COSC6310: Computer System Architecture

## Final Project

## **Objective**

The purpose of this project is to give you experience in building a Level 1 data cache simulator and run a set of experiments using your simulator. This is a team project, and each team has 3 members.

## **Specifications:**

1. You can write your program in Java.
2. Your Level 1 Data Cache Simulator should accept the following command-line options:   
   **-c <capacity> with <capacity> in KB: 4, 8, 16, 32, or 64.   
   -b <blocksize> with <blocksize> in bytes: 4, 8, 16, 32, 64, 128, 256, or 512.   
   -a <associativity> where <associativity> is integer size of set: 1, 2, 4, 8, 16.**

The baseline configuration (i.e., the first one that you should test) is: capacity = 8K, blocksize = 16 bytes (i.e., 4 words), set associativity = 4 with the following command if you are using the Java command (NOTE: the code we provide requires no spaces between the letter and number):

**java cache\_sim -c8 -b16 -a4**

1. Additionally, your cache should have the following functionality:

* write back (for write hits)
* write allocate (for write misses) - read the cache line, modify it with the write, then mark it as dirty.
* LRU replacement

1. The input to the cache simulator will be a sequence of memory access traces, one per line, terminated by end of file. In the following format, with a leading 0 for loads and 1 for stores.

* 0 <address>
* 1 <address> <dataword>

Each address will be the address of a 32-bit data word. The address and data are expressed in hexadecimal format.

1. To support testing of your data cache simulator, you will implement a simple model of main memory. The capacity of the memory should be 16 MB (4 megawords). At simulator start-up time, **initialize** the contents of each word to be the word's address. For example, the 32-bit word that starts at byte 0x00001004 should be initialized to 0x00001004.
2. The output of your program will consist of a set of statistics gathered during the simulation as well as the contents of the cache and memory at the end of the simulation. The format of the output is described as below. This consistent output format is required for automate grading.

**Format of Output:**   
**Key to the following description:   
base of output: (10 = decimal and 16 refers to hex)   
rate should be a real number, with 4 digits of accuracy   
0/1: print either 1 for True or 0 for False**   
  
-----------------------------------------------------------------------------  
STATISTICS:   
Misses: <Total (10)> <DataReads (10)> <DataWrites (10) >   
Miss Rate: <for total> <for dataread> <for datawrites>   
Number of Dirty Blocks Evicted From the Cache: <Total (10)>   
  
CACHE CONTENTS:   
Set\_Number Valid Tag Dirty Word0 Word1 ...   
<base 16> 0/1 <16> 0/1 <16> <16>...   
...   
  
MAIN MEMORY: {print 1024 words of memory starting at address 0x003f7f00. Print 8 words of memory per line, all values should be base 16}   
  
<address of word0> <word0> <word1> <word2> ... <word7>   
<address of word8> <word8> <word9> <word10> ... <word15>   
...   
<address of word1016> <word1016> <word1017> <word1018> ... <word1023>

We provide a framework below for the Java code to get you started. These files will help with input and output, so you can focus on cache simulation.

* **cache\_sim.java** is the top level entry point for the project and contains the parseParams routine.
* **io.java** contains routines for reading in data and writing out final cache statistics and contents. You should move these routines to your own classes and code.

1. Your program will be graded for correctness and design.

* **Design**: We expect to see well-chosen classes that reflect the application. Be sure that your method names and variable names reflect memory hierarchy terms. When done, anyone reading your code should have learned how a cache works. Towards that end, you should define constants such as "dirty".
* **Correctness**: You should create your own trace files to help you test and debug your cache configurations.

1. Keep in mind these details

* You should initialize all of the cache locations to be 'invalid' and initialize all of the tag and data fields to zero.
* All of the parameters on the **command line** are base 10 numbers (e.g. 32, 64, 128, etc.)
* The addresses and data in the trace files are expressed as base 16 (hexadecimal) numbers. Note that these numbers might (or might not) have extra zeros in front... e.g. 00de2a vs. de2a.
* Read your input from 'stdin' and write your output to 'stdout'. (i.e., read from the terminal and write to the terminal)
* Also, use the Java input and output stream utility to pull input from disk files and write output to disk files. For example:

**java cache\_sim -c32 -b16 -a4 < test.trace > test.output**   
NOTE: No spaces between the number and the letter, which the code we provide requires.

* Your output should be formatted in the Output format as described above. You will print out the entire contents of the cache, and 1024 words of memory starting at address 0x003f7f00.
* We have provided the output code and a tool to check it (see the testing section below). If you do not use the correct format, we will not grade your project.
* At the end of the trace, write all dirty blocks to the memory, so that the memory contains updated data. However, do not count them in "**Number of Dirty Blocks Evicted From the Cache**", since they have not really been evicted from the cache.

## **Testing your simulator**

We have provided a perl script (***checker.pl***) that checks the formatting of your program's output and sample files.

* In order to understand the code, you are required to get familiar with perl script programming, which is part of this project.
* We will use a variation of this script for grading. You should verify that it can parse your output correctly. We will not grade your project if we cannot parse your output.
* Input file: **sample\_input\_file.trace**
* Output file from above for configuration -c4 -b32 -a4 : **sample\_output\_file.txt**
* File containing input traces for test cases #1-#4: **smalltraces.txt**
* File containing main address trace: **mem.trace**

We are not providing the solutions to the four small test cases, but the test cases are small enough that you can hand calculate the results. Once your program is complete and passes the simple test cases, you can use the trace file from the compress benchmark (**mem.trace**) for a "stress test" and for the experiments as described below.

## **Report and Experiments**

After you test your cache simulator, you will use it as a tool to evaluate several cache configurations, to determine which one results in the lowest miss rate for the compress benchmark. The trace file for the compress benchmark in **mem.trace**. You will write a 3-page report summarizing the design of your simulator and your experimental results. Your report and experiments should be structured as followed.

1. Describe your experiences in the development of your code.

* How did you structure your code and why?
* What were the challenges in simulating a cache and memory?
* Describe how you partitioned the 32 bit addresses into tag, index, and offset, when given a particular configuration.

1. Rather than running all possible combinations of cache parameters, you will develop a plan that tests a partial set of combinations. Describe your plan and justify it.
2. Provide a table of the statistics you collected from each of the five best configurations in rank order (best first) and specify the criteria for choosing the best.
3. Discuss factors we did not consider in these experiments that architects must consider before selecting a specific cache configuration, and how these factors would influence the selection. You should consider potential cache access time (i.e., hit time), the die area that the cache might occupy, etc.
4. List a contribution table for each member’s contribution.

|  |  |  |
| --- | --- | --- |
| Aaron Rodgers | Clay Matthews | Jordy Nelson |
| 100% | 100% | 100% |

## **Delivery**

1. **Due Time**: Week 15, Thursday
2. All of your source files. All of these files must be well documented with comments.
3. If you use Eclipse for the project, please zip your whole project as one file.
4. Five output files: The output from simulating test cases 1 through 4, and the stress test (**mem.trace**) using the baseline configuration
5. An electronic copy of your report (in either word document format, named 'report.docx’ or PDF format, named 'report.pdf')
6. You need to zip everything together as one file, and submit it on Blackboard by the Due time. **Email submission is not accepted. Each team only needs to submit one copy on blackboard.**

## **Bonus**

1. Implement a GUI for this project will get 5~15% bonus