

# Digital Design & Computer Arch.

## Lecture 23: Memory Hierarchy and Caches

Prof. Onur Mutlu

ETH Zürich  
Spring 2022  
20 May 2022

# Extra Assignment 3: Amdahl's Law

---

- **Paper review**

- G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.

- **Optional Assignment – for 1% extra credit**

- **Write a 1-page review**
- Upload PDF file to Moodle – Deadline: June 15

- Strongly recommended that you **follow my guidelines for (paper) review**

# Readings for This Week and Next

---

## ■ Memory Hierarchy and Caches

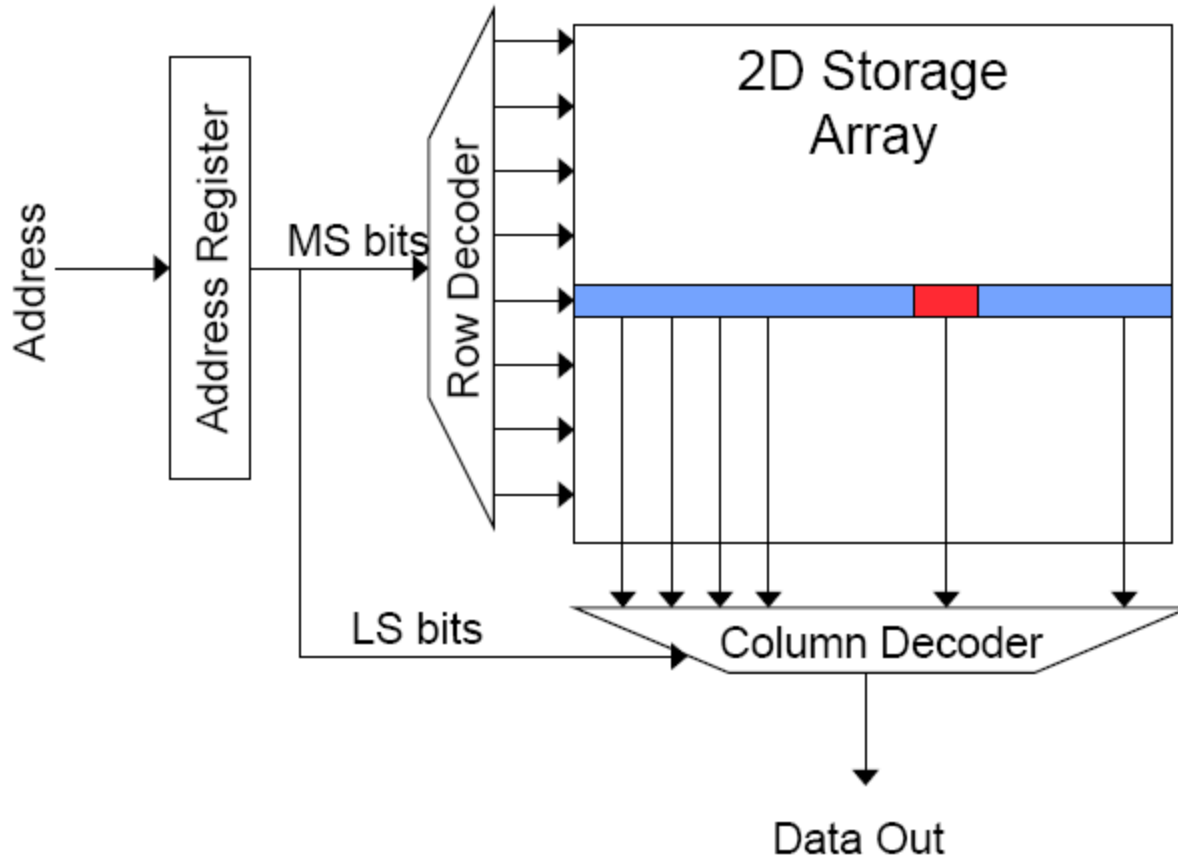
### ■ Required

- 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](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

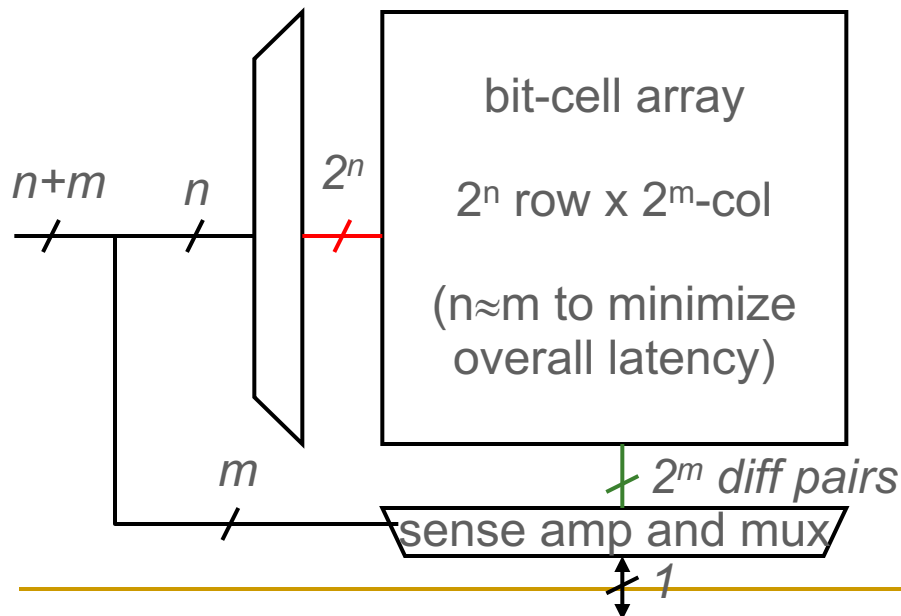
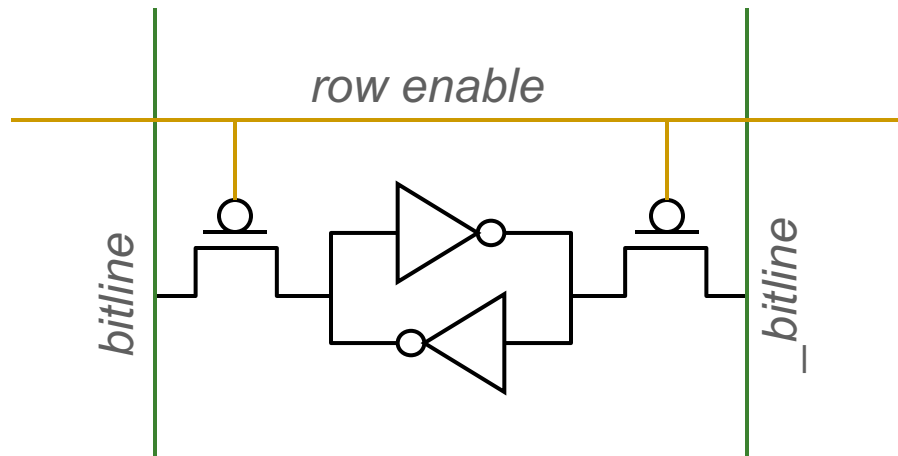


## ■ Read access sequence:

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

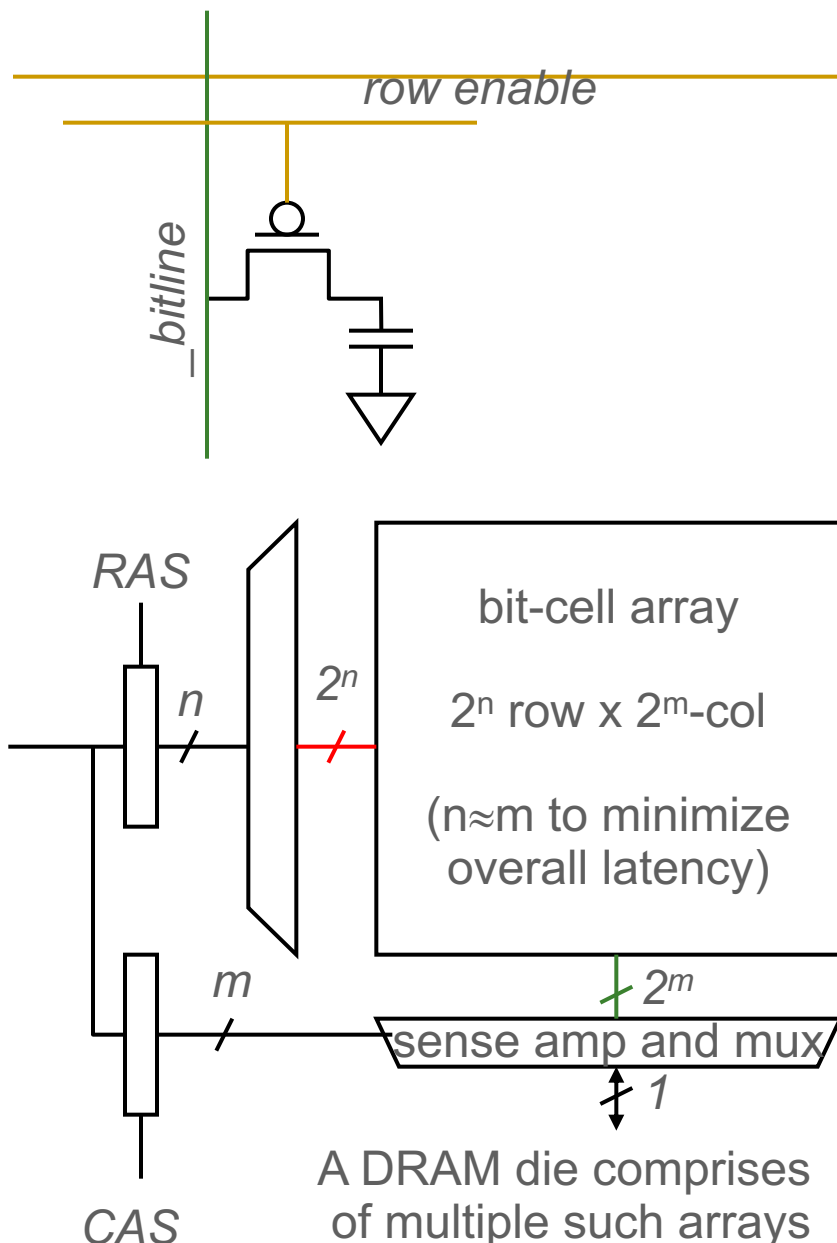
1. address decode
2. drive row select
3. selected bit-cells drive bitlines  
(entire row is read together)
4. differential sensing and column select  
(data is ready)
5. precharge all bitlines  
(for next read or write)

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

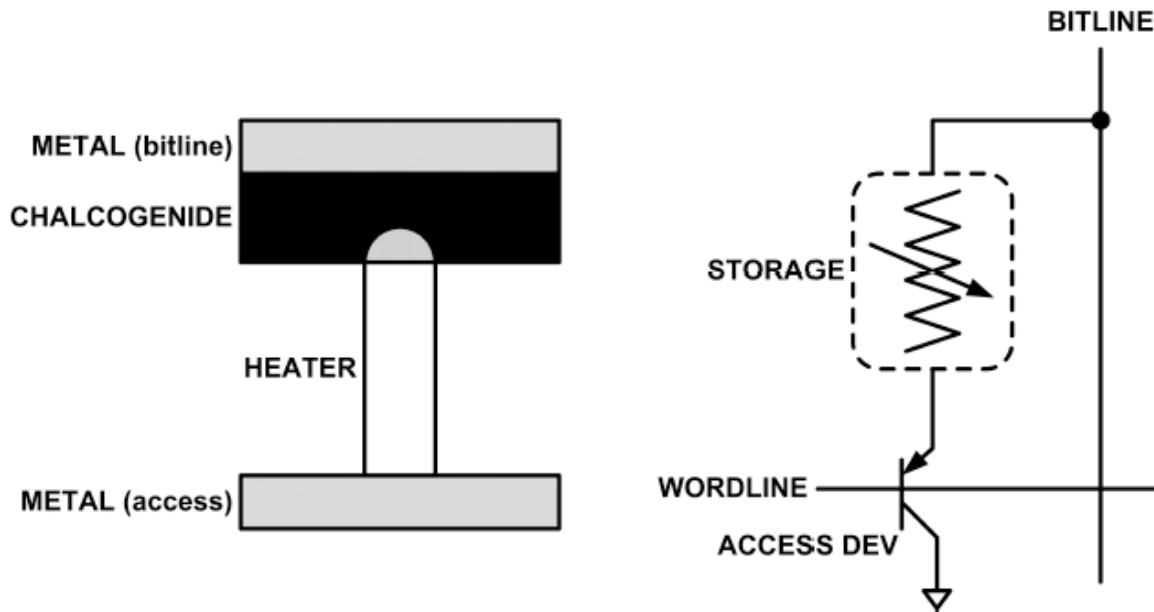
- ❑ Slower access (capacitor)
- ❑ Higher density (1T 1C cell)
- ❑ Lower cost
- ❑ Requires refresh (power, performance, circuitry)
- ❑ Manufacturing requires putting capacitor and logic together

## ■ 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 (heating and cooling based “phase change” operation)
- ❑ 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 to it)
- ❑ Higher access energy

# Recall: Charge vs. Resistive Memories

---

- Charge Memory (e.g., DRAM, Flash)
  - Write data by capturing charge  $Q$
  - Read data by detecting voltage  $V$
- Resistive Memory (e.g., PCM, STT-MRAM, memristors)
  - Write data by pulsing current  $dQ/dt$
  - Read data by detecting resistance  $R$

# Recall Promising New Memory Technologies

---

## ■ PCM

- Inject current to change material phase
- Resistance determined by phase

## ■ STT-MRAM

- Inject current to change magnet polarity
- Resistance determined by polarity

## ■ Memristors/RRAM/ReRAM

- Inject current to change atomic structure
- Resistance determined by atom distance

# More on Emerging Memory Technologies

## Phase Change Memory: Pros and Cons

- Pros over DRAM
  - ❑ Better technology scaling (capacity and cost)
  - ❑ Non volatile → Persistent
  - ❑ Low idle power (no refresh)
- Cons
  - ❑ Higher latencies: ~4-15x DRAM (especially write)
  - ❑ Higher active energy: ~2-50x DRAM (especially write)
  - ❑ Lower endurance (a cell dies after  $\sim 10^8$  writes)
  - ❑ Reliability issues (resistance drift)
- Challenges in enabling PCM as DRAM replacement/helper:
  - ❑ Mitigate PCM shortcomings
  - ❑ Find the right way to place PCM in the system

SAFARI

20



Computer Architecture - Lecture 15: Emerging Memory Technologies (ETH Zürich, Fall 2020)

1,047 views • Nov 14, 2020

👍 24 💬 0 ➦ SHARE ⚙️ SAVE ...



Onur Mutlu Lectures  
16.3K subscribers

ANALYTICS

EDIT VIDEO

[https://www.youtube.com/watch?v=AIE1rD9G\\_YU&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=28](https://www.youtube.com/watch?v=AIE1rD9G_YU&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=28)

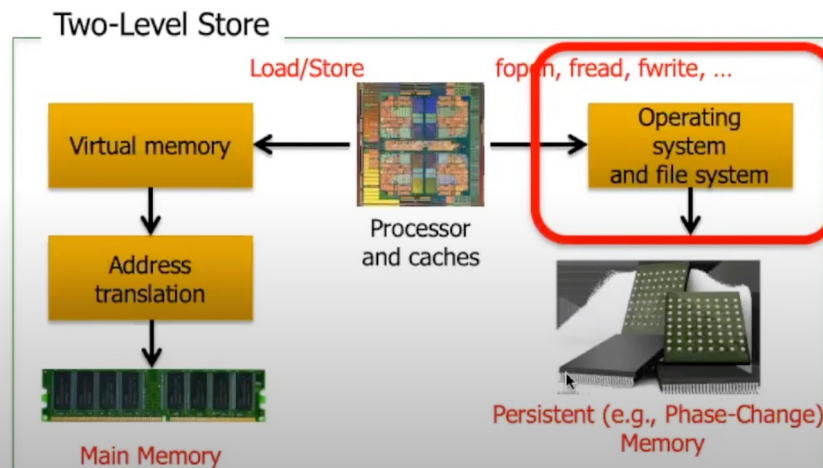


# More on Emerging Memory Technologies

## Two-Level Memory/Storage Model

- The traditional two-level storage model is a bottleneck with NVM

- **Volatile** data in memory → a **load/store** interface
- **Persistent** data in storage → a **file system** interface
- Problem: Operating system (OS) and file system (FS) code to locate, translate, buffer data become performance and energy bottlenecks with fast NVM stores



11

Comp. Arch. - Lect. 16a: Opportunities & Challenges of Emerging Memory Tech. (ETH Zürich Fall 2020)

512 views • Nov 20, 2020

14 0 SHARE SAVE ...

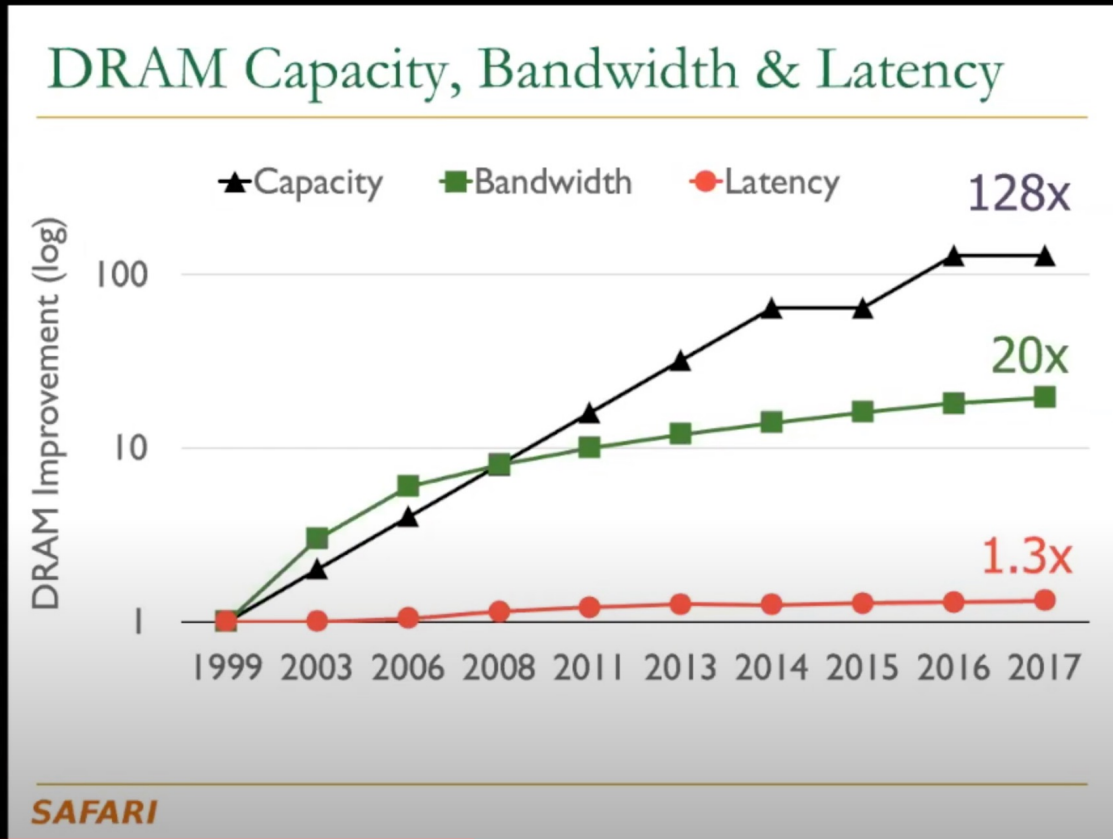


Onur Mutlu Lectures  
16.3K subscribers

ANALYTICS

EDIT VIDEO

# More on Memory Technologies



Computer Arch. - Lecture 3b: Memory Systems: Challenges and Opportunities (ETH Zürich, Fall 2020)

1,446 views • Sep 26, 2020

22 0 SHARE SAVE ...



**Onur Mutlu Lectures**  
16.3K subscribers

ANALYTICS

EDIT VIDEO

<https://www.youtube.com/watch?v=pmLszWGmMGQ&list=PL5Q2soXY2Zi9xidylgBxUz7xRPS-wisBN&index=29>

# Recall: Flash Memory & Solid-State Drives

- Flash memory was a very “doubtful” emerging technology
  - for at least two decades



*Proceedings of the IEEE, Sept. 2017*

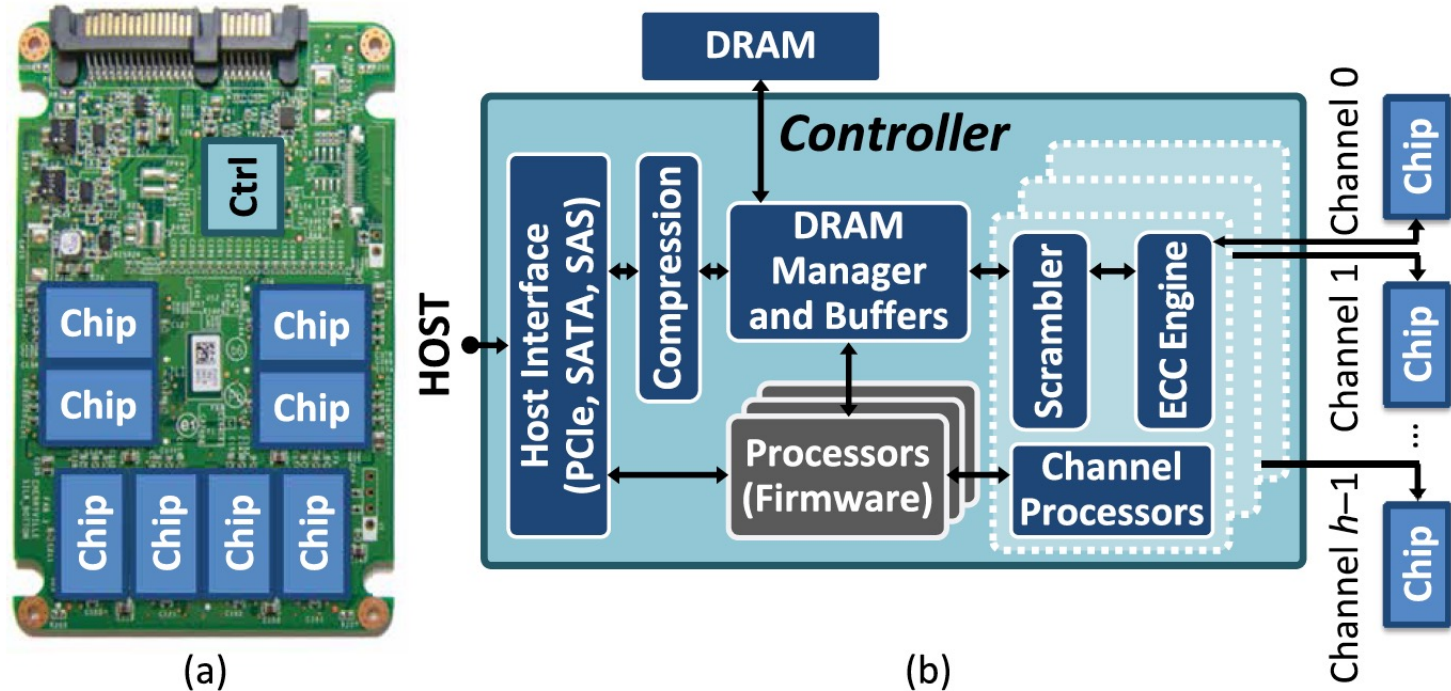
## Error Characterization, Mitigation, and Recovery in Flash-Memory-Based Solid-State Drives

By YU CAI, SAUGATA GHOSE, ERICH F. HARATSCH, YIXIN LUO, AND ONUR MUTLU

**ABSTRACT** | NAND flash memory is ubiquitous in everyday life today because its capacity has continuously increased and

**KEYWORDS** | Data storage systems; error recovery; fault tolerance; flash memory; reliability; solid-state drives

# Recall: A Flash Memory SSD Controller



**Fig. 1.** (a) SSD system architecture, showing controller (Ctrl) and chips. (b) Detailed view of connections between controller components and chips.



# Lecture on Flash Memory & SSDs

Planar vs. 3D NAND Flash Memory

	Planar NAND Flash Memory	3D NAND Flash Memory
Scaling	Reduce flash cell size, Reduce distance b/w cells	Increase # of layers
Reliability	Scaling hurts reliability	Not well studied!

SAFARI 20

ETH ZÜRICH HAUPTGEBÄUDE

Computer Architecture - Lecture 26: Flash Memory and Solid-State Drives (ETH Zürich, Fall 2020)

1,771 views • Dec 31, 2020

43 0 SHARE SAVE ...

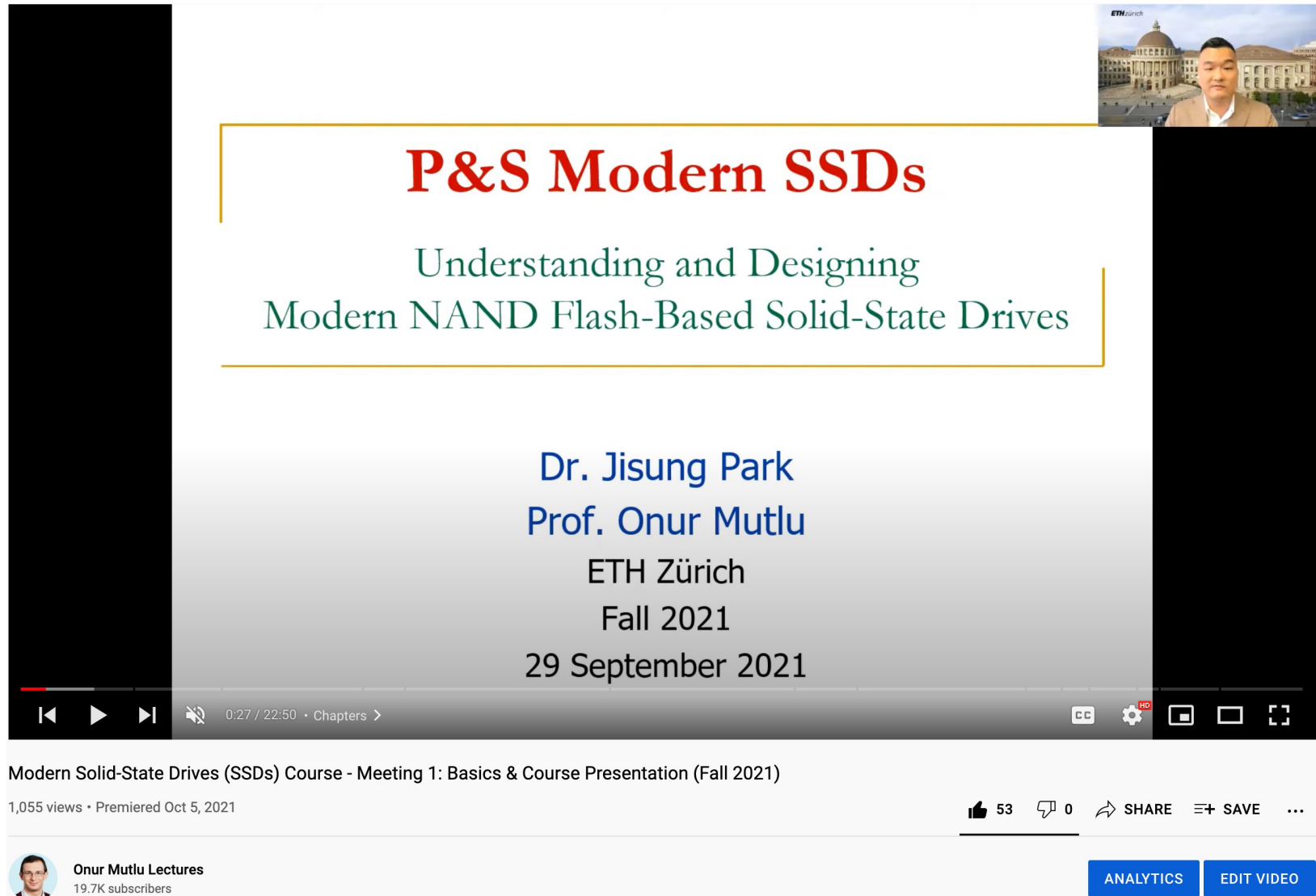


Onur Mutlu Lectures  
19.7K subscribers

ANALYTICS

EDIT VIDEO

# Special Course on Flash Memory & SSDs



**P&S Modern SSDs**

Understanding and Designing  
Modern NAND Flash-Based Solid-State Drives

Dr. Jisung Park  
Prof. Onur Mutlu  
ETH Zürich  
Fall 2021  
29 September 2021

Modern Solid-State Drives (SSDs) Course - Meeting 1: Basics & Course Presentation (Fall 2021)

1,055 views • Premiered Oct 5, 2021

Onur Mutlu Lectures  
19.7K subscribers

53 0 SHARE SAVE ...

ANALYTICS EDIT VIDEO

# Lectures on Memory Technologies

---

## ■ Computer Architecture, Fall 2020, Lecture 15

- ❑ Emerging Memory Technologies (ETH, Fall 2020)
- ❑ [https://www.youtube.com/watch?v=AIE1rD9G\\_YU&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=28](https://www.youtube.com/watch?v=AIE1rD9G_YU&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=28)

## ■ Computer Architecture, Fall 2020, Lecture 16a

- ❑ Opportunities & Challenges of Emerging Memory Tech (ETH, Fall 2020)
- ❑ <https://www.youtube.com/watch?v=pmLszWGmMGQ&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=29>

## ■ Computer Architecture, Fall 2020, Lecture 3b

- ❑ Memory Systems: Challenges & Opportunities (ETH, Fall 2020)
- ❑ <https://www.youtube.com/watch?v=Q2FbUxD7GHs&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=6>

# A Tutorial on Memory-Centric Systems

---

- Onur Mutlu,

## **"Memory-Centric Computing Systems"**

Invited Tutorial at *66th International Electron Devices Meeting (IEDM)*, Virtual, 12 December 2020.

[[Slides \(pptx\)](#) ([pdf](#))]

[[Executive Summary Slides \(pptx\)](#) ([pdf](#))]

[[Tutorial Video](#) (1 hour 51 minutes)]

[[Executive Summary Video](#) (2 minutes)]

[[Abstract and Bio](#)]

[[Related Keynote Paper from VLSI-DAT 2020](#)]

[[Related Review Paper on Processing in Memory](#)]

<https://www.youtube.com/watch?v=H3sEaINPBOE>



# Memory-Centric Computing Systems



Onur Mutlu

[omutlu@gmail.com](mailto:omutlu@gmail.com)

<https://people.inf.ethz.ch/omutlu>

12 December 2020

IEDM Tutorial

**SAFARI**

**ETH** zürich

Carnegie Mellon



0:06 / 1:51:05



IEDM 2020 Tutorial: Memory-Centric Computing Systems, Onur Mutlu, 12 December 2020

1,641 views • Dec 23, 2020

48 0 SHARE SAVE ...



Onur Mutlu Lectures  
13.9K subscribers

<https://www.youtube.com/watch?v=H3sEaINPBOE>

ANALYTICS

EDIT VIDEO

<https://www.youtube.com/onurmutlulectures>

# A Tutorial on Processing in Memory

---

- Onur Mutlu,  
**"Memory-Centric Computing"**  
*Education Class at Embedded Systems Week (ESWEEK),*  
Virtual, 9 October 2021.  
[Slides (pptx) (pdf)]  
[Abstract (pdf)]  
[Talk Video (2 hours, including Q&A)]  
[Invited Paper at DATE 2021]  
["A Modern Primer on Processing in Memory" paper]

**<https://www.youtube.com/watch?v=N1Ac1ov1JOM>**

# Memory-Centric Computing

Onur Mutlu

[omutlu@gmail.com](mailto:omutlu@gmail.com)

<https://people.inf.ethz.ch/omutlu>

9 October 2021

ESWEEK Education Class

**SAFARI**

**ETH** zürich

Carnegie Mellon



1:08 / 2:00:10



Embedded Systems Week (ESWEEK) 2021 Lecture - Memory-Centric Computing - Onur Mutlu - 9 October 2021

509 views • Premiered Dec 6, 2021

28 DISLIKE SHARE SAVE ...



Onur Mutlu Lectures  
20.7K subscribers

<https://www.youtube.com/watch?v=N1Ac1ov1JOM>

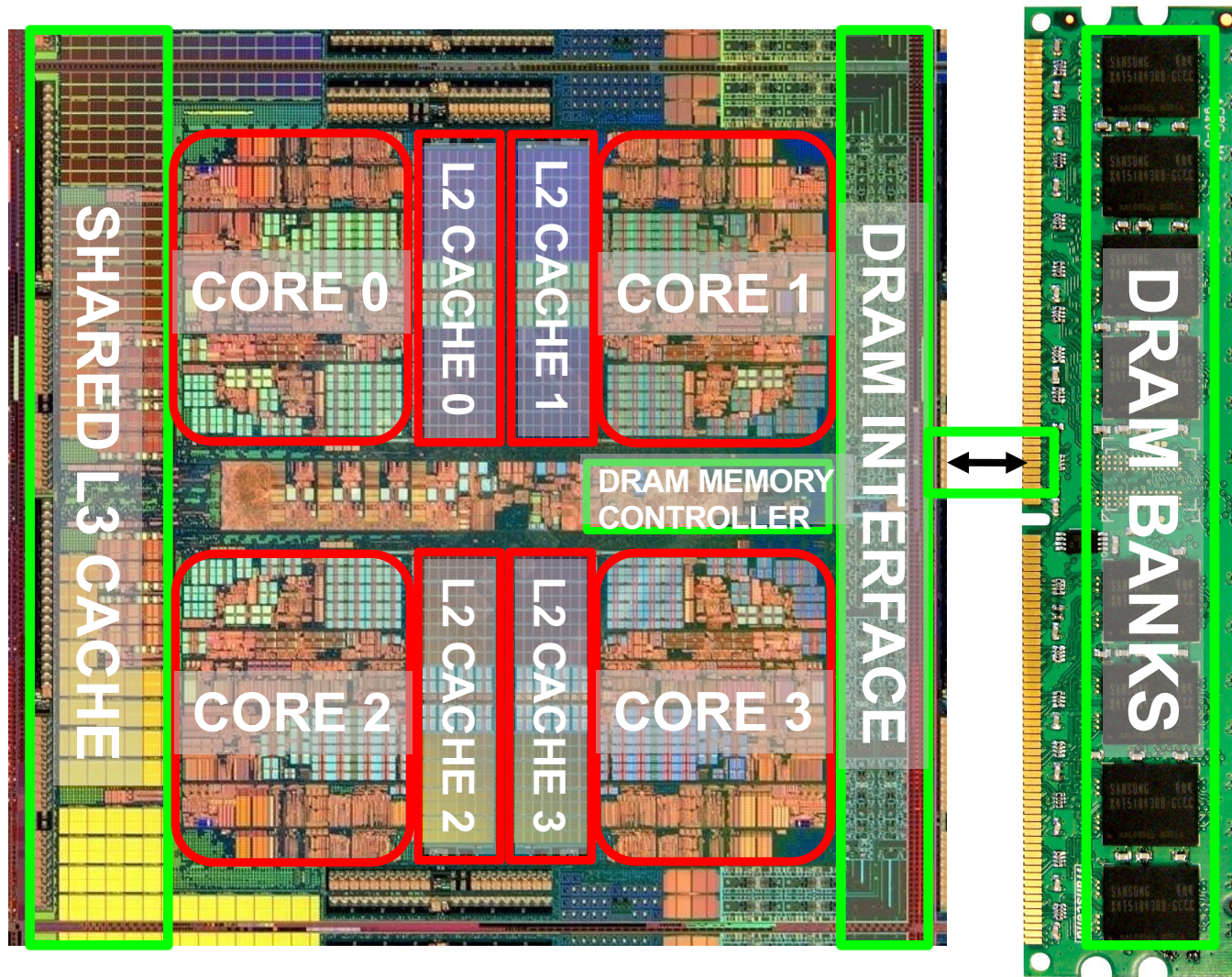
ANALYTICS

EDIT VIDEO

<https://www.youtube.com/onurmutlulectures>

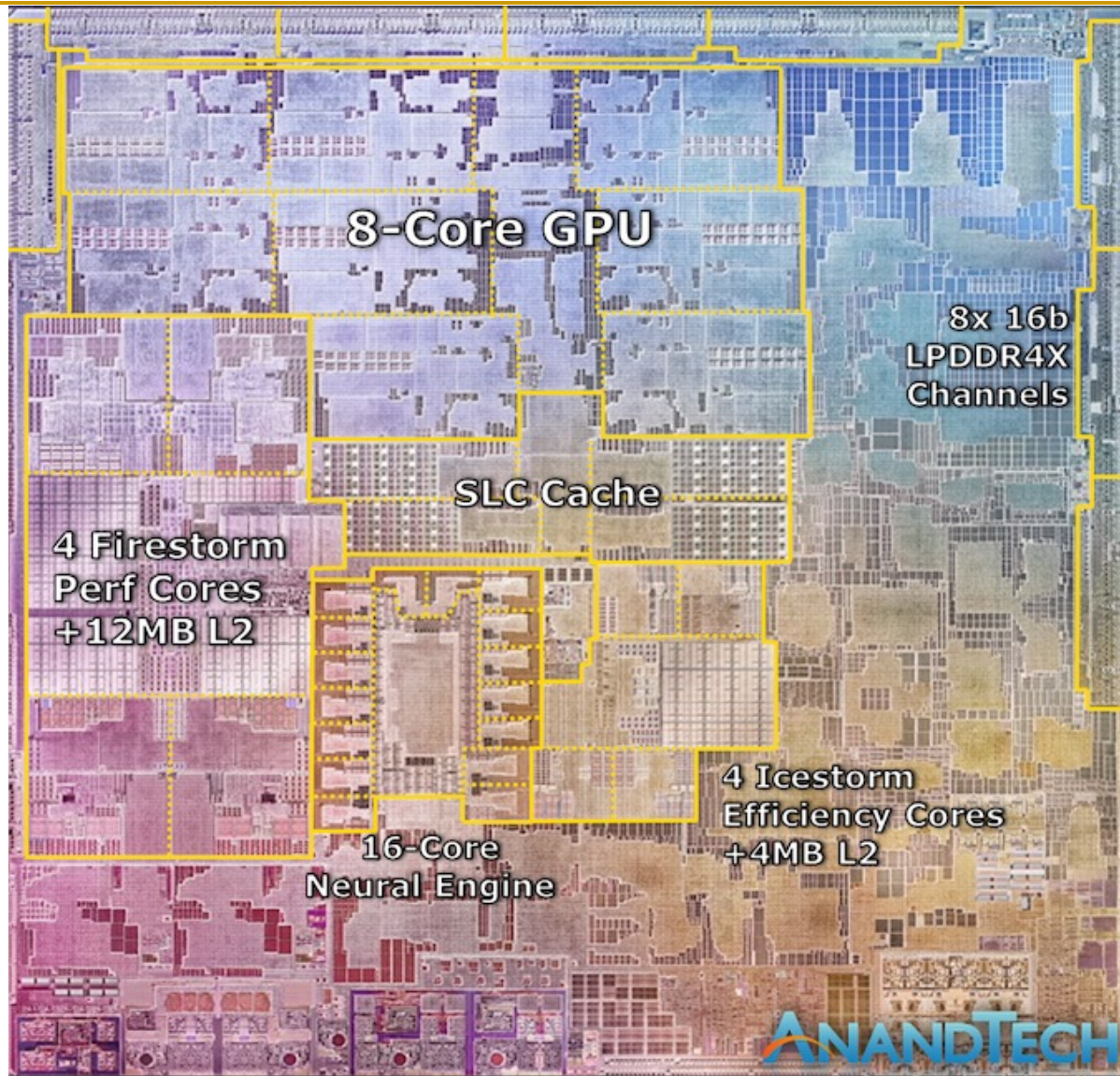
# The Memory Hierarchy

# Memory Hierarchy in a Modern System (I)



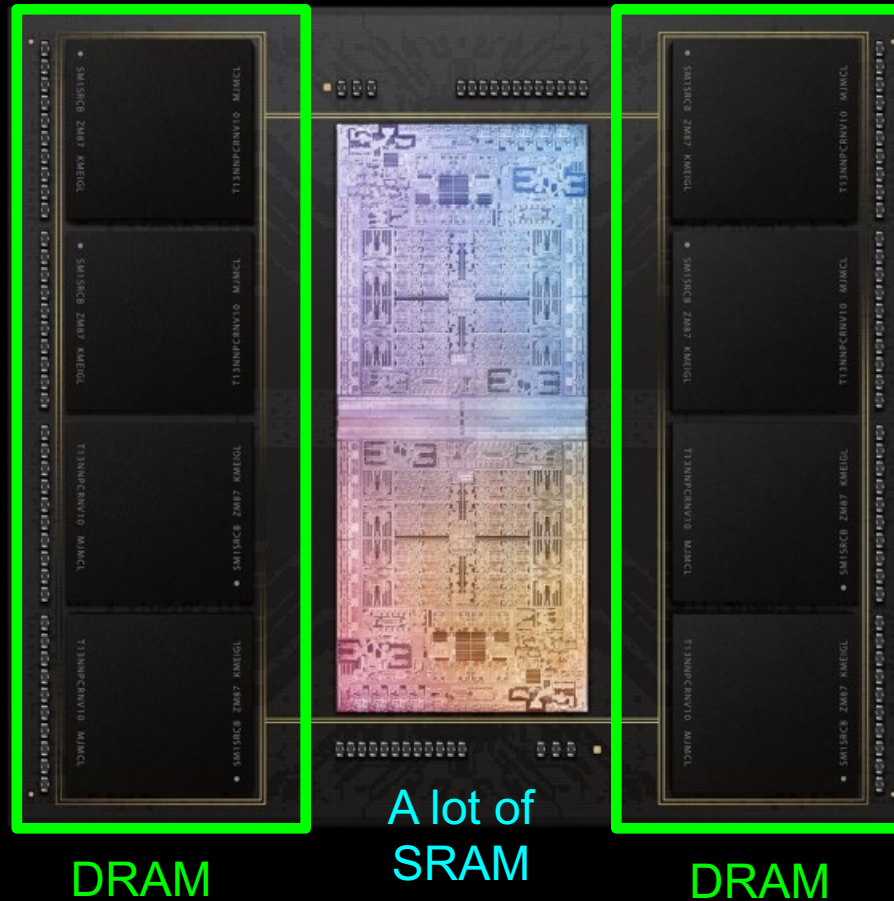


# Memory Hierarchy in a Modern System (II)



Apple M1,  
2021

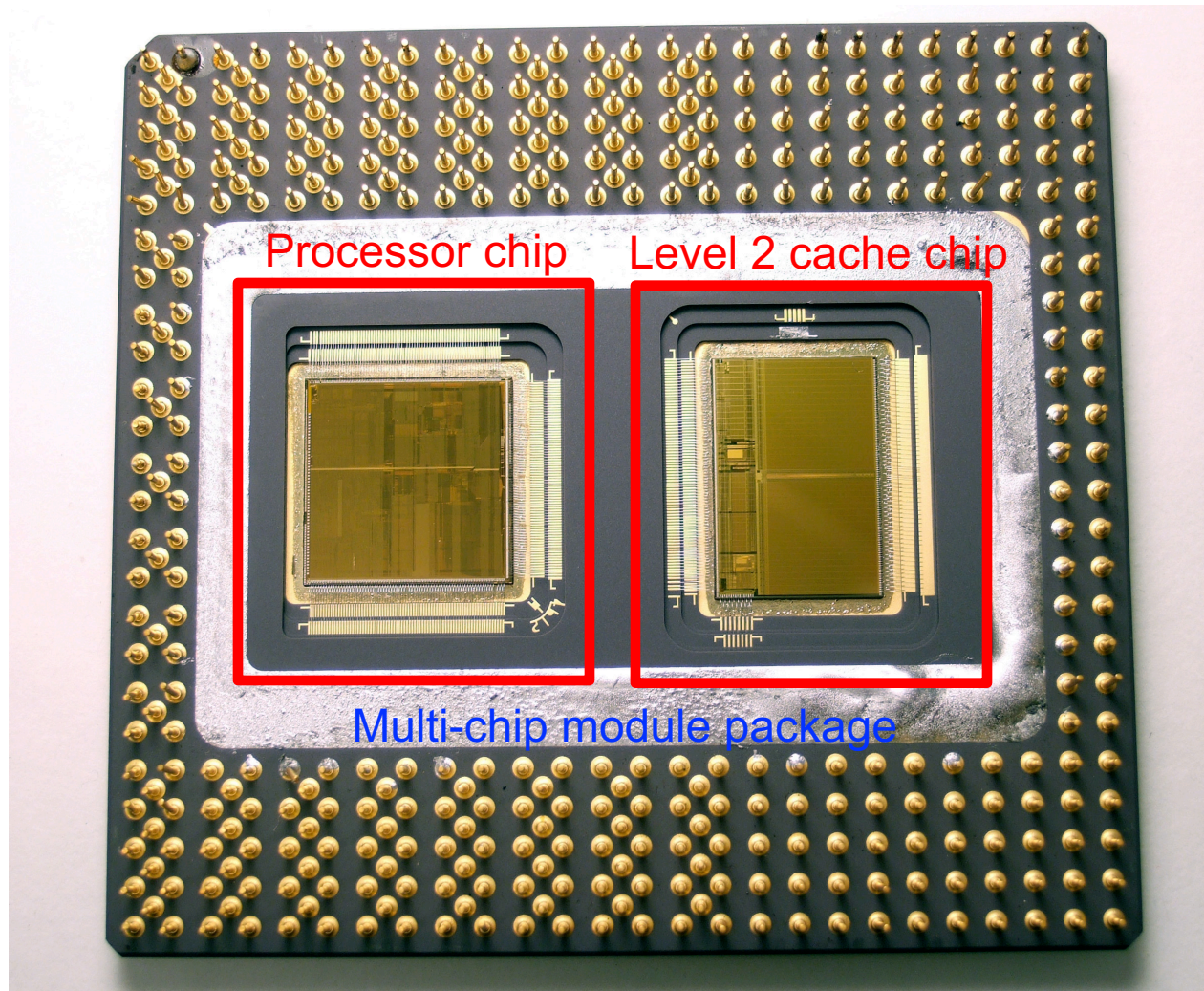
# Memory Hierarchy in a Modern System (III)



Apple M1 Ultra System (2022)



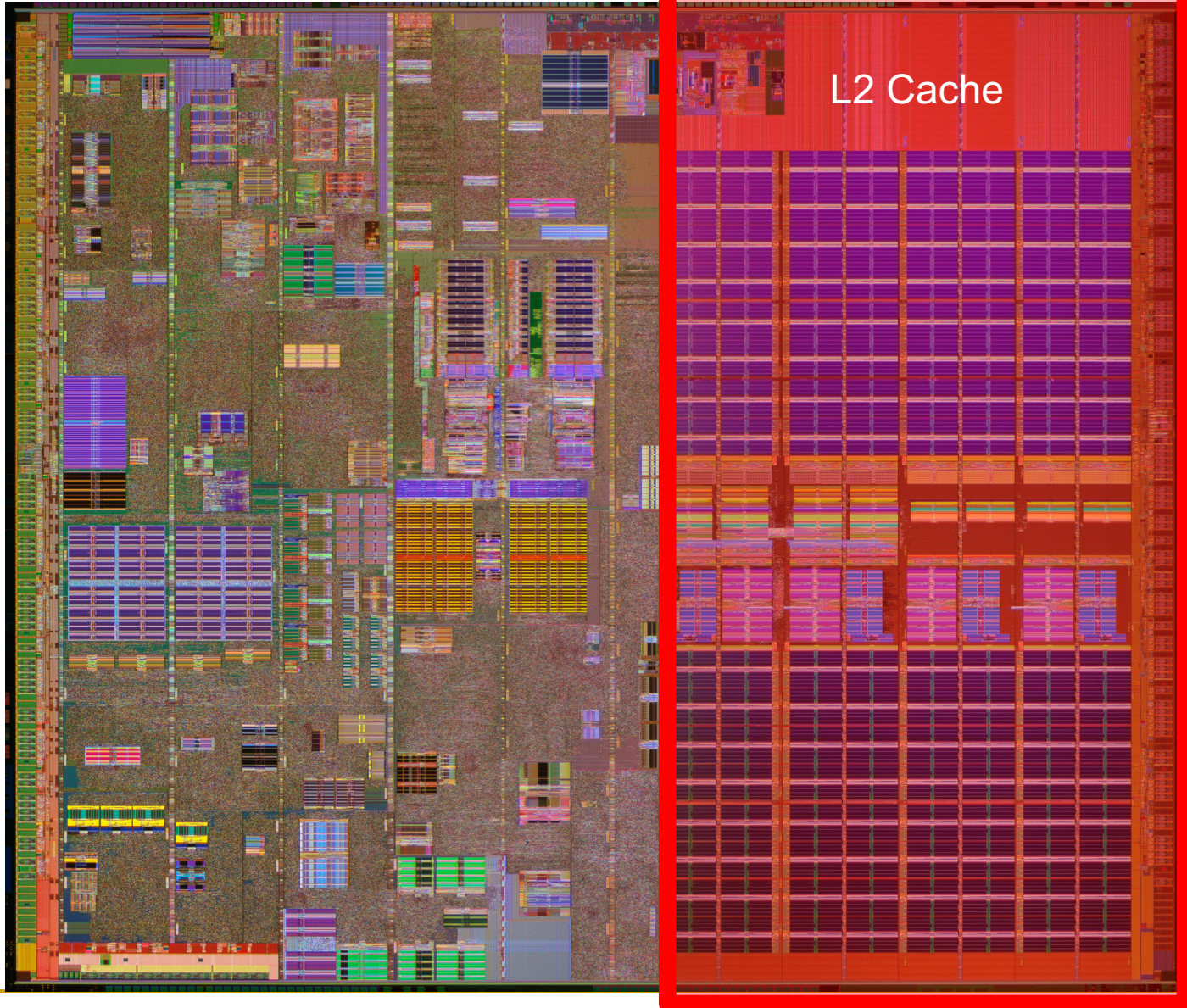
# Memory Hierarchy in an Older System



Intel Pentium Pro, 1995

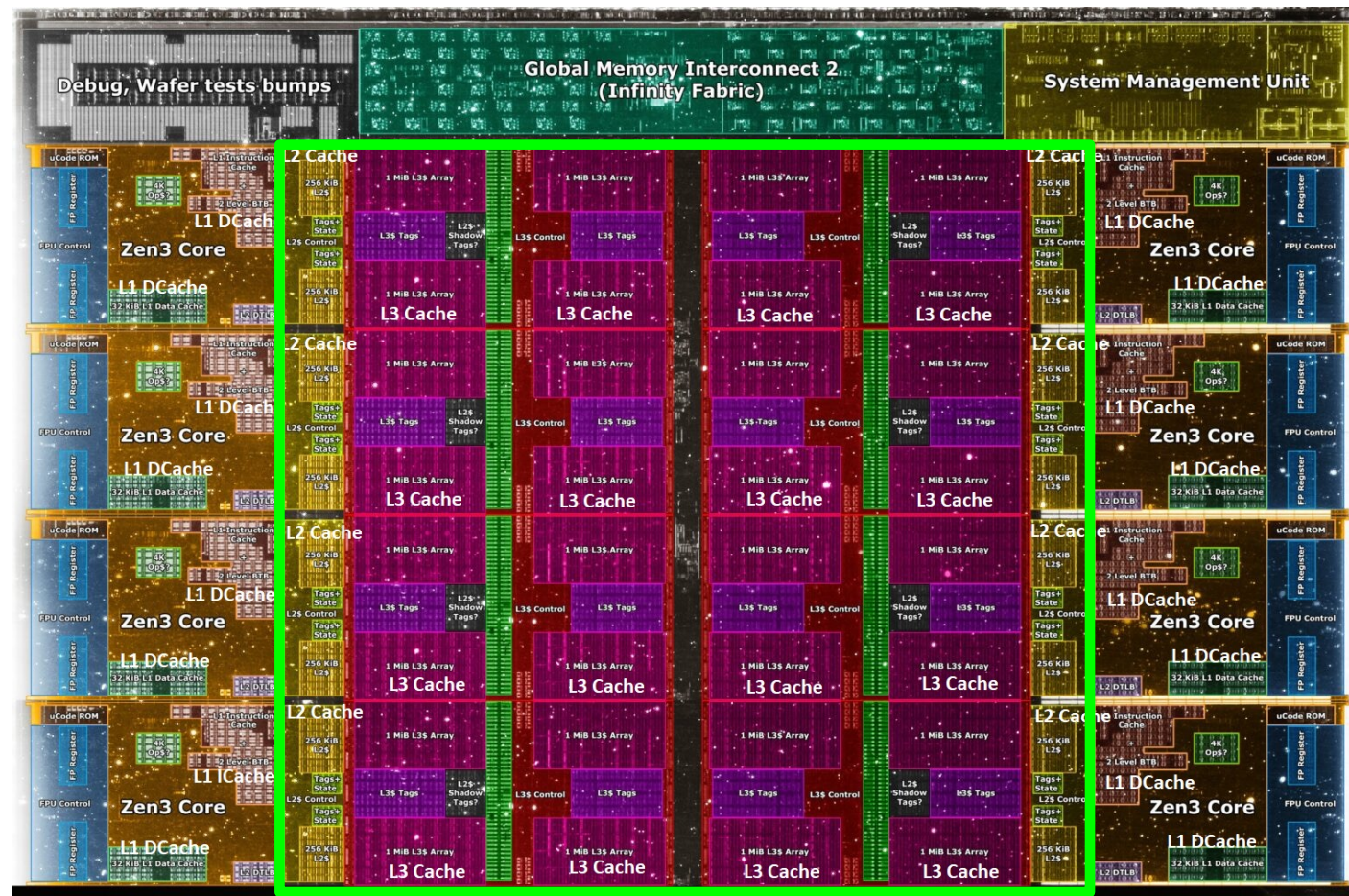


# Memory Hierarchy in an Older System





# Memory Hierarchy in a Modern System (IV)



Core Count:  
8 cores/16 threads

L1 Caches:  
32 KB per core

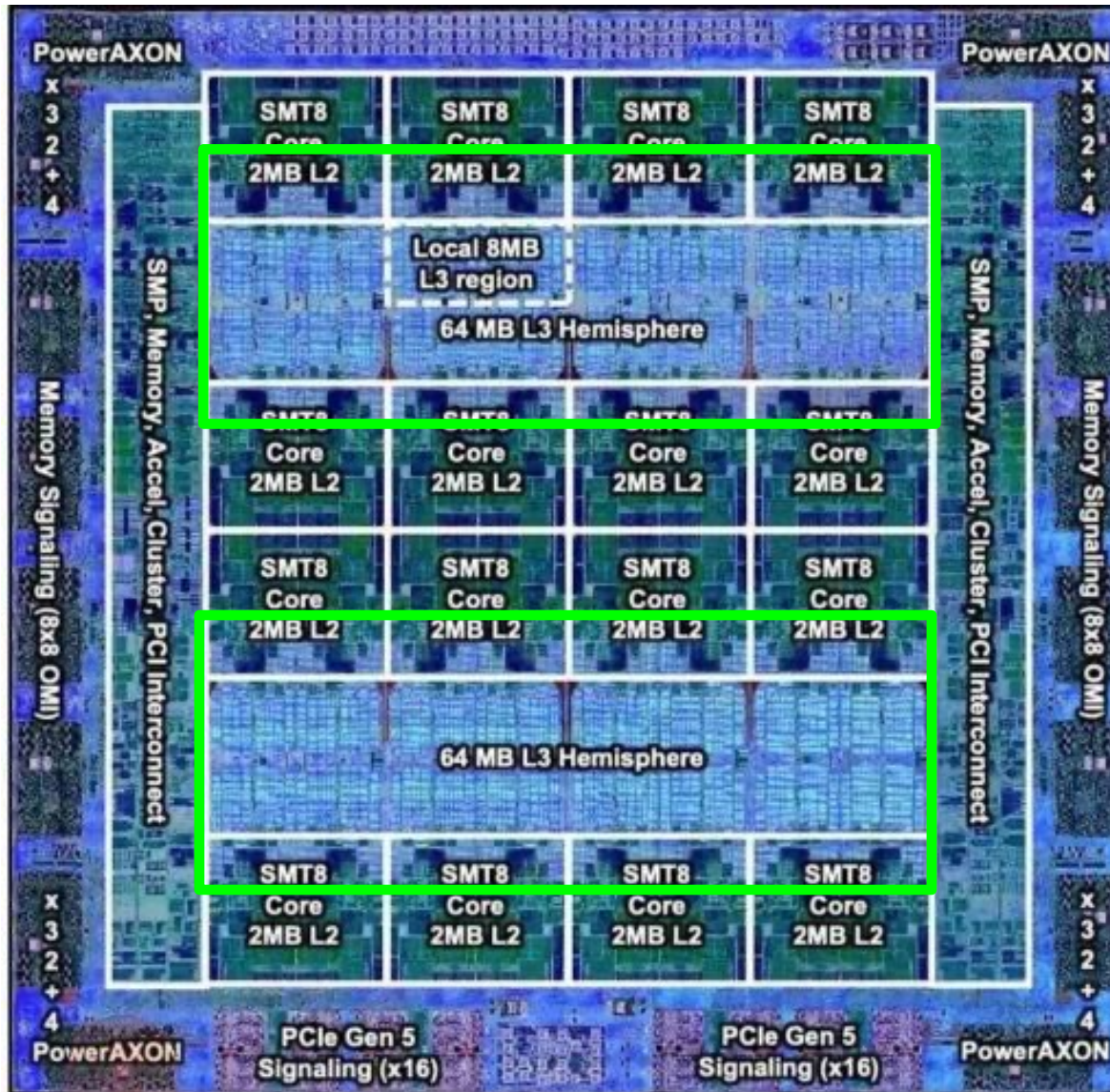
L2 Caches:  
512 KB per core

L3 Cache:  
32 MB shared

AMD Ryzen 5000, 2020



# Memory Hierarchy in a Modern System (V)



IBM POWER10,  
2020

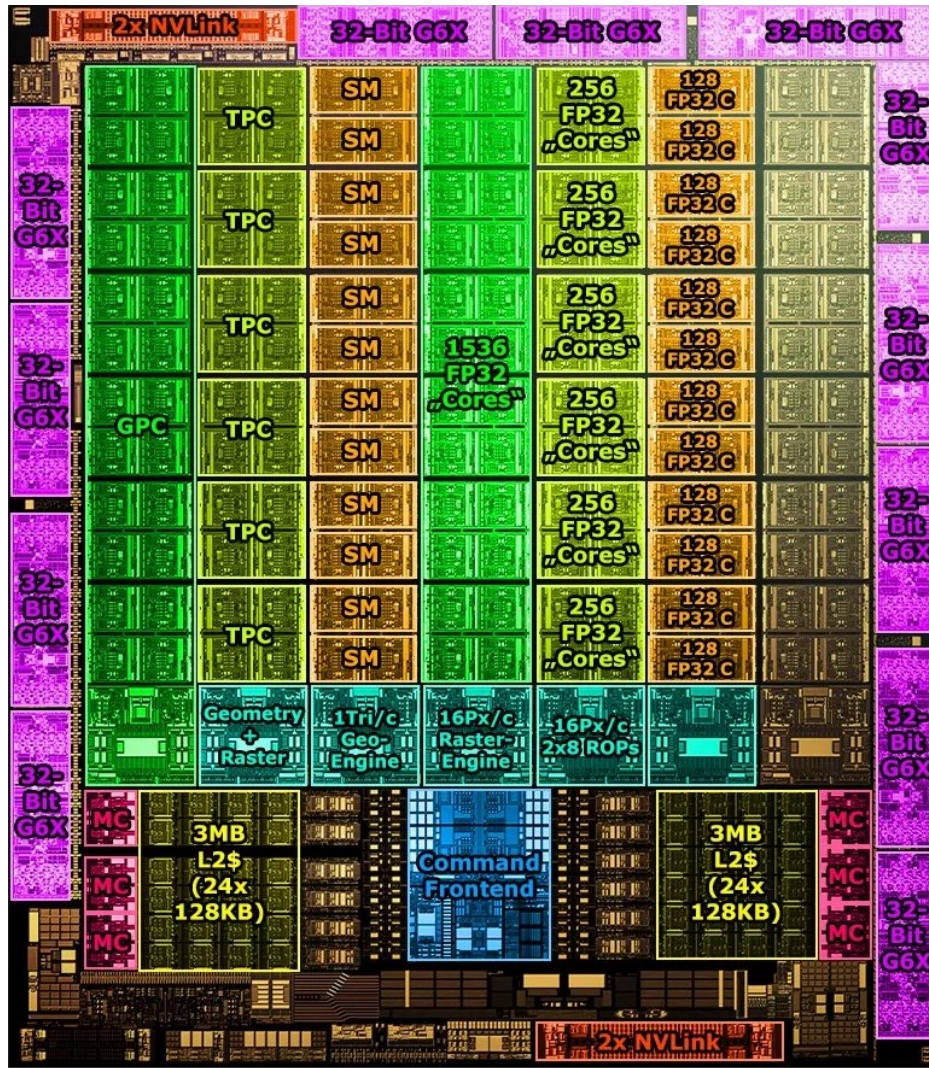
Cores:  
15-16 cores,  
8 threads/core

L2 Caches:  
2 MB per core

L3 Cache:  
120 MB shared



# Memory Hierarchy in a Modern System (VI)



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

# 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, < 1KByte, sub-nanosec
- ❑ SRAM, KByte~MByte, ~nanosec
- ❑ DRAM, Gigabyte, ~50 nanosec
- ❑ PCM-DIMM (Intel Optane DC DIMM), Gigabyte, ~300 nanosec
- ❑ PCM-SSD (Intel Optane SSD), Gigabyte ~Terabyte, ~6-10  $\mu$ s
- ❑ Flash memory, Gigabyte~Terabyte, ~50-100  $\mu$ s
- ❑ Hard Disk, Terabyte, ~10 millisec

## ■ Faster is more expensive (monetary cost 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 ~2022) scale with time

## ■ Other technologies have their place as well

- ❑ MRAM, RRAM, STT-MRAM, memristors, ... (not mature yet)

# The Problem (Table View)

---

Bigger is slower

Memory Device	Capacity	Latency	Cost per Megabyte
SRAM	< 1 KByte	sub-nanosec	
SRAM	KByte~MByte	~nanosec	< 0.3\$
DRAM	Gigabyte	~50 nanosec	< 0.03\$
PCM-DIMM (Intel Optane DC DIMM)	Gigabyte	~300 nanosec	< 0.004\$
PCM-SSD (Intel Optane SSD)	Gigabyte ~Terabyte	~6-10 $\mu$ s	< 0.001\$
Flash memory	Gigabyte ~Terabyte	~50-100 $\mu$ s	< 0.00008\$
Hard Disk	Terabyte	~10 millisec	< 0.00003\$

Faster is more expensive  
(\$\$\$ and chip area)

These sample values (circa ~2022) scale with time



# Aside: The Problem (2011 Version)

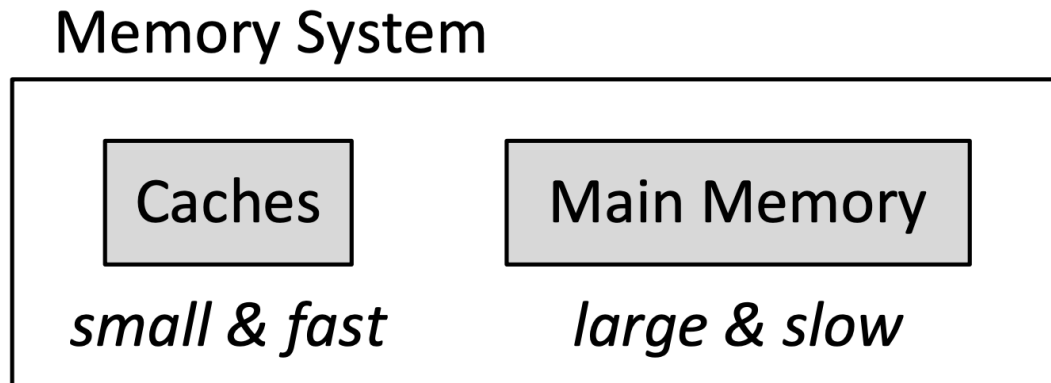
---

- 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 (monetary cost 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)

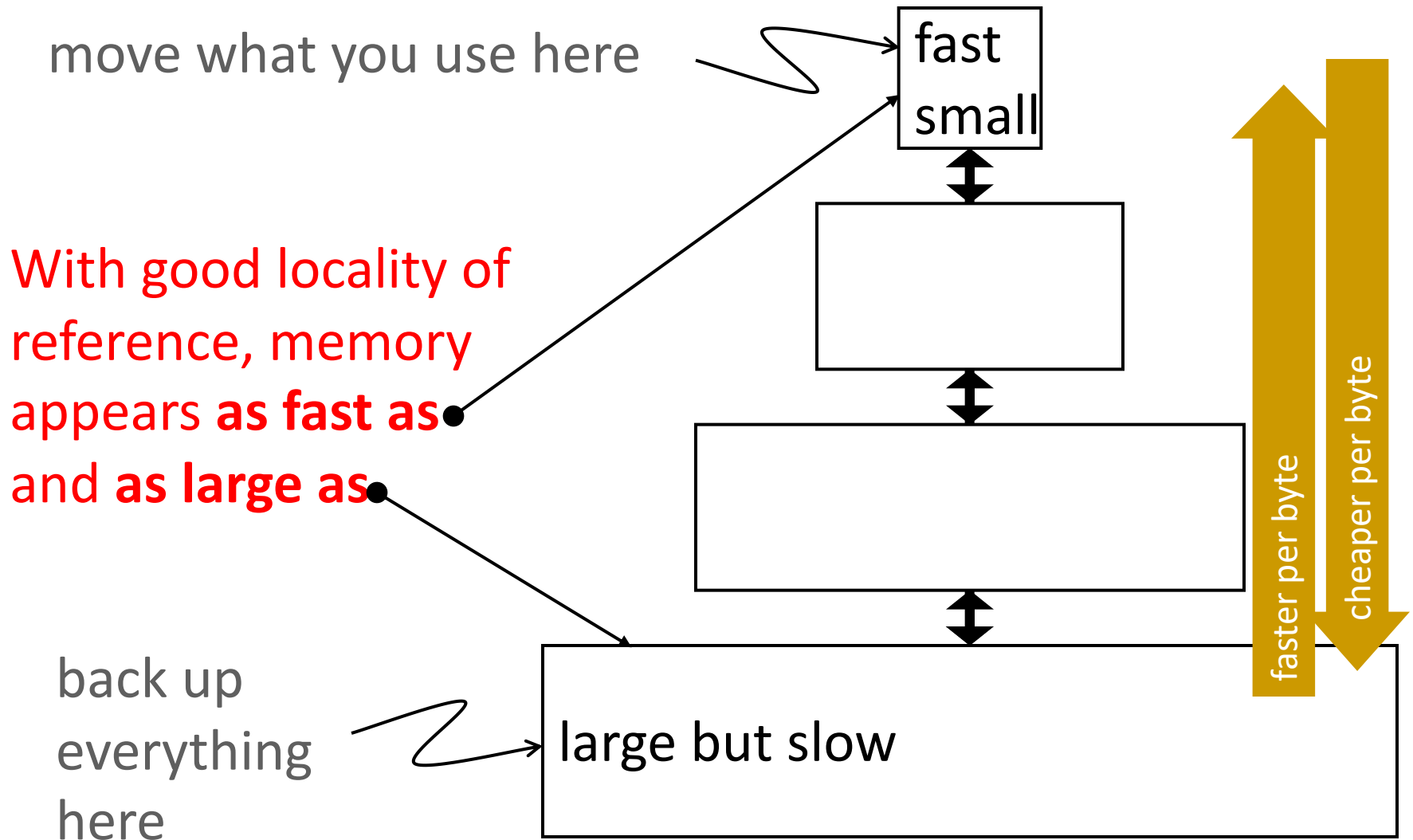
# Why Memory Hierarchy?

---

- 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)**

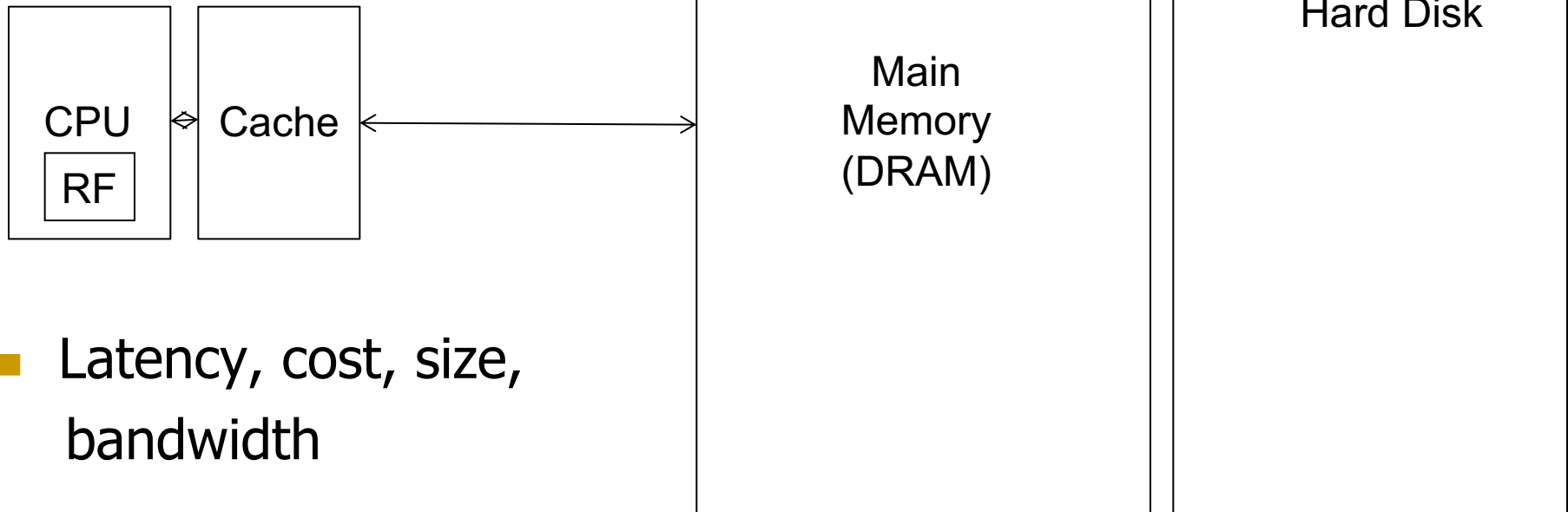


# The Memory Hierarchy



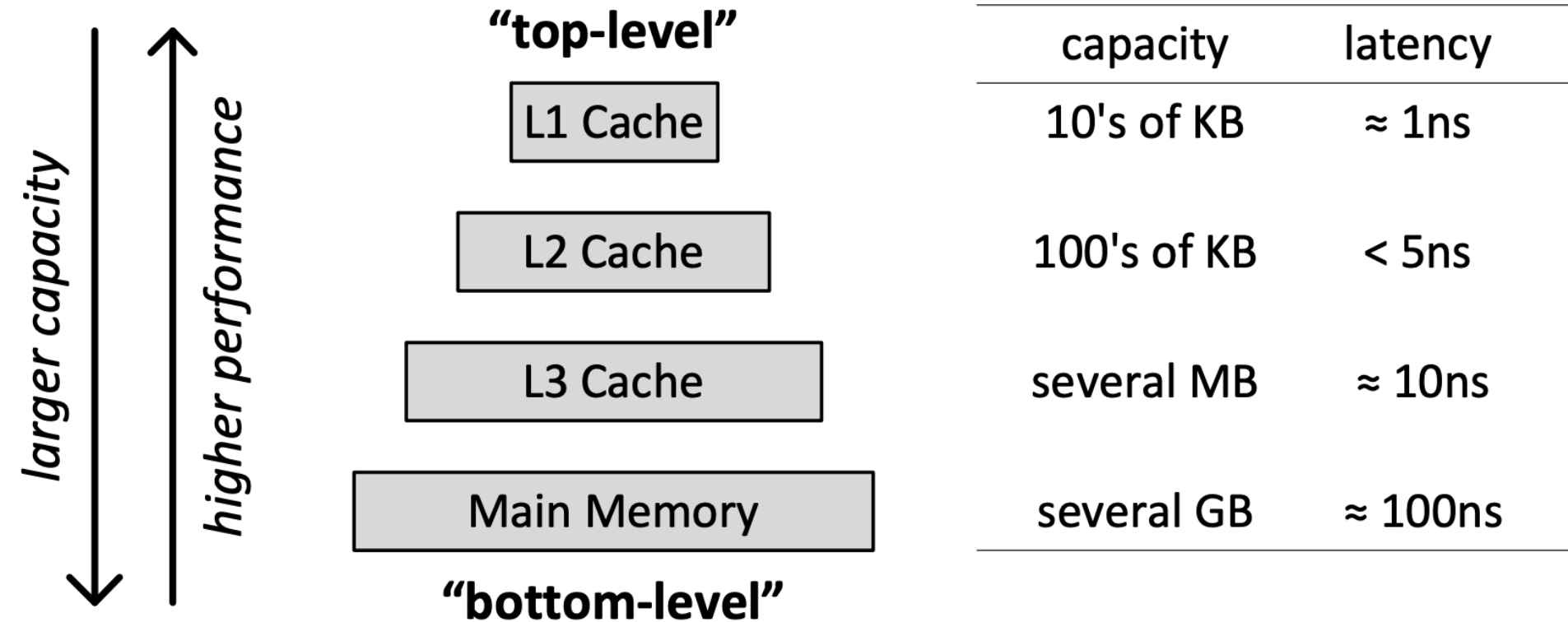
# Memory Hierarchy

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



- Latency, cost, size, bandwidth

# Memory Hierarchy Example



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

[https://people.inf.ethz.ch/omutlu/pub/memory-systems-introduction\\_computing-handbook14.pdf](https://people.inf.ethz.ch/omutlu/pub/memory-systems-introduction_computing-handbook14.pdf)

# Locality

---

- One's recent past is a very good predictor of their 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 → mostly 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

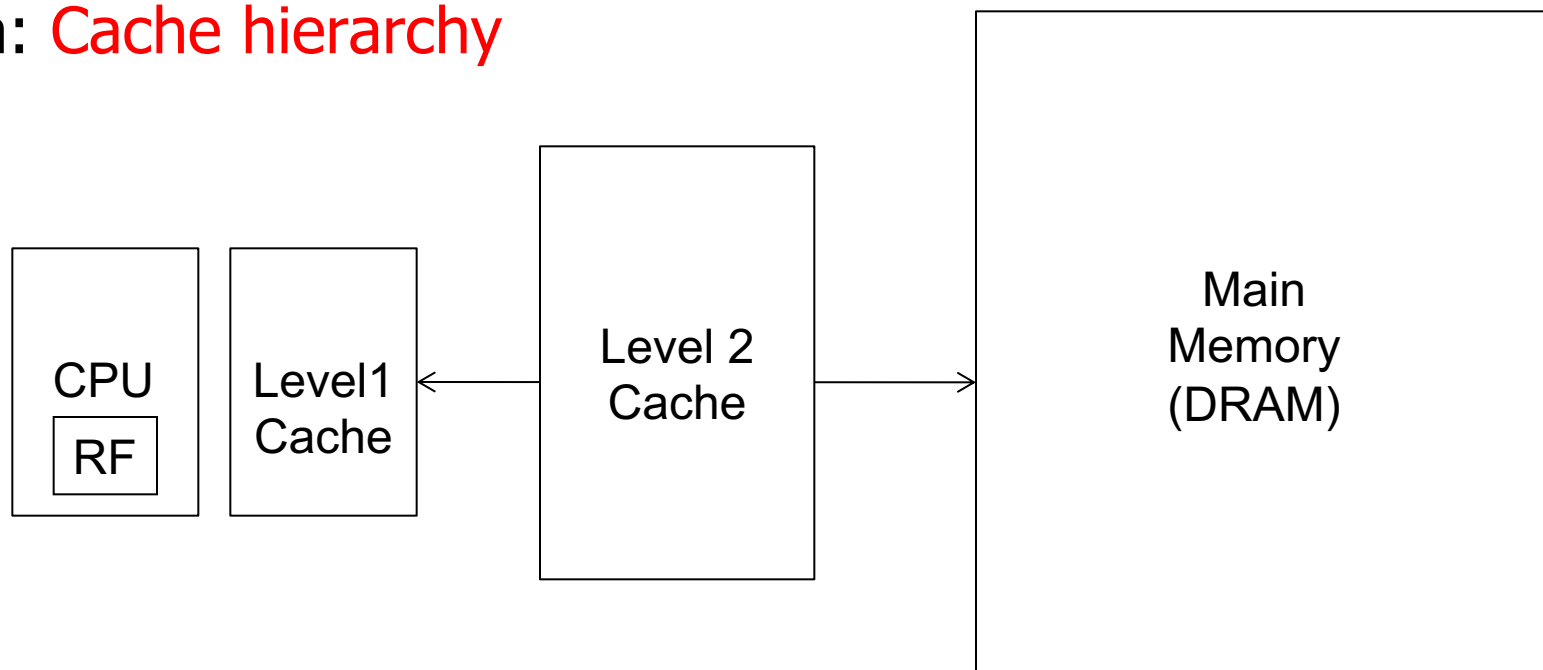
---

- 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**

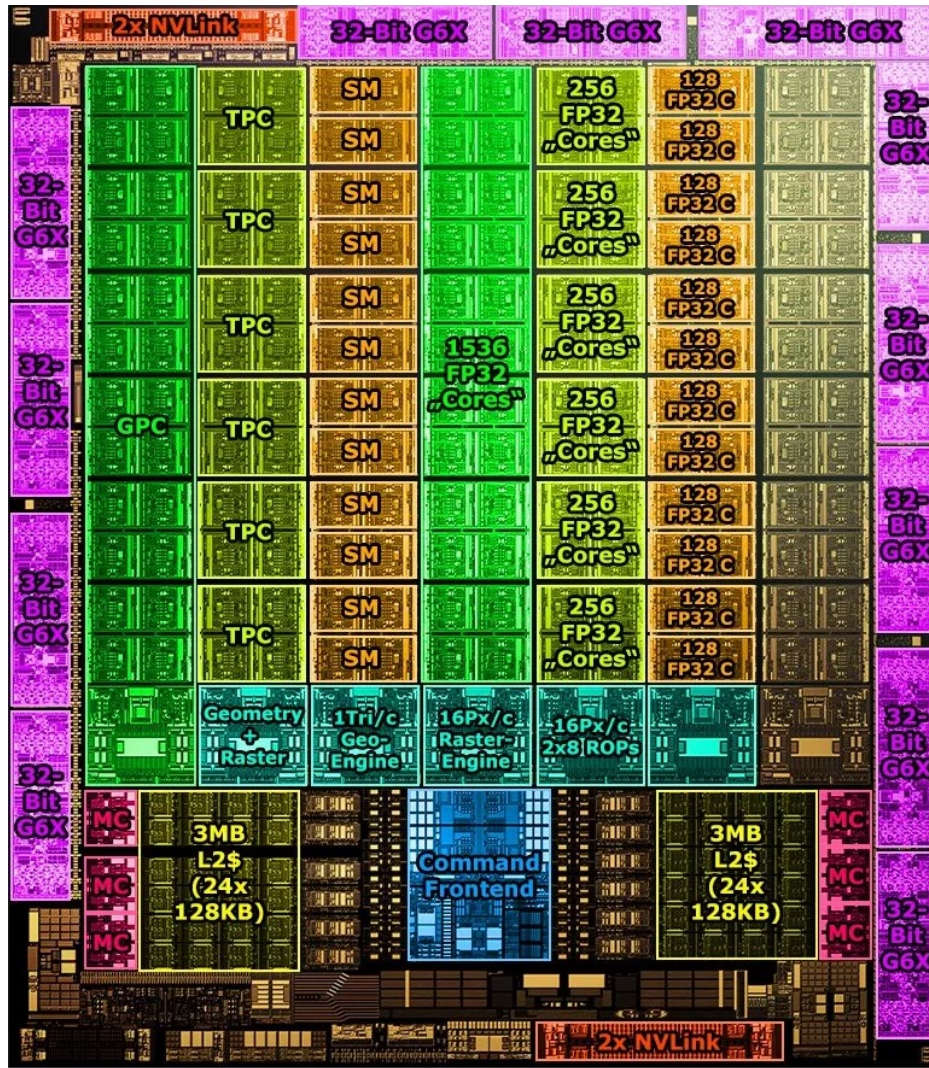


# A Note on Manual vs. Automatic Management

---

- **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

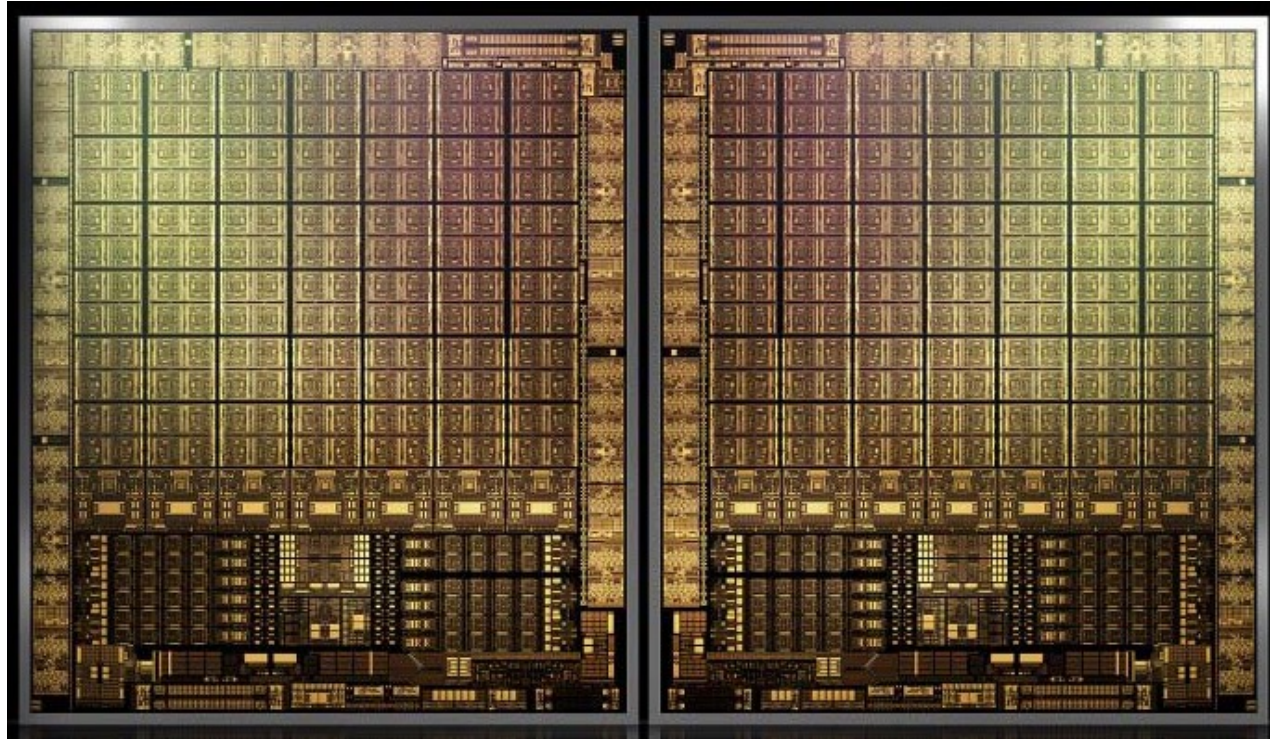
Can be used as L1 Cache and/or Scratchpad

## L2 Cache:

40 MB shared



# Caches and Scratchpad in a Modern GPU



Nvidia Hopper, 2022

## Cores:

144 Streaming  
Multiprocessors

## L1 Cache or Scratchpad:

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

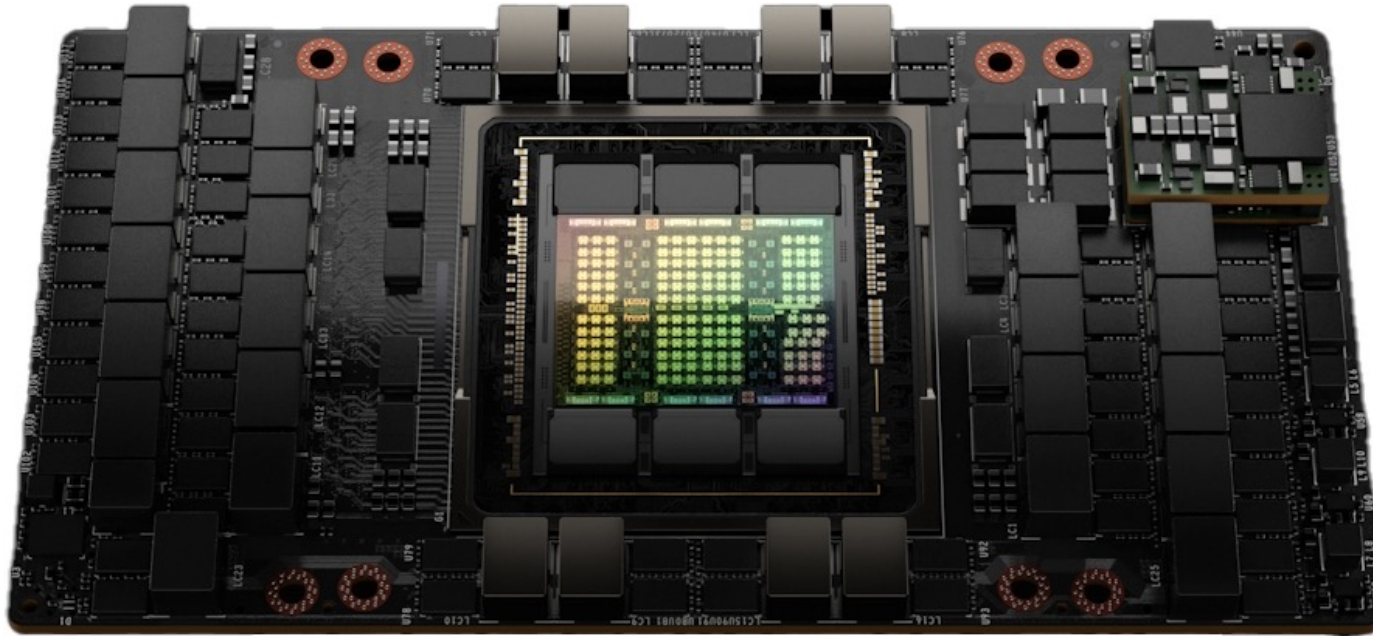
## L2 Cache:

60 MB shared



# Caches and Scratchpad in a Modern GPU

---



Nvidia Hopper,  
2022

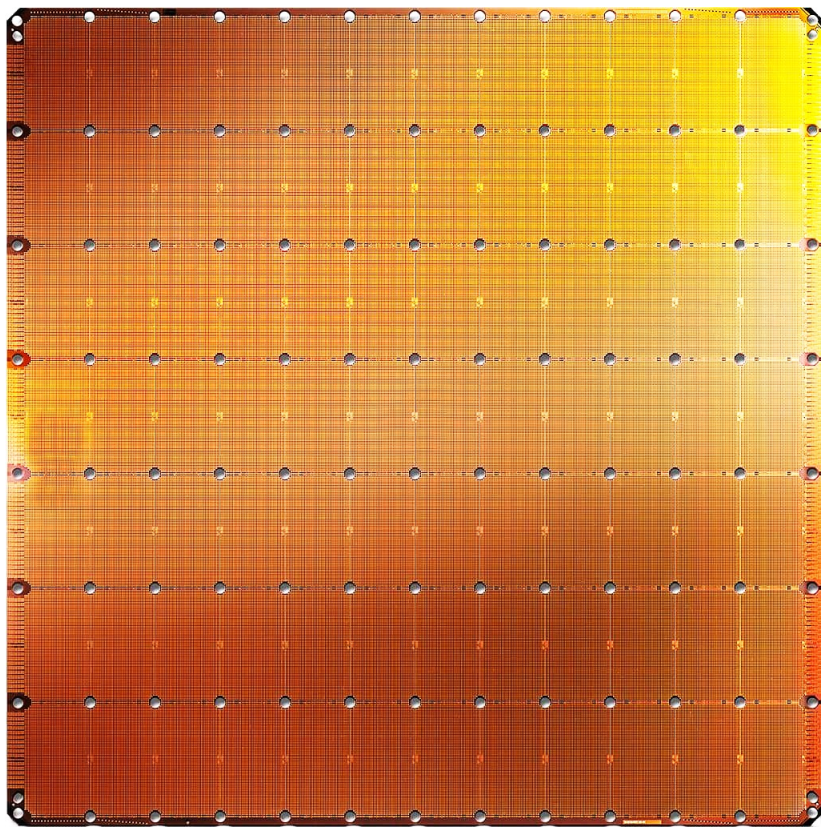
**Cores:**  
144 Streaming  
Multiprocessors

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

**L2 Cache:**  
60 MB shared

# Cerebras's Wafer Scale Engine (2019)

---



## **Cerebras WSE**

1.2 Trillion transistors

46,225 mm<sup>2</sup>

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



## **Largest GPU**

21.1 Billion transistors

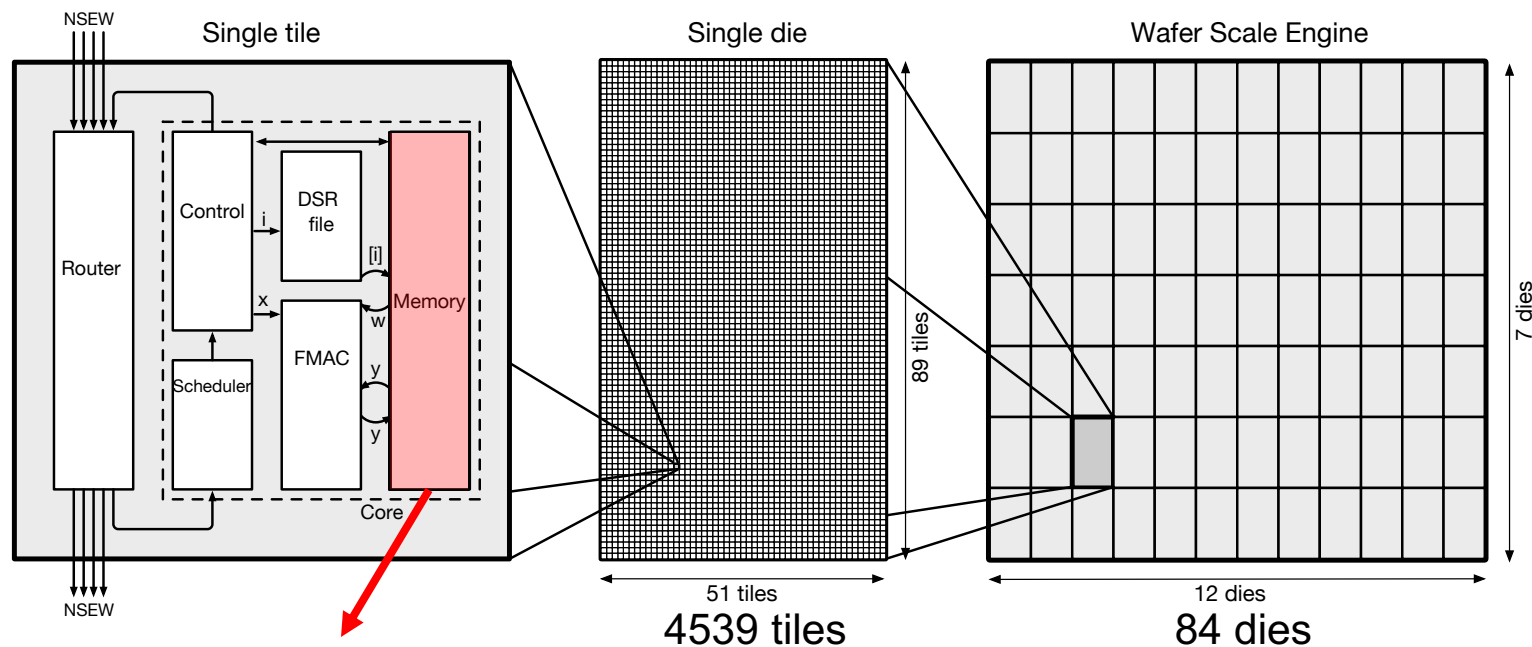
815 mm<sup>2</sup>

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



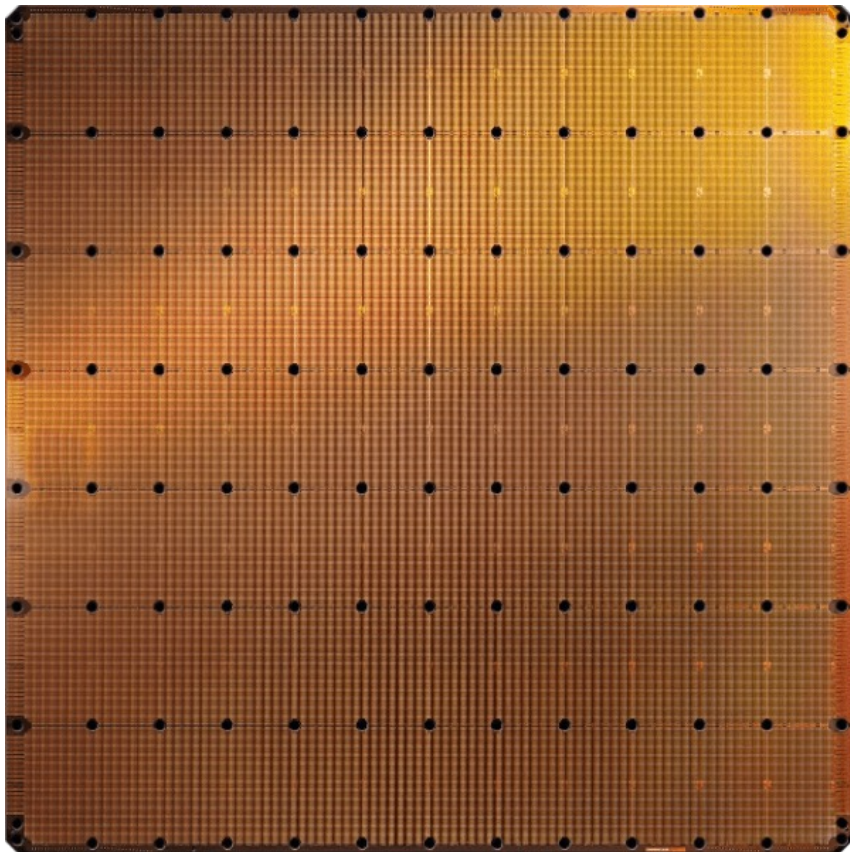
## ■ Scratchpad Memory

- ❑ 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



# Cerebras's Wafer Scale Engine-2 (2021)

---



## **Cerebras WSE-2**

2.6 Trillion transistors

46,225 mm<sup>2</sup>

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

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



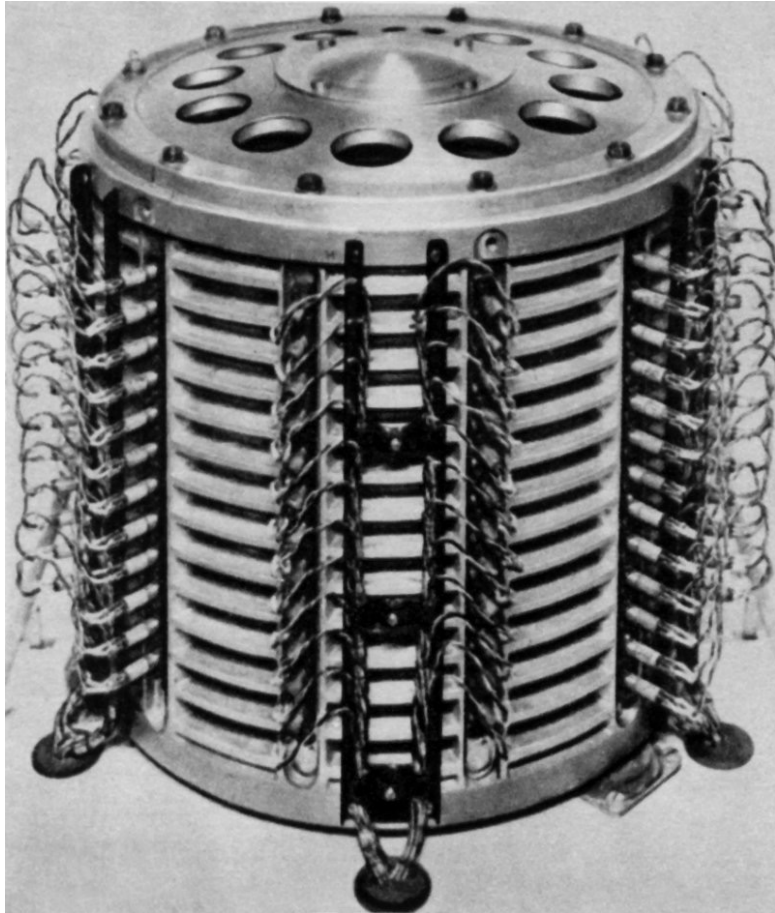
## **Largest GPU**

54.2 Billion transistors

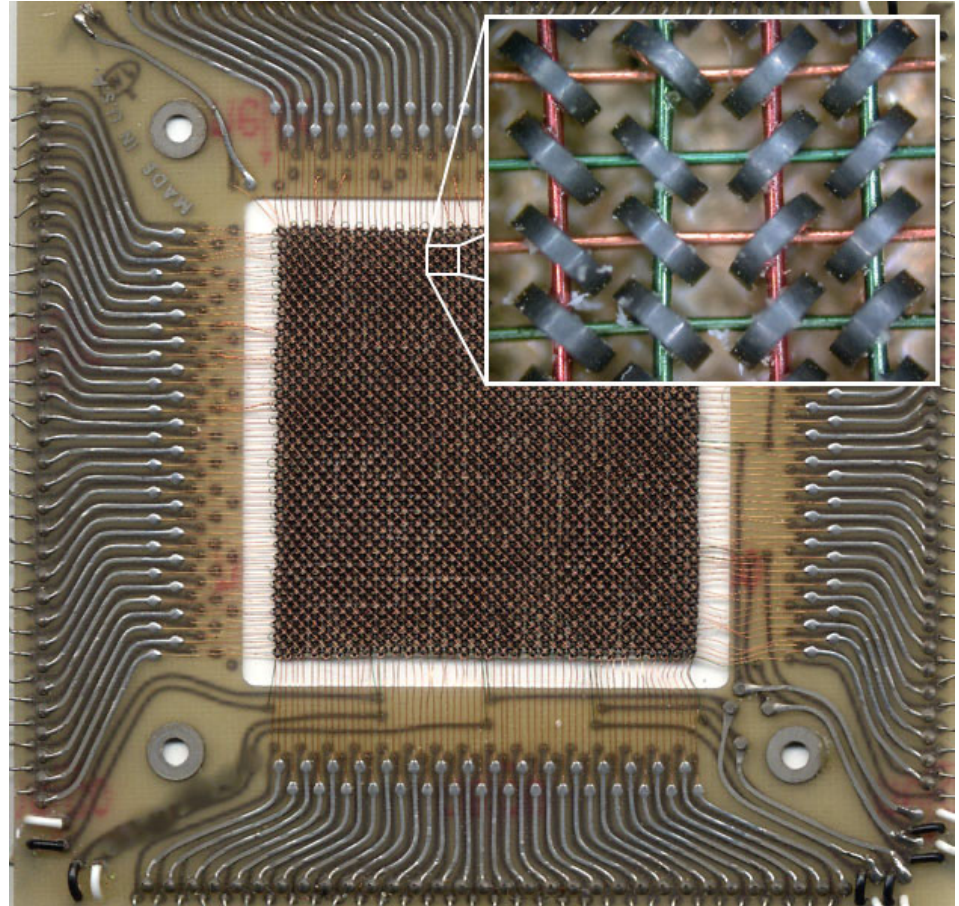
826 mm<sup>2</sup>

NVIDIA Ampere GA100

# A Historical Perspective



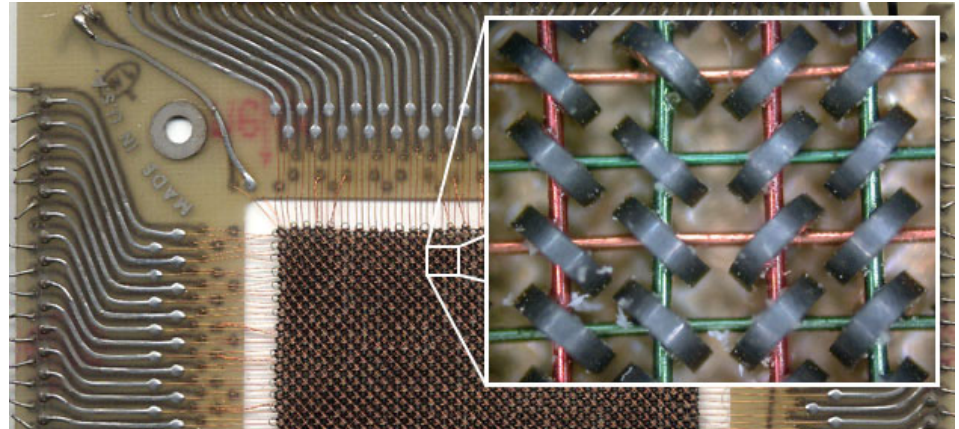
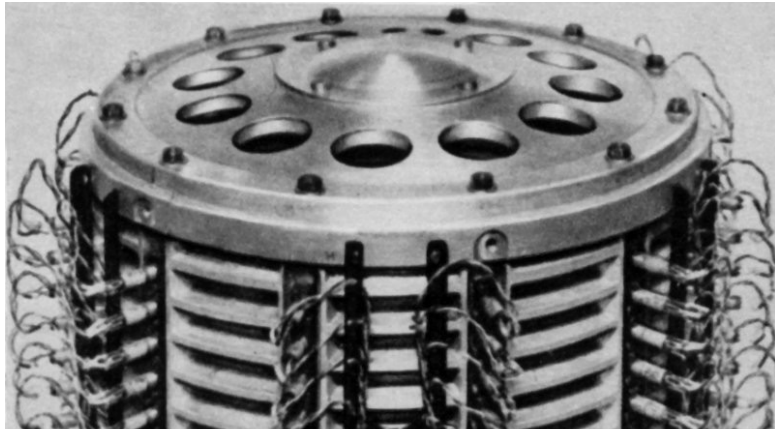
**Magnetic Drum Memory**  
Main Memory of 1950s-1960s



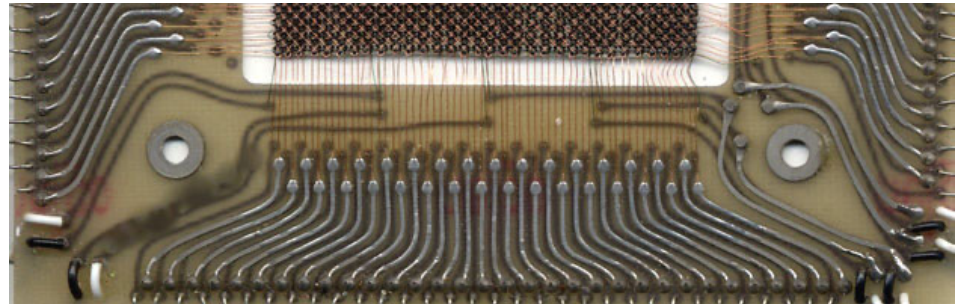
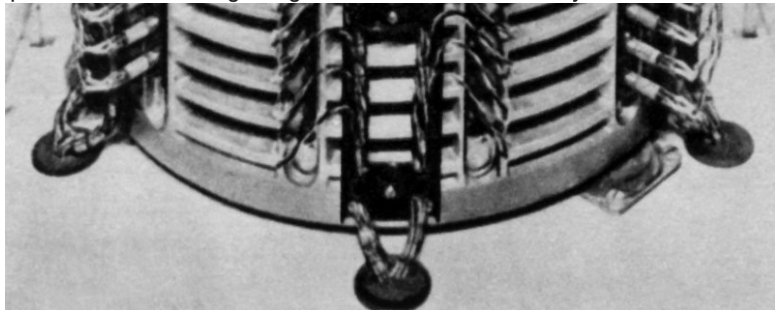
**Magnetic Core Memory**  
Main Memory of 1960s-1970s



# A Historical Perspective



Using smaller cores and wires, the [memory density](#) of core slowly increased, and by the late 1960s a density of about 32 kilobits per cubic foot (about 0.9 kilobits per litre) was typical. However, reaching this density required extremely careful manufacture, which was almost always carried out by hand in spite of repeated major efforts to automate the process. The cost declined over this period from about \$1 per bit to about 1 cent per bit. The introduction of the first [semiconductor memory](#) chips in the late 1960s, which initially created static random-access memory ([SRAM](#)), began to erode the market for core memory. The first successful dynamic random-access memory ([DRAM](#)), the [Intel 1103](#), followed in 1970. Its availability in quantity at 1 cent per bit marked the beginning of the end for core memory.<sup>[1]</sup>



**Magnetic Drum Memory**  
Main Memory of 1950s-1960s

**Magnetic Core Memory**  
Main Memory of 1960s-1970s

# 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.

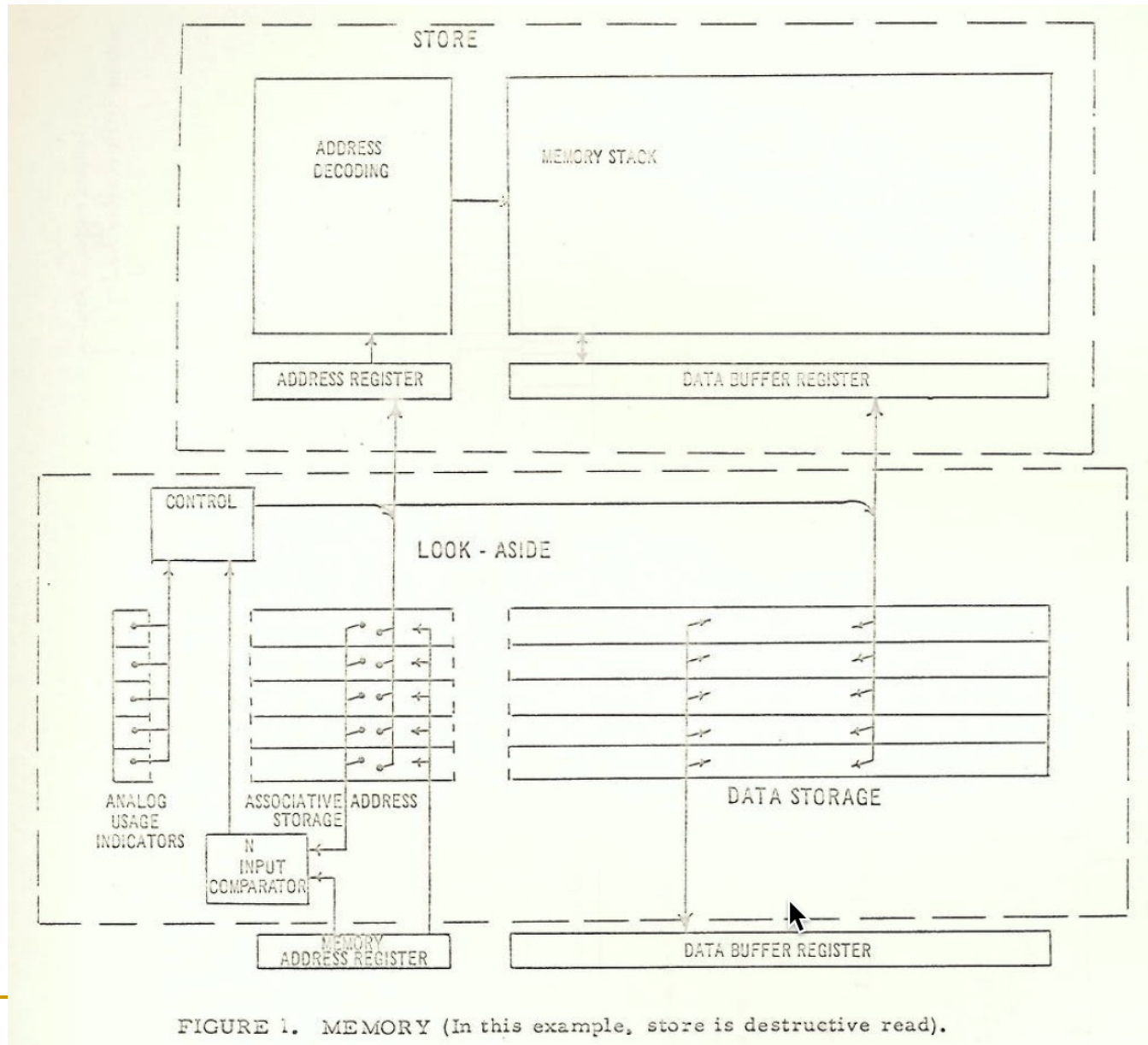
- “By a slave memory I mean one which **automatically accumulates to itself words** that come from a slower main memory, and keeps them available for subsequent use without it being necessary for the penalty of main memory access to be incurred again.”

# Historical Aside: Other Cache Papers

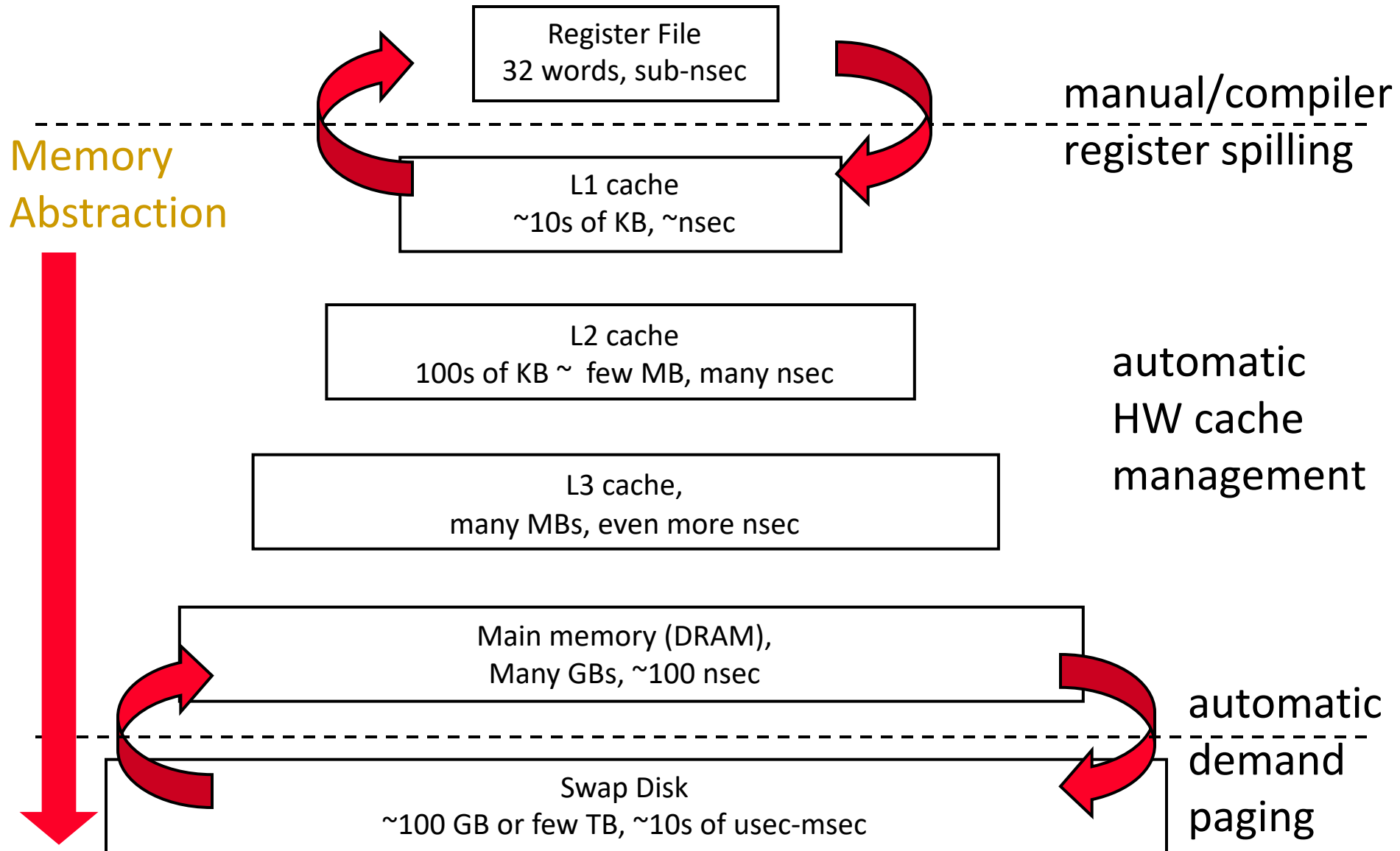
---

- 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)



# A Modern Memory Hierarchy





# Hierarchical Latency Analysis

---

- A given memory hierarchy level  $i$  has **intrinsic access time** of  $t_i$
- It also has **perceived access time**  $T_i$  that is longer than  $t_i$
- Except for the outer-most hierarchy level, 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 only the references that missed at  $L_{i-1}$

# Hierarchy Design Considerations

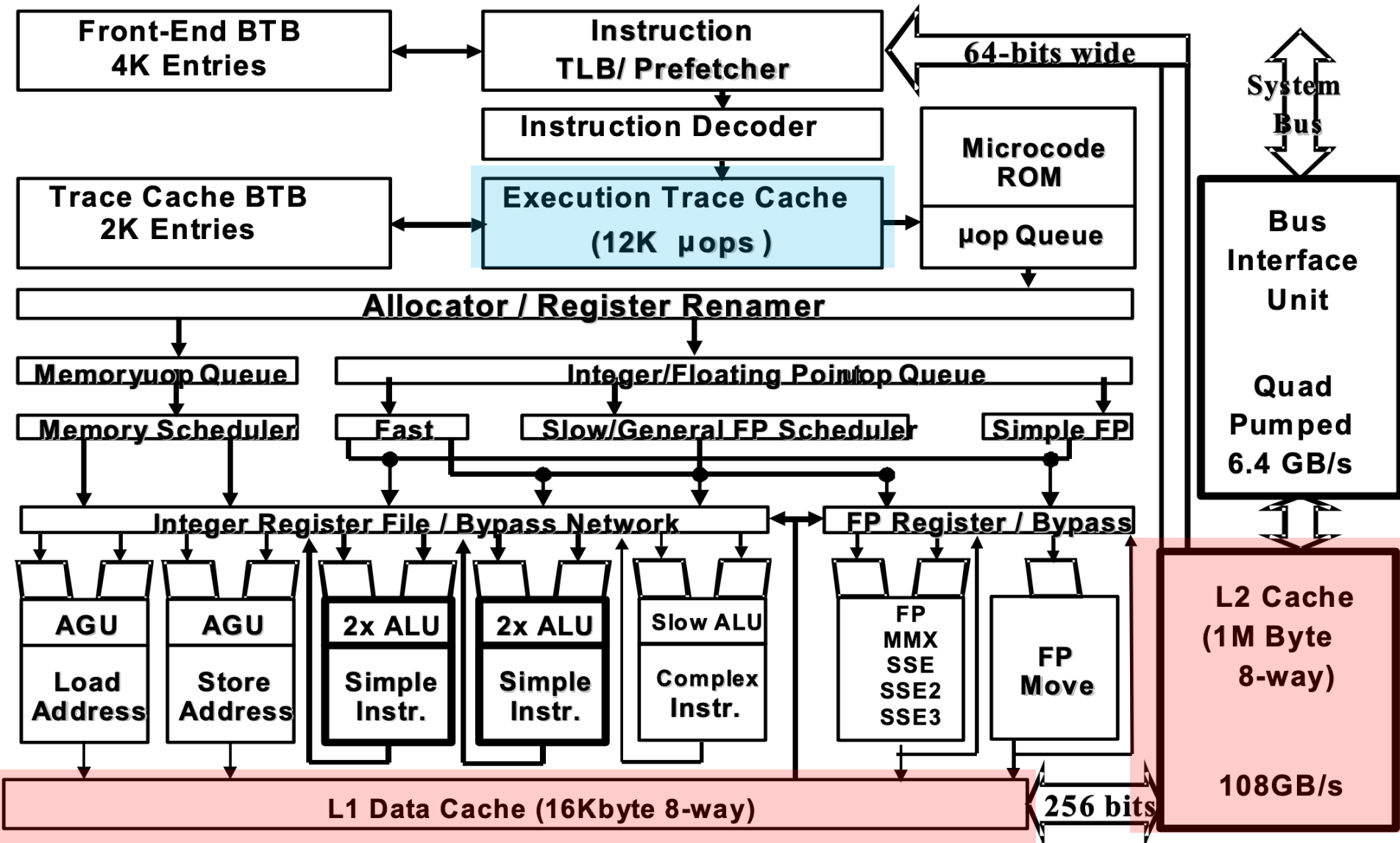
---

- Recursive latency equation

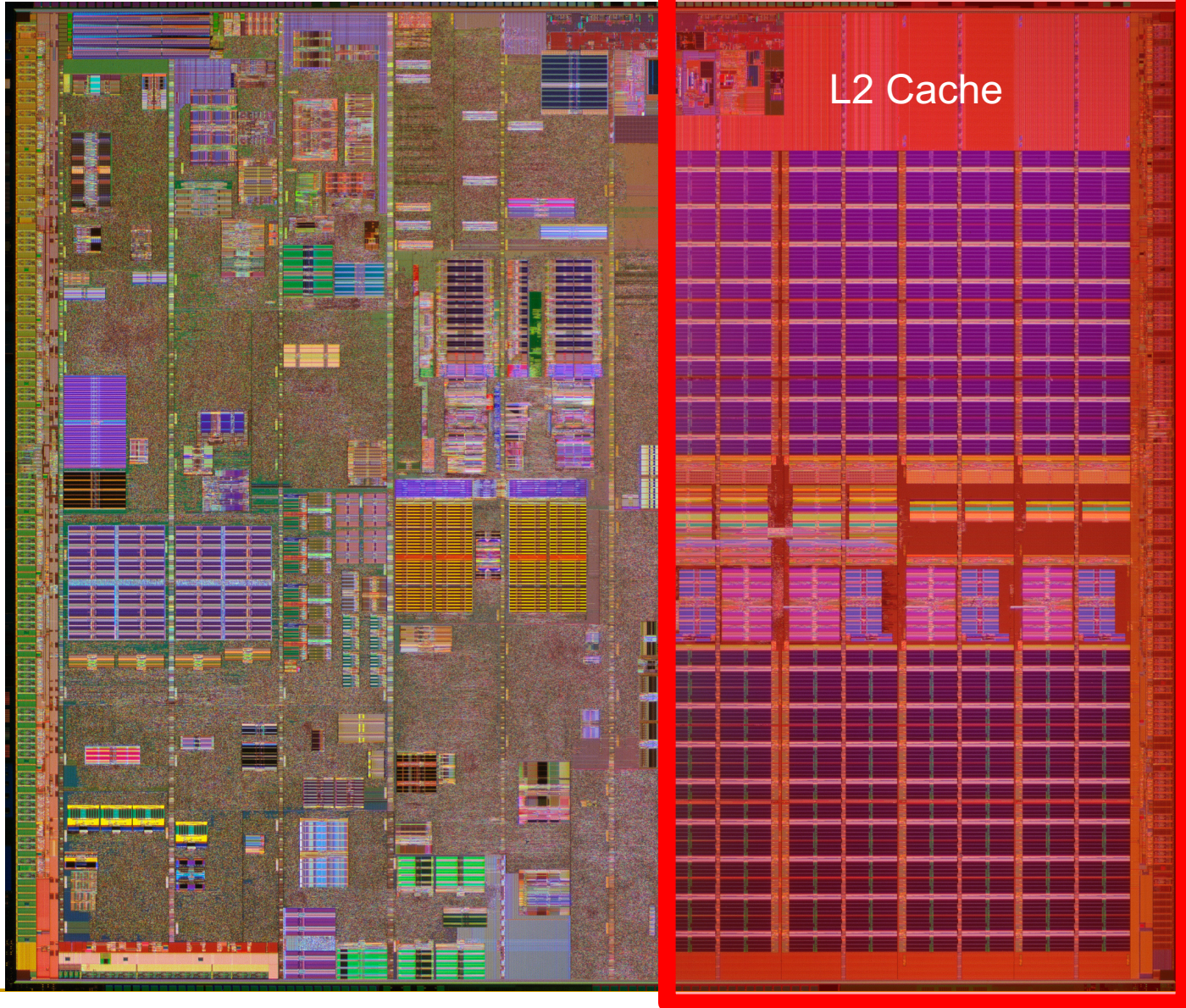
$$T_i = t_i + m_i \cdot T_{i+1}$$

- 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 outer hierarchy levels can help, but beware of increasing cost
  - introduce intermediate hierarchy levels as a compromise

# Intel Pentium 4 Example



# Intel Pentium 4 Example





# 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
  - $t_3 = \sim 50$ ns or 180 cyc
- Notice
  - best case latency is not 1
  - worst case access latencies are into 500+ cycles

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

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

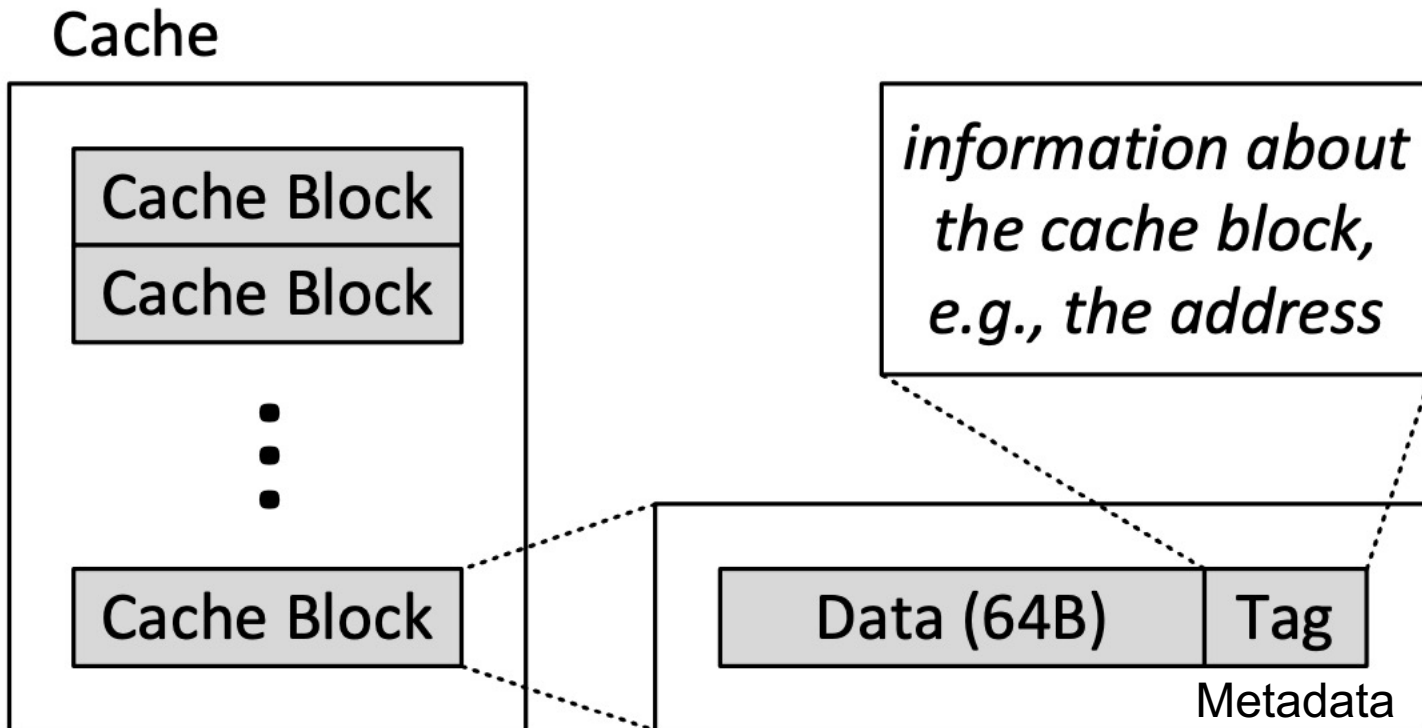
# Cache

---

- Any structure that “memoizes” 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

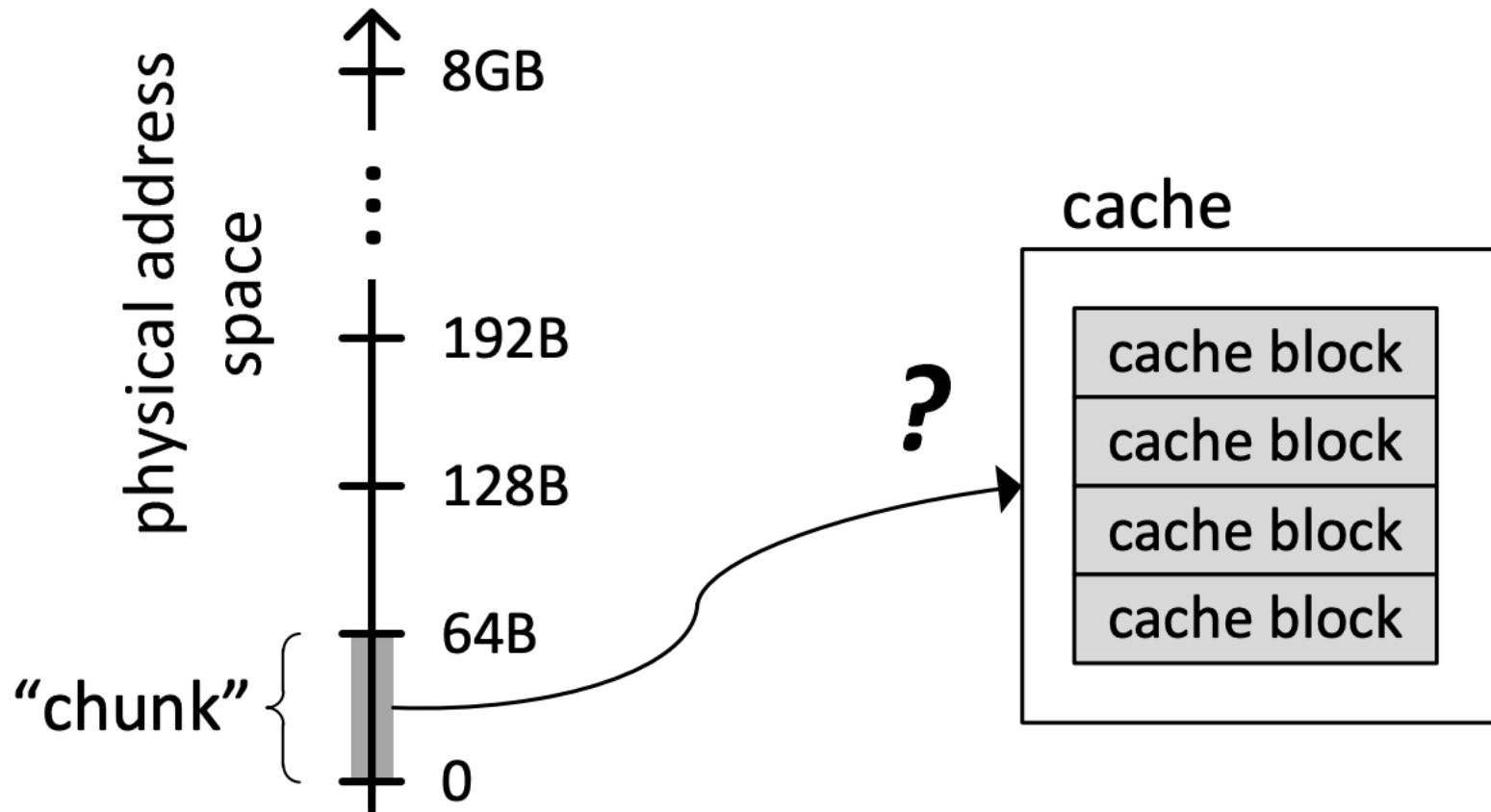
---



Kim & Mutlu, “[Memory Systems](https://people.inf.ethz.ch/omutlu/pub/memory-systems-introduction_computing-handbook14.pdf),” Computing Handbook, 2014  
[https://people.inf.ethz.ch/omutlu/pub/memory-systems-introduction\\_computing-handbook14.pdf](https://people.inf.ethz.ch/omutlu/pub/memory-systems-introduction_computing-handbook14.pdf)

# Logical Organization of a Cache (I)

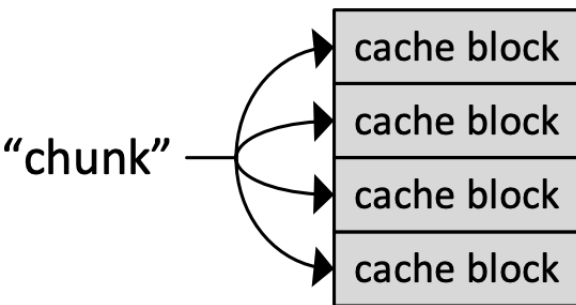
- A key question: How to map chunks of the main memory address space to blocks in the cache?
  - Which location in cache can a given “main memory chunk” be placed in?



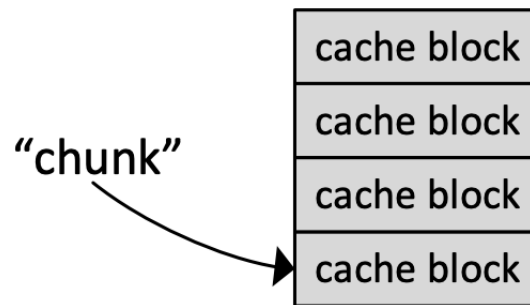
# Logical Organization of a Cache (II)

- 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?

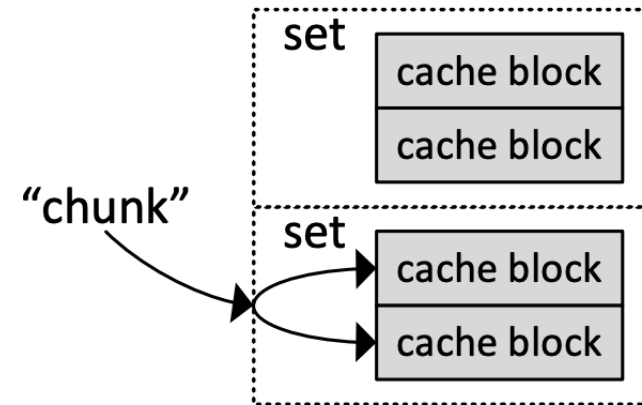
**fully-associative**



**direct-mapped**



**set-associative**





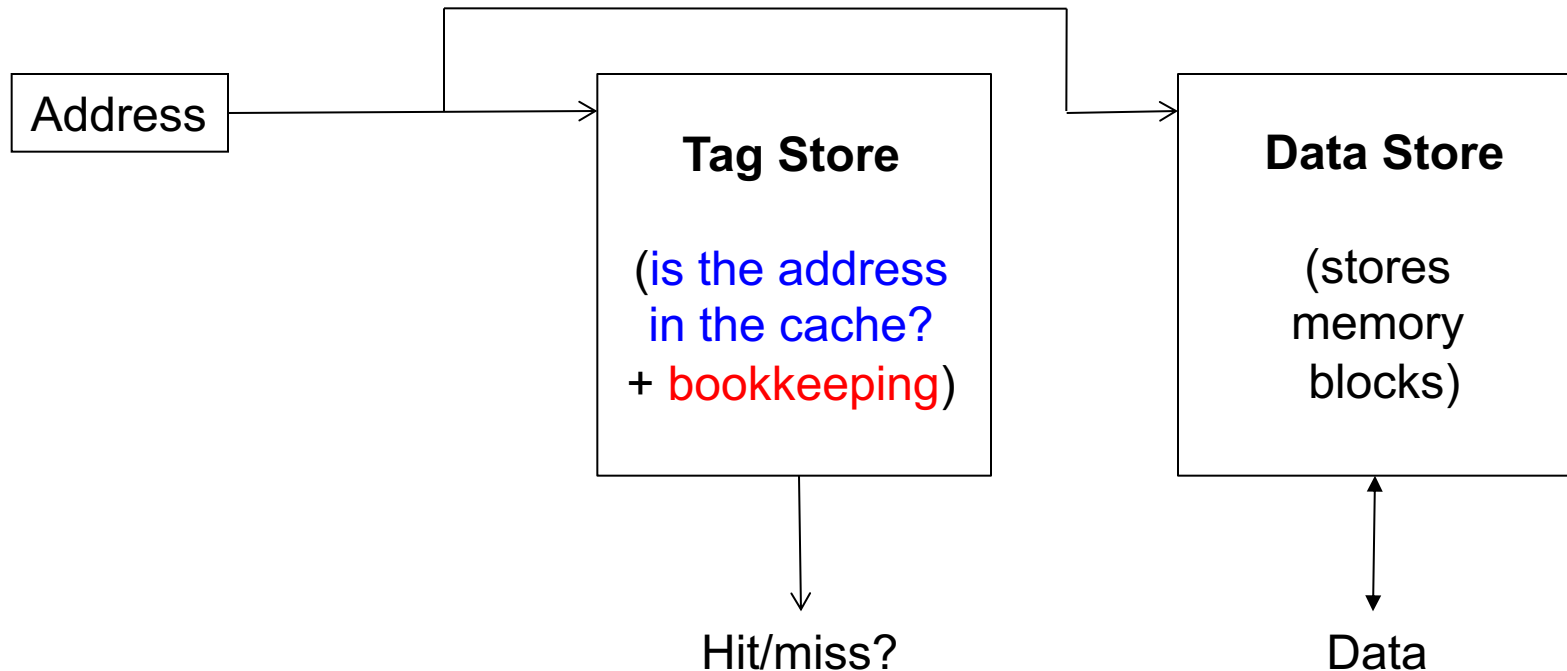
# Caching Basics

---

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

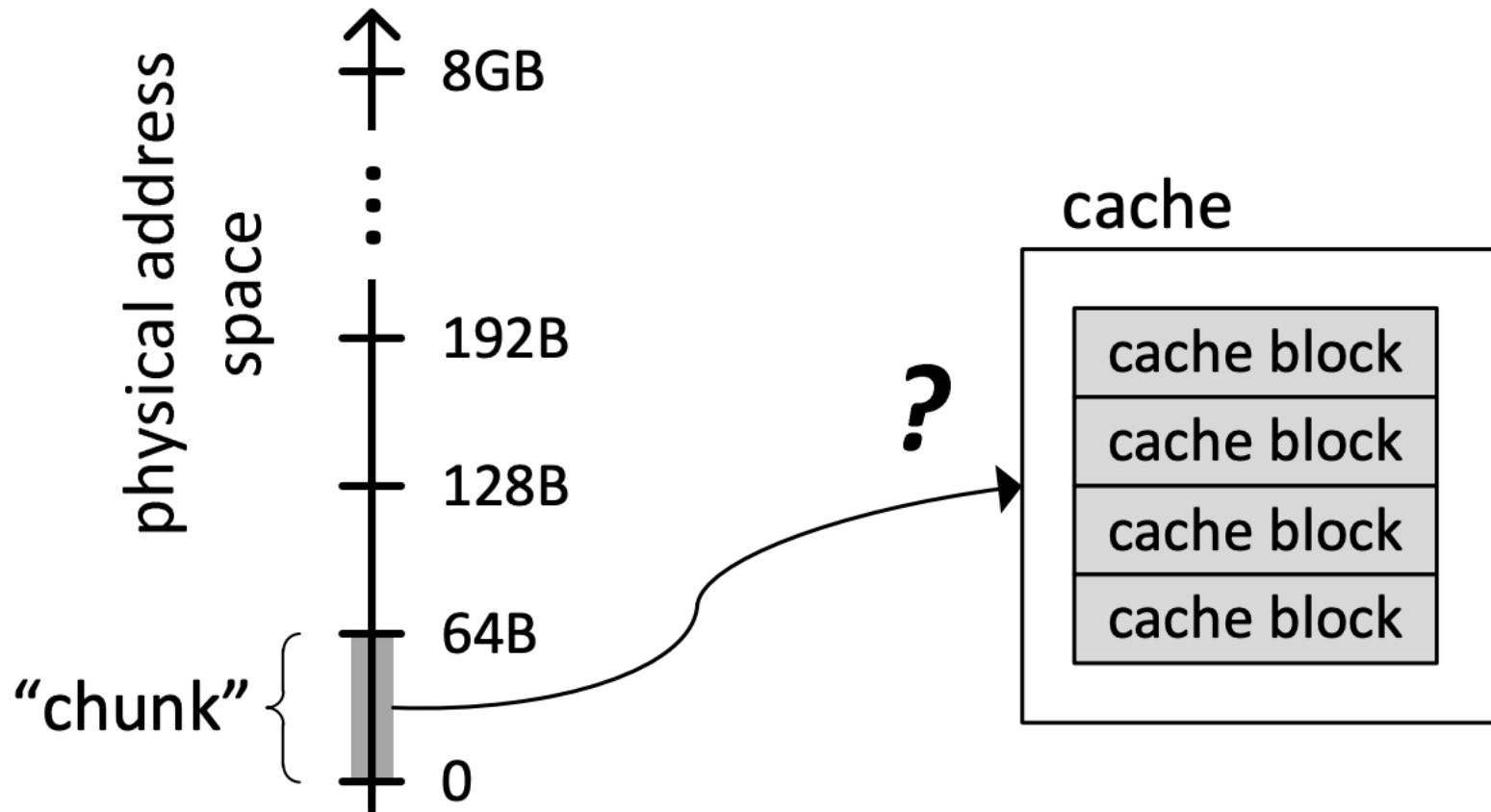
# A Basic Hardware Cache Design

---

- 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

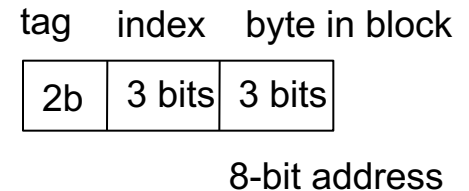
- Main memory logically divided into fixed-size chunks (**blocks**)
- **Cache** can house only a **limited** number of blocks



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

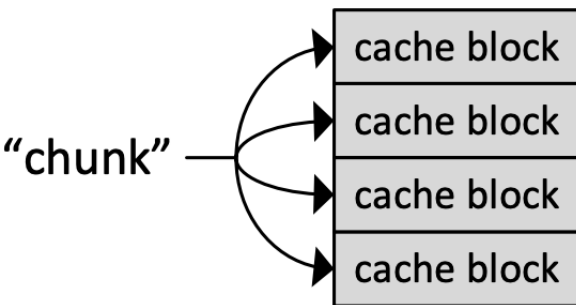




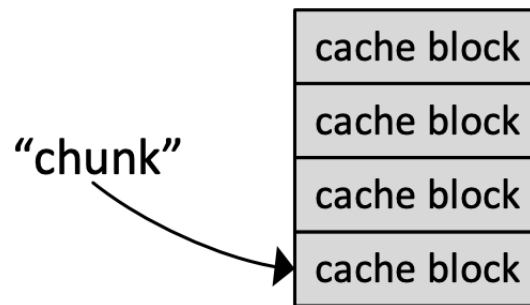
# Let's See A Toy Example

- 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

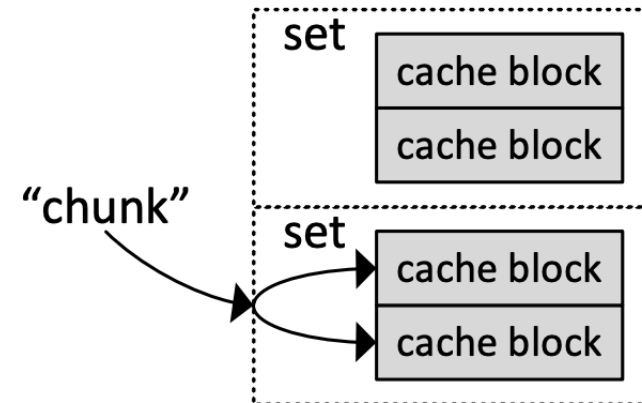
**fully-associative**



**direct-mapped**



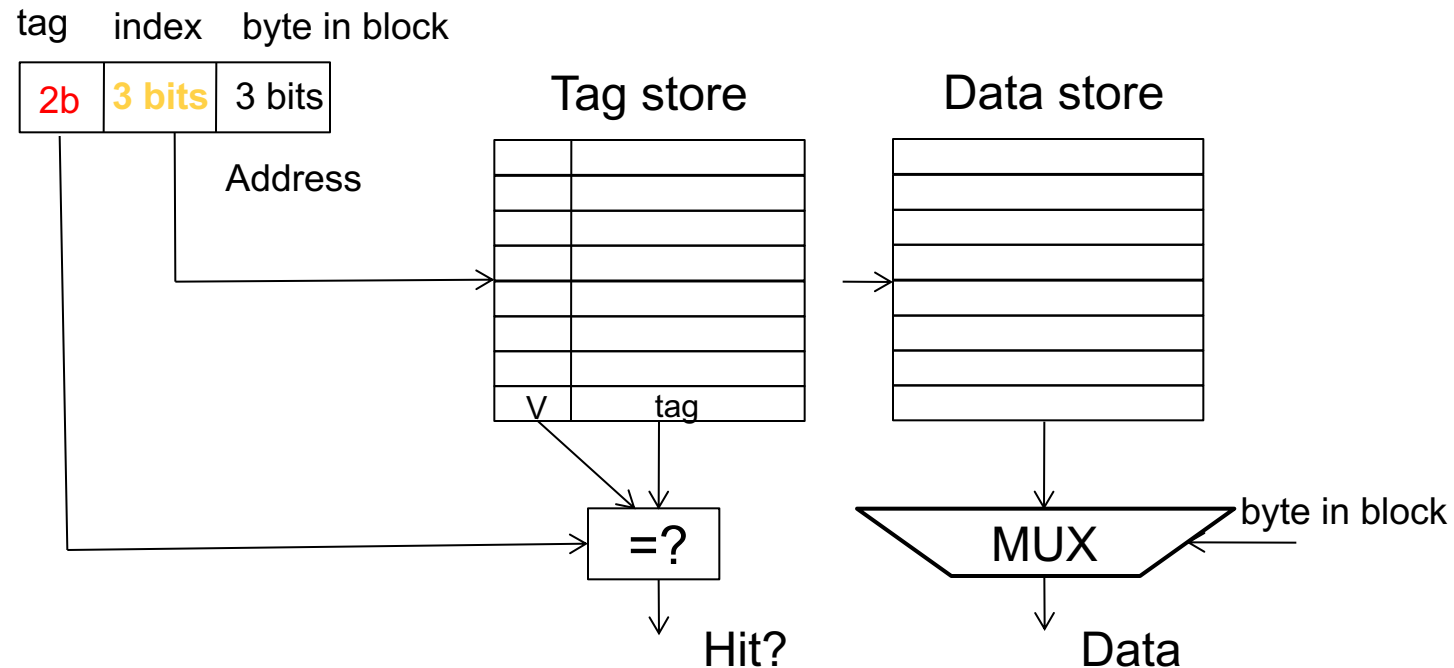
**set-associative**



# Direct-Mapped Cache: Placement and Access

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

- 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

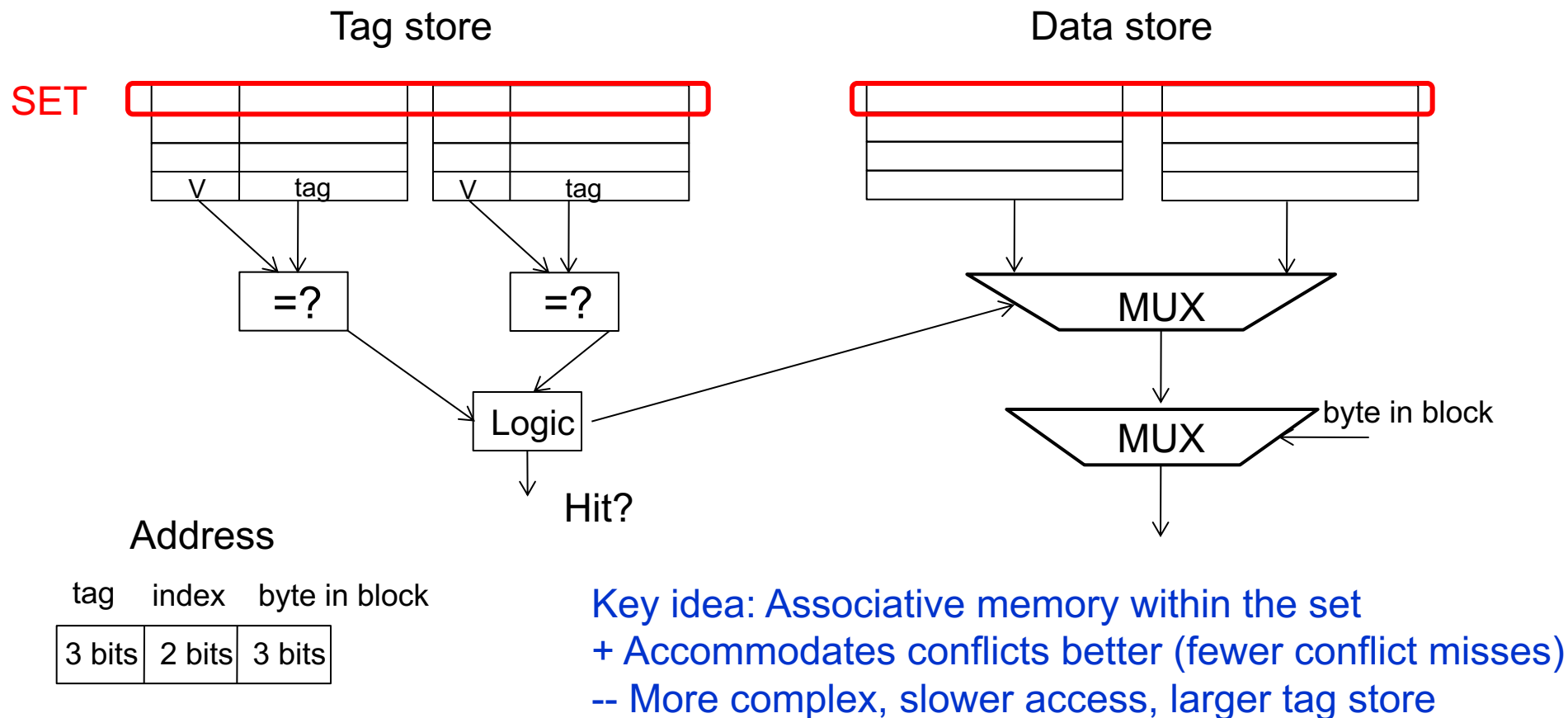
# Direct-Mapped Caches

---

- **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  $\rightarrow$  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, ...  $\rightarrow$  conflict in the cache index
  - All accesses are **conflict misses**

# Set Associativity

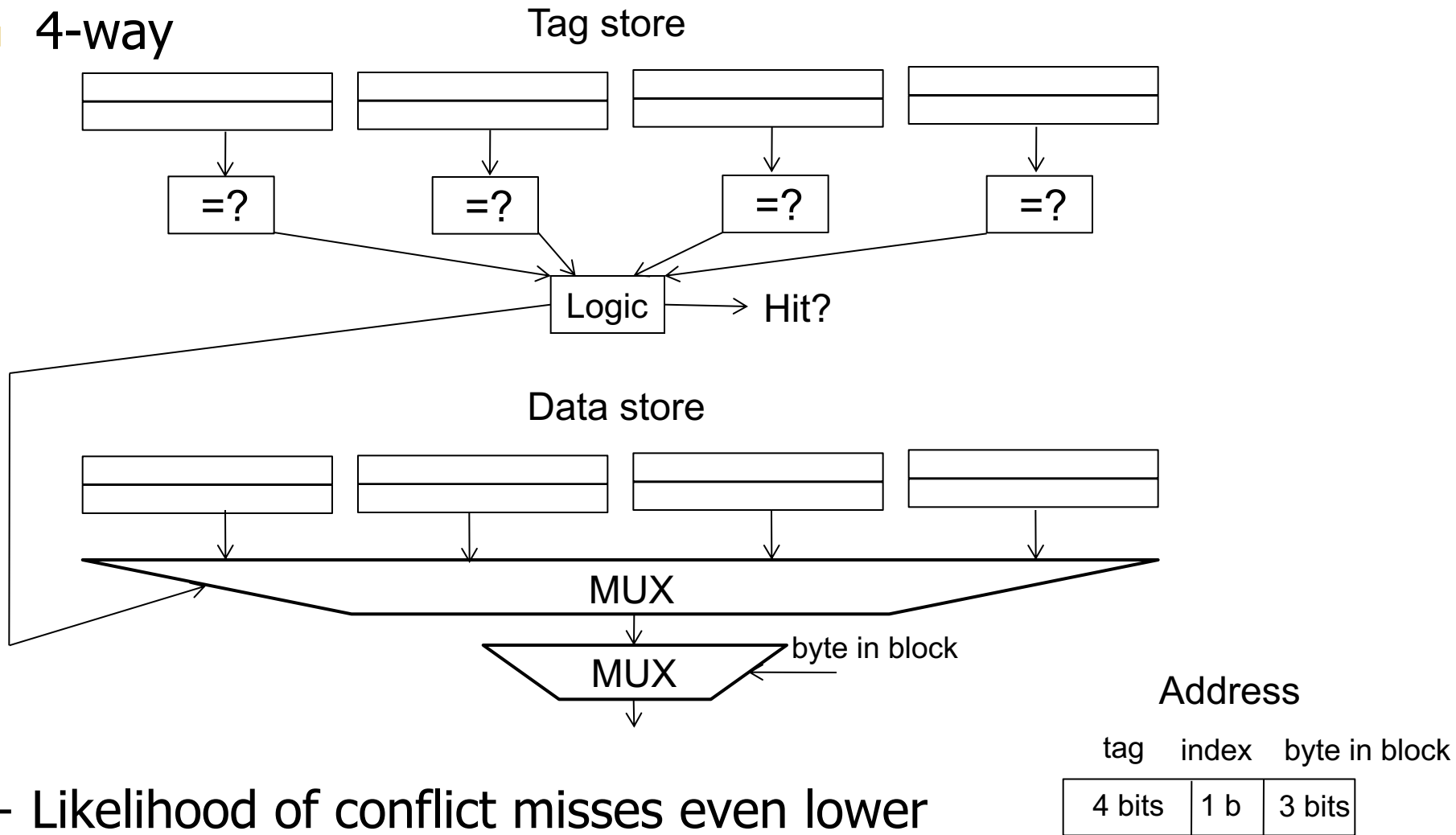
- Problem: Addresses  $N$  and  $N+8$  always conflict in direct mapped cache
- Idea: enable blocks with the same index to map to  $> 1$  cache location
- Example: Instead of having one column of 8, have 2 columns of 4 blocks



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

# Higher Associativity

## ■ 4-way



+ Likelihood of conflict misses even lower

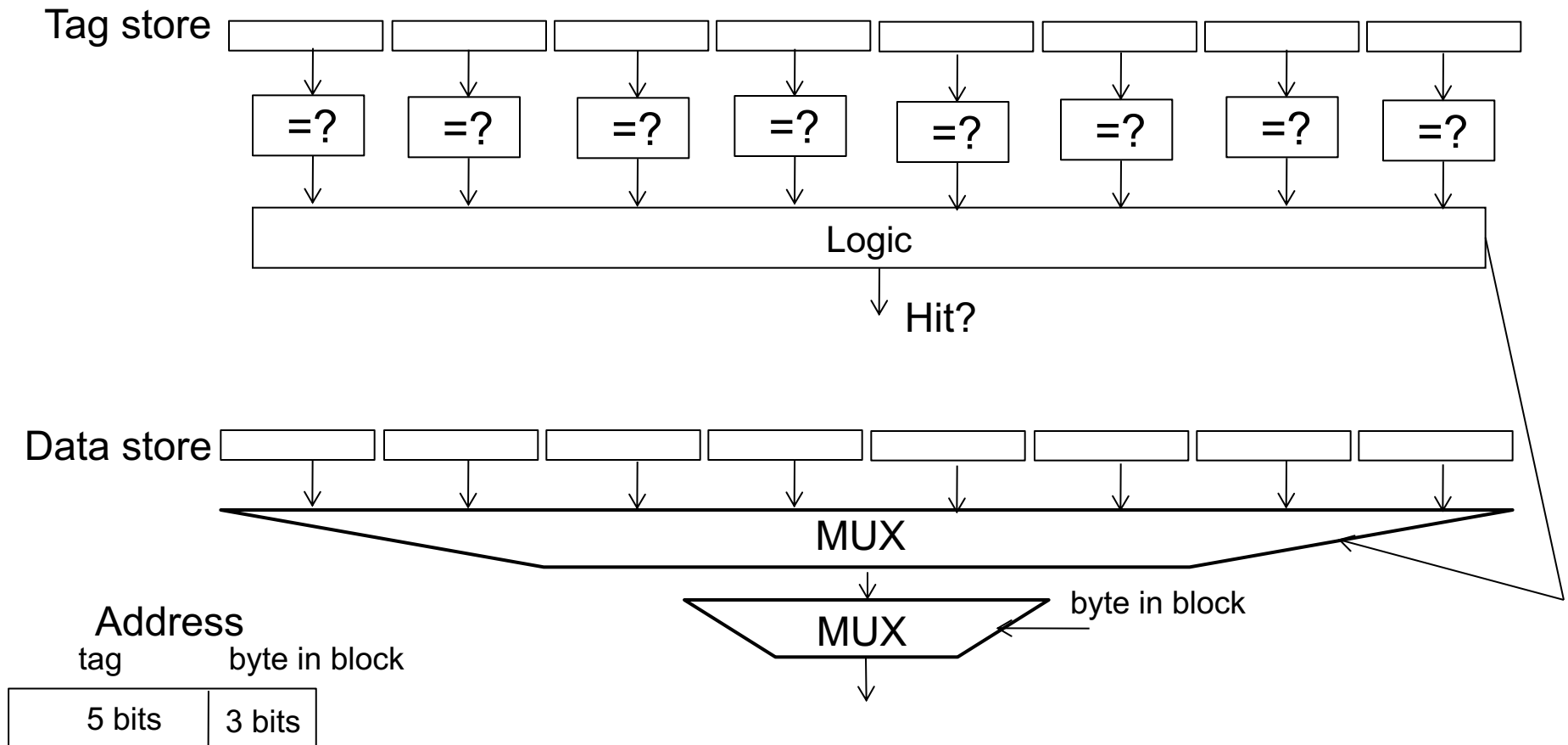
-- More tag comparators and wider data mux; larger tags

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



# Full Associativity

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

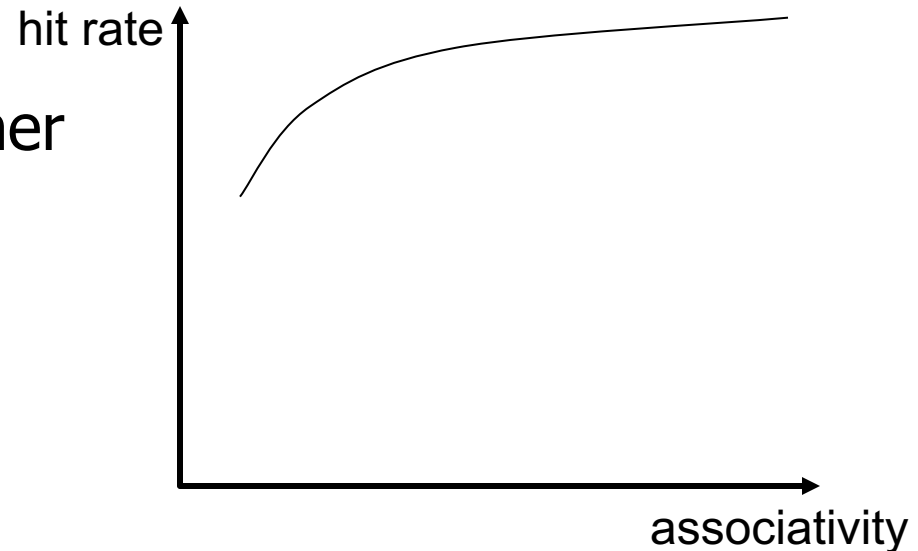


Fully associative cache: Any block can map to any location

# 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 2022  
20 May 2022



# Basic Cache Examples: For You to Study

# Cache Terminology

---

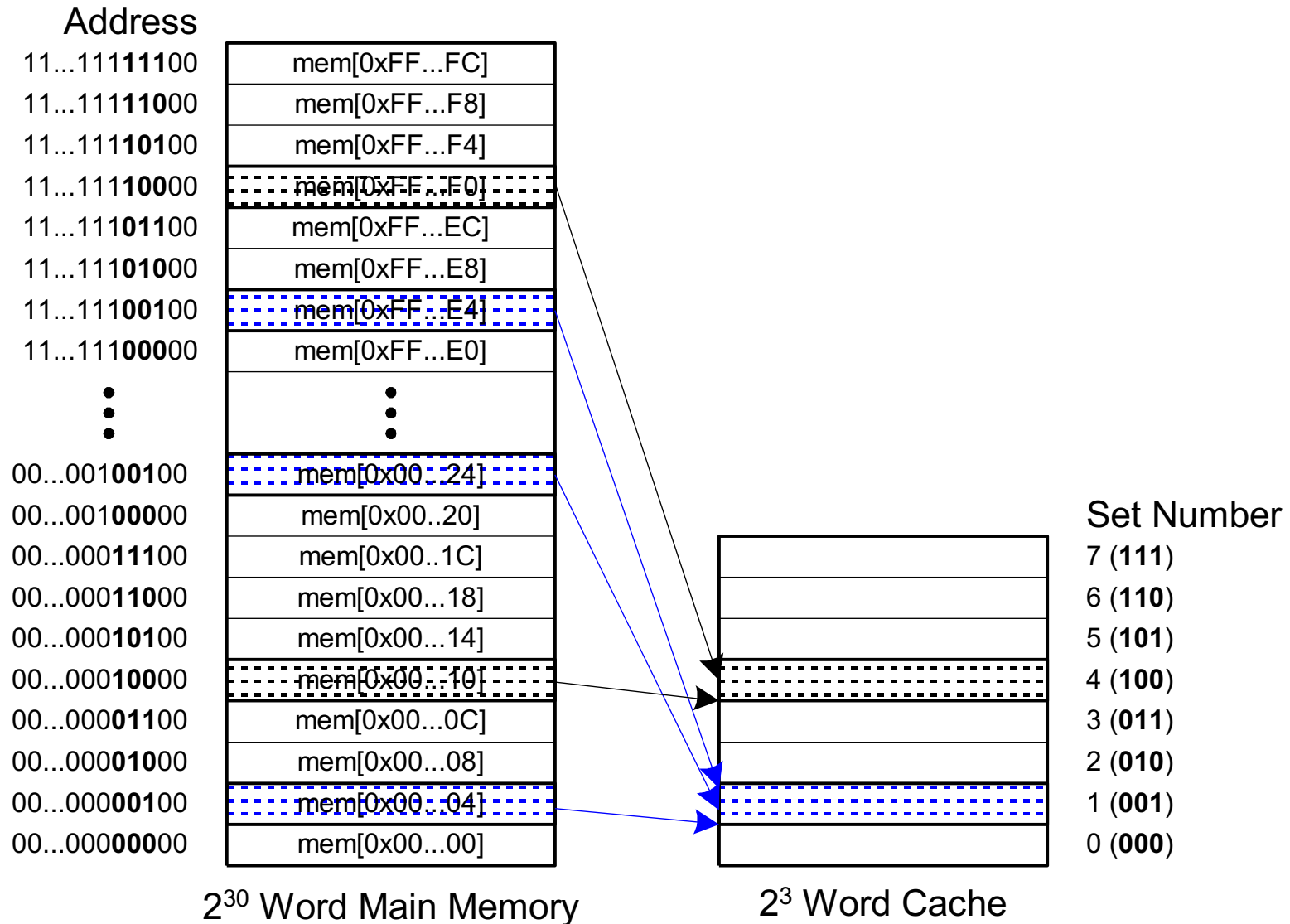
- Capacity ( $C$ ):
  - the number of data bytes a cache stores
- Block size ( $b$ ):
  - bytes of data brought into cache at once
- Number of blocks ( $B = C/b$ ):
  - number of blocks in cache:  $B = C/b$
- Degree of associativity ( $M$ ):
  - number of blocks in a set
- Number of sets ( $S = B/M$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

# How is data found?

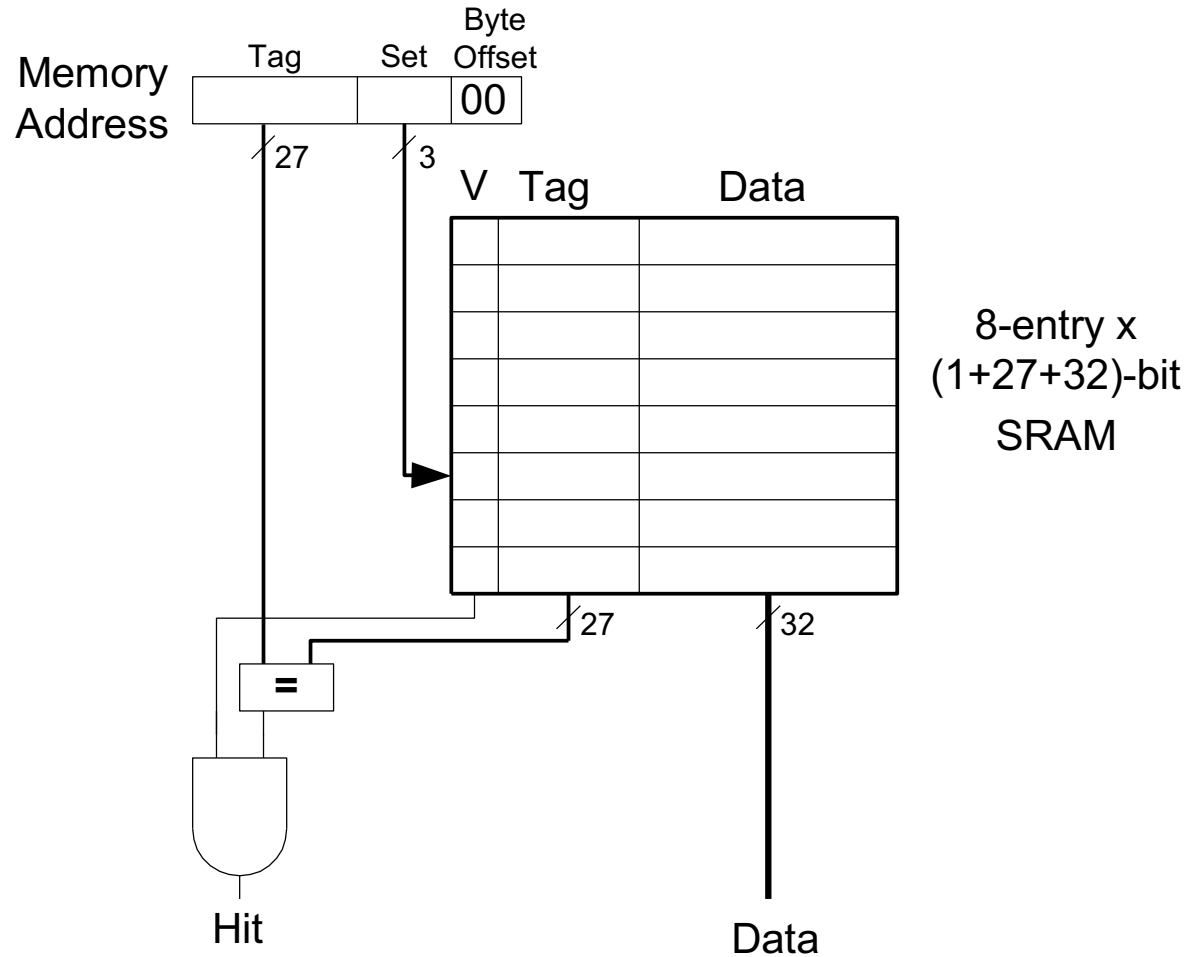
---

- 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

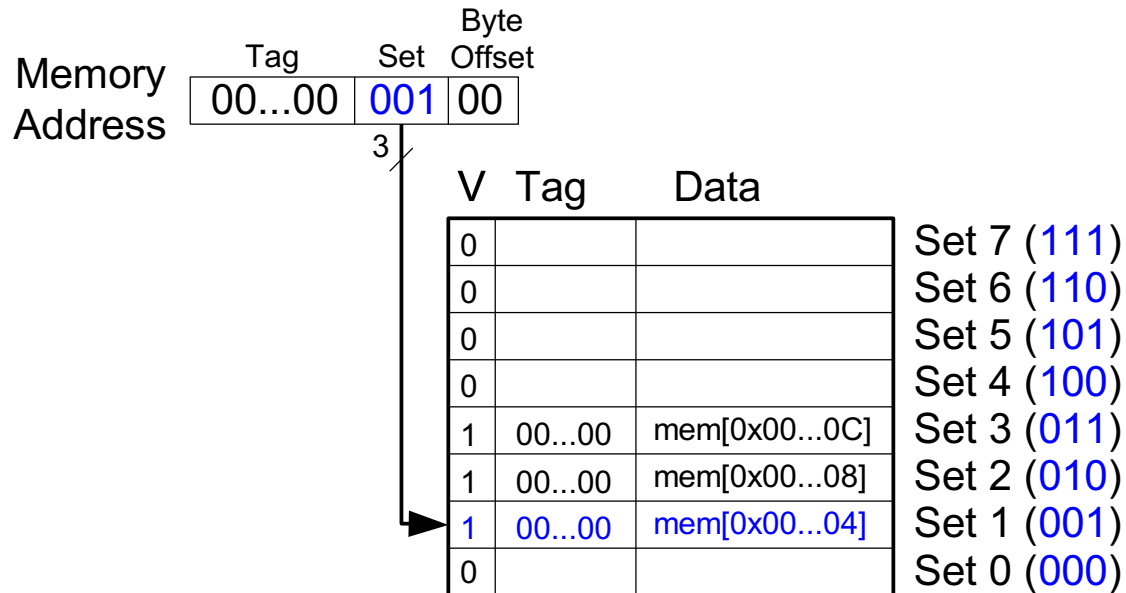


# Direct Mapped Cache Hardware





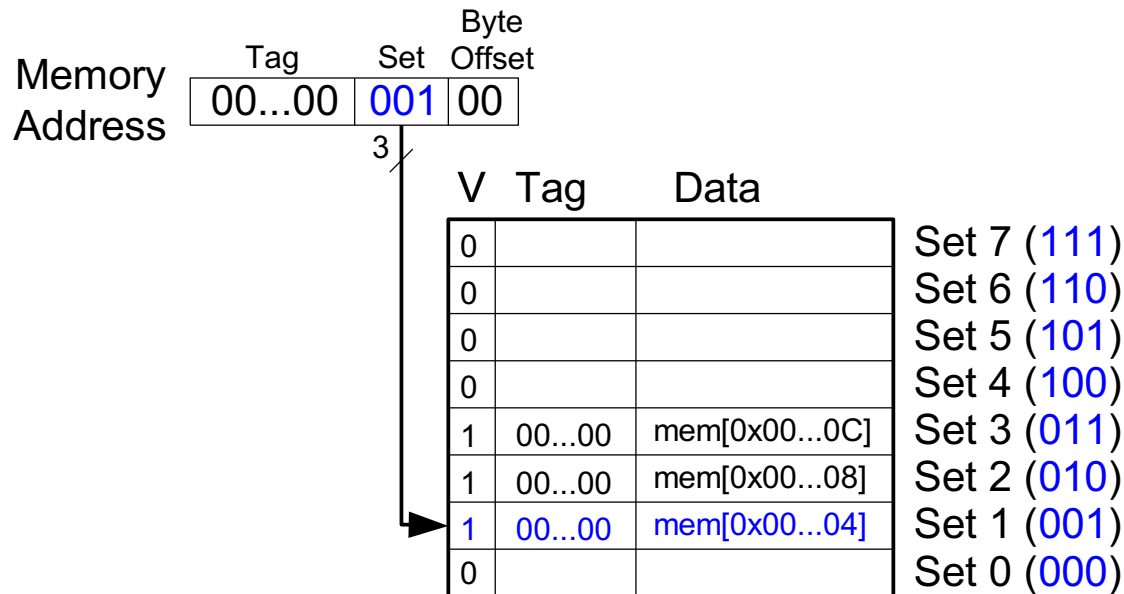
# Direct Mapped Cache Performance



```
# 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



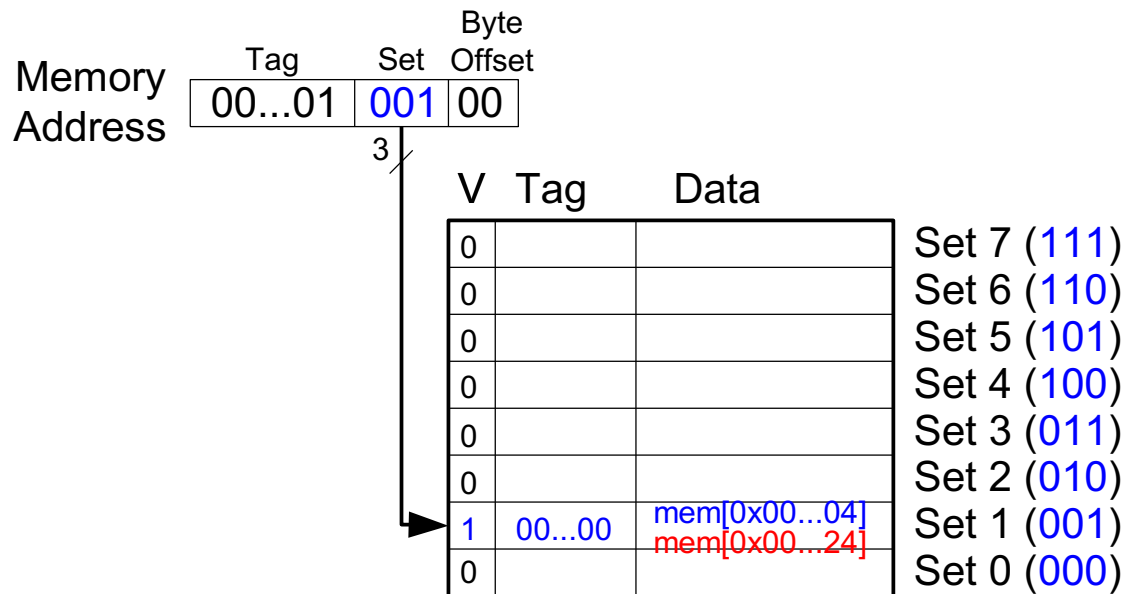
```
# MIPS assembly code
      addi $t0, $0, 5
loop: beq  $t0, $0, done
      lw   $t1, 0x4($0)
      lw   $t2, 0xC($0)
      lw   $t3, 0x8($0)
      addi $t0, $t0, -1
      j    loop
done:
```

$$\text{Miss Rate} = 3/15 =$$

20%

Temporal Locality  
Compulsory Misses

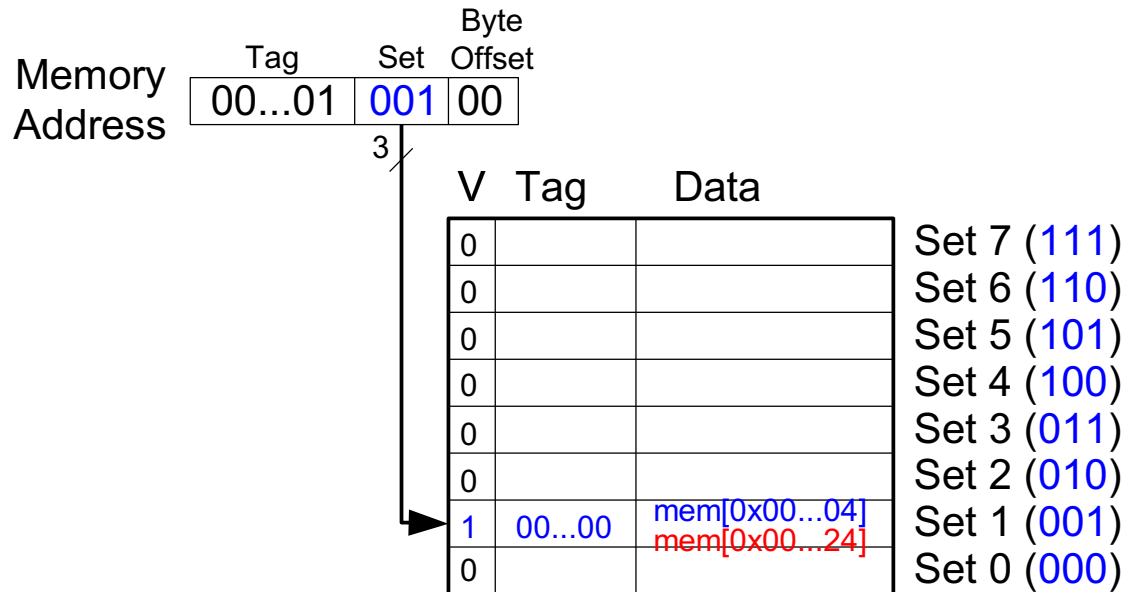
# Direct Mapped Cache: Conflict



```
# 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

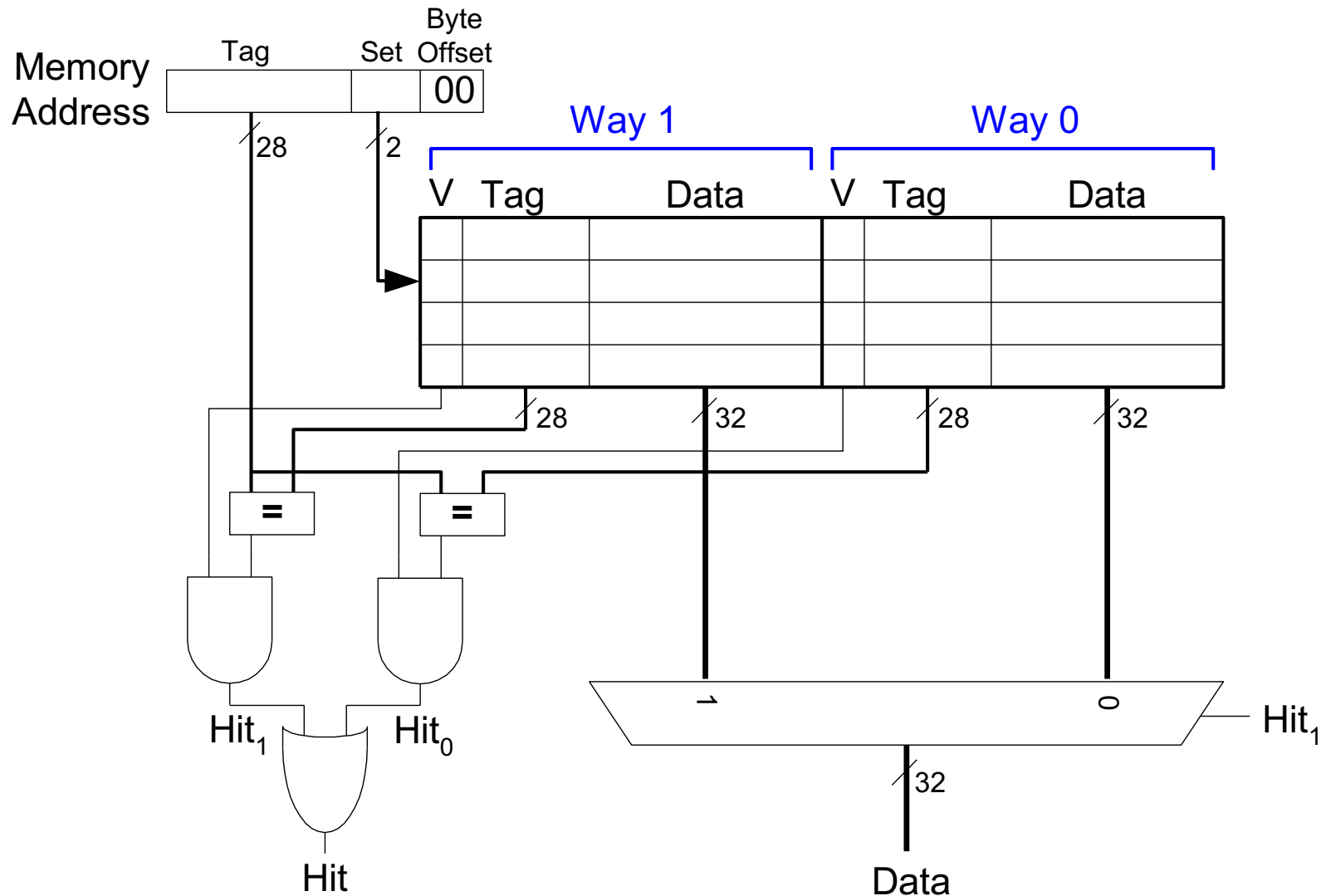


```
# MIPS assembly code
      addi $t0, $0, 5
loop: beq  $t0, $0, done
      lw   $t1, 0x4($0)
      lw   $t2, 0x24($0)
      addi $t0, $t0, -1
      j    loop
done:
```

$$\text{Miss Rate} = 10/10 = 100\%$$

**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 =*

Way 1

Way 0

Way 1			Way 0			
V	Tag	Data	V	Tag	Data	
0			0			Set 3
0			0			Set 2
1	00...10	mem[0x00...24]	1	00...00	mem[0x00...04]	Set 1
0			0			Set 0



# 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

Way 1

Way 0

Way 1			Way 0			
V	Tag	Data	V	Tag	Data	
0			0			Set 3
0			0			Set 2
1	00...10	mem[0x00...24]	1	00...00	mem[0x00...04]	Set 1
0			0			Set 0

# Fully Associative Cache

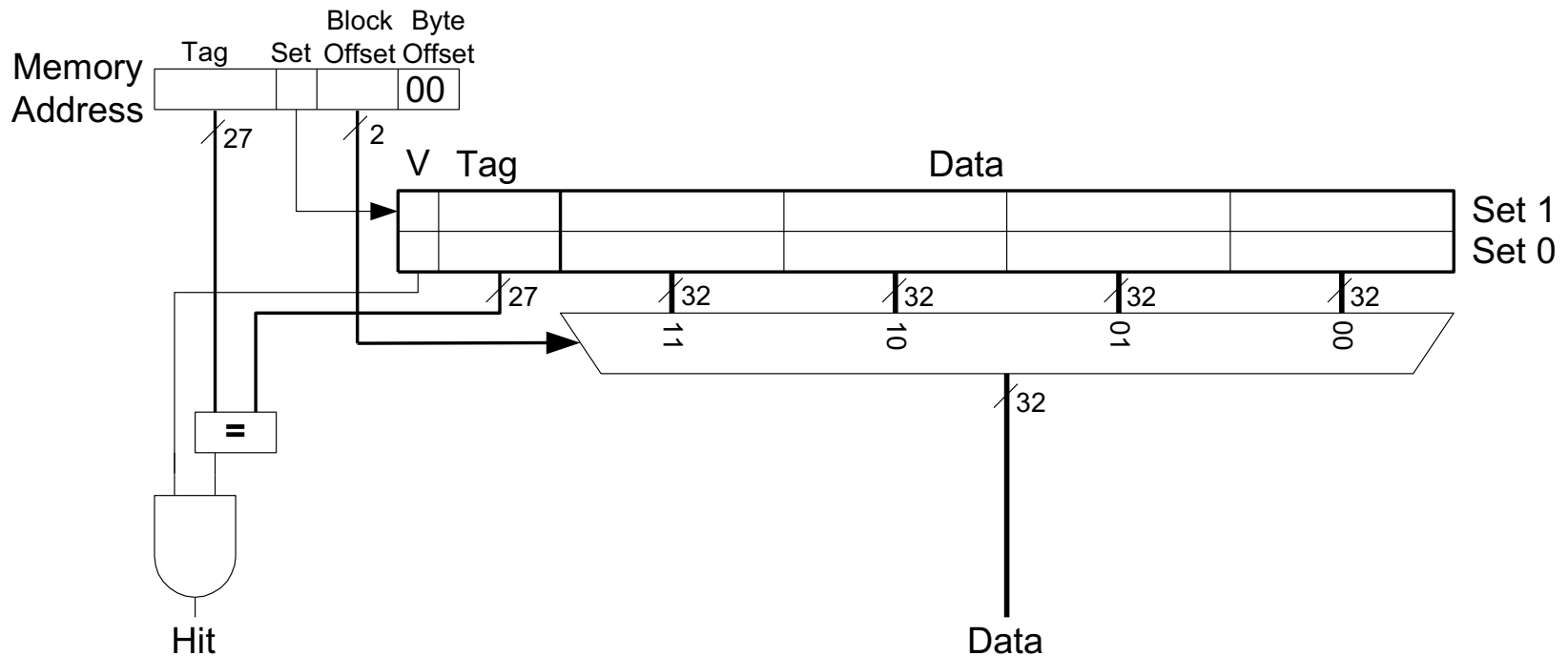
---

- No conflict misses
- Expensive to build

V	Tag	Data	V	Tag	Data	V	Tag	Data	V	Tag	Data	V	Tag	Data	V	Tag	Data	V	Tag	Data

# Spatial Locality?

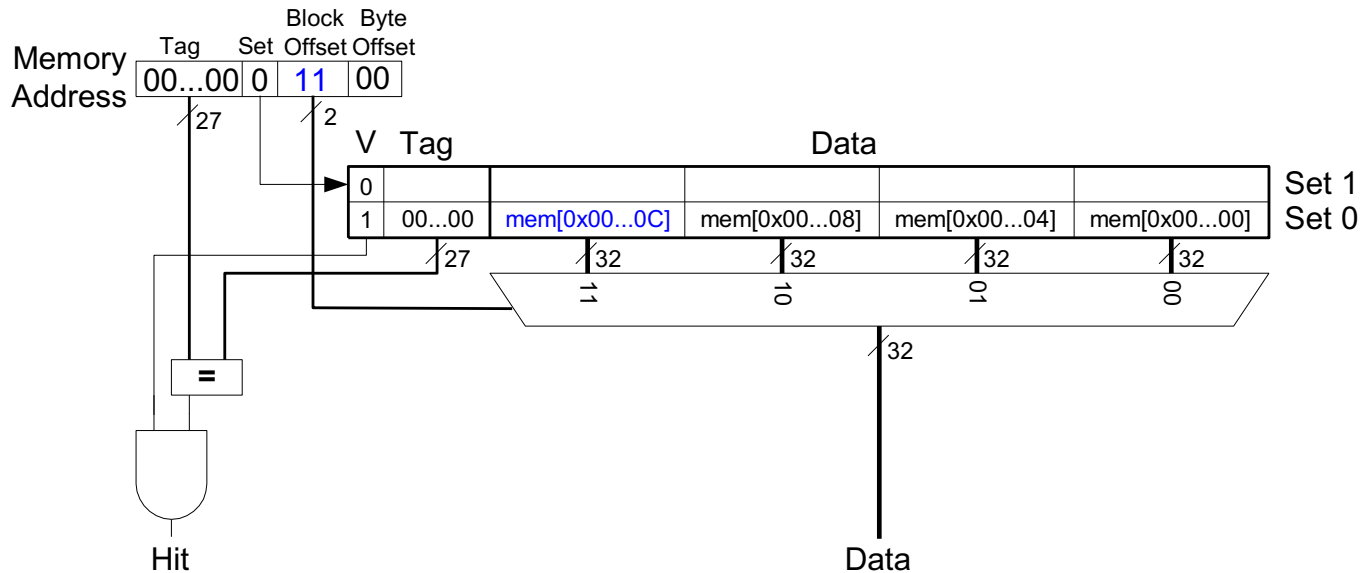
- Increase block size:
  - ❑ Block size,  $b = 4$  words
  - ❑  $C = 8$  words
  - ❑ Direct mapped (1 block per set)
  - ❑ Number of blocks,  $B = C/b = 8/4 = 2$



# Direct Mapped Cache Performance

```
loop:    addi $t0, $0, 5
        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 =*

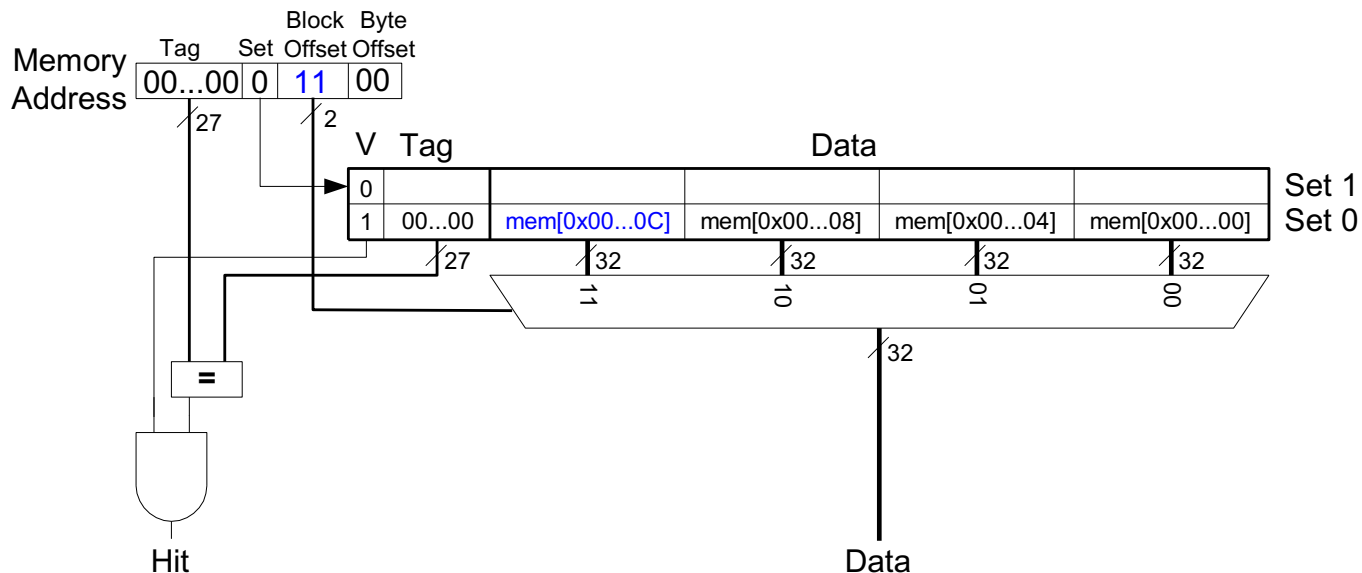


# Direct Mapped Cache Performance

```
loop:    addi $t0, $0, 5
        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)
```

(a)

V	U	Tag	Data	V	Tag	Data	Set Number
							3 (11)
							2 (10)
							1 (01)
							0 (00)

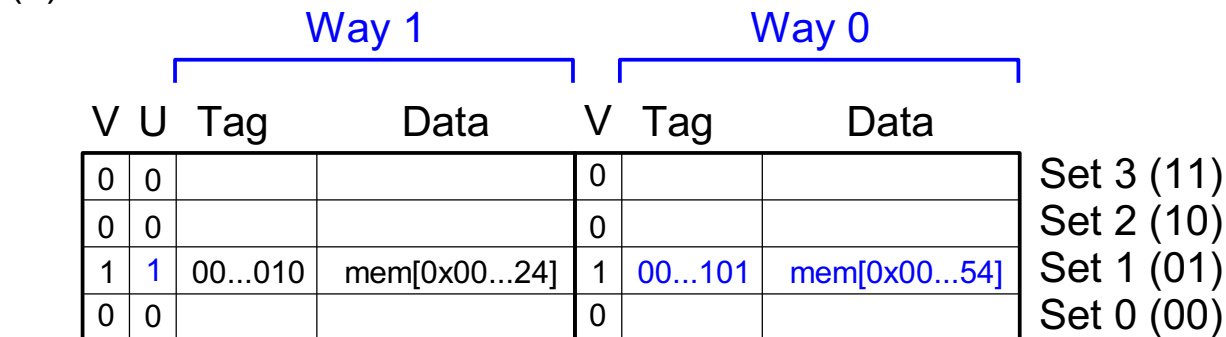
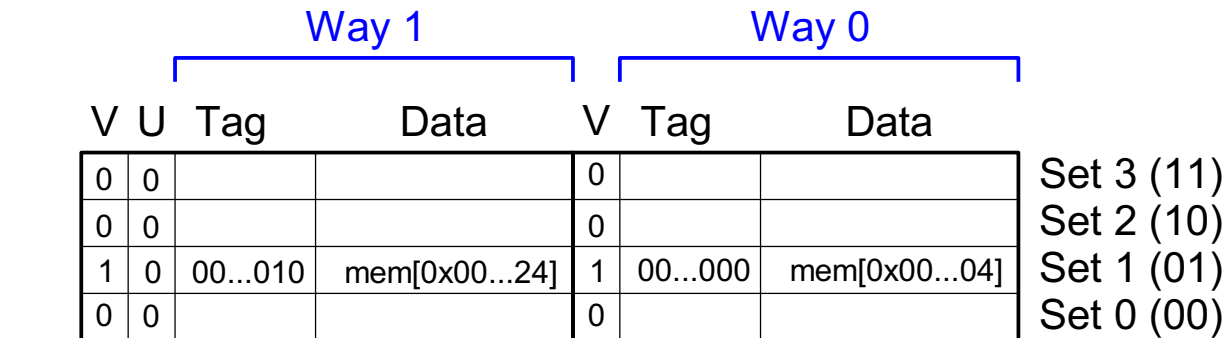
(b)

V	U	Tag	Data	V	Tag	Data	Set Number
							3 (11)
							2 (10)
							1 (01)
							0 (00)

# LRU Replacement

```
# MIPS assembly
```

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



We Will Cover The Remaining Slides  
in Future Lectures.  
They Are for Your Benefit.

# 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:
  - 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: 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 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 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

---

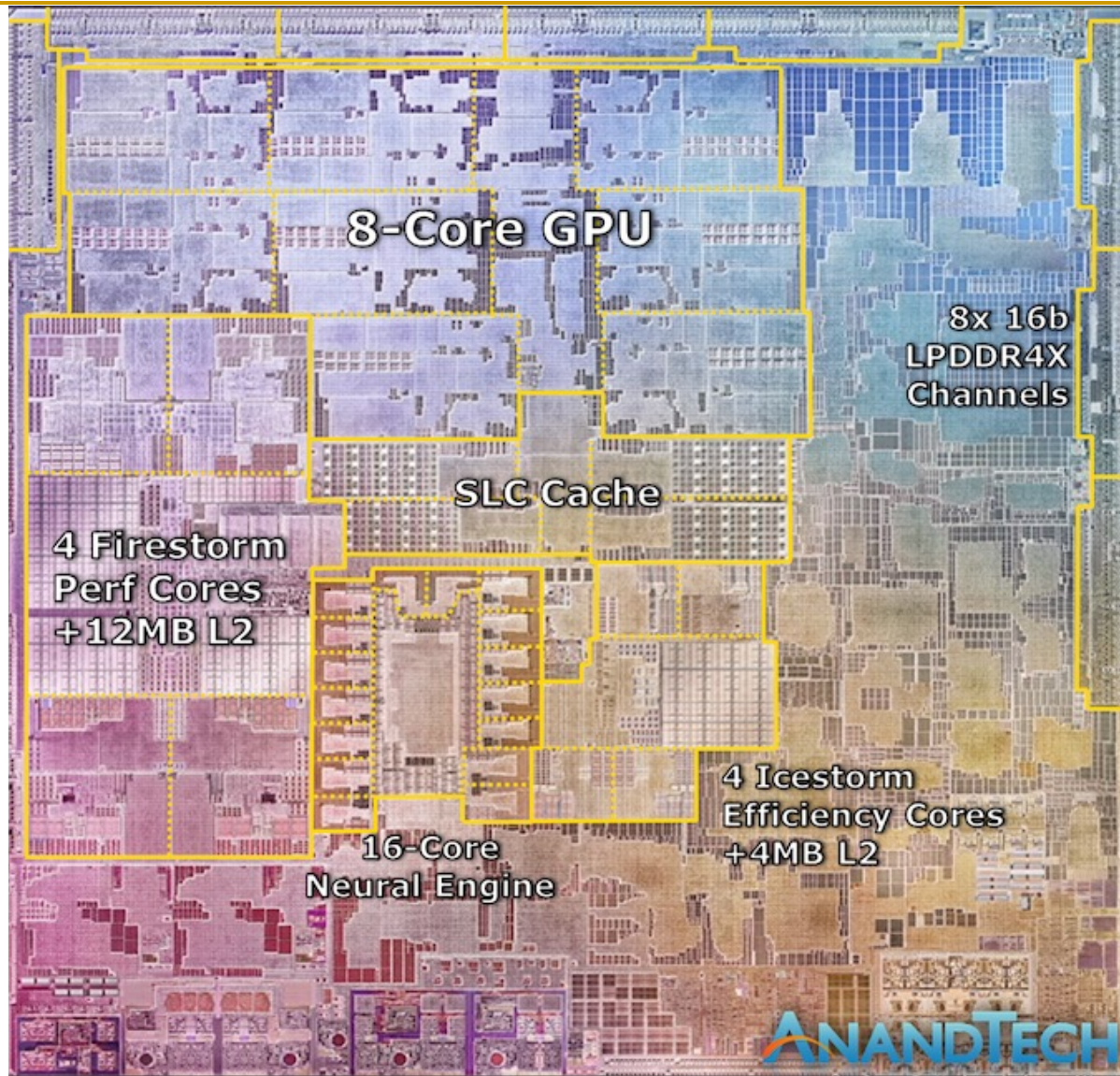
- **Separate or Unified?**
- **Pros and Cons of Unified:**
  - + Dynamic sharing of cache space: no overprovisioning that might happen with static partitioning (i.e., separate I and D caches)
  - Instructions and data can evict/thrash each other (i.e., no guaranteed space for either)
  - I and D are 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
- 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

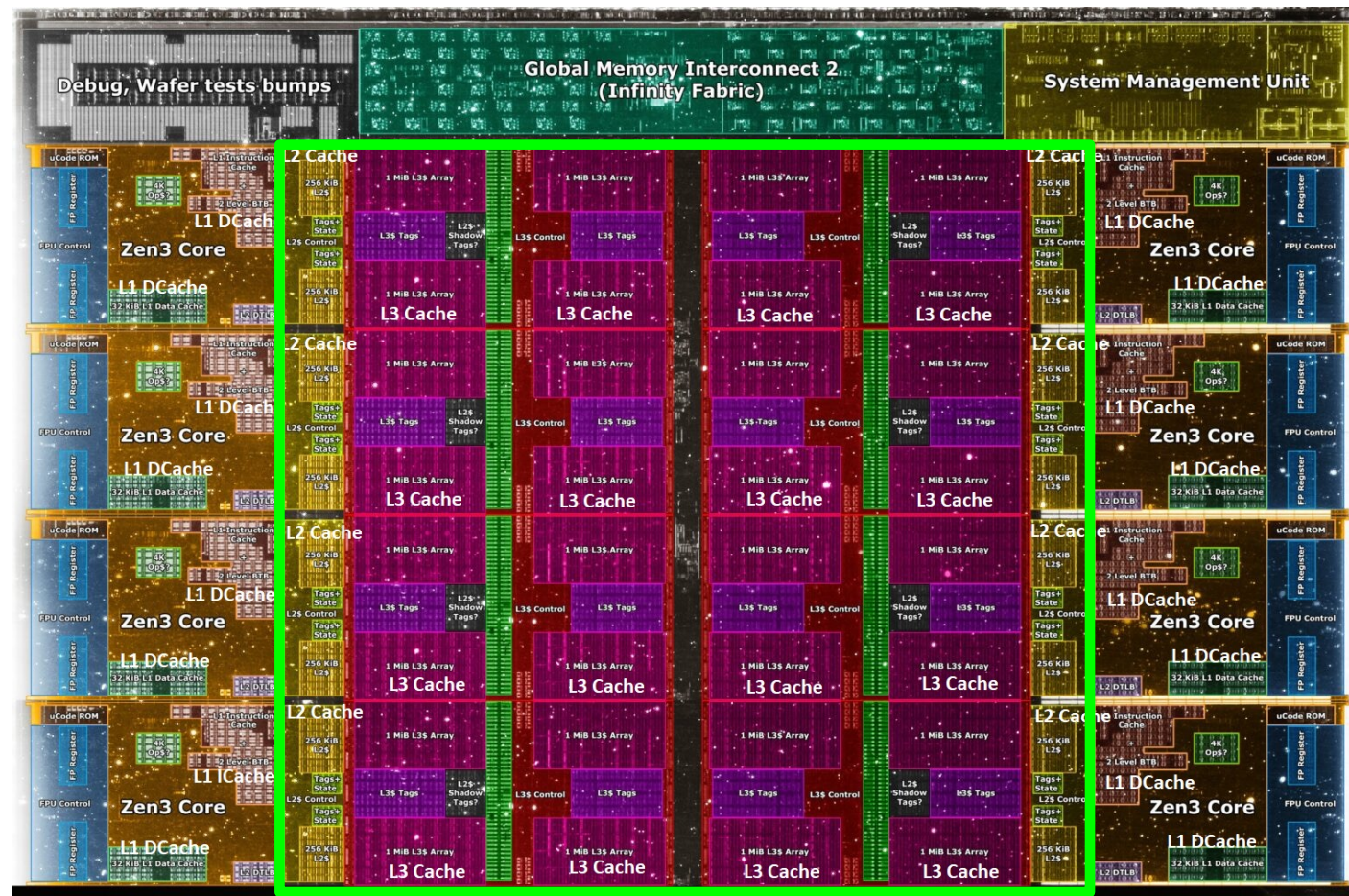
# Deeper and Larger Cache Hierarchies



Apple M1,  
2021



# Deeper and Larger Cache Hierarchies



Core Count:  
8 cores/16 threads

L1 Caches:  
32 KB per core

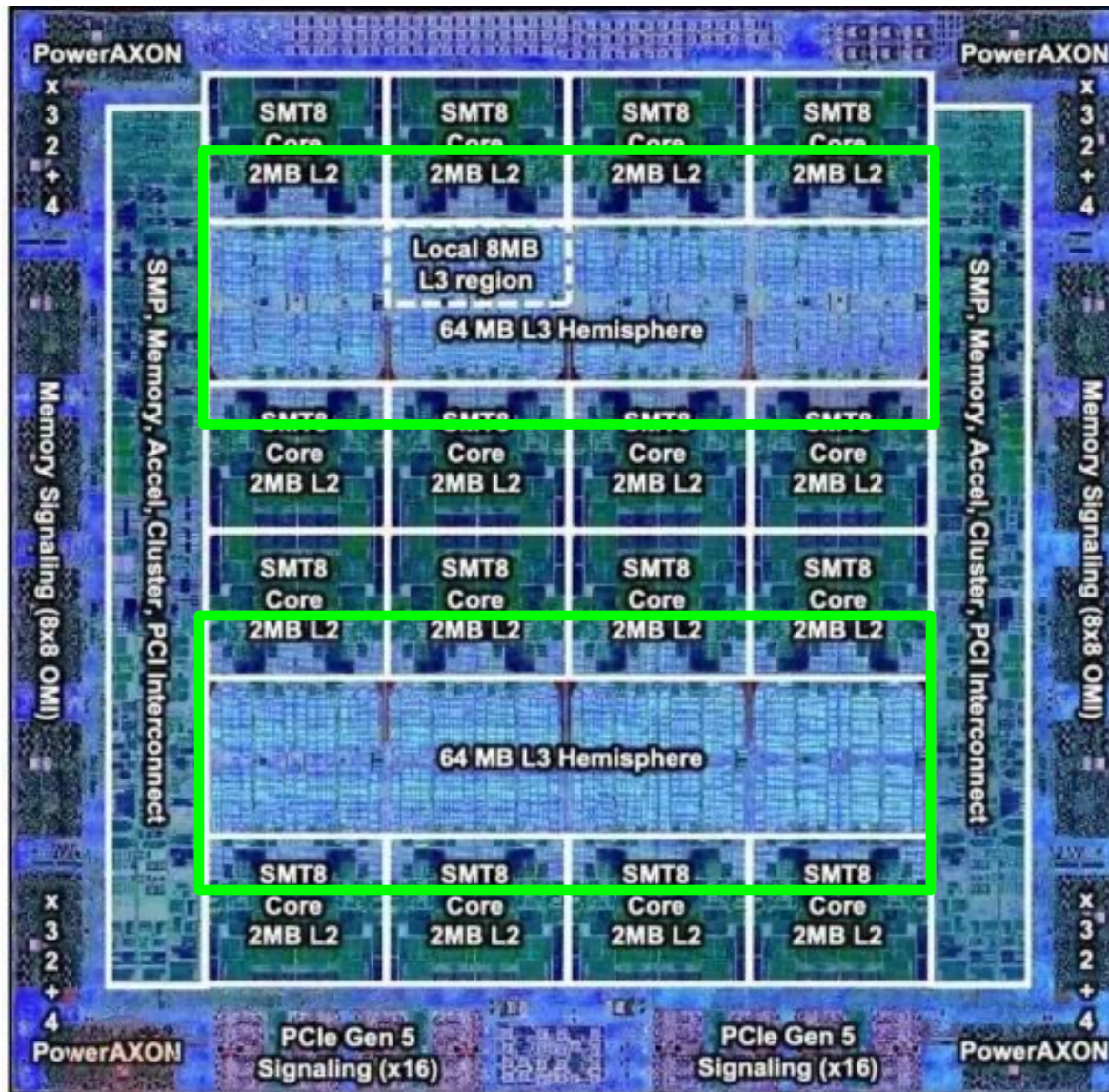
L2 Caches:  
512 KB per core

L3 Cache:  
32 MB shared

AMD Ryzen 5000, 2020



# Deeper and Larger Cache Hierarchies



IBM POWER10,  
2020

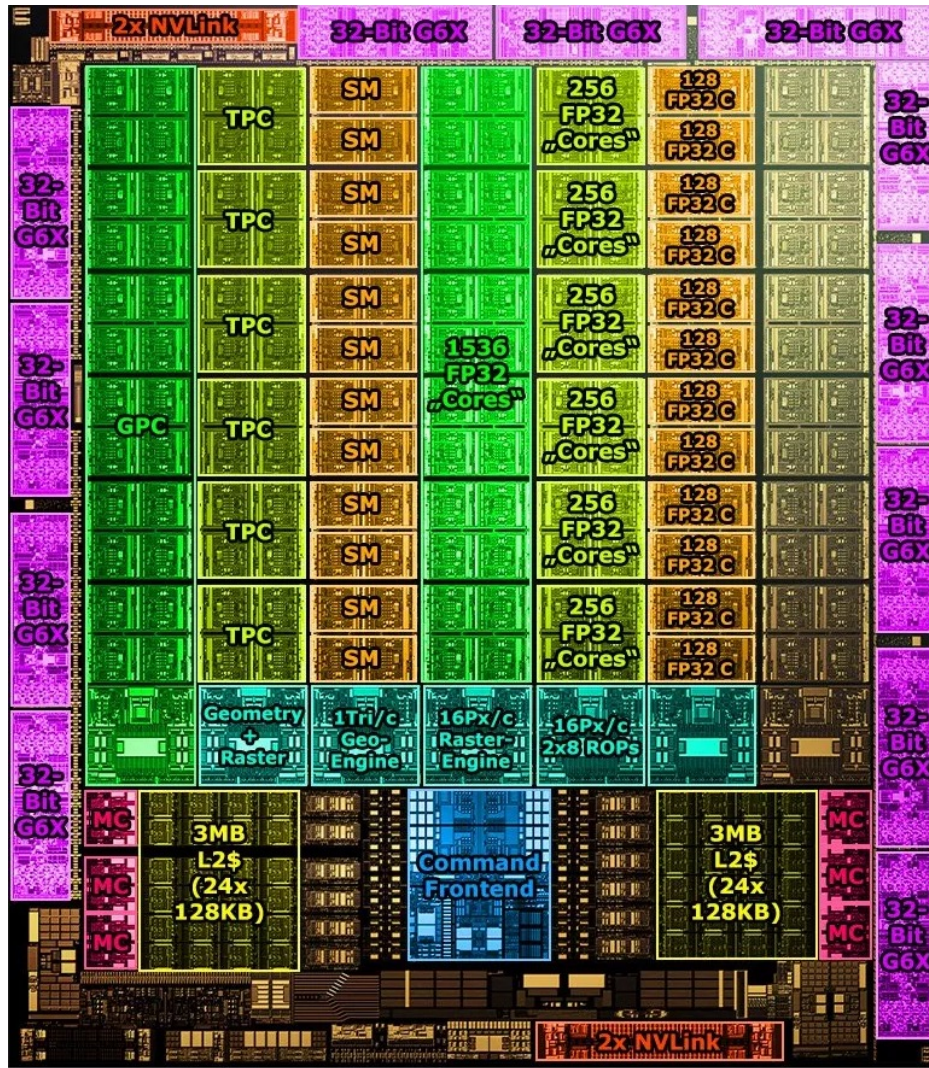
Cores:  
15-16 cores,  
8 threads/core

L2 Caches:  
2 MB per core

L3 Cache:  
120 MB shared



# 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

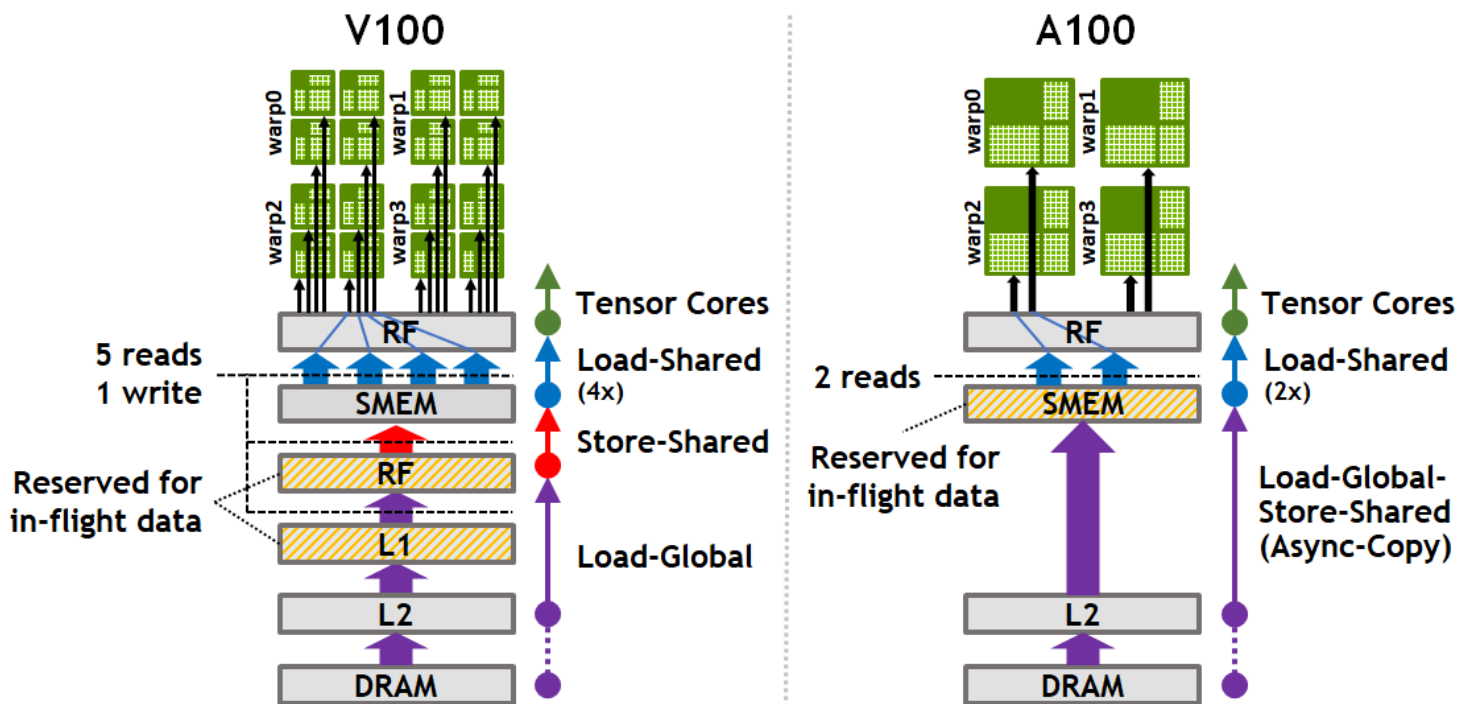
## L2 Cache:

40 MB shared



# NVIDIA V100 & A100 Memory Hierarchy

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



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

# Cache Performance

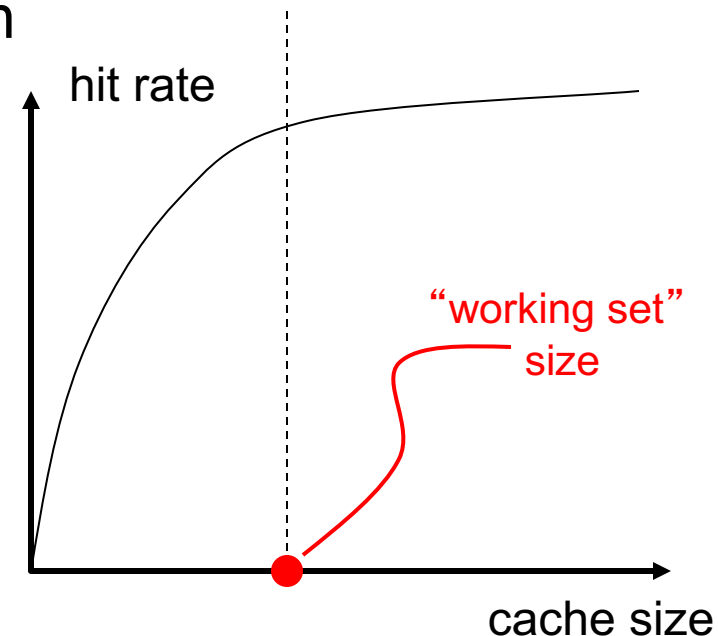
# Cache Parameters vs. Miss/Hit Rate

---

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

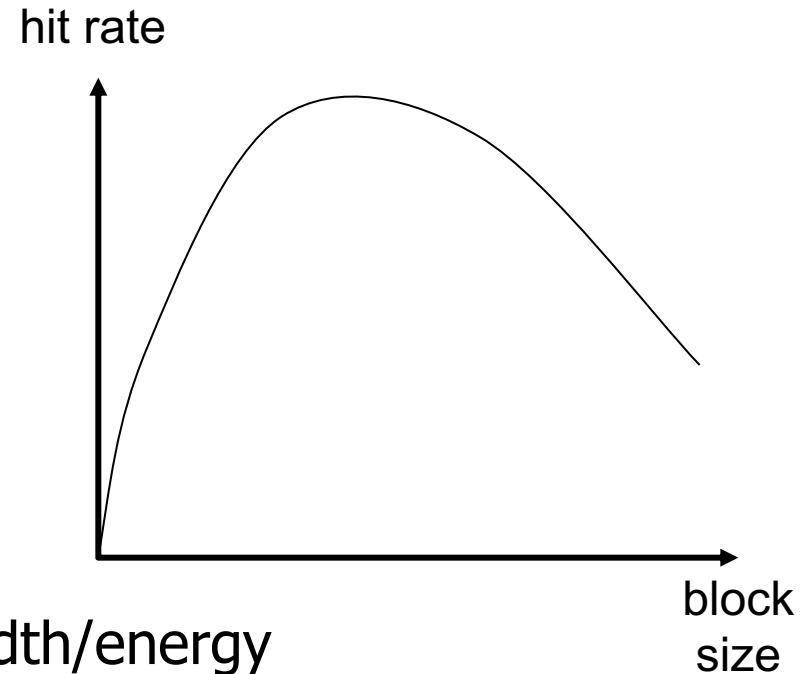
# Cache Size

- 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

---

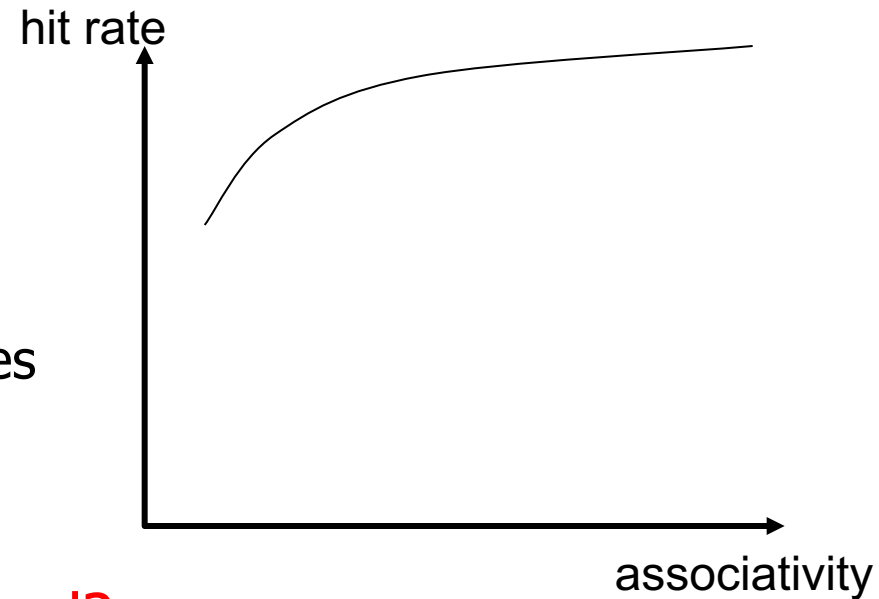
- Large cache blocks can take a long time to fill into the cache
  - Idea: Fill cache block **critical-word first**
  - Supply the critical data to the processor immediately
- Large cache blocks can waste bus bandwidth
  - Idea: Divide a block into **subblocks**
  - Associate separate valid and dirty bits for each subblock
  - **Recall: When is this useful?**



# Associativity

---

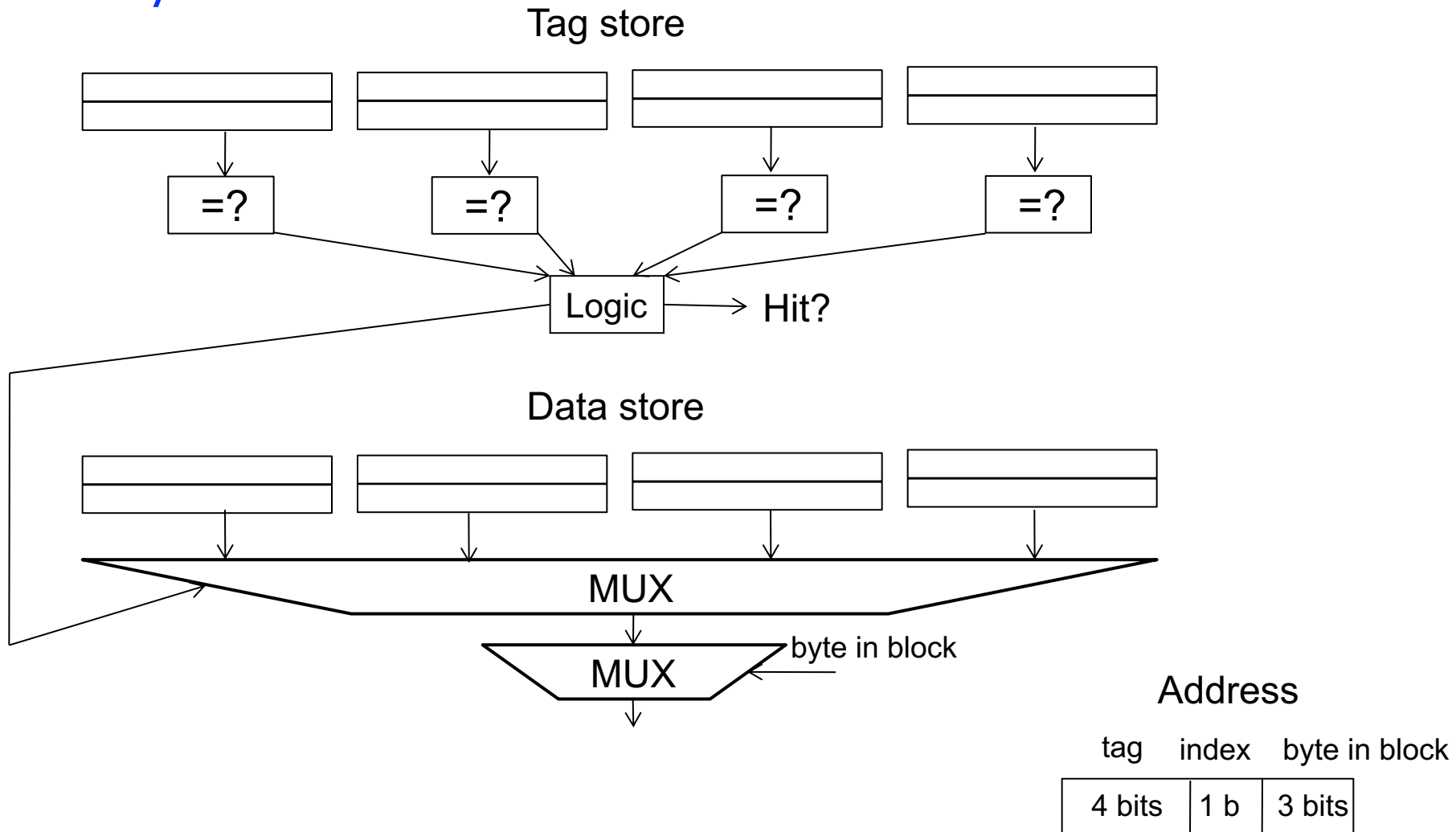
- How many blocks can be present in the same index (i.e., set)?
- Larger associativity
  - ❑ lower miss rate (reduced conflicts)
  - ❑ higher hit latency and area cost (plus diminishing returns)
- Smaller associativity
  - ❑ lower cost
  - ❑ lower hit latency
    - Especially important for L1 caches
- Is power of 2 associativity required?





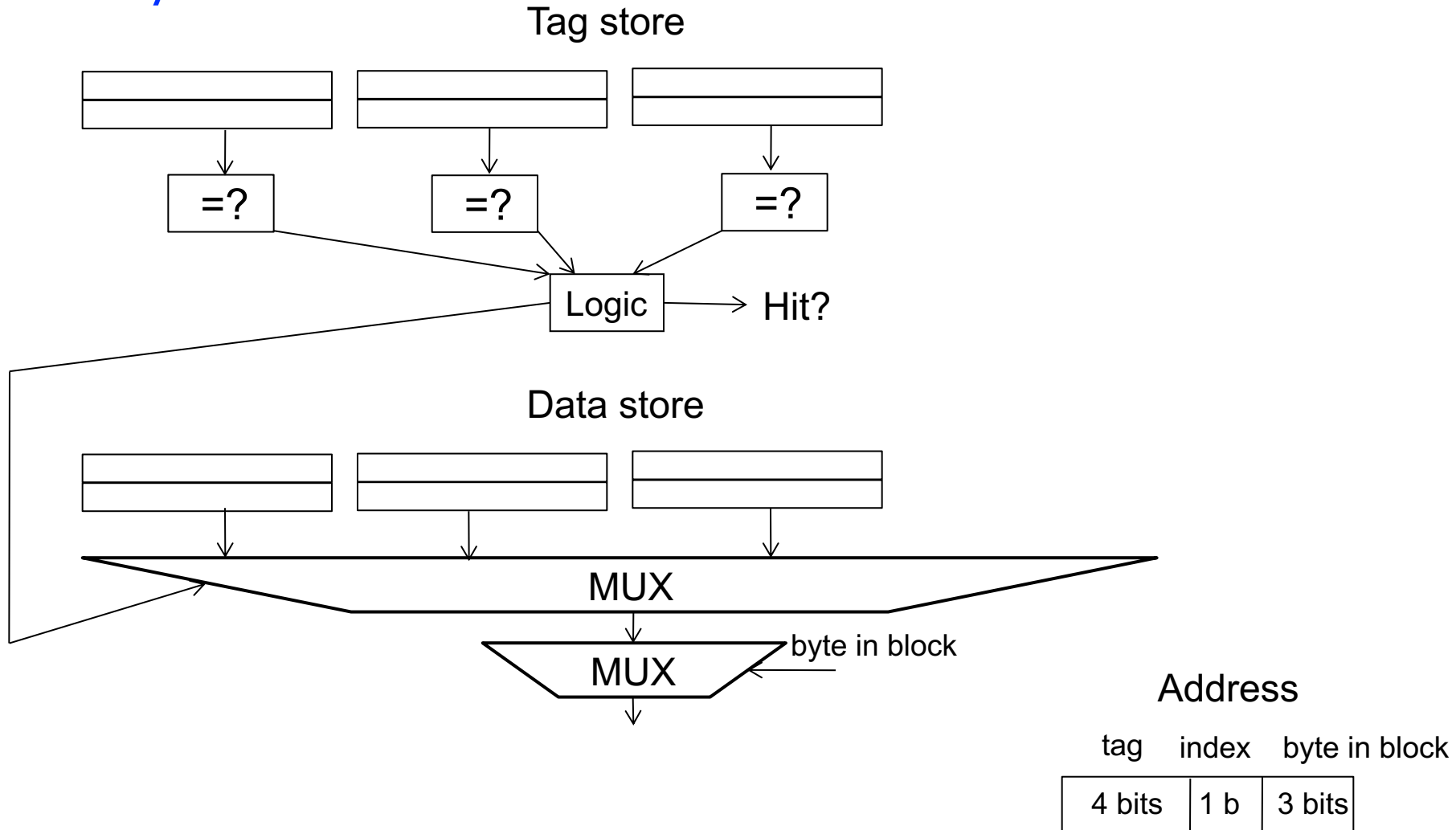
# Recall: Higher Associativity (4-way)

## ■ 4-way

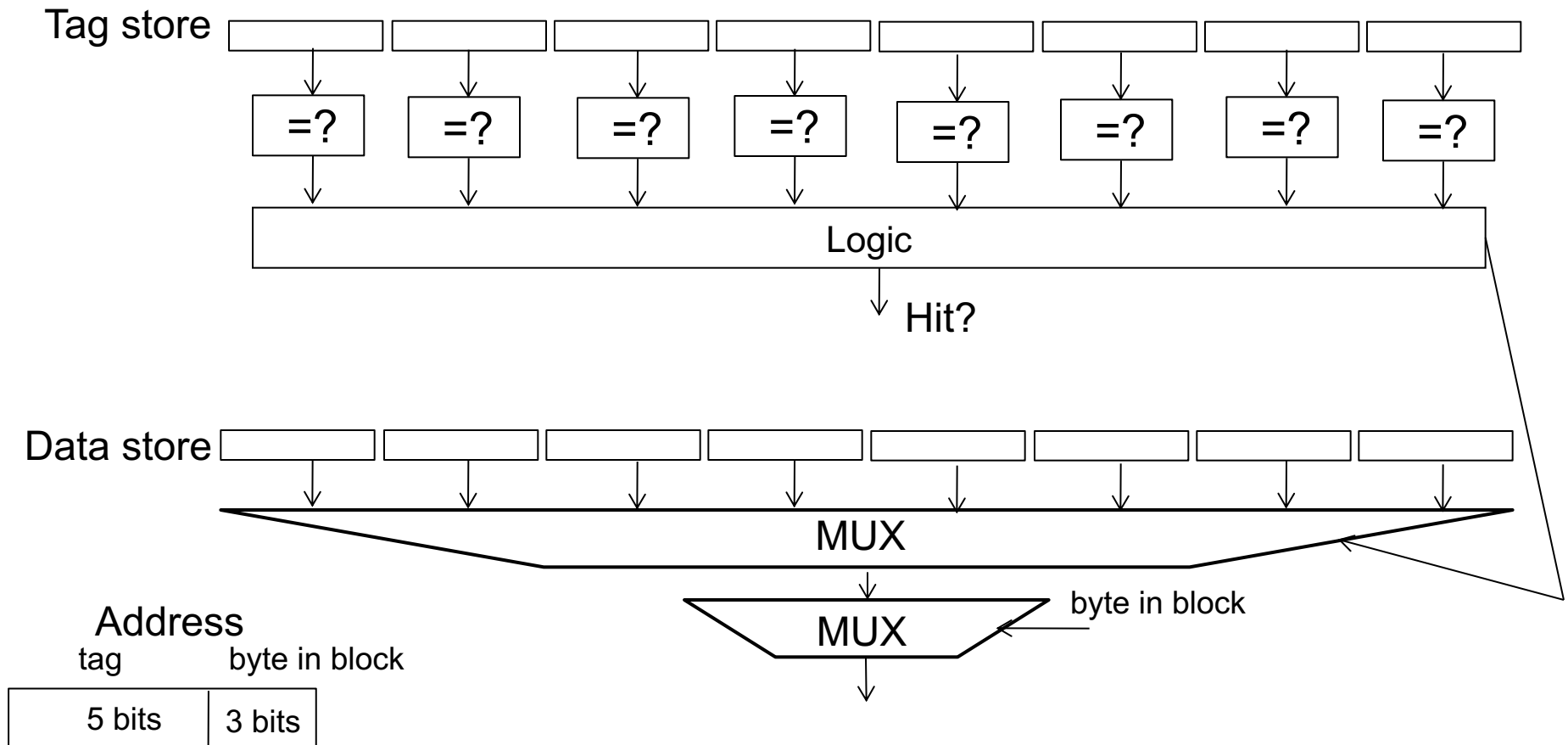


# Higher Associativity (3-way)

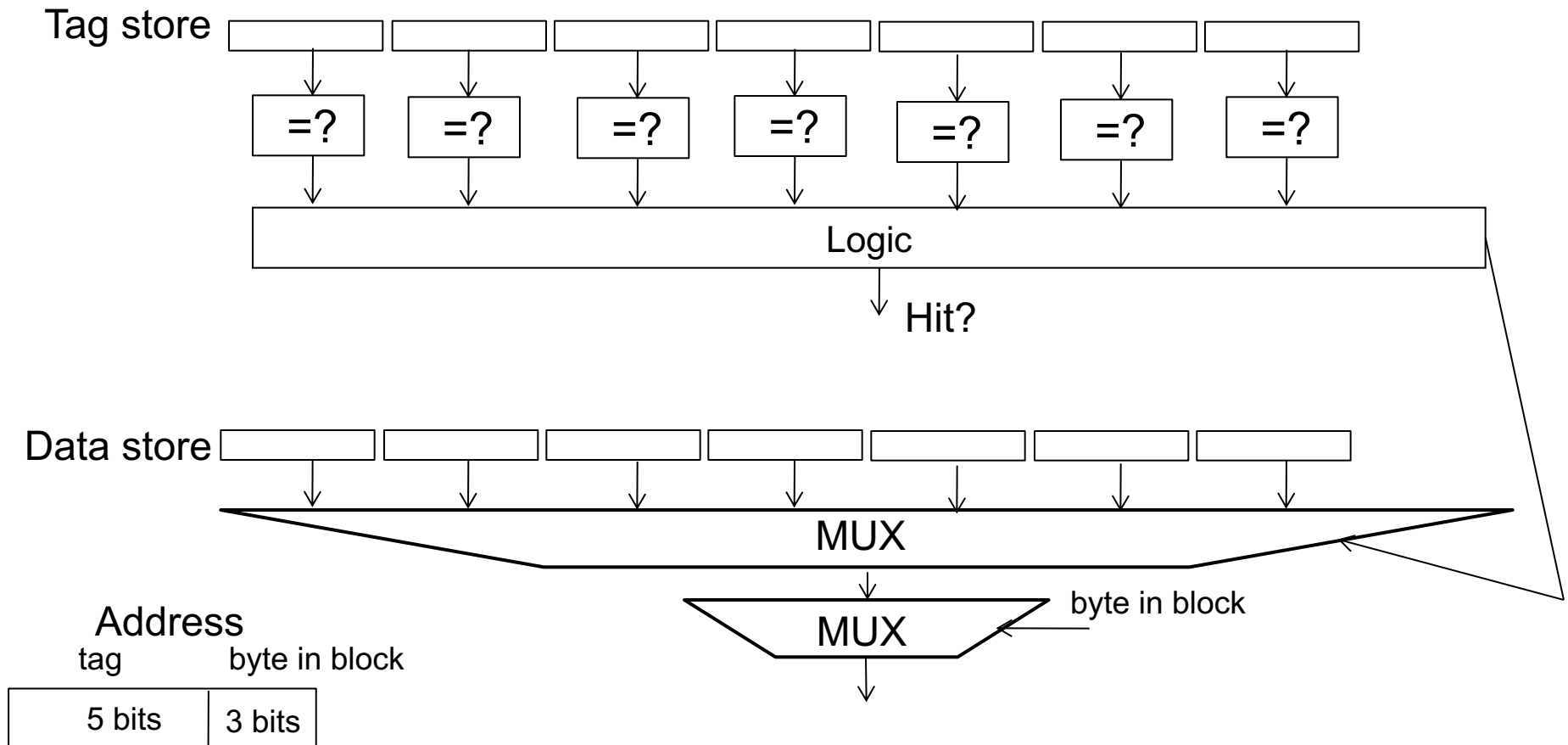
## ■ 3-way



# Recall: 8-way Fully Associative 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 should hit unless the cache block is displaced 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 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)

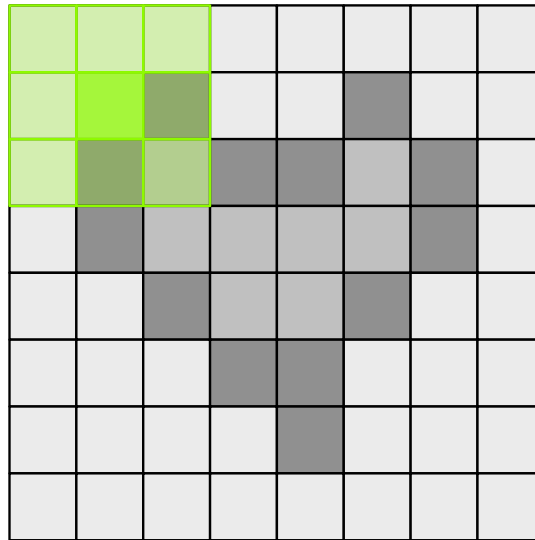
---

- **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

# Data Reuse: An Example from GPU Computing

- Same memory locations accessed by neighboring threads

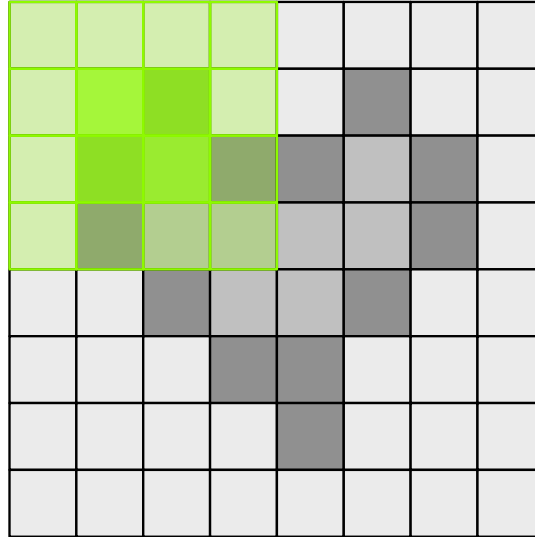
Gaussian filter applied on every pixel of an image



```
for (int i = 0; i < 3; i++){  
    for (int j = 0; j < 3; j++){  
        sum += gauss[i][j] * Image[(i+row-1)*width + (j+col-1)];  
    }  
}
```

# Data Reuse: Tiling in GPU Computing

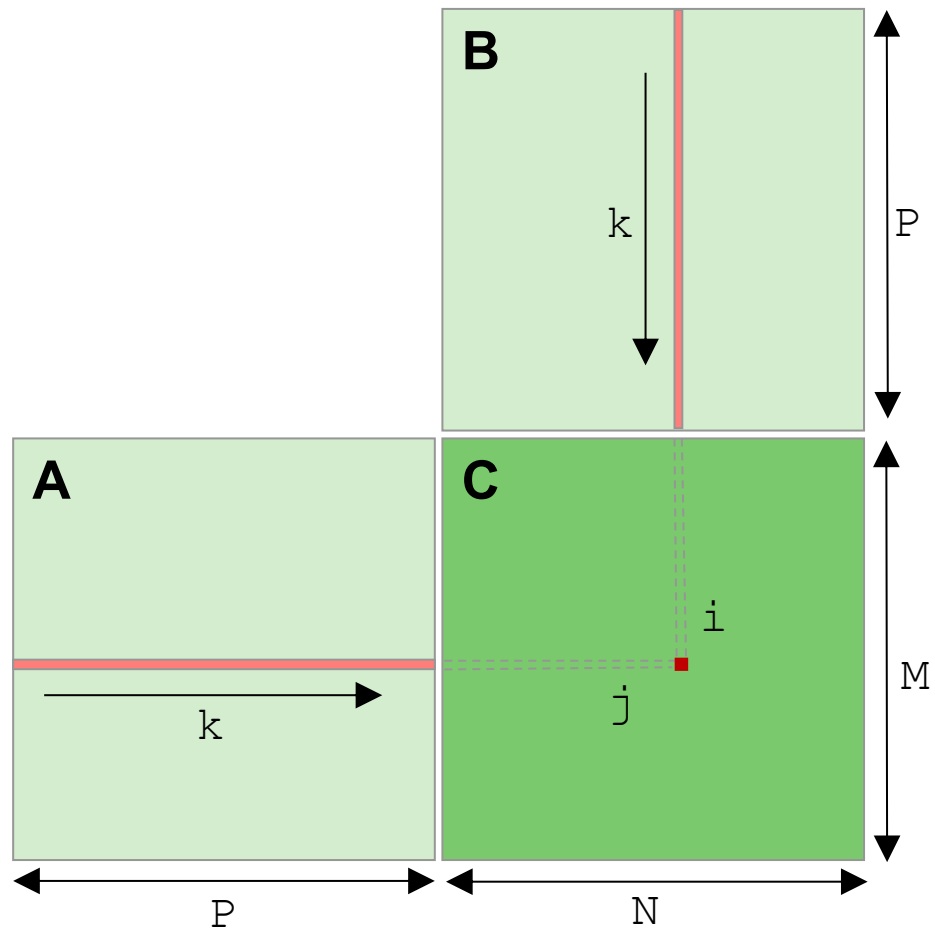
- To take advantage of data reuse, we divide the input into **tiles** that can be loaded into **shared memory (scratchpad memory)**



```
__shared__ int l_data[(L_SIZE+2)*(L_SIZE+2)];  
...  
Load tile into shared memory  
__syncthreads();  
for (int i = 0; i < 3; i++){  
    for (int j = 0; j < 3; j++){  
        sum += gauss[i][j] * l_data[(i+l_row-1)*(L_SIZE+2)+j+l_col-1];  
    }  
}
```

# Naïve Matrix Multiplication (I)

- Matrix multiplication:  $C = A \times B$
- Consider two input matrices A and B in row-major layout
  - A size is  $M \times P$
  - B size is  $P \times N$
  - C size is  $M \times N$





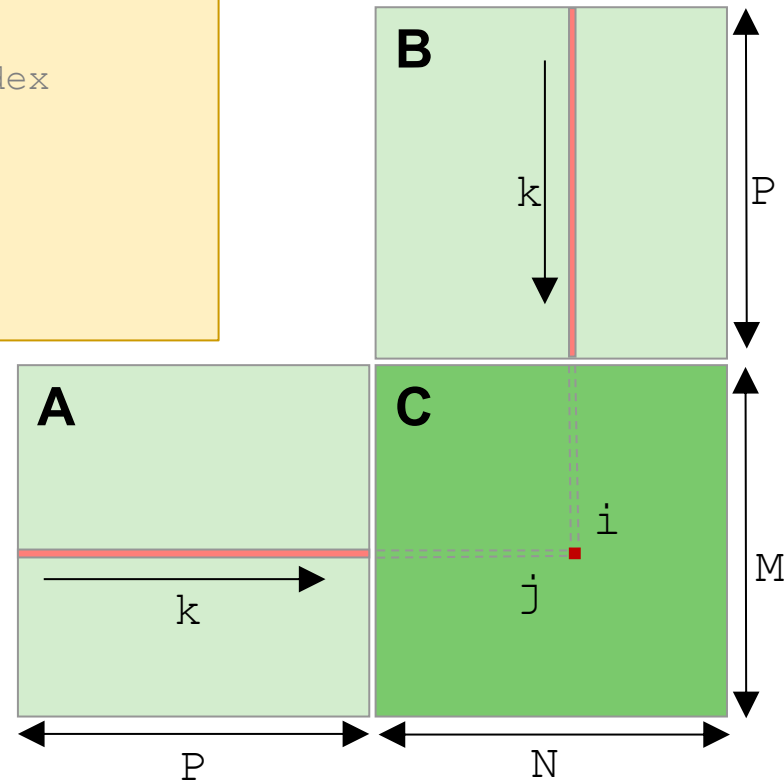
# Naïve Matrix Multiplication (II)

- Naïve implementation of matrix multiplication has **poor cache locality**

```
#define A(i,j) matrix_A[i * P + j]
#define B(i,j) matrix_B[i * N + j]
#define C(i,j) matrix_C[i * N + j]

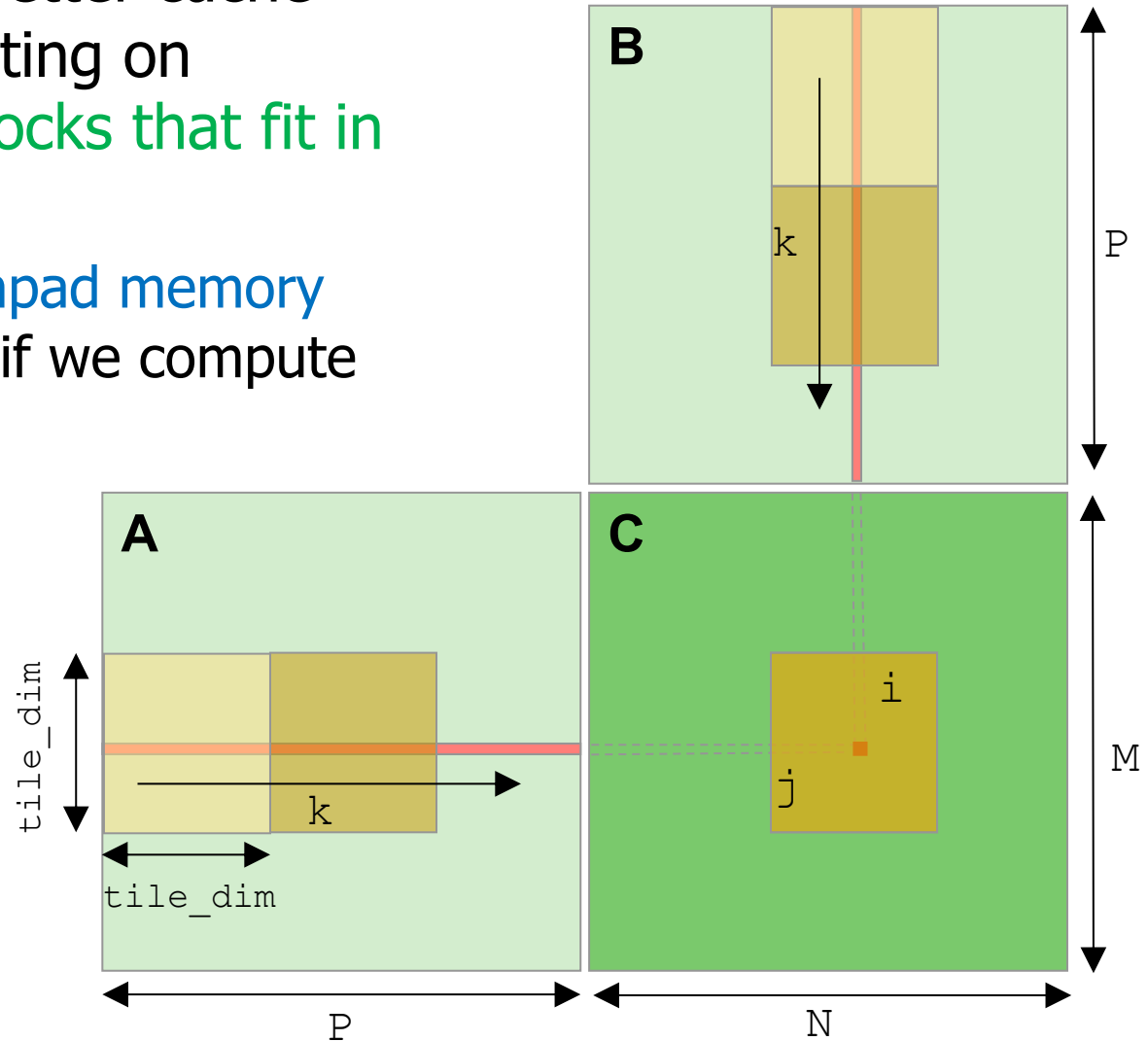
for (i = 0; i < M; i++){ // i = row index
    for (j = 0; j < N; j++){ // j = column index
        C(i, j) = 0; // Set to zero
        for (k = 0; k < P; k++) // Row x Col
            C(i, j) += A(i, k) * B(k, j);
    }
}
```

Consecutive accesses to B are far from each other, in **different cache lines**.  
Every access to B is likely to cause a cache miss



# Tiled Matrix Multiplication (I)

- We can achieve better cache locality by computing on smaller tiles or blocks that fit in the cache
  - Or in the **scratchpad memory** and **register file** if we compute on a GPU



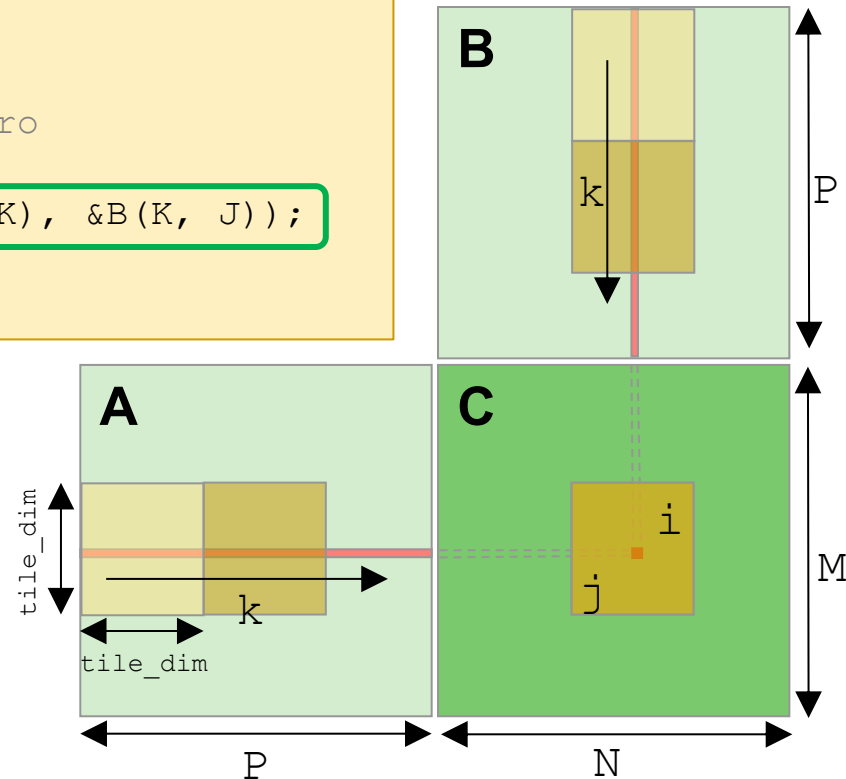
# Tiled Matrix Multiplication (II)

- **Tiled implementation** operates on submatrices (tiles or blocks) that fit fast memories (cache, scratchpad, RF)

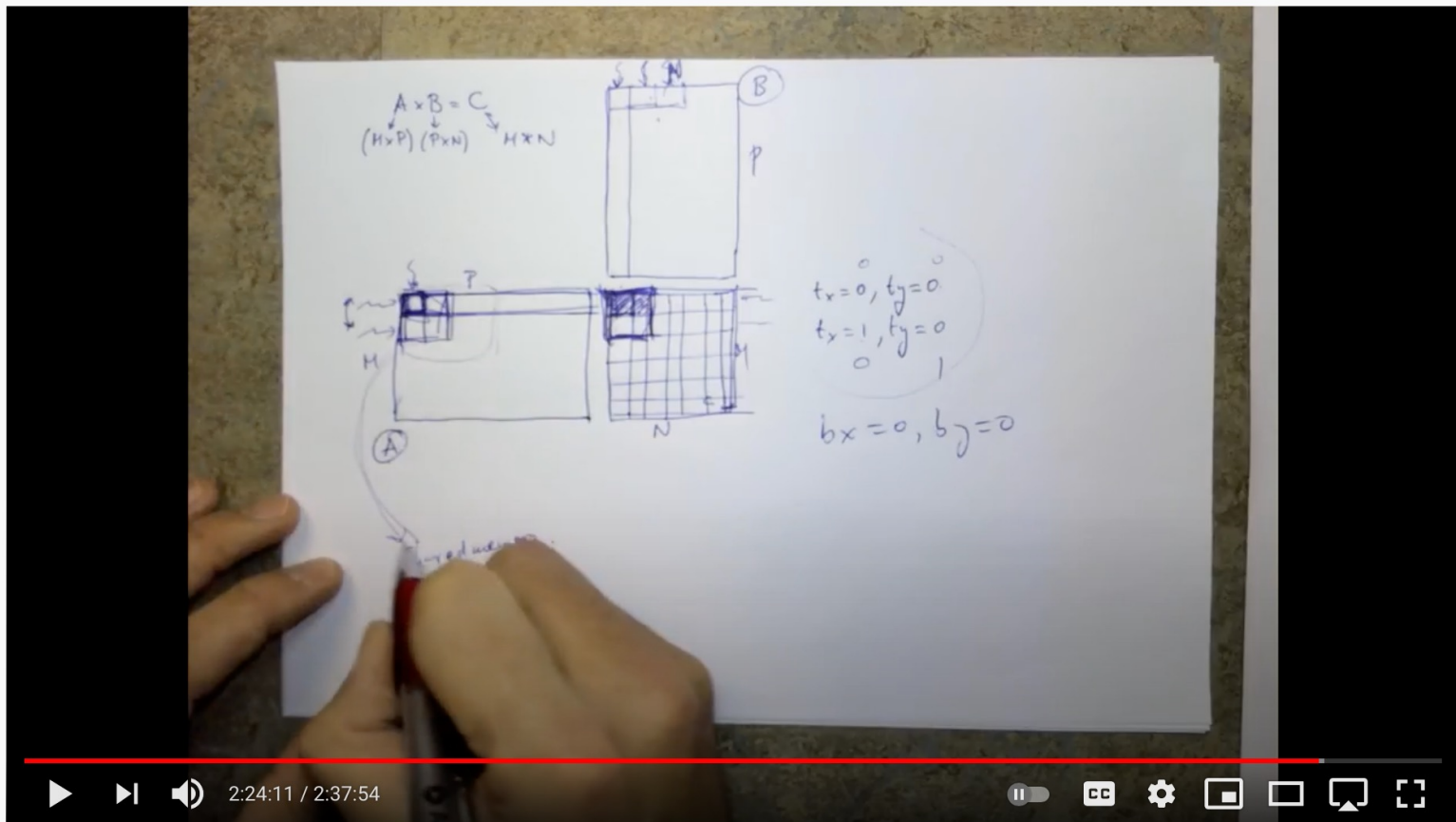
```
#define A(i,j) matrix_A[i * P + j]
#define B(i,j) matrix_B[i * N + j]
#define C(i,j) matrix_C[i * N + j]

for (I = 0; I < M; I += tile_dim){
    for (J = 0; J < N; J += tile_dim){
        Set_to_zero(&C(I, J)); // Set to zero
        for (K = 0; K < P; K += tile_dim)
            Multiply_tiles(&C(I, J), &A(I, K), &B(K, J));
    }
}
```

Multiply small submatrices (tiles or blocks)  
of size `tile_dim` x `tile_dim`



# Tiled Matrix Multiplication on GPUs



Computer Architecture - Lecture 9: GPUs and GPGPU Programming (ETH Zürich, Fall 2017)

14,426 views • Oct 23, 2017

225 2 SHARE SAVE ...



**Onur Mutlu Lectures**  
16.5K subscribers

SUBSCRIBED



# 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



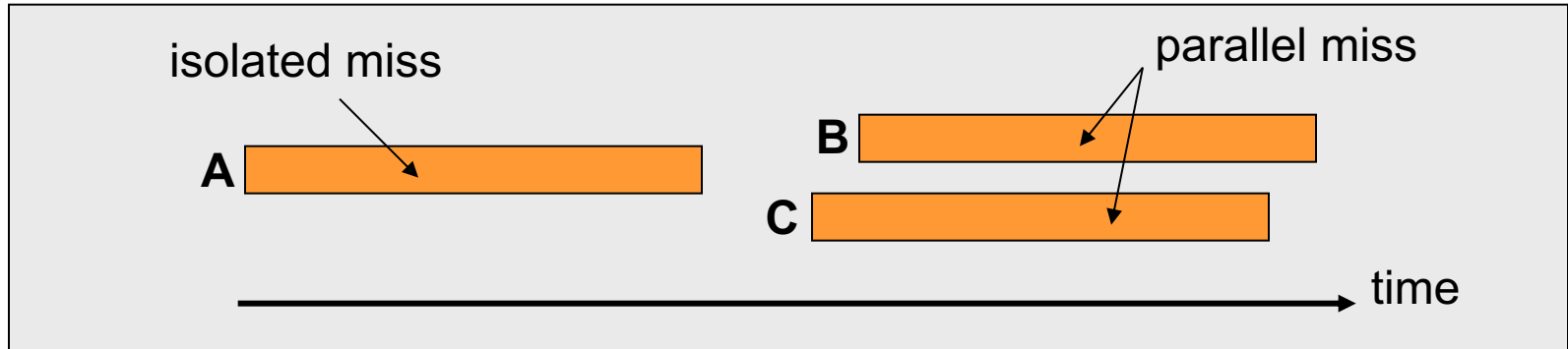
# Miss Latency/Cost

---

- 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 conflict
    - 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?
    - Is the incoming block going to evict a longer-to-refetch block?
    - ...

# 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?

---

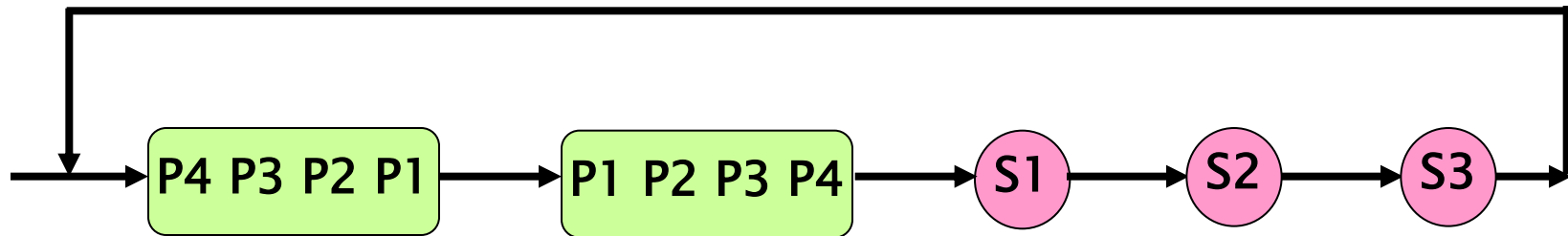
# Traditional Cache Replacement Policies

---

- ❑ 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

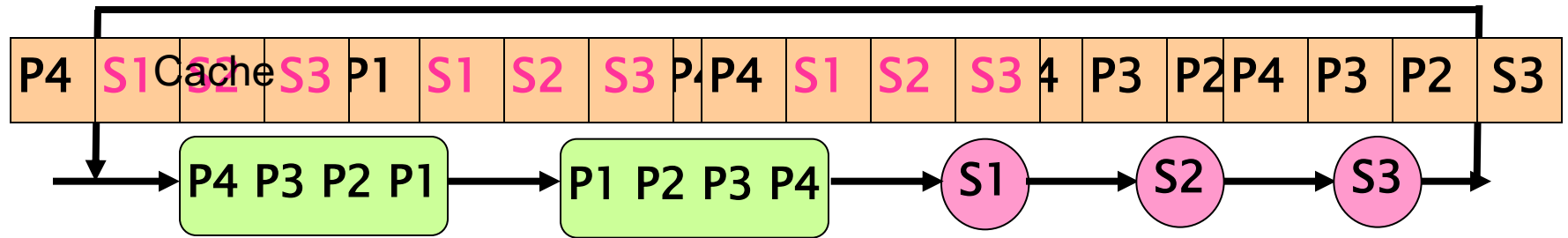
Two replacement algorithms:

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

For a fully associative cache containing 4 blocks

---

# Fewest Misses $\neq$ Best Performance



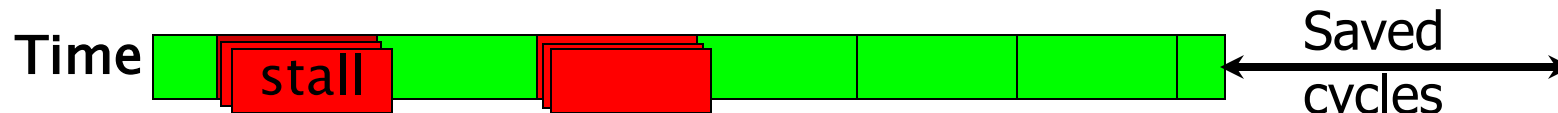
Hit/Miss H H H M H H H H M M M



**Misses=4**  
**Stalls=4**

Belady's OPT replacement

Hit/Miss H M M M H M M M H H H



**Misses=6**  
**Stalls=2**

MLP-Aware replacement

# Recommended: MLP-Aware Cache Replacement

---

- How do we incorporate MLP/cost into replacement decisions?
- How do we design a hybrid cache replacement policy?
- 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*

# Improving Basic Cache Performance

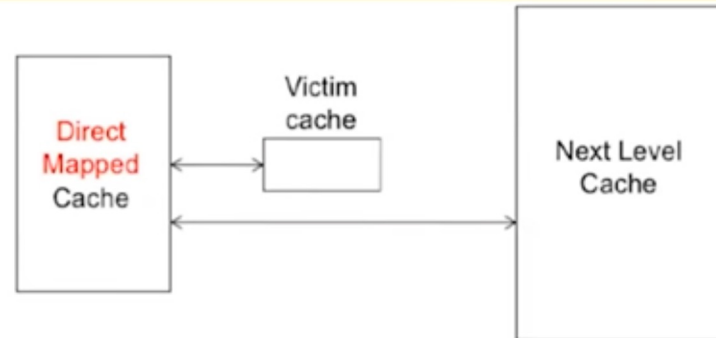
---

- 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
  - ❑ ...



# Lectures on Cache Optimizations (I)

## Victim Cache: Reducing Conflict Misses



- Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," ISCA 1990.
- Idea: Use a small fully-associative buffer (victim cache) to store recently evicted blocks
  - + Can avoid ping ponging of cache blocks mapped to the same set (if two cache blocks continuously accessed in nearby time conflict with each other)
  - Increases miss latency if accessed serially with L2; adds complexity

Computer Architecture - Lecture 3: Cache Management and Memory Parallelism (ETH Zürich, Fall 2017)

6,392 views • Sep 29, 2017

49 1 SHARE SAVE ...

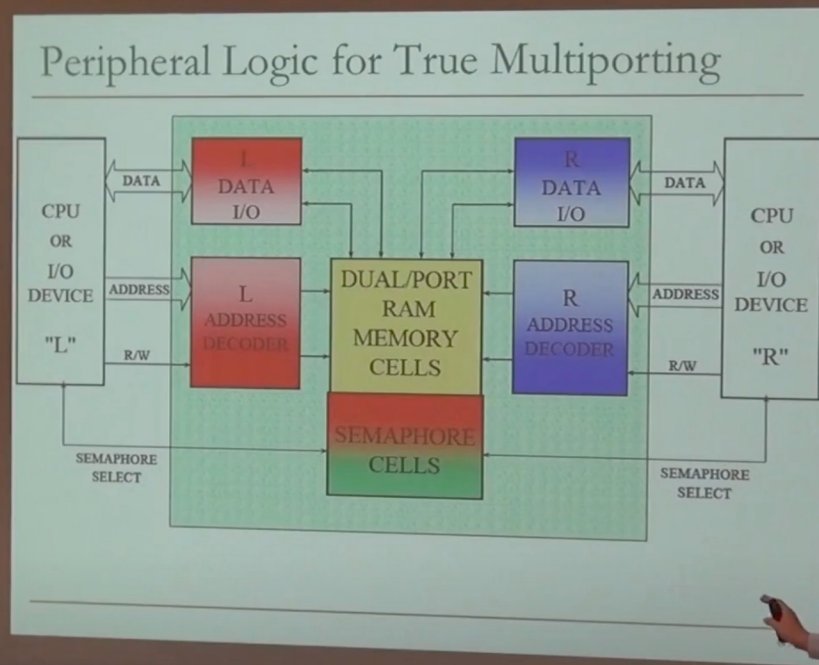


Onur Mutlu Lectures  
16.3K subscribers

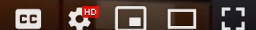
ANALYTICS

EDIT VIDEO

# Lectures on Cache Optimizations (II)



1:18:05 / 1:28:10



ETH ZÜRICH HAUPTGEBÄUDE

Computer Architecture - Lecture 4a: Cache Design (ETH Zürich, Fall 2018)

1,437 views • Sep 29, 2018

15 0 SHARE SAVE ...



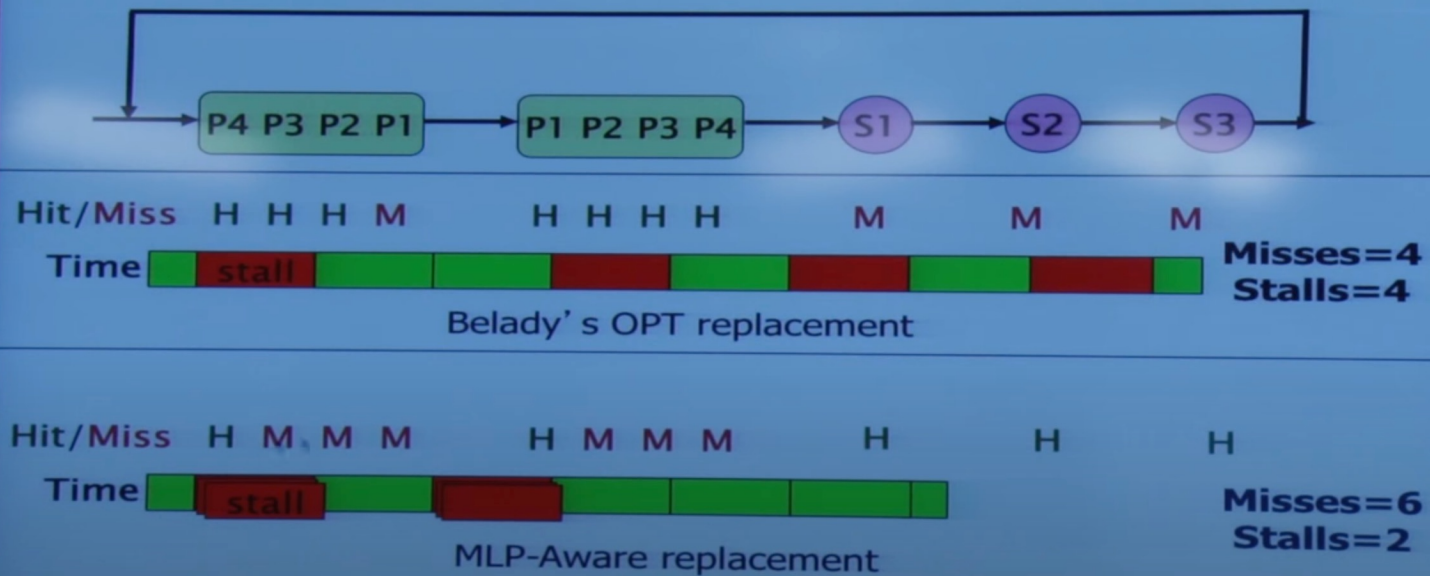
Onur Mutlu Lectures  
16.3K subscribers

ANALYTICS

EDIT VIDEO

# Lectures on Cache Optimizations (III)

Fewest Misses  $\neq$  Best Performance



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

# Lectures on Cache Optimizations

---

## ■ Computer Architecture, Fall 2017, Lecture 3

- ❑ Cache Management & Memory Parallelism (ETH, Fall 2017)
- ❑ [https://www.youtube.com/watch?v=OyomXCHNJDA&list=PL5Q2soXY2Zi9OhoVQBX\\_YFIZywZXCPI4M\\_&index=3](https://www.youtube.com/watch?v=OyomXCHNJDA&list=PL5Q2soXY2Zi9OhoVQBX_YFIZywZXCPI4M_&index=3)

## ■ Computer Architecture, Fall 2018, Lecture 4a

- ❑ Cache Design (ETH, Fall 2018)
- ❑ [https://www.youtube.com/watch?v=55oYBm9cifI&list=PL5Q2soXY2Zi9JXe3ywQMh\\_ylk\\_d5dI-TM7&index=6](https://www.youtube.com/watch?v=55oYBm9cifI&list=PL5Q2soXY2Zi9JXe3ywQMh_ylk_d5dI-TM7&index=6)

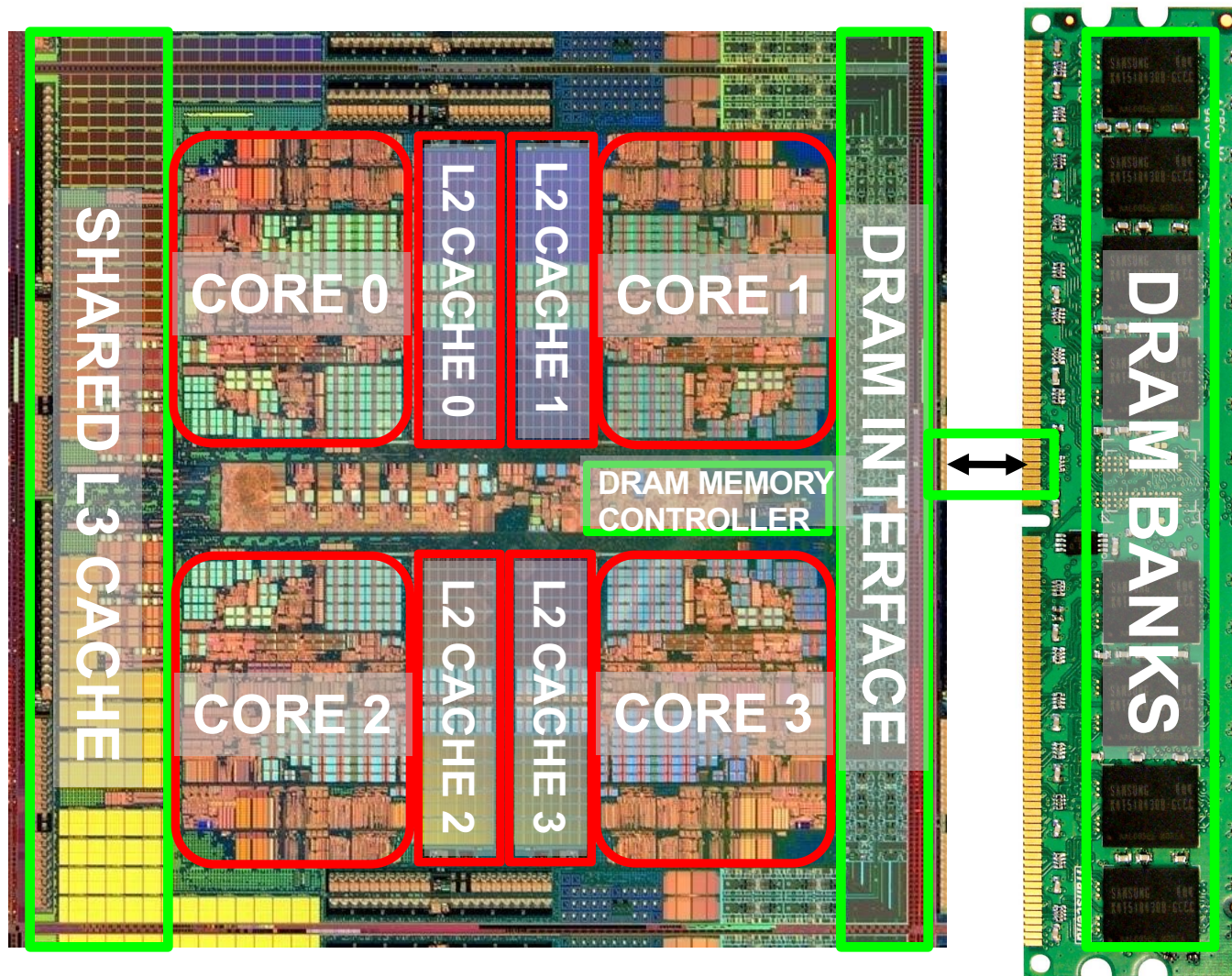
## ■ Computer Architecture, Spring 2015, Lecture 19

- ❑ High Performance Caches (CMU, Spring 2015)
- ❑ <https://www.youtube.com/watch?v=jDHx2K9HxIM&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=21>

# Multi-Core Issues in Caching

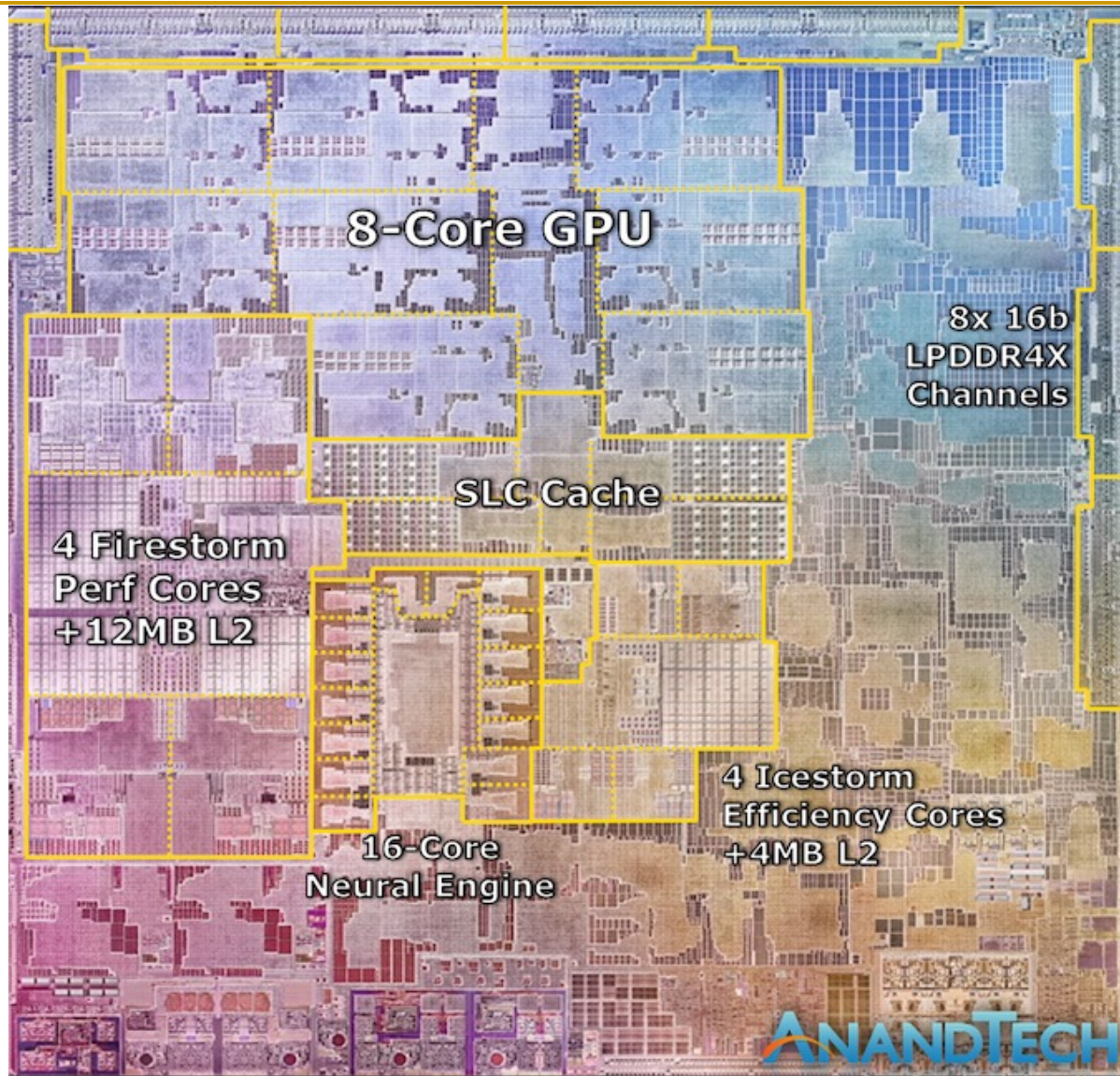


# Caches in a Multi-Core System





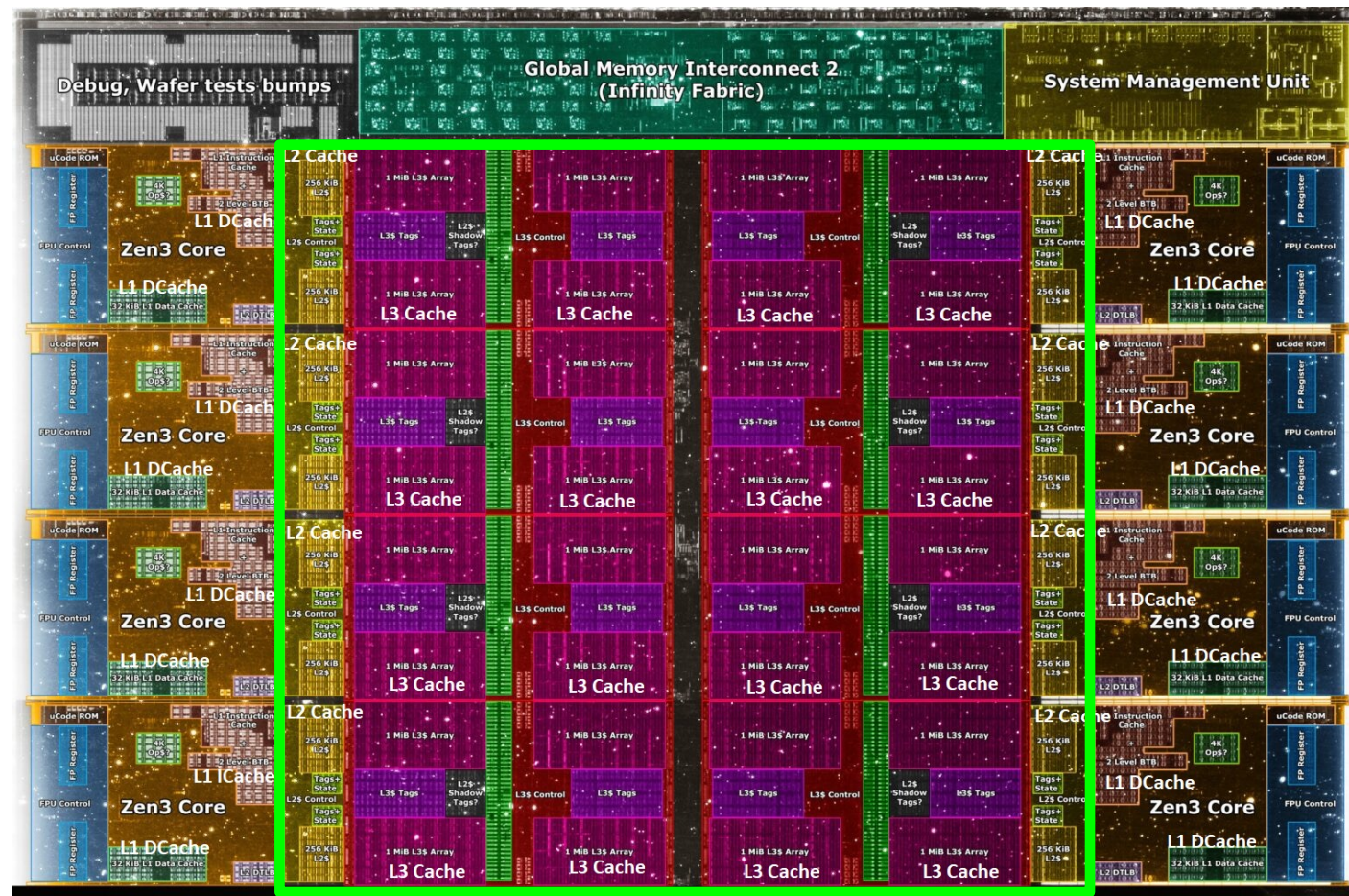
# Caches in a Multi-Core System



Apple M1,  
2021



# Caches in a Multi-Core System



Core Count:  
8 cores/16 threads

L1 Caches:  
32 KB per core

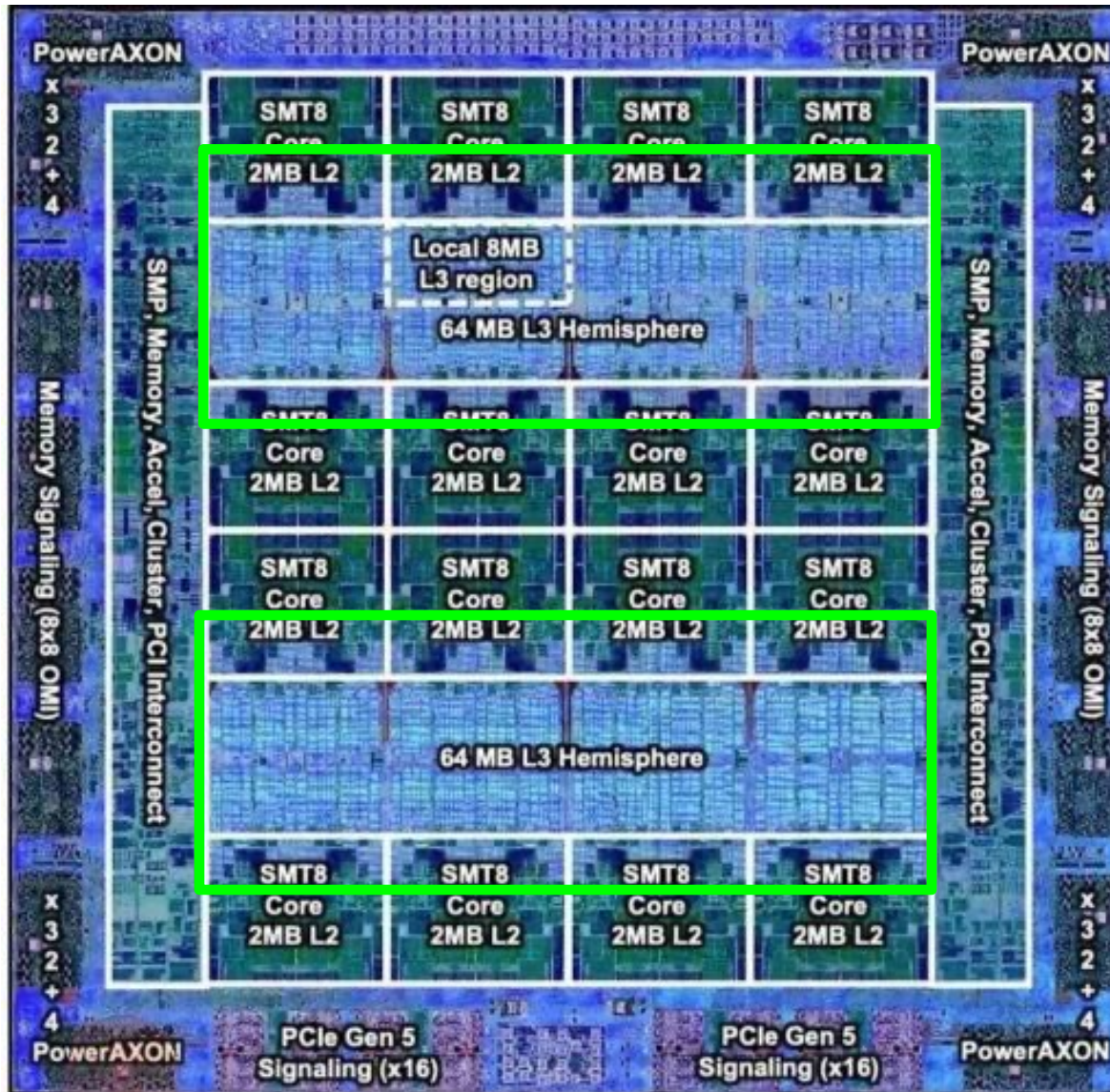
L2 Caches:  
512 KB per core

L3 Cache:  
32 MB shared

AMD Ryzen 5000, 2020



# Caches in a Multi-Core System



IBM POWER10,  
2020

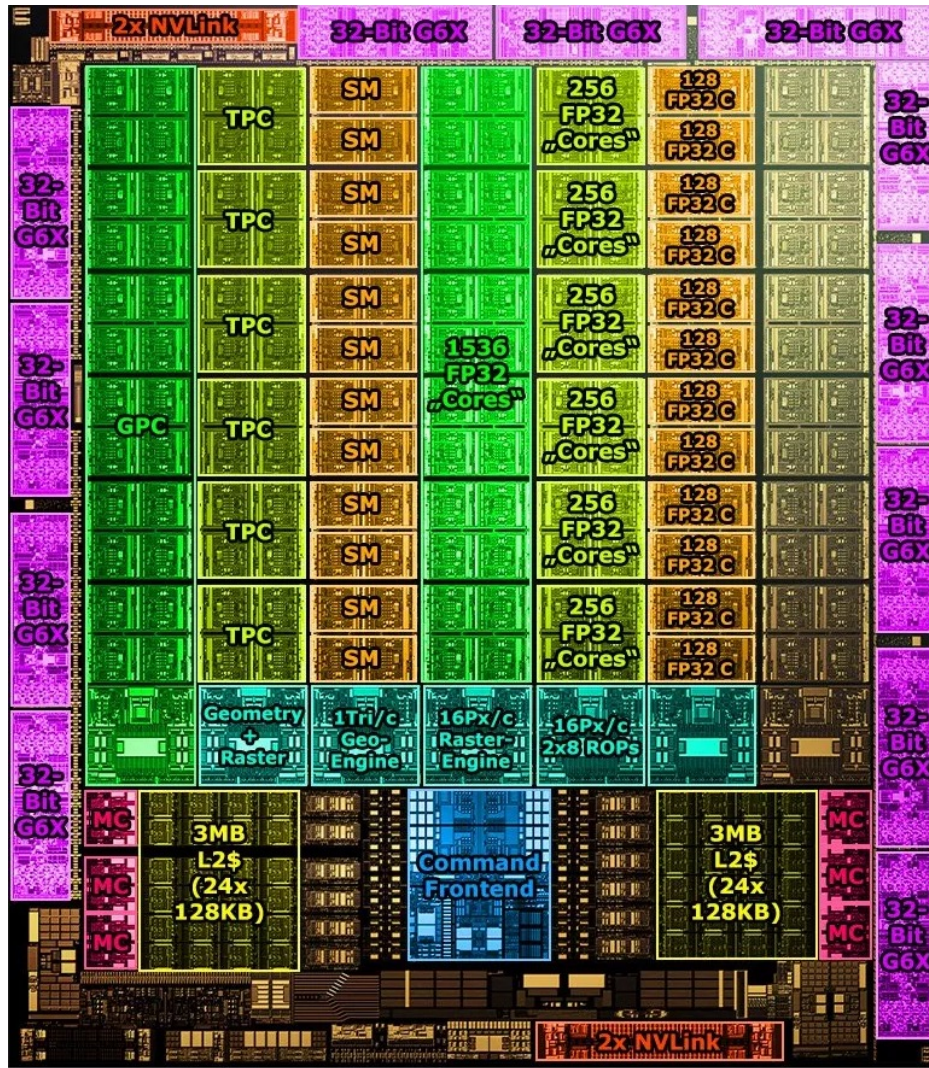
Cores:  
15-16 cores,  
8 threads/core

L2 Caches:  
2 MB per core

L3 Cache:  
120 MB shared



# Caches in a Multi-Core System



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

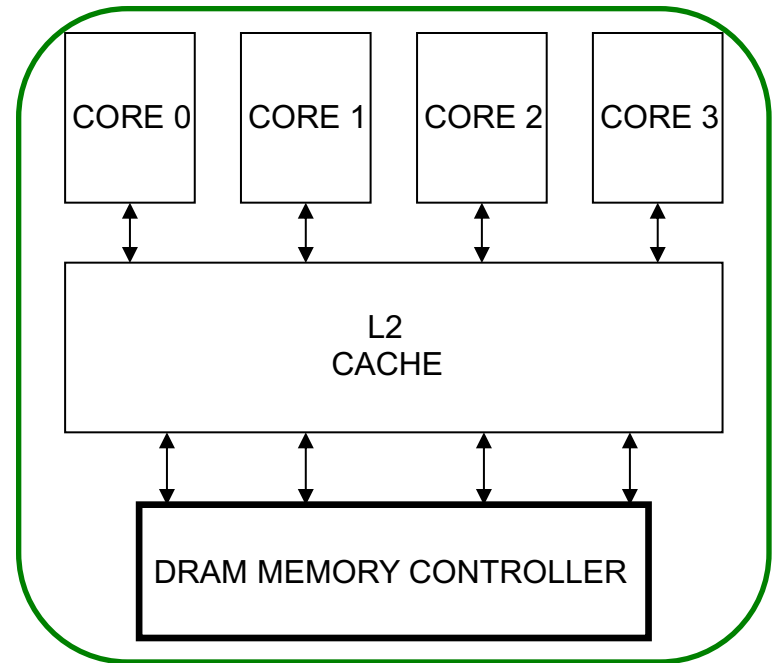
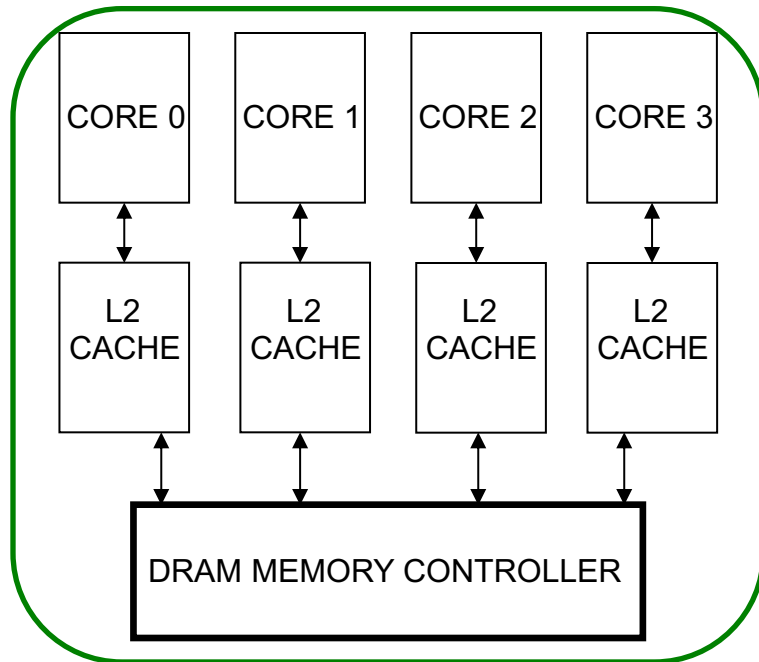
# Caches in Multi-Core Systems

---

- 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?
  - ❑ Should we store data in compressed format in some caches?
  - ❑ How do we do better reuse prediction & management in caches?

# 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

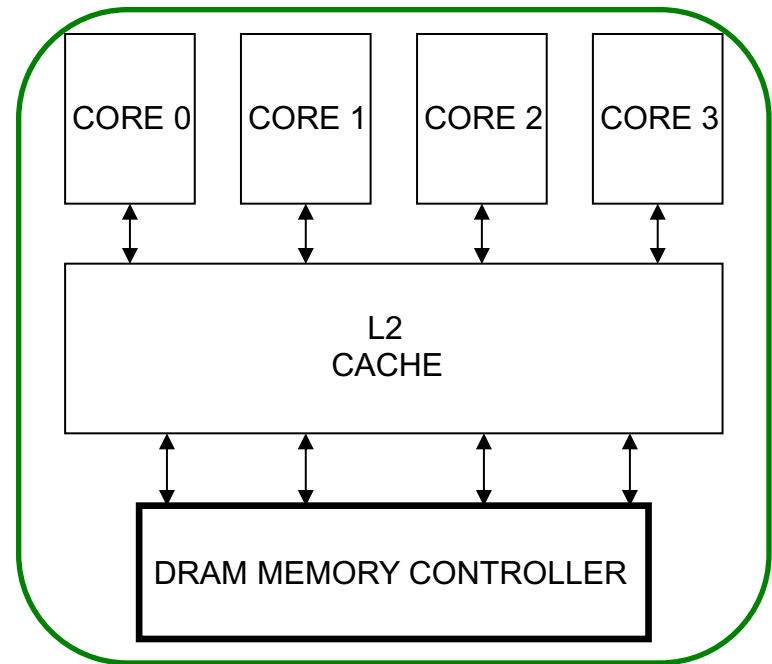
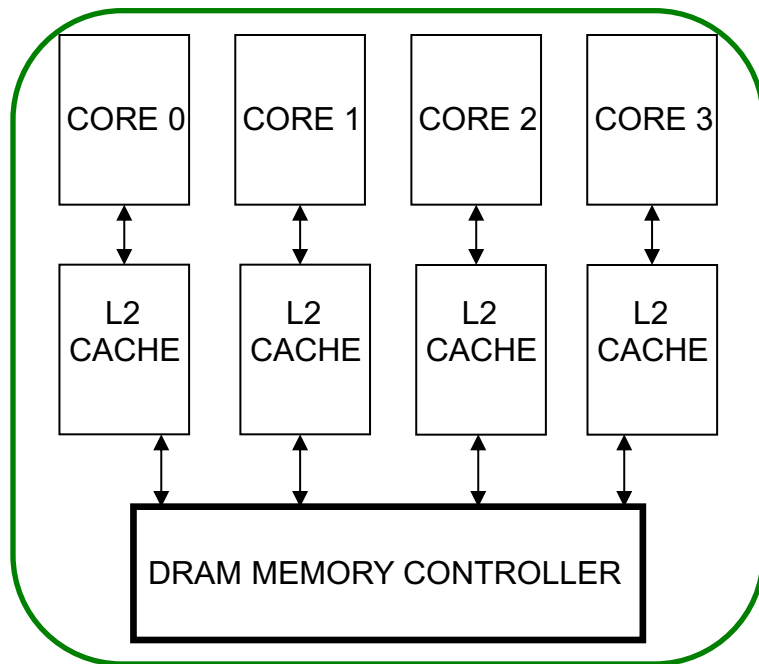
---

- 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?)

# Lectures on Multi-Core Cache Management

## Computer Architecture

### Lecture 15:

### Multi-Core Cache Management

Prof. Onur Mutlu  
ETH Zürich  
Fall 2017  
15 November 2017



0:35 / 2:33:03



Computer Architecture - Lecture 15: Multi-Core Cache Management (ETH Zürich, Fall 2017)

934 views • Nov 17, 2017

13 0 SHARE SAVE ...

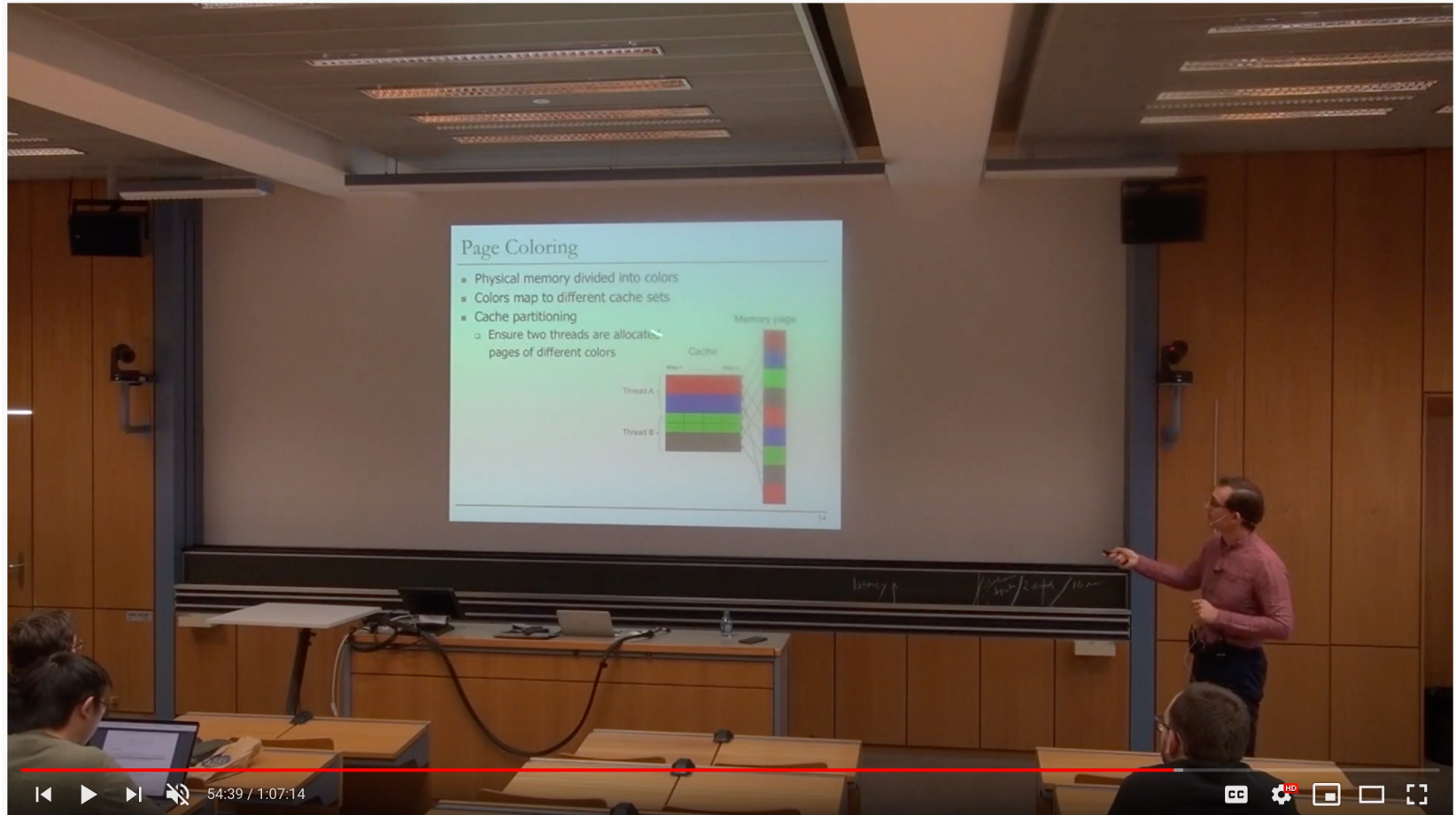


Onur Mutlu Lectures  
16.5K subscribers

ANALYTICS

EDIT VIDEO

# Lectures on Multi-Core Cache Management



ETH ZÜRICH HAUPTGEBÄUDE

Computer Architecture - Lecture 18b: Multi-Core Cache Management (ETH Zürich, Fall 2018)

744 views • Nov 23, 2018

12 0 SHARE SAVE ...



**Onur Mutlu Lectures**  
16.5K subscribers

ANALYTICS

EDIT VIDEO

# Lectures on Multi-Core Cache Management

The slide content is as follows:

### Approaches to Reuse Prediction

Use program counter or memory region information.

1. Group Blocks  
PC 1: [A B] PC 2: [S T]
2. Learn group behavior  
PC 1: [A B] PC 2: [S T]
3. Predict reuse  
PC 1: C → C  
PC 2: U → U

1. Same group ↗ same reuse behavior  
2. No control over number of high-reuse blocks

77

ETH ZÜRICH HAUPTGEBÄUDE  
Computer Architecture - Lecture 19a: Multi-Core Cache Management II (ETH Zürich, Fall 2018)

293 views • Dec 2, 2018

6 0 SHARE SAVE ...



**Onur Mutlu Lectures**  
16.5K subscribers

ANALYTICS

EDIT VIDEO

# Lectures on Multi-Core Cache Management

---

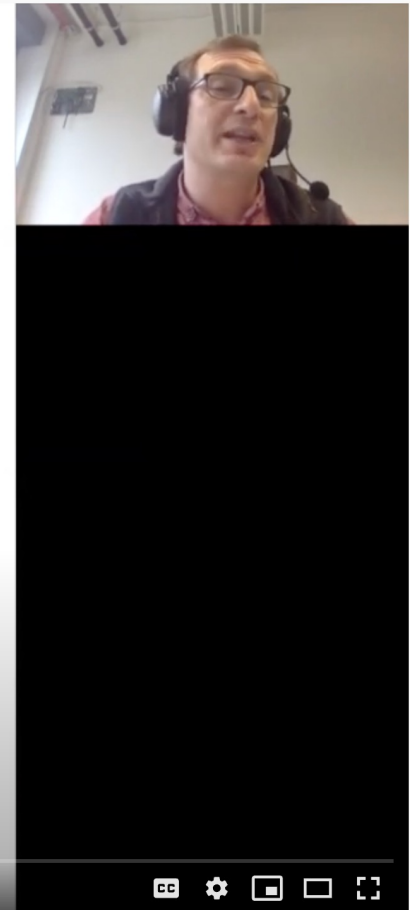
- **Computer Architecture, Fall 2018, Lecture 18b**
  - Multi-Core Cache Management (ETH, Fall 2018)
  - [https://www.youtube.com/watch?v=c9FhGRB3HoA&list=PL5Q2soXY2Zi9JXe3ywQMhylk\\_d5dI-TM7&index=29](https://www.youtube.com/watch?v=c9FhGRB3HoA&list=PL5Q2soXY2Zi9JXe3ywQMhylk_d5dI-TM7&index=29)
  
- **Computer Architecture, Fall 2018, Lecture 19a**
  - Multi-Core Cache Management II (ETH, Fall 2018)
  - [https://www.youtube.com/watch?v=Siz86\\_\\_PD4w&list=PL5Q2soXY2Zi9JXe3ywQMhylk\\_d5dI-TM7&index=30](https://www.youtube.com/watch?v=Siz86__PD4w&list=PL5Q2soXY2Zi9JXe3ywQMhylk_d5dI-TM7&index=30)
  
- **Computer Architecture, Fall 2017, Lecture 15**
  - Multi-Core Cache Management (ETH, Fall 2017)
  - [https://www.youtube.com/watch?v=7\\_Tqlw8gxOU&list=PL5Q2soXY2Zi9OhoVQBXyFIZywZXCPI4M\\_&index=17](https://www.youtube.com/watch?v=7_Tqlw8gxOU&list=PL5Q2soXY2Zi9OhoVQBXyFIZywZXCPI4M_&index=17)



# Lectures on Memory Resource Management

## QoS-Aware Memory Systems: Challenges

- How do we **reduce inter-thread interference**?
  - ❑ Improve system performance and core utilization
  - ❑ Reduce request serialization and core starvation
- How do we **control inter-thread interference**?
  - ❑ Provide mechanisms to enable system software to enforce QoS policies
  - ❑ While providing high system performance
- How do we **make the memory system configurable/flexible**?
  - ❑ Enable flexible mechanisms that can achieve many goals
    - Provide fairness or throughput when needed
    - Satisfy performance guarantees when needed



ETH ZÜRICH HAUPTGEBÄUDE

Computer Architecture - Lecture 11b: Memory Interference and QoS (ETH Zürich, Fall 2020)

735 views • Oct 31, 2020

14 0 SHARE SAVE ...



Onur Mutlu Lectures  
16.5K subscribers

ANALYTICS

EDIT VIDEO

# Lectures on Memory Resource Management

---

- **Computer Architecture, Fall 2020, Lecture 11a**
  - Memory Controllers (ETH, Fall 2020)
  - <https://www.youtube.com/watch?v=TeG773OgiMQ&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=20>
- **Computer Architecture, Fall 2020, Lecture 11b**
  - Memory Interference and QoS (ETH, Fall 2020)
  - <https://www.youtube.com/watch?v=0nnI807nCkc&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=21>
- **Computer Architecture, Fall 2020, Lecture 13**
  - Memory Interference and QoS II (ETH, Fall 2020)
  - <https://www.youtube.com/watch?v=Axye9VqQT7w&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=26>
- **Computer Architecture, Fall 2020, Lecture 2a**
  - Memory Performance Attacks (ETH, Fall 2020)
  - <https://www.youtube.com/watch?v=VJzZbwgBfy8&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=2>

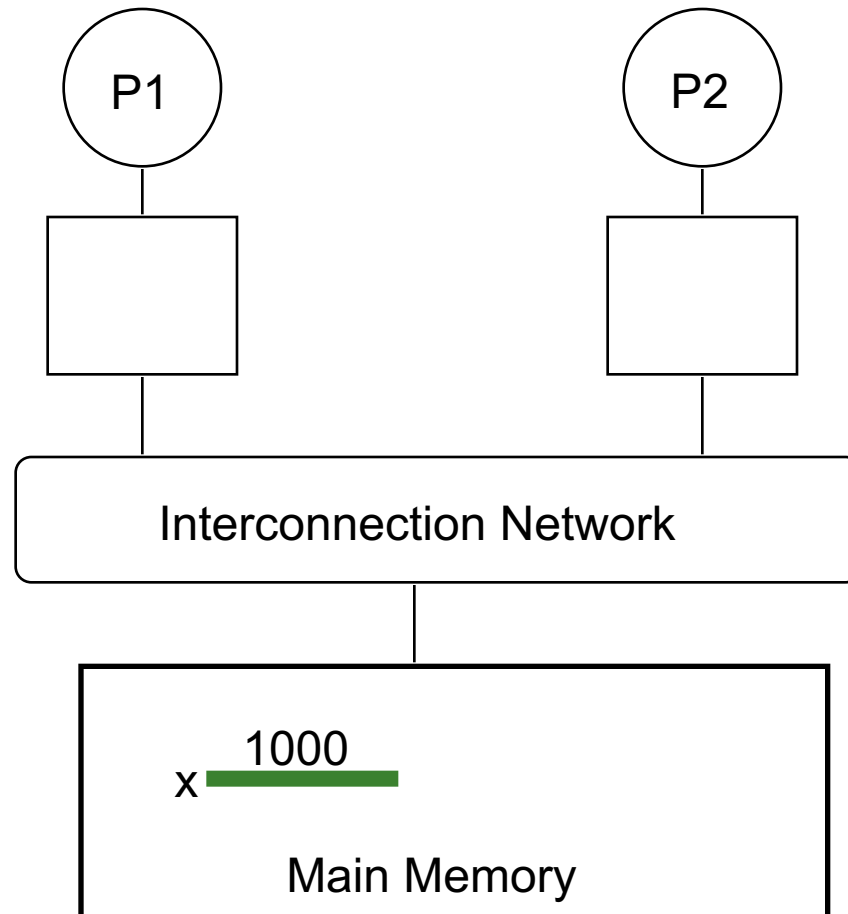


# Cache Coherence

# Cache Coherence

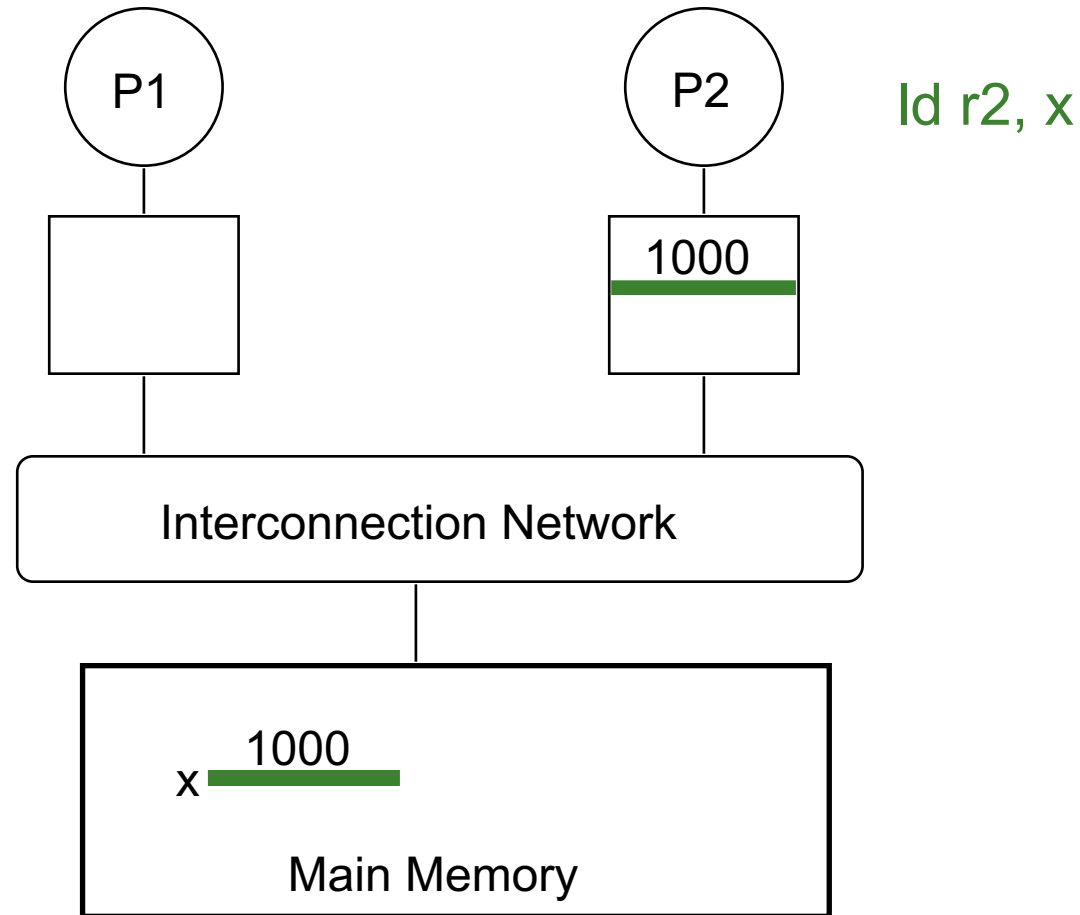
---

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



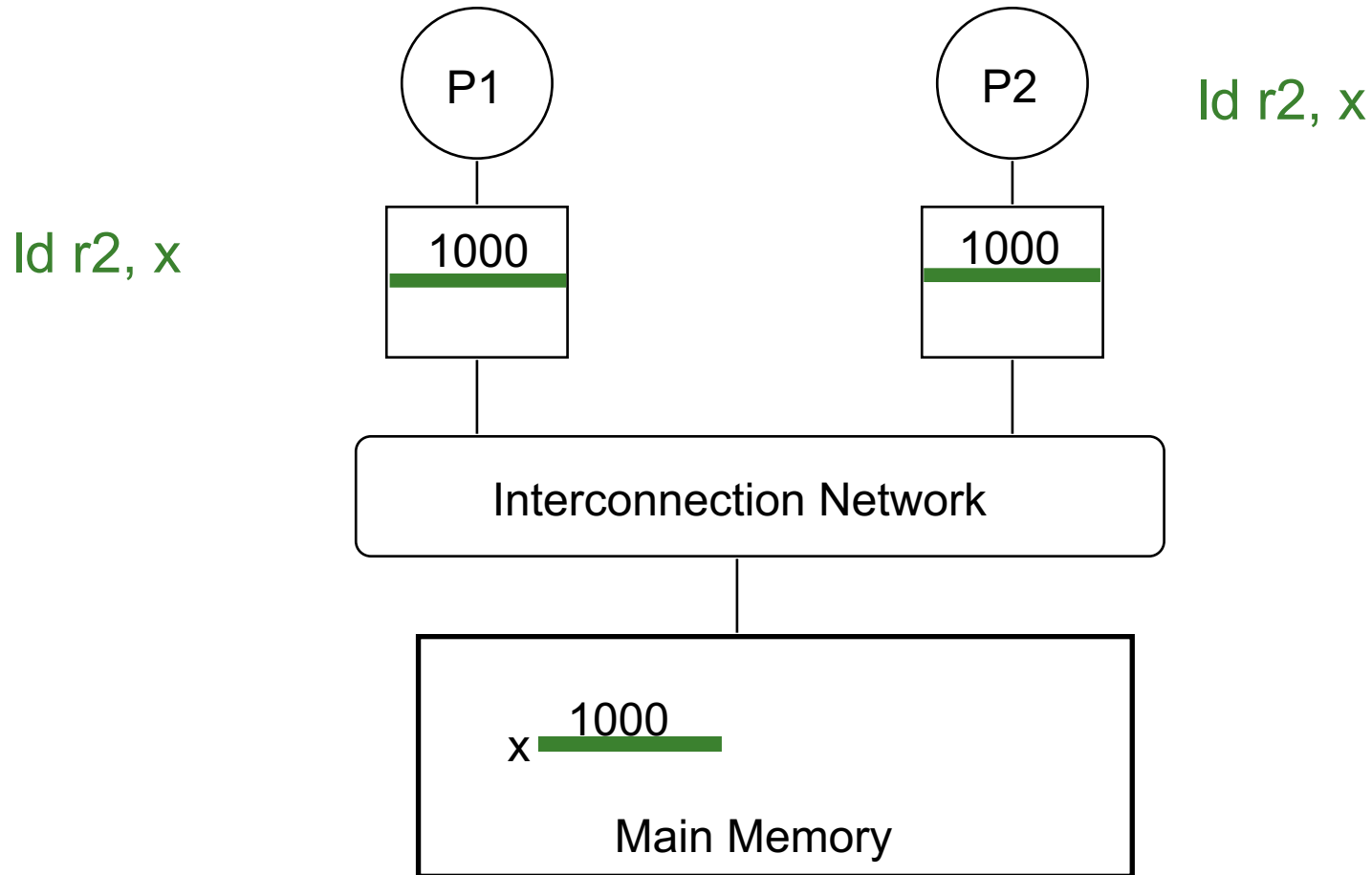
# The Cache Coherence Problem

---



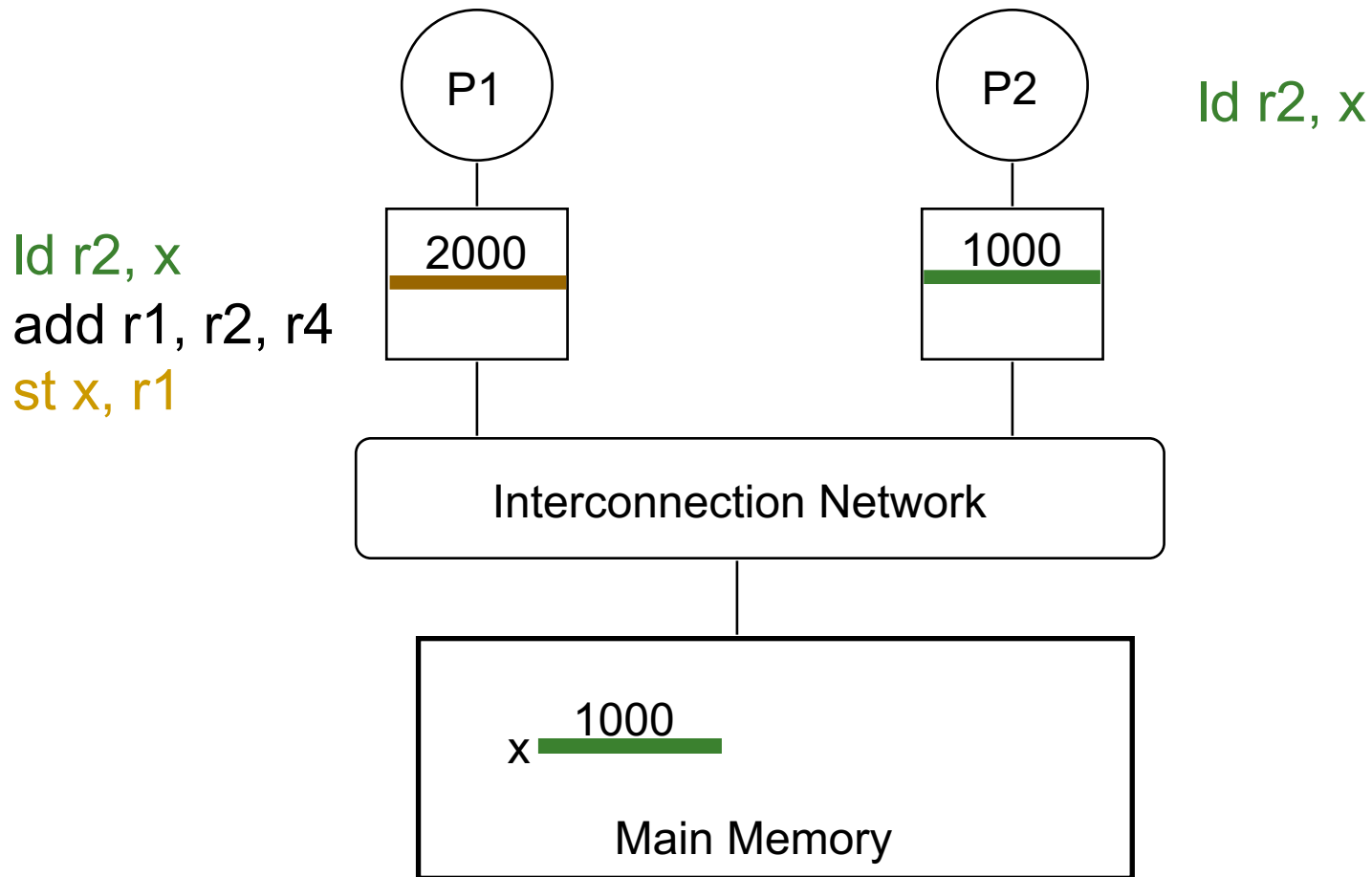
# The Cache Coherence Problem

---



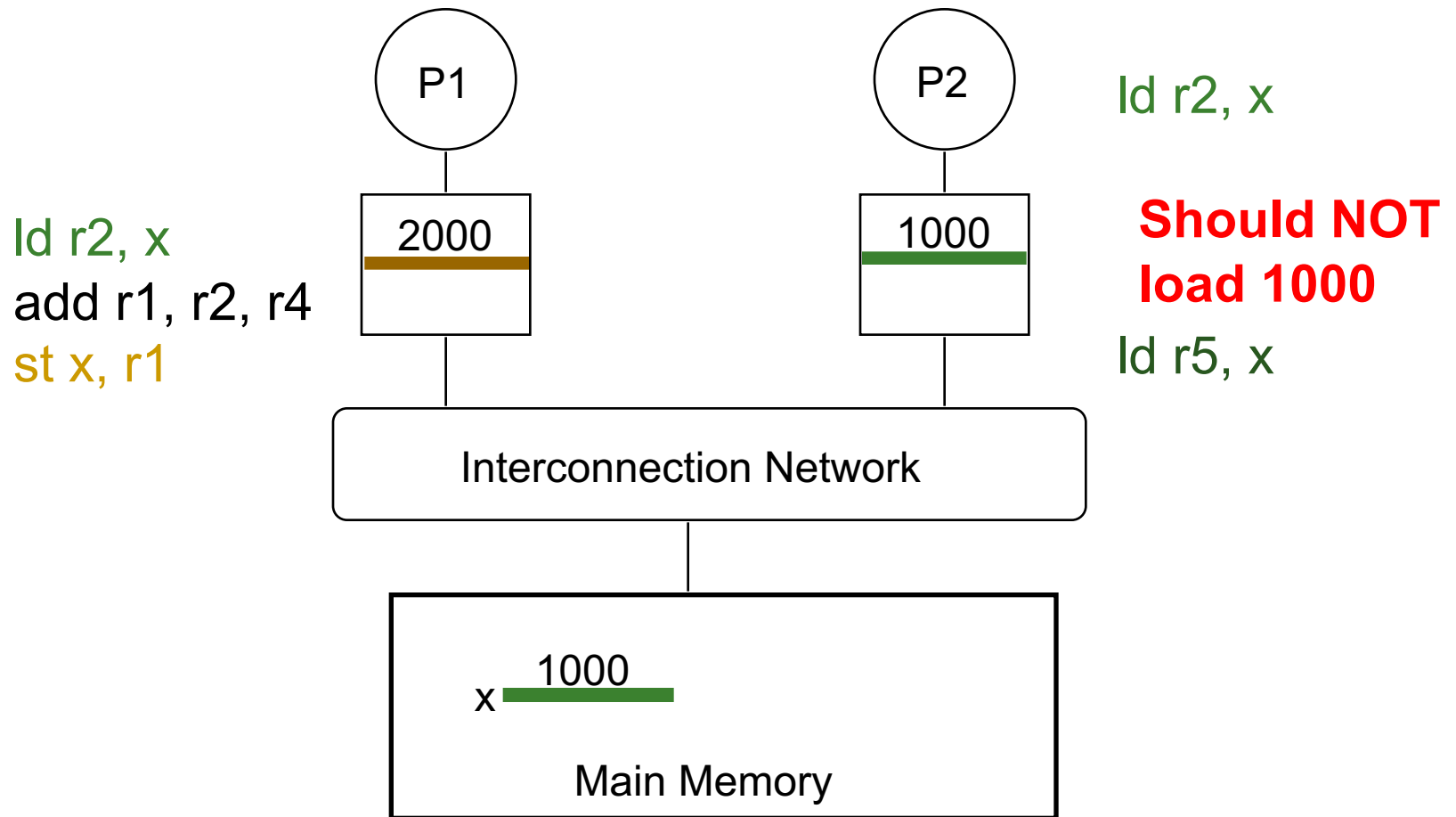
# The Cache Coherence Problem

---



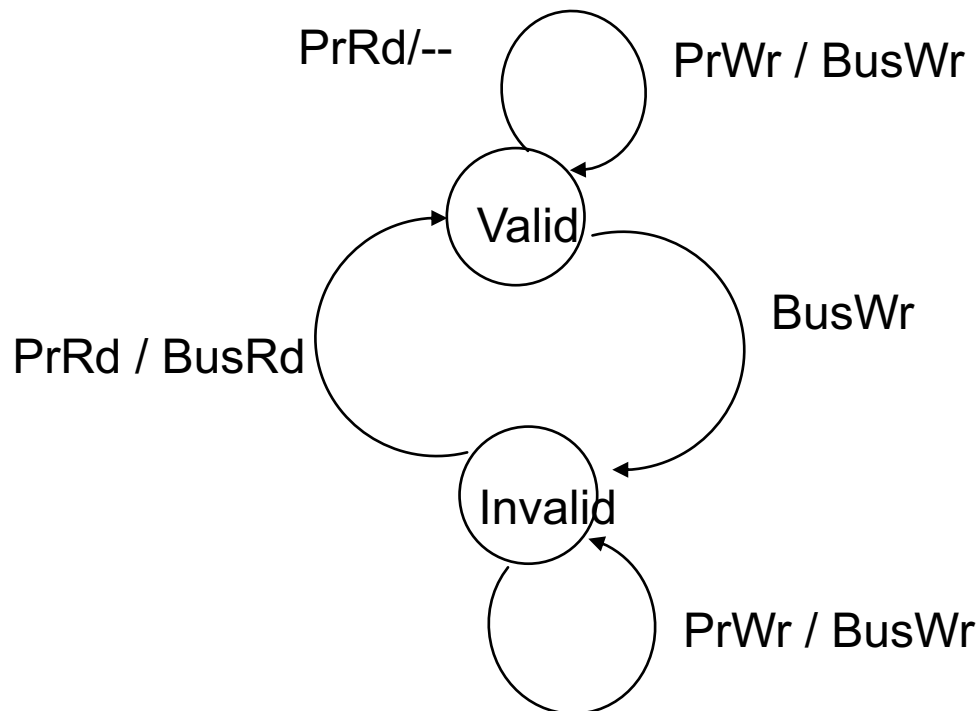
# The Cache Coherence Problem

---



# A Very Simple Coherence Scheme (VI)

- Idea: All caches “snoop” (observe) each other’s write/read operations. If a processor writes to a block, all others invalidate the block.
- A simple protocol:

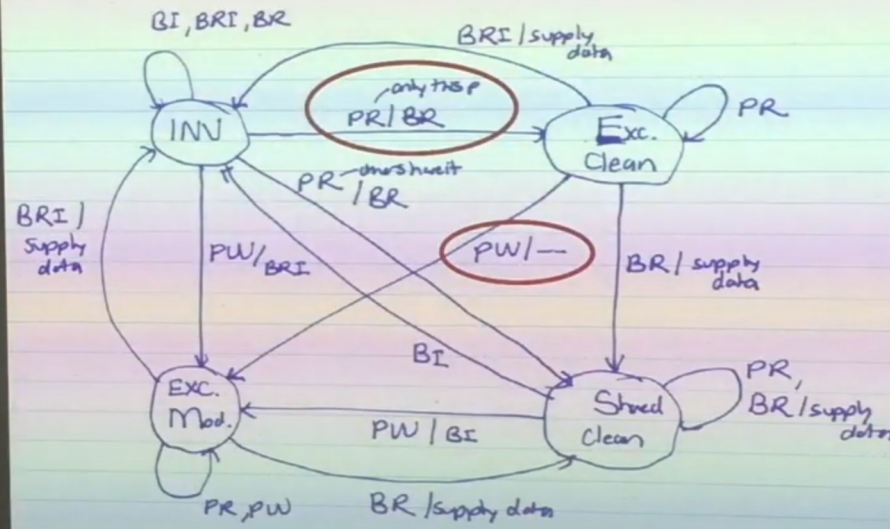


- Write-through, no-write-allocate cache
- Actions of the local processor on the cache block: PrRd, PrWr,
- Actions that are broadcast on the bus for the block: BusRd, BusWr



# Lecture on Cache Coherence

## MESI State Machine



Computer Architecture - Lecture 21: Cache Coherence (ETH Zürich, Fall 2020)

1,419 views • Dec 4, 2020



Onur Mutlu Lectures  
16.3K subscribers

27 0 SHARE SAVE ...

ANALYTICS

EDIT VIDEO

# Lecture on Memory Ordering & Consistency

**Diagram: Interconnection Network**

Processors:  $P_1$  (red),  $P_2$  (blue)

Shared Memory:  $F_1$  (red),  $F_2$  (blue)

**Timeline:**

- 0:  $P_1$ : A (Set  $F_1=1$ )  
•  $F_1=1$  complete from  $P_1$ 's view  
• A sent to memory  
 $P_2$ : X (Set  $F_2=1$ )  
•  $F_2=1$  complete from  $P_2$ 's view  
• X sent to memory
- 1:  $P_1$ : B (test  $F_2=0$ )  
• B (req  $F_2$ ) sent to mem.  
• B (load  $F_2$ ) started  
 $P_2$ : Y (test  $F_1=0$ )  
• Y (req  $F_1$ ) sent to mem.  
• Y (load  $F_1$ ) started
- 50: Mem. sends  $F_2(0)$  to  $P_1$   
• B (load  $F_2$ ) complete  
Mem. sends  $F_1(0)$  to  $P_2$   
• Y (load  $F_1$ ) complete
- 51:  $P_1$  is in Critical Section!  
 $P_2$  is in Critical Section!
- 100: Mem. completes  $F_1=1$  in memory  
TOO LATE!  
Mem. completes  $F_2=1$  in memory  
TOO LATE!

**For  $P_1$ :**  
A appeared to happen before X

**For  $P_2$ :**  
X appeared to happen before A

**$P_1$ 's VIEW**  
 $A \rightarrow B \rightarrow X$   
 $A \rightarrow X$

**$P_2$ 's VIEW**  
 $X \rightarrow Y \rightarrow A$   
 $X \rightarrow A$

**BOTH CANNOT BE CORRECT!**  
(from memory's perspective)

**$P_1$  and  $P_2$  saw an inconsistent order of operations in memory**

Computer Architecture - Lecture 20: Memory Ordering (Memory Consistency) (ETH Zürich, Fall 2020)

976 views • Dec 4, 2020

22 0 SHARE SAVE ...



Onur Mutlu Lectures  
16.5K subscribers

ANALYTICS

EDIT VIDEO

# Lecture on Cache Coherence & Consistency

---

## ■ Computer Architecture, Fall 2020, Lecture 21

- ❑ Cache Coherence (ETH, Fall 2020)
- ❑ <https://www.youtube.com/watch?v=T9WlyezeaII&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=38>

## ■ Computer Architecture, Fall 2020, Lecture 20

- ❑ Memory Ordering & Consistency (ETH, Fall 2020)
- ❑ <https://www.youtube.com/watch?v=Suy09mzTbiQ&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=37>

## ■ Computer Architecture, Spring 2015, Lecture 28

- ❑ Memory Consistency & Cache Coherence (CMU, Spring 2015)
- ❑ <https://www.youtube.com/watch?v=JfjT1a0vi4E&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=32>

## ■ Computer Architecture, Spring 2015, Lecture 29

- ❑ Cache Coherence (CMU, Spring 2015)
- ❑ <https://www.youtube.com/watch?v=X6DZchnMYcw&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=33>