Building a Virtual Machine

In this exercise, you will write a program that simulates the behavior of computer hardware. While our primary objective is to learn how physical computers work, the exercise will also give us a glimpse into how virtual machines form a critical component of high level language implementations like those of Python, JavaScript and Ruby.

By working through this exercise you will learn about both how physical computers work, and how virtual machines are implemented.

The Computer

The device we are simulating is much simpler a modern CPU, it has:

- 20 bytes of memory
- 3 registers
  - 2 general purpose registers for data
  - 1 "Program Counter" which stores the address of an instruction
- 5 instructions

Memory

For the computer in this exercise, main memory is 20 bytes. We will use a fixed size array to model memory in our virtual machine; virtual main memory is an array where `length == 20`. In a physical machine each particular memory address has a specific physical path that connects it to the rest of the hardware; in our virtual machine those paths are represented by the indices of the array.

Here is one way to picture our memory:

```
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19
__ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __
```

For our computer, memory is divided into 3 sections: instructions, input, and output.The instructions occupy the first 14 bytes, followed by 2 bytes for output and 4 bytes for two separate 2 byte inputs.:

```
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19
__ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __
INSTRUCTIONS --------------------------^ OUT-^ IN-1^ IN-2^
```

The two bytes of output always represent one 2-byte number. The two inputs each also always represent 2-byte numbers. Our computer is a "little endian" system, meaning the "least significant byte" of our inputs and outputs occupy the smaller array index location. Consider the following input:

```
byte 18: 10100001
byte 19: 00010100
```

Byte 16 corresponds to the binary digits for 2^0 through 2^7. It has the value `2^7 + 2^5 + 2^0 = 161`. Byte 17 corresponds to the binary digits for 2^8 through 2^15 and has the value `2^10 + 2^12 = 5120`. Therefore, these two bytes represent the number `5120+161=5281`

Another useful way to think about this is to label the device like this:

```
byte 18: 10100001 x 1
byte 19: 00010100 x 265
```

Thinking about it this way, we have the binary number 161 on top, and the binary number 20 on bottom. This yields: (1*161) + (256*20). We're relieved to find that `256*20=5120` so our interpretations are indeed equivalent.

In the above scenario, the user encoded the decimal number 5120 by dividing it by 256 and entering 14 in binary on the bottom row, then they entered the remainder 161 on the top row. Unfortunately, the device interface is a little confusing, but it cannot be changed.

Note that this construction means that the "little end" of the number (161 x 1) comes first in memory (byte 18) and the "big end" (20 x 256) comes after (byte 19). Intel computers encode multi-byte numbers in this order, so we are in good company.

Short Aside: Hexadecimal Notation

Decimal is base 10, binary is base 2, hexadecimal is base 16. Counting to 16 in hex looks like this:

`0 1 2 3 4 5 6 7 8 9 a b c d e f`

Binary values take a lot of space and are hard to read. So we often use hexadecimal notation, which is hard to read but takes up less space (and not any harder to read than binary, honestly). Every 4 bit pattern can be interpreted as number between 0 and 15. But with hex notation we can use a single digit and say `0` to `f`.

```
       byte 18                        byte 19
```

left nibble right nibble left nibble right nibble 1010 0001 0001 0100 a 1 1 4

Using hexadecimal to encode the number 5120 from above into our second input location looks

like this:

```
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19
__ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ a1 14
```

Rewriting the indexes in hex, (because that's how you will see them represented in most software that prints memory addresses) looks like this:

```
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13
__ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ a1 14
```

The output section of memory looks exactly like the 2 input sections we've already described but represents output.

Instructions

The rest of the array is for the stored program.

In order to write the stored program we need to understand one more characteristic about this computer: the "arithmetic unit" can't read values from main memory, only from two internal memory cells called "A" and "B". Unlike ram, these cells are 16bits wide, meaning the computer can do 16bit operations in a single "cycle" of work.

When the computer is done performing an arithmetic operation such as addition with the values from A and B it writes the result back to the A register. Just like it can't read from main memory, it also can't write to main memory.

We have 14 bytes of main memory left to encode our stored program, and this is what we need it to do (the trailing h means the number is hex):

load_word $a (10h) # Load input 1 into register a load_word $b (12h) # Load input 2 into register b add $a $b # Add the two registers, store the result in register a store_word $a (0Eh) # Store the value in register a to the output device halt

There are 5 kinds of instruction, and we can map each to a byte value

load_word 01h store_word 02h add 03h sub 04h halt FFh

There are 2 general purpose register addresses so we can map those to two byte values. There is also a program counter - a register that stores the address of the current instruction. The PC should always be incremented after each instruction, and no instruction can modify it's value in any other way.

A 01h B 02h PC 03h

The memory addresses are already byte values, so now we can represent the whole program in terms of bytes:

load_word $a (10h) 01h 01h 10h load_word $b (12h) 01h 02h 12h add $a $b 03h 01h 02h store_word $a (0eh) 02h 01h 0Eh halt FFh

And now we can write this stored program into memory.

```
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13
01 01 10 01 02 12 03 01 02 02 01 0E FF 00 00 00 __ __ __ __
                                        ^^^^^ ^^^^^ ^^^^^
                                        out   in X  in Y
```

For users to use the machine, they manually set the input (and instructions) then push a button causing the machine to "run". In our simulation this will mean pre-defining arrays, and calling a function, then printing the value in the arrays "output" location.

In a physical device it could mean a number of things, such as bringing a set of punch cards as instructions, manually flicking switches for the inputs, and reading some screen output or a series of light-bulbs that were on for each "on" bit in the output section of memory. Computers have been created in numerous ways prior to modern CPUs.

The Exercise

Write a "virtual computer" function that takes as input a reference to main memory (an array of 20 bytes), executes the stored program by fetching and decoding each instruction until it reaches halt, then returns. This function shouldn't return anything, but should have the side-effect of mutating "main memory".

Your virtual computer should have one piece of internal state, an array of three registers: PC (program counter), A, and B. Main memory is considered external state because it is passed in as an argument.

Write some tests for your virtual computer function using different stored programs and different input values. Your test inputs should just be array literals representing main memory, the expected outputs should be the same.