LAB REPORT:
CACHE SIMULATOR

Rodrigo Dias 99552
Diogo Rodrigues 102848
Francisco Nael Salgado 103156

Introduction

The code developed for the present report attempts to emulate a memory hierarchy divided in 3 stages, as per the assignment: a directly-mapped L1 cache; directly-mapped L1 and L2 caches; directly-mapped L1 cache and 2-way associative L2 cache. All the aforementioned stages feature a DRAM main memory which simply consists of a DRAM_SIZE array, and the access DRAM function writes/reads entire blocks, not words.

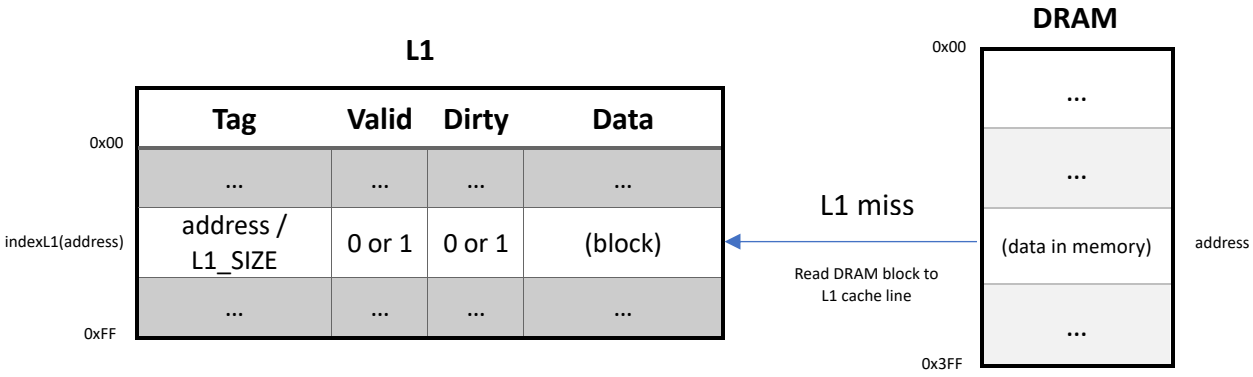
All memory is byte-addressable, and the caches utilize a write-back policy and LRU replacement. The data read/write processes correspond to words (32 bits = 4 bytes), not entire blocks (16 words each).
In all stages, the read/write interfaces utilize solely the function accessL1 with the corresponding mode argument, from which different behaviour is executed based on the exercise.

Since each block corresponds to 16 words = 64 bytes = 2^6 bytes, the 6 least significant bits in the address will correspond to the offset bits in both L1 and L2. Furthermore, since the DRAM has a size of 1024 * BLOCK_SIZE = 2^{16} bytes, we can deduce that, although we're using 32 bit addresses, only the 16 least significant bits should be used, in order to fit the DRAM.

Each cache in the hierarchy corresponds to a Cache type, which contains an array of CacheLine types (for 4.3, the L2 terminology changes due to associativity, to be explained later). The CacheLine objects contain fields relevant to the line such as the tag, validity and dirtiness, and obviously, a BLOCK_SIZE array for the data itself.

4.1

Since L1 has a size of 256 * BLOCK_SIZE, we deduce 256 = 2^8 cache lines ⇔ 8 bits for index. Therefore, 32 – 8 – 6 = 18 bits for the tag.
For the first exercise, accessL1 function execution goes as follows:



For a L1 miss, if the dirty field is 1, we first write the stored block in L1 back to memory. To get the address of the dirty block, we take its tag and multiply by L1_SIZE, to shift it 14 bits to the left. We then add to it the multiplication of the index (same as the dirty block) with BLOCK_SIZE to shift the index 6 bits (offset bits). Since we want to get the entire block from memory, the offset is zero. Taking this "dirty" address, we use accessDRAM to write the block to memory, then set the dirty field to 0.
For a L1 hit, we go straight to the next step.

Then, the word in the data field is written/read to/from the L1 cache line, with an offset specified by the address, and the write/read time is added to the global time. For a write operation, we set the dirty field to 1.

4.2

Since L2 has a different size to L1, the address is interpreted differently as well. L2 has a size of 2^9 blocks ⇔ 9 bits of index (instead of L1's 8).
In adding the L2 cache, the accessL1 function's only changes are in the access DRAM calls, which change to the accessL2 function (with the same arguments), since L2 comes before DRAM in the hierarchy. An accessL2 call means an L1 miss.
The accessL2 function operates almost exactly the same as the accessL1 function in the first exercise. One obvious change is in "calculating" the dirty address, since the index bits increase by one). Another change is that the entire block is read from L2 (as opposed to accessL1, where only the requested word was written/read), since accessL2 is not used to read words themselves, but to allocate from and provide entire blocks back to L1. Finally, L2 read/write times are added to global time instead of L1's.

4.3

For a 2-way associative L2 cache, we use a different datatype, AssociativeCache, which differs in that it consists of an array of sets, and each set consists of a 2-long array of lines (associativity ways). The lines use a new type, AssociativeCacheLine, which adds a field for the time the line has last been read/written (for LRU purposes).
The number of sets, that is, the number of indices in the sets array, is half that of the directly-mapped L2 cache in the 4.2 exercise. Therefore, we have one less bit for the index in the L2 address interpretation (indexL2 function): $\frac{2^9\ lines}{2\ ways}$ sets ⇔ 9 – 1 = 8 index bits.
The changes occur also in the function accessL2, obviously. For an index in the L2 cache, we now search the N=2 ways for the address tag. If there's a match and the line is valid, we have an L2 hit.
However, for each iteration of the "search", we also check for the first invalid way (if such exists), and we keep the way with the smallest last_used parameter, that is, the least recently used way.
Thus, if no match is found, an L2 miss occurs, and we need to identify which way to replace in the set. If we find an invalid way, such is prioritized. If not, the least recently used (LRU) way is replaced. Same as before, if the to-be-replaced way is dirty, we first write back the block to DRAM with the calculated dirty address, and then set the dirty field to 0.