1 General Code Context

1.1 Address Decomposition

To compute the address decomposition in every file we used the following code:

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

The division operation acts as a left shift, removiong lower bits from the address. The remainder operation helps us isolate the lower-order bits.

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

For the *index*, we start by subtracting 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 Policy

To implement the write-back policy, we utilize a the following parameter:

```
L1[index].Dirty = 0;
L1[index].Dirty = 1;
```

When the `Dirty` parameter is set to 1, it indicates that the information must be written to either the RAM or the L2 cache (depending on the function and exercise). After writing, we reset this parameter to 0 to prevent inadvertent duplicate writes. Conversely, we set it to 1 when a write operation has just been performed on the record.

1.3 Constants and Cache struct

We introduced two new constants, namely `L1_N_LINES` and `L2_N_LINES`, which hold the respective number of lines for each cache. Additionally, within the `Cache` structure, we incorporated an array to facilitate the storage of cache data.

2 Directly-Mapped L1 Cache

We start by initializing the cache and the RAM in the `init_cache` function. This entails setting all parameters to 0.

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 hit. This involves checking its validity (whether the valid parameter is set to 1, indicating data exists in the desired location) and comparing tags. If it is indeed a hit, we check what mode it is on (read or write). Then we proceed accordingly, marking the cache entry as dirty if the mode is to write (please refer for 1.2 for more information).

If the tags do not match or the line is still invalid, we are dealing with a miss. In that case, we check whether the entry is marked as dirty. If so, we write bock it to the RAM. Then we retrive data data from RAM and store it in the L1 cache, and proceed with either a read or write operation based on the mode. Again, if it is a write operation, we signal the entry as dirty. Lastly, we mark the cache entry as valid.

3 Directly-Mapped L2 Cache

In accordance with the provided instructions, we opted to repurpose the cache outlined in section 1 to serve as our L1 cache.

In order to simplify the initialization process for both the caches and the RAM, we consolidated these tasks into the `init_caches` function. Subsequently, for the setup of the L2 cache, we followed the same protocol as outlined for the L1 cache, detailed in section 1.

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

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 interact with the RAM. If additional clarity on the `accessL1` function is needed, the section above details how this funcion works.

4 2-Way L2 Cache

Blah