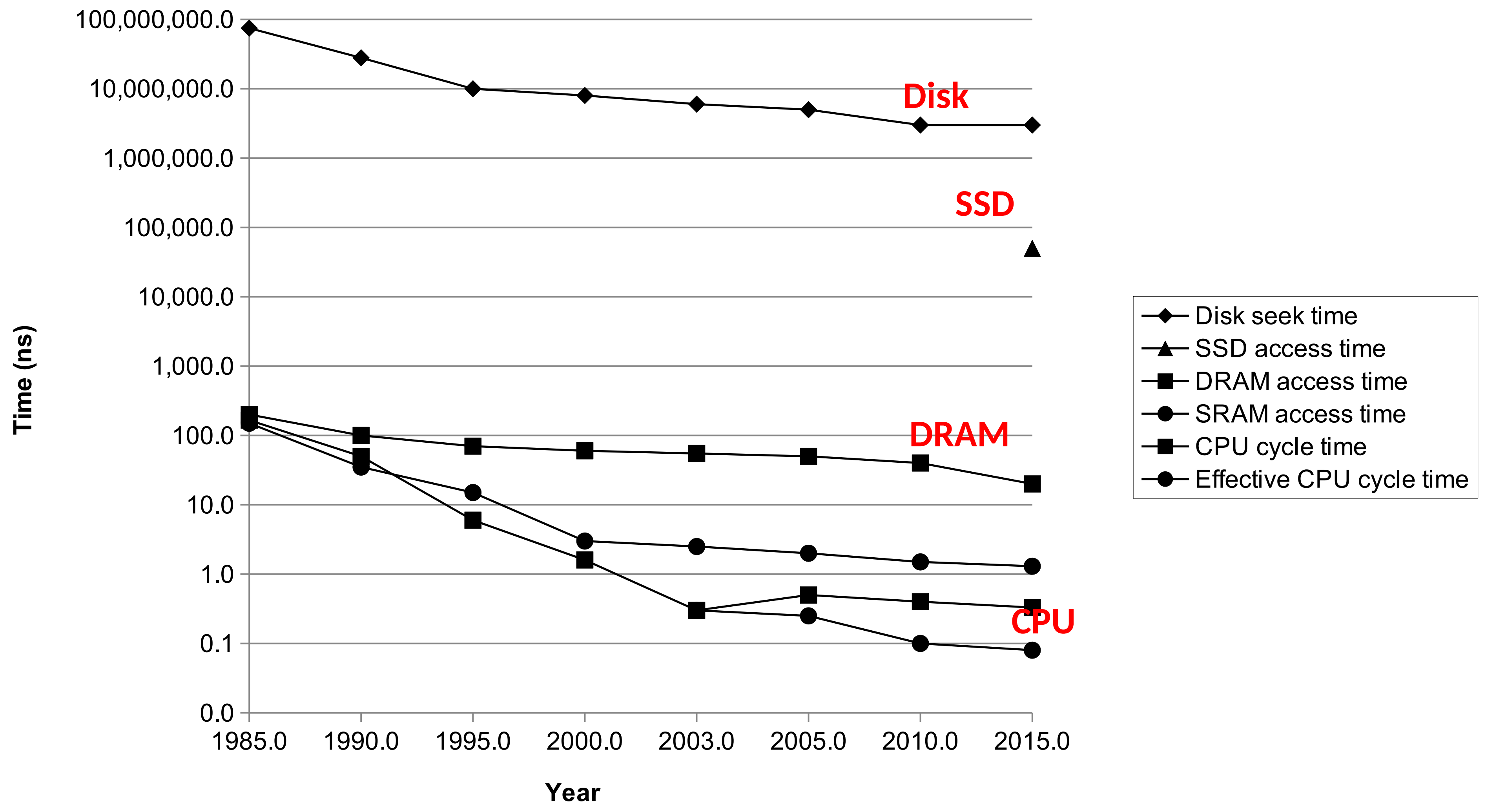
Computer Organization: A Programmer's Perspective

The Memory Hierarchy

Gal A. Kaminka
galk@cs.biu.ac.il

The CPU-Memory Gap

The gap widens between DRAM, disk, and CPU speeds.



<u>Key Question:</u>
How to have fast, cheap memory?

Principle of Locality

Programs tend to reuse data and instructions:

- near those they have used recently
- Or same as those they have used recently

  - Temporal locality: Recently referenced items are likely to be referenced in the near future.

  - Spatial locality: Items with nearby addresses tend to be referenced close together in time.

Example

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

Locality Example:

Data

Reference array elements in succession (stride-1 reference pattern): **Spatial locality**

Reference `sum` each iteration: **Temporal locality**

Instructions

Reference instructions in sequence: **Spatial locality**

Cycle through loop repeatedly: **Temporal locality**

Locality Qualitative Estimation

Question: Does this function have good locality?

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

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

Locality Example

Question: Does this function have good locality?

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

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

Important Skill for Professional Programmer:
Be able to look at code, get a qualitative sense of its locality

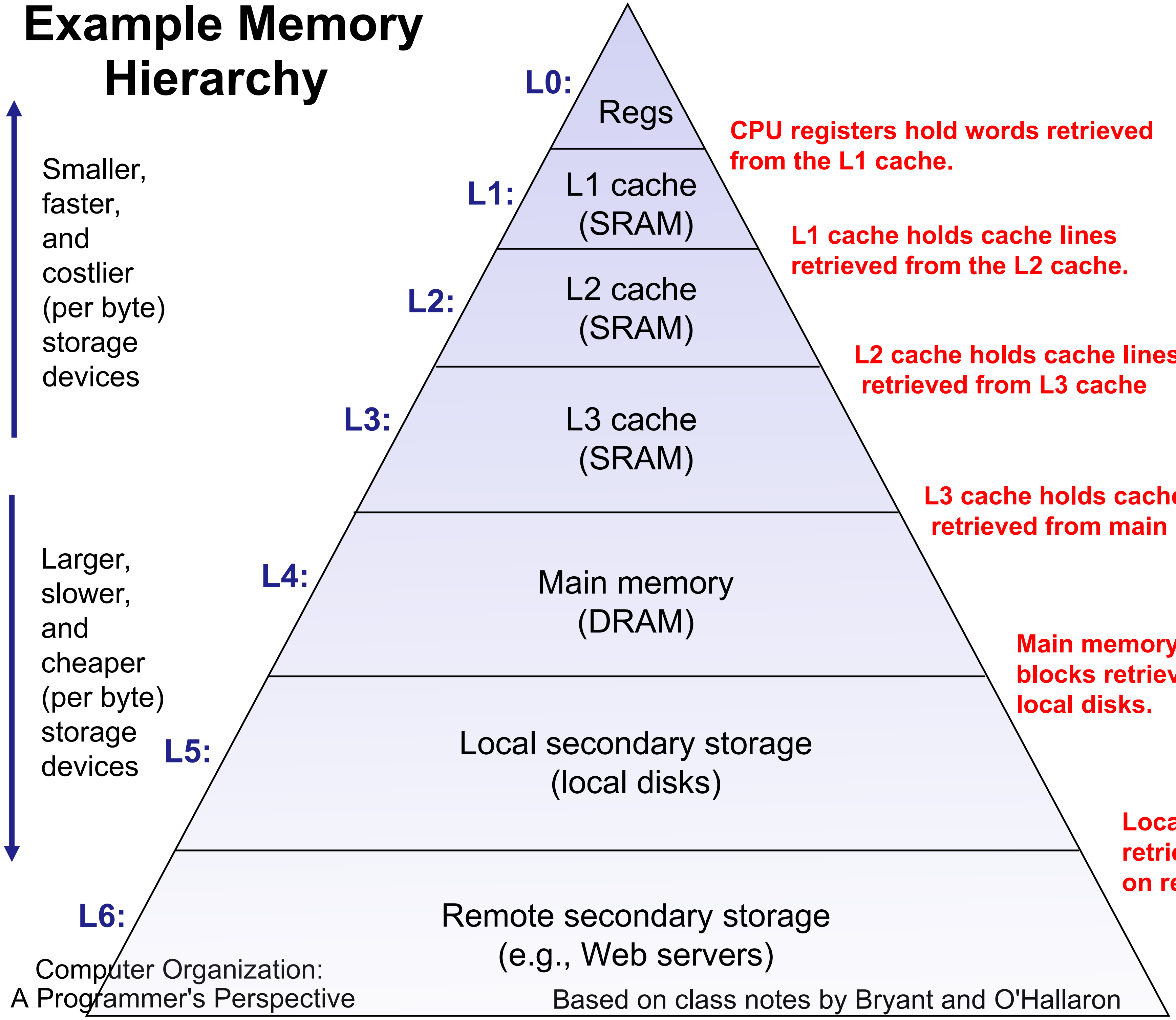
Memory Hierarchies

- Some fundamental and enduring properties of hardware and software:
  - Fast storage technologies cost more per byte and have less capacity.
  - The gap between CPU and main memory speed is widening.
  - Well-written programs tend to exhibit good locality.

- These properties complement each other beautifully.

- They suggest an approach for organizing memory and storage systems known as a memory hierarchy.

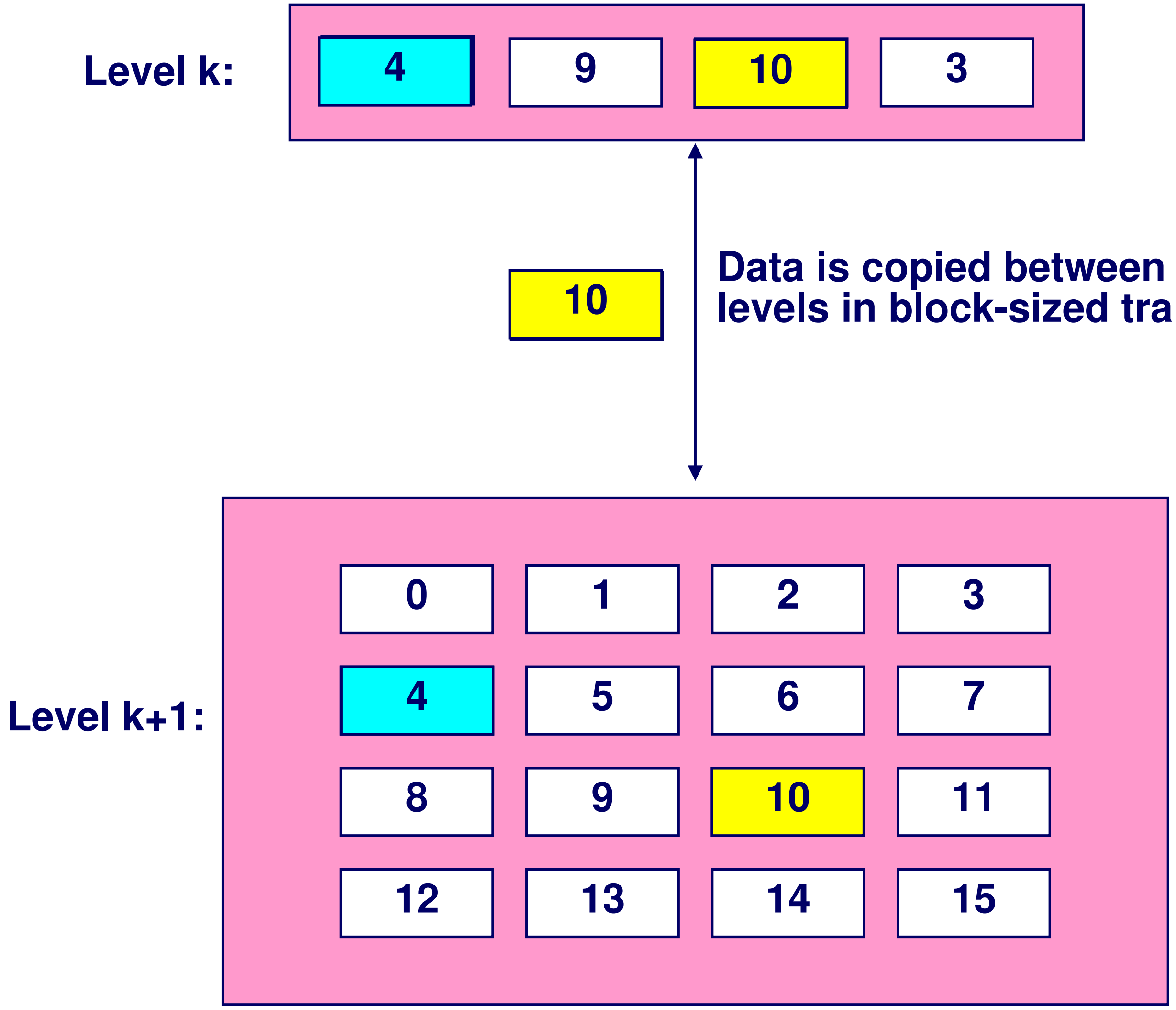
Example Memory Hierarchy


Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
Regs
L1:
L1 cache (SRAM)
L2:
L2 cache (SRAM)
L3:
L3 cache (SRAM)
L4:
Main memory (DRAM)
L5:
Local secondary storage (local disks)
L6:
Remote secondary storage (e.g., Web servers)
CPU registers hold words retrieved from the L1 cache.
L1 cache holds cache lines retrieved from the L2 cache.
L2 cache holds cache lines retrieved from L3 cache
L3 cache holds cache lines retrieved from main memory.
Main memory holds disk blocks retrieved from local disks.
Local disks hold files retrieved from disks on remote servers

Caches

- Cache: A smaller, faster storage 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?
  - 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.
  - Net effect: A large pool of memory 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.

Caching in a Memory Hierarchy

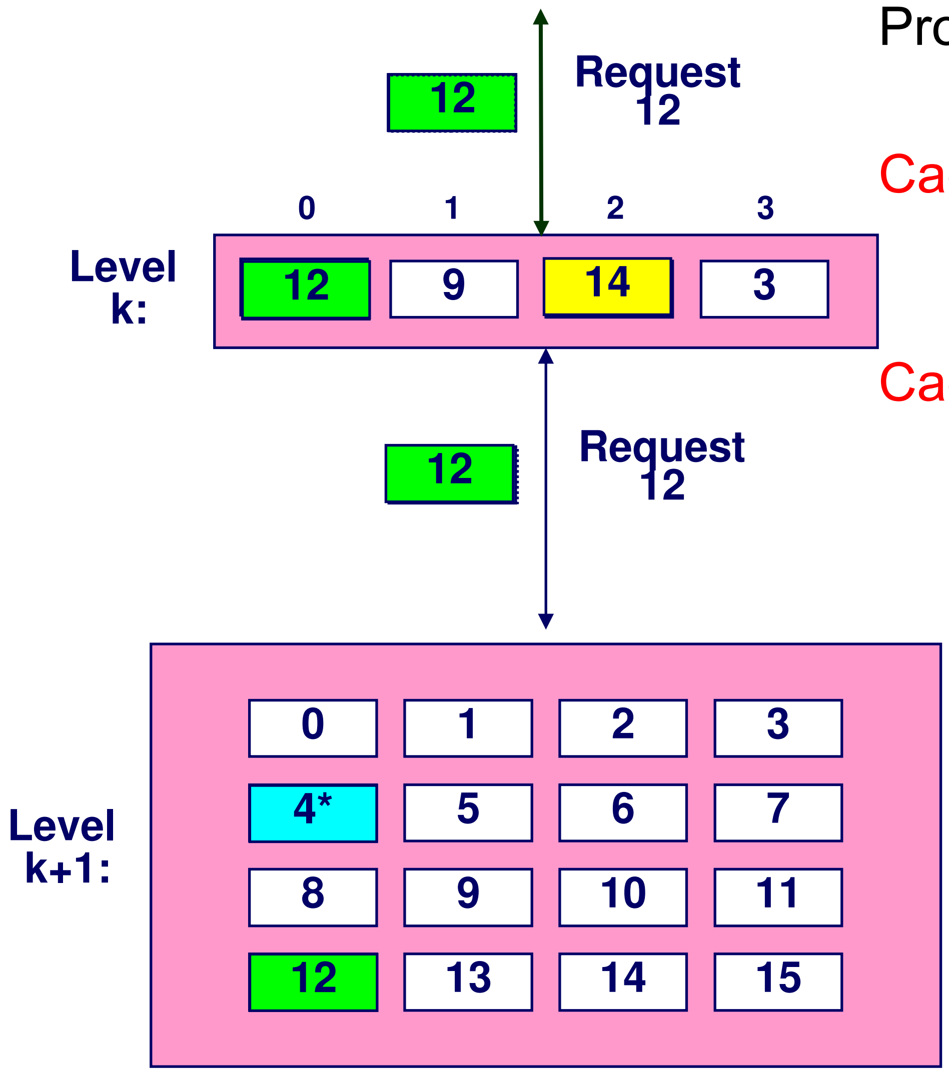


Smaller, faster, more expensive device at level k caches a subset of the blocks from level k+1

Data is copied between levels in block-sized transfer units

Larger, slower, cheaper storage device at level k+1 is partitioned into blocks.

General Caching Concepts



Program needs object d, which is stored in some block b.

Cache hit

- Program finds b in the cache at level k. E.g., block 14.

Cache miss

- b is not at level k, so level k cache must fetch it from level k+1. E.g., block 12.
- If level k cache is full, then some current block must be replaced (evicted). Which one is the "victim"?
  - Placement policy: where can the new block go? E.g., b mod 4
  - Replacement policy: which block should be evicted? E.g., LRU

General Caching Concepts

Types of cache misses:

- Cold (compulsary) 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 is placed in block (i mod 4) at level k+1.
  - 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.

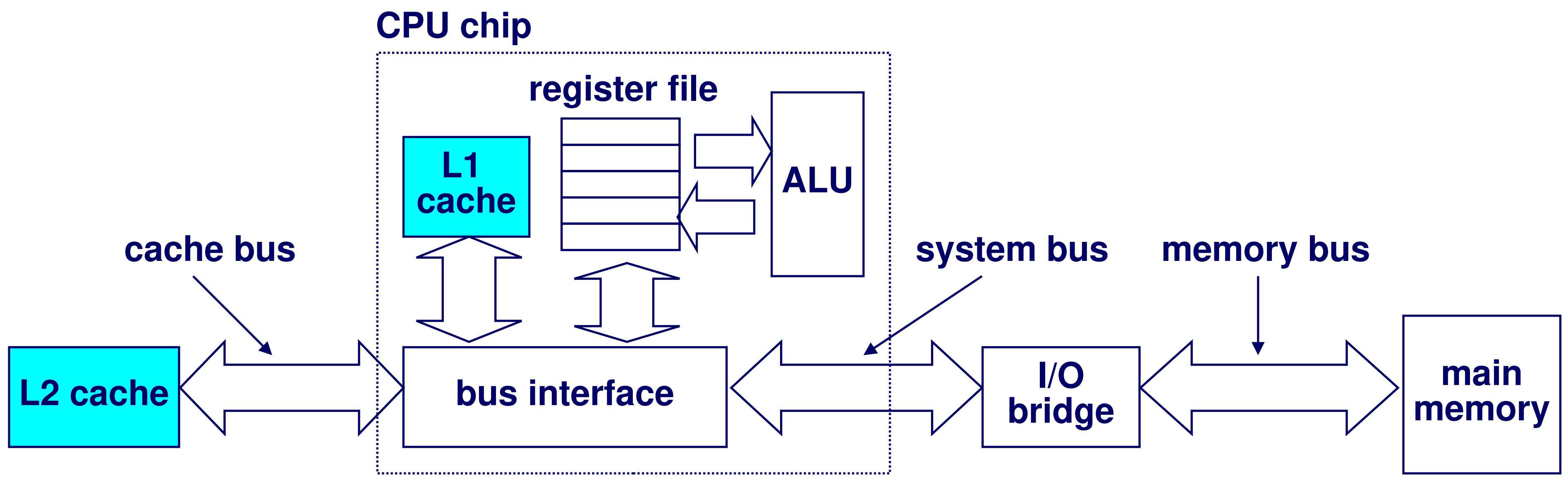
Cache Memories

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 L1, then in L2, then in main memory.

Typical bus structure:

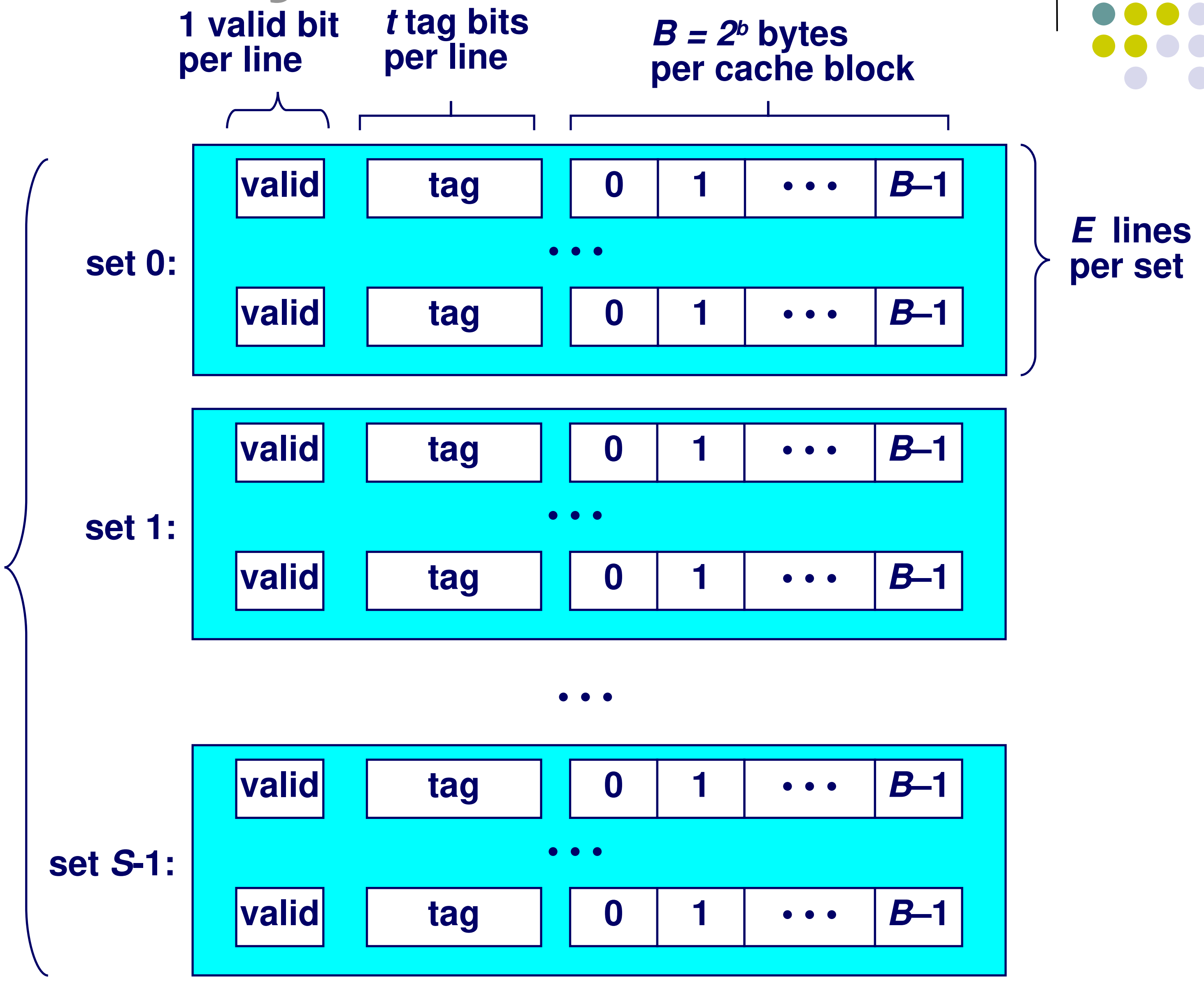


General Organization of a Cache Memory

Cache is an array of sets.

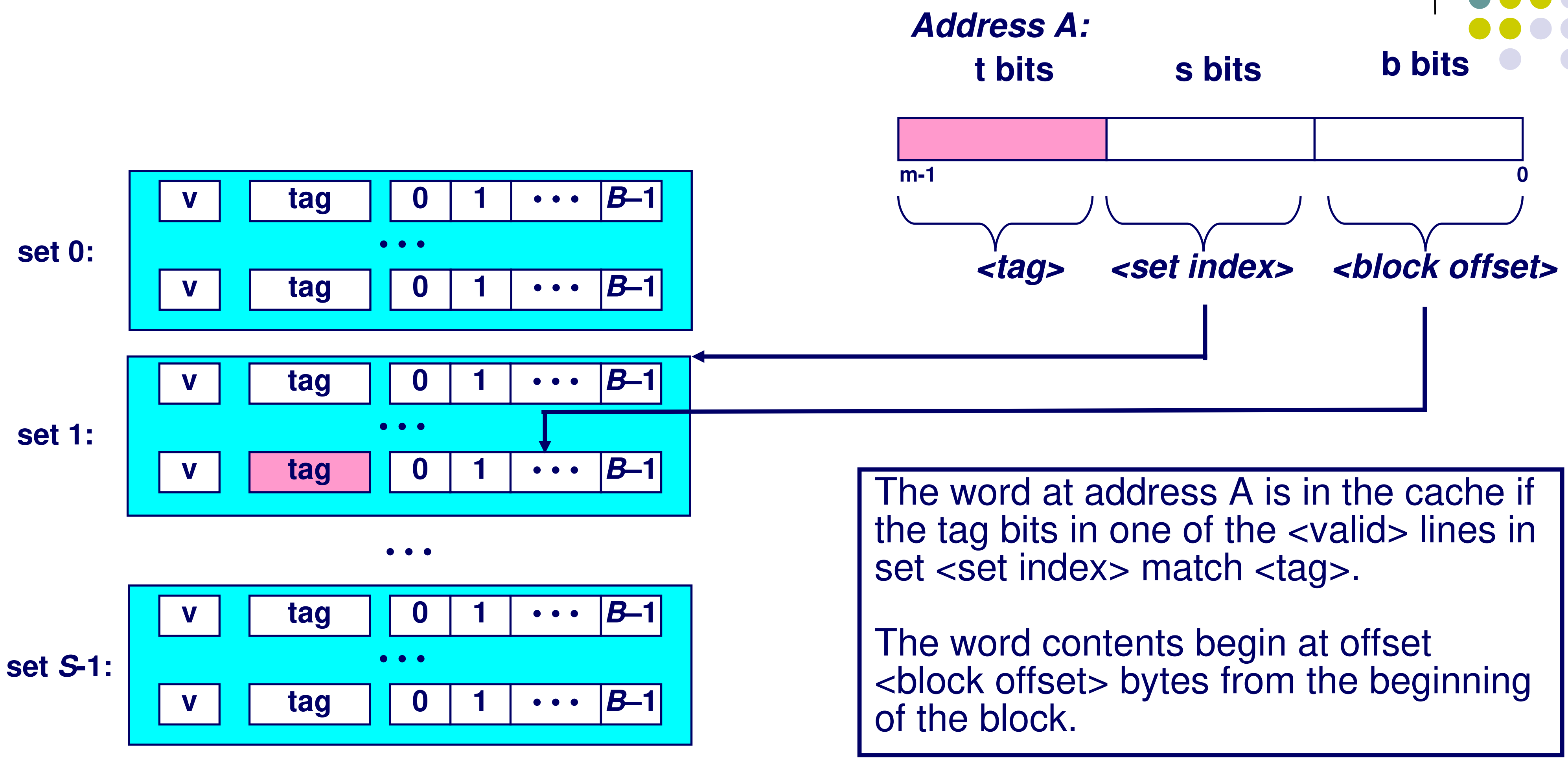
Each set contains one or more lines.

Each line holds a block of data.



Cache size: *C = B x E x S* data bytes

Addressing Caches



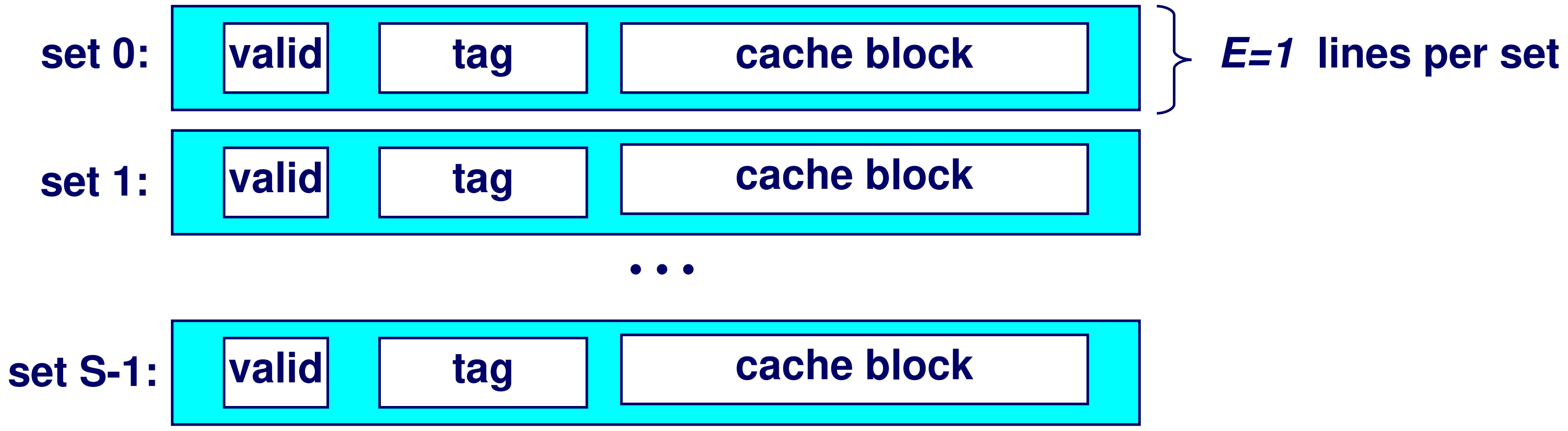
The word at address A is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>.

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

Direct-Mapped Cache

Simplest kind of cache

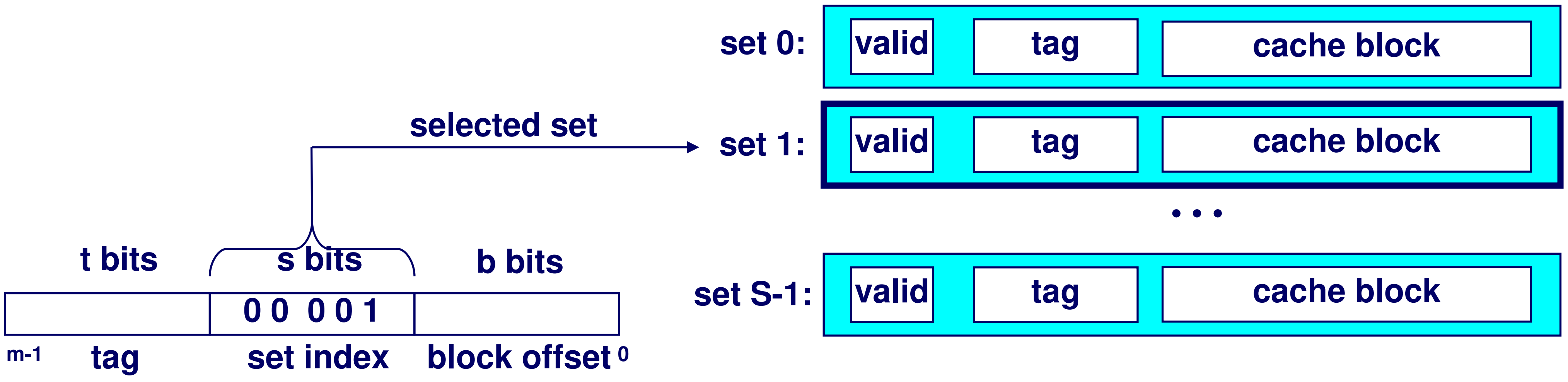
Characterized by exactly one line per set (E=1).



Accessing Direct-Mapped Caches

Set selection

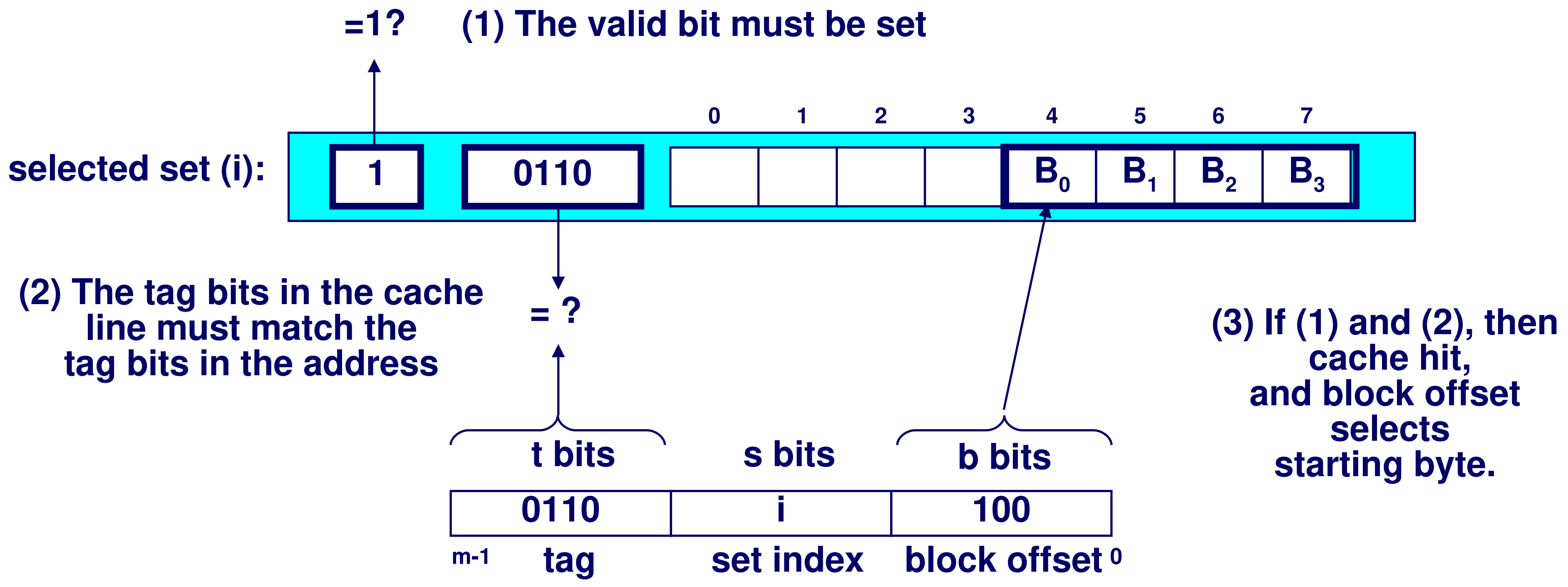
Use the set index bits to determine the set of interest.



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 (here 32bits made from 4 bytes $B_0..B_3$)



=1?
(1) The valid bit must be set
0
1
2
3
4
5
6
7
selected set (i):
1
0110
B_0
B_1
B_2
B_3
(2) The tag bits in the cache line must match the tag bits in the address
= ?
(3) If (1) and (2), then cache hit, and block offset selects starting byte.
t bits
s bits
b bits
0110
i
100
m-1
tag
set index
block offset
0

Direct-Mapped Cache Simulation

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

Address trace (reads):
0 [0000_2], 1 [0001_2], 13 [1101_2], 8 [1000_2], 0 [0000_2]

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

0 [0000_2] *(cold miss)*

(1)

v	tag	data

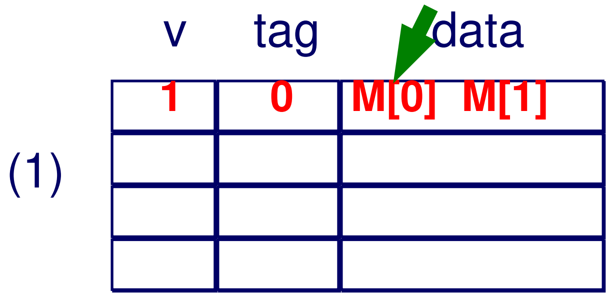
Direct-Mapped Cache Simulation

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

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

Address trace (reads):
0 [0000_2], 1 [0001_2], 13 [1101_2], 8 [1000_2], 0 [0000_2]

0 [0000_2] *(cold miss)*



v	tag	data
1	0	M[0] M[1]

Direct-Mapped Cache Simulation

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

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

Address trace (reads):
0 [0000_2], 1 [0001_2], 13 [1101_2], 8 [1000_2], 0 [0000_2]

1 [0001_2] ***(hit!)***

(2)

v	tag	data
1	**0**	**M[0] M[1]**

Direct-Mapped Cache Simulation

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

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

Address trace (reads):
0 [0000_2], 1 [0001_2], 13 [1101_2], 8 [1000_2], 0 [0000_2]

(1)

v	tag	data
1	0	M[0] M[1]

(3) 13 [1101_2] *(cold miss)*

v	tag	data
1	0	M[0] M[1]
1	1	M[12] M[13]

Direct-Mapped Cache Simulation

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

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

Address trace (reads):
0 [0000_2], 1 [0001_2], 13 [1101_2], 8 [1000_2], 0 [0000_2]

(1)

v	tag	data
1	0	M[0] M[1]

13 [1101_2] *(cold miss)*

(3)

v	tag	data
1	0	M[0] M[1]
1	1	M[12] M[13]

8 [1000_2] *(conflict miss)*

(4)

v	tag	data
1	1	M[8] M[9]
1	1	M[12] M[13]

Direct-Mapped Cache Simulation

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

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

Address trace (reads):
0 [0000_2], 1 [0001_2], 13 [1101_2], 8 [1000_2], 0 [0000_2]

(1)

v	tag	data
1	0	M[0] M[1]

(3) 13 [1101_2] *(miss)*

v	tag	data
1	0	M[0] M[1]
1	1	M[12] M[13]

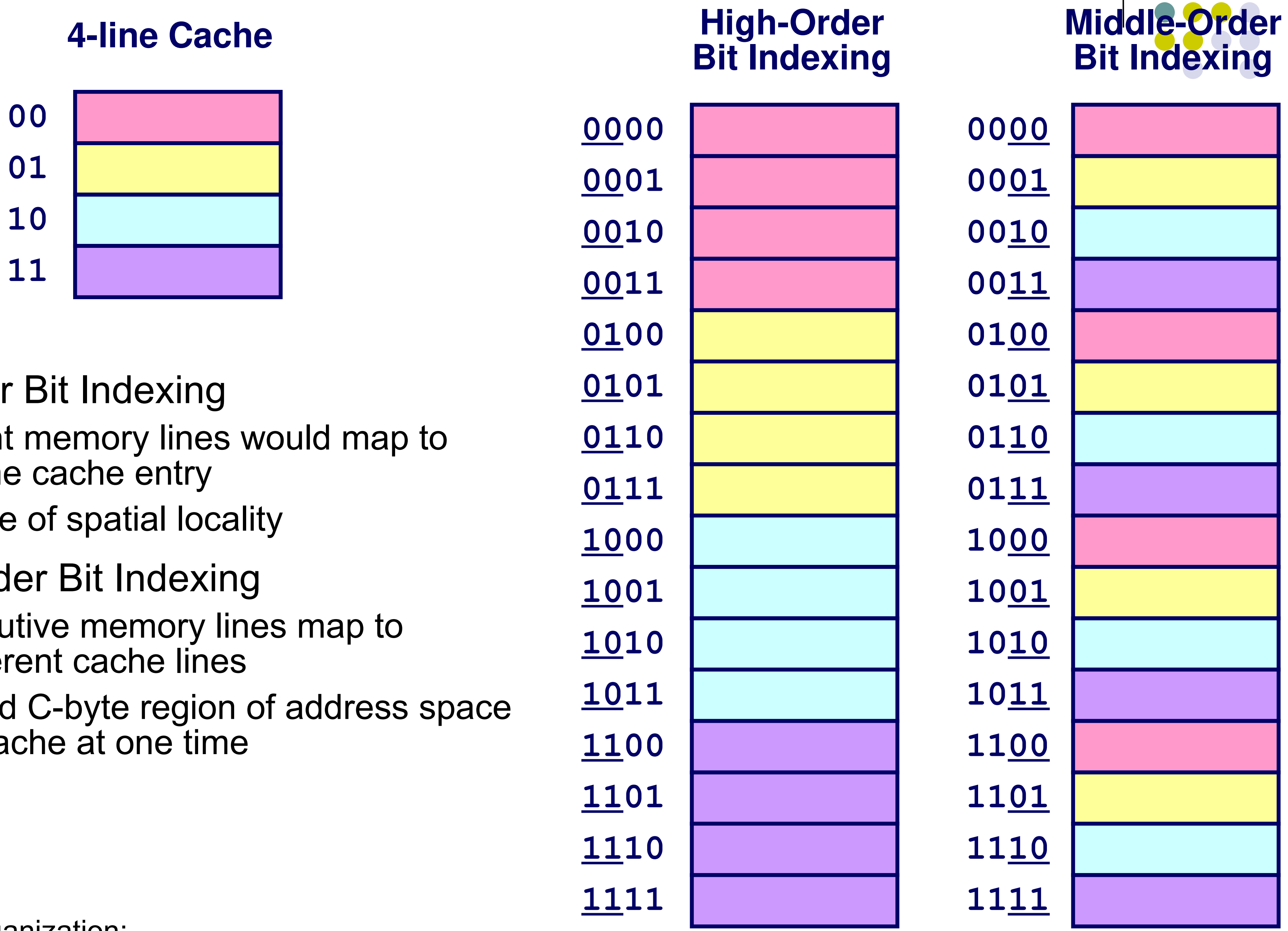
(4) 8 [1000_2] *(conflict miss)*

v	tag	data
1	1	M[8] M[9]
1	1	M[12] M[13]

(5) 0 [0000_2] *(conflict miss)*

v	tag	data
1	0	M[0] M[1]
1	1	M[12] M[13]

Why Use Middle Bits as Index?



High-Order Bit Indexing

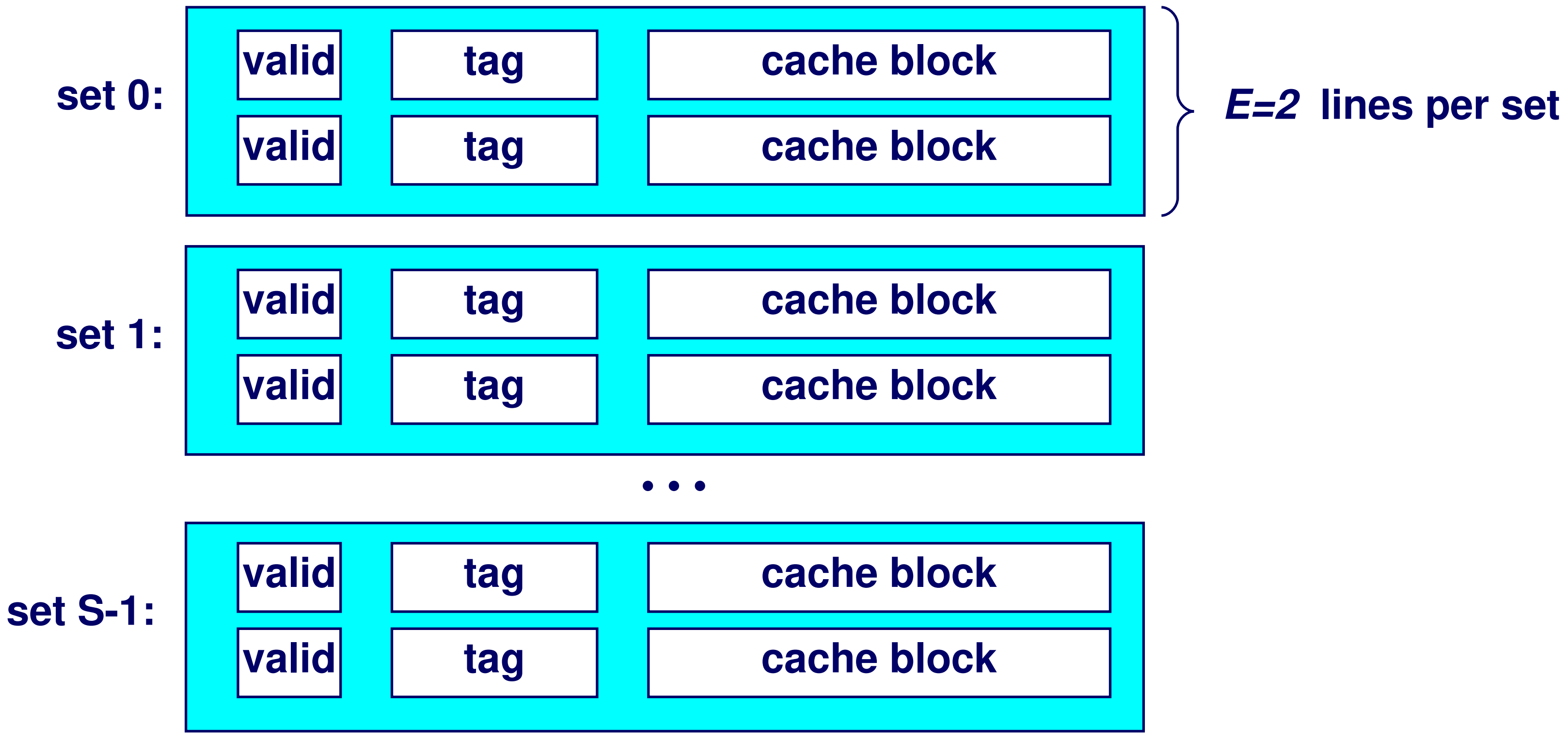
- Adjacent memory lines would map to same cache entry
- Poor use of spatial locality

Middle-Order Bit Indexing

- Consecutive memory lines map to different cache lines
- Can hold C-byte region of address space in cache at one time

Set Associative Caches

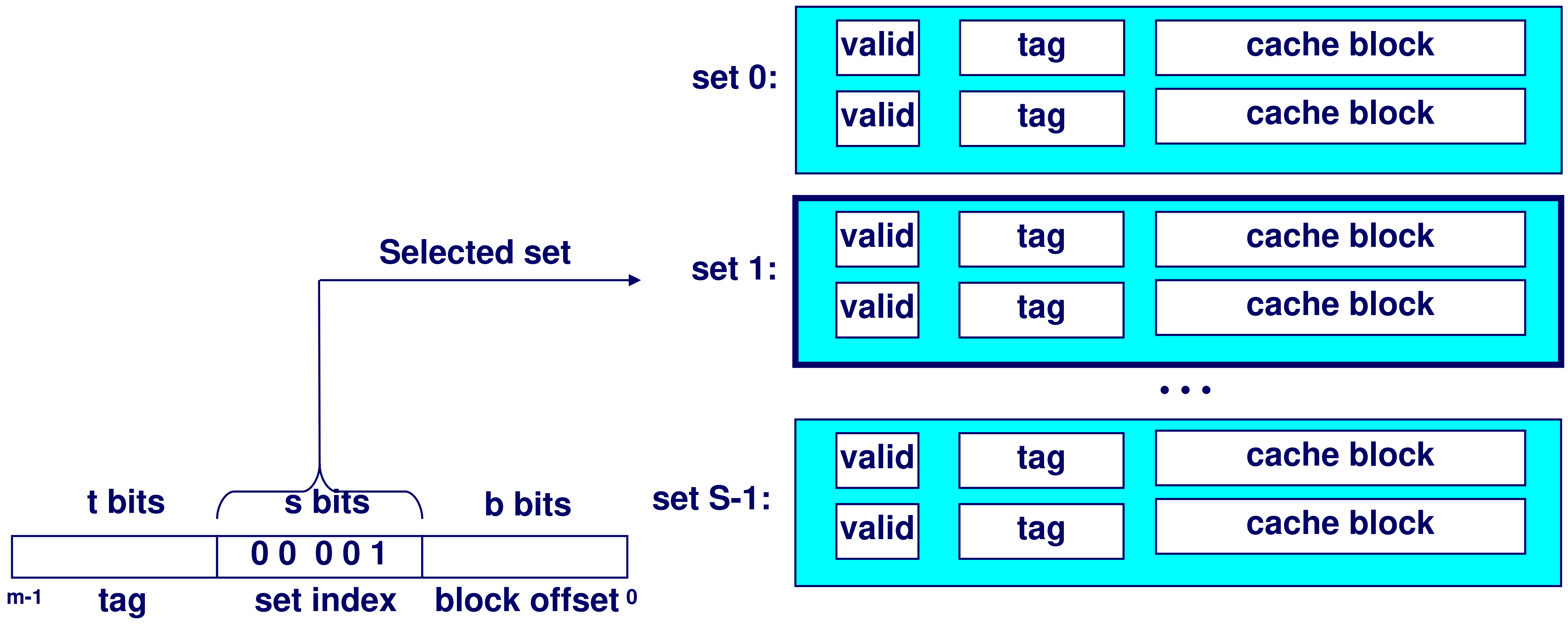
Characterized by more than one line per set



Accessing Set Associative Caches

Set selection

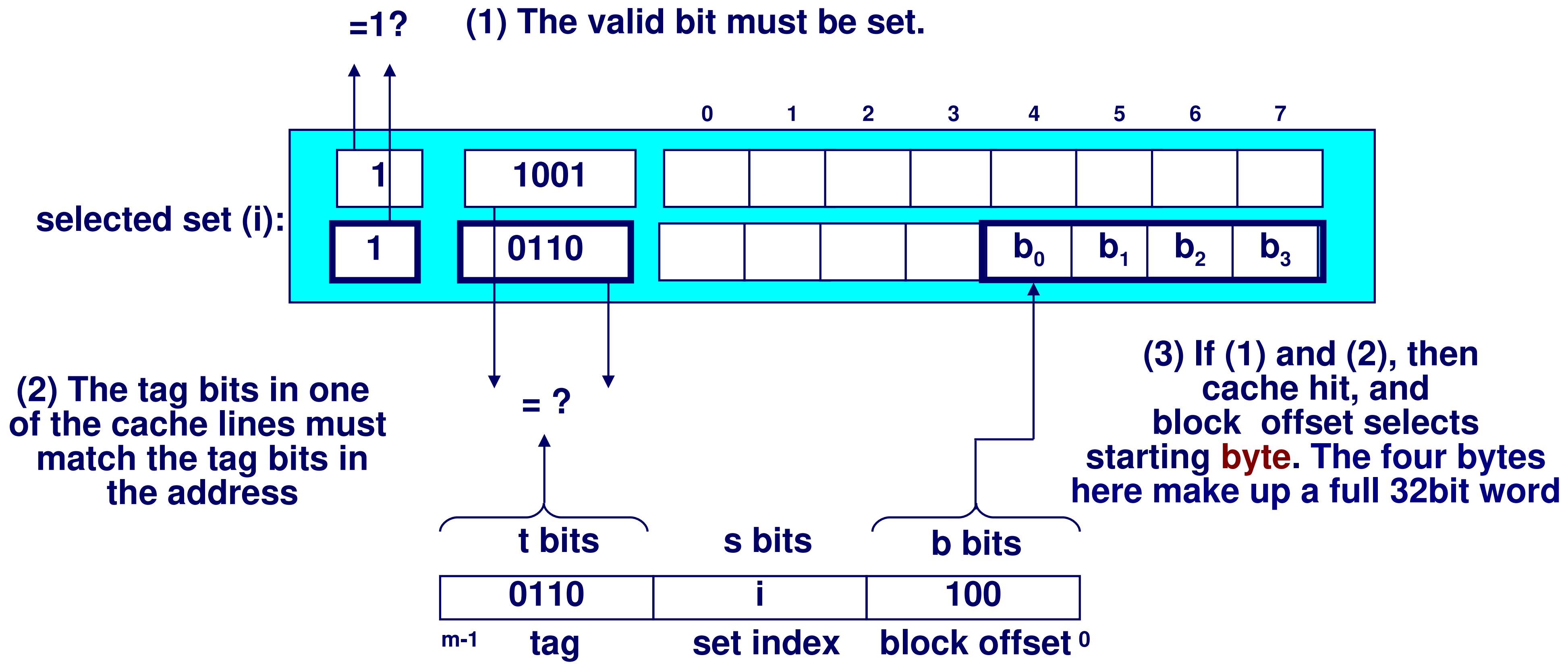
identical to direct-mapped cache



Accessing Set Associative Caches

Line matching and word selection

- must compare the tag in each valid line in the selected set.

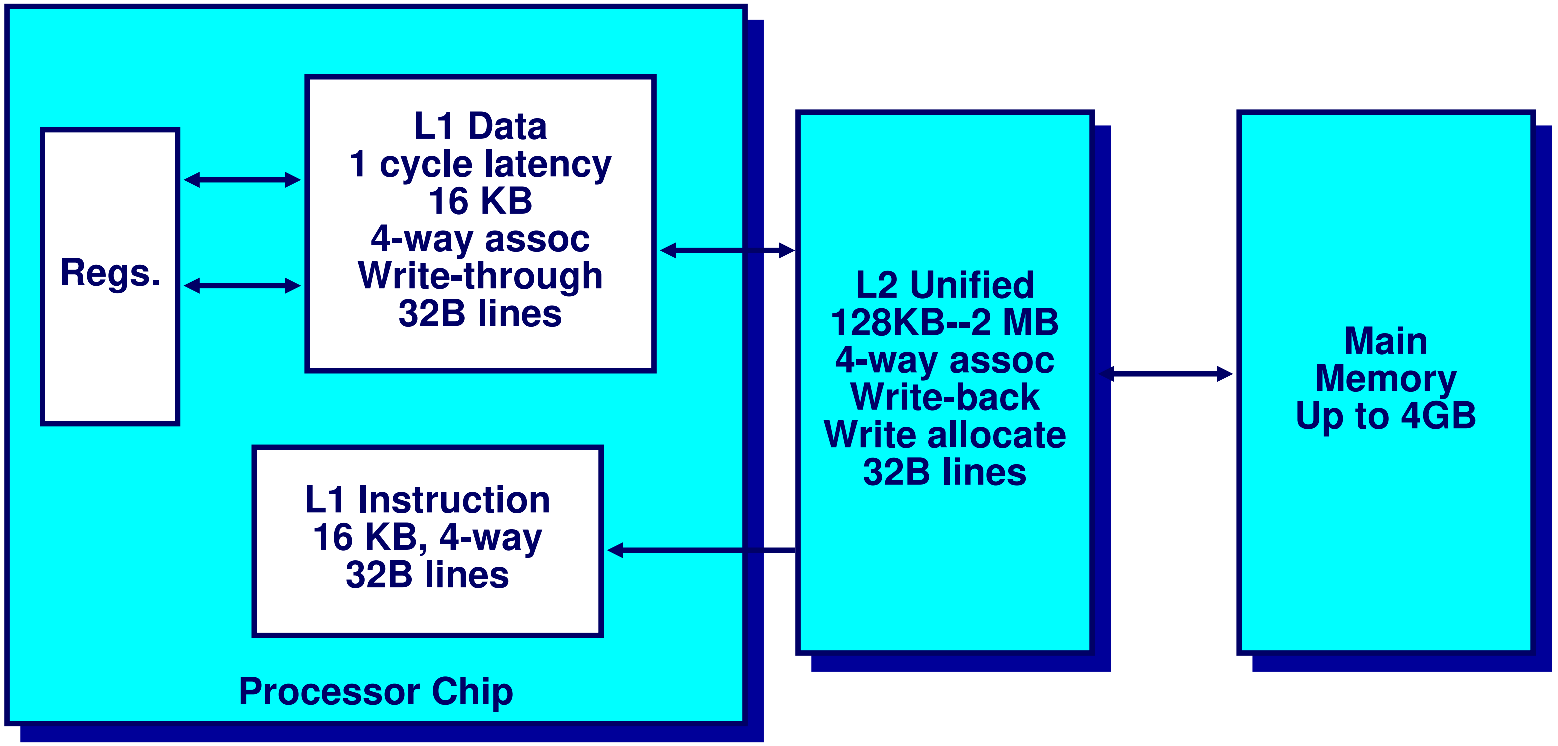


=1?
(1) The valid bit must be set.
0
1
2
3
4
5
6
7
selected set (i):
1
1001
1
0110
b_0
b_1
b_2
b_3
(2) The tag bits in one of the cache lines must match the tag bits in the address
= ?
(3) If (1) and (2), then cache hit, and block offset selects starting byte. The four bytes here make up a full 32bit word
t bits
s bits
b bits
0110
i
100
m-1
tag
set index
block offset
0

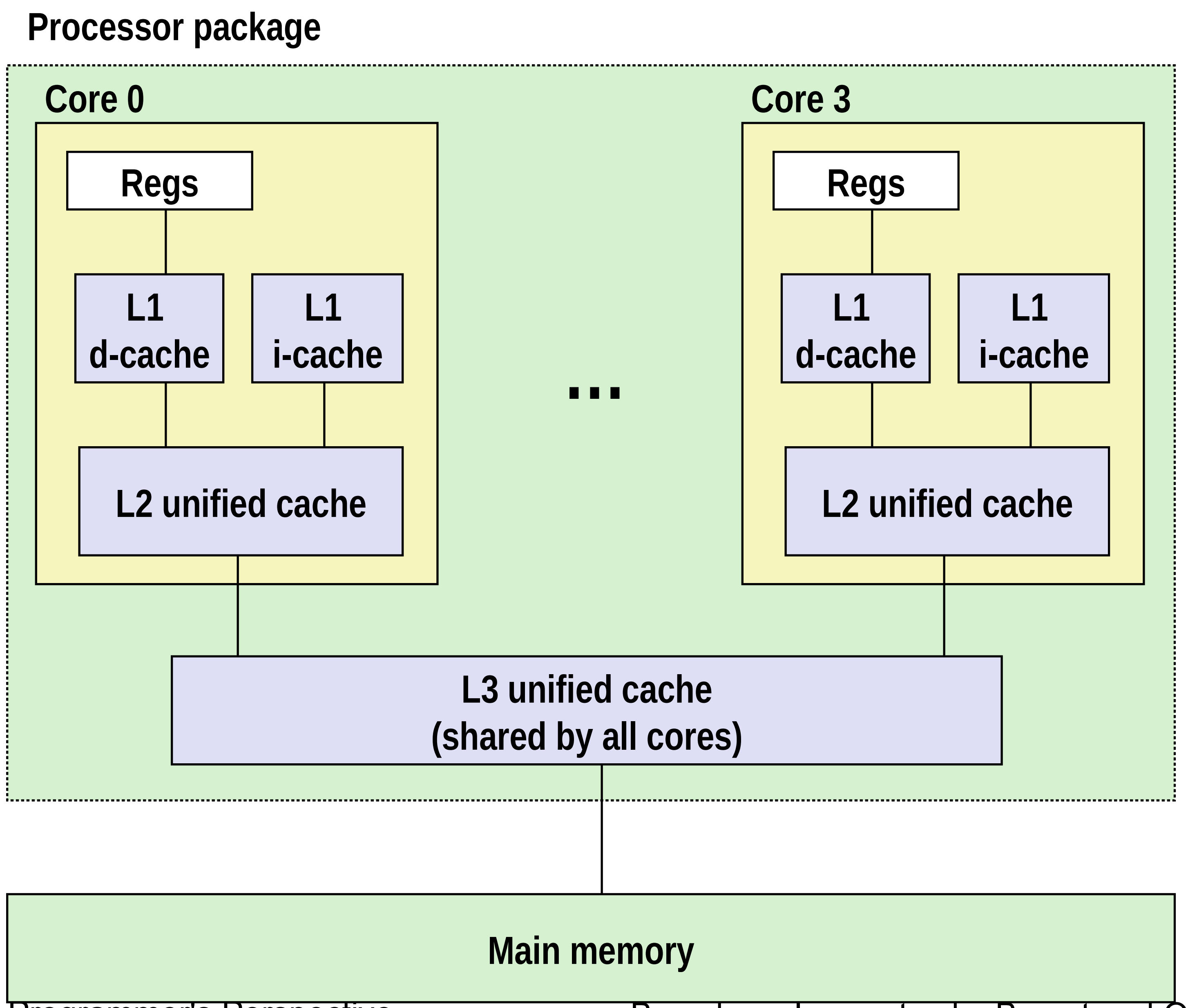
What about writes?

- **Multiple copies of data exist:**
  - L1, L2, L3, Main Memory, Disk
- **What to do on a write-hit?**
  - Write-through (write immediately to one level down)
  - Write-back (defer write to one level down 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 one level down, does not load into cache)
- **Typical**
  - Write-through + No-write-allocate
  - **Write-back + Write-allocate**

Intel Pentium Cache Hierarchy



Intel Core i7 Cache Hierarchy



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.