Programming Project #2

Due Date: 12/7, Tue., 11:59 p.m. ***Please submit via Blackboard. Please zip or make a tar ball of all your files, including source codes, output, etc., and name the file as LastName_FirstName_PP2.zip/.tar.gz.***

In this programming project, you are asked to develop a simple cache simulator and perform tests to observe how cache behaves. This programming project assists you for deep understanding of computer architecture in general, and cache hierarchy and memory systems in particular. This programming project intends to reinforce our discussions in class and your understandings of lecture topics.

Please note that we focus on simulating a **single-level cache** in this project and you do not need to consider a multi-level cache simulation. In addition, we assume a 32-bit machine and memory addresses fed from trace files are 32-bit byte addresses.

Part 1. Direct-mapped cache

We have provided a functional, sample code to you. In this part of the programming project, you are required to compile the *cachesim.c* we have provided on the Blackboard to generate the single-level direct-mapped cache simulator. Then run the *cachesim* with provided memory traces and report the result. We recommend you use *gcc* compiler, but you can use other C compilers too. The following example assumes a Linux/Unix/Mac environment with a bash shell. Depending on the platform you develop and test your codes, the commands can be different.

To compile the source codes, use a command like below. This command compiles cachesim.c source file and generates a binary, executable file, cachesim.

```
$ gcc cachesim.c -o cachesim
```

To run the sample simulator, please use a command like below. You need to use the argument *direct* to let simulator know it is the direct-mapped cache, and the argument *tracefile* represents the path of the file that contains the memory traces.

```
$ ./cachesim direct tracefile
```

Requirements of Part 1:

1. The provided sample simulator only measures the total cache hits and misses. Please make modifications to the *cachesim.c* source code to calculate the cache **miss rate** and **hit rate**, and use the print statements to print out the results.
2. Please compile and run the modified simulator and report the hit and miss rate for each given trace.

Part 2. Fully associative and n-way set associative cache

In Part 2 of this programming project, you are asked to further develop the simulator to simulate fully-associative cache and n-way set associative cache.

Requirements of Part 2:

1. Based on the direct-mapped cache simulator, please extend to support the simulation of a *fully-associative cache*.
2. Based on the direct-mapped cache simulator, please extend to support the simulation of an *n-way set associative cache*.
3. Both fully-associative cache and n-way set associative cache should replace a cache block (or cache line) with the **random** policy. It means that, if a set is full, then one random block should be picked and evicted. You can use a random number generator function (e.g., `rand()` or `srand()` in C) to generate the random number to select the replacement candidate. This replacement algorithm does not require keeping any information about the access history. Due to its simplicity, it has been used in numerous processors, e.g., ARM processors.
4. Users should be able to use the argument to switch between different cache models: direct-mapped, fully-associative, or n-way set associative.
5. Conduct the evaluation and analysis as specified in detail below.

Part 3. Evaluation and Analysis

The cache under simulation can be configured with different settings, e.g., different cache capacity (or cache size) and different cache line size. For instance, if we configure the cache size as 32KB and the cache line size as 64 bytes, then there are 512 cache lines in total (32KB/64B=512). Given the same cache size of 32KB, if the cache line size changed to 32 bytes, then we will have 1,024 cache blocks in total. Please note that the cache line size affects how to determine the index and tag. In this part, you are asked to perform the following evaluation with given memory traces:

Requirements of Part 3

1. Given a fixed cache size of 32KB, test the fully-associative, 8-way set associative, 4-way set associative, and 2-way set associative cache with cache line size of 16 bytes, 32 bytes, and 128 bytes, respectively. Please report the hit and miss rate in each case.
2. Given a fixed cache line size of 64 bytes, test the fully-associative, 8-way set associative, 4-way set associative, and 2-way set associative cache with the cache size of 16KB, 32KB and 64KB, respectively. Please report the hit and miss rate in each case.

Part 4. Extra Credit

If you are interested in earning a **40% extra credit**, please read the supplement file.

Expected Submission:

You should submit a single tarball/zipped file through the Blackboard containing the following:

- All your source codes;
- Output files for the result/test cases for part 1 and part 3 (in TXT format).
- **Extra credit (20%):** a report that summarizes all your test results in a table format, or in charts (i.e., plotting these data via a plotting tool like Excel, gnuplot, MatLab, etc.) and your findings/insights. There is no required format/template, so please be creative.

Grading Criteria:

Please find the detailed grading criteria from the below table. **Please note that we may require an in-person demo for grading this project.**

Part 1	Percentage %	Criteria
20%	50%	Correct modification
	50%	Correctness of result
Part 2	Percentage %	Criteria
60%	20%	Inline comments to briefly describe your code
	30%	Correctly implement the fully-associative cache simulation
	30%	Correctly implement the n-way set associative cache simulation
	20%	Implement the random replacement algorithm
Part 3	Percentage %	Criteria
20%	50%	Carry out and report specified test cases in step 1.
	50%	Carry out and report specified test cases in step 2.
20% Extra Credit		A technical report summarizing your results, findings, and insights.
Part 4	Percentage %	Criteria (Extra Credit)
40% Extra Credit	60%	Correctly implement the PLRU replacement algorithm.
	40%	Carry out specified test cases and report results and findings.

THE END.