

# CIT 593 Homework 11: C - File I/O – *Making the LC4 Assembler*

**Due Date: Monday 12/4 @11:59pm via canvas upload ONLY**

**READING:** *Chapter 11 of the book (basics of C), Chapter 12 (variables), Chapter 16 (pointers & arrays) are excellent references for this homework assignment.* If you have purchased a C-programming book, then you'll want to look to its chapter on strings in C.

## **SETUP FOR THIS HOMEWORK:**

The problems assigned below require the use of the Clang C-Compiler for x86 loaded on the VM

- **VIDEO-TUTORIAL-DEBUGGING in C:** If you are getting segfaults and having trouble with your program, watch this video on canvas to help you learn how to use the GDB debugger: Files->Resources->Tutorials-> *Video\_Tutorial-Debug-in-C\_GDB\_Valgrind\_Logging*
- **VIDEO-TUTORIAL-MAKEFILES:** If you are still struggling to understand Makefiles even after the last assignment as well as recitation, try this video: Files->Resources->Tutorials-> *Video\_Tutorial-Makefiles*

## **CREATING FOLDERS ON THE VM & USING TEXT EDITORS PRIOR TO ASSIGNMENT:**

- 1) Login to your VM, open the **terminal** and create a hw11 folder by typing:  

```
mkdir ~cis240/hw11
```

*You may alternatively use the File Manager in Ubuntu to do this*
- 2) Download the file: CIT593\_HW11\_C-FileIO\_HelperFiles.zip from canvas  
Be certain to copy it into the cis240/hw11 folder you created  
Unzip the file into your new HW11 folder by typing:  

```
cd ~cis240/hw11  
unzip CIT593_HW11_C-FileIO_HelperFiles.zip
```
- 3) Use the sublime text editor to edit the C program from the zip file for the first problem by typing:  

```
cd ~cis240/hw11  
subl assembler.c
```

## CIT 593 Homework 11: C - File I/O – *Making the LC4 Assembler*

### Assignment Overview:

From lecture you've learned that C is file-oriented and that working with files represents I/O devices in C. C places files into two categories: "text" and "binary." In this assignment you'll work with both types by reading in a text file and writing out a binary file. The text file that you will read in this assignment will be a .ASM file (a text file intended for PennSim) and the type of output file you'll generate will be a .OBJ file (the same type of binary file that PennSim would write out). Aside from reading and writing out the files, your task will be make a mini-LC4-Assembler! A program that reads in assembly language and generates its machine equivalent. This assignment will require a bit more programming rigour than we've had thus far, but now that you've gained a good amount of programming skill in this class and in others, it is the perfect time to tackle a large programming assignment.

### Problem #1: Reading in a text file (the .ASM file)

Open "assembler.c" from the helper files; it contains the main() function for the program.

Carefully examine the variables at the top:

```
char* filename = NULL ;
char  program [ROWS][COLS] ;
char  program_bin_str [ROWS][17] ;
unsigned short int program_bin [ROWS] ;
```

The first pointer variable "filename" will be a pointer to a string that contains the text file you'll be reading. Your program must take in as an argument the name of a .ASM file. As an example, once you compile your main() program, you would execute it as follows:

```
./assembler test.asm
```

In your last HW you learned how to use the arguments passed into main(). So the first thing to implement is to check if argc has arguments, and if it does, point "filename" to the argument that contains the passed in string that is the file's name. You should return from main() immediately (with an error message) if the caller doesn't provide an input file name as follows:

```
error1: usage: ./assembler <assembly_file.asm>
```

Start by updating "assembler.c" to read in the arguments. Compile your changes and test them before continuing. You should take this moment to setup your Makefile as well, it should contain **two** basic directives: **assembler** and **asm\_parser.o**

## CIT 593 Homework 11: C - File I/O – *Making the LC4 Assembler*

After you’ve successfully gotten the filename from the caller, the first function you must call will be:

```
int read_asm_file (char* filename, char program [ROWS][COLS] ) ;
```

The purpose of `read_asm_file()` is to open the ASM file, and place its contents into the 2D array: `program[][]`. You must complete the implementation of this function in the provided helper file: “`asm_parser.c`”. Notice, it takes in the pointer to the “filename” that you’ll open in this function. It also takes in the two dimensional array, *program*, that was defined back in `main()`. You’ll see that “ROWS” and “COLS” are two `#define`’ed constants in the file: `asm_parser.h`. Rows is set to 100 and COLS is set to 255. This means that you can only read in a program that is up to 100 lines long and each line of this program can be no longer than 255.

You’ll want to look at the class notes (or a C-reference textbook) to use `fopen()` to open the filename that has been passed in. Then you’ll want to use a function like: `fgets()` to read each line of the .ASM file into the `program[][]` 2D array. Be aware that “`fgets()`” will keep carriage returns (aka – the newline character) and you’ll need to strip these from the input.

Take a look at `test.asm` file that was included in the helper file. It contains the following program:

```
ADD R1, R0, R1
MUL R2, R1, R1
SUB R3, R2, R1
DIV R1, R3, R2
AND R1, R2, R3
OR R1, R3, R2
XOR R1, R3, R2
```

After you complete `read_asm_file()` and if you were to run it on `test.asm`, your 2D array: `program[][]` should contain the contents of the ASM file in this order:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	A	D	D		R	1	,		R	0	,		R	1	'\0'
1	M	U	L		R	2	,		R	1	,		R	1	'\0'
2	S	U	B		R	3	,		R	2	,		R	1	'\0'
3	D	I	V		R	1	,		R	3	,		R	2	'\0'
4	A	N	D		R	1	,		R	2	,		R	3	'\0'
5	O	R		R	1	,		R	3	,		R	2	'\0'	X
6	X	O	R		R	1	,		R	3	,		R	2	'\0'
7	'\0'	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Notice, there are no “newline” characters at the end of the lines.

If reading in the file is a success, return 0 from the function, if not, return 2 from the function and print an error to the screen: `error2: read_asm_file() failed`

Implement and test this function carefully before continuing on with the assignment.

## CIT 593 Homework 11: C - File I/O – *Making the LC4 Assembler*

### Problem #2: Parsing an Instruction

Once `read_asm_file()` is working properly, back in `main()`, you'll call the function: `parse_instruction()`, which is also located in `asm_file.c`:

```
int parse_instruction (char* instr, char* instr_bin_str) ;
```

#### **purpose, arguments & return value**

The purpose of this function is to take in a single row of your `program[][]` array and convert to its binary equivalent – in text form. The argument: `instr` must point to a row in `main()`'s 2D array: `program[][]`. The argument: `instr_bin_str` must point to the corresponding row in `main()`'s 2D array: `program_bin_str[][]`. If there no errors are encountered the function will return a 0 and if any error occurs (in this function) it should return the number 3 and an error message should be printed: `error3: parse_instruction() failed.`

Let's assume you've called `parse_instruction()` and "`instr`" points to the first row in your `program[][]` array:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
*instr->	A	D	D		R	1	,		R	0	,		R	1	'\0'

`Parse_instruction()` needs to examine this string and convert it into a binary equivalent. You'll need to use the LC4 ISA to determine the binary equivalent of an instruction. When your function returns, the memory pointed to by: `instr_bin_str`, should look like this:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
*instr_bin_str->	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	1	'\0'

Notice, this isn't actually binary, but it is the "ADD" instruction's binary equivalent in TEXT form. We will convert this string form of the binary instruction to HEX later in the function: `convert_instruction()`.

## CIT 593 Homework 11: C - File I/O – *Making the LC4 Assembler*

### **How to implement this function**

The purpose of converting the instruction to a binary string (instead of to the binary # it will eventually become), is so that you can build this string up little by little. Investigate the “strtok()” function in the standard C string library:

[https://www.tutorialspoint.com/c\\_standard\\_library/c\\_function\\_strtok.htm](https://www.tutorialspoint.com/c_standard_library/c_function_strtok.htm)

STR Tok() allows you to parse a string that is separated by “tokens”. In this function you’ll be parsing the string pointed to be “instr” and you’ll be building up the string pointed to by “instr\_bin\_str”. “instr” will contain spaces and commas (those will be your tokens). Your first call to strtok() on the “instr” string should return back the OPCODE: ADD, SUB, MUL, DIV, XOR, etc. The only thing common to all 26 instructions in the ISA is that the very first part of them is the opcode. Once you determine the OPCODE, you’ll call the appropriate helper function to parse the remainder of the instruction.

As an example, let’s say the OPCODE is ADD. For this problem, you do not need to worry about the “immediate” variants of the ADD instruction (or AND immediate) – when we test your code, we won’t use ADD – immediate or AND immediate instructions. Once you’ve determined the OPCODE is ADD, you would call the parse\_add() helper function. It will take the instruction (instr) as an argument, but also the *instr\_bin\_str* string because parse\_add will be responsible for determining the binary equivalent for the ADD instruction you are currently working on and it will update instr\_bin\_str.

```
int parse_add (char* instr, char* instr_bin_str ) ;
```

When parse\_add() returns, if no errors occurred during parsing the ADD instruction, instr\_bin should now be complete. At this time, you can return a 0 from parse\_instruction().

## CIT 593 Homework 11: C - File I/O – *Making the LC4 Assembler*

### Problem #3: Parsing an ADD instruction

```
int parse_add (char* instr, char* instr_bin_str ) ;
```

The helper function: `parse_add()`, should be called only by the `parse_instruction()` function. It has two `char*` arguments: `instr` and `instr_bin_str`. Because this function will only be called when an ADD OPCODE is encountered by `parse_instruction()`, `instr` will contain an ADD instruction and `instr_bin_str` should be empty. `[]`. If there no errors are encountered the function will return a 0 and if any error occurs (in this function) it should return the number 4 and an error message should be printed: `error4: parse_add() failed.`

The purpose of this function is to populate the `instr_bin_str`. Upon the function's start, the binary OPCODE can be immediately copied into the `instr_bin_str[0:3]`. Afterwards, the `strtok()` function can be used again to separate the registers: RD, RS, RT, from the "instr" string.

For each register: RD, RS, RT, the `parse_reg()` helper function should be called:

```
int parse_reg (char reg_num, char* instr_bin_str) ;
```

This function must take as input a number in character form and populate `instr_bin_str` with the appropriate corresponding binary number. For example, if RD = R0 for the ADD instruction, the '0' character would be passed in the argument: `reg_num`. `parse_reg()` would then copy into `instr_bin_str[4:6]` the characters '000'. A 5 should be returned if any errors occur and the standard error message should be printed; otherwise a 0 should be returned upon success.

To implement the `parse_reg()` function, you should consider using a `switch()` statement:

[https://www.tutorialspoint.com/cprogramming/switch\\_statement\\_in\\_c.htm](https://www.tutorialspoint.com/cprogramming/switch_statement_in_c.htm)

This helper function: `parse_reg()` should only parse one register at a time. Also, because it is not specific to the ADD instruction (nearly all instructions contain registers), it can be called from other functions that need their register's converted to binary. Example: `parse_mul()` should also call `parse_reg()`.

Note that `parse_add()` must also populate the SUB-OPCODE field in `instr_bin_str[10:12]`. When `parse_add()` returns, `instr_bin_str` should be complete. `parse_instruction()` should then return to `main()`. You will need to create a helper function for each instruction type, use `parse_add()` as a model. As an example, you'll need to create: `parse_mul()`, `parse_xor()`, etc – they will all be very similar functions, so perfect `parse_add()` before you attempt those other functions.

## CIT 593 Homework 11: C - File I/O – *Making the LC4 Assembler*

### Problem #4: Converting the binary string to hex

After `parse_instruction()` returns successfully to main, the function:

```
unsigned short int str_to_bin (char* instr_bin_str) ;
```

Should be called from `main()`, passing the recently parsed binary string from the array: `program_bin_str[X]`, where “X” represents the binary instruction that was just populated by the last call to `parse_instruction()`.

The purpose of this function is to take a binary string and convert it to a 16-bit binary equivalent and return it to the calling function. To implement this function, the string function, `strtol()` is recommended:

[https://www.tutorialspoint.com/c\\_standard\\_library/c\\_function\\_strtol.htm](https://www.tutorialspoint.com/c_standard_library/c_function_strtol.htm)

If a zero is returned from `strtol()`, return the error #6 should be returned and printed to the screen.

As an example of what this function should do, if it was called with the following argument:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
*instr_bin_str->	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	1	'\0'

then it should return: `0x1201`, that is the hex equivalent for this binary string. You can verify and print out what it returns by using `printf ("0x%X")`, which will print out integers in HEX format.

Once `str_to_bin()` is returned, it should be assigned to the corresponding spot in the unsigned short int array: `program_bin[X]`, where X matches the index from `program_bin_str[X]`.

## CIT 593 Homework 11: C - File I/O – Making the LC4 Assembler

*...back in main():*

The sequence of events in main() should be as follows:

- 1) Initialize all arrays to zero or '\0'
- 2) Call read\_asm\_file() to read the entire ASM file into the array: program[[]]. Using test.asm as an example, after read\_asm\_file() returns: program[[]] should then contain:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	A	D	D		R	1	,		R	0	,		R	1	'\0'
1	M	U	L		R	2	,		R	1	,		R	1	'\0'
2	S	U	B		R	3	,		R	2	,		R	1	'\0'
3	D	I	V		R	1	,		R	3	,		R	2	'\0'
4	A	N	D		R	1	,		R	2	,		R	3	'\0'
5	O	R		R	1	,		R	3	,		R	2	'\0'	X
6	X	O	R		R	1	,		R	3	,		R	2	'\0'
7	'\0'	X	X	X	X	X	X	X	X	X	X	X	X	X	X

*In a loop, for each row X in program[[]] {*

- 3) Call parse\_instruction(), passing it the current row in program[X][] as input to parse\_instruction(). When parse\_instruction() returns program\_bin\_str[X][] should be updated to have the binary equivalent (in string form).
- 4) Call str\_to\_bin() passing program\_bin\_str[X][] to it. Upon str\_to\_bin()'s return, program\_bin[X] should be updated to have the hex equivalent of the binary string from program\_bin\_str[X].

*} once the loop is complete program\_bin\_str[[]] should contain program[[]] equivalent:*

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	1	'\0'
1	0	0	0	1	0	1	0	0	0	1	0	0	1	0	0	1	'\0'
2	0	0	0	1	0	1	1	0	1	0	0	1	0	0	0	1	'\0'
3	0	0	0	1	0	0	1	0	1	1	0	1	1	0	1	0	'\0'
4	0	1	0	1	0	0	1	0	1	0	0	0	0	0	1	1	'\0'
5	0	1	0	1	0	0	1	0	1	1	0	1	0	0	1	1	'\0'
6	0	1	0	1	0	0	1	0	1	1	0	1	1	0	1	0	'\0'
7	'\0'	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Also after the loop is complete, the array: **program\_bin[]** should contain program\_bin\_str[[]]'s equivalent in binary (its shown in hex here):

0	0x1201
1	0x1449
2	0x1691
3	0x12DA
4	0x5283
5	0x52D3
6	0x52DA

program\_bin[] now represents the completely assembled program and can be written out to a .OBJ file in binary form.



## CIT 593 Homework 11: C - File I/O – *Making the LC4 Assembler*

### Problem #5: Writing out the object file

During lecture, we learned that a .OBJ file is a binary file. The format of the .OBJ file is discussed in lecture 11, slides 12-13. Recall from lecture that the .OBJ file's "CODE" header is as follows:

- *Code:* 3-word header (xCADE, <address>, <n>), n-word body comprising the instructions. This corresponds to the .CODE directive in assembly. *Recall that a "word" is 16-bits on the LC4.*

Given this information, the last function to implement is:

```
int write_obj_file (char* filename,
                   unsigned short int program_bin[ROWS] ) ;
```

The purpose of this function is to take the assembled program, represented in hex in `program_bin[]` and output it to a file with the extension: .OBJ. It must encode the file using the .OBJ file format specified in class. If "test.asm" was pointed to by filename, your program would open up a file to write to called: test.obj.

This function should do the following:

- 1) Take the filename passed in the argument "filename", change the last 3 letters to "obj"
- 2) Open up that file for writing and in "BINARY" format. NOTE, the file you'll create is not a text file, these are not strings you're writing, they are binary numbers.
- 3) Write out the first word in the header: 0xCADE
- 4) Write out the address your program should be loaded at: 0x0000 is the default we'll use
- 5) Count the number of rows that contain data in `program_bin[]`, then write out <n>
- 6) Now that the header is complete, write out the <n> rows of data in `program_bin[]`
- 7) Close the file using `fclose()`.

If any errors occur, return #7 and print the appropriate error message. Otherwise return a 0 and `main()` should then return 0 to the caller.

Examine your .OBJ file's contents. Use the utility `hexdump` to view your .OBJ file. From the linux prompt type:

```
hexdump test1.obj
```

Hexdump will show you the binary contents. Make certain it matches your expectation! It is your responsibility to test out files other than test1.asm. Also, you can test your .OBJ files by loading them into PennSim and seeing if they work! Please do this before submitting your work.

## CIT 593 Homework 11: C - File I/O – *Making the LC4 Assembler*

### **Extra Credit Opportunities:**

- 1) 2 points – improve your `read_asm_file()` to have it ignore comments in .ASM files
- 2) 5 points – improve your program to accept the .CODE and .ADDR directives. You'll need another array to hold onto addresses. You can call it: `unsigned short int addresses[ROWS]`.
- 3) 2 points – improve your program to handle ADD immediate and AND immediate instructions
- 4) 1 point for each additional instruction your program can support (including pseudo instructions), 1.5 points each, for supporting instructions that demand LABELS
- 5) 2 points – improve your program to handle .DATA directives.

## CIT 593 Homework 11: C - File I/O – *Making the LC4 Assembler*

### Directions on how to submit your work:

- Create a single zip file called: **LAST\_FIRST\_HW#.zip**
- The zip file should contain ONLY the files named in this assignment.
- There should not be any sub-directories within your zip file.

This single zip file would contain the following files (italicized names are for extra credit) :

**Makefile**  
**asm\_parser.h**  
**asm\_parser.c**  
**assembler.c**  
**test1.asm**

You will then upload ONLY 1 file to **canvas**: **FARMER\_THOMAS\_HW11.zip**

- DO NOT TURN IN ANY **.o** files or final **object** files: like program1 as an example
- Make certain that you submit the latest versions of your code.
- Submitting using any other compression type (.RAR, TAR, GZIP) will be rejected.

Paper/Email submissions will not be accepted for this assignment.