## Setting up Codio for this HW:

1) Open the Codio assignment via Coursera
2) From the Codio File-Tree click on: **lc4_memory.h** and **lc4_memory.c**

## Overview:

The goal of this HW is for you to write a program that can open and read in a .OBJ file created by PennSim, parse it, and load it into a linked list that will represent the LC4's program and data memories (similar to what PennSim's "loader" does). In the last HW, you created a .OBJ file. In this HW, you will be able to read in a .OBJ file and convert it back to the assembly it came from! This is known as reverse assembling (sometimes a disassembler).

## RECALL: OBJECT FILE FORMAT

The following is the format for the binary .OBJ files created by PennSim from your .ASM files. It represents the contents of memory (both program and data) for your assembled LC-4 Assembly programs. In a .OBJ file, there are 3 basic sections indicated by 3 header "types" = CODE, DATA, SYMBOL.

- *Code:* 3-word header (xCADE, <address>, <n>), n-word body comprising the instructions. This corresponds to the .CODE directive in assembly.
- *Data:* 3-word header (xDADA, <address>, <n>), n-word body comprising the initial data values. This corresponds to the .DATA directive in assembly.
- *Symbol:* 3-word header (xC3B7, <address>, <n>), n-character body comprising the symbol string. Note, each character in the file is 1 byte, not 2. There is no null terminator. Each symbol is its own section. These are generated when you create labels (such as "END") in assembly.

**LINKED LIST NODE STRUCTURE:**

In the file: lc4_memory.h, you'll see the following structure defined:

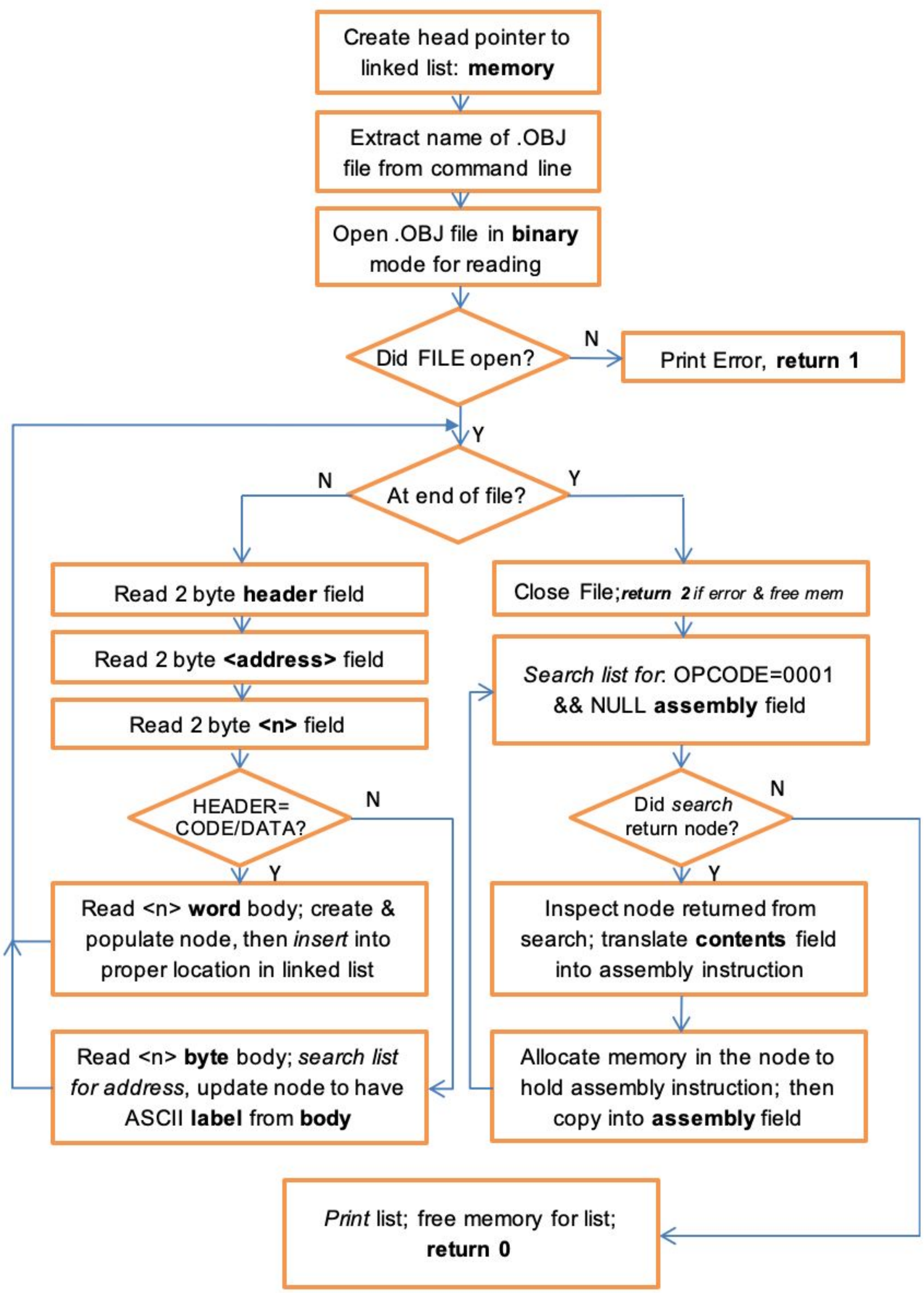
```
struct row_of_memory {
	short unsigned int address ;
	char * label ;
	short unsigned int contents ;
	char * assembly ;
	struct row_of_memory *next ;
} ;
```

The structure is meant to model a row of the LC4's memory: a 16-bit ***address***, & its 16-bit ***contents***. As you know, an address may also have a ***label*** associated with it. You will also recall that PennSim always shows the contents of memory in its "***assembly***" form. So PennSim reverse-assembles the contents and displays the assembly instruction itself (instead of the binary contents).

As part of this assignment, you will read in a .OBJ file and store each instruction in a NODE of the type above. Since they'll be an unknown # of instructions in the file, you'll create a linked list of the nodes above to hold all the instructions that are in the .OBJ file.

The details of how to implement all of this will be discussed in the sections of this document that follow.

## FLOW CHART: Overview of Program Operation



## IMPLEMENTATION DETAILS:

The first files to view in the helper file are **lc4_memory.h** and **lc4_memory.c**. In these files you will notice the structure that represents a **row_of_memory** as referenced above (see the section: ***LINKED_LIST_NODE_STRUCTURE*** above for the node's layout). You will also see several helper functions that will serve to manage a linked list of "rows_of_memory" nodes. Your job will be to implement these simple linked list helper functions using your knowledge from the last HW assignment.

Next, you will modify the file called: **lc4.c** It serves as the "main" for the entire program. The head of the linked list must be stored in main(), you will see in the provided lc4.c file a pointer named: **memory** will do just that. Main() will then extract the name of the .OBJ file the user has passed in when they ran your program from the argv[] parameter passed in from the user. Upon parsing that, it will call lc4_loader.c's open_file() and hold a pointer to the open file. It will then ask call lc4_loader.c's parse_file() to interpret the .OBJ file the user wishes to have your program process. Lastly it will reverse assemble the file, print the linked list, and finally delete it when the program ends. These functions are described in greater detail below. The order of the function calls and their purpose is shown in commends in the lc4.c file that you will implement as part of this assignment.

Once you have properly implemented lc4.c and have it accept input from the command line, a user should be able to run your program as follows:

```
./lc4 my_file.obj
```

Where "**my_file.obj**" can be replaced with any file name the user desires as long as it is a valid .OBJ file that was created by PennSim. If no file is passed in, your program should generate an error telling the user what went wrong, like this:

```
error1: usage: ./lc4 <object_file.obj>
```

There is no need to check that the filename ends in .obj nor should you append .obj to filename passed in without an extension.

## Problem 1) Implementing the LC4 Loader

Most of the work of your program will take place in the file: called: **lc4_loader.c**. In this file, you will start by implementing the function: **open_file()** to take in the name of the file the user of your program has specified on the command line (see lc4_loader.h for the definition of open_file()). If the file exists, the function should return a handle to that open file, otherwise a NULL should be returned.

Also in **lc4_loader.c**, you will implement a second function: **parse_file()** that will read in and parse the contents of the open .OBJ file as well as populate the linked_list as it reads the .OBJ file. The format of the .OBJ input file has been in lecture, but its layout has been reprinted above (see section*: INPUT_FILE_FORMAT*). As shown in the flowchart above, have the function read in the 3-word header from the file. You'll notice that all of the LC4 .OBJ file headers consist of 3 fields: header type, <address>, <n>. As you read in the first header in the file, store the <u>address</u> <u>field</u> and the <u><n> field</u> into local variables. Then determine the type of header you have read in: CODE/DATA/SYMBOL.

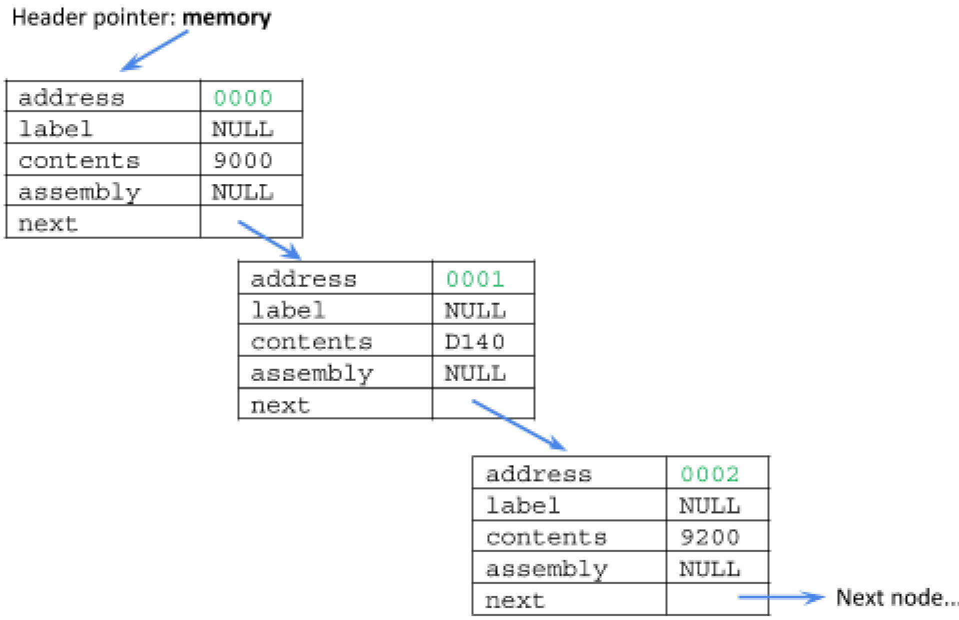
If you have read in a CODE header in the .OBJ file, from the file format for a .OBJ file, you'll recall the body of the CODE section is <n>-words long. As an example, see the hex listed below, this is a sample CODE section, notice the field we should correlate with **n**=0x**000C**, or decimal: 12. This indicates that the next 12-words in the .OBJ file are in fact 12 LC-4 instructions. Recall each instruction in LC4 is 1 word long.

```
CA DE 00 00 00 0C 90 00 D1 40 92 00 94 0A 25 00 0C 0C 66 00 48
01 72 00 10 21 14 BF 0F F8
```

From the example above, we see that the first LC-4 instruction in the 12-word body is: 9000. (that happens to be a CONST assembly instruction if you convert to binary). Allocate memory for a new node in your linked list to correspond to the first instruction (the section above: <u>**LINKED LIST NODE STRUCTURE,**</u> declares a structure that will serve as a blue-print for all your linked list nodes called: "row_of_memory"). As it is the first instruction in the body, and the address has been listed as 0000, you would populate the row_of_memory structure as follows.

| | |
|---|---|
| address | 0000 |
| label | NULL |
| contents | 9000 |
| assembly | NULL |
| next | NULL |

In a loop, read in the remaining instructions from the .OBJ file; allocate memory for a corresponding **row_of_memory** node for each instruction. As you create each **row_of_memory** add these nodes to your linked list (you should use the functions you've created in lc4_memory.c to help you with this). For the first 3 instructions listed in the sample above, your linked list would look like this:




The procedure for reading in the DATA sections would be identical to reading in the CODE sections. These would become part of the same linked list, as we remember PROGRAM and DATA are all in one "memory" on the LC-4, they just have different addresses.

For the following SYMBOL header/body:

`C3 B7 00 00 00 04 49 4E 49 54`

The address field is: 0x0000. The symbol field itself is: 0x**0004** bytes long. The next 4 bytes: `49 4E 49 54` are ASCII for: `INIT`. This means that the label for address: 0000 is INIT. Your program must search the linked list: memory, find the appropriate address that this label is referring to and populate the "label" field for the node. Note: the field: <**n**> tells us exactly how much memory to malloc() to hold the string, however you must add a byte to hold the NULL. 5 bytes in the case of: `INIT`. For the example above, the node: 0000 in your linked list, would be updated as follows:

| | |
|---|---|
| address | 0000 |
| label | **INIT** |
| contents | 9000 |
| assembly | NULL |
| next | |

Once you have read the entire file; created and added the corresponding nodes to your linked list, close the file and return to main(). If you encounter an error in closing the file, before exiting, print an error, but also free() all the memory associated with the linked list prior to exiting the program.

## Problem 2) Implementing the Reverse Assembler

In a new file: **lc4_disassembler.c**: write a third function (reverse_assemble) that will take as input the populated "memory" linked list (that parse_file() populated) – it will now contain the .OBJ's contents. reverse_assemble() must translate the hex representation of some of the instructions in the LC4 memory's linked list into their assembly equivalent. You will need to reference the LC4's ISA to author this function. To simplify this problem a little, you **DO NOT** need to translate every single HEX instruction into its assembly equivalent. Only translate instructions with the OPCODE: 0001 (ADD REG, MUL, SUB, DIV, ADD IMM)

As shown in the flowchart, this function will call your linked list's "search_opcode()" helper function. Your search_opcode() function should take as input an OPCODE (in the least signficant 4 bits - between 0 and 15) and return the first node in the linked list that matches the OPCODE passed in, but also has a NULL assembly field. When/if a node in your linked list is returned, you'll need to examine the "contents" field of the node and translate the instruction into its assembly equivalent. Once you have translated the contents filed into its ASCII Assembly equivalent, allocate memory for and store this as string in the "assembly' field of the node. Repeat this process until all the nodes in the linked list with an OPCODE=0001 have their assembly fields properly translated.

As an example, the figure below shows a node on your list that has been "found" and returned when the search_opcode() function was called. From the contents field, we can see that the HEX code: 128B is 0001 001 010 001 011 in binary. From the ISA, we realize the sub-opcode reveals that this is actually a MULTIPLY instruction. We can then generate the string **MUL R1, R2, R3** and store it back in the node in the assembly field. For this work, I strongly encourage you to investigate the **switch()** statement in C (any good book on C will help you understand how this works and why it is more practical than multiple if/else/else/else statements). *I also remind you that you must allocate memory strings before calling strcpy()!*

**NODE BEFORE UPDATE**

| | |
|---|---|
| address | 0009 |
| label | NULL |
| contents | 128B |
| assembly | NULL |
| next | |

**NODE AFTER UPDATE**

| | |
|---|---|
| address | 0009 |
| label | NULL |
| contents | 128B |
| assembly | **MUL R1, R2, R3** |
| next | |

## Problem 3) Putting it all together

As you may have realized main() should do only 3 things: 1) create and hold the pointer to your memory linked list. 2) Call the parsing function in lc4_loader.c. 3) Call the disassembling function in lc4_dissassembler.c. One last thing to do in main() is to call a function to print the contents of your linked list to the screen. Call the print_list() function In lc4_memory.c; you will need to implement the printing helper function to display the contents of your lc4's memory list like this:

```
<label>          <address>        <contents>       <assembly>
INIT             0000             9000
                 0001             D140
                 0002             9200
                 …
                 0009             128B             MUL R1, R2, R3
(and so on…)
```

Several things to note: There can be multiple CODE/DATA/SYMBOL sections in one .OBJ file. If there is more than one CODE section in a file, there is no guarantee that they are in order in terms of the address. In the file shown above, the CODE section starting at address 0000, came before the CODE section starting at address: 0010; there is no guarantee that this will always happen, your code must be able to handle that variation. Also, SYMBOL sections can come before CODE sections! What all of this means is that before one creates/allocates memory for a new node in the memory list, one should "search" the list to make certain it does not already exist. If it exists, update it, if not, create it and add it to the list!

Prior to exiting your program, you must properly "free" any memory that you allocated. We will be using a memory checking program known as valgrind to ensure your code properly releases all memory allocated on the heap! Simply run your program: lc4 as follows:

```
valgrind –leak-check=full lc4
```

**Valgrind should report 0 errors AND there should be no memory leaks prior to submission.**

***Note: we will run Valgrind on your submission, if it leaks memory, you will lose many points on this assignment. So watch the VIDEO, learn how to use Valgrind!!***

**Also note: If your code doesn't compile or even run, you will lose most of the points of this assignment!**

## TESTING YOUR CODE

When writing such a large program, it is a good strategy to “unit test.” This means, as you create a small bit of working code, compile it and create a simple test for it. As an example, once you create your very first function: add_to_list(), write a simple “main()” and test it out. Call it, print out your “test” list, see if this function even works. Run Valgrind on the code, see if it leaks memory. Once you are certain it works, and doesn’t leak memory, go on to the next function: “search_address()”; implement that, test it out.

DO NOT write the entire program, compile it, and then start testing it. You will never resolve all of your errors this way. You need to unit test your program as you go along or it will be impossible to debug.

***Where to get input files?***

In the last assignment, you created a .OBJ file. Try loading that file into Codio, and use your program on it. You know exactly how that program should disassemble. To test further, bring up PennSim, write a simple program in it, output a .OBJ from PennSim, then read into your program and see if you can disassemble it. You can create a bunch of test cases very easily with PennSim.

You should test your lc4 program on a variety of .OBJ file, not just simple examples. A selection of .OBJ files has been provided in the “obj files for student testing” folder in codio.

**STRUCTURING YOUR CODE:**

**Preloaded in Codio, you'll find some of the files named below. For the ones you don't see on Codio, you must create them and implement them as described in the assignment above.**

| | |
|---|---|
| `lc4.c` | - must contain your main() function. |
| `lc4_memory.c` | - must contain your linked list helper functions. |
| `lc4_memory.h` | - must contain the declaration of your <u>row_of_memory</u> structure & linked list helper functions |
| `lc4_loader.h` | - contains your loader function declarations. |
| `lc4_loader.c` | - must contain your .OBJ parsing function. |
| `lc4_disasembler.h` | - contains your disassembler function declarations. |
| `lc4_disasembler.c` | - must contain your disassembling function. |
| `Makefile` | - must contain the targets:<br>lc4_memory.o<br>lc4_loader.o<br>lc4_disassembler.o<br>lc4<br>All, clean and clobber |

***You <u>cannot</u> alter any of the existing functions in the .h files.***

## EXTRA CREDIT: A complete reverse assembler:

Finish the disassembler to translate any/all instructions in the ISA. Have you program print the linked list to the screen still, but also create a new output file: <users_input>.asm. In that file it should contain only the assembly program that you disassembled. If it works, I should be able to load it into PennSim , assemble it, and reproduce the identical .OBJ file that your .ASM file was derived from! Don't forget to add in the directives (.CODE, .DATA)…the ultimate test of your program will be getting it to assemble using PennSim!

If you do implement the extra credit, please make sure NOT to output assembly directives for opcodes other than 1 in print_list().

## Important Note on Plagiarism:

- We will scan your HW files for plagiarism using an automatic plagiarism detection tool.
- If you are unaware of the plagiarism policy, make certain to check the syllabus to see the possible repercussions of submitting plagiarized work (or letting someone submit yours).