# 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).*

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

   Some background: As you know already, the instruction
         LD  R1,  ADDR_1
   gets Mem[ ADDR_1 ] *(i.e. the value stored at the label ADDR_1 - let's say it is x3100)* directly,
   and loads (writes) that value into R1:
         R1 ← Mem[ ADDR_1 ]    =>  **R1 ← x3100**
   *Note that the label ADDR_1 must be closer than +/- 256 locations from the 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 - in fact, we have two such modes.*

The "Load Indirect" instruction
        LDI   R1,  ADDR_1
instead, loads Mem[ Mem[ADDR_1] ] into R1:
That is, it first fetches Mem[ ADDR_1 ] *(let's use the same value as above, x3100)*.
Then it uses that value as a **new address**, and fetches Mem[ x3100 ] *(i.e. the value <u>stored</u> at x3100 –
let's say it is #12)*, and finally loads that value into R1:
        R1 ← Mem[ Mem[ADDR_1]  ]   i.e.   R1 ←   Mem[ x3100 ]   i.e.   **R1 ← #12**

This round-about process is called "**indirection**", and the label is called a "**pointer**" *(sound familiar?),*
and it allows to get around the +/- 256 line limitation


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)    => **x3100 ← (R1)**   *(i.e. the value stored in R1)*


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

The instruction
        LDR   R1,  R5,  #0   ; *For this course, we will <u>always</u> set the offset to 0*
loads the value from Mem[ (R5) + offset 0 ]  into R1:  *(let's say (R5) == x3100, and Mem[ x3100 ] is #12)*
        R1 ← Mem[ (R5) ]   i.e.   R1 ← Mem[ x3100 ]   i.e.   **R1 ← #12**


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 (x3007):

The instruction
        STR   R1,  R5,  #0
copies the value from R1 *(let's say #12)* into the address stored in (R5):  *(let's say (R5) == x3100, as
above)*
        Mem[ (R5) ] ← (R1)   i.e.   **x3100 ← #12**

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 (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, 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.
Take the program from exercise 01 *(use your lab03_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 change 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) <u>only works with</u> <u>labels that are within +/- #256 memory locations of the instruction</u> *(see <u>here</u> 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 <u>indirect *(LDI, STI)*</u> and <u>relative *(LDR, STR)*</u> 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 *<u>label</u>* (aliased memory location) as a pointer.
The Relative memory addressing mode uses a *<u>register</u>* containing a memory location as a pointer.
*(The downside of both is that they require one more memory read than direct addressing).*

**Exercise 04: Loops**

*Review the workings of the NZP condition codes and the BR instruction [here](here).*

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.

**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 <u>memory addressing modes</u> – 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