6. Memory Mapping Techniques

Aim:

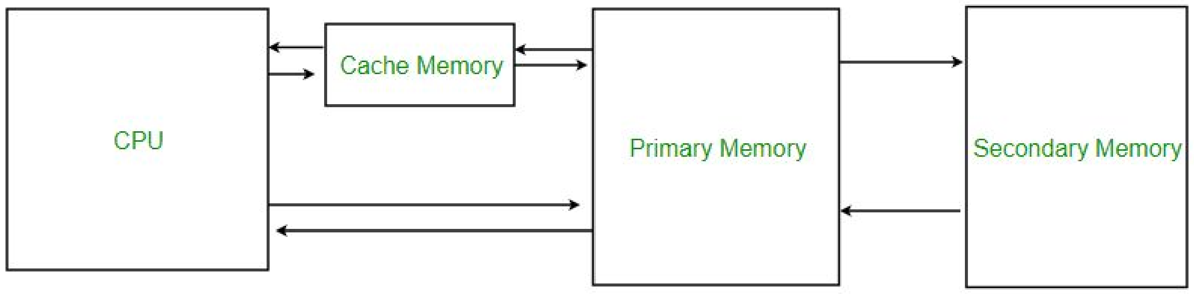
To study and implement various memory mapping techniques.

Theory:

Cache Memory

Cache Memory is a special high-speed memory. It is used to speed up and synchronizing with high-speed CPU. Cache memory is costlier than main memory or disk memory but economical than CPU registers. Cache memory acts as a buffer between RAM and the CPU. It holds frequently requested data and instructions so that they are immediately available to the CPU when needed.

Cache memory is used to reduce the average time to access data from the Main memory. The cache is a smaller and faster memory which stores copies of the data from frequently used main memory locations. There are various different independent caches in a CPU, which store instruction and data.



Cache Performance

When the processor needs to read or write a location in main memory, it first checks for a corresponding entry in the cache.

- If the processor finds that the memory location is in the cache, a cache hit has occurred and data is read from cache.
- If the processor does not find the memory location in the cache, a cache miss has occurred. For a cache miss, the cache allocates a new entry and copies in data from main memory, then the request is fulfilled from the contents of the cache.

The performance of cache memory is frequently measured in terms of a quantity called Hit ratio.

```
Hit ratio =  no. of hits/total accesses
```

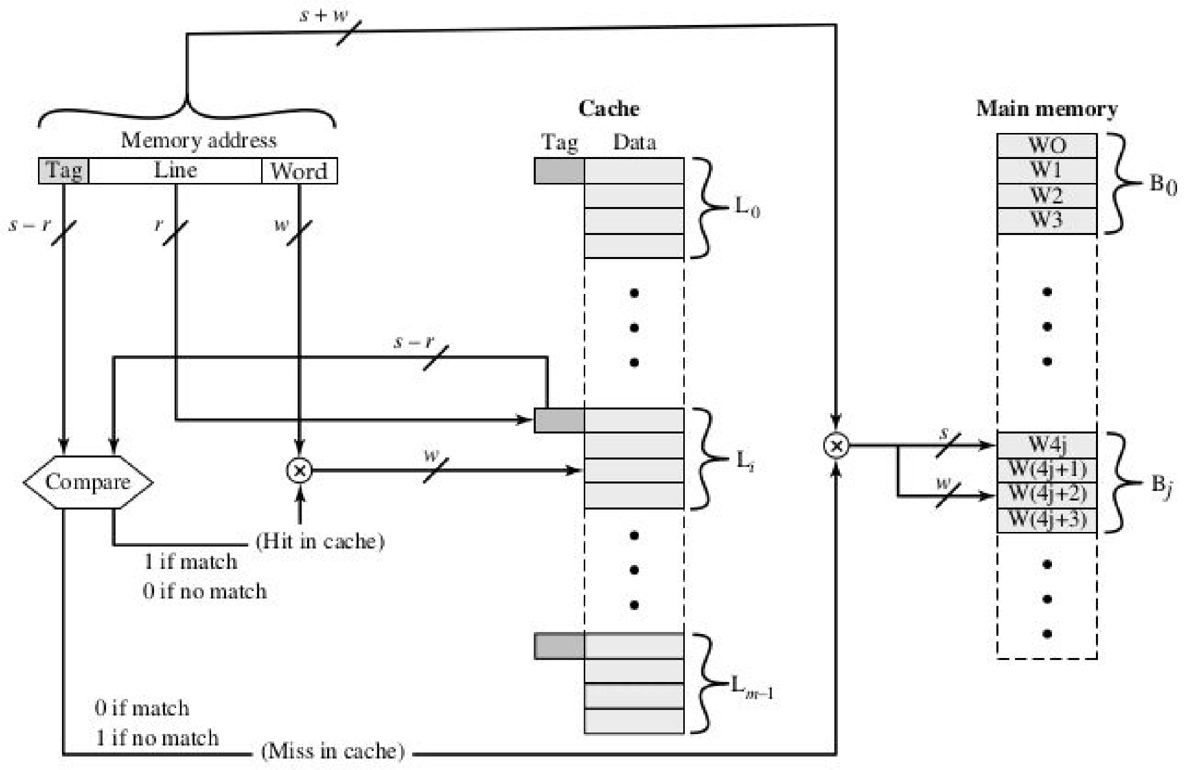
Cache Mapping

1. Direct Mapping

In Direct mapping, each memory block is always assigned to a specific line in the cache. If a line is previously taken up by a memory block when a new block needs to be loaded, the old block is trashed. Direct mapping's performance is directly proportional to the Hit ratio.

```
i = j modulo m
where
i=cache line number                                    j= main memory block number
m=number of lines in the cache
```

For purposes of cache access, each main memory address can be viewed as consisting of three fields. The least significant w bits identify a unique word or byte within a block of main memory. The remaining s bits specify one of the 2^s blocks of main memory. The cache logic interprets these s bits as a tag of s-r bits (most significant portion) and a line field of r bits. This latter field identifies one of the $m=2^r$ lines of the cache.



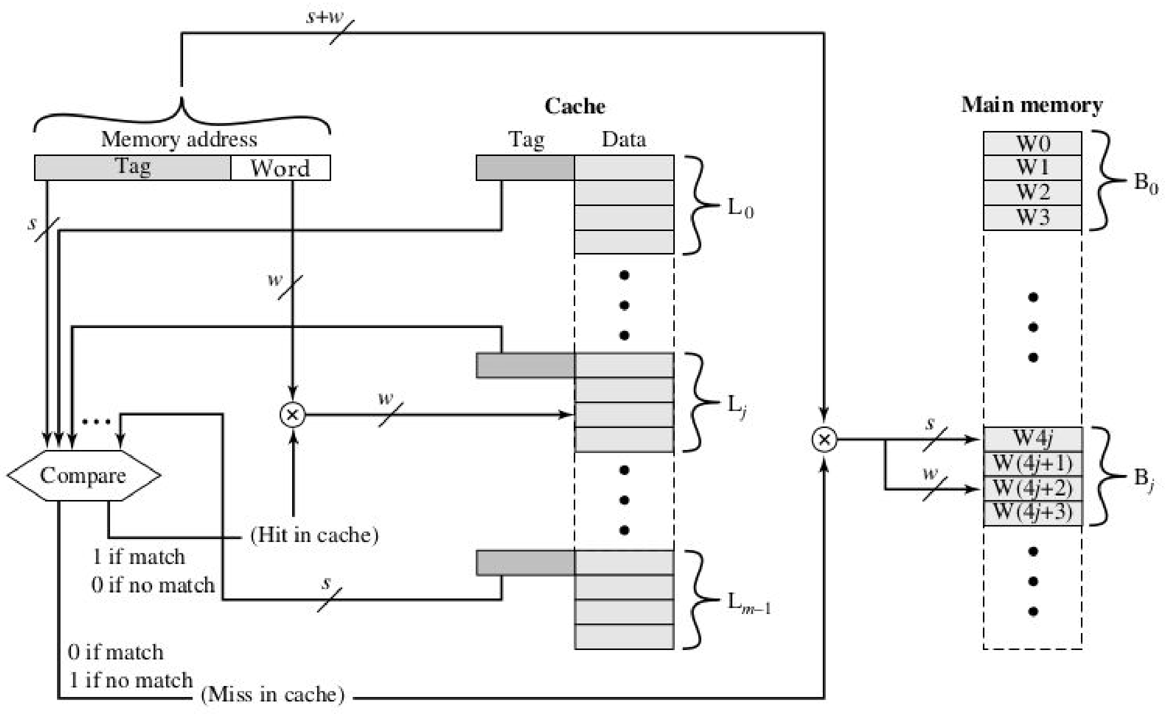
Its main disadvantage is that there is a fixed cache location for any given block. Thus, if a program happens to reference words repeatedly from two different blocks that map into the same line, then the blocks will be continually swapped in the cache, and the hit ratio will be low (a phenomenon known as thrashing).

2. Associative Mapping

Associative mapping overcomes the disadvantage of direct mapping by permitting each main memory block to be loaded into any line of the cache. In this case, the cache control logic interprets a memory address simply as a Tag and a Word field. The Tag field uniquely identifies a block of main memory. To determine whether a block is in the cache, the cache control logic must simultaneously examine every line's tag for a match.

```
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 = undetermined
Size of tag = s bits
```

With associative mapping, there is flexibility as to which block to replace when a new block is read into the cache. The principal disadvantage of associative mapping is the complex circuitry required to examine the tags of all cache lines in parallel.



3. Set-Associative Mapping

Set-associative mapping is a compromise that exhibits the strengths of both the direct and associative approaches while reducing their disadvantages.

In this case, the cache consists of a number of sets, each of which consists of a number of lines. The relationships are

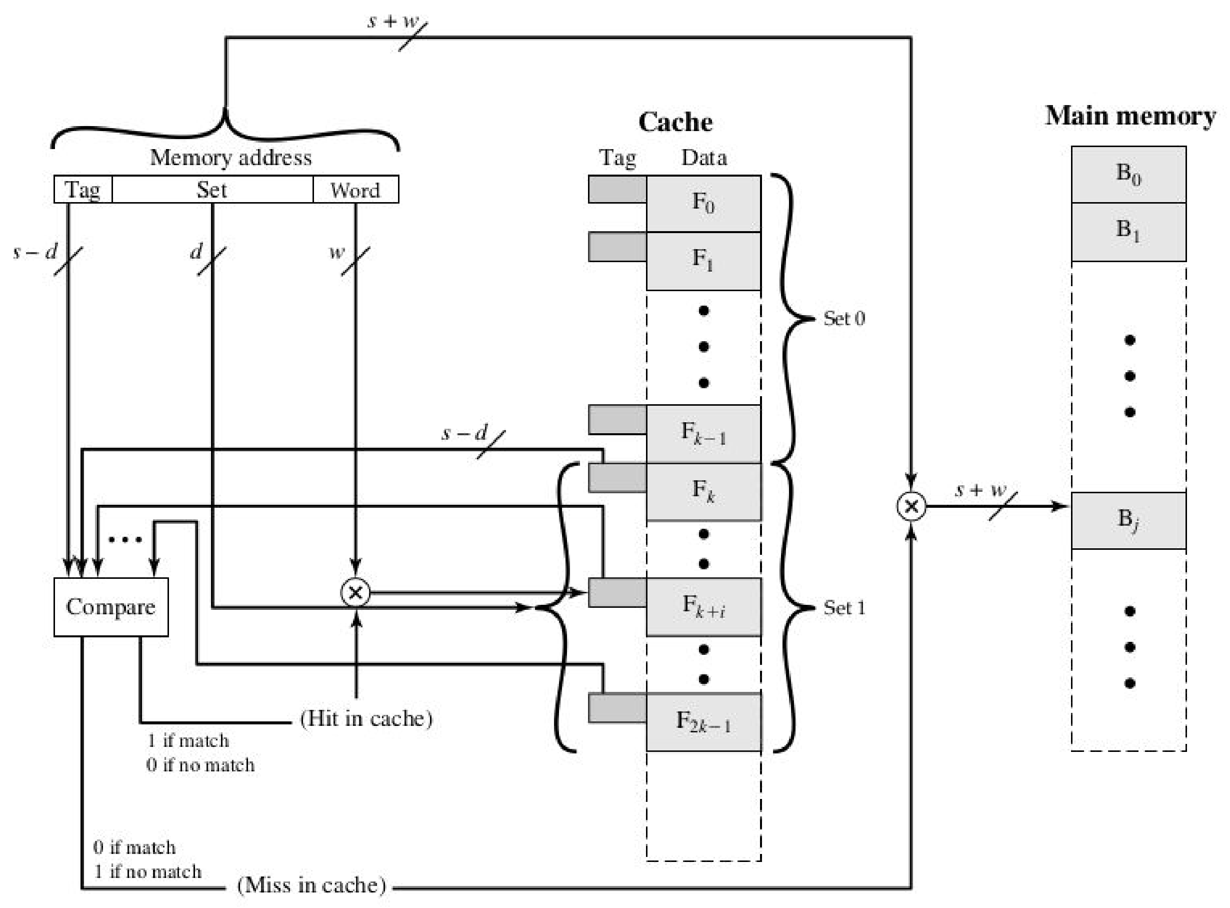
```
m = n * k
i = j modulo n
where
i = cache set number
j = main memory block number
m = number of lines in the cache
n = number of sets
k = number of lines in each set
```

This is referred to as k-way set-associative mapping. With set-associative mapping, block Bj can be mapped into any of the lines of set j. As with associative mapping, each word maps into multiple cache lines. For set-associative mapping, each word maps into all the cache lines in a specific set, so that main memory block B_0 maps into set 0, and so on. Thus, the set-associative cache can be physically implemented as n associative caches. It is also possible to implement the set-associative cache as k direct mapping caches. Each direct-mapped cache is referred to as a way, consisting of n lines. The first n lines of main memory are direct mapped into the n lines of each way; the next group of n lines of main memory are similarly mapped, and so on. For set-associative mapping, the cache control logic interprets a memory address as three fields: Tag, Set, and Word. The d set bits specify one of $v = 2^d$ sets. The s bits of the Tag and Set fields specify one of the 2^s blocks of main memory.

With fully associative mapping, the tag in a memory address is quite large and must be compared to the tag of every line in the cache. With k-way set-associative mapping, the tag in a memory address is much smaller and is only compared to the k tags within a single set.

To summarize,

```
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 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
```


s + w
Memory address
Tag
Set
Word
s − d
d
w
Cache
Tag
Data
F0
F1
Fk−1
Fk
Fk+i
F2k−1
Set 0
Set 1
s − d
Compare
(Hit in cache)
1 if match
0 if no match
0 if match
1 if no match
(Miss in cache)
s + w
Main memory
B0
B1
Bj

Program:

Direct Memory Mapping

```java
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.ArrayList;

import org.apache.commons.lang3.StringUtils;

/**
 * JAVA class to demonstrate the direct memory
 * mapping technique.
 *
 * @author jimil
 */
public class DirectMemoryMapping {

    static ArrayList<Line> cache = new ArrayList<Line>();
    static BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
    static int hitCount = 0;
    static int faultCount = 0;
    static int w = 0;
    static int r = 0;
    static int tagSize = 0;
    static int sw = 0;

    /**
     * @param args
     */
    public static void main(String[] args) throws Exception {
        System.out.print("Enter the size of main memory (in n-bit address):
");
        sw = Integer.parseInt(br.readLine());
        System.out.println();
        System.out.print("Enter the size of cache (in Kb): ");
        int cacheSize = Integer.parseInt(br.readLine());
        System.out.println();
        System.out.print("Enter the size of block (in bytes): ");
        int blockSize = Integer.parseInt(br.readLine());
        System.out.println();
        w = (int)(Math.log((double)blockSize)/Math.log(2));
        int x = (cacheSize*1024)/blockSize;
        r = (int)(Math.log((double)x)/Math.log(2));
        tagSize = sw - r - w;
        System.out.println("Memory Config: ");
        System.out.println("Tag(" + tagSize + " bits), Line(" + r + "
bits), Word(" + w + " bits)");
        for(int i = 0; i < x; i++) {
            Line l = new Line();
            l.setLine(i);
            cache.add(l);
        }
        int ch = 1;
        int count = 0;
        do {
            count++;
            System.out.println("Enter a physical address to check: ");
            int address = Integer.parseInt(br.readLine());
            String addr = Integer.toBinaryString(address);
            if(addr.length() != sw) {
                addr = StringUtils.leftPad(addr, sw, "0");
                System.out.println("Checking for address: " + addr);
            }
            check(addr);
            System.out.println("Press 1. to check another address and 0. to
exit");
            ch = Integer.parseInt(br.readLine());
        } while(ch != 0);
        System.out.println("Hit Ratio = " +
(double)hitCount/(double)count);
        System.out.println("Fault Ratio = " +
(double)faultCount/(double)count);
    }

    /**
     * A utility method that checks for the physical address
     * sent to it, inside cache and determines whether it is
     * a page fault or a page hit.
     *
     * @param addr: physical address to check for, in cache.
     */
    public static void check(String addr) {
        String tag = addr.substring(0, tagSize);
        String line = addr.substring(tagSize, addr.length() - w);
        String word = addr.substring(addr.length() - w);
        System.out.println("["+tag+"],["+line+"],["+word+"]");
        int cacheLine = Integer.parseInt(line, 2);
        for(Line lineObj: cache) {
            if(lineObj.getLine() == cacheLine) {
                if(tag.equalsIgnoreCase(lineObj.getTag())) {
                    hitCount++;
                    System.out.println("Cache Hit!");
                } else {
                    faultCount++;
                    System.out.println("Cache Miss!");
                    System.out.println("Updating tag of " + cacheLine + "
line to " + tag);
                    lineObj.setTag(tag);
                }
            }
        }
    }

}

/**
 * A POJO that models a cache line
 *
 * @author jimil
 */
class Line {
    private String tag;
    private int line;
    private int word;
    public String getTag() {
        return tag;
    }
    public void setTag(String tag) {
        this.tag = tag;
    }
    public int getLine() {
        return line;
    }
    public void setLine(int line) {
        this.line = line;
    }
    public int getWord() {
        return word;
    }
    public void setWord(int word) {
        this.word = word;
    }
}
```

Output:

```
Enter the size of main memory (in n-bit address): 32
Enter the size of cache (in Kb): 524288
Enter the size of block (in bytes): 64

Memory Config: Tag(3 bits), Line(23 bits), Word(6 bits)
Enter a physical address to check:
120
Checking for address: 00000000000000000000000001111000
[000],[00000000000000000000001],[111000]
Cache Miss!
Updating tag of 1 line to 000
Press 1. to check another address and 0. to exit
1
Enter a physical address to check:
121
Checking for address: 00000000000000000000000001111001
[000],[00000000000000000000001],[111001]
Cache Hit!
Press 1. to check another address and 0. to exit
1
Enter a physical address to check:
182
Checking for address: 00000000000000000000000010110110
[000],[00000000000000000000010],[110110]
Cache Miss!
Updating tag of 2 line to 000
Press 1. to check another address and 0. to exit
0
Hit Ratio = 0.3333333333333333
Fault Ratio = 0.6666666666666666
```

Associative Mapping

```java
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.ArrayList;

import org.apache.commons.lang3.StringUtils;

/**
 * JAVA class to demonstrate the fully associative
 * memory mapping technique.
 *
 * @author jimil
 */
public class AssociativeMapping {

    static ArrayList<CacheLine> cache = new ArrayList<CacheLine>();
    static BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
    static int hitCount = 0;
    static int faultCount = 0;
    static int w = 0;
    static int tagSize = 0;
    static int sw = 0;

    /**
     * @param args
     */
    public static void main(String[] args) throws Exception {
        System.out.print("Enter the size of main memory (in n-bit address):
");
        sw = Integer.parseInt(br.readLine());
        System.out.println();
        System.out.print("Enter the size of block (in bytes): ");
        int blockSize = Integer.parseInt(br.readLine());
        System.out.println();
        w = (int)(Math.log((double)blockSize)/Math.log(2));
        tagSize = sw - w;
        System.out.println("Memory Config: ");
        System.out.println("Tag(" + tagSize + " bits), Word(" + w + "
bits)");
        for(int i = 0; i < 100; i++) {
            CacheLine cl = new CacheLine();
            cache.add(cl);
        }
        int ch = 1;
        int count = 0;
        do {
            count++;
            System.out.println("Enter a physical address to check: ");
            int address = Integer.parseInt(br.readLine());
            String addr = Integer.toBinaryString(address);
            if(addr.length() != sw) {
                addr = StringUtils.leftPad(addr, sw, "0");
                System.out.println("Checking for address: " + addr);
            }
            check(addr);
            System.out.println("Press 1. to check another address and 0. to
exit");
            ch = Integer.parseInt(br.readLine());
        } while(ch != 0);
        System.out.println("Hit Ratio = " +
(double)hitCount/(double)count);
        System.out.println("Fault Ratio = " +
(double)faultCount/(double)count);
    }

    /**
     * A utility method that checks for the physical address
     * sent to it, inside cache and determines whether it is
     * a page fault or a page hit.
     *
     * @param addr: physical address to check for, in cache.
     */
    public static void check(String addr) {
        String tag = addr.substring(0, tagSize);
        String word = addr.substring(addr.length() - w);
        System.out.println("["+tag+"],["+word+"]");
        for(CacheLine lineObj: cache) {
            if(tag.equalsIgnoreCase(lineObj.getTag())) {
                hitCount++;
                System.out.println("Cache Hit!");
                break;
            } else {
                faultCount++;
                System.out.println("Cache Miss!");
                System.out.println("Updating tag to " + tag);
                lineObj.setTag(tag);
                break;
            }
        }
    }

}
/**
 * A POJO that models a cache line
 *
 * @author jimil
 */
class CacheLine {

    private String tag;
    private int word;
    public String getTag() {
        return tag;
    }
    public void setTag(String tag) {
        this.tag = tag;
    }
    public int getWord() {
        return word;
    }
    public void setWord(int word) {
        this.word = word;
    }

}
```

Output:

```
Enter the size of main memory (in n-bit address): 32
Enter the size of block (in bytes): 64

Memory Config: Tag(26 bits), Word(6 bits)
Enter a physical address to check:
120
Checking for address: 00000000000000000000000001111000
[00000000000000000000000001],[111000]
Cache Miss!
Updating tag to 00000000000000000000000001
Press 1. to check another address and 0. to exit
1
Enter a physical address to check:
121
Checking for address: 00000000000000000000000001111001
[00000000000000000000000001],[111001]
Cache Hit!
Press 1. to check another address and 0. to exit
1
Enter a physical address to check:
182
Checking for address: 00000000000000000000000010110110
[00000000000000000000000010],[110110]
Cache Miss!
Updating tag to 00000000000000000000000010
Press 1. to check another address and 0. to exit
0
Hit Ratio = 0.3333333333333333
Fault Ratio = 0.6666666666666666
```

Set Associative Mapping

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.ArrayList;

import org.apache.commons.lang3.StringUtils;

/**
 * JAVA program to demonstrate the Set Associative
 * memory mapping technique
 *
 * @author jimil
 */
public class SetAssociativeMapping {

    static ArrayList<CacheSet> cache = new ArrayList<CacheSet>();
    static BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
    static int hitCount = 0;
    static int faultCount = 0;
    static int w = 0;
    static int r = 0;
    static int tagSize = 0;
    static int sw = 0;

    /**
     * @param args
     */
    public static void main(String[] args) throws Exception {
        System.out.print("Enter the size of main memory (in n-bit address):
");
        sw = Integer.parseInt(br.readLine());
        System.out.println();
        System.out.print("Enter the size of cache (in Kb): ");
        int cacheSize = Integer.parseInt(br.readLine());
        System.out.println();
        System.out.print("Enter the size of block (in bytes): ");
        int blockSize = Integer.parseInt(br.readLine());
        System.out.println();
        boolean flag = false;
        int linePerSet = 0;
        while(!flag) {
            System.out.print("Enter the no of lines in each set: ");
            linePerSet = Integer.parseInt(br.readLine());
            for(int i = 1; i <= 5; i++) {
                if(i == Math.log(linePerSet)/Math.log(2)) {
                    flag = true;
                    break;
                }
            }
            if(flag == false) {
                System.out.println("No. of lines per set should be in
powers of 2. Please retry!");
            }
        }
        System.out.println();
        w = (int)(Math.log((double)blockSize)/Math.log(2));
        int x = (cacheSize*1024)/(blockSize*linePerSet);
        r = (int)(Math.log((double)x)/Math.log(2));
        tagSize = sw - r - w;
        System.out.println("Memory Config: ");
        System.out.println("Tag(" + tagSize + " bits), Set(" + r + " bits),
Word(" + w + " bits)");
        for(int i = 0; i < x; i++) {
            CacheSet set = new CacheSet();
            set.setId(i);
            for(int j = 0; j < linePerSet; j++) {
                CacheLineDTO l = new CacheLineDTO();
                l.setSet(i);
                l.setLine(j);
                set.add(l);
            }
            cache.add(set);
        }
        int ch = 1;
        int count = 0;
        do {
            count++;
            System.out.println("Enter a physical address to check: ");
            int address = Integer.parseInt(br.readLine());
            String addr = Integer.toBinaryString(address);
            if(addr.length() != sw) {
                addr = StringUtils.leftPad(addr, sw, "0");
                System.out.println("Checking for address: " + addr);
            }
            check(addr);
            System.out.println("Press 1. to check another address and 0. to
exit");
            ch = Integer.parseInt(br.readLine());
        } while(ch != 0);
        System.out.println("Hit Ratio = " +
(double)hitCount/(double)count);
        System.out.println("Fault Ratio = " +
(double)faultCount/(double)count);
    }

    /**
     * A utility method that checks for the physical address
     * sent to it, inside cache and determines whether it is
     * a page fault or a page hit.
     *
     * @param addr: physical address to check for, in cache.
     */
    public static void check(String addr) {
        String tag = addr.substring(0, tagSize);
        String set = addr.substring(tagSize, addr.length() - w);
        String word = addr.substring(addr.length() - w);
        System.out.println("["+tag+"],["+set+"],["+word+"]");
        int cacheSet = Integer.parseInt(set, 2);
        for(CacheSet cacheSetObj: cache) {
            if(cacheSetObj.getId() == cacheSet) {
                boolean flag = false;
                for(int i = 0; i < cacheSetObj.getLineSize(); i++) {
                    CacheLineDTO cacheLineDTO = cacheSetObj.get(i);
                    if(cacheLineDTO.getTag() == null) {
                        break;
                    }
                    if(cacheLineDTO.getTag().equalsIgnoreCase(tag)) {
                        System.out.println("Cache Hit!");
                        hitCount++;
                        flag = true;
                        break;
                    }
                }
                if(!flag) {
                    faultCount++;
                    System.out.println("Cache Miss! Updating tag of line 0 
to " + tag);
                    cacheSetObj.get(0).setTag(tag);
                    break;
                }
            }
        }
    }

}

/**
 * A POJO that models a cache line
 *
 * @author jimil
 */
class CacheLineDTO {
    private String tag;
    private int set;
    private int line;
    private int word;
    public String getTag() {
        return tag;
    }
    public void setTag(String tag) {
        this.tag = tag;
    }
    public int getSet() {
        return set;
    }
    public void setSet(int set) {
        this.set = set;
    }
    public int getLine() {
        return this.line;
    }
    public void setLine(int line) {
        this.line = line;
    }
    public int getWord() {
        return word;
    }
```

```java
    public void setWord(int word) {
        this.word = word;
    }
    @Override
    public String toString() {
        return "{ Tag = "+tag+", Set = " + set + ", Line = " + line + " }";
    }
}

/**
 * POJO class that models a cache set -
 * a collection of lines
 *
 * @author jimil
 */
class CacheSet {

    private int id;
    private ArrayList<CacheLineDTO> lines = new ArrayList<>();

    public void setId(int id) {
        this.id = id;
    }

    public int getId() {
        return this.id;
    }

    public void add(CacheLineDTO cacheLineDTO) {
        if(this.lines == null) {
            this.lines = new ArrayList<>();
        }
        this.lines.add(cacheLineDTO);
    }

    public CacheLineDTO get(int index) {
        if(index >= 0) {
            return this.lines.get(index);
        }
        return null;
    }
    public int getLineSize() {
        if(this.lines != null) {
            return this.lines.size();
        }
        return 0;
    }
}
```

Output:

```
Enter the size of main memory (in n-bit address): 16
Enter the size of cache (in Kb): 2
Enter the size of block (in bytes): 64
Enter the no of lines in each set: 2

Memory Config: Tag(6 bits), Set(4 bits), Word(6 bits)
Enter a physical address to check:
128
Checking for address: 0000000010000000
[000000],[0010],[000000]
Cache Miss! Updating tag of line 0 to 000000
Press 1. to check another address and 0. to exit
1
Enter a physical address to check:
129
Checking for address: 0000000010000001
[000000],[0010],[000001]
Cache Hit!
Press 1. to check another address and 0. to exit
1
Enter a physical address to check:
130
Checking for address: 0000000010000010
[000000],[0010],[000010]
Cache Hit!
Press 1. to check another address and 0. to exit
1
Enter a physical address to check:
2176
Checking for address: 0000100010000000
[000010],[0010],[000000]
Cache Miss! Updating tag of line 0 to 000010
```

```
Press 1. to check another address and 0. to exit
1
Enter a physical address to check:
2177
Checking for address: 0000100010000001
[000010],[0010],[000001]
Cache Hit!
Press 1. to check another address and 0. to exit
0
Hit Ratio = 0.6
Fault Ratio = 0.4
```

Conclusion:

Thus, through this experiment, I studied the various cache mapping techniques. I also implemented these Cache Mapping techniques. It was observed that Direct Mapping lead to thrashing when process requested words repeatedly from two different blocks that mapped into the same line. Associative Mapping overcame thrashing, however, it required complex circuitry to search for the tag exhaustively through all the lines in the cache. Set Associative Mapping reduced the possibility of thrashing by allowing blocks to map to any line within a fixed set number. It is also faster than Associative mapping, since exhaustive search is not required. Only the lines within the mapped set are to be checked for comparing the tags.