Lab 13

Lab 13 assignment is due on Friday, May 8th, by 9pm in your svn repository. For help, please check the Labs FAQ first. Specifically, please read the Terminal Troubleshooting and SVN Troubleshooting.

Please ask all questions about this assignment during the office hours or post questions in the WeChat group.

This lab is to be done on **a Linux workstation or** in **VM**. If you run your own Linux distribution, please see Installing LC-3 tools on your machine section for instructions how to install the tools needed in this lab.

Introduction to LC-3 assembly language

So far you have been writing programs in LC-3 machine language. As you may have noticed, machine language can be very difficult to read without sufficient comments; in fact, it is very likely that you have looked at your program, thought "What is this part supposed to do again?", and then had to trace through your program by hand to relearn what the code is doing. Assembly is the first step (outside of comments) in making machine language easier to read.

Similarities Between Machine Language and Assembly

Here is an example of a few lines of a program written in LC-3 machine and assembly languages. The first code block shows the program written in machine language, and the second block shows the same program written in assembly.

Machine Language:

```
0011 0000 0000 0000  ; specifies where the program will be stored (x3000)
0101 010 010 1 00000 ; clear R2 by ANDing it with x00000
0001 010 010 1 01100 ; add the decimal number 12 to R2, and store result in R2
1111 0000 0010 0101  ; halt
```

Assembly:

```
.ORIG  x3000      ; specifies where the program will be stored (x3000)

; Remember from the previous lab that the first line in a machine language program is not an instruction but
tells the LC-3 simulator where you want to load the program in the LC-3 memory
; You have to do the same with assembly programs too. The first line has to be a .ORIG
AND    R2,R2,#0   ; clear R2 by ANDing it with x00000
ADD    R2,R2,#12  ; add the decimal number 12 to R2, and store result in R2
TRAP   x25        ; halt
.END
```

As you can see, assembly is much like machine language. In fact, assembly can almost be directly translated into machine language. For example, look at the second line of code:

- `0101` is the opcode for `AND`
- `010` specifies R2 as the destination register
- `010` specifies R2 as the source register
- `1 00000` specifies the immediate value of 0 (#0 in assembly)

For immediate (a.k.a. literal) values, # indicates a decimal number and x indicates a hex number. For example, decimal 10 can be represented as #10 or xA

Each instruction in machine language consists of 16 1's and 0's, while each instruction in assembly consists of: LABEL OPCODE OPERANDS COMMENTS

LABEL and COMMENTS are optional, while OPCODE and OPERANDS are not. The above code contains OPCODE (AND), OPERANDS (R2,R2,#0), and COMMENTS, but not LABEL. We'll talk about labels more later.

Just because assembly is easier to read does not mean you can neglect comments. You no longer need to specify what a particular instruction is doing (such as R1 <- 5) but you still need to explain the purpose of specific segments of code.

Machine language and assembly have other similarities as well. For example:

1. Machine language files have the .bin extension, while assembly files have the .asm extension
2. .bin files are converted into .obj files with the LC-3 Converter ("lc3convert"), while .asm files are assembled into .obj files with the LC-3 Assembler ("lc3as")
3. Since a code written in either of the languages is converted into .obj files, we can use the "lc3sim" or "lc3sim-tk" programs

Writing Assembly Programs: Labels and Pseudo Ops

Other than being easily read, assembly language has a couple of powerful features that make life easy for the programmer- Labels and Pseudo Ops.

Labels

Consider the following code. Remember that PC+offset uses the *incremented* program counter.

; This (rather pointless) program loads R1 with data at the memory location x3005 ; R1 is then decremented by 1 until R1 becomes negative, at which ; point the value in R1 is stored at memory location x3005 and the program halts.

Opcode/ Pseudo Op	Operands	Comments
.ORIG	x3000	; program starts at memory location x3000
LD	R1, #4	; load R1 with data at memory location x3005 (PC+4)
ADD	R1, R1, #-1	; decrement R1 by 1
BRzp	#-2	; branch to x3001 (previous instruction, PC-2) if R1 is zero or positive
ST	R1, #1	; store R1 at memory location x3005 (PC+1)
HALT		
.FILL	x0005	
.END		

And now consider the same code with labels. Labels are symbolic names that identify memory locations that are referred to explicitly in the program. The assembler will use the labels to compute the correct offsets automatically.

; This (rather pointless) program loads R1 with data at the memory location specified by ; the label NUMBER. R1 is then decremented by 1 until R1 becomes negative, at which ; point the value in R1 is stored at NUMBER and the program halts.

Label	Opcode/ Pseudo Op	Operands	Comments
	.ORIG	x3000	;program starts at location x3000
	LD	R1, NUMBER	;load R1 with data at memory location specified by NUMBER
LOOP	ADD	R1, R1, #-1	;decrement R1 by 1
	BRzp	LOOP	;branch to LOOP if R1 is zero or positive
	ST	R1, NUMBER	;store R1 at memory location specified by NUMBER
	HALT		;halt program
NUMBER	.FILL	x0005	;memory location specified by label NUMBER
			:(in this case data=x0005)
	.END		

The above program uses two labels: LOOP and NUMBER. LOOP specifies a memory location that contains the target of a branch instruction (in this case the ADD R1,R1,#-1 instruction), while NUMBER specifies a memory location that is loaded or stored. These are the two reasons for directly referring to a memory location. Labels make referring to specific memory locations much easier and clearer; for example, it is easier to understand the instruction ST R2, RESULT than the instruction ST R2,#29. Labels also make branching easier because you can simply branch to a line of code with a label (such as LOOP1) in front of it, as opposed to having to find out the offset to the line's memory location. This is especially helpful when making changes to a program, since adding or removing lines may cause offsets to change. But when we use labels, the assembler computes the correct offsets automatically, saving us a lot of trouble.

Labels are 1 to 20 alphanumeric characters long (uppercase, lowercase, and decimal digits). Labels must start with a letter of the alphabet. Some examples are loop1, NUMBER, Result, r5d4

Labels are used to explicitly refer to a memory location. There is no need to give a location a label if it is not explicitly referenced in the program.

Type the two implementations into files (say implementation1.asm, implementation2.asm) and assemble then with the lc3as tool (lc3as implementation1.asm, lc3as implementation2.asm) and run them to see for yourself that both the implementation perform the same function.

Pseudo Ops

Pseudo-ops (also known as assembler directives) help the LC-3 assembler translate your code into a file that the LC-3 recognizes as a program (we'll talk about the LC-3 assembler itself in a bit). The above program has two pseudo-ops: .ORIG and .END. The other pseudo-ops are .FILL, .BLKW, and .STRINGZ. Below is a table detailing the five pseudo-ops used by the LC-3 assembler.

Pseudo-op	Format	Description
.ORIG	.ORIG #	Tells the LC-3 simulator where it should place the segment of code starting at address #.
.FILL	.FILL #	Place value # at that code line.
.BLKW	.BLKW #	Reserve # memory locations for data at that line of code.
.STRINGZ	.STRINGZ "<String>"	Place a null terminating string <String> starting at that location.
.END	.END	Tells the LC-3 assembler to stop assembling your code.

It is important to note that pseudo-ops *do not* refer to operations that will be performed during program execution (hence the word "pseudo"). Instead, pseudo-ops are simply messages to the LC-3 assembler to help the assembler during the translation process.

While the other pseudo-ops are optional, .ORIG and .END must **always** appear at the beginning and end of your assembly program, respectively. In fact, if either is not present (or is formatted incorrectly), the LC-3 assembler will produce an error message. Also **note that .END does not stop the program during execution, but simply marks the end of the source program (to halt the program use "TRAP x25")**.

The LC-3 Assembler

To better understand how labels work, we have give you here a few details about the LC-3 Assembler

The LC-3 assembler is much like the LC-3 converter in that it translates your code into a file that the LC-3 recognizes as a program. The LC-3 converter "converts" a .bin file into a .obj (object) file while the LC-3 assembler "assembles" a .asm file into a .sym (symbol) file as well as a .obj file. We'll talk about the symbol file in a little bit. Once you have created an assembly program (such as filename.asm) you may run the LC-3 assembler. The command to run the assembler is "lc3as" followed by your file name.

```
lc3as filename.asm
```

If the assembly process was successful (i.e. no errors) the assembler will create two files, filename.obj and filename.sym. You can then run the LC-3 simulator the same way as you would for machine code.

```
lc3sim filename.obj
```

OR

```
lc3sim-tk filename.obj
```

If your program does have errors, you will have to edit your .asm file and then re-assemble the file using lc3as. You may have noticed a pair of messages (hopefully both telling you there were 0 errors) when you ran lc3as. Unlike the converter, the assembler translates a program via a two-pass process. Why do you think the assembler requires two passes?

If the assembler tried to translate a program with only a single pass, the process would fail as soon as it encounters a label. For example, look at the line LD R1,NUMBER. This line says to load R1 with the data stored at the memory location specified by NUMBER. However, the assembler has no idea what NUMBER is at this point and thus the assembly process fails.

For this reason, the assembler uses two passes through the program. During the first pass a symbol table (the .sym file) is created; this table identifies the binary addresses that correspond to labels (for example, the address that corresponds to the label NUMBER. Now that the assembler knows what address each label refers to, the second pass may commence. The second pass translates the assembly language into machine language (the .obj file).

Lab assignment

Remember to "cd" into your working copy of the repository and then update it with the command "svn update" Since we added a lab13 directory to your repository, you should now have it in your working copy. Inside the lab13 directory you should find two files, **lab13.asm** and **lab13.txt**

lab13.asm contains a program that computes 5! (five factorial). However, there are several errors that prevent this program from being assembled. Run the LC-3 Assembler and fix the errors. (Hint: Make sure you understand what the code is doing before you start fixing offset errors). Describe the errors that occur when trying to assemble lab13.asm. Identify the cause of each error. Answer all questions in lab13.txt.

After the code assembles, search for other errors in the execution of the task and fix them. Remember that you cannot assume that registers are cleared before you start your code. Comments at the beginning and throughout the program explain what it is doing and how each register is used. The program is working correctly if the resulting answer is decimal 120 (x78) and is stored in memory location labeled RESULT.

Do not change label names. You can add more labels, but please do not change the names of the existing ones.

When answering Q1, provide line number at which a particular error occurred.

In case you fixed your code before you answer the questions, here is the original code with bugs.

```
; This program calculates X! (X factorial). It can calculate
;   different numbers (4!, 6!, etc.) by changing the value of the first memory
;   location at the bottom of the code. It is currently set up to calculate
;   5!. The program does not account for zero or negative numbers as input.

; This program primarily uses registers in the following manner:
; R0 contains 0 (registers contain zero after reset)
; R1 contains multiplication result (6x5 = 30, 30x4 = 120, etc)
; R2 contains -1
; R3 contains counter for outer loop
; R4 contains counter for inner loop
; R5 contains current sum

x3000

    LD    R1,INPUT            ; R1 contains input number
    LD    R2,x3010            ; R2 contains -1
    ADD   R3,R1,R2            ; R3 contains input number -1
    ADD   R3,,R3,R2           ; R3 contains input number -2
    ;   (initializes outer count)
OUTERLOOP   ADD   R4,R0,R3            ; Copy outer count into inner count

    ; This loop multiplies via addition (6x5 = 6+6+6+6+6 = 30,
    ;   30x4 = 30+30+30+30 = 120, etc)
INNERLOOP   ADD   R5,R5,R1            ; Increment sum
    ADD   R4,R4,R2            ; Decrement inner count
    BRzp  INNERLOOP           ; Branch to inner loop if inner count
    ;   is positive or zero
    ADD   R1,R0,R5            ; R1 now contains sum result from inner loop
    LD    R5,x300F            ; Clear R5 (previous sum) to 0
    ADD   R3,R3,R2            ; Decrement outer count
    BRpz  OUTERLOOP           ; Branch to outer loop if outer count
    ;   is positive or zero

ST   R1,x3011             ; This address contains X!
    TRAP x25

INPUT      .FILL  x0005        ; Input for X!, in this case X = 5
    .FILL  x0000
    .FILL  xFFFF               ; 2's complement of 1 (i.e. -1)
RESULT     .FILL  x0000        ; At program completion, the result is stored here
```