**Batch: C3 Roll No.: 16010123217**

**Experiment / assignment / tutorial No. 6**

|  |
| --- |
| **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 particular set of the cache.

➢ The tag field of the CPU address is then compared with the tags of all k lines

within that set.

➢ If the CPU tag matches to the tag of any cache line, a cache hit occurs.

➢ If the CPU tag does not match to the tag of any cache line, a cache miss occurs.

➢ In case of a cache miss, the required word has to be brought from the main

memory.

➢ If the cache is full, a replacement is made in accordance with the employed

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 = 2s+w words or bytes
* Block size = line size = 2w words or bytes
* Number of blocks in main memory = 2s+w / 2w = 2s
* Number of lines in cache = m = 2r
* Size of tag = (s-r) tags

#include<bits/stdc++.h>

using namespace std;

int main()

{

    int page\_size;

    int cache;

    int mainm;

    cout<<"Enter the Page Size \n";

    cin>>page\_size;

    cout<<"Enter the cache memory size \n";

    cin>>cache;

    cout<<"Enter the Main memory size \n";

    cin>>mainm;

    cout<<endl;

    int cache\_pages=cache/page\_size;

    int main\_pages=mainm/page\_size;

    vector<int>  main\_memory;

    int temp;

    cout<<"Enter the pages wanted for "<<main\_pages<< " enter -1 to stop";

    while(temp!=-1)

    {

        cin>>temp;

        if(temp==-1)

        {

            break;

        }

        if(temp>main\_pages)

        {

             cout<<"Exceed the memory Try Again";

        }

        else

        {

            main\_memory.push\_back(temp);

        }

    }

    cout<<"The input"<<endl;

    for(int i=0;i<main\_memory.size();i++)

    {

        cout<<main\_memory[i]<<" ";

    }

    cout<<"\n";

    int n=cache\_pages;

    vector<int> cache\_memory(cache\_pages,-1);

    int temp2;

    for(int i=0;i<main\_memory.size();i++)

    {

        cache\_memory[(main\_memory[i]-1)%cache\_pages]=main\_memory[i];

        for(int j=0;j<n;j++)

        {

            if (cache\_memory[j] != -1)

            {

                cout << setw(2) << cache\_memory[j] << " ";

            }

            else

            {

                cout << "- ";

            }

        }

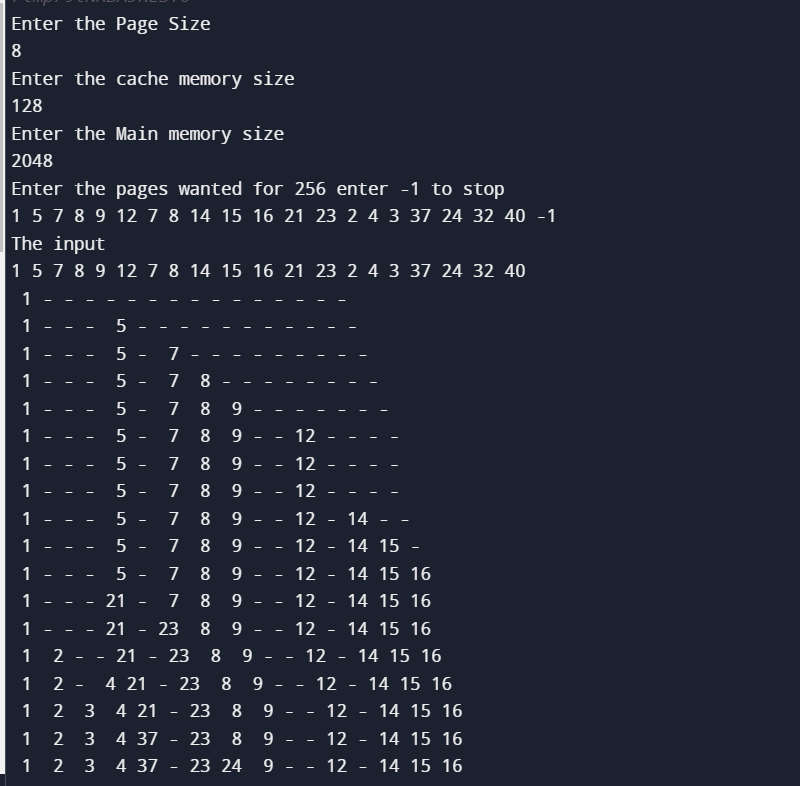
        cout<<endl;

    }

    return 0;

}

**Output:**



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

- Associative mapping allows any memory block to be placed in any cache line, offering flexibility.

- Each cache line features a tag field to compare with memory addresses for validity.

- Complex hardware and slower access times result from tag comparisons.

- Suitable for systems with diverse memory access patterns, offering better hit rates than direct-mapped caches.

- Common in high-performance processors, like multi-core CPUs, where flexibility and dynamic allocation of cache space are crucial.

**Set** **Associative Mapping Implementation**:

- Cache Organization: In a 2-way set associative cache, the cache is divided into sets, each containing two cache lines.

- Indexing: A portion of the memory address is used as an index to determine which set the data maps to. For example, with 4 sets, 2 bits are used for indexing.

- Tag Bits: The remaining bits of the memory address serve as tag bits, and each cache line in a set contains a tag field for comparison.

- Replacement Policy: When both lines in a set are occupied, a replacement policy like LRU decides which line to evict.

- Performance: 2-way set associative caches balance between speed and flexibility, offering improved hit rates compared to direct-mapped caches without the complexity of fully associative caches.

- Common Usage: This configuration is often found in modern processors to optimize memory access.

Code:

#include <bits/stdc++.h>

using namespace std;

int print(vector<vector<int>> & set\_cache,int cache\_lines,int num, int set\_lines,int m)

{

    int n=cache\_lines/2;

    cout<<"\nSet Cache"<<endl;

    bool inserted=false;

    int cnter=0;

    if(set\_cache[set\_lines][0]==-1)

    {

        inserted=true;

        set\_cache[set\_lines][0]=num;

    }

    else if(set\_cache[set\_lines][1]==-1 && inserted==false)

    {

        set\_cache[set\_lines][1]=num;

        cnter=1;

    }

    else

    {

        if(set\_cache[set\_lines][1]!=-1 && set\_cache[set\_lines][0]!=-1 && m==1)

        {

            set\_cache[set\_lines][0]=num;

            cnter=0;

        }

        else

        {

            set\_cache[set\_lines][1]=num;

           cnter=1;

        }

    }

    for(int i=0;i<n;i++)

    {

        for(int j=0;j<2;j++)

        {

            if(set\_cache[i][j]==-1)

            {

                cout<<"- ";

            }

            else

            {

                cout<<set\_cache[i][j]<<" ";

            }

        }

        cout<<endl;

    }

    return cnter;

}

void onlyprint(vector<vector<int>> & set\_cache,int cache\_lines)

{

     int n=cache\_lines/2;

    cout<<"\nSet Cache Hit"<<endl;

    for(int i=0;i<n;i++)

    {

        for(int j=0;j<2;j++)

        {

            if(set\_cache[i][j]==-1)

            {

                cout<<"- ";

            }

            else

            {

                cout<<set\_cache[i][j]<<" ";

            }

        }

        cout<<endl;

    }

}

int main()

{

    int page\_size;

    int cache\_size;

    int main\_memory\_size;

    cout << "Enter the Page Size: ";

    cin >> page\_size;

    cout << "Enter the cache memory size: ";

    cin >> cache\_size;

    cout << "Enter the Main memory size: ";

    cin >> main\_memory\_size;

    cout << endl;

    int cache\_lines = cache\_size / page\_size;

    int main\_pages = main\_memory\_size / page\_size;

    vector<int> main\_memory;

    int temp;

    int page\_count=0;

    cout<<"Enter the number to be Entered";

    while (page\_count < main\_pages)

    {

        cin >> temp;

        if (temp == -1)

        {

            break;

        }

        if (temp >= main\_pages)

        {

            cout << "Exceeded the memory. Try Again." << endl;

        }

        else

        {

            main\_memory.push\_back(temp);

        }

    }

    cout<<"Page References in Main Memory\n";

    for(int i=0;i<main\_memory.size();i++)

    {

        cout<<main\_memory[i]<<" ";

    }

    cout<<endl;

    vector<vector<int>> set\_cache(cache\_lines/2, vector<int> (2,-1));

    int set\_line;

    int m=0;

    for(int i=0;i<main\_memory.size();i++)

    {

        int set\_line = main\_memory[i] % (cache\_lines / 2);

        if (set\_line == 0)

        {

            set\_line = 7;

        }

        else

        {

            set\_line = set\_line - 1;

        }

            bool found = false;

            for (auto it = set\_cache[set\_line].begin(); it != set\_cache[set\_line].end(); ++it)

            {

                if (\*it == main\_memory[i])

                {

                    found = true;

                    break;

                }

            }

            if (found)

            {

                onlyprint(set\_cache,cache\_lines);

                continue;

            }

            else

            {

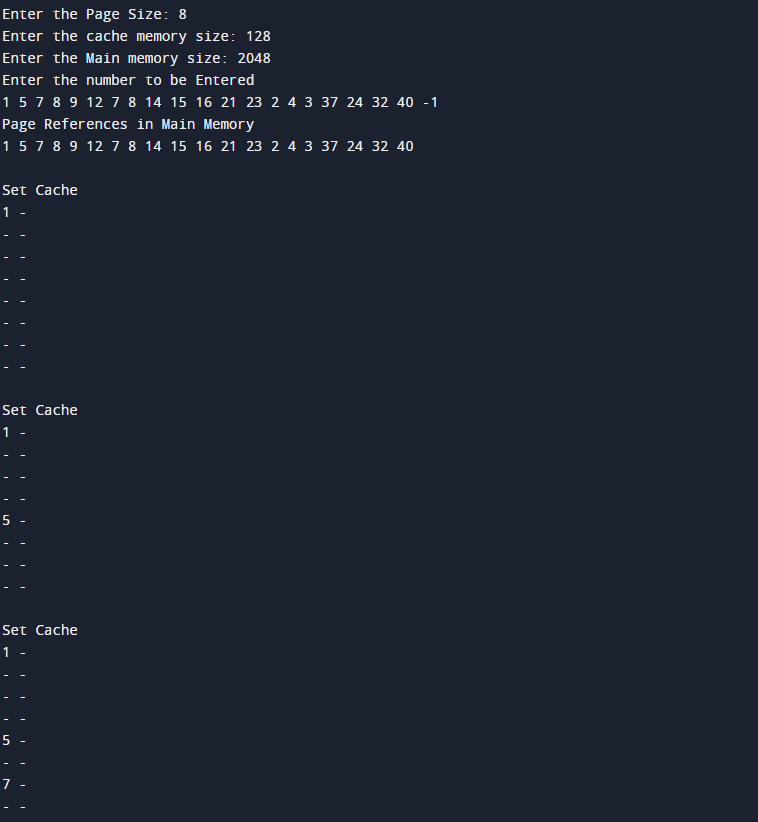
                m=print(set\_cache,cache\_lines,main\_memory[i],set\_line,m);

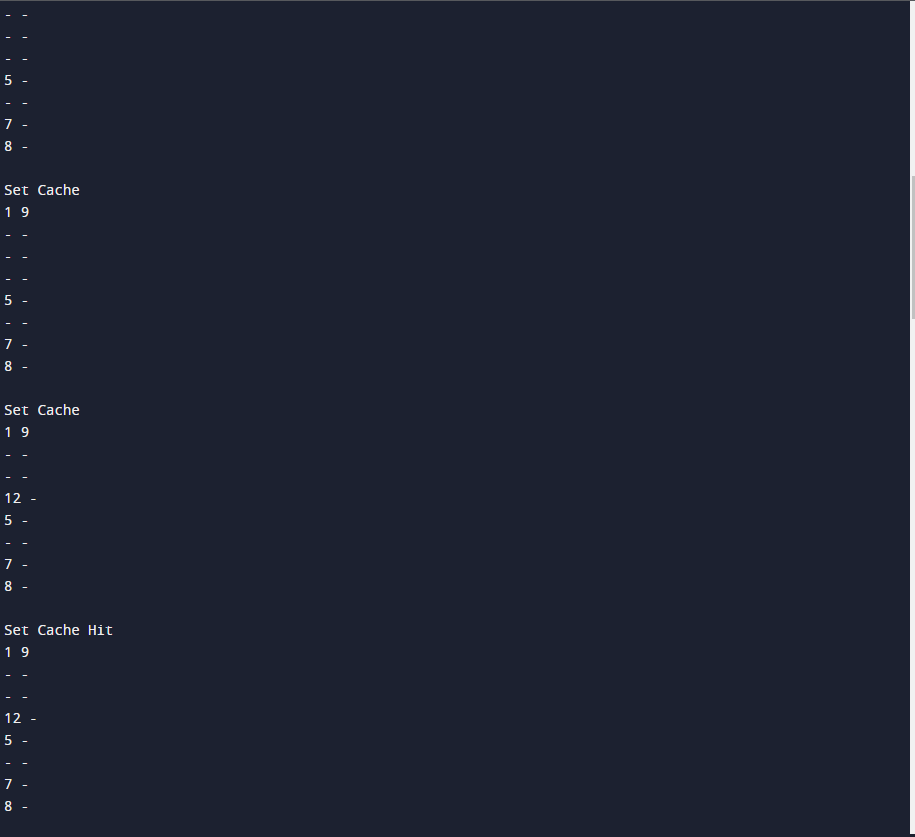
            }

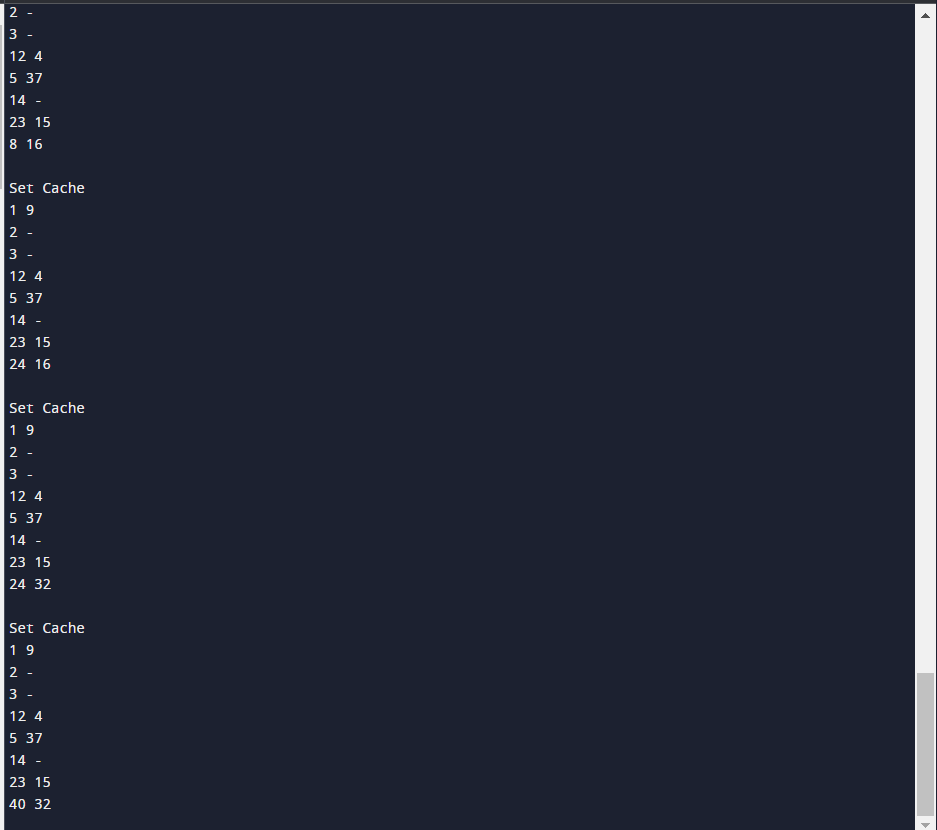
    }

    return 0;

}

Output:  






**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.**

Ans.

The fields would be i, j and m. I is the cache line number, j is the main memory block

number and m is the number of lines in the cache.

1. **i (Cache Line Number)**: This field identifies the specific cache line within the cache. It indicates which cache line in the cache the data is stored in. This field helps in direct mapping by specifying the location within the cache where the data should be stored.

2. **j (Main Memory Block Number)**: This field represents the main memory block number. It identifies the block or location in main memory from which data is being read or to which data is being written. It helps locate the data within the main memory.

3. **m (Number of Lines in the Cache)**: This field indicates the total number of cache lines available in the cache. It defines the size and capacity of the cache and limits the number of cache lines that can be used to store data from main memory.

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

Ans. As access time becomes faster, the cost per bit increases. As memory size increases, the cost per bit is smaller. Also, with greater capacity, access time becomes slower.

**Conclusion**

**We learned the concept of cache mapping and its implementation by coding it out and knew example for it.**

**Date: 11/10/24**