Setting up Codio for this HW:

1) Open the Codio assignment via Canvas
2) From the Codio **File-Tree** click on: **lc4_memory.h** and **lc4_memory.c**
3) **Remember to use the Codio "Pre-submission tests" function before submitting the assignment.**

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:

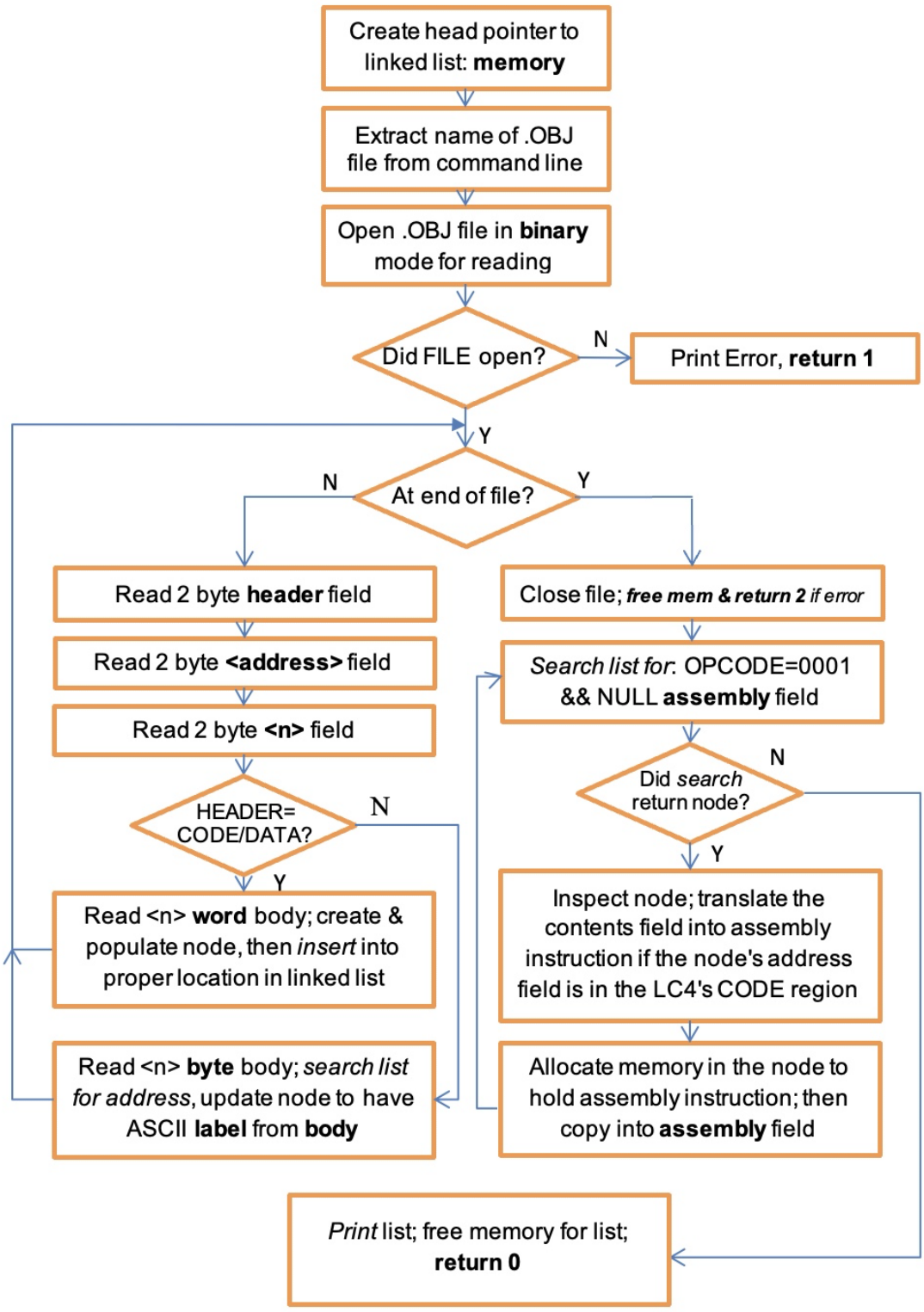
```
struct row_of_memory {
      short unsigned int address ;
      char * label ;
      short unsigned int contents ;
      char * assembly ;
      struct row_of_memory *next ;
} ;
```

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 and piece of data 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 following sections in this document.

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. You must implement ***everything*** listed by the comments in the starter code. If you wish to implement additional helper functions, feel free to add them to any .c file but remember to add the function prototypes to lc4_memory.h.

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 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 comments 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. The filename will be the only argument passed in.

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*: OBJECT_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 **`address`** field and the **`<n>`** field 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** = `0x000C`, 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.

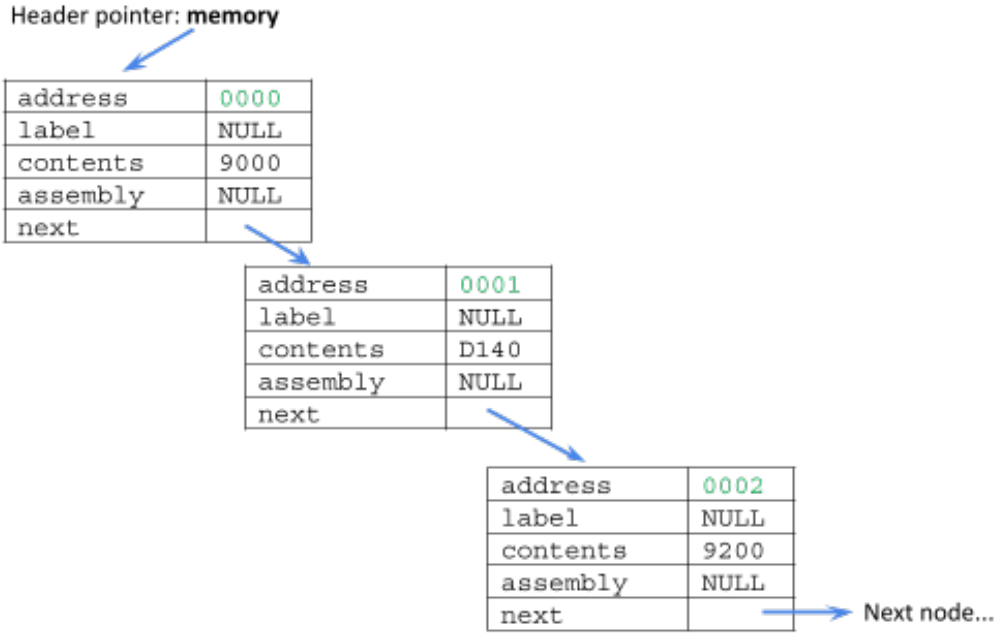
```
CA DE 00 00 00 0C 90 00 D1 40 92 00 94 0A 25 00 0C 0C 66 00 48
01 72 00 10 21 14 BF 0F F8
```

From the example above, we see that the first LC-4 instruction in the 12-word body is: **`90 00`**. (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: **LINKED LIST NODE STRUCTURE**, declares a structure called "`row_of_memory`", that will serve as a blue-print for all your linked list nodes). 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, **ordering the list by address** (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: 0x0004 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	

It might be that you come across a label for an address that has not yet been created in your linked list. In this case create a new node and add the label. The other node fields can be left blank. They will be eventually updated. It is possible that an address has two labels in the **.obj** file. In this case make sure to use the last one that occurs.

Once you have read the entire file; created and added the corresponding nodes to your linked list by address order, 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.

The `feof()` function may not work the way you think it should. The eof indicator only returns true AFTER you have attempted to read past the end of the file. If the file position is right at the end of the file but you haven't tried to read past the end of the file yet, `feof`() will be false. This means that you need to check all of your calls to **`fread()`** or **`fgetc()`** to make sure they didn't hit the end of the file.

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 (which is populated by the `parse_file()`) – this linked list now contains 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 refer to the ***LC4's ISA Instruction*** document 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`.
  - The immediate value can be formatted with `#`, `x`, `X`, and/or nothing
  - e.g. `ADD R1, R1, #10` == `ADD R1, R1, xF` == `ADD R1, R1, 10`

Data stored at an address in the DATA section should not be translated to an assembly instruction.

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 a 4-bit opcode and return the first node in the linked list that matches the opcode passed in, but also has a `NULL` assembly field. The opcode that is passed into `search_opcode()` is a number between 0 and 15 (`0x0` and `0xf`). The opcode in the instruction is in the leftmost bits of the instruction. For example, here's the definition of an **ADD** instruction from the ISA:

0001 ddds ss00 0ttt

but the opcode in the opcode parameter passed to `search_opcode()` will be:

0000 0000 0000 **0001**

when looking for an instruction with opcode = 000**1**. So you will need to use the **C-bitwise operators** to line these two values up before comparing them.

`search_opcode()` is meant to find the first instruction in the linked list where these two 4-bit fields match. There is an additional constraint that memory rows that already have the assembly instruction (reverse assembled) filled in don't count as a match.

Said another way, find the first instruction in memory with a matching opcode that hasn't already been reverse assembled.

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	

You will find that using C bitwise operators will greatly simplify your implementation. See https://www.tutorialspoint.com/cprogramming/c_bitwise_operators.htm for details.

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
END             000A            0
(and so on…)
```

Notice that **`<address>`** and **`<contents>`** fields are displayed in hexadecimal with leading zeroes. Please remove all debugging print statements before submitting. Only the memory list should be printed. If the contents of a node are 0, they should display as 0.

1. You can print (null) for label and assembly instructions when none exist, or leave them blank.
2. There may be very long labels, so that the other entries in that row are pushed to the right. That is fine.
3. You may find the formatting strings "**`%-20.04x`**" and "**`%-20s`**" useful in keeping your output aligned. This isn't critical but it is definitely easier to read.

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!

Checking Memory Leaks with `valgrind`:

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 test1.obj
```

where `test1.obj` is the name of the **`.obj`** file you want to test with.

The following section is in **bold and red** for a reason. Please make sure you fully understand it!

- **Valgrind should report 0 errors AND there should be no memory leaks and no accessing uninitialized memory locations prior to submission.**
- **Note: we will run Valgrind on your submission, if it returns errors, you will lose points on this assignment. So watch the VIDEO "TA Demo: Debugging in C (Valgrind)", 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 !**
- **Valgrind will find errors related to accessing uninitialized memory locations. This typically results from assuming that `malloc()` zeroes out the memory it returns. It does NOT. It can also happen when you access memory locations in an array or string beyond the end of the array or string. While your program may run correctly on your test cases when you have these errors, the results are unpredictable and may cause the test cases run by the autograder to fail.**

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 or accesses uninitialized memory locations. 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.

Remember to use the Codio "Pre-submission tests" function before submitting the assignment.

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.

Using `gdb` to debug your program is also HIGHLY recommended.

STRUCTURING YOUR CODE:

Preloaded in Codio, you'll find the files named below that you must implement.

`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 **`row_of_memory`** 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: **`lc4_memory.o`** **`lc4_loader.o`** **`lc4_disassembler.o`** **`lc4`** **`all, clean and clobber`**

You cannot 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 your program print the linked list to the screen still, but also create a new output file: ***<users_input>.asm*** .

If it works, we should be able **to assemble it in PennSim, load the `.obj` file into PennSim and have the memory in PennSim look the same as when we load the test case `.obj` file**.

- Don't forget to add in the directives (`.CODE`, `.DATA`)…
- the ultimate test of your program will be getting it to assemble using PennSim and produce the same results as the provided **`.obj`** file.

If you do implement the extra credit:

- In **`print_list()`,** please make sure **NOT to output assembly directives for opcodes other than opcode = 0001.** Instead, **leave the assembly directive column blank or (null) for the other opcodes.** Said another way, the `print_list()` output from your program should look like the example in problem 3 :

```
<label>          <address>          <contents>          <assembly>
                 ...
                 0002               9200
                 ...
```

Not as this :

```
<label>          <address>          <contents>          <assembly>
                 ...
                 0002               9200                CONST R2, 0
                 ...
```

- **Nevertheless, the assembly directives for the extra credit opcodes should still appear in the `.asm` file you generate.**

Notes:

- Don't forget to add directives, labels etc.
- **Partial Credit** is possible as we will run the program against multiple **`.obj`** files.
  - No partial credit is available for just reverse assembling the instructions and NOT producing a **`.asm`** file. **You MUST produce the .asm file to get any credit.**
- The different Registers in assembly instruction should be separate by comma
  - e.g. `ADD R1, R2, R3`
- When creating the **`.asm`** file, you should change the last 3 characters of the filename (**`argv[1]`**) from **`.obj`** to **`.asm`**.
- The EC is worth **15 points (out of 100)** so the highest grade on the assignment is **115%**

Extra Credit Testing Tips

- You should be able to assemble your **`.asm`** file with the PennSim '`as`' command and then load it back to PennSim with the PennSim '`ld`'command. The PennSim memory should be the same as when loading the testing **`.obj`** file.
- Note that your **`.obj`** file does not need to be identical to the provided testing **`.obj`** file. The

blocks can appear in a different order. As long as they both produce the same PennSim memory contents you will receive credit.

- Remember the '`as`' command will overwrite the provided testing **`.obj`** file unless you give it a different name. You can always copy the original **`.obj`** file back to the submit forlder if you overwrite it by mistake. This is the main reason we put the test cases in a separate folder.

Key Requirements:

- Your code must compile and run, `Valgrind` should report 0 errors and there should be no memory leaks prior to submission.
- `lc4.c` and `lc4_memory.c`
    - Function requirements explicitly laid out in starter code – implement ***everything*** specified by the comments as these are instructions as well. A copy of the starter functions is provided below in case of deletion
- `lc4_loader.c`
    - **`open_file()`**: read in a string ***`file_name`***; if the file exists, open it and return a pointer to the open file, otherwise return NULL
    - **`parse_file()`**:
        - take in an open file ***`src_file`*** and your linked list *memory*
        - in a loop, read in and parse the `<directive>`, `<address>`, and `<n>` to determine the type of header to appropriately allocate room for and populate a new row_of_memory, and add that node to your linked list using the functions in `lc4_memory.c`, ordering the memory by address
        - once entire file is read, and you've created and added the corresponding nodes to your linked list in address order, close the file and return to `main()`
- `lc4_disassembler.c`
    - **`reverse_assemble()`**:
        - take in your populated linked list memory
        - Find every `row_of_memory` in your linked list with the OPCODE = `0001` (ADD, MUL, SUB, DIV, ADD IMM), utilizing lc4_memory's `search_opcode()` function
        - For these nodes, translate the HEX instruction (*contents*) into its assembly equivalent and update the *assembly* field of the node
        - Check the node's address to make sure it belongs to the LC4's CODE region

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).

Hints from earlier semesters:

- Checking end of file. Have a look at: https://faq.cprogramming.com/cgi-bin/smartfaq.cgi?id=1043284351&answer=1046476070
- You can assume that the maximum length of an assembly instruction is 100, the maximum length of a label is 70 and the maximum length of a filename is 100.
- Helper Functions are fine. Do not change the headers of the predefined functions and remember to "**`#include <stdio.h>`** " in your **`.h`** files if your helper functions require definitions from **`stdio.h`**
- Do not use externs or global variables.
- The "`hexdump -C`" command displays an **`.obj`** file one byte at a time rather than one word at a time. The first column displayed by hexdump is the byte offset.
- You do not have to check that the addresses in the **`.obj`** file are valid
- The order of the steps in the flow chart provide guidance to help you but you can opt for a different order, as long as your program works.
- There are several possible errors: the input file can't be found, the input file isn't validly formatted, malloc can't find sufficient memory, etc. It is a best practice to print an error message and exit if any of those things happen but the autograder will not be testing those sort of edge cases.
- While we want your program to have no memory leaks, it is more important that your program actually runs.
- Make sure all of your loops that traverse the memory list look at the first and last element of the memory list. This is a VERY common mistake.
- Q: Do we need to be able to handle **`.obj`** that is both big and small endianness or will it be standardized?
  A: You should handle the **`.obj`** files that PennSim produces
- valgrind documentation available at https://www.valgrind.org/docs/manual/quick-start.html#quick-start.interpret
- **`add_to_list()`** should keep the memory list in sorted order by address. CODE vs. DATA isn't relevant.
- Q: Will memory locations in input files only be in user code and data space, or could they be in OS memory?
  A: They could be in OS memory as you will see in some of the sample **`.obj`** files provided.
- Q: i am wondering why is the int **`add_to_list()`** method has a double-pointer of "**`row_of_memory** head`**"? Can I delete one of the stars?
  A: No, the double pointer is required.
  - It is because when we add nodes to our linked-list, we want to be able to change the head pointer. Because we pass in a double pointer, we can change what memory (in **`parse_file()`**) points to, thus allowing us to change the head pointer and add nodes to the beginning of our linked-list. Otherwise, we would just be creating a local copy of the new linked-list non-accessible by **`main()`**.
- you are allowed to use the standard library of **`string.h`** .
- in `lc4_memory.c`, the function **`add_to_list()`**:

`/* check to see if there is already an entry for this address and update the contents.  no additional steps required in this case */`

- This step is asking you to look through memory and see if the address is already there. If the address is already present in the memory list, replace the "contents" with the new "contents". The alternative, which doesn't make sense, would be to create a new entry in the memory list with the same address.

- We will make sure that only files with **`.obj`** extensions are passed as arguments to **`./lc4`**
- The **x86** (the processor used by Codio) has a different **endianness** than the LC4. When doing **`fread()`**'s of 2 byte words, swapping occurs to adjust for this. That same swapping doesn't occur with the **`fgetc()`** or **`fread()`**'s with size 1.

`Segmentation Faults` **demystified:**

- `Segmentation faults` are VERY common failures in C programs. They can be hard to pin down.

First we should understand why they happen:

- Segmentation faults are a common class of error in programs written in languages like C that provide low-level memory access. They arise primarily due to errors in use of pointers for memory addressing, particularly illegal access. There are two common cases:
  - Case 1: trying to dereference a NULL pointer. This often happens when calling a function that returns a pointer or NULL in case of error. If you don't check the return value from the function and then proceed to dereference the pointer, you will get a segmentation fault.
  - Case 2: an incorrect assumption that memory on the stack (and heap) is initialized to zero. This is NOT the case. If you want the memory to be initialized to zero, you need to do this explicitly, possibly using the memset() function. You might have code that checks to see if the memory is 0 or NULL and then take some action based on this. If the memory is not initialized to zero, it will have random, unpredictable contents. This is often referred to as (X) or "don't care" in the lectures.
- So, how do you figure out where the segmentation fault is occurring? The simplest way to find out is using `GDB`. After compiling using the "`-g`" clang switch and running the program, remembering the procedure for running a program in gdb that takes a command line arguments (the gdb commands "`start`" and "`run`" allow you to specify the command line arguments), you will get the segmentation fault. You can then use the gdb "`where`" command which will tell you the line number in your program where the failure occurs.
  - For example:

```
gdb ./lc4
start test1.obj
```

  - runs gdb on your `lc4` program and starts it with `argv[1]` set to `test1.obj`