CS 231 – Lab #9 – Simple Cache Simulation

In class, we looked at the behavior of caches. Today in lab you will be working on a high-level language program that follows the same steps you did by hand, but now in a more automated fashion. The cache we will model today is just like the "small" cache example we have been working with: 8 lines and 4 words per line. However, the program is written so that other cache sizes are possible, but in testing you may focus on just this one size.

Log in as usual. On your USB drive or account, create a new folder called lab9, and do all your work in there. On the class Web site, you should find a folder called lab9. You should download the test input files 1.txt, 2.txt and 3.txt. I have also provided incomplete implementations for you to finish. If you prefer Python, then use cache.py. If you prefer Java, then you will need these 3 files: Driver.java, Program.java, and Cache.java. Whichever language you use, the total amount of code to add will be about the same. If you are using Java, then all of the code you need to write will be in Cache.java.

Here are the major steps of the program. Most of the implementation has already been done for you. But two important parts are missing. They are identified by comments in the code.

1. Initialize the cache. We can represent the cache contents as a 2-dimensional array or list. And the set of tags can be represented as a 1-dimensional array or list. **An important part of the implementation has been left for you to finish**: Inside cache.py or Cache.java you should see a function called calcBits. The purpose of this function is to determine the number of bits to assign to the tag, the line number, and the word number. These values need to be calculated based on the cache configuration (number of lines in the cache, and the number of words per line).

2. We read the input. The input consists of a list of hexadecimal addresses, each of which represents the address of something in memory we could be fetching. These addresses will be stored in a collection. In cache.py, this is the "list". In Cache.java, we use a LinkedList to hold the addresses. A ListIterator object is used to traverse the list. To save you some time, the code as currently written has already obtained the appropriate address from the list for you to process inside the run() method.

3. Now we are ready to run the simulation. For each address, you need to determine if it is a hit or miss in cache. **This is the 2nd part of the program you need to finish.** Your program should count the total number of hits and misses. (These variables have already been declared.) If a miss, then your program must also bring in the appropriate line into cache. I have included code that will print the contents of the cache. For now, it is guarded with "if true …" – please change the if condition so that we only print the cache state when there is a miss.

4. When the simulation is over, we print the final results of the simulation – the total number of hits and misses, as well as the miss rate. The miss rate is defined as the percentage of memory accesses that were misses.

As you work on your implementation, you should focus just on input files 1.txt and 2.txt. The input file 3.txt is much longer, and you should only attempt that one when you think your implementation is finished.