Assignment 1b — Introduction to the LC-3

Due: Friday, January 19th

The LC-3 understands fifteen basic instructions, which can be directly encoded into memory. For this assignment, you'll familiarize yourself with encoding and running simple LC-3 programs.

Deliverables:

Demo — *none*

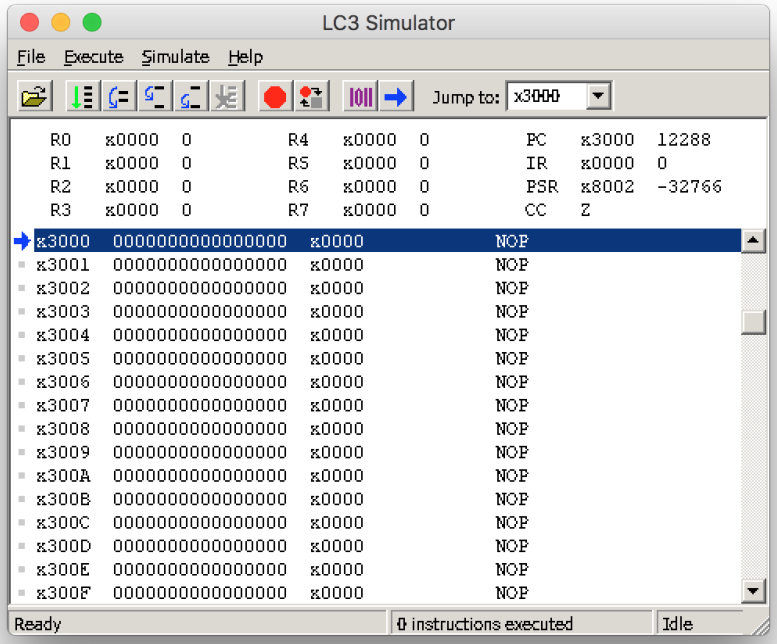
Write-up — Answer the questions throughout this assignment. Fill out your answers *at the end* of this assignment, where the questions have been copied with space for answers.

Handin — `testFile.txt`
GitHub Classroom Link: `https://classroom.github.com/a/edPeADgC`

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: Touring the LC-3 Simulator

Take a quick look around the LC-3 simulator, which you installed in the previous assignment. It probably looks something like this (shown below is the Windows simulator running on macOS through Wine):



(This assignment will refer often to features of the Windows simulator. These features can all be found in the other simulators, but they may be displayed or activated differently.)

Notice that this window displays the contents of the eight registers as well as those of the PC, the IR, and the condition codes, along with the PSR, which we'll talk about later in the course.

The simulator also allows you to scroll through the contents of memory. The addresses of each location are shown in the leftmost column, followed by the value contained at that location.

Each value in memory is displayed in three ways: in binary, in hexadecimal, and as an instruction. Recall that instructions are data, which means that any data stored in memory could potentially be interpreted as instructions. Not all data is valid instructions, however — "`NOP`" is displayed for values that are (N)ot (OP)erations.

Finally, the simulator includes buttons for running, halting, breaking, and stepping through sequences of instructions. If you have used IDE debuggers in the past, you might be familiar with these operations.

1. What is the LC-3's Program Counter set to by default?
2. What is the value at memory location `0x0023` *in binary*?
3. What is the value at memory location `0xFFFE` *in hexadecimal*?
4. What are the binary opcode *and* the name of the instruction at memory location `0x04E6`?
   [*Hint:* Most simulators have a "Jump to" field, allowing you to quickly view a location in memory.]

Part 2: Running LC-3 Instructions

By double-clicking on registers or memory locations, you can change their values.

1. Set the value of `R0` to decimal `#100` and the value of `R1` to hexadecimal `0x23`.
   (a) What is the value of `R0` *in hexadecimal*?
   (b) What is the value of `R1` *in decimal*?

Notice that, by default, there's a blue arrow next to memory location `0x3000`. This arrow indicates the current position of the Program Counter (and you can double check that it matches the displayed value of the PC).

Change the value of memory location `0x3000` to be an `ADD` instruction that adds the values of `R0` and `R1`, placing their sum into `R2`. The easiest way to do this is to write out the encoded instruction in binary, translate that into hexadecimal, then type that into the simulator.

Recall that during the Fetch phase of the instruction cycle, the PC is always incremented. This means that, unless told otherwise, the LC-3 will simply run through memory, one location after the other, executing each value as though it were an instruction until it either reaches the end of memory or encounters some other fatal error.

By double clicking on the gray square next to each memory address, you can set break points. Set a break point on location `0x3001`. This ensures that the LC-3 will halt once execution reaches that location (i.e., after just the one `ADD` instruction has run).

Now, run the simulator by clicking the green down arrow button. If all goes well, the LC-3 should halt with the PC pointing to `0x3001`. Congratulations! You've just told a computer to perform an operation by inputting an instruction directly into memory.

2. Having set the values of `R0` and `R1` to `#100` and `0x23`, respectively, and having run an `ADD R2, R0, R1`, what is the value of `R2` *in hexadecimal*?

[*Note:* A significant portion of the work you will do in this course will involve programming the LC-3. If you are unsure about *any* of the above steps, whether or not you have done them correctly, or why they work, please do not hesitate to ask.]

Part 3: "Add", "And", and "Not"

The LC-3 only has three computational instructions: `ADD`, `AND`, and `NOT`. However, we can perform any computation by building off of those three instructions.

(In fact, your CPE friends may know that we really only need two instructions — addition can be performed, albeit quite tediously, using logical operations, all of which can be reduced to either "or" and "not" or "and" and "not". But I digress.)

For example, we can perform subtraction by adding a negative number. As it turns out, a two's complement binary number can be converted between positive and negative by first negating it, then adding one.

The following instructions compute `R2 = R0 - R1`:

```
1 NOT R2, R1
2 ADD R2, R2, #1
3 ADD R2, R0, R2
```

Translated into binary, that's:

```
1001 010 001 111111
0001 010 010 1 00001
0001 010 000 0 00 010
```

... and in hexadecimal:

```
0x947F
0x14A1
0x1402
```

Let's break that down:

- First, the value of `R1` is negated, and the result placed into `R2`. We're using `R2` as a sort of temporary variable here — we don't want to mutate the value of `R1`.
- We then add an immediate decimal 1, denoted `#1`, to the value of `R2`, storing the sum back into `R2`. This means that `R2` now contains *negative* `R1`.
- Then we can complete the computation by adding `R0` to `R2`, storing the result in `R2`.

And, as you did earlier in Part 2, you should always test solutions such as this by running them in an LC-3 simulator.

1. For each of the following problems, provide a solution *in hexadecimal*.
   - Your solutions should work no matter what values are in the registers — do not assume that a register's value is zero.
   - Test your solutions thoroughly! Test them with all zeroes. Test them with all ones. Test them with powers of two. Test them with negative numbers. Test them with positive numbers.
     [*Hint:* You can change the value of the PC just like you can the values of the registers; you don't have to restart the simulator for every test.]

   (a) Using just *one* instruction, copy the value of `R0` into `R1`.

   (b) Using just *one* instruction, clear `R0` (set all of its bits to zero).

   (c) Using just *one* instruction, clear bit 2 of `R0`, but leave the rest of the bits unchanged.
   For example, if `R0` contains `1111 1111 1111 1111` (`0xFFFF`), then your instruction should change its contents to `1111 1111 1111 1011` (`0xFFFB`).

Part 4: Set Up GitHub Classroom

GitHub is an online host for a separate piece of software named simply "git". Git tracks changes you make to files inside a directory so that you can easily undo changes if necessary while working on a project.

GitHub Classroom allows your instructors to accept electronic submissions through GitHub. For this part of this assignment, you'll make sure that you can submit a file through GitHub Classroom.

You might be familiar with git and GitHub from previous classes or projects. If not, the following links might be helpful:

- `https://help.github.com/articles/set-up-git/`
- `https://help.github.com/articles/cloning-a-repository/`
- `https://help.github.com/articles/managing-files-using-the-command-line/`
- `https://help.github.com/categories/managing-remotes/`
- `https://services.github.com/on-demand/downloads/github-git-cheat-sheet/`

1. Enter your Cal Poly and GitHub usernames in this form: `https://goo.gl/jTK3Bx`. This allows us to determine who should get credit for your electronically submitted assignments.
2. Accept the GitHub Classroom assignment: `https://classroom.github.com/a/edPeADgC`. This will create a new empty repository for you on GitHub.
3. Clone the repository and, within it, create a blank file named `testFile.txt`.
4. Add your file to git, commit your changes, and push them to GitHub. Refresh your GitHub Classroom repository and ensure that your test file shows up.

Part 1: Touring the LC-3 Simulator

1. What is the LC-3's Program Counter set to by default?

2. What is the value at memory location `0x0023` *in binary*?

3. What is the value at memory location `0xFFFE` *in hexadecimal*?

4. What are the binary opcode *and* the name of the instruction at memory location `0x04E6`?
   [*Hint:* Most simulators have a "Jump to" field, allowing you to quickly view a location in memory.]

Part 2: Running LC-3 Instructions

1. Set the value of `R0` to decimal `#100` and the value of `R1` to hexadecimal `0x23`.
   (a) What is the value of `R0` *in hexadecimal*?

   (b) What is the value of `R1` *in decimal*?

2. Having set the values of `R0` and `R1` to `#100` and `0x23`, respectively, and having run an `ADD R2, R0, R1`, what is the value of `R2` *in hexadecimal*?

Part 3: "Add", "And", and "Not"

1. For each of the following problems, provide a solution *in hexadecimal*.
   - Your solutions should work no matter what values are in the registers — do not assume that a register's value is zero.
   - Test your solutions thoroughly! Test them with all zeroes. Test them with all ones. Test them with powers of two. Test them with negative numbers. Test them with positive numbers.

     [*Hint:* You can change the value of the PC just like you can the values of the registers; you don't have to restart the simulator for every test.]

   (a) Using just *one* instruction, copy the value of `R0` into `R1`.

   (b) Using just *one* instruction, clear `R0` (set all of its bits to zero).

   (c) Using just *one* instruction, clear bit 2 of `R0`, but leave the rest of the bits unchanged.

   For example, if `R0` contains `1111 1111 1111 1111` (`0xFFFF`), then your instruction should change its contents to `1111 1111 1111 1011` (`0xFFFB`).