Code

Runner.java

```java
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.IntBuffer;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Arrays;

public class Runner {

    private final static String FILE_NAME = "gcc1.trace";
    private final static String OUTPUT_FILE_NAME = "subtrace.txt";


    public static void main(String... aArgs) throws IOException, IllegalArgumentException{

        // Start a timer
        final long startTime = System.currentTimeMillis();

        // Read the file
        byte[] bytes = readSmallBinaryFile(FILE_NAME);


        // Create an int array from bytes
        int[] traces = getAsIntArray(bytes, false);
        System.out.println("Number of 32bit ints: " + traces.length + ", number of traces = " + traces.length / 2 + "\n");

        // See how long it takes to read the file
        final long timeAfterRead = System.currentTimeMillis();

        // Get subset of integers for examination
        // int[] littleEndianSnippet = Arrays.copyOfRange(littleEndian, 250000 , 250100);

        // Print out the contents
        // displayIntArray(littleEndian);

        // Write a subset to test file for inspection
        // writeSmallBinaryFile(bytes, OUTPUT_FILE_NAME, 100);


        // Run the assignment simulation
        runAssignment(traces);

        // Run the tutorial
//        runTutorial();

        // Stop the timer
        final long finishTime = System.currentTimeMillis();

        System.out.println("\nFile Reading Time: " + (timeAfterRead - startTime) + "ms");
        System.out.println("Cache Simulation Time: " + (finishTime - timeAfterRead) + "ms");
        System.out.println("Total Elapsed Time: " + (finishTime - startTime) + "ms");

    }


    /*
     *********************************************************
     * Helper Methods
     *********************************************************/

    /**
     * Sets up and runs the simulation for the tutorials (like the online animation)
     */
    public static void runTutorial() {
        int[] tutorialData = {
                0x0000, 0x0004, 0x000c, 0x2200, 0x00d0, 0x00e0, 0x1130, 0x0028, 0x113c, 0x2204,
                0x0010, 0x0020, 0x0004, 0x0040, 0x2208, 0x0008, 0x00a0, 0x0004, 0x1104, 0x0028, 0x000c,
                0x0084, 0x000c, 0x3390, 0x00b0, 0x1100, 0x0028, 0x0064, 0x0070, 0x00d0, 0x0008, 0x3394
        };

        // Pick some parameters here and pick same on website
        Cache iCache = new Cache(32, 8, 4, 8);
        Analyser analyser = new Analyser(iCache, tutorialData);
        analyser.analyseTutorial();
    }
```

```java
/**
 * Sets up and runs the simulation for the assignment
 * @param traces they int array from the tracefile
 */
private static void runAssignment(int[] traces) {
    // Create the caches
    Cache iCache = new Cache(16, 1, 1024);
    Cache dCache = new Cache(16, 8, 256);

    // Analyse the cache
    Analyser analyser = new Analyser(iCache, dCache, traces);
    analyser.analyseAssignment();
}

/**
 * Builds array of 32 bit words from the bytes read from the trace file
 * @param bytes the byte array to get the ints from
 * @param bigEndian true if data is in big endian
 * @return int array
 */
private static int[] getAsIntArray(byte[] bytes, boolean bigEndian) {
    IntBuffer intBuffer = ByteBuffer.wrap(bytes)
            .order(bigEndian ? ByteOrder.BIG_ENDIAN : ByteOrder.LITTLE_ENDIAN)
            .asIntBuffer();
    int[] array = new int[intBuffer.remaining()];
    intBuffer.get(array);
    return array;
}


/**
 * Displays the byte arrays in a specific radix formatted nicely
 * @param bytes array to print
 * @param radix base to print in
 * @param numToShow number of elements from the array to print
 */
private static void displayBytes(byte[] bytes, int radix, int numToShow) {
    System.out.println("Displaying in base " + radix);
    for(int i=0;i<(numToShow*8); i++){

        // Split the byte strings nicely
        if(i % 8 == 0) {
            System.out.println();
        } else if(i % 4 == 0) {
            System.out.print("  -  ");
        }

        // Create the String
        String str = Integer.toString(bytes[i], radix);

        // Pad binary strings with X's
        if(radix == 2){
            str = String.format("%8s", str).replace(' ', 'X');
        }

        System.out.print(str + " ");
    }
    System.out.println("\n");
}


/**
 * Prints the integer array in multiple bases
 * @param arr array to print
 */
private static void displayIntArray(int[] arr) {
    System.out.println("Hex, Dec, Binary");
    for(int i : arr) {
        System.out.println(Integer.toString(i, 16));
        System.out.println(Integer.toString(i));
        System.out.println(Integer.toString(i, 2) + "\n");
    }
}

/**
 * Reads the bytes in from the tracefile
 * @param filename name of tracefile
 * @return byte array containing all bytes in that file (little endian)
 * @throws IOException if anything scary happens when reading the file
 */
private static byte[] readSmallBinaryFile(String filename) throws IOException {
    Path path = Paths.get(filename);

    return Files.readAllBytes(path);
}
```

```
    /**
     * Writes a subset of the bytes to a file for easier viewing
     * @param bytes the full bytes array
     * @param filename name of the output file
     * @throws IOException if something scary happens when writing the file
     */
    private static void writeSmallBinaryFile(byte[] bytes, String filename, int number) throws IOException {
        Path path = Paths.get(filename);
        byte[] snippet = Arrays.copyOfRange(bytes, 0 , number);
        Files.write(path, snippet); //creates, overwrites
    }
}
```

Analyser.java

```
import java.util.Arrays;

class Analyser {

    private final static boolean DEBUG = false;
    private final static int DEBUG_MAX_TRACES_TO_PROCESS = 100;

    private final static int ADDRESS_MASK = 0x007FFFFF;
    private final static int CYCLE_MASK = 0xE0000000;
    private final static int BURST_MASK = 0x18000000;

    private final static int CYCLE_SHIFT = 29;
    private final static int BURST_SHIFT = 27;

    private Cache iCache, dCache;
    private int[] traces;

    /**
     * Creates an analyser for simulations with two caches
     * Assumes traces is [word00, word01, word10, word11 ...etc]
     * Assumes words are in big endian format
     * Assumes wordsX0 lowest (address) byte have not yet been shifted
     * @param iCache the instruction cache to be used
     * @param dCache the data cache to be used
     * @param traces int array of the traces
     */
    Analyser(Cache iCache, Cache dCache, int[] traces) {
        this.iCache = iCache;
        this.dCache = dCache;
        this.traces = traces;
        System.out.println("Traces Length: " + this.traces.length);
    }

    /**
     * Creates an analyser for simulations with one cache
     * @param cache the cache to be used for the simulation
     * @param traces the traces to be used for the simulation
     */
    Analyser(Cache cache, int[] traces) {
        this.iCache = cache;
        this.traces = traces;
    }

    /**
     * Runs the basic (single cache) tutorial simulations
     */
    void analyseTutorial() {
        for(int trace : traces) {
            iCache.feedAddress(trace, 0);
        }

        System.out.println("\nInstruction Cache");
        iCache.printResults();
    }


    /**
     * Runs the more complex assignment (multi cache) simulations
     * @throws IllegalArgumentException if something scary happens
     */
    void analyseAssignment() throws IllegalArgumentException {

        // Keep track of the different cache access to ensure everything seems appropriate
        int skips = 0;
        int dataReads = 0;
        int dataWrites = 0;
        int instructionReads = 0;

        // Keeps track of the frequencies of the different cache accesses
        int[] types = new int[8];
```

```java
    // Process some or all of the traces
    int tracesToAnalyse = DEBUG ?  2 * DEBUG_MAX_TRACES_TO_PROCESS : traces.length;
    System.out.println("Analysing traces: " + tracesToAnalyse / 2);

    // We only actually process half that number of traces since there is two words per trace
    float actualNumTraces = tracesToAnalyse / 2;

    // Iterate over all of the traces
    for(int i=0;i<tracesToAnalyse; i+=2) {

        int word = traces[i];

        // Print the trace
        // String binary = Integer.toString(word, 2);
        // System.out.println(binary.length() + ": " + binary);

        // Extract the relvant bits (three angles is unsigned bit shift)
        int cycleType = (word & CYCLE_MASK) >>> CYCLE_SHIFT;
        int burstCount = (word & BURST_MASK) >>> BURST_SHIFT;

        // Get bits 24 - 2 of the address
        int address = word & ADDRESS_MASK;

        // Shift bits to add the missing two LS zeros
        address <<= 2;


        // Ensure trace was valid
        if(burstCount > 3 || burstCount < 0 || cycleType > 7 || cycleType < 0) {
            String error = "Invalid Trace: cycle type: " + cycleType + ", burstCount: " + burstCount;
            throw new IllegalArgumentException(error);
        }

        // Increment number of these cycles that have occurred
        types[cycleType]++;


        // Feed the address to the appropriate cache
        // 4: instruction read, 6: data read, 7: data write
        if(cycleType == 4) {
            instructionReads++;
            iCache.feedAddress(address, burstCount);
        } else if(cycleType == 6) {
            dataReads ++;
            dCache.feedAddress(address, burstCount);
        } else if(cycleType == 7) {
            dCache.feedAddress(address, burstCount);
            dataWrites ++;
        } else {
            skips++;
        }
    }

    // Print the distributions of cycle types and other info
    System.out.println("\n\nData Reads: " + dataReads * 100 / actualNumTraces + "%");
    System.out.println("Data Writes: " + dataWrites * 100 / actualNumTraces+ "%");
    System.out.println("Instruction Reads: " + instructionReads * 100  / actualNumTraces + "%");
    System.out.println("Skips: " + skips * 100 / actualNumTraces + "%");
    System.out.println("Cycle Types (encoded): " + Arrays.toString(types));
    System.out.println("Analysed " + (instructionReads + dataReads + dataWrites) + " traces, skipped " + skips);

    System.out.println("\nInstruction Cache");
    iCache.printResults();

    System.out.println("\nData Cache");
    dCache.printResults();
}

/**
 * Adds missing bits to LSB of the address
 * This is **NOT** used as I _think_ its the same as the shifting done instead
 */
private int addMissingBitsToStart(int address) {
    // System.out.println("Initial Address: " + Integer.toBinaryString(address));

    int addressUpper = address & 0xFFFFFF00;
    // System.out.println("Address Upper: " + Integer.toBinaryString(addressUpper));

    int addressLower = address & 0x000000FF;
    // System.out.println("Address Lower: " + Integer.toBinaryString(addressLower));

    addressLower <<= 2;
    // System.out.println("Address Lower Shifted: " + Integer.toBinaryString(addressLower));

    address = addressUpper + addressLower;
    // System.out.println("Final Address: " + Integer.toBinaryString(address));
```

```
        return address;
    }
}
```

Cache.java

```
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;

class Cache {

    private static final int DEFAULT_ADDRESS_SIZE = 25;
    private int addressSize = DEFAULT_ADDRESS_SIZE;


    private int l;
    private int k;
    private int n;

    private int setMask;
    private int offsetMask;
    private int tagMask;

    private int setShift;
    private int offsetShift;
    private int tagShift;

    private int hits = 0;
    private int misses = 0;
    private int timestamp = 0;
    private HashMap<Integer, HashMap<Integer, TagData>> sets;

    /**
     * Create a Cache with a specified addressSize
     * @param l number of bytes in each cache line
     * @param k number of cache lines per set (directories)
     * @param n number of sets
     * @param addressSize size of the physical address (required for calculating # bits for offset)
     */
    Cache(int l, int k, int n, int addressSize) {
        this(l, k, n);
        this.addressSize = addressSize;
    }

    /**
     * Creates a cache with default (assignment) address size of 25 bits
     * @param l number of bytes in each cache line
     * @param k number of cache lines per set (directories)
     * @param n number of sets
     */
    Cache(int l, int k,  int n){
        this.l = l;
        this.k = k;
        this.n = n;
        System.out.println("Created " + l + ", " + k + ", " + n + " cache (" + l*k*n + "kb)");

        // Get number of bits for l
        Double math = (Math.log(l) / Math.log(2));
        int offsetBits = math.intValue();

        // Get number of bits for n
        math = Math.log(n) / Math.log(2);
        int setBits = math.intValue();
        System.out.println(offsetBits + ", " + setBits);

        // Offset mask is least significant l bits (after shifting)
        offsetMask = (1 << offsetBits) - 1;
        offsetShift = 0;
        System.out.println("Offset Mask: " + Integer.toBinaryString(offsetMask));

        // Set mask is least significant n bits (after shifting)
        setMask = (1 << setBits) - 1;
        setShift = offsetBits;
        System.out.println("Set Mask: " + Integer.toBinaryString(setMask));

        // Tag mask is least significant remainder of bits (after shifting)
        tagMask = (1 << (addressSize - offsetBits - setBits)) -1;
        tagShift = offsetBits + setBits;
        System.out.println("Tag Mask: " + Integer.toBinaryString(tagMask) + "\n");

        // Instantiate the hash maps for our cache
        sets = new HashMap<>(n, 1);
        for(int i=0; i<n; i++) {
            sets.put(i, new HashMap<>(k, 1));
```

```java
    }
}

/**
 * Requests data from the cache
 * @param physicalAddress "addressSize" bit physical address
 * @param burstCount the number of adjacent memory address requested
 * @throws IllegalArgumentException if the physical address is invalid
 */
void feedAddress(int physicalAddress, int burstCount) throws IllegalArgumentException{

    // Extract the info
    int setNumber = (physicalAddress >> setShift) & setMask;
    int offset = (physicalAddress >> offsetShift) & offsetMask;
    int tagNumber = (physicalAddress >> tagShift) & tagMask;

    // Ensure values are valid
    if(setNumber < 0 || setNumber > setMask || offset < 0 || offset > offsetMask || tagNumber < 0 || tagNumber > tagMask
        String error = "Invalid Cache Values: setNumber: " + setNumber+ ", tagNumber: " + tagNumber + ", offset: " + offs
        throw new IllegalArgumentException(error);
    }

    // Display the values
    // System.out.println("\n" + Integer.toHexString(physicalAddress));
    // System.out.println("Set: " + setNumber + ", Offset: " + offset + ", Tag: " + tagNumber + ", Burstcount: " +burstC

    // Get the k tags in this set
    HashMap<Integer, TagData> set = sets.get(setNumber);

    // Check for the tag we are interested in
    TagData tagData = set.get(tagNumber);

    // Check for tag match
    if(tagData != null) {
        // System.out.println(Integer.toHexString(physicalAddress) + ": Hit found");
        hits += burstCount + 1;

        // Update this tags last access time
        tagData.lastAccess = ++timestamp;
    } else {
        // System.out.println(Integer.toHexString(physicalAddress) + ": Miss");
        misses++;

        // Think the memory address that were adjacent count as hits since they will technically be read from the cache?
        hits += burstCount;

        // Check if k directories are full
        if(set.values().size() < k) {
            // System.out.println("Compulsory miss - k directories not full, inserting..");
            set.put(tagNumber, new TagData(++timestamp));
        } else {
            int lruTag = getLRU(set);
            // System.out.println(Integer.toHexString(physicalAddress) + ": directory full, removing lru: " + lruTag);
            set.remove(lruTag);
            set.put(tagNumber, new TagData(++timestamp));
        }
    }
}


/**
 * Performs the least recently used algorithm
 * @param set the set from the cache that needs to have an eviction
 * @return the tag to be evicted
 */
private int getLRU(HashMap<Integer, TagData> set) {

    // Find the tag with the smallest timestamp as its last access
    int minAccess = Integer.MAX_VALUE;
    int lruTag = -1;
    for(Map.Entry<Integer, TagData> mapEntry : set.entrySet()) {
        int lastAccess = mapEntry.getValue().lastAccess;
        if(lastAccess < minAccess) {
            minAccess = lastAccess;
            lruTag = mapEntry.getKey();
        }
    }

    return lruTag;
}

/**
 * Prints the results of the simulation
 */
void printResults() {
    System.out.println("Total accesses: " + (misses + hits));
    System.out.println("Misses: " + misses);
    System.out.println("Hits: " + hits);
```

```
        System.out.println("Hit Rate: " + (float)hits * 100 / (hits + misses) + "%");
    }

    /**
     * Data structure which could be expanded to actually hold some data
     */
    class TagData {

        int lastAccess;

        TagData() {
            this.lastAccess = 0;
        }

        TagData(int lastAccess) {
            this.lastAccess = lastAccess;
        }
    }
}
```