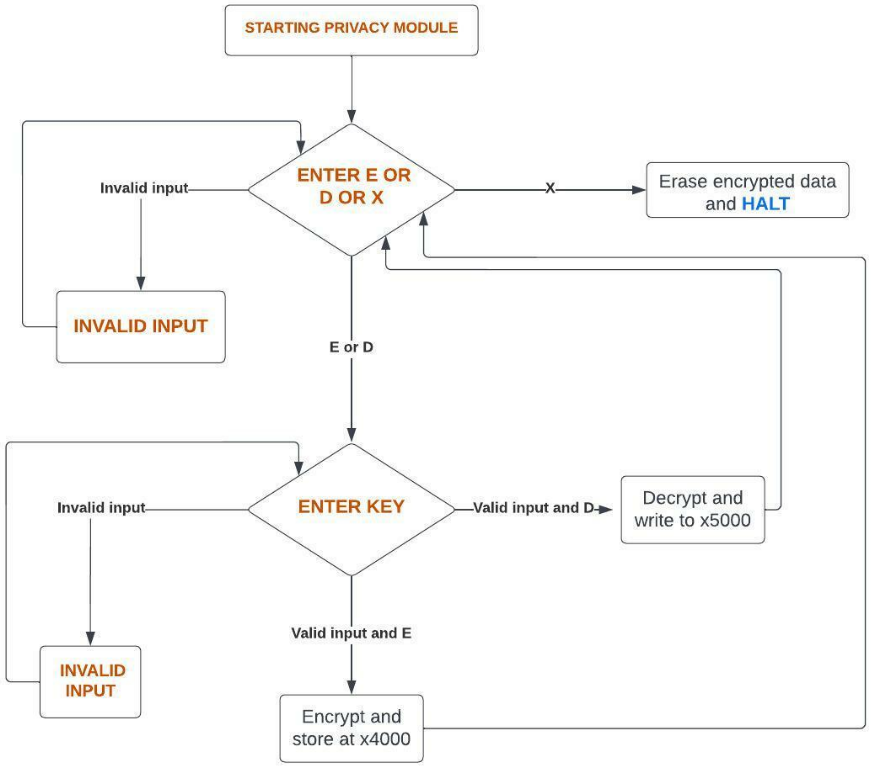
Project 4: Simple Encryption

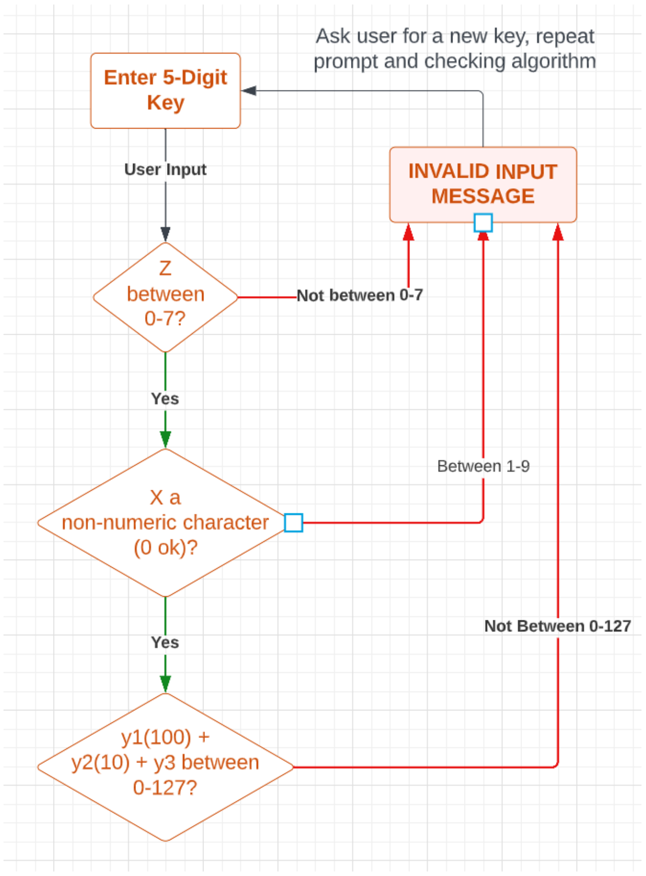
I. Flowchart and Pseudocode

During the initial stages of this project, I created a basic abstracted flowchart that documented the "outline" of the program:

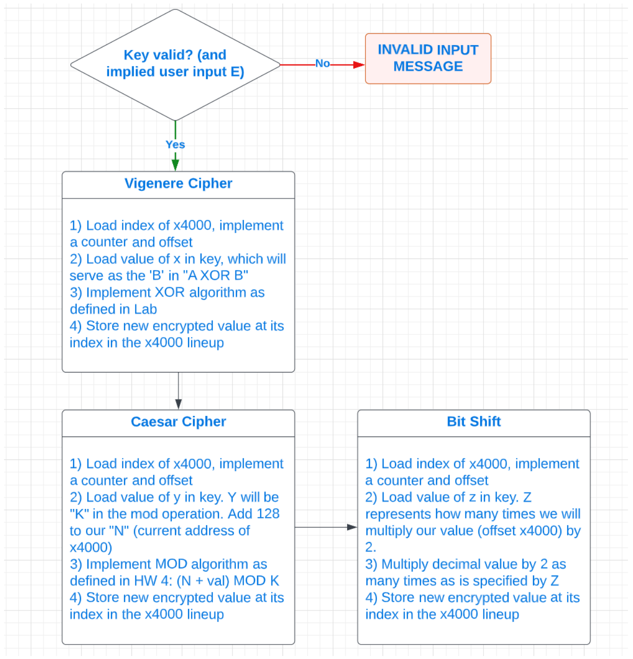


This flowchart illustrates a very **basic** implementation of the encryption program. It follows all of the user inputs (as well as what the program will print) and follows each of the steps of the program. First, the system prints "**STARTING PRIVACY MODULE**", and then prompts the user to enter **E**, **D**, or **X** as an input. If the input is invalid (not E,D, or X) the system will print out an "Invalid input" message. Otherwise, if E or D, the program will proceed to implement either encryption or decryption depending on the specification. If the user enters X, the program will erase the encrypted data and halt.

On a more specific level, I created three flowcharts describing how encryption, decryption, and the "key check" subroutines worked. These flowcharts contain more detailed pseudocode about the methods that are implemented in each step of the program:


Ask user for a new key, repeat prompt and checking algorithm
Enter 5-Digit Key
User Input
INVALID INPUT MESSAGE
Z between 0-7?
Not between 0-7
Yes
Between 1-9
X a non-numeric character (0 ok)?
Yes
Not Between 0-127
y1(100) + y2(10) + y3 between 0-127?

Flowchart for Key Check Method



Vigenere Cipher

1) Load index of x4000, implement a counter and offset
2) Load value of x in key, which will serve as the 'B' in "A XOR B"
3) Implement XOR algorithm as defined in Lab
4) Store new encrypted value at its index in the x4000 lineup

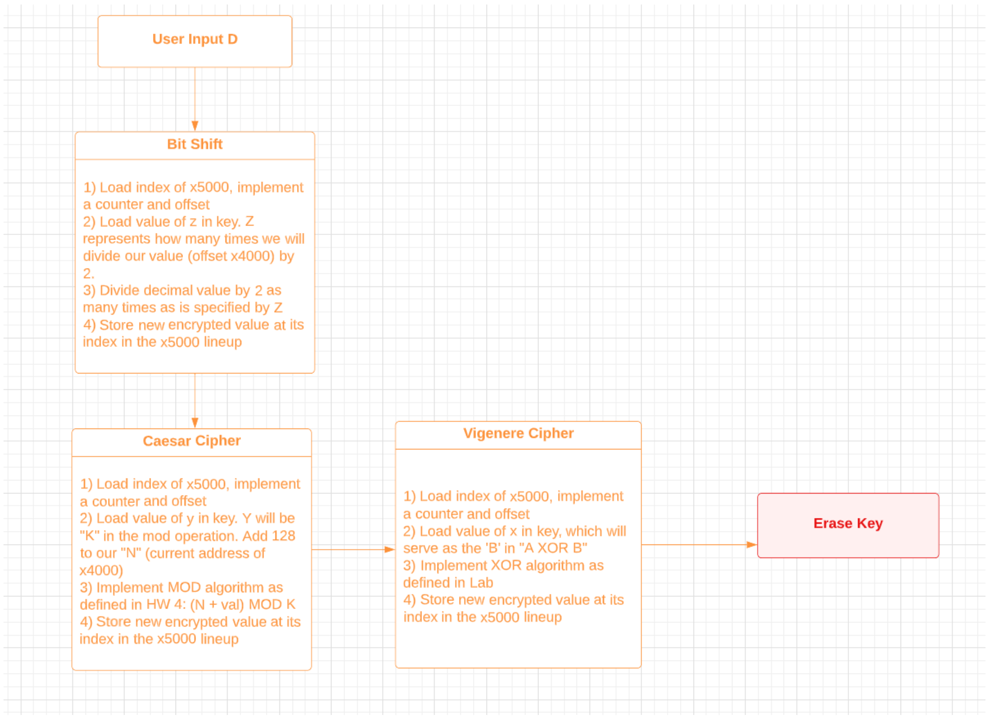
Caesar Cipher

1) Load index of x4000, implement a counter and offset
2) Load value of y in key. Y will be "K" in the mod operation. Add 128 to our "N" (current address of x4000)
3) Implement MOD algorithm as defined in HW 4: (N + val) MOD K
4) Store new encrypted value at its index in the x4000 lineup

Bit Shift

1) Load index of x4000, implement a counter and offset
2) Load value of z in key. Z represents how many times we will multiply our value (offset x4000) by 2.
3) Multiply decimal value by 2 as many times as is specified by Z
4) Store new encrypted value at its index in the x4000 lineup

Flowchart for Encryption


User Input D
Bit Shift
1) Load index of x5000, implement a counter and offset
2) Load value of z in key. Z represents how many times we will divide our value (offset x4000) by 2.
3) Divide decimal value by 2 as many times as is specified by Z
4) Store new encrypted value at its index in the x5000 lineup
Caesar Cipher
1) Load index of x5000, implement a counter and offset
2) Load value of y in key. Y will be "K" in the mod operation. Add 128 to our "N" (current address of x4000)
3) Implement MOD algorithm as defined in HW 4: (N + val) MOD K
4) Store new encrypted value at its index in the x5000 lineup
Vigenere Cipher
1) Load index of x5000, implement a counter and offset
2) Load value of x in key, which will serve as the 'B' in "A XOR B"
3) Implement XOR algorithm as defined in Lab
4) Store new encrypted value at its index in the x5000 lineup
Erase Key

Flowchart for Decryption

Finally, I wrote out pseudocode that organized and explained the steps of the program provided in the specifications. This pseudocode ties together all four of the separate flowcharts I created, providing a step-by-step interface for Project 4.

```
print STARTING PRIVACY MODULE
1) print ENTER E OR D OR X
    if invalid input
        print INVALID INPUT
        go back to 1)
    if X
        go to 5)
    if E or D
        go to 2)

2) print ENTER KEY
//note: 'valid input' is 5 char, single digit <8 followed by
//non-numeric character and then 3-digit num btwn 0-127
    if invalid input
        print INVALID INPUT
        go back to 2)
    if D and valid input
        go to 3)
    if E and valid input
        go to 4)

3) decryption
  (a) decrypt string
      (i) bit shift RIGHT
      (ii) caesar cipher
      (iii) vignere cipher
  (b) store in memory x5000
  (c) go back to 1)
```

```
4) print ENTER MESSAGE (encryption)
  (a) user input 10 char
  (b) read input and store in x4000, encrypt
      (i) vignere cipher
      (ii) caesar cipher
      (iii) bit shift LEFT
  (c) go back to 1)

5) erase memory locations where encrypted data was stored
   halt
```

II. Descriptions of Subroutines

A. subINVALID

This subroutine simply prints out a message (invalid) that tells the user that their query input was invalid, and then loops back to take a new input. For example, if prompted to enter E, D, X, and the user enters 'A', this will print the invalid message and ask for a new input.

B. subENCRYPT

This subroutine calls many other subroutines that complete the encryption process. First, it calls subKEYIN to collect a user key input and check if that input is valid. Then, if the key is valid, subENCRYPT takes in a 10-character message and encrypts it based on the key. subENCRYPT uses JSRR to enter three other subroutines, including subXOR, subCAESAR, and subERK. subXOR and subCAESAR complete the encryption process, and subERK erases the key to preserve encapsulation/data hiding once the encryption has been completed. At the end of subENCRYPT, the program returns back to the 'main' control structure and the user is prompted for another input (E,D,X).

C. subDECRYPT

subDECRYPT decrypts the user message in the reverse order of subENCRYPT. It stores a key in a similar way using subKEYIN, and then copies over the encrypted message to address x5000 to begin the decryption process. It calls the subCAESARDEC method (decrypts the Caesar Cipher) and then calls the subXORDEC method (decrypts the XOR). Finally, it calls subERK to erase the key again.

D. subKEYIN

This subroutine checks that all of the key inputs are valid inputs. For the first checkZ, the program checks if z is between 0-7 in order to implement bit shift encryption and decryption. If the input is valid, the program will then proceed to checkX, and if not, will branch to an InvRange loop that prints the invalid message and asks for a new (correct) key. In checkX, the program checks to see if x is NOT between 1-9, that is, x is either zero or a non-numeric character. A similar condition repeats, and if x is a valid input, the program proceeds to check

Y. Because Y consists of 3 inputs, the program goes to subMULT to check if the total value of Y is between 0-127. If all of the inputs are valid, subKEYIN returns to whatever point in the code it was called in (encrypt or decrypt) after storing all of the **valid** key values at x3500.

E. subEXIT
subEXIT simply jumps to subERM, which will erase the message. So, if a user input is 'X' and the user has decided to exit the program, their encrypted message will be erased and the machine will HALT.

F. subERK
subERK erases the key stored beginning at x3500 by looping through each value and replacing it with zero.

G. subERM
subERM functions similarly to subERK in that it begins at x4000 and loops through each value, replacing it with zero to effectively "erase" the encrypted message.

H. subMULT
This subroutine is called within the subKEYCHECK subroutine, and its purpose is to return a 3-digit value of Y. It multiplies each of the individual y-values (y1, y2, y3) by their corresponding weight in the hundreds or tens place. So, for example, y1 is multiplied by 100 using repeated addition and added to a result. Then, y2 is multiplied by 10 using repeated addition and added to the same result. Finally, y3 needs no alteration because it is already weighted in the ones decimal place, so it is simply added to the preexisting (y1 + y2). Additionally, this subroutine checks for "zero" cases, where the program skips over the multiplication step entirely if the key entered is zero (for example, entering a key of 048 will skip over the multiplication of y1 by 100 for efficiency).

I. subXOR
This subroutine loops through each value beginning at x4000 and performs the XOR algorithm on each character of the message. It follows the format A XOR B, where A is the character in the message and B is the entered key. The actual XOR algorithm is detailed in the comments of the code, but bitwise NOT and AND operations are essentially used to compute $(AB')'(BA')'$.

J. subCAESAR
This subroutine loops through each value beginning at x4000 and computes the value of the character at that address, plus the key entered, MOD 128. So, the formula that it computes is $(p + k)\ MOD\ 128$ for each p in the encrypted message. The program performs the MOD operation using repeated subtraction implemented as division, and utilizes a loop to subtract until the value is negative. When the value is negative, 128 is added to it for a final time and we have the modulus result.

K. subXORDEC
This subroutine loops through each value beginning at x5000 and performs the XOR algorithm on each character of the message. It follows the format A XOR B, where A is the character in the message and B is the entered key. The actual XOR algorithm is detailed in the comments of the code, but bitwise NOT and AND operations are essentially used to compute $(AB')'(BA')'$. Its only difference from subXOR is that it begins at x5000 (to compute on the decrypted message) instead of beginning at x4000.

L. subCAESARDEC
This subroutine loops through each value beginning at x4000 and computes the value of the character at that address, plus the key entered, MOD 128. So, the formula that it computes is $(p + k)\ MOD\ 128$ for each p in the encrypted message. The program performs the MOD operation using repeated subtraction implemented as division, and utilizes a loop to subtract until the value is negative. When the value is negative, 128 is added to it for a final time and we have the modulus result. This subroutine's only difference from subCAESAR is that it begins at x5000 (to compute on the decrypted message) instead of beginning at x4000.

M. subBITSHIFTL (not implemented in code)
This subroutine uses the key Z to implement a left bit shift, which shifts the binary value of the character in the message to the left *Z* times and pads the created space on the right with zeros. Although not implemented in the code, this method is achieved by multiplying the decimal value of each character in the message by 2 as many times as is specified by *Z*.

N. subBITSHIFTR (not implemented in code)
This subroutine uses the key Z to implement a right bit shift, which shifts the binary value of the character in the message to the right *Z* times and pads the created space on the left with zeros. Although not implemented in the code, this method is achieved by dividing the decimal value of each character in the message by 2 as many times as is specified by *Z*.

III. Answers to Questions

A. How do you decrypt a message that has been encrypted using this (XOR) scheme?

1. To decrypt a message that has been encrypted using XOR, we can simply input the decrypted message and repeat the XOR algorithm using the key. Similar to the method used to encrypt, we can iterate through each character in the 10-character message and 'XOR' it with the value of the key. Finally, it is important that, since we implement the Vigenere Cipher **first** when we encrypt, we call that method **last** when we decrypt. The SAD decrypt algorithm works in the reverse order of the SAD encrypt

algorithm in terms of which particular method (Vigenere, Caesar, Bit Shift) is implemented at what time.

B. How do you shift left in LC3?
    1. To shift left in LC3 and encrypt the message entered by the user, we can multiply the decimal value of each character in the message by 2 as many times as is specified by *Z*. Multiplying by 2 is equivalent to bit-shifting to the left once, so we will multiply by 2 *Z* times. This is equivalent to converting the decimal value to binary and performing a bit shift manually, and it computes the value in fewer steps when coding.

C. How do you shift right in LC3?
    1. To shift right in LC3 and encrypt the message entered by the user, we can divide the decimal value of each character in the message by 2 as many times as is specified by *Z*. Dividing by 2 is equivalent to bit-shifting to the right once, so we will divide by 2 *Z* times. This is equivalent to converting the decimal value to binary and performing a bit shift manually, and it computes the value in fewer steps when coding.