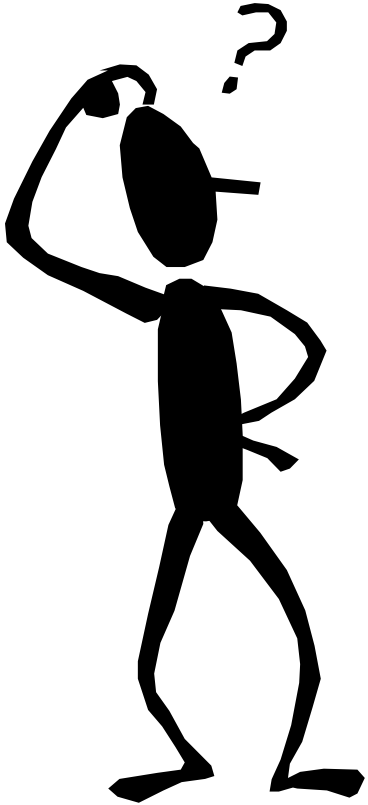


More cache organizations



- We want to store a copy of our most frequently used data in a small, fast **cache** memory, so we can speed up most data reads and writes.
 - Normal programs exhibit **temporal and spatial locality**, which makes it easier to guess what data will be “most frequently used.”
 - Last week we showed a basic **direct-mapped cache**, and discussed the need for additional **tag** and **valid** bits.
- Today we’ll explore some more complex cache organizations.
 - How can we take advantage of spatial locality too?
 - How can we reduce the number of potential conflicts?

Four important questions

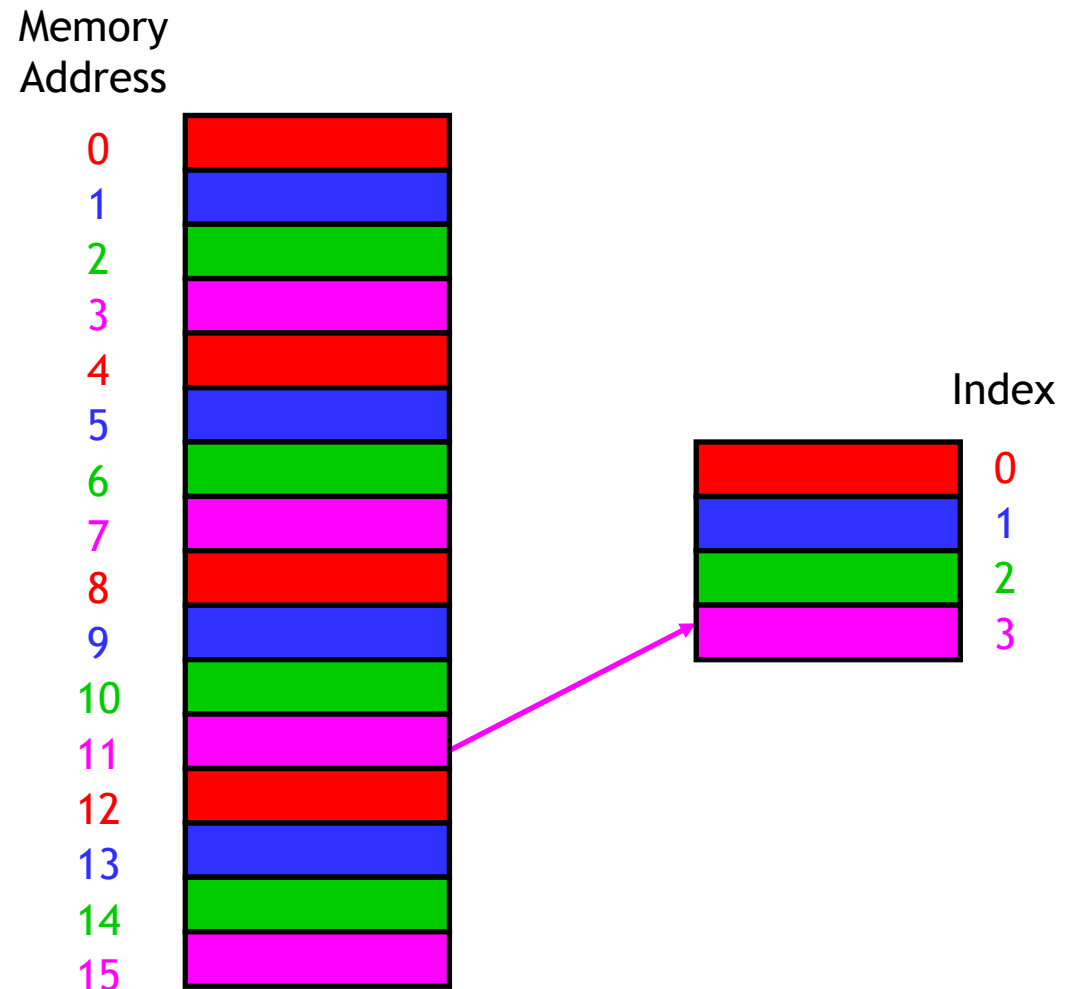


1. When we copy a block of data from main memory to the cache, where exactly should we put it?
2. How can we tell if a word is already in the cache, or if it has to be fetched from main memory first?
3. Eventually, the small cache memory might fill up. To load a new block from main RAM, we'd have to replace one of the existing blocks in the cache... which one?
4. How can *write* operations be handled by the memory system?

- Questions 1 and 2 are related—we have to know where the data is placed if we ever hope to find it again later!

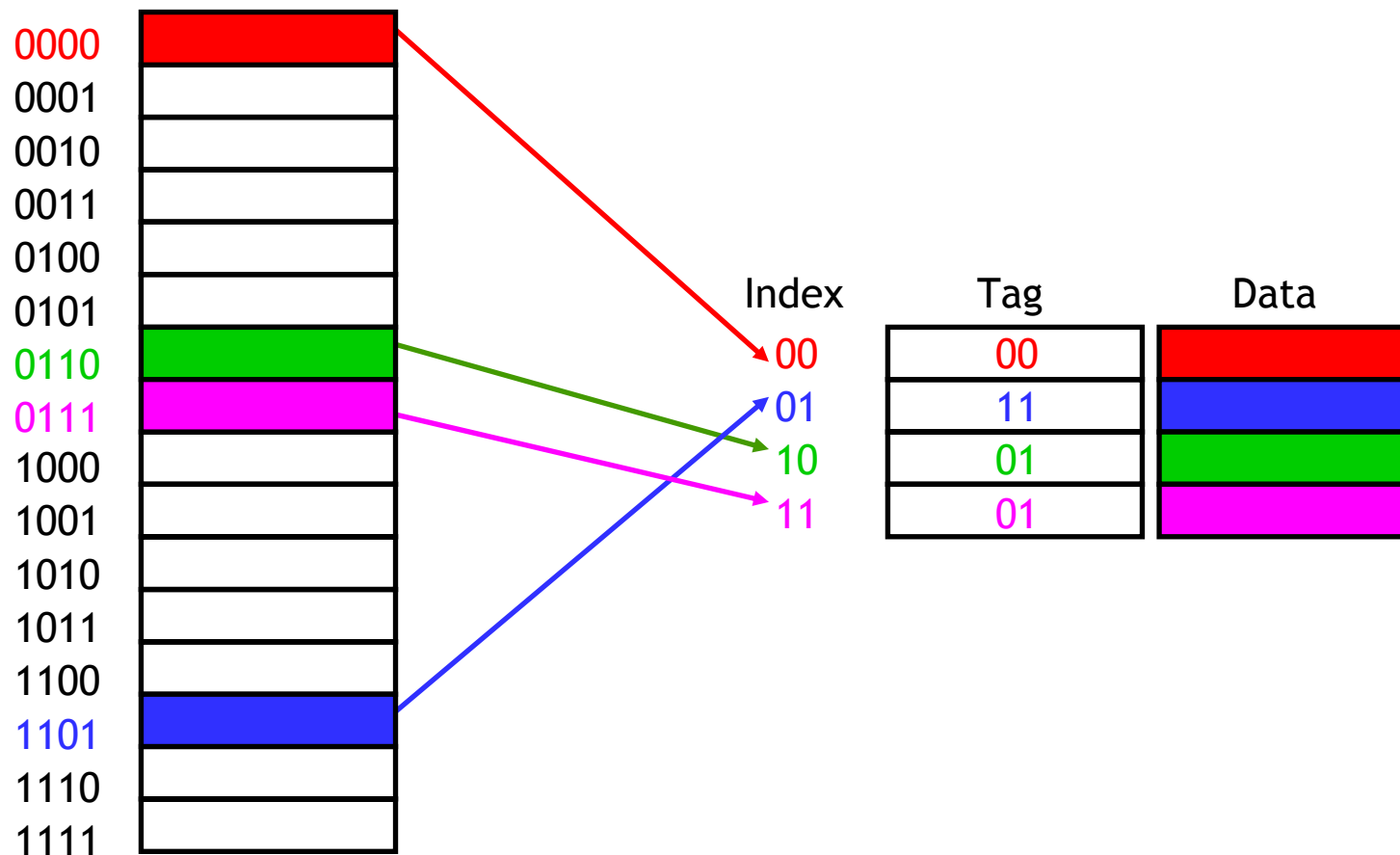
Direct-mapped caches

- Last time we introduced caches with **direct mapping**, where the main memory address determines which cache block stores the data.
- If the cache contains 2^k blocks, then data from address i would be stored in block $i \bmod 2^k$.
- Equivalently, you can look at the lowest k bits of the memory address.
- For example, data from memory address **11** maps to cache block **3** on the right, since $11 \bmod 4 = 3$ and since the lowest two bits of 1011 are 11.



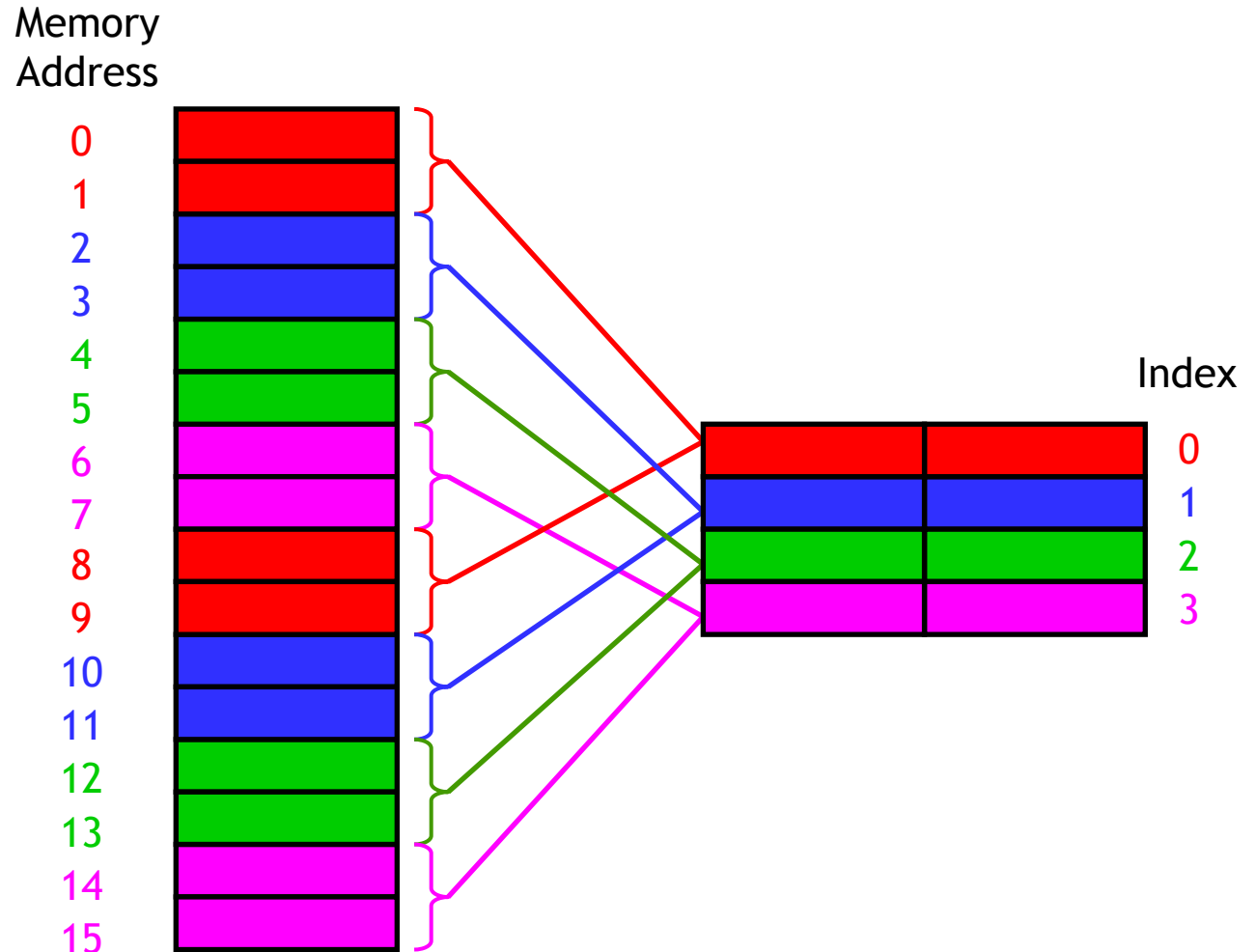
Tags

- To find data stored in the cache, we need to add **tags** to distinguish between different memory locations that map to the same cache block.
- Below, you can see that data from memory address **7 (0111)** is stored in cache block 3, as opposed to data from addresses 3, 11 or 15.



Spatial locality

- One-byte cache blocks don't take advantage of **spatial locality**, which predicts that an access to one address will be followed by an access to a nearby address.
- What we can do is make the cache block size larger than one byte.
- Here we use two-byte blocks, so we can load the cache with two bytes at a time.
- If we read from address **12**, the data in addresses **12 and 13** would both be copied to cache block **2**.

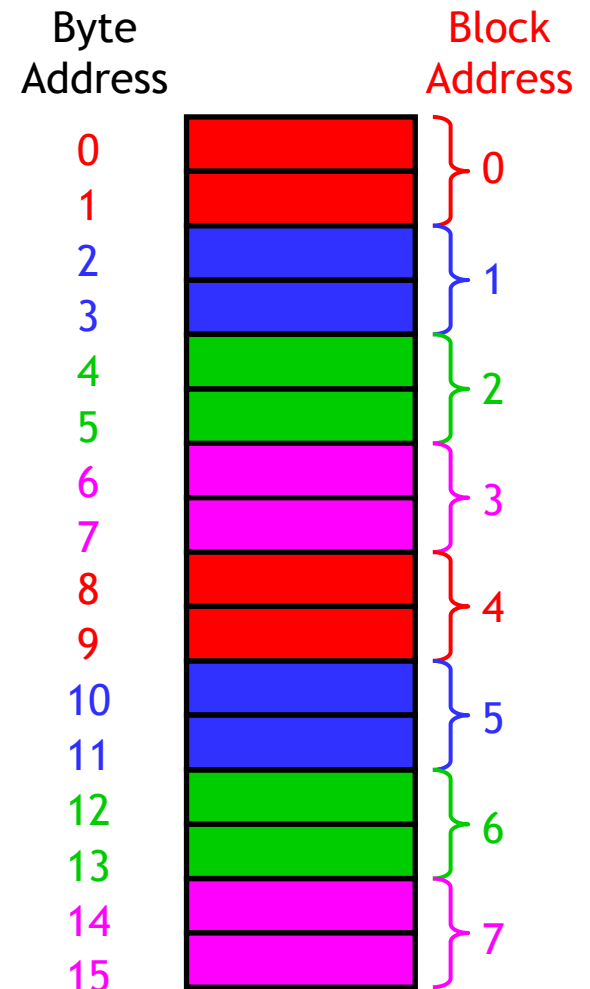


Block addresses

- Now how can we figure out where data should be placed in the cache?
- It's time for **block addresses**! If the cache block size is 2^n bytes, we can conceptually split the main memory into 2^n -byte chunks too.
- To determine the block address of a byte address i , you can do the integer division

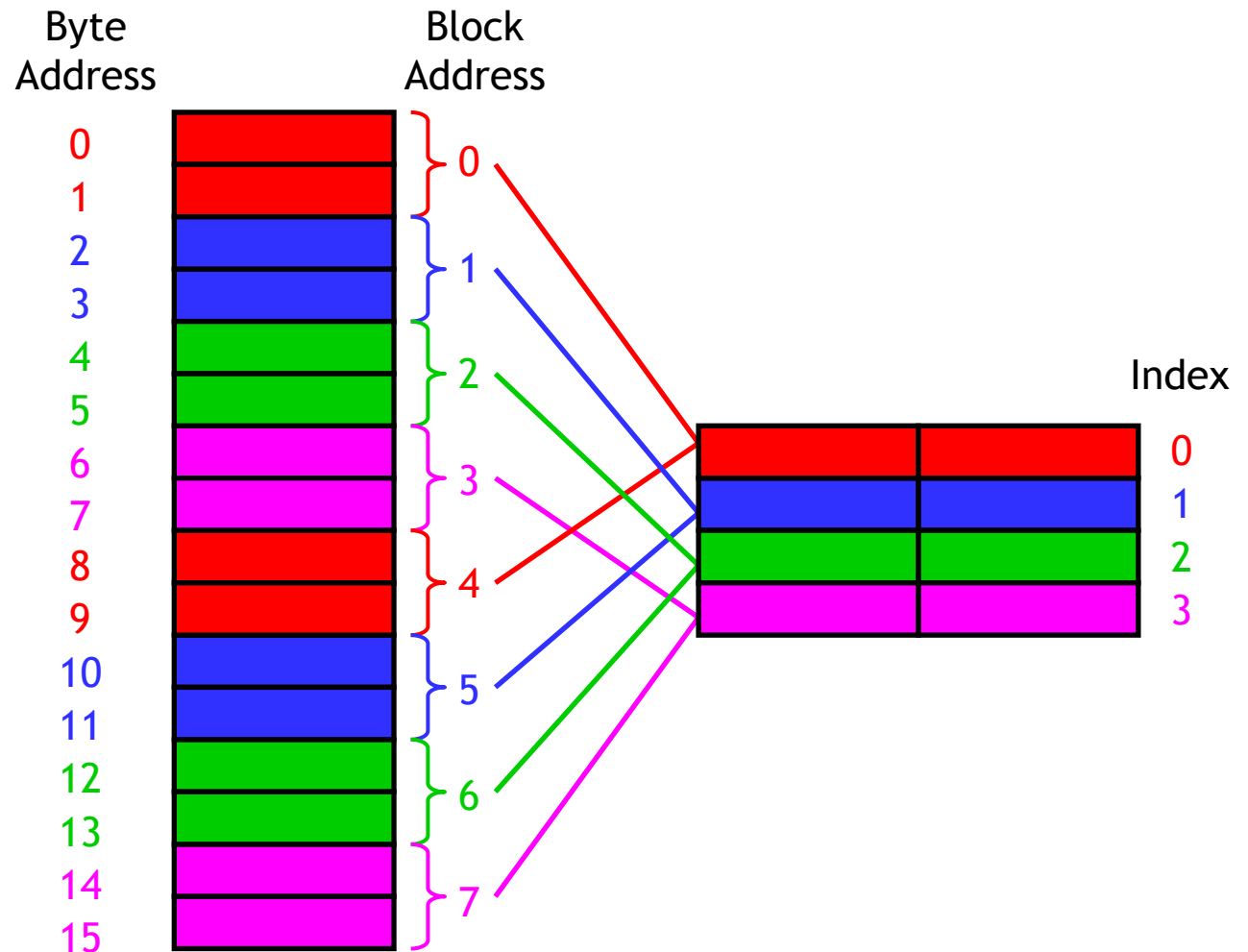
$$i / 2^n$$

- Our example has two-byte cache blocks, so we can think of a 16-byte main memory as an "8-block" main memory instead.
- For instance, memory addresses 12 and 13 both correspond to block address 6, since $12 / 2 = 6$ and $13 / 2 = 6$.



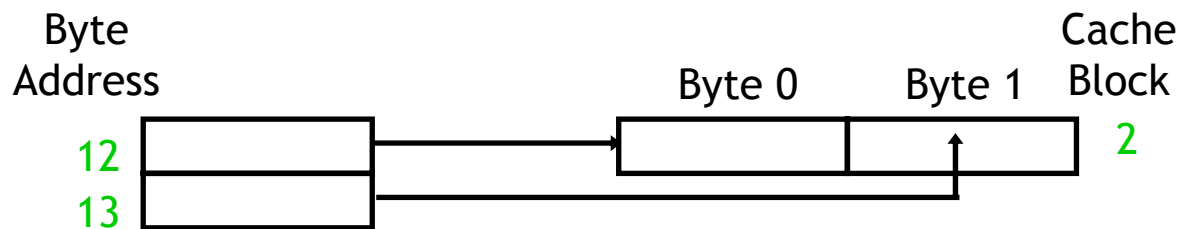
Cache mapping

- Once you know the block address, you can map it to the cache as before: find the remainder when the block address is divided by the number of cache blocks.
- In our example, memory block 6 belongs in cache block 2, since $6 \bmod 4 = 2$.
- This corresponds to placing data from memory *byte* addresses 12 and 13 into cache block 2.



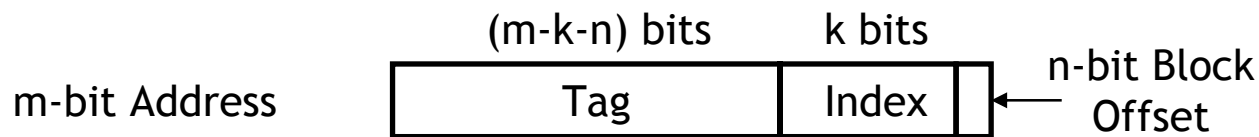
Data placement within a block

- When we access one byte of data in memory, we'll copy its entire *block* into the cache, to hopefully take advantage of spatial locality.
- In our example, if a program reads from byte address 12 we'll load all of memory block 6 (both addresses 12 and 13) into cache block 2.
- Note byte address 13 corresponds to the *same* memory block address! So a read from address 13 will also cause memory block 6 (addresses 12 and 13) to be loaded into cache block 2.
- To make things simpler, byte i of a memory block is always stored in byte i of the corresponding cache block.

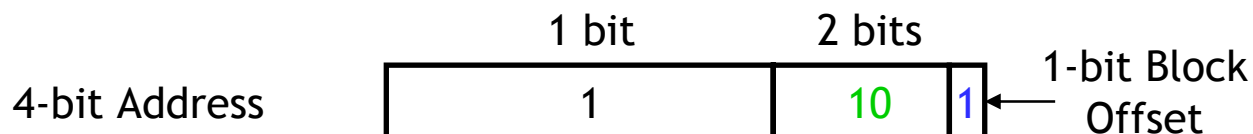


Locating data in the cache

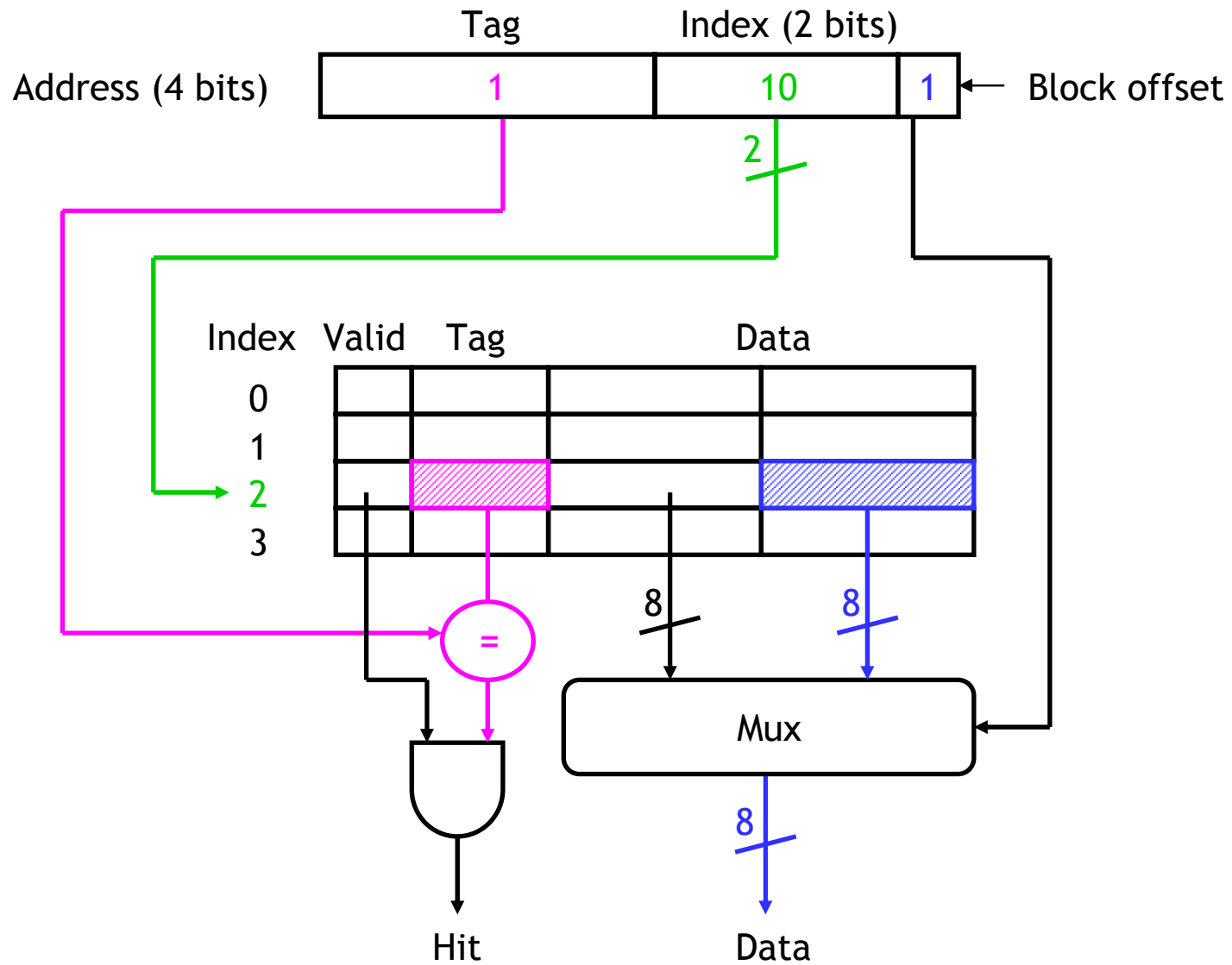
- Let's say we have a cache with 2^k blocks, each containing 2^n bytes.
- We can determine where a byte of data belongs in this cache by looking at its address in main memory.
 - k bits of the address will select one of the 2^k cache blocks.
 - The lowest n bits are now a **block offset** that decides which of the 2^n bytes in the cache block will store the data.



- Our example used a 2^2 -block cache with 2^1 bytes per block. Thus, memory address 13 (1101) would be stored in byte 1 of cache block 2.

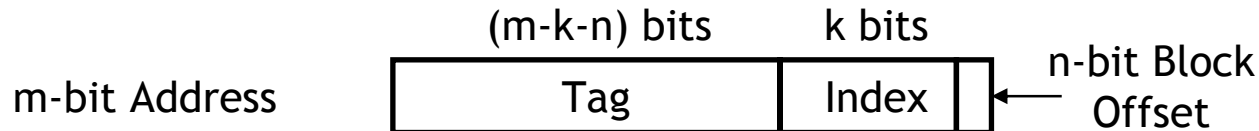


A picture



Using arithmetic

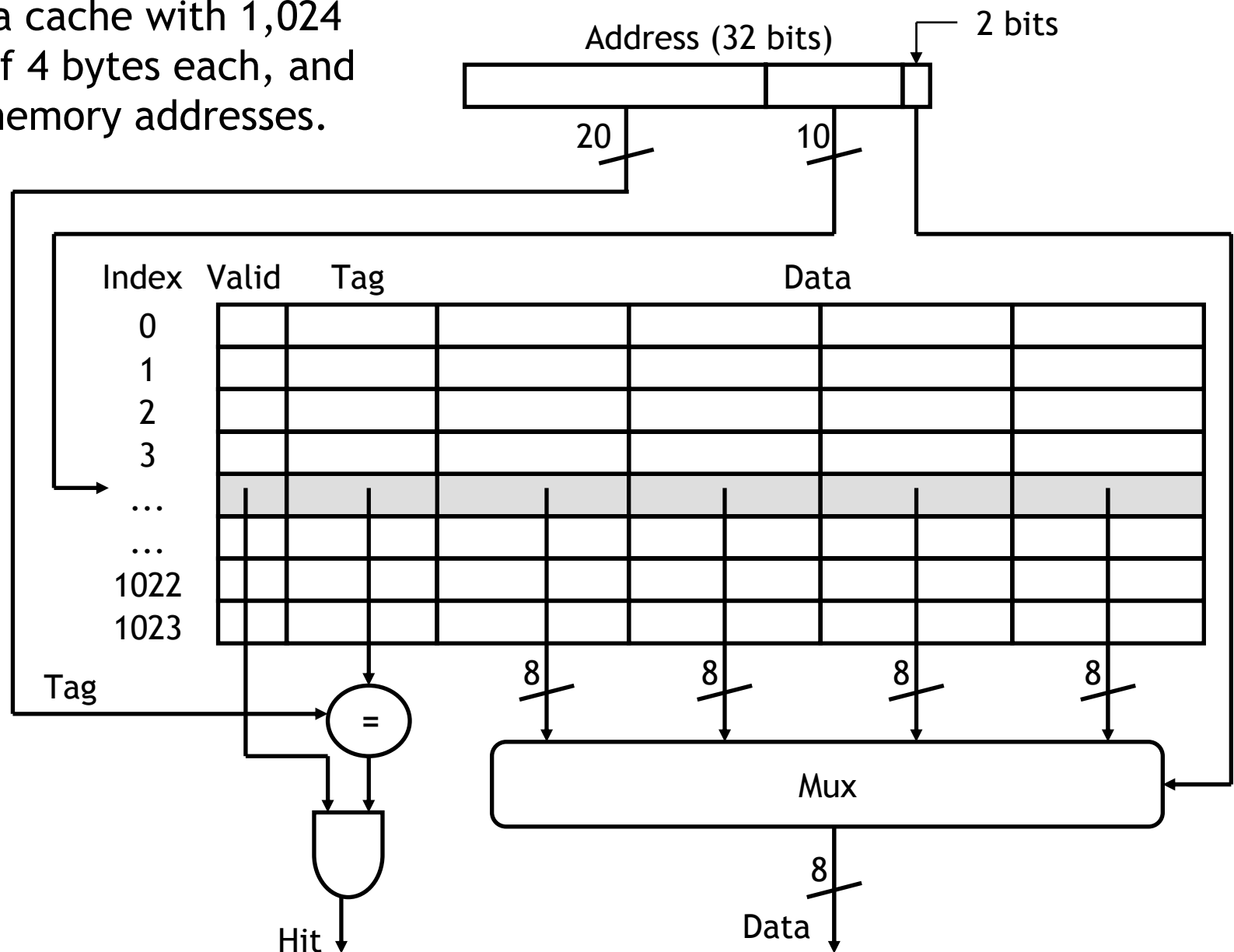
- An equivalent way to find the right location within the cache is to use arithmetic again.



- We can find the **index** in two steps, as outlined earlier.
 - Do integer division of the address by 2^n to find the block address.
 - Then mod the block address with 2^k to find the index.
- The **block offset** is just the memory address mod 2^n .
- For example, we can find address 13 in a 4-block, 2-byte per block cache.
 - The block address is $13 / 2 = 6$, so the index is then $6 \bmod 4 = 2$.
 - The block offset would be $13 \bmod 2 = 1$.

A diagram of a larger example cache

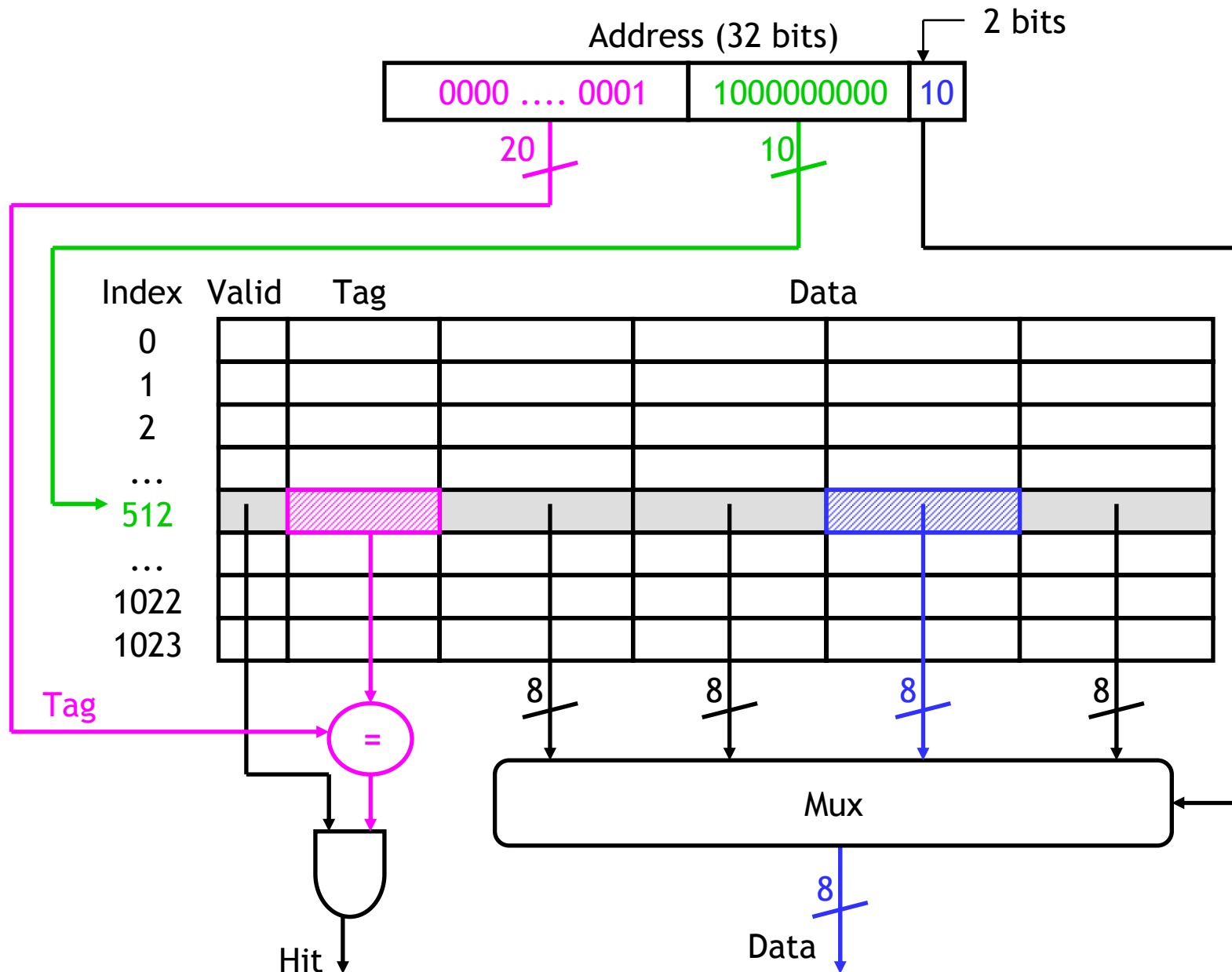
- Here is a cache with 1,024 blocks of 4 bytes each, and 32-bit memory addresses.



A larger example cache mapping

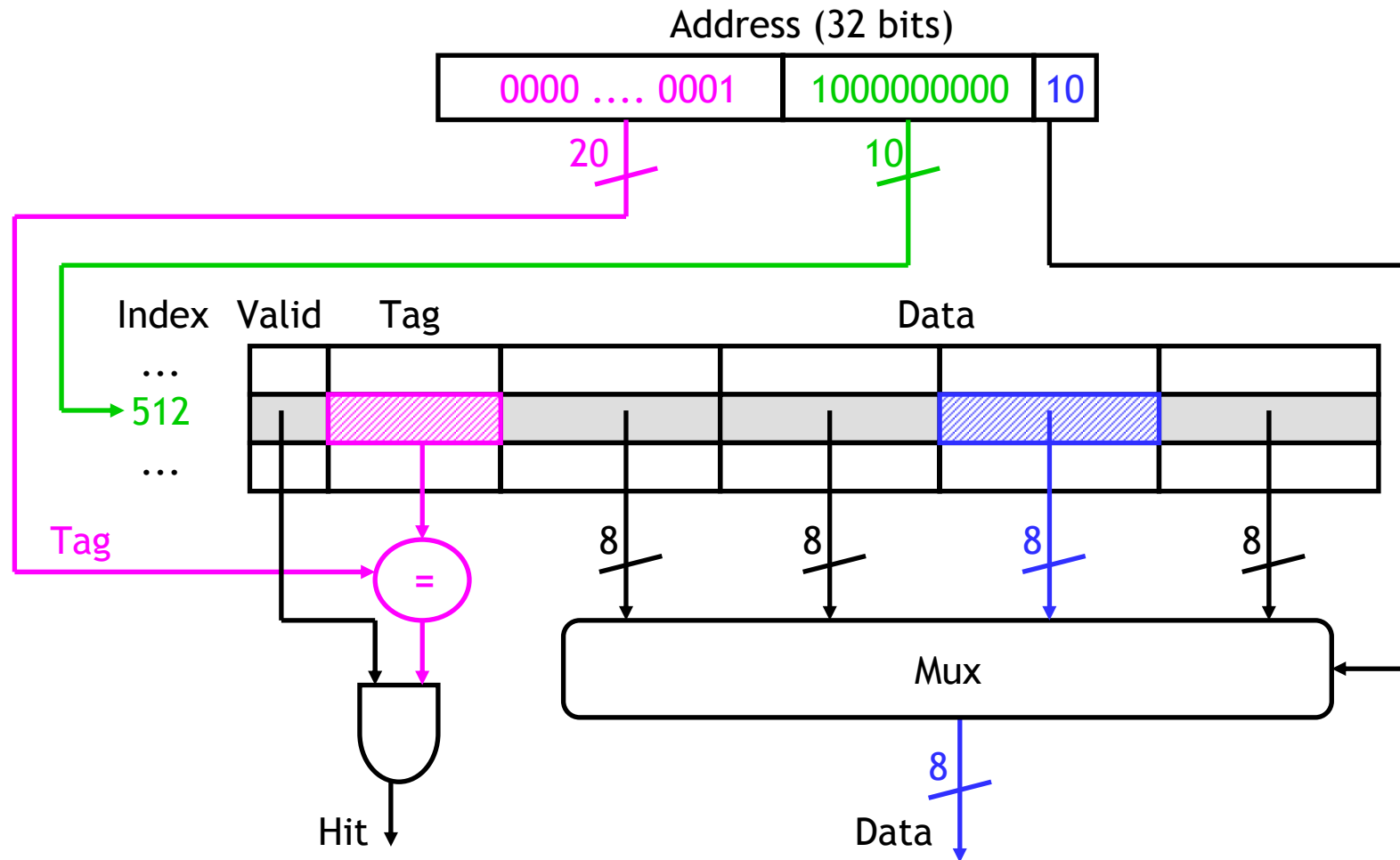
- Where would the byte from memory address 6146 be stored in this direct-mapped 2^{10} -block cache with 2^2 -byte blocks?
- We can determine this with the binary force.
 - 6146 in binary is 00...01 1000 0000 00 10.
 - The lowest 2 bits, 10, mean this is the second byte in its block.
 - The next 10 bits, 1000000000, are the block number itself (512).
- Equivalently, you could use your arithmetic mojo instead.
 - The block offset is $6146 \bmod 4$, which equals 2.
 - The block address is $6146/4 = 1536$, so the index is $1536 \bmod 1024$, or 512.

A larger diagram of a larger example cache mapping



What goes in the rest of that cache block?

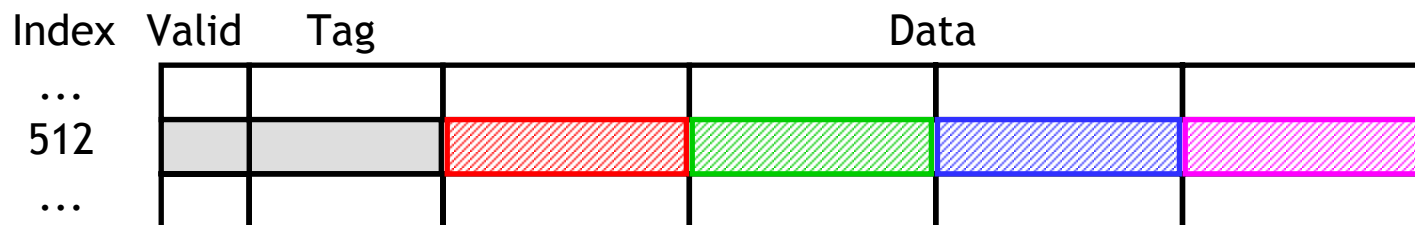
- The other three bytes of that cache block come from the same memory block, whose addresses must all have the same index (1000000000) and the same tag (00...01).



The rest of that cache block

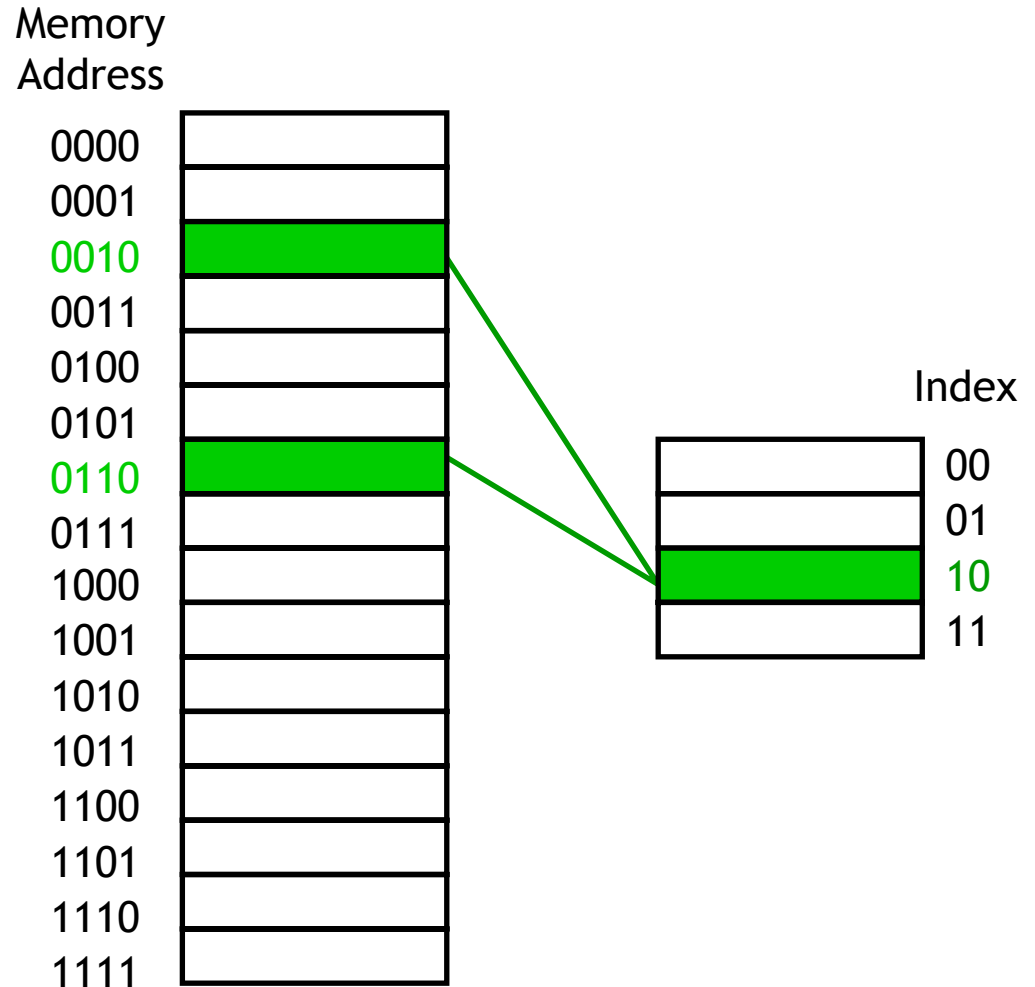
- Again, byte i of a memory block is stored into byte i of the corresponding cache block.
 - In our example, memory block 1536 consists of byte addresses 6144 to 6147. So bytes 0-3 of the cache block would contain data from address 6144, 6145, 6146 and 6147 respectively.
 - You can also look at the lowest 2 bits of the memory address to find the block offsets.

Block offset	Memory address	Decimal
00	00..01 1000000000 00	6144
01	00..01 1000000000 01	6145
10	00..01 1000000000 10	6146
11	00..01 1000000000 11	6147



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.

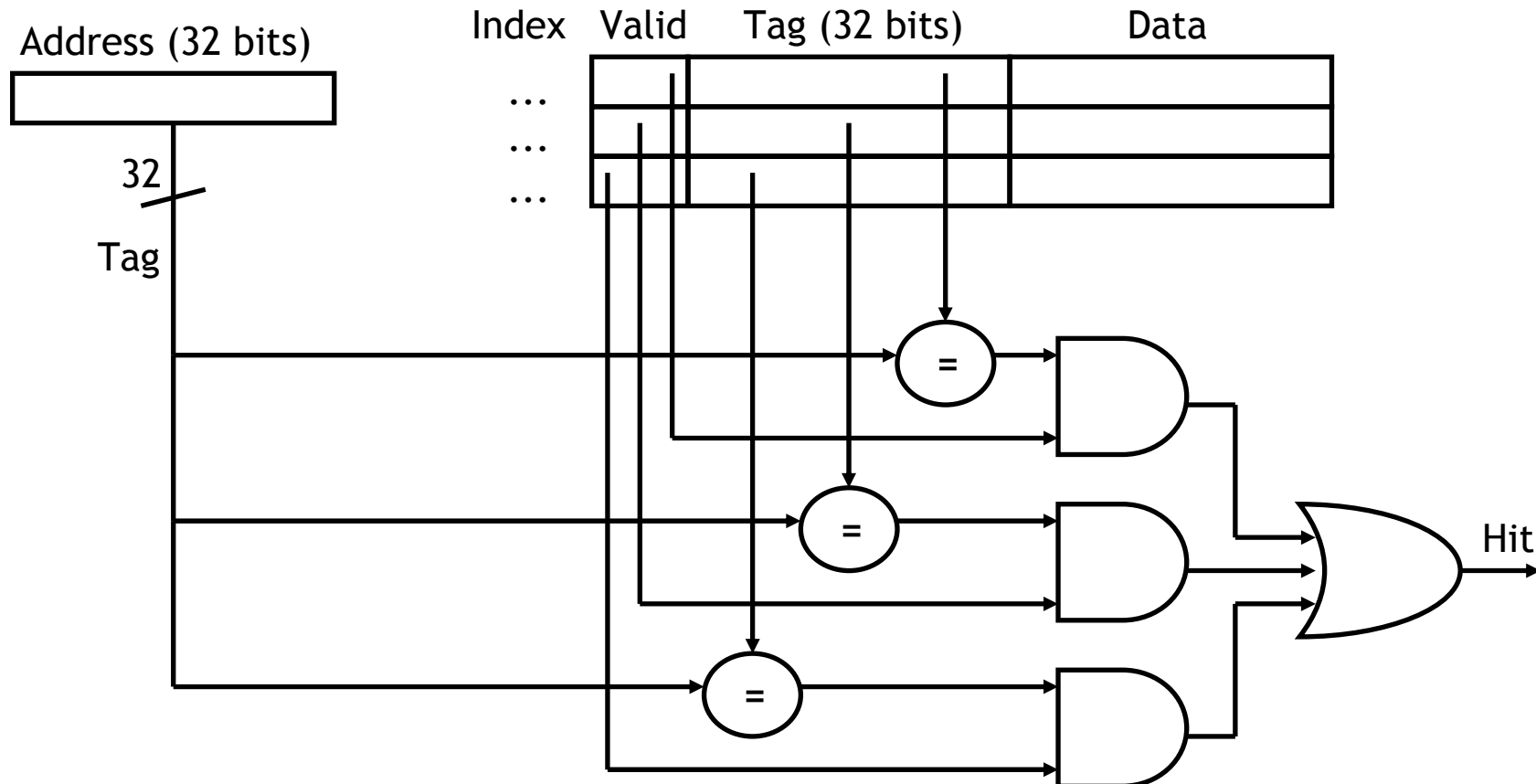


A fully associative cache

- 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

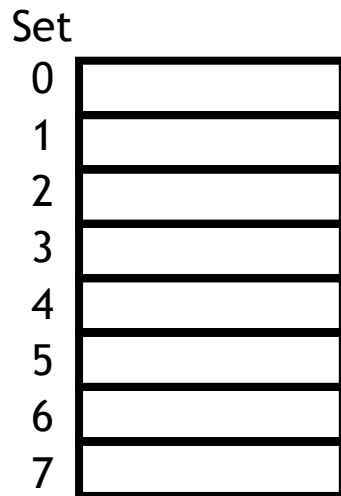
- However, a fully associative cache is expensive to implement.
 - Because there is no index field in the address anymore, the *entire* address must be used as the tag, increasing the total cache size.
 - Data could be anywhere in the cache, so we must check the tag of *every* cache block. That's a lot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



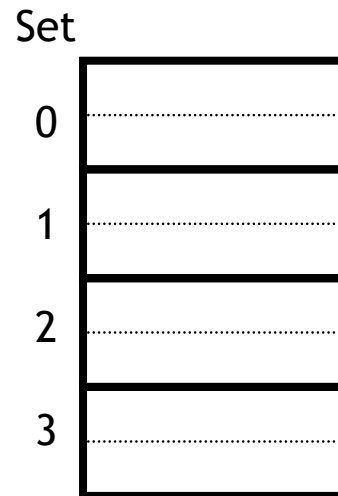
Set associativity

- 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.

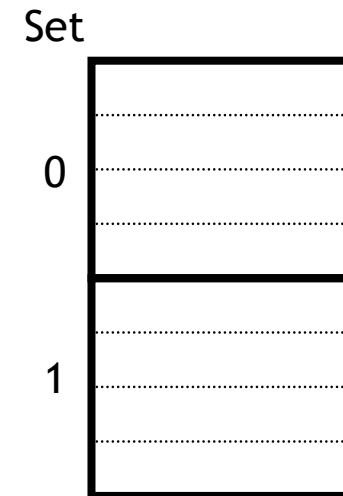
1-way associativity
8 sets, 1 block each



2-way associativity
4 sets, 2 blocks each

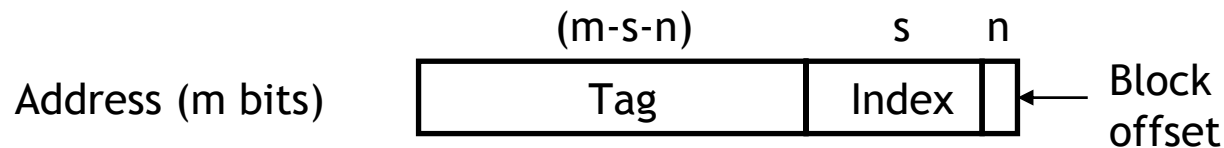


4-way associativity
2 sets, 4 blocks each



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.

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

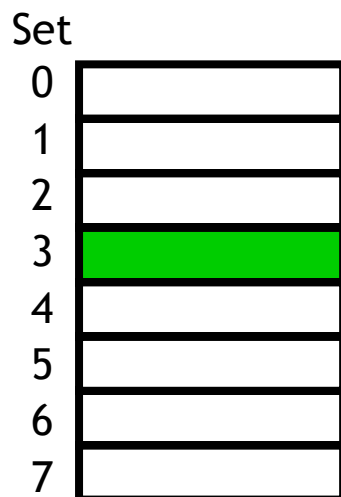
$$\text{Block Address} = \text{Memory Address} / 2^n$$

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

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**.
- 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.

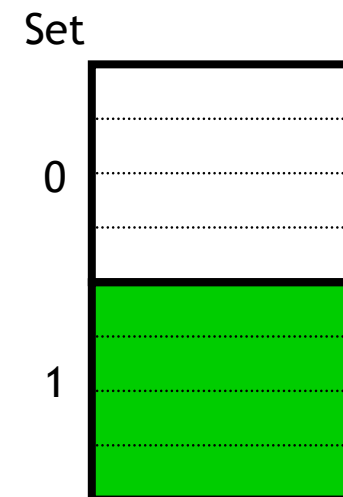
1-way associativity
8 sets, 1 block each



2-way associativity
4 sets, 2 blocks each



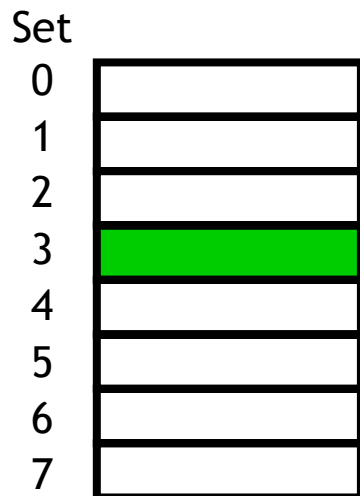
4-way associativity
2 sets, 4 blocks each



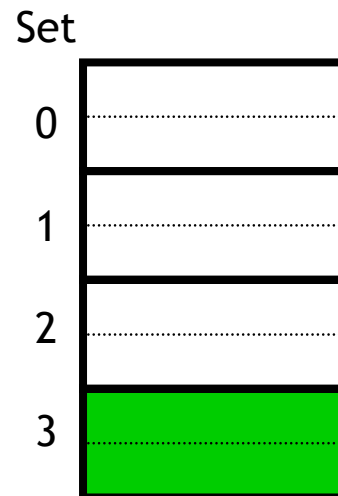
Block replacement

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

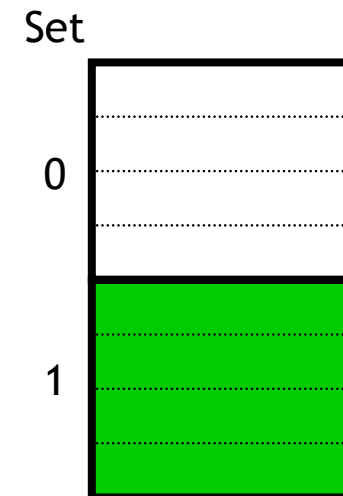
1-way associativity
8 sets, 1 block each



2-way associativity
4 sets, 2 blocks each

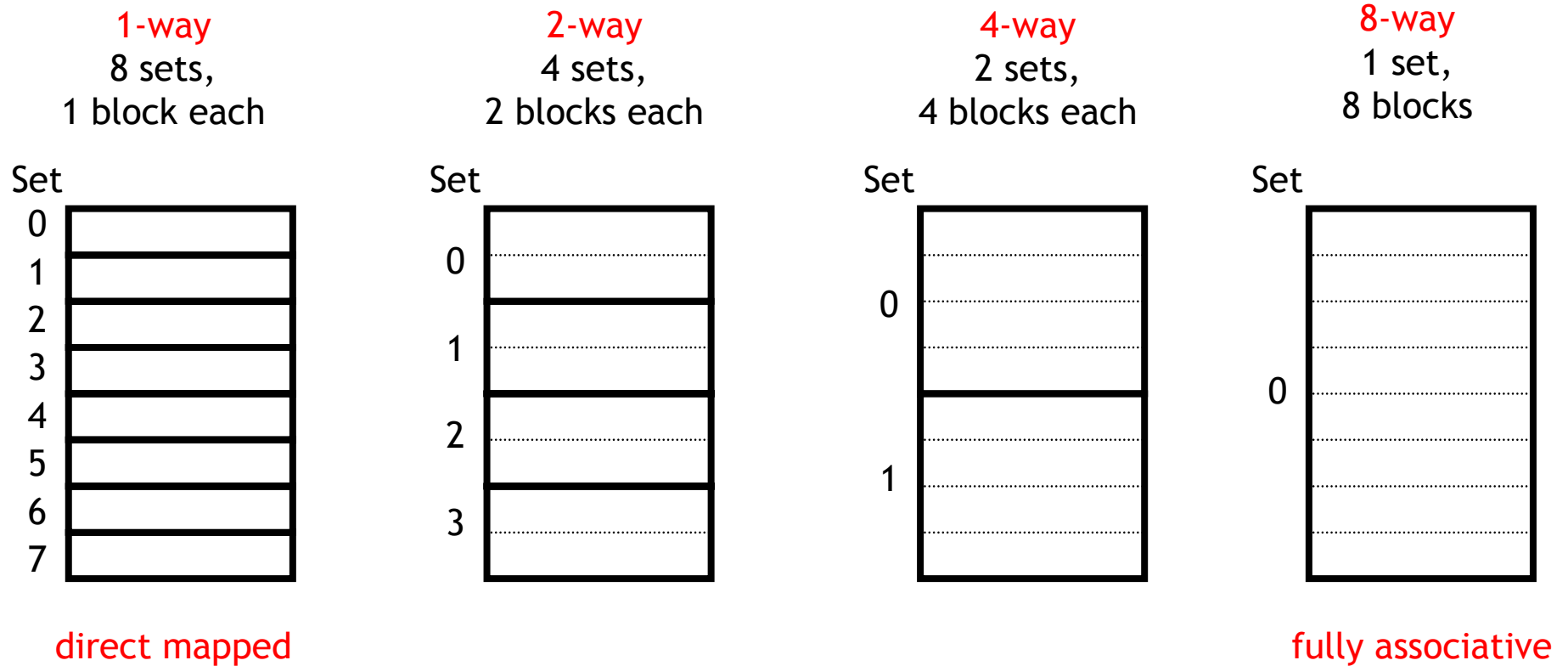


4-way associativity
2 sets, 4 blocks each



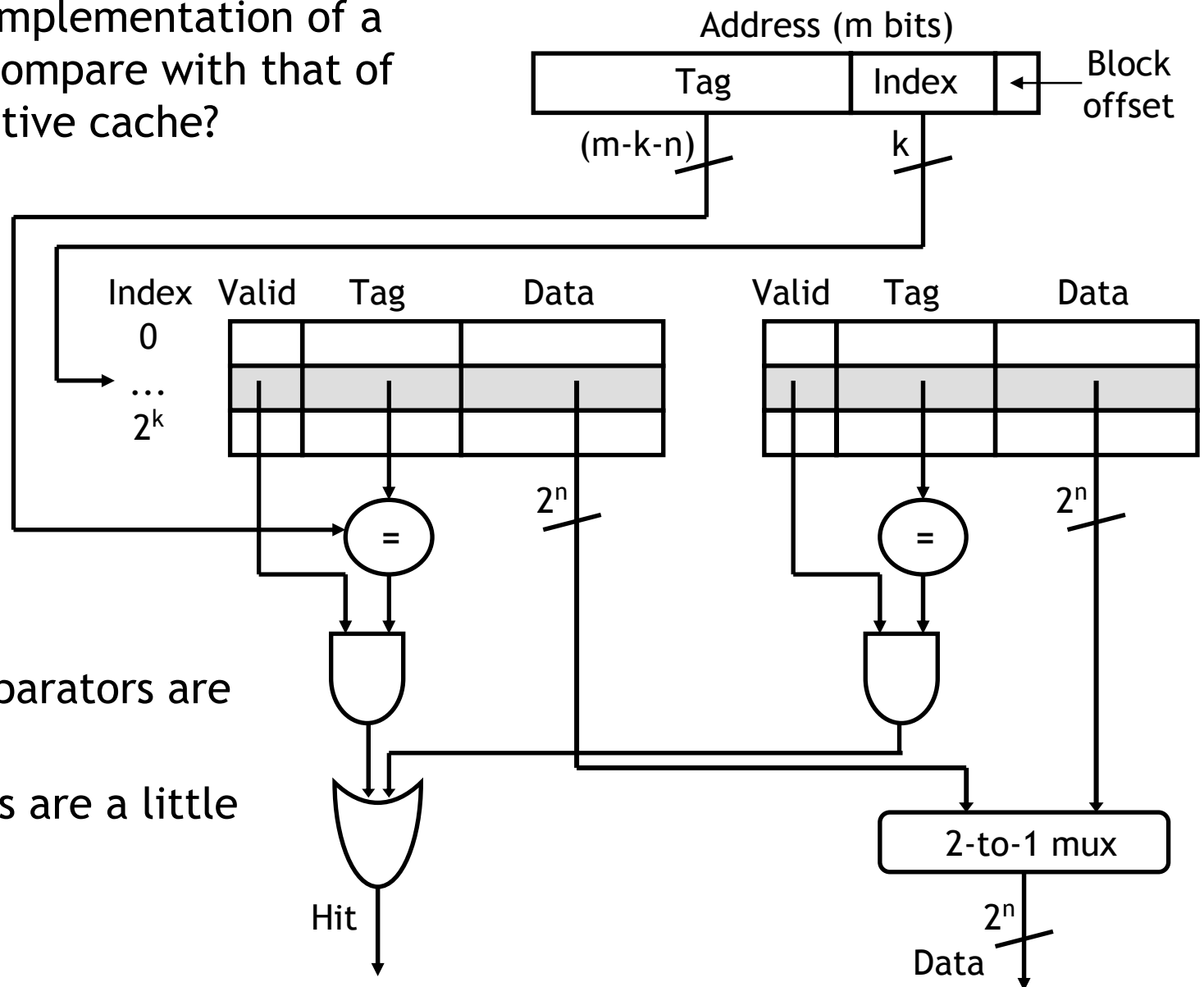
Set associative caches are a general idea

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