CS 150 Lab 3

Secret Message Decoder

Goals

In this lab you will become familiar with the LC-3 assembler and simulator, and write an assembly language program.

So, let's...

- Learn the LC-3 assembly language.
- Learn how to assemble assembly code into machine language.
- Learn to program simple looping constructs in assembly.
- Learn how to input characters from the LC-3 console.
- Learn how to output characters to the LC-3 console.
- Learn the various LC-3 addressing modes.
- Learn how to use LC-3 condition codes.
- Learn how arithmetic and logical operations of the LC-3 work.
- Learn how to do register comparisons on the LC-3.
- Improve our debugging skills.

Lab Description...

1. To successfully complete this lab:

   "Write a program in LC-3 assembly language that encodes/decodes a secret message."

2. The basic algorithm for this lab might be:
   1. Initialize/setup character buffers and variables.
   2. Prompt for a secret message to encode/decode.
   3. Get an input string of characters into message buffer.
      1. Init buffer.
      2. Get/echo characters (including a period) until a period ('**.**') is entered.
      3. Store characters in a memory array.
      4. If a period ('**.**') is the first character, **HALT** the program.

4. "Encode" message buffer ('**XOR**' each character with a "secret character").
5. "Reverse" all the characters in the message buffer.
6. Output response message to the console.
7. Output encoded/decoded message to the console.
8. Go back to step 1.

3. The following guidelines should be followed in designing your program:
    - Your input character buffer routine should accept only valid "printable" characters (x20 thru x7E).
    - Valid characters (including the terminating period) should be echoed back to the console as they are entered.
    - Any character other than a "printable" character (x20 thru x7E) should echo to the console as a bell character (x07). This can be tested by trying "**ENTER**" or "**TAB**" and making sure a bell character, "□" is output.
    - Due to our encoding algorithm, the character "**j**" will be encoded as a "**DEL**" or **x7F** character which **cannot** be entered at the keyboard. Likewise, a semi-colon encodes to a period which is the terminating character. Do not worry about these anomalies (unless you want extra credit).
4. An example output of your program might be:

    Enter Message: Now is the time for all good men.
    Encoded Message: {px5qzzr5yyt5gzs5px|a5p}a5f|5bz[.
    Enter Message: {px5qzzr5yyt5gzs5px|a5p}a5f|5bz[.
    Encoded Message: Now is the time for all good men.

Before You Begin...

1. You might want to first read the following helps.
    - How to Encode a String...

        For this lab, you should "encode" a string using the following algorithm:

        1. For each character in the string, exclusive OR the character with a hexadecimal x0015 value.
        2. Reverse the characters in the string. If the number of characters in the string is odd, the middle character will remain in the same position.

    - How to XOR Numbers...

        The LC-3 ISA does not have an "exclusive OR" (XOR) instruction. An exclusive OR operation must be implemented using only ADD, AND, and NOT instructions.

Logically, the exclusive OR (XOR) operation can be seen as either of the following operations:

A XOR B == (A AND NOT B) OR (B AND NOT A)
A XOR B == (A OR B) AND NOT (A AND B)

It is easy to compute the bit-wise AND and NOT of two values in LC-3 assembly because there are instructions to perform those operations. However, computing the bit-wise OR is more difficult because there is no OR instruction. The bit-wise OR operation could be logically implemented using either of the following operations:

A OR B == (A AND NOT B) ADD B
A OR B == NOT((NOT A) AND (NOT B))

DeMorgan's law states that you can compute the bit-wise OR of two binary values by first inverting each of these values, "ANDing" the inverted values together, and then, inverting the result. In algorithmic form (assuming the two values are in R1 and R2):

1. Invert the value in R1 (overwriting R1).
2. Invert the value in R2 (overwriting R2).
3. AND R1 and R2, place the result in R3.
4. Invert the value in R3.

Thus, using the above equivalence operations, you can create an exclusive OR operation.

2. Second, decide on your approach to solving the problem... You might want to consider:
    - What information will my program prompt for?
    - What is considered valid input?
    - What should my output look like?
    - How will I test my program?

3. Third, layout your program structure. Use a systematic decomposition approach to your problem until you arrive at LC-3 assembly instruction.

4. Create a directory for your Decoder lab files.

5. Use your favorite text editor to create and edit your assembly program. Or, if you are using a Windows system, the editor and assembler are included in the same executable (**LC3Edit.exe**).

6. Review the instructions on how to create, load, and execute a LC-3 program for your particular system.

7. Finally, when you have decided what, where, and how, proceed with your implementation.

Hints and Suggestions

- Make sure you save your assembly files with the "**.asm**" extension or they will not be assembled by the assembler.
- Include comments that explain the abstract flow of your program as you write your program. Coming back later and adding comments loses much of the value comments provide in program development. Essentially, it will save you time in the long run.
- Use meaningful names for labels and variables.
- The assembly directive "**PUTS**" is a trap instruction that will print all characters in a string until reaching a **NULL** value. To use this directive, make sure your string ends with a **NULL** character.
- Do not use the "**IN**" trap instruction for this program, since it prints out a ':' and echoes the input back.
- Use the **HALT** trap instruction to terminate your program.
- Remember, "**XOR**" is not part of the LC-3 ISA, so you'll need to create an algorithm that will use **AND**'s, **ADD**'s, and **NOT**'s to implement this operation.

Let's Get Started...

Your algorithmic solution has to be systematically decomposed down to the level that is "syntactically correct" for a LC-3 assembler. This is not a hard process, but one that requires exactness. Carefully follow the steps below for a successful completion of this lab.

Step 1

After you have systematically decomposed your solution to the problem down to the LC-3 ISA abstraction level (assembly), students usually choose one of the following programming methods:

1. Type the whole program into your editor, get it to assemble, load it into the simulator and execute the program. Doesn't work, something's wrong, haven't a clue where to start... try this... try that... 15 hours later and still in the lab...
2. **OR**, they incrementally develop their program by writing a small "working" program (even as small as **HALT** (TRAP x25)) and then after that, they never "break" it. They add a few lines, assemble, and test. Add a few more lines of code, assemble, and test. Soon, they're done, passed off, and out of the lab in an hour.

It's your choice, but please CHOOSE WISELY!

Begin by entering a small program into your editor that you know will work. Add the assembler directives to load your program at memory location **x3000** (**.orig**) and put an end assembly directive (**.end**) at the end of your program. Save your program in a file with a "**.asm**" extension. For example:

```
        .ORIG x3000
start   lea R0, inmsg              ; point to input prompt
        PUTS                       ; output to console (trap 0x22)
        lea R0, outmsg             ; point to response message
        PUTS                       ; output to console (trap 0x22)
        HALT                       ; (trap 0x25)
inmsg   .STRINGZ "\nEnter Message:"
outmsg  .STRINGZ "\nEncoded Message:"
        .END
```

Note: Your assembly program should be well commented.

Step 2

Your program needs to be assembled before it can be "executed" by the LC-3 simulator. Undoubtedly, the first few passes through the assembler will disclose "syntax" errors. Correct these and continue the edit/assemble cycle until you have a "clean" assembly.

Now load your program object file into your simulator and verify that it works (ie. prints the message and halts). **DO NOT** leave **Step 2** until this works! If it doesn't assemble, load, or whatever, make the necessary changes and try again.

Step 3

From here on you should "incrementally" add functionality to your program by making small, meaningful changes to your program. Assemble often to "catch" syntax errors.

After a "clean" assembly, load your program object file into your simulator and verify that it works the way you intended. If there is a problem, you know where to fix it! Do this over and over until the lab is completed!

Work out an incremental approach to completing the lab. You may choose to "stub out" various parts of your program until they can be implemented. This might mean that you "make up" data (like a "canned" input buffer) and then later replace that code with the "real" code. The point is, "**DON'T BREAK YOUR PROGRAM!**".

Here is a suggested list of incremental steps to complete your lab:

1. Implement and test code to output both the request and response messages to the console.

2. Implement and test code to get characters from the keyboard, check for valid characters, and echo to the console.
3. Implement and test code that stores valid input characters into a memory array that is terminated with a **NULL** character.
4. Implement and test code to output the input string to the console when a "**.**" character is entered.
5. Implement and test code to exclusive **OR** all the characters in the input string with the hexadecimal character **x15**.
6. Implement and test code to reverse the characters in the encoded string (ie., "the first shall be last and the last shall be first").
7. Implement and test your program termination condition (ie., when a period is the only character entered).

And Finally...

1. **NOTE:** The 'bell' character (0x07) may not make a noise on your machine.
2. Try several input strings to verify your program works correctly. Test your program by letting your program encode an "encoded message". The original message should appear!
3. You will need to pass off your Decoder Lab with a CSAC TA. They will need to see your commented LC-3 assembly source code and the running program on the LC-3 simulator. If you are using a Windows system, be prepared to demonstrate your program on your computer.
4. You **ARE REQUIRED** to electronically submit your assembly source code program as part of your lab report (a pdf file) before a grade can be awarded! Go to blackboard to submit the lab report.

Grading Criteria

This lab is worth 30 points as follows:

5 points	Lab Report should contain systematic decomposition notes and flowchart
5 points	Your assembly code program uses good coding style. This includes good use of comments, meaningful label names, effective use of white space, etc.
5 points	Your program prompts the user for a character string. After a string has been entered and encoded, your program outputs a response message to the console followed by the encoded string.
5 points	Your input routine correctly detects and echoes valid "printable" characters (x20 thru x7E) - all others echo as a "beep" or bell character,

	"□". The input string is terminated when a period is entered and is correctly stored in memory as an array (ending with a **NULL** character).
5 points	The input string is correctly encoded/decoded using an exclusive "or" (**XOR**) operation.
3 points	The encoded string is correctly reversed.
2 points	Your program **HALT**s when only a period is entered after the prompt.

In addition to the above points, the following bonus applies:

+3 points	If a semi-colon '**;**' (**x3B**) is encountered in the input string, change it to a '**j** (**x6A**) character. If a lower case letter '**j**' (**x6A**) is encountered in the input string, change it to a semi-colon '**;**' (**x3B**).
+2 points	If checked off before Thursday by any CSAC TA or if checked off by Daniel J. Evans or Yi Guo anytime.