UNIVERSITY OF CALIFORNIA, SANTA CRUZ
BOARD OF STUDIES IN COMPUTER ENGINEERING

CMPE12/L: COMPUTER SYSTEMS AND ASSEMBLY LANGUAGE

Lab 4: Caesar Cipher
Worth 75 points (65 lab + 10 report)

Lab Objectives:

In this lab, you will build on your LC3 skills, and apply them to a more complex program.

This lab focuses on the design and use of subroutines. These are modular pieces of code that can run independently from other elements of your code. These are analogous to functions in higher-level languages. As coding projects become larger and more complex, good subroutine design becomes essential to the design process.

This lab also introduces you to indexing multidimensional arrays.

In this lab, you will create a Caesar Cipher, which encrypts or decrypts a user-supplied string. Your code must read the string, store it, and then display the encrypted and plaintext versions of the string.

Lab Overview:

You want to be able to hang out with the cool kids that use the Caesar Cipher (see http://en.wikipedia.org/wiki/Caesar_cipher) to tell jokes on the internet, so you decide to write a program to help you encrypt and decrypt short strings using this cipher.

While there are many approaches to making a program that meets these specifications, we want you to learn to make subroutines that have their own specifications. These are discussed in detail later in the lab, but the subroutines we want you to design are:

- **STORE**(char_to_store, row, col): Store an ASCII character in the specified row and column of a private array
- **LOAD**(row, col): Return the ASCII character that is stored in the specified row and column of the same private array
- **PRINT_ARRAY**() : Print the contents of the private array
- **ENCRYPT**(char_to_encrypt, cipher_key) Cipherize a given ASCII character, and return the result.
- **DECRYPT**(char_to_decrypt, cipher_key) De-cipher a given ASCII character, and return the result.

These are written in function notation, but this is somewhat improper, since a subroutine is not quite the same thing as a function (Question: What is the difference between a subroutine and a function?)

For every subroutine you write:

- Make a flowchart for the subroutine. This flowchart must be readable and should correspond to your code (that is, your LC3 labels should correspond to the labels on your flowchart's nodes). The subroutine's input and output should be clearly described.
- In code, use header comments to clearly describe the subroutine's interface – that is, clearly describe the registers used to get input and output from the subroutine.
- The subroutine should be *call-safe*. That is, the subroutine should only change things its specification says it will change. Use ST instructions at the beginning of your subroutine to save the original values in your registers, and LD at the end to restore them.

You may write more subroutines if you find it useful, but you *must* have the above subroutines.

AFTER you have written and tested your subroutines, you will design a full program that you can use to encrypt and decrypt short messages.

PART 0: Example subroutine

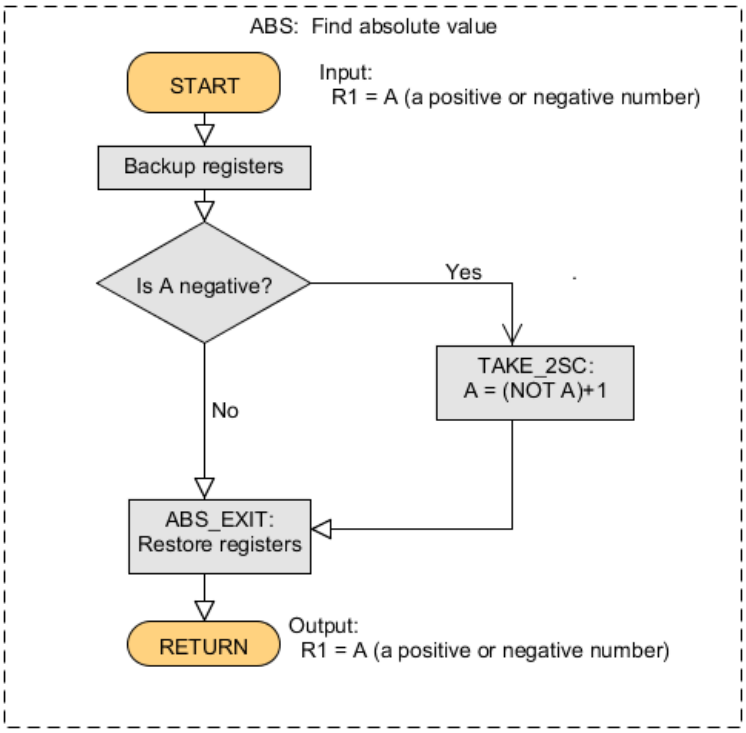
To help you meet our expectations for the subroutines you will design, code, and test, here is a quick overview of the elements of a subroutine, along with an example.

Subroutines are pieces of code that you can run independently of other code. You've already used a few: PUTS, GETC, and PUTC are all subroutines. When you call PUTS, you don't have to worry about how it will affect the rest of your code – it doesn't. PUTS changes R1 as it runs, but also changes it back before it exits.

Subroutines are really useful! They enable you to :

- re-use the same code in a variety of different places
- make your code more readable and compact
- test subroutine code separately from more complex projects

Here is a flowchart and some sample code for a subroutine called ABS, which takes the absolute value of a number. Notice that we wrote a file just for the purpose of testing ABS. Once we've run the tests, and are satisfied that ABS works, we can copy and paste it into a different bit of code.




```
; Multiply:  Subroutine to multiply two registers, with test code

.orig x3000
;-----------TEST CODE:  use breakpoints to observe results-----------
; Give R2 a test value (to make sure it doesn't get overwritten):
     AND R2, R2, #0
     ADD R2, R2, #5

; Find |-12|
     AND R1, R1, #0
     ADD R1, R1, #-12

     JSR ABS
     NOP  ; Breakpoint here:  At this instruction, R1 should be +12


; Find |+12|
     AND R1, R1, #0
     ADD R1, R1, #12

     JSR ABS
     NOP  ; Breakpoint here:  At this instruction, R1 should be 12
HALT
; --------------- END TEST CODE ------------------------------
```

```
; -------------------------------------------
;           ABS SUBROUTINE
; Inputs:
;   R1 = A      (positive or negative)
; Outputs:
;   R1 = |A|    (absolute value of input)
; -------------------------------------------

R2_ORIGINAL     .fill 0
SIGN_MASK  .fill x8000 ;a bitmask for the 2SC sign bit

ABS:
     ;Backup the registers we use:
     ST R2, R2_ORIG

     ;Is A negative?
     LD R2, SIGN_MASK
     AND R2, R1, R2
     BRZ ABS_EXIT    ;if sign bit is 0, A is not negative
     BRNP TAKE_2SC
          TAKE_2SC:
          NOT R1, R1
          ADD R1, R1, #1
          BR ABS_EXIT

     ABS_EXIT:
     ;restore registers:
     LD R2, R2_ORIG
RET
; -------- END OF ABS SUBROUTINE ------------

.end
```

Part 1: Accessing a private array via LOAD, STORE, and PRINT_ARRAY

First, read the specifications below for the LOAD, STORE, and PRINT_ARRAY subroutines. Make flowcharts for the subroutine.

AFTER you've made your flowcharts, make a file called "Cipher_Array.asm," and in it, implement your subroutines and CIPHER_ARRAY, along with some test code.

Your code will create an array, called CIPHER_ARRAY, of size 2x50 (two rows and 50 columns). One row should contain the user's plaintext, and one row should contain the encrypted text. But it must be a single array – that is, you should use `.blkw` exactly once (you will probably want to store a variable called ROW_LENGTH or something similar).

You may store CIPHER_ARRAY in row major form (that is, the elements of each row occupies 50 continuous addresses in memory) or column major form (elements of each column occupy continuous addresses). There are advantages and disadvantages to each approach.

CIPHER_ARRAY should be private, in the sense that your code only uses the LOAD, STORE, and PRINT subroutines to access the array. If you access the array from your main code, you will lose points!

The subroutines should follow these specifications:

- **LOAD:**
  - Inputs:
    - R1 = the index of a column of CIPHER_ARRAY (0-49)
    - R2 = the index of a row of CIPHER_ARRAY (0 or 1)
  - Outputs:
    - R0 = the contents of CIPHER_ARRAY[R1, R2]
  - LOAD must not change the contents of CIPHER_ARRAY.
  - LOAD must be call-safe, returning the contents of every register besides R0 to their values before the call.
- **STORE:**
  - Inputs:
    - R1 = the index of a column of CIPHER_ARRAY (0-49)
    - R2 = the index of a row of CIPHER_ARRAY (0 or 1)
    - R0 = an ASCII character to store.
  - Outputs: None
  - STORE will change the value in CIPHER_ARRAY[R1,R2] to the contents of R0.
  - STORE must be call-safe.
- **PRINT_ARRAY:**
  - INPUTS: None
  - OUTPUTS: None
  - PRINT_ARRAY will display the contents of CIPHER_ARRAY on the console. The result should be readable, aesthetically pleasing, and it should be clear which row is plaintext and which is encrypted.
  - PRINT_ARRAY should be call-safe, and should not alter the contents of CIPHER_ARRAY.

Test your subroutines thoroughly! Try loading and storing to various places in the array. Try loading and storing *outside* the array (that is, try something like STORE(500,500,500) and see what happens).

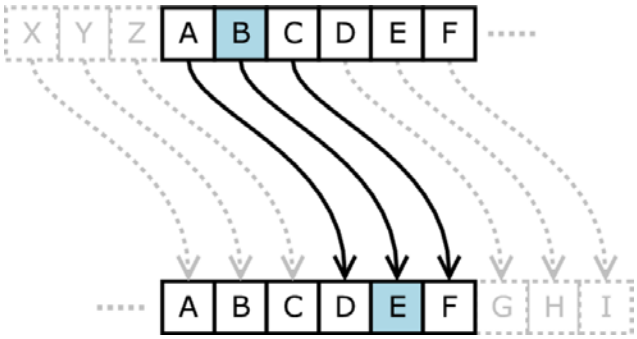
Part 2: Implementing the Caesar Cipher:

First, read the specifications for the ENCRYPT and DECRYPT subroutines. Make flowcharts for the subroutine.

AFTER you've made your flowcharts, make a file called "Encrypt_Decrypt.asm," and in it, implement your subroutines, along with some test code.

Make an LC3 file called "encrypt_decrypt.asm." In it, you will write and test two subroutines that implement a Caesar Cipher.

A Caesar Cipher is a very simple encryption. It operates on one character at a time, and shifts that character through the alphabet by a particular distance (a "key"), wrapping around at Z. For example, here is how a Caesar Cipher would map using a key of 3:



This is sometimes called "rotation by 3," or "ROT-3."

Note that the Caesar Cipher only encrypts letters of the alphabet, so punctuation marks are unaffected.

Here's an example of how you might encrypt a letter of the alphabet using a function in pseudocode:

```
encrypt(plaintext_char, key):
     if (plaintext_char is not in the alphabet):
          return plaintext_char
     else:
          encrypted_char = plaintext_char + key
          if (encrypted_char is not in the alphabet):
               return encrypted_char
          else:
               return encrypted_char - length_of_alphabet
```

It is up to you to implement this pseduocode in LC3, as well as figure out how to modify it for decryption.

You must write the following subroutines:

- **ENCRYPT:**
  - Inputs:
    - R0: A character to encrypt
    - Key: A number between 1 and 25 to shift R0 by
  - Outputs:
    - R0: The input character, EN-crypted by shifting RIGHT

    - ENCRYPT only encrypts letters of the alphabet. Lower-case letters should remain lower-case, uppercase letters should remain upper-case, and punctuation marks and numbers should be unaffected.
    - ENCRYPT should be call-safe, returning any registers besides R0 to their original values.
- **DECRYPT:**
    - Inputs:
        - R0: A character to encrypt
        - Key: A number between 1 and 25 to shift R0 by
    - Outputs:
        - R0: The input character, DE-crypted by shifting LEFT
    - ENCRYPT only encrypts letters of the alphabet. Lower-case letters should remain lower-case, uppercase letters should remain upper-case, and punctuation marks and numbers should be unaffected.
    - ENCRYPT should be call-safe, returning any registers besides R0 to their original values.

Be sure to test these subroutines carefully! Call the subroutine several times, printing the inputs and outputs each time. Test against various punctuation marks and values for the key. Make sure it works before you try to put it into a larger code system!

PART 3: Ceasar Cipher program:

Now that you've written and tested your subroutines, it's time to put them into a larger piece of code with a user interface. Read the spec below, and draw up a flowchart.

AFTER you've made your flowchart, make a file called "Lab_4.asm," and in it, implement your main code. Copy and paste your well-tested subroutines from their test-code files.

Your code should have a user interface that does the following:

- Print a prompt that asks the user to enter 'E' for encrypt, 'D' for decrypt, or 'X' to quit.
- 'X' results in a goodbye message followed by termination of the program.
- If the user enters 'D' or 'E':
    - Prompt for a cipher key (that is, a number between 1 and 25) and store the result.
    - Then, prompt for a message of up to 50 characters, and store the response in one row (or column) of a 2-D array.
    - Then, in the other row (or column), store the encrypted (or decrypted) version of the user's input.
    - Finally, display both the encrypted and plaintext contents of the array.
- After encryption/decryption, return to the original prompt.
- You may assume the user enters only valid input.

Here is an example of how interaction with this code could look:

```
Hello, welcome to my Caesar Cipher program!
Do you want to (E)ncrypt or D(ecrypt) or e(X)it?
> D
What is the cipher (1-25)?
> 13
What is the string (up to 200 characters)?
> Fpubby vf sha!!
Here is your string and the decrypted result
<Encrypted>: Fpubby vf sha!!
<Decrypted>: School is fun!!
> Do you want to (E)ncrypt or D(ecrypt) or e(x)it?
> X
Goodbye!!
```

Your main code should not interact with CIPHER_ARRAY directly. Instead, use STORE, LOAD, and PRINT_ARRAY to access the array. Similarly, your main code should not check characters for

To get some "fun" test input, do an internet search for "rot13 jokes." (Make sure you test other rotation values as well though!)

Deliverables:

Files to Submit on Canvas:

- **Lab5.asm**
- **Lab report**

Files to show to staff during checkoff:

- **A minimum of 6 flowcharts:**
  - **One for main code**
  - **One for each required subroutine**
  - **One for each other subroutine you create**
- **2 test code files:**
  - **Cipher_array.asm**
  - **Encrypt_Decrypt.asm**

As always, EVERY SINGLE DOCUMENT should have headers with your name, section, lab number, date, and TA name.

For checkoff, you must demonstrate this lab to the TA/tutor, either before submission or up to one week after the due date. Be prepared to answer questions about the design process and functionality of your program.

Points:

User Interface (prompts, get user input, exit): 10 pts

Create a single array of size 100, which is accessed as a 2x50 array by LOAD/STORE: 10 pts

PRINT_ARRAY works correctly, gives readable output: 5 pts

ENCRYPT correctly ciphers a character, is call-safe, and meets specifications: 15 pts

DECRYPT correctly de-ciphers a character, is call-safe, and meets specifications: 15 pts

Have complete, readable flow charts for the programs: 5 pts

Have required comments in the code: 5 pts

Lab Report: 10 pts