Batch: A1	**Roll No.: 16010123012**
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)

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:

Cache memory: The cache is a smaller, faster memory which stores copies of the data from the most frequently used main memory locations. As long as most memory accesses are cached memory locations, the average latency of memory accesses will be closer to the cache latency than to the latency of main memory.

2. Hit Ratio: You want to increase as much as possible the likelihood of the cache containing the memory addresses that the processor wants.

Hit Ratio= No. of hits/ (No. of hits + No. of misses)

There are only fewer cache lines than the main memory blocks, an algorithm is needed for mapping main memory blocks into cache lines. Further a means is needed for determining which main memory block currently occupies in a cache line. The choice of cache function dictates how the cache is organized. Three techniques can be used.

1. Direct mapping.
2. Associative mapping.

3. Set Associative mapping.

Direct Mapped Cache: The direct mapped cache is the simplest form of cache and the easiest to check for a hit. Since there is only one possible place that any memory location can be cached, there is nothing to search; the line either contains the memory information we are looking for, or it doesn't. Unfortunately, the direct mapped cache also has the worst performance, because again there is only one place that any address can be stored. Let's look again at our 512 KB level 2 cache and 64 MB of system memory. As you recall this cache has 16,384 lines (assuming 32-byte cache lines) and so each one is shared by 4,096 memory addresses. In the absolute worst case, imagine that the processor needs 2 different addresses (call them X and Y) that both map to the same cache line, in alternating sequence (X, Y, X, Y). This could happen in a small loop if you were unlucky. The processor will load X from memory and store it in cache. Then it will look in the cache for Y, but Y uses the same cache line as X, so it won't be there. So Y is loaded from memory, and stored in the cache for future use. But then the processor requests X, and looks in the cache only to find Y. This conflict repeats over and over. The net result is that the hit ratio here is 0%. This is a worst case scenario, but in general the performance is worst for this type of mapping.

Fully Associative Cache: The fully associative cache has the best hit ratio because any line in the cache can hold any address that needs to be cached. This means the problem seen in the direct mapped cache disappears, because there is no dedicated single line that an address must use. However (you knew it was coming), this cache suffers from problems involving searching the cache. If a given address can be stored in any of 16,384 lines, how do you know where it is? Even with specialized hardware to do the searching, a performance penalty is incurred. And this penalty occurs for all accesses to memory, whether a cache hit occurs or not, because it is part of searching the cache to determine a hit. In addition, more logic must be added to determine which of the various lines to use when a new entry must be added (usually some form of a "least recently used" algorithm is employed to decide which cache line to use next). All this overhead adds cost, complexity and execution time.

Set Associative Cache (To be filled in by students)

After CPU generates a memory request,

- The set number field of the address is used to access the corresponding set in the cache.
- The tag field of the CPU address is compared with the tags of all k
- lines within that set.
- If the CPU tag matches the tag of any cache line, a cache hit occurs.
- If the CPU tag does not match the tag of any cache line, a cache miss occurs.
- In the case of a cache miss, the required data must be fetched from main memory.
- If the cache is full, a replacement is made according to the designated replacement policy.

Direct Mapping Implementation:

The mapping is expressed as

i=j modulo m

i=cache line number

j= main memory block number

m= number of lines in the cache

- Address length = (s+w) bits
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2^{w} words or bytes
- Number of blocks in main memory = $2^{s+w} / 2^{w} = 2^{s}$
- Number of lines in cache = m = 2^{r}
- Size of tag = (s-r) tags

Associative Mapping Implementation: **(To be filled in by students)**

• Address length = (s + w) bits
• Number of addressable units= 2 s + words or bytes
• Block size=line size = 2 words or bytes
• Number of blocks in main memory = 2 s
• Number of lines in cache = undefined
• Size of tags = s bits

SetAssociative Mapping Implementation:

m=v*k
i=j modulo n, where:
• i=cache line number
• j=main memory block number
• m=number of lines in the cache
• v=number of sets
• k=number of lines in each set
• Addressable length=(s+w) bits
• Number of addressable units= 2 s+$w$words or bytes
• Block size=line size=2 $w$words or bytes
• Number of blocks in main memory=2 s
• Number of lines in set=k
• Number of sets = v = 2 d
• Number of lines in cache = m = kv = k*2 d
• Size of cache = k * 2 d+$w$words or bytes
• Size of tag = (s-d) bits

Code:

Direct Mapping

```
#include <iostream>

#include <vector>

#include <cstdlib>

using namespace std;


int main() {

    int memoryLines, blocks;

    cout << "Enter number of main memory lines: ";

    cin >> memoryLines;

    cout << "Enter number of blocks in the main memory: ";

    cin >> blocks;

    vector<vector<int>> blockMemory(blocks, vector<int>(4));

    vector<int> mainMemory(memoryLines);
```

```
cout << "\nEnter the main memory data:" << endl;
for (int i = 0; i < memoryLines; i++) {
    cout << "Line no. " << i + 1 << ": ";
    cin >> mainMemory[i];
}

int k = 0;
for (int i = 0; i < blocks; i++) {
    for (int j = 0; j < 4; j++) {
        blockMemory[i][j] = mainMemory[k++];
    }
}

cout << "\nDirect Mapped Cache:\n";
for (int i = 0; i < blocks; i++) {
    cout << "Block " << i << ": ";
    for (int j = 0; j < 4; j++) {
        cout << blockMemory[i][j] << " ";
    }
    cout << endl;
}

cout << "\nSample Cache:\n";
```

```
    for (int i = 0; i < blocks; i++) {
        int randomIndex = rand() % 4;
        cout << blockMemory[i][randomIndex] << " ";
    }
    return 0;
}
```

```
Enter number of main memory lines: 8
Enter number of blocks in the main memory: 2

Enter the main memory data:
Line no. 1: 11
Line no. 2: 2
Line no. 3: 4
Line no. 4: 5
Line no. 5: 3
Line no. 6: 76
Line no. 7: 2
Line no. 8: 32

Direct Mapped Cache:
Block 0: 11 2 4 5
Block 1: 3 76 2 32

Sample Cache:
5 2

=== Code Execution Successful ===
```

SET ASSOCIATIVE

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

```
class TwoWaySetAssociativeCache {
private:
    const int cacheSize;
    const int setSize = 2;
    const int numberOfSets;
    const int blockSize;
    const int memorySize;
    const int tagSize;

    vector<vector<int>> cache;
    vector<vector<int>> tagArray;
    vector<vector<bool>> valid;
    vector<vector<int>> lru;

public:
    TwoWaySetAssociativeCache(int r, int w, int s)
        : cacheSize(static_cast<int>(pow(2, r))),
          numberOfSets(cacheSize / setSize),
          blockSize(static_cast<int>(pow(2, w))),
          memorySize(static_cast<int>(pow(2, s + w))),
          tagSize(s - static_cast<int>(log2(numberOfSets)) - w),
          cache(numberOfSets, vector<int>(setSize)),
          tagArray(numberOfSets, vector<int>(setSize)),
```

```
        valid(numberOfSets, vector<bool>(setSize, false)),
        lru(numberOfSets, vector<int>(setSize, 0)) {}

    void accessMemory(int address) {
        int blockNumber = address / blockSize;
        int index = blockNumber % numberOfSets;
        int tag = blockNumber >> (static_cast<int>(log2(numberOfSets)) +
static_cast<int>(log2(blockSize)));

        for (int i = 0; i < setSize; i++) {
            if (valid[index][i] && tagArray[index][i] == tag) {
                cout << "Cache hit! Data found in set " << index << ", line " << i << endl;
                updateLRU(index, i);
                return;
            }
        }
        cout << "Cache miss! Loading data into set " << index << endl;
        loadBlockToCache(blockNumber, index, tag);
    }

private:
    void loadBlockToCache(int blockNumber, int index, int tag) {
        int lineToReplace = findLineToReplace(index);
        cache[index][lineToReplace] = blockNumber;
        tagArray[index][lineToReplace] = tag;
```

```
        valid[index][lineToReplace] = true;
        updateLRU(index, lineToReplace);
        cout << "Block " << blockNumber << " loaded into set " << index << ", line " <<
lineToReplace << endl;
    }

    int findLineToReplace(int index) {
        for (int i = 0; i < setSize; i++) {
            if (!valid[index][i]) {
                return i;
            }
        }
        return (lru[index][0] == 0 ? 0 : 1);
    }
    void updateLRU(int index, int lineUsed) {
        lru[index][0] = (lineUsed == 0) ? 1 : 0;
        lru[index][1] = (lineUsed == 1) ? 1 : 0;
    }

public:
    void displayCache() {
        cout << "\nCache State:" << endl;
        for (int i = 0; i < numberOfSets; i++) {
            cout << "Set " << i << ": ";
            for (int j = 0; j < setSize; j++) {
```

```
            if (valid[i][j]) {
                cout << "Tag = " << tagArray[i][j] << ", Data = Block " << cache[i][j] << "
| ";
            } else {
                cout << "Invalid | ";
            }
        }
        cout << endl;
    }
  }
};

int main() {
  int r = 4;
  int w = 2;
  int s = 8;
  TwoWaySetAssociativeCache cache(r, w, s);
  cache.accessMemory(10);
  cache.accessMemory(26);
  cache.accessMemory(10);
  cache.displayCache();
  return 0;
}
```

```
Cache miss! Loading data into set 2
Block 2 loaded into set 2, line 0
Cache miss! Loading data into set 6
Block 6 loaded into set 6, line 0
Cache hit! Data found in set 2, line 0

Cache State:
Set 0: Invalid | Invalid |
Set 1: Invalid | Invalid |
Set 2: Tag = 0, Data = Block 2 | Invalid |
Set 3: Invalid | Invalid |
Set 4: Invalid | Invalid |
Set 5: Invalid | Invalid |
Set 6: Tag = 0, Data = Block 6 | Invalid |
Set 7: Invalid | Invalid |


=== Code Execution Successful ===
```

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.

- **Tag**: This field identifies the specific block of data in the main memory that is currently stored in a cache line. It is used to check if the data in the cache corresponds to the requested address.
- **Index**: This field specifies which cache line the data is mapped to. It determines the set (or line) in the cache where the data from main memory should be stored or looked up.
- **Block Offset**: This field indicates the specific byte within a block of data. It is used to pinpoint the exact location of the requested data within the cache line.

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

- Faster access time is directly proportional to cost per bit, meaning that as the speed of access time increases, the cost per bit also rises.
- Memory capacity is inversely proportional to cost per bit, indicating that as memory capacity increases, the cost per bit decreases.
- Memory capacity is inversely proportional to access time, which means that as memory capacity increases, the speed of access time tends to decrease.

Conclusion

As a result of the experiment, various mapping techniques were understood. The task was accomplished by developing programs to demonstrate these techniques.

Date: 07 / 10 / 24