Caches II

CSE 351 Winter 2018

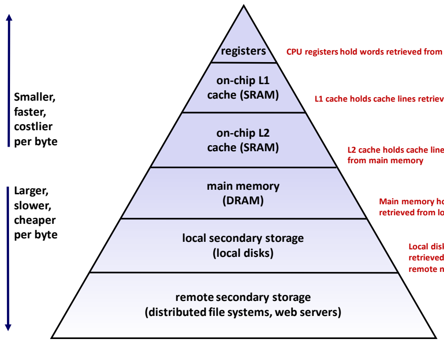
Instructor:
Mark Wyse

Teaching Assistants:
Kevin Bi
Parker DeWilde
Emily Furst
Sarah House
Waylon Huang
Vinny Palaniappan

Administrative

- Lab 3 due *Friday* (2/16)
- Homework 4 released today (Structs, Caches)
- Midterm Regrade Requests due Friday (2/16)

An Example Memory Hierarchy



Smaller, faster, costlier per byte

Larger, slower, cheaper per byte

registers

on-chip L1 cache (SRAM)

on-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

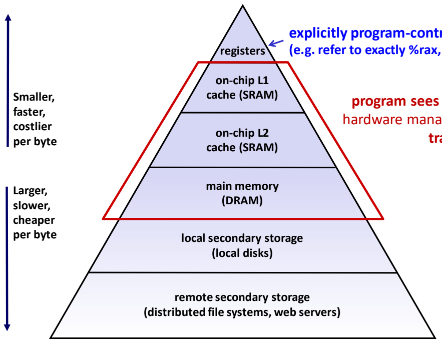
L1 cache holds cache lines retrieved from L2 cache

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



Smaller, faster, costlier per byte

Larger, slower, cheaper per byte

registers

on-chip L1 cache (SRAM)

on-chip L2 cache (SRAM)

main memory (DRAM)

local secondary storage (local disks)

remote secondary storage (distributed file systems, web servers)

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

program sees “memory”; hardware manages caching transparently

Memory Hierarchies

- Fundamental idea of 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
- Why do memory hierarchies work?
  - Because of locality, programs tend to access the data at level k more often than they access the data at level k+1
  - Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit
- *Big Idea:* The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top

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

- Fundamental Equation: $C = S * E * B$
- Cache Size (C): total capacity (Bytes) of cache
- Block Size (B): unit of transfer between $ and Mem
- Sets (S): collection of blocks
  - Cache can be thought of as an “array of sets”
- Associativity (E): number of cache blocks per set
- Address Bits (m): number of bits in address

Cache Organization (1)

- Block Size (B): unit of transfer between $ and Mem
  - Given in bytes and always a power of 2 (*e.g.* 64 Bytes)
  - Blocks consist of adjacent bytes (differ in address by 1)
    - Spatial locality!
- Offset field
  - Low-order $\log_2(B) = b$ 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)

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

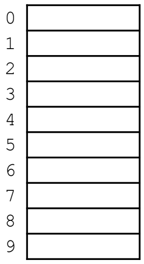
Cache Organization (2)

- 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/B)
  - Example: C = 32 KB = 512 blocks if using 64-Byte 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!

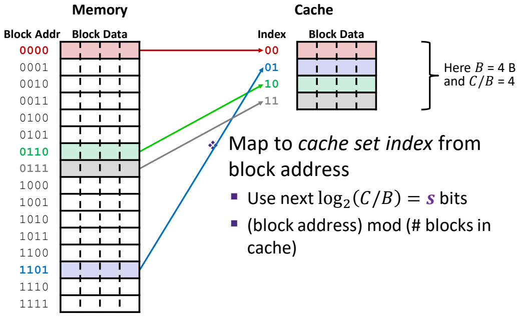
Review: Hash Tables for Fast Lookup

Insert:
5
27
34
102
119

Apply hash function to map data to “buckets”

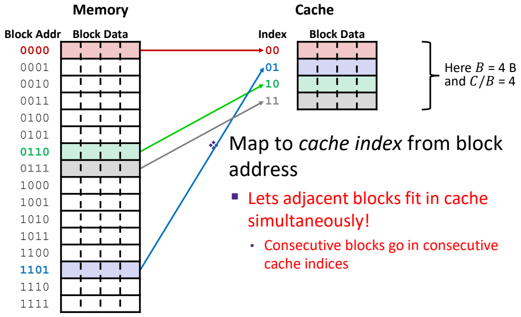


Place Data in Cache by Hashing Address

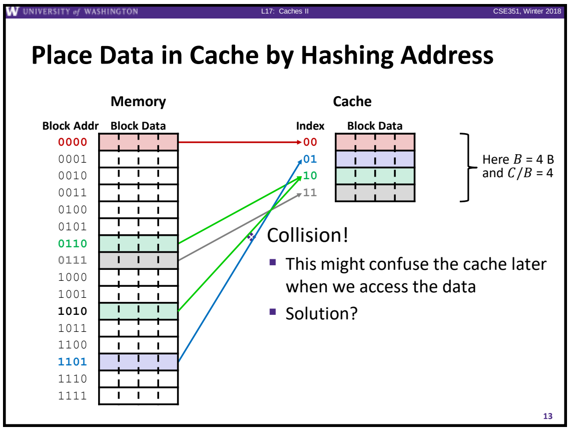


- Map to *cache set index* from block address
  - Use next $\log_2(C/B) = s$ bits
  - (block address) mod (# blocks in cache)

Place Data in Cache by Hashing Address

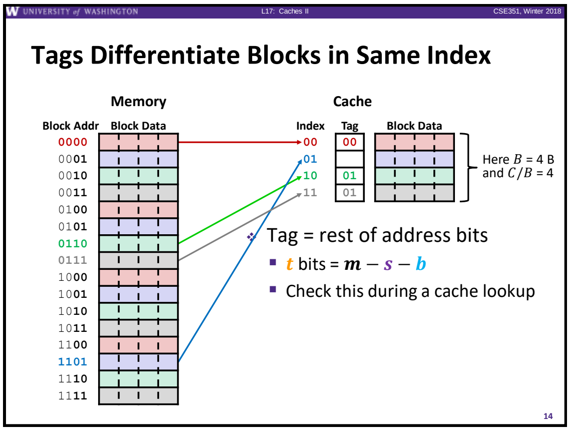


- Map to *cache index* from block address
  - Lets adjacent blocks fit in cache simultaneously!
    - Consecutive blocks go in consecutive cache indices



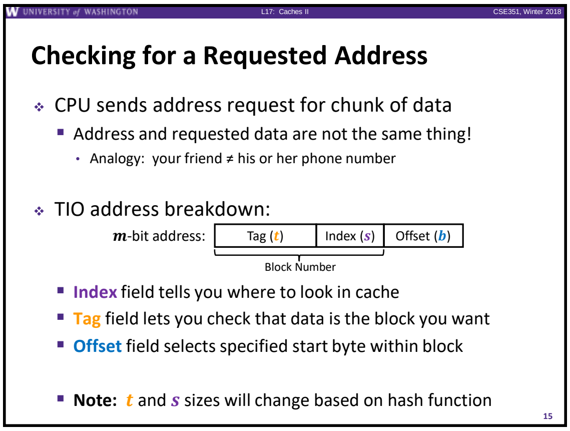
Place Data in Cache by Hashing Address

- 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 = $m - s - b$
  - Check this during a cache lookup



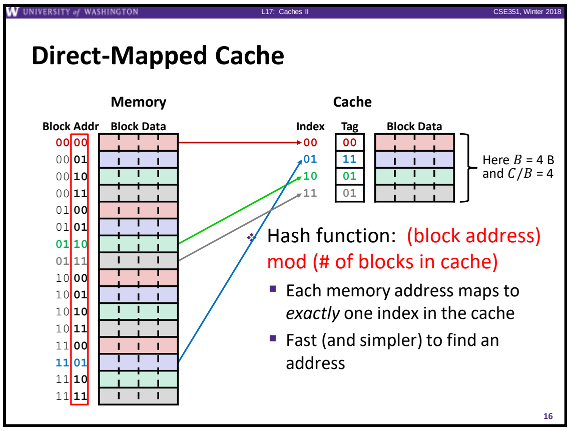
Checking for a Requested Address

- 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: | Tag (t) | Index (s) | Offset (b) |

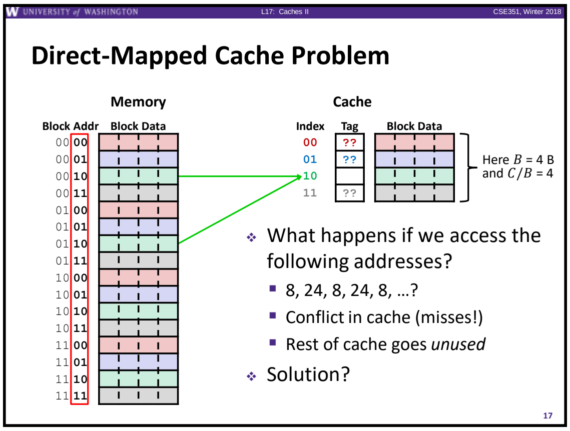
Block Number

  - **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



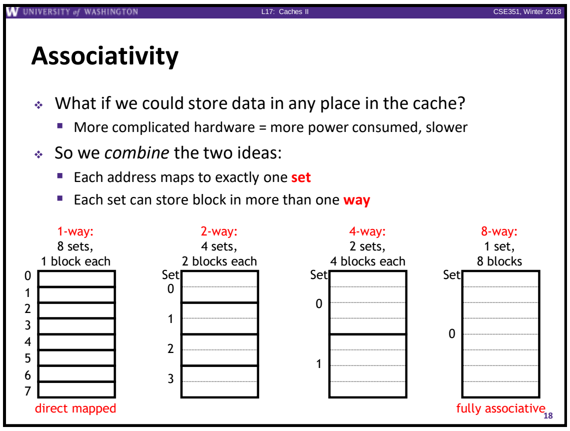
Direct-Mapped Cache

- 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



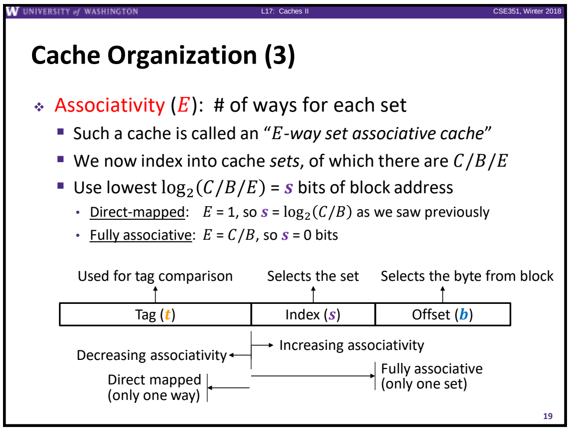
Direct-Mapped Cache Problem

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

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

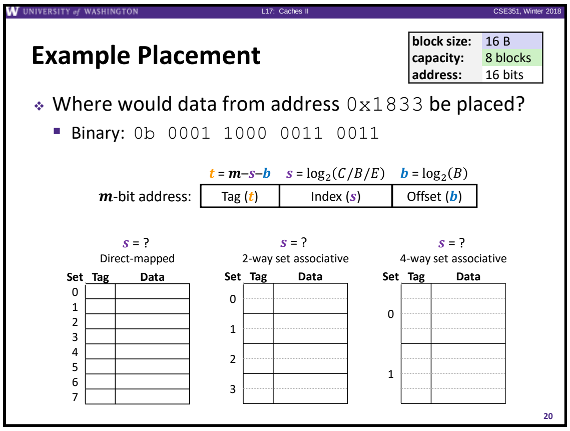
Used for tag comparison | Selects the set | Selects the byte from block

Tag (t)	Index (s)	Offset (b)

Decreasing associativity ← → Increasing associativity

Direct mapped (only one way)

Fully associative (only one set)



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–b s = $\log_2(C/B/E)$ b = $\log_2(B)$

m-bit address:

Tag (t)	Index (s)	Offset (b)

s = ? Direct-mapped

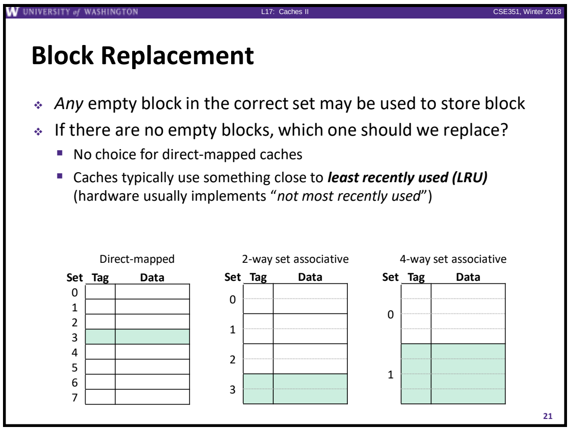
Set	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

s = ? 2-way set associative

Set	Tag	Data
0		
1		
2		
3		

s = ? 4-way set associative

Set	Tag	Data
0		
1		



Block Replacement

- *Any* empty block in the correct set may be used to store block
- If there are no empty blocks, which one should we replace?
  - No choice for direct-mapped caches
  - Caches typically use something close to ***least recently used (LRU)***
    (hardware usually implements “*not most recently used*”)

Direct-mapped

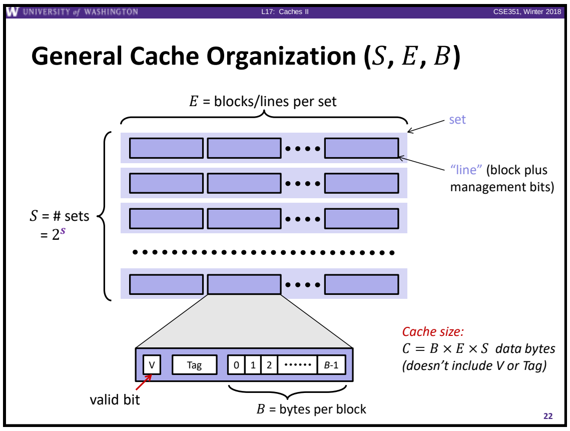
Set	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

2-way set associative

Set	Tag	Data
0		
1		
2		
3		

4-way set associative

Set	Tag	Data
0		
1		



General Cache Organization (S, E, B)

E = blocks/lines per set

set

“line” (block plus management bits)

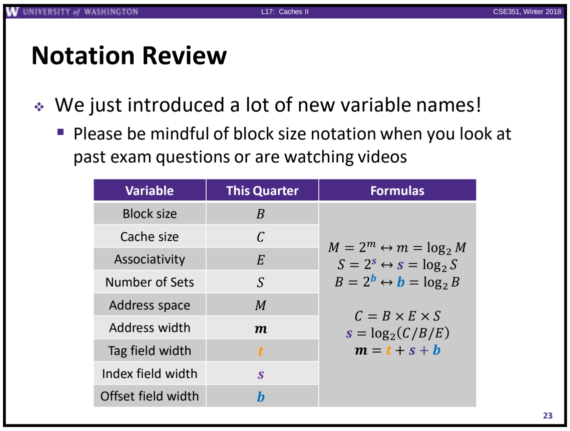
S = # sets = 2^s

Cache size:
$C = B \times E \times S$ data bytes
(doesn’t include V or Tag)

V | Tag | 0 | 1 | 2 | ······ | B-1

valid bit

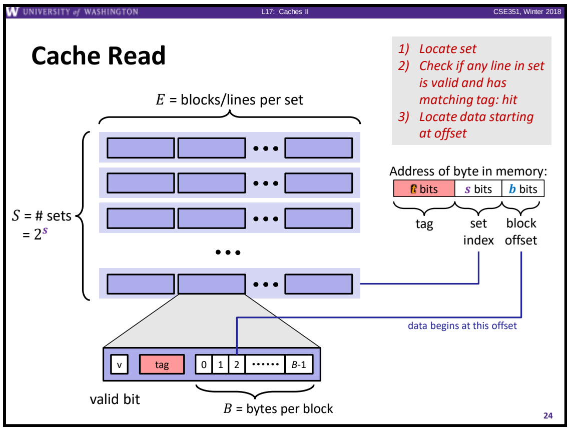
B = bytes per block



Notation Review

- We just introduced a lot of new variable names!
  - Please be mindful of block size notation when you look at past exam questions or are watching videos

Variable	This Quarter	Formulas
Block size	B	$M = 2^m \leftrightarrow m = \log_2 M$ $S = 2^s \leftrightarrow s = \log_2 S$ $B = 2^b \leftrightarrow b = \log_2 B$ $C = B \times E \times S$ $s = \log_2(C/B/E)$ $m = t + s + b$
Cache size	C	
Associativity	E	
Number of Sets	S	
Address space	M	
Address width	m	
Tag field width	t	
Index field width	s	
Offset field width	b	



Cache Read

1) *Locate set*
2) *Check if any line in set is valid and has matching tag: hit*
3) *Locate data starting at offset*

E = blocks/lines per set

S = # sets = 2^s

Address of byte in memory:

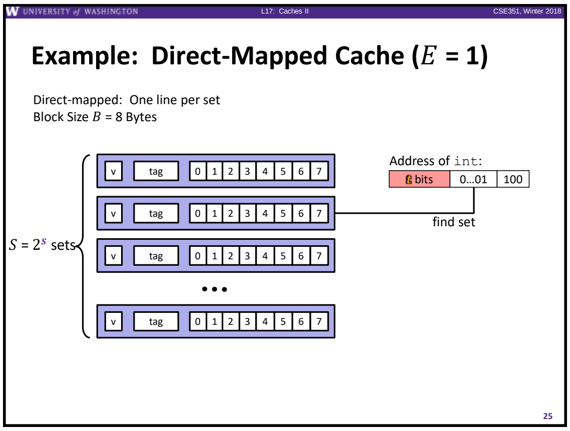
t bits	s bits	b bits
tag	set index	block offset

data begins at this offset

v | tag | 0 | 1 | 2 | ······ | B-1

valid bit

B = bytes per block



Example: Direct-Mapped Cache (E = 1)

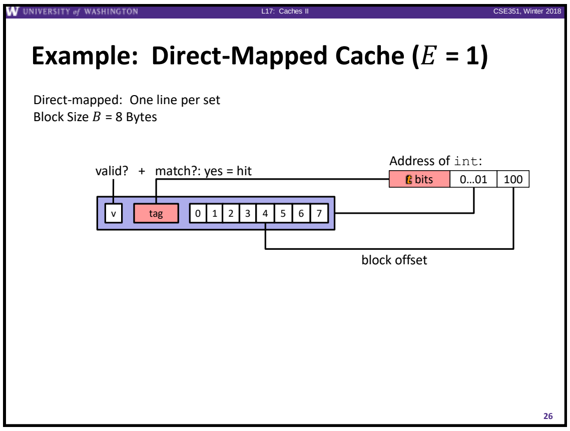
Direct-mapped: One line per set
Block Size B = 8 Bytes

$S = 2^s$ sets

Address of `int`:

t bits	0…01	100

find set



Example: Direct-Mapped Cache (E = 1)

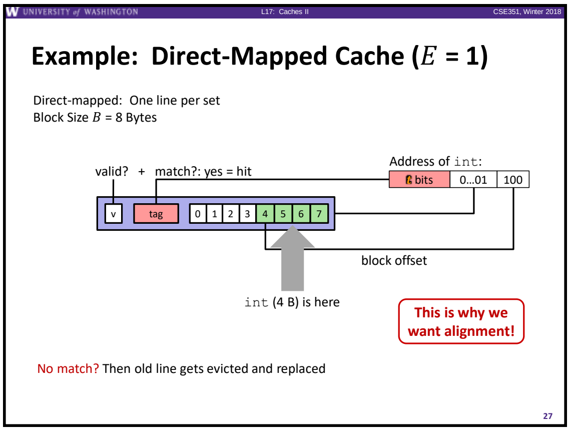
Direct-mapped: One line per set
Block Size B = 8 Bytes

Address of `int`:

t bits	0…01	100

valid? + match?: yes = hit

block offset



Example: Direct-Mapped Cache (E = 1)

Direct-mapped: One line per set
Block Size B = 8 Bytes

Address of `int`:

t bits	0…01	100

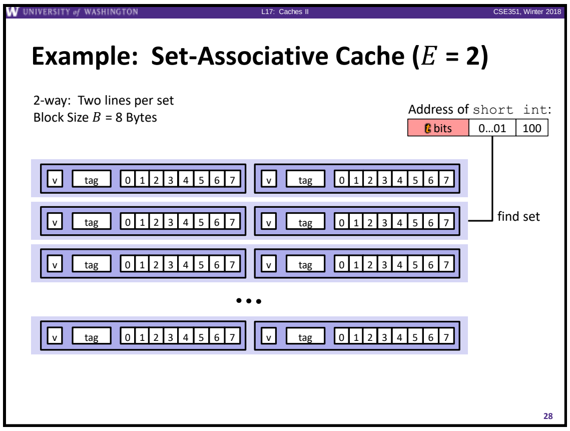
valid? + match?: yes = hit

block offset

`int` (4 B) is here

This is why we want alignment!

No match? Then old line gets evicted and replaced



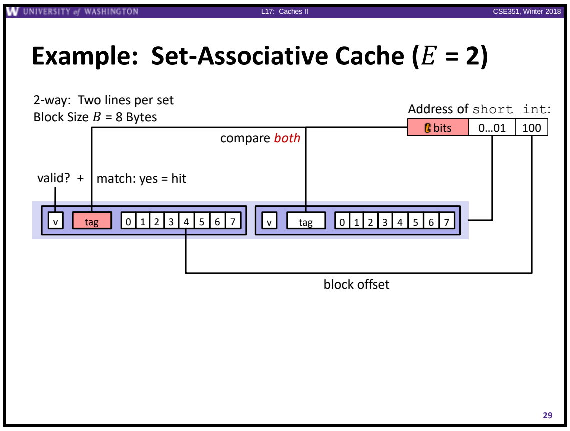
Example: Set-Associative Cache (E = 2)

2-way: Two lines per set
Block Size B = 8 Bytes

Address of `short int`:

t bits	0…01	100

find set



Example: Set-Associative Cache (E = 2)

2-way: Two lines per set
Block Size B = 8 Bytes

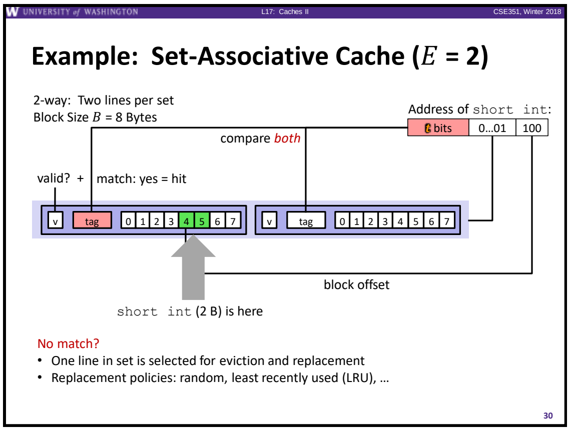
Address of `short int`:

t bits	0…01	100

compare *both*

valid? + match: yes = hit

block offset



Example: Set-Associative Cache (E = 2)

2-way: Two lines per set
Block Size B = 8 Bytes

Address of `short int`:

t bits	0…01	100

compare *both*

valid? + match: yes = hit

block offset

`short int` (2 B) is here

No match?

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), …