MPCA Unit 3

Introduction

CPU performance increases a lot faster(55%) than memory performance(7%).
So there is a bottleneck of sorts because of the slower increase in the speed of memory.

Faster Memory is more expensive per bit than Slower Memory.

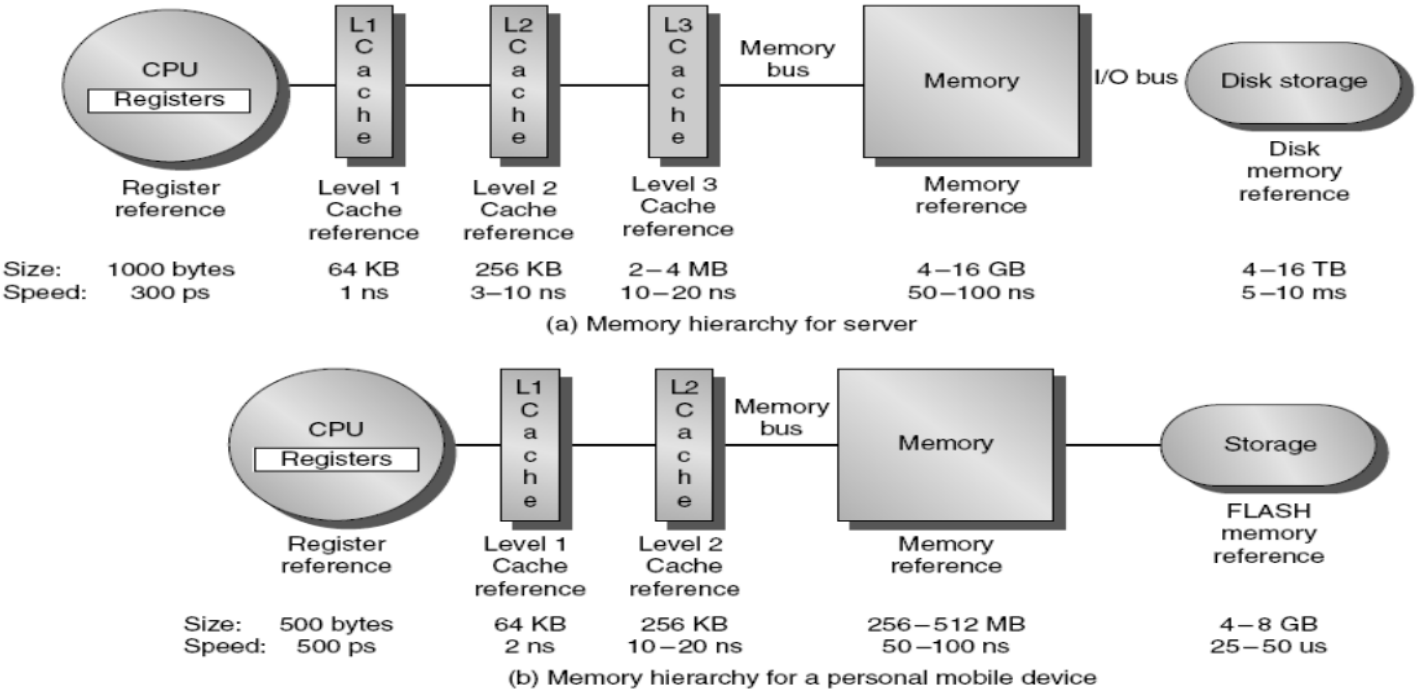
1. Memory Latency is the time it takes to transfer a word of data to or from memory
2. Memory Bandwidth is the number of bits or bytes that can be transferred in one second.

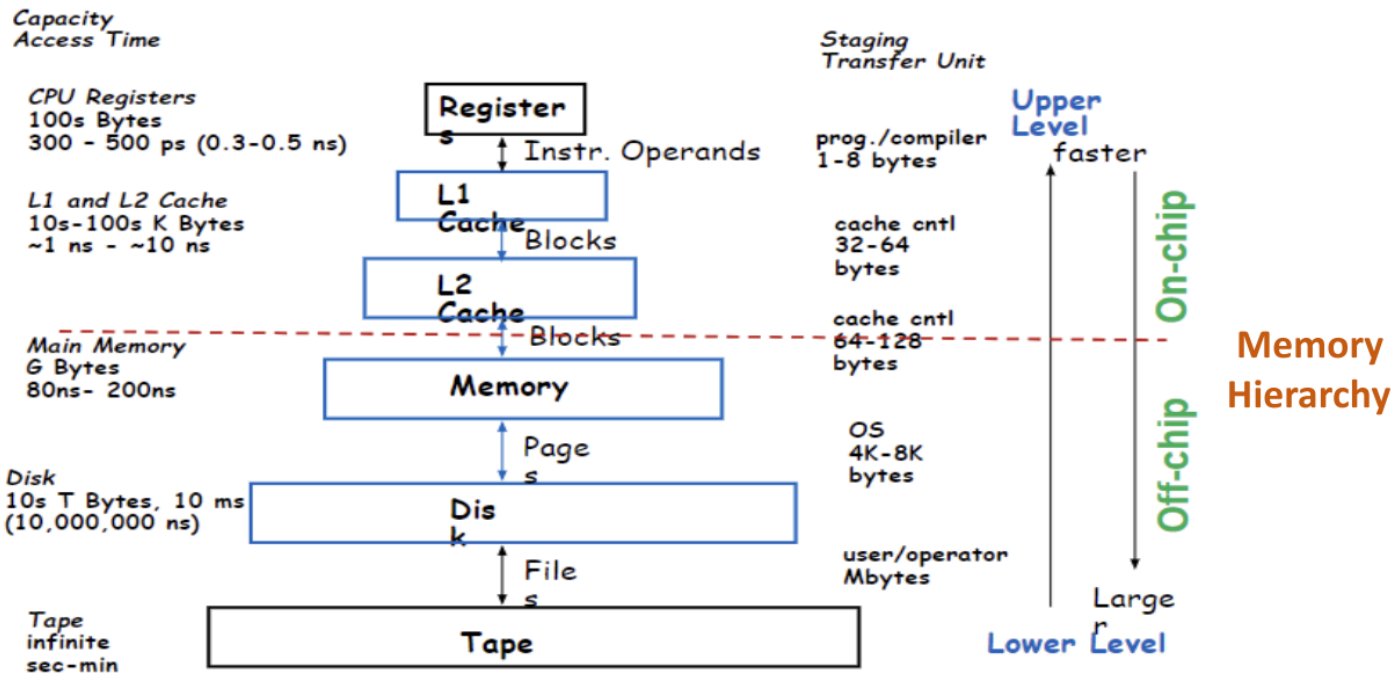
Requirement Of Programmers :

1. Unlimited amount of memory
2. low latency

What is expected of Designers :
To provide a large enough memory allotment with reasonable speed at an affordable cost.

Memory Hierarchy

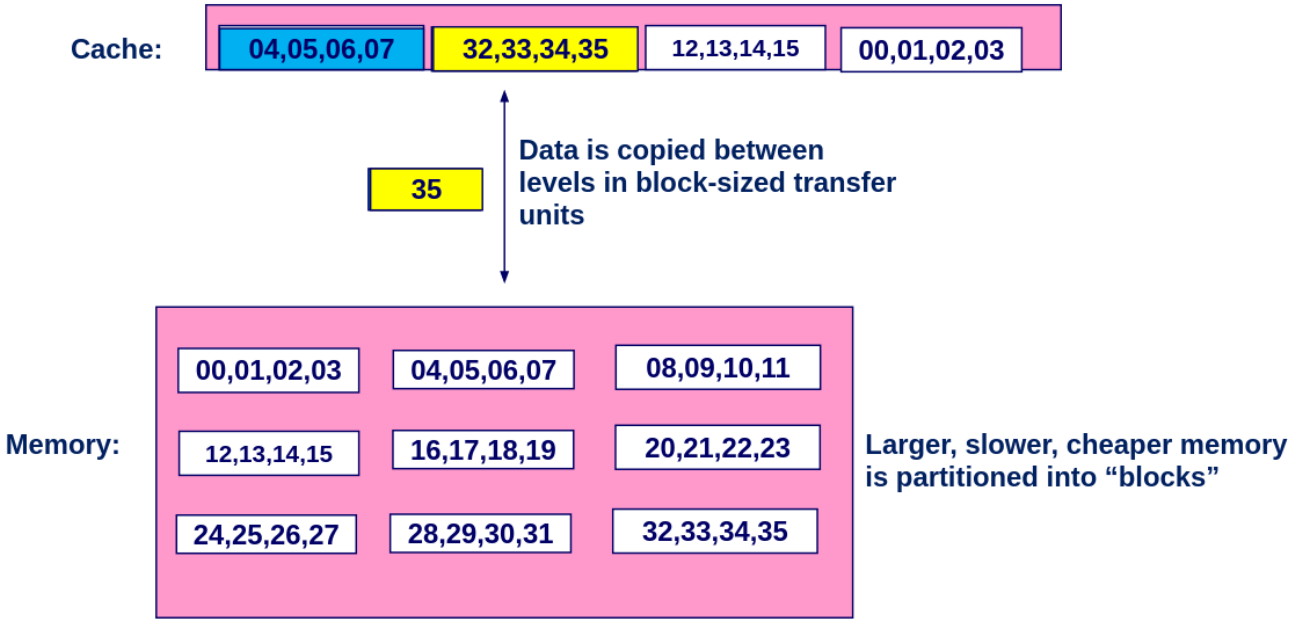




Cache

Cache memory is an architectural arrangement which makes the main memory appear faster to the processor than it really is.

Basic working principle :
Small portions of reused blocks of a slower memory are stored into a smaller but faster memory set called a "cache"



90/10 rule comes from empirical observation:
"A program spends 90% of its time in 10% of its code"

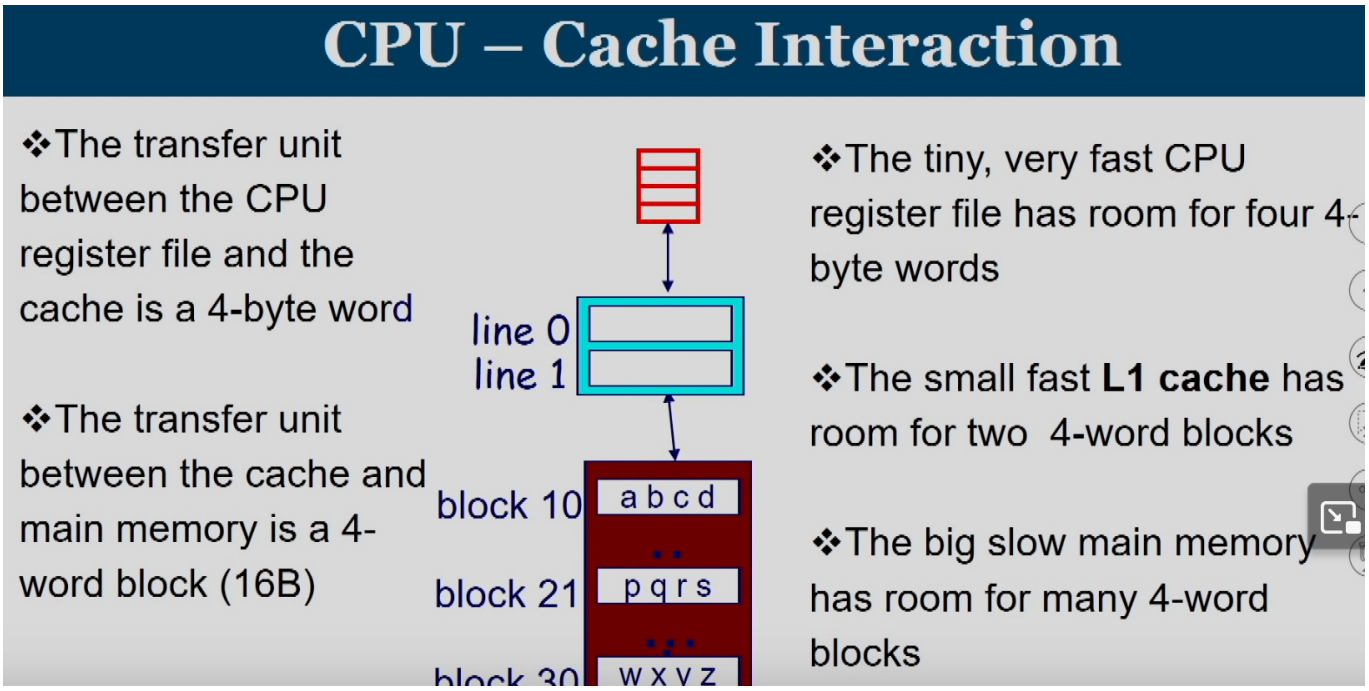
Cache memory is based on the property of computer programs known as “locality of reference”.

- Temporal locality (locality in time): if an item is referenced, it will tend to be referenced again soon.
- Spatial locality (locality in space): if an item is referenced, items whose addresses are close by will tend to be referenced soon.

Another type of locality not in syllabus : Arithmetic Locality.

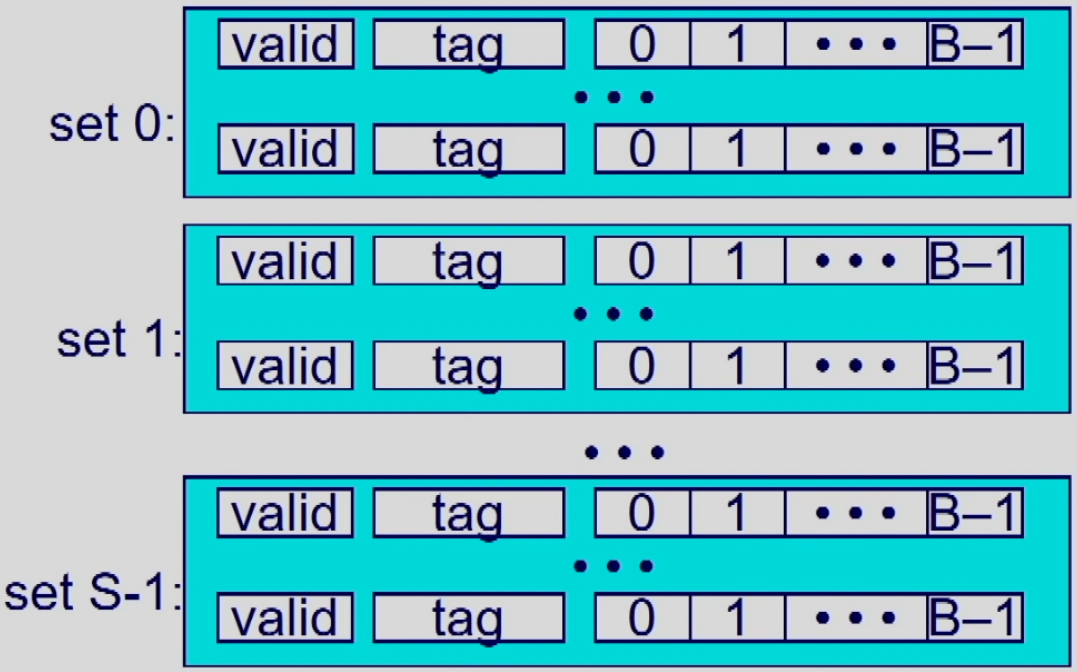
Cache Fundamentals

- Block/Line : Minimum unit of information that can be either present or not be present in a cache level.
- Hit : An access where the data requested by the processor is present in the cache.
- Miss : An access where the data requested by the processor is not present in the cache.
- Hit Time: Time to access the cache memory block and return the data to the processor.
- Hit Rate / Miss Rate : Fraction of memory access found (not found) in the cache.



Note : Transfer Unit between the disk and cache is in terms of a 4 word block. Transfer Unit between the CPU and cache is a 4 byte word.

General Organization of a Cache



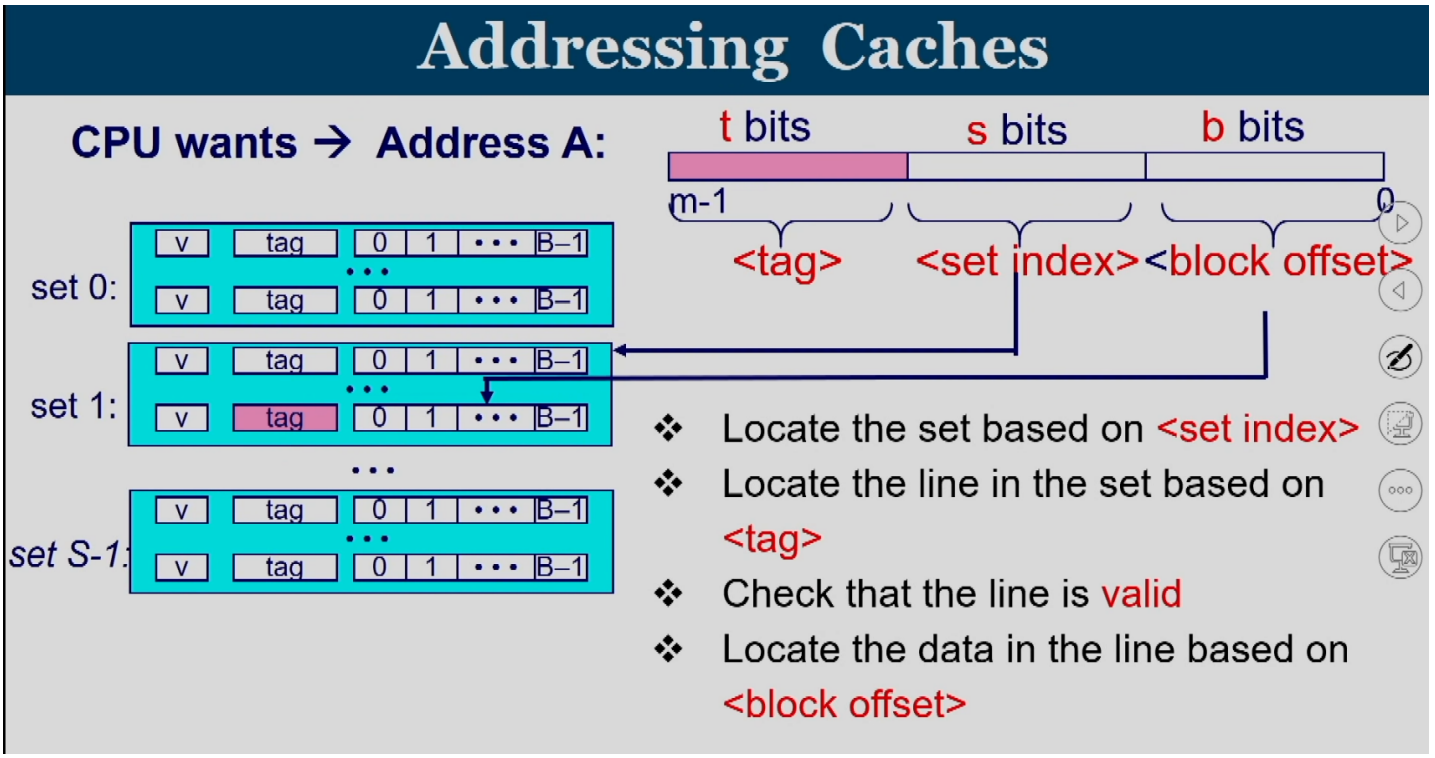
- 1 valid bit per line
- t tag bits per line
- 2^b bytes per cache block, B = 2^b
- There is E lines per set.

  S = 2^s

  Cache size :

$$C = B * E * S$$

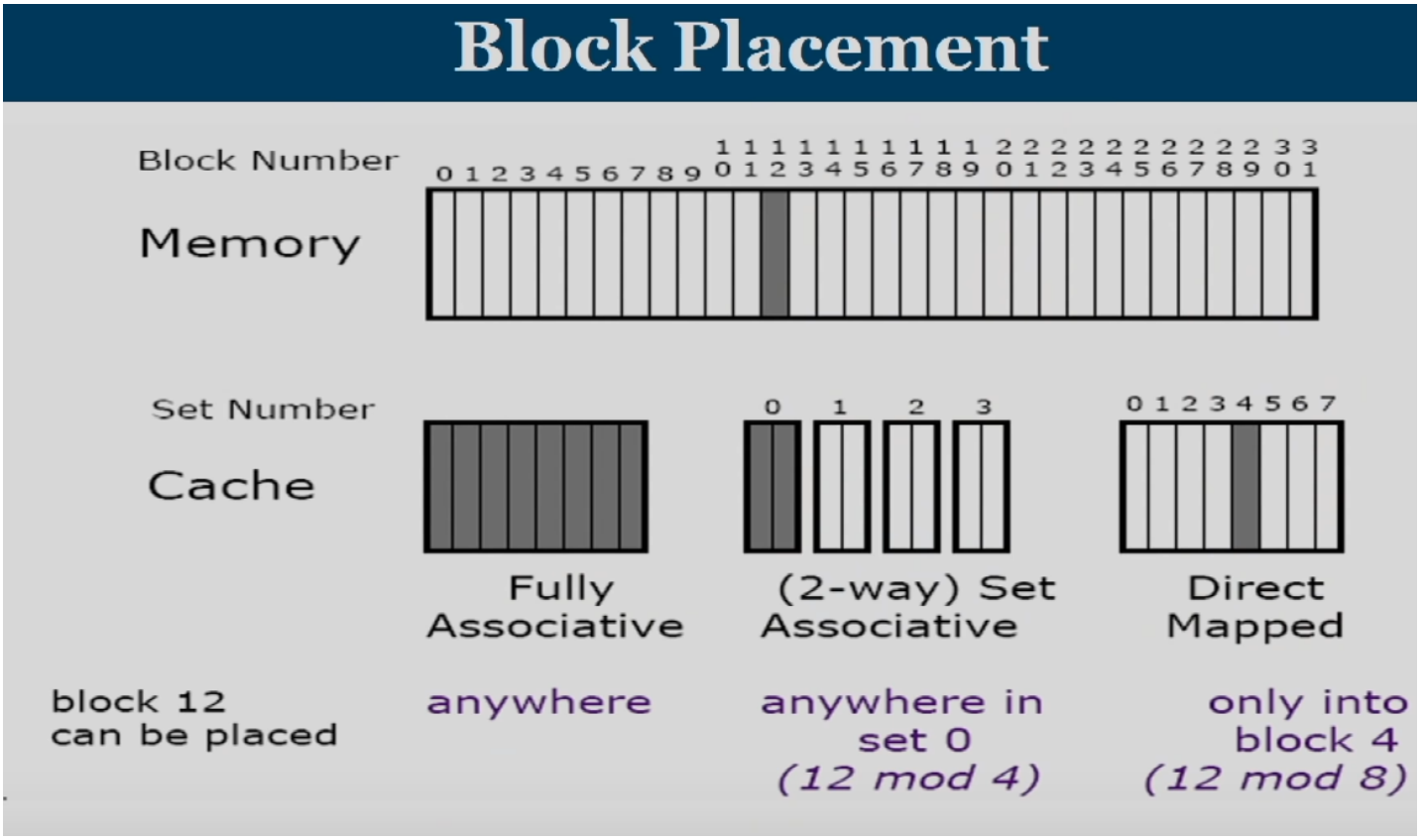
Cache Addressing



Four Cache Memory Design Choices.

1. Where can a block be placed in the cache ? -: Block Placement.
2. How is a block found if its in the upper level ? -: Block Identification.
3. Which block should be replaced on a miss ? -: Block Replacement.
4. What happens on a write ? -: Write Strategy.

Block Placement



Different Types of Block Placements

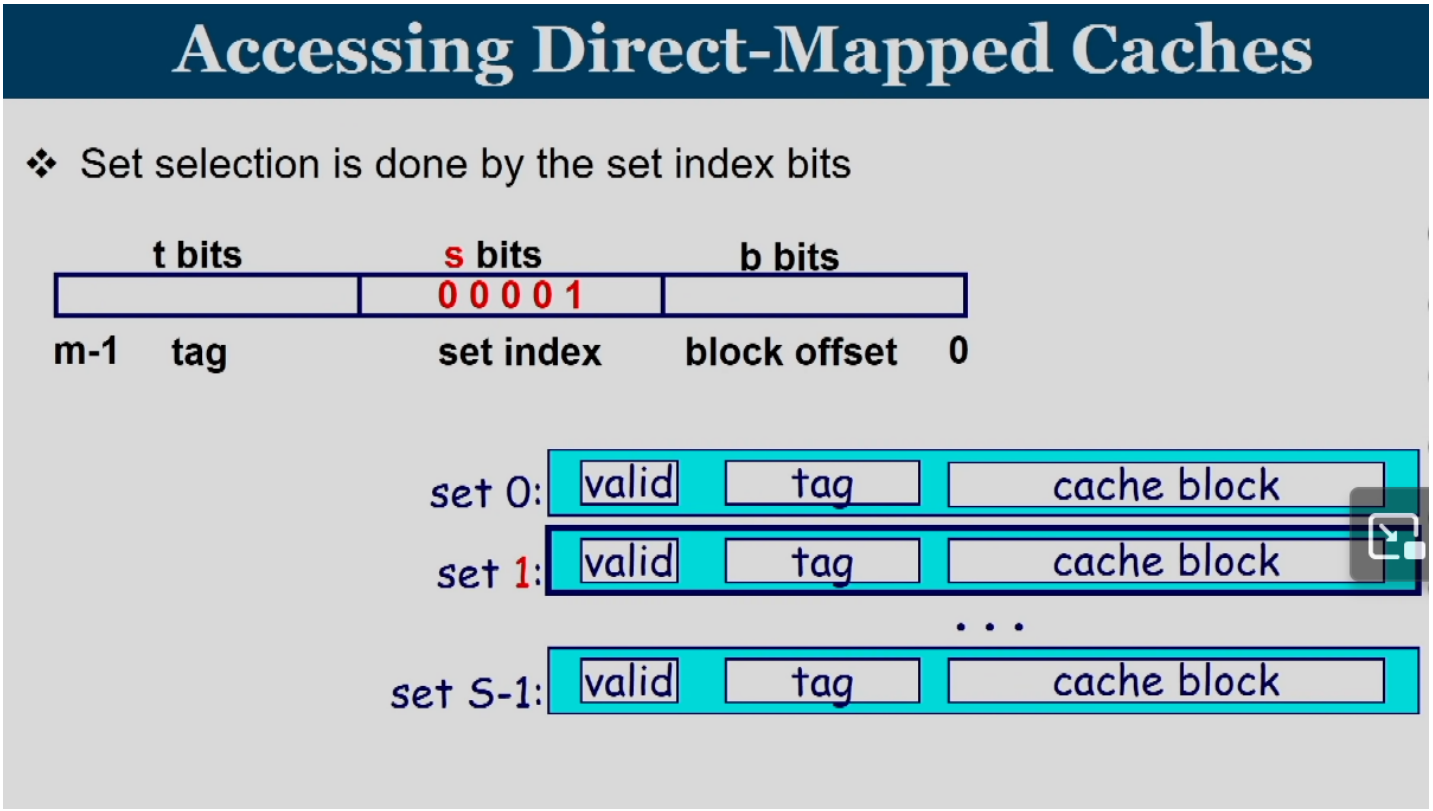
1. Direct Mapped
2. Set Associative
3. Fully Associative

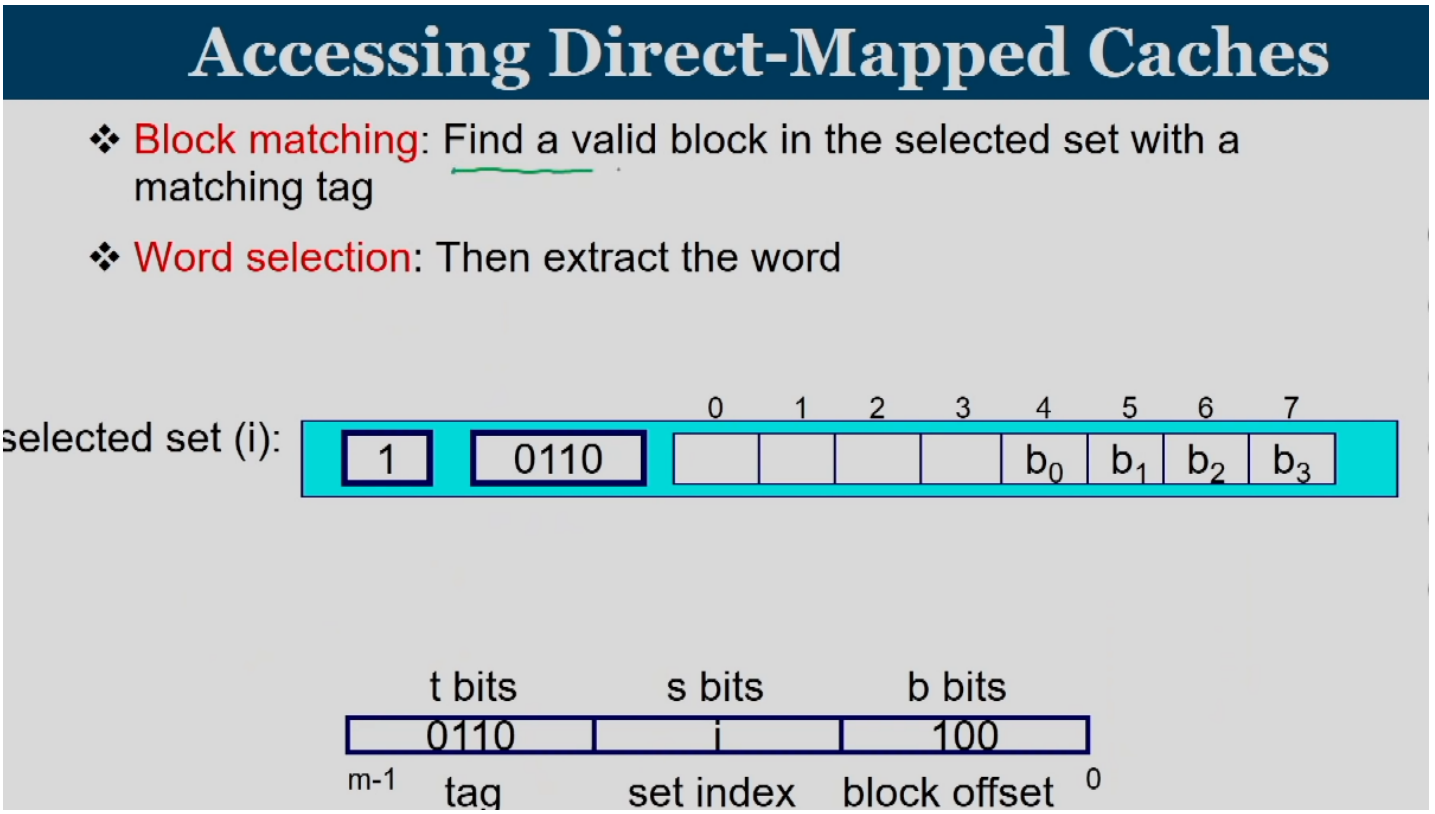
Direct Mapping

Mapping Function is the modulus function. N mod L (N is the number of blocks in main memory, L is the number of lines)

Direct Mapped Cache

- Simplest kind of cache, easy to build.
- Only 1 tag compare required per access.
- Characterized by exactly one line per set.



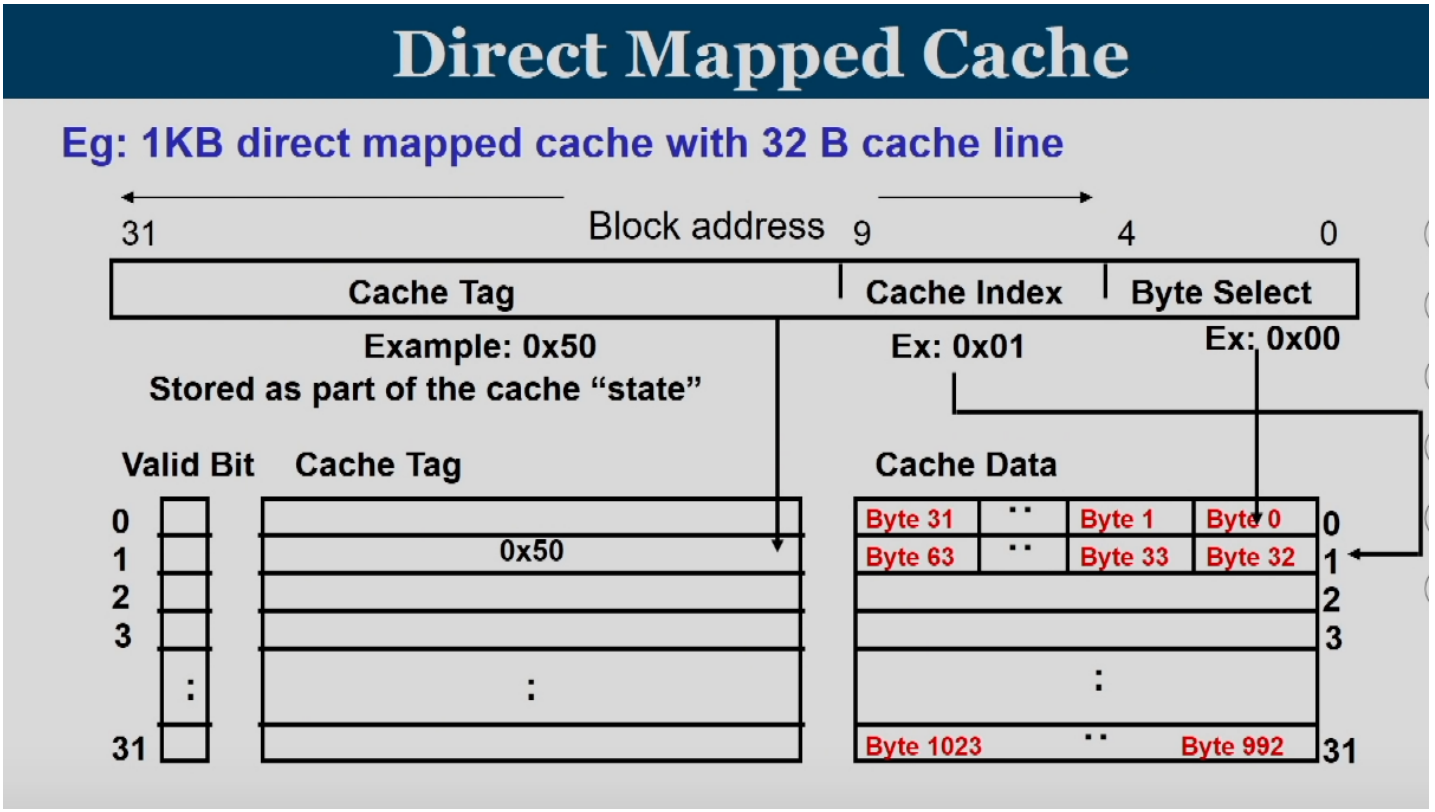


Steps >>

1. Find a valid block, To do this the valid bit must be *1*, valid bit is the first bit.
2. The tag bits in the cache line must match the tag bits in the address.

Once Conditions *1* and *2* are met, we call this a cache hit.

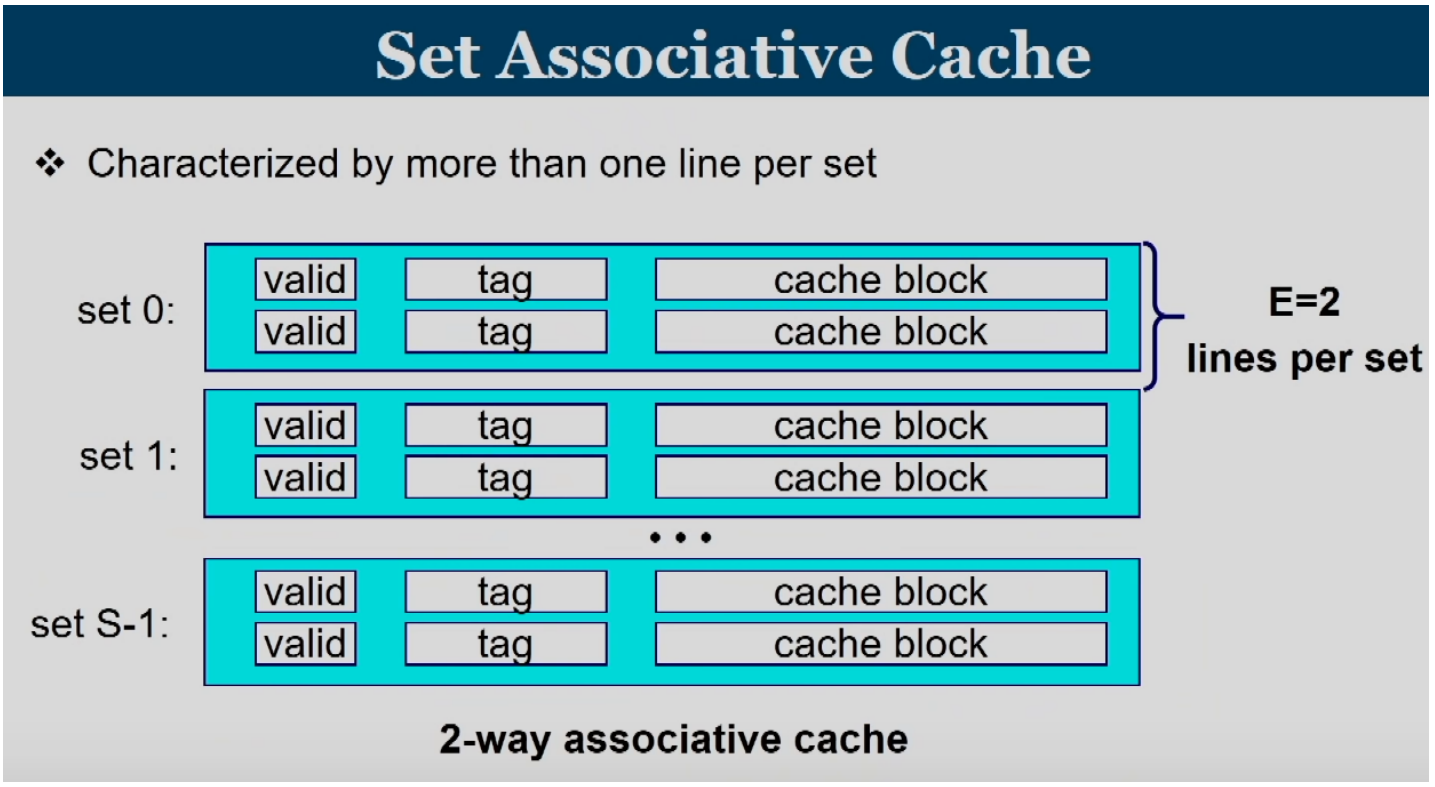
If Cache Hit occurs the offset selects the starting bit

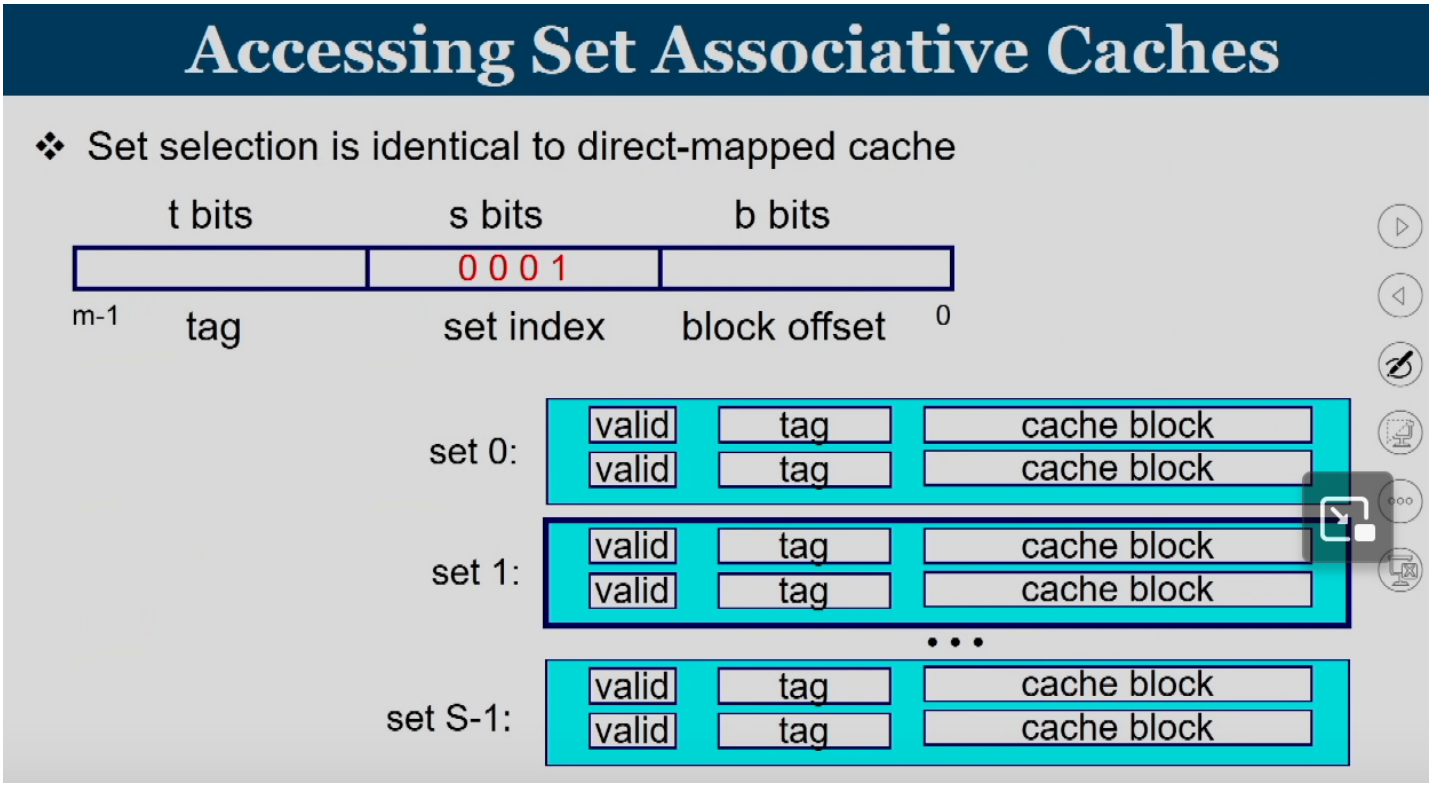


2 way Set Associative Mapping

Mapping Function is a modified modulus function. N mod (L/n) (n : n-way set associative mapping)

Set Associative Cache





Steps >>

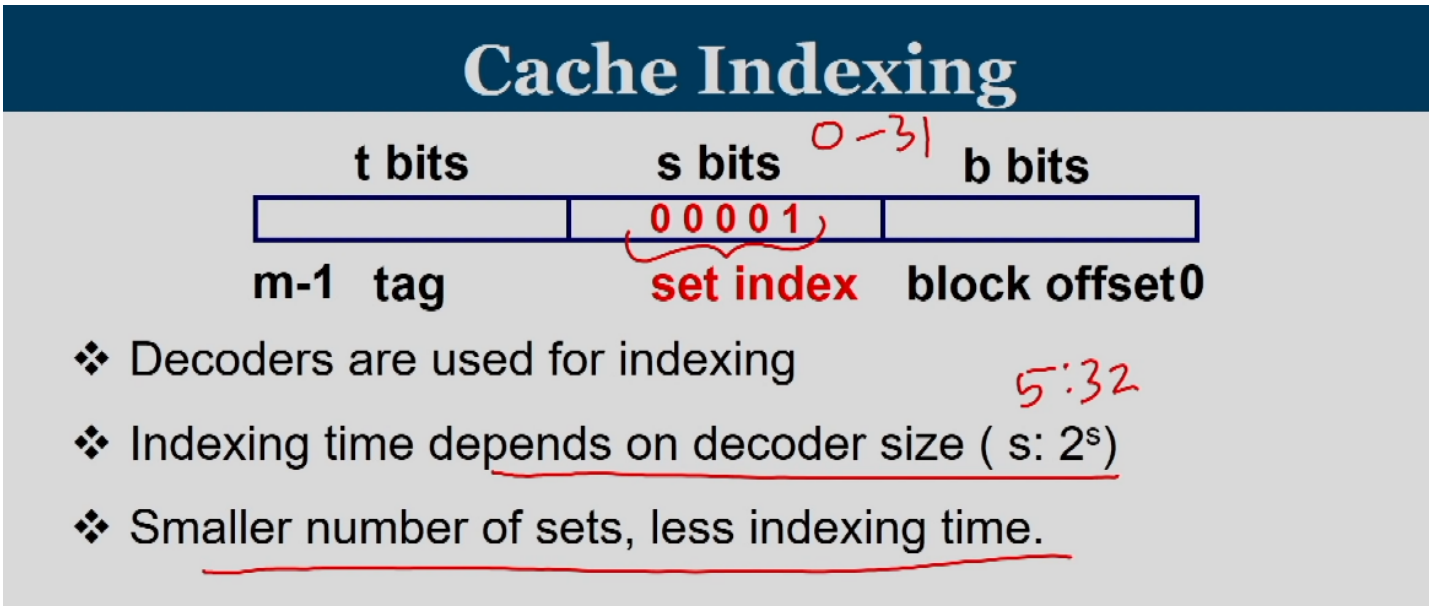
1. Set bits provides the index, this selects a set. This time instead of one, we have 2 possible cache lines.
2. To find all valid lines, valid bit must be *1*.
3. Then to find the correct line, tag bits are matched.

Once *1*, *2*, *3* happen, cache hit is said to happen.

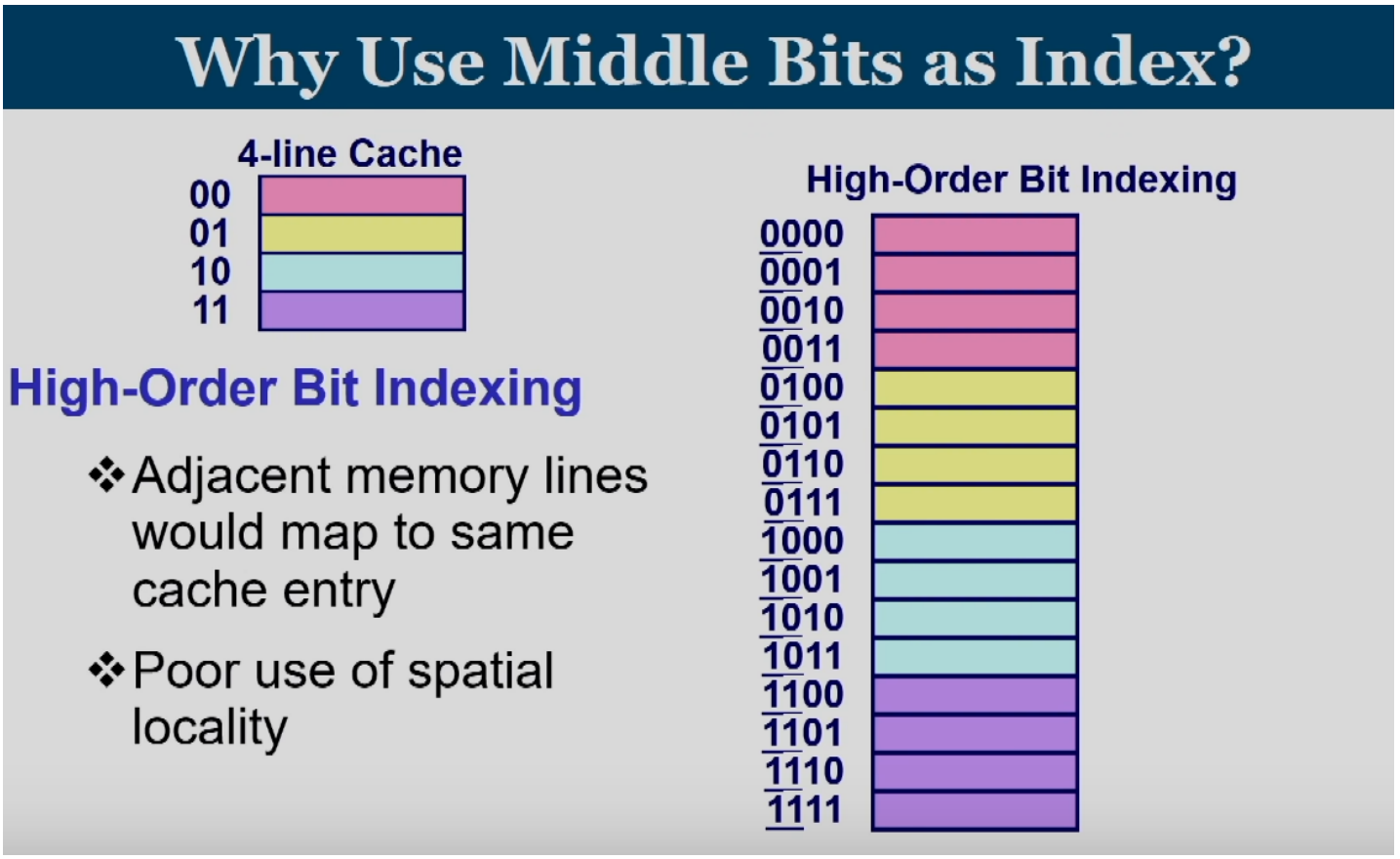
Fully Associative Mapping

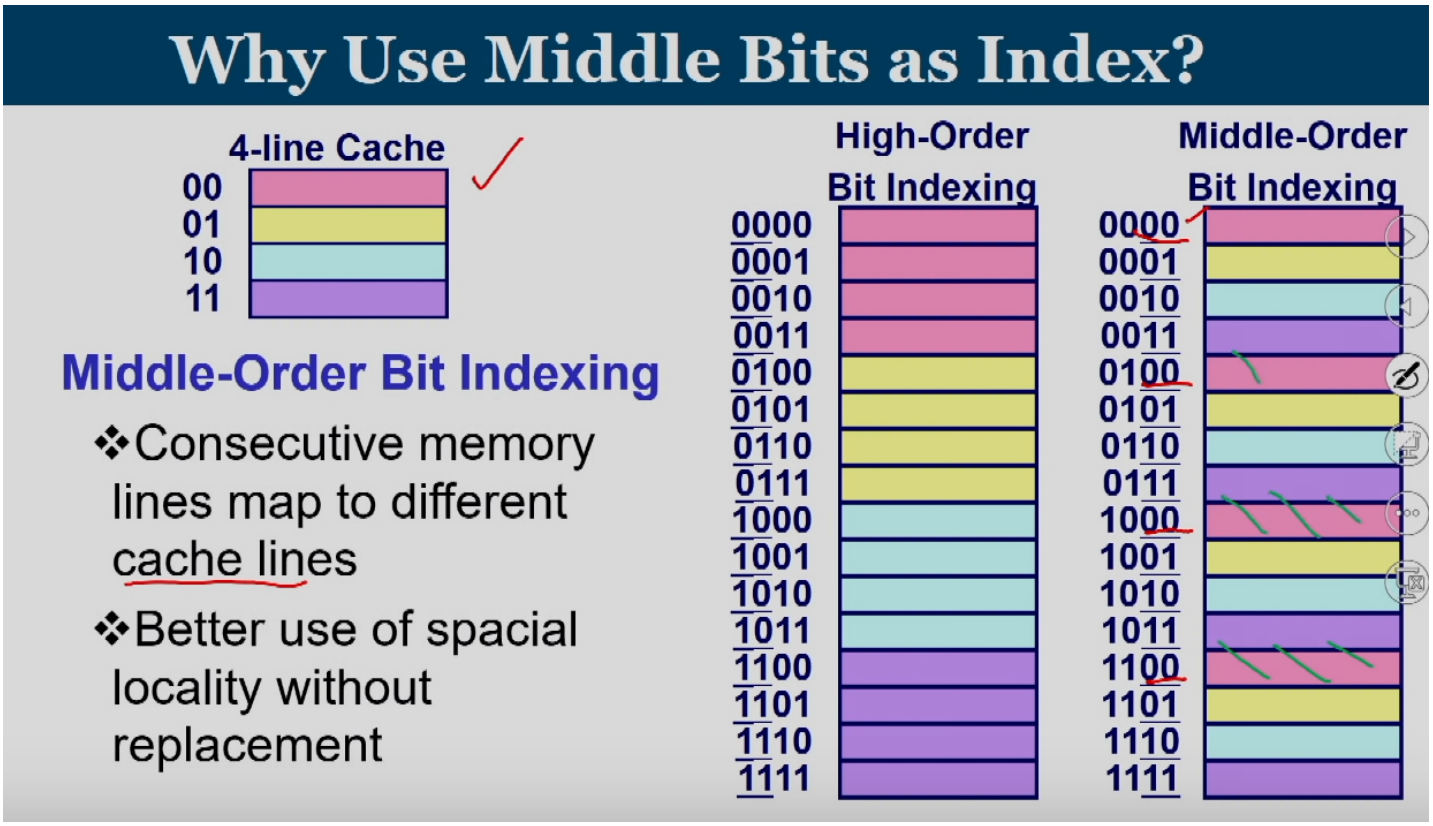
Block can be placed anywhere

Cache Indexing



Why are we using middle bits for indexing ?





Here's why middle-order indexing benefits from spatial locality:

Reduced Conflict Misses:

- With middle-order indexing, the index (which determines the cache block) is based on the middle bits of the address. This allows nearby addresses (frequently accessed due to spatial locality) to map to different cache blocks. This reduces conflict misses, which occur when multiple addresses try to occupy the same cache block, forcing eviction of potentially useful data.

Efficient Handling of Large Objects:

- When accessing a large object spanning multiple cache blocks, middle-order indexing ensures different sections of the object can reside in separate cache blocks. This avoids thrashing, where continuous access to a large object evicts other frequently used data.

High-order bit indexing, on the other hand, wouldn't be as effective:

- **Increased Conflict Misses:** With high-order bits as the index, nearby addresses would likely map to the same cache block. This can lead to frequent conflicts and eviction of useful data due to spatial locality.

In essence, middle-order indexing leverages the knowledge of spatial locality to optimize cache usage and reduce misses. This translates to faster program execution.

Block Replacement

1. Direct-mapped cache, the position that each memory block occupies in the cache is fixed. As a result, the replacement strategy is trivial.
2. Associative and set-associative mapping provide some flexibility in deciding which memory block occupies which cache block.

3. When a new block is to be transferred to the cache, and all the positions it may occupy are full, which block in the cache should be replaced?

When a miss occurs, cache controller must select a block to be replaced with the desired data.

Cache Replacement policies (algorithms)

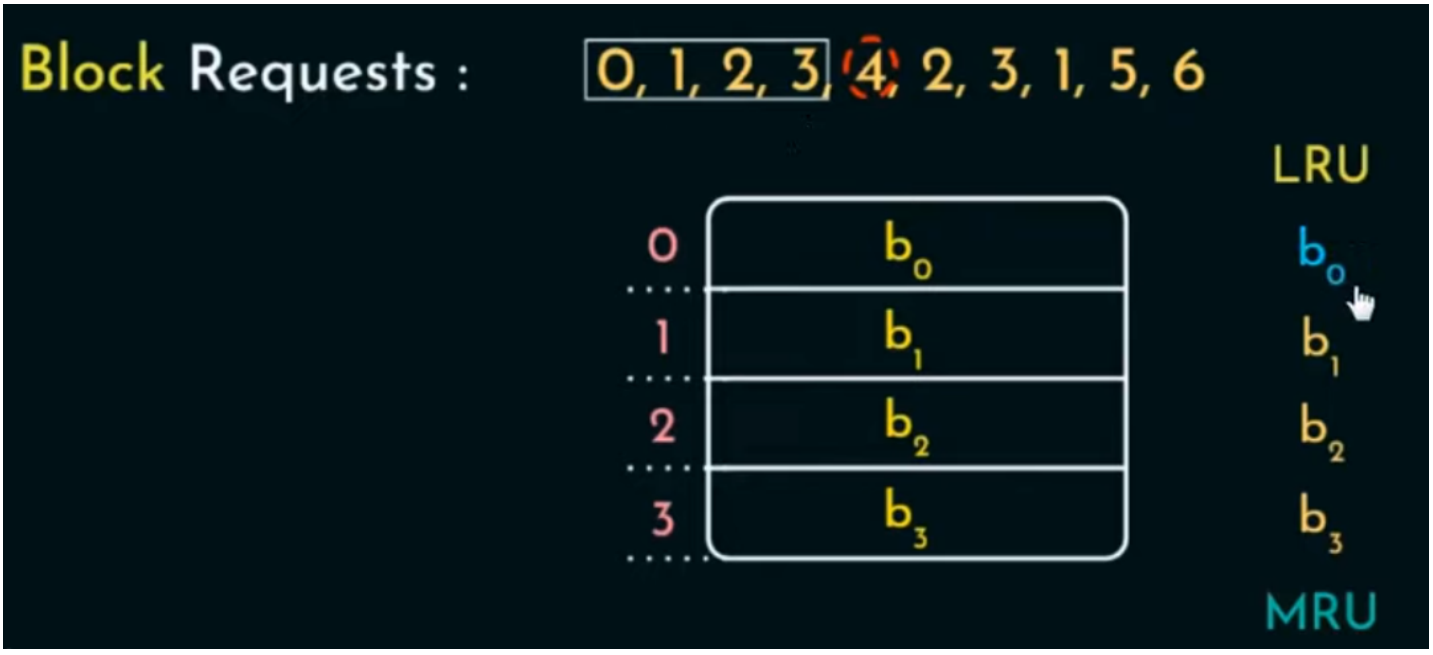
1. LRU : Least Recently Used
2. FIFO : First In First Out
3. LFU : Least Frequently Used
4. Random

Goals of cache replacement policies

> Miss Penalty Reduction

> Hit Rate Increase

LRU



Least Recently Used :

- Exploits *Temporal Locality*.
- Evicts *least recently referred* block.

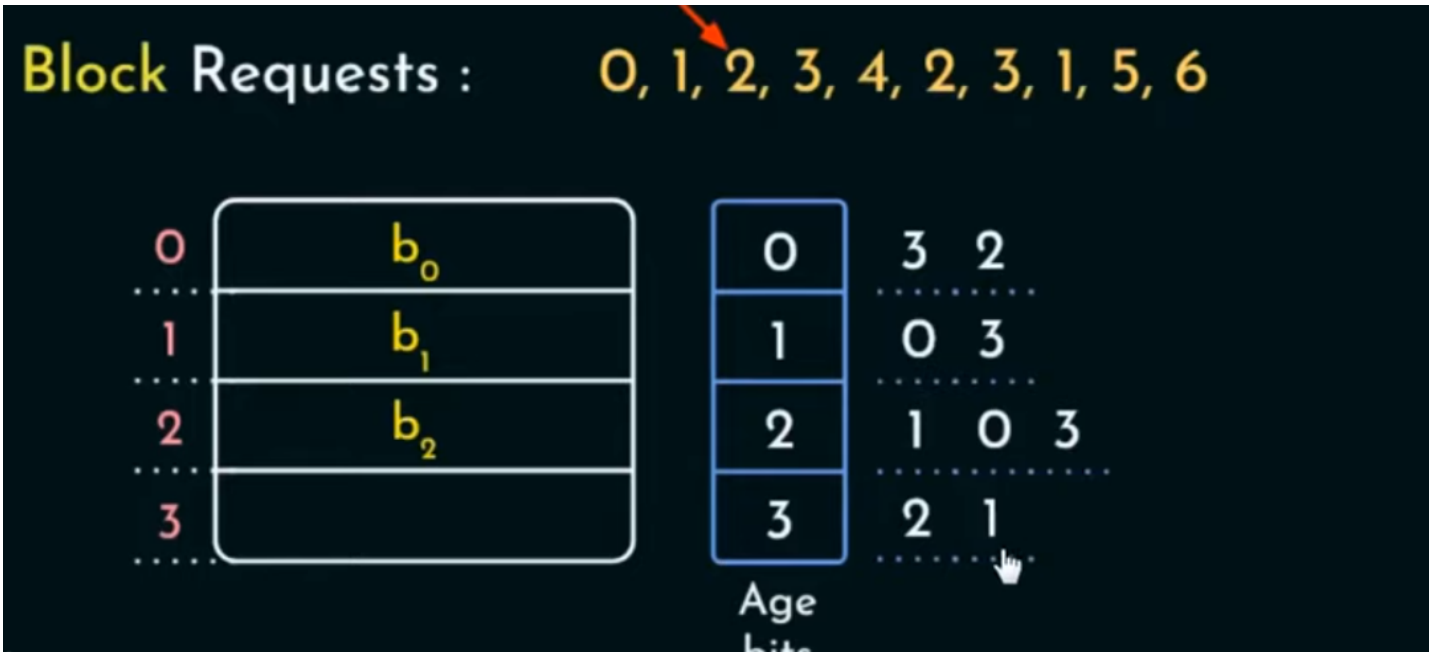
> If misses occur the bits are filled. Once all bits get filled, the least recently referred block is replaced by the block due to which the miss occurred.

> A list with the most and the least recently used blocks is maintained. This list gets updated whenever any hits/misses occur.

> As we can see in the image above, bo is the least recently used block and hence it gets replaced by 4 and accordingly the list gets updated where in b4 is placed at the end which is where the most

> recently used blocks are placed.
> This continues indefinitely.

LRU Implementation



Age bits are used.

once a block is placed, its age bit value is 3. and the Age bit values for the rest of the blocks is decreased by 1.

Note : Initially the Age bits are 0 to n for block number 0 to n. when one of these blocks gets filled, it changes to 3 and the rest of the age bits are decreased by 1. Age bits which are 0 aren't decreased any further they remain 0.

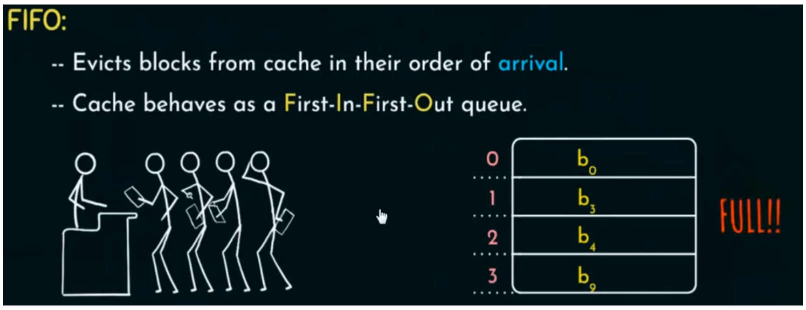
Calculating the number bits required to keep track of the age :

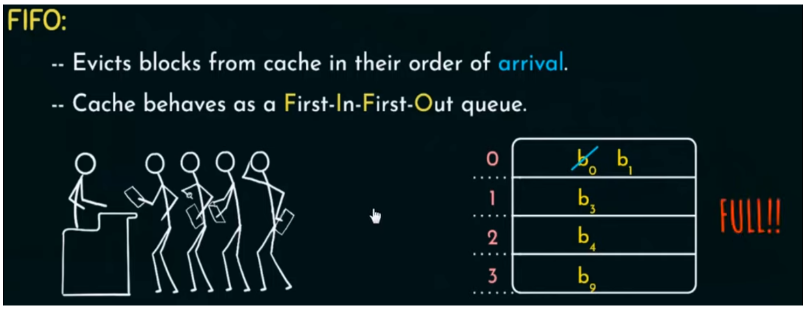
no of possible numbers for the first : 4
.. for the second : 3
.. for the 3rd : 2
.. for the 4th : 1

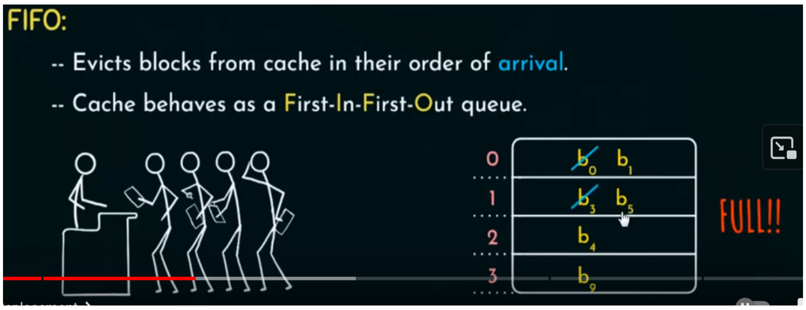
so $4^* 3^*2^*1 = 24$
$\log2(24) =\sim 5$

FIFO

As the name suggests first in first out.

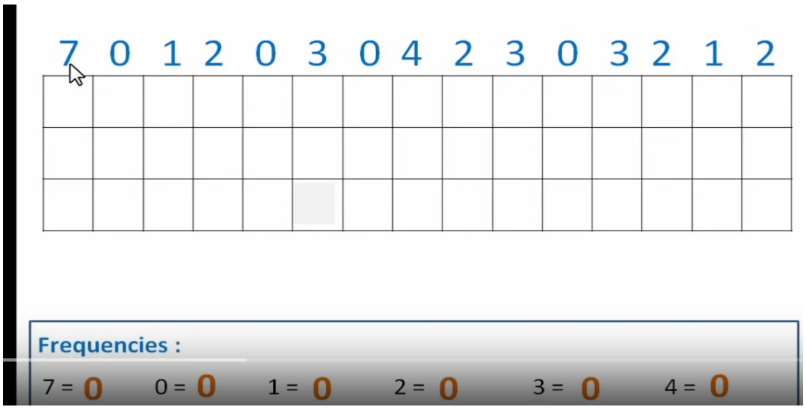




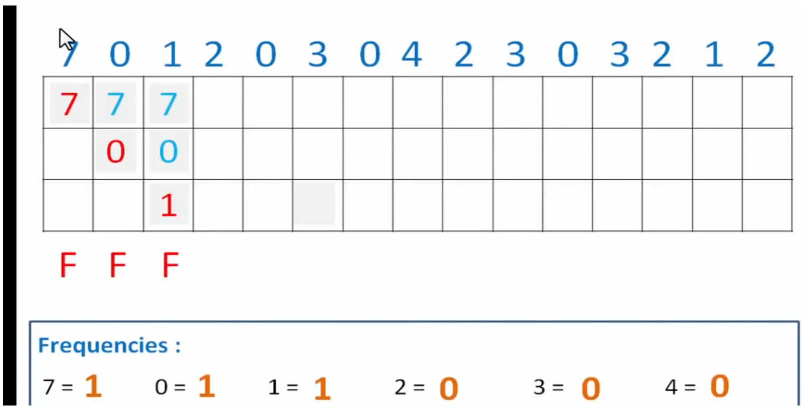


LFU

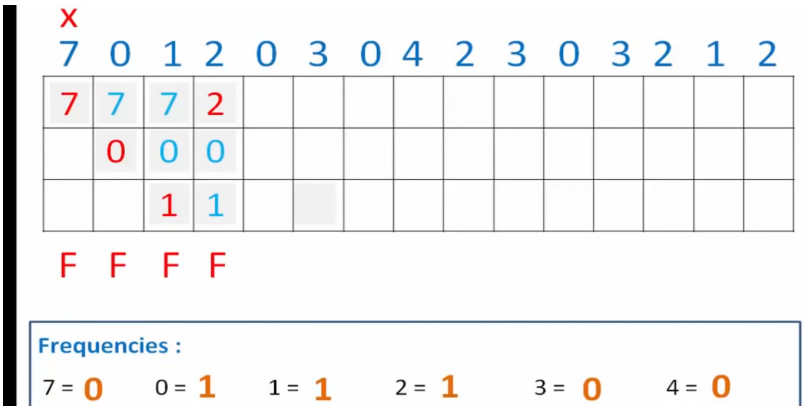
Least Frequently Used

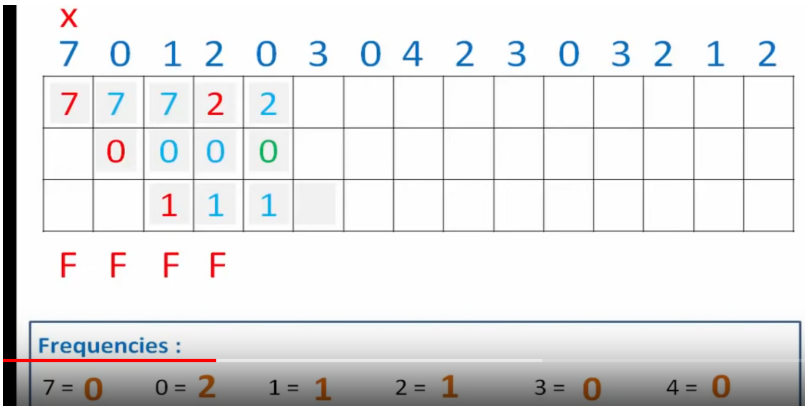


The blocks represent page frame buffer, the numbers on the block are the page requests.



Frequency is the same so FIFO is used. 7 came first so it gets removed.





Examples

Q1]

Exercise 1: Direct Mapping

PES UNIVERSITY

Consider a Direct Mapping cache with 8 cache blocks (numbered 0-7) and the following sequence of memory block requests.
4, 3, 25, 8, 19, 6, 25, 8, 16, 35, 45, 22, 8, 3, 16, 25, 7
Which of the memory blocks will be present in the cache at the end of the sequence?
Also, calculate the hit ratio and miss ratio.

4, 3, 25, 8, 19, 6, 25, 8, 16, 35, 45, 22, 8, 3, 16, 25, 7

8,16 25 6,22 3,19,35 4 45 7

j mod 8

Exercise 1: Direct Mapping

Given 4, 3, 25, 8, 19, 6, 25, 8, 16, 35, 45, 22, 8, 3, 16, 25, 7

8,16 | 25 | 6,22 | 3,19,35 | 4 | 45 | 22 | 7

Color is based on **j mod 8**

Line 0	~~8,16,8~~,16
Line 1	25
Line 2	
Line 3	~~3, 19,35,~~ 3
Line 4	4
Line 5	45
Line 6	~~6,~~22
Line 7	7

Miss= 14 out of 17 access

4	MISS
3	MISS
25	MISS
8	MISS
19	MISS
6	MISS
25	HIT
8	HIT
16	MISS
35	MISS
45	MISS
22	MISS
8	MISS
3	MISS
16	MISS
25	HIT
7	MISS

Q2]

Exercise 2: FIFO-2 Way Set Associative

Consider a 2-way set associative cache with 8 cache blocks (numbered 0-7) and the following sequence of memory block requests:
4, 3, 25, 8, 19, 6, 25, 8, 16, 35, 45, 22, 8, 3, 16, 25, 7
Which of the memory blocks will be present in the cache at the end of the sequence if ***FIFO cache replacement policy is used***? Also, calculate the hit ratio and miss ratio.

Given 4, 3, 25, 8, 19, 6, 25, 8, 16, 35, 45, 22, 8, 3, 16, 25, 7

j mod 4

4,8,16 | 25,45 | 6,22 | 3,7,19,35

Exercise 2: FIFO-2 Way Set Associative

Given 4, 3, 25, 8, 19, 6, 25, 8, 16, 35, 45, 22, 8, 3, 16, 25, 7

Set 0	~~4,16~~
	8
Set 1	25
	45
Set 2	6
	22
Set 3	~~3,35~~,7
	~~19~~,3

Miss= 12 out of 17 access

4,8,16 25,45 6,22 3,7,19,35

4	MISS
3	MISS
25	MISS
8	MISS
19	MISS
6	MISS
25	HIT
8	HIT
16	MISS
35	MISS
45	MISS
22	MISS
8	HIT
3	MISS
16	HIT
25	HIT
7	MISS

Q3]

Exercise 3: FIFO- Fully Associative

Consider a Fully Associative Mapping cache with 8 cache blocks (numbered 0-7) and the following sequence of memory block requests:
4, 3, 25, 8, 19, 6, 25, 8, 16, 35, 45, 22, 8, 3, 16, 25, 7
Which of the memory blocks will be present in the cache at the end of the sequence? Also, calculate the hit ratio and miss ratio.

Given 4, 3, 25, 8, 19, 6, 25, 8, 16, 35, 45, 22, 8, 3, 16, 25, 7

Line 0	
Line 1	
Line 2	
Line 3	
Line 4	
Line 5	
Line 6	
Line 7	

Exercise 3: FIFO- Fully Associative

Given 4, 3, 25, 8, 19, 6, 25, 8, 16, 35, 45, 22, 8, 3, 16, 25, 7

Line 0	~~4, 45~~
Line 1	~~3,~~22
Line 2	~~25,~~3
Line 3	~~8,~~25
Line 4	~~19,~~7
Line 5	6
Line 6	16
Line 7	35

Miss= 13 out of 17 access

4	MISS
3	MISS
25	MISS
8	MISS
19	MISS
6	MISS
25	HIT
8	HIT
16	MISS
35	MISS
45	MISS
22	MISS
8	HIT
3	MISS
16	HIT
25	MISS
7	MISS

Q4]

Exercise 4: LRU-Fully Associative

Consider a Fully Associative Mapping cache with 8 cache blocks (numbered 0-7) and the following sequence of memory block requests:
4, 3, 25, 8, 19, 6, 25, 8, 16, 35, 45, 22, 8, 3, 16, 25, 7
Which of the memory blocks will be present in the cache at the end of the sequence? Also, calcula the hit ratio and miss ratio.

Given 4, 3, 25, 8, 19, 6, 25, 8, 16, 35, 45, 22, 8, 3, 16, 25, 7

Line 0	
Line 1	
Line 2	
Line 3	
Line 4	
Line 5	
Line 6	
Line 7	

Exercise 4: LRU-Fully Associative

Given 4, 3, 25, 8, 19, 6, 25, 8, 16, 35, 45, 22, 8, 3, 16, 25, 7

Line 0	~~4,~~ 45
Line 1	~~3,~~22
Line 2	25
Line 3	8
Line 4	~~19,~~3
Line 5	~~6,~~7
Line 6	16
Line 7	35

Miss= 12 out of 17 access

4	MISS
3	MISS
25	MISS
8	MISS
19	MISS
6	MISS
25	HIT
8	HIT
16	MISS
35	MISS
45	MISS
22	MISS
8	HIT
3	MISS
16	HIT
25	HIT
7	MISS

Q5]

Exercise 5: LRU-4 Way Set Associative

Consider a 4-way set associative mapping with 16 cache blocks. The memory block requests are in the order-
0, 255, 1, 4, 3, 8, 133, 159, 216, 129, 63, 8, 48, 32, 73, 92, 155
If LRU replacement policy is used, which cache block will not be present in the cache? Also, calculate the hit ratio and miss ratio.

Set 0	~~0,~~48
	~~4,~~ 32
	8
	~~216,~~92
Set 1	1
	133
	129
	73

Set 2	
Set 3	~~255,~~ 155
	3
	159
	63

0 % 4 = 0
255 % 4 = 3
1 % 4 = 1
4 % 4 = 0
3 % 4 = 3
8 % 4 = 0
133 % 4 = 1
159 % 4 = 3
216 % 4 = 0
129 % 4 = 1
63 % 4 = 3
8 % 4 = 0
48 % 4 = 0
32 % 4 = 0
73 % 4 = 1
92 % 4 = 0
155 % 4 = 3

0	MISS
255	MISS
1	MISS
4	MISS
3	MISS
8	MISS
133	MISS
159	MISS
216	MISS
129	MISS
63	MISS
8	HIT
48	MISS
32	MISS
73	MISS
92	MISS
155	MISS

Q6]

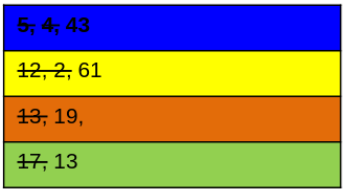
Practice Exercise

Consider the cache has 4 blocks. For the memory references-
5, 12, 13, 17, 4, 12, 13, 17, 2, 13, 19, 13, 43, 61, 19
What is the hit ratio for the following cache replacement algorithms-

i. Fully Associative FIFO
ii. Fully Associative LRU
iii. Direct mapping
iv. 2-way set associative mapping using LRU

Fully Associative FIFO

5, 12, 13, 17, 4, 12, 13, 17, 2, 13, 19, 13, 43, 61, 19



HIT RATIO = 5/15

5	MISS
12	MISS
13	MISS
17	MISS
4	MISS
12	HIT
13	HIT
17	HIT
2	MISS
13	HIT
19	MISS
13	MISS
43	MISS
61	MISS
19	HIT

Fully Associative LRU

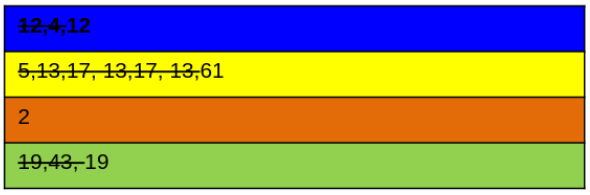
5, 12, 13, 17, 4, 12, 13, 17, 2, 13, 19, 13, 43, 61, 19

HIT RATIO = 6/15

5	MISS
12	MISS
13	MISS
17	MISS
4	MISS
12	HIT
13	HIT
17	HIT
2	MISS
13	HIT
19	MISS
13	HIT
43	MISS
61	MISS
19	HIT

Direct Mapping

5, 12, 13, 17, 4, 12, 13, 17, 2, 13, 19, 13, 43, 61, 19



5%4	1
12%4	0
13%4	1
17%4	1
4%4	0
2%4	2
19%4	3
43%4	3
61%4	1

HIT RATIO = 1/15

5	Miss
12	Miss
13	Miss
17	Miss
4	Miss
12	Miss
13	Miss
17	Miss
2	Miss
13	Miss
19	Miss
13	Hit
43	Miss
61	Miss
19	Miss

Set Associative Mapping with LRU

5, 12, 13, 17, 4, 12, 13, 17, 2, 13, 19, 13, 43, 61, 19

Set 0	12
	~~4~~,2
Set 1	~~5,17,19,43,~~ 19
	~~13,~~ 61

5%2	1
12%2	0
13%2	1
17%2	1
4%2	0
2%2	0
19%2	1
43%2	1
61%2	1

HIT RATIO = 5/15

5	Miss
12	Miss
13	Miss
17	Miss
4	Miss
12	Hit
13	Hit
17	Hit
2	Miss
13	Hit
19	Miss
13	Hit
43	Miss
61	Miss
19	Miss

Write Strategy

Write-through and write-back are two different policies used in cache memory systems to handle write operations to the cache and main memory. Here's an explanation of each:

1. Write through: In the write-through policy, whenever data is written or updated in the cache, the same write operation is also performed on the corresponding main memory location. This ensures that the data in the main memory is always consistent with the data in the cache.

The steps involved in a write through operation are: a) The processor writes the data to the cache. b) The cache controller writes the same data to the corresponding main memory location.

The main advantage of the write-through policy is that it maintains strict data consistency between the cache and main memory. However, it can lead to higher memory traffic and longer write latencies since every write operation needs to go to the relatively slower main memory.

2. Write back (or write-behind): In the write-back policy, when data is written or updated in the cache, the write operation is initially performed only in the cache. The corresponding main memory location is not updated immediately. Instead, the cache keeps track of which cache lines have been modified (using a "dirty" bit or similar mechanism).

The steps involved in a write-back operation are: a) The processor writes the data to the cache. b) The cache controller updates the "dirty" bit or status for that cache line to indicate that it has been modified. c) When the modified cache line needs to be evicted (due to a cache miss or replacement policy), the cache controller writes the updated data back to the main memory.

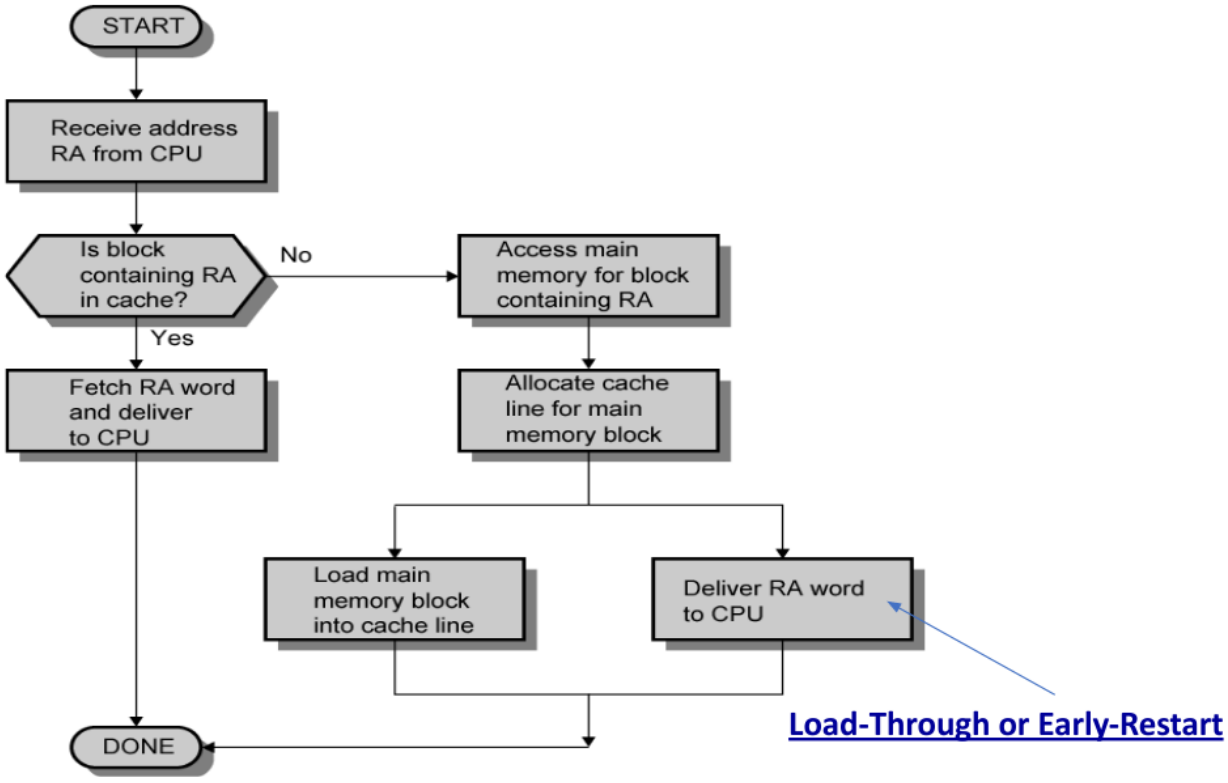
The main advantage of the write-back policy is that it reduces memory traffic and can improve performance by minimizing the number of writes to the relatively slower main memory. However, it

introduces the risk of data inconsistency between the cache and main memory if the system crashes or loses power before the modified data in the cache is written back to main memory.

To mitigate this risk, systems using the write-back policy often employ additional mechanisms, such as periodic cache flushing (writing all modified data back to main memory) or battery-backed caches that can safely preserve the data in case of power loss.

Both write-through and write-back policies have their advantages and disadvantages, and the choice between them depends on the specific requirements of the system, such as performance, data consistency, and power considerations.

Read: Hit or Miss



Load-through or early-restart

- Block of words containing this requested word is transferred from the memory.

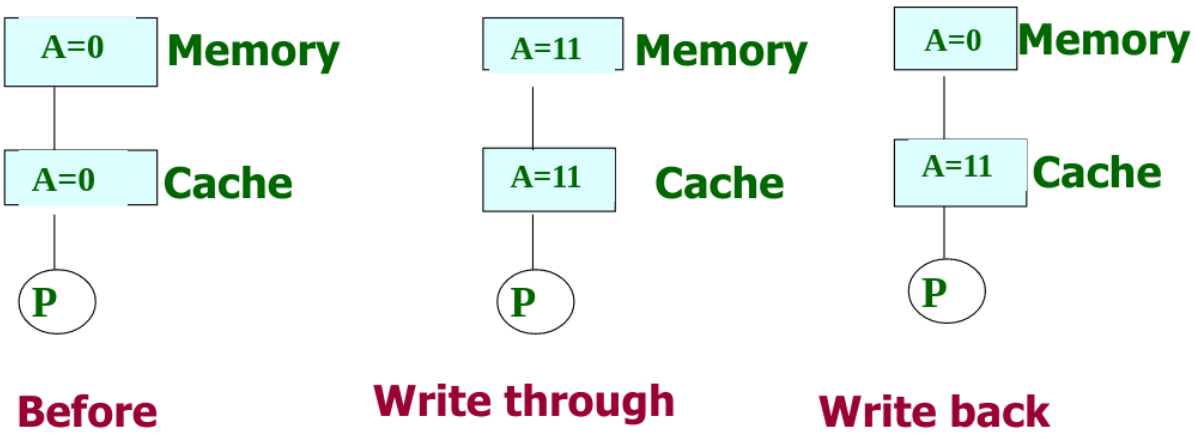
- After the block is transferred, the desired word is forwarded to the processor.

- The desired word may also be forwarded to the processor as soon as it is transferred

  without waiting for the entire block to be transferred.

- This is called **load-through or early-restart.**

Write On Hit



Write Through on Hit

Updated Simultaneously

Valid Bit	TAG	Data
1	10100	A=11
1	10100	B=5
1	10100	C=6

10100…..1010..0000	A= 11
10100…..1010..0001	B=5
10100…..1010..0010	C=6
00010…..1010..0000	P=100

- Avoid Inconsistency.
- Purpose of having Cache is dishonored.
- Some design, buffer the update and continue executing other instruction
- Cache Coherence Problem

Write Back on Hit: With Validation

Before Write A

Valid Bit	Dirty Bit	TAG	Data
1	0	10100	A=0
1	0	10100	B=5
1	0	10100	C=6

- Each Line in the cache has Valid Bit and Dirty Bit
- **Valid Bit** indicates that no other program has changed the Data
- **Dirty Bit** indicates that Data is written or not written

10100…..1010..0000	**A= 0**
10100…..1010..0001	B=5
10100…..1010..0010	C=6
00010…..1010..0000	P=100

Write Back on Miss: With Validation

On Miss : Two Possibilities

1: Bring the Block on the Line **(Write Allocate)**

2: Update the Block in the Memory Directly **(Write No Allocate)**

In a write allocate cache. If a write misses, then the cache subsystem will read the enclosing cache block from memory into the cache, and retry the write.

In a write no allocate cache, if a write hits, the cache will be updated. If the write misses, it will go directly to memory and not be added to the cache.

Generally you want to use write no allocate when the written data will not be read by the CPU in the near future, and you want to use write-allocate when the data will be read again soon.

Write no allocate is generally better when you are filling in IO buffers, because they will likely never be read again by the CPU. Write allocate is generally better when you are, for example, pushing arguments on the stack for a procedure call. The called function will be immediately reading those, so they should be in the cache.

Performance Analysis

To examine the performance of a memory system, the important factors are:

- How long does it take to send data from the cache to the CPU? (Hit Time)
- How long does it take to copy data from memory into the cache? (Miss Penalty)
- How often do we have to access main memory? (Miss Rate)

- CPU performance is measured by
  - **Instruction Count**
  - **Miss Rate**
  - Both independent of hardware
- A better measure of memory hierarchy performance is - **Average memory access time.**

Average memory access time = Hit time + Miss rate x Miss penalty.
where Hit time – Time to hit in the cache.

The components of an average access time can be measured either as
- Absolute time on a hit, say, 0.25 to 1.0 ns.
- Number of clock cycles that the processor waits for the memory, such as 150 to 200 clock cycles.

Note:
- Average memory access time is still an indirect measure of performance., although, better measure than miss rate.
- This formula can help to decide between spilt cache and a unified cache.

We now account for the number of cycles during which the processor is stalled waiting for a memory access, which we call the memory stall cycles. The performance is then the product of the clock cycle time and the sum of the processor cycles and the memory stall cycles:

$$\text{CPU execution time} = (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle time}$$

- Clock cycles for a cache hit should be considered part of
  - CPU execution clock cycles? or
  - Memory stall clock cycles?

- **Most widely accepted, is to include in CPU execution clock cycles.**

Three steps are taken when a cache needs to load data from the main memory.

1: Sending Address to RAM
2: Accessing data from RAM
3: Receiving data from RAM

Example: If the buses from the CPU to the cache and from the cache to RAM are all one word wide. Memory accesses take 15 cycles, If the cache has one-word blocks, How much time required for filling a block from RAM **(*i.e.*, the miss penalty)?**

1. It takes 1 cycle to send an address to the RAM.
2. If there is a 15-cycle latency for each RAM access.
3. It takes 1 cycle to return data from the RAM.

Miss Penalty is: 1 + 15 + 1 = 17 clock cycles

Example2: If the buses from the CPU to the cache and from the cache to RAM are all one word wide. Memory accesses take 15 cycles, If the cache has **four-word blocks**, How much time required for filling a block from RAM **(*i.e.*, the miss penalty)?**

Solution 1: If the cache has four-word blocks, then loading a single block would need four individual main memory accesses, and a miss penalty of 68 cycles!

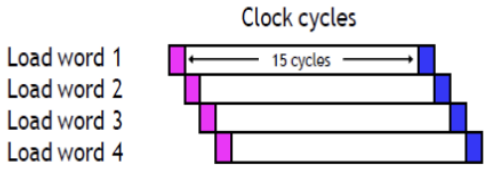
4 x (1 + 15 + 1) = 68 clock cycles

Solution 2: Widen the Bus to read multiple word in single shot.

1 + 15 + 1 = 17 cycles

But it will be Costly!☹

Solution 3: Create individual bank for each word and Overlap the Latencies.




```
1 + 15 + (4 x 1) = 20 cycles
```

- The magenta cycles represent sending an address to a memory bank.
- Each memory bank has a 15-cycle latency, and
- It takes another cycle (shown in blue) to return data from the memory.

This is the same basic idea as pipelining!
—As soon as we request data from one memory bank, we can go ahead and request data from another bank as well.
—Each individual load takes 17 clock cycles, but four overlapped loads require just 20 cycles.

AMAT = Hit Time + (Miss Rate x Miss Penalty)

- This is just averaging the amount of time for cache hits and the amount of time for cache misses.
- Lower AMAT is better
- Reduce Miss Penalty or Miss Rate

	Cache #1	Cache #2
Block size	32-bytes	64-bytes
Miss rate	5%	4%

Which cache configuration would be better?
Assume both caches have single cycle hit times.
Memory accesses take 15 cycles, and the memory bus is 8-bytes wide

Cache #1 : 32-byte memory access takes 20 cycles:
1 (send address) + 15 (memory access) + 4 (four 8-byte transfers)
Miss Penalty=1+15+32B/8B=20cycles

AMAT=1+0.05x20=2

Cache #2 : 64-byte memory access takes 24 cycles:
1 (send address) + 15 (memory access) + 8 (four 8-byte transfers)
Miss Penalty = 1 + 15 + 64B/8B = 24 cycles

AMAT = 1 + 0.04 x 24 = 1.9

We Know that

CPU Execution Time = IC x Overall CPI x Cycle time

Overall CPI = (Base CPI + CPU Stalls)

How does cache hits and misses affect system performance?

Overall CPI = (Base CPI + CPU Stalls + Memory Stalls)

Memory Stall Cycles = Memory Accesses x Miss Rate x Miss Penalty

Assume that 33% of the instructions in a program are data accesses. The cache hit ratio is 97% and the hit time is one cycle, but the miss penalty is 20 cycles.

Solution

Memory Stall Cycles = Memory Accesses x Miss Rate x Miss Penalty

=(0.33) x 0.03 x 20 cycles
= 0.2cycles

CPI = (Base CPI + CPU Stalls + Memory Stalls)

CPI = [1 + 0 + 0.2] = 1.2

CPU Execution Time = IC x 1.2 x cycle Time

This code is 1.2 times slower than a program with a “perfect” CPI of 1!

What if we could *double* the CPU performance so the CPI becomes 0.5, but memory performance remained the same?

CPI= 0.5+0.2= 0.7
CPU time = IC x (0.7) x Cycle time

The overall CPU time improves by just 1.2/0.7 = 1.71 times ☹

Assume we have a computer where the cycles per instruction (CPI) is 1.0 when all memory accesses hit in the cache. The cache is a unified (data + instruction). The only data accesses are loads and stores, and these total 50% of the instructions. If the miss penalty is 25 clock cycles and the miss rate is 2%, how much faster would the computer be if all instructions were cache hits?

SOLUTION:
First compute the performance for the computer that always hits:

CPU execution time = IC × (CPI + Memory stall cycles) × Clock cycle
= IC × (CPI + 0) × Clock cycle
= IC × 1.0 × Clock cycle

Now for the computer with the real cache, first we compute memory stall cycles:

CPU Execution Time = IC x (Base CPI + CPU Stalls + Memory Stalls) x Cycle time

CPU Execution Time = IC x (Base CPI + CPU Stalls + Memory Stalls) x Cycle time

CPU Execution Time = IC x (1 + 0 + Memory Stalls) x Cycle time

Memory Stall Cycles = Memory Accesses x Miss Rate x Miss Penalty
= (1+0.5) x .02 x 25
= 0.75

// Where memory access (Instruction Fetch + Data Access)

CPU Execution Time = IC x (1 + 0 + 0.75) x Cycle time
= IC x (1.75) x Cycle time

The performance ratio is the inverse of the execution times:

$$Speedup = \frac{\text{CPU Execution Time } with\ real\ cache}{\text{CPU Execution Time } with\ all\ hits} = \frac{1.75}{1} = 1.75$$

Hence, The computer with no cache misses is 1.75 times faster.

Given data

- I-cache miss rate = 2%
- D-cache miss rate = 4%
- Miss penalty = 100 cycles
- Base CPI (ideal cache) = 2
- Load & stores are 36% of instructions

Miss cycles per instruction

- I-cache: $0.02 \times 100 = 2$
- D-cache: $0.36 \times 0.04 \times 100 = 1.44$

Actual CPI = 2 + 2 + 1.44 = 5.44

Speedup = IC X 5.44 X Clock Cycle/ IC X 2 X Clock Cycle

Conclusion: Ideal CPU is 5.44/2 =2.72 times faster

Consider a pipelined processor that has an average CPI of 1.8 without accounting for memory stalls. I-Cache has a hit rate of 95% and the D-Cache has a hit rate of 98%. Assume that memory reference instructions account for 30% of all the instructions executed. Out of these 80% are loads and 20% are stores. On average, the read-miss penalty is 20 cycles and the write-miss penalty is 5 cycles. Compute the effective CPI of the processor accounting for the memory stalls.

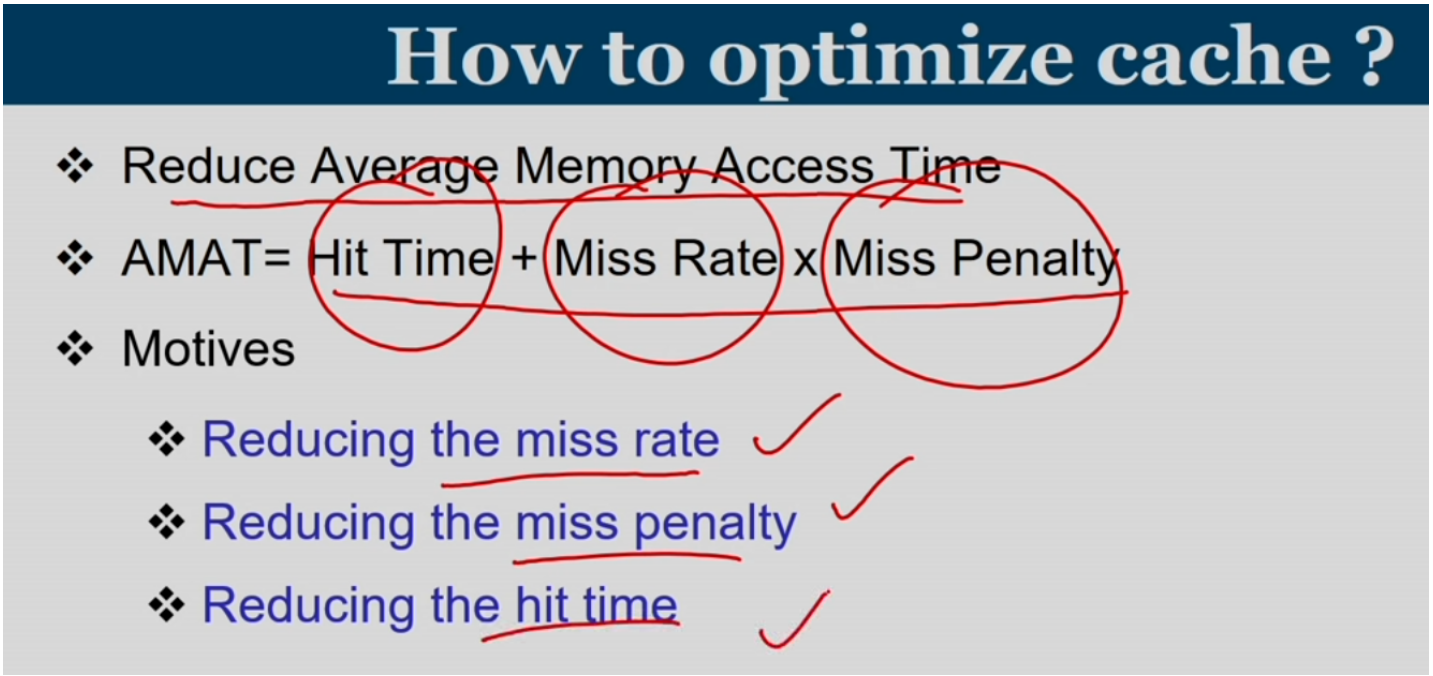
Note:
Avg CPI = 1.8;
I-cache miss ratio = (1-0.95) = **0.05**;
D-cache miss ratio = (1-0.98) = 0.02; 30% mem references (80% loads & 20% stores)
Read miss penalty = 20 cycles; Write miss penalty = 5 cycles;

Cost of instruction misses = cache miss rate * read miss penalty
= 1***0.05*** * 20
= 1 cycle per instruction

- Cost of data read misses = fraction of memory reference instructions in program * fraction of memory reference instructions that are **loads** * **D-cache miss rate** * **Read miss penalty**
= 0.3 * 0.8 * 0.02 * 20
= 0.096 cycles per instruction
- Cost of data write misses = fraction of memory reference instructions in the program * fraction of memory reference instructions that are **stores** * **D-cache miss rate** * **Write miss penalty**
= 0.3 * 0.2 * 0.02 * 5
= 0.006 cycles per instruction

Effective CPI = Avg CPI + Effect of I-Cache on CPI + Effect of D-Cache on CPI
= 1.8 + 1 + 0.096 + 0.006
= **2.902**

Cache Optimization



These three parameters are not independent

CACHE OPTIMIZATIONS

The classical approach to improve cache behaviour is to reduce miss rates.

Three categories misses are:

- **Compulsory :**
  - The very first access to block cannot be in the cache. So, the block must be brought into the cache.
  - These are called cold-start misses or first reference misses.

- **Capacity:**
  - If the cache cannot contain all the blocks needed during execution of a program, capacity misses [in addition to compulsory misses] will occur because of blocks being discarded and later retrieved.

Three categories misses are:

- **Conflict:**

  - If the block placement strategy is set associative or direct mapped, conflict misses [in addition to compulsory and capacity misses] will occur because a block may be discarded and later retrieved if too many blocks map to its set. These misses are also called collision misses.

  - The idea is that hits in a fully associative cache that become misses in an n-way set associative cache, due to more than n requests on some popular sets.

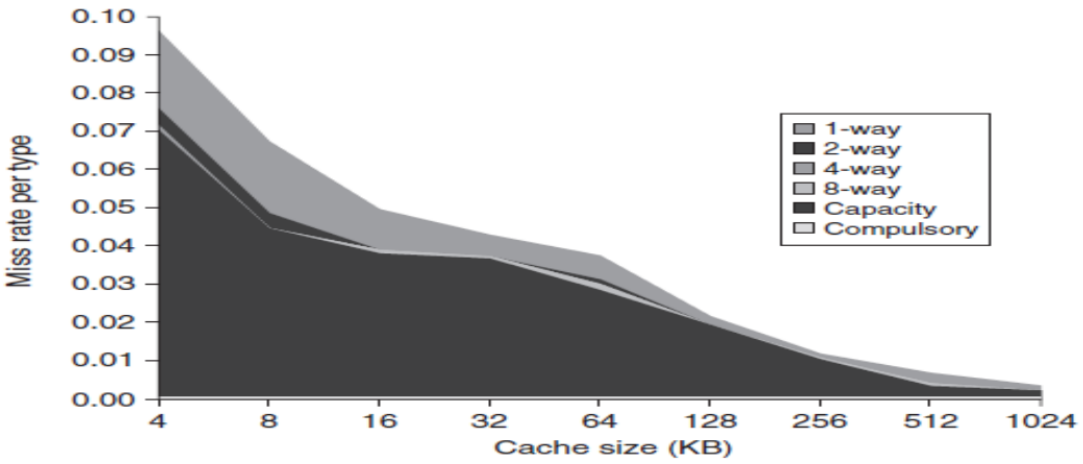
NOTE: Further adding a 4th C , for coherence misses due to flushes to keep multiple caches coherent in a multiprocessor environment.

Cache Optimizations

- **Compulsory Misses :**
  - Those that occur in an infinite cache.
- **Capacity Misses:**
  - Those that occur in a fully associative cache.
- **Conflict Misses:**
  - Those that occur going from fully associative to eight-way associative, four–way associative , and so on…

To show the benefit of associativity, conflict misses are divided into misses caused by each decrease in associativity.
Here are the four divisions of conflict misses and how they are calculated:

- ***Eight-way***—Conflict misses due to going from fully associative to eight-way associative
- ***Four-way***—Conflict misses due to going from eight-way associative to four way associative
- ***Two-way***—Conflict misses due to going from four-way associative to two way associative
- ***One-way***—Conflict misses due to going from two-way associative to one way associative

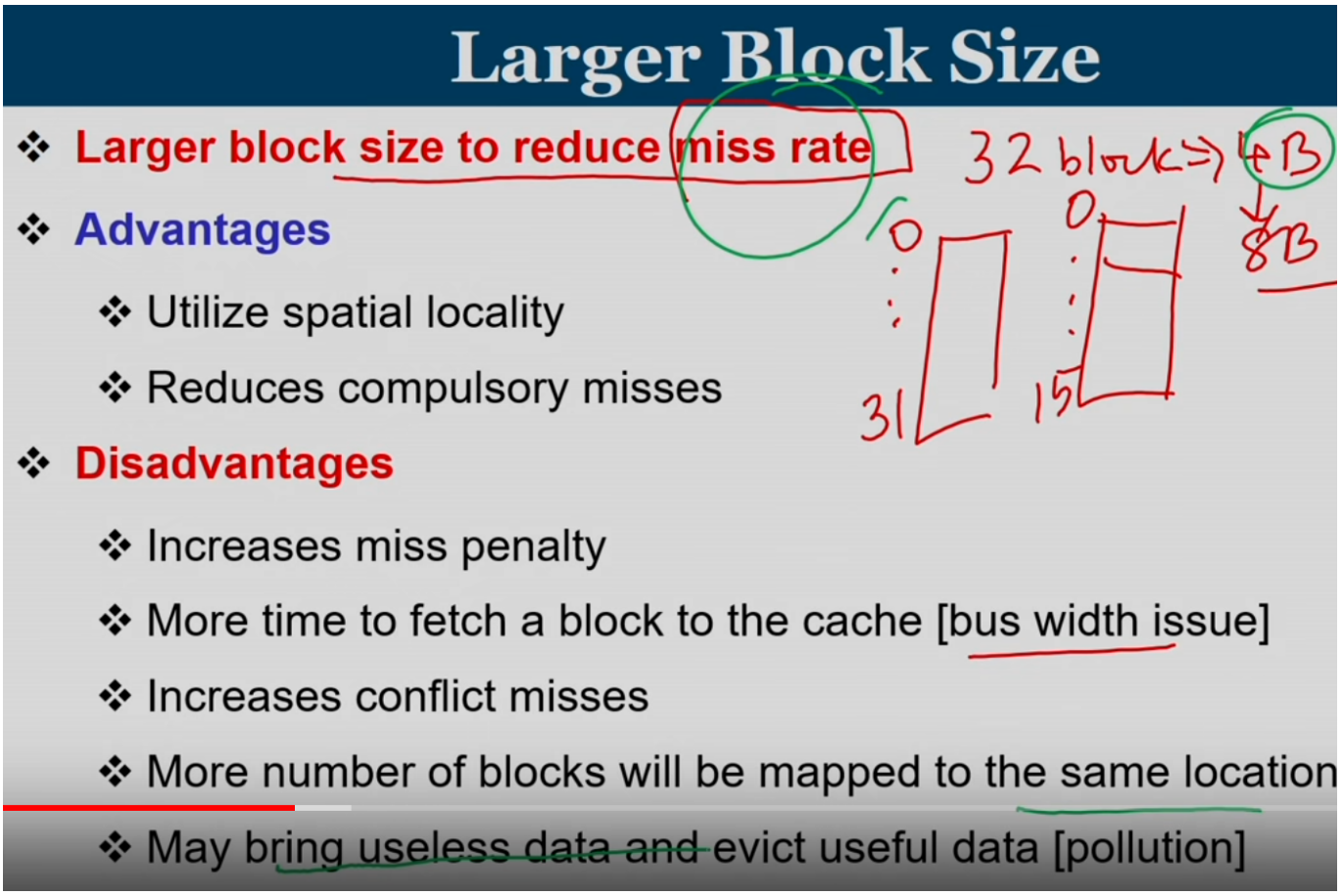


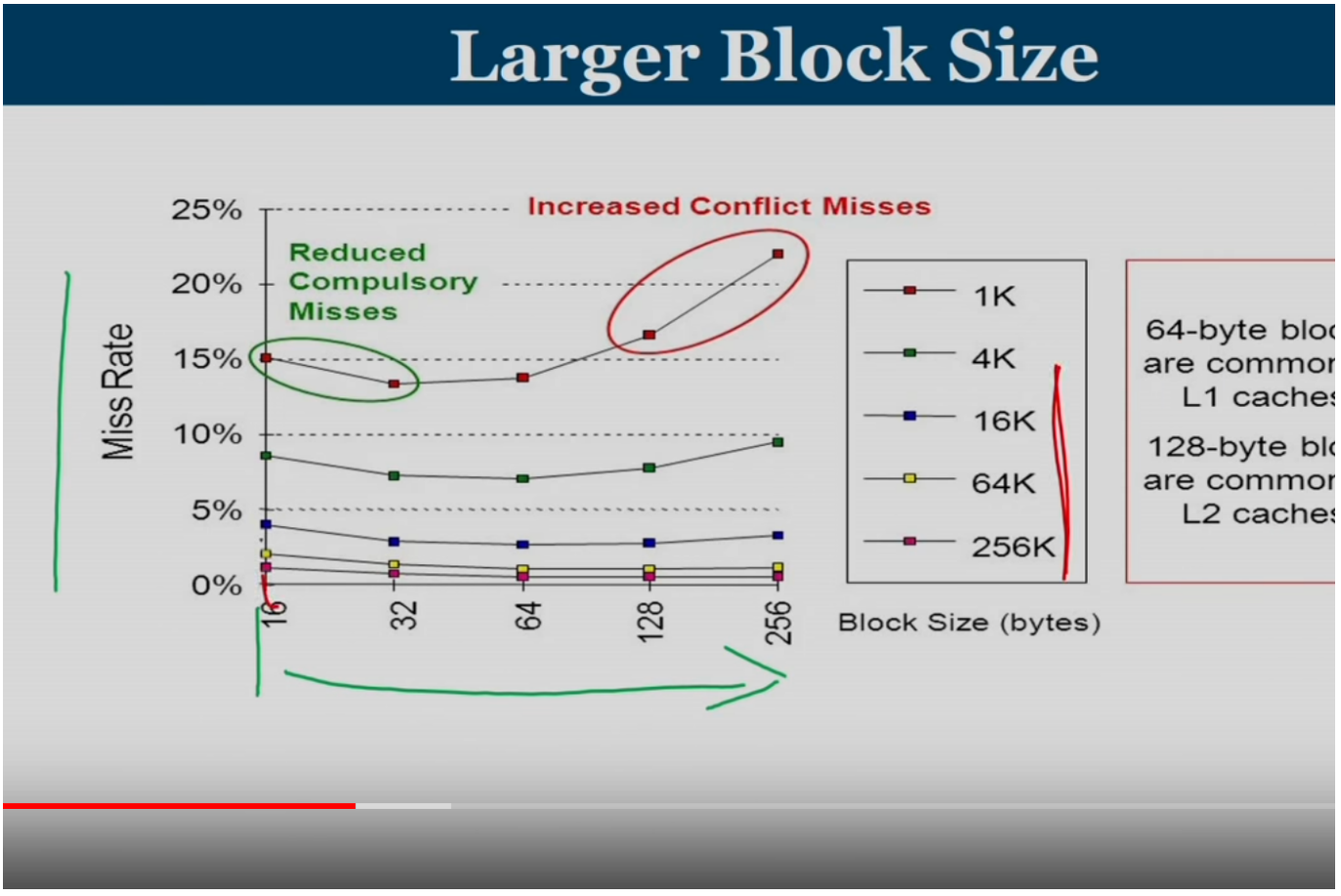
Total miss rate for each size cache according to the three C's for the data diagram shows the actual data cache miss rates,

- **First Optimization:** Larger Block Size to Reduce Miss Rate.
- **Second Optimization:** Larger Caches to Reduce Miss Rate.
- **Third Optimization:** Higher Associativity to Reduce Miss Rate.
- **Fourth Optimization :** Multilevel Caches to Reduce Miss Penalty.
- **Fifth Optimization:** Giving Priority to Read Misses over Writes to Reduce Miss Penalty.
- **Sixth Optimization:** Avoiding Address Translation during Indexing of the Cache to Reduce Hit Time.

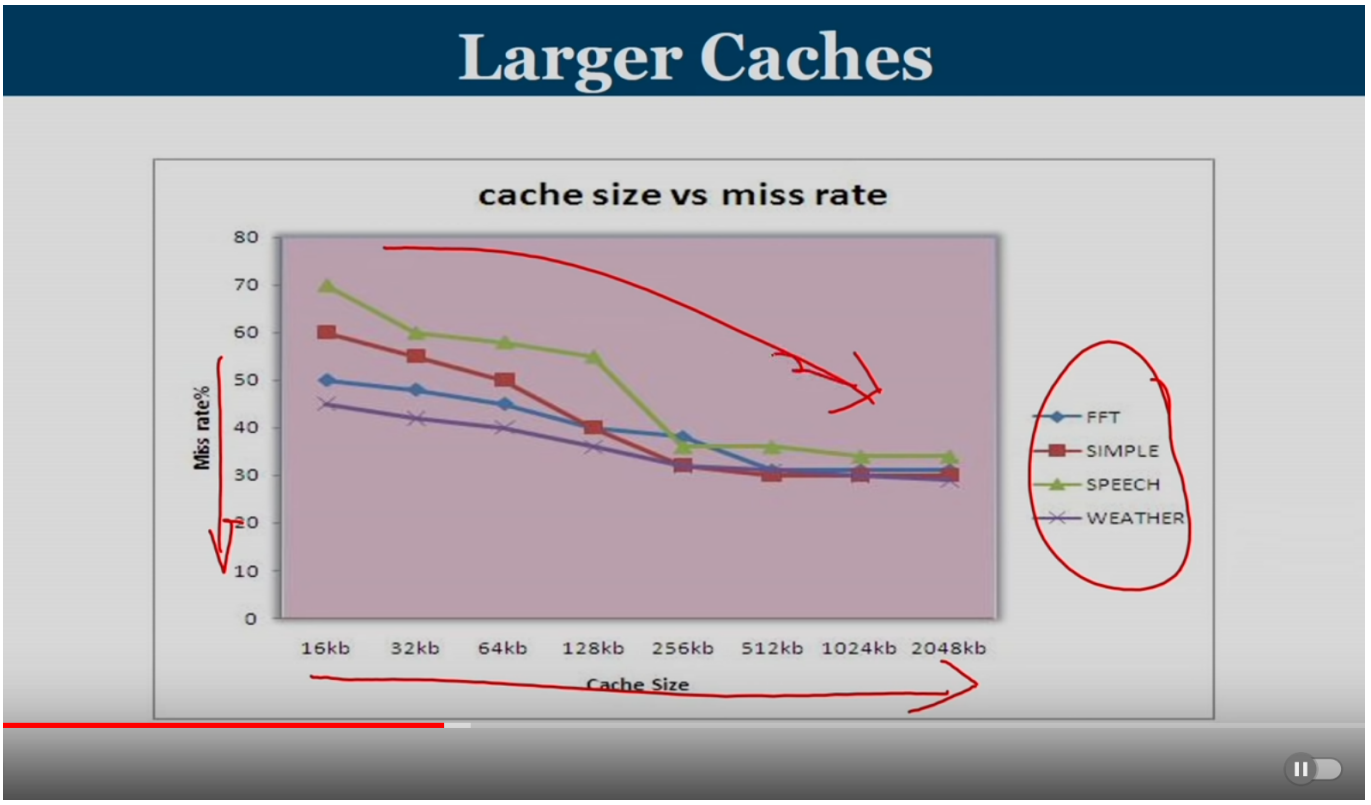
Six basic cache optimizations can be organized into three categories.

- **Reducing the miss rate**
  - Larger block size, Larger cache size, and higher associativity.
- **Reducing the miss penalty**
  - Multilevel caches and giving reads priority over the writes.
- **Reducing the time to hit in the cache**
  - Avoiding address translation when indexing the cache.






```
![[Pasted image 20240418100928.png]]
```



Optimization 1 & 2: Reduce Miss Rate

Block Size vs Cache Size

Block size	Cache size			
	4K	16K	64K	256K
16	8.57%	3.94%	2.04%	1.09%
32	7.24%	2.87%	1.35%	0.70%
64	7.00%	2.64%	1.06%	0.51%
128	7.78%	2.77%	1.02%	0.49%
256	9.51%	3.29%	1.15%	0.49%

Observation1:
Miss Rate Increases if Cache Size is small and Block size is Large

4 K Cache with 256 Block size has highest Miss Rate

Optimization 1 & 2: Reduce Miss Rate

Block Size vs Cache Size

Block size	Cache size			
	4K	16K	64K	256K
16	8.57%	3.94%	2.04%	1.09%
32	7.24%	2.87%	1.35%	0.70%
64	7.00%	2.64%	1.06%	0.51%
128	7.78%	2.77%	1.02%	0.49%
256	9.51%	3.29%	1.15%	0.49%

Observation2:
Miss Rate Decreases if Cache Size is Large and Block size is Large

256 k Cache with 256 Block Size has less Miss Rate

Block Size vs Cache Size

Block size	Cache size			
	4K	16K	64K	256K
16	8.57%	3.94%	2.04%	1.09%
32	7.24%	2.87%	1.35%	0.70%
64	7.00%	2.64%	1.06%	0.51%
128	7.78%	2.77%	1.02%	0.49%
256	9.51%	3.29%	1.15%	0.49%

Observation 3:
Miss Rate Decreases if Cache Size is Large and Block size is Small

Miss Rate Decreases if Cache Size is Large and Block Size is Large

However, If Cache Size if large, Hit Time, Space, Power and Cost Increases.

Higher Associativity to Reduce Miss Rate

- Fully Associative cache are best, as no replacement is required till cache is full.
- Set Associative cache balances the Search Time and Replacement.

Advantage:

- Reduce Conflict Miss
- Reduce Miss Rate and Eviction rate.

Disadvantage:

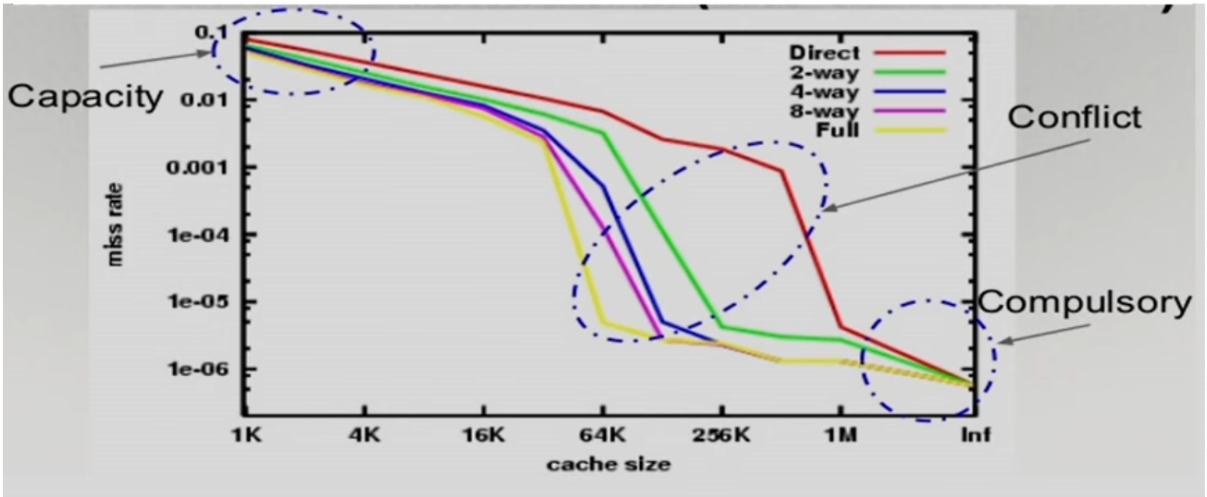
- Increases Hit Time (Due to increase in search time by comparing each TAG)
- Complex design than Direct Map.

Replacement

- 2-way is better than Direct Mapping
- 4-way is better than 2-way
- 8-way is better than 4-way,

Search time

8-way, 16-way may become closer to Fully Associative.



Replacement

- 2-way is better than Direct Mapping
- 4-way is better than 2-way
- 8-way is better than 4-way,

Search time

8-way, 16-way may become closer to Fully Associative.

- For practical purposes (i.e., in real-world scenarios), an eight-way set associative cache is almost as effective as a fully associative cache in reducing cache misses for these sized caches.
- Cache rule of thumb, is that a direct mapped cache of size N has about the same miss rate as a two-way set associative cache of size N/2.
- Improving one aspect of the average memory access time comes at the expense of another. Increasing block size reduces miss rate while increasing miss penalty, and greater associativity can come at the cost of increased hit time.

Example Assume that higher associativity would increase the clock cycle time as listed below:

$$\text{Clock cycle time}_{2\text{-way}} = 1.36 \times \text{Clock cycle time}_{1\text{-way}}$$
$$\text{Clock cycle time}_{4\text{-way}} = 1.44 \times \text{Clock cycle time}_{1\text{-way}}$$
$$\text{Clock cycle time}_{8\text{-way}} = 1.52 \times \text{Clock cycle time}_{1\text{-way}}$$

Assume that the hit time is 1 clock cycle, that the miss penalty for the direct-mapped case is 25 clock cycles to a level 2 cache (see next subsection) that never misses, and that the miss penalty need not be rounded to an integral number of clock cycles. Using Figure B.8 for miss rates, for which cache sizes are each of these three statements true?

$$\text{Average memory access time}_{8\text{-way}} < \text{Average memory access time}_{4\text{-way}}$$
$$\text{Average memory access time}_{4\text{-way}} < \text{Average memory access time}_{2\text{-way}}$$
$$\text{Average memory access time}_{2\text{-way}} < \text{Average memory access time}_{1\text{-way}}$$

	Associativity			
Cache size (KB)	1-way	2-way	4-way	8-way
4	3.44	3.25	3.22	**3.28**
8	2.69	2.58	2.55	**2.62**
16	2.23	**2.40**	**2.46**	**2.53**
32	2.06	**2.30**	**2.37**	**2.45**
64	1.92	**2.14**	**2.18**	**2.25**
128	1.52	**1.84**	**1.92**	**2.00**
256	1.32	**1.66**	**1.74**	**1.82**
512	1.20	**1.55**	**1.59**	**1.66**

Figure B.13 Average memory access time using miss rates in Figure B.8 for parameters in the example. Boldface type means that this time is higher than the number to the left, that is, higher associativity *increases* average memory access time.

Answer Average memory access time for each associativity is

$$\begin{aligned}
\text{Average memory access time}_{8\text{-way}} &= \text{Hit time}_{8\text{-way}} + \text{Miss rate}_{8\text{-way}} \times \text{Miss penalty}_{8\text{-way}} \\
&= 1.52 + \text{Miss rate}_{8\text{-way}} \times 25 \\
\text{Average memory access time}_{4\text{-way}} &= 1.44 + \text{Miss rate}_{4\text{-way}} \times 25 \\
\text{Average memory access time}_{2\text{-way}} &= 1.36 + \text{Miss rate}_{2\text{-way}} \times 25 \\
\text{Average memory access time}_{1\text{-way}} &= 1.00 + \text{Miss rate}_{1\text{-way}} \times 25
\end{aligned}$$

The miss penalty is the same time in each case, so we leave it as 25 clock cycles. For example, the average memory access time for a 4 KB direct-mapped cache is

$$\text{Average memory access time}_{1\text{-way}} = 1.00 + (0.098 \times 25) = 3.44$$

and the time for a 512 KB, eight-way set associative cache is

$$\text{Average memory access time}_{8\text{-way}} = 1.52 + (0.006 \times 25) = 1.66$$

Using these formulas and the miss rates from Figure B.8, Figure B.13 shows the average memory access time for each cache and associativity. The figure shows that the formulas in this example hold for caches less than or equal to 8 KB for up to four-way associativity. Starting with 16 KB, the greater hit time of larger associativity outweighs the time saved due to the reduction in misses.

Note that we did not account for the slower clock rate on the rest of the program in this example, thereby understating the advantage of direct-mapped cache.

Observing the cache performance formula,

Avg. memory access time = Hit time + Miss rate x Miss penalty,

Improvements in Miss penalty is as advantageous as improvements in miss rate.

The performance gap between processor & memory raises a question:
Should I make the cache faster to keep the pace with the speed of the processor?
Or
Make the cache larger to overcome the widening gap between the processor & the main memory?

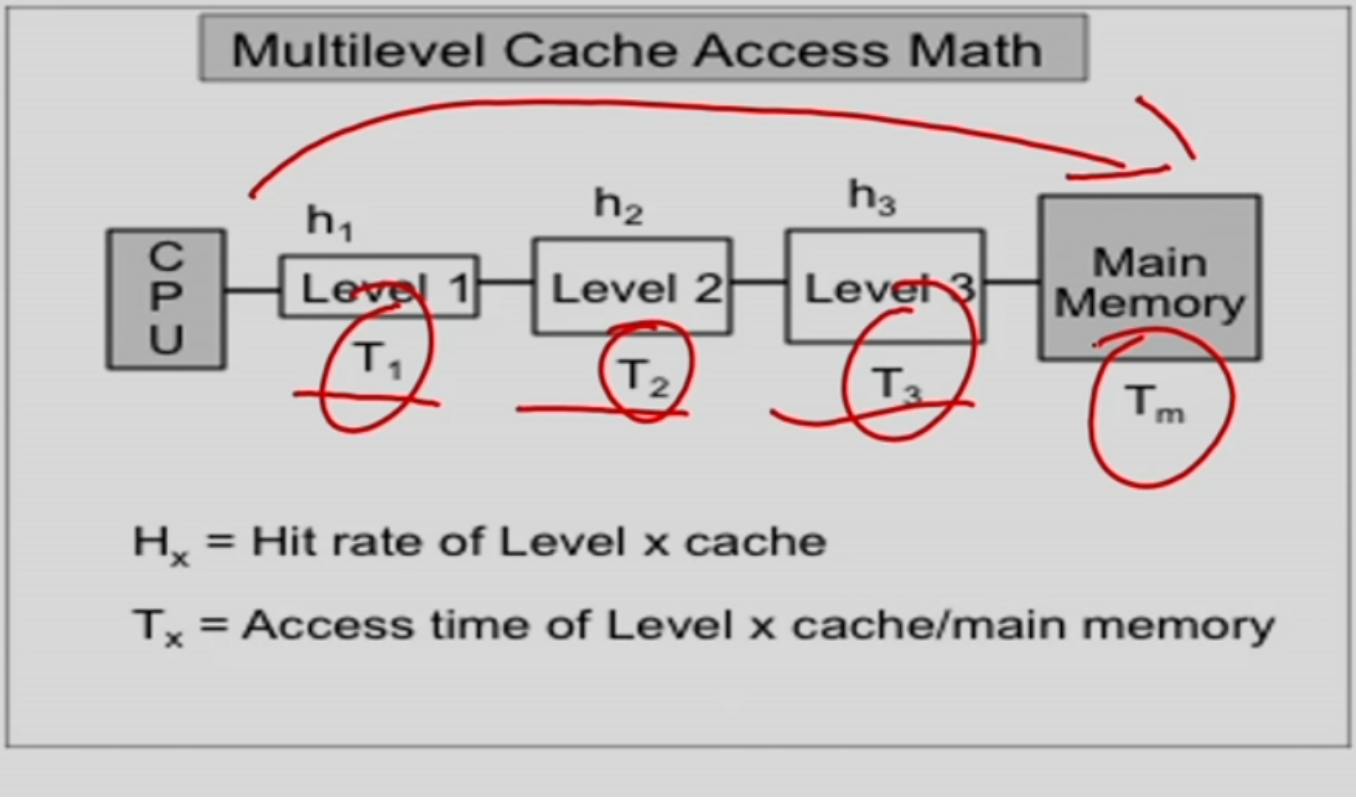
Answer for these questions is to do both.

Adding another level of cache between memory & original cache simplifies the decision.

Multilevel caches

- **Multilevel caches to reduce miss penalty**
- Performance gap between processors and memory
- **Caches should be faster** to keep pace with the speed of processors, **AND cache should be larger** to overcome the widening gap between the processor and main memory
- Add another level of cache between the cache and memory.
- The first-level cache (L1) can be small enough to match the clock cycle time of the fast processor. [Low hit time]
- The second-level cache (L2) can be large enough to capture many accesses that would go to main memory, thereby lessening the effective miss penalty. [Low miss rate]

Multilevel caches



$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times \text{Miss penalty}_{L1}$$

$$\text{Miss penalty}_{L1} = \text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2}$$

$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2})$$

In this formula, the second-level miss rate is measured on the leftovers from the first-level cache. To avoid ambiguity, these terms are adopted here for a two-level cache system:

- *Local miss rate*—This rate is simply the number of misses in a cache divided by the total number of memory accesses to this cache. As you would expect, for the first-level cache it is equal to Miss rate_{L1}, and for the second-level cache it is Miss rate_{L2}.
- *Global miss rate*—The number of misses in the cache divided by the total number of memory accesses generated by the processor. Using the terms above, the global miss rate for the first-level cache is still just Miss rate_{L1}, but for the second-level cache it is Miss $\text{rate}_{L1} \times$ Miss rate_{L2}.

$$\text{Average memory stalls per instruction} = \text{Misses per instruction}_{L1} \times \text{Hit time}_{L2} + \text{Misses per instruction}_{L2} \times \text{Miss penalty}_{L2}$$

Example Suppose that in 1000 memory references there are 40 misses in the first-level cache and 20 misses in the second-level cache. What are the various miss rates? Assume the miss penalty from the L2 cache to memory is 200 clock cycles, the hit time of the L2 cache is 10 clock cycles, the hit time of L1 is 1 clock cycle, and there are 1.5 memory references per instruction. What is the average memory access time and average stall cycles per instruction? Ignore the impact of writes.

Answer The miss rate (either local or global) for the first-level cache is 40/1000 or 4%. The local miss rate for the second-level cache is 20/40 or 50%. The global miss rate of the second-level cache is 20/1000 or 2%. Then

$$\begin{aligned}\text{Average memory access time} &= \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2}) \\ &= 1 + 4\% \times (10 + 50\% \times 200) = 1 + 4\% \times 110 = 5.4 \text{ clock cycles}\end{aligned}$$

To see how many misses we get per instruction, we divide 1000 memory references by 1.5 memory references per instruction, which yields 667 instructions. Thus, we need to multiply the misses by 1.5 to get the number of misses per 1000 instructions. We have 40×1.5 or 60 L1 misses, and 20×1.5 or 30 L2 misses, per 1000 instructions. For average memory stalls per instruction, assuming the misses are distributed uniformly between instructions and data:

$$\begin{aligned}\text{Average memory stalls per instruction} &= \text{Misses per instruction}_{L1} \times \text{Hit time}_{L2} + \text{Misses per instruction}_{L2} \\ &\quad \times \text{Miss penalty}_{L2} \\ &= (60/1000) \times 10 + (30/1000) \times 200 \\ &= 0.060 \times 10 + 0.030 \times 200 = 6.6 \text{ clock cycles}\end{aligned}$$

If we subtract the L1 hit time from the average memory access time (AMAT) and then multiply by the average number of memory references per instruction, we get the same average memory stalls per instruction:

$$(5.4 - 1.0) \times 1.5 = 4.4 \times 1.5 = 6.6 \text{ clock cycles}$$

As this example shows, there may be less confusion with multilevel caches when calculating using misses per instruction versus miss rates.

Note that these formulas are for combined reads and writes, assuming a write-back first-level cache. Obviously, a write-through first-level cache will send *all* writes to the second level, not just the misses, and a write buffer might be used.

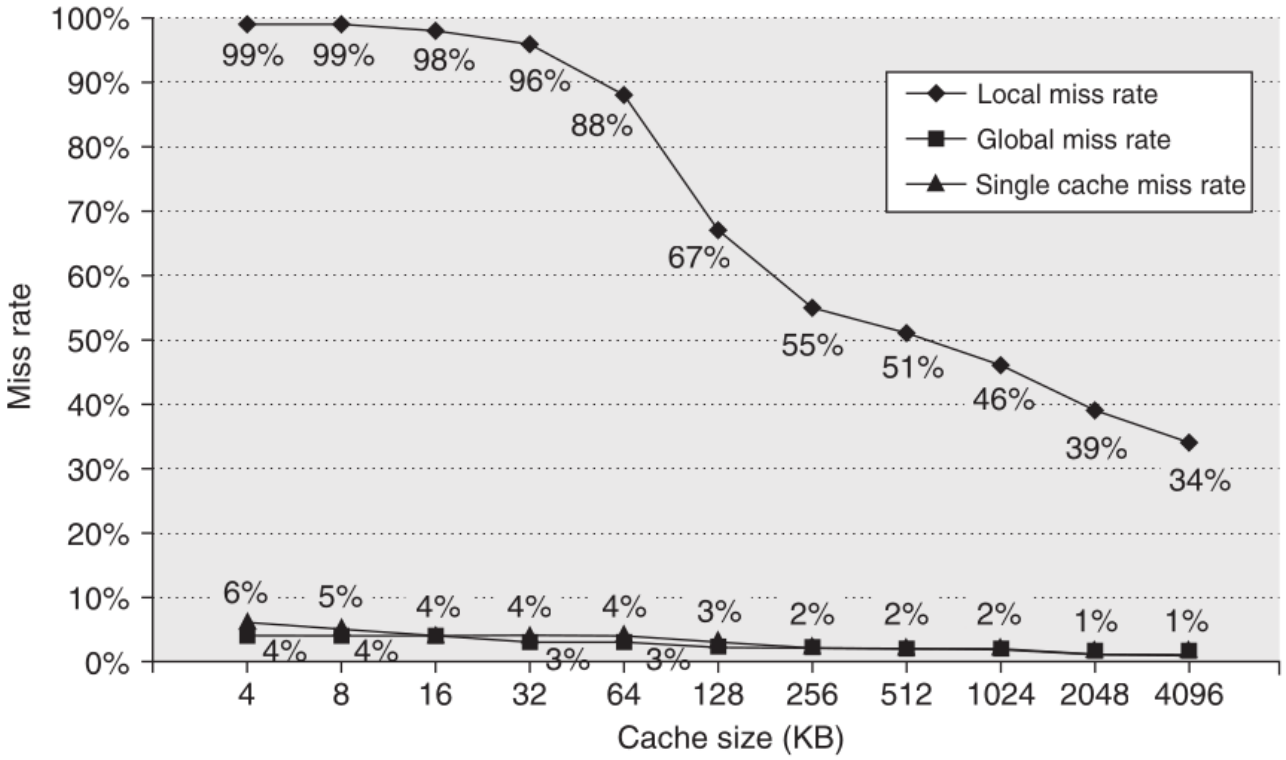


Figure B.14 Miss rates versus cache size for multilevel caches. Second-level caches *smaller* than the sum of the two 64 KB first-level caches make little sense, as reflected in the high miss rates. After 256 KB the single cache is within 10% of the global miss rates. The miss rate of a single-level cache versus size is plotted against the local miss rate and global miss rate of a second-level cache using a 32 KB first-level cache. The L2 caches (unified) were two-way set associative with replacement. Each had split L1 instruction and data caches that were 64 KB two-way set associative with LRU replacement. The block size for both L1 and L2 caches was 64 bytes. Data were collected as in Figure B.4.

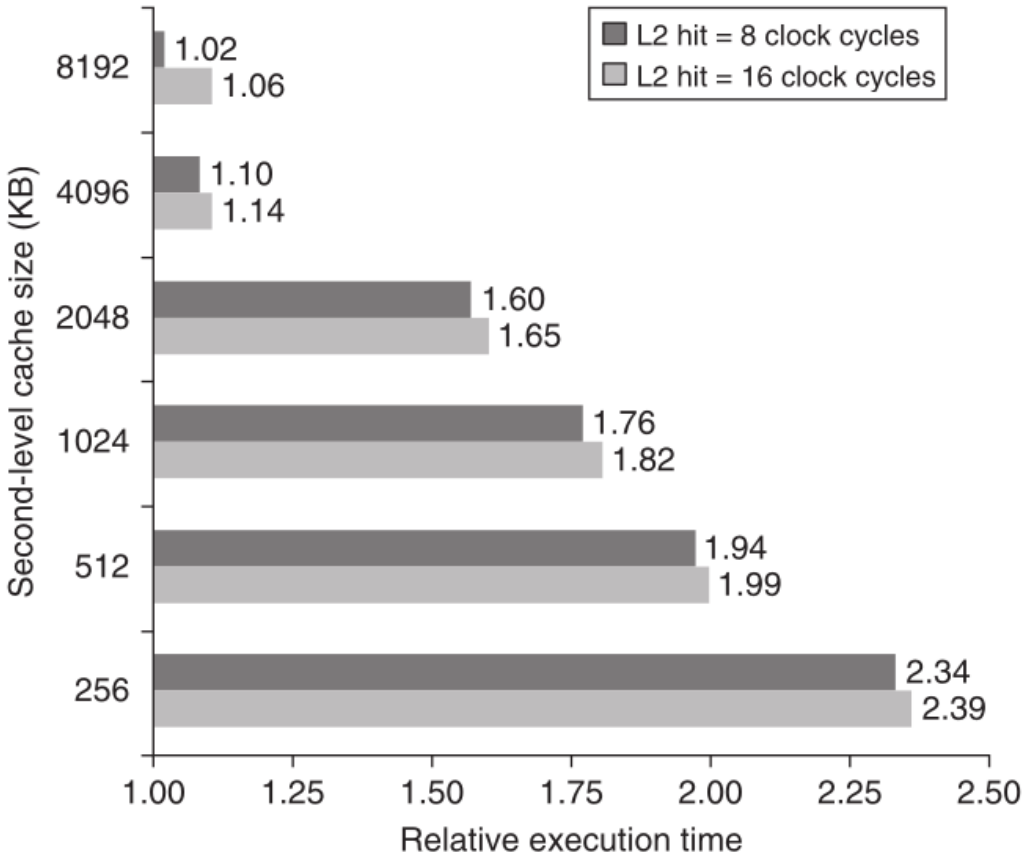


Figure B.15 Relative execution time by second-level cache size. The two bars are for different clock cycles for an L2 cache hit. The reference execution time of 1.00 is for an 8192 KB second-level cache with a 1-clock-cycle latency on a second-level hit. These data were collected the same way as in Figure B.14, using a simulator to imitate the Alpha 21264.

Figures B.14 and B.15 show how miss rates and relative execution time change with the size of a second-level cache for one design. From these figures we can gain two insights. The first is that the global cache miss rate is very similar to the single cache miss rate of the second-level cache, provided that the second-level cache is much larger than the first-level cache. Hence, our intuition and knowledge about the first-level caches apply. The second insight is that the local cache miss rate is *not* a good measure of secondary caches; it is a function of the miss rate of the first-level cache, and hence can vary by changing the first-level cache. Thus, the global cache miss rate should be used when evaluating second-level caches.

With these definitions in place, we can consider the parameters of second-level caches. The foremost difference between the two levels is that the speed of the first-level cache affects the clock rate of the processor, while the speed of the second-level cache only affects the miss penalty of the first-level cache. Thus, we can consider many alternatives in the second-level cache that would be ill chosen for the first-level cache. There are two major questions for the design of the second-level cache: Will it lower the average memory access time portion of the CPI, and how much does it cost?

The initial decision is the size of a second-level cache. Since everything in the first-level cache is likely to be in the second-level cache, the second-level cache should be much bigger than the first. If second-level caches are just a little bigger, the local miss rate will be high. This observation inspires the design of huge second-level caches—the size of main memory in older computers!

One question is whether set associativity makes more sense for second-level caches.

Example Given the data below, what is the impact of second-level cache associativity on its miss penalty?

- Hit time$_{L2}$ for direct mapped = 10 clock cycles.
- Two-way set associativity increases hit time by 0.1 clock cycle to 10.1 clock cycles.
- Local miss rate$_{L2}$ for direct mapped = 25%.
- Local miss rate$_{L2}$ for two-way set associative = 20%.
- Miss penalty$_{L2}$ = 200 clock cycles.

Answer For a direct-mapped second-level cache, the first-level cache miss penalty is

$$\text{Miss penalty}_{\text{1-way L2}} = 10 + 25\% \times 200 = 60.0 \text{ clock cycles}$$

Adding the cost of associativity increases the hit cost only 0.1 clock cycle, making the new first-level cache miss penalty:

$$\text{Miss penalty}_{\text{2-way L2}} = 10.1 + 20\% \times 200 = 50.1 \text{ clock cycles}$$

In reality, second-level caches are almost always synchronized with the first-level cache and processor. Accordingly, the second-level hit time must be an integral number of clock cycles. If we are lucky, we shave the second-level hit time to 10 cycles; if not, we round up to 11 cycles. Either choice is an improvement over the direct-mapped second-level cache:

$$\text{Miss penalty}_{\text{2-way L2}} = 10 + 20\% \times 200 = 50.0 \text{ clock cycles}$$
$$\text{Miss penalty}_{\text{2-way L2}} = 11 + 20\% \times 200 = 51.0 \text{ clock cycles}$$

Now we can reduce the miss penalty by reducing the *miss rate* of the second-level caches.

Another consideration concerns whether data in the first-level cache are in the second-level cache. *Multilevel inclusion* is the natural policy for memory hierarchies: L1 data are always present in L2. Inclusion is desirable because consistency between I/O and caches (or among caches in a multiprocessor) can be determined just by checking the second-level cache.

The passage discusses the concept of multilevel inclusion in the context of memory hierarchies, particularly in systems with multiple levels of caches. Here's a detailed explanation:

1. Memory hierarchy: Modern computer systems employ a memory hierarchy, where data is stored in multiple levels of memory with varying speeds and capacities. Typically, there are smaller but faster caches (L1, L2, L3, etc.) closer to the processor, and a larger but slower main memory (DRAM).
2. Multilevel inclusion: Multilevel inclusion is a policy that dictates that data present in a lower-level cache (e.g., L1) must also be present in the higher-level caches (e.g., L2, L3). In other words, the lower-level cache is a strict subset of the higher-level cache.

3. L1 data in L2 cache: The passage specifically mentions that under the multilevel inclusion policy, data present in the first-level cache (L1) are always present in the second-level cache (L2). This means that the L1 cache is a subset of the L2 cache, and any data that exists in L1 must also exist in L2.
4. Consistency and coherency: The main advantage of multilevel inclusion is that it simplifies the process of maintaining consistency and coherency between the caches and the main memory, or among caches in a multiprocessor system.
5. Checking L2 cache: With multilevel inclusion, to ensure consistency between the caches and the main memory (or among caches in a multiprocessor system), it is sufficient to check the second-level cache (L2). If the data is present and up-to-date in the L2 cache, it implies that the lower-level caches (L1) also have the correct data, as they are subsets of L2.
6. I/O and cache consistency: The passage specifically mentions that consistency between I/O (Input/Output operations, such as reading from or writing to main memory) and caches can be determined just by checking the second-level cache. This is because, with multilevel inclusion, if the data is present and up-to-date in L2, it ensures that the lower-level caches (L1) also have the correct data.

In summary, multilevel inclusion is a policy that simplifies cache coherency and consistency by ensuring that lower-level caches are subsets of higher-level caches. This allows the system to check only the higher-level cache (L2) to determine consistency between caches and main memory or among caches in a multiprocessor system, as the lower-level caches (L1) will always have the correct data if it is present and up-to-date in the higher-level cache.

One drawback to inclusion is that measurements can suggest smaller blocks for the smaller first-level cache and larger blocks for the larger second-level cache. For example, the Pentium 4 has 64-byte blocks in its L1 caches and 128-byte blocks in its L2 cache. Inclusion can still be maintained with more work on a second-level miss. The second-level cache must invalidate all first-level blocks that map onto the second-level block to be replaced, causing a slightly higher first-level miss rate. To avoid such problems, many cache designers keep the block size the same in all levels of caches.

The passage discusses a drawback of the multilevel inclusion policy and a potential solution to address this issue. Here's a detailed explanation:

1. Drawback of inclusion: One drawback of the multilevel inclusion policy is that it can prevent the system from using different block sizes for different levels of the cache hierarchy. Measurements and optimizations may suggest that smaller block sizes are more efficient for the first-level (L1) cache, while larger block sizes are better suited for the second-level (L2) cache.
2. Example: The passage cites the example of the Pentium 4 processor, where the L1 caches have a block size of 64 bytes, while the L2 cache has a larger block size of 128 bytes. This configuration violates the strict multilevel inclusion policy, as a single L2 block cannot be entirely contained within a single L1 block.
3. Maintaining inclusion: Even with different block sizes, inclusion can still be maintained, but it requires additional work on a second-level cache miss.
4. L2 cache miss handling: When there is a miss in the L2 cache, the cache controller needs to fetch the requested data from the main memory or a higher-level cache. However, due to the larger

block size in the L2 cache, multiple L1 cache blocks may need to be invalidated or updated to maintain inclusion.

5. Invalidating L1 blocks: To maintain inclusion, the L2 cache controller must invalidate all L1 cache blocks that map onto the L2 block being replaced. This process ensures that the L1 cache does not contain stale or inconsistent data compared to the L2 cache.
6. Higher L1 miss rate: The process of invalidating L1 cache blocks can lead to a slightly higher miss rate in the L1 cache. This is because data that was previously present in the L1 cache may need to be fetched again from the L2 cache or main memory after being invalidated.
7. Solution: To avoid the complexities and potential performance implications of maintaining inclusion with different block sizes, many cache designers choose to keep the block size the same across all levels of the cache hierarchy.

By maintaining the same block size across all cache levels, the cache controllers can seamlessly maintain inclusion without the need for additional bookkeeping or invalidation operations. This simplifies the cache design and avoids the potential performance penalty of higher L1 miss rates due to invalidations.

However, it's important to note that using different block sizes can sometimes provide performance benefits, especially if the access patterns and working set sizes differ significantly between the cache levels. In such cases, cache designers may choose to implement more complex inclusion mechanisms or even consider non-inclusive policies, weighing the trade-offs between complexity and potential performance gains.

> However, what if the designer can only afford an L2 cache that is slightly bigger than the L1 cache? Should a significant portion of its space be used as a redundant copy of the L1 cache? In such cases a sensible opposite policy is *multilevel exclusion:* L1 data are *never* found in an L2 cache. Typically, with exclusion a cache miss in L1 results in a swap of blocks between L1 and L2 instead of a replacement of an L1 block with an L2 block. This policy prevents wasting space in the L2 cache. For example, the AMD Opteron chip obeys the exclusion property using two 64 KB L1 caches and 1 MB L2 cache.
>
> As these issues illustrate, although a novice might design the first- and second-level caches independently, the designer of the first-level cache has a simpler job given a compatible second-level cache. It is less of a gamble to use a write-through, for example, if there is a write-back cache at the next level to act as a backstop for repeated writes and it uses multilevel inclusion.
>
> The essence of all cache designs is balancing fast hits and few misses. For second-level caches, there are many fewer hits than in the first-level cache, so the emphasis shifts to fewer misses. This insight leads to much larger caches and techniques to lower the miss rate, such as higher associativity and larger blocks.

The passage discusses several important concepts related to the design and organization of multi-level cache hierarchies. Let's go through each point in detail:

1. Block size across cache levels: The passage starts by considering the scenario where the designer can only afford an L2 cache that is slightly bigger than the L1 cache. In such a case, if the

multilevel inclusion policy is followed (where L1 data must be present in L2), a significant portion of the L2 cache space would be used as a redundant copy of the L1 cache, which is inefficient.
2. Multilevel exclusion policy: To address this issue, the passage introduces the concept of multilevel exclusion policy. Under this policy, L1 data is never found in the L2 cache. Instead of replacing an L1 block with an L2 block on a miss, a swap of blocks between L1 and L2 takes place. This policy prevents wasting space in the L2 cache by keeping it exclusive from the L1 cache.

The AMD Opteron chip is given as an example that follows the exclusion property, using two 64 KB L1 caches and a 1 MB L2 cache.

3. Interdependence of cache levels: The passage then highlights that although a novice might design the first- and second-level caches independently, the designer of the first-level cache has a simpler job if the second-level cache is compatible. For example, using a write-through policy in the L1 cache is less risky if there is a write-back cache at the next level (L2) to act as a backstop for repeated writes, and if it uses multilevel inclusion.
4. Balancing hits and misses: The essence of all cache designs is balancing fast hits (when data is found in the cache) and few misses (when data needs to be fetched from main memory or a higher-level cache). For second-level caches, there are many fewer hits than in the first-level cache, so the emphasis shifts to reducing misses.
5. Techniques for reducing misses in L2 cache: To reduce misses in the second-level cache, the passage suggests using larger cache sizes and techniques like higher associativity and larger block sizes. These techniques can help lower the miss rate, thereby improving overall performance.

Fifth Optimization: Giving Priority to Read Misses over Writes

This optimization serves reads before writes have been completed. We start with looking at the complexities of a write buffer.

With a write-through cache the most important improvement is a write buffer of the proper size. Write buffers, however, do complicate memory accesses because they might hold the updated value of a location needed on a read miss.

Example Look at this code sequence:

```
SW R3, 512(R0)     ;M[512] ← R3     (cache index 0)
LW R1, 1024(R0)    ;R1 ← M[1024]    (cache index 0)
LW R2, 512(R0)     ;R2 ← M[512]     (cache index 0)
```

Assume a direct-mapped, write-through cache that maps 512 and 1024 to the same block, and a four-word write buffer that is not checked on a read miss. Will the value in R2 always be equal to the value in R3?

Answer Using the terminology from Chapter 2, this is a read-after-write data hazard in memory. Let's follow a cache access to see the danger. The data in R3 are placed into the write buffer after the store. The following load uses the same cache index and is therefore a miss. The second load instruction tries to put the value in location 512 into register R2; this also results in a miss. If the write buffer hasn't completed writing to location 512 in memory, the read of location 512 will put the old, wrong value into the cache block, and then into R2. Without proper precautions, R3 would not be equal to R2!

The simplest way out of this dilemma is for the read miss to wait until the write buffer is empty. The alternative is to check the contents of the write buffer on a read miss, and if there are no conflicts and the memory system is available, let the read miss continue. Virtually all desktop and server processors use the latter approach, giving reads priority over writes.

The cost of writes by the processor in a write-back cache can also be reduced. Suppose a read miss will replace a dirty memory block. Instead of writing the dirty block to memory, and then reading memory, we could copy the dirty block to a buffer, then read memory, and *then* write memory. This way the processor read, for which the processor is probably waiting, will finish sooner. Similar to the previous situation, if a read miss occurs, the processor can either stall until the buffer is empty or check the addresses of the words in the buffer for conflicts.

Now that we have five optimizations that reduce cache miss penalties or miss rates, it is time to look at reducing the final component of average memory access time. Hit time is critical because it can affect the clock rate of the processor; in many processors today the cache access time limits the clock cycle rate, even for processors that take multiple clock cycles to access the cache. Hence, a fast hit time is multiplied in importance beyond the average memory access time formula because it helps everything.

The passage discusses several techniques and optimizations related to handling read and write misses in a cache, as well as reducing hit times. Here's a detailed explanation:

1. Write buffer and read miss handling: The passage presents a dilemma when a read miss occurs while there are outstanding writes in the write buffer. Two approaches are discussed: a) The simplest approach is to make the read miss wait until the write buffer is empty before processing the read miss. b) The alternative approach, used by virtually all desktop and server processors, is to check the contents of the write buffer on a read miss. If there are no conflicts with the read miss address and the memory system is available, the read miss is allowed to continue. This approach gives priority to reads over writes.
2. Optimizing write-back cache: The passage describes an optimization for write-back caches when a read miss replaces a dirty (modified) memory block. a) Instead of first writing the dirty block to memory and then reading the missed data, the dirty block is copied to a buffer. b) The missed data is then read from memory. c) Finally, the dirty block from the buffer is written back to memory.

This optimization allows the processor read, for which the processor is likely waiting, to finish sooner.

3. Read miss and buffer conflicts: Similar to the write buffer scenario, when a read miss occurs while there are words in the buffer, the processor can either: a) Stall until the buffer is empty, or b) Check the addresses of the words in the buffer for conflicts with the read miss address.
4. Reducing cache miss penalties and miss rates: The passage mentions that it has covered five optimizations that reduce cache miss penalties or miss rates, referring to the techniques discussed earlier.
5. Importance of hit time: The passage then shifts focus to reducing the final component of average memory access time: hit time. a) Hit time is critical because it can affect the clock rate of the processor. In many modern processors, the cache access time limits the clock cycle rate, even for processors that take multiple clock cycles to access the cache. b) A fast hit time is multiplied in importance beyond the average memory access time formula because it helps improve overall performance.

Sixth Optimization: Avoiding Address Translation during Indexing of the Cache to Reduce Hit Time

The passage discusses the sixth optimization technique, which is avoiding address translation during cache indexing to reduce hit time. It explains the concept of virtual caches and contrasts them with physical caches. Here's a detailed explanation:

Virtual vs. Physical Caches:
1. Virtual caches use virtual addresses from the processor to index the cache and compare tags.
2. Physical caches use physical addresses translated from virtual addresses to access memory.

Rationale for Virtual Caches:
1. Virtual caches eliminate the need for address translation during cache hits, which are more common than misses.
2. This approach follows the guideline of making the common case fast.

Reasons for not using Virtual Caches: a. Protection: Page-level protection checks are part of the virtual-to-physical address translation process and must be enforced, even for virtual caches. b. Process Switching: Every time a process is switched, virtual addresses refer to different physical addresses, requiring the cache to be flushed. Using process identifiers (PIDs) in the cache tags can help avoid cache flushes when a PID is recycled. c. Synonyms/Aliases: Operating systems and user programs may use different virtual addresses for the same physical address, leading to duplicate copies of data in a virtual cache. Physical caches avoid this issue.

Hardware Solutions for Synonyms (Antialiasing):
1. Hardware mechanisms guarantee each cache block a unique physical address.
2. Example: AMD Opteron checks all possible locations on a miss to invalidate any matching physical addresses.
Software Solutions (Page Coloring):
1. Software can force aliases to share some address bits, restricting the problem.

2. Page coloring in older UNIX versions required aliases to be identical in the last 18 bits, effectively increasing the page offset for the cache.

I/O Interaction with Virtual Caches:
1. I/O typically uses physical addresses, requiring mapping to virtual addresses to interact with a virtual cache.

Virtually Indexed, Physically Tagged Alternative:
1. Use part of the page offset (identical in virtual and physical addresses) to index the cache.
2. Translate the virtual part of the address while the cache is being read, and use physical addresses for tag comparison.
3. Limitation: Direct-mapped cache size cannot exceed the page size due to the index constraint.

Organization with Virtually Indexed, Physically Tagged Caches:
1. The cache is indexed using part of the virtual address (page offset).
2. The virtual part of the address is translated in parallel using Translation Lookaside Buffers (TLBs).
3. The tag comparison uses physical addresses obtained from the TLB translation.

Avoid address translation for indexing

- **Avoid address translation in cache indexing to reduce hit time**
- Software uses virtual address (VA), but memory accessed using physical address (PA)
- VA to PA has to be done before cache look up **[MMU, TLB]**
- If indexing & tag comparison in virtual address, better hit time
- **Solution**: Start indexing with VA
- In the meantime do address translation and get PA and tag.
- Role of TLB, TLB access in critical path.
- Indexing by virtual address and tagging by physical address (VIPT cache)

What is the Problem?

Accessing data from Cache involve:

- Indexing
- Tagging

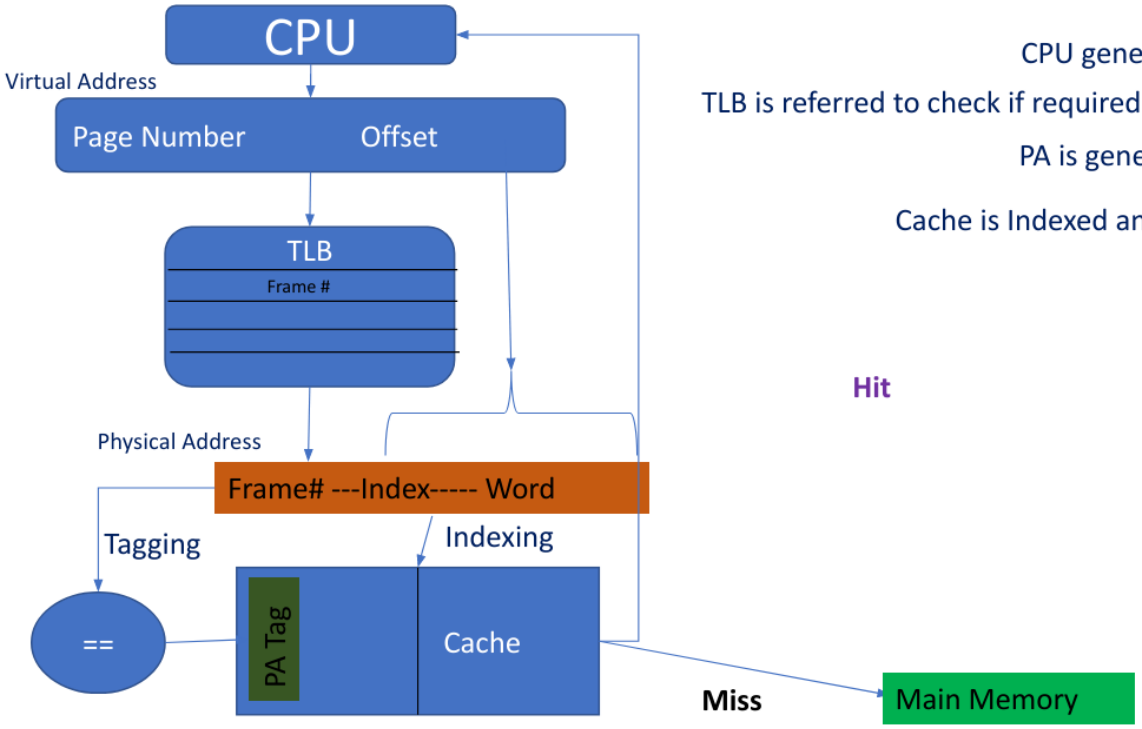
Converting Virtual Address to Physical Address take some time.

Solution ☺

Virtually Indexed & Physically Tagged

- Do not wait for VA to PA translation.
- Extract Index information from Virtual Address
- Extract Tag Information from the Physical Address.

Physically Indexed and Physically Tagged (PIPT)

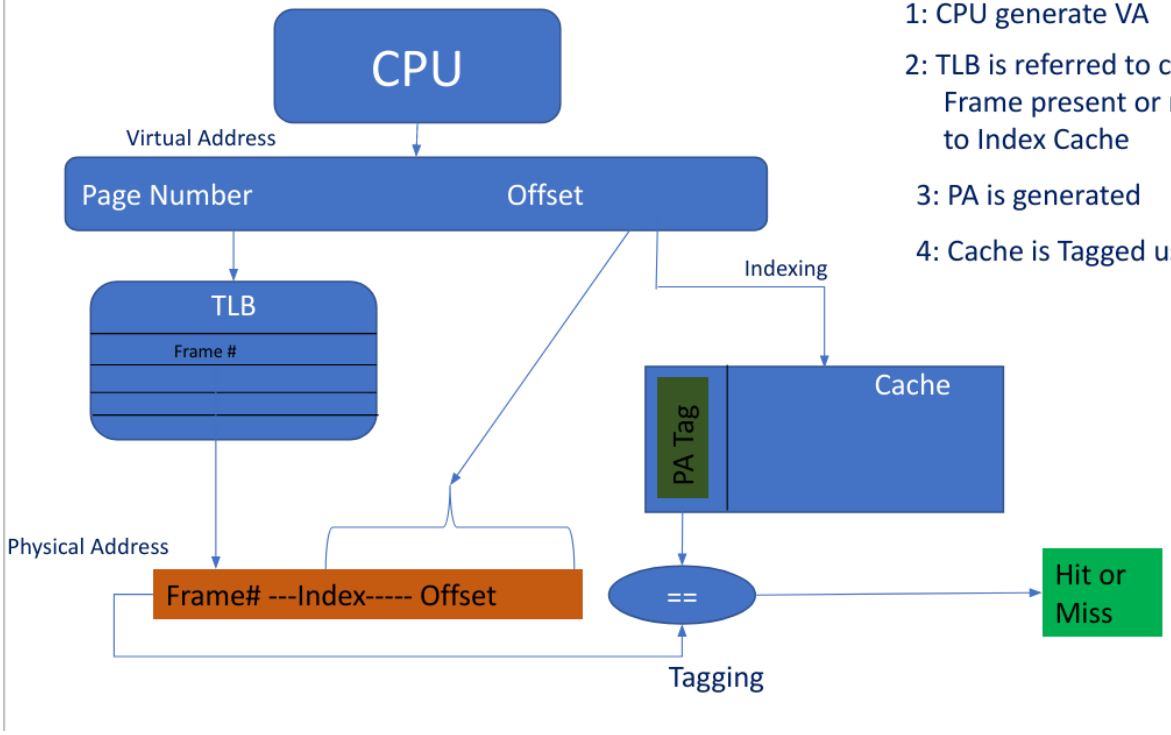


CPU generate VA

TLB is referred to check if required Frame present or not

PA is generated

Cache is Indexed and Tagged using PA



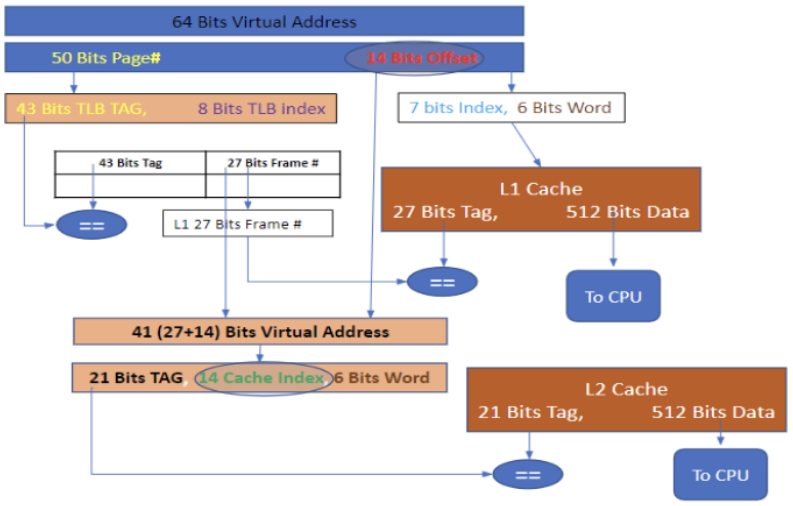
1: CPU generate VA

2: TLB is referred to check if required Frame present or not & Offset is used to Index Cache

3: PA is generated

4: Cache is Tagged using PA

6th Optimization : Optimized to Improve the Hit Time



- **Modified Specification for Cache**
- 64 bit Virtual Address
- 41 bit Physical Address
- 16 KB (2^14)Page Size
- **2 Way, 256 Entry TLB**
- 16 KB (2^13) Direct Mapped L1 Cache
- Block Size = 64 Bytes (2^6 words)
- 4 MB, 4 Way, L2 Cache
- 2^22/2^6x 2^2= 2^14

Note that page coloring is simply set associative mapping applied to virtual memory

Associativity can keep the index in the physical part of the address and yet still support a large cache. Recall that the size of the index is controlled by this formula:

$$2^{\text{Index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}}$$

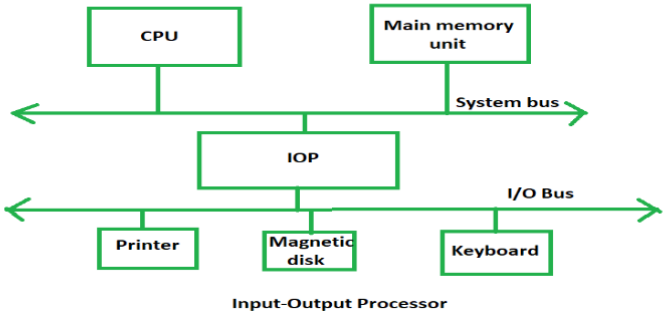
For example, doubling associativity and doubling the cache size does not change the size of the index. The IBM 3033 cache, as an extreme example, is 16-way set associative, even though studies show there is little benefit to miss

rates above 8-way set associativity. This high associativity allows a 64 KB cache to be addressed with a physical index, despite the handicap of 4 KB pages in the IBM architecture.

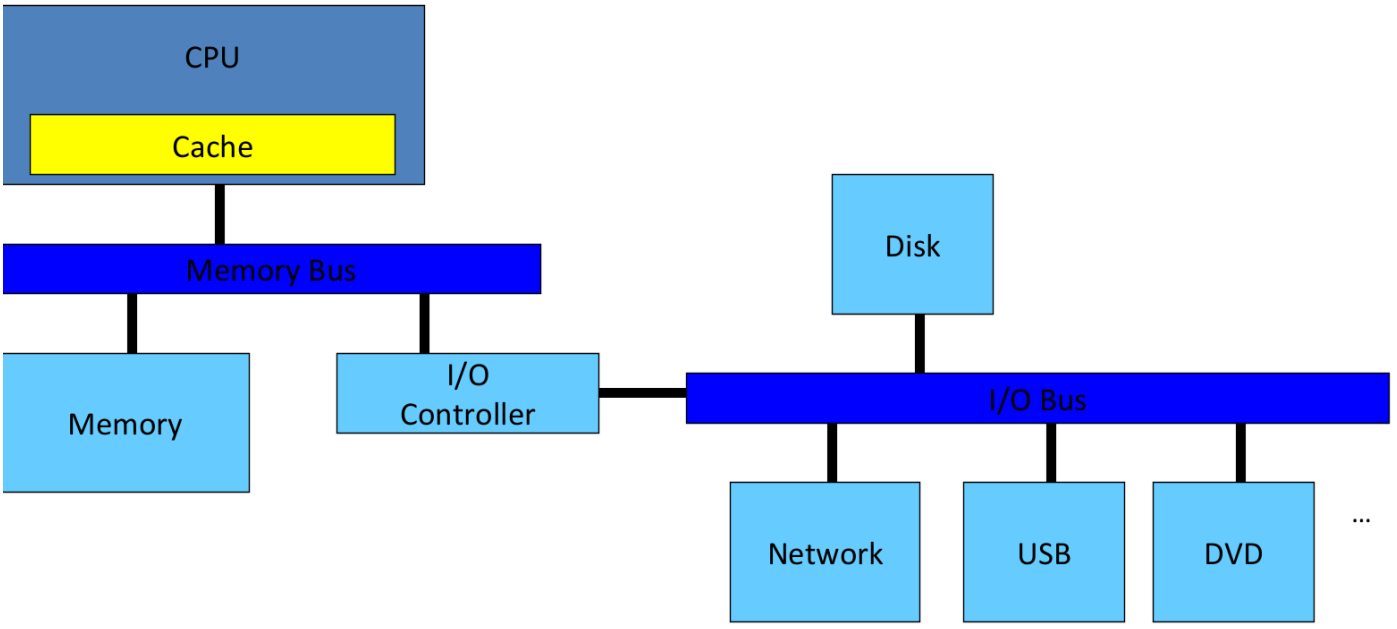
I/O Bus Architecture

Accessing I/O Devices - Introduction

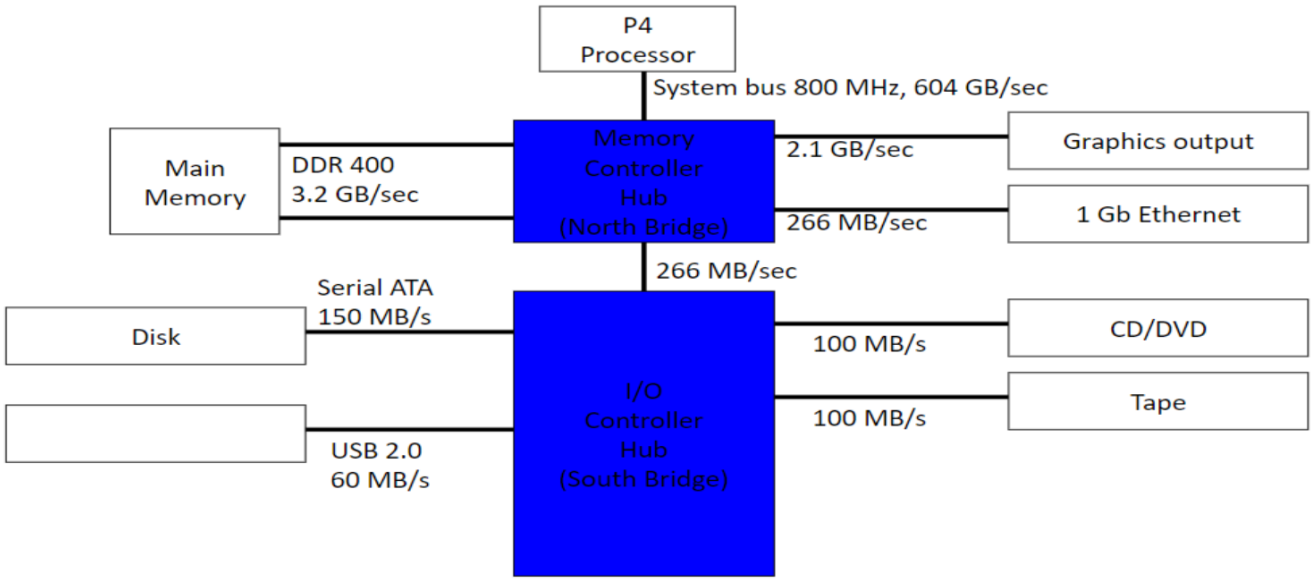
- The method that is used to transfer information between internal storage and external I/O devices is known as I/O interface.
- A typical link between the processor , memory and several peripherals.
- These devices work at varying speeds.
  - Ex: Monitors, Mouse, Keyboards, Video cameras, etc.,
  - Data transfer rate can either be regular or irregular.



Accessing I/O Devices – I/O Hierarchy

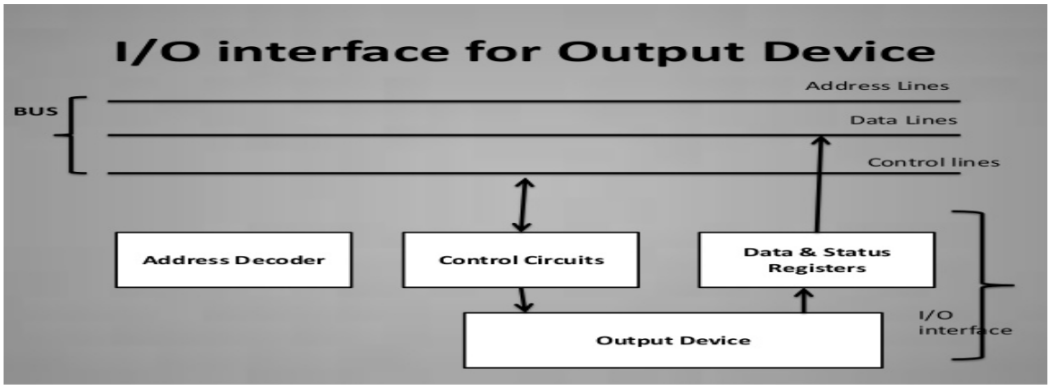


Accessing I/O Devices – Intel Example



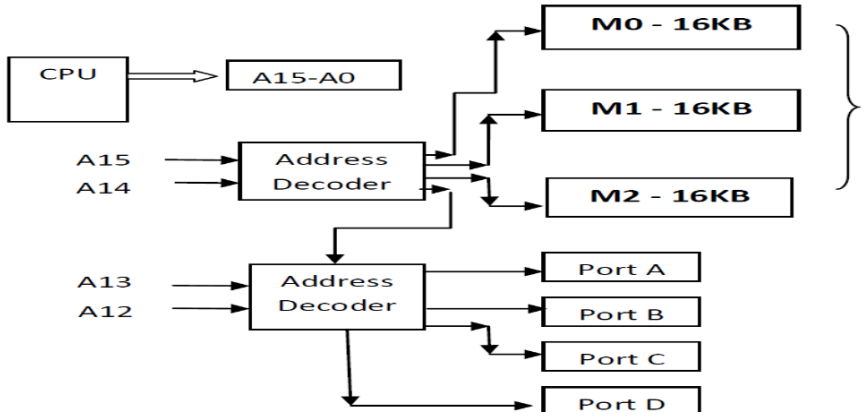
Accessing I/O Devices - Introduction

- There exists special hardware components between CPU and peripherals to supervise and synchronize all the input and output transfers that are called interface units.
- The I/O **Bus** consists of data lines, address lines and control lines.
- The I/O **bus** from the processor is attached to all peripherals interface.
- To communicate with a particular device, the processor places a device address on address lines.



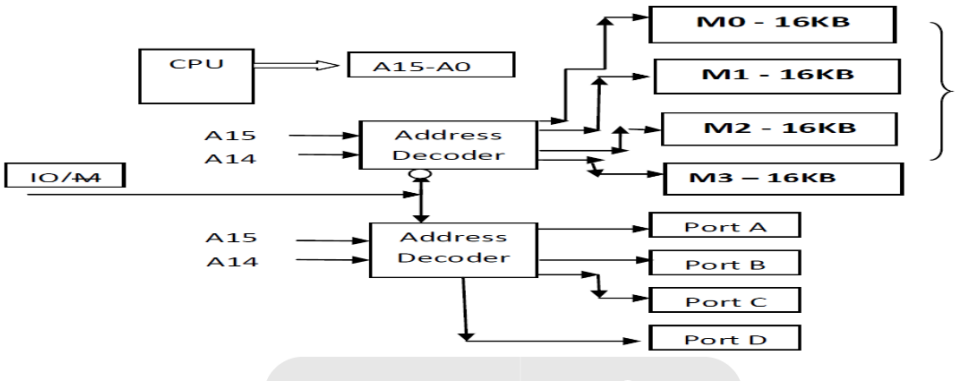
Accessing I/O Devices - Memory Mapped Device Interface

- **Memory Mapped I/O device interface:**
  - Same address decoder selects memory and I/O ports.
  - Some memory address space is occupied by the I/O devices.
  - All data transfer instructions to / from memory an be used to transfer to/from I/O devices.
  - Processor need not have separate instructions for I/O.



Accessing I/O Devices - I/O Mapped device interface

- **I/O Mapped I/O device interface:**
  - Separate instructions for I/O data transfer [IN / OUT].
  - Processor signal identifies whether a generated address refers to memory or I/O device.
  - Separate address decoder for selecting memory and I/O ports.
  - Complete memory address space can be utilized.



Accessing I/O Devices - Data Transfer Techniques

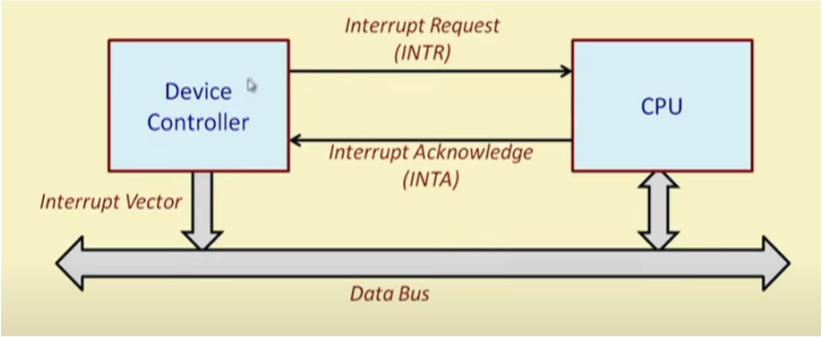
- Data transfer to and from the peripherals may be done in any of these ways:
  1. **Programmed I/O:** **CPU executes a program and that transfers data between I/O device and Memory**
     a. **Synchronous** : Fixed rate of transfer
     b. **Asynchronous :** Handshaking – polling for sending / receiving the data
     c. **Interrupt- Driven :(next slide)**
  2. **Direct memory access(DMA):** **An external controller directly transfers data between I/O device and Memory without CPU intervention.**

Accessing I/O Devices – Interrupt/Exception?

c. **Interrupt- Driven:**
   - CPU initiates the data transfer and proceeds to perform some other task.
   - When I/O module is ready for data transfer, it informs the CPU by activating a signal (Interrupt request).
   - The CPU suspends the task it was doing, services the request from the device and return back to the task it was doing.
- **Advantages:**
  - CPU time is not wasted by polling the device
  - CPU time is required only during the data transfer plus some overheads for transferring and returning the control.

c. **What happens when an interrupt request arrives?**

- At the end of the execution of the current instruction, PC and the status register contents are saved in the stack automatically.
- Interrupt is acknowledged, interrupt vector is obtained based which the control transfers to the appropriate ISR.
- After handling the interrupt, the ISR executes a special instruction *return from interrupt(RTI)*.

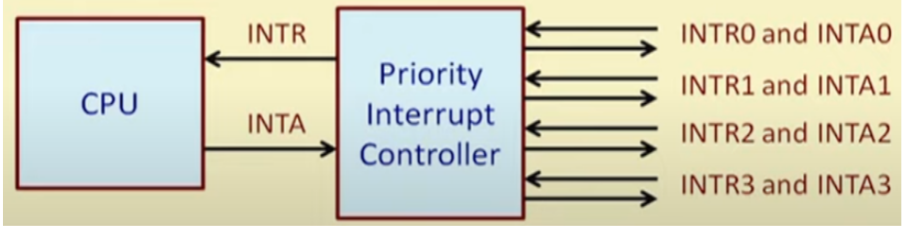


ARM Interrupt Vector Table

	Address	Exception	Mode in Entry
1	0x00000000	Reset	Supervisor
2	0x00000004	Undefined instruction	Undefined
3	0x00000008	Software Interrupt	Supervisor
4	0x0000000C	Abort (prefetch)	Abort
5	0x00000010	Abort (data)	Abort
x	0x00000014	Reserved	Reserved
6	0x00000018	IRQ (external interrupt)	IRQ
7	0x0000001C	FIQ (fast interrupt)	FIQ

Accessing I/O Devices – Multiple devices interrupting CPU

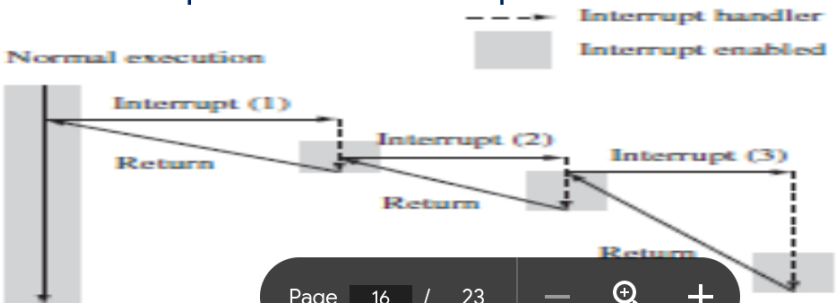
- Common solution is to use a **Priority Interrupt Controller.**
  - Interrupt controller interacts with CPU on one side and multiple devices on the other side.
  - For simultaneous interrupt requests, interrupt priority is defined.
  - Interrupt controller is responsible for sending the interrupt vector to the CPU.



Accessing I/O Devices - How is interrupt nesting handled?

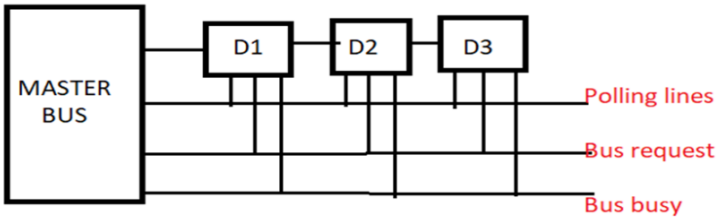
Consider a scenario.

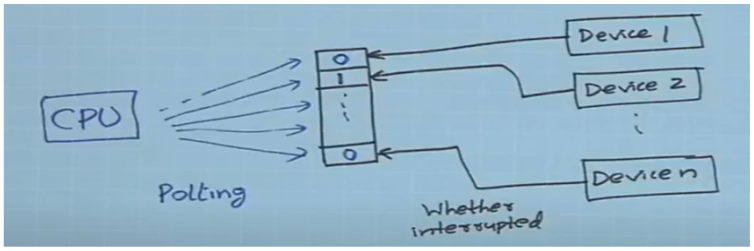
- A device D0 has interrupted and the CPU is servicing the (executing the ISR) device D0.
- In the meantime, device D1 has interrupted.
- Two possible scenarios are here:
  1. D1 will interrupt the ISR for D0, get processed first, and then the ISR for device D0 will be resumed.
     This creates problem for multi nesting.
  2. Disable the interrupt system automatically whenever an interrupt is acknowledged.
     This will not require nested interrupts to be handled.



Accessing I/O Devices – Polling Technique

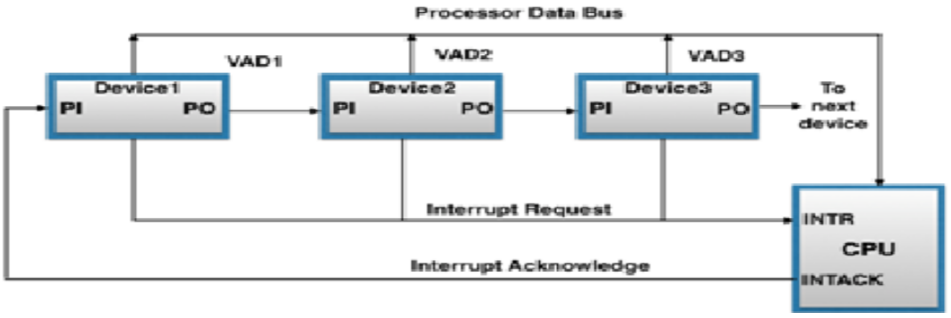
- Each device can have a status bit whether the device has interrupted?
- CPU can poll the status bit to check for the device which has interrupted.





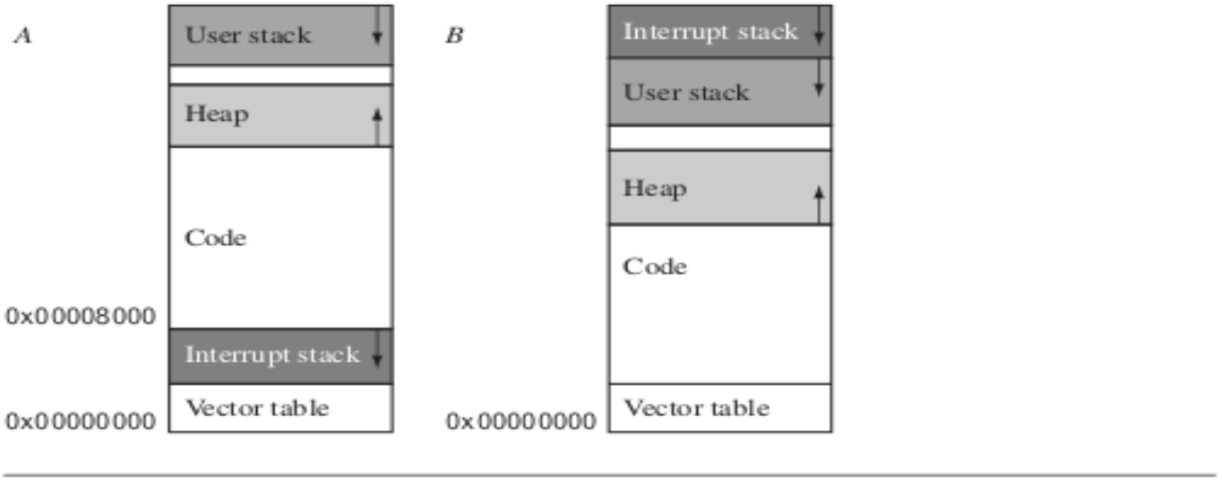
Accessing I/O Devices – Daisy Chain Technique

- In Daisy chain technique, INTR line is common to all the devices.
- INTA line is connected in a daisy chain fashion [as shown].
- This allows to propagate serially through the devices.
- A device when it receives INTA, passes the signal to the next device only if it has not interrupted.
- Else, it stops the propagation of INTA and puts the identifying code on the data bus.
- Thus the device that is electrically closest to the CPU will have the highest priority.



Accessing I/O Devices – Basic Interrupt Stack Design and Implementation

Stack Layouts:



Accessing I/O Devices - Basic Interrupt Stack Design and Implementation

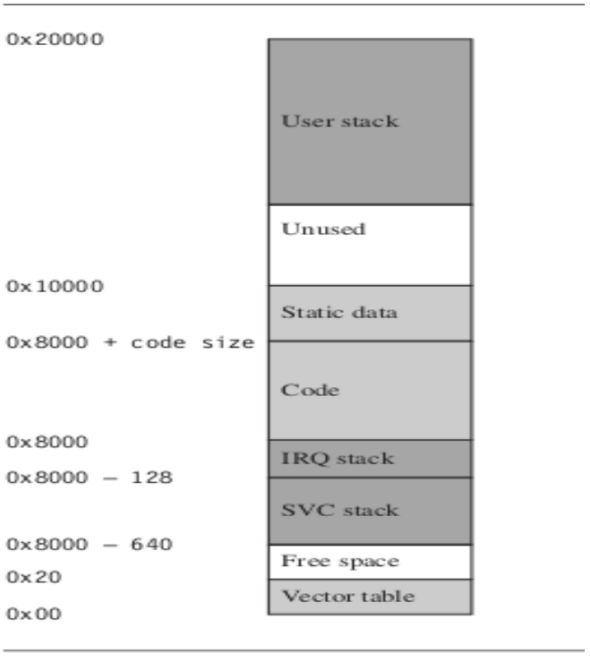
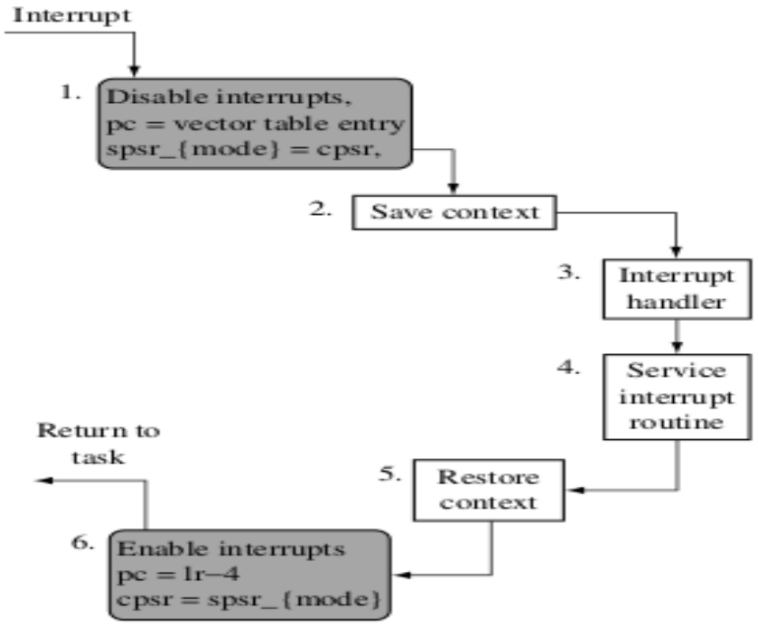


Figure 9.7 Example implementation using layout *A*.

Accessing I/O Devices – Interrupt Handling Schemes \ Non-Nested Interrupt Handler

- Various stages in a NNIH –
  - Enable Interrupts
    - spsr ← spsr_{interrupt request mode}
    - pc is restored

```
interrupt_handler
        SUB     r14,r14,#4                      ; adjust lr
        STMFD   r13!,{r0-r3,r12,r14}            ; save context
        <interrupt service routine>
        LDMFD   r13!,{r0-r3,r12,pc}^            ; return
```



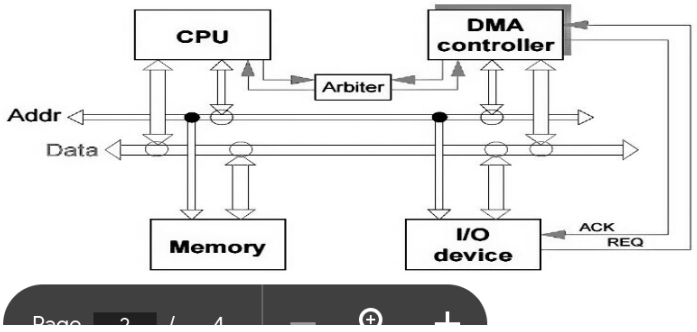
Direct Memory Access(DMA) Controller

What is actually DMA?

- The direct memory access is a system where the samples are saved in the memory of the system while the processor does something else.

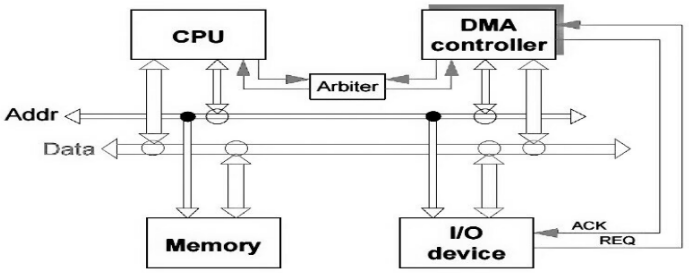
Why we need DMA?

- The assumption about the I/O machines like keyboards, mouse, and printer etc. are genuinely very slow when compared with the central processing unit (CPU).
- To overcome this issue an interrupt handler was used and the I/O machine produces all the signals that the CPU produce, then the I/O machine can bypass the information alienates to central processing unit and hence increases the speed.



Direct Memory Access(DMA) Controller

- **Cycle Stealing Process:** Traditionally it is a method of accessing RAM or bus without interfering with the CPU.
- DMA allows I/O controllers to read or write RAM without CPU intervention.
- Clever exploitation of specific CPU or bus timings can permit the CPU to run at full speed without any delay if external devices access memory not actively participating in the CPU's current activity and complete the operations before any possible CPU conflict.
- Such systems are nearly dual-port RAM without the expense of high speed RAM.
- Most systems halt the CPU during the **steal**, essentially making it a form of DMA by another name



Universal Serial Bus Architecture

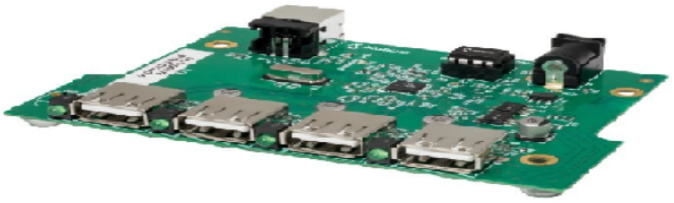
Universal Serial Bus is a data interface used with computers enabling the computer to send and receive data as well as providing power to some peripherals like disc drives, Flash memory sticks and the like so that separate power sources are not needed for each item

Universal Serial Bus Architecture

- USB Peripherals are slaves responding to commands from hosts
- When a peripheral is attached to the USB network, the host communicates with the device.
  - To learn its identity
  - To discover which device driver is required
    - This is called Enumeration
    - It is supported as the device driver for the USB port on the host.
- **Power :**
  - USB devices can pull limited amount power from the bus.
- **Speed:**
  - Low : 1.5Mbps
  - Full : 12Mbps
  - High : 480 Mbps

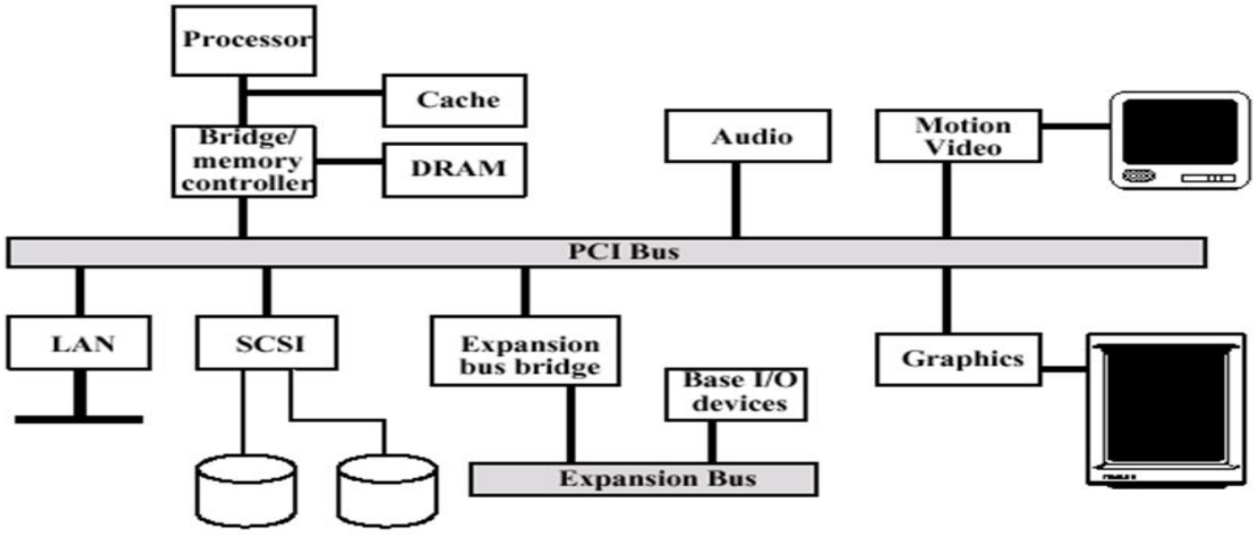
Universal Serial Bus Architecture

- USB Devices are of two types.
  - Stand-alone:
    - Single Function Units like Mouse,etc.,
  - Compound Devices:
    - More than one peripheral sharing a USB port.
      Ex: Video Camera (both audio and video).
- USB Hubs:
  - Hubs are bridge
  - Themselves are USB devices
  - Hubs detect topology changes due to insertion and deletion of devices.

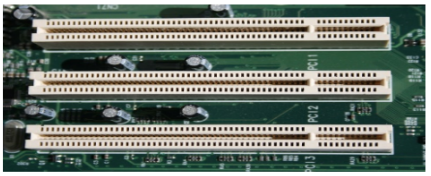


Peripheral Component Interconnect : PCI-Bus Architecture

- PCI is a local computer bus for attaching hardware devices in a computer.



Peripheral Component Interconnect : PCI-Bus Architecture



Features:

- **32 bit bus**
- **Transfer rate : 133MB/s**
- Any PCI device may initiate a transaction.
- First, it must request permission from a PCI bus arbiter on the motherboard.
- The arbiter grants permission to one of the requesting devices.
- The initiator begins the address phase by broadcasting a 32- bit address plus a 4-bit command code, then waits for a target to respond.
- All other devices examine this address and one of them responds a few cycles later.

SCSI – Bus Architecture

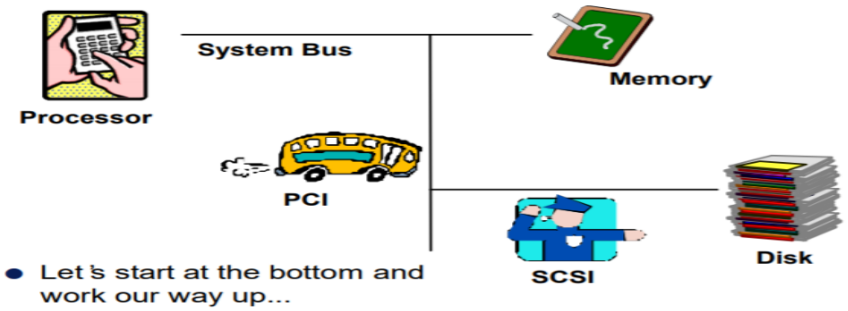
Small Computer System Interface – SCSI :

- Derived from **SASI –** Shugart Associates System Interface
- Provided as a bridge between hard disk low-level interface and a host computer
- Standard Interface and communication protocol for connecting computer peripherals.
- SCSI is used to increase performance and deliver faster
- Bus can address upto 8 devices (0 to 7).
- Provide larger expansion for devices such as CDs, scanners, etc.,
- Based on the Client Server Architecture
- Clients are Initiators – creates(begins)and sends SCSI commands to the target.
- Servers are Targets – Collection of logical units – carries out the requested task.
- Early SCSI assumes a bus based architecture

SCSI – Bus Architecture

Small Computer System Interface – SCSI :

Parallel SCSI: SCSI-1 : 8 bit wide bus – speed 3.5MHz
SCSI-2 : 16 /32 bit wide – speed 10MHz to 20MHz
SCSI-3 : Also called as ultra SCSI.
Cable length- 3 mts.
Speed – 20MHz



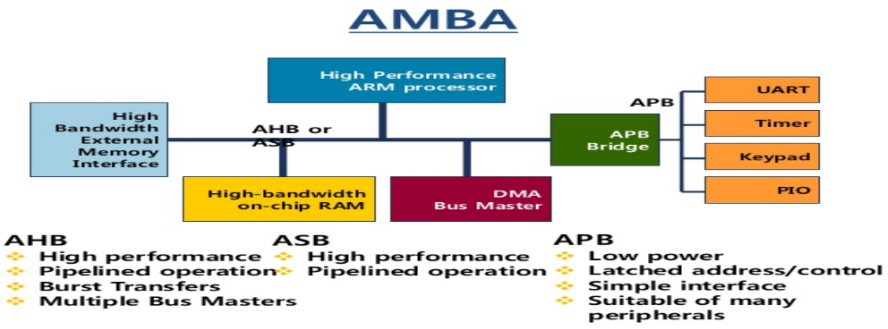
SCSI – Bus Architecture

SCSI Bus Operation in phases:

- Bus Free
- Arbitration
- Selection / Reselection : target in control
  - Target decides when to recv msg, cmd, data from initiator.
  - OR send status to initiator
- Message
  - Used by target to send / recv protocol information.
    Eg : Identify, cmd complete, msg parity error, etc.,
- Data, command, message, status

AMBA (Advanced Microprocessor Bus Architecture) protocol

- The Arm AMBA protocols are an open standard, on-chip interconnect specification for the connection and management of functional blocks in a System-on-Chip (SoC).
- It facilitates right-first-time development of multi-processor designs with large numbers of controllers and peripherals.
- Features of AMBA Interfaces:
  - IP reuse is essential in reducing SoC development costs and timescales. AMBA specifications provide interface standards that enables IP reuse if the following essential requirements are met:
  - Flexibility:
  - Wide Adaption:
  - Compatibility:
  - Support:



APB (Advanced Peripheral Bus) protocol

- The Advanced Peripheral Bus (APB) is part of the Advanced Microcontroller Bus Architecture (AMBA) protocol family.
- It defines a low-cost interface that is optimized for minimal power consumption and reduced interface complexity.
- The APB protocol is not pipelined, use it to connect to low-bandwidth peripherals that do not require the high performance of the AXI protocol.
- The APB protocol relates a signal transition to the rising edge of the clock, to simplify the integration of APB peripherals into any design flow.
- Every transfer takes at least two cycles.
- The APB can interface with:

- AMBA Advanced High-performance Bus (AHB)
- AMBA Advanced High-performance Bus Lite (AHB-Lite)
- AMBA Advanced Extensible Interface (AXI)
- AMBA Advanced Extensible Interface Lite (AXI4-Lite)