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 creating an array for the memory data, and our L1Cache. After that, the initCache function will set all the valid, dirty and tag bits, and data to 0. When acessing the L1Cache, the adress is splitted into tag, index, and offset. Using the obtained index, The acessL1 function will then verify the value of the valid bit and compare the obtained tag with the tag in the cache array. If the valid bit is set to 1 (there is data in the location we want to acess) and the tags are the same, the function will either read the data from the cache or write the data in the cache, setting the Dirty bit to 1.

If the valid bit is set to 0 (there is no data written) or the tags are not the same, the function will verify the value of the dirty bit. If the dirty bit is set to 1, the cache data is saved into memory.

Finally, the function will , again, read the data from the cache or write the data onto the cache, updating the dirty bit to either 0 or 1 (respectively), the tag to the given one and setting the valid bit to 1.

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