Optimizing Cache Performance

Today's Lecture

- Use larger cache blocks to take advantage of spatial locality
- Use set associativity to resolve "hashing collisions"

For a byte-addressable machine with 16-bit addresses

Which picture best represents a cache that is direct-mapped, each block holds one byte, and has eight cache blocks

16 − 3 = 13

A

Block Index	2-bit Tag	8-bit data
000		
001		
010		
011		
100		
101		
110		
111		

B

Block Index	8-bit Tag	8-bit data
000		
001		
010		
011		
100		
101		
110		
111		

C

Block Index	13-bit Tag	8-bit data
000		
001		
010		
011		
100		
101		
110		
111		

3 bits

For a byte-addressable machine with 16-bit addresses

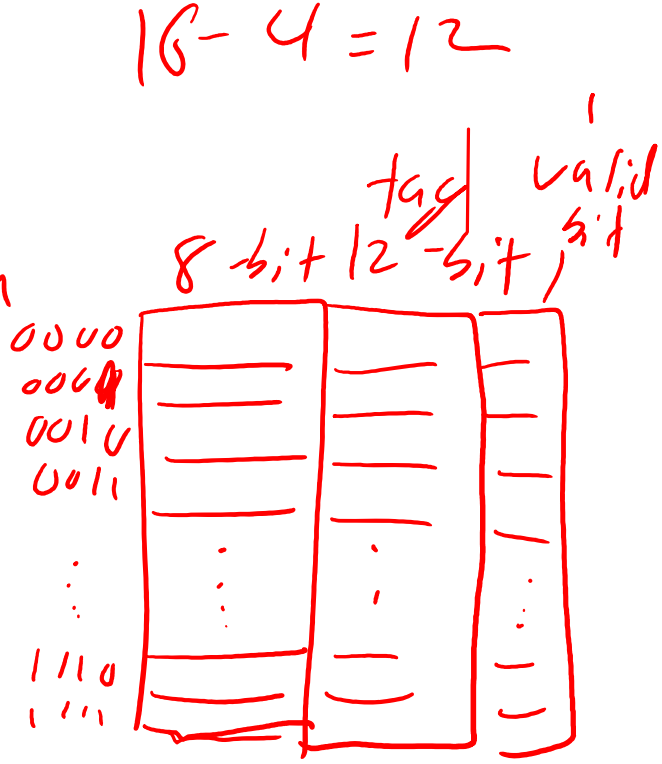
A cache has the following characteristics:

- It is direct-mapped (as discussed last time)
- Each block holds one byte
- The cache index is the four least significant bits

Two questions:

- How many blocks does the cache hold?

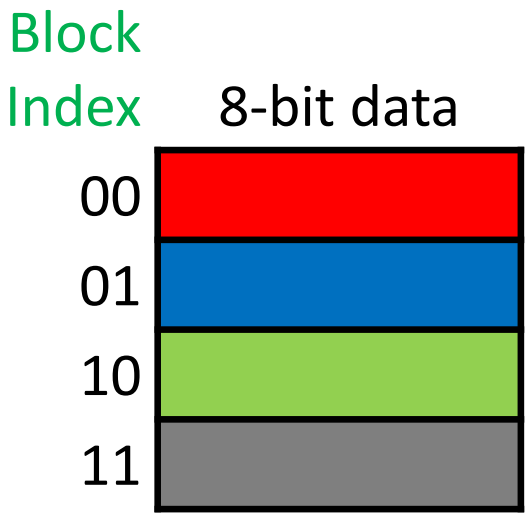
a) 1
b) 2
c) 4
d) 8
e) 16



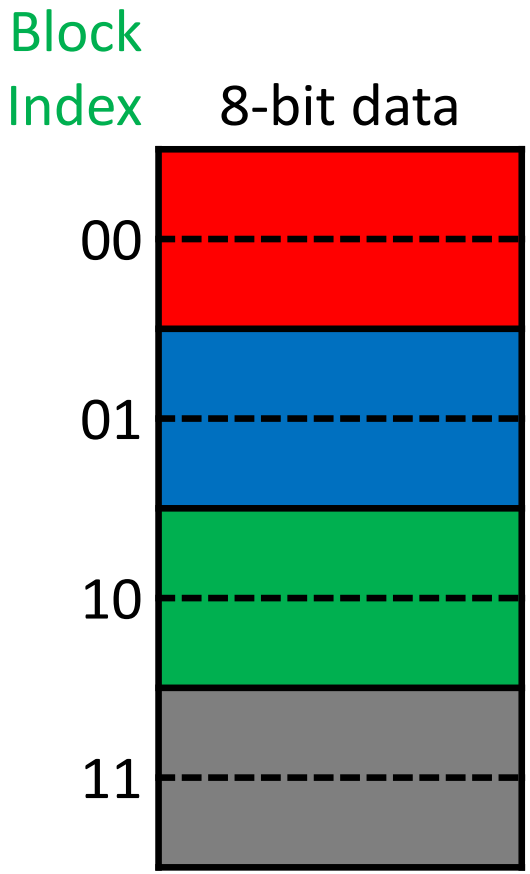
- How many bits are stored at each cache block (*e.g.*, for the data array, tags, etc.)?

a) 8 b) 9 c) 12 d) 20 e) 21

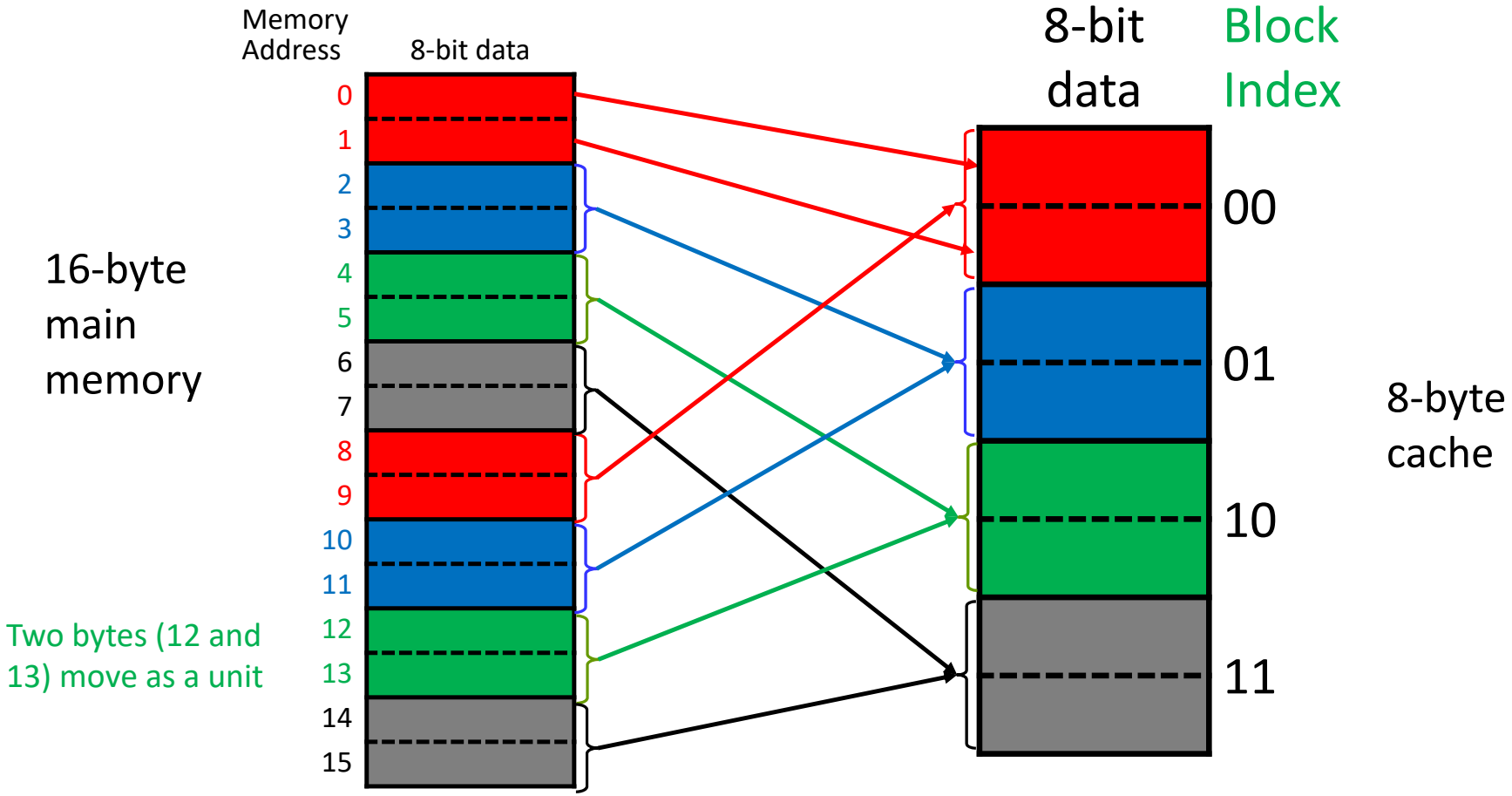
1-byte cache blocks do not take advantage of spatial locality



Create larger cache blocks

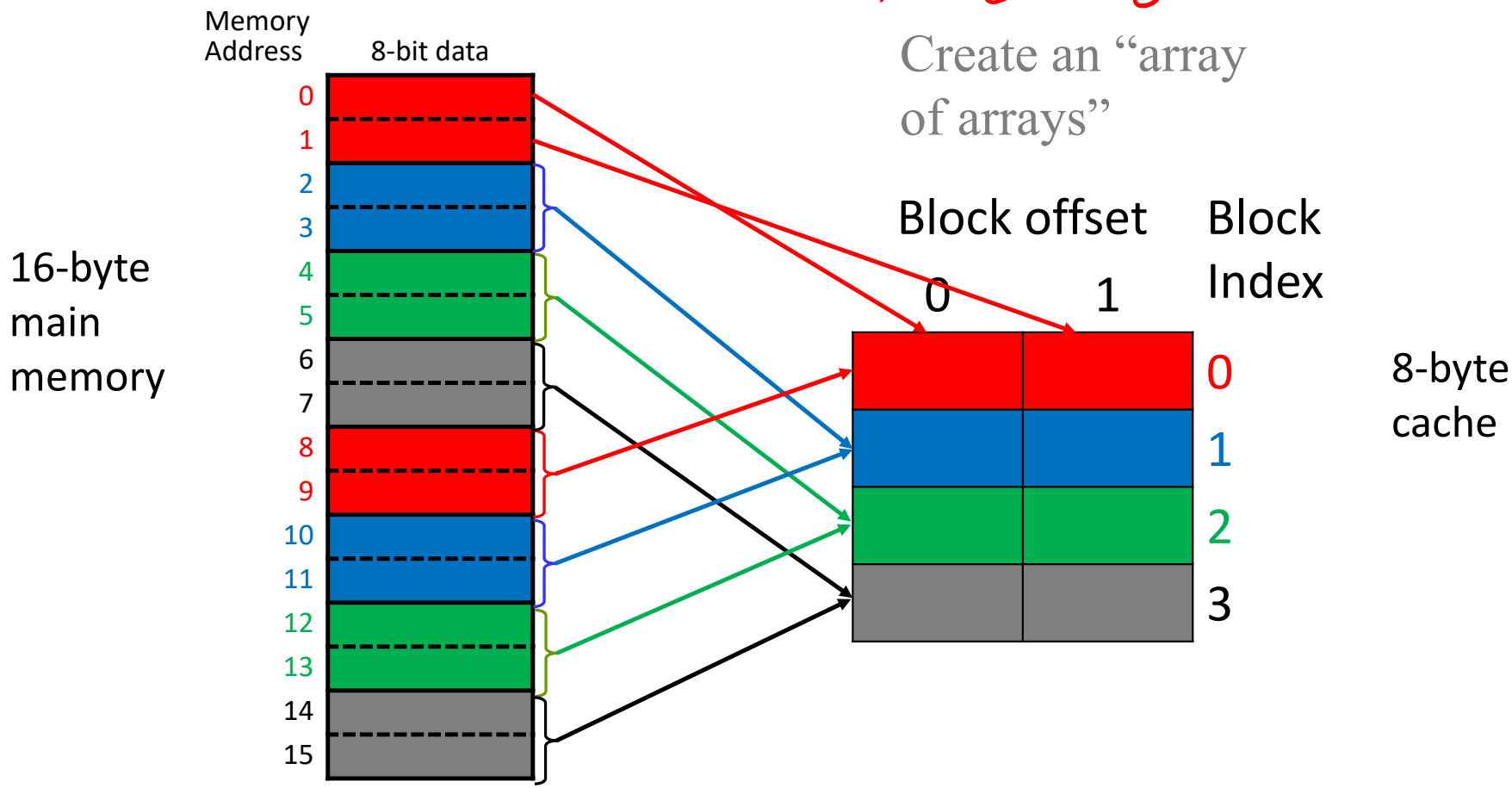


Creating larger cache blocks moves adjacent blocks as a unit

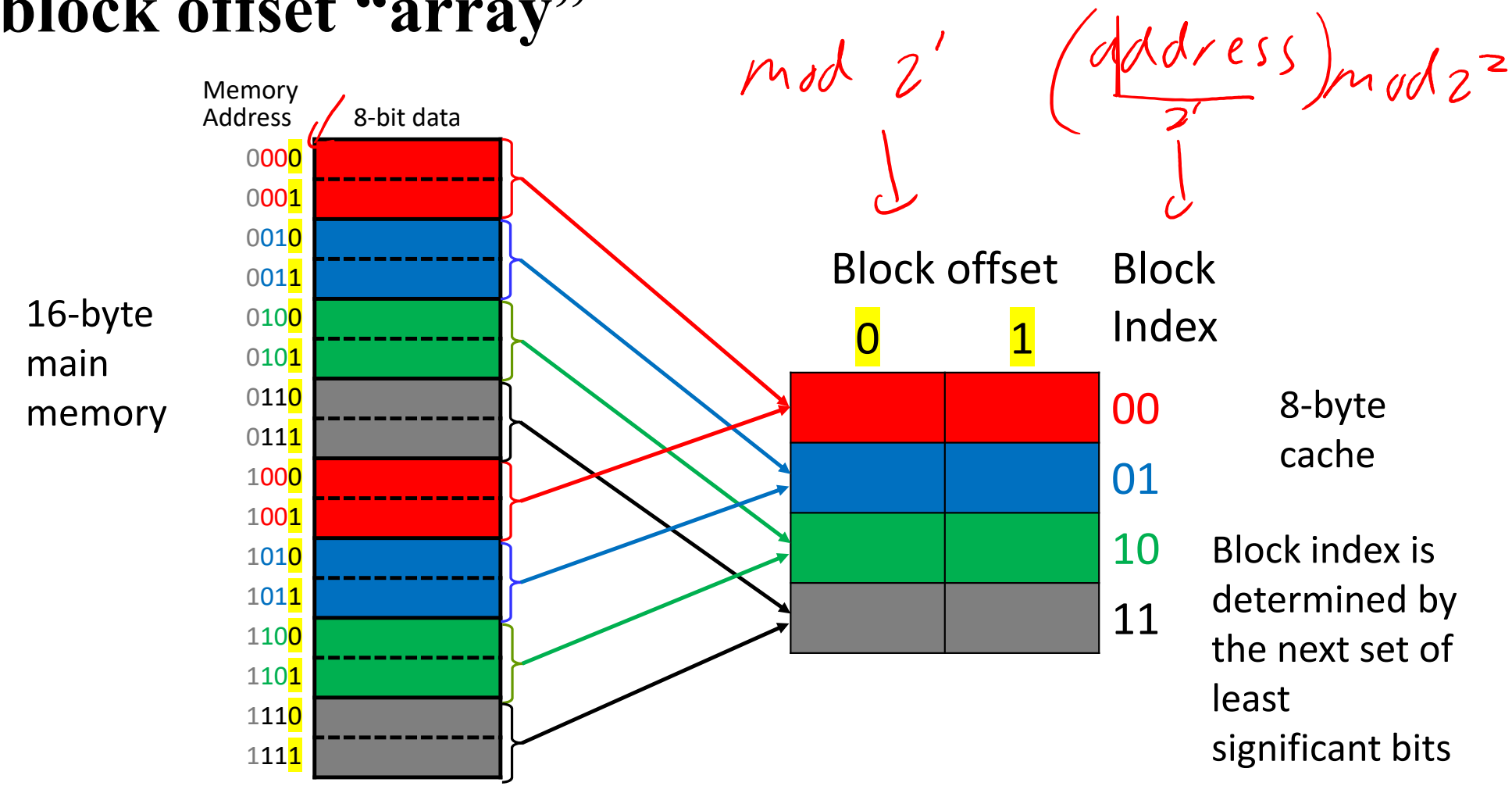


Each cache block is an array indexed by a block offset

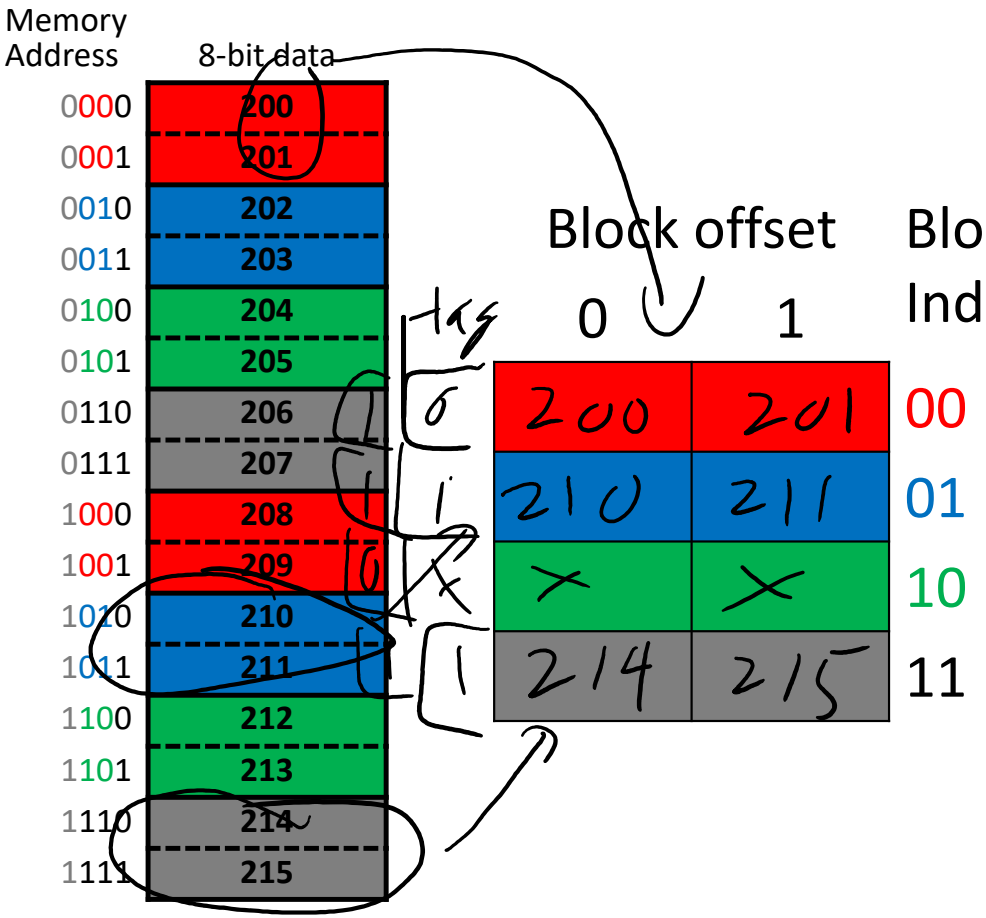
M[bi][bo]



The least-significant bit(s) is used to index the block offset "array"



Data gets moved in cache blocks and not bytes



Consider the following set of memory loads

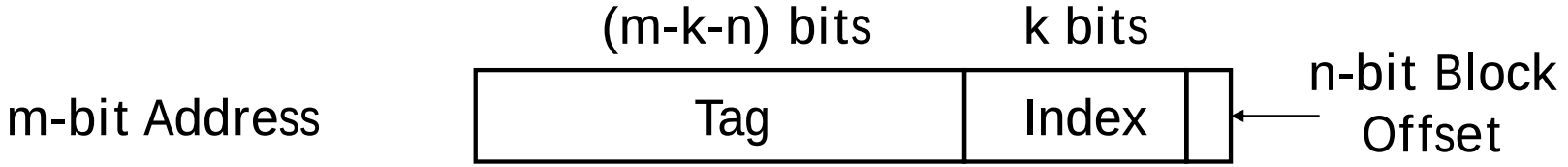
```
load 0
load 15
load 10
load 1
```

What is stored in block index 01, offset 1?

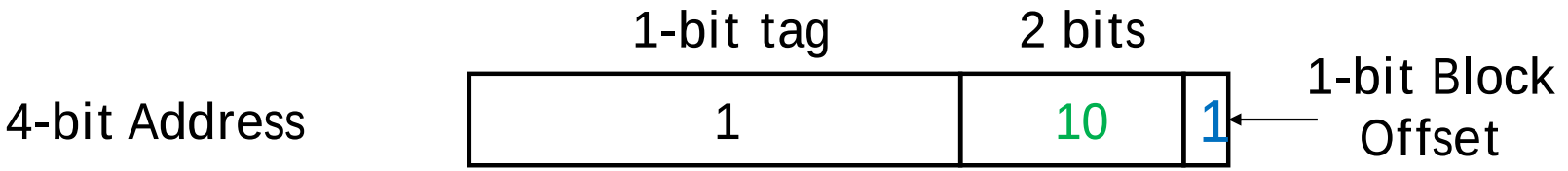
a) 201
b) 210
c) 211
d) 215
e) invalid

To increase cache block size, subdivide an m-bit address into tag, index, and block offset

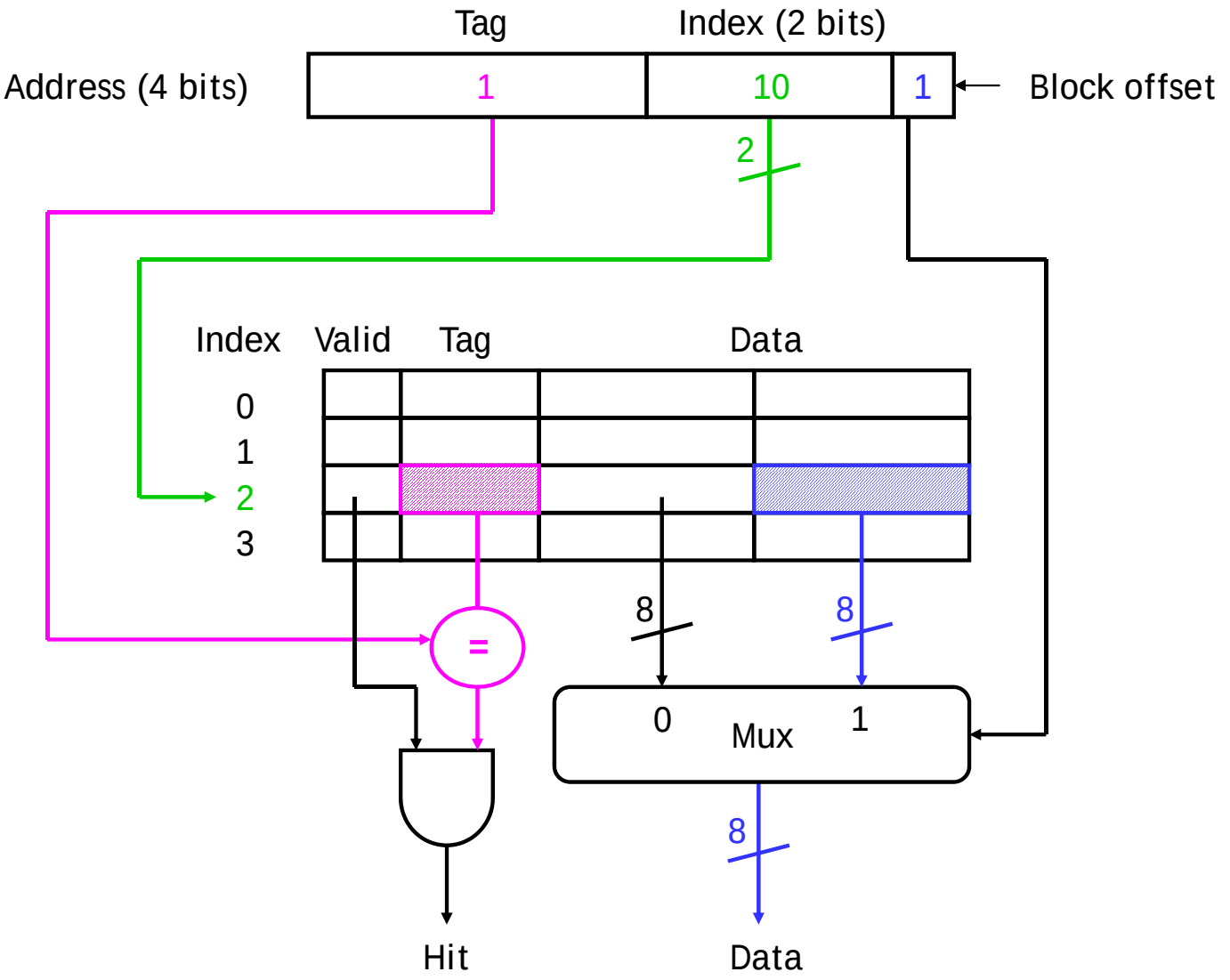
- Suppose we have a cache with 2^k blocks, each containing 2^n bytes.
  - Lowest *n* bits are the block offset that decides which of the 2^n bytes in the cache block will store the data.
  - Next *k* bits of the address select one of the 2^k cache blocks.

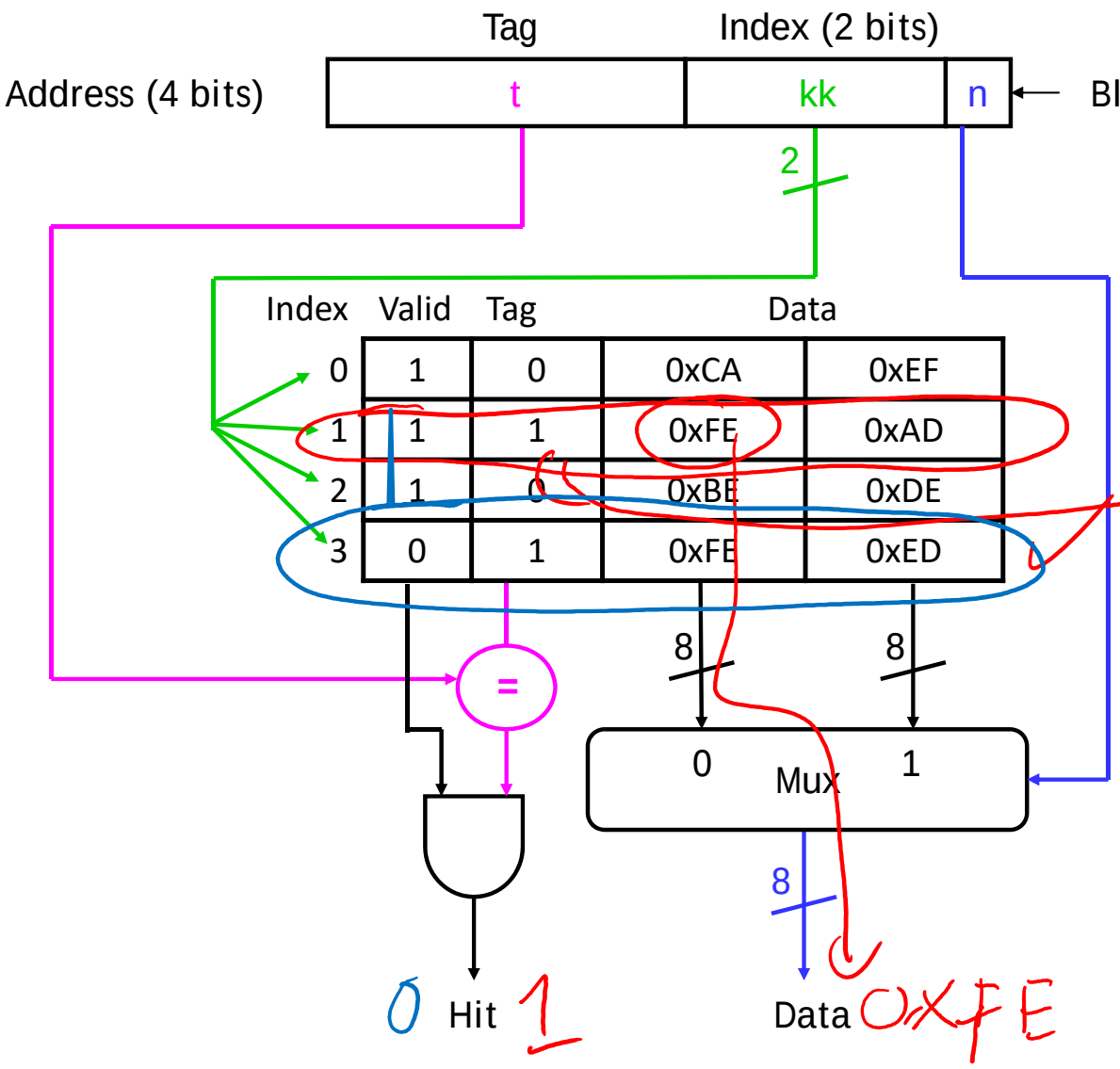


- Example: 2^2-block cache with 2^1 bytes per block. Memory address 13 (1101) would be stored in offset 1 of cache block 2.



Implement block offset with a multiplexer





For the addresses below, what byte is read from the cache (or is there a miss)?

- 1010
- 1110
- 0001
- 1101

a) Miss, tag mismatch
b) Miss, invalid
c) Hit, 0xDE
d) Hit, 0xEF
e) Hit, 0xFE

For a byte-addressable machine with 16-bit addresses

A cache has the following characteristics:

- It is direct-mapped
- Each block holds four bytes
- The cache has 32 cache blocks

2 bits offset

5 bits offset

0

How many bits are used for the tag?

a) 2
b) 4
c) 5
d) 7
e) 9

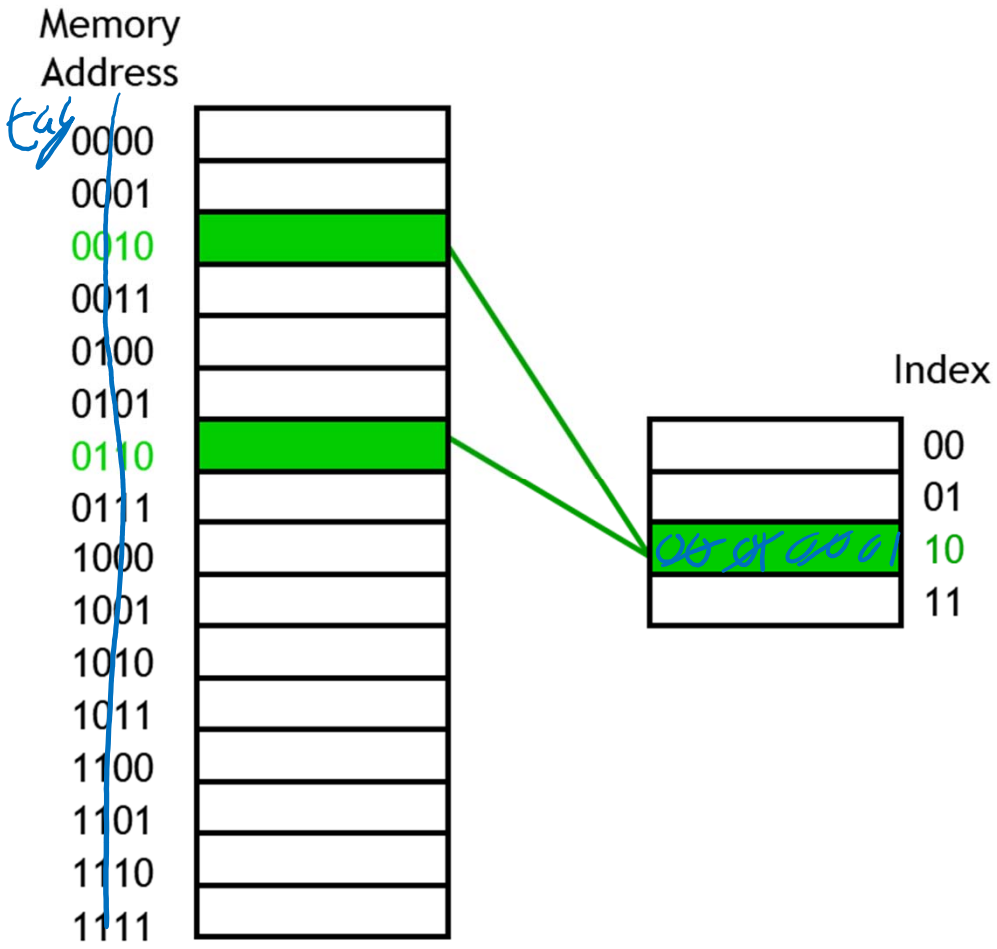
16 - 5 - 2 = 9

Direct-mapped caches fall short when addresses collide

- Example: what happens if a program uses addresses 2, 6, 2, 6, 2, …?

M M M M

thrashing

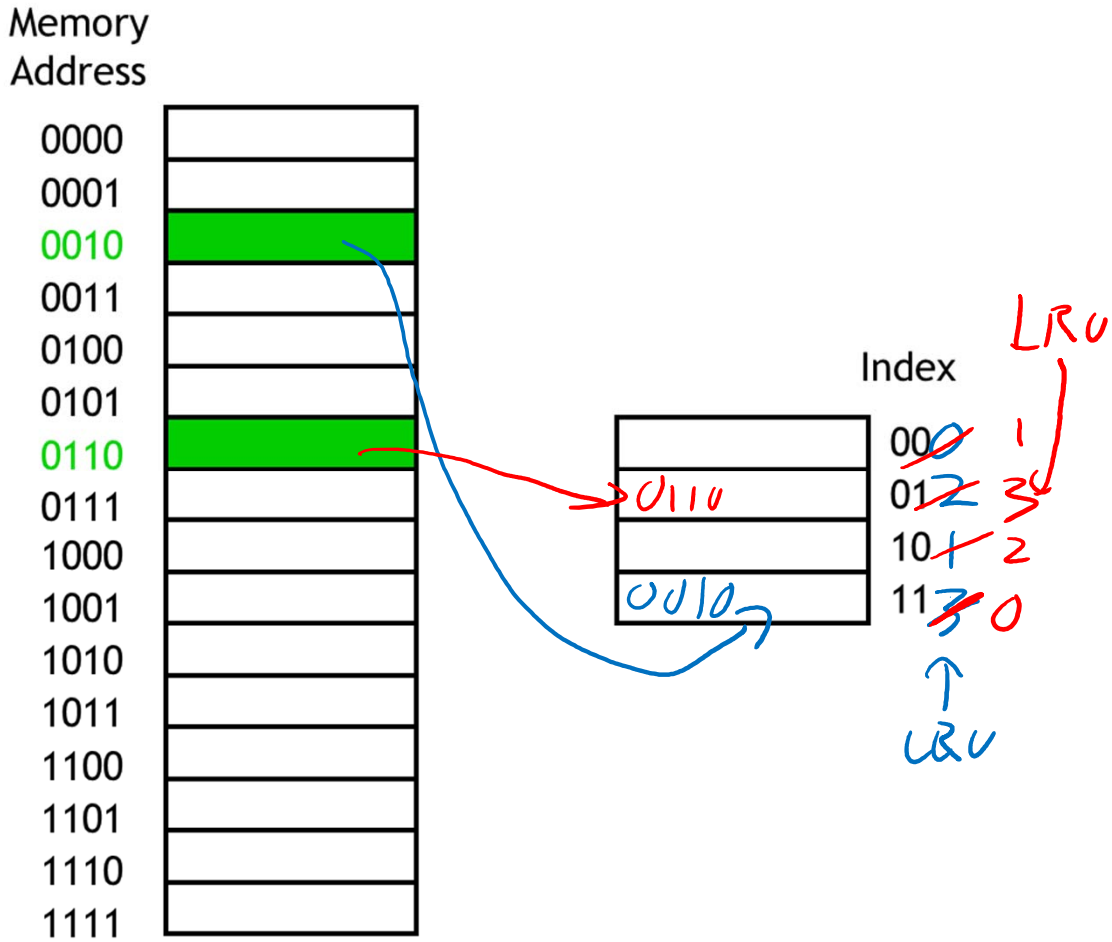


A fully-associative cache permits data to be stored in *any* cache block

- When data is fetched from memory, it can be placed in *any* unused block of the cache.
- Maximize temporal locality by keeping the most recently used data in the cache, replace the least-recently used (LRU) when cache is full

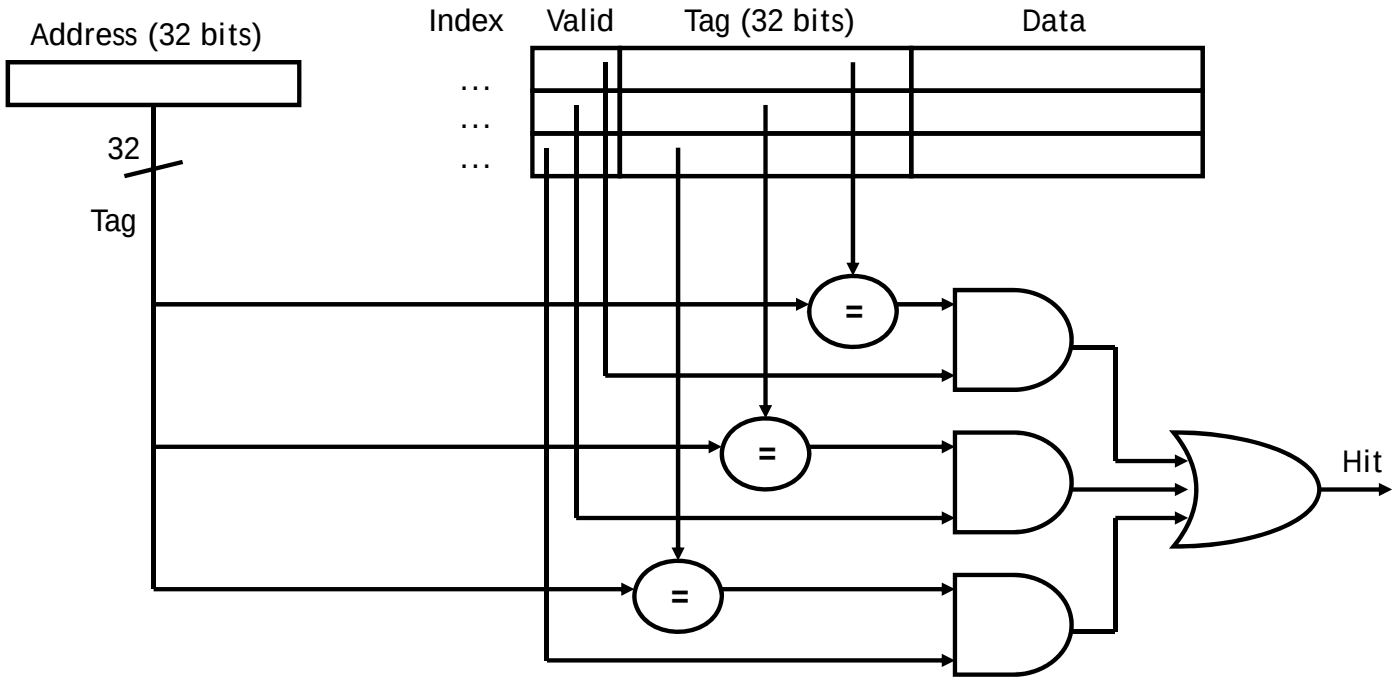
A fully associative cache can map addresses anywhere, eliminating thrashing

- Example: what happens if a program uses addresses 2, 6, 2, 6, 2, …?



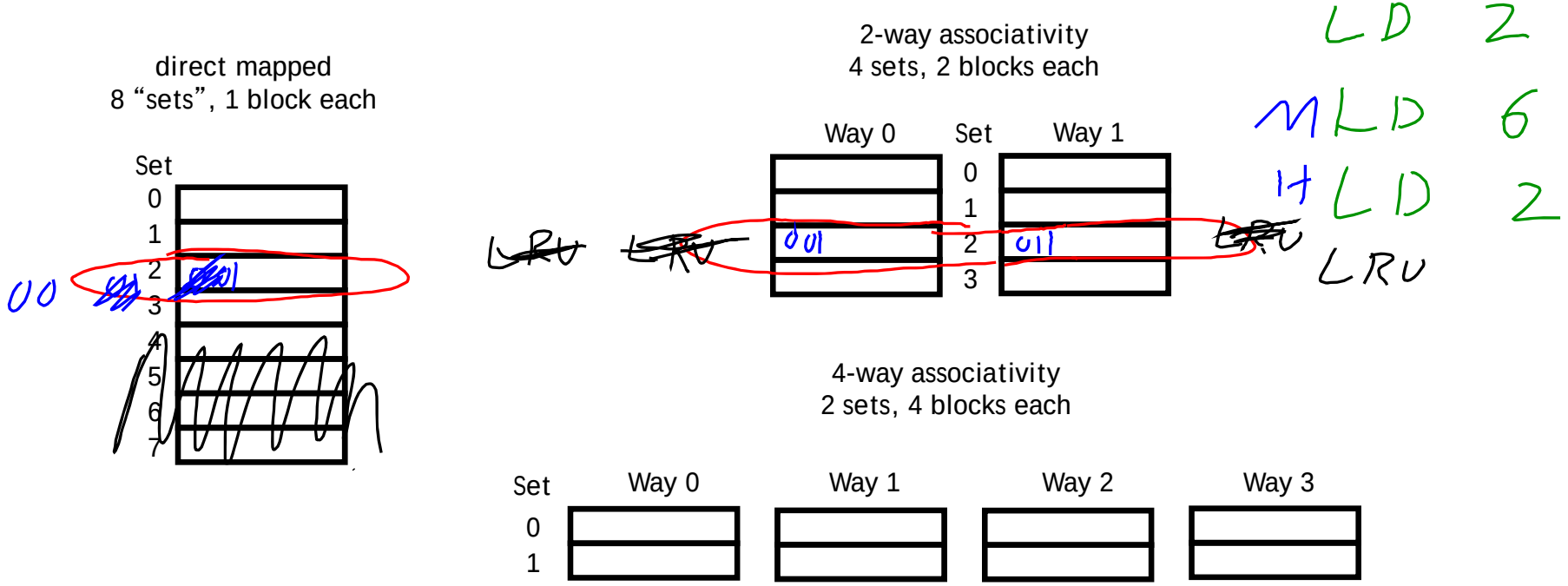
A fully-associative cache is expensive because we need to store the entire tag!

- Data could be anywhere in the cache, so we must check the tag of *every* cache block. That's a lot of comparators!

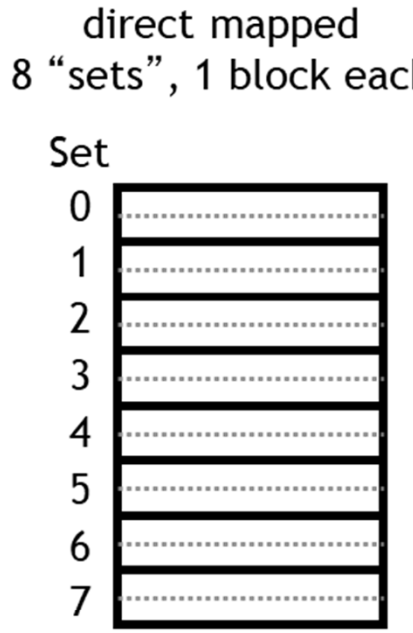


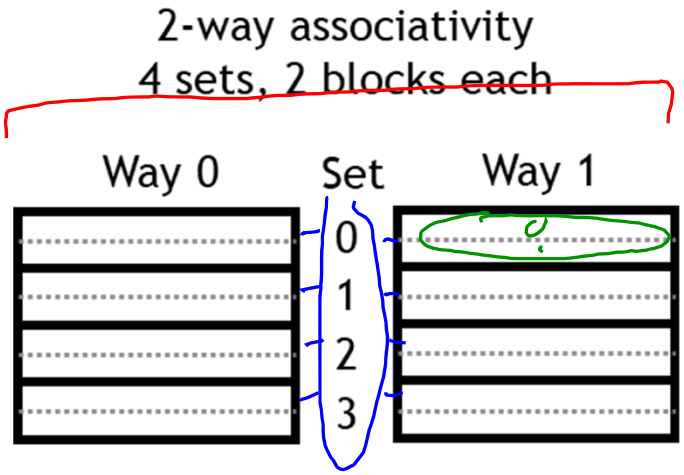
A set-associative cache organizes cache blocks into groups called sets

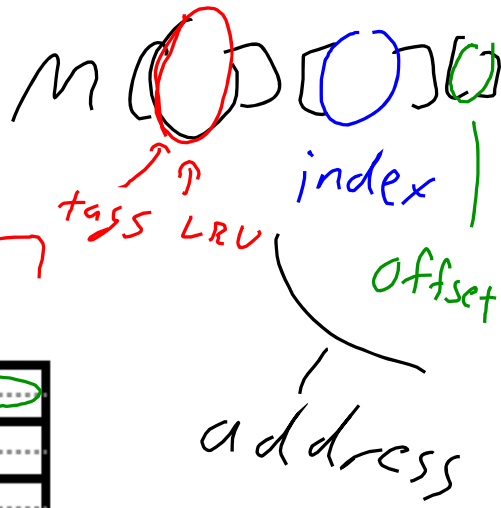
- Each memory address maps to exactly one set in the cache, but data may be placed in any block within that set.
- If each set has 2^x blocks, the cache is a 2^x-way associative cache.

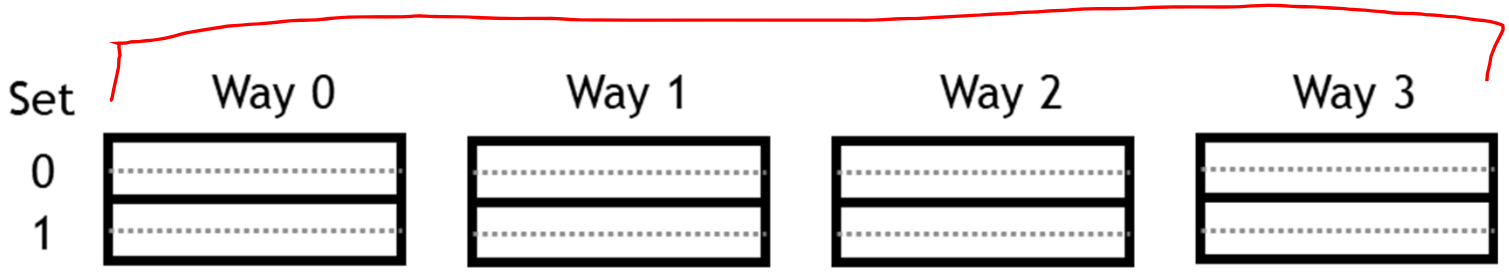


Blocks can still have multiple bytes



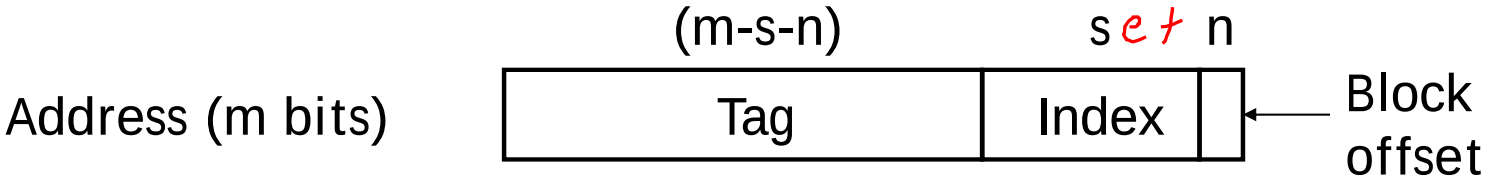






To find data, subdivide the address into tag, index, and block offset as before

- Memory has m-bit address
- Cache has 2^s sets and each block has 2^n bytes



- Our arithmetic computations now compute a set index, to select a *set* within the cache instead of an individual block.

Block Offset = Memory Address mod 2^n

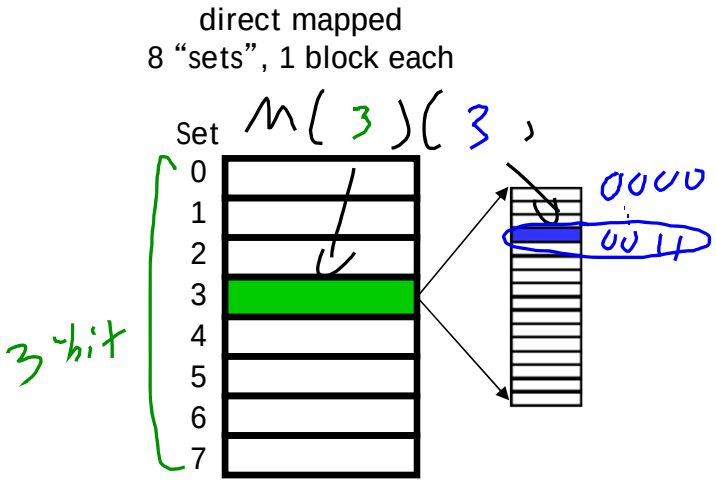
Set Index = (Memory Address / 2^n) mod 2^s

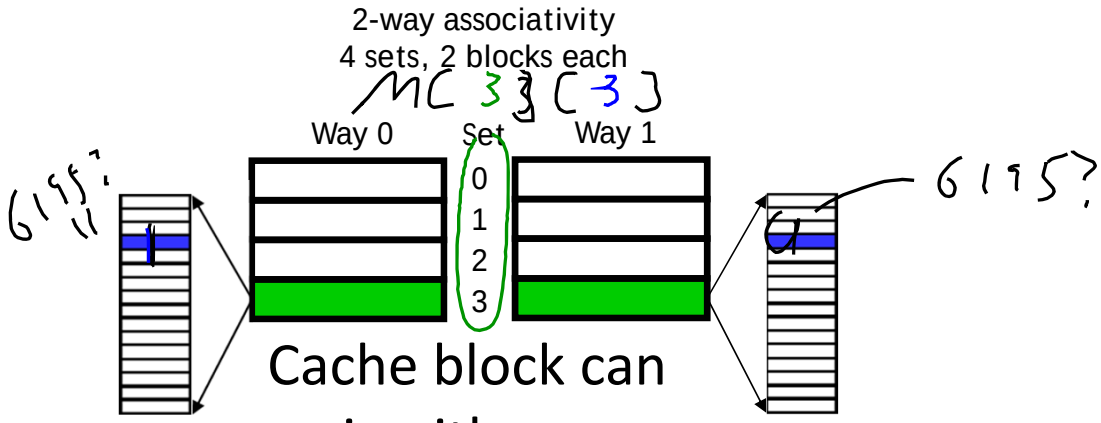
Example: Placement of address 6195 in caches with 8, 16-byte cache blocks

- 6195 in binary is 00...0110000 011 0011.
- Each block has 16 bytes, so the lowest 4 bits are the block offset.

The next three bits (011) are the set index.

The next two bits (11) are the set index.



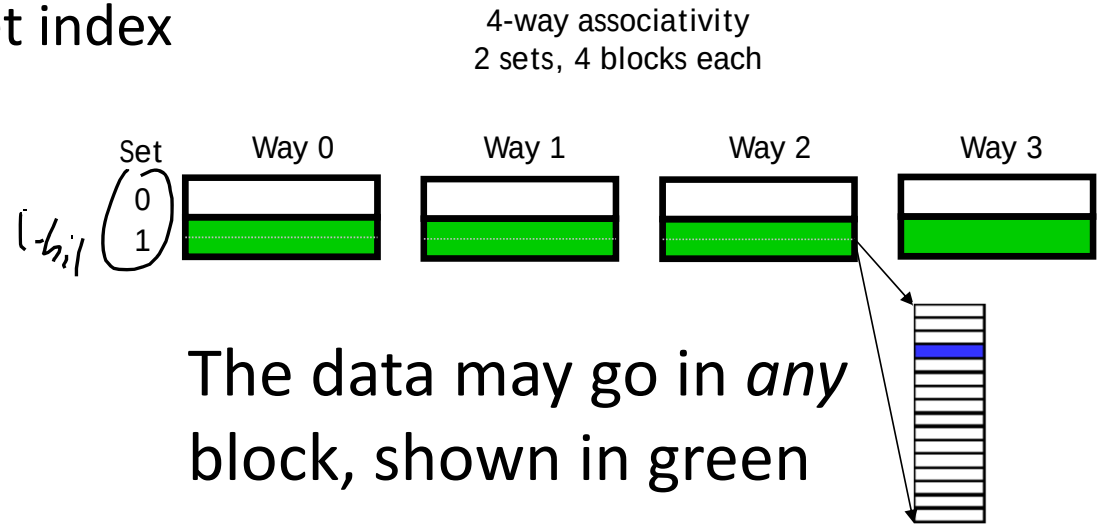


Cache block can go in either way

Example: Placement of address 6195 in caches with 8, 16-byte cache blocks

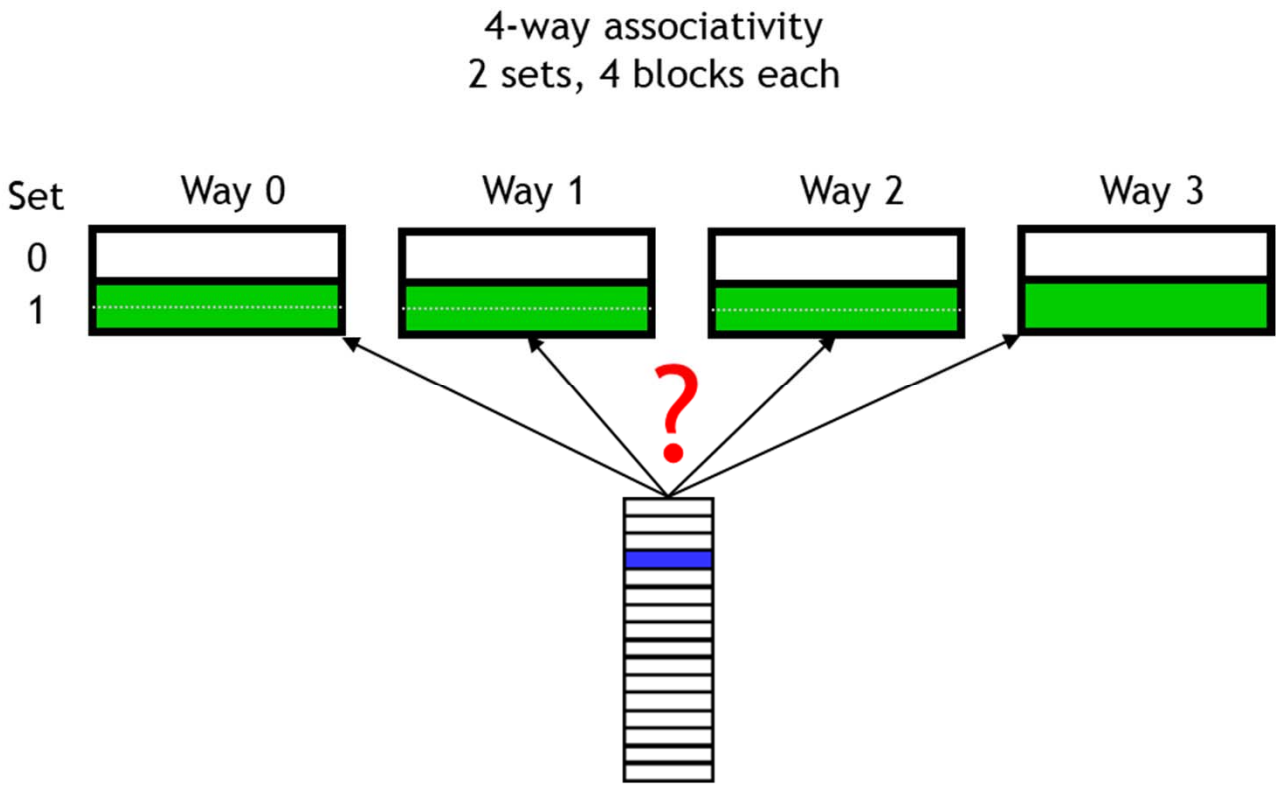
- 6195 in binary is 00...011000001 1 0011.
- Each block has 16 bytes, so the lowest 4 bits are the block offset.

the next one bit (1) is the set index



The data may go in *any* block, shown in green

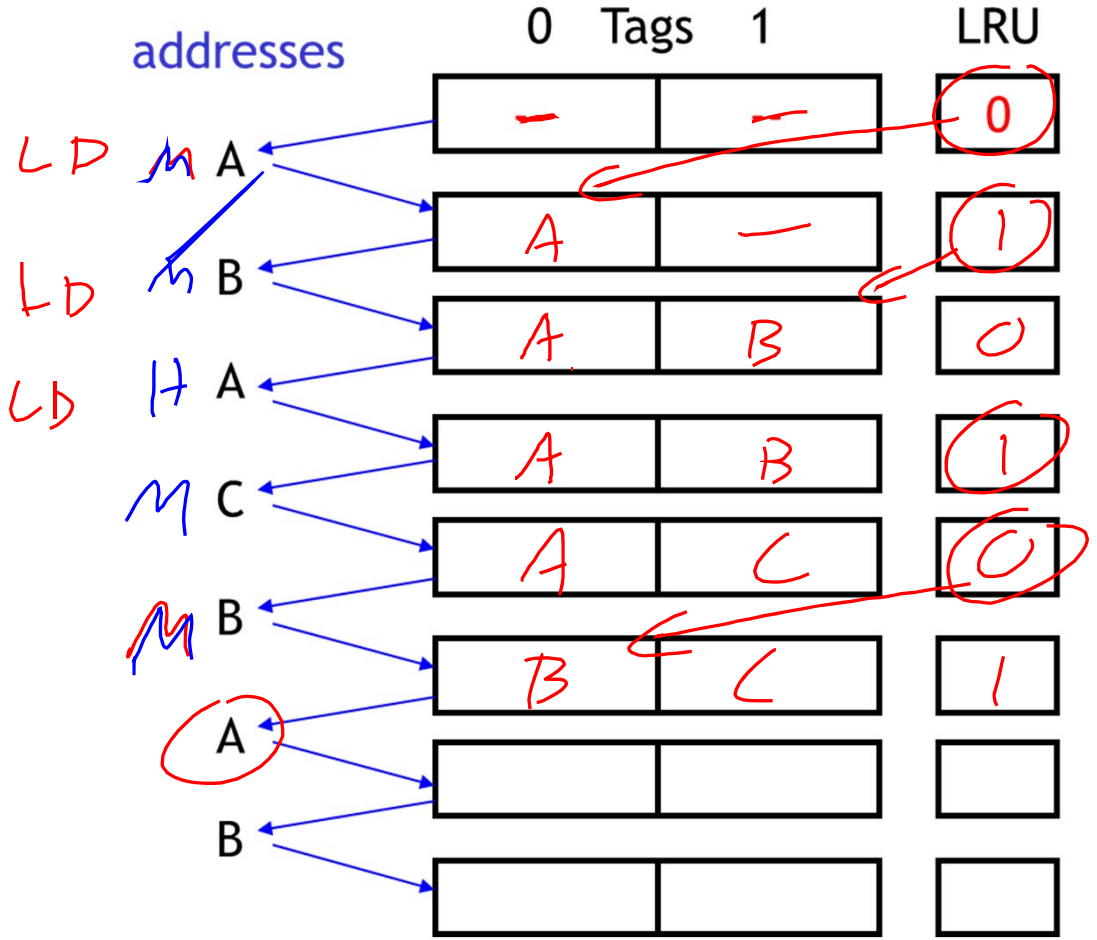
Place new data in empty cache blocks if possible, else, replace the least-recently used



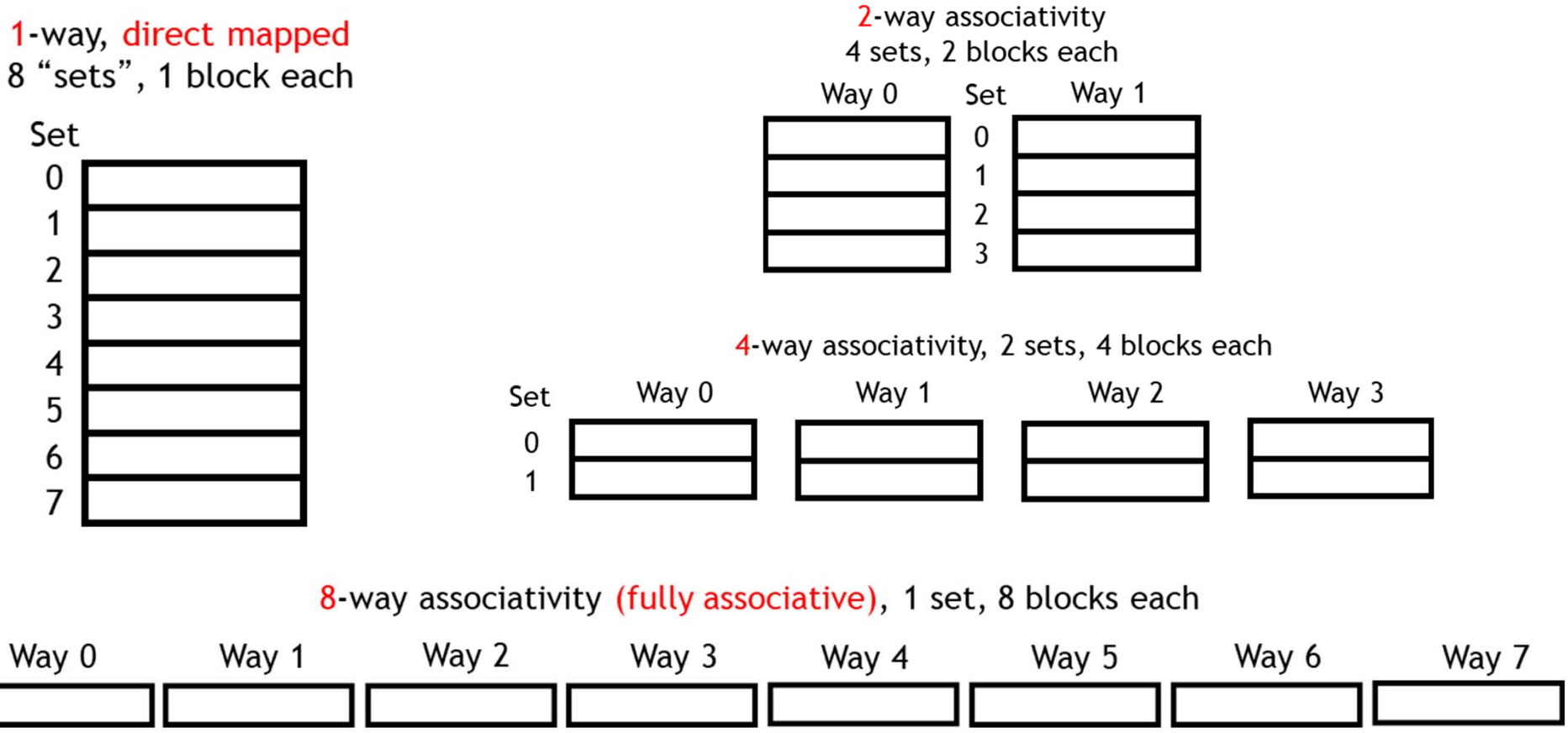
Approximate LRU for high associativity

Given a fully-associative cache with two blocks, which memory accesses miss in the cache?

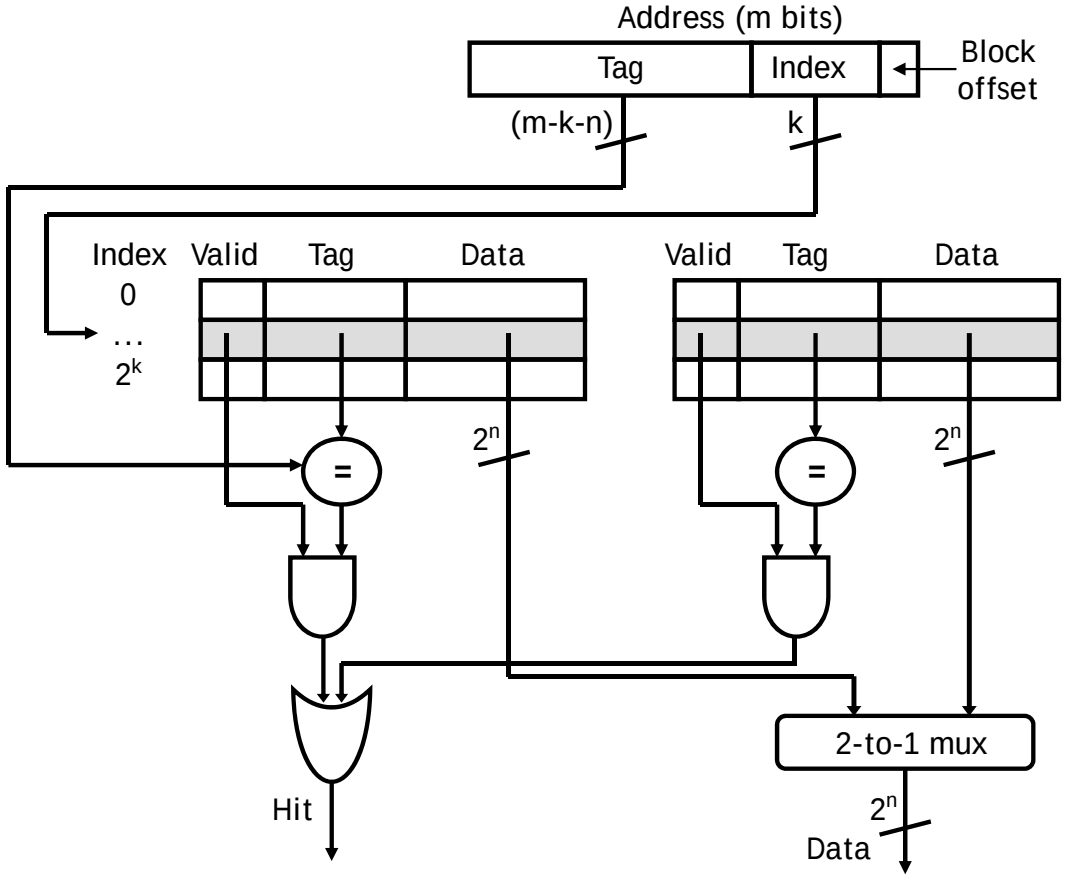
Assume distinct addresses go to distinct blocks



Direct-mapped and fully-associative caches are types of set-associative caches



Set associativity is implemented by using a multiplexer to choose from two identical caches



Summary

- Larger block sizes can take advantage of spatial locality by loading data from not just one address, but also nearby addresses, into the cache.
- Associative caches assign each memory address to a particular set within the cache, but not to any specific block within that set.
  - Set sizes range from 1 (direct-mapped) to 2^k (fully associative).
  - Larger sets and higher associativity lead to fewer cache conflicts and lower miss rates, but they also increase the hardware cost.
  - In practice, 2-way through 16-way set-associative caches strike a good balance between lower miss rates and higher costs.
- Next time, we'll talk more about measuring cache performance, and also discuss the issue of *writing* data to a cache.