CPSC 313: Computer Hardware and Operating Systems
Assignment #1, due Saturday February 13 at 11:59PM.

**NOTE: Late penalty of 33.333% per day, pro-rated by the minute. Late assignments not accepted after 2 days.**

# 1 Objectives

After completing this assignment, you should be

- Familiar with how machine instructions are represented in memory
- Able to write a C program that reads binary data and manipulates raw bytes to decode the in memory representation of a Y86-64 instruction.
- Be able to describe the functionality of the fetch stage of the Y86-64 processor

# 2 Special Notes

1. This assignment may be done in pairs or alone. Groups of 3 or more are not permitted. You are strongly encouraged to work with a partner.
2. Your program must be written in C. C++ or any other language will not be marked and you will get 0.
3. In the top level directory typing make must produce the executable program *fetch*. (The provided Makefile already does this, so it you make changes to it don't break it.)
4. To be marked, the program must compile and run on the department provided Linux undergraduate servers. If your program does not compile by simply typing make in the assignment directory it will not be marked and awarded 0. In this situation no part marks will be granted regardless of the amount of work done.
5. Your program must compile without warnings to be eligible for full marks. (If it has compilation errors it will not be marked see above.) You are not permitted to modify the makefile or change compilation options to suppress warnings or to allow the program to compile if there are errors.
6. You are not allowed to modify the makefile to suppress warnings, to eliminate the need for prototypes, or to change the version of the compiler used.
7. You will be docked marks if you add executable files or object code files to the repo. The only type of new files you are allowed to add to the repo are .c and .h files, or other files subsequently approved by the instructor.

# 3 Introduction

The goal of this assignment is to simulate the fetch stage of a Y86-64 processor. You will be provided with a number of raw binary files that represent the object code for the Y86-64 processor. Your program will incrementally read these files and simulate the fetch stage of the processor. For each instruction fetched it will printout the various internal register values as determined by the fetch stage. You will be provided with a function call to do the actual printing to ensure that all assignments produce output in the required format.

# 4 Setting up your team and getting your repo

Just like assignment 1 we will use version control (in the guise of Atlassian Stash, which uses git) to distribute the files that you will be working on. It will also be the way that you handin your assignment.

As indicated above you will be allowed to do this assignment either working by yourself, or with a partner; teams of three or more are not allowed. Before you can access the git repo with your copy of the assignment, you will need to register your team, or yourself if you aren't working with a partner, using the *Register your team link* available on the Assignment page of Connect. You must do this even if you are working by yourself in this case you leave the Partner field blank. When you are working with a partner, both of you must register your team before the assignment repo will be created. For additional information on using git and the format of repo names etc., see the description in assignment 1.

As you work on your code you will want to periodically "push" changes to the Stash repository. It is highly recommended that you do this frequently. You might want to push these changes every time you stop working on the code and when you get to important implementation milestones.

## 4.1 Repo Files

When you clone the assignment repo you will have the following files in the top level directory:

- coverpage.txt - the coverpage all people working on the assignment was modify and commit
- fetchstage.c - starting code for the assignment
- printInternalReg.c - the file containing the routine printReg() that formats the the registers for printing.
- printInternalReg.h - The file with the prototypes and constants needed by printReg()
- Makefile - A file with the compilation rules for building the *fetch* program

# 5 Your task

Once you get your repo you should go into the top level directory and type make and then run the fetch program. (This assumes you are on a department Linux machine. The code is pretty generic so it should compile and run on most Unix type machines, but you never know.) At this point it will print some information as to how it should be run. The program has one required parameter and a second optional one. The parameters are:

- The name of the file containing the object code to fetch.
- The optional 2nd argument specifies the initial value the program counter (PC) is to be set to. If this argument is omitted then the default value to set the PC to is 0.

The programming task can be described in a fairly straight forward manner:

1. Process command line arguments, set the PC, and open the data file.
2. Use the value of the PC to determine where to read from in the binary file.
3. Read data from that location and perform the actions of the fetch stage.
4. Call the printReg() routine to printout the values of the internal registers
5. Set the PC to valP and go back to step 2 until there is an error or no more data.

A description of the parameters that need to be passed to printReg() can be found in the file printRoutines.c. Initially, if you run the provided program with the name of any readable file for the first parameter the program will provide a printout of what a line of fetched code is to look like. Your program must use the provided function to print the internal register values.

## 5.1 Interpreting Instructions

When started, your program is to seek to the offset of the input file as specified by the initial program counter value. Your program will keep fetching and printing instructions until one of the termination conditions is reached. The termination conditions are:

- Normal termination: An attempt to read at the location specified by the program counter returns an EOF. The program is to print, on a line by itself "Normal termination" and exit.
- The opcode for the instruction is invalid: The program is to print "Invalid opcode YYY at XXX" and exit. YYY is to be replaced with the hex representation of the byte containing the opcode and function code. XXX is to be replaced with the hex value of the address of this invalid instruction.

- The function code for the instruction is invalid: The program is to print "Invalid function code YYY at XXX." and exit. YYY is to be replaced with the hex representation of the byte containing the opcode and function code. XXX is to be replaced with the hex value of the address of this invalid instruction.

- Insufficient bytes to complete the instruction fetch. In this situation you can read the first byte to determine the instruction and how many subsequent bytes to read, but there aren't that many bytes available. For example, you might not be able to read the register byte, or you might be able to read the register byte but there might not be enough bytes for valC, or if the instruction doesn't have a register byte you might again be short the required number of valC bytes. In this situation you are to first call printReg() with as much information as possible. (i.e. if you read the registers but not valC, have the instruction and registers printed, If part of valC was read set the the unread bytes to 0 and call printReg() as appropriate. Once that has been done your program is to print "Memory access error at XXX, required YYY bytes, read ZZZ bytes." and then exit. XXX is to be replaced with the hex representation of the address the start of the instruction was being read from. YYY is to be replaced with the decimal value of the number of bytes this instruction is to occupy in memory. For example if the instruction were *rrmovq* it would be 1 and for *call* it would be 9. ZZZ is to be replaced with the actual number of bytes read while interpreting the instruction. (i.e. YYY - ZZZ will be the number of missing bytes.)

After printing a fetched instruction the PC is updated to valP and the process starts over. Note this means that for instructions like jmp, call, ret, nothing special needs to be done. Your code simply assumes that the next instruction to be executed follows the current instruction. If the program were actually executing that might not be the case, but this approach simplifies the program.

### 5.1.1 That halt instruction

The halt instruction is 00. When interpreting memory your program may encounter multiple halt instructions in a row. Sometimes this is intentional and other times it may simply be that the program is fetching instructions from an area of memory set to 0. To avoid printing many halt instructions in a row your program is to keep track of how many consecutive halt instructions it has encountered. Once it has printed 5 consecutive halt instructions it is to stop making calls to printReg() until either a non-halt instruction or termination condition is encountered.

### 5.1.2 Unconditional moves and jumps

If an unconditional jump is encountered the instruction mnemonic to use is jmp. If an unconditional move is encountered the instruction mmnemoic to use is rrmovq.

### 5.1.3 Test files

In the assignment directory you will find some test files. (More will be added as the due date approaches) The files ending in .mem are the binary files your program is to read and act like the fetch stage on. The source used to produce the .mem file is contained in the .s with the same name. Under no circumstances are you to commit these files to your repo. If you do you will be penalized 10 percent.

## 6 Deliverables

You are to use stash to submit your assignments by simply making sure that your final version is pushed to the stash repo. Of course you should be regularly pushing your code to track the changes you are making. It is good idea to make sure that every time you get something working you commit and push it.

For your final commit you will want to make sure that you commit the appropriate versions of the following files:

- The modified versions of any of the initial .c, .h or Makefile files.
- Any added .c or .h files needed to allow your program to compile and run. If you add .h or .c files make sure to modify the Makefile appropriately. **DO NOT BREAK THE MAKEFILE.**
- The filled in version of coverpage.txt. Follow the directions in the coverpage.txt file on how to fill-in and submit the file.

When you have pushed your final version of the assignment you should sign-in to stash to verify that you have committed and pushed the versions of the files you want marked. You might want to clone a new version of your repo just to make sure everything is as you expect.