CS061 - Lab 1

Welcome to the LC-3!

1 High Level Description

Today's lab will get you set up, and cover the basics of how to use the LC3 Assembler/emulator, and have you write a tiny program in LC-3 Assembly language.

The specs for today's lab are rather lengthy, as there are a lot of new ideas & activities to cover.
But don't panic - each step is pretty simple, and most future lab specs will be shorter (but harder!)

2 Objectives for This Week

1. Setting up LC3Tools
2. Introduction to the basics of LC3 programming
3. Exercise 0: Hello World!
4. Exercise 1: Implement, run, and inspect a simple program yourself
5. So am I an assembly language programmer yet??

NOTICES:

1. ***We do encourage you to help one another and actively collaborate on your lab exercises***
   (but NOT on your programming assignments, not EVER!)
   If you have taken this class before, you are required to delete all lab exercises and programming assignment code from your previous attempt.

   See also the class collaboration policy.

2. ***You will need a full-powered laptop or desktop computer to work on***
   *See the **BCOE laptop policy** for minimum laptop requirements.*

3. ***Read "Intro to Assembly Language"*** *(review of material covered in your first lecture).*
   You MUST read this BEFORE lab.
   In general, for all future labs, you must read the lab specs before attending lab, and make a start coding at least the first exercise before lab: this is even more important now that labs are virtual!

3 Lab 1

3.0 Getting set up

Download the correct version base on your operating system here
https://github.com/chiragsakhuja/lc3tools/releases/tag/v2.0.2
Mac: LC3Tools-2.0.2.dmg
Windows: LC3Tools-2.0.2.exe

3.1 Basic skeleton for LC-3 programs

Similar to C++, every LC-3 program has a certain amount of overhead necessary for making everything work properly.

In **C++**, you might have something like the following:

```
//
// File: main.cpp
// Description: A simple hello world-esque program
// Author: Noobie Q. Programmer
// Date created: 1/1/2019
// Date last modified: 1/1/2019
//
#include <iostream>
using namespace std;

int main()
{
  cout << "Wow, this is the most fun I've had in years!" << endl;
}
```

This source code is passed to a *compiler*, which ultimately produces a machine language *executable*.

In LC3, a basic program will have the following look:

```
;
; Author: Noobie Q. Programmer
; UCR NetID/CS username: nprog001
; Lab Section: xxx
; TA: Sean Foley
; Lab 1
; Date created: 1/1/2020
;

.orig x3000                 ; **ALL** your programs will start at x3000
; Instructions (i.e. LC-3 code)
HALT                         ; end of program code
; Local Data                 ; pseudo-ops for hard-coding data go here

.end                        ; .end is like the "}" after main() in C++.
                            ; It means "no more code or data"
```

If you haven't figured it out by now, LC-3 comments are denoted by semi-colons. They can be on a line by themselves or on the same line as actual code (after the code).

This source code is passed to an ***assembler*** - a much simpler beast than a compiler, but like a compiler it produces a machine language executable.

Pseudo-ops *(the keywords that start with a* `'.'` *like* `.end` *- see next section for more details)* are like compiler directives, in that they tell the assembler how to set things up.

3.2 Basics of LC-3 programming

An LC-3 program consists of three basic elements: instructions, pseudo-ops, and labels.

Labels are simply "aliases" for memory addresses. These are really, really useful because they relieve the programmer of the responsibility of figuring out which line of code is at which memory address, allowing us to use symbolic names to point to data and instructions.

Pseudo-ops tell the assembler how to set things up before starting to translate the source code into machine language. They are a bit like compiler directives in C++, such as `#include`.

For example, the pseudo-op .FILL tells the assembler to write a hard-coded value into memory (i.e. RAM, or system memory). For example, the two .FILL pseudo-ops in Table 1 would write the decimal value 6 into memory at x3005 *(aka DEC6)*, and the decimal value 12 into memory at x3006 *(aka DEC12)* before the program begins execution.
Note that you just need to label the lines - you don't need to know the actual memory addresses corresponding to those labels (that's the assembler's job!).

Table 1: Labels and Pseudo-ops

Memory address	Label	Pseudo-Op	Hard-coded value
x3005	DEC6	.FILL	#6
x3006	DEC12	.FILL	#12

(The '#' before the numbers means decimal, i.e base 10; 'x' means hexadecimal, i.e. base 16).

Now we can refer to these memory locations (and therefore access the data stored there) using the names DEC6 and DEC12.
We will learn about another pseudo-op .STRINGZ in a moment (in section 3.3).

Instructions: There are 15 LC-3 instructions, in three distinct categories *(see this tutorial for more)*:

- Arithmetic / Logic operations (addition, bitwise AND, bitwise NOT)
- Data movement (moving data back & forth between memory and local registers)
- Program control (changing the order of execution of code - branches & loops)

The following sections illustrate one or two instructions from each of these categories.

3.2.1 Arithmetic / Logic Operations

These instructions are used for manipulating values and doing calculations - Adding two numbers, bitwise ANDing two words together, or taking the bitwise logical NOT of a word.
The LC-3 has 8 general purpose, 16-bit local registers (analogous to variables in C++), named R0 through R7, that can be used as temporary storage for performing such operations.

The ADD instruction (2's complement integer data type)
This instruction adds two numbers, and writes the result to a register. It comes in two flavors:
One in which both operands come from local registers; the other in which the first operand comes from a register, while the second is actually embedded in the instruction itself ("immediate" mode).
In the immediate mode, the embedded value can only be in the range {-16, +15}.
See the ADD tutorial *(also on the Canvas Resources page, "LC3 instruction set")* for more details

3.2.2 Data movement

Data movement instructions are used when you want to copy a value from a memory location into a local register ("loading"), or copy a value from a register to a memory location ("storing").
You will be doing this a ***lot***, since the LC3 can operate only on values in registers, not in memory.

The LD instruction ("Load Direct" - probably the most frequently used instruction!)
This instruction copies the contents of a specified memory location (identified by a label) into a register.
See the LD tutorial

The LEA instruction ("Load Effective Address")
This instruction translates a *label* – the name you give to a memory location – back into the 16-bit *memory address* it stands for, and stores that *address* into the specified register.
There is no actual data movement here – just the decoding of a label.
See the LEA tutorial

3.2.3 Program control

These instructions are used in control structures such as BRANCHING, LOOPING, and SUBROUTINE CALLS *(a simpler version of function calls in higher-level languages)*.
By default, a program starts at the first instruction and executes one instruction after another in order until it reaches the end ("sequential execution").
Control instructions allow us to alter the default sequencing in several ways.

The BR instruction (“Conditional Branch”) can be used to transfer control to the instruction at a specified label based on a *condition*.
All the control structures you are familiar with from C++ (if; if-else; if-else if-else; for; while; do-while; switch) can be built using this instruction, sometimes in combination with the JMP instruction.

But before we can learn how it works, we have to know about the Condition Codes:
Three "flags" (single-bit registers) are set whenever a value is written into any one of the local registers, indicating whether the value being written is Negative, Zero, or Positive. These flags are called the N Z P Condition Codes, and allow us to make decisions based on the last value written to a register - referred to as the last modified register (“**LMR”**).

The decisions are made by the conditional branch instruction BR, which has three possible modifiers: {n, z, p}.
As you can guess, the n modifier asks whether the N Condition Code is currently set or not *(was the value just written negative?)*; and likewise for the z and p modifiers.
The BR instruction can be modified by any combination of n, z, p. It will cause a branch to the labeled instruction ***IF AND ONLY IF*** any one of the specified conditions is met.
See the BR tutorial

And now, we can finally get started on the actual lab exercises :)

3.3 **Exercise 0: lab1_ex0** *(aaargh!! not "hello world" again?!?!)*

As always, the very first program we will write will simply output the message "Hello World!"
Though simple, it will illustrate several of the points above, plus some things you will fully understand only later on *(exactly like when you first encountered* `cout` *in C++)*:

- We will store the string as an array of ASCII characters in memory, starting at an address labeled "message", using the pseudo-op .STRINGZ
  This pseudo-op stores a string in memory, one character per memory address, with a zero ("the null character" or '\0') after the last character.
  You will recognize this as a "c-string", i.e. a null-terminated character array. So the line:
  `shrug .STRINGZ "whatever"`
  will write the characters 'w' 'h' 'a' 't' 'e' 'v' 'e' 'r' followed by x0000, to the nine memory locations starting with the labeled address `shrug`.
- We will use the LEA instruction to translate the label "shrug" into its actual address, which we will store in R0.
- We will invoke an i/o subroutine called PUTS *(can also be written as* TRAP x22*)*, which uses R0 to locate the string *(it knows when to stop because the last value in the char array is '\0')*.

```
;
; Hello world example program
; Also illustrates how to use PUTS (aka: Trap x22)
;
.ORIG x3000
;-------------------
; Instructions
;-------------------
      LEA R0,  MSG_TO_PRINT    ; R0 <-- the location of the label: MSG_TO_PRINT
      PUTS                     ; Prints string defined at MSG_TO_PRINT

      HALT                     ; terminate program
;-------------------
; Local data
;-------------------
      MSG_TO_PRINT    .STRINGZ     "Hello world!!!\n"     ; store 'H' in an address labelled
                                                          ; MSG_TO_PRINT and then each
                                                          ; character ('e', 'l', 'l', 'o', ' ', 'w', ...) in
                                                          ; it's own (consecutive) memory
                                                          ; address, followed by #0 at the end
                                                          ; of the string to mark the end of the
                                                          ; string
.END
```

Now write this program for yourself:

1. Open up LC3Tools (the application you just installed)
2. Copy in the code from the above image & save the file *(you can leave the editor open if you wish)*.
3. Assemble your code by hitting the wrench icon on the left
4. To load your code, on the top right you should see a CPU icon, click on that and it will take you to a new window (the Simulator)
5. Click on the power button on the left and make sure that your memory is all zeroed out

6. Click the folder button and open up the file you just compiled
7. In the “Jump to Location” field type in x3000
8. Hit Run (▸) on the left side
9. If you want to rerun your code, hit the Reload button (↻)
10. If you need to trace your code (ie debug), use the Step Over button (↷) on the left

3.4 Exercise 1: lab1_ex1 - A “real” program

Now Hello World is out of the way, we can write a program that actually does some processing: it will multiply a number, which we will store in advance in memory, by six *(How hard can that be, right??!?)*

We will also take this opportunity to introduce the style you ***MUST*** use for all your LC-3 programs.

See the image below:

First comes the header, which will always be included in your starter code: obviously, you have to always fill it out with your personal details.

Next, we have the line `.orig x3000` *(you may use upper or lower case for all LC-3 instructions, as you wish)*. This indicates where in memory the code will reside. The LC3 assembler requires all programs to be loaded at or above x3000: all your programs ***must*** load the “main” routine to x3000
(just try something else and see what happens!! On second thoughts – don't!)

The code itself multiplies 6 by 12.
Since the LC-3 has no multiply instruction *(really!!)*, the operation has to be carried out by adding 12 into an accumulator 6 times.
We could have added 6 into an accumulator 12 times: why would that not be a good idea?

```
;-----------------------------------------------
; Foley, Sean
; Login: sfoley (sfoley@cs.ucr.edu)
; Section: xxx
; TA: Sean Foley
; Lab 01
;-----------------------------------------------
.orig x3000
    ;--------------------
    ; Instructions
    ;--------------------
    LD R1, DEC_0                ; R1 <-- #0
    LD R2, DEC_12               ; R2 <-- #12
    LD R3, DEC_6                ; R3, <-- #6

    DO_WHILE_LOOP
        ADD R1, R1, R2          ; R1 <-- R1 + R2
        ADD R3, R3, #-1         ; R3 <-- R3 - #1
        BRp DO_WHILE_LOOP       ; if (R3 > 0): goto DO_WHILE_LOOP
    END_DO_WHILE_LOOP

    HALT                        ; halt program (like exit() in C++)
    ;-------------------
    ; Local data
    ;-------------------
    DEC_0       .FILL    #0     ; put #0 into memory here
    DEC_12      .FILL    #12    ; put #12 into memory here
    DEC_6       .FILL    #6     ; put #6 into memory here

.end
```

Can you see how this program works?

The next three lines load hard-coded values from memory into registers R1, R2, and R3 respectively:
R1 is an "accumulator" - i.e. at the end, it will hold the result - so we have to initialize it to 0;
R2 holds the multiplicand *(the number to be multiplied)*; and
R3 holds the multiplier - we will use it as a loop counter, to keep track of how many times the loop will execute *(i.e. we will create a "counter-controlled" loop)*.
Make sure you understand how the BRp instruction controls the loop - this is the foundation of all Assembly Language control structures!!

TIP:

Actually, copying values from memory to register (and vice versa) is rather inefficient.
Mostly, we can't avoid it - our fixed data has to be stored in memory.
However, there is one special value for which we can do better - 0!
There is a much cleaner way to "zero out" a register than copying a hard coded 0 from memory to register:

```
AND R1, R1, x0    ; R1 <- (R1) AND x0000
```

This does what we call a "bitwise AND" of R1 with 0, i.e. it ANDs each of the 16 bits in R1 with the corresponding bit in x0000, and stores the result (x0000) back in R1 - a LOT quicker than

reading from memory!

Note how the program data is stored in a block separated from the code, ***following*** the HALT instruction, which ends code execution - this way, the processor does not attempt to "execute" the data as though it were code!
And note how we use comments to clearly distinguish the two blocks.

Now that you have had a nice lengthy overview of some basic LC-3 instructions and have seen how to use the LC3Tools assembler/emulator, it is time for you to implement the program yourself.

Open file **lab1_ex1.asm** for editing, type in the code from the image, save it, and open it in LC3Tools.
Remember to replace the first LD R1 instruction with the AND instruction, as per the tip above.

This time, instead of hitting Run, step through each line of code using the **Step Over (↷)** button in the Simulator until you reach the HALT instruction. Notice how the value of R1 keeps going up by 12 until it reaches *(guess what??)* 72.
(Now, how did it know to stop at just the right number??)

Finally, congratulate yourself on a job well done :)

3.5 Submission

Your TA will tell you how to signal that you want help, and how to demo your lab exercises.
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 *by the end of the week* to get full credit. Penalties apply if you demo later than the current week.
Office hours are posted on Canvas (under Syllabus -> Office Hours).

4 So what do I know now?

You have now mastered the following skills:

- How to write basic LC-3 assembly language code
- What labels are and how to use them
- How to use the .FILL and .STRINGZ pseudo-ops to store program data in memory
- The LD, LEA, BR, HALT (aka Trap x25) instructions
- The PUTS output routine (aka: Trap x22)
- How to create, save, run, and step through an LC-3 program
- And you now know that the LC-3 can't even multiply two integers by itself!

Not bad for the first day :)