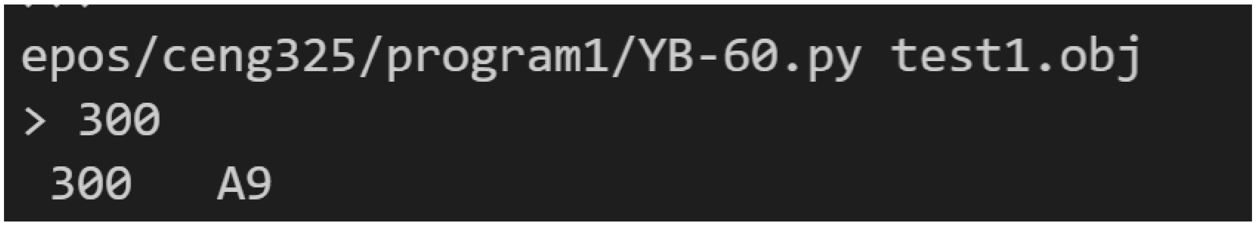
The YB-60 emulator can be run by navigating to the folder which the Python file is stored in, and typing “python3 YB-60.py” at the command line. You can also add another argument, the name of the Intel HEX format file which you wish to pass in. An example for this usage would be: “python3 YB-60.py test1.obj”. In some rare cases, the “python3” may not be required- keep this in mind if the program is not running as expected.

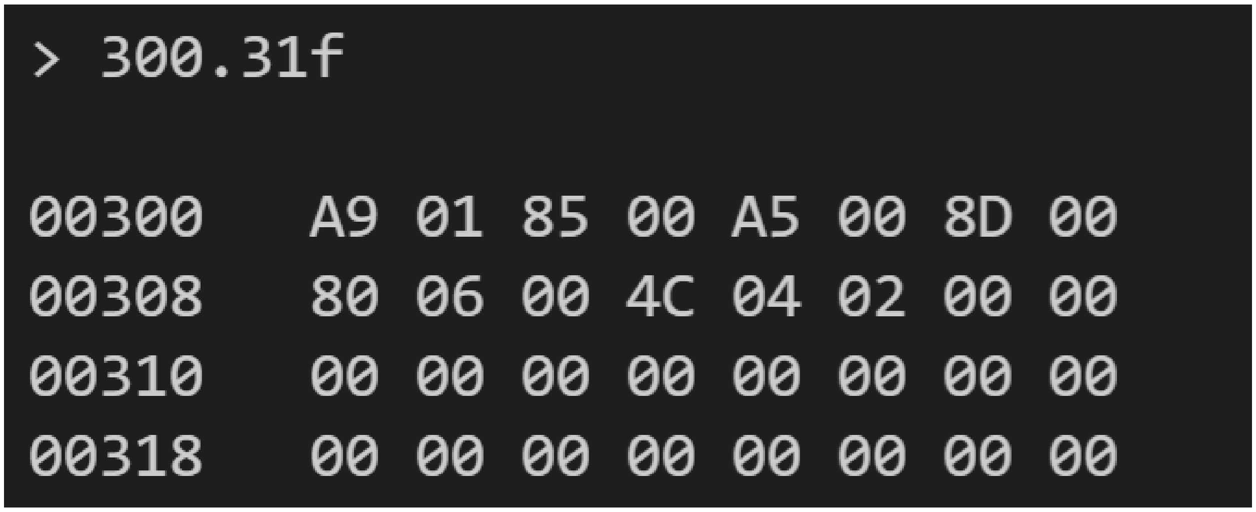
Upon starting the program, you will be prompted with a “>”. Various commands can be typed in at this point. These include:

Display a memory address: By typing in a memory address (in hex), you can view the contents of the address. Example:



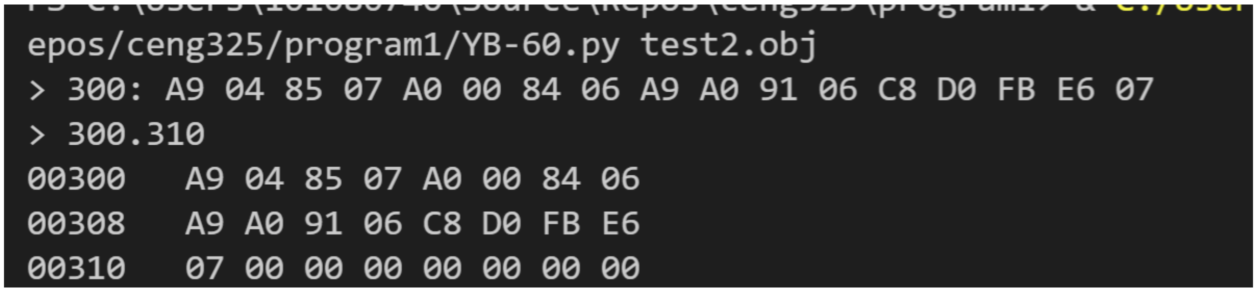
```
epos/ceng325/program1/YB-60.py test1.obj
> 300
 300   A9
```

Display a range of memory addresses: By typing in a hex memory address, followed by a period, followed by another memory address, you can view the contents of all the memory addresses in the range which you specified. These addresses are output in blocks of 8 at a time. Example:



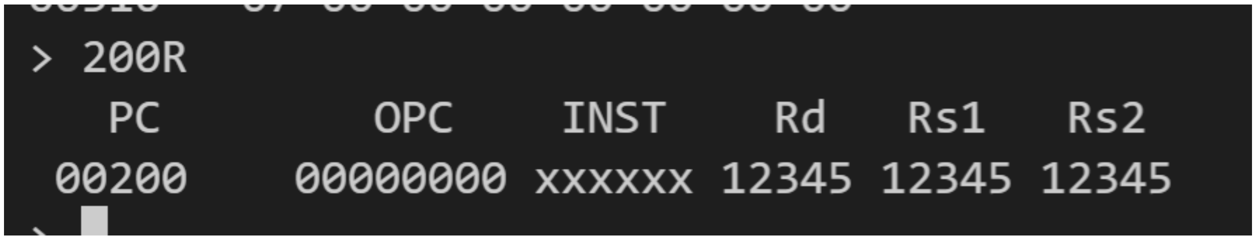
```
> 300.31f


00300   A9 01 85 00 A5 00 8D 00
00308   80 06 00 4C 04 02 00 00
00310   00 00 00 00 00 00 00 00
00318   00 00 00 00 00 00 00 00
```

Edit memory locations: By typing in the memory address you wish to start at, followed by “: “ and then the data that you wish to fill the addresses with, you can change the data stored in the addresses. Example:



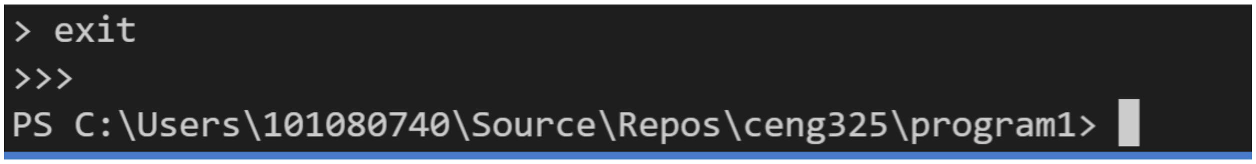
```
epos/ceng325/program1/YB-60.py test2.obj
> 300: A9 04 85 07 A0 00 84 06 A9 A0 91 06 C8 D0 FB E6 07
> 300.310
00300   A9 04 85 07 A0 00 84 06
00308   A9 A0 91 06 C8 D0 FB E6
00310   07 00 00 00 00 00 00 00
```

Run a program: More functionality will be added for this later, but as of now all that this option does is set the program counter to the address which was input, clear the registers, and stop. This can be accomplished by inputting the register which you want to start from followed by an R, with no spaces. Example:



```
> 200R
   PC         OPC     INST     Rd    Rs1    Rs2
 00200    00000000 xxxxxx 12345 12345 12345
```

Exit the monitor: The easiest way to exit the monitor is to type "exit" at the prompt. The monitor will output ">>>" to signify that it has stopped successfully, and then exit the program. Example:



```
> exit
>>>
PS C:\Users\101080740\Source\Repos\ceng325\program1>
```

Functions:

readValuesAndFillMemory

This function reads the lines in from the input file, checking each record type and filling whatever data is to be filled in. I tested this function with lots of print statements, confirming that each variable was set to the expected value before moving on. This was important because all of the future functionality in the program depended on being able to reliably get the information in from the file.

monitor

This function outputs the monitor prompt to the screen and awaits a request from the user. Once the user's request has been input, it checks to see which function to send the information from the command line to. This function is more of a hub to call all of the other ones. At the end of the function, it calls itself recursively- this way, the user can use the monitor multiple times without rerunning the program. This function was tested by inputting various requests from the monitor and confirming that they were each sent to the correct function down the line.

runProgram

This function was written very simply to set the register and output what is expected. The testing was for the formatting of the output. There was not much more to do with this one just yet because all that it is expected to do at this point is set the program counter to the register which was input. I could visually check this based on the terminal output.

editMemoryAddress

This function is another one which is called by monitor, and it edits the memory addresses as directed by the console input. I checked this first by printing the memory addresses myself. After I got my display functions tested and working I was able to use the display command at the monitor prompt to check that the memory addresses were edited appropriately.

displayMemoryAddress

This function simply goes into the memory and outputs the contents of the address. This was easy to test, as I knew what the contents of the addresses should be after reading in each file. I was able to simply print the contents of the address and make sure that what this function outputted to the console matched.

displayRangeOfMemoryAddresses

This function is similar to the displayMemoryAddresses function, but it displays addresses based on the range input from the console. It outputs the contents of the addresses in groups of eight, meaning that if the console requests something that does not evenly divide by eight then a bit of extra data will be output. On each new line, the current memory location is output for ease of viewing. This was tested similarly to the displayMemoryAddress function, in that I knew what each address should be, and I could make sure that what was output matched the data that I had as well as what was put in from the file. This became a very important function for testing that the others worked, as well.

Program

I tested the whole program using the provided testing files, as well as doing some of my own random poking and prodding with the monitor as I went. By the end, all of the functionality for each test file worked as expected, and I followed the sample inputs and outputs in the write-up to further confirm. Beyond that, it was just a lot of checking as I went, using the little interface which I created in the command prompt.