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 02 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 02 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

Write a program that takes a single character from the keyboard, using Trap x20 (the GETC BIOS routine).
<u>NOTE</u>: *whenever you GETC, you must always immediately OUT, to echo the captured character* - *otherwise the user has no feedback to confirm what key they just typed (we call that "ghost typing").*
The <u>only</u> 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, examine the contents of R0: 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" (i.e. assembler "directives" that instruct the assembler program how to set up the object code - just like the C++ compiler directives such as `#include`).
If you have not done so already, review the [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")*, which is a bit like .STRINGZ without any data: it tells the assembler to simply set aside **n** memory locations for later use. 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 vale #25 at the memory address labelled DEC_25
                        ; this will be located at the first address following ARRAY_1
```
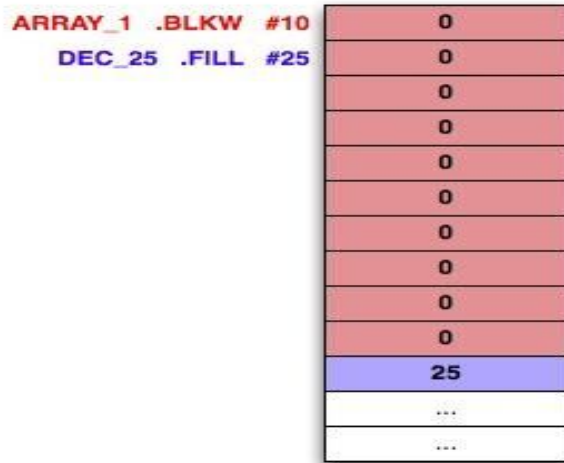

Figure 1: .BLKW Illustration

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 <u>not</u> guaranteed by the LC3 specs to be initialized to 0.

3.2   More indirection!

<u>Exercise 1</u>
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 <u>Lab 2 Exercise 03</u> from last week into the lab03_ex1.asm file in your current directory, but this time use just a *single* pointer in the local data block. Call it DATA_PTR.
You know how to get the first data item from the remote location: how will you get the next?
Hint: LDI won't work, because we don't have a pointer to it in an accessible memory location.

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 (lab03_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.
● 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
Now copy Exercise 02 into lab03_ex3.asm, and complete it by adding another loop to traverse the array again, using your knowledge of i/o to _output_ each stored character to the console, <u>one per line</u> (i.e. print a newline '\n' = ASCII x0A - <u>after each character</u>).

Your program will now be complete: it will accept exactly 10 characters from input, storing them one by one in an 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.

Copy Exercise 3 into your lab03_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 decide when to stop.
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)*, and then output them to the console in a separate loop.

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?
  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 this problem 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 demo in office hours: you will have 1 week to do so.

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 prompts to communicate with the user.
● Did I mention that sentinel-controlled loops are very very important?