<u>**CS061 - Lab 02**</u>

1 High Level Description

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

2 Goals for This Week

1. New LC3 instructions
2. Exercises 1 – 5
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 LC3 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 these steps.

From now on, we will use a standard notation to describe all these processes called **Register Transfer Notation**:

- (Rn) means "the *contents* of Register n" - i.e the value currently stored there.
- 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 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

The ST instruction - see the ST tutorial (Piazza: LC3 Resources -> LC3 Instructions)
The Store instruction 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 instruction - see the LDI tutorial
The Load Indirect instruction is similar to the LD instruction, except that there is one level of *indirection*

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`**

The 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 stored 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?)*
The STI instruction - see the STI tutorial
The Store Indirect instruction works exactly the same as LDI, only in the opposite direction. Rather than loading a value from Mem[Mem[someLabel]] into a register, 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. whatever is currently stored in R1)*

The LDR instruction - see the LDR tutorial
The Load Relative instruction achieves the same indirection as 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'll leave the offset equal to 0
```
loads the value from Mem[(R5)] *(let's say (R5) == x3100)* into R1:

`R1 ← Mem[ (R5) ]  i.e.  R1 ← Mem[ x3007 ]  i.e.` **`R1 ← #12`**

The STR instruction - see the STR tutorial
The Store Relative instruction works exactly the same as LDR, only in the opposite direction.
In the example, the instruction stores the value from R1 into the address stored in R5 (x3007):

`(R5) ← (R1)   i.e.` **`x3100 ← (R1)`** *`(i.e. whatever is currently stored in R1)`*

The Trap instruction - see Table 1 and the TRAP tutorial
Trap instructions are actually calls to a set of 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 be using 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 – 5

You are required to complete at least Exercises 1- 4 ***in lab***.

Exercise 5 is still required, but if you can't finish it in lab you may complete it any time prior to next week's lab *(note, however, that such exercises are usually intended as an aid to your programming assignments, so you would be well-advised to complete them as soon as you can).*

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 would 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 worked.

Exercise 02: Indirect memory addressing mode

Sometimes, the data we need is located in some remote region of memory, which may or may not have label aliases.

Take the program from exercise 01 *(copy it into a new file called e.g. lab2_ex2.asm – don't just edit the original!!)* and replace the data stored at DEC_65 and HEX_41 with two far away ***addresses*** such as x4000 and x4001 respectively (you don't need to change the labels), and change the labels to DEC_65_PTR and HEX_41_PTR *(the PTR stands for "pointer")*

You can load the data into the new locations by placing the following code at the end of your current data block:

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

(Make sure you understand what's going on here!)

But now we have a problem!
In the LC3, the Direct memory addressing mode (LD, ST instructions) only works with labels that are within +/- #256 memory locations of the instruction *(we'll explain why later)* – 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: you won't need the "new" labels – in fact you can now remove them).

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 or "Register") memory addressing mode

Start with the same setup as in exercise 02: You have two labeled locations (or “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.

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 (memory location) as a pointer.
The Relative memory addressing mode uses a register as a pointer.
(The downside is that indirection requires 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)* 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?*

<u>Exercise 05: Console I/O</u>
(may be completed after lab, and does not need to be submitted with your lab5.tgz)
See Table 1 above & Appendix A, table A.2 in the text (p. 543) for this exercise.

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?

Now replace GETC and OUT with a single TRAP x23 (IN) call *(this is the only time you will use this trap)*
What are the differences and similarities?

Finally add code to output an entire <u>*string*</u> of your choosing. Start off by just repeating exercise 0 from last week, but this time you have to figure out exactly how it works.
You will store the characters of a string in a block of memory (i.e. an array) using the pseudo-op .STRINGZ *(check the array in simpl - what is the last character? How big is the array? What do you call a character array like this in C++?);* and then output it to console with two steps:

- load into R0 the starting address of the array: **LEA R0, string_label**
  (note that this is quite different from outputting a single character as in Ex. 04 - there you load the ascii code for the character into R0, not the address of the character!)
- invoke the string output "trap" routine: PUTS

Now, use the skills you learnt in exercise 4 to output the characters in your array one at a time - i.e. build your own version of PUTS.

The big difference between this and Ex. 04 is that this time, you don't know how many iterations to make (at least not without counting the characters by hand!). So how will you stop the loop?
Hint: what is the value of the last character in your array? Can you use that to branch out of the loop?

3.3 So what now??

Before next lab *(in fact before tackling your homework & programming assignment this week)*, make sure you ...

- are totally familiar with the instruction description notation at the start of Section 3.2
  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 addressing modes – direct, indirect, and relative
- master the 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
- know how to build a simple counter-controlled loop
- can use all the bios routines – i.e the Trap instructions that manage console i/o: GETC, OUT, and PUTS