# CS 211: Computer Architecture, Spring 2014
# Programming Assignment 4: Cache
# Due: 11:55 PM, 5/5/2014

## 1 Overview

This assignment is designed to give us a better understanding about cache behavior. We will write a cache simulator using C programming language. This assignment is much complex than it looks like. Be careful! It will require a substantial implementation effort as well as time and effort to explore and analyze cache behaviors. The usual warning goes double for this assignment: **do not procrastinate**. The programs have to run on iLab machines.

## 2 Memory Access Trace

The files trace[1|2].txt, provided with assignment description, are example memory access traces of different length. Each line is for one memory access. First column is the address of the instruction that is causing memory access. R (W) indicates the operation is a memory read (write). Finally, last column is the memory address that is being accessed.

## 3 Cache Simulator

You will implement a cache simulator to evaluate different configurations of caches, running it on different traces files. The followings are the requirements for the simulation:

1. Simulate only one level cache: **L1**;
2. The **size of the cache**, **associativity** and **blocksize** are parameterizable;
3. Replacement algorithm: **LRU**;
4. Implement **write through cache** and **write back cache**.

### 3.1 Invocation Interface

Implement a program **c-sim** that will simulate the operation of a cache. Your program **c-sim** should support the following usage interface:

c-sim [-h] <cache size> <assoc> <block size> <write policy> <trace file>

where:

<cache size> is the total size of the cache. This should be a power of 2.

<assoc> is one of:

**direct** - simulate a direct mapped cache;

**assoc** - simulate a fully associative cache;

**assoc:n** - simulate an *n-way* associative cache.

<block size> is an power of 2 integer that specifies the size of the cache block.

<write policy> is one of:

**wt** - simulate a write through cache;

**wb** - simulate a write back cache.

<trace file> is the name of a file that contains a memory access trace.

If -h is given as an argument, your program should just print out help for how a user can run the program and then quit.

As examples, running the simulator using the following options should produce:

./c-sim 16384 direct 4 wb trace1.txt

Cache hits: 710738
Cache misses: 30478
Memory reads: 30478
Memory writes: 12539

./c-sim 4096 assoc:4 16 wt trace2.txt

Cache hits: 8363
Cache misses: 1637
Memory reads: 1637
Memory writes: 2861

./c-sim 4096 assoc 4 wb trace2.txt

Cache hits: 6725
Cache misses: 3275
Memory reads: 3275
Memory writes: 2251

### 3.2 Simulation Details

1. (a) When your program first starts running, all entries in the cache should be invalid; (b) the number of bits in the tag, cache address, and byte address are then determine by the cache size and block size; (c) Your simulator should simulate the operation of a cache according to the given parameters for the given trace; (d) at the end, it should print out the number of cache hits, cache misses, memory reads and memory writes.

2. For this assignment: (a) a write-miss causes both a read and a write from the cache to the memory; (b) future reads or writes to any location in the newly brought in block will hit in the cache until the block is evicted because of replacement. This makes write-through and write-back caches look alike as much as possible, easing your implementation.

3. For a write-back cache: (a) add a bit (called the dirty bit); (b) set the dirty bit when writing to a block; (c) when evicting a block, if dirty bit is set, write to lower-level (memory).

## 4 Submission

You have to e-submit the assignment using Sakai. Your submission should be a tar file named `pa4.tar`. To create this file, put your source code, Makefile, and readme.pdf in a folder named pa1; `cd` to the directory containing this folder; then run the following command:

```
tar -cvf pa4.tar pa4
```

To check that you have correctly created the tar file, you should copy it (`pa4.tar`) into an empty directory and run the following command:

```
tar -xvf pa4.tar
```

This should extract all the files that we are asking for below directly into the empty directory.

You should use tar command to compress your submission. If you use the compression tools like zip, rar, you MUST make sure that your submission can be decompressed and opened, if not, you would lose all the points for doing so.

Your tar file must contain:

- readme.pdf: this file should describe the design and implementation of your cache simulator. How did you implement the direct mapped cache, the n-way associative cache, the replacement algorithm, and the write back vs. write through caches? For this assignment, you do not have to worry about analyzing the time and space behavior of your program.
- Makefile: there should be at least two rules in your makefile:

  `c-sim:` build your `c-sim` executable.

  `clean:` prepare for rebuilding from scratch.

- source code: all source code files necessary for building `c-sim`. You are recommended to put any global definitions and function declarations in the header file, while put function definitions in the source file.

We will compile and test your program on the iLab machines so you should make sure that your program compiles and runs correctly on these machines. You must compile all C code using the gcc compiler with the `-ansi -pedantic -Wall -m32` flags.

# 5 Grading Guidelines

## 5.1 Functionality

This is a large class so that necessarily a significant part of your grade will be based on programmatic checking of your program. That is, we will build a binary using the Makefile and source code that you submitted, and then test the binary for correct functionality against a set of inputs. Thus:

- You should make sure that we can build your program by just running `make`.
- You should test your code as thoroughly as you can and MAKE SURE they can run on the ilab machines. *In particular, your code should be adept at handling exceptional cases.* For example, `c-sim` should *not* crash if the argument trace file does not exist.

Be careful to follow all instructions. If something doesn't seem right, ask.

## 5.2 Coding Style

Having said the above about functionality, it is also important that you write "good" code. Thus, *part of your grade will depend on the quality of your code.* Here are some guidelines for what we consider to be good:

- Your code is modularized. That is, your code is split into pieces that make sense, where the pieces are neither too small nor too big.
- Your code is well documented with comments. This does not mean that you should comment every line of code. Common practice is to document each function (the parameters it takes as input, the results produced, any side-effects, and the function's functionality) and add comments in the code where it will help another programmer figure out what is going on.
- You use variable names that have some meaning (rather than cryptic names like `aa`).

Further, you should observe the following protocols to make it easier for us to look at your code:

- Define prototypes for all functions.
- Place all prototype, `typedef`, and `struct` definitions in header (.h) files.