CS061 - Lab 04

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 03 review
2. Pseudo-ops revisited
3. More indirection! Exercise 01
4. Hurray for Arrays! Exercises 02 - 03
5. More loopiness! Exercises 04 - 05
6. So do I know everything yet??

3 Processing data

Lab 03 review
If there is anything you don't yet understand about the differences between **LEA**, **LD**, **LDI**, and **LDR** - ask now!

Exercise 0
Write a program that takes a single character from the keyboard, using Trap x20 (GETC).
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, and sets the upper byte to 0 *(check your ascii table to confirm the code for the character you typed)*.
Now add TRAP x21 (OUT) to your code.
What does this do?
You don't need to submit this exercise - it's just to get you started.

3.1 Pseudo-ops revisited

All assembly languages have several "pseudo-ops" (i.e. assembler directives that instruct the assembler program in how to set up the object code). If you have not done so already, review the LC-3 Pseudo-ops summary in the LC-3 Resources folder (Course Resources page).

Already in lab 1, you encountered most of the LC3 pseudo ops:
.ORIG tells the assembler where to start loading the code;
.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
and .STRINGZ creates a c-string, or null-terminated character array.

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.
We'll be using this pseudo-op in today's exercises.

Example:
```
ARRAY_1    .BLKW   #10     ; Reserves 10 locations, starting at address ARRAY_1
DEC_25     .FILL #25       ; stores the vale #25 at the memory address labelled DEC_25
```

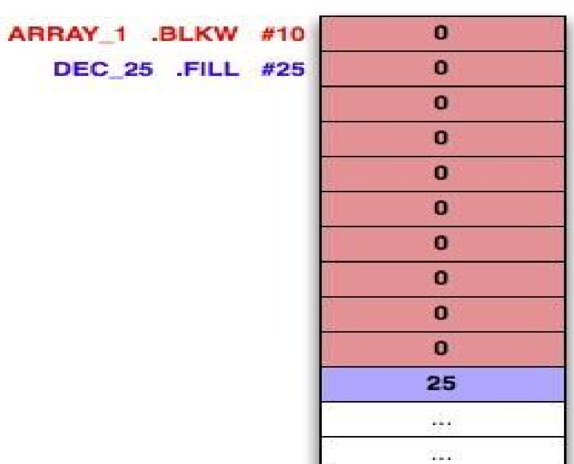

Figure 1: .BLKW Illustration

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 01</u>
Let's take another look at Exercise 03 from last week *(I hope you kept it!)*

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 that to get to the next.
Well, with a judicious combination of data movement and arithmetic operations, you can do just that.

So rewrite Exercise 03 (call it lab04_ex1.asm) from last week, but this time use just a <u>*single*</u> 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!

<u>Exercise 02</u>
The technique you just discovered *(if you didn't, go back to exercise 01, 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 (call it lab04_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.

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 ***<u>must</u>*** *be known at compile time, and can't be changed after that).*

<u>Exercise 03</u>
Now complete the previous program by traversing the array again (i.e. add another loop), using your knowledge of i/o to output each stored character to the console, one per line (i.e. print a newline - 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 an array, and then output the whole list to console.
Call it lab04_ex3.asm

3.4 More loopiness!

<u>Exercise 04</u>
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.

Re-write exercise 03 (in a new file called lab04_ex4.asm) so that you can input a sequence of characters as long as you like *(within reason - for now, let's keep it to less than 100)*, and decide when to stop. The program will store them in an array starting at x4000, and then output them to the console.

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 – <u>sentinel-controlled loops</u>—that you will be using to manage i/o for the rest of the course.

3.5 Submission

Add, commit, and push your lab04_ex1.asm through lab04_ex4.asm files to your lab 4 GitHub repo.

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?