# CS2110 Fall 2013
# Homework 12

Version: 3

**This assignment is due by:**

Day: Friday, December 6th
Time: 11:54:59pm

**Notice:**

**COMPLETE EITHER THIS ASSIGNMENT OR THE MALLOC ASSIGNMENT**

**OR THE MODE 0 GBA GAME ASSIGNMENT FOR HOMEWORK 12.**

**YOU ARE NOT REQUIRED TO DO ALL THREE. DO NOT SUBMIT IN 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/she 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 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 via

   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.

**General Rules**

1. In addition any code you write (if any) 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.
2. Although you may ask TAs for clarification, you are ultimately responsible for what you submit.
3. Please read the assignment in its entirety before asking questions.
4. Please start assignments early, and ask for help early. Do not email us the night the assignment is due with questions.

**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. 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.

3. 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.
4. Do not submit compiled files that is .class files for Java code and .o files for C code.
5. Do not submit links to files. We will not grade assignments submitted this way as it is easy to change the files after the submission period ends.

# 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 as2obj to produce these) and should correctly support
all instructions except for RTI. You must also support a certain set of user input commands.

You are not building a GUI based lc3 simulator. Only a command line based one. Think lc3runner but with a command line front-end on top of it.

You are also not building an lc3 assembler I myself have already taken care of this to you via the program as2obj. You will take files generated from my program and load them into your simulator and then execute the LC3 code from there.

# Command line front end

All of these user input 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 will be in HEXADECIMAL i.e. x0, x1000, xFFFF)

**If end is not given then you should only print out memory[start]**

list start [end]

Disassembles the contents of memory from start to end, inclusive.

(start and end will be in HEXADECIMAL i.e. x0, x1000, xFFFF)

**If end is not given then you should disassemble only memory[start]**

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.

# .obj File Format

### Aside "About Binary files"

A binary file is a file "containing machine-readable information that must be read by an application; characters use all 8 bits of each byte" What this means is that instead of information being presented in a human readable format it is in a machine readable format.

Say I wrote the value 48 to a file.
To do this in a human readable format I would write ASCII character '4' followed by ASCII character '8' to a file. In a binary file I would just write character 48 to a file. The difference is if I opened both files in a text editor I would see 48 in the first, and 0 in the second.

The advantage of binary files is that when I read the data from the file I don't have to do any unnecessary Integer.parseInt(str)/scan.nextInt() (for you Java people) or atoi(str) (in C). The second advantage is that I save a lot of space storing 48 in ASCII takes 2 bytes while writing one char containing 48 is 1 byte.

### The format

An LC-3 assembled object file has a simple binary file format. The first 16-bit value is the starting address to place code. The second 16-bit value is the number of values to put starting at the starting address. Each subsequent value is data to put at the current address.

### Endianness

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:

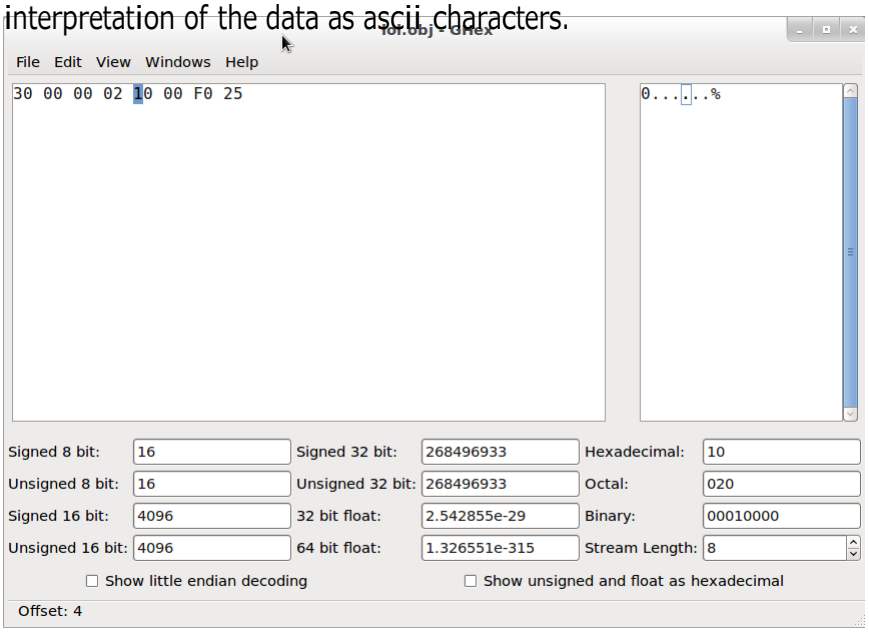
```
x0000 x0001
+++
| x00 | x30 |
+++
```

What does this mean for you? This means that you use fread to read a u16 from the file the bytes will be swapped and you will have to swap them back! Alternatively you may get individual bytes from the file using fgetc and then pack them into an unsigned short.
For more information - http://en.wikipedia.org/wiki/Endianness

**Example**

```
.orig x3000
        ADD R0, R0, R0
        HALT
.end
```

Would be stored on disk as the following (you may use hd on a .obj file or get ghex2 "sudo apt-get install ghex" to view binary files). Note that the characters on the right text box show the interpretation of the data as ascii characters.




The first two bytes are x30, and x00 which means the program is to be loaded starting at x3000. The next two bytes x00 and x02 means there are 2 shorts to be stored starting at x3000. The last two bits of data x1000 and xF025 is the assembled assembly instructions.

**But what if I have multiple .orig statements?**

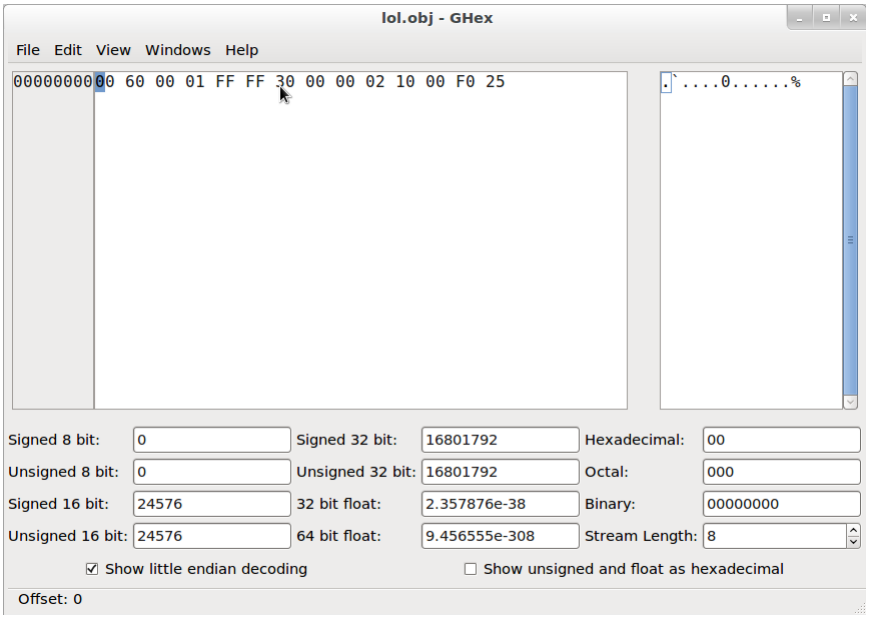
**Example**

```
.orig x60
        .fill -1
.end

.orig x3000
        ADD R0, R0, R0
        HALT
.end
```

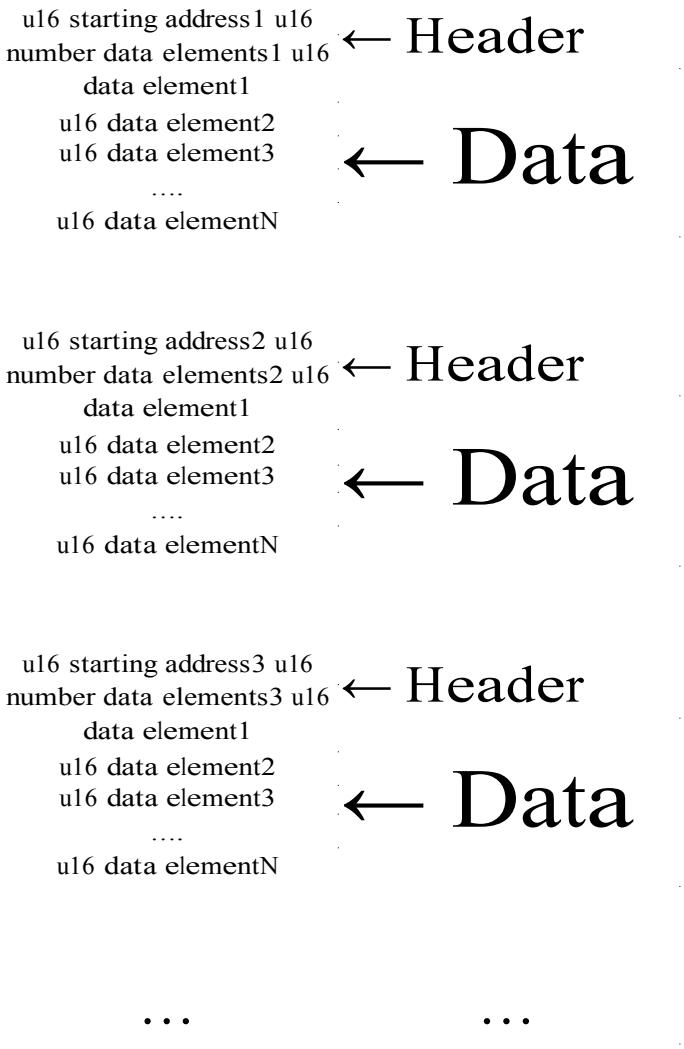
Theres no need to fear! So if you have multiple .orig statements then after you have read all of the data from one .orig statement then the next 16 bits of data will be the new starting address followed by a new number of elements for that .orig statement.

Here is the example output. The mouse pointer is pointing at the starting address for the second piece of data.



**Summary of the as2obj .obj file format**

Below is a more formal explanation of the above.
The starting address and number of data elements form a header. This header tells you where to put the
data and how much data there is. After you have read all of the data if are are not at the EOF (end of file) yet then you should read the next header and process the data. You know you are done when you have reached end of file.




... ...

# 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 xFE00 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 1000 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 1101 is an invalid opcode. We will not use this in any test cases.

# Program Initialization

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 do anything special with 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 and when disassembling disassemble it to a JMP instruction.

# Traps

Whats the LC-3 without traps? For this requirement you will be implementing a fake 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 does the following:

R7 = PC;†

PC = mem[ZEXT(trapvect8)];

Where PC is the ALREADY incremented PC.

**You only do the above if the trap vector isn't one shown in the table below**

| Vector | Name | Pseudocode |
|---|---|---|
| x20 | GETC | R0 = character read in from stdin do not echo back the character inputed |
| 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! You must also echo the character inputed |
| 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 and set the halted flag in the lc3machine struct. |

# Disassembling

You are to also support disassembling data into instructions via the list command of the simulator (which will call cmd_list) and through code via the function lc3_disassemble (which will be called by cmd_list) . For example if I had the data 0x1000. You should report that the instruction that is disassembled from that data is ADD R0, R0, R0.

As a special case 0x0000 should disassemble to NOP. Notice how none of the NZP bits are set so this instruction will never branch hence a NOP or no operation.

For TRAPs you may just disassemble them as TRAP xNUM (example. HALT → TRAP x25).

For RTI, and the reserved opcode 1101 You can just simply display them as ERROR as your simulator does not handle these.

You will implement the function lc3_disassemble to disassemble and instruction and print

it. Here is the format for each instruction.
RX means a register (ex. R0, R7, R5).
Imm5 means a 5 bit 2's complement immediate value (ex. -5, -1, -16, 15).
PCOffset9 means a 9 bit 2's complement offset to the PC.
PCOffset11 means a 11 bit 2's complement offset to the PC
offset6 means a 6 bit 2's complement offset
VECTOR is a 8 bit unsigned number express it with hex with a leading x. (ex x25).
**Print all offsets in decimal, print all immediate values in decimal, print out the trap vector with a leading. Do not print out a leading # for decimal values. You do not need to disassemble with symbol information.**

BR PCOffset9
BRN PCOffset9
BRNZ PCOffset9
BRNP PCOffset9
BRZ PCOffset9
BRZP PCOffset9
BRP PCOffset9
BRNZP PCOffset9 ;note this is the same as BR PCOffset9
ADD RX, RX, RX
ADD RX, RX, imm5
LD RX, PCOffset9
ST RX, PCOffset9
JSR PCOffset11
JSRR RX
AND RX, RX, RX
AND RX, RX, imm5
LDR RX, RX, offset6
STR RX, RX, offset6
NOT RX, RX
LDI RX, PCOffset9
STI RX, PCOffset9

JMP RX
LEA RX, PCOffset9
TRAP VECTOR

# 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 */
int halted;
} lc3machine;
```

Please refer to the function headers for more comments on what to do for each function. Keep the front end code in lc3sim.c and the actual simulation code in lc3.c

Keep in mind this assignment will be autograded and I will be calling each of your functions directly.

# 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.

# Other useful functions you should know about

For console input we will not be typing any invalid forms of the user input commands mentioned in the assignment. A couple of using functions for parsing input and described below as always you can look these up in the man pages another resource is http://www.cplusplus.com/reference/clibrary/cstring/
even though its a C++ site it also documents the C functions you can use within C

**char *strtok(char *str, const char *delim);**

The strtok() function parses a string into a sequence of tokens. On the first call to strtok() the string to be parsed should be specified

in str. In each subsequent call that should parse the same string, str should be NULL.

The delim argument specifies a set of characters that delimit the tokens in the parsed string. The caller may specify different strings in delim in successive calls that parse the same string.

**long int strtol(const char *nptr, char **endptr, int base);**

Each call to strtok() returns a pointer to a null-terminated string containing the next token. This string does not include the delimiting character. If no more tokens are found, strtok() returns NULL.

A sequence of two or more contiguous delimiter characters in the parsed string is considered to be a single delimiter. Delimiter characters at

the start or end of the string are ignored. Put another way: the tokens returned by strtok() are always nonempty strings.

The strtol() function converts the initial part of the string in nptr to a long integer value according to the given base, which must be between 2 and 36 inclusive, or be the special value 0.

The string may begin with an arbitrary amount of white space (as determined by isspace(3)) followed by a single optional '+' or '-' sign. If base is zero or 16, the string may then include a "0x" prefix, and the number will be read in base 16; otherwise, a zero base is taken as 10 (decimal) unless the next character is '0', in which case it is taken as 8 (octal).

The remainder of the string is converted to a long int value in the obvious manner, stopping at the first character which is not a valid digit in the given base. (In bases above 10, the letter 'A' in either upper or lower case represents 10, 'B' represents 11, and so forth, with 'Z' representing 35.)

If endptr is not NULL, strtol() stores the address of the first invalid character in *endptr. If there were no digits at all, strtol() stores

the original value of nptr in *endptr (and returns 0). In particular, if *nptr is not '¥0' but **endptr is '¥0' on return, the entire string
is valid.

# Testing

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

So a good testing strategy is to do this in baby steps.
0. You can implement the front end first or just set up a minimal environment where the program is loaded and you print things after execution is completed
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.

**Really If you do the above this is NOT how to code big projects or do things in the real world. I have given you fair warning!**

# 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. I would prefer you did not however. If you do please let me know what each file does in a README.
****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. -std=c99 -pedantic -Wall -O2 -Werror
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