CS061 - Lab 2

1   High Level Description

Today's lab will cover some new LC-3 instructions, introducing the marvels of _memory addressing modes_, and some more I/O (input/output),

2   Goals for This Week

1. New LC-3 instructions using memory addressing modes _direct_, _indirect_, and _relative_.
2. Exercises 1 – 4:
   Practice using the three memory addressing modes;
   Building a simple loop using the _conditional branch instruction_; and
   Simple input/output using BIOS routines.
3. So what now??

3.1   New LC-3 Instructions

As you know, in the LC-3, values can be stored in system memory (RAM), and copied from there to the eight general purpose registers named R0 through R7, or in several special purpose registers *(more about these later on).*
And as you are beginning to see, **_all_** microprocessor instructions involve getting values from one or other of those places, manipulating them in some fashion, 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 "copy the **_value_** on the right of the arrow to the **_location_** on the left", rather like the C++ assignment operator '='
  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 LC-3 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 (also available at Canvas -> Files -> General Resources -> LC3 Assembly Language Guides -> LC-3 instructions in detail).
This is the "inverse" of the LD instruction: it copies 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]** does this: **R1 <- x4A20**

  *Now, for reasons that will become clear later on, the label ADDR_1 **must be no further away from the LD instruction than +/- 256 locations!***

if you attempt to LD or ST with a label further away than that, you will get an <u>assembly</u> 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 limitation - and, in fact, we have two such modes.

The LDI (Load Indirect) instruction - see the LDI tutorial
This looks a lot 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 <u>stored</u> 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. <u>R1 ←</u>**
<u>**x00C4**</u>

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*
that 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
Like LDI, this uses indirection, 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 <u>always</u> set the third parameter (called an <u>offset</u>) to 0.*
This copies the value from Mem[(R6) + offset 0] into R1:
(let's say (R6) = x3100, and Mem[x3100] is again x4A20)
        R1 ← Mem[(R6)+x0]   i.e. **R1 ← Mem[x3100]   i.e. <u>R1 ← x4A20</u>**

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.
*Note that ALL BIOS routines work with **ASCII characters** only!*
(Check this ASCII table or this image - ALWAYS ignore the Decimal & Octal columns!!)
When you "get" a character from the keyboard into R0, you get the ascii code representing that character. When you want to output a character/symbol to the console, you first have to copy the ascii code for that character into R0. PUTS allows you to output an entire array of ascii characters.

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 in an array until a 0 is encountered. *Used for printing strings.*
TRAP x23	IN	***You will never use this instruction!*** Prompt the user to input a single character; get the input from the keyboard & store it in R0, and echo it to the console.
TRAP x25	HALT	Terminates the program

If you use PUTS instead of OUT to print out a character you will get a messy string
3.2   Exercises 1 – 4

Exercise 1: Direct memory addressing mode (LD and ST)

Write a program in lab2_ex1.asm 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. *Check the binary values stored in the two locations - what do you notice?*
Look up an ASCII table *(use the back of your text or this table)* to see what these values represent when interpreted as <u>characters</u> rather than <u>numbers</u>.

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

Exercise 2: Indirect memory addressing mode (LDI and STI)

Sometimes, the data we need is located in some remote region of memory.
Copy the program from exercise 1 into your lab2_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 to remote data)*.

You can load the data into the new locations by putting the following code at the end of your current data block, <u>after</u> the HALT instruction and the .END pseudo-op. Notice that in LC3Tools, each `.orig` needs its own `.end`, just like how a function in C++ needs opening and closing braces.

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

But now we have a problem!
As we have already seen, the LC-3 Direct memory addressing mode (LD, ST instructions) <u>only works with labels that are within +/- #256 memory locations of the instruction</u> *(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 <u>back</u> into addresses x4000 and x4001 using **STI**.
Again, inspect the registers to make sure it worked as expected!

Exercise 3: Relative memory addressing mode (LDR and STR)

Use your lab2_ex3.asm file
Start with the same setup as in exercise 2: 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 (i.e. the actual remote memory addresses)_ 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 2 – 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.
(Both require one more memory read than direct addressing - can you explain where that occurs?).

Exercise 4: Loops

Use the conditional branch instruction (**BR**) to construct a *counter-controlled loop (the same control structure you learned in lab 1 and assn 1)*

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 as an ascii character (Trap x21, or OUT), then immediately increment R0 by 1.
Use R1 as the *loop counter* by decrementing it by 1 each iteration, right before the BR nstruction controlling the loop, just as you did in lab 1.
If you do it right, the loop will be executed exactly x1A times.
Think 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

Your TA will maintain a Google Sheet for questions and demos.
Demo your lab exercises to your TA **before you leave the lab**.
If you are unable to complete all exercises in the lab, demo the rest during office hours *before* your next lab to get full credit. You can demo during the next lab for -3 points.
Office hours are posted on Canvas (under Syllabus -> Office Hours) and Discord (#office-hours).

4   So what do I know now?

You should now ...

- Be comfortable with the Register Transfer Notation (RTN) as a way of describing instructions.
  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 LC-3 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 the bios routines – i.e. the Trap instructions that manage console i/o: GETC, OUT, PUTS and HALT.