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. If you do not have the ability or the determination to ask for help on part 1, you will not be able to do the second part. You have been warned.

SIZE LIMIT – There is a size limit on part 1 of 16K. Several submissions are in the 6-7K range. Some are as high at 10K. If your program takes more that 16K you are doing things very, very inefficiently. Come talk to me about what you are doing wrong. The size limit for part 2 is 32K.

The Assignment - Part 1 - The First Pass

1. For this entire assignment, case is unimportant. That is your assembler should recognize .ORIG or .orig.
2. For this entire assignment, if you print anything, it must always be followed by a newline.
3. For this entire assignment, the only allowed libraries are stdio.h, stdlib.h, and string.h.
4. **ALL ERROR MESSAGES are listed in a table at the end of this document. You must print error messages EXACTLY as specified. Functions findOrigin and first pass should return -1 to indicate errors.**

   YOU MUST DETECT ALL ERRORS in part 1.

5. Copy the starter file named ***assemble.cc*** from AsULearn into your working directory. The rest of this document is going to assume you will copy and create files in the same directory as assemble.cc.

   You are free to work in any environment you want, but you should make sure everything compiles and runs on student before submitting to Web-CAT.

6. For this assignment, you will need to have a source assembly file and open it for reading. Section 18.5 of your book talks about file I/O in C. I am not going to cover it in class. I will answer questions about it, however. The file that you will be using for testing is called ***test1.asm***. Copy ***test1.asm*** from the AsULearn page. You can and should create other test files to test other formats. The test1.asm file from AsULearn DOES NOT cover every possible configuration of all instructions.

7. You are given an ***assemble*** function. Your job in this part will be to create the ***findOrigin, firstPass***, and ***printLabels*** functions as called from ***assemble***.

8. Variables in **assemble**.

   a. The string ***filename*** will be passed to your assemble function from main. It is the name of the .asm file that should be assembled.

   b. The FILE pointer infile is used to interact with the file on the hard drive. One of the useful things we can do is rewind a file to the beginning, so we can search through it again. You will need this ability later.

      ```
      //Reset file scanning back to beginning.
      rewind(infile);
      ```

   c. Labels in the LC3 represent addresses. Labels can be used as jump points for loops or branches or can be used like variables. Labels for this assignment are going to be very specific.

      All labels will begin with the letter L (upper or lower case) followed by a single digit.
      The only labels used in this assignment will be L0 through L9.

      To be able to assemble each instruction, it is important that ALL label addresses be resolved before the instruction translation begins. We will use something called a "Two Pass Assembler". The first pass is part 1 of this assignment. Our first pass will read the entire file but will simply be looking for labels and saving the memory locations associated with each. Later, the second pass, will read through the file again and will translate the instructions.

      This array will hold the memory locations of the addresses of each label
         labels[0] should hold the address of L0
         lables[1] should hold the address of L1
         etc…

d. The ***lc*** variable.

   This variable represents the location counter and is the memory location that the next instruction read from the file should be loaded in memory. It functions the same as the program counter (PC) in the hardware. The origin is how an assembly programmer for the LC3 specifies where to load this program. So, the initial value of ***lc*** should be the value specified in the origin since this is the position of the first instruction.

9. You are given some starter code for ***findOrigin*** which takes a file pointer as an argument. When you have completed ***findOrigin,*** it should return the origin value as an integer.

   The while loop given to you will search through the infile one line at a time. The value of each line of the assembly program will be read into the char array line as a null terminated string.

   Your job is to look through the string and see if it begins with .orig. If the origin is found, read the address as a hex or decimal value and return it.

   If the first line of the program (not counting blank lines and comments) is NOT an origin line, findOrigin should print the origin not found error message and return -1.

   If you reach the end of file while searching for the origin you should print the origin not found error and return -1.

   If the address value read is too big, you should print an error message and return -1.

   An origin line has the following format.

   ```
   .orig  x3000                    or                    .orig  12356
   ```

   NOTE: The **.orig** or the **x** can be upper or lower case.
   NOTE: There could be multiple spaces or tabs before or after the .orig.
   NOTE: There will always be at least one space between the .orig and the address.
   NOTE: The address can be any number of digits but will always fit into an int.
   If an x or X is present in the value, it indicates that the number is a hexadecimal number. If there is no x or X then the number indicated is decimal.

10. Create a new function named **firstPass** which takes the file pointer, the labels array, and the integer location counter (***lc***) as arguments and returns an int for an error code. This function should read the assembly file and process and save all label addresses as described in the next few steps.
11. Put a call to **firstPass** near the top of the **assemble** function after the call to **findOrigin**. Do not run **firstPass** if **findOrigin** returned -1.

12. Function **firstPass** should rewind the input file. Then it should read each line and interpret as described below.

    Read a line.

    If the line is a comment, a blank line or the .orig directive, don't do anything.
    If the line is a label on a line by itself, save the lc to the labels array at the appropriate position.
    For example, if L3, store the lc value in labels[3].
    If the line is a label followed by .fill, save the lc in labels AND increment lc.
    If the line is .end, return 0 for done with no error.
    If the end of file is reached before .end is found print the error and return -1.
    If the line is one of the allowed instructions (ADD, AND, NOT, LD, LDR, ST, STR, BR, and TRAP) increment the lc.
    If the line is anything else print the unknown instruction error and return -1.

    The assembly language program will have labels in two different formats. You need to differentiate between the two. There will not be any badly formatted code other than specific errors listed which means ANY line that begins with a label and does not have any text following can be considered to be the first label format. Anything else will be the second label format. (CAREFUL. The L1 below could be preceded or followed by spaces and tabs and still count as a label on a line by itself.)

    FIRST FORMAT: You will see a label on a line by itself like this indicating a loop or branch. All you have to do in this case is save the lc to the labels array.

    ```
    L1                              ;C code would be:    labels[1] = lc;
            ADD R1, R2, R3
    ```

    SECOND FORMAT: Or you will see a label like the following which represents labels used as variables. In this case you must save the lc to the labels array and then increment the lc.

    ```
    L4      .fill     0             ;C code would be:   labels[4] = lc;  lc++;
    ```

    NOTE THAT YOU HAVE TO BE CAREFUL BECAUSE LD AND LDR ALSO BEGIN WITH L

13. Create a function named ***printLabels*** which takes the labels array as an argument. Print labels should simply print the labels array in the following format on a single line with a single newline at the end. All values should be printed in decimal. There should be one space after each comma. There should be one space before AND after the = symbol. There should be no other spaces.

    ```
    labels = {-1, -1, 12289, -1, 12292, -1, -1, -1, -1, -1}
    ```

14. Put a call to **printLabels** in the **assemble** function, just after the call to **firstPass**. Do not call **printLabels** if **firstPass** or **findOrigin** returned an error.

15. You should test your code thoroughly with different asm files with different labels.

    You can compile using the following command. Download main0.cc, main1.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    main0.cc
```

-Wall and -Werror WILL STOP COMPILE ON WARNING. 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.

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

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
```

After you get everything working with main0.cc. Recompile and test with main1.cc. The main1.cc program will individual functions.

16. On AsULearn is a program called check1.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.

17. After thoroughly testing your code, submit your ***assemble.cc*** file to Web-CAT as part 1 of this assignment. You will receive 50 points for part 1 and 50 points for part 2.

    You MUST complete part 1 to be able to do part 2. If you do not have the ability or the determination to ask for help on part 1, you will not be able to do the second part. You have been warned.

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
```

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 single character as a number. It is fine to use sscanf to read the origin but it is not good to use for single digit numbers like the digit on a register or a label.

   Assume line has been advanced past the L in a label and is pointing at the label number.

   ```
   char ch = *line;  //The character goes into ch
   sscanf(&ch, "%d", &n); //Weird things can happen here
   ```

   Sscanf reads multidigit numbers from a string. Sscanf reads the character in ch, but then tries to read the character AFTER ch to see if it is also a number 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. There is a sscanf_example.cc file in the files folder.

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. This is the equivalent in Java to str="";
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.