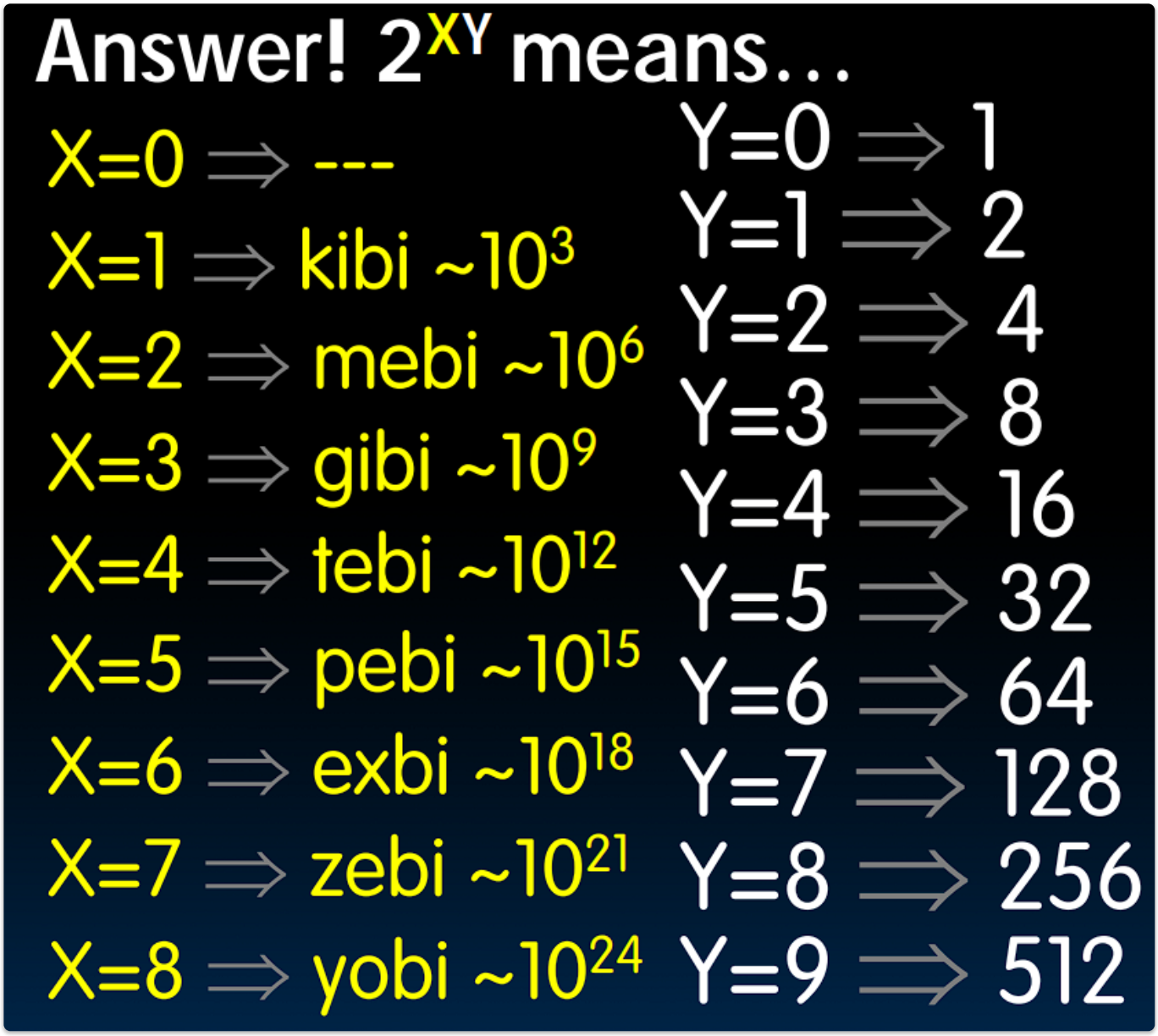
Binary Prefix

Name	Abbr	Factor
kibi	Ki	$2^{10} = 1,024$
mebi	Mi	$2^{20} = 1,048,576$
gibi	Gi	$2^{30} = 1,073,741,824$
tebi	Ti	$2^{40} = 1,099,511,627,776$
pebi	Pi	$2^{50} = 1,125,899,906,842,624$
exbi	Ei	$2^{60} = 1,152,921,504,606,846,976$
zebi	Zi	$2^{70} = 1,180,591,620,717,411,303,424$
yobi	Yi	$2^{80} = 1,208,925,819,614,629,174,706,176$

Kissing me gives ten percent extra zeal & youth!

The way to remember numbers



Memory Hierarchy Basis

- Temporal locality: If we use it now, chances are we'll want to use it again soon.

> *Keep most recently accessed data items closer to the processor.*

- Spatial locality: If we use a piece of memory, chances are we'll use the neighboring pieces soon.

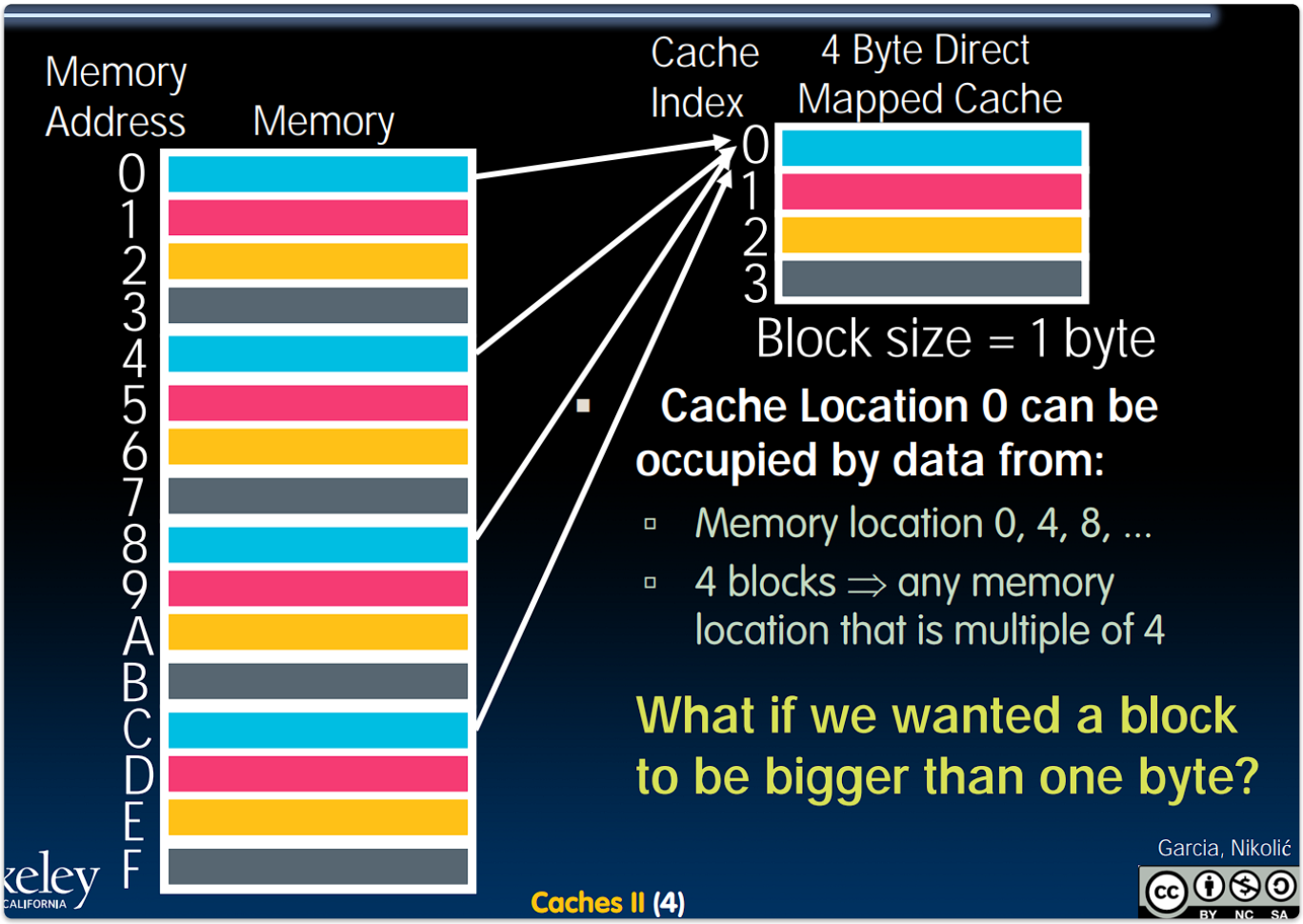
> *Move blocks consisting of contiguous words closer to the processor.*

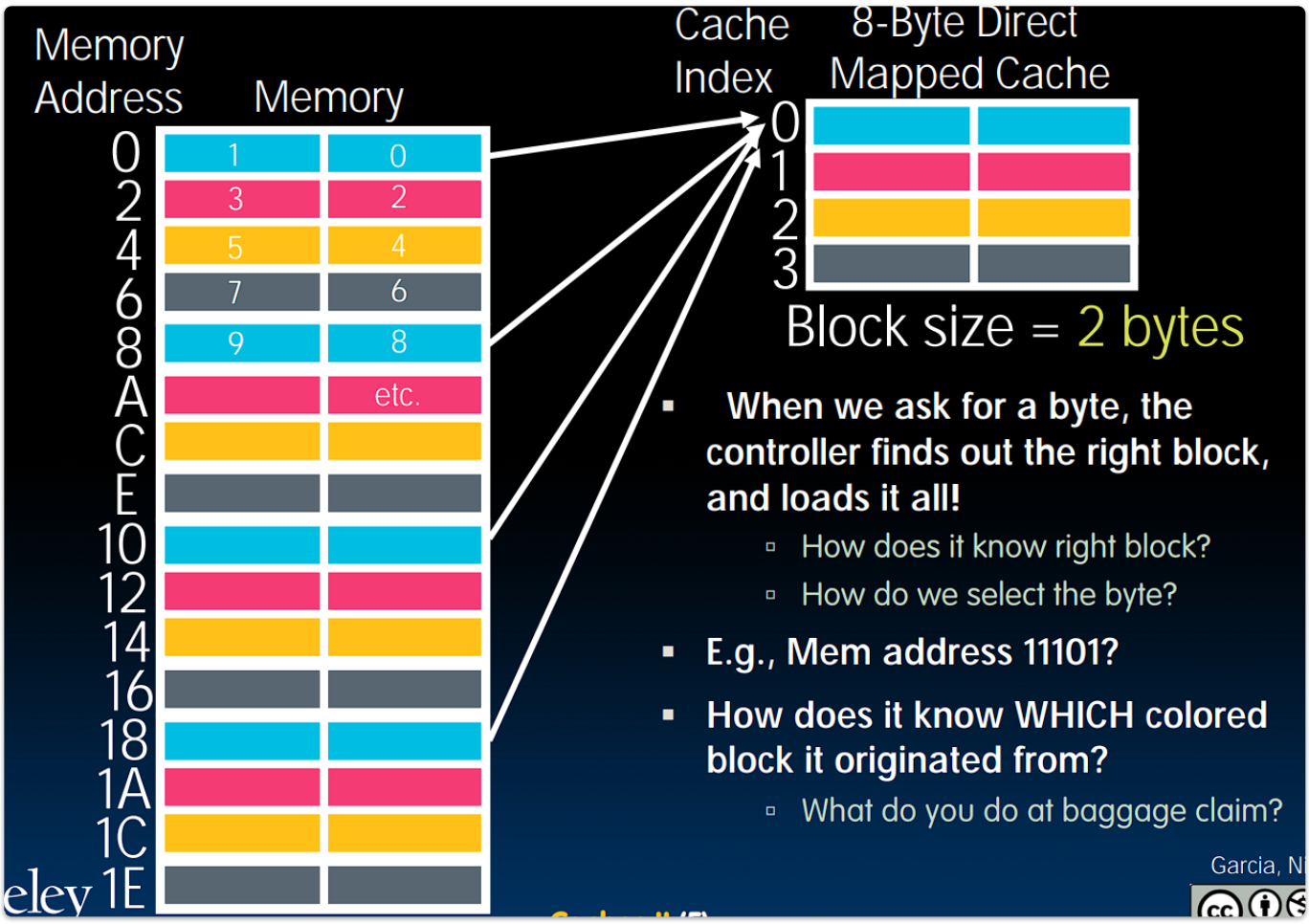
Cache

Caches provide an illusion to the processor that the memory is infinitely large and infinitely fast

Direct Mapped Caches

In a direct-mapped cache, each memory address is associated with one possible block within the cache


Memory Address
Memory
0
1
2
3
4
5
6
7
8
9
A
B
C
D
E
F
Cache Index
4 Byte Direct Mapped Cache
0
1
2
3
Block size = 1 byte
Cache Location 0 can be occupied by data from:
Memory location 0, 4, 8, ...
4 blocks ⇒ any memory location that is multiple of 4
What if we wanted a block to be bigger than one byte?
Garcia, Nikolić
Caches II (4)


Memory Address
Memory
0
2
4
6
8
A
C
E
10
12
14
16
18
1A
1C
1E
1
0
3
2
5
4
7
6
9
8
etc.
Cache Index
8-Byte Direct Mapped Cache
0
1
2
3
Block size = 2 bytes
When we ask for a byte, the controller finds out the right block, and loads it all!
How does it know right block?
How do we select the byte?
E.g., Mem address 11101?
How does it know WHICH colored block it originated from?
What do you do at baggage claim?
Garcia, N

tag to check if have correct block

index to select block

byte offset within block

- All fields are read as unsigned integers.
- Index
  - specifies the cache index (which "row"/block of the cache we should look in)
- Offset
  - once we've found correct block, specifies which byte within the block we want
- Tag
  - the remaining bits after offset and index are determined; these are used to distinguish between all the memory addresses that map to the same location

One More Detail: Valid Bit

- When start a new program, cache does not have valid information for this program
- Need an indicator whether this tag entry is valid for this program
- Add a "valid bit" to the cache tag entry

  0 → cache miss, even if by chance, address = tag

  1 → cache hit, if processor address = tag

Writes, Block, Sizes, Misses

Write

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 and cache inconsistent.
  - needs to be updated when block is replaced.
- Operating system flushes cache before I/O.

(No) Write allocate

- Write-allocate
  - On a write miss, you bring the line into the cache and then update the line
- No write-allocate
  - On a write miss, don't bring the line into the cache, you only update memory
- For both, you always bring the line in on a read miss

Replacement Policies

- **LRU** (Least Recently Used) — When we decide to evict a cache block to make space, we select the block that has been used the furthest back in time of all the blocks.
- **Random** - When we decide to evict a cache block to make space, we randomly select one of the blocks in the cache to evict.
- **MRU** (Most Recently Used) — When we decide to evict a cache block to make space, we select the block that has been used the most recently of all the blocks.

Block Size Tradeoff

Cache line/block

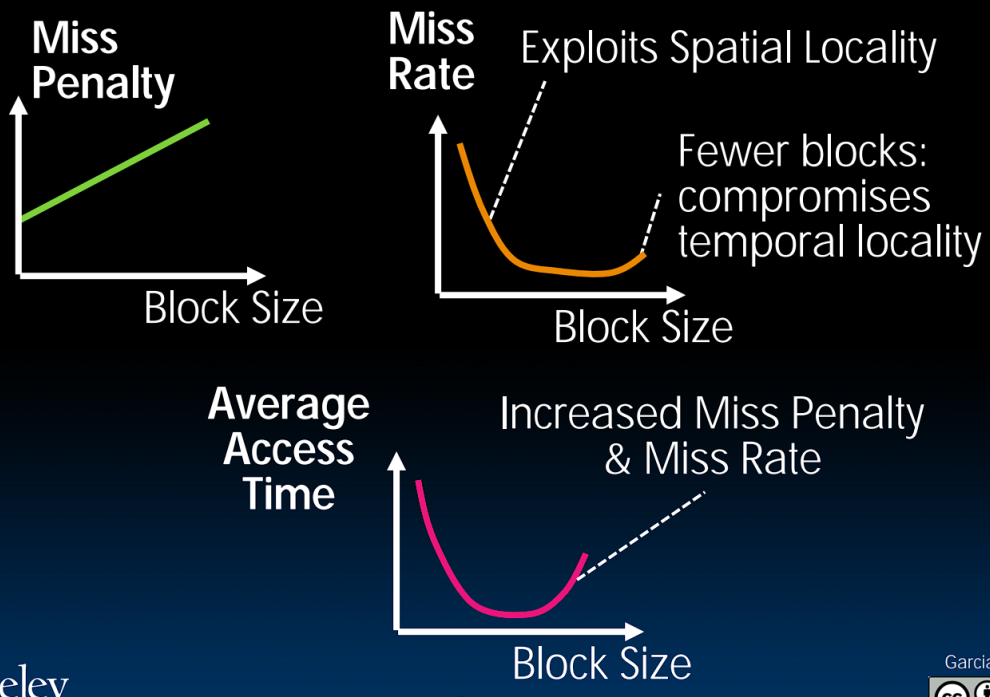
- The smallest unit of memory that can be transferred between the main memory and the cache
- Each line has its own entry in the cache

- **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

- 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



Berkeley UNIVERSITY OF CALIFORNIA

Caches III (31)

Garcia, Niko

Types of Cache Missed

Compulsory(forced) 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.
- **Every block of memory will have one compulsory miss (NOT only every block of the cache)**

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

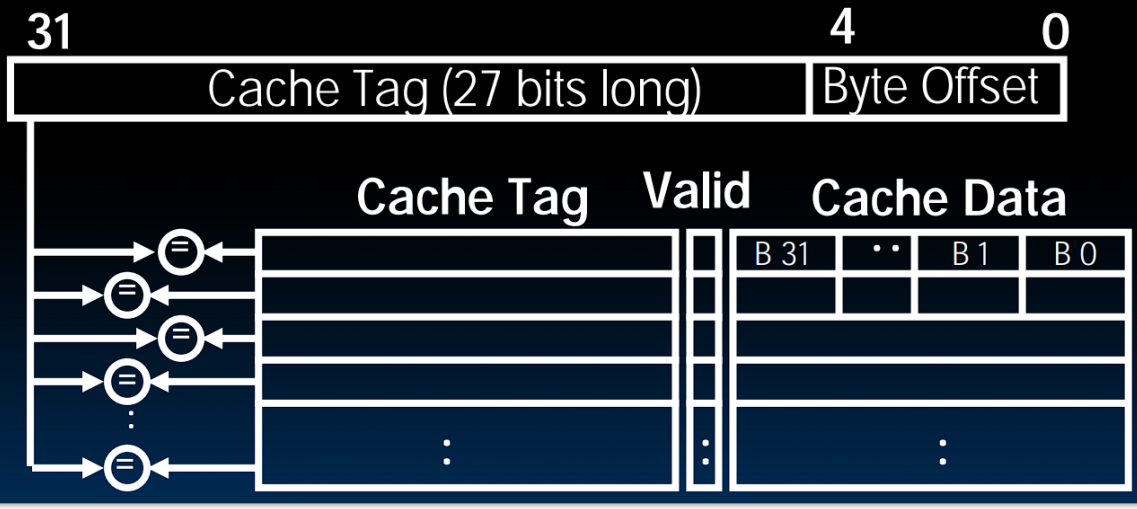
- Solution 1: Make the cache size bigger
  - Fails at some point
- Solution 2: Multiple distinct blocks can fit in the same cache Index?

Fully Associative Caches

- **Memory address fields:**
  - Tag: same as before
  - Offset: same as before
  - Index: non-existant

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



- **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

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

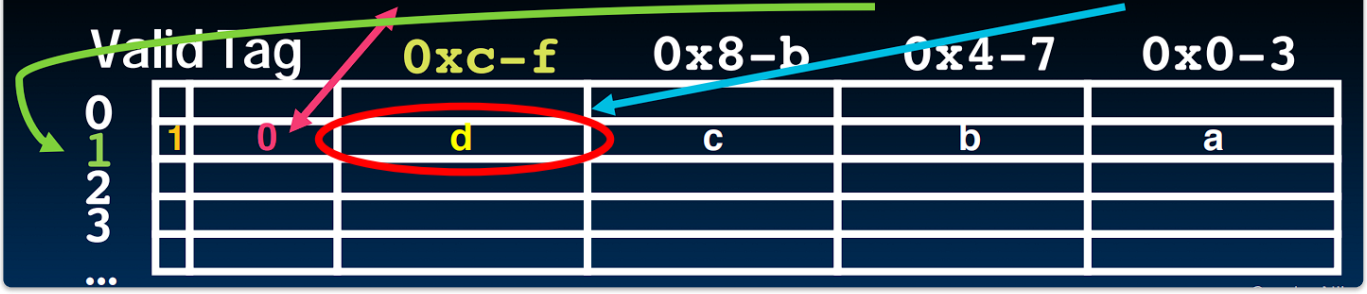
This is the primary type of miss for Fully Associative caches.

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 index offset

0000000000000000 0000000001 1100 ☑

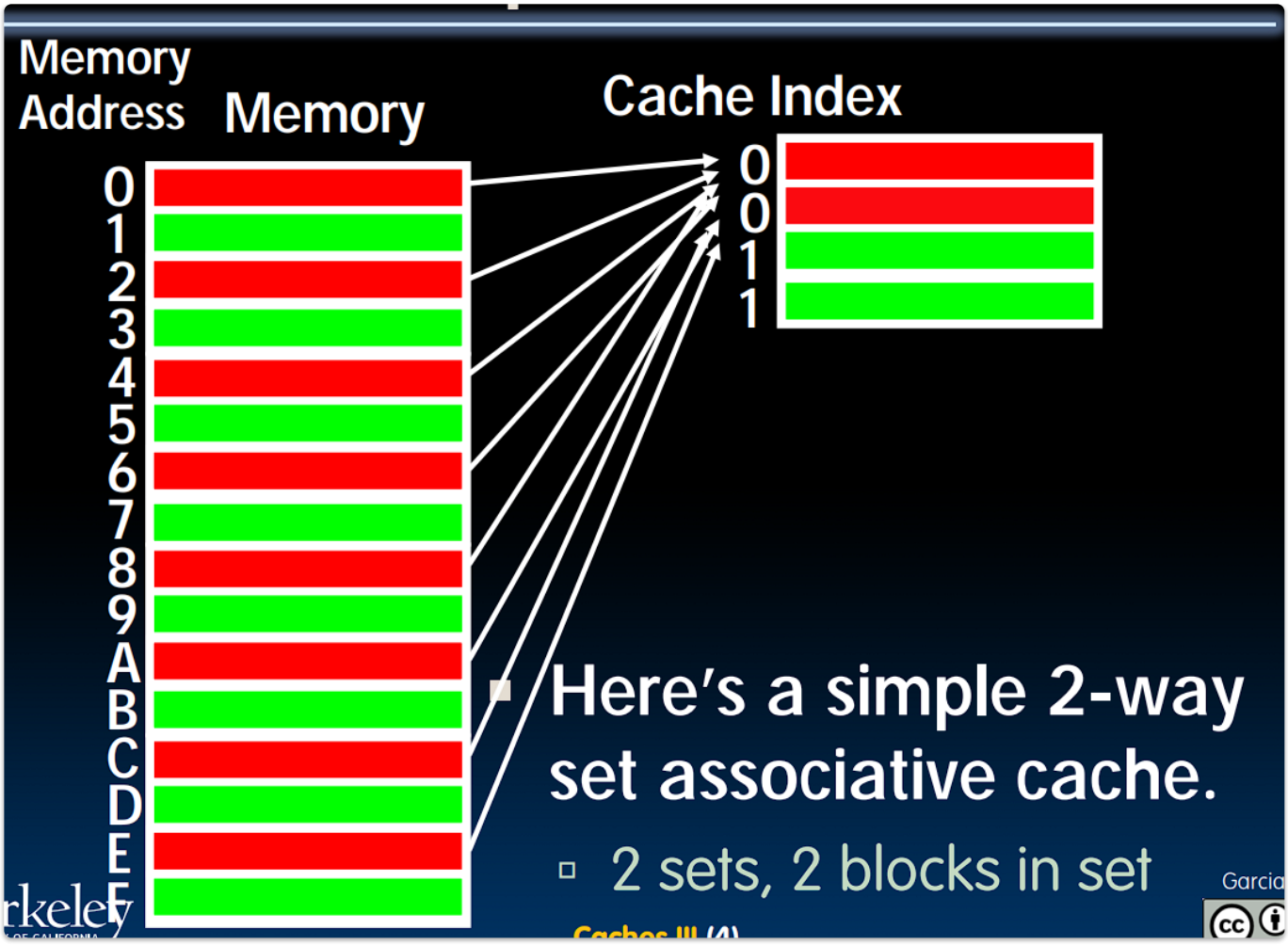


Set-Associative Cache

- **Memory address fields:**
  - Tag: same as before
  - Offset: same as before
  - Index: points us to the correct "row" (called a set in this case)

So what's the difference?

- each set contains multiple blocks
- once we've found correct set, must compare with all tags in that set to find our data
- Size of $ is # sets x N blocks/set x block size



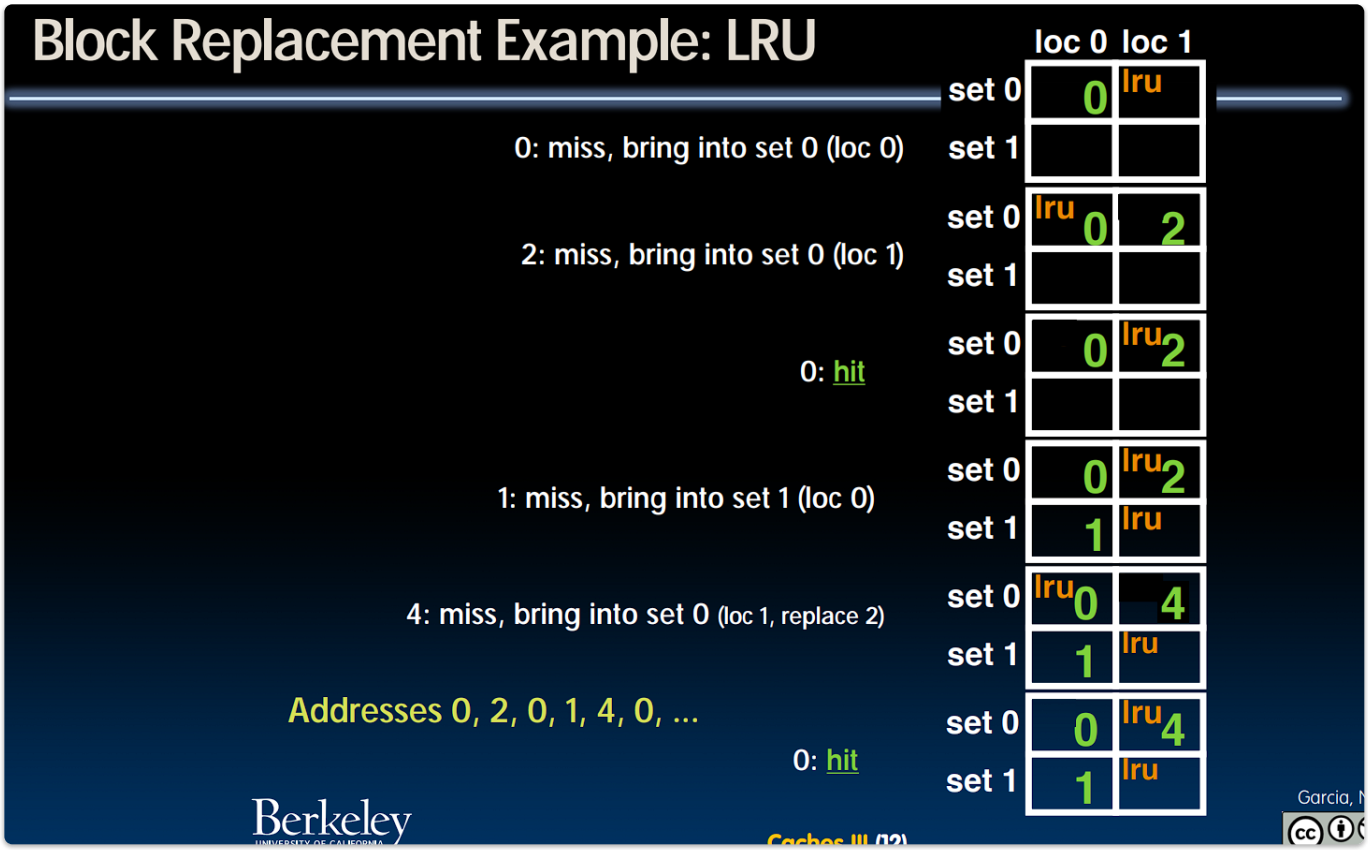
- **Basic Idea**
  - cache is direct-mapped w/respect to sets
  - each set is fully associative with N blocks in it
- **Given memory address:**
  - Find correct set using Index value.
  - Compare Tag with all Tag values in that set.
  - If a match occurs, hit!, otherwise a miss.
  - Finally, use the offset field as usual to find the desired data within the block.

In fact, for a cache with M blocks,

- it's Direct-Mapped if it's 1-way set assoc
- it's Fully Assoc if it's M-way set assoc
- so these two are just special cases of the more general set associative design

Block Replacement Policy

- **LRU (Least Recently Used)**
  - Idea: cache out block which has been accessed (read or write) least recently
  - Pro: temporal locality ➔ recent past use implies likely future use: in fact, this is a very effective policy
  - Con: with 2-way set assoc, easy to keep track (one LRU bit); with 4-way or greater, requires complicated hardware and much time to keep track of this
- **FIFO**
  - Idea: ignores accesses, just tracks initial order
- **Random**
  - If low temporal locality of workload, works ok

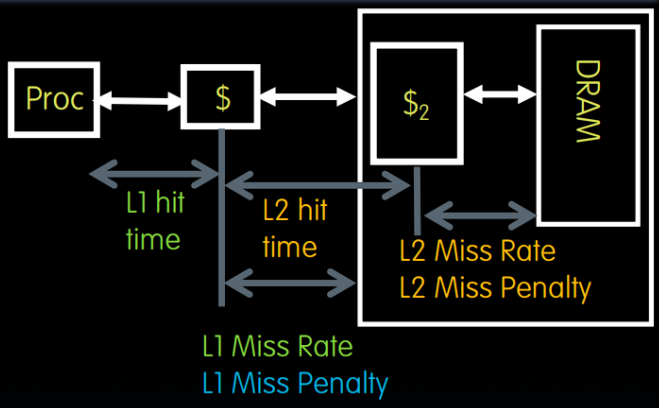


Average Memory Access Time

Big Idea

- How to choose between associativity, block size, replacement & write policy?
- Design against a performance model
  - Minimize: Average Memory Access Time
    = Hit Time
    + Miss Penalty x Miss Rate
  - influenced by technology & program behavior
- Create the illusion of a memory that is large, cheap, and fast - on average
- How can we improve miss penalty?

Analyzing Multi-level cache hierarchy



Avg Mem Access Time =
L1 Hit Time + L1 Miss Rate * L1 Miss Penalty

L1 Miss Penalty =
L2 Hit Time + L2 Miss Rate * L2 Miss Penalty

Avg Mem Access Time =
L1 Hit Time + L1 Miss Rate *
(L2 Hit Time + L2 Miss Rate * L2 Miss Penalty)

- Assume
  - L1 Hit Time = 1 cycle
  - L1 Miss rate = 5%
  - L2 Hit Time = 5 cycles
  - L2 Miss rate = 15% (% L1 misses that miss)
  - L2 Miss Penalty = 200 cycles
- L1 miss penalty = 5 + 0.15 * 200 = 35
- Avg mem access time = 1 + 0.05 x 35
  = 2.75 cycles

Example: without L2 cache

- Assume
  - L1 Hit Time = 1 cycle
  - L1 Miss rate = 5%
  - L1 Miss Penalty = 200 cycles
- Avg mem access time = 1 + 0.05 x 200
  = 11 cycles
- 4x faster with L2 cache! (2.75 vs. 11)