Caches II

CSE 351 Summer 2024

Instructor:
Ellis Haker

Teaching Assistants:
Naama Amiel
Micah Chang
Shananda Dokka
Nikolas McNamee
Jiawei Huang

Administrivia

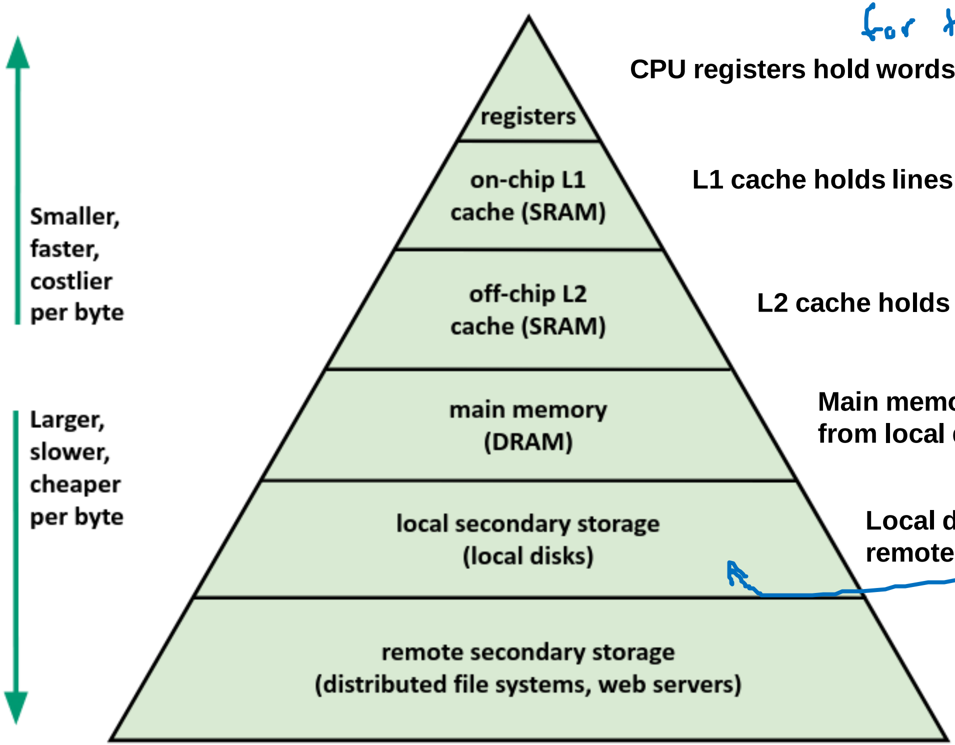
- Labs 2 and 3 extended 🥳 🥳 🥳
  - Regular Lab 2 date was **yesterday (7/21)** (late due date is **tomorrow (7/23)**)
  - Lab 3 will be due **Sunday 7/28** (late due date **Tuesday 7/30**)
- Today:
  - HW13 Due (11:59pm)
  - HW15 and 16 released
    - Combined, due Monday 7/29
  - **Quiz 2 Released (11:59pm)**
- Wednesday 7/24
  - RD16 Due (1pm)
  - HW14 Due (11:59pm)

Mid-Quarter Updates

- Extra office hour on Zoom (link on the course calendar)
  - Wednesdays 6pm-7pm
  - Ellis this week, Shananda after that
- Additional resources
  - Videos on the course website Topic Videos page
  - Optional textbook - *Computer Systems: a Programmer's Perspective*
    - Readings on the course website Schedule
    - Copies in the Allen School study center and my office
- I'm trying to slow down in lecture
  - Please stop me if I'm going too fast :)

Memory Hierarchy Review

each layer is like a "cache" for the one below it



Smaller, faster, costlier per byte

Larger, slower, cheaper per byte

registers

on-chip L1 cache (SRAM)

off-chip L2 cache (SRAM)

main memory (DRAM)

local secondary storage (local disks)

remote secondary storage (distributed file systems, web servers)

CPU registers hold words retrieved from L1 cache

L1 cache holds lines (i.e. blocks) retrieved from L2 cache

L2 cache holds lines retrieved from main memory

Main memory holds disk blocks retrieved from local disks

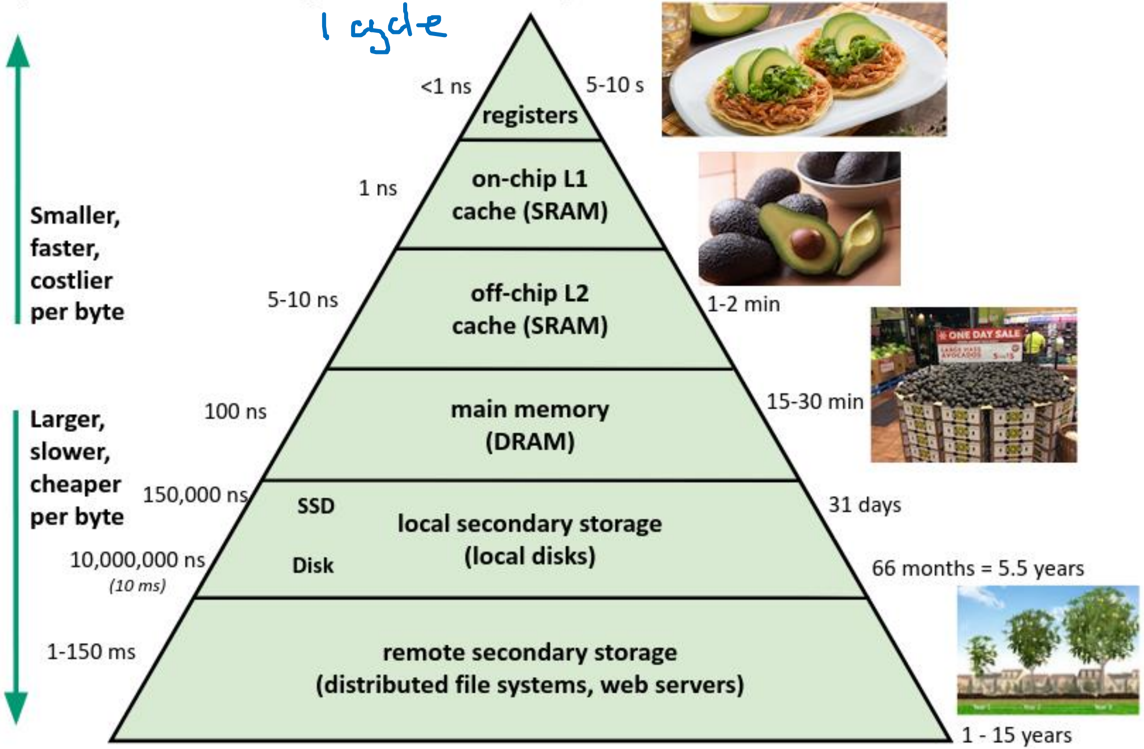
Local disks hold files retrieved from disks on remote network servers

usually for files, but also used for program data when RAM runs out

Memory Hierarchy Review (pt 2)

could do 100+ other instructions in the time it takes to do one mem access!

1 cycle



Smaller, faster, costlier per byte

Larger, slower, cheaper per byte

<1 ns — registers — 5-10 s

1 ns — on-chip L1 cache (SRAM) — 1-2 min

5-10 ns — off-chip L2 cache (SRAM)

100 ns — main memory (DRAM) — 15-30 min

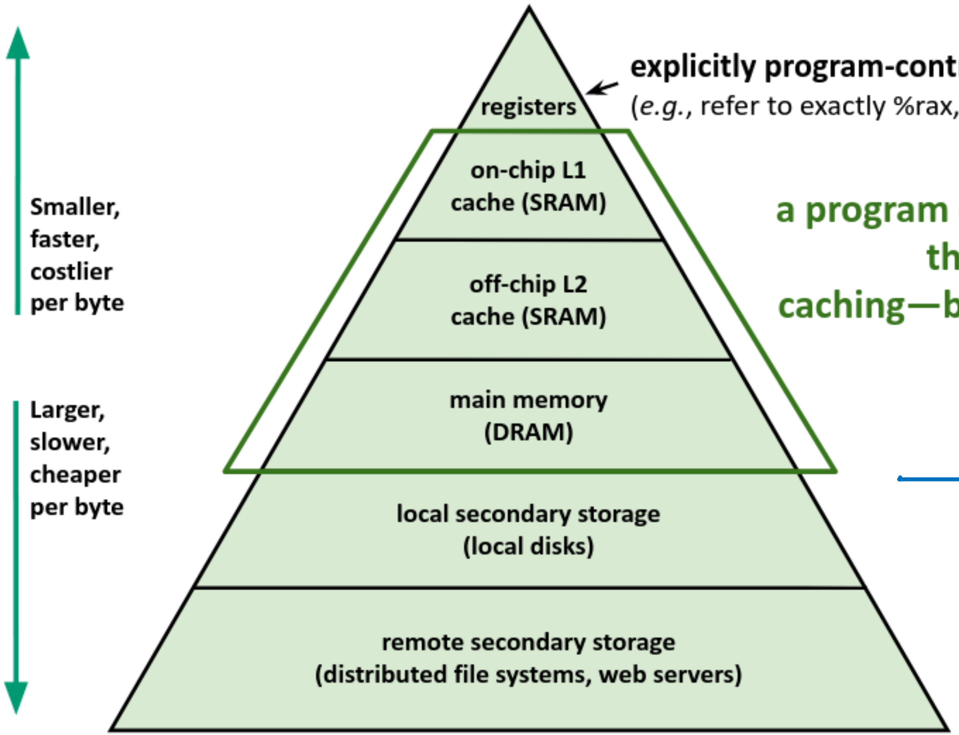
150,000 ns — SSD — 31 days

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

local secondary storage (local disks)

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

Memory Hierarchy Review (pt 3)



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

program requests data at an address, HW checks caches 1st, then memory

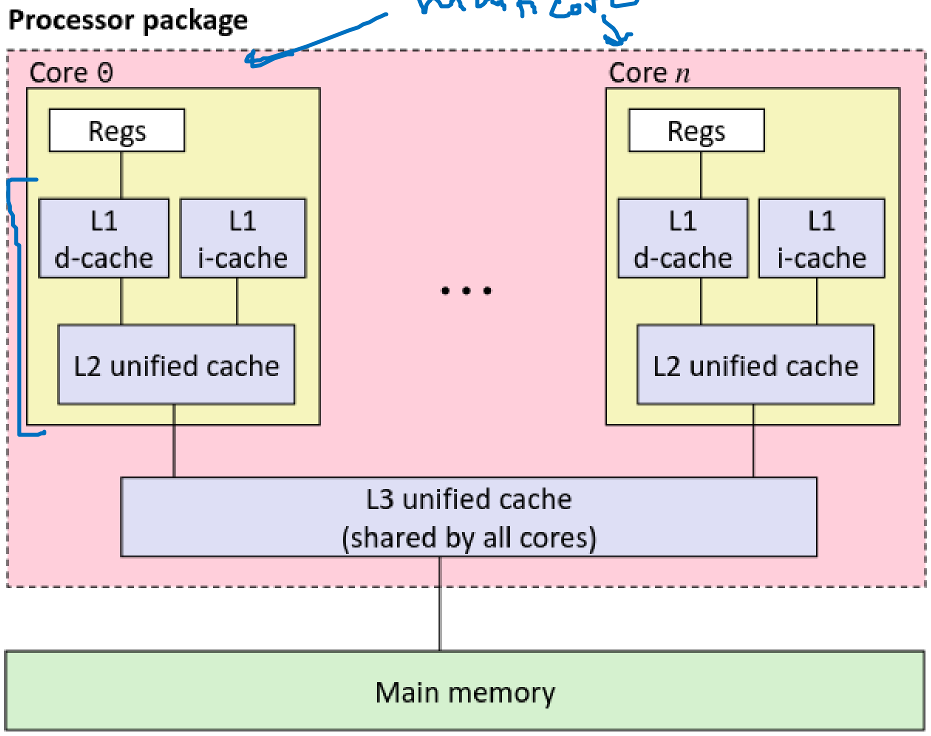
a program only sees "memory"; the hardware manages caching—but always good to be cache aware!

controlled by OS

take 333, 451

Example Microarchitecture: Intel Core i7

all blocks the same in this example, but don't have to be!



multicore

- **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

these numbers are outdated
– trade secrets

Caches

- Cache basics
- Principle of locality
- Memory hierarchies

from reading

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

Review Question

We have a **direct-mapped cache** with the following parameters:

- Block size of $\overset{2^3}{8}$ bytes
- Cache size of 4 KiB $= 4 \cdot 2^{10} = 2^{12}$

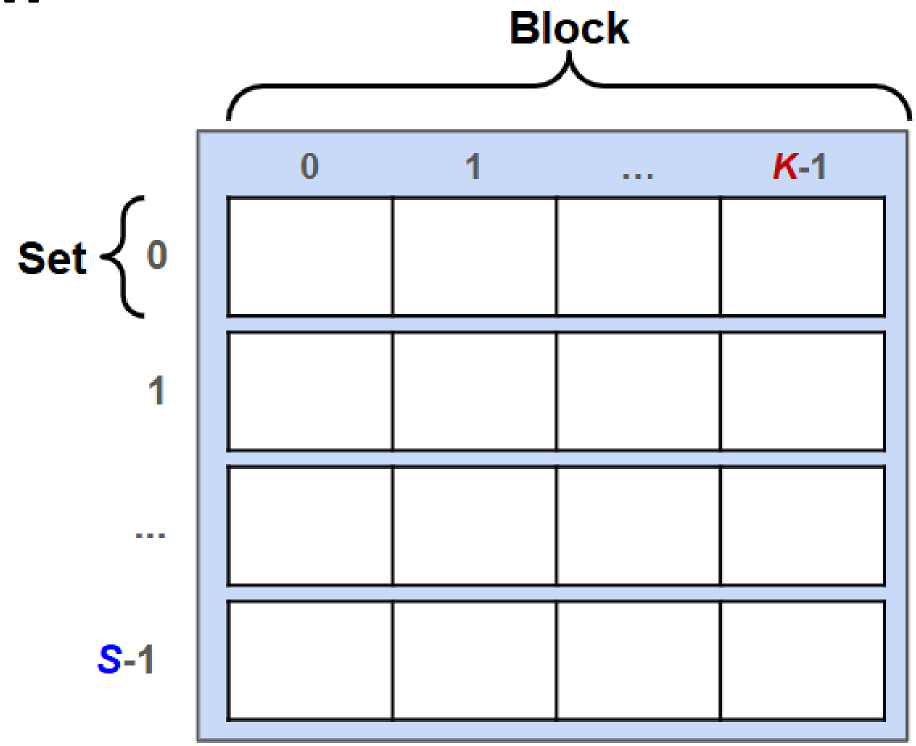
1. How many blocks can the cache hold? 2^{12} B ÷ 2^3 B/block $= 2^9 = \boxed{\text{512 blocks}}$
2. How many bits wide is the block offset field? $\log_2(\text{block size}) = \boxed{\text{3 bits}}$
3. Which of the following addresses (could be multiple) would fall under block 3?
   A) 0x3 = 0b 0~~011~~ → block 0
   [✓ B)] 0x1F = 0b 00011~~111~~ → block 3
   C) 0x30 = 0b 00110~~000~~ → block 6
   D) 0x38 = 0b 00111~~000~~ → block 7

block # = address ÷ blocksize

x ÷ 8 == x >> 3 (in binary)

Reading Terminology Review

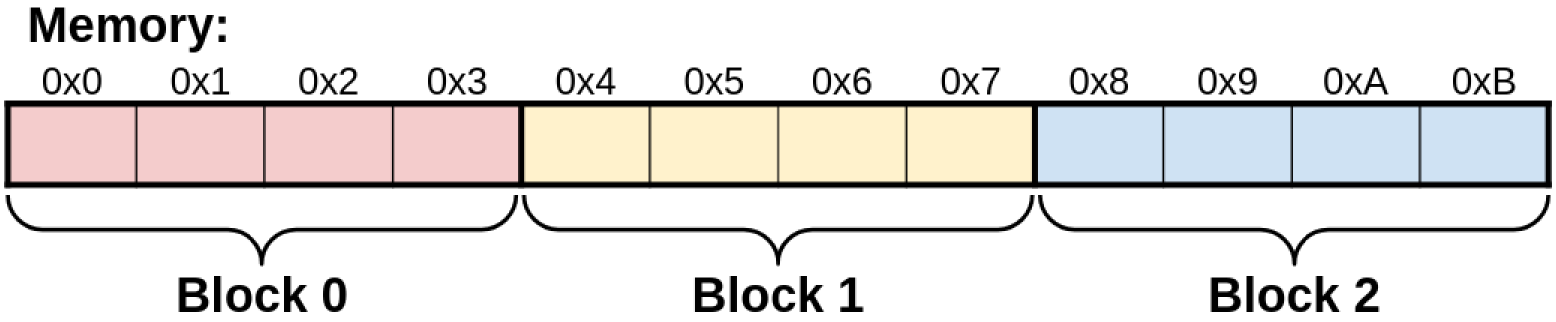
- Cache Parameters
  - Block size (K)
  - Cache size (C bytes, or S sets)
- Address fields
  - Block offset (k bits wide)
  - Block number (also called “block address”)
    - Index field (s bits wide)
    - Tag (t bits wide)



Cache Organization: Block Size

- **Block Size** (***K***): unit of transfer between cache and memory
  - Given in bytes and always a power of 2
  - Blocks are aligned and consist of adjacent bytes
    - Spatial locality!

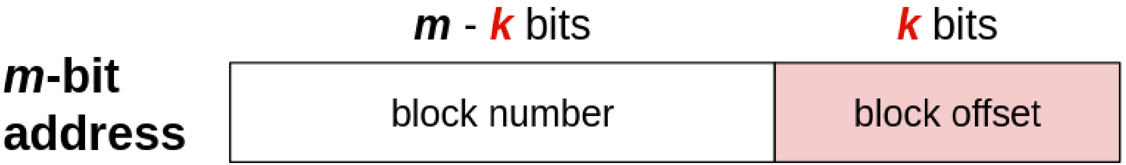
Example: ***K*** = 4B



Cache Organization: Block Size (pt 2)

remember within-same-block from lab1a!

- Given block size ***K***:
  - Address ÷ ***K*** = **block number** (i.e. which block this address belongs to)
  - Address % ***K*** = **block offset** (i.e. where in the block this address is located)
- Define ***k*** = $\log_2(K)$
  - Lowest ***k*** bits of address tell us the block offset



m - ***k*** bits

k bits

***m*-bit address**

block number	block offset

<u>Example</u>: If we have 6-bit addresses and ***K*** = 4B, which block does address 0x15 belong to? What is its offset within that block?

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

$0b\ 0101\boxed{01}$

↑ remove leading 0s to get 6-bit addr

block# = 0b 0101 = 5

offset = 0b 01 = 1

Cache Organization: Cache Size

- **Cache size** (***C***) = how much data the cache can hold
  - Does not include any metadata
  - If size is (***C***) bytes, then the cache can hold ***C***/***K*** blocks
    - Ex: if ***C*** = 32KiB and ***K*** = 64B, then the cache can hold 512 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 data structure provides fast lookup?

Hash table!

Hash Tables for Fast Lookup

- Divide cache into "buckets" (**sets**)
  - Apply hash function to map each block to a set
  - What's a simple hash function we can use?

Example: If we have 10 sets, what indices should each of these blocks go into?

modulo by # of sets

- 5
- 27
- 34
- 102

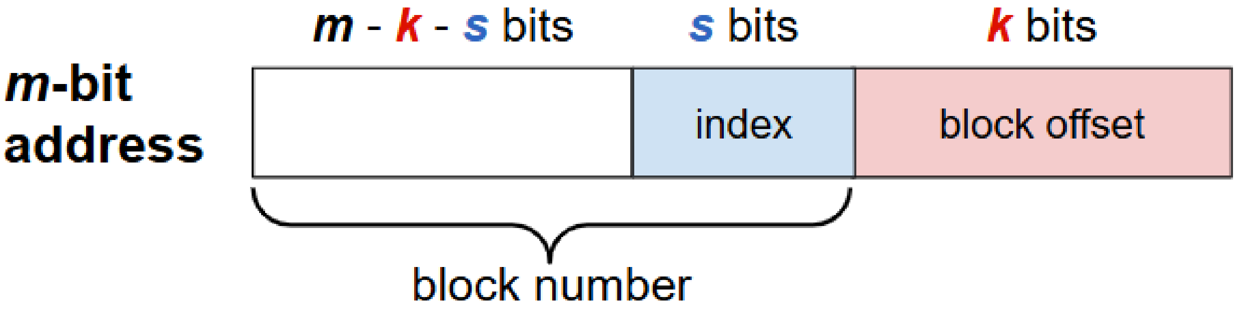
In base 10, % by ten means taking the lowest digit - easy!

In general, for a number in base b, % by b^n means taking the lowest n digits

0	
1	
2	102
3	
4	34
5	5
6	
7	27
8	
9	

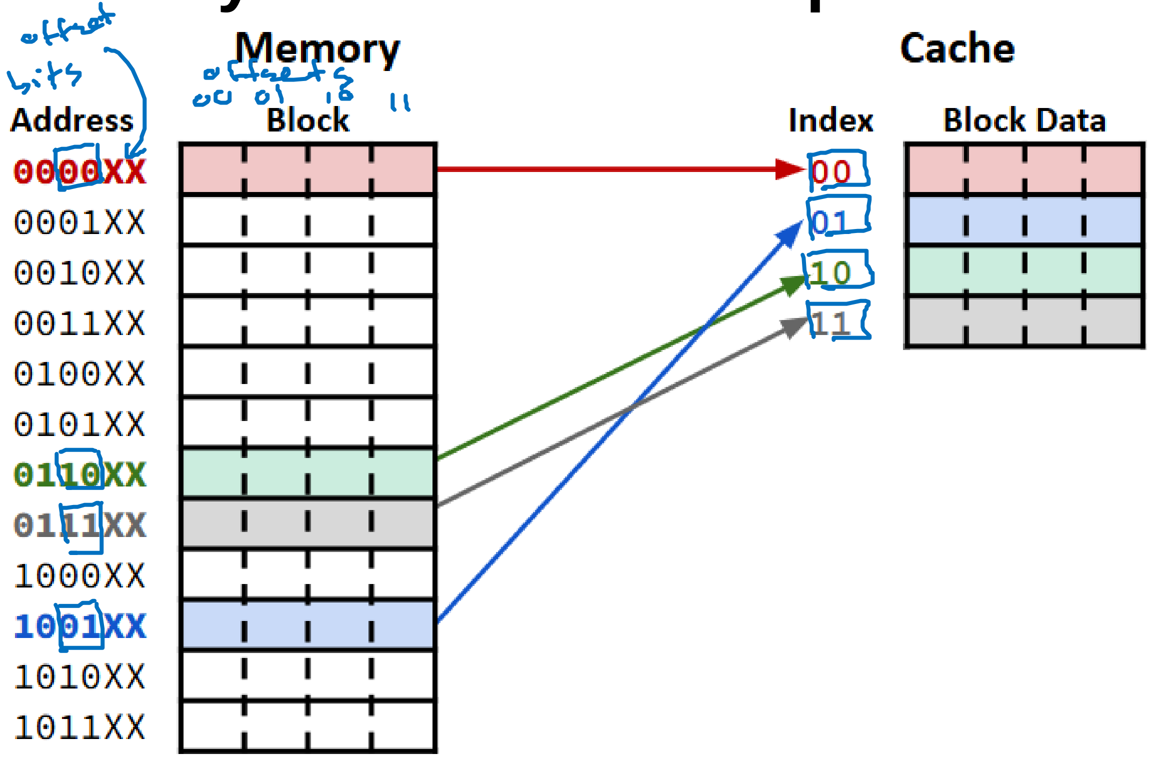
Cache Organization: Sets

- **Number of sets** (***S***) = cache size (***C***) ÷ block size (***K***)
  - Always a power of 2
  - Block number % ***S*** = **set index** (i.e. where in the cache this block goes
- Define ***s*** = $\log_2(S)$
  - Lowest ***s*** bits of the **block number** tell us the index → same as % by 2^s



Memory and Cache Example

In this example:
K = 4B
S = 4



offset bits

Memory

offsets
00 01 10 11

Address

Block

0000XX
0001XX
0010XX
0011XX
0100XX
0101XX
0110XX
0111XX
1000XX
1001XX
1010XX
1011XX

Cache

Index

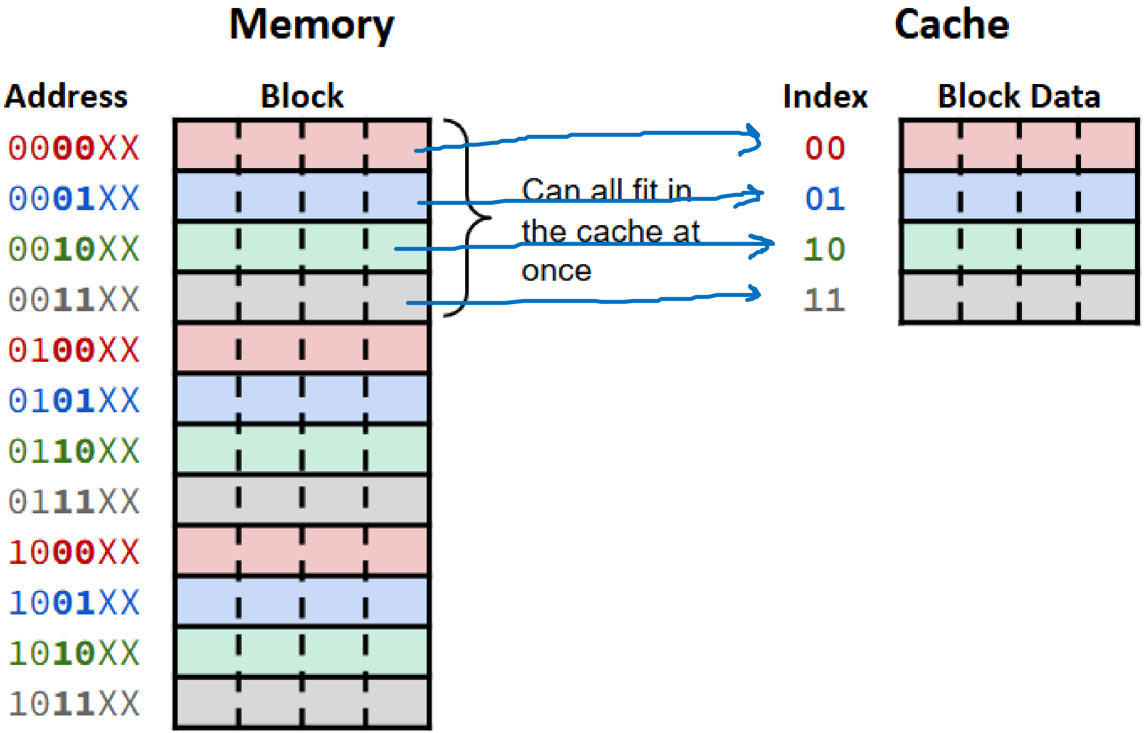
Block Data

00
01
10
11

- Map blocks to cache sets
  - Block# mod ***S*** = index

Memory and Cache Example (pt 2)

In this example:
K = 4B
S = 4



Memory

Cache

Address	Block
0000XX	
0001XX	
0010XX	
0011XX	
0100XX	
0101XX	
0110XX	
0111XX	
1000XX	
1001XX	
1010XX	
1011XX	

Index	Block Data
00	
01	
10	
11	

Can all fit in the cache at once

- Map blocks to cache sets
  - Block# mod S = index
- **Adjacent blocks can fit into the cache at the same time!**
  - Map to consecutive sets

Polling Question

$k = 2$ $s = 2$

- 6-bit addresses, block size ***K*** = 4 B, and our cache holds ***S*** = 4 blocks
- The CPU requests data at address `0x2A`. $= 0b00\underbrace{10\,1010}_{\text{6-bit address}}$
  - Which index can this address be found in?
  - Which 3 other addresses can be found in the same block? (No Ed poll for this one)

$$\overbrace{10\underbrace{10}_{\text{index}}}^{\text{block \#}}\underbrace{10}_{\text{offset}}$$

index $= 0b10 = \boxed{2}$

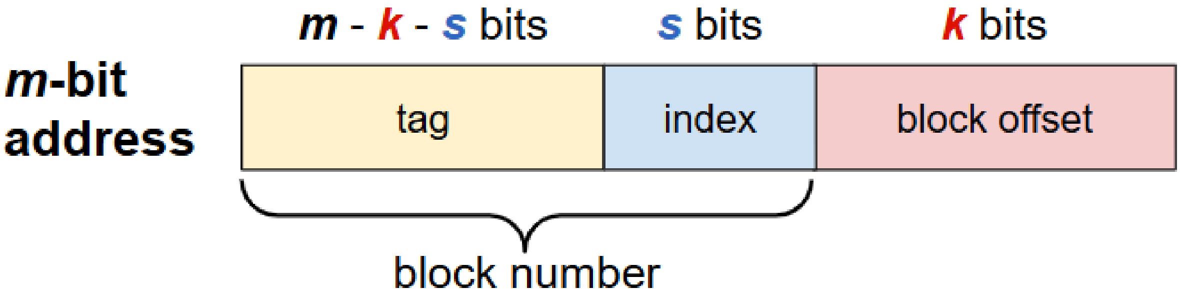
other addresses must have same block #

along w/ 0x2A, form a contiguous 4B block

$$\left\{\begin{array}{l} 0b1010\,00 = 0x28 \\ 0b1010\,01 = 0x29 \\ 0b1010\,11 = 0x2B \end{array}\right.$$

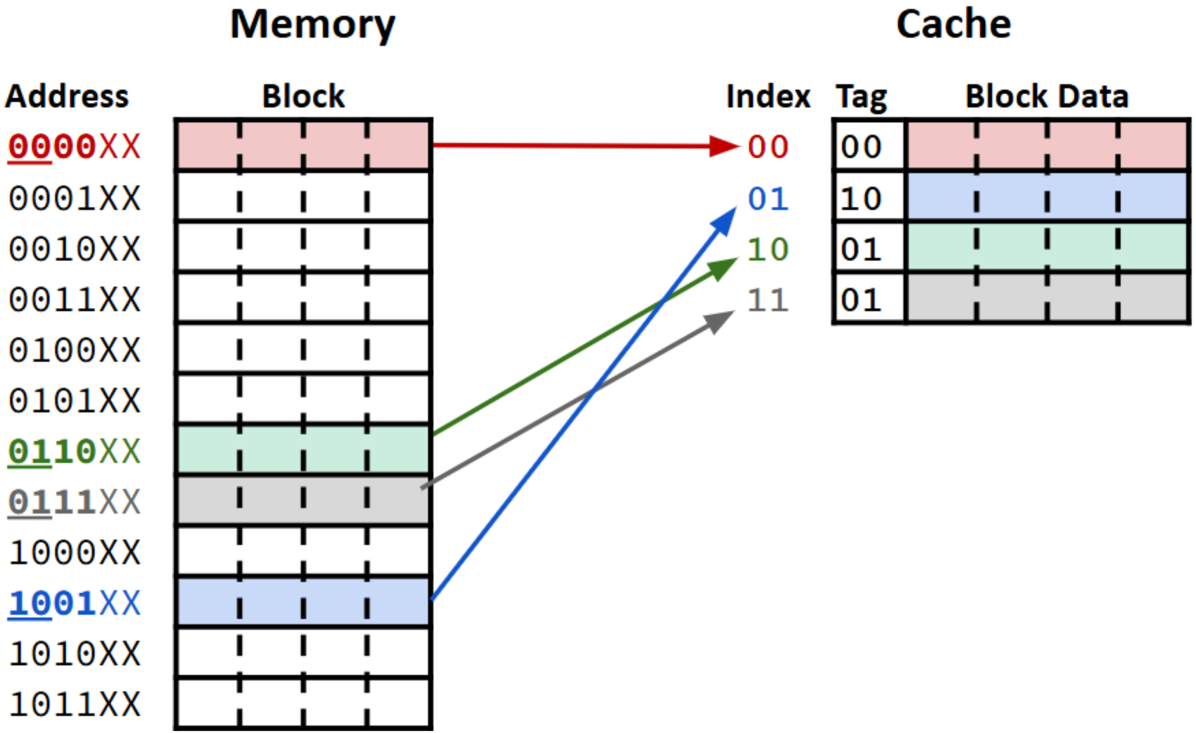
Cache Organization: Sets and Tags

- **Problem**: multiple blocks in memory will map to the same set
  - There will always be more blocks than sets because cache is smaller than memory
  - If we look in a set in the cache, how can we tell which block in memory it has?
- **Solution**: store the remaining bits of the block number as a **tag**



Memory and Cache Example (pt 3)

In this example:
K = 4B
S = 4



Memory

Address	Block
0000XX	
0001XX	
0010XX	
0011XX	
0100XX	
0101XX	
0110XX	
0111XX	
1000XX	
1001XX	
1010XX	
1011XX	

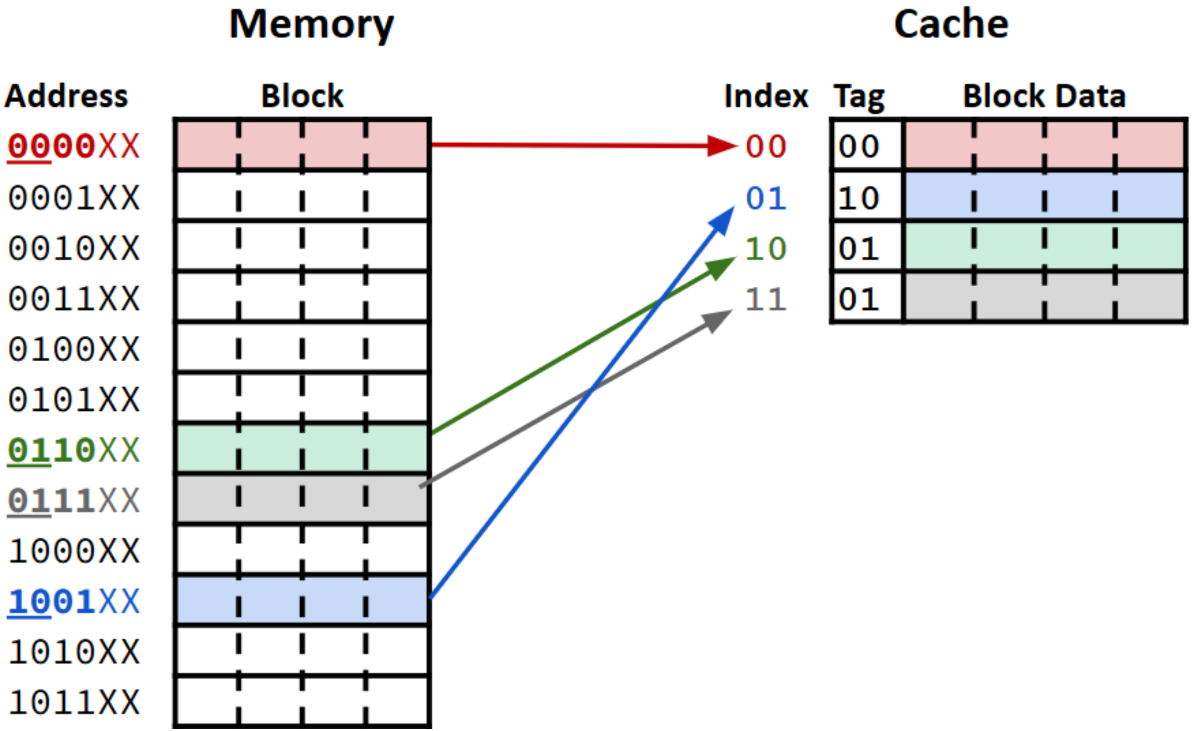
Cache

Index	Tag	Block Data
00	00	
01	10	
10	01	
11	01	

- Save the tag in the cache along with the data block
  - All bits of the block# not used for the index

Memory and Cache Example (pt 4)

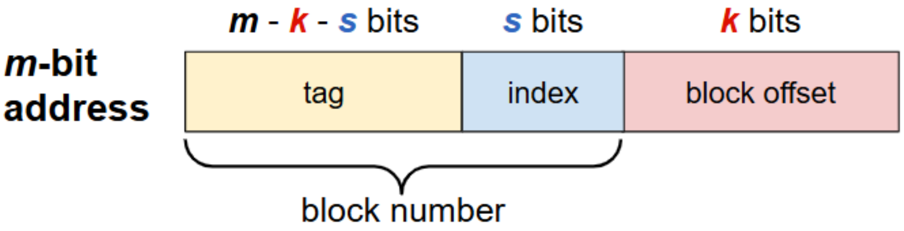
In this example:
K = 4B
S = 4



- Save the tag in the cache along with the data block
  - All bits of the block# not used for the index
- On lookup, check the tag to make sure we have the right block

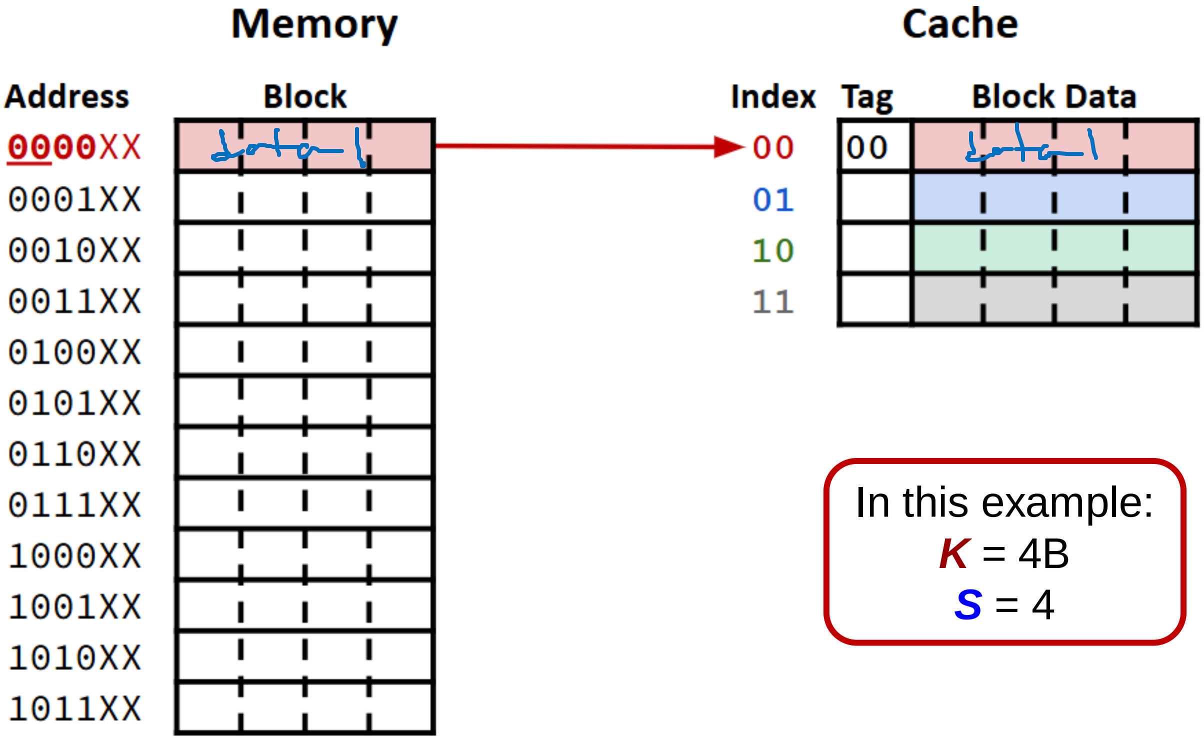
Accessing Data

1. CPU requests a chunk of data at some address
2. Break address up into **Tag**, **Index**, and **Offset**
   a. **O** = lowest ***k*** bits, **I** = next ***s*** bits, **T** = remaining bits
   b. Check set **I** in the cache
   c. If the tag matches **T**, return the data starting at offset **O**
   d. Otherwise, load block from memory
      i. Goes into set **I**, update tag to match
      ii. Then return the data at offset **O**



Accessing Data Example: Before

- Block 0 already loaded into the cache
- CPU requests 2B of data at address `0b010001`



Memory

Address	Block
0000XX	data1
0001XX	
0010XX	
0011XX	
0100XX	
0101XX	
0110XX	
0111XX	
1000XX	
1001XX	
1010XX	
1011XX	

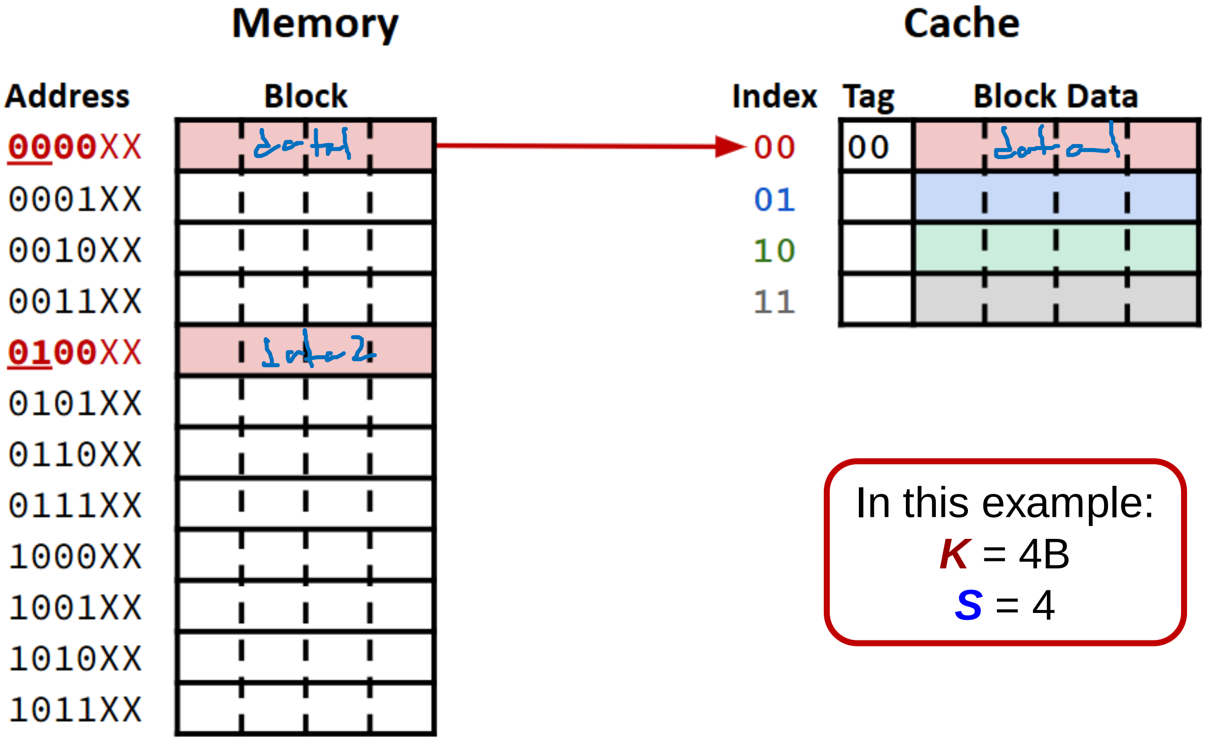
Cache

Index	Tag	Block Data
00	00	data1
01		
10		
11		

In this example:
K = 4B
S = 4

Accessing Data Example: T/I/O breakdown

- *k* = 2, *s* = 2
- CPU requests data at address `0b010001`
    - ○ **T** = `0b01`
    - ○ **I** = `0b00`
    - ○ **O** = `0b01`



Memory

Address	Block
0000XX	data1
0001XX	
0010XX	
0011XX	
0100XX	data2
0101XX	
0110XX	
0111XX	
1000XX	
1001XX	
1010XX	
1011XX	

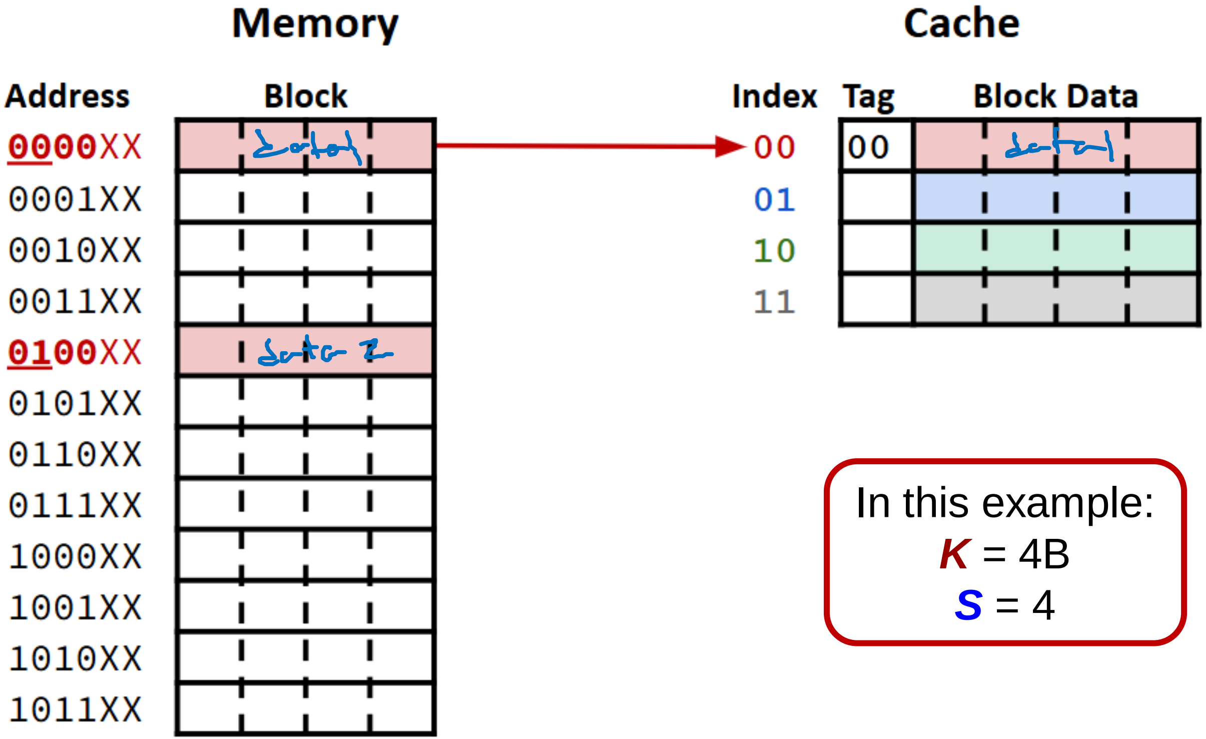
Cache

Index	Tag	Block Data
00	00	data1
01		
10		
11		

In this example:
K = 4B
S = 4

Accessing Data Example: Checking Set

- **T** = `0b01`, **I** = `0b00`, **O** = `0b01`
- Set 0 has tag 00, doesn't match
  - *Cache miss!*



Memory

Address	Block
0000XX	data1
0001XX	
0010XX	
0011XX	
0100XX	data 2
0101XX	
0110XX	
0111XX	
1000XX	
1001XX	
1010XX	
1011XX	

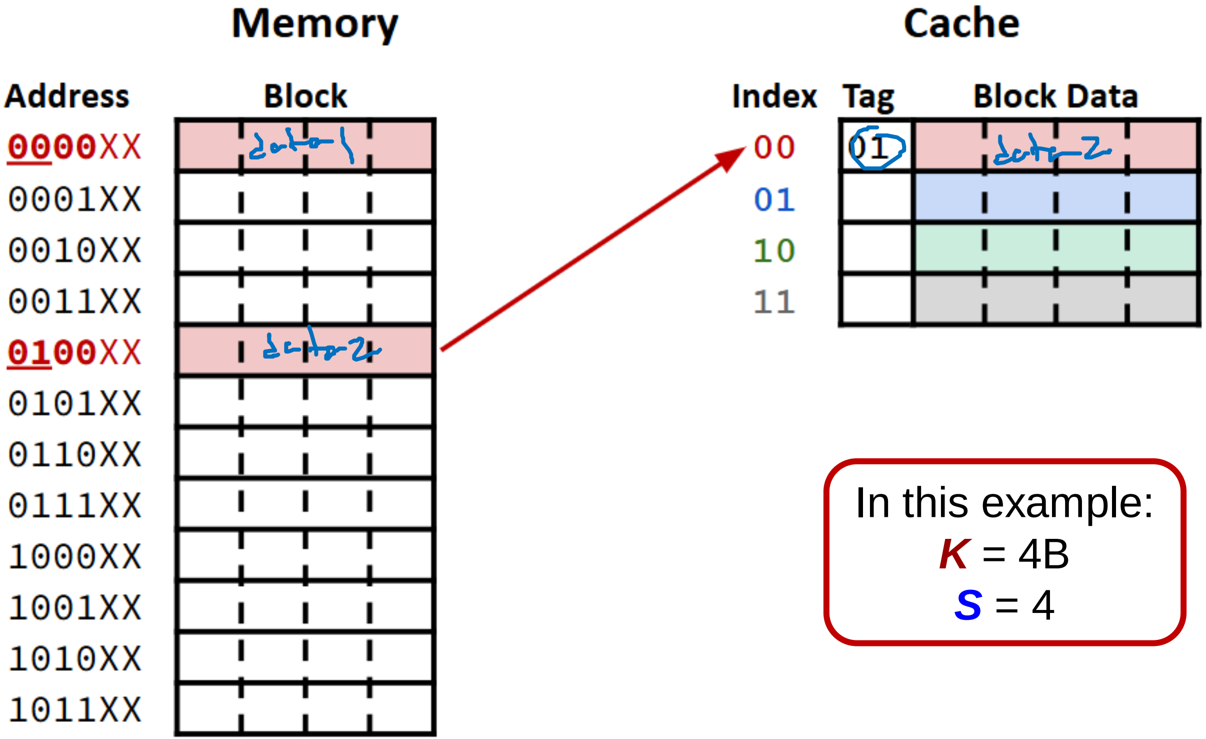
Cache

Index	Tag	Block Data
00	00	data1
01		
10		
11		

In this example:
K = 4B
S = 4

Accessing Data Example: Loading from Memory

- T = `0b01`, I = `0b00`, O = `0b01`
- Store block 4 (0b0100) into the cache in set 0
  - Update Tag



Memory

Address	Block
0000XX	data-1
0001XX	
0010XX	
0011XX	
0100XX	data-2
0101XX	
0110XX	
0111XX	
1000XX	
1001XX	
1010XX	
1011XX	

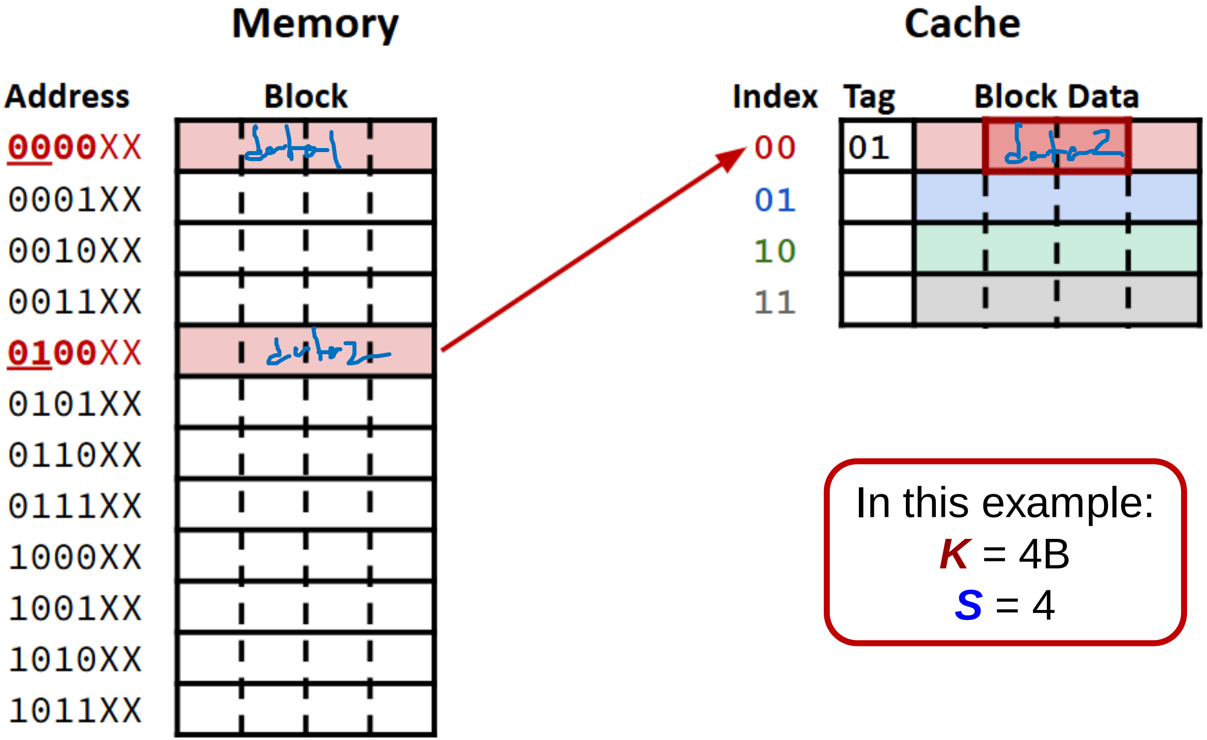
Cache

Index	Tag	Block Data
00	01	data-2
01		
10		
11		

In this example:
K = 4B
S = 4

Accessing Data Example: Returning Data

- **T** = `0b01`, **I** = `0b00`, **O** = `0b01`
- Return data starting at offset 1



Memory

Address	Block
0000XX	data1
0001XX	
0010XX	
0011XX	
0100XX	data2
0101XX	
0110XX	
0111XX	
1000XX	
1001XX	
1010XX	
1011XX	

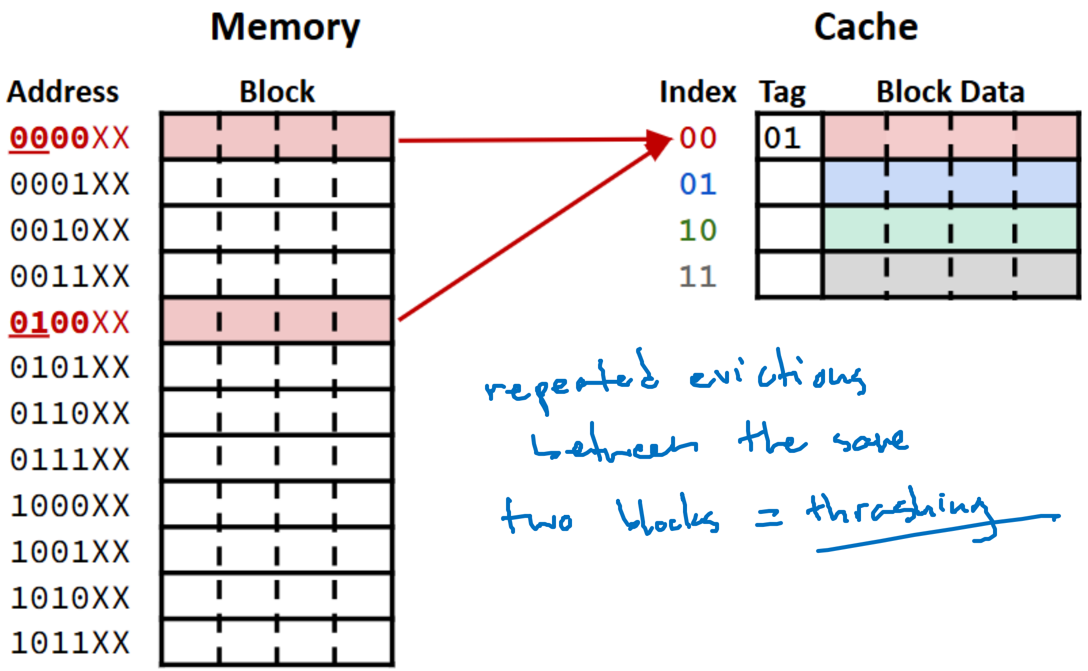
Cache

Index	Tag	Block Data
00	01	data2
01		
10		
11		

In this example:
K = 4B
S = 4

Collisions

- **Problem**: multiple blocks map to the same set
  - Collision occurs when we try to load a block into a set that already has data
    - Evict the old block to make room
  - How can we fix this?
    - *Next lecture!*



Memory

Address	Block
0000XX	
0001XX	
0010XX	
0011XX	
0100XX	
0101XX	
0110XX	
0111XX	
1000XX	
1001XX	
1010XX	
1011XX	

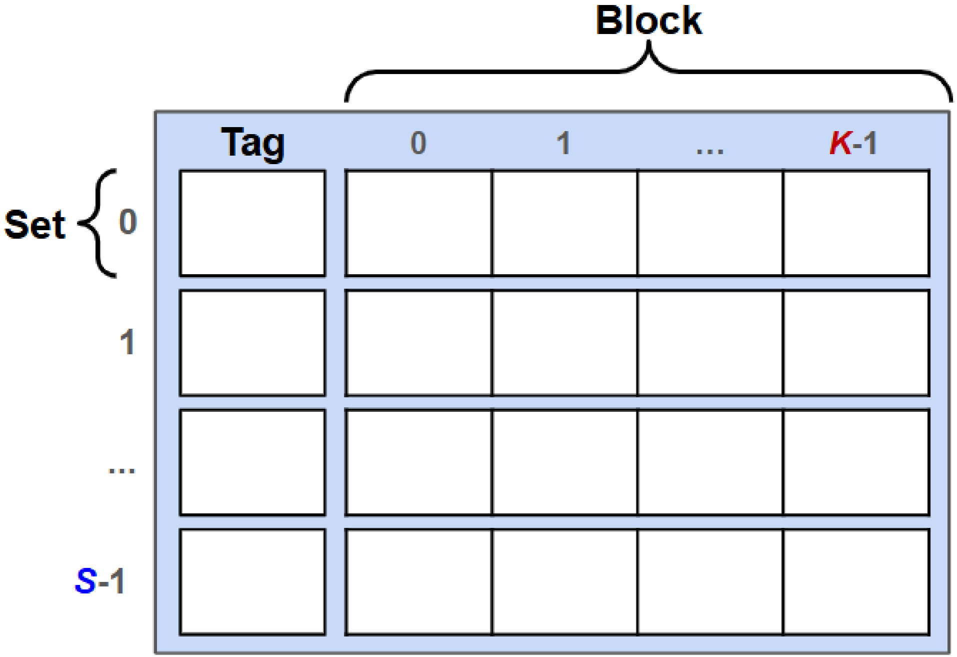
Cache

Index	Tag	Block Data
00	01	
01		
10		
11		

repeated evictions between the same two blocks = thrashing

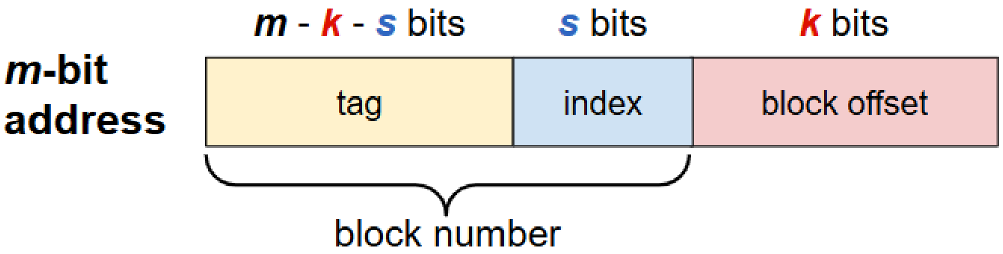
Summary: Cache Terminology

- Memory is broken up into aligned **blocks**
- Cache is broken up into **sets**
  - Each set holds one block (*for now*)
  - Store **tag** along with data block
  - Sets referenced by their **index**
- **Cache size** = number of bytes of data the cache can hold
  - Number of sets * block size



Summary: Address Translation

- Block size = K
  - $k = \log_2(K)$
- Cache size = C
- Number of sets = $S = C \div K$
  - $s = \log_2(S)$



- Divide addresses (a) into fields
  - **Offset** = lowest k bits = a % K
    - Starting location within a block
  - **Index** = next s bits = $(a \div K)$ % S
    - Which set the block is in
  - **Tag** = Remaining bits = $(a \div K) \div S$
    - Used to distinguish different blocks with the same index