THE UNIVERSITY OF TEXAS AT ARLINGTON

A Project Report

On

Cache Controller Design

Instructor:
Dr. Jason Losh

By-
Deepti Saxena (1001937586)

Shrutika Chaudhari (1002026761)

DEPARTMENT OF ELECTRICAL ENGINEERING
FALL 2022

INDEX

INTRODUCTION:

Cache is the data storage layer, which caches the data from memory, which is frequently used, so that future requests for that data are served up faster than is possible by accessing the primary data storage location i.e memory. A good cache results in a high hit ratio which means tag and line bits are matched i.e data is present in cache when fetched. When data is not present in cache then it is a cache miss.

Project Overview

The goal of this project is to determine the best architecture for a cache controller that interfaces to a 32-bit microprocessor with a 32-bit data bus. While the microprocessor is general purpose in design and performs a number of functions, it is desired to speed up certain signal processing functions, such as a fast Fourier transform (FFT) routine.

Specifications

➔ The project allows 4GiB of SDRAM to be interfaced with a 32-bit data bus (arranged as 230 x 32-bits) and the cache is limited to 256 KiB in size. The size of the FFT is 32768 points(512KiB) in normal operation.

➔ The miss penalty used in the project is 60 ns (for first memory access) and 17ns for subsequent accesses in the same SDRAM word line and the cache hit time is 1 ns.

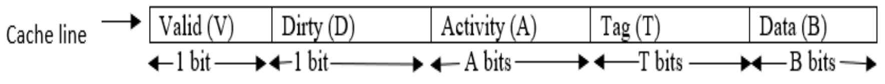
➔ In this Project we have used N ways as 1,2,4,8,16 cache lines (L) 65536, write strategies used are write-back allocate, write-through allocate, and write-through non-allocate, and cache line block replacement strategy used is LRU

➔ The Radix2FFT () is used to evaluate the best cache performance among the write strategies (write-back, write-through allocate, or write-through non-allocate).

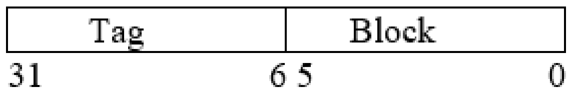
➢ Types of Cache:

1) Fully Associative Cache

- A fully associative cache allows data to be stored in any cache block. When data is retrieved from memory, it can be stored in any unused cache block.
- If all the blocks are already in use, it replaces the least recently used data with the new data in cache.



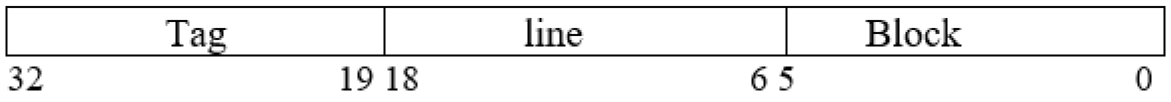
Address:



2) Direct Mapped Cache

In direct mapping, a particular block of main memory can map to only one particular line of the cache.

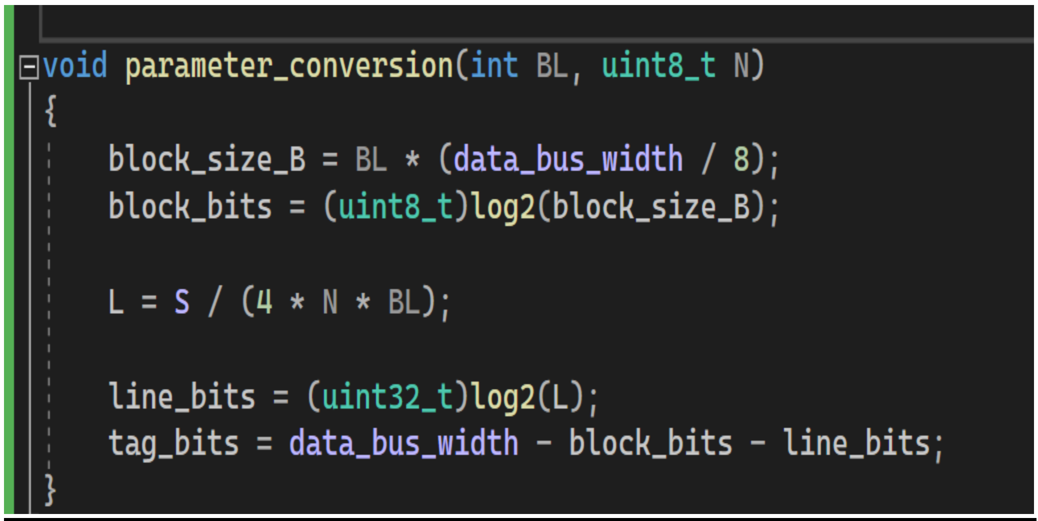
Address



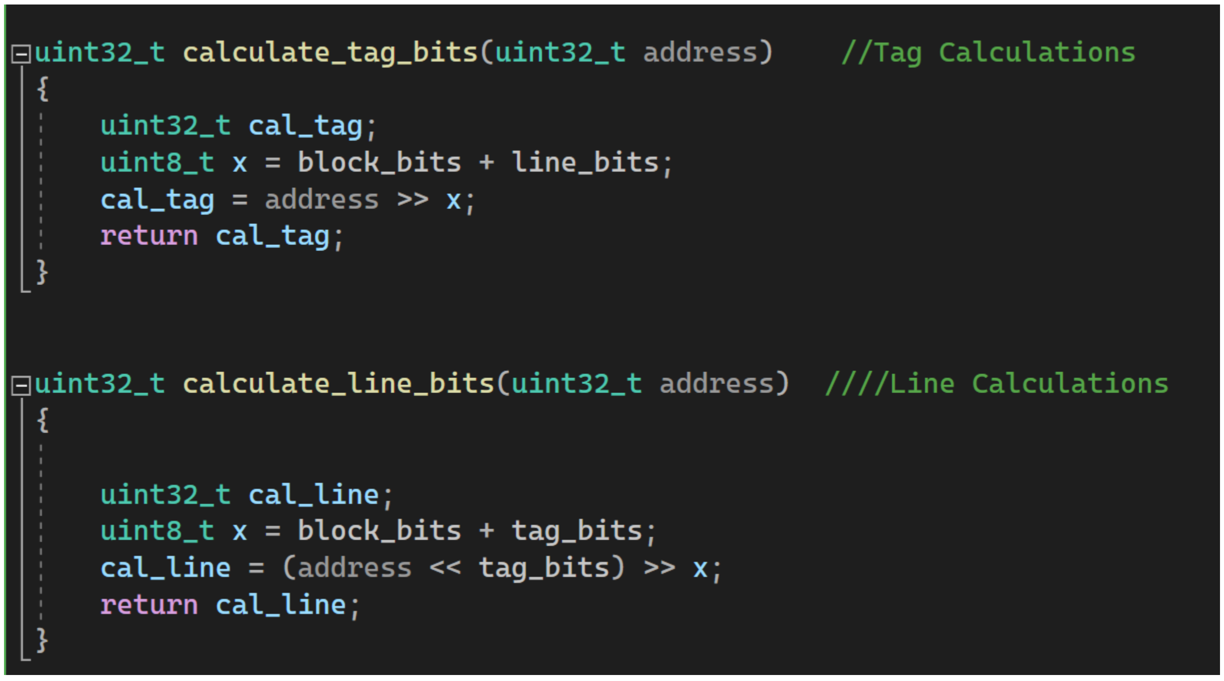
3) Set Associative Cache

- Set-associative cache is a hybrid of direct-mapped and fully associative cache.
- A set-associative cache is represented by a L x N matrix. The cache is divided into ‘L’ cache lines, each with ‘N’ sets.

➢ Calculations for Cache line and tag bits from address :



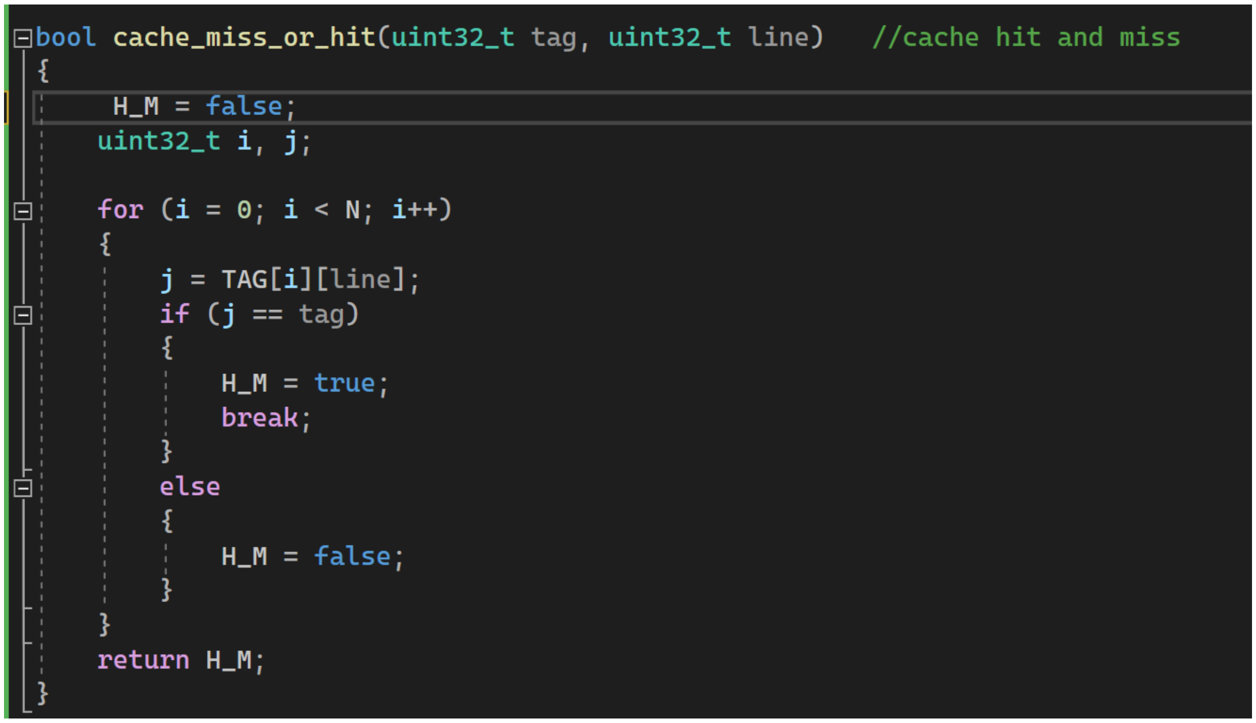
```
void parameter_conversion(int BL, uint8_t N)
{
    block_size_B = BL * (data_bus_width / 8);
    block_bits = (uint8_t)log2(block_size_B);

    L = S / (4 * N * BL);

    line_bits = (uint32_t)log2(L);
    tag_bits = data_bus_width - block_bits - line_bits;
}
```



```
uint32_t calculate_tag_bits(uint32_t address)    //Tag Calculations
{
    uint32_t cal_tag;
    uint8_t x = block_bits + line_bits;
    cal_tag = address >> x;
    return cal_tag;
}


uint32_t calculate_line_bits(uint32_t address)  ////Line Calculations
{

    uint32_t cal_line;
    uint8_t x = block_bits + tag_bits;
    cal_line = (address << tag_bits) >> x;
    return cal_line;
}
```

➢ <u>Cache Hit and Miss</u>

- When the Microprocessor wants to read the data from memory, it will check the data first in cache rather than going to memory (as it increases the access time to read data). If Microprocessor finds the data in cache, it is called Cache Hit.
- If the Microprocessor did not find the data in cache, it is called Cache Miss. To Improve the Cache performance the cache hit ratio should be higher than the cache miss ratio.



```
bool cache_miss_or_hit(uint32_t tag, uint32_t line)    //cache hit and miss
{
     H_M = false;
    uint32_t i, j;

    for (i = 0; i < N; i++)
    {
        j = TAG[i][line];
        if (j == tag)
        {
            H_M = true;
            break;
        }
        else
        {
            H_M = false;
        }
    }
    return H_M;
}
```

➢ <u>Replacement Strategy</u>

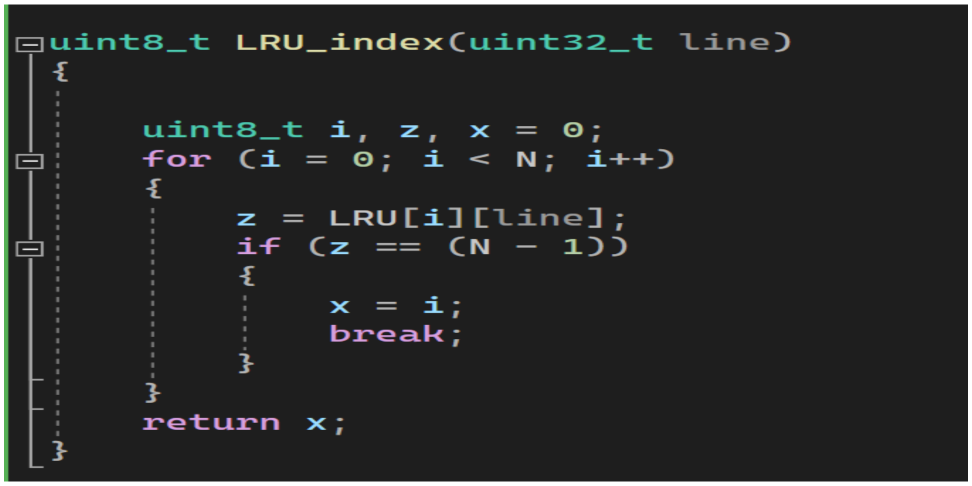
There are Three replacement Strategy

1) Least Recently Used (LRU)
2) Round Robin (RR)
3) Random Replacement Policy

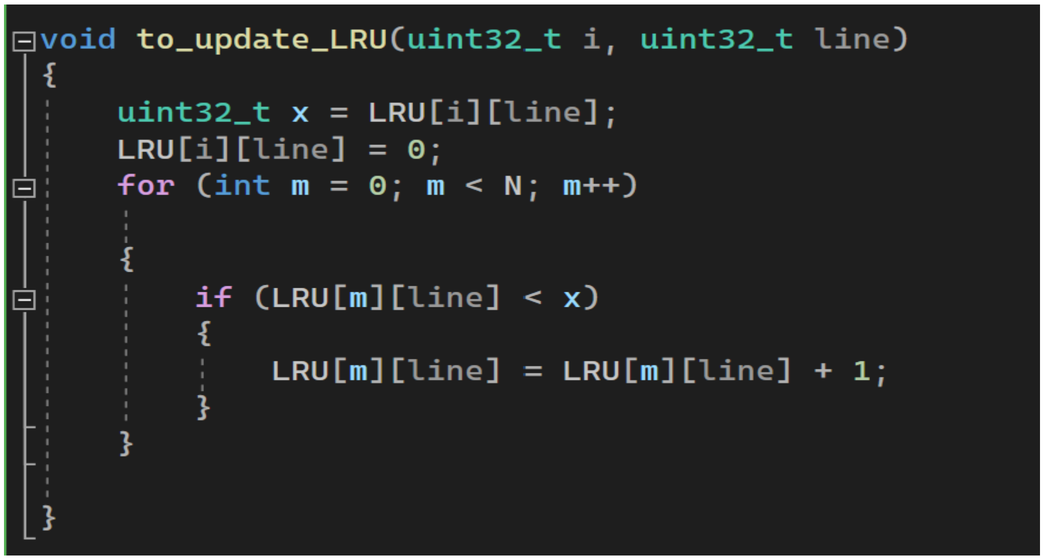
In this Project we are using the Least Recently Used Strategy for Cache:

➢ **<u>Least Recently Used (LRU) :</u>**

- In LRU when the cache is full, it defines the policy for replacing the oldest unused block from the cache to make room for the new block from memory.
- Anytime we read or write a block we need to update the LRU for the line.
- The two logic blocks implemented for LRU. The first one is to search the index of the line and second block is to update the LRU.



```
uint8_t LRU_index(uint32_t line)
{

    uint8_t i, z, x = 0;
    for (i = 0; i < N; i++)
    {
        z = LRU[i][line];
        if (z == (N - 1))
        {
            x = i;
            break;
        }
    }
    return x;
}
```

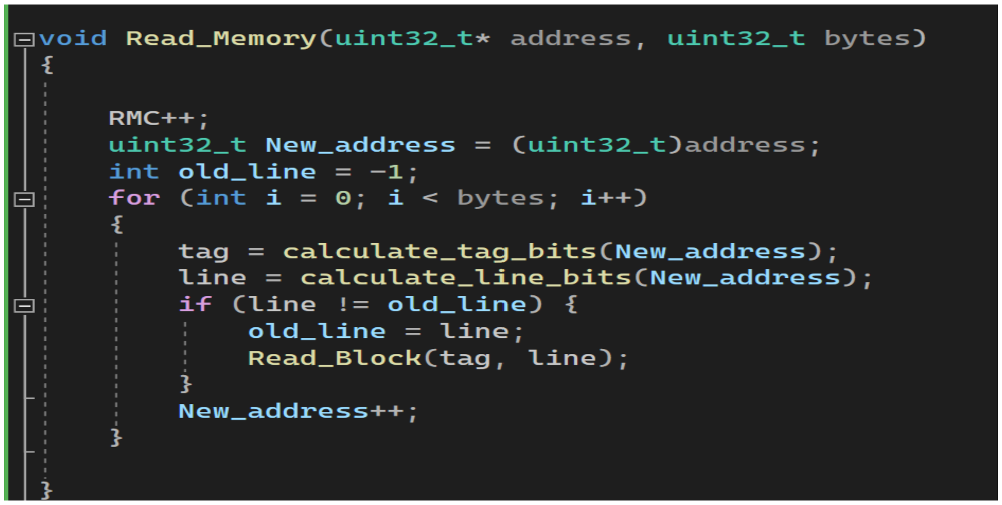


```
void to_update_LRU(uint32_t i, uint32_t line)
{
    uint32_t x = LRU[i][line];
    LRU[i][line] = 0;
    for (int m = 0; m < N; m++)

    {

        if (LRU[m][line] < x)
        {
            LRU[m][line] = LRU[m][line] + 1;
        }
    }

}
```

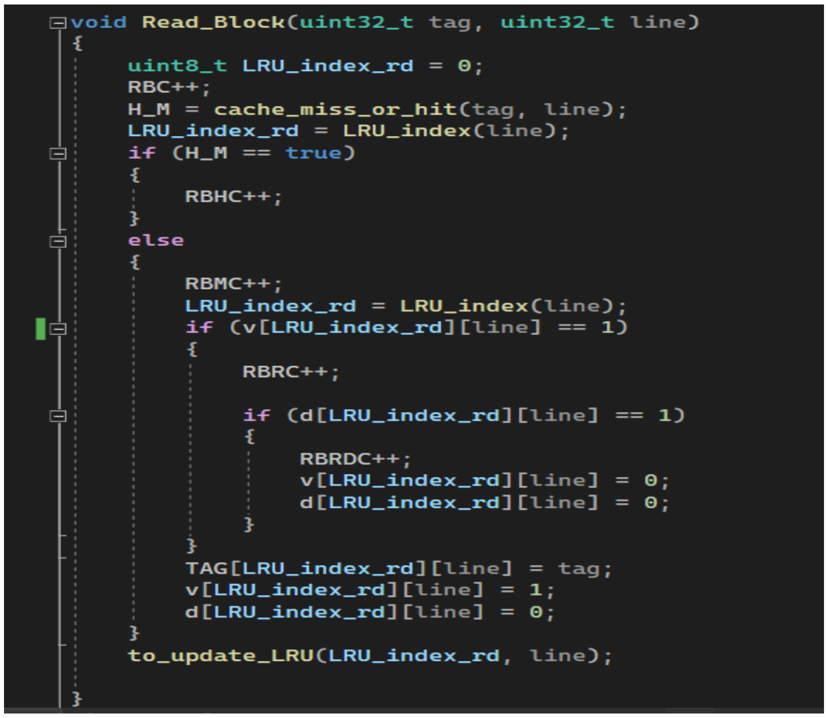
➢ <u>**Read Memory and Read block:**</u>

- In the read memory, we are calculating the line and tag bits from obtained address of the i th index.

- Then in read block, if cache hit has occurred then the read block counter will increase otherwise it will check for valid and dirty bit to replace the block and every time the LRU will be updated.



```
void Read_Memory(uint32_t* address, uint32_t bytes)
{

    RMC++;
    uint32_t New_address = (uint32_t)address;
    int old_line = -1;
    for (int i = 0; i < bytes; i++)
    {
        tag = calculate_tag_bits(New_address);
        line = calculate_line_bits(New_address);
        if (line != old_line) {
            old_line = line;
            Read_Block(tag, line);
        }
        New_address++;
    }

}
```



```
void Read_Block(uint32_t tag, uint32_t line)
{
    uint8_t LRU_index_rd = 0;
    RBC++;
    H_M = cache_miss_or_hit(tag, line);
    LRU_index_rd = LRU_index(line);
    if (H_M == true)
    {
        RBHC++;
    }
    else
    {
        RBMC++;
        LRU_index_rd = LRU_index(line);
        if (v[LRU_index_rd][line] == 1)
        {
            RBRC++;

            if (d[LRU_index_rd][line] == 1)
            {
                RBRDC++;
                v[LRU_index_rd][line] = 0;
                d[LRU_index_rd][line] = 0;
            }
        }
        TAG[LRU_index_rd][line] = tag;
        v[LRU_index_rd][line] = 1;
        d[LRU_index_rd][line] = 0;
    }
    to_update_LRU(LRU_index_rd, line);
}
```

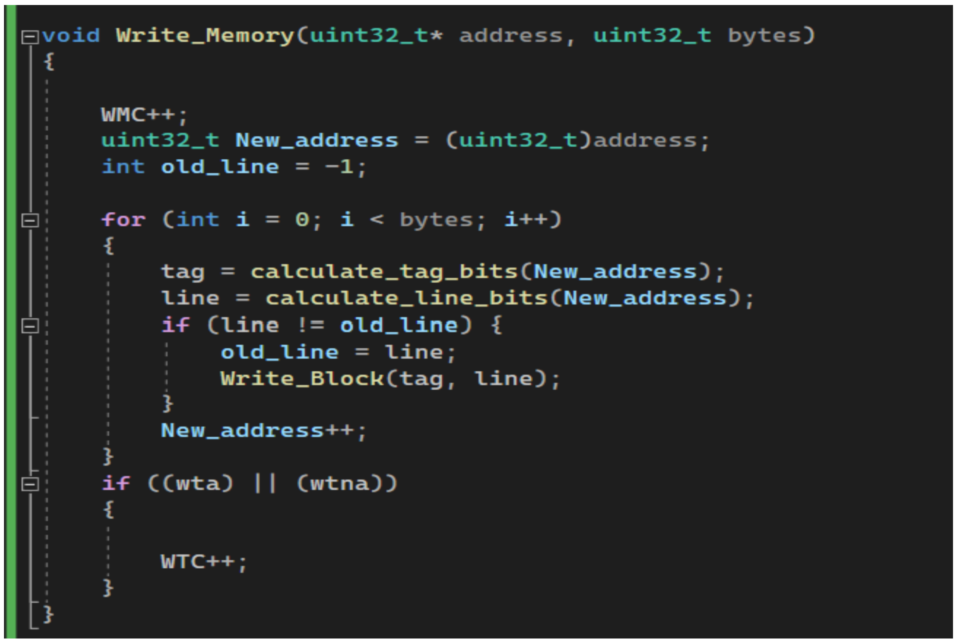
➢ Write Memory and Write block:

- The write memory function is accessed for the write back, write allocate and write through non allocate strategy. For all three the line and tag bits are calculated and based on that the write block function has called.

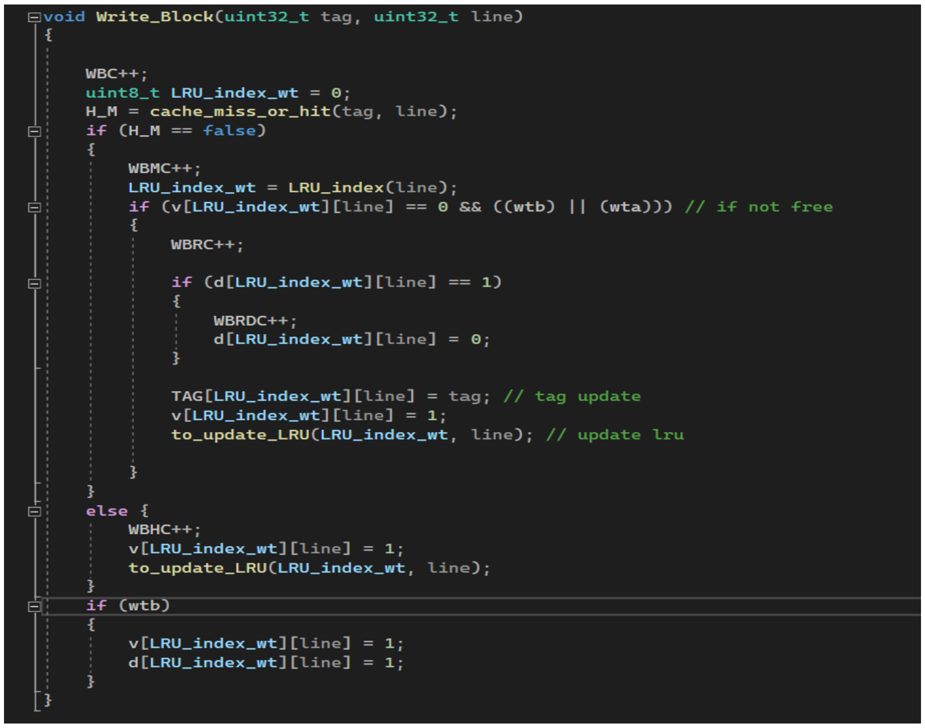
- In the write block, if cache hit has occurred then the write memory hit count has been incremented otherwise it will check for valid and dirty bit to replace ethe block in memory.

- For the write through allocate and write through non allocate, every time it accesses the memory the write through counter has been incremented.

- And for every condition LRU update will be there.



```
void Write_Memory(uint32_t* address, uint32_t bytes)
{

    WMC++;
    uint32_t New_address = (uint32_t)address;
    int old_line = -1;

    for (int i = 0; i < bytes; i++)
    {
        tag = calculate_tag_bits(New_address);
        line = calculate_line_bits(New_address);
        if (line != old_line) {
            old_line = line;
            Write_Block(tag, line);
        }
        New_address++;
    }
    if ((wta) || (wtna))
    {

        WTC++;
    }
}
```



```
void Write_Block(uint32_t tag, uint32_t line)
{

    WBC++;
    uint8_t LRU_index_wt = 0;
    H_M = cache_miss_or_hit(tag, line);
    if (H_M == false)
    {
        WBMC++;
        LRU_index_wt = LRU_index(line);
        if (v[LRU_index_wt][line] == 0 && ((wtb) || (wta))) // if not free
        {
            WBRC++;

            if (d[LRU_index_wt][line] == 1)
            {
                WBRDC++;
                d[LRU_index_wt][line] = 0;
            }

            TAG[LRU_index_wt][line] = tag; // tag update
            v[LRU_index_wt][line] = 1;
            to_update_LRU(LRU_index_wt, line); // update lru

        }
    }
    else {
        WBHC++;
        v[LRU_index_wt][line] = 1;
        to_update_LRU(LRU_index_wt, line);
    }
    if (wtb)
    {
        v[LRU_index_wt][line] = 1;
        d[LRU_index_wt][line] = 1;
    }
}
```

➢ **<u>Cache write strategies:</u>**

1. **<u>Write back:</u>**
    - In this strategy the data is updated only in the cache. If the data is present in the cache, then the hit occurs. In the case of miss, it will copy the data from memory to cache. In the strategy the memory write will not be there.

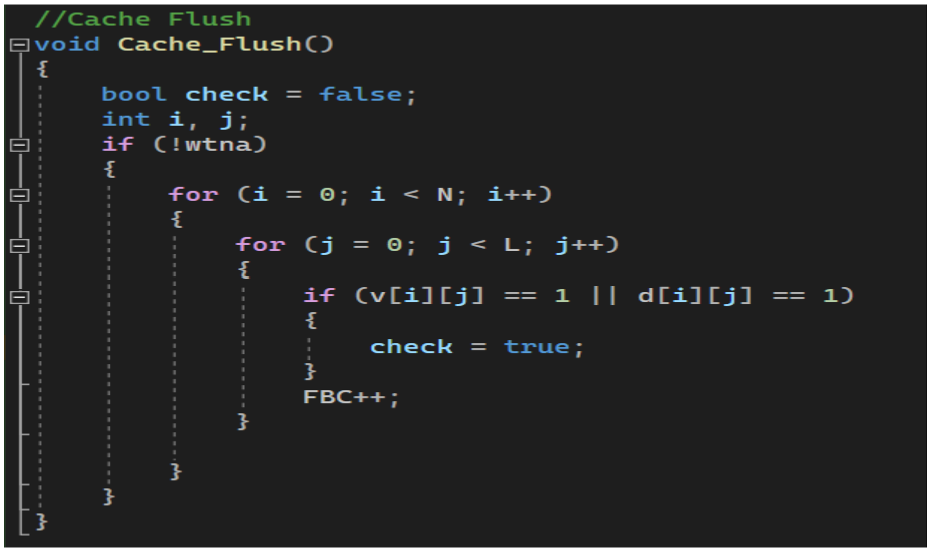
2. **<u>Write through allocate</u>**
    - In this strategy, the data is written into the memory and then updated into the cache simultaneously, regardless of hit or miss.
    - In this case, if write misses then the cache will read the data from memory i.e update the data into cache and then retry the write

3. **<u>Write through non allocate :</u>**
    - If hit occurs, then the data will update into cache but for miss it will directly go to memory and process the further task in memory.

➢ **<u>Cache flush logic:</u>**

- Cache flush logic works for total number of ways to each and every line calculated at that particular write strategy and BL number.

- For every line number, if valid and dirty bits are set then only the flush back counter will increment.



```
//Cache Flush
void Cache_Flush()
{
    bool check = false;
    int i, j;
    if (!wtna)
    {
        for (i = 0; i < N; i++)
        {
            for (j = 0; j < L; j++)
            {
                if (v[i][j] == 1 || d[i][j] == 1)
                {
                    check = true;
                }
                FBC++;
            }

        }
    }
}
```

Results

1. **Test container results:**

In the test container, the logic for memory accesses while read and write the memory and block replacement has been tested.

2. **Using FFT:**

The FFT routine is generally used to speed the processor system. Here, the radix2FFT function was used to calculate the largest hit in performance and based on that total access time was calculated. And it is observed that at BL = 8, N = 1 has the best results for hits.

Time calculation Formulae:

1. Read time = (RBHC*1nsec) +(RBMC *(60nsec+(BL-1) *17nsec)) +RBRDC*(60nsec+(BL-1)*17nsec)

2. Write time (WB) = WBHC*1nsec +(WBMC*(60nsec+(BL-1) *17nsec)) + WBRDC*(60nsec+(BL-1) *17nsec)
3. Write time (WTA) = WTC *60nsec + WBHC*1nsec +WBMC*(60nsec+(BL-1) *17nsec)
4. Write time (WTNA) = WTC *60nsec+WBHC*1nsec
5. Total time for memory = Read time + Write time (WB/WTA/WTNA) + FBC*60nsec

6. Write block Hit ratio = (WBHC/(WBHC+WBMC)
7. Read block Hit ratio = (RBHC/(RBHC+RBMC))
8. Total Hit Ratio = Read block Hit ratio + Write block Hit ratio
9. Total Miss Ratio = (1 - Read block Hit ratio) + (1 - Write block Hit ratio)

Reference :

1. https://www.cukashmir.ac.in/cukashmir/User_Files/imagefile/DIT/StudyMaterial/OperatingSystemBTech/BTechCSE_3_Rizwana_BTCS304_unit3_B.pdf