Lab 6 Preparation Guide

Lab 6 again requires that you make use of the LC-3 tools installed on the lab machines, this time using the LC-3 assembler instead of the conversion tool. This document will give you some guidance as to how to make use of those tools. Nothing in this document is graded, nor need you turn anything in. You are welcome to learn the tools in any way that you find suitable. However, if you do not learn the tools, you may find it difficult to complete Lab 6.

So far, you have been writing programs in LC-3 machine language. As you may have noticed, machine language can be difficult to read without sufficient comments; in fact, it is 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.

In this guide, you will learn the differences between LC-3 machine language and LC-3 assembly language, how the features of assembly language simplify the work done by the programmer, and how to assemble an LC-3 assembly language program into an object file. Then you will be offered an optional exercise to let you practice.

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  ; starting address of program (x3000)
0101 010 010 1 00000 ; clear R2 by ANDing it with x0000
0001 010 010 1 01100 ; add decimal 12 to R2, and store result in R2
1111 0000 0010 0101  ; halt
```

```
; assembly
; Remember that the first line in a machine language program is not
; an instruction, but tells the LC-3 simulator where the program
; starts in memory.  In assembly language, .ORIG serves the same
; purpose.
.ORIG  x3000      ; starting address of program (x3000)
AND    R2,R2,x0   ; clear R2 by ANDing it with x0000
ADD    R2,R2,#12  ; add decimal 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 (x0 in assembly)

For immediate values, '#' indicates a decimal number and 'x' indicates a hexadecimal number. For example, decimal 10 can be represented as #10 or xA.

Each instruction in machine language consists of 16 1s and 0s, while each instruction in assembly consists of:

LABEL OPCODE OPERANDS COMMENTS

LABEL and COMMENTS are optional, while OPCODE and OPERANDS are not (although a line may be blank or contain only a comment). The AND instruction in the previous 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 that 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 an **`.obj`** file, we can use the "**`lc3sim`**" or "**`lc3sim-tk`**" programs to simulate execution of the program in both cases.

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.

Opcode/ Pseudo Op	Operands	Comments
		; 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
.ORIG	x3000	; and the program halts.
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 (PC - 2) if R1 is zero or positive
ST	R1, #1	; store R1 at memory location x3005 (PC+1)
HALT		
.FILL	x0005	
.END		

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 uses the labels to compute the correct offsets automatically.

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

The program above uses two labels: LOOP and NUM. LOOP specifies a memory location that contains the target of a branch instruction (in this case the ADD R1,R1,#-1 instruction), while NUM 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 start with an alphabetic character (A to Z, in either case).
- Labels can include letters, digits, and the underscore character ('_').
- Labels are case insensitive (so "NuM" is the same as "NUM" and "num").
- Examples include loop1, NUMBER, Result, r5d4
- Labels are used to explicitly refer to a memory location.
- There is no need to label a location unless that location is referenced in the program.

Pseudo-ops and Directives
Pseudo-ops and directives help the LC-3 assembler translate the assembly language into machine code (we'll talk about the LC-3 assembler itself later). The program shown in the last section had two directives, .ORIG and .END, and one pseudo-op, HALT. Pseudo-ops produce bits for memory locations, while directives simply provide information to the LC-3 assembler.

Additional pseudo-ops include .FILL, .BLKW, and .STRINGZ as well as IN and OUT (these are shorthand for TRAP instructions, as is HALT). The table below describes some of the pseudo-ops and directives supported by the LC-3 assembler.

Pseudo-op/ Directive	Format	Description
.ORIG	.ORIG <N>	Tells the LC-3 simulator that the program should start at address N.
.FILL	.FILL <N>	Place value N in the next memory location.
.BLKW	.BLKW <N>	Reserve N memory locations for data at that line of code.
.STRINGZ	.STRINGZ "<String>"	Place a NUL-terminated string <String> in the next memory locations.
.END	.END	Tells the LC-3 assembler to stop assembling your code. The assembler WILL NOT READ the rest of your file!

While most 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 "HALT" (TRAP x25)**.**

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 an **`.obj`** (object) file while the LC-3 assembler "assembles" an .**`asm`** file into a **`.sym`** (symbol) file as well as an **`.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 is successful (no errors are generated) the assembler creates two files, **`filename.obj`** and **`filename.sym`**. You can then run the LC-3 simulator the same way as you would for machine code.

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).

An Exercise (Optional)
Update your Subversion repository to obtain the **`lab6prep`** subdirectory, which contains the file **`prep6.asm`**. The program is meant to compute 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**). 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.