Assignment 3 — Traps and I/O

Due: Wednesday, January 31st

An encryption algorithm is one that alters data using one or more "keys", such that only those with the appropriate keys can recover and view the original data. Encryption algorithms are among the many, many algorithms that take some form of user input and produce some form of output.

The LC-3 supports one form of input, the keyboard, and one form of output, the terminal, and its OS provides service routines to accomplish common I/O tasks. For this assignment, you'll practice assembly I/O by implementing a simple encryption program that runs on the LC-3.

Deliverables:

Demo — Demonstrate your encryption program (running on the LC-3 simulator of your choice).

Write-up — *none*

Handin — `encrypt.asm`
GitHub Classroom Link: `https://classroom.github.com/a/Nnte43uN`

For this assignment, you may work alone or with *one* partner. However, the handin portion must be completed individually.

Part 1: Basic LC-3 I/O

Begin by accepting the GitHub Classroom assignment: `https://classroom.github.com/a/Nnte43uN`.

Take a quick read through `echo.asm`. This program simply reads characters you type on the keyboard and echoes them back to the LC-3 simulator's terminal, doing so indefinitely. It also demonstrates a number of features of LC-3 assembly programs, including labels, trap codes, and assembler directives.

As the file extension indicates, `echo.asm` is an assembly program, so its instructions are more easily read by humans than the machine language programs you've encountered so far. However, as an assembly program, `echo.asm` must be assembled before it can be run in an LC-3 simulator:

Windows Simulator Users:

1. Run `LC3Edit.exe` and open `echo.asm` within it.
2. Click the "asm" button in the toolbar. There should be no errors reported at the bottom of the window.
3. Verify that you now at least one[1] new file: `echo.obj`.
4. Run `Simulate.exe` and load `echo.obj` into the simulator.

Web Simulator Users:

1. Click the "Assemble" button.
2. Drag (or paste the contents of) `echo.asm` into the popup and click "Assemble". You should see a green field reporting no errors.
3. Click "Load into Simulator".

[1] LC3Edit will likely generate some other files, including `echo.sym`, the symbol table for the assembly program.

The most basic of LC-3 I/O service routines are `TRAP x20` and `TRAP x21`, known as "`GETC`" and "`OUT`" in LC-3 assembly, respectively. Recall that `GETC` reads a single ASCII character from the keyboard, storing it in `R0`, while `OUT` writes a single ASCII character from `R0` to the terminal.

If you have not already done so, run `echo.asm`. It won't appear to do anything unless you type characters into the LC-3's terminal (which appears as a second window in the Windows simulator and as a text box on the right in the Web simulator).

Every time you type a character into the LC-3's terminal, `echo.asm` reads that character into `R0`, then immediately writes it right back out.

Normally, when you type input using keyboard, you can see the characters as you type them. This is actually not a feature of keyboard input. Rather, the program taking input (your text editor, your web browser, your terminal...) is responsible for printing characters as they're typed. LC-3 programs are no exception.

Try removing the `OUT` instruction. Notice that even though you changed the number of instructions within the loop, you don't have to adjust the offset of the `BR` at the end of the loop — the assembler recalculates the label for you. Re-assemble the program, rerun it in the simulator, and you'll no longer see the characters as you type.

Part 2: A Simple Encryption Algorithm

Your program will take two pieces of input: an encryption key and a short message to be encrypted. The key will be used to alter the characters within the message.

The algorithm you are to implement is as follows:

Input: A numeric encryption key and a string to encrypt

Output: The encrypted string

---

1. Read the encryption key from the keyboard.
2. Read the message to encrypt from the keyboard.
3. For each character in the message, do:
   - **3a.** Toggle bit 0: if bit 0 contains a '`1`', replace that bit with a '`0`', and vice versa.
   - **3b.** Add the key to the result.
   - **3c.** Store the encrypted character.
4. Write the encrypted message to the terminal.

For example, suppose the chosen encryption key is 6, and an 'A' is typed:

- The ASCII code for 'A' is `0x41`, or `0000 0000 0100 0001` as a 16-bit binary value.
- The low bit is a one, so it will be flipped to a zero, producing `0000 0000 0100 0000`.
- The key, 6, will be added, producing `0000 0000 0100 0110`.
- That corresponds to `0x46`, the ASCII code for 'F'
- Thus, in the encrypted message, the 'A' will be replaced by an 'F'.

Part 3: The LC-3 Implementation

All right, let's expand on those steps in order to implement them as an LC-3 assembly program:

1. Prompt the user to type an encryption key.
   - This key will be single digit in the range $[1, 9]$.
   - You may assume that the user will provide valid input; you need not do any input error checking.
   - The user will *not* hit the "Enter" key here; as soon as you read the single digit, move on.
   - Note that `GETC` will place the ASCII value of a numeral in `R0` — you'll need to convert that character into an integer.
2. Read and store that digit. Whether you store it in memory or in a register is up to you.
3. Prompt the user to type a message to encrypt.
   - This message will be at most 20 characters long, not including a terminating newline.
   - Once again, you may assume that the user will provide valid input.
   - A newline, once read, should appear as `0x0A`[2].
4. Read and store the message in memory, one character per memory location.
   - For the sake of simplicity, you may allocate space for 21 characters (including the terminating null char), even if the user's message is shorter. Do not allocate space for more than 21 characters.
5. For each character in the message, do:
   5a. Toggle bit 0: if bit 0 contains a '`1`', replace that bit with a '`0`'; if it contains a '`0`', replace it with a '`1`'. Note that you are *not* `NOT`'ing the entire value, merely toggling the rightmost bit.
   5b. Add the key to the result. Note that, depending on the chosen key, characters whose ASCII codes are higher than `0x75` may be unprintable once encrypted — you don't need to handle this case.
   5c. Store the encrypted character back into memory. You may choose whether to overwrite the original message or store the encrypted message elsewhere.
6. Print out the encrypted message.

Sample Run:

```
Encryption Key (1-9): 2
Input Message: Asm > Java
Encrypted Message: Btn#A#Mbyb
```

(Where italicized characters are typed by the user and echoed back by your program. Note that your program's output will be tested using `diff`, so your output format *must* match exactly.)

Further Requirements:

- Write your program in a file named "`encrypt.asm`".
- Your program must start at memory location `0x3000`.
- Your program must use LC-3 traps to perform input and output.
- Your program must be able to be run multiple times by resetting the PC to `0x3000`. It should not require that the LC-3 be reinitialized, that any files be reloaded, or that any registers be reset.
- Your source code must include a comment with your name (and your partner's name, if applicable).

[2]Some simulators use `0x0D` as a newline. My automated tests will use the *nix command line simulator, which uses `0x0A`.