

**Computer Architecture 3**

**Formal Element: Cache Simulator**

Lecturer: Dr Raymond Lynch

Student Name: Jack Harding

Student Number: C15441798

Date: 8 November 2018

**Contents**

[Introduction 3](#_Toc529466053)

[Fully Associative 4](#_Toc529466054)

[Direct Mapped 5](#_Toc529466055)

[2-Way Associative 5](#_Toc529466056)

[Objective 6](#_Toc529466057)

[Direct Mapped 6](#_Toc529466058)

[Code 6](#_Toc529466059)

[Results 8](#_Toc529466060)

[2-Way Associative 8](#_Toc529466062)

[Code 8](#_Toc529466063)

[Results 10](#_Toc529466066)

[Fully Associative 10](#_Toc529466067)

[Code 10](#_Toc529466068)

[Results 12](#_Toc529466071)

[Conclusion 12](#_Toc529466072)

[References 13](#_Toc529466073)

[Figure 1 CPU-DRAM Speeds 3](#_Toc529466079)

[Figure 2 Fully Associative Diagram 4](#_Toc529466080)

[Figure 3 Direct Mapped Diagram 4](#_Toc529466081)

[Figure 4 2-Way Associative Diagram 5](#_Toc529466082)

[Figure 5 Direct Mapped Hit/Miss 7](#_Toc529466083)

[Figure 7 2-Way Set Associative Results 9](#_Toc529466084)

[Figure 6 Fully Associative Results 11](#_Toc529466085)

# Introduction

Cache is small, high-speed memory in a computer. It is used to increase efficiency and is usually found between CPU and RAM, its efficiency depends on the hit ratio, higher being more efficient. Due to DRAM’s relatively low speed, cache is essential for high-speed computing but is expensive to manufacture when compared to DRAM. The need for cache grew in the 1980s with the divergence of CPU and DRAM speeds, CPUs got faster at much higher rate requiring a faster memory type to act as a buffer.

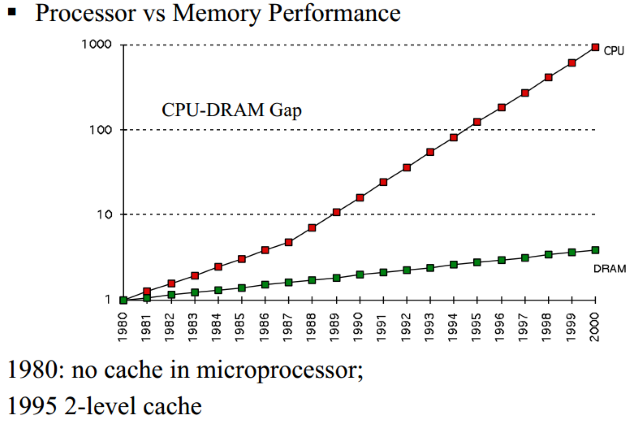


Figure 1 CPU-DRAM Speeds

There are multiple levels of cache in modern processors, usually three, each level is denoted L1-L3, L1 being the fasted but also the smallest, and L3 having larger storage but lower speed. Typically, in modern multicore processors each core has its own L1 and L2 cache with the larger L3 shared between all cores. In the past cache could be located externally from the CPU (in the motherboard) but has since changed. [3]

To make good use of cache, Locality of Reference (LOR) must be used. LOR allows the cache to store subsequent instructions while waiting on the CPU to execute current instructions. The type of LOR used here is Spatial LOR, when fetching an instruction from memory, the next few instructions should be brought on the cache line (4-line cache is this case) with the four instructions being stored in cache.

A hit in cache could be defined by the data being found immediately, without further loading. A miss is where the data is not found in cache and must be read from memory. A miss can either cause long delays or worse yet a crash.

The impact hit rates have on computer performance are significant, typically an L1 cache achieves between 95-97%, the difference of two percent translates to far higher potential latency, a miss meaning the instruction must be fetched from the memory, significantly increasing the time taken to execute the code.

## Fully Associative

The memory address requested by the processor is compared to all the cache lines simultaneously, a block in memory can be assigned to any slot in the cache. When a request arrives for a memory address, all tag fields are searched for a match, if the tag is found a hit occurs, if not a miss.

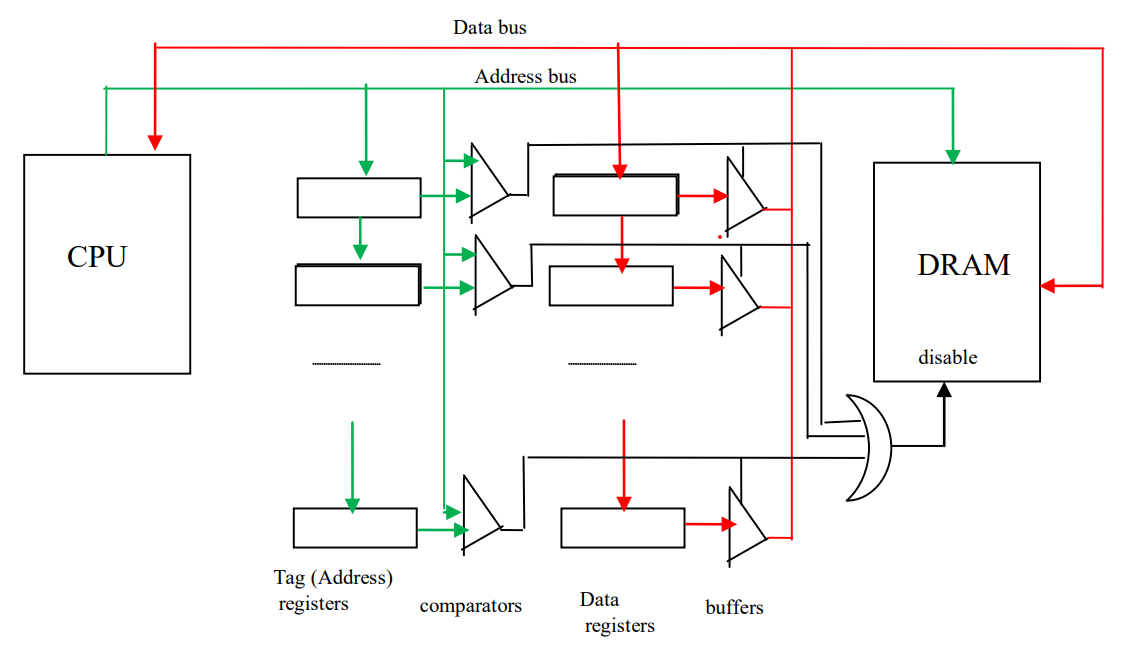


Figure 2 Fully Associative Diagram

Fully associative offers great performance but is very expensive due to the number of components, each line requires a comparator meaning far greater complexity and overall cost of production. Fully associative cache is used practically in TLB (Translation Lookaside Buffer) but only in very small amounts. TLB is required when translating virtual addresses to physical addresses, it greatly improves speed as well as supporting multiple users on a computer with read and write bits.

## Direct Mapped

Direct mapped differs from fully associative in that each block can only go into one cache line, this serves as an advantage compared to fully associative in that the desired block can only be on one line. For direct mapped cache the address is split into the lower and upper halves. The lower half is as index for the tag RAM, the contents of current upper address is compared to the old address, if they match a hit occurs.

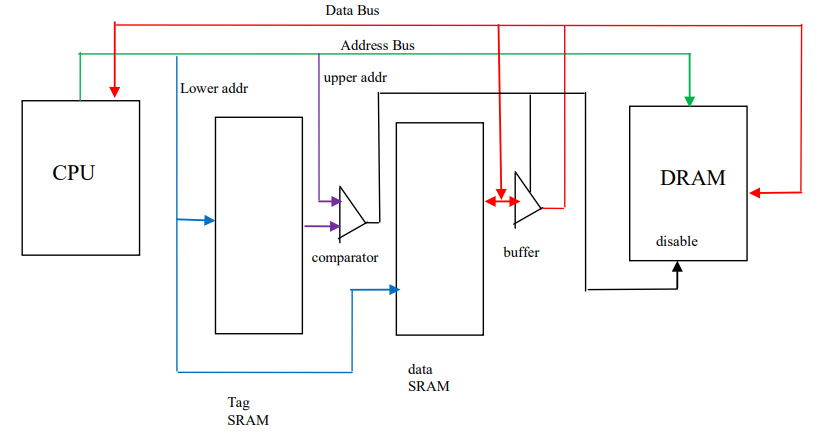


Figure 3 Direct Mapped Diagram

Direct mapped is cheaper and faster than full associative but a drawback is conflict miss. Conflict miss is when to two addresses correspond to a single entry in cache, even stale entries within cache cannot be evicted to make room because the location in cache is determined by the address (lower hit rate).

## 2-Way Associative

Direct mapped cache has a low hit rate therefore increasing fetch time, negatively affecting performance. A solution to this is to use an additional direct mapped cache which is essentially stacked on top of the other direct mapped cache, as shown in Figure 4.

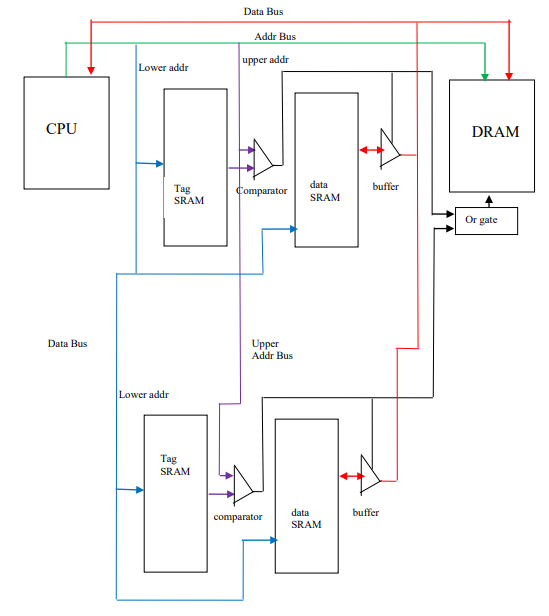


Figure 4 2-Way Associative Diagram

Direct mapped has too many misses and fully associative is too complex, 2-way set associative fits between the two with higher hit rates and almost as cheap implementation. The higher hit rates are due to more blocks allowing previously loaded addresses to be stored. 2-way can be used in some L1 cache which requires minimal loading time but where hit rate is not paramount, with higher sets used for the other cache levels.

# Objective

The objective of this report is to simulate instructions loaded into a cache in a C++ script. The addresses are to be input from a text file containing roughly ten addresses, the same applies for the data. The types used include:

* Direct Mapped
* Associative
* 2-Way Set Associative

As each address is read it is compared to the current cache entries, the success of the cache loading is printed using a hit and miss counter. Along with the hit/mis, the set number, byte number and tag number are to be printed.

# Direct Mapped

## Code

The code below shows a structure used to store all the parameters of a typical cache entry. The *upprAddr* is the tag address used to compare the cache blocks. The *validFlag* is used to ensures that each cache entry is updated and valid. *LRUFlag* is used to evict the least recently used cache entries in set-associative cache. *data[4]* is used as a buffer for four 8-bit entries.

struct CacheEntry {

    unsigned short upprAddr; // 16 bit uppr addr

    bool validFlag; // prevents same copy, flag is set if data valid

    bool LRUFlag; // Least Recently Used

    char data[4]; // 32 bit data bus

};

A cache entry is created for each address read in from the text file. The code below shows the variable declarations and initialisations inside the main function. The *CPUAddr* is the full 32-bit address. *CPULwr* and *CPUUppr* are the 16-bit lower and upper addresses split before entering the tag RAM. *setNo* is the specific cache entry. There are hit and miss counters initialized to zero.

unsigned int CPUAddr; // 32 bit CPU Addr

unsigned short CPULwr; // lwr half

unsigned short CPUUppr; // uppr half

unsigned short setNo; // which element in way0[]

unsigned short byteNo; // indicates byte

unsigned int hits = 0, misses = 0; // hit miss counter

ifstream myAddr("testAddresses.txt"); // addr text file used

CacheEntry way0[16384]; // 2^14 cache entries

string index; // used to store line

*ifstream* is used to add the file containing the address to the code, a *CacheEntry* object is created called *way0* which stores all cache entries.

The first line below loops while it has not reached the end of the file (eof), the address read in is converted to hex and stored in the *CPUAddr* variable. *CPUAddr* is anded with 0000FFFF to give the lower 16-bits, the same is done for the *CPUUppr* but with FFFF0000 and bit shifted by 16. *setNo* is anded with FFFC and bit shifted to give a 14-bit address.

while (!myAddr.eof()) { //while the end of file is NOT reached

stringstream lineIn(sAddr); // 1st line of address file

    myAddr >> hex >> CPUAddr; // line treated as hex

    CPULwr = CPUAddr & 0x0000ffff; // lwr half anded with ffff to give lwr

    CPUUppr = (CPUAddr & 0xffff0000) >> 16; // uppr half anded with ffff, bit shifted by 16

    setNo = (CPULwr & 0xfffc) >> 2; // which element in way0[]

    byteNo = CPULwr & 3; // the lower addr is anded to give

    if (CPUUppr == way0[setNo].upprAddr ) { // Hit

        cout << "Hit" << endl;

        hits++; // hits counter

    }

    else {   // Miss

        cout << "Miss" << endl;

        misses++;

    }

}

cout << "Hits: " << hits << endl; // counts the number of hits

cout << "Misses: " << misses << endl; // number of misses

myAddr.close(); //closing the file

The ifstatement checks whether the element in the array of *way0* object is equal to the contents of the address in the *myAddr*. This causes a hit if true along with incrementing the hit counter, vice versa for a miss. At the end of the code the total number of hits/misses are printed along with the other information, and the file is closed.

## Results

## 

Figure 5 Direct Mapped Hit/Miss

In Figure 5, the left-most column shows the full 32-bit CPU address while the next is the 16-bit upper address, the final column is the set number. The hit counter shows a total of 4 and misses come to 13. Giving a hit rate of 23.5%, a very low hit rate and could not be used in real-world systems. This may have the lowest hit rate of all the cache architectures, but it is the simplest design, requiring the least of components and code.

# 2-Way Associative

## Code

CacheEntry way0[16384], way1[16384]; // 2^14 cache entries, second direct mapped cache

bool up, val up1, val1;

The declarations are identical to that of the Direct Mapped Cache with an exception for second *CacheEntry* used as the second set of the cache and the four Boolean values used later to tidy the ifstatements below.

up = way0[setNo].upprAddr == CPUUppr; // tidy code

val = way0[setNo].validFlag;

up1 = way1[setNo].upprAddr == CPUUppr;

val1 = way1[setNo].validFlag;

if (up && val) { // Hit

cout << "Hit: " << hex << CPUAddr << endl;

hits++;

way0[setNo].LRUFlag = 0; // way0 using most recently used

way1[setNo].LRUFlag = 1; // way1 using Least Recently Used

}

else if (up1 && val1) { // Hit

cout << "Hit: " << hex << CPUAddr << endl;

hits++;

way0[setNo].LRUFlag = 1; // way0 using LRU

way1[setNo].LRUFlag = 0; // way1 using non-LRU

## }

The code above shows the cases for a hit in a 2-way set associative cache. *up* and *val* are used to tidy the code. The first if statement is when the *upprAddr* in the cache entry *way0* matches the *CPUUppr* stored previously. This statement is checking the first set of the cache and so far, is the same as using direct mapped cache mentioned previously. Inside the statement the hits counter is incremented along with the LRUFlag for *way0* set to zero (Most Recently Used) and for way1 LRUFlag is set to one (Least Recently Used). After finding a hit in *way0* the *LRUFlag* must reset so the most recently used cache entry is evicted, the opposite applies for *way1*.

The else if is checking if the next set of the cache has a matching entry, hits is incremented like before, but the opposite occurs with the LRUFlag, way0 is least recently used and way1 is most recently used. The LRUFlag is set for *way0* to evict the least recently used cache entry and for *way1* the most recently used is to be evicted.

if (way0[setNo].LRUFlag == 1) { // enter if way0 is LRU

cout << "Full Addr: 0x" << hex << CPUAddr; // tidies output

cout << " Miss: 0x" << hex << CPUUppr << " way0" << endl;

way0[setNo].upperAddr = CPUUppr; // sets uppr addr in way0[setNo] to CPUUppr

way0[setNo].validFlag = 1; // entry is valid

way0[setNo].LRUFlag = 0; // way0 to most recently used

way1[setNo].LRUFlag = 1; // way1 to least recently used

misses ++; // miss

}

else { // enter when way1 is LRU

cout << "Full Addr: 0x" << hex << CPUAddr;

cout << " Miss: 0x" << hex << CPUUppr << " way1" << endl;

way1[setNo].upperAddr = CPUUppr; // sets uppr addr in way1[setNo] to CPUUppr

way1[setNo].validFlag = 1; // entry is valid

way1[setNo].LRUFlag = 0; // way1 to most recently used

way0[setNo].LRUFlag = 1; // way0 to least recently used

misses ++;

## }

When neither way0 nor way1 has a matching entry (*else*), the miss counter is incremented and a nested if statement checks whether the *LRUFlag* is set. In the valid case, that entry in *way0* is made equal to the *CPUUppr*, so if that address is loaded in again a hit will occur. The *validFlag* is set along with the *LRUFlag* being reset. The *validFlag* is set so that the data in cache is updated and valid. The same occurs with way1 when LRUFlag is not set, the difference being the LRUFlag for way1 is set rather than reset.

## Results

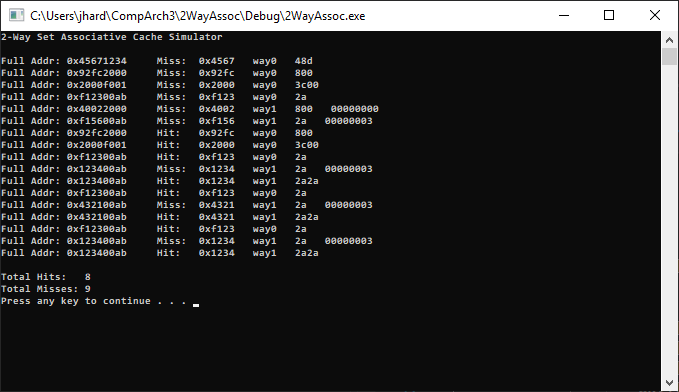


Figure 7 2-Way Set Associative Results

In Figure 6, the columns are the same as before, but the set number of the cache is shown on the far-right column as way0 or way1. The hit counter shows a total of 8 and misses come to 9. Giving a hit rate of 47%, considerably better than direct mapped but slightly behind associative. This cache design seems to be most useful of all the designs with a significantly higher hit rate than direct mapped but with far less complexity and cost than fully associative.

# Fully Associative

## Code

unsigned short CPUUppr, hits = 0, misses = 0, setNo = 0, counter = 0;

unsigned char byteNo;

## bool isFull = 0, isHit = 0, fAddr, val; // is cache full/hit, tidy code

The declarations are identical to that from the direct mapped, but with extra Boolean variables for checking if the cache is full (*isFull*), *isHit* for whether a hit occurred. The short variable setNo is used when the cache isn’t full, counter is used as an index for iterating through all the addresses.

while (setNo <= counter) {

if ((way0[setNo].fAddr == CPUAddr) && (way0[setNo].validFlag)) {

cout << "Hit: " << hex << CPUAddr;

hits++; // increments hit counter

isHit = 1; // indicates a hit

}

setNo++; // used when cache isn't full

}

The while loop above is inside an if statement checking whether the value of setNo has exceeded that of counter, this is used when the cache is being filled, after the check for a hit is made the setNo is incremented until setNo equals counter.

if (!counter) {

way0[counter].fAddr = CPUAddr; // storing address

way0[counter].validFlag = 1; // valid flag to 1

way0[counter].LRUFlag = 0; // Most Recently Used

counter++; // counter=0, must be incremented

}

The code above is inside an if that checks if a hit occurred (*isHit*). The if above checks if the counter is equal to zero, in that case the *CPUAddr* is made equal to that cache entry (at element counter) along with LRU being reset and the counter incremented.

else { // being used now

for (int i = counter; i > 0; i--) { // iterating down old addr

way0[i] = way0[i - 1]; // moving old addr to front

way0[i].LRUFlag = 1; // using LRU when counter>0

}

way0[0].fAddr = CPUAddr; // storing recent addr first

way0[0].validFlag = 1;

way0[0].LRUFlag = 0; // first element is MRU

counter++;

misses++;

if (counter == 16384)

isFull = 1; // cache is full

}

If the counter is more than zero the old addresses must be checked by starting at counter value and iterate down, the loop moves the addresses up one element along with the *LRUFlag* being set. This counts as a miss and the *CPUAddr* is made equal to the first address of the cache along with that entry being set to most recently used. Finally, if the counter equates to the size of the cache (16384) the *isFull* is true.

else { // cache is now full

for (int i = 0; i < 16384; i++){ // all cache entries have relevant addresses

fAdd = (way0[i].fAddr == CPUAddr);

val = (way0[i].validFlag); // clean up code

if (fAdd && val){ // hit when addr match and valid

hits++;

isHit = 1; // using isHit so prog know miss occured

}

else { //miss

for (i = 16384; i > 0; i--) {

way0[i] = way0[i - 1];// shifting all addr by one space

way0[i].LRUFlag = 1;

}

way0[0].fAddr = CPUAddr;

way0[0].validFlag = 1;

way0[0].LRUFlag = 0; // MRU

misses++;

}

}

## }

When the cache is full, iterating through all the cache entries until a hit occurs. In the case of a miss the addresses are shifted by one inside the second for loop. The full CPU address is set equal to the current CPU address, along with the miss counter incremented.

## Results



Figure 6 Fully Associative Results

In Figure 6, the columns are the same as before. The hit counter shows a total of 9 and misses come to 7. Giving a hit rate of 56.3%, better than the other two but could not be used in real-world systems. This may have the highest hit rate of all the cache architectures, but it has the most complex design, costs far more to manufacture and its code is more complex.

# Conclusion

The direct mapped architecture offers a less complex design than that of fully associative cache, the number of comparators required is reduced to one with only one entry, meaning reduced access times and cost. Reduced complexity also means reduced hit rate causing the CPU to access DRAM, incurring huge delay costs.

2-way set associative is the cache architecture used most in industry and it seems obvious from the results. The hit rate is close to that of fully associative but lacks the price and complexity. Unfortunately, the speed at which the cache loads the addresses wasn’t displayed above but if it did it would show 2-way being far faster without the need to search every entry.

The fully associative suffered from the opposite issues, with high hit rates but with high levels of complexity. The design of fully associative requires a comparator for each entry, increasing price and implementation complexity. When loading an address into the cache any entry can be used to store it, when accessing that address again the entire cache must be searched, increasing access times greatly.

All caches used are obviously beneficial to the fetch time of a CPU and all seem to address the issue of the DRAM/CPU speed divergence, with direct mapped having the least expensive implementation, fully associative having the highest hit rate, and 2-way finding a happy medium between the two with marginally lower hit rates than fully associative and slightly more costly design.

# References

[1] D. Das, "What is Locality of Reference in Cache Memory with Diagram", *CSETutor*, 2018. [Online]. Available: https://www.csetutor.com/locality-of-reference-in-cache-memory/. [Accessed: 18- Oct- 2018].

[2] J. Hruska, "How L1 and L2 CPU Caches Work, and Why They're an Essential Part of Modern Chips - ExtremeTech", *ExtremeTech*, 2018. [Online]. Available: https://www.extremetech.com/extreme/188776-how-l1-and-l2-cpu-caches-work-and-why-theyre-an-essential-part-of-modern-chips. [Accessed: 21- Oct- 2018].

[3] "What's difference between CPU Cache and TLB? - GeeksforGeeks", *GeeksforGeeks*, 2018. [Online]. Available: https://www.geeksforgeeks.org/whats-difference-between-cpu-cache-and-tlb/. [Accessed: 23- Oct- 2018].