MCPU Emulator

Joe Leveille and Colin Braun

Subproject Summary

The purpose of an emulator is to show the user what is actually happening on the chip, as it is impossible to physically see if a CPU is running as intended. So, the emulator takes the same input as the CPU and emulates(!) what the CPU will do with those instructions.

We created an emulator which would take in a file containing the instructions to be loaded directly into the RAM, execute each instruction, and print out the registers and changes to memory as it runs.

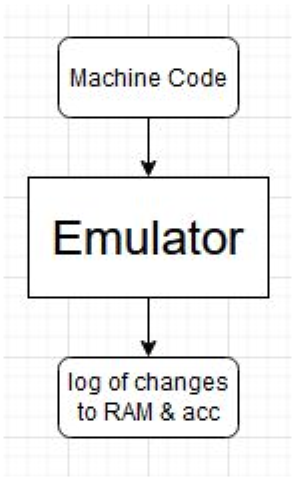

Figure 1. Big picture diagram of our subproject

How Does it Work?

The diagram below shows how the program handles its job - first reading the machine code into our simulated RAM, which is the Python list named “mem”. Each instruction is broken into the opcode and argument, then interpreted as specified in the provided MCPU documentation. At the start of each instruction, the states of the relevant registers will be displayed. Finally, the script will stop once the program runs the final instruction in location 63. Examples of the specific inputs and outputs are discussed in sections “Expected File Format” and “Emulator Output”. The script can be run from a terminal with Python 3 installed by running the following command:

Command to send output to a log file:

```
py ./emulator.py machineCodeFile.txt > out.log
```

The argument “machineCodeFile.txt” is exactly what it sounds like - the file that should contain the initial contents of the simulated RAM given by the assembler. The “>” specifies that the output from our emulator should be placed in the file in the next argument, which is “out.log” in this case. The file out.log will be created and filled with the program output, or cleared and updated if it had previously existed in the present directory. If the “ > out.log” is ommitted from the command above, the script will run as normal, but will output to the terminal window instead.

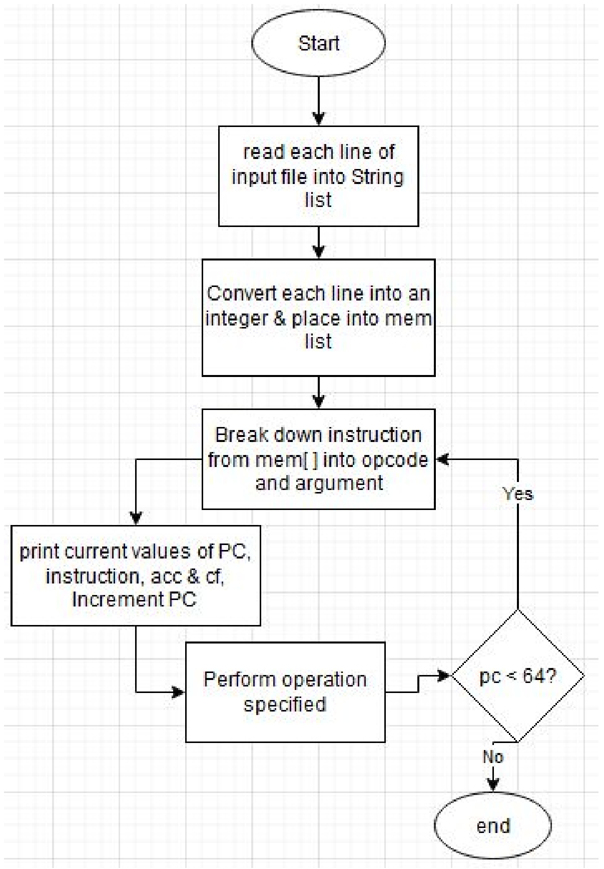
Start
read each line of input file into String list
Convert each line into an integer & place into mem list
Break down instruction from mem[ ] into opcode and argument
print current values of PC, instruction, acc & cf, Increment PC
Perform operation specified
pc < 64?
Yes
No
end

Figure 2. Detailed flowchart - Shows the workflow of the program

Expected File Format

The input file that is read is an ASCII text file produced by the assembler. Each line contains a byte of memory in hex, with or without the 0x designator, as either will be interpreted correctly by the Python script.

Example Input:

Line Num	Value	Action to Emulate
1 (0x00)	0x47	Add the value stored at 0x07 to the accumulator

2 (0x01)	0x8F	Store the value in the accumulator into 0x0F
3 (0x02)	0x4F	Add the value stored at 0x0F to the accumulator
4 (0x03)	0x3F	NOR the accumulator with the number at address 0x3F (which is 0).
5 (0x04)	0x8C	Store the value in the accumulator into 0x0C
6 (0x05)	0xF8	Jump if carry not set (it **is**, so the jump **does not** happen) to 0x38.
7 (0x06)	0xF8	Jump if carry not set (it **is not**, so the jump **does** happen) to 0x38.
8 (0x07)	0x83	Data. This is read and used by the first instruction.
9-64	0x00	Unused memory. If jumped to, will begin executing NORs until done.

Emulator Output

The emulator will print out the values of each register onto a line **before** each instruction is executed.

Whenever a change is made to RAM, it will be printed like so:

- `+12 0x3F` (meaning RAM[12] <= 0x3F).

Example Output from Example Input:

```
pc,ir,acc,cf:  0     0x47   0     0
pc,ir,acc,cf:  1     0x8f   131   0
+ 15 0x83
pc,ir,acc,cf:  2     0x4f   131   0
pc,ir,acc,cf:  3     0x3f   6     1
pc,ir,acc,cf:  4     0x8c   249   1
+ 12 0xf9
pc,ir,acc,cf:  5     0xf8   249   1
pc,ir,acc,cf:  6     0xf8   249   0
pc,ir,acc,cf:  56    0x0    249   0
pc,ir,acc,cf:  57    0x0    0     0
pc,ir,acc,cf:  58    0x0    184   0
pc,ir,acc,cf:  59    0x0    0     0
pc,ir,acc,cf:  60    0x0    184   0
pc,ir,acc,cf:  61    0x0    0     0
pc,ir,acc,cf:  62    0x0    184   0
pc,ir,acc,cf:  63    0x0    0     0
```

Example Output Analysis

The above output shows the results of running the aforementioned example input. The following shows a instruction-by-instruction analysis of why the output is what it is:

1. The first instruction will take the value at 0x07 (which is 0x83 or 131_{10}), and add it to the accumulator (which starts with a value of 0).
2. The value in the accumulator is then stored into 0x0F, and it is here that we can see the emulator displays the change to RAM ("+15 0x83").
3. Next, the value at 0x0F (which we just changed to 0x83 or 131_{10}), is added to the accumulator. However, we can't fit 0x83+0x83 into the accumulator, so it overflows. The carry flag is set, and we will end up with a value of (0x83+0x83)-0x100 = $131_{10}+131_{10}-256_{10} = 6_{10}$ = 0x06.
4. The accumulator is then NOR'd with the value located at 0x3F (which is 0x00) and stored back into the accumulator. NORing a number with 0 is the same as inverting the number, so INV(0x06) = 0xF9 = 249_{10}
5. The value stored in the accumulator (0xF9 = 249_{10}) is stored in memory location 0x0C.
6. We now want to perform a jump if carry not set, but the carry is set. However, by executing the jump if carry not set instruction, the carry flag is cleared (you can see this in the output).
7. We then perform another jump if carry not set instruction, forcing the jump to the desired location of 0x38 = 56_{10}, a location near the end of memory full of zeros.
8. Normally, if we wanted to end the program, we would jump to the end of the program (0x3F = 63_{10}), but for the sake of demonstration, we show what happens when memory initialized to zeros (likely by the assembler because the assembly code provided did not take up the full 64 bytes of memory) near the end of the program results in. Since these are read as instructions, we are, on each byte until the end of the program, NORing the value of the accumulator with the value stored at address 0x00 (which is 0x47 = 71_{10}). This produces the remaining output for PC = 56_{10} to 63_{10} as shown.