CS2110 Fall 2011
LC3 - Simulator

This assignment is due by:

Day: Thursday December 8th

Time: 11:55 pm

Notice:

COMPLETE EITHER THIS ASSIGNMENT OR THE MALLOC ASSIGNMENT OR THE MODE 0 GBA GAME ASSIGNMENT FOR HOMEWORK 13.

YOU ARE NOT REQUIRED TO DO ALL THREE. DO NOT SUBMIT MORE THAN ONE

Warning:

Your submission must compile with our flags or we will simply not grade it and give it a zero. After you submit download your submission again and unzip in a clean directory and build it.

Rules and Regulations

Academic Misconduct

Academic misconduct is taken very seriously in this class. Homework assignments are collaborative. However, each of these assignments should be coded by you and only you. This means you may not copy code from your peers, someone who has already taken this course, or from the Internet. You may work with others **who are enrolled in the course,** but each student should be turning in their own version of the assignment. Be very careful when supplying your work to a classmate that promises just to look at it. If he turns it in as his own you will both be charged. We will be using automated code analysis and comparison tools to enforce these rules. **If you are caught you will receive a zero and will be reported to Dean of Students.**

Submission Guidelines

1. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency let us know ***IN ADVANCE*** of the due time supplying relevant documentation (i.e. note from the dean, doctor's note, etc). Extensions will only be granted to those who contact us in advance of the deadline and no extensions will be made after the due date.

2. You are also responsible for ensuring that what you turned in is what you meant to turn in. No excuses, what you turn in is what we grade. In addition your assignment must be turned in on T-Square. When you submit the assignment you should get an email from T-Square telling you that you submitted the assignment. If you do not get this email that means that you did not complete the submission process correctly. Under no circumstances whatsoever we will accept any email submission of an assignment. Note: if you were granted an extension you will still turn in the assignment over T-Square.

3. There is a random grace period added to all assignments and the TA who posts the assignment determines it. The grace period will last at least one hour and may be up to 6 hours and can end on a 5 minute interval; therefore, you are guaranteed to be able to submit your assignment before 12:55AM and you may have up to 5:55AM. As stated it can end on a 5 minute interval so valid ending times are 1AM, 1:05AM, 1:10AM, etc. **Do not ask us what the grace period is we will not tell you**. So what you should take from this is not to start assignments on the last day and depend on this grace period past 12:55AM. There is also no late penalty for submitting within the grace period. If you can not submit your assignment on T-Square due to the grace period ending then you will receive a zero, no exceptions.
4. Although you may ask TAs for clarification but you are ultimately responsible for what you submit.

Submission Conventions

1. All files you submit for assignments in this course should have your name at the top of the file as a comment for any source code file, and somewhere in the file, near the top, for other files.
2. In addition any code you write must be clearly commented and the comments must be meaningful. You should comment your code in terms of the algorithm you are implementing we all know what the line of code does.
3. When preparing your submission you may either submit the files individually to T-Square (preferred) or you may submit an archive (zip or tar.gz only please) of the files.
4. If you choose to submit an archive please don't zip up a folder with the files, only submit an archive of the files we want.
5. Do not submit compiled files that is .class files for Java code and .o files for C code.

Overview

For this option for homework 13, you will be implementing a simple simulator for the LC-3 processor. It should be able to read in an assembled file (use lc3as to produce these) and should correctly support all instructions except for **RTI**. You must also support a certain set of commands.

You are not building a GUI based lc3 simulator. Only a command line based one.

File I/O in C

The following 6 functions are part of the C library for reading from a file.
All are defined in the file stdio.h:

I suggest you read the man pages on all of these functions to get more information about them.

FILE *fopen(const char *path, const char *mode)

This function opens the file whose name is the string pointed to by path and creates a stream object that you can use with the other functions to read or write to the file.

The mode parameter specifies how the file should be opened:

r Open the file for reading
r+ Open the file for reading and writing
w Truncate or create the file for writing
w+ Truncate or create the file for reading and writing
a Open the file for writing, but start at the end of the file
a+ Open the file for reading and writing, but start at the end of the file

int fclose(FILE *stream)

Close the stream object. Returns 0 if successful, else returns the macro EOF.

int fgetc(FILE *stream)

Read a character from the current file stream, advancing the stream by one byte. If the end of the file has been reached, EOF is returned; otherwise, the next byte is returned as an unsigned char cast to an int.

char *fgets(char *s, int size, FILE *stream)

Read in at most size - 1 bytes from the stream into the buffer pointed to by s. If the end of the file or a newline character has been reached, stop reading. Newline characters are stored in the buffer; the end of file character is not. The character after the last one read is set to be the null ('\0') character.

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)

> Read in nmemb elements of data, each size bytes long, from the stream into the buffer pointed to by ptr. The number of elements read from the file is returned; if the end of the file is reached, this number may be smaller than size.

.obj File Format

An LC-3 assembled object file has a simple file format. The first 16-bit value is the starting address **to place code**. The second 16-bit value is the value of memory at that location; the third value is the value at the subsequent location; this continues on until the end of a file. If you have multiple .orig statements, the assembler simply chooses the lowest one to make the start address and zero-fills the intermediate slots of memory.

One important thing to note is that these files are stored as big-endian data. What this means is that if I want to store the 16-bit value x3000 in the file, on the disk it would look like:

```
 x0000 x0001
+-----+-----+
| x30 | x00 |
+-----+-----+
```

The x86 processor is little-endian, which means that individual bytes are stored in memory in reverse order. So if we try to store an unsigned short with value x3000 using fwrite, we would see the following on disk:

```
 x0000 x0001
+-----+-----+
| x00 | x30 |
+-----+-----+
```

What does this mean for you? This means that your code to read a u16 from the file cannot use fread with an unsigned short. Instead, you must get individual bytes from the file using fgetc and then pack them into an unsigned short.

Console Input

One of the things you have to do in this assignment is read input from the console. Console input is represented by the special file pointer stdin; you can use the same functions for reading from files for console input. In addition, the function getchar() acts as if fgetc(stdin) were called.

Whenever you want the user to enter something, you should have some sort of prompt. For example, when GDB asks for a command, it displays:
(gdb)
Your simulator should display a similar line when asking for input.

To actually read the input, you could use one of two methods: fgets, which reads in up to the newline character or to a maximum buffer size or scanf, which reads in data formatted in a specific manner

You should be prepared to sanitize some user input. Leading and trailing whitespace must be ignored, so typing in "help" and " help " must both execute the "help" command. The `strtok' function may be helpful.

Every line that the user inputs is treated as a command followed by some arguments to the command. Arguments are separated by any non-zero amount of whitespace separates two arguments or an argument and the command. So the lines "step 1" and "step 1" are equivalent. Not all commands may take arguments.

If the user passes in extra arguments, you may ignore them. If the user tries to execute a command that doesn't exist, you should produce an error message which should suggest trying help. If the user types in nothing, you should execute the last command again.

Commands

All of these commands must be supported by your simulator

step [n]
    If no argument, Executes 1 instruction
    Otherwise, executes n instructions
quit
    Quits the simulator.
continue
    Runs until the program halts.
registers
    Dumps the registers of the machine. It should display each register in both hexadecimal and signed decimal. It should also dump the value of the PC (in hex) and the current condition code.
dump start [end]
    Dumps the contents of memory from start to end, inclusive. (start and end are in HEXADECIMAL i.e. 0x0, 0x1000, 0xFFFF)
setaddr addr value
    Sets memory address addr to value. Addr is in hexadecimal and value is in decimal
setreg Rn value
    Sets a register to a value ex setreg R0 10 will set R0 to 10.

The [] syntax means that the argument is optional.

Writing the simulator

For any simulator, you need to have a representation of the current state of the machine. This is comprised of the entire memory map and all of the registers of the machine. The registers include not just the general-purpose registers but also such things as the program counter and the condition codes register.

For the purposes of this assignment, you can ignore the special device registers located at xFFE0 and above and assume that the space there is regular memory.

Below is the binary representations of the instructions you need to implement:

```
     1 1 1 1 1 1
     5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
 BR |0 0 0 0|n z p|    PCoffset9    |
ADD |0 0 0 1| D R | SR1 |0 0 0| SR2 |
ADD |0 0 0 1| D R | SR1 |1|  imm5   |
 LD |0 0 1 0| D R |    PCoffset9    |
 ST |0 0 1 1| S R |    PCoffset9    |
JSR |0 1 0 0|1|      PCoffset11     |
JSRR|0 1 0 0|0 0 0|BaseR|0 0 0 0 0 0|
AND |0 1 0 1| D R | SR1 |0 0 0| SR2 |
AND |0 1 0 1| D R | SR1 |1|  imm5   |
LDR |0 1 1 0| D R |BaseR|  offset6  |
STR |0 1 1 1| S R |BaseR|  offset6  |
NOT |1 0 0 1| D R | SR1 |1 1 1 1 1 1|
LDI |1 0 1 0| D R |    PCoffset9    |
STI |1 0 1 1| S R |    PCoffset9    |
JMP |1 1 0 0|0 0 0|BaseR|0 0 0 0 0 0|
LEA |1 1 1 0| D R |    PCoffset9    |
TRAP|1 1 1 1|0 0 0 0|    Vector8    |
```

Notes:
The ADD, AND, LD, LDI, LDR, LEA, and NOT instructions are the only ones that modify the condition codes register, based on whether or not the value to go into the destination register is negative, zero, or positive.

Opcode 0b1000 is listed as RTI in Appendix A. you will not be expected to implement this. We will not use this in any test cases.

Opcode 0b1101 is an invalid opcode. We will not use this in any test cases.

Program execution intially starts at x3000 meaning when your simulator starts up the pc should be x3000. The condition codes register is initialized to Z see enum in lc3.h, and the values of other registers are indeterminant, i.e., we will not rely them on being filled to any particular value on startup.

Do not handle invalid forms of the instructions, what do I mean about this? Consider the JMP instruction above when bottom 6 bits are not all zero. In this case just execute it as a JMP instruction.

Traps

Whats the LC-3 without traps? For this requirement you will be implementing a pseudotrue (oxymoron?) trap system. What I mean by this is that you will implement GETC, OUT, IN, PUTS, PUTSP, and HALT directly in C hence “fake traps”

For any other trap number you will do what the real implementation of trap does. From PattPatelAppA.pdf (I suggest reading over the pseudocode of all of the instructions again.) A trap instruction is the following:

R7 = PC;†

PC = mem[ZEXT(trapvect8)];

Where PC is the **ALREADY** incremented PC**.**

Hints of how to do the fake traps (pseudocode).

Vector	Name	Pseudocode
x20	GETC	R0 = character read in from stdin
x21	OUT	Print out character in R0
x22	PUTS	R0 contains an address keep printing out characters until you see an address with x0000 in it.
x23	IN	Like GETC but you must print a prompt of your choosing!
x24	PUTSP	Each address contains two characters to print out use bitmasking/shifting to print each out. Do not print out any NUL characters. Stop when you see an address with x0000.
x25	HALT	Stop execution at once. You must also decrement the pc.

You should also reread PattPatelAppA.pdf in the section where all of the traps are explained.

The Code

Down to the nitty-gritty details. I have defined the lc3 structure you are to use. Do not change this or suffer heavy penalties. Likewise do not add global/static variables or any of the like I could be running tests with multiple lc3machines and doing something like this will break having multiple of them. These are the only fields you need. Do not change any of the prototypes by changing the types or adding removing any parameters. You may write helper functions.

```
typedef struct
{
        short mem[65536]; /* Memory */
        short regs[8]; /* The eight registers in the LC-3 */
        unsigned short pc; /* The pc register */
        unsigned char cc; /* The condition code register the value will be one of the enum values above
*/
} lc3machine;
```

Keep in mind this assignment will be autograded, and I will be calling all of these functions directly.

Testing

You should first test EACH instruction to ensure it works. You can verify your output is correct by also running the file in simpl or Complx.

So a good testing strategy is to do this in baby steps.

1. Write one-two line assembly programs for each instruction and feed it into your simulator and check the output afterward.
2. Start writing bigger tests. Hey you know those old lab assignments I had you do? Good test cases.
3. Start testing with homeworks. HW6-7 TL3-TL4 are good candidates. All of these have input and output so your simulator should work for all of these! (of course if your code is not correct you should correct them first!).

I assure you that if you start with 3 you will never find that one bug in that one instruction you've implemented and you will give up saying “its too hard” This is why you start out with the baby steps to catch glaring errors with the instructions. Debugging simulators is tough and if you do the above you will minimize frustration.

Code documentation

YOU MUST COMMENT YOUR CODE for this assignment! If you do not have any comments in your code, then you will lose points for it. We don't expect every line of code to have comments, but we expect that you will have a lot of comments so that we can figure out what you are trying to do in your code.

Submission Requirements / Deliverables

1) You may have multiple C files.

   ****NOTE****

   You will have to modify your Makefile (:O) if you do this.

2) You may NOT core dump (segmentation fault).

3) Your code must compile cleanly with the flags provided.

   gcc -std=c99 -pedantic -Wall -O2 -Werror <files>

   If your code does not compile with these flags, you WILL receive a ZERO.

4) You must turn in all of your files. If you wrote some test code for your simulator, please turn that in as well.

5) Be sure to comment your code thoroughly! You will lose points if it is not

   commented!

6) Do not print any extra output. You will lose points for any spurious output.

7) FILES TO TURN IN

   - Makefile
   - All .c and .h files you use

8) GOOD LUCK