Batch: B2 Roll No.: 16010124107

Experiment / assignment / tutorial No.8

TITLE : Implementation of Cache Mapping Techniques.

AIM: To study and implement concept of various mapping techniques designed for cache memory.

Expected OUTCOME of Experiment: (Mention CO/CO's attained here)

Learn and evaluate memory organization and cache structure.

Books/ Journals/ Websites referred:

1. Carl Hamacher, Zvonko Vranesic and Safwat Zaky, "Computer Organization", Fifth Edition, TataMcGraw-Hill.
2. Dr. M. Usha, T. S. Srikanth, "Computer System Architecture and Organization", First Edition, Wiley-India.

Pre Lab/ Prior Concepts:

CODE:

DIRECT MAPPING:

```
#include <iostream>
#include <vector>
using namespace std;

struct CacheLine {
    int tag;
    bool valid;
    int data;
};
```

```
int main() {
    int cacheSize = 4;
    int blockSize = 1;
    int memorySize = 16;

    vector<CacheLine> cache(cacheSize, {-1, false, -1});

    int addresses[] = {0, 1, 2, 3, 4, 1, 8, 9, 0, 12};
    int n = sizeof(addresses) / sizeof(addresses[0]);

    cout << "Direct Mapped Cache Simulation\n";
    cout << "--------------------------------\n";

    for (int i = 0; i < n; i++) {
        int address = addresses[i];
        int index = (address / blockSize) % cacheSize;
        int tag = (address / blockSize) / cacheSize;

        cout << "Accessing address " << address << ": ";

        if (cache[index].valid && cache[index].tag == tag) {
            cout << "HIT in cache line " << index << endl;
        }
        else {
            cout << "MISS – Loading block into cache line " << index <<
endl;
            cache[index].valid = true;
            cache[index].tag = tag;
            cache[index].data = address;
        }
    }
    return 0;
}
```

OUTPUT:

```
PS C:\Users\syeda\OneDrive\Desktop\personal> cd "c:\Users\syeda\
g++ exp8.cpp -o exp8 } ; if ($?) { .\exp8 }
Direct Mapped Cache Simulation
--------------------------------
Accessing address 0: MISS: Loading block into cache line 0
Accessing address 1: MISS: Loading block into cache line 1
Accessing address 2: MISS: Loading block into cache line 2
Accessing address 3: MISS: Loading block into cache line 3
Accessing address 4: MISS: Loading block into cache line 0
Accessing address 1: HIT in cache line 1
Accessing address 8: MISS: Loading block into cache line 0
Accessing address 9: MISS: Loading block into cache line 1
Accessing address 0: MISS: Loading block into cache line 0
Accessing address 12: MISS: Loading block into cache line 0
PS C:\Users\syeda\OneDrive\Desktop\personal>
```

ASSOCIATIVE:

```cpp
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

struct CacheBlock {
    int tag;
    bool valid;
    int data;
};

int main() {
    int cacheSize = 4;
    int memorySize = 16;
    vector<CacheBlock> cache(cacheSize, {-1, false, -1});
    queue<int> fifo;
    int addresses[] = {0, 1, 2, 3, 4, 1, 8, 9, 0, 12};
    int n = sizeof(addresses) / sizeof(addresses[0]);
```

```
    cout << "\nFully Associative Cache Simulation\n";
    cout << "-----------------------------------\n";

    for (int i = 0; i < n; i++) {
        int address = addresses[i];
        int tag = address;
        bool hit = false;

        for (int j = 0; j < cacheSize; j++) {
            if (cache[j].valid && cache[j].tag == tag) {
                cout << "Accessing address " << address << ": HIT in
cache block " << j << endl;
                hit = true;
                break;
            }
        }

        if (!hit) {
            cout << "Accessing address " << address << ": MISS: Loading
into cache\n";

            if (fifo.size() == cacheSize) {
                int replaceIndex = fifo.front();
                fifo.pop();
                cache[replaceIndex].tag = tag;
                cache[replaceIndex].data = address;
                fifo.push(replaceIndex);
            } else {
                for (int j = 0; j < cacheSize; j++) {
                    if (!cache[j].valid) {
                        cache[j].valid = true;
                        cache[j].tag = tag;
                        cache[j].data = address;
                        fifo.push(j);
                        break;
                    }
                }
```

```
            }
        }
    }
return 0;
}
```

OUTPUT:

```
g++ exp8.cpp -o exp8 } ; if ($?) { .\exp8 }

Fully Associative Cache Simulation
-----------------------------------
Accessing address 0: MISS: Loading into cache
Accessing address 1: MISS: Loading into cache
Accessing address 2: MISS: Loading into cache
Accessing address 3: MISS: Loading into cache
Accessing address 4: MISS: Loading into cache
Accessing address 1: HIT in cache block 1
Accessing address 8: MISS: Loading into cache
Accessing address 9: MISS: Loading into cache
Accessing address 0: MISS: Loading into cache
Accessing address 12: MISS: Loading into cache
PS C:\Users\syeda\OneDrive\Desktop\personal>
```

SET ASSOCIATIVE
CODE:

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

struct CacheBlock {
    int tag;
    bool valid;
    int data;
```

```
};

int main() {
    int cacheSize = 4;          // total blocks
    int ways = 2;               // 2-way associative
    int sets = cacheSize / ways;

    vector<vector<CacheBlock>> cache(sets, vector<CacheBlock>(ways, {-1,
false, -1}));
    vector<queue<int>> fifo(sets);   // FIFO per set

    int addresses[] = {0, 1, 2, 3, 4, 1, 8, 9, 0, 12};
    int n = sizeof(addresses) / sizeof(addresses[0]);

    cout << "\n2-Way Set Associative Cache Simulation\n";
    cout << "--------------------------------------\n";

    for (int i = 0; i < n; i++) {
        int address = addresses[i];

        int setIndex = address % sets;     // simple mapping
        int tag = address;                 // tag = address (since no
block offset used)

        bool hit = false;

        // Search in the set
        for (int w = 0; w < ways; w++) {
            if (cache[setIndex][w].valid && cache[setIndex][w].tag ==
tag) {
                cout << "Access " << address
                     << ": HIT in Set " << setIndex << ", Way " << w <<
endl;
                hit = true;
                break;
            }
        }
```

```
        if (!hit) {
            cout << "Access " << address
                 << ": MISS: Load into Set " << setIndex << endl;

            // If set is full: replace FIFO way
            if (fifo[setIndex].size() == ways) {
                int replaceWay = fifo[setIndex].front();
                fifo[setIndex].pop();

                cache[setIndex][replaceWay].tag = tag;
                cache[setIndex][replaceWay].data = address;

                fifo[setIndex].push(replaceWay);
            }
            else {
                // Fill empty way
                for (int w = 0; w < ways; w++) {
                    if (!cache[setIndex][w].valid) {
                        cache[setIndex][w].valid = true;
                        cache[setIndex][w].tag = tag;
                        cache[setIndex][w].data = address;

                        fifo[setIndex].push(w);
                        break;
                    }
                }
            }
        }
    }
    return 0;
}
```

OUTPUT:

```
g++ exp8.cpp -o exp8 } ; if ($?) { .\exp

2-Way Set Associative Cache Simulation
---------------------------------------
Access 0: MISS: Load into Set 0
Access 1: MISS: Load into Set 1
Access 2: MISS: Load into Set 0
Access 3: MISS: Load into Set 1
Access 4: MISS: Load into Set 0
Access 1: HIT in Set 1, Way 0
Access 8: MISS: Load into Set 0
Access 9: MISS: Load into Set 1
Access 0: MISS: Load into Set 0
Access 12: MISS: Load into Set 0
PS C:\Users\syeda\OneDrive\Desktop\perso
```

Post Lab Descriptive Questions:

1. For a direct mapped cache, a main memory is viewed as consisting of 3 fields. List and define 3 fields.

Fields in a Direct-Mapped Cache Address

A main memory address in direct-mapped cache is divided into three fields:

- Tag field:
  - Identifies which block of main memory is stored in the cache.
  - Used to check if the required data is present in the cache (cache hit/miss).
- Index (or Line field):
  - Points to a specific cache line where a memory block can be placed.
  - Determines the location in the cache.
- Block Offset (or Word field):

- Indicates the specific word or byte within a cache block.
- Used to select the exact data within that block.

2. What is the general relationship among access time, memory cost, and capacity?

Relationship among Access Time, Memory Cost, and Capacity

- Fast access time implies High cost which implies Small capacity
- Slow access time implies Low cost which implies Large capacity
- Faster memories (like cache and registers) are expensive and smaller in size.
- Slower memories (like main memory and secondary storage) are cheaper and larger.
- Hence, a memory hierarchy is used to balance speed, cost, and capacity effectively

Conclusion:
This experiment successfully demonstrated cache mapping techniques.

Date: 7/11/25