Reading Strings

Programming Lab #3
CEG 3310/5310: Computer Organization

PURPOSE

The average computer scientist programs in assembly very rarely. However, some familiarity with assembly level programming is necessary to develop the fundamental understanding of how high-level programs get executed (which itself is necessary to understand high-end performance issues). In this lab, students will gain familiarity with basic low-level programming.

ASSIGNMENT

In this lab you will construct an assembly routine that puts the ASCII representation of an input string in memory at a specified location. The LC-3 machine language provides several useful TRAP calls for I/O (Patt 2e Appendix A, p 541). For example, GETC (TRAP x20) gets one character from the keyboard and places its ASCII code in R0. OUT (TRAP x21) performs the opposite function – it takes the contents of R0 and outputs the specified character (using ASCII encoding) to the display.

The LC-3 provides two *output* TRAPS for strings: PUTS (TRAP x22) and PUTSP (TRAP x24). Both of these expect (in R0) a starting address for a string, and output the encoded string to the display. Your task is to produce two programs that provide the "opposite" function of PUTS and PUTSP – that is, they take a string as *input* and place the string into memory at a location specified in R0. You will create two separate stand-alone programs: one for GETS and one for GETSP. The programs do **not** need to be implemented as subroutines or TRAP calls.

Using the LC-3 simulator, you will construct an assembly-level program that prompts the user for a string (the end of the string will be delimited by the character x0A, the ASCII character "Enter"). You will store the ASCII string representation of this input (including a terminating NULL character) in memory. The "Enter" character should **not** be considered part of the string and thus should **not be stored**. For the purposes of demonstration, I would like you to store the string at address x3100 (add the appropriate code to your program) and show that you have successfully stored the string by printing it back to the display using PUTS or PUTSP. Your program will be located in memory at location x3000.

The following is an example output for both GETS.asm and GETSP.asm:

```
Enter string to echo: Go Raiders!
Go Raiders!
----- Halting the processor -----
```

Part 1:

Use the Pilot dropbox to turn in your **well-commented** assembly code implementing the GETS operation. Your program should be named GETS.ASM. Note: Well-commented means "easy to follow high-level flow and design decisions". You do not have to comment every line of code, but you should have high-level comments for each 3-10 lines that represent a functional block of code or high-level idea. You will also need to demonstrate your working program for the TA in the lab session on or before the due date. *Your program should run correctly when the machine is randomized before loading your program!*

Part 2:

Use the Pilot dropbox to turn in your well-commented assembly code for the GETSP operation. Your program should be named GETSP.ASM. You will also need to demonstrate your working program for the TA in the lab session on or before the due date. *Your program should run correctly when the machine is randomized before loading your program, and for odd- or even-length strings*.

NOTE ABOUT ONLINE LC3 SIMULATOR

The online simulator at (https://wchargin.com/lc3web/) does not have a working PUTSP trap service routine. If you want to test your code please verify the locations in memory visually. You should still have a fully functional GETSP subroutine. For example, if I type in the string "abcde" using my GETSP subroutine, the following memory addresses should look like this:

Address	Data
x3100	x6261
x3101	x6463
x3102	x0065
x3103	x0000

GRADING

A key element of assembly language programming is documenting your code in such a way that your intended algorithm is easy to understand. As such, a significant portion of the points for this lab are based upon your comments/documentation.

Points will be assigned out of a maximum of 50 as follows:

20 points: Project correctly implements GETS.

15 points: Project correctly implements GETSP.

15 points: In-line documentation (comments) and programming style (no spaghetti code).