Assignment 3 — Assembly Languages

Due: Monday, October 12th

Assembly languages allow programs to be written in a format that is more human-readable than bare '0's and '1's, yet is still essentially a direct translation of machine code. Not only are the instructions themselves represented symbolically, but macros can be provided for common operations such as input or output.

Deliverables:

GitHub Classroom: `https://classroom.github.com/a/D0R66QeU`

Required Files: `encrypt.asm`

Optional Files: *none*

Part 1: An Encryption Scheme

A *Caesar Cipher*[1] encrypts a string by shifting each character by some fixed integer value, the *key*:

ENCRYPT(`string`, `key`)

Input: A string of ASCII characters, `string`, and an integer, `key`
Output: The result of shifting every character in `string` by `key`

1: **let** `result` be an empty string
2: **for** each character `c` in `string` **do**
3: Append (`c` + `key`) to `result`
4: **return** `result`

...for example, if `string` = "hello" and `key` = 1, then the encryption results in "ifmmp".

Part 2: Encrypting Strings

Complete the assembly program in `encrypt.asm`:

- Your program's assembly code must begin at memory location `0x3000`.
- Your program must prompt the user to type both a key and an unencrypted string, then print the corresponding encrypted string.
- It must be possible to rerun your program by manually resetting the PC to `0x3000`. It should not require that the LC-3 be reinitialized or that any files be reloaded.

You may assume that the key will be an integer between 0 and 9, inclusive, that the string will be at most 32 characters long, and that the user will hit the 'Enter' key after both the key and the string. You may further assume that all input will consist of valid, printable ASCII characters.

For example:

```
Encryption key (0-9): 3
Unencrypted string: Veni, vidi, vici
Encrypted string: Yhql/#ylgl/#ylfl
```

Your program will be tested using `diff`, so its printed output must match *exactly*.

[1] Although this cipher is simple by modern standards and easily broken by computers, it was likely effective against Julius Caesar's enemies — most of them were illiterate.

To help you get started, note the following caveats:

- When reading and writing characters and strings, remember that `GETC`, `OUT`, and `PUTS` all *assume* that the relevant information is in `R0`. You cannot specify any alternative registers for input and output; organize your program's data accordingly.
- When a character is typed, it will not automatically be echoed back to the terminal[2]. Your program must do this, so that the user can see what they typed.
- When the user types in the numerical encryption key, they are typing a *character*, not an *integer*. For example, if the user types '3', this will appear as the ASCII code `0x33`. In order to correctly perform the encryption, you must then convert this character into an integer[3].
- When the 'Enter' key is struck, it also counts as a typed character — its ASCII code is `0x0A`. This should not be considered part of the unencrypted input string.
- When adding a key to a character, it is generally possible that the resulting sum is $<$`0x20` or $>$`0x7E`. The character is then said to be *unprintable*: its code falls outside the range of an unsigned, 7-bit integer or has a special purpose[4]. In such cases, you must instead output '`?`' (the question mark, ASCII code `0x3F`) in place of the unprintable character.

Develop this program incrementally: make sure your program can read, store, and print back the unencrypted string first, then consider the actual encryption. In turn, make sure your program can encrypt simple strings before considering the check for unprintable characters.

Remember, a distinct advantage of assembly code is that it is much easier to extend or refactor than machine code — for example, you do not have to worry about recalculating all of the address offsets whenever an instruction is added or removed.

Part 3: Submission

The following files are required and must be pushed to your GitHub Classroom repository by the deadline:

- `encrypt.asm` — A working assembly program for encrypting strings, as specified.

The following files are optional:

- *none*

Any files other than these will be ignored.

[2] On a real computer, some combination of the shell, the operating system, or the terminal itself would handle this.
[3] Note that all of the Hindu-Arabic numerals have consecutive ASCII codes.
[4] For example, `0x07` is the *bell character*, commonly used to send warning sounds to terminals.