Lecture 12
Cache Memories

CS213 – Intro to Computer Systems

Branden Ghena – Winter 2025

Slides adapted from:
St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Administrivia

- Deadline reminders
  - Homework 3 on Thurday
  - Attack Lab next week Tuesday

- Next week
  - Homework 4 & SETI Lab come out

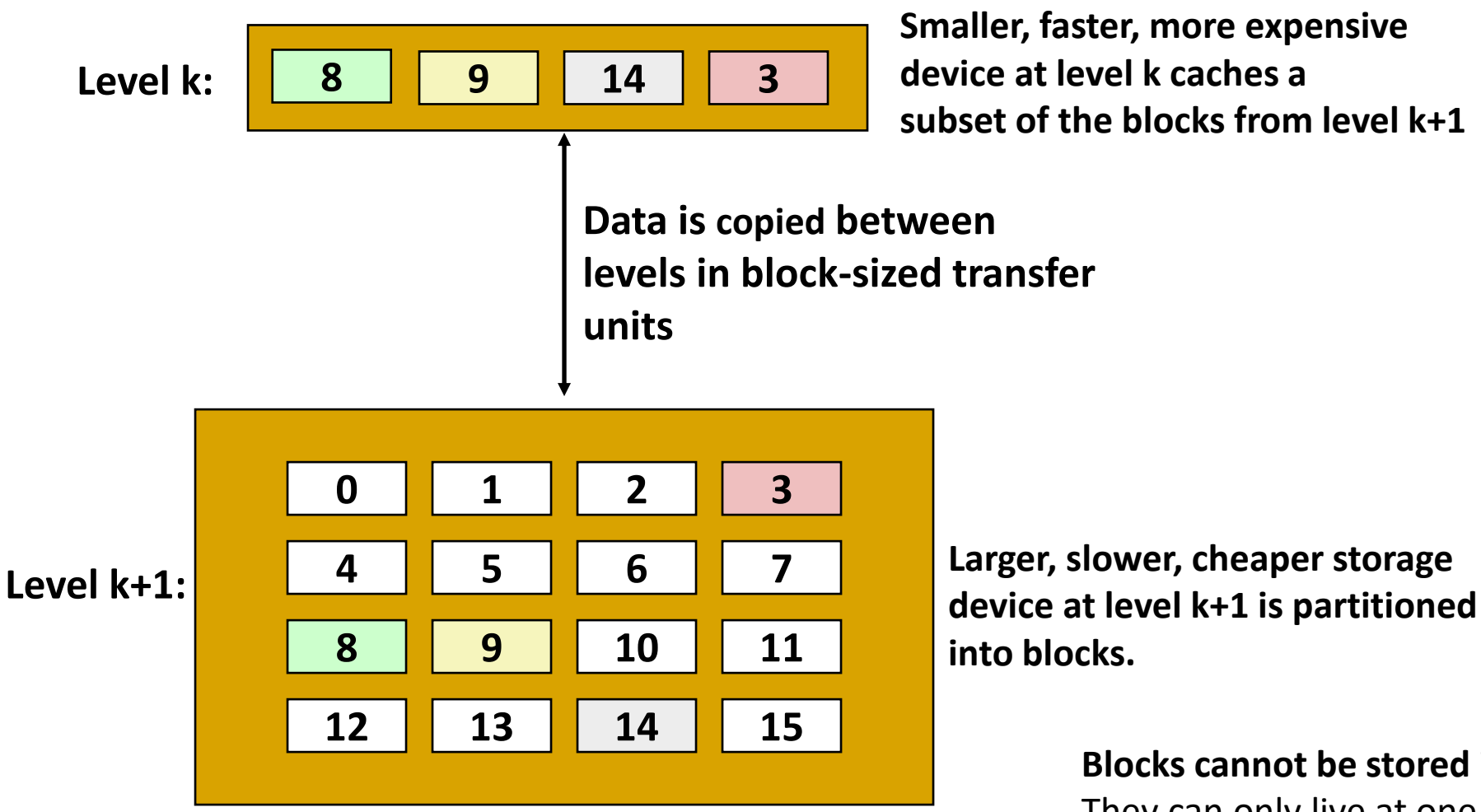
  - No lecture next week Thursday
    - Enjoy the break

Today's Goals

- Understand how locality makes a cache useful

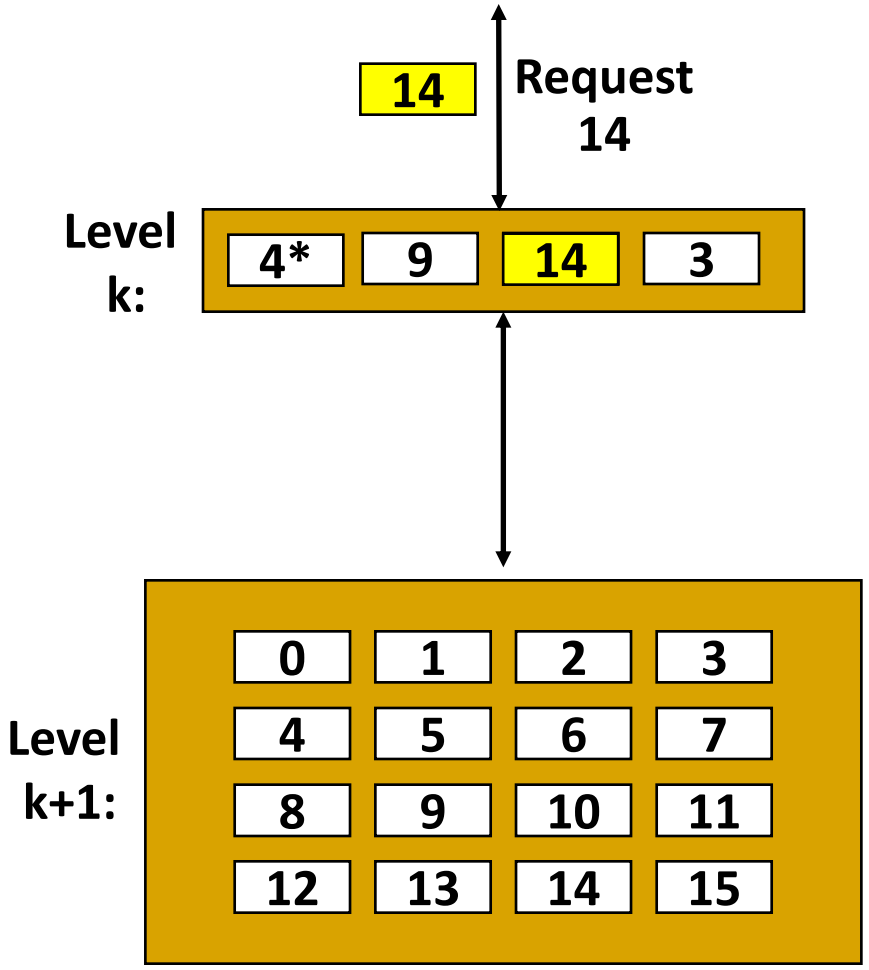
- Discuss organization of various cache designs
  - Direct-mapped caches
  - N-way set-associative caches
  - Fully-associative caches

Caching in a memory hierarchy



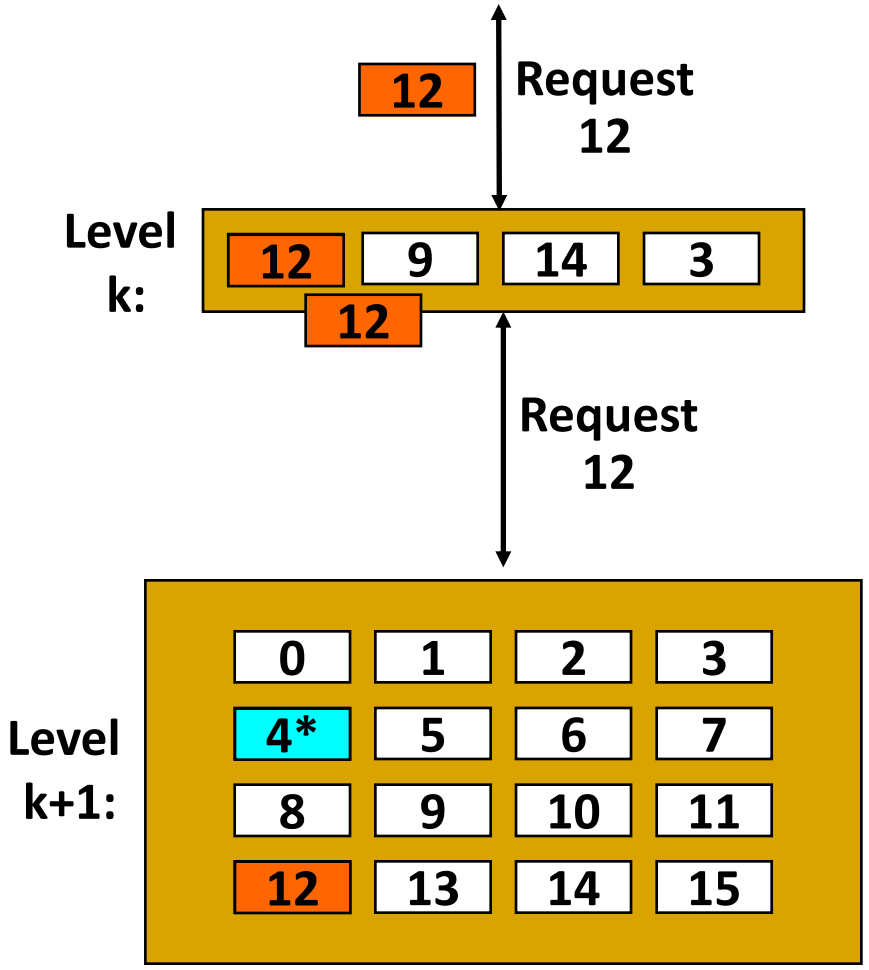
Blocks cannot be stored in an arbitrary location!
They can only live at one of a fixed set of locations.
In this example: they must be in the same "column" for both levels.

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

General caching concepts



" * " means the block is *dirty*
(i.e., it has been modified)

- 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 the level **k** cache must fetch it from level **k+1**,
    e.g., block 12
  - If the level-k cache is full, then some current block must be replaced (**evicted**). Which one is the "victim"?
    - Here, we pick 4; same column as 12
    - 4 is "dirty", need to write back to k+1
    - More on this next lecture

Outline

- **Locality of Reference**
- Cache Organization
- Associativity

Caching speeds up code

- Cache: smaller, faster storage device that keeps copies of a subset of the data in a larger, slower device
  - If the data we access is already in the cache, we win!
  - Can get access time of faster memory, with overall capacity of larger

- But how do we decide which data to keep in the cache?
  - Can we predict which data is likely to be necessary in the future?

Locality

- Goal: predict which data the CPU will want to access
  - So we can bring it to (and keep it in!) fast memory
  - Problem: memory is huge! (billions of bytes) how do you decide which to save?

- Principle of Locality
  - Programs tend to access data in predictable ways

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

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

Types of locality practice

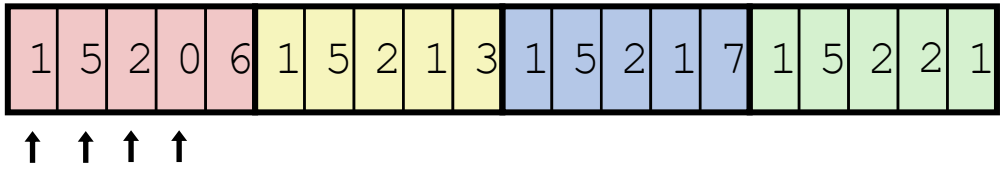
- Temporal locality
  - Recently referenced items are likely to be referenced in the near future
- Spatial locality
  - Items with nearby addresses tend to be referenced close together in time

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

- Quiz: what kind of locality?
  - Data
    - Reference array elements in succession: **Spatial locality**
    - Reference sum each iteration: **Temporal locality**
  - Instructions
    - Execute instructions in sequence: **Spatial locality**
    - Cycle through loop repeatedly: **Temporal locality**

Locality example

- Can get a sense for whether a function has good locality just by looking at its memory access patterns
- Does this function have good locality?




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

Temporal or spatial locality?

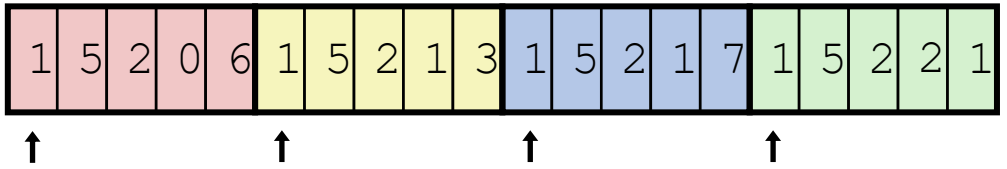
Spatial: accesses to array
Temporal: accesses to sum

- ***Yes!***
  - Array is accessed in same row-major order in which it is stored in memory
  - **a** through **a+3** , **a+4** through **a+7**, **a+8** through **a+11**, etc.

Locality example

- Does this function have good locality?

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



- ***No!***
  - Scans array column-wise instead of row-wise
  - **a** through **a+3**, then **a+4*N** through **a+4*N+3**, etc.
  - Holy jumping around memory Batman!

- More on that in a later lectures

Locality to the Rescue!

- How can we exploit locality to bridge the CPU-memory gap?
  - Use it to determine which data to put in a cache!

- Spatial locality
  - When level *k* needs a byte from level *k+1*, don't just bring one byte
  - Bring neighboring bytes as well!
  - Good chances we'll need them too in the near future

- Temporal locality
  - Anything accessed goes in the cache, and we'll try to keep it there for a while
  - Good chances we'll need it again in the near future

- Result: most accesses should be cache hits!
  - Memory system: size of largest memory, with speed close to that of fastest memory

Cache misses will still happen

- Only 1%-0.1% of memory is in the cache
  - So we'll sometimes need to access the other 99%

- When evaluating system performance, the most important part is understanding *why* a cache miss occurred

What causes a cache miss?

- **Cold (compulsory) miss**
  - Cold misses occur when a block is accessed for the first time
  - No one ever accessed it, so there wasn't any reason to bring it into cache

- **Capacity miss**
  - Occurs when the set of active cache blocks (*working set*) is larger than the cache
  - There's no way the working set can all fit in the cache, so there will be misses

- **Conflict miss**
  - In most caches, blocks cannot be stored in any available slot
  - If two blocks need to go in the same slot, need to evict the old one to store the new!
  - If after that, we need to access the old block, conflict miss!
    - We had a conflict, evicted a block, and now we miss trying to access that block
  - **Note**: can happen even when there is "room" elsewhere in the cache!
  - We'll show examples of this next lecture

Break + Question

Miss types: Cold, Capacity, Conflict

- When you first start up a program, it runs really slowly for a few seconds. What kind of cache misses are occurring?

- When you have too many browser tabs open and active, all of the tabs run more slowly. What kind of cache misses are occurring?

Break + Question

Miss types: Cold, Capacity, Conflict

- When you first start up a program, it runs really slowly for a few seconds. What kind of cache misses are occurring?
  - Cold (aka Compulsory) misses. The data has never been loaded before!
- When you have too many browser tabs open and active, all of the tabs run more slowly. What kind of cache misses are occurring?
  - Capacity misses. You have too much data to fit it all in the cache
    - Could be Conflict misses as well, but probably not

Outline

- Locality of Reference
- **Cache Organization**
- Associativity

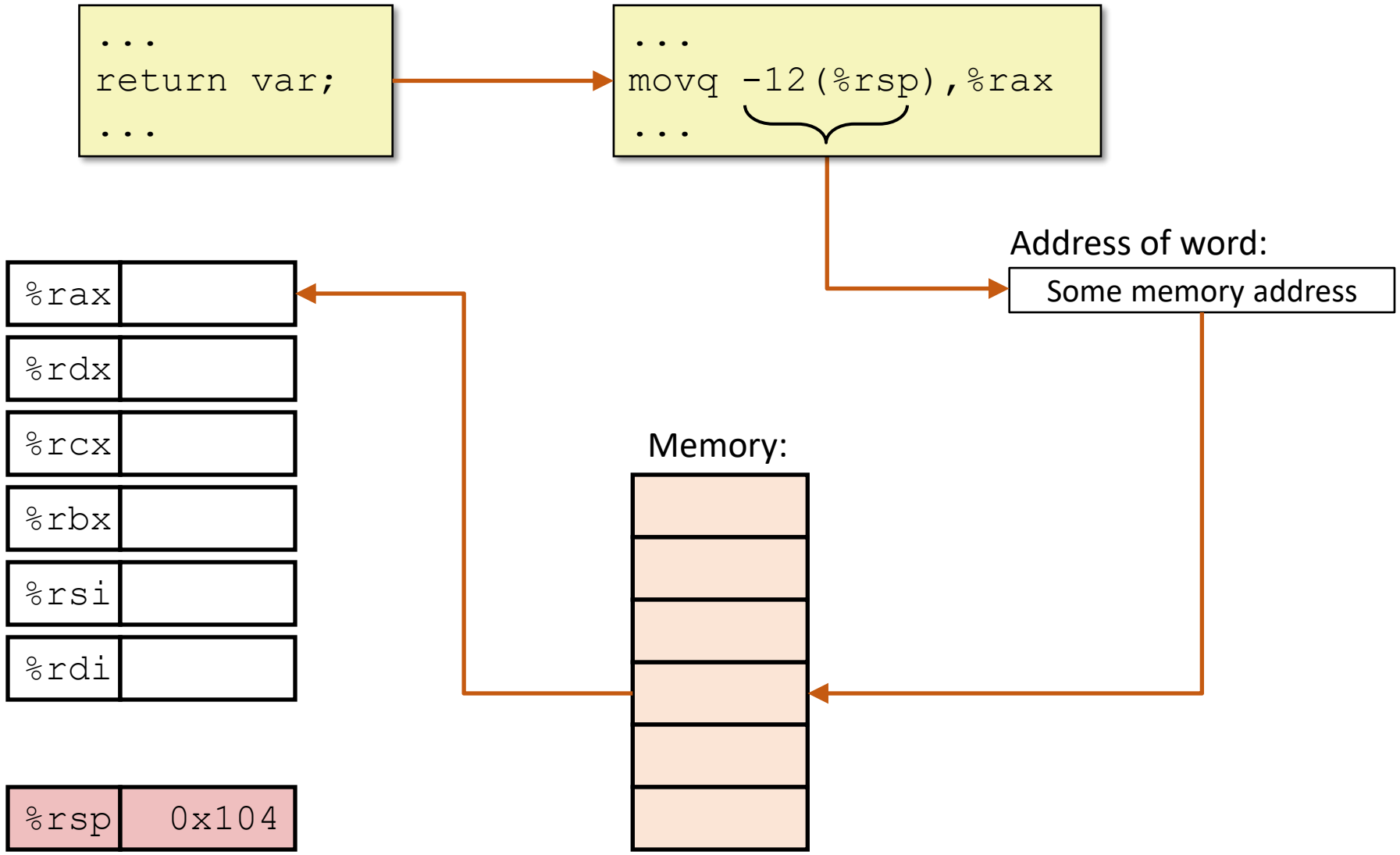
Cache memories

- A specific instance of the general principle of caching
  - Small, fast SRAM-based memories between CPU and main memory
  - Can include multiple levels
    - L1 = small, but really fast, L2 = larger, slower, L3, etc.

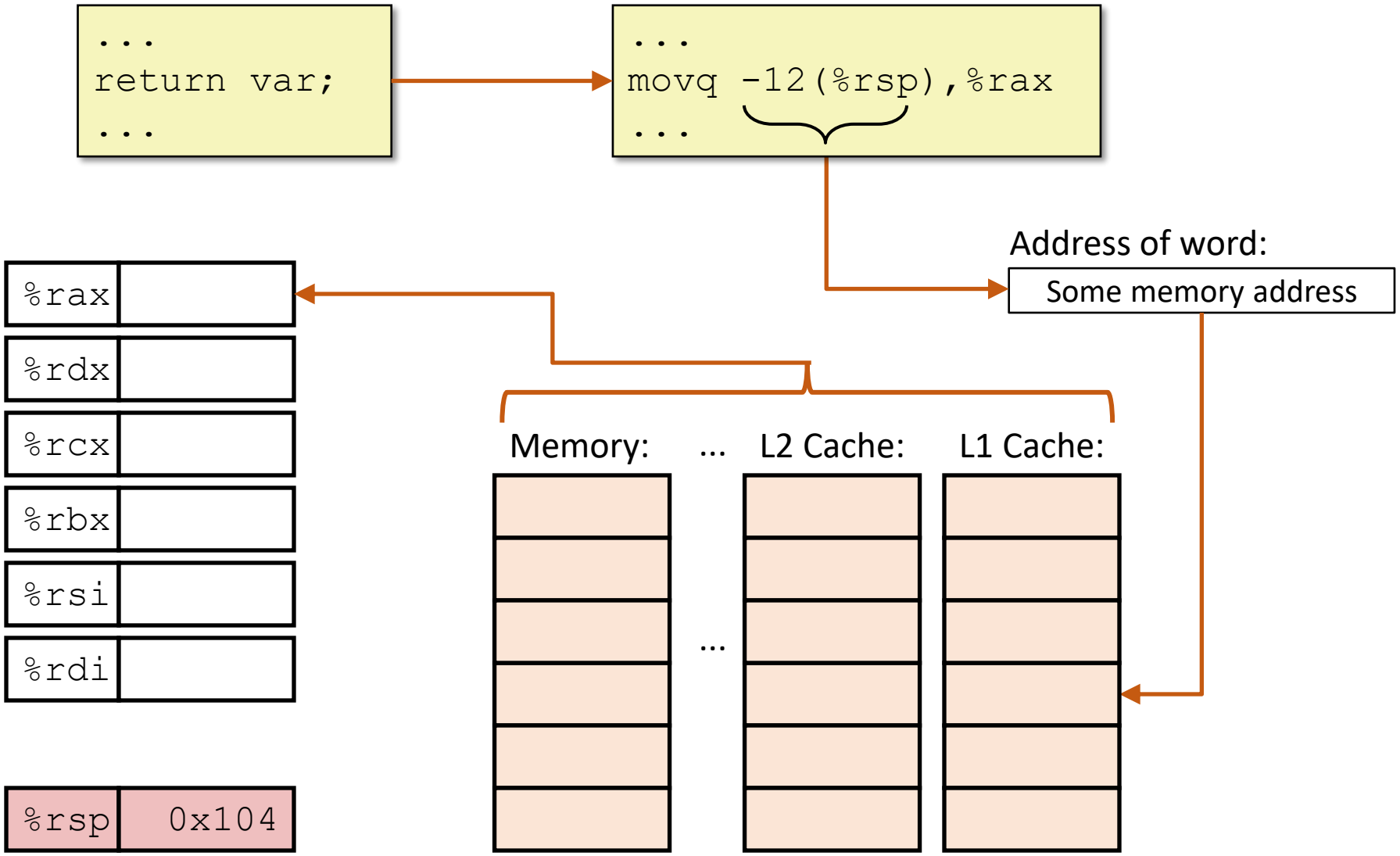
- CPU looks for data in caches first
  - e.g., L1, then L2, then L3, then finally in main memory as a last resort

- Mechanisms we'll see today are implemented in *hardware*

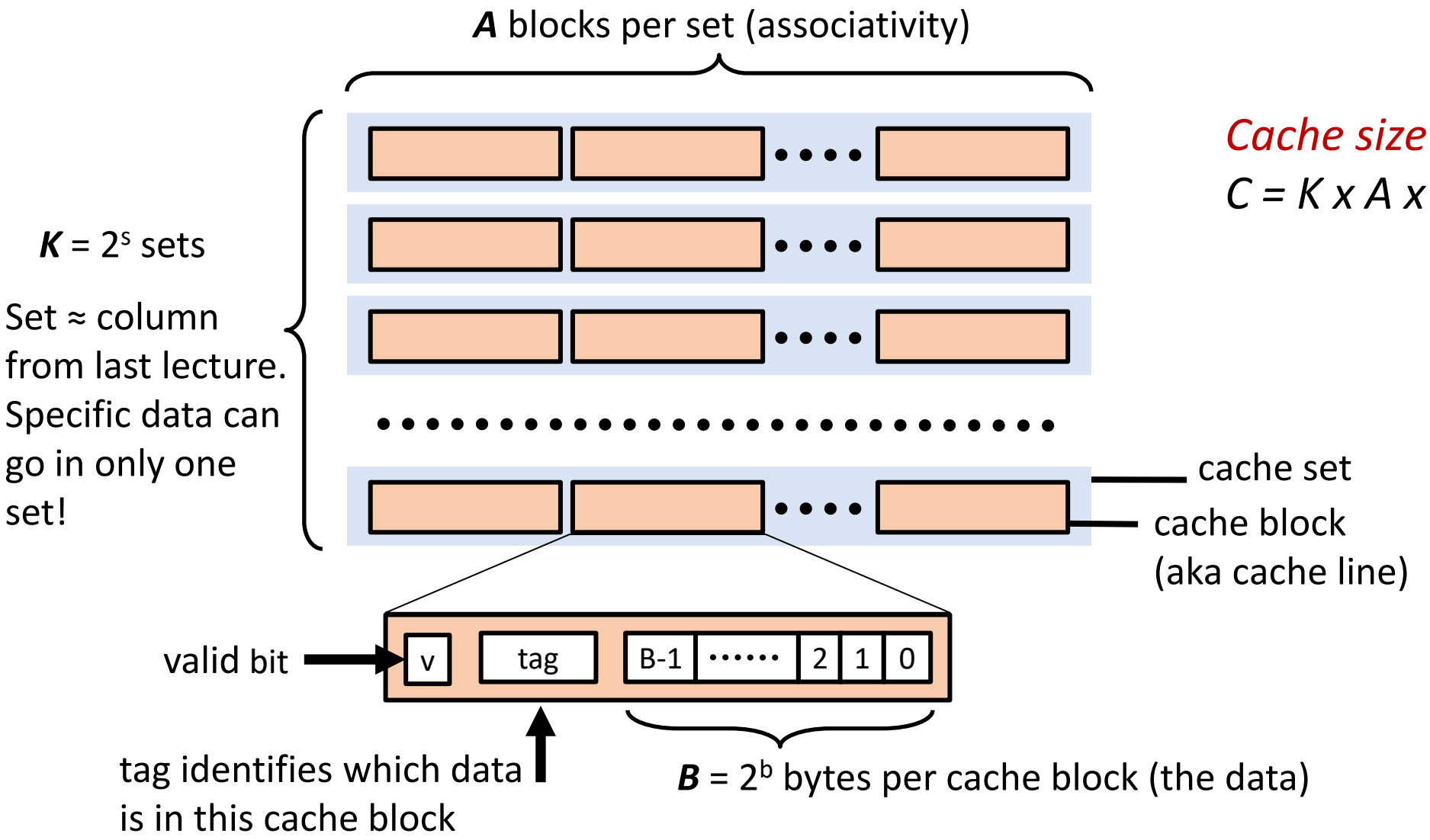
How You Probably Thought a Memory Access Worked



How a Memory Access Actually Works



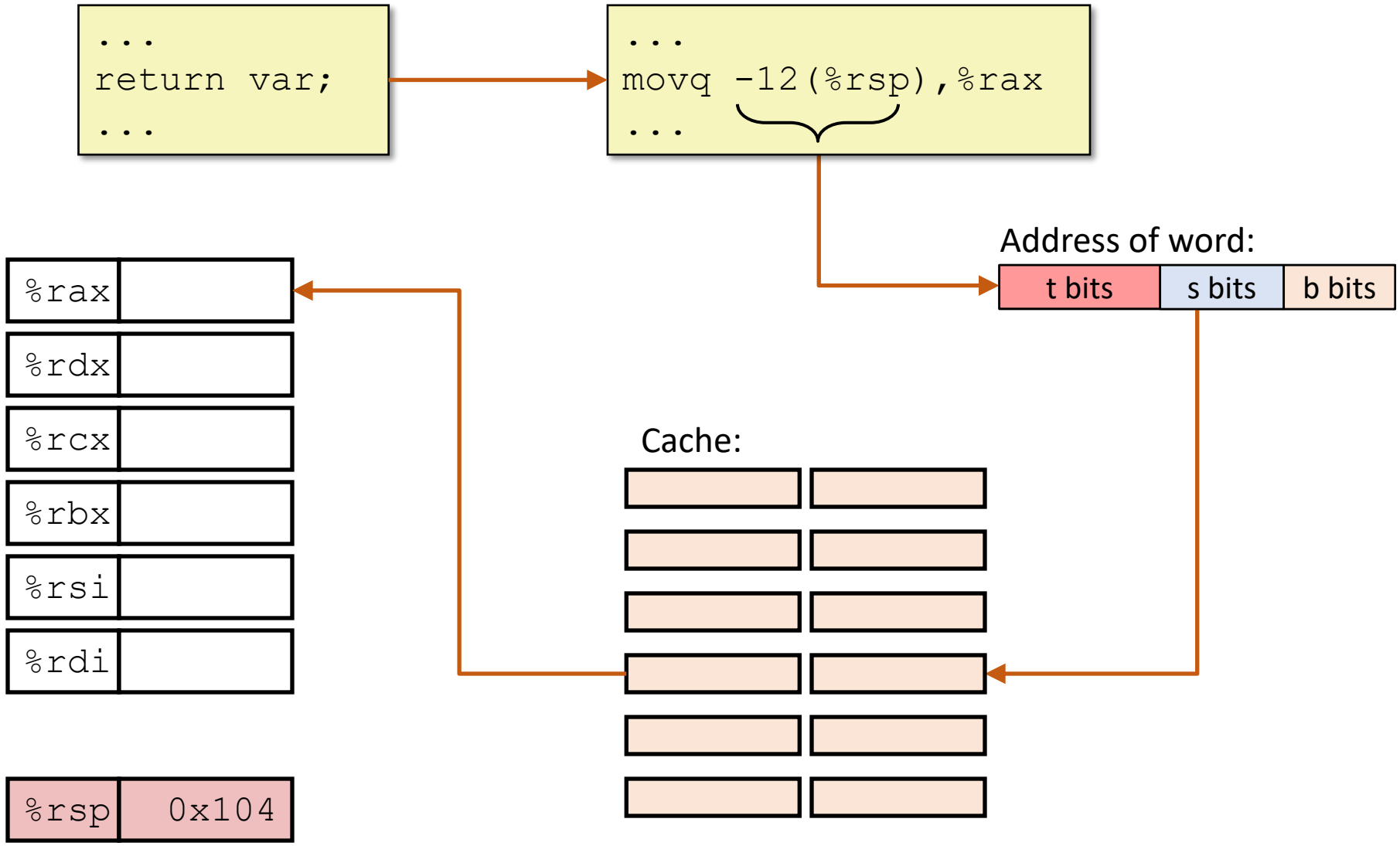
General Cache Organization (S, A, B)



Cache size:

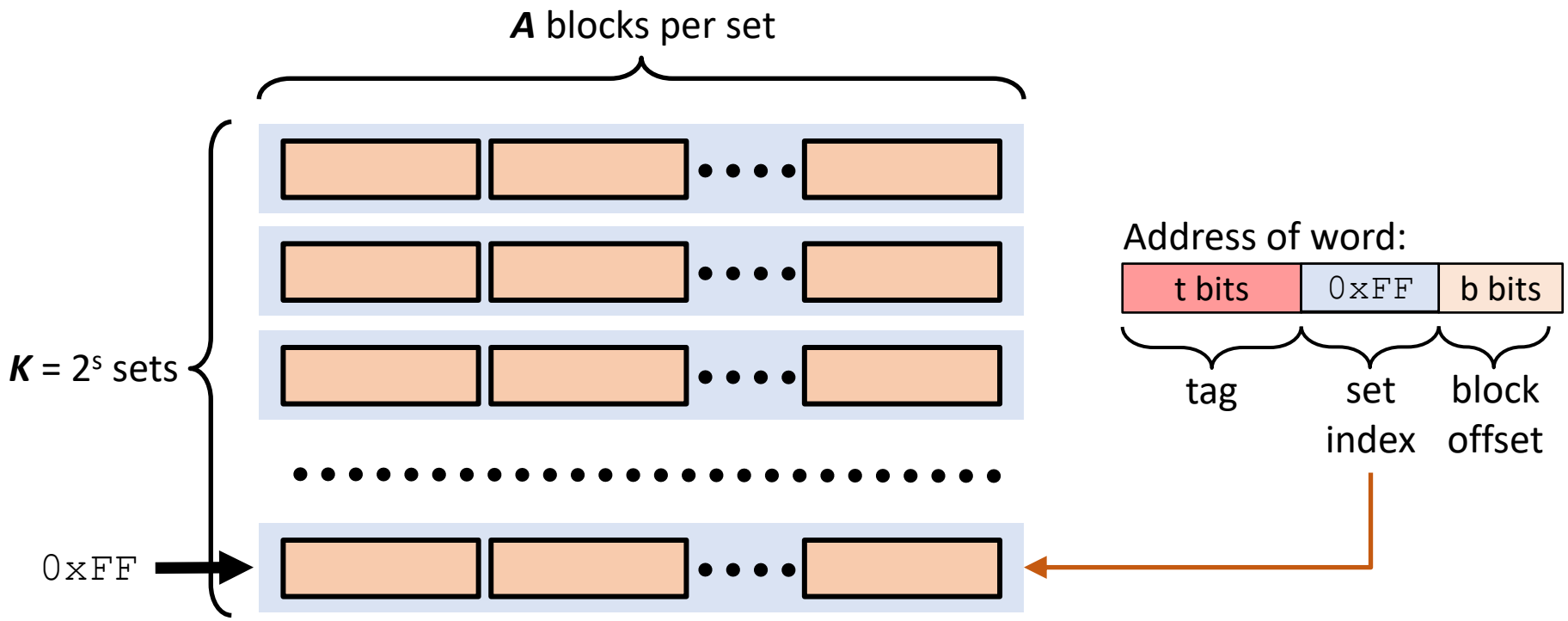
C = K x A x B data bytes

Cache Access



Cache Read (1): Locate Set

- Locate set

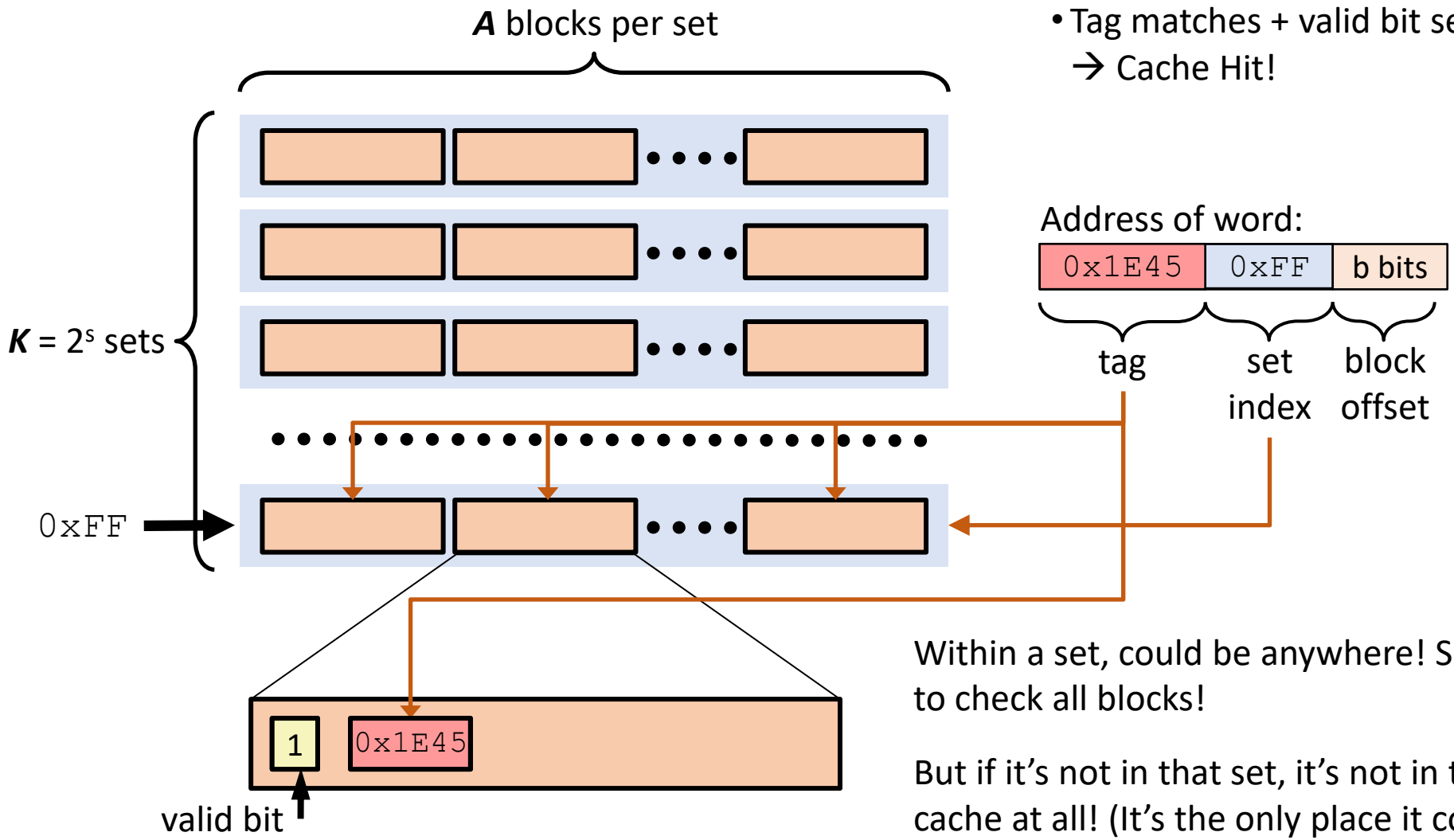


Each address maps to a particular set!
Data has to be stored at that particular set!

Even if that set is full and there would be space elsewhere!
(That's where conflict misses come from.)

Cache Read (2): Tag Match + Valid

- Locate set
- Locate block in set
- Tag matches + valid bit set
  → Cache Hit!

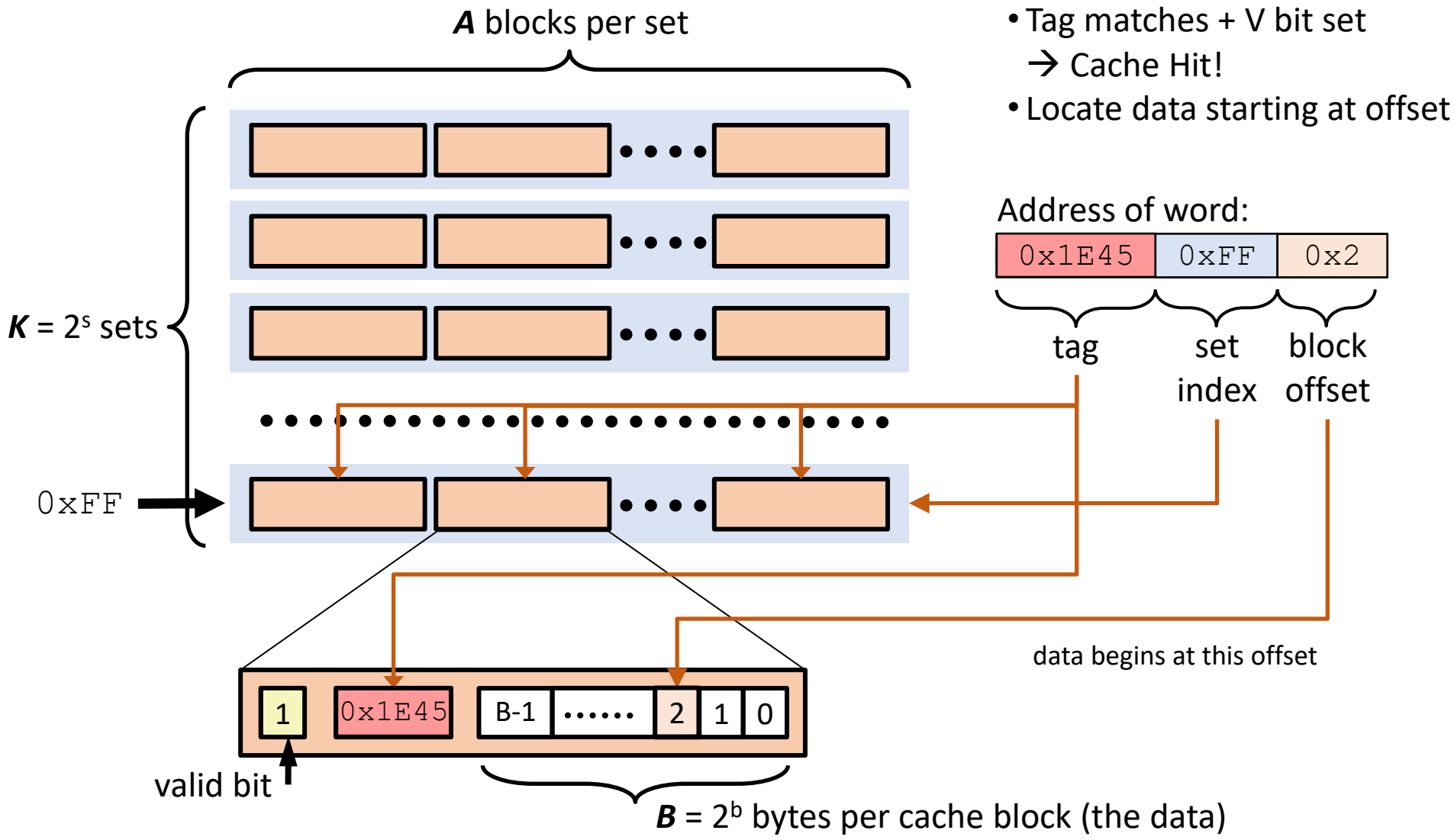


Within a set, could be anywhere! So, need to check all blocks!

But if it's not in that set, it's not in the cache at all! (It's the only place it could be.)

Cache Read (3): Block Offset

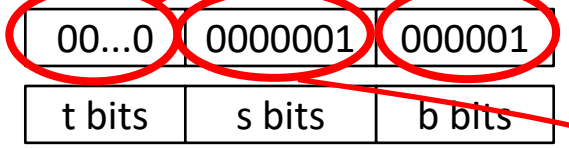
- Locate set
- Locate block in set
- Tag matches + V bit set
  → Cache Hit!
- Locate data starting at offset



Example: 128 sets, 64 bytes per block

Goal: Get byte M[65] from cache

$65_{10} = 100\ 0001_2$

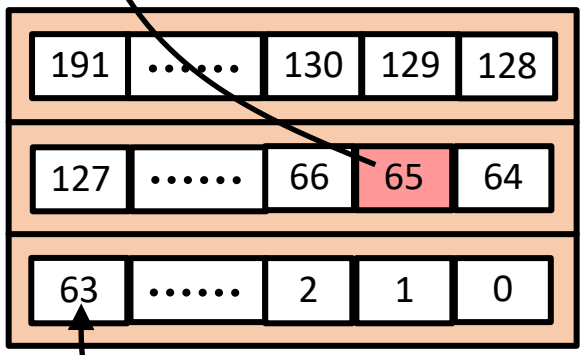


64 bytes per block → b = 6 bits

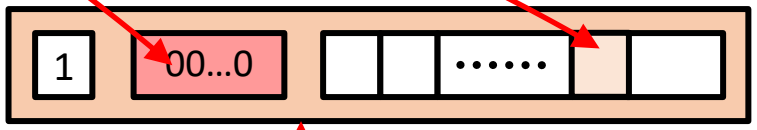
128 sets → s = 7 bits

remaining address bits → t bits

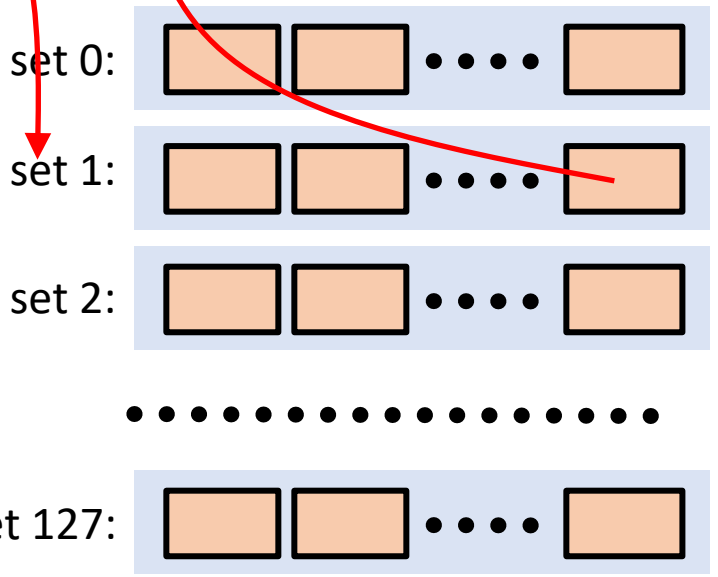
Memory:



address of a byte in memory



A blocks per set

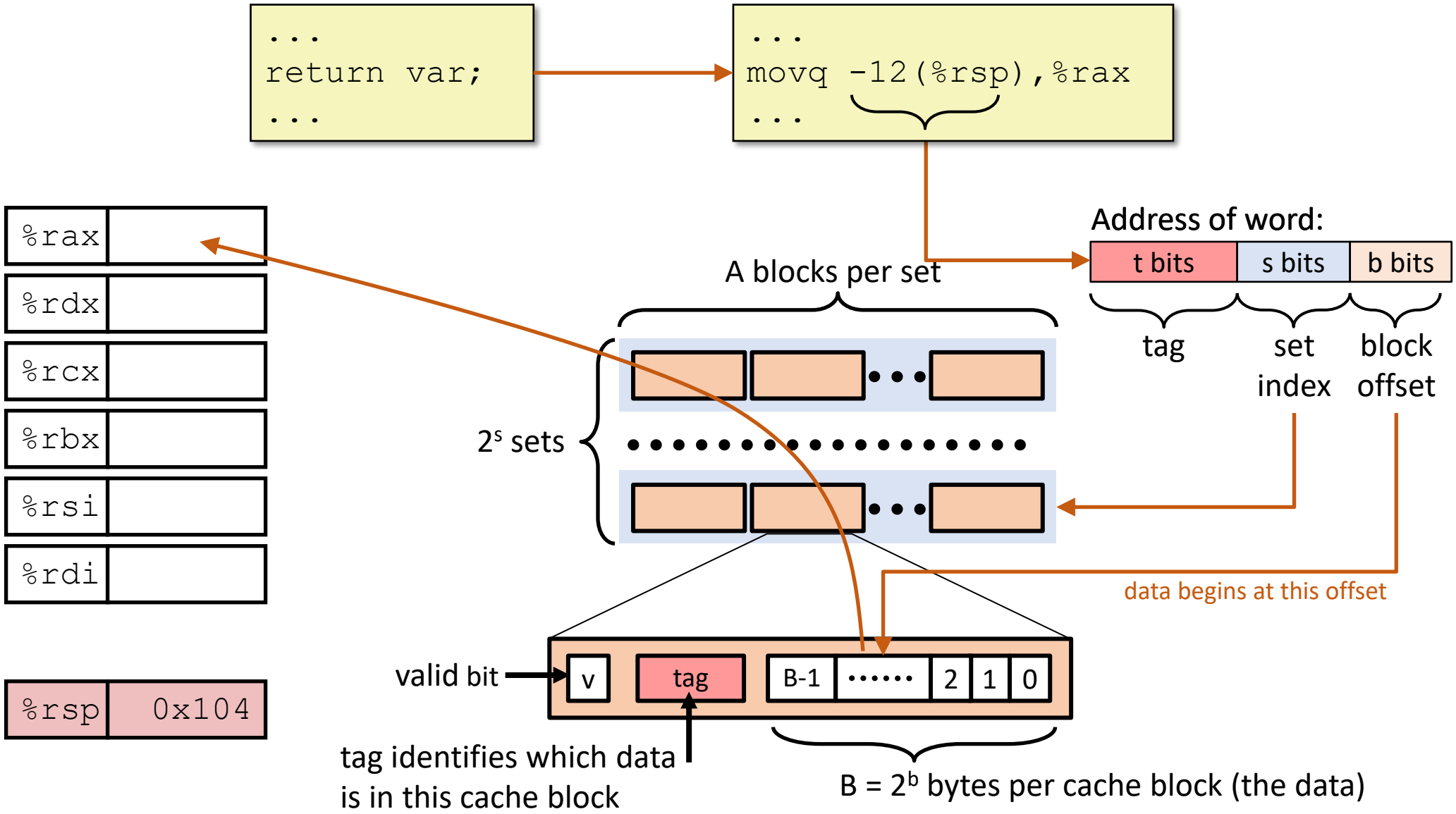


QUIZ #1: which set should we look in?

QUIZ #2: which tag are we looking for?

QUIZ #3: which byte within the block is the one that we want?

Cache access overview

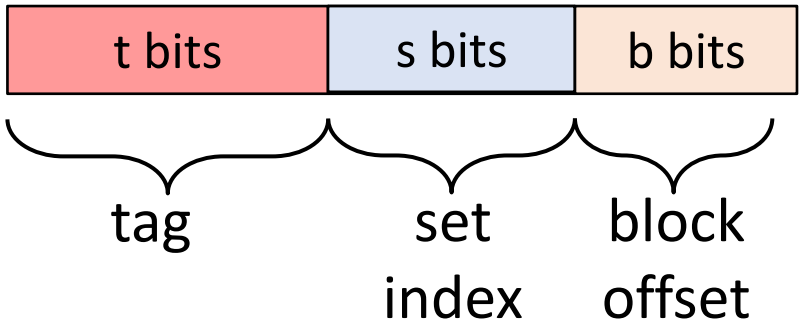


Break + Question

- 64-bit, byte-addressed system
- 32 kB cache
  - 512 sets and 64-byte blocks

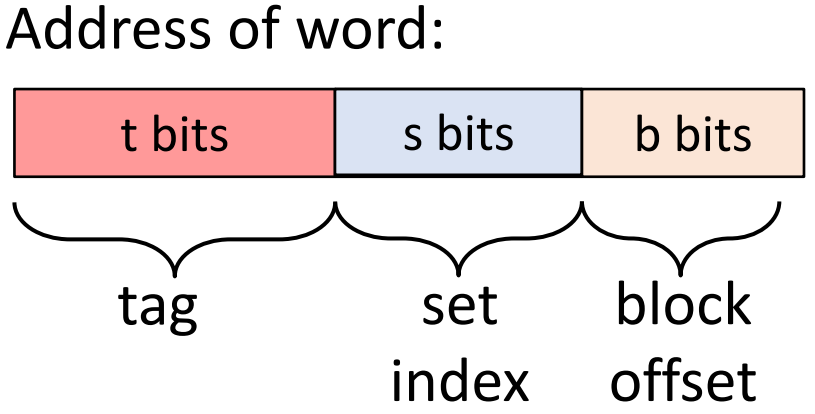
- How many bits for Tag?
  - A: 6 bits
  - B: 9 bits
  - C: 17 bits
  - D: 49 bits

Address of word:



Break + Question

- 64-bit, byte-addressed system
- 32 kB cache
  - 512 sets and 64-byte blocks



- How many bits for Tag? (6 bits for block, 9 bits for set)
  - A: 6 bits
  - B: 9 bits
  - C: 17 bits
  - **D: 49 bits** (Tag is remaining bits. 64 - 6 - 9 = 49)

What about writes?

- Multiple copies of data exist:
  - L1, L2, Main Memory, Disk

  - Don’t want them to get (or at least not to stay) out of sync!
    - Otherwise, who do you believe?

- Multiple configuration options that a cache could have

Write configurations

- What to do on a write-hit?
  - ***Write-through*** (write immediately to memory)
  - ***Write-back*** (delay write until we evict this cache block)
    - Need a dirty bit (indicate if block differs from memory)
    - We had an example of that last lecture

- What to do on a write-miss?
  - ***Write-allocate*** (load into cache, update block in cache)
    - Good if more writes to the location follow
  - ***No-write-allocate*** (writes immediately to memory, doesn't bring into cache)

- Typical combinations
  - **Write-back + Write-allocate ← by far the most common**
  - Write-through + No-write-allocate

Outline

- Locality of Reference
- Cache Organization
- **Associativity**

Cache memory associativity

- When designing a cache, a number of parameters to choose
  - Total size (C), cache block size (B), number of sets (K), …
- The most interesting one: associativity (A)
  - i.e., how many cache blocks per set
  - Has a significant impact on effectiveness (and complexity!)

Associativity choices

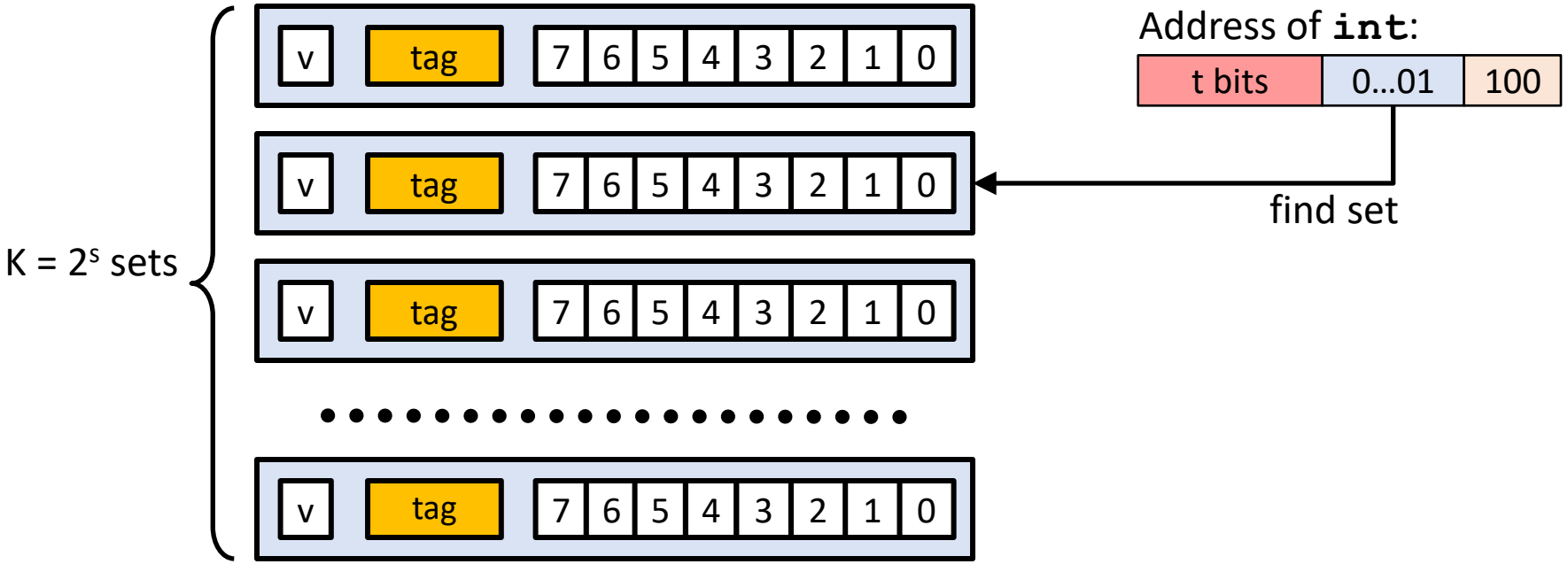
- Associativity 1 → **direct-mapped caches**
  - One cache block per set, data blocks can only go in that one cache block
  - Whenever we place data in a set, must evict whatever is there

- Associativity >1 → **set-associative caches**
  - Can keep multiple blocks that would map to the same set

- Single set → **fully-associative caches**
  - Any block can go anywhere, 1 big set, tag is all that matters
  - Very rare for cache memories due to expensive hardware

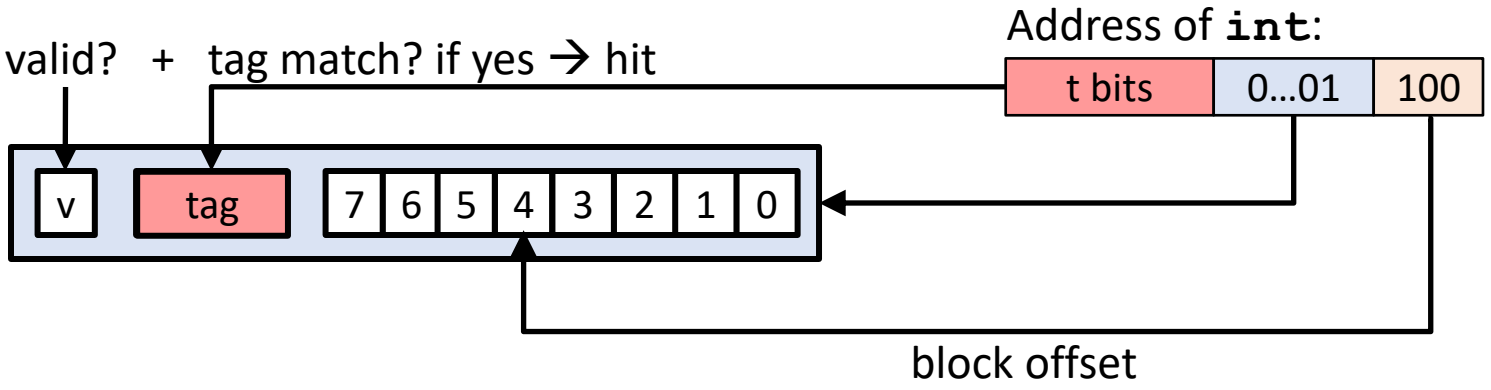
Direct-mapped cache (associativity = 1)

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



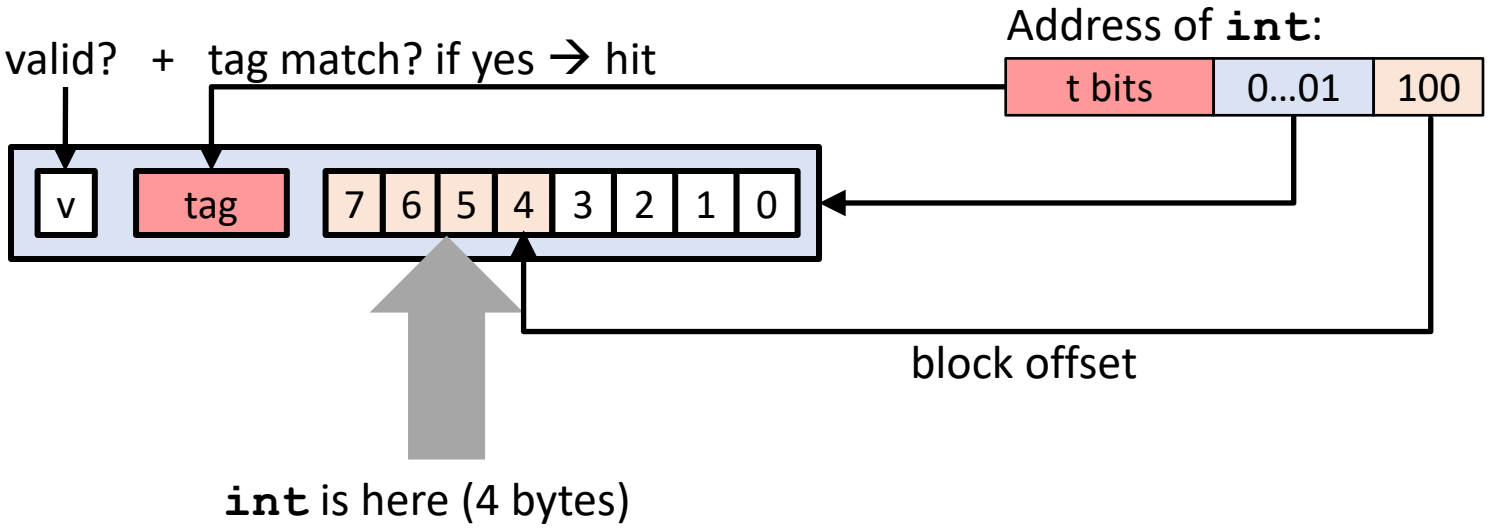
Direct-mapped cache (associativity = 1)

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



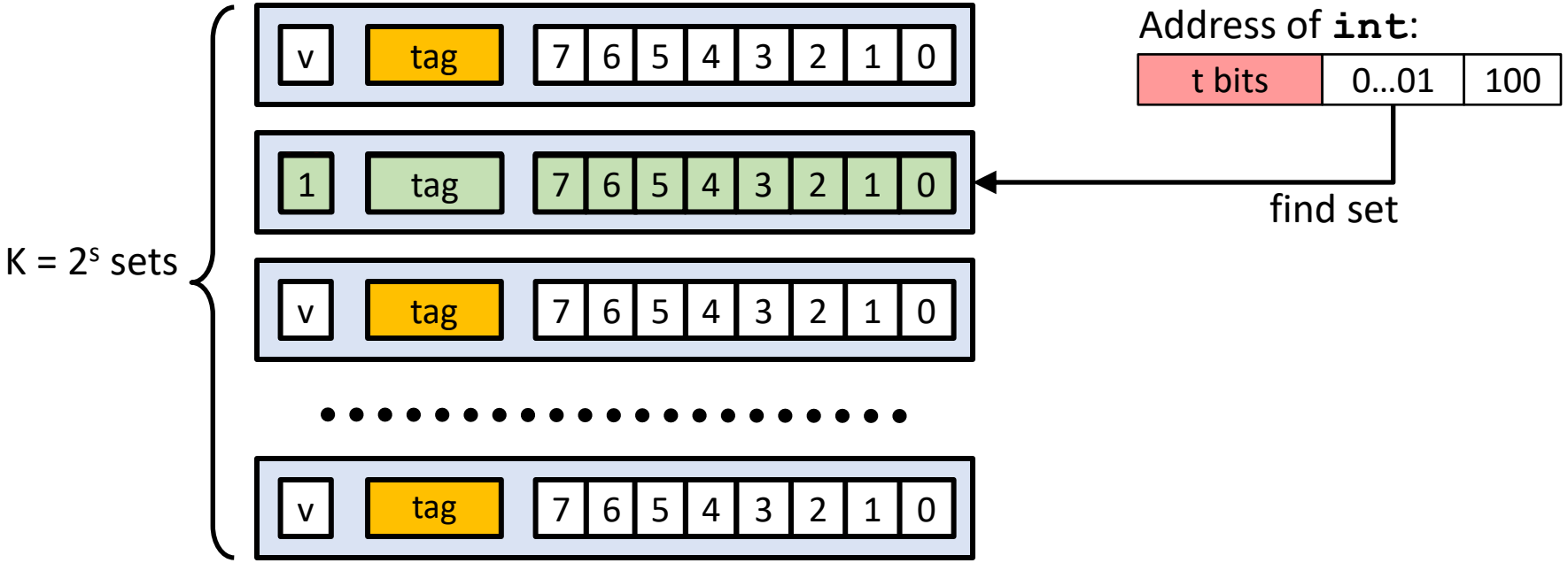
Direct-mapped cache (associativity = 1)

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



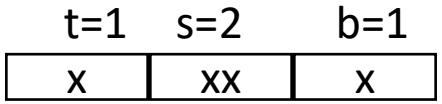
Direct-mapped cache (associativity = 1)

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



If tag doesn't match or valid bit is not set: cache miss!
→ old block is evicted and replaced with currently requested one

Direct-mapped cache simulation



M=16 addresses,
byte-addressable
B=2 bytes/block
K=4 sets
A=1 blocks/set

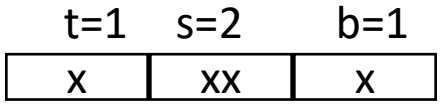
Memory

offset	**0**	**1**	**2**	**3**
0x0	m[0]	m[1]	m[2]	m[3]
0x4	m[4]	m[4]	m[6]	m[7]
0x8	m[8]	m[9]	m[10]	m[11]
0xC	m[12]	m[13]	m[14]	m[16]

m[0] is the value of memory at address 0

The actual values are irrelevant for this problem

Direct-mapped cache simulation



M=16 addresses,
byte-addressable
B=2 bytes/block
K=4 sets
A=1 blocks/set

Every Block in the cache holds two bytes, so we can split memory into blocks

Every two bytes is a block. And blocks are aligned (so bytes 1 and 2 are separate blocks)

Memory

offset	**0**	**1**	**2**	**3**
0x0	m[0]	m[1]	m[2]	m[3]
0x4	m[4]	m[4]	m[6]	m[7]
0x8	m[8]	m[9]	m[10]	m[11]
0xC	m[12]	m[13]	m[14]	m[16]

m[0] is the value of memory at address 0

The actual values are irrelevant for this problem

Direct-mapped cache simulation

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

M=16 addresses,
byte-addressable
B=2 bytes/block
K=4 sets
A=1 blocks/set

Memory

offset	**0**	**1**	**2**	**3**
0x0	m[0]	m[1]	m[2]	m[3]
0x4	m[4]	m[4]	m[6]	m[7]
0x8	m[8]	m[9]	m[10]	m[11]
0xC	m[12]	m[13]	m[14]	m[16]

Address trace
(reads, one byte per read):

0 [0 00 0_2] miss

1 [0 00 1_2] hit

7 [0 11 1_2] miss

8 [1 00 0_2] miss

1 [0 00 1_2] miss

	v	tag	block	
set 00_2	1	0	m[1]	m[0]
set 01_2	0			
set 10_2	0			
set 11_2	1	0	m[7]	m[6]

What are the types of each miss here?

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

M=16 addresses,
byte-addressable
B=2 bytes/block
K=4 sets
A=1 blocks/set

Memory

offset	**0**	**1**	**2**	**3**
0x0	m[0]	m[1]	m[2]	m[3]
0x4	m[4]	m[4]	m[6]	m[7]
0x8	m[8]	m[9]	m[10]	m[11]
0xC	m[12]	m[13]	m[14]	m[16]

Address trace
(reads, one byte per read):

0 [0 00 0_2]	miss	Compulsory Miss
1 [0 00 1_2]	hit	
7 [0 11 1_2]	miss	Compulsory Miss
8 [1 00 0_2]	miss	Compulsory Miss
1 [0 00 1_2]	miss	Conflict Miss

	v	tag	block	
set 00_2	1	0	m[1]	m[0]
set 01_2	0			
set 10_2	0			
set 11_2	1	0	m[7]	m[6]

Options:
- Compulsory
- Capacity
- Conflict

***Conflict misses**:*
There is “room” in the cache,
but two blocks map to the same set;
one evicts the other!

Pause for questions on direct-mapped caches

Associativity choices

- Associativity 1 → **direct-mapped caches**
  - One cache block per set, blocks can only go in that one block
  - Whenever we place data in a set, must evict whatever is there

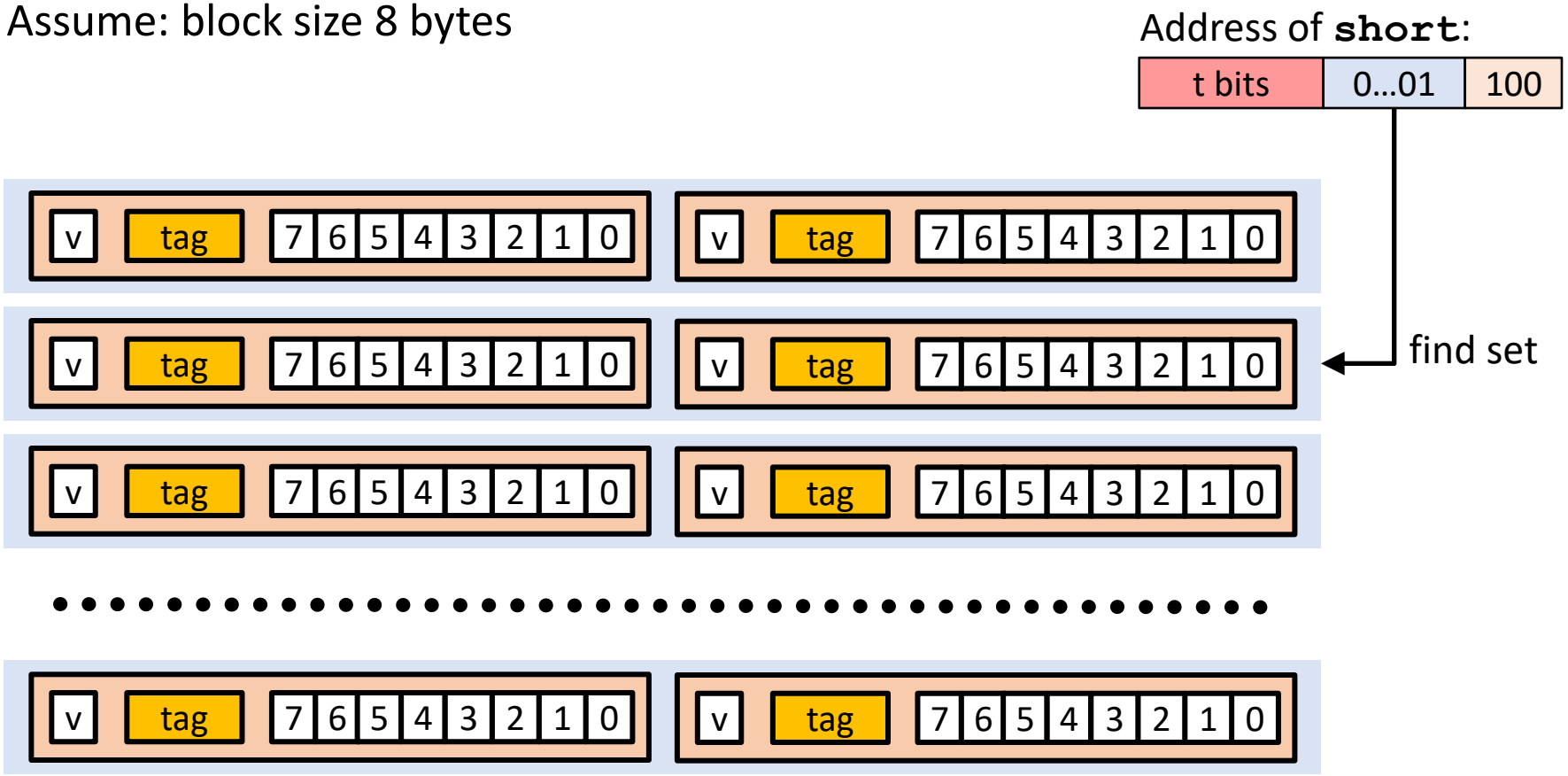
- Associativity >1 → **set-associative caches**
  - Can keep multiple cache blocks that would map to the same set

- Single set → **fully-associative caches**
  - Any cache block can go anywhere, 1 big set, tag is all that matters
  - Very rare for cache memories due to expensive hardware

2-way set-associative cache (associativity = 2)

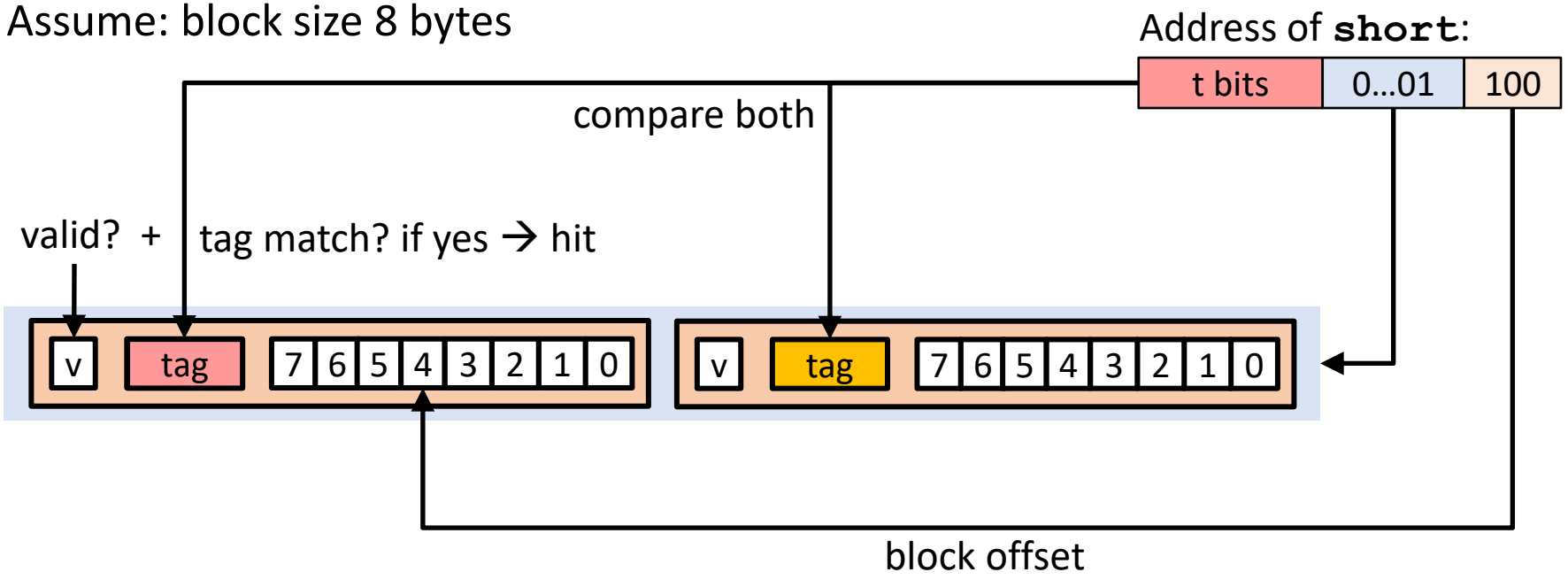
A = 2: Two blocks per set

Assume: block size 8 bytes



2-way set-associative cache (associativity = 2)

A = 2: Two blocks per set
Assume: block size 8 bytes

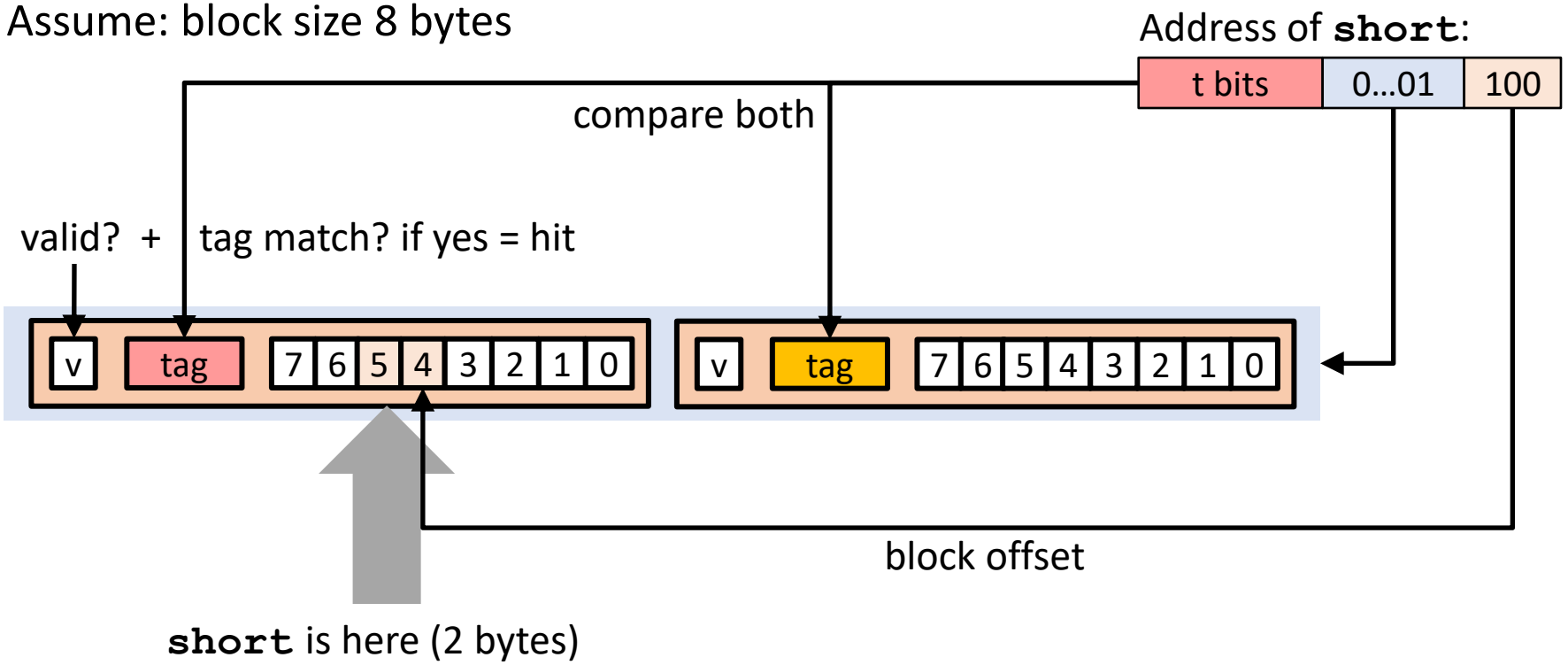


The data we want is either on the left, or on the right, or not in the cache at all.
It can't be anywhere else! Addresses map to a single set!

2-way set-associative cache (associativity = 2)

A = 2: Two blocks per set

Assume: block size 8 bytes



If no match:

- One block in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), …
  - More clever → lower miss rate, but harder to implement in hardware

2-way set-associative cache simulation

M=16 addresses, byte-addressable,
B=2 bytes/block, K=2 sets, A=2 blocks/set

Same total size and block size as before.
Associativity (and thus # of sets) changed.

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

Address trace (reads, one byte per read):

0	[00 0 0_2]	miss
1	[00 0 1_2]	hit
7	[01 1 1_2]	miss
8	[10 0 0_2]	miss
0	[00 0 0_2]	hit

The same address sequence in the direct mapped cache resulted in:

miss
hit
miss
miss
miss

Higher associativity =
Less likely to have to evict!

Temporal locality: want data in cache to *stay* in cache!

Set 0

v	Tag	Block
1	00	M[0-1]
1	10	M[8-9]

Set 1

v	Tag	Block
1	01	M[6-7]
0		

Pause for questions on set-associative caches

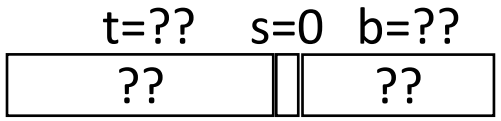
Fully-associative caches

- What changes with fully-associative caches?
  - Anything can go anywhere
  - Only one set (s = 0 bits)

- Otherwise, same steps as for a set-associative cache
  - Compare tag against all blocks in the set

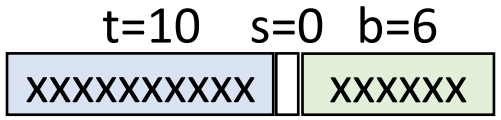
Fully-Associative Cache Practice

- Fully-associative cache on a 16-bit system
  - One set (fully associative!)
  - Eight, 64-byte blocks



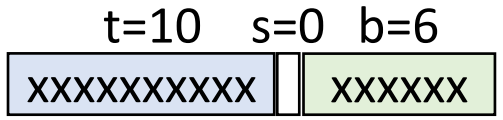
Fully-Associative Cache Practice

- Fully-associative cache on a 16-bit system
  - One set (fully associative!)
  - Eight, 64-byte blocks



Fully-Associative Cache Practice

- Fully-associative cache on a 16-bit system
  - One set (fully associative!)
  - Eight, 64-byte blocks

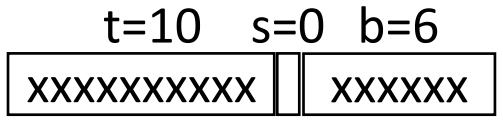


Tag: 0x000	Tag: 0x1FF	Tag: 0x010	Tag: 0x011	Tag: 0x050	Tag: 0x051	Tag: 0x052	Tag: 0x300

- Are the following addresses in the cache?
  - 0x0400
  - 0x0410
  - 0xC002
  - 0xC048

Fully-Associative Cache Practice

- Fully-associative cache on a 16-bit system
  - One set (fully associative!)
  - Eight, 64-byte blocks

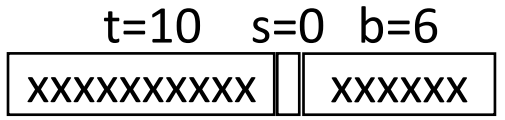


Tag: 0x000	Tag: 0x1FF	Tag: 0x010	Tag: 0x011	Tag: 0x050	Tag: 0x051	Tag: 0x052	Tag: 0x300

- Are the following addresses in the cache?
  - 0x0400⇨0b0000 0100 0000 0000
  - 0x0410⇨0b0000 0100 0001 0000
  - 0xC002⇨0b1100 0000 0000 0010
  - 0xC048⇨0b1100 0000 0100 1000

Fully-Associative Cache Practice

- Fully-associative cache on a 16-bit system
  - One set (fully associative!)
  - Eight, 64-byte blocks

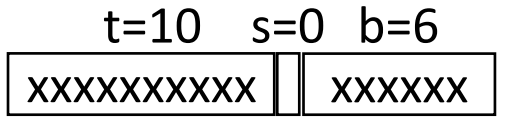


Tag: 0x000	Tag: 0x1FF	Tag: 0x010	Tag: 0x011	Tag: 0x050	Tag: 0x051	Tag: 0x052	Tag: 0x300

- Are the following addresses in the cache?
  - 0x0400⇨0b0000 0100 0000 0000
  - 0x0410⇨0b0000 0100 0001 0000
  - 0xC002⇨0b1100 0000 0000 0010
  - 0xC048⇨0b1100 0000 0100 1000

Break + Question

- Fully-associative cache on a 16-bit system
  - One set (fully associative!)
  - Eight, 64-byte blocks



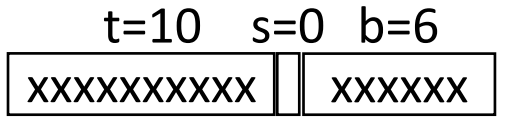
Tag: 0x000	Tag: 0x1FF	Tag: 0x010	Tag: 0x011	Tag: 0x050	Tag: 0x051	Tag: 0x052	Tag: 0x300

- Are the following addresses in the cache?
  - 0x0400⇨0b0000 0100 0000 0000
  - 0x0410⇨0b0000 0100 0001 0000
  - 0xC002⇨0b1100 0000 0000 0010
  - 0xC048⇨0b1100 0000 0100 1000

You figure out the rest!

Break + Question

- Fully-associative cache on a 16-bit system
  - One set (fully associative!)
  - Eight, 64-byte blocks

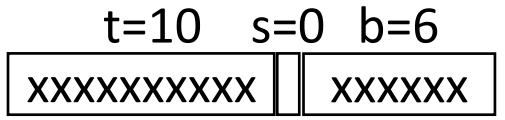


Tag: 0x000	Tag: 0x1FF	Tag: 0x010	Tag: 0x011	Tag: 0x050	Tag: 0x051	Tag: 0x052	Tag: 0x300

- Are the following addresses in the cache?
  - 0x0400⇨0b0000 0100 0000 0000 → Tag 0x010 **HIT**
  - 0x0410⇨0b0000 0100 0001 0000 → Tag 0x010 (same block!) **HIT**
  - 0xC002⇨0b1100 0000 0000 0010
  - 0xC048⇨0b1100 0000 0100 1000

Break + Question

- Fully-associative cache on a 16-bit system
  - One set (fully associative!)
  - Eight, 64-byte blocks



Tag: 0x000	Tag: 0x1FF	Tag: 0x010	Tag: 0x011	Tag: 0x050	Tag: 0x051	Tag: 0x052	Tag: 0x300

- Are the following addresses in the cache?
  - 0x0400⇨0b0000 0100 0000 0000 → Tag 0x010 **HIT**
  - 0x0410⇨0b0000 0100 0001 0000 → Tag 0x010 (same block!) **HIT**
  - 0xC002⇨0b1100 0000 0000 0010 → Tag 0x300 **HIT**
  - 0xC048⇨0b1100 0000 0100 1000 → Tag 0x301 (different block!) **MISS**

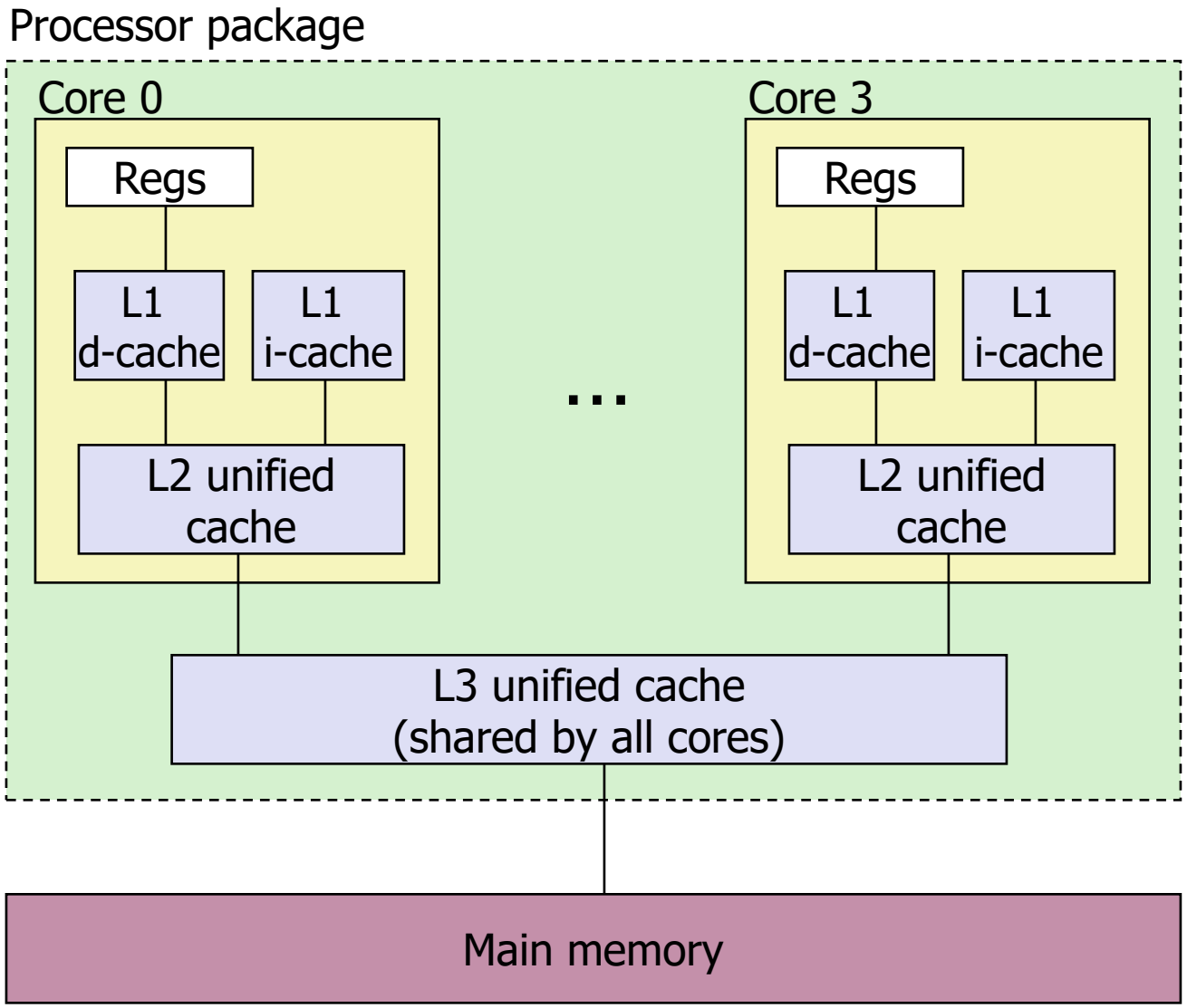
Associativity Pros and Cons

- Direct-mapped
  - Simplest to implement: look-up compares tag with 1 cache block
    → requires fewer transistors, which can be used elsewhere on the chip
  - Conflicts can easily lead to *thrashing*
    - Two cache blocks map to the same set, program needs both, and they keep kicking each other out of the cache. Lots of misses. Bad times.

- Set-associative
  - More complex implementation: requires more (HW) tag comparators
  - Lower miss rate than direct-mapped caches (fewer conflict misses)
    - 2-way is a significant improvement over direct-mapped
    - 4-way is a more modest improvement over 2-way, and so on

- Fully-associative
  - One comparator per cache block in the cache means a LOT of hardware. Ouch.
    - Often a deal-breaker for hardware
  - Very low miss rate!

Intel Core i7 Cache Hierarchy



L1 i-cache and d-cache:
- 32 KB, 8-way,
- Access: 4 cycles

Keep separate caches for instructions and data. Don't want them to step on each other's toes!

L2 unified cache:
- 256 KB, 8-way,
- Access: 11 cycles

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

Last resort before going to main memory (slow!) So want this large and highly-associative, to have very few misses.

Block size: 64 bytes for all caches.

Outline

- Locality of Reference
- Cache Organization
- Associativity