Simple Assembler

What is an assembler? An assembler turns an assembly language program written by humans in a very specific format into machine language that a microprocessor can understand. It is similar to the way C is compiled into machine language. The last step of compiling a C program is actually to run an assembler, but g++ handles that for you.

Assembly language instructions translate directly into machine code. There are some calculations that must be performed and it is usually best to let the computer handle those instead of trying to manually calculate each one individually. You can do the manual translation without and assembler, but it is time consuming.

The finished machine language for the LC3 processor is a series of four digits, 16 bit hex numbers without the x (since we know they are hex). The initial value will be the origin value or the first memory location of the first instruction when the program is loaded into memory. The rest of the values are translations of the instructions. We will be printing these and not saving them to a file.

This assignment will be broken into two parts. Assemblers typically have two passes. Pass one will read all of the labels and a second pass will actually do the translating.

Part 1 of this assignment will be to create the first pass. Find all labels and note the memory location associated with each.

Part 2 will be to read through the file again and translate each instruction into its machine language form and print the machine language form to the to the screen. Since the numbers are ASSUMED to be hex numbers, you DO NOT put an x before them when printing.

You MUST complete part 1 to be able to do part 2.

SIZE LIMIT – There is a size limit on part 2 of 32K. There is no reason ever for you to exceed this limit. You are doing things very, very inefficiently if you even come close to this limit. Come and see me if you have exceeded 32K so we can talk about how you should program.

The Assignment - Part 2- The Second Pass

In this part of the assignment you are going to print the machine language program to the screen. You will print the original ***lc*** and then read each instruction, translate it into hex and print that hex value until the .end directive is reached.

The instructions you will be implementing will be ADD, AND, NOT, LD, LDR, ST, STR, BR, and TRAP. You will create functions to translate and print the hex value for each of these.

1. For this entire assignment, case is unimportant. That is your assembler should recognize ADD or Add.
2. For this entire assignment, if you print anything, it must always be followed by a newline.
3. Remove the call to **printLabels** from the **firstPass** function from part 1 of this assignment.

4. Create a new function named **secondPass** which takes the file pointer, the labels int array, and the location counter integer as arguments. Uncomment the call to **secondPass** in your **assemble** function.
5. The **secondPass** function should read each line from the input .asm file. You will need to restart scanning of the file you have opened to the beginning. You can do this with the ***rewind*** function in the stdio.h library. Look it up. Blank lines should be ignored. Lines which begin with a semicolon are considered comments and should be ignored.

6. **The first thing secondPass should do is to print the lc value as a four digit hex number without a leading x or X or 0x or 0X followed by a newline.**

7. Your **secondPass** function should then read the lines one at a time, like the ***firstPass*** function.
   NOTE: Case is unimportant. Your program should recognize *ADD* or *add* as a correct instruction.
   If the line is a blank line (possibly contains spaces and tabs), ignore it.
   If the line begins with a semicolon (possibly contains spaces and tabs), ignore it.
   If the line starts with .orig ignore it.
   If the line is a label on a line by itself, ignore it.
   If the line is a label followed by .fill, increment the ***lc*** and print 0000. (All fills will be .fill 0)
   If the line is the .end directive you should return from **secondPass**.
   If the line is an instruction, increment lc and call the appropriate get method as described in the next step. The get method will return the integer value of the machine language interpretation of the instruction.
   DO NOT INCREMENT LC IN THE GET FUNCTION. DO IT BEFORE CALLING THE GET FUNCTION.
   You should not have to do any error checking in part 2 since everything was checked for correctness in part 1.

8. Create a different function for getting each instructions machine language value. The opcodes we will be using are ADD, AND, NOT, LD, LDR, ST, STR, BR, and TRAP. Each function should take the instruction string as the first parameter. **If** the instruction uses a label **(i.e. uses pcoffset)** then also pass the labels array and the location counter as an integer in that order after the instruction string. The function should interpret the operands and return the integer value of the instruction.

   The name of the methods you must create are; ***getAdd, getAnd, getNot, getTrap, getLdr, getStr, getLd, getSt, and getBr,***.

   Create all of these functions in assemble.cc. Initially have them simply return 0.

   As you read a line of text from the input file, you should pass it to the get function which parses the data and returns the correct integer value. Upon return from the get function, **secondPass** should **PRINT** the value as a four-digit hexadecimal number. Do not print any leading hex indicator like x, X, 0x, or 0X. **The HEX values should use UPPER-CASE letters. That is use A, B, C, D, E, F. Don't use a, b, c, d, e, f.**

9. Complete each of the get functions. I would recommend finishing each get function in the order listed below. I would finish each function and thoroughly test it before moving on to the next. Test by running your code against small programs. **You MUST finish ADD and AND before finishing any of the pcoffset instructions.** ADD and AND are used as filler instructions for testing random branches and your tests will fail if these are not implemented first.

Test asm files are provided. They DO NOT cover all possibilities. You must modify these to test your program thoroughly. Put spaces and tabs between parts of the instruction. Make sure to use maximum and minimum possible values for the immediate instructions. Run your code on the asm file and verify each output by hand. You translate what the answer should be on a piece of paper and then make sure your program output is correct.

getAdd - The add instruction has both a register and an immediate form. Your getAdd function should recognize this appropriately.
getAnd - Very similar to getAdd. You could probably copy getAdd and make minor revisions.
getNot - Note that bits 0 through 4 are always 1 in the NOT instruction.
getTrap - The trap instruction will only have hex values x20 through x25 as operands.
getLdr - Note that the offset6 IS NOT a PCOFFSET. This instruction DOES NOT need labels or the lc.
getStr - Very similar to getLdr. You could probably copy getLdr and make minor revisions.
getLd - Uses PCOFFSET. Note that the LC should be incremented before calling. So when you calculate the PCOFFSET you DO NOT NEED TO ADD 1 because it has already been added to the LC.
getSt - Very similar to getLd. You could probably copy getLd and make minor revisions.
getBr - Uses the PCOFFSET. Pay particular attention to the way NZP are stored in the instruction.

10. **An example:** Interpreting the first instruction from test1.asm.

    The first line after the origin is: **ADD R1, R2, R3**

    This instruction says to add the value from register 2 to the value in register 3 and store the result in register 1.

    There could be multiple spaces or tabs before or after the word ADD.
    There could be multiple spaces or tabs between ADD and the first register.
    There will always be at least 1 space or tab between ADD and the first register.
    There could be no space or tab or multiple spaces and tabs before or after any comma.
    There will NEVER be any spaces or tabs between the R and the number following for register specifications.
    You don't have to worry about malformed instructions. I won't put any in your test files. If the instruction starts with the word ADD, then it will always be correctly formatted.

    You must scan through the line with a loop and find the important parts of the instruction.

    Assuming you read the first line from test1.asm.

    In **assemble,** you should find the word ADD as the first non-space, non-tab text. This would cause you to call **getAdd**, passing the entire line as an argument. ADD doesn't have a pcoffset so you **should not** pass the ***lc*** or the labels array.

    Note that there are two types of ADD instructions as shown in Figure A.2 of Appendix A of your book. Function **getAdd** must handle both.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Add (register)	0	0	0	1	DR			SR1			0	0	0	SR2		
Add (immediate)	0	0	0	1	DR			SR1			1	IMM5				

NOTE: Register ADD has a 0 in bit 5. Immediate ADD has 1 in bit 5.

NOTE: Bits 4 and 3 in register ADD are unused and set to 0.

Inside of get**Add** you do the following on the instruction line (**ADD R1, R2, R3)**

Scan for the destination register and determine that it is R1 (register 1).

Scan for the first source register and determine it is R2 (register 2).

Scan for the next operand. There are two adds and they differ by the third operand.

See figure A.2 in appendix A of your book.

If the next operand begins with an R, you know it is a register ADD.

If the next operand begins with a # symbol, you know it is an immediate ADD.

Since this instruction has R3 we know it is a register ADD and register 3 is the other source.

You scanned the following.

0001 – Add (ADD is always 0001, AND is always 0101, etc… Look at Appendix A of your book.)

001 – Register 1 is the destination register

010 – Register 2 is the first operand source register

011 – Register 3 is the is the second operand source register

The next step is to assemble the pieces into an integer using SHIFT (<<) and OR(|).

0001001010000011

return the value

11. **Another example - immediate values:** Interpreting the fourth instruction from test1.asm.

The first line after the origin follows. This instruction says to AND the value from register 3 with the immediate value -10 and store the result in register 0.

AND R0, R3, #-10

There could be multiple spaces or tabs before or after the word AND.
There could be multiple spaces or tabs between AND and the first register.
There will always be at least 1 space or tab between AND and the first register.
There could be no space or tab or multiple spaces and tabs before or after any comma.

You don't have to worry about malformed instructions. I won't put any in your test files. If the instruction starts with the word AND, then it will always be correctly formatted.

You must scan through the line and find the important parts of the instruction.

In **assemble,** you should find the word AND as the first non-space, non-tab text. This would cause you to call **getAnd**, passing the entire line as an argument. AND doesn't have a pcoffset so you don't need the lc or the labels array.

Note that there are two types of AND instructions as shown in Figure A.2 of Appendix A of your book.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
And (register)	0	0	0	1	DR			SR1			0	0	0	SR2		
And (immediate)	0	0	0	1	DR			SR1			1	IMM5				

NOTE: Register AND has a 0 in bit 5. Immediate AND has 1 in bit 5.

NOTE: Bits 4 and 3 in register AND are unused and set to 0.

Inside of **getAnd** you do the following on the instruction line (**AND R0, R3, #-10**)

Scan for the destination register and determine that it is R0 (register 0).

Scan for the first source register and determine it is R3 (register 3).

Scan for the next operand. There are two ands and they differ by the third operand.

See figure A.2 in appendix A of your book.

If the next operand begins with an R you know it is a register ADD.

If the next operand begins with a # you know it is an immediate ADD.

Since this instruction has a # you know it is an immediate value. So you must translate the number (-10) to a 5 bit 2's complement value: 10110.

You scanned the following.

0101 – Always this for AND.

000 – Register 0 is the destination register

011 – Register 3 is the first operand source register

10110 – negative 10 is the is the second operand 5 bit immediate value

The next step is to assemble the pieces into an integer using SHIFT (<<) and OR(|).

0101000011110110

Return the value.

12. DON'T PRINT ANY BLANK LINES IN THE OUTPUT!
13. At the end of this file is some help on getting started.
14. On AsULearn is a program called check2.cc. Download, compile, and run in the same folder as assemble.cc to ensure that you have declared all required functions with the proper parameters and return types.

    NOTE ON CHECK1.CC and CHECK2.CC – You MAY have to use these on student. There are library issues sometimes on locally installed copies of GCC. I think some regular expression libraries are different.

15. You should test your code thoroughly.

    You can compile using the following command. Download main2.cc and test1.asm from the AsULearn page. Make sure to store these in the same folder with your assemble.cc.

```
g++  -Wall  -Werror    -o assemble    assemble.cc    main2.cc
```

-Werror WILL STOP COMPILE ON WARNINGS. IF YOU IGNORE THE WARNINGS YOUR CODE MAY RUN DIFFERENTLY ON Web-CAT AND MAY CAUSE YOU TO FAIL TESTS EVEN IF EVERYTHING LOOKS GOOD IN YOUR TESTS.

-Werror won't show everything that can cause failures, but it can find some problems for you. Clear all warnings to be safe.

Run your code with the following command. By default this tests your code against test1.asm.

```
./assemble
```

You can test other asm files by putting the name after the ./assemble command like this.

```
./assemble  test2.asm
```

16. Submit your ***assemble.cc*** file to Web-CAT under the part 2 assignment. You will receive 50 points for part 1 and 50 points for part 2.

test1.asm – This is an example of each instruction

For this example only, note that the program was read into x3000.

This is one big program so the first "ADD" is stored at x3000, the second "ADD" is at x3001, etc…

What your program should print	The lines from test1.asm
Nothing	;Test file for assembly
Nothing	
Nothing	;This is a comment
Nothing	
3000	.orig x3000
1283	ADD R1, R2, R3
10EA	add R0, R3, #10
Nothing	
5242	and R1, R1, R2
50F6	AND R0, R3, #-10
Nothing	L0
0E09	BR L1 ;Note that BR should BR translated as BRNZP
09FE	BRN L0
0407	BRZ L1
03FC	BRP L0
0C05	BRNZ L1
0BFA	BRNP L0
0603	BRZP L1
0FF8	BRNZP L0
220A	LD R1, L2
62BB	LDR R1, R2, #-5
Nothing	L1
973F	NOT R3, R4
3A07	ST R5, L2
7C7B	STR R6, R1, #-5
F020	TRAP x20
F021	TRAP x21
F022	TRAP x22
F023	TRAP x23
F024	TRAP x24
F025	TRAP x25
0000	L2 .FILL 0
0000	L3 .FILL 0
0000	L4 .FILL 0
Nothing	
Nothing	.END

This is not a functioning program, so don't try to execute it on any LC3 simulators. It will give errors. This program is simply meant to give you an example of all of the statements you should be able to interpret.

./assemble test1.asm should print the following

```
3000
1283
10EA
5242
50F6
0E09
09FE
0407
03FC
0C05
0BFA
0603
0FF8
220A
62BB
973F
3A07
7C7B
F020
F021
F022
F023
F024
F025
0000
0000
0000
```

this is test1.asm with addresses listed

```
;Test file for assembly

;This is a comment

.orig x3000
        ADD R1, R2, R3              12288
        add R0, R3, #10             12289

        and R1, R1, R2              12290
        AND R0, R3, #-10            12291
L0
        BR L1                       12292 <--L0
        BRN L0                      12293
        BRZ L1                      12294
        BRP L0                      12295
        BRNZ L1                     12296
        BRNP L0                     12297
        BRZP L1                     12298
        BRNZP L0                    12299
        LD R1, L2                   12300
        LDR R1, R2, #-5             12301
        L1
        NOT R3, R4                  12302 <-- L1
        ST R5, L2                   12303
        STR R6, R1, #-5             12304
        TRAP x20                    12305
        TRAP x21                    12306
        TRAP x22                    12307
        TRAP x23                    12308
        TRAP x24                    12309
        TRAP x25                    12310
        L2      .FILL   0           12311 <--L2
L3      .FILL   0                   12312 <--L3
L4      .FILL   0                   12313 <--L4

.END                                STOP HERE

haha
```

Some unHelpful Hints

1. You may want to create a method for removing leading and trailing spaces from a string. You may want to remove all spaces. If you remove all spaces then you can be sure of the exact position of things like register numbers and immediate values.
2. You may want to create a method for turning all lowercase characters in a string to uppercase. This is an easy way to make sure case isn't an issue. If you convert all to uppercase it will take care of things like .orig, .ORIG, .Orig, .OrIg, .oriG very easily.

   Inside of a for loop

   ```
   if ( c[i] >= 'a' && c[i] <= 'z') c[i] = c[i] - 32;
   ```

3. Check out the following functions: strtok, sscanf, fscanf, rewind.
4. Note that strncmp compares the first n characters of a string (like startsWith in Java).
5. If you need to turn a single character into an int, simply subtract 48. This is especially useful for labels and registers.

   ---L1 (Assume this line is read into a char array named line. The dashes represent spaces in this example only.)

   If I find that L is at position 3, I can easily find the label number using the following:

   int v = line[4] - 48;

   or if the line read is "ADD R1, R2, R3"

   ```
   int r1 = line[5] - 48;
   int r2 = line[9] - 48;
   int r3 = line[13] - 48;
   ```

   the trick is finding the Rs.

6. **DO NOT** use sscanf to read a character as a number.

   ```
   char ch = *line;
   sscanf(&c, "%d", &n);
   ```

   scanf reads the character in ch, but then tries to read the character AFTER ch and you just don't know what is stored there. Your program will run sometimes but fail at other times, such as on the test.
7. Looping through a null terminated string (s).
   for (int i = 0; s[i] != 0; i++)
8. DO NOT use the ***exit*** function. It will exit the testing program and any uncompleted tests will not execute.
9. Always initialize variables and strings.
   int num; //Who knows what is in num, can lead to weird errors.
   int num = 0; //Now we know exactly what is in num
   char str[128]; //Using string functions or printing can have odd results.
   char str[128]; str[0]=0; //Now str is an empty string.

10. Careful copying the asm files to your local directories. Some editors will remove all tabs and replace them with spaces. If that happens and you don't account for tabs in your code (because your test file didn't have them), then your program will run correctly on your test files but fail on the test files on Web-CAT because they DO have tabs in them.

Error Table

Error	Display Message
Origin not first line in the asm file.	ERROR 1: Missing origin directive. Origin must be first line in program.
Origin address too big. It is greater than the maximum value possible to be represented in 16 bits.	ERROR 2: Bad origin address. Address too big for 16 bits.
Unknown instruction. If an instruction starts with ABC for example.	ERROR 3: Unknown instruction.
Missing end. If the file runs out of data before the .end directive is found.	ERROR 4: Missing end directive.

If one of the above errors is encountered, display the message EXACTLY as shown followed by a newline and then return -1 from **firstPass** or **findOrigin** to indicate an error

No other errors need be checked.

You do not have to worry about malformed instructions. If an instruction starts with a proper opcode, then you can assume the line will be correctly formed as shown in the appendix A. You do, however, need to take spacing into account.

Errors should be in order from 1 to 4 as listed above. So if a file has no .orig and no .end, it should display the missing origin message.

I would highly suggest you read the file one line at a time instead of character by character.

```
//The following will read and print a file.
//Limit is a #define,  infile is an opened file pointer, line is a char array.
while (fscanf(infile, "%[^\n]s", line) != EOF && count < LIMIT)
{
	char discard = fgetc(infile);  //Get rid of the newline that we didn't read with the fscanf
	printf("%s\n", line);

	//strncmp only compares the first n characters of a string.  Can act like startsWith in java.
	if (strncmp(line, ".orig", 5) == 0) //Of course this only works if no spaces or tabs before .orig on line
		//Process origin		//And it also doesn't work with .ORIG
	else if (strncmp(line, ".end", 4) == 0)
		//process  .end
	else …


	line[0] = 0;  //Initialize line to empty string.
		//Needed for blank line detection if line ONLY contains \n then fscanf won't read anything.
}
```