Program Write-up

Jennifer Kulich

December 10 2019

1 Part 1

This program is to create an emulator that can store and print memory from an array. The user can either store memory in the list, print a range of addresses, print one address, or 'run' th program. This is all done through the terminal.

1.1 How To Run

This should be run from the command line and requires IntelHex. " It is recommended to use the included Pipfile and Pipfile.lock with pipenv install (which can be installed with pip install pipenv). This will setup the virtualenv for you, which can be invoked with pipenv shell," (Matthew Krohn- because I couldn't word it well enough). Then, you can type 'python3 t34.py' in the command line for the program to run and input to be taken in. If you wish to run a file, you can add the file name to the command line after t34.py.

1.2 How the Program Works

It is assumed that there will be correct input, so there is no checking to make sure that the input is correct. However, there is checking to make sure that the memory being stored or accessed is not outside the bounds of the array. If that happens, a friendly message will be output to the screen, and the program will continue. If the user wishes to exit the program, they can type 'exit', ctrl-C, or ctrl-D.

1.2.1 Storing Memory

For storing memory, the user will enter the starting location (in Hex) where they want to start storing memory, a ':' and then what they are storing. In the code, it will separate the starting position in the memory and the items that are being input. I used the .split on the : and then the ' ' so a list could be returned. Then, the list of what to store is put into the memoryList.

1.2.2 Printing Memory

The user can either choose to print a range from the memory or just a single address. If they want a single address from memory, only that address and what is in the memoryList is printed. If they give a range, the program will print everything in that range from the memoryList. The format for this is to start by printing the current place in memory first, the next 8 bits, and then repeating on a new-line until everything is printed. If the starting position is not an 8-bit multiple, the output will be indented to indicate that.

1.2.3 Loading From a File

The program can also load storing memory from a file. This is done using the IntelHex library. This makes things super easy because it can take the file, separate the addresses and what goes into that place. This makes it a simple for-loop to store the memory from the file.

1.2.4 Running the Program

If the user inputs a memory address with an ’R’ at the end, the program will print out the format and then print out the memory address. The memory address will also be stored in a variable that is called programCounter.

1.3 How I Tested

I tested this by using the input provided to us in the write-up along with the files given to us. I also put in my own input along the way and after to make sure that is was correct. Testing it was not terribly hard, and it was clear when the program was correct.

2 Part 2

This part was to print out the program counter, opcode, instruction, the memory and the registers for instructions until a break occurs.

2.1 What I Did

I first started by getting the user input from where the would like would like to start printing in memory- opcode. In another file, I have a dictionary called instructions. Each opcode had its corresponding instruction name and amod which were given to us in the file. I then entered a while loop that will run until the the instruction given becomes a break- opcode 00. I grabbed the instruction and amod from the dictionary using the opcode the user gave us. I then went into a bunch of if else statements and doing what needed to be done based on the instruction code. If the user wishes to run the program again without quitting, the registers are reset, but the memory is not. I could write what each of the instructions did, but I don't think anyone wants to read it since they're in the document given to us. I wrote out what to do on the harder instructions, but most of them were very clear in what they were doing.

2.2 How I tested

I tested by running my code against the two pieces of code given to us in the program document. For the ones that weren't given to us, I tested them by hand to make sure they were doing what I thought they were doing. It was kind of hard to test the ones that weren't given to us since I wasn't always sure what the output should be

3 Part 3

This part of the program was to continue implementing instructions just like in part 2. Not a whole lot to explain other than the it was a little more complicated.

3.1 How Far I Got

I would like to be graded to tier B. I got most of it done and tested some.

3.2 For C Grade

3.2.1 What I did

I implemented the immediate and zeropage instructions for ones given to us. From what I understood, the difference was where items were stored.

3.2.2 How I tested

I tested by by comparing what I had against the examples given to us. The rest were tested by me making up my own test cases. It's kind of hard to tell if they're correct or not when I don't really have the answers/what the output should be. There were plenty of times where I thought I coded things correctly, but they weren't correct based on the sample output given to us. For some, I was able to assume that because they were done correct in the last program, they would be done correctly now. I could also use some immediate outputs to test the zeropage outputs and vise-versa.

3.3 For B Grade

3.3.1 What I did

For this section, I attempted to implement the jumping, branching, relative, and absolute opcodes. Things were pretty much the same as in part C except how the data from memory was grabbed and stored (through the operands)

3.3.2 How I tested

I tested by comparing my outputs to the outputs given to us. I pretty much tested the same way I did in part 2. A lot of the functions were similar, so I didn't have to worry about testing them to much if the part C ones were tested well enough that I trusted them.