Course on: "Advanced Computer Architectures"

Introduction to cache memories

Prof. Cristina Silvano
Politecnico di Milano
email: **cristina.silvano@polimi.it**

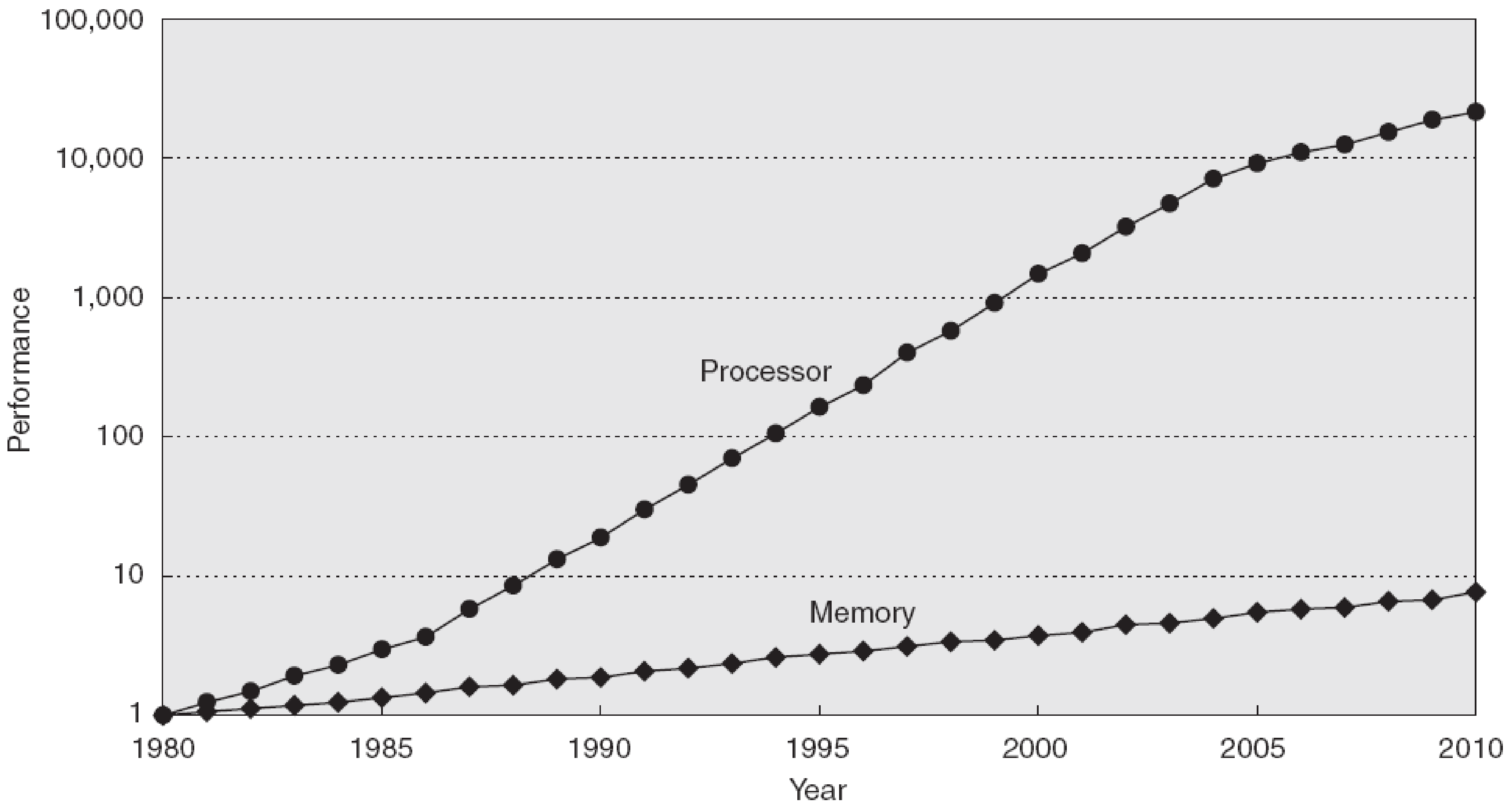
Summary

- Main goal
- Spatial and temporal locality
- Levels of memory hierarchy
- Cache memories: basic concepts
- Architecture of a cache memory:
  - Direct Mapped Cache
  - Fully associative Cache
  - N-way Set-Associative Cache
- Performance Evaluation
- Memory Technology

Main goal

- To increase the performance of a computer through the memory system in order to:
  - Provide the user the illusion to use a memory that is simultaneously large and fast
  - Provide the data to the processor at high frequency

Problem: Processor-memory performance gap



Solution: to exploit the principle of locality of references

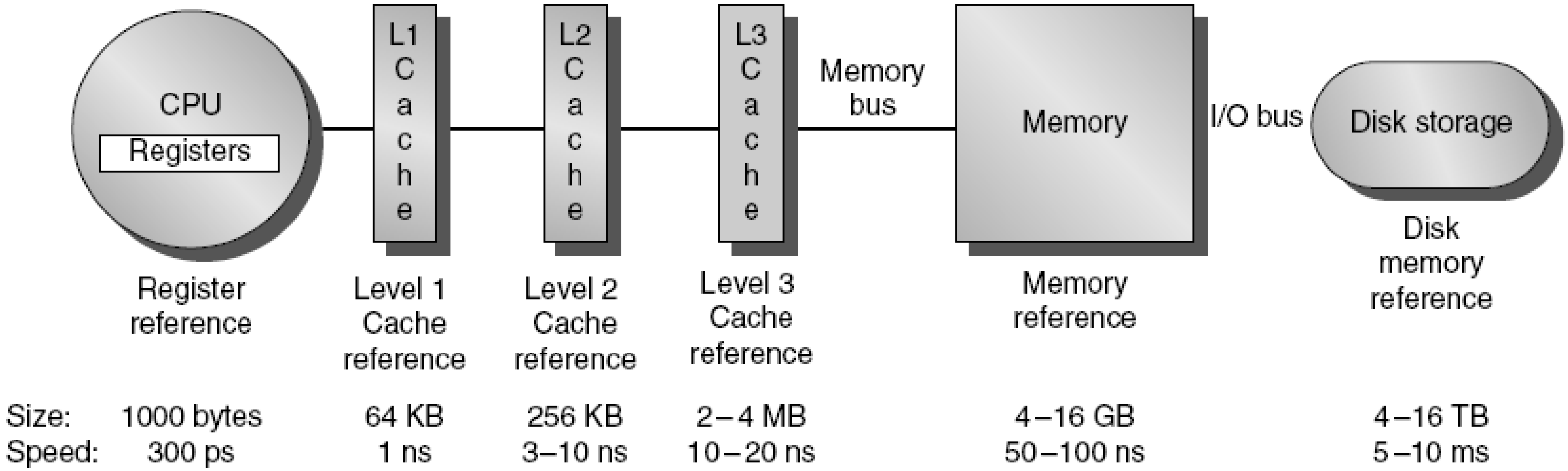
- **Temporal Locality:** when there is a reference to one memory element, the trend is to refer again to the same memory element soon (i.e., instruction and data reused in loop bodies)

- **Spatial Locality:** when there is a reference to one memory element, the trend is to refer soon at other memory elements whose addresses are close by (i.e., sequence of instructions or accesses to data organized as arrays or matrices)

Caches

- Caches exploit both types of predictability:
  - Exploit **temporal locality** by keeping the contents of recently accessed locations.
  - Exploit **spatial locality** by fetching blocks of data around recently accessed locations.

Levels of Memory Hierarchy



CPU

Registers

L1 Cache

L2 Cache

L3 Cache

Memory bus

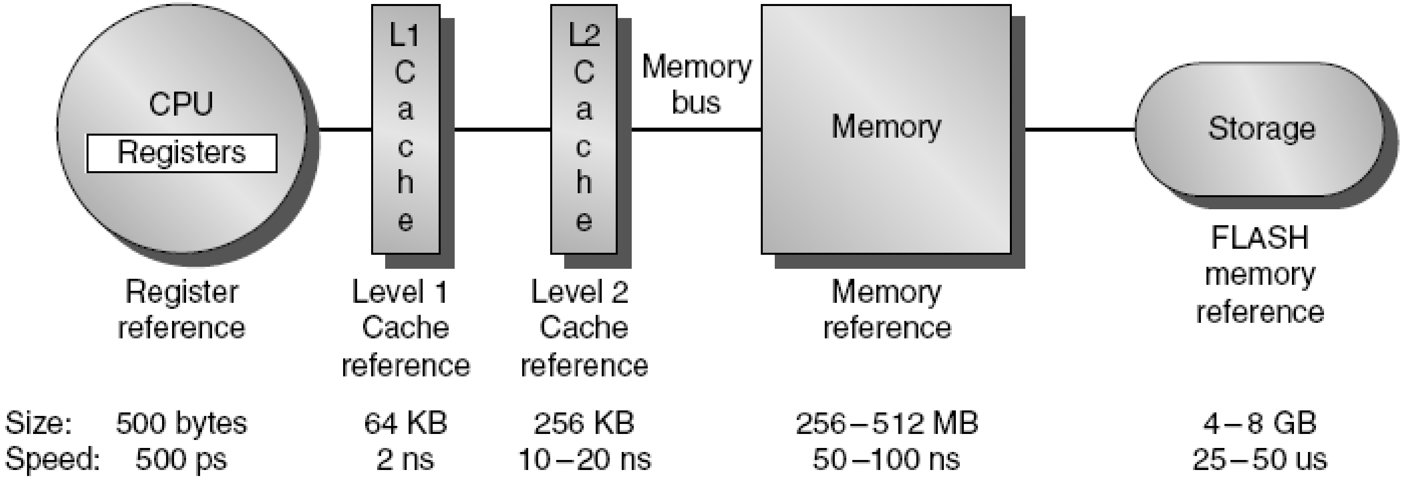
Memory

I/O bus

Disk storage

	Register reference	Level 1 Cache reference	Level 2 Cache reference	Level 3 Cache reference	Memory reference	Disk memory reference
Size:	1000 bytes	64 KB	256 KB	2–4 MB	4–16 GB	4–16 TB
Speed:	300 ps	1 ns	3–10 ns	10–20 ns	50–100 ns	5–10 ms

(a) Memory hierarchy for server



CPU

Registers

L1 Cache

L2 Cache

Memory bus

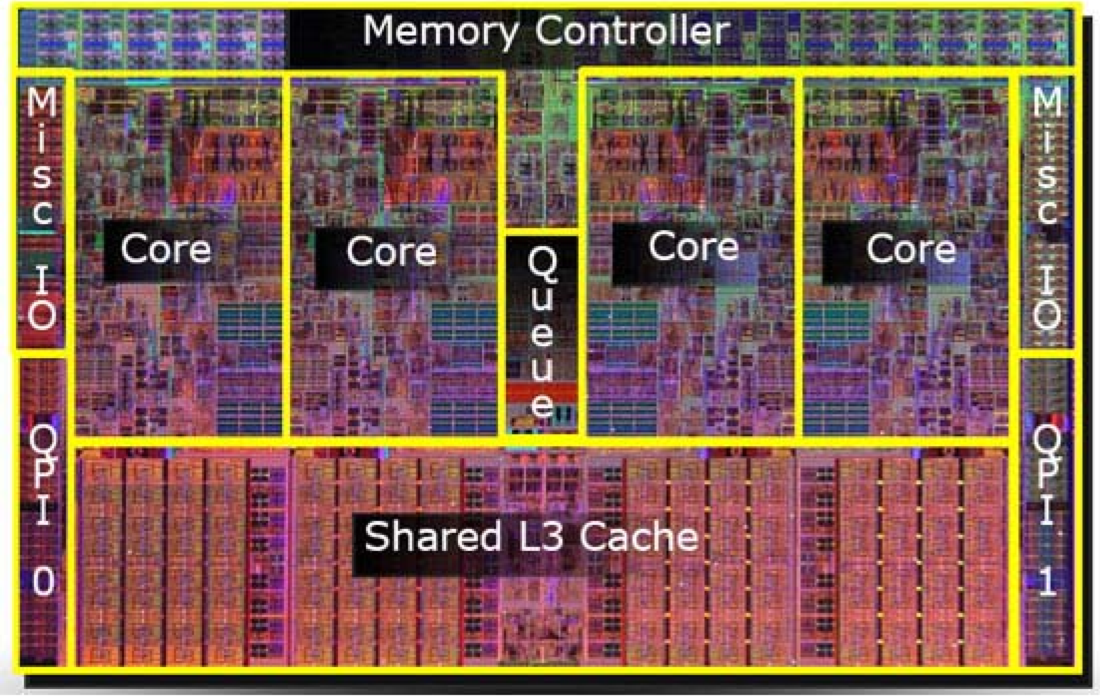
Memory

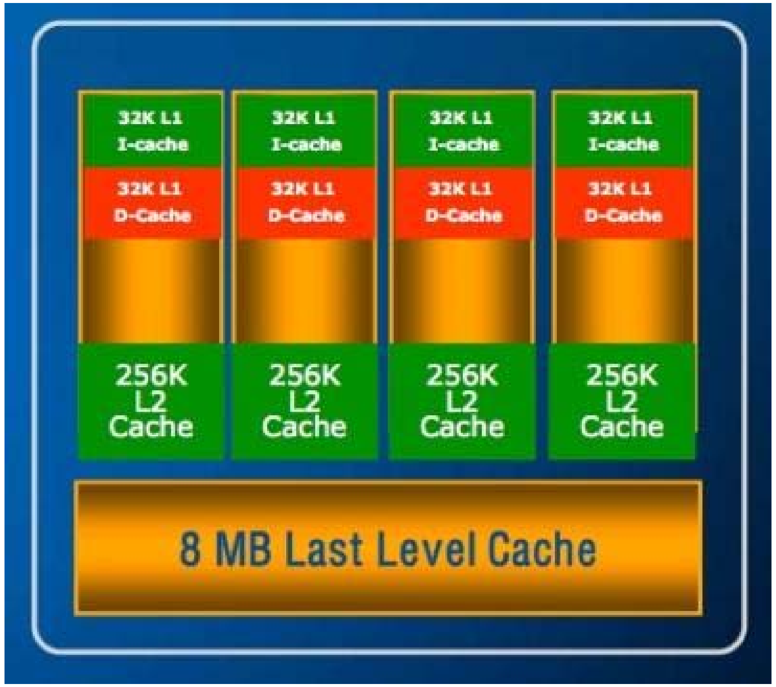
Storage

	Register reference	Level 1 Cache reference	Level 2 Cache reference	Memory reference	FLASH memory reference
Size:	500 bytes	64 KB	256 KB	256–512 MB	4–8 GB
Speed:	500 ps	2 ns	10–20 ns	50–100 ns	25–50 us

(b) Memory hierarchy for a personal mobile device

Nehalem Architecture: Intel Core i7





Levels of the memory hierarchy

Levels of Memory Hierarchy

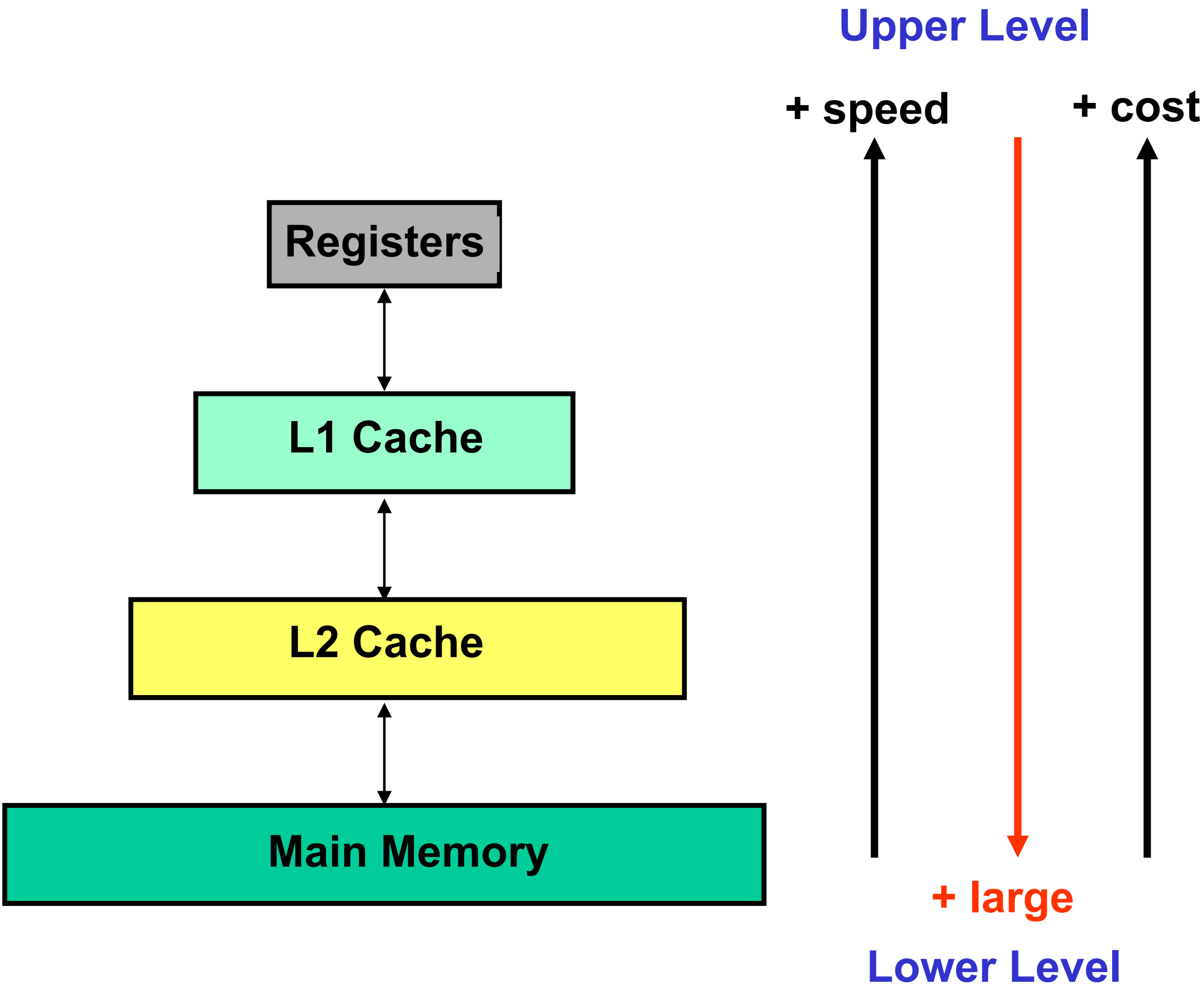
Size
Access time

CPU registers
< 1 KB
300 ps

L1 Cache (SRAM)
< 64 KB
1 ns

L2 Cache (SRAM)
< 256 KB
3 - 10 ns

Main Memory(DRAM)
4 – 16 GB,
50 – 100 ns



Basic Concepts

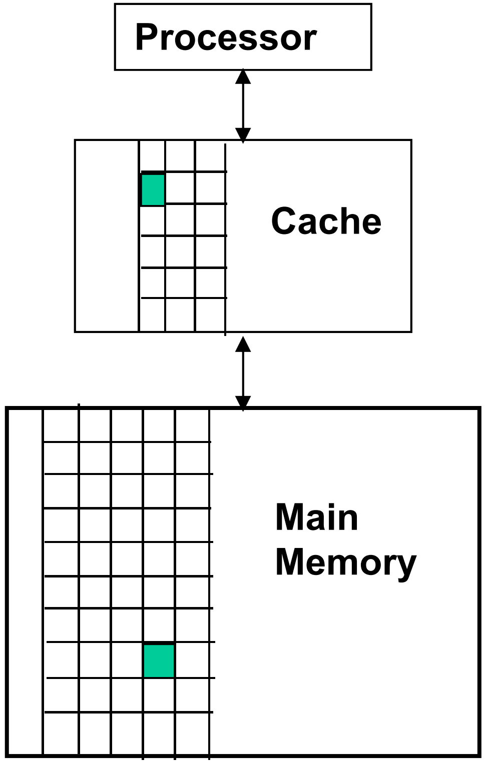
- The memory hierarchy is composed of several levels, but data are copied between two adjacent levels.
- Let us consider two levels: cache and main memory
- The cache (**upper** level) is smaller, faster and more expensive than the main memory (**lower** level).
- The minimum chunk of data that can be copied in the cache is the **block** or **cache line.**
- To exploit the spatial locality, the block size must be a **multiple** of the word size in memory
  - Example: 128-bit block size = 4 words of 32-bit
- The number of blocks in cache is given by:

  Number of cache blocks = Cache Size / Block Size

- Example: Cache size 64KByte; Block size 128-bit (16 Bytes)
  $\Rightarrow$ Number of cache blocks = 4 K blocks

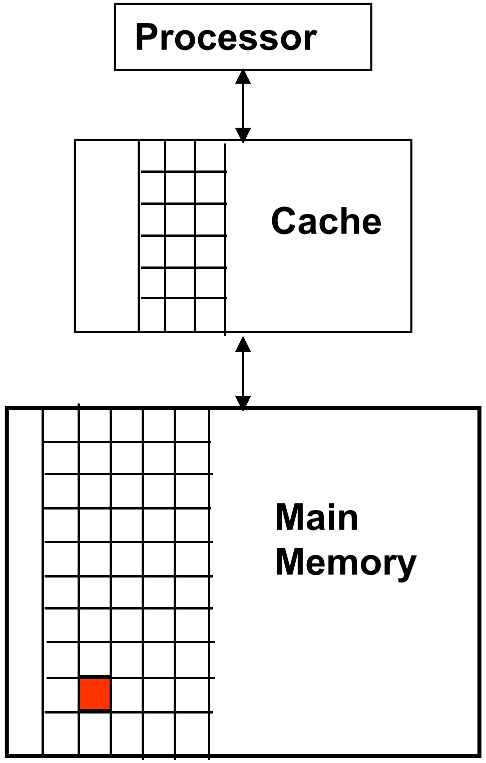
Cache Hit

- If the requested data is found in one of the cache blocks (upper level) ⇒ there is a ***hit*** in the cache access



Cache Miss

- If the requested data is not found in in one of the cache blocks (upper level) ⇒ there is a ***miss*** in the cache access
  ⇒ to find the block, we need to access the lower level of the memory hierarchy
- In case of a data miss, we need:
  - ***To stall*** the CPU;
  - To require to block from the main memory
  - To copy (write) the block in cache;
  - To repeat the cache access (hit).



Memory Hierarchy: Definitions

- Hit: data found in a block of the upper level
- **Hit Rate:** Number of memory accesses that find the data in the upper level with respect to the total number of memory accesses

$$\textbf{Hit Rate} = \frac{\textbf{\# hits}}{\textbf{\# memory accesses}}$$

- **Hit Time:** time to access the data in the upper level of the hierarchy, including the time needed to decide if the attempt of access will result in a hit or miss

Memory Hierarchy: Definitions

- Miss: the data must be taken from the lower level
- **Miss Rate:** number of memory accesses not finding the data in the upper level with respect to the total number of memory accesses

$$\textbf{Miss Rate} = \frac{\textbf{\# misses}}{\textbf{\# memory accesses}}$$

- By definition: **Hit Rate + Miss Rate = 1**
- **Miss Penalty :** time needed to access the lower level and to replace the block in the upper level
- **Miss Time = Hit Time + Miss Penalty**
- Typically: **Hit Time << Miss Penalty**

Average Memory Access Time

AMAT = Hit Rate * Hit Time + Miss Rate * Miss Time

where:

Miss Time = Hit Time + Miss Penalty

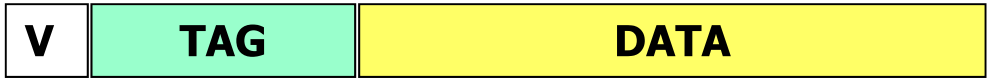
Hit Rate + Miss Rate = 1

⇒ **AMAT= Hit Time + Miss Rate * Miss Penalty**

Cache Structure

Each entry in the cache includes:

1. **Valid bit** to indicate if this position contains valid data or not. At the bootstrap, all the entries in the cache are marked as INVALID
2. **Cache Tag** contains the value that univocally identifies the memory address corresponding to the stored data.
3. **Cache Data** contains a copy of data (block or cache line)



Problem of block placement

- **Problem:** Given the address of the block in the main memory, where the block can be placed in the cache (upper level)?
- We need to find the correspondence between the memory address of the block and the cache address of the block
- The correspondence depends on the cache architecture:
  - **Direct Mapped Cache**
  - **Fully Associative Cache**
  - **n-way Set-Associative Cache**

Direct Mapped Cache

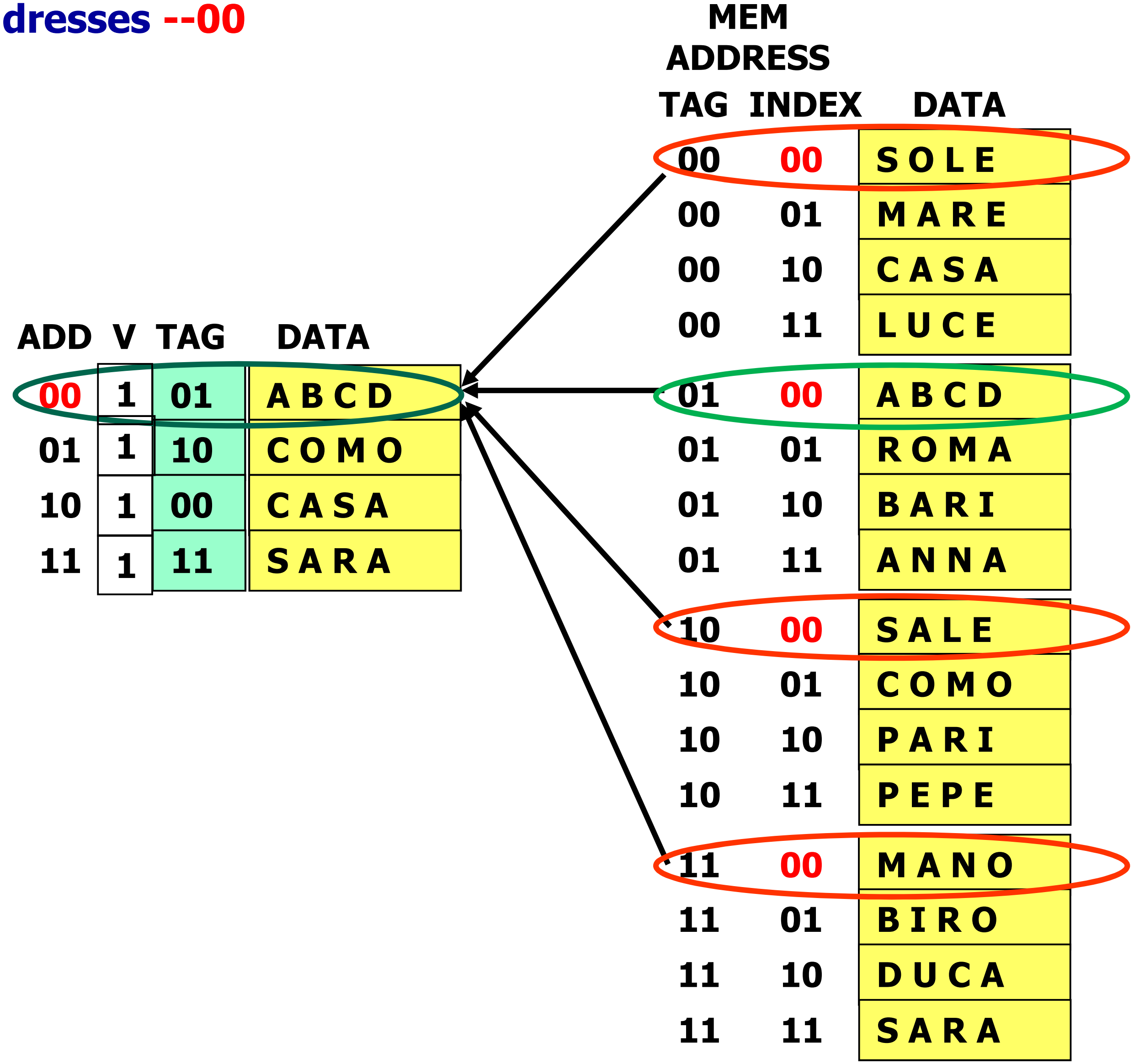
- Each memory location corresponds to one and only one cache location.
- The cache address of the block is given by:

$$\textbf{(Block Address)}_{\textbf{cache}} = \textbf{(Block Address)}_{\textbf{mem}} \textbf{ mod (Num. of Cache Blocks)}$$

Memory_Address

<table>
<tr><td colspan="2">Block_Address</td><td rowspan="2">Block Offset</td></tr>
<tr><td>Tag</td><td>Index</td></tr>
</table>

Example: Direct Mapped Cache with 4 words and memory of 16 words. The cache location 00 can be occupied by data coming from the memory addresses --00

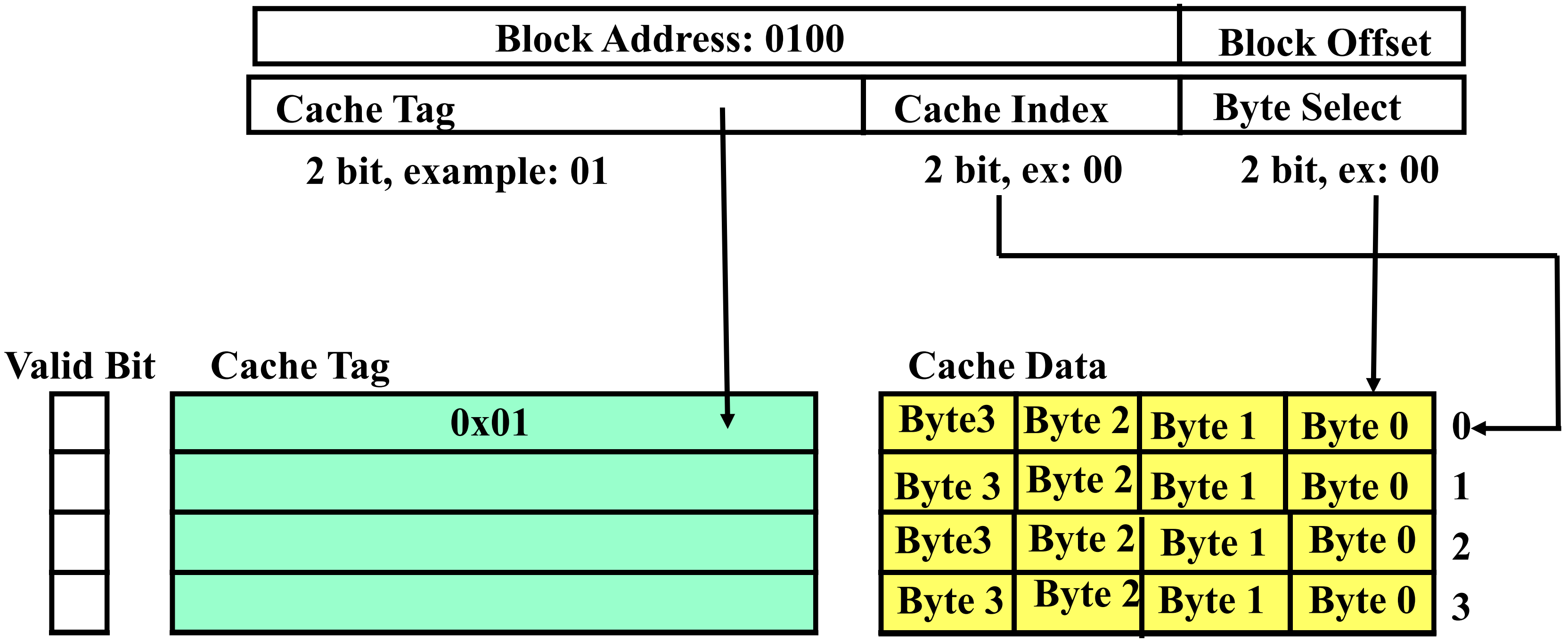


ADD	V	TAG	DATA
00	1	01	A B C D
01	1	10	C O M O
10	1	00	C A S A
11	1	11	S A R A

MEM ADDRESS

TAG	INDEX	DATA
00	00	S O L E
00	01	M A R E
00	10	C A S A
00	11	L U C E
01	00	A B C D
01	01	R O M A
01	10	B A R I
01	11	A N N A
10	00	S A L E
10	01	C O M O
10	10	P A R I
10	11	P E P E
11	00	M A N O
11	01	B I R O
11	10	D U C A
11	11	S A R A

Simple Direct Mapped Cache, 4 blocks



Block Address: 0100 | Block Offset

Cache Tag | Cache Index | Byte Select

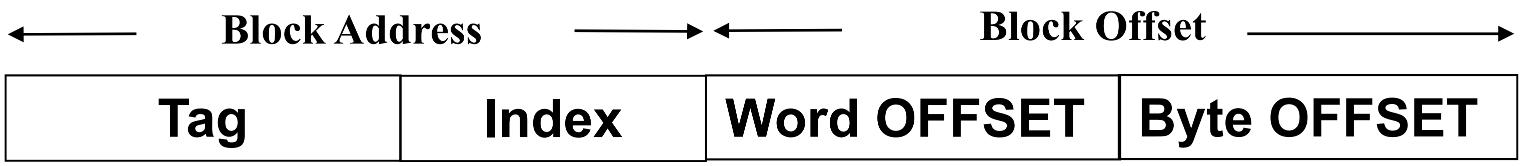
2 bit, example: 01

2 bit, ex: 00

2 bit, ex: 00

Valid Bit	Cache Tag	Cache Data				
	0x01	Byte3	Byte 2	Byte 1	Byte 0	0
		Byte 3	Byte 2	Byte 1	Byte 0	1
		Byte3	Byte 2	Byte 1	Byte 0	2
		Byte 3	Byte 2	Byte 1	Byte 0	3

Direct Mapped Cache: Addressing



Block Address

Block Offset

Tag	Index	Word OFFSET	Byte OFFSET

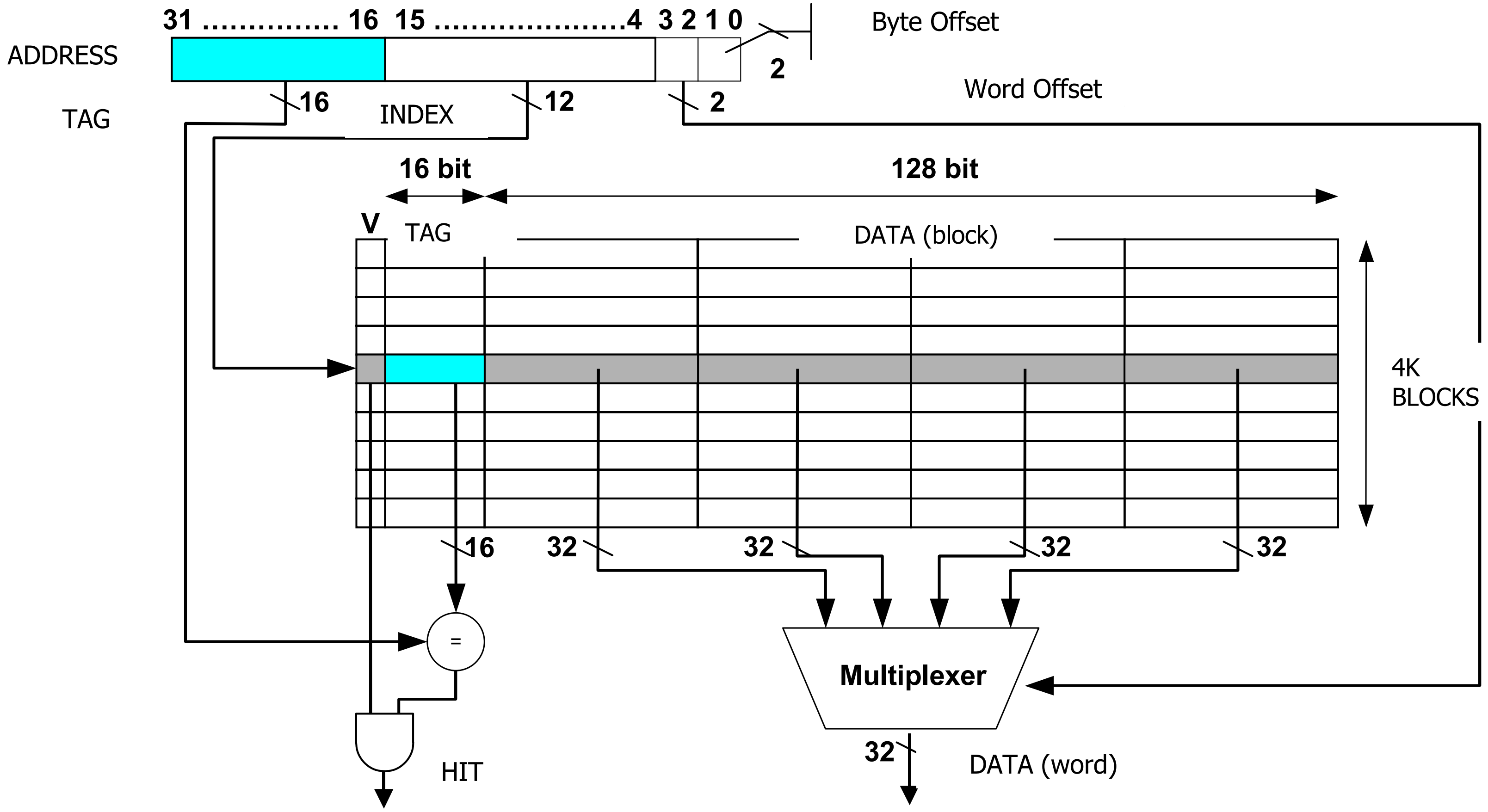
- Memory Address (**N** bit) composed of 4 fields:
  - **Byte offset** in the word: to identify the given byte within the word: **B** bit
    - If the memory is not byte addressable: B = 0.
  - **Word offset** in the block: to identify the given word within the block: **K** bit.
    - If the block contains only one word K = 0.
  - **Index** identifies the block : **M** bit
  - **Tag** to be compared to the cache tag associated to the block selected by the index : **N – (M + K + B)** bit

Direct Mapped Cache: Example

- Memory address composed of **N = 32 bit**
- Cache size **64 K Byte**
- Block size **128 bit**
- Number of blocks = Cache Size / Block Size = 64 K Byte / 16 Byte = **4 K blocks**
- Structure of the memory address:
  - **Byte Offset: B = 2 bit**
  - **Word Offset**: **K = 2 bit**
  - **Index**: **M = 12 bit**
  - **Tag**: **16 bit**

Tag (16 bit)	**Index (12 bit)**	**WO (2 bit)**	**BO (2 bit)**

Direct Mapped Cache: 64KByte cache size and 128bit block size



Fully associative cache

- In a *fully associative cache,* the memory block can be placed in any position of the cache

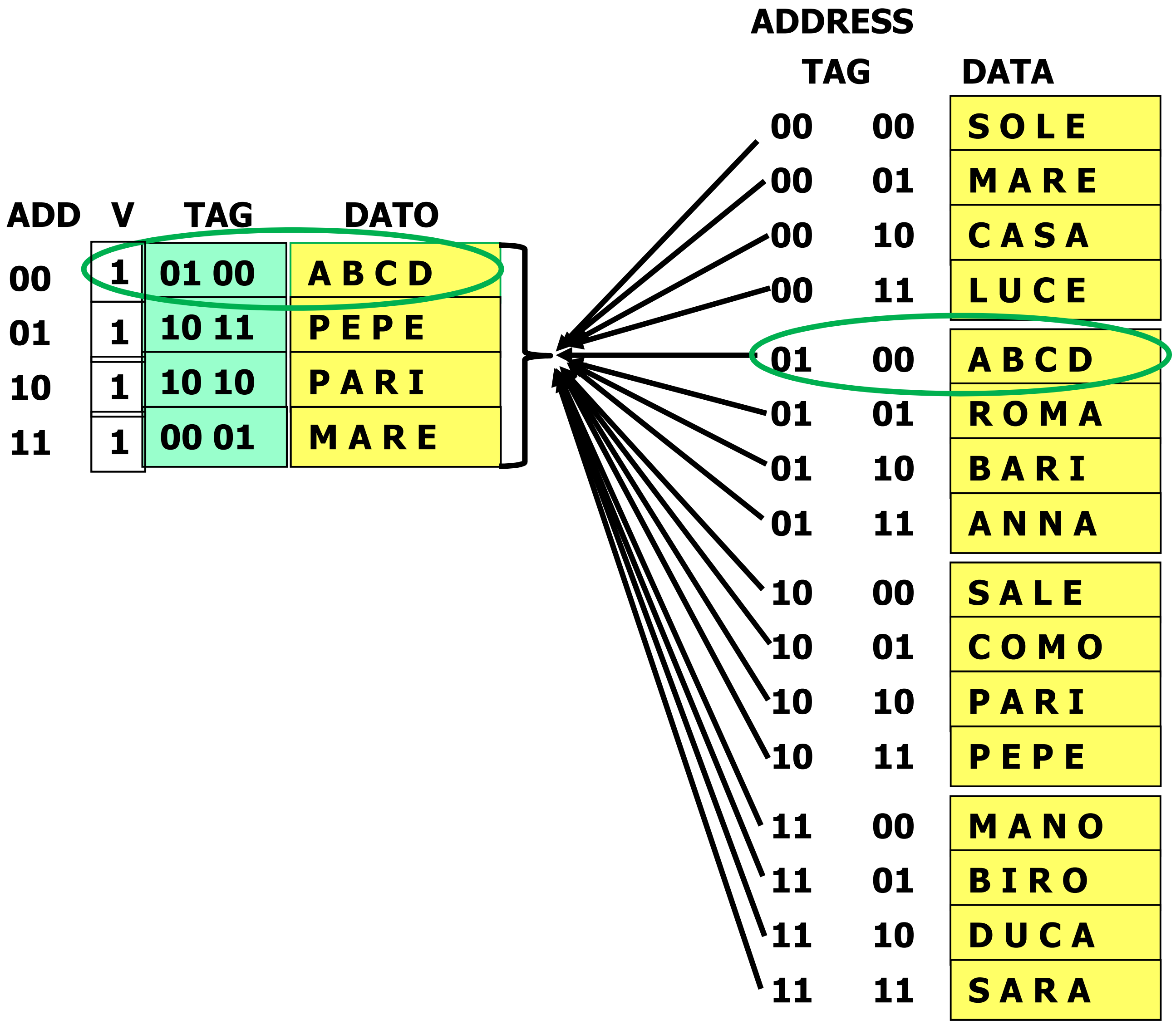
⇒ All the cache blocks must be checked during the search of the block

- The index does not exist in the memory address, there are the tag bits only

Number of blocks = Cache Size / Block Size

Tag (n-4 bit)	WO (2 bit)	BO (2 bit)

Example: Fully Associative Cache with 4 words and memory composed of 16 words. Any cache location can be occupied by data coming from any memory location



ADD	V	TAG	DATO
00	1	01 00	ABCD
01	1	10 11	PEPE
10	1	10 10	PARI
11	1	00 01	MARE

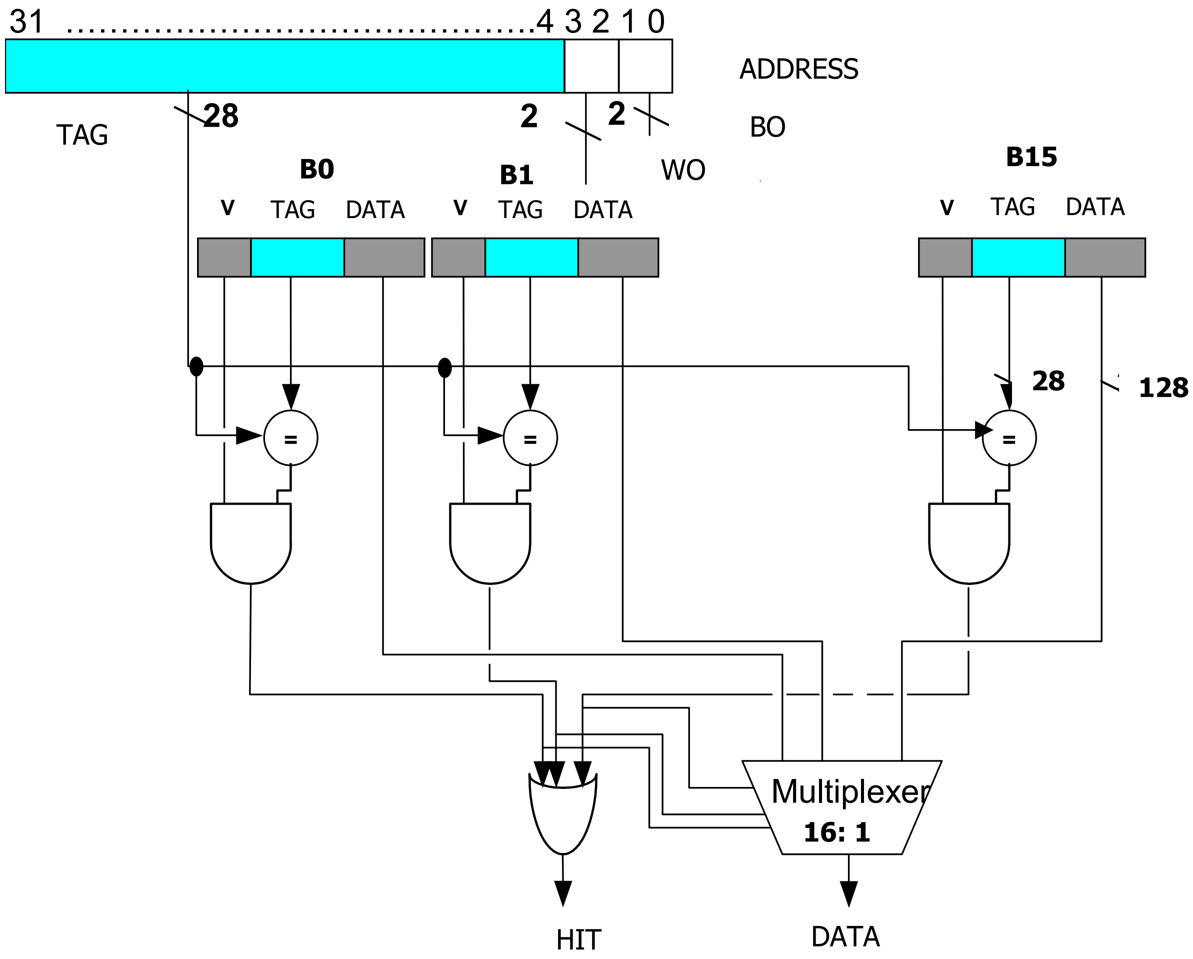
ADDRESS		
TAG		DATA
00	00	SOLE
00	01	MARE
00	10	CASA
00	11	LUCE
01	00	ABCD
01	01	ROMA
01	10	BARI
01	11	ANNA
10	00	SALE
10	01	COMO
10	10	PARI
10	11	PEPE
11	00	MANO
11	01	BIRO
11	10	DUCA
11	11	SARA

Fully Associative Cache: Example

- Memory address composed of **N = 32 bit**
- Cache size **256 Byte**
- Block size **128 bit = 16 B**
- Number of blocks = Cache Size / Block Size = 256 Byte / 16 Byte = **16 blocks**
- Structure of the memory address:
  - **Byte Offset: B = 2 bit**
  - **Word Offset**: **K = 2 bit**
  - **Tag**: **28 bit**

Tag (28 bit)	**WO (2 bit)**	**BO (2 bit)**

Fully Associative Cache:
256Byte Cache size and 16Byte block size



n-way Set Associative Cache

- Cache composed of sets, each set composed of *n* blocks:

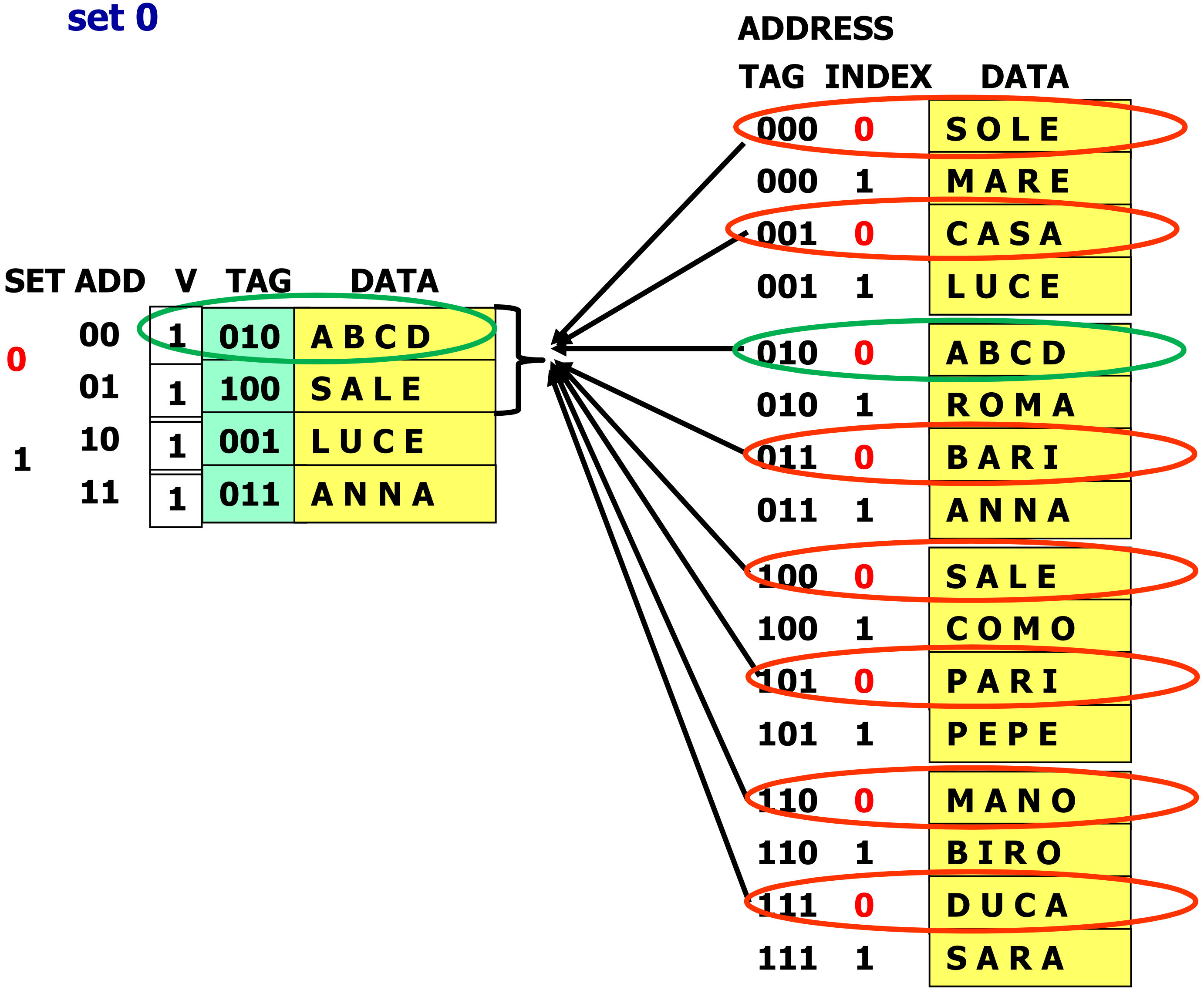
Number of blocks = Cache Size/ Block Size

Number of sets = Cache Size / (Block Size x n)

- The memory block can be placed in any block of the set ⇒ the search must be done on all the blocks of the set.
- Each memory block corresponds to a single set of the cache and the block can be placed in whatever block of the *n* blocks of the set

$$\textbf{(Set)}_{\textbf{cache}} = \textbf{(Block address)}_{\textbf{mem}} \textbf{ mod (Num. sets in cache)}$$

Example: 2-waySet Associative Cache with 4 words and memory composed of 16 words. The set 0 of the cache can be occupied by data coming from the memory addresses ---0. The block can be place in any of the 2 blocks of the set 0



SET	ADD	V	TAG	DATA
0	00	1	010	ABCD
	01	1	100	SALE
1	10	1	001	LUCE
	11	1	011	ANNA

ADDRESS		
TAG	INDEX	DATA
000	0	SOLE
000	1	MARE
001	0	CASA
001	1	LUCE
010	0	ABCD
010	1	ROMA
011	0	BARI
011	1	ANNA
100	0	SALE
100	1	COMO
101	0	PARI
101	1	PEPE
110	0	MANO
110	1	BIRO
111	0	DUCA
111	1	SARA

n-way Set Associative Cache: Addressing

Cache Structure

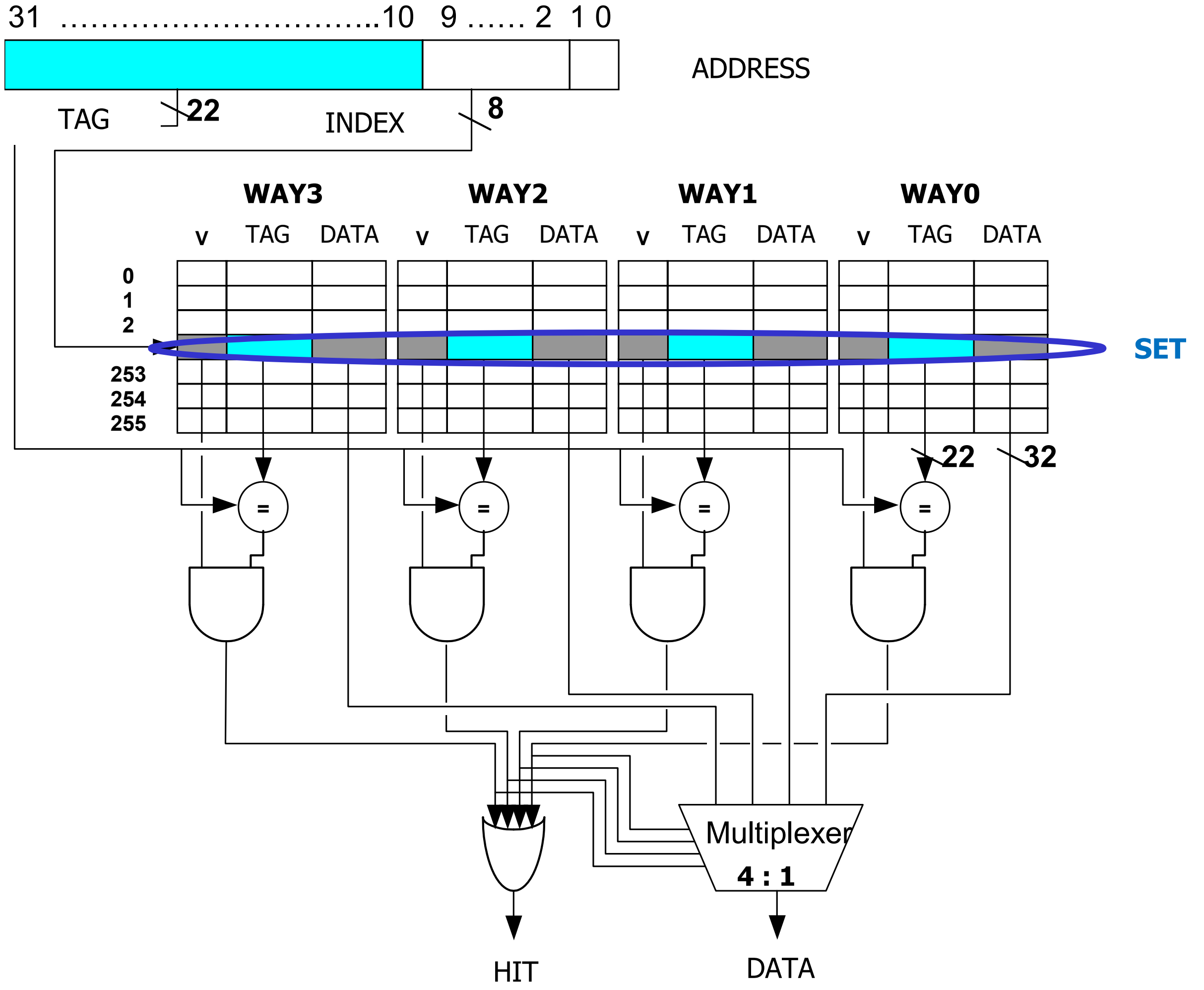
Tag	**Index**	**Word OFFSET**	**Byte OFFSET**

- **N** bit memory address composed of 4 fields:
  - **Byte offset: B bit**
  - **Word offset: K bit**
  - **Index** to identify the set: **M bit**
  - **Tag: N – (M + K + B) bit**

4-way Set Associative Cache: Example

- Memory address: **N = 32 bit**
- Cache size **4KByte**
- Block size **32 bit**
- Number of blocks = Cache Size / Block Size = 4 K Byte / 4 Byte = **1 K blocks**
- Number of sets = Cache Size/ (Block size x n)= 4 K Byte / (4 Byte x 4) = **256 sets**
- Structure of the memory address:
  - **Byte Offset: B = 2 bit**
  - **Word Offset: K = 0 bit**
  - **Index: M = 8 bit**
  - **Tag: 22 bit**

4-way Set Associative Cache: 4KB cache size and 32bit block size



Recap: Block Placement

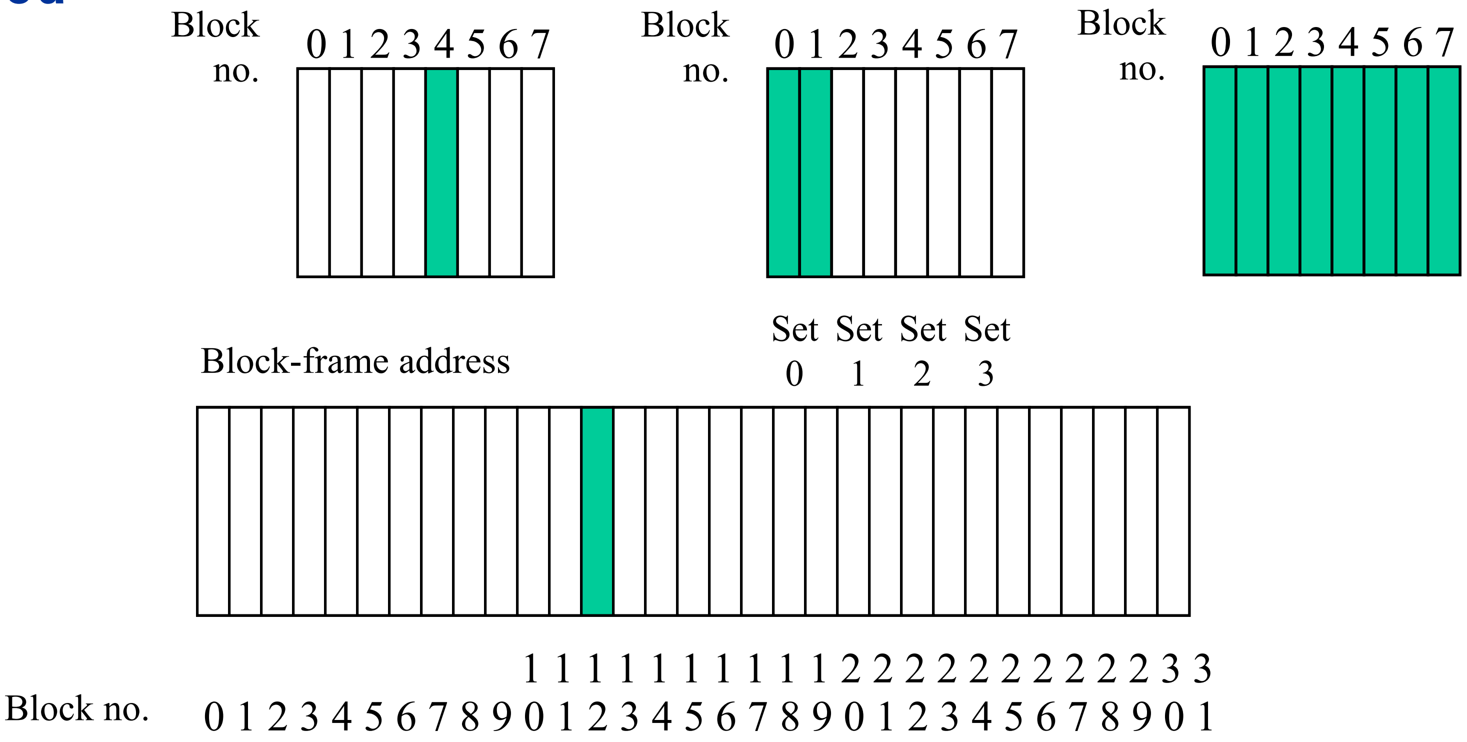
- *How can block 12 be placed in the 8-block cache?*

Block 12 can be placed

Direct Mapped: **only into block 4 (12 mod 8)**

2-way Set Associative: **anywhere in set 0 (12 mod 4)**

Fully Associative: **anywhere**



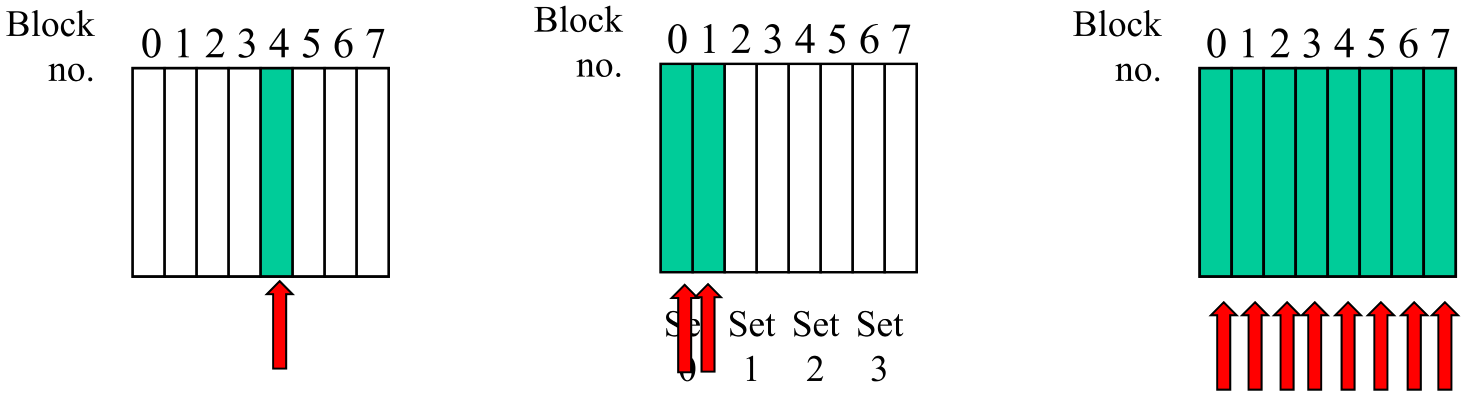
Recap: Block Identification

Block 12 can go

Direct Mapped: only into block 4 (12 mod 8)

2-way Set Associative: anywhere in set 0 (12 mod 4)

Fully Associative: anywhere



- **Direct Mapped:** Compute block position (Block Number MOD Number of blocks), compare tag of the block and verify valid bit
- **Set-associative:** Identify set (Block Number MOD Number of sets), compare tags of the set and verify valid bit
- **Fully Associative:** Compare tags in every block and verify valid bit

Recap: Increment of associativity

- Main advantage:
  - Reduction of miss rate
- Main disadvantages:
  - Higher implementation cost
  - Increment of hit time
- The choice among direct mapped, fully associative and set associative caches depends on the tradeoff between implementation cost of the associativity (both in time and added hardware) and the miss rate reduction
- *Increasing associativity shrinks index bits, expands tag bits*

Memory_Address

<table>
<tr><td colspan="2">Block_Address</td><td rowspan="2">Block Offset</td></tr>
<tr><td>Tag</td><td>Index</td></tr>
</table>

The problem of block replacement

- In case of a **miss** in a **fully associative cache**, we need to decide which block to replace: any block is a potential candidate for the replacement
- If the cache is **set-associative**, we need to select among the blocks in the given set.
- For a direct mapped cache, there is only one candidate to be replaced (no need of any block replacement strategy)
- Main strategies used to choose the block to be replaced in associative caches:
  - **Random (or pseudo random)**
  - **LRU (Least Recently Used)**
  - **FIFO (First In First Out)** – oldest block replaced

The problem of the write policy

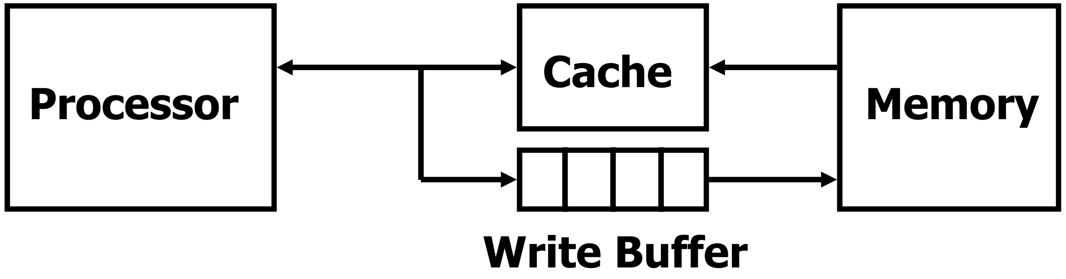
- **Write-Through:** the information is written to both the block in the cache and to the block in the lower level memory

- **Write-Back:** the information is written only to the block in the cache. The modified cache block is written to the lower level memory **only** when it is replaced due to a miss.
  Is a block clean or dirty? We need to add a **DIRTY bit**
  - At the end of the write in cache, the cache block becomes **dirty (modified)** and the main memory will contain a value different with respect to the value in the cache: the main memory and the cache are **not coherent**

Main advantages: Write-Back vs Write-Through

- **Write-Back:**
  - The block can be written by the processor at the frequency at which the cache, and not the main memory, can accept it
  - Multiple writes to the same block require only a single write to the main memory.
- **Write-Through:**
  - Simpler to be implemented, but to be effective it requires a **write buffer** to do not wait for the lower level of the memory hierarchy (to avoid write stalls)
  - The read misses are cheaper because they do not require any write to the lower level of the memory hierarchy
  - Memory always up to date

Write buffer



- **Basic idea:** Insert a FIFO buffer to do not wait for lower level memory access (typical number of entries: 4 to 8)
  - Processor writes data to the cache and to the write buffer
  - The memory controller writes the contents of the buffer to memory
- **Main problem:** Write buffer saturation
- Write through always combined with write buffer

Write Miss Options: Write Allocate vs No Write Allocate

- *What happens on a write miss?*
- **Write allocate** (or “fetch on write”): allocate new cache line in cache then write (double write to cache!)
  - Usually means that you have to do a “read miss” to fill in rest of the cache-line!
  - Alternative: per/word valid bits
- **No write allocate** (or “write-around”):
  - Simply send write data to lower-level memory - don’t allocate new cache line!

Write-Back vs Write-Through
Write Allocate vs No Write Allocate

- To manage a write miss, both options (Write Allocate and No Write Allocate) can be used for both write policies (Write-Back and Write-Through), but usually:

- **Write-Back** cache uses the **Write Allocate** option (hoping next writes to the block will be done again in cache)

- **Write-Through** cache uses the **No Write Allocate** option (hoping next writes to the block will be done again in memory)

Hit and Miss: Read and Write

- **Read Hit:**
  - Read data in cache
- **Read Miss:**
  - CPU stalls, data request to memory, copy in cache *(write in cache),* repeat of *cache read* operation

- **Write Hit:**
  - Write data both in cache and in memory **(write-through)**
  - Write data in cache only **(write-back):** memory copy only when it is replaced due to a miss
- **Write Miss:**
  - CPU stalls,
  - Data request to memory, copy in cache *(write in cache),* repeat of *cache write* operation **(write allocate)**
  - Simply send write data to lower-level memory **(no write allocate)**

Cache Structure

Recap: Four Questions about Memory Hierarchy

- Q1: Where can a block be placed in the upper level?
  (Block placement)

- Q2: How is a block found if it is in the upper level?
  (Block identification)

- Q3: Which block should be replaced on a miss?
  (Block replacement)

- Q4: What happens on a write?
  (Write strategy)

Q1: Where can a block be placed in the upper level?

Recap

- ***Block placement:***
  - Direct Mapped
  - Fully Associative
  - n-way Set Associative

Recap

Q2: How is a block found if it is in the upper level?

- ***Block identification:*** **compare tag(s)**
  - Fully Associative Mapping:
    - Compare tags in every block and verify valid bit
  - Direct Mapping:
    - Compute block position (Block Number MOD Number of blocks), compare tag of the block and verify valid bit
  - Set-associative Mapping:
    - Identify set (Block Number MOD Number of sets), compare tags of the set and verify valid bit
- No need to check index or block offset

Q3: Which block should be replaced on a miss?

- ***Block Replacement:***
  - Easy choice for direct mapped caches
  - Set associative or fully associative caches:
    - Random
    - LRU (Least Recently Used)
    - FIFO

Recap

Q4: What happens on a write?

- ***Write Policies:***
  - Write through
  - Write back

- ***Write Miss Options:***
  - Write Allocate
  - No Write Allocate

Performance Evaluation: Impact of memory hierarchy on CPU_{time}

$\mathbf{CPU}_{time}$ = (CPU exec cycles + Memory stall cycles) x T_{CLK}

where: T_{CLK} = T clock cycle time period

CPU exec cycles = IC x CPI_{exec}

IC = Instruction Count

(CPI_{exec} includes ALU and LD/STORE instructions)

Memory stall cycles = IC x Misses per instr x Miss Penalty

⇒ $\mathbf{CPU}_{time}$ = IC x (CPI_{exec} + Misses per instr x Miss penalty) x T_{CLK}

where:

Misses per instr = Memory Accesses Per Instruction x Miss rate

⇒ $\mathbf{CPU}_{time}$ = IC x (CPI_{exec} + MAPI x Miss rate x Miss penalty) x T_{CLK}

Performance evaluation: Impact of memory hierarchy on CPU_{time}

$\mathbf{CPU}_{\mathbf{time}} = \mathrm{IC} \times (\mathrm{CPI}_{exec} + \mathrm{MAPI} \times \text{Miss rate} \times \text{Miss penalty}) \times T_{CL}$

If considering also the ***stalls*** due to pipeline hazards:

$\mathbf{CPU}_{\mathbf{time}} = \mathrm{IC} \times (\mathrm{CPI}_{exec} + \textit{Stalls per instr} + \mathrm{MAPI} \times \text{Miss rate} \times \text{Miss penalty}) \times T_{CLK}$

Cache Performance Evaluation

Cache Performance

- Average Memory Access Time:

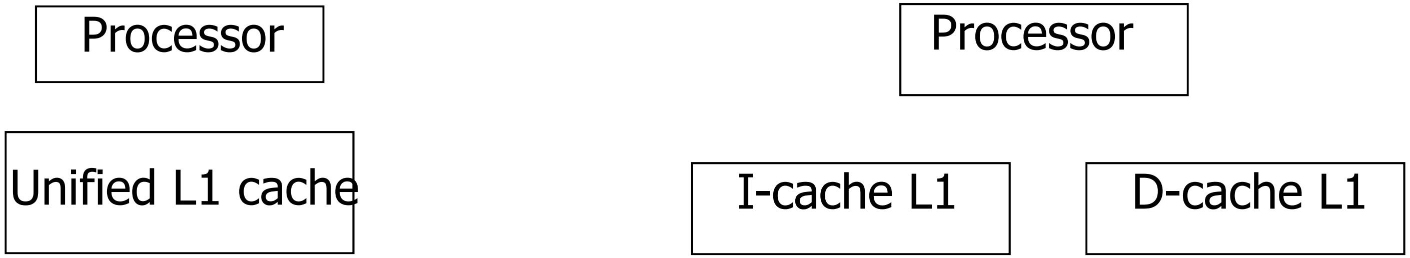
  AMAT= Hit Time + Miss Rate * Miss Penalty

- How to improve cache performance:
  1. Reduce the hit time
  2. Reduce the miss rate
  3. Reduce the miss penalty

Cache Performance Evaluation

Cache Performance Evaluation

Unified Cache vs Separate I$ & D$ (Harvard architecture)



To better exploit the locality principle

- Average Memory Access Time for Separate I$ & D$

  AMAT= % Instr. (Hit Time + I$ Miss Rate * Miss Penalty)
  + % Data (Hit Time + D$ Miss Rate * Miss Penalty)

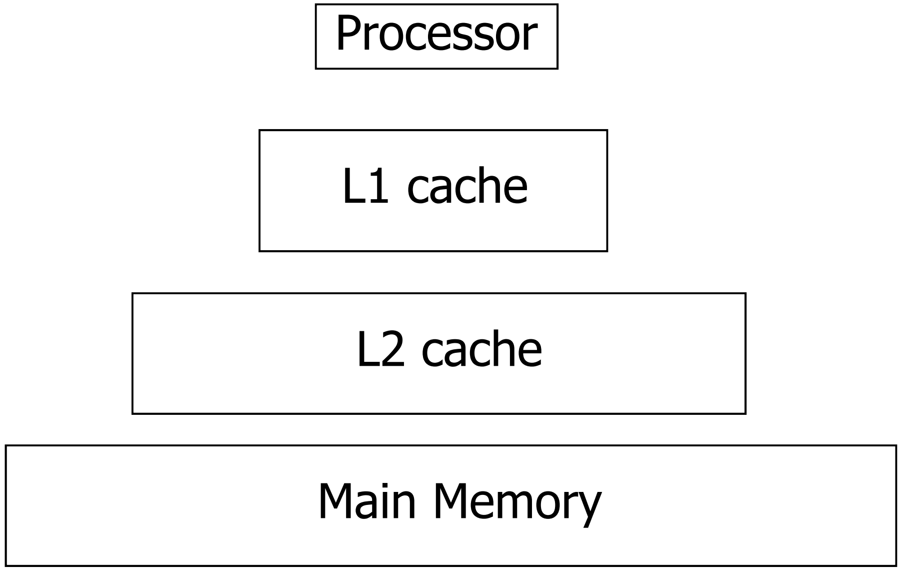
- Usually: I$ Miss Rate << D$ Miss Rate

Second Level Cache

Miss Penalty Reduction: Second Level Cache

Basic Idea:

- L1 cache small enough to match the fast CPU cycle time
- L2 cache large enough to capture many accesses that would go to main memory reducing the effective ***miss penalty***



Processor

L1 cache

L2 cache

Main Memory

Second Level Cache

AMAT for L1 and L2 Caches

AMAT = Hit Time$_{L1}$ + Miss Rate$_{L1}$ x Miss Penalty$_{L1}$

where:

Miss Penalty$_{L1}$ = Hit Time$_{L2}$ + Miss Rate$_{L2}$ x Miss Penalty$_{L2}$

$\Rightarrow$ **AMAT** = Hit Time$_{L1}$ + Miss Rate$_{L1}$ x (Hit Time$_{L2}$ + Miss Rate$_{L2}$ x Miss Penalty$_{L2}$)

Second Level Cache

Local and global miss rates

- Definitions:
  - ***Local miss rate:*** misses in this cache divided by the total number of memory accesses *to this cache*: Miss rate$_{L1}$ for L1 and Miss rate$_{L2}$ for L2
  - ***Global miss rate:*** misses in this cache divided by the total number of memory accesses generated by the CPU: for L1, the global miss rate is still just ***Miss Rate$_{L1}$,*** while for L2, it is ***(Miss Rate$_{L1}$ x Miss Rate$_{L2}$)***
  - **Global Miss Rate** is what really matters: it indicates what fraction of memory accesses from CPU go all the way to main memory

Example

- Let us consider a computer with a L1 cache and L2 cache memory hierarchy. Suppose that in 1000 memory references there are 40 misses in L1 and 20 misses in L2.
- What are the various miss rates?

Miss Rate $_{L1}$ = 40 /1000 = 4% (either local or global)

Miss Rate $_{L2}$ = 20 /40 = 50%

- Global Miss Rate for Last Level Cache (L2):

Miss Rate $_{L1\ L2}$ = Miss Rate$_{L1}$ x Miss Rate$_{L2}$=

(40 /1000) x (20 /40) = 2%

Second Level Cache

AMAT for L1 and L2 Caches

AMAT = Hit Time$_{L1}$ + Miss Rate$_{L1}$ x (Hit Time$_{L2}$ + Miss Rate$_{L2}$ x Miss Penalty$_{L2}$)

$\Rightarrow$ **AMAT** = Hit Time$_{L1}$ + Miss Rate$_{L1}$ x Hit Time$_{L2}$ + Miss Rate$_{L1\ L2}$ x Miss Penalty$_{L2}$

Memory stalls per instructions for L1 and L2 caches

- Average memory stalls per instructions:

Memory stall cycles per instr = Misses per instr x Miss Penalty

- Average memory stalls per instructions for L1 and L2 caches:

Memory stall cycles per instr = Misses$_{L1}$ per instr X Hit Time$_{L2}$ + Misses$_{L2}$ per instr X Miss Penalty$_{L2}$

Impact of L1 and L2 on CPU_{time}

$\textbf{CPU}_{\textbf{time}}$ = IC x (CPI_{exec} + Memory stall cycles per instr) x T_{CLK}

where:

Memory stall cycles per instr =
  $\text{Misses}_{\text{L1}}$ per instr X Hit Time_{L2} +
  $\text{Misses}_{\text{L2}}$ per instr X Miss $\text{Penalty}_{\text{L2}}$

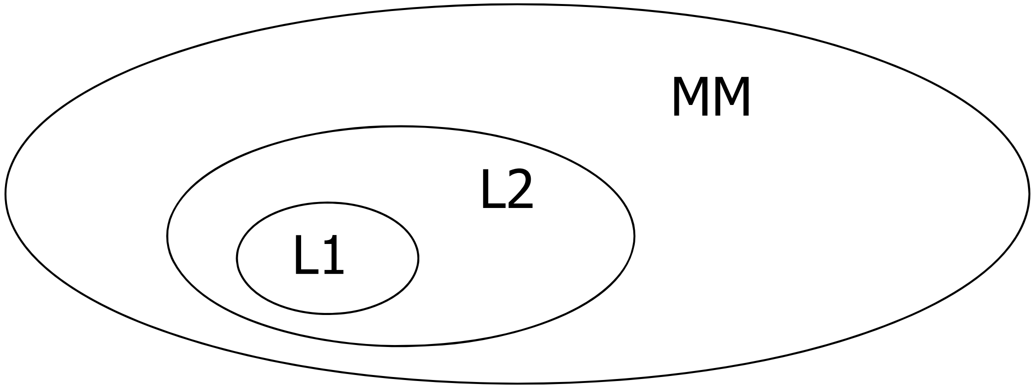
$\text{Misses}_{\text{L1}}$ per instr = Memory Accesses Per Instr x Miss rate_{L1}

$\text{Misses}_{\text{L2}}$ per instr = Memory Accesses Per Instr x Miss rate $_{\text{L1 L2}}$

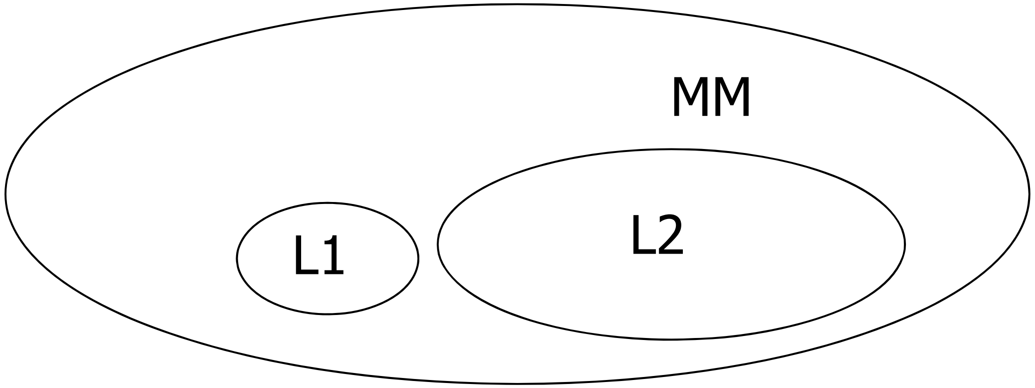
$\Rightarrow$ $\textbf{CPU}_{\textbf{time}}$ = IC x (CPI_{exec} + MAPI x MR_{L1} x HT_{L2} + MAPI x $\text{MR}_{\text{L1 L2}}$ x MP_{L2}) x T_{CLK}

L1 and L2 caches

- Multi-level Inclusion:



- Multi-Level Exclusion:

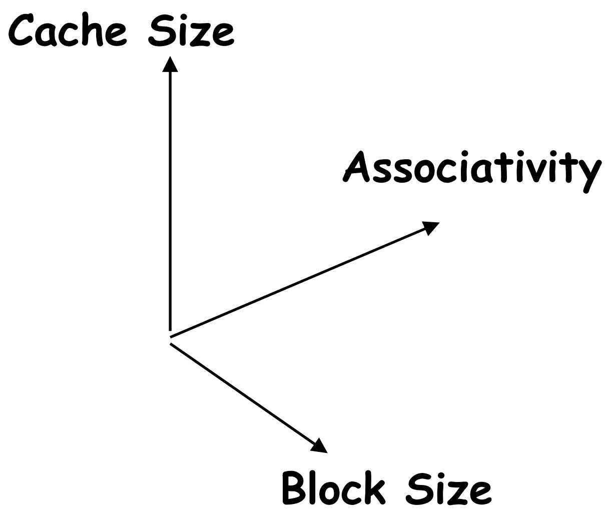


Memory Hierarchy: Summary

- Cache: small, fast storage used to improve average access time to slow lower level memory.
- Cache exploits spatial and temporal locality:
  - Present to the user as much memory it is available at the cheapest technology.
  - Provide access at the speed offered by the fastest technology.

Summary: the cache design space

- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back



Memory Technology

- Performance metrics
  - Latency is concern of cache
  - Bandwidth is concern of multiprocessors and I/O
  - Access time
    - Time between read request and when desired word arrives
  - Cycle time
    - Minimum time between unrelated requests to memory
- **DRAM used for main memory**
- **SRAM used for cache**

Memory Technology

- **SRAM**
  - Requires low power to retain bit
  - Requires **6 transistors/bit**
- **DRAM**
  - Must be re-written after being read
  - Must also be periodically **refreshed**
    - Every ~ 8 ms
    - Each row can be refreshed simultaneously
  - Requires **1 transistor/bit**
  - Address lines are multiplexed:
    - Upper half of address: row access strobe (RAS)
    - Lower half of address: column access strobe (CAS)

Memory Technology

- Amdahl law:
  - Memory capacity should grow linearly with processor speed
  - Unfortunately, memory capacity and speed has not kept pace with processors

- Some optimizations:
  - Multiple accesses to same row
  - Synchronous DRAM
    - Added clock to DRAM interface
    - Burst mode with critical word first
  - Wider interfaces
  - Double data rate (DDR)
  - Multiple banks on each DRAM device

Memory Optimizations

- DDR:
  - DDR2
    - Lower power (2.5 V -> 1.8 V)
    - Higher clock rates (266 MHz, 333 MHz, 400 MHz)
  - DDR3
    - 1.5 V
    - 800 MHz
  - DDR4
    - 1-1.2 V
    - 1600 MHz
- GDDR5 is graphics memory based on DDR3

Memory Optimizations

- Graphics memory:
  - Achieve 2-5 X bandwidth per DRAM vs. DDR3
    - Wider interfaces (32 vs. 16 bit)
    - Higher clock rate
      - Possible because they are attached via soldering instead of socketted DIMM modules

- Reducing power in SDRAMs:
  - Lower voltage
  - Low power mode (ignores clock, continues to refresh)

Flash Memory

- Type of EEPROM
- Must be erased (in blocks) before being overwritten
- Non volatile
- Limited number of write cycles
- Cheaper than SDRAM, more expensive than disk
- Slower than SRAM, faster than disk

Memory Dependability

- Memory is susceptible to cosmic rays
- *Soft errors:* dynamic errors
  - Detected and fixed by error correcting codes (ECC)
- *Hard errors:* permanent errors
  - Use sparse rows to replace defective rows

- Chipkill: a RAID-like error recovery technique