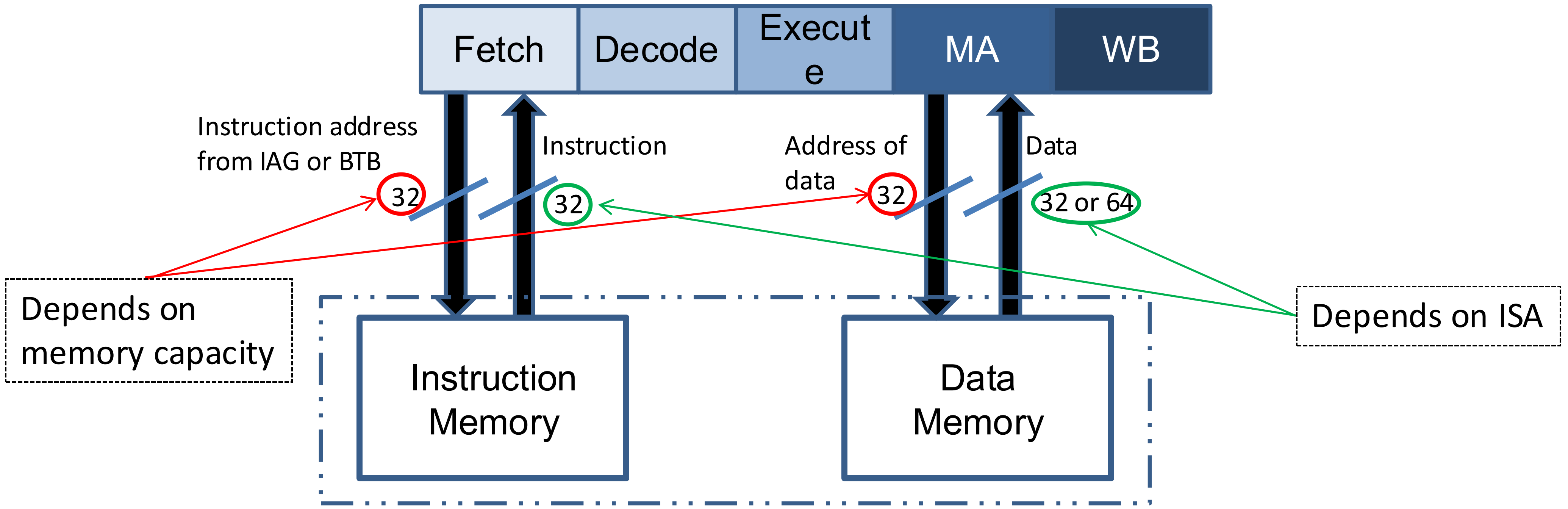
CS6L047: Advanced Computer Architecture

Cache

(Appendix B, Chapter 2, HP 6e)



- We usually treat “Instruction Memory” and “Data Memory” as black boxes so far.
- In this module, we’ll dive in!

Ideal Memory!

- Fast Speed matches with the processor
- Large If it were up to us, we would wish for infinite memory space!
- Low cost Obviously! If not free ☺
- Highly reliable My data is precious to me! ~~Stolen or lost!~~
- High Bandwidth Infinite! Accessing one data or multiple should take same time.
- Consumes less power Increasing interest in this aspect!
- Requires less area Not so pressing at present!

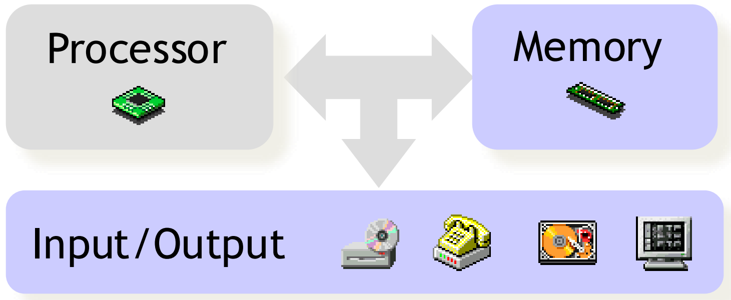
- How to meet these requirements?

5MB hard disk in 1956 by IBM.

1TB micro SD card by Micron. Announced on 28th Feb 2019.

Memory Systems

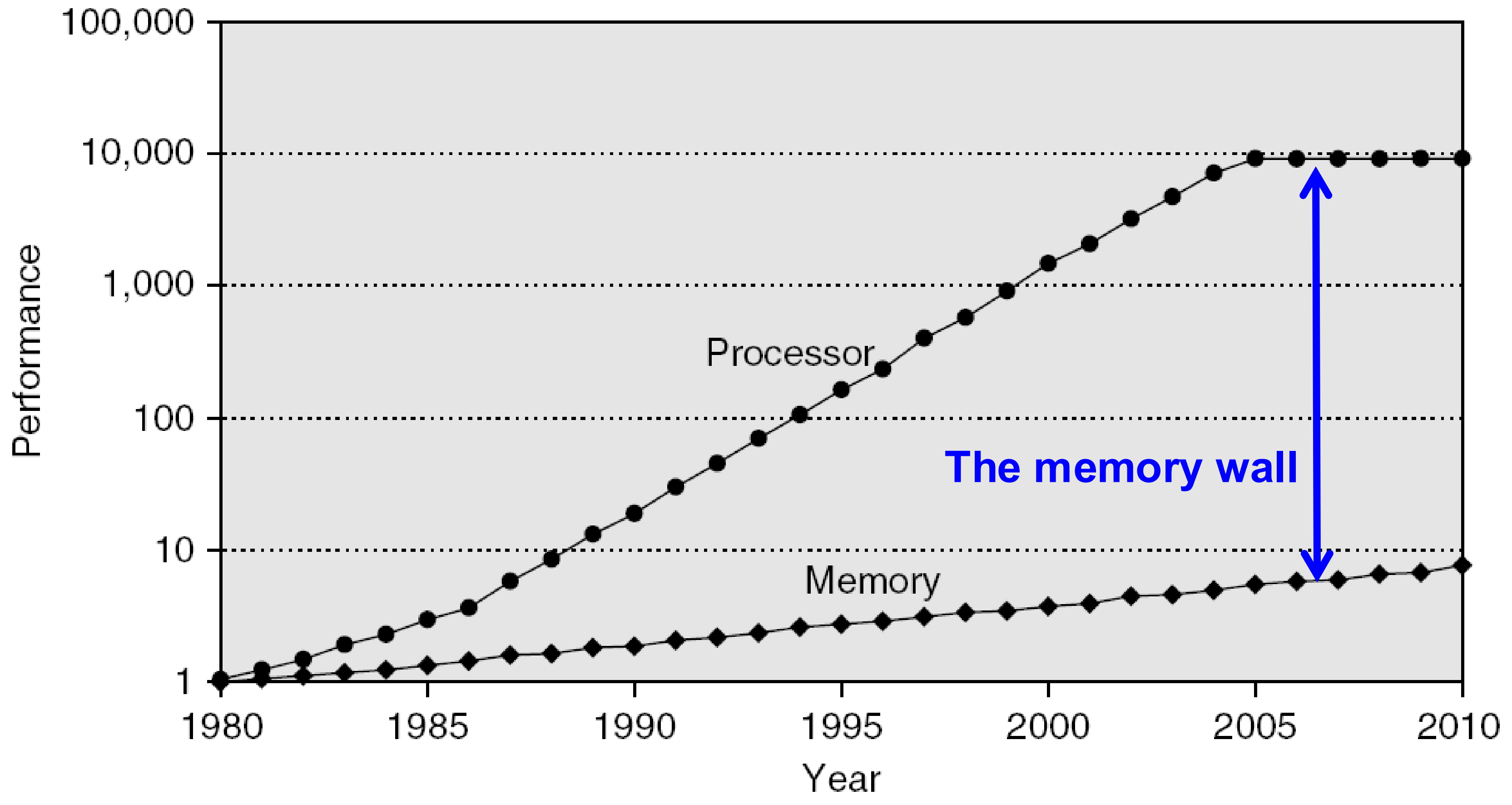
- How can we supply the CPU with enough data to keep it busy?
- We will focus on memory issues,
  - which are frequently bottlenecks that limit the performance of a system.



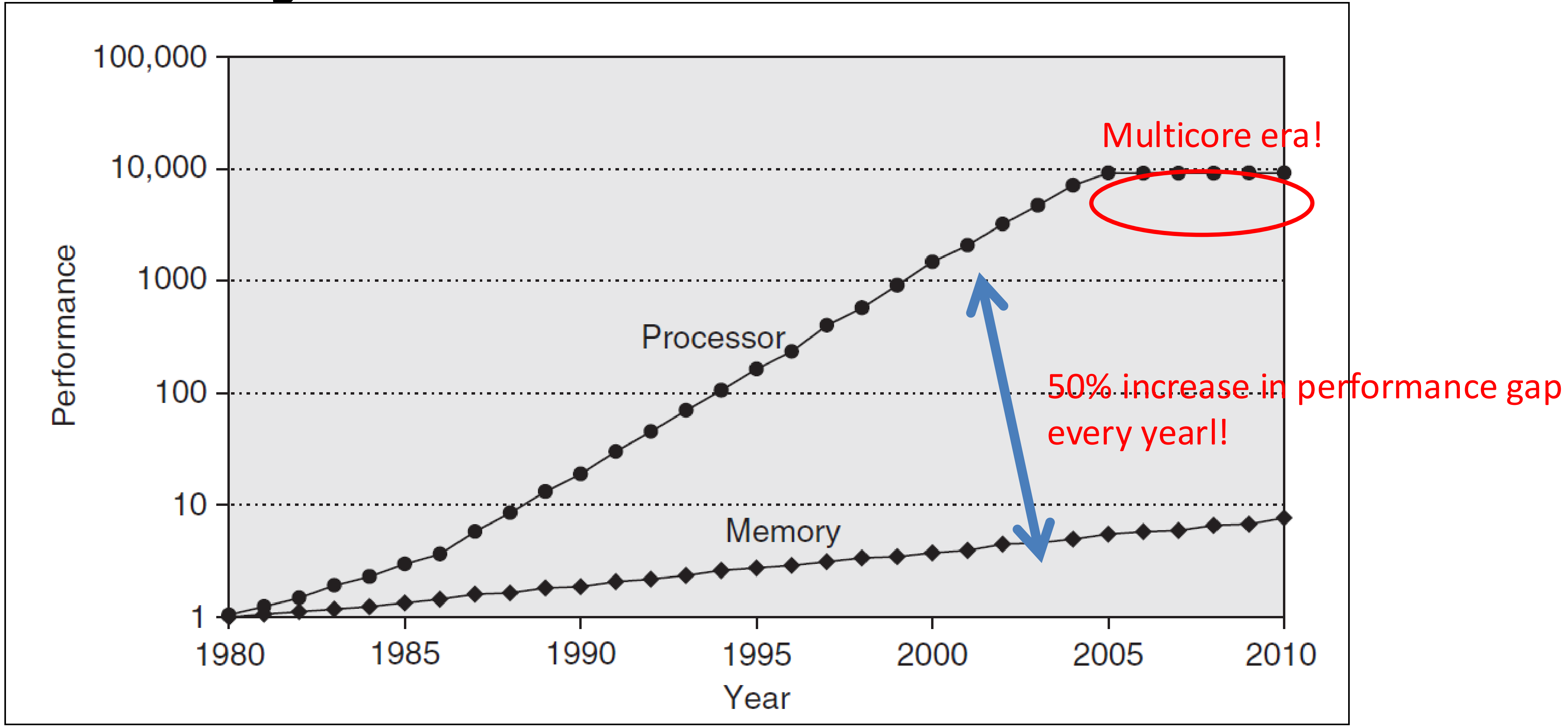
Storage	Speed	Cost	Capacity	Delay	Cost/GB
Static RAM	Fastest	Expensive	Smallest	0.5 – 2.5 ns	$1,000’s
Dynamic RAM	Slow	Cheap	Large	50 – 70 ns	$10’s
Hard disks	Slowest	Cheapest	Largest	5 – 20 ms	$0.1’s

- Ideal memory: large, fast and cheap

Performance Gap



The Memory Wall



- Memory latency was an issue !
- Now, both latency and bandwidth are important !

From, Computer Architecture: A Quantitative Approach 4/e by Hennessey and Patterson

Reality - A classic example

- You are sitting in the Library (preparing for end-sem):
  - Page you are reading
  - Book you are holding
  - Table you are sitting at, with possibly few books of current interest
  - Book racks
- What is the key observation here?
  - At any given moment, we would need only few books to read!
- Something similar with the programs we write –
  - Programs access a relatively small portion of their address space at any instant of time!

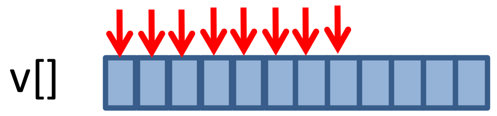
Principle of locality!

  - Predominantly there are two types of locality:
    1. If a data location is referenced, it will tend to be referenced again in near future! Temporal locality!
    2. If a data location is referenced, nearby locations tend to get referenced soon! Spatial locality!

Understanding Locality!

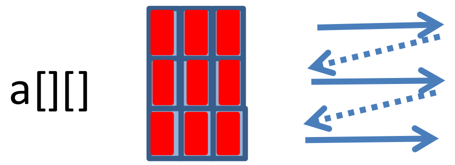
Ex1:

```
int sumvec(int v[N])
{
    int i, sum = 0;

    for (i = 0; i < N; i++)
        sum += v[i];
    return sum;
}
```



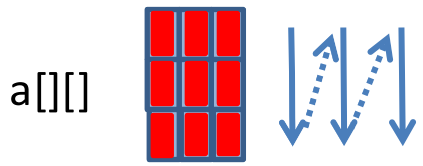
Ex2:

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

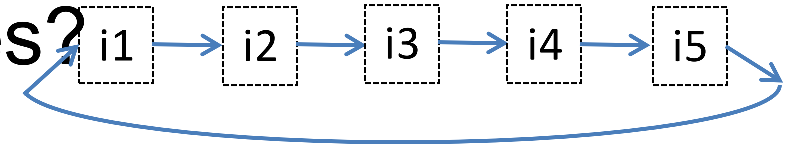


Ex3:

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



- What about locality of instruction fetches?

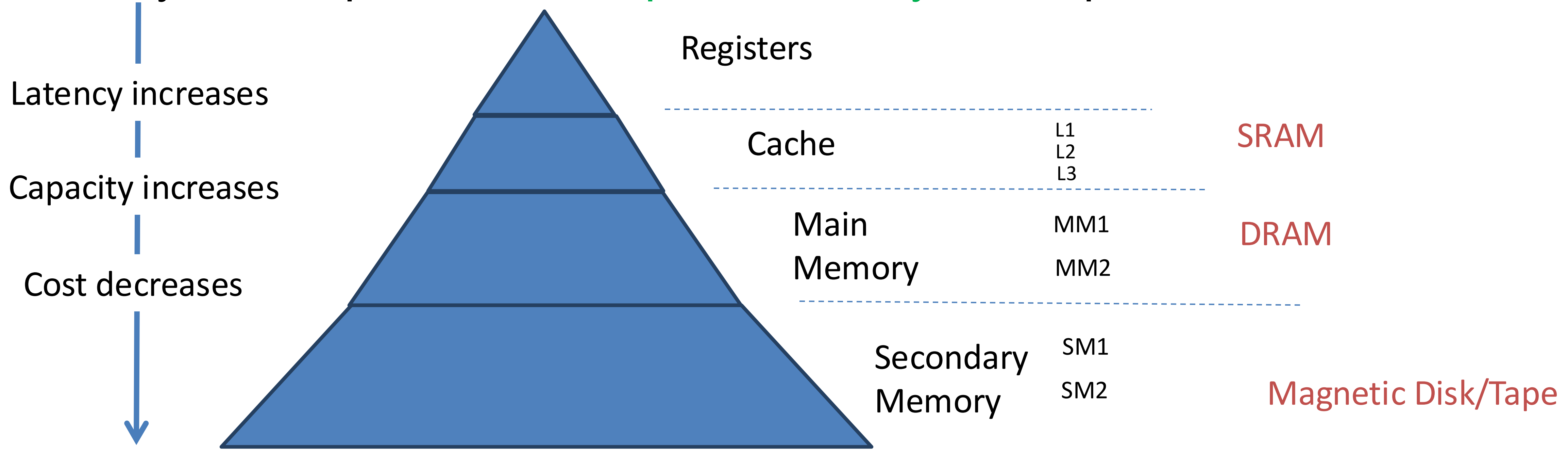


- Summary:
  1. Programs that repeatedly reference the same variables exhibit good temporal locality.
  2. Programs with k-stride access pattern, smaller the size better with the spatial locality.
  3. Loops have good temporal and spatial locality with respect to instruction fetches.

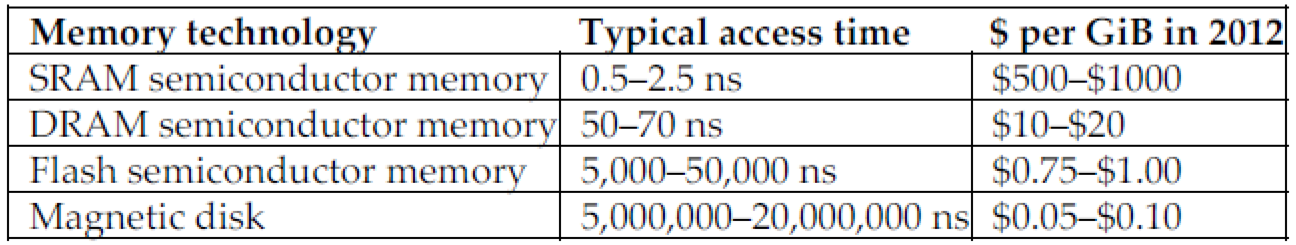
Ref.: Section 6.2 from Computer System: A Programmer's Perspective by Bryant O'Hallaron, 2e.

The Memory Hierarchy!

- Practical way of achieving a fast and large memory at a low cost!
  - Heavily relies upon the 'Principle of Locality' concept.

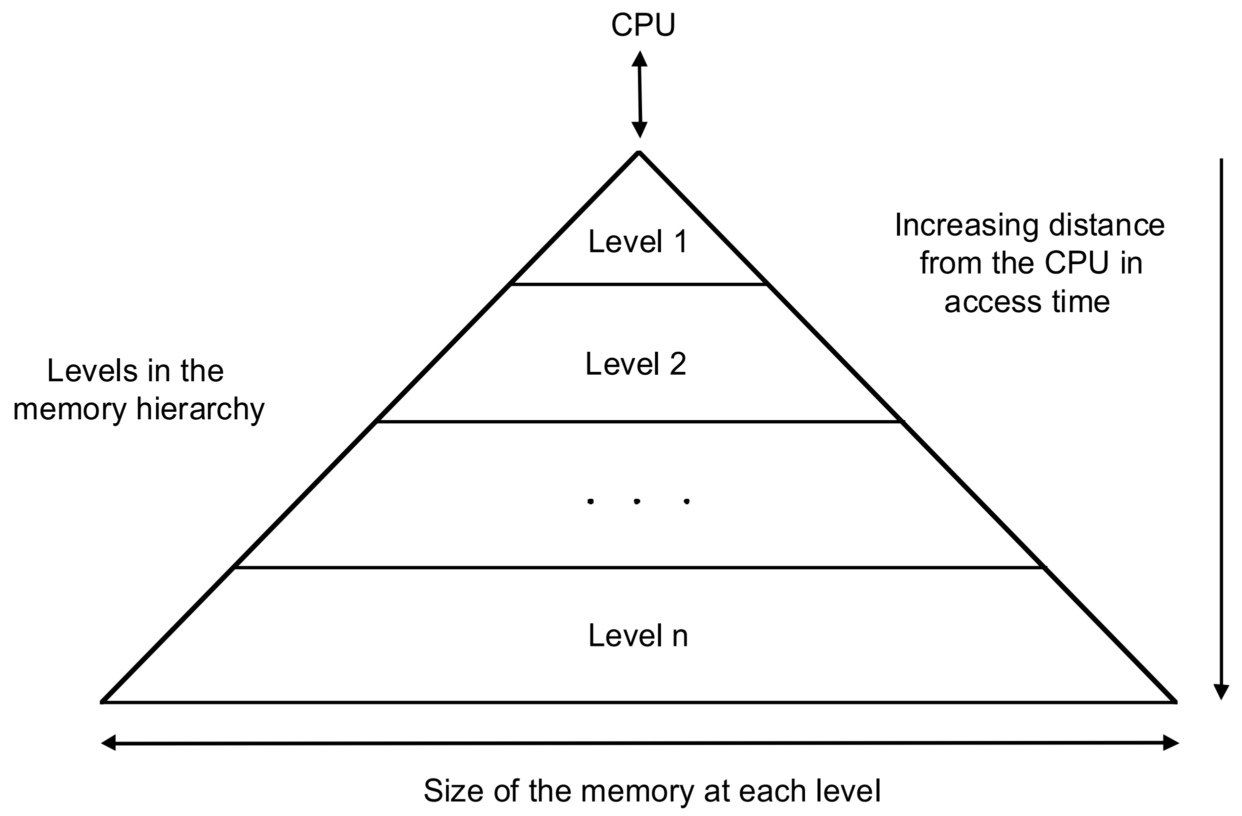


Because of various memory technologies!

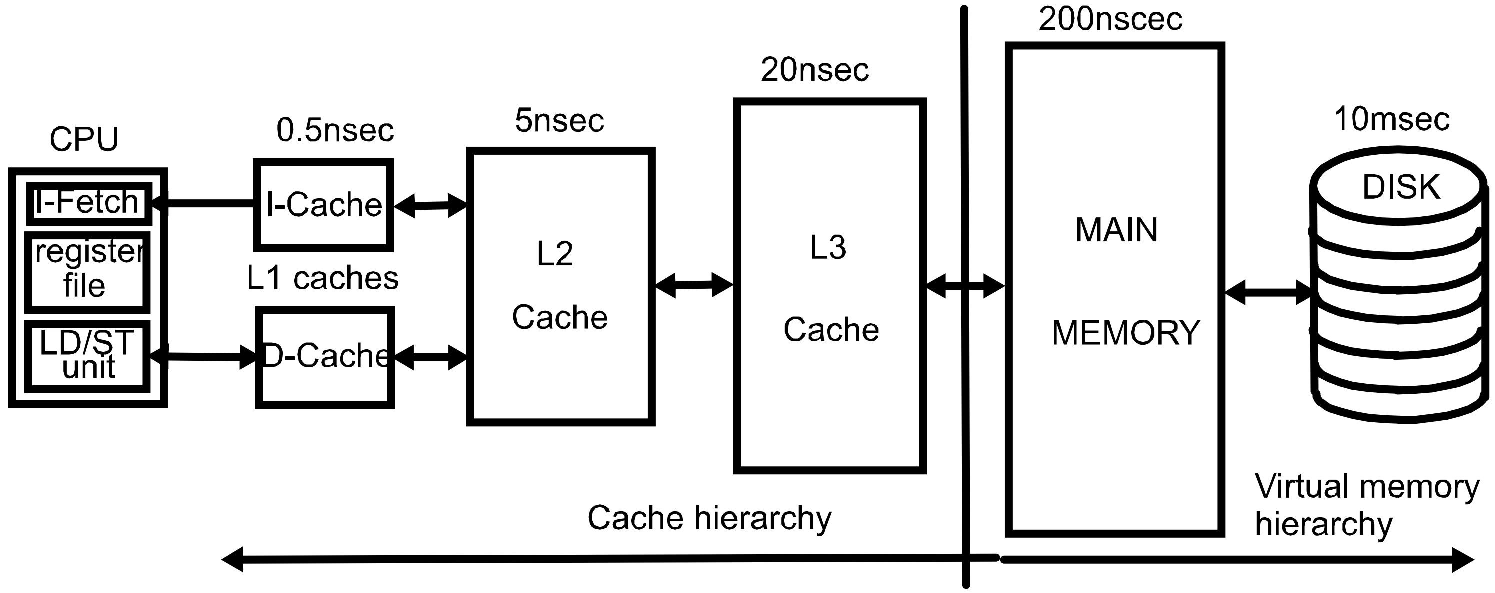


Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10

- Memory hierarchy – put small and fast memories closer to CPU, large and slow memories further away

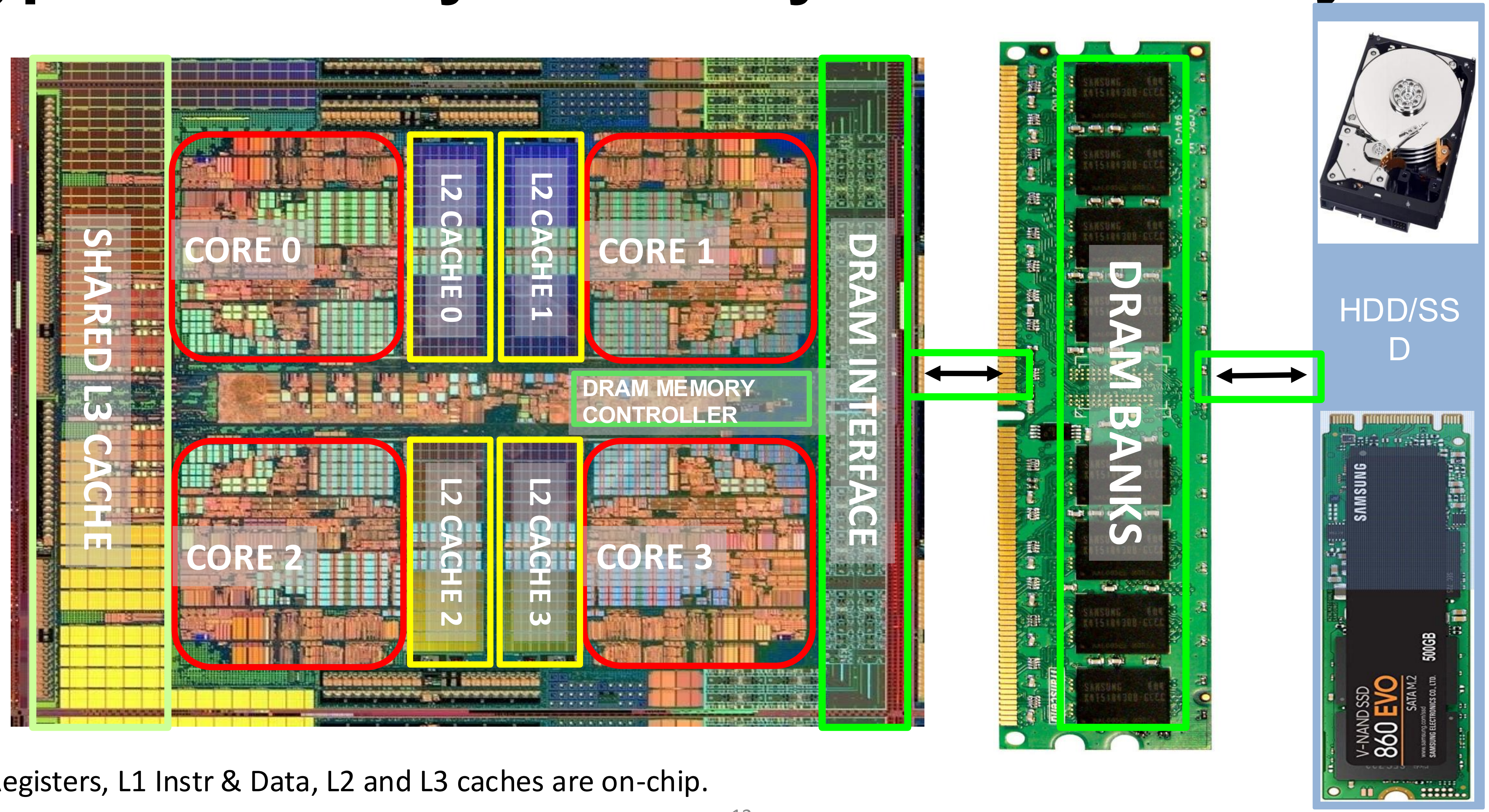

CPU
Level 1
Level 2
. . .
Level n
Levels in the memory hierarchy
Increasing distance from the CPU in access time
Size of the memory at each level

Typical Memory Hierarchy



- Two different types of locality:
  - Temporal locality: if an item is referenced, it will tend to be referenced again soon
  - Spatial locality: if an item is referenced, items whose addresses are close tend to be referenced soon

Typical Memory Hierarchy in a Modern System



Registers, L1 Instr & Data, L2 and L3 caches are on-chip.

Managing Data Movement in Memory Hierarchies

- **Manual Management**
  - **Programmer controls data movement** between memory levels
  - ✅ Useful in:
    - Embedded processors (e.g., on-chip scratchpad SRAM instead of cache)
    - GPUs (e.g., “shared memory”)
  - ❌ **Challenging for large programs**
    - High programming effort
    - Error-prone and hard to scale

Managing Data Movement in Memory Hierarchies

- **Automatic Management**
  - **Hardware handles data movement** transparently
  - ✅ Easier for programmers
    - No need to understand cache size or behavior
    - Programs remain **correct** regardless of memory hierarchy
- **But What About Performance?**
  - To write a **fast** program:
    - You **must understand** the memory hierarchy
    - Optimize data access patterns to exploit caches and memory bandwidth

Time to pause!

CACHE BASICS

Definitions: Hits and misses

- A cache hit occurs if the cache contains the data that we’re looking for.
  - Hits are good, because the cache can return the data much faster than main memory.

- A cache miss occurs if the cache does not contain the requested data.
  - This is bad, since the CPU must then wait for the slower main memory.

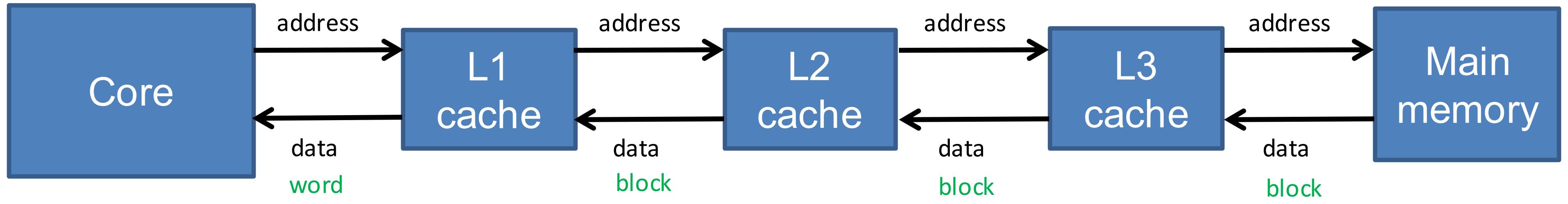
- There are two basic measurements of *cache performance*.
  - The hit rate is the percentage of memory accesses that are handled by the cache.
  - The miss rate (1 – hit rate) is the percentage of accesses that must be handled by the slower main RAM.

Cache memory

- Static RAM (SRAM) technology
  - Each cell is made up of 6 transistors (6T)
  - A cell stores one bit (`0’ or `1’)
- ‘Static’ because it does not need Refresh to retain cell state.
  - DRAM needs Refresh, we’ll look at it later.
- Volatile in nature.
- Multiple levels of cache are most common.
  - Working of all the levels is same.

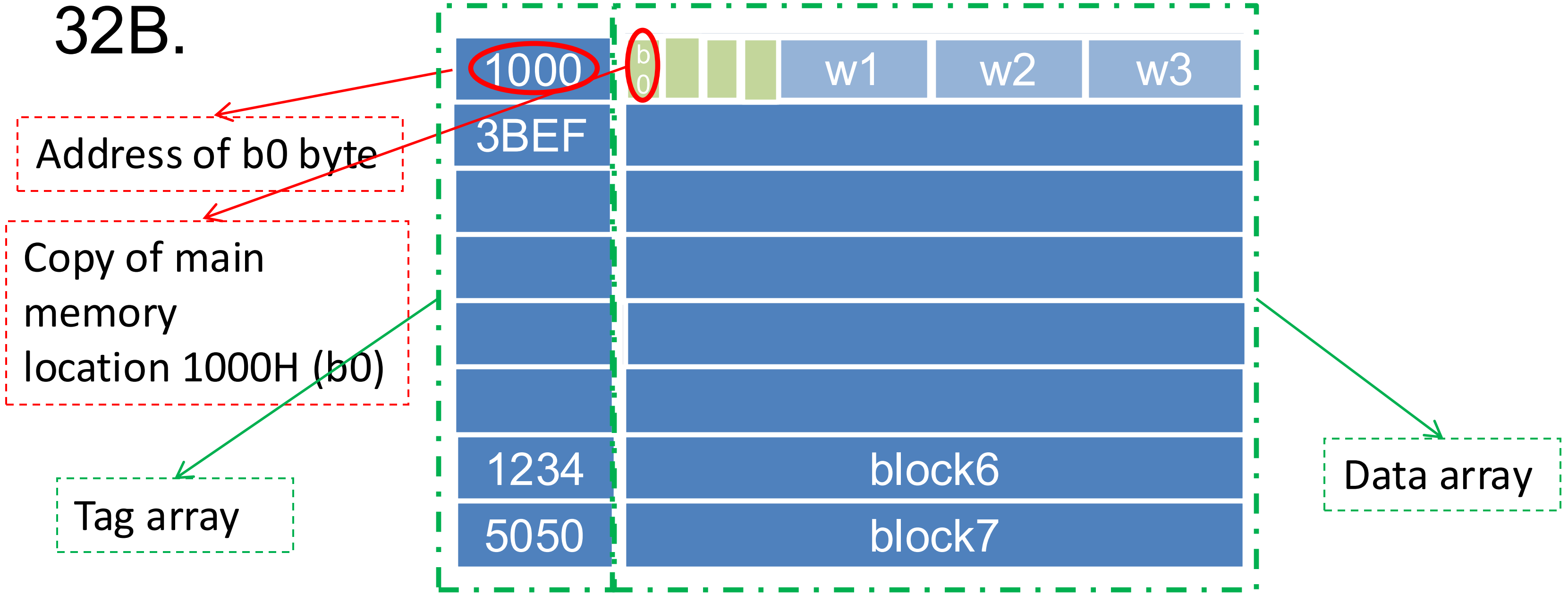
Why are we taking a block of data instead of a word?

To exploit spatial locality!!



Cache internals

- Cache basically is a collection of blocks.
- A cache block or cache line is set of words (4B or 8B).
  - Typical cache block sizes: 32B, 64B.
- Cache size = Number of blocks * block size
  - Ex: A 32KB cache is made up of 512 blocks of 64B each or 1024 blocks of 32B.



8 blocks of 16B.

4B words => 4 words per block.

1 word has 4*8 SRAM cells, i.e., 32

1 block has 4*32 SRAM cells, i.e., 128

This 8 block cache

Data array has 1024 SRAM cells.

What about Tag array?

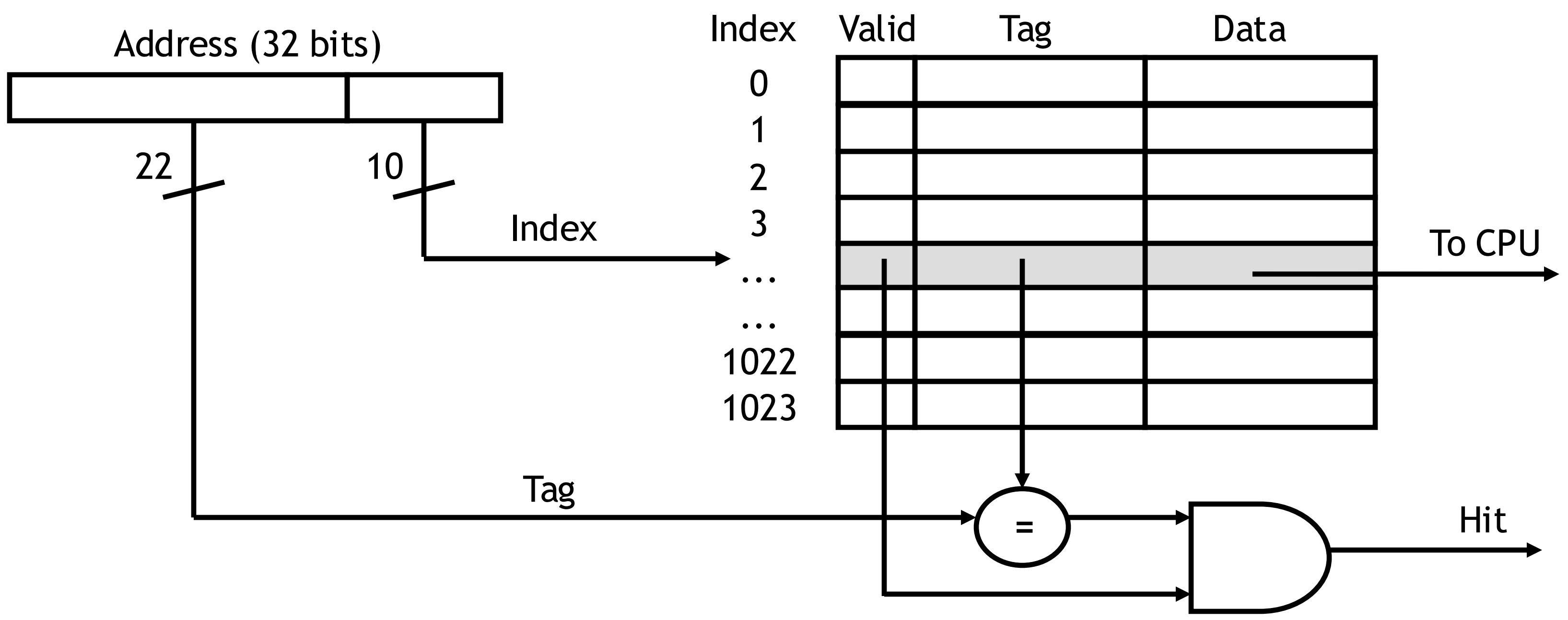
Cache working

- Upon receiving a address from the processor, the control circuitry in the cache (namely, cache controller) will
  - Compare this address with those of Tag array.
    1. If there is a match, then its called Cache Hit.
       - Cache controller will access Data array and send a copy of the requested word to the processor.
    2. Else Cache Miss.
       - Cache controller will forward this address to the next level in the hierarchy.
- Four questions to understand overall working of the cache:

1. Where can a block be placed?	Block placement
2. How is a block found?	Block identification
3. Which block to replace on a miss?	Block replacement
4. What happens on a write?	Write strategy

What happens on a cache hit

- When the CPU tries to read from memory, the address will be sent to a cache controller.
  - The lowest *k* bits of the block address will index a block in the cache.
  - If the block is valid and the tag matches the upper (*m* – *k*) bits of the *m*-bit address, then that data will be sent to the CPU.
- Here is a diagram of a 32-bit memory address and a 2^{10}-byte cache.

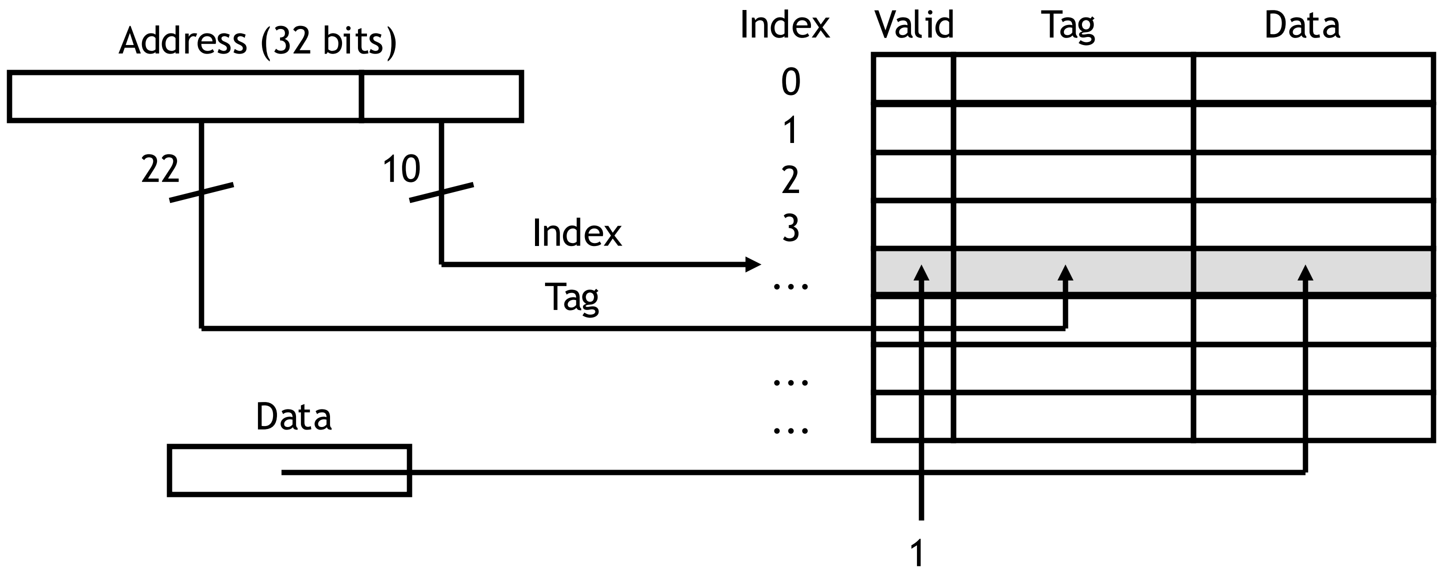


What happens on a cache miss

- On cache hit, CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy
  - Instruction cache miss
    - Restart instruction fetch
  - Data cache miss
    - Complete data access
- The delays that we have been assuming for memories (e.g., 1 cycle) are really assuming cache hits.

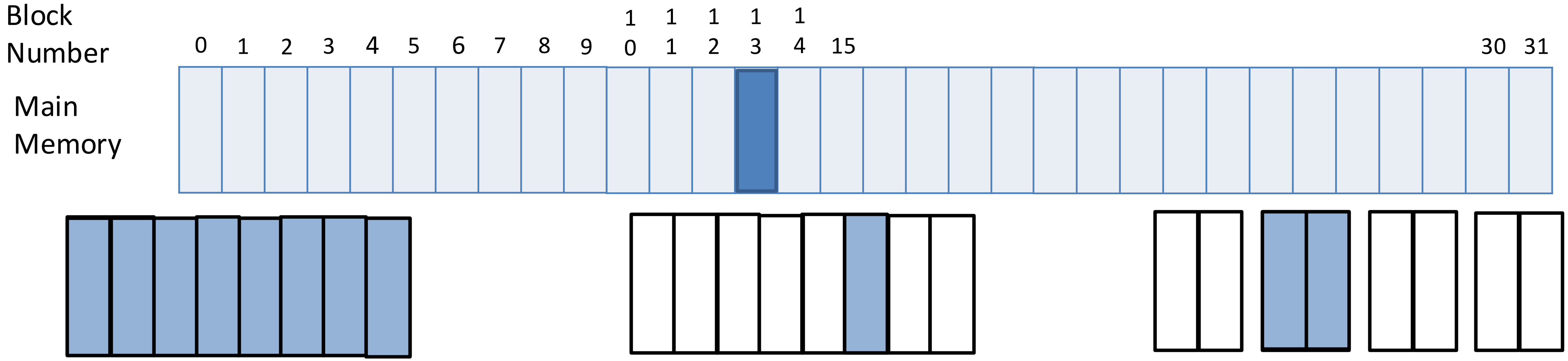
Loading a block into the cache

- After data is read from main memory, putting a copy of that data into the cache is straightforward.
  - The lowest *k* bits of the block address specify a cache block.
  - The upper (*m* – *k*) address bits are stored in the block's tag field.
  - The data from main memory is stored in the block's data field.
  - The valid bit is set to 1.



Cache working - Q1 Block Placement

1. Where can a block be placed in cache?
   1. A block can be placed anywhere in the cache — Fully-associative
   2. A block has only one place it can appear in cache — Direct-mapped
   3. A block can be placed in a restricted number of places in the cache — Set-associative
      - n possible places? n-way set associative cache.



Fully-associative

Block 13 can be placed anywhere

Direct-mapped

Block 13 can be placed only in set 5 (13 mod 8)

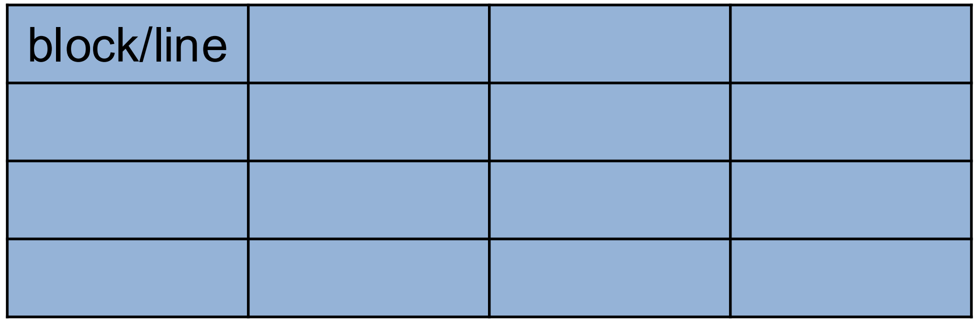
2-way Set-associative

Block 13 can be placed anywhere in set 1 (13 mod 4)

Cache Sets and Ways

Ways: Block can go anywhere

Sets:
Block
mapped
by
addr

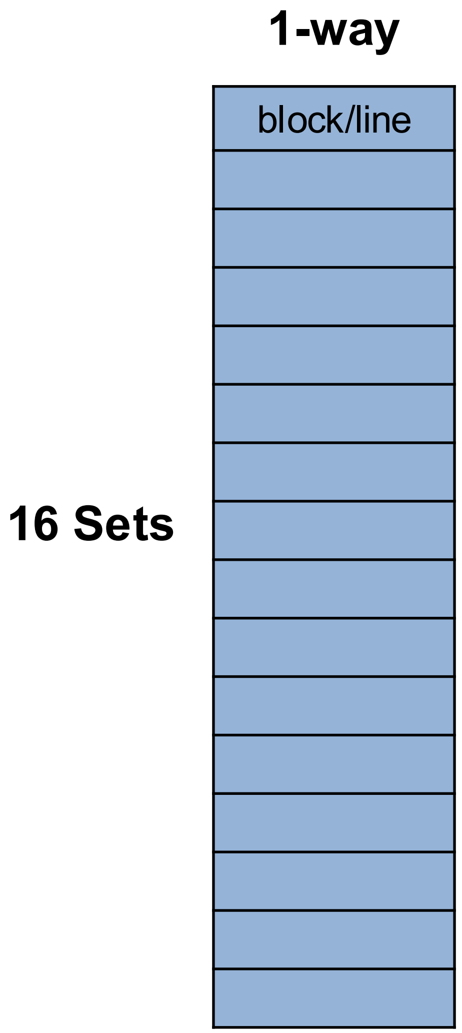


n-way set associative

(4-way set associative)

Example: Cache size = 16 blocks

Direct-mapped Cache

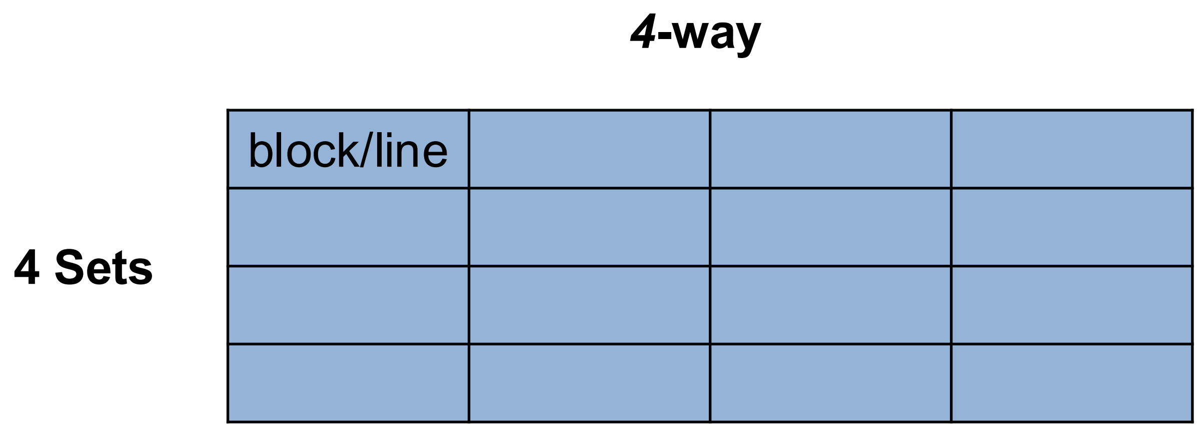


Direct mapped cache
Each block maps to only one cache line

aka

1-way set associative

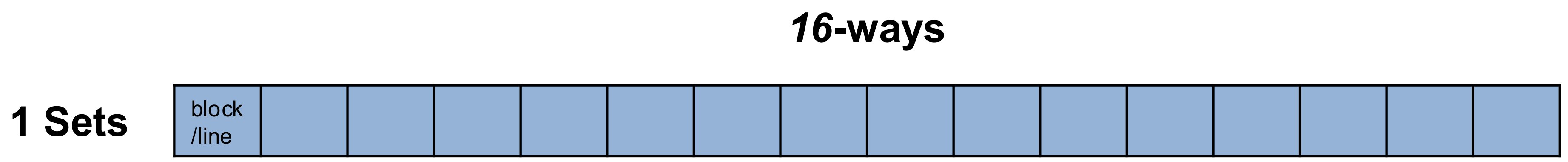
Set Associative Cache



n-way set associative
Each block can be mapped to a set of *n*-lines
Set number is based on block address

(4-way set associative)

Fully Associative Cache

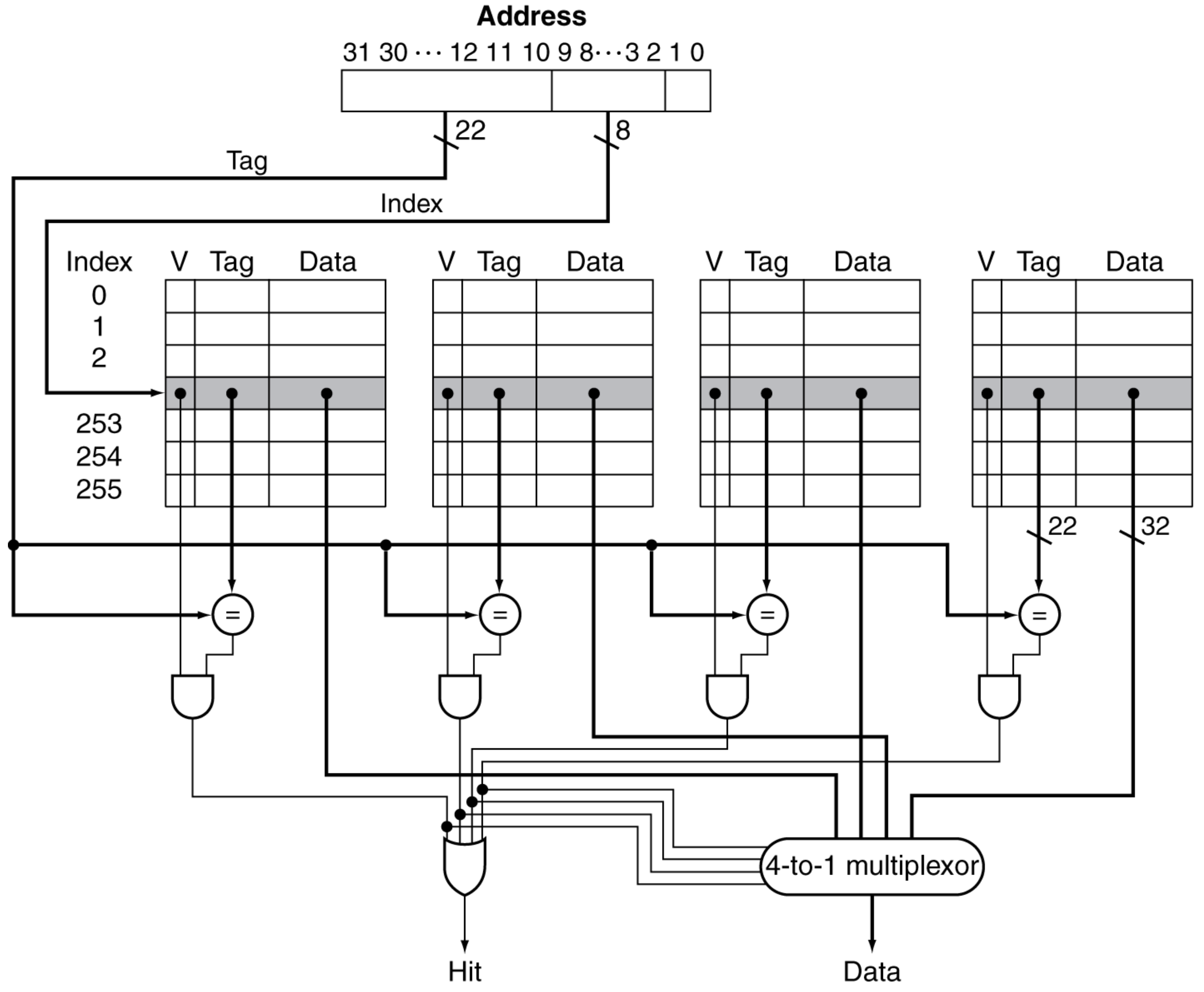


Fully associative
Each block can be mapped to any cache line

aka

m-way set associative
where *m* = size of cache in blocks

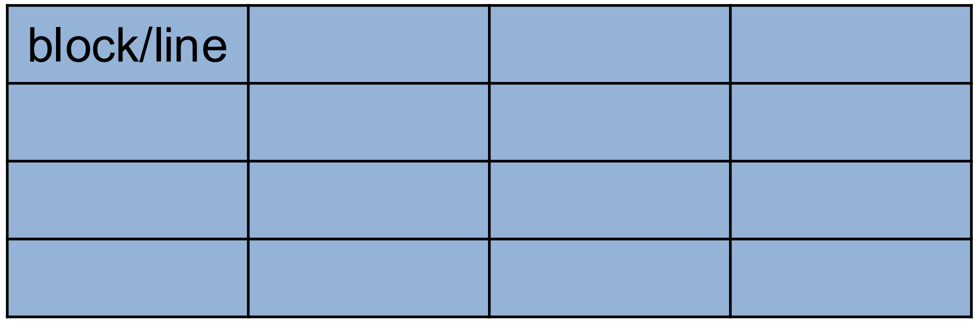
Set Associative Cache Organization



Cache Addressing

***n*-Ways**: Block can go anywhere

***s*-Sets**: Block mapped by addr



Address	Tag (remainder) bits = 32-s-b	Index (sets) bits = $\log_2 s$	Offset (block size) bits = $\log_2 b$

s = number of sets
n = number of ways
b = block size in bytes

Cache size = $s * n * b$
(area of a rectangle?)

Cache Addressing

Ex. 64KB cache, direct mapped, 16 byte block

Address	Tag (remainder) bits = 32-s-b	Index (sets) bits = $\log_2 s$	Offset (block size) bits = $\log_2 b$
	16	12	4

s = number of sets
n = number of ways
b = block size in bytes

Cache size = $s * n * b$

Cache Addressing

Ex. 64KB cache, 2-way assoc., 16 byte block

Address	Tag (remainder) bits = $32-s-b$	Index (sets) bits = $\log_2 s$	Offset (block size) bits = $\log_2 b$
	17	11	4

s = number of sets
n = number of ways
b = block size in bytes

Cache size = $s * n * b$

Cache Addressing

Ex. 64KB cache, fully assoc., 16 byte block

Address

Tag (remainder) bits = 32-s-b	Index (sets) bits = $\log_2 s$	Offset (block size) bits = $\log_2 b$
28	0	4

s = number of sets
n = number of ways
b = block size in bytes

Cache size = $s * n * b$

What if the cache fills up?

- What to do if we run out of space in our cache, or if we need to reuse a block for a different memory address?

- A miss causes a new block to be loaded into the cache, automatically overwriting any previously stored data.
    - Which block do we overwrite?

- Cache replacement policies!

CACHE REPLACEMENT POLICIES

Replacement Policy

- Direct mapped: no choice

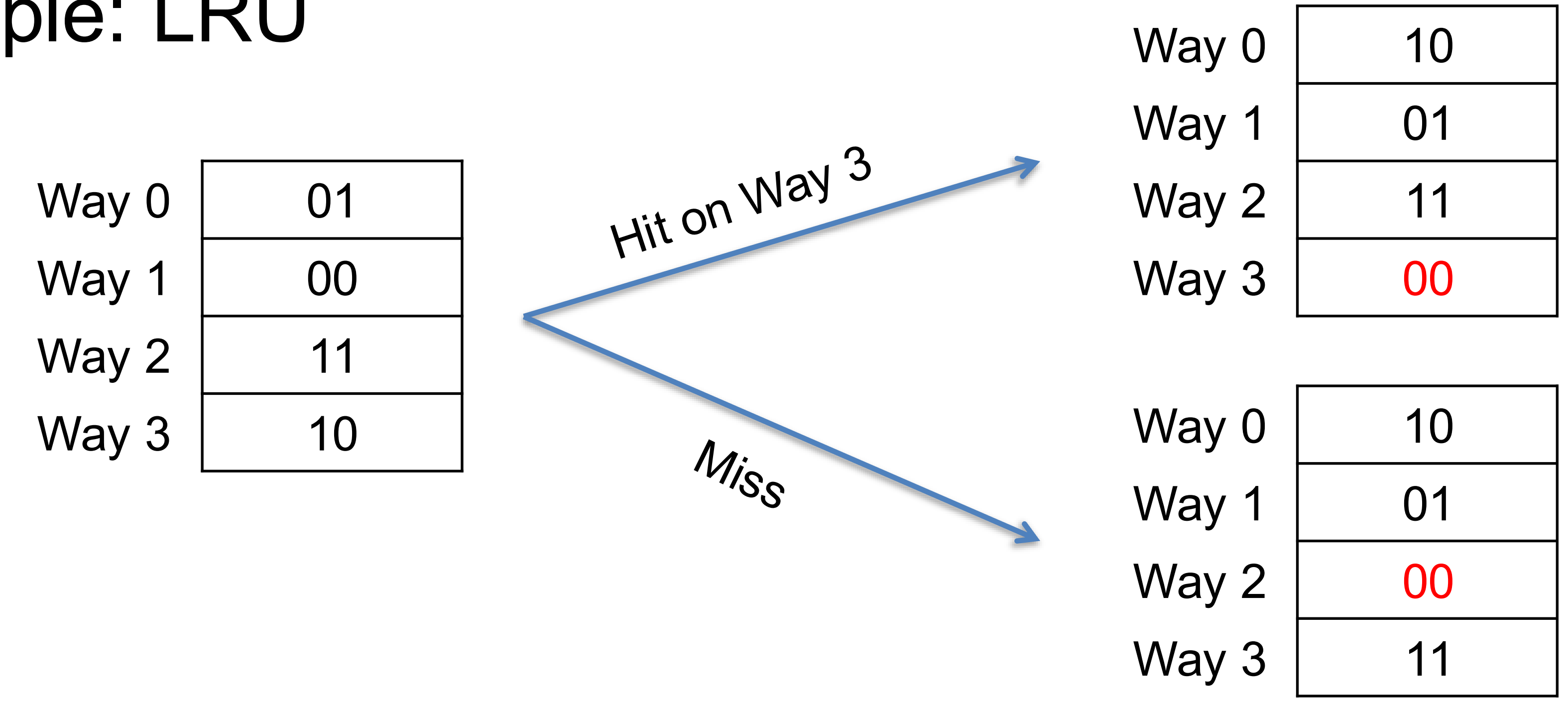
- Set associative
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set

- Least-recently used (LRU)
  - Choose the one unused for the longest time
    - Simple for 2-way, manageable for 4-way, too hard beyond that

- Random
  - Gives approximately the same performance as LRU for high associativity

Cache Replacement Policies

- Picks which block to replace within the set
- Ex. - Random, First In First Out (FIFO), Least Recently Used (LRU), Psuedo-LRU
- Example: LRU

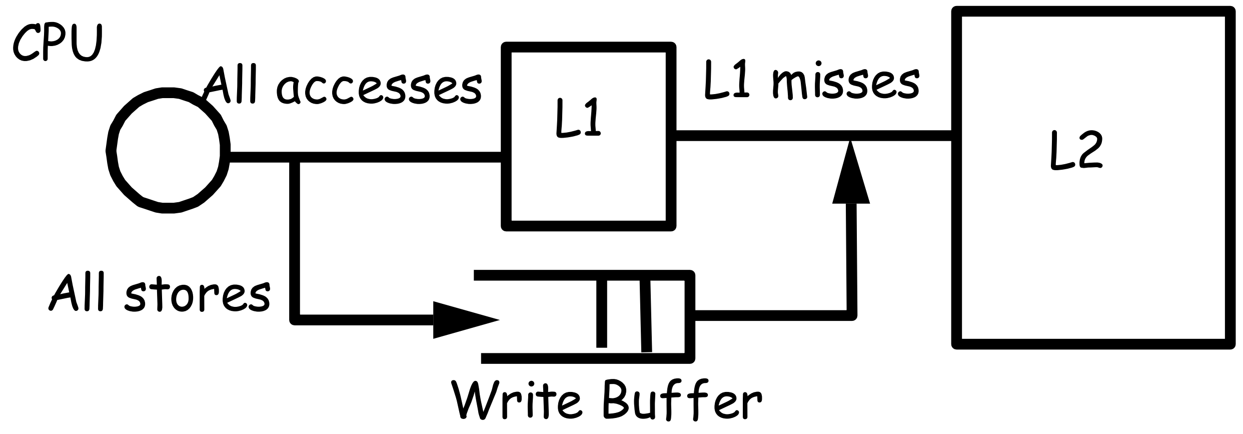


WRITEBACK POLICIES

Previous slides mainly handles memory reads.
How about memory writes?

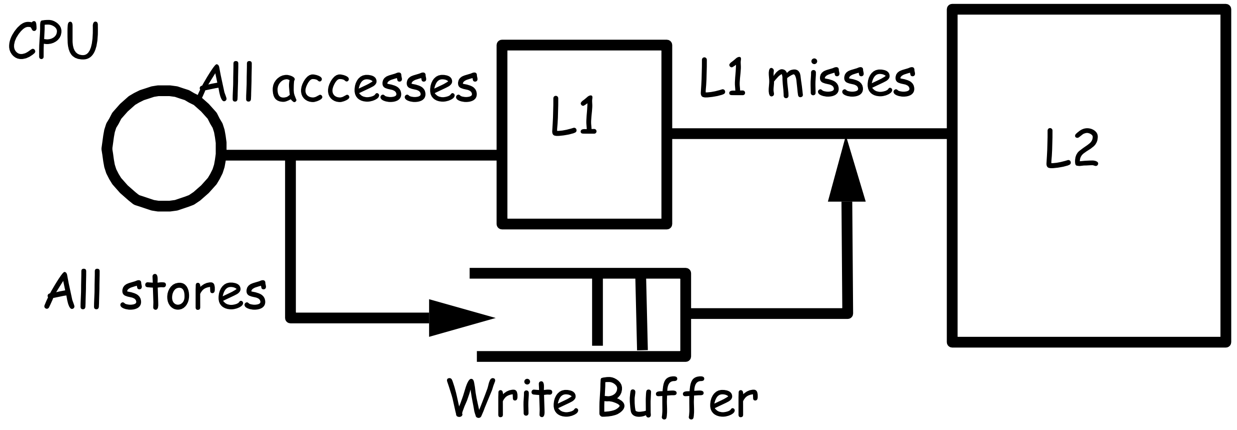
Write-Through

- On data-write hit, could just update the block in cache
  - But then cache and memory would be inconsistent
- Write through: also update memory



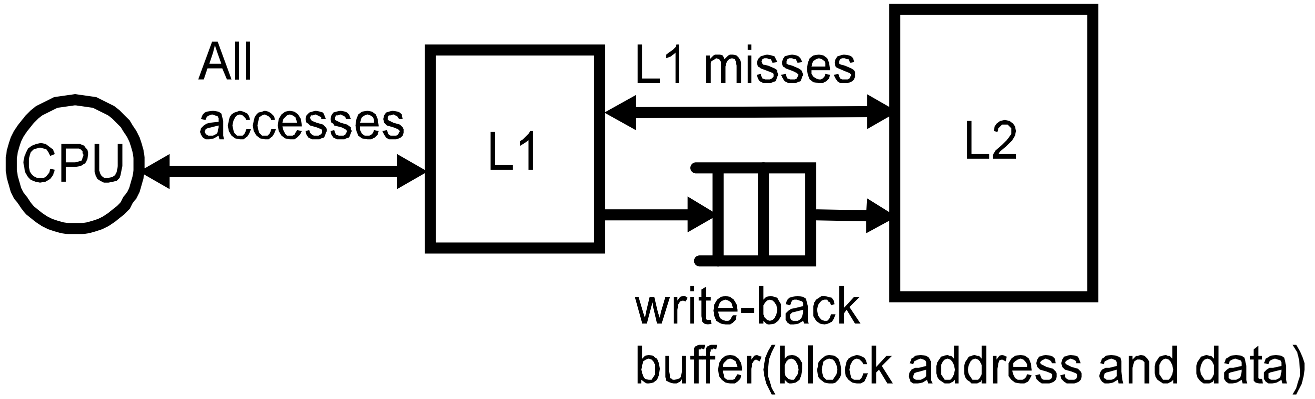
Write-Through

- But makes writes take longer
  - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Average CPI = 1 + 0.1 × 100 = 11
- Solution: write buffer (Holds data waiting to be written to memory)
  - CPU continues immediately
    - Only stalls on write if write buffer is already full



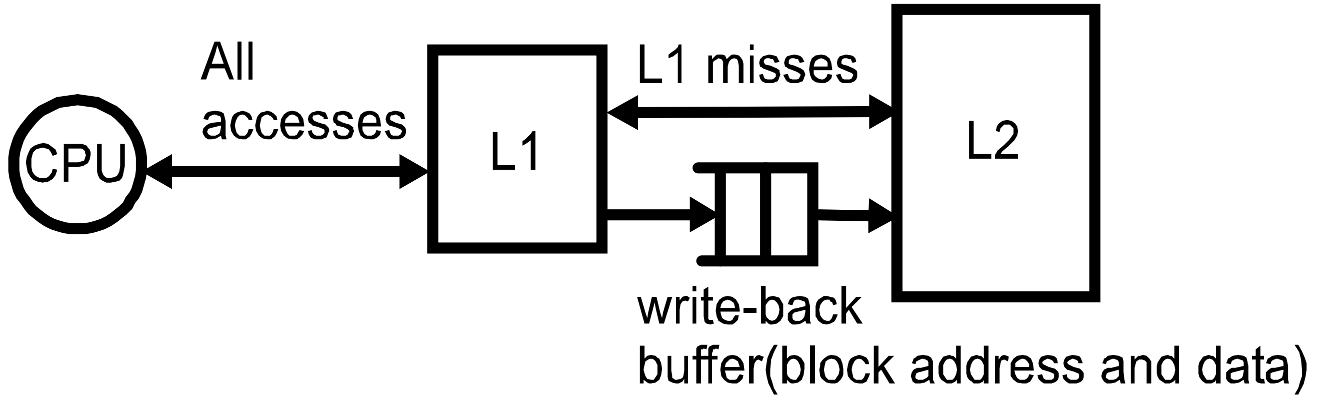
Write-Back

- Alternative: On data-write hit, just update the block in cache
  - Keep track of whether each block is dirty
- When a dirty block is replaced
  - Write it back to memory
  - Can use a write buffer to allow replacing block to be read first



Write Allocation

- What should happen on a write miss?



- For write-back: Usually fetch the block
- Alternatives for write-through
  - Allocate on miss: fetch the block
  - Write around: don’t fetch the block

CACHE PERFORMANCE

Measuring Cache Performance

- Components of CPU time
  - Program execution cycles: Includes cache hit time
  - Memory stall cycles: Mainly from cache misses

- With simplifying assumptions:

$$\text{Memory stall cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

Measuring Cache Performance

- I-cache miss rate = 2%, D-cache miss rate = 4%, Miss penalty = 100 cycles,
  Base CPI (ideal cache) = 2,
  Load & stores are 36% of instructions

- What is the actual CPI?

- Memory stall cycles per instruction to:
  - D-cache: 0.36 × 0.04 × 100 = 1.44
  - I-cache: 1.0 × 0.02 × 100 = 2
- Actual CPI = 2 + 2 + 1.44 = 5.44

$$\text{Memory stall cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

Average Access Time

- Hit time is also important for performance
- Average memory access time (AMAT)
  - AMAT = Hit time + Miss rate $\times$ Miss penalty

- Example
  - CPU with 1ns clock, hit time = 1 cycle,
    miss penalty = 20 cycles, I-cache miss rate = 5%
  - AMAT = 1 + 0.05 $\times$ 20 = 2ns

Progress so far ……

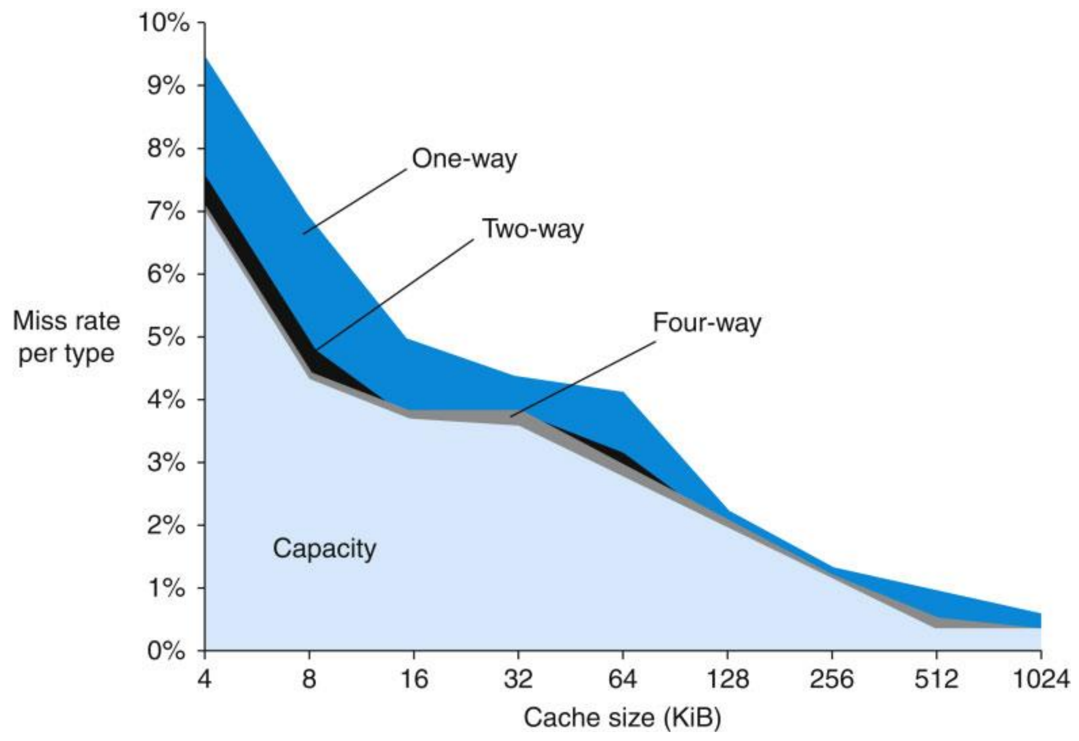
- Four questions to understand overall working of the cache:
  1. Where can a block be placed? **Block Placement**
     - Cache organization and Design
     - Cache Addressing , sets, ways
  2. How is a block found? **Block Identification**
     - Measure Cache Performance, Average Memory Access Time (AMAT)
  3. Which block to replace on a miss? **Replacement Policies**
     - Depends on Cache Organization [Direct/Set Associative/Fully Associative]
     - FIFO, L**R**U (Recently), Random, L**F**U (Frequently)
  4. What happens on a write? **Write Back Policies**
     - Write back/ Write Through
     - Write Allocation

Cache Performance

- Miss rate
  - Fraction of cache access that result in a miss
- Causes of misses (3C’s)
  - Compulsory (cold)
    - First reference to a block
  - Capacity
    - Space is not sufficient to host data or code
  - Conflict
    - When two memory blocks map on the same cache block in direct-mapped or set-associative caches

Sources of Misses – 3C's

- Compulsory misses (aka cold start misses)
  - **First access** to a block
  - Unavoidable when data is accessed for the first time
- Capacity misses
  - Due to **finite cache size**
  - A **replaced block** is later accessed again
- Conflict misses (aka collision misses)
  - In a **non-fully associative cache**
  - Due to competition for entries in a set
  - **Would not occur** in a fully associative cache of the same total size



Measuring/Classifying Misses

- How to find out each type?

- Cold misses:
  - Simulate a fully-associative infinite cache size

- Capacity misses:
  - Simulate fully-associative cache, then deduct cold misses

- Conflict misses:
  - Simulate target cache configuration then deduct cold and capacity misses

Measuring/Classifying Misses

- Classification is useful to understand how to eliminate misses

- High capacity misses
  - → need larger cache

- High conflict misses
  - → need higher associativity

CACHE OPTIMIZATIONS

Think of Cache Like a Fast-Access Bookshelf

Small and snḍ simple first-level caches

- Like keeping only your most-used books on a small shelf near your desk for guick access

Critical timing path = Steps to grab the right book

1. Find the shelf (addressing tag memory)
2. Check the book title (comparing tags)
3. Pick-the correct book (selecting the right set)

Direct-mapped caches

- Like having only one spot for each book—easy to check and grab at the same time

Lower associativity saves power

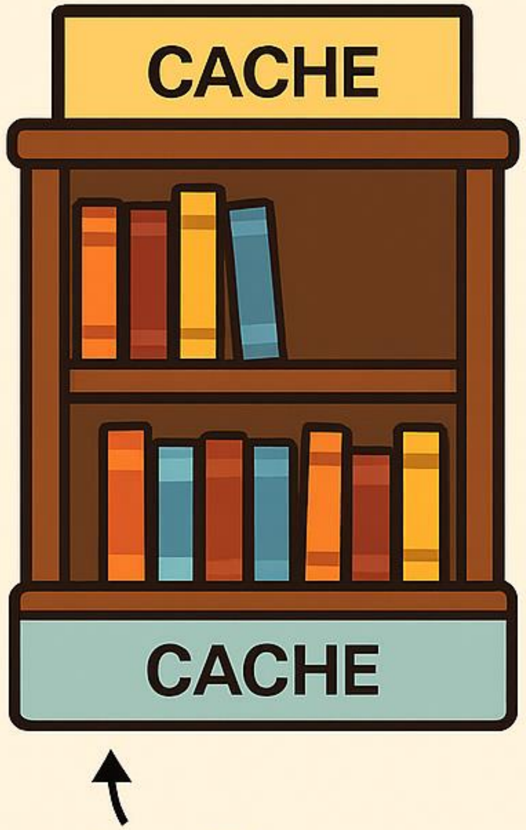
- Fewer shelves to scan means less effort (fewer cache lines accessed)

Measuring/Classifying Misses

- Classification is useful to understand how to eliminate misses

- High capacity misses
  - → need larger cache

Reduce Capacity Misses: Make the Shelf Bigger



Trade-offs

- Longer hit time: Bigger shelf = more scanning before you find the book

Idea: Increase the cache size so more data fits inside

Imagine your desk bookshelf is too small, and you keep running to the main library. Adding a bigger shelf means fewer trips—but it take more space, costs more, and might slow you down when searching

Trade-offs

- Longer hit time: Bigger shelf = more scanning before you find the book
- Higher cost and power: Larger furniture needs more material and energy
- Popular for off-chip caches: Like adding a big storage cabinet outside your room—great for bulk storage

For off-chip caches

Objective: Minimize Miss Rate and Miss Penalty

Think of your cache like a personal bookshelf The goal is to avoid running to the library (main memory) and save time when you do

1. Increase Block Size
→ Bring Bigger Book Bundles

✅ Reduces compulsory misses

⚠ Like borrowing a whole series instead of one book – you need fewer trips

Drawbacks: Bigger bundles take more space and time to carry → may increase capacity & conflict misses and miss penalty

3. Increase Associativity
→ Add More Slots per Shelf

✅ Reduces conflict misses

⚠ Like having flexible spots for books – less chance of eviction

Drawbacks: More complexity = slower lookup and higher energy → increases **hit time** and power consumption

3. Increase Associativity
→ Add More Slots per Shelf

⚠ Like having flexible spots for books – less chance of

Cache Optimization Objective: Minimize the miss rate

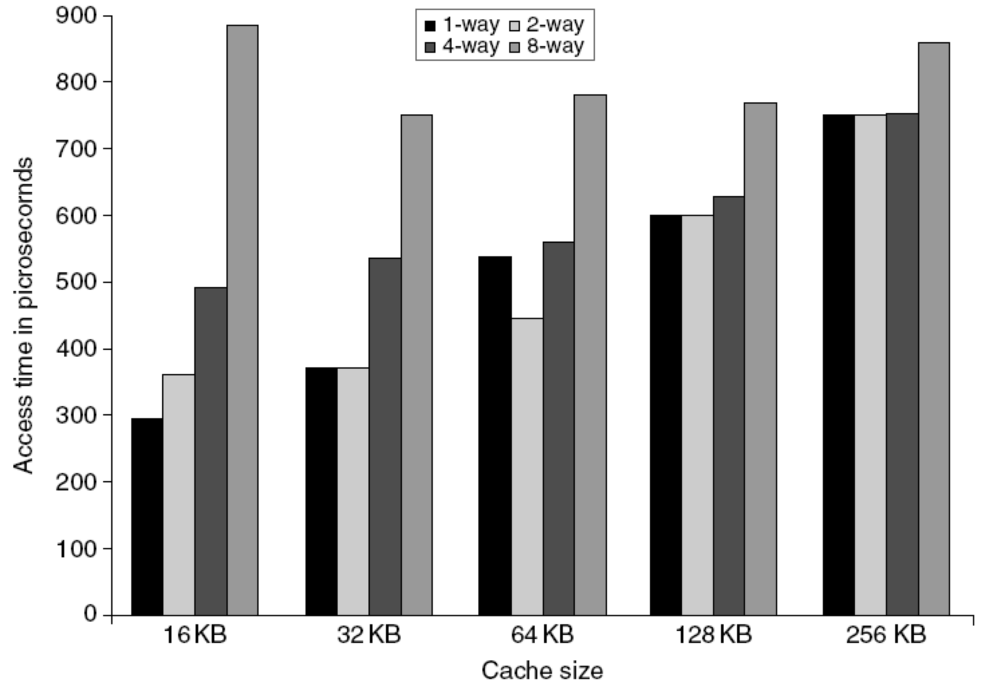
- **1. Increase Block Size**
  - ✅ Reduces **compulsory misses**, ⚠️ May increase **capacity & conflict misses**, and **miss penalty**
  - **Block Size** → **Bigger book bundles** (fewer trips, but heavier and slower).
- **2. Increase Total Cache Capacity**
  - ✅ Lowers **miss rate**, ⚠️ Raises **hit time** and **power consumption**
  - **Cache Capacity** → **Bigger shelf** (more books, but harder to search and costly).
- **3. Increase Associativity**
  - ✅ Reduces **conflict misses**, ⚠️ Increases **hit time** and **power consumption**
  - **Associativity** → **Flexible slots** (reduces conflicts, but adds complexity and power cost).

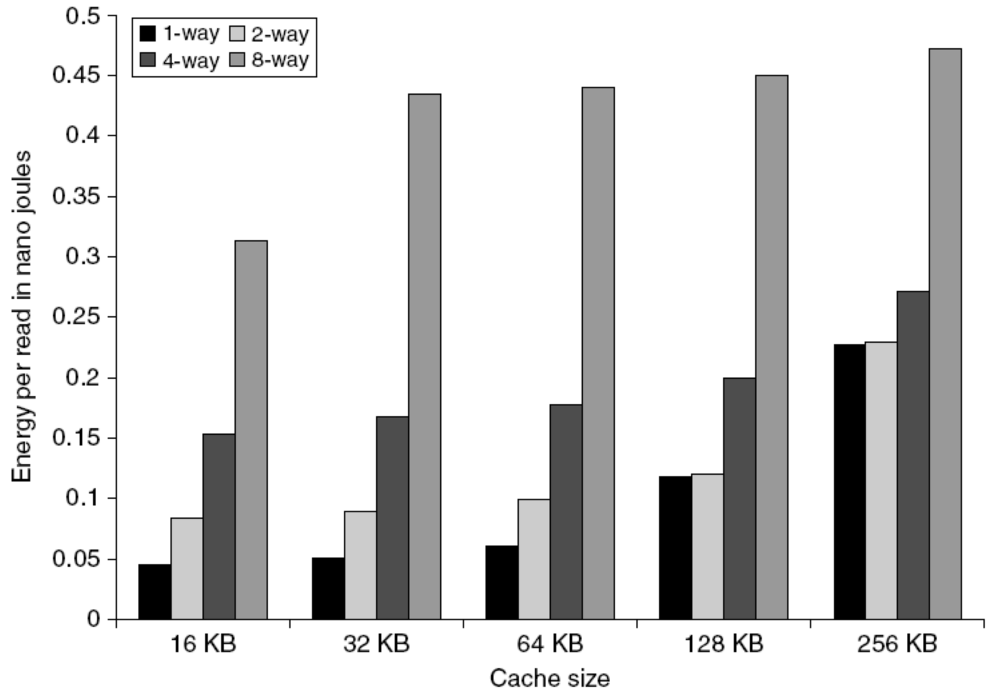
Cache Optimizations Basics

Objective: Minimize the miss rate and miss penalty

- **4. Add More Cache Levels**
  - ✅ Reduces **overall memory access time**
- **5. Prioritize Read Misses Over Writes**
  - ✅ Lowers **miss penalty**
- **6. Avoid Address Translation in Cache Indexing**
  - ✅ Reduces **hit time**

L1 Size and Associativity





Access time vs. size and associativity

Energy per read vs. size and associativity

Optimization 4: Multi-Level Caches
Focus: Miss Penalty

4. Add More Cache Levels → Create a Chain of Shelves

- ✅ **Reduces overall memory access time**
- **Analogy:**
  - Imagine you have a small shelf on your desk (L1),
  - a bigger shelf in your room (L2),
  - and a large cabinet in the hallway (L3).
  - Instead of running all the way to the library (main memory), you check the closest shelf first, then the next, and so on.
  - **Result:** Faster access because you rarely need to go to the library.

What Should be Our Focus?

- **Reduce Miss Penalty OR Reduce Miss Rate**

- **Memory Wall Problem:**
  Processors are sprinting ahead, but memory is lagging behind.

Question:

- Should we **make the cache faster** (keep pace with the runner)?
- Or **make the cache bigger** (build more rest stops along the track)?

Inclusive Cache → “Everything in L1 is also in L2”

- **Analogy:**
  Think of L1 as your desk shelf and L2 as a room bookshelf.
  Every book on your desk is also in the room shelf.
- **Why desirable?**
  Easy to check consistency—just look at the bigger shelf (L2).
- **Block size issue:**
  If desk shelf uses small books (64B) and room shelf uses big books (128B), removing one big book means clearing multiple small ones → messy!
- **Solution:**
  Keep block sizes the same for both shelves.

My Chip space can Accomodate only a SMALL L2 cache

- **Exclusive Cache → "No duplicates between shelves"**
- **Analogy:**
  - Desk shelf and room shelf never hold the same book. If you miss on the desk shelf, swap a book from the room shelf instead of adding a duplicate.
- **Benefit:**
  - Saves space on the big shelf.
- **Example:**
  - AMD Opteron uses two 64 KiB L1 shelves and a 1 MiB L2 shelf with this policy.

Optimization 5: Read Misses prioritized over Writes

Focus: Miss Penalty

5. Prioritize Read Misses Over Writes → Serve Readers First

- ✅ **Lowers miss penalty**
- **Analogy:**
  - Think of a bookstore where customers are waiting. People who want to **read now** (read misses) get priority over those who want to **return books or make changes** (writes).
  - Why? Reading is urgent for progress, while writing can wait a little without hurting performance.

5. Prioritize Read Misses Over Writes → Serve Readers First

- ✅ **Lowers miss penalty**
- **Analogy:**
  - Think of a bookstore where customers are waiting. People who want to **read now** (read misses) get priority over those who want to **return books or make changes** (writes).
  - Why? Reading is urgent for progress, while writing can wait a little without hurting performance.

5. Prioritize Read Misses Over Writes

- Serve reads before writes have been completed.
  - **Analogy:**
    - Imagine a busy café: customers who want to **read the menu and order** (read requests) are served first, even if the chef is still finishing previous dishes (writes). This keeps the line moving.
- Write buffer complexity:
  - **Question:** How big should the buffer be in a write-through cache?
  - **Purpose:** Hold updated values that might be needed during a read miss

5. Prioritize Read Misses Over Writes

- **Two Approaches**
  - **Wait for the buffer to empty**
    - Like waiting for the chef to finish all pending dishes before taking a new order—slow and inefficient.
    - Read miss waits until the write buffer is empty
  - **Check the buffer and proceed if safe**
    - If no conflicts and the kitchen is ready, take the new order immediately.
    - Check the contents of the write buffer on a read miss, and if there are no conflicts and the memory system is available, let the read miss continue
    - ✅ **All modern systems use this approach** for speed.

Optimization 6: Avoid Address Translation during Indexing of the Cache to Reduce Hit Time.

Focus: Hit Time

Address Translation During Cache Indexing

- Refers to the step where the processor converts a **virtual address** (used by programs) into a **physical address** (used by memory hardware) before using it to index into the cache.
- **Why does this matter?**
  - Most modern systems use **virtual memory**, so the CPU generates virtual addresses.
  - The cache typically works with **physical addresses** for correctness (especially in multiprogramming).
  - If you wait for the **translation** (via the TLB – Translation Lookaside Buffer) before indexing the cache, it adds extra delay to the **hit time**.
    - Virtual address → TLB → Physical address → Cache indexing (slow path)
    - Optimized path: Virtual index first, physical tag later?

Address Translation During Cache Indexing

- **Analogy:**
  - Imagine you have a **nickname** (virtual address) and a **legal name** (physical address).
  - If the librarian insists on checking your legal name before letting you grab a book, you waste time.
- **Optimization:**
  - Use the nickname directly to find the shelf (index the cache) while verifying later if it matches the legal name (tag check).
  - This technique is called **Virtually Indexed, Physically Tagged (VIPT)** caching:
    - **Index with virtual address** (fast).
    - **Tag compare with physical address** (ensures correctness).

Topics Covered…… UG Syllabus

- Four questions to understand overall working of the cache:
  1. Where can a block be placed? **Block Placement**
     - Cache organization and Design
     - Cache Addressing , sets, ways
  2. How is a block found? **Block Identification**
     - Measure Cache Performance, Average Memory Access Time (AMAT)
  3. Which block to replace on a miss? **Replacement Policies**
     - Depends on Cache Organization [Direct/Set Associative/Fully Associative]
     - FIFO, L**R**U (Recently), Random, L**F**U (Frequently)
  4. What happens on a write? **Write Back Policies**
     - Write back/ Write Through
     - Write Allocation

Topics Covered…… PG Syllabus

- Different types of cache misses and how to measure them
- Metrics in Cache Performance
  - AMAT = Hit time + Miss rate × Miss penalty
- Cache optimizations
  - **Miss Rate:** Increase Block Size/Total Cache Capacity/Associativity
  - **Miss Penalty:** More Cache Levels, Prioritize Read Misses Over Writes
  - **Hit Time:** Avoid Address Translation in Cache Indexing

End of Cache Topic/Syllabus