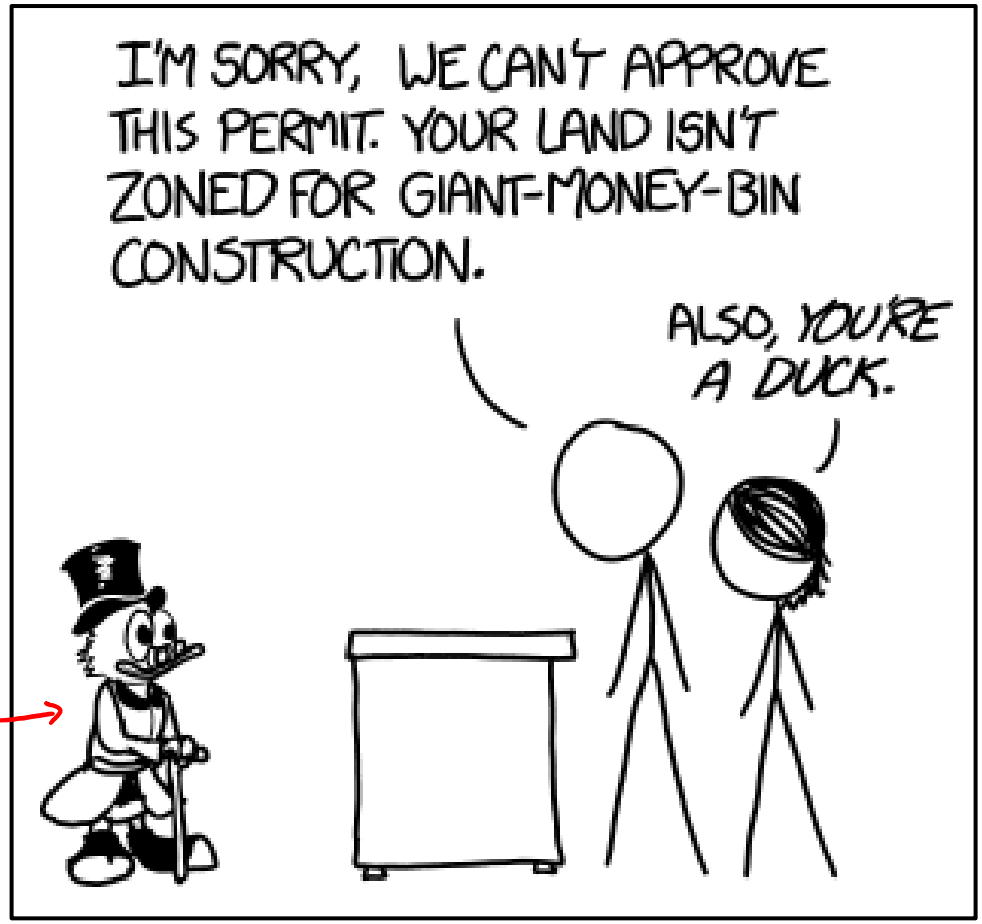
Memory & Caches II

CSE 351 Autumn 2021

Instructor:
Justin Hsia

Teaching Assistants:
Allie Pfleger
Anirudh Kumar
Assaf Vayner
Atharva Deodhar
Celeste Zeng
Dominick Ta
Francesca Wang
Hamsa Shankar
Isabella Nguyen
Joy Dang
Julia Wang
Maggie Jiang
Monty Nitschke
Morel Fotsing
Sanjana Chintalapati



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

Relevant Course Information

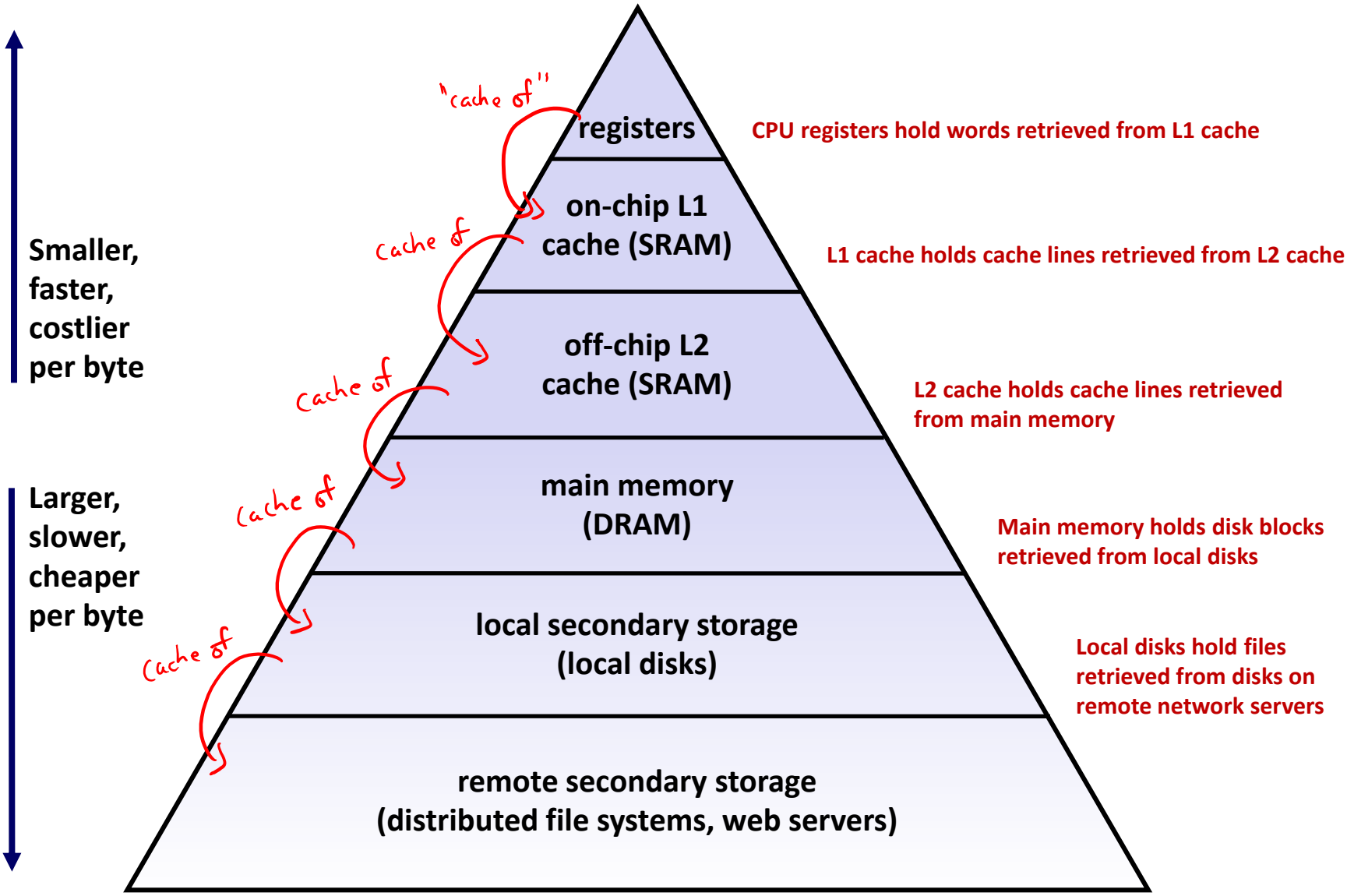
- Mid-quarter Survey due Wednesday (11/10)
- hw16 due Friday (11/12)
- hw19 due *next* Wednesday (11/17)
  - Don't wait too long, this is a BIG hw (includes this lecture)
- Lab 3 due Friday (11/12)
- Midterm grades will be released when we can
  - Regrade requests will be available afterward

Memory Hierarchies (Review)

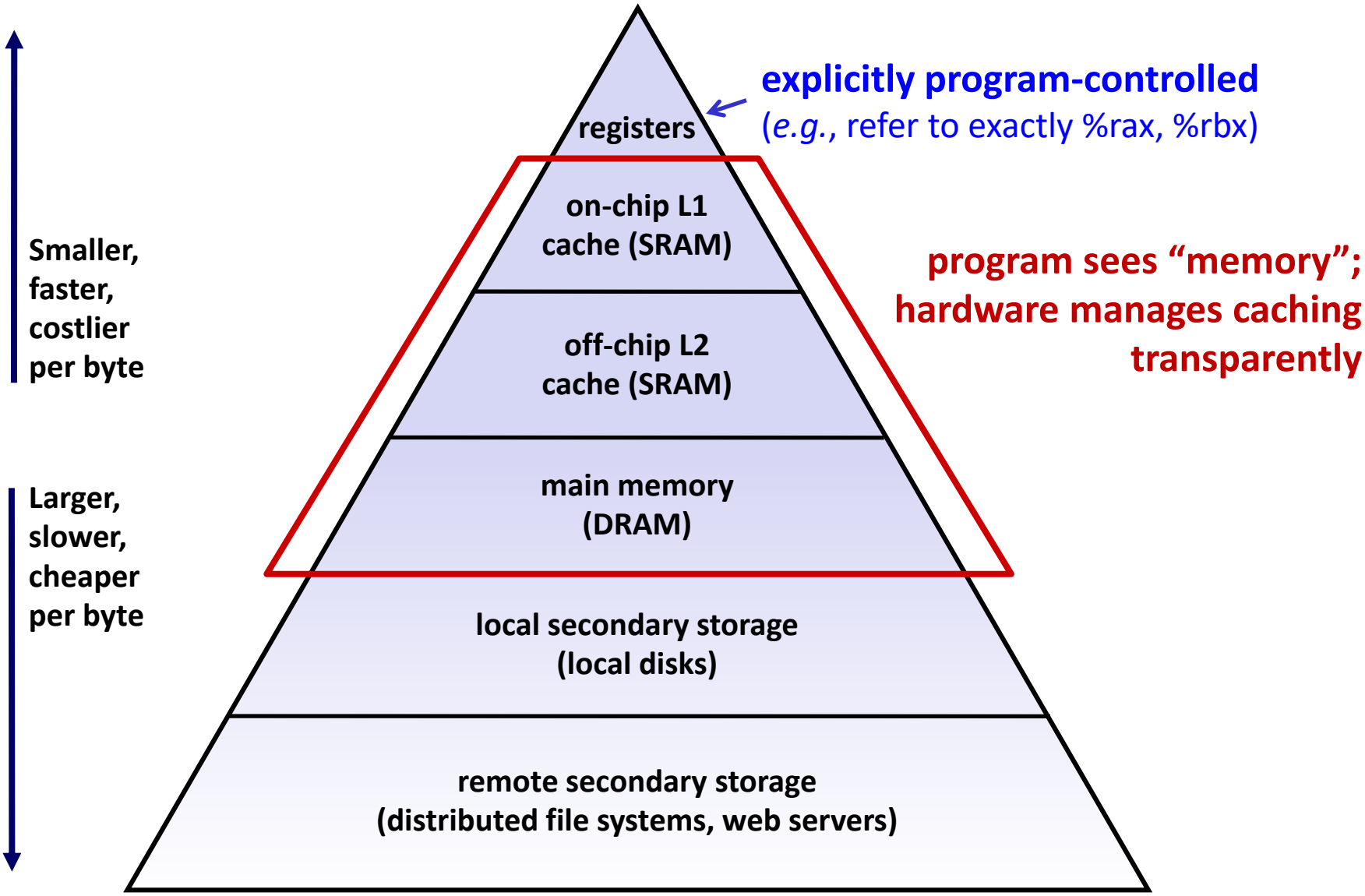
- Some fundamental and enduring properties of hardware and software systems:
  - Faster storage technologies almost always cost more per byte and have lower capacity
  - The gaps between memory technology speeds are widening
    - True for: registers ↔ cache, cache ↔ DRAM, DRAM ↔ disk, etc.
  - Well-written 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
    - For each level k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1

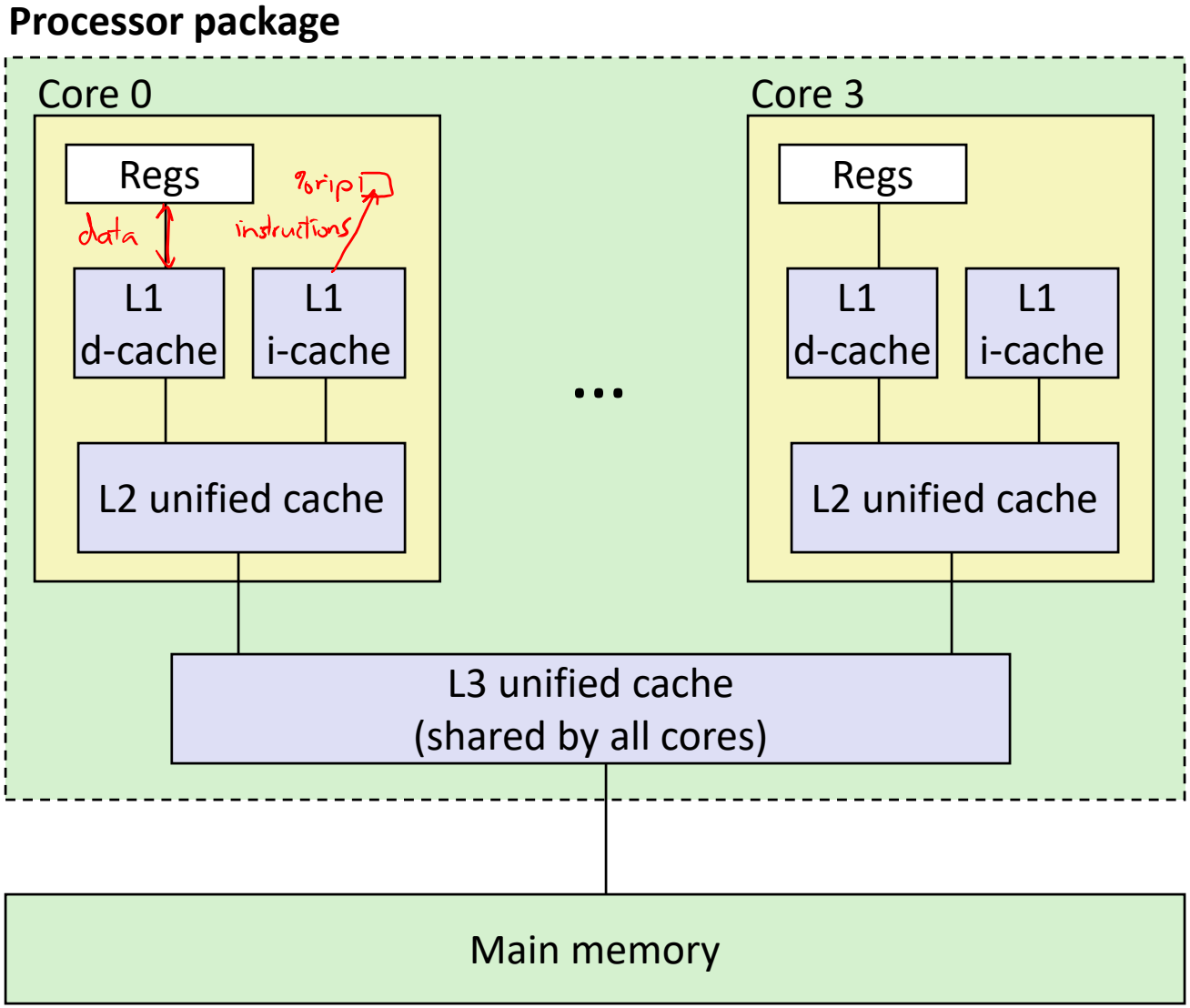
An Example Memory Hierarchy



An Example Memory Hierarchy



Intel Core i7 Cache Hierarchy



Block size:
64 bytes for all caches

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

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

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

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

Reading Review

- Terminology:
  - Memory hierarchy
  - Cache parameters: block size (K), cache size (C)
  - Addresses: block offset field (k bits wide)
  - Cache organization: direct-mapped cache, index field

- Questions from the Reading?

Review Questions

- We have a direct-mapped cache with the following parameters:
  - Block size of 8 bytes $K = 2^3\ B$
  - Cache size of 4 KiB $C = 2^{12}\ B$

  2^2 ↑ ↑ 2^{10}
- How many blocks can the cache hold? $C/K = 2^{12-3} = 2^9 =$ 512 blocks
- How many bits wide is the block offset field? $k = \log_2(K) =$ 3 bits
- Which of the following addresses would fall under block number 3?

$\lfloor 3/8 \rfloor = 0$ $\lfloor 31/8 \rfloor = 3$ $\lfloor 48/8 \rfloor = 6$ $\lfloor 56/8 \rfloor = 7$

A. 0x3 0b 00 000 11 — block num 0

B. 0x1F 0b 01 111 11 — block num 3

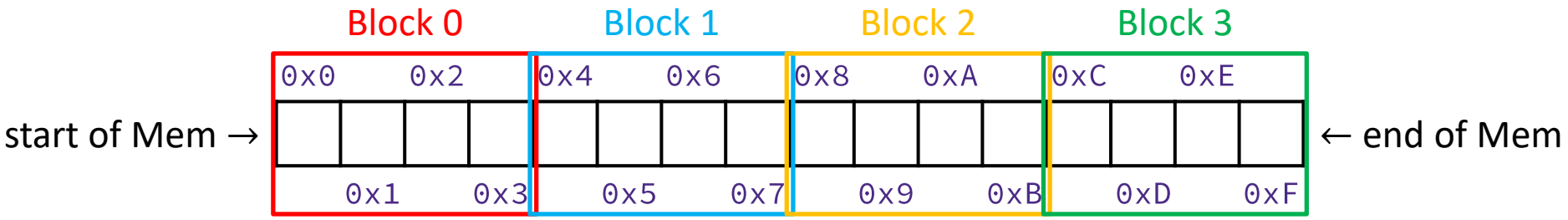
C. 0x30 0b 11 000 00 — block num 6

D. 0x38 0b 11 100 00 — block num 7

Cache Organization (1)

Note: The textbook uses "B" for block size

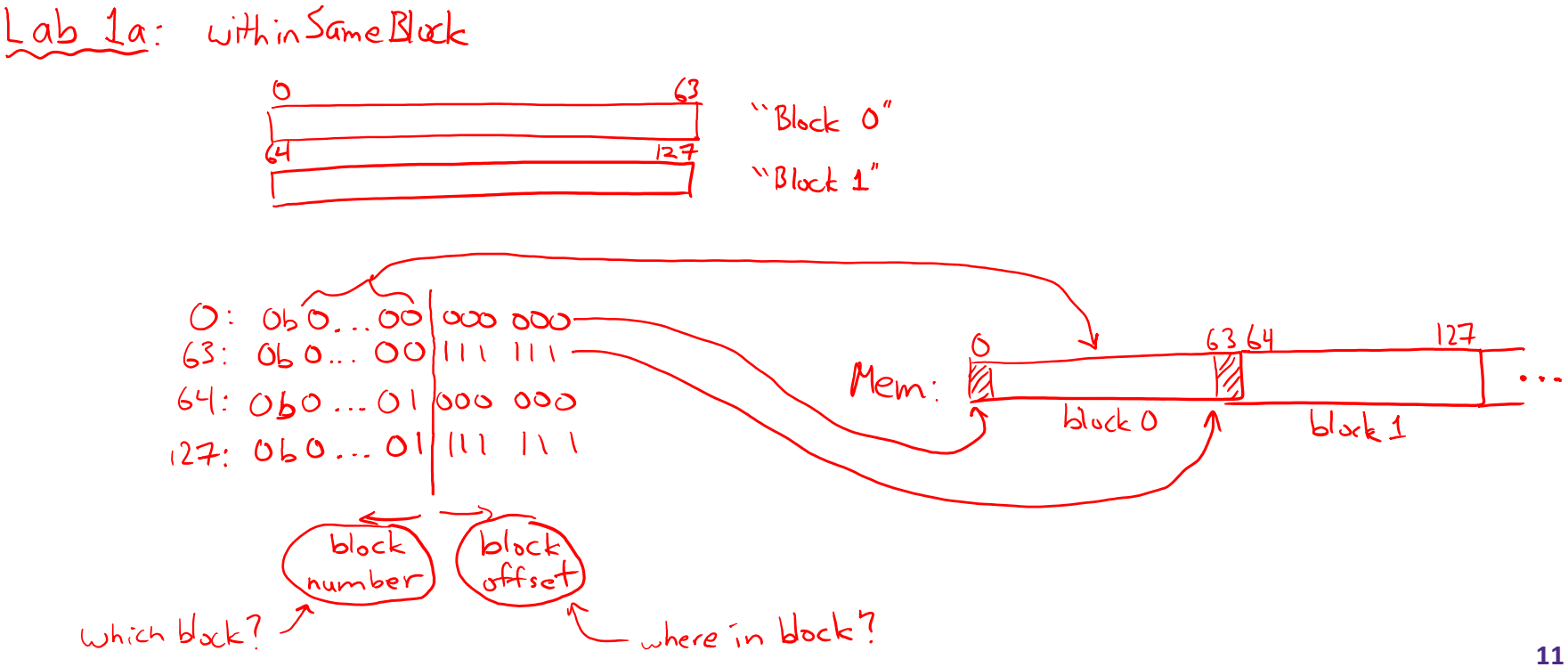
- Block Size (K): unit of transfer between \$ and Mem
  - Given in bytes and always a power of 2 (*e.g.*, 64 B)
  - Blocks consist of adjacent bytes (differ in address by 1)
    - Spatial locality!
  - Small example ($K = 4$ B):



Cache Organization (1)

Note: The textbook uses "B" for block size

- Block Size (K): unit of transfer between $ and Mem
  - Given in bytes and always a power of 2 (*e.g.*, 64 B)
  - Blocks consist of adjacent bytes (differ in address by 1)
    - Spatial locality!



Cache Organization (1)

Note: The textbook uses "b" for offset bits

- Block Size (K): unit of transfer between $ and Mem
  - Given in bytes and always a power of 2 (*e.g.*, 64 B)
  - Blocks consist of adjacent bytes (differ in address by 1)
    - Spatial locality!

$x \% 2^n$ = value of the lowest n bits

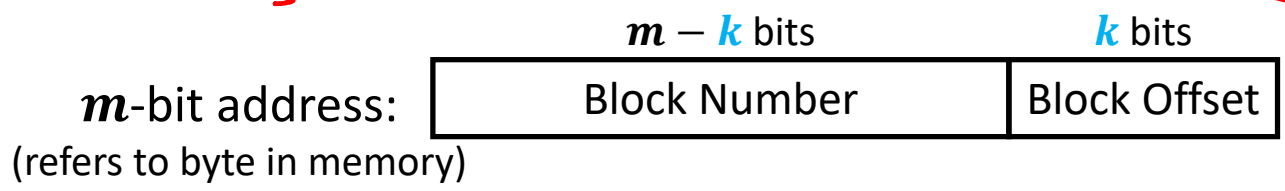
0b … 2^{n+1} 2^n 2^{n-1} … 2^1 2^0

- Offset field
  - Low-order $\log_2(K) = k$ bits of address tell you which byte within a block
    - (address) mod 2^n = n lowest bits of address
  - (address) modulo (# of bytes in a block)

64 6

remainder

How many bits do I need to specify every byte in a block?

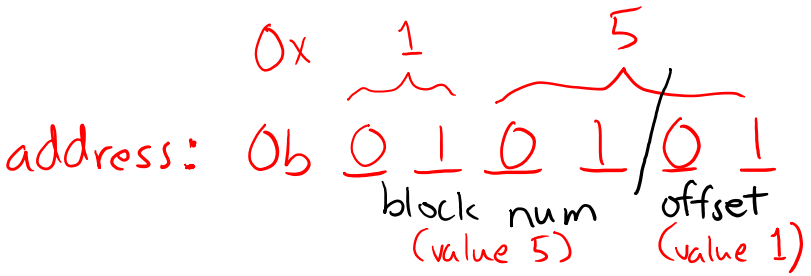


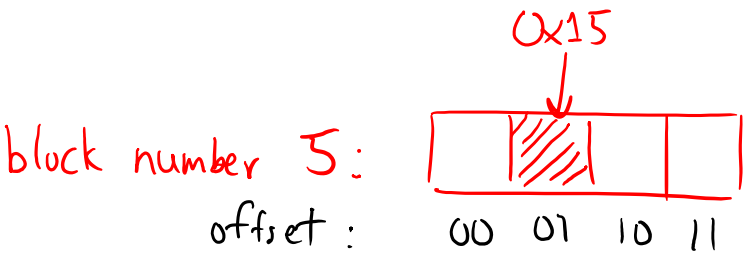
Cache Organization (1)

Note: The textbook uses "b" for offset bits

- Block Size (K): unit of transfer between $ and Mem
  - Given in bytes and always a power of 2 (*e.g.*, 64 B)
  - Blocks consist of adjacent bytes (differ in address by 1)
    - Spatial locality!

- Example:
  - If we have 6-bit addresses (m) and block size K = 4 B, which block and byte does `0x15` refer to?





offset width = $\log_2(K) = \log_2(4) = 2$ bits

Cache Organization (2)

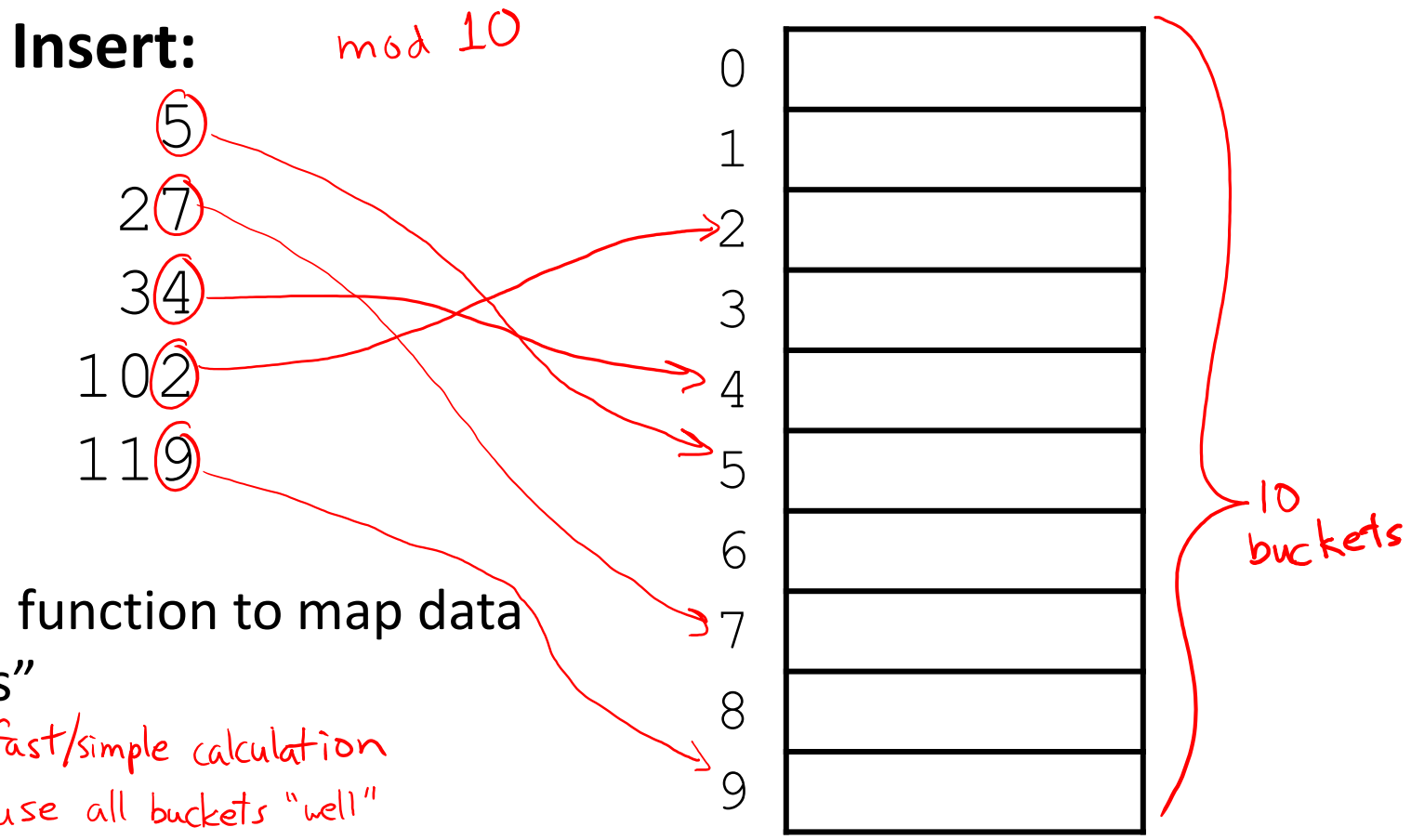
- Cache Size (C): amount of *data* the $ can store
  - Cache can only hold so much data (subset of next level)
  - Given in bytes (C) or number of blocks (C/K)
  - Example: C = 32 KiB = 512 blocks if using 64-B blocks

$2^5 \times 2^{10} = 2^{15}$ B $\times \frac{1 \text{ block}}{2^6 \text{ B}} = 2^9$ blocks

- Where should data go in the cache?
  - We need a mapping from memory addresses to specific locations in the cache to make checking the cache for an address **fast**

- What is a data structure that provides fast lookup?
  - Hash table!

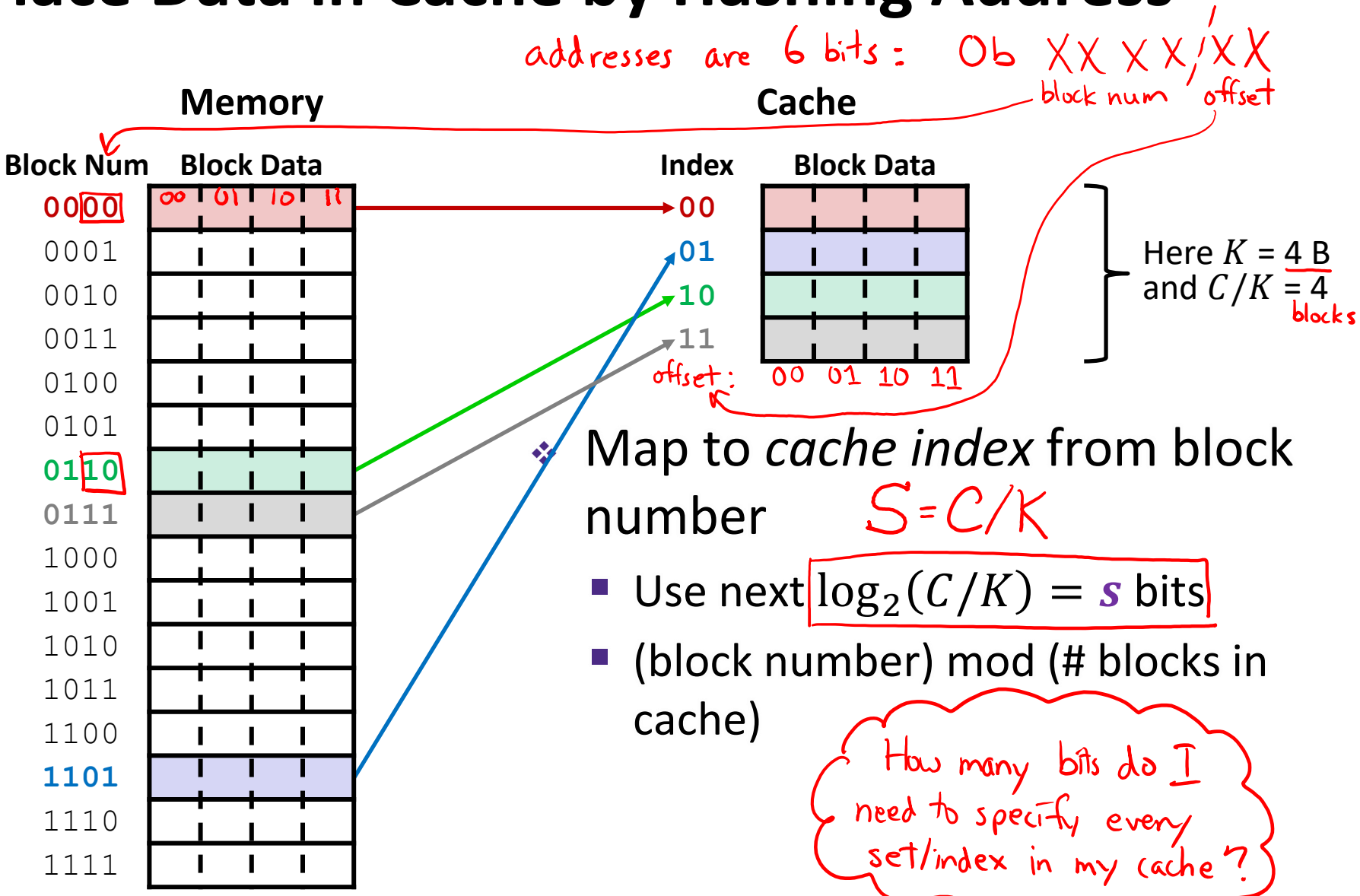
Hash Tables for Fast Lookup



Apply hash function to map data to "buckets"

Goals: ① fast/simple calculation
② use all buckets "well"

Place Data in Cache by Hashing Address

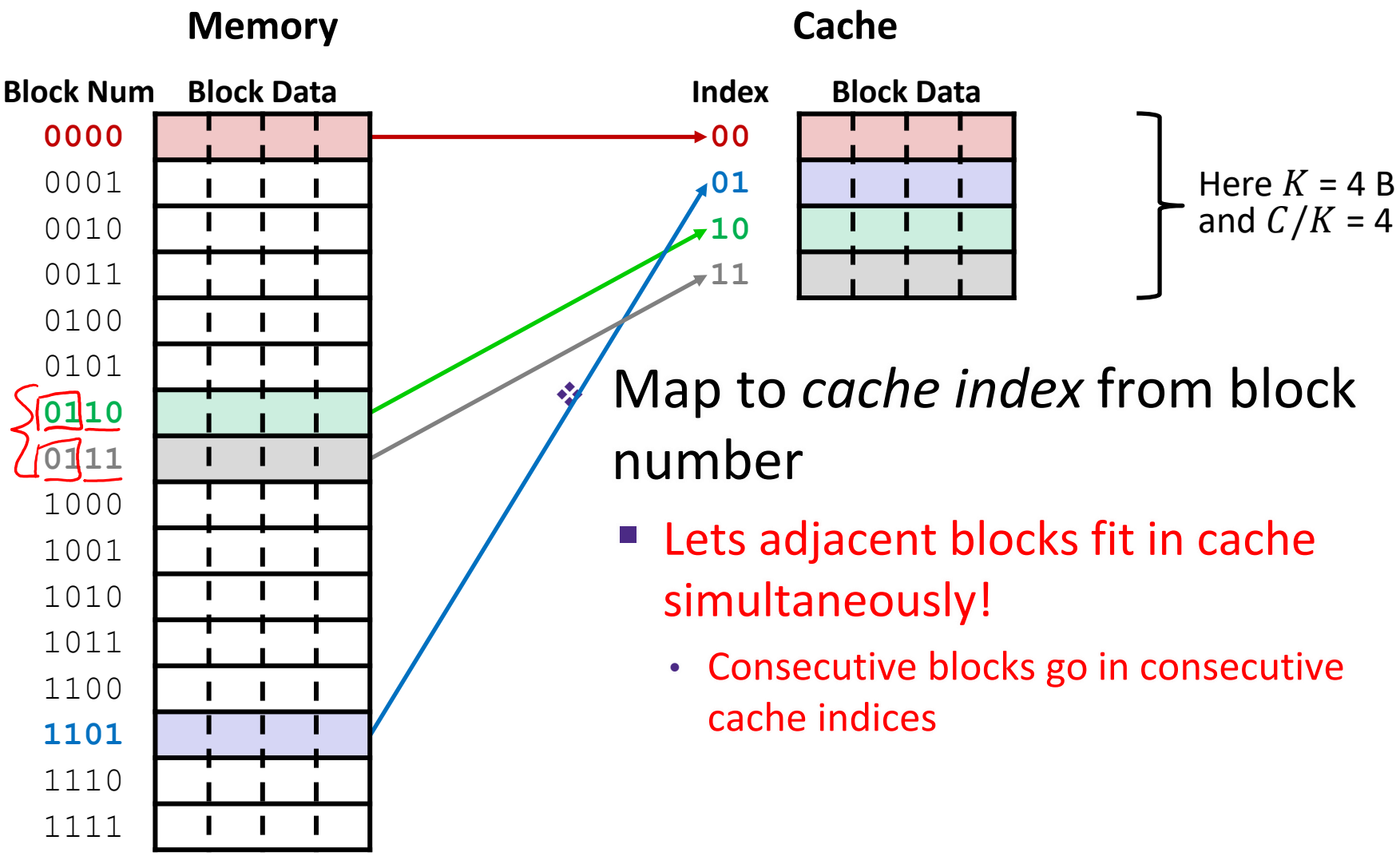


Map to *cache index* from block number $S = C/K$

- Use next $\log_2(C/K) = s$ bits
- (block number) mod (# blocks in cache)

How many bits do I need to specify every set/index in my cache?

Place Data in Cache by Hashing Address

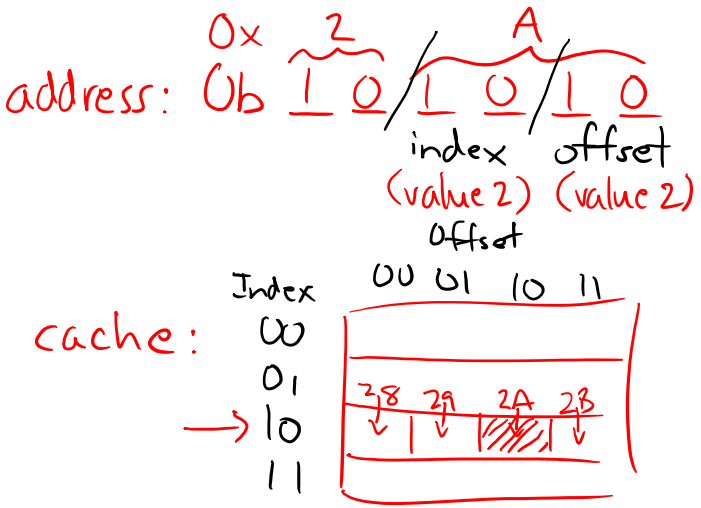


Map to *cache index* from block number

- Lets adjacent blocks fit in cache simultaneously!
  - Consecutive blocks go in consecutive cache indices

Polling Question

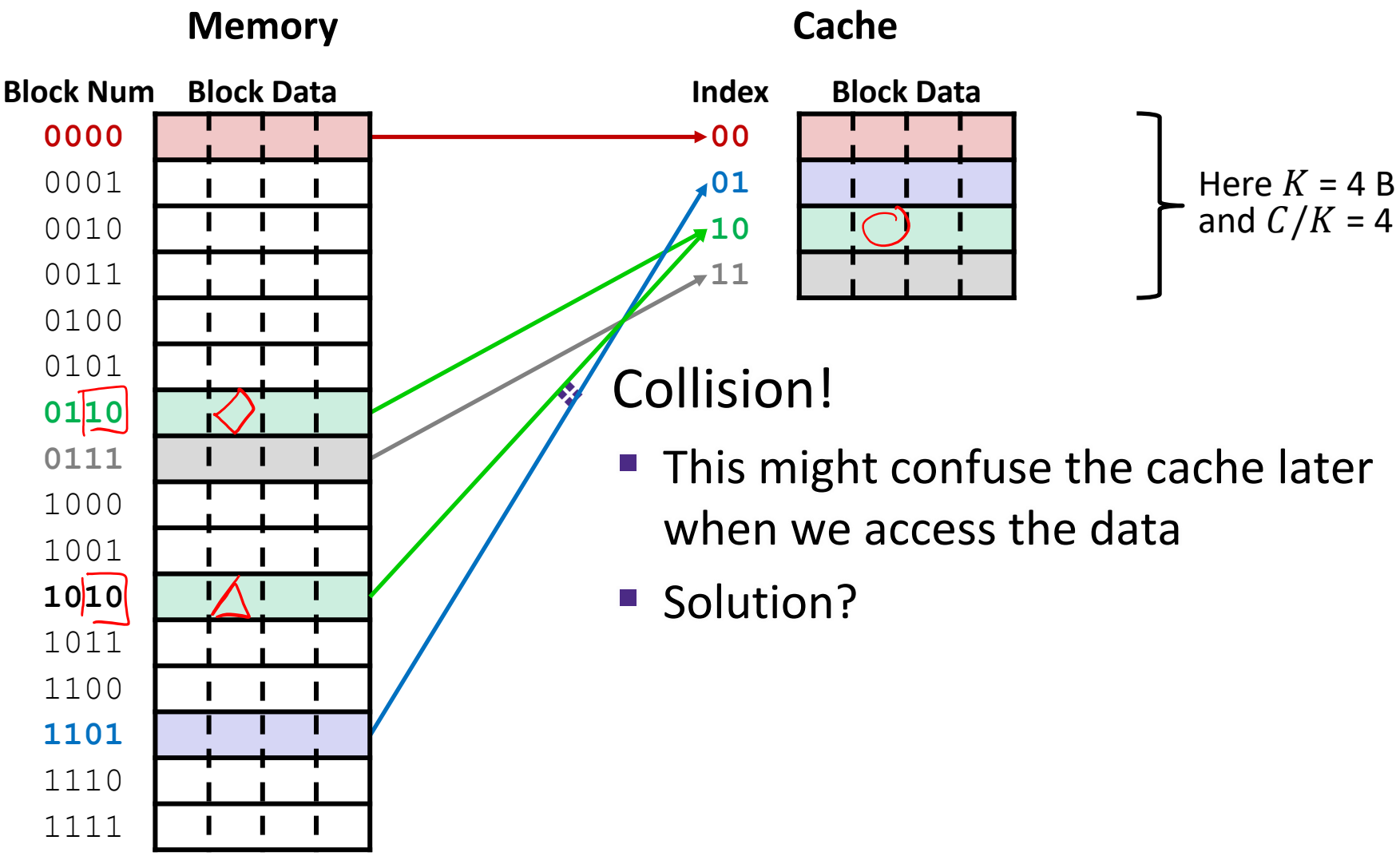
- 6-bit addresses, block size K = 4 B, and our cache holds S = 4 blocks. (m; $= C/K$, $s = \log_2(4) = 2$ bits)
- A request for address **0x2A** results in a cache miss. Which index does this block get loaded into and which 3 other addresses are loaded along with it?
  - Vote on Ed Lessons



Index 2 (0b10)

along with: 0b1010 00 = 0x28
01 = 0x29
11 = 0x2B

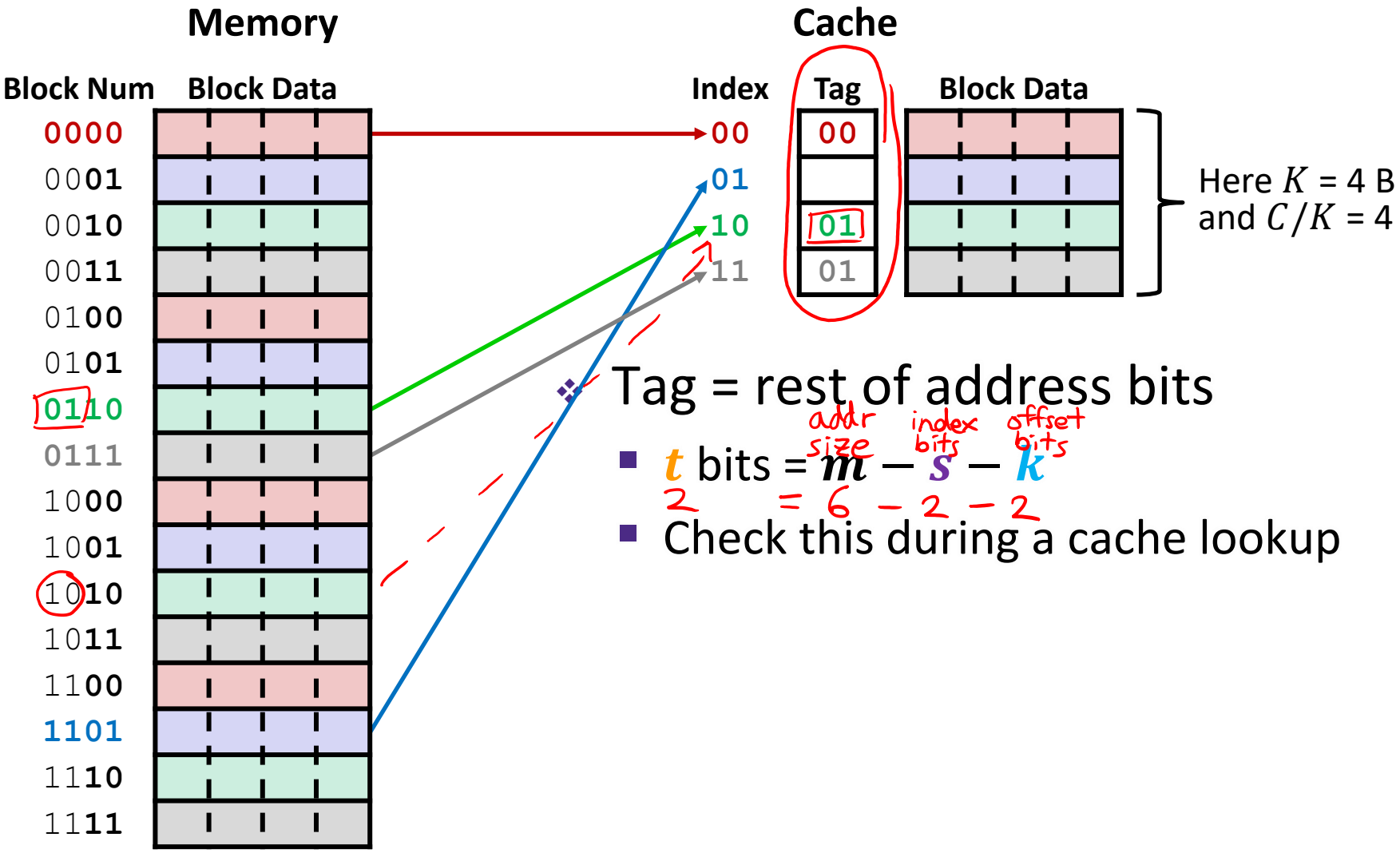
Place Data in Cache by Hashing Address



Collision!

- This might confuse the cache later when we access the data
- Solution?

Tags Differentiate Blocks in Same Index



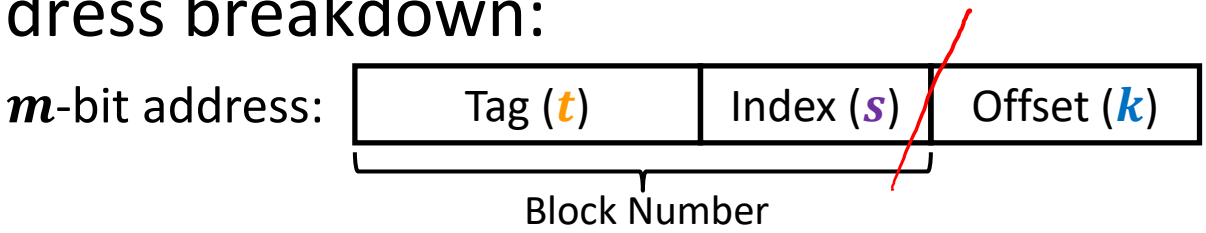
Tag = rest of address bits

- t bits = $m - s - k$ (addr size, index bits, offset bits)
  - $t = 6 - 2 - 2$
- Check this during a cache lookup

Checking for a Requested Address

- CPU sends address request for chunk of data
  - Address and requested data are not the same thing!
    - Analogy: your friend ≠ their phone number

- TIO address breakdown:



1. **Index** field tells you where to look in cache
2. **Tag** field lets you check that data is the block you want
3. **Offset** field selects specified start byte within block

- **Note:** t and s sizes will change based on hash function

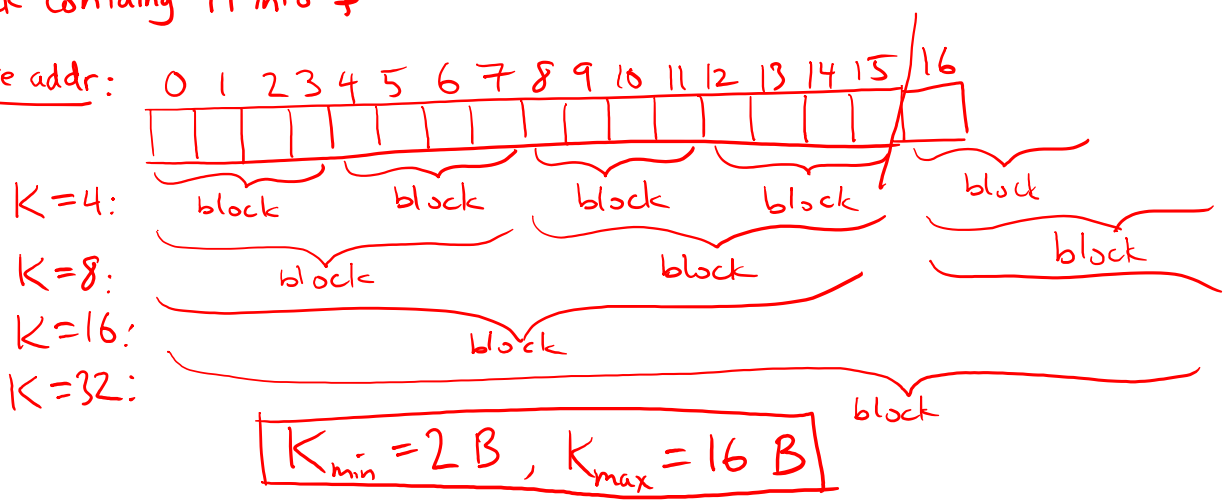
Cache Puzzle

- Based on the following behavior, which of the following block sizes is NOT possible for our cache?
  - Cache starts *empty*, also known as a *cold cache*
  - Access (addr: hit/miss) stream:
    - (14: miss), (15: hit), (16: miss)

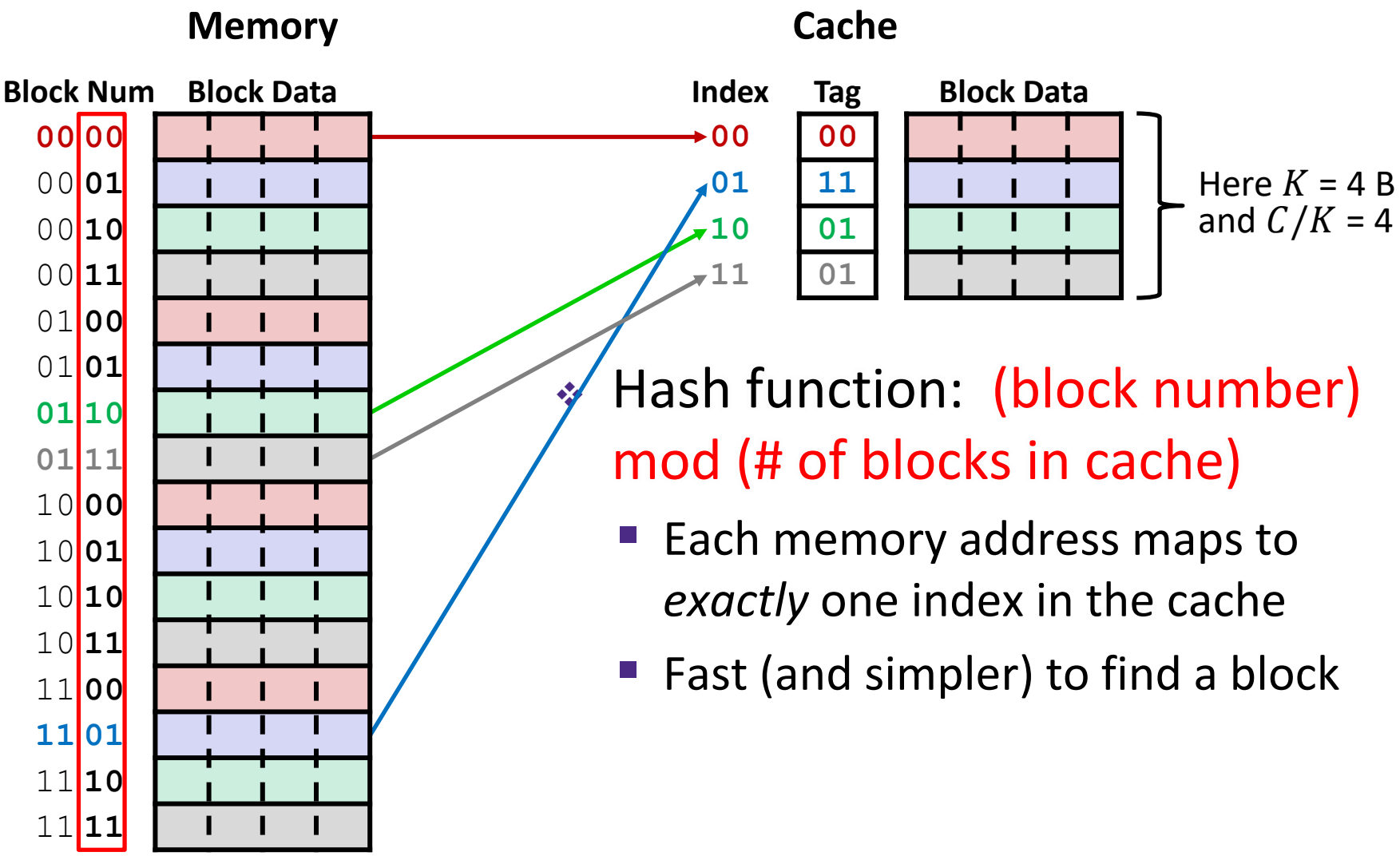
hit: block with data already in $
miss: data not in $, pulls block containing data from Mem

↳① pulls block containg 14 into $
↳② 14 & 15 are in the same block
↳③ 16 is in a different block

A. 4 bytes
B. 8 bytes
C. 16 bytes
D. 32 bytes
E. We're lost…



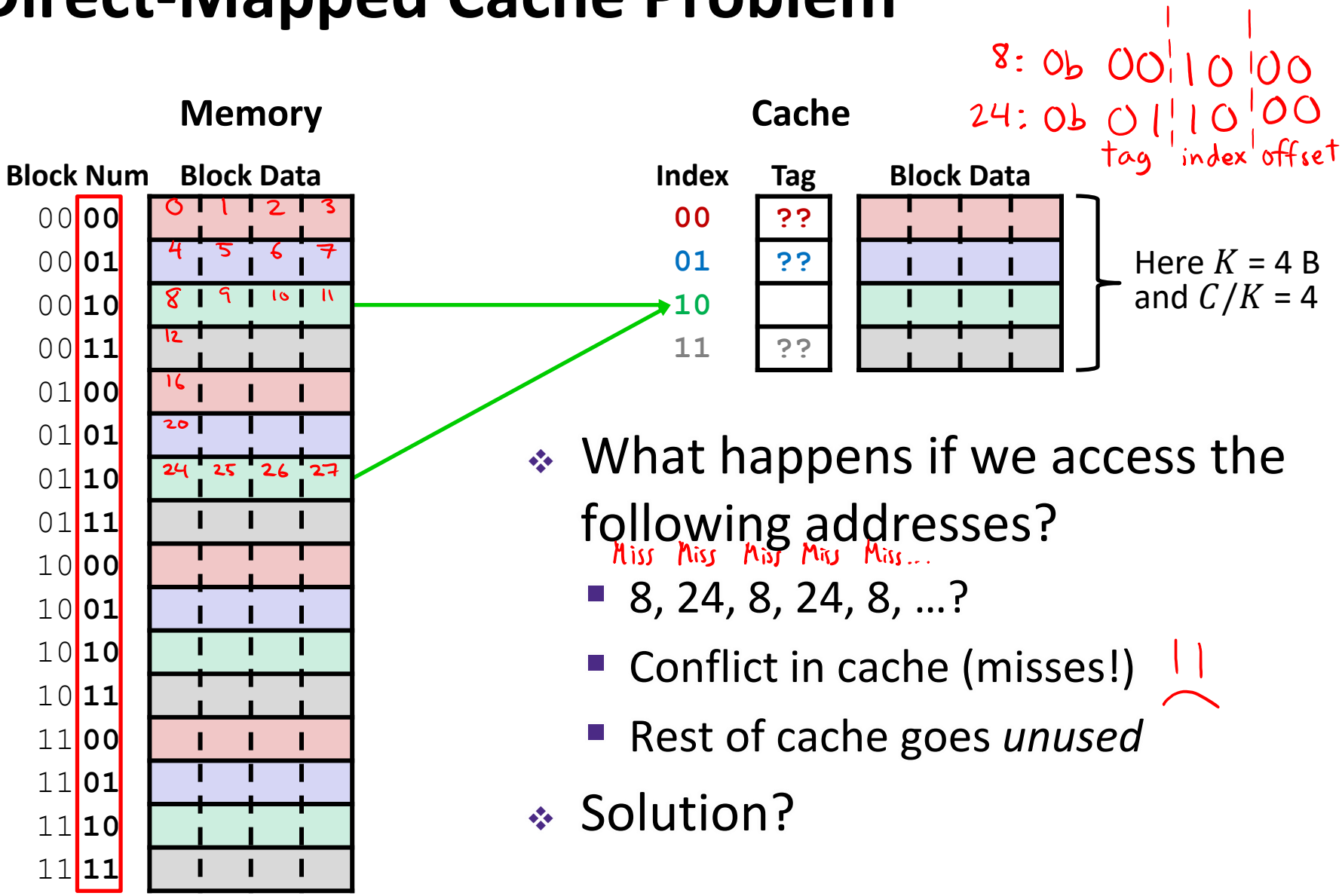
Summary: Direct-Mapped Cache



Hash function: (block number) mod (# of blocks in cache)

- Each memory address maps to *exactly* one index in the cache
- Fast (and simpler) to find a block

Direct-Mapped Cache Problem



- What happens if we access the following addresses? Miss Miss Miss Miss Miss...
  - 8, 24, 8, 24, 8, …?
  - Conflict in cache (misses!)
  - Rest of cache goes *unused*
- Solution?