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!

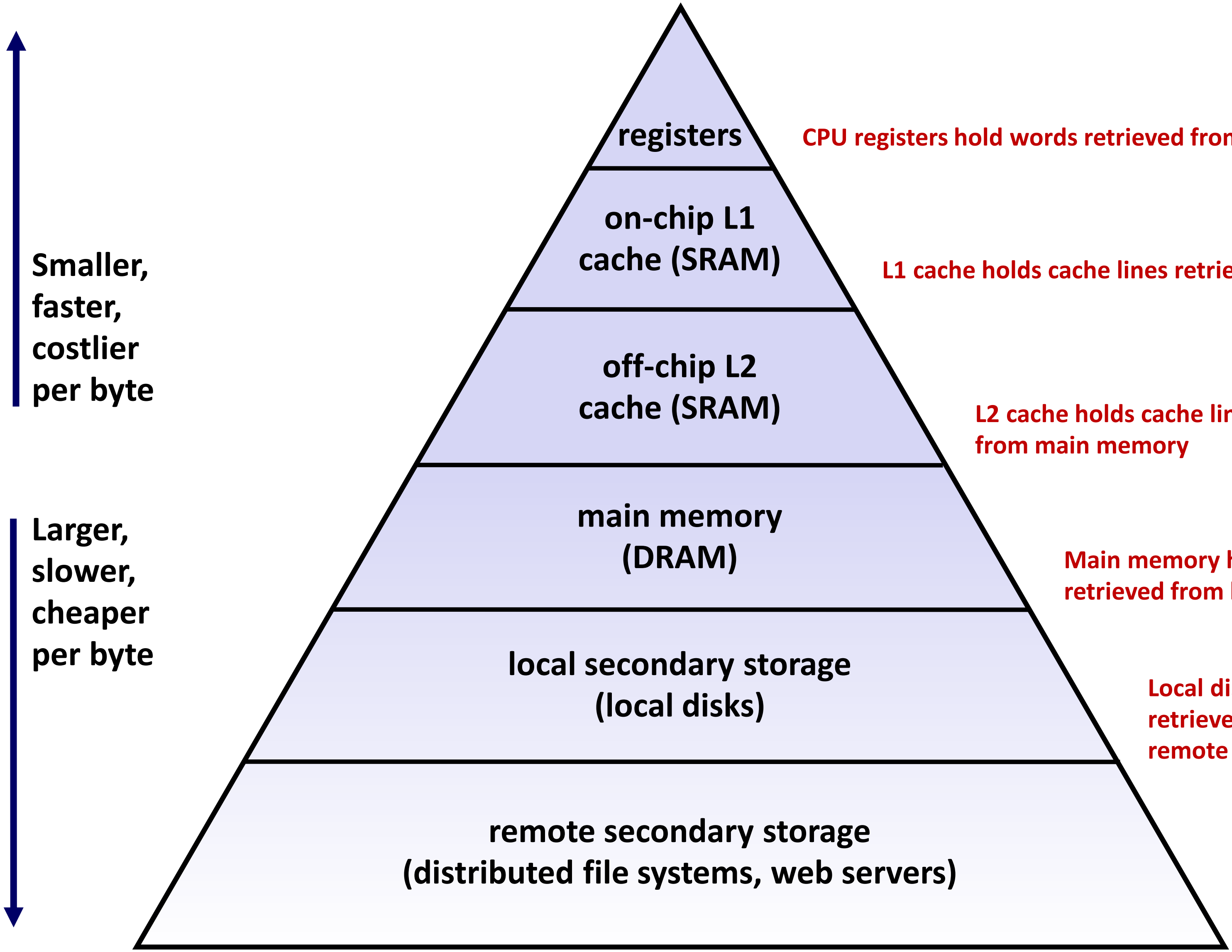
- Unit Summary 2 Due next Wednesday (8/5) – 11:59pm

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



CPU registers hold words retrieved from L1 cache

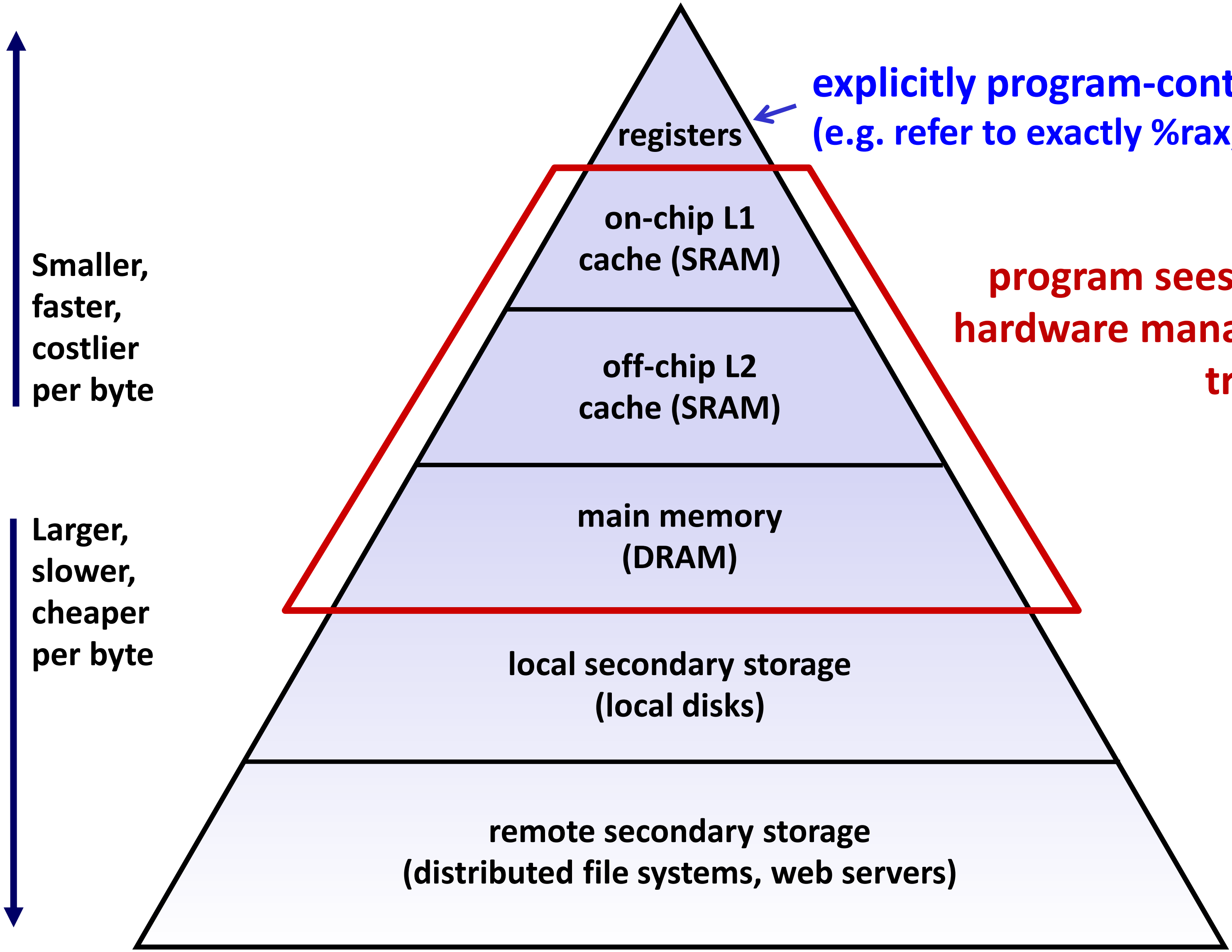
L1 cache holds cache lines retrieved from L2 cache

L2 cache holds cache lines retrieved from main memory

Main memory holds disk blocks retrieved from local disks

Local disks hold files retrieved from disks on remote network servers

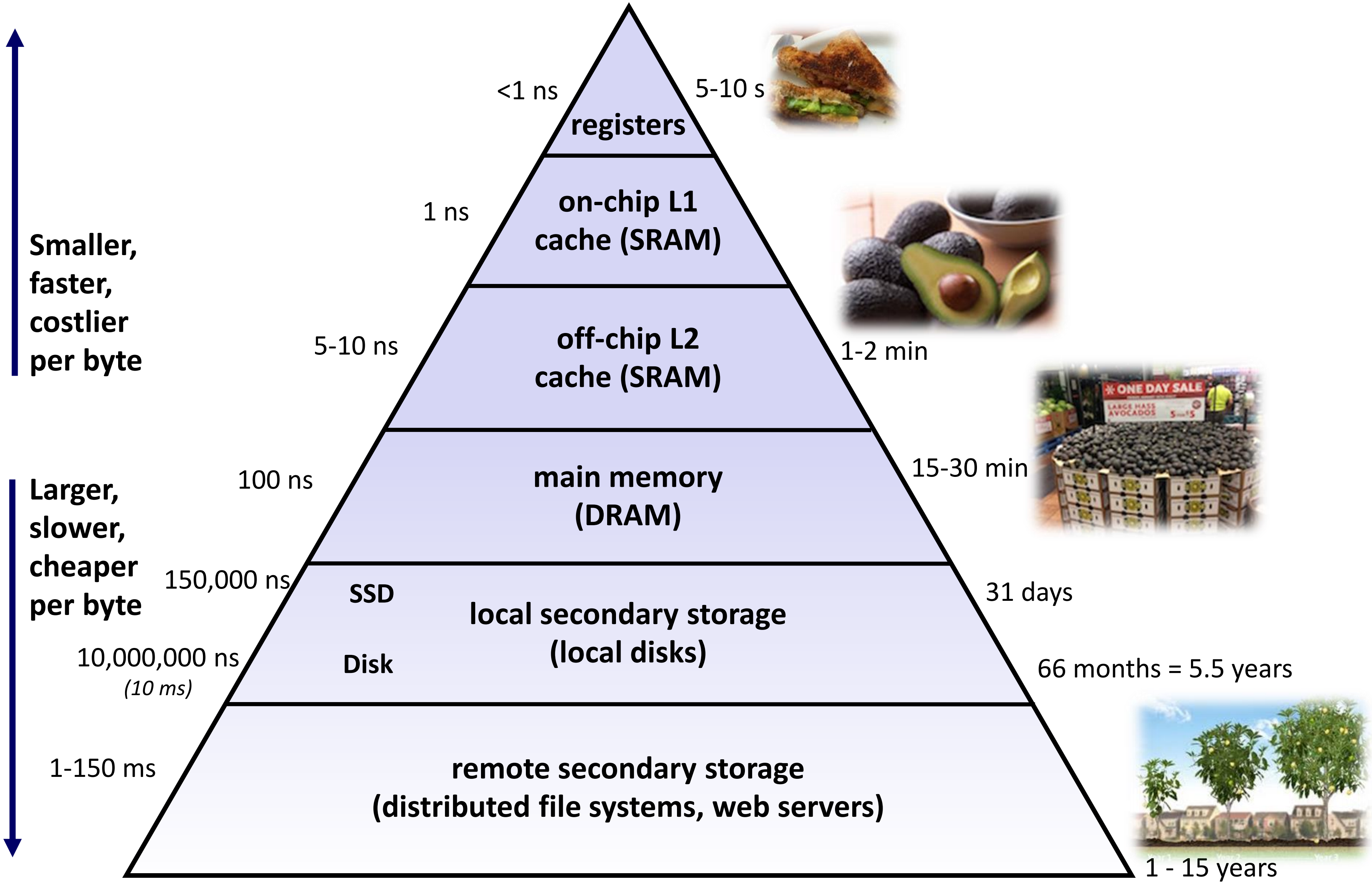
An Example Memory Hierarchy



explicitly program-controlled
(e.g. refer to exactly %rax, %rbx)

program sees "memory";
hardware manages caching
transparently

An Example Memory Hierarchy



Smaller, faster, costlier per byte

Larger, slower, cheaper per byte

<1 ns registers 5-10 s

1 ns on-chip L1 cache (SRAM)

5-10 ns off-chip L2 cache (SRAM) 1-2 min

100 ns main memory (DRAM) 15-30 min

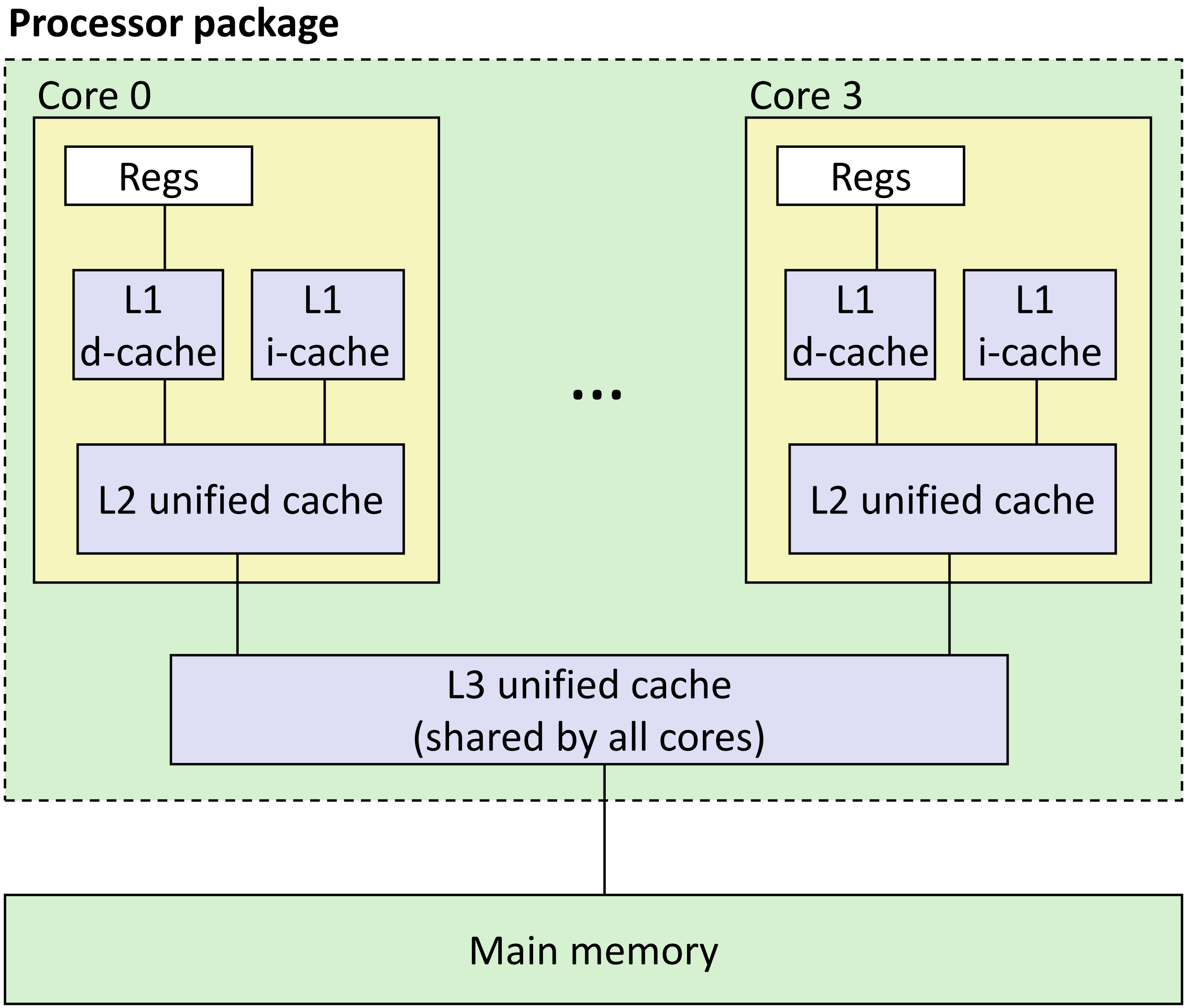
150,000 ns SSD 31 days

local secondary storage (local disks)

10,000,000 ns *(10 ms)* Disk 66 months = 5.5 years

1-150 ms remote secondary storage (distributed file systems, web servers) 1 - 15 years

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

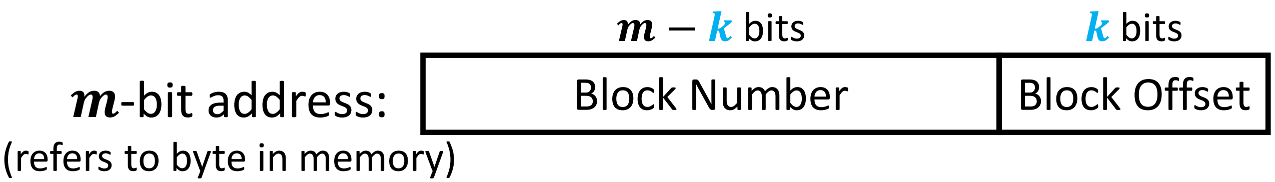
- Block Size (K): unit of transfer between $ and Mem
  - Given in bytes and always a power of 2 (*e.g.* 64 B)
  - Blocks consist of adjacent bytes (differ in address by 1)
    - Spatial locality!

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 $\log_2(K) = \boldsymbol{k}$ 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)



$\boldsymbol{m}-\boldsymbol{k}$ bits

$\boldsymbol{k}$ bits

$\boldsymbol{m}$-bit address:

(refers to byte in memory)

Block Number	Block Offset

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…**	

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
- 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

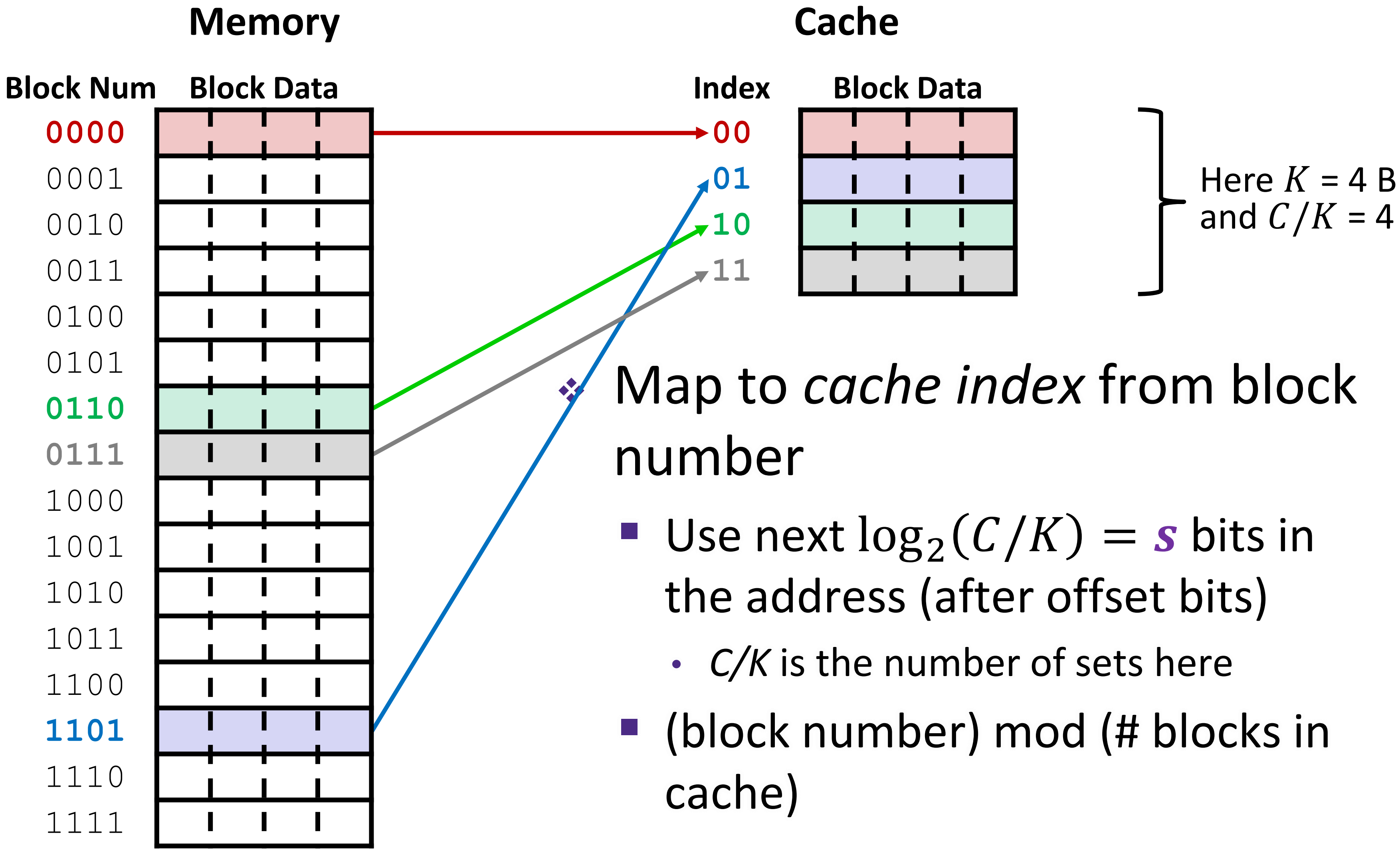
Insert:

5
27
34
102
119

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Apply hash function to map data to “buckets”

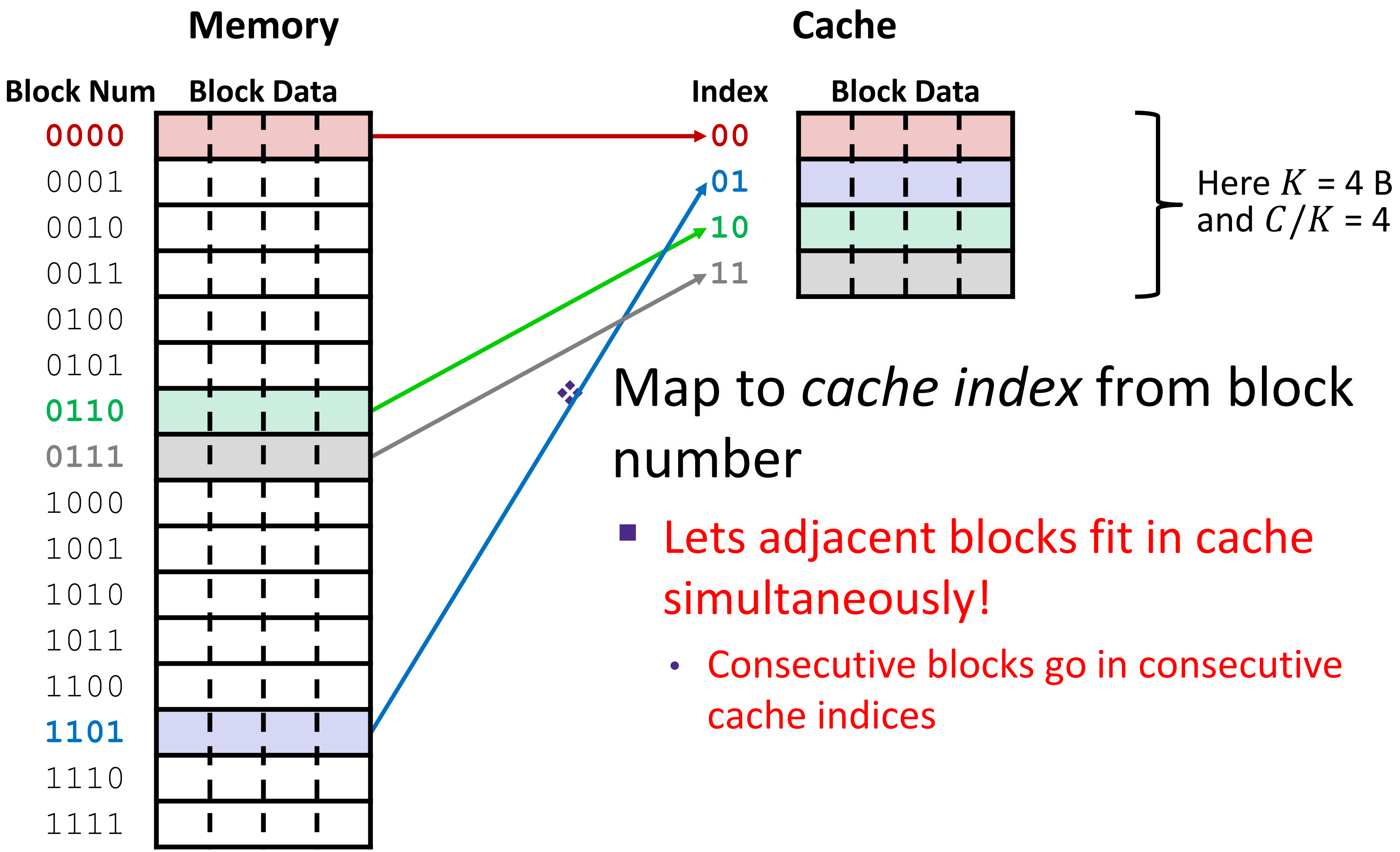
Place Data in Cache by Hashing Address



Here K = 4 B and C/K = 4

- Map to *cache index* from block number
  - Use next $\log_2(C/K) = s$ bits in the address (after offset bits)
    - *C/K* is the number of sets here
  - (block number) mod (# blocks in cache)

Place Data in Cache by Hashing Address



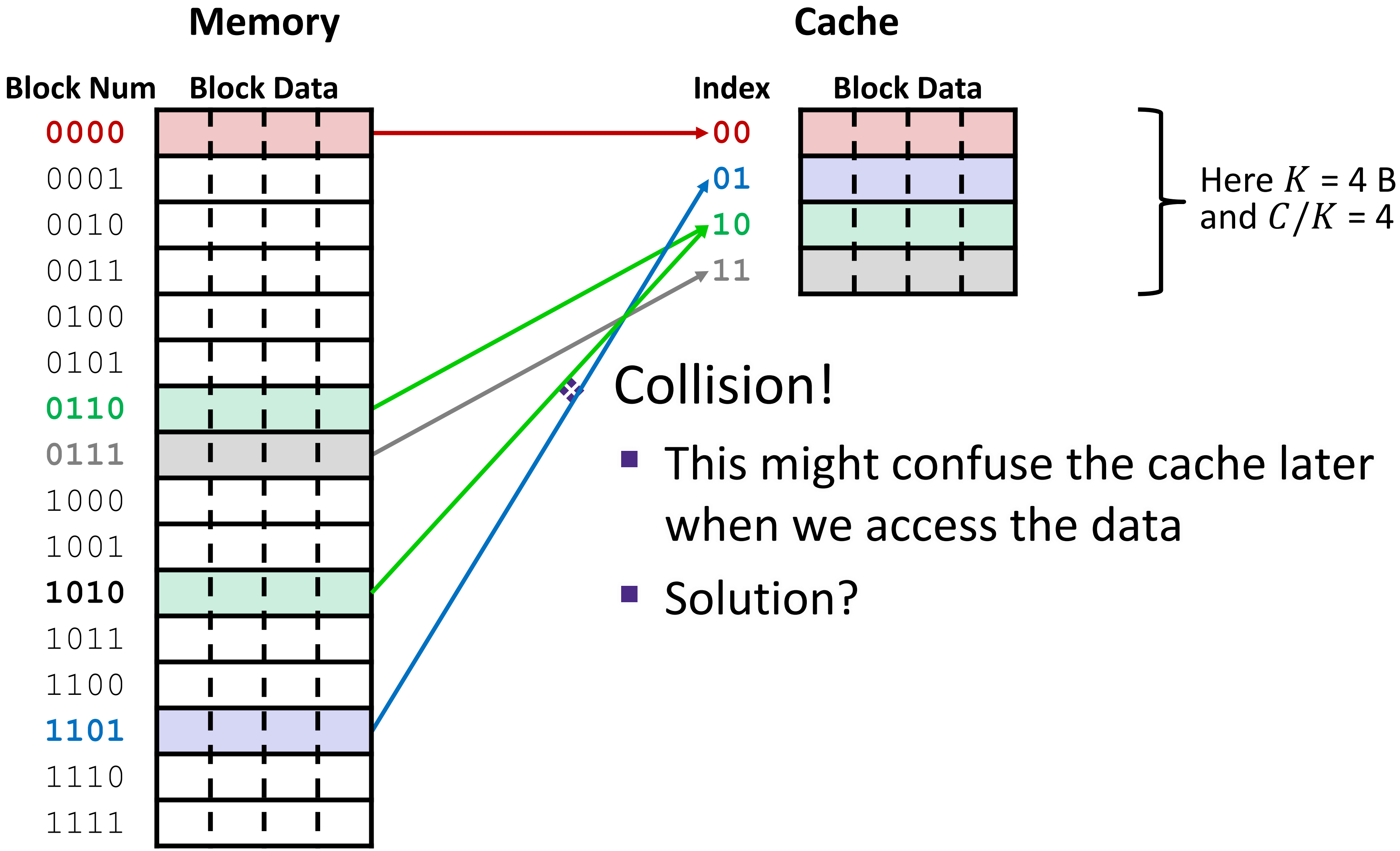
Here K = 4 B and C/K = 4

- Map to *cache index* from block number
  - 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 set index does this block get loaded into and which 3 other addresses are loaded along with it?
  - No voting for this question

Place Data in Cache by Hashing Address

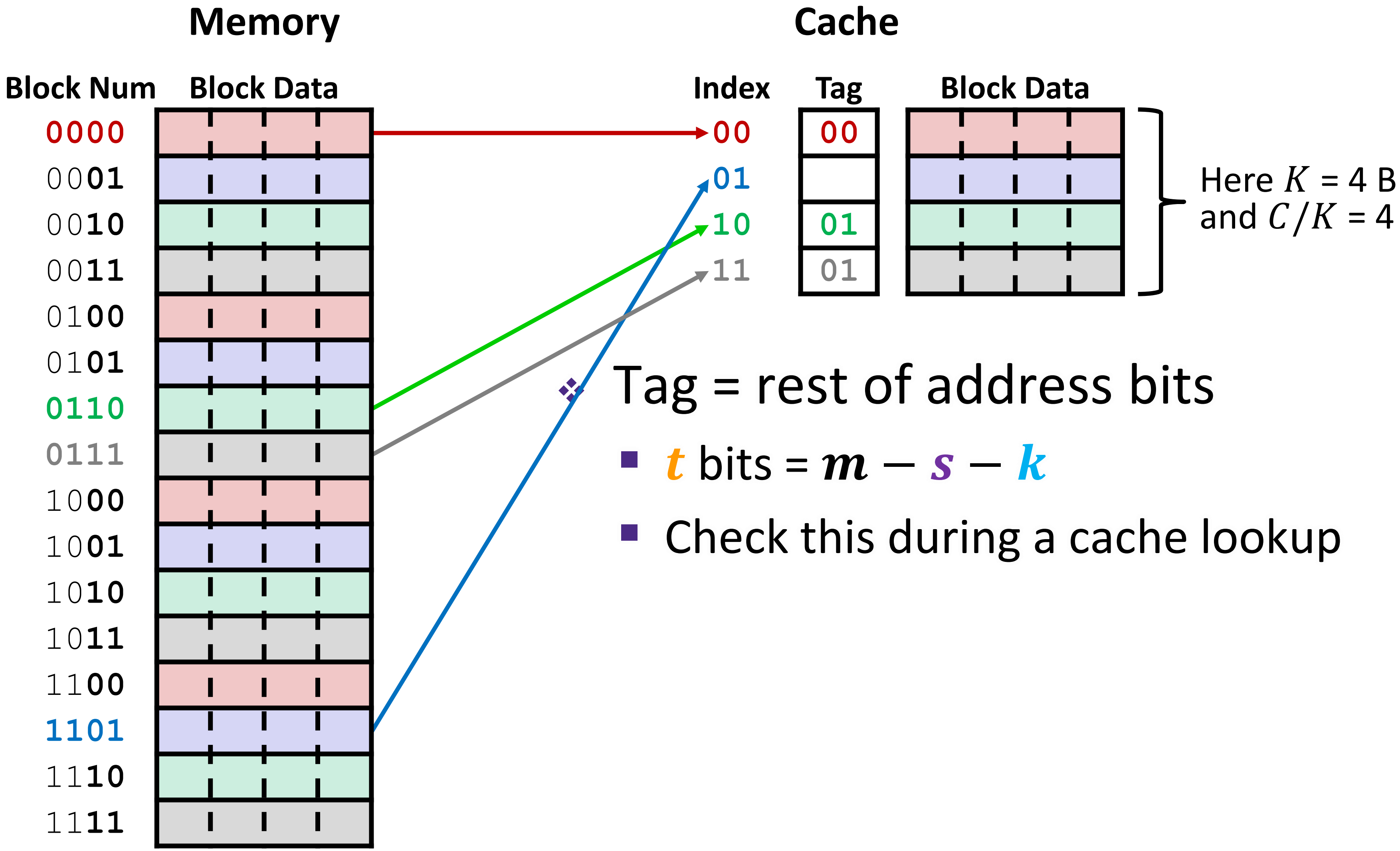


Here K = 4 B and C/K = 4

Collision!

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

Tags Differentiate Blocks in Same Index

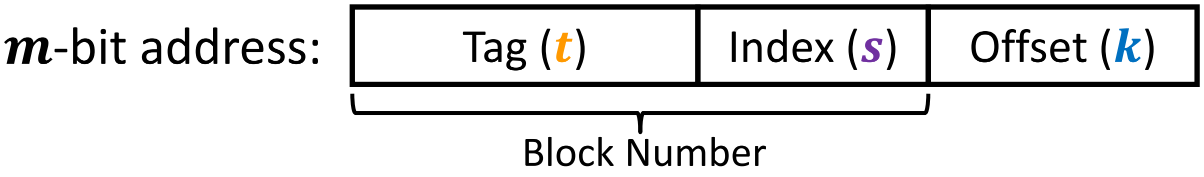


- Tag = rest of address bits
  - t bits = $m - s - k$
  - Check this during a cache lookup

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:

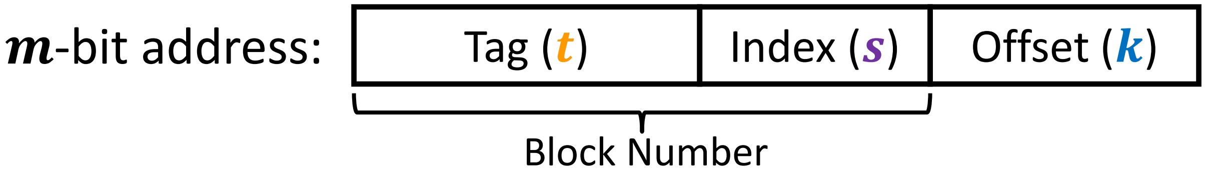


  - **Index** field tells you where to look in cache
  - **Tag** field lets you check that data is the block you want
  - **Offset** field selects specified start byte within block

  - **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) = $\log_2(K)$ =
  - Index bits (s) = $\log_2(num\ sets)$ =
  - Tag bits (t) = Rest of the bits in the address =



- What are the fields for address 0xBA?
  - Tag bits (unique id for block):
  - Index bits (cache set block maps to):
  - Offset bits (byte offset within block):

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)

A. 4 bytes

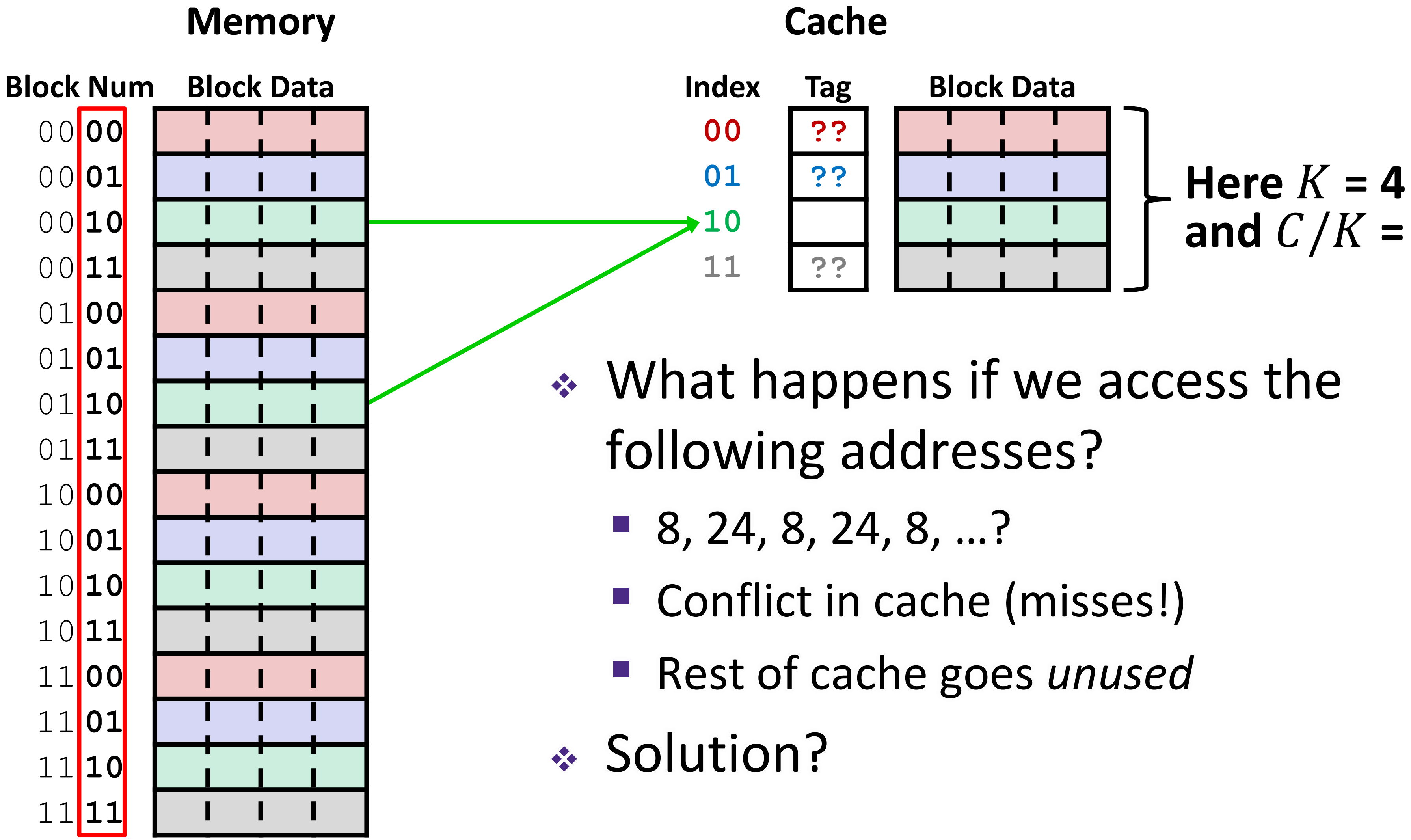
B. 8 bytes

C. 16 bytes

D. 32 bytes

E. We’re lost…

Direct-Mapped Cache Problem

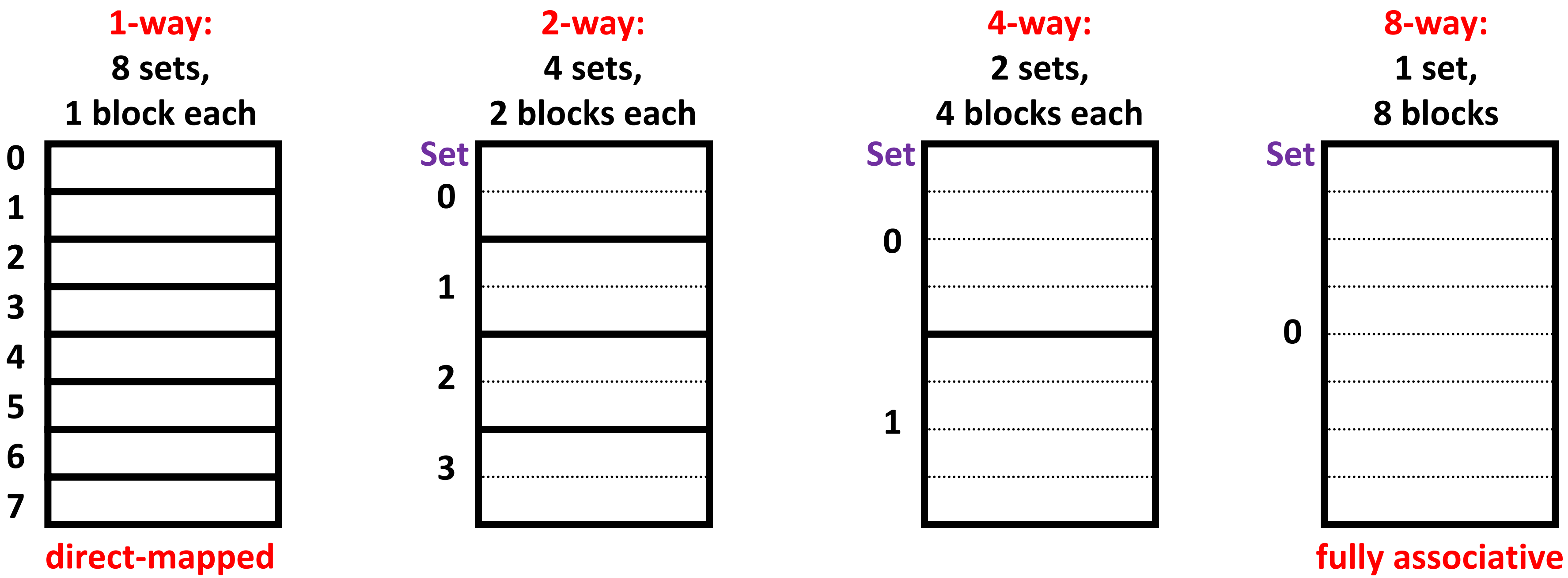


Here K = 4 and C/K =

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

Associativity

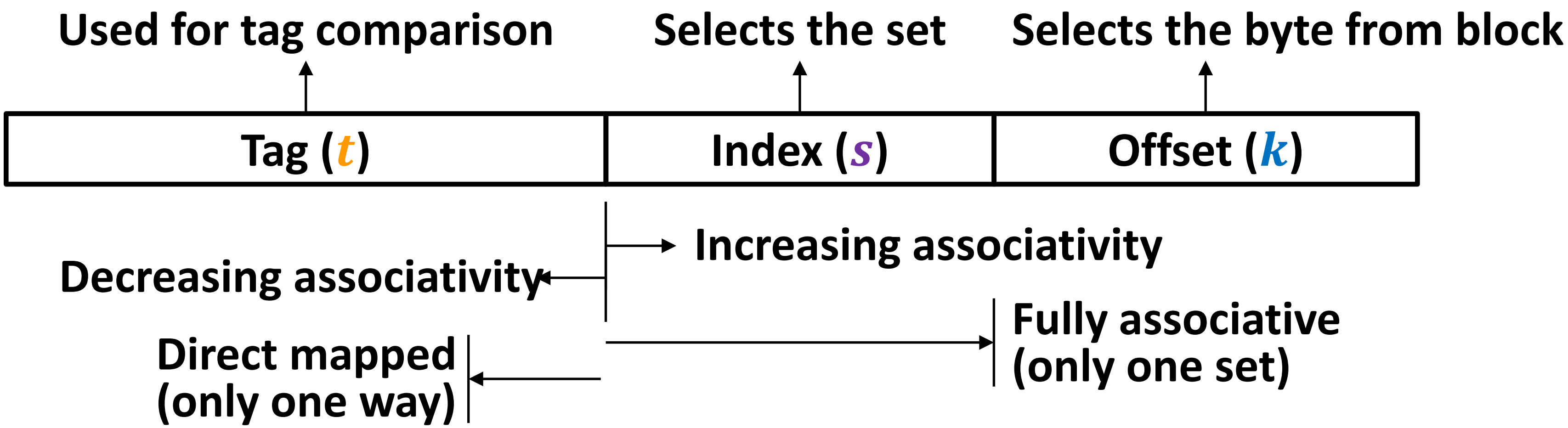
- What if we could store data in any place in the cache?
  - More complicated hardware = more power consumed, slower
- So we *combine* the two ideas:
  - Each address maps to exactly one **set**
  - Each set can store block in more than one **way**



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 $\log_2(C/K/E)$ = $\boldsymbol{s}$ 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 tag comparison

Selects the set

Selects the byte from block

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

Increasing associativity

Decreasing associativity

Fully associative (only one set)

Direct mapped (only one way)

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`

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

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

s = ?
Direct-mapped

Set	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

s = ?
2-way set associative

Set	Tag	Data
0		
1		
2		
3		

s = ?
4-way set associative

Set	Tag	Data
0		
1		