Caches II

CSE 351 Summer 2020

Instructor:
Porter Jones

Teaching Assistants:
Amy Xu
Callum Walker
Sam Wolfson
Tim Mandzyuk

Administrivia

- Questions doc: https://tinyurl.com/CSE351-7-29

- hw15 due Friday (7/31) – 10:30am
- No homework due Monday!

- Lab 3 due Friday (7/31) – 11:59pm
  - You get to write some buffer overflow exploits!

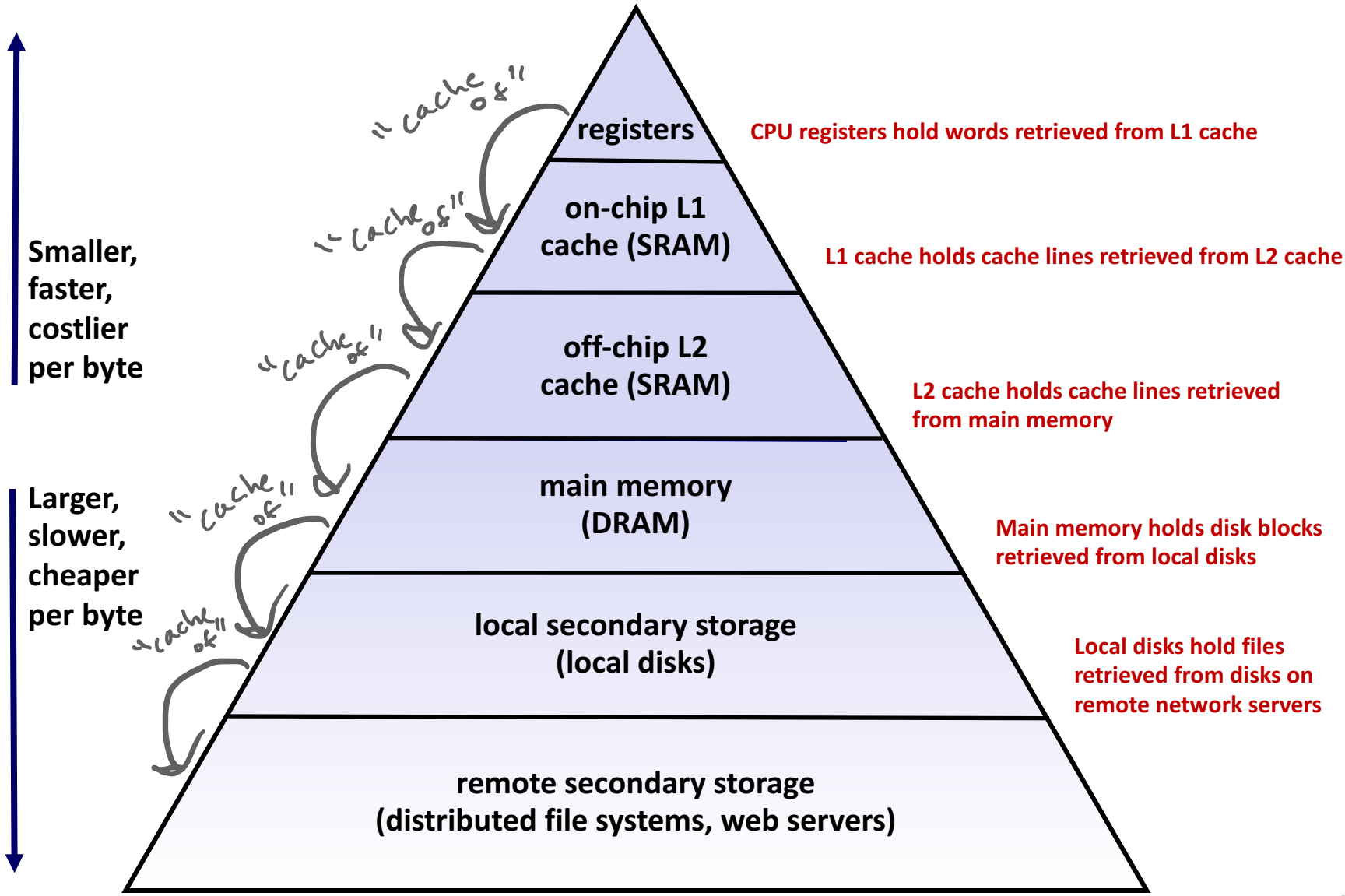
- Unit Summary 2 Due next Wednesday (8/5) – 11:59pm

Memory Hierarchies

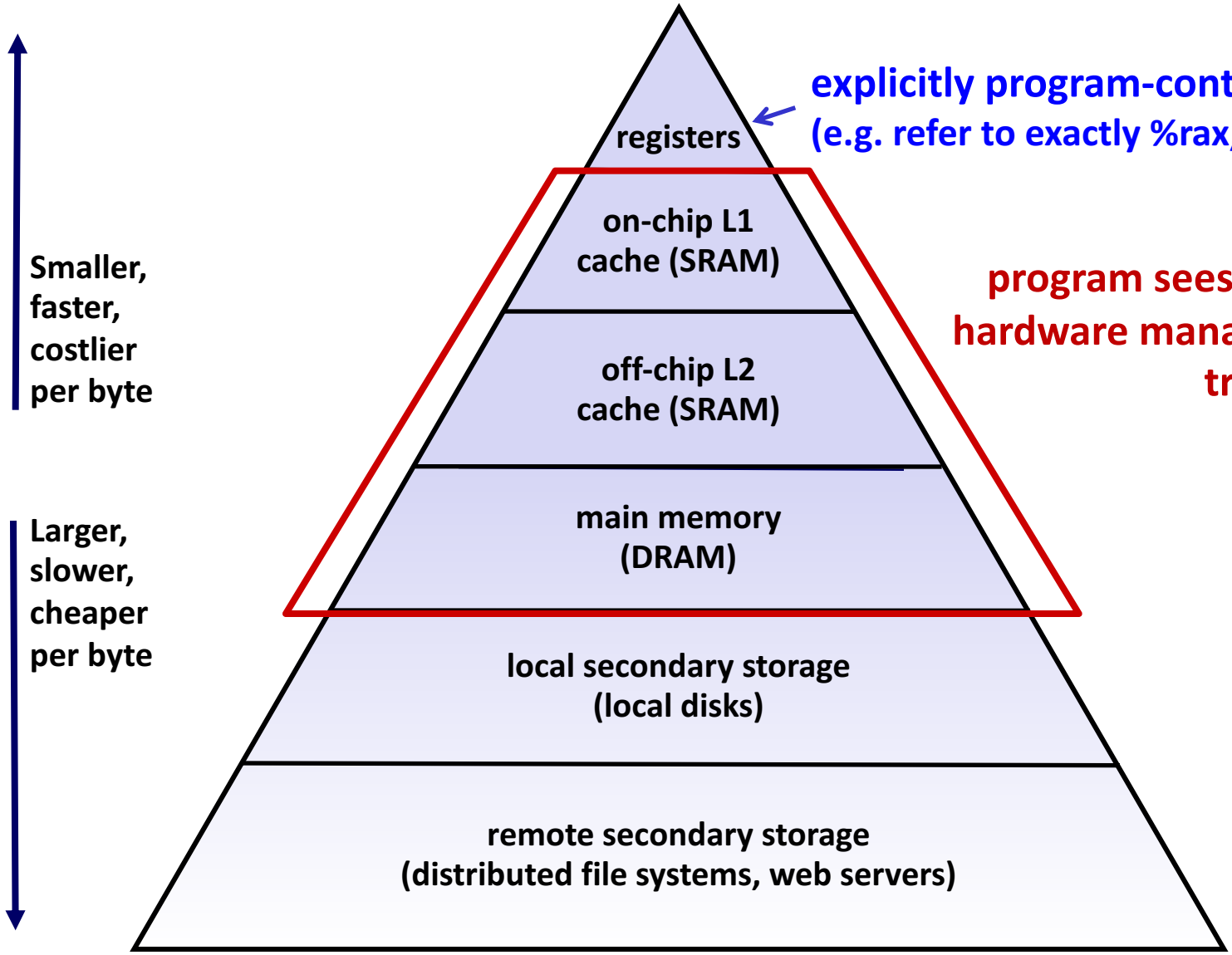
- Some fundamental and enduring properties of hardware and software systems:
  - Faster storage technologies almost always cost more per byte and have lower capacity
  - The gaps between memory technology speeds are widening
    - True for: registers ↔ cache, cache ↔ DRAM, DRAM ↔ disk, etc.
  - Well-written 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

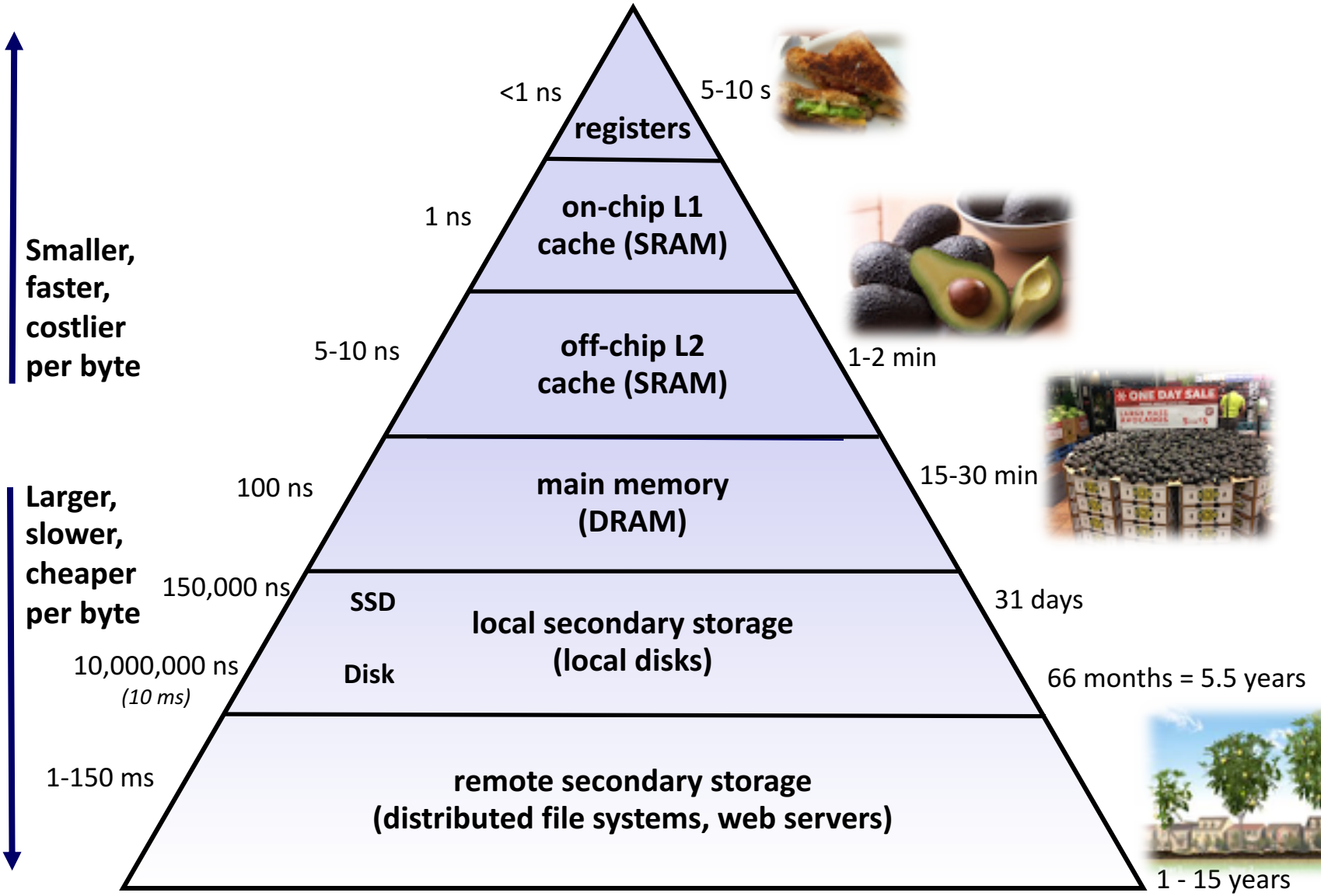
An Example Memory Hierarchy



An Example Memory Hierarchy



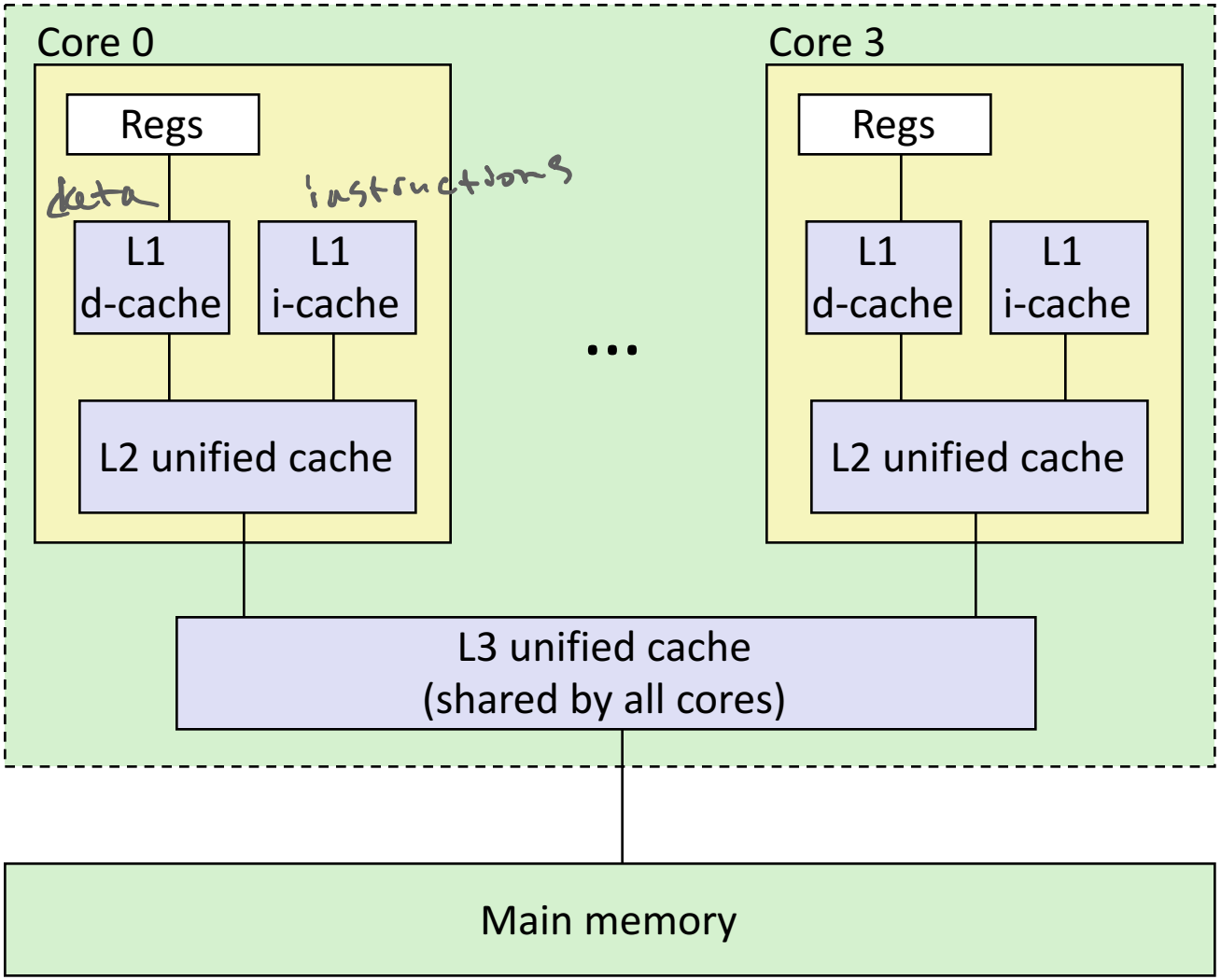
An Example Memory Hierarchy



Intel Core i7 Cache Hierarchy

Core: What executes instructions

Processor package



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

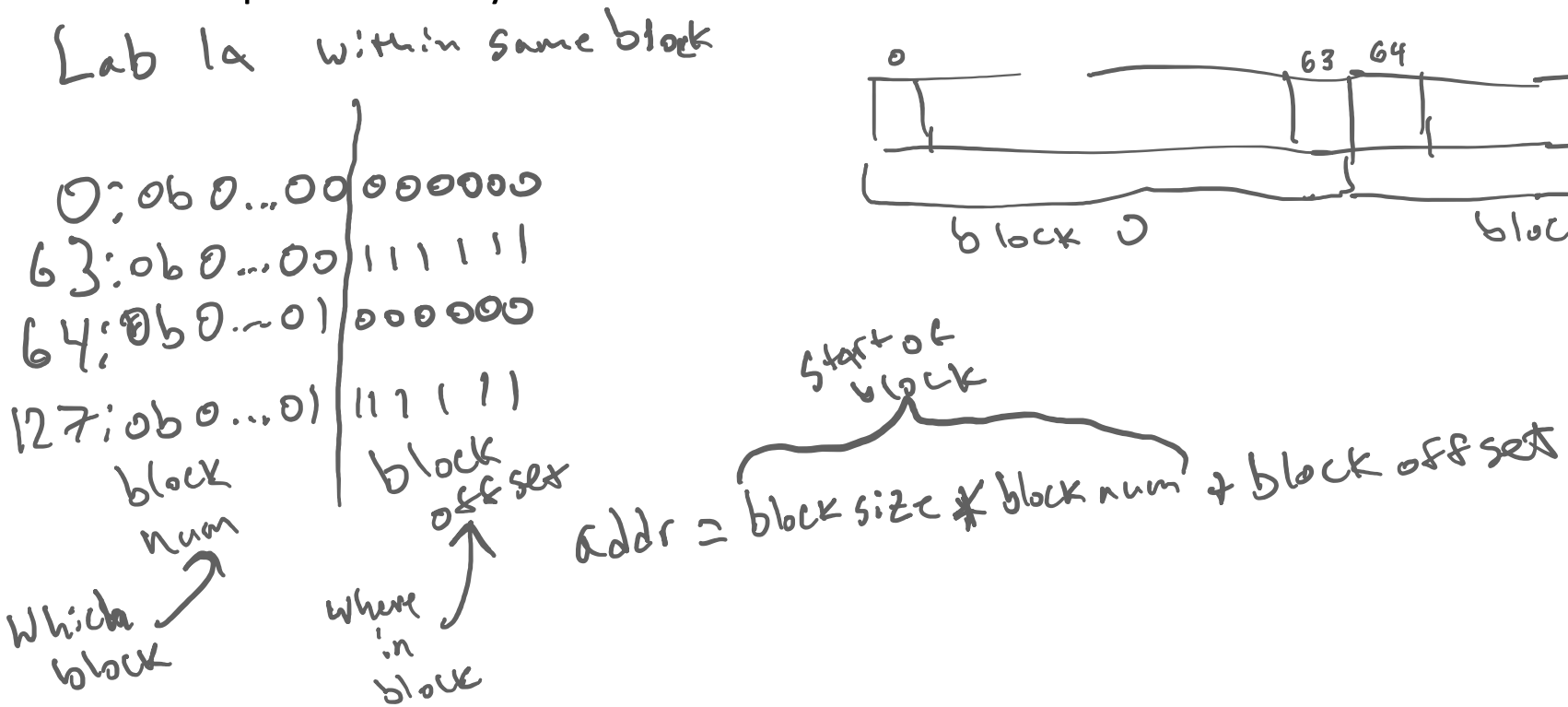
Making memory accesses fast!

- 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

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)

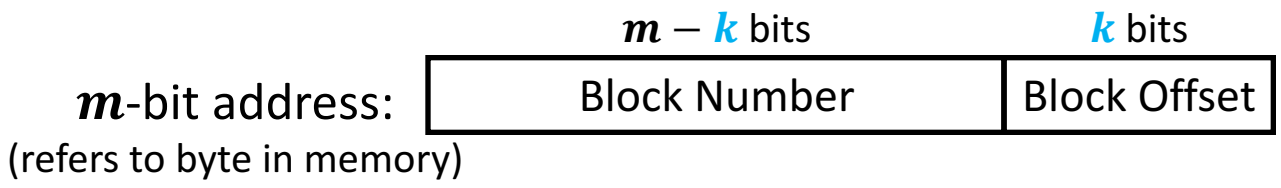
Note: The textbook uses "b" for offset bits

- 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 lowest n bits

- 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)

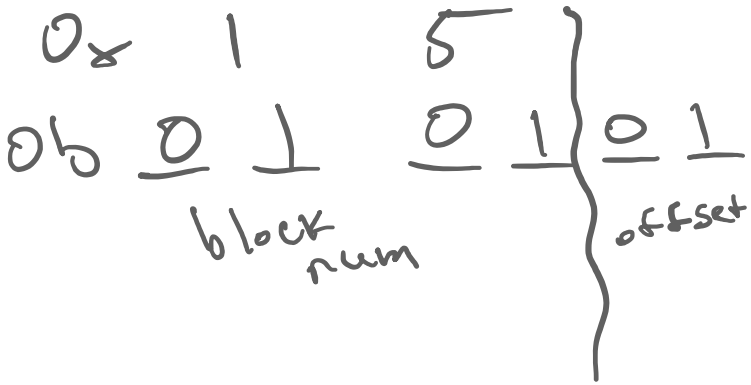
How many bits do I need to specify every byte in a block?



Polling Question [Cache II-a]

- If we have 6-bit addresses and block size K = 4 B, which block and byte does 0x15 refer to?
  - Vote at: http://pollev.com/pbjones

	Block Num	Block Offset
A.	1	1
B.	1	5
C.	5	1
D.	5	5
E.	We're lost…	



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

Cache Organization (2)

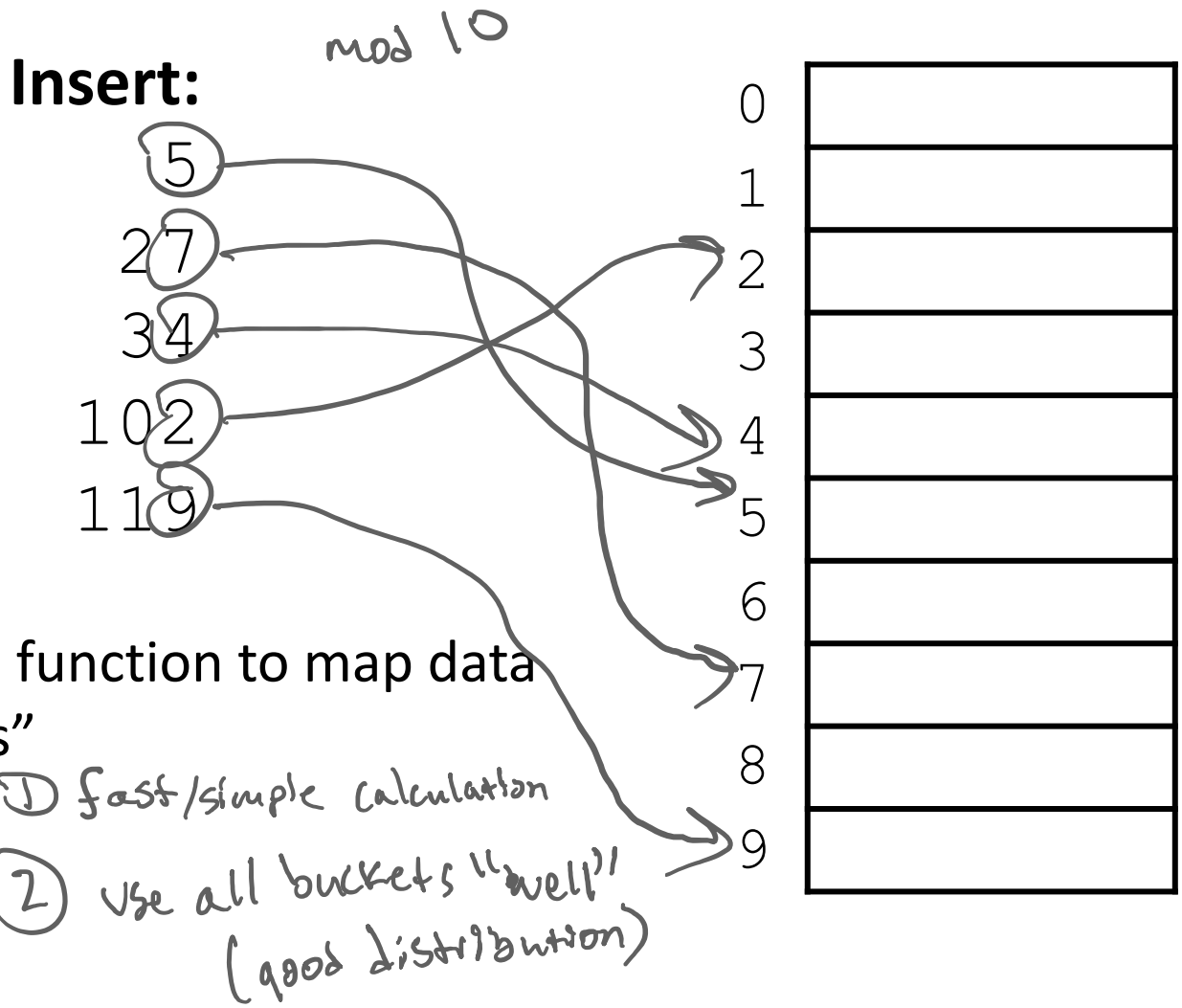
in bytes

- 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

$2^5 * 2^{10} / 2^6 = 2^9$ 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!

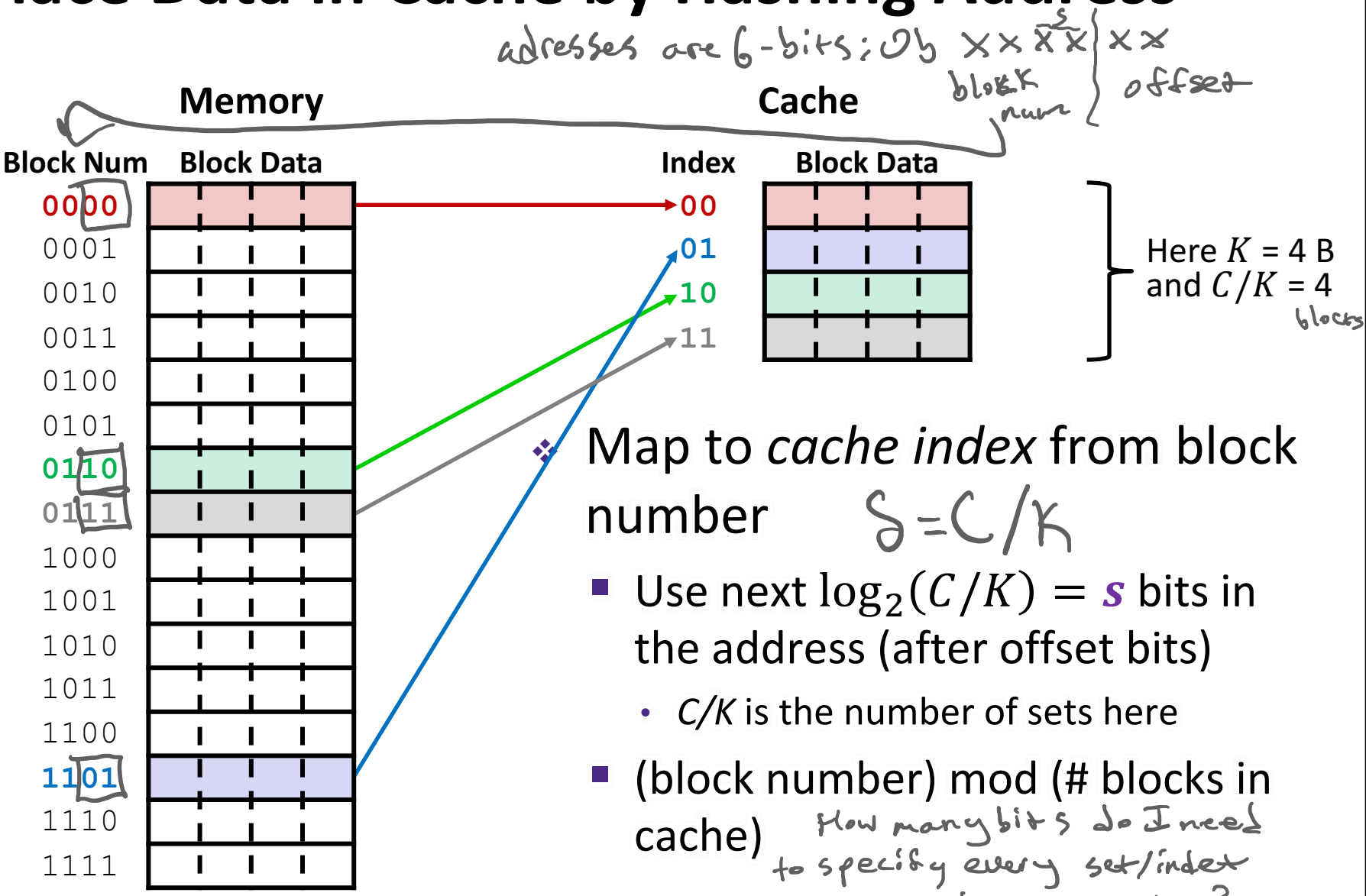
Review: Hash Tables for Fast Lookup



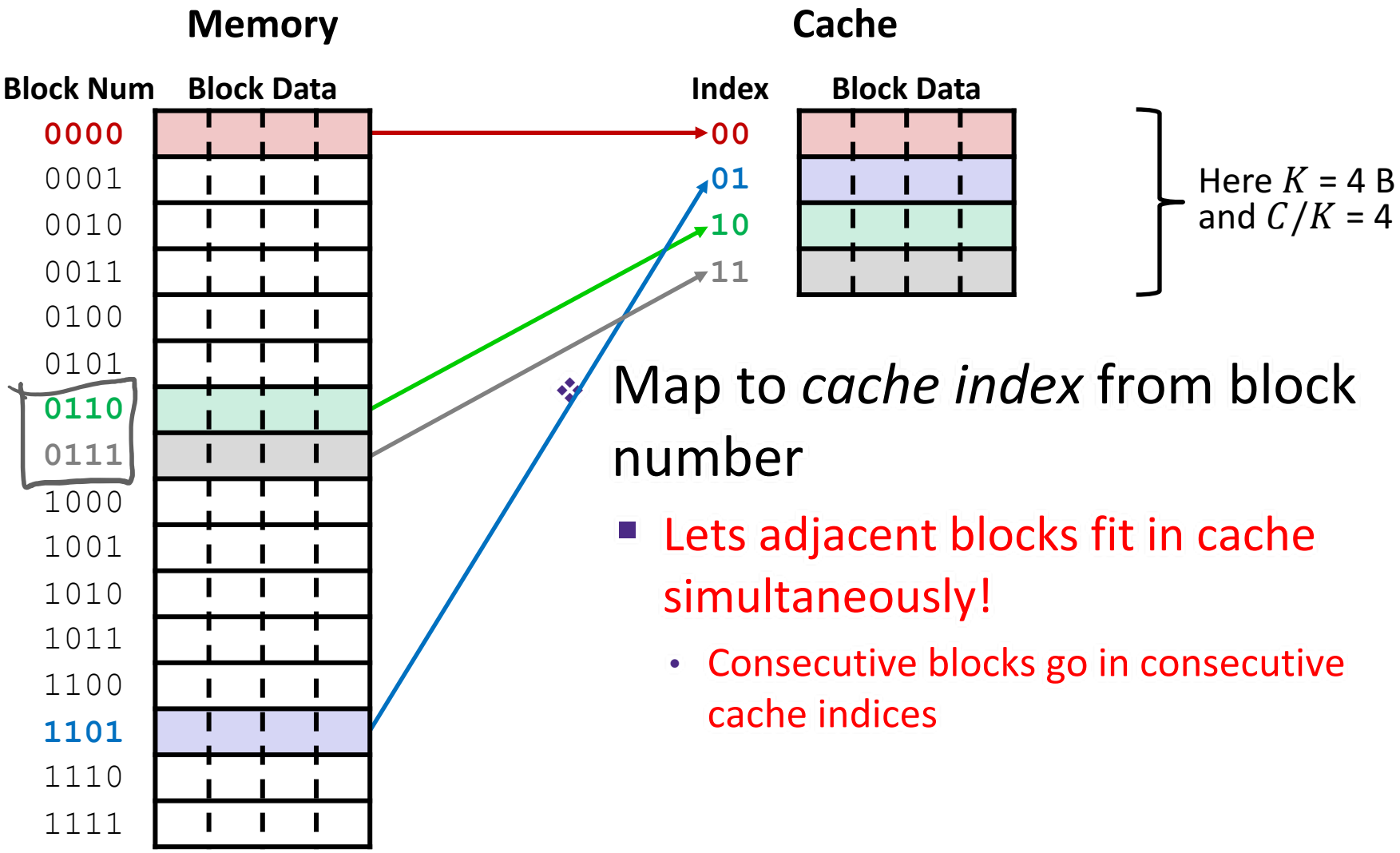
Apply hash function to map data to "buckets"

Goals: 1) fast/simple calculation
2) use all buckets "well" (good distribution)

Place Data in Cache by Hashing Address



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

Practice Question

num offset bits = $k = \log_2(4) = 2$

- 6-bit addresses, block size K = 4 B, and our cache holds S = 4 blocks. $= C/K$ $(C = 16B)$ num index bits $= s = \log_2(4) = 2$
- A request for address **0x2A** results in a cache miss. Which set index does this block get loaded into and which 3 other addresses are loaded along with it?
  - No voting for this question

address: 0b 10 | 10 | 10 (block num | index | offset)

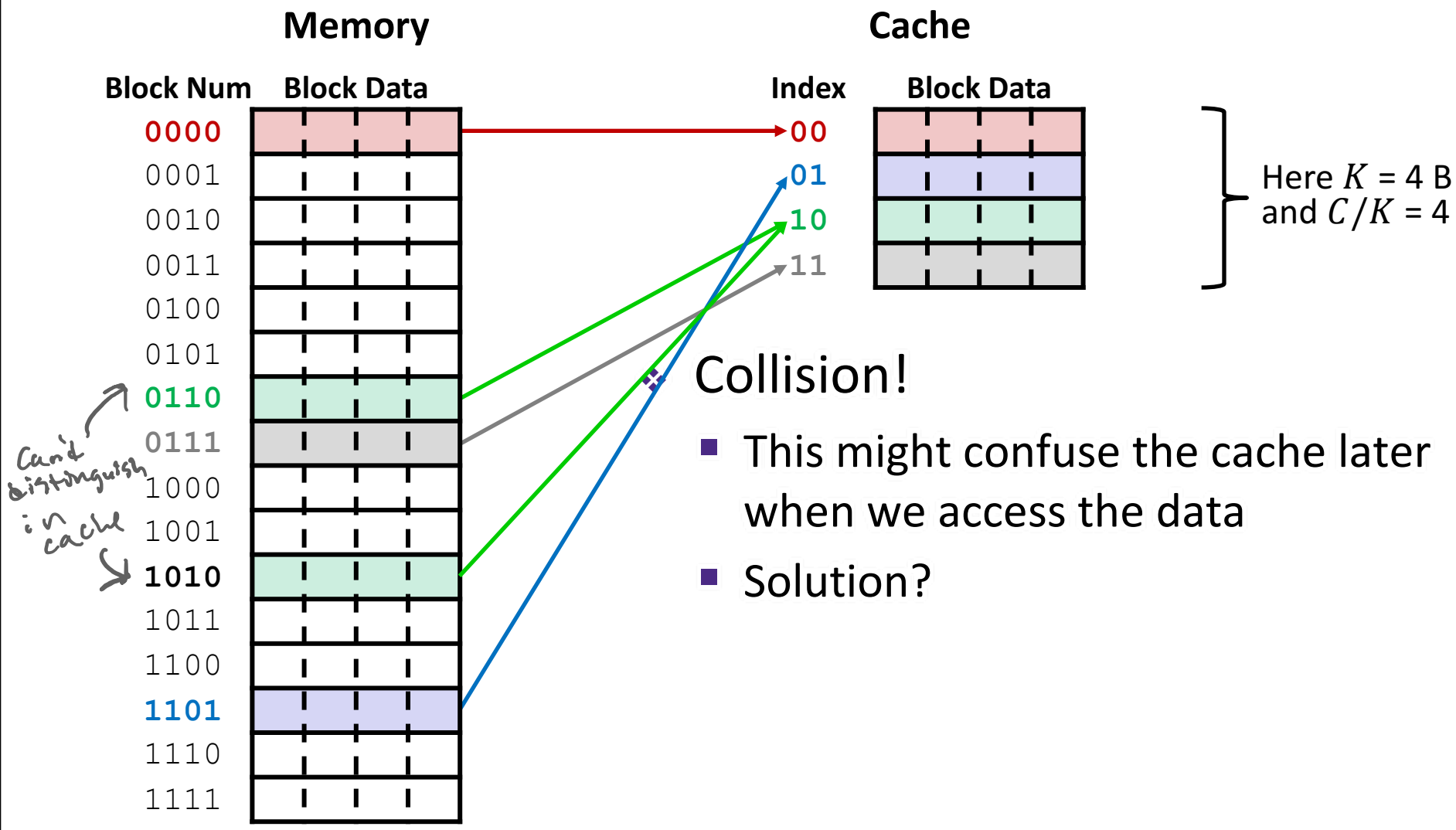
Index = 0b10 = 2

addresses w/ block num 1010

0b101000 = 28
0b101001 = 29
0b101010 = 2A
0b101011 = 2B

These are loaded into cache

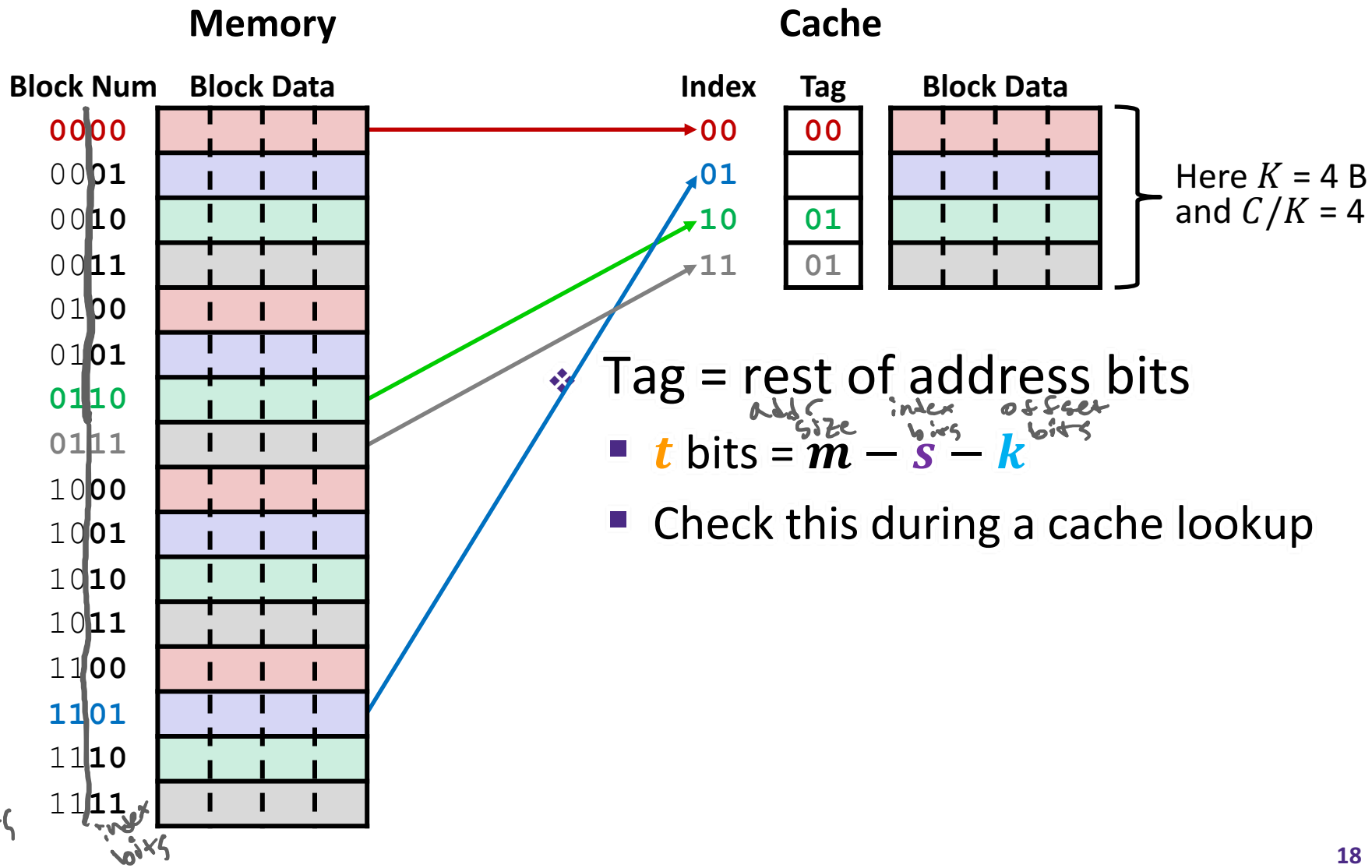
Place Data in Cache by Hashing Address



Collision!

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

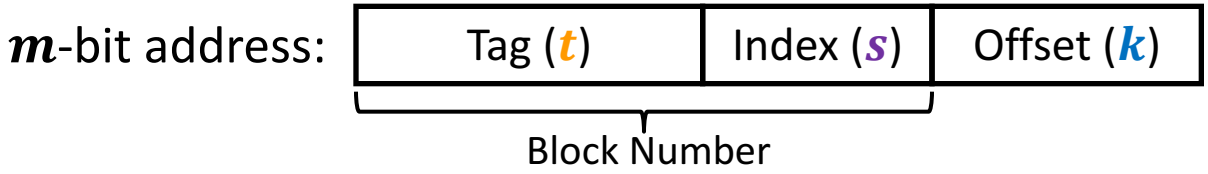
Tags Differentiate Blocks in Same Index



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:

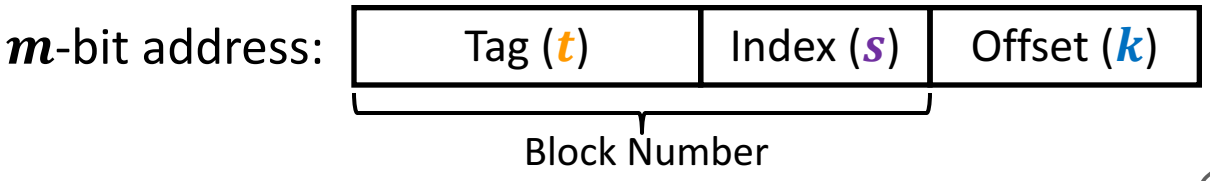


  - **Index** field tells you where to look in cache
  - **Tag** field lets you check that data is the block you want
  - **Offset** field selects specified start byte within block

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

Checking for a Requested Address Example

- Using 8-bit addresses.
- Cache Params: block size (K) = 4 B, cache size (C) = 32 B (which means number of sets is C/K = 8 sets).
  - Offset bits (k) = $\log_2(K)$ = $\log_2(4) = 2$ bits
  - Index bits (s) = $\log_2(num\ sets)$ = $\log_2(8) = 3$ bits
  - Tag bits (t) = Rest of the bits in the address = $8 - 2 - 3 = 3$ bits



tag index offset
0b10 | 11 | 10

- What are the fields for address 0xBA?
  - Tag bits (unique id for block): 0b101 = 5
  - Index bits (cache set block maps to): 0b110 = 6
  - Offset bits (byte offset within block): 0b10 = 2

Cache Puzzle [Cache II–b] Vote at http://pollev.com/pbjones

- 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 is already in cache!

miss: block is not in cache, pulls block from memory and puts it in cache

① pulls block w/14 into cache

② 15 is in the same block as 14

③ 16 is not in block w/14 and 15

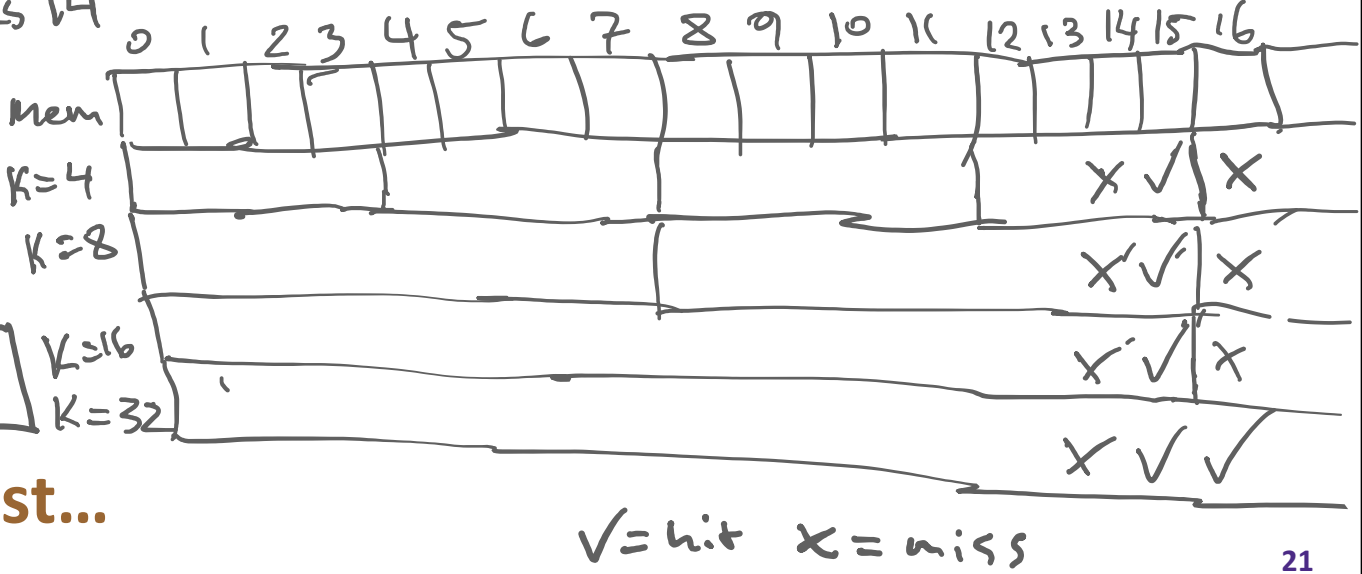
A. 4 bytes

B. 8 bytes

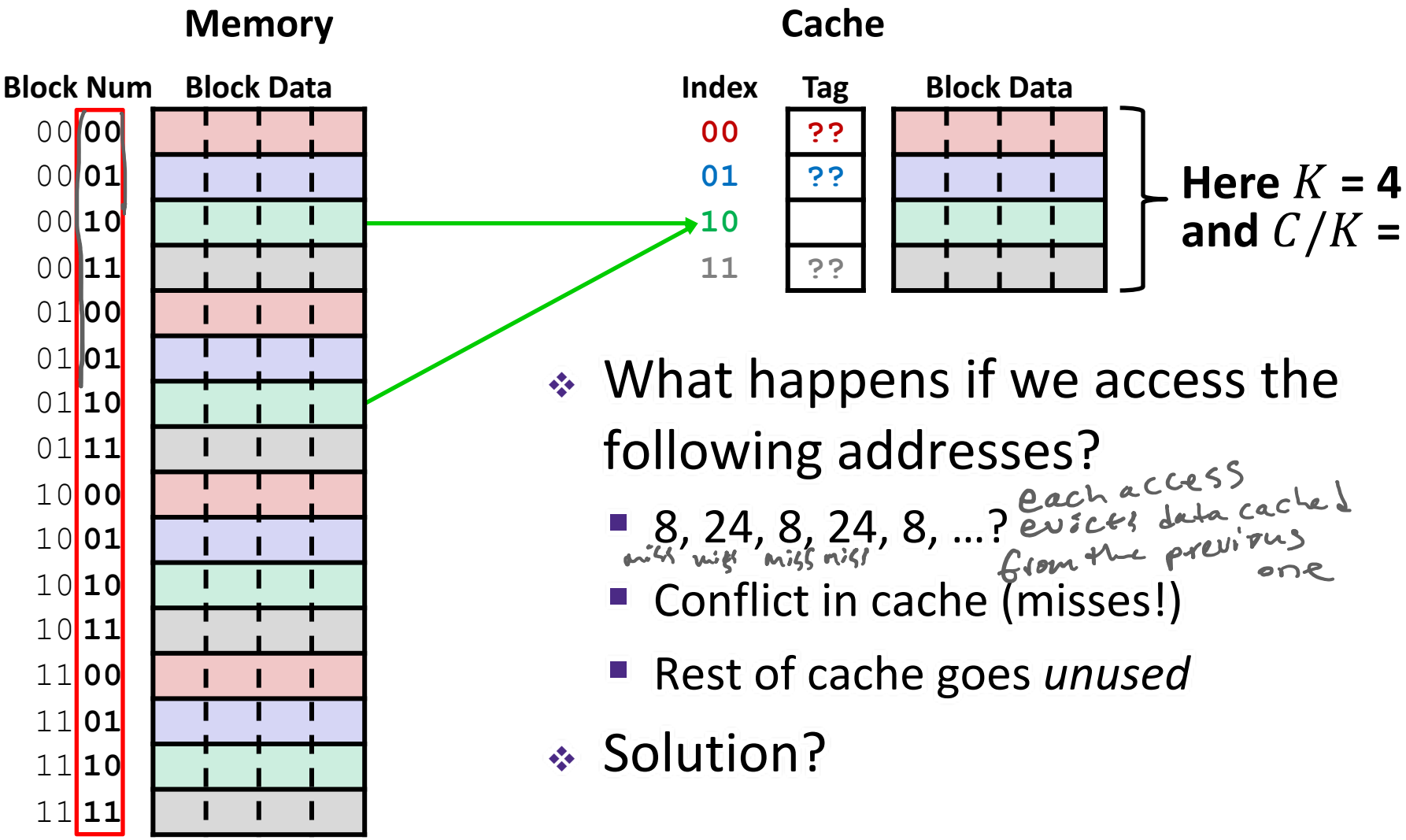
C. 16 bytes

D. 32 bytes

E. We're lost…



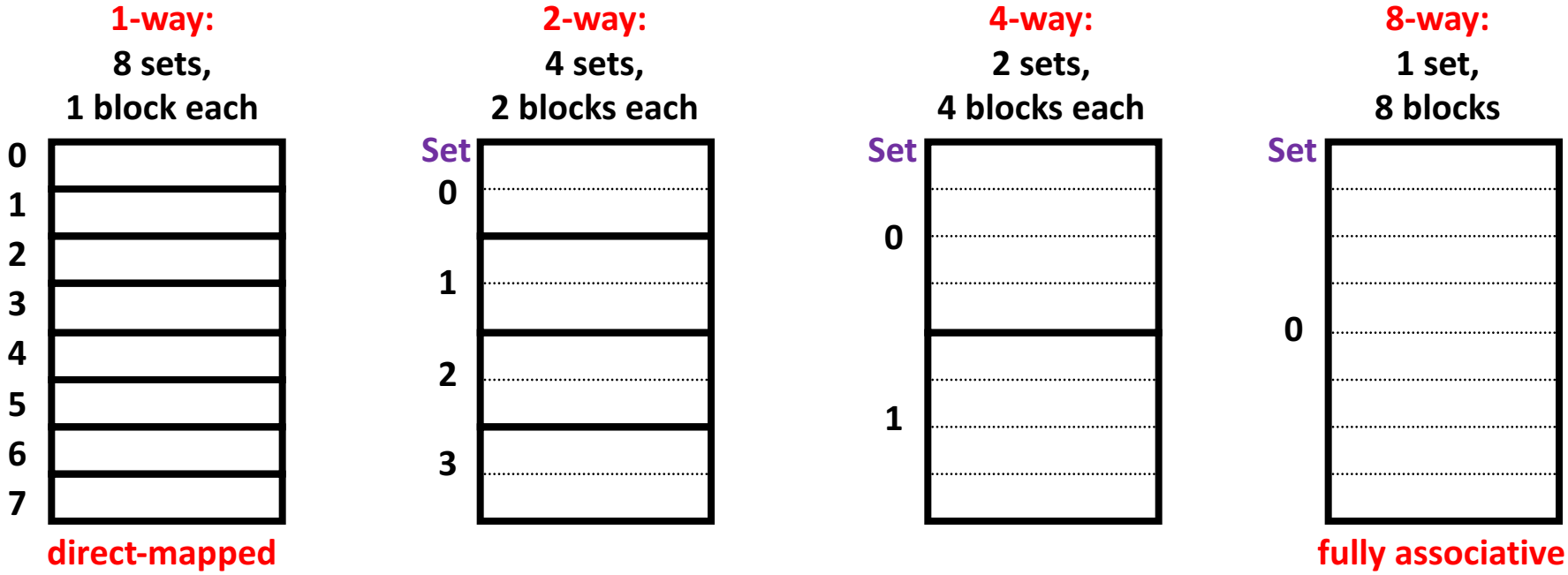
Direct-Mapped Cache Problem



- What happens if we access the following addresses?
  - 8, 24, 8, 24, 8, …? (miss miss miss miss) each access evicts data cached from the previous one
  - Conflict in cache (misses!)
  - Rest of cache goes *unused*
- Solution?

Associativity

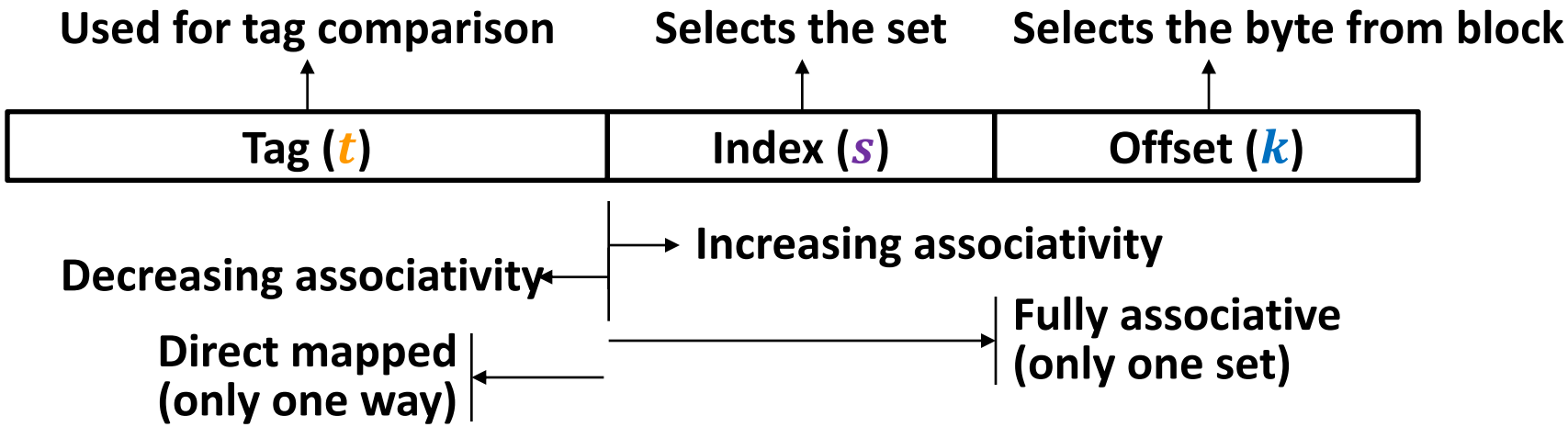
- What if we could store data in any place in the cache?
  - More complicated hardware = more power consumed, slower
- So we *combine* the two ideas:
  - Each address maps to exactly one **set**
  - Each set can store block in more than one **way**



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 $S = C/K/E$
  - Use lowest $\log_2(C/K/E)$ = s bits of block address
    - Direct-mapped: E = 1, so s = $\log_2(C/K)$ as we saw previously
    - Fully associative: $E = C/K$, so s = 0 bits



Example Placement

block size:	16 B
capacity:	8 blocks
address:	16 bits

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

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

m-bit address:	Tag (t)	Index (s)	Offset (k)

s = ? 3 bits — Direct-mapped

Set	Tag	Data
000		
001		
010		
011	✓	✓
100		
101		
110		
111		

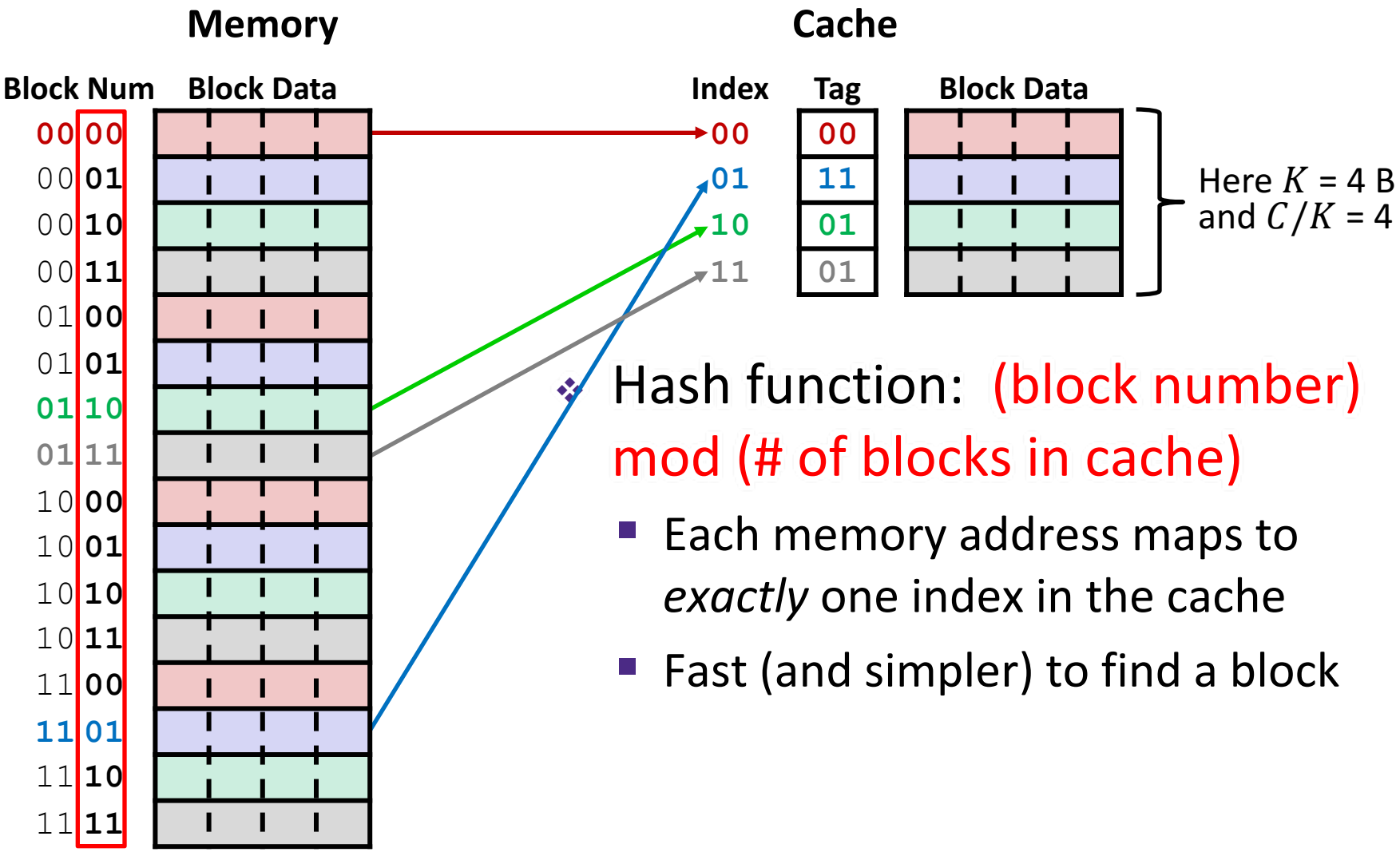
s = ? 2 bits — 2-way set associative

Set	Tag	Data
00		
01		
10		
11	✓	✓

s = ? 1 bit — 4-way set associative

Set	Tag	Data
0		
1	✓	✓
	✓	✓
	✓	✓
	✓	✓

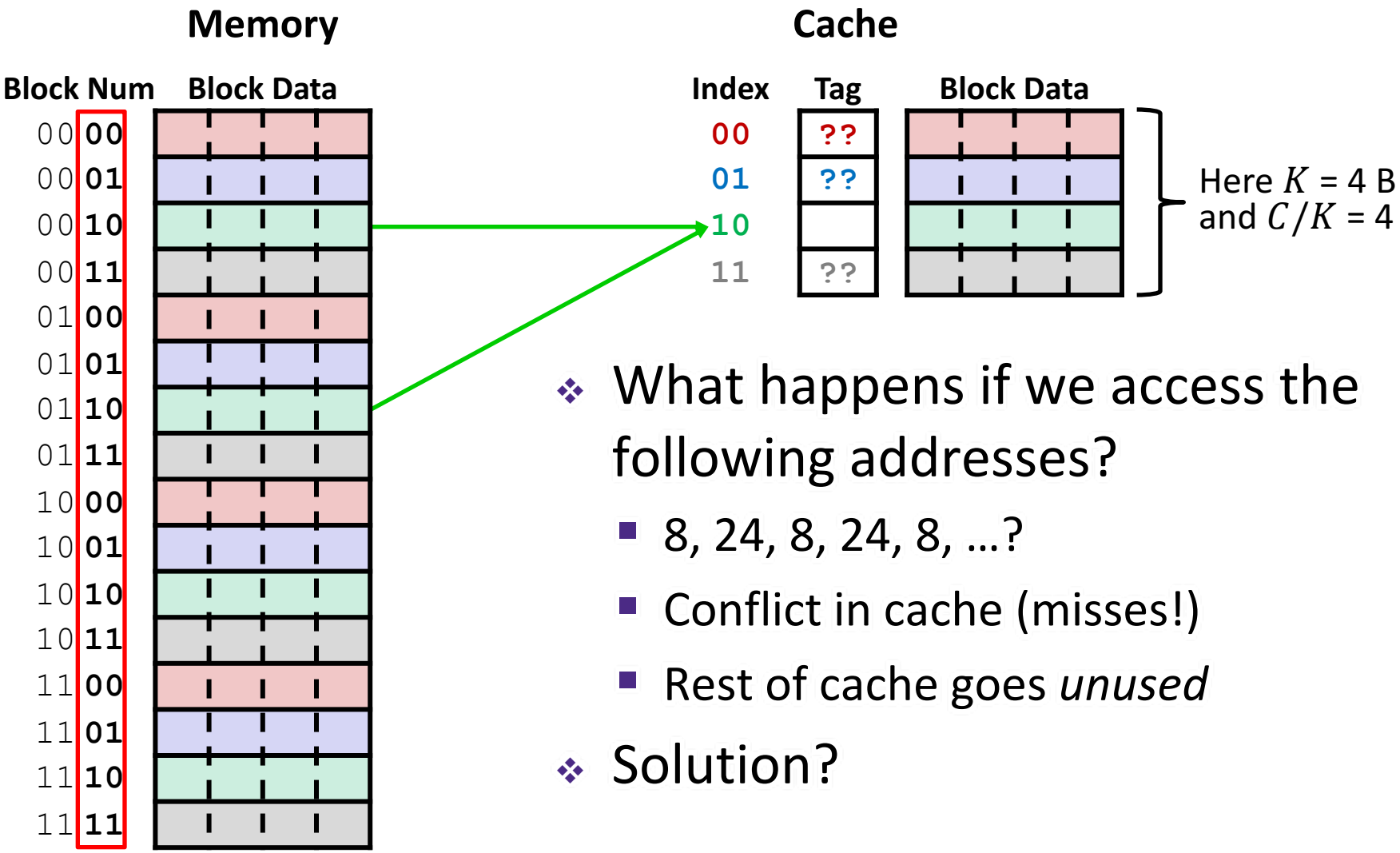
Direct-Mapped Cache



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?
  - 8, 24, 8, 24, 8, …?
  - Conflict in cache (misses!)
  - Rest of cache goes *unused*
- Solution?

Notes Diagrams