Project Report Lab 1

General Overview:

For the implementation of this project, we used the source code provided by the teachers as a base for all tasks.
This simulator emulates a cache so, its first step is to divide the memory address into tag, index, and offset. The offset size is the same in all the tasks and, because each block has 16 4-byte words and the cache is byte-addressable, has 6 bits. The index size (and consequentially the tag size) is variable and will be explained further in the description of each task.
Secondly, the index is used to find the respective cache line, the tag and the line's valid bit to check for a hit and the offset to access the desired byte in the block.
Also common to the three tasks is the write policy. The project implements a write-back process where new data is written to the cache and the dirty bit is switched to signal that the cache is inconsistent with the data on the DRAM and the DRAM is only updated when the block is replaced in the cache. In case of a write miss, the program uses a write allocate logic, in which it retrieves the data to the cache and writes the desired data onto it.

4.1. Directly-Mapped L1 Cache:

To implement the first task of our project, we altered the cache structure to allow it to have multiple lines instead of just one so it could simulate a proper and realistic cache. The number of lines (L1_NUM_LINES = 256) is defined as an extra variable in the cache.h file, even though it could be calculated as L1_SIZE / BLOCK_SIZE, as we figured it would make the code more readable.
As for the actual code in the L1Cache file, we used shifts (left and right) to split addresses into usable information: 18 bits for the tag, 8 bits for the index, and 6 for the offset. Then, we use the calculated index to get the respective cache line and we proceed to check said line for the calculated tag and evaluate the access as a cache hit or miss, as per the source code.

4.2. Directly-Mapped L2 Cache:

In this task, the L1 cache is the same as the previous task. However, we added a second cache, L2, for the program to access in the case of a miss in the L1 cache. This cache has double the lines (L2_NUM_LINES = 512).

Because of this cache's structure, the shifts needed for retrieving the tag, index, and offset are different because the address now has 17 tag bits, 9 index bits, and 6 offset bits because an extra index bit is needed to map to all the cache's lines.

With this implementation, the program only accesses the DRAM in case of a miss in both caches.

4.3. 2-Way Associate L2 Cache:

For the last task, we changed the logic of the L2 cache so, instead of being comprised of lines, the cache has an array of sets with 2 lines inside of each set (hence the term 2-way associative). To maintain the initial size of the L2 cache unchanged, we defined L2_NUM_SETS as 256 and SET_NUM_LINES as 2 in the cache.h file. With this alteration the addresses are split in the same way as in the L1 Cache for the L2 Cache, because the index maps to the set instead of the line and when inside the set, both of its lines are checked for a hit.
In the implementation, the new data structure for the cache's sets not only stores the array of lines but also the index of the set and an integer – oldest - that can either be 0 or 1 depending on the set's line that contains the LRU (last recently used) block.
Given the new cache logic, when we search for data in the L2 cache, we now have to check both lines of the set indicated by the index for a hit. In case of a miss, in the previous implementations we simply replaced the block but, in this third task we have to choose which block to remove from cache. This is achieved with the help of the "oldest" attribute and therefore, the LRU block is replaced and the set's "oldest" attribute is switched with a ternary operator.

Prof. David Valente
Project done by:
Carlota Domingos 107016
Joana Vaz 106078
Mariana Santana 106992