

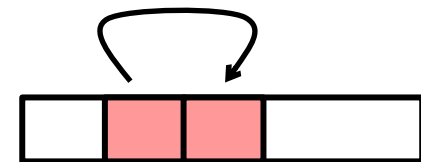
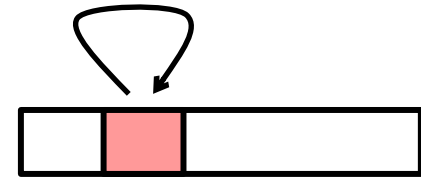
# Cache Memories

Troels Henriksen  
**HPPS**

**Based on slides by:**  
Randal E. Bryant and David R. O'Hallaron

# Locality

- **Principle of Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently
- **Temporal locality:**
  - Recently referenced items are likely to be referenced again in the near future
- **Spatial locality:**
  - Items with nearby addresses tend to be referenced close together in time



# Locality Example

```
sum = 0;  
for (i = 0; i < n; i++)  
    sum += a[i];  
return sum;
```

## ■ Data references

- Reference array elements in succession (stride-1 reference pattern).
- Reference variable `sum` each iteration.

**Spatial locality**

**Temporal locality**

## ■ Instruction references

- Reference instructions in sequence.
- Cycle through loop repeatedly.

**Spatial locality**

**Temporal locality**

# Qualitative Estimates of Locality

- **Claim:** Being able to look at code and get a qualitative sense of its locality is a key skill for a competent programmer.
- **Question:** Does this function have good locality with respect to array *a*? (The array is in row-major order.)

```
int sum_array_rows(int M, int N, int *a)
{
    int sum = 0;

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

# Locality Example

- **Question:** Does this function have good locality with respect to array `a`?

```
int sum_array_cols(int M, int N, int *a)
{
    int sum = 0;

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

# Locality Example

- **Question:** Can you permute the loops so that the function scans the 3-d array `a` with a stride-1 reference pattern (and thus has good spatial locality)?

```
int sum_array_3d(int M, int N, int *a)
{
    int sum = 0;

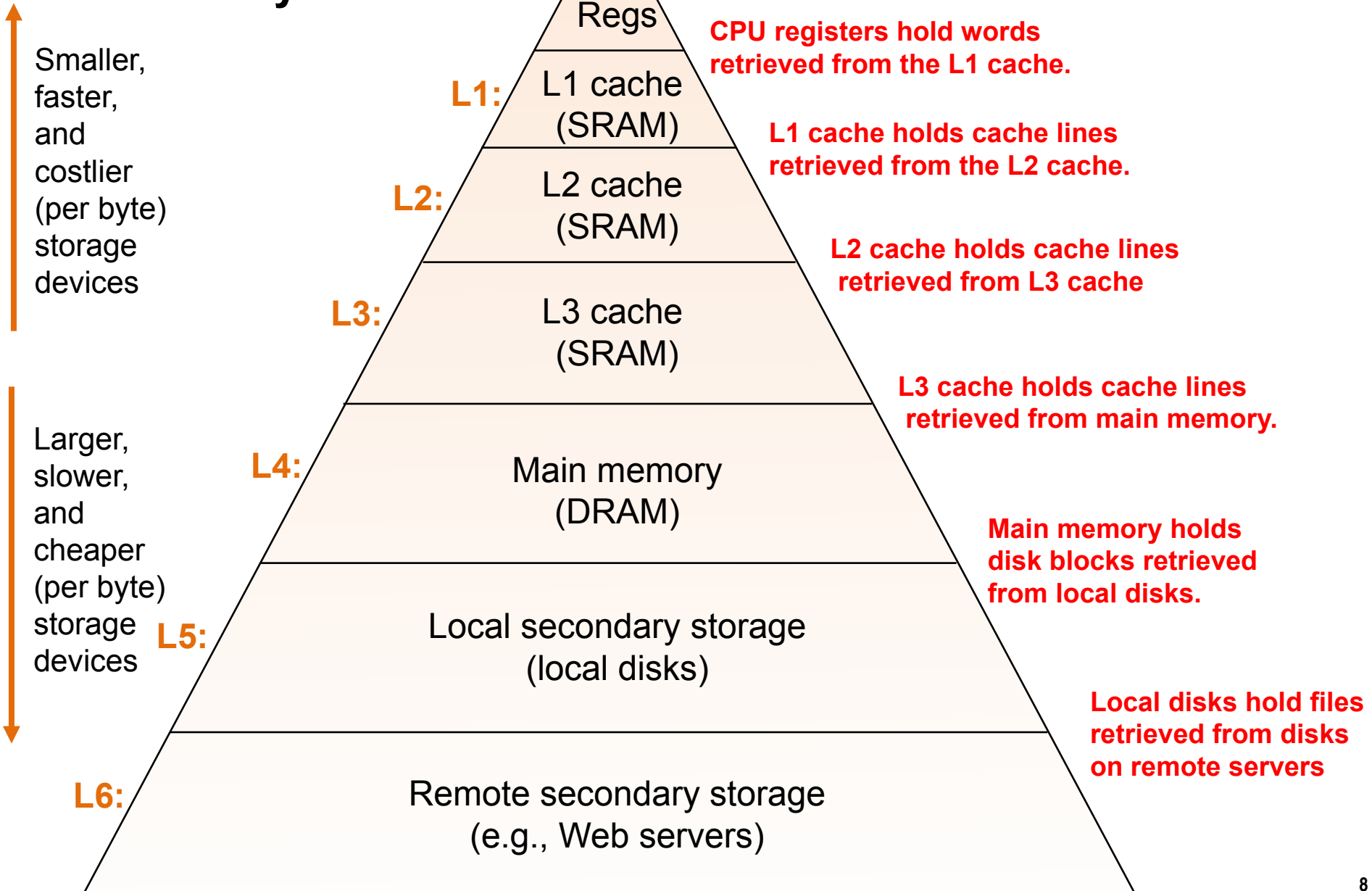
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
            for (int k = 0; k < N; k++)
                sum += a[k*(N*N)+i*N+j];

    return sum;
}
```

# Memory Hierarchies

- **Some fundamental and enduring properties of hardware and software:**
  - Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
  - The gap between CPU and main memory speed is widening.
  - Well-written programs tend to exhibit good locality.
- **These fundamental properties complement each other beautifully.**
- **They suggest an approach for organizing memory and storage systems known as a **memory hierarchy**.**

# Example Memory Hierarchy

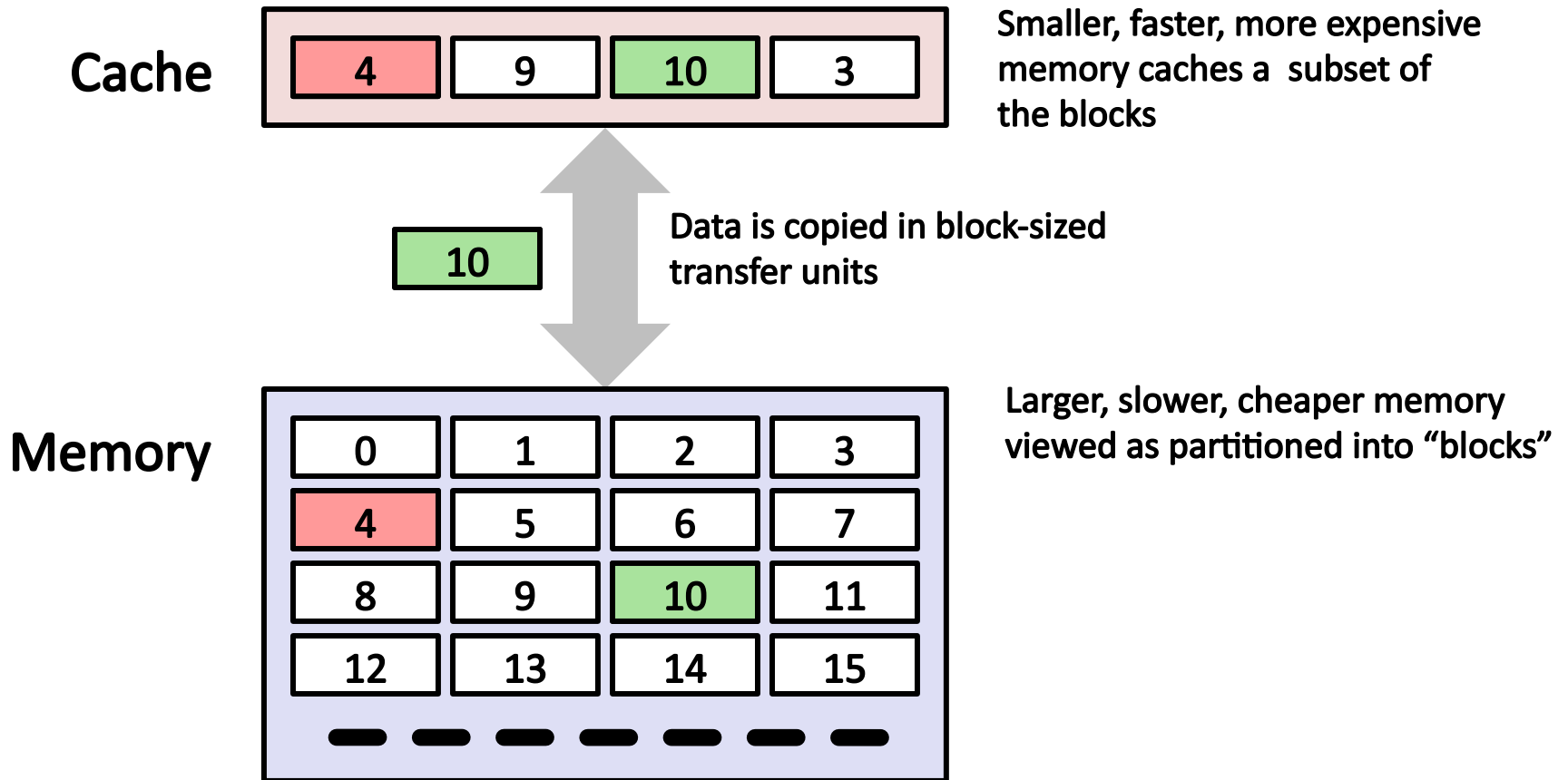




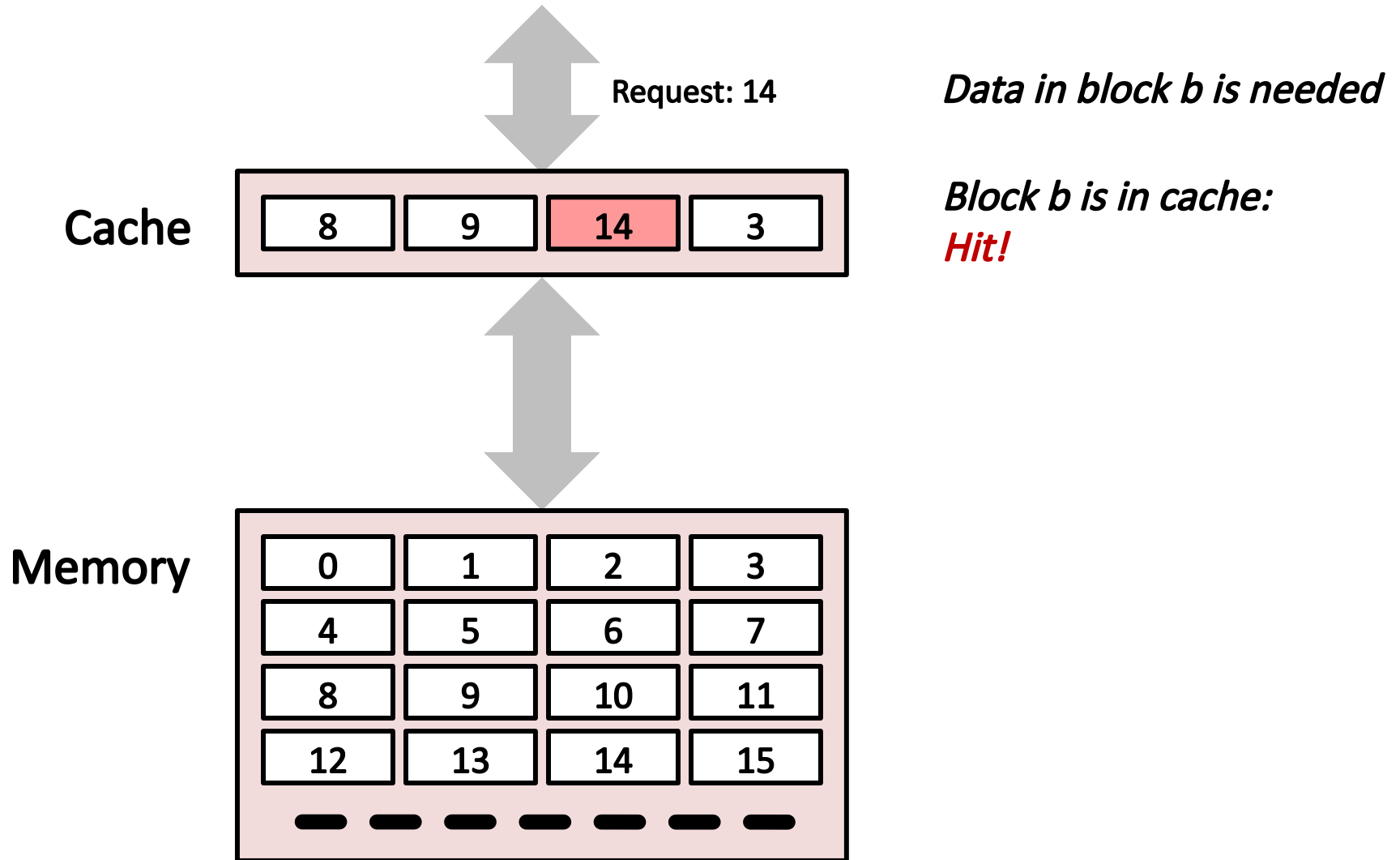
# Caches

- **Cache:** A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- **Fundamental idea of a memory hierarchy:**
  - For each  $k$ , the faster, smaller device at level  $k$  serves as a cache for the larger, slower device at level  $k+1$ .
- **Why do memory hierarchies work?**
  - Because of locality, programs tend to access the data at level  $k$  more often than they access the data at level  $k+1$ .
  - Thus, the storage at level  $k+1$  can be slower, and thus larger and cheaper per bit.
- **Big Idea:** The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

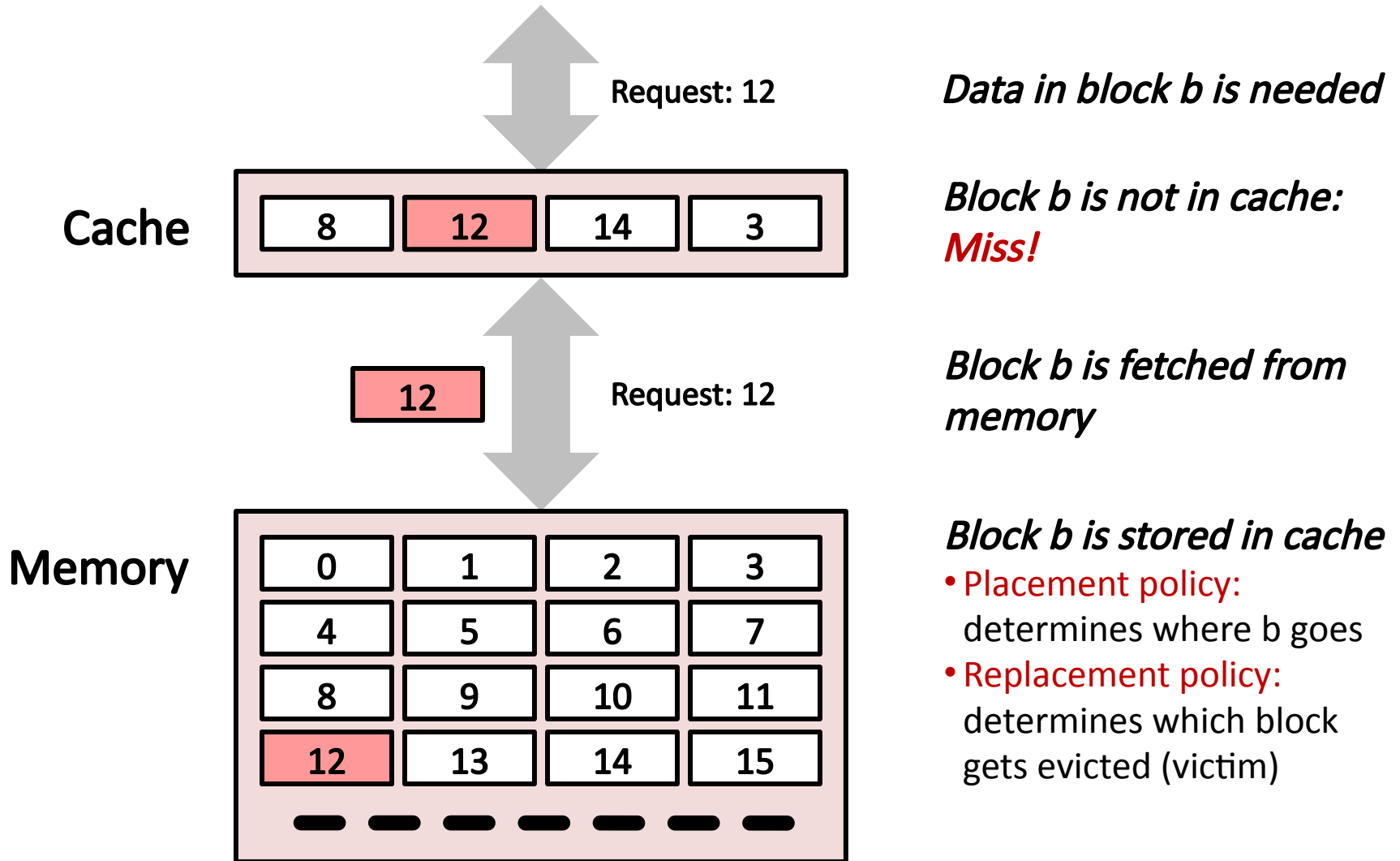
# General Cache Concepts



# General Cache Concepts: Hit



# General Cache Concepts: Miss



# General Caching Concepts:

## Types of Cache Misses

### ■ Cold (compulsory) miss

- Cold misses occur because the cache is empty.

### ■ Conflict miss

- Most caches limit blocks at level  $k+1$  to a small subset (sometimes a singleton) of the block positions at level  $k$ .
  - E.g. Block  $i$  at level  $k+1$  must be placed in block  $(i \bmod 4)$  at level  $k$ .
- Conflict misses occur when the level  $k$  cache is large enough, but multiple data objects all map to the same level  $k$  block.
  - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.

### ■ Capacity miss

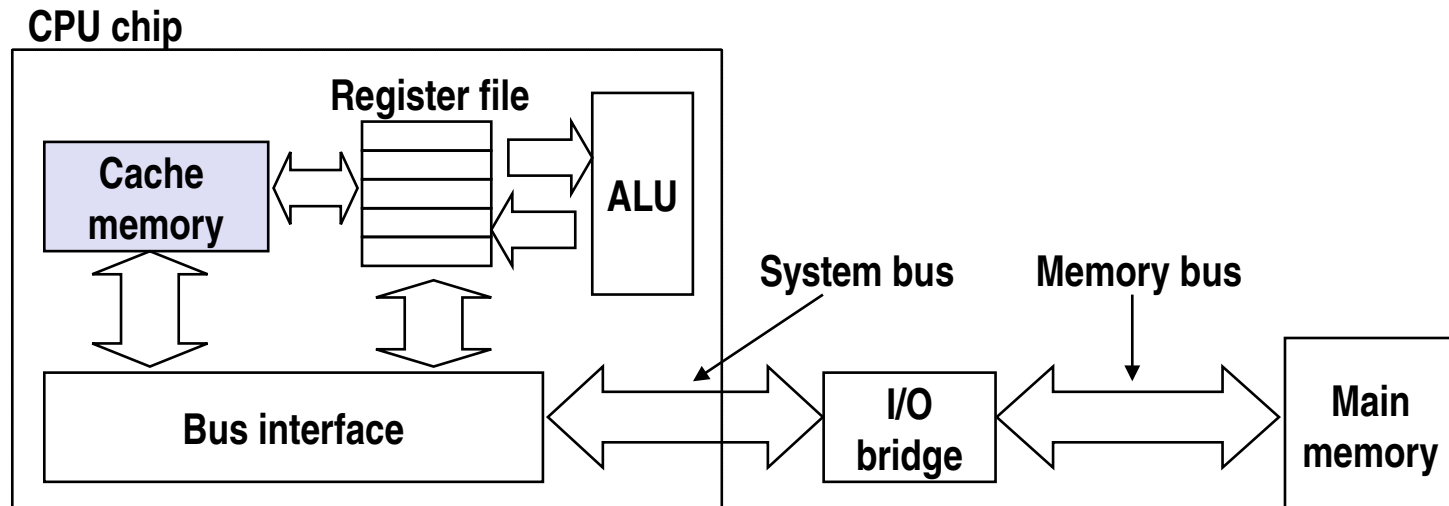
- Occurs when the set of active cache blocks (**working set**) is larger than the cache.

# Examples of Caching in the Mem. Hierarchy

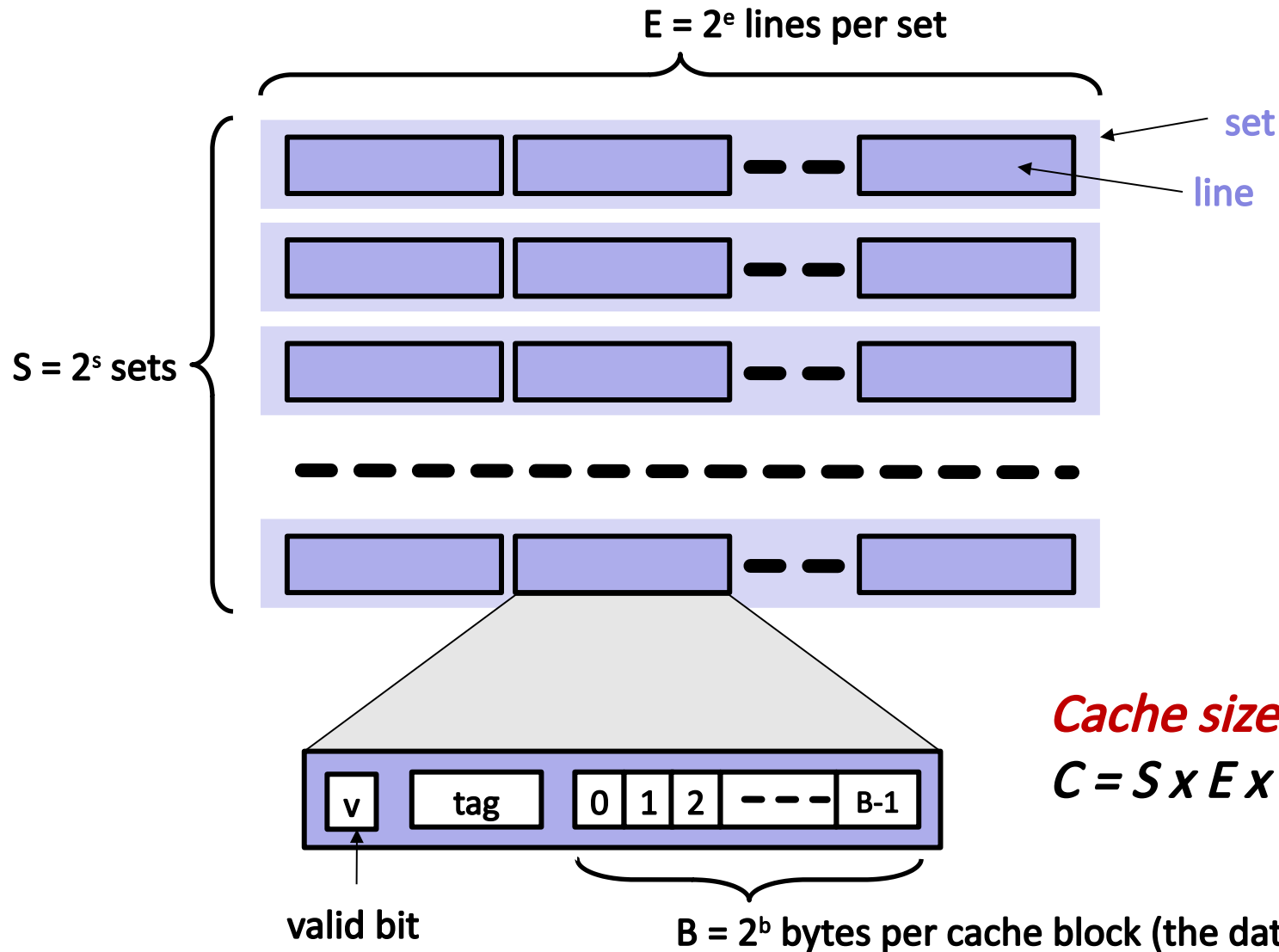
| Cache Type           | What is Cached?      | Where is it Cached? | Latency (cycles) | Managed By       |
|----------------------|----------------------|---------------------|------------------|------------------|
| Registers            | 4-8 bytes words      | CPU core            | 0                | Compiler         |
| TLB                  | Address translations | On-Chip TLB         | 0                | Hardware MMU     |
| L1 cache             | 64-byte blocks       | On-Chip L1          | 4                | Hardware         |
| L2 cache             | 64-byte blocks       | On-Chip L2          | 10               | Hardware         |
| Virtual Memory       | 4-KB pages           | Main memory         | 100              | Hardware + OS    |
| Buffer cache         | Parts of files       | Main memory         | 100              | OS               |
| Disk cache           | Disk sectors         | Disk controller     | 100,000          | Disk firmware    |
| Network buffer cache | Parts of files       | Local disk          | 10,000,000       | NFS client       |
| Browser cache        | Web pages            | Local disk          | 10,000,000       | Web browser      |
| Web cache            | Web pages            | Remote server disks | 1,000,000,000    | Web proxy server |

# Cache Memories

- **Cache memories** are small, fast SRAM-based memories managed automatically in hardware
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in cache
- Typical system structure:



# General Cache Organization (S, E, B)

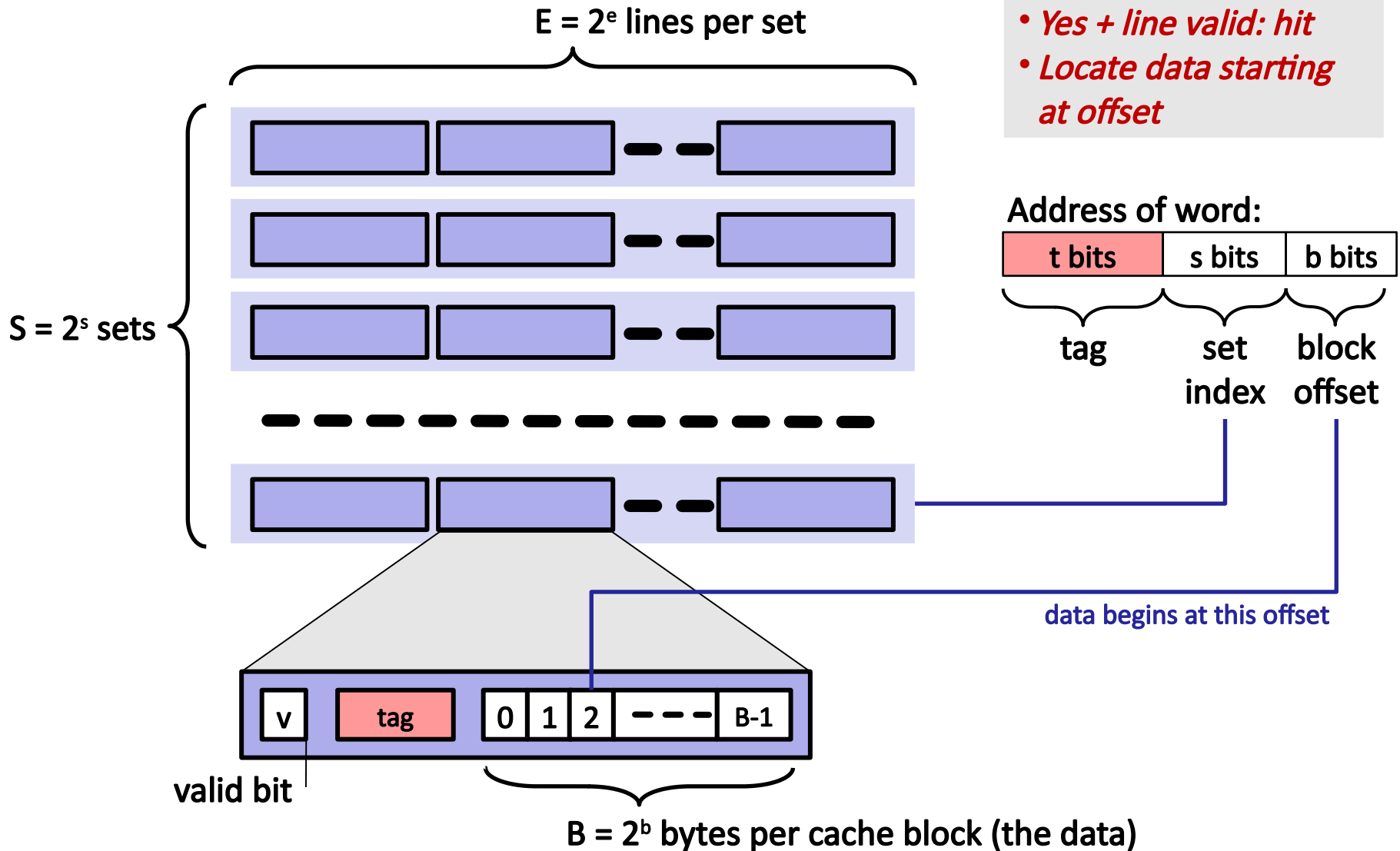


**Cache size:**

$$C = S \times E \times B \text{ data bytes}$$



# Cache Read

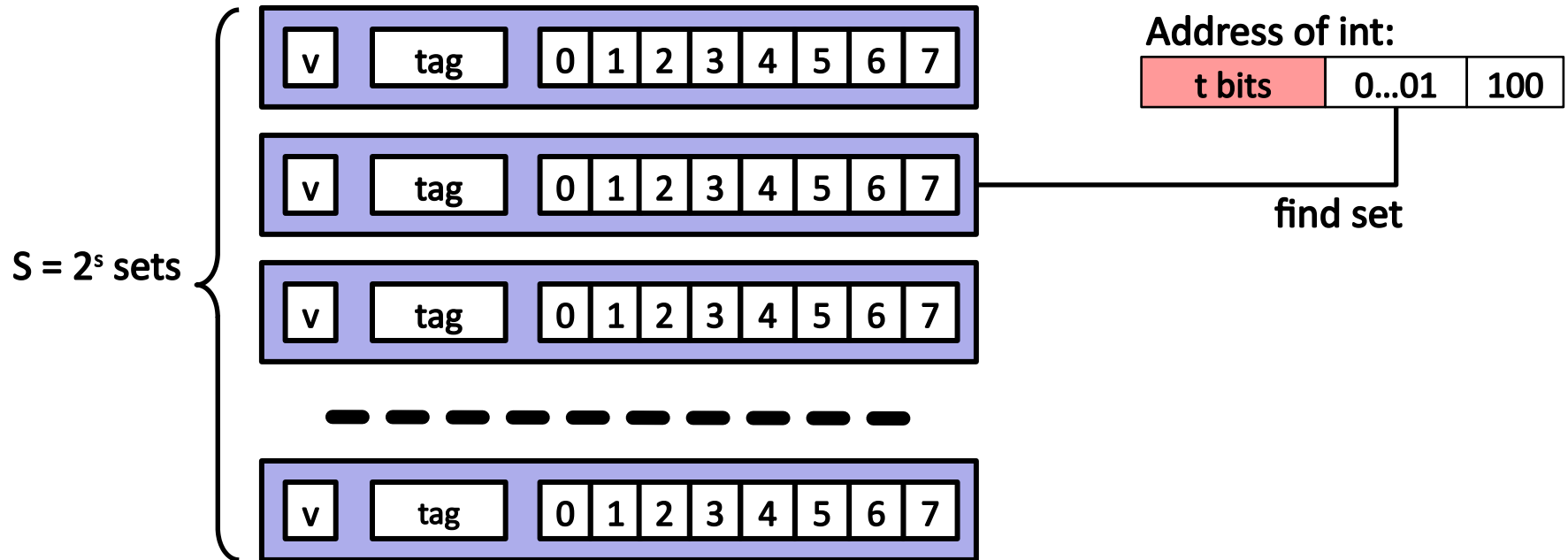


- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*

# Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set

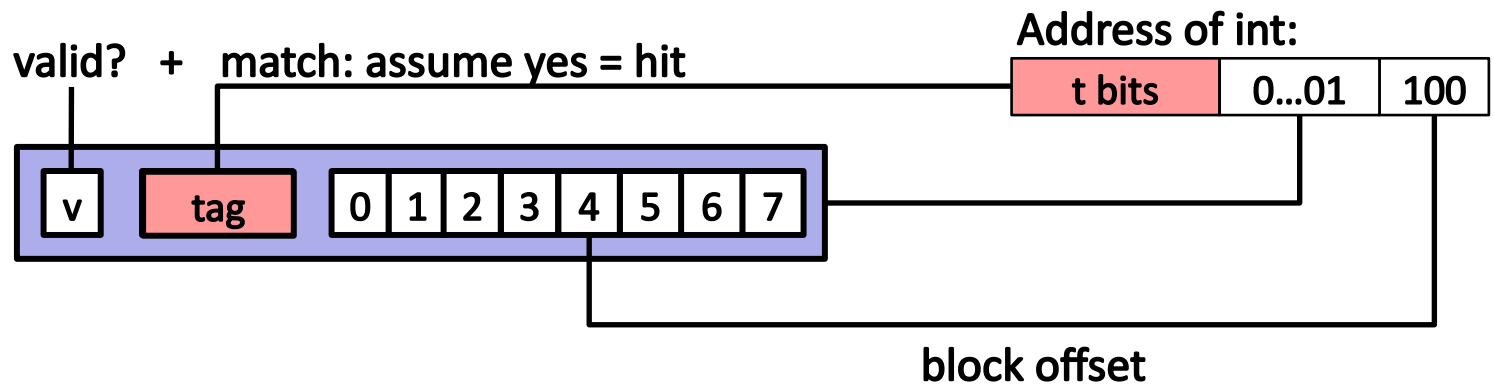
Assume: cache block size 8 bytes



# Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set

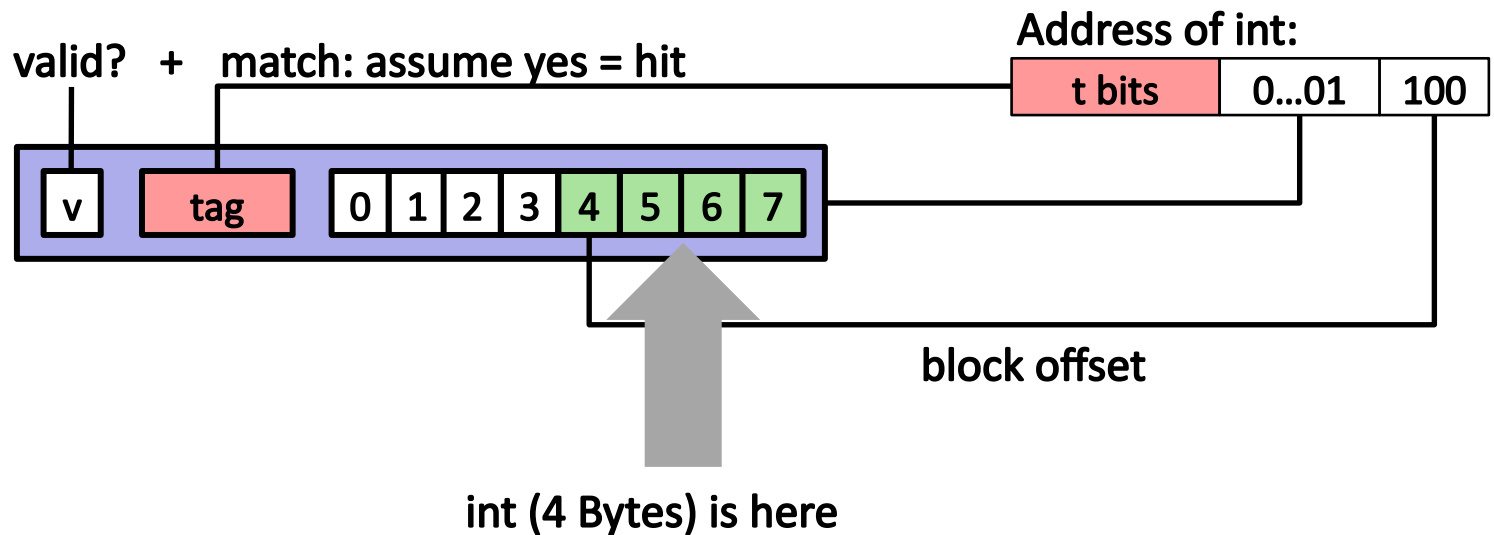
Assume: cache block size 8 bytes



# Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set

Assume: cache block size 8 bytes



**If tag doesn't match:** old line is evicted and replaced

# Direct-Mapped Cache Simulation

|     |     |     |
|-----|-----|-----|
| t=1 | s=2 | b=1 |
| x   | xx  | x   |

M=16 bytes (4-bit addresses), B=2 bytes/block,  
S=4 sets, E=1 Blocks/set

Address trace (reads, one byte per read):

0 [0000<sub>2</sub>], miss

1 [0001<sub>2</sub>], hit

7 [0111<sub>2</sub>], miss

8 [1000<sub>2</sub>], miss

0 [0000<sub>2</sub>] miss

|       | v | Tag | Block  |
|-------|---|-----|--------|
| Set 0 | 1 | 0   | M[0-1] |
| Set 1 |   |     |        |
| Set 2 |   |     |        |
| Set 3 | 1 | 0   | M[6-7] |

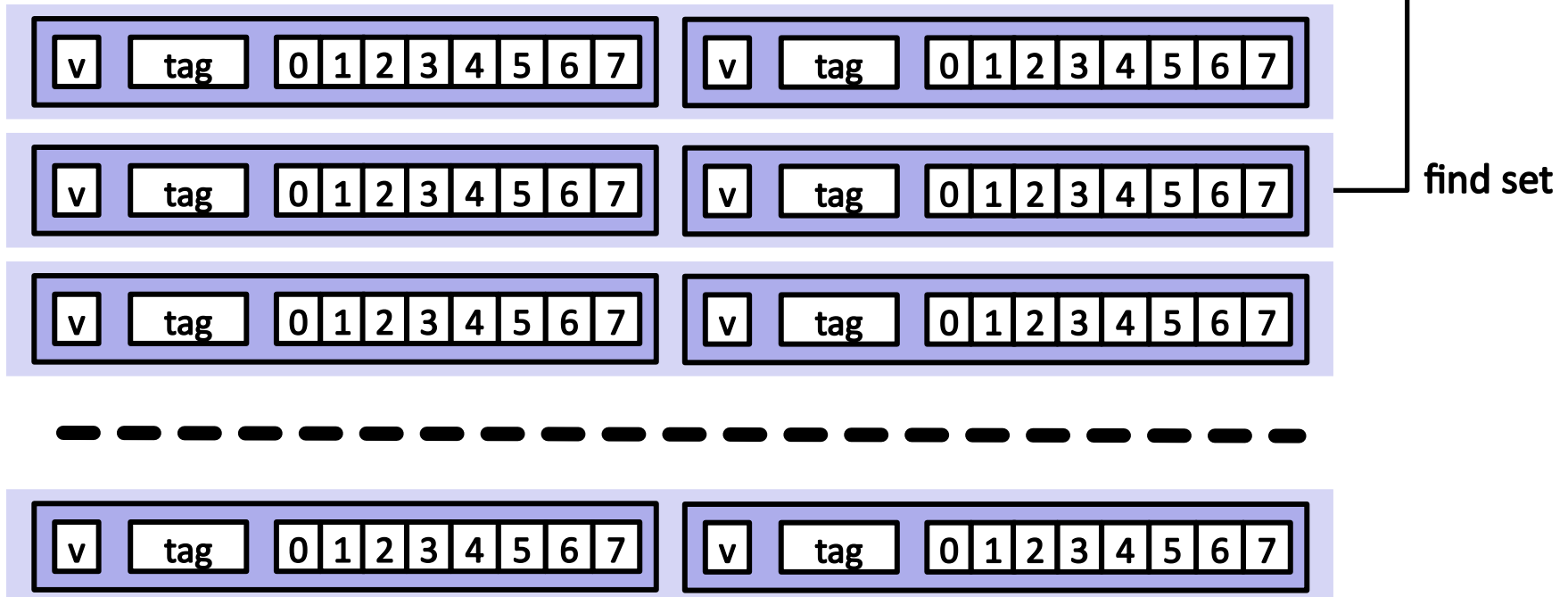
# E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache block size 8 bytes

Address of short int:

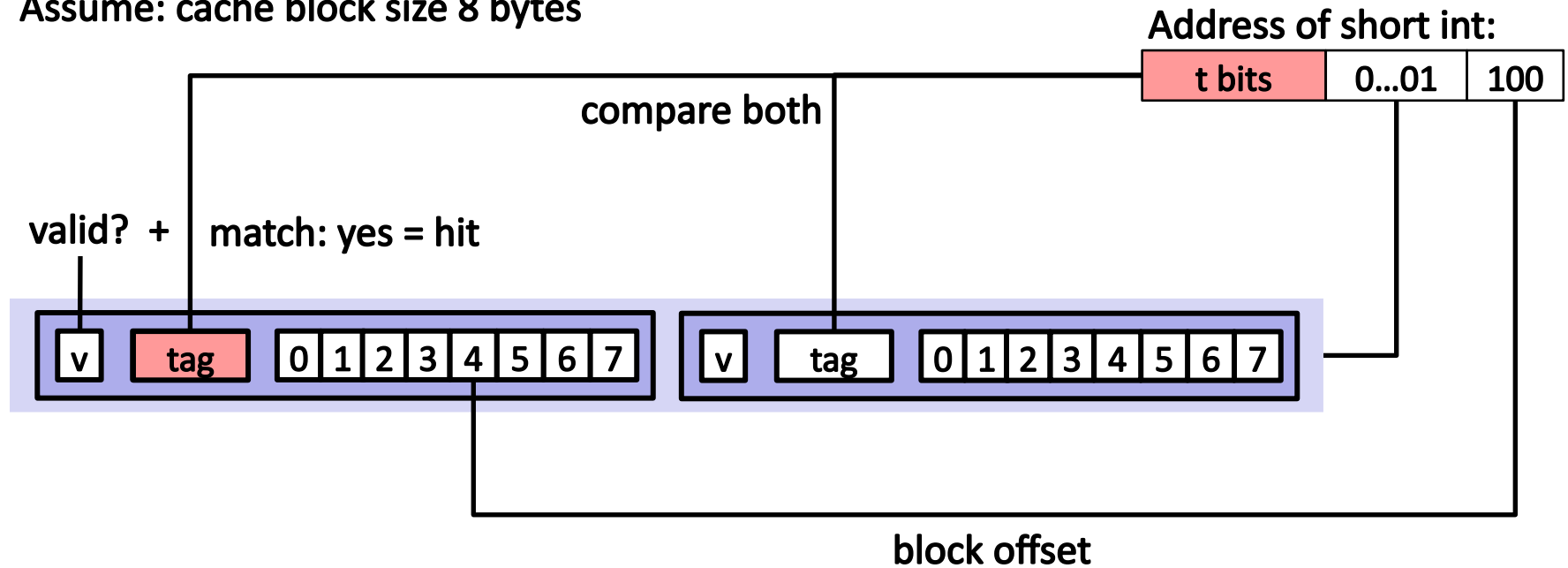
|        |        |     |
|--------|--------|-----|
| t bits | 0...01 | 100 |
|--------|--------|-----|



# E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

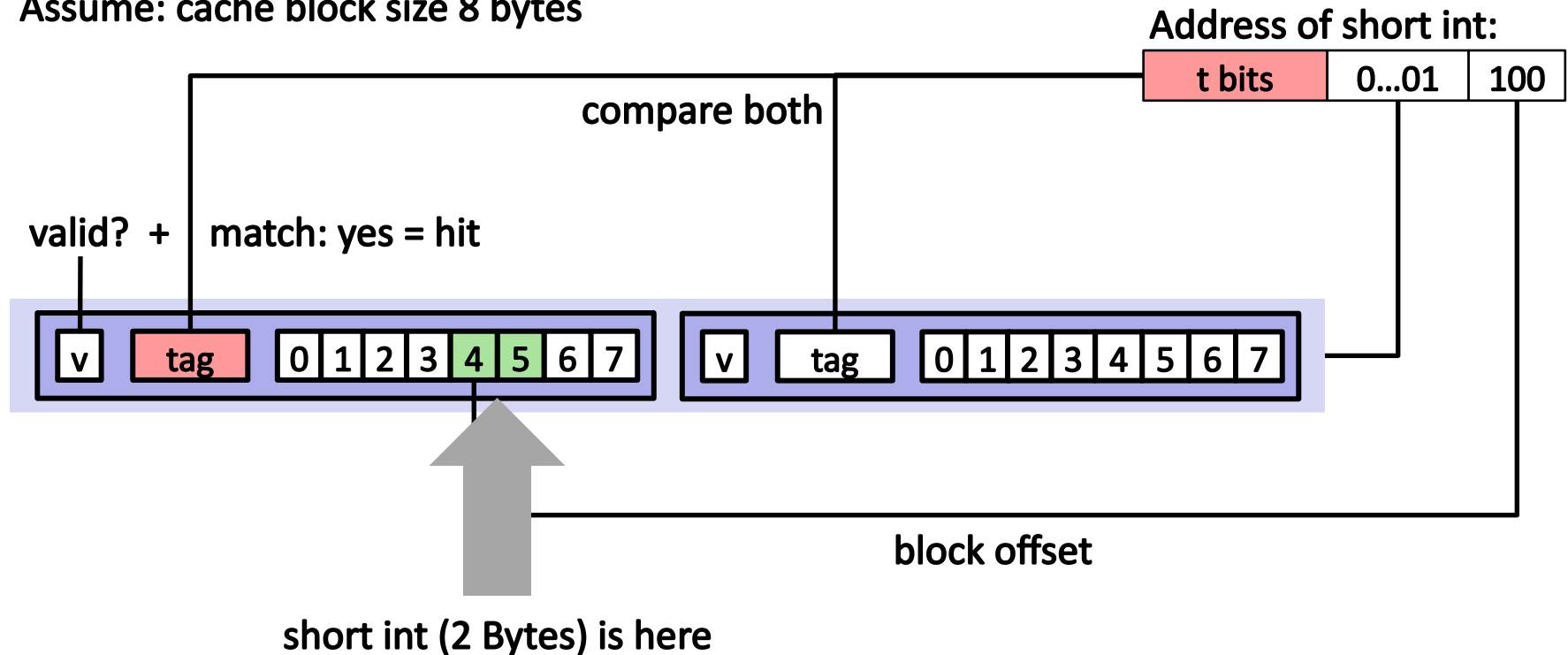
Assume: cache block size 8 bytes



# E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache block size 8 bytes



## No match:

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), ...



# 2-Way Set Associative Cache Simulation

|     |     |     |
|-----|-----|-----|
| t=2 | s=1 | b=1 |
| xx  | x   | x   |

M=16 byte addresses, B=2 bytes/block,  
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0 [0000<sub>2</sub>], miss  
 1 [0011<sub>2</sub>], hit  
 7 [0111<sub>2</sub>], miss  
 8 [1000<sub>2</sub>], miss  
 0 [0000<sub>2</sub>] hit

|       | v | Tag | Block  |
|-------|---|-----|--------|
| Set 0 | 1 | 00  | M[0-1] |
|       | 1 | 10  | M[8-9] |
| Set 1 | 1 | 01  | M[6-7] |
|       | 0 |     |        |

# What about writes?

## ■ Multiple copies of data exist:

- L1, L2, L3, Main Memory, Disk

## ■ What to do on a write-hit?

- **Write-through** (write immediately to memory)
- **Write-back** (defer write to memory until replacement of line)
  - Need a dirty bit (line different from memory or not)

## ■ What to do on a write-miss?

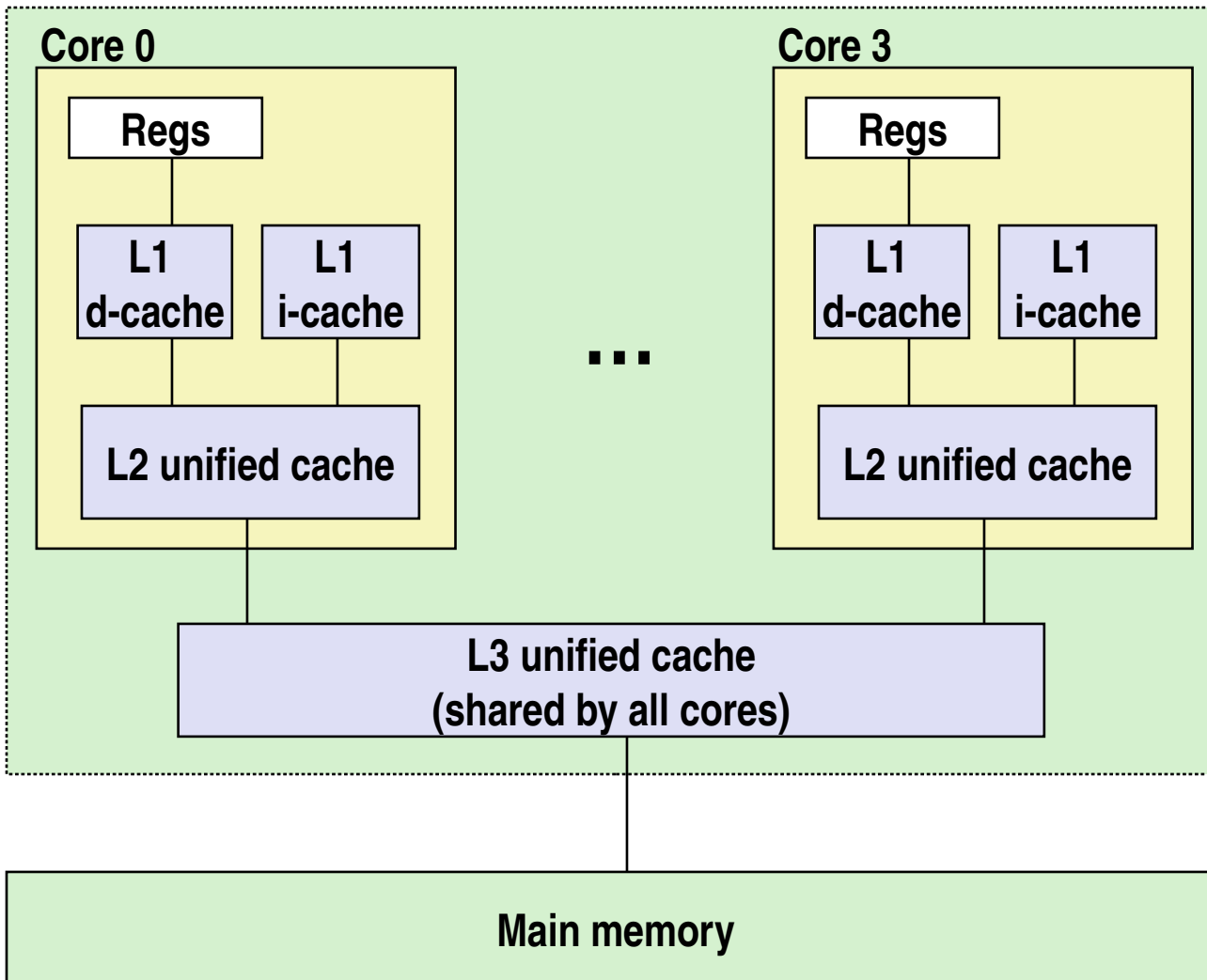
- **Write-allocate** (load into cache, update line in cache)
  - Good if more writes to the location follow
- **No-write-allocate** (writes straight to memory, does not load into cache)

## ■ Typical

- Write-through + No-write-allocate
- **Write-back + Write-allocate**

# Intel Core i7 Cache Hierarchy

## Processor package



**L1 i-cache and d-cache:**

32 KB, 8-way,  
Access: 4 cycles

**L2 unified cache:**

256 KB, 8-way,  
Access: 10 cycles

**L3 unified cache:**

8 MB, 16-way,  
Access: 40-75 cycles

**Block size:** 64 bytes  
for all caches.

# Cache Performance Metrics

## ■ Miss Rate

- Fraction of memory references not found in cache (misses / accesses)  
=  $1 - \text{hit rate}$
- Typical numbers (in percentages):
  - 3-10% for L1
  - can be quite small (e.g.,  $< 1\%$ ) for L2, depending on size, etc.

## ■ Hit Time

- Time to deliver a line in the cache to the processor
  - includes time to determine whether the line is in the cache
- Typical numbers:
  - 4 clock cycle for L1
  - 10 clock cycles for L2

## ■ Miss Penalty

- Additional time required because of a miss
  - typically 50-200 cycles for main memory (Trend: increasing!)

# Let's think about those numbers

- **Huge difference between a hit and a miss**
  - Could be 100x, if just L1 and main memory
- **Would you believe 99% hits is twice as good as 97%?**
  - Consider:
    - cache hit time of 1 cycle
    - miss penalty of 100 cycles
  - Average access time:
    - 97% hits:  $1 \text{ cycle} + 0.03 * 100 \text{ cycles} = 4 \text{ cycles}$
    - 99% hits:  $1 \text{ cycle} + 0.01 * 100 \text{ cycles} = 2 \text{ cycles}$
- **This is why “miss rate” is used instead of “hit rate”**

# Writing Cache Friendly Code

- **Make the common case go fast**
  - Focus on the inner loops of the core functions
- **Minimize the misses in the inner loops**
  - Repeated references to variables are good (**temporal locality**)
  - Stride-1 reference patterns are good (**spatial locality**)

**Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories**

# Cache Summary

- **Cache memories can have significant performance impact**
- **You can write your programs to exploit this!**
  - Focus on the inner loops, where bulk of computations and memory accesses occur.
  - Try to maximize spatial locality by reading data objects with sequentially with stride 1.
  - Try to maximize temporal locality by using a data object as often as possible once it's read from memory.
- **Focus of next video!**