CS061 - Lab 02
Welcome to the LC3

1 High Level Description

Today's lab will cover the basics of how to use the LC3 Assembler/emulator, and write a basic program in LC3 Assembly language.

The text for today's lab is rather long.
Don't panic! Most labs will not be this verbose! There is much introductory material to cover.

2 Objectives for This Week

1. Learn the basic skeleton for any LC3 program file
2. Introduction to the basics of LC3 programming
3. Exercise 0: Hello World!
4. Exercise 01: Implement, run, and inspect a simple program yourself
5. So am I an assembly language programmer yet??

3 Assembly Language Intro

3.0 Read "Intro to Assembly Language" (review of lecture material).

3.1 Basic skeleton for any LC3 program

Similar to C++, every LC3 program has a certain amount of overhead necessary for making everything work properly. In C++, you may have something like the following:

```
//
// File: main.cpp
// Description: A simple hello world-esque program
// Author: Sean Foley
// Date created: 12/19/2009
// Date last modified: 12/19/2009
//
#include <iostream>
using namespace std;

int main()
{
  cout << "Egads, 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:

```
;
; Foley, Sean
; Login: sfoley (sfoley@cs.ucr.edu)
; Section: xxx
; TA: Sean Foley
; Lab 01
;

.orig x3000                ; **ALL** your programs will start at x3000
; Instructions
; LC3 instruction code goes here
; Local Data
                           ; pseudo-ops for hard-coding data go here
.end                       ; .end is like the "}" after main() in C++.
                           ; It means "no more code to compile!"
```

If you haven't figured it out by now, in LC3 comments are denoted with semi-colons. They can be on a line by themselves or on the same line as actual code (at the end of the line).

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

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

3.2 Introduction to the basics of LC3 programming

An LC3 program consists of three basic elements: pseudo-ops, instructions, 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, thus allowing us to use symbolic names to access memory locations containing 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++, like #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 value 6 into memory at x3005 and the value 12 into memory at x3006 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 psudo-op .STRINGZ in a moment (in section 3.3).

Instructions: There are 15 LC3 instructions, in three distinct categories:

- Arithmetic / Logic operations (addition, boolean operations, etc)
- Data movement (moving data between RAM and local registers)
- Program control (changing the order of execution of code - if statements, for loops, etc)

Following are 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 AND'ing two numbers together, or taking the bitwise logical NOT of a number.
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).
See the ADD tutorial *(also on the Piazza 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 (called "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 into a register.
The memory location in question is specified by its label.
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 *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. We will see later how it is used to support subsequent data movement instructions.
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)*.

The BR instruction ("Conditional Branch")
Ordinarily, a program starts at the first instruction and executes one line of code after another until it reaches the end ("sequential execution").
The Branch instruction can be used to *alter* this flow of execution based on a *condition*.

All the control structures you are familiar with from C++ (if, if-else, for, while, do-while, etc) can be built using this instruction, often 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, and indicate 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"**).

Those decisions are actually 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 written negative?)*; and likewise for the z and p modifiers.
BR causes a branch to the labelled instruction ***IF AND ONLY IF*** any one of the specified conditions is met.
See the BR tutorial

3.3 **Exercise 0: lab02_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 labelled "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.
  shrug .STRINGZ "whatever"
  will write the characters 'w' 'h' 'a' 't' 'e' 'v' 'e' 'r' followed by x0000, to the nine memory locations starting at the labelled address shrug.
- We will use the LEA instruction to translate the label into its actual address, which we will store in R0.
- We will invoke an i/o subroutine called PUTS, which uses R0 to locate the start of the string to be output *(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. First (as with all your labs and assignments) open Piazza to the GitHub Classroom Lab and Assignment Links post, and click on the lab 1 link to create your lab-1 GitHub repo, then clone it to your cs account, where it will be called lab-1-<your github username>/
2. In your terminal, cd into the local repo you just cloned, and open your ex_0 file in gedit (or geany, or equivalent):
   `gedit lab02_ex0.asm &`

3. Copy in the code from the above image & save the file
4. From your terminal, launch the LC3 assembler & emulator by typing
   `simpl lab02_ex0.asm &`
   This will open two windows:
   a. the emulator itself, showing the state of the processor, including the assembled code (aka "machine language" or "binary") in memory; and
   b. the console, which is where all output from the program is displayed (and where you will type input to the program in later programs).
5. Now you will open a third, the Text Window, the text interface to the emulator.
   This can be used to enter command line input to the emulator, but we will use it mostly to watch for warnings and error messages.
   Check the Text Window box at the bottom center of the emulator window.
6. Hit the **Run** button at the bottom of the emulator window (or enter the word "run" in the Text Window input box), and observe the words “Hello World!” magically appear in the console window!

3.4 Exercise 01: lab02_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.

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`
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: but since the LC3 has no multiply instruction, the operation has to be carried out by adding 12 into an accumulator 6 times.
We could have done this by adding 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 number to be multiplied;
and R3 is used as a loop counter (to keep track of how many times the loop will execute – i.e. 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 - most of 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 to copy 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 bit 0, and stores the result (0) back in R1. Done!!
> And it is a LOT quicker than reading from memory!

Note how the data is stored in a block separated from the code, ***after*** the HALT instruction, which ends code execution.

And finally, note how we use comments to "partition" the file into instructions and data.
Now that you have had a nice lengthy overview of some basic LC3 instructions and seen how to use the simpl assembler/emulator, it is time for you to implement the program yourself.

Open file **lab02_ex1.asm** for editing, type in the code from the image, and open it in simpl.

This time, instead of hitting Run, step through each line of code using the **Step** button in the emulator 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 youself on a job well done :)

3.5 Submission

Add, commit, and push your lab02_ex0.asm and lab02_ex1.asm files to your lab 2 GitHub repo.

4 So what do I know now?

You have now mastered the following skills:

- Using GitHub repos to store and manage your work, and cloning them to local git repos
- That all LC-3 programs start at x3000 or above
- 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 LC3 program
- And you now know that the LC-3 can't even multiply two integers!

Not bad for 2 weeks :)