CMPE 12L Lab 4 - Winter 2014
Secret Encoder/Decoder

Due: February 23, 2014
100 Points (40 Report, 60 Work)

1 Background

Cryptography, from the Greek word kryptos, meaning "hidden," deals with the science of hiding a message from eyes you do not want to understand the contents of the message. The desire to do this has existed ever since humankind was first able to write. There are many ways to keep something secret. One way is to physically hide the document. Another way is to encrypt the text you wish to remain secret. Some people use this to keep others from understanding the contents of the files in their computer. Encrypting a file requires an input message, called the "plain text," and an encryption algorithm. An encryption algorithm transforms "plain text" into "cipher text." Just like a door needs a key to lock and unlock it, an encryption algorithm often requires a key to encrypt and decrypt a message. Just like a homeowner can selectively give the key to the front door to only those he wants to allow unaccompanied access, the author of a message can selectively give the encryption key to only those he wants to be able to read the message. In order for someone to read the encrypted message, he has to decrypt the cipher text, which usually requires the key. For example, suppose the plain text message is HELLO WORLD. An encryption algorithm consisting of nothing more than replacing each letter with the next letter in the alphabet would produce the cipher text IFMMP XPSME. If someone saw IFMMP XPSME, he would have no idea what it meant (unless, of course, he could figure it out, or had the key to decrypt it.) The key to encrypt in this case is "pick the next letter in the alphabet," and the decryption key is "pick the previous letter in the alphabet." The decryption algorithm simply replaces each letter with the one before it, and presto: the plain text HELLO WORLD is produced.

2 Assignment

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

2.1 Input

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

- The prompt: (E)ncrypt/(D)ecrypt:
    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: Encryption Key:

  1. The user will type an integer number from 0 to 31 followed by the ENTER key.
  2. Your program will convert this string of digits to one unsigned binary byte, store it in x3201, and use it to encrypt or decrypt the message.

- The prompt: Input Message:

  1. The user will input a character string from the keyboard, terminating the message with the ENTER key.
  2. One constraint: Messages must be less than or equal to #20 characters. (Recall: # means the number is decimal.)
  3. Your program will store the message, starting in location x3202. Since the message is restricted to #20 characters, you must reserve locations x3202 to x3216 to store the message. This includes the final ENTER character to denote the end of the message, but this is not encrypted. (Note that #20 = x14.)

2.2 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.

2.3 String to Decimal Conversion

The encryption key will be a string of ASCII digits input by the user, but your encryption algorithm needs this to be an unsigned binary number. You must implement a **subroutine** that converts a string one or two ASCII characters to a single unsigned byte. While the input format only requires this to be 0 to 31, make this general so that it can handle 0 to 99.

You should make this a function call using an JSR istruction. 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 to the stack and restore the registers before returning. 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 "10s" 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 10s and three 1s. These must all be added together to get the final result, #23.

2.4 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-numeric digit in the encryption key, it should display:

```
THAT IS AN ILLEGAL ENCRYPTION KEY. PLEASE TRY AGAIN.
```

and prompt the user again. If the user enters an encryption key that is too large, it should display:

```
OUT OF RANGE ENCRYPTION KEY. PLEASE TRY AGAIN.
```

If the input message excedes 20 characters (i.e., the 21st input character is not ENTER), it should immediately display:

```
MESSAGE EXCEDES 20 CHARACTERS. PLEASE TRY AGAIN.
```

and re-prompt them for a new input message.

2.5 Encryption/Decryption Algorithm

The encryption algorithm transforms each ASCII character in the message as follows:

- the low order bit of the character will be toggled. That is, if it is a 1, it will be replaced by a 0, if it is a 0, it will be replaced by a 1.
- The key will be added to the result of step 1 above.

For example, if the input (plain text) is A and the encryption key is 6, the program should take the ASCII value of A, 01000001, toggle bit 0, producing 01000000 and then add the encryption key, 6. The final output character would be 01000110, which is the ASCII value F.

The decryption algorithm is the reverse. That is,

- Subtract the encryption key from the ASCII code.
- Toggle the low order bit of the result of step 1.

For example, if the input (cipher text) is F, and the encryption key is 6, we first subtract the encryption key (i.e., we add -6) from the ASCII value of F, 01000110 + 11111010, yielding 01000000. We then toggle bit 0, producing 01000001, which is the ASCII value for A. The result of the encryption/decryption algorithm should be stored in locations x3217 to x322A.

2.6 Output

Your program should output the encrypted or decrypted message to the screen. Note that the encryption/decryption algorithm stored the message to be output starting in location x3216.

3 Format

Your program must be a text file of assembly code (i.e., a .asm file) that assembles with no errors.

4 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 21 memory locations each so that you may store ENTER as well. Your memory should look like:
   x3200 <- E/D
   x3201 <- Encryption key
   x3202 through x3216 <- input buffer
   x3217 through x322A <- output buffer
4. The ENTER key is mapped to the line feed character on the LC-3 (ASCII x0A)
5. Acceptable inputs to be encrypted are any ASCII characters within the range x20 to x5A
6. Acceptable outputs to be displayed are the ASCII characters within the range x20 to x7B.
7. For further reading on the subject of cryptography and its role in history, "The Code Book" by Simon Singh is recommended.

5 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 "Lab 4" 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.

5.1 Check-off

For this lab, as with most labs, you will need to demonstrate your lab when it is finished to the TA or tutor and get it signed off. You will also need to submit your lab files using eCommons.

5.2 Grading template

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

5.2.1 Requirements

☐ (20 pts) Input/Output

- Is the user properly prompted for input?
- Is the E/D key properly read?
- Are the bounds of the encryption key properly checked?
- Are the bounds of the input message properly checked?

☐ (20 pts) Integer Conversion Subroutine

- Is the encryption key converted properly to binary?
- Does the number conversion properly use a JSR instruction?
- Are the registers stored on the stack?

☐ (20 pts) Encryption/Decryption

- Does it correctly encrypt a message?
- Does it correctly decrypt a message?
- Is the required memory layout properly used?

5.2.2 Lab write-up requirements

In the lab write-up, we will be looking for the following things. The lab report is worth 40 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 key and plain text would produce the output cipher text that has the character x7B?
- What key and plain text would produce the output cipher text that has the character x20?
- Is there any additional error checking that would be good to have?
- Why should the subroutine call save/restore the registers?
- What problem would there be if we allowed the encryption key to be up to 99?
- 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?