# CSE340: Computer Architecture

## Handout_Chapter - 5: Large and Fast: Exploiting Memory Hierarchy


Inspiring Excellence

Prepared by: Partha Bhoumik (PBK)
Course Coordinator, CSE340

# Background

In recent years, improving CPU performance has become increasingly difficult. Instead of making single CPUs faster, designers now focus on increasing the number of processors or CPU cores to handle more tasks simultaneously.

However, simply adding more cores isn't enough. A major challenge in modern computing is ensuring that data is accessed quickly and efficiently. This is why much of today's hardware research focuses on memory optimization.

**Imagine** you have a program with 1000 lines of code, written in a high-level language. When executed, this translates to 10,000 lines of RISC-V assembly instructions. While the program as a whole may access many different memory locations, at any given moment, only a small subset of instructions and memory locations are actively in use.

Consider a simple loop inside that 1000-line program:

```
for (i = 0; i < 100; i++) {
   array[i] = array[i] + 1;
}
```

Even though the entire program consists of 10,000 lines, the processor repeatedly executes just three lines of code for a significant amount of time (100 iterations). This means that during this period, the CPU is primarily focused on a small section of code and memory rather than the entire program.

Now, again, think about a function for factorial calculation,

```
int factorial(int n) {
   int result = 1;
   for (int i = 1; i <= n; i++) {
      result *= i;  // Updating the same variable repeatedly
   }
   return result;
}
```

Here too, even though the function is part of a larger program, during the execution of this function, only a single variable is being updated.

Although your larger program is going to access a lot of different memory locations; your scope of investigation is restricted to a small subset of instructions and a small subset of memory locations at the time.
This behavior is universal in all programs, no matter how complex.

From this observation, we get two important attributes that all programs have:

**Temporal Locality**: Items accessed recently are likely to be accessed again soon.
Ex. instructions in a loop

**Spatial Locality:** Items near those accessed recently are likely to be accessed soon.
Ex. sequential instruction access, array data.

We can take advantage of these two characteristics by copying the instructions and data we need into a faster location for a short period—let's say 1 second.

For example, if we could fit these instructions and data into registers for 1 second, the CPU would not need to load or store anything from memory during that time. This means the CPU can run at full speed without waiting for data. Once we are done with the data and instructions, we load a new chunk from memory and continue.

If we follow this pattern, then we reduce frequent memory accesses and keep the CPU running efficiently. This is the core concept behind **memory hierarchy**.
This idea goes beyond just registers. To keep frequently used data easily accessible, we add intermediate levels of memory instead of always relying on main memory. These levels help the CPU access data faster, improving overall performance.

# Memory Hierarchy

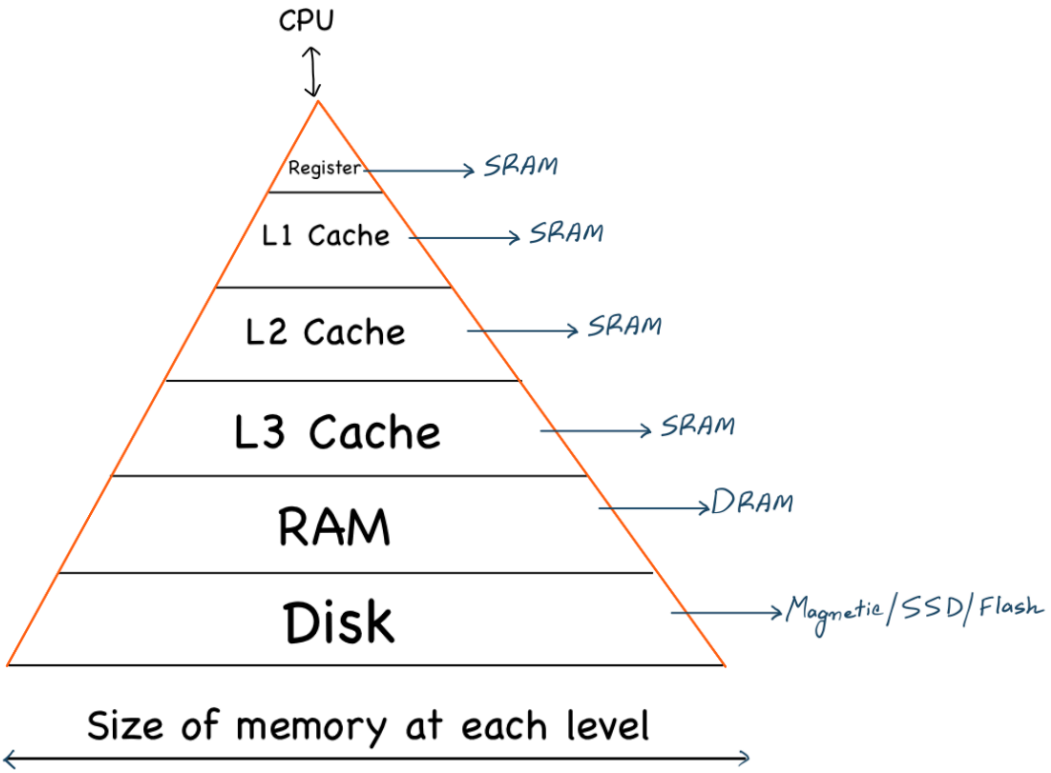



Figure: The basic structure of a memory hierarchy.

Computer memory is designed as a hierarchy to take advantage of the principle of locality. This hierarchy consists of multiple levels of memory, each with different speeds, sizes, and costs.

Key Characteristics of Memory Hierarchy:
**Faster Memory** (Closer to **CPU**)
i. Higher speed (low access time)
ii. Limited in size due to high cost.

**Slower Memory** (Far away from **CPU**)
i. Lower speed (higher access time)
ii. Larger size due to lower cost.

# DRAM vs SRAM

Dynamic Random Access Memory uses 1 capacitor and 1 transistor to store 1 bit of data. The data is stored as a charge in the capacitor and a transistor is then used to access this stored charge, either to read the value or to write. The data stored in the capacitor slowly leaks away with time, so it needs regular refreshing. Refresh means to read the contents and write them back. Because of this refreshing, we call this a Dynamic RAM

On the contrary, Static Random Access Memory uses 6-8 transistors to store 1 bit of data. SRAMs do not need to be refreshed. As long as power is applied, the value can be kept continuously.

Read this to know more:
https://www.geeksforgeeks.org/difference-between-sram-and-dram/

# Different terms related to Cache

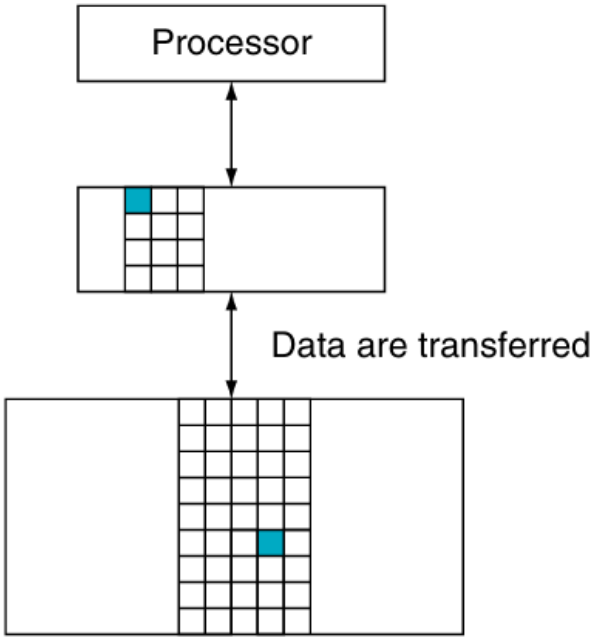



Figure: Memory Hierarchy Levels

The minimum unit of information that can be either present or not present in a cache is called a block or line as shown in the above figure.

If the data requested by the processor appears in some block in the upper level, this is called a **hit**.

If the data are not found in the upper level, the request is called a **miss**. The lower level in the hierarchy is then accessed to retrieve the block containing the requested data.

**Hit rate** or **Hit ratio** is the percentage of memory accesses found in the upper level(cache); it is often used as a measure of the performance of the memory hierarchy. (Hits/ Accesses)

**Miss rate** or **Miss ratio** (1 - Hit Ratio) is the fraction of memory accesses not found in the upper level(cache).

**Hit time** is the time required to access a level of the memory hierarchy, including the time needed to determine whether the access is a hit or a miss. Since the upper level is smaller and faster, the hit time is very low.

**Miss penalty** is the time to replace a block in the upper level with the corresponding block from the lower level, plus the time to deliver this block to the processor. Since lower-level memory is much slower, the miss penalty is high and can significantly slow down performance.

An ideal design of cache memory should try to increase the hit rate and reduce the miss penalty.

# Different types of Cache Organizations:

i. Direct Mapped Cache;
ii. Fully Associative;
iii. Set Associative.

## Direct Mapped Cache

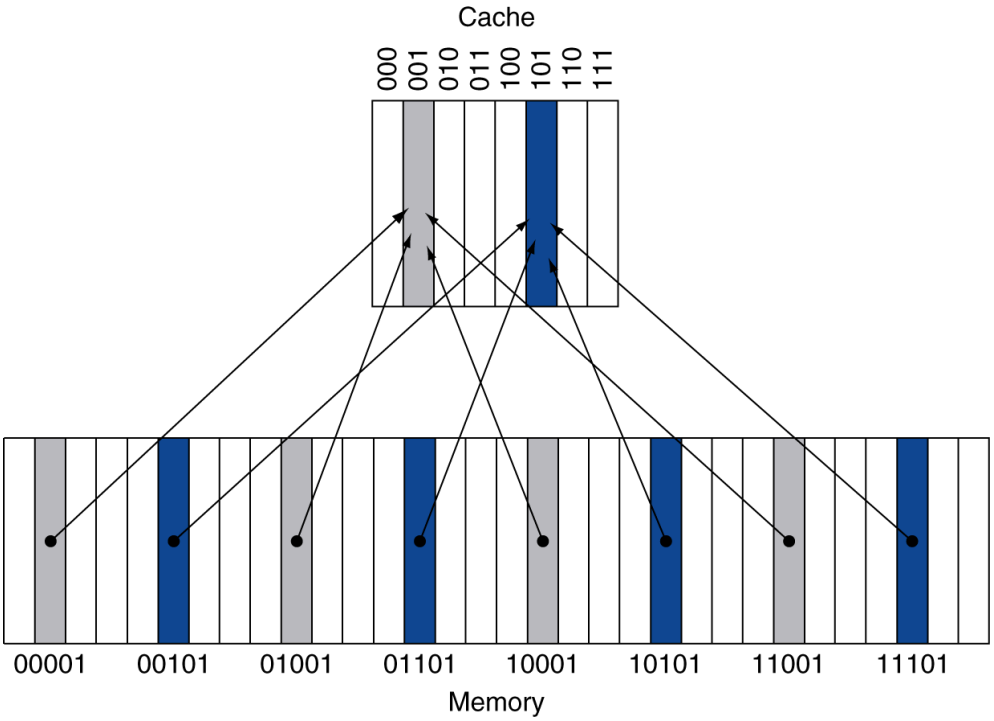
Direct-mapped cache is the simplest form of cache organization. Each block of memory maps to exactly one cache block. Lookups in a direct-mapped cache are faster. This is also easy to implement in hardware. But if two memory blocks that map to the same cache block, one overwrites the other.

Now the question is,

A main memory block will be mapped to which cache index?

$$Cache\ Index\ =\ Block\ address\ mod\ number\ of\ blocks\ in\ the\ cache$$

Follow the explanation to have a better idea.



Explanation:
The above diagram represents how main memory blocks are mapped to a cache with 8 blocks using direct-mapped cache organization.

The upper portion represents a cache, with 8 blocks labeled using 3-bit binary indices (000 to 111, or 0–7 in decimal).
Each block in the cache can hold one block of data from main memory.

The lower portion represents the main memory, showing block addresses in 5-bit binary (00001, 00101, etc.).

Now let's try to map the main memory blocks into cache memory.

Mapping:
$$Cache\ Index\ =\ Block\ address\ mod\ number\ of\ blocks\ in\ the\ cache$$

For example:

| Main Memory Block | Formula | Mapped Cache Index |
|---|---|---|
| 00001 (1) | 1 mod 8 = 1 | 001 |
| 00101 (5) | 5 mod 8 = 5 | 101 |
| 01001 (9) | 9 mod 8 = 1 | 001 |
| 01101 (13) | 13 mod 8 = 5 | 101 |
| 10001 (17) | 17 mod 8 = 1 | 001 |
| 11001 (25) | 25 mod 8 = 1 | 001 |
| 11101 (29) | 29 mod 8 = 5 | 101 |

From this table, you can also see some conflicts, such as,
00001, 01001, 10001, 11001, all these main memory addresses are mapped to a single cache index 001.

# Address Subdivision

When a CPU accesses memory, it uses a memory address. This address is split into three main parts:

$$Memory\ Address\ size\ = [Tag\ bits]\ +\ [Index\ bits]\ +\ [Offset\ bits]\ ;\ [\ ]\ =\ size\ of$$

Index Bits:

This field is used to select the specific cache block where the main memory block will be mapped. In a direct-mapped cache, the Index field points to a single cache block.

Formula to calculate the size of the index bits(for direct-mapped):

$$Index\ bit\ size\ =\ log_2(Number\ of\ cache\ blocks)$$

Example:

If cache has 64 blocks → Index bit size = $\log_2(64) = 6$ bits

Tag Bits:

This field is used to check if the block in the cache is the correct one or not(to detect a cache hit).

Formula to calculate the size of the Tag bits(for direct-mapped):

$$Tag\ bits\ =\ Total\ address\ bits\ -\ Index\ bits\ -\ Offset\ bits$$

Example:

If address is 32 bits, index = 6 bits, offset = 4 bits:

Tag bit size = $32 - 6 - 4 = 22$ bits

Offset Bits:

This field is used to select the exact byte within a block.

Formula to calculate the size of the Offset bits(for direct-mapped):

$$Offset\ bit\ size\ =\ log_2(Block\ size\ in\ bytes)$$

Example:
If Block size = 16 bytes, Offset bit size = $\log_2(16) = 4$ bits

Valid Bit:

This field is used to identify whether the cache block contains valid data or not.
Remember: the valid bit is not part of the memory address.

Size: Always 1 bit per cache block.
0 → Block is invalid (empty or outdated)
1 → Block contains valid data

**For a better understanding, please follow this simulation.**

Setup:
Cache Size = 8 blocks;
Block Size = 1 byte or 8 bits;
Offset bits size = Log2(1) = 0; Hence, no Offset bits in the address.

| Index | V | Tag | Data |
|---|---|---|---|
| 000 | N | | |
| 001 | N | | |
| 010 | N | | |
| 011 | N | | |
| 100 | N | | |
| 101 | N | | |
| 110 | N | | |
| 111 | N | | |

Access 1: Memory Address = 22 (10110)
Index = 22 mod 8 = 6 = 110
Tag bits will be the remaining msb bits = 10

At the start, the cache is empty:
All valid bits = 0, meaning the cache doesn't contain any data yet.
So even though index 6 is pointing to the correct location, the valid bit is not set.
Hence, Access 1 is a miss.

| Word addr | Binary addr | Hit/miss | Cache block |
|---|---|---|---|
| 22 | 10 110 | Miss | 110 |

As this is a miss, now the CPU will fetch the corresponding block from main memory and place it in the cache.

| Index | V | Tag | Data |
|---|---|---|---|
| 000 | N | | |
| 001 | N | | |
| 010 | N | | |
| 011 | N | | |
| 100 | N | | |
| 101 | N | | |
| 110 | Y | 10 | Mem[10110] |
| 111 | N | | |

Access 2: Memory Address = 26 (11010)
Index = 26 mod 8 = 2 = 010
Tag bits will be the remaining msb bits = 11

Access 2 is also a miss.

| Word addr | Binary addr | Hit/miss | Cache block |
|---|---|---|---|
| 22 | 10 110 | Miss | 110 |
| 26 | 11 010 | Miss | 010 |

As this is a miss, now the CPU will fetch the corresponding block from main memory and place it in the cache.

| Index | V | Tag | Data |
|---|---|---|---|
| 000 | N | | |
| 001 | N | | |
| 010 | Y | 11 | Mem[11010] |
| 011 | N | | |
| 100 | N | | |
| 101 | N | | |
| 110 | Y | 10 | Mem[10110] |
| 111 | N | | |

Access 3: Memory Address = 22 (10110)
Index = 22 mod 8 = 6 = 110
Tag bits will be the remaining msb bits = 10
In this case, Tag bits match with the table and the valid bit is set to 1 (Y)
So, Access 3 is a hit.

| Word addr | Binary addr | Hit/miss | Cache block |
|---|---|---|---|
| 22 | 10 110 | Miss | 110 |
| 26 | 11 010 | Miss | 010 |
| 22 | 10 110 | Hit | 110 |

Access 4: Memory Address = 26 (11010)
Index = 26 mod 8 = 2 = 010
Tag bits will be the remaining msb bits = 11
In this case, Tag bits match with the table and the valid bit is set to 1 (Y)
So, Access 4 is a hit.

| Word addr | Binary addr | Hit/miss | Cache block |
|---|---|---|---|
| 22 | 10 110 | Miss | 110 |
| 26 | 11 010 | Miss | 010 |
| 22 | 10 110 | Hit | 110 |

| | | | |
|---|---|---|---|
| 26 | 11 010 | Hit | 010 |

Access 5: Memory Address = 18 (10010)
Index = 18 mod 8 = 2 = 010
Tag bits will be the remaining msb bits = 10
In this case, although the valid bit is set to 1 but the tag bits do not match with the table.
So, Access 5 is a miss.

| Word addr | Binary addr | Hit/miss | Cache block |
|---|---|---|---|
| 22 | 10 110 | Miss | 110 |
| 26 | 11 010 | Miss | 010 |
| 22 | 10 110 | Hit | 110 |
| 26 | 11 010 | Hit | 010 |
| 18 | 10010 | Miss | 010 |

As this is a miss, now the CPU will fetch the corresponding block from main memory and place it in the cache.

| Index | V | Tag | Data |
|---|---|---|---|
| 000 | N | | |
| 001 | N | | |
| 010 | Y | 11 | Mem[11010] |
| 011 | N | | |
| 100 | N | | |
| 101 | N | | |
| 110 | Y | 10 | Mem[10110] |
| 111 | N | | |

Can you calculate the Hit and Miss ratio now?
Hit ratio = Number of Hit / Total Access
Miss ratio = Number of Miss / Total Access

Here, Total access = 5, Number of Hit = 2, Number of Miss = 3
So,
Hit ratio = 2/5; 40% of the time, the cache served the data without needing to access main memory (hit).
Miss ratio = 3/5;  60% of the time, the data was not in the cache and had to be fetched (miss), causing a delay.

## Index Bits Calculation

Example1:
Given,  Cache size = 16 blocks
Find the index bits for the block address 125.
Or,
Find the cache index where the block address 125 will be mapped?

**Solution**:
Cache Index = Block Address Mod Number of cache blocks
= 125 mod 16
= 1111101 mod $2^4$
= 1101 = 13

So, Block address 125 will be mapped to cache index 13(1101)

Example2:
Given,  Cache size = 16 blocks; Block Size = 64 bits
Find the index bits for the memory address 320.

**Solution**:
Memory address = 320 =  101000000
Index bits size = log2(number of cache blocks) = log2(16) = 4
Block Size = 64 bits = 8 bytes
Offset bits size = log2(block size in bytes) = log2(8) = 3
Tag bits size = Total address size - Index bits size - Offset bits size
= 9 - 4 - 3 = 2

| Tag bits | Index bits | Offset bits |
|---|---|---|
| 10 | 1000 | 000 |

So, index bits = 1000

**Alternate solution,**

Block address = memory address/size of block in bytes (take the floor value of the division)

= 320 / 8 = 40

Cache Index = Block Address Mod Number of cache blocks

= 40 mod 16

= 101000 mod $2^4$

= 1000 = 8

## How to determine a cache hit or miss?

Given an address and you need to determine whether it will be a cache hit or miss.

Steps:

1. Find the index, tag and offset bits from the given address;
2. Use the index bit to find the correct block in the cache;
3. If (the valid bit of that cache index == 1):
    If (tag bits of the address == the tag bits of that cache index):
        Cache Hit;
    Else:
        Cache Miss;
  Else:
      Cache miss;

Example: Given a cache table:

| Index | Valid bit | Tag | Data |
|---|---|---|---|
| 000 | 1 | 100 | Data |
| 001 | 0 | | |
| 010 | 1 | 011 | Data |
| 011 | 1 | 001 | Data |

| | | | |
|---|---|---|---|
| 100 | 0 | | |
| 101 | 1 | 000 | Data |
| 110 | 1 | 010 | Data |
| 111 | 0 | | |

Given, the memory address 100110; determine whether it will be a cache hit or miss?
Note: the block size is 1 byte.

Solution:
The cache has 8 blocks → Index size = $\log_2(8) = 3$ bits
Block size = 1 byte → Offset = $\log_2(1) = 0$ bits
Total address size = 6 bits → Tag size = $6 - 3 - 0 = 3$ bits

Breakdown of the address 100110:
Tag = 100
Index = 110

From the cache table:
1. Valid bit at index 110 = 1
2. Now we check if the tag bits match or not;
   Tag bit from the table for index 110 is 010
   Which is not the same as the tag bit generated from the table.

So, it will be a miss.

## Block Size Considerations

### Benefits of Larger Block Size

Reduces Miss Rate (Initially) => Due to spatial locality: if you access address A, you are likely to access A+1, A+2, etc. A larger block brings more nearby data into the cache with each miss.
So, fewer memory accesses. Hence, fewer misses.

### Issues of Larger Block Size (Especially in Fixed Cache Size)

1. Fewer Blocks in the Cache: If total cache size is fixed, larger blocks = fewer blocks. This leads to more conflict misses (more data fighting for fewer slots).
2. Cache Pollution: If you load a large block but only use a few bytes, the rest is wasted. This "pollutes" the cache and evicts potentially useful data.
3. Increased Miss Penalty: Larger block = more data to fetch from main memory on a miss. Takes longer → higher latency on each miss. This can cancel out the benefit of reduced miss rate.

### Techniques to Reduce Penalty

1. Early Restart: Start sending data to the CPU as soon as the requested word is fetched. Don’t wait for the entire block.
2. Critical-Word-First: Request the word you need first, even if it’s in the middle of the block. The remaining words can be loaded in the background.

## What happens when a Cache Miss occurs

In case of any cache miss CPU must fetch the data from the next level of memory which is usually the main memory. This causes a pipeline stall. (CPU has to pause while waiting for data)

Instruction Cache Miss: It happens when the CPU tries to get an instruction to run, but that instruction is not found in the cache. The processor stops fetching instructions until the required instruction is brought in from memory. (Restart of the instruction fetch process)
Data Cache Miss: It happens when a load instruction refers to data that is not present in the cache. To solve this, CPU completes data access.

# Cache Hit on write operation

A write hit occurs when the CPU wants to store (write) data to a memory location and that location is already present in the cache. Since the data is already in the cache, the CPU can perform the write quickly. But how this write is handled depends on the cache's write policy.
Different cache write policies to follow for cache write hit:
i. Write-Through;
ii. Write-Back;

## Write-Through:

On a write hit,
Data is written to both the cache and main memory simultaneously. This keeps memory always up-to-date. It also ensures memory and cache are always in sync.
This approach is a bit slower because every write includes accessing the slower main memory.
Solution: Using a write buffer. Holds data waiting to be written to memory. Allows the CPU to continue without stalling—only stalls if the buffer is full.

## Write-Back:

On a write hit,
Data is updated in the cache block only. Afterwards, mark the cache block as dirty. When a dirty block is replaced, write it back to memory.
You can use a write buffer here, too. A write buffer temporarily holds the dirty block while the new block is fetched into the cache.
This allows the memory system to:
i. Start reading in the new block right away.
ii. Write the old dirty block to memory in the background (asynchronously).
This overlap saves time and helps avoid CPU stalls.

## Cache Miss on write operation

A write miss occurs when the CPU wants to store (write) data to a memory location, but that location is not present in the cache. Since the cache does not have the data block yet, it must decide what to do — and this depends on the cache's write policy.

Different cache write policies to follow for cache write miss:
i. Write Allocation.
ii. No-Write Allocation.

### Write Allocation:

On a write miss,
The block is brought into the cache, and then the write is performed in the cache using either write-through or write-back.

### No-Write Allocation:

On a write miss,
The write is performed directly in the main memory, and the block is not loaded into the cache.

Write Around = Write-Through + No Write Allocate. Explore yourself.

# Average Memory Access Time(AMAT)

AMAT = Hit time + Miss rate × Miss penalty

Example: Given that the CPU has a 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, and I-cache miss rate = 5%, find the AMAT and how many cycles are required per instruction?

Solution:

Miss penalty = 20 cycles = 20 x 1ns = 20 ns
AMAT = Hit time + Miss rate × Miss penalty
= 1 + (5/100) x 20
= 2
AMAT = 1 + 0.05 × 20 = 2ns
1ns = 1 cycle
2 ns = 2 cycle
So, 2 cycles per instruction

## Multilevel Cache

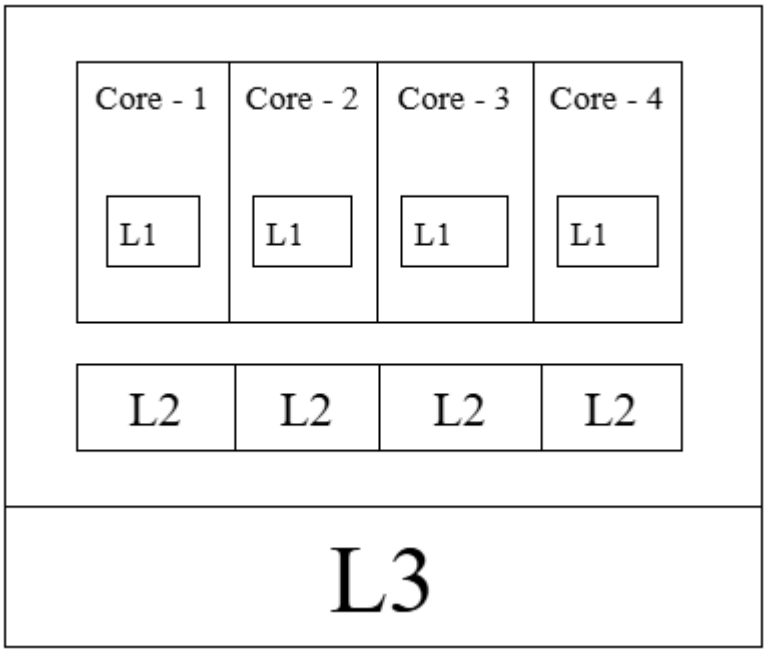



Figure: Multilevel Cache

# Formulas

1. $Cache\ Index\ = Block\ address\ mod\ number\ of\ blocks\ in\ the\ cache$ (Direct mapped cache)
2. $Cache\ Index\ size\ = log_2(Number\ of\ cache\ blocks)$
3. $Offset\ bit\ size\ = log_2(Block\ size\ in\ bytes)$
4. $Tag\ bit\ size\ = Total\ Address\ Size\ - Cache\ Index\ size\ - Offset\ bit\ size$
5. $Memory\ Address\ consists\ of\ Tag\ bits, Index\ bits, Offset\ bits$ [$Order\ is\ fixed$]
6. $Block\ Address\ = \frac{Memory\ Address}{Block\ Size\ in\ bytes}$ [Always take the floor value of this answer.]
7. $AMAT\ = Hit\ time\ + Miss\ rate \times Miss\ penalty$
8. $Hit\ Ratio\ = \frac{Total\ Hit}{Total\ Access}$
   $= 1\ - Miss\ Ratio$
9. $Miss\ Ratio\ = \frac{Total\ Miss}{Total\ Access}$
   $= 1\ - Hit\ Ratio$