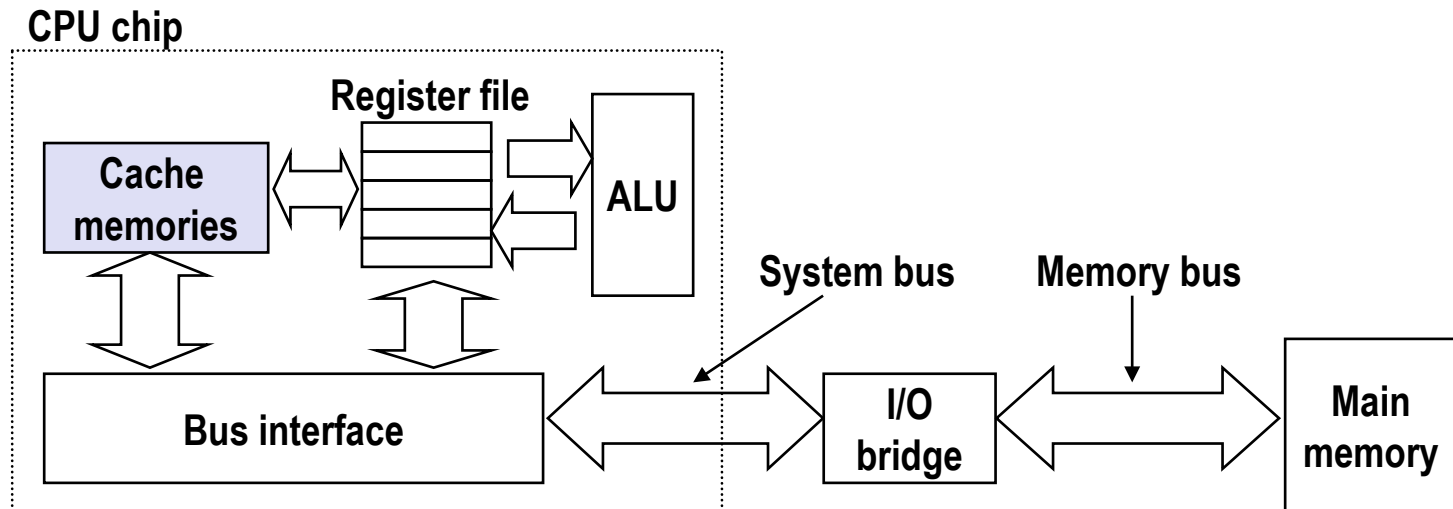
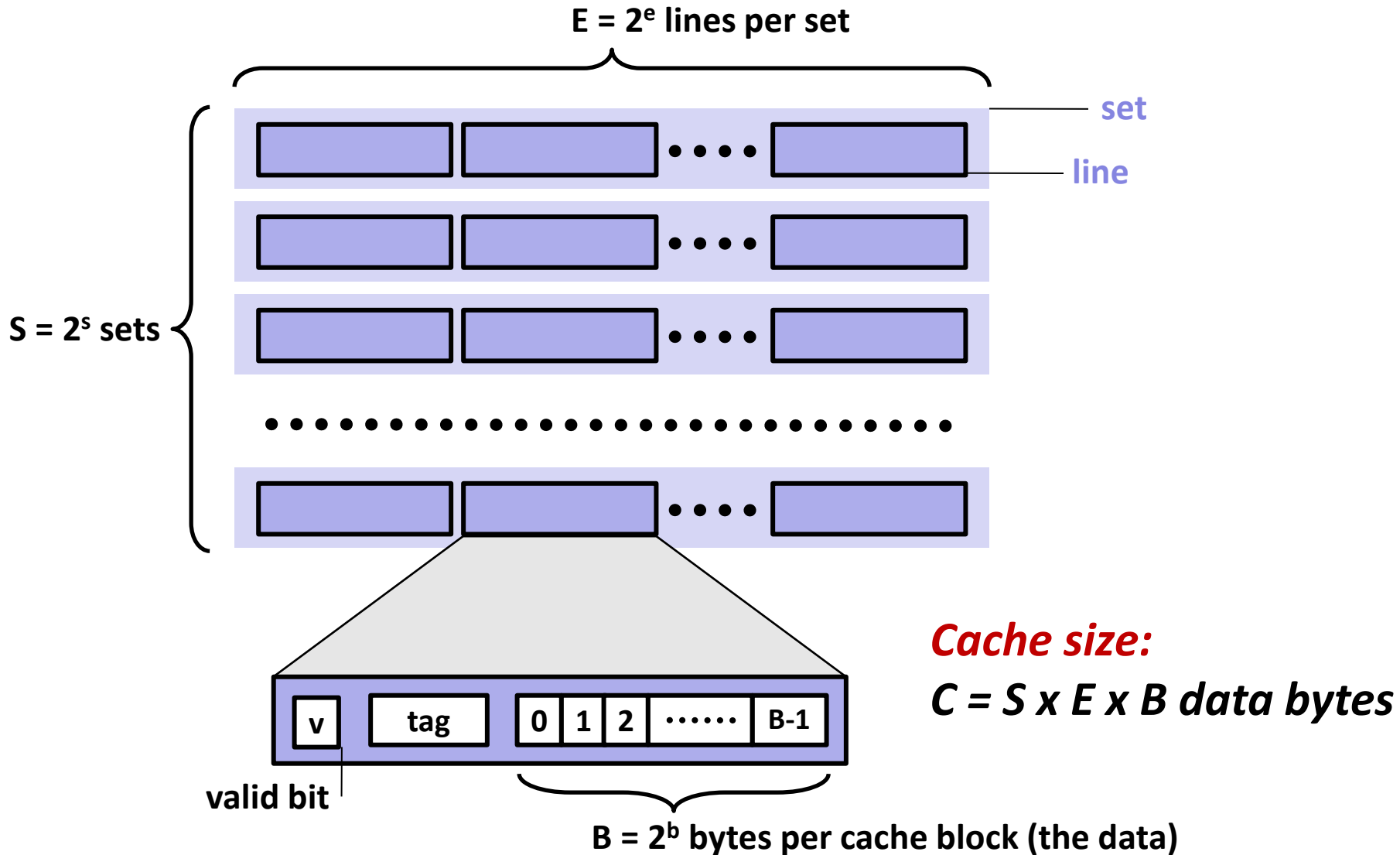


Cache Memories

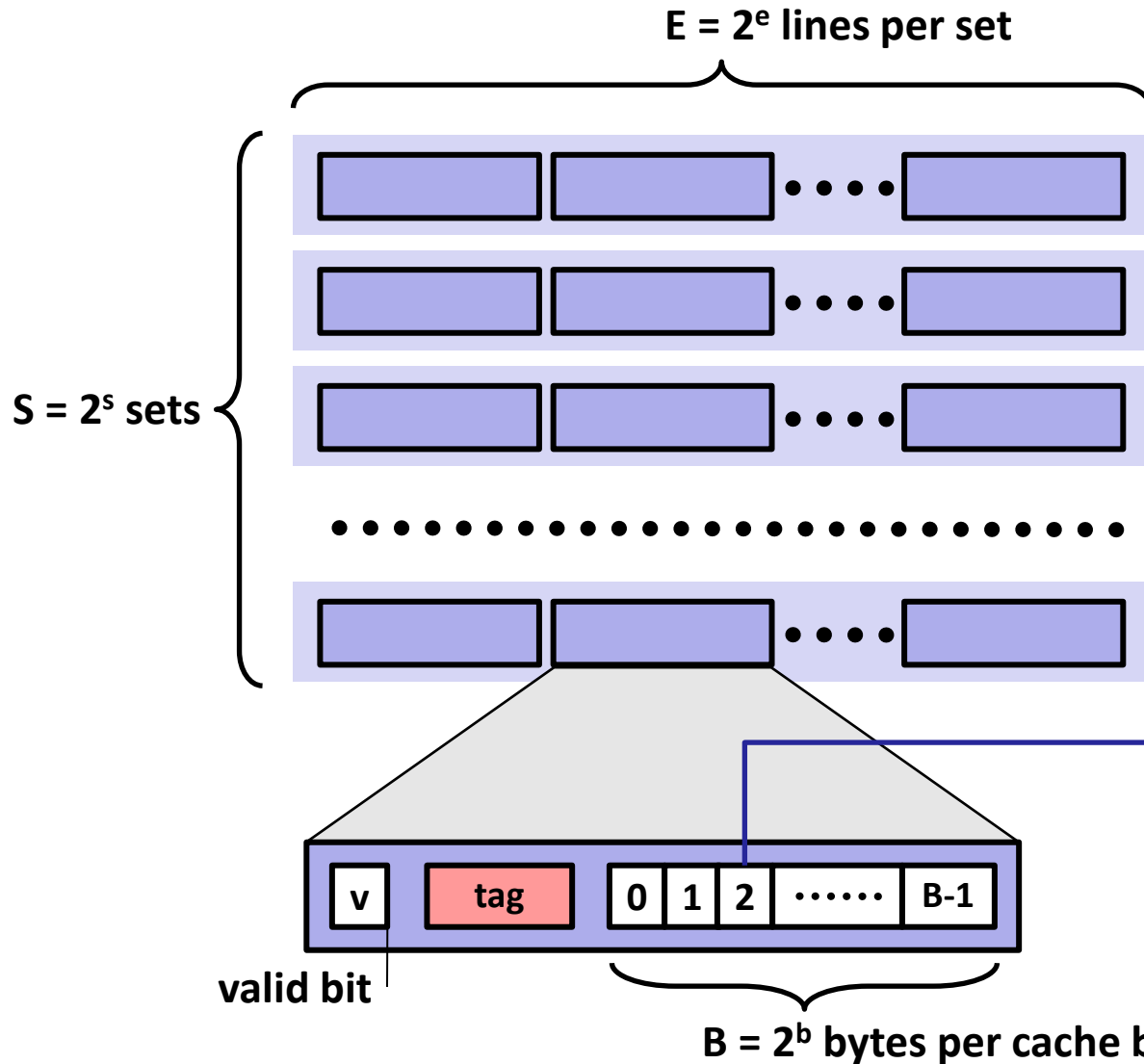
- **Cache memories** are small, fast SRAM-based memories managed automatically in hardware.
 - Hold frequently accessed blocks of main memory
- CPU looks first for data in caches (e.g., L1, L2, and L3), then in main memory.
- Typical system structure:



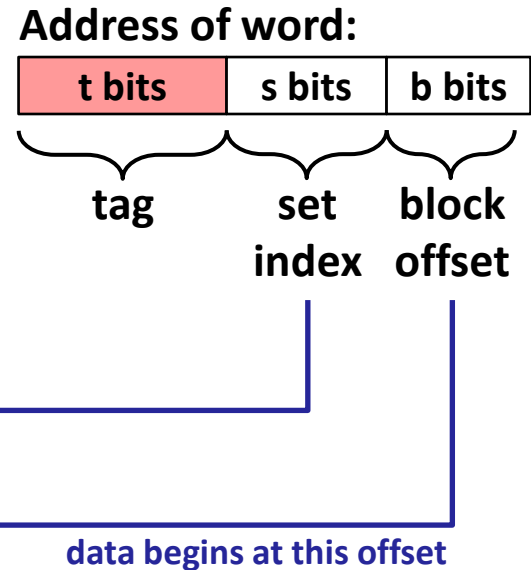
General Cache Organization (S, E, B)



Cache Read



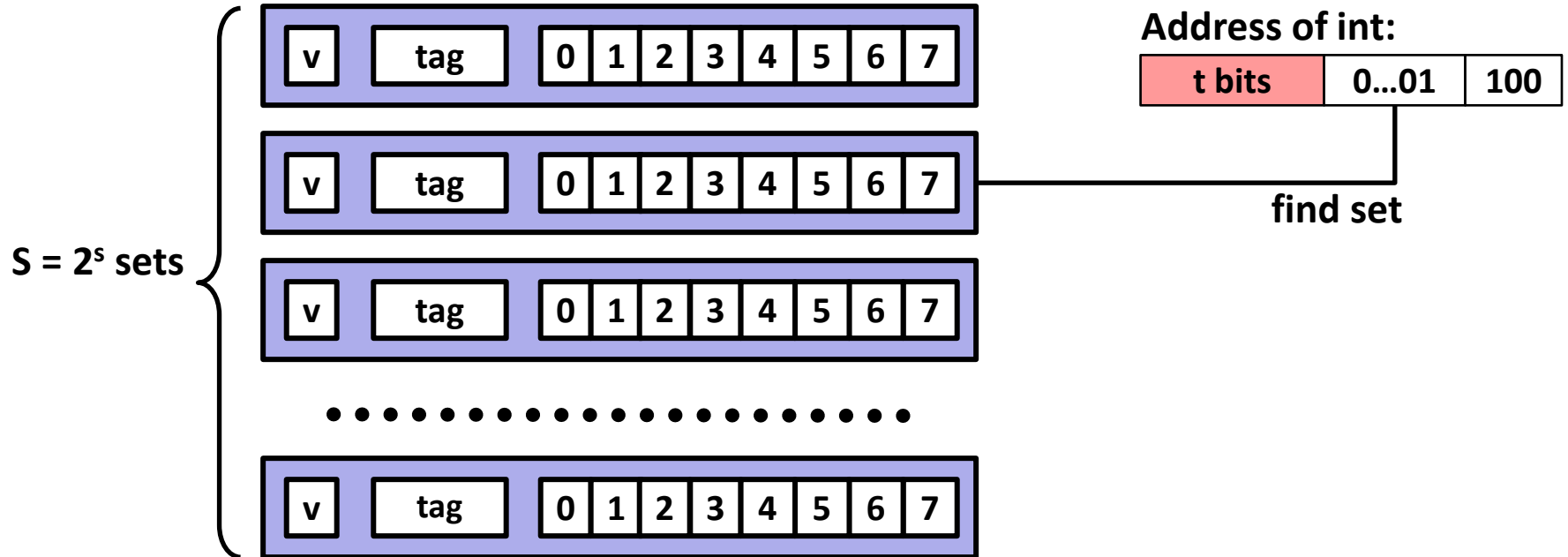
- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*



Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set

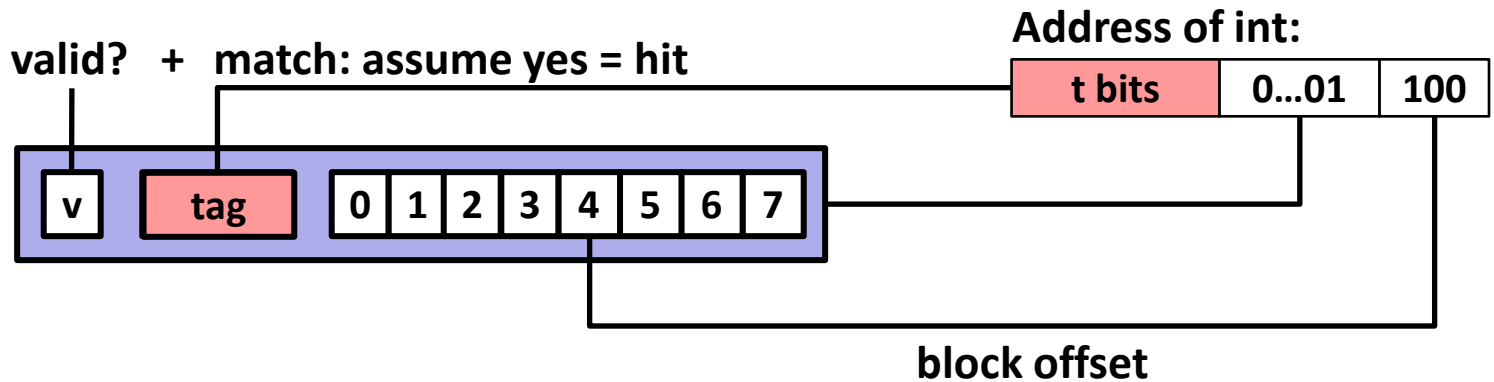
Assume: cache block size 8 bytes



Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set

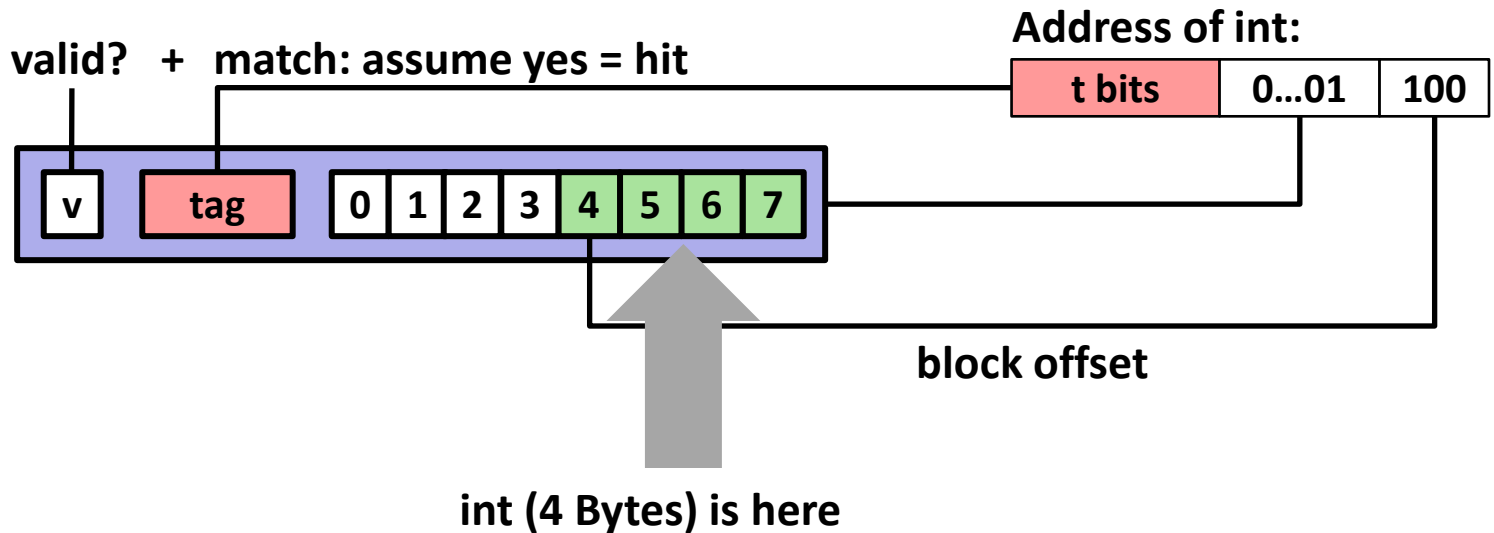
Assume: cache block size 8 bytes



Example: Direct Mapped Cache (E = 1)

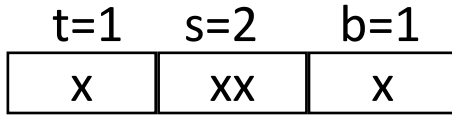
Direct mapped: One line per set

Assume: cache block size 8 bytes

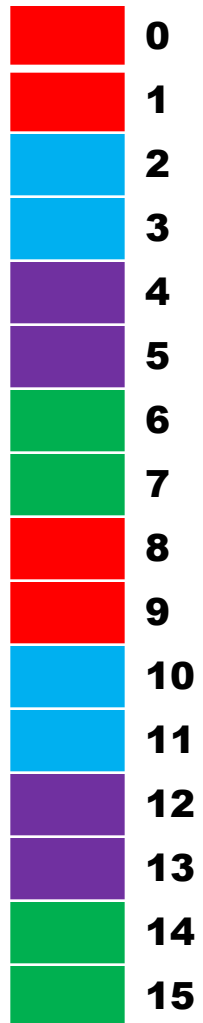


No match: old line is evicted and replaced

Direct-Mapped Cache Simulation



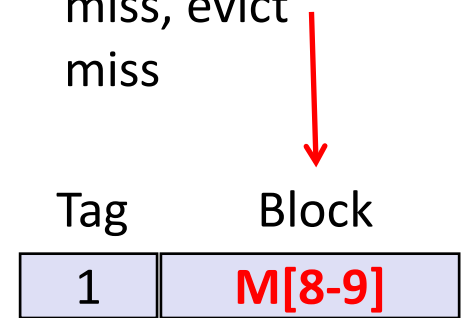
M=16 byte addresses, B=2 bytes/block,
S=4 sets, E=1 Blocks/set



Address trace (reads, one byte per read):

0	[<u>0000</u> ₂],	miss
1	[<u>0001</u> ₂],	hit
7	[<u>0111</u> ₂],	miss
8	[<u>1000</u> ₂],	miss, evict
12	[<u>1100</u> ₂]	miss

	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1	0		
Set 2	1	1	M[12-13]
Set 3	1	0	M[6-7]

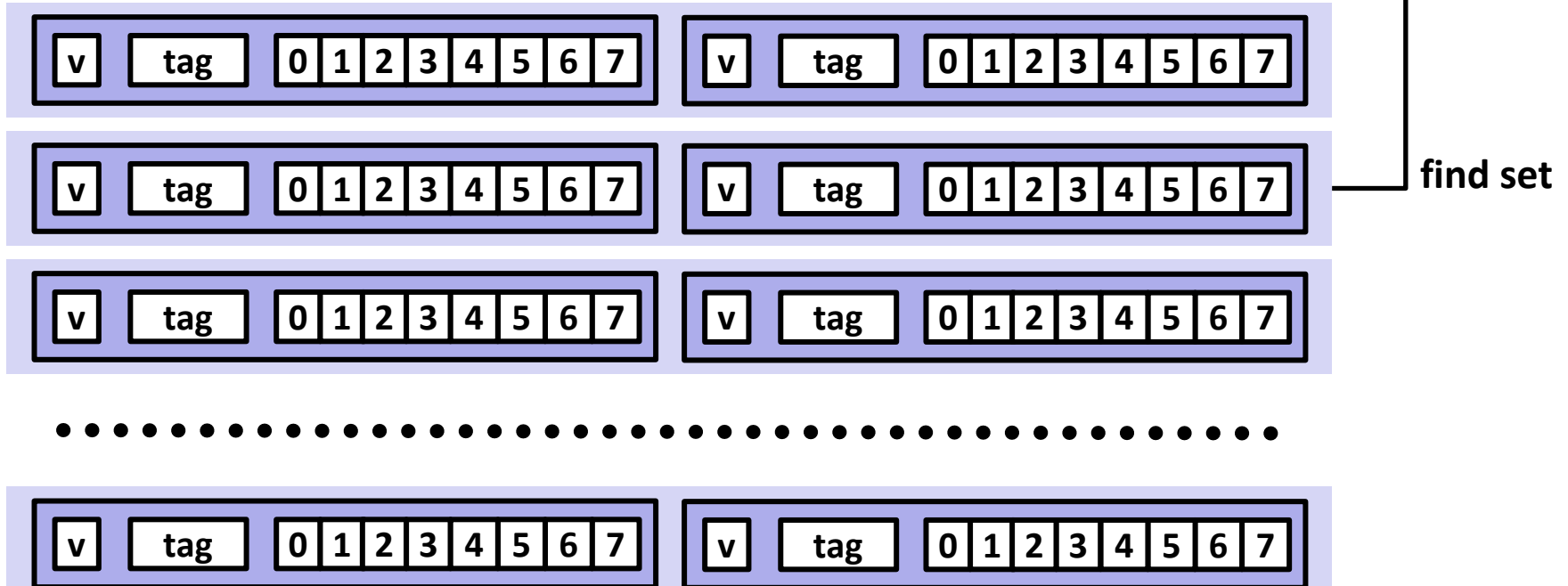
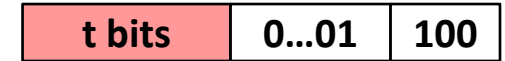


E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache block size 8 bytes

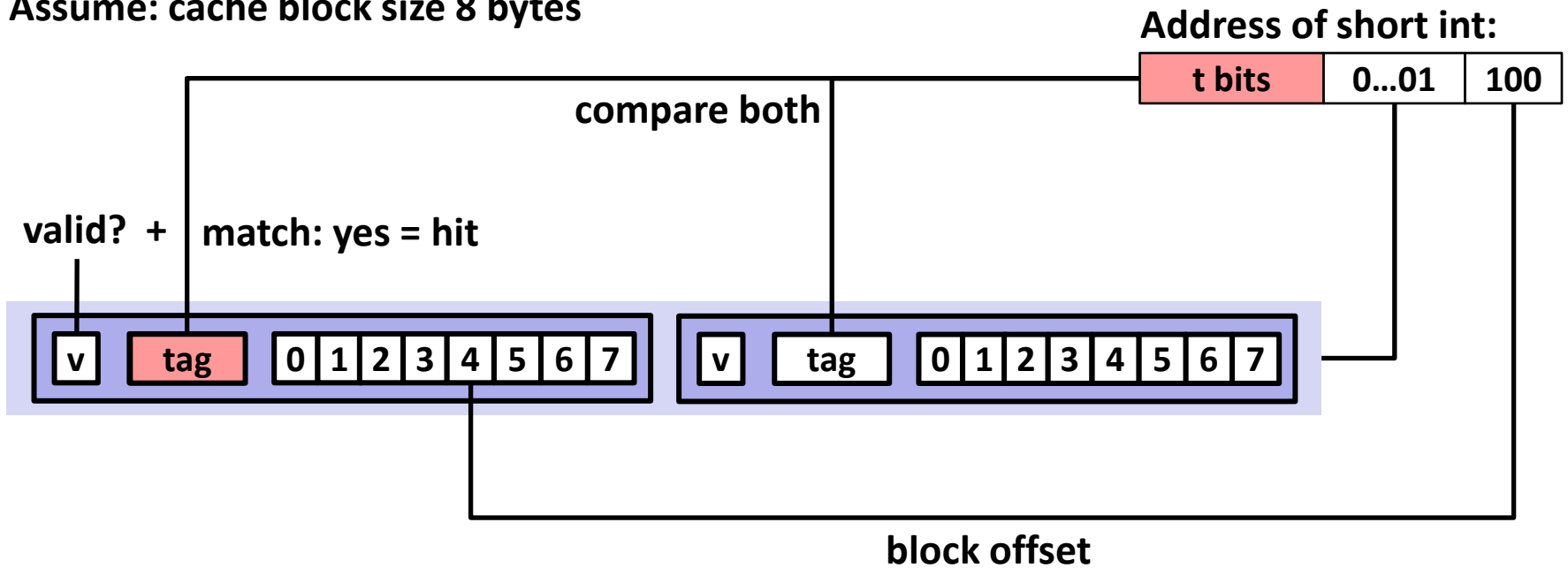
Address of short int:



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

E = 2: Two lines per set

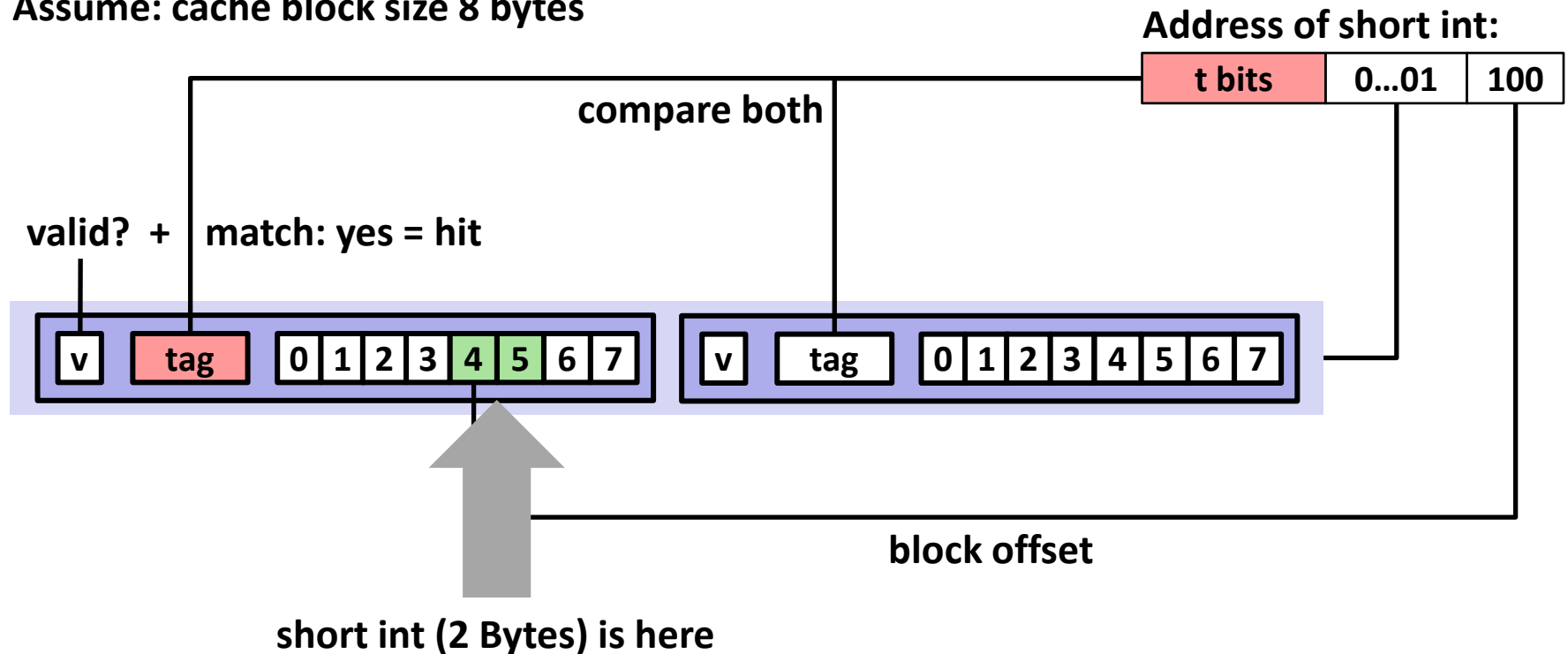
Assume: cache block size 8 bytes



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

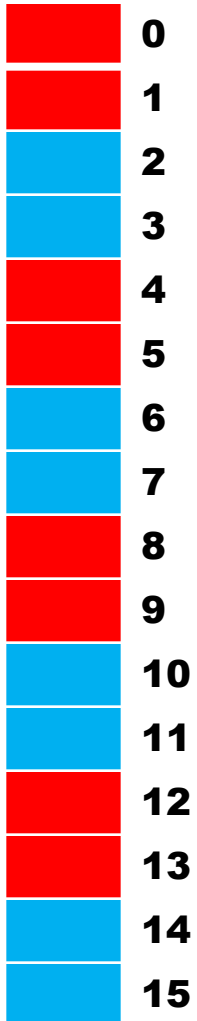
t=2	s=1	b=1
xx	x	x

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

Address trace (reads, one byte per read):

0	[0000 ₂],	miss
1	[0001 ₂],	hit
7	[0111 ₂],	miss
8	[1000 ₂],	miss
14	[1110 ₂]	miss

	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	1	11	M[14-15]

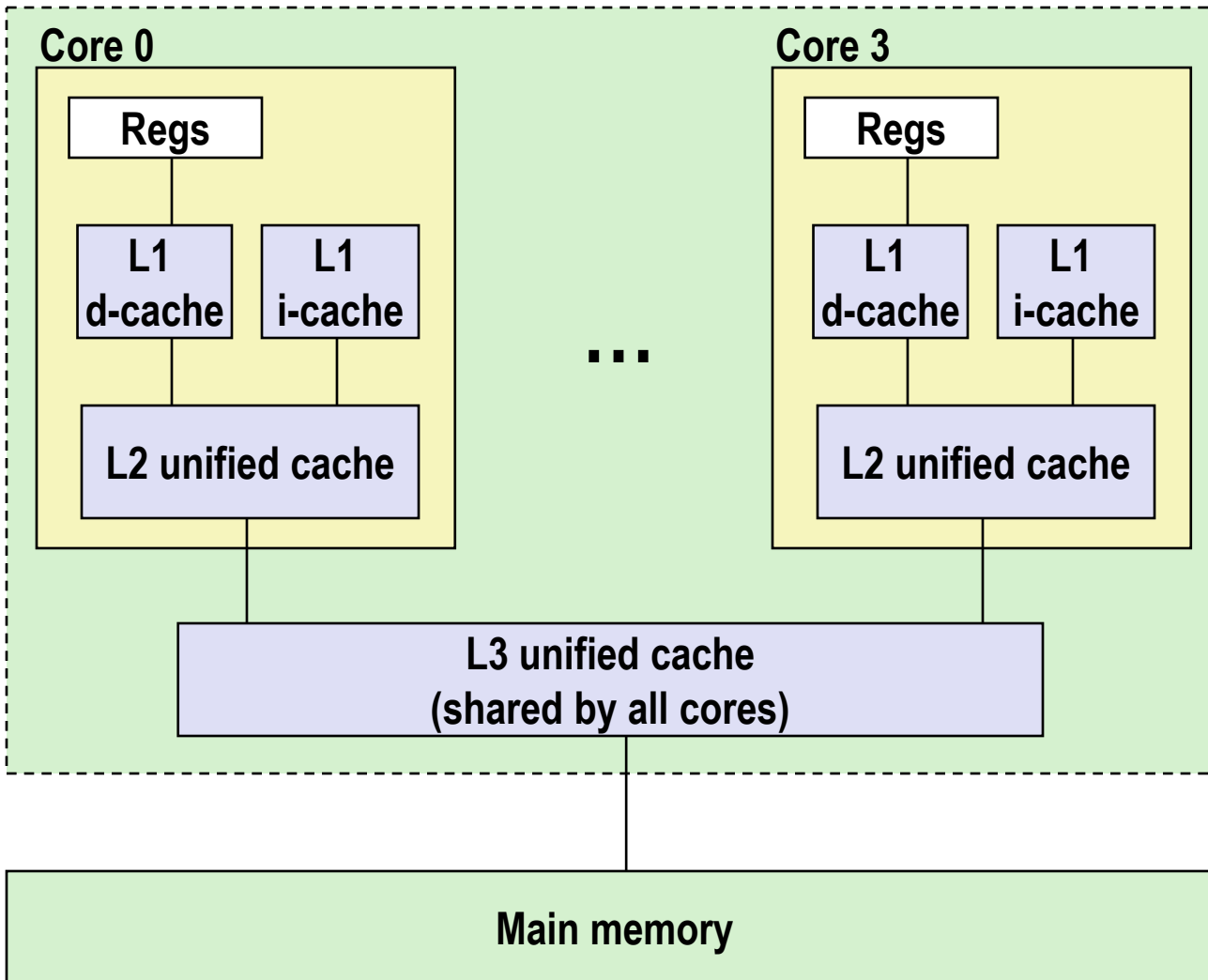


What about writes?

- **Multiple copies of data exist:**
 - L1, L2, 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 immediately to memory)
- **Typical**
 - Write-through + No-write-allocate
 - **Write-back + Write-allocate**

Intel Core i7 Cache Hierarchy, Gen 6, '16

Processor package



L1 i-cache and d-cache:

32 KB each
i 4 way, d 8-way,
Access: 4 cycles

L2 unified cache:

256 KB, 4-way,
Access: 11 cycles

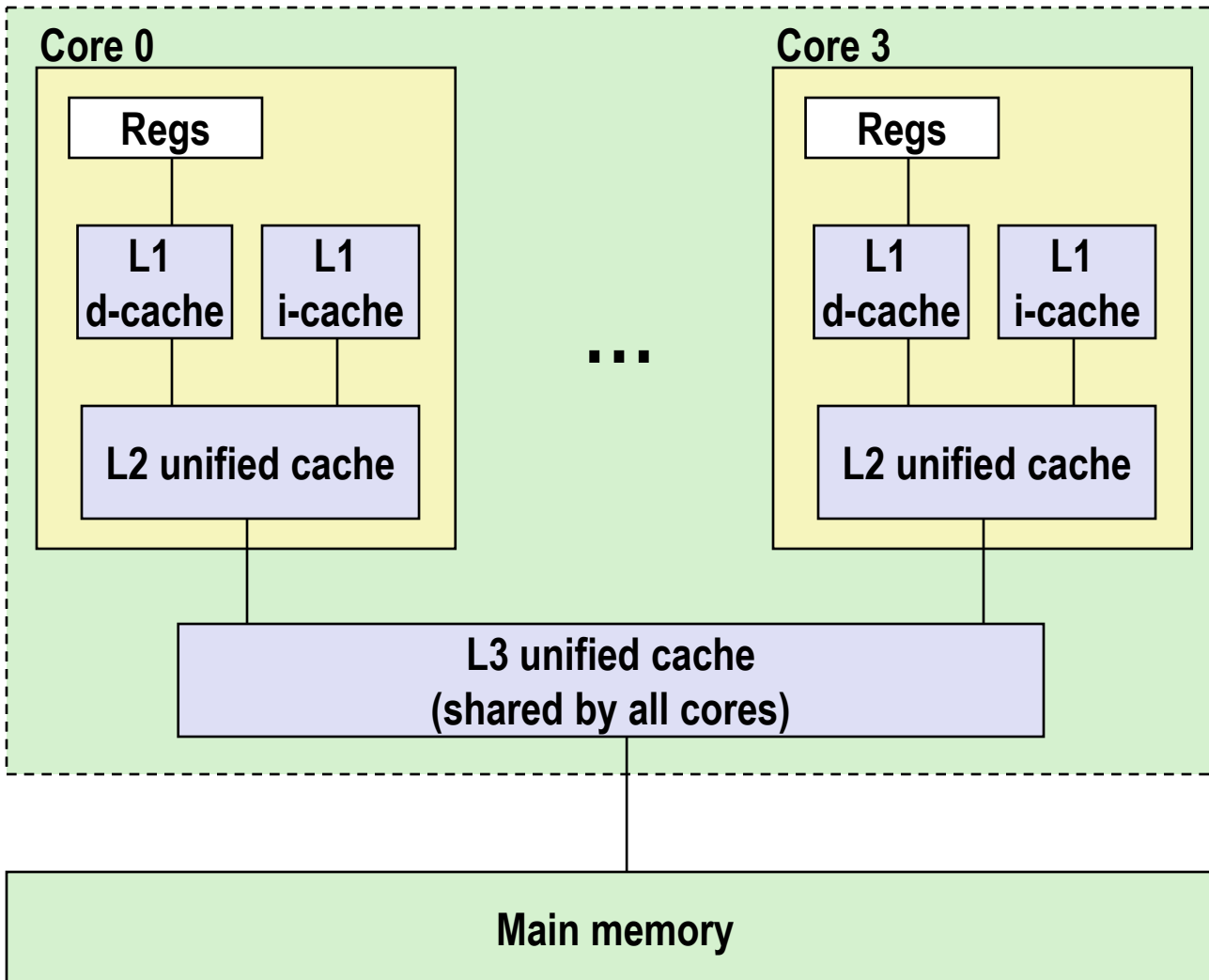
L3 unified cache:

8 MB, 16-way,
Access: 30-40 cycles

Block size: 64 bytes for
all caches.

Intel Core i7 Cache Hierarchy, 11 Gen, '20

Processor package



L1 i-cache and d-cache:

32 KB i, 48 KB d
i 8 way, d 12-way,
Access: 4 cycles

L2 unified cache:

1.25 MB, 20-way,
Access: 11 cycles

L3 unified cache:

12 MB, 12-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)
= $1 - \text{hit rate}$
- 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:
 - 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!)

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.

Concluding Observations

- **Programmer can optimize for cache performance**
 - How data structures are organized
 - How data are accessed
 - Nested loop structure
 - Blocking is a general technique
- **All systems favor “cache friendly code”**
 - Getting absolute optimum performance is very platform specific
 - Cache sizes, line sizes, associativities, etc.
 - Can get most of the advantage with generic code
 - Keep working set reasonably small (temporal locality)
 - Use small strides (spatial locality)