**CSCE 312: Computer Organization – Final Project**

Texas A&M University, Spring 2020

Date: 04/25/2020

|  |  |
| --- | --- |
| **Student 1 Name:** Charles Dinges  **Student 2 Name:** Andrew Whitman | **Student 1 UIN:** 327001104  **Student 2 UIN:** 526009154 |

**Introduction**

This report discusses the construction of a minimized command line based cache simulator. As a team, we were required to design the structure of the cache in the code—deciding which data structures would best simulate the cache—and implement the logic needed to read and write to the cache, view the cache data, and view the RAM data.

**Design and Implementation**

The cache simulator we developed is composed of four classes named Memory, Set, Cache, and CacheLine. The Memory class has a single private member which is a vector of strings. This vector holds each byte of memory in the RAM as a string. The Set class has three private members. The first two are integers E and B, representing the number of lines per set (associativity) and the block size respectively, and the third is a vector of CachLines. The Cache class has many private members which represent the structure of the cache, such as the cache size in bytes, the block size in bytes, the number of cache lines per set (associativity), the number of sets, and the various number of bits and their types for accessing cache and memory location. This class also has a member of type Memory named “RAM” and a vector of Sets. Finally, the CacheLine class represents an individual line of data in the cache. This has private members representing the block size, valid bit, dirty bit, tag bits, the time when the line was last accessed, and the frequency that the line is accessed. It also has a vector of strings which holds each byte of data. We decided to use vectors to eliminate any complexity of pointers on the heap. Vectors are simple to access and modify on the stack, so we knew that this would be a simple but effective approach.

In order to read and write to the cache, we needed to make use of each class and their functions that interact with one another. For both functions, a valid input is first checked. The input must be given as “0x\_\_” with the right-hand side being a hexadecimal value greater than or equal to 0, but no greater than the size of the RAM. We then convert this hex value into binary to get the correct set, tag, and offset bits. Using these bits, we locate the data byte in the RAM, extract the entire line that it is in, and search for it in the cache by comparing tag bits. Depending on the function used (read or write) and whether it is a hit or a miss, protocol dictated by the user is followed to update the cache and/or the RAM.

**Conclusion**

Our implementation was simple because of the use of vectors, but it was complicated by the number of classes we created. We found that when reading or writing to the cache, many functions were called on one class to another and then on the second class to a third. This was tedious and unnecessary. If we were to do this again, we would simplify the number of classes used in order to simplify the code and strip down unneeded sections. Overall, our code is complete, effective, and correct. It simulates a minimized cache well.