Programming Project
CS3375 Computer Architecture
Fall 2017

Instructor: Dr. Yong Chen **Office:** EC315 **email:** yong.chen@ttu.edu
Instructor Office Hours: 11 a.m. – 12 p.m., TR, or by appointment
TA: Mr. Dian Li **Office**: EC201A **email**: dian.li@ttu.edu
TA Office Hours: Mon. and Wed., 2 p.m. - 3:30 p.m.

Updated on 10/31/2017: added an additional 20% extra credit; please check out Part 5.

Due Date: 11/30, 11:59 p.m, soft copy via Blackboard.
Late submissions are accepted till 12/5, 11:59 p.m., with 10% penalty each day.
No submissions accepted after 12/5, 11:59 p.m.

Please zip or make a tar ball of all your files, including source codes, output, etc., and name the file as LastName_FirstName_Project.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 **NRU (Not-Recently Used)** policy. It means that, if a set is full, then one of NRU blocks should be picked and evicted. Please see hints below for more info.
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. Please test both fully-associative and n-way set associative cache with given memory traces. In the case of n-way set associative cache, please test with 4-way, 8-way, and 16-way, respectively. Please report the miss rate and hit rate.

Hints for **NRU** replacement implementation (e.g. used by Intel Itanium, Sparc T2):

1. In the NRU replacement algorithm, each cache block has one *reference* bit. This bit is set to 1, if this cache block is referenced, i.e. accessed (either read or write). This bit can be cleared to 0 periodically, as described below.
2. When a replacement candidate is needed, if one cache block with the reference bit as 0, then this block is an NRU block and can be a candidate for eviction to make the room for the new cache block. If there are multiple NRU blocks, one NRU block can be randomly chosen as the replacement candidate. If all blocks in a set have the reference bit as 1, then clear the reference bit to 0 for all blocks, and randomly pick one block as the replacement candidate.

Part 3. Evaluations

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:

Requirements of Part 3

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

Part 4. Extra Credit

If you are interested in earning a 20% extra credit, please read the supplement file.

Part 5. Extra Credit

If you develop a Makefile to automate the compilation process with the *make* utility, you can earn another 20% extra credit. A sample Makefile can be found from slide 16 of lecture 18. More info about Makefile and make utility can be found from: http://www.gnu.org/software/make/manual/html_node/index.html.

Expected Submission:

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

- Source codes for all parts 1, 2, and 3.
- Output files for the result/test cases for part 1, part 2, and part 3 (in TXT format).
- A brief report that summarizes all your test results either in a table format, or in charts (i.e. plotting these data via a plotting tool like Excel, gnuplot, MatLab, etc.)

Grading Criteria:

Part 1	Percentage %	Criteria
20%	50	Correct modification
	50	Correctness of result
Part 2	Percentage %	Criteria
60%	10	Inline comments to briefly describe your code
	25	Correctly implement the fully-associative cache simulation
	25	Correctly implement the n-way set associative cache simulation
	20	Implement the NRU replacement algorithm
	20	Successfully carry out specified test cases
Part 3	Percentage %	Criteria
	50	Carry out and report specified test cases in step 1.

20%	50	Carry out and report specified test cases in step 2.
Part 4	*Percentage %*	*Criteria (Extra Credit)*
20% Extra Credit	*60*	*Correctly implement the PLRU replacement algorithm.*
	20	*Correctly implement the RR replacement algorithm.*
	20	*Carry out specified test cases and report results and findings.*
Part 5	*Percentage %*	*Criteria (Extra Credit)*
20% Extra Credit	*100*	*Correctly implement a functional Makefile that can automate compilation process with the make utility.*

Please note that we may require an in-person demo for grading this project.