# CS203 Fall ’18 - Lab 2

Due: 11 November 2018, 11:59 pm PST

## 1. Objective

The goal of lab 2 is to design and implement a simple cache simulator, with support for direct-mapped, set-associative, and fully-associative mapping. In this lab, we also go further and investigate the effect of victim cache on the cache subsystem performance.

## 2. Memory Access Traces

To facilitate the testing of your cache simulator, we will utilize two sets of traces collected from a run of gcc. The traces `gcc-10K.memtrace`, and `gcc-1M.memtrace` are available at the following git repository: https://bitbucket.org/danwong/cs203-labs-f16 under the Cachesim subdirectory. The traces contain ~10 thousand and ~1.5 million entries. A sample of the trace is given below:

```
L -200 7fffe7ff088
L 0 7fffe7fefd0
S 8 12ff228
S 8 12ff208
L 0 a295e8
```

Each line in the trace file contains the memory instruction type (L = load, S = store), the offset, and the memory address in hexadecimal. Note that this trace was obtained from an x86 machine, and thus the memory address is 44-bits. For the purpose of this class, we can assume 32-bits and you can truncate the most significant 12 bits.

## 3. Cachesim

Now that you have a better idea of how simulators are implemented, this project will be a bit more open ended. You will implement from scratch a C++/Java/Python based cache simulator (let’s call it Cachesim). Cachesim is a performance simulator, with the main focus on collecting cache miss and cache hit rates.

### 3.1. Part 1

a) In this part, you should develop a Cachesim with input arguments (flags) listed in table below:

| Flag | i | cs | bs | w[1] |
|---|---|---|---|---|
| Meaning | Input file | Total Cache Size (KB) | Cache Block Size (B) | Number of Ways |
| Possible Values | - | 4096>cs>1 | 2, 4, 8, 16, 32, 64 | 0, 1, 2, 4, 8, 16 |

[1] For flag “w”, value 0 represents a fully-associative, value 1 represents a direct-mapped, and other values represent number of ways in a set-associative cache.

## 3.2 Part 2

A victim cache is a small cache placed in the refill path of CPU cache. It stores all the blocks evicted from that level of cache. In this part, Victim cache should be added to Cachesim. Consider following notes in your implementation:

- Victim cache is fully associative, and uses LRU as replacement policy.
- Victim cache behavior is explained in the below table.

| | | CPU Cache | |
|---|---|---|---|
| | | Hit | Miss |
| Victim Cache | Hit | - | - Replace the block in the victim cache and the one in the cache - Set the new entry in victim cache as the most recently used block |
| | Miss | - | - Get the block from the next level - The block evicted from the cache gets stored in victim cache |

- Add "vs" to Cachesim flags in order to specify the size of the victim cache, which is number of cache blocks that can be stored in it. The possible values for "vs" argument are 4, 8, and 16.

### Question

Using Cachesim, fill the tables of *Lab2_answer_tamplate* file with **hit rate**. Compare Table1 and Table2, and explain the differences. Assume we have a 512KB cache and 16B block size, and use `gcc-1M` trace for Table 2.

### Notes

- For simplicity, we can make the following assumptions:
  - Maximum of 4MB cache size
  - Assume Least Recently Used (LRU) cache replacement policy. Initialize all entries, tag, and LRU-bits of the cache to 0's.
- For sample code of how to read a file, refer back to the Pipesim source code.
- Output the cache miss and hit rate, as well as the number of sets, ways, and number of address bits for the tag, index, and offset.

### Deliverables

1. **ilearn**: please include a **shell script that produces each entry of the tables**, and documentation on how to run your script. Submit your source code and the script in a single zip file.
2. **Gradescope**: please upload your answers in the provided template answer file (*Lab2_answer_tamplate)* as a pdf.