CIS 240 Fall 2014

Homework #8: LC4 Simulator {200 pts}

"You're going to need a bigger boat"

Preamble:

At the risk of stating the obvious, this assignment is one of the longest and most challenging projects in this course. You would be well advised to start thinking about this assignment early since it will take a lot of time.

Note that there are several subtle and annoying differences between C compilers on different machines. For this assignment you are expected to use the clang compiler and eniac's runtime environment. If you choose to use a different system to do development you are solely responsible for making sure that your end result compiles and runs correctly on eniac. The TAs cannot and will not be responsible for getting code to run on the wide variety of platforms and compilers in use today. More specifically the TAs will not be responsible for answering questions of the form, "how do I get <fill in the blank> to run on Windows/Mac/Ubuntu". Because of the differences between compiler implementations and C libraries on different operating systems getting something to compile and run on one system does not necessarily guarantee that it will work on a different machine. You should plan on making absolutely sure that your program will compile and run correctly on eniac. The safest way to do that is to develop on that platform.

The Assignment:

In this assignment you will be writing a simpler version of the assembly level simulator that is part of PennSim. Since you have had lots of experience with PennSim it should be pretty familiar to you by now. Don't worry you are only being asked to create a *command line* version of the simulator, that generates a trace from an input program, you are not expected to reproduce the PennSim GUI.

Your program will be invoked from the command line as follows:

```
>> trace output_filename first.obj second.obj third.obj ….
```

The first argument is the name of the file you should output your results to. The remaining arguments are the names of LC4 object files which you should load into your simulator.

The basic operation of your trace program is outlined below:

1. Read all of 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 the value of the PC and the value of the 16 bit instruction to the binary output file. You also need to print out to the standard output (stdout) a line indicating the instruction that the simulator is currently processing in the following format: 0x8201 : ADD R4, R2, R1. The first number is the address of the instruction in memory in hex, the string after the address is the LC4 mnemonic for the instruction - you are responsible for looking at the current 16 bit instruction and generating an appropriate ASCII string that indicates what the instruction is in the same way that PennSim shows you your instructions in the memory window.
4. **Your program should exit if the PC value becomes 0x80FF**. Note that this address is in the trap table so you can trigger an exit in your assembly code using an appropriate TRAP instruction. This means that in most cases the last entry of your output file will correspond to the instruction at 0x80FF.

Simulating Instructions

We will essentially be testing your program by comparing it’s output to that of PennSim. If you are in doubt about how a particular instruction should work see what PennSim does and then make your code do that. Note for example that the MOD, DIV and MUL instructions in LC4 treat the operands as unsigned values (of course for multiplication the unsigned and 2C variants actually produce the same result, that is NOT true for MOD and DIV). We will test your program by loading object files of our creation and your code should simulate them exactly as PennSim would.

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.

Exceptions

There are three types of errors that will cause PennSim to complain and your code must catch 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.

Your Code.

In this assignment we are testing your ability to perform multi-file development in C so you are required to split your program into **at least** 3 files, LC4.c, ObjectFiles.c and trace.c. We provide you with a file called LC4.h that defines some of the data structures that we require you to use and declares some functions that you are required to implement. More specifically LC4.h declares a type called MachineState which models the state of the registers and memory of your LC4 processor.

Code File LC4.c

LC4.c should contain the code that is used to model and simulate the LC4 processor, it must include definitions of the following functions:

<u>void Reset (MachineState *theMachineState);</u>

Reset the machine state as PennSim would do. You should use this function to initialize the machine state at the start of the simulation.

<u>int UpdateMachineState (MachineState *theMachineState);</u>

This function simulates the action of the LC4 processor over 1 clock cycle by updating the MachineState structure that it is passed. In the course of doing this update your code should check for the exceptions listed in the Exceptions section of this assignment, it should return a 1 if it detects a type 1 exception, 2 for a type 2 exception and 3 for a type 3 exception. If there was no problem the function should just return 0.

Your UpdateMachineState function **<u>must</u>** do it's job by calling the following functions: RS, RT, AluMux, regInputMux and PCMux which are all declared in LC4.h. These functions are supposed to simulate the action of various portions of the datapath. These functions return unsigned short ints that correspond to the 16 bit outputs produced by the corresponding structures in the LC4 schematic. You are expected to use the control signal inputs to correctly determine the output of each structure. When we test your code we may use combinations of control signals that do NOT correspond to LC4 instructions to make sure that you are faithfully simulating the hardware.

Code File ObjectFiles.c:

All of your code for reading object files should be contained in the file ObjectFiles.c, you should define at least a function called:

int ReadObjectFile (char *filename, MachineState *theMachineState);

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. If all is well this function should return a zero, if an error is detected with the file such as an illegal filename or a formatting error the function should return a non-zero value and an informative error message should be printed out.

You should declare the functions that this module provides in the file ObjectFiles.h, a skeleton of this file is provided.

Code File trace.c:

Finally you should have a file called trace.c which contains at minimum the main routine and implements the required behavior of the program.

Makefile

Since your progam will be split into several files you will be required to include a Makefile that we can use to compile your code from the provided source. This Makefile should have at least 4 targets. It should create the object files LC4.o and ObjectFiles.o from the associated source and header files, it should create the main program trace, from trace.c and the aforementioned object files and there should be a fourth target called clean which should remove the executable and object files leaving only the source files and the Makefile.

Note that you are free to add as many additional functions as you need to the files. You can also add more files to the project if you want, you just need to ensure that your Makefile handles them appropriately.

Format of the LC4 Object files

Here is the file format for LC4/PennSim object files. The files are section based, and there are five kinds of sections: code, data, symbol, filename, and linenumber. The sections can be interleaved. Here are the formats of each of the sections:

- **Code**: 3-word header (xCADE, <address>, <n>), n-word body comprising the instructions.

- **Data**: 3-word header (xDADA, <address>, <n>), n-word body comprising the initial data values.

- **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.

• **File name**: 2-word header (xF17E, <n>), n-character body comprising the filename string. Note, each char- acter in the file is 1 byte, not 2. There is no null terminator. Each file name is its own section.

• **Line number**: 4-word header (x715E, <addr>, <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.

Although you need to recognize and parse all sections correctly, only the code and data sections actually carry information that is used to populate LC4 memory.

One word of warning about LC4/PennSim object files. They are "big-endian". What does this mean? Well, 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.

Output file format

Your output file should consist of a series of 16 bit values. These 16 bit numbers are entered in the file in pairs, the first number in the pair is the value of the PC corresponding to the current instruction, the second number is the 16 bit instruction at that location. You can write these values to the file with the fwrite command. You don't have to worry about endianness in this case since the test program that checks your output will also be running on Eniac so it will use the same byte ordering convention.

Hint

In order to decode the LC4 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 fields, and immediates. You will probably find it convenient to use a switch statement to handle the various instruction types.

Solid 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 of 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. Note that at this stage you can write code to initialize the code sections of memory to whatever instruction sequence you like.

Once you have this working to your satisfaction you can turn your attention to reading object files and initializing the LC4 simulator memory. For debugging purposes, you can have your program print out the contents of relevant sections of the memory to verify that the program is loaded correctly.

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.

Extra Credit {20 pts}

Some of the test programs will actually write to the portion of memory corresponding to the bitmapped display. For extra credit when the program exits your trace program should produce an additional file named image.ppm which contains the final image on the display. You can find information on the PPM image file format at the following Wikipedia page.

http://en.wikipedia.org/wiki/Netpbm_format

You should use the 'P6' binary format with 8 bits per color channel. It's the simplest color format. Your program would essentially interrogate the contents of memory starting at address 0xC000 and write the appropriate RGB values into the file to produce the requisite image. You could then view this image on eniac using the gimp tool.

Submission instructions.

Create a directory containing all of your code files. You should name the directory as follows FIRST_LAST_HW_##. For example CHRIS_BROWN_HW4. If you have a long first or last name feel free to shorten it if you can do so without ambiguity. Create a compressed version of your directory. On windows you can sometimes do this by right clicking on the directory and then selecting "Send To -> Compressed Folder". On the Mac you can right click on the directory and select "Compress". Once you have compressed your directory into a single zip file you can submit it on Blackboard as per usual.

Your directory should contain the following items:

1. All of the source code files required to build trace.
2. A Makefile as specified above that can be used to build the trace program
3. A README file containing notes for the TAs if necessary

We will use the Makefile you provide to compile your code. Then we will test the code with our own inputs. If the code doesn't compile or doesn't run you should not expect many points.

Please make sure that you submit the latest versions of your code. A number of people seem to mess this up and submit assignments that are incomplete or incorrectly formatted. At this stage in the course you should know the drill so we are not likely to be very forgiving of silly errors.