# Final Project: Cache memory simulation

```
import java.util.*;

public class Main {

    // Cache block structure
    static class CacheBlock {
        int tag;
        boolean valid;
        boolean dirty; // For write-back policy
        int lruCounter; // For LRU tracking

        CacheBlock() {
            this.valid = false;
            this.dirty = false;
            this.lruCounter = 0;
        }
    }

    // Constants for policy readability
    static final int LRU = 0;
    static final int FIFO = 1;
    static final int WRITE_THROUGH = 0;
    static final int WRITE_BACK = 1;

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Input cache and memory configurations
        try {
            System.out.println("Enter the cache size (in blocks): ");
            int cacheSize = scanner.nextInt();

            System.out.println("Enter the block size (in words): ");
            int blockSize = scanner.nextInt();

            System.out.println("Enter the associativity (1 for direct-mapped, N for N-way): ");
            int associativity = scanner.nextInt();

            System.out.println("Enter the replacement policy (0 for LRU, 1 for FIFO): ");
            int replacementPolicy = scanner.nextInt();
```

```
System.out.println("Enter the write policy (0 for Write-Through, 1 for Write-Back): ");
int writePolicy = scanner.nextInt();

// Hard-coded memory access sequence
List<Integer> addresses = Arrays.asList(4, 8, 16, 4, 20, 8, 24, 32, 4, 20);

// Derived parameters
int numSets = cacheSize / (blockSize * associativity);
CacheBlock[][] cache = new CacheBlock[numSets][associativity];
int[] fifoPointer = new int[numSets]; // FIFO pointers for each set
Arrays.fill(fifoPointer, 0); // Initialize FIFO pointers

// Initialize cache blocks
for (int i = 0; i < numSets; i++) {
    for (int j = 0; j < associativity; j++) {
        cache[i][j] = new CacheBlock();
    }
}

int cacheHits = 0;
int cacheMisses = 0;

// Process memory accesses
for (int address : addresses) {
    int blockIndex = (address / blockSize) % numSets;
    int tag = address / (blockSize * numSets);

    boolean hit = false;
    int lruIndex = -1;
    int maxLru = Integer.MIN_VALUE;

    // Check for cache hit
    for (int i = 0; i < associativity; i++) {
        if (cache[blockIndex][i].valid && cache[blockIndex][i].tag == tag) {
            hit = true;
            cacheHits++;

            // Update LRU counter
            cache[blockIndex][i].lruCounter = 0;
            for (int j = 0; j < associativity; j++) {
                if (j != i && cache[blockIndex][j].valid) {
                    cache[blockIndex][j].lruCounter++;
                }
```

```
                }
                System.out.println("Cache hit for address " + address);
                break;
            }
        }

        if (!hit) {
            // Cache miss
            cacheMisses++;
            System.out.println("Cache miss for address " + address);

            // Find block to replace (LRU or FIFO)
            int replaceIndex = -1;
            if (replacementPolicy == LRU) { // LRU
                for (int i = 0; i < associativity; i++) {
                    if (!cache[blockIndex][i].valid) {
                        replaceIndex = i; // Use invalid block if available
                        break;
                    }
                    if (cache[blockIndex][i].lruCounter > maxLru) {
                        maxLru = cache[blockIndex][i].lruCounter;
                        lruIndex = i;
                    }
                }
                if (replaceIndex == -1) {
                    replaceIndex = lruIndex;
                }
            } else { // FIFO
                replaceIndex = fifoPointer[blockIndex];
                fifoPointer[blockIndex] = (fifoPointer[blockIndex] + 1) % associativity;
            }

            // Handle write-back policy
            if (writePolicy == WRITE_BACK && cache[blockIndex][replaceIndex].valid &&
cache[blockIndex][replaceIndex].dirty) {
                // Write back dirty block to memory
                System.out.println("Writing back dirty block to memory (Tag: " +
cache[blockIndex][replaceIndex].tag + ")");
            }

            // Update the replaced block
            cache[blockIndex][replaceIndex].valid = true;
            cache[blockIndex][replaceIndex].tag = tag;
            cache[blockIndex][replaceIndex].dirty = (writePolicy == WRITE_BACK);
```

```
                cache[blockIndex][replaceIndex].lruCounter = 0;

                // Update other blocks' LRU counters
                for (int i = 0; i < associativity; i++) {
                    if (i != replaceIndex && cache[blockIndex][i].valid) {
                        cache[blockIndex][i].lruCounter++;
                    }
                }
            }
        }

        // Display simulation results
        System.out.println("\nSimulation Results:");
        System.out.println("Total Cache Hits: " + cacheHits);
        System.out.println("Total Cache Misses: " + cacheMisses);
        System.out.printf("Hit Rate: %.2f%%\n", ((double) cacheHits / addresses.size()) * 100);
        System.out.printf("Miss Rate: %.2f%%\n", ((double) cacheMisses / addresses.size()) *
100);

    } catch (InputMismatchException e) {
        System.err.println("Invalid input: " + e.getMessage());
    } finally {
        scanner.close();
    }
  }
}
```

OutPut: Enter the cache size (in blocks):
9
Enter the block size (in words):
8
Enter the associativity (1 for direct-mapped, N for N-way):
1
Enter the replacement policy (0 for LRU, 1 for FIFO):
1
Enter the write policy (0 for Write-Through, 1 for Write-Back):
1
Cache miss for address 4
Cache miss for address 8
Writing back dirty block to memory (Tag: 0)
Cache miss for address 16

Writing back dirty block to memory (Tag: 1)
Cache miss for address 4
Writing back dirty block to memory (Tag: 2)
Cache miss for address 20
Writing back dirty block to memory (Tag: 0)
Cache miss for address 8
Writing back dirty block to memory (Tag: 2)
Cache miss for address 24
Writing back dirty block to memory (Tag: 1)
Cache miss for address 32
Writing back dirty block to memory (Tag: 3)
Cache miss for address 4
Writing back dirty block to memory (Tag: 4)
Cache miss for address 20
Writing back dirty block to memory (Tag: 0)

Simulation Results:
Total Cache Hits: 0
Total Cache Misses: 10
Hit Rate: 0.00%
Miss Rate: 100.00%