CS 2461: Project 4 – Simple Encryption

Project Rules: You should work on your own on this project; projects 4,5,6 carry more weight than projects 1,2,3. You can use the code handed out for the calculator program, and you can reuse your code from Homework 4 and Lab assignments. Your project must meet the required specifications described in this document. There is no collaboration, of any sort, allowed on this project (we will be checking submissions for code similarity). You can discuss general LC3 programming issues/tips and can ask the instruction team for help on both LC3 programming as well as general design tips. **You must follow the specifications very carefully - including the messages to be printed**.

Recommendation: Read this project description as soon as possible. It is also recommended that you write out 'pseudo code' (or a flow chart) for this program – this will help you get started with writing the assembly program.

In this project you are required to implement an "encryption module", in LC3 assembly, that allows users to encrypt their messages using a simple encryption algorithm (using a variation of the Caesar's cipher algorithm and One Time Pads and an option of the rotate operation) that mimics the ARX (Add Rotate Xor) ciphers of today (the Rotate aspect is de-emphasized in the grading rubric in this project). In this assignment, you will develop a program to encrypt a string of exactly 10 characters using an encryption algorithm described in this document. (The characters from the keyboard, in terms of their ASCII representation, are all printable characters with ASCII representation from x21, for the symbol "!", to x7E which is the symbol ~). Background on the crypto aspects, as well as details of the encryption algorithms you are required to implement are provided at the end of this document (in Appendix A).

1. Encryption Algorithm

In this programming project you will implement a simple ARX cipher (with a rotational component replaced by a shift operation) encryption algorithm which we call "SAD" based on a composition of three methods – Caeser's cipher, the Vigenere Cipher, and a right/left shift (all of which are described in detail at the end of this document in Appendix A). The goal is to encrypt a message, consisting of ASCII characters, using a key provided by the user. To decrypt the message correctly, the same key must be used by the decryption algorithm.

2. Project Requirements and Specifications

Your encryption/decryption programs (in LC3 assembly) **must** meet the following requirements – including the messages to be printed to the display and the memory locations to store results/messages. Your program **must not** print any messages besides the prompts specified and the INVALID INPUT message – all prompts *must* be in uppercase. Any deviation from this, without approval from the instructor, or the TAs will result in points being deducted. The program must use the SAD encryption algorithm described in this document. You will find that you can use your homework solutions and the code we handed out for most of the modules/functions that you need to implement; for example the XOR function from Labs and Modulo function from HW4, and you can use the code from HW4 or the Calculator program (provided on blackboard) to design the main interface as well as the ASCII to Binary conversion functions.

Specifications:
Your program will prompt the user for different inputs from the keyboard, as shown below, and then proceed to process the inputs depending on the user input. The prompts (and messages to be printed) are shown in red font -- we have included a file prompts.asm with the set of messages your program is allowed to print. To preserve privacy, input entered by the user will not be echoed to the display, i.e., you will need to use GETC and not IN to read a character from the keyboard. (Note: in real systems, the standard practice is to echo a * -- but you will NOT implement this method.)

(0) When you start the program, it first prints out the greeting **`STARTING PRIVACY MODULE`** and then goes to Step 1.

(1)The prompt:

`ENTER E OR D OR X`

- The user will type E or D or X. If the user input is not one of these three commands (E,D,X) then the program prints the **INVALID INPUT** message and goes back to printing the prompt and waiting for user input.
- Your program will accept that (valid input) character, store it someplace (you should specify clearly in your comments and report, where this is stored in memory) and will use the input, as we shall see momentarily.
- If the user types X then the program first erases the encrypted data (i.e., replaces the contents in memory where the encrypted data was stored with zeros), and then halts.

(2) If either E or D is chosen in (1) then the program prints the prompts:

`ENTER KEY`

- The user will type their key -- the key in this project consists of exactly 5 characters: a single digit less than 8 followed by a non-numeric character (or zero) followed by a 3 digit **number** between 0 and 127. Again, we will assume the user can successfully hit digit keys on the keyboard. To enter a 3 digit number the user must always enter three digits – for example to enter the number 27 they will enter 027.
- If the user enters an invalid key then the program must print **`INVALID INPUT`** and loop back to ask for the key entry **`ENTER KEY`**

- Your program must check for an invalid (or valid) key **after** it has read all five characters for the key.

(a) Your program will accept this input, store it someplace, and use it to encrypt or decrypt the message.
    - Important: You will need to figure out how to store and interpret this length 5 key entered by the user. Also, if you store the key permanently (i.e., after writing encrypted message) then you have a security vulnerability (keys should not be stored once encryption is completed!). Read details of the encryption algorithm to determine what needs to be done.

(b) If the input in Step 1 was D then it goes to Step (3). If the input in Step 1 was E then it goes to step (4). If the input in Step 1 was X then it goes to step (5).

(3) If the input in (1) was D (decryption) then your program must (i) decrypt the string (some ciphertext/message you stored) using the key entered in step 2 and (ii) write it (i.e., store it) to memory starting at location x5000. The cipher text string is a 10 character cipher text (i.e., after encryption) stored starting at location with label MESSAGE – details of encryption follow in step (4).

- *Note: to protect against losing the encrypted data stored starting at MESSAGE if a wrong key was entered, make sure you do not store the decrypted message into MESSAGE -- this will cause your original message to be overwritten.*
- The decrypted message must be written to memory locations starting at x5000. Note that if the user entered the wrong key, then it will write out (to memory x5000) a message (string of characters) that will not be equal to the original message.
- After decrypting the program returns to Step 1 to prompt for next user input.

(4) If the input in (1) was E (i.e.,encryption) then after getting the key in Step (2), print the prompt:

```
ENTER MESSAGE
```

a. The user will input a string/sequence of *exactly* 10 characters (can only be the ascii printable characters and do not include control characters) from the keyboard.

b. *Your program must read the input string of 10 characters and store the encrypted input string in memory location(with label) MESSAGE starting at x4000*. In other words, using the value of the key entered in Step(2), and the encryption algorithm provided, you should generate the cipher text and store the cipher text into memory starting at location labeled with MESSAGE (i.e., the label of the starting address of the location is MESSAGE).

c. Next, the program returns to Step 1 where it can prompt for next user input.

(5) If the input in (1) was X (i.e., Exit) then the program erases the memory locations where the encrypted data was stored (erase is simply writing 0 into the locations) and exits and halts, i.e., the locations of MESSAGE are erased.

3. Hand-in Requirements – What you need to submit.

You are required to hand-in two components: (1) your assembly code and (2) a report (in PDF) describing your implementation – this should include pseudo code. Your assembly code must be a `.asm` file. You must name the file as **firstinitial-lastname.asm**. For example, if your name is Jane Doe then the filename is J-Doe.asm. The report is meant to assess the effectiveness of your design and we are looking (a) for an algorithm describing your implementation, including description of each subroutine you have designed, and (b) answers to the (underlined) questions in this document. You **must** document your code, and include a flow-chart describing your solution. Note that you will get a grade of zero if your program does not work and you do not submit a report that describes your algorithm in detail.

4. Grading Criteria

- You must submit code that assembles correctly without errors – if your code does not assemble then you get zero points. If the code crashes during testing of specifications, you cannot earn more than 60% of the points.
- You must document your code – you lose up to 10% for poor (or no) code documentation.
- The code **must** meet all the specifications (user interface including following the rules for the prompts and messages to be printed, encryption and decryption algorithms).
- Your program must print **ONLY** the messages provided in the prompt.asm file as described in this document; you cannot print out even new lines etc.
- The encryption algorithms must be correctly implemented. If you implement two (out of three) of the encryption algorithms correctly and meet all specifications then you can earn up to 93% (i.e., 56 out of 60).
    - *We strongly recommend that you first correctly implement a program with the two encryption algorithms before implementing the third* (bit shift).

- Performance of the code – pay attention to minimizing redundant code.
- Approximate grade distribution:
    - Level 1: 10% Assembles without errors
    - Level 2: 17% Correct printing of all the prompts and user can interact with all of the prompts. Correct input of 5 character key that follows the format specified (single digit <8, non-numeric character or 0, and 3 digits). And user can input message of length exactly 10 – with no echo to the screen. Your program must allow entering 0 for each key field; and

value of 0 for a key field is equivalent to skipping that encryption algorithm. For example, 00027 will skip both bit shift and the Vigenere and will encrypt using only Caeser's cipher using 027 as the key for this algorithm. Similarly, 00000 will result in no encryption. Important: If your code does not follow specifications, including printing only the specified messages then you will not get full credit for Levels 3--9 even if they work correctly.

  - Level 3: 6% Error checking (valid keys, valid input etc.)
  - Level 4:17% Correct implementation of Caeser cipher. This level 4 encrypted message is stored at address x4000; and decrypted message is stored at x5000. (Check algorithms description to figure out how to test individual encryption algorithms.)
  - Level 5: 17% correct implementation of Vigenere cipher. This level 5 encrypted message is stored at address x4000; and decrypted message is stored at x5000.
  - Level 6: 20% Correct implementation of encryption/decryption algorithm using both Caeser cipher and Vigenere cipher.
  - Level 7: 5% If the user exits when prompted, the program zeros out any message it stores in memory and halts.
  - Level 8: 8% correct implementation of Bit Shift algorithm.
  - Level 9: 8% Correct implementation of all three encryption/decryption algorithms. Note: a complete and correct implementation of all three encryption algorithms could result in overall extra credit on this project.
- The description of the algorithms will provide a way to test the individual encryption algorithms by entering specific key fields.

Appendix A: Background and Encryption Algorithms

A1. Cryptography Background

Cryptography, from the Greek word *kryptos*, meaning "hidden", deals with the science of keeping information secret. 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, they have to decrypt the cipher text, which usually requires the key. Applications where a sequence of digits/numbers, such as the social security number, passwords or credit card numbers, have to be encrypted are numerous – examples include online purchasing (such as Amazon), patient records in health care systems.

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, they would have no idea what it meant (unless, of course, they could break the encryption, 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.

A2. Encryption Algorithm

The encryption algorithm you will implement is a composition of two well known (and not very secure!) techniques and a third (made up) technique meant to mimic the rotation cipher: (1) f_1 : Vigenere Cipher – this implements a lightweight variation of the one time pad techniques, (2) f_2 : Caeser's cipher (generalized to a shift cipher), and (3) f_3: shift bits. The **SAD** *encryption algorithm* used by your security module works as follows:

- System prompts user for a key of length 5 consisting of a single numeric digit less than 8, followed by one non-numeric character or the numeric character 0, followed by a 3 digit number between 0 and 127 (i.e., they must enter 3 digits – each 0 through 9). Assume user enters: $\mathbf{z_1x_1\ y_1y_2y_3}$ (the key must be of length 5):
    - Let z_1 denote the single digit number between 0 and 7 is
    - Let x_1 denote the non-numeric (or the character 0) character (x_1 cannot be a digit in the range 1 to 9)
    - and let $y_1y_2y_3$ denote the 3 digit number.

    - For example, if they enter the key `4&106` then $z_1=4$, $x_1=\&$, $y_1=1$, $y_2=0$, $y_3=6$ (for the 3 digit number 106). .
        - If they enter `30027` then *z1=3, x1=0, y1=0, y2=2, y3=7.*
- User then enters (as described in the specifications in section 2) the plain text string ***w*** of 10 characters from the keyboard.
- The string *w* is first encrypted using Vigenere cipher with key $\boldsymbol{K = x_1}$, i.e., the function f_1 , to get the cipher text $\boldsymbol{u = f_1(w)}$.
    - For example, with input key `4&106`, K='&' (the ASCII representation of &) for this phase of encryption.
- The string *u* is then encrypted using Caeser's cipher with key $\boldsymbol{K=y_1y_2y_3}$, i.e., the function f_2 , to get the cipher text $\boldsymbol{v= f_2(u) = f_2(f_1(w))}$.
    - For example, with input key `4&106`, K=106 for this phase of encryption
- Finally the string *v* is then encrypted using the shift operation with key $\boldsymbol{K=z_1}$, i.e., the function f_3, to get the cipher text $\boldsymbol{c = f_3(v)= f_3(f_2(f_1(w)))}$.
    - For example, with input key `4&106`, K=4 for this phase of encryption.
- <u>You need to figure out how decryption works</u> – you have to apply the inverse functions f_1^{-1} and f_2^{-1} and f_3^{-1} which you can derive based on the descriptions of f_1 (Caeser's cipher) and f_2 (Vigenere cipher),and f_3 (bit shift) in what follows.

Note – Grade Options: You can choose to implement a simpler encryption algorithm using only Caeser's cipher and Vigenere cipher (i.e., $\boldsymbol{f_2(f_1(w))}$ ***)*** *to earn up to 93%.* We recommend that you first implement this simpler solution before implementing the shift cipher.

<u>The algorithms</u>

A2.1: Function f_1 – Vigenere Cipher

The one time pad (OTP) encryption method is known to be the most secure form of encryption. In a simplistic definition of this scheme, a string of binary symbols $p_1 p_2 p_3 \ldots p_M$ of length M is encrypted using a key $k_1 k_2 k_3 \ldots k_M$ of length M to create a cipher text $c_1 c_2 c_3 \ldots c_M$ where $c_i = (p_i \oplus k_i)$ (where $\oplus$ is the exclusive OR operation). If the key is random and sufficiently long, encryption is impossible to break assuming the key is used exactly once. An example of a OTP for a single character from the keyboard (i.e., an ASCII character) would be a 7-bit key (since each ASCII character can be represented as an 7 bit number). In this project we will use a predecessor (and weaker form) of this method – a Vigenere Cipher – to implement our encryption function $\boldsymbol{f_1}$.

In our algorithm, the key can be a non-numeric character or the digit 0. given an input plain text string $p_1 p_2 p_3 \ldots p_M$ of length M (which is exactly 10 in this project) the cipher text is computed as $c_1 c_2 c_3 \ldots c_M$ where $c_i = (p_i \oplus K)$ where K is a 7-bit key (i.e., a 7 bit binary number corresponding to the ASCII representation of the character entered as the key), each p_i is a symbol/character from the keyboard, and $\oplus$ is the bitwise exclusive OR (XOR) operation.

Important: Recall that the user enters the five characters **z_1x_1 $y_1y_2y_3$** as the key, and for the Vigenere encryption the key is the character **x_1** .

- *If x_1 =0 then the key K=0 (i.e., the binary representation of the number 0), else key K is the ASCII code for the character x_1.*
- Note that by setting x_1=0, we are bypassing the Vigenere encryption (i.e, the XOR) since for any binary value A we have A XOR 0 = A.

For example, if the input string (to the Vigenere encryption algorithm) is **a8** and the key x_1 =@. The ascii codes for a8 are x61 and x38 respectively or binary strings 0110 0001 and 0011 1000. The key *K* is x40 (binary 0100 0000) then the encryption results in the two binary strings (01100001 $\oplus$ 0100 0000= 0010 0001 = x21) and (0011 1000 $\oplus$ 0100 0000 = 0111 1000 = x78). (Note: LC3 uses 16 bits but in these examples the higher order 8 bits are all set to 0, so we did not include the values of bits 15 through 8). Therefore the two encrypted values, i.e., $c_1 c_2$, are x21 and x78 which if printed to the display would print **!x**.

- If **x_1** =0, then K is set equal to 0 (16 bit representation of 0) and the exclusive OR results in the same string, i.e., the encrypted string would be the same as the plain text a8.

How do you decrypt a message that has been encrypted using this scheme ? You will need to work this out to complete the project. Recall properties of XOR operation to figure this out.

A2.1: Function f_2 – Caeser's Cipher

For the encryption function f_1, you will implement a simple version of an algorithm called Caesar's Cipher. The term Caesar's cipher is an ancient encryption technique used by Julius Caesar to send secret messages. At its heart is the concept of modulo arithmetic (recall this from your discrete math class—remainder in integer division). Caesar used it to encrypt the alphabet but the key was fixed at *K=3*. The shift cipher we use allows the key K to vary.

Recall that the user enters the five characters **z_1x_1 $y_1y_2y_3$** as the key. For Caeser's cipher the key K is the three digit number **$y_1y_2y_3$** and K can be an integer from 0 to 127. For example, if the user enters 036 for **$y_1y_2y_3$,** then the key K is the decimal value 36 (*this is where you need to convert from 3 character ASCII input to a 3 digit integer represented in 16-bit binary*).

Conceptually it works as follows. The input (message) plain text to be encrypted is a string of M symbols $p_1 p_2 p_3 ... p_M$, each symbol p_i can take on *N* different values. The other part of the input is the encryption key *K* which must be less than *N*. To encrypt the message each character (i.e., value p_i) in the input is replaced by the cipher text character/value $c_i = (p_i + K)$ *modulo N.*

- In this project, the value of N=128 (and K must be less than 128). (this is because each character to be encrypted is represented in 7 bit ASCII). Therefore, the cipher text character c_i is obtained by adding the key using modulo 128 arithmetic (addition mod 128). In other words, for each plaintext input character p_i , $c_i = (p_i + K)$ *Modulo 128.*

    - Example: If the input plain text character string is **)r** and the encryption key is 6, then in step 1 the program takes ASCII value of **r**, 01110010 (x72) which is 114 in decimal, and adds (modulo 128) encryption key 6 to get 120 decimal (binary 01111000 or x78). This is the ASCII value for **x**. It takes ASCII value of **)** which is x29 =00101001, decimal 41) and adds 6 (modulo 128) to get x2F or decimal 47. This is the ASCII value for **/** and therefore the string in cipher text is **/x**
    - **Note:** If the encryption key K=0 then the cipher text is the same as the plain text (i.e., there is no encryption).

- The decryption algorithm works in exactly the reverse order as the encryption. The user provides the key K, and we apply the algorithm to the cipher text $c_1 c_2 \ldots c_n$ (where $n=10$ since we are dealing with a length 10 string). We subtract the encryption key K from ASCII code for c_i. The subtraction is once again using modulo arithmetic. Therefore, we compute $p_i = (c_i - K)$ *modulo 128.*
    - Example: if the cipher text is G and the encryption key is 6, we first subtract encryption key (i.e., we add -6 modulo 128) from 01000111 (decimal 71) to get 01000001 (decimal 65).

- Note: Recall from cs1311 that $(x-y) \bmod N = (x+y') \bmod N$ where $y' = N-y \pmod N$ (i.e., y' is the inverse of y *modulo* N). Therefore to decrypt we can add the inverse of the key modulo N. For example, $(1-4) \bmod 10 = (1+6) \bmod 10$ since inverse of 4 is $10-4=6$. You need to use this property to implement your assembly code LC3, so make sure you understand how this works. Note that since the key K is less than N, the number N-K is a positive integer less than N.

A 2.3: Function f_3 – Bit Shift

The rotate aspect of an ARX cipher performs a circular rotate of the bit string. For this project, we replace the rotate operation with a simple bit shift – each 16 bit value will be shifted left K times where K is the key for this phase.

Recall that the user enters the five characters $\mathbf{z_1 x_1\ y_1 y_2 y_3}$ as the key. The key for the Bit shift is the decimal interpretation of $\mathbf{z_1}$. For example, if the user enters $\mathbf{z_1}$=3 then the key is the decimal value 3 (i.e, left shift 3 places).

- **Note:** If the user enters $\mathbf{z_1}$=0 then the string is not shifted left (i.e., shifted left 0 times).

For an input text string $p_1 p_2 p_3 \ldots p_M$ of length M (which is 10) the cipher text is computed as $c_1 c_2 c_3 \ldots c_M$ where $c_i = (p_i \ll K)$ where K is a single digit between 0 and 7, each p_i is a symbol/character from the keyboard, and << is the bitwise left shift operation. For example, if the input string is a (with ascii code x61 or binary string 0000000001100001) and the key K is 4 then the encryption results in the binary strings (0000000001100001 << 4 = 0000011000010000). The decryption would do a right shift by K to recover the plain text.

How do you shift left in LC3? How do you right shift in LC3 ? You will need to work this out to complete the project.

Comment: You can test correctness of each encryption algorithm by setting the corresponding key to 0. For example for the key K= $\mathbf{z_1 x_1\, y_1 y_2 y_3}$

- If z_1=0, then there is no left shift performed.
- If x_1=0 then there is no XOR operation performed on the input string.
- If $y_1 y_2 y_3$=000 then Caeser's cipher does not change the input.
- Therefore you can test each encryption algorithm as:
    - To test Vigenere (XOR), set z_1=0 and $y_1 y_2 y_3$=000
    - To test Shift, set x_1=0 and $y_1 y_2 y_3$=000
    - To test Caeser's cipher, set z_1=0 and x_1=0.

Appendix B:
Project 4 – Clarification on specifications and sample execution

We want to remind you that you must follow all the specifications very carefully. This includes the strings that need to be printed; for example the prompts or error messages (this include lowercase and uppercase requirements).

Below are some examples to illustrate how your program should behave on some inputs. To check the outcome of your encryption/decryption algorithms, you need to check the memory contents at x4000 (Message) and x5000 (decrypted string).

Example program execution: user input in **red bold**, program prompts/outputs in **blue**. NOTE: The user inputs are shown below for clarity but they are NOT echoed by the program. Comments are shown in italics (black font). Note (again!) that a user can enter 0's for the key field and this results in a particular encryption algorithm being skipped. This sample execution does not show the other standard prompts printed by the LC3 simulator (i.e, we included only the user input and the messages that should be printed by your program):

```
STARTING PRIVACY MODULE
ENTER E OR D OR X
E
```
*The user will **not** press <enter>*
```
ENTER KEY
0@067
ENTER MESSAGE
HeHeHeHeHe
```
The encrypted values stored starting at `x4000` *will be* `x4B, x68, x4B, x68, x4B, x68, x4B, x68, x4B, x68` *(corresponding to KhKhKhKhKh*
```
ENTER E OR D OR X
D
ENTER KEY
0@067
```
The encrypted values stored starting at `x4000` *will be* `x4B, x68, x4B, x68, x4B, x68, x4B, x68, x4B, x68` *(corresponding to KhKhKhKhKh*
The decrypted values stored starting at `x5000` *will be* `x48, x65, x48, x65, x48, x65, x48, x65, x48, x65`
```
ENTER E OR D OR X
X
```
The contents of memory location `x4000` *etc. will be* `x0`

Some additional Information:

1. Dealing with invalid input commands.

```
STARTING PRIVACY MODULE
```

```
ENTER E OR D OR X
B

INVALID INPUT
ENTER E OR D OR X
```

2. Invalid key entry

```
ENTER KEY
9&233
INVALID INPUT
ENTER KEY
```

3. User can enter "0" to skip that part of encryption for that key

```
STARTING PRIVACY MODULE
ENTER E OR D OR X
E
ENTER KEY
00000
ENTER MESSAGE
HeHeHeHeHe
```

The encrypted values stored starting at x4000 will be 'H' 'e' 'H' 'e' (etc).

```
ENTER E OR D OR X
E
ENTER KEY
00021
ENTER MESSAGE
HeHeHeHeHe
```

The encrypted values stored starting at x4000 will be `x5D, x7A, x5D`*, etc. corresponding to charcters] z]z etc..In other words, the only encryption applied is the Caesar cipher with key = 21 (the entry 021).*