UC Berkeley
Teaching Professor
Dan Garcia

CS61C

Great Ideas in Computer Architecture

(a.k.a. Machine Structures)

UC Berkeley
Teaching Professor
Lisa Yan

Caches III

cs61c.org

The way to remember #s

What is 2^{37}?

How many bits to address (I.e., what's `ceil` $\texttt{log}_2$ (= `lg`) of) 2.5 TiB?

Answer! 2^{XY} means…

X=0 ⇒ ---

X=1 ⇒ kibi ~10^3

X=2 ⇒ mebi ~10^6

X=3 ⇒ gibi ~10^9

X=4 ⇒ tebi ~10^{12}

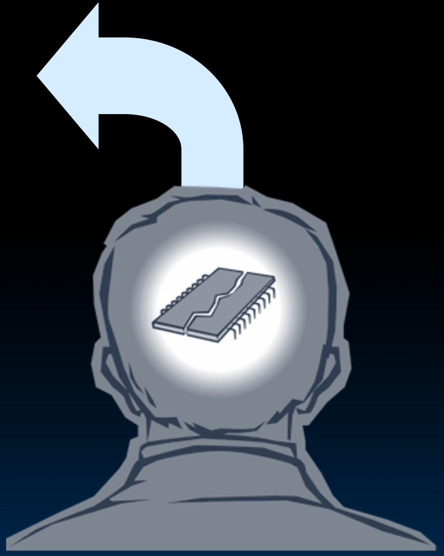
X=5 ⇒ pebi ~10^{15}

X=6 ⇒ exbi ~10^{18}

X=7 ⇒ zebi ~10^{21}

X=8 ⇒ yobi ~10^{24}

Y=0 ⇒ 1
Y=1 ⇒ 2
Y=2 ⇒ 4
Y=3 ⇒ 8
Y=4 ⇒ 16
Y=5 ⇒ 32
Y=6 ⇒ 64
Y=7 ⇒ 128
Y=8 ⇒ 256
Y=9 ⇒ 512



MEMORIZE!

☑

Berkeley
UNIVERSITY OF CALIFORNIA

Direct Mapped Example

Accessing data in a direct mapped cache

- Ex.: 16KB of data, direct-mapped, 4 word blocks
  - Can you work out height, width, area?
- Read 4 addresses

```
1.  0x00000014
2.  0x0000001C
3.  0x00000034
4.  0x00008014
```

- Memory values here:

Memory

Address (hex)	Value of Word
…	…
00000010	a
00000014	b
00000018	c
0000001C	d
…	…
00000030	e
00000034	f
00000038	g
0000003C	h
…	…
00008010	i
00008014	j
00008018	k
0000801C	l
…	…

Accessing data in a direct mapped cache

- 4 Addresses:
  - `0x00000014, 0x0000001C, 0x00000034, 0x00008014`
- 4 Addresses divided (for convenience) into Tag, Index, Byte Offset fields

```
000000000000000000 0000000001 0100
000000000000000000 0000000001 1100
000000000000000000 0000000011 0100
000000000000000010 0000000001 0100
```

Tag Index Offset

Example: 16 KB Direct-Mapped Cache, 16B blocks

- **Valid bit:** determines whether anything is stored in that row (when computer initially powered up, all entries invalid)

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
•••				•••		
1022	0					
1023	0					

1. Read 0x00000014

- 000000000000000000 0000000001 0100

Tag Index Offset

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

So we read block 1 (0000000001)

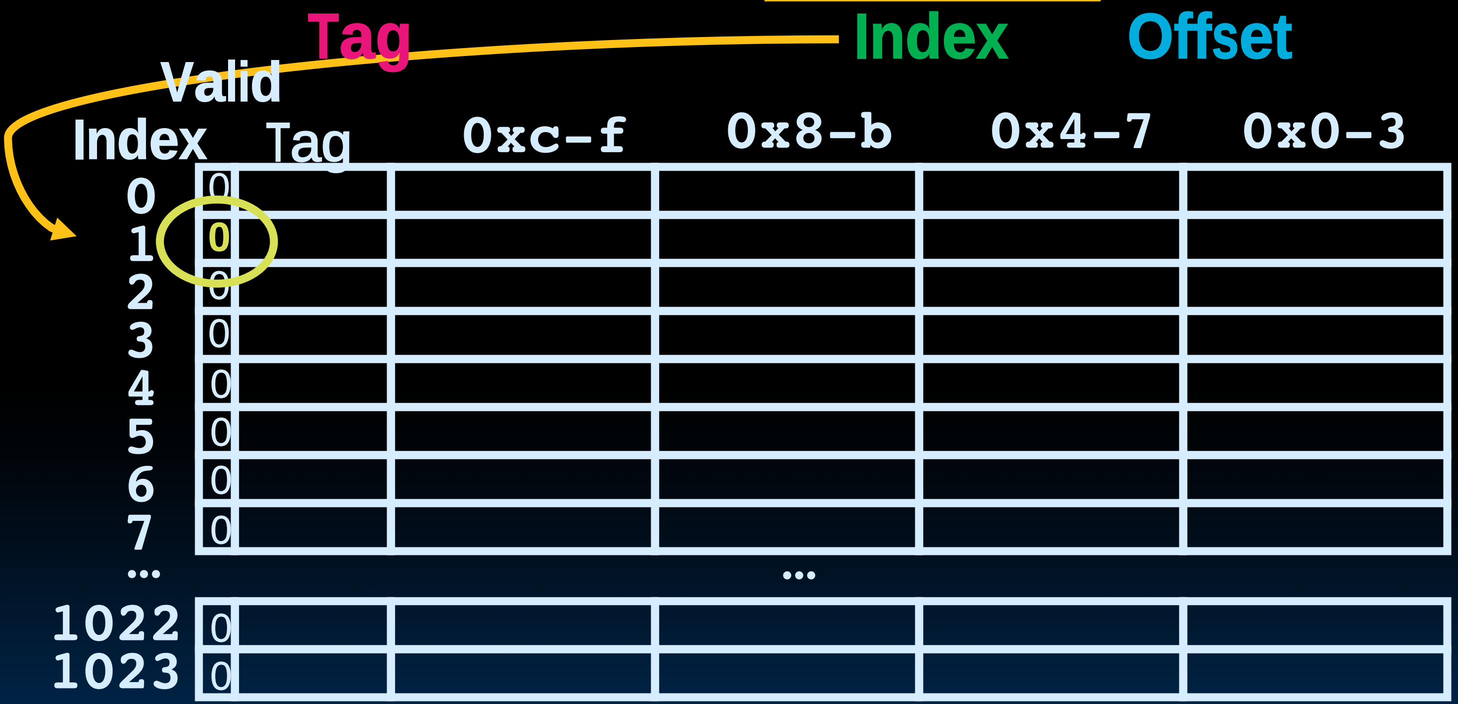
- 000000000000000000 0000000001 0100

Tag Index Offset

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

No valid data

- 0000000000000000000 0000000001 0100



Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

So load that data into cache, setting tag, valid

- 000000000000000000 0000000001 0100

Tag Index Offset

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Read from cache at offset, return word b

- 00000000000000000 0000000001 0100

Tag Index Offset

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

2. Read 0x0000001C = 0…00 0..001 1100

- 000000000000000000 0000000001 1100

Tag Index Offset

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

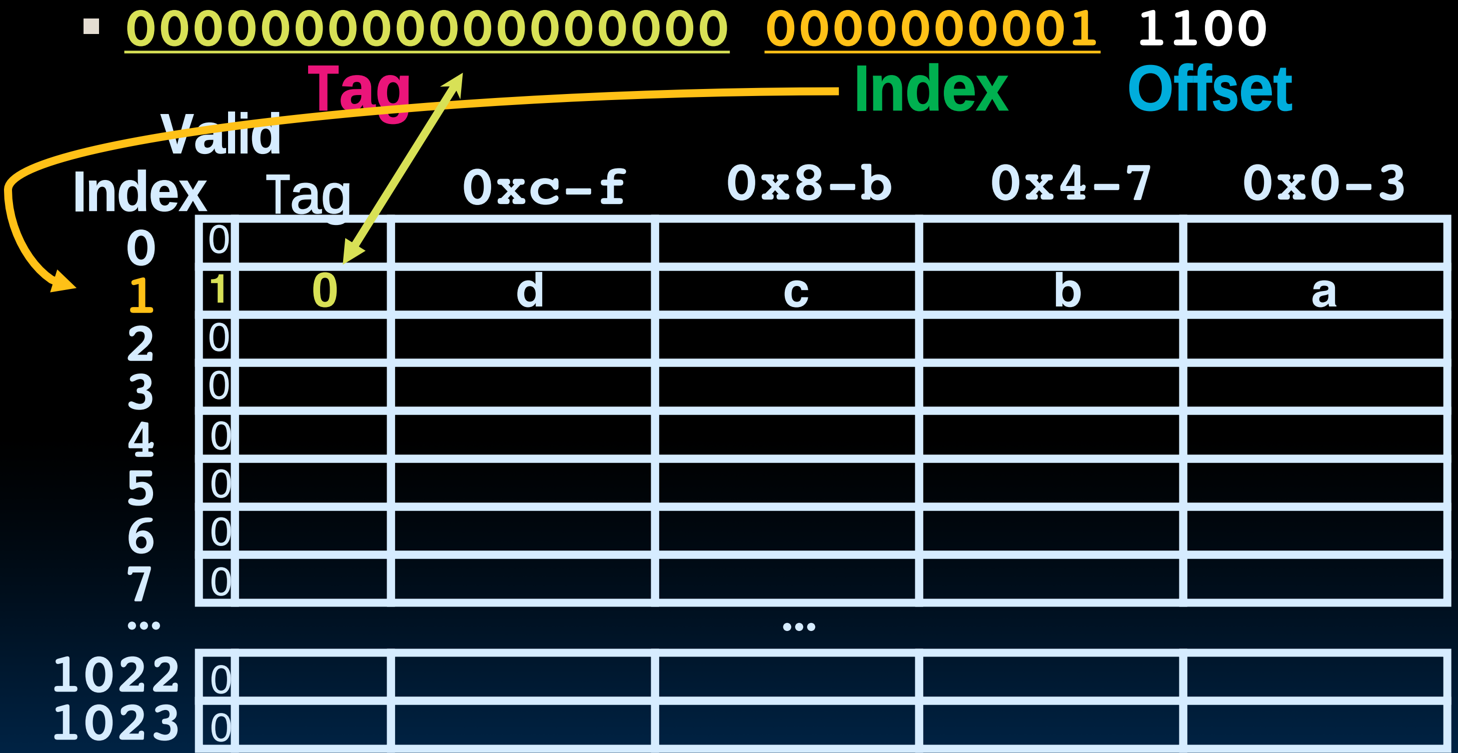
Index is Valid

- 000000000000000000 0000000001 1100

Tag Index Offset

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

Index is Valid, Tag Matches

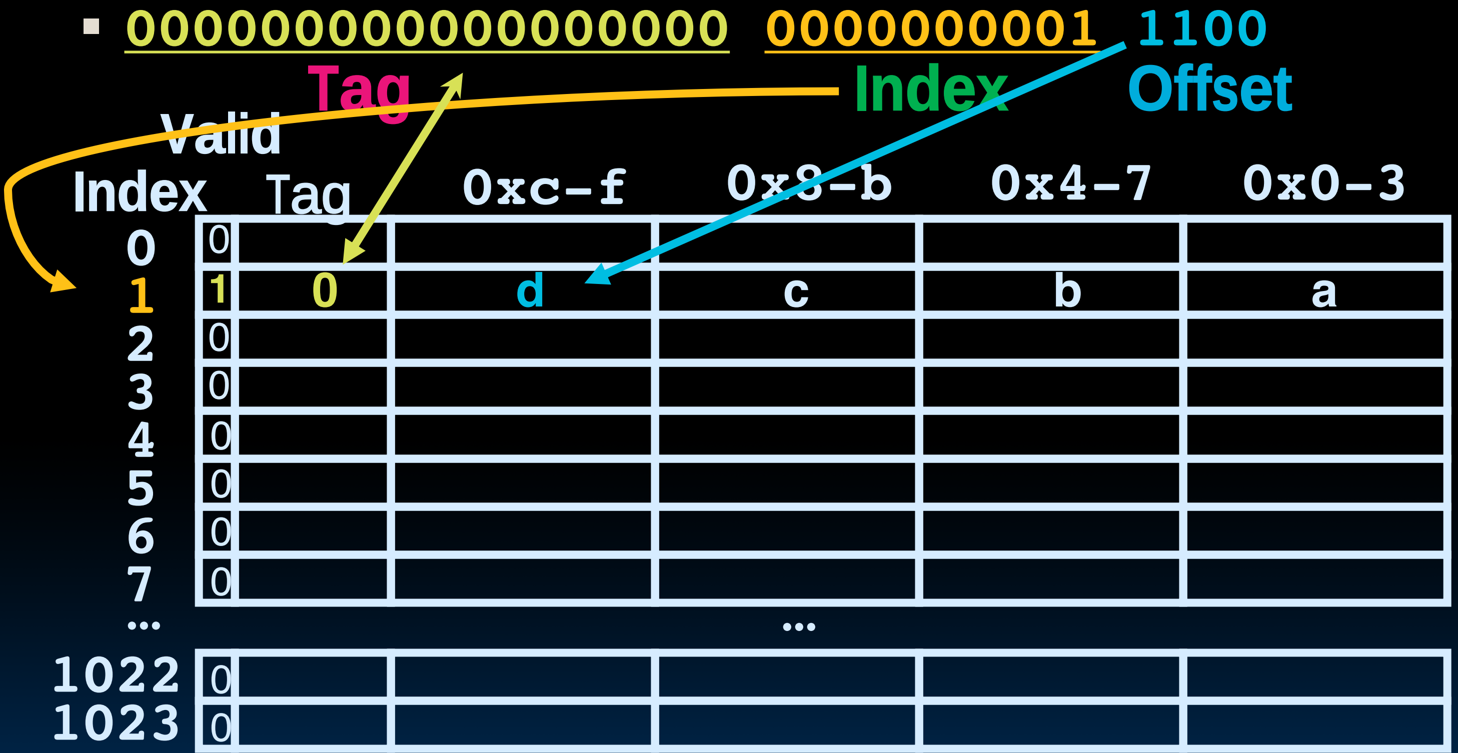


- 00000000000000000000 0000000001 1100

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Index is Valid, Tag Matches, return d

- 000000000000000000 0000000001 1100



Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

3. Read 0x00000034 = 0…00 0..011 0100

- 000000000000000000 0000000011 0100

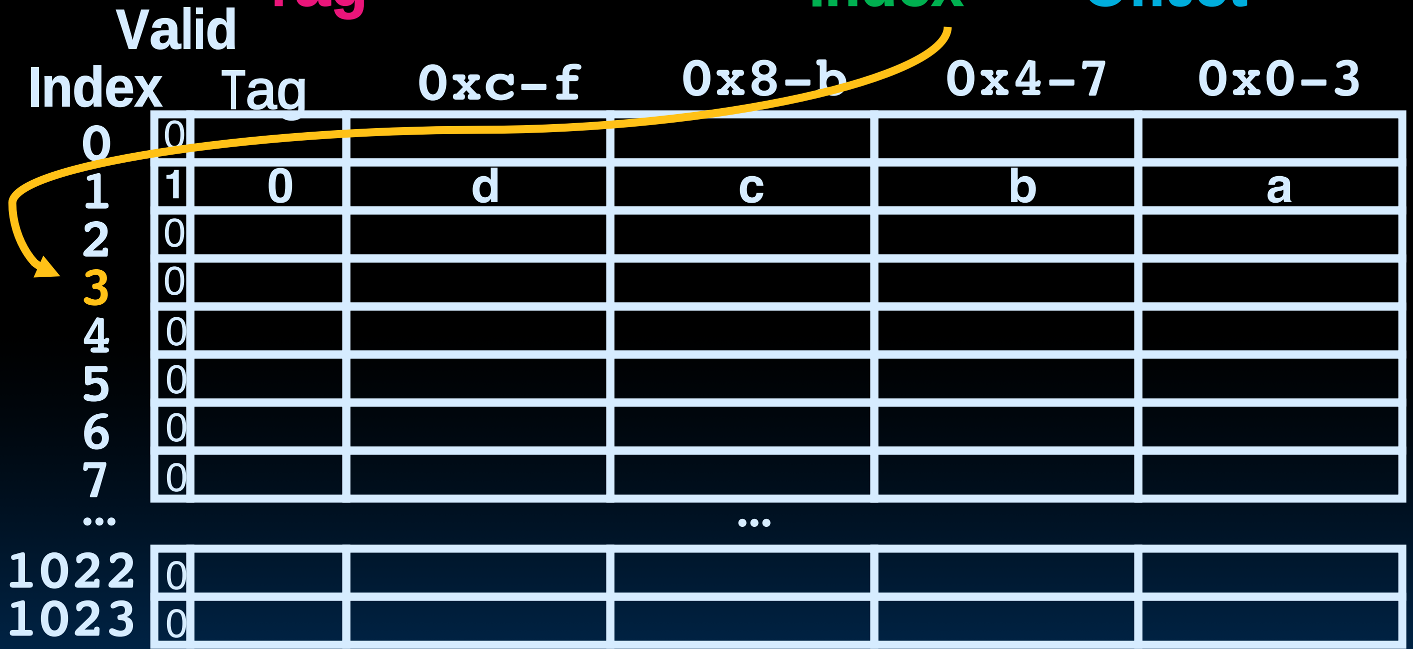
Tag Index Offset

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

So read block 3

- 000000000000000000 <u>0000000011</u> 0100

Tag Index Offset

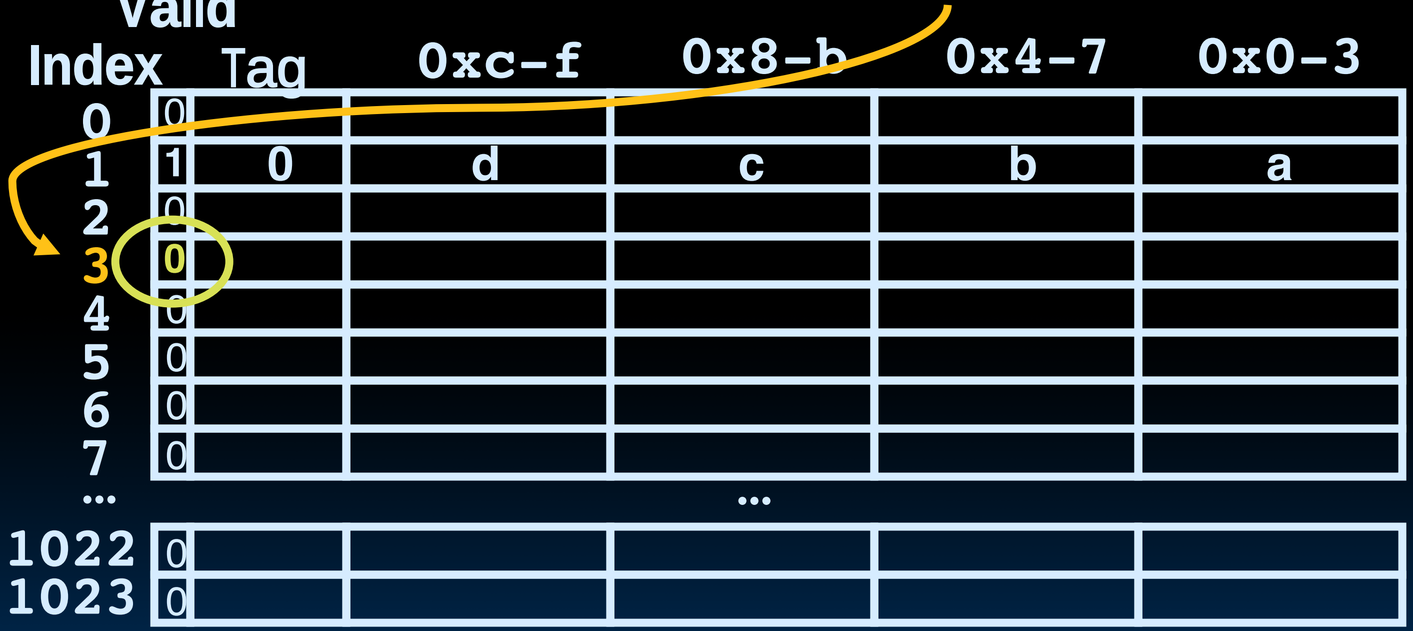


Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

No valid data

- 0000000000000000000 0000000011 0100

Tag Index Offset

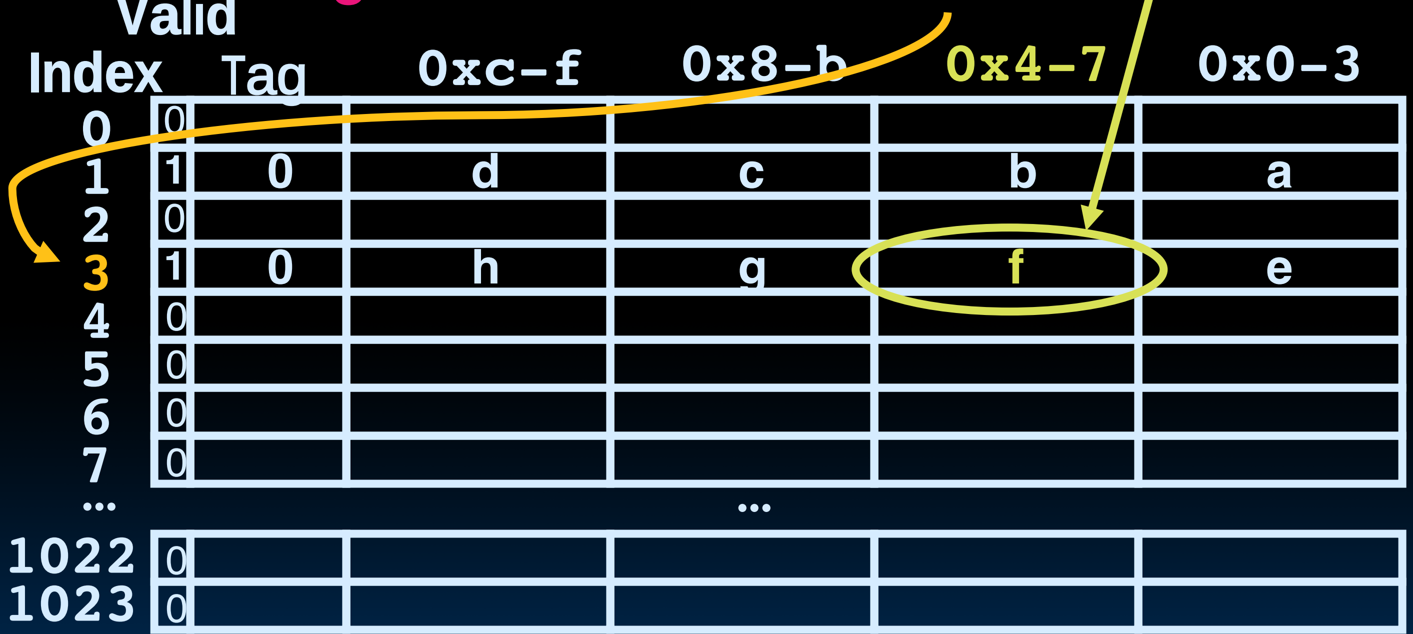


Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
…			…			
1022	0					
1023	0					

Load that cache block, return word f

- 00000000000000000000 0000000011 0100

Tag Index Offset



Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

4. Read 0x00008014 = 0…10 0..001 0100

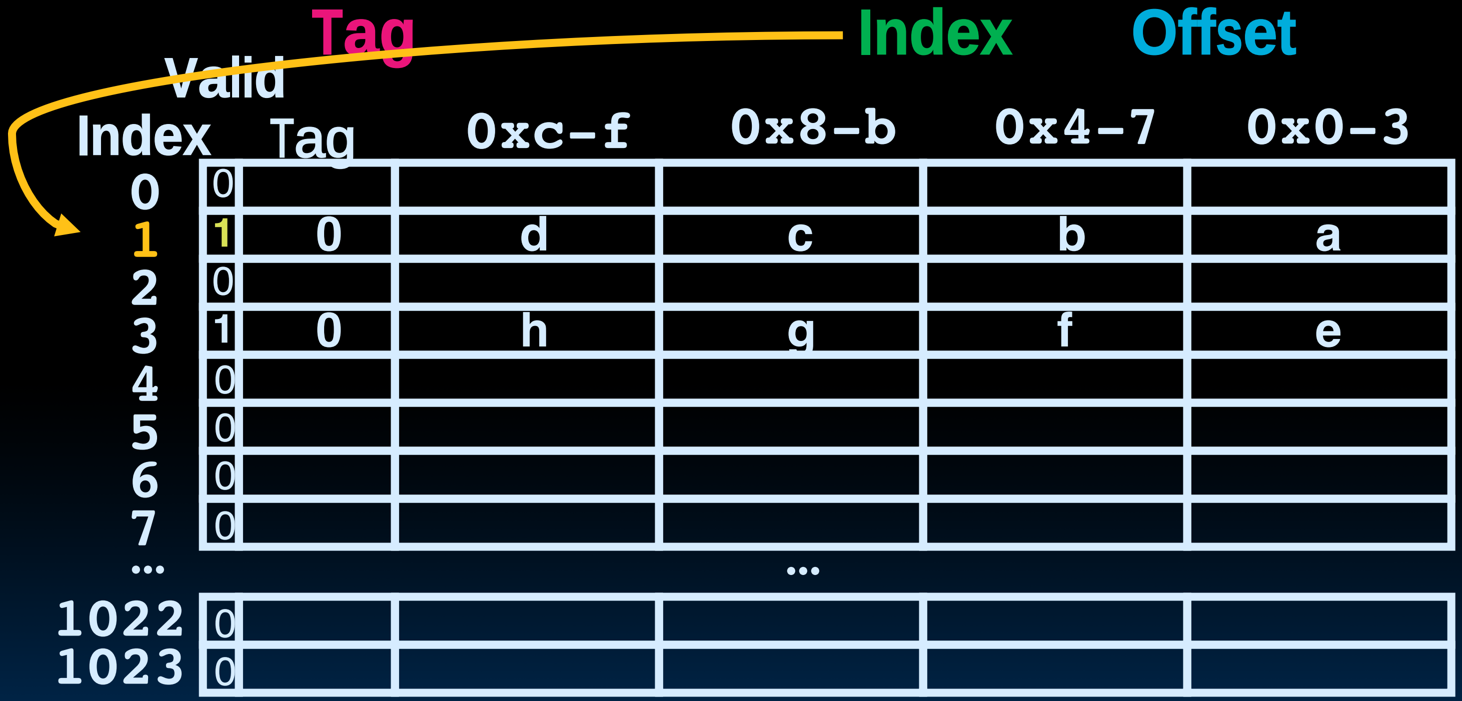
- 000000000000000010 0000000001 0100

Tag Index Offset

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
•••				•••		
1022	0					
1023	0					

So read Cache Block 1, Data is Valid

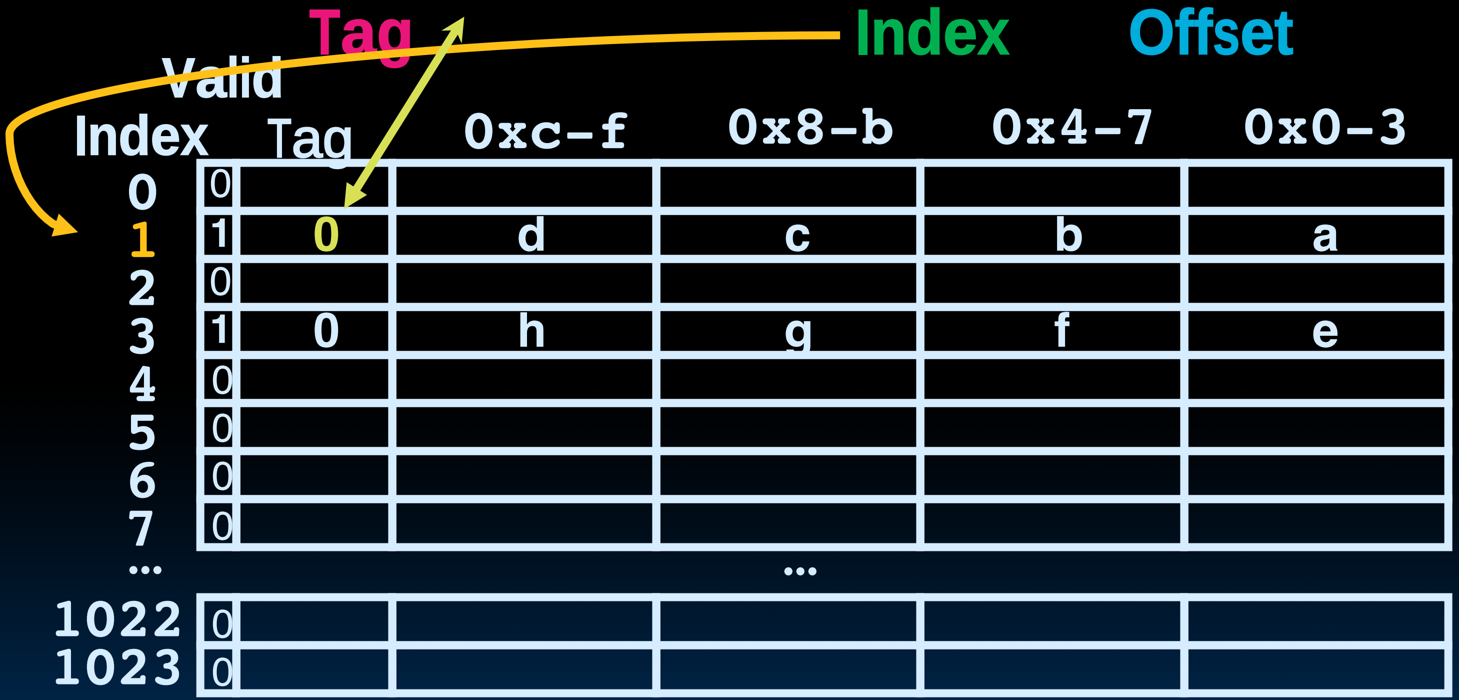
- 00000000000000000010 0000000001 0100



Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Cache Block 1 Tag does not match (0 ≠ 2)

- 000000000000000000010 0000000001 0100



Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Miss, so replace block 1 with new data & tag

- 000000000000000010 0000000001 0100

Tag Index Offset

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	2	l	k	j	i
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

And return word J

- 000000000000000010 0000000001 0100

Tag Index Offset

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	2	l	k	j	i
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Do an example yourself. What happens?

- Chose from: Cache: Hit, Miss, Miss w. replace
  Values returned: a ,b, c, d, e, ..., k, l
- Read address **0x00000030** ?
  `000000000000000000 0000000011 0000`
- Read address **0x0000001c** ?
  `000000000000000000 0000000001 1100`

Index						
0	0					
1	1	2	l	k	j	i
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					

Answers

- **`0x00000030` a hit**

  Index = 3, Tag matches, Offset = 0, value = e

- **`0x0000001c` a miss**

  Index = 1, Tag mismatch, so replace from memory, Offset = `0xc`, value = d

- **Since reads, values must = memory values whether or not cached:**
  - `0x00000030` = e
  - `0x0000001c` = d

Memory

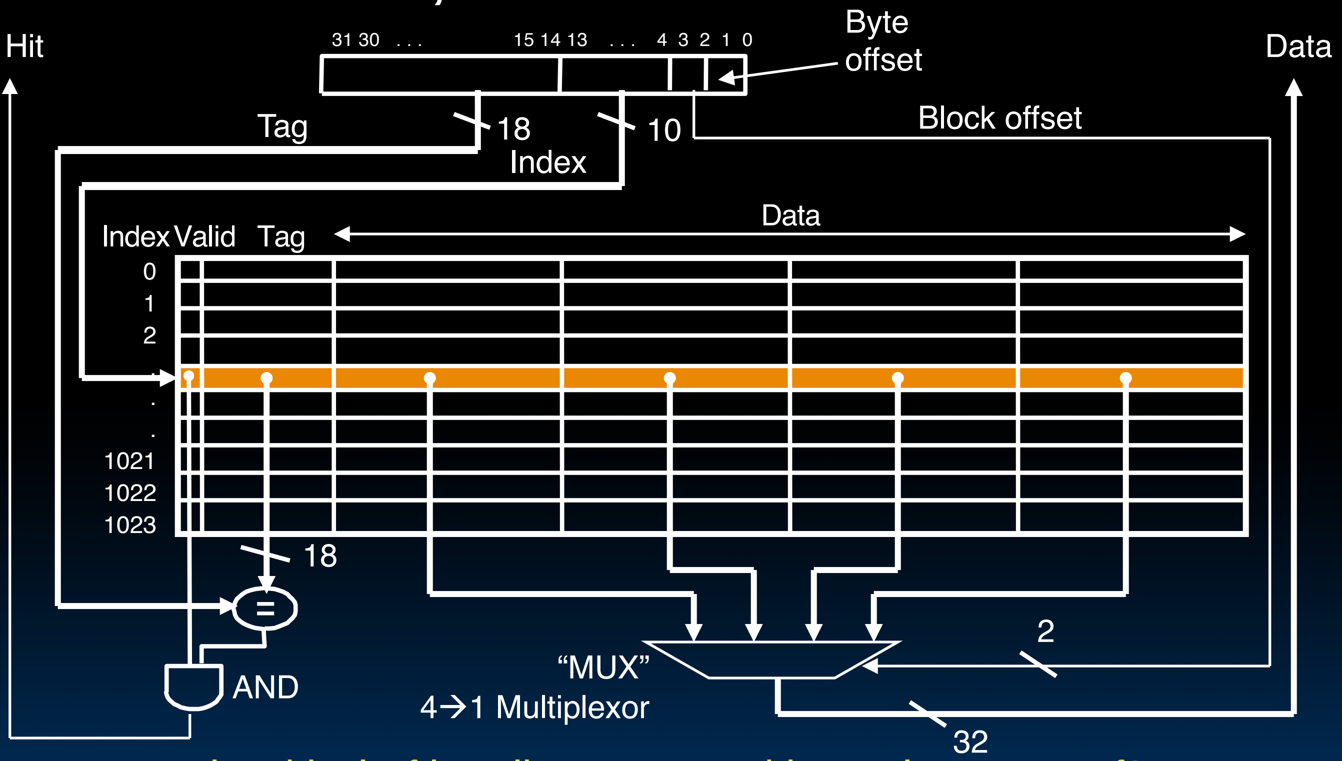
Address (hex)	Value of Word
...	...
00000010	a
00000014	b
00000018	c
0000001C	d
...	...
00000030	e
00000034	f
00000038	g
0000003C	h
...	...
00008010	i
00008014	j
00008018	k
0000801C	l
...	...

☑

Writes, Block Sizes, Misses

Multiword-Block Direct-Mapped Cache

- **Four words/block, cache size = 4K words**



What kind of locality are we taking advantage of?

Garcia, Yan

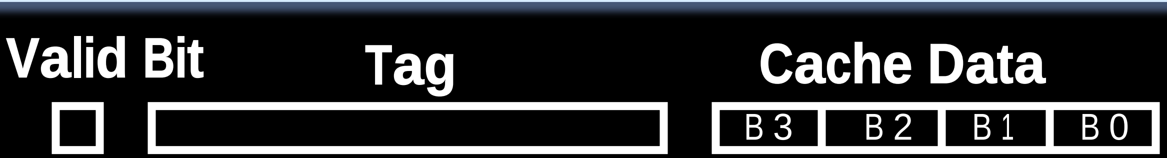
What to do on a write hit?

- **Write-through**
  - Update both cache and memory
- **Write-back**
  - update word in cache block
  - allow memory word to be "stale"
  - add 'dirty' bit to block
    - memory & Cache inconsistent
    - needs to be updated when block is replaced
  - …OS flushes cache before I/O…
- **Performance trade-offs?**

Block Size Tradeoff

- **Benefits of Larger Block Size**
  - Spatial Locality: if we access a given word, we're likely to access other nearby words soon
  - Very applicable with Stored-Program Concept
  - Works well for sequential array accesses
- **Drawbacks of Larger Block Size**
  - Larger block size means larger miss penalty
    - on a miss, takes longer time to load a new block from next level
  - If block size is too big relative to cache size, then there are too few blocks
    - Result: miss rate goes up

Extreme Example: One Big Block



Valid Bit

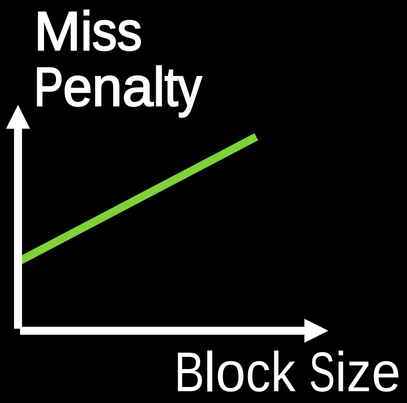
Tag

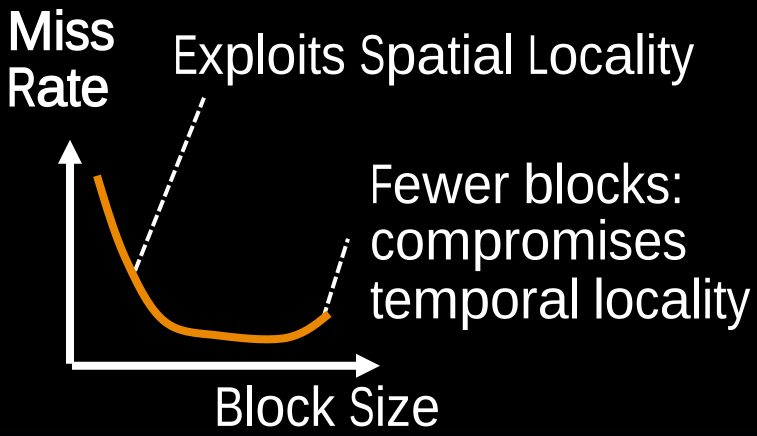
Cache Data

B 3 | B 2 | B 1 | B 0

- Cache Size = 4 bytes Block Size = 4 bytes
  - Only ONE entry (row) in the cache!
- If item accessed, likely accessed again soon
  - But unlikely will be accessed again immediately!
- The next access will likely to be a miss again
  - Continually loading data into the cache but discard data (force out) before use it again
  - Nightmare for cache designer: Ping Pong Effect

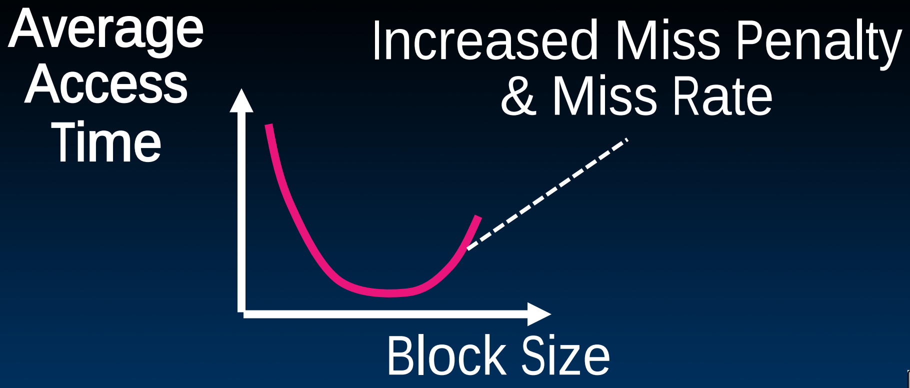
Block Size Tradeoff Conclusions





Exploits Spatial Locality

Fewer blocks: compromises temporal locality



Increased Miss Penalty & Miss Rate

Types of Cache Misses (1/2)

- **“Three Cs” Model of Misses**
- **1st C: Compulsory Misses**
  - occur when a program is first started
  - cache does not contain any of that program’s data yet, so misses are bound to occur
  - can’t be avoided easily, so won’t focus on these in this course
  - Every block of memory will have one compulsory miss (NOT only every block of the cache)

Types of Cache Misses (2/2)

- 2nd C: **Conflict Misses**
  - miss that occurs because two distinct memory addresses map to the same cache location
  - two blocks (which happen to map to the same location) can keep overwriting each other
  - big problem in direct-mapped caches
  - how do we lessen the effect of these?
- **Dealing with Conflict Misses**
  - Solution 1: Make the cache size bigger
    - Fails at some point
  - Solution 2: Multiple distinct blocks can fit in the same cache Index?

☑

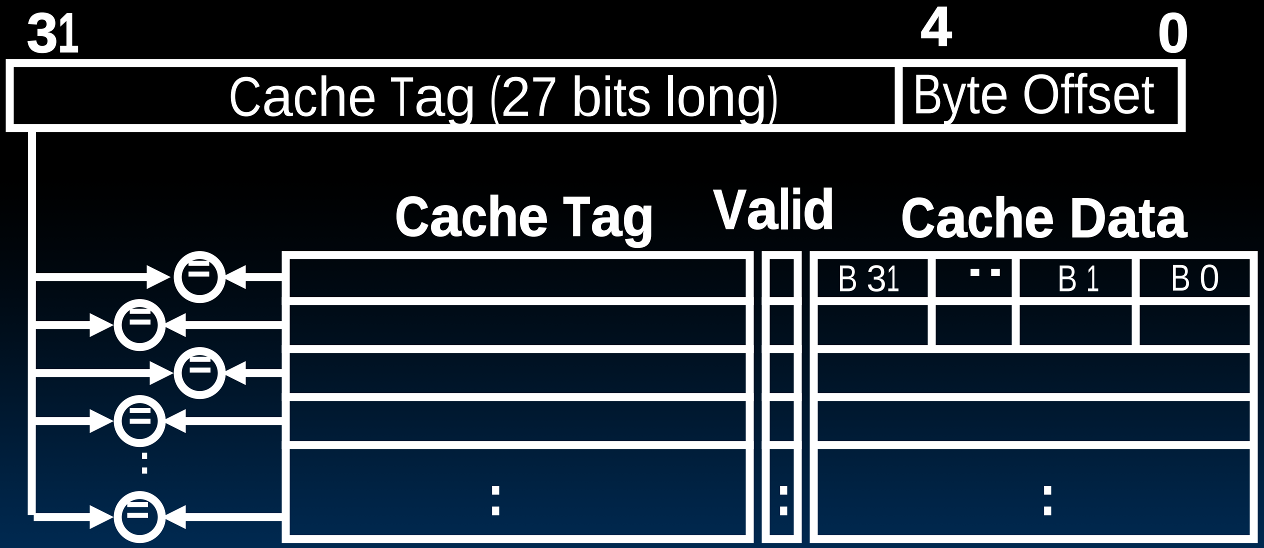
Fully Associative Caches

Fully Associative Cache (1/3)

- **Memory address fields:**
  - Tag: same as before
  - Offset: same as before
  - Index: non-existant
- **What does this mean?**
  - no “rows“: any block can go anywhere in the cache
  - must compare with all tags in entire cache to see if data is there

Fully Associative Cache (2/3)

- **Fully Associative Cache (e.g., 32 B block)**
  - compare tags in parallel



Fully Associative Cache (3/3)

- **Benefit of Fully Assoc Cache**
  - No Conflict Misses (since data can go anywhere)
- **Drawbacks of Fully Assoc Cache**
  - Need hardware comparator for every single entry: if we have a 64KB of data in cache with 4B entries, we need 16K comparators: infeasible

Final Type of Cache Miss

- 3rd C: **Capacity Misses**
  - miss that occurs because the cache has a limited size
  - miss that would not occur if we increase the size of the cache
  - sketchy definition, so just get the general idea
- **This is the primary type of miss for Fully Associative caches.**

How to categorize misses

- **Run an address trace against a set of caches:**
  - First, consider an infinite-size, fully-associative cache. For every miss that occurs now, consider it a compulsory miss.
  - Next, consider a finite-sized cache (of the size you want to examine) with full-associativity. Every miss that is not in #1 is a capacity miss.
  - Finally, consider a finite-size cache with finite-associativity. All of the remaining misses that are not #1 or #2 are conflict misses.
  - (Thanks to Prof. Kubiatowicz for the algorithm)

When poll is active, respond at **pollev.com/ddg**

Text **DDG** to **22333** once to join

L25 For a given cache size: a larger block size can cause a lower hit rate than a smaller one

You can often make your code run faster if you know your computer's cache size

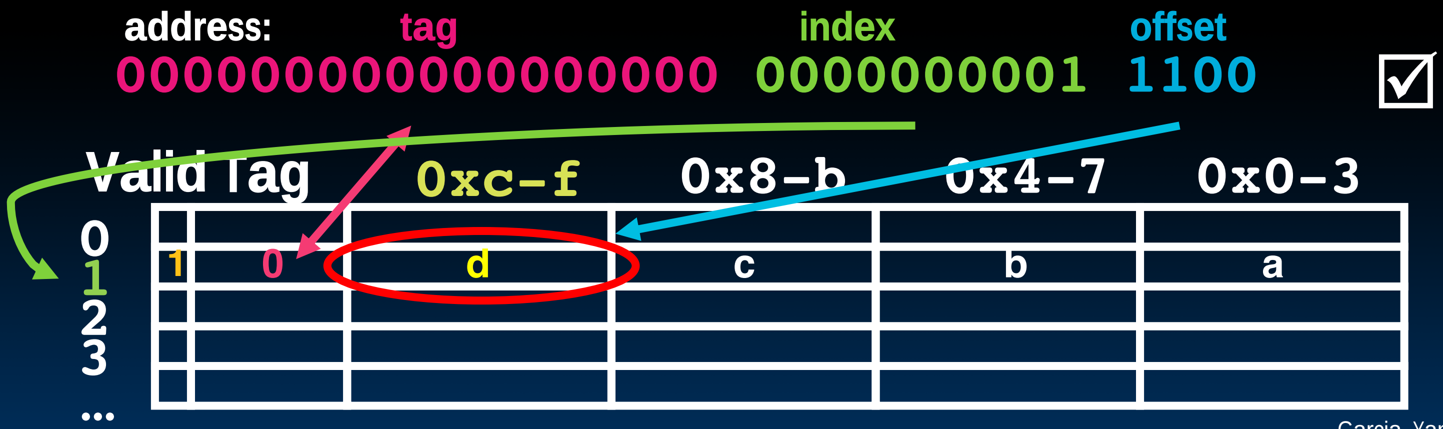
Spatial locality means keep the most recent data closer to the processor

Powered by Poll Everywhere

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

And in Conclusion…

1. **Divide into T I O bits, Go to Index = I, check valid**
    1. If 0, load block, set valid and tag (COMPULSORY MISS) and use offset to return the right chunk (1,2,4-bytes)
    2. If 1, check tag
        1. If Match (HIT), use offset to return the right chunk
        2. If not (CONFLICT MISS), load block, set valid and tag, use offset to return the right chunk



address: tag 0000000000000000000 index 0000000001 offset 1100 ☑

	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0						
1	1	0	d	c	b	a
2						
3						
…						