CS061 - Lab 03

1 High Level Description

Today's lab will cover some new LC3 instructions, introducing the marvels of *memory addressing modes*, and some more I/O (input/output),

2 Goals for This Week

1. New LC3 instructions using memory addressing modes direct, indirect, and relative.
2. Exercises 1 – 4:
   Practice using the three memory addressing modes direct, indirect & relative;
   Building a simple loop using the conditional branch instruction; and
   Simple input/output using BIOS routines.
3. So what now??

3.1 New LC3 Instructions

As you know, in the LC3, values can be stored in system memory (RAM), or in the eight general purpose registers named R0 through R7, or in several special purpose registers *(more about this later on)*.
It might not be obvious yet, but ***all*** microprocessor instructions involve getting values from some of those places, manipulating them, and writing the result to another place - so we need a way to describe this process.

From now on, we will use a standard notation to describe all such actions called
Register Transfer Notation (RTN):

- (Rn) means "the *contents* of Register n" - i.e the value currently stored in that register.
- Mem[xnnnn] means "the *contents of memory address* xnnnn"
- ⟵ means "write the ***value*** on the right of the arrow to the ***location*** on the left"
  (the location on the left may be either a memory address or a local register), so:

  `R3 ⟵ Mem[ x3042 ]`
  means copy the *contents* of memory location x3042 to *Register 3*.

  `Mem[ x3120 ] ⟵ (R6)`
  means copy the *contents* of Register 6 to the *memory location x3120*

  `R3 ⟵ (R1) + (R2)`
  means add the *contents* of R1 and R2, and write the result to R3.

  Also, as we saw last week, a label is an alias for a memory location (an address), so:
  `Mem[ myAnswer ] ⟵ (R4)`
  means copy the *contents* of R4 to the *memory location* corresponding to the *label* myAnswer.

New LC3 instructions this week: ST, LDI, STI, LDR, STR, Trap
Note: further details on all LC-3 instructions can be found here and in Appendix A of the text.

The ST (Store) instruction - see the ST tutorial (Piazza Resources -> LC3 Instructions)
This is simply the opposite of the LD instruction: it stores the contents of a register directly into a location in memory specified by a label *(overwriting any previous content, of course).*

Before we go on to the other memory addressing modes, we need some background.
Consider the instruction:

LD R1, ADDR_1 ; as you now know, this means R1 <- Mem[ADDR_1]
Let's say that ADDR_1 is an alias for memory address x3100;
and that the value stored at that address is x4A20 = #18976 = b0100 1010 0010 0000
In other words: **`R1 ⟵ Mem[ x3100 ]`** means **`R1 <- x4A20`**

However, for reasons that will become clear later on, the label ADDR_1 ***may be no more than +/- 256 locations away from the LD/ST instruction!***
if you attempt to LD or ST with a label further away than that, you will get an assembly error:
"Instruction references label that cannot be represented in a 9-bit signed PC offset" (see here).
So we need a mode that gets around that limitiation - and, in fact, we have two such modes.

The LDI (Load Indirect) instruction - see the LDI tutorial
This is looks a bit like the LD instruction, except that there is one level of *indirection*:

`LDI R1, ADDR_1 ;` loads Mem[Mem[ADDR_1]] into R1:

That is, it first fetches Mem[ADDR_1] *(let's use ADDR_1 = x3100; and Mem[ADDR_1] = x4A20 again)*. Then it uses that value as a ***new address***, and fetches Mem[x4A20] *(i.e. the value stored at x4A20 – let's say that is x00C4)*, and finally copies that value into R1:

`R1 ← Mem[ Mem[ADDR_1] ]  i.e.` **`R1 ← Mem[ x4A20 ]`** `i.e.` **`R1 ← x00C4`**

This round-about process is called "**indirection**", and the label is called a "**pointer**" *(sound familiar?),* and it allows us to get around the +/- 256 line limitation *(although the label itself must still be within range)*

The STI (Store Indirect) instruction - see the STI tutorial
This works just like LDI, only in the opposite direction.
It takes the value from a register and copies (stores) it to memory at the address given by Mem[someLabel] *(overwriting previous content).*

`Mem[ Mem[ ADDR_1 ] ] ← (R1)  =>` **`address x4A20 ← (R1)`**

(i.e. the value currently stored in R1 is copied to the address stored in ADDR_1 = x3100, which is x4A20)

The LDR (Load Relative) instruction - see the LDR tutorial
This uses indirection like LDI, except that it uses a *register* as the pointer rather than a memory address:

The instruction

`LDR R1, R6, #0` *; For this course, we will always set the offset to 0*

copies the value from Mem[(R6) + offset 0] into R1:
(let's say (R6) = x3100, and Mem[x3100] is again x4A20)

`R1 ← Mem[ (R6) ]  i.e.` **`R1 ← Mem[ x3100 ]`** `i.e.` **`R1 ← x4A20`**

The STR (Store Relative) instruction - see the STR tutorial
This works just like LDR, only in the opposite direction.
In the example, the instruction stores the value from R1 into the address stored in R5 (x3100):

The instruction

`STR R1, R6, #0`

copies the value currently stored in R1 into the address stored in R5:
(Again, let's say (R6) == x3100, as above)

`Mem[ (R6) ] ← (R1)  i.e.` **`address x3100 ← (R1)`**

By convention, we usually use ***R6*** *as a "Base Register", i.e. the pointer register in LDR/STR instructions.*

The Trap instruction - see Table 1 below and the TRAP tutorial and p. 543 in the text.

Trap instructions are actually calls to a set of BIOS subroutines (a bit like what C++ calls "functions"). You saw one of them (PUTS) last week. You may use either the "TRAP xnn" form or the alias in your code.

Table 1: Trap instructions

Invocation	**Alias**	**Effect**
TRAP x20	GETC	R0 ← one character of input (ascii) from the keyboard (no echo)
TRAP x21	OUT	print (R0) to the screen as an ascii character
TRAP x22	PUTS	Jump to memory location (R0) and print the contents of that and each succeeding memory location until a 0 is encountered (used for printing strings)
TRAP x23	IN	***You will never use this instruction!*** Prompt the user to input a character; get the input from the keyboard in R0, and echo it to the console
TRAP x25	HALT	Terminates the program

3.2 Exercises 1 – 4

REMINDER: Always open the Text Window of your simpl LC-3 emulator!
(by checking the "Text Window" box at the bottom as soon as you open simpl)

Exercise 01: Direct memory addressing mode

Write an assembly language program that uses .FILL pseudo-ops to load the values #65 (decimal 65) and x41 (hexadecimal 41) into two memory locations with the labels DEC_65 and HEX_41 respectively.
Look up an ASCII table (use the back of your text or this table to see what these values represent when interpreted as characters rather than numbers.

Then use the **LD** instruction to load these two values into registers R3 and R4 respectively.
Run the program, and inspect the registers to make sure it did what you expected.

Exercise 02: Indirect memory addressing mode

Sometimes, the data we need is located in some remote region of memory.
Copy the program from exercise 01 into your lab-3_ex2.asm file and replace the data stored at DEC_65 and HEX_41 with two far away ***addresses*** such as x4000 and x4001 respectively, and rename the labels to DEC_65_PTR and HEX_41_PTR *(because your labels are now pointers)*.

You can load the data into the new locations by putting this at the end of your current data block, before the .END pseudo-op:

```
            ;; Remote data
            .orig x4000
NEW_DEC_65 .FILL #65
NEW_HEX_41 .FILL x41
```

But now we have a problem!
As we have already seen, the LC3 Direct memory addressing mode (LD, ST instructions) only works with labels that are within +/- #256 memory locations of the instruction *(see here for full details)* – so our new data locations are too far away from the code to be accessed with LD and ST, even though we have provided new labels for them.
Try using the LD instruction with these new labels and see what happens!

But never fear – the **indirect** and **relative** addressing modes will come to the rescue!
Replace the LD instruction with one using **LDI** to load the data into R3 and R4
(Hint: use the _PTR labels, not the NEW_ ones – in fact you can now remove those, they are of no use!)

Next, add code to increment the values in R3 and R4 by 1.
(What ascii characters do they correspond to now?)
Finally, store the incremented values *back* into addresses x4000 and x4001 using **STI**.
Again, inspect the registers to make sure it worked as expected!

Exercise 03: Relative memory addressing mode

Use your lab03_ex3.asm file
Start with the same setup as in exercise 02: You have two labeled locations ("pointers"), which contain two "remote" memory addresses; your actual data is stored at those remote addresses.

Directly load (**LD**) the values of the *pointers* into R5 and R6 respectively.
Now, use the relative memory addressing mode (**LDR**) to load the remote data into R3 and R4, using R5 and R6 as "Base Registers" - i.e. the registers that hold the memory pointers.

Perform the same manipulation as in exercise 02 – i.e. increment the values in R3 & R4, then store those incremented values back into x4000 and x4001, this time using **STR** *(inspect your registers to confirm, as always)*.

SUCCESS!!

You have now used the two memory addressing modes indirect *(LDI, STI)* and relative *(LDR, STR)* to accomplish exactly the same goal – each uses indirection ("pointers") to access memory locations that were too far away to be reached by LD:

The Indirect memory addressing mode uses a *label* (aliased memory location) as a pointer.
The Relative memory addressing mode uses a *register* containing a memory address as a pointer.
(The downside of both is that they require one more memory read than direct addressing).

Exercise 04: Loops

Use the conditional branch instruction (**BR**) to construct a *counter-controlled loop*.

Hard-code *(i.e. use .FILL)* local data values x61 and x1A, and load them into R0 and R1 respectively.
Inside a loop, output to console the contents of R0 (Trap x21, or OUT), then immediately increment R0 by 1.
Use R1 as the loop counter – i.e. the loop should be executed exactly x1A times.
Think carefully about how and when to terminate the loop: do you use BRn or BRnz or BRz or BRzp or ??

*What does this loop do? Can you figure it out **before** you run it?*

3.3 Submission

Add, commit, and push your lab03_ex1.asm through lab03_ex4.asm files to your lab 3 GitHub repo, and demo to your TA.
Note: to "stage" multiple files (i.e. "add", get all your modified files ready for full inclusion in your project), you can do either:

```
git add *.asm  ; this will stage all asm files
```

or

```
git add .      ; this will stage ALL modified files
```

or

```
git add -A     ; ditto
```

4 So what do I know now?

You should now ...

- Be totally familiar with the way of describing instructions, Register Transfer Notation (RTN)
  This will become fundamental to our understanding of how a microprocessor functions, which is the ultimate purpose of the entire course – so make sure you get the hang of it!
- understand the three LC3 memory addressing modes – direct, indirect, and relative
- master the LD/ST, LDI/STI, LDR/STR instructions, and the concept of indirection (pointers) – specifically, how to load from and store to locations in memory that are too far away from your code to be reached by the direct addressing mode (LD/ST).
- know how to build a simple counter-controlled loop
- Be able to use all the bios routines – i.e. the Trap instructions that manage console i/o: GETC, OUT, and PUTS