Caches II

CSE 351 Autumn 2018

Instructor:

Justin Hsia

Teaching Assistants:

Akshat Aggarwal
An Wang
Andrew Hu
Brian Dai
Britt Henderson
James Shin
Kevin Bi
Kory Watson
Riley Germundson
Sophie Tian
Teagan Horkan

Administrivia

- Homework 4 released tomorrow (Structs, Caches)
- Lab 3 due Friday (11/9)

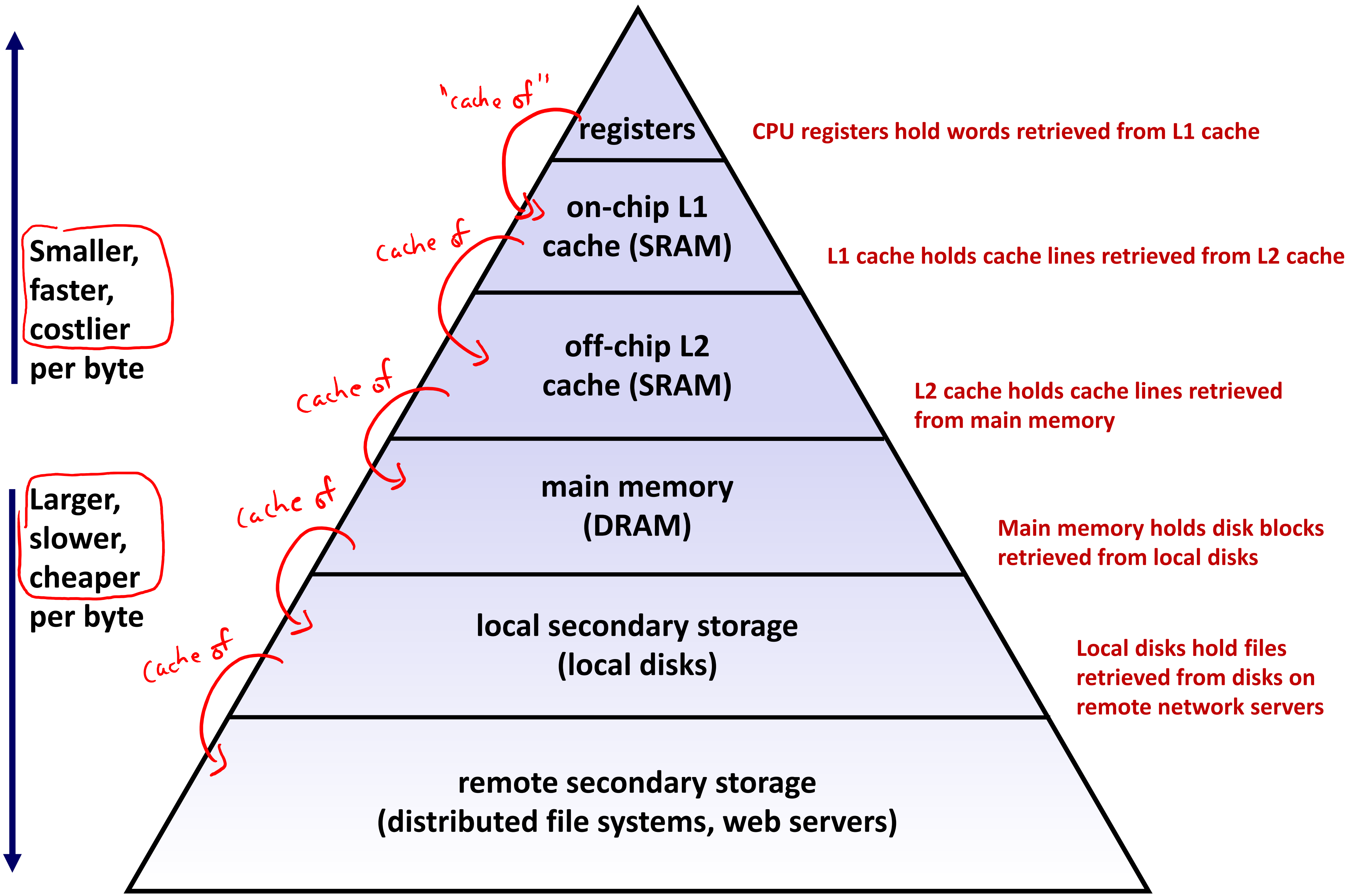
- **Mid-Quarter Survey Feedback**
  - Pace is “moderate” to “a bit too fast”
  - Canvas quiz answer keys are annoying, but instant homework feedback is great

Memory Hierarchies

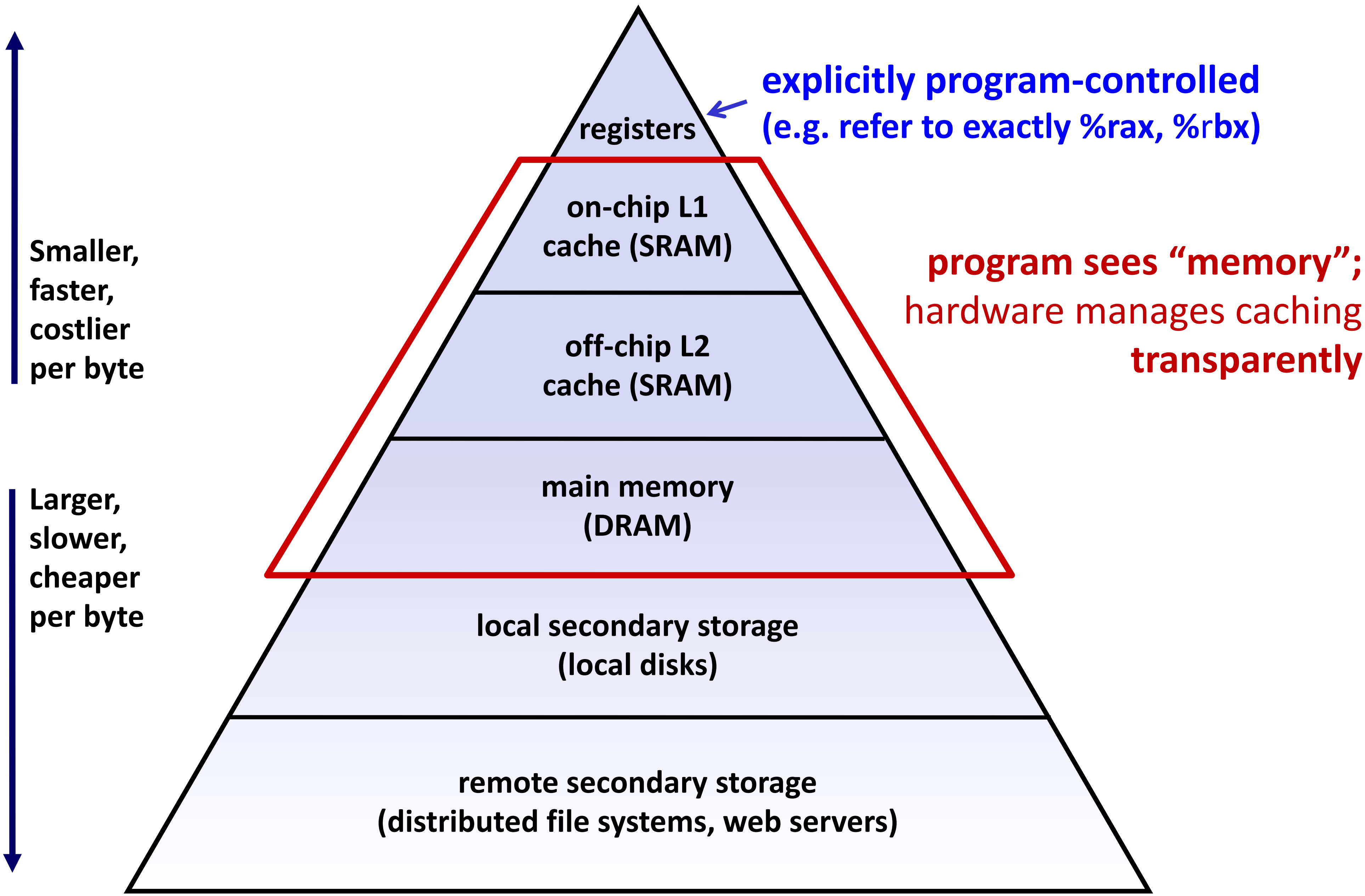
- Some fundamental and enduring properties of hardware and software systems:
  - Faster storage technologies almost always cost more per byte and have lower capacity
  - The gaps between memory technology speeds are widening
    - True for: registers ↔ cache, cache ↔ DRAM, DRAM ↔ disk, etc.
  - Well-written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

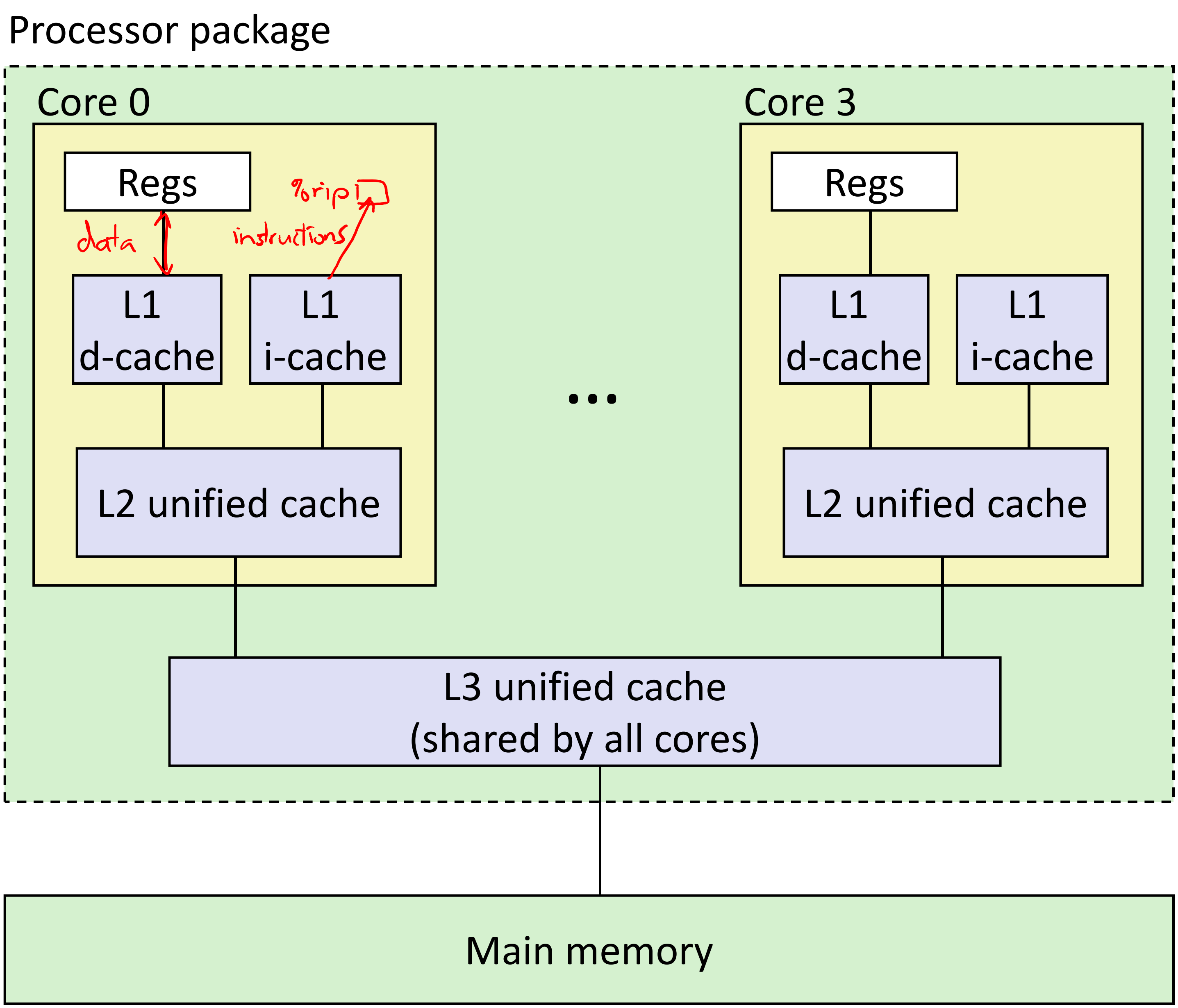
An Example Memory Hierarchy



An Example Memory Hierarchy



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

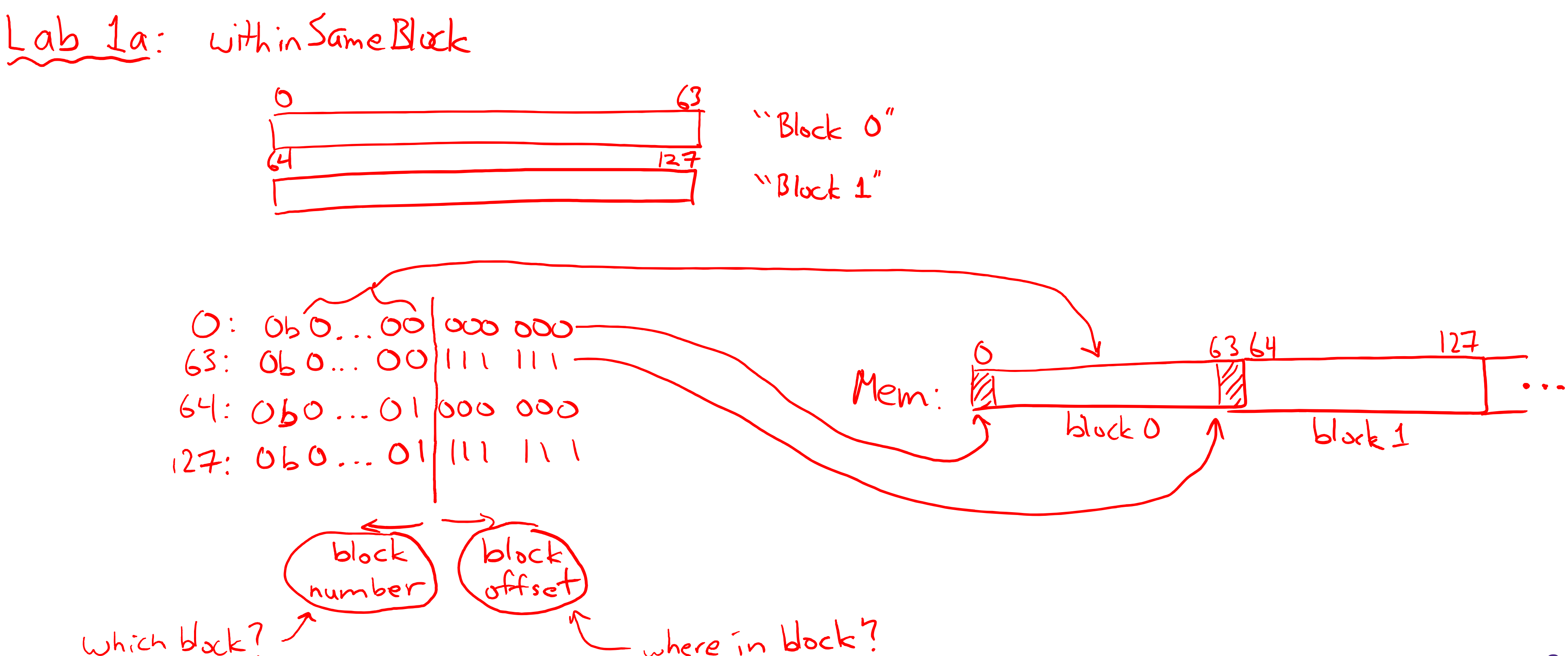
Making memory accesses fast!

- 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

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!
- 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

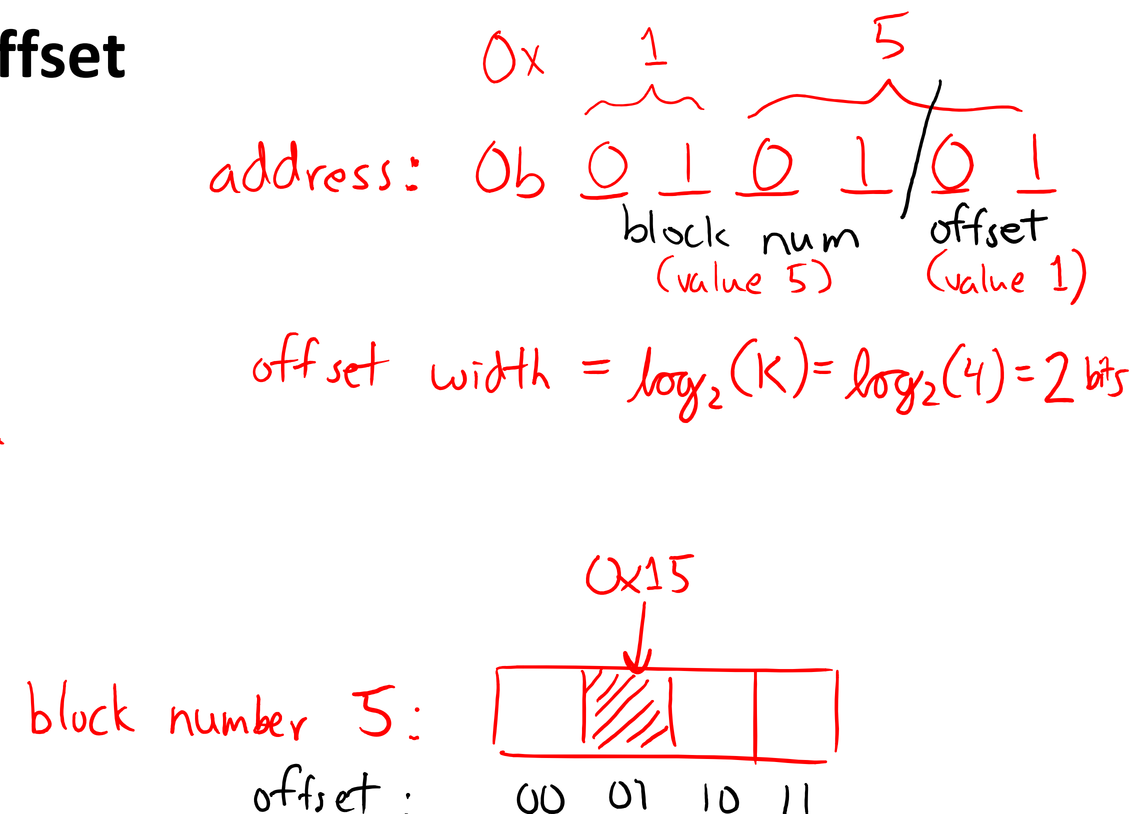
How many bits do I need to specify every byte in a block?

	$m - k$ bits	k bits
m-bit address: (refers to byte in memory)	Block Number	Block Offset

Peer Instruction Question

- 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/justinh

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



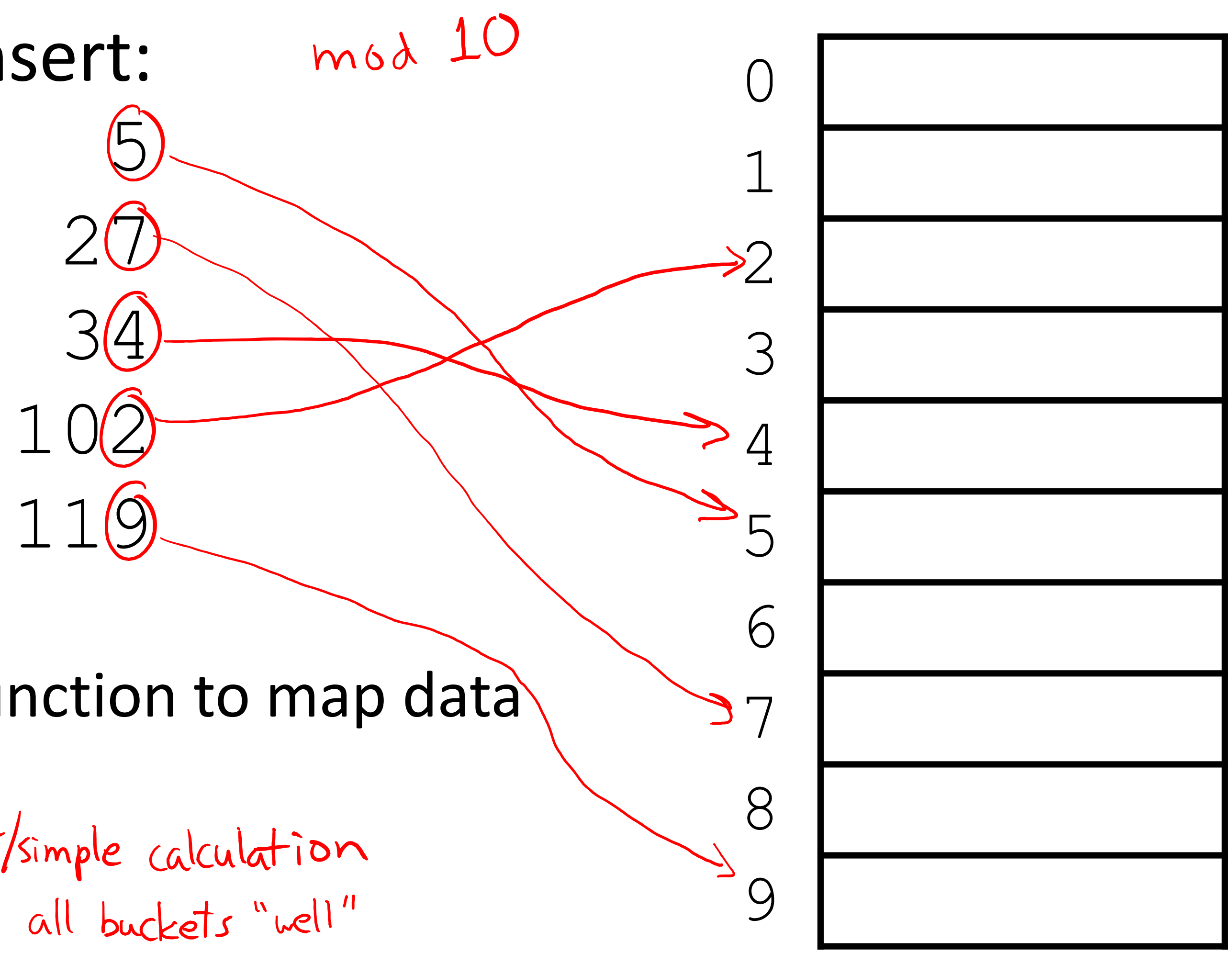
Cache Organization (2)

- 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)
  - <u>Example</u>: C = 32 KiB = 512 blocks if using 64-B blocks

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

- 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!

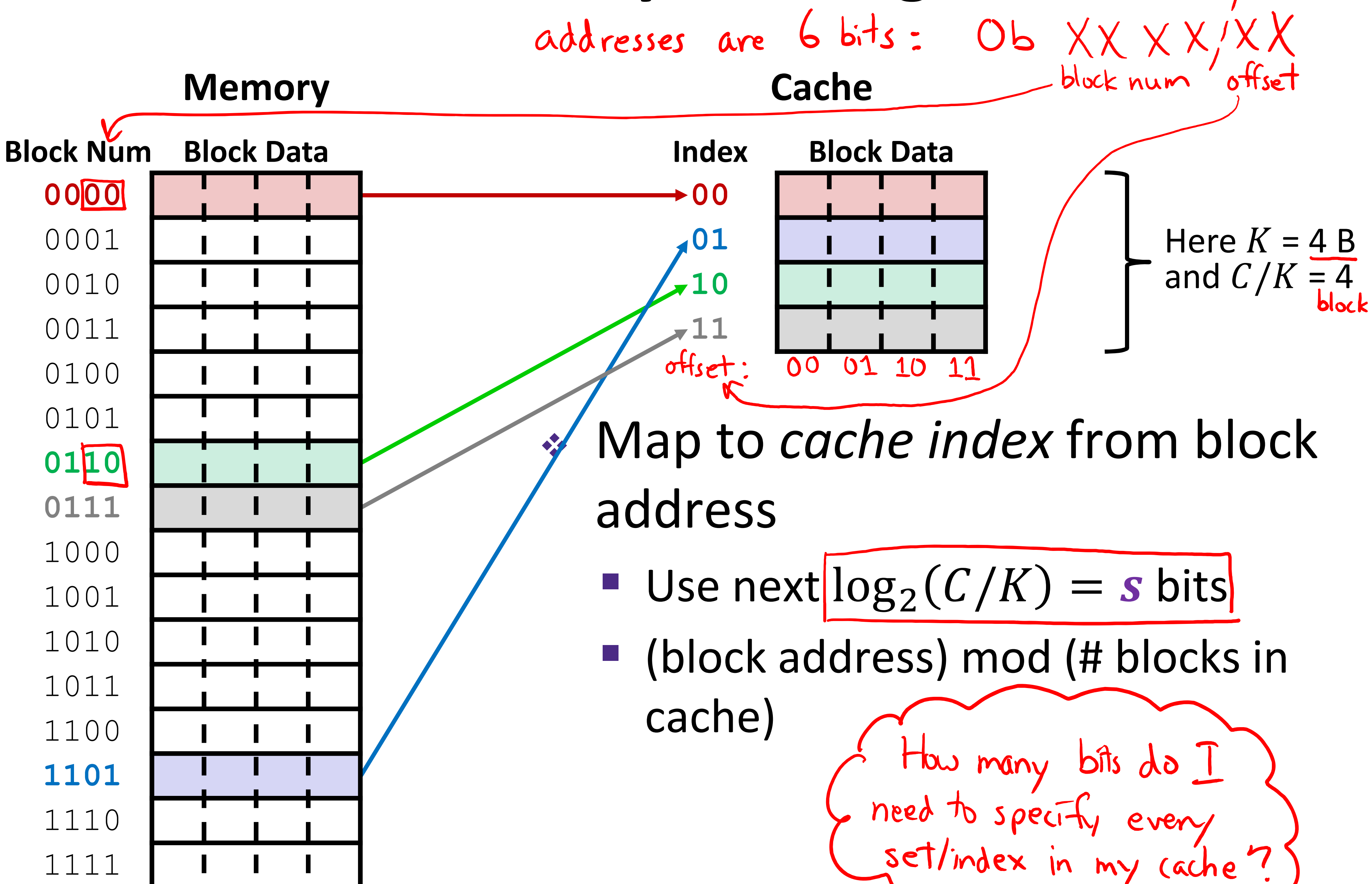
Review: Hash Tables for Fast Lookup



Apply hash function to map data to "buckets"

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

Place Data in Cache by Hashing Address

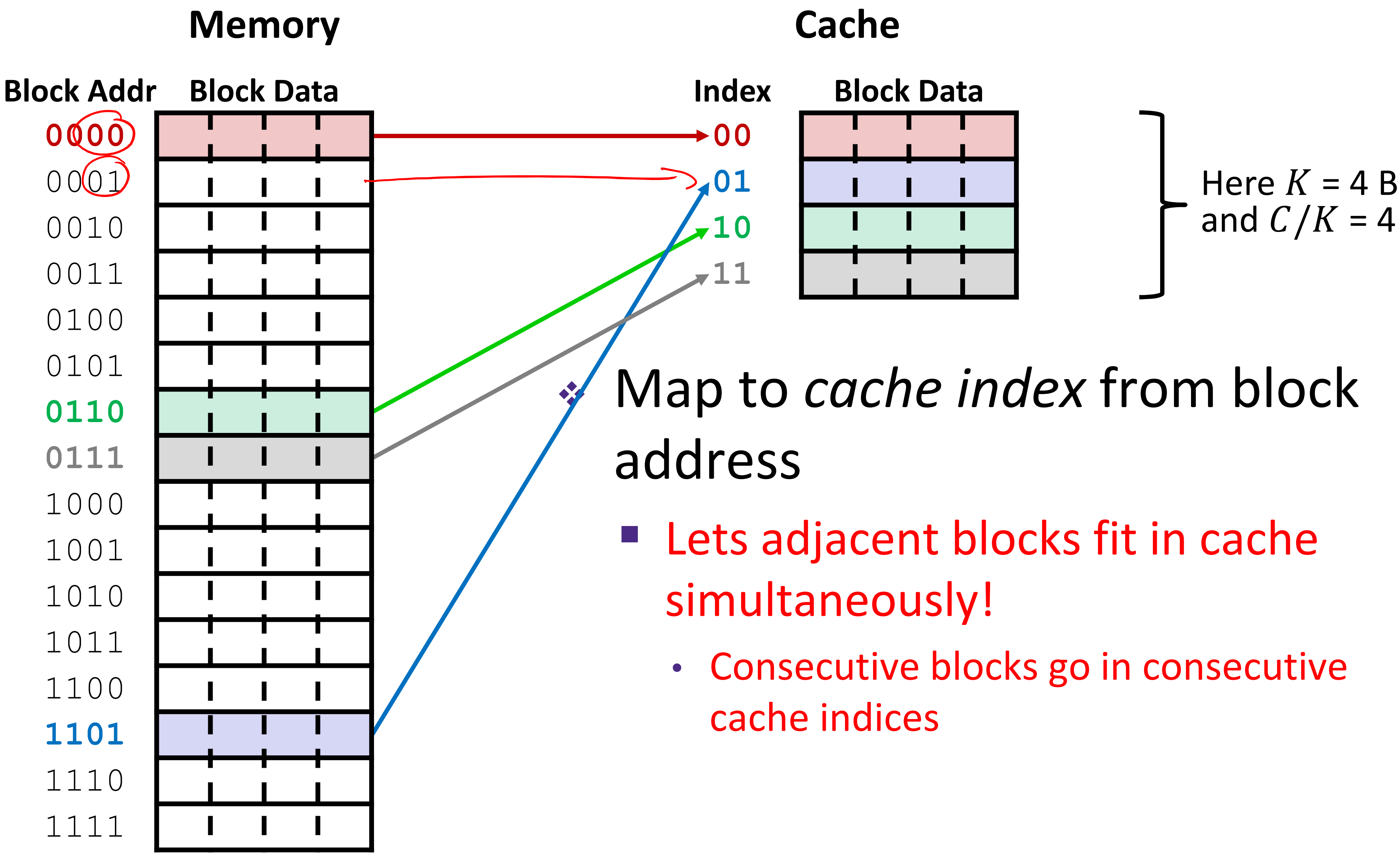


Map to *cache index* from block address

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

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

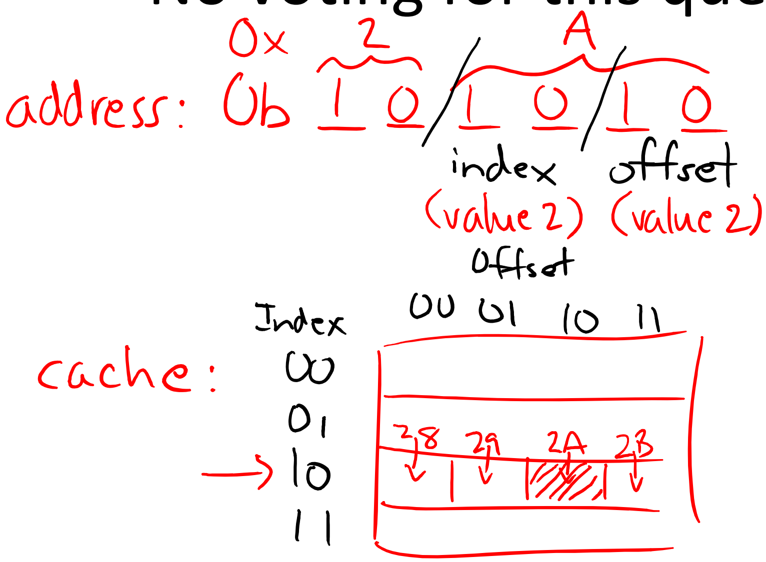
Place Data in Cache by Hashing Address



- Map to *cache index* from block address
  - Lets adjacent blocks fit in cache simultaneously!
    - Consecutive blocks go in consecutive cache indices

Practice Question

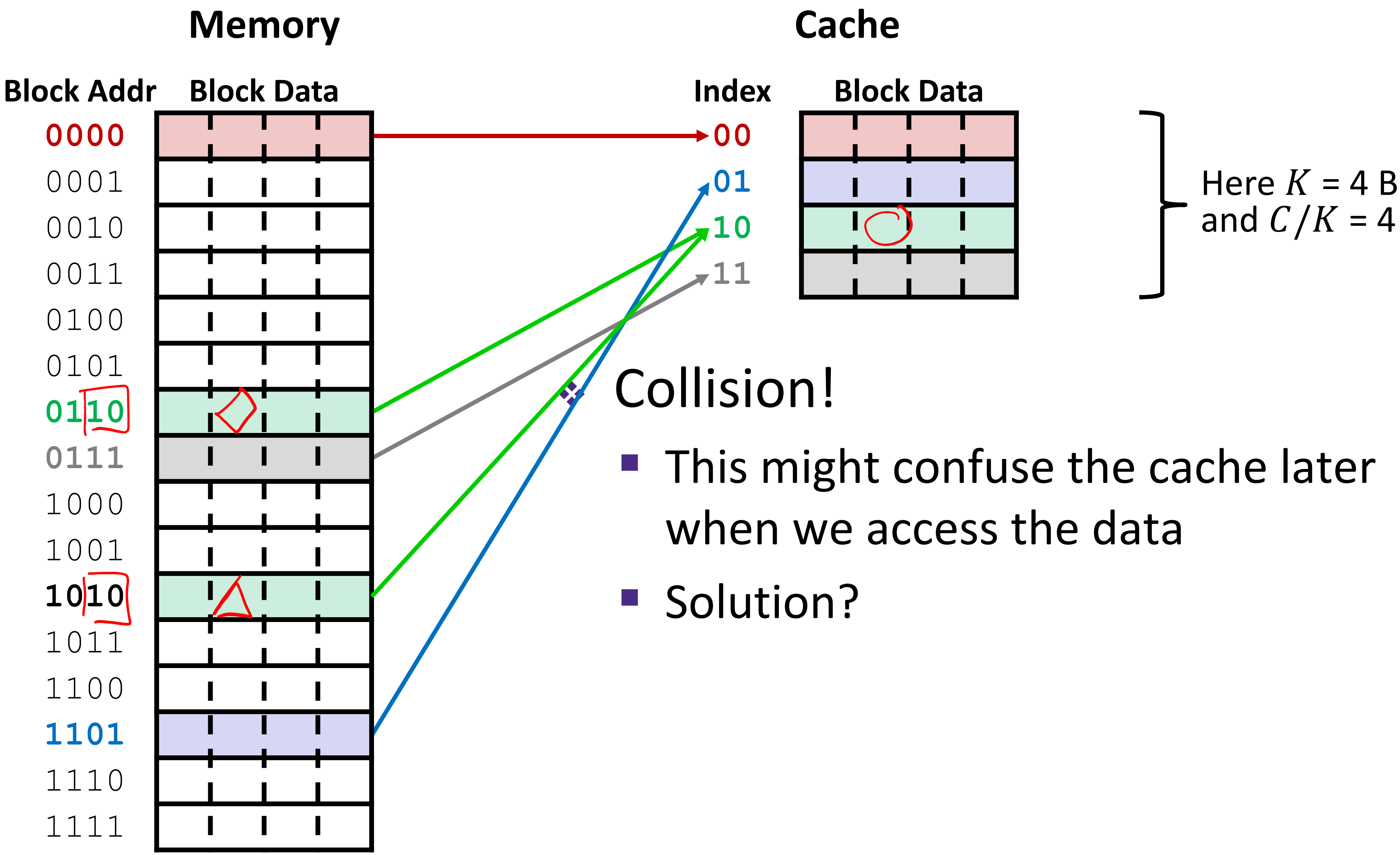
- 6-bit addresses, block size K = 4 B, and our cache holds S = 4 blocks.
- A request for address **0x2A** results in a cache miss. Which index does this block get loaded into and which 3 other addresses are loaded along with it?
  - No voting for this question



Index 2 (0b10)

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

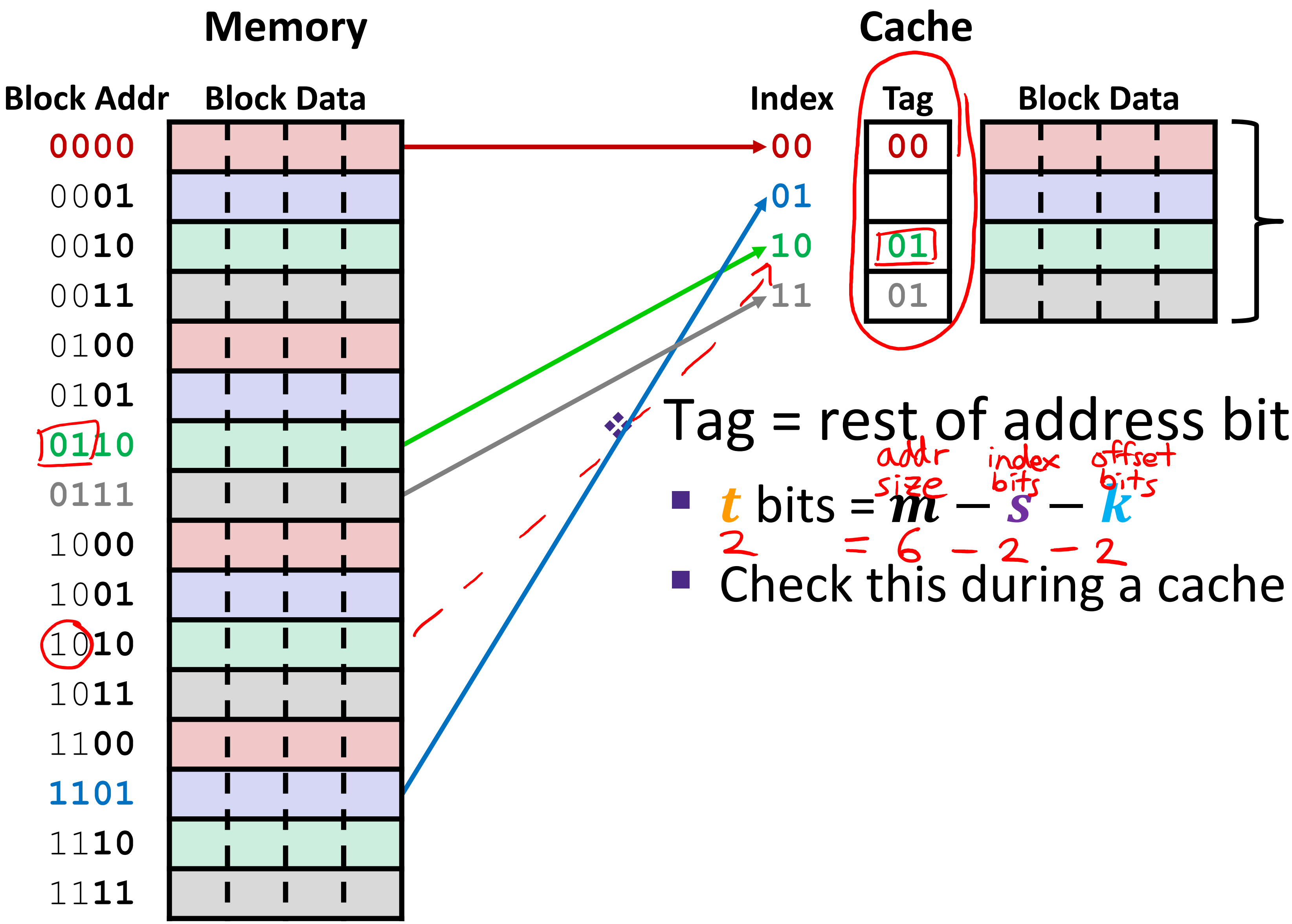
Place Data in Cache by Hashing Address



Collision!

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

Tags Differentiate Blocks in Same Index



Here K = 4 B and C/K = 4

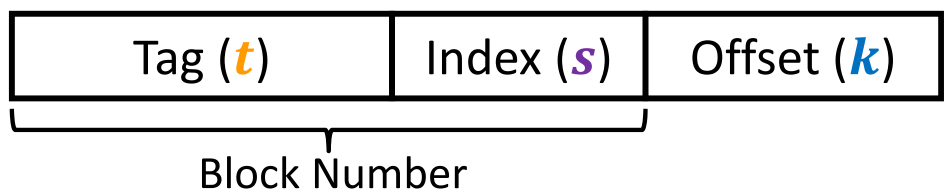
- Tag = rest of address bits
  - t bits = $m - s - k$
  - Check this during a cache lookup

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 ≠ his or her phone number

- TIO address breakdown:

m-bit address:



  - **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

Cache Puzzle

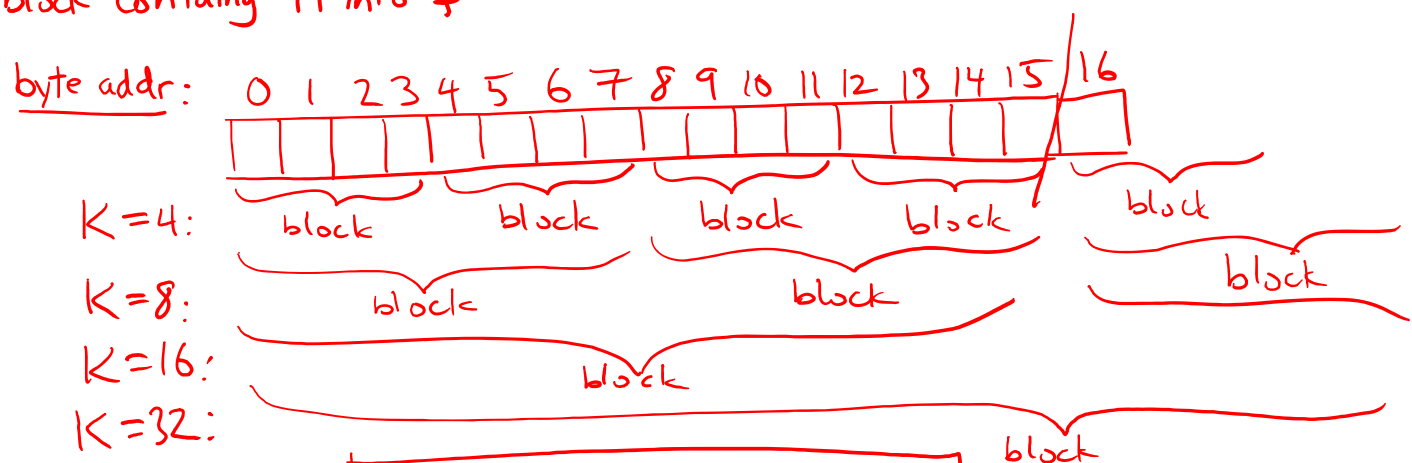
Vote at http://PollEv.com/justinh

- 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

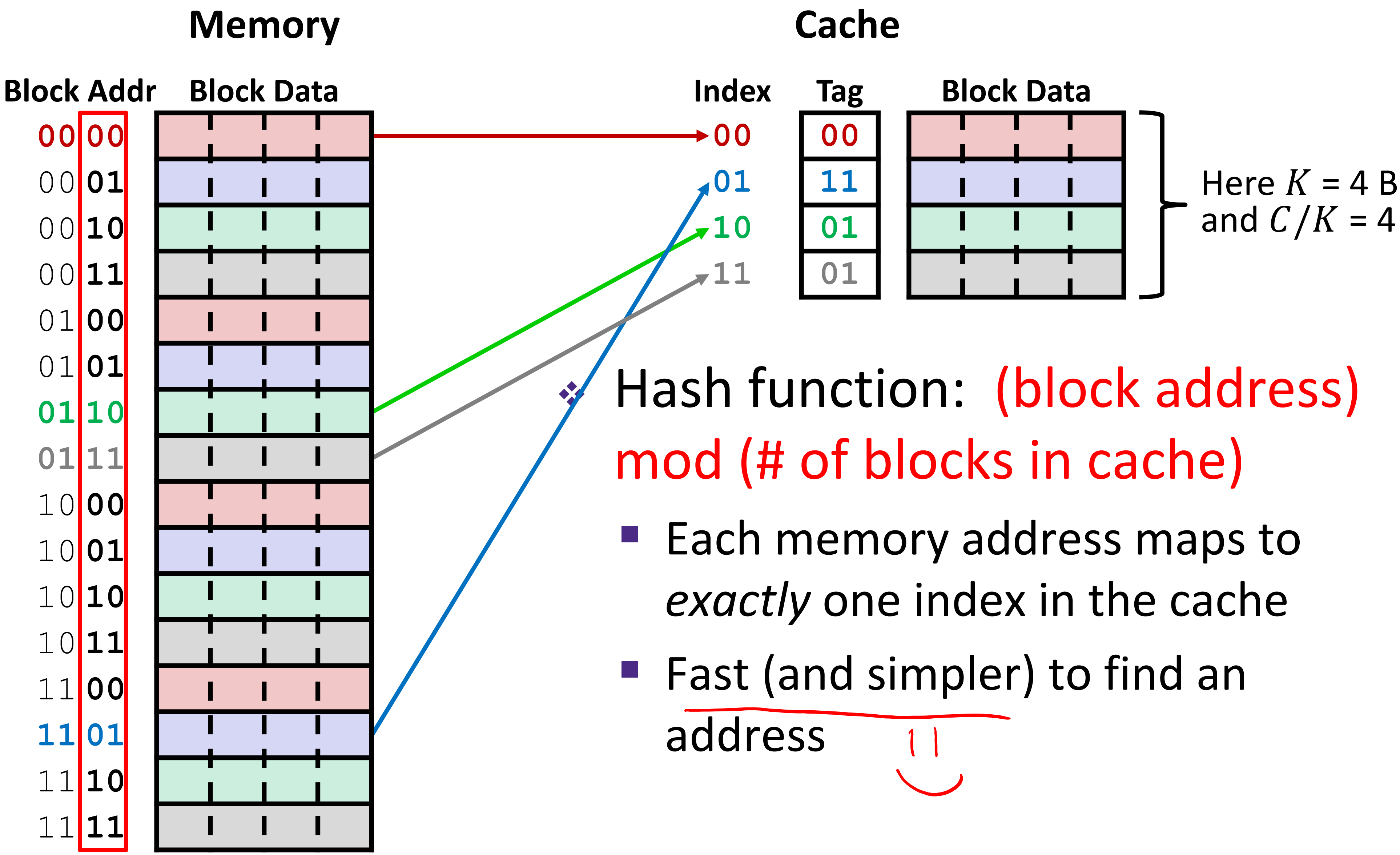
(1) pulls block containg 14 into \$
(2) 14 & 15 are in the same block
(3) 16 is in a different block

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



$K_{min} = 2$ B, $K_{max} = 16$ B

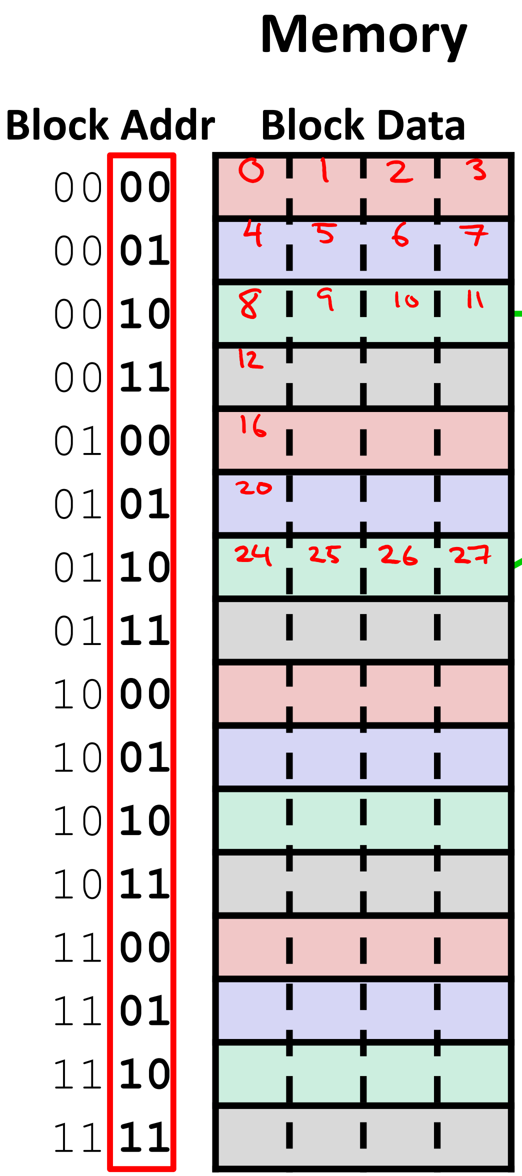
Direct-Mapped Cache

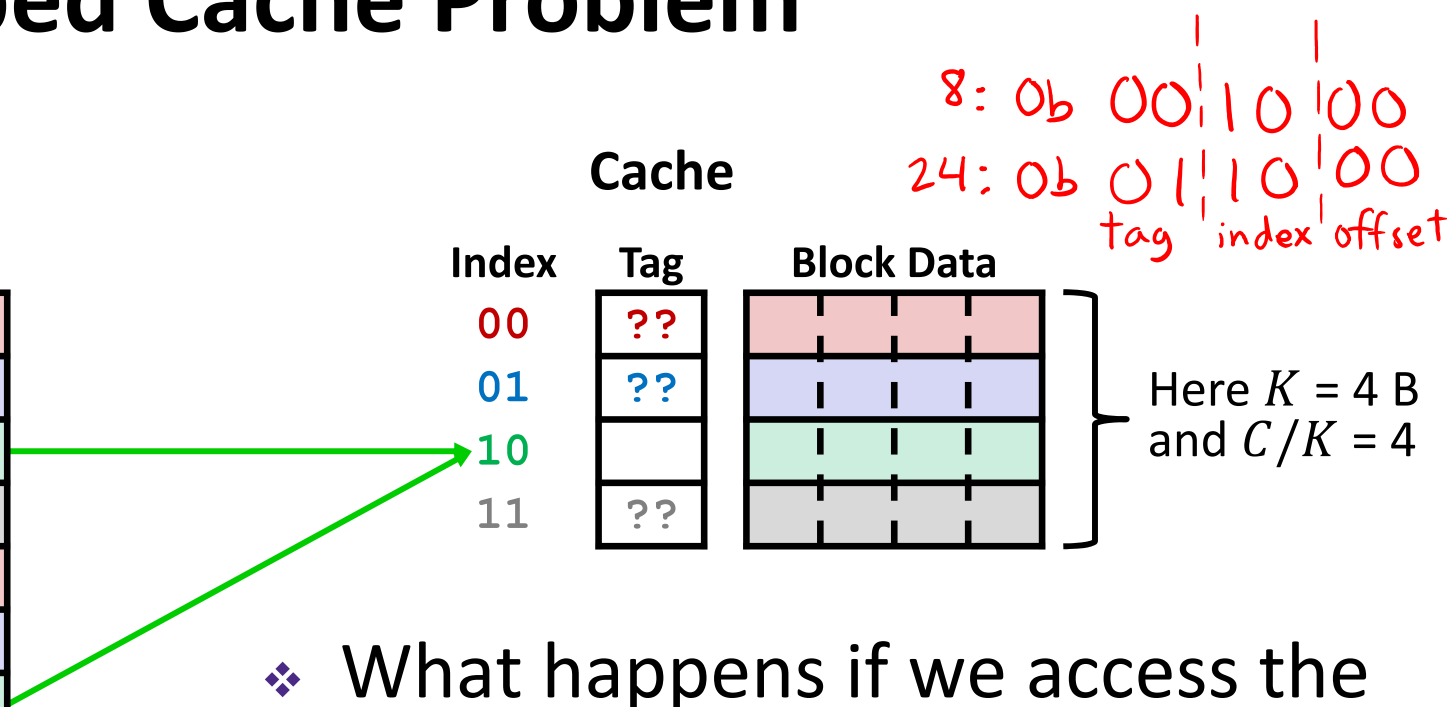


Hash function: (block address) mod (# of blocks in cache)

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

Direct-Mapped Cache Problem





- What happens if we access the following addresses?
  - 8, 24, 8, 24, 8, …?
    Miss Miss Miss Miss Miss…
  - 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**