MP1. Printing a Histogram

Due Date: Wed, Sept. 3, 2014 at 10:00 p.m.

In this first programming assignment, you will extend code that we develop in class to compute a histogram of letters and non-letters in a string. Your final program will print the resulting histogram in hexadecimal to the monitor. **Please read the entire document, including the grading rubric, before you begin programming.**

The Routine

The routine for each week of our class will be the same. You will receive the materials for the week's assignment no later than Monday. Before Friday, you should at a minimum read through the assignment and the code given to you. We also encourage you to try to do some parts of the assignment before Friday.

On Friday, we will meet together in lab for programming studio. We will have a specific piece of the assignment that we work on together as a class. If you have already completed this piece, you can help other people understand how it should be done. If you have run into problems, you will have a chance to ask questions. If you have yet to start the assignment, you will not be able to make much use of the opportunity. **Get in the habit of managing the time that you allot to your assignments; you will need this skill to succeed as an engineer.**

You then have until the following Wednesday at 10 p.m. to complete the remainder of the assignment, which will make use of the part that we developed in programming studio.

As mentioned in the overview for the course, **all work other than the part developed in programming studio must be done by your own group.**

The Pieces

This week, you are given the histogram code that we developed in class. You should read through it to make sure that you understand how it works.

In programming studio, we will develop code to print a value stored in a register as a hexadecimal number to the monitor, which will involve turning each group of four bits into a digit, calculating the corresponding ASCII character, and printing that character to the monitor.

The remaining part of the assignment requires that you use the hexadecimal printing code to print the contents of the histogram to the monitor. For the string shown below

```
This is a test of the counting frequency code. AbCd...WxYz.
```

the output produced by a correct program appears below the string.

```
@ 000F
A 0002
B 0001
C 0004
D 0002
E 0005
F 0002
G 0001
H 0002
I 0003
J 0000
K 0000
L 0000
M 0000
N 0003
O 0003
P 0000
Q 0001
R 0001
S 0003
T 0005
U 0002
V 0000
W 0001
X 0001
Y 0002
Z 0001
```

Details

As a first step, we suggest that you peruse the copy of the code included with this specification and make sure that you understand how it works.

You can then choose between two pieces. First, you can write the code that manages printing all of the histogram bin labels, spaces, and newlines, and handles the loop control over bins. You will need to write this code yourself at some point. Second, you can start writing code to print a hexadecimal number from a register. We will develop that code as a class during programming studio on Friday, but you are welcome to do it yourself ahead of time, if you prefer.

When you are contemplating how to write one of the pieces, start by systematically decomposing the problem to the level of LC-3 instructions. You need not turn in any flow charts, but we strongly advise you not to try to write the code by simply sitting down at the computer and starting.

Once you are ready to start writing code (next paragraph explains how to obtain a copy), first write a register table so as to have it in plain sight while you work. You may also want a copy on a piece of paper. Then sketch out the flow of the program using comments. Then write the instructions.

We've added a couple of extra lines of code at the bottom for a simple test (the one given as an example above). When you have debugged a little, the section on testing (on the next page) reminds you of how you can make use of a few scripted tests that we have provided.

Checking Out a Copy: A copy of the starting code has been placed in your Subversion repository. To check out a copy, *cd* to the directory in which you want your copy stored, then type:

```
svn co https://subversion.ews.illinois.edu/svn/fa14-ece198kl/[netid]/MP1
```

Replace "netid" with your NetID. Recall that the Subversion ("svn") checkout command ("co") makes a copy of the files stored in your repository in the directory in which you execute the command. The copy will be called "MP1" – type *cd MP1* to enter it.

If for some reason we have missed you - possibly if you were not in the roster when we imported the copies to student repositories - you will need to get your own copy and place it into your repository. **Note: if your checkout worked, you can skip these steps.**

1. Check out a copy from the class' shared directory by typing:

```
svn co https://subversion.ews.illinois.edu/svn/fa14-ece198kl/_shared/MP1
```

2. Change directory into the directory that you have created: *cd MP1*
3. From within the *MP1* directory, import the directory into your subversion account by typing the following **all on one line**, using your own NetID:

```
svn import -m "Create program 1 directory."
https://subversion.ews.illinois.edu/svn/fa14-ece198kl/[netid]/MP1
```

4. Go back out of the MP*1* directory by typing: *cd..*
5. Remove the copy associated with the class' repository: *rm -rf MP1*
6. Check out a copy of your MP*1* directory as above (before these numerical steps).

Use your working copy of the lab to develop the code. Commit changes as you like, and make sure that you do a final commit once you have gotten everything working. **Commit a working copy and make a note of the version number**

With your assembly code, you must include a paragraph describing your approach as well as a table of registers and their contents. Avoid using R7 if possible, as any TRAP instructions (such as the OUT traps you will need to print to the monitor) will overwrite its contents and may confuse you.

Specifics:

- Your program must be called *prog1.asm* – we will NOT grade files with any other name
- Your code must begin at memory location x3000 (just don't change the code you're given in that sense).
- The last instruction executed by your program must be a HALT (TRAP x25).
- You must not corrupt the histogram created by the code given to you. You should not change that part of the code.
- Your output must match the desired format exactly, as shown in this specification and in the test files provided.
- Your program must use an iteration over bins in the histogram.
- You may not make assumptions about the initial contents of any register (before the histogram code executes, that is).
- You may assume that the string is valid.
- You may use any registers, but we recommend that you avoid using R7.

Tools: Use a text editor on a Linux machine (*vi*, *emacs*, or *pico*, for example) to write your program for this lab. Use the LC-3 simulator in order to execute and test the program. Your code must work on the EWS lab machines to receive credit, so make sure to test it on one of those machines before handing it in.

Testing: You should test your program thoroughly before handing in your solution. Remember, when testing your program, *you* need to set the relevant memory contents appropriately in the simulator. You may want to use a separate ASM file to specify test inputs, as discussed below. When we grade your lab, we will initialize the memory for you. Developing a good testing methodology is essential for being a good programmer. For this assignment, you should run your program multiple times for different functions and inputs and double check the output by hand.

We have also given you three sample test inputs, *testone.asm, testtwo.asm,* and *testthree.asm*. You will need to assemble these files. For each test, we have provided a script file to execute your program with the test input --*runtestone, runtesttwo,* and *runtestthree* – and a correct version of the output: *testoneout, testtwoout,* and *testthreeout*. Look at the script: load the sample input, then load your program. The "reset" command/button will not work, since you are using more than one program. Set the PC by hand if necessary (for example, *r pc 3000*), or re-load both files (input and your program) whenever you need to restart a debug effort.

First you must debug -don't assume that you can simply run the script to debug your code.

Once you think that your code is working, you can execute the script for the first test by typing the following:

```
lc3sim -s runtestone > myoutone
```

This command executes the LC-3 simulator on the script file and saves the output to the file *myoutone*. If the command does not return, your program is stuck in an infinite loop (press CTRL-C). Otherwise, you can compare your program's output with our program's by typing: *diff myoutone testoneout*

Note that your final register values need not match those of the output that we provide, but all other outputs must match exactly.
To test the program on one of your own inputs. First assemble your program using-

```
lc3as prog1.asm
```

Next you can open the graphical simulator by typing **lc3sim-tk** on the command line or run the command line simulator using the **lc3sim** command. Open one of the lc3 script files (runtestone, runtesttwo, runtestthree) and their corresponding test files (testone.asm, testtwo.asm,testthree.asm) to learn how to create and run your own test cases.

If you implement challenges that change the output, you can find files with which to compare your modified output in the *challenge-outputs* subdirectory.

Handin: Include the introductory paragraph in the introParagraph.txt file (**You will not receive credit for the introductory paragraph if it is not in this file!**) and write down your partners' names in the partners.txt file. Be sure to commit the final version of your program before the deadline for the assignment. (From inside your *prog1* directory, type *svn commit -m "My submission is done."*)

Grading Rubric:

Functionality (50%)

- 15% - program prints one line for each bin
- 15% - program labels each bin correctly in output
- 10% - program prints number of occurrences correctly for each bin
- 10% - program spaces line correctly for each bin

Style (20%)

- 15% - program uses a single iteration to print lines of output
- 5% - program separates code from data (put your new data with the existing data)

Comments, clarity, and write-up (30%)

- 5% - introductory paragraph clearly explaining program's purpose and approach used (introParagraph.txt)
- 10% - code includes table of registers for your new section that explains their meaning and contents as used by the code
- 15% - code is clear and well-commented (every line)

Note that correct output (and the points awarded for it) also depends on not somehow mangling the contents of the histogram.