**Task #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 test1.asm­

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

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 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:

A picture containing old

Description automatically generatedNotice, 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 task.

**Task #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 2Darray: 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:

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 (I CAN LOOK THIS UP FOR YOU). When your function returns, the memory pointed to by: instr\_bin\_str, should look like this:



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\_instrunction().

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**

STRTOK() 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 instrunction, instr\_bin

should now be complete. At this time, you can return a 0 from parse\_instrunction().

**Task #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**

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: prase\_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\_instrunction() 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.

**Task #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**

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:



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].

**…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:

Table

Description automatically generated

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:

A picture containing old, light, stone, square

Description automatically generated

Also after the loop is complete, the array: program\_bin[] should contain program\_bin\_str[][]’s

equivalent in binary (it’s shown in hex here):

Table

Description automatically generated

program\_bin[] now represents the completely assembled program and can be written out to a

.OBJ file in binary form.

0

2

**Task#5: Writing out the object file**

a .OBJ file is a binary file. 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! Note:

depending on how you write out your files, you may encounter “endianness” variations. Look

carefully at the word document called Project12 Helper to learn about endianness. Your .OBJ file must have the correct endianness to be loaded into PennSim. YOU MUST TEST YOUR .OBJ FILES IN PENNSIM