*Course on: "Advanced Computer Architectures"*

# Introduction to cache memories


Prof. Cristina Silvano, Politecnico di Milano
cristina.silvano@polimi.it

# Summary

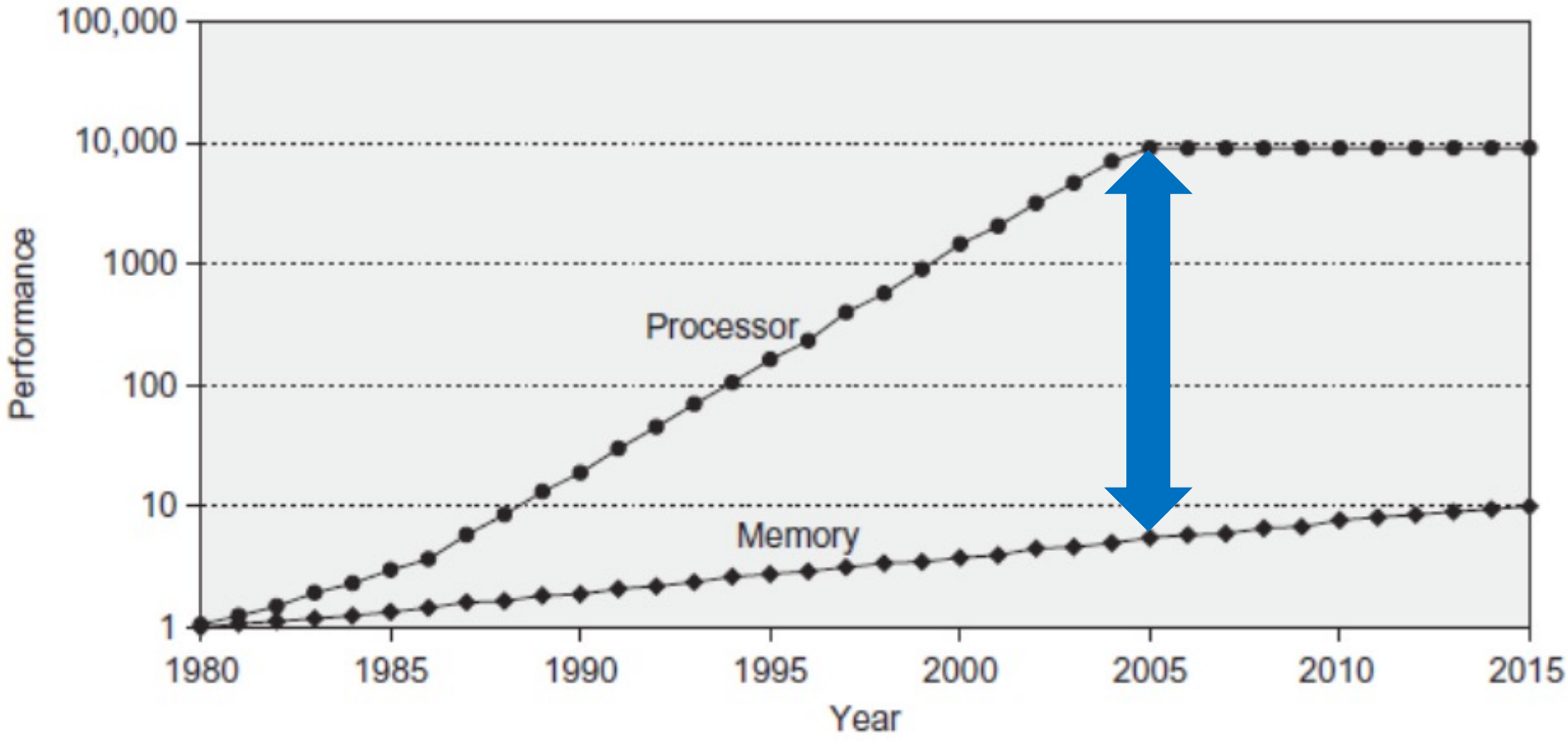
- Main goal
- Spatial and temporal locality
- Levels of memory hierarchy
- Cache memories: basic concepts
- Architecture of a cache memory:
  - Direct Mapped Cache
  - Fully associative Cache
  - N-way Set-Associative Cache
- Performance Evaluation
- Memory Technology

# Main goal

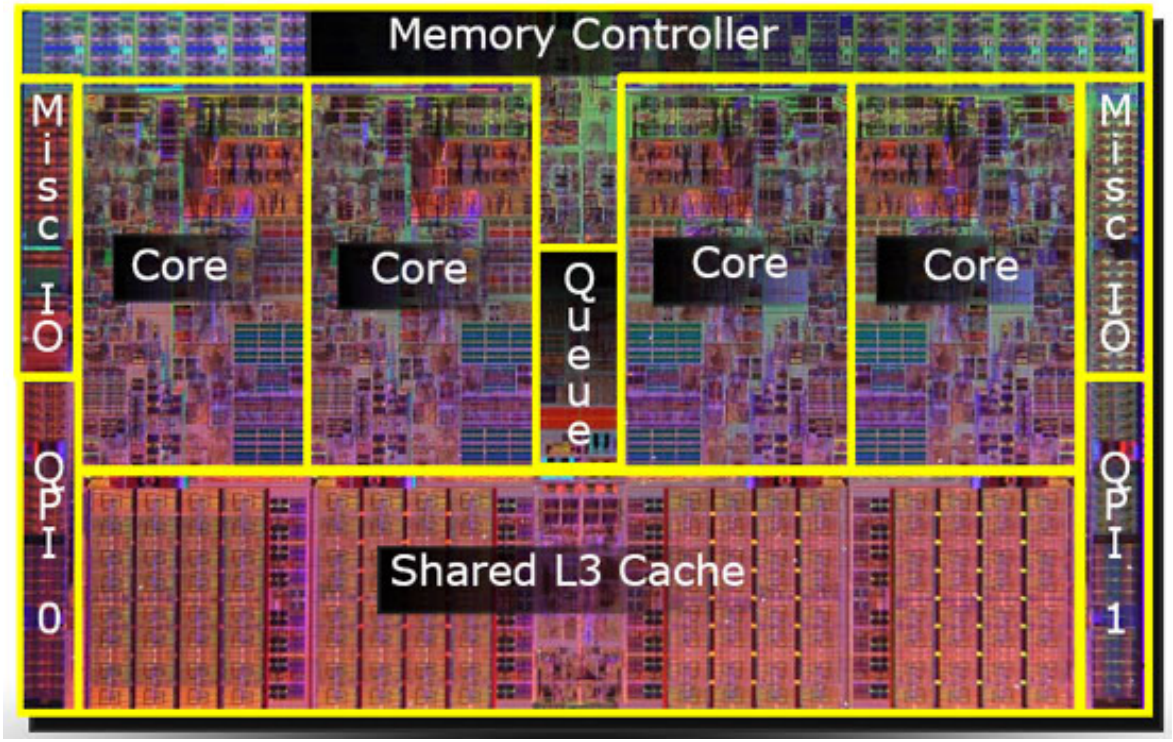
To increase the performance of a computer through the memory system in order to:

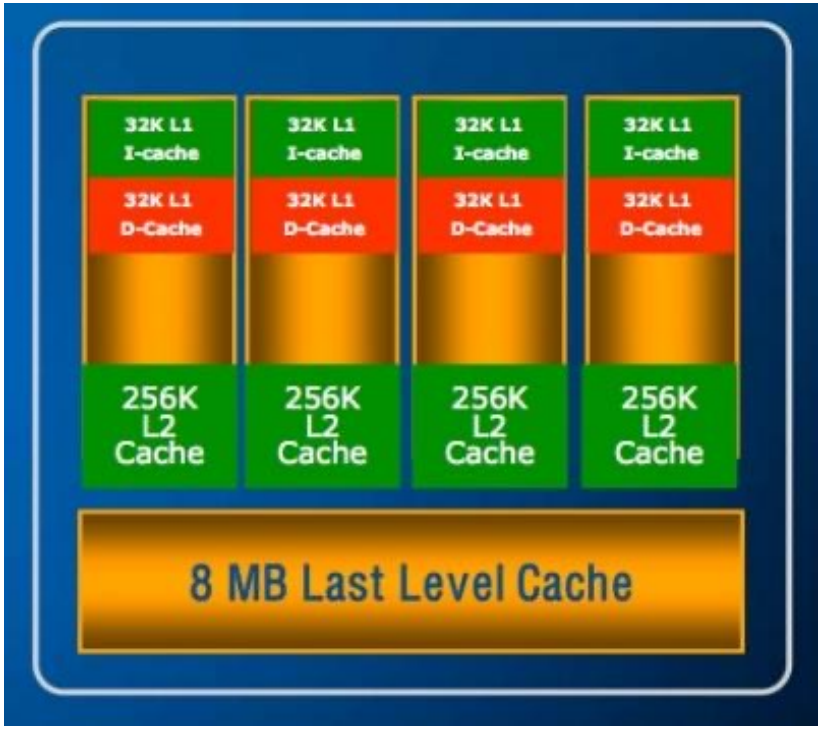
- Provide the user the illusion to use a memory that is simultaneously large and fast
- Provide the data to the processor at high frequency

# Problem: Processor-memory performance gap



# Nehalem Architecture: Intel Core i7




Memory hierarchy design became more crucial with multi-core processors

# Memory Hierarchy Design

- Memory hierarchy design becomes more crucial with multi-core processors:
  - Aggregate peak bandwidth grows with # cores:
    - Intel Core i7 can generate two memory references per core per clock
    - Four cores and 3.2 GHz clock

      25.6 billion 64-bit data references/second +

      12.8 billion 128-bit instruction references

      = 409.6 GB/s!
    - DRAM bandwidth is only 6% of this (25 GB/s)
    - Requires:
      - Multi-port, pipelined caches
      - Two levels of cache per core
      - Shared third-level cache on chip

# 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)

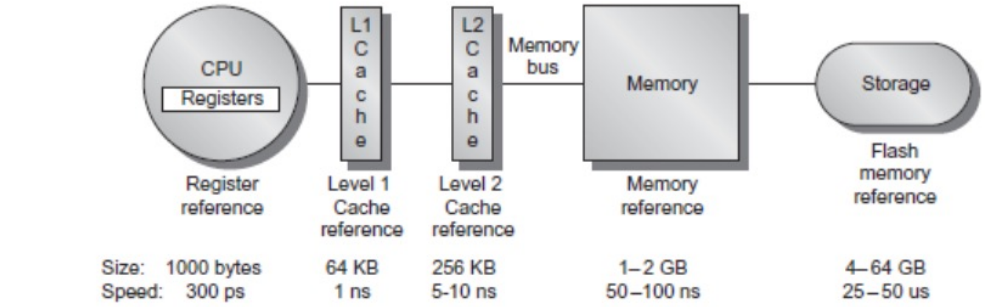
- **Spatial Locality:** when there is a reference to one memory element, the trend is to refer soon at other memory elements whose addresses are close by
  (i.e., sequence of instructions or accesses to data organized as arrays or matrices)

# Caches

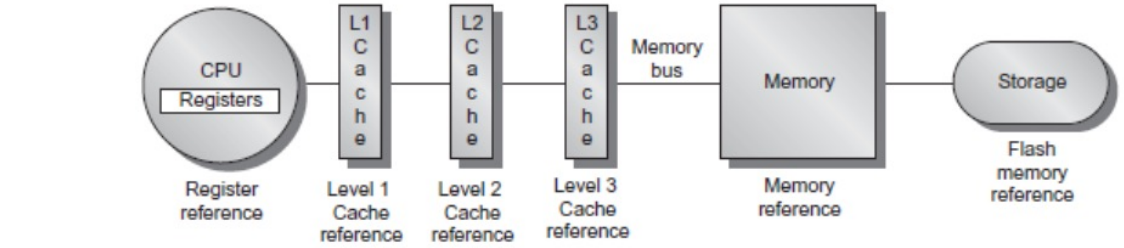
- Caches exploit both types of predictability:
  - Exploit **temporal locality** by keeping the contents of recently accessed memory locations.
  - Exploit **spatial locality** by fetching blocks of data around recently accessed memory locations.

# Levels of Memory Hierarchy

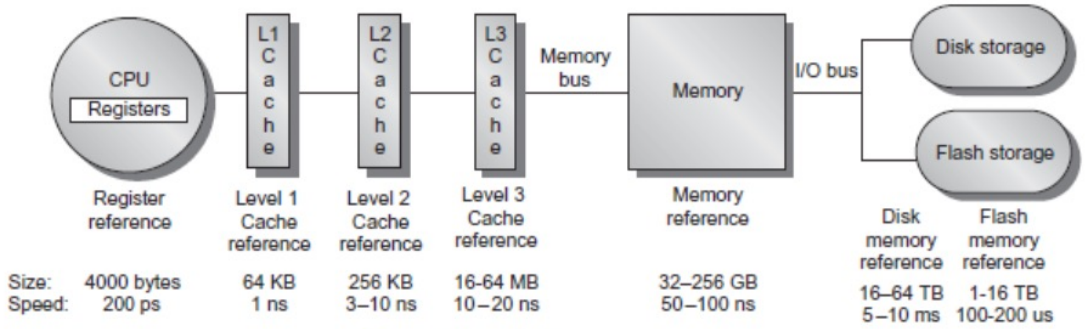





(A) Memory hierarchy for a personal mobile device




(B) Memory hierarchy for a laptop or a desktop




(C) Memory hierarchy for server

# Basic Concepts

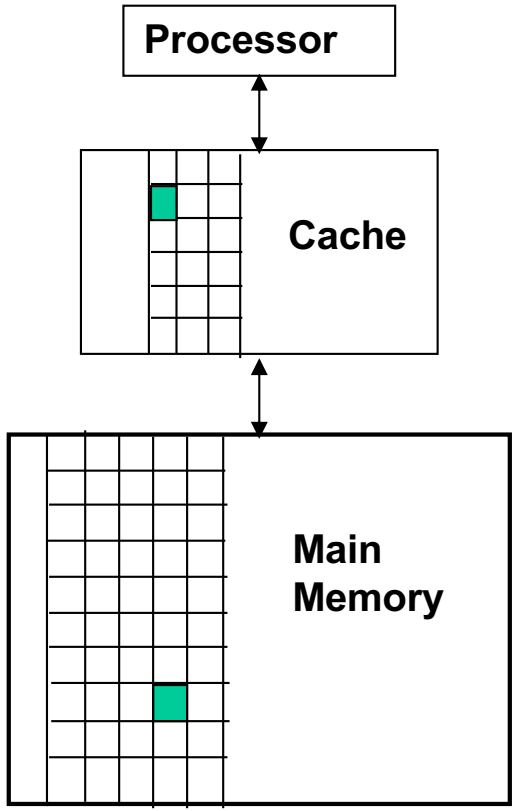
- In general, the memory hierarchy is composed of several levels.
- Let us consider 2 levels: cache and main memory
- The cache *(upper level)* is smaller, faster and more expensive than the main memory *(lower level).*
- The minimum chunk of data that can be copied in the cache is the **block** or **cache 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 64K-Byte; Block size 128-bit (16-Byte)

  $\Rightarrow$ Number of cache blocks = 64K-Byte / 16-Byte = 4K blocks

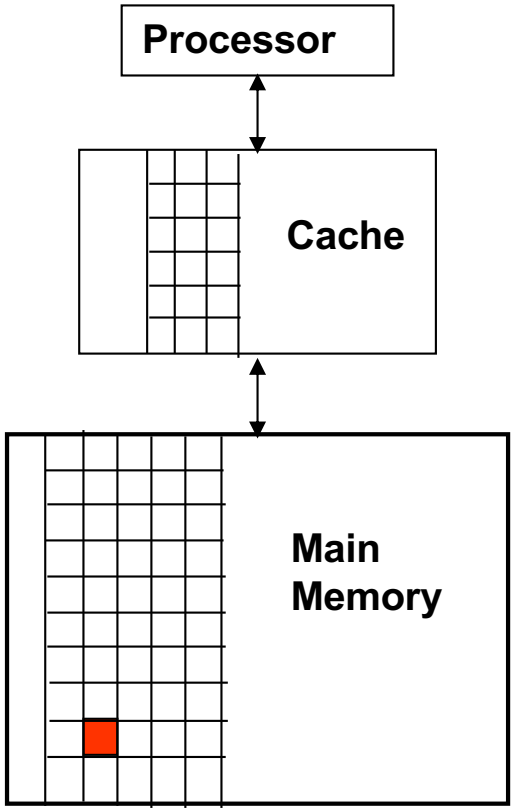
# Cache Hit

- 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

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

- **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 Time = Hit Time + Miss Penalty**

  Where the **Miss Penalty** is the time needed to access the lower level and to replace the block in the upper level

  Typically, we have that: **Hit Time << Miss Penalty**

# Average Memory Access Time

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

Being: **Miss Time = Hit Time + Miss Penalty**

=> **AMAT** = Hit Rate * Hit Time + Miss Rate * **(Hit Time + Miss Penalty**

$\Rightarrow$ **AMAT** = (Hit Rate + Miss Rate) * Hit Time + Miss Rate * Miss Penalty

By definition: Hit Rate + Miss Rate = 1

$\Rightarrow$ **AMAT= Hit Time + Miss Rate * Miss Penalty**

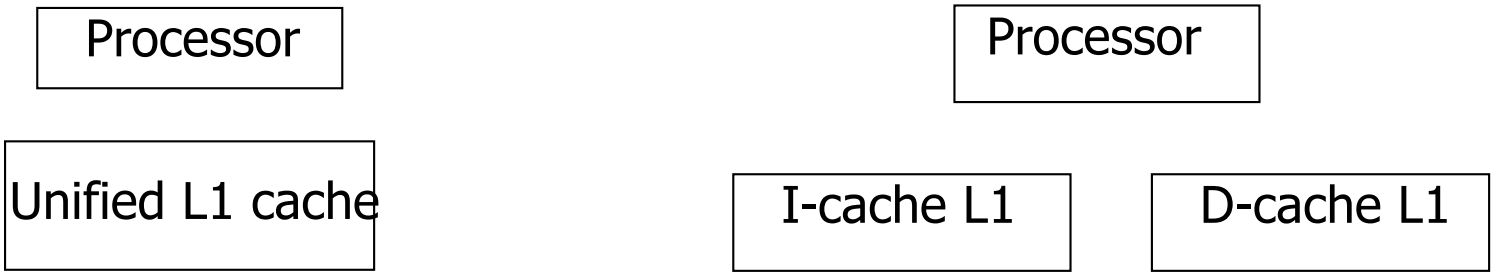
# Recap on Cache Performance

- **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

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




*To better exploit the locality principle*

- For **separate I$ & D$ (Harvard architecture):**

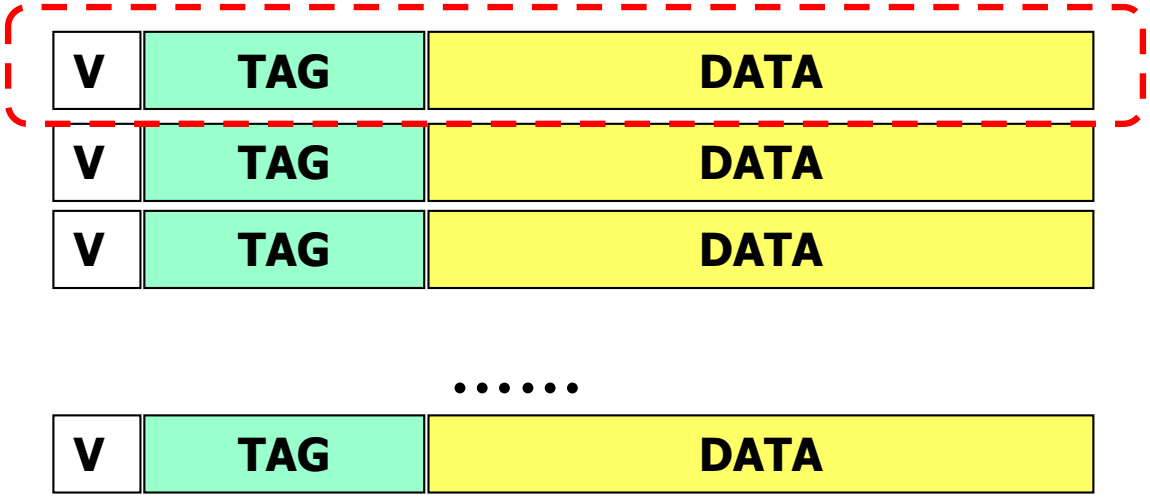
  $\mathbf{AMAT_{Harvard}}$ = % Instr. (Hit Time + Miss Rate I$ * Miss Penalty) +
  % Data (Hit Time + Miss Rate D$* Miss Penalty)

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

# Cache Structure

**Each entry (cache line)** 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(s)** 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)



# 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)***

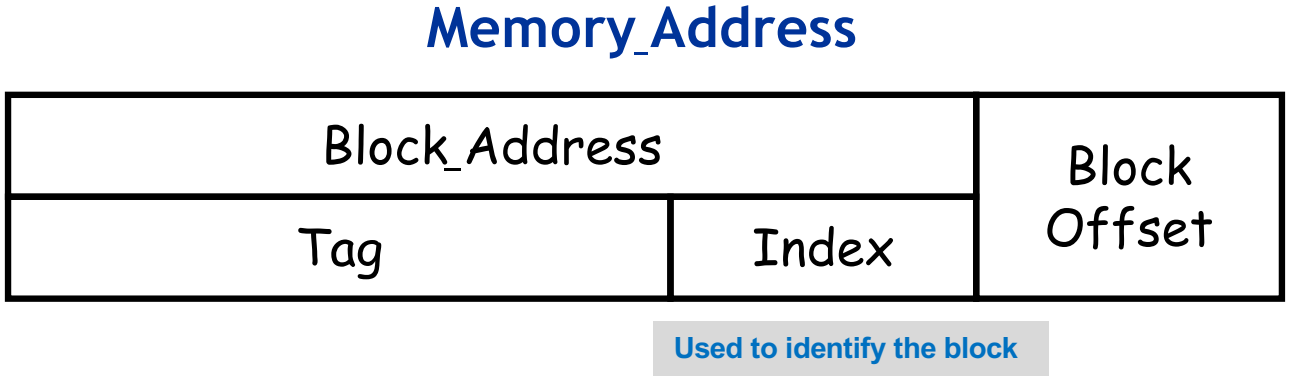
# Q1: Where can a block be placed in the upper level?

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

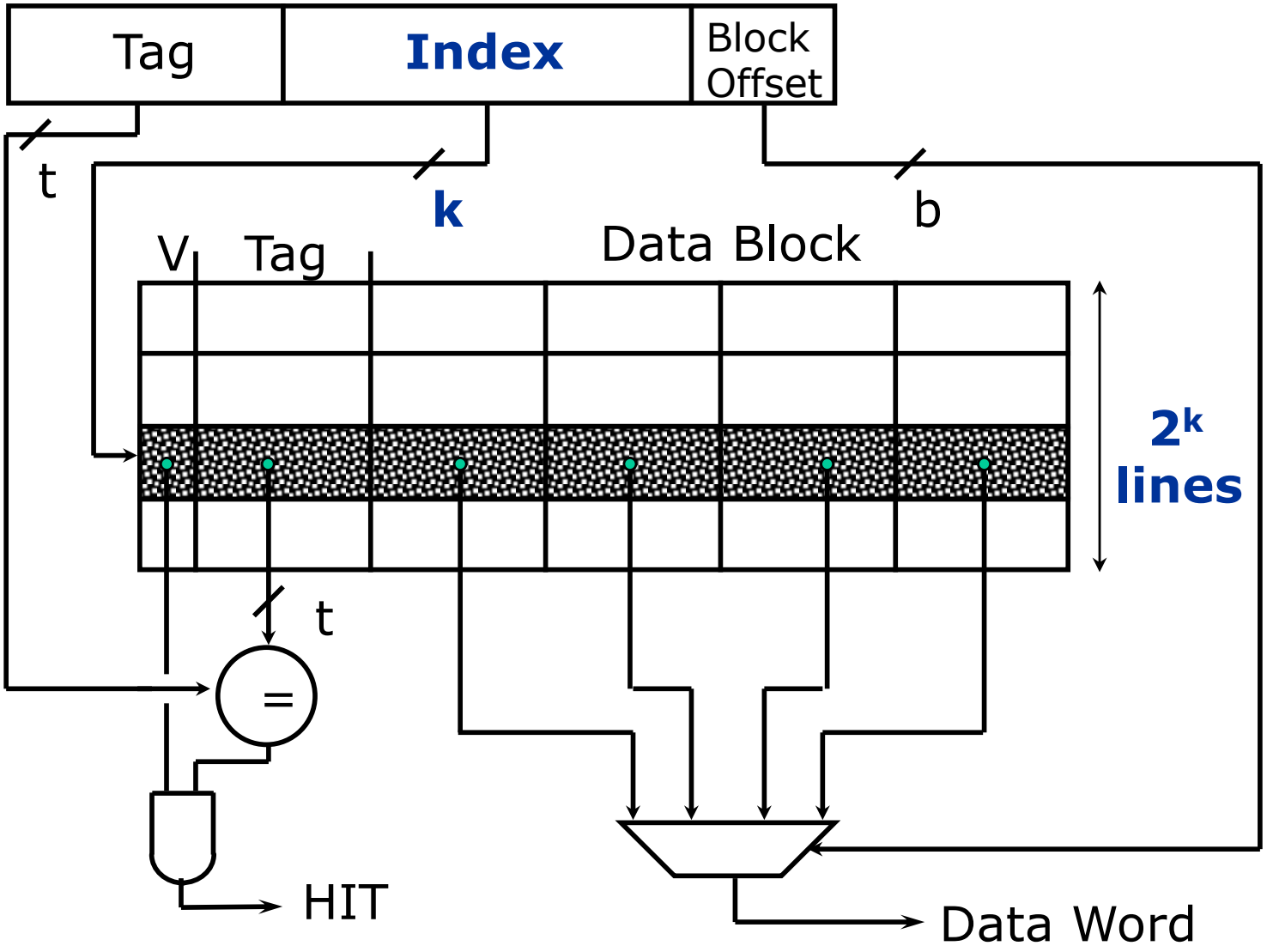
# Direct Mapped Cache

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

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



# Direct-Mapped Cache

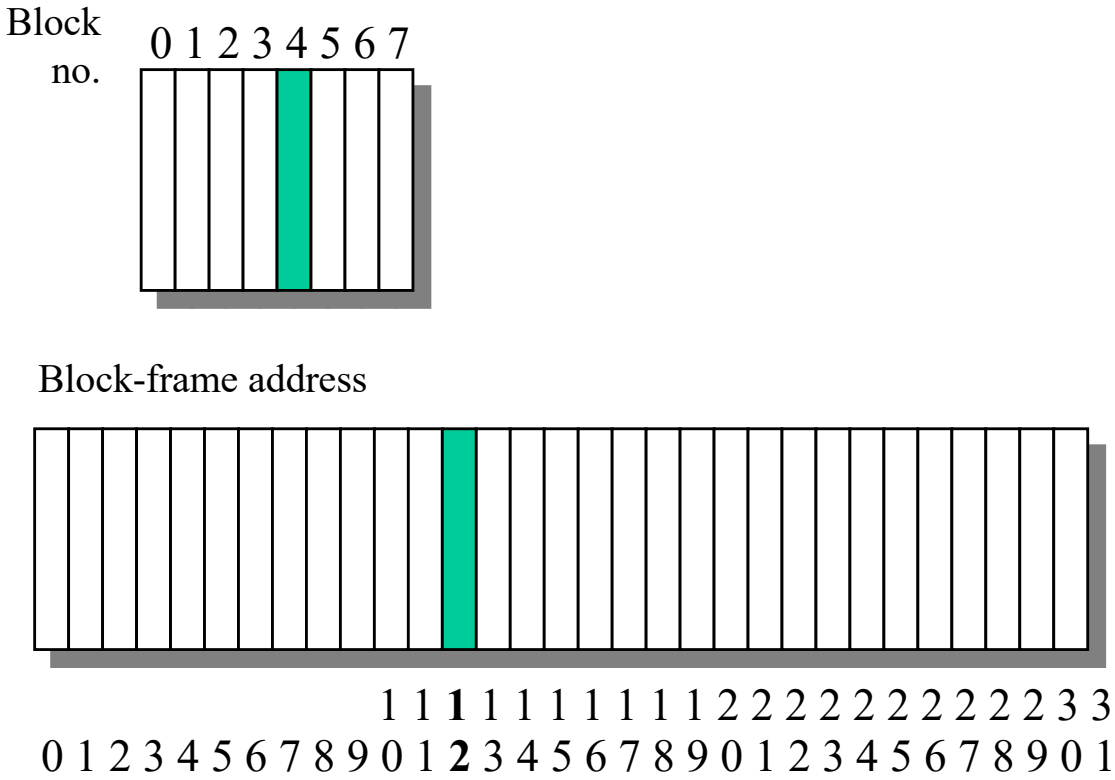


# Recap on Q1: Block Placement

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

**Block 12 can be placed**

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



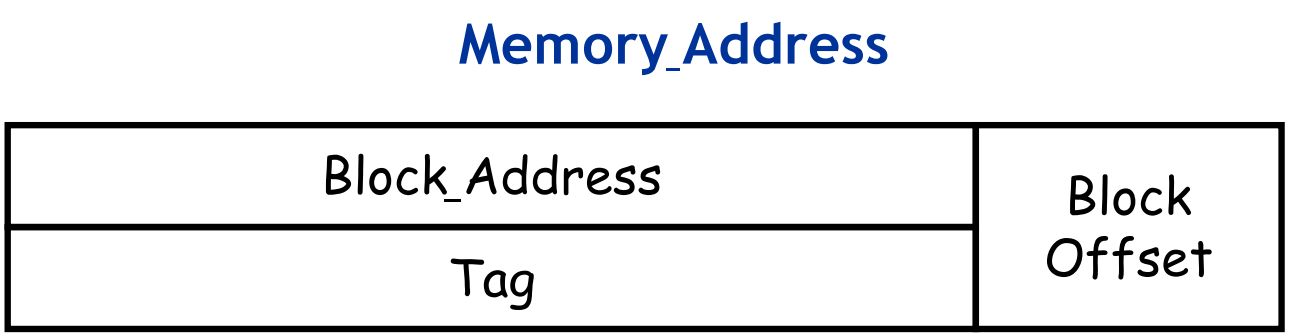
# Fully associative cache

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

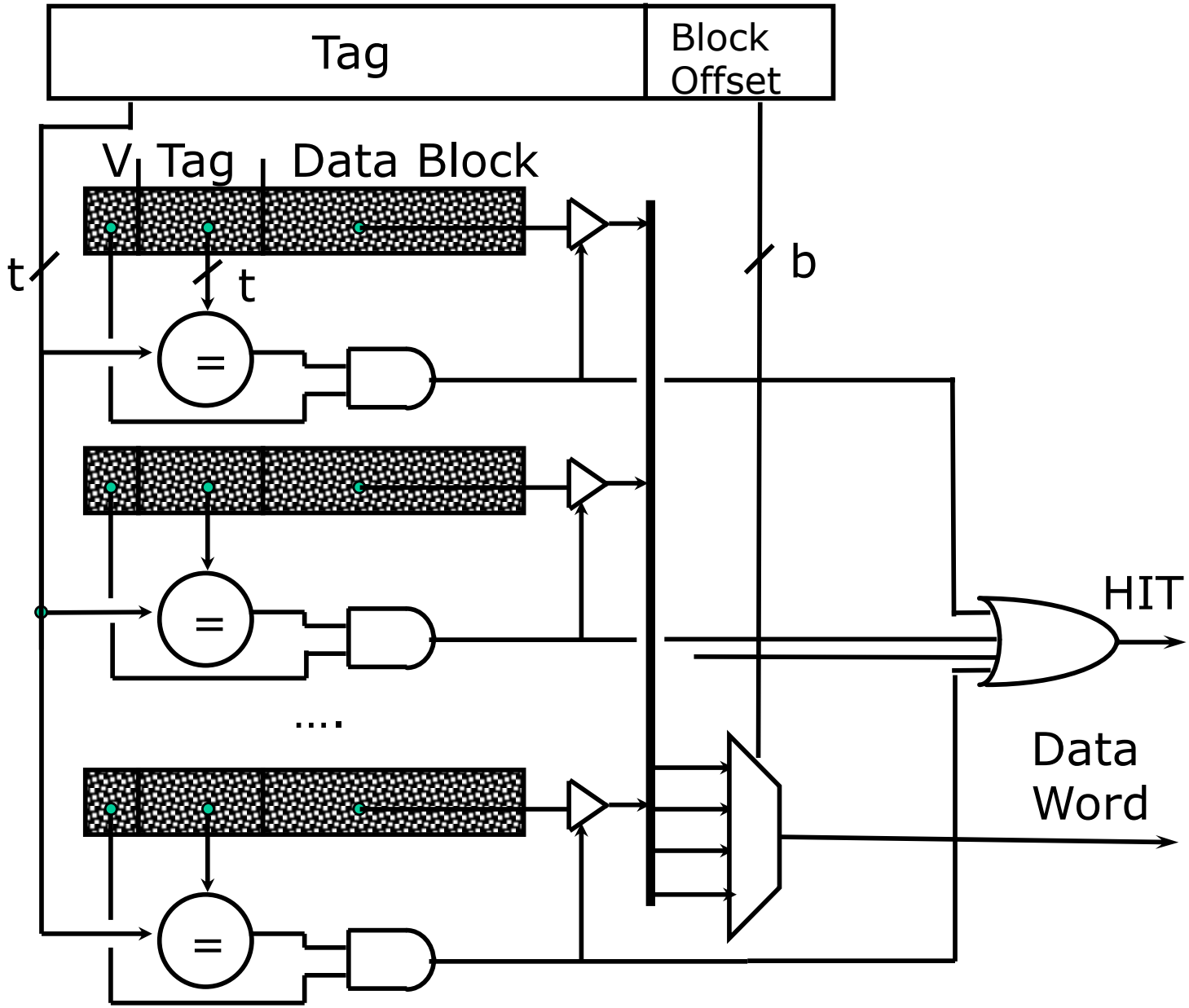
⇒ All the cache blocks must be checked during the search of the block

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

**Number of blocks = Cache Size / Block Size**



# Fully Associative Cache

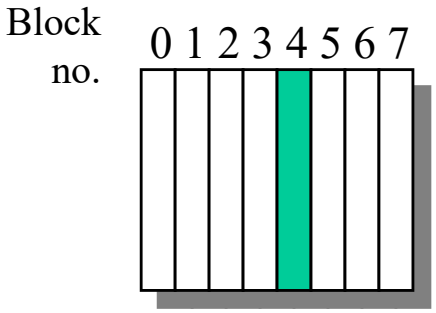


# Recap on Q1: Block Placement

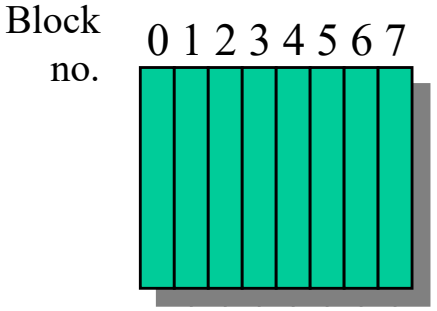
- *Where can block 12 be placed in the 8-block cache?*

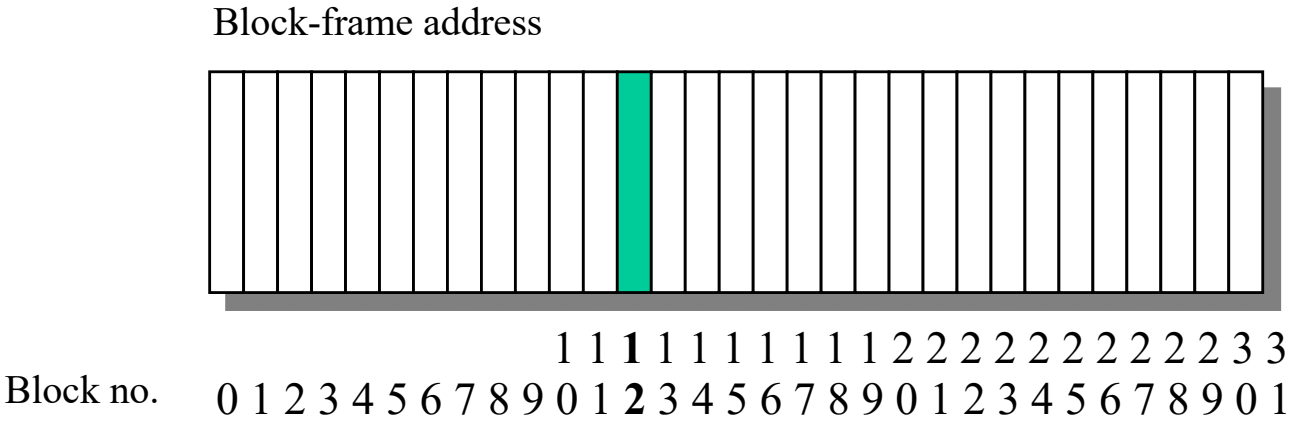
**Block 12 can be placed**

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




**Fully Associative: anywhere**



# n-way Set Associative Cache

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

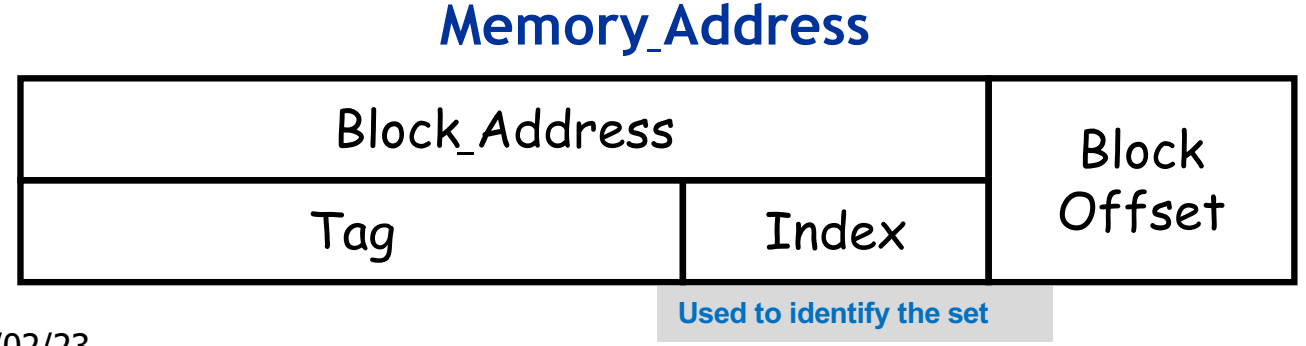
**Number of blocks = Cache Size/ Block Size**

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

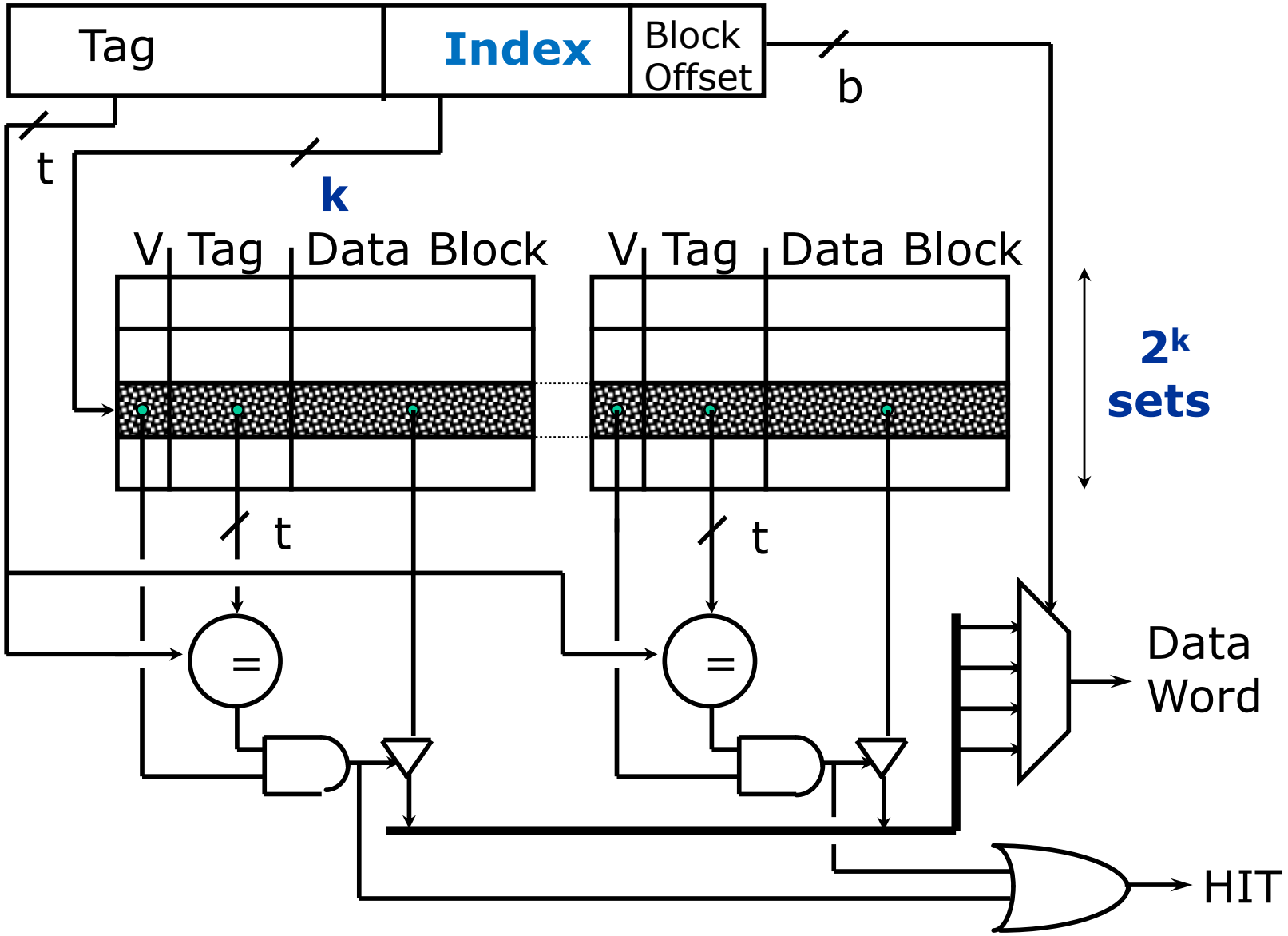
- The memory block can be placed in any block of the set
  $\Rightarrow$ the search must be done on all the blocks of the set.

- 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:

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



# 2-Way Set-Associative Cache



# Recap on Q1: Block Placement

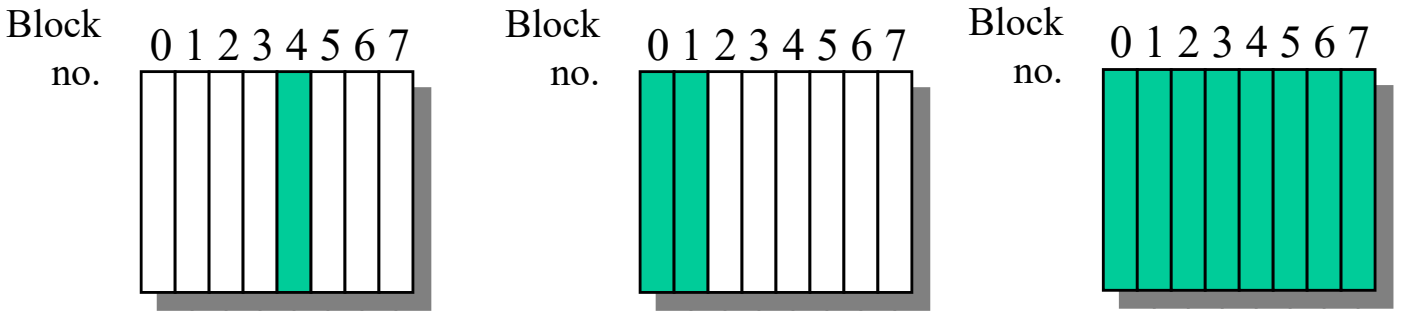
- *Where can block 12 be placed in the 8-block cache?*

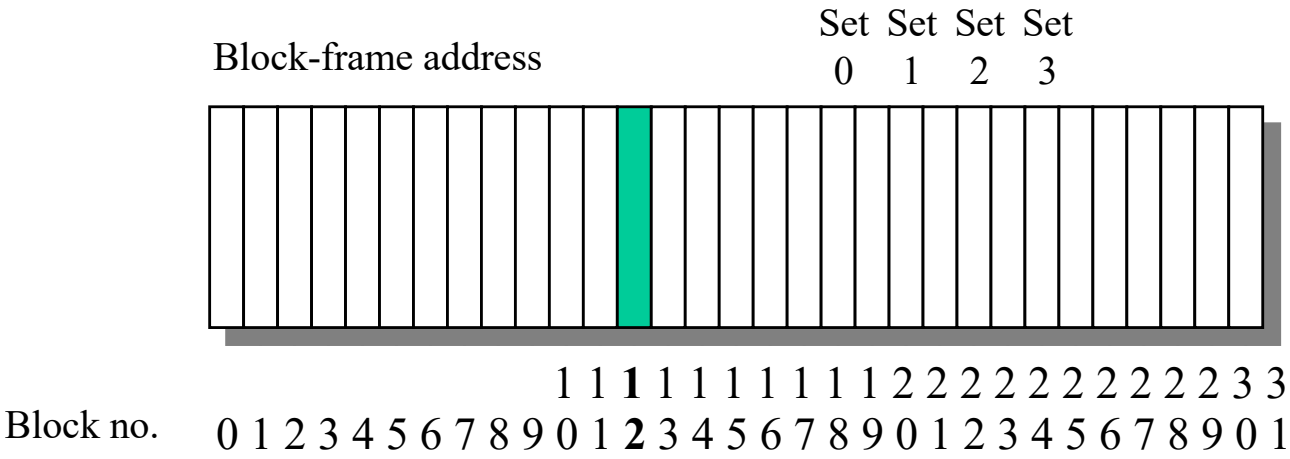
**Block 12 can be placed**

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

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

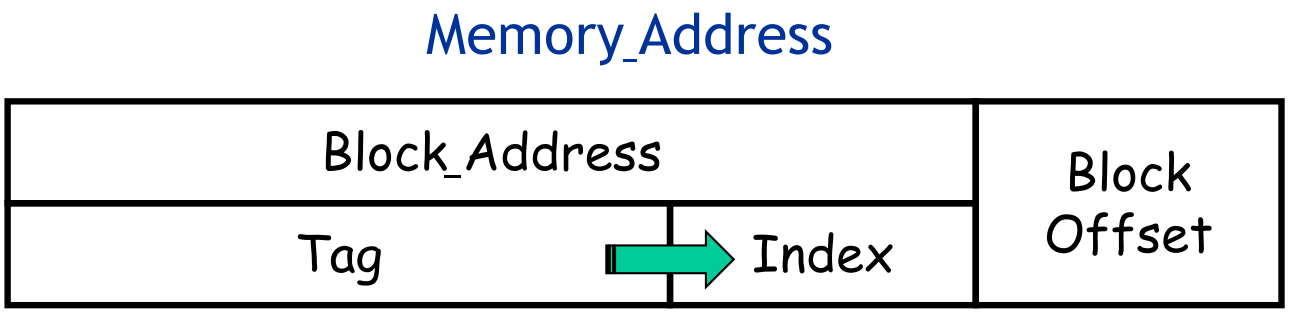
**Fully Associative:** **anywhere**



# 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*



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

- ***The problem of block placement:***
  - Direct Mapped
  - Fully Associative
  - n-way Set Associative

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

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

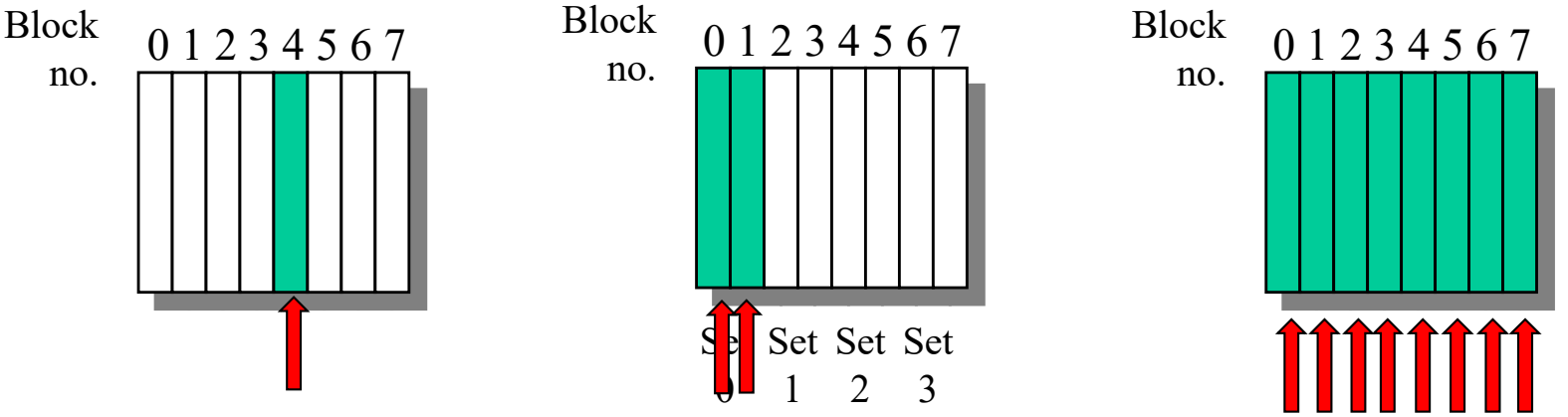
# Recap on Q2: Block Identification

**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:** Identify block position by index (Block Number MOD Number of blocks), compare tags of block and verify valid bit
- **Set-associative:** Identify set by index (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

# Q3: Which block should be replaced on a miss?

- 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

# Recap on Q3: Which block should be replaced on a miss?

- ***The problem of block replacement:***
  - Easy choice for direct mapped caches
  - Set associative or fully associative caches:
    - **Random**
    - **LRU (Least Recently Used)**
    - **FIFO**

# Q4: What happens on a write?

**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

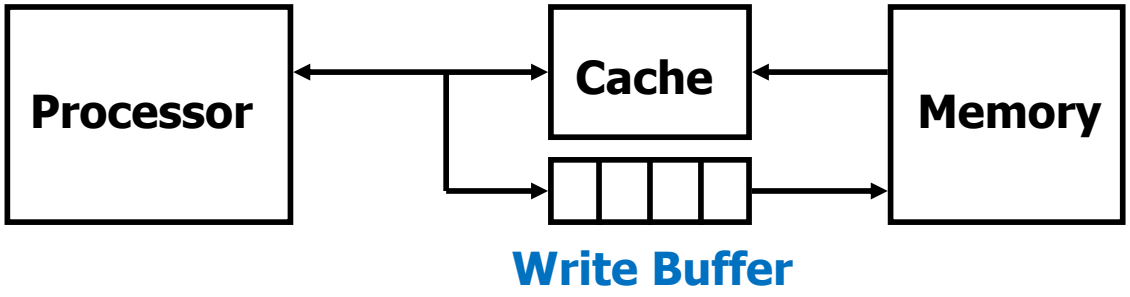
- **Write-Back:** the information is written only to the block in cache. The modified cache block is written to the lower-level memory **only** when it is replaced due to a miss.
  *Is a block clean or dirty? We need to add a* ***DIRTY bit***
  - At the end of the write in cache, the cache block becomes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

# Main advantages: Write-Back vs Write-Through

- **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-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 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, but also used combined with write back.

# More on Q4: Write Miss Options

- ***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)

# Recap on Q4: What happens on a write?

- ***Write Policies:***
  - Write through
  - Write back
- ***Write Miss Options:***
  - Write Allocate
  - No Write Allocate

# Symmary:
# Hit and Miss & Read and Write

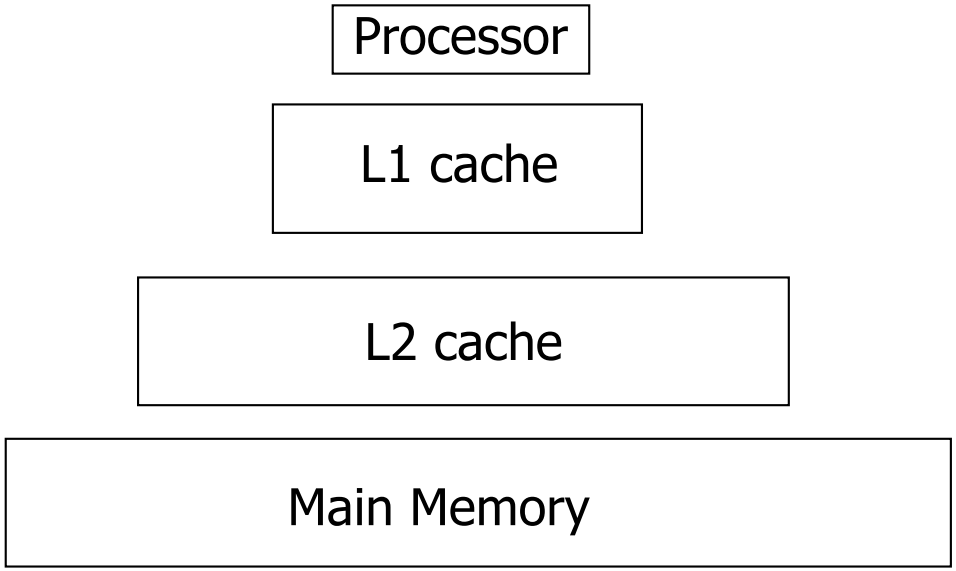
- **Read Hit:**
  - Read data in cache
- **Read Miss:**
  - CPU stalls, data request to memory, copy in cache *(write in cache),* repeat of *cache read* operation

- **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)**

# 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***



# 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$_{L2}$)

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

# Local and global miss rates

Definitions:

- ***Local miss rate:*** misses in this cache divided by the total number of memory accesses *to this cache*: the Miss $rate_{L1}$ for L1 and the 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$_{L1}$***
  - 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%

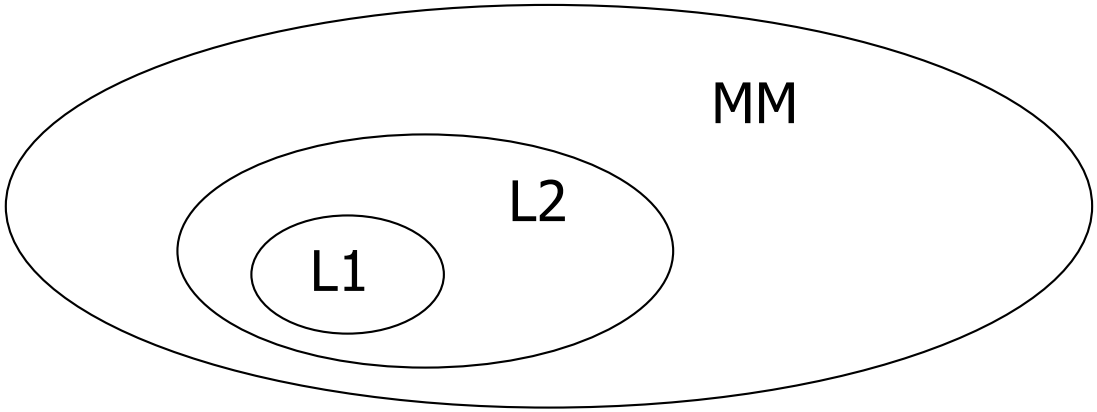
***Global Miss Rate for Last Level Cache (L2):***

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

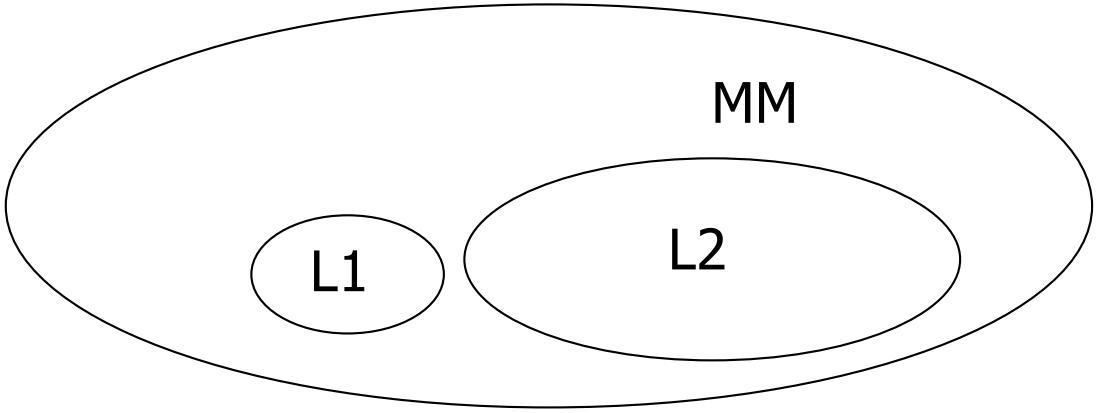
= (40 /1000) x (20 /40) = 2%

# L1 and L2 caches

- Multi-level Inclusion:




- Multi-Level Exclusion:



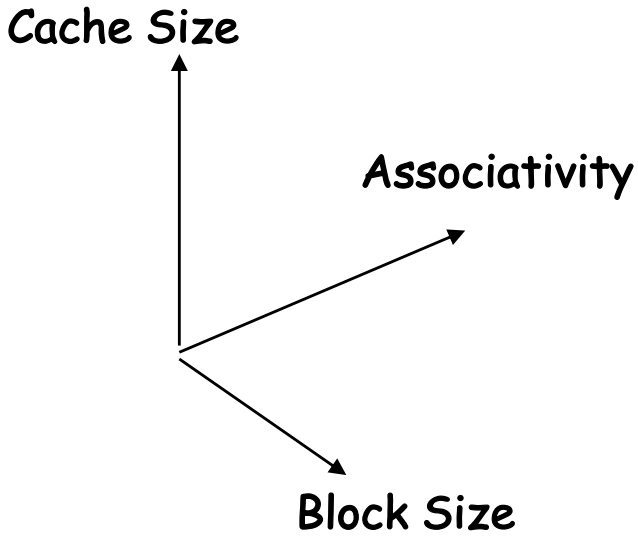
# Summary on caches

- **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

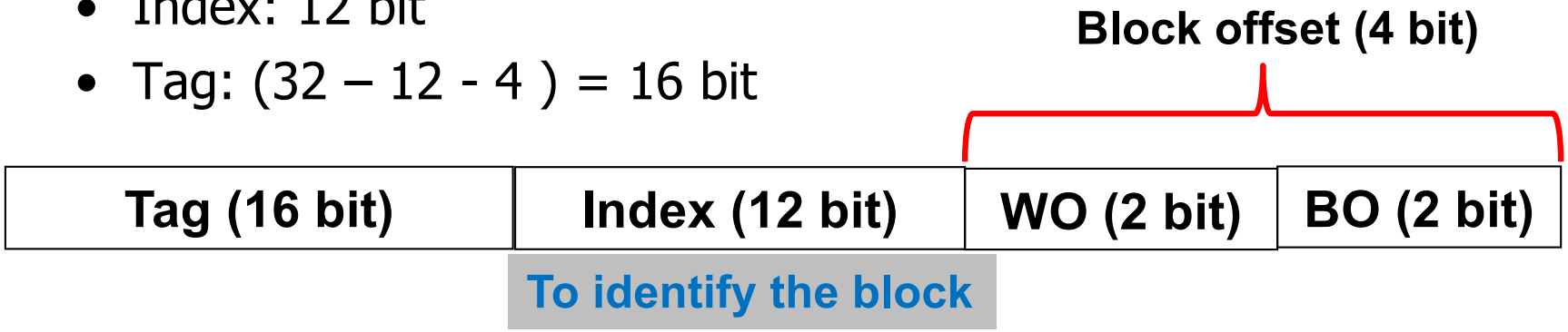



- More dimensions:
  - Replacement policy
  - Write-through vs write-back

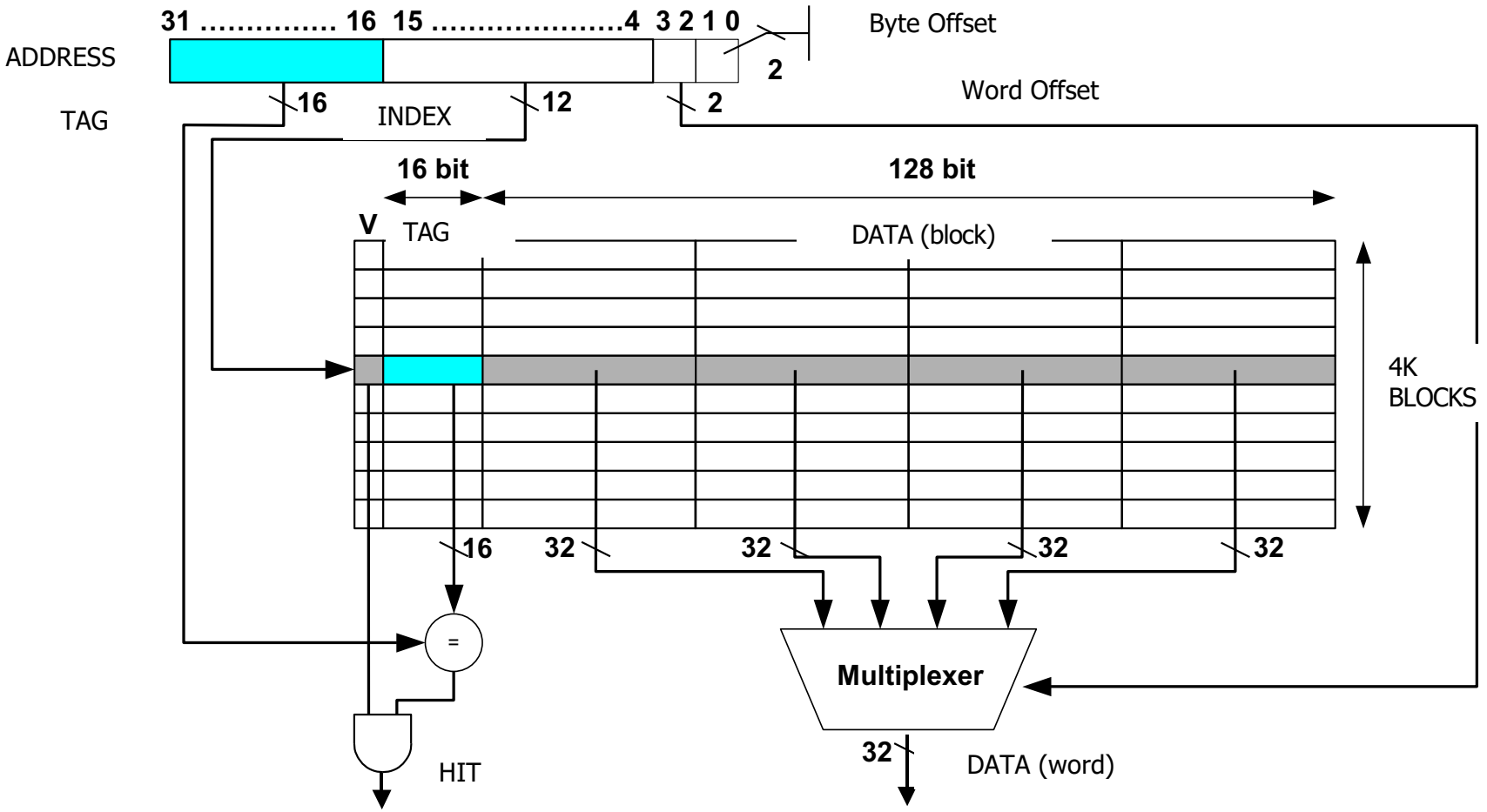
# CACHE DESIGN: SOME EXAMPLES

# Direct Mapped Cache: Example

- Memory address composed of N = 32 bit
- Cache Size 64 K Byte
- Block Size 128 bit = 16 Byte => 4 bit Block Offset
- Number of blocks = Cache Size / Block Size = 64 K Byte / 16 Byte = 4 K blocks => $2^{12}$ blocks => 12 bit Index
- Structure of the memory address:
  - Byte Offset: BO = 2 bit
  - Word Offset: WO = 2 bit
  - Index: 12 bit
  - Tag: (32 – 12 - 4 ) = 16 bit

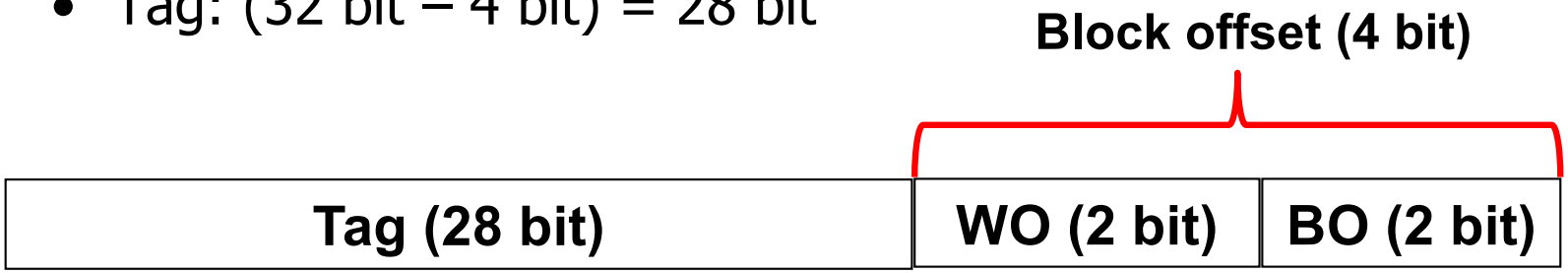


# Direct Mapped Cache: 64K-Byte cache size and 128-bit block size

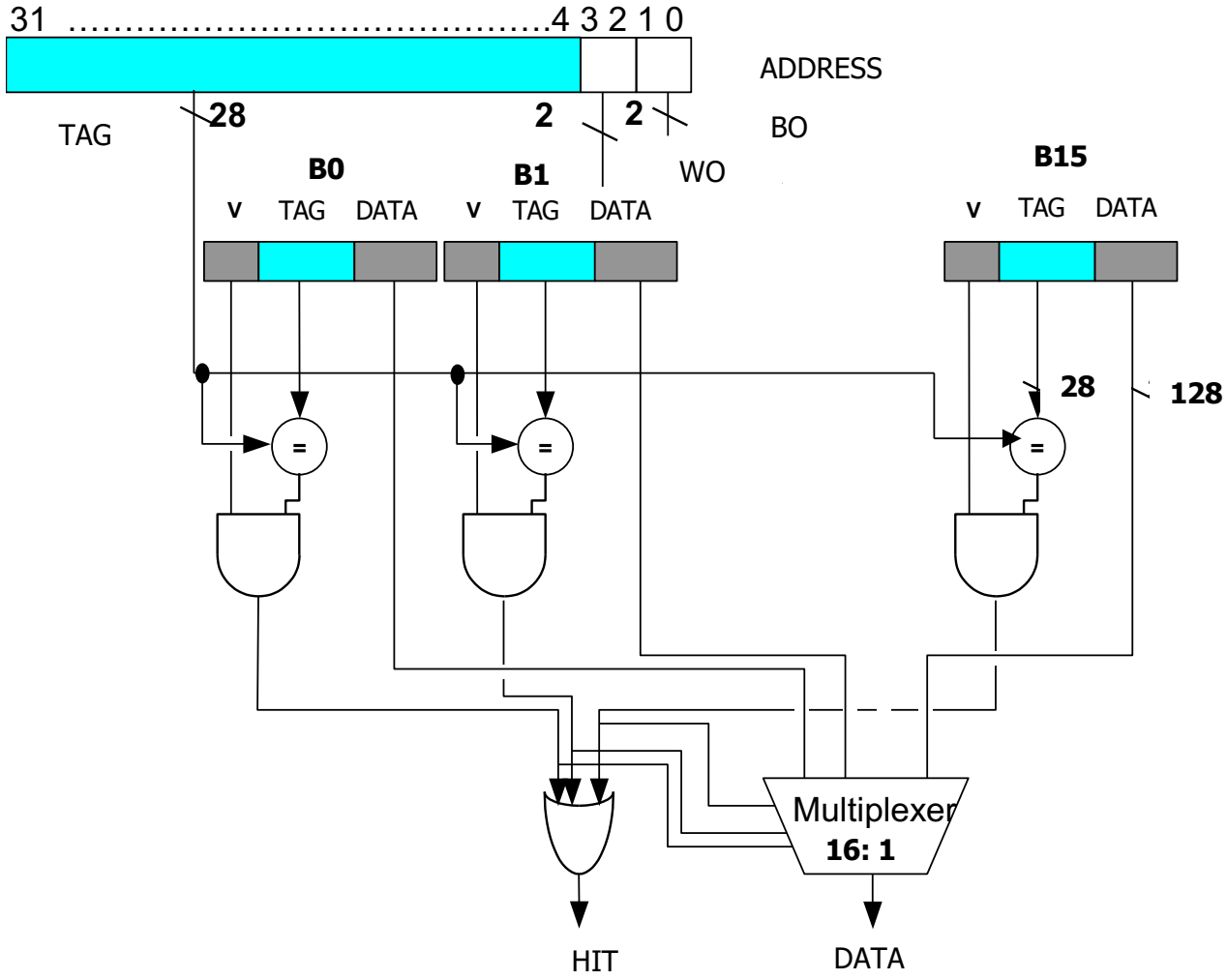


# Fully Associative Cache: Example

- Memory address composed of N = 32 bit
- Cache size 256 Byte
- Block size 128 bit = 16 Byte => 4 bit Block Offset
- Number of blocks = Cache Size / Block Size =
  = 256 Byte / 16 Byte = 16 Blocks
- Structure of the memory address:
  - Byte Offset: BO = 2 bit
  - Word Offset: KO = 2 bit
  - Tag: (32 bit – 4 bit) = 28 bit

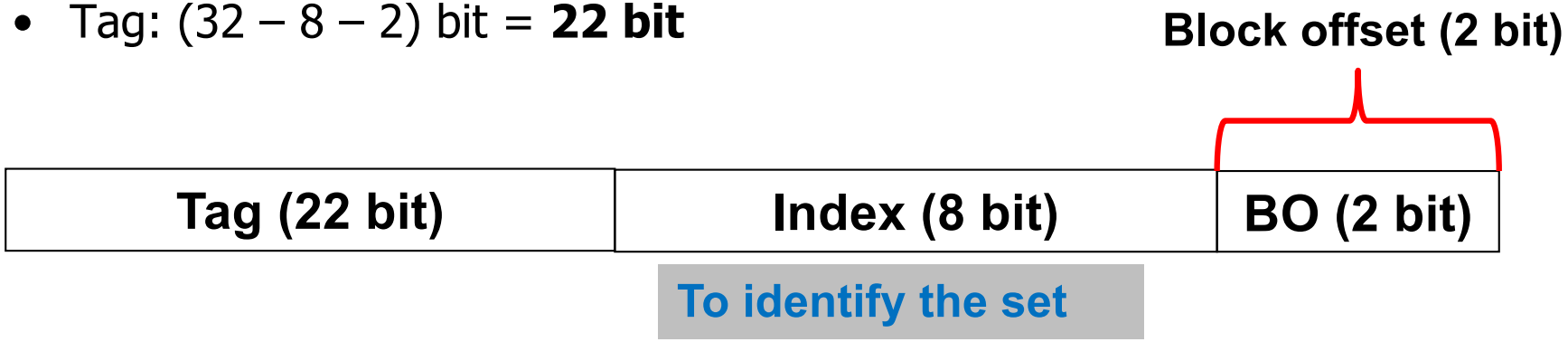


# Fully Associative Cache: 256-Byte cache size and 16-Byte block size

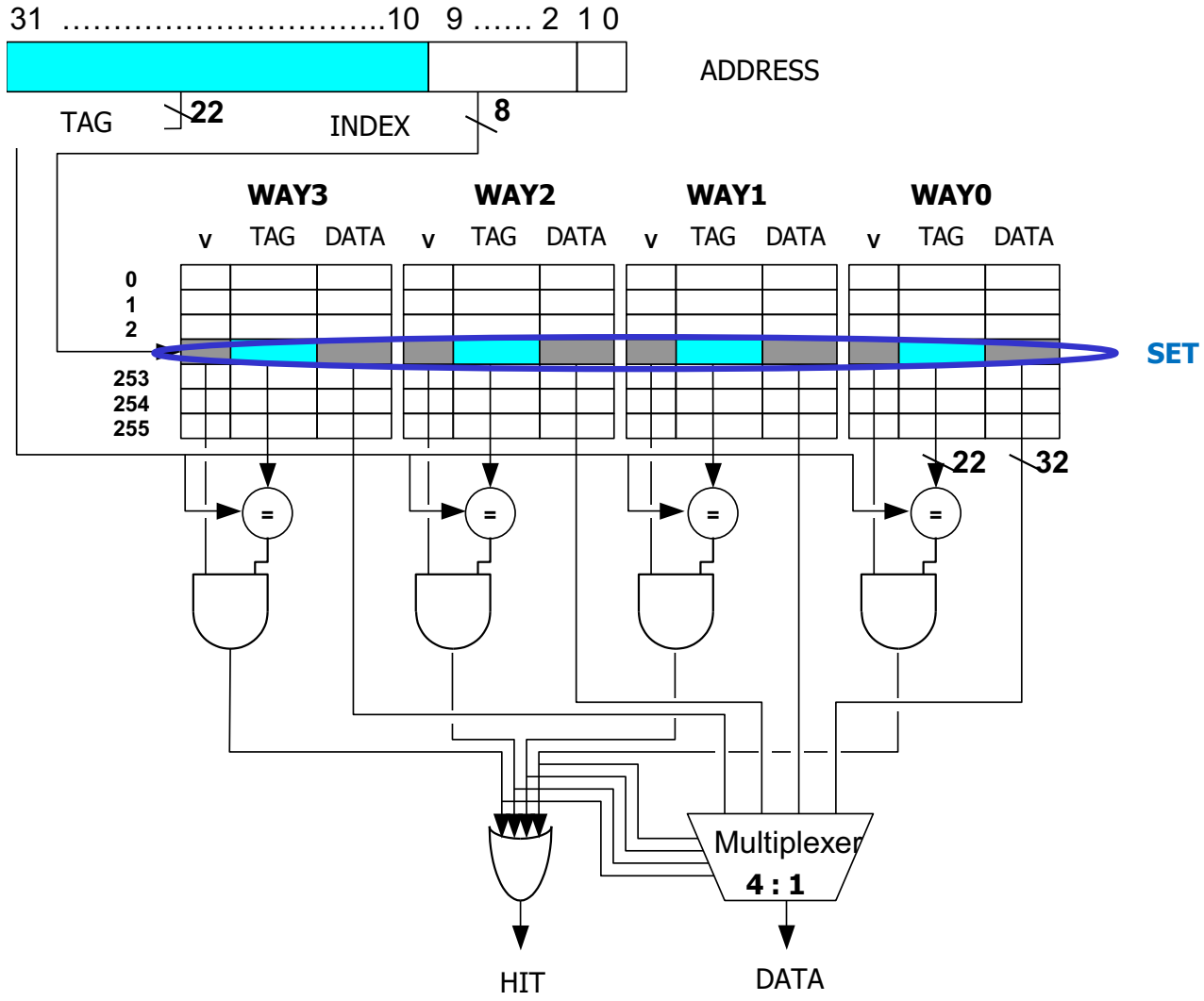


# 4-way Set Associative Cache: Example

- Memory address: N = 32 bit
- Cache Size 4KByte
- Block size 32 bit = 4 Byte => **2 bit Block Offset**
- 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 = $2^8$ sets => **8 bit Index**
- Structure of the memory address:
  - Byte Offset: BO = 2 bit
  - Word Offset: WO = 0 bit
  - Index: 8 bit
  - Tag: (32 – 8 – 2) bit = **22 bit**



# 4-way Set Associative Cache: 4K Byte cache size and 32-bit block size



# 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)

# References

- Appendix C *«Review of Memory Hierarchy»* of the textbook:
  J. Hennessey, D. Patterson,
  *"Computer Architecture: a quantitative approach"*
  *4* th Edition, Morgan-Kaufmann Publishers.