Digital Design & Computer Arch.

Lecture 23: Memory Hierarchy and Caches

Prof. Onur Mutlu

ETH Zürich

Spring 2021

27 May 2021

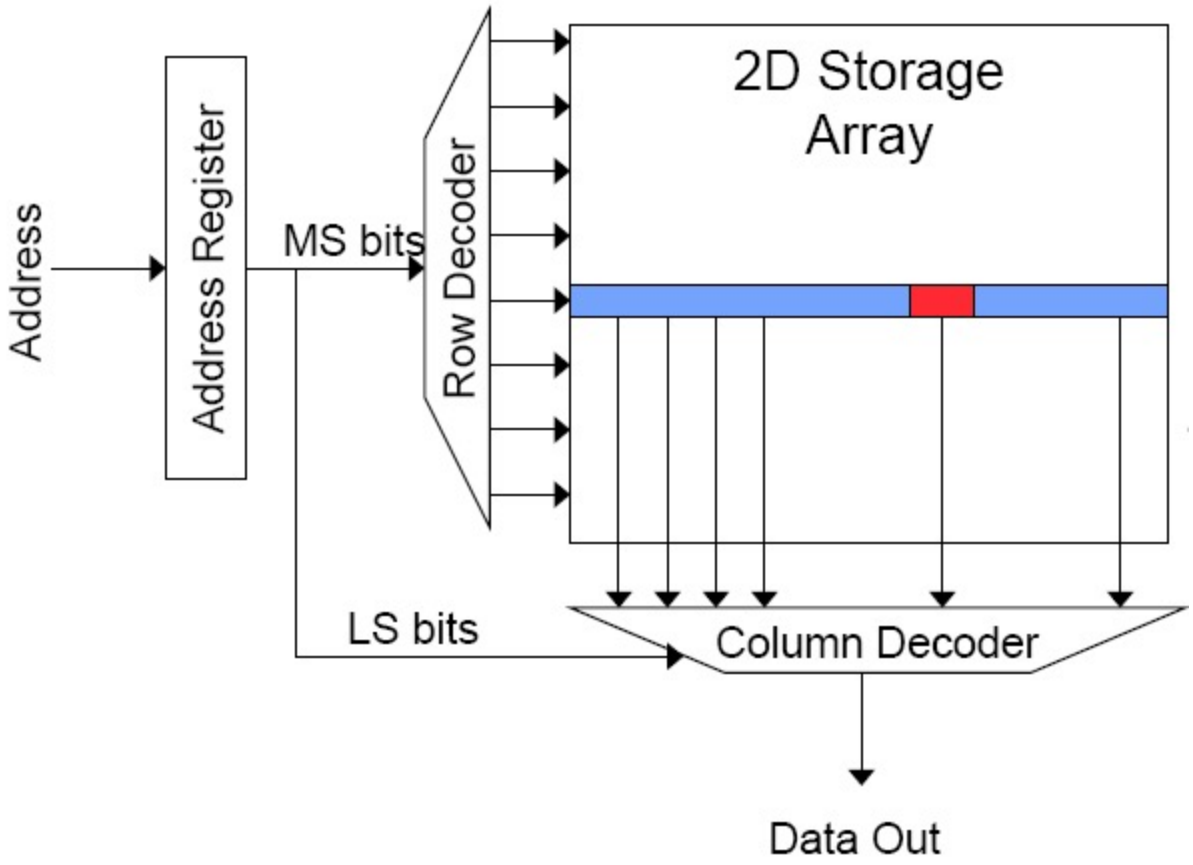
Readings for This Lecture and Next

- Memory Hierarchy and Caches

- Required
  - H&H Chapters 8.1-8.3
  - Refresh: P&P Chapter 3.5
  - Kim & Mutlu, "Memory Systems," Computing Handbook, 2014.
    - https://people.inf.ethz.ch/omutlu/pub/memory-systems-introduction_computing-handbook14.pdf

- Recommended
  - An early cache paper by Maurice Wilkes
    - Wilkes, "Slave Memories and Dynamic Storage Allocation," IEEE Trans. On Electronic Computers, 1965.

Recall: Memory Bank Organization and Operation



- Read access sequence:

1. Decode row address & drive word-lines

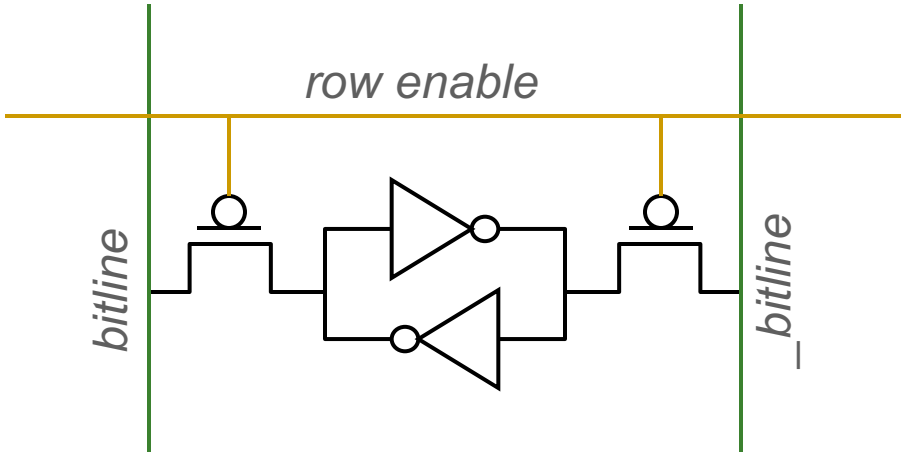
2. Selected bits drive bit-lines
   - Entire row read

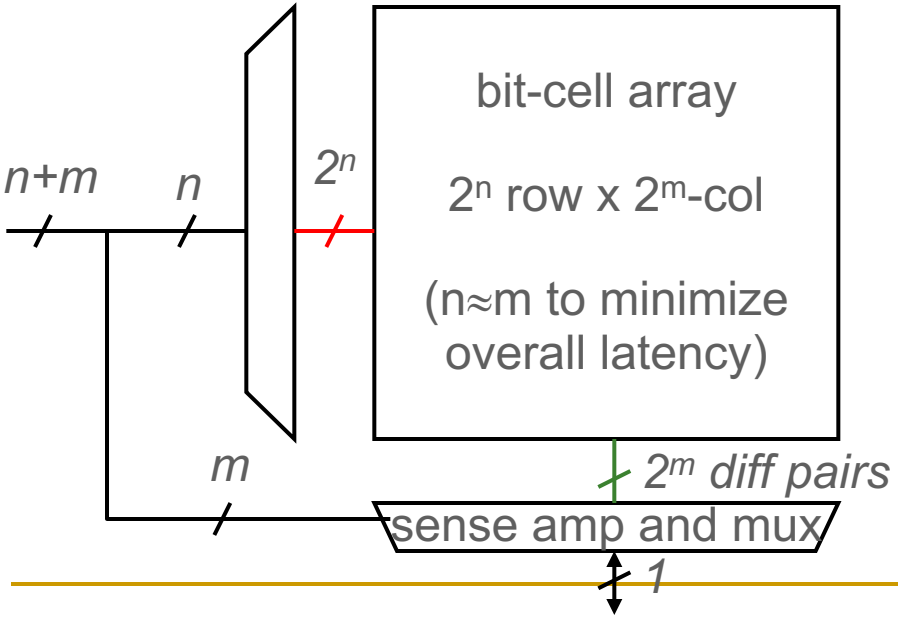
3. Amplify row data

4. Decode column address & select subset of row
   - Send to output

5. Precharge bit-lines
   - For next access

Recall: SRAM





Read Sequence

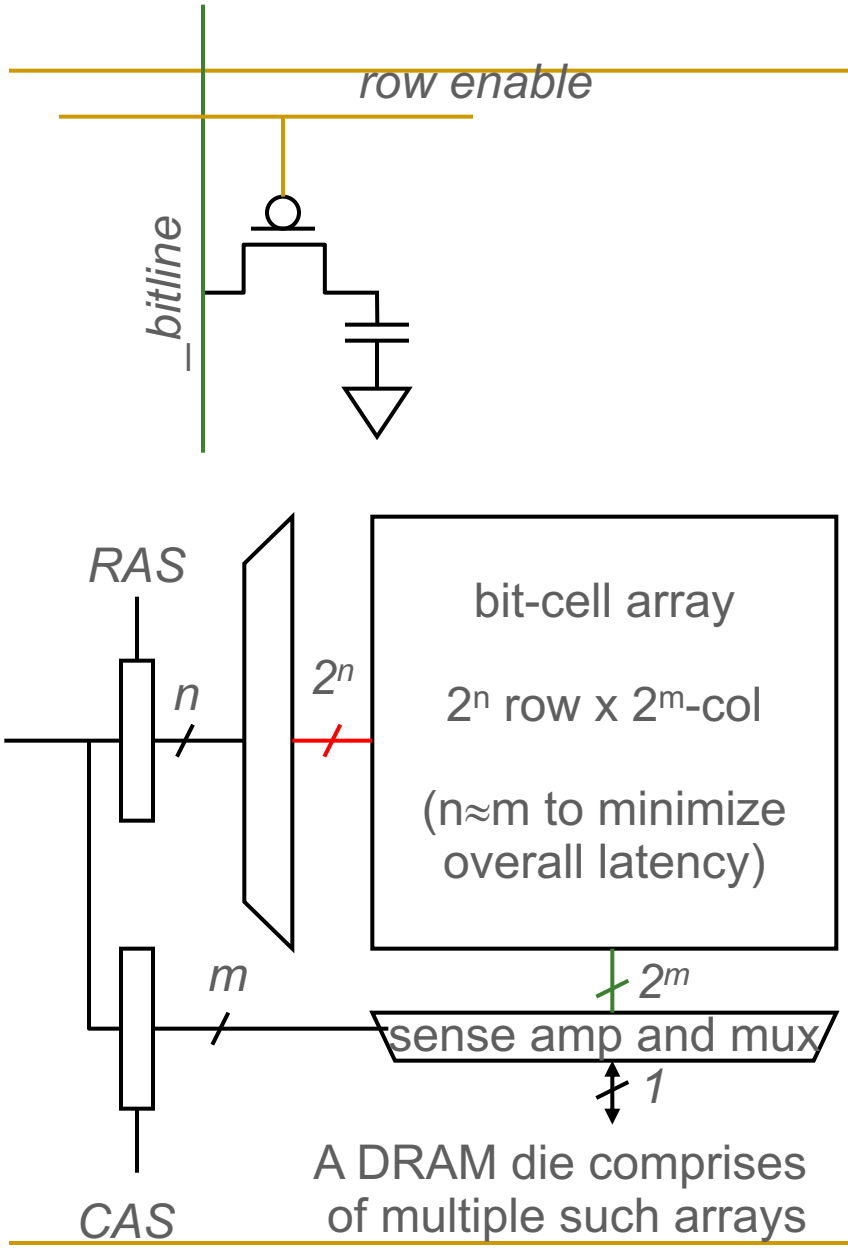
1. address decode
2. drive row select
3. selected bit-cells drive bitlines
   (entire row is read together)
4. differential sensing and column select
   (data is ready)
5. precharge all bitlines
   (for next read or write)

Access latency dominated by steps 2 and 3

Cycling time dominated by steps 2, 3 and 5

- step 2 proportional to 2^m
- step 3 and 5 proportional to 2^n

Recall: DRAM



Bit stored as charge on node capacitor (non-restorative)

- bit cell loses charge when read
- bit cell loses charge over time

Read Sequence

1~3 same as SRAM

4. a "flip-flopping" sense amp amplifies and regenerates the bitline, data bit is mux'ed out
5. precharge all bitlines

Destructive reads

Charge loss over time

Refresh: A DRAM controller must periodically read each row within the allowed refresh time (10s of ms) such that charge is restored

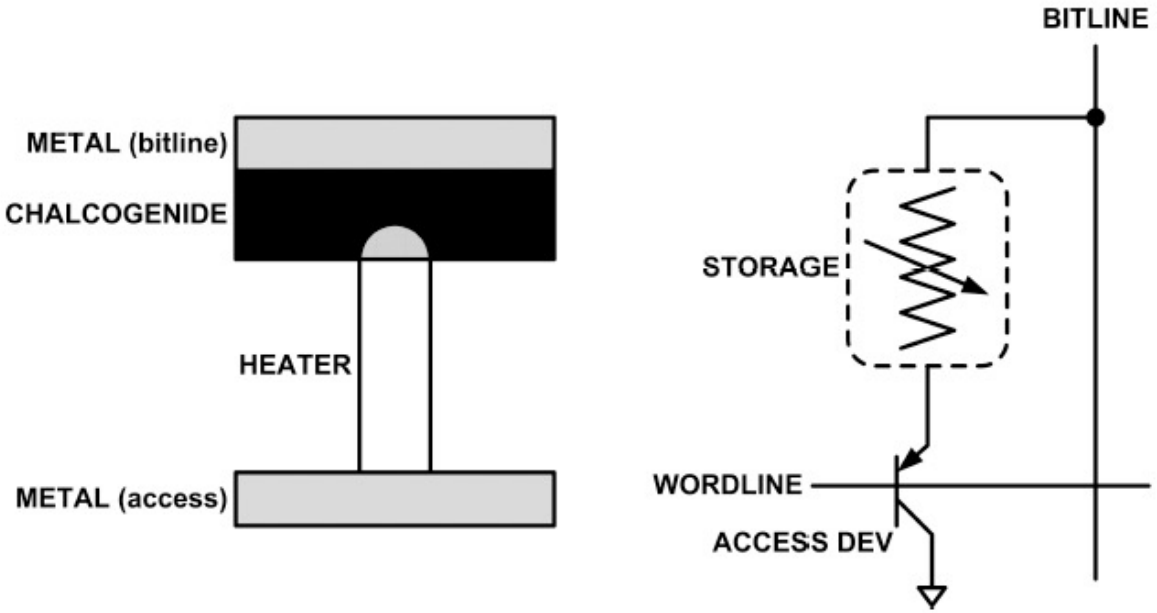
Recall: DRAM vs. SRAM

- DRAM
  - Slower access (capacitor)
  - Higher density (1T 1C cell)
  - Lower cost
  - Requires refresh (power, performance, circuitry)
  - Manufacturing requires putting capacitor and logic together

- SRAM
  - Faster access (no capacitor)
  - Lower density (6T cell)
  - Higher cost
  - No need for refresh
  - Manufacturing compatible with logic process (no capacitor)

Recall: Phase Change Memory

- Phase change material (chalcogenide glass) exists in two states:
  - Amorphous: Low optical reflexivity and high electrical resistivity
  - Crystalline: High optical reflexivity and low electrical resistivity



PCM is resistive memory: High resistance (0), Low resistance (1)

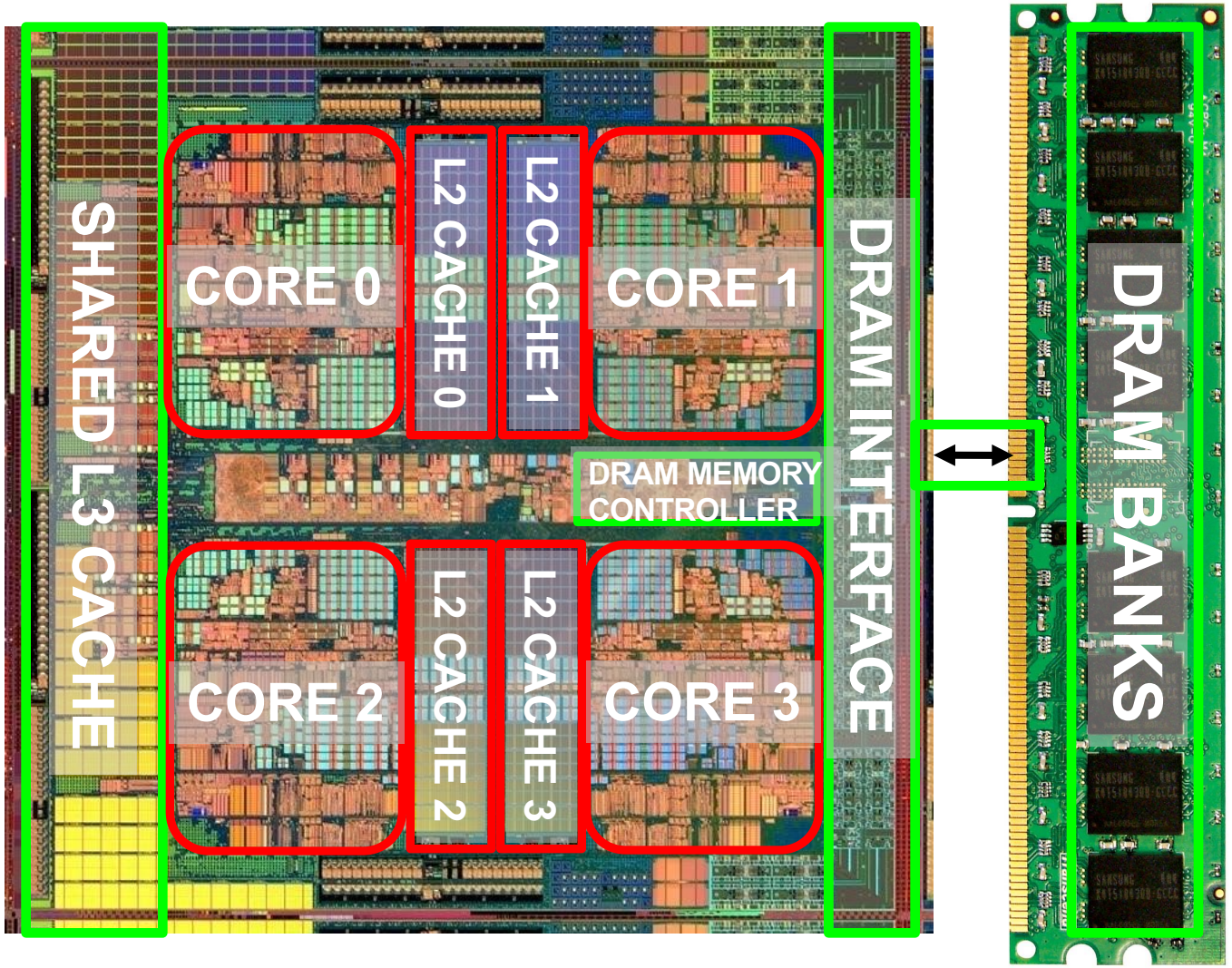
Lee, Ipek, Mutlu, Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," ISCA 2009.

Recall: DRAM vs. PCM

- DRAM
  - Faster access (capacitor)
  - Lower density (capacitor less scalable) → higher cost
  - Requires refresh (power, performance, circuitry)
  - Manufacturing requires putting capacitor and logic together
  - Volatile (loses data at loss of power)
  - No endurance problems
  - Lower access energy
- PCM
  - Slower access (no capacitor)
  - Higher density (phase change material more scalable) → lower cost
  - No need for refresh
  - Manufacturing requires less conventional processes – less mature
  - Non-volatile (does not lose data at loss of power)
  - Endurance problems (a cell cannot be used after N writes)
  - Higher access energy

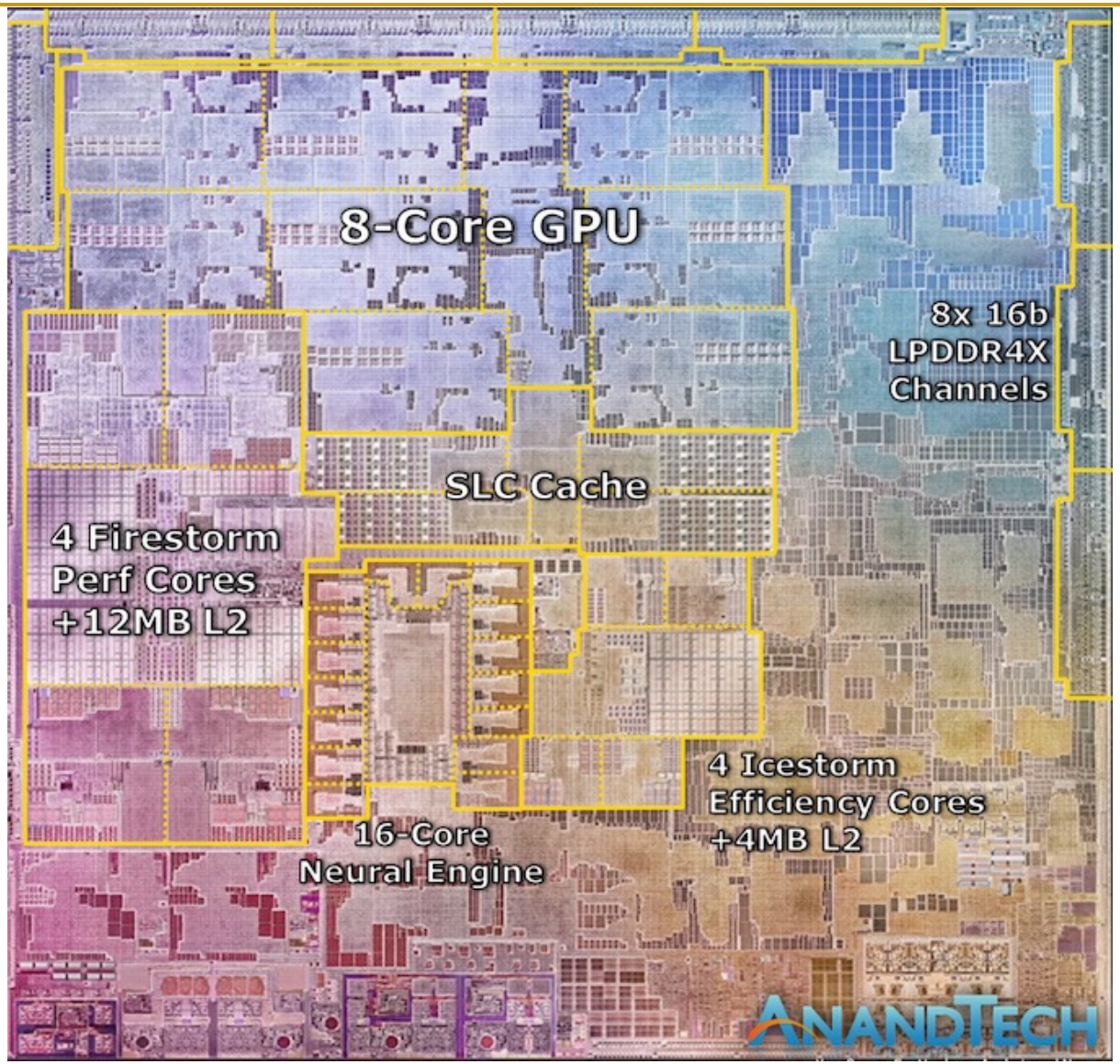
The Memory Hierarchy

Memory Hierarchy in a Modern System (I)



AMD Barcelona, circa 2006

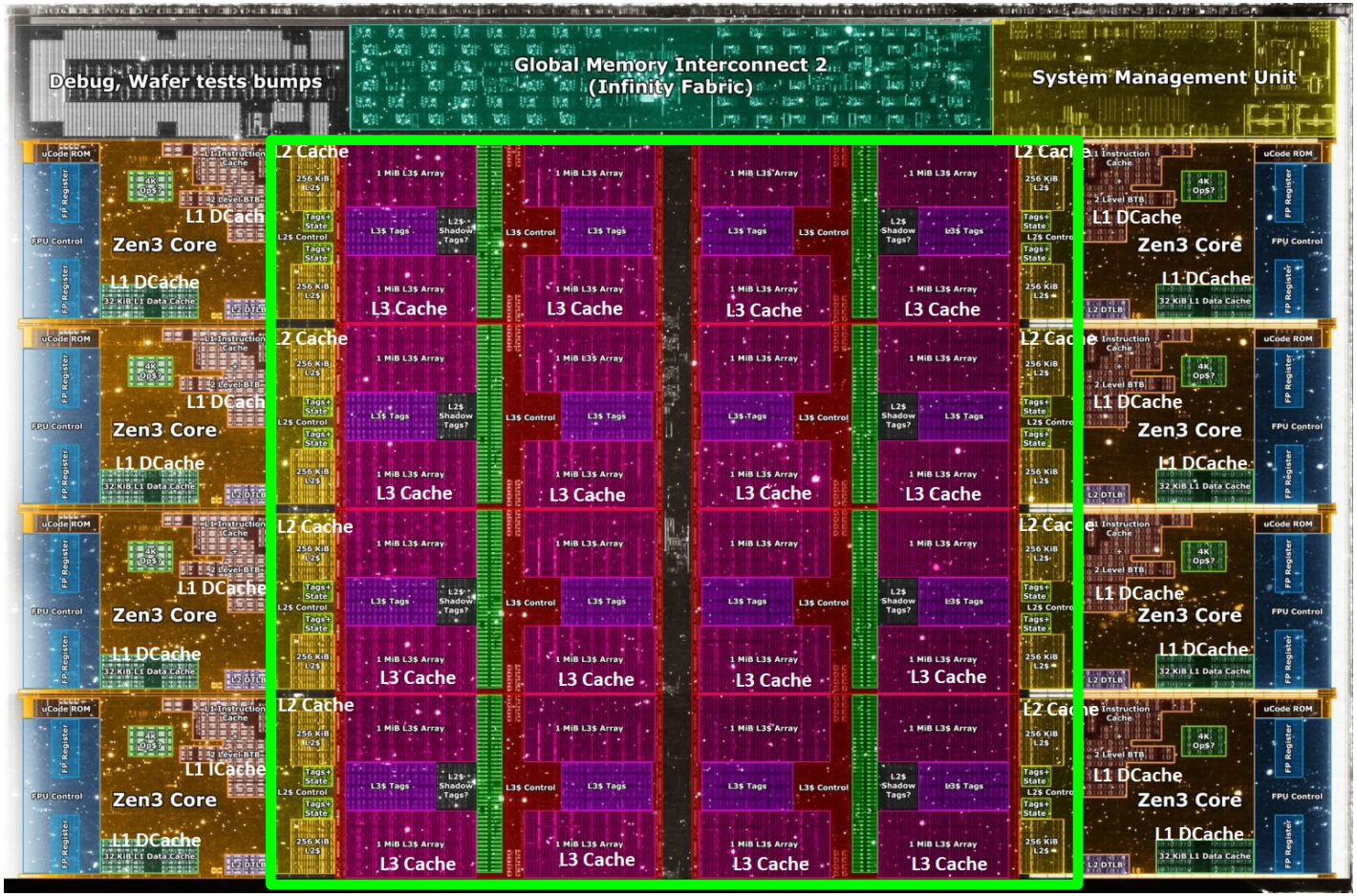
Recall: A Large Fraction of Modern Chips is Memory



Apple M1, 2021

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

Memory Hierarchy in a Modern System (II)



AMD Ryzen 5000, 2020

Core Count:
8 cores/16 threads

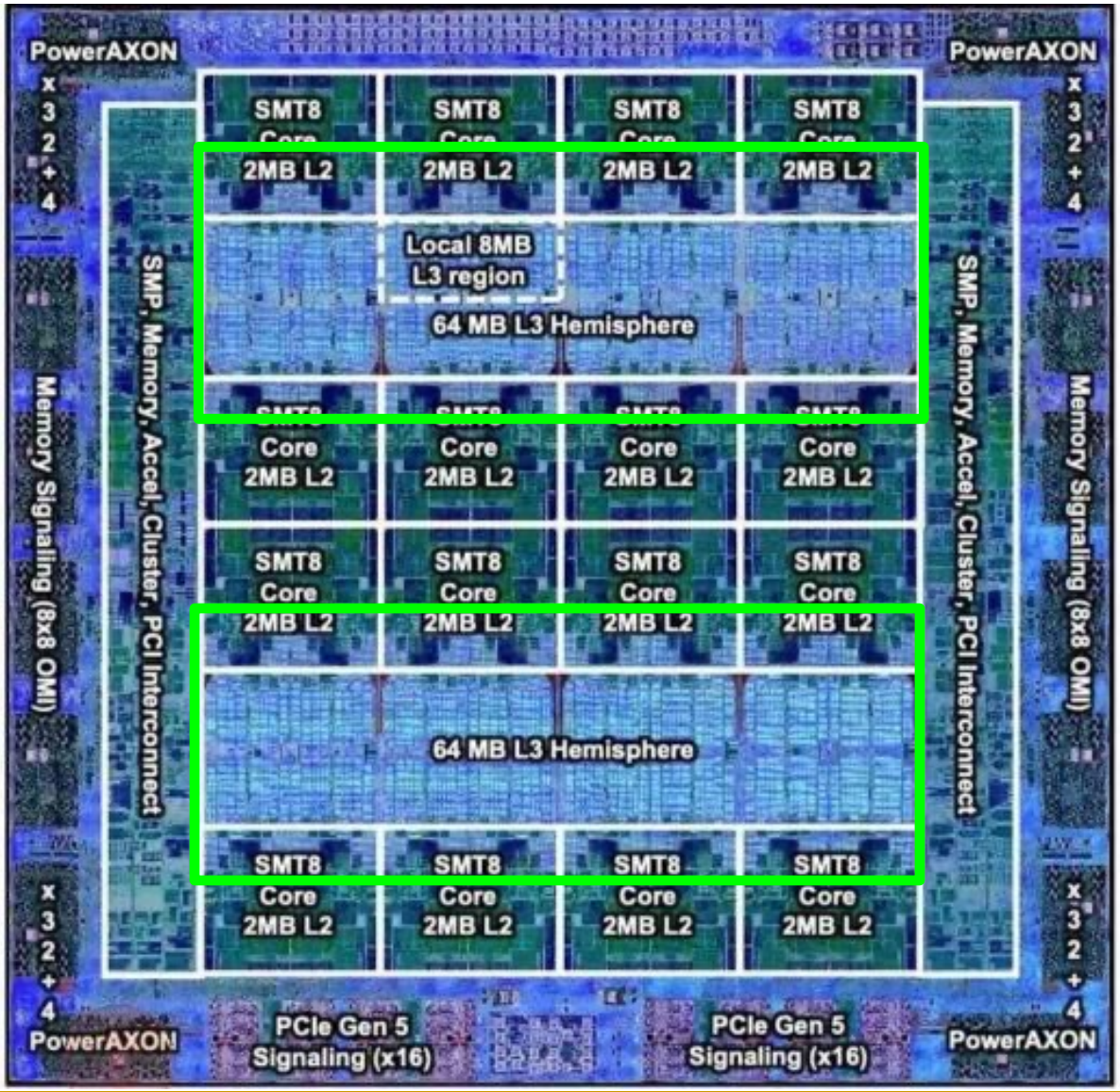
L1 Caches:
32 KB per core

L2 Caches:
512 KB per core

L3 Cache:
32 MB shared

https://wccftech.com/amd-ryzen-5000-zen-3-vermeer-undressed-high-res-die-shots-close-ups-pictured-detailed/

Memory Hierarchy in a Modern System (III)



IBM POWER10, 2020

Cores:

15-16 cores,
8 threads/core

L2 Caches:

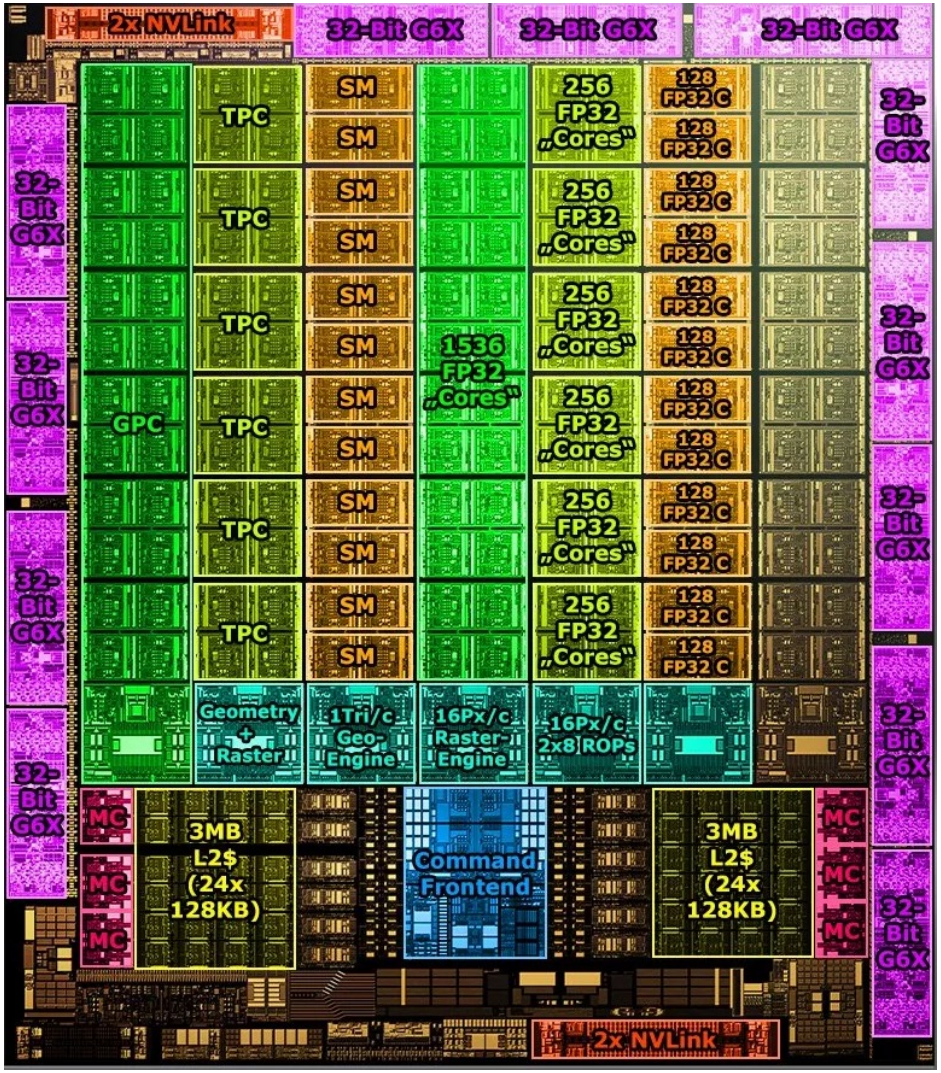
2 MB per core

L3 Cache:

120 MB shared

https://www.it-techblog.de/ibm-power10-prozessor-mehr-speicher-mehr-tempo-mehr-sicherheit/09/2020/

Memory Hierarchy in a Modern System (IV)



Nvidia Ampere, 2020

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

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

Ideal Memory

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

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. SSD vs. Disk vs. Tape

- Higher bandwidth is more expensive
  - Need more banks, more ports, more channels, higher frequency or faster technology

The Problem

- Bigger is slower
  - SRAM, 512 Bytes, sub-nanosec
  - SRAM, KByte~MByte, ~nanosec
  - DRAM, Gigabyte, ~50 nanosec
  - PCM-DIMM (Intel Optane DC DIMM), Gigabyte, ~200 nanosec
  - PCM-SSD (Intel Optane SSD), Gigabyte, ~10 μs
  - Flash memory, Gigabyte~Terabyte, ~100 μs
  - Hard Disk, Terabyte, ~10 millisec
- Faster is more expensive (dollars and chip area)
  - SRAM, < 0.3$ per Megabyte
  - DRAM, < 0.03$ per Megabyte
  - PCM-DIMM (Intel Optane DC DIMM), < 0.004$ per Megabyte
  - PCM-SSD, < 0.001$ per Megabyte
  - Flash memory, < 0.00008$ per Megabyte
  - Hard Disk, < 0.00003$ per Megabyte
  - These sample values (circa ~2021) scale with time
- Other technologies have their place as well
  - MRAM, RRAM, STT-MRAM, … (not mature yet)

The Problem (Table View)

Bigger is slower

Memory Device	Capacity	Latency	Cost per Megabyte
SRAM	512 Bytes	sub-nanosec	
SRAM	KByte~MByte	~nanosec	< 0.3$
DRAM	Gigabyte	~50 nanosec	< 0.03$
PCM-DIMM (Intel Optane DC DIMM)	Gigabyte	~200 nanosec	< 0.004$
PCM-SSD (Intel Optane SSD)	Gigabyte ~Terabyte	~10 µs	< 0.001$
Flash memory	Gigabyte ~Terabyte	~100 µs	< 0.00008$
Hard Disk	Terabyte	~10 millisec	< 0.00003$

Faster is more expensive (dollars and chip area)

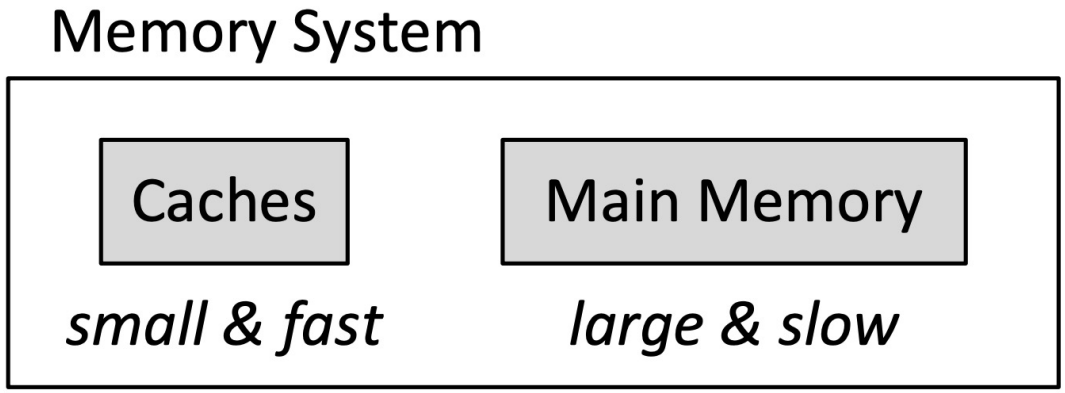
These sample values (circa ~2021) scale with time

Aside: The Problem (2011 Version)

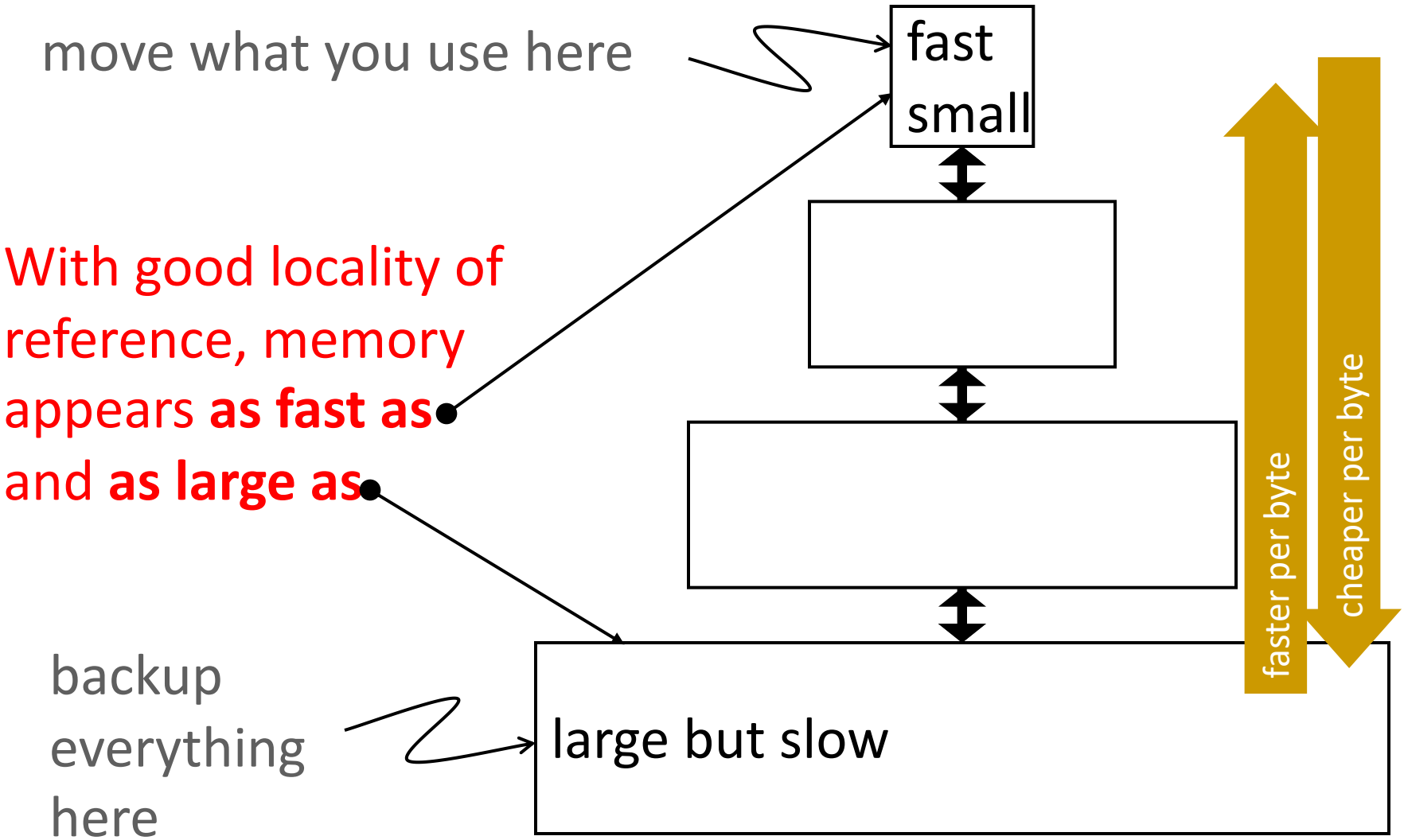
- Bigger is slower
  - SRAM, 512 Bytes, sub-nanosec
  - SRAM, KByte~MByte, ~nanosec
  - DRAM, Gigabyte, ~50 nanosec
  - Hard Disk, Terabyte, ~10 millisec
- Faster is more expensive (dollars and chip area)
  - SRAM, < 10$ per Megabyte
  - DRAM, < 1$ per Megabyte
  - Hard Disk < 1$ per Gigabyte
  - These sample values (circa ~2011) scale with time
- Other technologies have their place as well
  - Flash memory (mature), PC-RAM, MRAM, RRAM (not mature yet)

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)

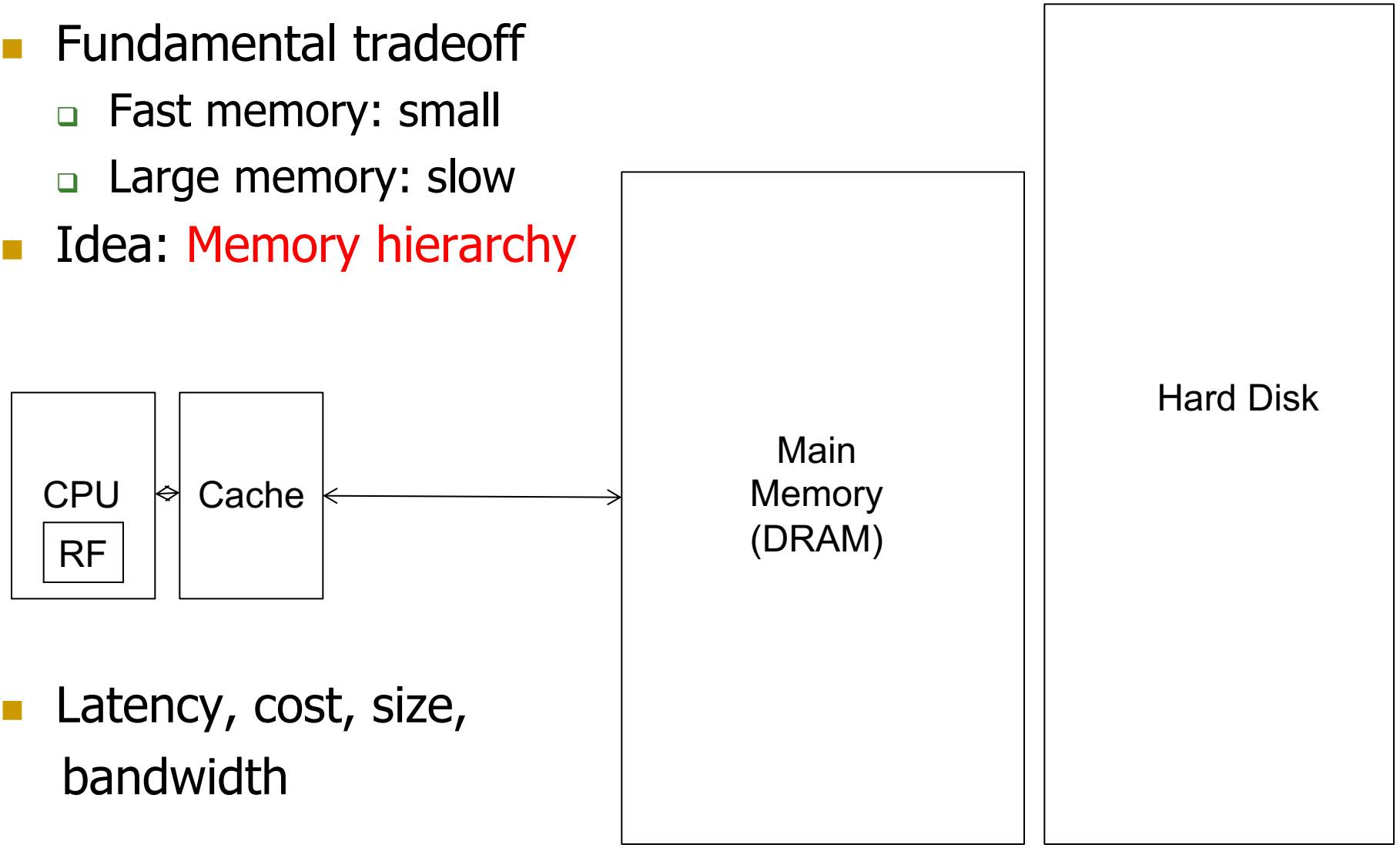


The Memory Hierarchy



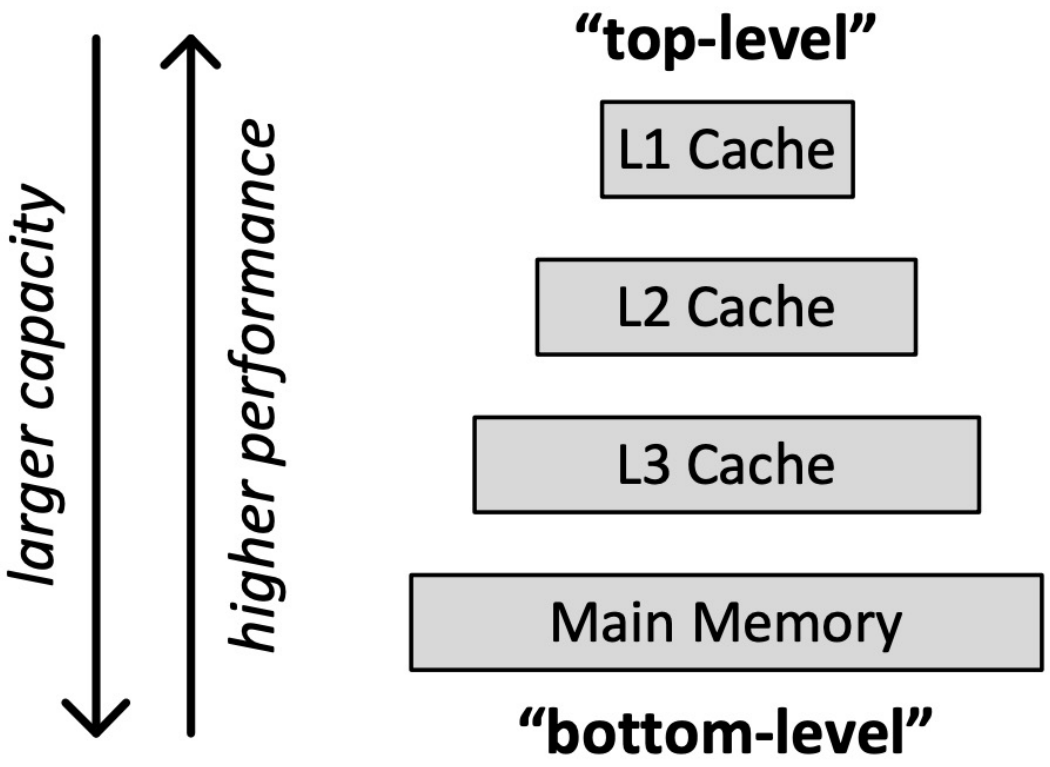
Memory Hierarchy

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



- Latency, cost, size, bandwidth

Memory Hierarchy Example



capacity	latency
10's of KB	≈ 1ns
100's of KB	< 5ns
several MB	≈ 10ns
several GB	≈ 100ns

Kim & Mutlu, "Memory Systems," Computing Handbook, 2014
https://people.inf.ethz.ch/omutlu/pub/memory-systems-introduction_computing-handbook14.pdf

Locality

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

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

Caching Basics: Exploit Temporal Locality

- Idea: Store recently accessed data in automatically-managed fast memory (called cache)
- Anticipation: same mem. location 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."

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-size 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.

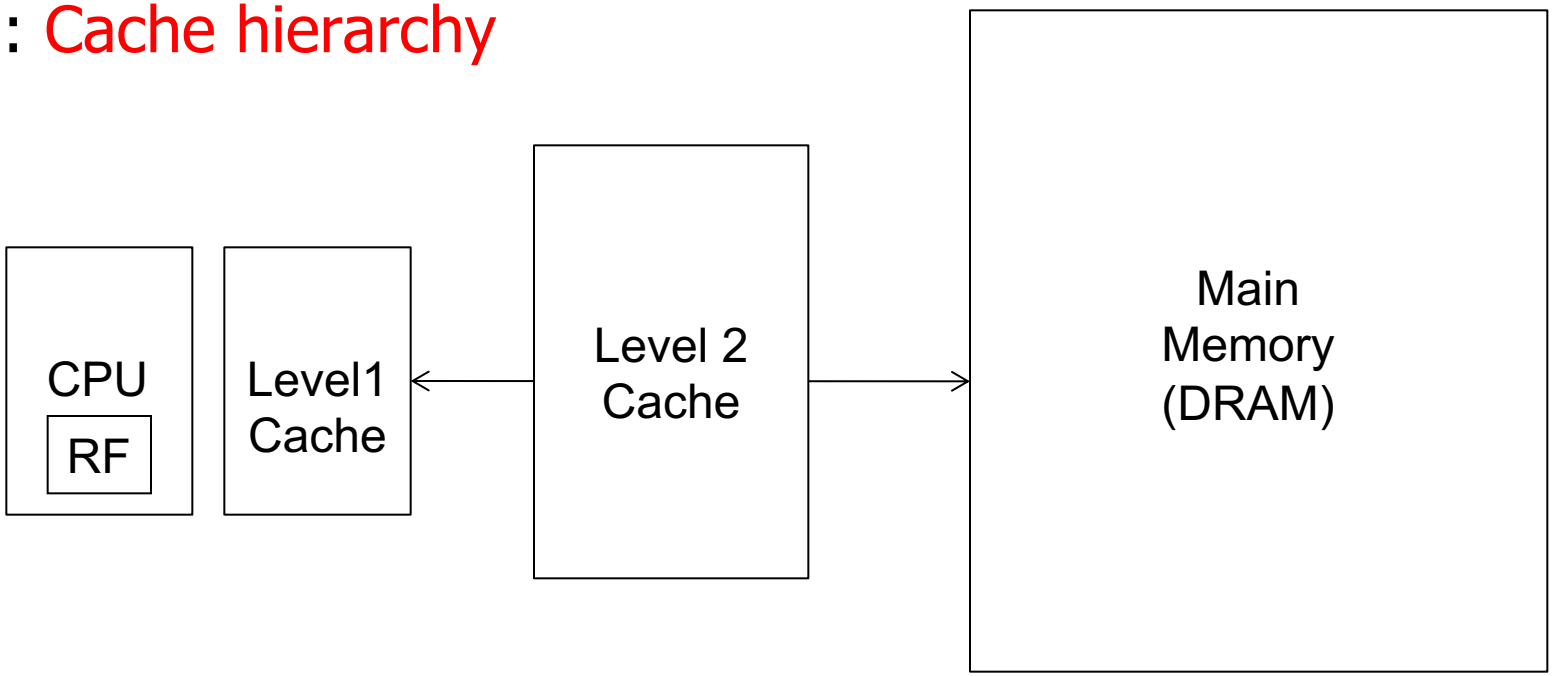
The Bookshelf Analogy

- 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

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



A Note on Manual vs. Automatic Management

- Manual: Programmer manages data movement across levels

  -- too painful for programmers on substantial programs

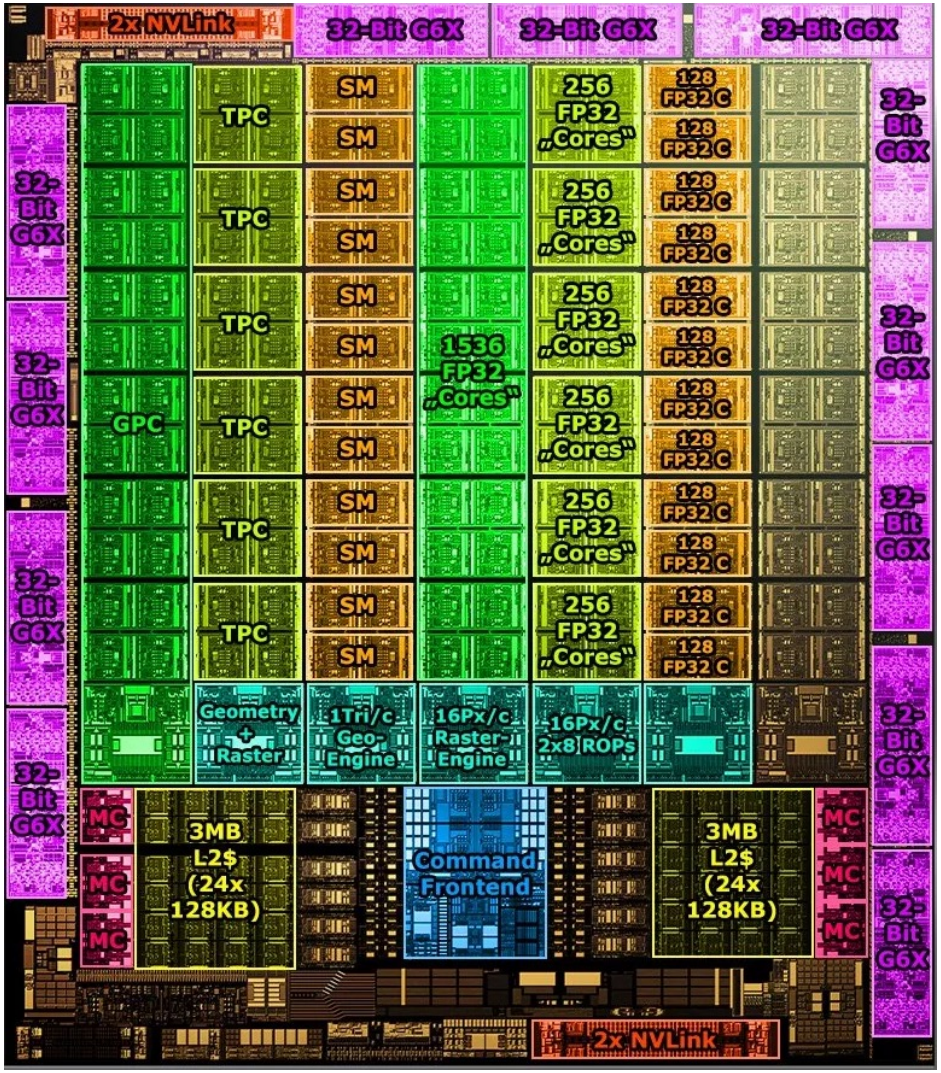
  - "core" vs "drum" memory in the 1950s
  - done in embedded processors (on-chip scratchpad SRAM in lieu of a cache), GPUs (called "shared memory"), ML accelerators, …

- 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 caches
    - 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?)

Caches and Scratchpad in a Modern GPU



Nvidia Ampere, 2020

Cores:

128 Streaming Multiprocessors

L1 Cache or Scratchpad:

192KB per SM

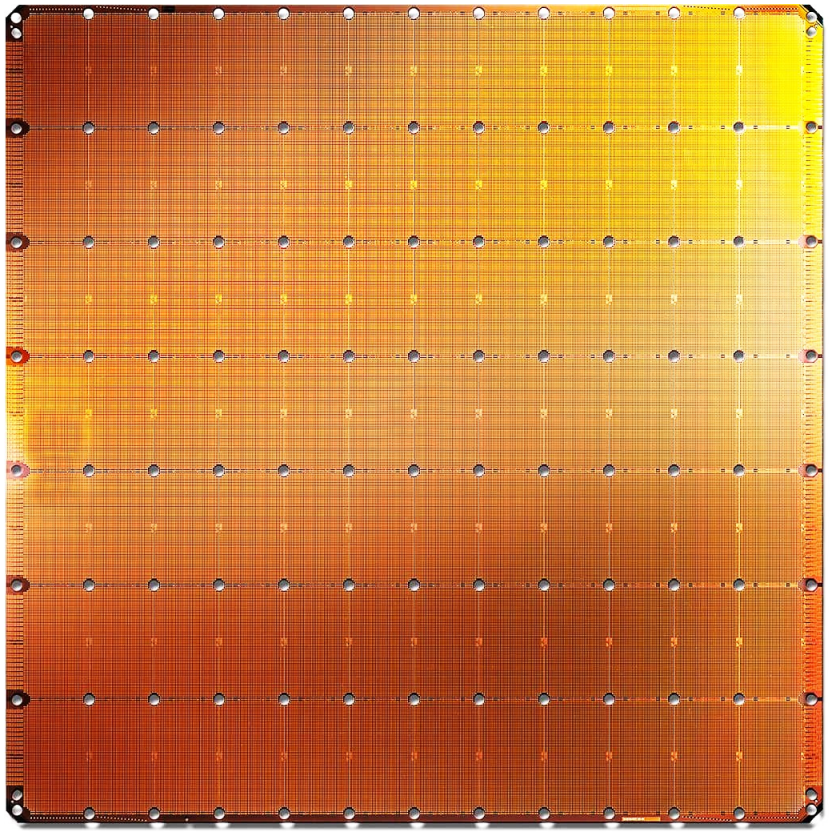
Can be used as L1 Cache and/or Scratchpad

L2 Cache:

40 MB shared

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

Cerebras's Wafer Scale Engine (2019)



- The largest ML accelerator chip
- 400,000 cores
- **18 GB of on-chip memory**
- **9 PB/s memory bandwidth**

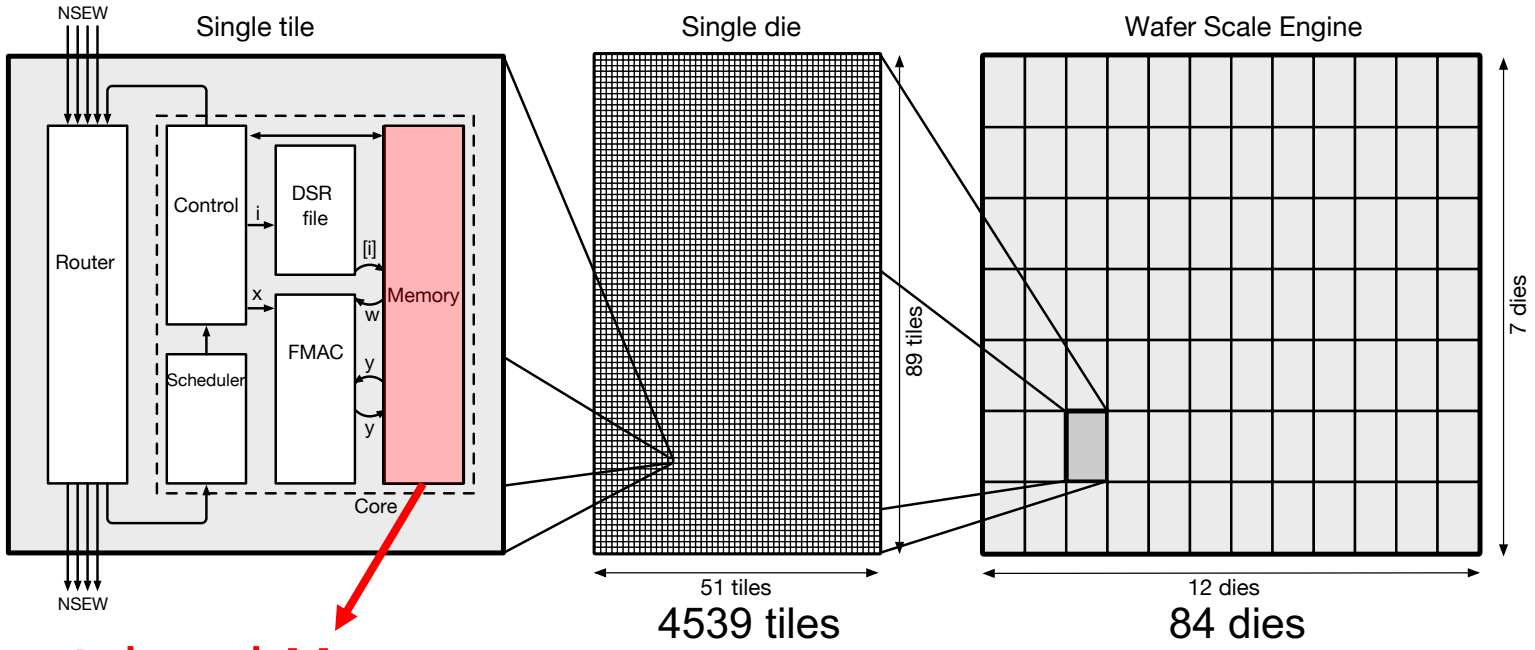
Cerebras WSE
1.2 Trillion transistors
46,225 mm^2

Largest GPU
21.1 Billion transistors
815 mm^2
NVIDIA TITAN V

https://www.anandtech.com/show/14758/hot-chips-31-live-blogs-cerebras-wafer-scale-deep-learning

https://www.cerebras.net/cerebras-wafer-scale-engine-why-we-need-big-chips-for-deep-learning/

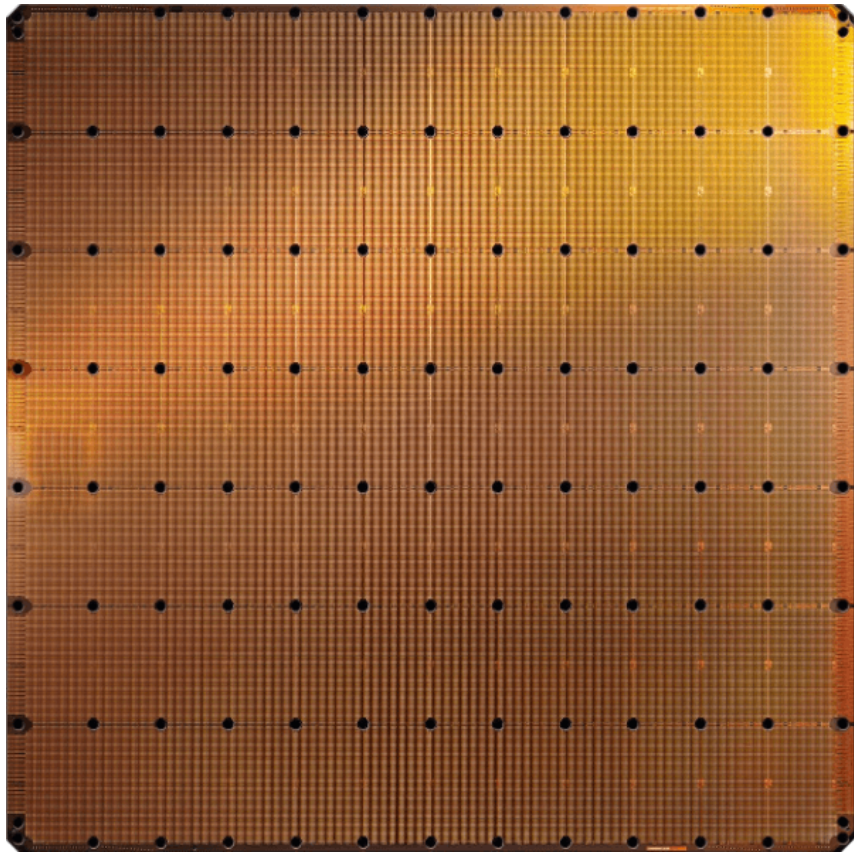
Scratchpad Memory in Cerebras WSE



- Scratchpad Memory
  - Highly parallel and distributed scratchpad SRAM memory with 2D mesh interconnection fabric across tiles
  - 16-byte read and 8-byte write single-cycle latency
  - 48 KB scratchpad in each tile, totaling 18 GB on the full chip
  - No shared memory

Rocki et al., “Fast stencil-code computation on a wafer-scale processor.” SC 2020.

Cerebras's Wafer Scale Engine-2 (2021)



- The largest ML accelerator chip
- 850,000 cores
- **40 GB of on-chip memory**
- **20 PB/s memory bandwidth**

Cerebras WSE-2
2.6 Trillion transistors
46,225 mm^2

Largest GPU
54.2 Billion transistors
826 mm^2
NVIDIA Ampere GA100

https://cerebras.net/product/#overview

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."

Historical Aside: Other Cache Papers

- Fotheringham, "Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store," CACM 1961.
  - http://dl.acm.org/citation.cfm?id=366800

- Bloom, Cohen, Porter, "Considerations in the Design of a Computer with High Logic-to-Memory Speed Ratio," AIEE Gigacycle Computing Systems Winter Meeting, Jan. 1962.

Cache in 1962 (Bloom, Cohen, Porter)

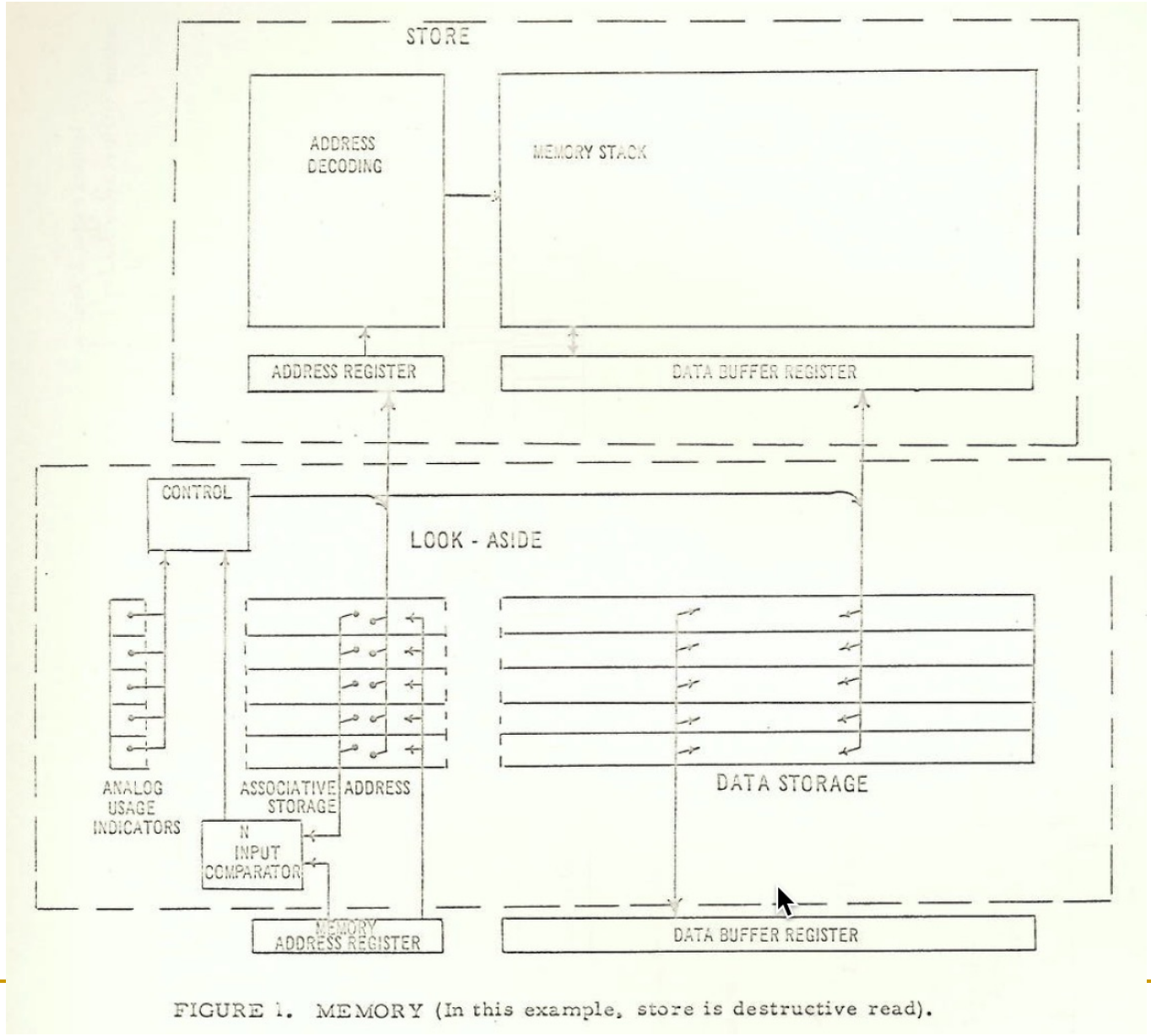
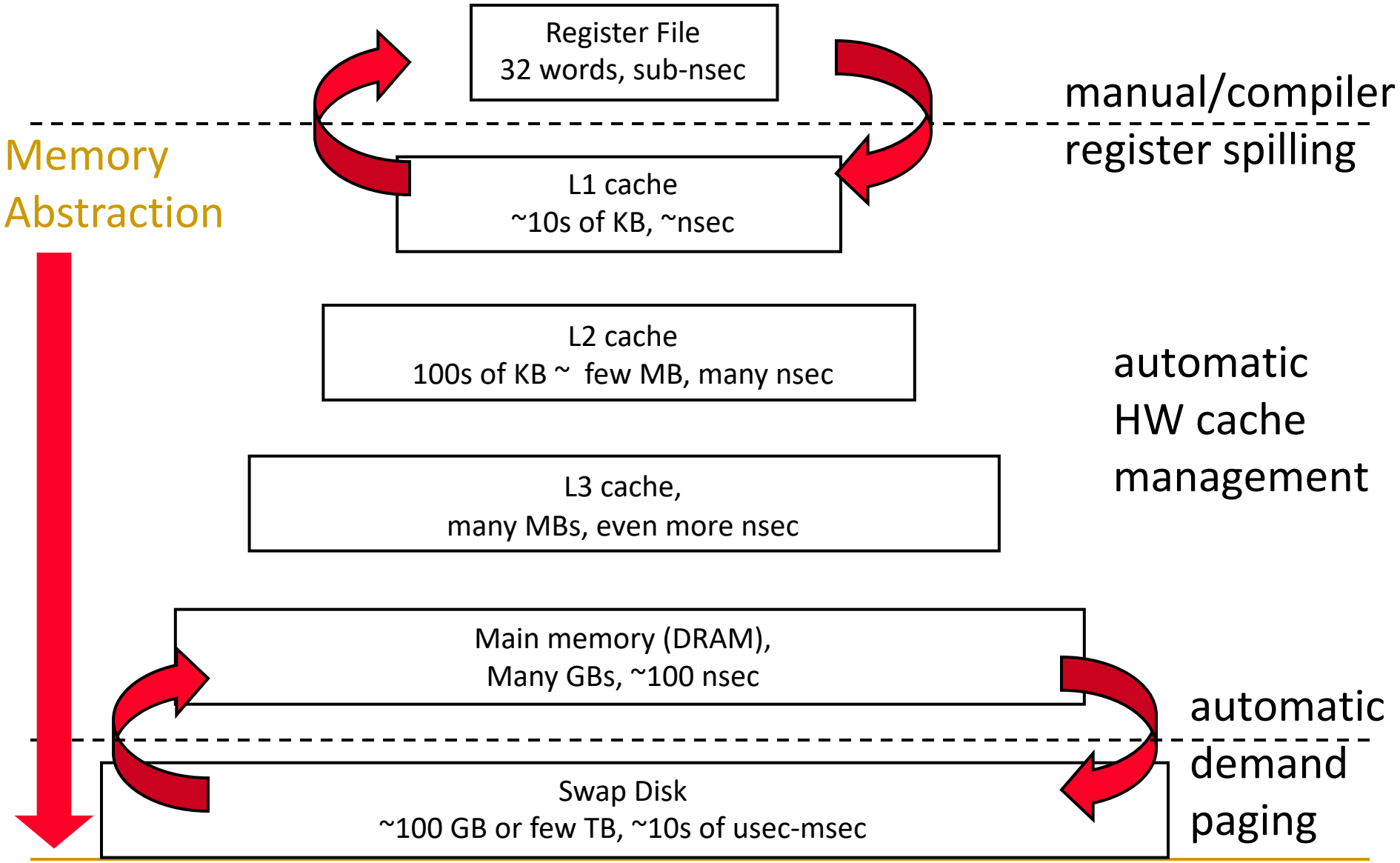


FIGURE 1. MEMORY (In this example, store is destructive read).

A Modern Memory Hierarchy



Hierarchical Latency Analysis

- For a given memory hierarchy level i it has a technology-intrinsic access time of t_i, The **perceived access time** T_i 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) you "hit" and access time is t_i
  - a chance (miss-rate m_i) you "miss" and access time $t_i + T_{i+1}$
  - $h_i + m_i = 1$
- Thus

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

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

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

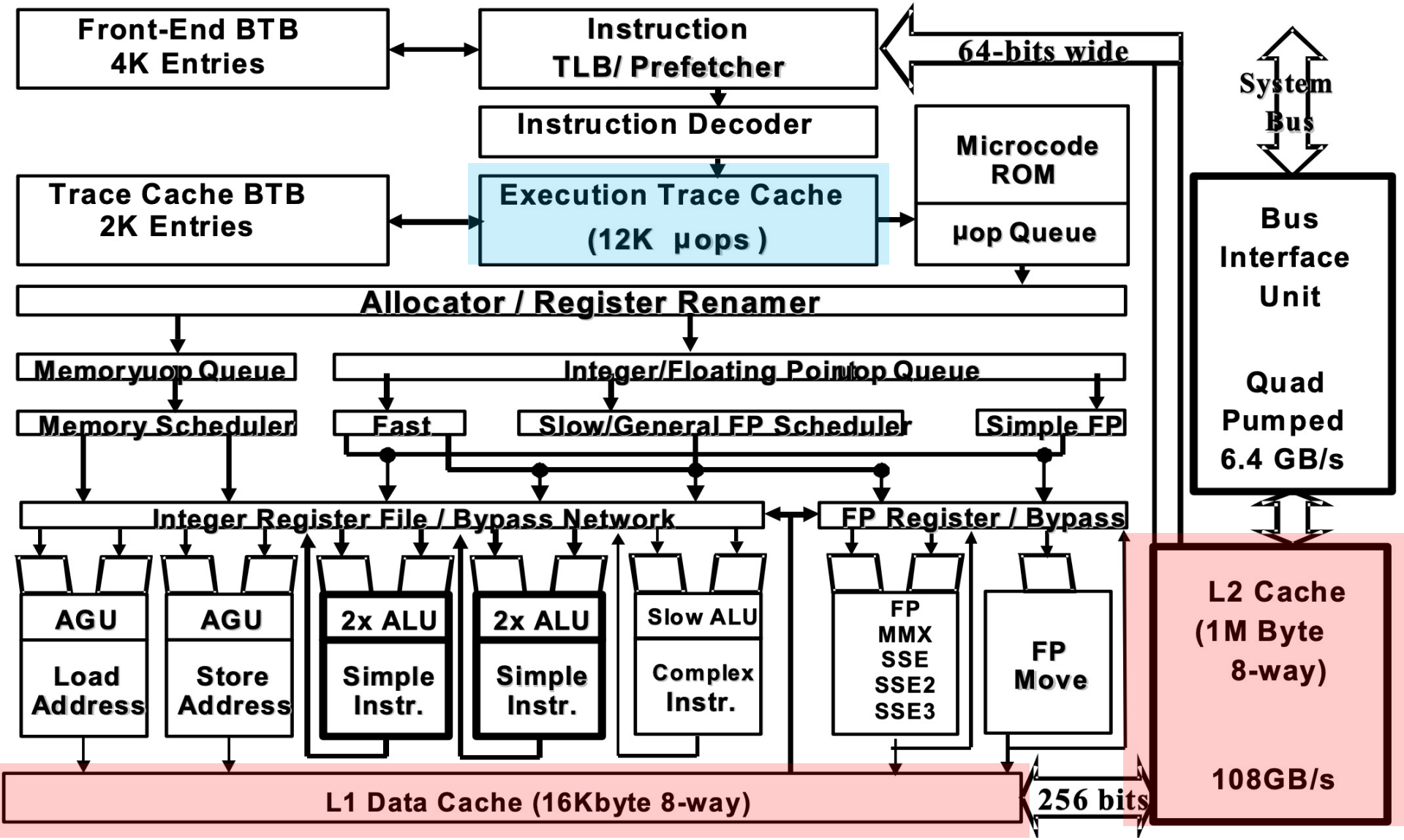
$$T_i = t_i + m_i \cdot T_{i+1}$$

- 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, prefetching::anticipate what you will need)

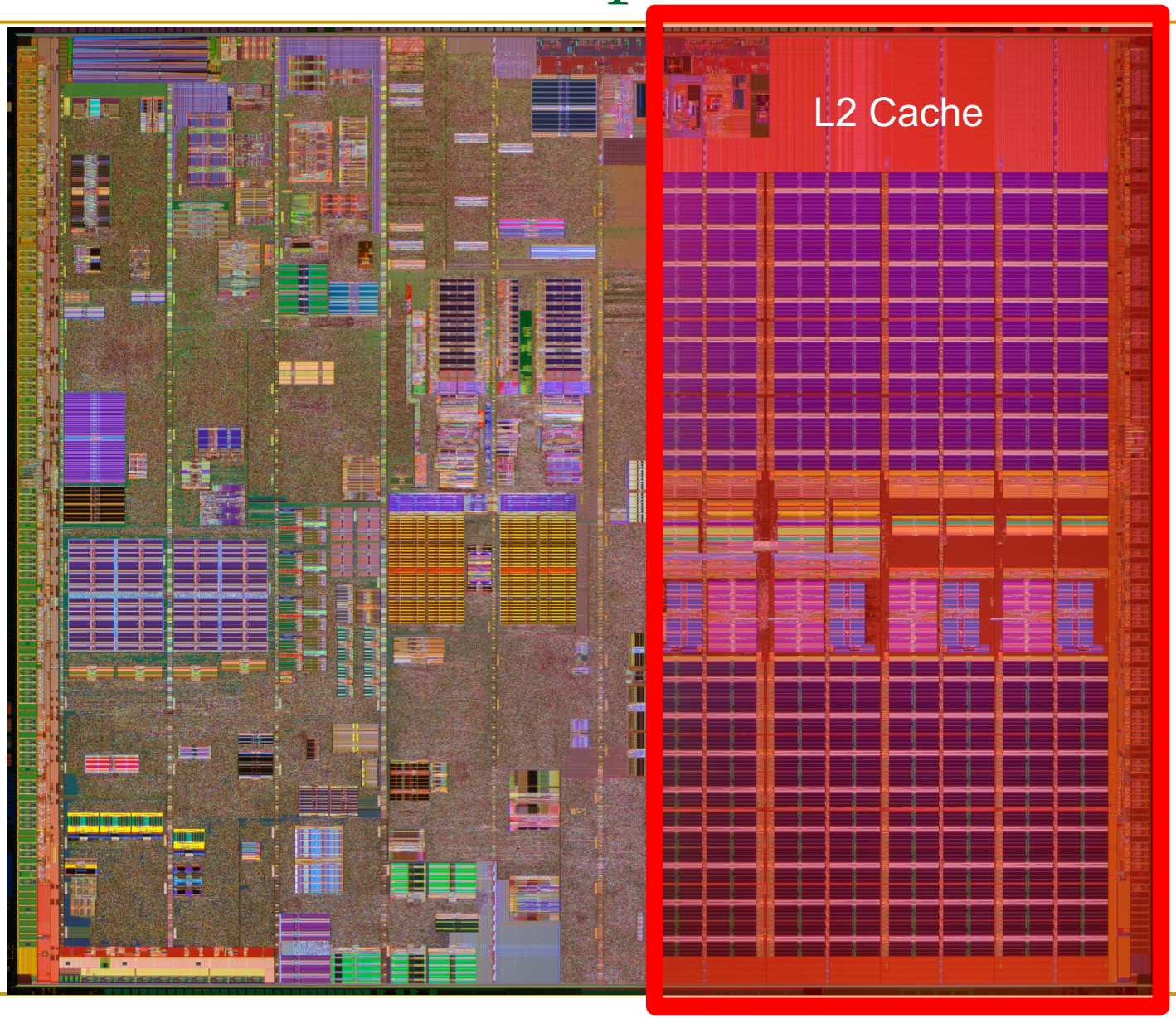
- Keep T_{i+1} low
  - faster lower hierarchies, but beware of increasing cost
  - introduce intermediate hierarchies as a compromise

Intel Pentium 4 Example



Boggs et al., "The Microarchitecture of the Pentium 4 Processor," Intel Technology Journal, 2004.

Intel Pentium 4 Example



https://download.intel.com/newsroom/kits/40thanniversary/gallery/images/Pentium_4_6xx-die.jpg

Intel Pentium 4 Example

- 90nm P4, 3.6 GHz
- L1 D-cache
  - C_1 = 16 kB
  - t_1 = 4 cyc int / 9 cycle fp
- 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

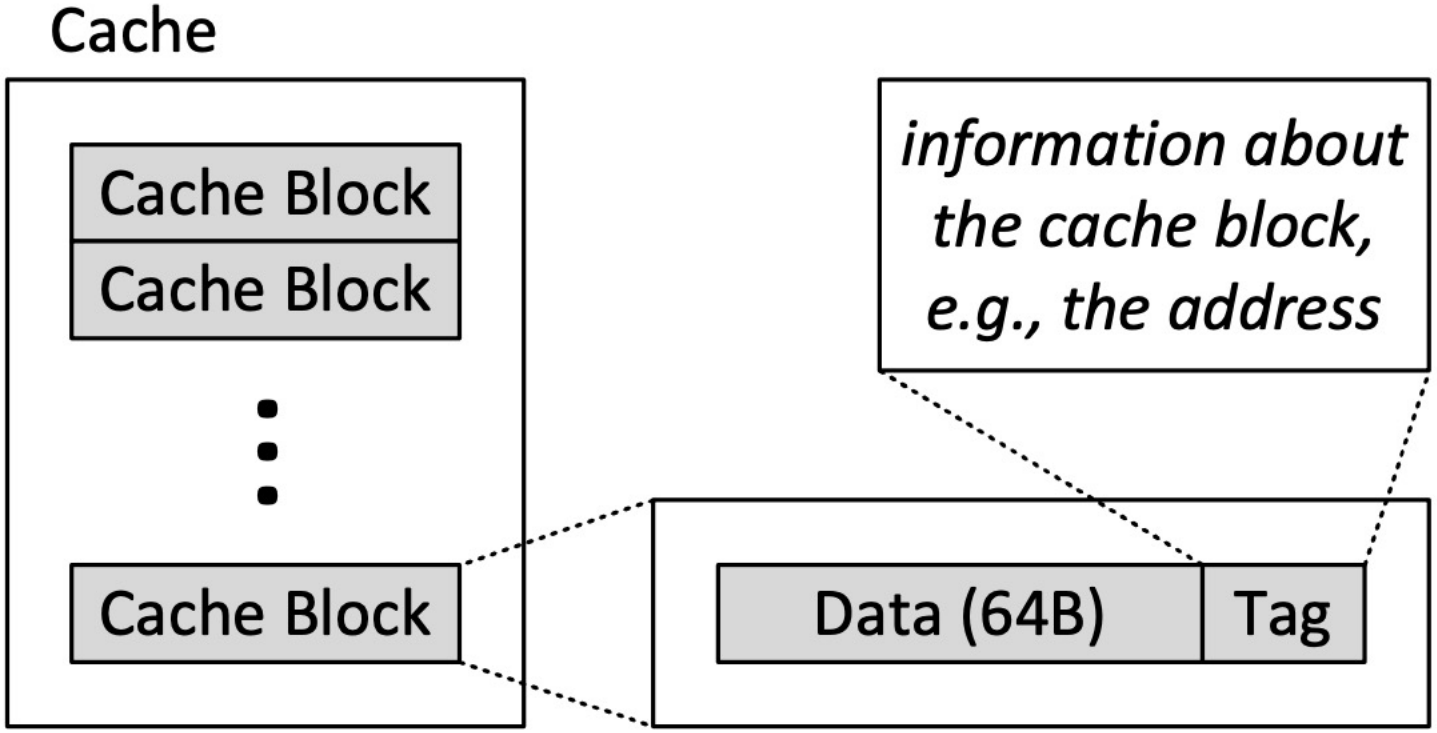
Cache Basics and Operation

Cache

- Any structure that "memoizes" frequently used results/data
  - to avoid repeating the long-latency operations required to reproduce/fetch the results/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

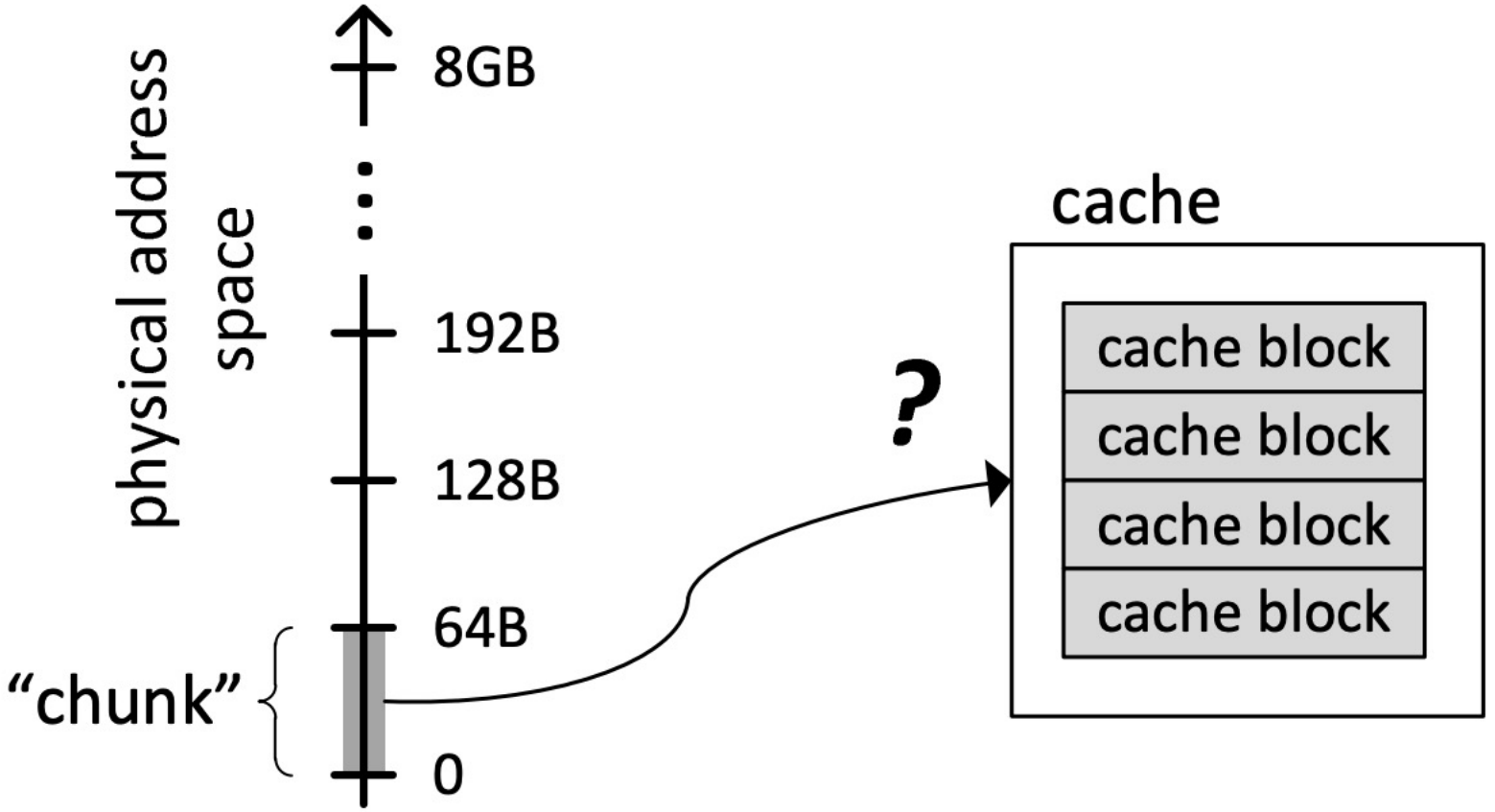
Conceptual Picture of a Cache



Kim & Mutlu, "Memory Systems," Computing Handbook, 2014
https://people.inf.ethz.ch/omutlu/pub/memory-systems-introduction_computing-handbook14.pdf

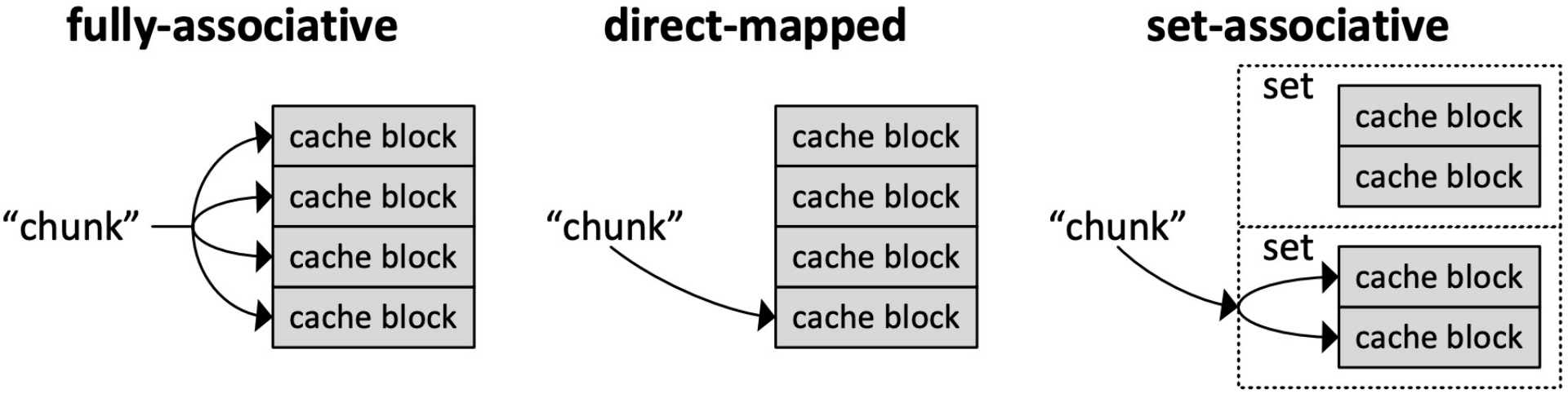
Logical Organization of a Cache (I)

- 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)

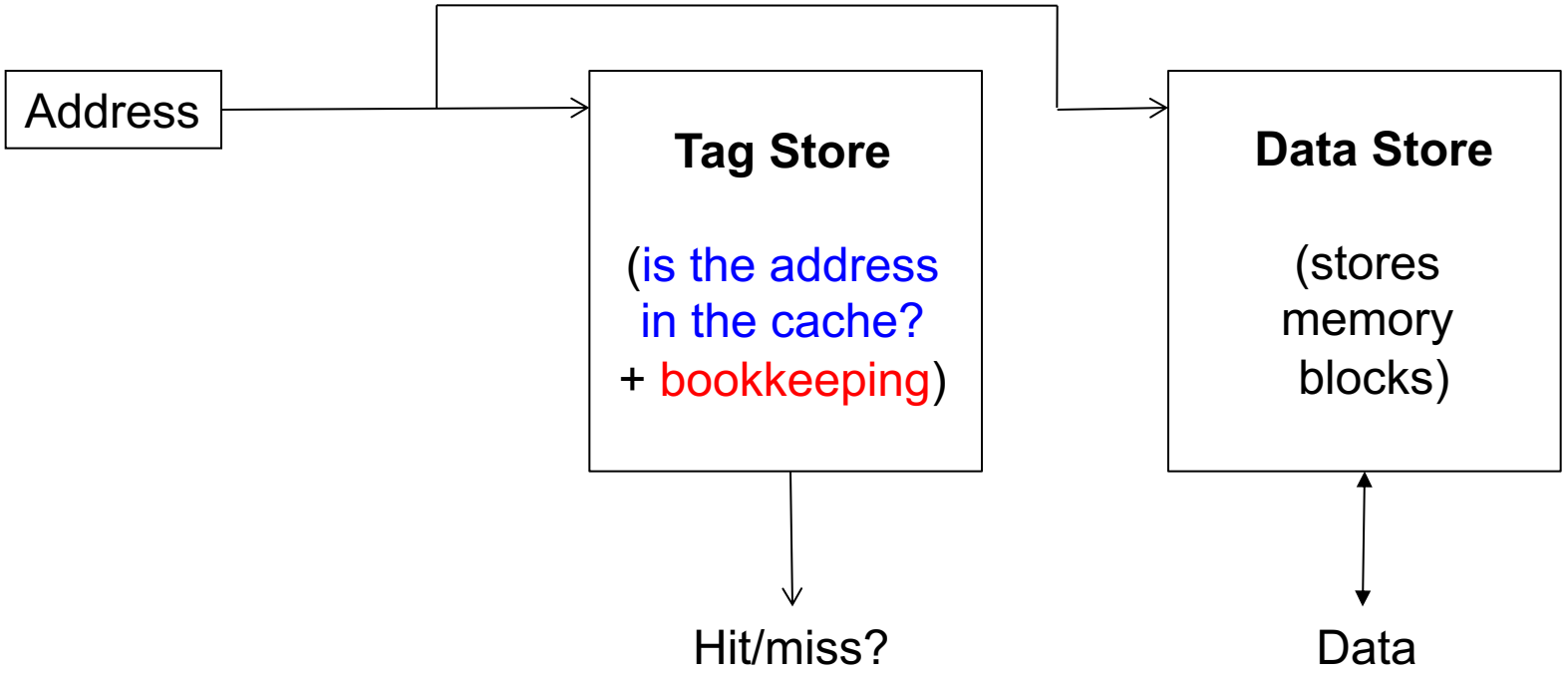
- A key question: How to map chunks of the main memory address space to blocks in the cache?
  - Which location in cache can a given "main memory chunk" be placed in?



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



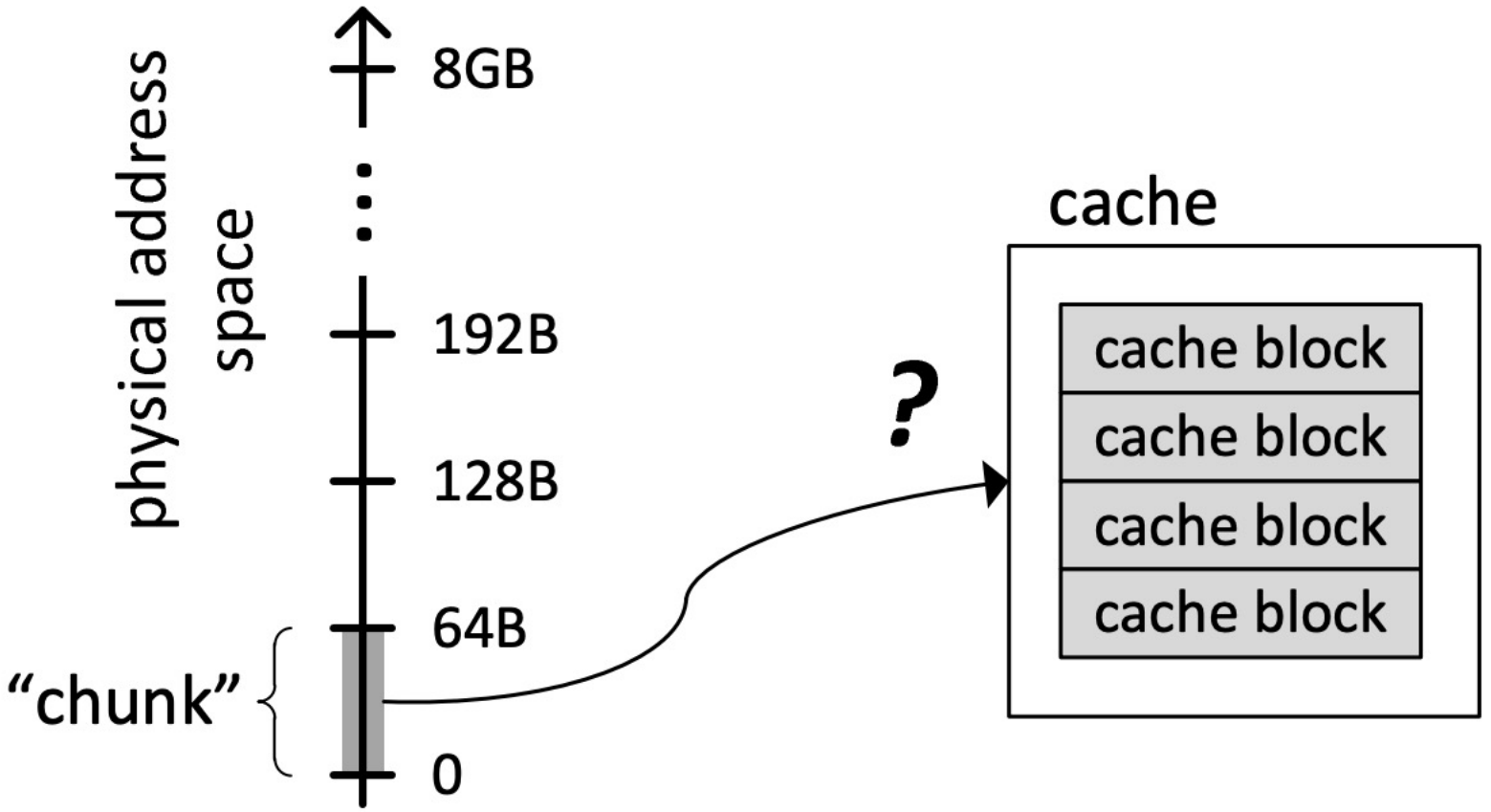
- Cache hit rate = (# hits) / (# hits + # misses) = (# hits) / (# accesses)
- Average memory access time (AMAT)
  = (hit-rate * hit-latency) + (miss-rate * miss-latency)
- Important Aside: *Is reducing AMAT always beneficial for performance?*

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

Blocks and Addressing the Cache

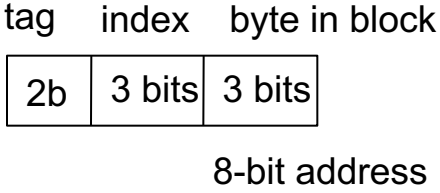
- Main memory logically divided into fixed-size chunks (**blocks**)
- **Cache** can house only a **limited** number of blocks



Blocks and Addressing the Cache

- Main memory 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 address

  2) check valid bit in tag store

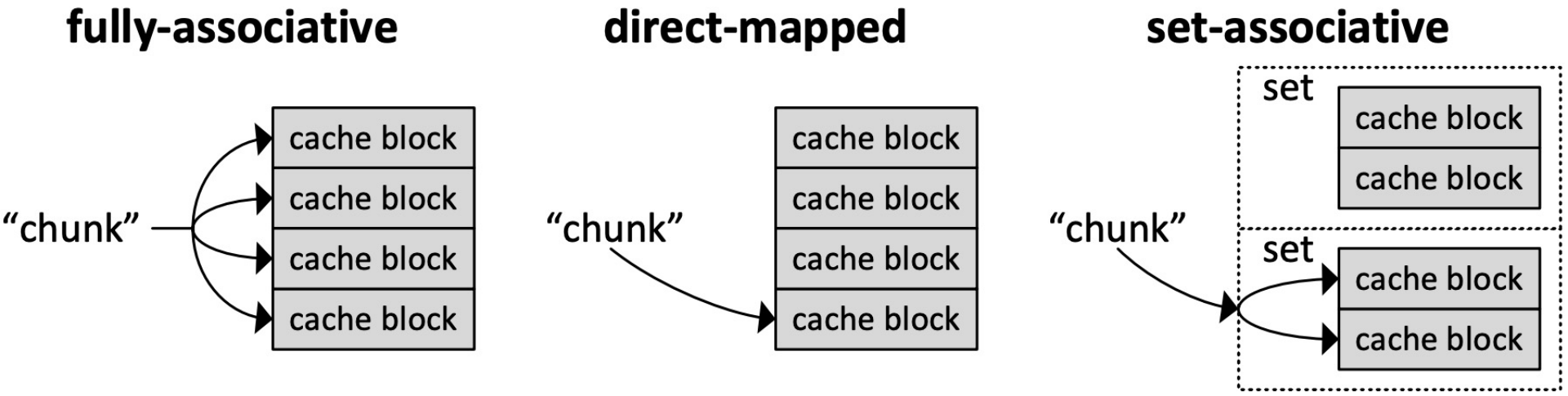
  3) compare tag bits in address with the stored tag in tag store

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

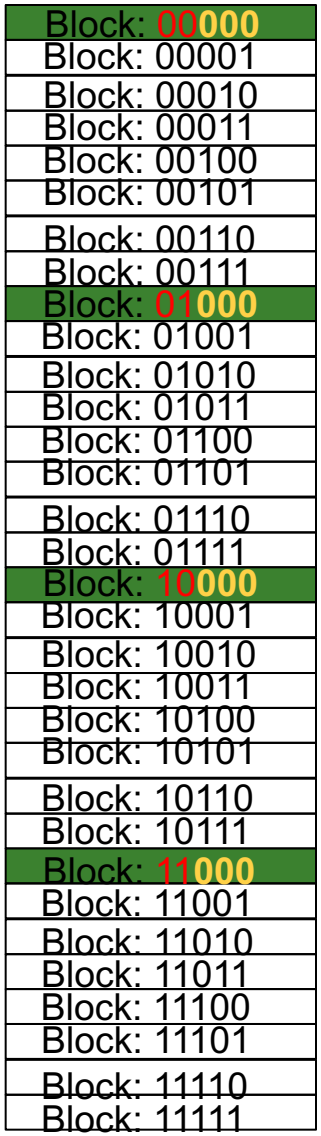
Let's See A Toy Example

- 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

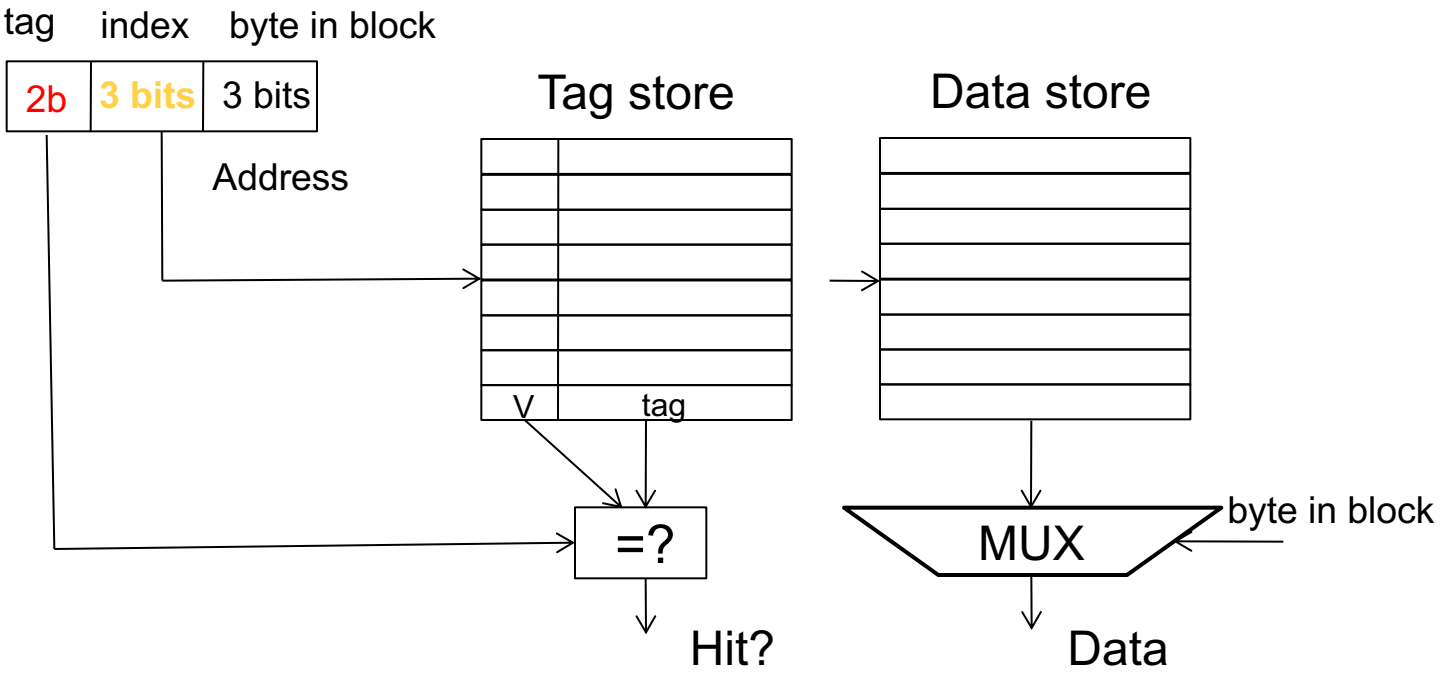
- Toy example: 256-byte memory, 64-byte cache, 8-byte blocks



Direct-Mapped Cache: Placement and Access



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



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

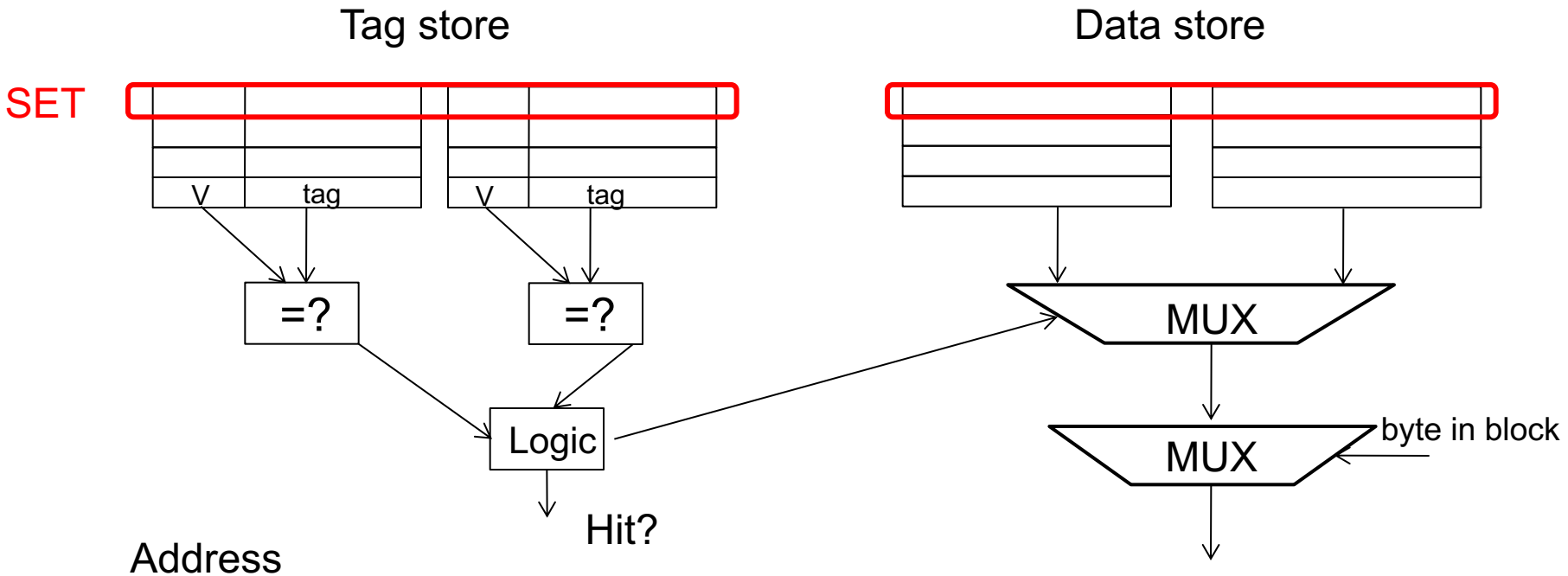
Direct-Mapped Caches

- 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 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

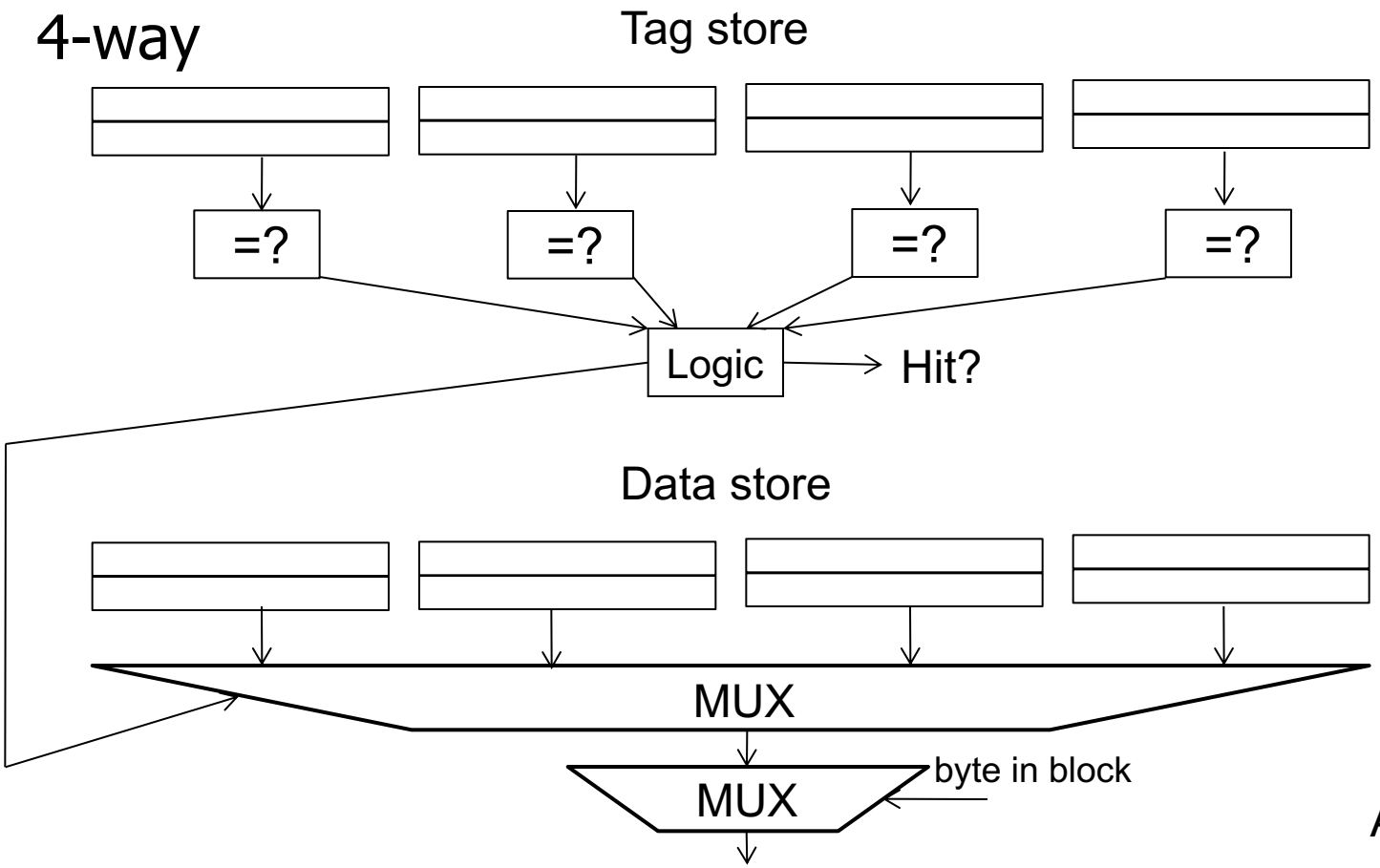
- Addresses N and N+8 always conflict in direct mapped cache
- Instead of having one column of 8, have 2 columns of 4 blocks



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

Higher Associativity

- 4-way



Address

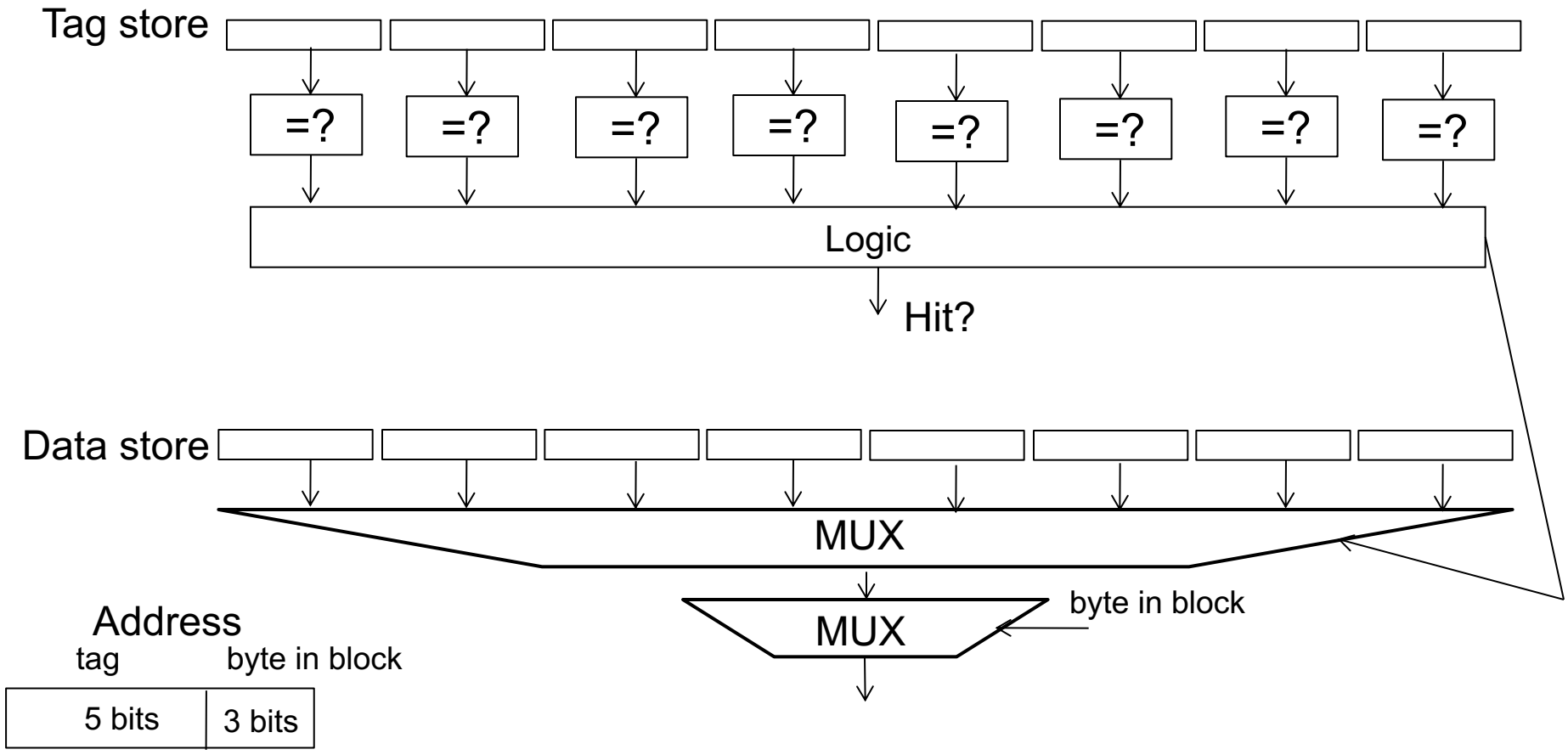
tag	index	byte in block
4 bits	1 b	3 bits

+ 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)?

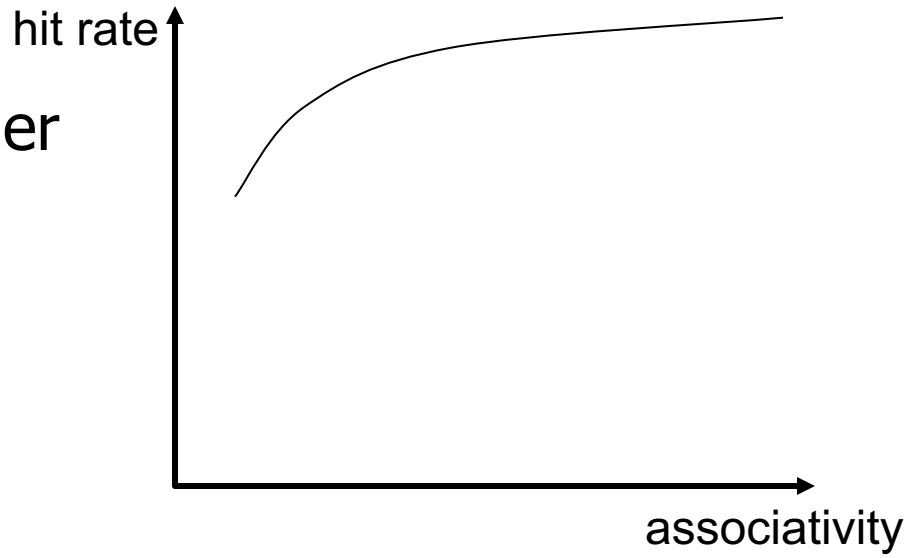
- Higher associativity

  ++ Higher hit rate

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

  -- More expensive hardware (more comparators)

- Diminishing returns from higher associativity



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?

Digital Design & Computer Arch.

Lecture 23: Memory Hierarchy and Caches

Prof. Onur Mutlu

ETH Zürich

Spring 2021

27 May 2021

We Will Cover The Remaining Slides in Future Lectures

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:
  - 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:
  - 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 and miss overlapping
  - Qureshi et al. "A Case for MLP-Aware Cache Replacement," ISCA 2006.

Recommended Reading

- Key observation: Some misses more costly than others as their latency is exposed as stall time. Reducing miss rate is not always good for performance. Cache replacement should take into account cost of misses.

- Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt, **"A Case for MLP-Aware Cache Replacement"**
  *Proceedings of the 33rd International Symposium on Computer Architecture (**ISCA**)*, pages 167-177, Boston, MA, June 2006. Slides (ppt)

A Case for MLP-Aware Cache Replacement

Moinuddin K. Qureshi Daniel N. Lynch Onur Mutlu Yale N. Patt
Department of Electrical and Computer Engineering
The University of Texas at Austin
{moin, lynch, onur, patt}@hps.utexas.edu

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

  + 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

  + 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?
  - Allocate on write miss: Yes
  - No-allocate on write miss: No

- Allocate on write miss

  + 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

  + 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)

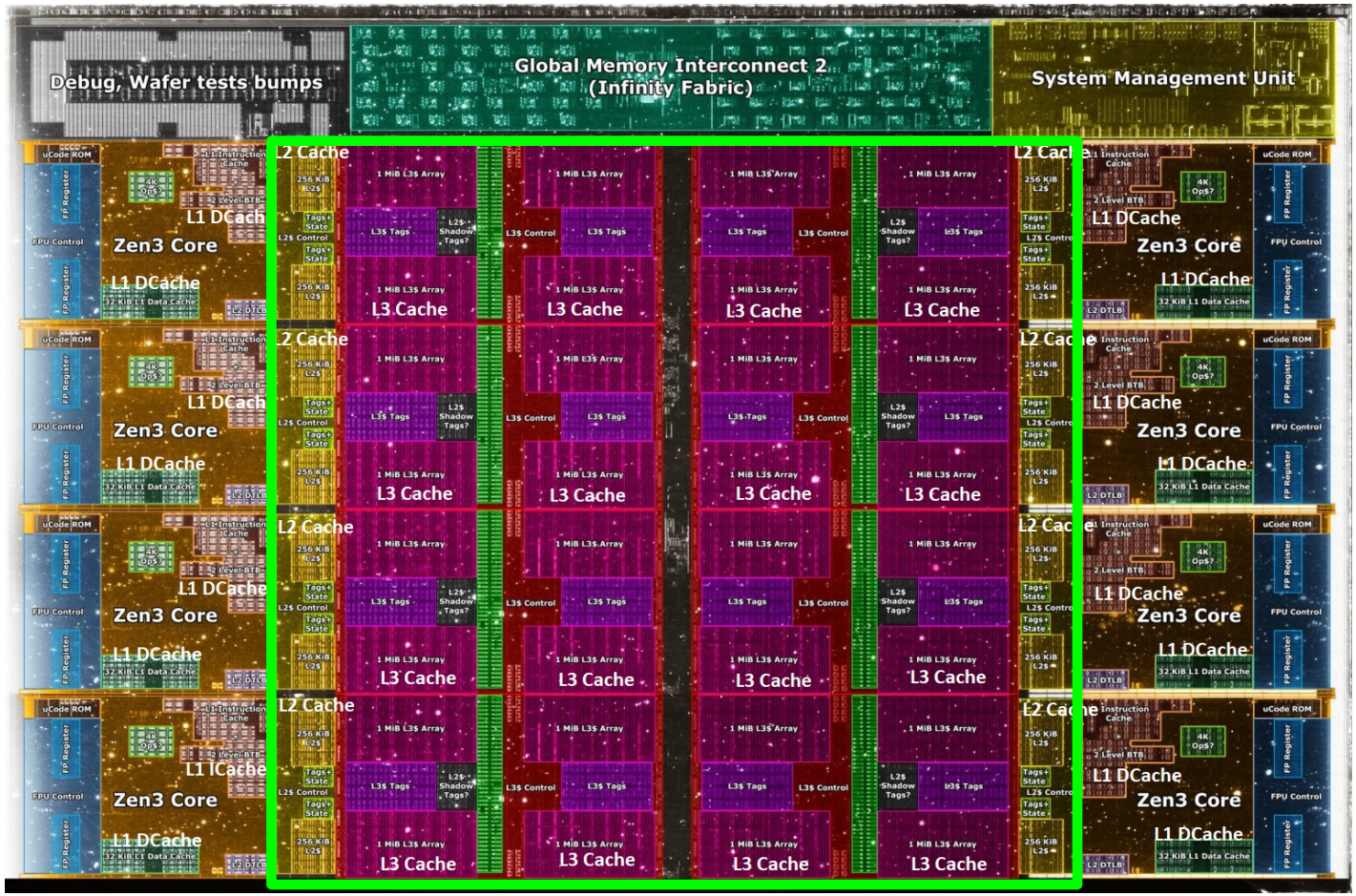
  -- I and D are accessed in different places in the pipeline. Where do we place the unified cache for fast access?

- First level caches are almost always split
  - Mainly for the last reason above
- Higher level caches are almost always unified

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 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
- 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

Deeper and Larger Cache Hierarchies



AMD Ryzen 5000, 2020

Core Count:

8 cores/16 threads

L1 Caches:

32 KB per core

L2 Caches:

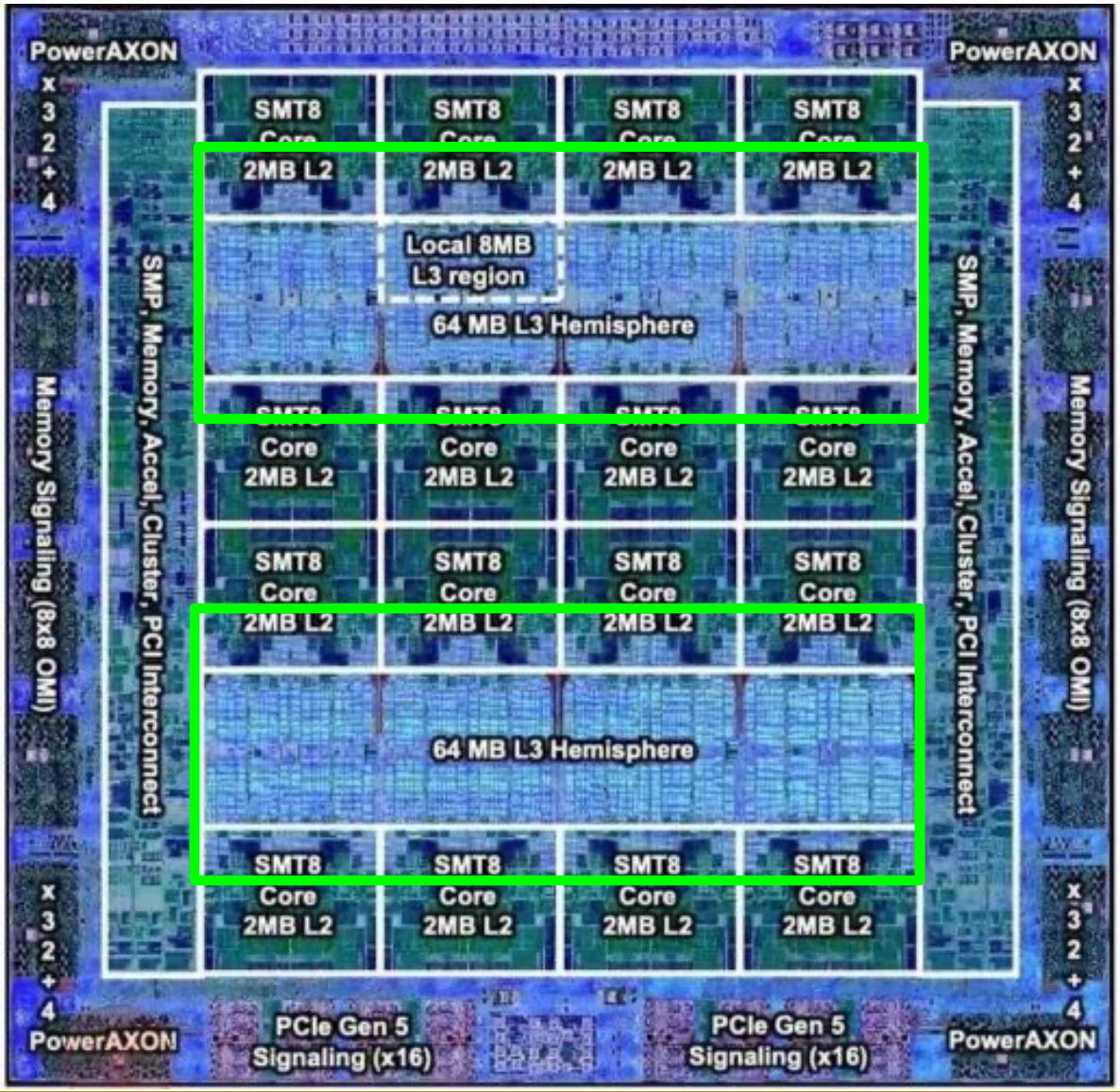
512 KB per core

L3 Cache:

32 MB shared

https://wccftech.com/amd-ryzen-5000-zen-3-vermeer-undressed-high-res-die-shots-close-ups-pictured-detailed/

Deeper and Larger Cache Hierarchies



IBM POWER10, 2020

Cores:

15-16 cores,
8 threads/core

L2 Caches:

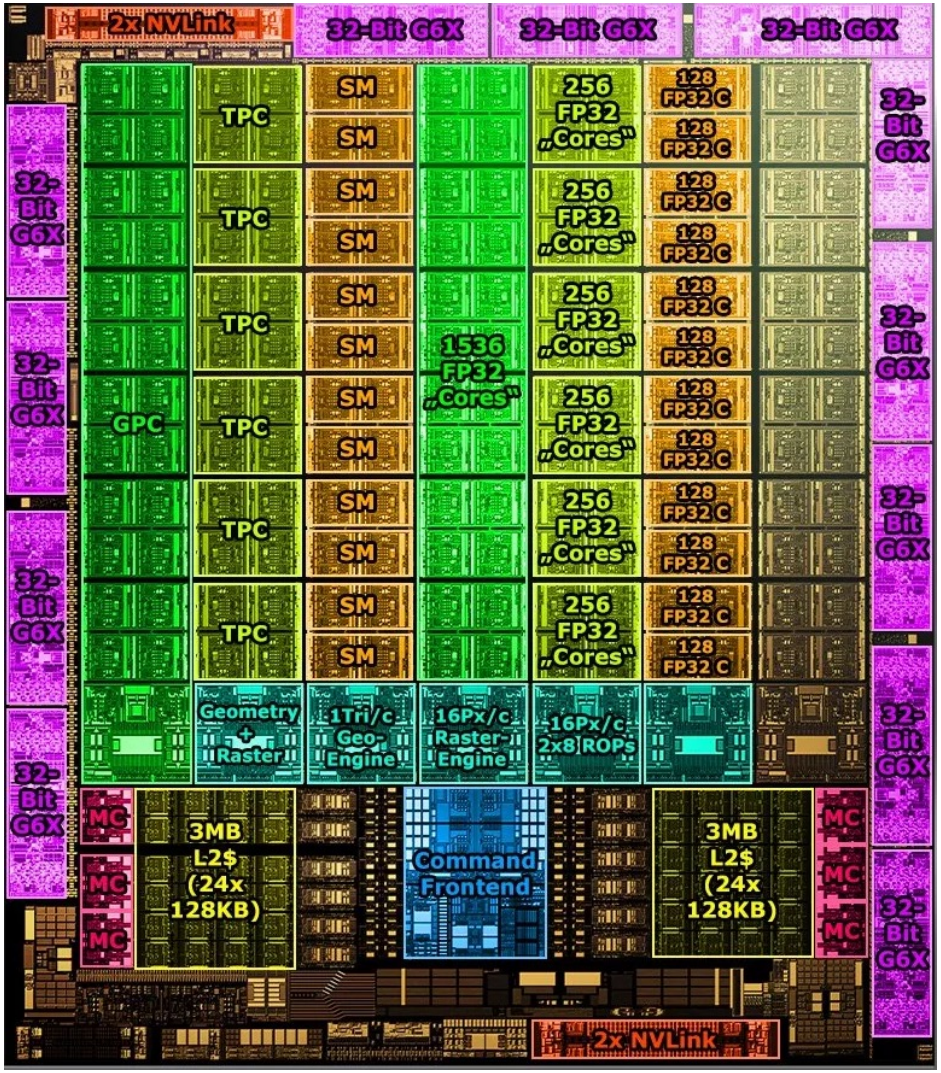
2 MB per core

L3 Cache:

120 MB shared

https://www.it-techblog.de/ibm-power10-prozessor-mehr-speicher-mehr-tempo-mehr-sicherheit/09/2020/

Deeper and Larger Cache Hierarchies



Nvidia Ampere, 2020

Cores:

128 Streaming Multiprocessors

L1 Cache or Scratchpad:

192KB per SM

Can be used as L1 Cache and/or Scratchpad

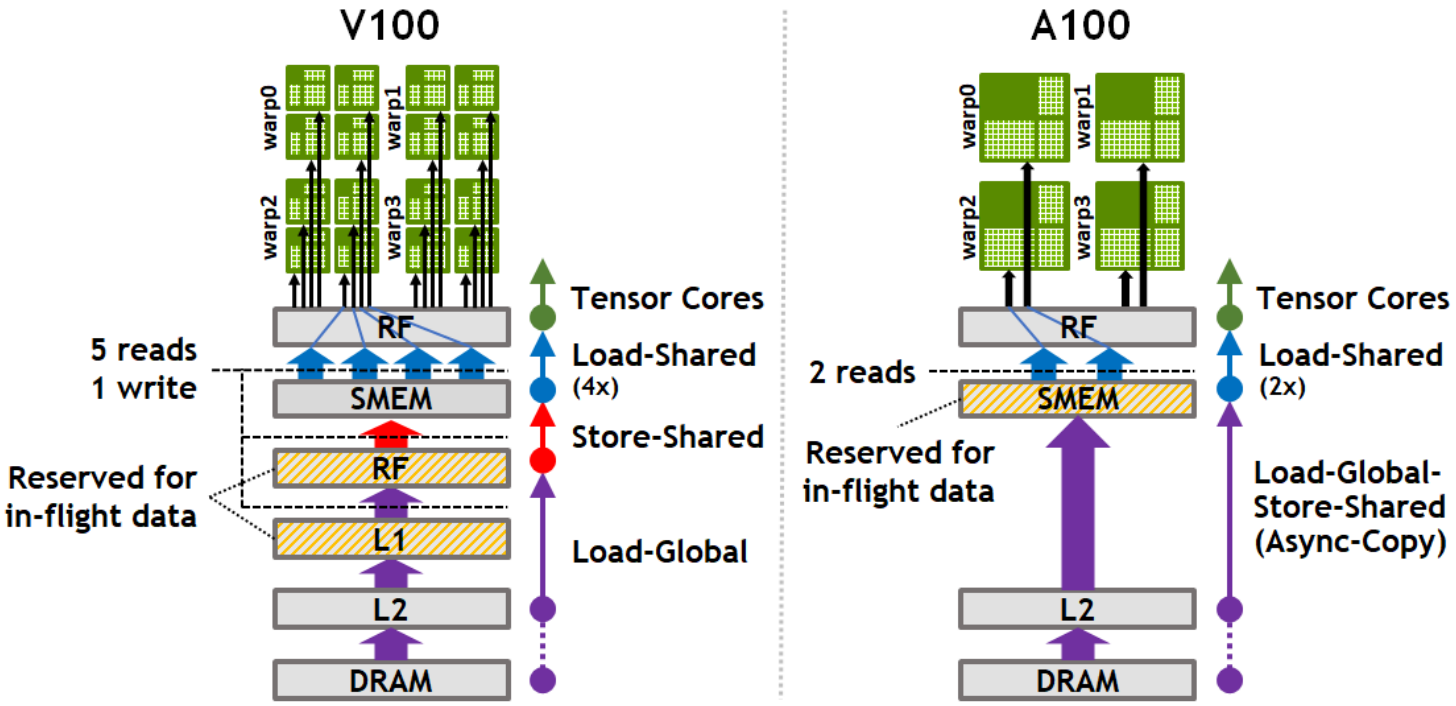
L2 Cache:

40 MB shared

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

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.

https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

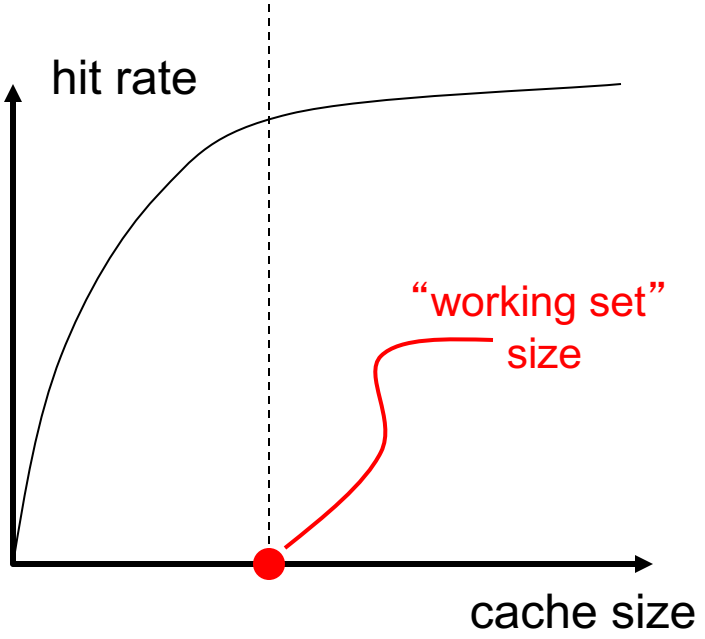
Cache Performance

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

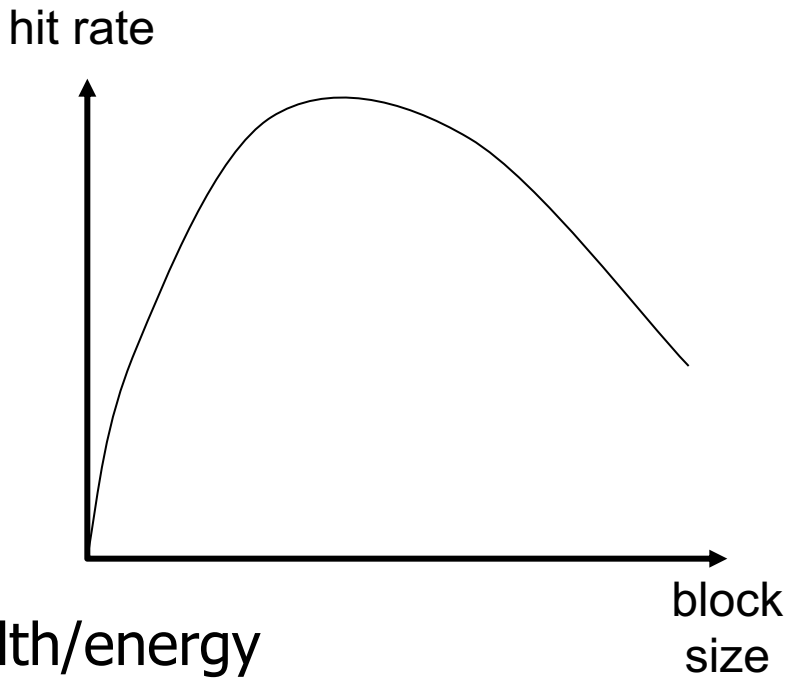


Block Size

- 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

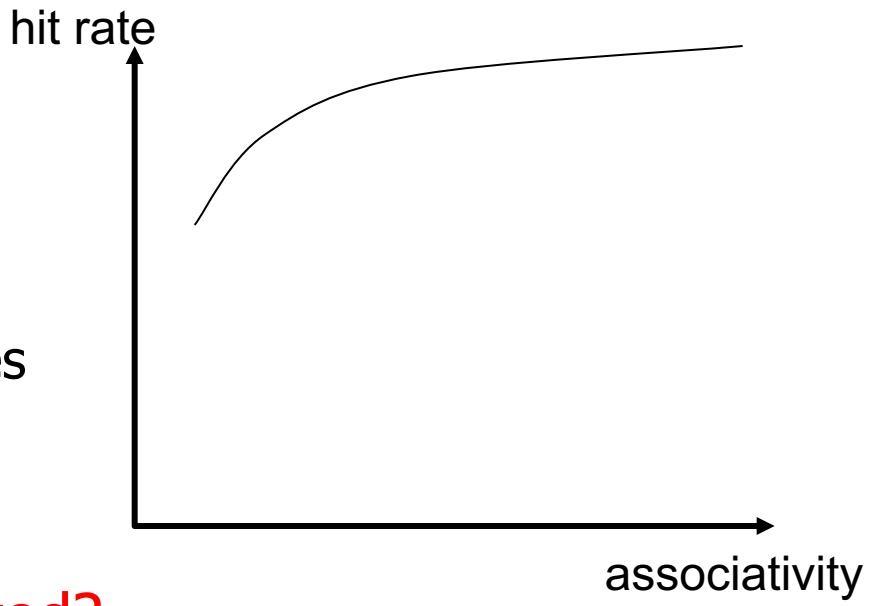
- Large cache blocks can waste bus bandwidth
  - divide a block into subblocks
  - associate separate valid and dirty bits for each subblock
  - Recall: When is this useful?

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

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 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

Restructuring Data Layout (I)

```
struct Node {
    struct Node* next;
    int key;
    char [256] name;
    char [256] school;
}

while (node) {
    if (node→key == input-key) {
        // access other fields of node
    }
    node = node→next;
}
```

- Pointer based traversal (e.g., of a linked list)
- Assume a huge linked list (1B nodes) and unique keys

- Why does the code on the left have poor cache hit rate?
  - "Other fields" occupy most of the cache line even though rarely accessed!

Restructuring Data Layout (II)

```
struct Node {
    struct Node* next;
    int key;
    struct Node-data* node-data;
}

struct Node-data {
    char [256] name;
    char [256] school;
}

while (node) {
    if (node→key == input-key) {
        // access node→node-data
    }
    node = node→next;
}
```

- Idea: separate frequently-used fields of a data structure and pack them into a separate data structure
- Who should do this?
  - Programmer
  - Compiler
    - Profiling vs. dynamic
  - Hardware?
  - Who can determine what is frequently used?

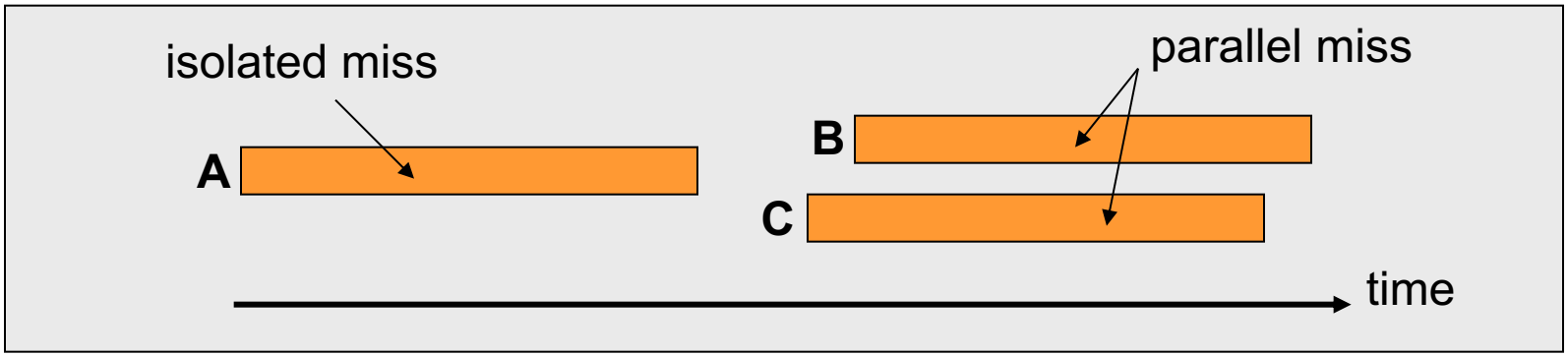
Improving Basic Cache Performance

- 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

Miss Latency/Cost

- What is miss latency or miss cost affected by?
  - Where does the miss get serviced from?
    - Local vs. remote memory
    - What level of cache in the hierarchy?
    - Row hit versus row miss
    - Queueing delays in the memory controller and the interconnect
    - …
  - How much does the miss stall the processor?
    - Is it overlapped with other latencies?
    - Is the data immediately needed?
    - …

Memory Level Parallelism (MLP)



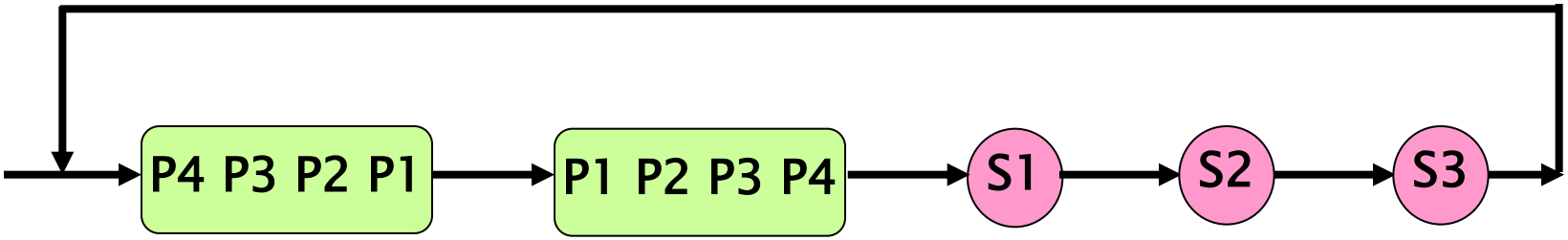
- Memory Level Parallelism (MLP) means generating and servicing multiple memory accesses in parallel [Glew' 98]
- Several techniques to improve MLP (e.g., out-of-order execution)
- MLP varies. Some misses are isolated and some parallel

How does this affect cache replacement?

Traditional Cache Replacement Policies

- Traditional cache replacement policies try to reduce miss count
- Implicit assumption: Reducing miss count reduces memory-related stall time
- Misses with varying cost/MLP breaks 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



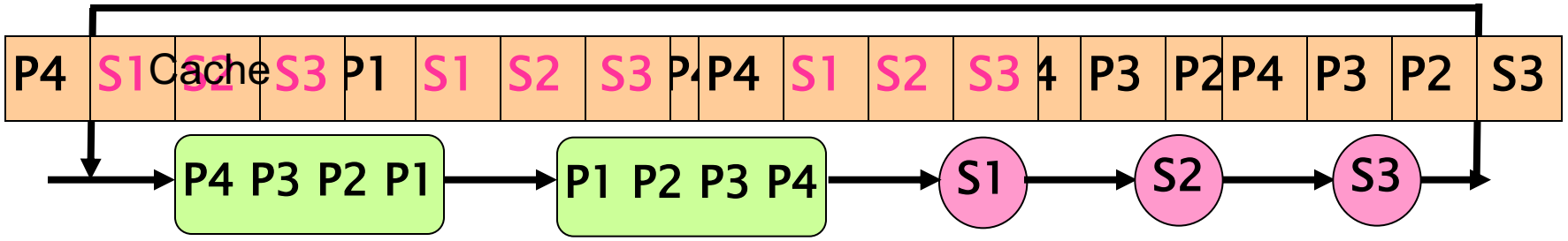
Misses to blocks P1, P2, P3, P4 can be parallel
Misses to blocks S1, S2, and S3 are isolated

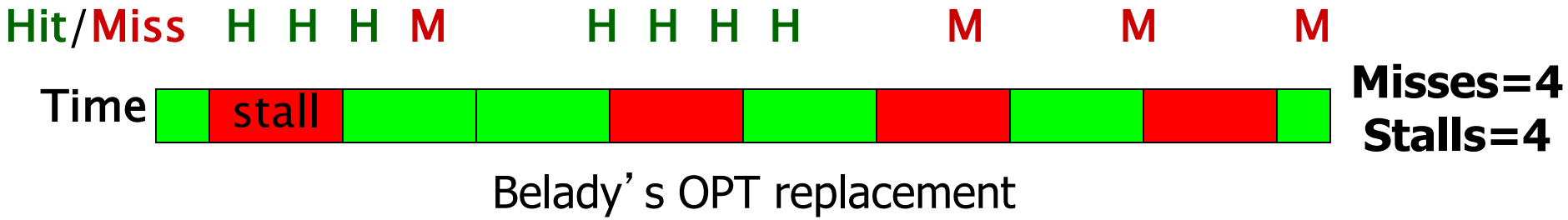
Two replacement algorithms:

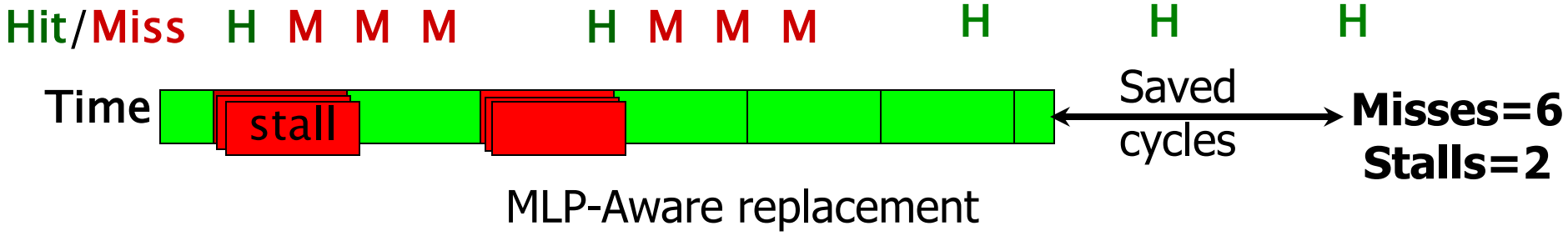
1. Minimizes miss count (Belady's OPT)
2. Reduces isolated miss (MLP-Aware)

For a fully associative cache containing 4 blocks

Fewest Misses ≠ Best Performance







Recommended: MLP-Aware Cache Replacement

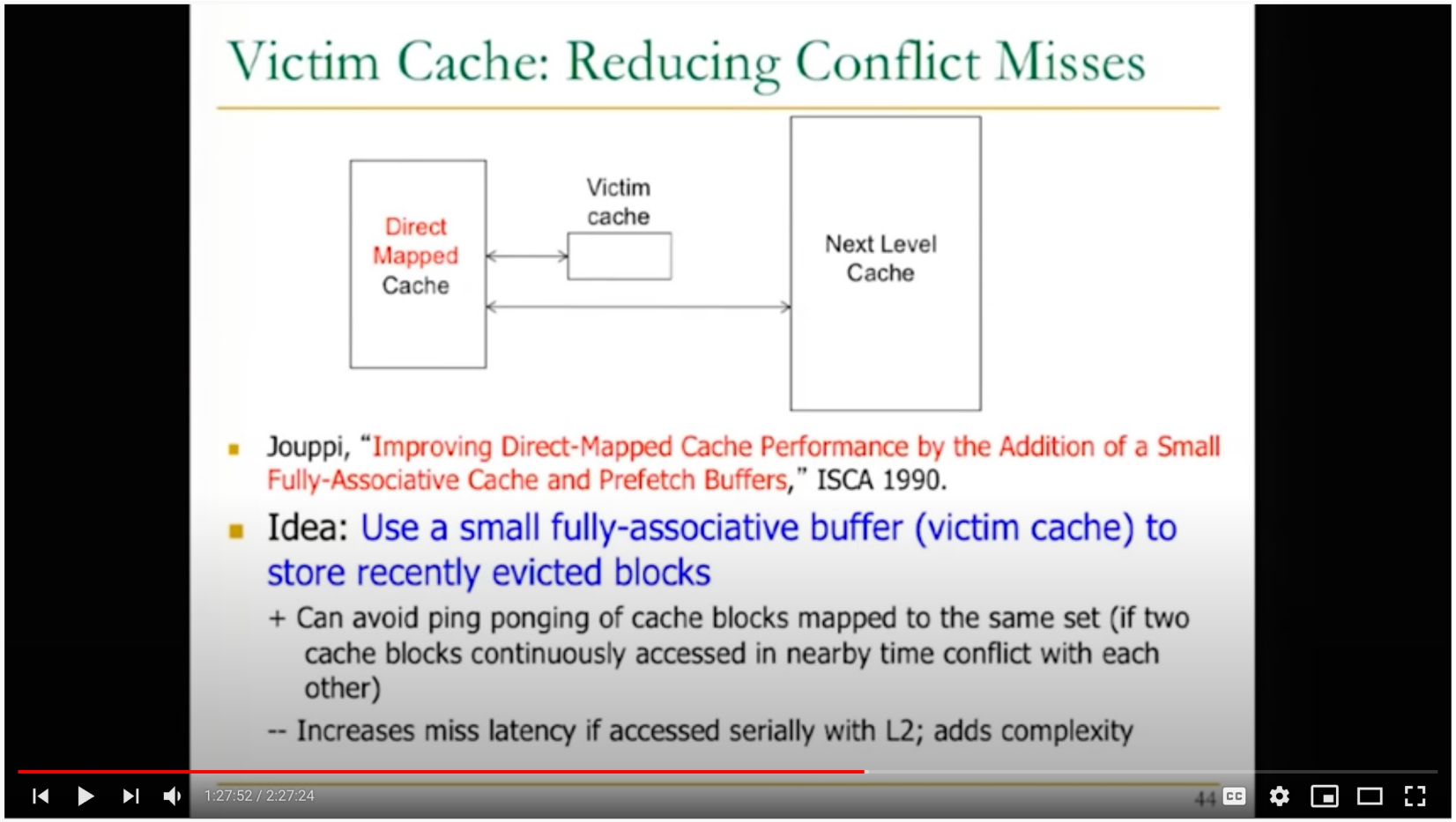
- How do we incorporate MLP into replacement decisions?

- Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.

A Case for MLP-Aware Cache Replacement

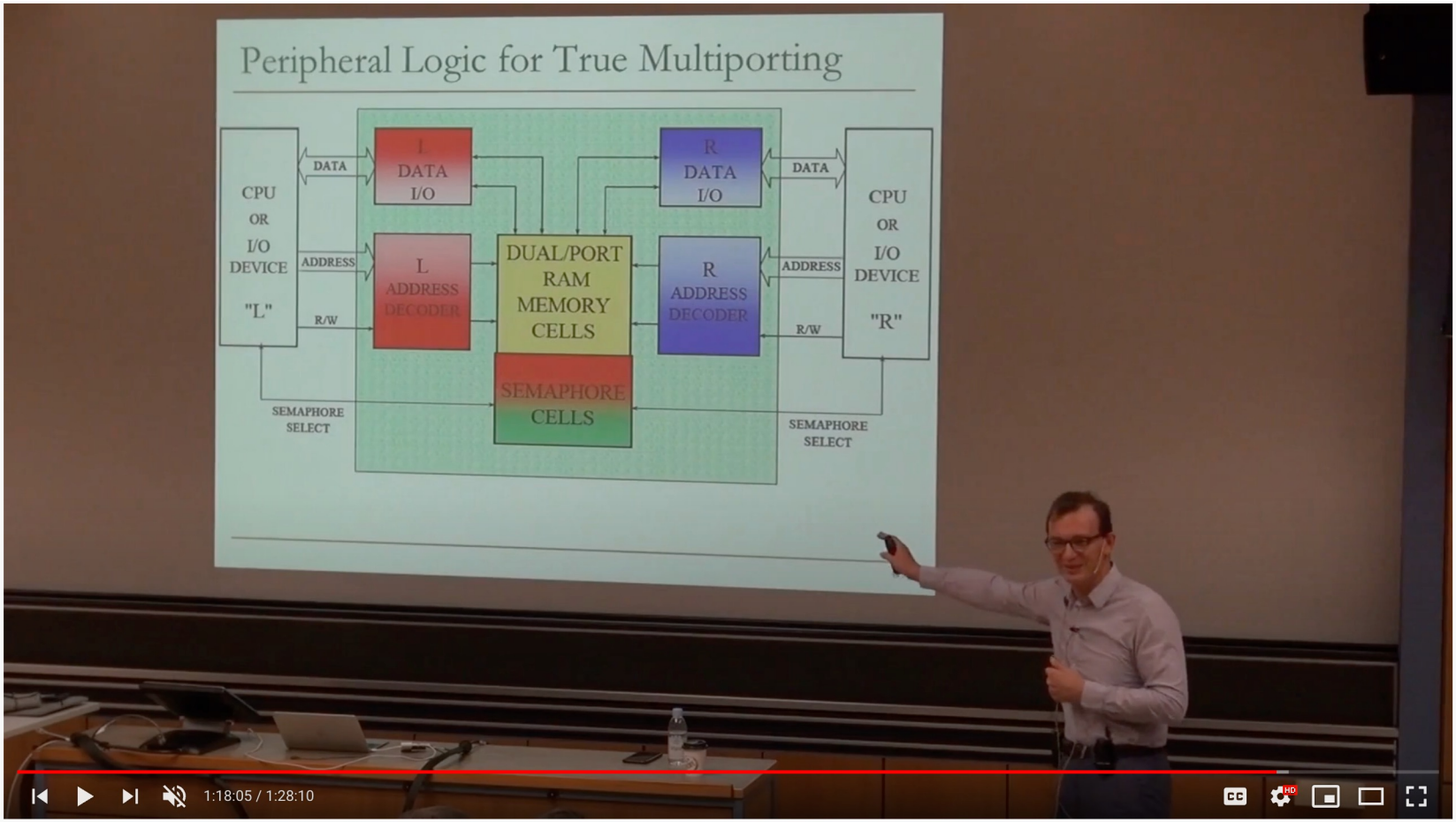
Moinuddin K. Qureshi Daniel N. Lynch Onur Mutlu Yale N. Patt
Department of Electrical and Computer Engineering
The University of Texas at Austin
{*moin, lynch, onur, patt*}*@hps.utexas.edu*

Lectures on Cache Optimizations



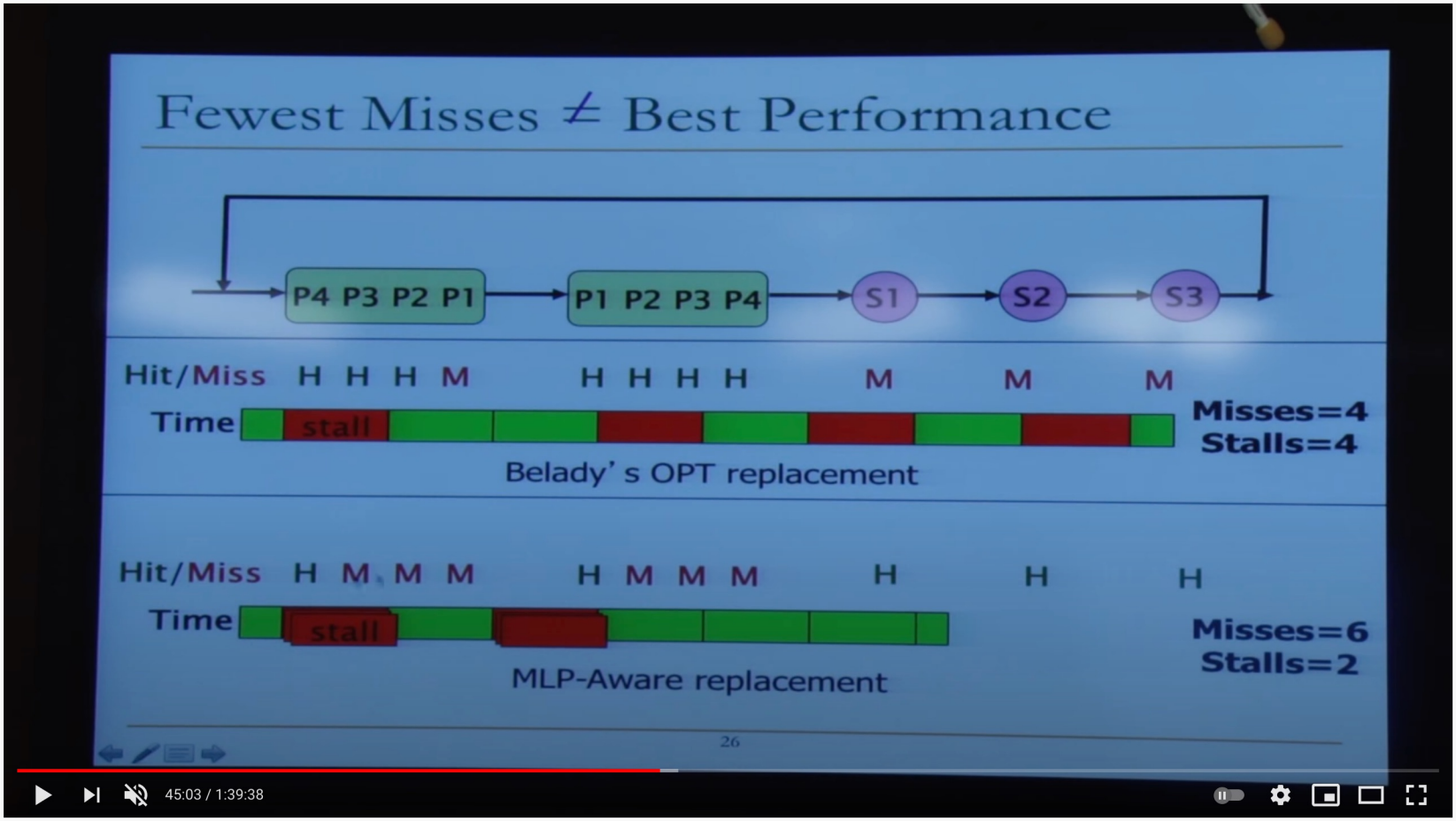
https://www.youtube.com/watch?v=OyomXCHNJDA&list=PL5Q2soXY2Zi9OhoVQBXYFIZywZXCPl4M_&index=3

Lectures on Cache Optimizations



https://www.youtube.com/watch?v=55oYBm9cifI&list=PL5Q2soXY2Zi9JXe3ywQMhylk_d5dI-TM7&index=6

Lectures on Cache Optimizations



Lecture 19. High Performance Caches - Carnegie Mellon - Comp. Arch. 2015 - Onur Mutlu

9,737 views • Mar 5, 2015 63 1 SHARE SAVE

Carnegie Mellon Computer Architecture
23.2K subscribers

ANALYTICS EDIT VIDEO

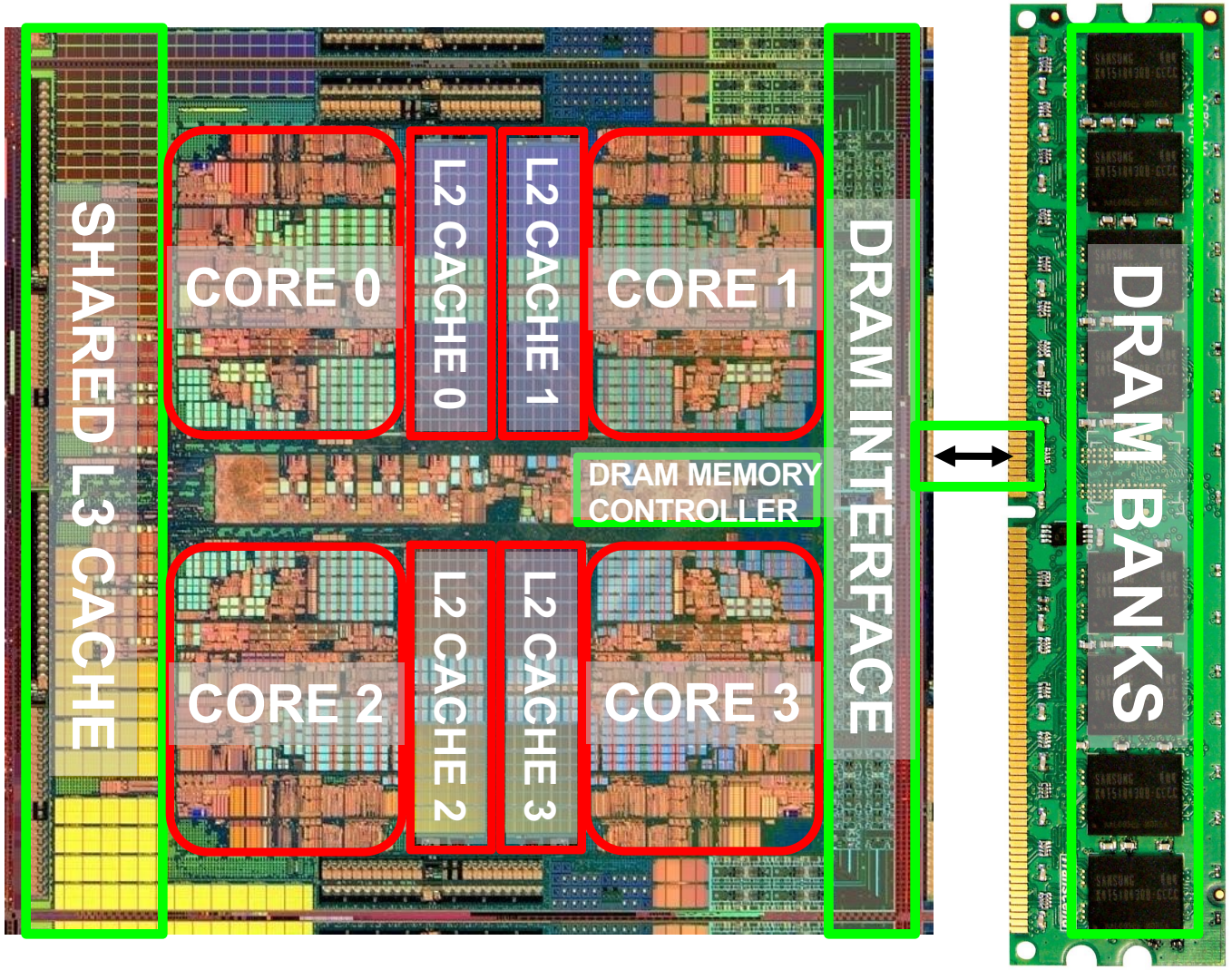
https://www.youtube.com/watch?v=jDHx2K9HxlM&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=21

Lecture on Cache Optimizations

- Computer Architecture, Fall 2017, Lecture 3
  - Cache Management & Memory Parallelism (ETH, Fall 2017)
  - https://www.youtube.com/watch?v=OyomXCHNJDA&list=PL5Q2soXY2Zi9OhoVQBXYFIZywZXCPl4M_&index=3
- Computer Architecture, Fall 2018, Lecture 4a
  - Cache Design (ETH, Fall 2018)
  - https://www.youtube.com/watch?v=55oYBm9cifI&list=PL5Q2soXY2Zi9JXe3ywQMhylk_d5dI-TM7&index=6
- Computer Architecture, Spring 2015, Lecture 19
  - High Performance Caches (CMU, Spring 2015)
  - https://www.youtube.com/watch?v=jDHx2K9HxlM&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=21

Multi-Core Issues in Caching

Caches in a Multi-Core System



Caches in Multi-Core Systems

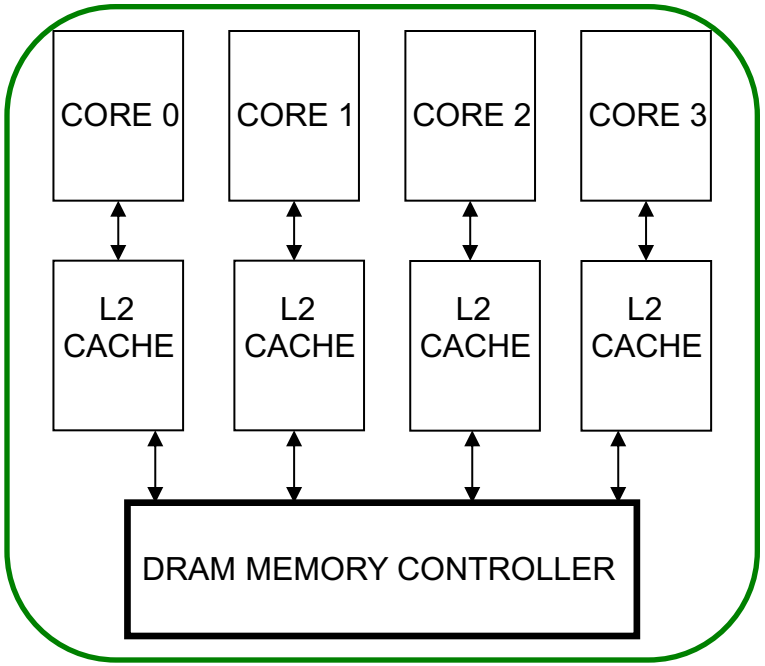
- Cache efficiency becomes even more important in a multi-core/multi-threaded system
  - Memory bandwidth is at premium
  - Cache space is a limited resource across cores/threads

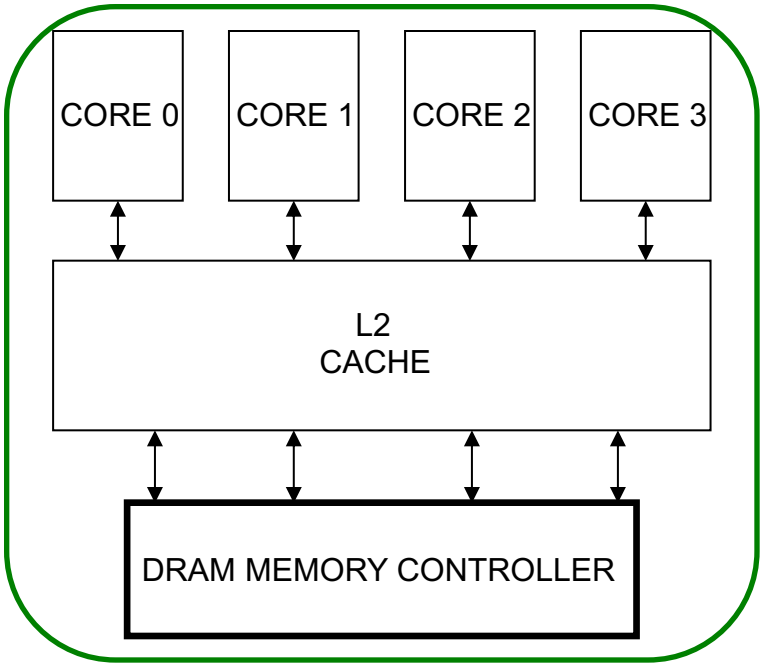
- How do we design the caches in a multi-core system?

- Many decisions
  - Shared vs. private caches
  - How to maximize performance of the entire system?
  - How to provide QoS to different threads in a shared cache?
  - Should cache management algorithms be aware of threads?
  - How should space be allocated to threads in a shared cache?

Private vs. Shared Caches

- Private cache: Cache belongs to one core (a shared block can be in multiple caches)
- Shared cache: Cache is shared by multiple cores





Resource Sharing Concept and Advantages

- Idea: Instead of dedicating a hardware resource to a hardware context, allow multiple contexts to use it
  - Example resources: functional units, pipeline, caches, buses, memory
- Why?

+ Resource sharing improves utilization/efficiency → throughput
  - When a resource is left idle by one thread, another thread can use it; no need to replicate shared data
+ Reduces communication latency
  - For example, data shared between multiple threads can be kept in the same cache in multithreaded processors
+ Compatible with the shared memory programming model

Resource Sharing Disadvantages

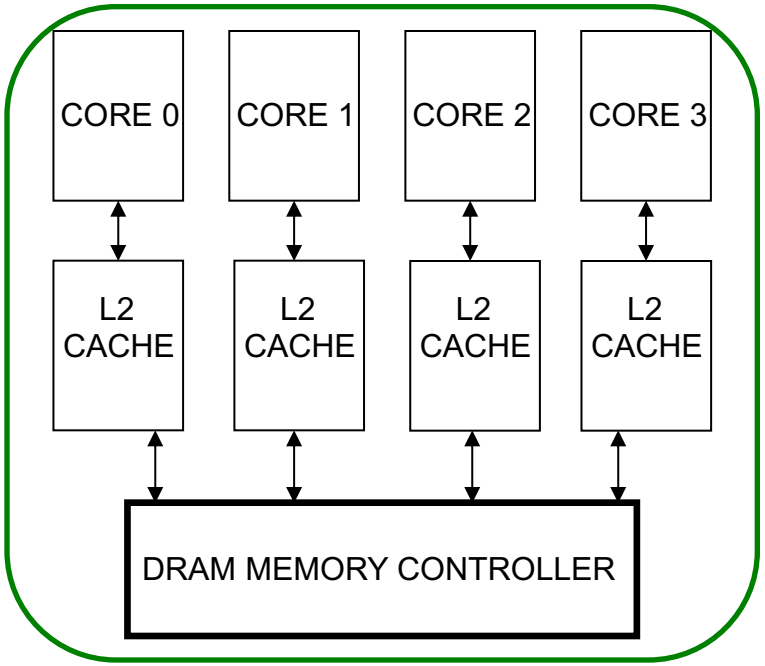
- Resource sharing results in contention for resources
  - When the resource is not idle, another thread cannot use it
  - If space is occupied by one thread, another thread needs to re-occupy it

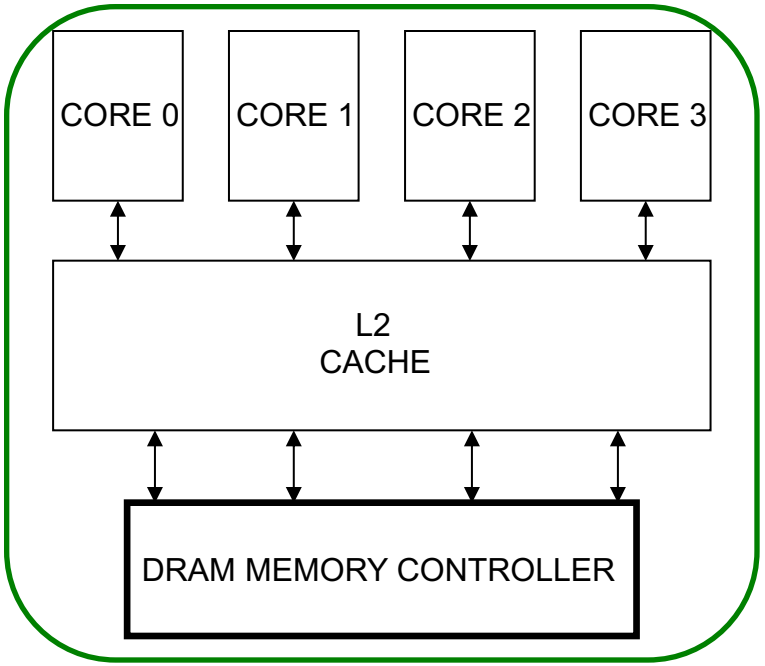
- Sometimes reduces each or some thread's performance
  - Thread performance can be worse than when it is run alone
- Eliminates performance isolation → inconsistent performance across runs
  - Thread performance depends on co-executing threads
- Uncontrolled (free-for-all) sharing degrades QoS
  - Causes unfairness, starvation

Need to efficiently and fairly utilize shared resources

Private vs. Shared Caches

- Private cache: Cache belongs to one core (a shared block can be in multiple caches)
- Shared cache: Cache is shared by multiple cores





Shared Caches Between Cores

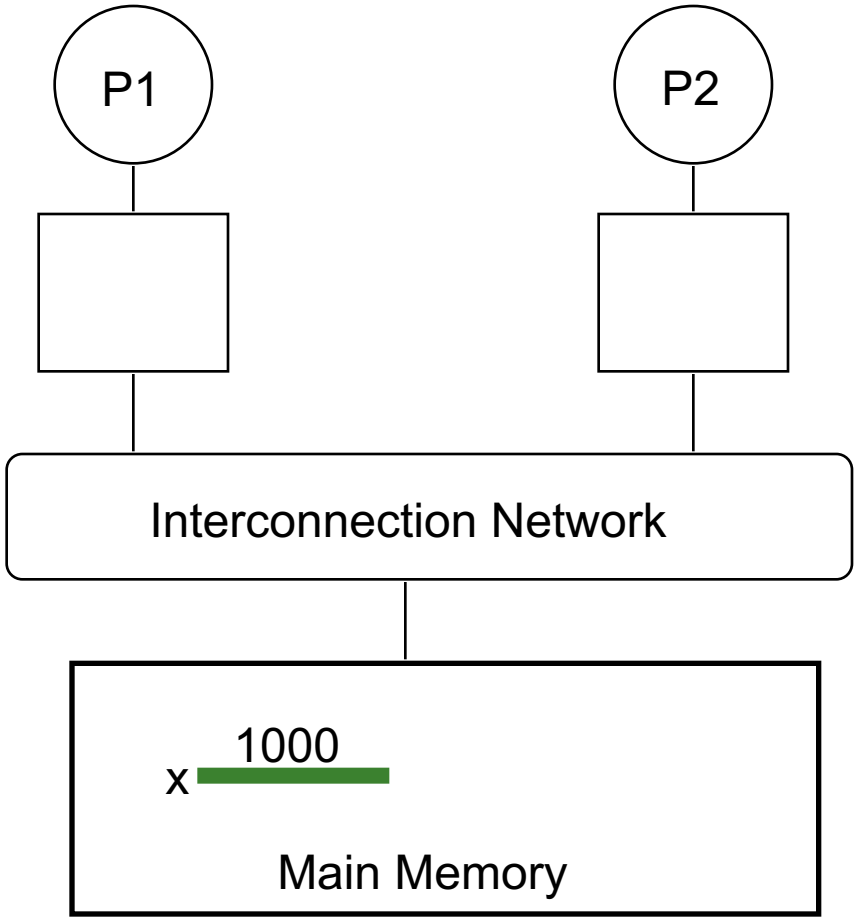
- Advantages:
  - High effective capacity
  - Dynamic partitioning of available cache space
    - No fragmentation due to static partitioning
    - If one core does not utilize some space, another core can
  - Easier to maintain coherence (a cache block is in a single location)

- Disadvantages
  - Slower access (cache not tightly coupled with the core)
  - Cores incur conflict misses due to other cores' accesses
    - Misses due to inter-core interference
    - Some cores can destroy the hit rate of other cores
  - Guaranteeing a minimum level of service (or fairness) to each core is harder (how much space, how much bandwidth?)

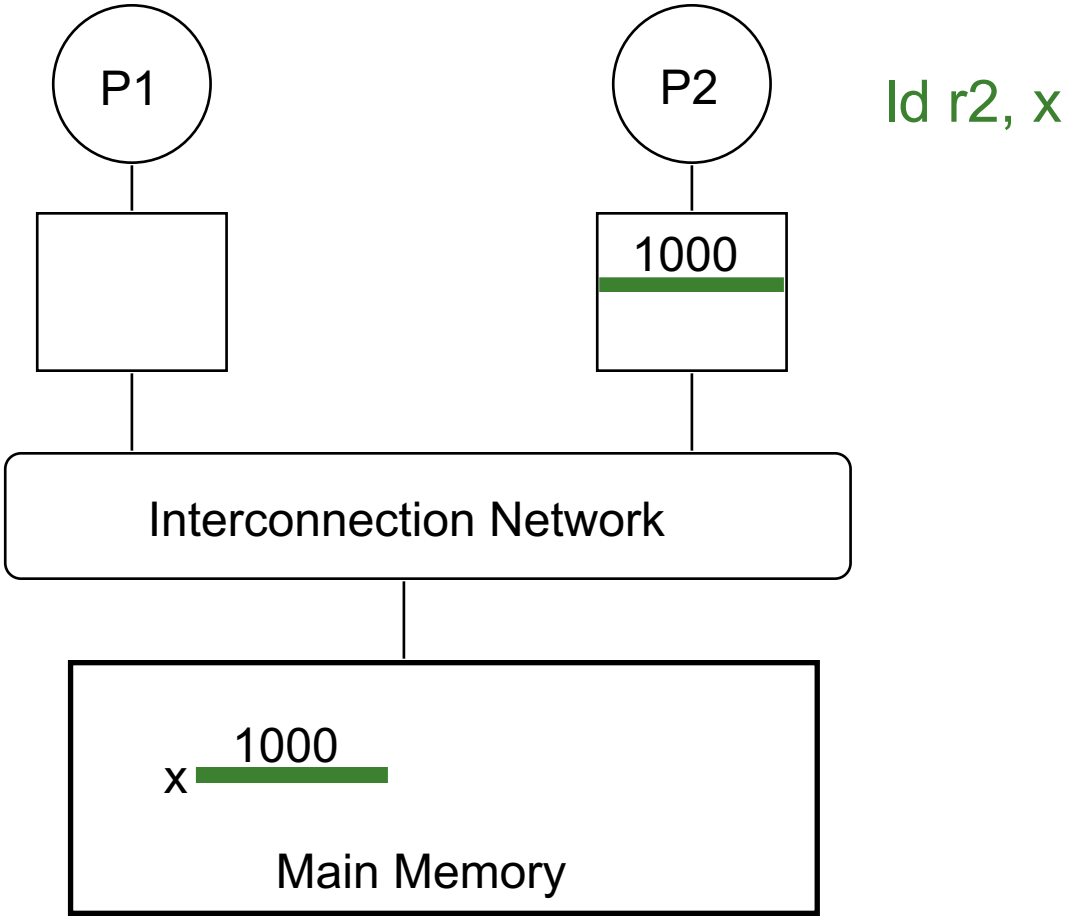
Cache Coherence

Cache Coherence

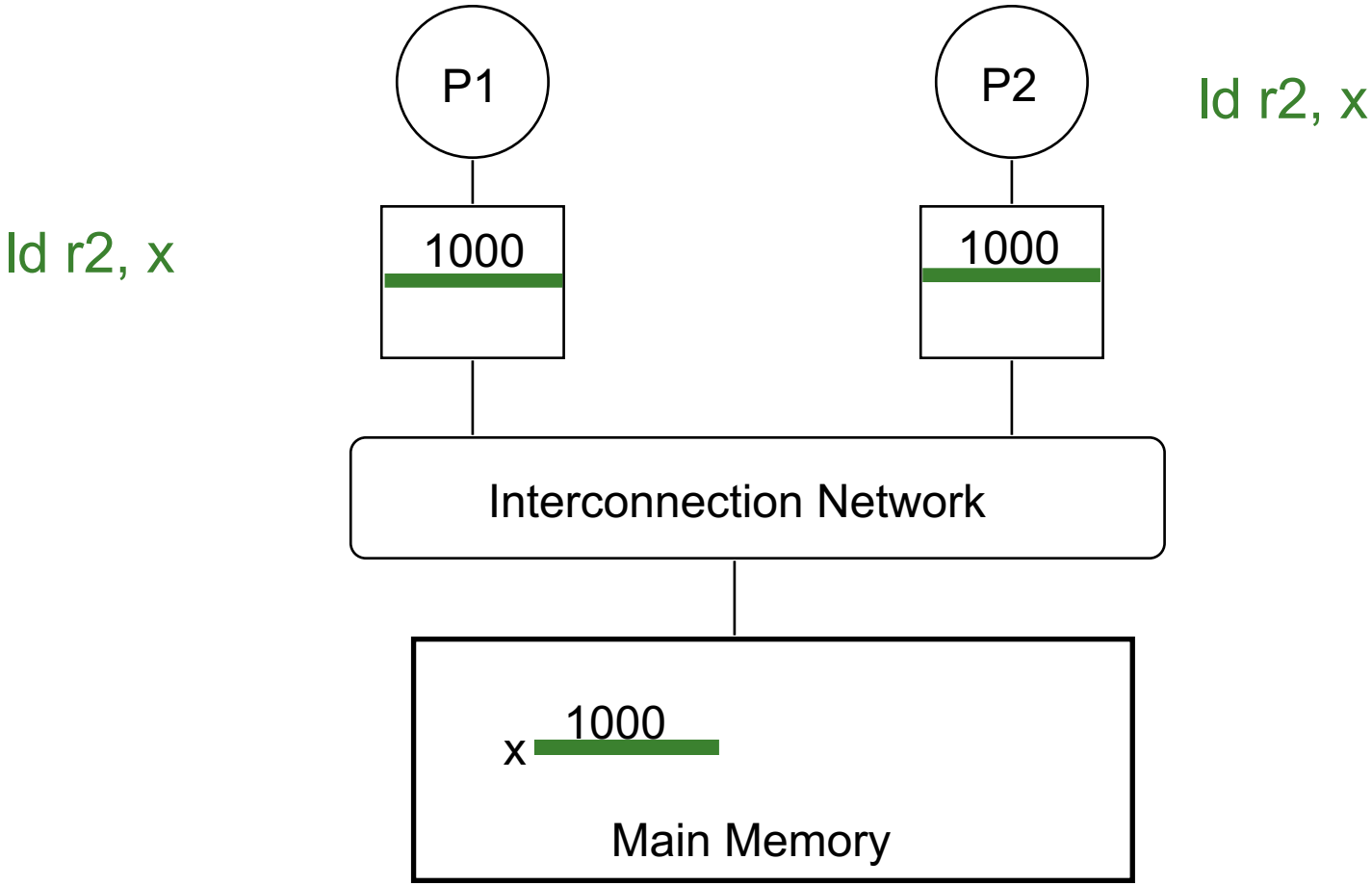
- Basic question: If multiple processors cache the same block, how do they ensure they all see a consistent state?



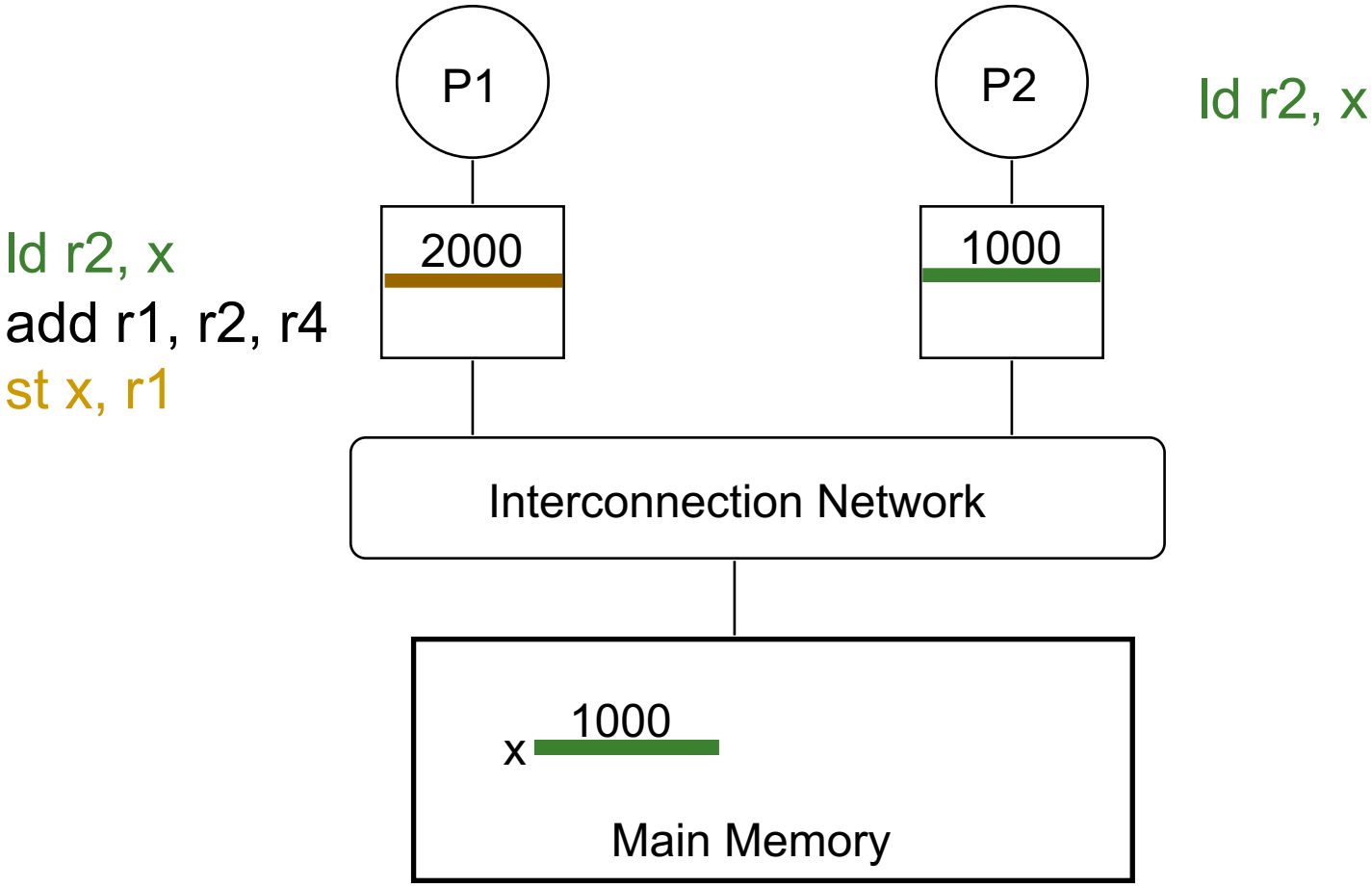
The Cache Coherence Problem



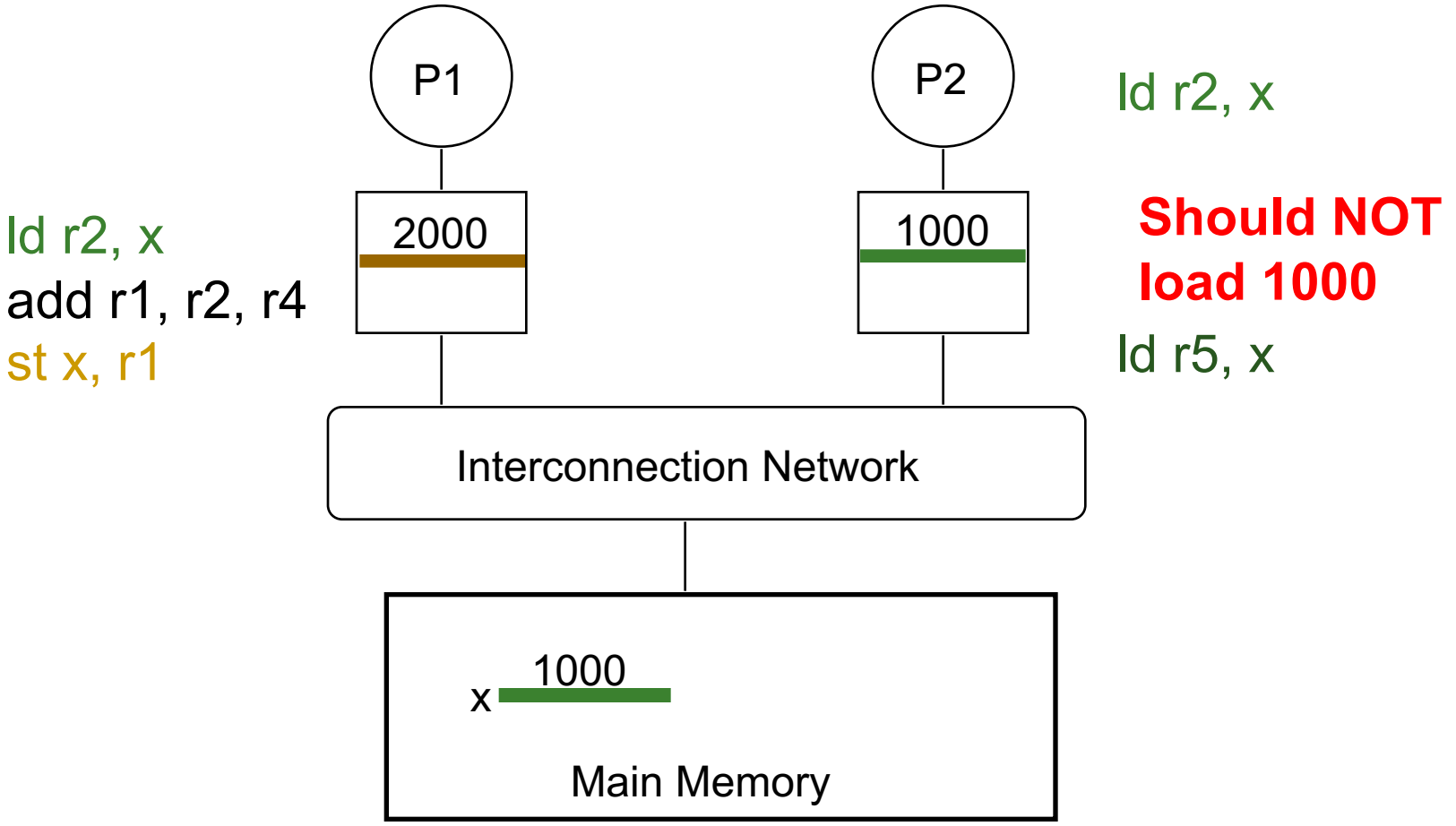
The Cache Coherence Problem



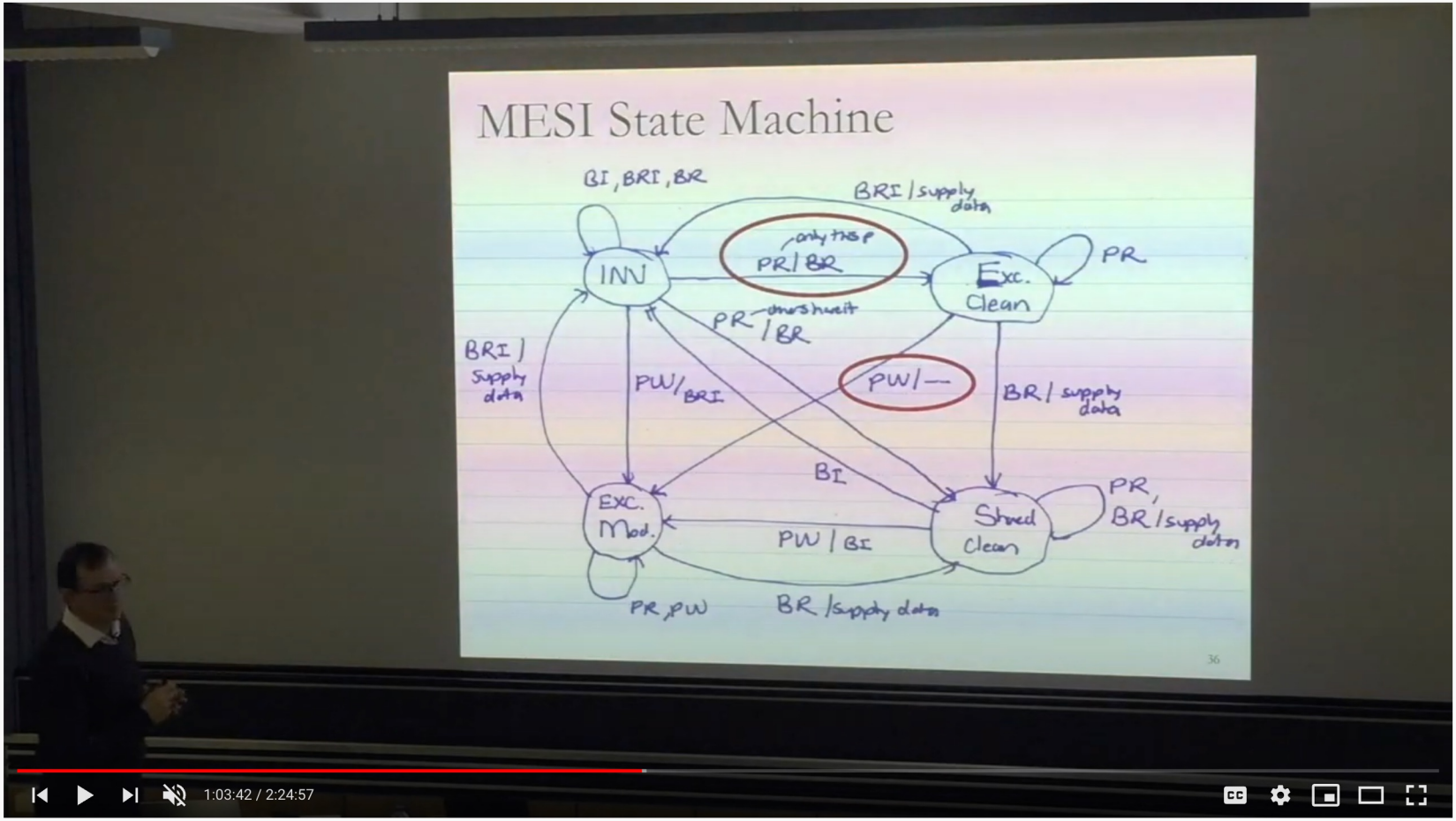
The Cache Coherence Problem



The Cache Coherence Problem



Lecture on Cache Coherence



https://www.youtube.com/watch?v=T9WlyezeaII&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=38

Lecture on Cache Coherence

- Computer Architecture, Fall 2020, Lecture 21
  - Cache Coherence (ETH, Fall 2020)
  - https://www.youtube.com/watch?v=T9WlyezeaII&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=38

Basic Cache Examples: For You to Study

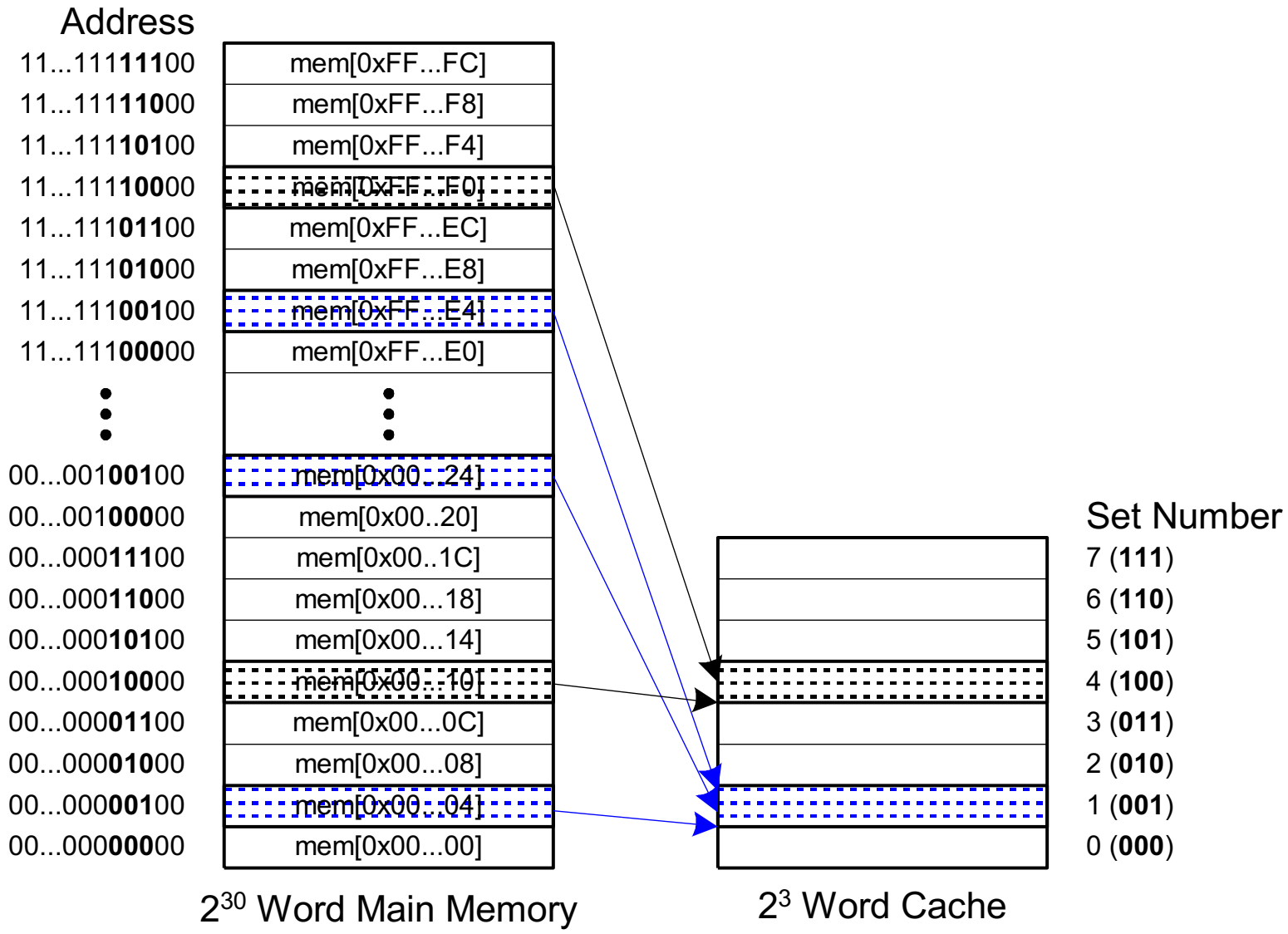
Cache Terminology

- Capacity (*C*):
  - the number of data bytes a cache stores
- Block size (*b*):
  - bytes of data brought into cache at once
- Number of blocks (*B = C/b*):
  - number of blocks in cache: *B = C/b*
- Degree of associativity (*N*):
  - number of blocks in a set
- Number of sets (*S = B/N*):
  - each memory address maps to exactly one cache set

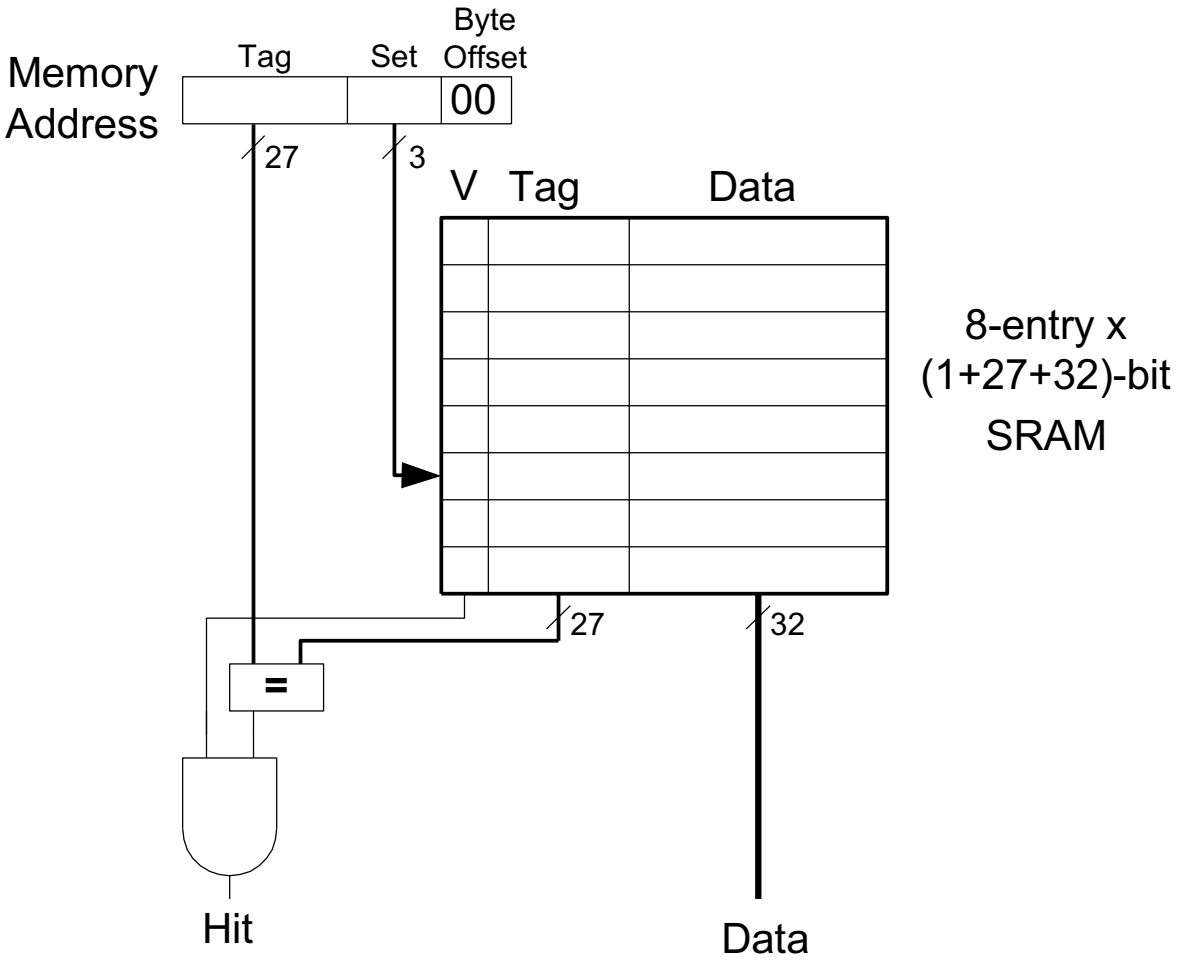
How is data found?

- Cache organized into *S* sets
- Each memory address maps to exactly one set
- Caches categorized by number of blocks in a set:
  - Direct mapped: 1 block per set
  - N-way set associative: N blocks per set
  - Fully associative: all cache blocks are in a single set
- Examine each organization for a cache with:
  - Capacity (*C* = 8 words)
  - Block size (*b* = 1 word)
  - So, number of blocks (*B* = 8)

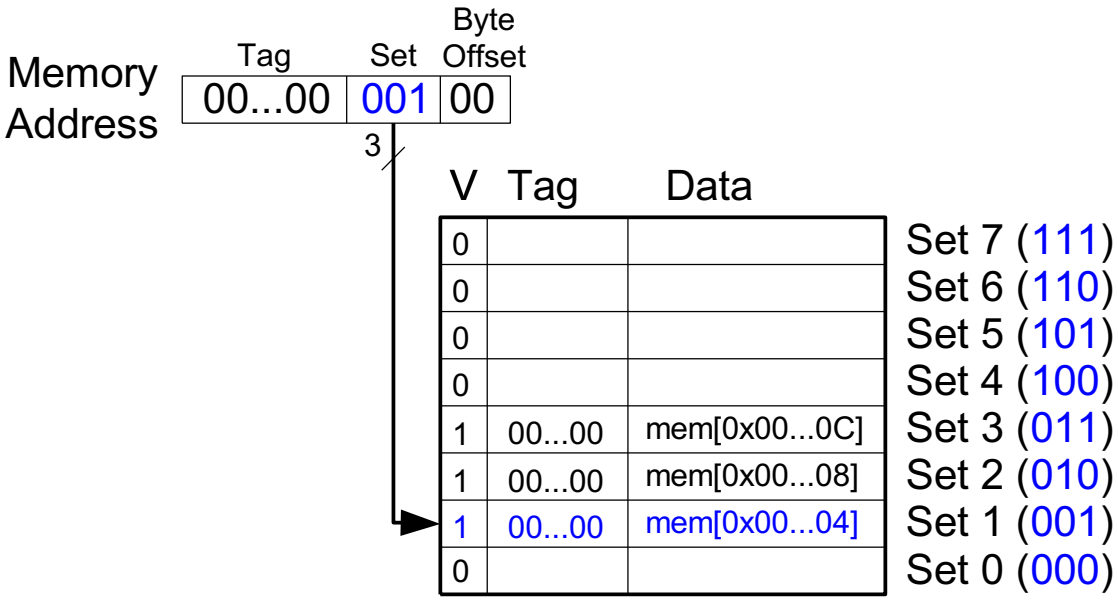
Direct Mapped Cache



Direct Mapped Cache Hardware



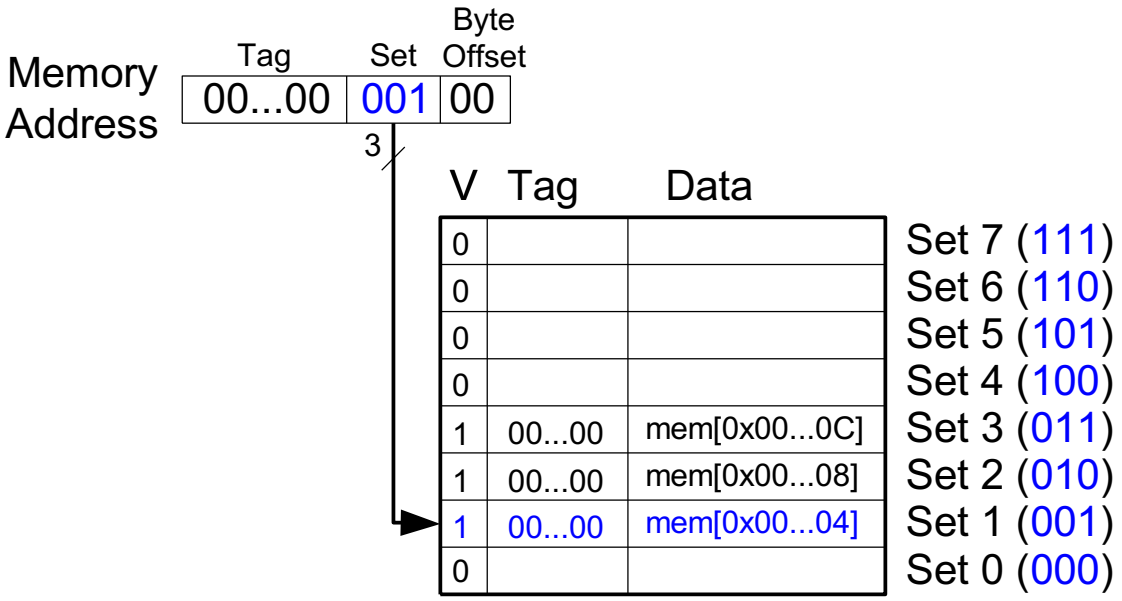
Direct Mapped Cache Performance




```
# MIPS assembly code
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0xC($0)
        lw   $t3, 0x8($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate =

Direct Mapped Cache Performance




```
# MIPS assembly code
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0xC($0)
        lw   $t3, 0x8($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate = 3/15

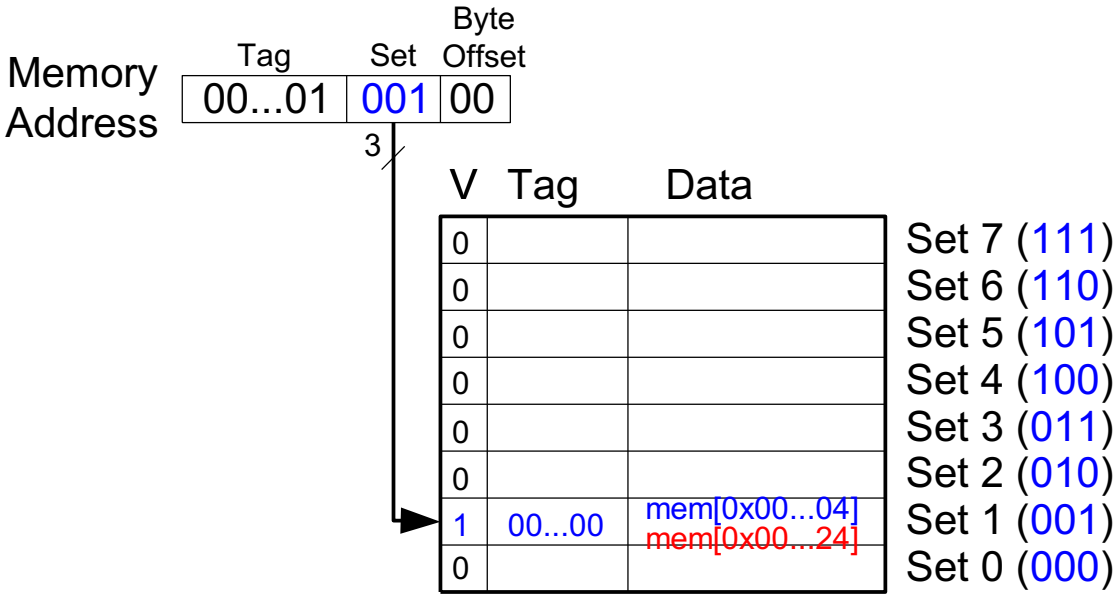
=

20%

Temporal Locality

Compulsory Misses

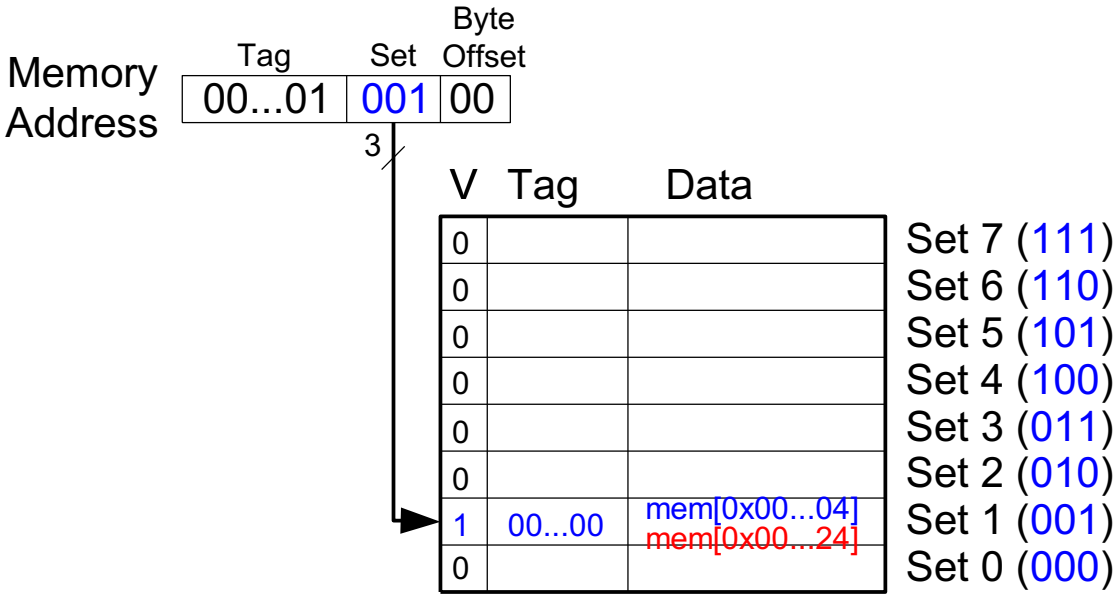
Direct Mapped Cache: Conflict




```
# MIPS assembly code
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate =

Direct Mapped Cache: Conflict

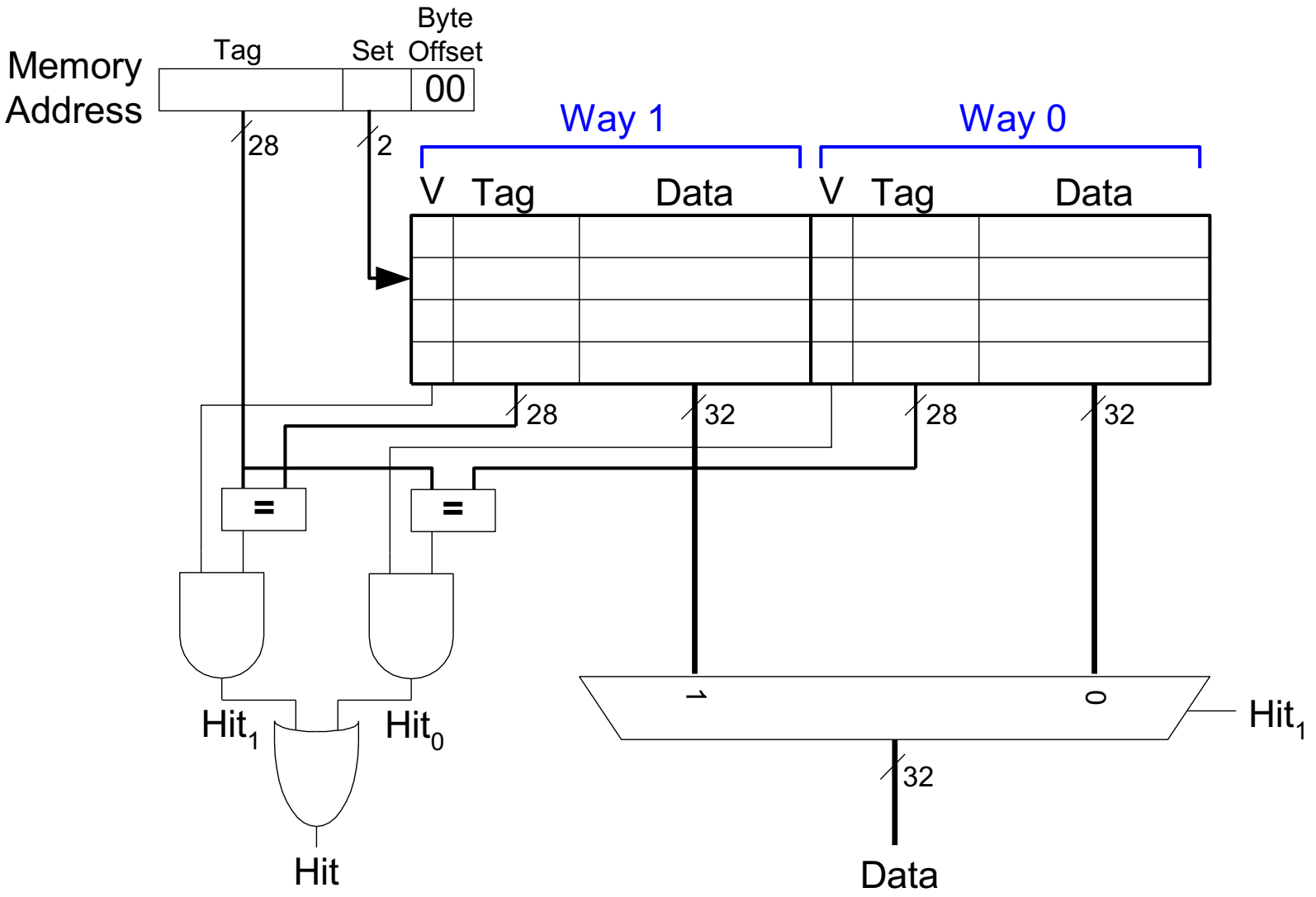



```
# MIPS assembly code
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate = 10/10
= 100%

Conflict Misses

N-Way Set Associative Cache

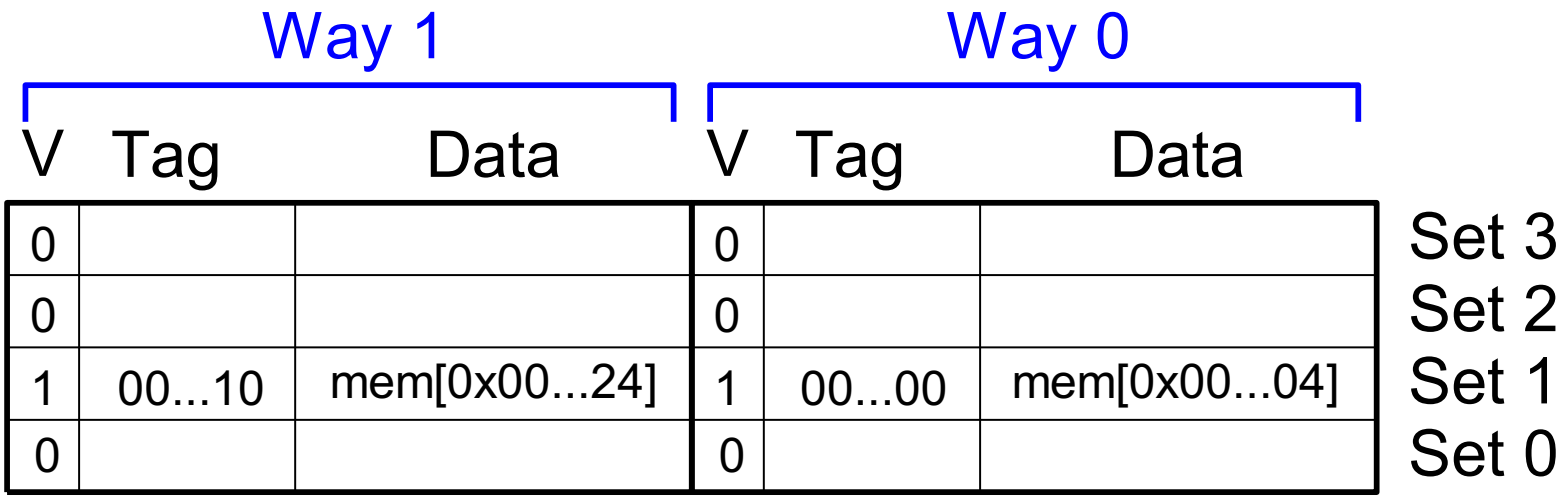


N-way Set Associative Performance

```
# MIPS assembly code

        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate =



N-way Set Associative Performance

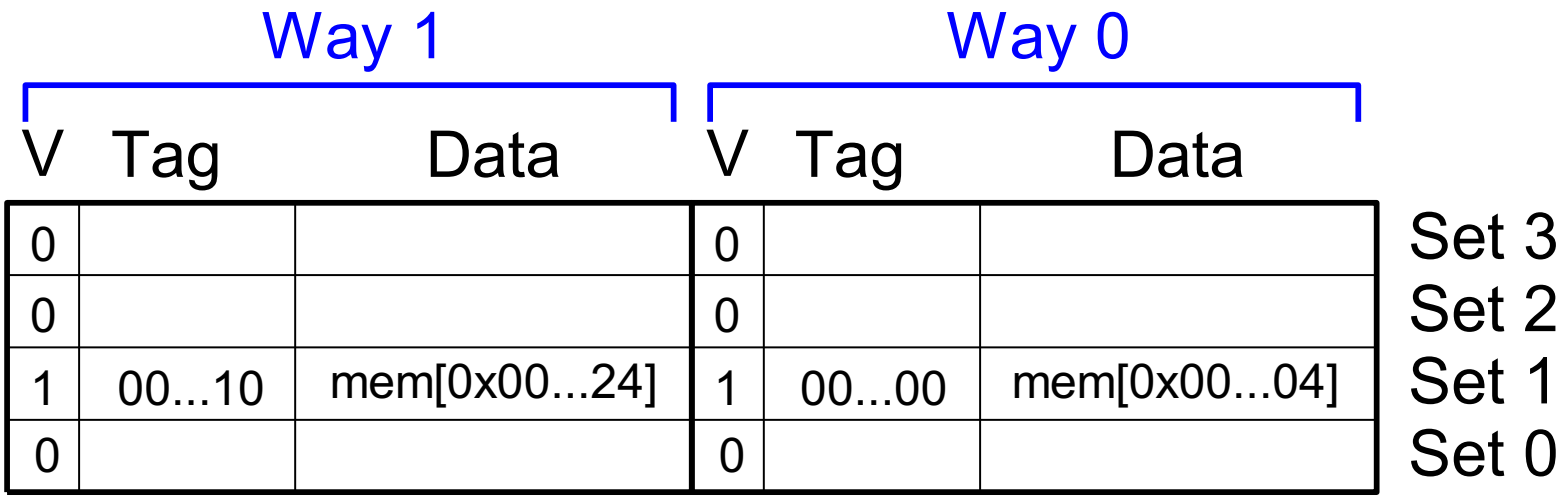
```
# MIPS assembly code

        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate = 2/10

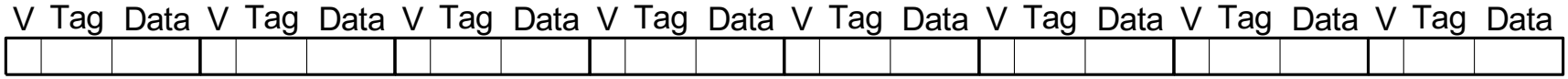
= 20%

Associativity reduces conflict misses



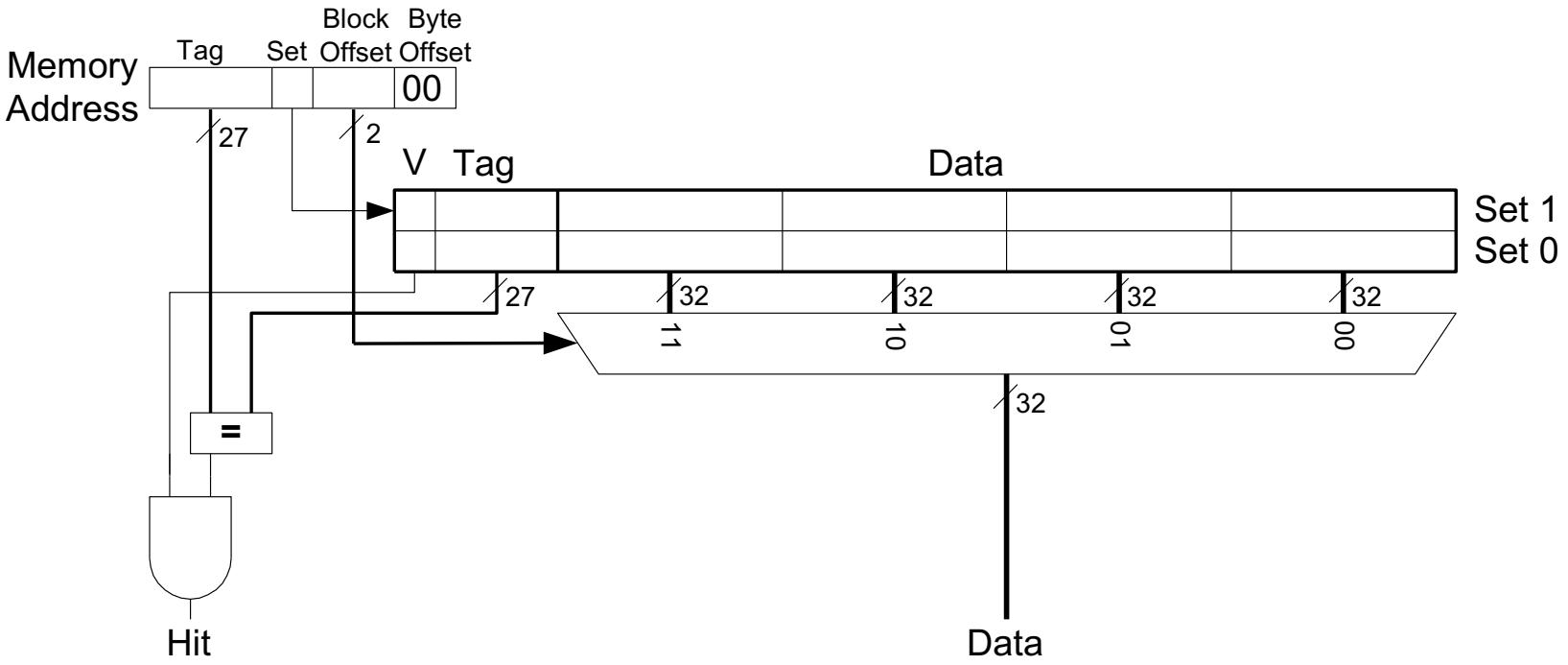
Fully Associative Cache

- No conflict misses
- Expensive to build



Spatial Locality?

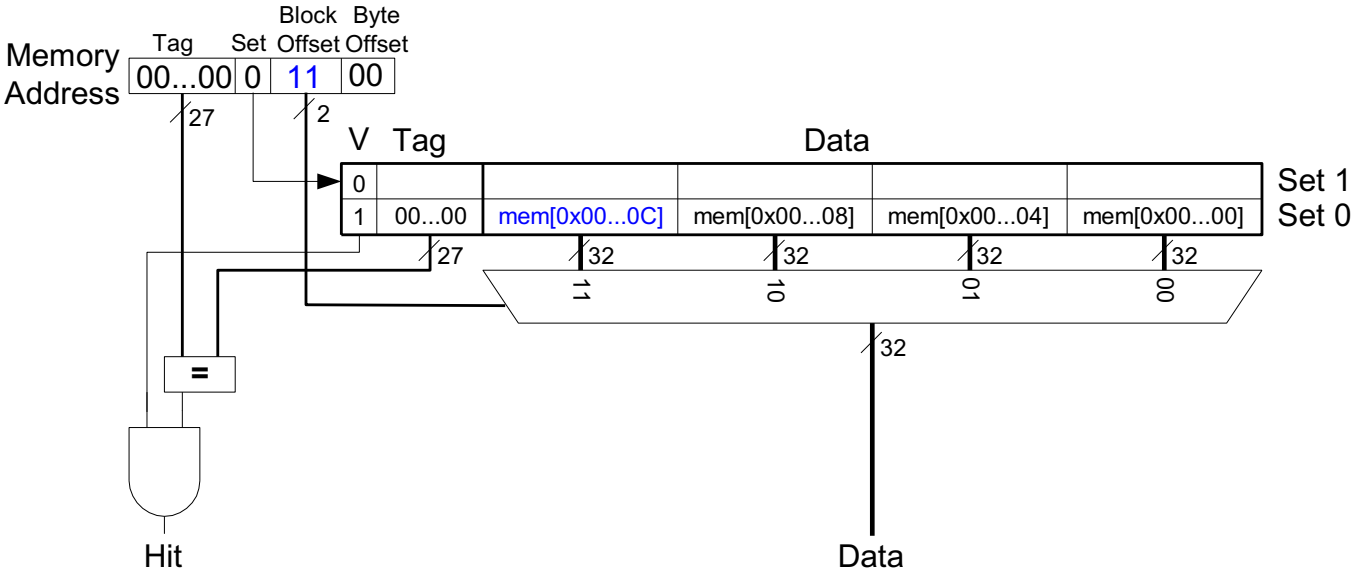
- Increase block size:
  - Block size, ***b* = 4** words
  - *C* = 8 words
  - Direct mapped (1 block per set)
  - Number of blocks, *B* = *C*/*b* = 8/4 = 2



Direct Mapped Cache Performance

```
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0xC($0)
        lw   $t3, 0x8($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate =



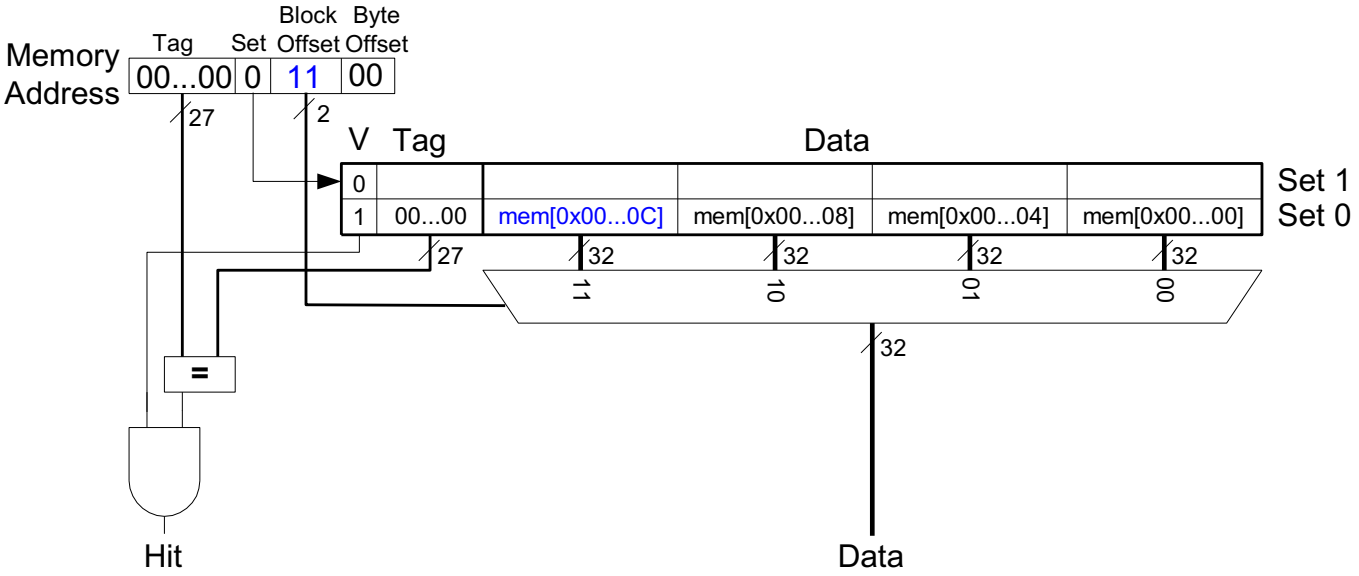
Direct Mapped Cache Performance

```
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0xC($0)
        lw   $t3, 0x8($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate = 1/15

= 6.67%

Larger blocks reduce compulsory misses through spatial locality



Cache Organization Recap

- Main Parameters
  - Capacity: ***C***
  - Block size: ***b***
  - Number of blocks in cache: ***B*** = *C*/*b*
  - Number of blocks in a set: ***N***
  - Number of Sets: ***S*** = *B*/*N*

Organization	Number of Ways (N)	Number of Sets (S = B/N)
Direct Mapped	1	B
N-Way Set Associative	1 < N < B	B / N
Fully Associative	B	1

Capacity Misses

- Cache is too small to hold all data of interest at one time
  - If the cache is full and program tries to access data X that is not in cache, cache must evict data Y to make room for X
  - **Capacity miss** occurs if program then tries to access Y again
  - X will be placed in a particular set based on its address

- In a direct mapped cache, there is only one place to put X

- In an associative cache, there are multiple ways where X could go in the set.

- How to choose Y to minimize chance of needing it again?
  - Least recently used (LRU) replacement: the least recently used block in a set is evicted when the cache is full.

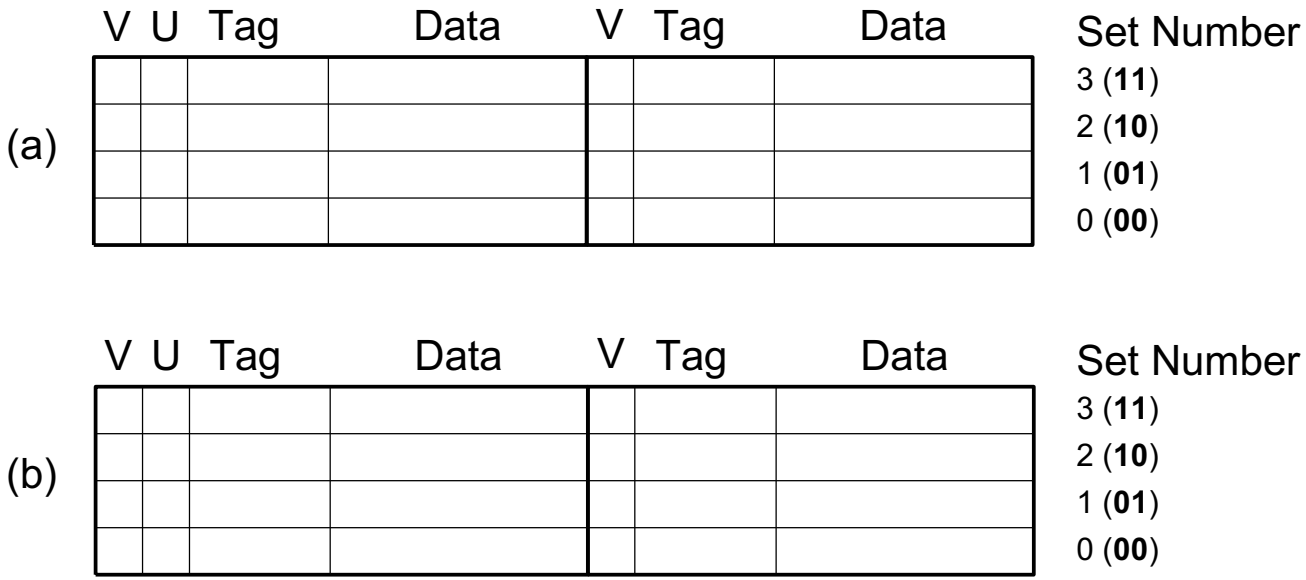
Types of Misses

- Compulsory: first time data is accessed
- Capacity: cache too small to hold all data of interest
- Conflict: data of interest maps to same location in cache
- Miss penalty: time it takes to retrieve a block from lower level of hierarchy

LRU Replacement

```
# MIPS assembly

lw $t0, 0x04($0)
lw $t1, 0x24($0)
lw $t2, 0x54($0)
```



LRU Replacement

```
# MIPS assembly

lw $t0, 0x04($0)
lw $t1, 0x24($0)
lw $t2, 0x54($0)
```