课程编号180086085404P2008H 2024-2025学年秋季学期

计算机体系结构

第5讲 缓存（Cache）

缓存体系结构、缓存操作

主讲教师： 刘珂

2024年10月8日

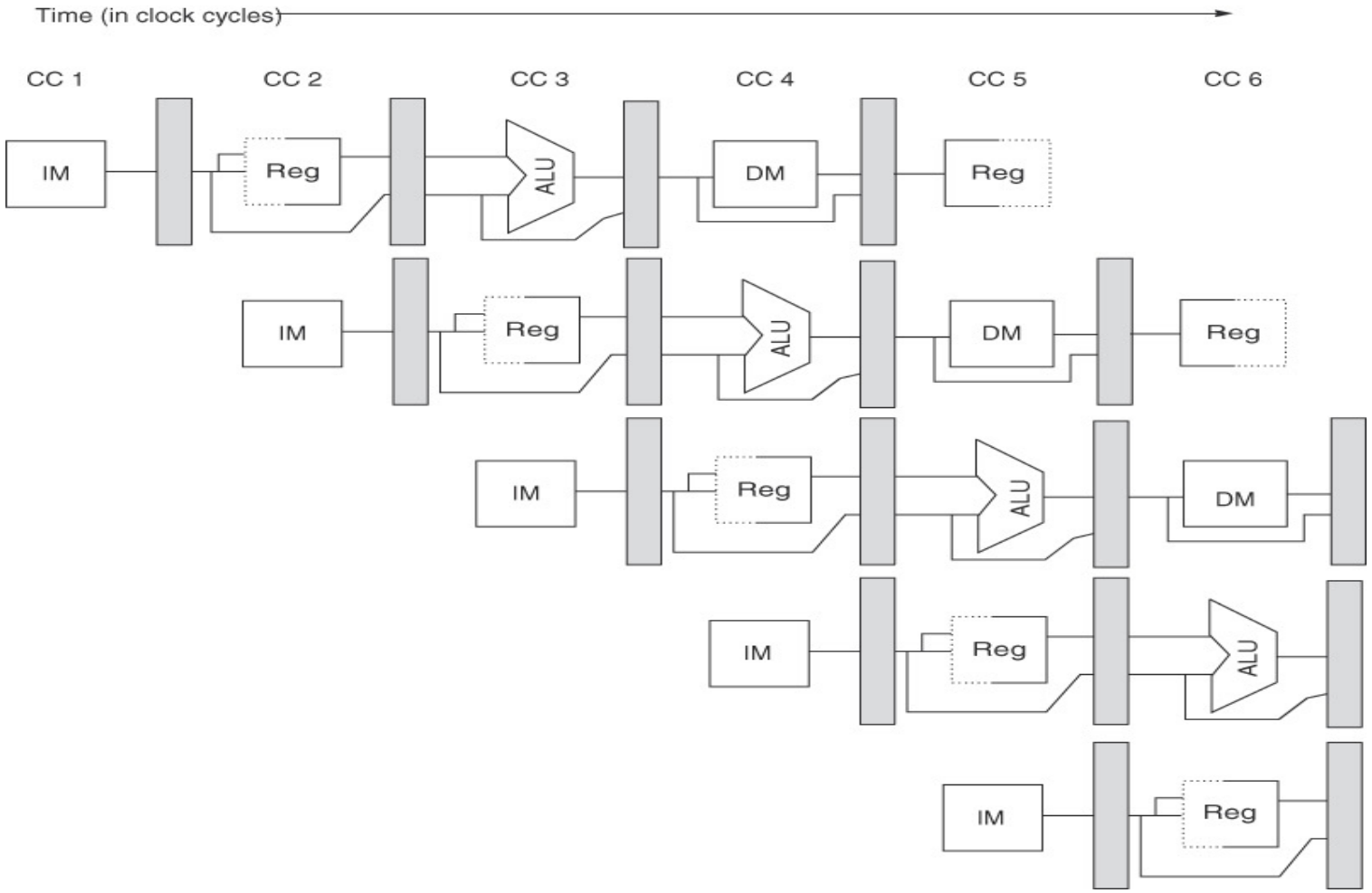
中国科学院大学 University of Chinese Academy of Sciences
中科院计算所 INSTITUTE OF COMPUTING TECHNOLOGY,CAS

Recall

- **Goal: Improving pipeline performance**
- **Issue: Out-of-order execution may result in imprecise exceptions**
  - Solution: Reorder Buffer (ROB) – Ensuring Precise Exceptions
- **Issue: The maximum throughput of a pipeline will not exceed one instruction per cycle.**
  - Solution: Superscalar – Enabling the simultaneous execution of multiple instructions, reducing CPI.
- **Issue: Stall the pipeline until we know the next fetch address**
  - Solution: Branch Prediction – Predict the next fetch address (to be used in the next cycle) , so no wasted cycle(s) on correct prediction
    - Whether the fetched instruction is a branch
    - Branch direction
    - Branch target address (if taken)

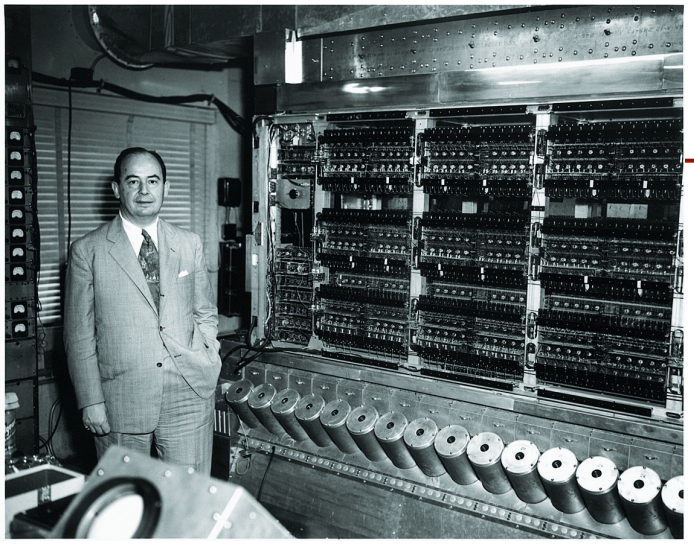
Recall

- 5-stage pipeline



A Computing System

- Three key components
- Computation
- Communication
- Storage/memory



Burks, Goldstein, von Neumann, "Preliminary discussion of the logical design of an electronic computing instrument," 1946.

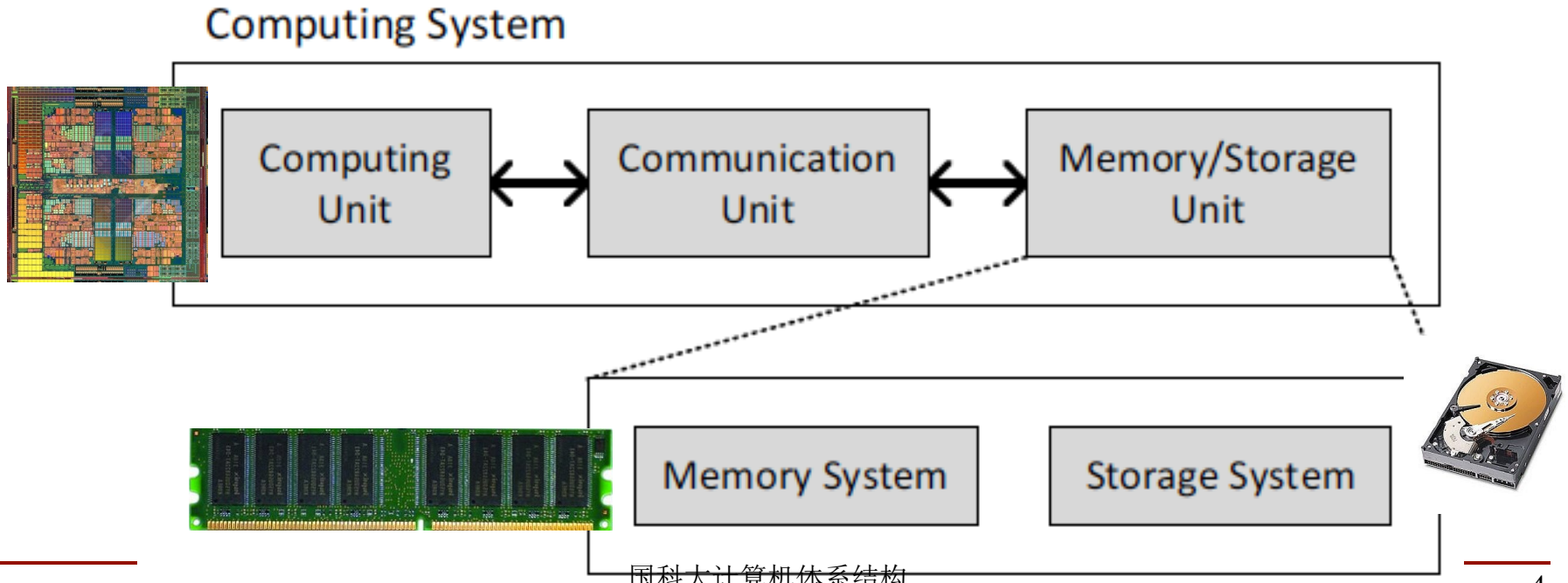
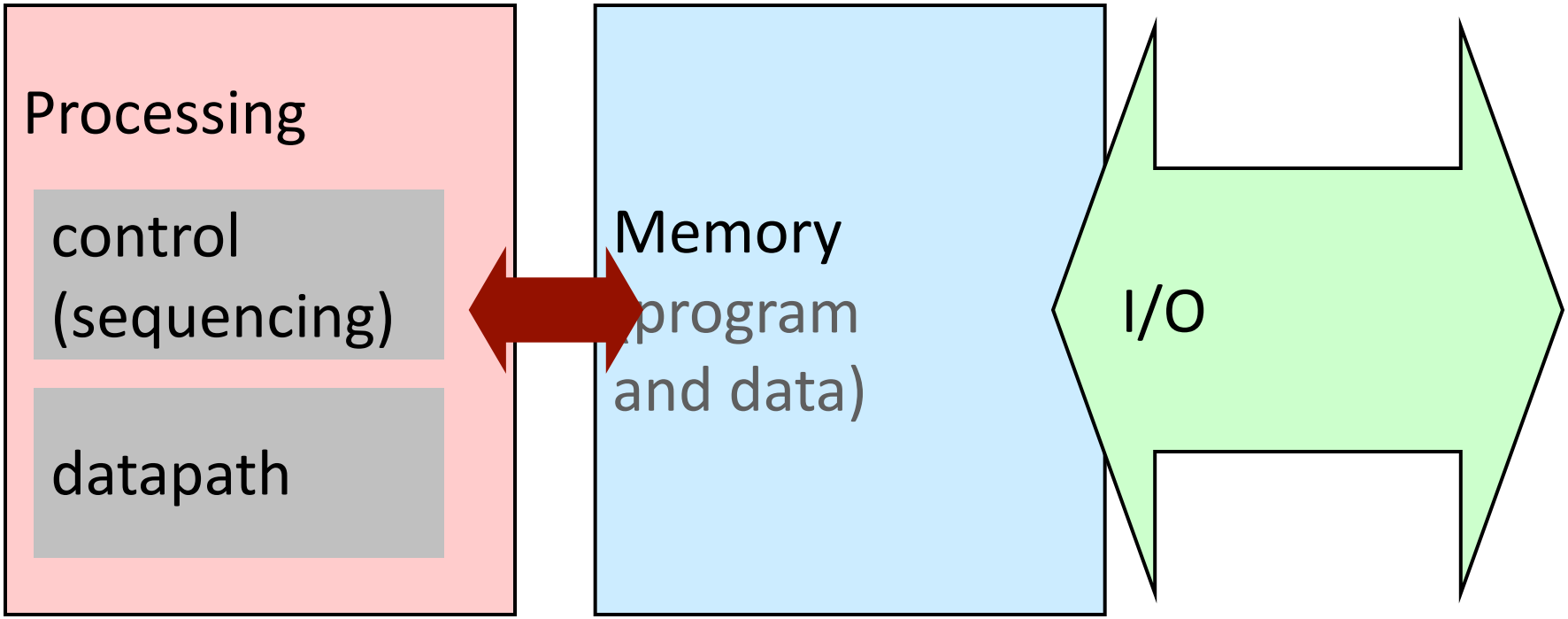


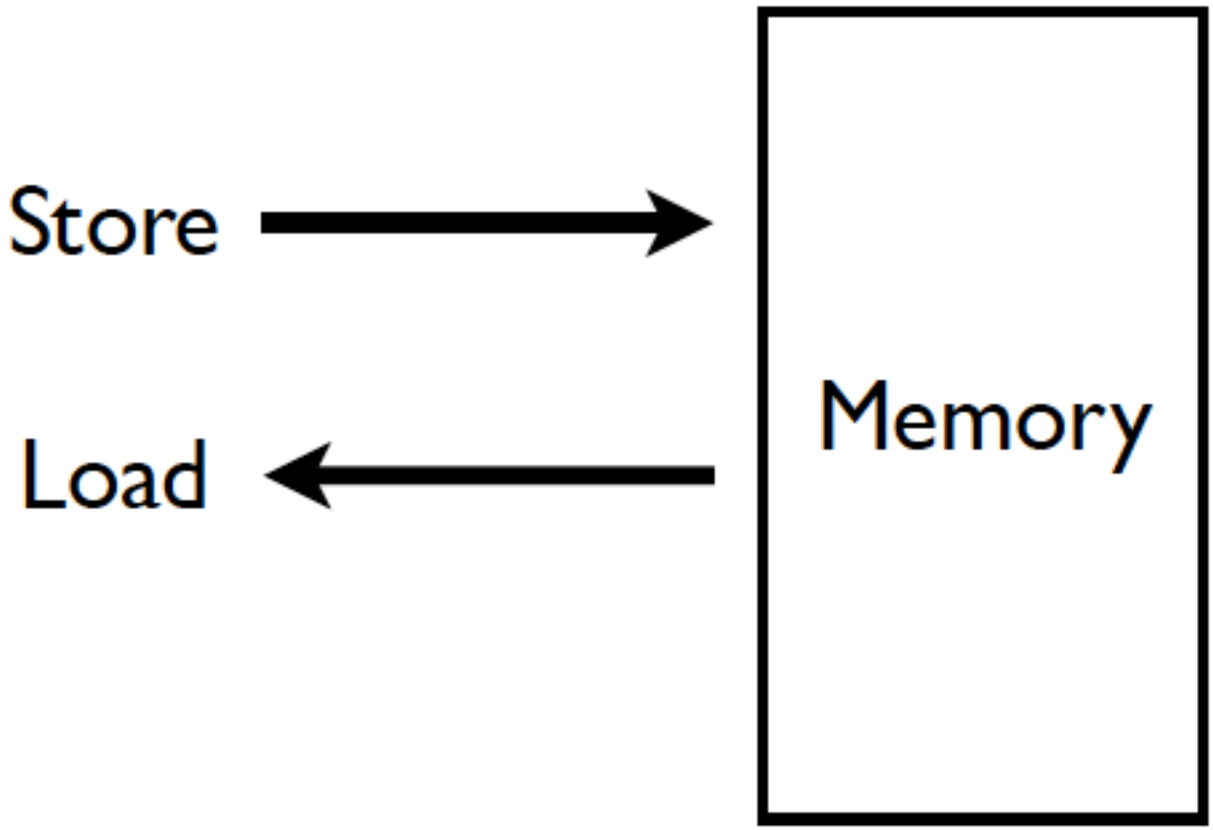
Image source: https://lbsitbytes2010.wordpress.com/2013/03/29/john-von-neumann-roll-no-15/

What is A Computer?

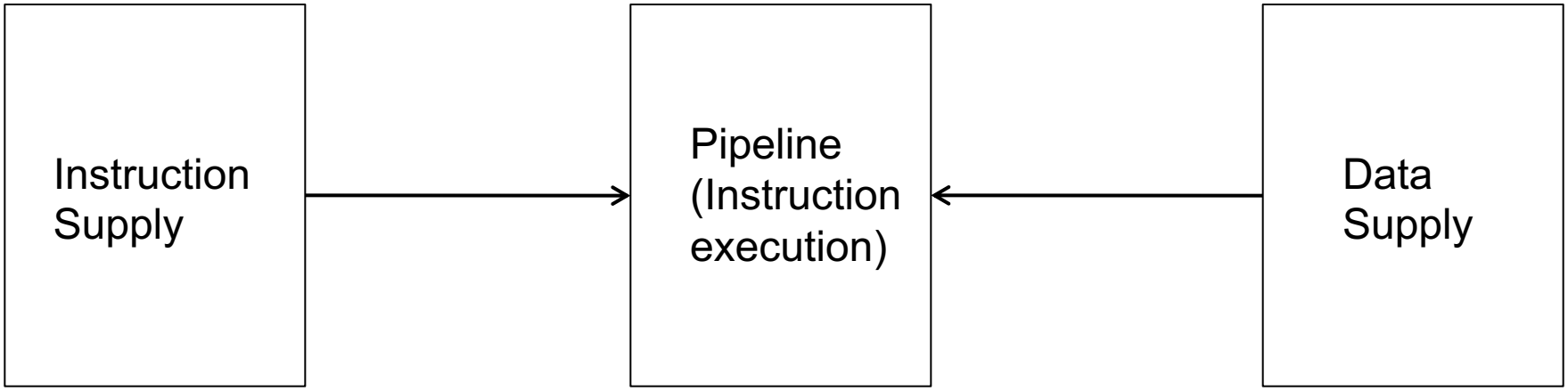
- We will cover all three components



Memory (Programmer's View)



Idealism



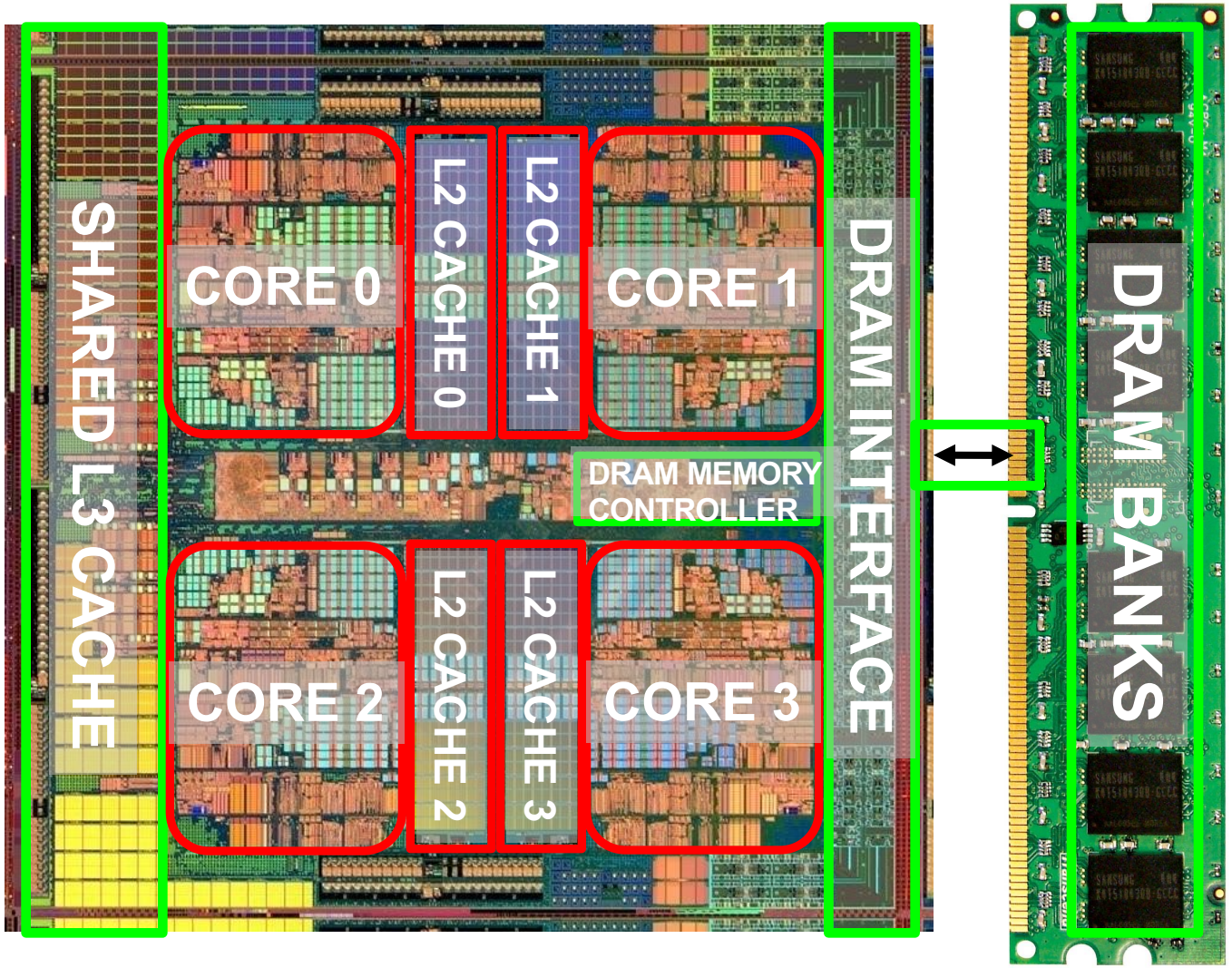
- Zero latency access
- Infinite capacity
- Zero cost
- Perfect control flow

- No pipeline stalls
- Perfect data flow (reg/memory dependencies)
- Zero-cycle interconnect (operand communication)
- Zero latency compute
- Zero energy consumption

- Zero latency access
- Infinite capacity
- Infinite bandwidth
- Zero cost

The Memory Hierarchy

Memory in a Modern System



Ideal Memory

- Zero access time (latency)
- Infinite capacity
- Zero cost
- Infinite bandwidth (to support multiple accesses in parallel)
  - Banking
  - Multi-porting

The Problem

- Ideal memory's requirements oppose each other
- Bigger is slower
  - Bigger → Takes longer to determine the location
- Faster is more expensive
  - Memory technology: SRAM vs. DRAM vs. Disk vs. Tape
- Higher bandwidth is more expensive
  - Need more banks, more ports, higher frequency, or faster technology

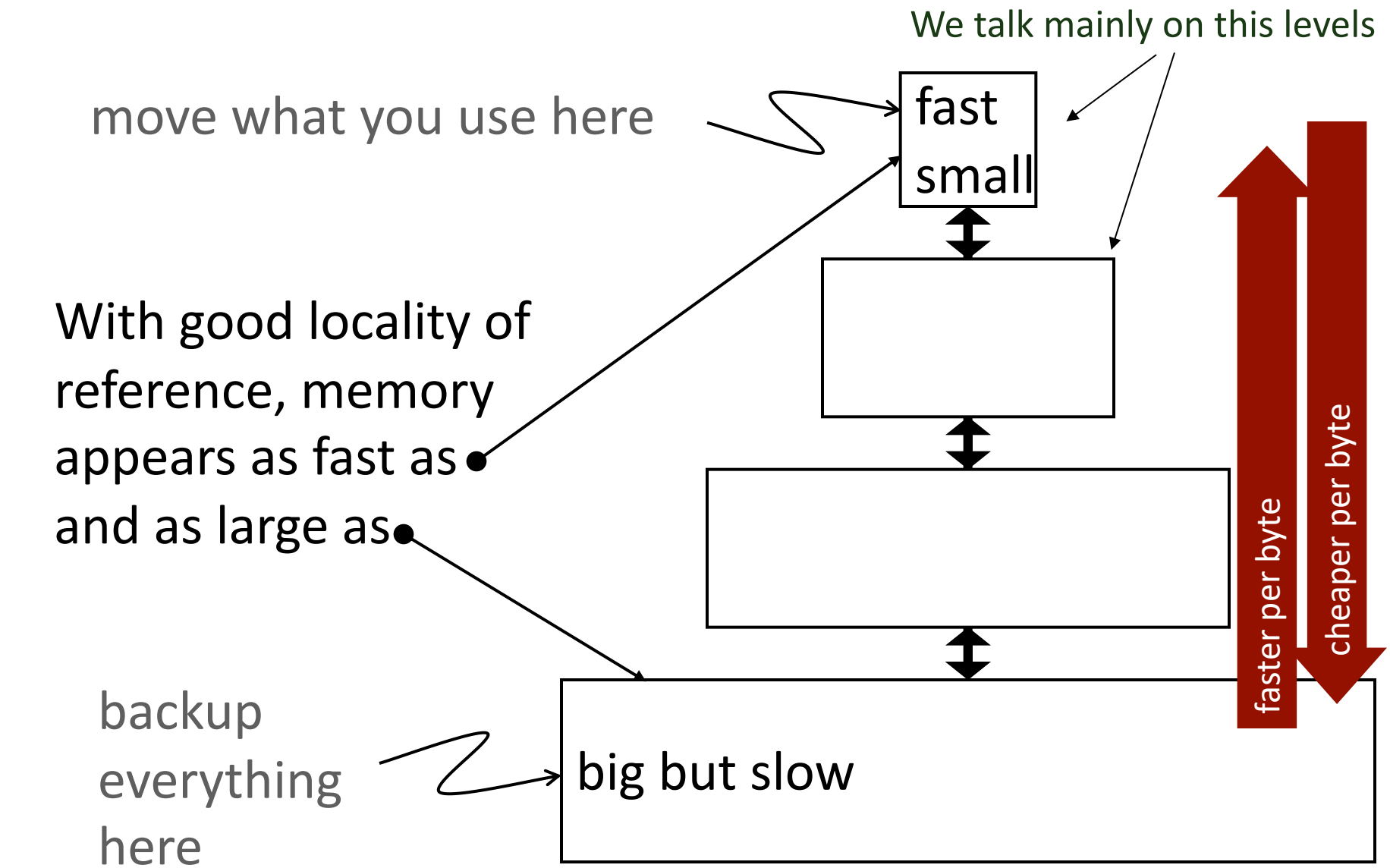
The Problem

- Bigger is slower
  - SRAM, 512 Bytes, sub-nanosec
  - SRAM, KByte~MByte, ~nanosec
  - DRAM, Gigabyte, ~50 nanosec
  - Hard Disk, Terabyte, ~10 millisec
- Faster is more expensive (dollars and chip area)
  - SRAM, < 10$ per Megabyte
  - DRAM, < 1$ per Megabyte
  - Hard Disk < 1$ per Gigabyte
  - These sample values (circa ~2011) scale with time

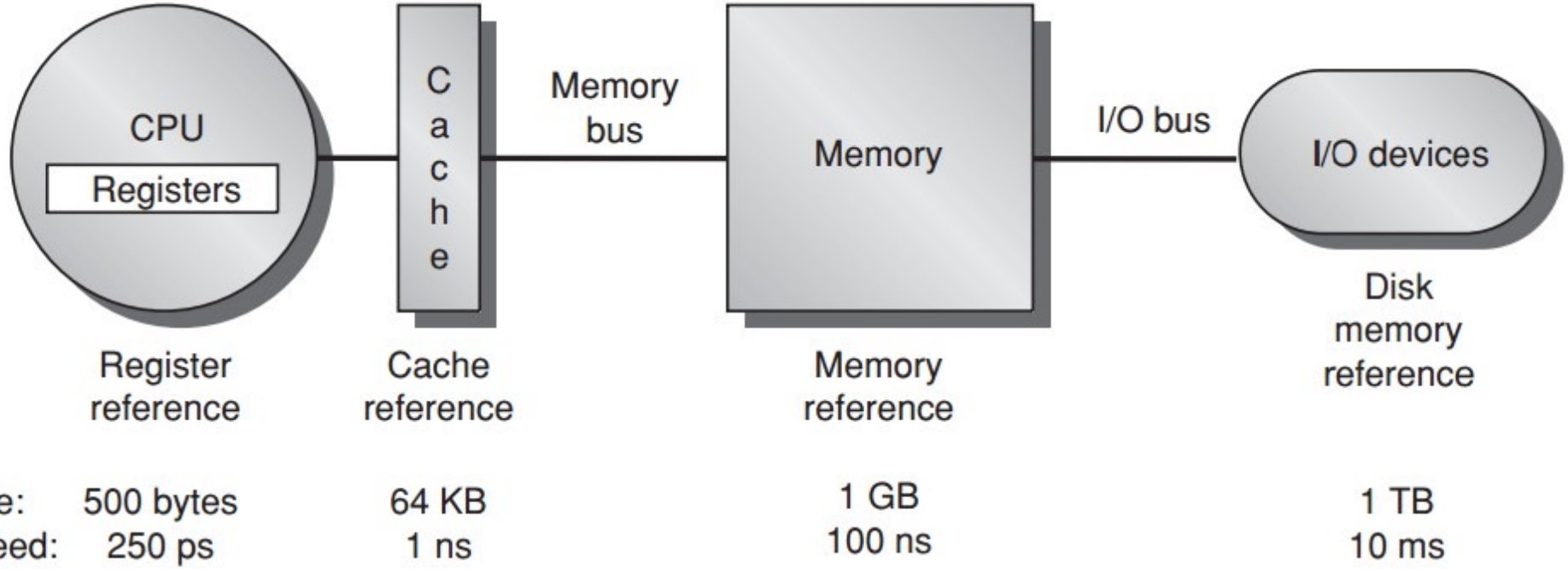
Why Memory Hierarchy?

- We want both fast and large
- But we cannot achieve both with a single level of memory
- Idea: Have multiple levels of storage (progressively bigger and slower as the levels are farther from the processor)（内存的层次结构）and ensure most of the data the processor needs is kept in the fast(er) level(s)

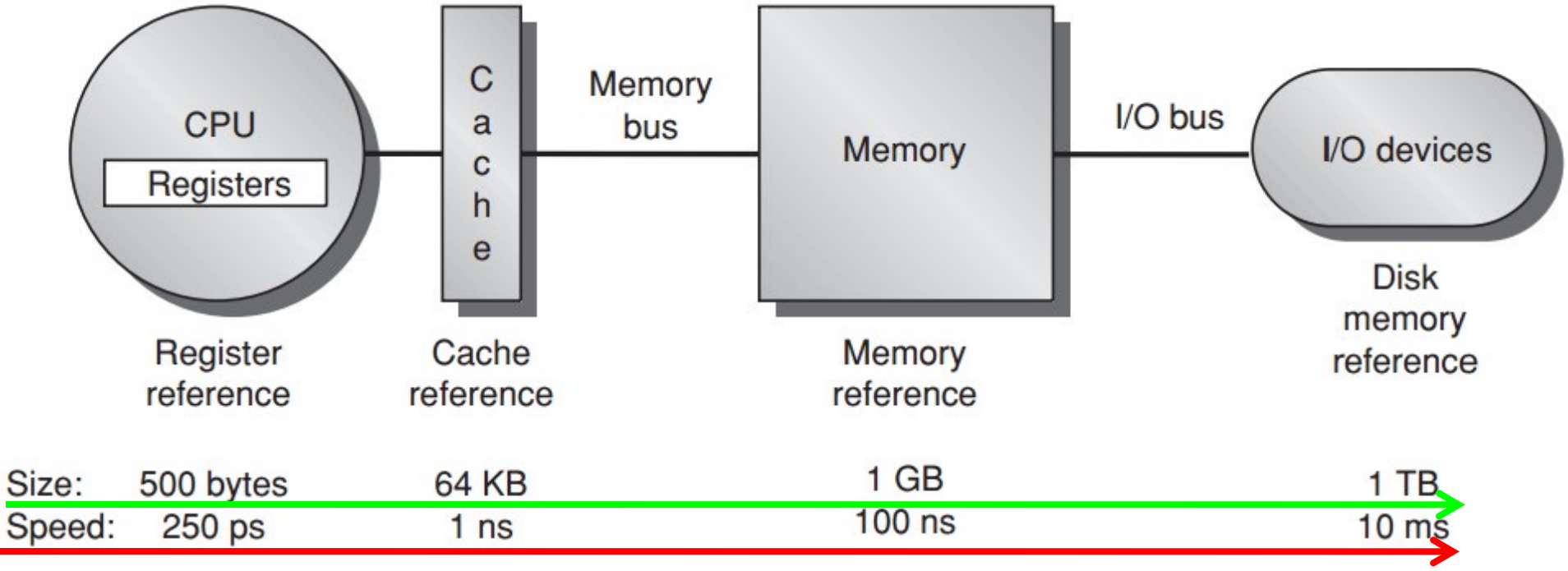
The Memory Hierarchy



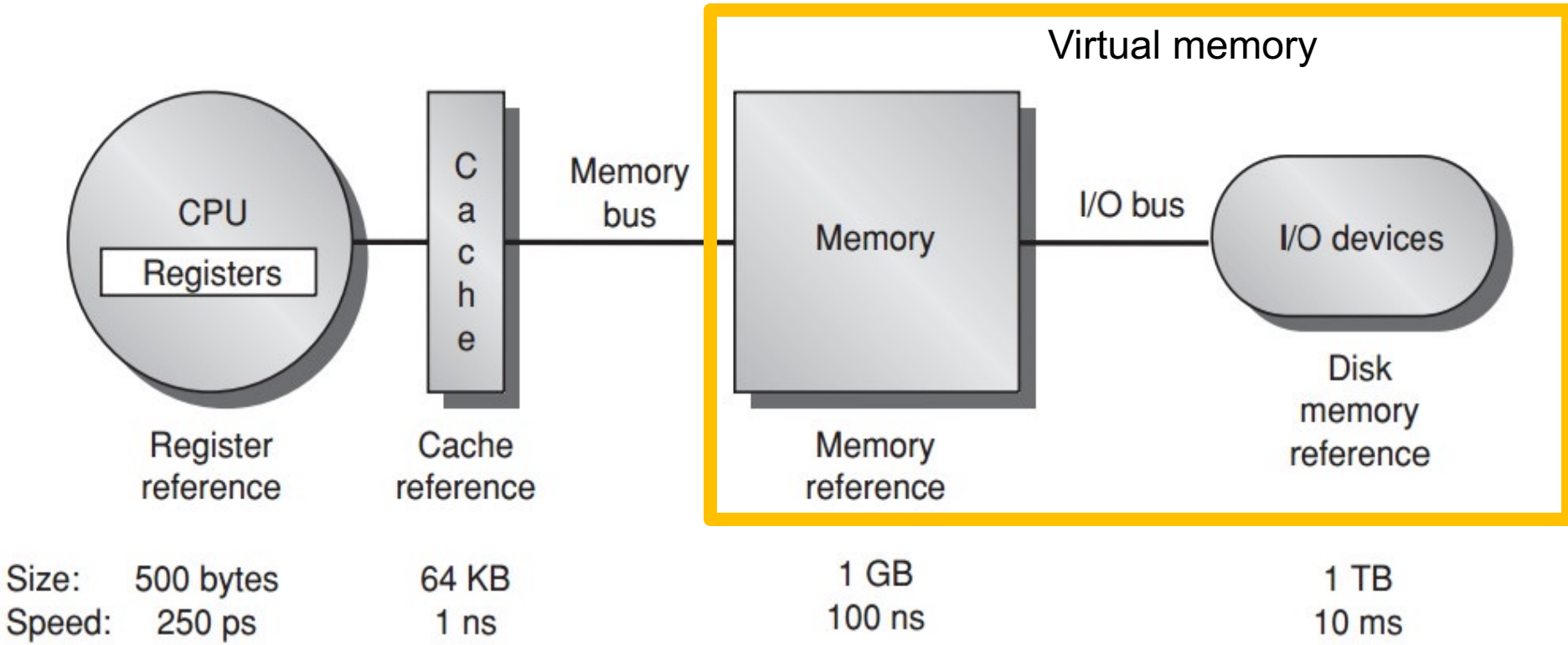
Memory Hierarchy



Memory Hierarchy

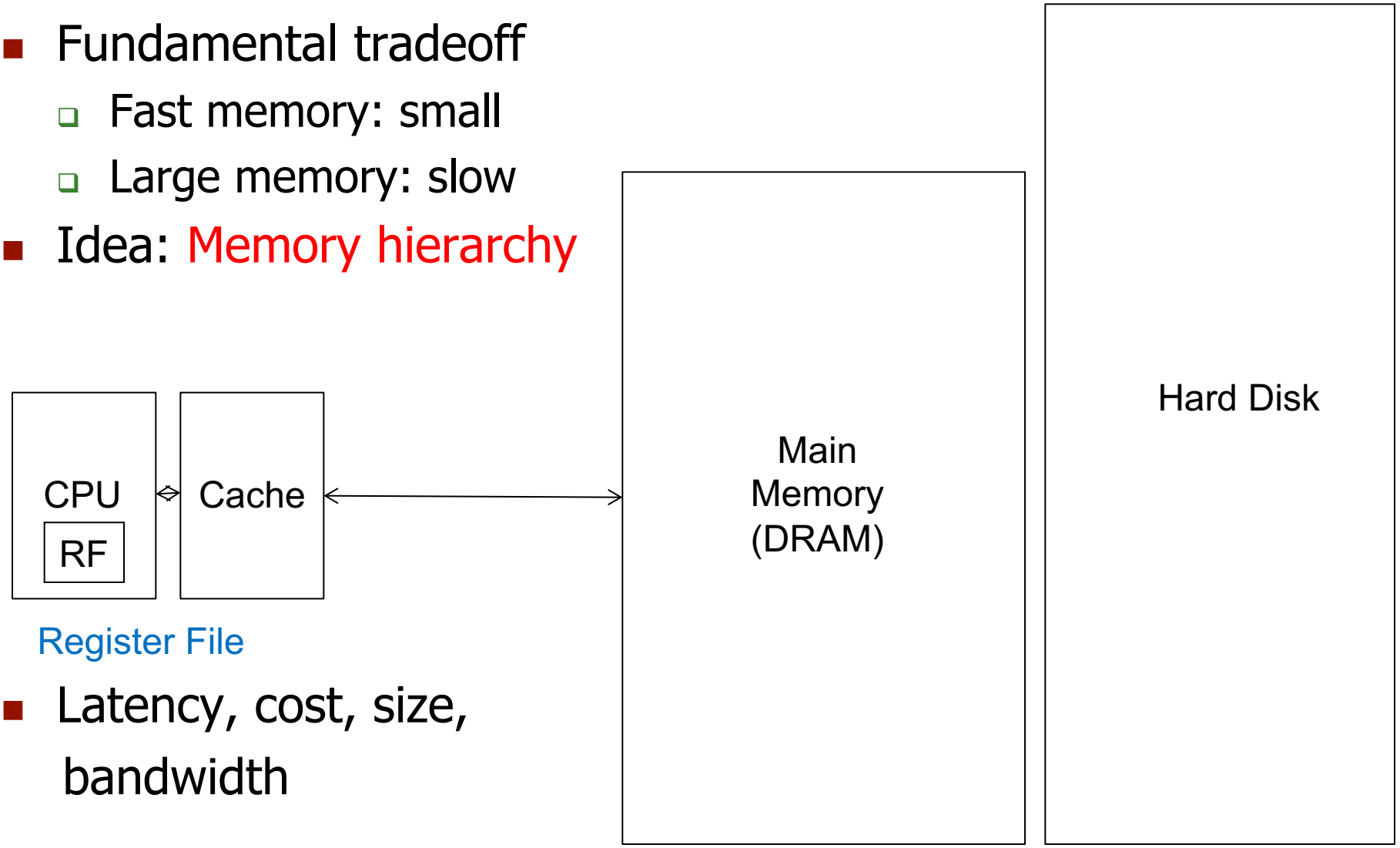


Memory Hierarchy



Memory Hierarchy

- Fundamental tradeoff
  - Fast memory: small
  - Large memory: slow
- Idea: Memory hierarchy



Register File

- Latency, cost, size, bandwidth

Locality 局部性

- One's recent past is a very good predictor of his/her near future.

- Temporal Locality（时间局部性）: If you just did something, it is very likely that you will do the same thing again soon
  - since you are here today, there is a good chance you will be here again and again regularly

- Spatial Locality（空间局部性）: If you did something, it is very likely you will do something similar/related (in space)
  - every time I find you in this room, you are probably sitting close to the same people

Memory Locality

- A "typical" program has a lot of locality in memory references
  - typical programs are composed of "loops"

- Temporal: A program tends to reference the same memory location many times and all within a small window of time

- Spatial: A program tends to reference a cluster of memory locations at a time
  - most notable examples:
    - 1. instruction memory references
    - 2. array/data structure references

Exploit Temporal Locality

- Idea: Store recently accessed data in automatically managed fast memory (called cache)
- Anticipation: the data will be accessed again soon

- Temporal locality principle
  - Recently accessed data will be again accessed in the near future
  - This is what Maurice Wilkes had in mind:
    - Wilkes, "Slave Memories and Dynamic Storage Allocation," IEEE Trans. On Electronic Computers, 1965.
    - "The use is discussed of a fast core memory of, say 32000 words as a slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory."

Exploit Spatial Locality

- Idea: Store addresses adjacent to the recently accessed one in automatically managed fast memory
  - Logically divide memory into equal size blocks
  - Fetch to cache the accessed block in its entirety
- Anticipation: nearby data will be accessed soon

- Spatial locality principle
  - Nearby data in memory will be accessed in the near future
    - E.g., sequential instruction access, array traversal
  - This is what IBM 360/85 implemented
    - 16 Kbyte cache with 64 byte blocks
    - Liptay, "Structural aspects of the System/360 Model 85 II: the cache," IBM Systems Journal, 1968.

The Bookshelf Analogy

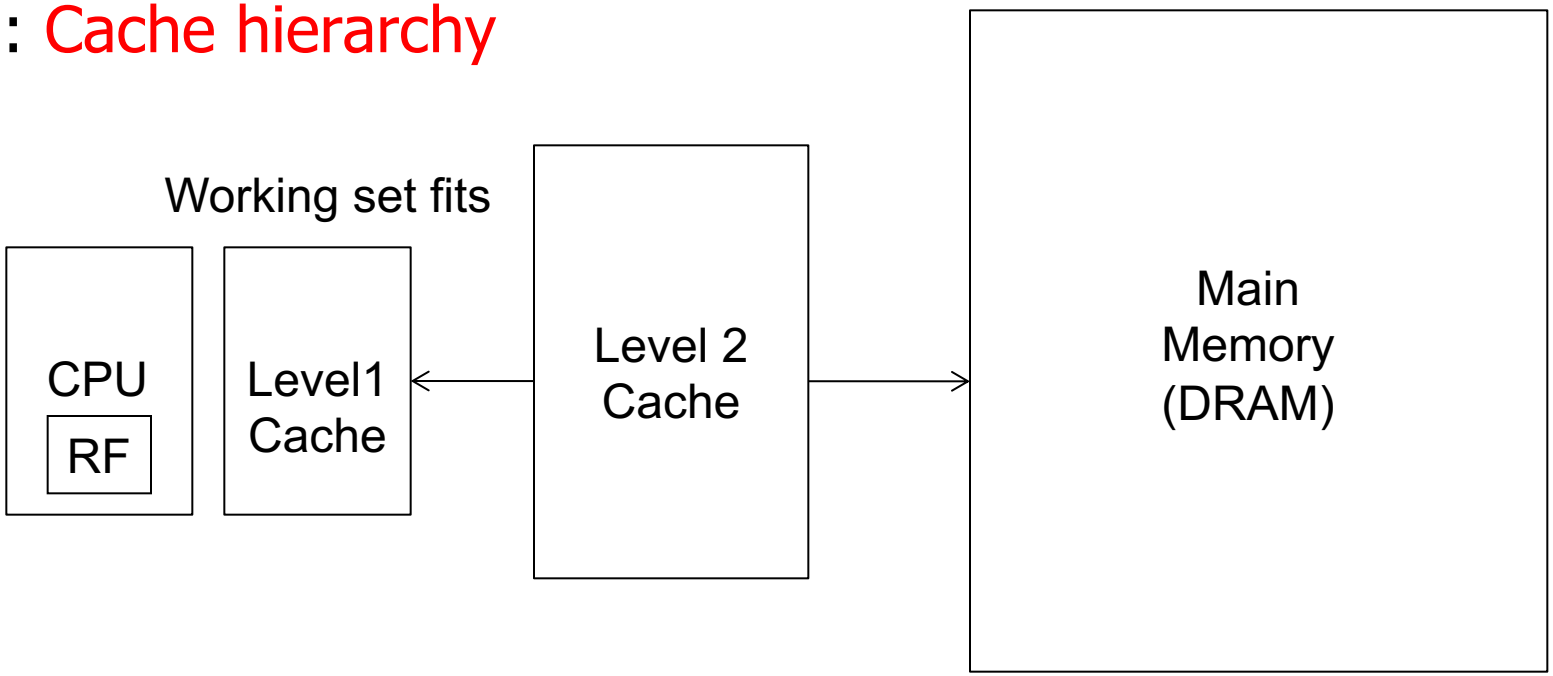
- Book in your hand
- Desk
- Bookshelf
- Boxes at home
- Boxes in storage

- Recently-used books tend to stay on desk
  - Comp Arch books, books for classes you are currently taking
  - Until the desk gets full
- Adjacent books in the shelf needed around the same time
  - If I have organized/categorized my books well in the shelf
  - “Book cache management”

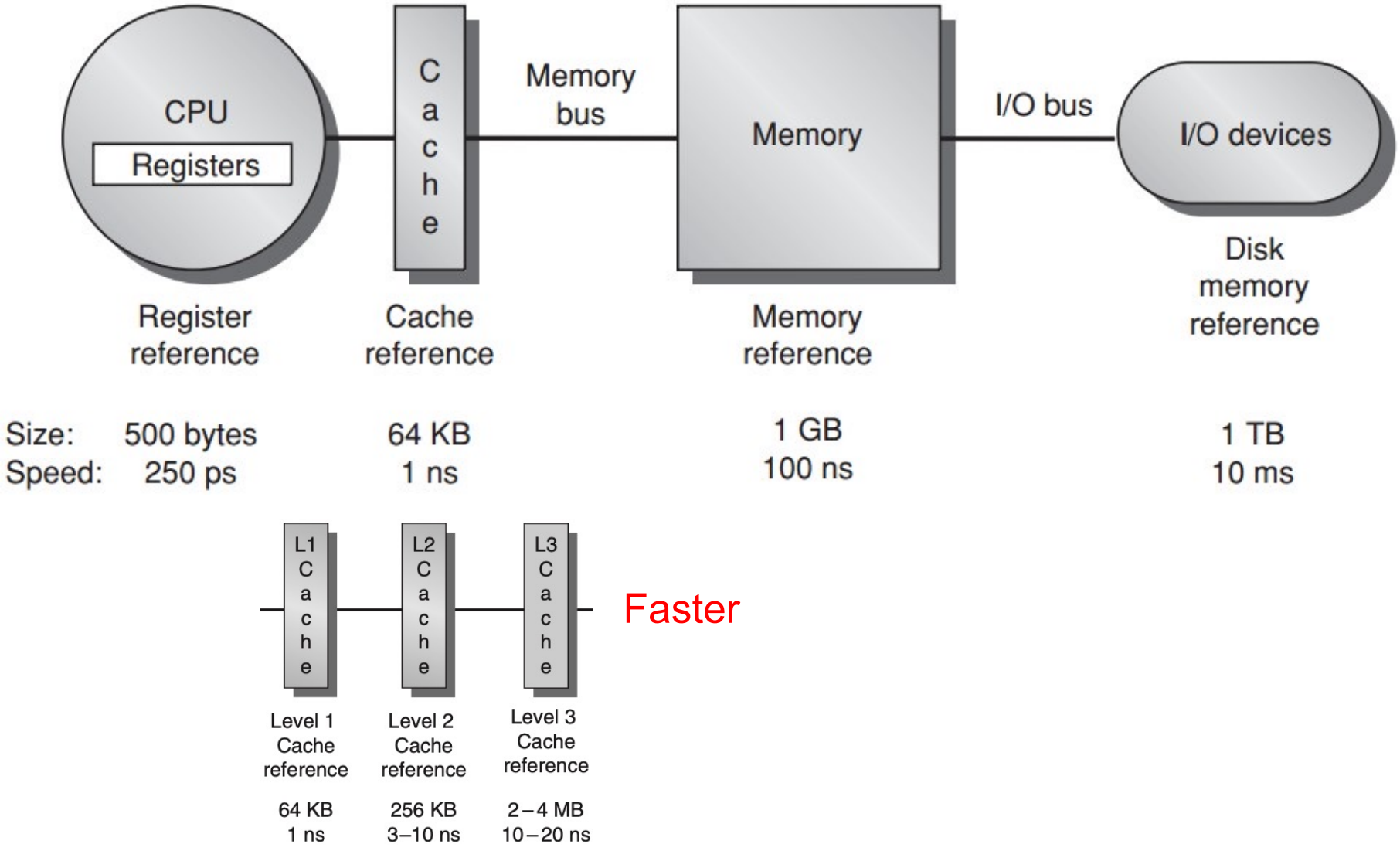
The Cache Hierarchy

Caching in a Pipelined Design

- The cache needs to be tightly integrated into the pipeline
  - Ideally, access in 1-cycle so that load-dependent operations do not stall
- High frequency pipeline → Cannot make the cache large
  - But, we want a large cache AND a pipelined design
- Idea: Cache hierarchy

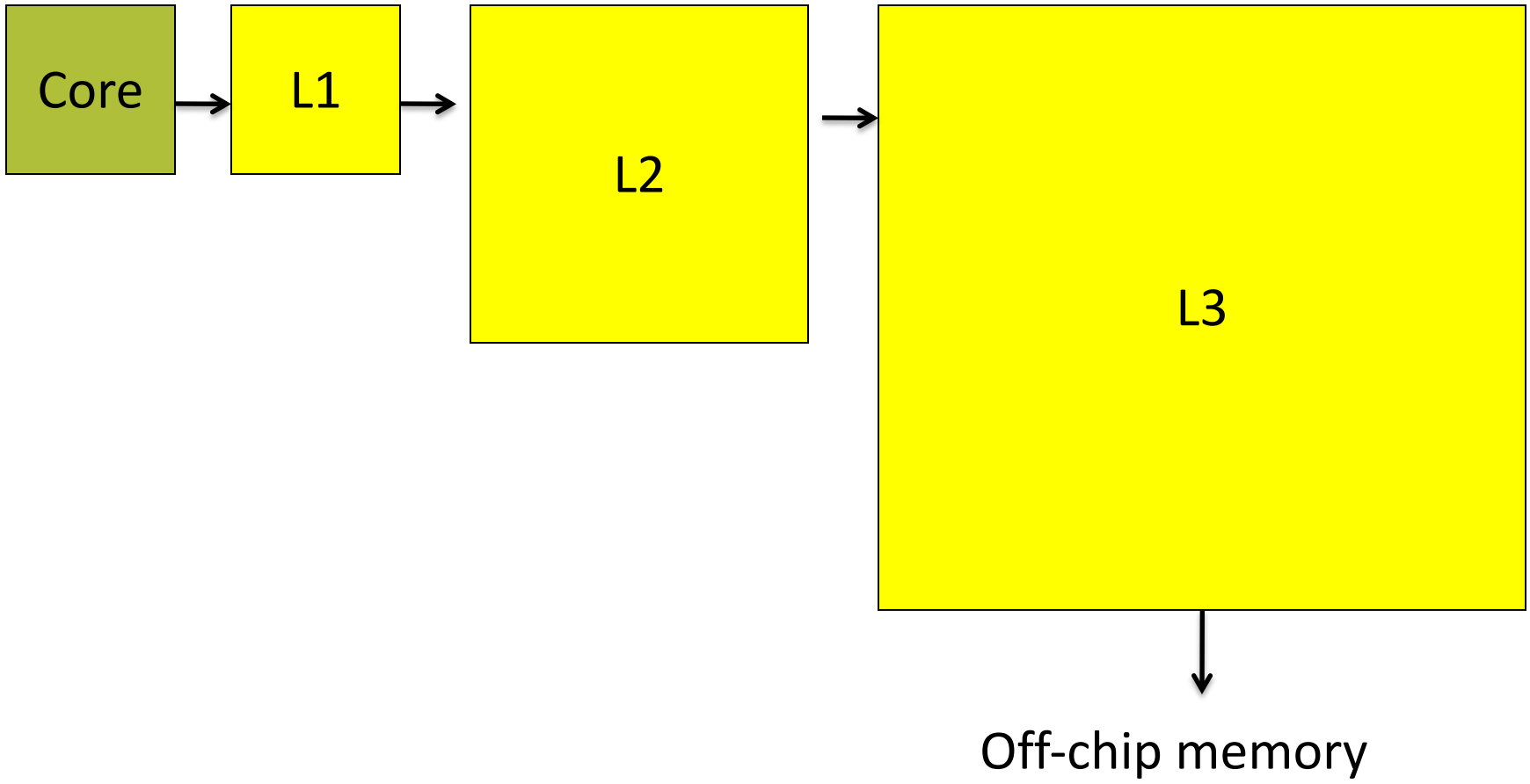


Cache Hierarchy



The Cache Hierarchy

- Example



A Note on Manual vs. Automatic Management

- Manual: Programmer manages data movement across levels

  -- too painful for programmers on substantial programs

  - Still done in some embedded processors (on-chip scratch pad SRAM) and GPUs (called "shared memory")

- Automatic: Hardware manages data movement across levels, transparently to the programmer

  ++ programmer's life is easier

  - The average programmer doesn't need to know about it
    - You don't need to know how big the cache is and how it works to write a "correct" program! (What if you want a "fast" program?)

Automatic Management in Memory Hierarchy

- Wilkes, "Slave Memories and Dynamic Storage Allocation," IEEE Trans. On Electronic Computers, 1965.

Slave Memories and Dynamic Storage Allocation

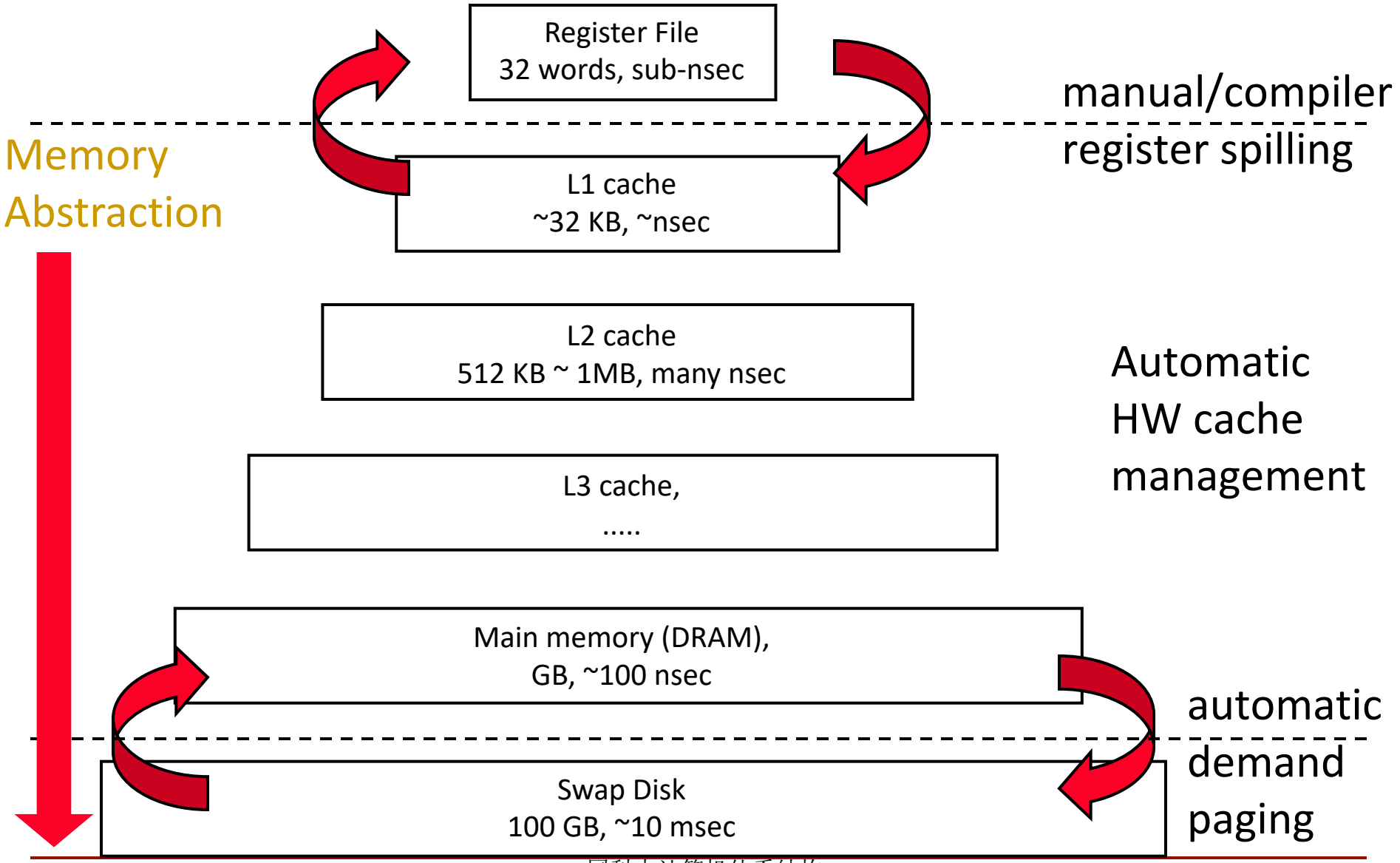
M. V. WILKES

SUMMARY

The use is discussed of a fast core memory of, say, 32 000 words as a slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory.

- "By a slave memory I mean one which automatically accumulates to itself words that come from a slower main memory, and keeps them available for subsequent use without it being necessary for the penalty of main memory access to be incurred again."

A Modern Memory Hierarchy



Hierarchical Latency Analysis

- Cache Hit/Miss
  - When the processor **can/cannot** find a requested data item in the cache
- For a given memory hierarchy level i it has a technology-intrinsic access time of t_i (固有访问时间), the perceived access time T_i (感知访问时间) is longer than t_i
- Except for the outer-most hierarchy, when looking for a given address there is
  - a chance (hit-rate h_i)（命中率）you "hit" and access time is t_i
  - a chance (miss-rate m_i)（缺失率）you "miss" and access time $t_i + T_{i+1}$
  - $h_i + m_i = 1$
- Thus $T_i = h_i \cdot t_i + m_i \cdot (t_i + T_{i+1})$

  Hierarchical latency to access the remaining memory levels

  $T_i = t_i + m_i \cdot T_{i+1}$ ← Miss penalty

  Hierarchical equation（递归）
  - h_i and m_i are defined to be the hit-rate and miss-rate of just the references that missed at L_{i-1}

Hierarchy Design Considerations

- Recursive latency equation

$$T_i = t_i + m_i \cdot T_{i+1}$$

- The goal: achieve desired T_1 within allowed cost
- $T_i \approx t_i$ is desirable
- Keep m_i low
  - increasing capacity C_i lowers m_i, but beware of increasing t_i
  - lower m_i by smarter cache management
    - replacement – anticipate what you don't need, pseudo-LRU, NRU, DRRIP
    - prefetching – anticipate what you will need, stride stream prefetching
    - victim caches, cache compression
- Keep T_{i+1} low
  - faster lower hierarchies, but beware of increasing cost
  - introduce intermediate hierarchies as a compromise

Intel Pentium 4 Example

- 90nm P4, 3.6 GHz
- L1 D-cache
  - C_1 = 16K
  - t_1 = 4 cyc int / 9 cycle fp
    - Int: integer, fp: floating point
- L2 D-cache
  - C_2 =1024 KB
  - t_2 = 18 cyc int / 18 cyc fp
- Main memory
  - t_3 = ~ 50ns or 180 cyc
- Notice
  - best case latency is not 1
  - worst case access latencies are into 500+ cycles

if m_1=0.1, m_2=0.1
T_1=7.6, T_2=36

if m_1=0.01, m_2=0.01
T_1=4.2, T_2=19.8

if m_1=0.05, m_2=0.01
T_1=5.00, T_2=19.8

if m_1=0.01, m_2=0.50
T_1=5.08, T_2=108

The management of small cache does matter

Cache Operations and Basics

Cache in Engineering

- Generically, any structure that "memorizes" frequently used results to avoid repeating the long-latency operations required to reproduce the results from scratch, e.g. a web cache

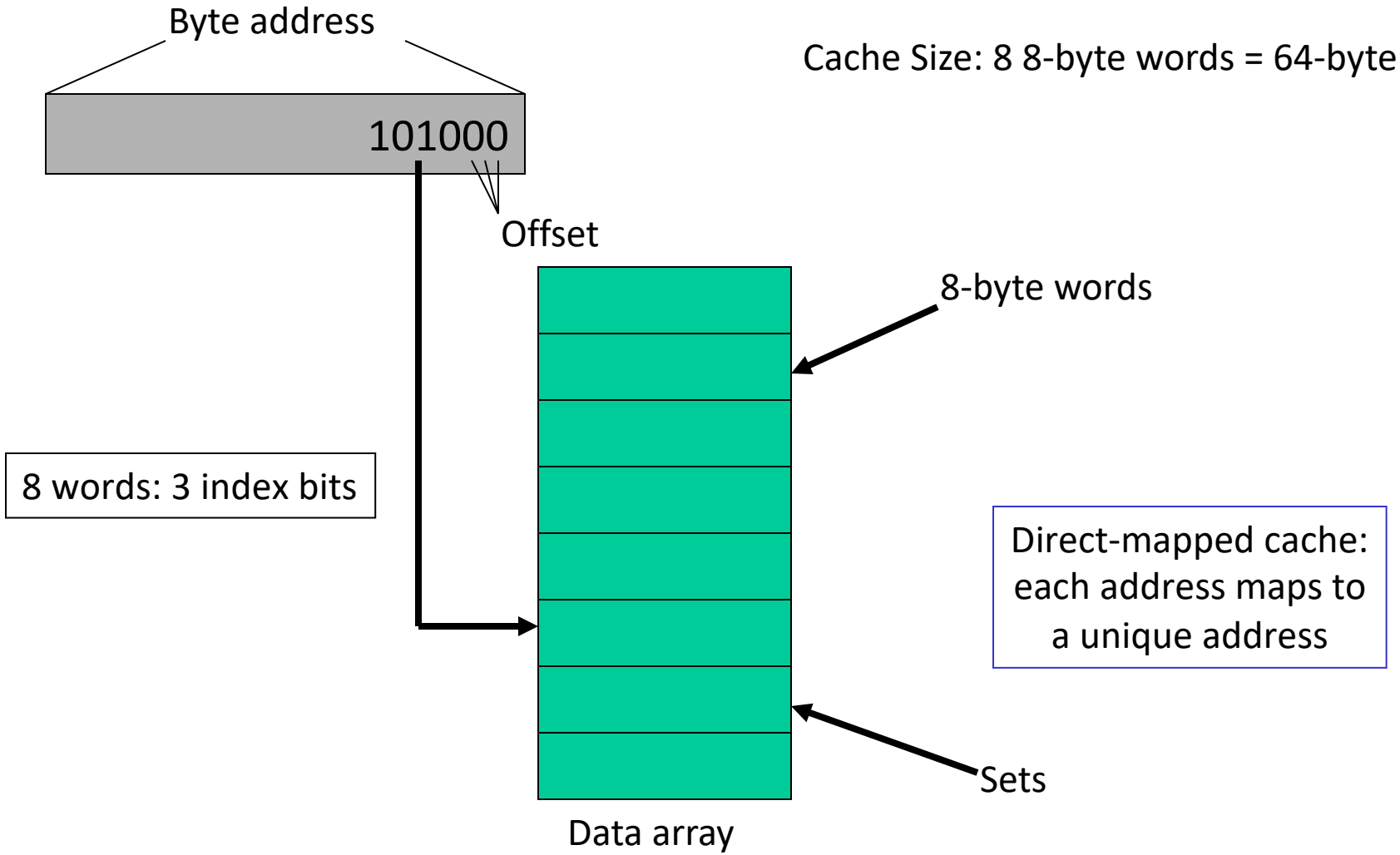
- Most commonly in the processor design context: an automatically-managed memory structure based on SRAM
  - Memorize in SRAM the most frequently accessed DRAM memory locations to avoid repeatedly paying for the DRAM access latency
  - Management policies
    - What data bring to cache?
    - What data keep in cache?
    - …

A Basic Hardware Cache

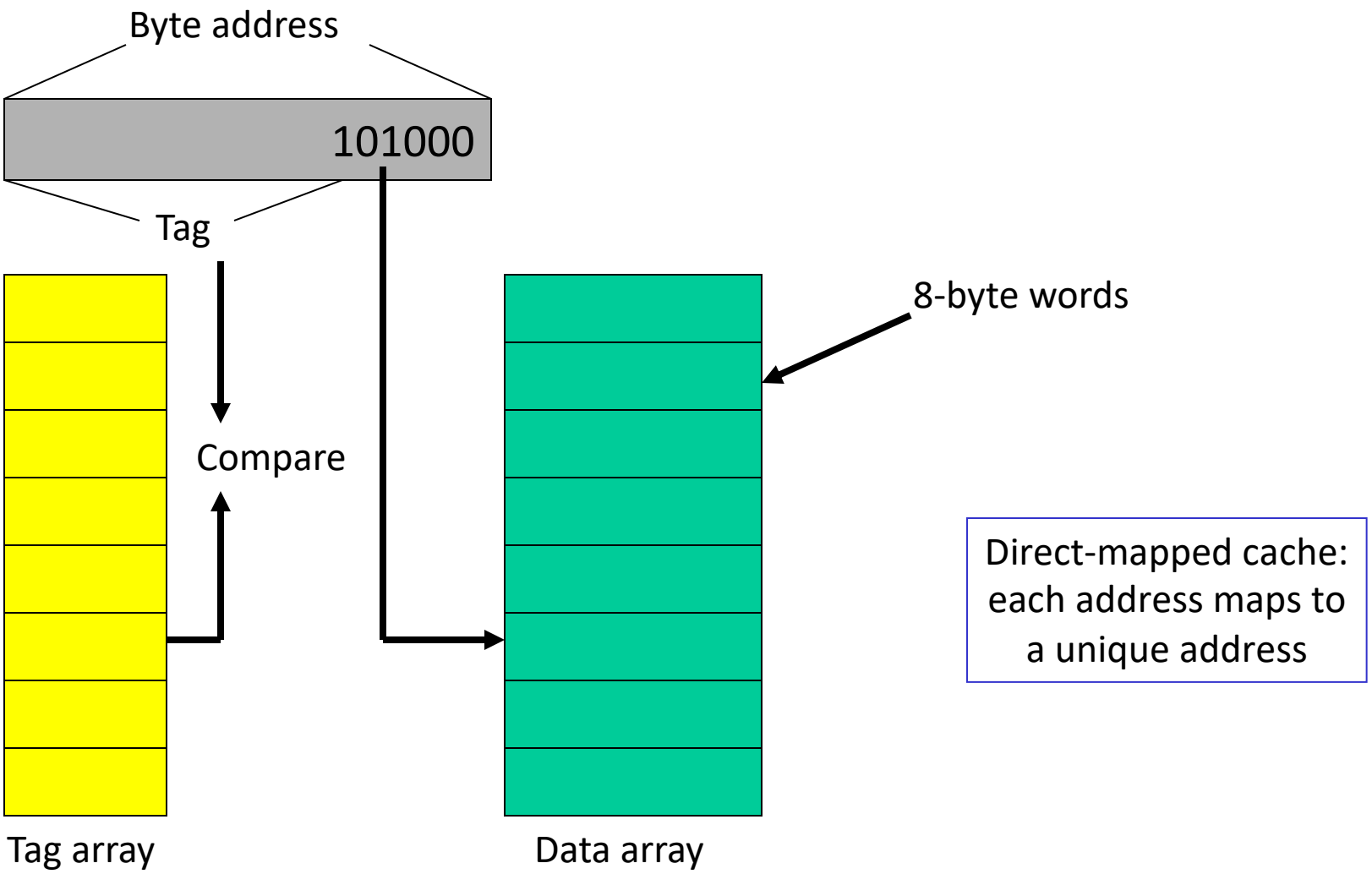
- We will start with a basic hardware cache example and cache operations
- Then, we will formalize some cache basic concepts
- Last, we will examine a multitude of ideas and innovations to make cache performance better

Accessing the Cache

- Example

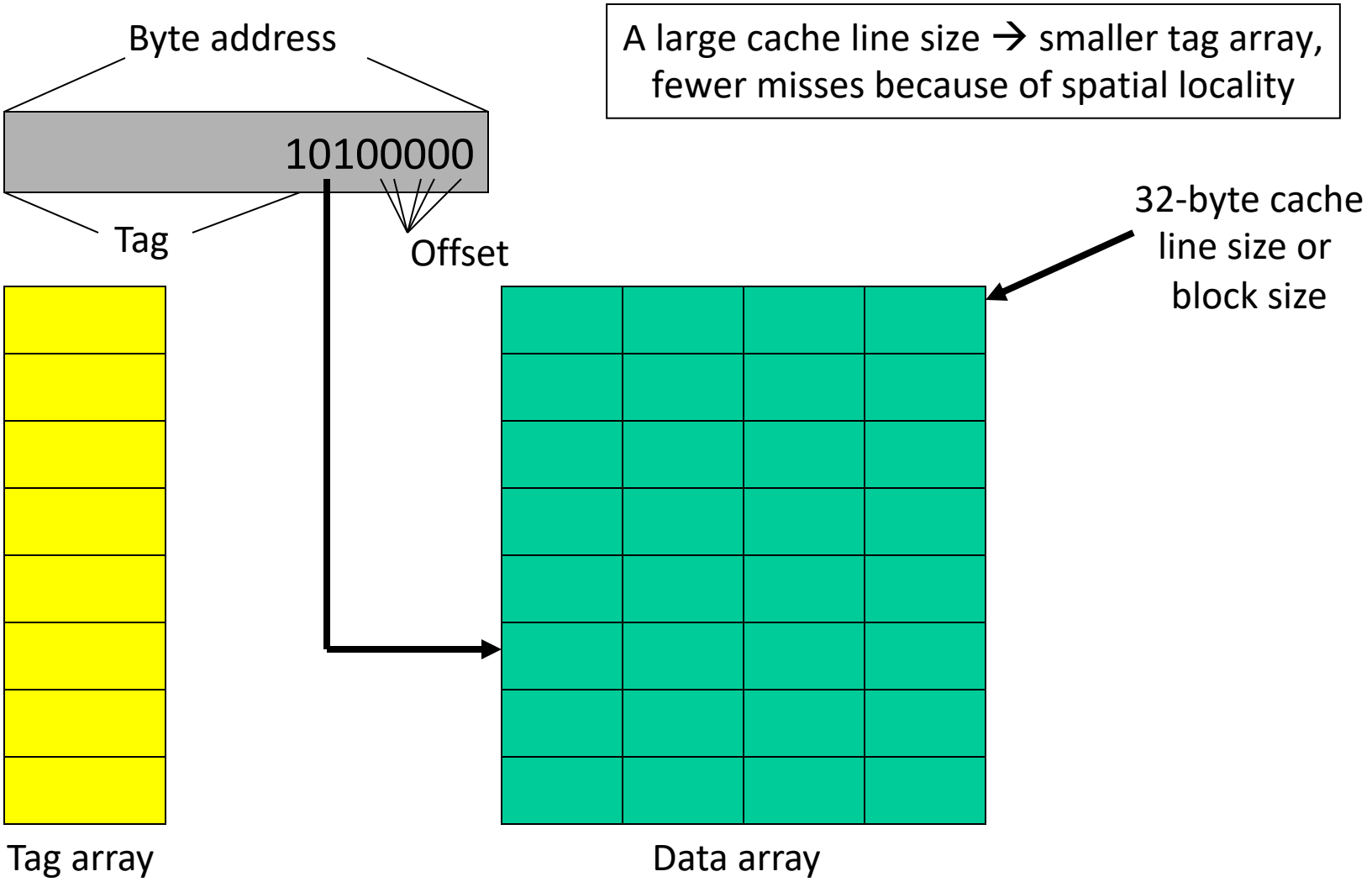


The Tag Array

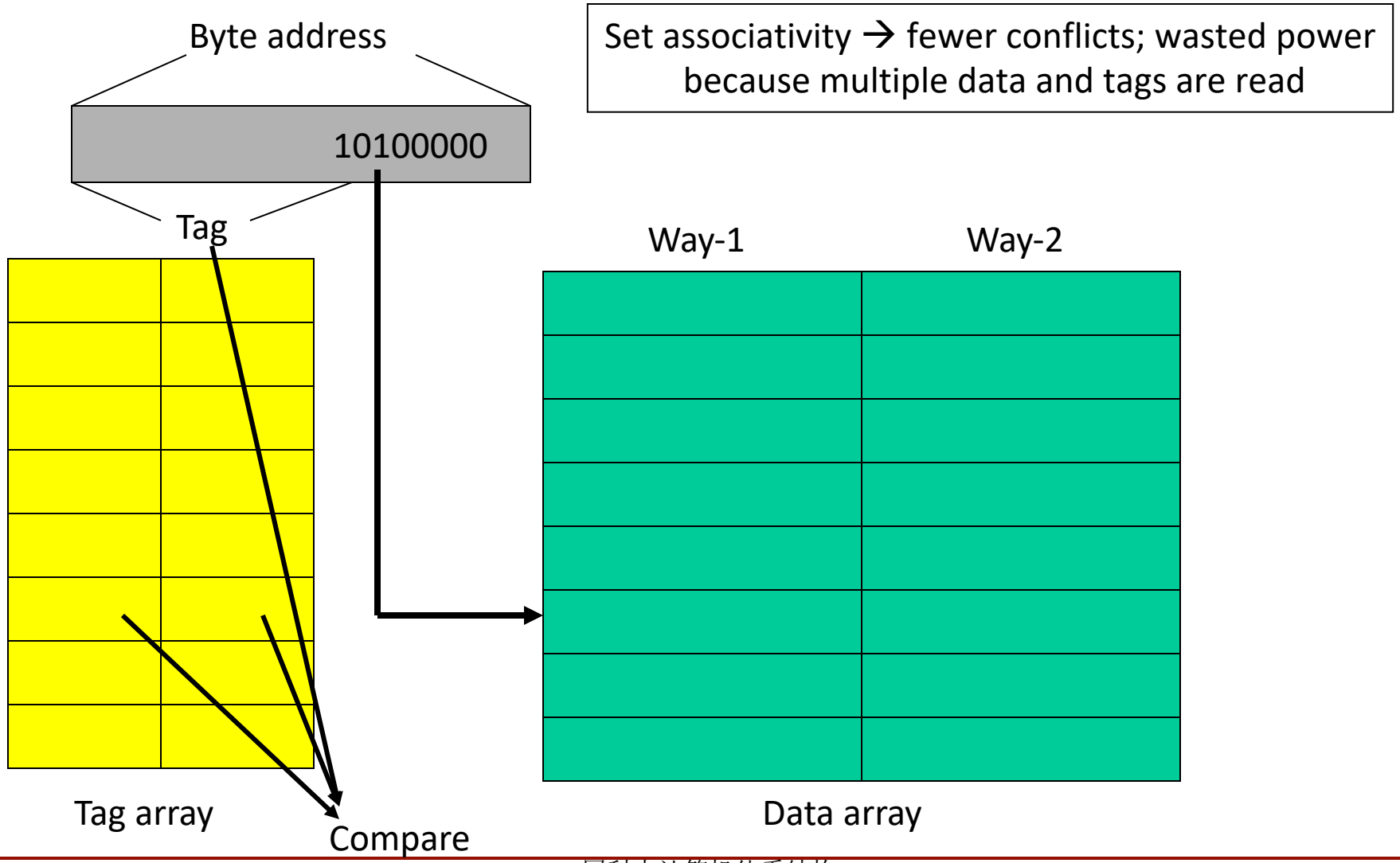


Direct-mapped cache: each address maps to a unique address

Increasing Block Size



Associativity



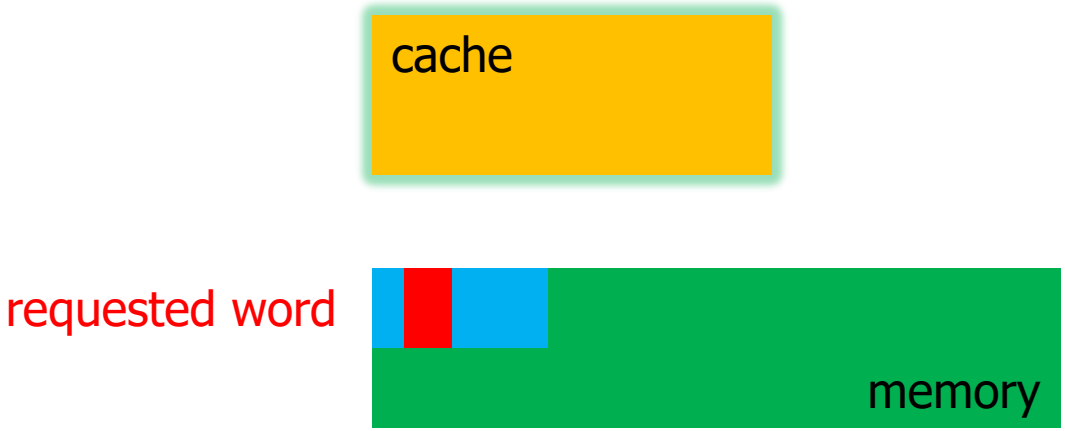
Set associativity → fewer conflicts; wasted power because multiple data and tags are read

Example

- 32 KB 4-way set-associative data cache array with 32 byte line sizes, assume a 40-bit address
- How many sets? 256
- How many index bits (8), offset bits (5), tag bits (27)?
- How large is the tag array (27 Kb)?

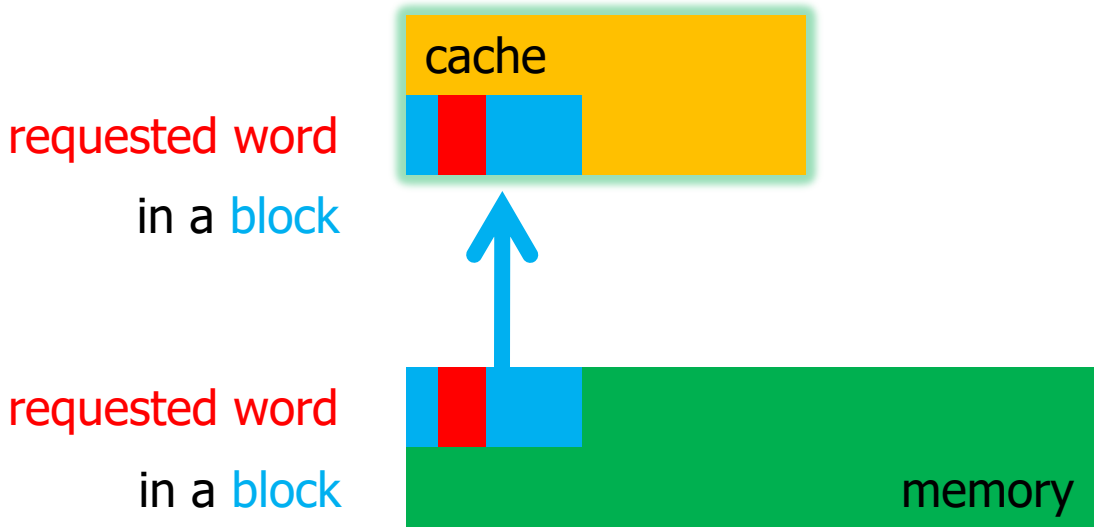
Cache Basics: Cache Block/Line

- Block (line)（缓存块）: Unit of storage in the cache
  - Memory is logically divided into cache blocks that map to locations in the cache
  - A fixed-size collection of data **containing the requested word**, retrieved from the main memory and placed into the cache



Cache Block/Line

- A fixed-size collection of data **containing the requested word**, retrieved from the main memory and placed into the cache



Caching Basics

- On a reference:
  - HIT: If in cache, use cached data instead of accessing memory
  - MISS: If not in cache, bring block into cache
    - Maybe have to kick something else out to do it（Replacement）
- Some important cache design decisions
  - Placement: where and how to place/find/index a block in cache?
  - Replacement: what data to remove/evict/replace to make room in cache?
  - Granularity of management: large or small blocks? Subblocks?
  - Write policy: what do we do about writes? Write-back, write-through
  - Instructions/data: do we treat them separately?

Types of Cache Misses

- Compulsory misses: happens the first time a memory word is accessed – the misses for an infinite cache
- Capacity misses: happens because the program touched many other words before re-touching the same word – the misses for a fully-associative cache
- Conflict misses: happens because two words map to the same location in the cache – the misses generated while moving from a fully-associative to a direct-mapped cache
- Sidenote: can a fully-associative cache have more misses than a direct-mapped cache of the same size?

Reducing Miss Rate

- Large block size – reduces compulsory misses, reduces miss penalty in case of spatial locality – increases traffic between different levels, space waste, and conflict misses
- Large cache – reduces capacity/conflict misses – access time penalty
- High associativity – reduces conflict misses
  - rule of thumb: 2-way cache of capacity N/2 has the same miss rate as 1-way cache of capacity N – more energy

More Caches Basics

- L1 caches are split as instruction and data; L2 and L3 are unified
- The L1/L2 hierarchy can be inclusive, exclusive, or non-inclusive
- On a write, you can do write-allocate or write-no-allocate
- On a write, you can do writeback or write-through; write-back reduces traffic, write-through simplifies coherence
- Reads get higher priority; writes are usually buffered
- L1 does parallel tag/data access; L2/L3 does serial tag/data

Tolerating Miss Penalty

- Out of order execution: can do other useful work while waiting for the miss – can have multiple cache misses
  - Cache controller has to keep track of multiple outstanding misses (non-blocking cache)

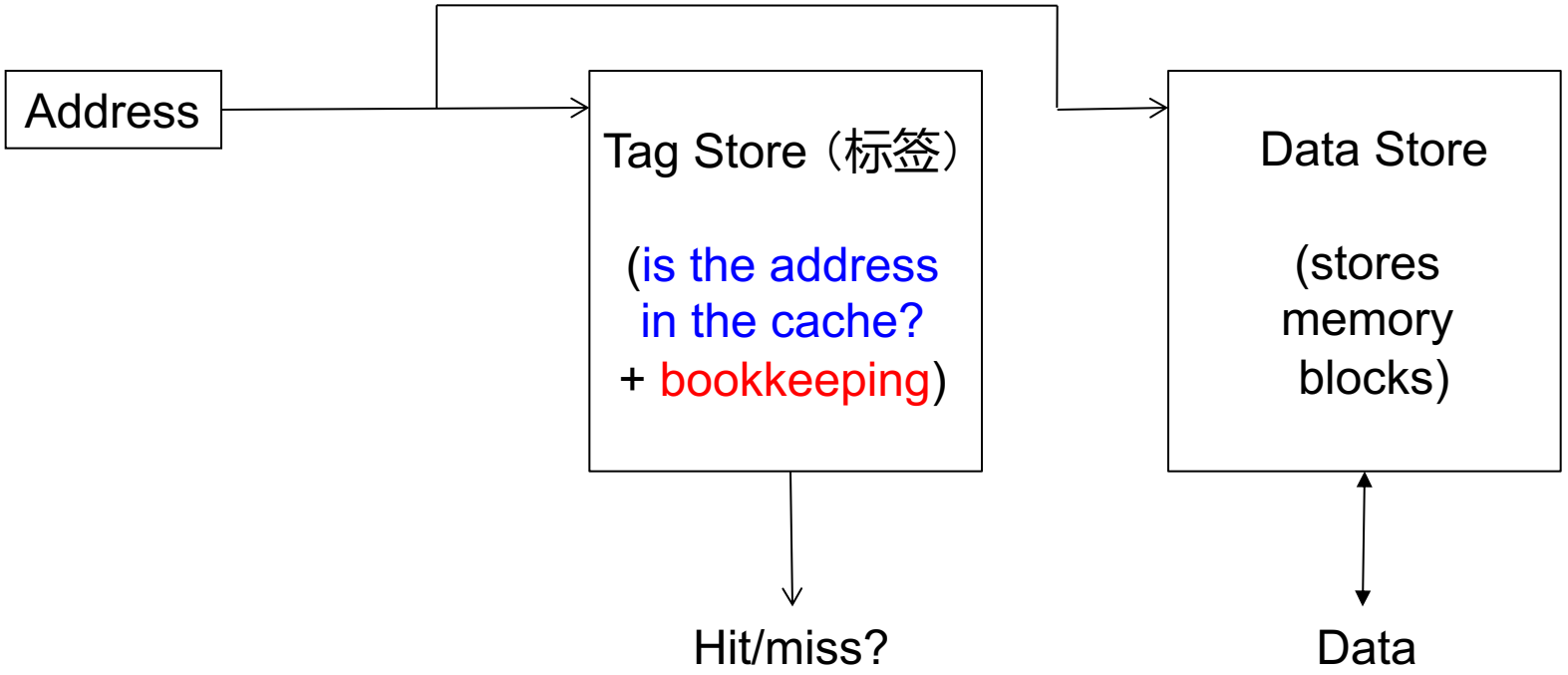
- Hardware prefetching into prefetch buffers – aggressive prefetching can increase contention for buses

Prefetching

- Hardware prefetching can be employed for any of the cache levels
- It can introduce cache pollution – prefetched data is often placed in a separate prefetch buffer to avoid pollution – this buffer must be looked up in parallel with the cache access
- Aggressive prefetching increases "coverage", but leads to a reduction in "accuracy" – wasted memory bandwidth
- Prefetches must be timely: they must be issued sufficiently in advance to hide the latency, but not too early (to avoid pollution and eviction before use)

Cache Abstraction and Structure

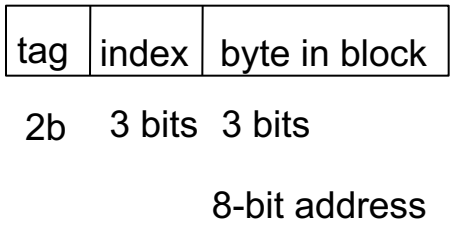
Cache Abstraction and Metrics



- Cache hit rate = (# hits) / (# hits + # misses) = (# hits) / (# accesses)
- Average memory access time (AMAT)
  = (hit-rate * hit-latency) + (miss-rate * miss-latency) (include hit latency)
- Aside: *Is reducing AMAT always beneficial for performance?*
  - *Only considering latency, doest not consider parallelism, e.g., banking*

Blocks and Addressing the Cache

- Memory is logically divided into fixed-size blocks
- Each block maps to a location in the cache, determined by the index (索引) bits or set (组) bits in the address
  - used to index into the tag and data stores



- Cache access:

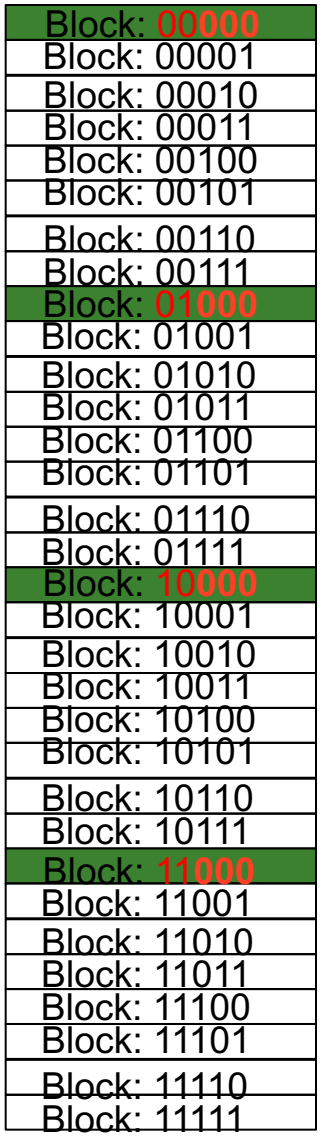
  1) index into the tag and data stores with index bits in address

  2) check valid bit in tag store

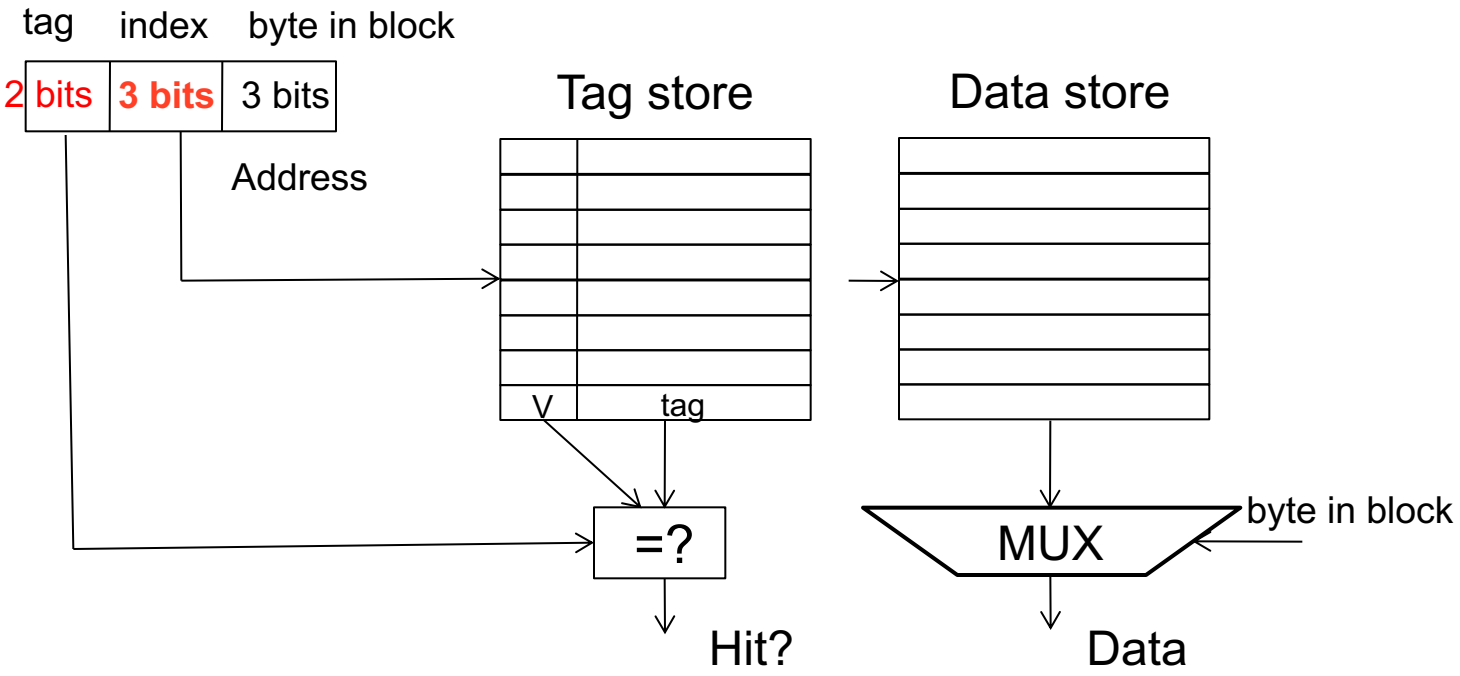
  3) compare tag bits in address with the stored tag in tag store

- If a block is in the cache (cache hit), the stored tag should be valid and match the tag of the block

Direct-Mapped Cache : Placement and Access



- Assume byte-addressable memory: 256 bytes, 8-byte blocks → 32 blocks
- Assume cache: 64 bytes, 8 blocks
  - Direct-mapped: A block can go to only one location



  - Addresses with same index contend for the same location
    - Cause conflict misses

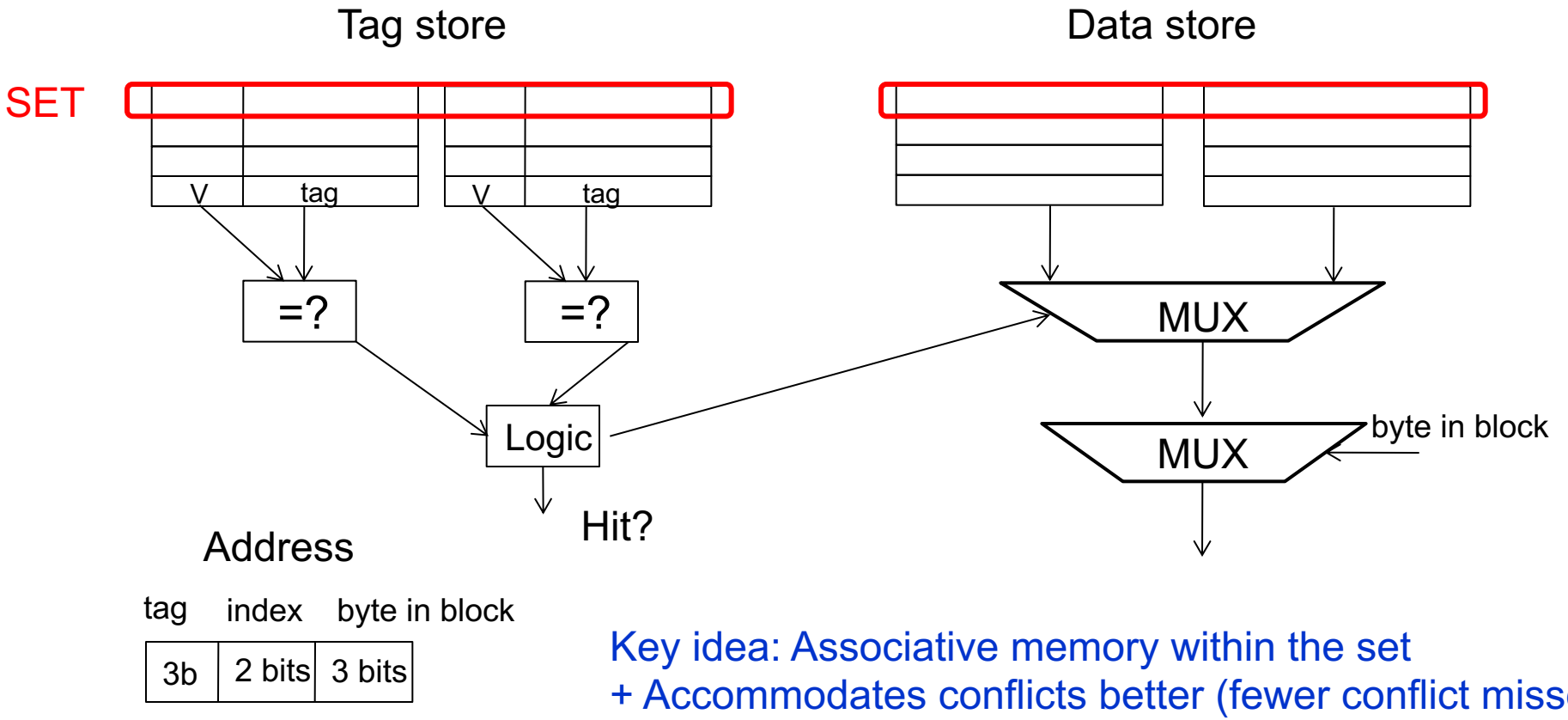
Direct-Mapped Caches

- Direct-mapped cache: Two blocks in memory that map to the same index in the cache cannot be present in the cache at the same time
  - One index → one entry

- Can lead to 0% hit rate if more than one block accessed in an interleaved manner map to the same index
  - Assume addresses A and B have the same index bits but different tag bits
  - A, B, A, B, A, B, A, B, … → conflict in the cache index
  - All accesses are conflict misses

Set Associativity

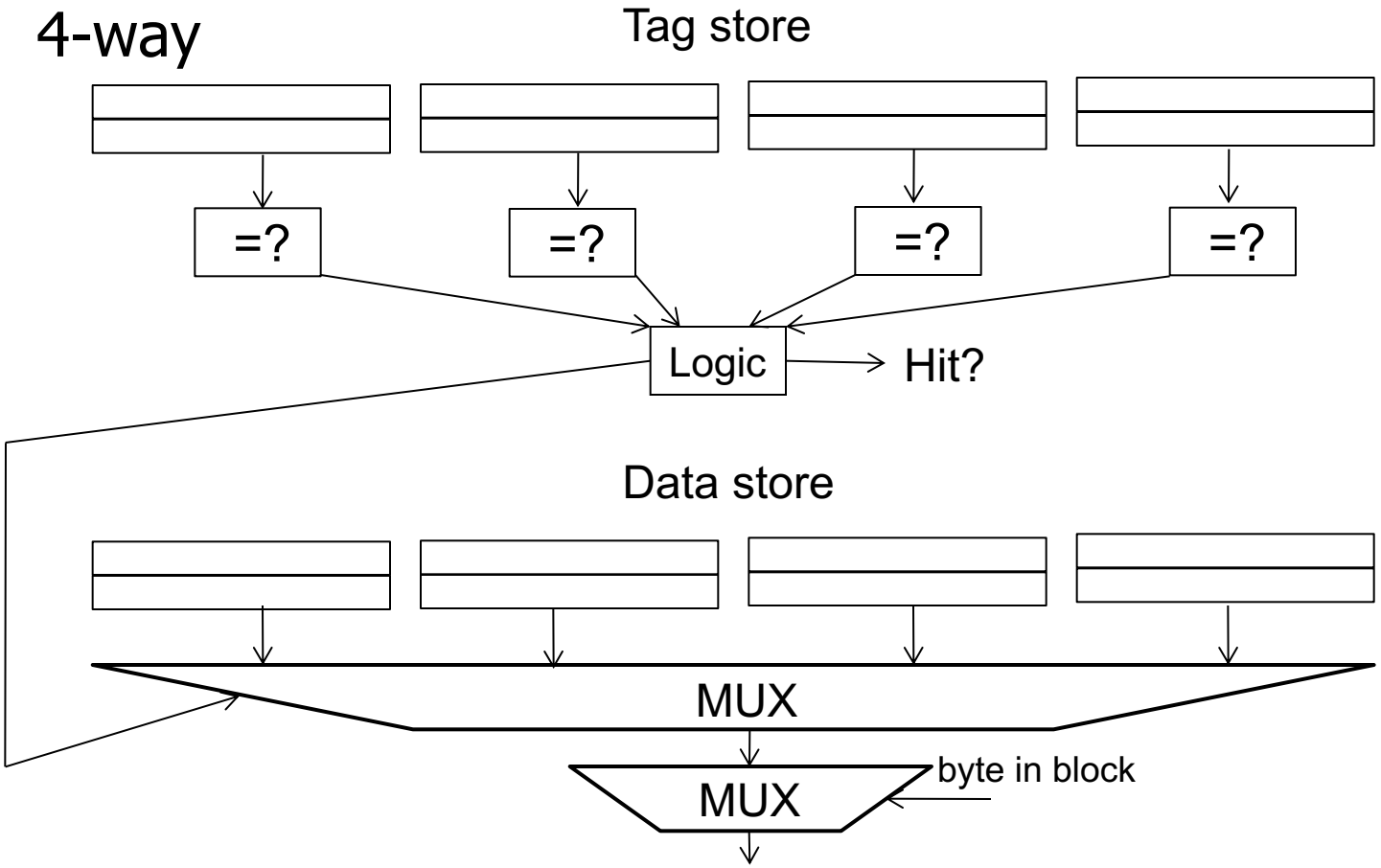
- Addresses 0 and 8 always conflict in direct mapped cache
- Instead of having one column of 8 blocks, have 2 columns of 4 blocks



Key idea: Associative memory within the set
+ Accommodates conflicts better (fewer conflict misses)
-- More complex, slower access, larger tag store

Higher Associativity

- 4-way

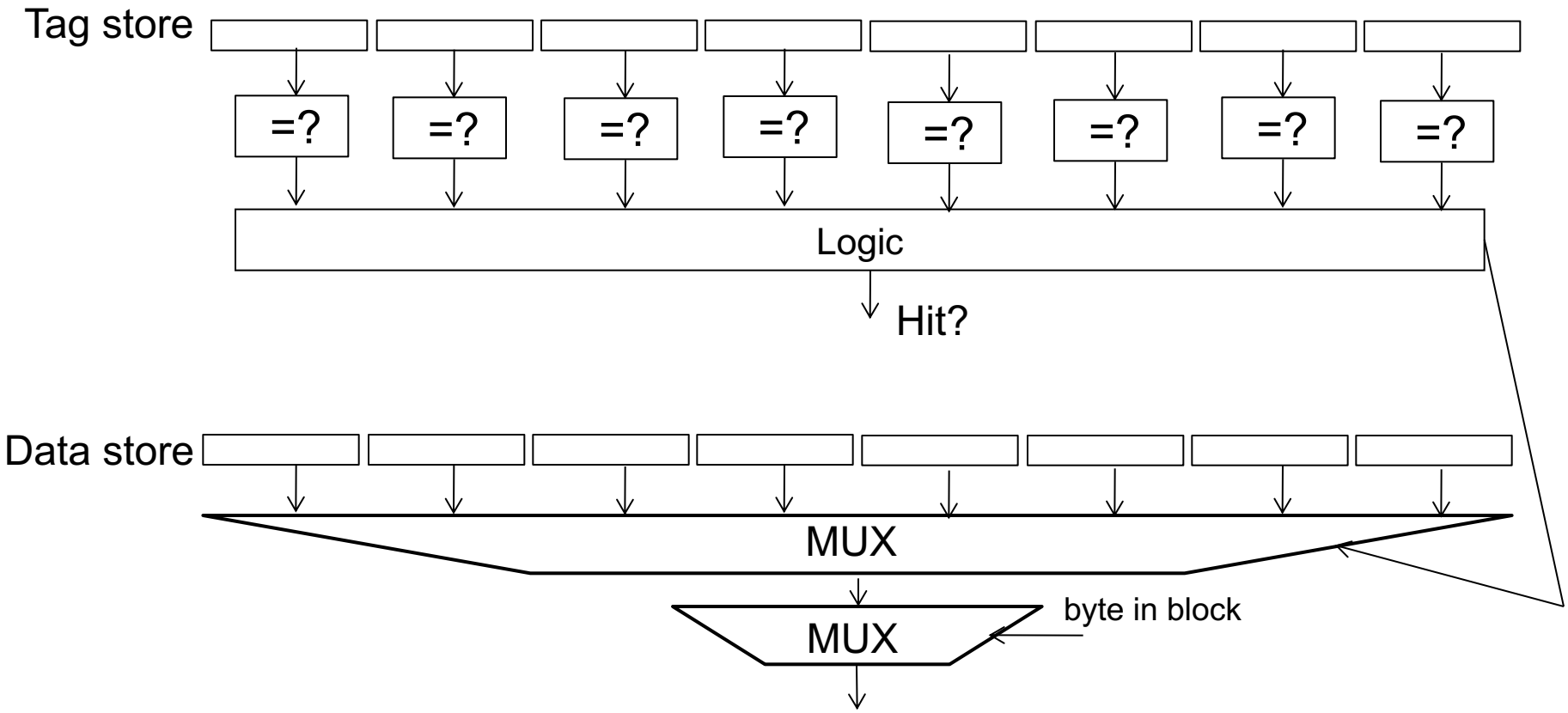


+ Likelihood of conflict misses even lower

-- More tag comparators and wider data mux; larger tags

Full Associativity

- Fully associative cache
  - A block can be placed in any cache location



Associativity (and Tradeoffs)

- Degree of associativity: How many blocks can map to the same index (or set)?

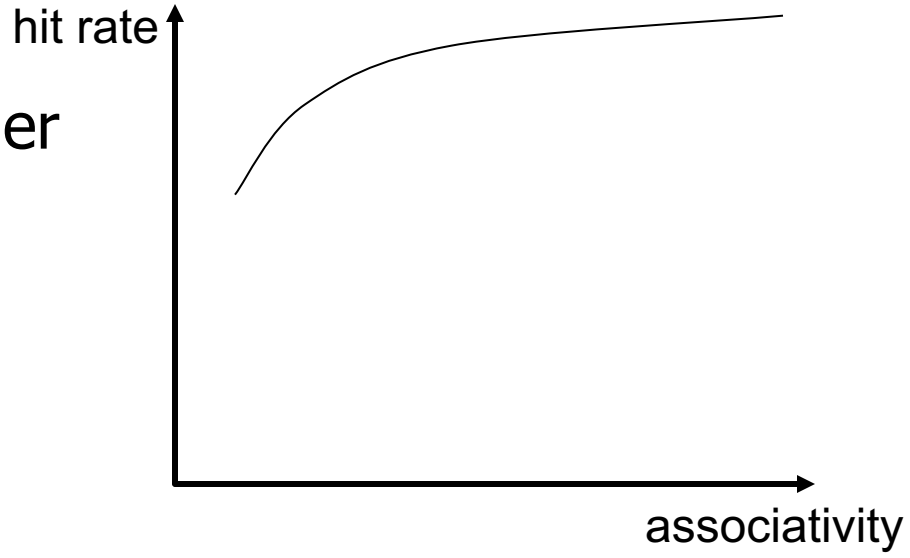
- Higher associativity

  ++ Higher hit rate

  -- Slower cache access time (hit latency and data access latency)

  -- More expensive hardware (more comparators)

- Diminishing returns from higher associativity



Innovations to Reducing Miss Rate

- Victim caches
- Better replacement policies – pseudo-LRU, NRU, DRRIP
  -- insertion, promotion, victim selection
- Prefetching, cache compression

Victim Cache

- A direct-mapped cache suffers from misses because multiple pieces of data map to the same location
- The processor often tries to access data that it recently discarded – all discards are placed in a small victim cache (4 or 8 entries) – the victim cache is checked before going to L2
- Can be viewed as additional associativity for a few sets that tend to have the most conflicts

Issues in Set-Associative Caches

- Think of each block in a set having a "priority"
  - Indicating how important it is to keep the block in the cache
- Key issue: How do you determine/adjust block priorities?
- There are three key decisions in a set:
  - Insertion, promotion, eviction (replacement)

- Insertion: What happens to priorities on a cache fill?
  - Where to insert the incoming block, whether or not to insert the block
- Promotion: What happens to priorities on a cache hit?
  - Whether and how to change block priority
- Eviction/replacement: What happens to priorities on a cache miss?
  - Which block to evict and how to adjust priorities

Eviction/Replacement Policy

- Which block in the set to replace on a cache miss?
  - Any invalid block first
  - If all are valid, consult the replacement policy
    - Random
    - FIFO
    - Least recently used (how to implement?)
    - Not most recently used
    - Least frequently used?
    - Least costly to re-fetch?
      - Why would memory accesses have different cost?
    - Hybrid replacement policies
    - Optimal replacement policy?

These Slides Maybe Covered in the Next Lecture

Implementing LRU

- Idea: Evict the least recently accessed block
- Problem: Need to keep track of access ordering of blocks

- Question: 2-way set associative cache:
  - What do you need to implement LRU perfectly?

- Question: 4-way set associative cache:
  - What do you need to implement LRU perfectly?
  - How many different orderings possible for the 4 blocks in the set?
  - How many bits needed to encode the LRU order of a block?
  - What is the logic needed to determine the LRU victim?

Approximations of LRU

- Most modern processors do not implement "true LRU" (also called "perfect LRU") in highly-associative caches

- Why?
  - True LRU is complex
  - LRU is an approximation to predict locality anyway (i.e., not the best possible cache management policy)

- Examples:
  - NRU, not most recently used
  - DRRIP

Eviction/Replacement Policy

- NRU: every block in a set has a bit; the bit is made zero when the block is touched; if all are zero, make all one; a block with bit set to 1 is evicted

- DRRIP: using multiple NRU bits (3 bits), incoming blocks are set to the high number, so they are close to be being evicted; similar to place the incoming block near the head of LRU list instead of near the tail

Cache Replacement Policy: LRU or Random

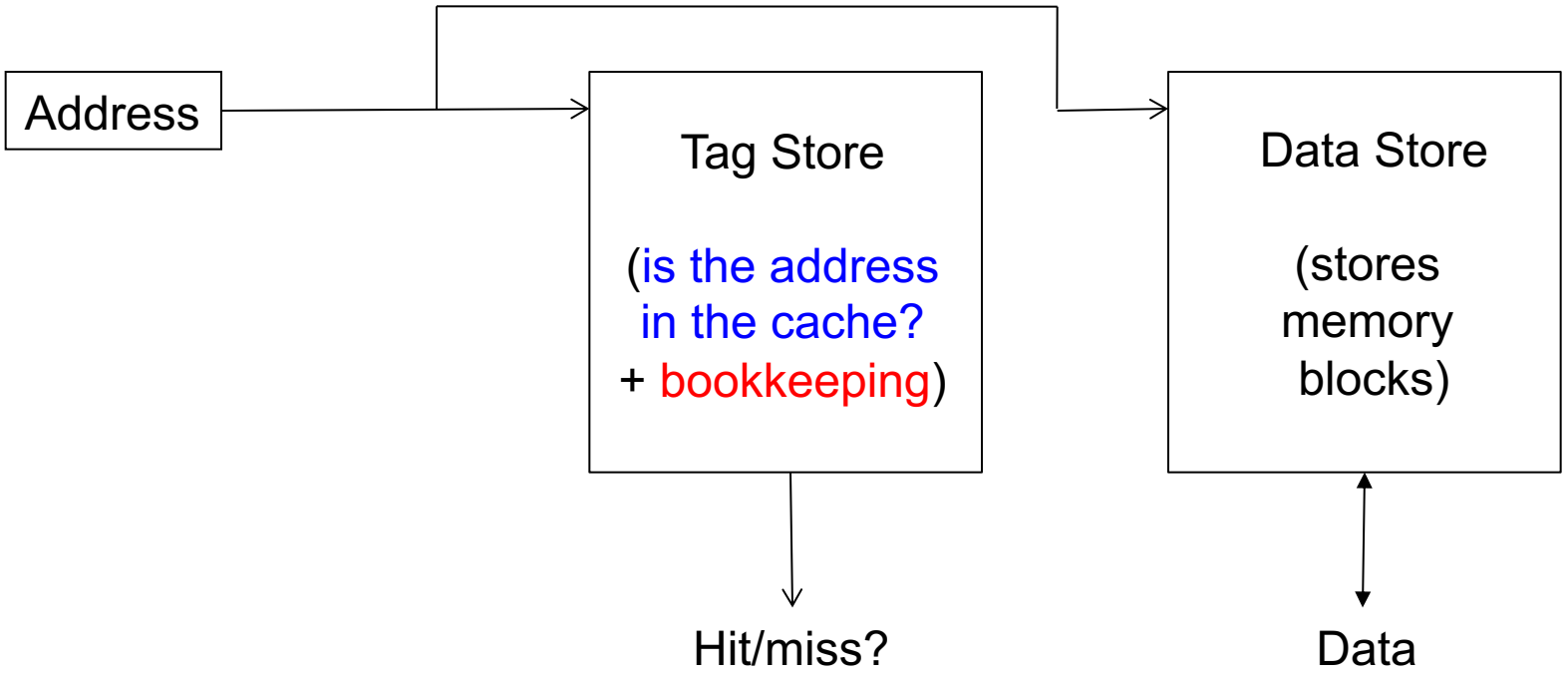
- LRU vs. Random: Which one is better?
  - Example: 4-way cache, cyclic references to A, B, C, D, E
    - 0% hit rate with LRU policy
- Set thrashing: When the "program working set" in a set is larger than set associativity
  - Random replacement policy is better when thrashing occurs
- In practice:
  - Depends on workload
  - Average hit rate of LRU and Random are similar
- Best of both Worlds: Hybrid of LRU and Random
  - How to choose between the two? Set sampling
    - See Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.

What Is the Optimal Replacement Policy?

- Belady's OPT
  - Replace the block that is going to be referenced furthest in the future by the program
  - Belady, "A study of replacement algorithms for a virtual-storage computer," IBM Systems Journal, 1966.
  - How do we implement this? Simulate?

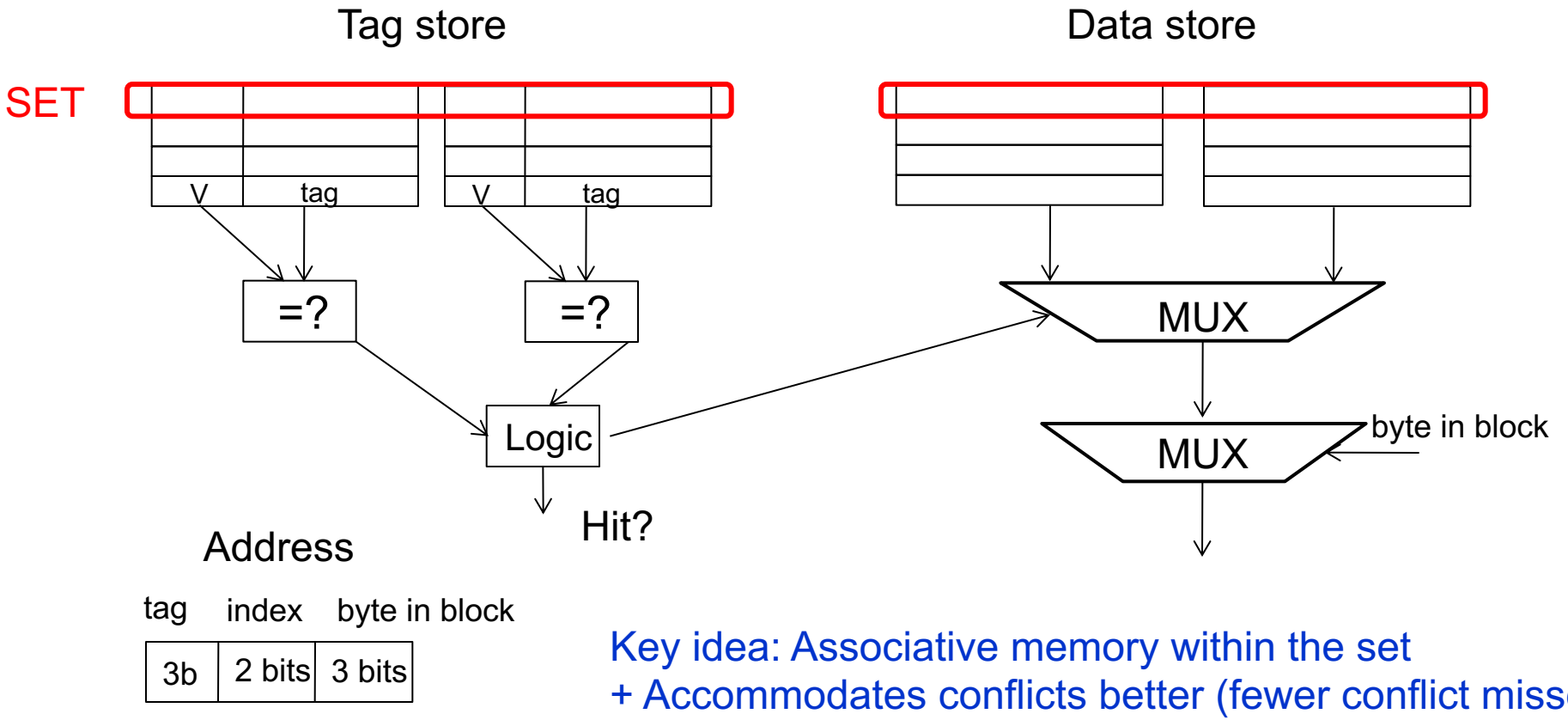
- Is this optimal for minimizing miss rate?
- Is this optimal for minimizing execution time?
  - No. Cache miss latency/cost varies from block to block!
  - Two reasons: Remote vs. local caches
  - Qureshi et al. "A Case for MLP-Aware Cache Replacement," ISCA 2006.

Recall: Cache Structure



Recall: Set Associativity

- Addresses 0 and 8 always conflict in direct mapped cache
- Instead of having one column of 8, have 2 columns of 4 blocks



Key idea: Associative memory within the set
+ Accommodates conflicts better (fewer conflict misses)
-- More complex, slower access, larger tag store

What's In A Tag Store Entry?

- Valid bit
- Tag
- Replacement policy bits

- Dirty bit?
  - Write back vs. write through caches

Handling Writes (I)

- When do we write the modified data in a cache to the next level?
  - Write through: At the time the write happens
  - Write back: When the block is evicted

- Write-back: **写回模式:** 更新cache时，并不同步更新memory

  + Can combine multiple writes to the same block before eviction
    - Potentially saves bandwidth between cache levels + saves energy

  -- Need a bit in the tag store indicating the block is "dirty/modified"

- Write-through: **写直达模式:** CPU向cache写入数据时，同时向memory写

  + Simpler

  + All levels are up to date. Consistency: Simpler cache coherence because no need to check close-to-processor caches' tag stores for presence

  -- More bandwidth intensive; no combining of writes

Handling Writes (II)

- Do we allocate a cache block on a write miss?
  - Yes in default

- Allocate on write miss:**将要写的地址所在的块先从memory 读到 cache 中，然后写cache**

  + Can combine writes instead of writing each of them individually to next level

  + Simpler because write misses can be treated the same way as read misses

  -- Requires transfer of the whole cache block

- No-allocate: **将要写的内容直接写回memory**

  + Conserves cache space if locality of writes is low (potentially better cache hit rate)

Handling Writes (III)

- What if the processor writes to an entire block over a small amount of time?
- Is there any need to bring the block into the cache from memory in the first place?
- Why do we not simply write to only a *portion* of the block, i.e., subblock
  - E.g., 4 bytes out of 64 bytes
  - Problem: Valid and dirty bits are associated with the entire 64 bytes, not with each individual 4 bytes

Subblocked (Sectored) Caches

- Idea: Divide a block into subblocks (or sectors)
  - Have separate valid and dirty bits for each subblock (sector)
  - Allocate only a subblock (or a subset of subblocks) on a request

++ No need to transfer the entire cache block into the cache
(A write simply validates and updates a subblock)

++ More freedom in transferring subblocks into the cache (a cache block does not need to be in the cache fully)
(How many subblocks do you transfer on a read?)

-- More complex design

-- May not exploit spatial locality fully

Instruction vs. Data Caches

- Separate or Unified?

- Pros and Cons of Unified:
  + Dynamic sharing of cache space: no overprovisioning that might happen with static partitioning (i.e., separate I and D caches)
  -- Instructions and data can thrash each other (i.e., no guaranteed space for either)
  -- I and D are accessed in different places in the pipeline. Where do we place the unified cache for fast access? Main reason using separate caches in most of processors
- First level caches are almost always split
  - Mainly for the last reason above
- Higher level caches are almost always unified
  - the first reason

Multi-level Caching in a Pipelined Design

- First-level caches (instruction and data)
  - Decisions very much affected by cycle time
  - Small, lower associativity; latency is critical
  - Tag store and data store accessed in parallel
- Second-level caches
  - Decisions need to balance hit rate and access latency
  - Usually large and highly associative; latency not as important
  - Tag store and data store accessed serially
    - Latency is not important but hit rate, energy saving
- Serial vs. Parallel access of levels
  - Serial: Second level cache accessed only if first-level misses
  - Second level does not see the same accesses as the first
    - First level acts as a filter (filters some temporal and spatial locality)
    - Management policies are therefore different

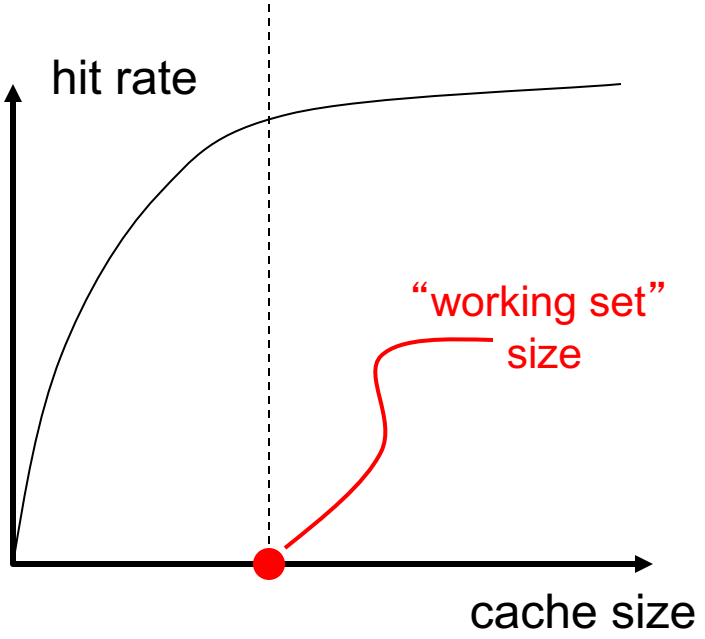
Improving Cache Performance

Cache Parameters vs. Miss/Hit Rate

- Cache size
- Block size
- Associativity
- Replacement policy
- Insertion/Placement policy

Cache Size

- Cache size: total data (not including tag) capacity
  - bigger can exploit temporal locality better
  - not ALWAYS better
- Too large a cache adversely affects hit and miss latency
  - smaller is faster => bigger is slower
  - access time may degrade critical path
- Too small a cache
  - doesn't exploit temporal locality well
  - useful data replaced often
- Working set: the whole set of data the executing application references
  - Within a time interval

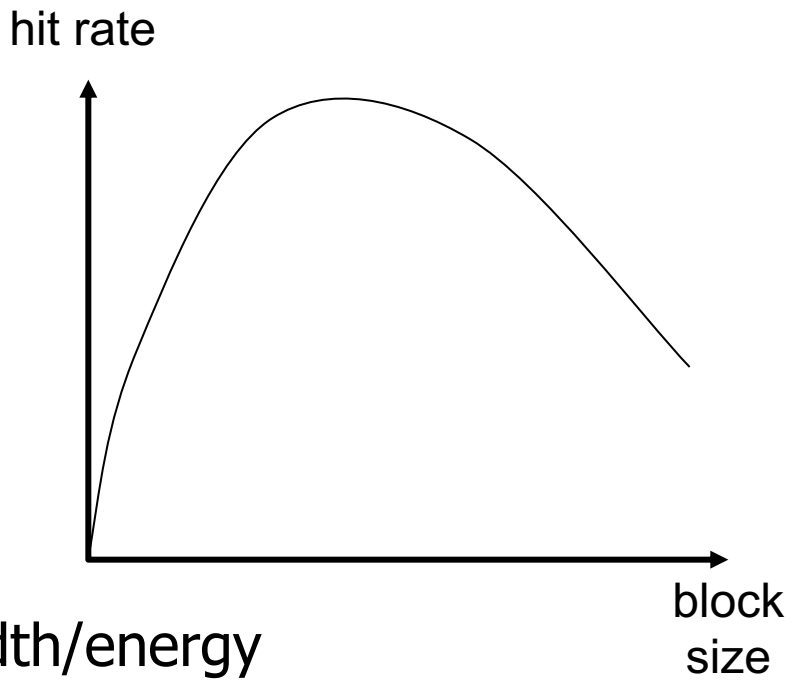


Block Size

- Block size is the data that is associated with an address tag
  - not necessarily the unit of transfer between hierarchies
    - Sub-blocking: A block divided into multiple pieces (each w/ V/D bits)

- Too small blocks
  - don't exploit spatial locality well
  - have larger tag overhead

- Too large blocks
  - too few total # of blocks → less temporal locality exploitation
  - waste of cache space and bandwidth/energy if spatial locality is not high



Large Blocks: Critical-Word and Subblocking

- Large cache blocks can take a long time to fill into the cache
  - fill cache line critical word first
  - restart cache access before complete fill

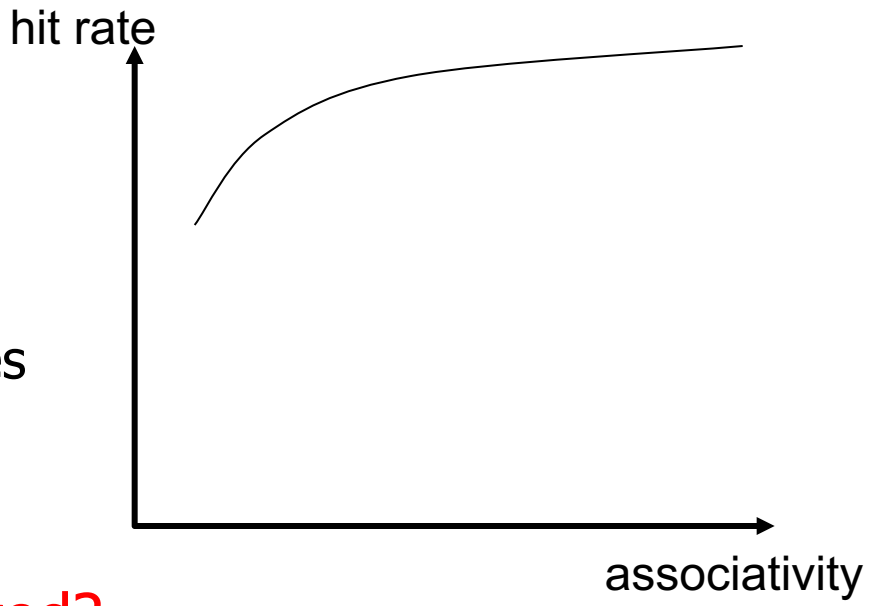
- Large cache blocks can waste bus bandwidth
  - divide a block into subblocks
  - associate separate valid and dirty bits for each subblock
  - Recall: When is this useful?

Associativity

- How many blocks can be present in the same index (i.e., set)?

- Larger associativity
  - lower miss rate (reduced conflicts)
  - higher hit latency and area cost (plus diminishing returns)

- Smaller associativity
  - lower cost
  - lower hit latency
    - Especially important for L1 caches



- Is power of 2 associativity required?

Classification of Cache Misses

- Compulsory miss
  - first reference to an address (block) always results in a miss
  - subsequent references should hit unless the cache block is displaced for the reasons below
- Capacity miss
  - cache is too small to hold everything needed
  - defined as the misses that would occur even in a fully-associative cache (with optimal replacement) of the same capacity
- Conflict miss
  - defined as any miss that is neither a compulsory nor a capacity miss

How to Reduce Each Miss Type

- Compulsory
  - Caching cannot help
  - Prefetching can: Anticipate which blocks will be needed soon
- Conflict
  - More associativity
  - Other ways to get more associativity without making the cache associative
    - Victim cache
    - Better, randomized indexing
    - Software hints?
- Capacity
  - Utilize cache space better: keep blocks that will be referenced
  - Software management: divide working set and computation such that each “computation phase” fits in cache

How to Improve Cache Performance

- Three fundamental goals
- Reducing miss rate
  - Caveat: reducing miss rate can reduce performance if more costly-to-refetch blocks are evicted
- Reducing miss latency or miss cost
- Reducing hit latency or hit cost
- The above three **together** affect performance

Improving Basic Cache Performance

- Reducing miss rate
  - More associativity
  - Alternatives/enhancements to associativity
    - Victim caches, hashing, pseudo-associativity, skewed associativity
  - Better replacement/insertion policies
  - Software approaches
- Reducing miss latency/cost
  - Multi-level caches
  - Critical word first
  - Subblocking/sectoring
  - Better replacement/insertion policies
  - Non-blocking caches (multiple cache misses in parallel)
  - Multiple accesses per cycle
  - Software approaches

Acknowledgements

- EPFL, Onur Mutlu, Digital Design and Computer Architecture, 2020, 2023
- 计算机体系结构:量化研究方法(中文版第六版)
- UCSD CSE 240
- Washington University CSE3 78
- 国科大，胡伟武，计算机体系结构
- CMU, CS 447 Computer Architecture
- UC Berkeley, CS 152 and CS 61C
- 国科大南京学院，安学军等，计算机体系结构
- 浙江大学，计算机体系结构
- 南京大学，计算机体系结构