CS061 - Lab 3

Loops & Arrays

1 High Level Description

Today's lab will introduce arrays: constructing and traversing them with both counter-controlled loops and (new) sentinel-controlled loops.

2 Our Destination This Week

1. Lab 2 review
2. Pseudo-ops revisited
3. More indirection! Exercise 1
4. Hurray for Arrays! Exercises 2 - 3
5. More loopiness! Exercise 4
6. So do I know everything yet??

3 Processing data

Lab 2 review
If there is anything you don't yet understand about the differences between **LEA**, **LD/ST**, **LDI/STI**, and **LDR/STR** - ask now!

Remember to always open the Text Window of your simpl LC-3 emulator!

Exercise 0

Study the Intro to LC-3 i/o.
Now write a program that takes a single character from the keyboard, using Trap x20 (aka GETC).

Whenever you GETC, you must always immediately OUT (Trap x21), to echo the captured character *- otherwise the user has no feedback to confirm what key they just typed (we call that "ghost typing").*
The *only* exception to this might be when capturing a password, when you don't want to echo the characters for security reasons *(another approach is to echo every character with '*')*.

Using the simpl emulator, "step" through the program and examine the contents of R0 immediately following the GETC routine: you will see that LC3 stores each character as an 8-bit ascii code in the lower byte of a 16-bit word, setting the upper byte to 0 *(check an ascii table to confirm the code for the character you typed)*.

3.1 Pseudo-ops revisited

All assembly languages have several “pseudo-ops”, assembler "directives" that instruct the assembler program how to set up the object code - just like C++ compiler directives like `#include`
A quick review: read LC-3 Pseudo-ops summary in the LC-3 Resources folder.

You encountered most of the LC3 pseudo ops already in lab 1:
.ORIG tells the assembler where to start loading the code *(usually x3000, except when we tell you otherwise)*;
.END tells the assembler that there is no more code to assemble (like the `'}'` after main() in C++)
.FILL stashes a single “hard-coded” value into a memory location; you can specify it as a decimal (#), hex (x), or ascii character (use `' '`, e.g. `'a'` or `'\n'`)
and .STRINGZ creates a c-string, i.e. a null-terminated character array. You may use \n inside a string to insert a newline at any point.

The last of the pseudo-ops we'll need is .BLKW *(“block words”)*: it tells the assembler to set aside **n** memory locations for later use. Note that it does not explicitly "zero out" the designated memory space, so there may be "junk values" in those locations if they have been used previously.
We'll be using this pseudo-op in today's exercises.

Example:

```
ARRAY_1   .BLKW  #10     ; Reserves 10 memory locations, starting at address ARRAY_1
DEC_25    .FILL #25      ; stores the value #25 at the memory address labelled DEC_25
                         ; this will be located at the first address following ARRAY_1
```

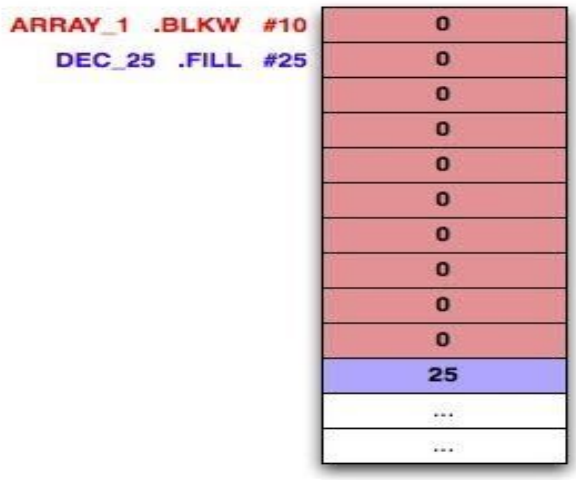

Figure 1: .BLKW Illustration

Note that even though the label DEC_25 immediately follows label ARRAY_1 in the code, its address is **(address of ARRAY_1) + #10**
Note also that the ten memory locations are not guaranteed by the LC3 specs to be initialized to 0.

We need to know the actual starting address of an array - in this example, the address named ARRAY_1
In lab 2, we did that by placing the array at a specified remote address, and then hard-coding that address in a local pointer.
This time, we will be setting up an array in the local data block. To get the starting address of a local array, you will need to understand the instruction **LEA**, introduced back in lab 1 (See the LEA tutorial).

3.2 More indirection!

Exercise 1
Let's take another look at Exercise 3 from last week.

At the time, we told you to hard code both remote addresses in the local data block – but this is not really very efficient *(suppose you had a dozen data values up there ...)*

We know that the two data values are in contiguous memory locations (x4000 and x4001), so we ought to be able to just hard-code the first address and then use *address arithmetic* to get to the next.

So copy your Lab 2 Exercise 3 from last week into the lab3_ex1.asm file in your current directory, but this time use just a *single* pointer in the local data block, and .FILL the address of the start of the remote data block (your remote .ORIGIN). Call the local pointer DATA_PTR.
You already know how to get the first data item from the remote location (you did that in lab 2): how will you get the next?

Hint: LDI won't work, because we don't have a pointer to it in the remote memory location.

When you have figured this out, run the code (read the two remote data values, modify them, & store the modified data back to the remote location).
As in lab 2, manually inspect the memory in simpl to confirm.

3.3 Hurray for Arrays!

Exercise 2
The technique you just discovered *(if you didn't, go back to exercise 1, and don't join us here until you've figured it out!)* is a standard mechanism for constructing and traversing *arrays* – not just in LC3 programming, but in *all* languages.

So now write a program (lab3_ex2.asm) that builds and populates an array in the *local* data area:

- Use .BLKW to set up a blank array of 10 locations in the *local* data area (no remote data this time)
- This is where you will have to use the LEA instruction to get the actual starting address of the array!
- Then have the program prompt the user to enter exactly 10 characters from the keyboard, and populate that array with the characters as they are input.

This will require a counter-controlled loop (DO-WHILE loop), which you should already know how to construct. *(Check here if you need a refresher!)*
Test your program, inspecting the contents of the array in simpl after every character input.
*(By the way: you should now understand one of the deep mysteries of C++ - why the number of elements in a static array variable **must** be known at compile time, and can't be changed after that).*

Exercise 3
First, make sure you have read the basic i/o doc
Now copy exercise 2 into lab3_ex3.asm, and extend it by adding another loop to traverse the array again, and output each stored character to the console, one per line (i.e. print a newline '\n' = ASCII x0A - after each character).
Your program will now be complete: it will accept exactly 10 characters from input, storing them one by one in a remote array, and then output the whole list to console.

3.4 More loopiness!

Exercise 4
In the previous exercises, you were able to traverse the arrays because you knew up front how many elements were in them – in fact that number was hard-coded into the .BLKW pseudo-op.

Can you think of a way to create and/or traverse an array without knowing beforehand how many elements it will contain, and without using .BLKW?
Hint: think about the difference between counter & sentinel control of loops.

Another hint: we will actually need *two separate sentinels!* (see below for detail)

First, you have to use a specific key to tell the program to stop collecting characters from the keyboard; *then* you have to store a sentinel in the array to mark the end of the collected

characters.
Now copy Exercise 3 into your lab3_ex4.asm file, and modify it as follows:

- It must capture a sequence of characters as long as you like *(within reason - for now, let's keep our tests to less than 100)*, and stop when you enter your "input" sentinel character.
- The program will store each character as it is entered in an array starting at x4000 *(we will just assume for now that there is a vast amount of free memory up there)*, with a sentinel character marking the end of the data; and then output them to the console in a separate loop.
- Remove the newline after each character output - i.e. the entire array will now be output on a single line. DO terminate that line with a newline.

There are three separate problems to be solved here:

- How do I communicate to the program that I have finished input?
  Hint: what is the most common keyboard method of signaling "I'm done with this message" in, say, Facebook chat?
- How do I build the array so that it "knows" where it stops - i.e. what sentinel do I store at the end of the array? This can be different from your "input" sentinel.
  Hint: how does .STRINGZ do it?
- How does the program know when to stop traversing the array for output?
  See previous hint, and think PUTS!

Once you solve these problems you will have in your algorithmic toolkit a very powerful technique – *sentinel-controlled loops*—that you will be using to manage i/o for the rest of the course.

3.5 Submission

Demo your lab exercises to your TA ***before you leave lab***.

If you are unable to complete all exercises in lab, show your TA how far you got, and request permission to complete it after lab.
Your TA will usually give you partial credit for what you have done, and allow you to complete & demo the rest later for full credit, so long as you have worked at it seriously for the full 3 hours.
When you're done, demo it to any of the TAs or instructors in office hours ***before your next lab***.
Office hours are posted on Piazza, under the "Staff" tab.

4 So what do I know now?

You should now know:

- How the .BLKW pseudo-op works to reserve memory locations
- How to use .BLKW to build arrays, and use counter-controlled loops to traverse them
- How to test for a specific value, and use such tests in sentinel-controlled loops.
- How to build arrays without .BLKW, and use sentinel-controlled loops to traverse them.
- How to use LEA to get the actual address corresponding to a label.
- How to use prompts to communicate with the user.
- How to do basic i/o
- Did I mention that sentinel-controlled loops are very, very important?