

CMPE 12L Lab 4 - Fall 2014

— — .-. / -.-. — -.. .

Prof. Matthew Guthaus

Due: November 14, 2014 5pm

75 Points (20 Report, 55 Work)

Corrections

- 11/6/14: Corrected valid inputs to be decoded to include x41-x46 (A to F).
- 11/7/14: Updated text to say that you may omit R6 PUSH/POP in subroutine.

Prerequisites

- Read through this **entire** lab assignment.
- Read about Morse Code on Wikipedia: http://en.wikipedia.org/wiki/Morse_code

Tutor/TA Review

- Overview of I/O in LC3
- What's required

Overview

Morse code is a method of transmitting text using a series of on and off signals. These signals can be light, an audio tone, or even vibrations. The information is contained in the duration of the on signal and the pattern of on/off signals. Morse is convenient because it can transfer or receive a message with very little equipment and a bit of skill. Amateur radio (“Ham”) tests used to require Morse code since it requires very little radio frequency bandwidth. However, computers are able to easily convert Morse code so it is no longer a requirement for these licenses. In this assignment, you will implement a Morse code converter. It will take a message in either English text or in Morse code and convert it to the other format.

Morse code is a representation of the standard English alphabet via a sequence of dots and dashes. The basics of Morse code relies on a “beat” during which a transmitter can either be on or off. A dot is one beat while a dash is equal to 3 dots in time. One beat is in between each

dot and dash. The space between two characters is 3 beats and the space between two words is 7 beats. For example, “-.-” would be the character ‘D’ according to Figure 1. You can find more examples on the Wikipedia link in the preliminary reading.

We are going to use a simplified, byte-aligned version of Morse code to simplify this lab. First, a dot is a 1 and a dash is a 0. Second, you can assume that each character in Morse code is a single byte and is left aligned. In other words, the most-significant bit (MSB) of the byte is the start of the character. This is actually very convenient, because you can ignore the beats between dots/dashes and characters. For example, the character A in Morse is “.-” which will be represented by the binary “10”. We then pad this value with a single 1 and a bunch of 0’s. The 1 is termed a STOP bit in serial communications, but you will learn about this type of thing in future classes. This would give us 10100000b or xA0 for the character A. Similarly, D will be 01110000 (x70). You must store the data for Morse code as a table of 72 bytes. The first byte is the ASCII character/digit and the second byte is the encoding similar to in Table 1. You can use indirect addressing to retrieve the values of translation in either direction by searching through the table. You may reorder the rows or columns of your table if you like.


ASCII	Morse	Morse	Binary	Hex
A	.-	10	10100000	A0
B	-...	0111	01111000	78
C	-.-.	0101	01011000	58
D	-.-.	011	01110000	70
E	.	1	11000000	C0
F	...-	1101	11011000	D8
G	-.-.	001	00110000	30
H	1111	11111000	F8
I	..	11	11100000	E0
J	.-...	1000	10001000	88
K	-.-.	010	01010000	50
L	.-.-.	1011	10111000	B8
M	--	00	00100000	20
N	-.	01	01100000	60
O	---	000	00010000	10
P	.-.-.	1001	10011000	98
Q	-.-.	0010	00101000	28
R	.-.-.	101	10110000	B0
S	...	111	11110000	F0
T	-	0	01000000	40
U	...-	110	11010000	D0
V	...-	1110	11101000	E8
W	.-.-.	100	10010000	90
X	-.-.-	0110	01101000	68
Y	-.--	0100	01001000	48
Z	..--	0011	00111000	38
1	.-...	10000	10000100	84
2	..-.-	11000	11000100	C4
3	...--	11100	11100100	E4
4-	11110	11110100	F4
5	11111	11111100	FC
6	-...-	01111	01111100	7C
7	-....	00111	00111100	3C
8	---.-	00011	00011100	1C
9	----.	00001	00001100	0C
0	-----	00000	00000100	04

Table 1: Morse code and our simplified Morse encoding

The LC-3 Assembler

You don’t need to write any machine code in this lab as we will rely on the LC-3 Assembler. The LC-3 assembler works in two passes. The first pass constructs a symbol table and the next pass performs the actual conversion to assembly language. The symbol table is a list of symbolic addresses that later get mapped to real addresses. If you used a branch in your last lab, do

you remember having to keep adjusting the offset when the number of instructions in your loop changed? This will automatically do that for you. It basically helps the programmer avoid manually computing addresses.

- Start the LC-3 editor, LC3Edit.exe. You can find it on the PCServer in *F : \ClassFolders\ComputerEngineering\CE12L\LC3*.
- Copy the echo.asm sample code in Figure 1 into the LC3Edit program.
- Compile the code by selecting Translate->Assemble or clicking on the “asm” button (). Save the file in your mounted home directory (*X : *).
- Check for errors. If there were problems assembling, correct the errors now. A properly-assembled file will generate the message in Figure 2. A program with errors will list the errors. Figure 3 shows echo.asm assembled.

Right away, you can see the syntax for a label. START specifies the instructions at the beginning of the program and will map to x3000. If however, a relative instruction uses START, it will automatically compute the offset for you. Similarly, GREETING, USERSTRING, etc. provide an address label for some convenient data in memory.

There are also assembler short cuts for using types of data or compiler directives. “.ORIG” tells the assembler to put the instructions at a specific address in memory. “.STRINGZ” allows you to input an ASCII string that gets converted to binary in memory. “.FILL” lets you initialize memory to a value. What other directives can you find in the text book or on the web?

Assignment

Implement, in LC-3 assembly language, an encode/decode program that meets the following requirements:

Input

Your program should prompt the user for two separate inputs from the keyboard, as follows:

- The prompt: (E)ncode/(D)ecode:
 1. The user will type upper-case E or D.
 2. Your program will accept that character, store it in x3200, and use it, as we shall see momentarily.
- The prompt: Input Message:
 1. The user will input a character or hex string from the keyboard, terminating the message with the ENTER key.
 2. One constraint: Messages must be less than or equal to #30 characters. (Recall: # means the number is decimal.)

```

; CMPE12 - Fall 2014
; echo.asm
;
; This program echoes a user's input.

; The code will begin in memory at the address
; specified by .orig <number>.
.ORIG    x3000

START:
; clear all registers that we may use
AND R0, R0, 0
AND R1, R0, 0
AND R2, R0, 0
AND R3, R0, 0
AND R4, R0, 0

; print out a greeting
LEA R0, GREETING
PUTS

; get a user-entered character (result in R0)
; echo it back right away (otherwise it isn't visible)
GETC
PUTC

; store entered string (otherwise it may be overwritten)
ST R0, USERINPUT

; print out a newline and some other stuff
LEA R0, NEWLINE
PUTS

; print out the user's input
LD R0, USERINPUT
PUTC

; print out the rest of the sentence
LEA R0, USERSTRING
PUTS

; stop the processor
HALT

; data declarations follow below

; strings
GREETING: .STRINGZ "Welcome to the sample program.\nPlease enter a letter: "
USERSTRING: .STRINGZ " <-- is the character you entered.\n"
NEWLINE: .STRINGZ "\n--> ";

; variables
USERINPUT: .FILL 0

; end of code
.END

```

Figure 1: Assembly program to display echo a users input to the screen: echo.asm

```
Starting Pass 1...
```

```
Pass 1 - 0 error(s)
```

```
Starting Pass 2...
```

```
Pass 2 - 0 error(s)
```

Figure 2: The status window for a properly-assembled program will show 0 errors for each of the two passes.

3. Your program will store the message, starting in location x3201. Since the message is restricted to #30 characters, you must reserve locations x3201 to x3220 to store the message. This includes the final ENTER character to denote the end of the message, but this is not encoded/decoded. (Note that #30 = x1E.)

Input/Output

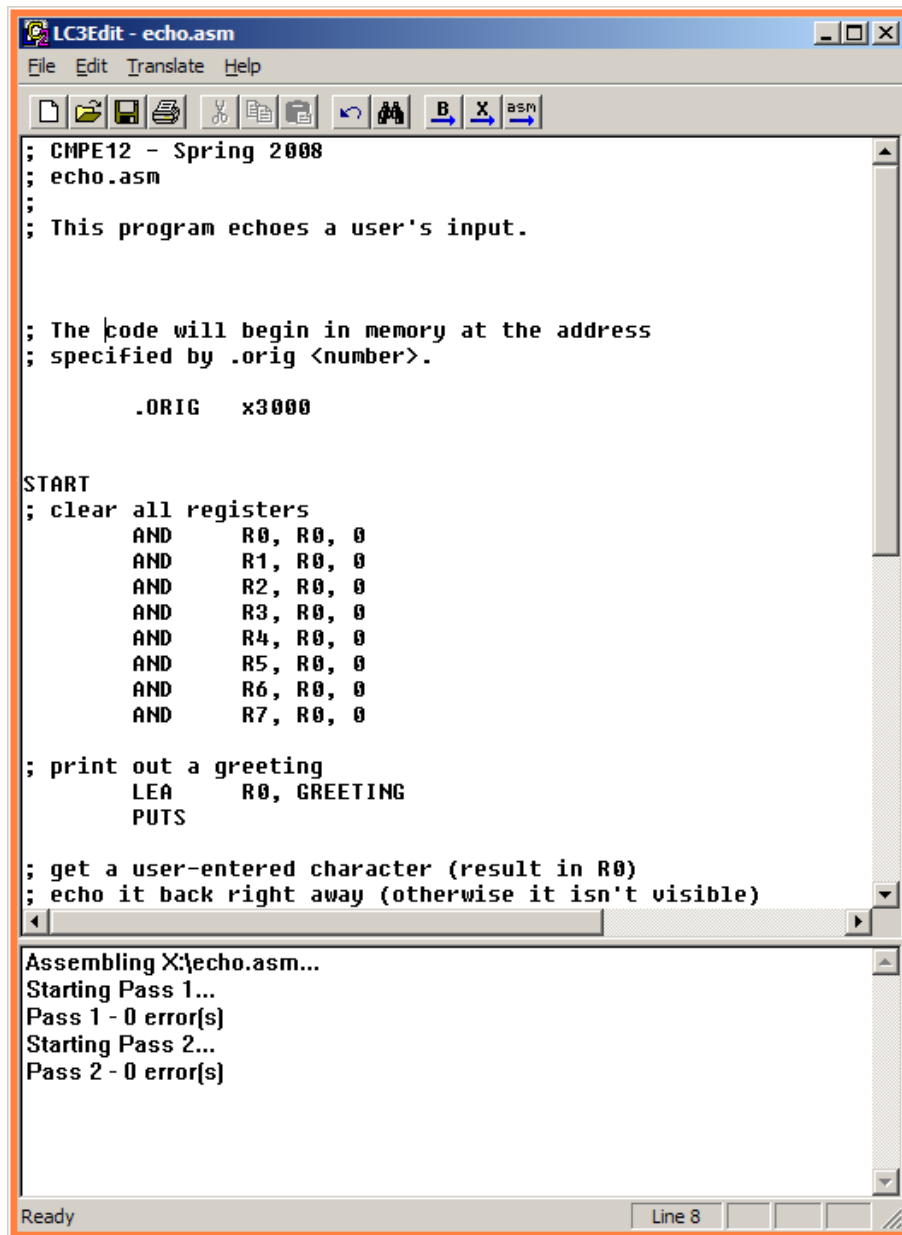
To continually read from the keyboard without first printing a prompt on the screen, use TRAP x20. That is, for each key you wish to read, the LC-3 operating system must execute the TRAP x20 service routine as shown in the echo.asm example. As done in echo.asm, if you follow TRAP x20 with the instruction TRAP x21, the character the user types will be displayed on the screen.

Hex String to Binary Conversion

If the user selected Decode, the input string will be in ASCII text (two bytes) that represent the hexadecimal, but your algorithm needs this to be a binary number in a single byte. (You could avoid this, but it is a requirement that you convert to this!) You must implement a **subroutine** that converts two ASCII hex characters to a single byte. You can assume that the characters will only be valid hexadecimal (x0-xF).

You should make this a function call using an JSR instruction. At the end of the subroutine before the return, a subroutine should leave the registers in the same state as when it was called. This ensures that the program that is calling the subroutine does not get corrupted data. In your subroutine, save **ALL** registers (except R6 – because it is the stack pointer) to the stack and restore the registers before returning – even if you don’t use them. In general, some of this saving/restoring may not be needed if all the registers are not used within the subroutine. This procedure ensures that the subroutine can be used by any code and is how a compiler would call a subroutine function.

In order to convert the number to binary, you need to add the appropriate number of “16s” and “1s” to the final number. This can be done by looping over each digit. For example, if the user inputs 23, there will be two 16s and three 1s. These must all be added together to get the final result, x35.



The screenshot shows the LC3Edit window titled "LC3Edit - echo.asm". The menu bar includes File, Edit, Translate, and Help. The toolbar contains icons for file operations and assembly. The main text area displays the following assembly code:

```
; CMPE12 - Spring 2008
; echo.asm
;
; This program echoes a user's input.

; The code will begin in memory at the address
; specified by .orig <number>.

        .ORIG    x3000

START
; clear all registers
    AND    R0, R0, 0
    AND    R1, R0, 0
    AND    R2, R0, 0
    AND    R3, R0, 0
    AND    R4, R0, 0
    AND    R5, R0, 0
    AND    R6, R0, 0
    AND    R7, R0, 0

; print out a greeting
    LEA    R0, GREETING
    PUTS

; get a user-entered character (result in R0)
; echo it back right away (otherwise it isn't visible)
```

The bottom status bar shows "Ready" and "Line 8".

The assembly output window at the bottom shows the following text:

```
Assembling X:\echo.asm...
Starting Pass 1...
Pass 1 - 0 error(s)
Starting Pass 2...
Pass 2 - 0 error(s)
```

Figure 3: Viewing echo.asm in the LC-3 editor. It has been assembled with no errors.

Binary to Hex String Conversion

This routine will convert an encoded Morse byte to two ASCII bytes for output. You may implement this either as a table or algorithmically. In general, it is the opposite of the hex to binary conversion above. This should also be a subroutine using the JSR instruction and save/restore all the registers. How did you implement it?

Error Checking

Your program should perform error checking. If the user does not input an upper-case E or D, it should display:

THAT IS AN ILLEGAL CHARACTER. PLEASE TRY AGAIN.

and prompt the user again. If the user selects a non-hexadecimal digit in a Morse code or a character other than A-Z or 0-9 in ASCII, it should display:

THAT IS AN ILLEGAL STRING. PLEASE TRY AGAIN.

and prompt the user again. If the input message exceeds 30 ASCII characters (i.e., the 31st input character is not ENTER), it should immediately display:

MESSAGE EXCEEDS 30 CHARACTERS. PLEASE TRY AGAIN.

and re-prompt them for a new input message. Note, that this means the maximum Morse to ASCII length will actually be 15 characters since each hex value requires two bytes in ASCII.

Encode/Decode Algorithm

The encode algorithm should take each ASCII character and output the two ASCII characters that represent the hex Morse encoding.

Output

Your program should output the encoded or decoded message to the screen. The encoding/decoding algorithm should store the message to be output starting in location x3221.

Format

Your program must be a text file of assembly code (i.e., a .asm file) that assembles with no errors. Please add comments to your code to help readability and understanding of the grader.

Extra Information

1. The starting location of your assembly code is to be x3000
2. The starting location of your data should be x3200
3. The input/output message buffers should be 31 memory locations each so that you may store ENTER as well. Your memory should look like:
x3200 <- E/D
x3201 through x3220 <- input buffer
x3221 through x3240 <- output buffer

4. The ENTER key is mapped to the line feed character on the LC-3 (ASCII x0A)
5. Acceptable inputs to be encoded are any digit ASCII characters within the range x30 to x39 and letters x41 to x5A (You can convert lower case if you want, but it isn't required.)
6. Acceptable inputs to be decoded are any digit ASCII characters within the range x30 to x39 and any letters x41 to x46 (A to F).
7. Acceptable outputs to be displayed are the ASCII characters within the range x30-x39 and x41-x5A.

You should demonstrate two test cases to your TA/tutor:

- Convert "SOS" to Morse code
- Convert "B810B8" to an ASCII string.

Lab Submission

Your lab will be submitted via your eCommons account. Please log in to eCommons using your UCSC account and attach the following files to your "Lab4" assignment submission:

- lab4.asm
- lab4_report.pdf

Note that the final report must be submitted in PDF format. Make sure to confirm that your assignment is SAVED and SUBMITTED before the deadline. You may resubmit your assignment an unlimited number of times up until the due date.

Check-off

For this lab, as with most labs, you will need to demonstrate your lab when it is finished to a TA/tutor and get it signed off. This must be done in any of your lab sections BEFORE the final deadline. Since this lab is due on Friday at 5pm, you will need to do this in your Wednesday or Thursday lab section.

If you end up needing a late check-off, we will do this (in person) using the files you submit in eCommons. You are only charged late days based on the submission time of these files and can get checked off in your next lab section or in TA office hours (preferred).

Grading template

This is a suggested grading rubric. It is also a good general guideline before submitting your lab to check off these points.

Requirements

☐ (20 pts) Input/Output

- Is the user properly prompted for input?
- Is the E/D key properly read?
- Are the bounds of the input message properly checked? (For Morse and ASCII)

☐ (15 pts) Conversion Subroutine

- Is the Morse converted properly to binary?
- Does the number conversion properly use a JSR instruction?
- Is the binary to hex string properly done?
- Are all the registers (except R6) stored on the stack?
- Are all the registers (except R6) restored from the stack?

☐ (20 pts) Encode/Decode

- Does it correctly encode a message?
- Does it correctly decode a message?
- Is the Morse conversion table stored as encoded binary (1 byte per character only)?
- Is the required memory layout properly used?

Lab write-up requirements

In the lab write-up, we will be looking for the following things. The lab report is worth 20 points. We do not break down the point values; instead, we will assess the lab report as a whole while looking for the following content in the report.

Along with the usual items, you should answer the following questions:

- What would be the Morse output for the message, “HELLO WORLD!”?
- What would be the ASCII output for the Morse code, “40101020A06048F0C058B0C040F0”?
- Is there any additional error checking that would be good to have?
- Why should the subroutine call save/restore the registers?
- How big is your conversion table?
- How many cycles does it take to convert two ASCII hex characters to a binary byte?
- How many cycles does it take to convert a binary byte to two ASCII hex characters?
- In your subroutine call, how many registers actually need to be saved/restored? (i.e. how many does your subroutine use?) How would this reduce your code?