# Chapter 2

## Memory Hierarchy Design

## Part 1: The Basics

"Ideally one would desire an indefinitely large memory capacity such that any particular … word would be immediately available. … We are … forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible."

– A. W. Burks, H. H. Goldstine, and J. von Neumann, *Preliminary Discussion of the Logical Design of an Electronic Computing Instrument (1946)*

# Acknowledgements

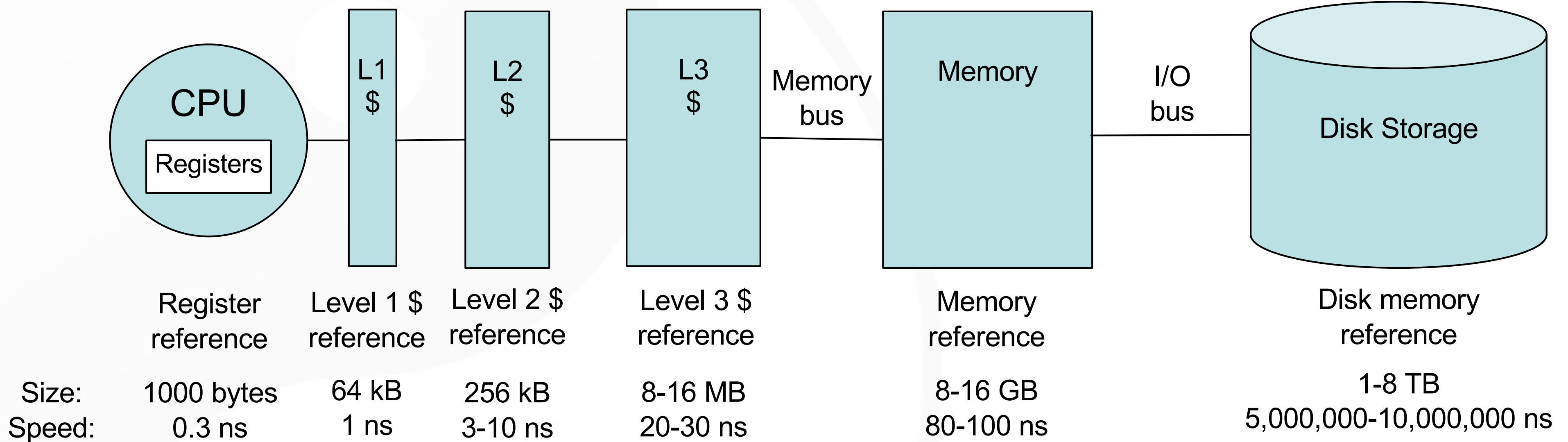

- Thanks to many sources for slide material
  - © 1990 Morgan Kaufmann Publishers, © 2001-present Elsevier
    Computer Architecture: A Quantitative Approach by J. Hennessy & D. Patterson
  - © 1994 Morgan Kaufmann Publishers, © 2001-present Elsevier
    Computer Organization and Design by D. Patterson & J. Hennessy
  - © 2002 K. Asinovic & Arvind, MIT
  - © 2002 J. Kubiatowicz, University of California at Berkeley
  - © 2006, © 2010 No Starch Press for Inside the Machine by J. Stokes
  - © 2007 W.-M. Hwu & D. Kirk, University of Illinois & NVIDIA
  - © 2007-2010 J. Owens, University of California at Davis
  - © 2010 CRC Press for Introduction to Concurrency in Programming Languages by M. Sottile, T. Mattson, and C. Rasmussen
  - © 2017, IBM POWER9 Processor Architecture by Sadasivam et al., IBM
  - © 2016, © 2019 POWER9 Processor User's Manual, IBM
  - © The OpenPOWER Foundation

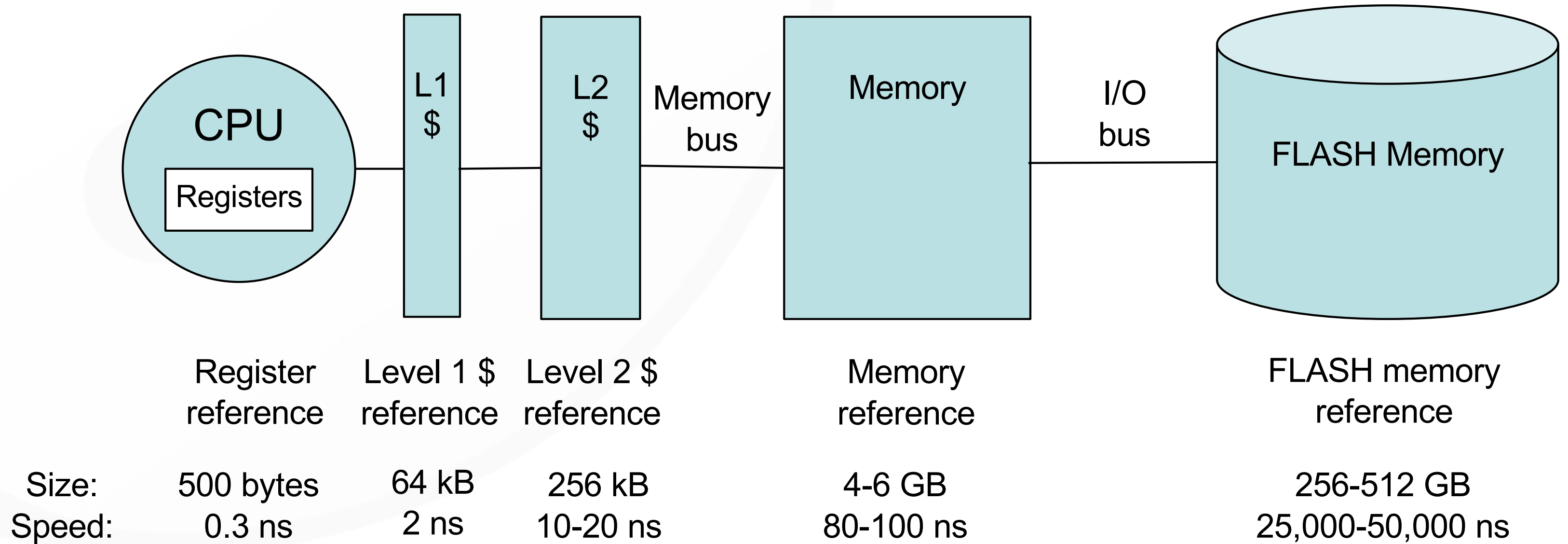
# Introduction

- What do programmers want?
  - *Unlimited amounts of memory* with *low latency*
- Problem:
  Fast memory *more expensive per bit* than slower memory
- Solution: Organize memory system into a hierarchy
  - Entire addressable memory available in largest, slowest memory
  - Incrementally smaller and faster memories, each containing a subset of the memory below it, proceed in steps up toward the processor
- Outcome: Deliver illusion of a fast & large memory to CPU
  - How? *Temporal locality and spatial locality* ensure that nearly all references can be found in smaller memories

# Memory Hierarchy

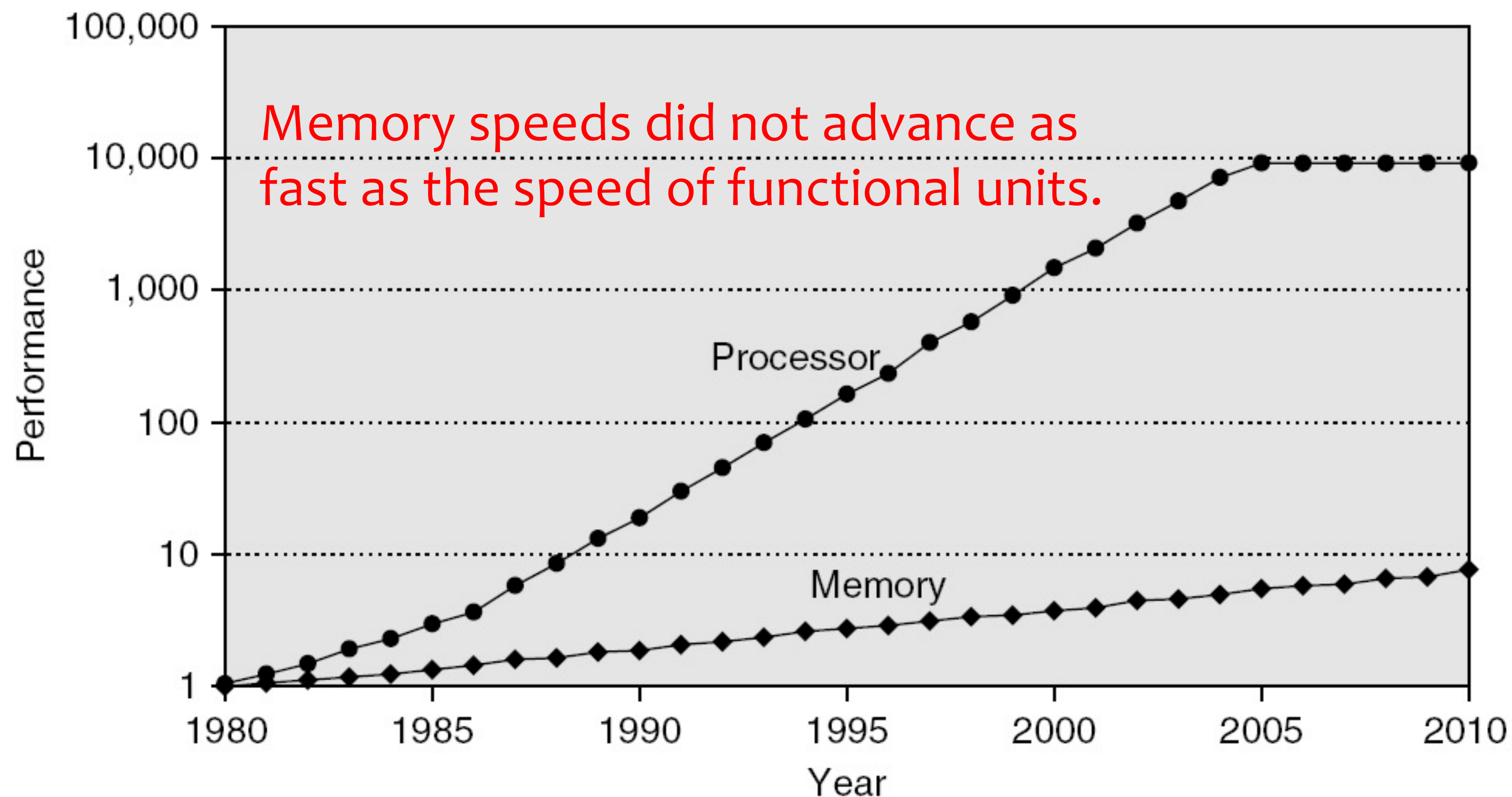
Key: $ = cache



# Cache: Multicore Feature of Interest

- Introduced in the 1960s as a way to overcome the "memory wall"




Consequence: Processor outruns memory, leading to decreased utilization

# Idle Time

- What happens when you go out to main memory?
  *Idle time.*
  - Decreased utilization = less work per unit time
  - Idle time = time doing nothing = $$$ wasted
- Overcoming the "Memory Wall"
  - Caches were *not* the sole fix for idle time.
  - Early Days: Preemptive multitasking and time sharing were actually the dominant methods.

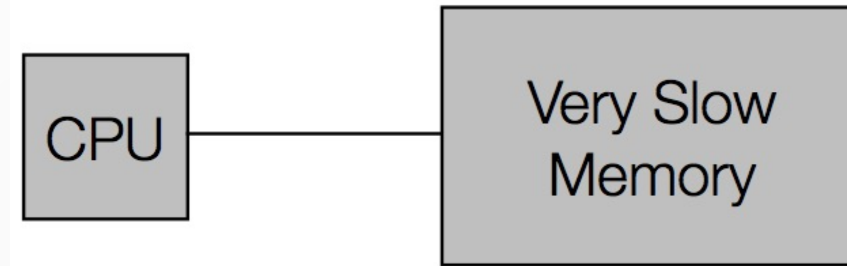
    ... but every program inevitably must go out to memory
    ... not always enough jobs to swap in while others wait for memory
    … also, do you really want to be preempted every time you fetch data?
  - So, caches important (for now :-) …

# Memory Hierarchy Design

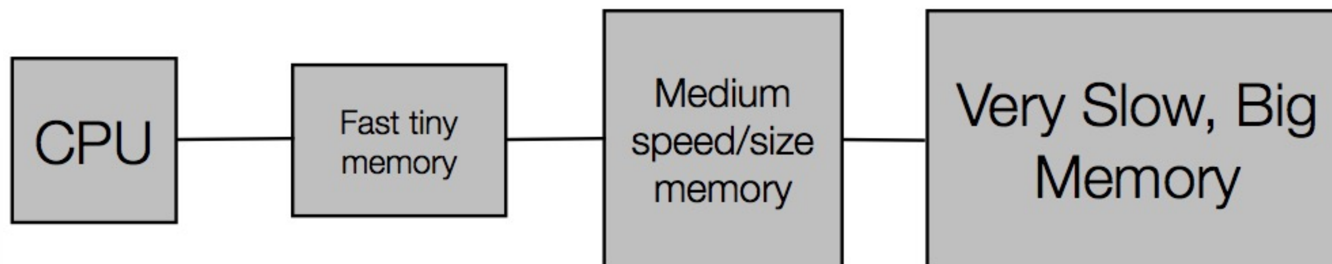
- Memory hierarchy design becomes more crucial with recent multi-core processors
  - Aggregate peak bandwidth grows with # cores:
    - Intel Core i7 can generate two references per core per clock
    - Four cores and 3.2 GHz clock
      - How much memory bandwidth needed?
      - Hint: 64-bit data references & 128-bit instruction references
    - DRAM bandwidth is only 25 GB/s (circa 2011-2012)
  - How to bridge the gap?
    - Multi-port, pipelined caches
    - Two levels of cache per core
    - Shared third-level cache on chip

# Caches: An Overview

- Traditional Single-Level Memory

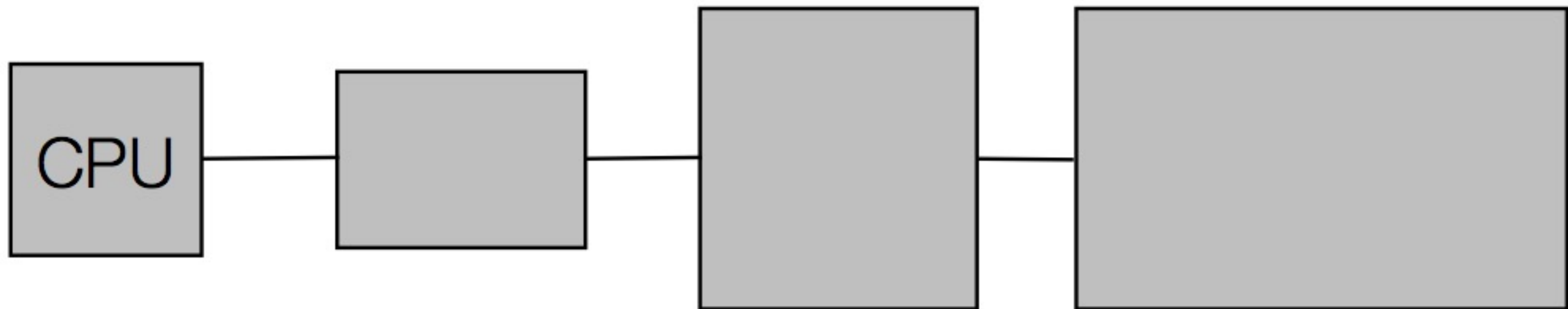



- Multiple Memory Levels



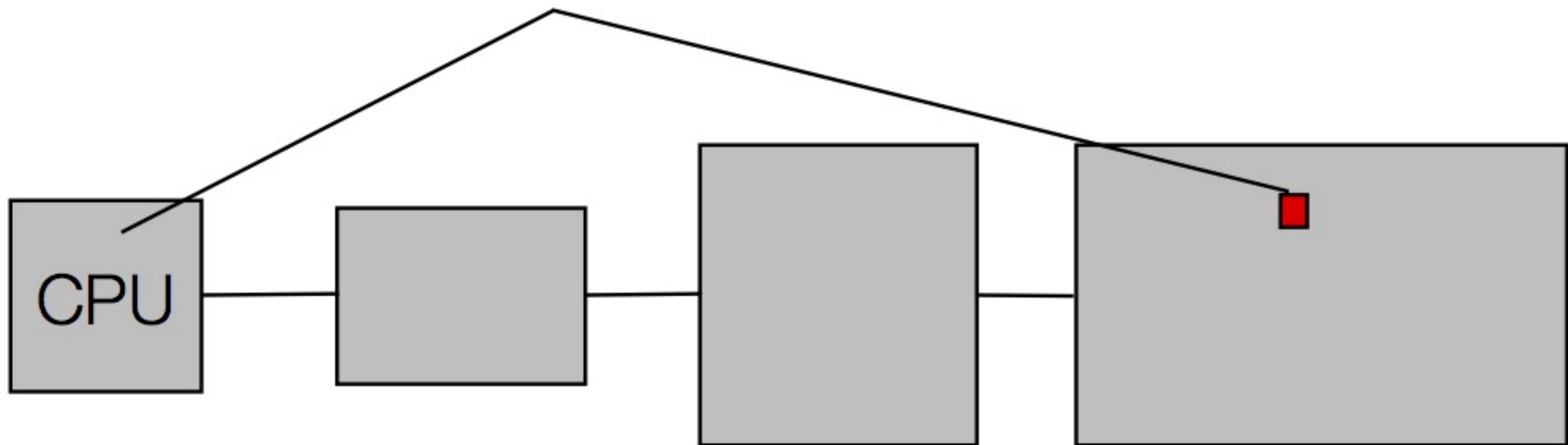
# Caches: In Action

- Access a location in memory
- Copy the location and its neighbors into the faster memories closer to the CPU.



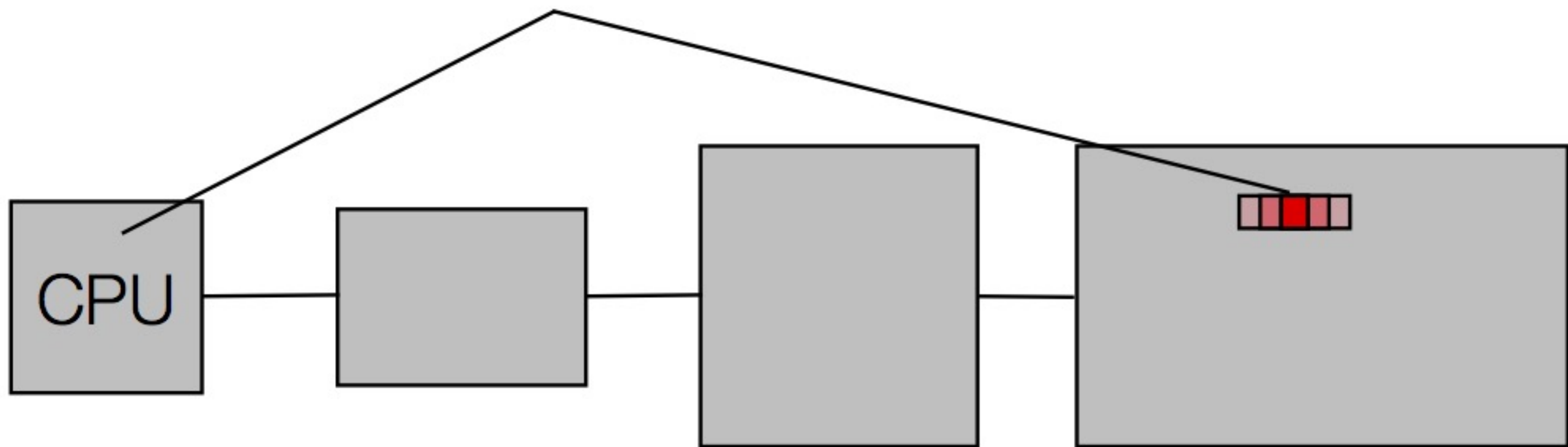
# Caches: In Action

- Access a location in memory
- Copy the location and its neighbors into the faster memories closer to the CPU.



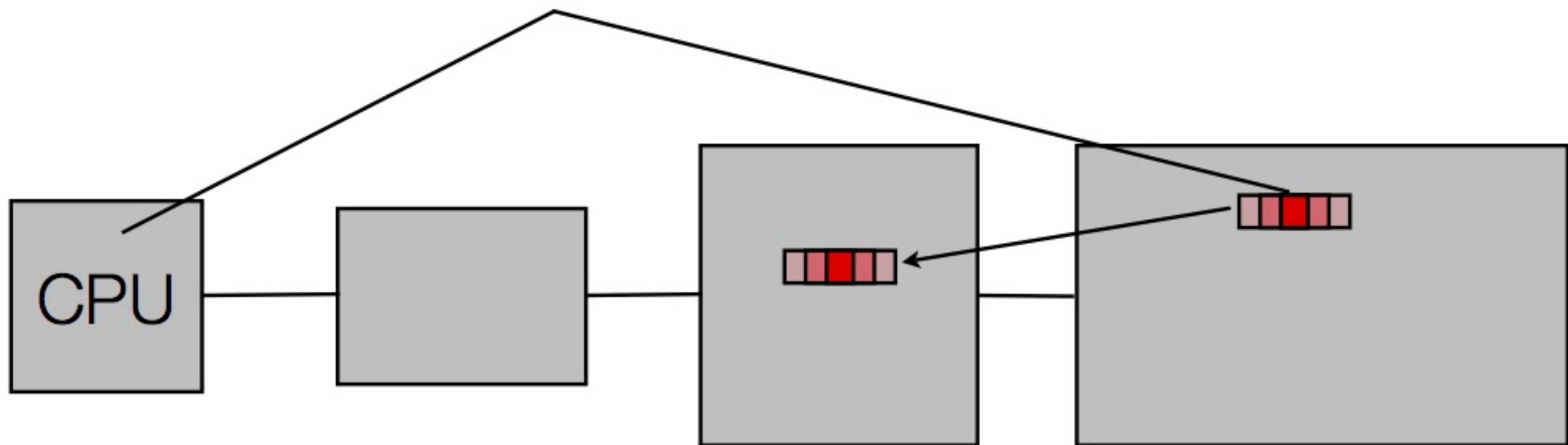
# Caches: In Action

- Access a location in memory
- Copy the location and its neighbors into the faster memories closer to the CPU.



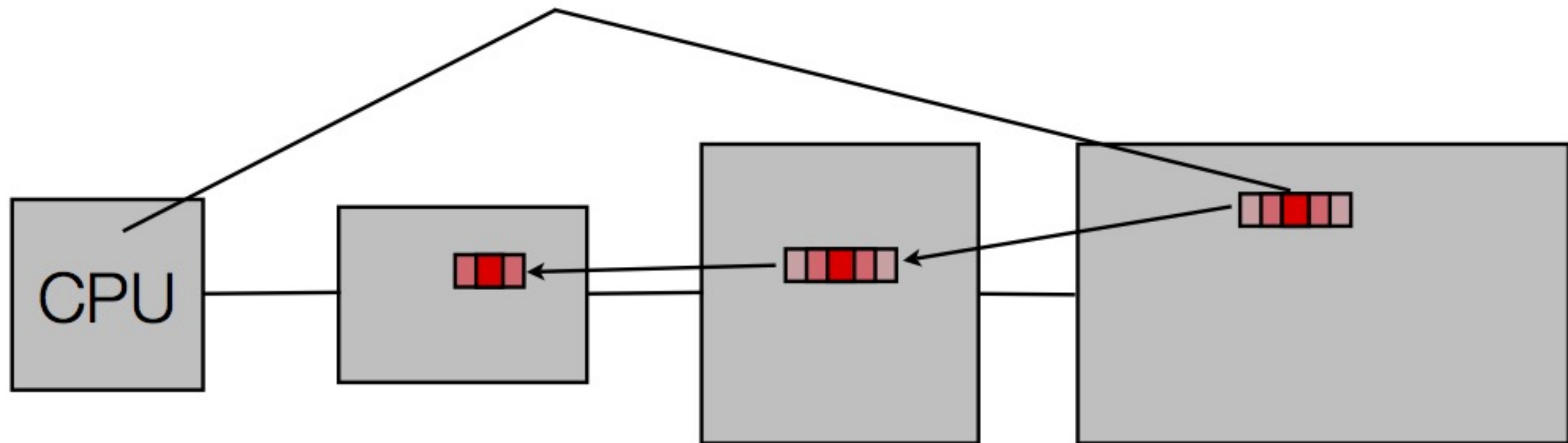
# Caches: In Action

- Access a location in memory
- Copy the location and its neighbors into the faster memories closer to the CPU.



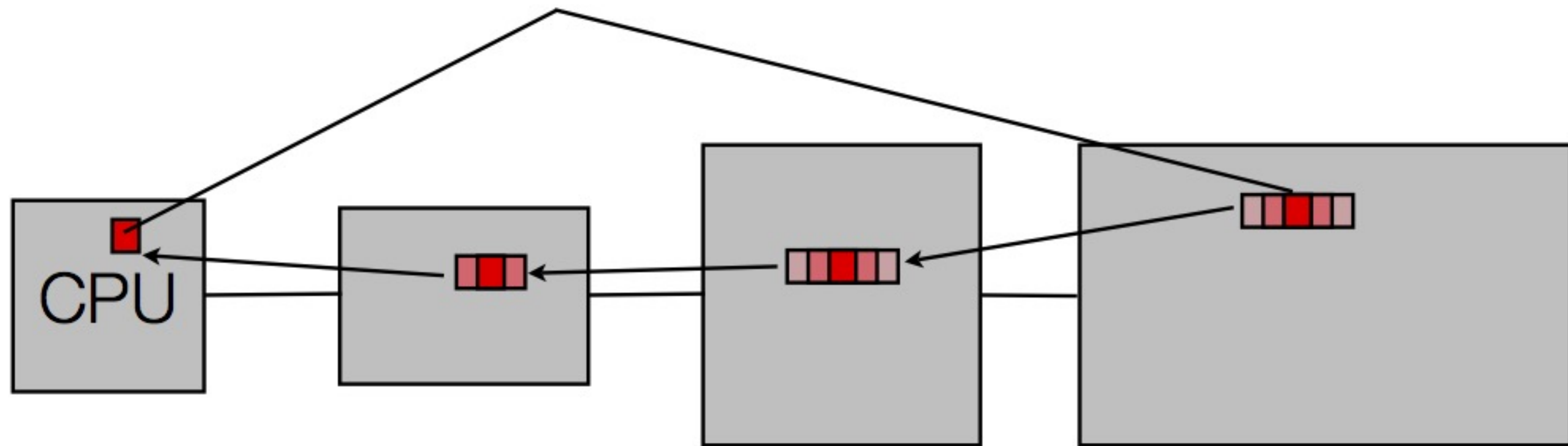
# Caches: In Action

- Access a location in memory
- Copy the location and its neighbors into the faster memories closer to the CPU.



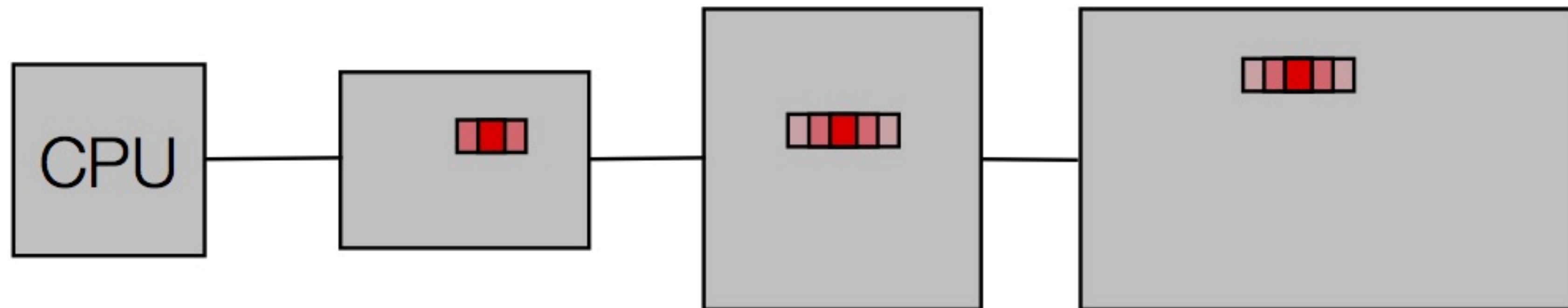
# Caches: In Action

- Access a location in memory
- Copy the location and its neighbors into the faster memories closer to the CPU.



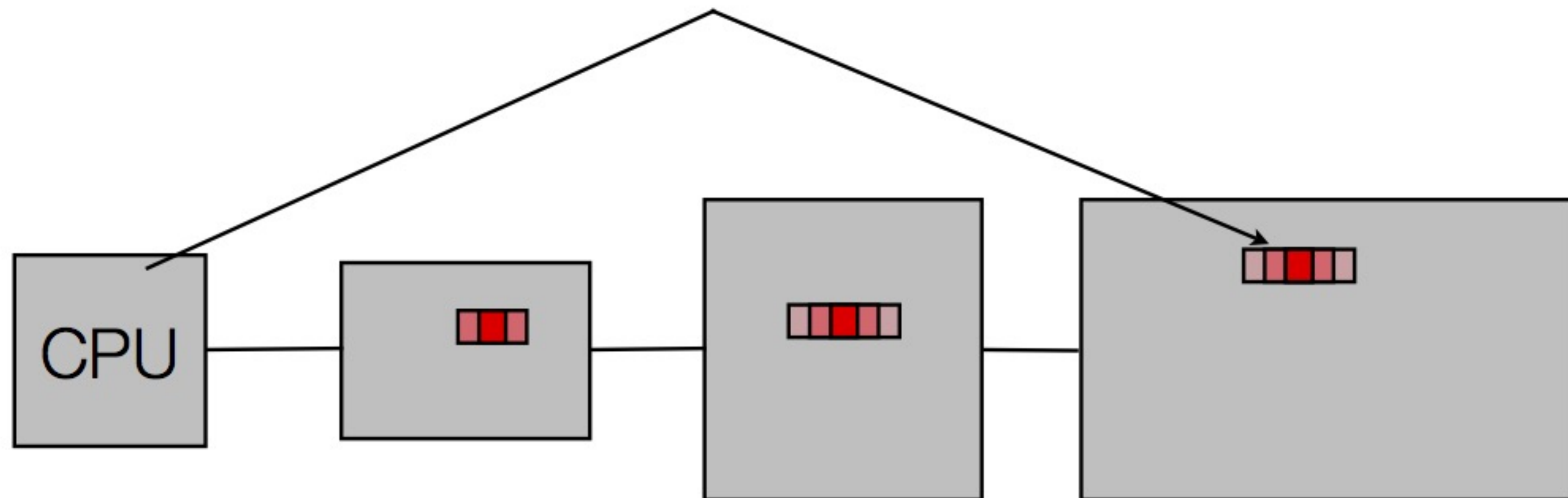
# Caches: In Action

- Next time you access memory, if you already pulled the address into one of the cache levels in a previous access, the value is provided from the cache.



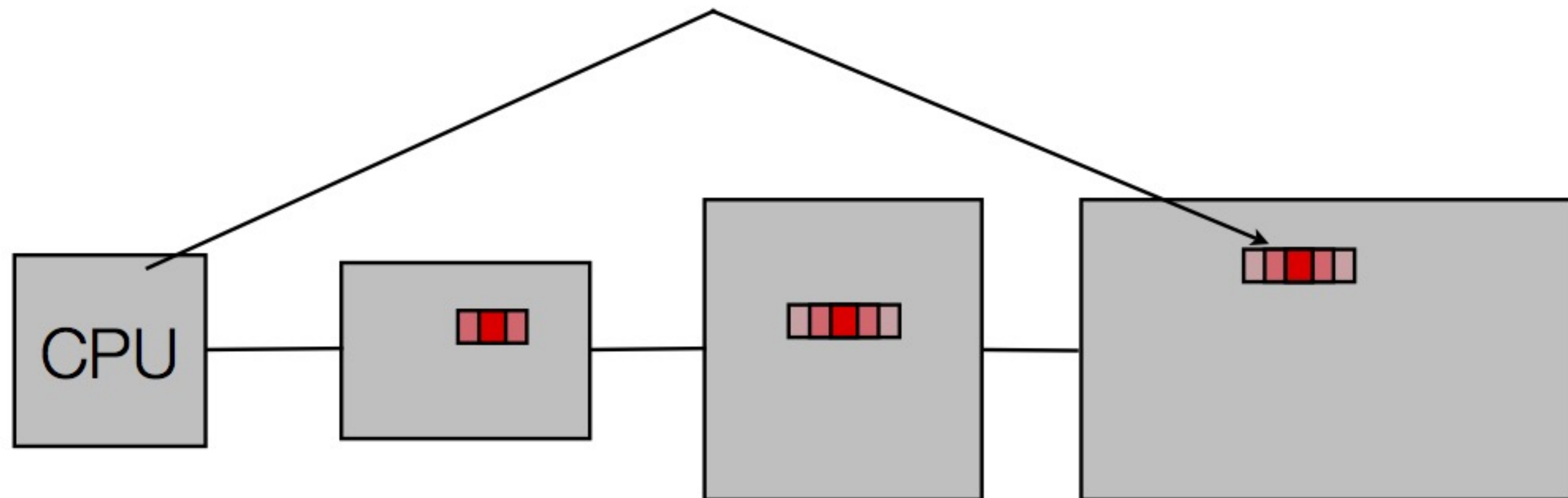
# Caches: In Action

- Next time you access memory, if you already pulled the address into one of the cache levels in a previous access, the value is provided from the cache.



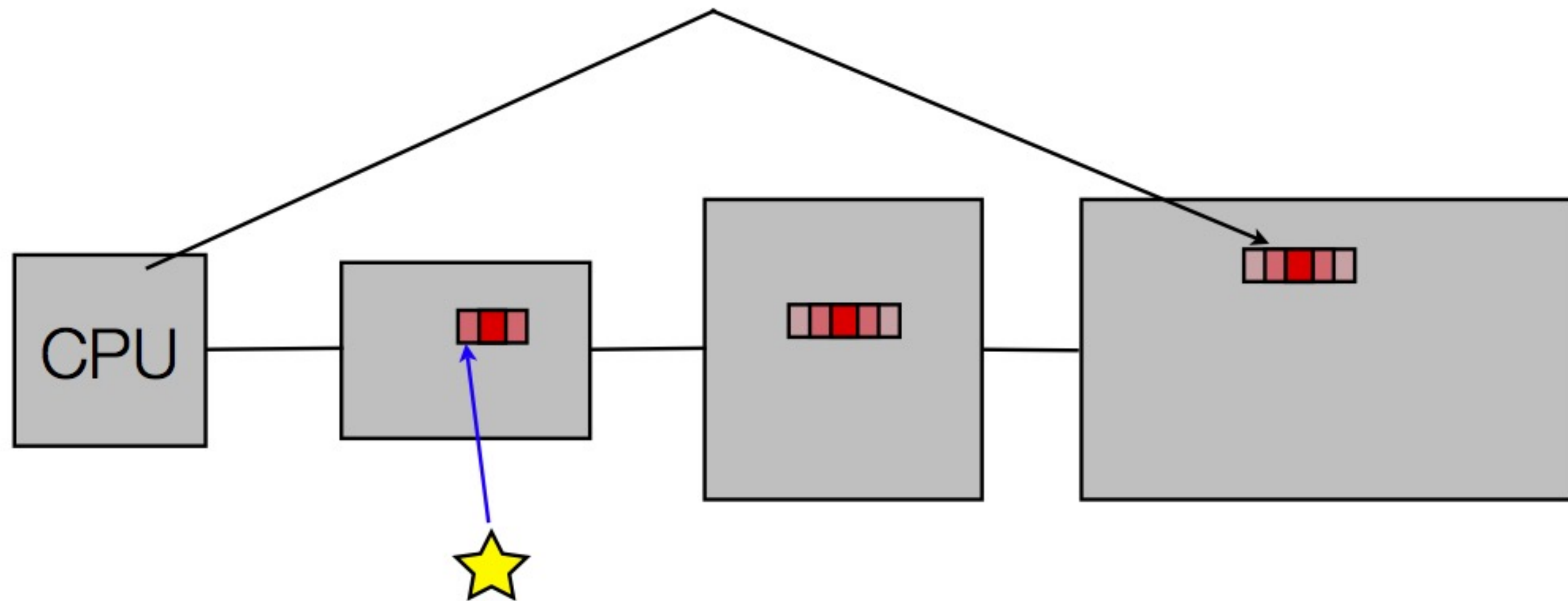
# Caches: In Action

- Next time you access memory, if you already pulled the address into one of the cache levels in a previous access, the value is provided from the cache.



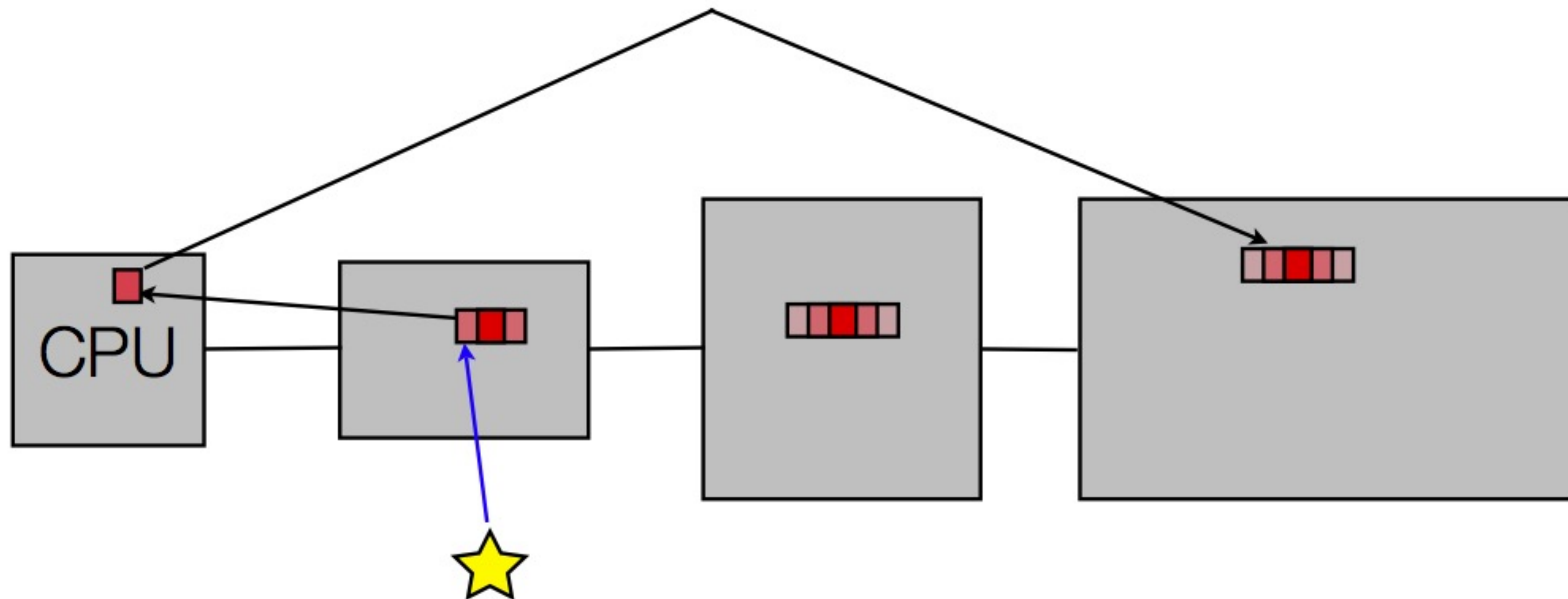
# Caches: In Action

- Next time you access memory, if you already pulled the address into one of the cache levels in a previous access, the value is provided from the cache.



# Caches: In Action

- Next time you access memory, if you already pulled the address into one of the cache levels in a previous access, the value is provided from the cache.

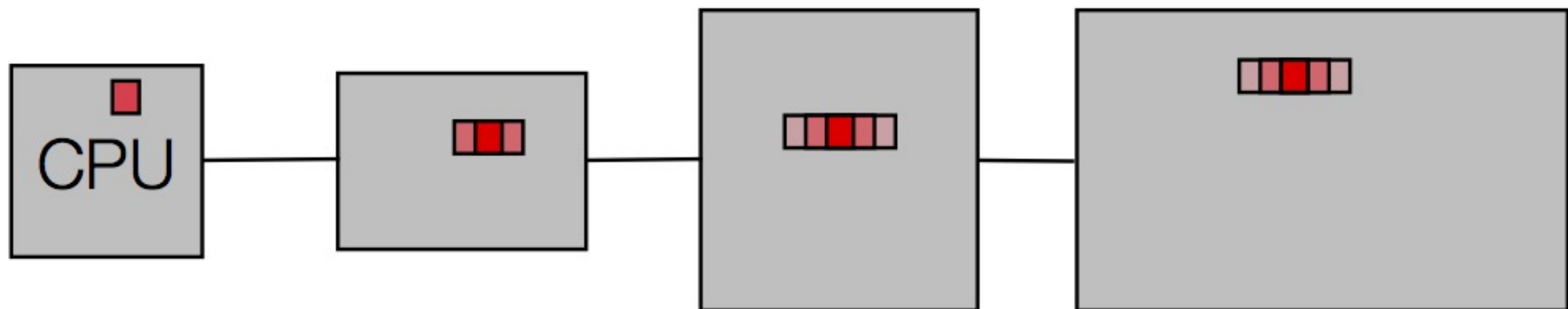


# Why Do Caches Work? Locality

- Spatial and Temporal
  - Locations near each other in space (address) are highly likely to be accessed near each other in time.
- Cost?
  - A high cost for one access but amortize this out with faster accesses afterwards.
- Burden?
  - The machine makes sure that memory is kept consistent. If a part of cache must be reused, the cache system writes the data back to main memory before overwriting it with new data.
  - Hardware cache design deals with managing mappings between the different levels and deciding when to write back down the hierarchy.

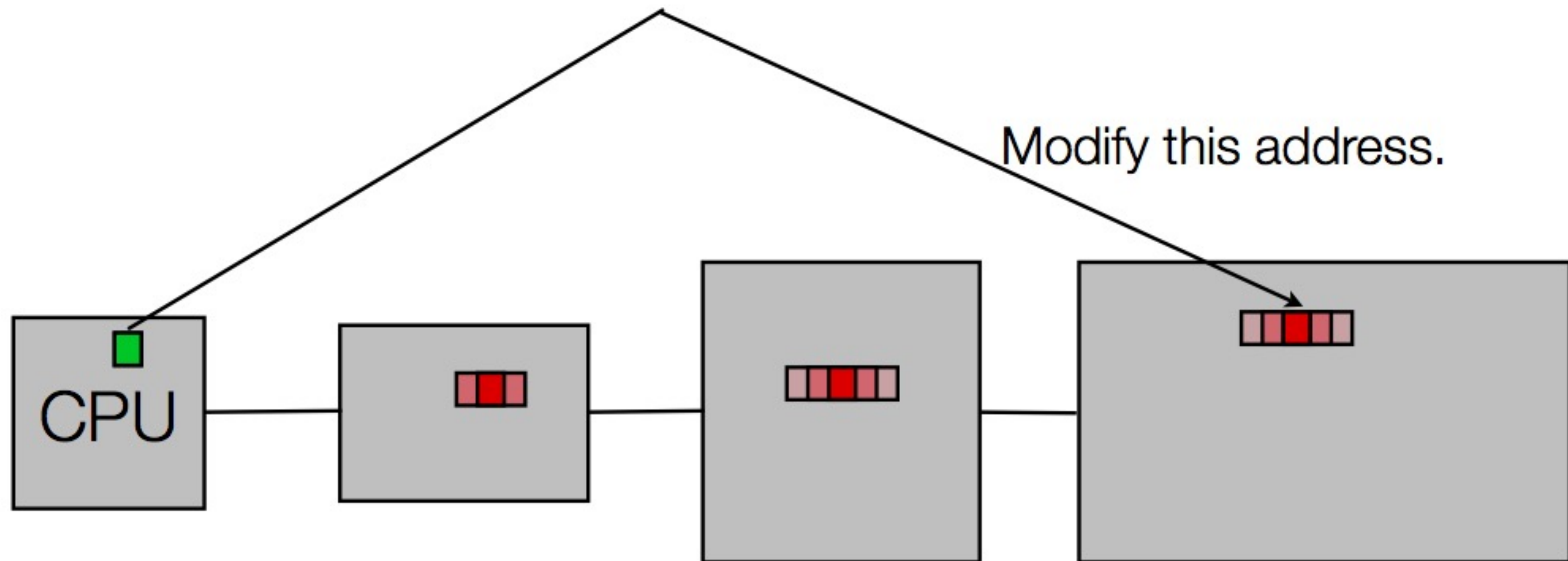
# Caches: Memory Consistency?

- What happens when you modify something in memory?



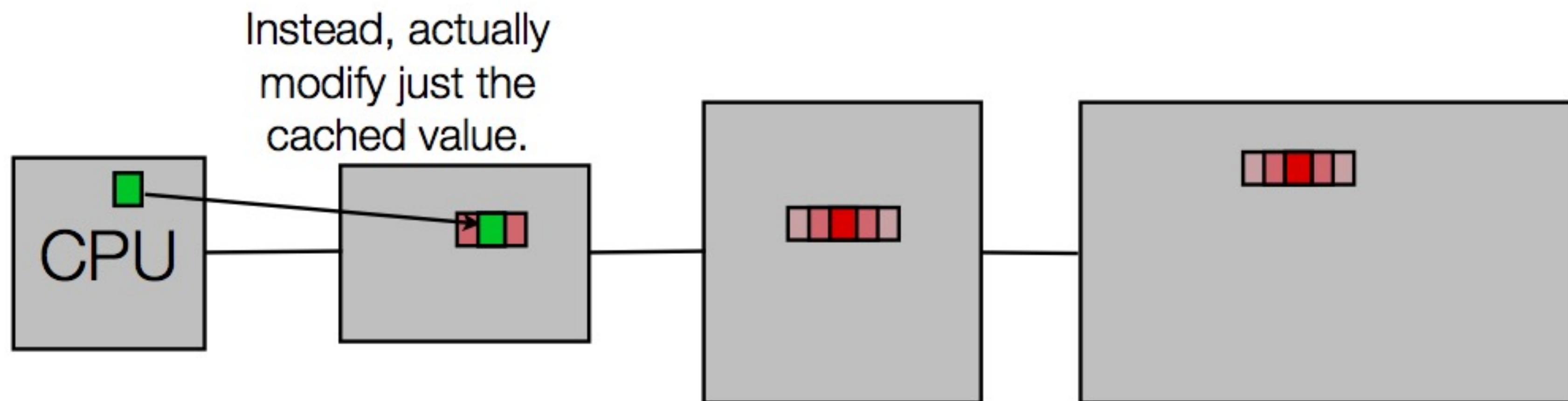
# Caches: Memory Consistency?

- What happens when you modify something in memory?



# Caches: Memory Consistency?

- What happens when you modify something in memory?
- Writes to memory become cheap. Only go to slow memories when needed. Called **write-back memory.**

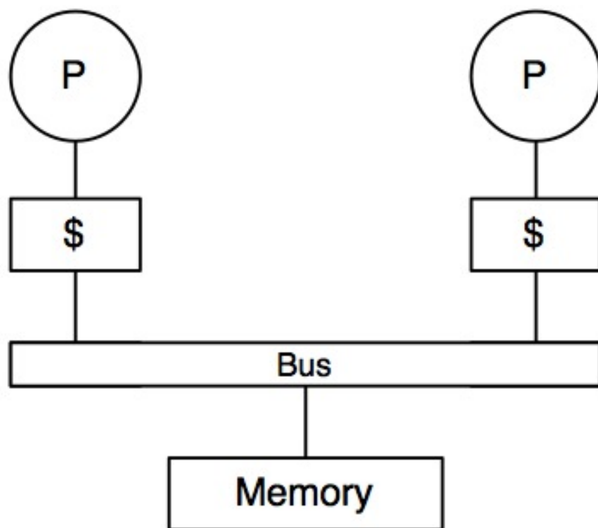


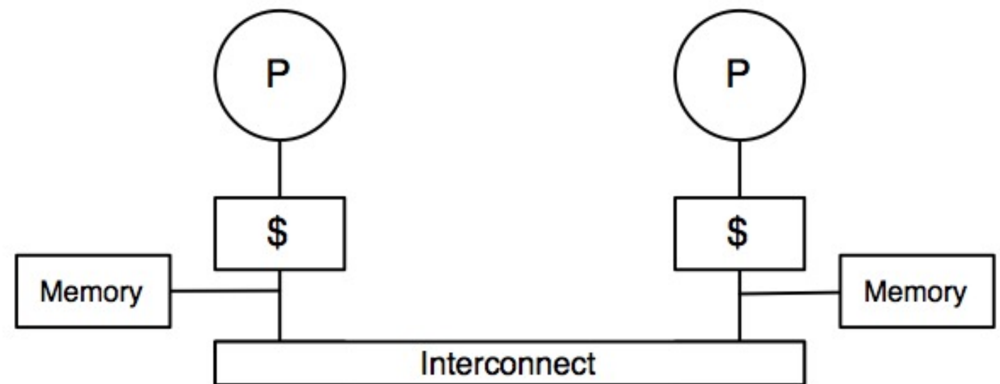
# Caches: Memory Consistency?

- Eventually written values must make it back to the main store.
- When?
  - Typically, when a cache block is replaced due to a cache miss, where new data must take the place of old.
- The programmer does NOT see this.
  - Hardware takes care of all this … but things can go wrong very quickly when you modify this model.
    - Example: Cell Broadband Engine, Tilera, and so on.

# Common Memory Models

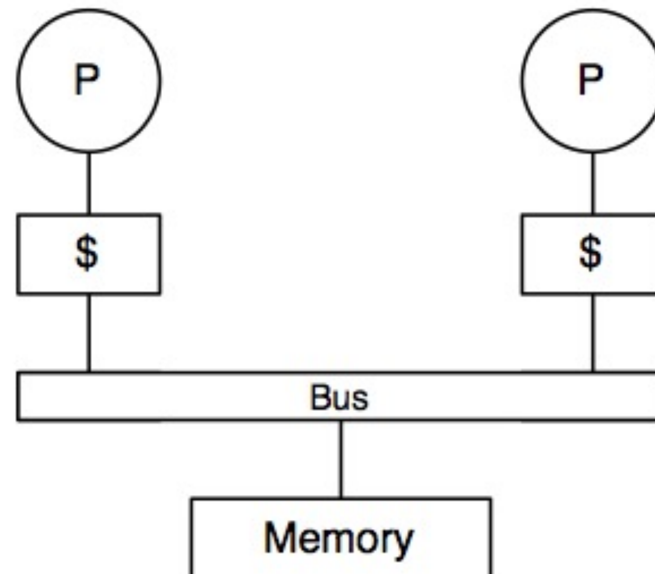
- Shared Memory Architecture
- Distributed Memory Architecture



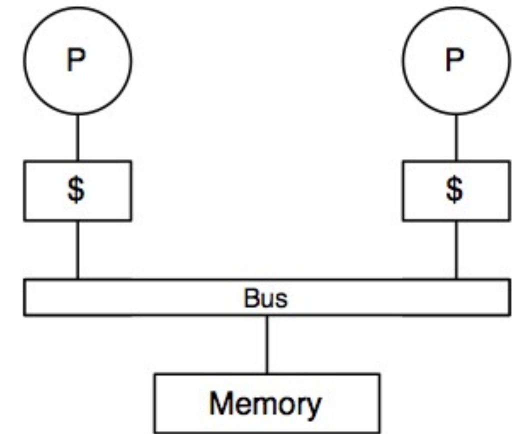



# Memory Models: Shared Memory

- Before
  - Only one processor has access to modify memory.
- How do we avoid problems when multiple cache hierarchies see the same memory?

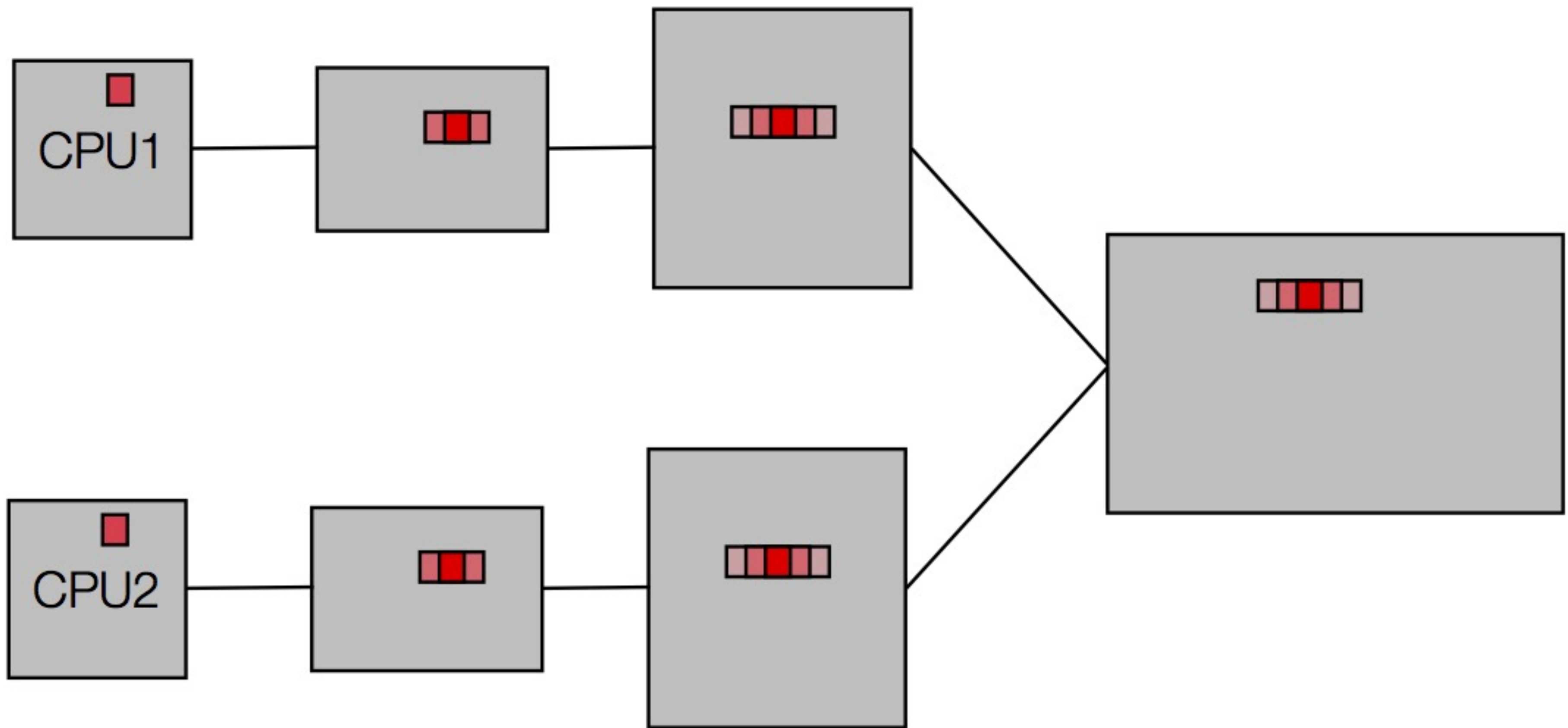


# Caching Issues

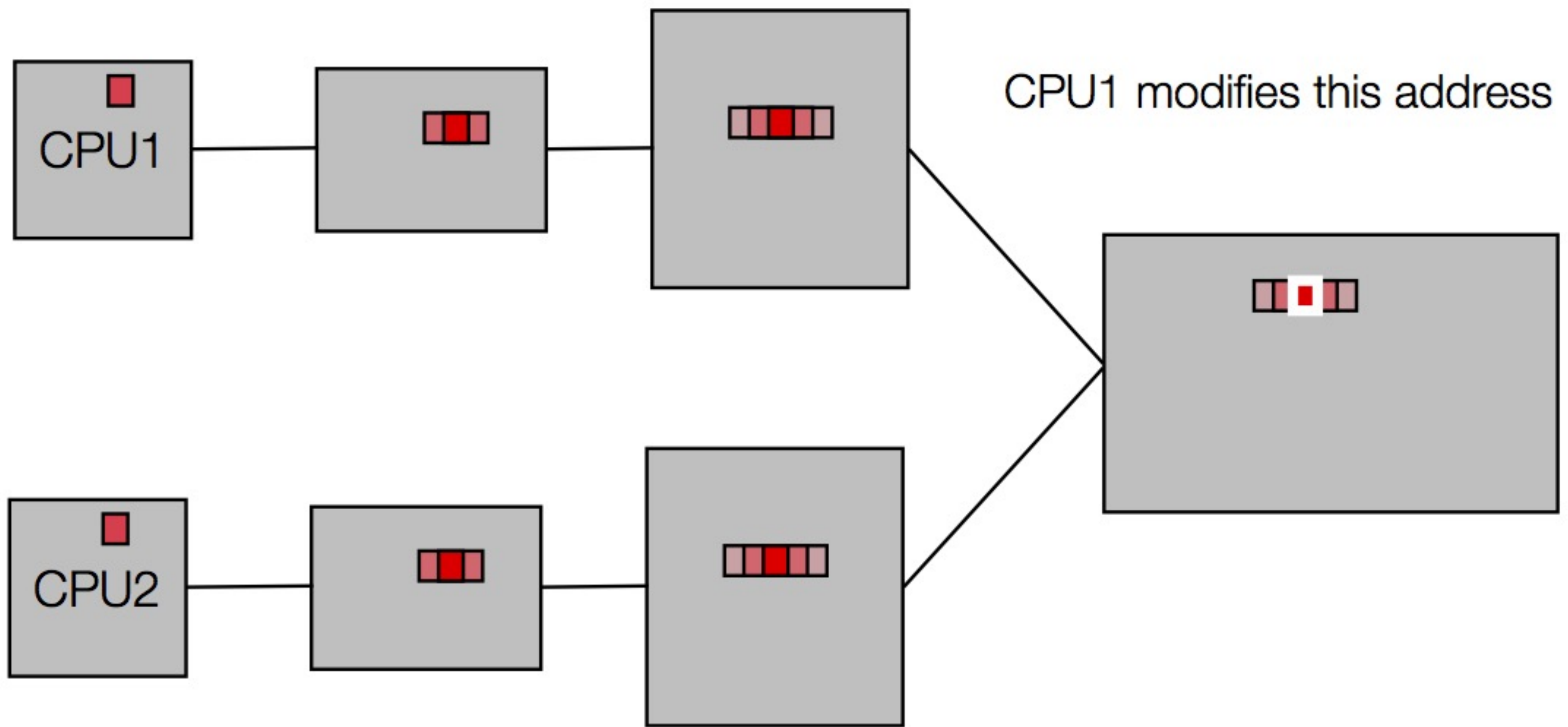



- Assume two processors load locations that are neighbors, so data is replicated in the local processor caches.
- Now, let one processor modify a value.
- The memory view is now inconsistent. One processor sees one version of memory, the other sees a different version.
- How do we resolve this *in hardware* such that the advantages of caches are still seen by application developers in terms of performance while ensuring a consistent (or coherent) view of memory?

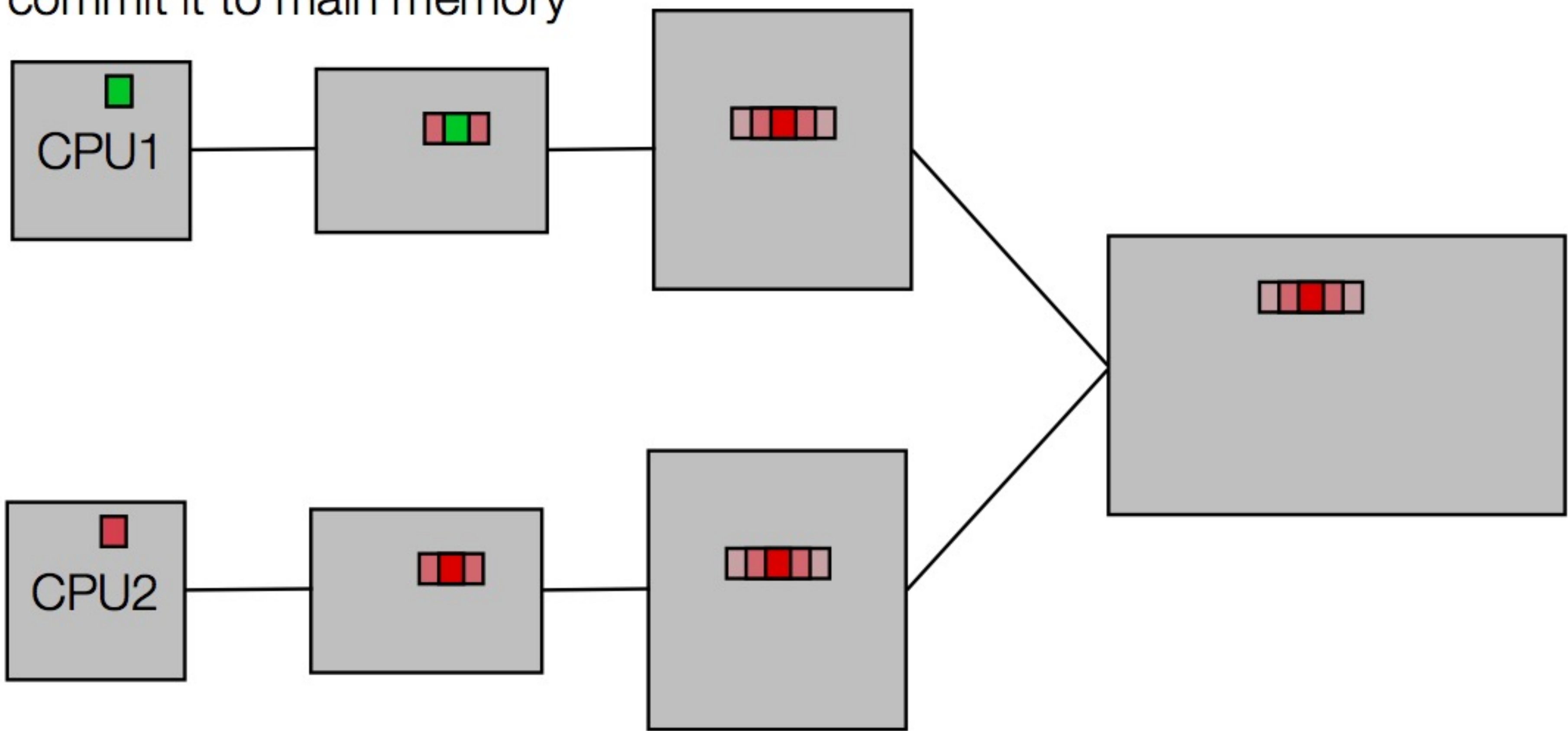
# Caching Issues



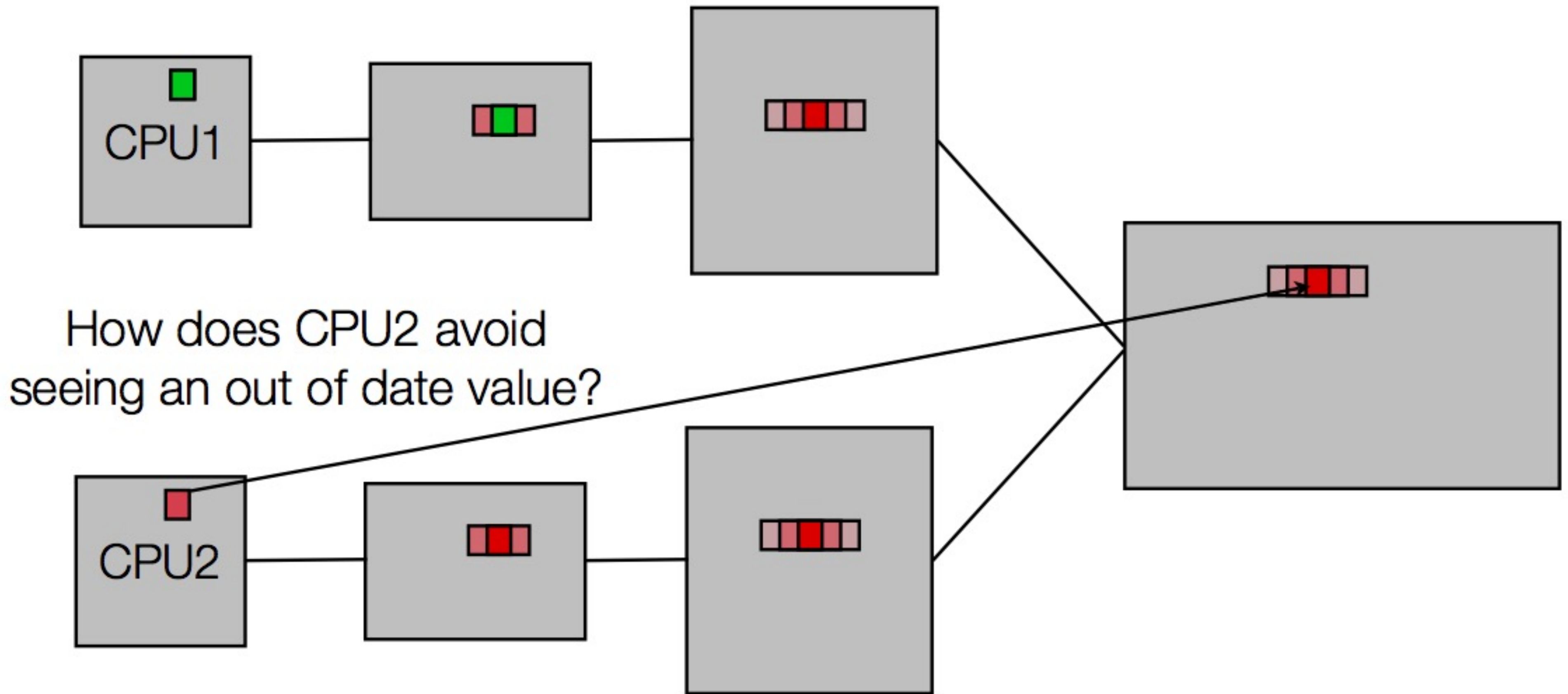
# Caching Issues



# Caching Issues



# Caching Issues



# Caching: Memory Consistency?

- Easy to see “memory consistency” problem if we restrict each cache hierarchy to being isolated from the others, ***only sharing main memory***.
- Key insight
  - Make this inconsistency “go away” by making the caches aware of each other.

# What is Memory Coherence?

- Definition (Courtesy: “Parallel Computer Architecture” by Culler and Singh)
  1. Operations issued by any particular process occur in the order in which they were issued to the memory system by that process.
  2. The value returned by each read operation is the value written by the last write to that location in the serial order.
- Assumption: *The above requires a hypothetical ordering for all read/write operations by all processes into a total order that is consistent with the results of the overall execution.*
- Sequential Consistency (SC)
  - The memory coherence hardware assists in enforcing SC.

# Implicit Properties of Coherence

- The key to solving the cache-coherence problem is the hardware implementation of a *cache-coherence protocol.*
- A cache-coherence protocol takes advantage of two hardware features
  1. State annotations for each cache block (often just a couple bits per block).
  2. Exclusive access to the bus by any accessing process.

# Bus Properties

- All processors on the bus see the same activity.
- Every cache controller sees bus activity in the same *order.*
- Serialization at the bus level results from the phases that compose a bus transaction:
  - *Bus arbitration:* The bus arbiter grants exclusive access to issue commands onto the bus.
  - *Command/address:* The operation to perform ("Read", "Write"), and the address.
  - *Data:* The data is then transferred.

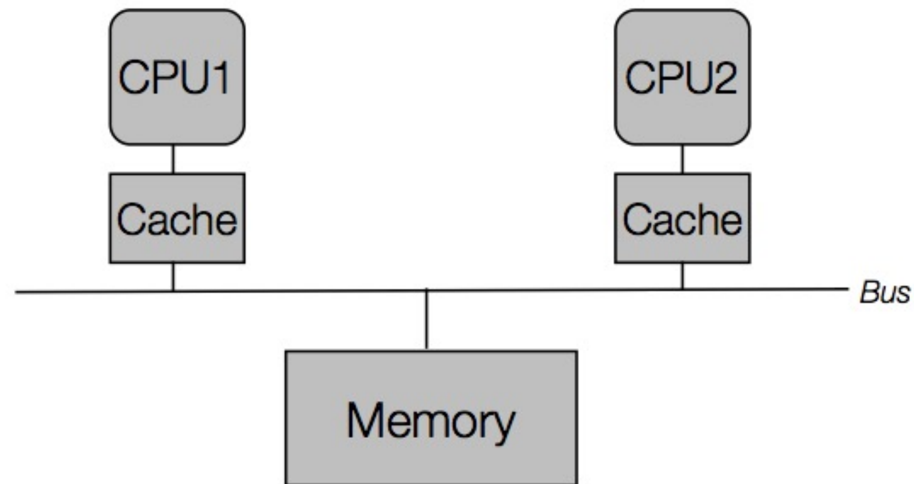
# Granularity

- Cache coherence applies at the *block level.*
- Recall that when you access a location in memory, that location *and* its neighbors are pulled into the cache(s). These are *blocks.*

*Note: To simplify the discussion, we will only consider a single level of cache. The same ideas translate to deeper cache hierarchies.*

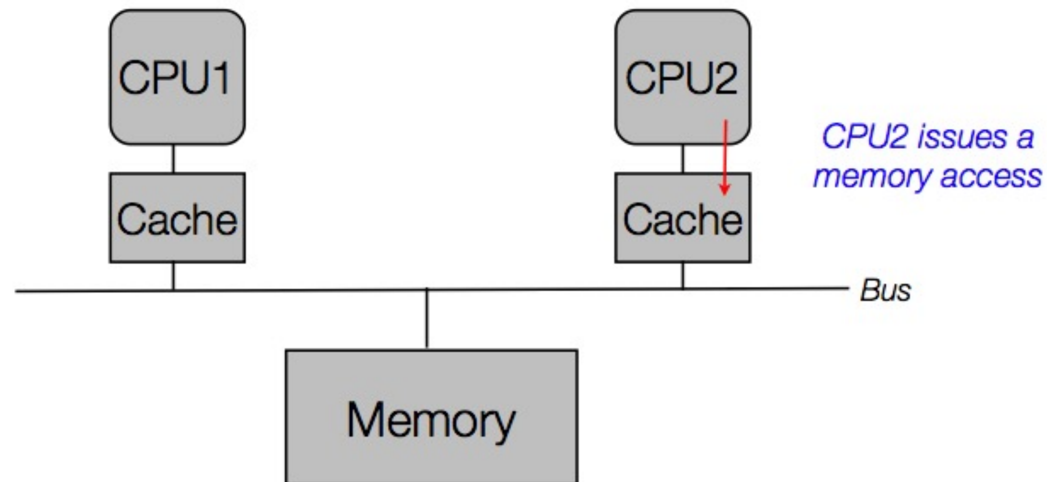
# Cache Coherency via the Bus

- Key Idea: Bus Snooping
  - All CPUs on the bus can see activity on the bus regardless of if they initiated it.



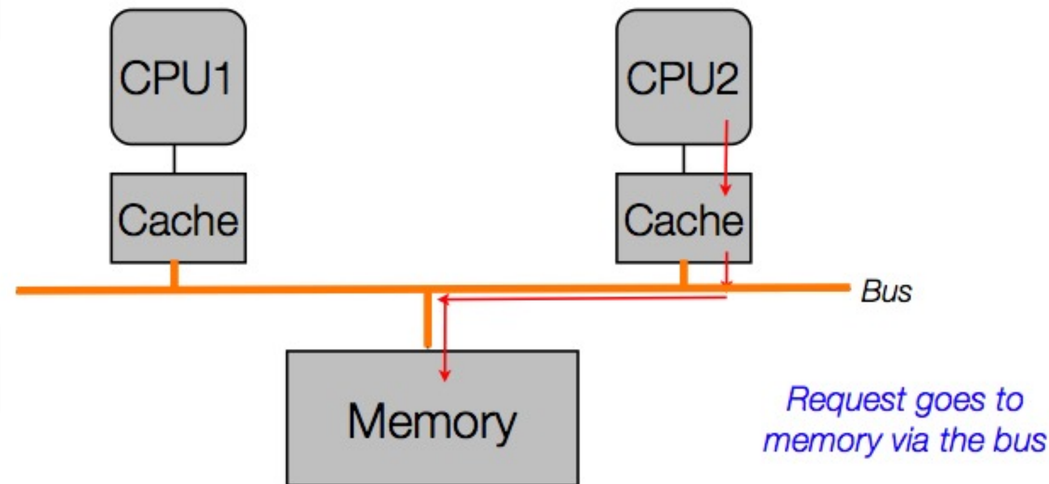
# Cache Coherency via the Bus

- Key Idea: Bus Snooping
  - All CPUs on the bus can see activity on the bus regardless of if they initiated it.



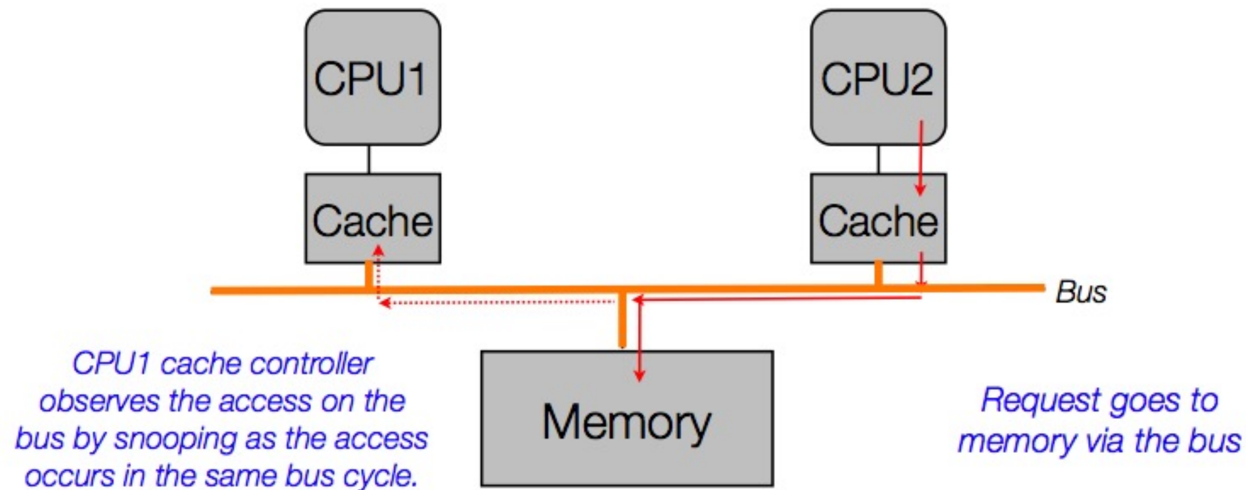
# Cache Coherency via the Bus

- Key Idea: Bus Snooping
  - All CPUs on the bus can see activity on the bus regardless of if they initiated it.



# Cache Coherency via the Bus

- Key Idea: Bus Snooping
  - All CPUs on the bus can see activity on the bus regardless of if they initiated it.



# Invalidation vs. Update

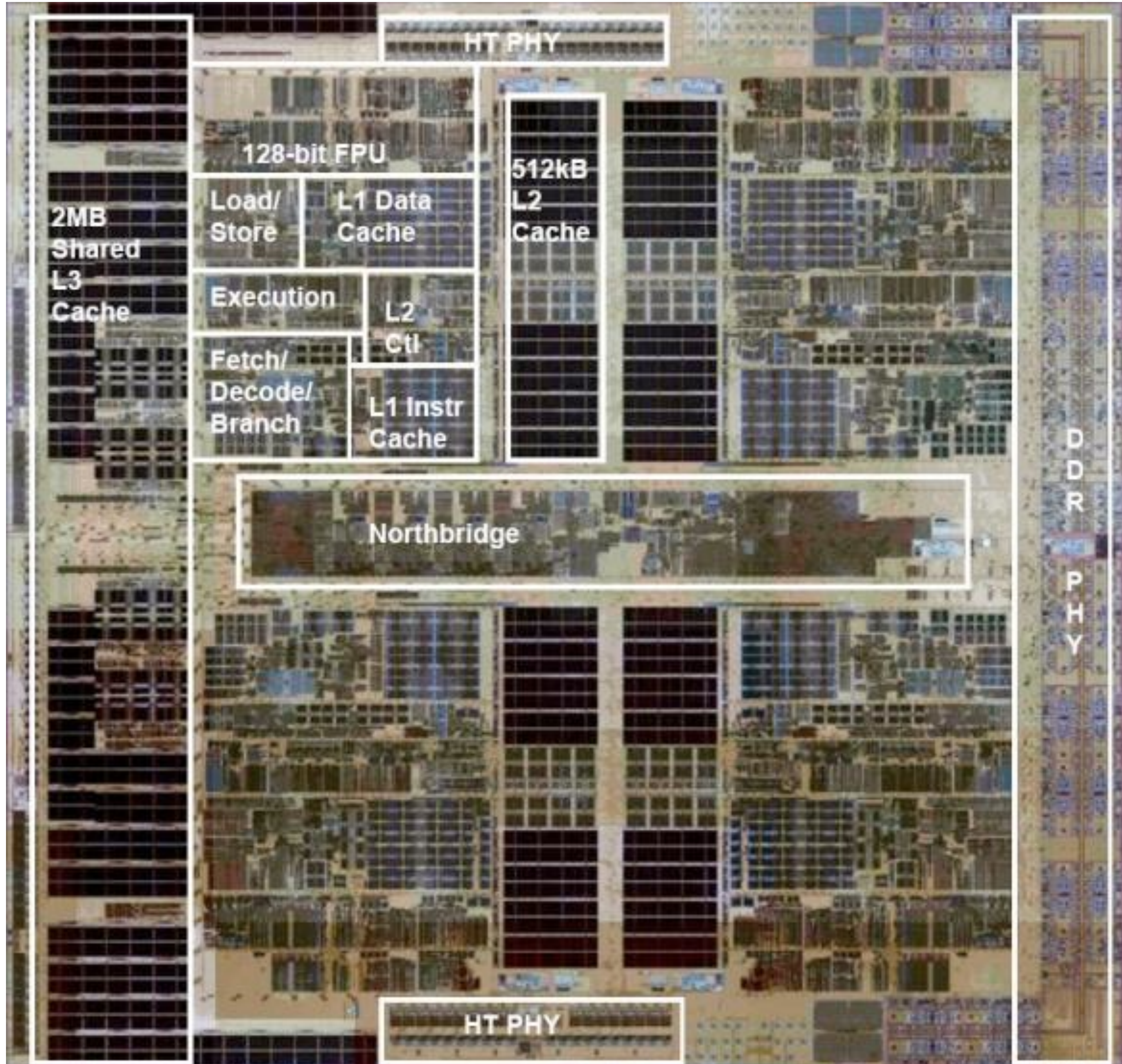
- A cache controller snoops and sees a write to a location that it has a now-outdated copy of.
  - What does it do?
- **Invalidation**
  - Mark cache block as *invalid*, so when CPU accesses it again, a miss will result and the updated data from main memory will be loaded. Requires one bit per block to implement.
- **Update**
  - See the write and update the caches with the value observed being written to main memory.

# CACHING: UNDER THE COVERS

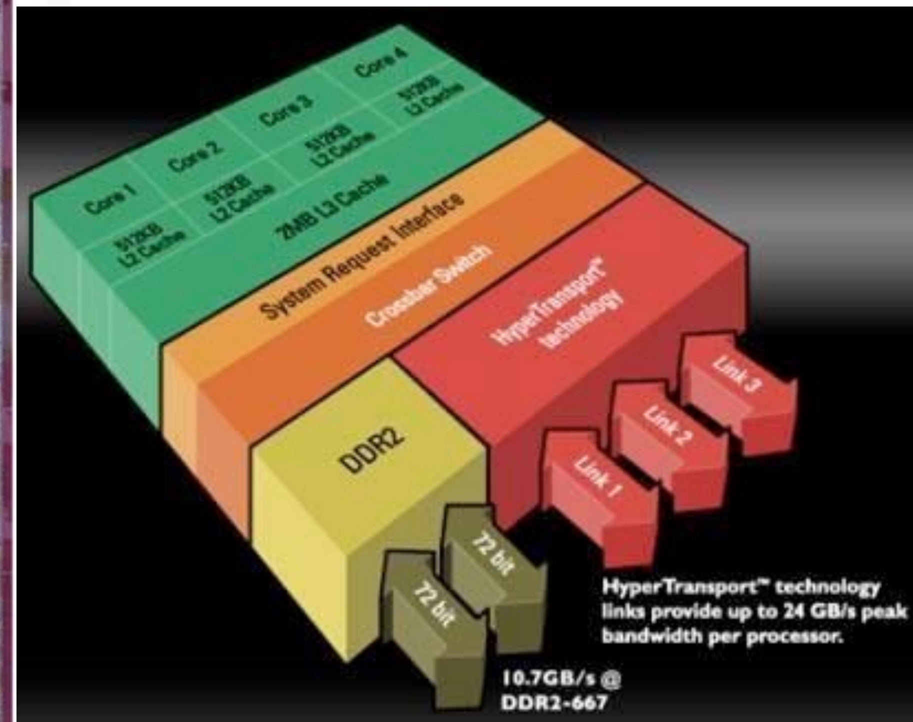
# Performance and Power

- High-end microprocessors have > 10 MB on-chip cache
  - Consumes large amount of area and power budget

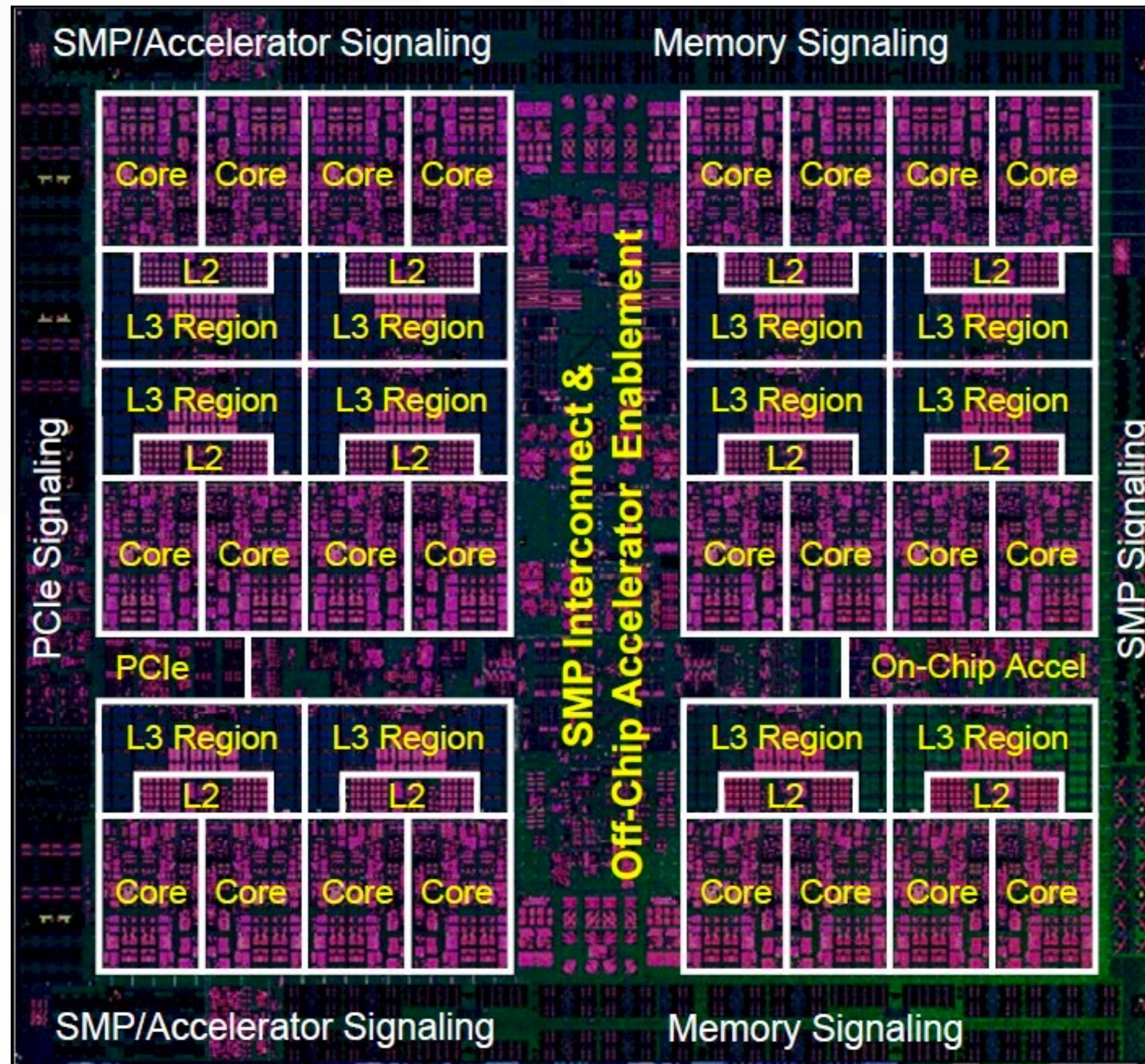
# AMD Opteron Rev. H Quad-Core




Where are the caches?

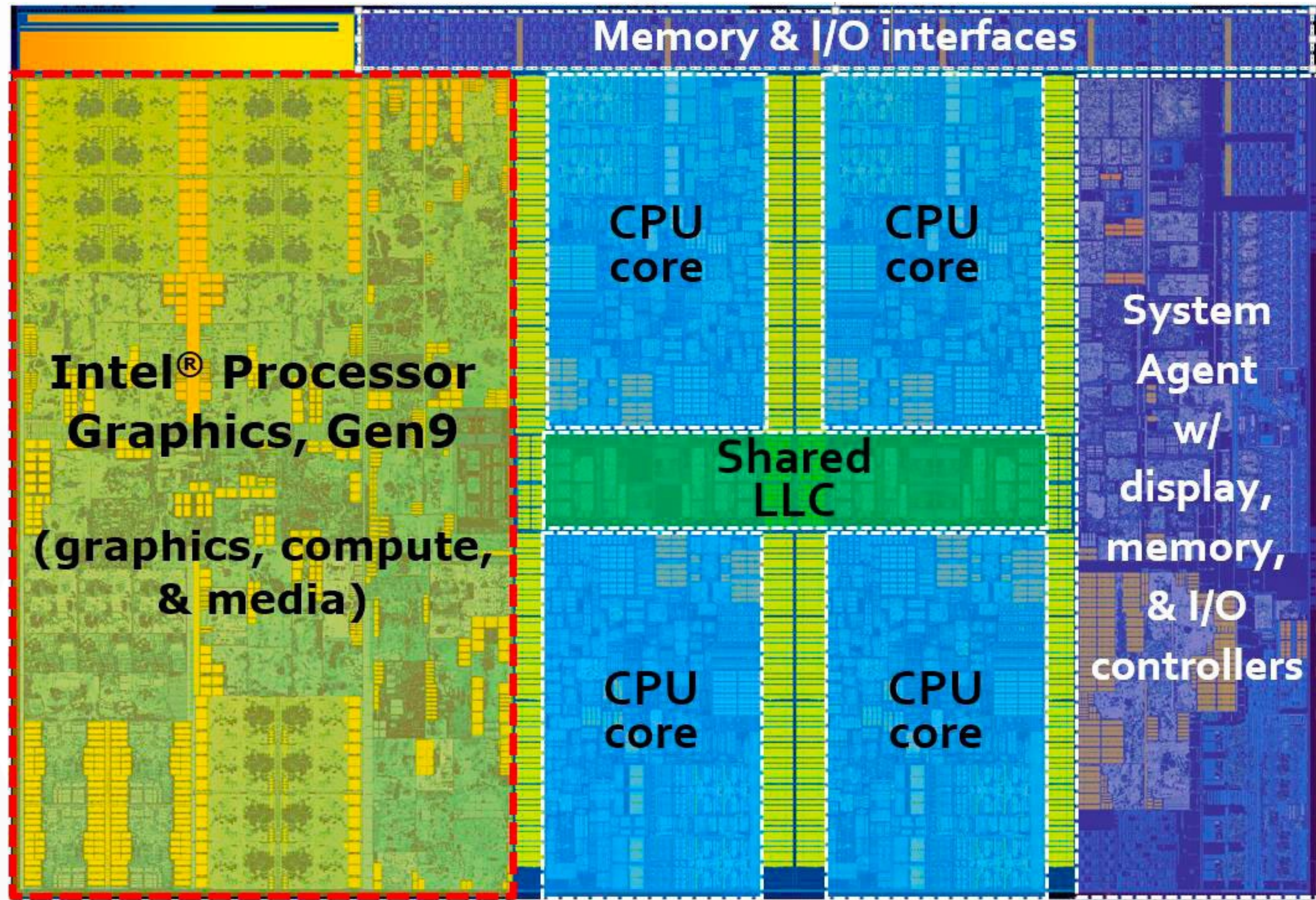


# IBM: Power9




Source: IBM

# Intel Skylake



# Memory Hierarchy: Algorithm

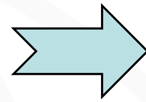
- When a word is not found in the cache, a miss occurs
  - Fetch word from lower level in hierarchy, requiring a higher latency reference
  - Lower level may be another cache or the main memory
  - Also fetch the other words contained within the block
    - Takes advantage of spatial locality
  - Place block into cache in any location within its set, determined by address
    - *block address MOD number of sets*

When to Write Blocks:
# Strategies to Write to the Cache

- *Write-Back*
  Only update lower levels of hierarchy when an updated block is replaced
  - On a write miss, the CPU reads the entire block from memory where the write address is, updates the value in cache, and marks the block as modified (aka dirty).
- *Write-Through*
  Immediately update lower levels of hierarchy
  - When the processor writes, even to a block in cache, a bus write is generated.
  - *Both strategies use write buffer to make writes asynchronous*
- Write-back is more efficient with respect to bandwidth usage on the bus, and hence, ubiquitously adopted.

Where to Write Blocks:

# Associativity



# How is the cache organized?

- *n-way set associative*, where $n$ = # of blocks per set

*Direct-mapped cache (1-way set associative)*
→ one block per set

*Fully associative cache (n-way set associative)*
→ n blocks in one set

| Tag | Index | Offset |
|---|---|---|
| xx | 000 | 00 |
| xx | 001 | 00 |
| xx | 010 | 00 |
| xx | 011 | 00 |
| xx | 100 | 00 |
| xx | 101 | 00 |
| xx | 110 | 00 |
| xx | 111 | 00 |

block address

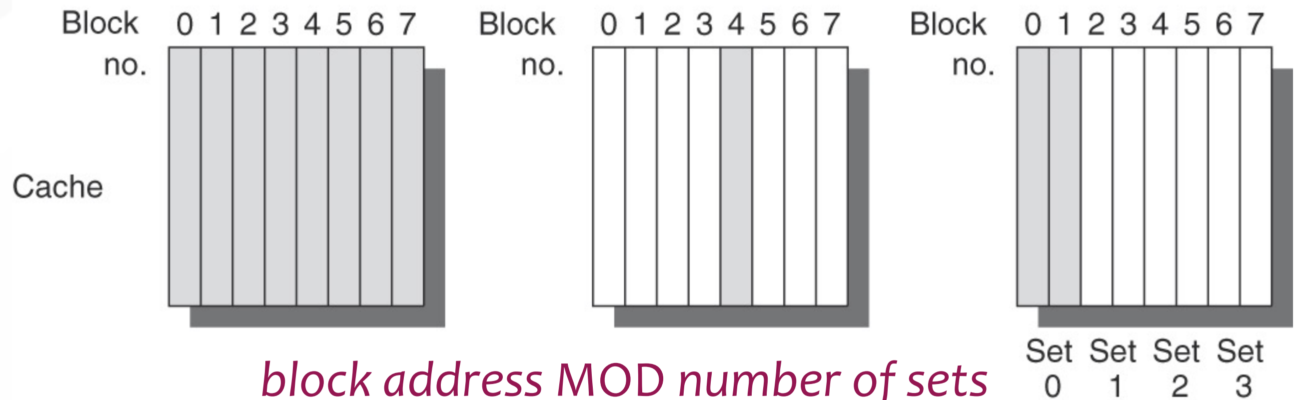
| Tag | Index | Offset |
|---|---|---|
| xxx | 00 | 00 |
| xxx | 01 | 00 |
| xxx | 10 | 00 |
| xxx | 11 | 00 |

block address

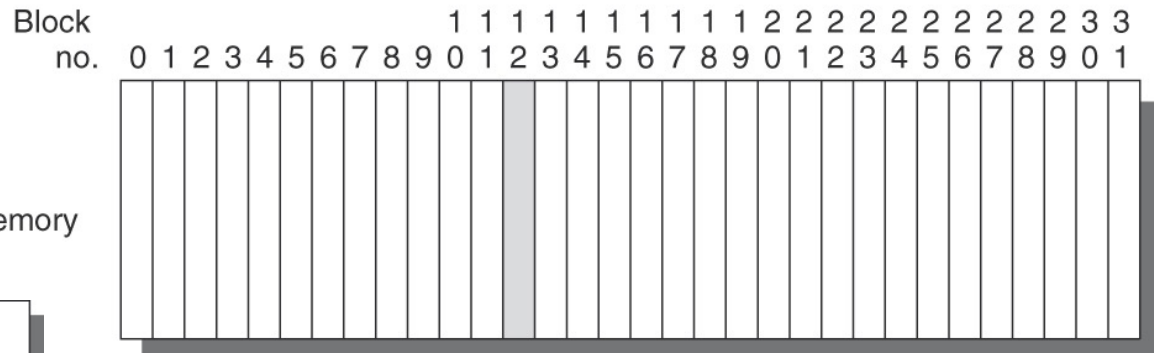
Fully associative: block 12 can go anywhere

Direct mapped: block 12 can go only into block 4 (12 MOD 8)

Set associative: block 12 can go anywhere in set 0 (12 MOD 4)

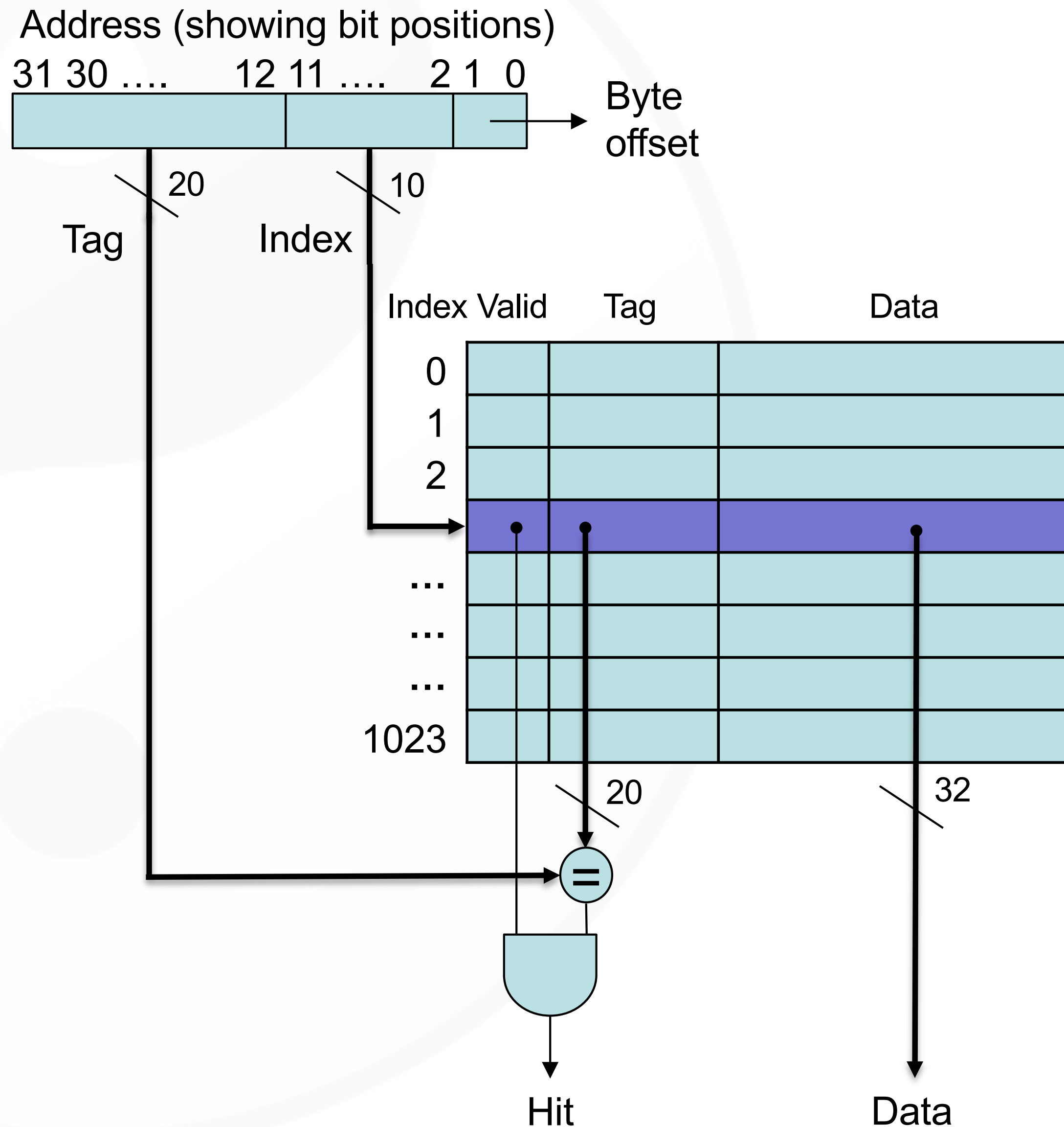



Block frame address

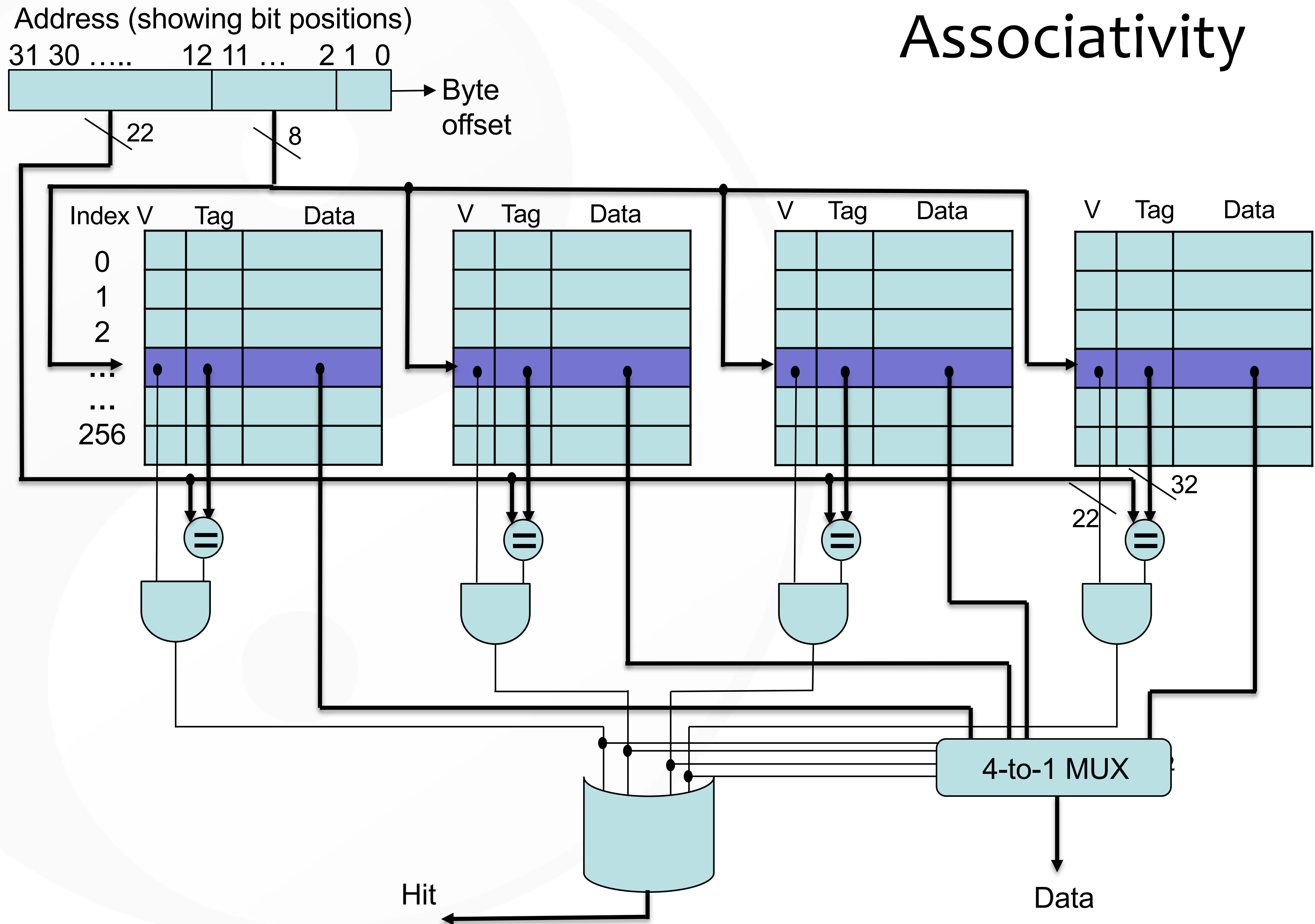



| Block address | | Block offset |
|---|---|---|
| Tag | Index | |

# Where to Write Blocks: Associativity



# Where to Write Blocks: Associativity



## How to Evaluate Cache Organization:
# Miss Rate

- Miss Rate
  - Fraction of cache access that result in a miss
- Causes of Misses
  - Compulsory
    - First reference to a block
  - Capacity
    - Blocks discarded and later retrieved
  - Conflict
    - Program makes repeated references to multiple addresses from different blocks that map to the same location in the cache

# Other Metrics?

$$\frac{\text{Misses}}{\text{Instruction}} = \frac{\text{Miss rate} \times \text{Memory accesses}}{\text{Instruction count}} = \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}}$$

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

- Note: Speculative and multithreaded processors may execute other instructions during a miss
  - Reduces performance impact of misses

# Basic Cache Optimizations

- Larger block size
  - Reduces compulsory misses
  - Increases capacity and conflict misses, increases miss penalty
- Larger total cache capacity to reduce miss rate
  - Increases hit time, increases power consumption
- Higher associativity
  - Reduces conflict misses
  - Increases hit time, increases power consumption
- Higher number of cache levels
  - Reduces overall memory access time, increases complexity
- Giving priority to read misses over writes
  - Reduces miss penalty, increases complexity
- Avoiding address translation in cache indexing
  - Reduces hit time