Caches III

CSE 351 Autumn 2017

Instructor:

Justin Hsia

Teaching Assistants:

Lucas Wotton

Michael Zhang

Parker DeWilde

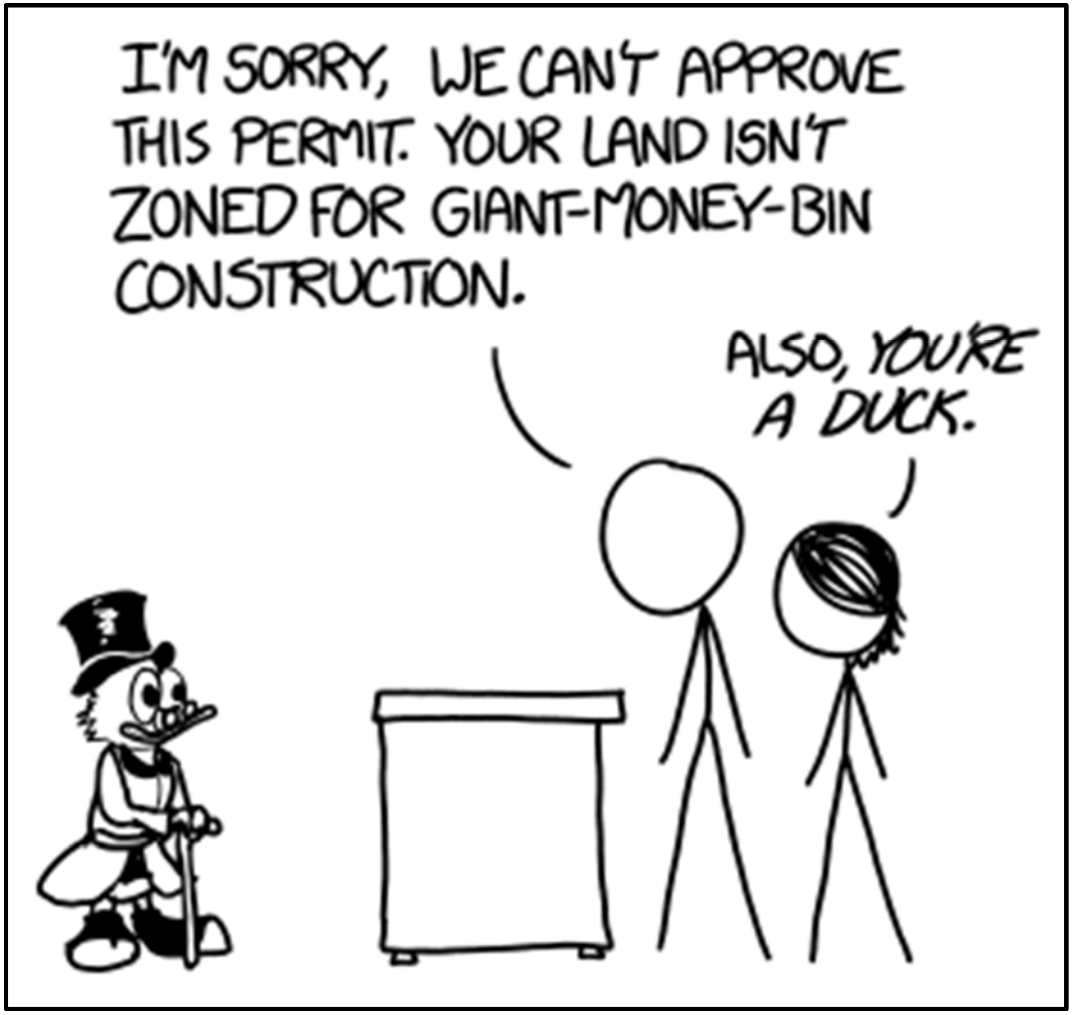
Ryan Wong

Sam Gehman

Sam Wolfson

Savanna Yee

Vinny Palaniappan


I'M SORRY, WE CAN'T APPROVE THIS PERMIT. YOUR LAND ISN'T ZONED FOR GIANT-MONEY-BIN CONSTRUCTION.
ALSO, YOU'RE A DUCK.

https://what-if.xkcd.com/111/

Administrivia

- Midterm regrade requests due end of tonight
- Lab 3 due Friday
- HW 4 is released, due next Friday (11/17)
- No lecture on Friday – Veteran’s Day!

Making memory accesses fast!

- Cache basics
- Principle of locality
- Memory hierarchies
- Cache organization
  - Direct-mapped (*sets*; index + tag)
  - **Associativity (*ways*)**
  - **Replacement policy**
  - **Handling writes**
- Program optimizations that consider caches

Associativity

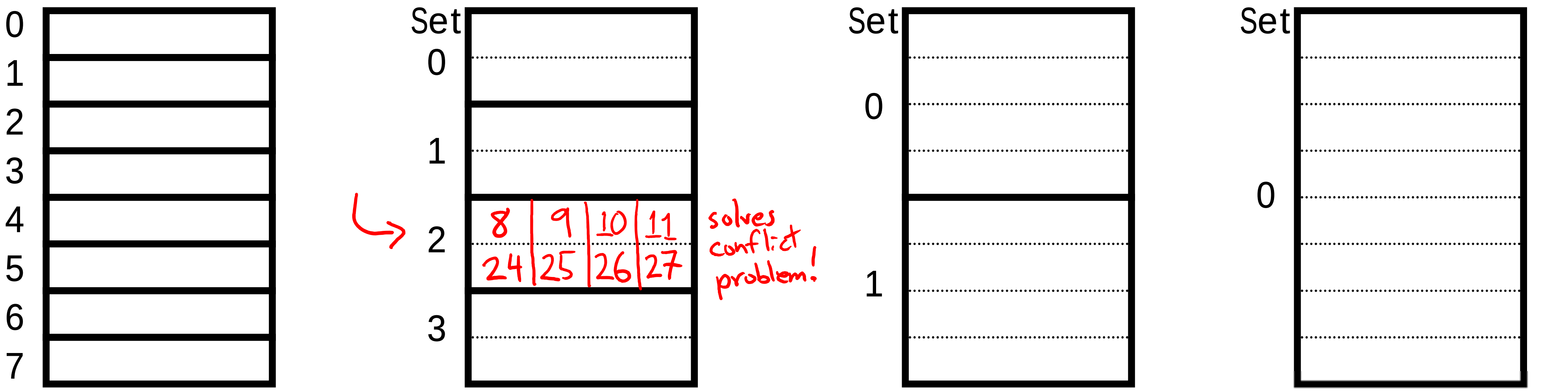
- What if we could store data in any place in the cache?
  - More complicated hardware = more power consumed, slower
- So we *combine* the two ideas:
  - Each address maps to exactly one **set**
  - Each set can store block in more than one **way**

1-way:
8 sets,
1 block each

2-way:
4 sets,
2 blocks each

4-way:
2 sets,
4 blocks each

8-way:
1 set,
8 blocks



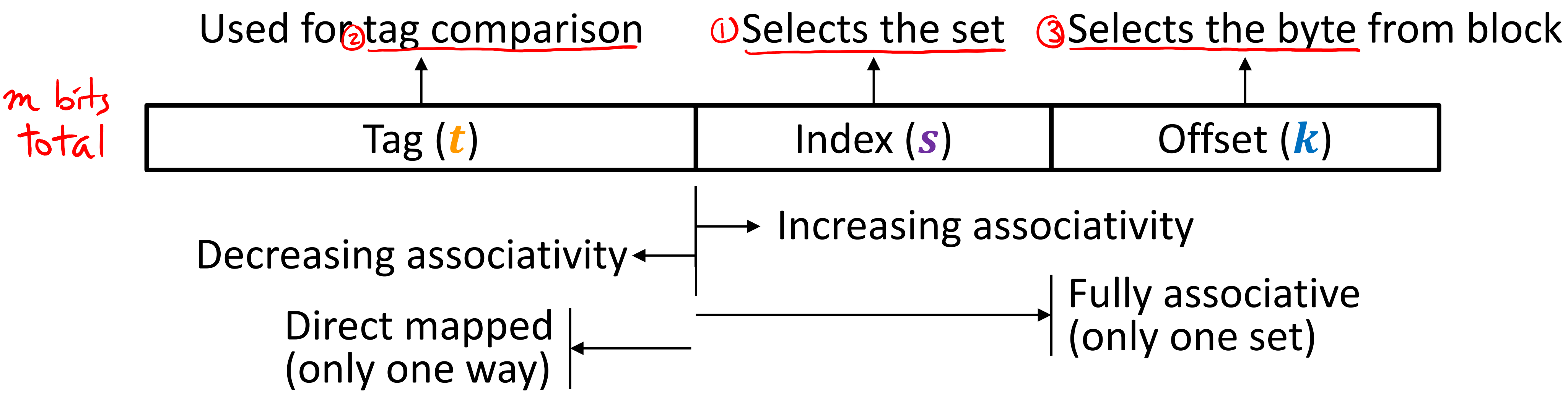
direct mapped

fully associative

Cache Organization (3)

Note: The textbook uses "b" for offset bits

- Associativity (E): # of ways for each set
  - Such a cache is called an "*E-way set associative cache*"
  - We now index into cache *sets*, of which there are $\underline{C/K/E} = S$ sets
  - Use lowest $\boxed{\log_2(C/K/E) = \boldsymbol{s} \text{ bits}}$ of block address
    - Direct-mapped: $E = 1$, so $\boldsymbol{s} = \log_2(C/K)$ as we saw previously
    - Fully associative: $E = C/K$, so $\boldsymbol{s} = 0$ bits



Used for ② tag comparison

① Selects the set

③ Selects the byte from block

m bits total

Tag (t)	Index (s)	Offset (k)

Decreasing associativity

Increasing associativity

Direct mapped (only one way)

Fully associative (only one set)

Example Placement

block size:	16 B	K
capacity:	8 blocks	C/K
address:	16 bits	m

- Where would data from address `0x1833` be placed?
  - Binary: `0b 0001 1000 0011 0011`

$I_{E=1}$ $I_{E=4}$ $I_{E=2}$ offset

$t = m – s – k$ $s = \log_2(C/K/E)$ $k = \log_2(K) = 4$

16

m-bit address:

Tag (t)	Index (s)	Offset (k)

s = ? $\log_2(8/1) = 3$ bits

Direct-mapped (E=1)

	Set	Tag	Data
(000)	0		
(001)	1		
(010)	2		
(011)	3		✓
(100)	4		
(101)	5		
(110)	6		
(111)	7		

s = ? $\log_2(8/2) = 2$ bits

2-way set associative (E=2)

	Set	Tag	Data
(00)	0		
(01)	1		
(10)	2		
(11)	3		✓
			✓

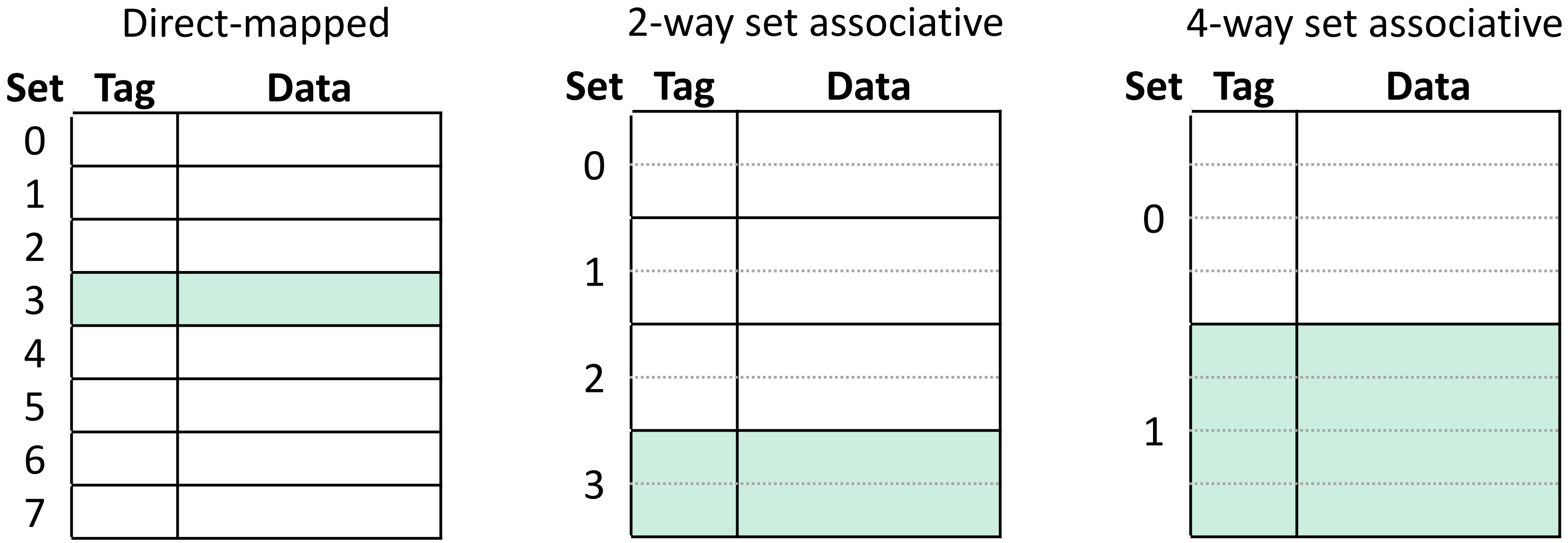
s = ? $\log_2(8/4) = 1$ bit

4-way set associative (E=4)

	Set	Tag	Data
(0)	0		
(1)	1		✓
			✓
			✓
			✓

Block Replacement

- *Any* empty block in the correct set may be used to store block
- If there are no empty blocks, which one should we replace?
  - No choice for direct-mapped caches
  - Caches typically use something close to ***least recently used (LRU)*** (hardware usually implements "*not most recently used*")



Peer Instruction Question

$C = 2^{11}$ B $K = 2^7$ B

- We have a cache of size 2 KiB with block size of 128 B. If our cache has 2 sets, what is its associativity?
  - Vote at http://PollEv.com/justinh

cache holds $C/K = 2^{11-7} = 2^4 = 16$ blocks

A. 2

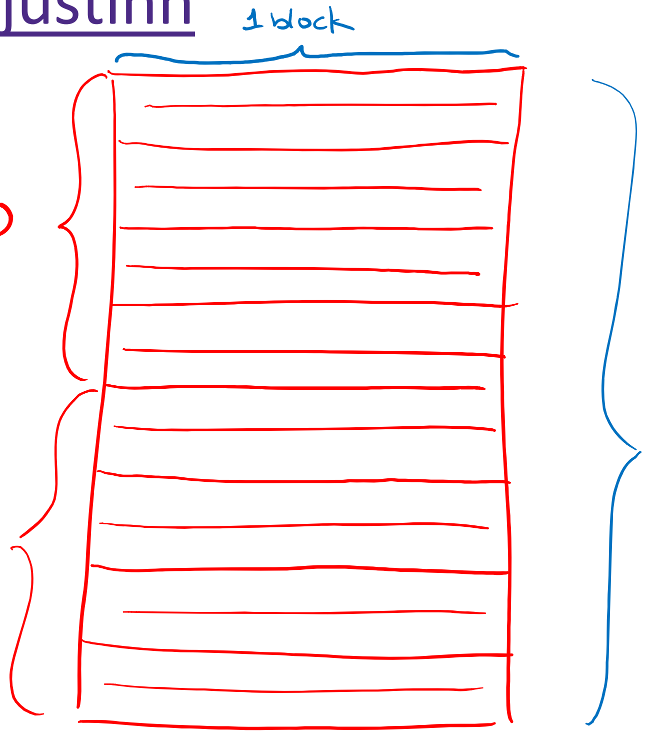
B. 4

[✓ C. 8]

D. 16

E. We're lost…

each set has 8 blocks, so $E=8$



$S = C/K/E$

$E = (C/K)/S$
$= 16/2 = 8$

$m=16$

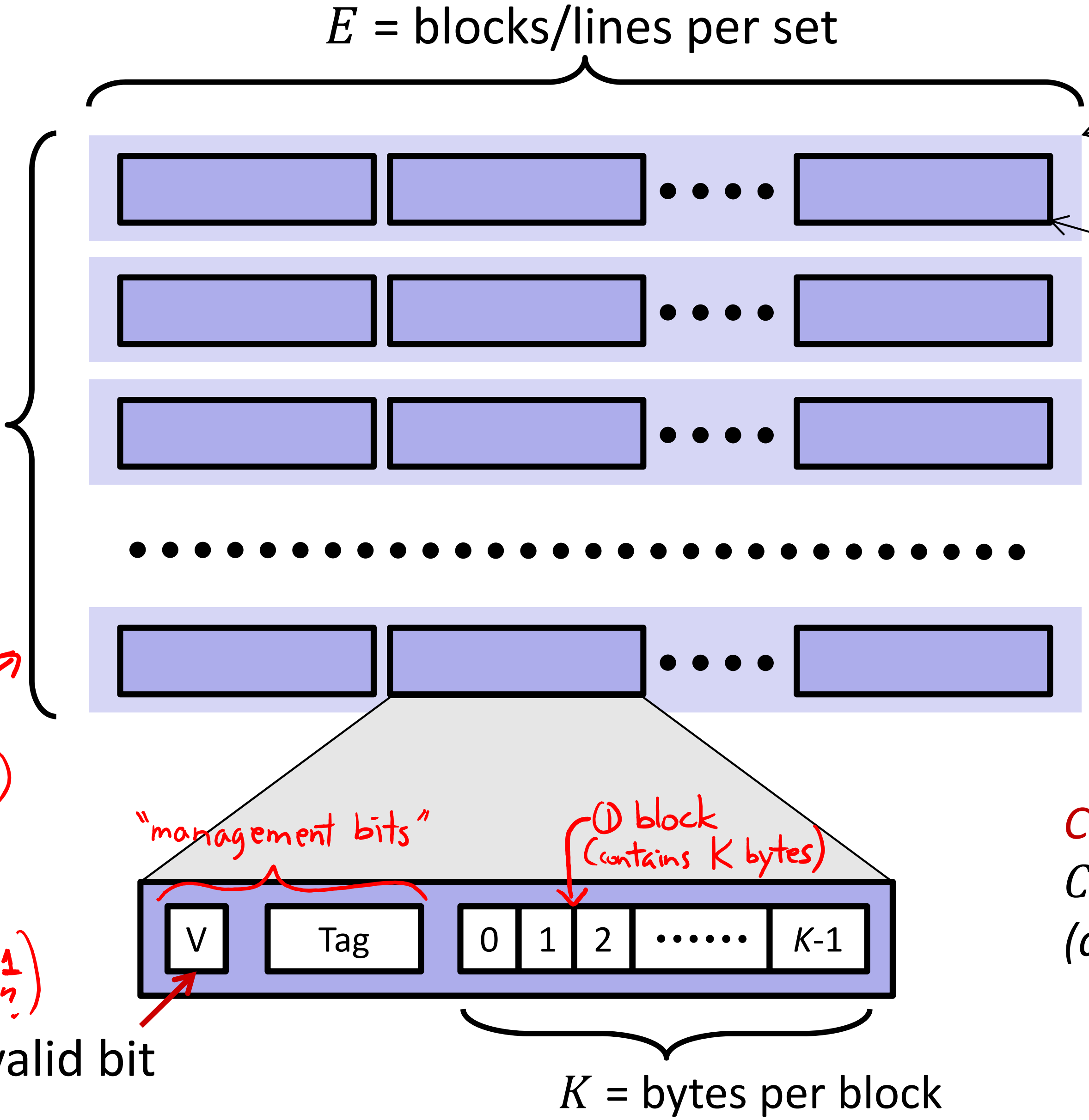
- If addresses are 16 bits wide, how wide is the Tag field? $k = \log_2(K) = 7$ bits, $s = \log_2(S) = 1$ bit, $t = m-s-k = \boxed{8 \text{ bits}}$

General Cache Organization (S, E, K)

associativity

sets

block size



③ set (contains E lines)

② "line" (block plus management bits)

"Layers" of a cache:

① block (data)

② line

③ set

④ cache

④ cache (contains S sets)

(is this data valid/1 or garbage/0?)

Cache size:

$C = K \times E \times S$ data bytes

(doesn't include V or Tag)

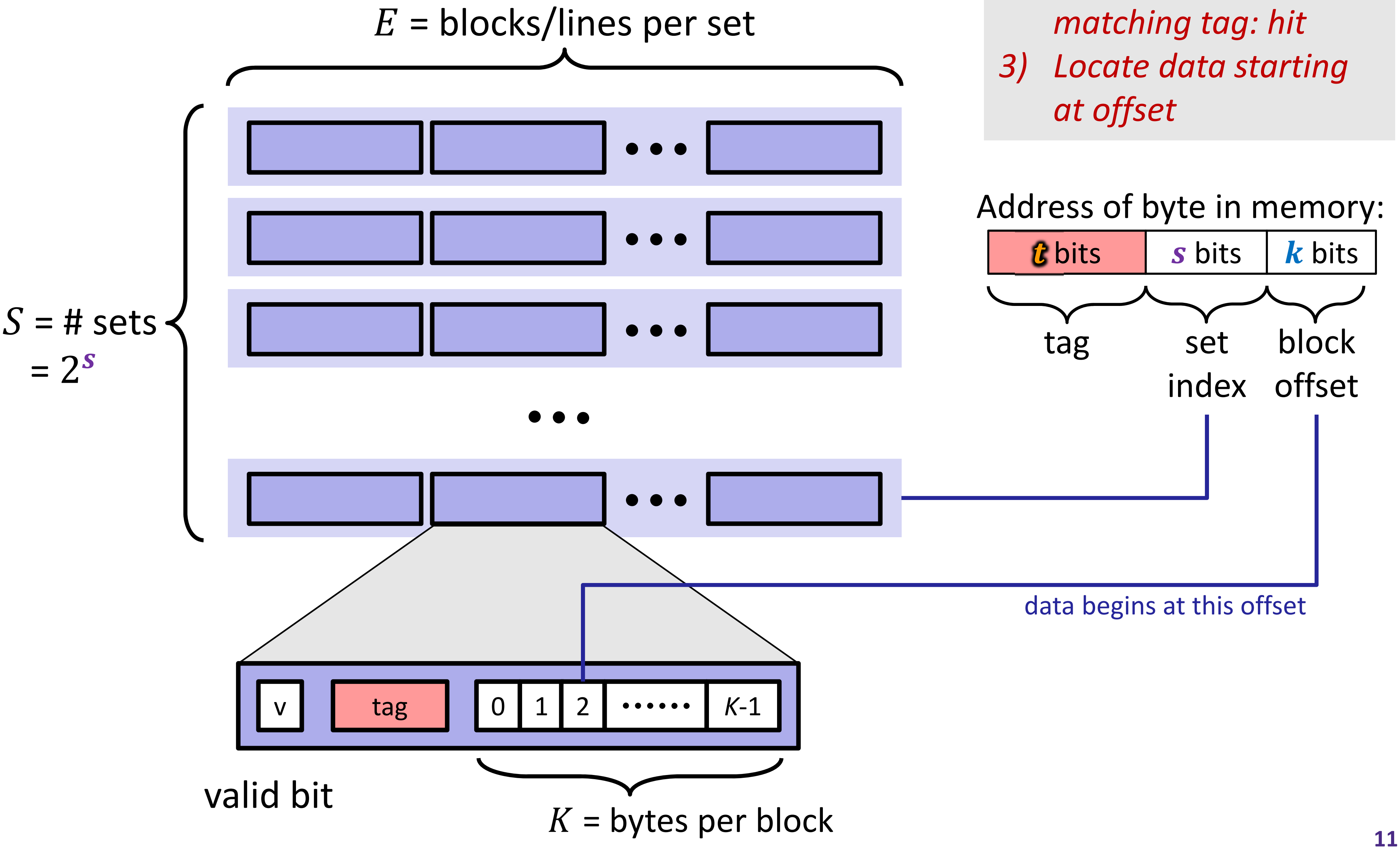
Notation Review

- We just introduced a lot of new variable names!
  - Please be mindful of block size notation when you look at past exam questions or are watching videos

Variable	This Quarter	Formulas
Block size	K (B in book)	$M = 2^{\boldsymbol{m}} \leftrightarrow \boldsymbol{m} = \log_2 M$ $S = 2^{\boldsymbol{s}} \leftrightarrow \boldsymbol{s} = \log_2 S$ $K = 2^{\boldsymbol{k}} \leftrightarrow \boldsymbol{k} = \log_2 K$ $C = K \times E \times S$ $\boldsymbol{s} = \log_2(C/K/E)$ $\boldsymbol{m} = \boldsymbol{t} + \boldsymbol{s} + \boldsymbol{k}$
Cache size	C	
Associativity	E	
Number of Sets	S	
Address space	M	
Address width	$\boldsymbol{m}$	
Tag field width	$\boldsymbol{t}$	
Index field width	$\boldsymbol{s}$	
Offset field width	$\boldsymbol{k}$ ($\boldsymbol{b}$ in book)	

Cache Read

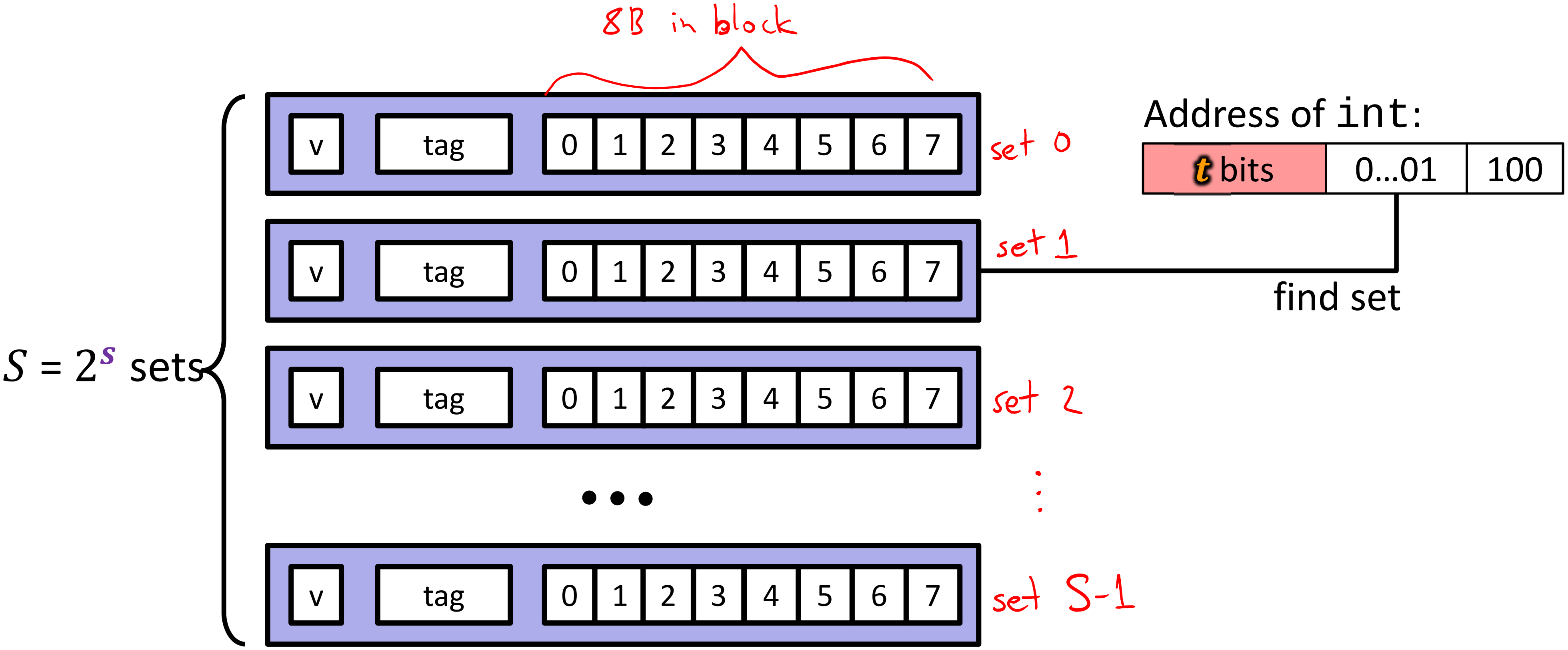
1) *Locate set*
2) *Check if any line in set is valid and has matching tag: hit*
3) *Locate data starting at offset*



Example: Direct-Mapped Cache (E = 1)

Direct-mapped: One line per set

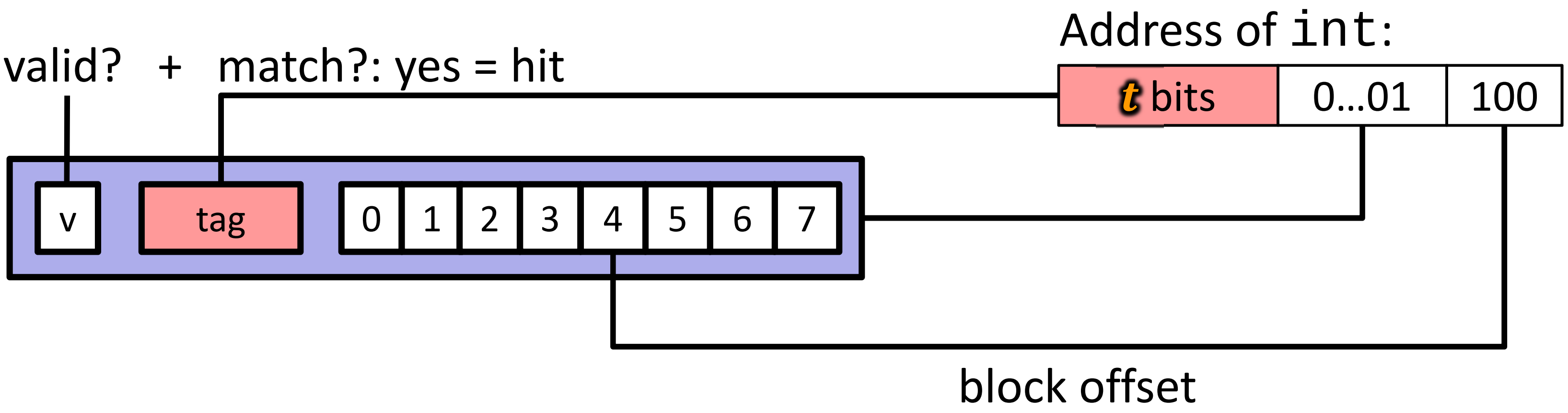
Block Size K = 8 B



Example: Direct-Mapped Cache (E = 1)

Direct-mapped: One line per set

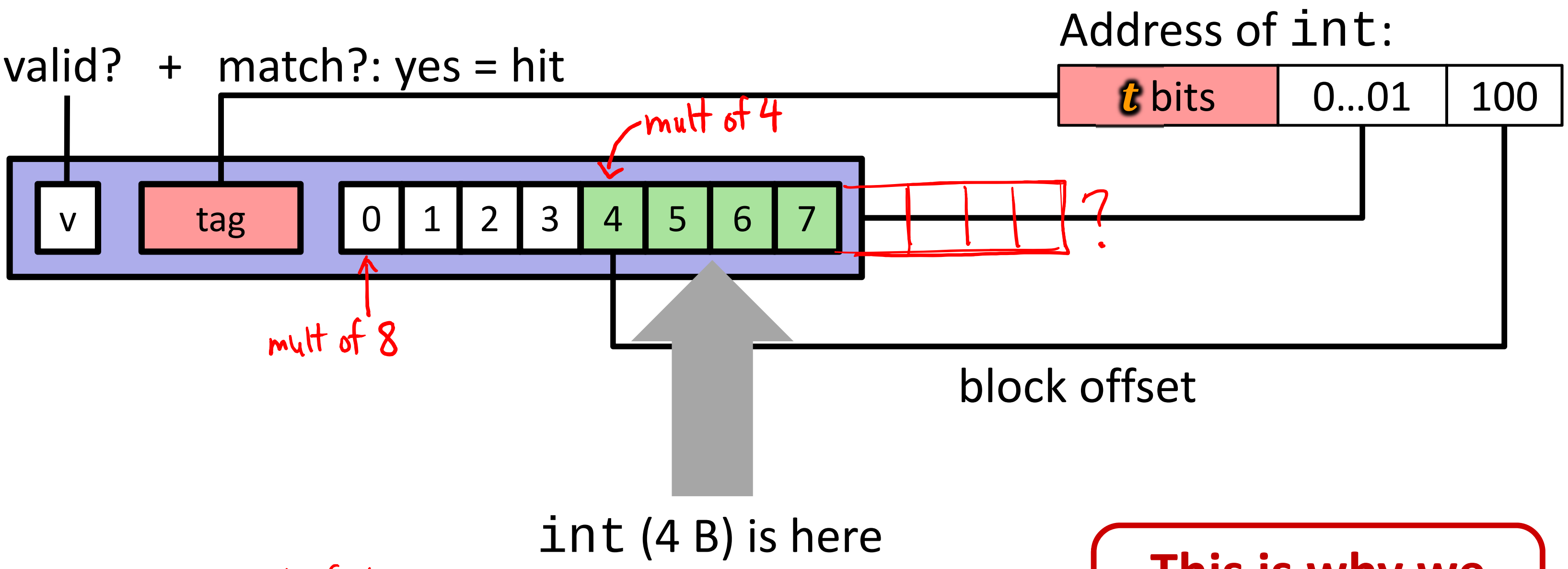
Block Size K = 8 B



Example: Direct-Mapped Cache (E = 1)

Direct-mapped: One line per set

Block Size K = 8 B



what if this was long (8B)?

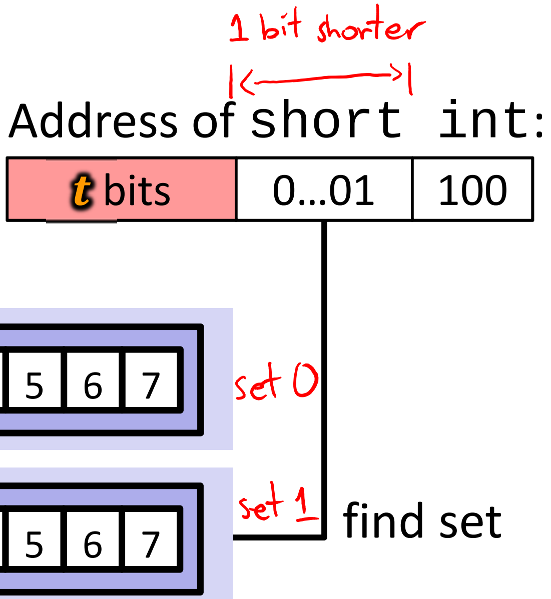
This is why we want alignment!

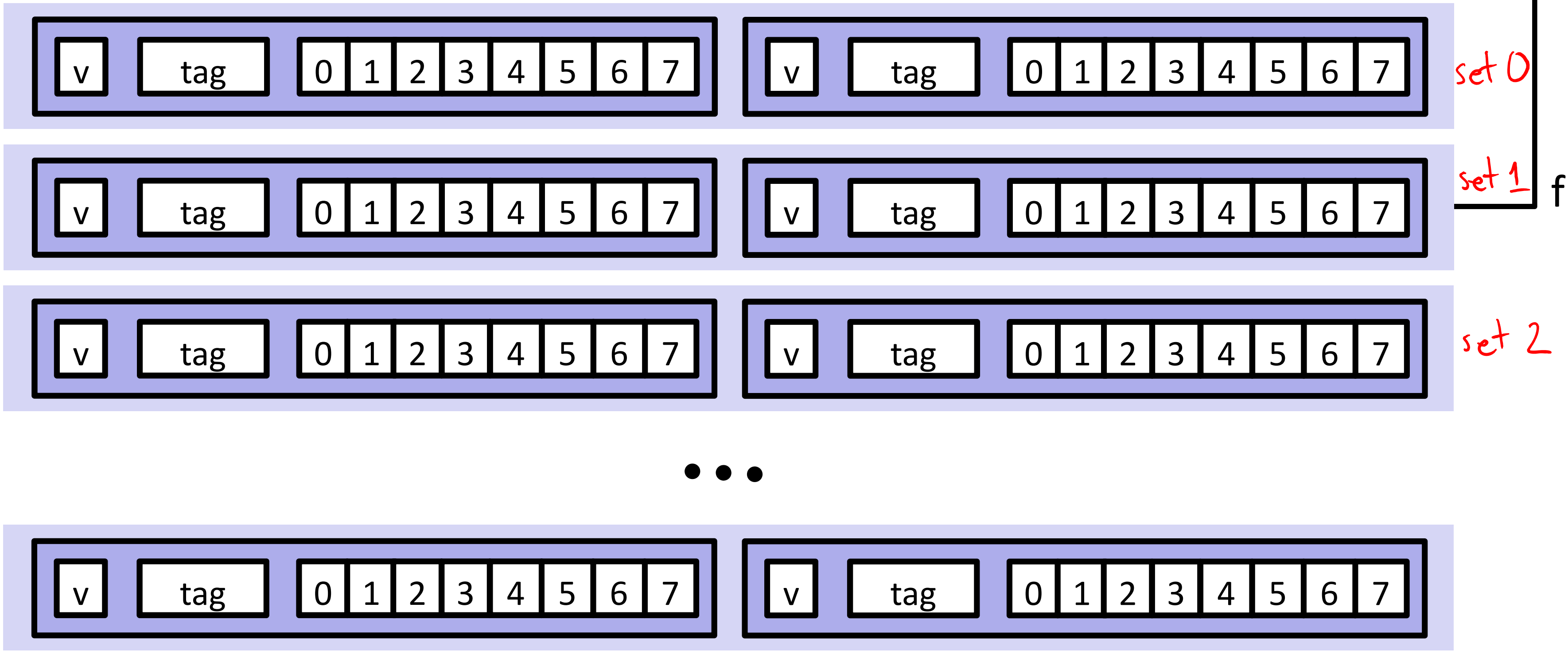
no unnecessary extra cache accesses across block boundaries

No match? Then old line gets evicted and replaced

Example: Set-Associative Cache (E = 2)

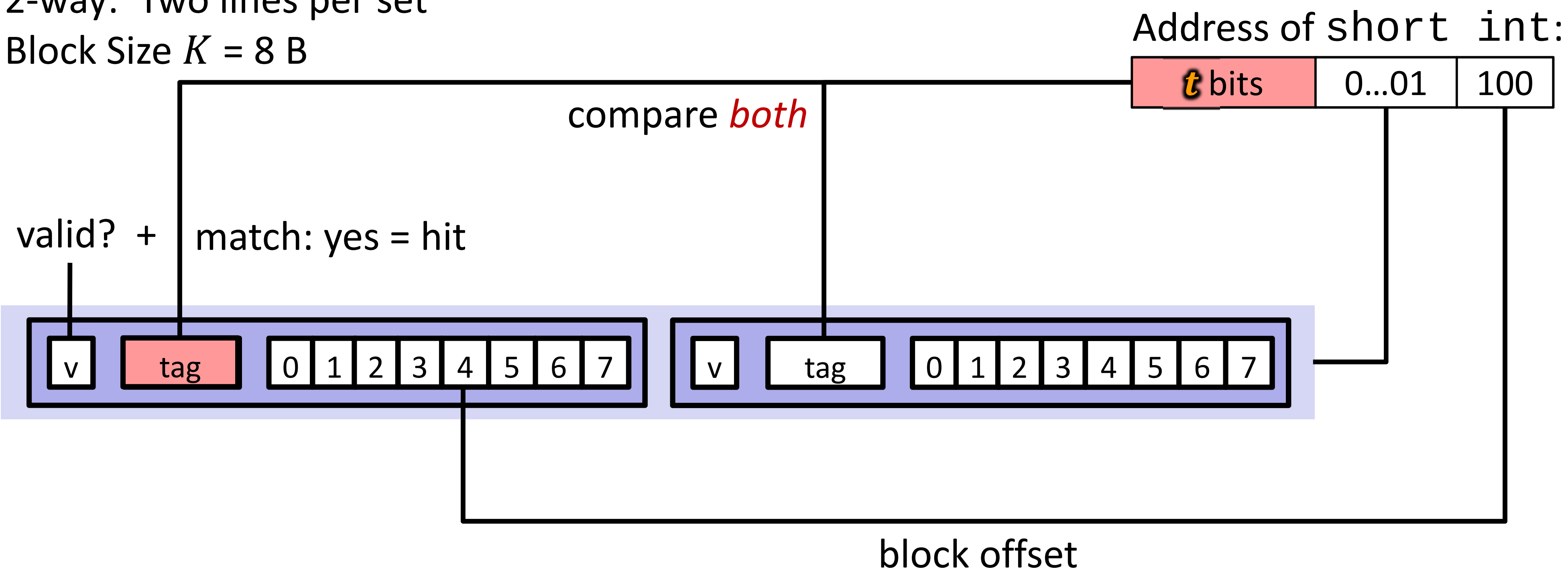
2-way: Two lines per set
Block Size K = 8 B





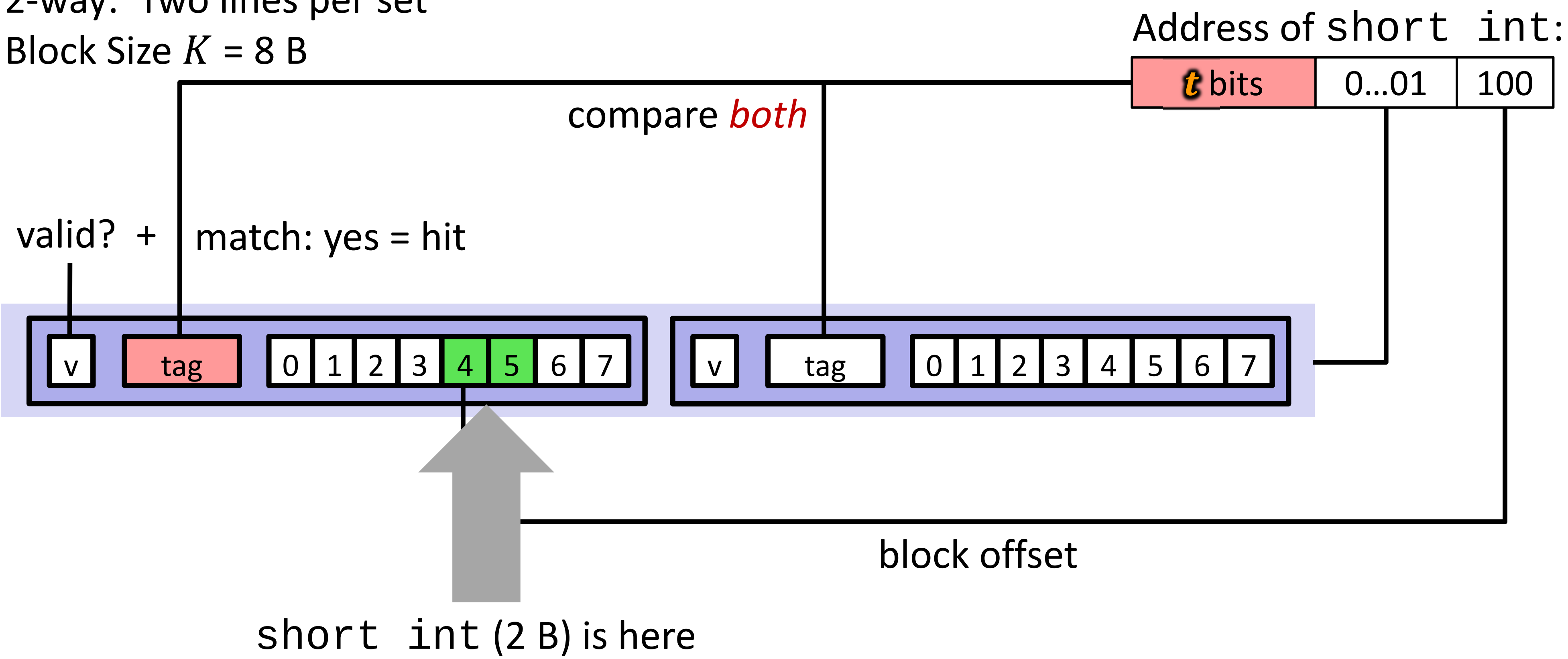
Example: Set-Associative Cache (E = 2)

2-way: Two lines per set
Block Size K = 8 B



Example: Set-Associative Cache (E = 2)

2-way: Two lines per set
Block Size K = 8 B



No match?

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), …

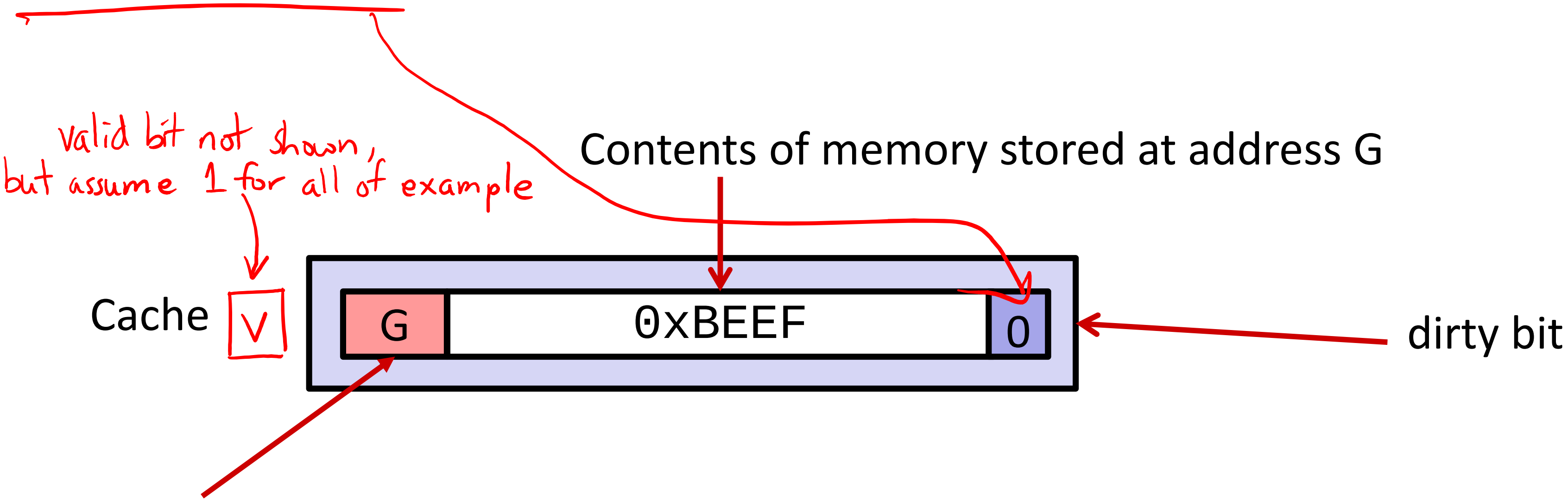
Types of Cache Misses: 3 C's!

- Compulsory (cold) miss
  - Occurs on first access to a block
- Conflict miss
  - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
    - *e.g.* referencing blocks 0, 8, 0, 8, ... could miss every time
  - Direct-mapped caches have more conflict misses than E-way set-associative (where E > 1)
- Capacity miss
  - Occurs when the set of active cache blocks (the *working set*) is larger than the cache (just won't fit, even if cache was *fully-associative*)
  - **Note:** *Fully-associative* only has Compulsory and Capacity misses

What about writes?

- Multiple copies of data exist:
  - L1, L2, possibly L3, main memory
- What to do on a write-hit? (block/data already in $)
  - Write-through: write immediately to next level
  - Write-back: defer write to next level until line is evicted (replaced)
    - Must track which cache lines have been modified ("*dirty bit*") ← extra management bit only for write-back $
- What to do on a write-miss? (block/data not currently in $)
  - Write-allocate: ("fetch on write") load into cache, update line in cache
    - Good if more writes or reads to the location follow
  - No-write-allocate: ("write around") just write immediately to memory
- Typical caches:
  - Write-back + Write-allocate, usually ☆
  - Write-through + No-write-allocate, occasionally

Write-back, write-allocate example



valid bit not shown, but assume 1 for all of example

Contents of memory stored at address G

Cache V

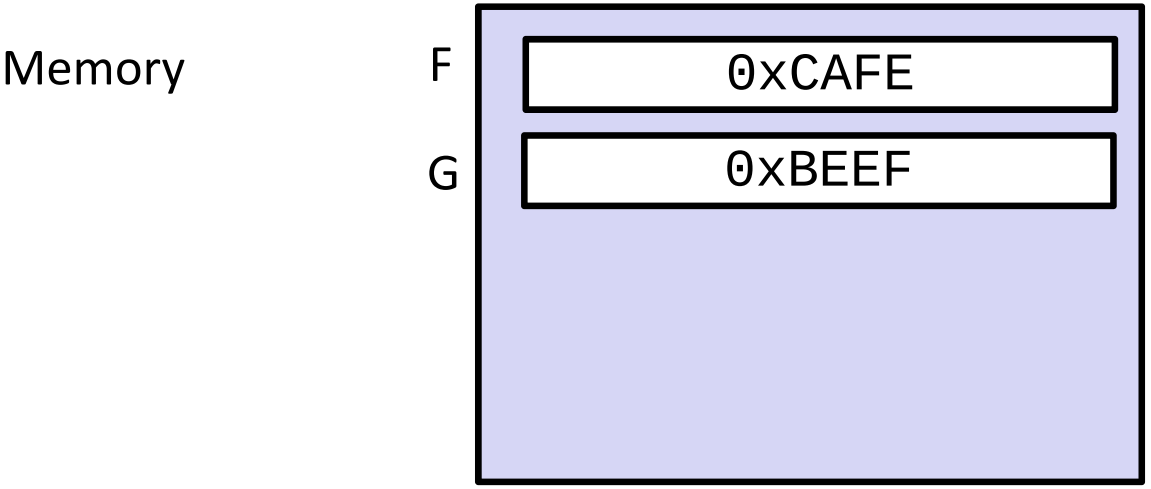
G

0xBEEF

0

dirty bit

tag (there is only one set in this tiny cache, so the tag is the entire block address!)



Memory

F 0xCAFE

G 0xBEEF

In this example we are sort of ignoring block offsets. Here a block holds 2 bytes (16 bits, 4 hex digits).

Normally a block would be much bigger and thus there would be multiple items per block. While only one item in that block would be written at a time, the entire line would be brought into cache.

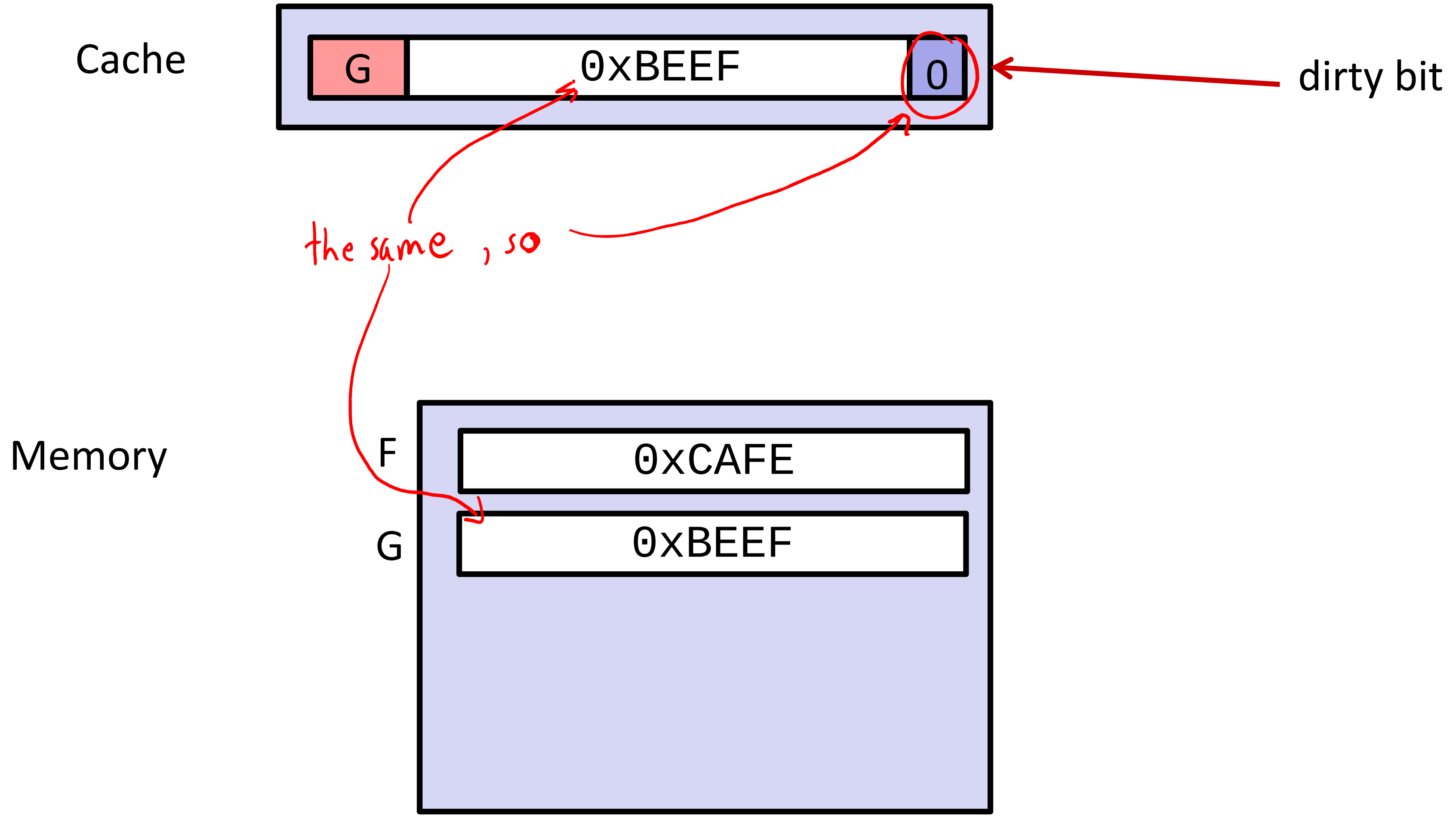
Write-back, write-allocate example

write miss

```
mov 0xFACE, F
```

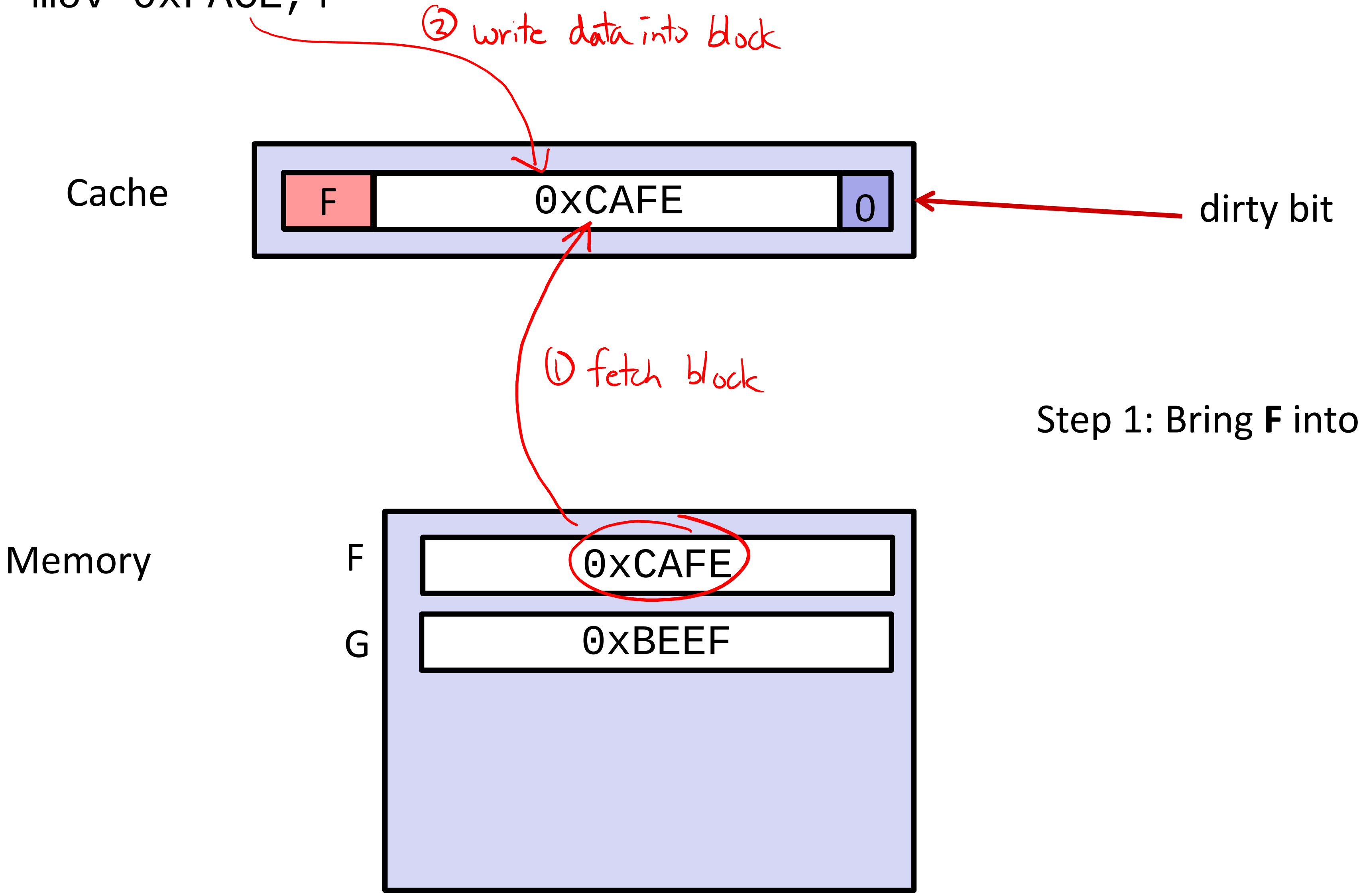
① check cache for F → miss

② pull block into $, then write



Write-back, write-allocate example

```
mov 0xFACE, F
```



Cache

F

0xCAFE

0

dirty bit

Memory

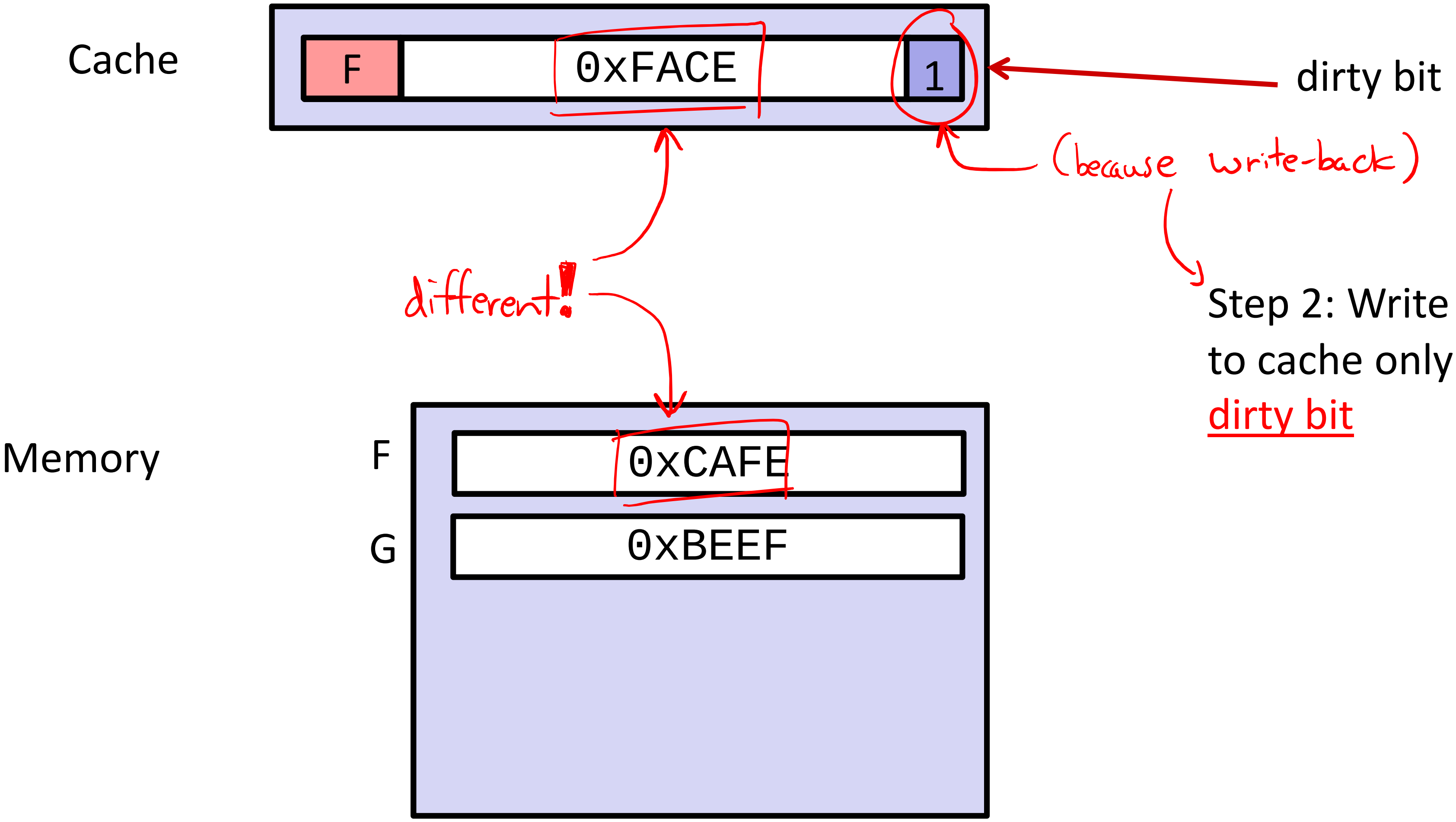
F 0xCAFE

G 0xBEEF

Step 1: Bring **F** into cache

Write-back, write-allocate example

```
mov 0xFACE, F
```

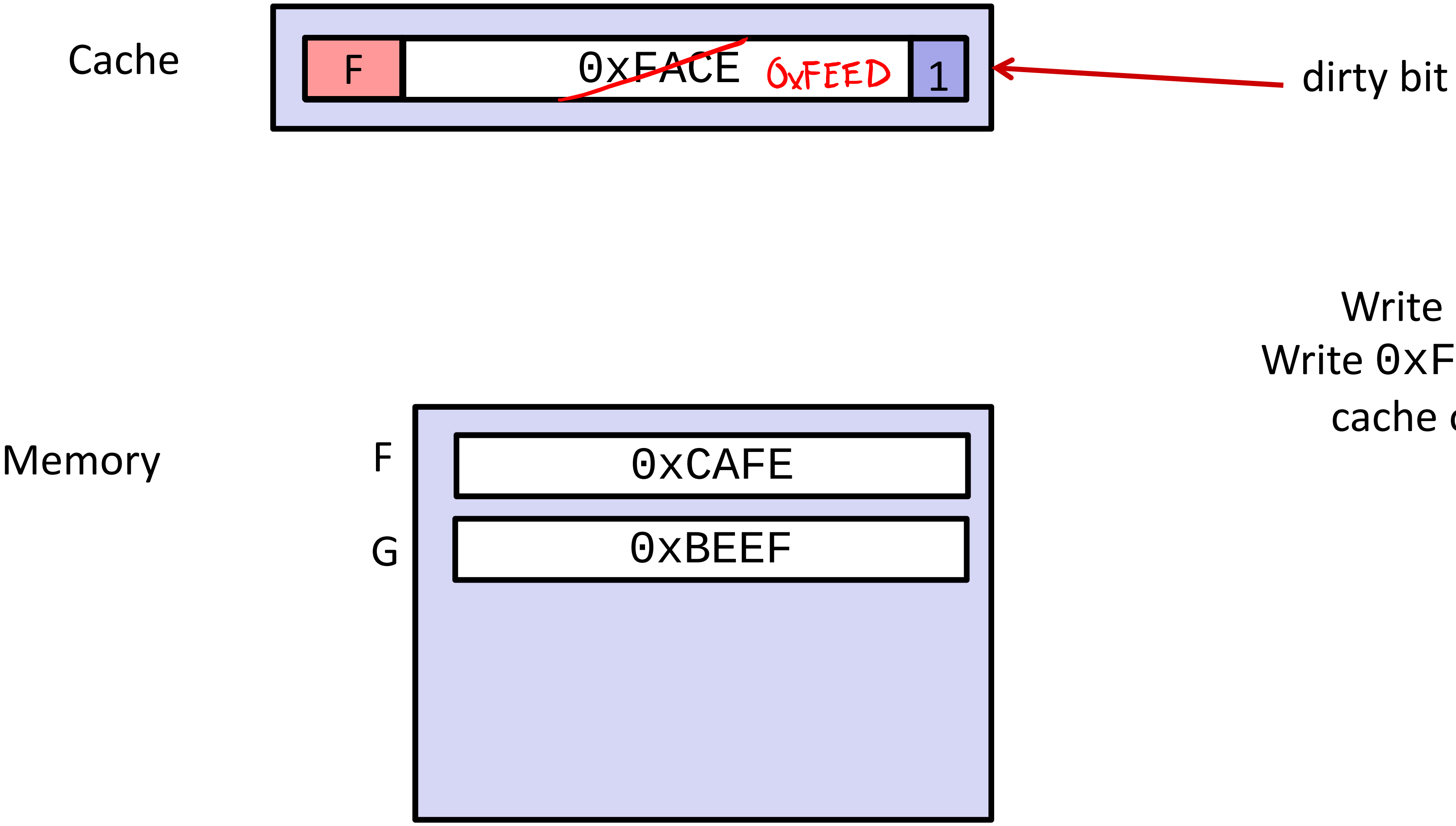


Step 2: Write 0xFACE to cache only and set dirty bit

Write-back, write-allocate example

```
mov 0xFACE, F      mov 0xFEED, F
```

write hit

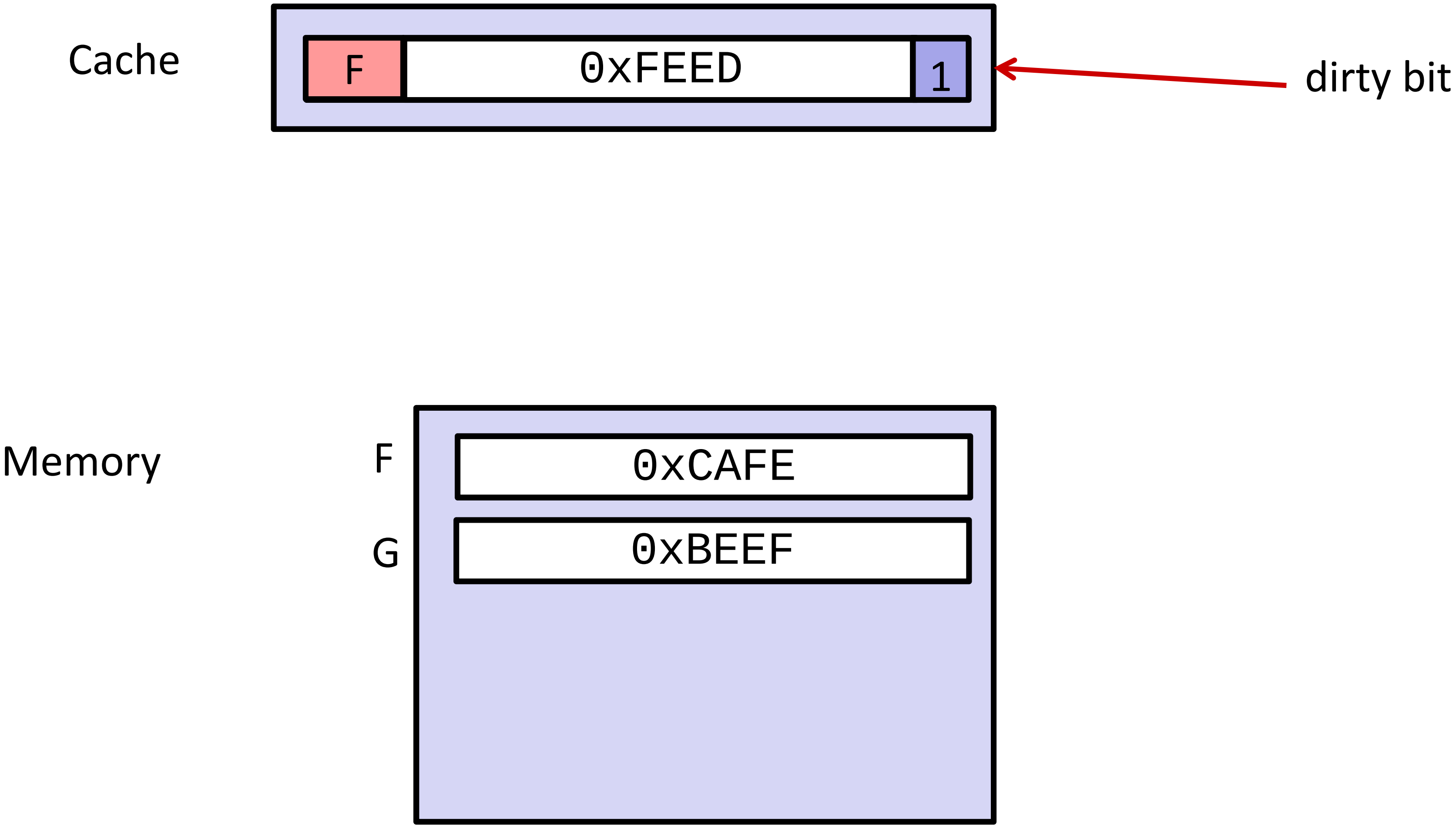


Write hit!
Write 0xFEED to cache only

Write-back, write-allocate example

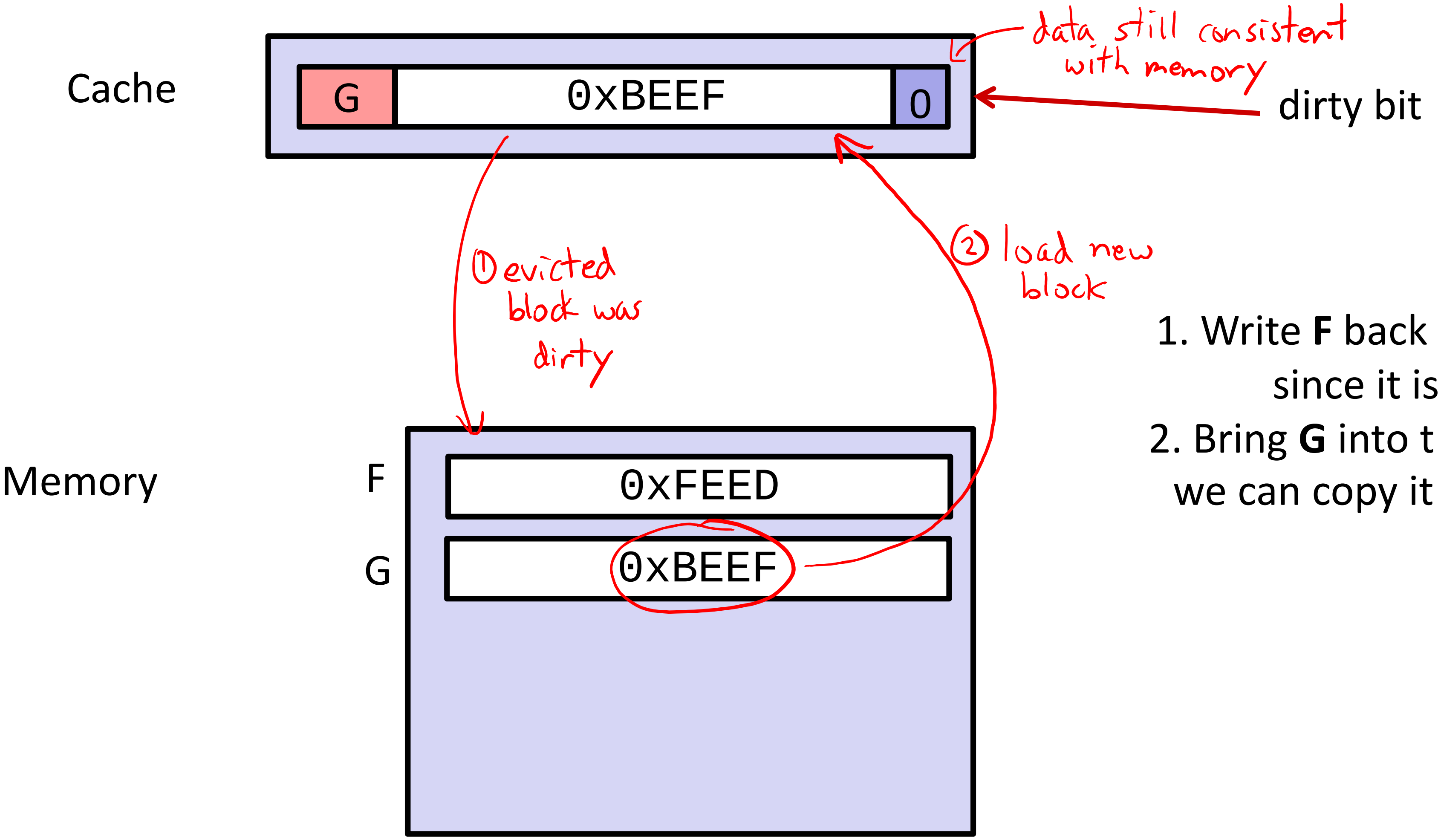
`mov 0xFACE, F` `mov 0xFEED, F` `mov G, %rax`

read miss



Write-back, write-allocate example

`mov 0xFACE, F`   `mov 0xFEED, F`   `mov G, %rax`



1. Write **F** back to memory since it is dirty
2. Bring **G** into the cache so we can copy it into `%rax`

Peer Instruction Question

- Which of the following cache statements is FALSE?
  - Vote at http://PollEv.com/justinh

[✓ A.] We can reduce compulsory misses by decreasing our block size

smaller block size pulls fewer bytes into $ on a miss

B. We can reduce conflict misses by increasing associativity

more options to place blocks before evictions occur

C. A write-back cache will save time for code with good temporal locality on writes

frequently-used blocks rarely get evicted, so fewer write-backs

D. A write-through cache will always match data with the memory hierarchy level below it

yes, its main goal is data consistency

E. We're lost…

Example Cache Parameters Problem

2^{20} B ⇒ $m = 20$ bits

(MP)

- 1 MiB address space, 125 cycles to go to memory. Fill in the following table:

	Cache Size (C)	4 KiB = 2^{12} B
	Block Size (K)	16 B = 2^{4} B
	Associativity (E)	4-way = 2^{2}
	Hit Time (HT)	3 cycles
	Miss Rate (MR)	20%
	Write Policy	Write-through
	Replacement Policy	LRU
$m - s - k$	Tag Bits	10
$\log_2(C/K/E)$	Index Bits	6
$\log_2(K)$	Offset Bits	4
HT+MR*MP	AMAT	AMAT = 3 + 0.2 * 125 = 28

Example Code Analysis Problem

overall MR $= \frac{3}{4}\left(\frac{1}{4}\right) + \frac{1}{4}(0) = 3/16$

- Assuming the cache starts cold (all blocks invalid), calculate the **miss rate** for the following loop:
  - m = 20 bits, C = 4 KiB, K = 16 B, E = 4

holds 2^{12} B of data (half of int_ar[])

```
#define AR_SIZE 2048
```

$= 2^{11}$ ints $= 2^{13}$ B of data

$t=10,\ s=6,\ k=4$

```
int int_ar[AR_SIZE], sum=0;     // &int_ar=0x80000
```

int_ar[0] accesses first 4 B (offset 0) of a cache block in set 0.

```
for (int i=0; i<AR_SIZE; i++)
   sum += int_ar[i];
```

① read i

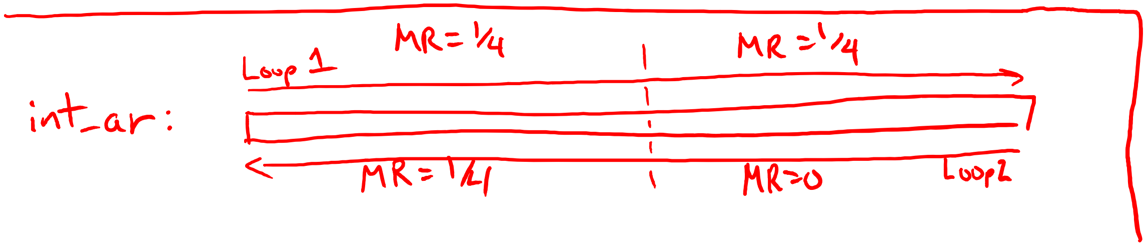
same thing, but in reverse order

```
for (int j=AR_SIZE-1; j>=0; j--)
   sum += int_ar[i];
```

② read i

cache block: | int 0 | int 1 | int 2 | int 3 | 16B = 4 ints

Miss (compulsory) Hit Hit Hit = 1/4 miss rate



Loop1: never re-visit blocks.
first half of loop fills entire \$ with data from lower half of int_ar[].
second half of loop replaces entire \$ data with upper half of int_ar[].

Loop2: first half of loop uses upper half of int_ar[], which is already in the \$ (miss rate of 0).
second half of loop replaces entire \$ data with lower half of int_ar[].