# REPORT

## How code works?

Code starts with taking the arguments. If there are no arguments provided, it simply starts from the default states of cache (Default states are explained it at *How to run from command line* section). Later, it creates the fundamental elements of cache simulation such as I/O variables and cache's itself. Those two are the basic steps for simulation to begin, as it is desired.

The important part of code begins with reading the lines. Program will read each of the lines, pretend that they are the addresses wanted to be touched in cache and taking action according to the situation it should be. In each of the line there are few possibilities that cache can do. It can be "miss", "hit", "miss replaced – miss new", and "write back". Each of the action means what it means in computer architecture actually. However, we would like to briefly explain what they do mean according to the behavior of our cache in the simulation:

- *Miss* means the address did not find in the cache, so cache had to bring from the memory to have the desired data.
- *Hit* means the address was found in the cache, so cache simply returns to desired data to CPU.
- *Write back* is a technique of writing data. Write back means if cache's data has been modified, when an address needs replacement with some other address write to the memory the modified status then take the new address into the cache. For this purpose, keep a "dirty bit" to see if it is modified. If it is modified, make dirty bit 1 (Initially 0).
- *Miss replaced – miss new state* is a state what we have made up, actually. That state is created to observe whether the replaced data is a modified or not. If the data, which is going to be replaced, does not have a dirty bit, it simply

replaced with the new address instead of writing to the memory first. It is different than the write back. To have accurate write back states, we have created a made up state so that we can understand whether the replacement is a just a replacement or it is a write back.

The detailed information about how cache took an action against the provided addresses in the addresses stream file is written in the detailed log file. It explicitly writes what cache did against the addresses. Furthermore, opening the debug option (means giving the last argument as 1) will cause to write into the detailed log file the initial state of cache in each of 100 steps. However, independent to the debug option, you will be able to see the final state of the cache in every run, at the end of the detailed log file.

Cache class is created to be able to have a class that behaved like a cache. It is the cache using random policy as its replacement method. In its *constructor* it makes every necessary calculations and its *add* method is controlling how the cache should take action against the address provided. Note that addresses are converted into 32-bit binary addresses in the Test class and Cache will behave according to those addresses.

LRUCache is created to be able to have a class that behaved like a cache. It is the cache using LRU policy as its replacement method. In its *constructor*, it takes the necessary calculations from cache class but additionally takes “n” which is the history reset frequency and its *add* method is controlling how to cache should take action against the addresses provided, similar to cache with random policy.

The only and major difference between Cache and LRUCache is one using random replacement and the other using LRU replacement when they have to replace a data from cache. Both will use write back, but the one who has to be deleted in an index will be decided differently. Cache class will randomly select an address in the cache and replace it with the new address whereas LRUCache will select the least recently used one and replace it. To be

keep track of the least recently used ones, LRU used the method of explained in the homework's PDF, the history caches. It simply follows the instructions in the PDF and obey according to the finite state machine diagram provided us in the PDF.

## How to run from command line?

We had hard times running the code so we would like to make a brief explanation of how to run the code.

Source files, addresses stream files and the log files are all included in the .zip we have provided to you. To run the project first of all you have to created .class files for each of the .java files. To accomplish this, following command is necessary:

**javac *.java**

*(*This command will create all. class files, which are necessary to run the simulation.)*

There are two options of running the cache simulation. One is running with arguments by providing arguments or without the arguments. If you do not provide any arguments to the program it will assume policy is random, associativity is 1, cache size is 512, block size is 4, history reset frequency is 100, address stream file is *addresses.txt*, detailed log file is *log.txt* and debug is 0. To be able to run with those default values, you only have to type:

**java Test**

If you would like to run simulation in specific situation, you should provide arguments. In this case, you should run the following command:

**java Test random 2 1024 16 100 addresses-2.txt log.txt 0**

*(*If you enter at least one argument, it will wait for you to provide all the arguments. You cannot specify some of the arguments and leave the others blank. You should either provide all the arguments or not provide any of the arguments at all.)*

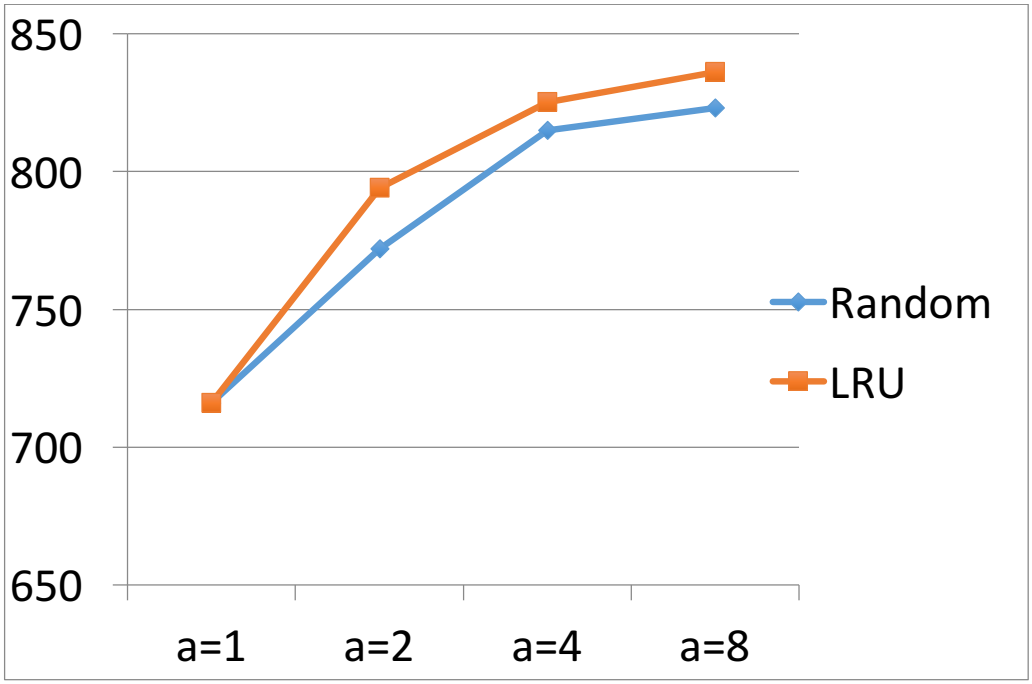
There are 7 arguments; code will wait to be given. These are respectively:

- *policy:* type of the cache; can be either lru or random
- *associavity*: An integer (<= 8) specifying the associativity of the cache
- *cache size:* Total size of the cache in bytes
- *block size:* Block size of the cache in bytes
- *history reset frequency:* <Integer>
- *file name:* Address stream file name
- *log file name:* Detailed log file name
- *debug:*1 to see initial states of cache at each of 100 steps, 0 not to see.

## Graphs and Results

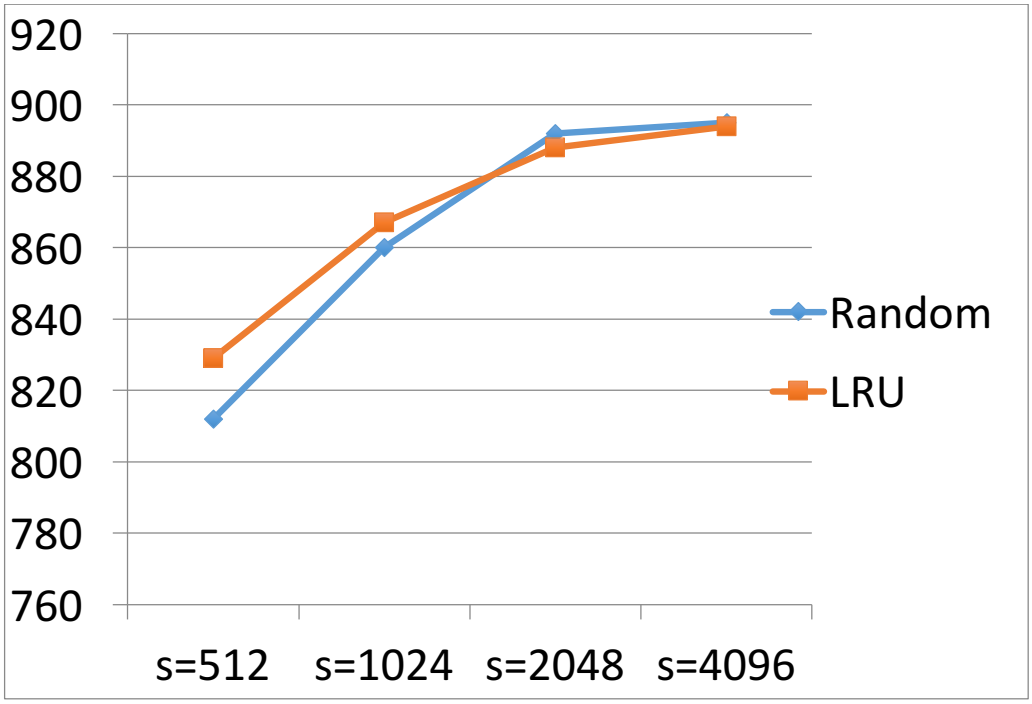
You can see graphs that we draw with datas, which obtained from our program, and explanations about these graphs in below lines.

**Graph 1 (Number of hits-Associativity)**



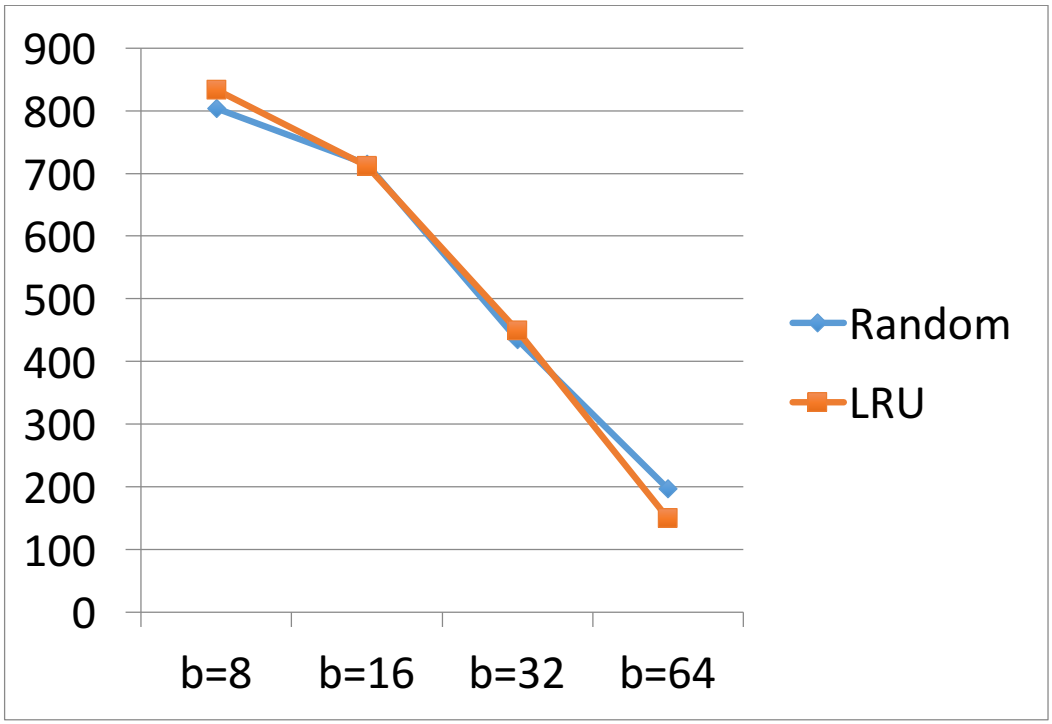
The graph illustrates trend for number of hits with increasing associativity while keeping cache and block sizes constant (s=512 b=4). As we expected, while associativity is increasing, number of hits is also increasing in graph. Because, if associativity will increase, then you can put more data in same index and reduce conflict misses. Therefore, number of cache hits should increase.

**Graph 2 (Number of hits-Cache Size)**




The graph illustrates trend for number of hits with increasing cache size while keeping associativity and block size constant (a=8 b=4). As we expected, while cache size is increasing, number of hits is also increasing in graph. Because, if cache size will increase, then you can keep more data in the cache and reduce conflict misses. Therefore, number of cache hits should increase.

**Graph 3 (Number of hits-Block Size)**




The graph illustrates trend for number of hits with increasing block size while keeping associativity and cache size constant (s=1024 a=8). Normally, while block size is increasing number of hits should increase, but here addresses were not contiguous because of that cache did not work as we expected. For this case, we wrote a new addresses txt, you can see datas, we obtained from that one, in last page of the ***"Results"*** file. You will realize, if block size will increase, then number of hits also increasing.

Thank you for your time…

Kutay Demireren
Engin Can Kurt