SYSTEMS I — LAB 5
An introduction to *k*-system assembly programming

In today's lab, we will do some assembly programming. In particular, we're going to use a program that simulates the *k*-system CPU—an artificially created system for this course. You're going to get set up with this simulator and its corresponding assembler. Then you will write some assembly code of your own, which you will then test with the assembler and simulator.

Introduction to assembly 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 assembler and 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. **Get the assembly code:** Use the following command to obtain a sample assembly code program, being careful to include the tilde (˜) before my username and the trailing space followed by a period (`.`):

```
$ cp ~sfkaplan/public/cs16/lab-5/add-two-numbers.asm .
```

4. **Modify the assembly code:** Run *Emacs*, a programming text editor, to examine the `add-two-numbers.asm` file. In the following command, be sure to include the trailing ampersand (`&`), causing the text editor to run in the *background*—that is, to run while allowing you to enter more commands:

```
$ emacs add-two-numbers.asm &
```

We will discuss, in the lab, what the components of this file are and how to interpret them. Furthermore, you should read this documentation/tutorial on using Emacs. Get familiar enough to change the constants assigned into `%G0` and `%G1` in the first two instructions to values of your choice.

5. **Assemble:** Use the *k*-system *assembler* to translate the text-based assembly code into binary *machine code*, like so:

```
$ k-assembler add-two-numbers.asm
DEBUG [1]: @0x00000000: COPY  %G0  1[0x00000001]
DEBUG [1]: @0x00000010: COPY  %G1  16[0x00000010]
DEBUG [1]: @0x00000020: ADD   %G2  %G0  %G1
DEBUG [1]: @0x00000030: JUMP  @+end[0x00000000]
```

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

```
$ k-simulator add-two-numbers.vmx
DEBUG [1]: Bus.addDevice(): newDevice =
  [ type = RAM, ( 0x00001000, 0x00005000 ) ]
DEBUG [1]: Bus.addDevice(): newDevice =
  [ type = ROM, ( 0x00007000, 0x00007040 ) ]
[pc = 0x00007000]:
```

We will examine in class what this output means. Most importantly, the simulator leaves you at a prompt that shows the current value of the *program counter (PC)*. You will notice that this number (and some others shown by the assembler and simulator) are shown in *hexadecimal*, also known as *base 16*. We will discuss why that representation has been chosen, and how to read it. Specifically, the prefix on a number to indicate that it is in hexadecimal is $0x$. Moreover, a hexidecimal digit is one of 16 values, from 0 to 9, and then from a to f. Thus, here is a table that shows some values in decimal, binary, and hexidecimal:

decimal	**binary**	**hexidecimal**	**decimal**	**binary**	**hexidecimal**
0	0000	0	16	10000	10
1	0001	1	17	10001	11
2	0010	2	18	10010	12
3	0011	3	19	10011	13
4	0100	4	20	10100	14
5	0101	5	21	10101	15
6	0110	6	22	10110	16
7	0111	7	23	10111	17
8	1000	8	24	11000	18
9	1001	9	25	11001	19
10	1010	a	26	11010	$1a$
11	1011	b	27	11011	$1b$
12	1100	c	28	11100	$1c$
13	1101	d	29	11101	$1d$
14	1110	e	30	11110	$1e$
15	1111	f	31	11111	$1f$
			32	100000	20

7. **Take a step:** Make the simulator execute a single instruction—take a *step*—like so:

```
[pc = 0x00007000]: step 1
DEBUG [0]: [@0x00007000]
  0x 00024800 00000003 00000001 00000000:
  COPY    %0x00000003 0x00000001
[pc = 0x00007010]:
```

   First, the simulated CPU fetched the `COPY` instruction at main memory address `0x7000`, and carried it out by loading the immediate constant `0x1` into register `%3`, also known to us as `%G0`. Second, the simulator *incremented the PC* so that it points to the next machine code instruction in main memory at address `0x7010`. The simulator is now paused in that state, ready to execute the next instruction, waiting for your command.

8. **Verify the step:** Just to be sure that this instruction was performed correctly, we can check the contents of register `%G0`, a.k.a., `%3`. Specifically, the following command will show the contents of that register:

```
[pc = 0x00007010]: showregister %3
%3 = 0x00000001
```

9. **Take a few more steps:** Make the simulator complete the program, reaching the infinite loop that ends the program:

```
[pc = 0x00007010]: step 4
DEBUG [0]: [@0x00007010]
  0x 00024800 00000004 00000010 00000000:
  COPY    %0x00000004 0x00000010
DEBUG [0]: [@0x00007020]
  0x 00064440 00000005 00000003 00000004:
  ADD     %0x00000005 %0x00000003 %0x00000004
DEBUG [0]: [@0x00007030]
  0x 00112000 00000000 00000000 00000000:
  JUMP   ->@+0x00000000
DEBUG [0]: [@0x00007030]
  0x 00112000 00000000 00000000 00000000:
  JUMP   ->@+0x00000000
[pc = 0x00007030]:
```

10. **Verify the result:** The final result of the program should be contained in register `%G2`, or `%5`. So:

```
[pc = 0x00007030]: showregister %5
%5 = 0x00000011
[pc = 0x00007030]:
```

Notice that the output of the `showregister` command is a hexadecimal value. To convert that value into decimal, you can use your brain along with paper and pen, or you can use Google Calculator.

If you want to see some of the other capabilities of the simulator, simply enter the command `help` to see a command listing. We will discuss, in lab and as these assignments progress, how to use most of them.

Make a new, modified program

You have seen what an assembly code program looks like, and you have seen how to assemble and execute the program using the simulator. Now write your own small calculation program. Specifically, start by **copying a skeleton assembly code file**:

```
$ cp ~sfkaplan/public/cs16/lab-5/formula.asm .
```

Open this new file, `formula.asm`, with *Emacs*:

```
$ emacs formula.asm &
```

Specifically, after the `__start` label, insert code to do what the comments suggest. Specifically, write instructions that will load the given constants into registers, and then write instructions that carry out the given (simple) formula.

Once written, **assemble** your program and then **run it** in the simulator. If you find errors, edit the assembly code, run the assembler again, and then run the simulator again. Repeat this *edit-assemble-simulate cycle* until the program operates correctly.

Make a third, more challenging program

Copy one more skeleton assembly code file:

```
$ cp ~sfkaplan/public/cs16/lab-5/find-max.asm .
```

One this file with *Emacs*. In it, you will find a comment that explains the goal of this program. Specifically, you must write instructions that examine each of the $length$ values listing following the label $array$, and find the maximum of those values. That maximum should be left in the main memory space labeled max.

In order to write this program, we will have to discuss main memory locations, how arrays of integers are arranged and used, and how *pointers*—main memory addresses—can be used to move through arrays. We will discuss these concepts during the lab and during lectures.

How to submit your work

We will be using the `cs16-submit` command to turn in programming work. Specifically, you should submit your completed `formula.asm` and `find-max.asm`, like so:

```
cs16-submit lab-5 formula.asm find-max.asm
```

This assignment is due on Monday, November 9, at 11:00 am