SYSTEMS I — LAB 5
An introduction to the *mini-k* ISA

In today's lab, we will do some machine code programming. In particular, we're going to use a program that simulates the *mini-k* processor—an artificially created CPU for this course. You will write some assembly code, and then you will convert it to machine code by hand, enter that into a file, and the use that file as the contents of RAM on a simulator computer.

Introduction to assembly/machine-code programming

The following steps will lead you through your first assembly language program. Here, the goal is to get used to the tools involved in writing, assembling, testing, and debugging these programs. After you get this pre-written and simple program working, you will then need to write a program of your own.

1. **Login:** In order to use the simulator, you must first logon to `romulus.amherst.edu` or `remus.amherst.edu`, which are UNIX (Linux) systems. To login using *Xming*, software that allows you to login graphically to these servers, follow the Windows *Xming* instructions that describe how to use this software on the Windows machines in Seeley Mudd 014. Notice that this page also describes how to install and use *Xming* on your Windows machine. If you have a Mac, follow the Mac *Xming* instructions.

2. **Make a directory:** When you first login, you will be working in your *home directory*—the UNIX analog of your *My Documents* folder. Within this directory, you should make a *subdirectory* (a folder) for your work for this lab. Specifically, enter the following command to create and then change into that subdirectory:

   ```
   $ mkdir lab-5
   $ cd lab-5
   ```

3. **Create a machine code file:** Use a special command to create a new file into which you will write your machine code (and later load it into the simulator):

   ```
   $ touch just-halt.img
   ```

4. **Write the machine code:** Run a special program called `hexedit` in order to add and change byte values, in hexadecimal,[1] to a machine code file. Within this program, you will be able to type the values held in the file, one hexadecimal nybble at a time, while a text version of the file is shown on the right side of the screen. Notice that for most byte-sized values that you create here, little meaningful text will be created in that space. We really only care of about the binary values, and not much about the text:

[1] See Appendix A for a quick introduction to (or review of) *hexadecimal*, or *base 16*, numeric representation.

```
$ hexedit just-halt.img
```

Within this program create a *one-byte machine code program where the bits are all 0*. By typing `00`, you will create that byte with that value. That opcode is for the HALT instruction, which is all that we want this first program to do.

Then type *F2* to save the change to the file, and then *Control-X* to exit `hexedit`, bringing you back to the shell prompt.

5. **Run the simulator:** Start the *mini-k simulator*—a program that reads and and runs the machine code produced by the assembler, just like a real CPU would:

```
$ k-simulator just-halt.img

WARNING:  Image file ended at byte 0.
          Zero-filling remaining RAM.
DEBUG:    pc = 0x00  0x00  HALT
Halting!
  acc = 0x00  reg[0] = 0x00  reg[1] = 0x00
              reg[2] = 0x00  reg[3] = 0x00
```

We will examine in lab what this output means. The warnings can be ignored; what we care about most begins with each line that begins `DEBUG:`. These lines show you the current pc value, the machine code instruction itself, and then a *disassembly* of that instruction: an interpretation of the *opcode* and the *operands*. The instruction is then *executed* (carried out), and the registers—the *accumulator* and the named registers in the *register file*—are shown.

In short, this is the smallest, simplest example of creating and running a program. The machine code file is loaded as the contents of RAM (with extra zeros for those bytes whose values aren't specified, out to 256 bytes of RAM), the pc is set to 0, and then the processor proceeds with its *fetch-decode-execute cycle* until it is *halted*.

Try the program from class

Now that you see how this works, write the program that we've worked out in class. That is, write a program that will:

1. Assign the constant 3 into register `%rA`.
2. Assign the constant 21 into register `%rB`.
3. Add $3 + 21$.
4. Store the final result into register `%rC`.

Write this program in assembly on paper, and then work out the machine code that follows from it. Use `hexedit` to then create that machine code file and test your program on the simulator.

The following table may be handy:

```
HALT = 0x0,   // Stop processing
COPY = 0x1,   // Copy from the accumulator to a register.
NOT  = 0x2,   // Bitwise inversion
AND  = 0x3,   // Bitwise logical conjuction
OR   = 0x4,   // Bitwise logical disjunction
ADD  = 0x5,   // Arithmetic addition
SUB  = 0x6,   // Arithmetic subtraction
LOAD = 0x7,   // Copy from main memory to a register
STOR = 0x8,   // Copy from a register to main memory
SEQ  = 0x9,   // Set the accumulator if EQual
SLT  = 0xa,   // Set the accumulator if Less Than
SGT  = 0xb,   // Set the accumulator if Greater Than
BRIS = 0xc,   // BRanch If the given register is Set
BRIC = 0xd,   // BRanch If the given register is Clear
SLA  = 0xe,   // Set the Lower nybble of the Accumulator
SUA  = 0xf    // Set the Upper nybble of the Accumulator
```

A tricker challenge, using main memory

Place the values 21 and 3 into your image value so that it will be loaded into RAM. Then use the LOAD instruction to copy these values into registers, SUBtract them, and then use the STOR instruction to move the result into the highest address in RAM (`0xff`).

How to submit your work

We will be using the `cs161-submit` command to turn in programming work. Specifically, you should submit your completed work like so, from within your `lab-5` directory:

```
cs161-submit lab-5 *.img
```

This assignment is due on Thursday/Friday, November 31/December 01, at the start of lab

A A quick tutorial on hexadecimal

Hexadecimal—or *base 16* numeric representation—uses the digits 0 through 9, followed by the letters a through f. Since there are 16 distinct hexadecimal digits, each one may correspond to a unique 4-bit value, or *nybble*, (since there are $2^4 = 16$ possible 4-bit values). We can enumerate this correspondence:

hexadecimal	**binary**	**decimal**
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
a	1010	10
b	1011	11
c	1100	12
d	1101	13
e	1110	14
f	1111	15

This correspondence is useful because it allows us to compactly represent lengthy binary values. For our architecture and its 8-bit word size, each word may be represented by a pair of hexadecimal values. You will see, in using our simulator, that the output is given in hexadecimal for this reason. You will quickly become adept at translating between binary and hexadecimal in your head.