HARVARD
School of Engineering
and Applied Sciences

Caching

CS61, Lecture 12

Prof. Stephen Chong

October 12, 2010

Announcements

- Lab 3 design checkpoint deadline
  - Meet with your TF by Thursday 10pm
  - If you haven't already, arrange a meeting now!

Topics for today

- The Principle of Locality
- Memory Hierarchies
- Caching Concepts
- Direct-Mapped Cache Organization
- Set-Associative Cache Organization
- Multi-level caches
- Cache writes

Background: Locality

- Principle of Locality:
  - **Programs tend to reuse data and instructions "near" those they have used recently.**
- **Temporal locality**: Recently referenced memory addresses are likely to be referenced in the near future.
- **Spatial locality**: Similar memory addresses tend to be referenced close together in time.

Locality Example

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

- Locality of program data
  - Array elements are referenced in succession
  - Location `sum` is referenced on each iteration
- Locality of program code
  - Instructions within loop body executed in sequence
  - Loop is cycled over potentially many times

Locality Example

- Claim: Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

- Question: Does this function have good locality?

```
int sum_array_rows(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?

```
int sum_array_cols(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;
}
```

Locality Example

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

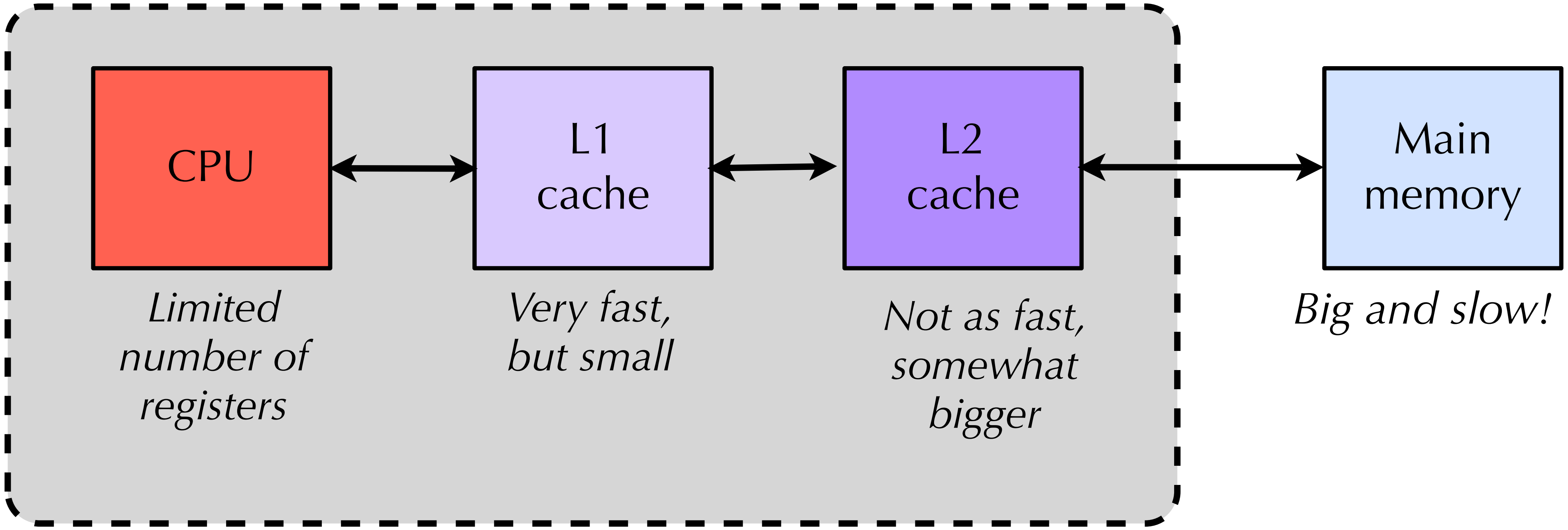
```
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

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

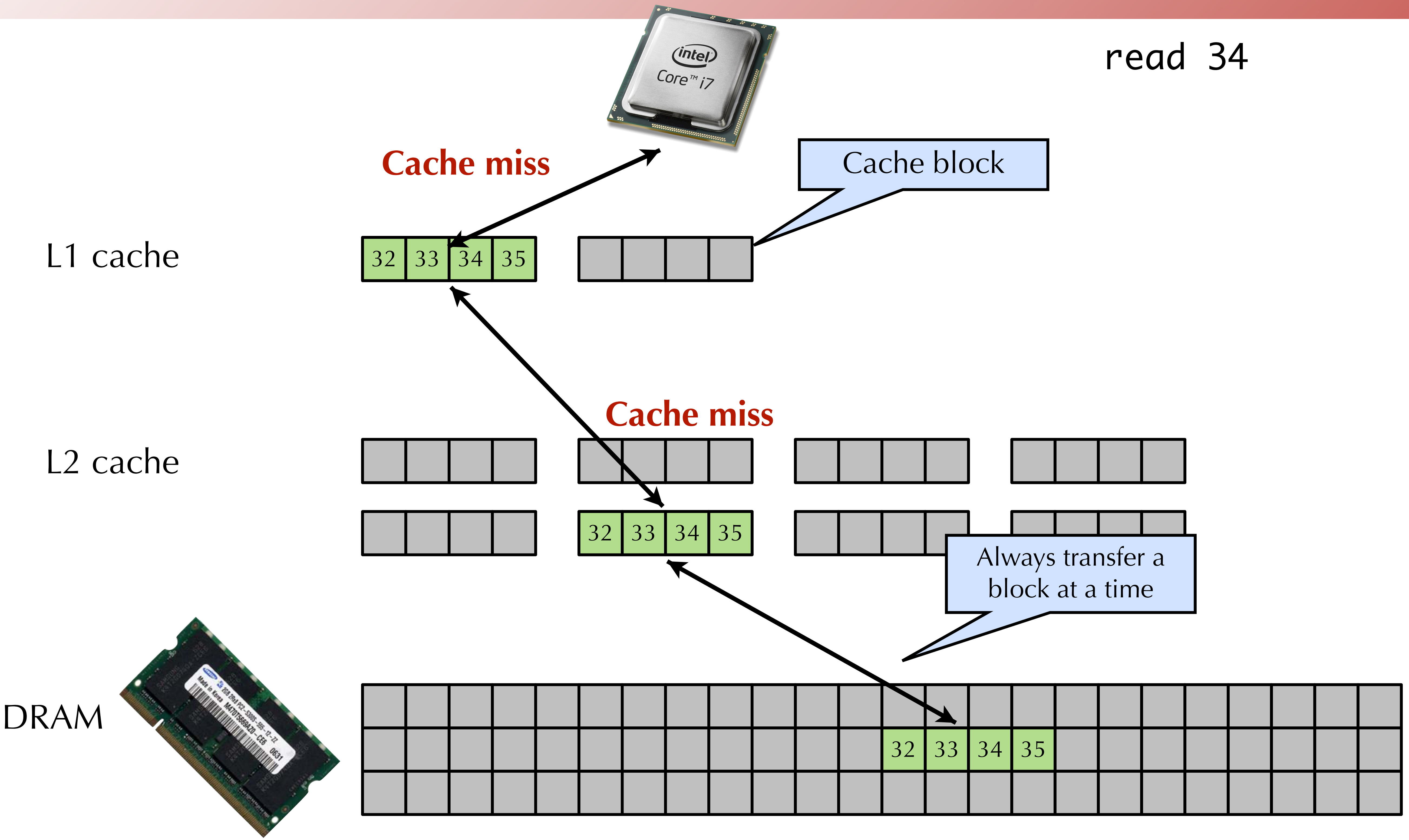
Cache Memories

- Cache memories are small, fast 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 system structure:



Processor, L1, and L2 cache are typically on a single chip or package

Caching example



```
read 34
```

Cache miss

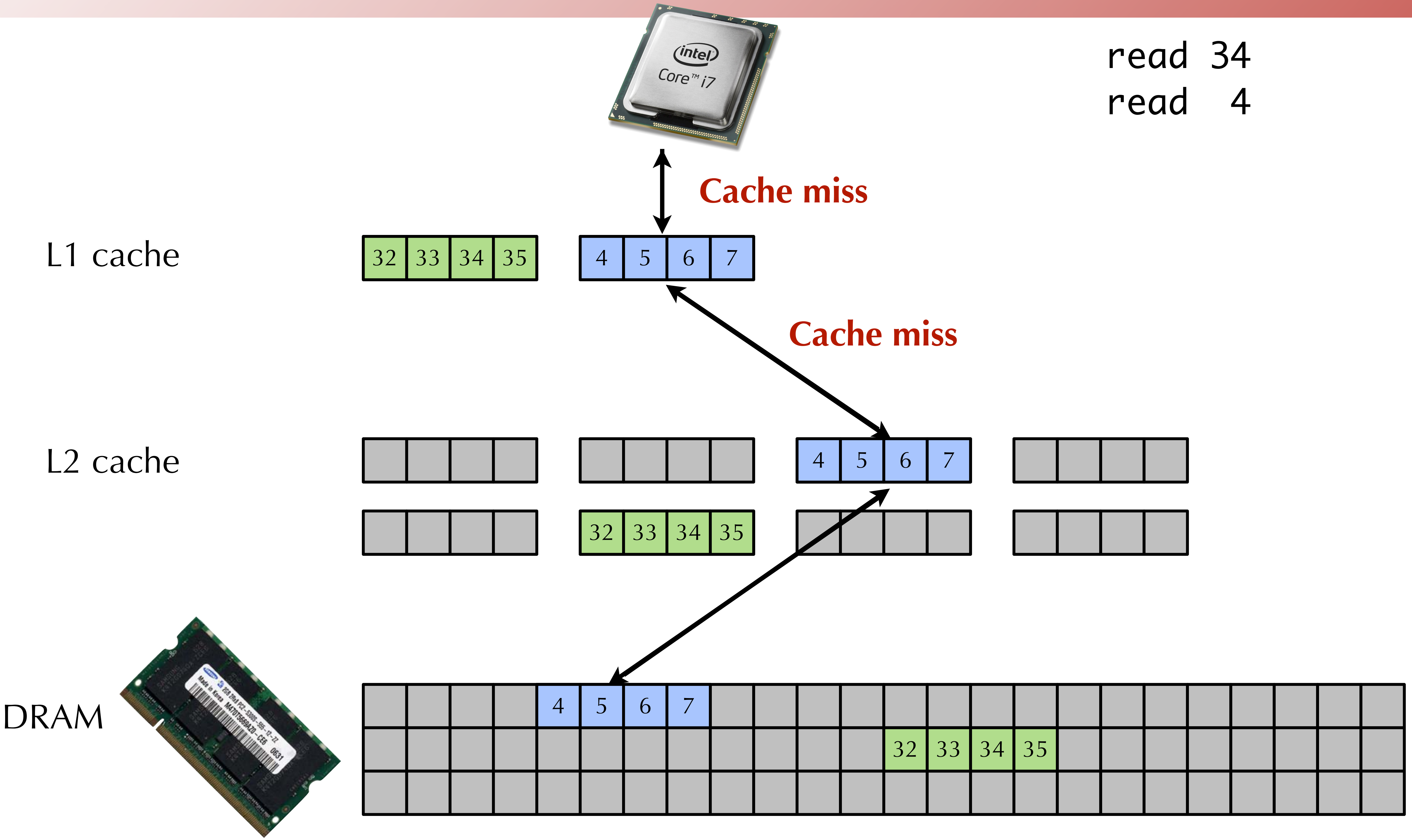
Cache block

Cache miss

Always transfer a block at a time

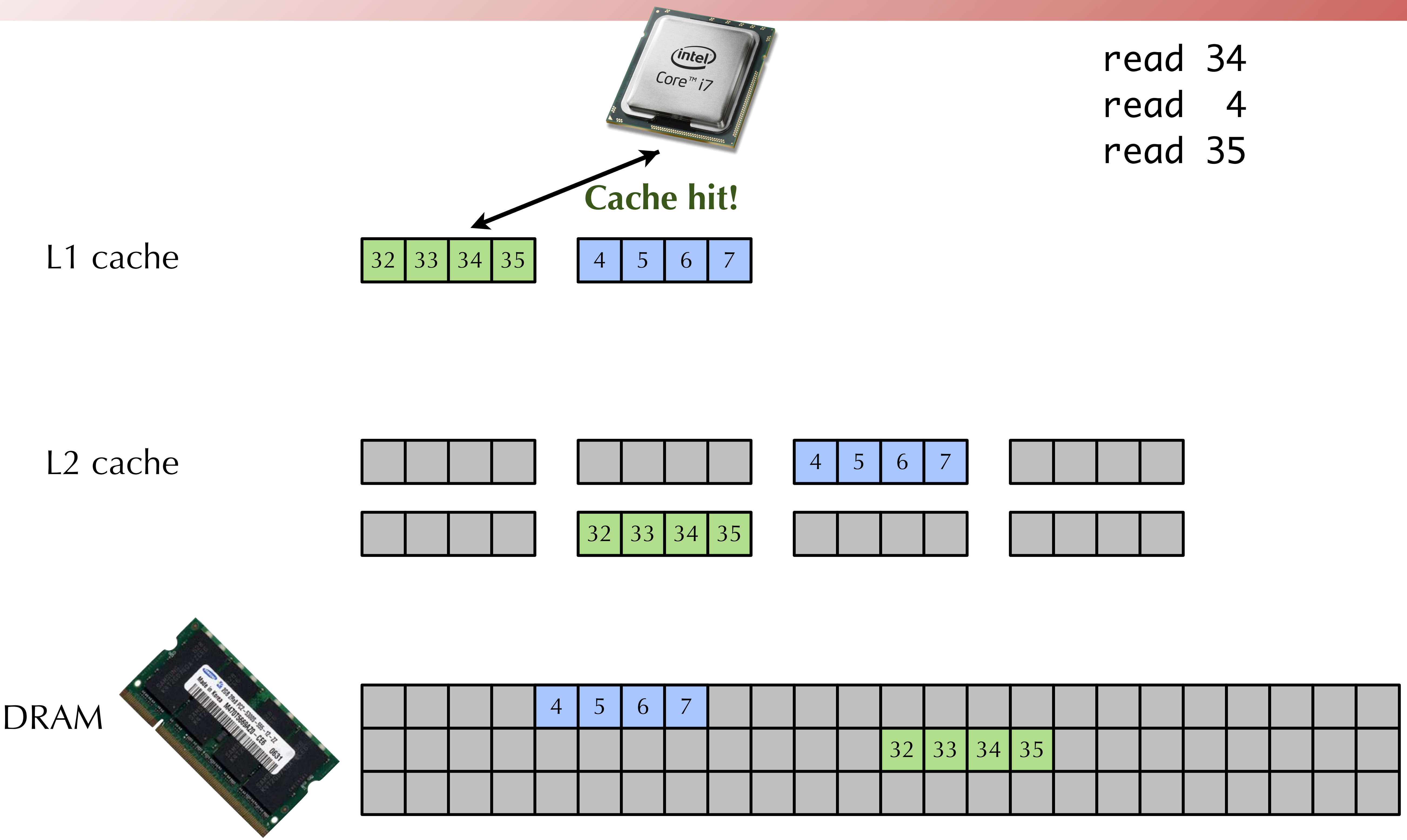
Caching example

```
read 34
read  4
```



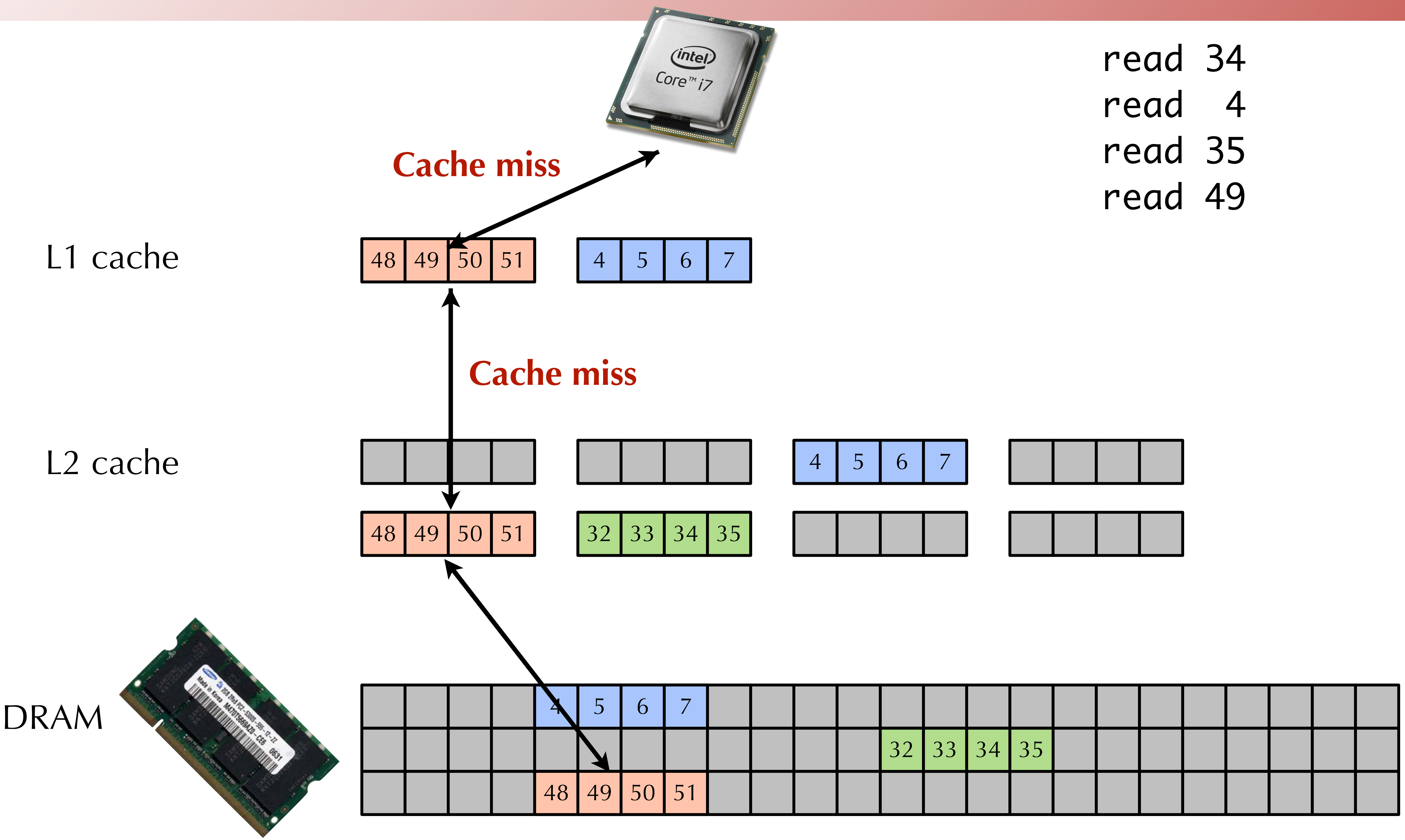
Caching example

```
read 34
read  4
read 35
```



Caching example

```
read 34
read  4
read 35
read 49
```



Cache miss

L1 cache

48 49 50 51

4 5 6 7

Cache miss

L2 cache

4 5 6 7

48 49 50 51

32 33 34 35

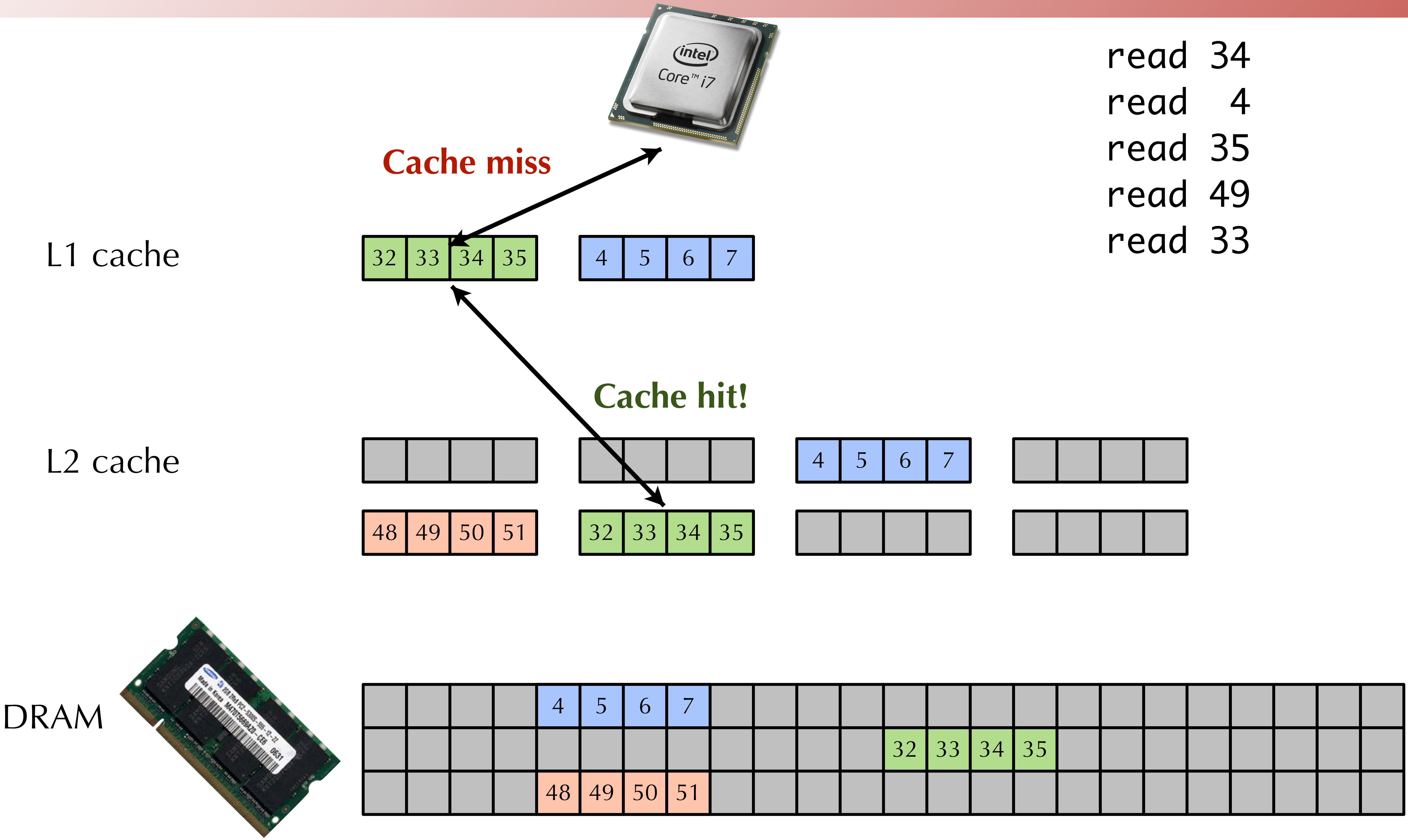
DRAM

4 5 6 7

32 33 34 35

48 49 50 51

Caching example



```
read 34
read  4
read 35
read 49
read 33
```

Caching example: access times

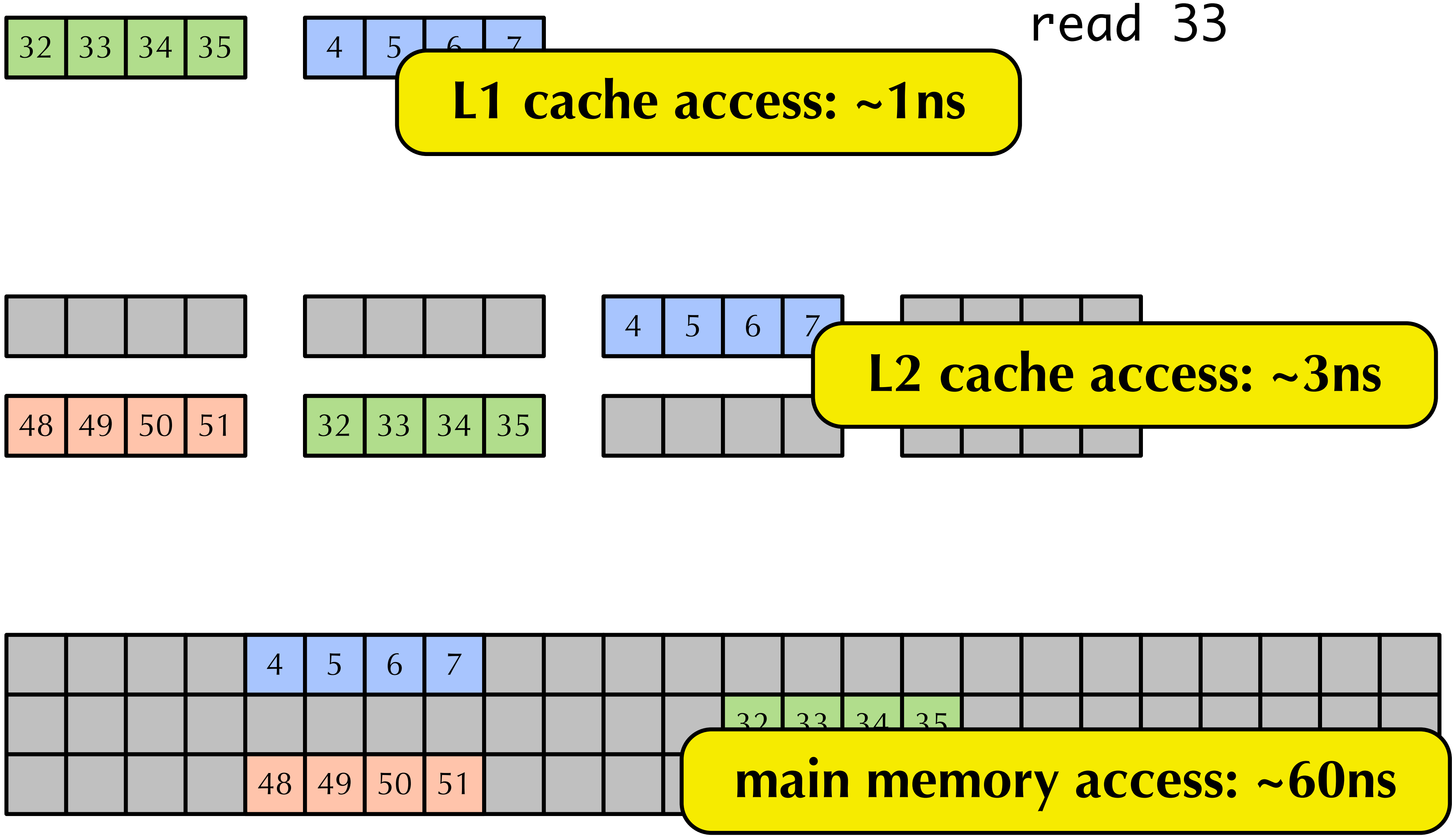
Register access: ~0.5ns

```
read 34
read  4
read 35
read 49
read 33
```

L1 cache

L2 cache

DRAM



L1 cache access: ~1ns

L2 cache access: ~3ns

main memory access: ~60ns

A few key points.

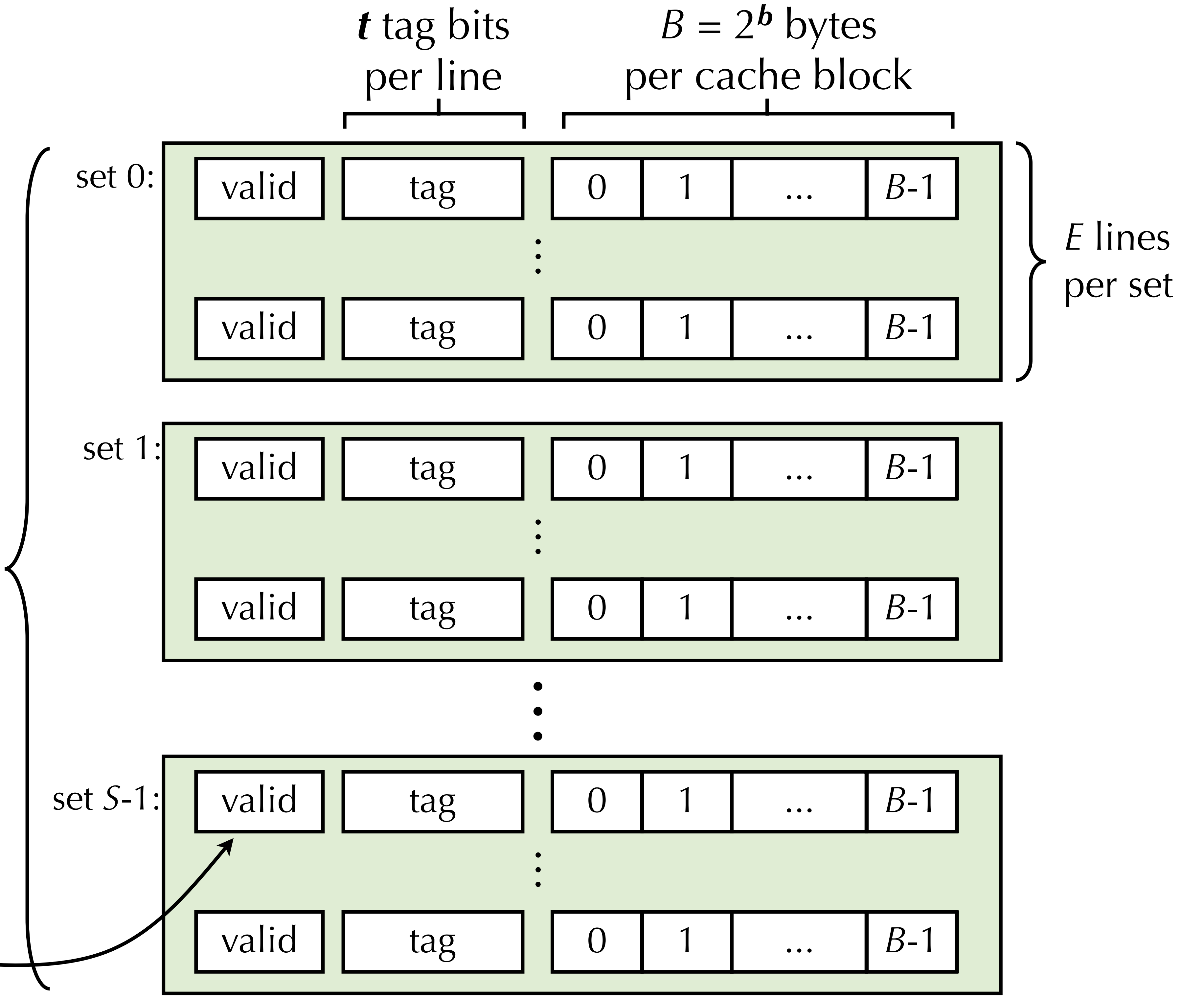
- Data transferred from memory to cache (and between cache levels) an entire **cache block** at a time.
  - Amortizes the overhead of a cache miss by getting more data from the lower level.
  - A block typically holds 32 or 64 bytes of data.
- CPU first looks in L1, then L2, then DRAM.
  - Can have a cache miss at one level and a cache hit at another!
- These examples only showed reads from memory.
  - We will talk about writes later.
- Issue #1: When CPU looks in the cache, how does it know what is stored in a given cache block?
- Issue #2: when we have to evict a cache block, which one do we evict?
  - Lots of ways of doing this.

General Organization of a Cache

Cache is an array of **sets**.

Each set contains one or more **lines**.

Each line holds a **block** of data.

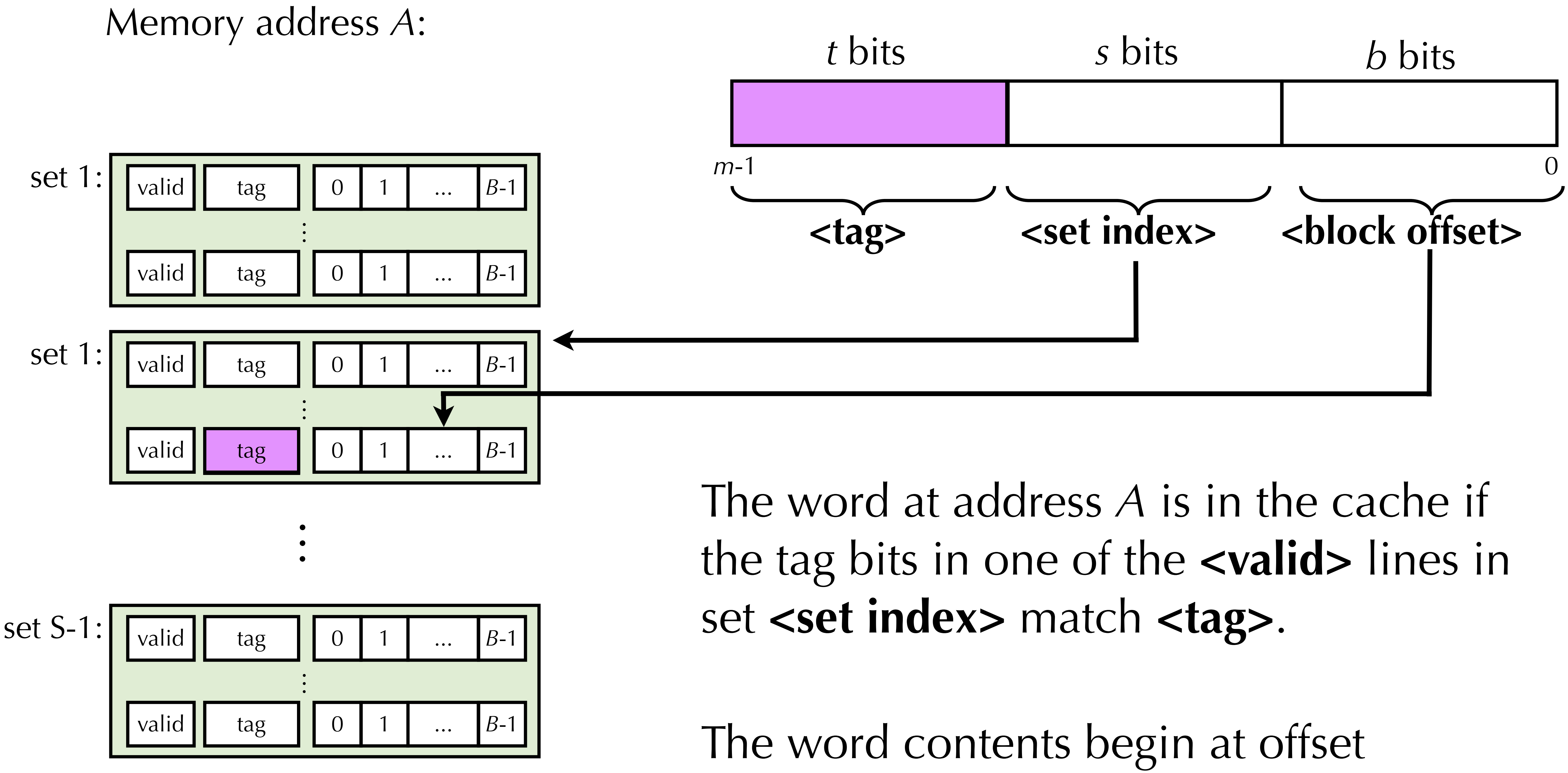


S = 2^s sets

1 valid bit per line

Cache size:
$C = B \times E \times S$ data bytes

Addressing Caches

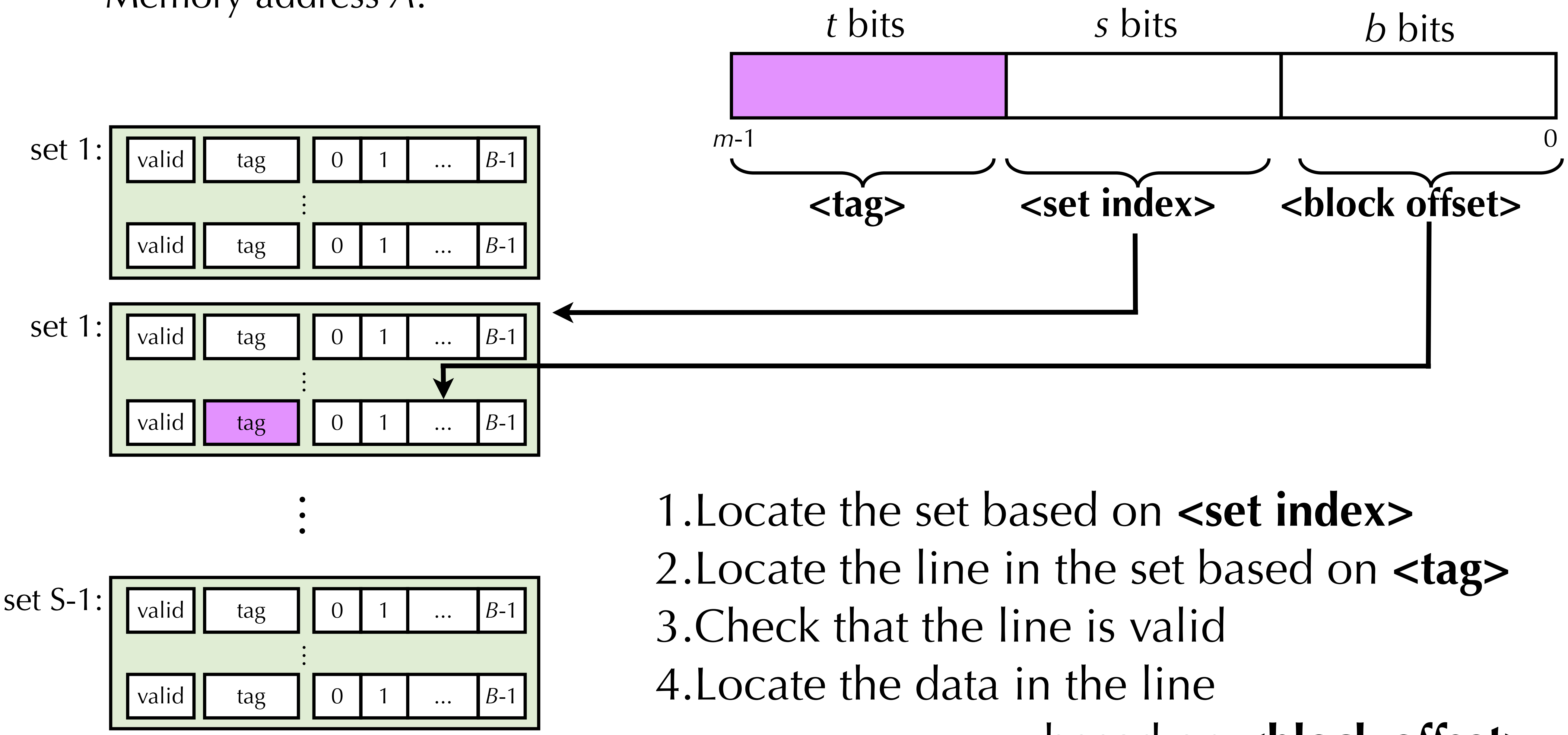


The word at address *A* is in the cache if the tag bits in one of the **<valid>** lines in set **<set index>** match **<tag>**.

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

Addressing Caches

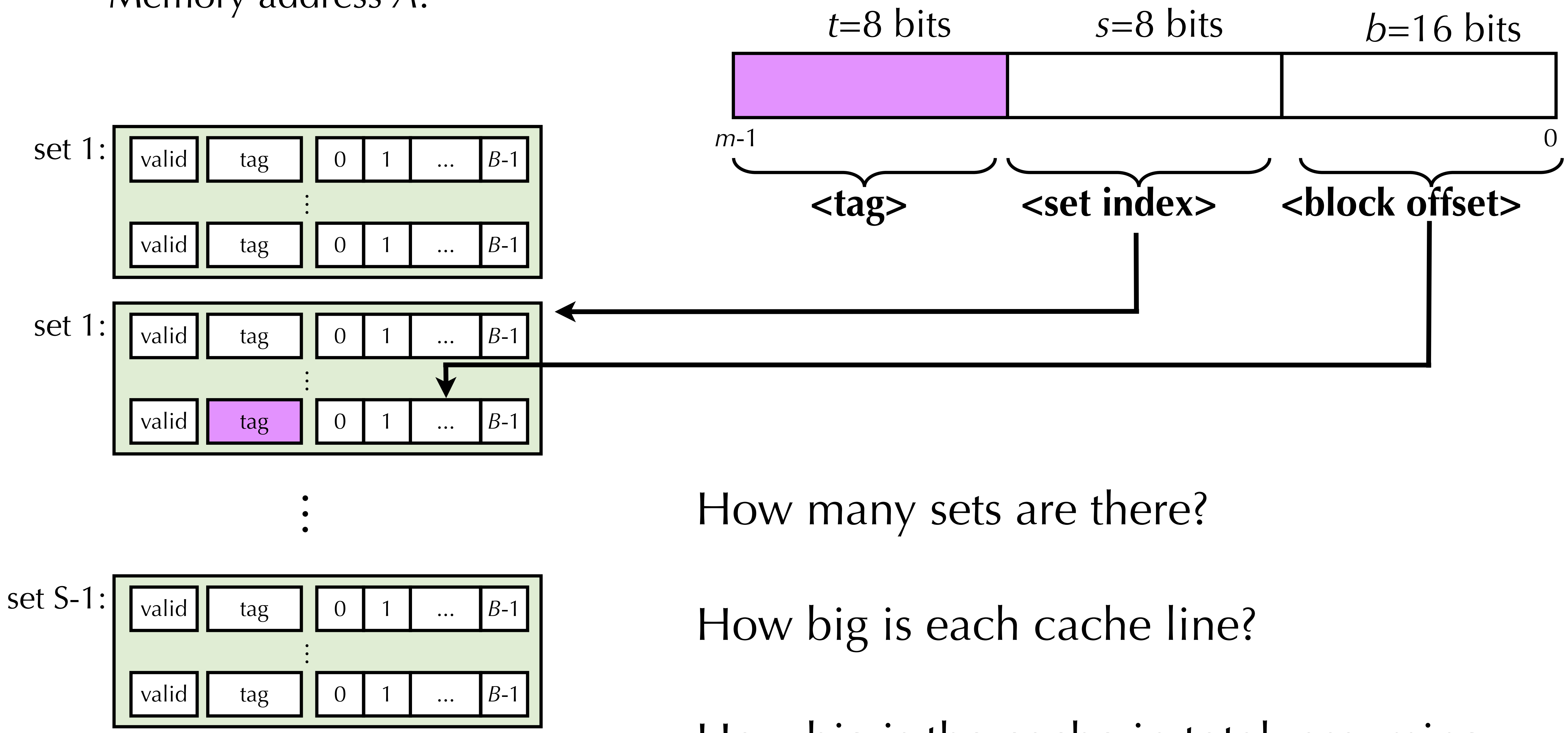
Memory address *A*:



1. Locate the set based on **<set index>**
2. Locate the line in the set based on **<tag>**
3. Check that the line is valid
4. Locate the data in the line based on **<block offset>**

Addressing Caches

Memory address *A*:



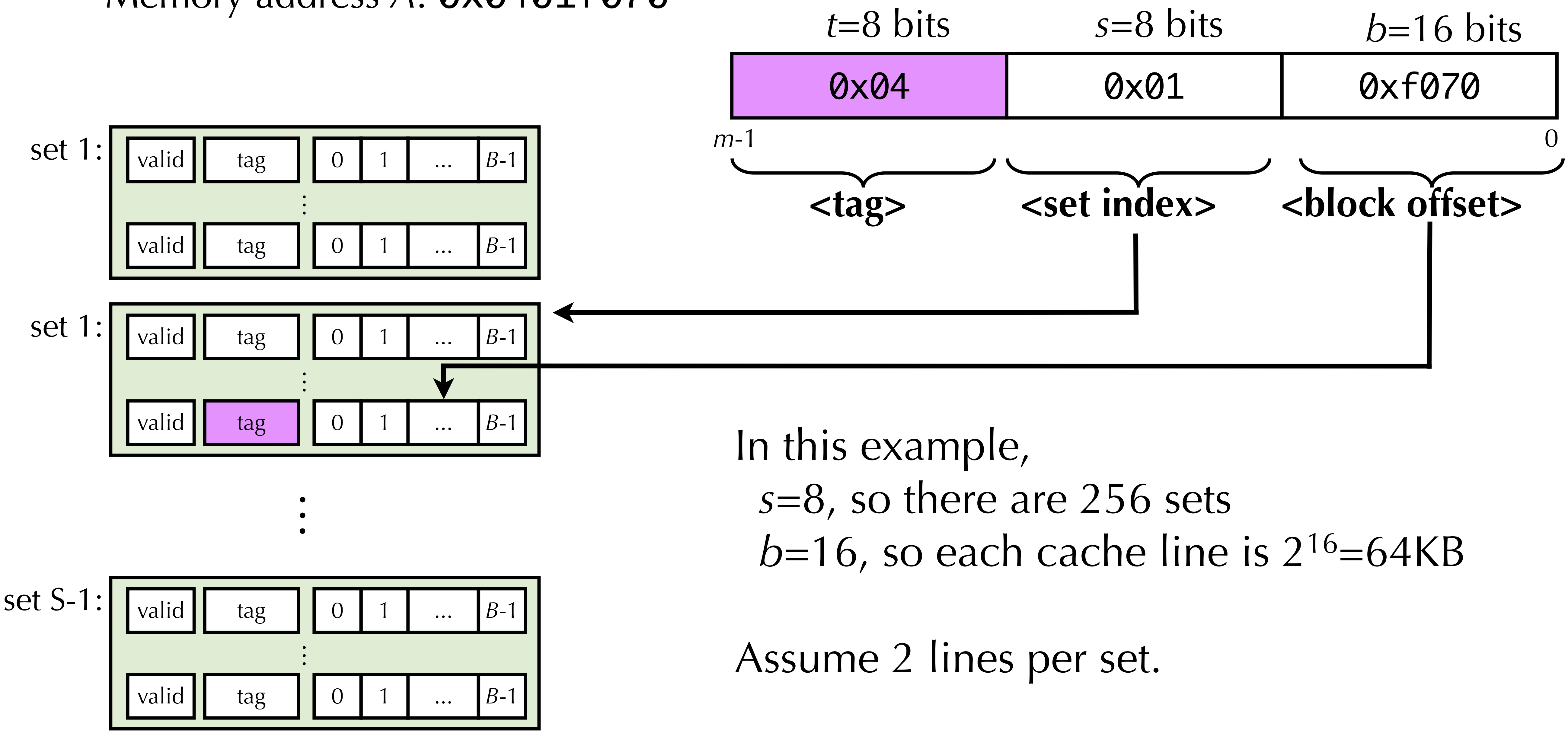
How many sets are there?

How big is each cache line?

How big is the cache in total, assuming two cache lines per set?

Addressing Caches

Memory address *A*: `0x0401f070`



In this example,
s=8, so there are 256 sets
b=16, so each cache line is 2^{16}=64KB

Assume 2 lines per set.

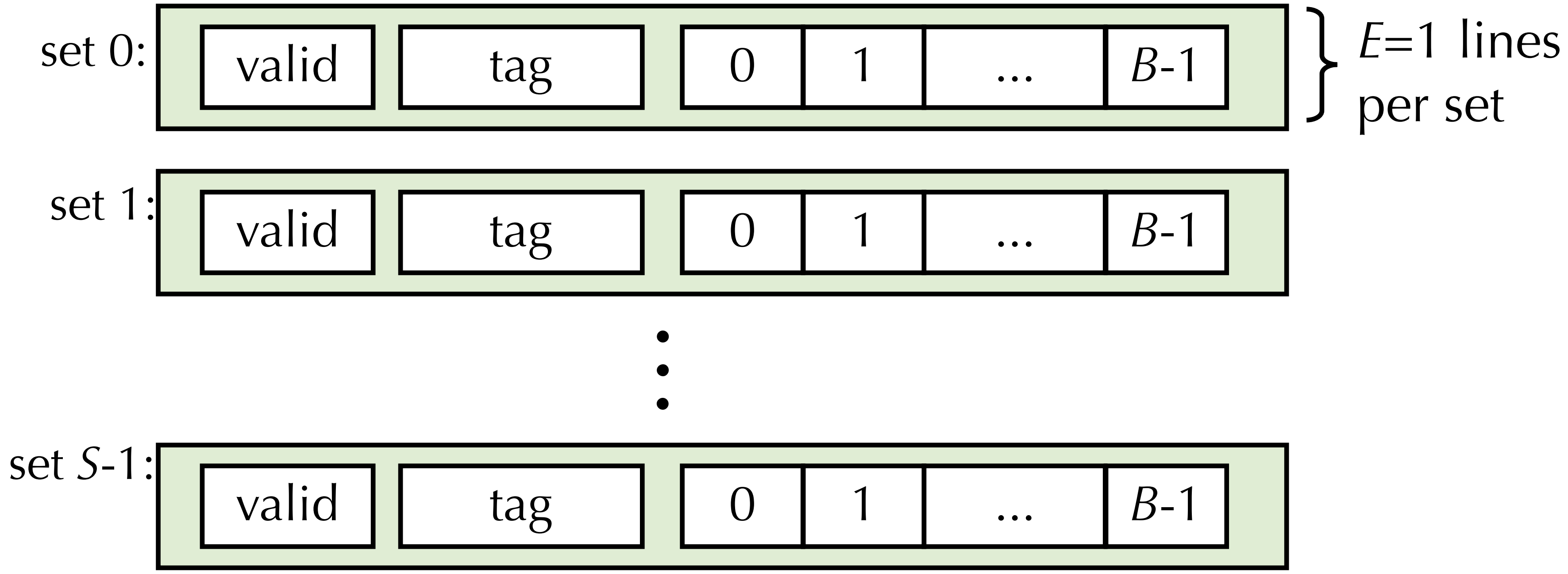
Total cache size is 256 x 2 x 64KB = 32 MB

Topics for today

- The Principle of Locality
- Memory Hierarchies
- Caching Concepts
- Direct-Mapped Cache Organization
- Set-Associative Cache Organization
- Multi-level caches
- Cache writes

Direct-Mapped Cache

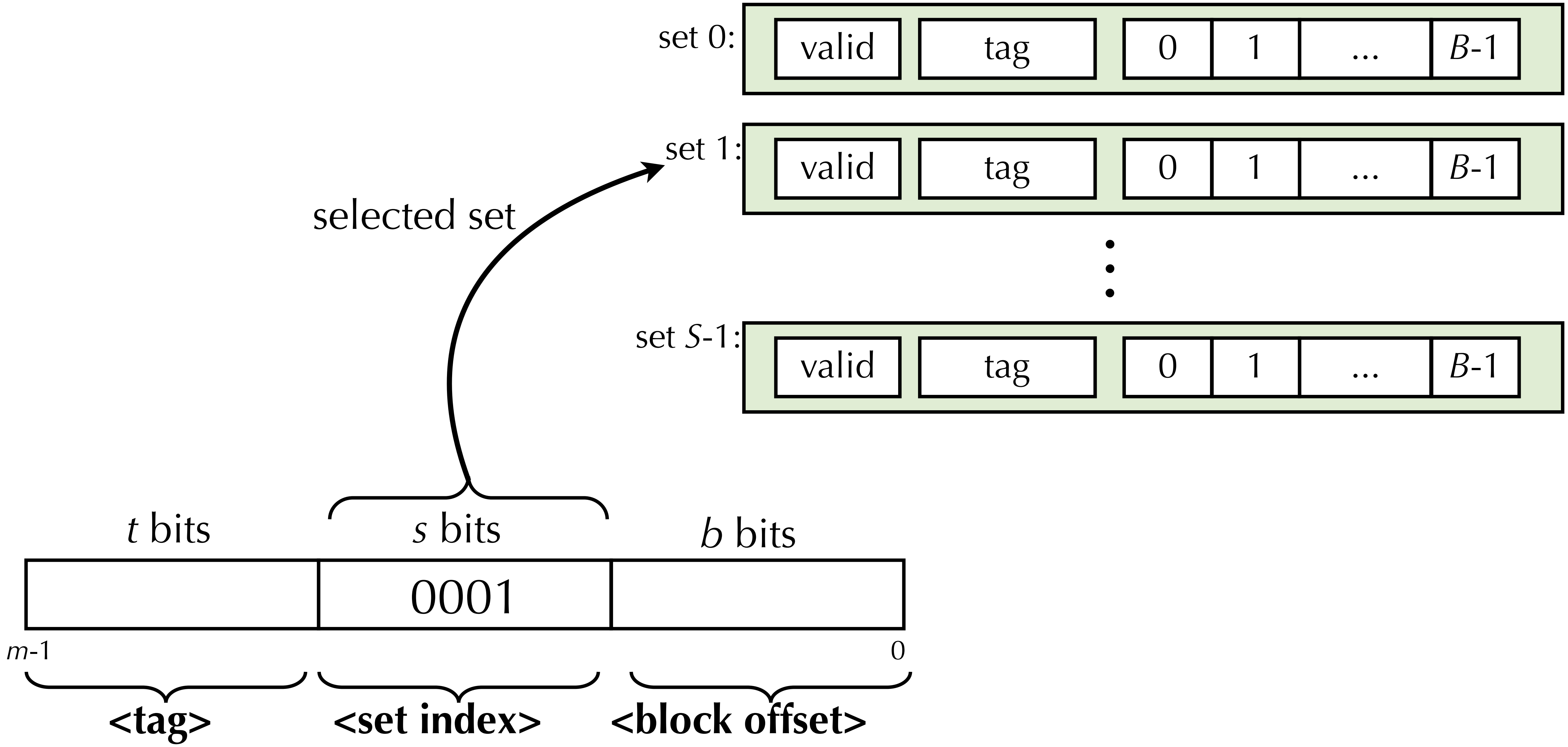
- Each set has exactly one line.
- Simplest kind of cache, and easy to build.
  - Only 1 tag compare required per access



Cache size: $C = B \times S$ data bytes

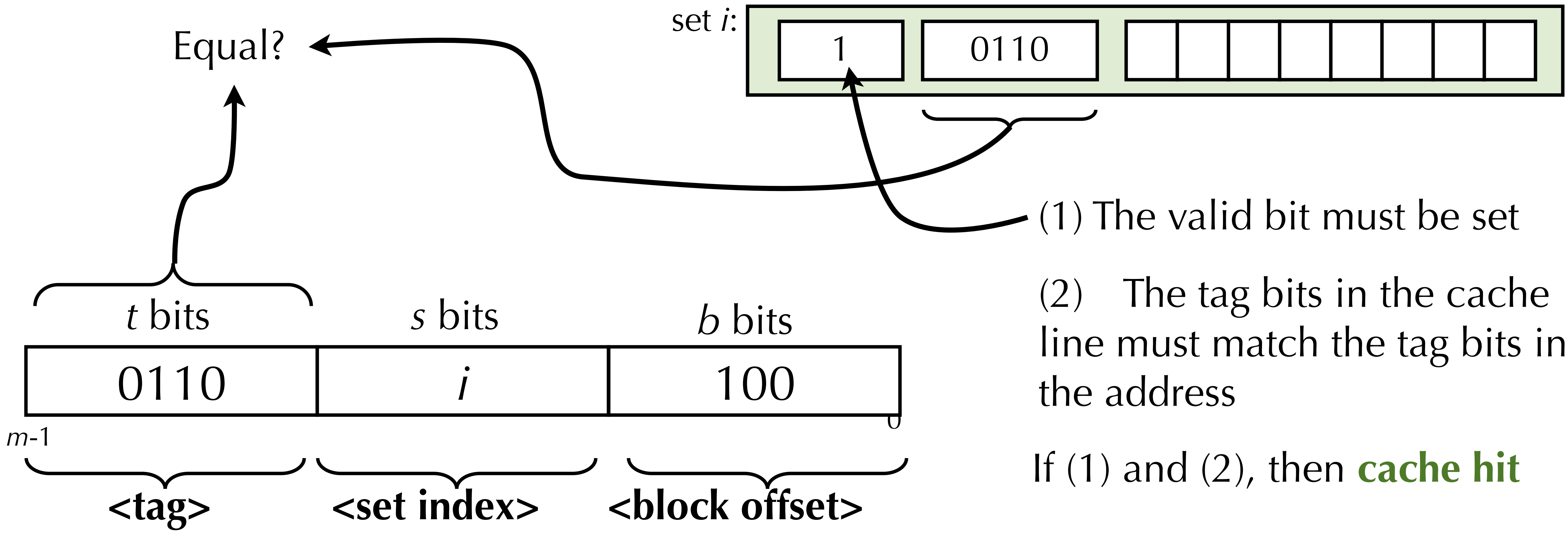
Accessing Direct-Mapped Caches

- 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*
  - **Byte selection**: Then extract the byte that we want



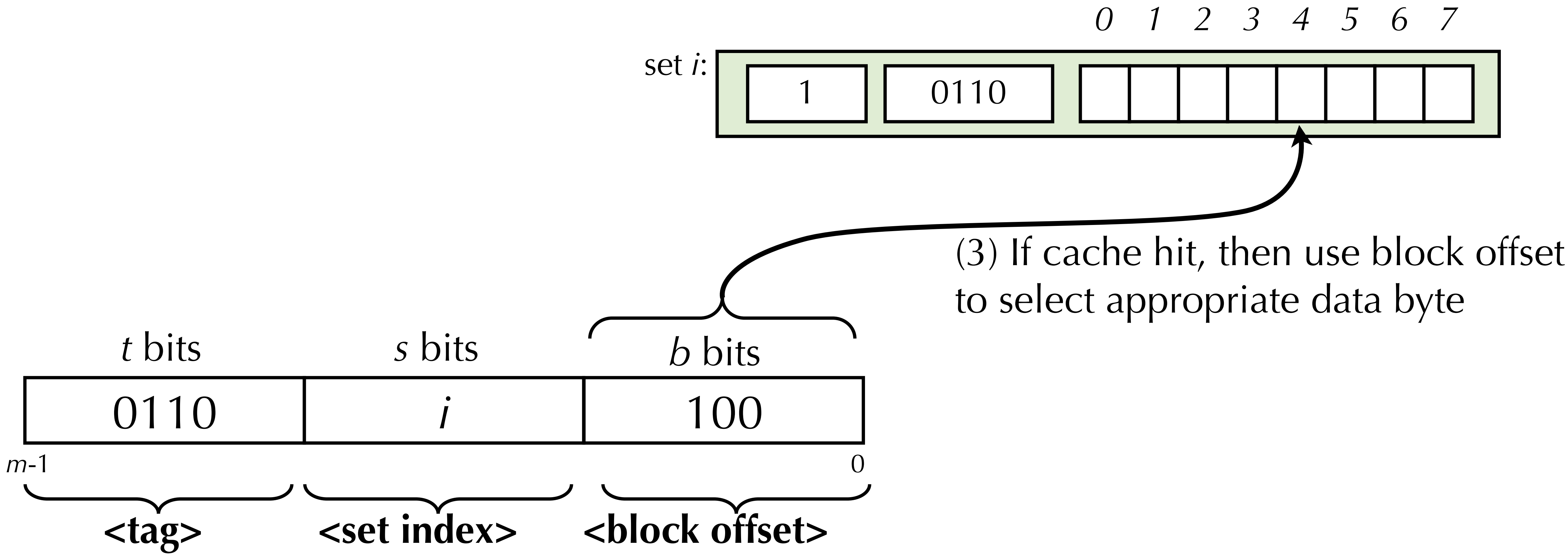
(1) The valid bit must be set

(2) The tag bits in the cache line must match the tag bits in the address

If (1) and (2), then **cache hit**

Accessing Direct-Mapped Caches

- Line matching and word selection
  - **Line matching**: Find a *valid* line in the selected set with a *matching tag*
  - **Byte selection**: Then extract the byte that we want



(3) If cache hit, then use block offset to select appropriate data byte

Handling cache misses

- A **cache miss** occurs if:
  - Tag does not match the corresponding line in the cache, or
  - Valid bit in the line is not set

Handling cache misses

- In event of a cache miss for address *A*:
  - 1) Read an entire **block** from next-lowest cache level
    - Take address *A*, set the *b* least significant bits to zero
    - E.g., suppose *A* = `0x43b7` and *b*=4 bits (block size = 2^4 bytes = 16 bytes)
      - Base address of block is `0x43b0`
      - Block contains contents of addresses `0x43b0` through `0x43bf`
  - 2) Insert block into the corresponding cache line
    - May require **evicting** contents of cache line, if line was valid already
  - 3) Set the tag bits corresponding to address *A*
  - 4) Set valid bit to 1

Types of cache misses

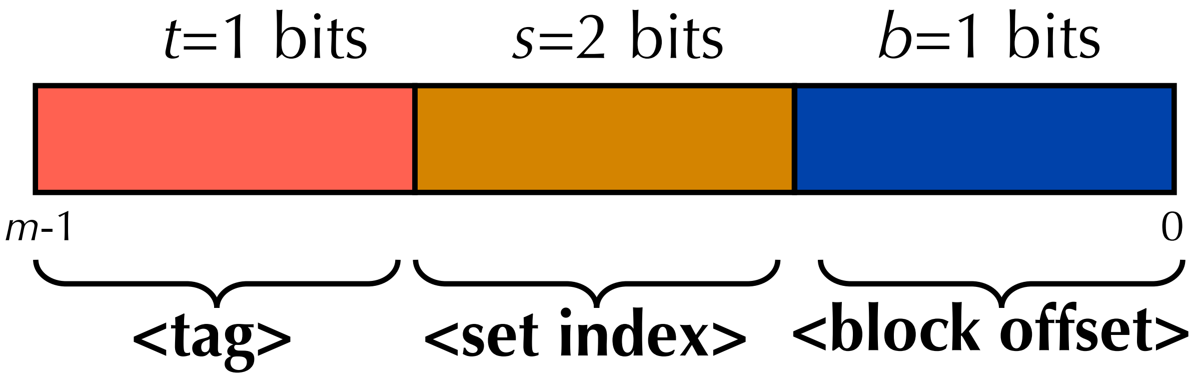
- **Cold (compulsory) miss**
  - A miss that occurs because the cache is initially empty.
  - Nothing you can do about it: thus, “compulsory”.
- **Conflict miss**
  - When reading something into the cache requires that we evict something else.
  - BUT... the cache is not full!
  - Example: If set index matches a line that is already occupied, have to evict what is already there (even if there are empty blocks in the cache!)
- **Capacity miss**
  - Occurs when the cache is too small to store everything currently being accessed by the program.
  - **Working set**: Set of “actively used” cache lines
  - The working set varies over time, depending on what the program is doing.

Direct-Mapped Cache Simulation

- Simple example:
  - 4 bit addresses (memory size = 16 bytes)
  - 1 tag bit, 2 set index bits, 1 block offset bit

Address trace (reads):

$A = 0 = 0000_2$ **cache miss**



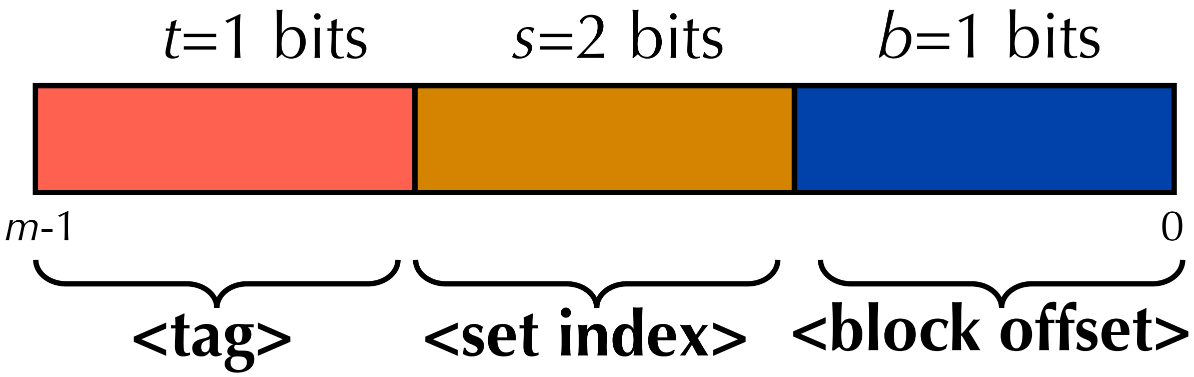
	valid	tag	data
Set 0	1	0	M[0], M[1]
Set 1	0		
Set 2	0		
Set 3	0		

Direct-Mapped Cache Simulation

- Simple example:
  - 4 bit addresses (memory size = 16 bytes)
  - 1 tag bit, 2 set index bits, 1 block offset bit

Address trace (reads):

$A = 0 = 0000_2$	**cache miss**
$A = 1 = 0001_2$	**cache hit**



	valid	tag	data
Set 0	1	0	M[0], M[1]
Set 1	0		
Set 2	0		
Set 3	0		

Direct-Mapped Cache Simulation

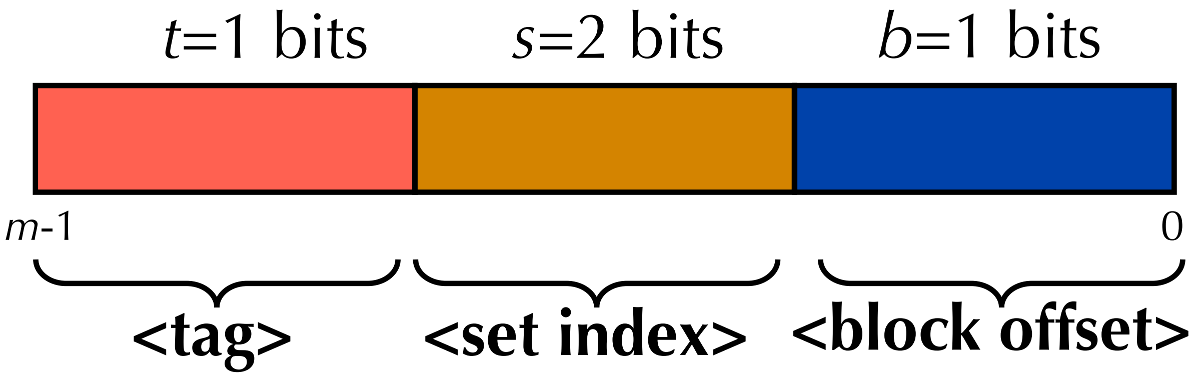
- Simple example:
  - 4 bit addresses (memory size = 16 bytes)
  - 1 tag bit, 2 set index bits, 1 block offset bit

Address trace (reads):

$A = 0 = 0000_2$ **cache miss**

$A = 1 = 0001_2$ **cache hit**

$A = 7 = 0111_2$ **cache miss**



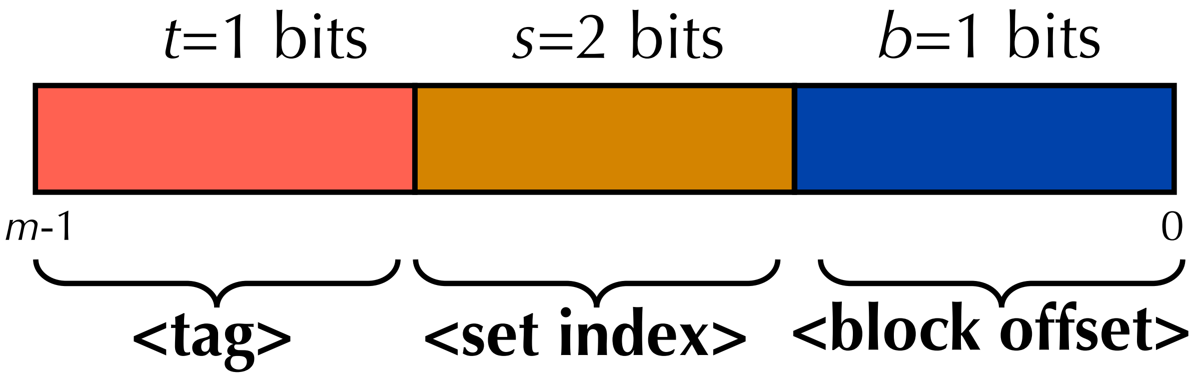
	valid	tag	data
Set 0	1	0	M[0], M[1]
Set 1	0		
Set 2	0		
Set 3	1	0	M[6], M[7]

Direct-Mapped Cache Simulation

- Simple example:
  - 4 bit addresses (memory size = 16 bytes)
  - 1 tag bit, 2 set index bits, 1 block offset bit

Address trace (reads):

$A = 0 = 0000_2$	**cache miss**
$A = 1 = 0001_2$	**cache hit**
$A = 7 = 0111_2$	**cache miss**
$A = 8 = 1000_2$	**cache miss + evict**



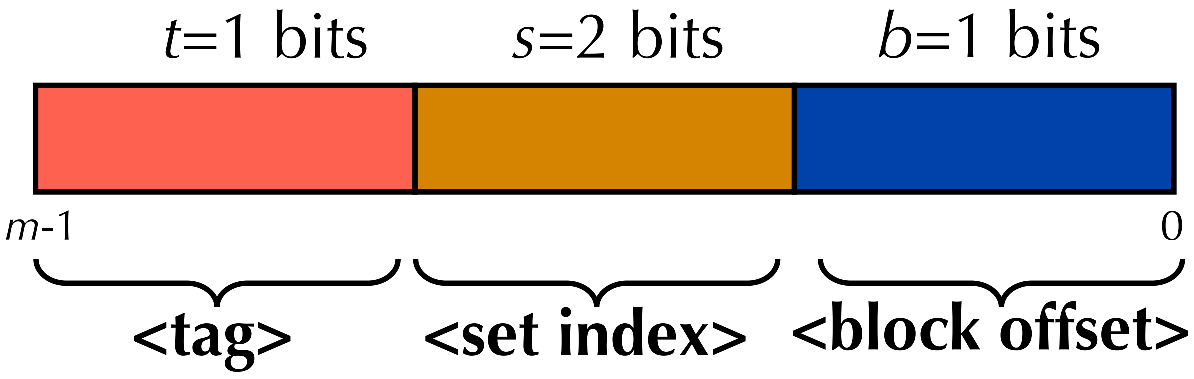
	valid	tag	data
Set 0	1	1	M[8], M[9]
Set 1	0		
Set 2	0		
Set 3	1	0	M[6], M[7]

Direct-Mapped Cache Simulation

- Simple example:
  - 4 bit addresses (memory size = 16 bytes)
  - 1 tag bit, 2 set index bits, 1 block offset bit

Address trace (reads):

$A = 0 = 0000_2$	**cache miss**
$A = 1 = 0001_2$	**cache hit**
$A = 7 = 0111_2$	**cache miss**
$A = 8 = 1000_2$	**cache miss + evict**
$A = 0 = 0000_2$	**cache miss + evict**



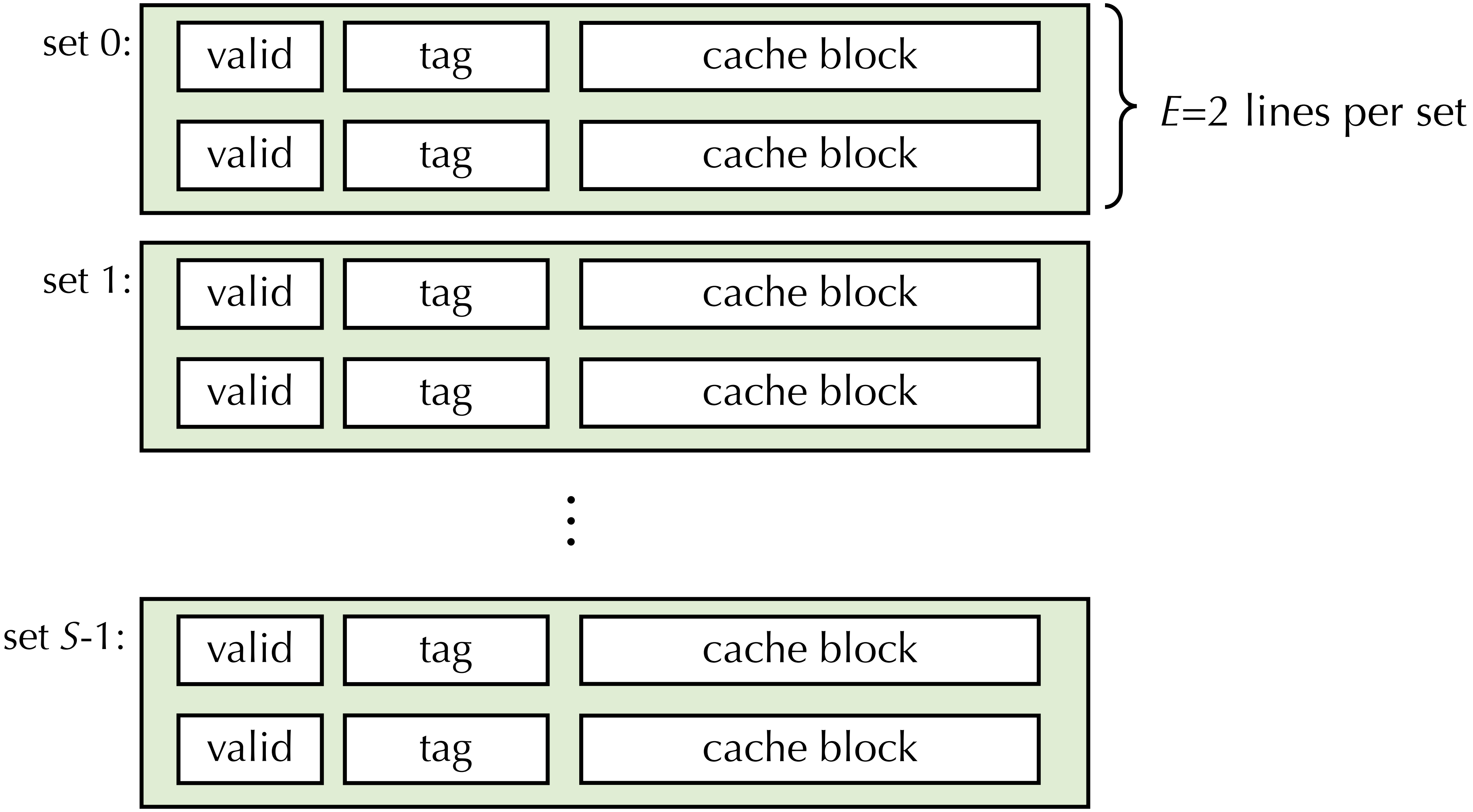
	valid	tag	data
Set 0	1	0	M[0], M[1]
Set 1	0		
Set 2	0		
Set 3	1	0	M[6], M[7]

Topics for today

- The Principle of Locality
- Memory Hierarchies
- Caching Concepts
- Direct-Mapped Cache Organization
- Set-Associative Cache Organization
- Multi-level caches
- Cache writes

Set Associative Caches

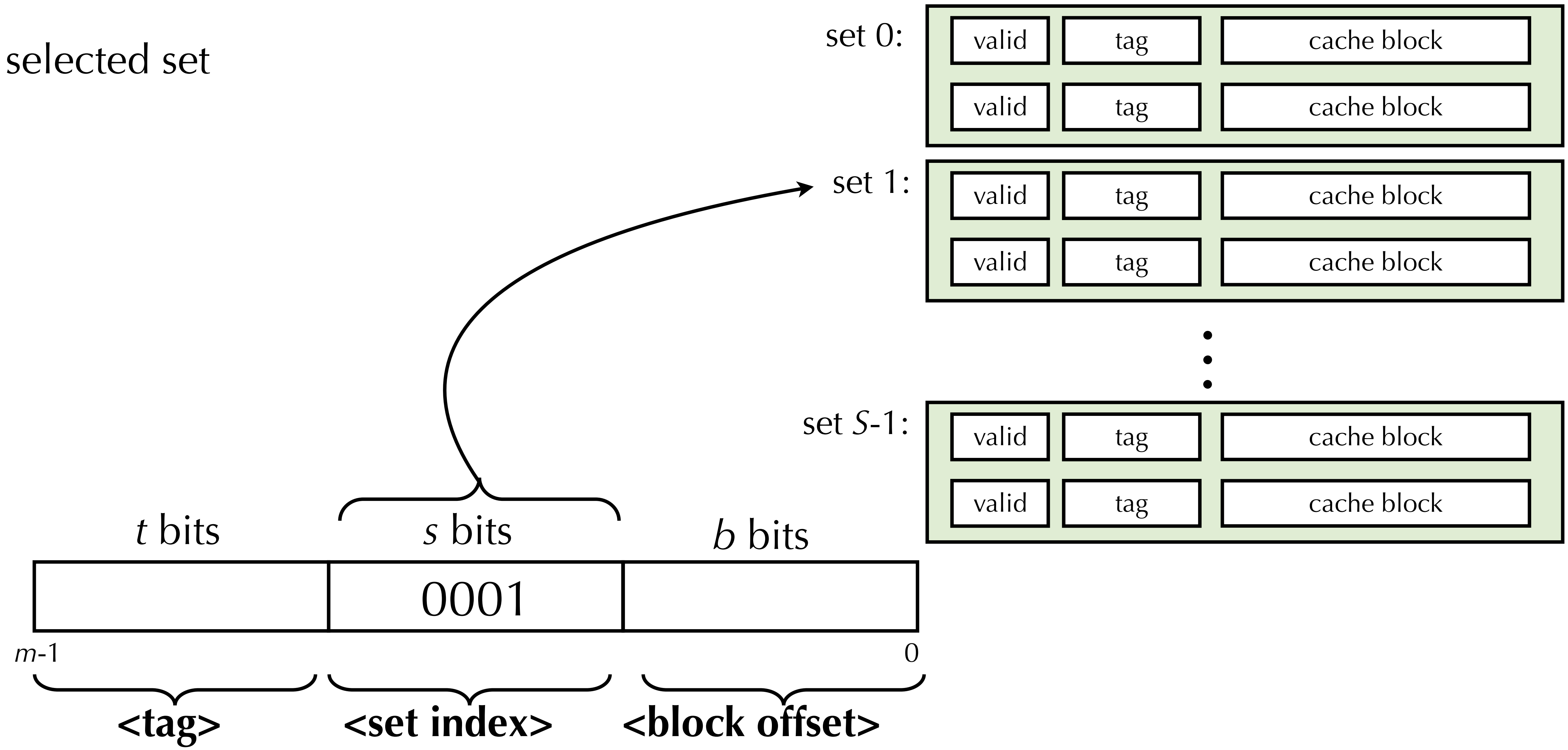
- Characterized by more than one line per set
  - Can use **any line in the set** to store a given cache block.



E-way associative cache

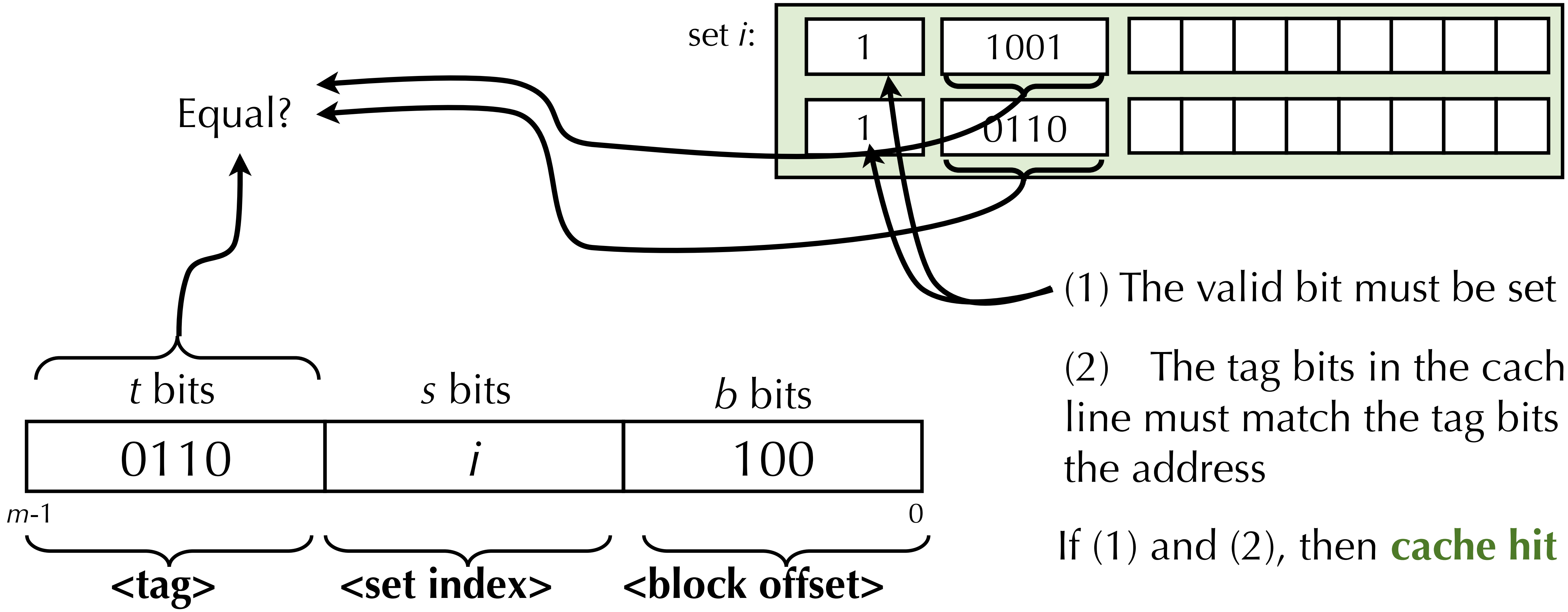
Accessing Set Associative Caches

- Set selection: same as direct-mapped cache
  - Use the set index bits to determine the set of interest.



Accessing Set Associative Caches

- Line matching and word selection: same as direct-mapped
  - **Line matching**: Find a *valid* line in the selected set with a *matching tag*. Check **all valid lines**.
  - **Byte selection**: Then extract the byte that we want



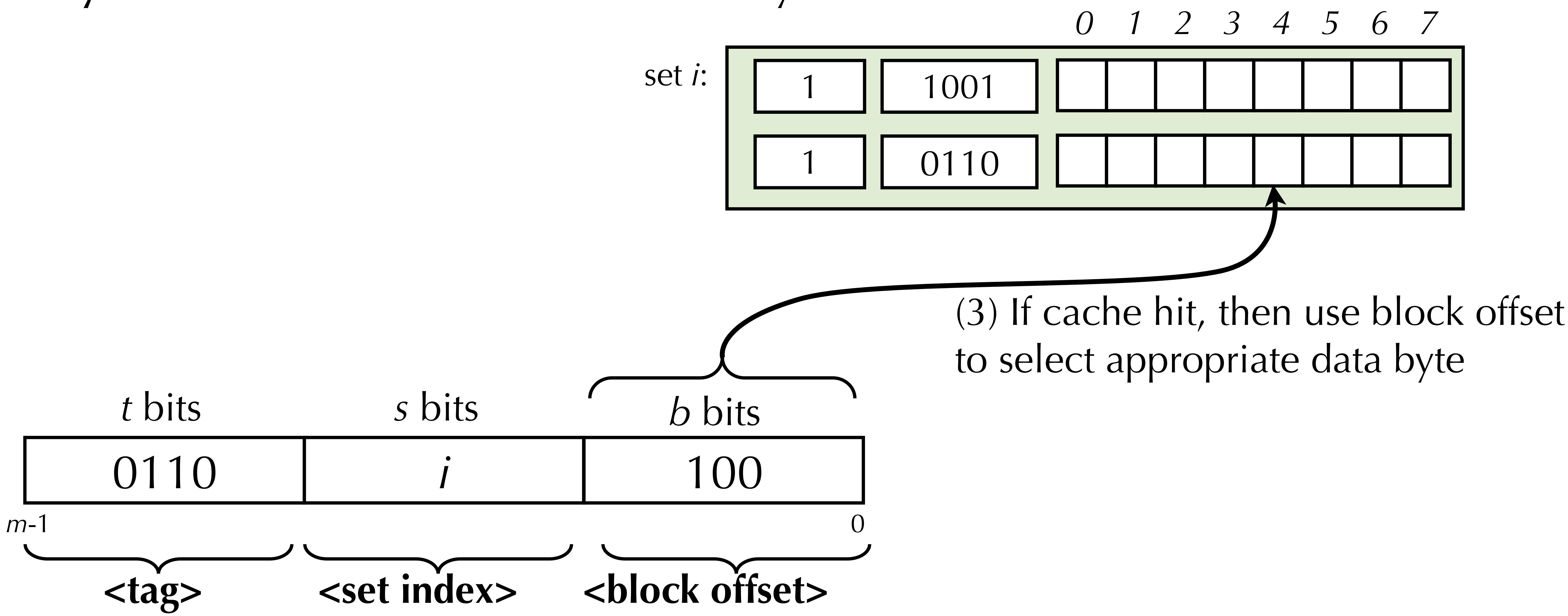
(1) The valid bit must be set

(2) The tag bits in the cache line must match the tag bits in the address

If (1) and (2), then **cache hit**

Accessing Set Associative Caches

- Line matching and word selection: same as direct-mapped
  - **Line matching**: Find a *valid* line in the selected set with a *matching tag*. Check **all valid lines**.
  - **Byte selection**: Then extract the byte that we want



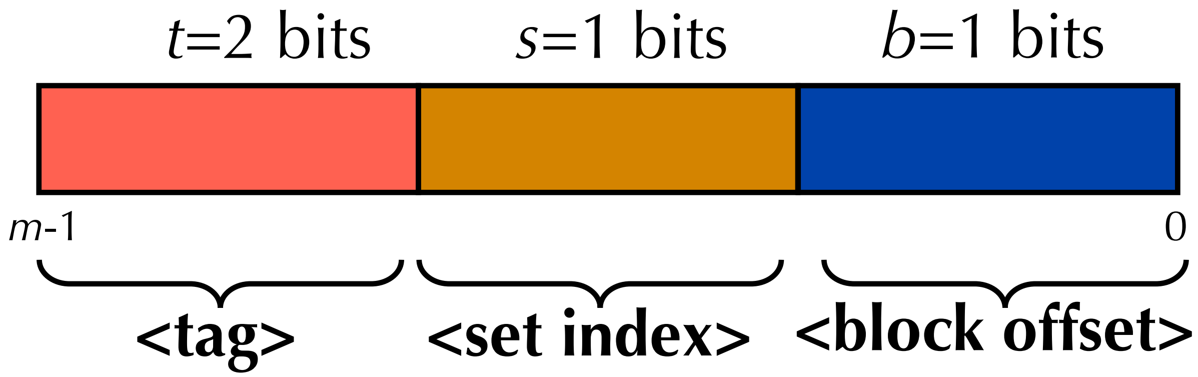
(3) If cache hit, then use block offset to select appropriate data byte

2-Way Associative Cache Simulation

- Simple example:
  - 4 bit addresses
  - 2 tag bits, 1 set index bit, 1 block offset bit

Address trace (reads):

$A = 0 = 0000_2$ **cache miss**



	valid	tag	data
Set 0	1	00	M[0], M[1]
	0		
Set 1	0		
	0		

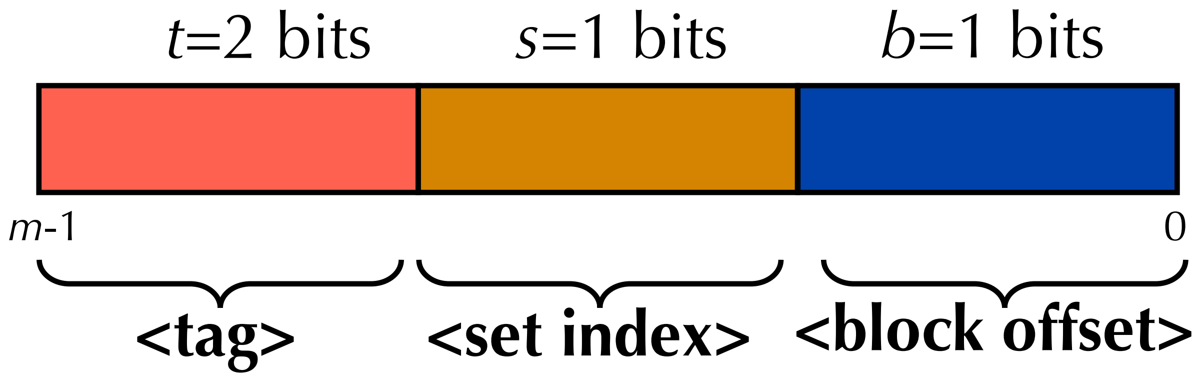
2-Way Associative Cache Simulation

- Simple example:
  - 4 bit addresses
  - 2 tag bits, 1 set index bit, 1 block offset bit

Address trace (reads):

$A = 0 = 0000_2$ **cache miss**

$A = 1 = 0001_2$ **cache hit**



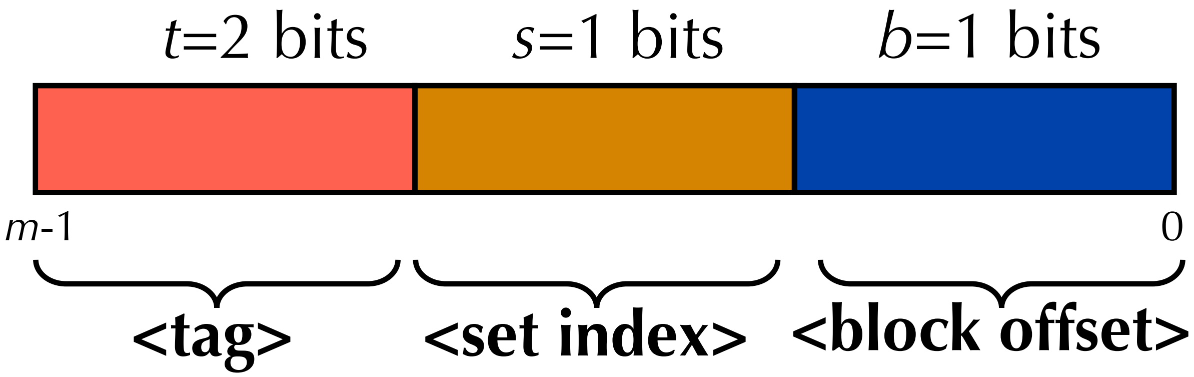
	valid	tag	data
Set 0	1	00	M[0], M[1]
	0		
Set 1	0		
	0		

2-Way Associative Cache Simulation

- Simple example:
  - 4 bit addresses
  - 2 tag bits, 1 set index bit, 1 block offset bit

Address trace (reads):

$A = 0 = 0000_2$	**cache miss**
$A = 1 = 0001_2$	**cache hit**
$A = 7 = 0111_2$	**cache miss**



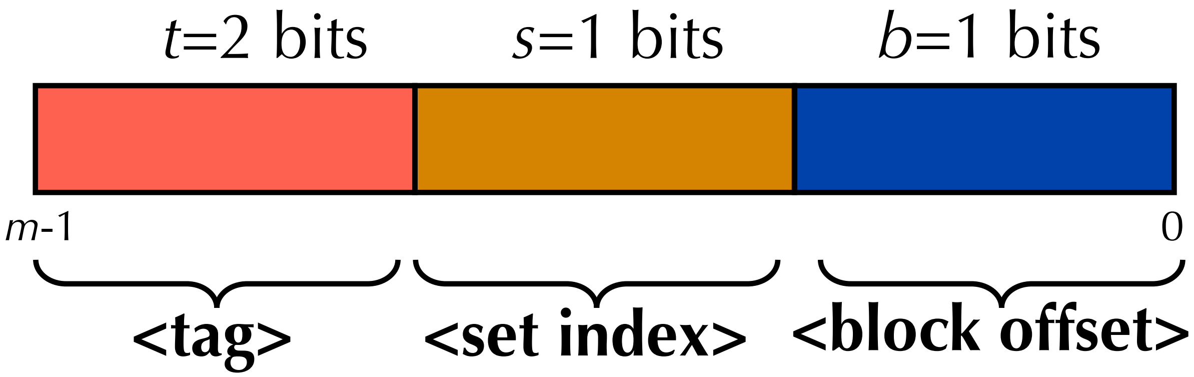
	valid	tag	data
Set 0	1	00	M[0], M[1]
	0		
Set 1	1	01	M[6], M[7]
	0		

2-Way Associative Cache Simulation

- Simple example:
  - 4 bit addresses
  - 2 tag bits, 1 set index bit, 1 block offset bit

Address trace (reads):

$A = 0 = 0000_2$	**cache miss**
$A = 1 = 0001_2$	**cache hit**
$A = 7 = 0111_2$	**cache miss**
$A = 8 = 1000_2$	**cache miss**



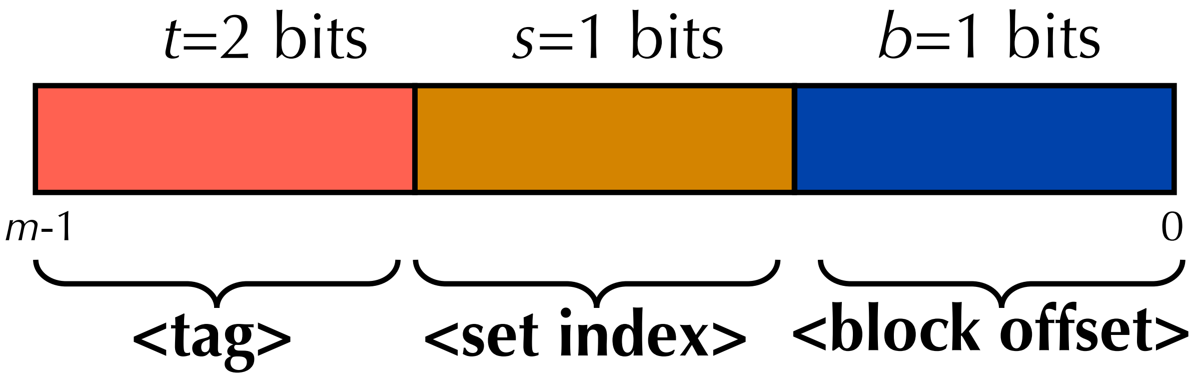
	valid	tag	data
Set 0	1	00	M[0], M[1]
	1	10	M[8], M[9]
Set 1	1	01	M[6], M[7]
	0		

2-Way Associative Cache Simulation

- Simple example:
  - 4 bit addresses
  - 2 tag bits, 1 set index bit, 1 block offset bit

Address trace (reads):

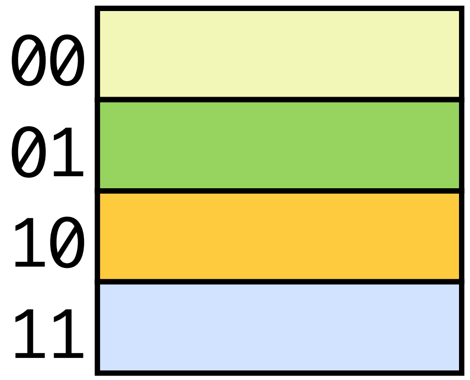
$A = 0 = 0000_2$	cache miss
$A = 1 = 0001_2$	cache hit
$A = 7 = 0111_2$	cache miss
$A = 8 = 1000_2$	cache miss
$A = 0 = 0000_2$	cache hit



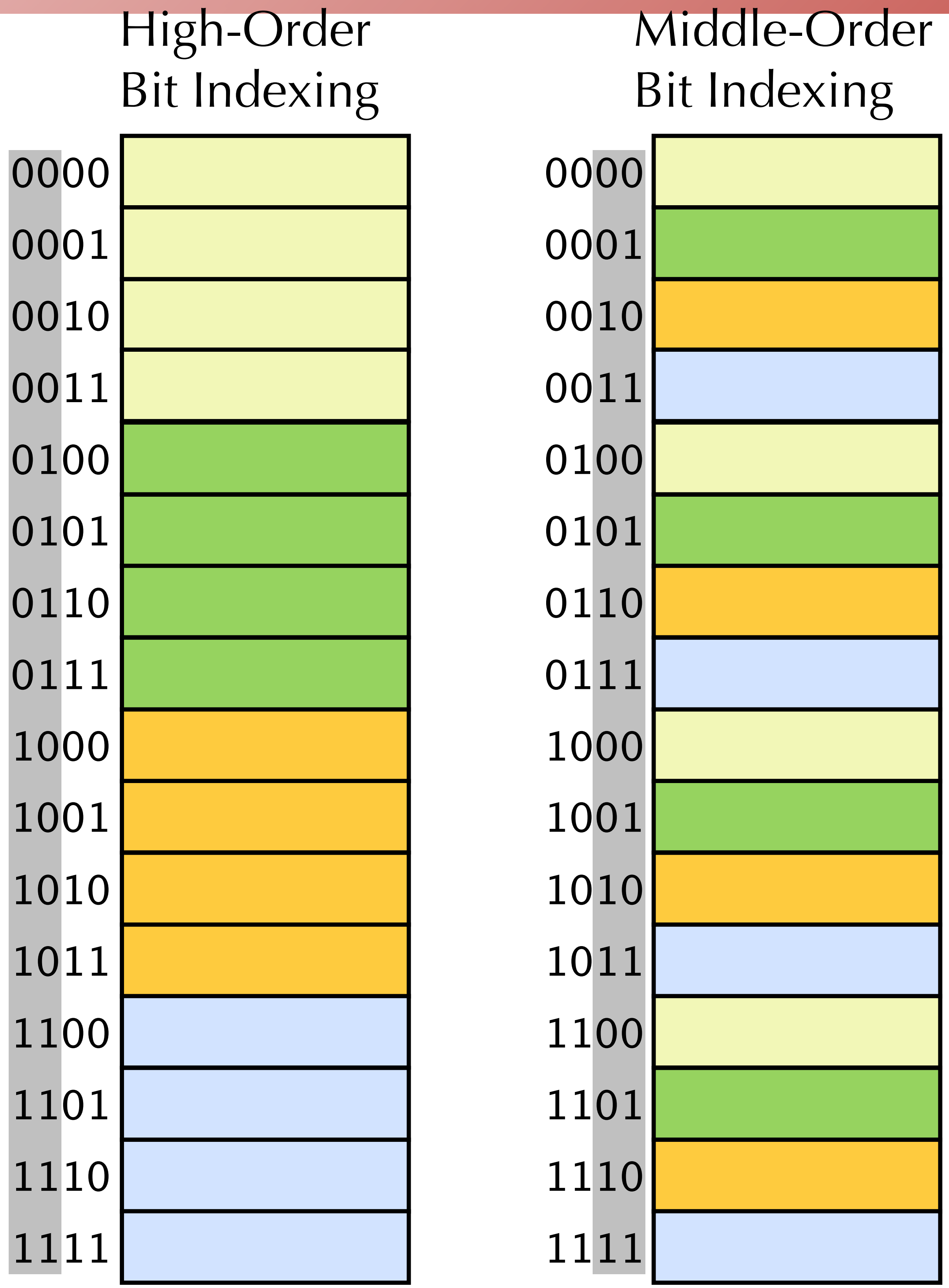
	valid	tag	data
Set 0	1	00	M[0], M[1]
	1	10	M[8], M[9]
Set 1	1	01	M[6], M[7]
	0		

Why use the middle bits as the set index?

Four-set cache



- High-Order Bit Indexing
  - Adjacent memory lines would map to same set
  - Poor use of spatial locality
- Middle-Order Bit Indexing
  - Consecutive memory lines map to different cache lines
  - Can hold *S*×*B*×*E*-byte contiguous region of address space in cache at one time



Maintaining a Set-Associative Cache

- When a set has multiple lines, which line do we use?
- Several different policies are used:
  - **Least Recently Used** (LRU) – must keep track of last access time for each line
  - **Not recently Used** (NRU) – Similar idea but not exact: Just track whether or not a cache line has been accessed during the last “N” accesses.
  - **Random** – just pick any one of the lines randomly

Summary of cache concepts

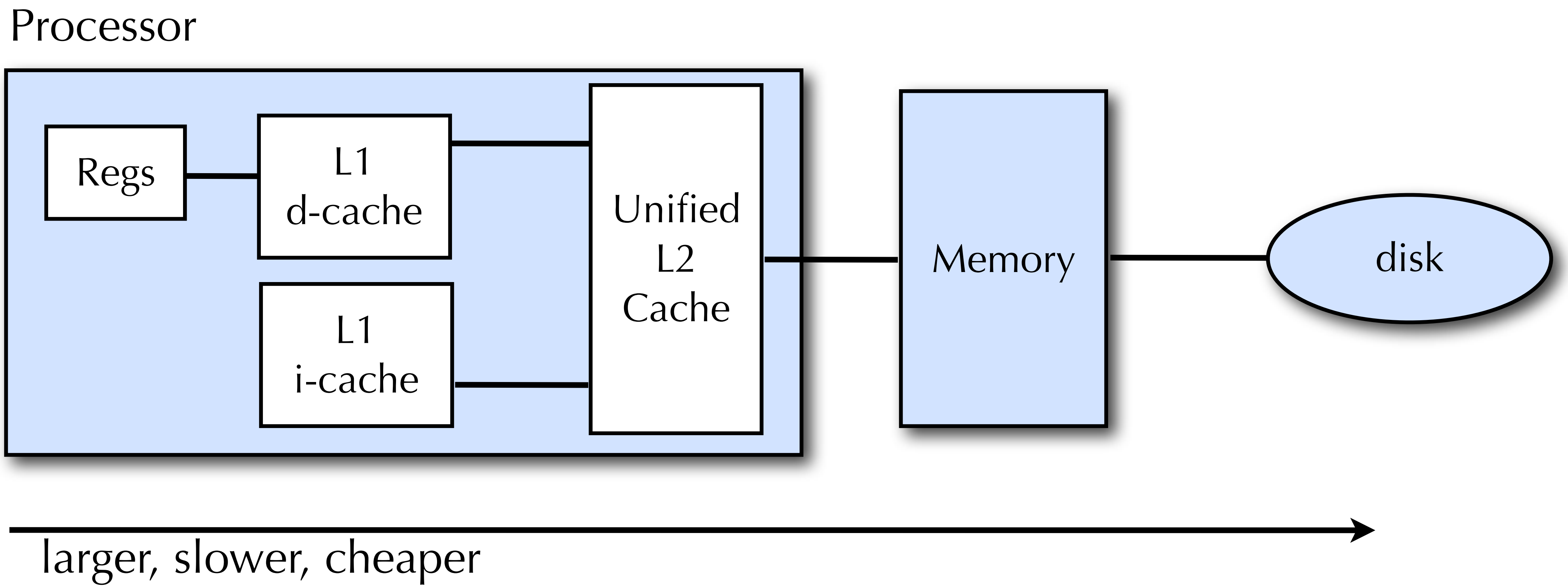
- A **block** is a fixed-sized packet of information that moves back and forth between a cache and a lower level of memory hierarchy.
- A **line** is a container in a cache that stores a block, as well as other info such as valid bit and tag bits.
- A **set** is a collection of one or more lines.
  - Sets in direct-mapped caches have a single line
  - Sets in set-associative caches have multiple lines

Topics for today

- The Principle of Locality
- Memory Hierarchies
- Caching Concepts
- Direct-Mapped Cache Organization
- Set-Associative Cache Organization
- Multi-level caches
- Cache writes

Multi-Level Caches

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

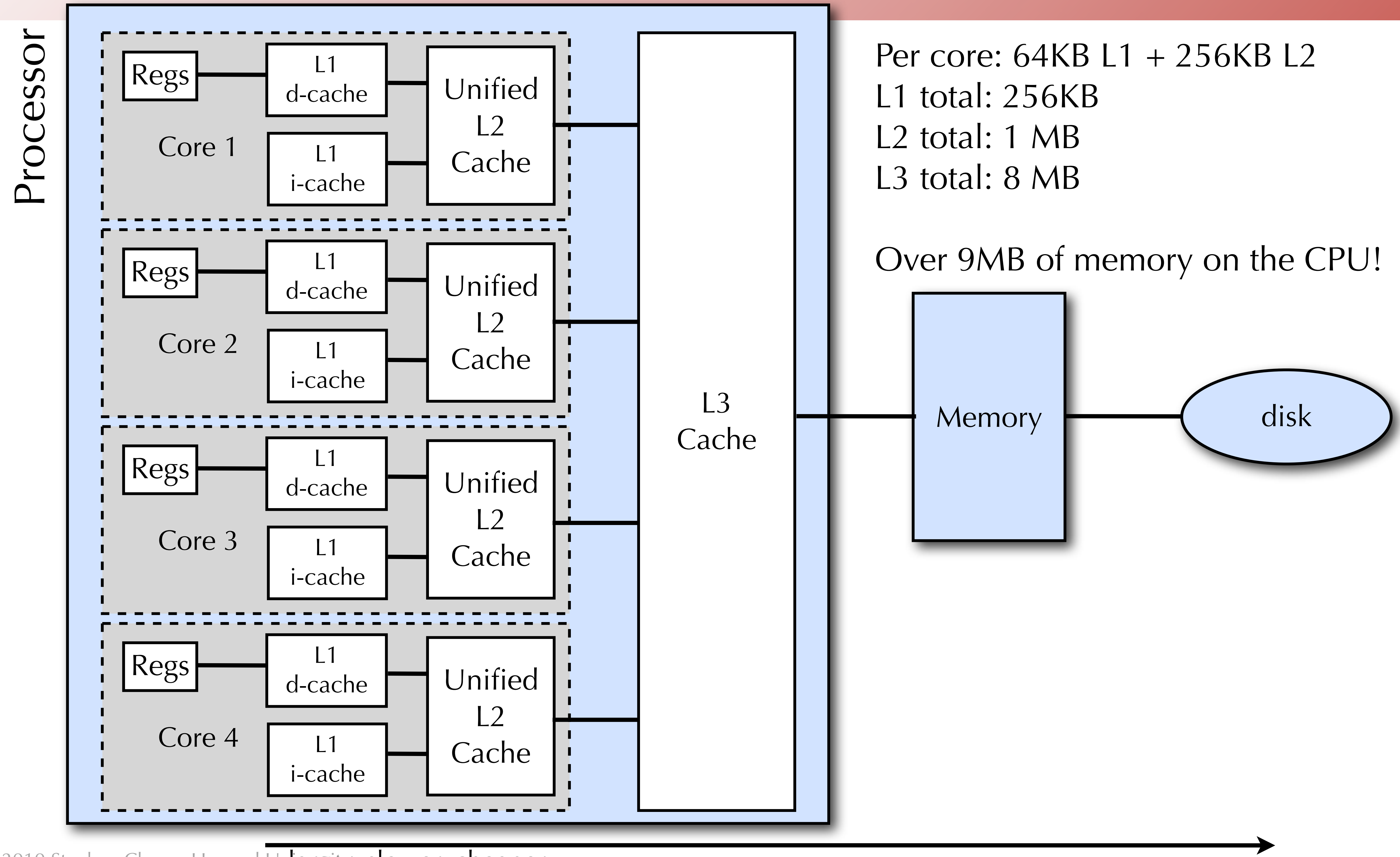


Intel Core i7-965 processor specs

- 3.2 GHz clock speed, four cores
- L1 cache:
  - 64 KB (32 KB instruction, 32KB data), per core; 8-way set associative, 64 sets
  - 4 CPU cycles to access (~ 1 ns)
- L2 cache
  - 256KB, per core; 8-way set associative, 512 sets
  - 10 CPU cycles to access (~3 ns)
- L3 cache:
  - 8 MB, is shared across all cores; 8-way set associative, 8192 sets
  - 35-40 CPU cycles to access (~12 ns)
- Main memory:
  - Up to 1 TB
  - About 60 ns to access – 60× slower than L1 cache, 20× slower than L2, 5× slower then L3

http://techreport.com/articles.x/15818
http://ark.intel.com/Product.aspx?id=37149&processor=i7-965&spec-codes=SLBCJ
http://software.intel.com/en-us/articles/who-moved-the-goal-posts-the-rapidly-changing-world-of-cpus/

Intel Core i7-965 processor specs



Per core: 64KB L1 + 256KB L2
L1 total: 256KB
L2 total: 1 MB
L3 total: 8 MB

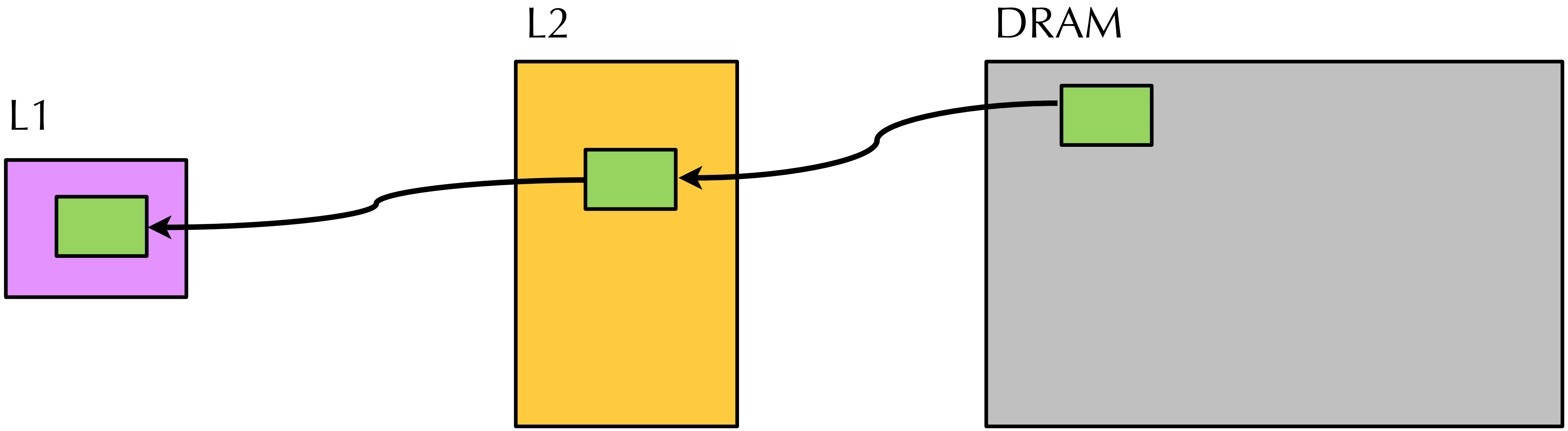
Over 9MB of memory on the CPU!

Topics for today

- The Principle of Locality
- Memory Hierarchies
- Caching Concepts
- Direct-Mapped Cache Organization
- Set-Associative Cache Organization
- Multi-level caches
- Cache writes

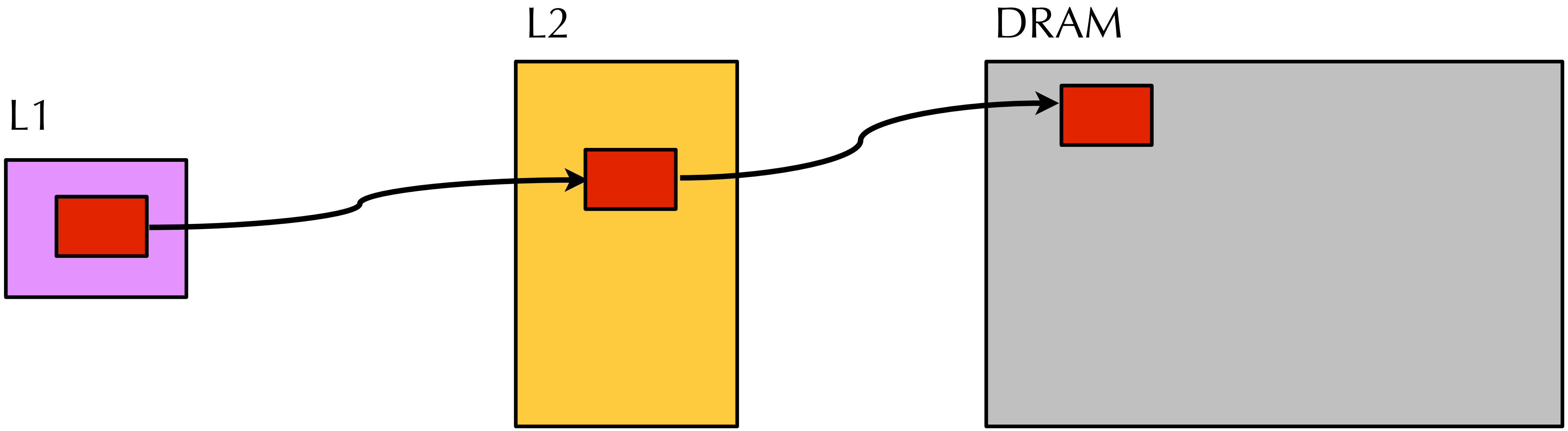
Handling cache writes

- On **reads**, the memory system fetches data from lower levels and moves it into higher levels.
  - For example, moving data from L2 cache to L1 cache, or from DRAM to L2 cache.
- On a **write**, where do we put the modified data?



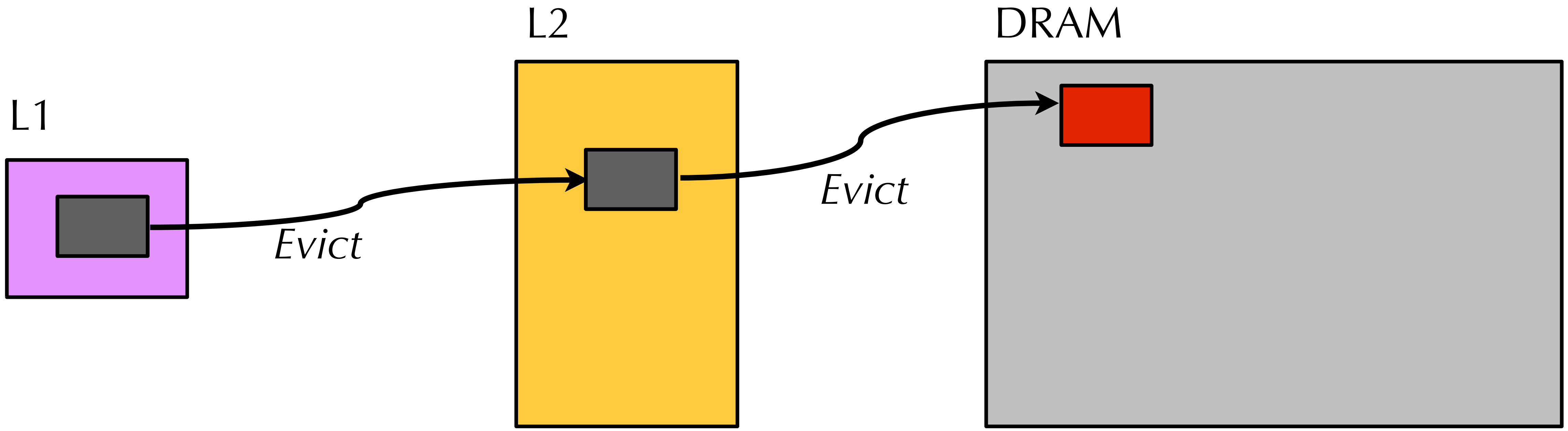
Handling cache writes

- Approach #1: **Write through** policy
  - Data written through to lower levels of cache as soon as it is modified.
  - Simple, but causes an (expensive) memory write for each cache write.



Handling cache writes

- Approach #2: **Write back** policy
  - Only write data to highest-level cache.
  - When cache line is **evicted**, write back to next lower level cache.
  - Caches must keep track of whether each line is **dirty** (needs writing back)



Next lecture

- Measuring cache size and performance.

- Optimizing programs to exploit cache locality.