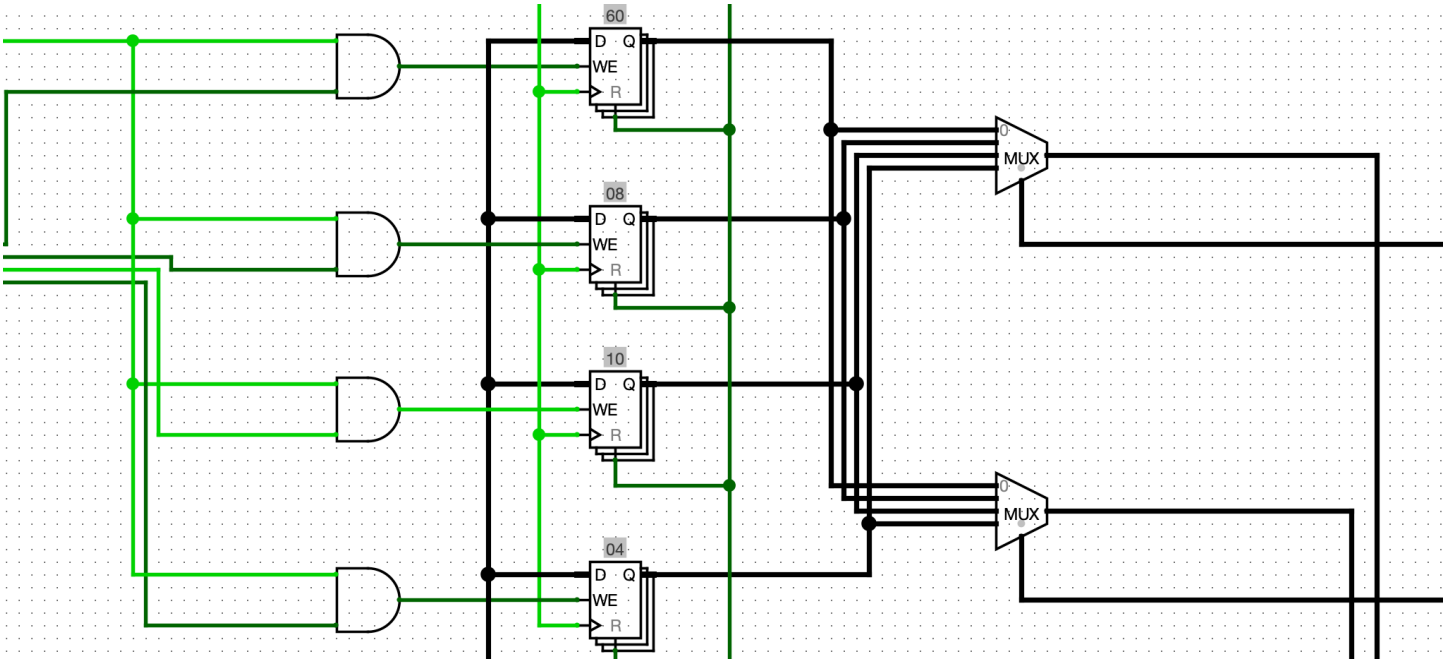
Overview:



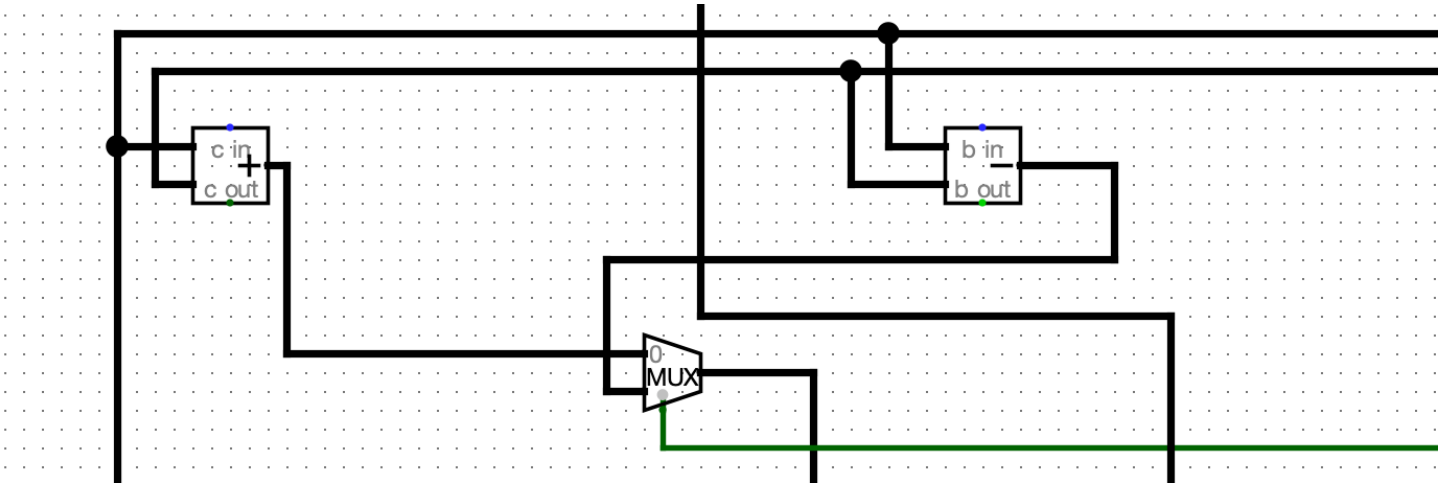
Instruction Memory: This part of the CPU loads each instruction in the RAM one address at a time. The PC is incremented by 1 every 2 clock cycles. This means that every 4 ticks of the clock is one clock cycle.



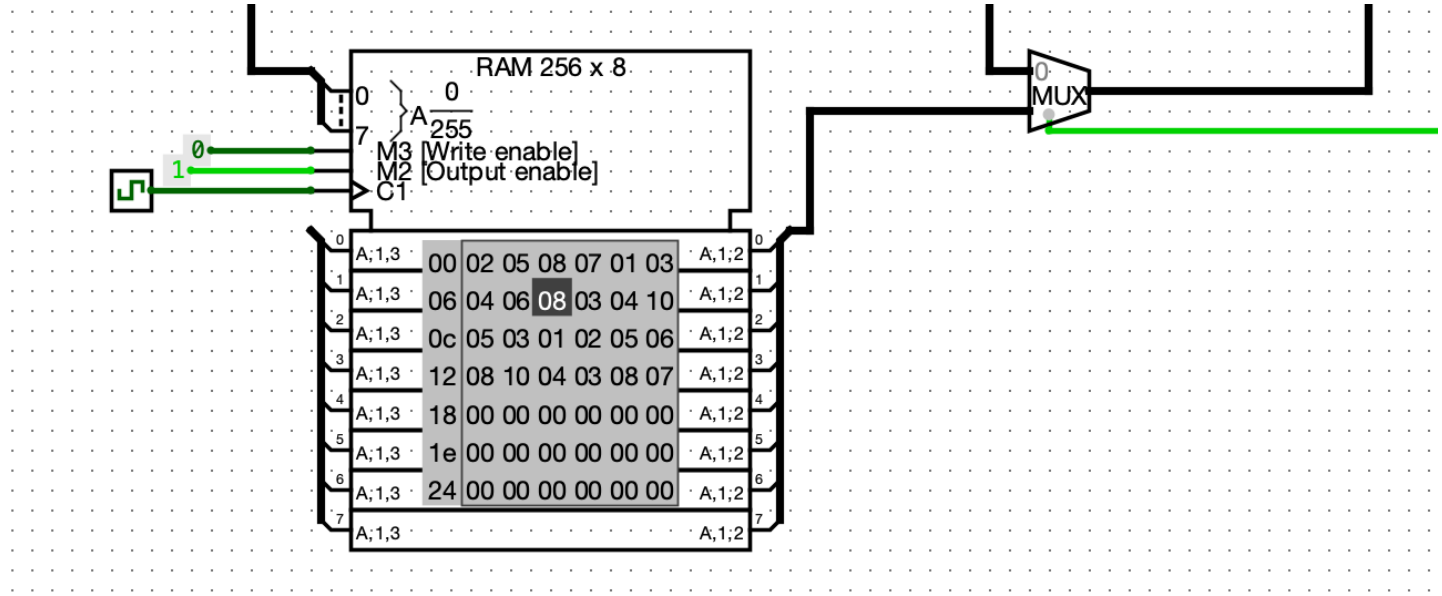
Register File: Our register file consists of 4 general-purpose registers that can store values.



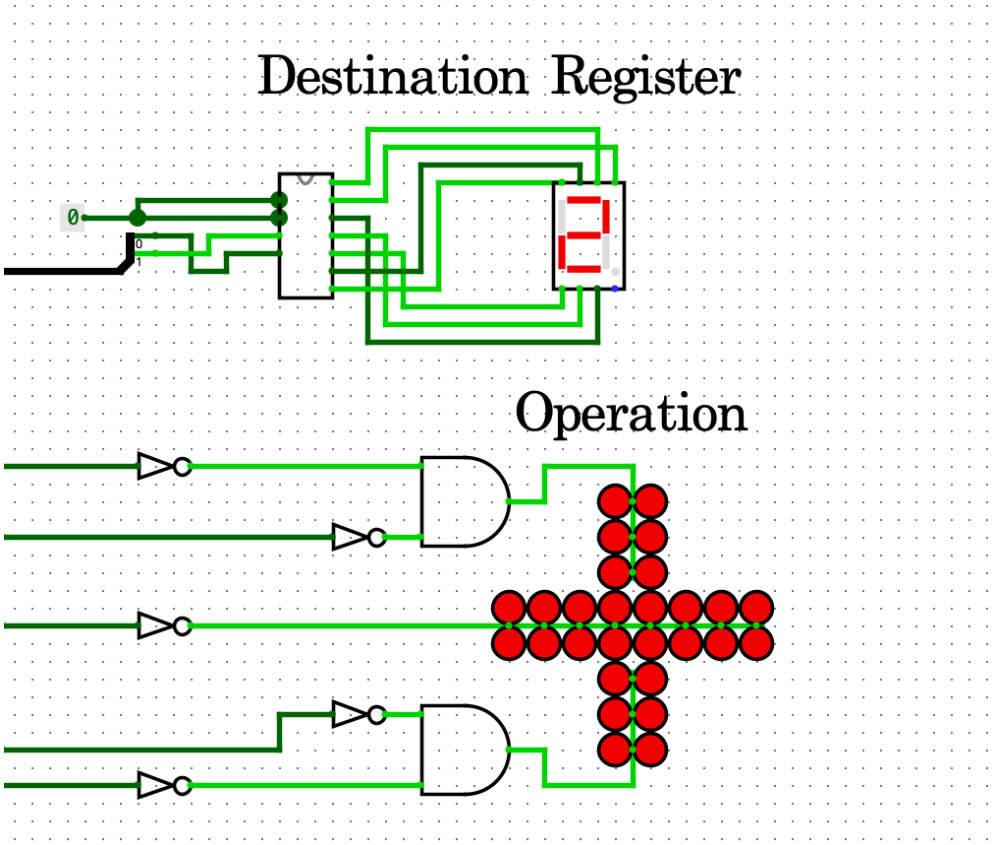
ALU: Our assembly language has capabilities of addition and subtraction which is why we have an adder and a subtractor that takes in the data from two registers. The multiplexor chooses between the sum and the difference based on what action is specified in the instruction.



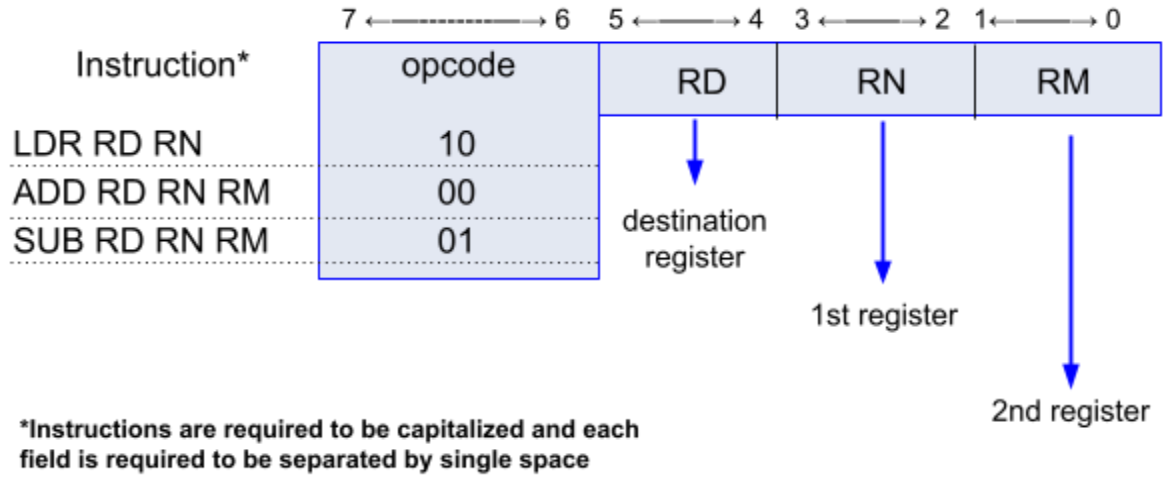
Data Memory: Any data being loaded from memory comes from this RAM. It is being controlled by its own clock that runs at 2 ticks per clock cycle. The multiplexor takes in both the value from data memory or the value from the ALU and will return either based on the control signal. The data is then written back to the register file.



LED Displays: The 7 segment LED display shows the register number that the value is being written to for the user to be able to follow the flow of data throughout the CPU. Under that, the operation that is being performed is also demonstrated with LEDs. For the LDR instruction, the lights are turned off. For ADD and SUB, the corresponding mathematical symbol is depicted.



4 Instruction Implementation



Our instruction design is similar to the standard 32 bit ARM instruction; however, we only use four registers, so our instruction is implemented with 8 bits. Since we only have four registers, we only need 2 bits to distinguish between each one (00, 01, 10, 11). This means that each field of the instruction (opcode, destination register, 1st register, 2nd register) is 2 bits. There are 4 fields of the instruction, so 4 fields * 2 bits equals a total of 8 bits. There is one instruction that only has three fields; LDR has just opcode, destination register, and 1st register. In this instance, the last field is assumed to be 00, which is 2 bits in length, making the total machine code for LDR 8 bits.