# 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 Policy

To implement the write-back policy, we utilize the `Dirty` flag:

```
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 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.

# 2 Directly-Mapped L1 Cache

We start by initializing the cache parameters 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 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 it back to the DRAM. Then we retrive data from the DRAM 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 2 to serve as our L1 cache.

In order to simplify the initialization process for both the caches and the DRAM, we consolidated these tasks into the `initCaches` function. Subsequently, for the setup of the L2 cache, we followed the same protocol as outlined for the L1 cache, detailed in section 2.

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 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 the DRAM.

# 4 2-Way L2 Cache

Blah