

1 General Code Context

1.1 Address Decomposition

To compute the address decomposition we used the following code (example for L1 Cache, but similar logic for the rest):

```
tag = address / (L1_N_LINES * BLOCK_SIZE);
index = (address / BLOCK_SIZE) % L1_N_LINES;
offset = address % BLOCK_SIZE;
```

The division operation acts as a shift left because we are dealing with powers of 2, removing the least significant bits from the address. The multiplication operation, which acts as a shift right, can be used to undo the previous operation. The remainder operation helps us isolate the least significant bits.

So, to find the *tag*, we first need to remove the lower bits related to the *index* (L1_N_LINES) and *offset* (BLOCK_SIZE).

For the *index*, we start by shifting out the *offset* bits. Then, we apply the remainder operation using the number of lines, which gives us the *index* bits.

As for the *offset*, we use the remainder operation with the block size to obtain the *offset* bits.

1.2 Write-back Technique

To implement the write-back technique, we utilize the Dirty flag (example for L1 Cache, but similar logic for the rest):

```
if (L1[index].Dirty) {
    accessDRAM(L1[index].Tag * L1_SIZE + index * BLOCK_SIZE,
               L1[index].Data, MODE_WRITE);
}
```

When a write operation occurs, the **Dirty** flag gets set to 1, indicating that it has been modified. When a **Dirty** cache line is evicted from the cache (due to cache replacement or other reasons), the data is then written into the next level in the memory hierarchy (DRAM or L2 Cache, depending on the function and exercise). Finally, we reset the **Dirty** flag to 0 and, if possible, a new block is obtained from that same memory level, essentially swapping blocks with that level.

1.3 Constants and Cache struct

We introduced two new constants, **L1_N_LINES** and **L2_N_LINES**, which hold the respective number of lines for each cache. Additionally, within the **CacheLine** struct, we incorporated an array (**Data**) to facilitate the storage of cache data per line. The **CacheLineL2** struct for the 2-Way L2 Cache also includes a **Time** parameter, which is used for the LRU algorithm. We also defined a new constant for the associativity of the L2 Cache (**ASSOC_L2**) that can be changed.

2 Directly-Mapped L1 Cache

We start by initializing the cache's line flags in the `initCache` function.

In the `accessL1` function, we extract the tag, index and offset bits as described in [Address Decomposition](#). Then we determine if the cache access is a miss. This involves checking the line's validity (whether the `Valid` flag is set to 0, indicating that valid data doesn't exist in the line) and comparing tags (if they differ, we have a miss).

If it is indeed a miss, it checks whether the entry is marked as dirty. If so, it follows the write-back technique mentioned in [section 1.2](#), in this case writing back to the DRAM. After that, it makes sure the line's `Valid` flag is set to 1, resets the `Dirty` flag, as previously mentioned, and saves the tag in the line.

Finally, we proceed with the corresponding operation based on the mode. If it is a write operation, we set the `Dirty` flag to 1.

3 Directly-Mapped L2 Cache

In accordance with the provided instructions, we opted to repurpose the cache outlined in [section 2](#) to serve as our L1 Cache in the memory hierarchy.

In order to simplify the initialization process for both of the caches, we included both of them into the `initCaches` function.

Regarding the `accessL1` function, minimal alterations were made, primarily involving rerouting it to access the L2 Cache rather than directly interfacing with the DRAM in the event of a cache miss.

Since it wasn't specified in the statement, we opted for a non-inclusive non-exclusive (NINE) L2 Cache for the inclusion policy. We saw this as the easiest way to implement the caches, because there would be no need to check for back invalidation.

The implementation of the L2 Cache was a relatively straightforward process. We essentially duplicated the `accessL1` function, named it `accessL2` and, instead of re-routing to the L2 Cache, we directed it to the DRAM.

4 2-Way Associative L2 Cache

The key difference between this implementation and the one in [section 3](#) is the 2-Way associativity.

The `initCaches` function now initializes all of the flags for each line within each cache set. The only other big change was in the `accessL2` function, that now employs code for the 2-Way set-associative approach, meaning we have to take into account the associativity when calculating the tag and index (now has one less bit). The function now has to iterate all of the lines within the indexed set and test the `Valid` flag and tag for each one. If none of the lines are free (valid and have a tag that matches), then we have a miss.

We have a LRU (least recently used) policy for replacing lines within the set, and so we save, for every line, the `Time` parameter on every operation. Therefore, when a miss occurs, the program obtains the line with the lowest `Time` (least recently used) and then follows the same protocol as before for getting a new block from the DRAM. The write-back technique is also the [same as before](#).