

Lecture 18

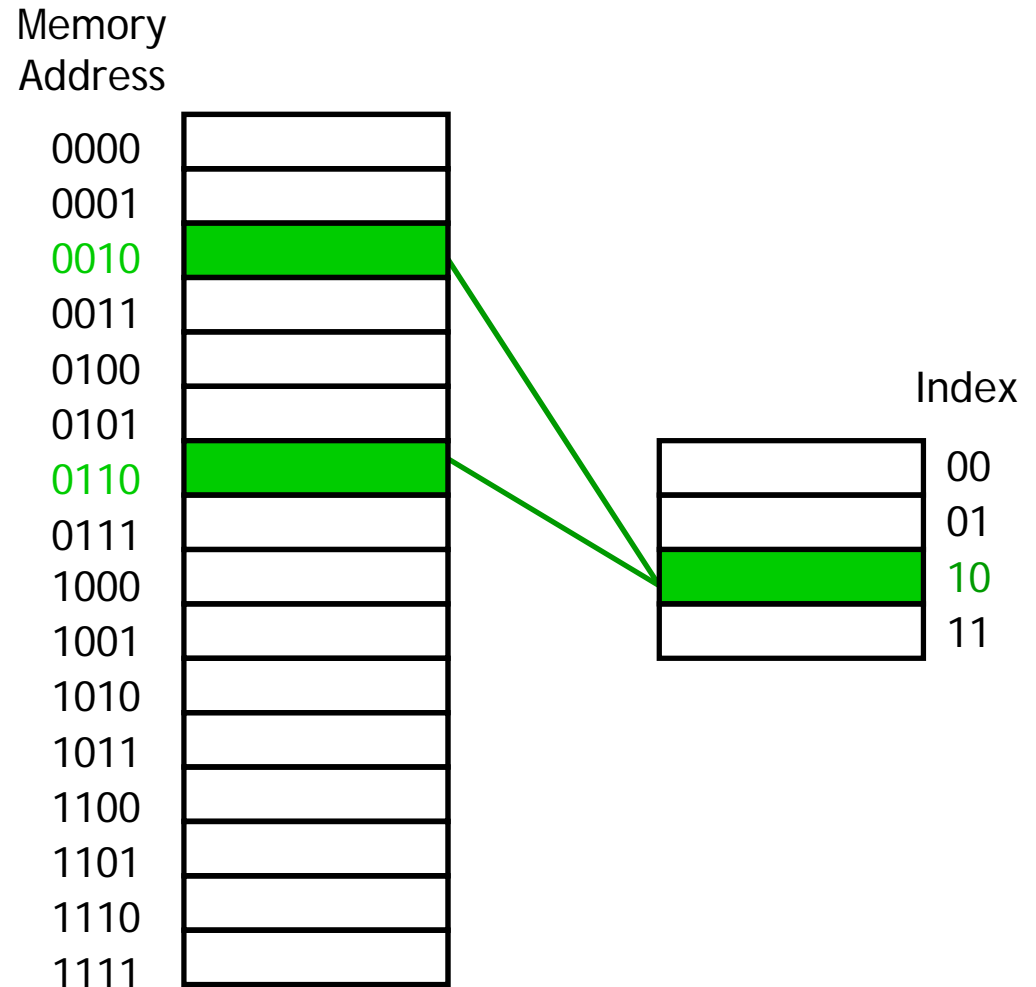
Review:

- What is cache block (or line)?
- What is an index?
- What is an offset?

- Today:
 - More cache organizations
- Later:
 - Dealing with writes
 - Virtual memory

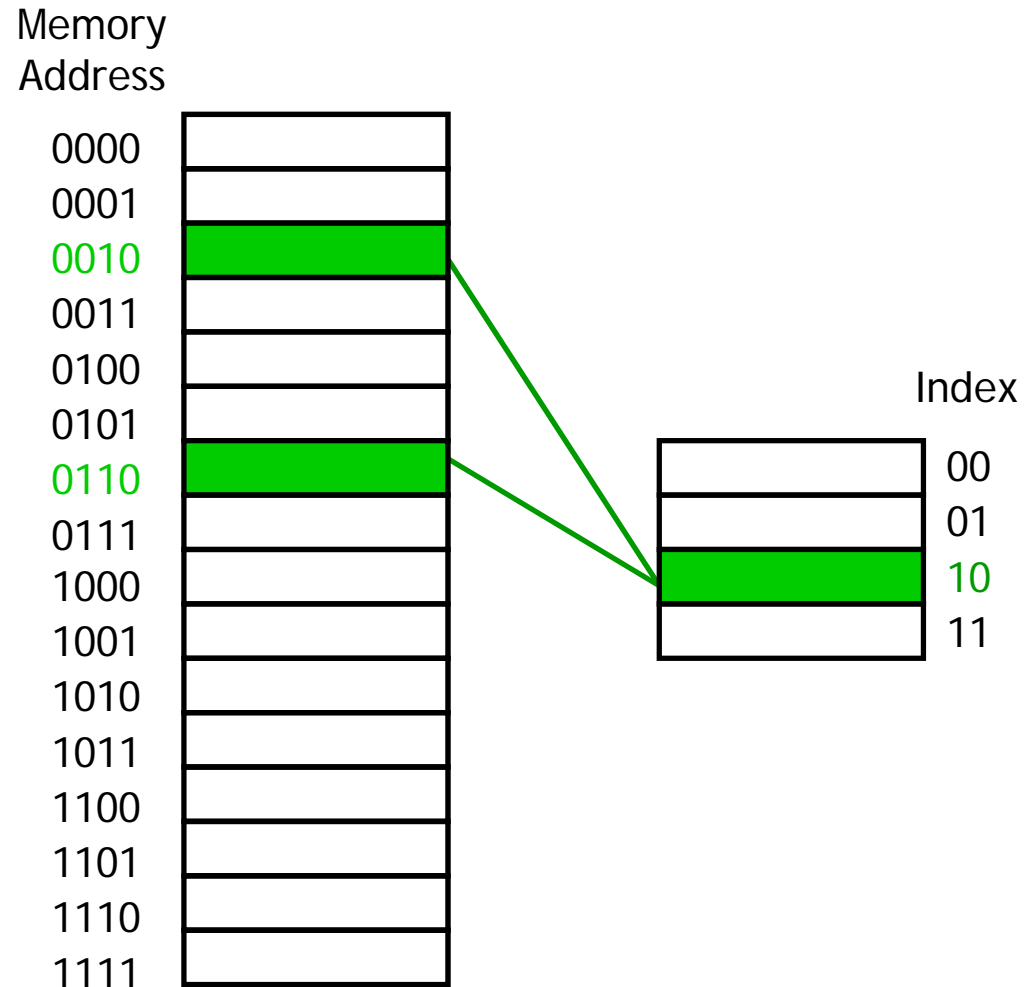
Disadvantage of direct mapping

- The direct-mapped cache is easy: indices and offsets can be computed with bit operators or simple arithmetic, because each memory address belongs in exactly one block.
- But, what happens if a program uses addresses *2, 6, 2, 6, 2, ...?*



Disadvantage of direct mapping

- The direct-mapped cache is easy: indices and offsets can be computed with bit operators or simple arithmetic, because each memory address belongs in exactly one block.
- However, this isn't really flexible. If a program uses addresses 2, 6, 2, 6, 2, ..., then each access will result in a cache miss and a load into cache block 2.
- This cache has four blocks, but direct mapping might not let us use all of them.
- This can result in more misses than we might like.

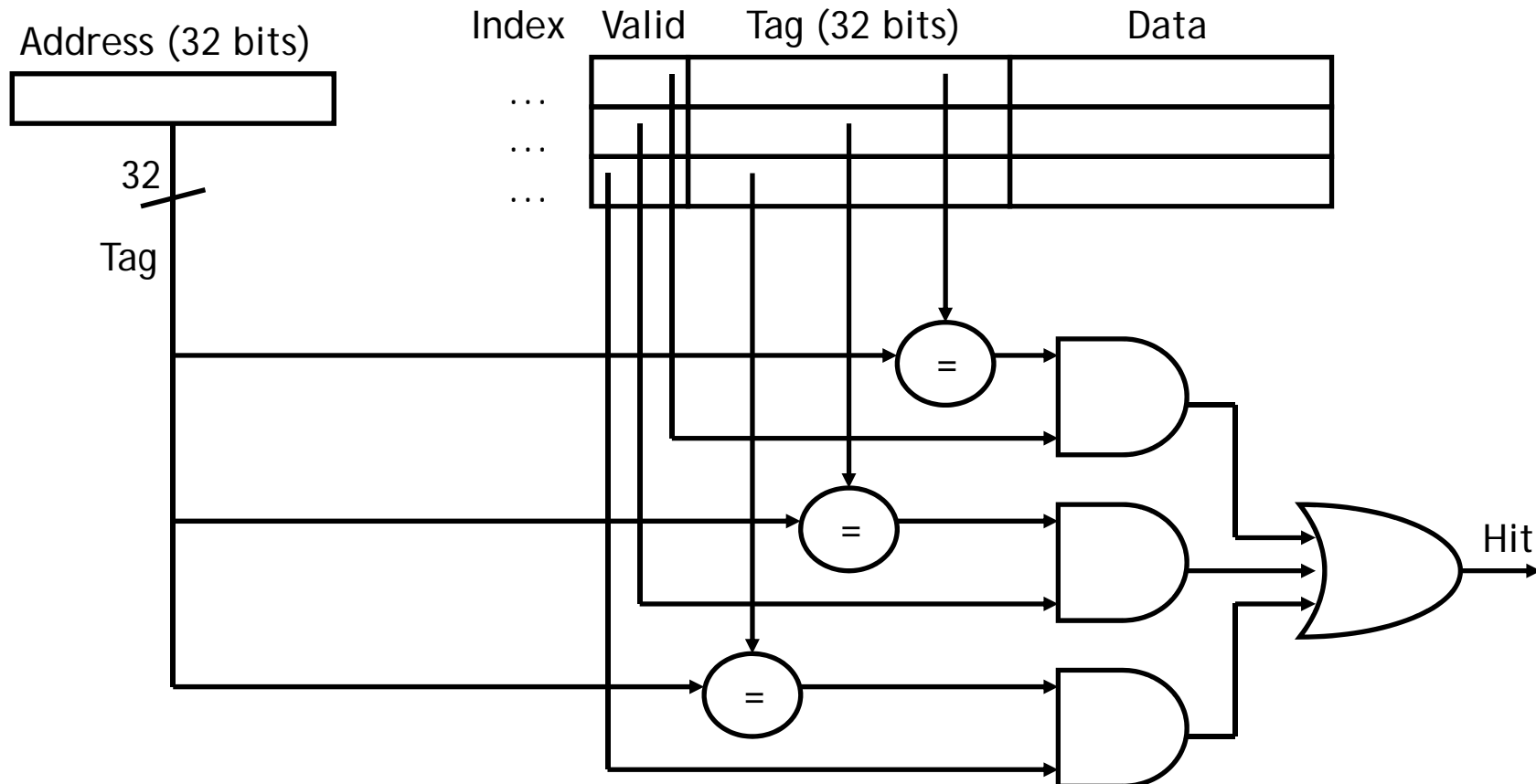


A fully associative cache

- A **fully associative cache** permits data to be stored in *any* cache block, instead of forcing each memory address into one particular block.
 - When data is fetched from memory, it can be placed in *any* unused block of the cache.
 - This way we'll never have a conflict between two or more memory addresses which map to a single cache block.
- In the previous example, we might put memory address 2 in cache block 2, and address 6 in block 3. Then subsequent repeated accesses to 2 and 6 would all be hits instead of misses.
- If all the blocks are already in use, it's usually best to replace the **least recently used** one, assuming that if it hasn't used it in a while, it won't be needed again anytime soon.

The price of full associativity

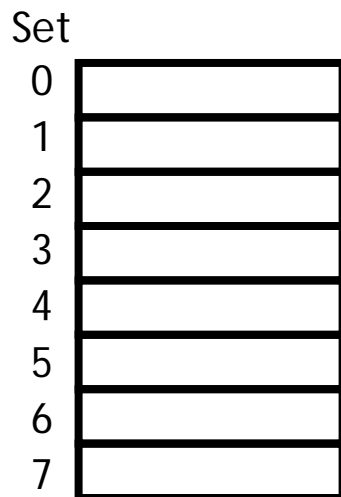
- However, a fully associative cache is expensive to implement.
 - Because there is no index field in the address anymore, the *entire* address must be used as the tag, increasing the total cache size.
 - Data could be anywhere in the cache, so we must check the tag of *every* cache block. That's a lot of comparators!



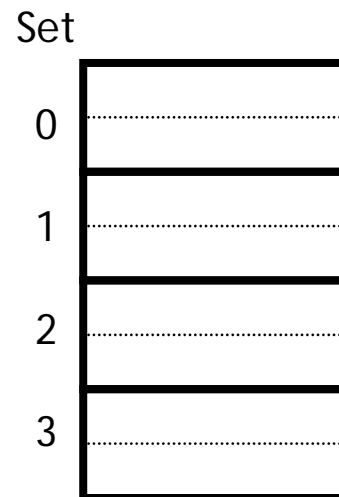
Set associativity

- An intermediate possibility is a **set-associative cache**.
 - The cache is divided into *groups* of blocks, 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 an **2^x -way associative cache**.
- Here are several possible organizations of an eight-block cache.

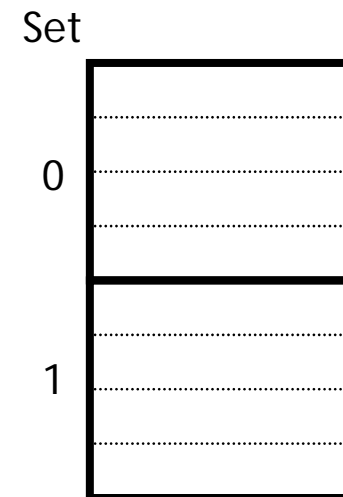
1-way associativity
8 sets, 1 block each



2-way associativity
4 sets, 2 blocks each

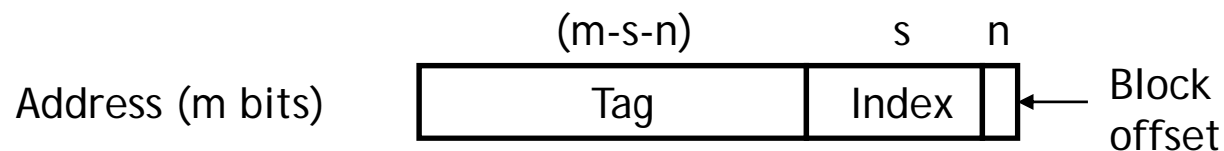


4-way associativity
2 sets, 4 blocks each



Locating a set associative block

- We can determine where a memory address belongs in an associative cache in a similar way as before.
- If a cache has 2^s sets and each block has 2^n bytes, the memory address can be partitioned as follows.



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

$$\text{Block Offset} = \text{Memory Address} \bmod 2^n$$

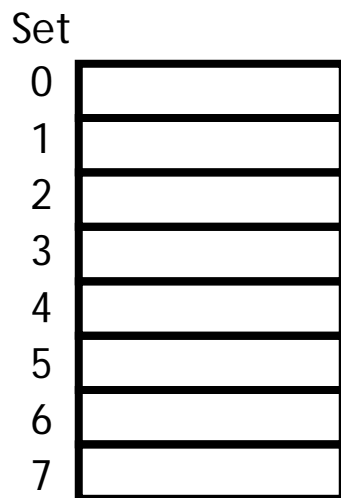
$$\text{Block Address} = \text{Memory Address} / 2^n$$

$$\text{Set Index} = \text{Block Address} \bmod 2^s$$

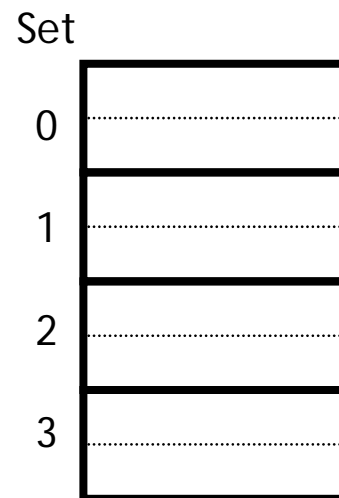
Example placement in set-associative caches

- Where would data from memory byte address 6195 be placed, assuming the eight-block cache designs below, with 16 bytes per block?
- 6195 in binary is 00...0110000 **011** **0011**.

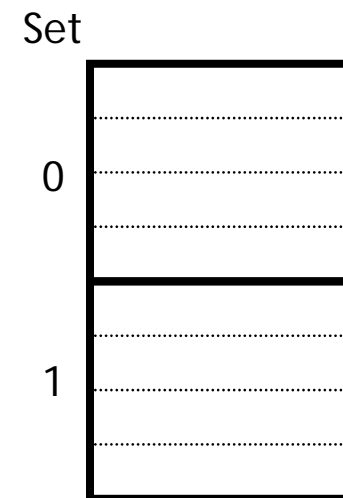
1-way associativity
8 sets, 1 block each



2-way associativity
4 sets, 2 blocks each



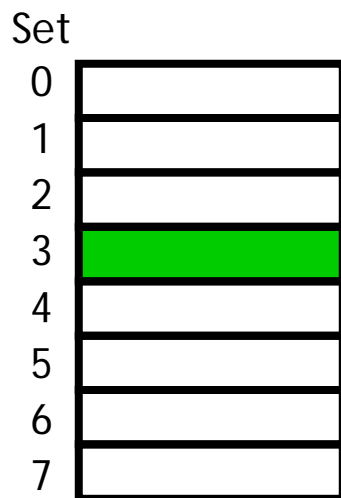
4-way associativity
2 sets, 4 blocks each



Example placement in set-associative caches

- Where would data from memory byte address 6195 be placed, assuming the eight-block cache designs below, with 16 bytes per block?
- 6195 in binary is 00...0110000 **011** **0011**.
- Each block has 16 bytes, so the **lowest 4 bits are the block offset**.
- For the 1-way cache, the next three bits (**011**) are the set index.
For the 2-way cache, the next two bits (**11**) are the set index.
For the 4-way cache, the next one bit (**1**) is the set index.
- The data may go in *any* block, shown in green, within the correct set.

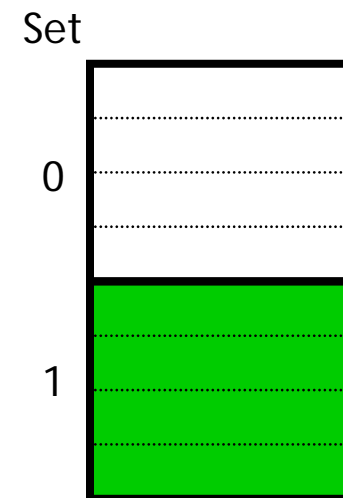
1-way associativity
8 sets, 1 block each



2-way associativity
4 sets, 2 blocks each



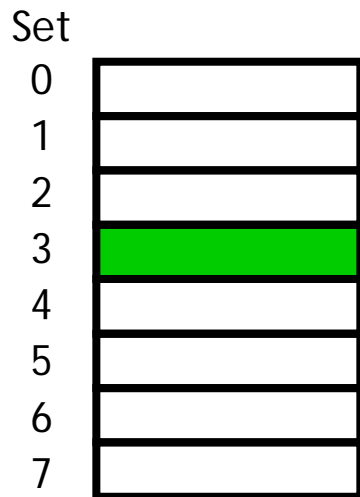
4-way associativity
2 sets, 4 blocks each



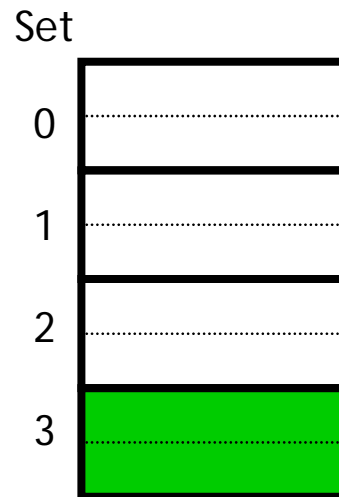
Block replacement

- Any empty block in the correct set may be used for storing data.
- If there are no empty blocks, which one should we replace?

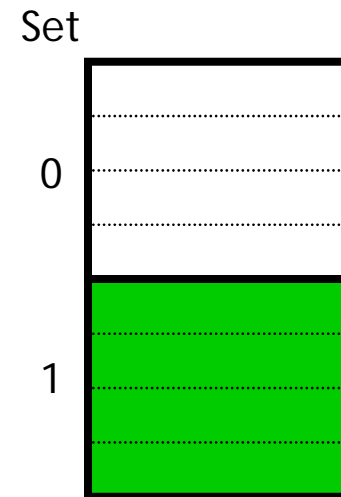
1-way associativity
8 sets, 1 block each



2-way associativity
4 sets, 2 blocks each



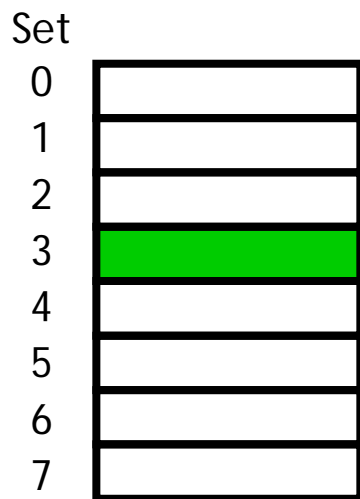
4-way associativity
2 sets, 4 blocks each



Block replacement

- Any empty block in the correct set may be used for storing data.
- If there are no empty blocks, the cache controller will attempt to replace the least recently used block, just like before.
- For highly associative caches, it's expensive to keep track of what's really the least recently used block, so some approximations are used. We won't get into the details.

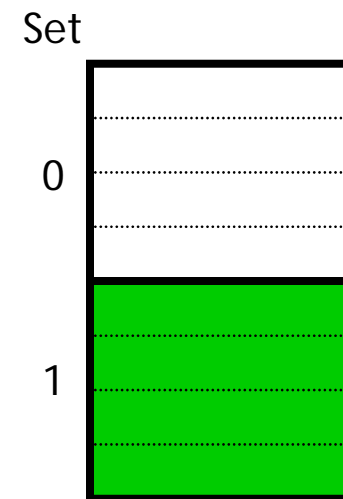
1-way associativity
8 sets, 1 block each



2-way associativity
4 sets, 2 blocks each

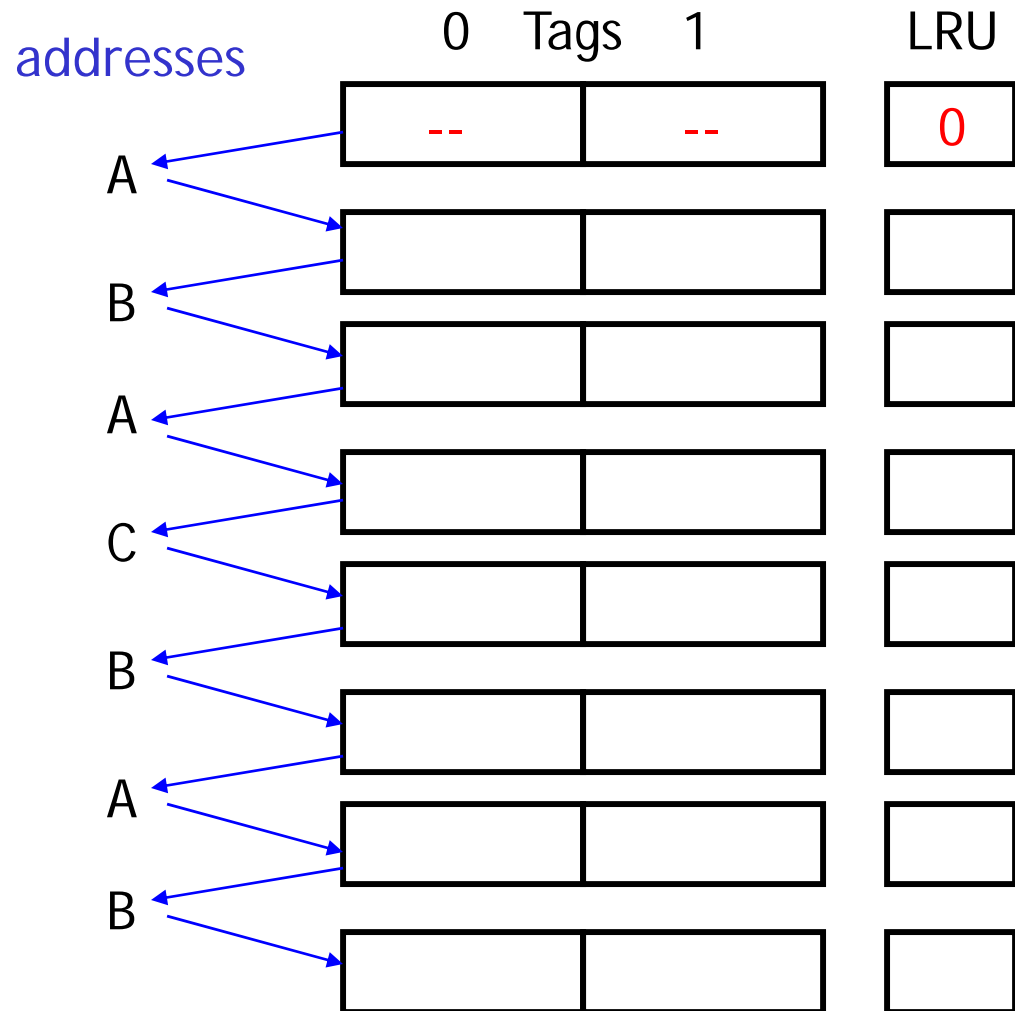


4-way associativity
2 sets, 4 blocks each



LRU example

- Assume a fully-associative cache with two blocks, which of the following memory references miss in the cache.
 - assume distinct addresses go to distinct blocks

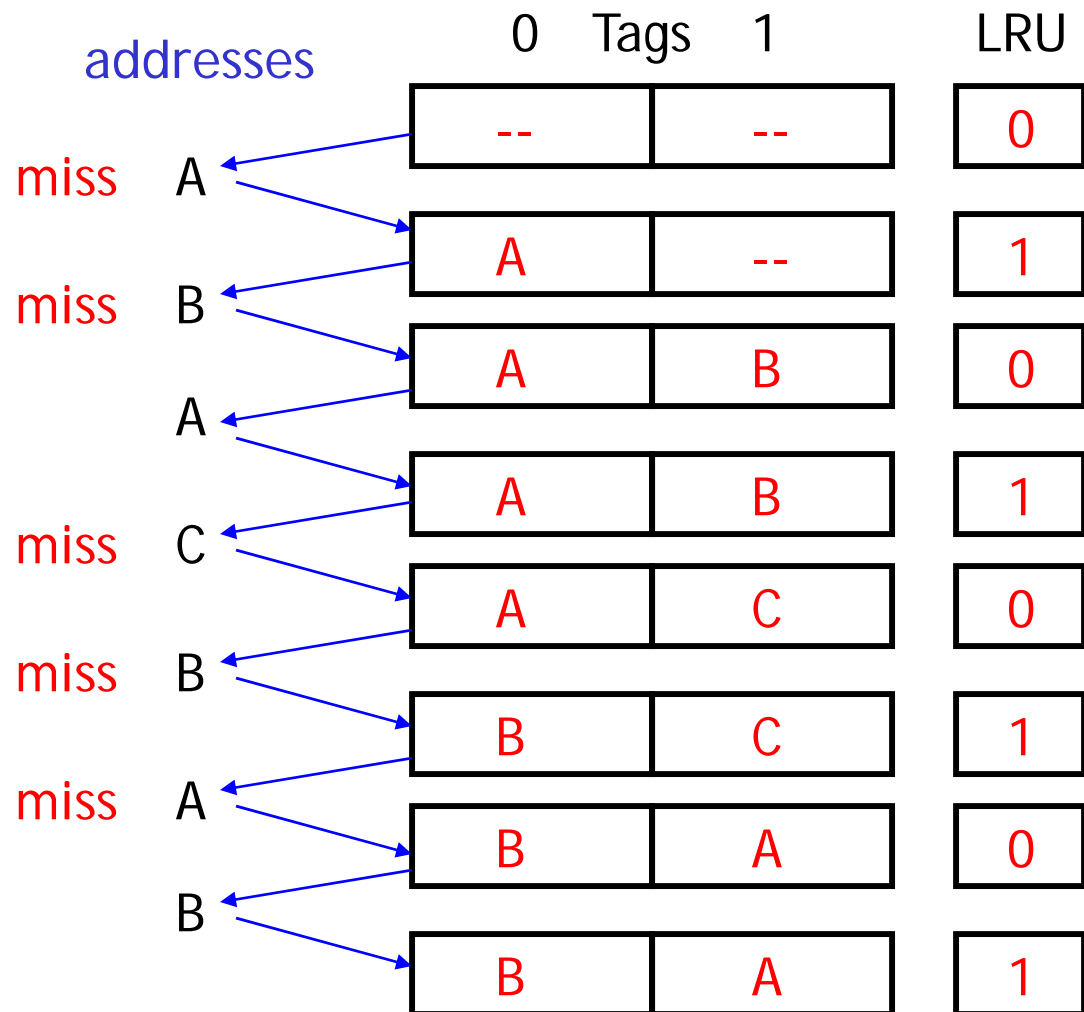


LRU example

- Assume a fully-associative cache with two blocks, which of the following memory references miss in the cache.
 - assume distinct addresses go to distinct blocks

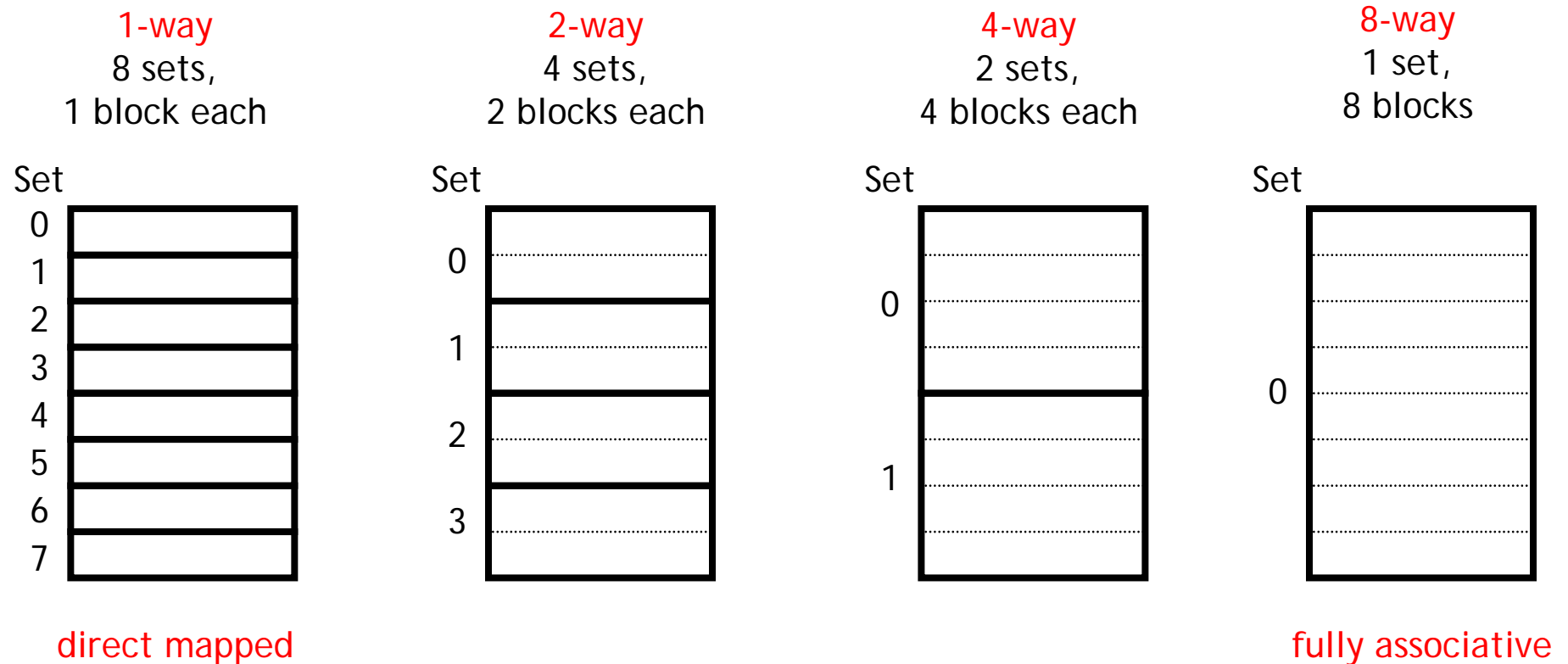
On a miss, we replace the LRU.

On a hit, we just update the LRU.



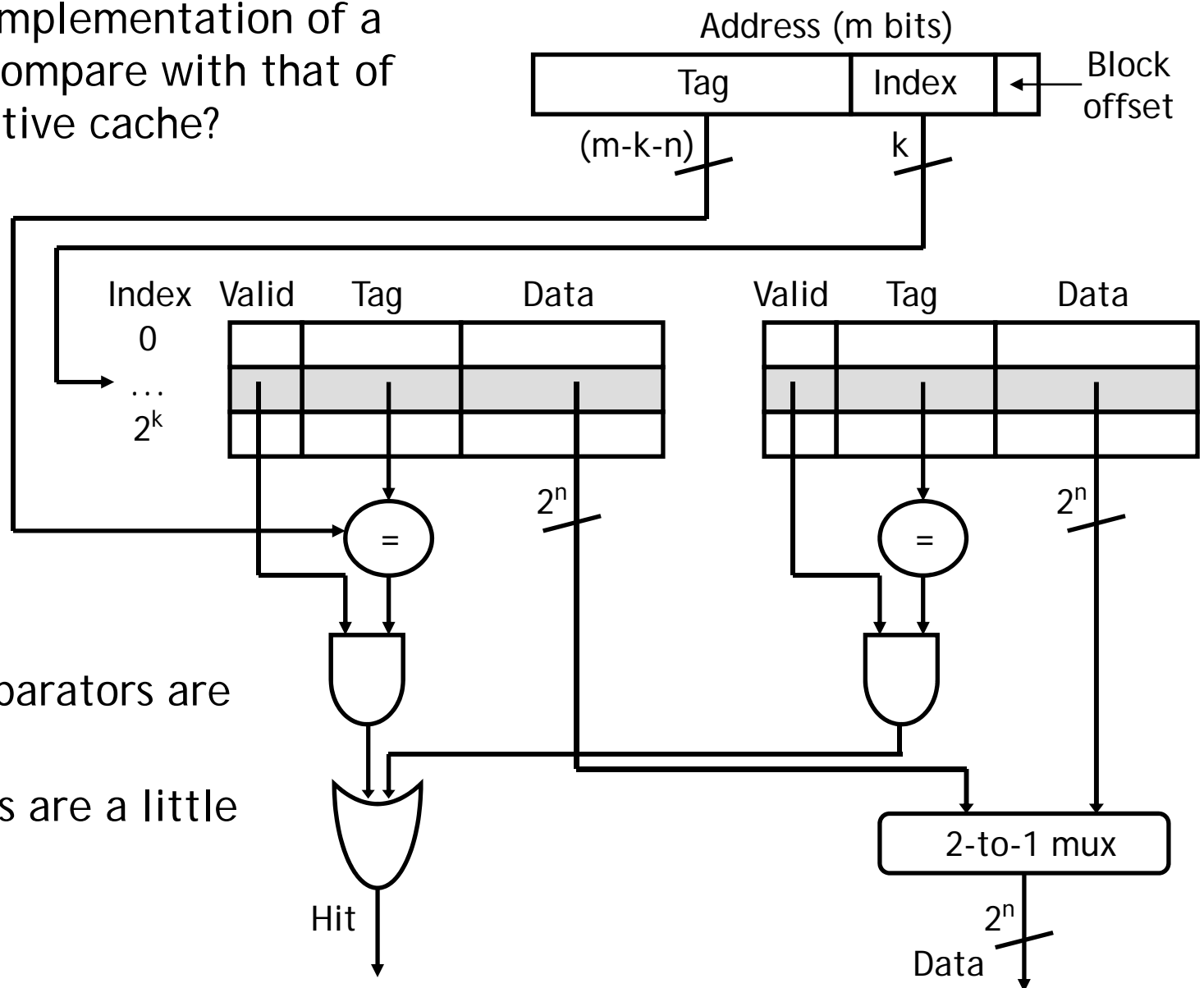
Set associative caches are a general idea

- By now you may have noticed the 1-way set associative cache is the same as a **direct-mapped** cache.
- Similarly, if a cache has 2^k blocks, a 2^k -way set associative cache would be the same as a **fully-associative** cache.



2-way set associative cache implementation

- How does an implementation of a 2-way cache compare with that of a fully-associative cache?



- Only two comparators are needed.
- The cache tags are a little shorter too.

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, we'll talk more about measuring cache performance, and also discuss the issue of *writing* data to a cache.