KING'S College LONDON

Hardware Design

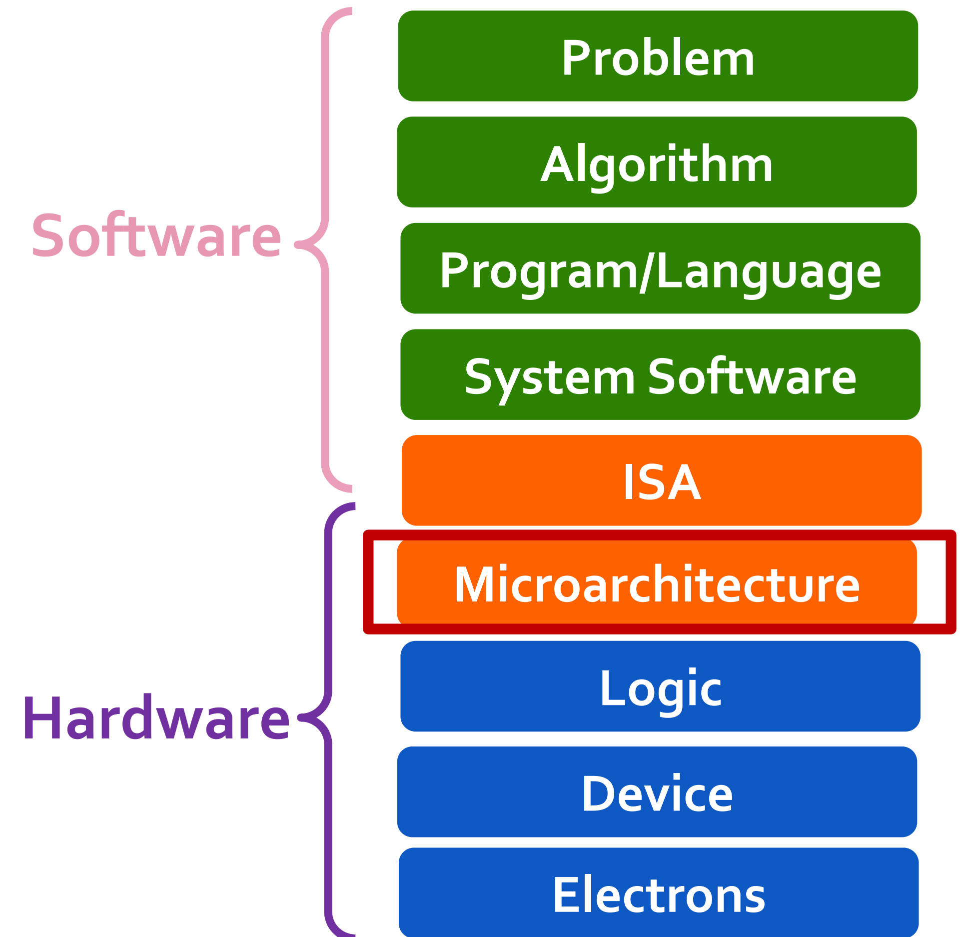
Lecture 8: Cache

Dr. Haiyu Mao

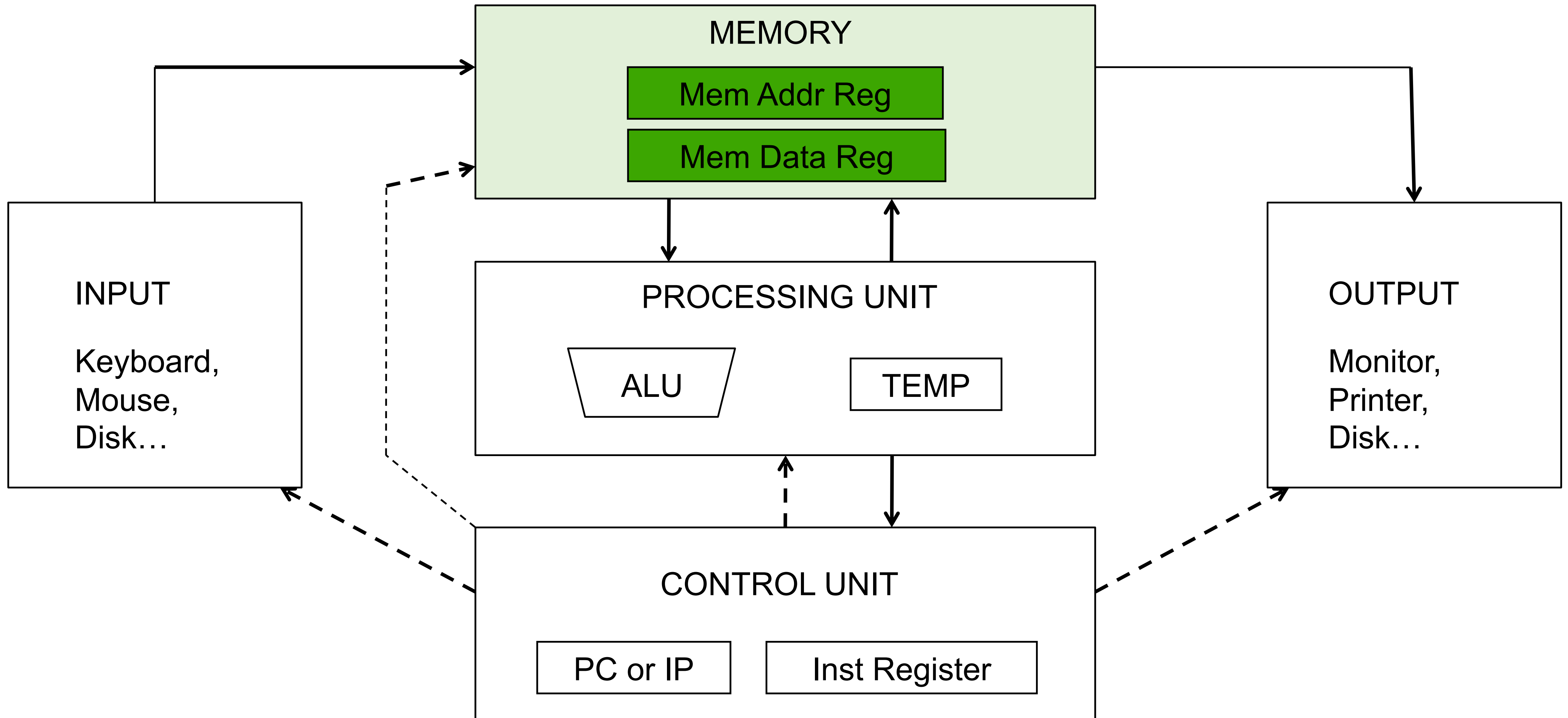
19.03.2026

What We Learned & Will Learn

- ❑ The von Neumann model
- ❑ Instruction Set Architectures (ISA)
- ❑ Assembly programming: LC-3 and MIPS
- ❑ Microarchitecture: basics
- ❑ Microarchitecture: Single-cycle
- ❑ Microarchitecture: Multi-cycle
- ❑ Pipelining
- ❑ **Cache and Memory**

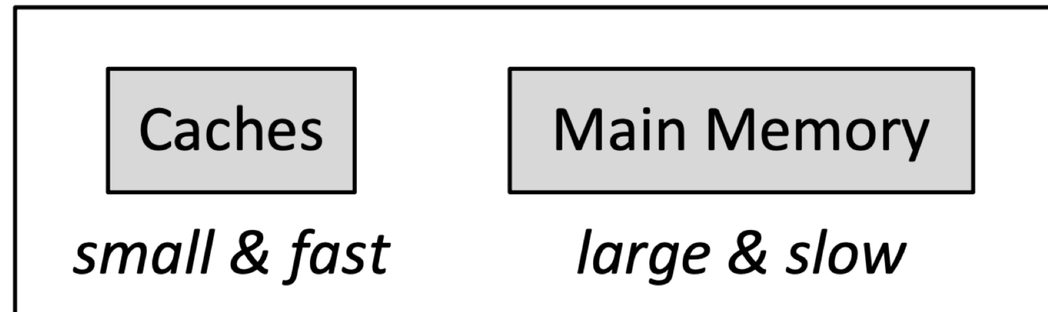


Recall: The von Neumann Model

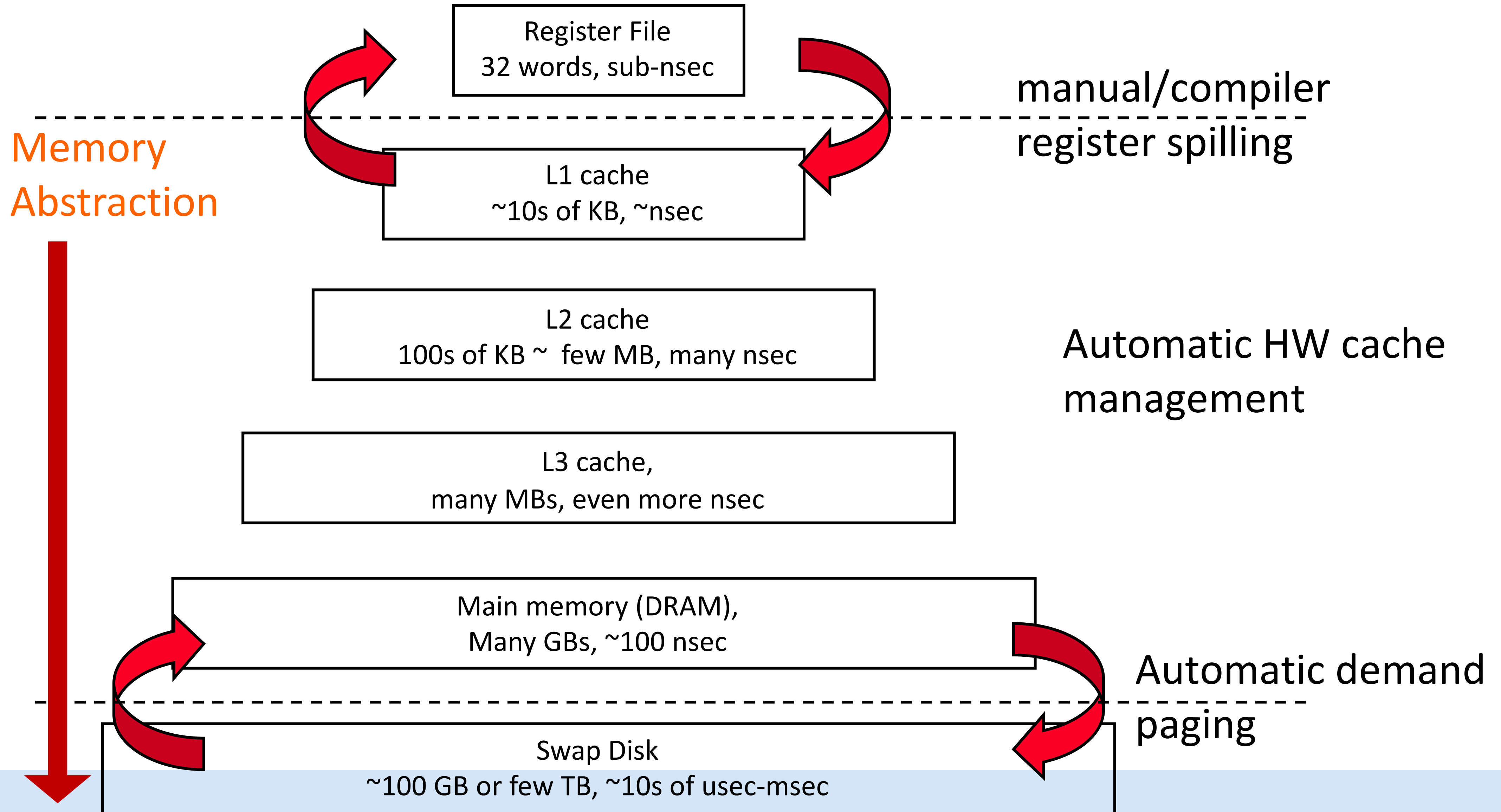


Recall: Why Memory Hierarchy?

- 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) and ensure most of the data the processor needs is kept in the fast(er) level(s)



Recall: A Modern Memory Hierarchy



Recall: 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 nearby memory locations within a window of time
  - most notable examples:
    1. instruction memory references → mostly sequential/streaming
    2. references to arrays/vectors → often streaming/strided

Recall: Caching Basics: Exploit Temporal Locality

- Idea: Store recently accessed data in automatically-managed fast memory (called cache)
- Anticipation: The same memory location will be accessed again soon
- Temporal locality principle
  - Recently accessed data will be accessed again 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."

Recall: Caching Basics: Exploit Spatial Locality

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

Hardware Design

Cache Basics and Operation

L1 cache
~10s of KB, ~nsec

L2 cache
100s of KB ~ few MB, many nsec

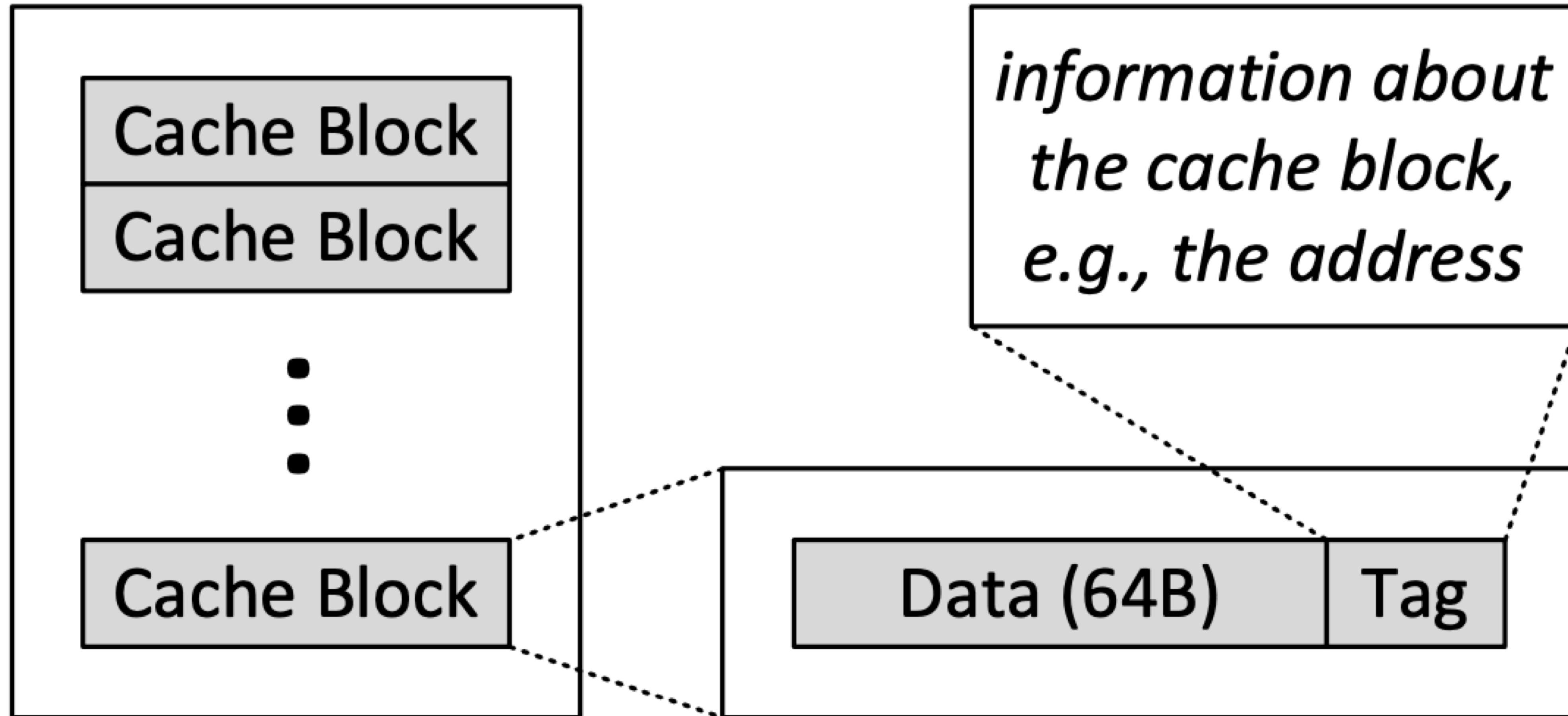
L3 cache,
many MBs, even more nsec

Cache

- Any structure that "memoizes" used (or produced) data
  - to avoid repeating the long-latency operations required to reproduce/fetch the data from scratch
  - e.g., a web cache

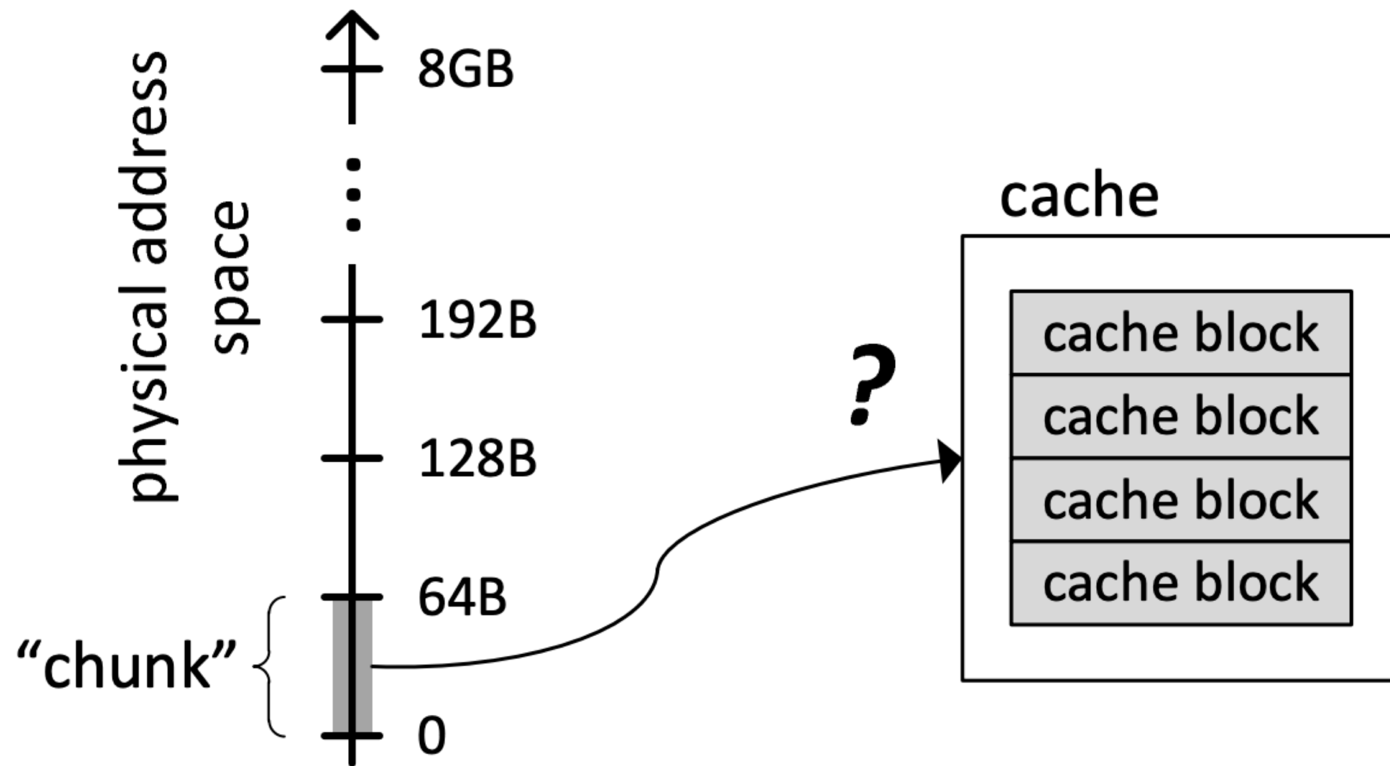
- Most commonly in the processor design context:
  an automatically-managed memory structure
  - e.g., memoize in fast SRAM the most frequently or recently accessed DRAM memory locations to avoid repeatedly paying for the DRAM access latency

Conceptual Picture of a Cache



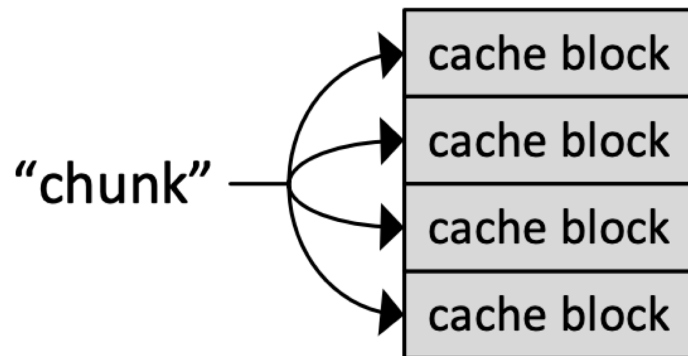
Logical Organization of a Cache (I)

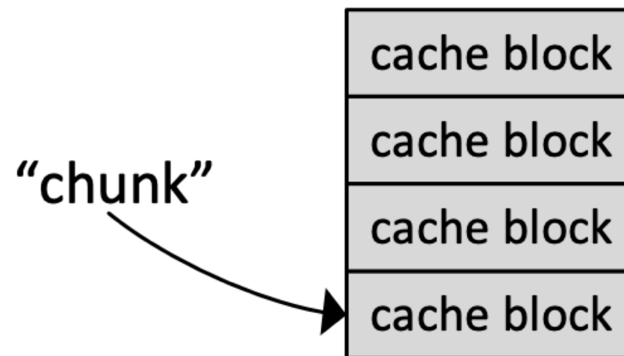
- A key question: How to map chunks of the main memory address space to blocks in the cache?
  - Which location in cache can a given "main memory chunk" be placed in?

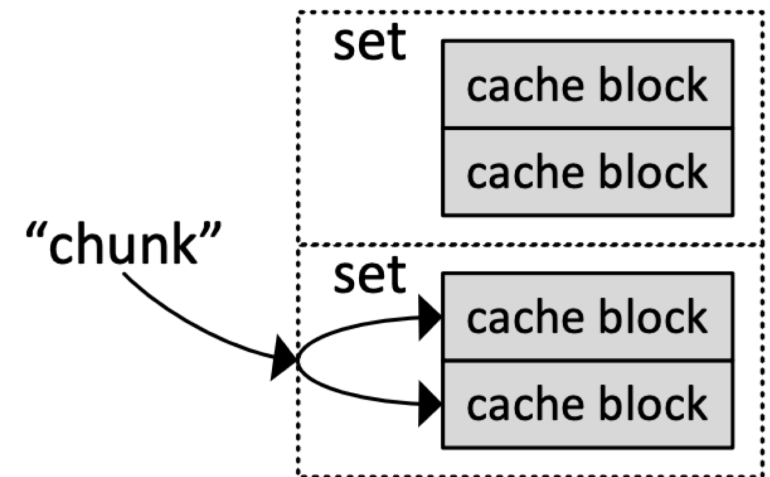


Logical Organization of a Cache (II)

- A key question: How to map chunks of the main memory address space to blocks in the cache?
  - Which location in cache can a given "main memory chunk" be placed in?







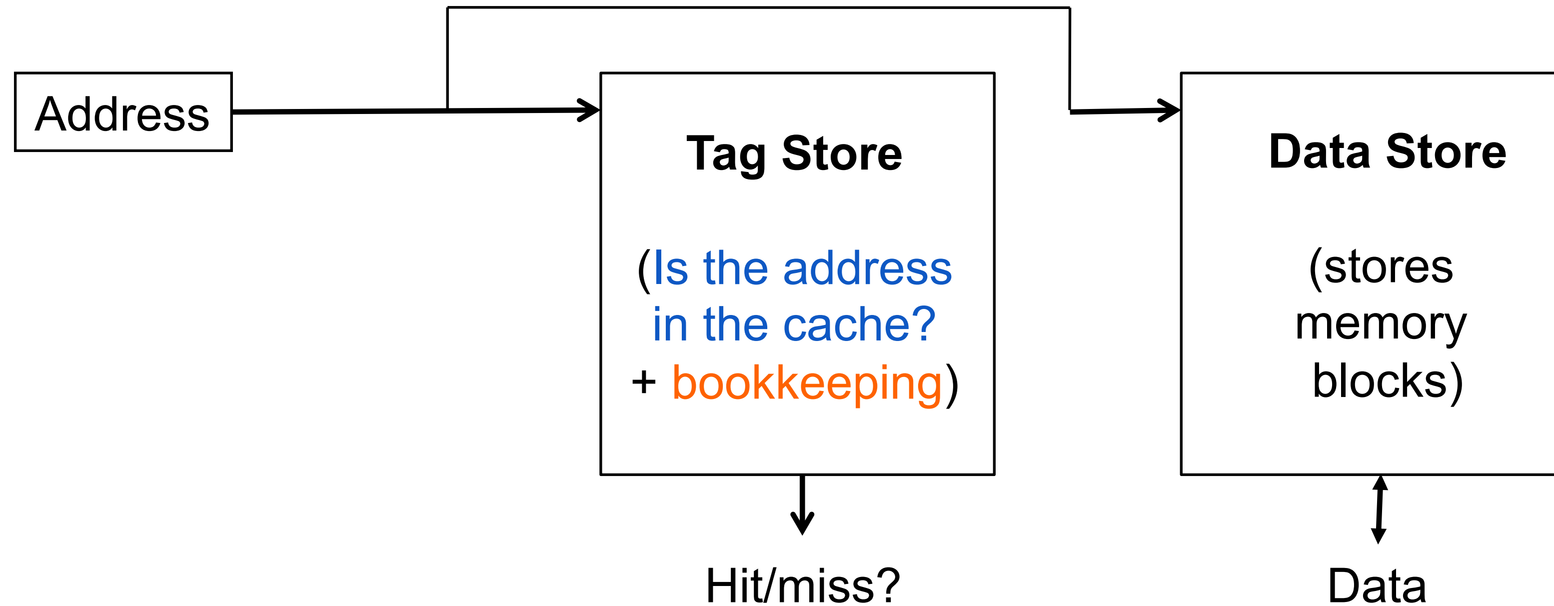
Caching Basics

- **Block (line):** Unit of storage in the cache
  - Memory is logically divided into blocks that map to potential locations in the cache

- On a reference:
  - **HIT**: If in cache, use cached data instead of accessing memory
  - **MISS**: If not in cache, bring block into cache
  - May have to evict some other block

- Some important cache design decisions
  - **Placement**: where and how to place/find a block in cache?
  - **Replacement**: what data to remove to make room in cache?
  - **Granularity of management**: large or small blocks? Subblocks?
  - **Write policy**: what do we do about writes?
  - **Instructions/data**: do we treat them separately?

Cache Abstraction and Metrics



- Cache hit rate = (# hits) / (# hits + # misses) = (# hits) / (# accesses)
- Average memory access time (AMAT)

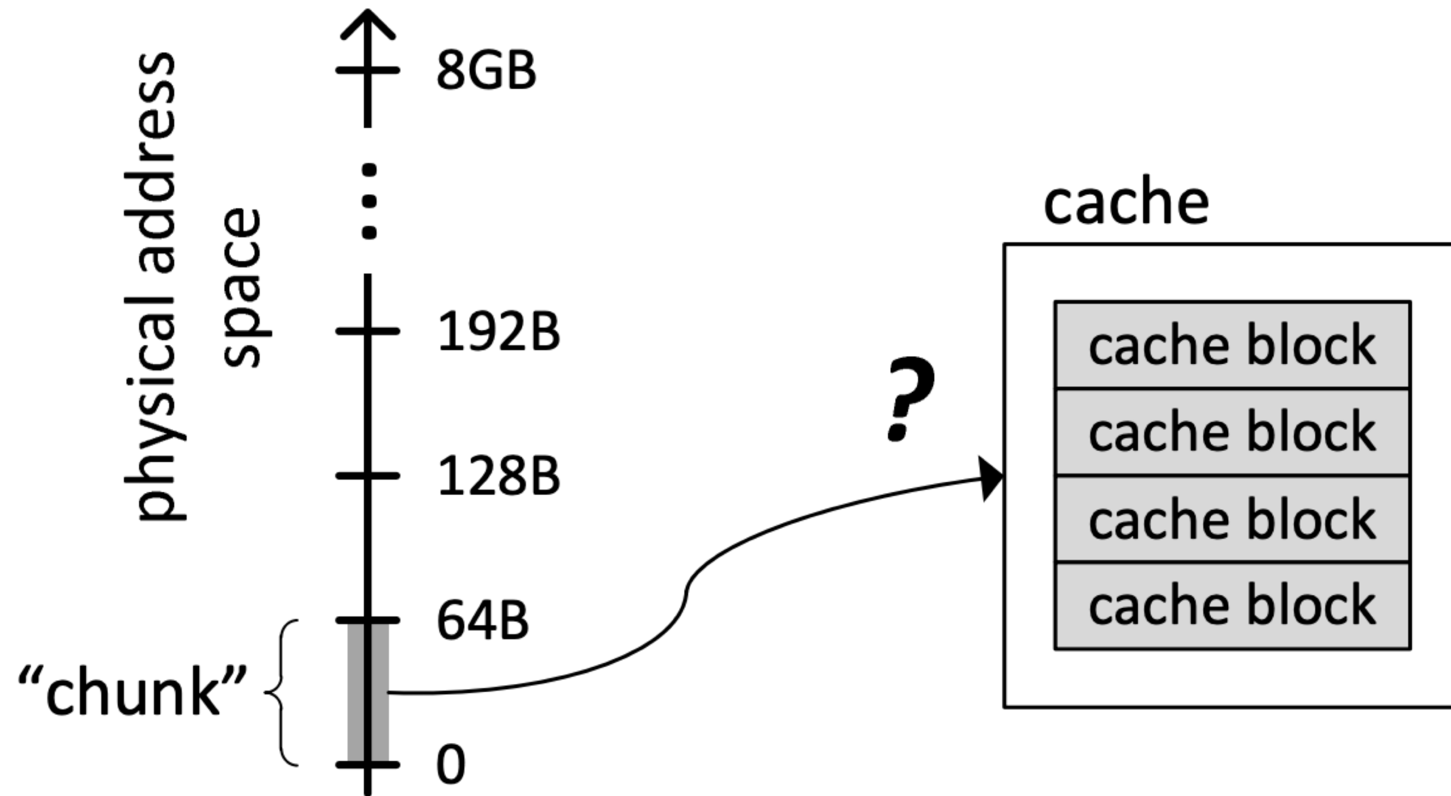
  = (hit-rate * hit-latency) + (miss-rate * miss-latency)
- Important Aside: *Is reducing AMAT always beneficial for performance?*

A Basic Hardware Cache Design

- We will start with a basic hardware cache design
- Then, we will examine a multitude of ideas to make it better (i.e., higher performance)

Blocks and Addressing the Cache

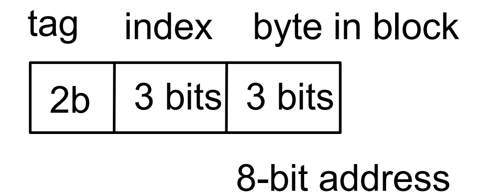
- Main memory is logically divided into fixed-size chunks (blocks)
- Cache can house only a limited number of blocks



Blocks and Addressing the Cache

- Main memory is logically divided into fixed-size chunks (**blocks**)
- **Cache** can house only a **limited** number of blocks

- Each **block address** maps to a potential location in the cache, determined by the **index bits** in the address
  - used to index into the tag and data stores

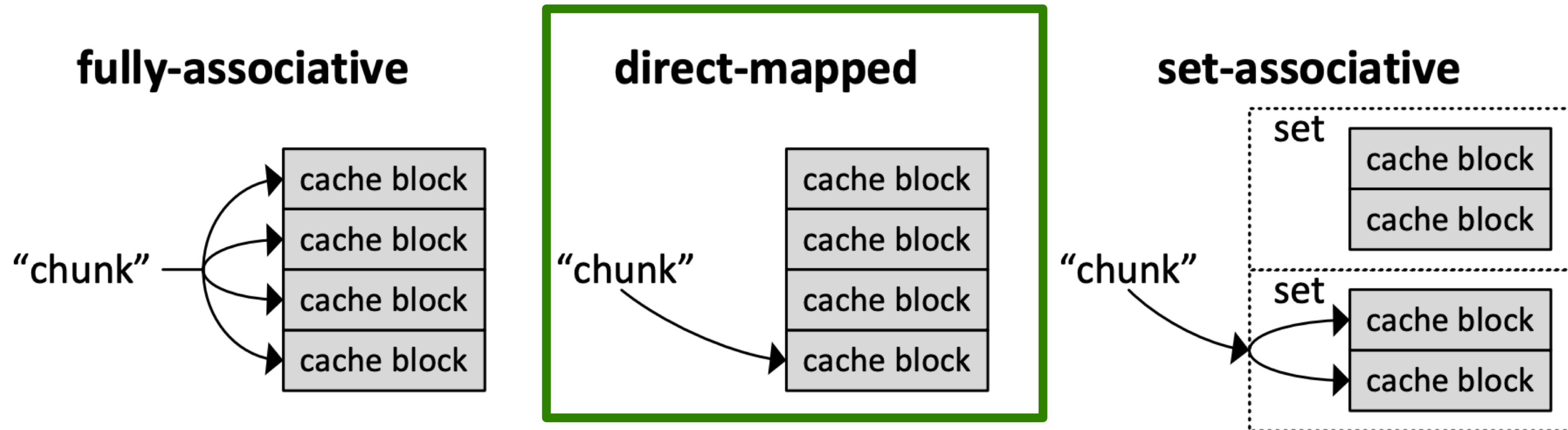


- Cache access:
  - 1) index into the tag and data stores with index bits in the address
  - 2) check the valid bit in the tag store
  - 3) compare tag bits in the address with the stored tag in the tag store

- If the stored tag is valid and matches the tag of the block, then the block is in the cache (cache hit)

Let’s See A Toy Example

- We will examine a direct-mapped cache first
- Direct-mapped: A given main memory block can be placed in **only one possible location** in the cache
- Toy example: 256-byte memory, 64-byte cache, 8-byte blocks

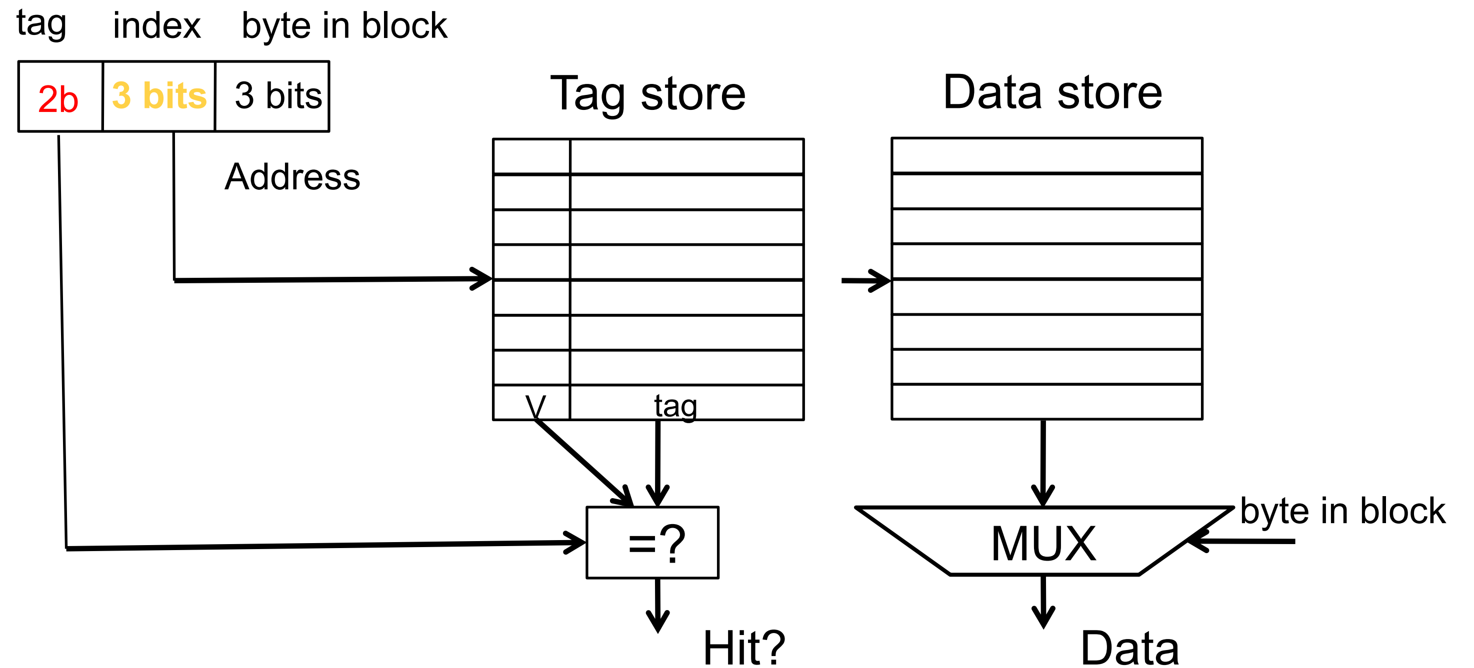


Direct-Mapped Cache: Placement and Access

Main memory
Block: 00000
Block: 00001
Block: 00010
Block: 00011
Block: 00100
Block: 00101
Block: 00110
Block: 00111
Block: 01000
Block: 01001
Block: 01010
Block: 01011
Block: 01100
Block: 01101
Block: 01110
Block: 01111
Block: 10000
Block: 10001
Block: 10010
Block: 10011
Block: 10100
Block: 10101
Block: 10110
Block: 10111
Block: 11000
Block: 11001
Block: 11010
Block: 11011
Block: 11100
Block: 11101
Block: 11110
Block: 11111

Main memory

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



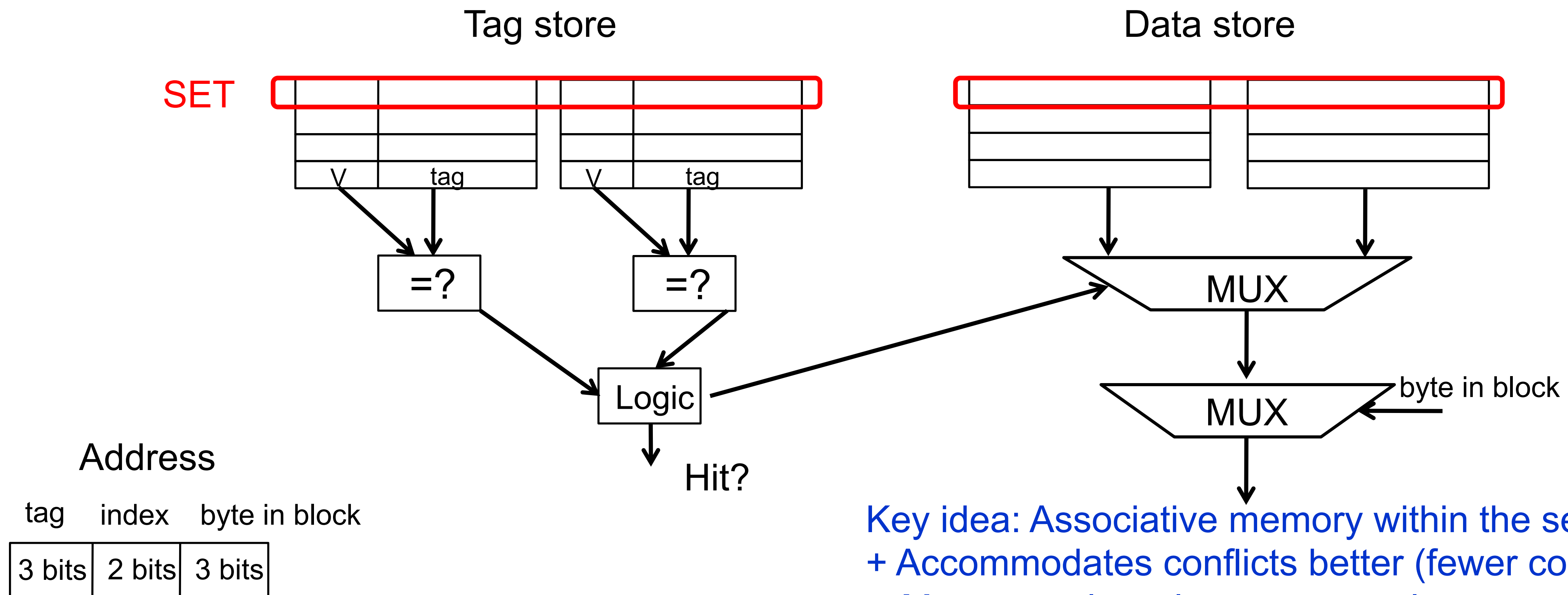
  - Blocks with the same index contend for the same cache location
    - Cause conflict misses when accessed consecutively

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 maps 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

- Problem: Addresses N and N+8 always conflict in a direct-mapped cache
- Idea: enable blocks with the same index to map to > 1 cache location
- Example: Instead of having one column of 8, have 2 columns of 4 blocks

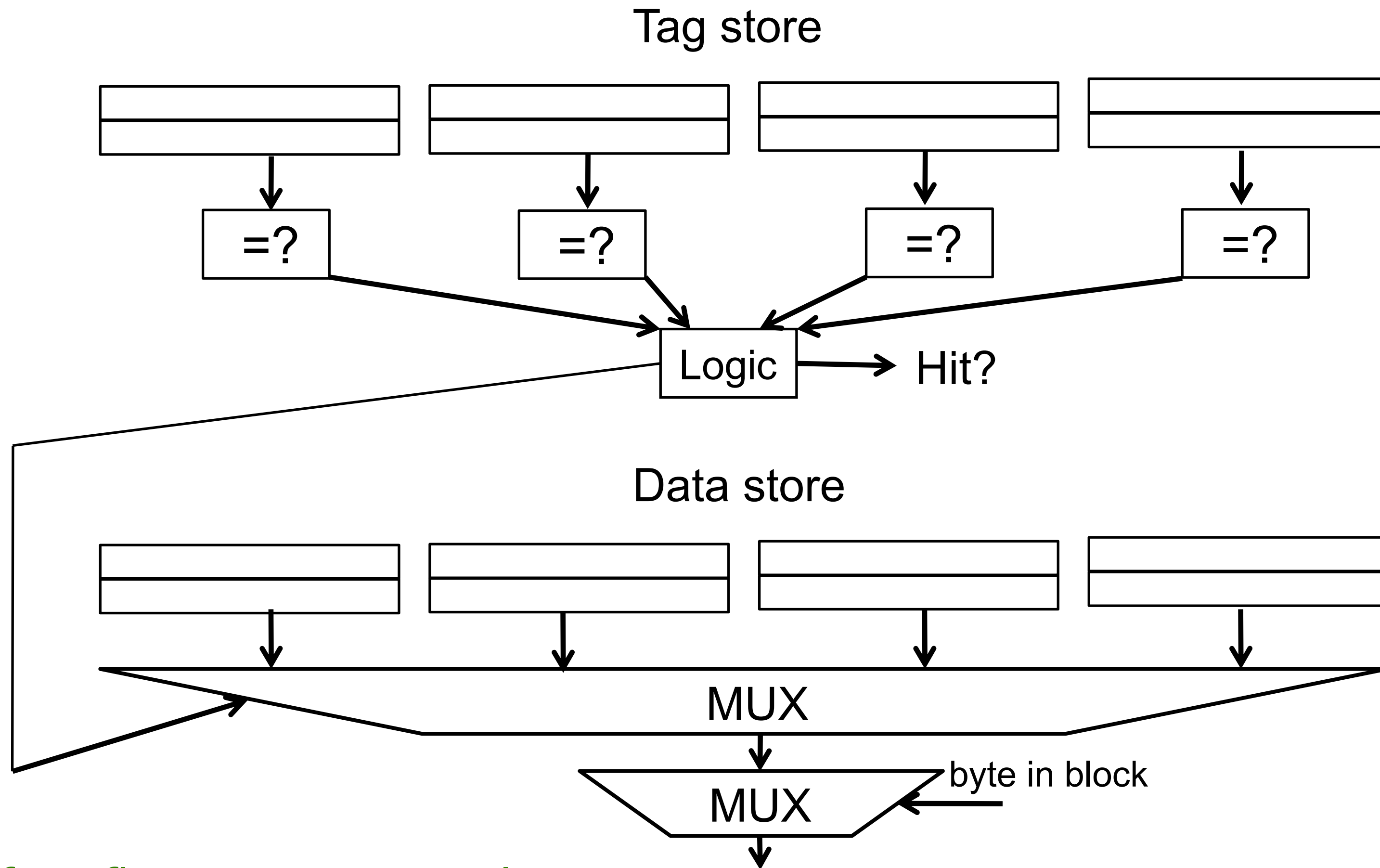


Key idea: Associative memory within the set
+ Accommodates conflicts better (fewer conflict misses)
-- More complex, slower access, larger tag store

2-way set associative cache: Blocks with the same index can map to 2 locations

Higher Associativity

- 4-way



Address

tag	index	byte in block
4 bits	1 b	3 bits

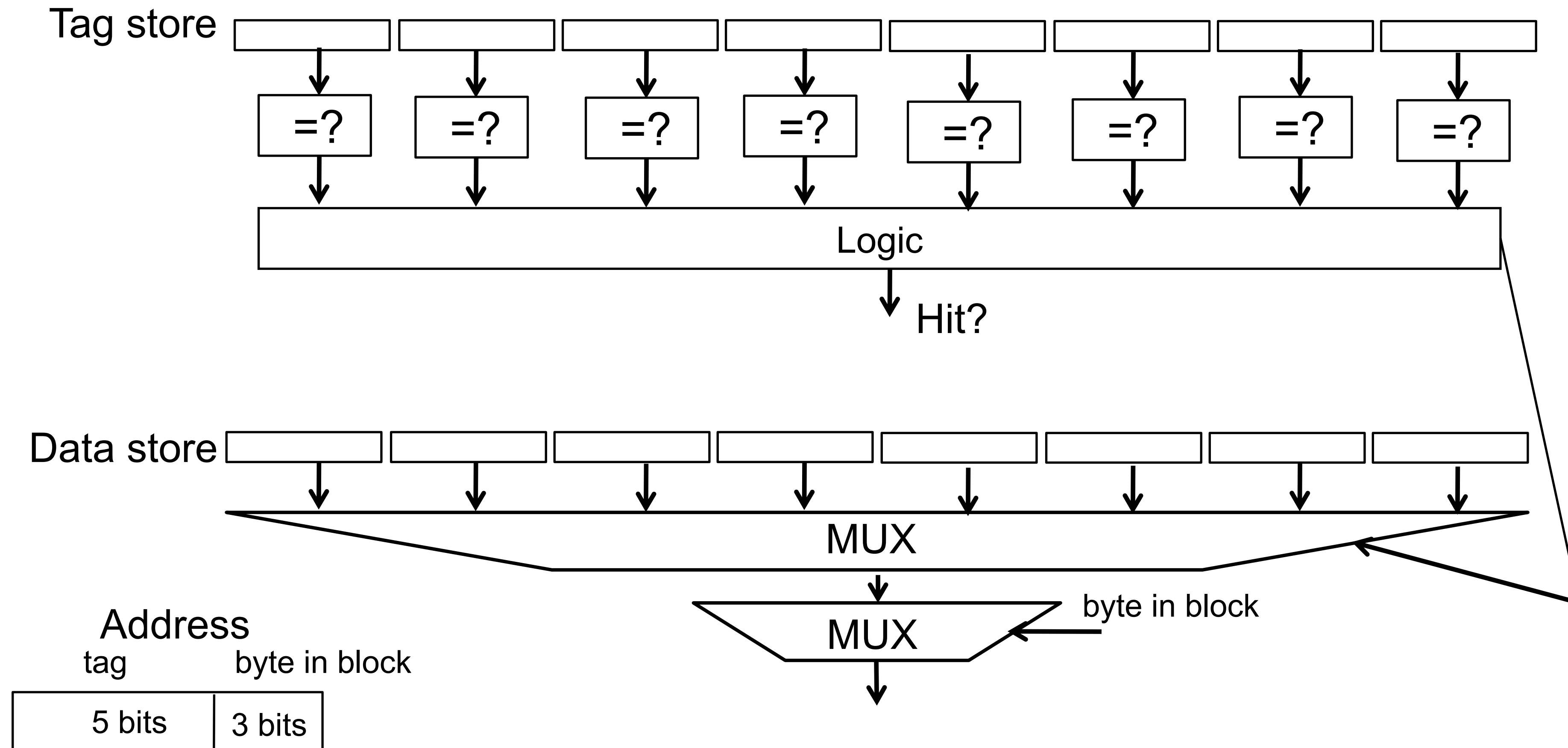
\+ Likelihood of conflict misses is even lower

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

4-way set associative cache: Blocks with the same index can map to 4 locations

Full Associativity

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



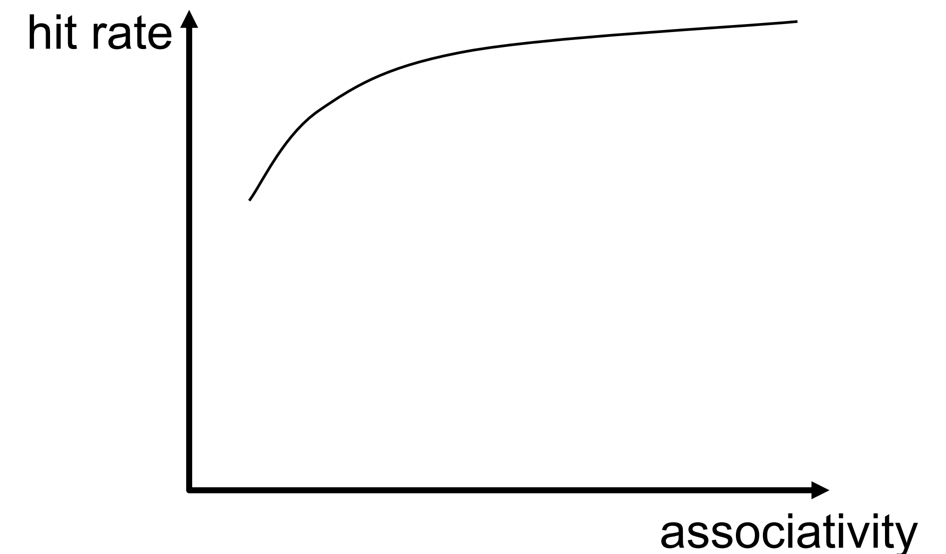
Fully associative cache: Any block can map to any location in the cache

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



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 a different cost?
    - Hybrid replacement policies
    - Optimal replacement policy?

Implementing LRU

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

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

- Question: 4-way set associative cache:
  - What do you minimally need to implement LRU perfectly?
  - How many different access orders are 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?

- Repeat for N-way set associative cache

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:
  - Not MRU (not most recently used)
  - Hierarchical LRU: divide the N-way set into M "groups", track the MRU group and the MRU way in each group
  - Victim-NextVictim Replacement: Only keep track of the victim and the next victim

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:
  - Performance of replacement policy 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: Where the miss is serviced from and miss overlapping
  - Qureshi et al. "A Case for MLP-Aware Cache Replacement," ISCA 2006.

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 cache

  \+ 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 cache

  \+ Simpler design

  \+ All levels are up to date & consistent → Simpler cache coherence: 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?
  - Allocate on write miss: Yes
  - No-allocate on write miss: No

- Allocate on write miss

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

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

  -- Requires transfer of the whole cache block

- No-allocate

  + Conserves cache space if locality of written blocks 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

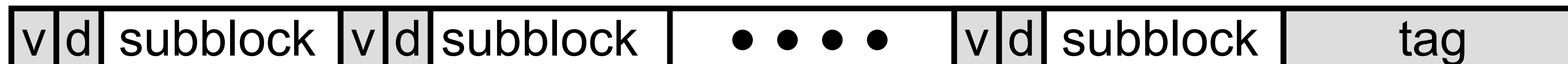
(A write simply validates and updates a subblock)

++ More freedom in transferring subblocks into the cache (a cache block doesn't need to be in cache fully)

(How many subblocks do you transfer on a read?)

-- More complex design; more valid and dirty bits

-- May not exploit spatial locality fully



Instruction vs. Data Caches

- Separate or Unified?

- Pros and Cons of Unified:

  + Dynamic sharing of cache space → **better overall cache utilization**: no overprovisioning that might happen with static partitioning of cache space (i.e., separate I and D caches)

  -- Instructions and data can evict/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?

- First level caches are almost always split
  - Mainly for the last reason above – pipeline constraints
- Outer level caches are almost always unified

Multi-level Cache Design & Management

- Cache level greatly affects cache design & management

- First-level caches (instruction and data)
  - Decisions very much affected by cycle time & pipeline structure
  - Small, lower associativity; latency is critical
  - Tag store and data store are usually 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 can be accessed serially

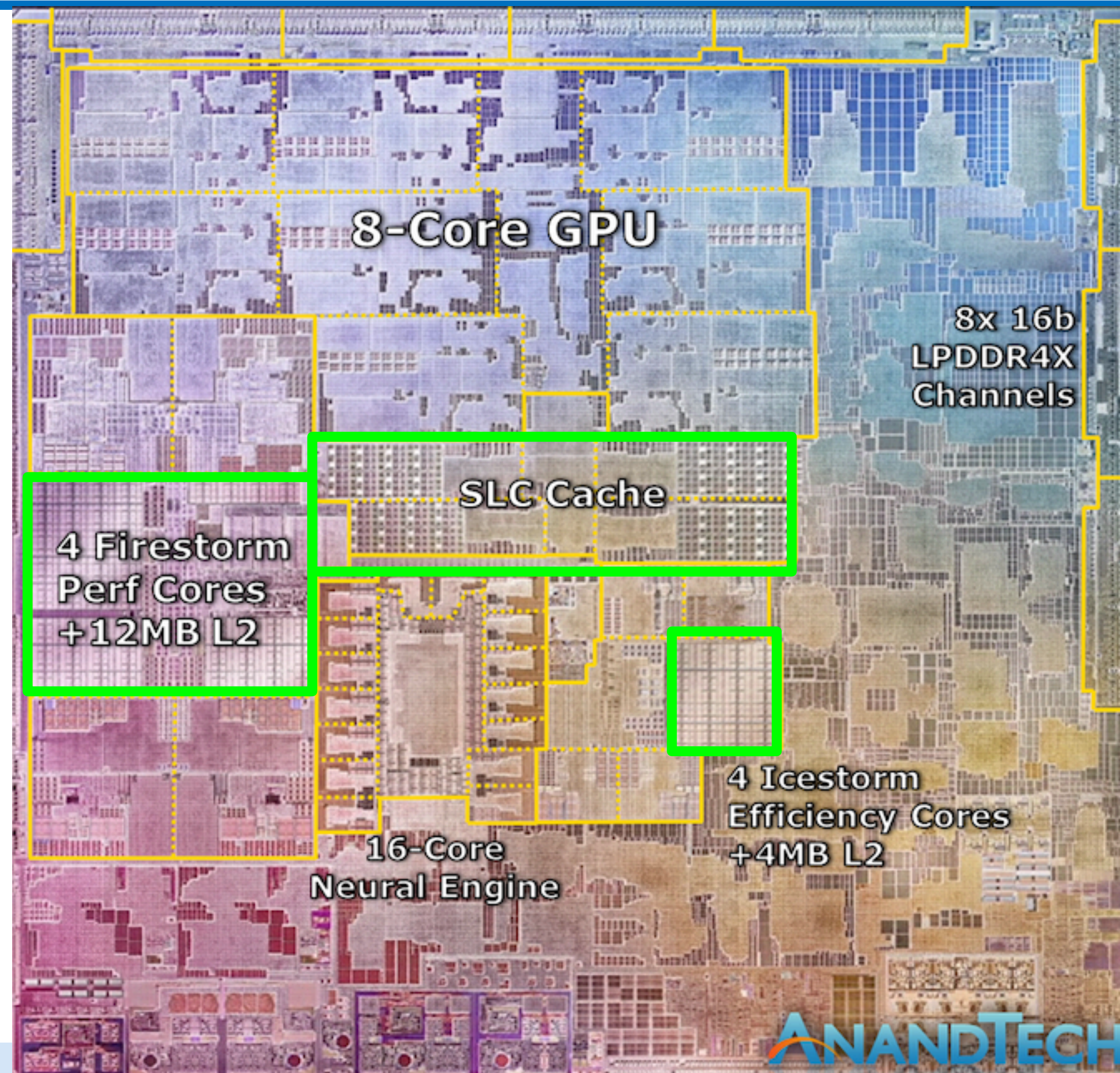
- Further-level (larger) caches
  - Access energy is a larger problem due to cache sizes
  - Tag store and data store are usually accessed serially

Serial vs. Parallel Access of Cache Levels

- Parallel: Next-level cache accessed in parallel with the previous level → a form of speculative access
  + Faster access to data if the previous level misses
  -- Unnecessary accesses to the next level if the previous level hits

- Serial: Next-level cache accessed only if previous-level misses
  -- Slower access to data if the previous level misses
  + No wasted accesses to the next level if the previous level hits
  - Next level does not see the same accesses as the previous level
    - The previous level acts as a filter (filters some temporal & spatial locality)
    - **Management policies are different across cache levels**

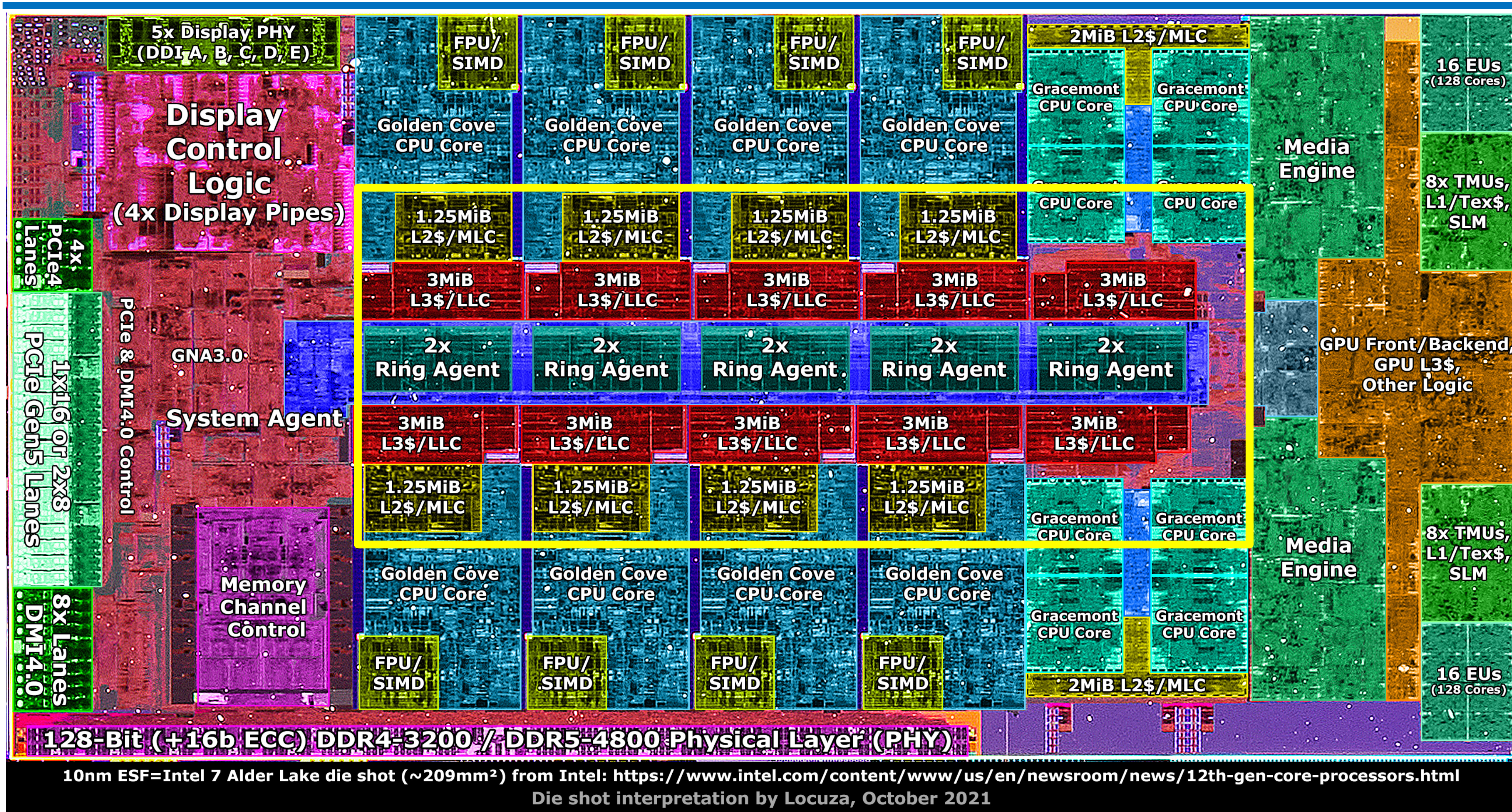
Deeper and Larger Cache Hierarchies



Apple M1,
2021

Source: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested

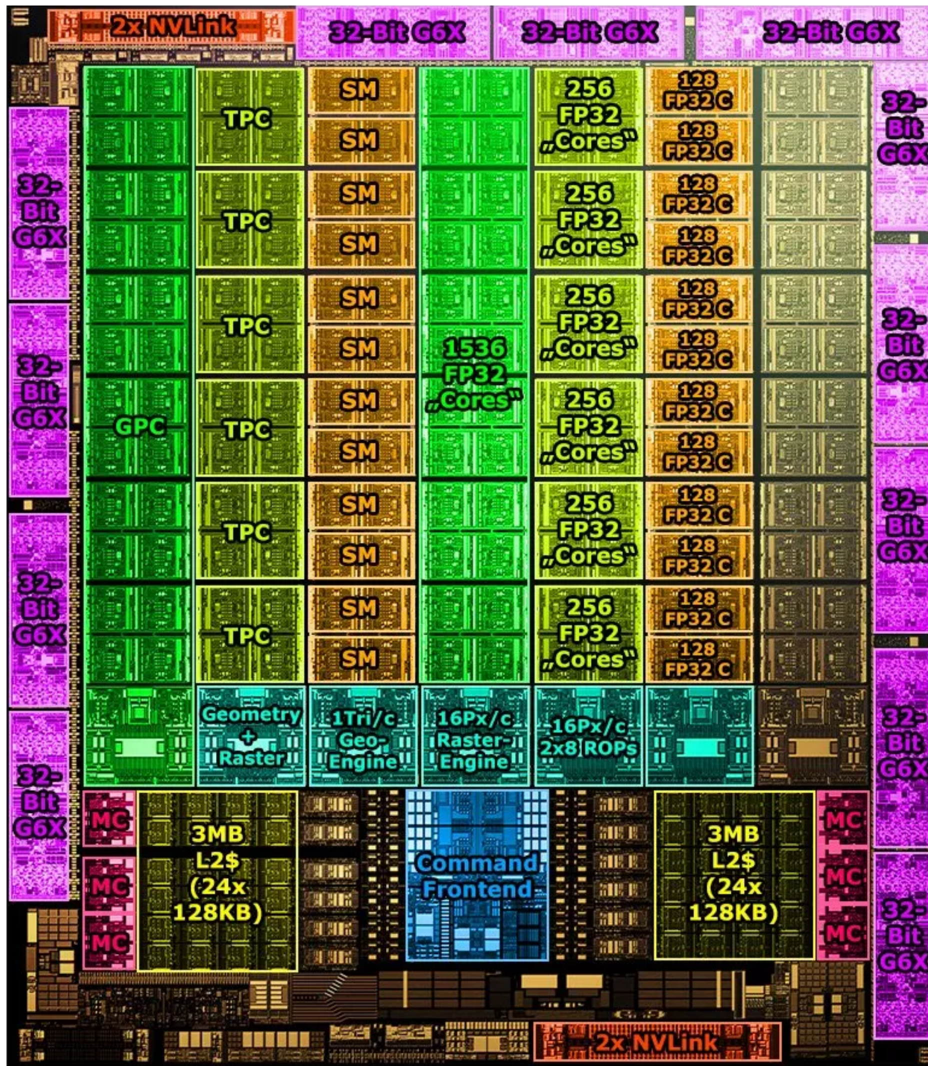
Deeper and Larger Cache Hierarchies



Intel Alder Lake, 2021

Source: https://twitter.com/Locuza_/status/1454152714930331652

Deeper and Larger Cache Hierarchies



Cores:

128 Streaming Multiprocessors

L1 Cache or Scratchpad:

192KB per SM

Can be used as L1 Cache and/or Scratchpad

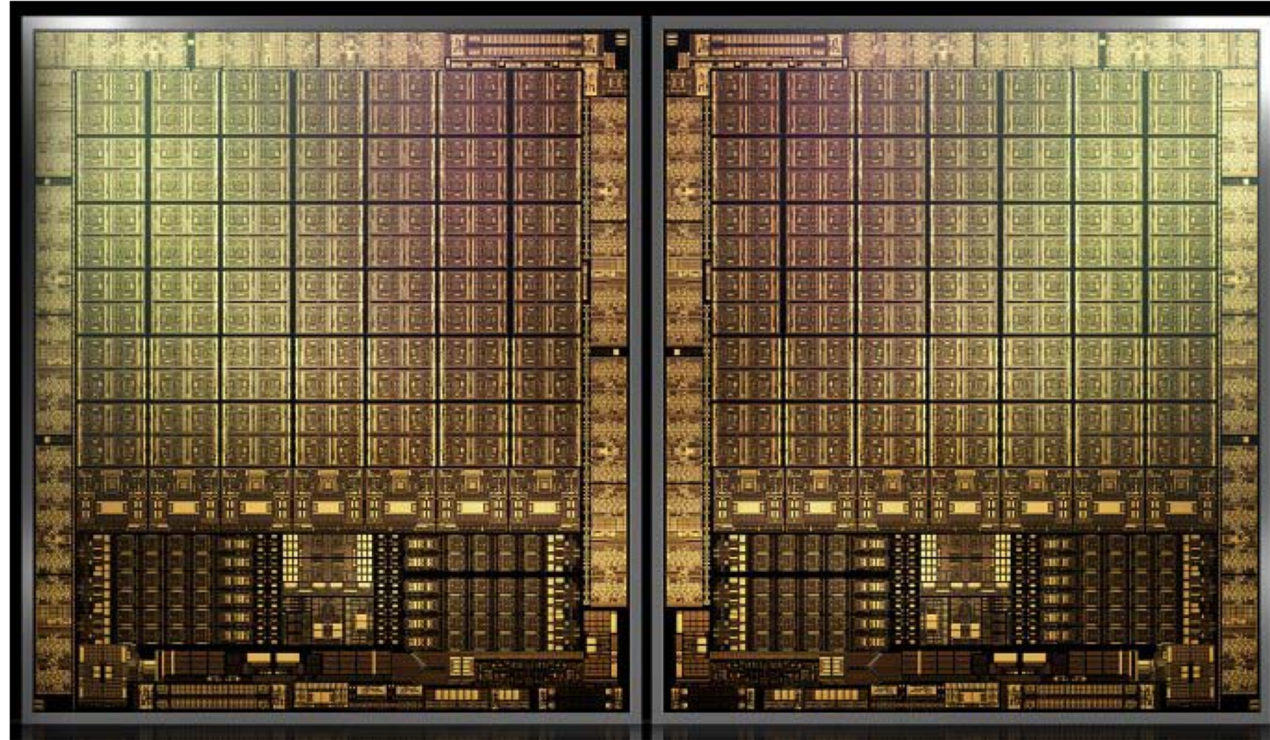
L2 Cache:

40 MB shared

Nvidia Ampere, 2020

https://www.tomshardware.com/news/infrared-photographer-photos-nvidia-ga102-ampere-silicon

Deeper and Larger Cache Hierarchies



Nvidia Hopper, 2022

Cores:
144 Streaming Multiprocessors

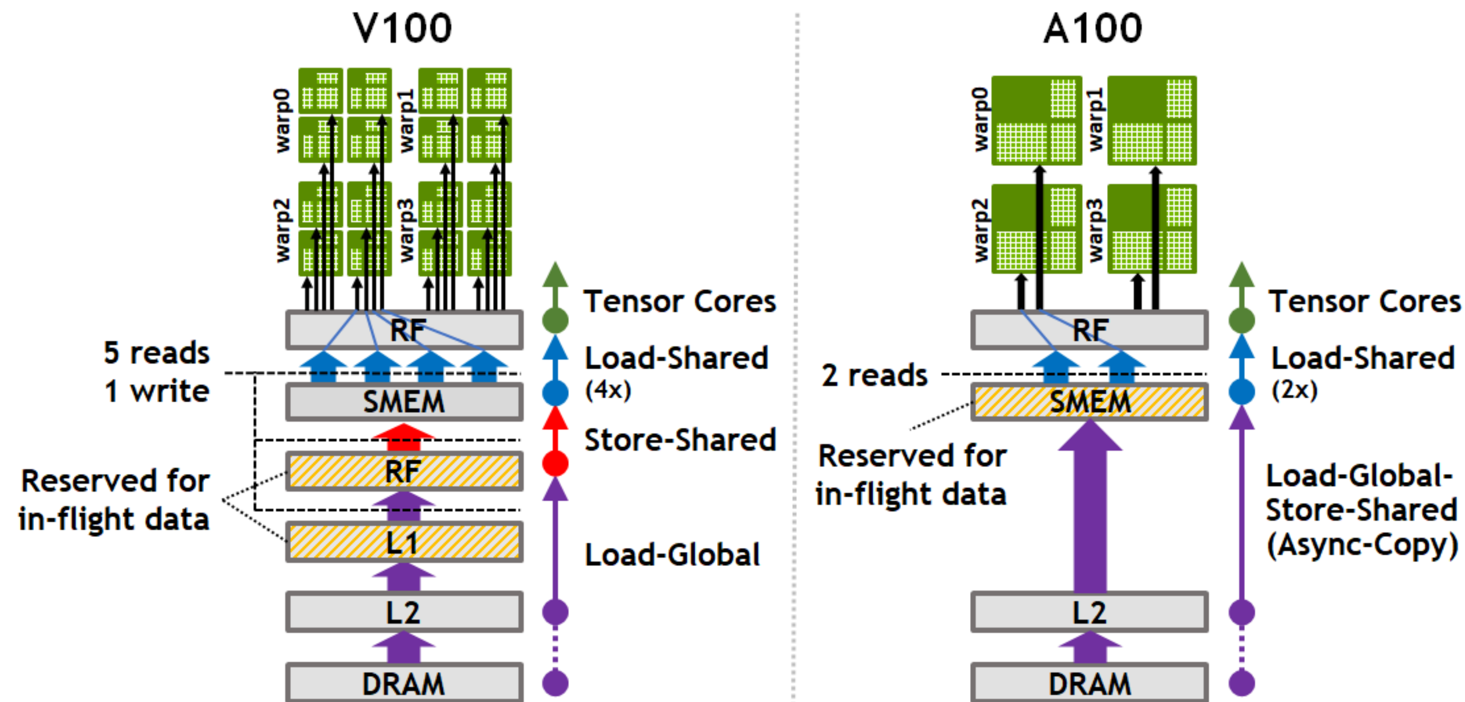
L1 Cache or Scratchpad:
256KB per SM
Can be used as L1 Cache and/or Scratchpad

L2 Cache:
60 MB shared

https://wccftech.com/nvidia-hopper-gpus-featuring-mcm-technology-tape-out-soon-rumor/

NVIDIA V100 & A100 Memory Hierarchy

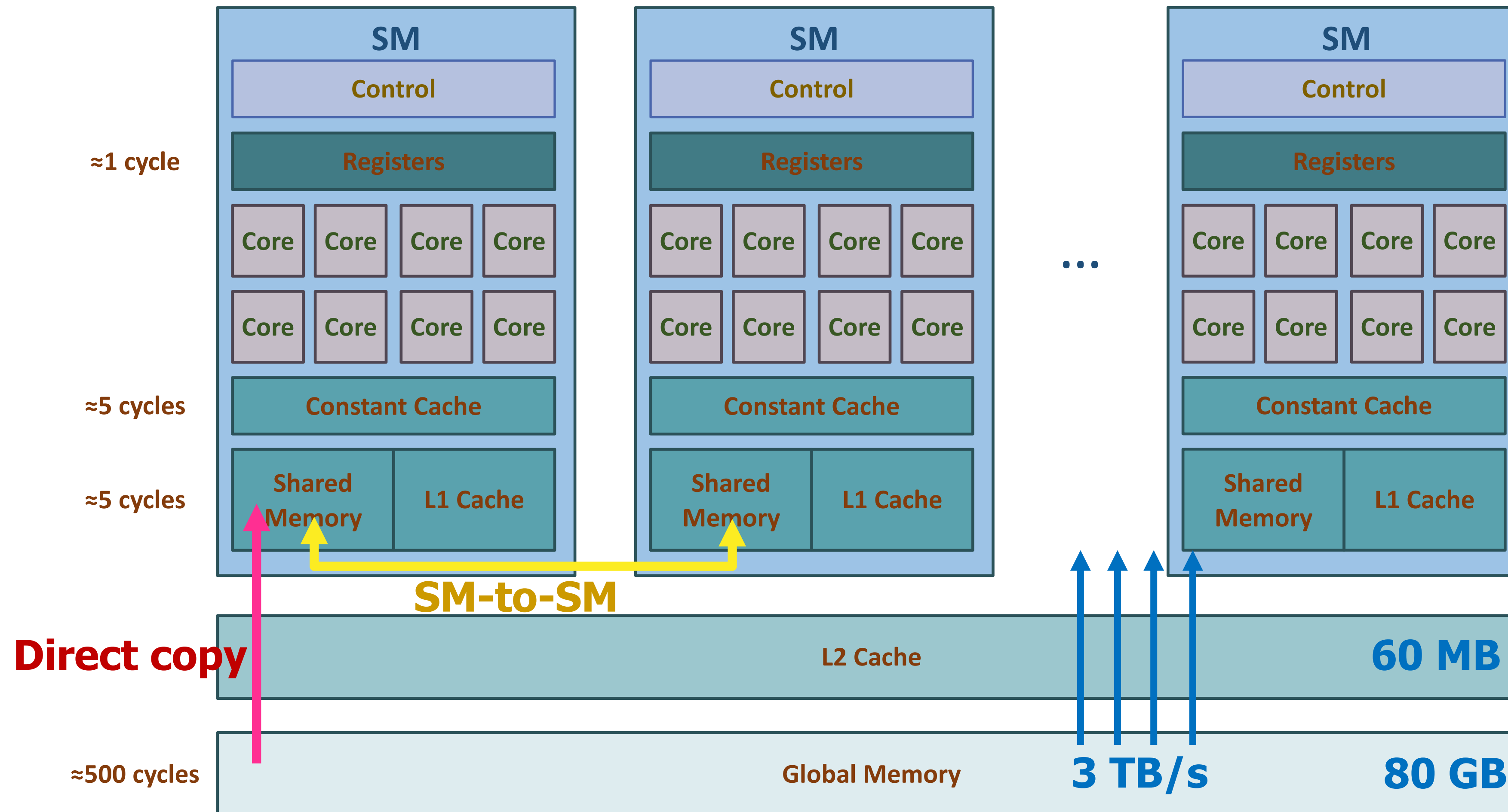
- Example of data movement between GPU global memory (DRAM) and GPU cores.



A100 improves SM bandwidth efficiency with a new load-global-store-shared asynchronous copy instruction that bypasses L1 cache and register file (RF). Additionally, A100's more efficient Tensor Cores reduce shared memory (SMEM) loads.

A100 feature:
Direct copy from L2 to scratchpad, bypassing L1 and register file.

Memory in the NVIDIA H100 GPU



Multi-Level Cache Design Decisions

- Which level(s) to place a block into (from memory)?
- Which level(s) to evict a block to (from an inner level)?
- Bypassing vs. non-bypassing levels
- Inclusive, exclusive, non-inclusive hierarchies
  - Inclusive: a block in an inner level is always included also in an outer level → simplifies cache coherence
  - Exclusive: a block in an inner level does not exist in an outer level → better utilizes space in the entire hierarchy
  - Non-inclusive: a block in an inner level may or may not be included in an outer level → relaxes design decisions

Hardware Design

Cache Performance

L1 cache
~10s of KB, ~nsec

L2 cache
100s of KB ~ few MB, many nsec

L3 cache,
many MBs, even more nsec

Cache Parameters vs. Miss/Hit Rate

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

Cache Size

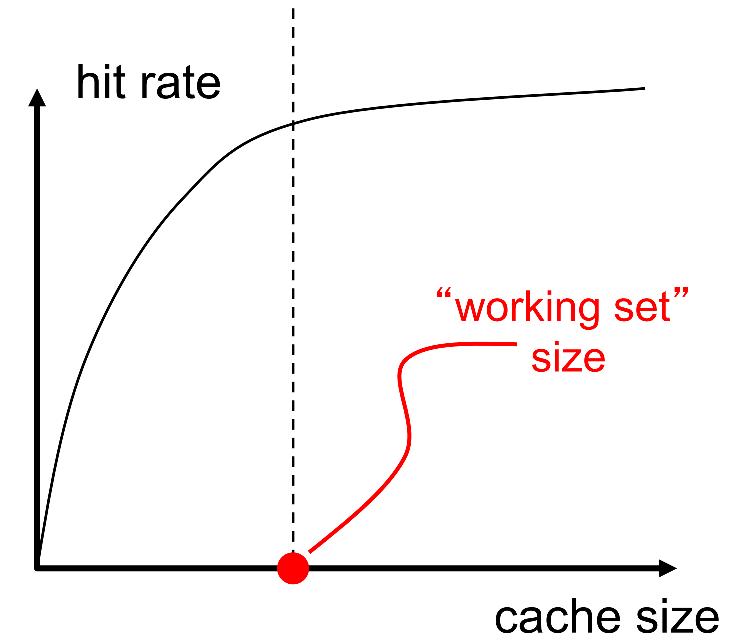
- Cache size: total data capacity (not including tag store)
  - bigger cache can exploit temporal locality better

- Too large a cache adversely affects hit and miss latency
  - bigger is slower

- Too small a cache
  - does not exploit temporal locality well
  - useful data is often replaced

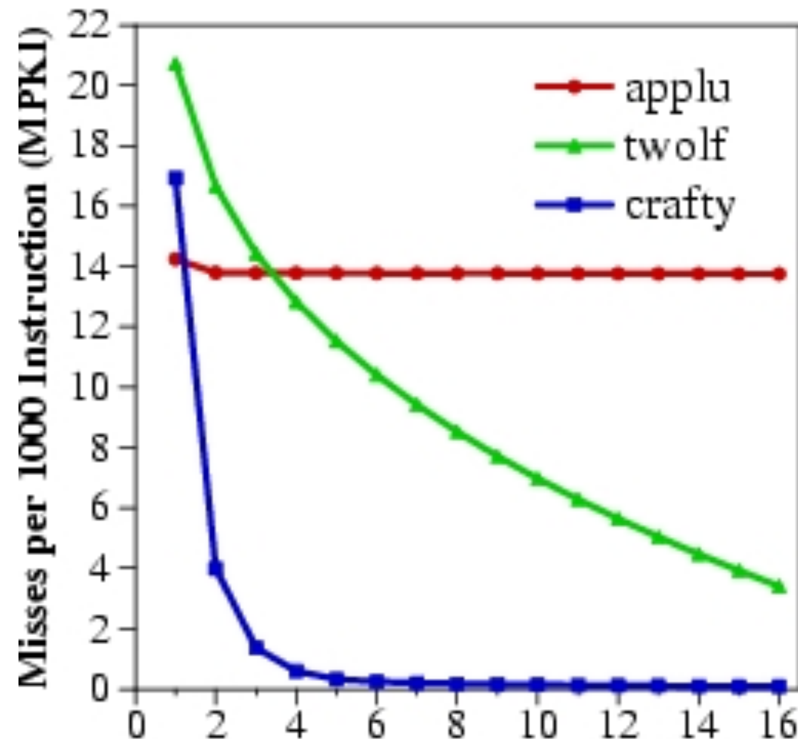
- Working set: the entire set of data

the executing application references
  - Within a time interval



Benefits of Larger Caches Widely Varies

- Benefits of cache size widely varies across applications



Number of ways from 16-way 1MB L2

Low Cache Utility

High Cache Utility

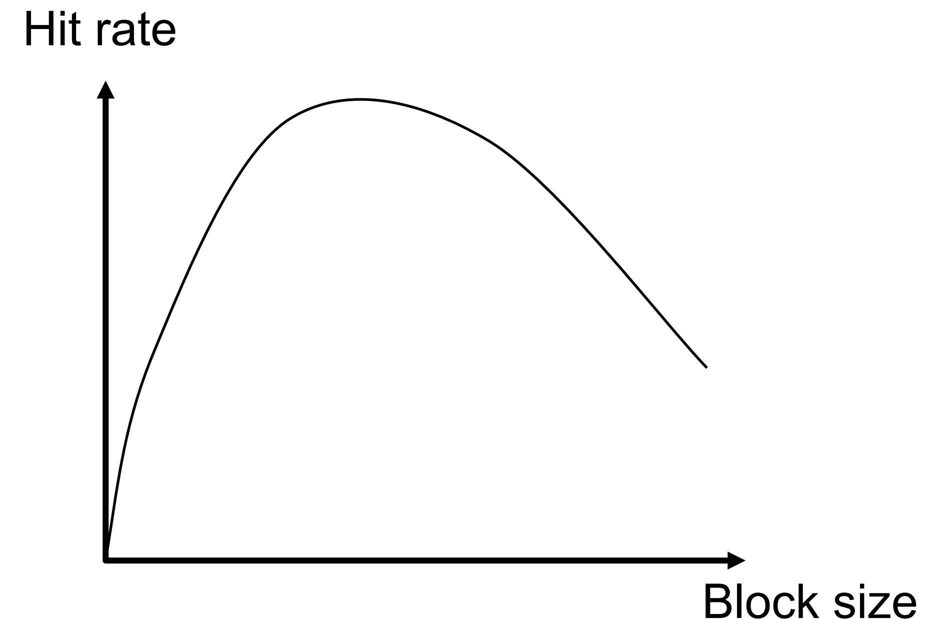
Saturating Cache Utility

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
  - do not exploit spatial locality well
  - have larger tag overhead

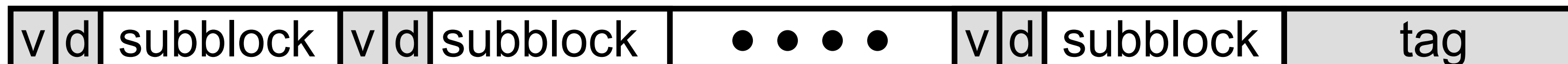
- Too large blocks
  - too few total blocks → do not exploit temporal locality well
  - waste cache space and bandwidth/energy if spatial locality is not high



Large Blocks: Critical-Word and Subblocking

- Large cache blocks can take a long time to fill into the cache
  - Idea: Fill cache block critical-word first
  - Supply the critical data to the processor immediately

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

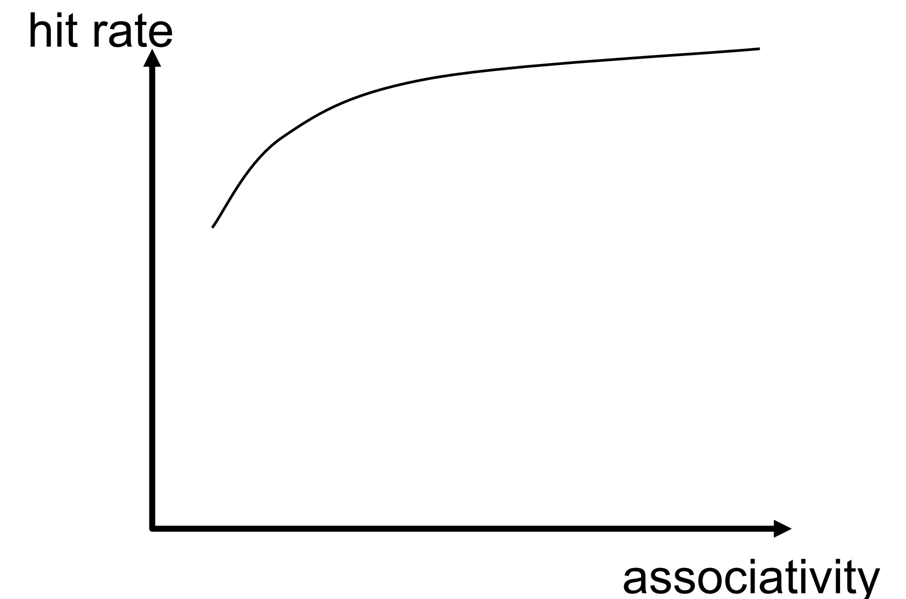


Associativity

- 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

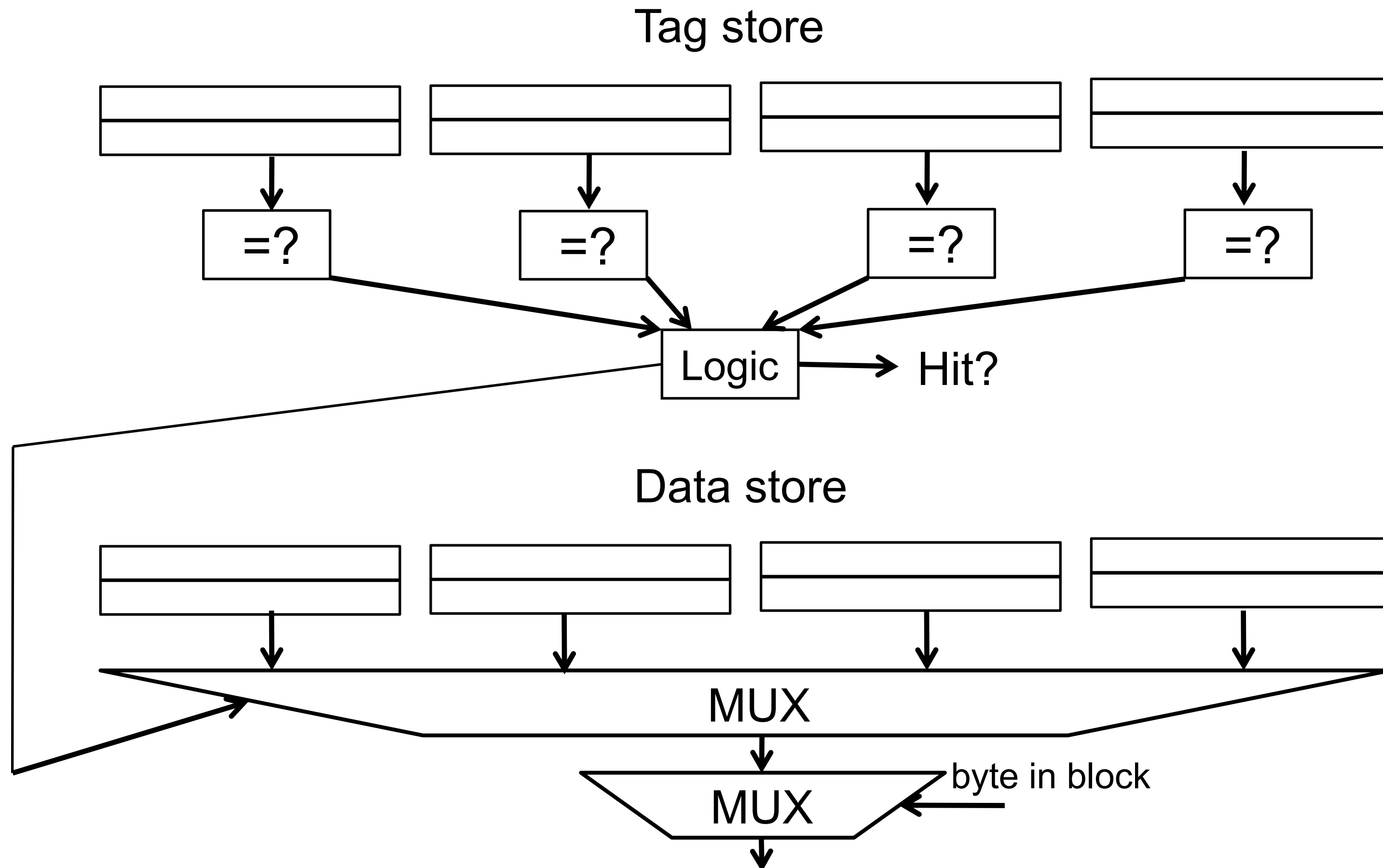
- Smaller associativity
  - lower cost
  - lower hit latency
    - Especially important for L1 caches



- Is power of 2 associativity required?

Recall: Higher Associativity

- 4-way



Address

tag	index	byte in block
4 bits	1 b	3 bits

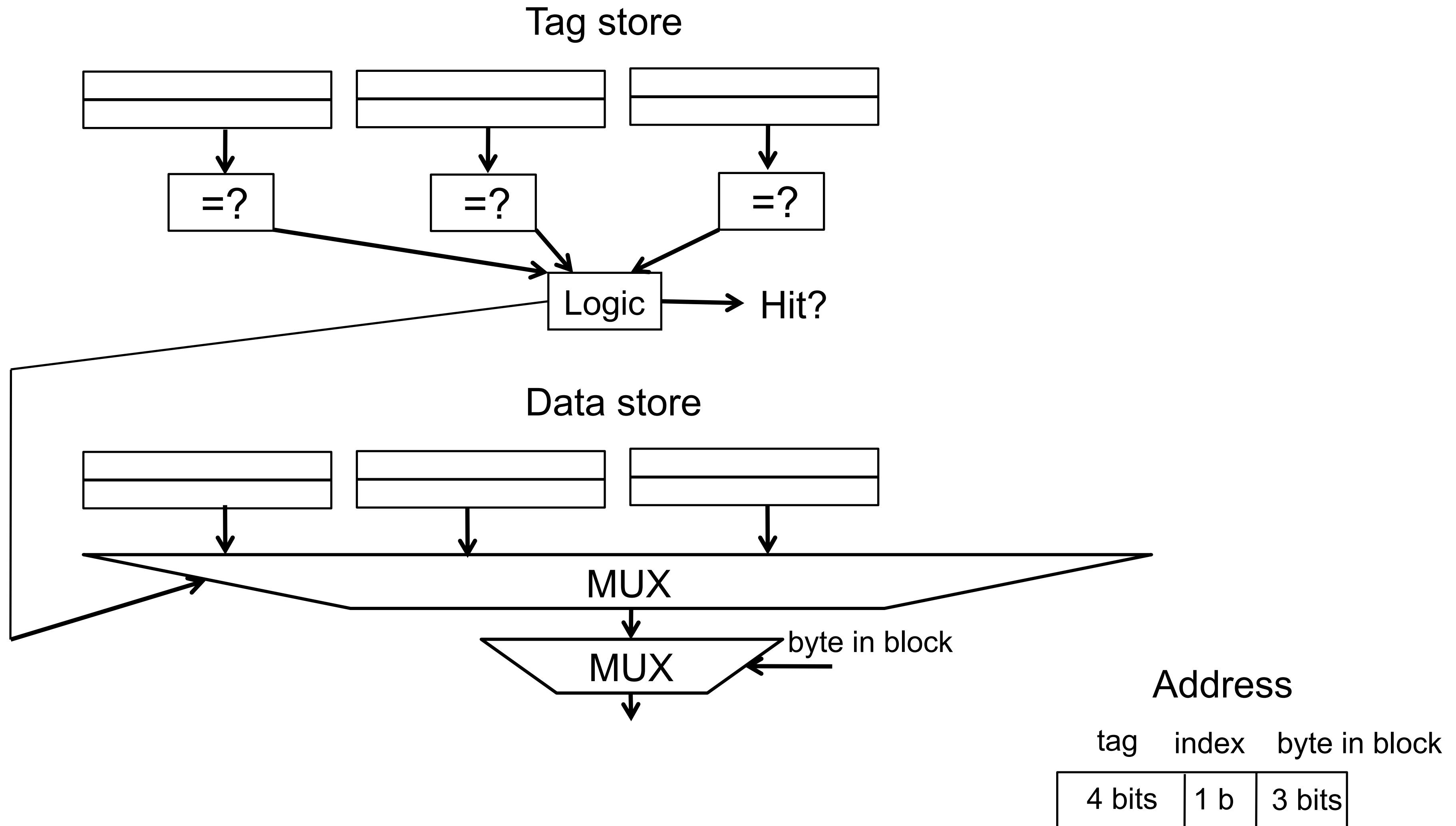
+ Likelihood of conflict misses is even lower

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

4-way set associative cache: Blocks with the same index can map to 4 locations

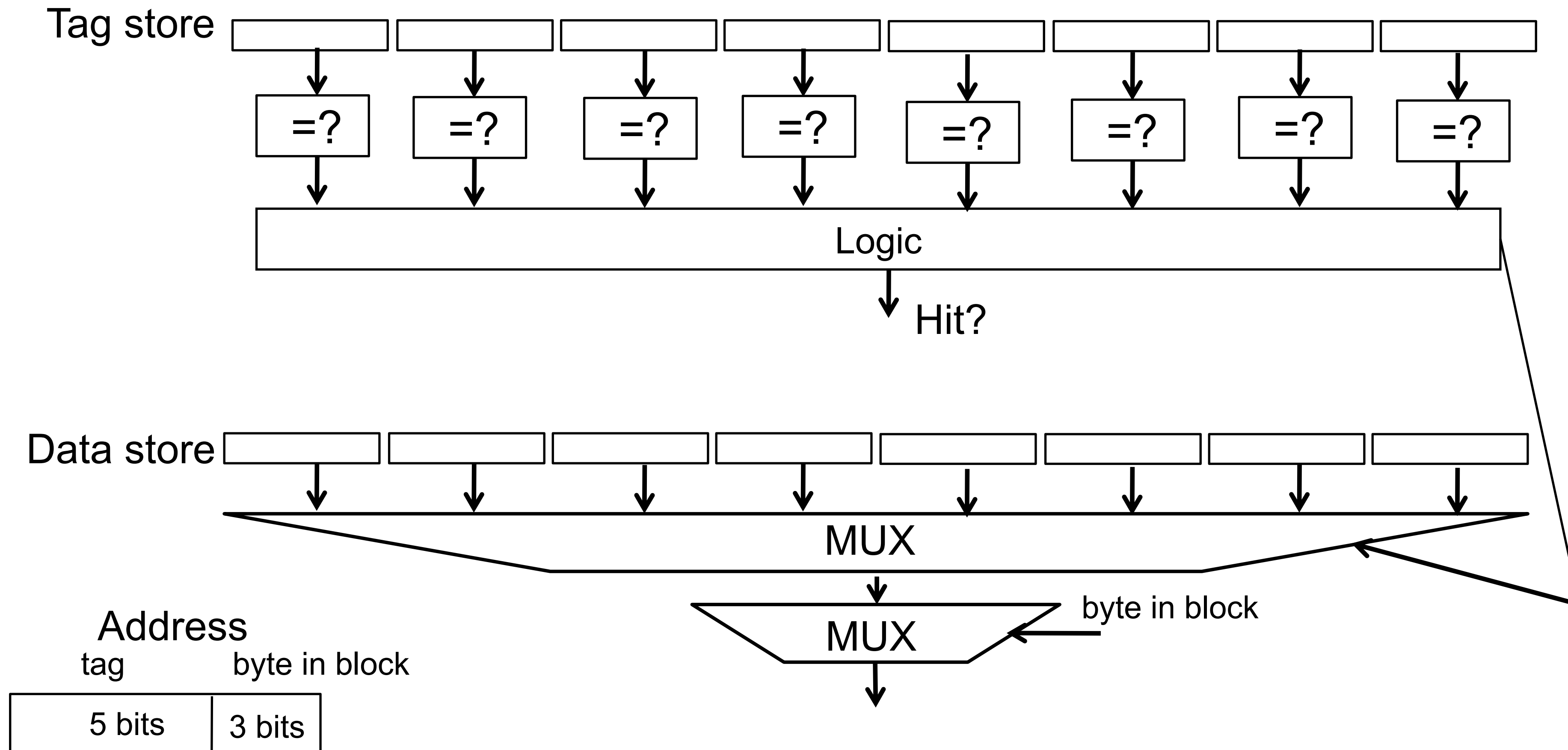
Higher Associativity (3-way)

- 3-way



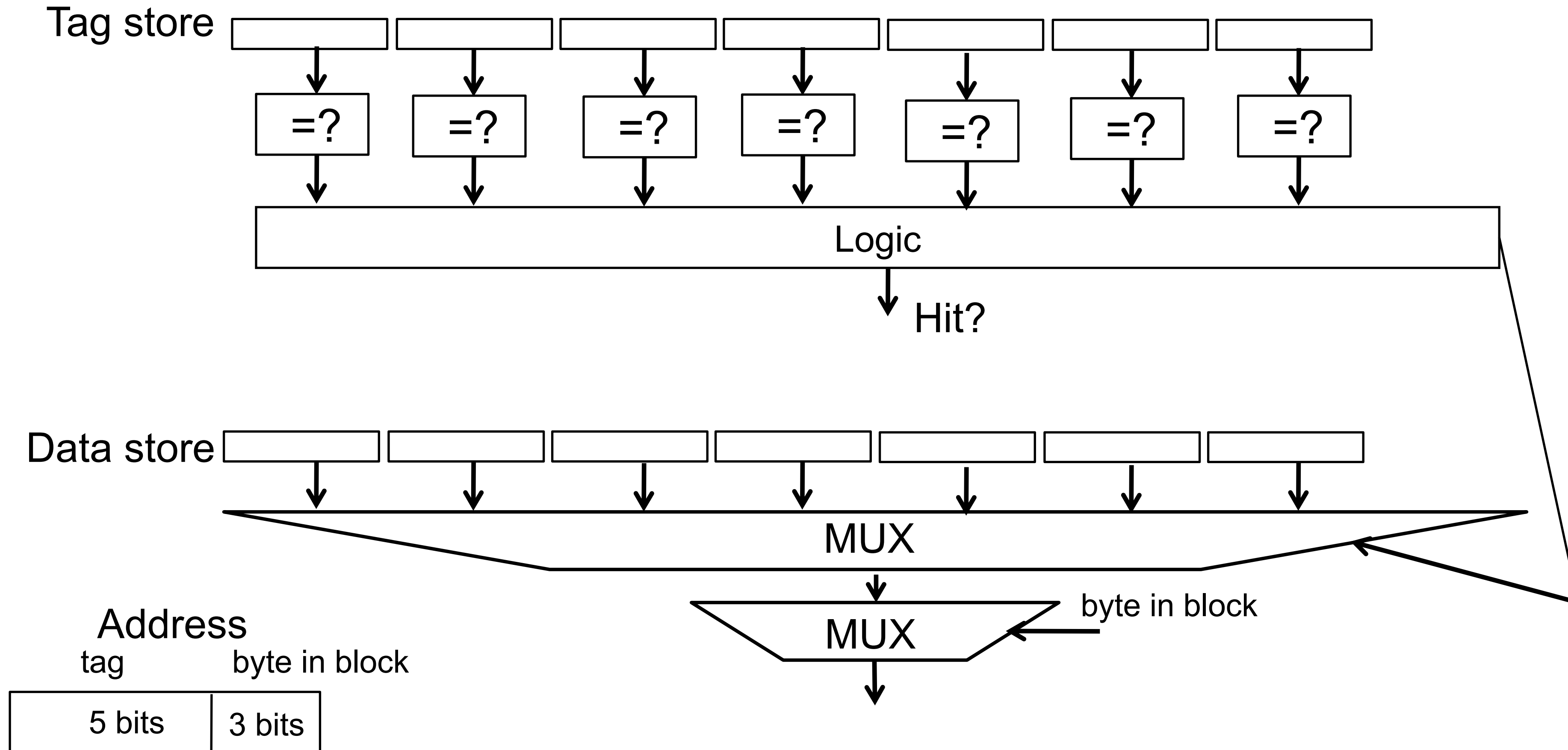
Recall: 8-way Full Associativity Cache

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



Fully associative cache: Any block can map to any location in the cache

7-way Fully Associative Cache



Classification of Cache Misses

- Compulsory miss
  - first reference to an address (block) always results in a miss
  - subsequent references to the block should hit in cache unless the block is displaced from cache for the reasons below

- Capacity miss
  - cache is too small to hold all needed data
  - 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 (only accessed data) cannot help; larger blocks can
  - Prefetching helps: 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 into the cache
    - Software hints for eviction/replacement/promotion
- 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

Software Approaches for Higher Hit Rate

- Restructuring data access patterns
- Restructuring data layout

- Loop interchange
- Data structure separation/merging
- Blocking
- …

Restructuring Data Access Patterns (I)

- Idea: Restructure data layout or data access patterns
- Example: If column-major
  - x[i+1,j] follows x[i,j] in memory
  - x[i,j+1] is far away from x[i,j]

Poor code

```
for i = 1, rows
    for j = 1, columns
        sum = sum + x[i,j]
```

Better code

```
for j = 1, columns
    for i = 1, rows
        sum = sum + x[i,j]
```

- This is called a loop interchange
- Other optimizations can also increase hit rate
  - Loop fusion, array merging, …

Restructuring Data Access Patterns (II)

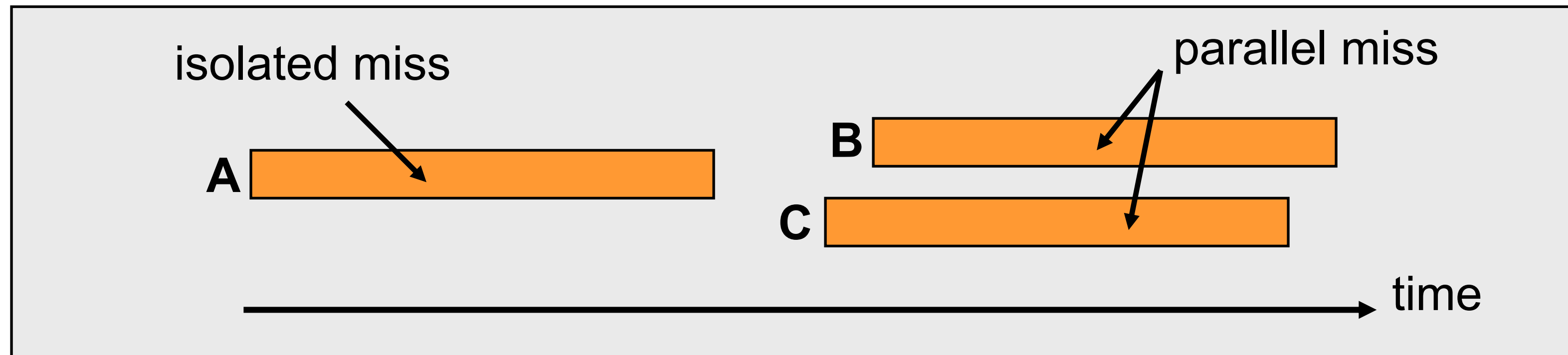
- Blocking
  - Divide loops operating on arrays into computation chunks so that each chunk can hold its data in the cache
  - Avoids cache conflicts between different chunks of computation
  - Essentially: Divide the working set so that each piece fits in the cache

- Also called Tiling

Miss Latency/Cost

- What is miss latency or miss cost affected by?
  - Where does the miss get serviced from?
    - What level of cache in the hierarchy?
    - Row hit versus row conflict in DRAM (bank/rank/channel conflict)
    - Queueing delays in the memory controller and the interconnect
    - Local vs. remote memory (chip, node, rack, remote server, …)
    - …
  - How much does the miss stall the processor?
    - Is it overlapped with other latencies?
    - Is the data immediately needed by the processor?
    - Is the incoming block going to evict a longer-to-refetch block?
    - …

Memory Level Parallelism (MLP)



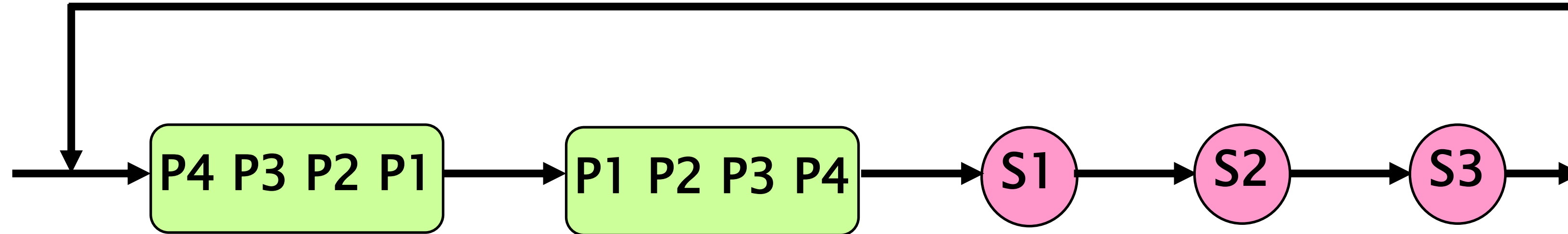
- Memory Level Parallelism (MLP) is the notion of generating and servicing multiple memory accesses in parallel [Glew'98]
- Several techniques improve MLP (e.g., out-of-order execution)
- MLP varies. Some misses are isolated and some parallel

How does this affect cache replacement?

Traditional Cache Replacement Policies

- Traditional cache management policies try to reduce the miss count
- Implicit assumption: Reducing miss count reduces memory-related stall time
- Misses with varying cost/MLP break this assumption!
- Eliminating an isolated miss helps performance more than eliminating a parallel miss
- Eliminating a higher-latency miss could help performance more than eliminating a lower-latency miss

An Example



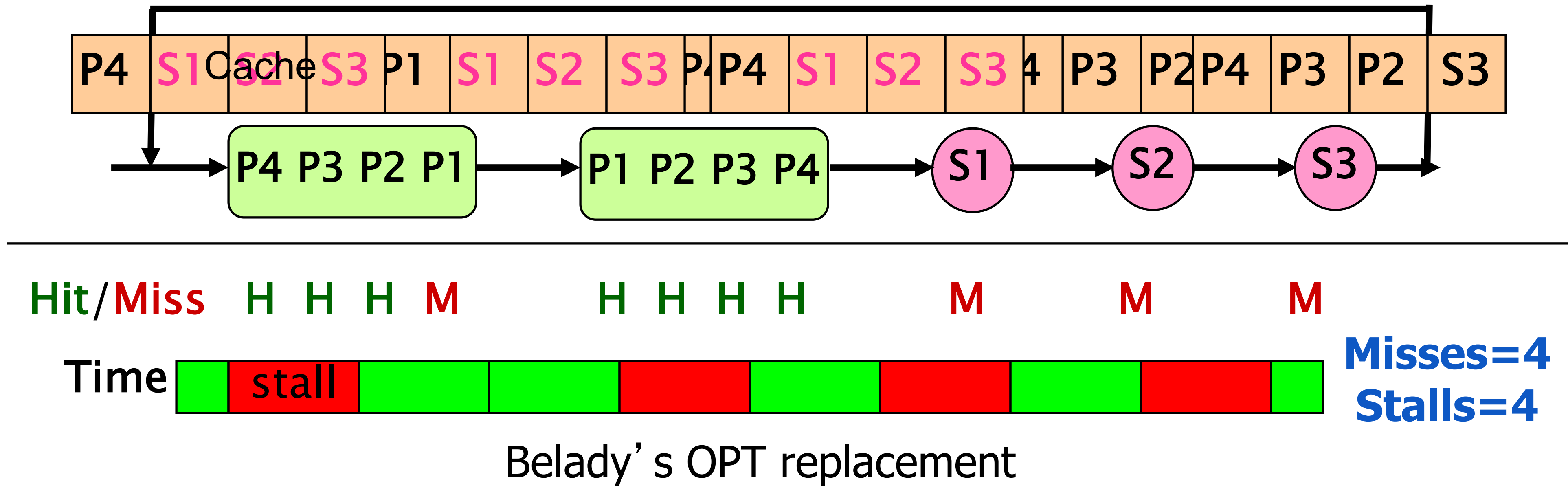
Misses to blocks P1, P2, P3, P4 can be serviced in parallel
Misses to blocks S1, S2, and S3 are isolated (i.e., serviced serially)

Two replacement algorithms:

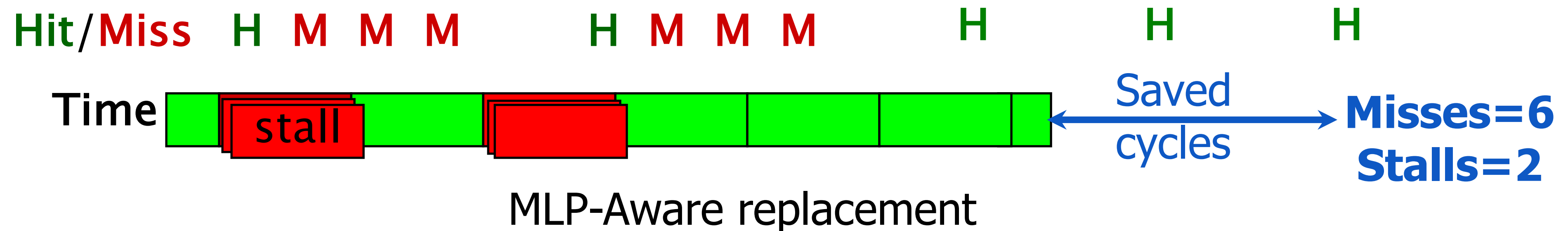
1. Minimizes miss count (Belady's OPT)
2. Reduces isolated miss (MLP-Aware)

For a fully associative cache containing 4 blocks

Fewest Misses ≠ Best Performance



Belady's OPT replacement



MLP-Aware replacement

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
  - ...

KING'S College LONDON

Hardware Design

Lecture 8: Cache

Dr. Haiyu Mao

19.03.2026