

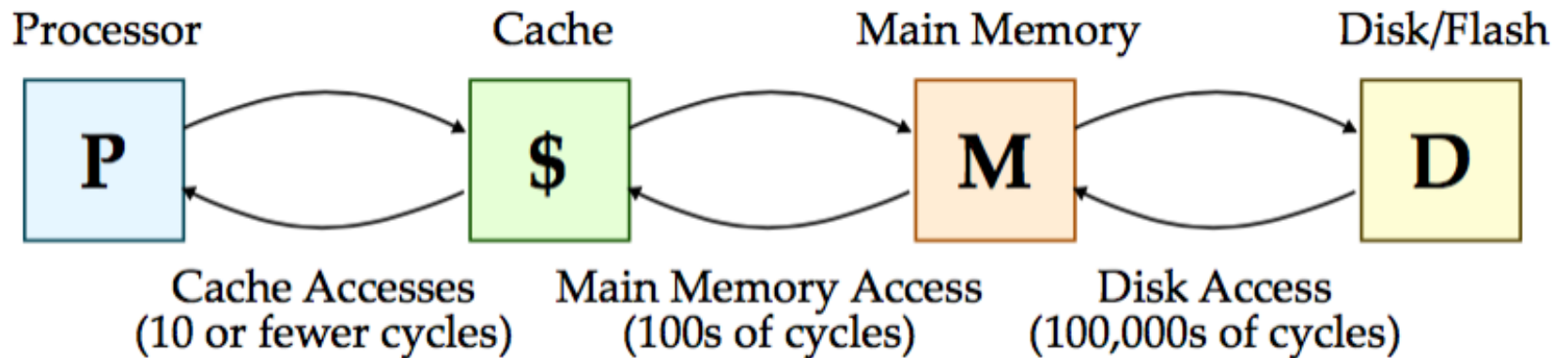
FU05 Computer Architecture

11. Memory Hierarchy: Cache (メモリ階層化: キャッシュ)

Ben Abdallah Abderazek

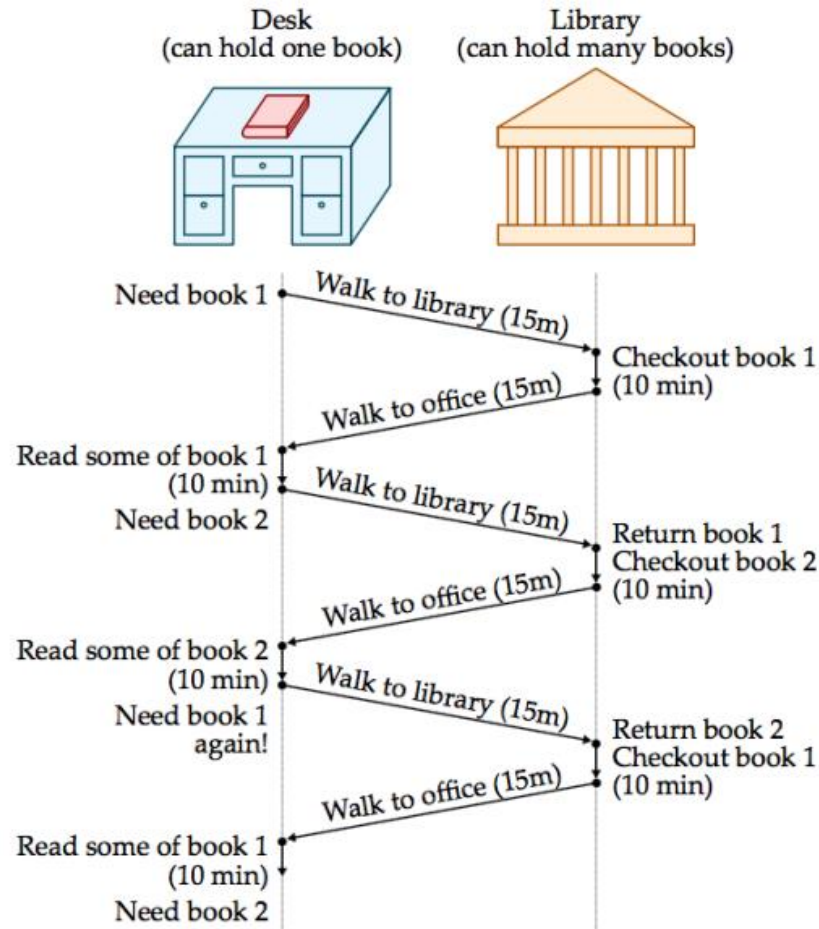
E-mail: benab@u-aizu.ac.jp

Cache Memories in Computer System



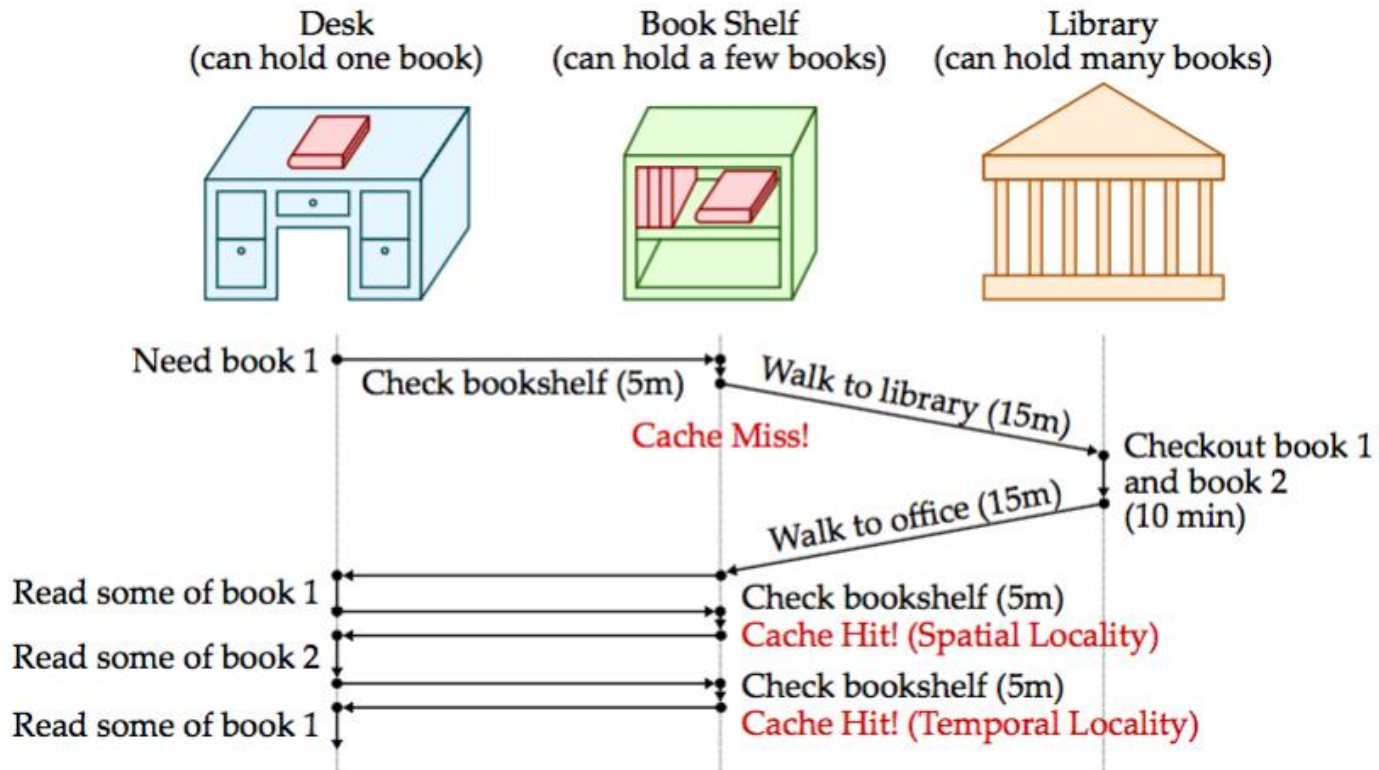
Scenario 1: Desk + Library, No Bookshelf

“Cache”



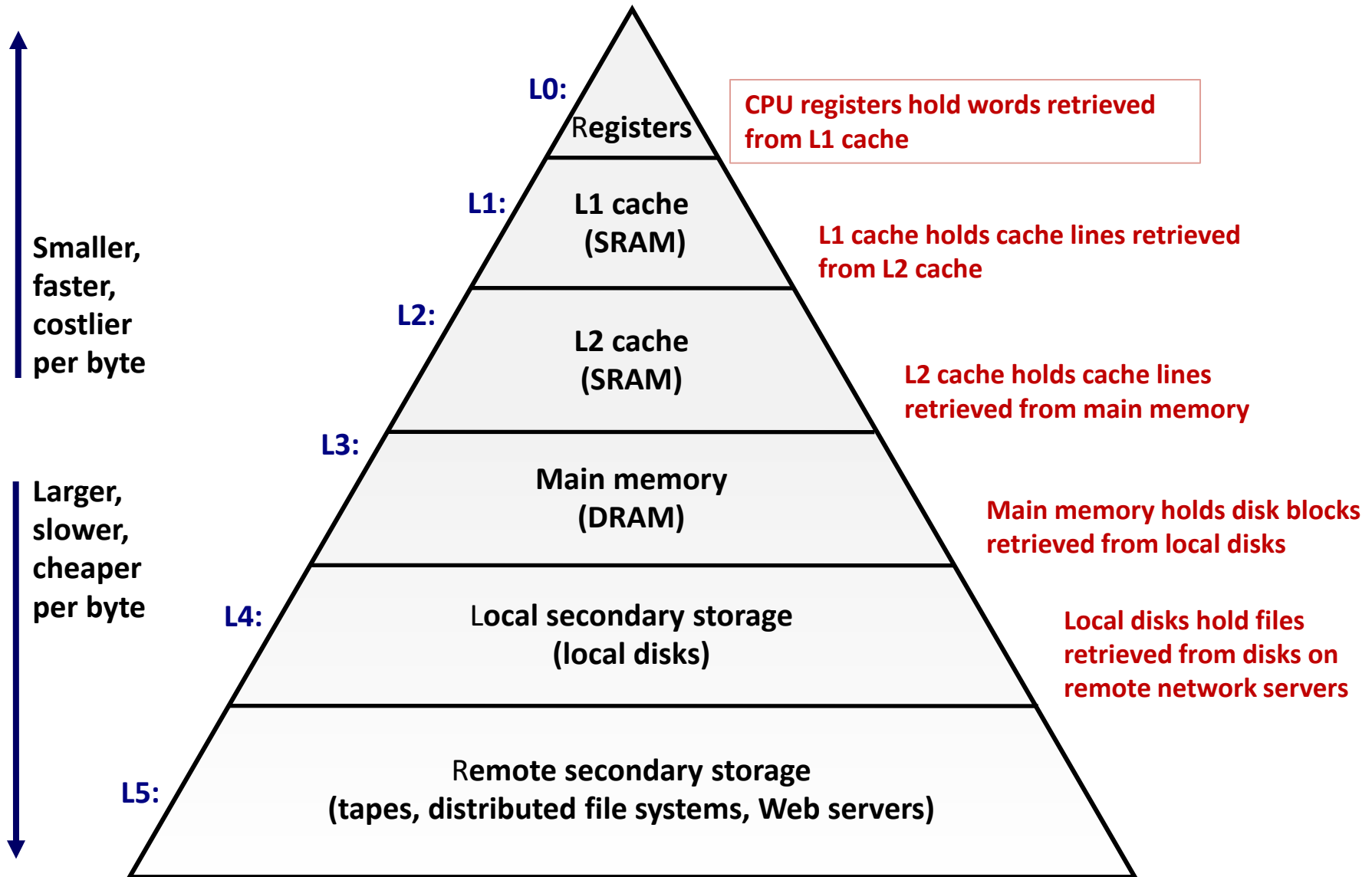
- Average latency: 40 minutes
- Average throughput: 1.2 books/hour

Scenario 2: Desk + Library with Bookshelf “Cache”

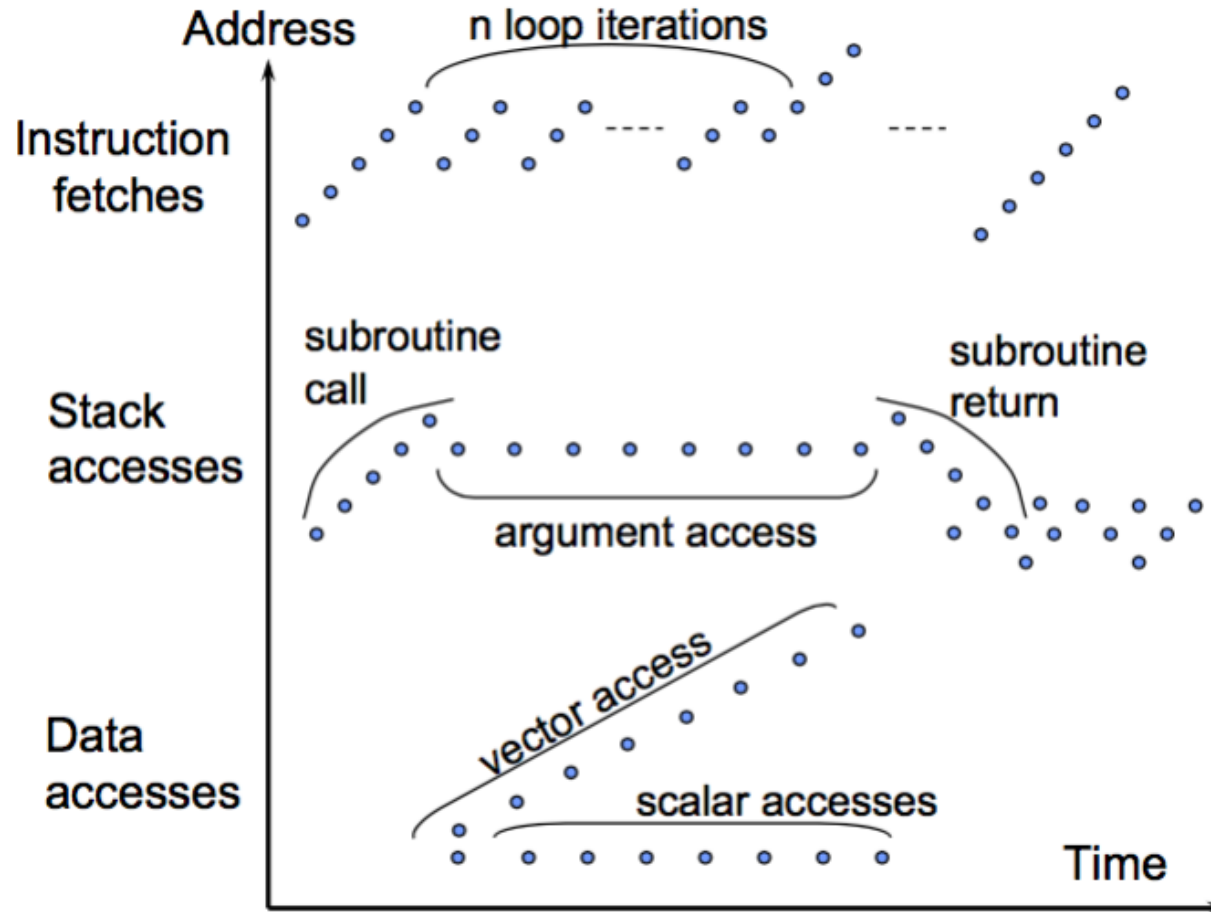


- Average Latency: < 20min
- Average Throughput: 2 books/hour

An Example of Memory Hierarchy



Typical Data Access Pattern instruction vs data access, **temporal** vs **spatial** locality



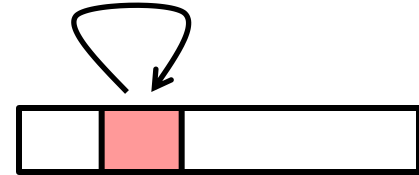
Principle of Locality

- Programs access a small proportion of their address space at any time
- Temporal locality
 - Items accessed recently are likely to be accessed again soon
 - e.g., instructions in a loop, induction variables
- Spatial locality
 - Items near those accessed recently are likely to be accessed soon
 - E.g., sequential instruction access, array data

Principle of Locality

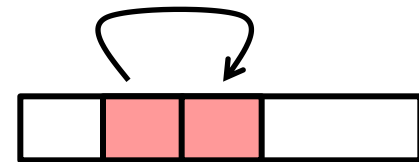
- Temporal locality:

- 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

Locality Example

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

```
int sum_array_cols(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;
}
```

Locality Example

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

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

    for (j = 0; j < N; j++) //outer loop over columns (poor locality )
        for (i = 0; i < M; i++) // inner loop over rows
            sum += a[i][j];
    return sum;
}
```

Array Memory Layout (Row-Major Order):

```
[ a[0][0] a[0][1] a[0][2] ... a[0][N-1] ]
[ a[1][0] a[1][1] a[1][2] ... a[1][N-1] ]
[ a[2][0] a[2][1] a[2][2] ... a[2][N-1] ]
...
[ a[M-1][0] a[M-1][1] a[M-1][2] ... a[M-1][N-1] ]
```

Traversal Pattern:

1. Start at column 0 (a[0][0], a[1][0], a[2][0], ...)
2. Move to column 1 (a[0][1], a[1][1], a[2][1], ...)
3. Continue similarly until column N-1 (a[0][N-1], a[1][N-1], a[2][N-1], ...)

This function does not have good locality with respect to the array **a**. The reason lies in the **memory access pattern**:

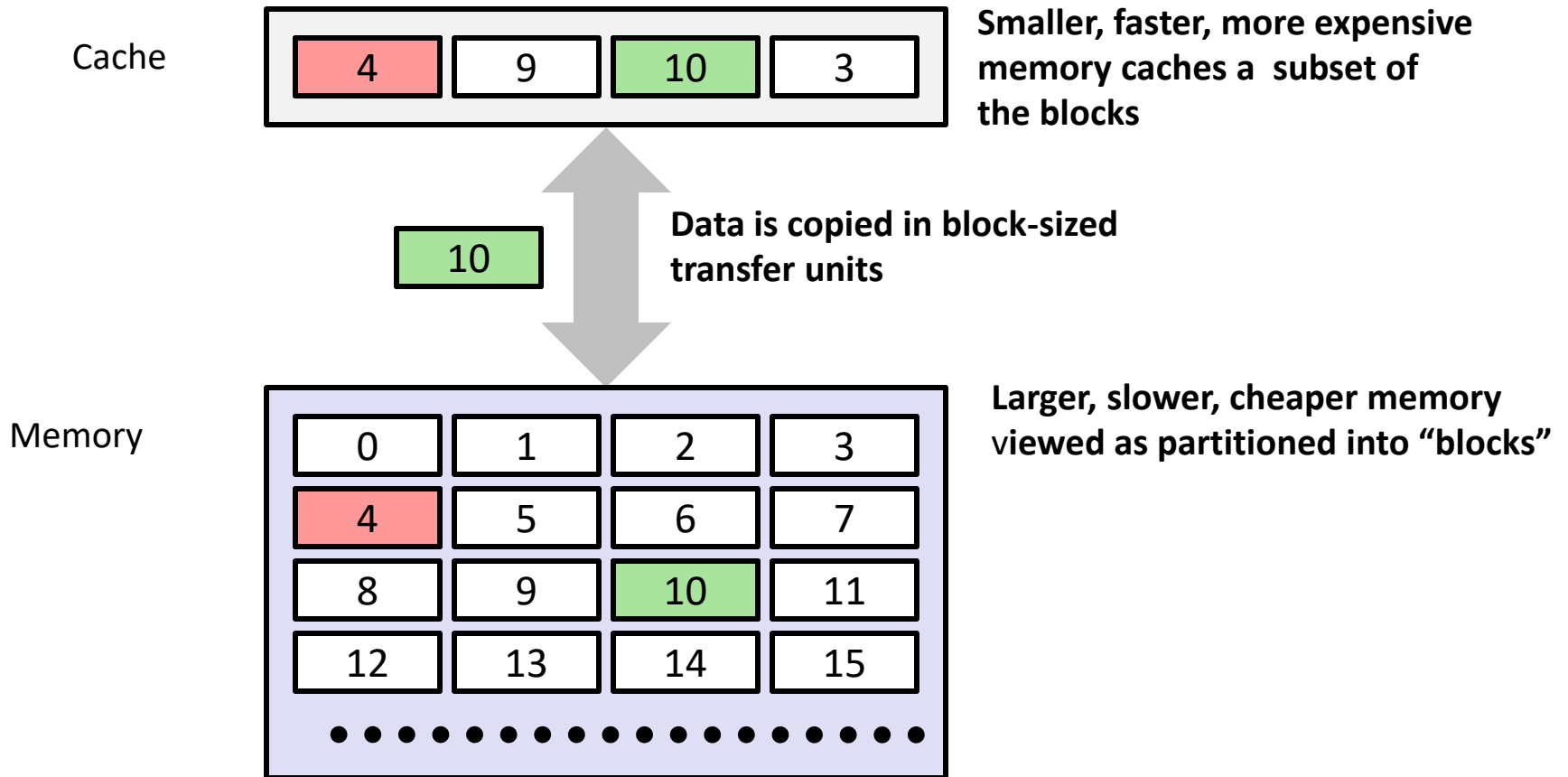
- In the inner loop, the function accesses **a[i][j]** where **i** increments while **j** stays constant.
[a[0][0]] -> [a[1][0]] -> [a[2][0]] ... [a[M-1][0]] -> [a[0][1]] -> [a[1][1]] ...
- This means the code is traversing the array column by column. However, in typical row-major order (used in languages like C), elements in the same row are stored contiguously in memory.
(e.g., a[0][0], a[0][1], a[0][2]) are stored contiguously in memory.
- Accessing memory column by column causes "cache misses," as it doesn't align with how the data is laid out in memory.

For better locality, the function should ideally process the array row by row, which would align with the row-major storage of the array.

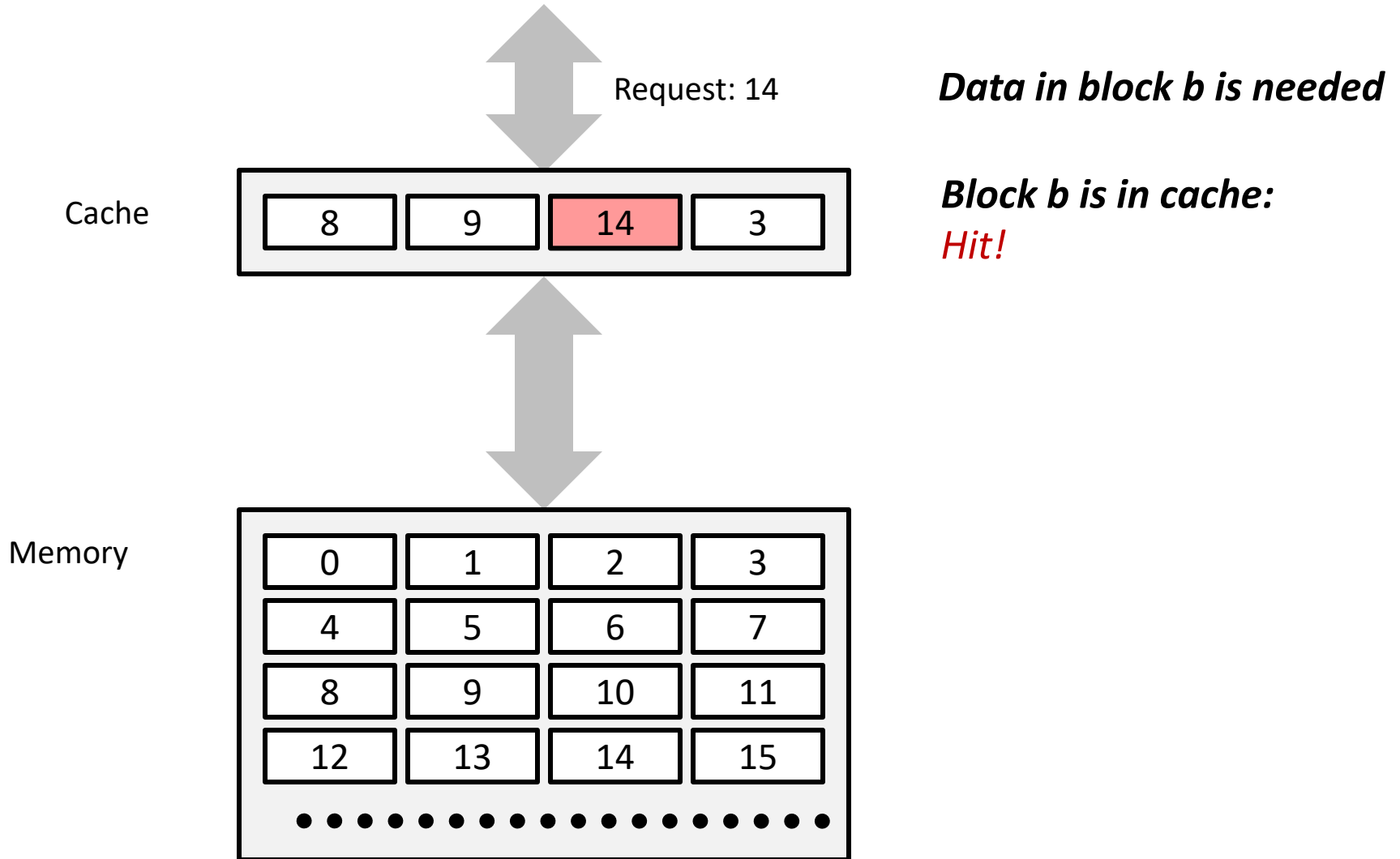
Taking Advantage of Locality

- Memory hierarchy
- Store everything on disk
- Copy recently accessed (and nearby) items from disk to smaller DRAM memory
 - Main memory
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
 - Cache memory attached to CPU

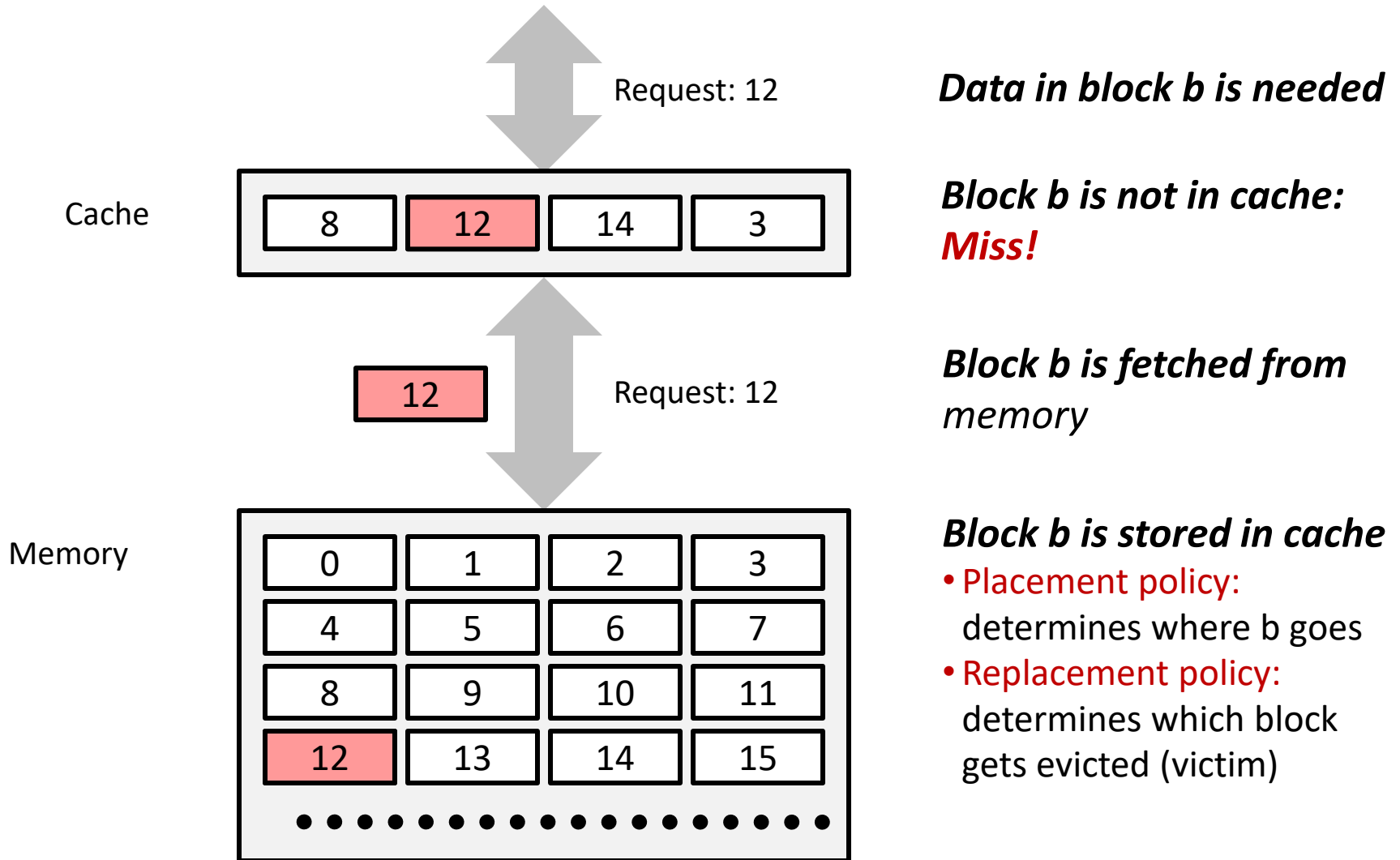
General Cache Concepts



General Cache Concepts: Hit

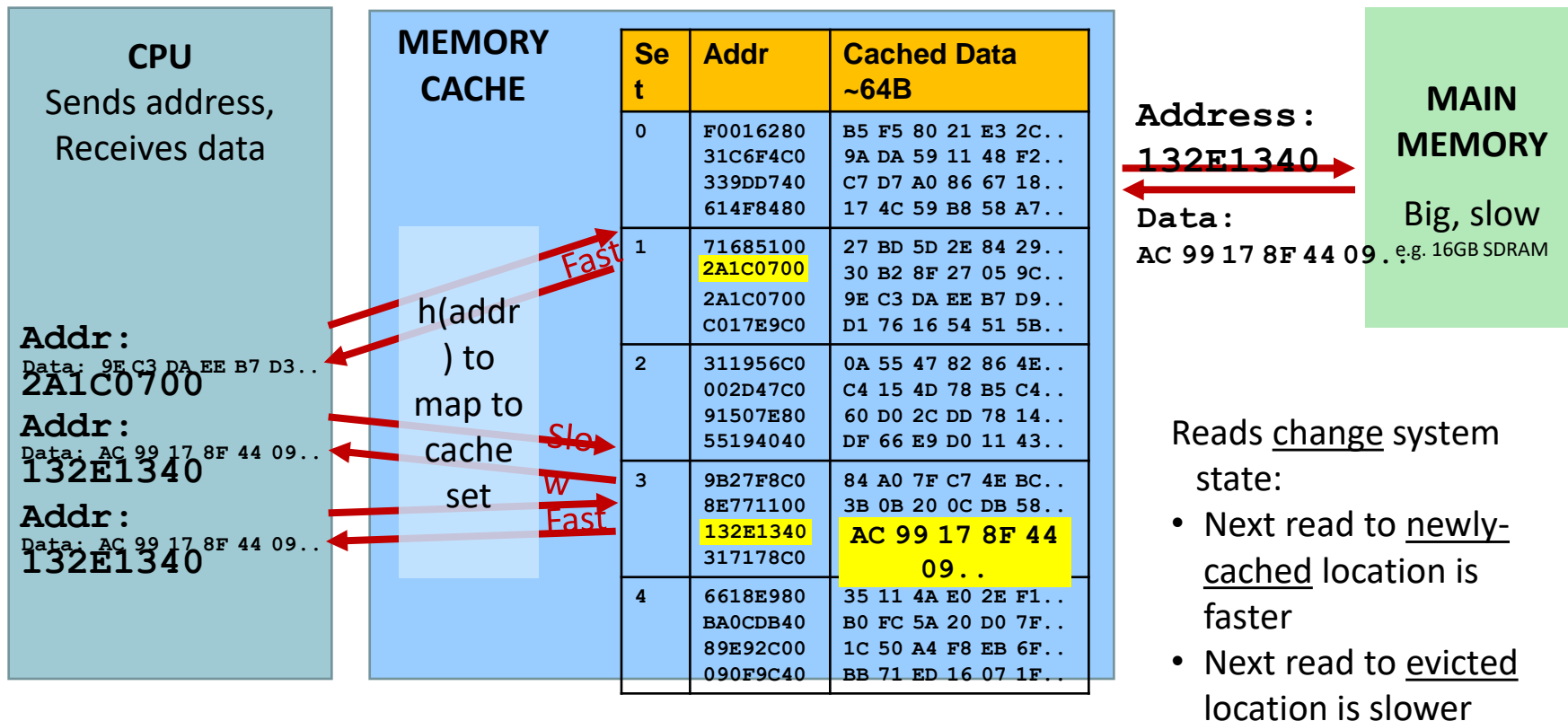


General Cache Concepts: Miss

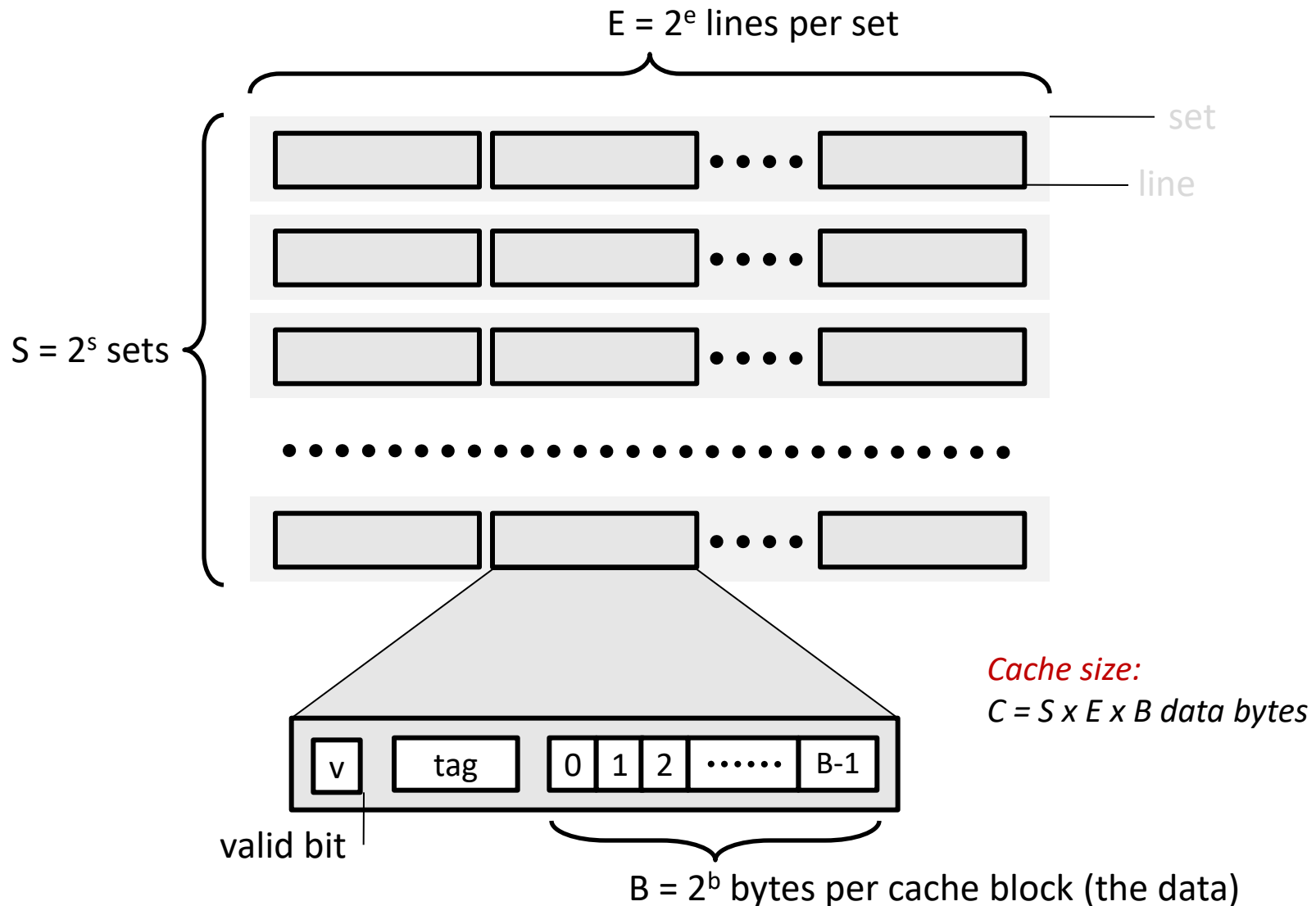


Example

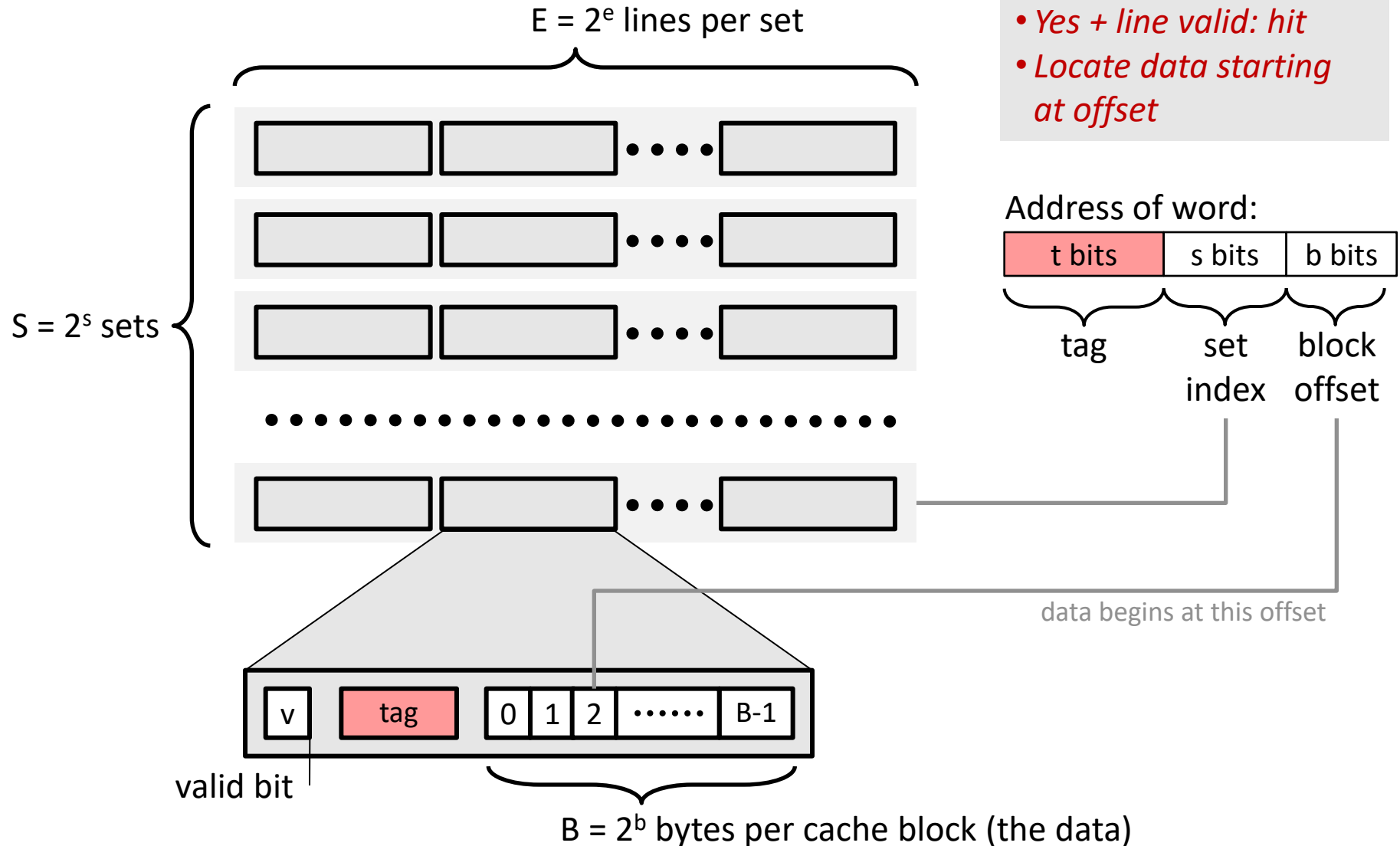
- Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



General Cache Organization (S, E, B)

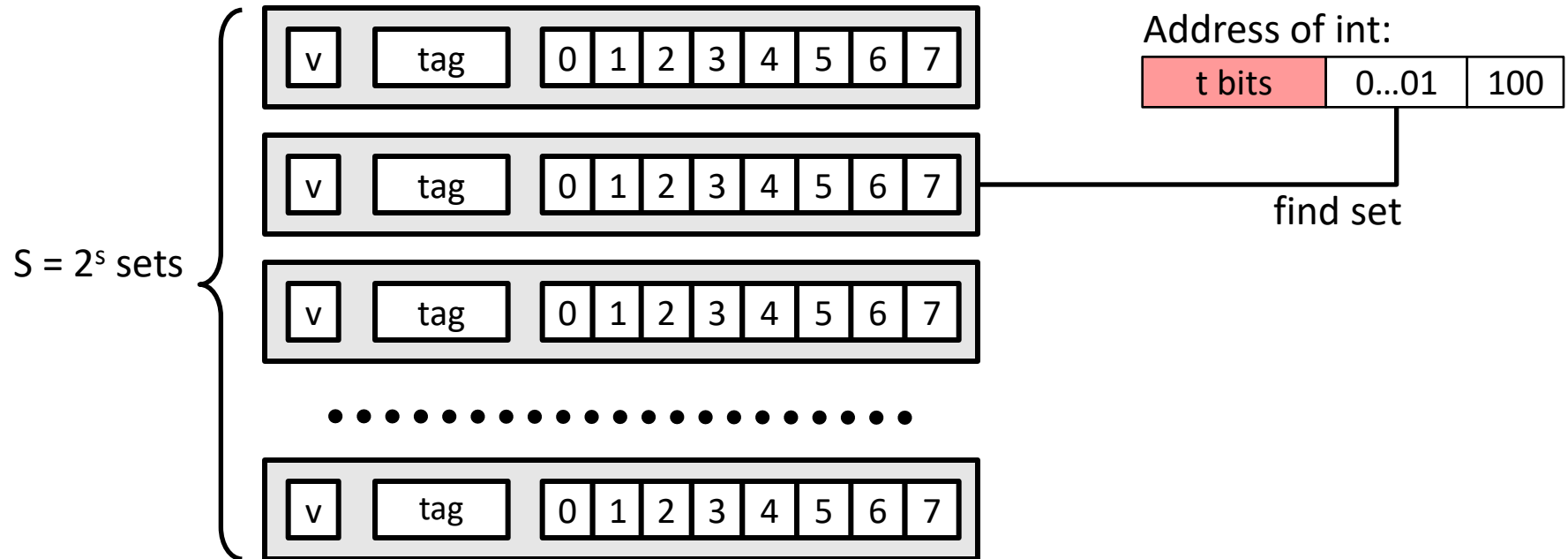


Cache Read



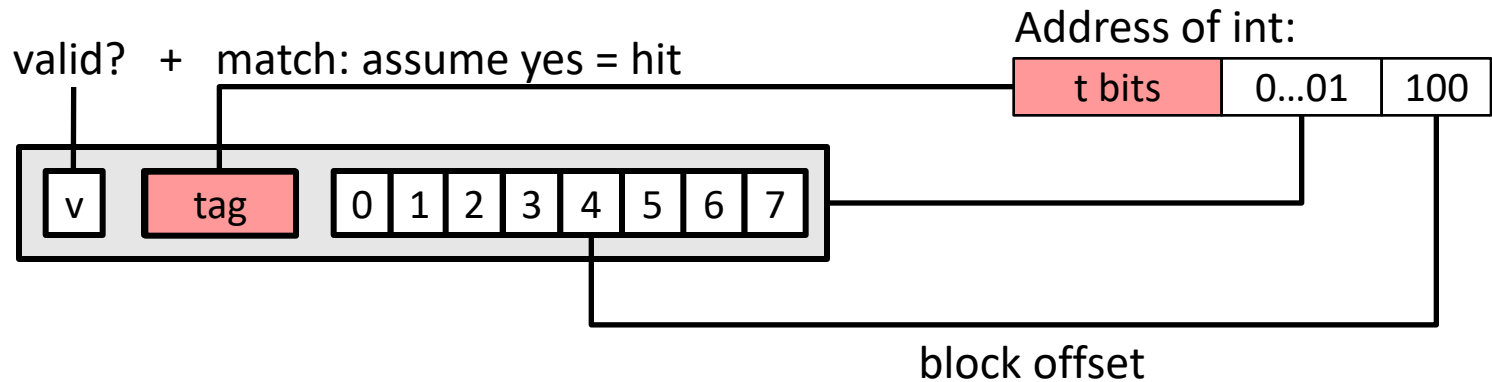
Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes



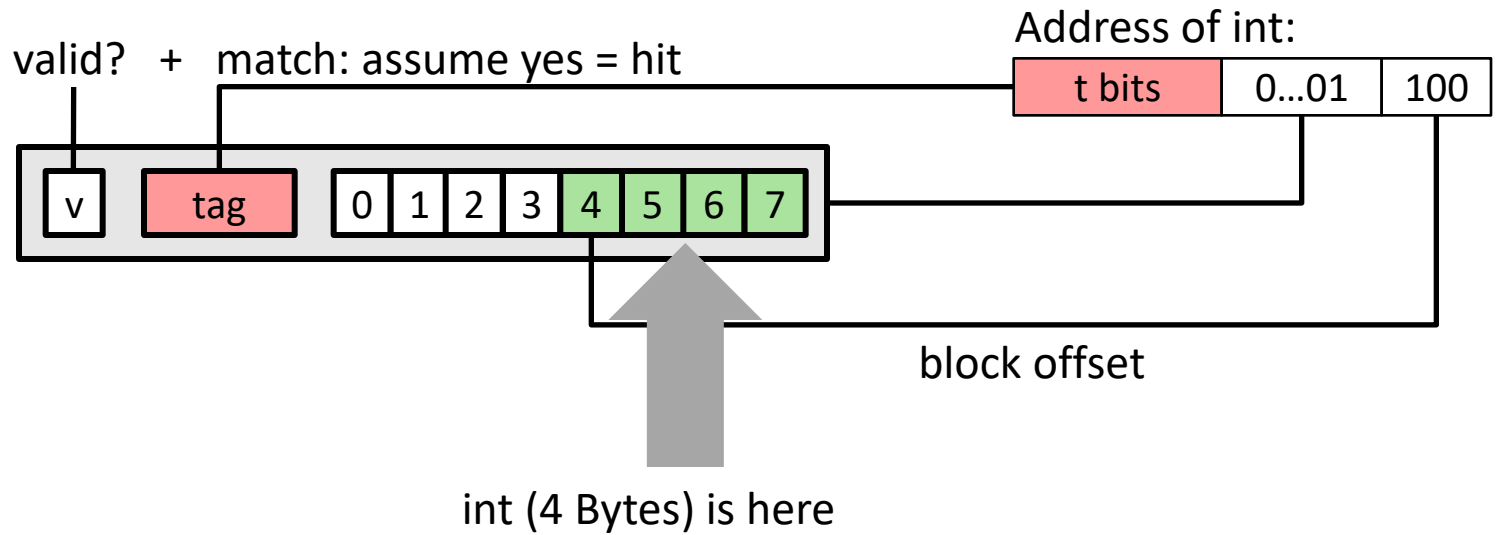
Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes



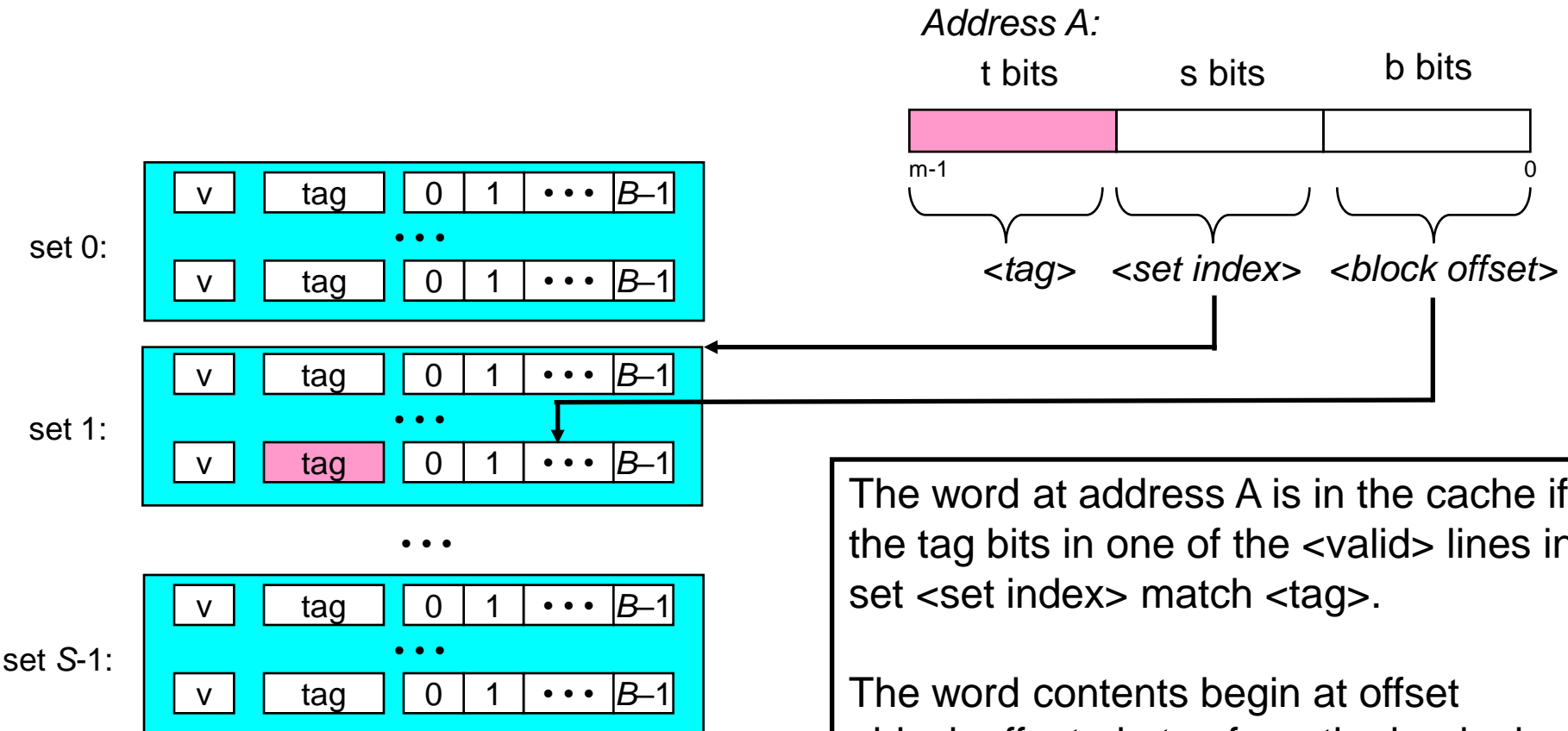
Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes



No match: old line is evicted and replaced

Addressing Caches

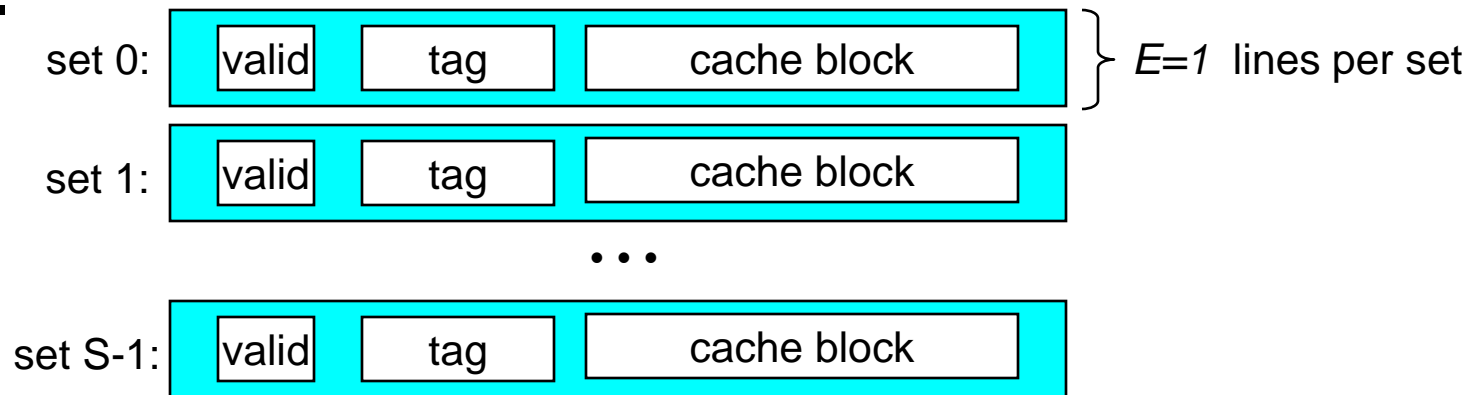


The word at address A is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>.

The word contents begin at offset <block offset> bytes from the beginning of the block.

Direct-Mapped Cache

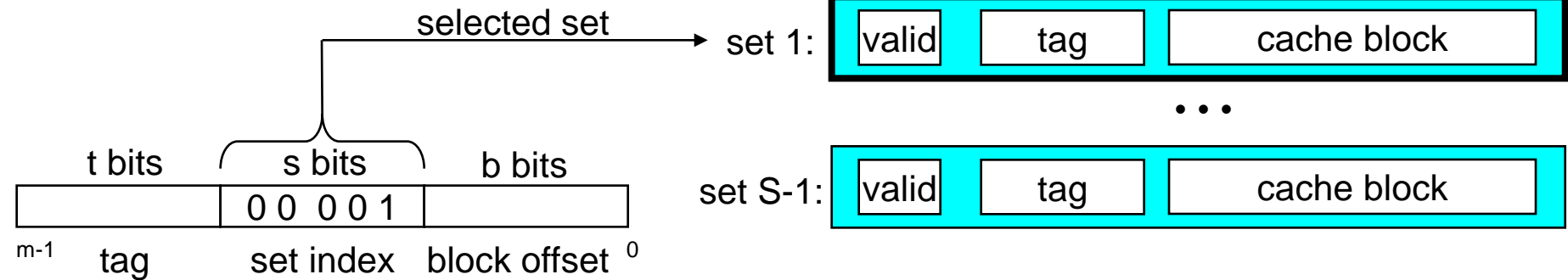
- Simplest kind of cache
- Characterized by exactly one line per set.



Accessing Direct-Mapped Caches

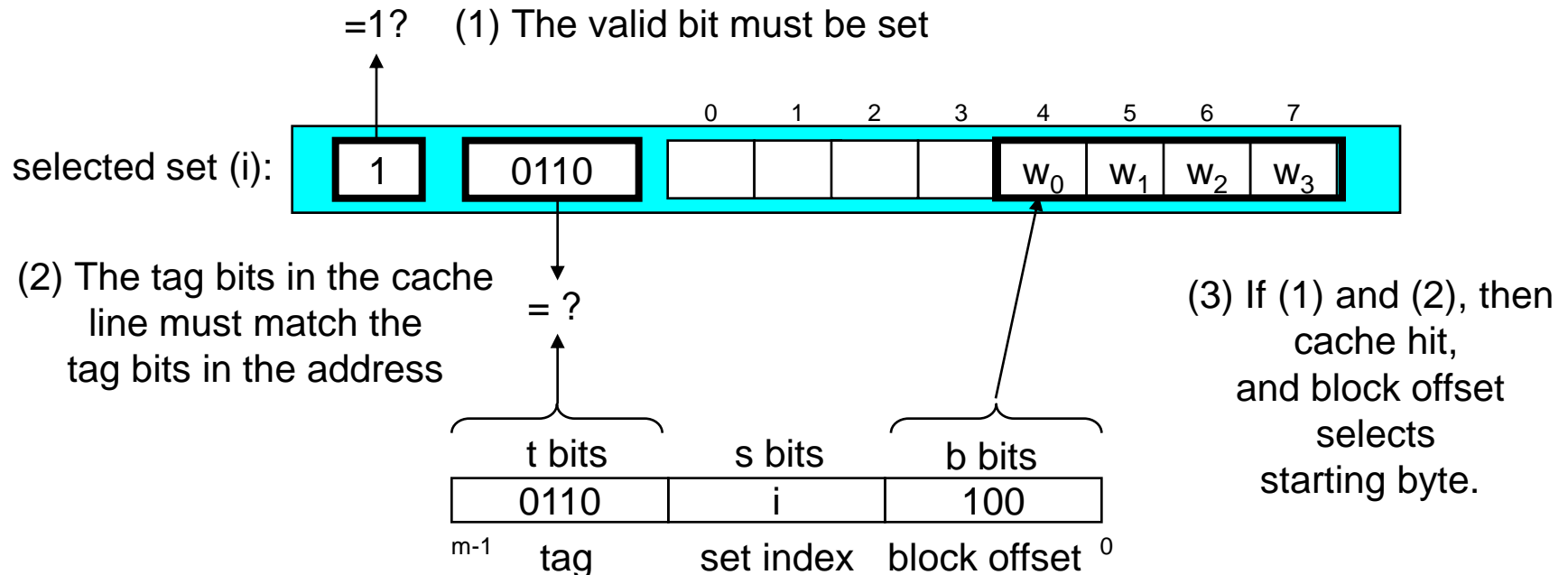
- Set selection

- Use the set index bits to determine the set of interest.



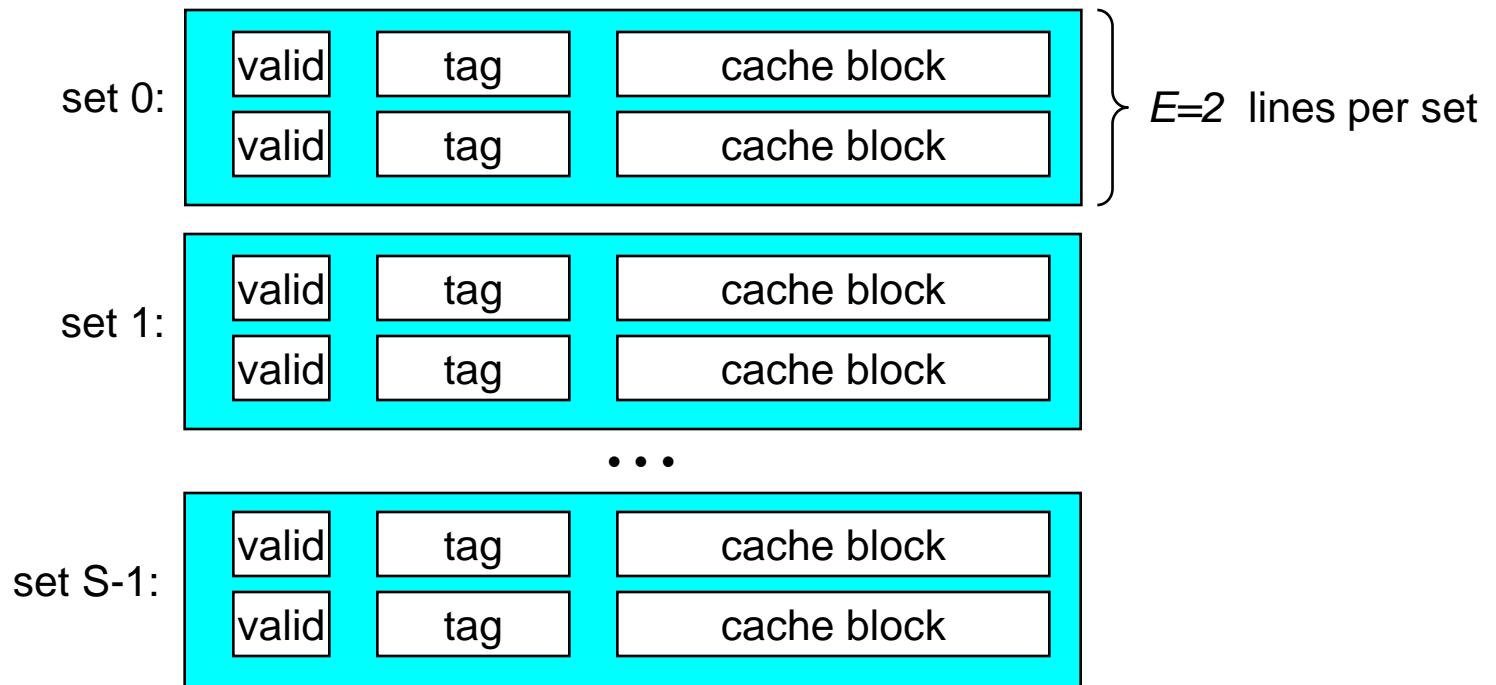
Accessing Direct-Mapped Caches

- Line matching and word selection
 - **Line matching:** Find a valid line in the selected set with a matching tag
 - **Word selection:** Then extract the word



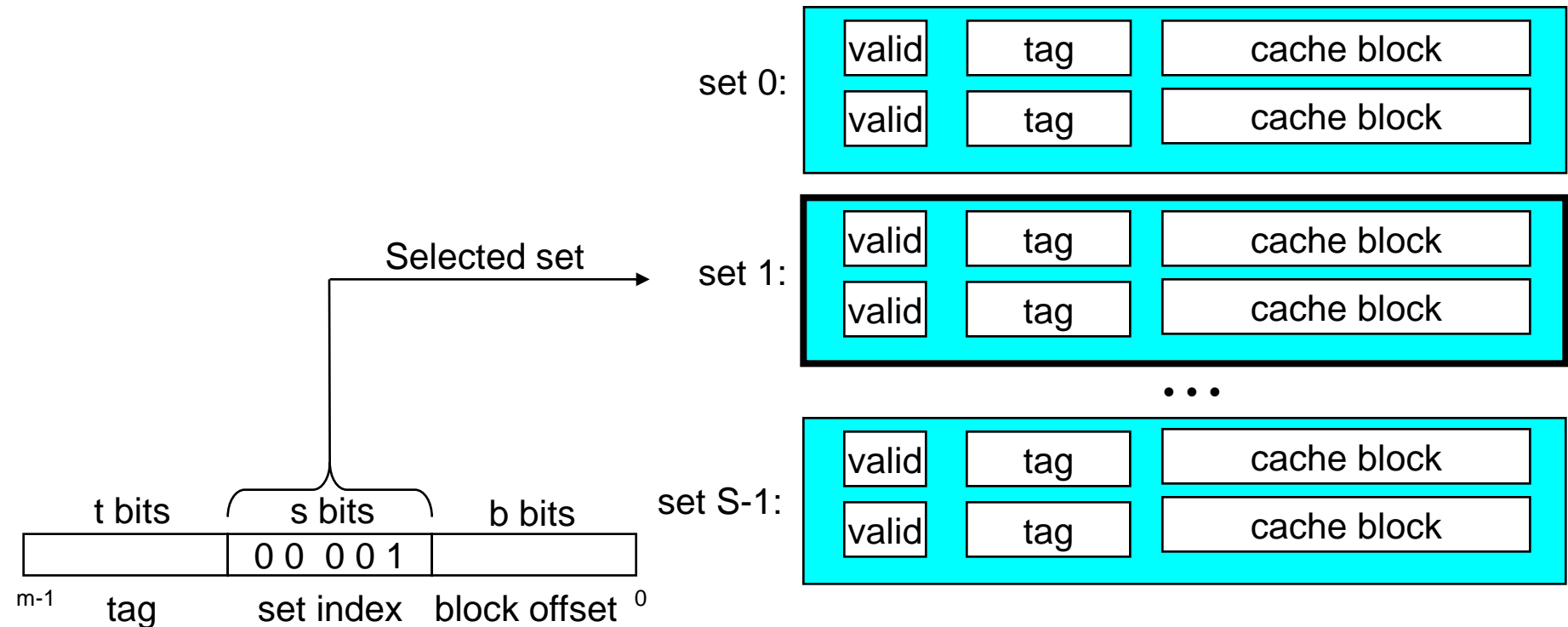
Set Associative Caches

- Characterized by more than one line per set



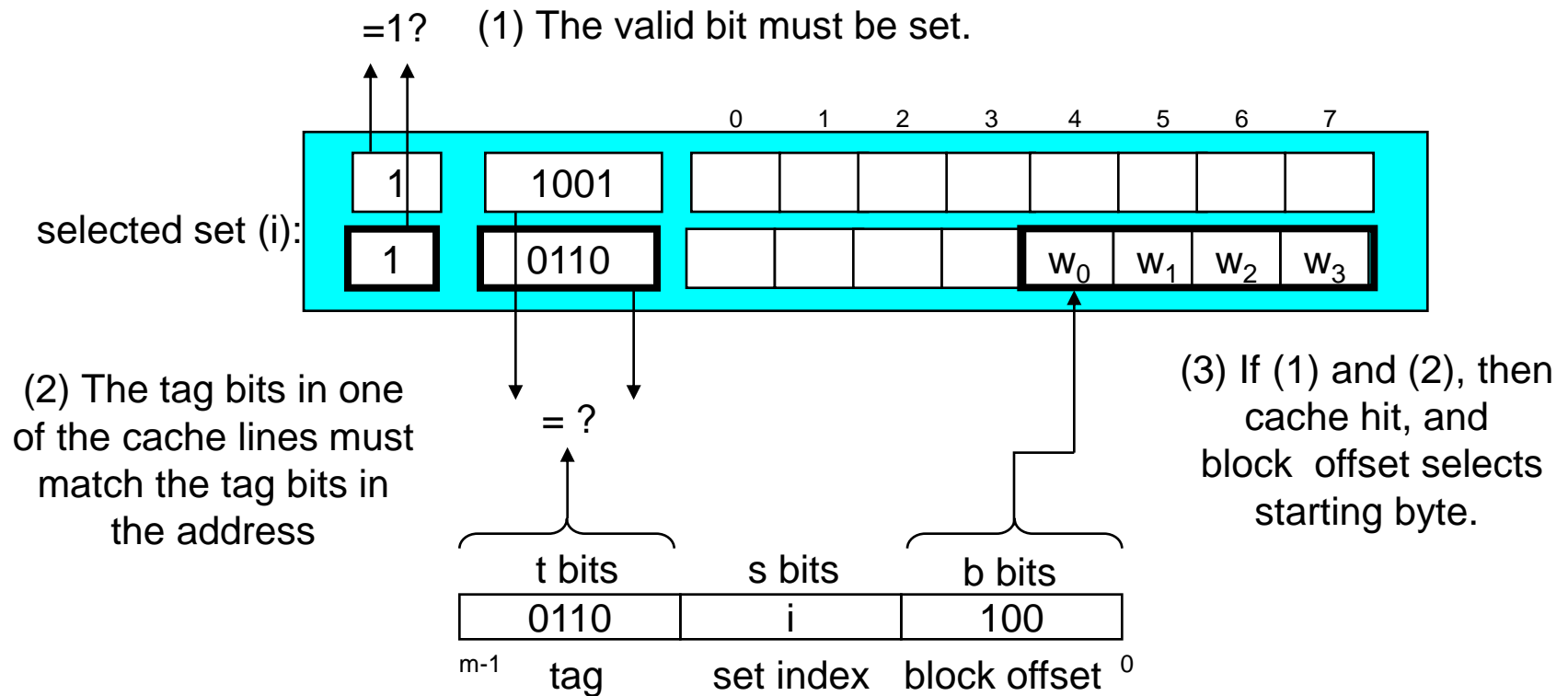
Accessing Set Associative Caches

- Set selection
 - identical to direct-mapped cache



Accessing Set Associative Caches

- Line matching and word selection
 - must compare the tag in each valid line in the selected set.

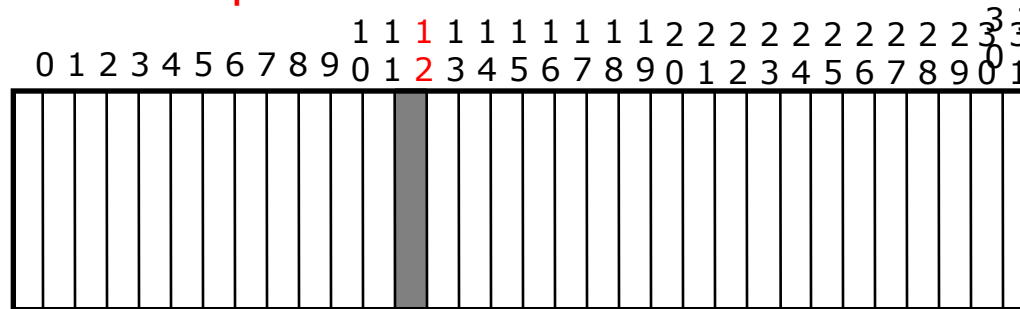


Placement policy

block 12 can be placed

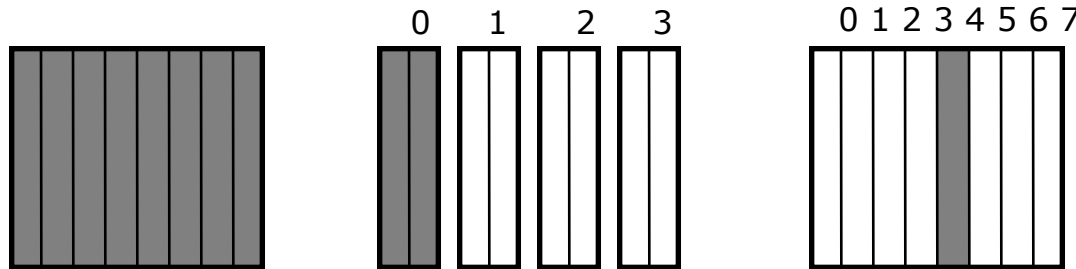
Block
Number

Memory



Set
Number

Cache

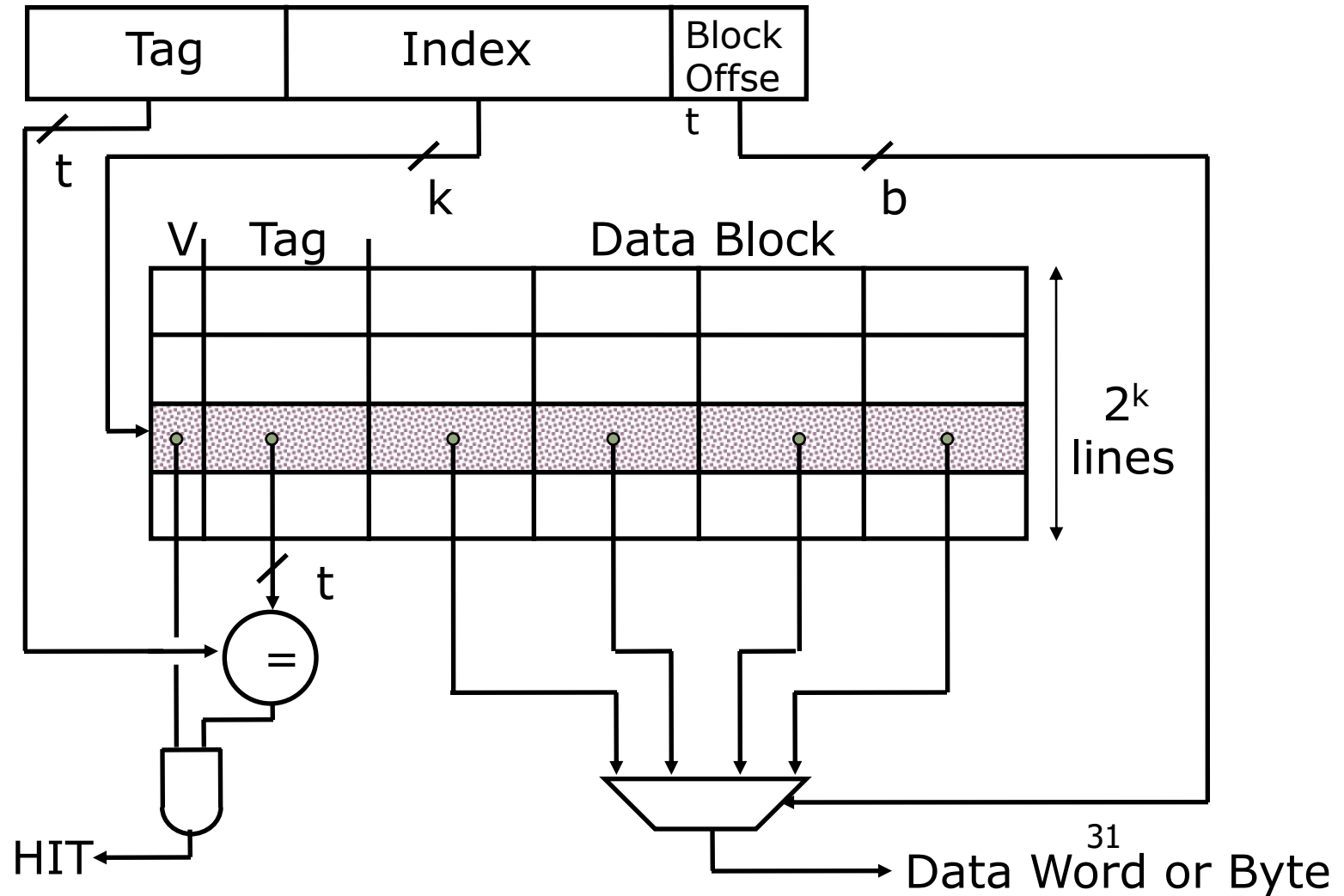


Fully
Associative
anywhere

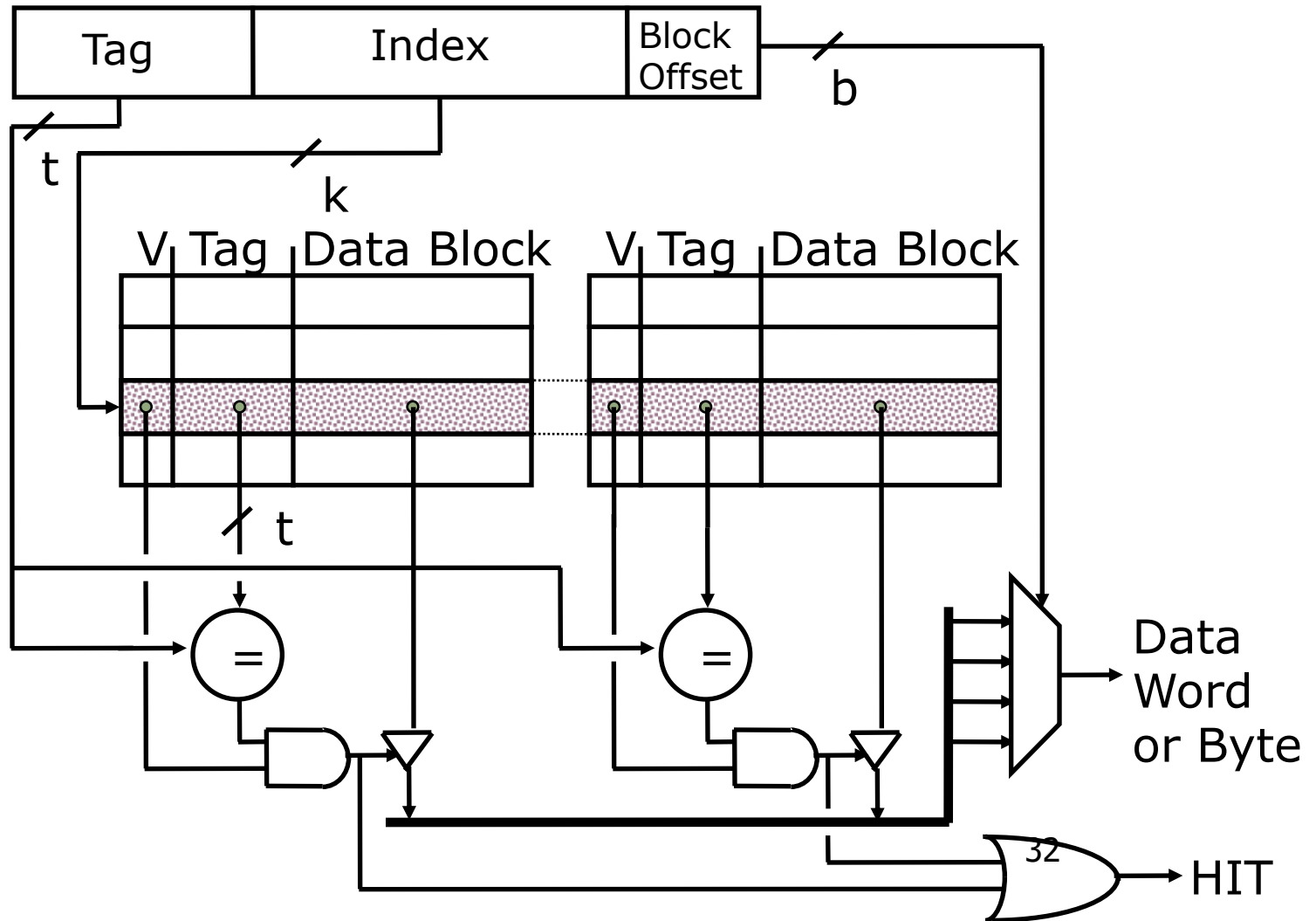
(2-way) Set
Associative
anywhere in
set 0
($12 \bmod 4$)

Direct
Mapped
only into
block 4
($12 \bmod 8$)

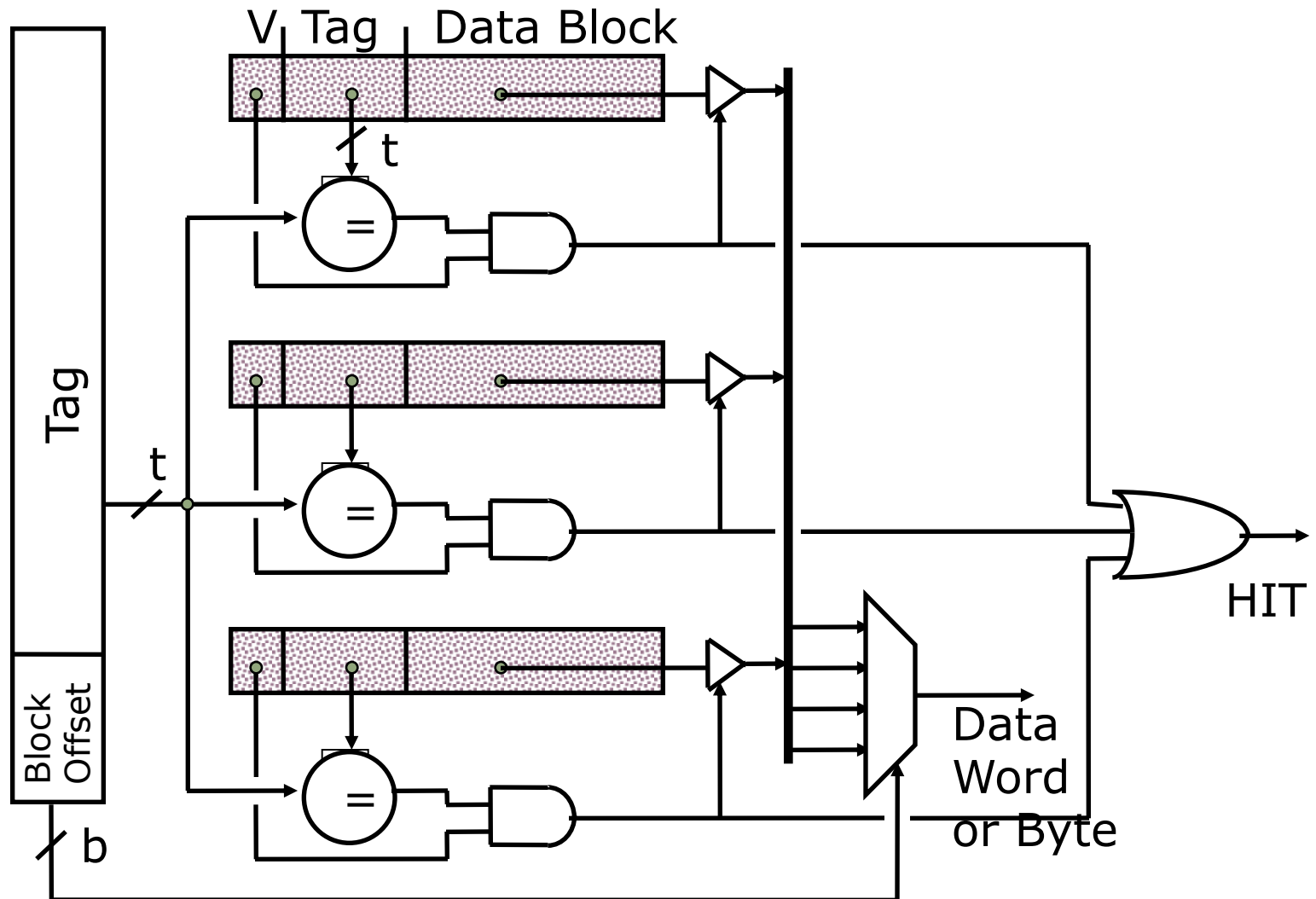
Direct-Mapped Cache



2-Way Set-Associative Cache

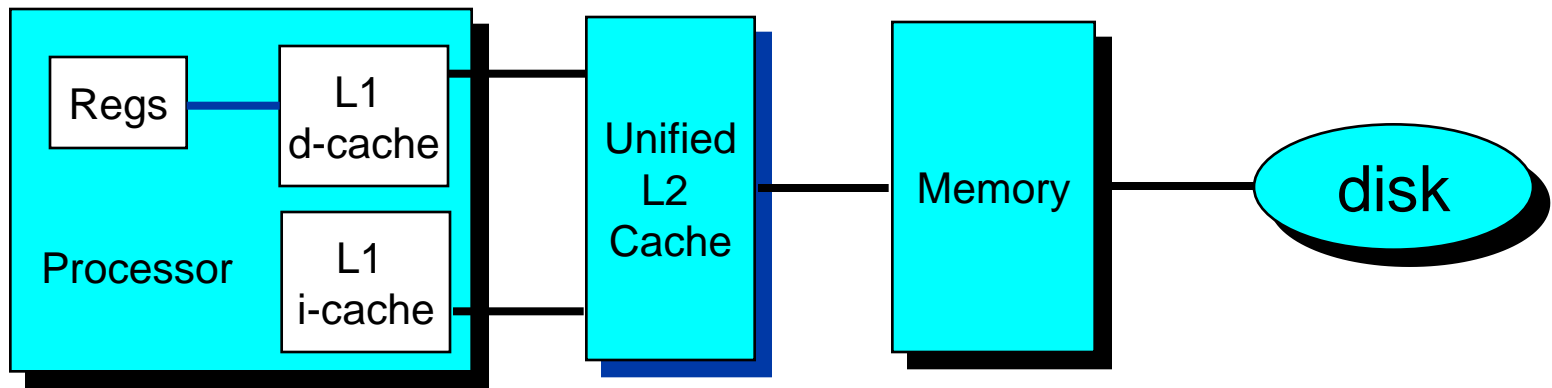


Fully Associative Cache




Multi-Level Caches

- Options: separate **data** and **instruction caches**, or a **unified cache**

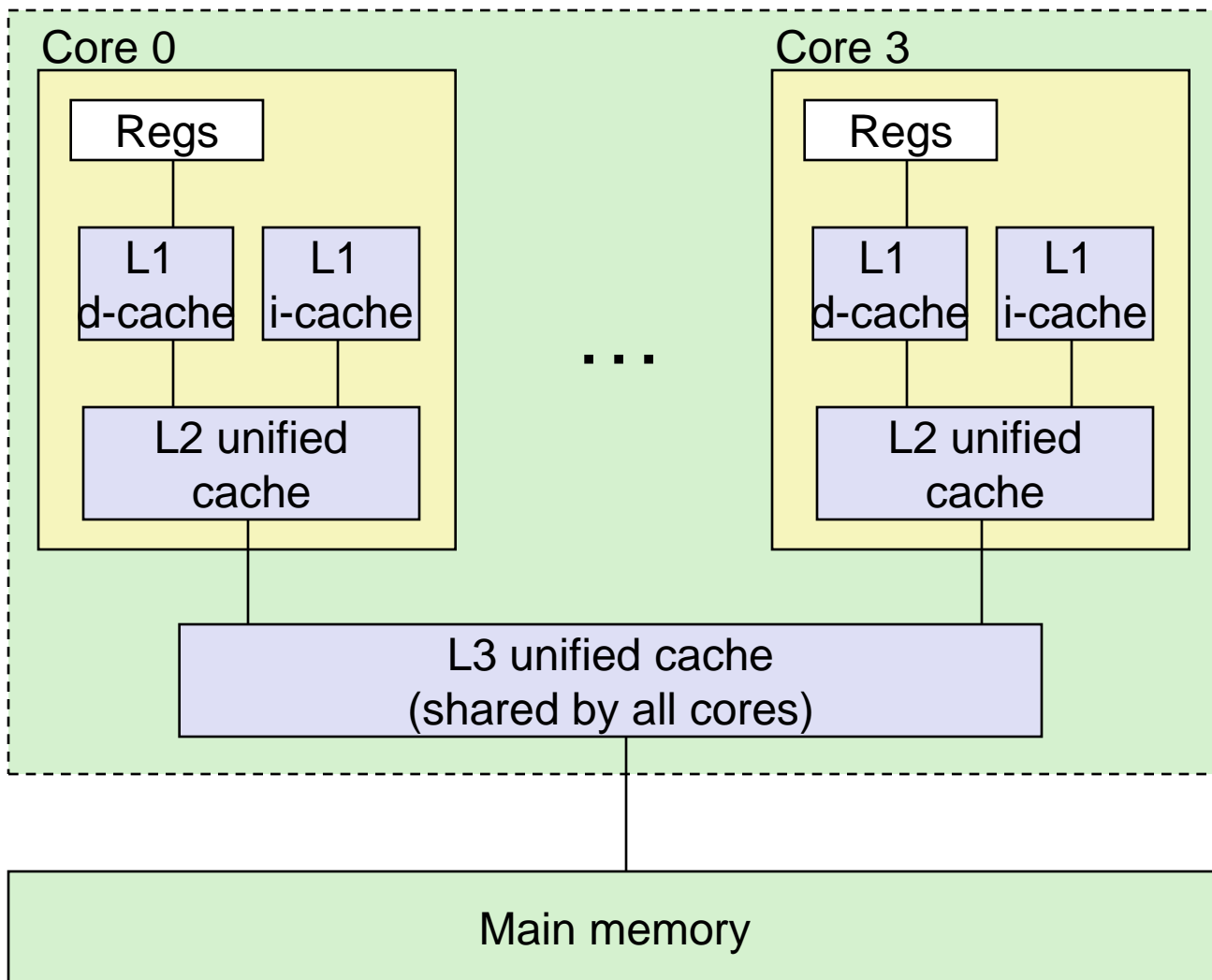


size:	200 B	8-64 KB	1-4MB SRAM	128 MB DRAM	30 GB
speed:	3 ns	3 ns	6 ns	60 ns	8 ms
\$/Mbyte:			\$100/MB	\$1.50/MB	\$0.05/MB
line size:	8 B	32 B	32 B	8 KB	


 larger, slower, cheaper

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: 11 cycles

L3 unified cache:
8 MB, 16-way,
Access: 30-40 cycles

Block size: 64 bytes for
all caches.

Cache Performance Metrics

■ Miss Rate

- Fraction of memory references not found in cache (misses / accesses)
= 1 – 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:
 - 1-2 clock cycle for L1
 - 5-20 clock cycles for L2

■ Miss Penalty

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

Lets 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.

Memory Technology

- Static RAM (SRAM)
 - 0.5ns – 2.5ns, \$2000 – \$5000 per GB
- Dynamic RAM (DRAM)
 - 50ns – 70ns, \$20 – \$75 per GB
- Magnetic disk
 - 5ms – 20ms, \$0.20 – \$2 per GB
- Ideal memory
 - Access time of SRAM
 - Capacity and cost/GB of disk

Exercise

- (a) What are the two characteristics of program memory accesses that caches exploit?
- (b) Why is miss rate not a good metric for evaluating cache performance? What is the appropriate metric? Give its definition. What is the reason for using a combination of first and second- level caches rather than using the same chip area for a larger first-level cache?

Solution:

Exercise

- (a) What are the two characteristics of program memory accesses that caches exploit?
- (b) Why is miss rate not a good metric for evaluating cache performance? What is the appropriate metric? Give its definition. What is the reason for using a combination of first and second- level caches rather than using the same chip area for a larger first-level cache?

Solution:

(a)

Temporal and spatial locality

(b)

The ultimate metric for cache performance is average

access time: $t_{avg} = t_{hit} + \text{miss-rate} * t_{miss}$. The miss rate is only one component of this equation. A cache may have a low miss rate, but an extremely high penalty per miss, making it lower-performing than a cache with a higher miss rate but a substantially lower miss penalty. Alternatively, it may have a low miss rate but a high hit time (this is true for large fully associative caches, for instance).

Multiple levels of cache are used for exactly this reason. Not all of the performance factors can be optimized in a single cache. Specifically, with t_{miss} (memory latency) given, it is difficult to design a cache which is both fast in the common case (a hit) and minimizes the costly uncommon case by having a low miss rate. These two design goals are achieved using two caches. The first level cache minimizes the hit time, therefore it is usually small with a low-associativity. The second level cache minimizes the miss rate, it is usually large with large blocks and a higher associativity.