Assignment 2 — LC-3 Programming

Due: Wednesday, January 24th

In previous assignments, you've instructed the LC-3 to execute singular, standalone instructions by encoding those instructions directly into memory. The LC-3 is, however, capable of executing multiple instructions in succession. For this assignment, you'll continue exploring the LC-3's instruction set and write your first true LC-3 program.

Deliverables:

Demo — Demonstrate your solution for Part 3a (running on the LC-3 simulator of your choice). For extra credit, also demonstrate your solution for Part 3b.

Write-up — Answer the questions throughout this assignment. Fill out your answers *at the end* of this assignment, where the questions have been copied with space for answers.

Handin — `part3a.bin` (required)
`part3b.bin` (extra credit)
GitHub Classroom Link: `https://classroom.github.com/a/uBhn4-Ka`

For this assignment, you may work alone or with *one* partner. However, the handin portion must be completed individually. Remember to write both of your names on the write-up.

Part 1: The LC-3 Instruction Set

As with any (good) programming language you'll encounter, LC-3 instructions have a reference describing their syntax, although in this case, that reference is rather small, since the LC-3 only understands fifteen instructions:

- `http://users.csc.calpoly.edu/~cesiu/csc225/lc3/lc3instSet.pdf`

Since instructions are encoded as sequences of bits just like any other data, it's possible to interpret an instruction as some other type of data; it's also possible to interpret some other piece of data as an instruction.

For instance, in addition to numbers, addresses, and instructions, binary values could be interpreted as characters according to their ASCII values. As you may have seen in previous classes, the ASCII table is a mapping of binary codes to some of the most basic characters and symbols, as shown here:

- `https://en.wikipedia.org/wiki/ASCII#Printable_characters`

1. What LC-3 instruction is formed by the binary representation of your initials?

   For example, Professor Hatalsky's initials are 'P' and 'H', which correspond to the ASCII codes `0x50` and `0x48`, respectively. In binary, this is `0101 0000 0100 1000`, which is the LC-3 instruction "`AND R0, R1, R0`" (technically, bit 3 should be a '`0`', but that's just padding and will be ignored).

   If your initials do not form a legal instruction, explain why.

Part 2: `addNums`, an LC-3 Program

Begin by accepting the GitHub Classroom assignment: `https://classroom.github.com/a/uBhn4-Ka`. Your repository for this assignment contains some starting files, including two complete LC-3 programs.

Take a quick look at the `addNums.bin` file (despite the "`.bin`" extension, it's just a plain text file that can be opened with your favorite text editor). This file contains a complete LC-3 program, in the form of instructions encoded in binary.

Actually, that's not entirely true. `addNums.bin` contains instructions written using ones and zeroes, but those are the *characters* '1' and '0', not actual bits. The file also contains comments (a '`;`' indicates the start of a comment), clearly not in binary. We'll convert this to a true binary file before loading it into the LC-3.

The LC-3 itself does not support file I/O. However, since it would be tedious and error-prone to load programs into the LC-3 by encoding their instructions directly into memory, the LC-3 simulators are capable of loading data into memory from files.

Windows Simulator Users:

1. Run `LC3Edit.exe` and open `addNums.bin` within it.
2. Click the "B" button in the toolbar. There should be no errors reported at the bottom of the window.
3. Still in LC3Edit, open `data.hex`.
4. Click the "X" button in the toolbar. Again, you should see no errors.
5. Verify that you now have two new files: `addNums.obj` and `data.obj`.
6. Run `Simulate.exe`.
7. Load both `addNums.obj` and `data.obj` into the simulator.

Web Simulator Users:

1. Click the "Raw" button.
2. Drag (or paste the contents of) `addNums.bin` into the popup and click "Process". You should see a green field reporting no errors.
3. Click "Load into Simulator".
4. Repeat this process for `data.hex` (you may need to clear the contents of the "Raw" popup, leftover from the first file you loaded).

Notice that the program was loaded starting at memory location `0x3000`, while the data was loaded starting at `0x3100`. Now that we've loaded our instructions (along with the data they'll process) into the LC-3, we need to make sure that the LC-3 knows where to find its first instruction. Make sure the Program Counter contains `0x3000`.

Being a complete program, the `addNums` program ends with a special instruction at location `0x3009`, "`TRAP HALT`", that tells the LC-3 that the program has finished. However, the LC-3 does some cleanup before it shuts down, which might mess up some of the registers' contents. For the exploratory purposes of this assignment, we don't want that to happen. Set a breakpoint on `0x3009`.

The LC-3 simulators allow you to step through a program, executing just one instruction at a time, which allows you to see the state of the computer after each instruction has been executed. Step through the program until you understand what it does.

You'll need to reset the PC to `0x3000` if you want to restart the program from the beginning. Once you understand what each instruction does, reset the PC and run the program to completion. The LC-3 should halt with the PC set to `0x3009`.

1. What is the total sum calculated by the `addNums` program?
2. How is `R2` used by the program?
3. Why is `R2` incremented by the instruction at `0x3005`?
4. What does `R4` represent, and why is it initially set to decimal 10?

Part 3a: Counting '1's

An important characteristic of a binary value is the number of '1's within it. Develop an LC-3 program that counts the number of bits set to '1' in the value at memory location `0x3100`. Store this number in memory at location `0x3101`.

For example, if memory contains:

Address	Data
	...
0x30FF	?
0x3100	0110 1000 1111 0110
0x3101	?
0x3102	?
	...

Then, after running your program, it should contain:

Address	Data
	...
0x30FF	?
0x3100	0110 1000 1111 0110
0x3101	0000 0000 0000 1001
0x3102	?
	...

Strive to make your program as short as possible. A clear, concise program should accomplish this task in about 11 instructions. A slightly less readable program can do it in 8 (including the final `HALT`).

While working on this program, you may find it useful to read through `evenOdd.bin`. This program determines the parity of the value at memory location `0x3030`, storing '1' at memory location `0x3031` if it is odd and '0' if it is even.

Requirements:

- Write your program in a file named "`part3a.bin`".
- Your program must start at memory location `0x3000`.
- After running your program, memory location `0x3101` must contain the number of '1's in the value at memory location `0x3100`.
- You must not mutate the value stored at memory location `0x3100`.
- Aside from loading the value at location `0x3100` and storing the count at location `0x3101`, you may not use memory for data.
- Your program must use a loop to iterate through the bits in the value at location `0x3100` (the value whose '1's are being counted).
- Your program must be able to be run multiple times by resetting the PC to `0x3000`. It should not require that the LC-3 be reinitialized, that any files be reloaded, or that any registers be reset.
- Each instruction in your program must be commented.
- Your source code must include comments describing the purpose of each register you use. Additionally, include a comment with your name (and your partner's name, if applicable).

Part 3b: Reversing Bits

Note that this part is *extra credit*. I highly suggest that you complete and thoroughly test your solution to Part 3a before beginning this part.

Develop an LC-3 program that reverses the bits in the value at memory location `0x3100`. Store this reversed value in memory at location `0x3101`.

For example, if memory contains:

Address	Data
	...
0x30FF	?
0x3100	1010 1001 0101 1111
0x3101	?
0x3102	?
	...

Then, after running your program, it should contain:

Address	Data
	...
0x30FF	?
0x3100	1010 1001 0101 1111
0x3101	1111 1010 1001 0101
0x3102	?
	...

Requirements:

- Write your program in a file named "`part3b.bin`".
- Your program must start at memory location `0x3000`.
- After running your program, memory location `0x3101` must contain the reversed value at memory location `0x3100`.
- You must not mutate the value stored at memory location `0x3100`.
- Aside from loading the value at location `0x3100` and storing the count at location `0x3101`, you may not use memory for data.
- Your program must use a loop.
- Your program must be able to be run multiple times by resetting the PC to `0x3000`. It should not require that the LC-3 be reinitialized, that any files be reloaded, or that any registers be reset.
- Each instruction in your program must be commented.
- Your source code must include comments describing the purpose of each register you use. Additionally, include a comment with your name (and your partner's name, if applicable).

Part 1: The LC-3 Instruction Set

1. What LC-3 instruction is formed by the binary representation of your initials?

   For example, Professor Hatalsky's initials are 'P' and 'H', which correspond to the ASCII codes `0x50` and `0x48`, respectively. In binary, this is `0101 0000 0100 1000`, which is the LC-3 instruction "`AND R0, R1, R0`" (technically, bit 3 should be a '`0`', but that's just padding and will be ignored).

   If your initials do not form a legal instruction, explain why.

Part 2: `addNums`, an LC-3 Program

1. What is the total sum calculated by the `addNums` program?

2. How is `R2` used by the program?

3. Why is `R2` incremented by the instruction at `0x3005`?

4. What does `R4` represent, and why is it initially set to decimal 10?