POLITECNICO MILANO 1863 | DIPARTIMENTO DI ELETTRONICA INFORMAZIONE E BIOINGEGNERIA

Advanced Computer Architectures

Cache and memory hierarchy

A.Y. 2024/2025 | Christian Pilato (christian.pilato@polimi.it)

POLITECNICO MILANO 1863 | DIPARTIMENTO DI ELETTRONICA INFORMAZIONE E BIOINGEGNERIA

Outline

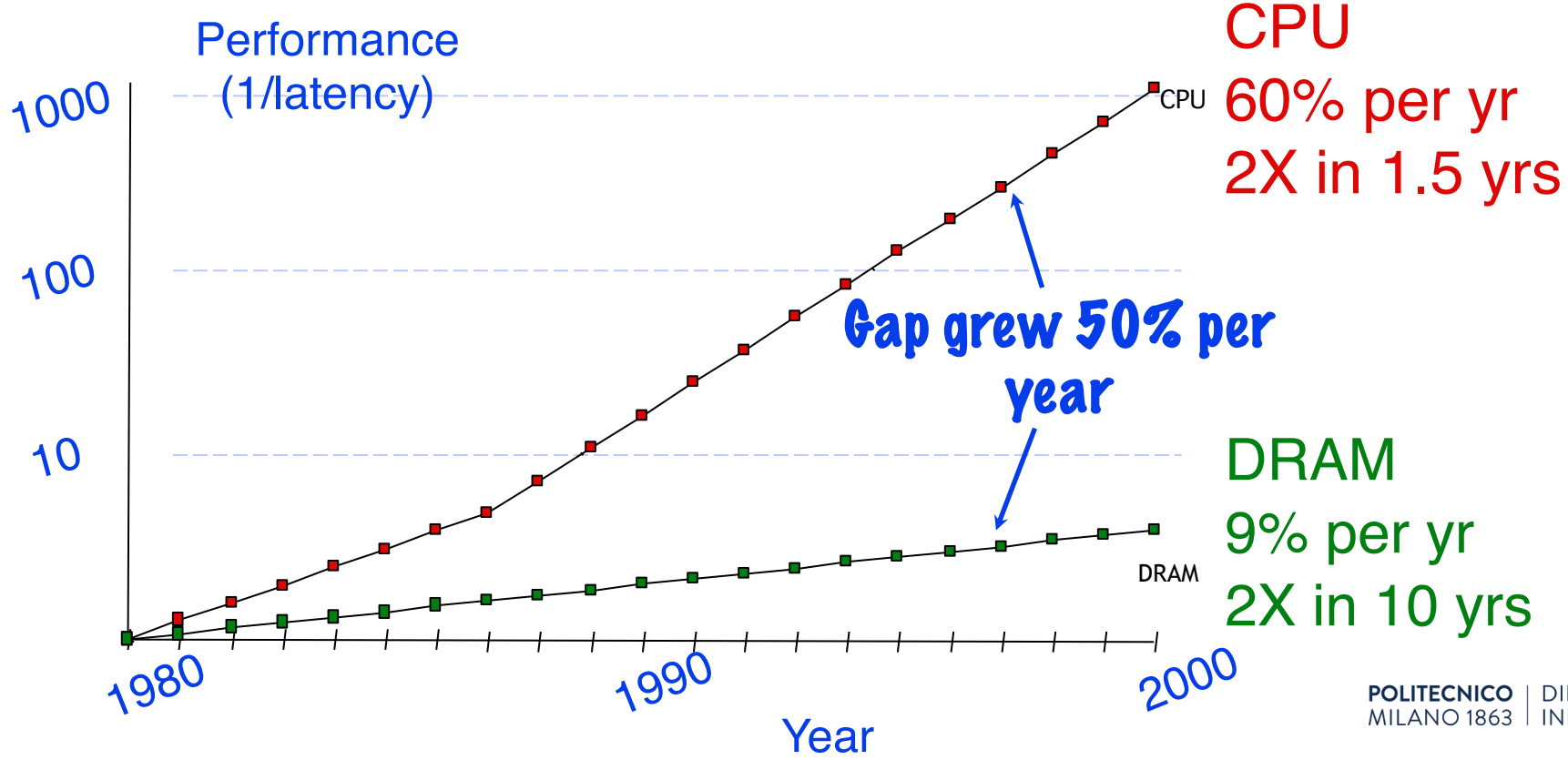
- Introduction to memory hierarchy
- Introduction to caches
- Questions on caches
- Performance evaluation
- Memory technology
- Virtual memory

Since 1980, CPU has outpaced DRAM ...

How do architects address this gap?

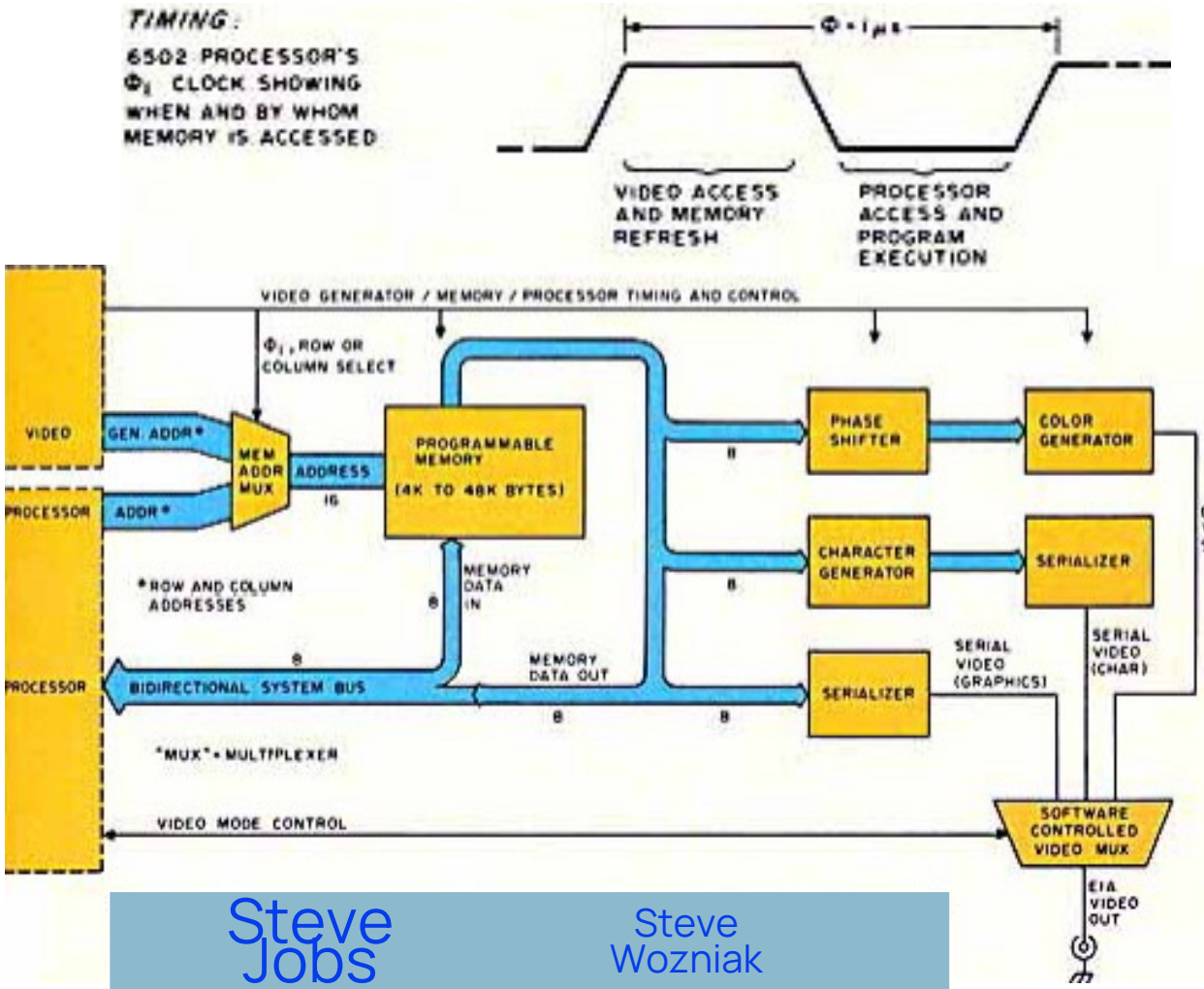
- Put small, fast "cache" memories between CPU and DRAM
- Create a "memory hierarchy"

Four-issue 2GHz superscalar accessing 100ns DRAM could execute 800 instructions during time for one memory access!



1977: DRAM faster than microprocessors

No need of hierarchy



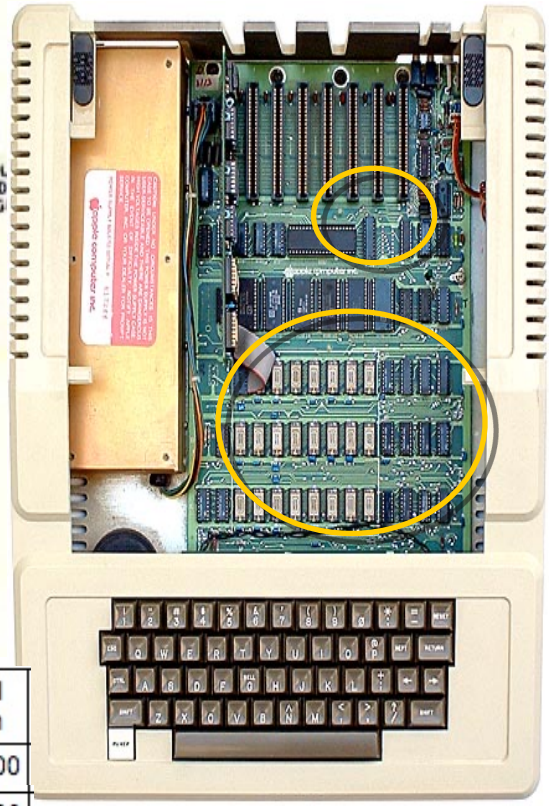
Steve Jobs

Steve Wozniak

Apple II (1977)

CPU: 1000 ns

DRAM: 400 ns



RAM Complement	Apple II System
4K	$ 1,298.00
48K	2,638.00

Processor – Memory gap

From 1980 to 1986:

- DRAM latency decreases 9% per year
- CPU performance increases 1.35x per year

After 1986:

- Performance increase for CPUs to 1.55 x per year
- DRAM performance constant

CPU performance started growing much faster than DRAM latency

Memory Hierarchy Design

Memory hierarchy design becomes more crucial with recent multi-core processors:

- Aggregate peak bandwidth grows with # cores:
  - Intel Core i7 can generate two 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

High-end microprocessors have >10 MB on-chip cache

- Consumes a large amount of area and power budget

Addressing the processor-memory performance gap

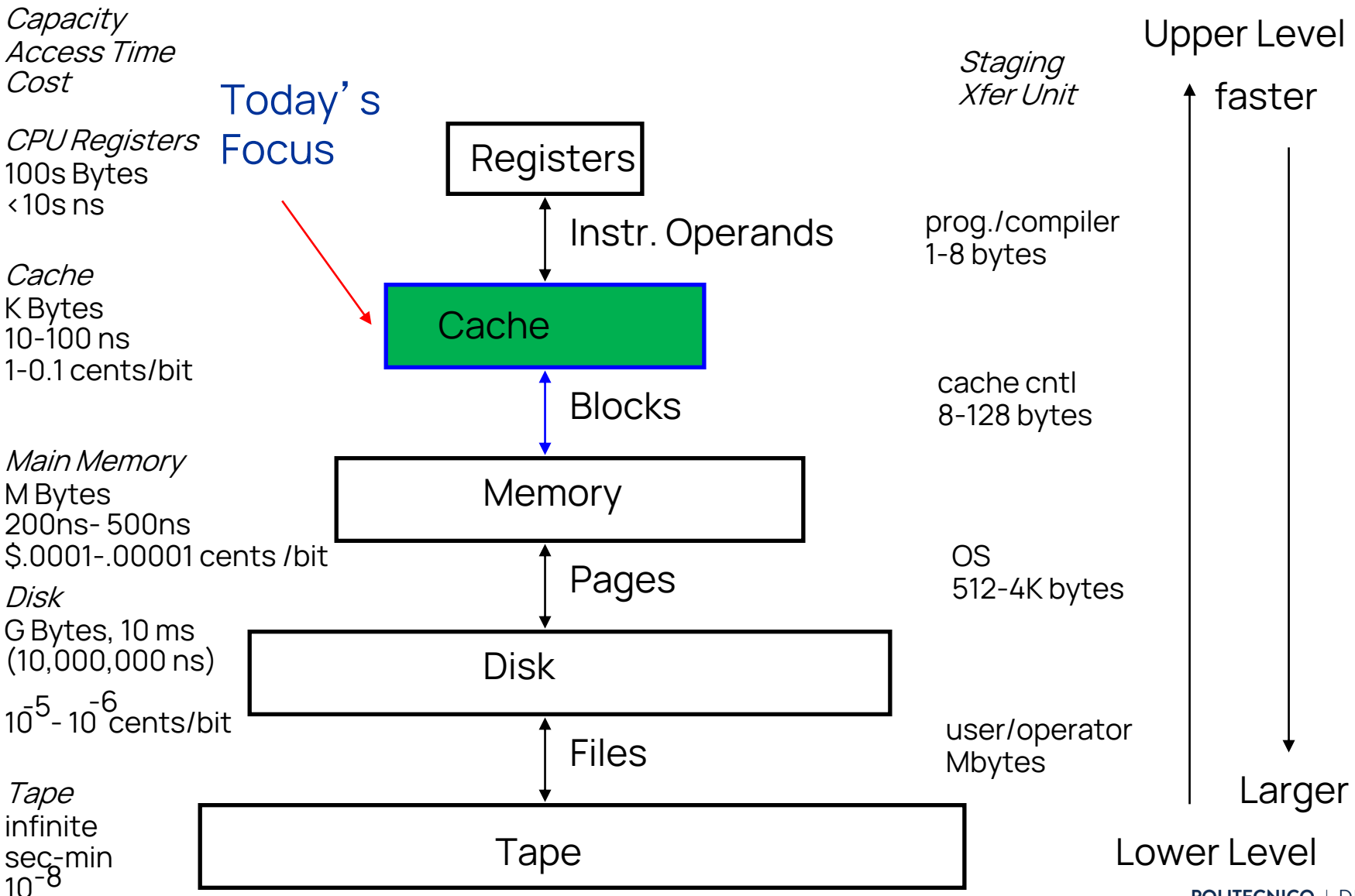
Goal

- Illusion of large, fast, cheap memory
- Let programs address a memory space that scales to the disk size, at a speed that is usually as fast as register access

Solution

- Memory hierarchy with different technologies, costs and sizes and different access mechanisms
- Put smaller, faster “cache” memories between CPU and DRAM. Create a “memory hierarchy”

Levels of the Memory Hierarchy



The principle of locality

Principle of Locality: Program access a relatively small portion of the address space at any instant of time

Two predictable properties of memory references:

- Temporal Locality: If a location is referenced, it is likely to be referenced again in the near future (e.g., loops, reuse)

- Spatial Locality: If a location is referenced, it is likely that locations near it will be referenced in the near future (e.g., straight-line code, array access)

- Last 15 years, HW relied on locality for speed

Memory Hierarchy: Apple iMac G5

Managed by compiler

Managed by hardware

Managed by OS, hardware, application

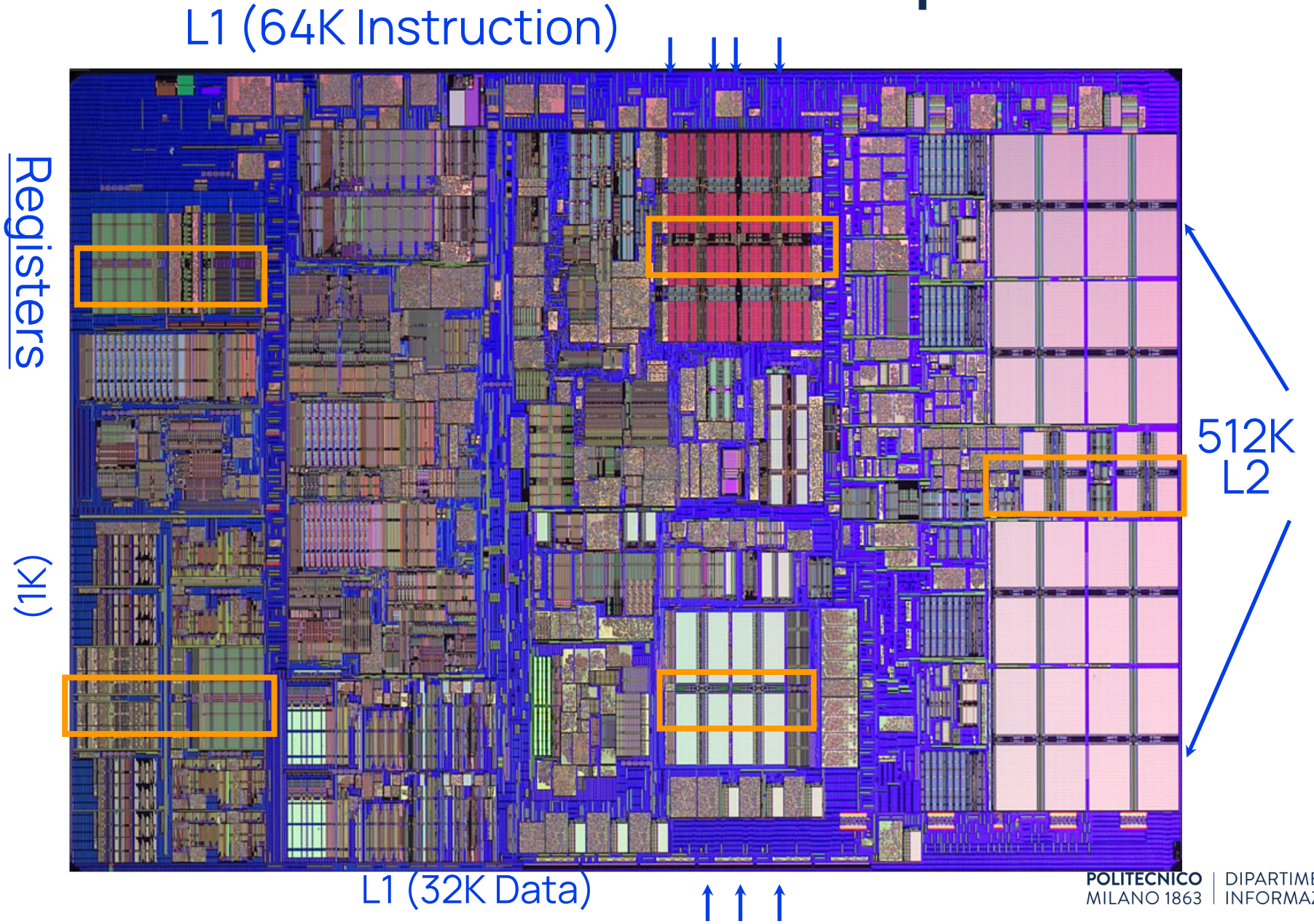
07	Reg	L1 Inst	L1 Data	L2	DRAM	Disk
Size	1K	64K	32K	512K	256M	80G
Latency Cycles, Time	1, 0.6 ns	3, 1.9 ns	3, 1.9 ns	11, 6.9 ns	88, 55 ns	10^7, 12 ms

Mac G5 1.6 GHz

Goal: Illusion of large, fast, cheap memory

Let programs address a memory space that scales to the disk size, at a speed that is usually as fast as register access

iMac's PowerPC 970: All caches on-chip



Caches

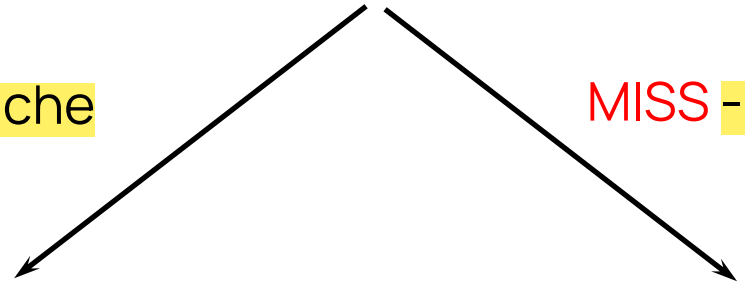
Caches exploit both types of predictability:

- Exploit **temporal locality** by remembering the contents of recently accessed locations
- Exploit **spatial locality** by fetching blocks of data around recently accessed locations

Cache algorithm: read

Memory is completely managed by hardware, CPU is not involved

Look at Processor Address, search cache tags to find match. Then either



HIT - Found in Cache

Return copy of data from cache

MISS - Not in cache

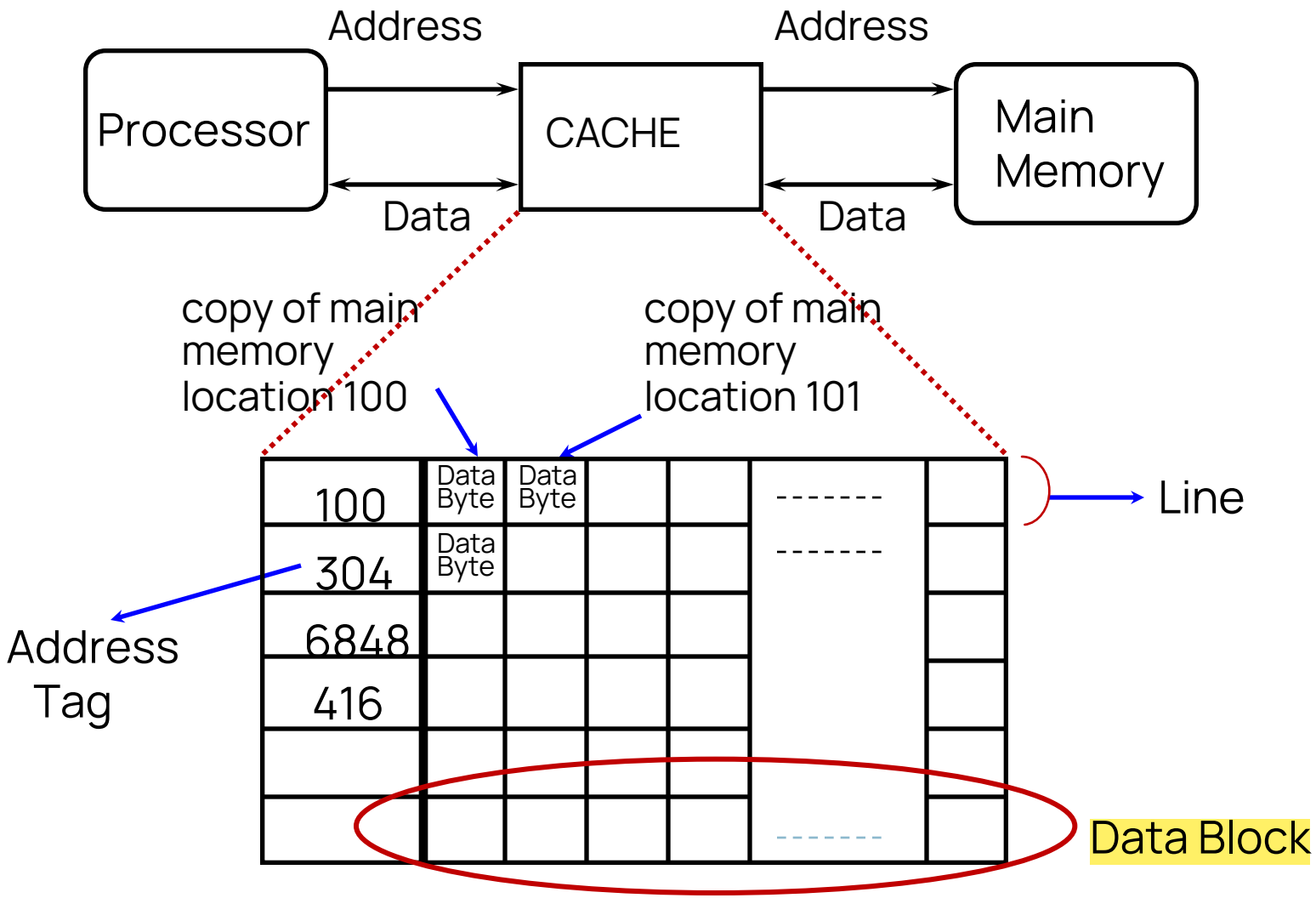
Read block of data from Main Memory

Wait …

Return data to processor and update cache

Hit Rate = fraction of accesses found in cache
Miss Rate = 1 – Hit rate
Hit Time = RAM access time + time to determine HIT/MISS
Miss Time = time to replace block in cache + time to deliver block to processor

Inside a cache



I can decompose the complete address in an identifier of the block and an offset

4 Questions on memory hierarchy

Q1: Where can a block be placed in the cache?

(Block placement)

put blocks one after the other means that in order to search for a block you need to "scan" all the blocks

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

(Block identification)

based on how I place blocks in memory it has an impact on how to find them in memory

Q3: Which block should be replaced on a miss?

(Block replacement)

Q4: What happens on a write?

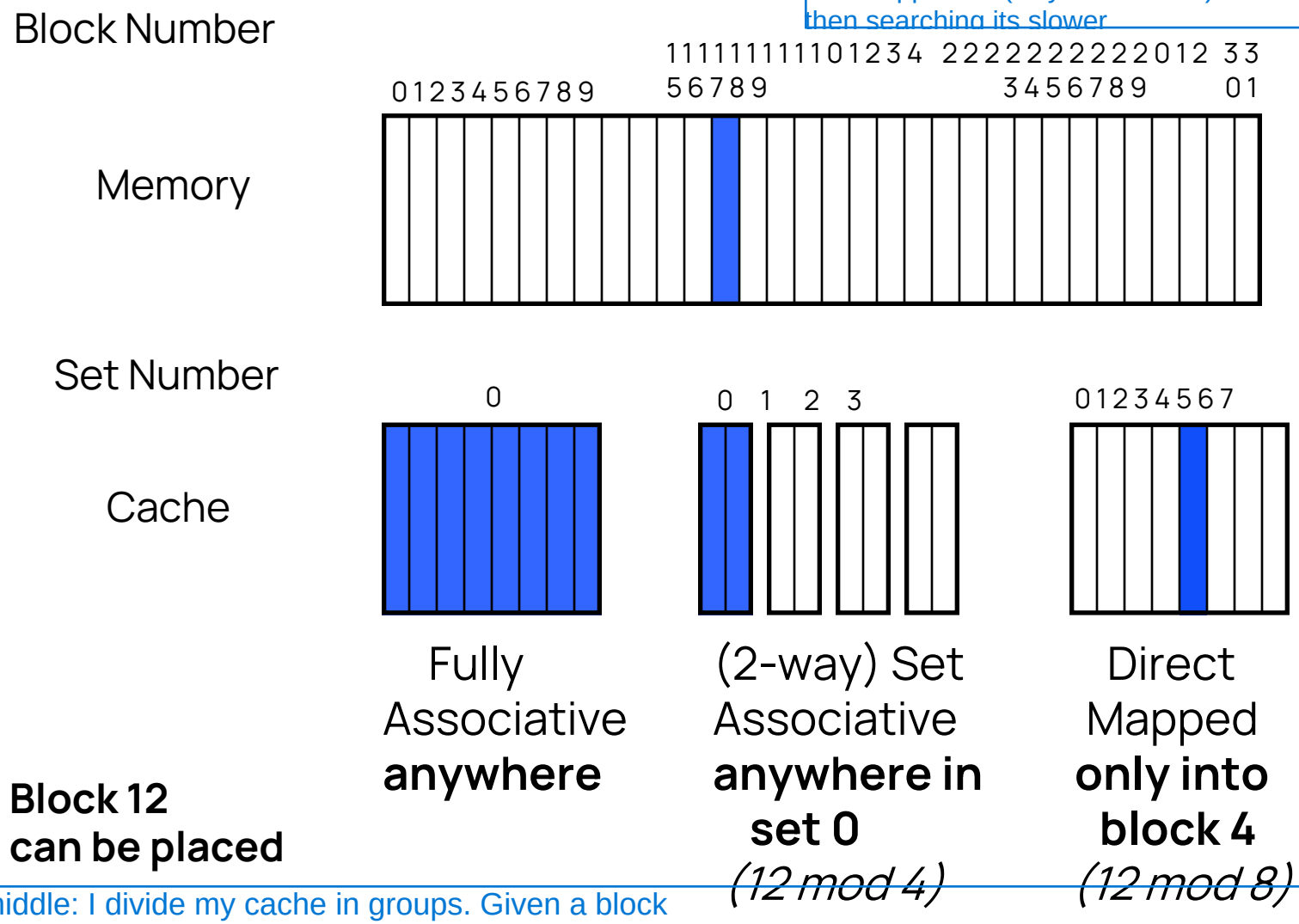
(Write strategy)

In cache there are copy of data, so if modified, only the copy gets modified.
We need to assure that other cpu that access the same block can see the changes.

Also, since the memory is volatile, it can happen that battery becomes empty and system shotdowm. actually a little part of battery is used to "save" the changes before the shutdown

Q1: Where can a block be placed?

Ex: cache with 8 blocks. For placing a block, we can take its address, take its module 8 rest and place it in cache in the position that equals the rest.
This is a very simple and immediate approach, but conflicts can arise (blocks mapped to the same position).
Other approach (fully associative): blocks can be positioned everywhere, but then searching its slower



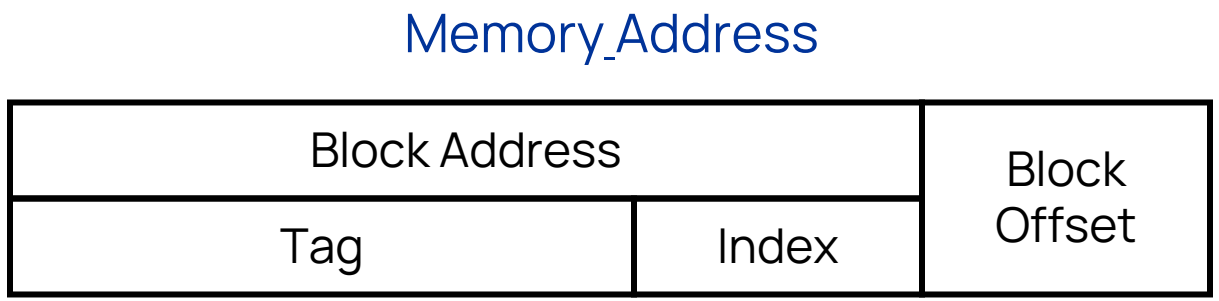
Best approach is in the middle: I divide my cache in groups. Given a block address I map it to a certain block executing the module operation, and then I put it into the group "freely" (associativity inside groups)

Sources of cache misses

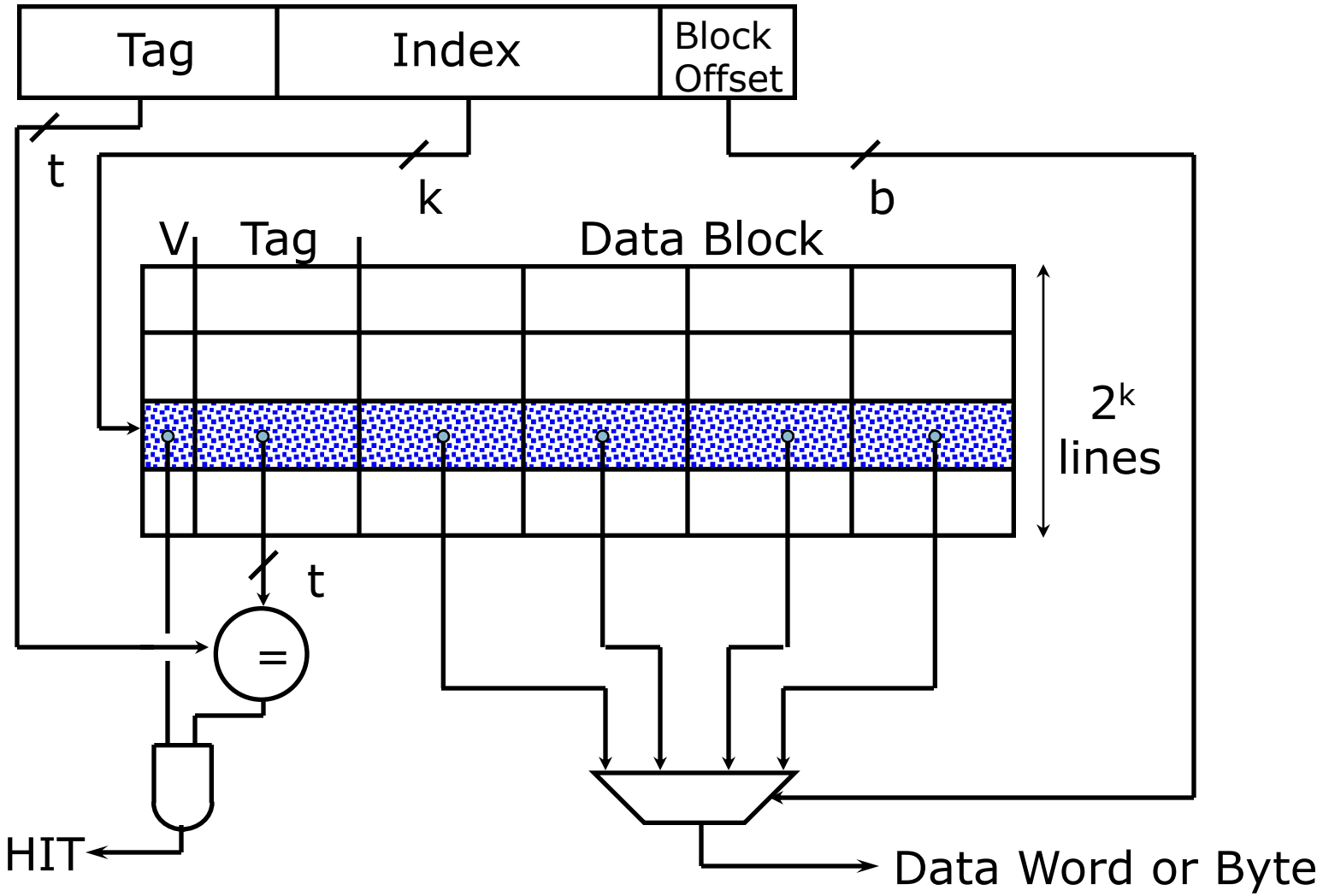
- Compulsory (cold start or process migration, first reference): first access to a block
  - "Cold" fact of life: not a whole lot you can do about it
  - Note: If you are going to run "billions" of instructions, Compulsory Misses are insignificant
- Capacity:
  - Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size
- Conflict (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity
- Coherence (Invalidation): other process (e.g., I/O) updates memory

Q2: How is a block found?

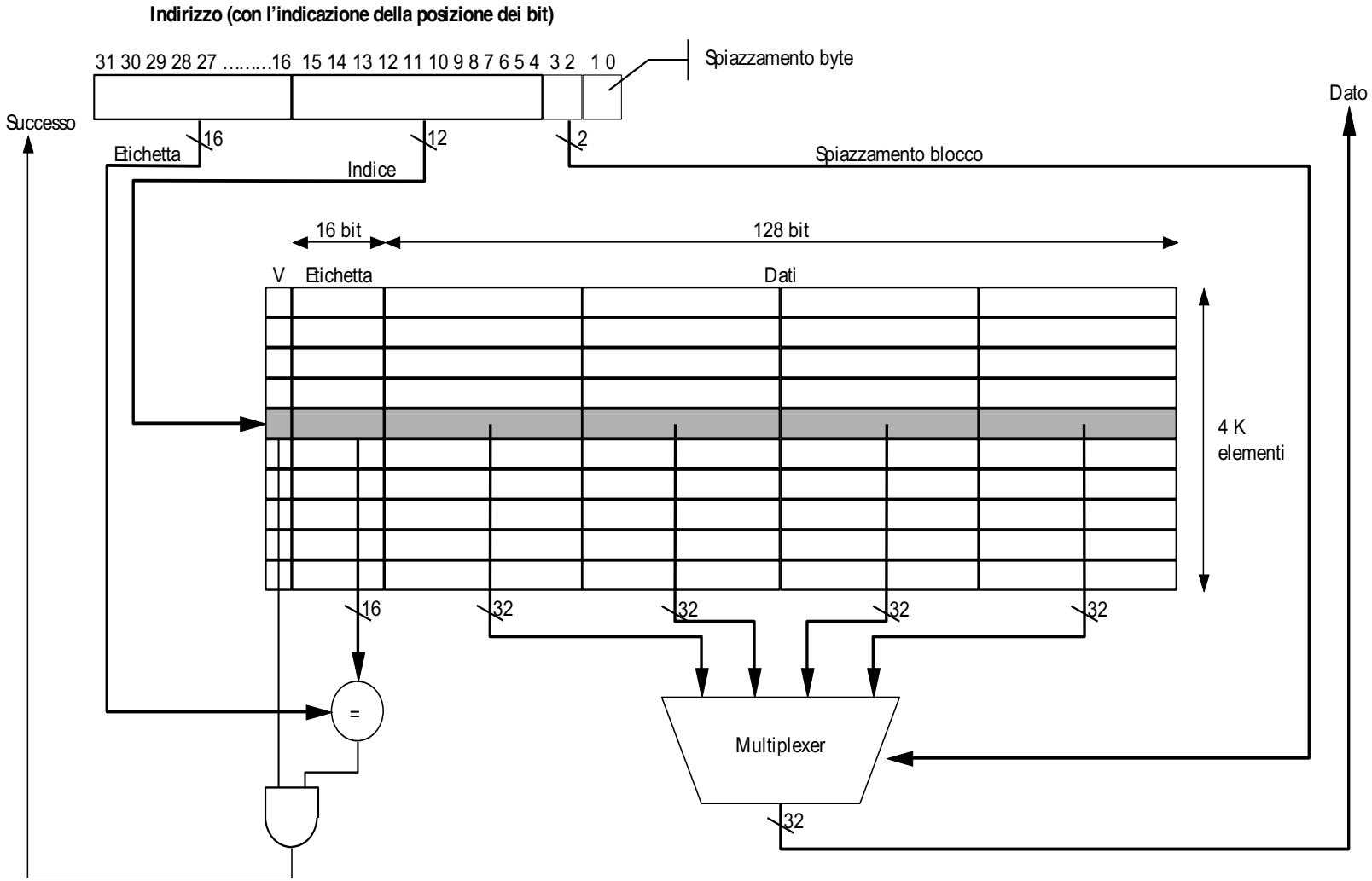
- Index selects which set to look in
- Tag used to identify the actual copy
  - If no candidates match, then declare cache miss
- Block is the minimum quantum of caching
  - Data select field used to select data within block
  - Many caching applications don't have data select field
- Tag on each block
  - No need to check index or block offset
- Increasing associativity shrinks index, expands tag. Fully Associative caches have no index field



Direct-Mapped Cache

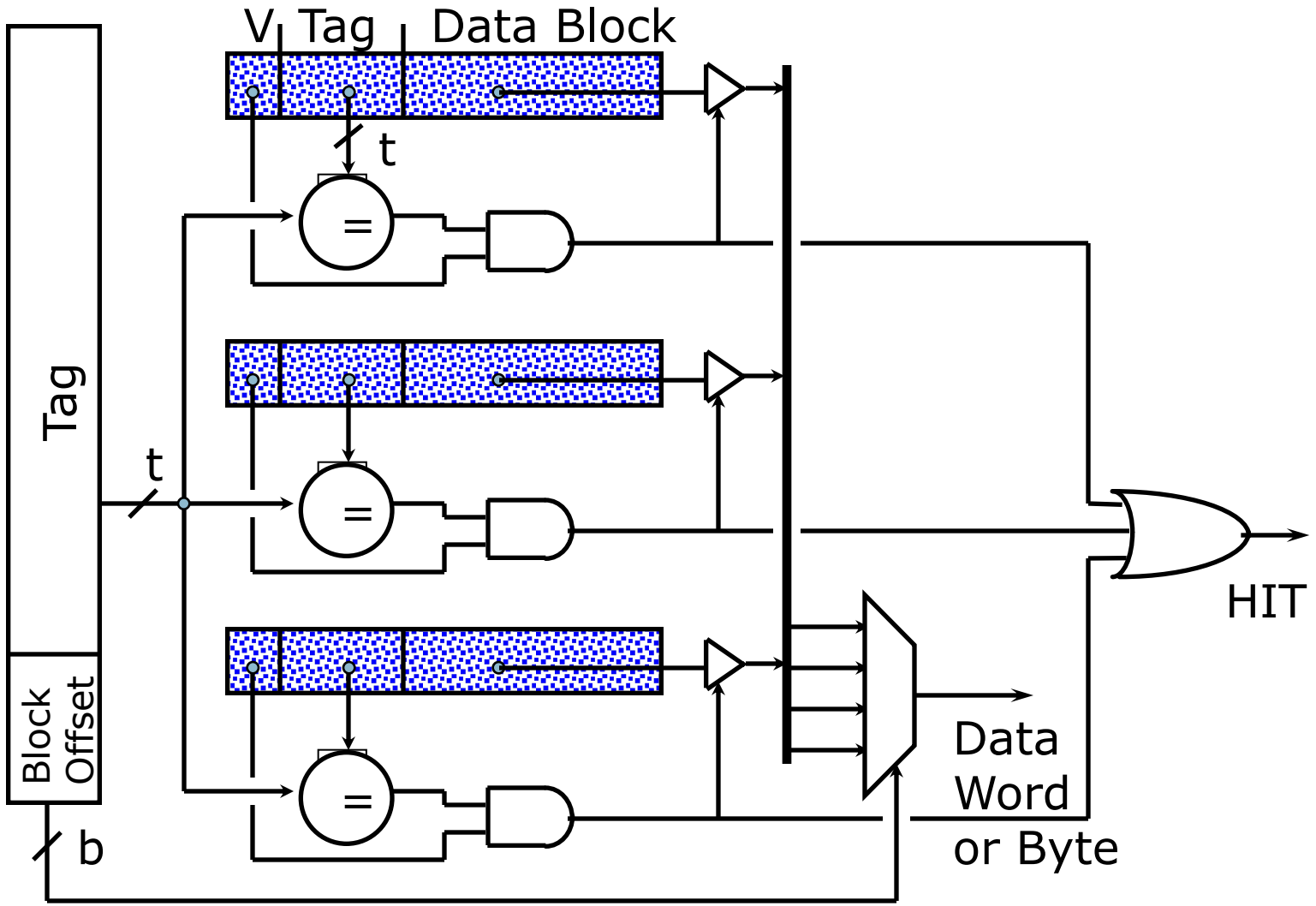


Direct-Mapped Cache

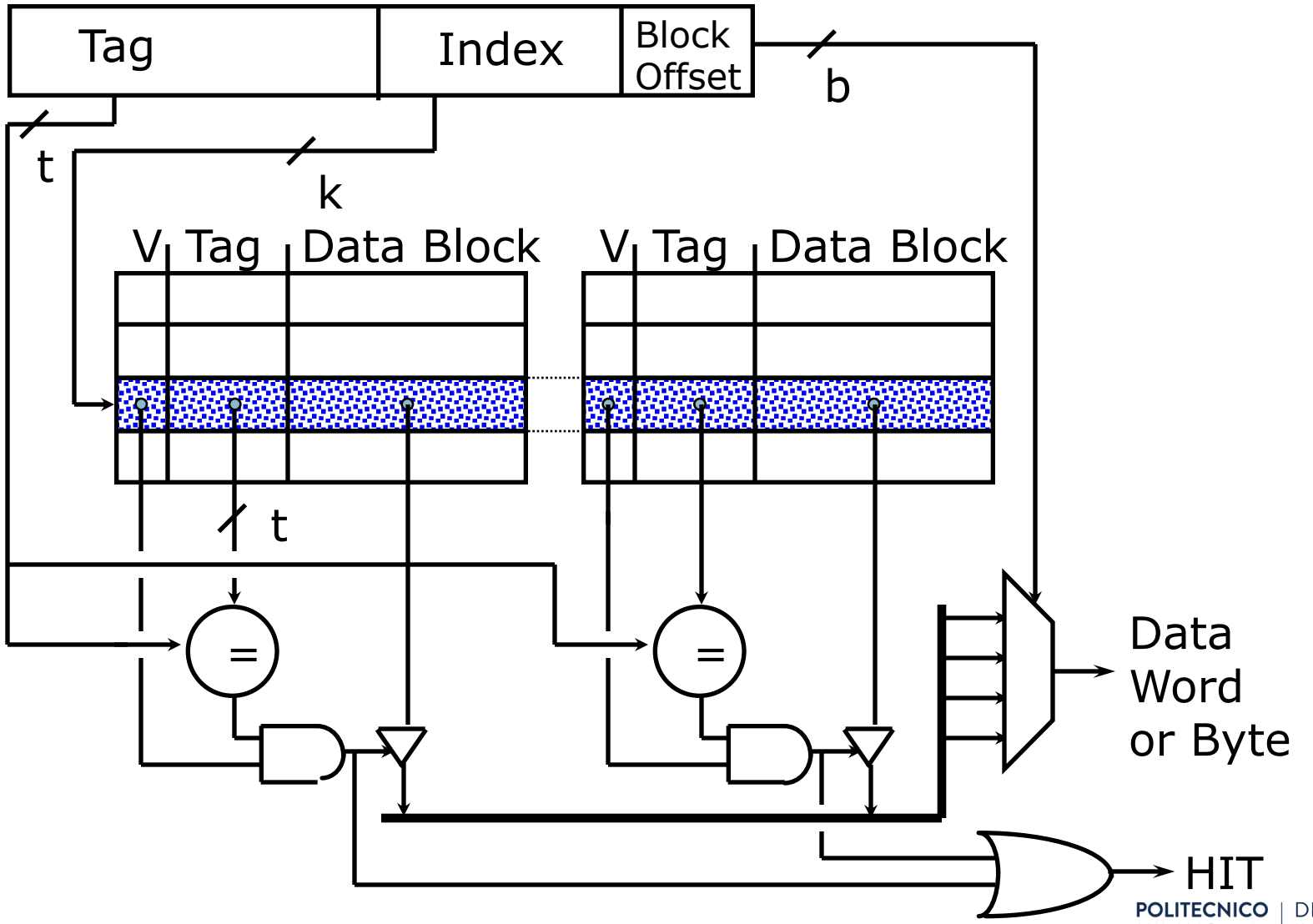


Fully Associative Cache

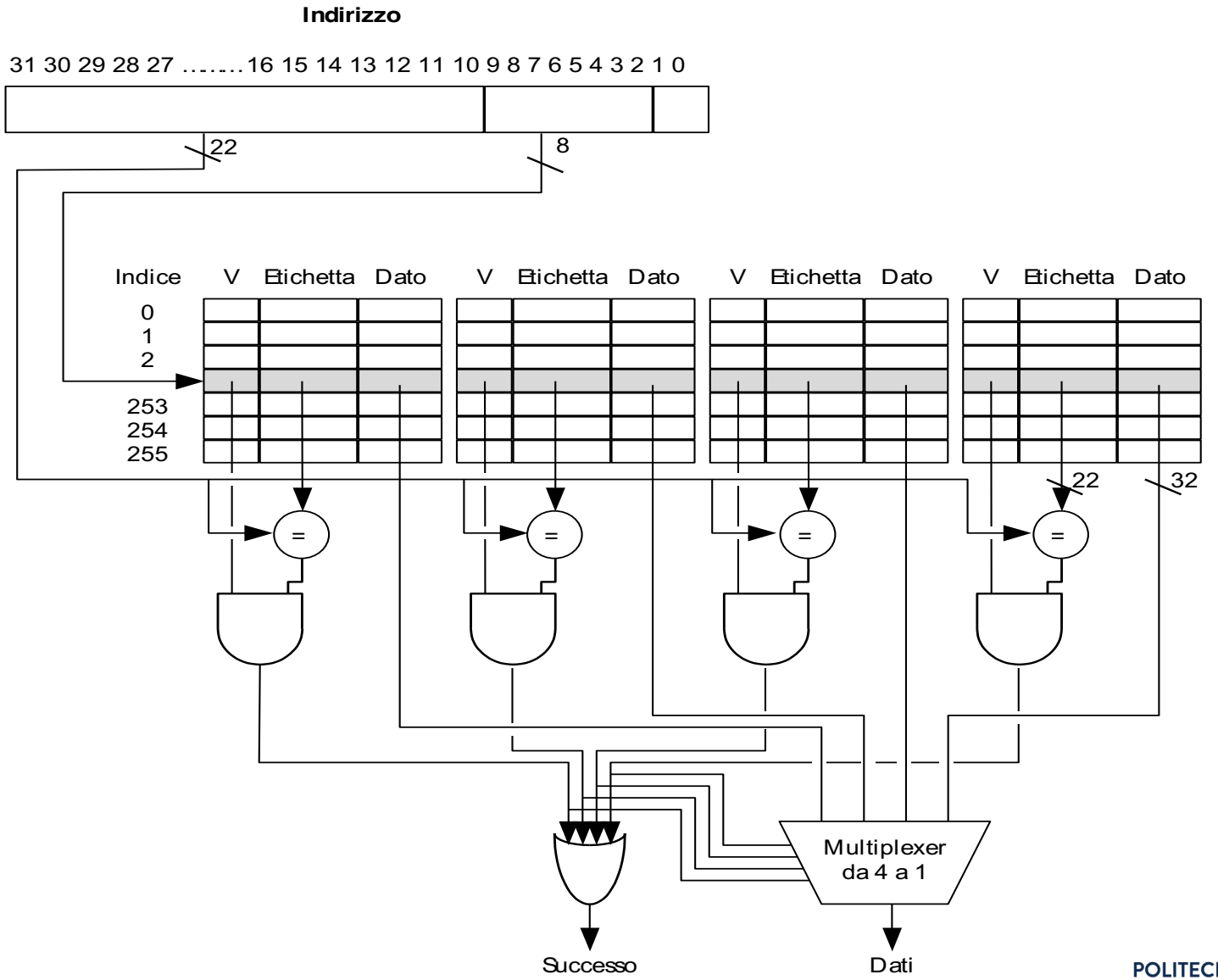
tag compared to every tag



2-Way Set-Associative Cache

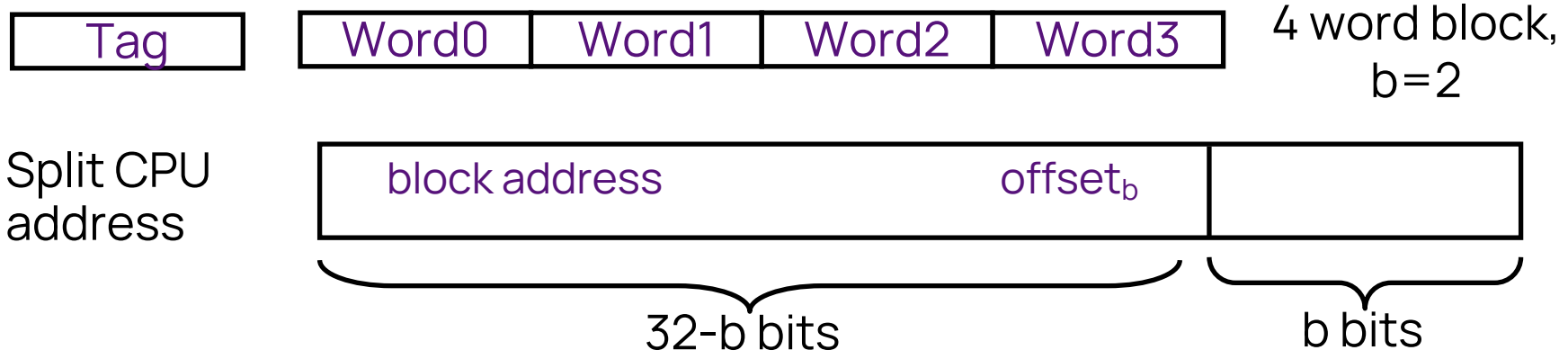


Architecture



Block Size and Spatial Locality

Block is unit of transfer between the cache and memory



Larger block size has distinct hardware advantages

- less tag overhead
- exploit fast burst transfers from DRAM
- exploit fast burst transfers over wide busses

What are the disadvantages of increasing block size?

Fewer blocks => more conflicts. Can waste bandwidth.

Q3: Which block should be replaced on a miss?

- Easy for Direct Mapped
- Set Associative or Fully Associative:
  - Random
  - Least Recently Used (LRU)
    - LRU cache state must be updated on every access
    - true implementation only feasible for small sets (2-way)
    - pseudo-LRU binary tree often used for 4-8 way
  - First In, First Out (FIFO) a.k.a. Round-Robin
    - used in highly associative caches

--> this does not consider uses, but only how "old" is the block, it's not the same as the LRU

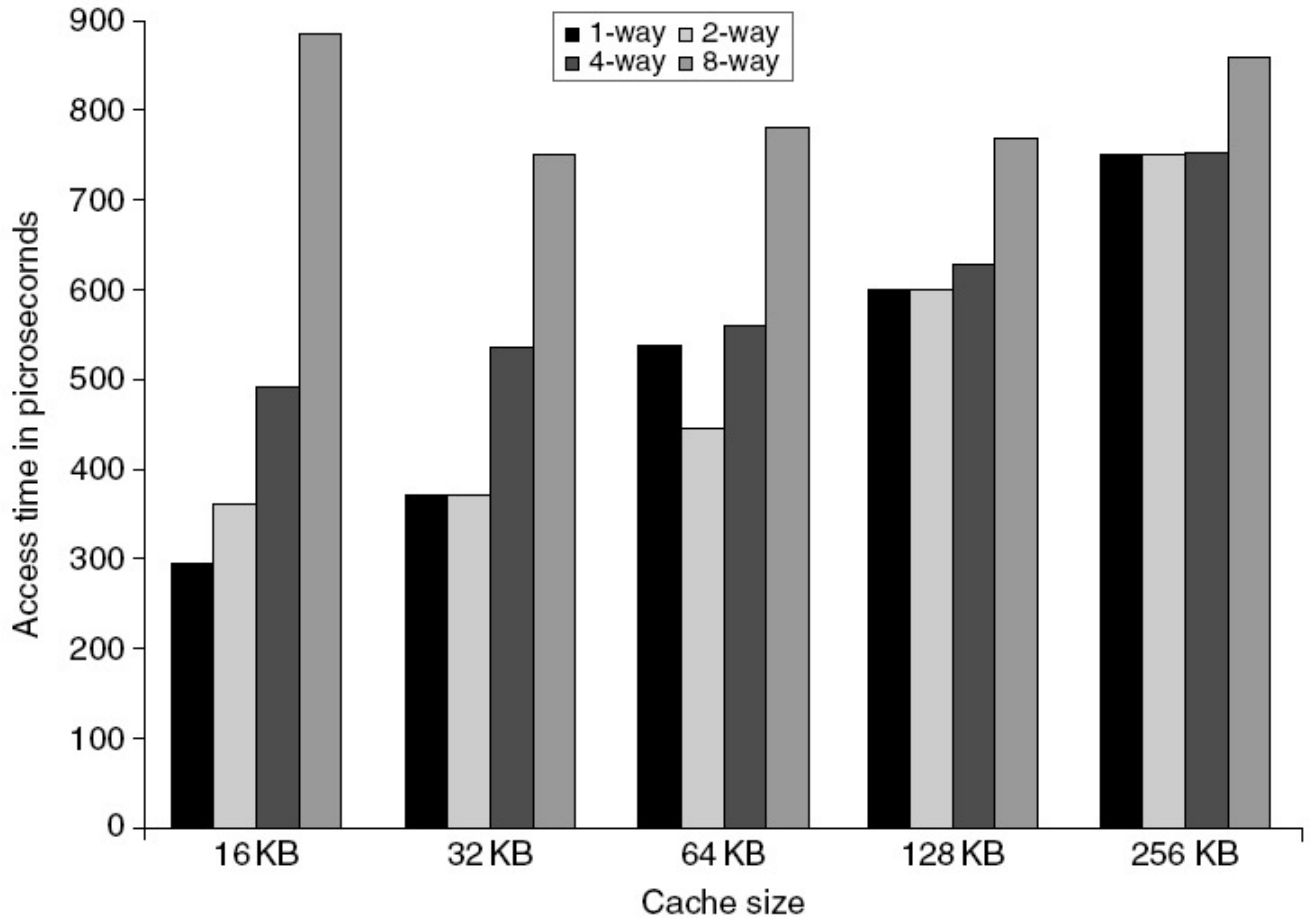
- Replacement policy has a second-order effect since replacement only happens on misses

How well random choice works

Assoc:	2-way		4-way		8-way	
Size	LRU	Ran	LRU	Ran	LRU	Ran
16K	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64K	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256K	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

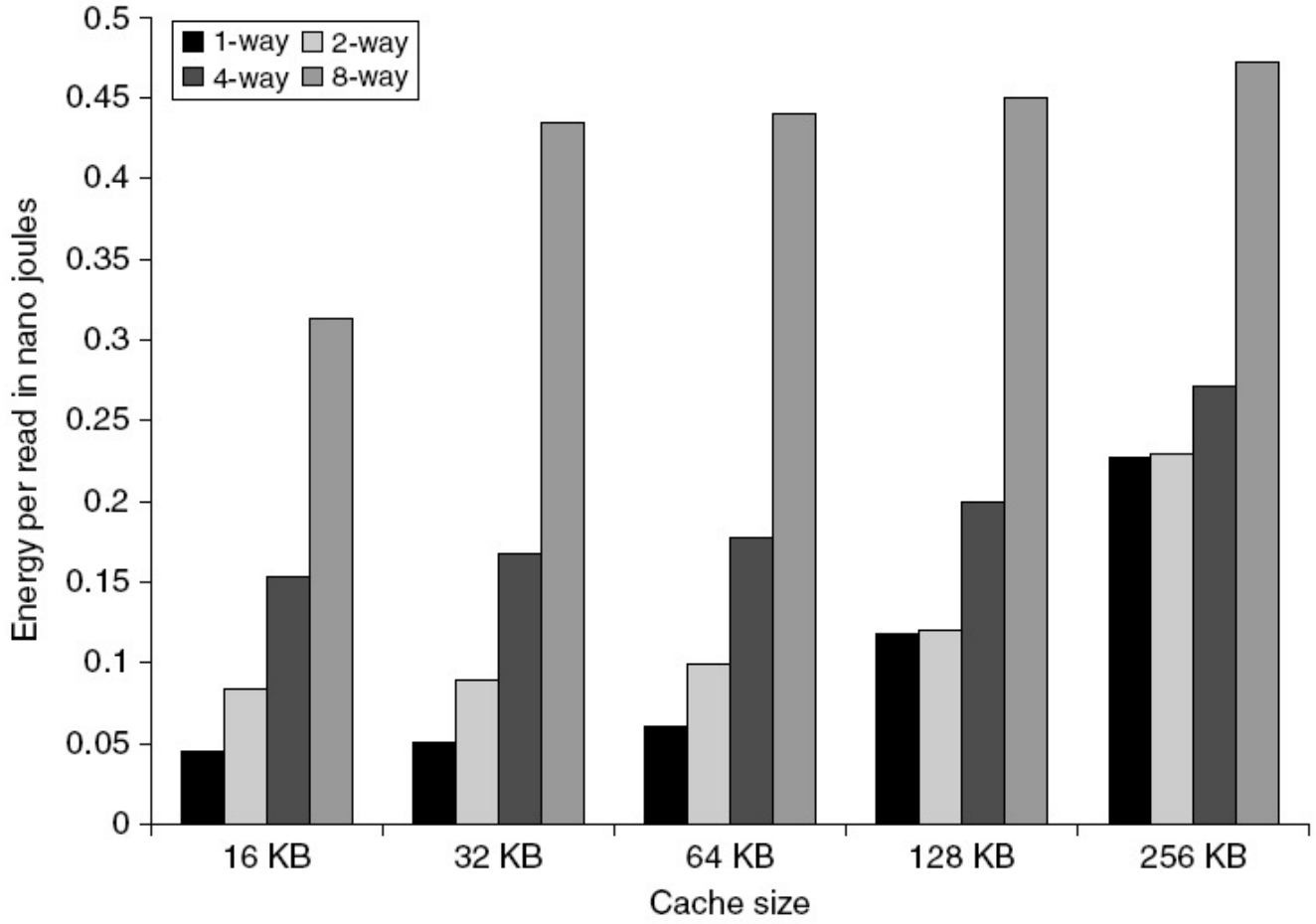
It's important to see the average behavior. In this case Random performs slightly worse than LRU, but it is the simplest solution, so might be the right choice

L1 Size and Associativity



Access time vs. size and associativity

L1 Size and Associativity



Energy per read vs. size and associativity

Q4: What happens on a write?

Cache hit:

- *write through:* write both cache & memory
  - generally higher traffic but simplifies cache coherence
- *write back:* write cache only (memory is written only when the entry is evicted)
  - a dirty bit per block can further reduce the traffic

Cache miss:

- *no write allocate:* only write to main memory
- *write allocate (aka fetch on write):* fetch into cache

Common combinations:

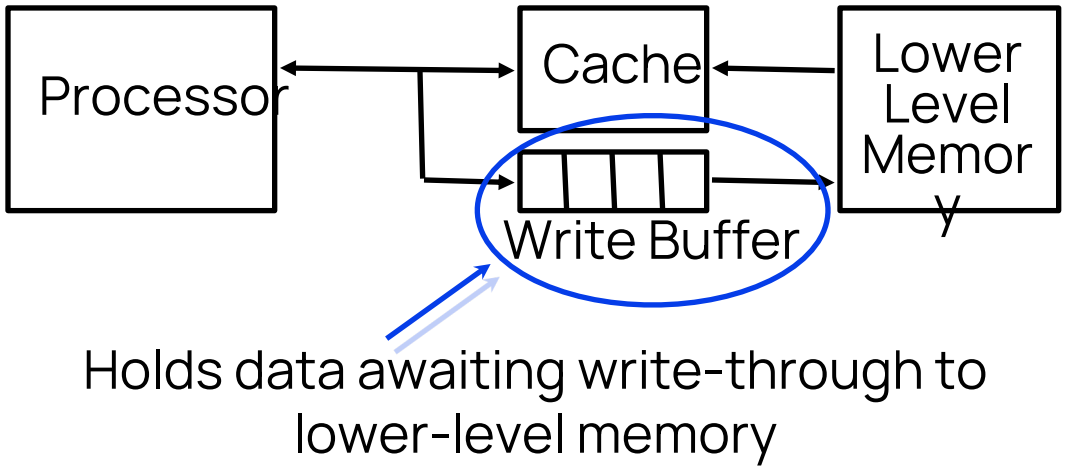
- **write through** with **no write allocate**
- **write back** with **write allocate**

Q4: What happens on a write?

	Write-Through	Write-Back
Policy	Data written to cache block also written to lower-level memory	Write data only to the cache Update lower level when a block falls out of the cache
Debug	Easy	Hard
Do read misses produce writes?	No	Yes
Do repeated writes make it to lower level?	Yes	No

Additional option -- let writes to an un-cached address allocate a new cache line ("write-allocate").

Write Buffers for Write-Through Caches



Q. Why a write buffer ?

A. So CPU doesn't stall

Q. Why a buffer, why not just one register ?

A. Bursts of writes are common.

Q. Are Read After Write (RAW) hazards an issue for write buffer?

A. Yes! Drain buffer before next read, or check write buffers for match on reads

Remember

The cache is faster than memories at lower levels of the hierarchy ⇨ **hit time** much less than the time requested to access memories at lower levels (main factor of **miss penalty**)

Miss penalty derives mainly from technology

High **Hit rate** ⇨ **average access time** near to **hit time**

How do we compute the
AVERAGE ACCESS TIME?

How memories affect system performance

Memory stall cycles: number of cycles in which the CPU is not working (***stalled***) waiting for memory access

Simplified assumptions:

- The cycle time includes the time necessary to manage a cache hit
- During a cache miss, the CPU is stalled

$$CPU\_execution\_time = (CPU\_clock\_cycles + Memory\_stall\_cycles) * clock\ cycle\ time$$

$$Memory\_stall\_cycles = Number\_of\_misses * miss\_penalty$$

A more detailed analysis

Memory_stall_cycles =
IC*(Misses/Instruction)*Miss_penalty =
IC*Reads_per_instruction*Read_miss_rate*Read_miss_penalty +
IC*Writes_per_instruction*Write_miss_rate*Write_miss_penalty

We can simplify by averaging reads and writes
Memory_stall_cycles = IC*(memory_accesses/instruction)*miss_rate*miss_penalty

A different figure of merit
Misses/instruction = miss_rate* (memory_accesses/instruction)

Independent of the hardware implementation, dependent on the architecture (related to the average number of memory accesses per instruction)

Average access time

$T_A = hit\_rate*hit\_time+miss\_rate*miss\_time$

$= hit\_rate*hit\_time + miss\_rate*(hit\_time + miss\_penalty)$

$= hit\_time*(hit\_rate+miss\_rate) + miss\_rate*miss\_penalty$

$= hit\_time + miss\_rate*miss\_penalty$.

Architectural choices may reduce the miss rate: ⇨ T_A closer to *hit_time*

5 Basic Cache Optimizations

Reducing Miss Rate

1. Larger Block size (compulsory misses)
2. Larger Cache size (capacity misses)
3. Higher Associativity (conflict misses)

Reducing Miss Penalty

4. Multilevel Caches

Reducing hit time

5. Giving Reads priority over Writes
    - E.g., Read completes before writes in write buffer

The cache design space

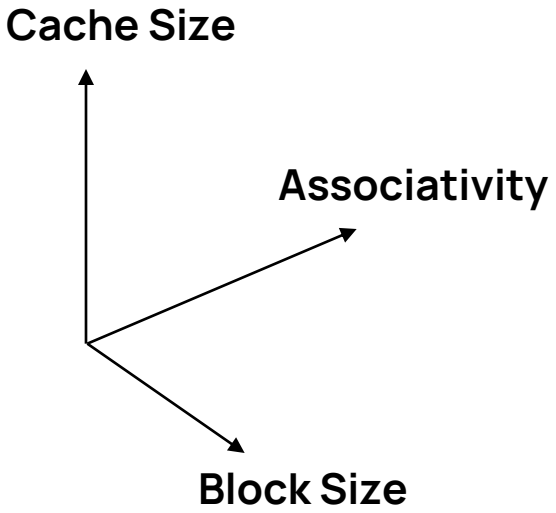
Several interacting dimensions

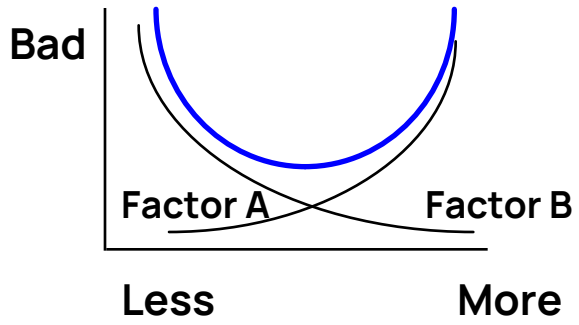
- cache size
- block size
- associativity
- replacement policy
- write-through vs write-back

The optimal choice is a compromise

- depends on access characteristics
  - workload
  - use (I-cache, D-cache, TLB)
- depends on technology / cost

Simplicity often wins





Memory Technology

- Performance metrics
  - Latency is a concern of cache
  - Bandwidth is a 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 bits
- 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
- One 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:

- Memory capacity should grow linearly with processor speed
- Unfortunately, memory capacity and speed have not kept pace with processors

Some optimizations:

- Multiple accesses to the 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

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

Virtual Memory

Protection via virtual memory

- Keeps processes in their own memory space

Role of architecture:

- Provide user mode and supervisor mode
- Protect certain aspects of the CPU state
- Provide mechanisms for switching between user mode and supervisor mode
- Provide mechanisms to limit memory accesses
- Provide TLB to translate addresses

Virtual Machines

- Supports isolation and security
- Sharing a computer among many unrelated users
- Enabled by raw speed of processors, making the overhead more acceptable

- Allows different ISAs and operating systems to be presented to user programs
  - “System Virtual Machines”
  - SVM software is called “virtual machine monitor” or “hypervisor”
  - Individual virtual machines run under the monitor are called “guest VMs”

Impact of VMs on Virtual Memory

Each guest OS maintains its own set of page tables

- VMM adds a level of memory between physical and virtual memory called “real memory”
- VMM maintains a **shadow page table** that maps guest virtual addresses to physical addresses
  - Requires VMM to detect guest’s changes to its own page table
  - Occurs naturally if accessing the page table pointer is a privileged operation

Questions?