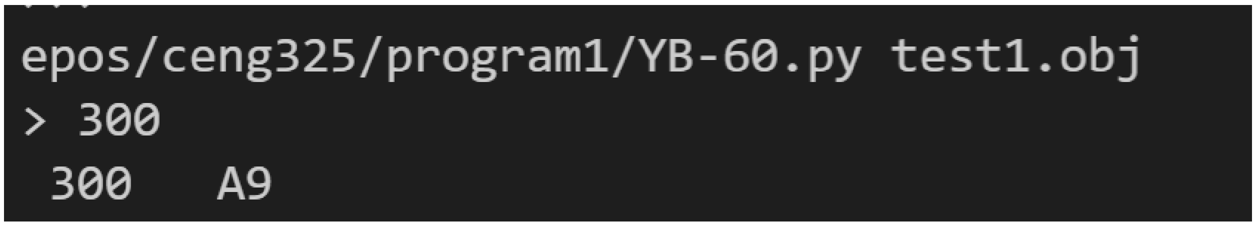
The YB-60 emulator consists of 33 registers, the Program Counter and x0 through x31. It is a 32-bit byte-addressable base integer architecture, and has 2^{20} bytes of memory. It supports the RV32I base instruction set and the RV32M standard extension, but may support more in the future.

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.

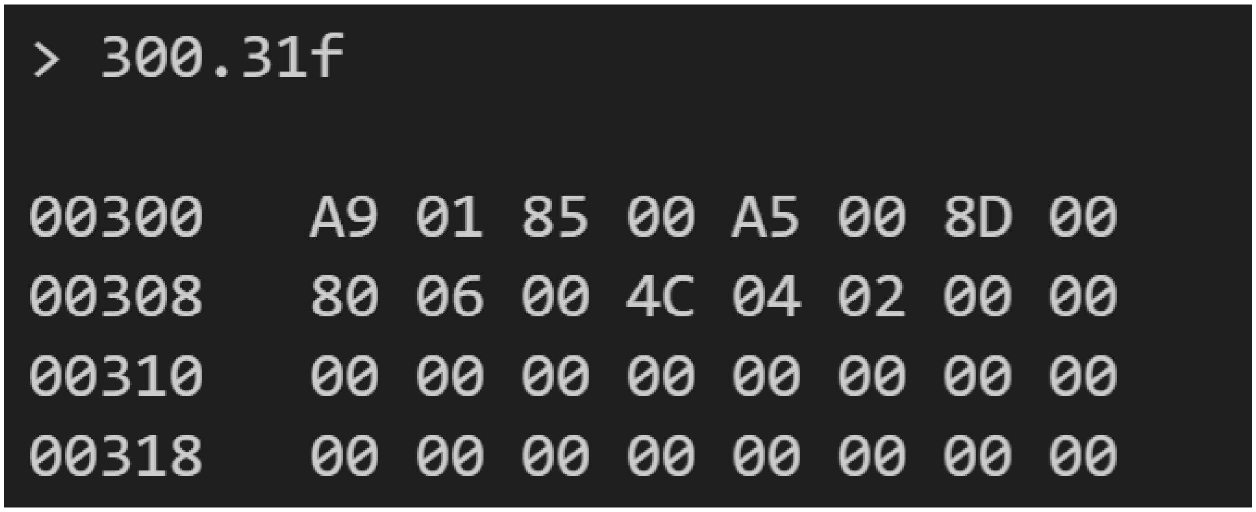
Upon starting the program, you will be prompted with a “>”. Various commands can be typed in at this point. These include:

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
```

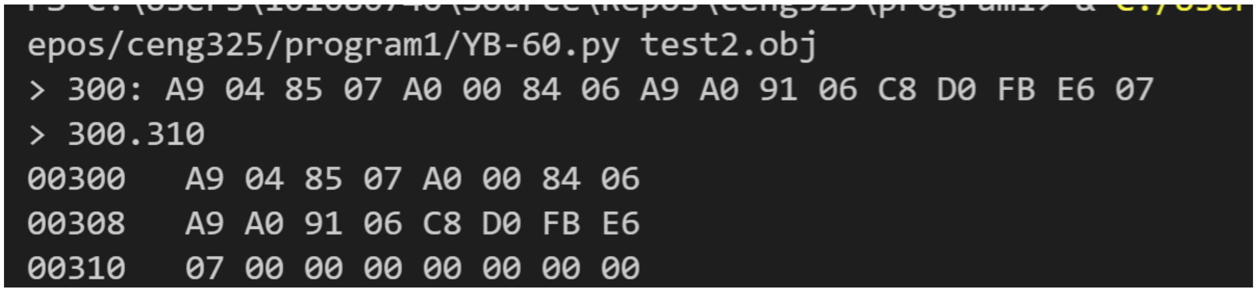
Display a range of memory addresses: By typing in a hex memory address, followed by a period, followed by another memory address, you can view the contents of all the memory addresses in the range which you specified. These addresses are output in blocks of 8 at a time. Example:



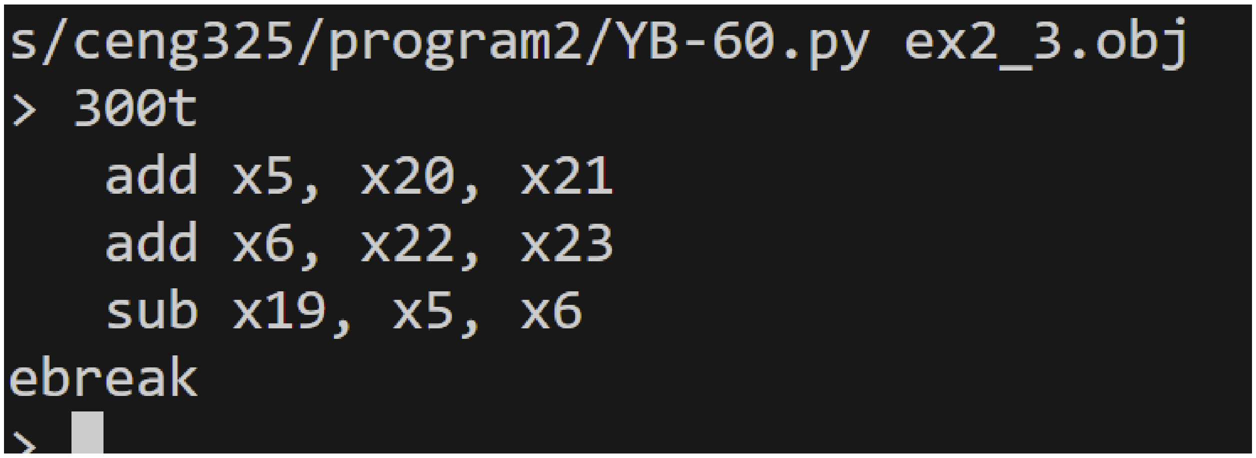
```
> 300.31f


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
```

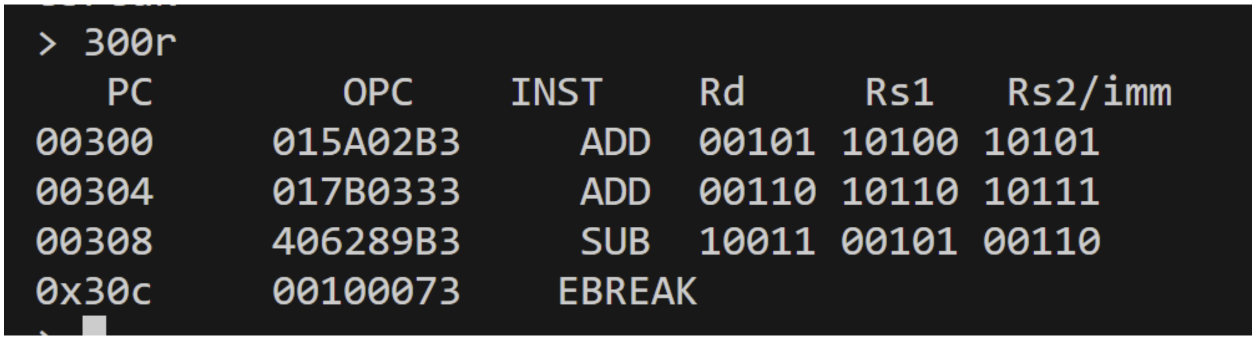
Edit memory locations: By typing in the memory address you wish to start at, followed by ": " and then the data that you wish to fill the addresses with, you can change the data stored in the addresses. Example:


```
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
```

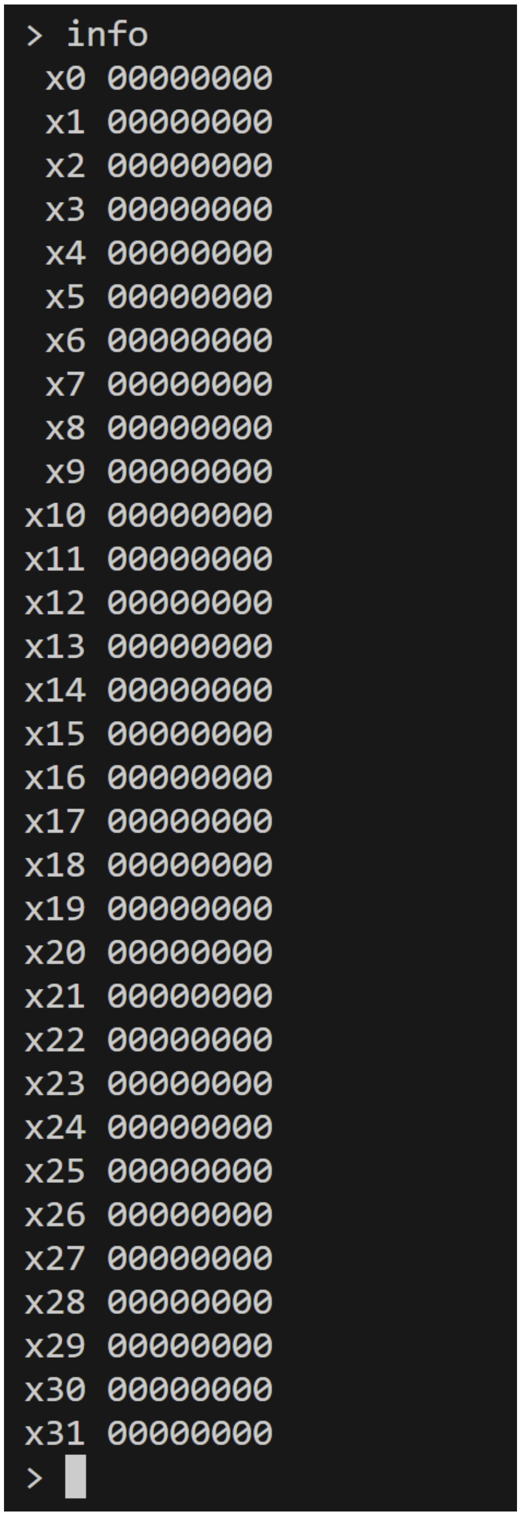
Disassemble object code: By typing in the starting memory address followed by a "t" or "T", you may disassemble the object code from that address until an EBREAK is reached. This code will be output to the console as readable instructions. Example:


```
s/ceng325/program2/YB-60.py ex2_3.obj
> 300t
   add x5, x20, x21
   add x6, x22, x23
   sub x19, x5, x6
ebreak
> 
```

Run a program: More functionality will be added for this later. As of now, this sets the program counter to the register set, and decodes the instructions from each little-endian set of instructions following the beginning register. This continues until an EBREAK is reached, and as of now just outputs the resulting instructions to the console. Example:

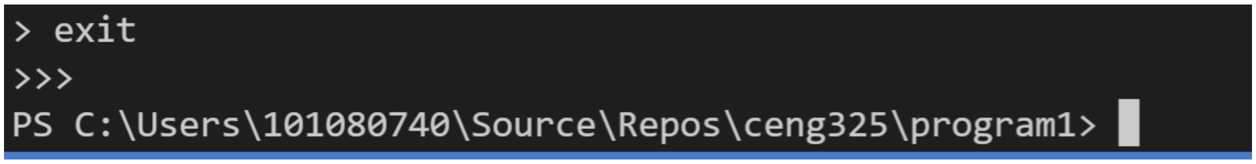

```
> 300r
  PC         OPC      INST     Rd      Rs1    Rs2/imm
00300      015A02B3     ADD   00101  10100  10101
00304      017B0333     ADD   00110  10110  10111
00308      406289B3     SUB   10011  00101  00110
0x30c      00100073    EBREAK
```

Display content of registers: By typing "info" into the monitor, you can output the contents of the registers of the YB-60. As of now, since the run functionality does not touch the registers, all of the registers will output as zeros. This should be fixed in a future generation of the YB-60 in which the run is fully implemented. Example:



```
> info
 x0 00000000
 x1 00000000
 x2 00000000
 x3 00000000
 x4 00000000
 x5 00000000
 x6 00000000
 x7 00000000
 x8 00000000
 x9 00000000
x10 00000000
x11 00000000
x12 00000000
x13 00000000
x14 00000000
x15 00000000
x16 00000000
x17 00000000
x18 00000000
x19 00000000
x20 00000000
x21 00000000
x22 00000000
x23 00000000
x24 00000000
x25 00000000
x26 00000000
x27 00000000
x28 00000000
x29 00000000
x30 00000000
x31 00000000
> 
```

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 has been rewritten from the previous iteration of the YB-60, in order to take in the filled memory and the starting address and decode the instructions from there. It takes chunks of hex, converting them into 32-bit binary strings in the correct format to be used for decoding. It calls the corresponding format function depending on which family the instruction is found to be in. That format function then returns with the necessary information from the instruction, found from the 32-bit binary string. Finally, the runProgram function prints the resulting instructions to the console. I was able to test this function using the test files provided in the documentation of the assignment, and I could use the monitor to access and run different memory locations from each file. I compared my output to what was expected, or what was given in the example outputs in the assignment document.

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.

Disassemble

This function works very similarly to the runProgram function for now, in that it reads in the memory, converting it to a 32-bit binary string, and decodes the instructions using the same algorithm. This continues on until an EBREAK is reached, the same as the runProgram function. These will diverge later, in that this function will not run the program in the end, but for now they do similar logic and output somewhat differently to the console- this function's output is made to be more readable to the human eye. Since this and the runProgram function are so similar, I was able to test this similarly, calling the disassemble function using the monitor and checking the output against what was expected.

I_format, S_format, B_format, R_format, U_format, and J_format

All of these functions take in the 32-bit binary string and decode it, depending on the family which the instruction is in. Each of these functions represents a family, so they each decode somewhat differently. This is all based on the reference sheet, and they each return a combination of some or all of the following : immediate value, instruction (mnemonic), rd, rs1, and rs2. These values will be used in the future to run the program, but for now they are just used to be output. These functions are all called in the disassemble and runProgram functions, and were tested using lots of print statements until I was confident that they were working as expected. After that, I was able to use the disassemble and runProgram functionalities of the monitor to test that these were still working as expected- if something didn't look right with my output, it was probably due to me assigning the wrong mnemonic to a specific funct3, or something of the sort.

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.