Assignment 4 — Subroutines

Due: Friday, February 9th

Assembly languages are not limited to monolithic programs containing instructions that simply run in sequence, one after another, with the occasional branch. Assembly languages can be used to build up all of the features we're familiar with from high-level languages.

For this assignment, you'll implement two examples of some of the most fundamental features of higher-level programming languages — callable code constructs and data structures — using LC-3 assembly: subroutines and arrays.

Deliverables:

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

Write-up — Answer the questions throughout this assignment. Fill out your answers *at the end* of this assignment, where the questions have been copied with space for answers.

Handin — `geti.asm`, `quiz.asm`, `data.asm`
GitHub Classroom Link: `https://classroom.github.com/a/tEWd5iMK`

For this assignment, you may work alone or with *one* partner. However, the handin portion must be completed individually. Remember to write both of your names on the write-up.

Part 1: LC-3 I/O and Traps

As you are probably aware by now, the LC-3 has two I/O devices, accessed via four specialized registers: the `KBSR`, `KBDR`, `DSR`, and `DDR`. Additionally, the LC-3's OS defines traps, service routines that read to and write from these specialized registers in order to perform common I/O tasks.

1. Recall that the `KBSR` is memory-mapped to location `0xFE00`. Is the `KBSR`'s interrupt bit enabled?

Traps are simply reusable sequences of instructions, and the starting address of a trap — the address of the first instruction in its sequence — is stored in the Trap Vector Table.

Recall that the Trap Vector Table is an array in memory, from locations `0x0000` through `0x00FF`, whose indices are trap codes.

2. At what address do the instructions for `TRAP x21` begin?

3. Briefly explain, line by line, the purpose of each instruction for `TRAP x21`.

Part 2: A "`GETI`" Subroutine

Begin by accepting the GitHub Classroom assignment: `https://classroom.github.com/a/tEWd5iMK`.

In your previous assignment, a portion of your program read one numeric character from the keyboard and converted that character into an integer. This is a fairly common task, so it's a decent candidate for conversion into a subroutine.

Develop an LC-3 subroutine that reads a single character from the keyboard, converts it into an integer, and returns that integer in `R0`.

Requirements:

- Define your subroutine in a file named "`geti.asm`", beginning at memory location `0x3300`.
- Your subroutine must be a complete assembly file (including `.ORIG` and `.END` directives).
- You may assume that the user will type a single integer in the range $[0, 9]$.
- You may not call any traps within your subroutine; implement polling in order to perform I/O.
- Your subroutine must echo the typed character to the terminal.
- Your subroutine must save and restore any registers it uses other than `R0` and `R7`.
- Your source code must include a comment with your name (and your partner's name, if applicable).

Thoroughly test your subroutine! Your assignment repository includes `getiDriver.asm`. You should be able to assemble both `geti.asm` and `getiDriver.asm`, load both `geti.obj` and `getiDriver.obj` into the simulator, set the `PC` to `0x3000`, and run the driver program.

This driver will call your subroutine and, if all goes well, echo back any typed numbers to the keyboard.

Part 3: Using `GETI` As a Trap

Traps call service routines, which are essentially nothing more than subroutines defined by the OS. Thus, traps can be added to the LC-3 by adding entries to the Trap Vector Table.

- Alter `getiDriver.asm` to call `TRAP x26`, rather than loading the address of your subroutine and performing a `JSRR`.
- Reassemble `getiDriver.asm` and load both `geti.obj` and `getiDriver.obj` into the simulator.
- Set memory location `0x0026` to the value `0x3300`. This adds an entry in the Trap Vector Table for `TRAP x26`, pointing to the first instruction of your "`geti`" subroutine.
- Verify that `getiDriver` still runs as expected.

Part 4: Arrays in LC-3 Assembly

Thus far, you've probably seen the data structure known as an "array" in two contexts: Firstly, in high-level languages such as Python and Java, where the implementation of an array is largely hidden from you. And secondly, in this course, where we've said that "memory is like a giant array whose indices are addresses."

As you'll see when we introduce C, an array in C is stored in memory as a contiguous block of memory locations. A variable referencing an array in a C program is nothing more than the address (or "pointer", to use the more specific terminology) of the beginning of that block of memory.

The same implementation can be applied to LC-3 assembly. We can, for example, declare an array of length ten by allocating ten contiguous locations in memory:

```
ARR .BLKW #10    ; Analogous to   "arr = [None] * 10"   in Python.
```

That array can then be assigned to a register by placing its address into a register:

```
LEA R0, ARR      ; Analogous to   "R0 = arr"   in Python.
```

Indexing, then, is simply the process of offsetting and loading that address:

```
LDR R1, R0, #2   ; Analogous to   "R1 = R0[2]"   in Python.
```

We'll use this method of declaring and indexing arrays for the following LC-3 program, organizing and accessing data in a predictable manner in order to make your program more flexible and easier to read.

Part 5: An LC-3 Quiz Program

Let's put that all together now by implementing a simple multiple choice quiz program. Your program will pose three questions to a user and read the user's responses. Each answer will be associated with a point value, and your program will print out a message based on the total points earned.

This program isn't going to be much longer than those you've written for previous assignments, but it is a significant conceptual step up. Here's how to get started:

1. Take a look at `sampleData.asm`. This is the format of the data your program must consume. Notably:
   - Each question is represented by two values: one string, plus an array of four integers — the question and the point values of its answers.
   - Each answer is worth between 0 and 10 points.
   - Notice, however, that the strings themselves are declared at the bottom of the file, while their addresses are stored above.
   - The LC-3's primitive assembler doesn't recognize labels from other files. This means that your quiz program can't use the labels from this data file; the addresses must instead be hardcoded.
   - Since the addresses are hardcoded, each piece of data must be in a predictable location, regardless of which data file is used. This means that the strings, which are variable in length, have to be moved to the end, so that we can accurately predict the addresses of the data above them.
   - The same is done for the result strings: the strings themselves are at the bottom of the file, while the "`RESULTS`" array contains addresses of strings.
2. That is to say:
   - Your program *must* be able to use data in this format.
   - All data files will match this format *exactly*.
   - `Q1STR` will *always* refer to `0x3500`, `Q1PTS` will always refer to `0x3501`, `Q2STR` will always refer to `0x3505`, `RESULTS` will always refer to `0x350F`...
3. But fear not! Check out `quiz.asm` — I've given you a starter file for this assignment that already includes examples of loading data from the data file. Quite conveniently, it also already includes labeled locations containing all of the addresses you should need from the data file.
4. Fill in `quiz.asm`. Your program should:
   (a) Display and prompt the user to answer Question 1.
      - You may use the LC-3's existing traps for this.
   (b) Read the user's answer from the keyboard.
      - Use your `GETI` trap, `TRAP x26`, to do this.
      - You may assume that the user will enter an integer in the range $[1, 4]$.
   (c) Determine the point value of the user's response.
   (d) Maintain a running total of the user's points.
   (e) Repeat for Question 2 and Question 3.
      - Consider writing some subroutines (within `quiz.asm`) to reduce code duplication.
   (f) Output a message based on the user's total score:
      - If the user's score is greater than or equal to 27, print the first message of the `RESULTS` array.
      - If the user's score is between 21 and 26, print the second message of the `RESULTS` array.
      - If the user's score is less than or equal to 20, print the third message of the `RESULTS` array.

5. If all goes well, you should be able to test your program:

   - Assemble and load `sampleData.asm` into the simulator.
   - Assemble and load `geti.asm` into the simulator.
   - Assemble and load `quiz.asm` into the simulator.
   - Set memory location `0x0026` to the value `0x3300`.
   - Check that the `PC` contains `0x3000`.
   - Run your program to completion.

   Sample Run (using `sampleData.asm`):

   ```
   In the Von Neumann Model, what is memory?
       1) An array indexed by addresses
       2) A hard disk drive
       3) An input and output device
       4) A series of punchcards
   Answer: 1
   What type of instruction is LEA?
       1) Computational
       2) Data movement
       3) Control
       4) Assembler directive
   Answer: 2
   What is the max of an unsigned 16-bit value?
       1) 256
       2) 32767
       3) 65536
       4) 65535
   Answer: 4
   Result: Looks like I should make the midterm harder.
   ```

   (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.)

6. You're almost done! To prove the flexibility of your program, create your own quiz by creating a new data file.

   - Name your new data file "`data.asm`".
   - Your new data file must follow the same format as `sampleData.asm` (I recommend simply copying the existing file, then changing the strings and point values.).
   - You should be able to run this new quiz *without* any modifications to `quiz.asm`.

Further Requirements:

- 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).

Part 1: LC-3 I/O and Traps

1. Recall that the `KBSR` is memory-mapped to location `0xFE00`. Is the `KBSR`'s interrupt bit enabled?

2. At what address do the instructions for `TRAP x21` begin?

3. Briefly explain, line by line, the purpose of each instruction for `TRAP x21`.