Where we begin

So we are starting at the point where we have a vector full of every custom struct/address in the trace file converted into binary from hex and broken down into the following four parts:

1) Set bits
2) Tag bits
3) Type of memory address (Store, Load and Modify)
4) Size

The Set bits and Tag bits are calculated dynamically by the program after being provided by command line arguments.

This plan will cover the DIRECT MAPPED CACHE ONLY.

What needs to happen first

Okay so first things first we take one of the data structs from the vector.

Then we need to check whether that set is already in the cache. We do this by taking the set bits from a single data structure in the vector that we are iterating through.

We iterate through the array checking the first element of every row and comparing them with the set bits taken as shown above.

If the set bits are in the cache already:

In this case we then return the row to the main function as we will need to compare the tag bits requested with the tag bits of the second element of this row.

If the set bits are there but the tag bits are different:

After checking the second element of the cache set the tag bits don't match. Oh no! This is a *cache miss* as the requested tag bits (or the block that the CPU wants) are not in the cache.

As this is a DMC and we cant have identical set bits in the cache together so we need to remove the tag bits of this particular line and replace them with the tag bits that we were comparing to.

So we take the row that has the correct set bits and change the second element to the new set bits. We then move this vector to the top of the 2d vector or the first vector in the vector of vectors as this is our abstraction for Most Recently Used (MRU).

If the set bits are there and the tag bits are the same:

We report this back as a *cache hit* and the vector that was found is moved to the "top" of the cache as it is MRU.

If the set bits are *not* in the cache already:

In this case it is relatively simple, we return a value (maybe a None or something else) to denote that the set (or collection of blocks) required is not in the cache, and as such this request from the CPU results in a *cache miss*.

All of the following steps MUST be preceded by a function that *ensures the cache is not beyond its row or cache set limit* as determined by checking the length of the 2d vector and ensuring it only contains as many vectors as cache sets.

If the cache is full:

To establish this we check all of the first row in the 2d array if all of the elements do not equal the default "empty" string we declare the cache full. We then determine which element was the last to be used and remove it (following the Least Recently Used or LRU policy). We then place this new element into the 2d vector in the bottom or last row (as the last element).

If the cache is *not* full:

To establish this we check every first element in every row of the 2d array, if any of the elements equal empty then the cache is not full. We then place this in the first "empty" row we find.

The ArrayRepresentationOfCache Structure

So we have assumed that our custom data structure has to contain the following elements:

1) The amount of rows or cache sets(taken and processed from command line)
2) The amount of columns or cache lines (taken and processed from command line)
3) The 2d vector (dynamically sized by the above two values)

The Methods

Here we detail everything we think we need this data structure to do through individual functions.

The “Constructor”

First we have a constructor equivalent that is used to initialise the values in the implementation of the data structure. This takes the values provided as arguments to this function and assigns it to the data structure that you are initialising, even creating a 2d vector based on the cache sets and cache lines.

is_set_in_the_cache

So first we need to establish whether the set sent from the CPU is in the cache. This function will do that and return the index of the vector if it is, and a None value if it is not. If the set is not in the cache it should iterate the *cache miss* store.

check_if_tag_bits_match

If the set is in the cache we now need to check if the tag bits are the same so we will use this function in conjunction with the above to determine if a set is in the cache and we can check if the tag bits match. If it matches it returns a true or a false boolean value.

If it is true it should also iterate the *cache hits* store, and if it is false it should add one to the *cache misses* store.

As we will have established by this point that the cache set is in the cache, the false value will also count as a *cache eviction* as in a DMC every cache set can only hold one cache block and therefore we have to evict one to make space for the new one.

is_cache_full

This is a relatively simple function that simply checks if any of the row's first elements have an “empty” String value to tell us if the cache is full or not. If it finds an “empty” string the cache is not full, if it does not the cache is full. The return value will be boolean.

is_cache_correct_size

This is another simple function to tell us if the cache is the correct size. This simply takes the value of the cache sets and compares it to the length of the vector. Something has gone drastically wrong if this ever returns a false boolean outside of insert into cache.

insert_into_cache

Every time something is inserted, modified or loaded it will be placed at the top of the cache. This means moving everything else one step down and removing a row if it is beyond the correct size of the cache.

But this will take arguments that allow it to push the vector into the cache and pop the last element if the above is the case. This will iterate the *cache evictions* store.

The flow of the program

```
Takes binary data struct from vector

Checks if set bits are in the cache

If they are:

    Check if tag bits match

    If they do:

        Add one to cache hit and move vector to “top” of 2d vector

    If they don't:

    Add one to cache miss and edit vectors second element to be tag bits and move
    vector to “top” of 2d vector

If they aren’t:

    Check if cache is full

    If it is:

        Insert vector at top of cache with push function and add one to cache miss
        Remove last vector and add one to cache eviction

    If it isn’t:

        Push vector onto top of 2d vectors and add one to cache miss
```