CIS 2400 Final Project: C – LC4 Simulator

Due Date: Wednesday 8/2 @11:59pm via gradescope

This is the most challenging assignment in our course and it is also weighted the heaviest, so please make certain to allocate the proper amount of time to complete this assignment on time.

Which Lectures Should I Watch to Complete this Assignment?: ***Modules 10-12***

What textbook sections should I read to complete this assignment? ***Chapters 11, 12, 16***
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 codio

- **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, try this video: Files->Resources->Tutorials-> ***Video_Tutorial-Makefiles***

Setting up Codio for this HW:

1) Login into your codio environment
2) Click on the "ready to go" link next to: "PROJECT"
3) From the codio "File-Tree" click on: **trace.c**

NOTE: If your program is crashing, before you can be helped via TA office hours, you must run **GDB** on your program. A small amount of detective work with GDB will save hours! Be prepared for TAs to tell you to run GDB before they will help you in office hours. *Watch the tutorial and run in on your program if it is crashing!*

CIS 2400 Final Project: C – LC4 Simulator

Assignment Overview (Parts 1 and 2)

This assignment is one of the longest and most challenging projects in this course. For that reason, it is the most heavily-weighted assignment assigned in the course. We cannot stress enough to start thinking about this assignment early since it will take a lot of time. As with previous assignments, your program *must* run on the virtual machine. You - and *only* you - are responsible for ensuring that it does.

As a broad overview of the assignment, you will write a simplified, command-line version of the assembly-level simulator that is part of PennSim (Don't worry, you are *not* being asked to reproduce the GUI). This essentially means you will be loading and processing machine code files (binary files produced by the LC4 assembler) and executing them just as PennSim would by keeping track of internal state (PC, PSR, registers, control signals, etc.). As an output, you will generate trace text files (format specified later) that contain information from each LC4 "cycle" as your program executes the loaded machine code. Note: you do not need to use dynamic memory (malloc() or free()) for this assignment!

This homework is broken up into two distinct parts. Part 1 is about building an OBJ file parser and loading its contents into an array representing the LC4's memory. Part 2 uses the contents of the LC4's memory (the populated memory array from part 1) and executes the program that was stored in the OBJ file. Part 2 is the most difficult part of the assignment.

Your program will be called **"trace"** and be invoked from the command line as follows:

```
./trace output_filename.txt first.obj second.obj third.obj …
```


The first argument is the name of the file your program will produce as output. The rest are the names of one or more LC4 *object* files that have been assembled to be loaded into your simulator.

In summary, the basic operation of your ***trace*** program (a.k.a. – the LC4 simulator) will be:

1. Read each object file specified on the command line & load the simulated LC4 memory
2. Initialize the state of the simulator just like PennSim does after a reset operation
3. Simulate the action of the LC4 processor one instruction at a time. For each cycle, add a line to the trace file (in the format as specified later).

Important Note on Testing & Grading:

Make certain to test your trace program with various .OBJ files that you create using PennSim well before submitting your milestone.

We may be using an auto-grader to grade submissions. We will attempt to award partial credit for each function implemented, but it is imperative that you stick to the filenames given for this reason and do not modify the given header files. **If your code does not compile**, no partial credit will be given.

CIS 2400 Final Project: C – LC4 Simulator

Part I – Reading in, parsing .OBJ files

Starter files (on codio): trace.c, LC4.h, loader.h, Makefile
Update for this milestone: trace.c, loader.c, Makefile

**DO NOT modify LC4.h and loader.h (you can't modify the header files in any way)!

Grading Note for Part I:

Part 1 is worth 15% the assignment, so complete it quickly to give you maximum time for Part 2.

Summary of Part I:
For this part of the assignment you must first read in and parse the contents of a PennSim-generated object file (a .OBJ file). As you may recall, a .OBJ file is an "assembled" version of a corresponding .ASM file. It is intended to be a listing of what must be loaded into memory (program and data memory). While a .ASM file is a text (or ASCII file), a .OBJ file is a binary file (meaning – non-ASCII). Remember, you can assemble any assembly file in PennSim with the "as" command (ie, "as output input" will produce "output.obj" from "input.asm").

As you read in and parse the contents of the .OBJ file, you will mimic PennSim's memory with a C-data structure. Examine the contents of the file: LC4.h and you will see a data structure entitled: *MachineState*. Notice it contains a field that is another structure: *memory*. That "memory" structure will hold the contents of the .OBJ file that you are parsing. The .OBJ file will specify where memory to place each instruction (or piece of data) you are reading in from the .OBJ file.

For this part of the assignment, once you've populated your simulator's memory with the contents of the .OBJ files, you will then "print out" the contents of your simulator's memory to an output file. You will write code that does this in trace.c.

Code for loader.c:

To do this, you will be creating a file called loader.c and implementing this function declared in loader.h:

```
// read an object file, modify machine state as described in writeup
int ReadObjectFile(char* filename, MachineState* CPU);
```

This function should take the name of an object file as input, read the contents of that input file which should be in the format listed below, and use those values to initialize the memory of the machine. A pointer to the machine state is passed in as an argument.

CIS 2400 Final Project: C – LC4 Simulator

Code for trace.c (Part 1):
In trace.c, for the milestone, you will load your machine memory with the contents of the argument object files by calling ReadObjectFile(). You should first ensure that the filenames are valid (meaning the files exist) before loading memory. If any of the filenames are invalid, do NOT load anything into memory, print out an error message, and terminate the program.

LC4 Object File Format:
To parse a .OBJ file that has been generated by PennSim, you'll need to understand how it is organized. The .OBJ file is a **binary** file - this means you cannot open it with a normal text editor, like sublime. This is because text editors try to interpret files as ASCII, so they will display seemingly random characters. If you do open the file in a text editor (Sublime, in particular), it is possible that the binary file will be corrupted or overwritten, so we strongly recommend not doing this. Instead, use the 'hexdump' command in the terminal to view the raw contents of the binary file.

Internally, a binary file is section-based, and there are 5 kinds of sections: code, data, symbol, file name, and line number. The sections can be interleaved (you cannot count on them to occur in any particular order) and each can appear more than once in a given object file. Although you need to recognize and parse all sections correctly, only the code and data sections carry information used to populate LC4 memory. The formats of these sections are as given below. You may assume .OBJ files generated by PennSim will match the above format and will not have any errors within them.

Code: 3-word header **(0xCADE, <address>, <n>**, n-word body comprising the instructions
Data: 3-word header **(0xDADA, <address>, <n>)**, n-word body comprising the initial data values
Symbol: 3-word header **(0xC3B7, <address>, <n>)**, n-character body of the symbol string
File name: 2-word header **(0xF17E, <n>)**, n-character body comprising the file name string
Line number: 4-word header (0x**715E**, <address>, <line>, <file-index>), no body. File-index is the index of the file in the list of file name sections; so, if your code comes from two C files, your line number directives should be attached to file numbers 0 or 1

Note that for symbol and file name sections, each character is 1 byte (not 2), there is no null terminator, and each symbol or file name is its own section.

Here is an example of what a binary file might look like (recall generating these in HW9):

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

Explanation: We see that the first two words are xCADE (code section specifier), meaning that this is a code section. We recall from the code section format that the next two words specify the starting memory address where the instructions should be placed, and the following two words indicate how many instructions are present in this code section. The bolded section above is the complete header. In this case, the address we will begin to load instructions into is 0x0000, and there are 0x000C (or decimal 12) instructions in this section. Recall each instruction in LC4 is 1-word long. 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).

CIS 2400 Final Project: C – LC4 Simulator

Your program must be able to read in at least one, but possibly multiple .OBJ files. It must load all of them into your simulator's memory. You will execute your program on codio as follows:

```
./trace output_filename.txt first.obj second.obj third.obj
```

In this example, you would have to open and parse the contents of 3 .OBJ files: first.obj, second.obj, third.obj and generate output_filename.txt. The specific names of the file will replace: first, second, third. If the *filename* and at least one .OBJ file aren't specified, print out a usage message & return (-1).

Endianness of .OBJ files

Depending on how you read in the .OBJ files you might encounter endianness. What does this mean? First, review the lecture slides…second, the fundamental units of memory and file storage are bytes or char's (8-bit numbers). Many data types (short's, int's) occupy multiple bytes. For instance, you can think of a 2-byte short as byte containing bits 15:8 and another byte containing bits 7:0. So what is this "big endian" deal? Well, "big endian" just says that multi-byte data-types are represented in files and in memory in most -significant - byte to least-significant byte order. So the short value x1234 looks in memory and in a file like x1234. So why does this not go without saying? Because there are some platforms which are "little-endian" and on these platforms, the value x1234 is laid out in memory and in files to look like x3412. And, wouldn't you know it? x86 is "little -endian". So when you fread() a short from an LC4 object file on an x86 host, you have to swap the bytes to get the value you expect. Your code must be able to handle the endianness of the .OBJ files correctly to properly load in the data in the .OBJ file.

Part 1 (Milestone) Output File Format:

Once you have parsed the file and loaded its contents into your simulator's memory data structure, you can move on to part 2. We recommend that you try printing it out, before moving onto part 2. Something like:

```
address: 00000 contents: 0x9000
address: 00001 contents: 0xD140
address: 00002 contents: 0x9200
...
address: 02010 contents: 0x128B
...
```

This printout isn't to be included with the final HW submission, we just find it helpful so you can verify the contents of the file has been read in correctly.

Makefile

Your milestone must compile using the Makefile we have provided (you must fix the Makefile to create the executable: **trace**, when "**make all**" is run). **If a functional *trace* executable is not produced when we run "make all" in your directory, you will lose the all credit for this milestone as there is very little "partial" credit that can be assigned for non-compiling or non-working code. Please do not submit the assignment without a working Makefile - TAs are more than happy to assist you with this in office hours or on Piazza.**

CIS 2400 Final Project: C – LC4 Simulator

Part II – The LC4 Simulator (worth 85% of the assignment)

Starter files (on codio): trace.c, LC4.c, loader.c
Turn in: trace.c (you will further modify this from Part 1), LC4.c, loader.c, Makefile

**DO NOT modify LC4.h and loader.h!

Summary of Part II:

There are 12 functions for you to implement. They are documented in LC4.h. Instead of one massive function simulating the entirety of the LC4 cycle, we have designed this implementation to be modular. Each function has one specific job, and the UpdateMachineState(MachineState* CPU, FILE* output) function should implement the overall cycle by calling the other functions. You should be calling UpdateMachineState() (and only UpdateMachineState()!) in trace.c. Please stick to this modular design, do not modify LC4.h, and implement all the functions as instructed.

Just like in PennSim, the default starting PC is 0x8200. This should be the same for your program. Additionally, your program should exit when the PC value becomes 0x80FF. When the PC reaches this value, please ***don't*** write out address 0x80FF and the value at that address before terminating - this will be clearer after you read the output file format section below. Note that this address is in the trap table so, for your own testing, you can trigger an exit in your assembly code using an appropriate TRAP instruction.

For this assignment you are not expected to simulate the operations of the I/O system, so you do not have to do anything special when the program performs access on the memory addresses that are associated with I/O registers or video memory, just treat these as regular load and store operations and update the memory accordingly.

The final operation of your trace program is outlined below:
1. Read all the object files specified on the command line and use them to load the simulated LC4 memory.
2. Initialize the state of the simulator just the way PennSim does it after a reset operation.
3. Simulate the action of the processor one instruction at a time. On every iteration you must write out a line to the output file (format described on next page).

Breakdown of the header file:

We provide for you some facilities in LC4.h to help you. You should use the *MachineState* struct, which has the PC, PSR, registers, memory, and a few more values (control signals, write-enables, etc.) to assist you in keeping track of the machine state.

The functions provided are meant to help you group many of the instructions by opcode, when possible. Some of the instructions don't share an opcode with any others - for these, you can just decode and execute them inside UpdateMachineState().

You'll want to use WriteOut() to write to the output file. Make sure to call this at an appropriate point in the program, after you've decoded and set the necessary values in your CPU.

CIS 2400 Final Project: C – LC4 Simulator

Output file format:

The output file generated by trace will be an ASCII file. For each instruction that is run during the trace, a new line is added to the trace file. Each line consists of 9 hex values and 1 binary:

1. the current PC
2. the current instruction (*written in binary*)
3. the register file WE
4. if regFileWE is high, which register is being written to
5. if regFileWE is high, what value is being written to the register file
6. the NZP WE
7. if NZP WE is high, what value is being written to the NZP register
8. the data WE
9. if data WE is high, the data memory address
10. if data WE is high, what value is being loaded or stored into memory

A sample output line could be:

8200 1001111000000000 1 7 0002 1 1 0 0000 0000

This would correspond to:

1. a PC = 0x8200
2. the instruction CONST R7 #2
3. the regFileWE as high
4. register 7 being written to
5. the value #2 being written to the register file

6. the NZP WE as high
7. the NZP bits being set to positive
8. the data WE as low
9. the data memory address set to 0000
10. the data value set to 0000

Note that there is a space between each of the values, and there is a newline at the end of a line to separate it from the next.

Also note that if a value isn't relevant for a particular instruction, it should be set to 0. For this CONST instruction above, since dataWE was low, both the data address and data value were 0. The same consideration should be made for values related to regFileWE and NZP WE.

Remember, each line of the output trace should be the state of the machine on that current cycle. This means that it's the PC at the *start* of that current cycle - not after the PC has been modified as a result of the instruction.

Additionally, PennSim has a **trace** feature, which makes testing your own work very convenient. You can turn it on by doing: **trace on <filename>**
This creates a trace file called <filename> when you run the program. You can then turn tracing off by doing **trace off**. With this, you'll be able to test the outputs of your own program against what PennSim produces. This is very helpful to understand the output format that we expect, as well as to verify that your simulator is working properly.

CIS 2400 Final Project: C – LC4 Simulator

Macros:

To decode the instructions, you will want to make use of the C facilities for manipulating bit fields. Operators such as &, |, << and >> can be used to slice and dice 16-bit values as necessary. You may also find it handy to use parameterized macros, an advanced preprocessor feature, to parse LC4 instructions. For instance,

```
#define INSN_OP(I) ((I) >> 12)
#define INSN_11_9(I) (((I) >> 9) & 0x7)
```

will extract bits [15:12] (the opcode) and bits [11:9] (the destination register) from an instruction, respectively. You may want to use similar macros to extract sub-- opcodes, register names, and immediate values. You will probably find it convenient to use a switch statement to handle the various instruction types. A handy tutorial on Macros is located here (scroll down to the Macros section):
https://www.cprogramming.com/tutorial/cpreprocessor.html

Also, a step-by-step tutorial on C-macros and bitwise operators is available on canvas under: Files->Tutorials->Tutorial-Bitwise-Operators.pdf

Advice:
Once again when confronted with a non-trivial programming task like this one you need to break the problem into pieces and start with something basic, get that working and then add another piece. If you try to write the whole thing in one go, you will only succeed in getting yourself horribly confused. This is particularly true with a language like C where bugs in one part of the code can cause mysterious effects in other parts of the system.

A large part of programming is testing and debugging. Whenever you write a piece of code, one of your first thoughts should be - how do I test it to convince myself that it does what it is supposed to do? This mode of development allows you to proceed incrementally, adding a little functionality at each stage, until you have the whole thing working.

In this case I would start out by writing the functions in the LC4.c file. When you are testing this component of your program you will probably find it convenient to print out on every iteration the current instruction being executed, the current control signals and the current state of the LC4 processor – all the register values, the PSR and the PC. You may even want to be able to step through the execution to track where things are going wrong. You would just need to put in a statement to have it wait for you to type something before advancing to the next instruction. Once you can initialize memory and simulate the processor you should have most of the functionality you need to implement the trace program and produce the required output file.

You should plan on testing your trace code by writing some assembly programs and then assembling them into object files using the as command in PennSim. You can then step through the code to verify that the instructions are handled correctly. We will also test your program by providing it with our own object files and comparing the trace you produce to the one produced by a reference implementation. We will provide a subset of those test cases to you for your testing.

CIS 2400 Final Project: C – LC4 Simulator

Error handling:

There are three types of errors that will cause PennSim to complain, and your code must identify them as well. These faults are listed below:

1. Attempting to execute a data section address as code
2. Attempting to read or write a code section address as data
3. Attempting to access an address or instruction in the OS section of memory when the processor is in user mode.

If any of these exceptions occur, you should print out that an error occurred and exit the program. Your trace should still be generated though, up **BUT NOT including** the instruction cycle that caused the exception.

Testing:

While we will provide some test files we will use to grade your assignment, you can use your own .obj files from previous assignments to test your simulator, or simply generate your own using PennSim.

You will need to add the following lines to the bottom of your .ASM programs that don't have an OS:


```
.OS
.CODE
.ADDR x8200
.FALIGN
CONST R7, #0
RTI
```


This is necessary, because your simulator must set the PC=x8200 when it starts, just like PennSim does.

As mentioned previously, you can also run a .asm file through PennSim and generate a trace, then run the file through your simulator to generate a trace, and then compare the two. You can use the **diff** command in the terminal to check where the two traces differ. If your trace doesn't differ at all from PennSim's, then your simulator is doing the right thing for that test program!

CIS 2400 Final Project: C – LC4 Simulator

Directions on how to submit your work:

- Submitting on **gradescope:**
  - Within codio, navigate to the "Project" Menu
    - Select: "Export as ZIP"
    - Codio will zip up all your files for you and allow you to save it on your computer
    - Unzip this file to your local computer
  - Upload the files contained from your codio environment 1 by 1 to **gradescope**
- Run the autograder on gradescope BEFORE you mark your assignment as complete in codio!
- Important Note on Plagiarism:
  - We will scan your HW files for plagiarism using automatic plagiarism detection tools
  - If you are unaware of the plagiarism policy, make certain to check the syllabus