COMP25212 Cache Memory Simulation
Lab Sessions 2, 3, 4

Duration: 3 two-hour lab sessions

Resources: Any computer having a Java compiler & JVM

AIMS

To investigate, by simulation, cache memory performance.

LEARNING OUTCOMES

• To understand the use and performance of cache systems

• To have further experience of developing java programs

DELIVERABLES

• Demonstrate your implementation of a direct mapped cache - lab session 3

• Demonstrate your fully-associative cache - lab session 4

INTRODUCTION TO CACHE MEMORY

This introduction is inevitably rather brief. It is supplemented by early COMP25212 lectures and by Chapter 10, especially section 10.3 Caches, of Steve Furber's book 'ARM system-on-chip architecture' 2nd edn Addison Wesley 2000. The diagrams here are from Furber and reproduced with permission.

A cache is literally a 'secret hiding place'. Cache techniques, both hardware and software, are used extensively in computer systems. The lab concerns cache memory which is always implemented in hardware.

Cache memory is a small amount of very fast memory used as temporary store for frequently used instruction and data memory locations. It is used because access time for the main RAM memory is slow compared with instruction execution speeds. Avoiding main memory accesses wherever possible, both for instruction fetching and for data accessing, improves overall performance. Cache memory relies on the fact that, in most well behaved large programs, only a small proportion of the total code and data are accessed during any short time interval. If we could keep in cache memory those fragments of code and data that are currently being used, then we could improve performance. A 'unified' cache is one used for both instructions and data. We will also consider later the use of separate caches for instructions and data. Under some circumstances two

separate caches can perform better than a single unified cache..

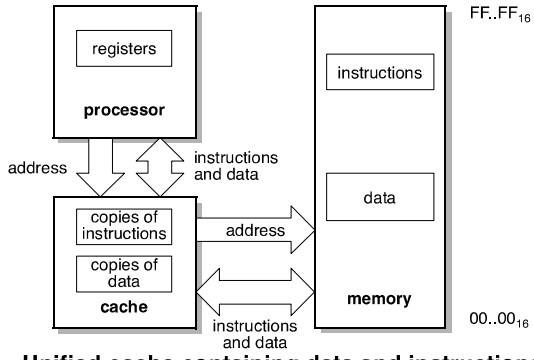
On each memory read (including instruction fetches) the required item is first looked for in the cache. This requires a comparison between the desired memory address and the addresses of items in the cache. If the required address is found this is a cache 'hit'. The item is accessed from the cache and no main memory access is necessary. Only if the item is not present in the cache is main memory accessed. When main memory is accessed the item is copied into the cache, in anticipation of possible future accesses. This is turn may require the rejection of some existing item in the cache to make space for the new item.

Each entry in the cache, comprising data and addressing information, is known as a cache line. In order to compare the desired memory address and the addresses of items in the cache it is necessary for each cache line to have typically three fields. The first field is address information, sometimes called a 'tag' field which enables us to find the correct data. The second field is the data field which, depending on the cache line size, may contain several words of data including the data item actually required. The third field is a Boolean 'valid' bit to indicate whether the line contains currently valid data.

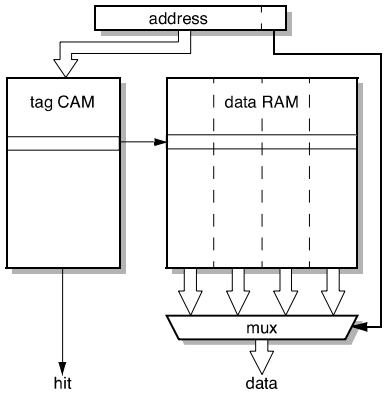
There are several ways of organising and using these cache fields. We consider fully associative, direct mapped and set associative organisations.

Fully Associative cache

In the fully associative model the tag field is a full byte address (32 bits in our case) truncated to a multiple of a power of two depending on the cache line size. So with a line size of 4 bytes the bottom 2 bits can be discarded, with 8 bytes the bottom 3 bits etc.. Any data can occupy any line of the cache. In order to find



Unified cache containing data and instructions (Furber)



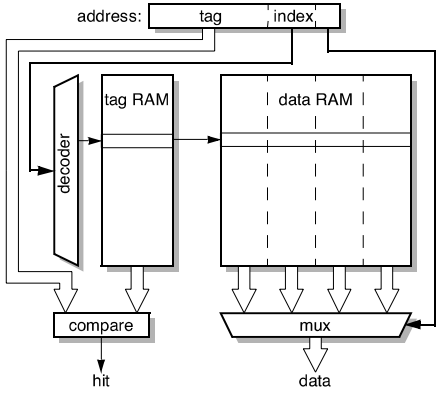
Fully associative cache organisation

whether an item is in the cache the presented address, suitably truncated, is compared with the tag field of each valid line. In hardware, speed is achieved by comparing all tag fields simultaneously in parallel using associative memory (sometimes called content addressed memory or CAM). If the required address is found this is a cache 'hit'. If the item is an instruction to be fetched or data to be read, the required value is read from the data field of the cache and the operation is complete.

If there is a cache miss then the data is read from main memory. Enough data to fill the complete cache line is copied into the cache, in anticipation of possible future accesses, and the corresponding tag and valid fields updated.

In a fully associative cache any item may occupy any line so a line must be chosen for the new item. If the chosen line is already valid the item already in the cache must be rejected to make space for the new item. Various rejection algorithms can be used, we will use a simple cyclic algorithm where each cache line is rejected in turn.

If the item is to be written the situation is more complicated. The basic technique we will use is to first establish whether the address to be written is in the cache. If it is then the item can be written to the cache. If it is not it will **not** be written to the cache. In both cases it will be written to main memory so that the memory is up to date as soon as the write takes place, regardless of whether the written item is cached. This strategy, called "write through with no allocate", is one of several possible writing strategies. In this strategy the write to memory is usually buffered so that it incurs no cost, except when memory contention occurs. If the



Direct-mapped cache organization

item is subsequently read, it will be cached at that point.

Direct mapped cache

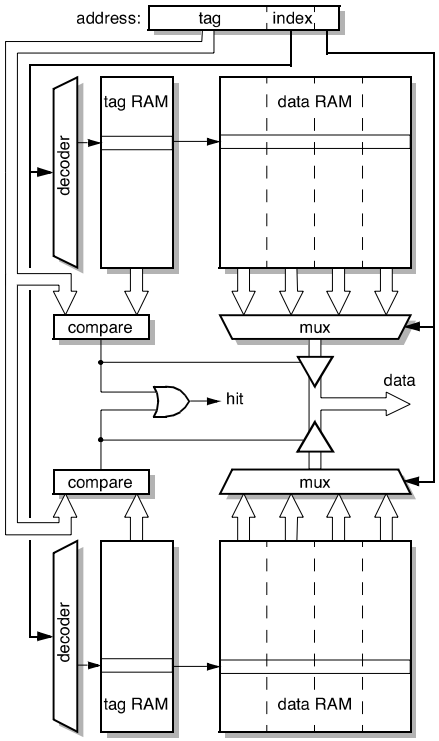
In a direct mapped cache the address is truncated, depending on the cache line size, as before. The truncated address is divided into two parts. The part called the index, is used to address a cache line directly and the rest is the tag which is stored along with the data. The size of the index depends on the number of lines in the cache, which must be a power of two. So, for example, a 32 line cache would use 5 address bits to select a cache line and the remaining bits would be the tag.

When a new address is presented to the cache, the index is formed and used to address the cache. However, the selected line may contain one of many tag values and so the stored tag must be compared with the tag formed from the presented address. If they are the same then the stored entry must have the same overall address and this is thus a cache hit, otherwise it is a miss. On a miss there is no choice about which cache line must be used, the data must be loaded into the selected line and the tag field updated. The technique on write is similar to that for the fully associative cache.

A direct mapped cache is cheaper to implement than a fully associative cache. It is really just a hash table implemented in hardware. The disadvantage is the probability of a lower proportion of 'hits' to total accesses (the hit rate) because of the inflexible replacement strategy

Set associative cache

A widely implemented compromise between the cost of an associative cache and the disadvantage of a direct mapped cache is a set associative cache. This is a combination of the two previous techniques. The set associative cache is implemented as a small number of direct mapped caches operating in parallel. An n-way set associative cache (where n might be typically 2 or 4) uses n direct mapped caches. An item might be placed in the relevant line of any of the direct mapped caches. So finding an item involves looking at the relevant line in all the the caches and doing an associative search on the tag fields to find which is the correct one.



Two-way Set Associative Cache

Set associative caches are covered in the lectures but will not be investigated in this practical exercise.

THE PROBLEM

The objective of this lab is to write a simple simulation, of the operation of a cache memory, in Java. This will then be used to compare the performance of various cache organisations, in particular the hit rates achieved.

A stream of addresses will be fed to your simulator from a trace file, which we prepared earlier by monitoring memory accesses made by a real program. Each of the addresses in the trace file is accompanied by information describing the type of access being made - data read, data write or instruction fetch. Only memory accesses are to be simulated; all other aspects of the behaviour of the program which produced the trace will be ignored. So the simulation will maintain the tag fields in the caches correctly. However the actual data read and written is not to be considered, no real data will be written to the data field of the caches. Main memory will not be simulated in any way, any mention of notional memory accesses below can just refer to a dummy data value.

The problem will be tackled in two stages with an intermediate deadline as follows:

- Understand the given code and run it on simple data in conjunction with the 'Secret' cache provided
- Implement a Direct mapped cache and test it on simple data (**Deliverable** in Lab Session 3)
- Implement a Fully Associative cache and test it on simple data (**Deliverable** in lab session 4)
- Compare and understand the comparative perfomance of the implemented caches with various cache configurations

The implementation will allow the total cache size and the cache line size to be any power of 2 allowing the performance of different cache sizes to be compared. It will also allow the possiblity of a single unified cache, containing both program instructions and data, or separate data and instruction caches.

Two trace files are provided to be used as follows:

- traceA: a simple artificial file intended for testing of your code
- traceB: is a real trace file of 40000 entries which should give meaningful results and which will be used for perfomance comparisons.

Ex. 25212.2 Direct Mapped Cache (lab sessions 2 & 3)

Preparation: Understand the given code

Copy the given set of files from /opt/info/courses/COMP25212/cachelab to your own COMP25212/cachelab directory. These are listed for convenience at the end of this lab script. The lab exercises build cumulatively on the given code; you may find it easiest to work in your COMP25212/cachelab directory throughout but labprint and submit will require the relevant files in separate directories ~/COMP25212/ex2 and ~/COMP25212/ex3.

In the file Cache.java is the definition of an abstract class Cache which has 3 abstract methods: **cacheSearch**, **cacheNewEntry** and **cacheWriteData**. In this lab exercises you will write a class which implements DirectMappedCache and in the next lab, AssocCache. For each of these you will need to define appropriate internal data structures, write a constructor to initialise these structures and design and code the bodies of the methods.

Cache.java also has a number of variables intended for collection of statistics, a method dumpStats to print these statistics and a number of methods to return individual statistics if required.

In the file CacheSim.java you are provided with a skeleton main method which opens the trace file provided as the first argument of the program run as:

java CacheSim tracefile

The skeleton main method then calls the constructor to create a cache (initially a 'secret' cache with a total size of 1024 bytes and a line size of 4 bytes) and calls the doSimul method using the created cache as a unified cache for both data and instructions.

doSimul calls getAccess to return the next MemAccess, comprising an atype (Read, Write or Fetch) and an addr, from the trace file. If the atype is a Read or Write the data cache is accessed, if a Fetch the Instruction cache. doSimul is written so that it will work equally well with a unified cache or with separate data and instruction caches. At the end of the trace file MemAccess returns a null pointer which is caught in doSimul to end the simulation run and call dumpStats to print the accumulated statistics. You do not need to understand the detailed working of getAccess.

The major functionality of the Cache class is provided by the methods **read** and **write** which access the cache. These are common for all cache types but call the methods **cacheSearch**, **cacheNewEntry** and **cacheWriteData** which perform the detailed operations for a particular cache type. Each cache type should therefore be a class which extends the Cache class and implements the versions of the above abstract methods.

The following descriptions of read and write are given to aid understanding.

public Object read (int addr)

Calls **cacheSearch** to find if the item addressed by **addr** is in the cache. If it is, then the entry is returned. Otherwise **cacheSearch** returns null and **read** calls **cacheNewEntry** to allocate the cache entry (notionally reading main memory in the process). Cache statistics are updated appropriately according to the results returned by **cacheSearch** and **cacheNewEntry**. The data **Object** accessed is returned although this does not contain any valid value and is not actually used.

As **addr** is a java **int** you may assume throughout that all addresses are 32 bits.

public void write (int addr, Object indata)

Calls **cacheSearch** to find if the address, **addr**, is in the cache. If it is, **cacheWriteData** is called to update the cache data otherwise the cache is left unchanged (implementing write through without load). The statistics are updated appropriately. For simplicity. it is assumed that the write through is buffered and that there is no memory contention so writing to memory does not incur any cost. Hence writes are not counted in the total of memory accesses. The data written is the Object **indata** but, as this is not actually used anywhere, the calls to write within doSimul use a dummy Integer(0) object.

Make sure that you understand fully the rest of the given code. Ask if anything is unclear.

Here is a brief specification of the 3 methods for which you have not been given any source code:

public Object cacheSearch (int addr)

Identifies whether the item at address **addr** is in the cache. Returns **null** if it is not. Otherwise returns an Object set to the value of the notional data. If a valid entry is found, its position should be remembered for use by **cacheWriteData** if it called subsequently by a **write** operation.

public oldCacheLineInfo cacheNewEntry (int addr)

Allocates the new entry to the appropriate cache line to the presented address **addr.** Notes the existing value of the valid bit in the **old_valid** field of the result **oldCacheLineInfo**. Sets all 3 fields of the cache line, notionally accessing main

memory to get the data value. Returns the old data value in the **data** field of the result so that it can be notionally written to main memory if required. The **read** method notes rejections in the statistics but the old data can be ignored as we are not concerned with actual data values.

public void cacheWriteData (Object data)

Simply updates the **data** field of the cache line using the **data Object** supplied as parameter. It assumes the cache line to be the same line that was found during the last search.

These 3 methods do not update any Cache statistics.

You are provided with the precompiled SecretCache.class which extends Cache.java and implements **cacheSearch**, **cacheNewEntry** and **cacheWriteData** for an undisclosed cache type. This is provided so that you can observe how the program should behave when given a valid cache implementation. The exact statistics will however vary with the cache type.

Running the Secret Cache Simulation

Compile the program CacheSim.java supplied remembering to leave the SecretCache.class file untouched.

Run the code against the file traceA using the command:

java CacheSim traceA

The code first uses a cache of size 32 bytes with a line size of 8 bytes, then a cache size of 8192 bytes with a line size of 32 bytes. The results should be as follows:

:

Table 1: Unified Cache - Secret

Cache Size	32	8192
Line Size	8	32
Total accesses	16	16
Total read accesses	14	14
Total write accesses	2	2
Read Misses	7	3
Write Misses	1	1
Total rejections	3	1
Total main memory accesses	7	3
Read Miss Rate	50.0%	21.4%
Write Miss Rate	50.0%	50.0%

Make sure that you understand what each of these results means. Ask if you do not.

Implementing the Direct Mapped Cache

The objective is to design, implement and test the class DirectMappedCache. Direct mapped is the simplest cache to implement so is tackled first.

To do this you will need to define appropriate internal data structures, write a constructor to initialise these structures and design and code the bodies of the 3 methods, **cacheSearch**, **cacheNewEntry** and **cacheWriteData**.

You clearly need a tag field and a valid field for each cache line. It is probably easiest to define each of these as an array which is created and initialised by the constructor. The data field is not strictly needed for these simulations but for completeness and generality it might be represented as an array of Objects.

For the direct mapped cache the useful part of the presented address is divided into 2 parts as shown in the table below. The least significant bits would be used to select a byte within a cache line and play no part until the actual data is accessed which is not part of our simulation. These bits can therefore be ignored. The number of bits to be ignored depends on the number of bytes in each cache line. The cache line is selected using the index field which we choose to be the

least significant non-ignored bits. The number of bits in the index field depends

Table 2: Division of (32 bit) address for direct mapped cache

tag	index	ignored

on the number of cache lines. Once the correct cache line is selected the cache tag field can be compared with the tag which is the remaining most significant bits of the presented address. In java the fields are most conveniently selected using suitable integer division and modulo operations (**you do not need to use log, exp or shift operations**). Note that the size of the cache in bytes and the size of each line in bytes are supplied as a variable parameters. They should be supplied to the constructor which can then deduce the number of lines. It is a mistake to build in fixed numbers.

The **cacheSearch** method can now be simply implemented to check whether the required data is present and return null if it is not. It can be assumed that **cacheSearch** will always be called before any call to **cacheWriteData** and that **cacheSearch** notes which line is to be used if the latter is called.

The **cacheNewEntry** method returns the previous value of the valid bit as well as updating the cache line. **cacheWriteData** merely updates the data field of the noted line.

Modify the main method in CacheSim.java to add calls to create and test a unified DirectMappedCache with the same cache and line sizes as in the previous exercise. Test your class using CacheSim.java and the trace file traceA. The results should be as follows

Cache Size	32	8192
Line Size	8	32
Total accesses	16	16
Total read accesses	14	14
Total write accesses	2	2
Read Misses	9	3
Write Misses	2	1
Total rejections	5	0
Total main memory accesses	9	3
Read Miss Rate	64.29%	21.4%
Write Miss Rate	100.0%	50.0%

Marking

During marking, you will be expected to run your program with traceB. When you have completed the exercise, you should run **submit** from within your COMP25212/ex2 directory which must contain the file DirectMapedCache.java. Before getting the exercise marked, you should run **labprint** from within the same directory and have the output available during marking.

Marks will be given for correct implementation and overall correct results for traceB. A small number of marks will be awarded for program style which will reflect concise efficient coding, sensible use of names and neat layout.

Feature	Mark
Correct Data Declarations	2
Correct Initialisation	2
Correct search method	4
Correct new entry method	4
Correct write data method	2
Correct traceB result	4
Overall style	2
Total	20

Ex. 25212.3 Fully Associative Cache (lab session 4)

The objective of this lab exercise is to implement the class AssocCache.java as a fully associative cache. This must be done without changing Cache.java which must continue to work with any type of cache.

You are advised to implement the internal data stuctures using arrays. The constructor performs initialisation similar to that for the direct mapped cache.

The tag field will now comprise all the address bits except the ignored bits.

Table 3: Division of address for fully associative cache

tag	ignored

cacheSearch must now look at the tag and valid bits of every line of the cache until a match is found. If none is found null is returned. Otherwise the data is returned and the position in the cache noted. Obviously in a real associative cache, the search of every line is done in parallel by hardware. In this simulation, a linear search is appropriate.

cacheNewEntry will put the given entry into the next line given by the cyclic rejection and allocation algorithm. This implies that an index must cycle through the cache entries. It will be initialised in the constructor and updated each time **cacheNewEntry** is called.

cacheWriteData will update the data field of the cache line previously noted by search.

Again you should modify the main method in CacheSim to create and test a unified AssocCache.java using the same cache and line sizes as previously.

Run with traceA. The results should be:

Cache Size	32	8192
Line Size	8	32
Total accesses	16	16
Total read accesses	14	14
Total write accesses	2	2
Read Misses	7	3
Write Misses	1	1
Total rejections	3	0
Total main memory accesses	7	3
Read Miss Rate	50.0%	21.4%
Write Miss Rate	50.0%	50.0%

You should also run your Associative Cache with traceB, note the miss rates which occur and compare them with the traceB results from your Direct Mapped Cache. During the marking process, you will be asked to run your simulation with traceB. When you have completed the exercise, you should run **submit** from within your COMP25212/ex3 directory which must contain the file AssocCache.java. Before getting the exercise marked, you should run **labprint** from within the same directory and have the output available during marking.

Marks will be given for correct implementation and overall correct results for traceB. A small number of marks will be awarded for program style which will reflect concise efficient coding, sensible use of names and neat layout

You should also be prepared to describe the results of your comparison between the two types of cache and draw any appropriate conclusions.

Feature	Mark
Correct Data Declarations	**1**
Correct Initialisation	**1**
Correct search method	**4**
Correct new entry method	**4**
Correct write data method	**1**
Correct traceB result	**4**
Comparison & Conclusions	**3**
Overall style	**2**
Total	**20**

Appendix: Listings of the given code

These listings were correct at the time of printing the manual. However, if changes or corrections prove necessary, the most up-to-date version will always be that online at opt/info/courses/COMP25212/cachelab

CacheSim.java

```
import java.io.*;
import java.util.*;

class MemAccess {
  //obtained from trace file by getAccess
  //atype is coded: Read = 0, Write = 1, Fetch = 2;
    int atype;
    int addr;
}

public class CacheSim {

    static BufferedReader in;
    final static int Read = 0, Write = 1, Fetch = 2;
    static int inst_fetches;
    static int csize,lsize; //total cache size and line size in bytes

    public static void main(String args[]){
        try {
         in = new BufferedReader(new FileReader(args[0]));
           in.mark(1000000);//needed to repeatedly read data file
         csize =32; //small cache for initial tests
           lsize =8;
         System.out.println("Unified Cache - Secret");
         Cache UnifiedCache = new SecretCache(csize,lsize);
         doSimul(UnifiedCache,UnifiedCache,true);

         csize =8192; //larger cache for real experiments
           lsize =32;
         System.out.println("Unified Cache - Secret");
         UnifiedCache = new SecretCache(csize,lsize);
         doSimul(UnifiedCache,UnifiedCache,true);
```

```
        } catch (FileNotFoundException e) {
          System.out.println("File "+args[0]+" Not Found");
    } catch (IOException ioe) {
          System.out.println("File Mark Failed");
    }
   }

            private static void doSimul(Cache DataCache, Cache
InstructionCache, boolean printing) {
    //For Unified cache make DataCache and InstructionCache the same
      //If printing is true statistics are printed
      System.out.println("Cache Size = "+csize+" Line Size = "+lsize);
    inst_fetches = 0;
    try{
       in.reset(); //read complete data file from start for each call of
doSimul
    } catch (IOException ioe) {
          System.out.println("File Reset Failed");
          return;
    }
      try {
       while(true) {
        MemAccess access = getAccess();
        if (access.atype == Read){
            //  System.out.println(Int.toHexString(access.addr)+" Read");
            DataCache.read(access.addr);
        }
        else if (access.atype == Write){
            //  System.out.println(Int.toHexString(access.addr)+" Write");
            DataCache.write(access.addr,new Integer(0));
        }
        else if (access.atype == Fetch){
            inst_fetches++;
            //  System.out.println(Int.toHexString(access.addr)+" Fetch");
            InstructionCache.read(access.addr);
        }
        else System.out.println("Unknown Access Type");
       }
    } catch (NullPointerException e) {
      //end of trace file and simulation run
      if (printing) {
```



```
        System.out.println("Trace Input Ended");
        if (InstructionCache == DataCache){
            System.out.println("\nUnified Cache Statistics\n");
            InstructionCache.dumpStats();
        }
        else {
            System.out.println("\nInstruction Cache Statistics\n");
            InstructionCache.dumpStats();
            System.out.println("Data Cache Statistics\n");
            DataCache.dumpStats();
        }

       }
      }
    }

    private static MemAccess getAccess() throws NullPointerException {
     // return values from next line of trace file or null at end of file
      try {
        MemAccess res = new MemAccess();
        StringTokenizer st = new StringTokenizer(in.readLine());
        res.atype = Integer.valueOf(st.nextToken()).intValue();
        res.addr = Integer.valueOf(st.nextToken(),16).intValue();
        return res;
     } catch (IOException e) {
        System.out.println("An unexpected I/O exception occured");
        return null;
     }
    }

}
```

Cache.java

```
public abstract class Cache{
   int size,line_size;//sizes in bytes initialised by constructor of subclass
   // counters used to generate performance statistics
    int total_accesses, read_accesses, write_accesses; //all total numbers
hit or miss
   // read includes read and fetch
   int read_misses, write_misses, rejections, main_mem_accesses;
   // rejections is when a valid line in the cache is replaced by a new line

   Cache(){
      //initialise statistics when new cache created
    total_accesses = 0;
    read_accesses = 0;
    write_accesses = 0;
    read_misses = 0;
      write_misses = 0;
    rejections = 0;
    main_mem_accesses = 0;
   }

  public void write(int addr, Object indata) {
      total_accesses++;
      write_accesses++;
      Object entry = cacheSearch(addr);
      if (entry != null) cacheWriteData(indata);
      else {
          write_misses++;
      }
  }

  public Object read(int addr) {
      total_accesses++;
      read_accesses++;
      Object entry = cacheSearch(addr);
      if (entry != null) return entry;
      else {
          read_misses++;
          main_mem_accesses++;
          oldCacheLineInfo old_line_info = cacheNewEntry(addr);
```



```
          if (old_line_info.old_valid) rejections++;  // the old valid value
        return old_line_info.data;
      }
  }

   abstract Object cacheSearch(int addr);
   abstract oldCacheLineInfo cacheNewEntry(int addr);
   abstract void cacheWriteData(Object entry);

   public void dumpStats(){
          System.out.println("Total accesses = "+ total_accesses);
          System.out.println("Total read accesses = "+ read_accesses);
          System.out.println("Total write accesses = "+ write_accesses);
          System.out.println("Read misses = "+ read_misses);
          System.out.println("Write misses = "+ write_misses);
          System.out.println("Total rejections = "+ rejections);
                  System.out.println("Total main memory accesses = "+
main_mem_accesses);
          System.out.println("Read miss rate = "+(float)100*read_misses/
read_accesses+"%");
          System.out.println("Write miss rate = "+(float)100*write_misses/
write_accesses+"%\n");
   }

   public int totalAccesses() {
    return total_accesses;
   }
   public int readAccesses() {
    return read_accesses;
   }
   public int writeAccesses() {
    return write_accesses;
   }
   public int readMisses() {
    return read_misses;
   }
   public int writeMisses() {
    return write_misses;
   }
   public int numRejections() {
    return rejections;
```

```
    }
    public int mainMemAccesses() {
     return main_mem_accesses;
    }
}
```

oldCacheLineInfo.java

```
public class oldCacheLineInfo {
    Object data;
    boolean old_valid;
}
```

trace A

```
2 20000100
0 10000010
2 20000104
0 10000014
2 20000108
0 10000020
2 2000010C
0 10000024
2 20000110
0 10000030
2 20000114
1 10000030
2 20000118
0 10000030
2 2000011C
1 10000130
```