SYSTEMS I — LAB 6
mini-k assembly code

We move from machine-code programming to assembly programming, writing a program's sequence of instructions as text and then using an *assembler* to translate, automatically, that text into the appropriate sequence of machine-code instructions. Moreover, we are going to advance from programs that progress from one instruction to the next in a linear sequence to programs that may *branch*, using instructions that *conditionally* use or re-use certain sequences of instructions by jumping forward or backward in the sequence of instructions.

1 Using the assembler

For the previous assignment, you hand-wrote your instructions in *assembly code*—the text-based, human-readable version—and then manually converted each instruction into the corresponding *machine code*—the binary encoding used by the processor. You entered the machine code into an *image file* with the `hexedit` tool, and then ran the *mini-k processor simulator* (a.k.a., `k-simulator`) with that image file, thus loading that machine code into RAM before the processor began its *fetch-decode-execute cycle*.

We will introduce a new tool for this lab, making the programming task a simpler one, and thus allowing you to approach more complex programming problems. The *assembler* translates assembly code into machine code automatically, transforming the text-based expression of each instruction into the binary encoding used by the processor. Our *mini-k* assembler is a handy tool that will help you perform the assignments provided below, in Section 2.

1.1 Additional capabilities on an assembler

The primary task of an assembler is to translate text-based instructions into machine code. However, it also provides two additional capabilities to ease the programming task. Specifically:

1. **Pseudo-instructions:** The assembler now recognizes two *pseudo-instructions* that it translates into some number of real instructions that carry out the equivalent work. The `SA` pseudo-instruction allows you to specify the full word to be loaded into the accumulator; it generates a pair of `SUA/SLA` instructions that load that word into the accumulator in two steps.

   Similarly, the new `LLAC` pseudo-instruction—*Load Label address into the ACumulator*—allows you to specify a label whose corresponding loading address is loaded into the accumulator. This pseudo-instruction also results in an `SUA/SLA` instruction pair to set the accumulator in two steps.

   Examples of how to use these instructions are shown in Section 1.2.

2. **Labels:** Any *statement* (that is, any *instruction* or *constant*) can be preceeded by a *label*—a name that can serve as a stand-in for the *loading address* (the location in main memory) of a statement. That address to which that label corresponds can then be loaded into the *accumulator* using the `LLAC` psuedo-instruction described above. That value can then be used with a `LOAD`, `STOR`, `BRIS`, or `BRIC` instruction (for example).

1.2 An example

To get started with using the assembler, you must first write some assembly code. Here, we will walk through a small example of writing a small program in assembly, using the assembler to convert it to machine code, and then running it with the simulator.

1. **Login to `remus/romulus`:** As with Lab 5, begin by logging into `remus/remus`—the college's UNIX servers. Open a terminal window to obtain a shell prompt.

2. **Create a directory:** Make a directory for this project and change into it. Specifically:

   ```
   $ mkdir lab-6
   $ cd lab-6
   ```

3. **Copy a sample assembly program:** From the public directory for this assignment, copy a sample assembly program and open it in `emacs` for examination:

   ```
   $ cp ~sfkaplan/public/COSC-161/lab-6/add-two-from-memory.asm .
   $ emacs add-two-from-memory.asm &
   ```

   You will notice that it contains a program similar to the one from Lab-5, loading two values from memory and summing them, and then storing them at address `0xFF`. Notice also that the code has many comments, some on their own lines, some trailing the lines of code themselves, always delineated by the semicolon. Finally, notice that you may specify both instructions and then constants. Determining the address of the constants was achieved in this program by counting the instructions, one byte each.

4. **Assemble the program:** Use the assembler to turn this assembly code into an executable image file:

```
$ k-assembler add-two-from-memory.asm add-two-from-memory.img
Attempting to parse the source assembly code:
[0x??]:         LLAC    x
[0x??]:         COPY    %rD
[0x??]:         LOAD    %rA     %rD
[0x??]:         LLAC    y
[0x??]:         COPY    %rD
[0x??]:         LOAD    %rB     %rD
[0x??]:         ADD     %rA     %rB
[0x??]:         COPY    %rC
[0x??]:         SA      0xff
[0x??]:         COPY    %rD
[0x??]:         STOR    %rC     %rD
[0x??]:         LOAD    %rD     %rD
[0x??]:         HALT
[0x??]:         0x03
[0x??]:         0x15
Done parsing.

Assigning load addresses to labels...done.

Showing the parsed instructions and constants:
[0x00]:         SUA     0x01
[0x01]:         SLA     0x00
[0x02]:         COPY    %rD
[0x03]:         LOAD    %rA     %rD
[0x04]:         SUA     0x01
[0x05]:         SLA     0x01
[0x06]:         COPY    %rD
[0x07]:         LOAD    %rB     %rD
[0x08]:         ADD     %rA     %rB
[0x09]:         COPY    %rC
[0x0a]:         SUA     0x0f
[0x0b]:         SLA     0x0f
[0x0c]:         COPY    %rD
[0x0d]:         STOR    %rC     %rD
[0x0e]:         LOAD    %rD     %rD
[0x0f]:         HALT
[0x10]:         0x03
[0x11]:         0x15
```

The first repetition of the code is the result of *parsing*—attempting to read each *statement*.

If a statement is printed here, then it was read by the assembler without error if an error message appears or the assembler reports that it is `done parsing`, then it failed to parse the first statement **not** shown in this output. Notice that, on the leftmost side of each line, the *loading address*—the location in main memory into which each statement will be copied—appears as `[0x??]`. This is the assembler's way of indicating that the loading addresses have not yet been calculated.

Having read the entire assembly code program, the assembler then performs a few critical tasks. First, it calculates the loading addreses of statement; second, assigns the correct address to each label; third and last, it creates the real instructions for each psuedo-instruction. For example, each `SA` psuedo-instruction is converted into a pair of real `SUA` and `SLA` instructions.

After assigning loading addresses and create real instructions for the psuedo-instructions, the assembler displays a second repetition of the code. You should therefore see the sequence of loading addresses at the leftmost portion of each statement (e.g., `[0x0b]`). Also, each psuedo-instruction will be replaced with its corresponding real instructions. Finally, the assembler will emit the machine code into the image file specified on the command line.

5. **Run the program on the simulator:** Use the executable image, with all of its machine code and constants as produced by the assembler, on the simulator:

```
$ k-simulator add-two-from-memory.img
```

You will see, in a form that is now familiar, the result of running this program on the simulated processor.

2 Your assignment

2.1 Summing an array

Write a program for which there are a set of constants that immediately follow the instructions. Specifically, imagine an *array* of constants, defined as:

- **A single value**–let us call it n—that occupies the first memory location after the final machine code instruction to be loaded into main memory, that specifies the *number of values to follow*, also known as the *length* of the array, and
- **The n values themselves** that occupy the next n memory locations. These values could be any arbitrary two's complement integers; you know only that there will be n of them.

Your program should **sum the array of n values** and **store the sum** at main memory address `0xff` before the program completes.

2.2 Producing the Fibonacci sequence

Once again, assume that your program will have a single constant, n, that is stored at the first main memory address after the last machine code instruction instruction. Assume that this n specifies the n^{th} Fibonacci number, defined as:

$$F(n) = \begin{cases} n & \text{if } n < 2 \\ F(n-1) + F(n-2) & \text{if } n \geq 2 \end{cases}$$

Write a program that loads n from this location. It should then, starting at address `0x80`, place $F(0)$, followed by $F(1)$ at address `0x81`, $F(2)$ at `0x82`, up to $F(n)$ at address `0x80 + n`. You may assume that $n \geq 2$.

3 How to submit your work

We will be using the `cs161-submit` command to turn in programming work. Specifically, you should submit your completed `sum.asm` and `fib.asm`, like so:

```
cs161-submit lab-6 count.asm fib.asm
```

This assignment is due on Nov 1/2, at the start of lab