Chapter 1

Introduction

1.1 Overview

Application programs need to deal with large data sets and therefore require a huge memory to hold data and the computation results. In most of the cases they also require computation to be performed as fast as possible. In recent years, speed of processors has enhanced appreciably faster than access time to main memory. Modern computer architectures employ memory hierarchies comprising one or more levels of cache for improving performance of the system. These caches are small but fast access memories, which store copies of recently accessed code or data blocks. The system designers need to meet the demand for a large storage capacity that can be accessed at faster speeds, but at the same time the overall system costs must be minimized. The cost of memories varies with the technology. There exists an inverse relation between the size of memory with its speed and cost. As the size of memory increases, both its speed and cost decrease. The disadvantage of cache level in the hierarchy is the cost, because it is a limited resource and must be well managed. All the data cannot be fitted into it but whenever a data block is required, the access time of moving the block of data from lower level of hierarchy can influence the process timing. Consequently, the blocks in the cache are handled by replacement policies. This strategy can affect the speed of the program in execution.

Cache replacement policies determine which data block is to be replaced from the cache. When a new data block is added in that place or if the cache is full and a new data block needs to be put into the cache, some other data block must be removed. Some of the popularly used replacement policies are Random Replacement, Least Recently Used (LRU) and Pseudo-LRU replacement policies. To design a system with improved capabilities a suitable replacement policy must be employed which in general is determined in terms of the hit rates achieved by the policy. Researchers and designers are putting a lot of efforts for achieving better hit rates in the various levels of the cache memory for faster execution of processes.

1.2 Motivation

With the advent of technology in VLSI technology the performance gap between processors and memories is widening. Processors are getting faster and bigger but the memories are not able to service the processor requirements. So the importance of the memory hierarchy has increased with advances in performance of processors. Several different solutions have been proposed to deal with the issue of increasing the cache performance.

Figure. 1.1 plots single processor performance projections against the historical performance improvement in time to access main memory. The processor line shows the increase in memory requests per second on average (i.e., the inverse of the latency between memory references), while the memory line shows the increase in DRAM accesses per second (i.e., the inverse of the DRAM access latency)[7]

A memory system consists of a hierarchy of caches. In the memory hierarchy, cache replacement policy decides which block of the page will be replaced in an independent manner, which determines the performance of whole memory hierarchy. In any kind of memory hierarchy, cache replacement policy is a major design parameter to overcome this problem.

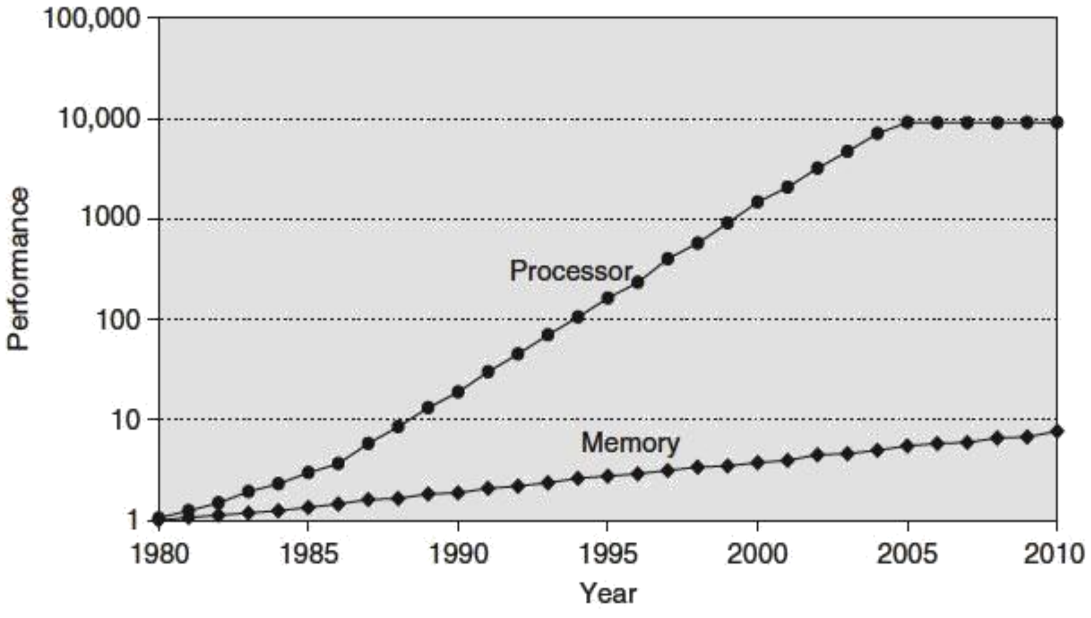


Figure 1.1: Processor vs. Memory Performance [7]

Cache replacement policy is the method used by cache to add a new data block and replace the old data block in cache. The efficiency of the replacement policy affects both the hit rate and the access latency of a cache system, not only in terms of hits and misses but in terms of bandwidth utilization and response time too. A poor replacement policy can increase the miss penalty and cause a lot of trace out of the cache. Because of these reasons a lot of research is done to find the best cache replacement policy.

1.3 Scope of Work

The scope of work for this project is to compare the three replacement policies viz. Random, LRU and Pseudo-LRU for cache memories with respect to Hit Rate by simulation. For the sake of comparison, a 1KB,2KB,4KB, 8KB, 16KB, 32KB and 64KB cache memory is taken as base on which the policies are executed. Selected parameters of cache are varied and the procedure is repeated again. The implementation of the memory controller and the required glue logic is carried out. Verilog language is used for the hardware implementation. Memory controllers for the Main Memory, L1 Cache and L2 Cache are realized using Verilog language.

The Microcontroller input signals like opcode, address and data requirement are captured from a real world application. The 1KB, 2KB, 4KB, 8KB, 16KB, 32KB,64KB cache memory is considered as L2 Cache, in which all the three replacement policies are executed and the resultant Hit Rate is noted down. Test bench is written for feeding the input signals to the memory controllers. The main scope of work can be categorized as:

- Development of memory controllers using Verilog language.
- Development of test bench using Verilog language.
- Simulation and Comparison of Hit Rates.

1.4 Organization of Report

The report is organized as follows

Chapter-1 Overall view of the project work including introduction, motivation, scope of work and objective is described in this chapter.

Chapter-2 This chapter gives the details of literature reviews carried out in connection with the thesis. It also introduces the concept of cache memories and the hierarchy implemented in memories.

Chapter-3 The Design methodology and the implementation of memory controllers are briefed in this section. Verilog modules realized for this work are listed out.

Chapter-4 In this chapter, the simulation results are discussed and the efficiency comparisons for the three replacement policies are tabulated.

Chapter-5 This chapter deals with the conclusion from the interpreted results and explores the future work for this technology.

Chapter 2

Literature Review

2.1 Memory Hierarchy System

Now a day's in most of the computers various memories are organized in the form of memory hierarchy. The architecture of the system consists of slower but larger auxiliary memory in addition to the main memory which can be accessed at relatively higher speed. On the other hand cache memory is small sized but can be accessed very quickly when used for high speed logic processing. The computer architecture is therefore arranged such that faster memories are lying at the top whereas slower ones at the bottom of the memory hierarchy.

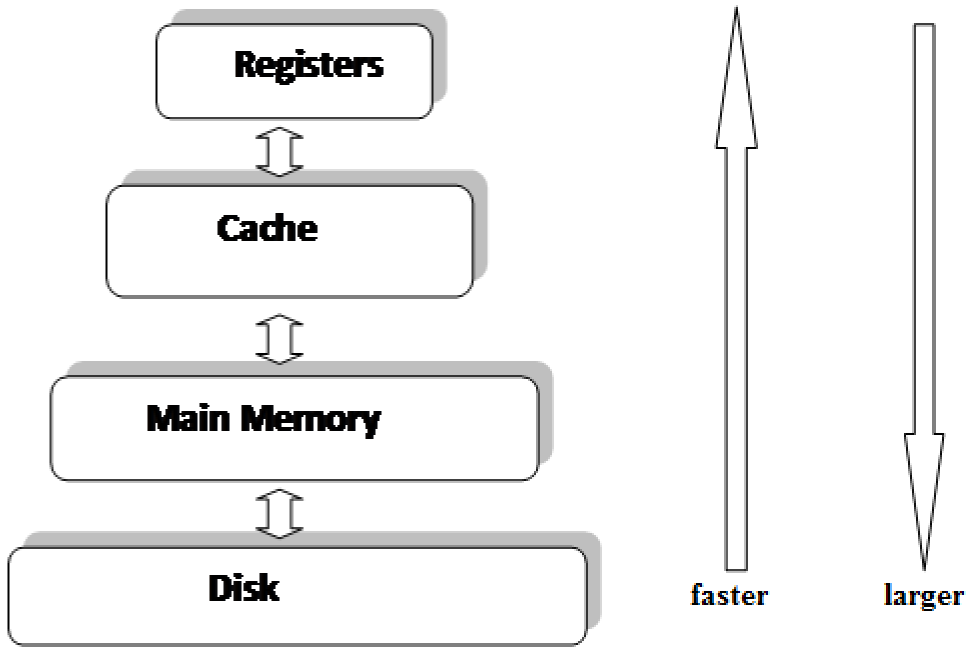


Figure 2.1: Memory Hierarchy

Table 2.1 Comparison of memories

Speed	**Memory**	**Description**
Faster	**Cache**	Cache memory is smaller but faster and is accessible for high speed processing logic. Cache itself is organized into several sub-levels L1, L2 and L3 in order of their increasing speeds.
	RAM	All instructions and data addresses for the processor arrives from RAM. RAM is very fast, but still it takes some time for the CPU to access it i.e., its latency is comparatively higher. RAM is stored in distant, dedicated chips attached to the motherboard, therefore much larger than cache memory.
Slower	**Disk**	Data retrieving from Floppy disk/CD-ROM/DVD ROM and hard disk storage take longer time. Disks are the slowest form of storage.

The memory cache is high-speed memory available inside the CPU in order to speed up access to data and instructions stored in RAM memory. A computer is completely useless if you don’t tell the processor (i.e., the CPU) what to do. This is done through a program, which is a list of instructions telling the CPU what to do.

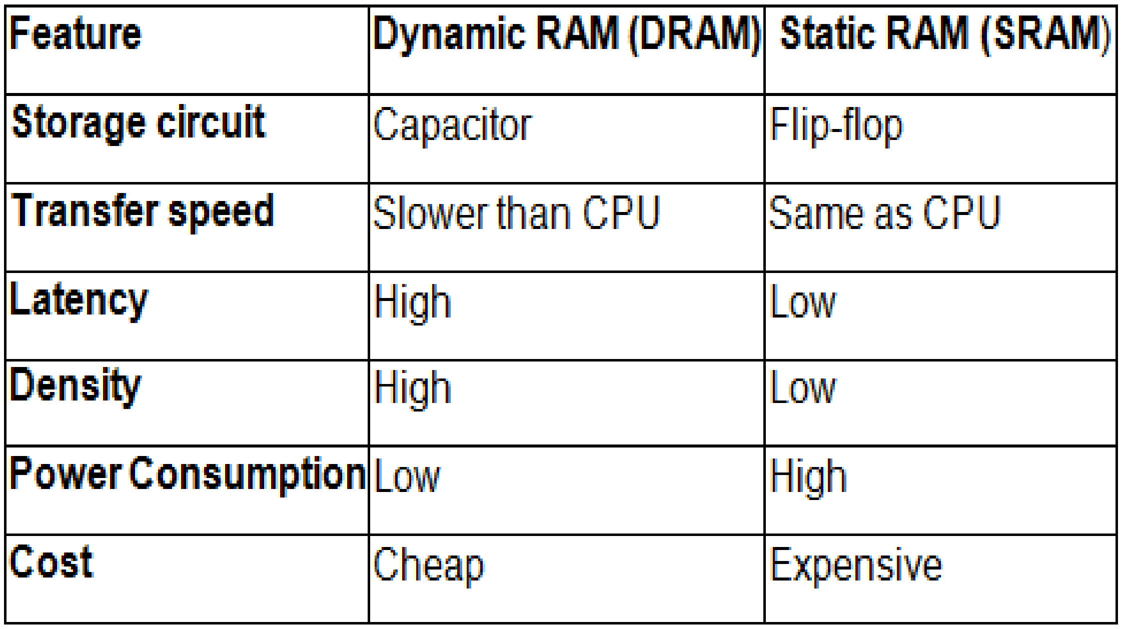
The CPU fetches programs from the RAM. The problem with the RAM is that when its power is cut, its contents are lost this classifies the RAM memory as a “volatile” medium. Thus programs and data must be stored on non-volatile media (i.e., where the contents aren’t lost after your turn your PC off) if you want to have them back after you turn off your PC, like hard disk drives and optical media like CDs and DVD level of Memory Hierarchy

2.2 Dynamic RAM vs. Static RAM

Dynamic (DRAM) and Static (SRAM) are two types of RAMs. PC employs DRAM. In DRAM each bit is stored in a capacitor inside the memory chip. It is possible to manufacture millions of capacitors on a small area, this is referred to as “high density”. Stored charge leaks from capacitors with time hence the capacitors need to be recharged periodically. This recharge process is called refresh. During refresh period data cannot be read or written. DRAM is cheaper than SRAM. Also, DRAM consumes less power than SRAM. The drawback with DRAM is that the DRAM data is not readily available hence its speed of operation is slow relative to CPU.

SRAM can operate at the same speed as CPU. In SRAM each data bit is stored on a flip-flop. Since flip-flops do not require refresh cycles they are able to deliver data with negligible latency. The problem with flip-flops is their large size relative to capacitors, because it requires several transistors to build a flip-flop. This implies that on the area occupied by SRAM with only one flip-flop, we could as well build a DRAM with hundreds of capacitors. Thus static memories have low density. Other associated drawbacks of SRAM are its relatively high cost and more power consumption. The table below presents differences between DRAM and SRAM.

Table 2.2 Comparison between Dynamic and Static RAM



Feature	Dynamic RAM (DRAM)	Static RAM (SRAM)
Storage circuit	Capacitor	Flip-flop
Transfer speed	Slower than CPU	Same as CPU
Latency	High	Low
Density	High	Low
Power Consumption	Low	High
Cost	Cheap	Expensive

Though SRAM is faster as compared to DRAM, its disadvantages don't allow its use as the main RAM circuit. In order to obtain optimum performance from SRAM and DRAM, a small amount of SRAM is employed between CPU and RAM. This is referred to as Cache memory. A cache memory is basically a small amount of SRAM located inside the CPU.

The cache memory controller copies recently used data from RAM to static memory. It now tries to guess what data the CPU may ask next, and loads such data to static memory before CPU asks for it. The CPU accesses cache memory instead of RAM. Since cache memory is smaller in size it takes quite less time to retrieve data from cache memory as compared to time taken to retrieve data from RAM directly. The more the CPU accesses Cache memory, the more is the speed of operation of the system.

2.3 Overview of Cache Memory

Cache memory is one of the very high speed memories employed in the computer system. Cache is quite small but high speed memory usually Static RAM (SRAM). Caching is used for reducing the request latency, time for servicing and therefore network traffic.

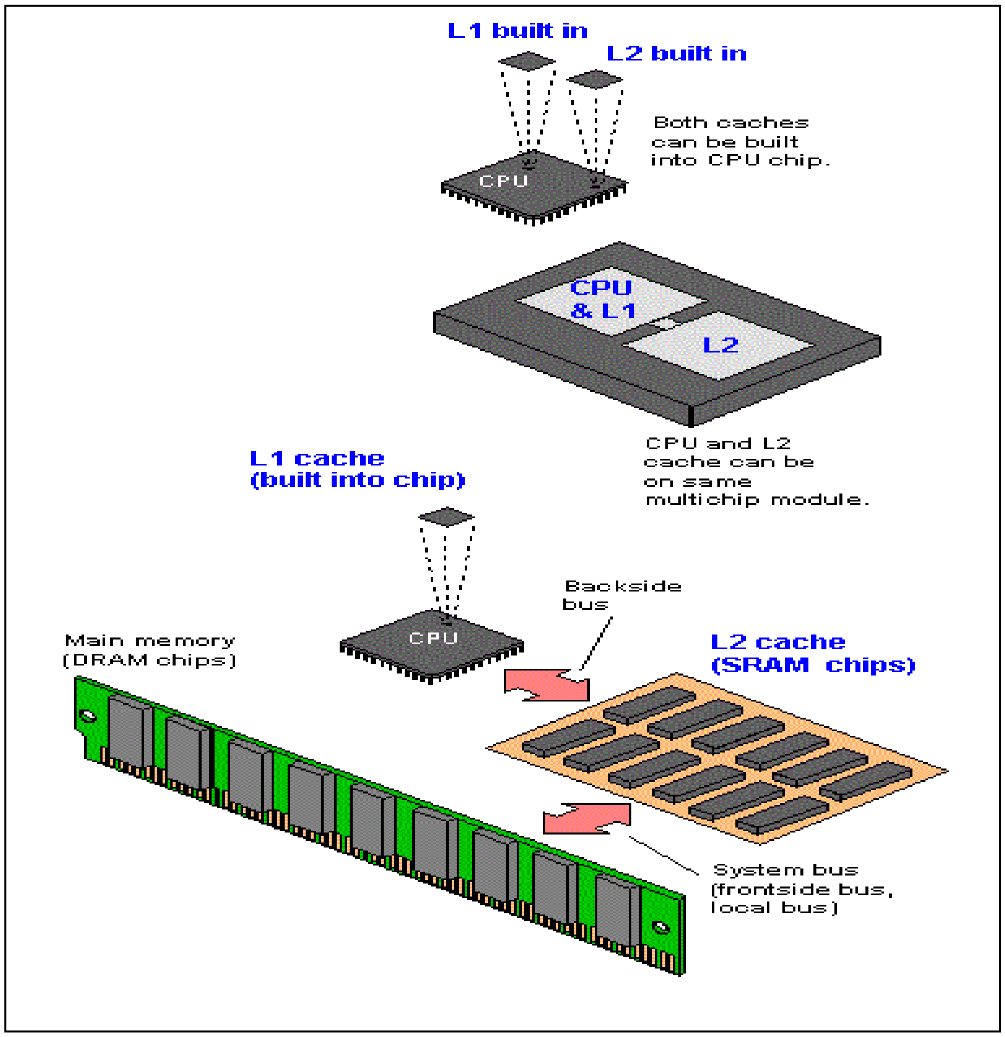

L1 built in
L2 built in
Both caches can be built into CPU chip.
CPU
CPU & L1
L2
CPU and L2 cache can be on same multichip module.
L1 cache (built into chip)
CPU
Backside bus
Main memory (DRAM chips)
L2 cache (SRAM chips)
System bus (frontside bus, local bus)

Figure 2.2: Cache memory[www.pcmag.com/encyclopedia/term/45870/l2-cache]

It is used to store small chunks of data that have been accessed most recently from the main memory and to be accessed again in the near future. To keep track of the blocks of the data which is currently stored and for determining its relation to the rest of the memory, Cache Controllers are used. Cache controller identifiers identify the blocks currently stored in Cache. An identifier comprises a tag, index and offset, valid and dirty bits for a full block of data. To retrieve an individual word inside a particular data block, block offset is used which is an address in the block itself. Cache controller uses these identifiers to process a read or write request issued by CPU. Thus cache controller performs important role in reading or writing data and in fetching whole data blocks from the slower but larger main memory. Data is always stored in blocks, each containing a number of words. Thus, when a cache hit occurs, request can be directly served by the cache, slower main memory does not need to be referenced which results in faster access to data. On the contrary if the block request is not served by the cache, cache miss takes place, therefore block has to be fetched from the main memory which increases request latency. The performance of the system will be improved if more and more blocks are read from the cache. However, the cache size is generally kept small to make the cache cost efficient.

2.4 Cache entry structure:

Tag	Index	Block Offset

Index refers to cache row or cache line from which the data has to be accessed.

Index Length defines the number of bits used for identifying individual row. For r cache rows index length (L) is given by:

$$L = \log_2(r) \text{ bits.}$$

The **block offset** specifies the desired data within the stored data block of a cache row. Typically the effective address is in bytes, so the block offset length is $\log_2(b)$ bits, where b is the number of bytes per data block.

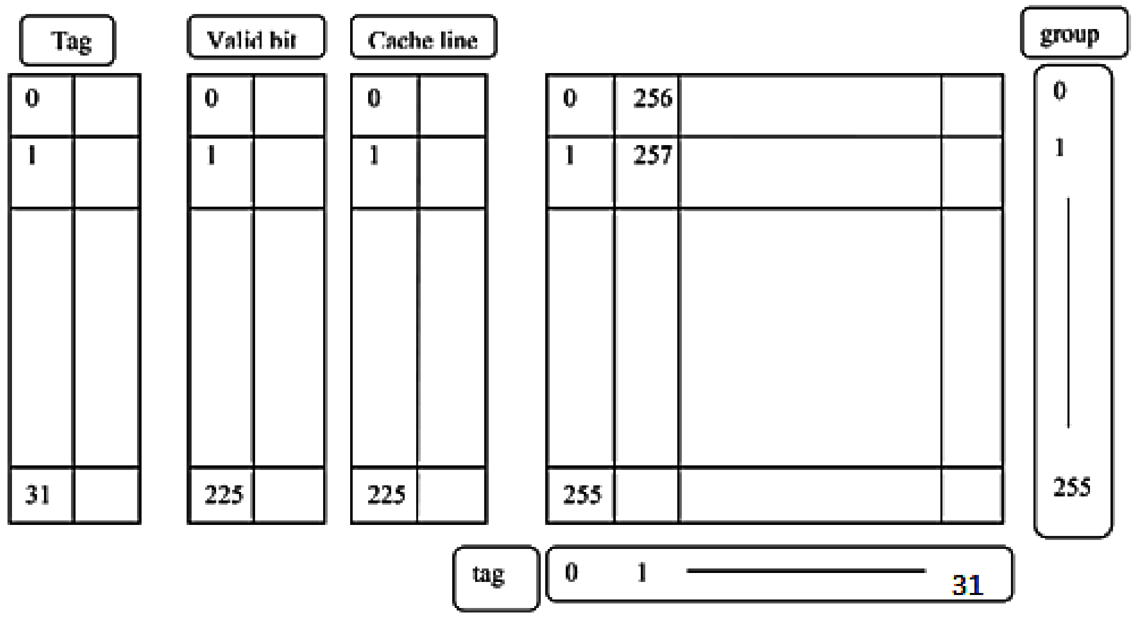
The tag field contains the most significant bits (MSBs) of the address. It is checked against the current row (the row is retrieved by index) to find if it is the required one or the irrelevant memory location having the same index bits as the required one. The tag length is expressed as address_length - index_length - block_offset_length.

Suppose for 2-way Set Associative Cache

Main memory address issued by CPU is divided into 3 fields: Tag, Index and Offset.

Tag (20 bit)	Index (8 bit)	Offset (4 bit)

The memory block can be divided in the form of two dimensional arrays. Since number of bits in index identification field are 8,there will be 256 (2^8)index lines in the cache and for 4offset bits, total 16bytes will be there in one index line. Remaining 20 bits are stored as tag.



2.5 Working of Cache

The working of the cache memory can be described as follows: To access the next instruction which is to be executed, CPU unit looks for it in the instruction cache (L1) first. If it doesn't find it there, CPU looks for the instruction in the L2 cache. If it is not found there also, CPU fetches the instruction from the RAM memory.

The event when CPU loads its instruction or the required data from the cache is called as a "hit" whereas a "miss" refers to event when CPU accesses the data, directly from the system RAM.

It is obvious that when a PC is turned ON, cache is completely empty therefore CPU needs to access data from RAM memory so it is inevitably a miss. Once the first instruction get loaded, then working of the cache comes into picture.

When an instruction from a memory location is loaded into the CPU, memory cache controller circuit loads a small block of the data just below the current position of CPU into cache memory.

Since, programs usually follow a sequential procedure, it is expected that CPU will request a memory location just below the current one. Since cache controller is already loaded with the data lying below the first memory location from which the CPU fetches data, there is probability finding the next data in the cache memory. Consequently, CPU doesn't require data access from outside because it is available in the cache memory embedded inside the CPU and can be accessed at internal clock rate of the CPU.

This small amount of data stored on the cache is called as cache line and is usually 64 bytes long. Besides storing this amount of data, cache memory controller keeps track of the memory location which is to be accessed next by the CPU. A prefetcher circuit can be used for loading more data, next to the first 64 bytes from the main RAM memory into the cache. During the program execution it continuously accesses memory locations for instructions and data in a sequential manner, the next memory location request issued by the CPU will already reside into the cache memory.

Working of the cache memory therefore can be summarized as:

- The CPU issues request for instruction or data stored in an address “x”
- As the required contents from the address “x” aren’t there inside the cache memory, the CPU needs to fetch it from the RAM directly.
- The cache memory controller loads a particular line (64 bytes typically) starting with address “x” into the cache. It is the data more than that requested by the CPU and if the program execution continues sequentially such that the next request is for address x+1, its contents will be already be loaded in the cache.
- Prefetcher is used for loading more data after this line, i.e., continues loading the contents from the address x+64 on the cache. Example: in Pentium 4 CPUs, 256 byte Prefetcher is employed, therefore it loads next 256 bytes after each line that is already loaded into the memory cache.

For sequentially running programs, a fetch directly from the main memory is never required except in the case of very first instruction.
However in actuality programs execution is not sequential always and there are several jumps associated with them. Therefore the main challenge of any cache controller is its ability to guess the address of the next jump of Program Counter (PC) and loading of contents present there into the cache before the arrival of any request from CPU. This will ensure the high speed as CPU accesses from the slower main memory are avoided. This function is often referred to as branch prediction and is employed in most of the modern processors. At least 80% hit rate is ensured in all the modern CPUs which means 80% of the time access from CPU is not required but occurs through the cache.

2.6 Cache Locality

Two types of locality can exist in a cache: Temporal and Spatial.

2.6.1 Temporal Locality

Temporal Locality refers to that locality in which a special block is referenced and is to be reused again in the near future within small time duration.

Suppose Block X is used at time t
Then at time t + i, block X is reused.
We fetch block X at time t, after few cycles, we again access same block at time t+i.

2.6.2 Spatial Locality

Spatial Locality states that if a special block is referenced at a particular time, then the nearby blocks will be referenced soon within relatively close storage locations.

If it is assumed that Block X is used at time t and at time t + i, block X + 1 is used.
If block X is accessed at some later time t then it is very likely that very next block X +1 will be accessed at time t +1, then it is called as spatial locality.

2.7 Levels of Cache

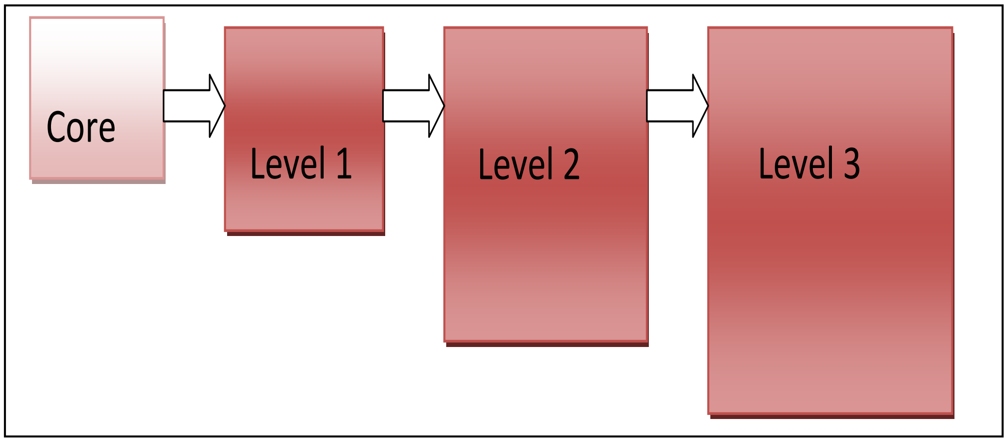


Figure 2.3: Level of cache

CPU cache is rather more complicated than other kind of caches and thus, CPU caches are divided into Levels usually called L1 and L2.

2.7.1 Level 1(L1) Cache

An L1 cache is a memory which is inbuilt in the Processor. If it is present, then the time consuming access from the main memory is avoided by the system. Typical sizes ranges from 8 to 64 KB are now used in modern processors. Level 1 cache is very fast because it runs at the speed of the processor since it is integrated into chip itself.

2.7.2 Level 2(L2) Cache

The L2 cache is another memory which is also found in processor. It is larger but slower in speed than L1 cache. Level 2 caches are used for accessing recent data that is not found in level 1 cache. Typically, its size ranges from 64 KB to 2 MB in modern processors.

2.7.3 Level 3(L3) Cache:

L3 Cache memory is present on the motherboard of the computer. It is an extra cache built outside of the processor to speed up the processing operations. It reduces the time gap between issued request and the data accessed, by operating much more quickly than a main memory. Now a day's L3 caches are used have more than 3 MB of storage in it.

If L1 and L2 caches are used together, then the lost information that is not present in L1 cache can be fetched from the L2 cache. L2 is usually a separate static RAM (SRAM) chip and it is placed between the CPU & DRAM (Main Memory).L1 is configured as separate instruction and data caches. The Level 2 cache is shared between one or more Level 1 caches and is much larger than L1. L1 cache is designed to maximize the hit rate and L2 to minimize the miss penalty.

Table.2.3: Comparison of L1 L2 L3

Cache level	Access	Size	Cost
Level 1	Faster	Smaller	large
Level 2	Moderate	Moderate	Moderate
Level 3	Slower	Lager	small

The difference between L1, L2 and L3 is in their sizes. L1 is smaller than L2 and L3. That's why the data can be easily found in the L1 than in L2 and L3 the access time much faster, if the data is not found in the L1 the data will be search in the L2 bigger cache and again if it is not there, an access to memory will be needed making the access much slower than either to L1 or L2.

2.8 Methods for managing cache

The two methods used for managing cache depending upon the architecture of the processors are: Inclusive and Exclusive

2.8.1 Inclusive

In some processors, all the data stored in the L1 cache is also present in L2 cache this is called "strictly inclusive".

2.8.2 Exclusive

In Several processors, the required data is uniquely available either in the L1 or the L2 but never in both. More amount of data can be stored by using exclusive method because there is no repetition of data on both the caches.

The main advantage of inclusive method is that when a system comprising several processors intends to remove low priority data from the cache, they only need to check L2 cache because the same data will be present in the L1 cache also. On the contrary, in exclusive cache method requires checking of data on both L1 and L2 caches, making the operation slower.

Modern processor units generally employ multiple levels of cache. In the cases where the first and smallest cache referred as L1 cache does not contain the requested data block, processor checks for it in the next level of storage called as L2 cache. Theoretically, this continues until a miss occurs in the last level of cache, subsequently data is retrieved from the main memory. Most of the modern general purpose processors comprise two levels of cache. Although processors consisting three cache levels and even higher are also slowly entering the marketplace.

Another feature of modern caching techniques is that the L1 cache is usually split into two parts. Out of which one is used for data and the other for instructions. Splitting not only improves miss rates but also improves bandwidth and several other performance features. Generally, both the request streams i.e., for data and instruction are combined at the second level cache L2.

The architecture in which this caching scheme consisting separate data and instructions streams is employed is called as Harvard Architecture.

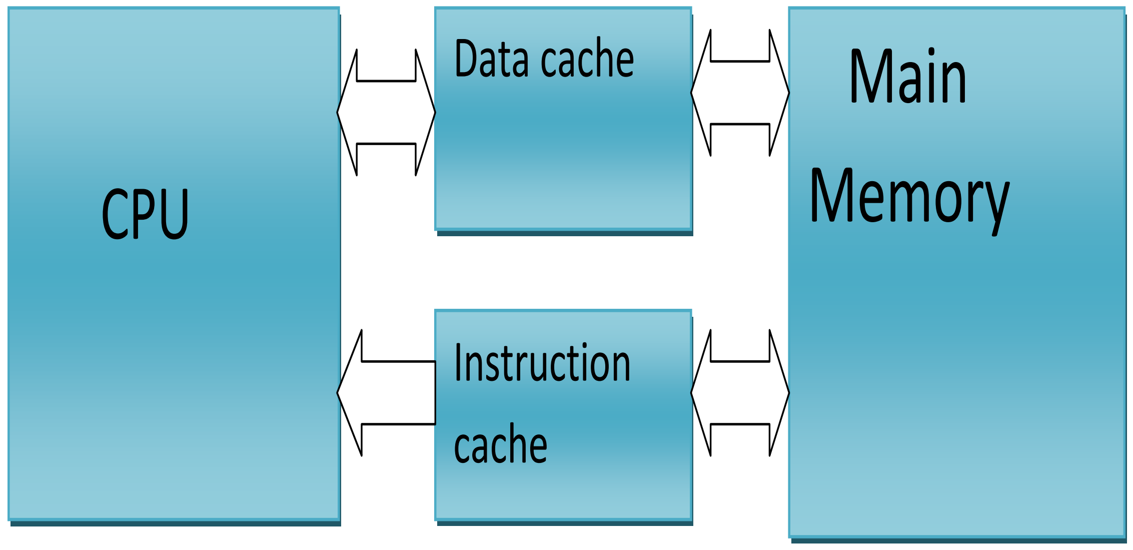


Figure 2.4: Harvard Architecture

2.9 Cache Design Elements

Cache design elements refer to some of the important prospects which we need to consider while designing any cache. In general they include cache terms like cache size, cache hits and misses, mapping functions, replacement policies, block size, and write policy etc. Some of them are briefly discussed in this section.

Basic Cache terms

Some cache terms that we use in general are the following:

2.9.1 Cache block

Cache block is the basic unit for storage in a cache. It may have multiple bytes or words of data.

2.9.2 Cache line

Cache line is same as cache block. It is to be noted that it is different from a “cache row”.

2.9.3 Cache set

Cache set refers to a row in the cache. The number of blocks contained in a set depends on the arrangement inside the cache, it may be direct mapped, set-associative or fully associative.

2.9.4 Tag

Tag for a cache allows it to translate the cache address to a unique CPU address. Since various regions of the memory can be mapped into a block of cache, therefore tag is required for differentiating them.

2.9.5 Cache hit

When CPU requires access to an address and it finds the matching block inside the cache, a hit said to be occurred. So access to RAM is not required.

2.9.6 Cache Miss

CPU tries to access an address, and there is no matching cache block, so the cache is forced to access RAM.

2.9.7 Valid bit

A bit of information that indicates whether the data in a block is used (1) or not used (0).

2.10 Cache Mappings

2.10.1 Direct mapping

In direct mapping cache is divided into small units called lines. Each of these lines is identified by an index bit in the TAG RAM. The main memory is divided into blocks; the size of each block is same as that of Cache memory. The lines in a cache memory correspond to locations within such memory blocks. Each line, drawn from a different block, can be drawn only from the locations corresponding to Cache. The block from which the line is drawn is identified by a tag (thus called TAG RAM).

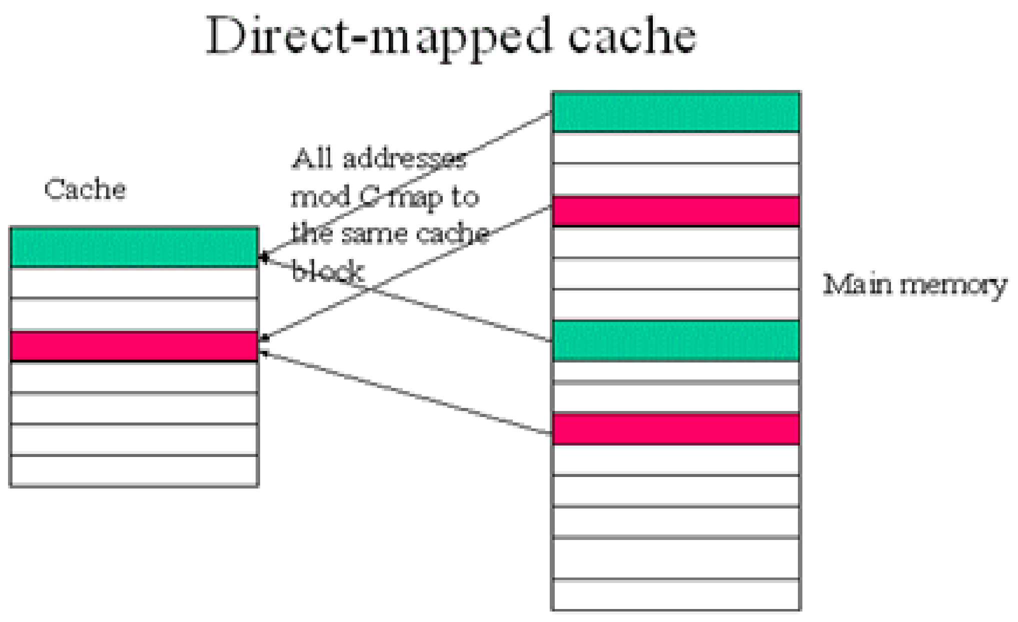


Figure2.5: Direct mapping cache

The cache controller, the routines (both hardware and software) that control the Cache operation can determine if a given byte is stored in the cache by checking the TAG for an index value. The problem here is that if a given program continuously moves between addresses with the same index value in different blocks of memory, this requires frequent cache refreshes which results in Cache misses.

2.10.2 Fully Associative mapping

In this type of mapping any main memory location can be placed in any cache location. It is possible to associate any part of Cache memory with any part of main memory. Lines of bytes from anywhere in the main memory can be put side by side in the cache. The disadvantage of this type of mapping is that the cache controller needs to check the address of every line in the cache in order to find out whether a memory request from the processor is a hit or miss.

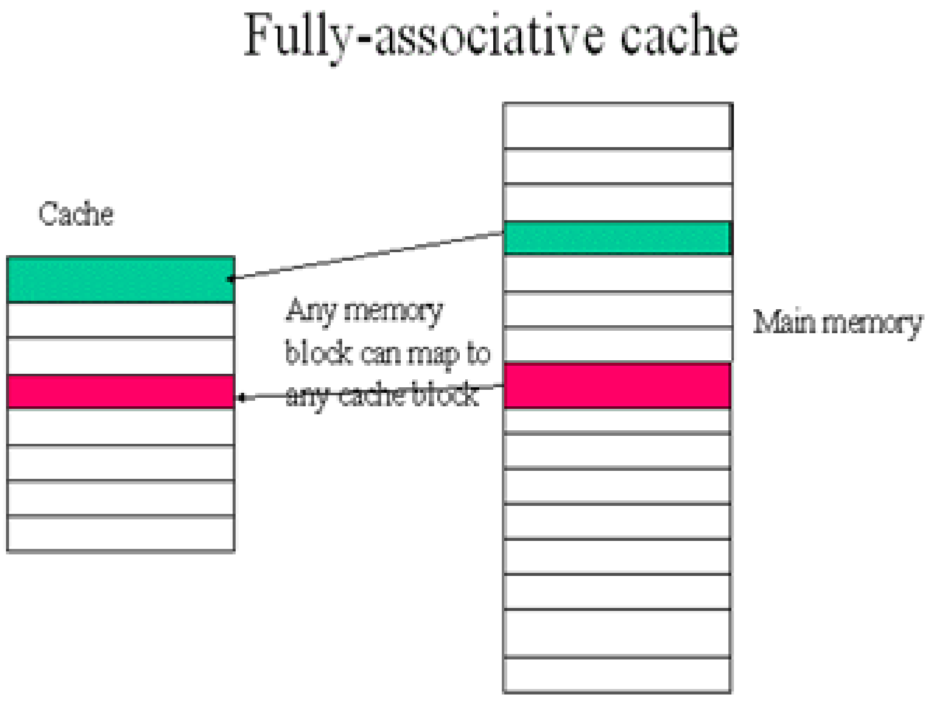


Figure 2.6: Fully Associative cache

Thus in such a cache the TAG must contain the full address of the main memory location stored at each cache location and replacement bits. When any location is accessed, this requires searching through entire TAG in order to find a match. Thus fully associative cache would be efficient only if the TAG is a Contents Addressable Memory (CAM). Since CAM chips are expensive, therefore fully associative cache systems are rarely used.

2.10.3 Set-associative mapping

It provides a compromise between direct mapping and fully associative mapping. Here cache is divided into several smaller direct-map areas. The cache is classified based on the number of ways it is divided into e.g. four-way set-associative Cache. This cache basically contains four smaller direct-mapped caches. This helps overcome the problem of moving between blocks with the same indexes. Thus Set Associative Cache is more efficient as compared to direct mapped cache.

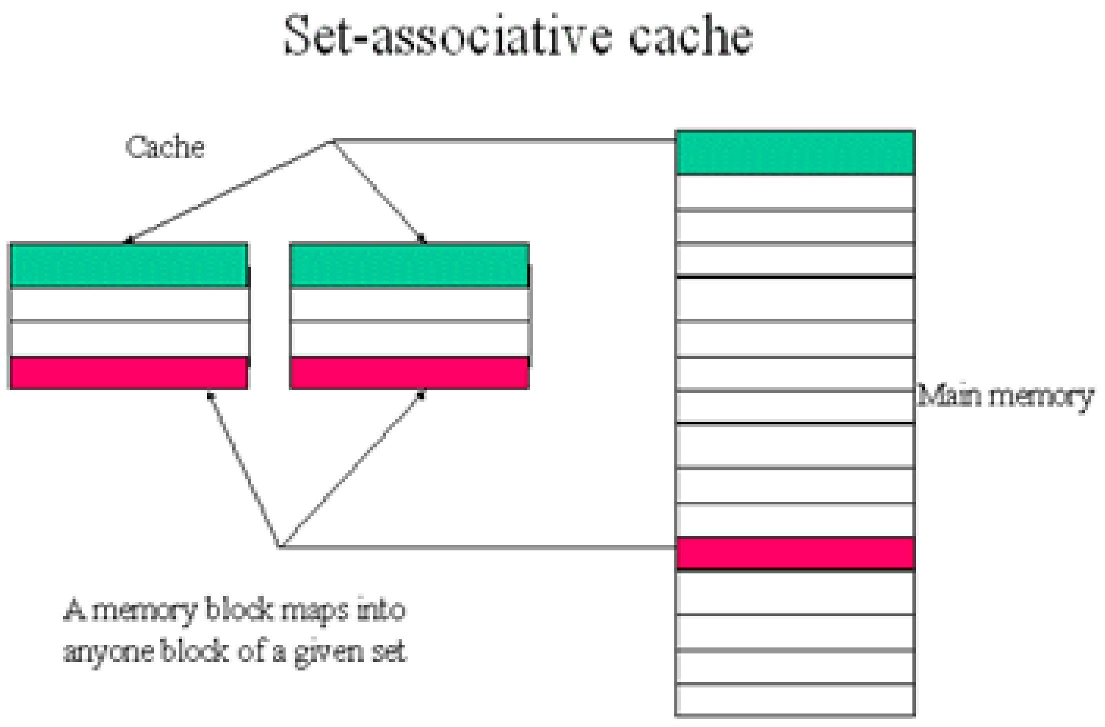


Figure2.7: Set Associative Cache

The disadvantage of Set Associative Cache is that it is very complex hence it is expensive to implement such a cache. Another drawback is that, the more “ways” the Cache is split into, the longer the cache controller must search in order to find out whether the requested information is located in the cache. This results in reduced speed of operation of the cache.Hence the more “ways” the cache is split into, the slower is the performance. A four way split provides a good compromise between performance and complexity.

2.11 Comparison of Cache Mapping Techniques

There is a critical tradeoff in cache performance that has led to the creation of the various cache mapping techniques described in the previous section. In order for the cache to have good performance you want to maximize both of the following:

- **Hit Ratio:**

  You want to increase as much as possible the likelihood of the cache containing the memory addresses that the processor wants. Otherwise, you lose much of the benefit of caching because there will be too many misses.

- **Search Speed:**

  You want to be able to determine as quickly as possible if you have scored a hit in the cache. Otherwise, you lose a small amount of time on every access, hit *or* miss, while you search the cache.

Table 2.4: Comparison of mapping functions

Cache Mapping	Hit ratio	Search performance
Direct	Good	Best
Associative	Best	Moderate
N-way Set associative	Very Good, Better as N Increases	Good, Worse as N Increases

2.12 Cache replacement policies

Cache replacement policies determine which data blocks should be replaced from the cache when a new data block is added in that place or if the cache is full and a new data block needs to be put into the cache, some other data block must be removed. Clearly the block that is to be removed must be from the same set as the new block.

In a direct-mapped cache, there is no need to selection and only one block to choose from the cache. However, in a set-associative or fully-associative cache, some determination must be made. This process is used is called Cache Replacement Policies.

- Random
- Least Recently Used (LRU)
- Pseudo-LRU

2.12.1 Random Replacement policy

It is the simplest among all the policies. In this policy when a cache data block is to be replaced, it replaces a block from any set of the cache randomly. In this policy data about the access history is not preserved. Though this is a simple to implement policy but its overall performance is worse because it does not utilize the locality principle.

2.12.2 Least Recently Used (LRU)

This algorithm determines the least recently used line which might be required in future and replaces that block. The LRU places the least recently used block at the top of the cache, whenever a new block is accessed. When Cache limit is reached, then the line that is least recently accessed is replaced beginning from the bottom of the cache.

For a given program, the allocated memory page will be deemed suitable for replacement if the said page has not been accessed for a certain period of time, provided:

1) The program does not need to access the page;

2) The program is conducting page faults (a sleeping process). In such a case the program is not able to access the page although it might have done so without the page faults. However, LRU algorithm implementations treat the two types of LRU pages equally and do not discriminate between them.

2.12.3 Pseudo-LRU

Pseudo-LRU (another cache algorithm) is developed to improve the Least Recently Used algorithm. PLRU is divided into two categories:

- Tree-PLRU
- Bit-PLRU

The basic idea with this algorithm is to approximate the true LRU scheme using a binary tree structure. This structure requires n-l memory cells in an n-way set associative cache. These cells are required to preserve the order of the recently referenced blocks.

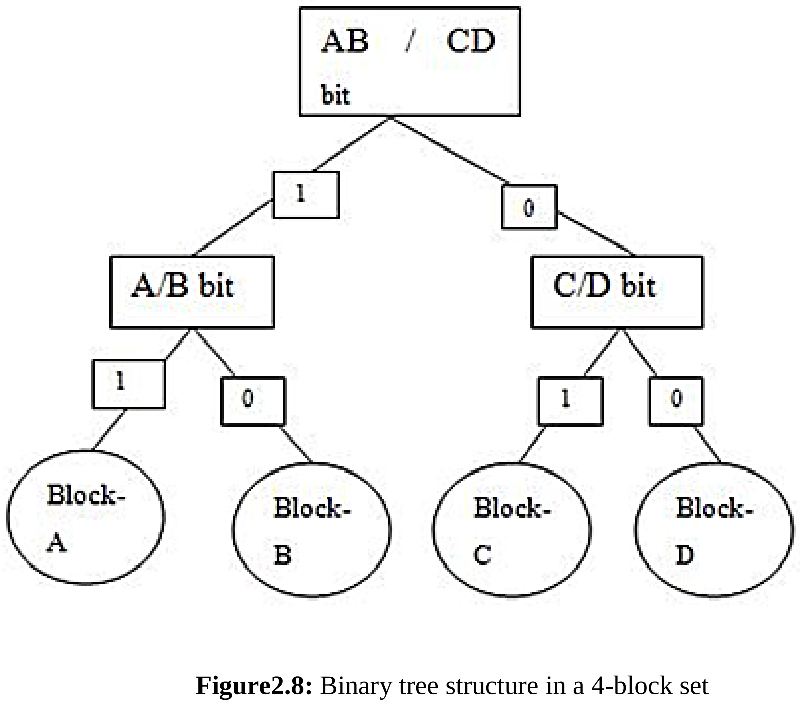


Figure2.8: Binary tree structure in a 4-block set

The tree shown below represents a 4-way set associative cache. Let there be four blocks (namely A, B, C and D) within each cache set. The blocks are represented by the leaves. There are three bits to store the history of the blocks: AB/CD, A/B and C/D bits. When a memory is referenced, the value of each history bit is updated.

When block A or block B is accessed, the AB/CD bit is set to 1, and whenever block C or block D are referenced, the AB/CD bit is set to 0. Similarly, A/B bit determined whether block A or block B is accessed on a memory access. If block A is referenced then A/B bit is set to 1 otherwise this bit is set to 0 if block B is referenced. Same is the case with C/D bit when block C (or block D) is referenced. [5]

2.13 Write policy

2.13.1 Write-Through Cache

Write-thorough cache directs write I/O onto a given cache and through to underlying permanent storage. After this it confirms I/O completion to the host. In this way it is ensured that data updates are safely stored, for example a shared storage array. This type of cache has a disadvantage that the I/O still experiences latency. This type of cache is good for applications that write and then re-read data frequently because such a cache results in low read latency.

Write-through cache directs write I/O onto cache and through to underlying permanent storage before confirming I/O completion to the host. This ensures data updates are safely stored on, for example, a shared storage array, but has the disadvantage that I/O still experiences latency based on writing to that storage. Write-through cache is good for applications that write and then re-read data frequently as data is stored in cache and results in low read latency.

2.13.2 Write-Around Cache

It is similar to write through cache but here write I/O is written directly to permanent storage and not on the cache. This prevents flooding of cache with write I/O that is not likely to be re-read subsequently. This cache has a disadvantage that a read request for recently written data results in a “Cache miss”. Also for each read request, the reading is to be done from slower bulk storage and hence it has higher latency.

2.13.3 Write-Back Cache

In this cache write I/O is directed to the cache and completion of the same is immediately confirmed to the host. This cache has low latency and high throughput for write-intensive applications. There exists data exposure risk with this type of cache, this is because the only copy of written data is in cache. It is for the user to decide whether write-back cache offers sufficient protection, because in such a cache data is exposed until it is staged to external storage. This type of cache finds application for mixed workloads because here both read and write I/O have similar response times.

2.14 Block size

In developing new caching algorithms the block size is an important factor. Whenever a block is fetched from the server and placed into the cache, then neighboring blocks in close proximity are also cached. This approach may provide higher hit ratio because of spatial locality. However, the cache hit ratio declines as the block size increases. This is because larger blocks require more space in the cache, so the cache can accommodate less blocks resulting in subsequent reduction in cache hit rate over time. In this project we have assumed that the cache block size is the sum of tag size and file block size. Moreover, the cache block can host only one file block.

2.15 Cache Miss

A failure to read or write data in the cache is referred to as cache miss. This results in much longer latency in a main memory access. Cache misses can be classified into following three categories:

- instruction read miss
- data read miss
- data write miss

In above categories instruction read miss causes largest delay, because the processor has to wait till the instruction is fetched from the main memory. Data read miss causes a smaller delay, because instructions independent of the cache read can be issued, in this way execution is continued till the data is returned from the main memory and execution of dependent instructions is resumed. Data write misses generally cause the shortest delays, because writes can be queued and there are few limitations on the execution of subsequent instructions.

Classifying Misses: 3 Cs

Compulsory— When the block required is not present in the cache then this block must be brought into the cache first. This is called compulsory miss. This is also referred to as cold start miss or first reference miss.

Capacity— This type of miss occurs when all the blocks required during execution of a program cannot be contained in the cache. This type of miss results due to blocks being discarded and later retrieved.

Conflict— If set associative or direct mapped type block-replacement strategy is used then conflict misses occur. This occurs because a block can be discarded and later retrieved if too many blocks map to its set. These misses are called collision misses or interference misses.

2.16 Cache Performance

The performance of a cache can be quantified in terms of the hit and miss rates, the cost of a hit, and the miss penalty, where a cache hit is a memory access that finds data in the cache and a cache miss is one that does not.

When reading, the cost of a cache hit is roughly the time to access an entry in the cache. The miss penalty is the additional cost of replacing a cache line with one containing the desired data.

Access time = (hit cost) + (miss rate)*(miss penalty)

=(Fast memory access time) + (miss rate)*(slow memory access time)

Chapter-3

Design Methodology

3.1 Objective

The goal of this thesis is the implementation of L1 and L2 cache for a single core processor and comparison of their replacement policies. In this thesis comparison among the three replacement policies i.e. random, Least Recently Used (LRU) and Pseudo-LRU is performed by using set associative mapping technique. Here 2-way, 4-way and 8-way set associative mapping are used and the hit ratios of the various replacement policies have been determined.

3.2 Details of implementation

The implementation of the memory controller and the required glue logic is carried out. Verilog language is used for the hardware implementation. Memory controllers for the main memory, L1 Cache and L2 Cache are realized using Verilog language. The functional block diagram for the memory controller is shown below in which L1, L2 and main memory are controlled by various signals. Instead of receiving the input from microcontroller, a trace file is used for providing addresses.

3.3 Model description

This model provides the operational function of the L1 cache to simulate the L2 cache controller for comparing replacement policies. The memory reference is being read from the trace file followed by its decoding in order to make a read or write request to cache L2. The function of each block with their signals are as follows.

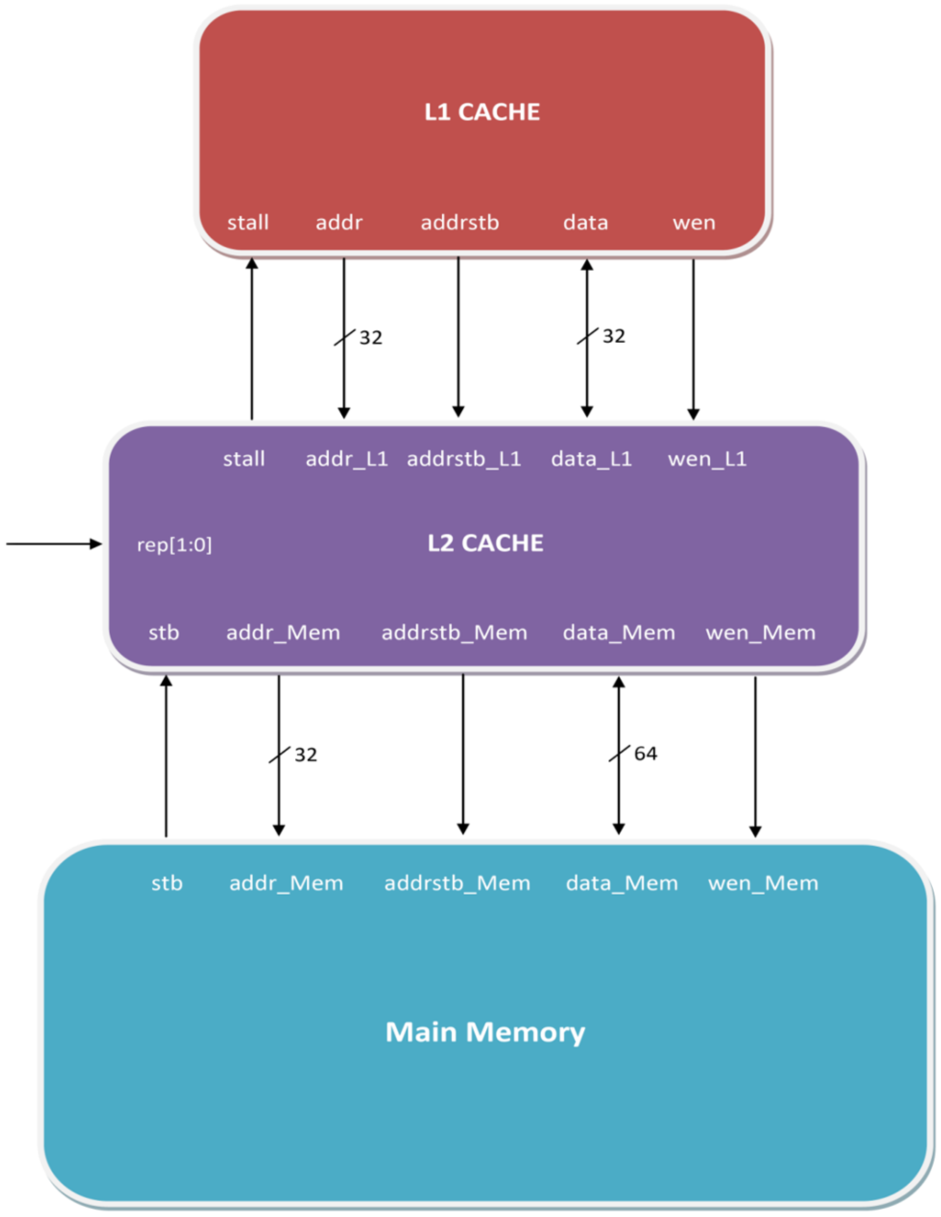


Figure 3.1: Block diagram of Memory Hierarchy Subsystem

3.3.1 L1 Cache

The various ports of the L1 cache are:

Input Port(s):

stall – It is an active high signal used to indicate whether the read/write cycle is over in the memory device that this is requesting data to. A fresh cycle can only start when stall is not in asserted state.

- (stall = 0) => L2 is busy processing request
- (stall = 1) => L2 can take new request

Output Port(s):

we – Write enable signal is an active low signal. Write enable signal is used to indicate whether the memory reference is a read or write. It should be asserted/de-asserted prior to providing address of the desired location.

- (we = 0) => request is a write
- (we = 1) => request is a read

addr[ADDRESS_WIDTH-1:0] – Address bus. Parameter ADDRESS_WIDTH determines the width of the address bus.

addrstb–It is toggled to provide an intimation to the memory device that the new address inputs are valid.

Inout Port(s):

data[DATA_WIDTH-1:0] –Data bus is a bi-directional bus to transfer the data to or from the L2. The width of the bus is determined by the Parameter DATA_WIDTH.

Mechanism for interfacing with L2Cache

This model is used for providing the memory address of 32 bits and therefore has an address bus addr[31:0].It also asserts or de-asserts the write enable (we) signal, which is an active low signal in accordance with type of the memory reference provided. Whenever there is an availability of a valid address or data (for memory write operation) on the bus, address strobe signal is toggled by this module. This toggling in turn triggers the L2 cache thereby latching the address, data and WE signal into it. For the time L2 cache is processing a request came from L1 cache, L2 cache asserts the stall signal. Since stall is an active high signal, when it is in high state(i.e. stall=1) L1 cannot send another reference request and therefore L1 cache is stalled as long as L2 de-asserts stall signal for further processing.

3.3.2 L2Cache

The purpose of using L2 cache is for processing the memory reference request that arrived from L1 cache and managing the references in the L2 cache. To perform these functions it uses one of the various replacement policies and also keeps on tracking the test statistics. It therefore acts as an interface to the Main memory.

Input Port(s)

1. Stb–It is Data strobe signal which is to be sent along with data bits.
2. we_L1– It is the write enable for L1 cache which is active low signal. Write enable indicates that the memory reference provided is for read or for write.
3. addrstb_L1–Address strobe of L1 signals the memory device that a valid address is available at its input. When it is in asserted state address on the bus is valid.
4. addr_L1[ADDRESS_WIDTH-1:0] – Address bus. The bus width is determined by the Parameter ADDRESS_WIDTH.

Output port(s)

1. stall
2. we_MEM
3. addr_MEM
4. addrstb_MEM

These output signals are similar to that of the L1 outputs which are directed towards the main memory.

Inout Port(s)

1. data_L1[31:0] –It is the 32 bit bidirectional data bus to/from L1 cache
2. data_MEM [63:0]– The bidirectional bus width towards the main memory is of 64 bits because of its larger size.

3.3.3 Main Memory Controller

This controller acts as the main memory controller in the L2 cache controller to simulate the replacement policy. How fast a reference request is fulfilled depends on the replacement policy used by the controller. It employs following important ports for its functioning.

Input Port(s)

1. we – Write enable signal is an active low signal which depicts whether the incoming memory request is read or write. It should be asserted or de-asserted before providing the address.

2. addr[ADDRESS_WIDTH-1:0] – Address bus. The bus width is determined by the Parameter ADDRESS_WIDTH.

Inout Port(s)

1. data[DATA_WIDTH-1:0] – Bidirectional data bus. The bus width is determined by the Parameter DATA_WIDTH.

2. stb – Data strobe signal. It is to be sent with data bits to indicate the valid data on the bus.

Mechanism for interfacing with L2 Cache

This model is employed for latching the address as well as WE signals from the address bus addr[31:0] and Write Enable (we) when the address strode addrstb is toggled. As long as the stb signal toggles L2 requests are stalled.stb signal is toggled by memory when it bursts out a chunk of data on the data bus[63:0] or it can be said that stb is toggled when a process is completed.

3.4 Cache Controller

In this project a behavioral approach for designing the logic circuit is followed. The logic responds properly to read and write requests made by the CPU. The possible actions available to the cache controller are reading or writing to cache memory, fetching the whole data blocks from main memory, or writing out blocks to main memory. The controller decides which of these actions to take based on the signals it stores internally, which describe the status of each block in the cache memory. Purposely, the controller have to compared the tag portion of the address with the tags stored for each of the blocks, as well as check the status of the dirty and valid bits for the block being targeted.

Design Flowchart for Read and Write operation

This section describes the operational flow of signals during read and write cycles of CPU. Behavioral Cases for Read and Write operation.

For Write operation

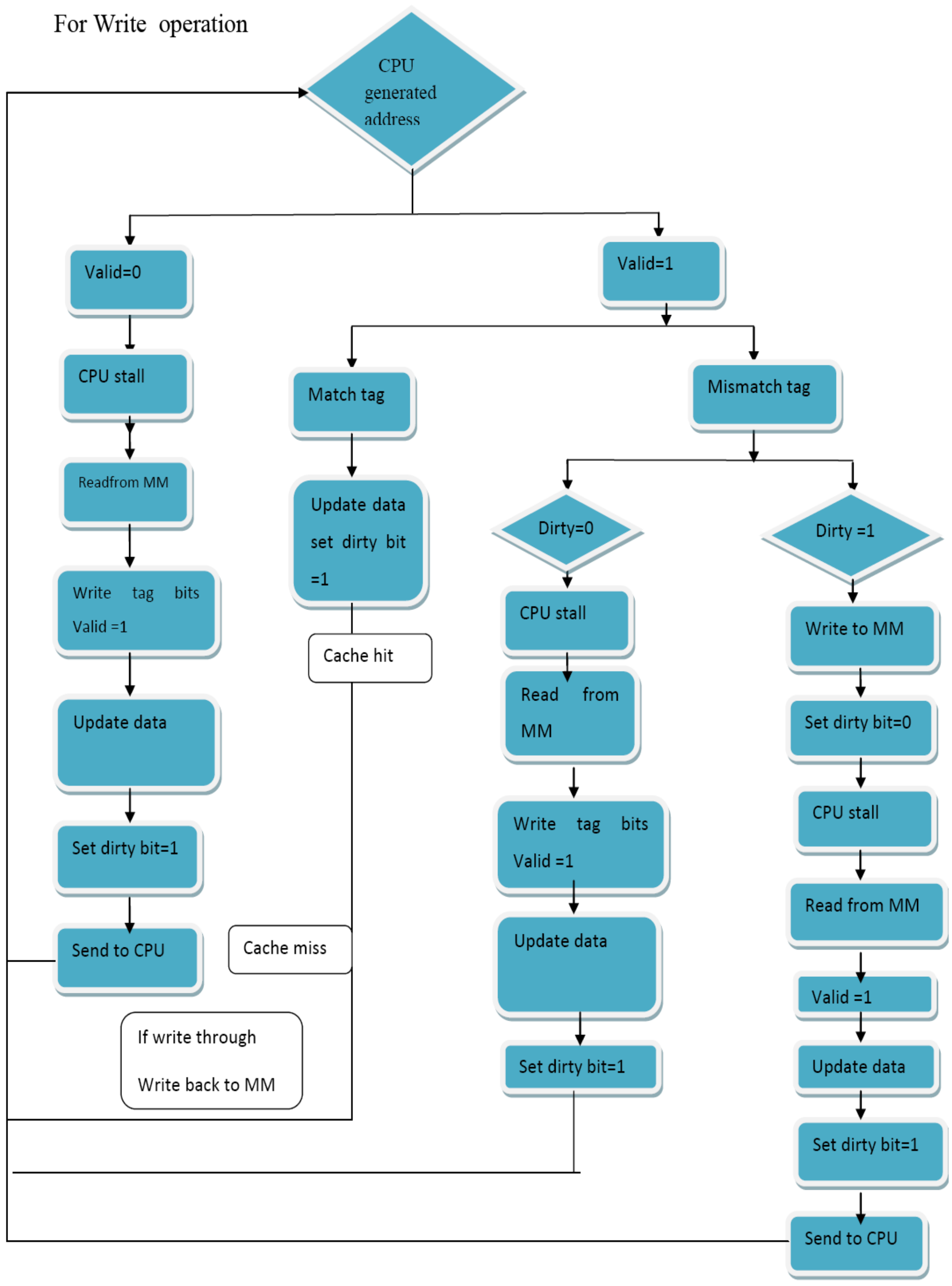


Figure 3.2 Flow diagram for write operation

3.4.1 Write Operation

Whenever CPU requires to write contents to a memory location, cache receives a write request from the CPU. CPU generates an address and write data for write operation. The address is searched in Cache for availability. If Valid bit =0, then data is not found in the cache which means a cache miss. The cache then requests the data from next memory in hierarchy. The CPU will wait till the data is available in the cache for the given address. After fetching data from new address, valid bit associated with the targeted block is set to 1. The data is then updated and dirty bit is set to 1(if write through policy is implemented then the data is immediately written back to main memory).

If there is an index match of the required address in the Cache, then it checks for tag. There are two possibilities, it is either a tag match or tag mismatch. If it is a match, then the data is updated and dirty bit is set to 1. There is a Cache hit in this case.

Dirty bit=0 (cache data= memory data)

Dirty bit=1(cache data $\neq$ memory data)

However, if there is a tag mismatch, this requires performing a block replacement in the Cache. First the **dirty** bit for the corresponding block must be checked.If the dirty bit is clear, the block replacement can be carried out. Requested addresses is sent to main memory and the corresponding data from main memory is stored in cache. The cache is updated with new data and the dirty and valid bits associated with the targeted block are set to 1 (if write through policy is implemented then the data is immediately written back to main memory).

If dirty bit=1, then the new data present in cache is first written back to main memory and dirty bit is cleared. The CPU will wait till the data is available in the cache for the given address. After fetching data from new address, valid bit associated with the targeted block is set to 1. The data is then updated and dirty bit is set to 1(if write through policy is implemented then the data is immediately written back to main memory).

For Read operation

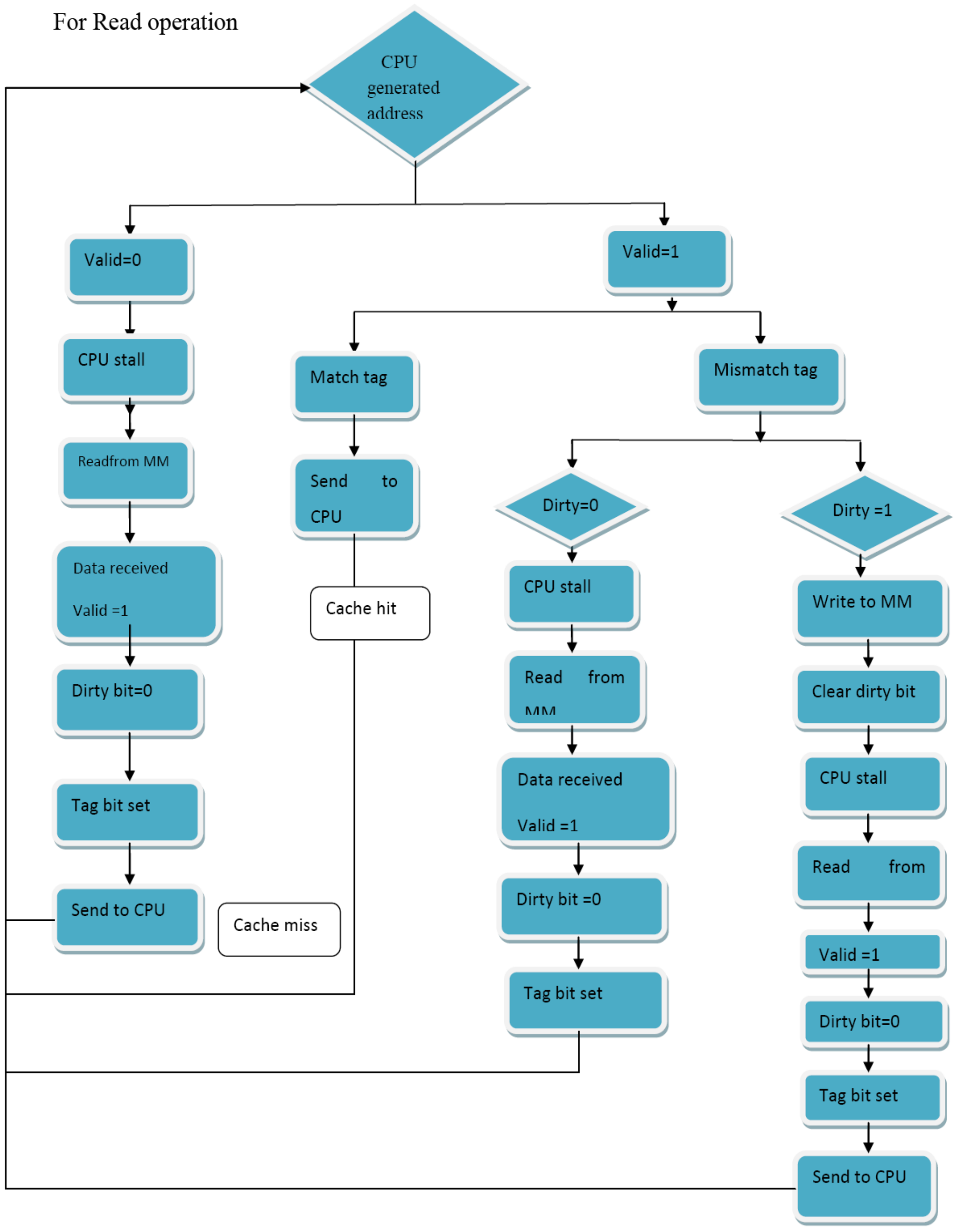


Figure 3.3 Flow diagram for Read operation

3.4.1 Read operation

Whenever CPU requests contents of a memory location, it generates an address and sends a read request to cache. The address is searched in cache for availability. If Valid bit =0, then data is not found in the cache which means a cache miss. The cache then requests the data from next memory in hierarchy. The CPU will wait till the data is available in the cache for the given address. After fetching data from new address, valid bit associated with the targeted block is set to 1. The required data is then send back to CPU.

If there is an index match of the required address in the Cache, then it checks for tag. There are two possibilities, it is either a tag match or tag mismatch. If it is a match, then the data is sent to CPU. There is a Cache hit in this case.

Dirty bit=0 (cache data= memory data)

Dirty bit=1(cache data $\neq$ memory data)

However, if there is a tag mismatch, this requires performing a block replacement in the Cache. First the **dirty** bit for the corresponding block must be checked.If the dirty bit is clear, the block replacement can be carried out. Requested addresses is sent to main memory and the corresponding data from main memory is stored in cache. The required data is sent back to CPU.

If dirty bit=1, then the new data present in cache is first written back to main memory and dirty bit is cleared. The CPU will wait till the data is available in the cache for the given address. After fetching data from new address, valid bit associated with the targeted block is set to 1. The required data is then sent back to CPU.

Chapter-4

Results and Discussion

4.1 Simulation Result

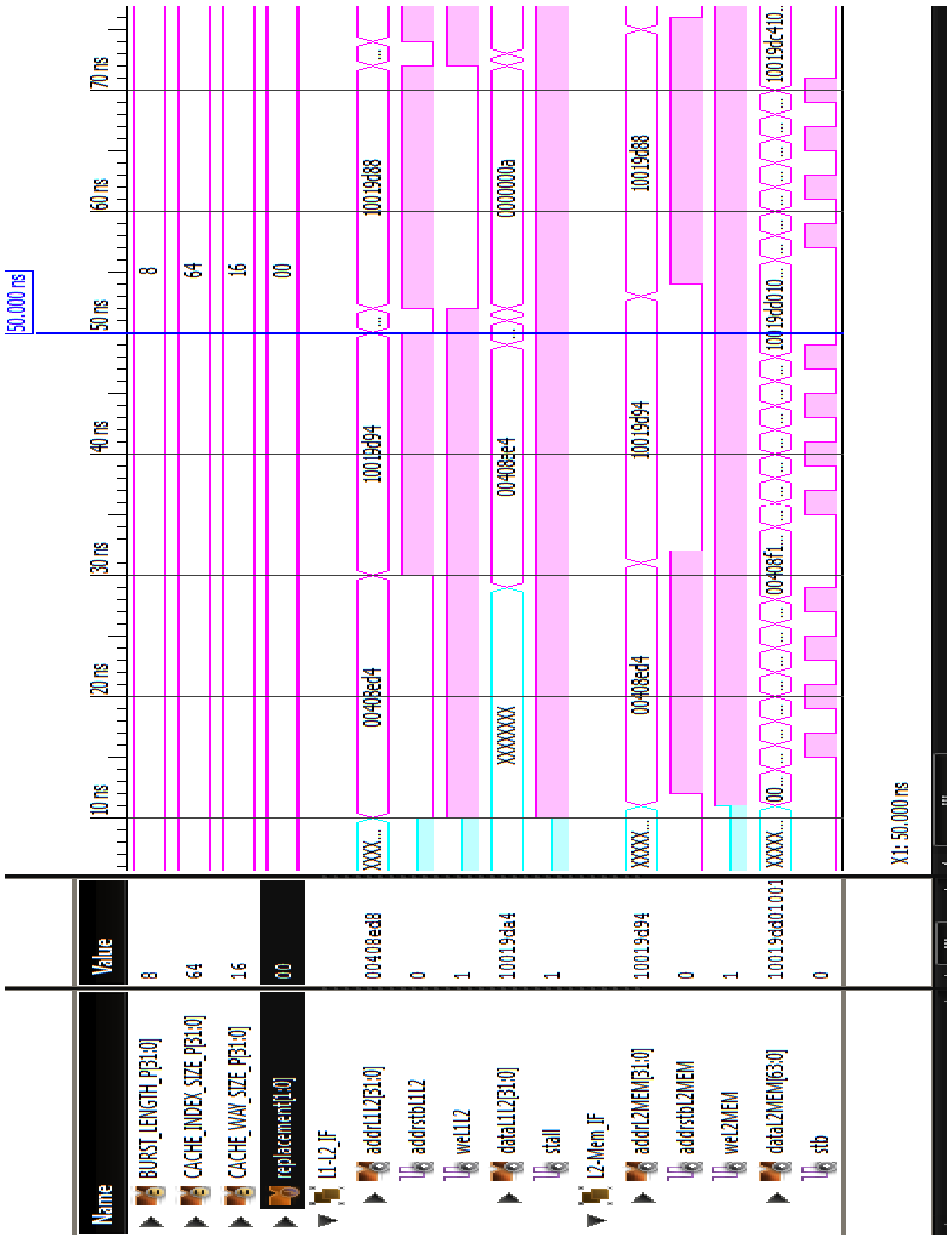


Figure. 4.1 Interface Signals of the memory modules in simulation

The Fig. 4.1 shows the simulated interface signals for the Cache controllers. The above snippet from simulation was done with following parameters:

- BURST_LENGTH = 8
- CACHE_WAY_SIZE = 16
- CACHE_INDEX_SIZE = 64
- REPLACEMENT_POLICY = 0
    - 0 = RandomReplacement Policy
    - 1 = LRU Replacement Policy
    - 2 = Pseudo-LRU Replacement Policy

Here we can see two groups of signals

- **L1-L2_IF**: The interface signals between L1 Cache and L2 Cache.
    - **addrL1L2**: Address bus from L1 to L2 Cache.
    - **addrstbL1L2**: this signal toggles whenever a new address is put on the addrL1L2 bus by the L1 Cache.
    - **weL1L2**: this signal tells the L2 Cache, weather the request is read or write.
    - **dataL1L2**: Rd/Wr data is put on this bi-directional bus.
    - **stall**: this signal is input to L1 Cache to stop it from giving new requests to L2 Cache when it is busy.

- **L2-Mem_IF**: The interface signals between L2 Cache and Main Memory.
    - **addrL2MEM**: Address bus from L2Cache to Main Memory.
    - **addrstbL2MEM**: this signal toggles whenever a new address is put on the addrL2MEM bus by the L2 Cache.
    - **weL2MEM**:this signal tells the Main Memory, weather the request is a read request or a write request to Main Memory.
    - **dataL2MEM**:Rd/Wr data is put on this bi-directional bus.

- **stb**: this signal toggles whenever a double word is sent from Main Memory to L2 Cache in response to a read request made to Main Memory from L2 Cache.

In the diagram, after the system comes out of the reset, we see a write command is issued as the weL1 L2 is high,and the required address and data are put on the respective buses. The address location is not present in the memory so the CacheMiss occures and the L2 cache sends a read request to Main Memory. The stall signal goes high till the write request is completed. The L2 Cache fetches data from Main Memory. The Memory read can be seen on the L2-MEM_IF signals. The Main Memory reads back 8 double words denoted by toggeling of stb signal 8 times.

Next, again a write command is issued, as the address is not present in the L2 Cache, the above procedure is again followed. The above two misses are called the compulsory miss or the cold start miss.

Then a read request is made to the next address. As the data is already present in the L2 Cache, there is a Cache Hit and no data is read from the Main Memory. The required data is sent back to the requester, in this case the L1 Cache.

The simulation was performed for 2500 read/write/instruction fetch requests like this in the project with varying parameter values of CACHE_WAY_SIZE, CACHE_INDEX_SIZE and REPLACEMENT_POLICY.

4.2 Comparison Results

The graph of Figure 4.2. The corresponding table 4.1 shows the comparative analyses of the observed data .From the obtained results one can easily distinguish different entities of efficiency existing among the three replacement policies i.e., Random, LRU and PLRU on the cache configuration.

The below graphs is obtained when 2-way, 4-way and 8-way set associative mapping has been used for implementation. For comparison replacement policies, cache sizes as well as index sizes are successively increased. For comparison 1KB, 2KB, 4KB, 8KB and 16KB, 32KB, 64KB cache size is employed and index sizes are varied in steps of 8, 16, 32, 64 and 128.

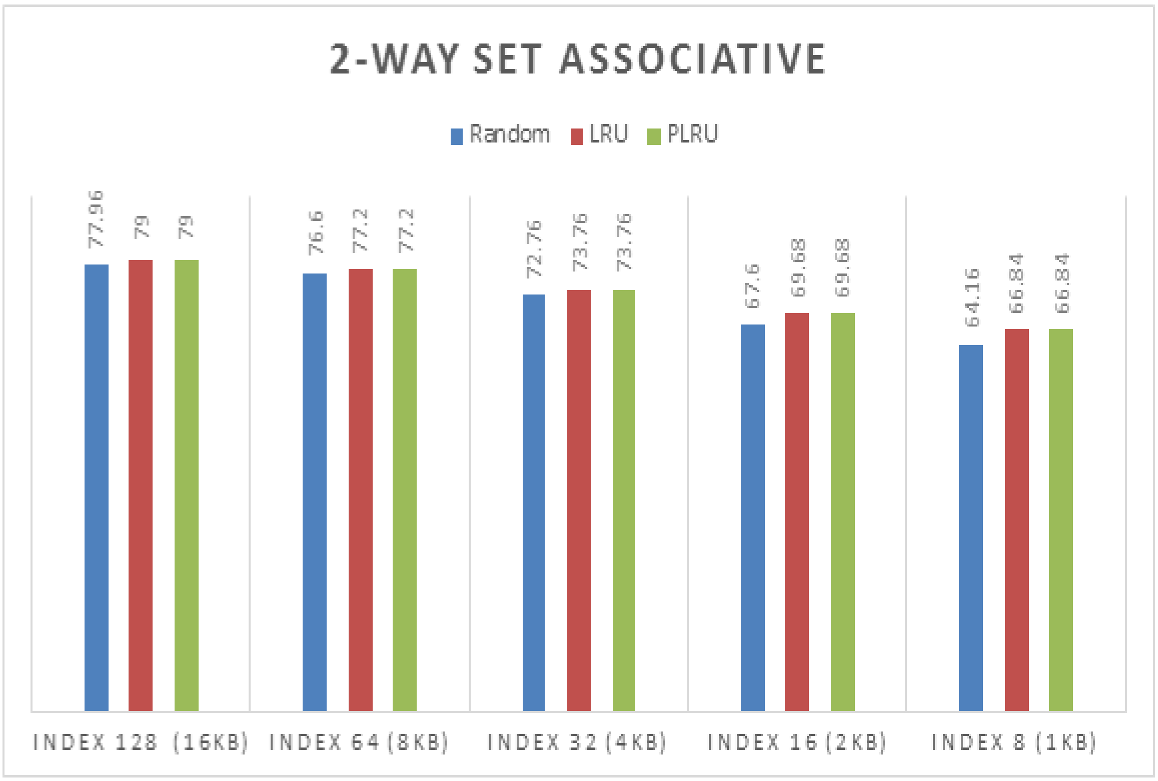


Figure 4.2 Comparison graph of replacement policies for 2-way SA

Table 4.1 Comparison graph of replacement policies for 2-way SA

Index size		**128**	**64**	**32**	**16**	**8**
Cache size(KB)		**16**	**8**	**4**	**2**	**1**
Hit Ratio %	**RANDOM**	77.96%	76.6%	72.76%	67.6%	64.16%
	LRU	79%	77.2%	73.76%	69.68%	66.84%
	PLRU	79%	77.2%	73.76%	69.68%	66.84%

In 2-way mapping, for same index and cache sizes LRU and PLRU are similar in performance but hit ratio obtained with random policy is approximately 1.5 to 2 % lower than the other two policies. Doubling the index line increases the hit rate from 2-4 %.

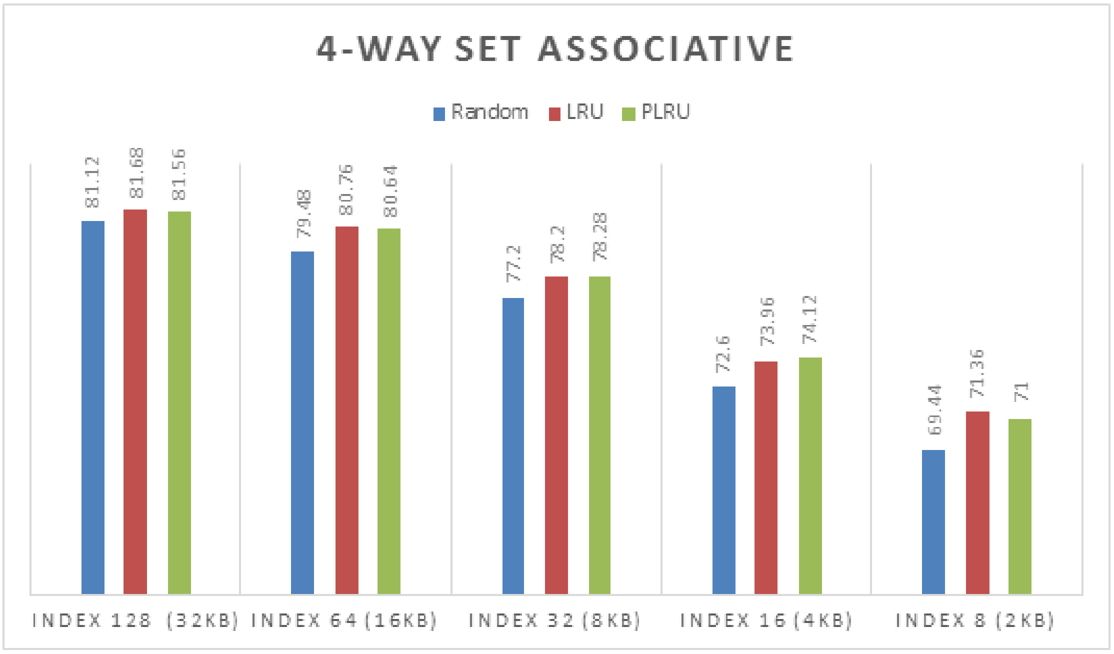


Figure 4.3 Comparison graph of replacement policies for 4-way SA

Table 4.2 Comparison of replacement policies for 4-way SA

Index size		**128**	**64**	**32**	**16**	**8**
Cache size(KB)		**32**	**16**	**8**	**4**	**2**
Hit Ratio %	**RANDOM**	81.12%	79.48%	77.2%	72.6%	69.44%
	LRU	81.68%	80.76%	78.2%	73.96%	71.36%
	PLRU	81.58%	80.64%	78.28%	74.12%	71%

In 4-way mapping, for same index and cache sizes LRU and PLRU are quite similar in performance but hit ratio obtained with random policy is approximately 1.1 to 1.2 percentage lower than the other two policies. Doubling the index line increases the hit rate from 1-4 %.

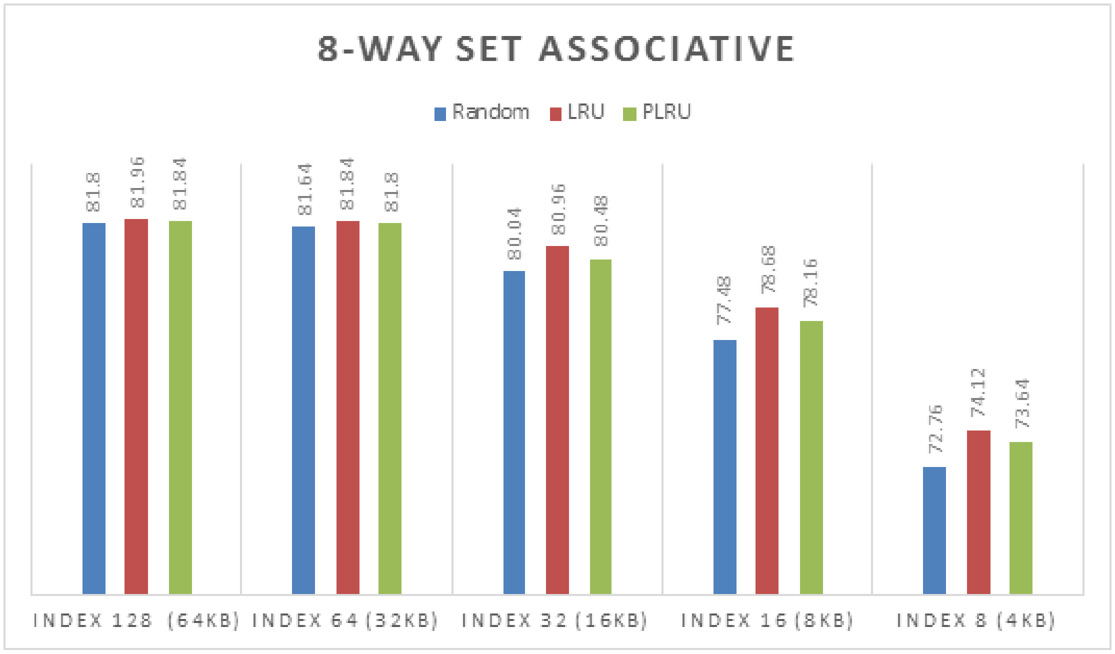


Figure 4.4 Comparison graph of replacement policies for 8-way SA

In 8-way mapping, for same index and cache sizes LRU and PLRU are quite similar in performance but hit ratio obtained with random policy is approximately 0.4 to 0.76 percentage lower than the other two policies. Doubling the index line increases the hit rate from 2-5 %.

Table 4.3 Comparison of replacement policies for 8-way SA

Index size		**128**	**64**	**32**	**16**	**8**
Cache size(KB)		**64**	**32**	**16**	**8**	**4**
Hit Ratio %	**RANDOM**	81.8%	81.64%	80.04%	77.48%	72.76%
	LRU	81.96%	81.84%	80.96%	78.68%	74.12%
	PLRU	81.84%	81.8%	80.48%	78.16%	73.64%

It has been observed that as we increase cache size or index size of cache, hit rate also increases. Moreover, there is not any significant difference between the hit rates of the LRU and PLRU but they are much better in performance than the random replacement policy. The worst results are obtained for smaller cache sizes where number of misses are higher as expected.

Depending on the cache sizes, relative efficiencies of the policies change but with a slight differences. However, LRU is an efficient replacement policy compared to Pseudo-LRU.

It has also been observed that maximum average hit ratio is obtained with LRU policy.

Taking example of 8KB cache for different ways, as we increase way size from 2-way to 4-way SA hit ratio increases by ~3.5%. However, from 4-way to 8-way SA hit ratio is increased by ~0.2%.

A common observation is that increasing the index size above 64 doesn’t gives comparable performance increment for lower number if index size.

Chapter-5

Conclusion

We have compared 3 different replacement policies for different sizes of cache and we have observed the following.

- Comparison of hit rates other than 2-Way SA cache.
    - LRU > PLRU > Random
- In 2-Way SA cache
    - LRU = PLRU > Random
    - This is because LRU and PLRU replacement policies are basically same on 2-Way SA cache.
- As we double the Way of cache (2-Way to 4-Way & 4-Way to 8-Way) the performance increases in general, but the percentage increase is not same.
    - 2-Way to 4-Way ~3.5%
    - 4-Way to 8-Way ~0.2%
    - It implies that we won’t get equal performance increment on doubling the Way-Size of cache.
    - To find the optimized Way-Size we have to strike a balance between the Cache-Size and Hit-Rate. In our case optimized Way-Size is 4.
    - Performance increment also depends upon the program in execution. So the optimized Way-Size can be different for different program.

FUTURE SCOPE

We have compared the Hit-Rate of three different replacement policies in this project. The next step can be comparing the number of gates used to implement the three replacement policies. This will help the designer to choose the best replacement policy taking into account the area and power constraints. As these constraints play a major role in today's design implementation.

The Hit-Rate also depends on the program (application) which the processor is running. If the program is more repetitive (more loops) then the Hit-Rate will improve, and it will decrease if program is more random. This can be another direction of work where we can compare the Hit-Rates with different types of programs.

References

[1] AkshayKanwar, AditiKhazanchi, LovenishSaluja, “Cache Memory Organization” International Journal Of Engineering And Computer Science ISSN:2319-7242,2013.

[2] Paulo J. D. Domingues,“Cache in a Memory Hierarchy”Retrived fromhttp://gec.di.uminho.pt/discip/minf/ac0304/ICCA04/Proc/T2-Cache.pdf.

[3] Pancham, Deepak Chaudhary, RuchinGupta, “Comparison of Cache Page Replacement Techniques to Enhance Cache Memory Performance” International Journal of Computer Applications (0975 – 8887) Volume 98– No.19, 2014.

[4] Hassan Ghasemzadeh, SepidehMazrouee, Mohammad Reza Kakoee, “Modified Pseudo LRU Replacement Algorithm” IEEE conference on Engineering of computer Based Systems (ECBS’06)pp.-376,2006.

[5] Amit S. Chavan, Kartik R. Nayak, Keval D. Vora, Manish D. Purohit and Pramila M. Chawan” A Comparison of Page Replacement Algorithms” IACSIT International Journal of Engineering and Technology, Vol.3, No.2,pp.171-174, 2011.

[6] Mohsen Soryani,Mohsen Sharifi, Mohammad Hossein Rezvani,“Performance Evaluation of Cache Memory Organizations in EmbeddedSystems”IEEE International Conference on Information Technology (ITNG'07),2007.

[7] Memory Hierarchy Design, 2015,Retrived from www.edn.com/designs/systems-design/4397051/2/Memory-Hierarchy-Design-part-1.

[8] Maria Grigoriadou, Maria Toula, Evangelos Kanidis, “Design and Evaluation of a Cache Memory Simulation Program” The 3rd IEEE International Conference on Advanced Learning Technologies (ICALT’03) 2003.

[9] Alan Jay Smith, “Line (Block) Size Choice For CPU Cache Memories” IEEE Transactions On Computers, Vol. C-36, No. 9, September 1987.

[10] Abu Asaduzzaman, “An Efficient Memory Block Selection Strategy to Improve the Performance of Cache Memory Subsystem” Proceedings of 14th International Conference on Computer and Information Technology (ICCIT 2011) 22-24, 2011.

[11] H. Al-Zoubi, A. Milenkovic and M. Milenkovic,“Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite” In ACMSE, 2004.

[12] M.MorrisMano,“Computer system Architecture” 3rdedition,Prentice HallISBN-10: 0131755633,1992.

[13] Yogesh S. Watile1, A. S. Khobragade” Design of Cache Memory with Cache Controller Using VHDL” International Journal of Innovative Research in Science, Engineering and Technology, july 2013