Lecture 15

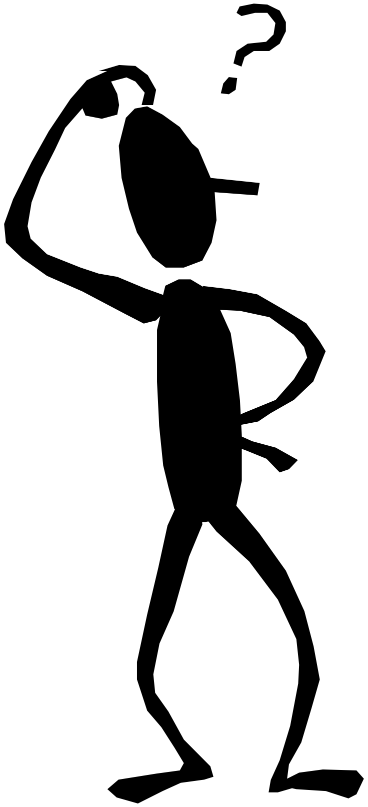
- Today:
  - How do caches work?

A simple cache design

- Caches are divided into blocks, which may be of various sizes.
  - The number of blocks in a cache is usually a power of 2.
  - For now we'll say that each block contains one byte. This won't take advantage of spatial locality, but we'll do that next time.
- Here is an example cache with eight blocks, each holding one byte.

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

Four important questions

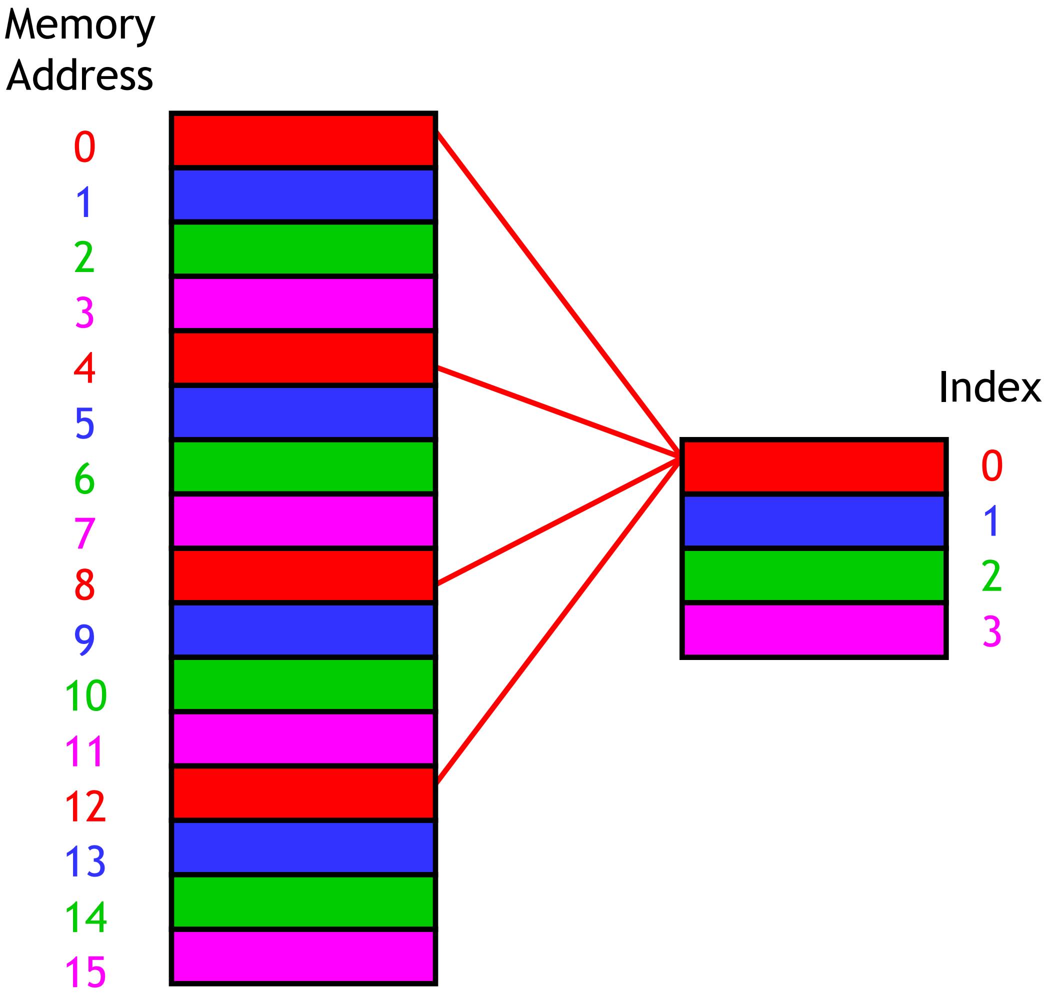


1. When we copy a block of data from main memory to the cache, where exactly should we put it?
2. How can we tell if a word is already in the cache, or if it has to be fetched from main memory first?
3. Eventually, the small cache memory might fill up. To load a new block from main RAM, we'd have to 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!

Where should we put data in the cache?

- A direct-mapped cache is the simplest approach: each main memory address maps to exactly one cache block.
- For example, on the right is a 16-byte main memory and a 4-byte cache (four 1-byte blocks).
- Memory locations 0, 4, 8 and 12 all map to cache block 0.
- Addresses 1, 5, 9 and 13 map to cache block 1, etc.
- How can we compute this mapping?



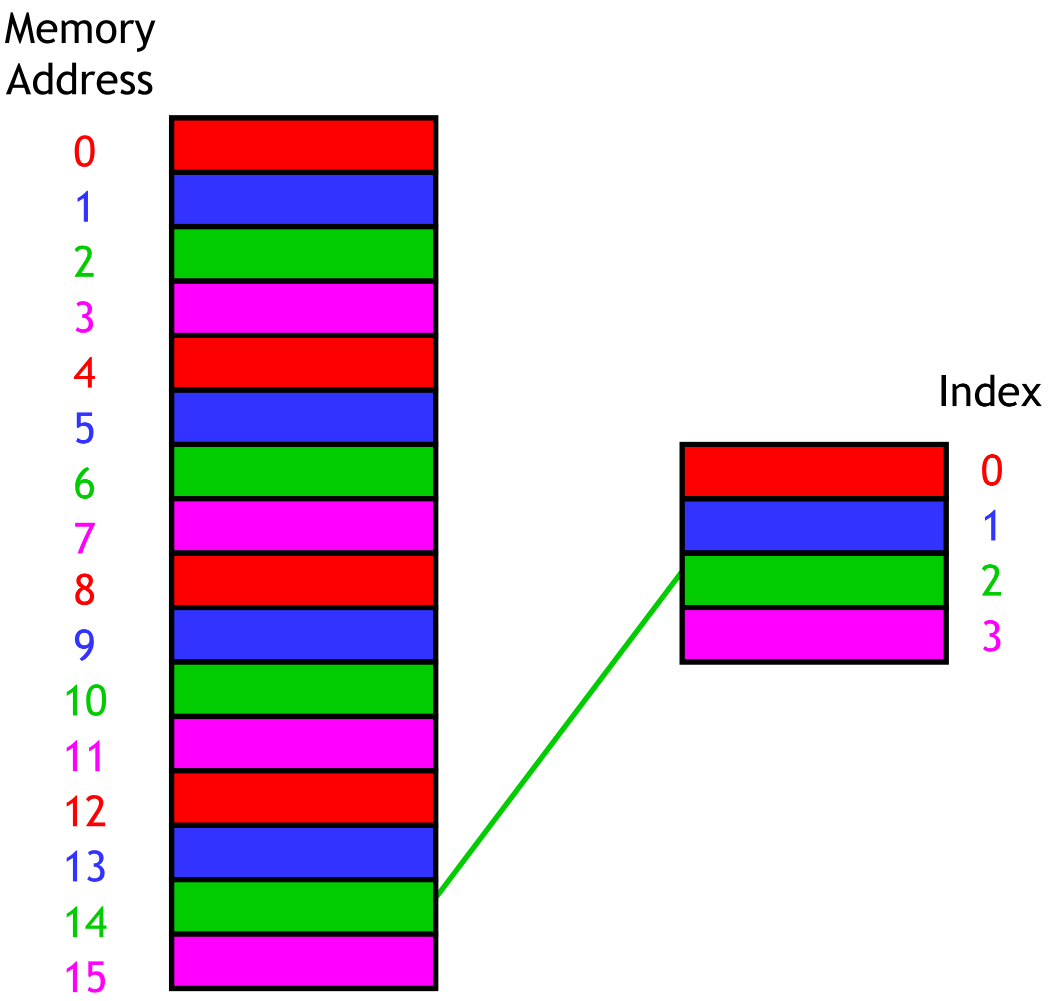
It’s all divisions…

- One way to figure out which cache block a particular memory address should go to is to use the mod (remainder) operator.
- If the cache contains 2^k blocks, then the data at memory address *i* would go to cache block index

  $i \bmod 2^k$

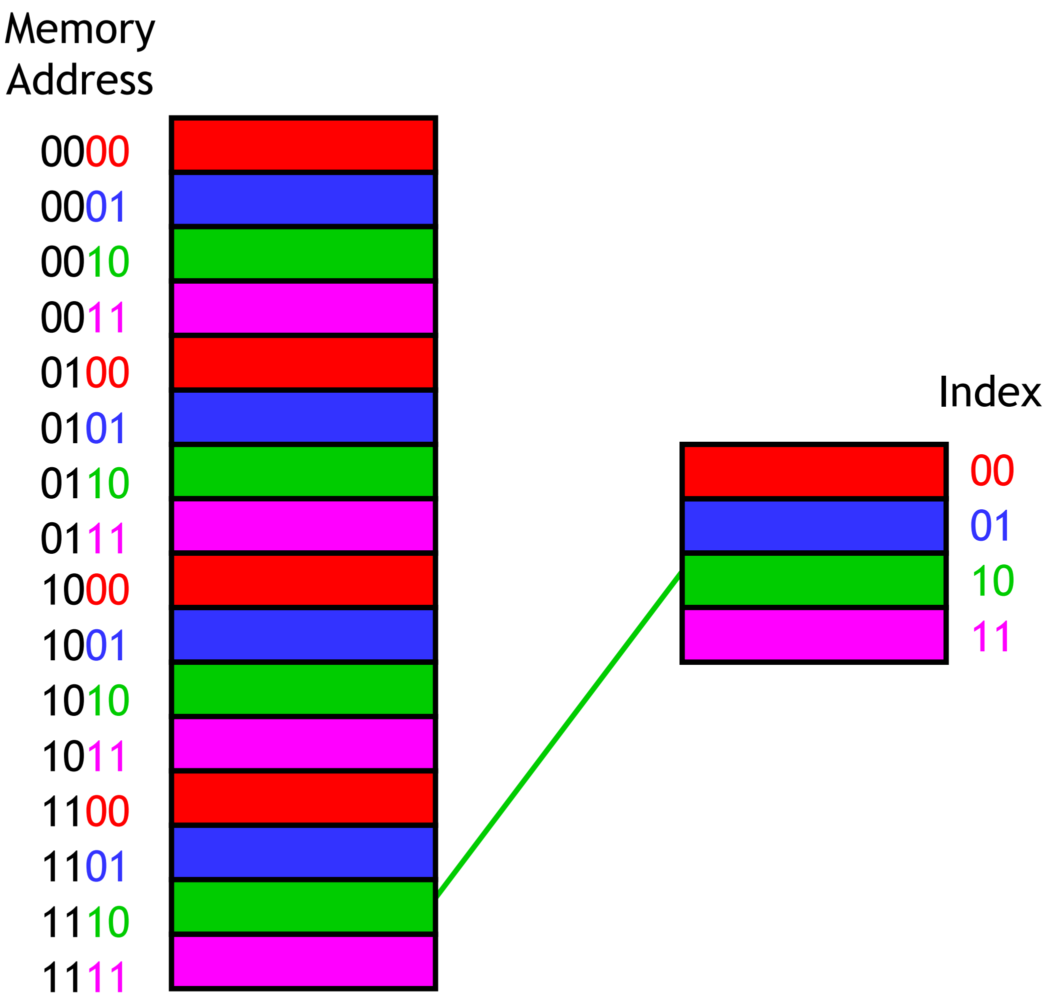
- For instance, with the four-block cache here, address 14 would map to cache block 2.

  14 mod 4 = 2



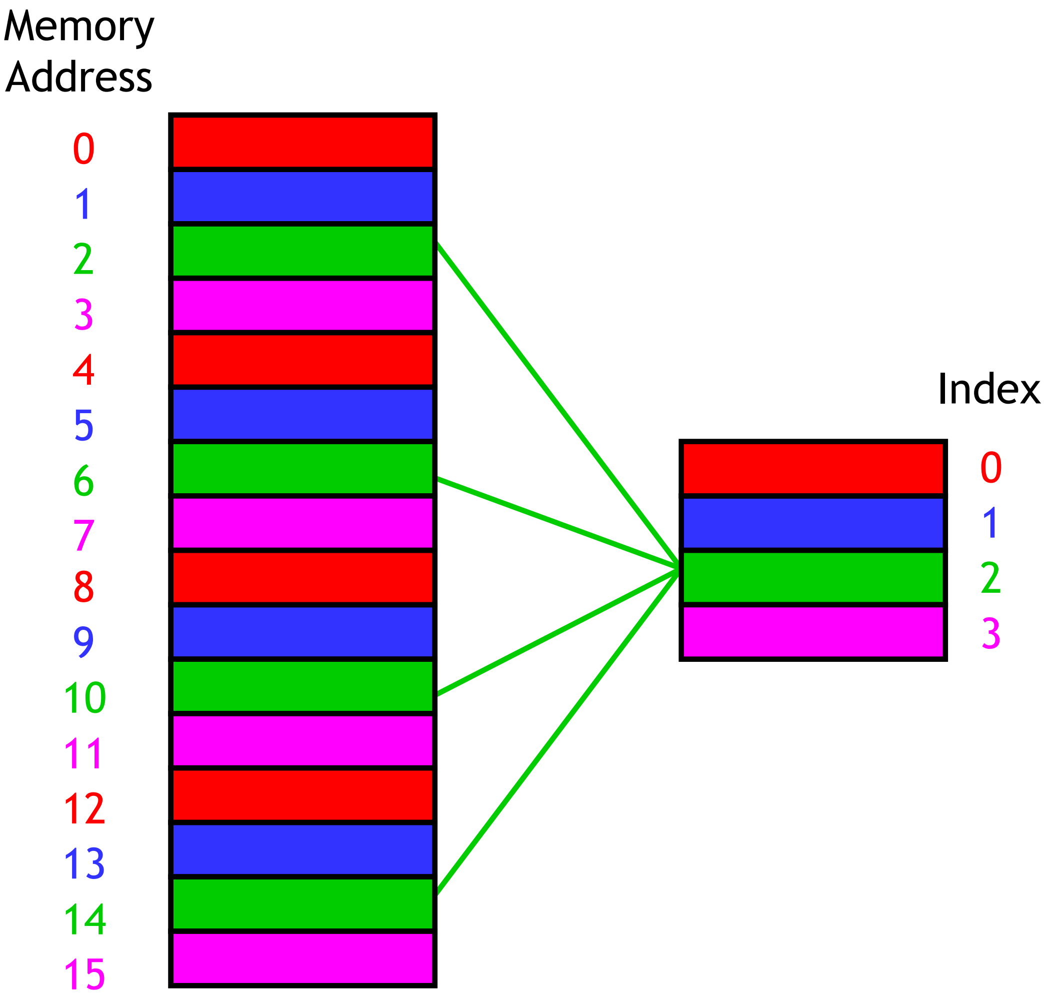
…or least-significant bits

- An equivalent way to find the placement of a memory address in the cache is to look at the least significant *k* bits of the address.
- With our four-byte cache we would inspect the two least significant bits of our memory addresses.
- Again, you can see that address 14 (1110 in binary) maps to cache block 2 (10 in binary).
- Taking the least *k* bits of a binary value is the same as computing that value mod 2^k.



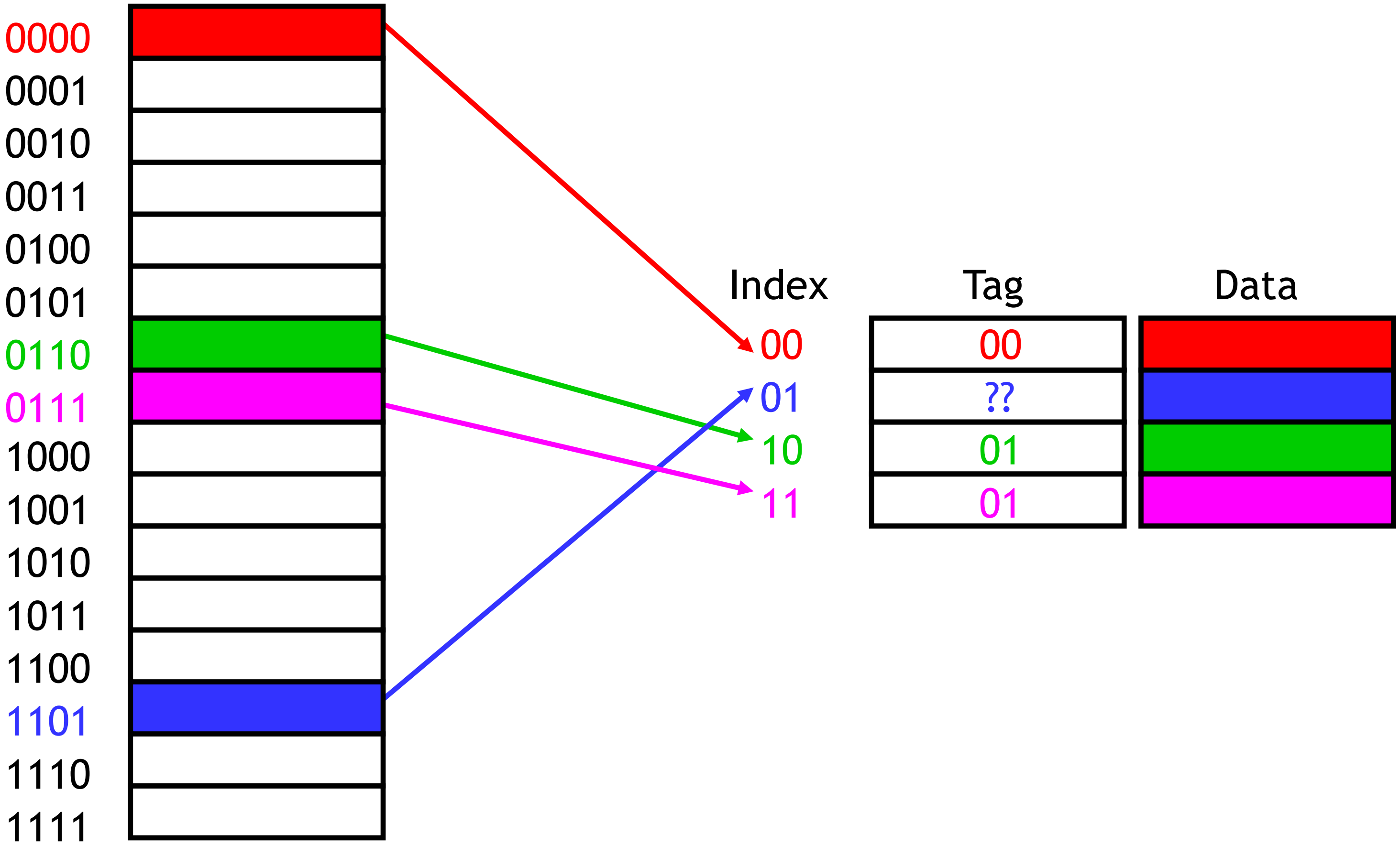
How can we find data in the cache?

- The second question was how to determine whether or not the data we're interested in is already stored in the cache.
- If we want to read memory address *i*, we can use the mod trick to determine which cache block would contain *i*.
- But other addresses might *also* map to the same cache block. How can we distinguish between them?
- For instance, cache block 2 could contain data from addresses 2, 6, 10 *or* 14.



Adding tags

- We need to add tags to the cache, which supply the rest of the address bits to let us distinguish between different memory locations that map to the same cache block.

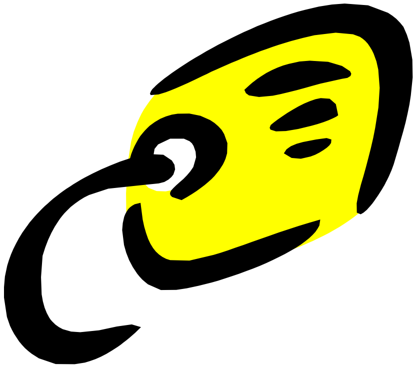


Index	Tag	Data
00	00	🟥
01	??	🟦
10	01	🟩
11	01	🟪

Figuring out what’s in the cache

- Now we can tell exactly which addresses of main memory are stored in the cache, by concatenating the cache block tags with the block indices.

Index	Tag	Data		Main memory address in cache block
00	00	🟥	→	00 + 00 = 0000
01	11	🟦	→	11 + 01 = 1101
10	01	🟩	→	01 + 10 = 0110
11	01	🟪	→	01 + 11 = 0111



One more detail: the valid bit

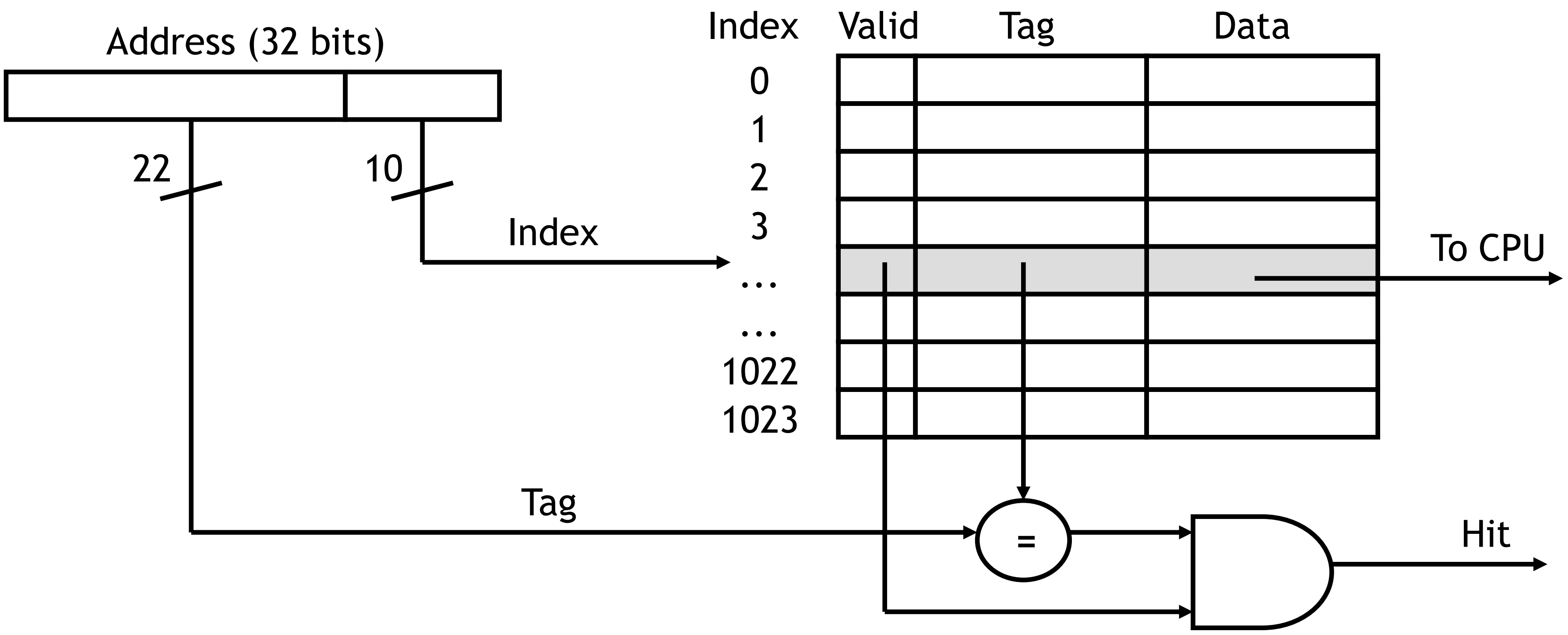
- When started, the cache is empty and does not contain valid data.
- We should account for this by adding a valid bit for each cache block.
  - When the system is initialized, all the valid bits are set to 0.
  - When data is loaded into a particular cache block, the corresponding valid bit is set to 1.

Index	Valid Bit	Tag	Data	Main memory address in cache block
00	1	00	🟥	00 + 00 = 0000
01	0	11	🟦	Invalid
10	0	01	🟩	???
11	1	01	🟪	???

- So the cache contains more than just copies of the data in memory; it also has bits to help us find data within the cache and verify its validity.

What happens on a cache hit

- When the CPU tries to read from memory, the address will be sent to a cache controller.
  - The lowest *k* bits of the address will index a block in the cache.
  - If the block is valid and the tag matches the upper ($m - k$) bits of the *m*-bit address, then that data will be sent to the CPU.
- Here is a diagram of a 32-bit memory address and a 2^{10}-byte cache.

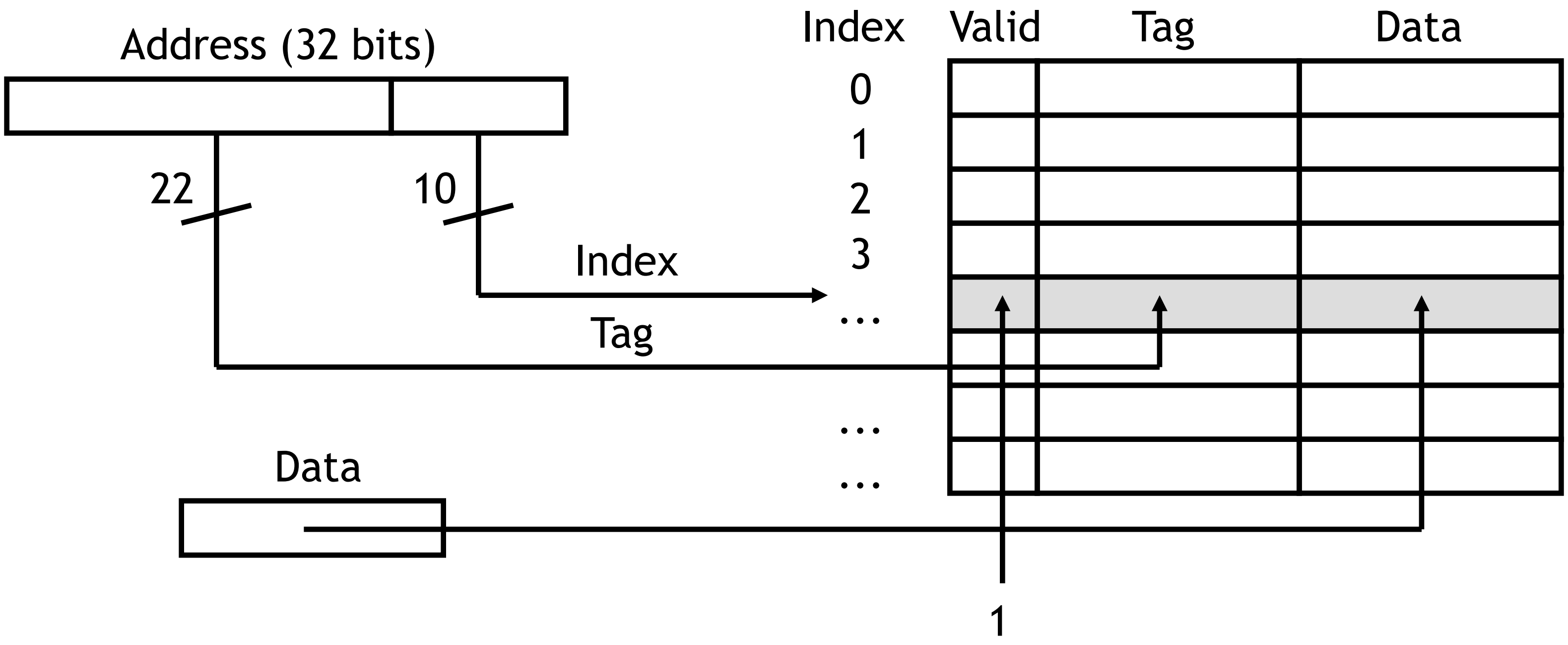


What happens on a cache miss

- The delays that we’ve been assuming for memories (e.g., 2ns) are really assuming cache hits.
  - If our CPU implementations accessed main memory directly, their cycle times would have to be much larger.
  - Instead we assume that most memory accesses will be cache hits, which allows us to use a shorter cycle time.
- However, a much slower main memory access is needed on a cache miss. The simplest thing to do is to stall the pipeline until the data from main memory can be fetched (and also copied into the cache).

Loading a block into the cache

- After data is read from main memory, putting a copy of that data into the cache is straightforward.
  - The lowest *k* bits of the address specify a cache block.
  - The upper (*m* − *k*) address bits are stored in the block’s tag field.
  - The data from main memory is stored in the block’s data field.
  - The valid bit is set to 1.



What if the cache fills up?

- Our third question was what to do if we run out of space in our cache, or if we need to reuse a block for a different memory address.
- We answered this question implicitly on the last page!
  - A miss causes a new block to be loaded into the cache, automatically overwriting any previously stored data.
  - This is a least recently used replacement policy, which assumes that older data is less likely to be requested than newer data.
- We'll see a few other policies next.

How big is the cache?

For a byte-addressable machine with 16-bit addresses with a cache with the following characteristics:

- It is direct-mapped
- Each block holds one byte
- The cache index is the four least significant bits

Two questions:

- How many blocks does the cache hold?
- How many bits of storage are required to build the cache (*e.g.*, for the data array, tags, etc.)?

More cache organizations

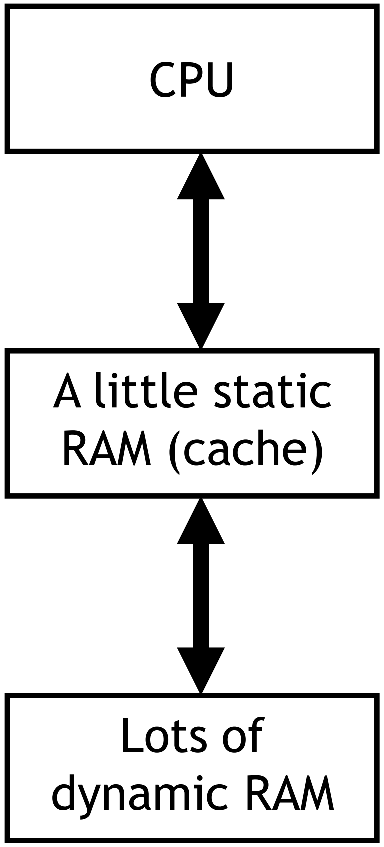
Now we'll explore some alternate cache organizations.

- How can we take advantage of spatial locality too?
- How can we reduce the number of potential conflicts?

- We'll first motivate it with a brief discussion about cache performance.

Memory System Performance

- To examine the performance of a memory system, we need to focus on a couple of important factors.
  - How long does it take to send data from the cache to the CPU?
  - How long does it take to copy data from memory into the cache?
  - How often do we have to access main memory?
- There are names for all of these variables.
  - The hit time is how long it takes data to be sent from the cache to the processor. This is usually fast, on the order of 1-3 clock cycles.
  - The miss penalty is the time to copy data from main memory to the cache. This often requires dozens of clock cycles (at least).
  - The miss rate is the percentage of misses.



CPU

A little static RAM (cache)

Lots of dynamic RAM

Average memory access time

- The average memory access time, or AMAT, can then be computed.

  AMAT = Hit time + (Miss rate x Miss penalty)

  This is just averaging the amount of time for cache hits and the amount of time for cache misses.
- How can we improve the average memory access time of a system?
  - Obviously, a lower AMAT is better.
  - Miss penalties are usually much greater than hit times, so the best way to lower AMAT is to reduce the miss penalty *or* the miss rate.
- However, AMAT should only be used as a general guideline. Remember that execution time is still the best performance metric.

Performance example

- Assume that 33% of the instructions in a program are data accesses. The cache hit ratio is 97% and the hit time is one cycle, but the miss penalty is 20 cycles.

$$\begin{aligned} \text{AMAT} &= \text{Hit time} + (\text{Miss rate x Miss penalty}) \\ &= \\ &= \end{aligned}$$

- How can we reduce miss rate?

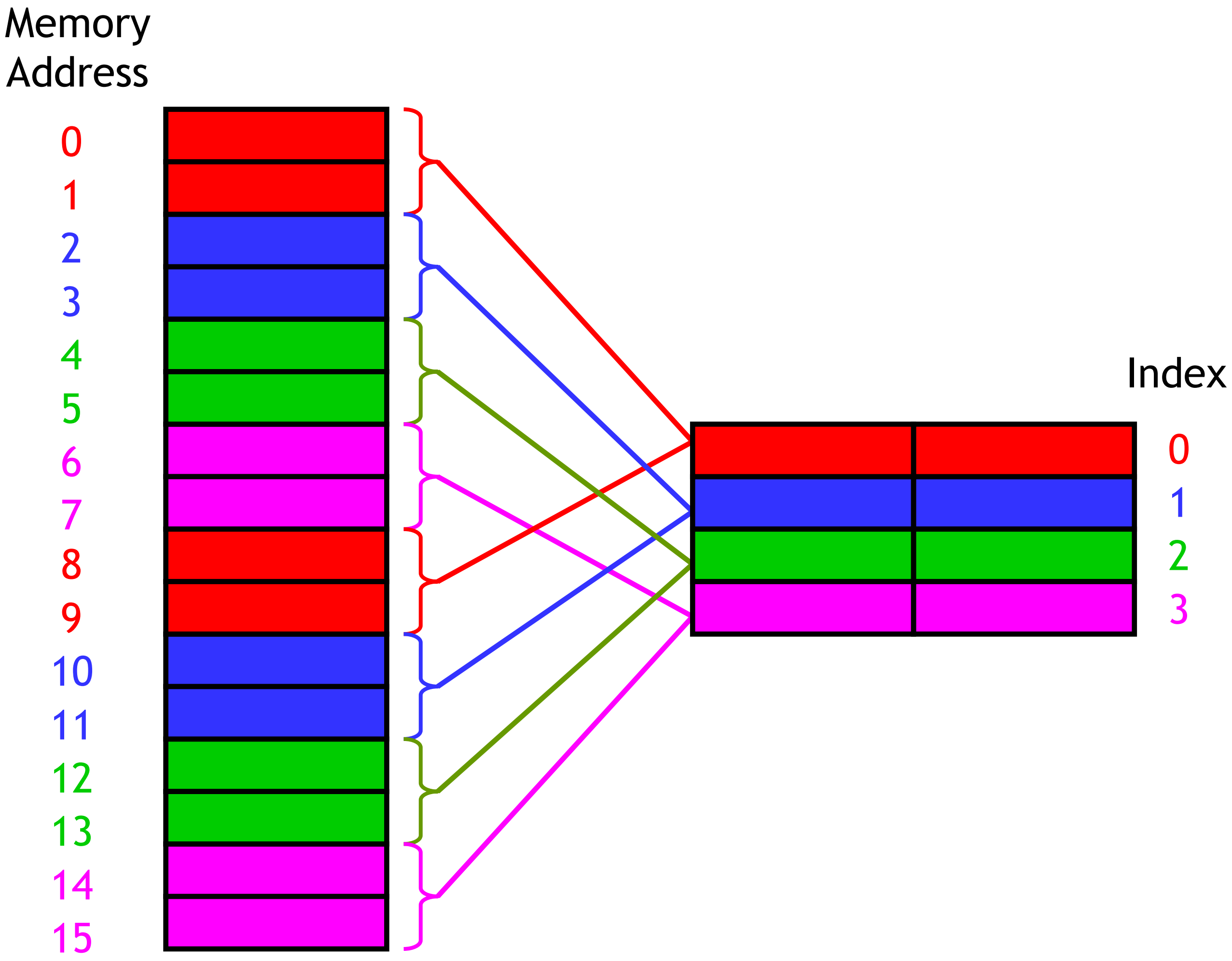
Spatial locality

- 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 can we do?

Spatial locality

- 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.

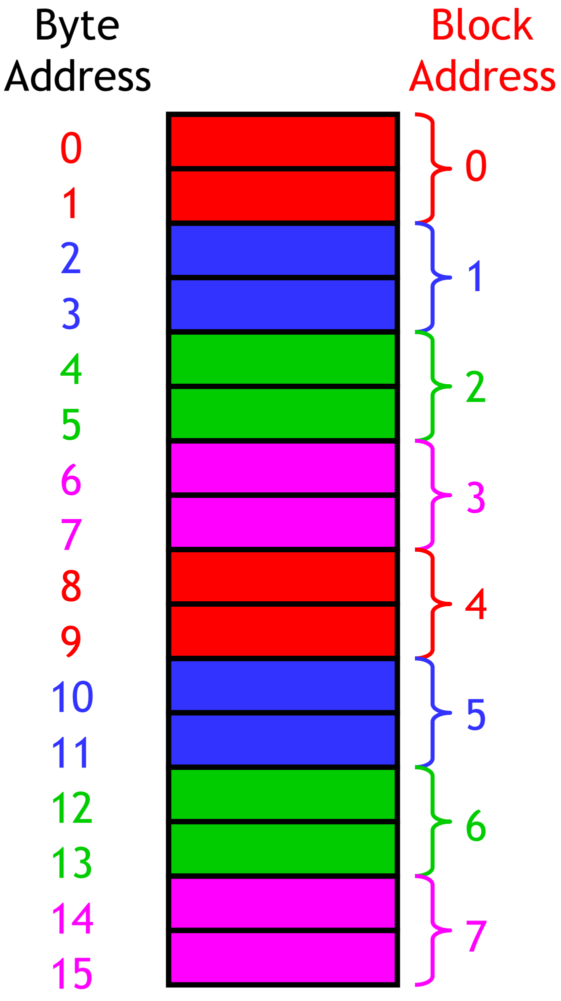


Block addresses

- 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

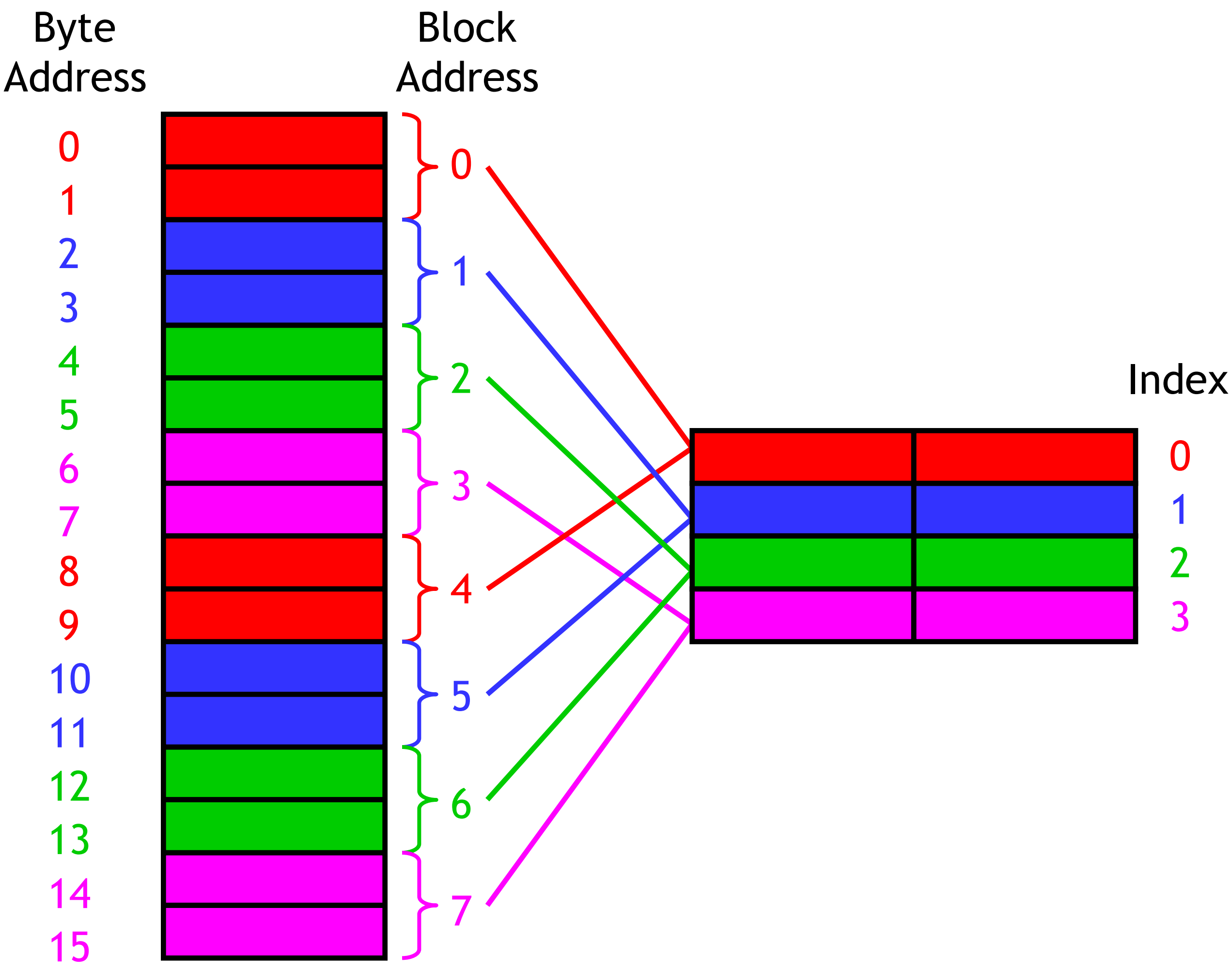
$$i\ /\ 2^n$$

- 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.



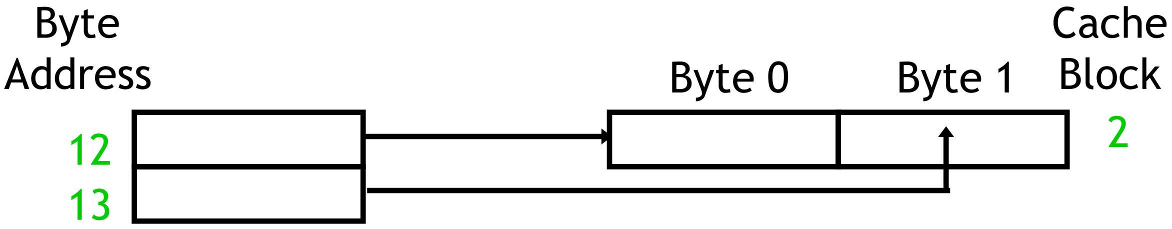
Cache mapping

- 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 mod 4 = 2.
- This corresponds to placing data from memory *byte* addresses 12 and 13 into cache block 2.



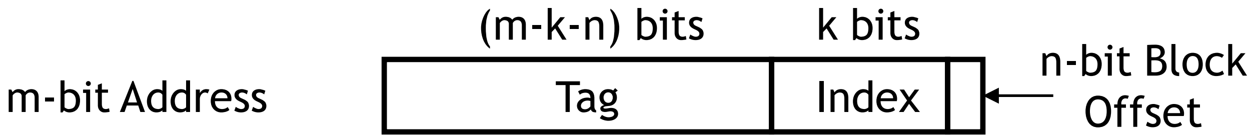
Data placement within a block

- 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.



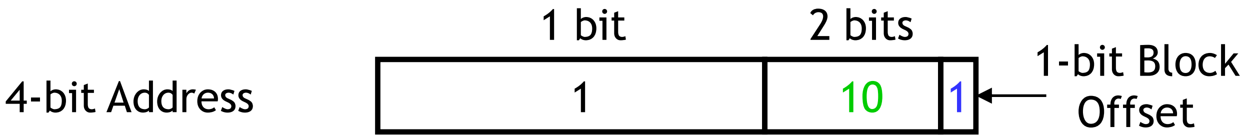
m-bit Address

(m-k-n) bits | k bits

Tag | Index |

n-bit Block Offset

- 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.



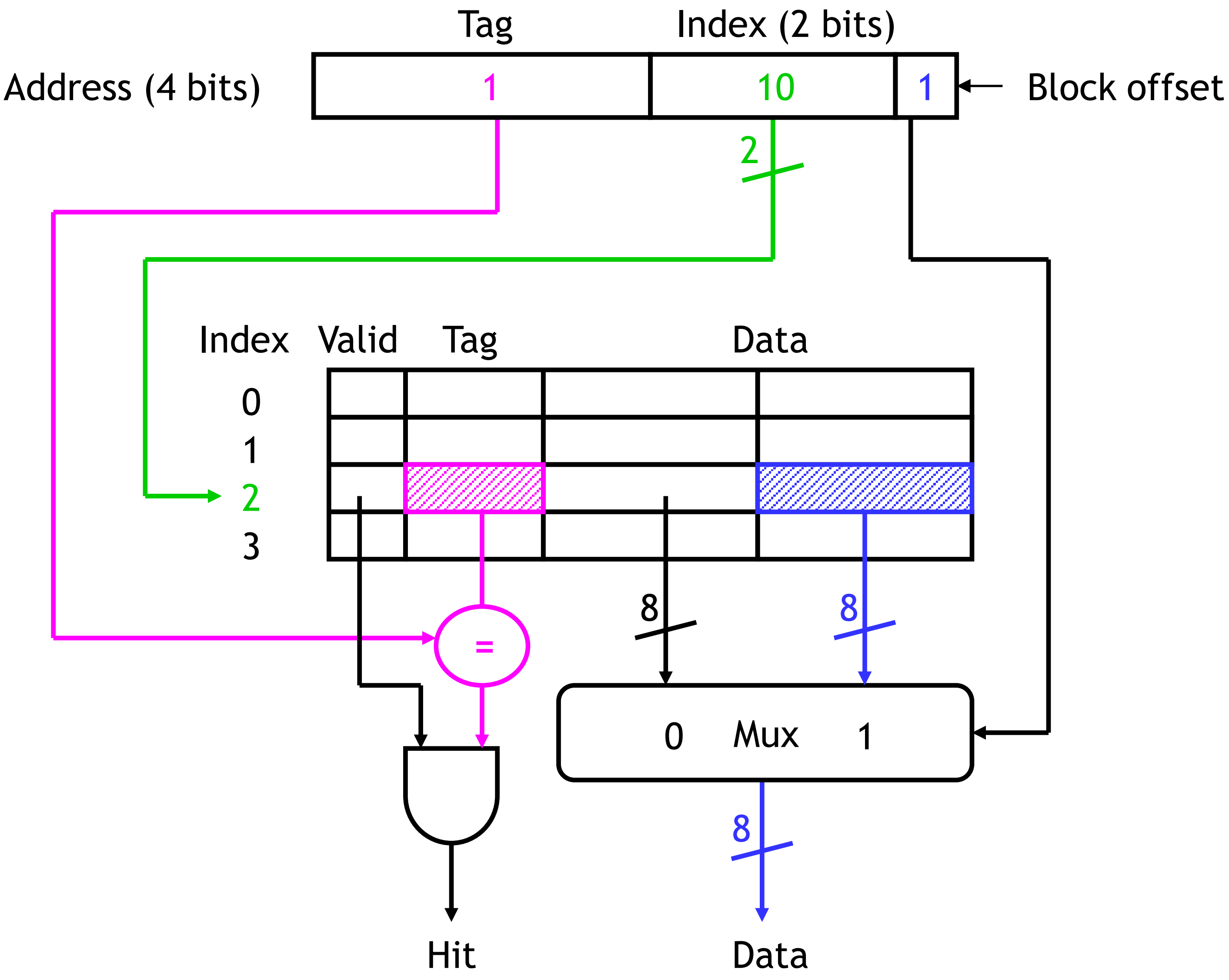
4-bit Address

1 bit | 2 bits

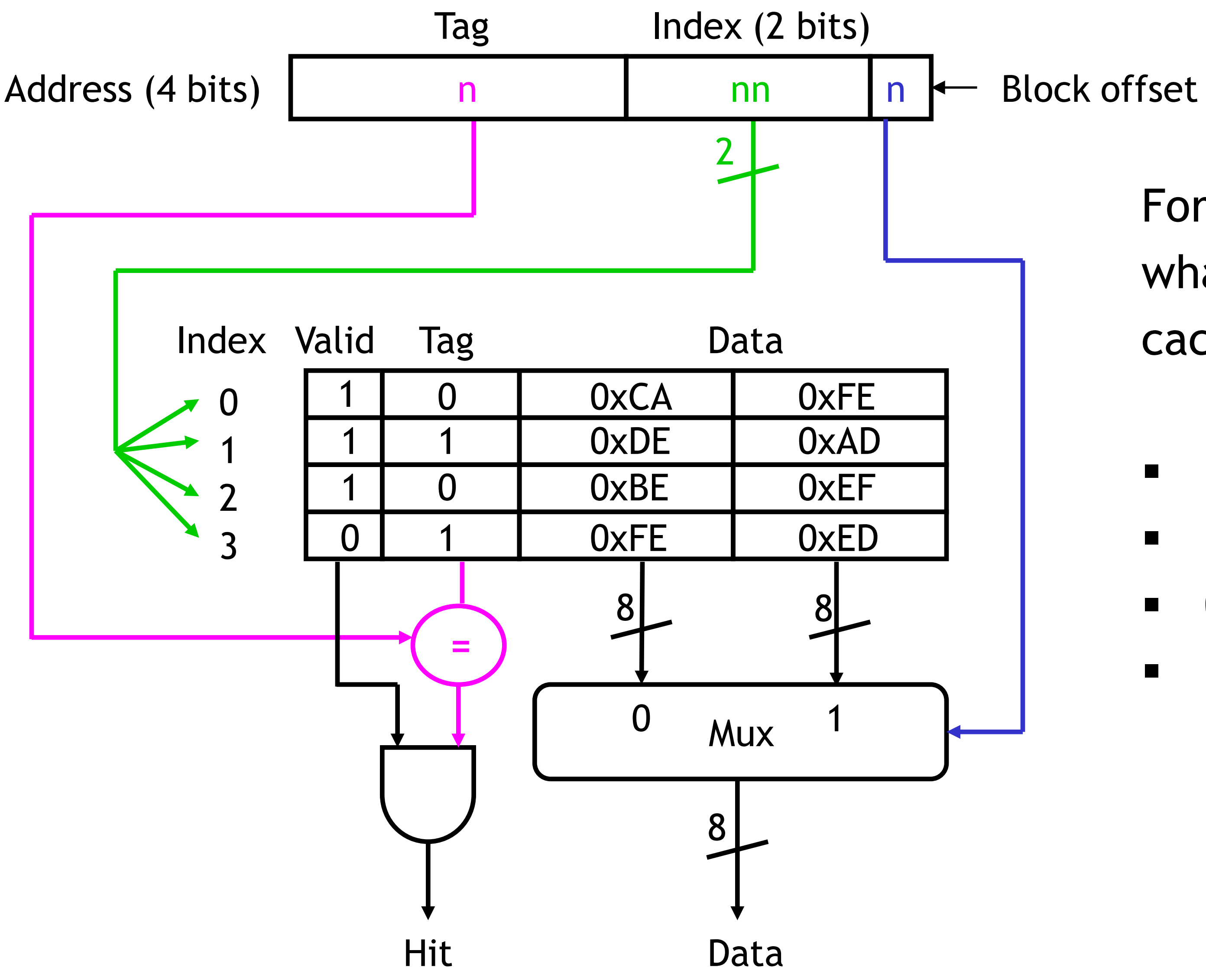
1 | 10 | 1

1-bit Block Offset

A picture


Tag
Index (2 bits)
Address (4 bits)
1
10
1
Block offset
2
Index
Valid
Tag
Data
0
1
2
3
8
8
=
0
Mux
1
8
Hit
Data

An exercise



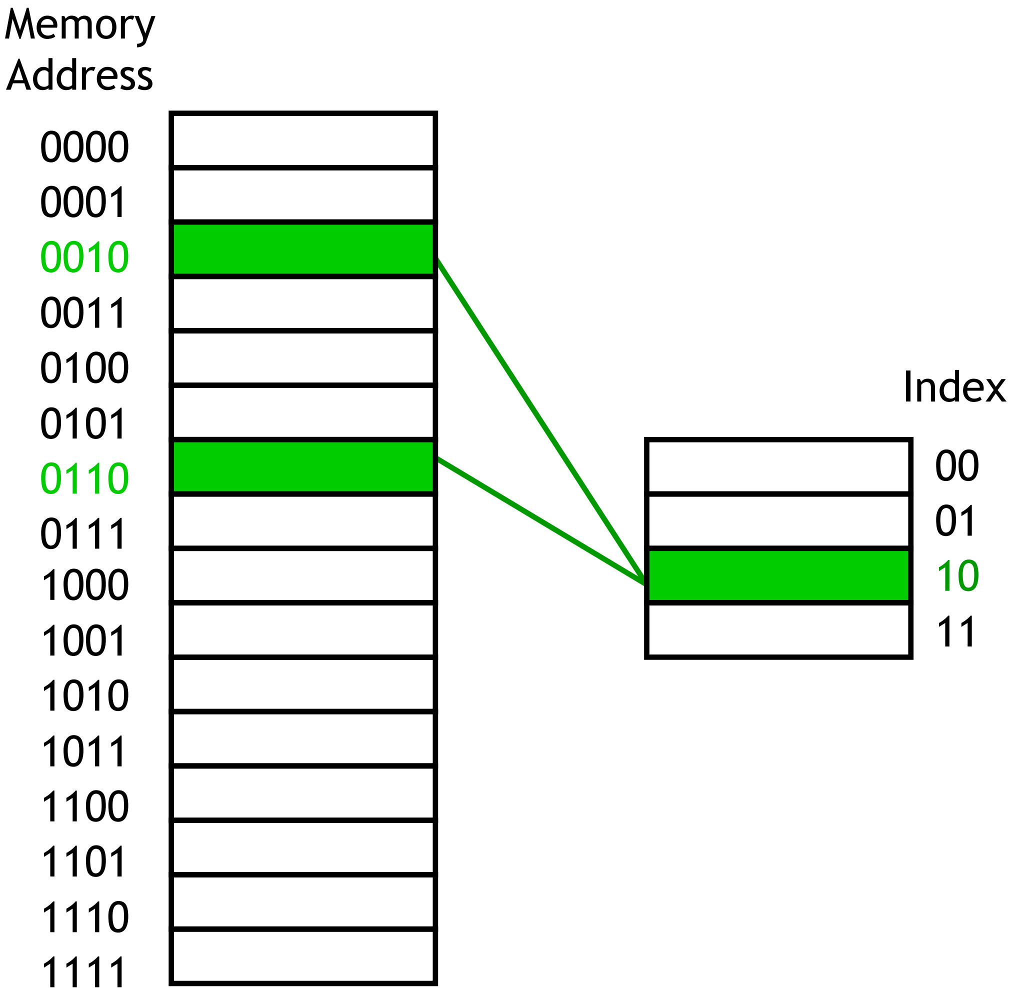
Index	Valid	Tag	Data	
0	1	0	0xCA	0xFE
1	1	1	0xDE	0xAD
2	1	0	0xBE	0xEF
3	0	1	0xFE	0xED

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

- 1010
- 1110
- 0001
- 1101

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, …?

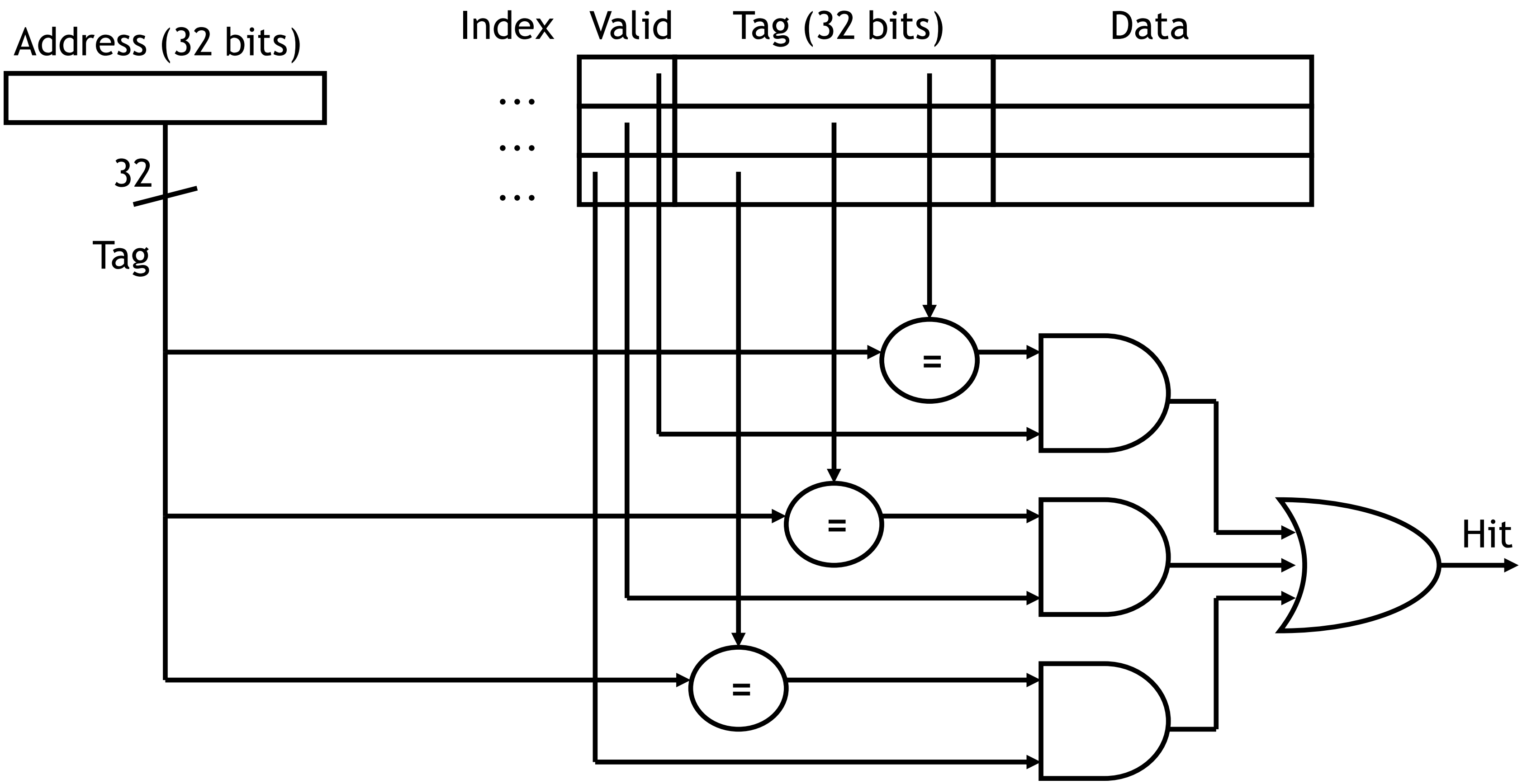


A fully associative cache

- 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.

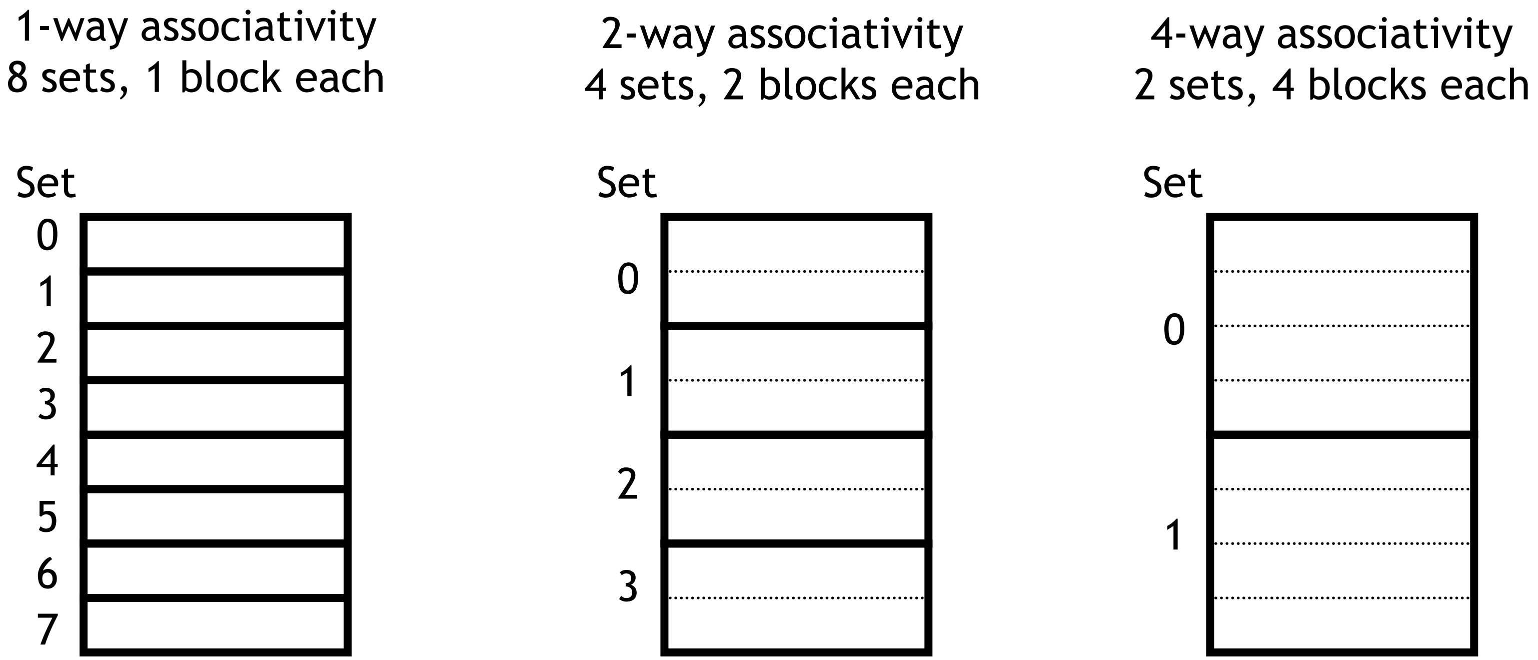
The price of full associativity

- 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.



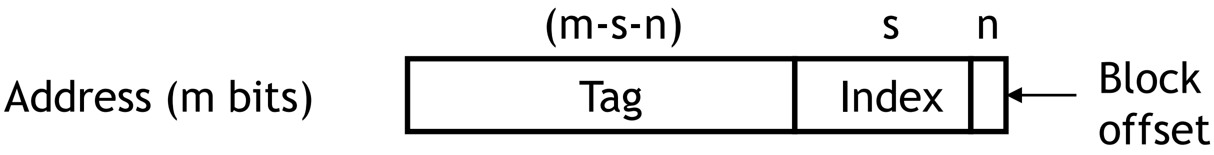
1-way associativity
8 sets, 1 block each

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.



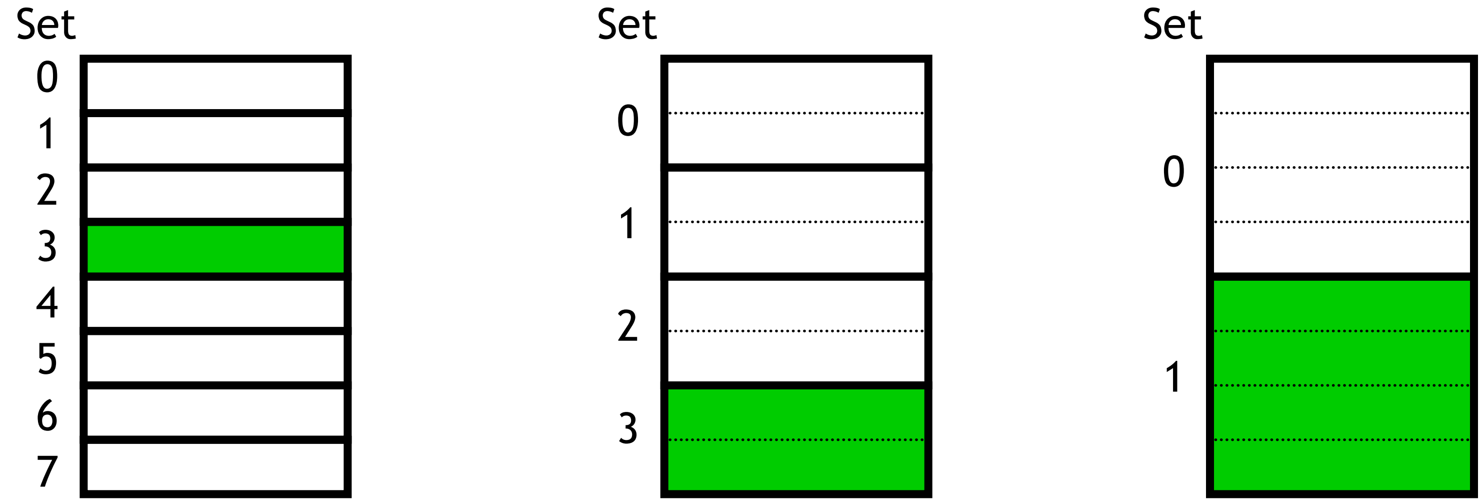
- Our arithmetic computations now compute a set index, to select a *set* within the cache instead of an individual block.

Block Offset = Memory Address mod 2^n

Block Address = Memory Address / 2^n
Set Index = Block Address mod 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

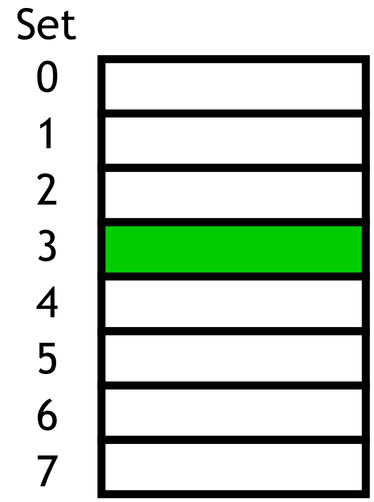
2-way associativity
4 sets, 2 blocks each

4-way associativity
2 sets, 4 blocks each

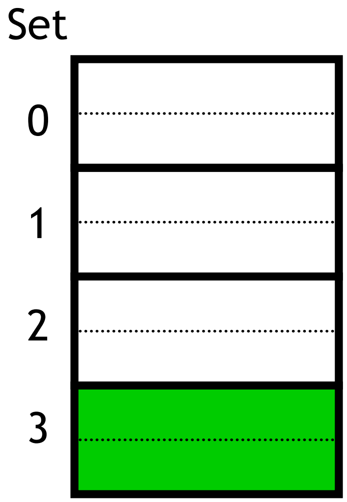
Block replacement

- 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.
- 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.

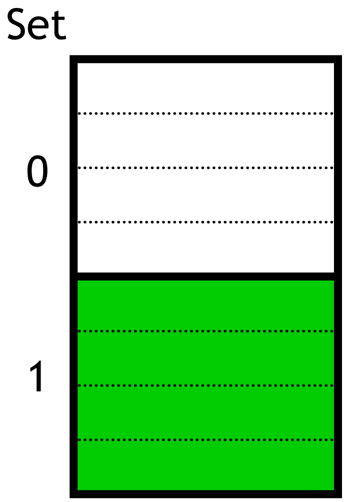
1-way associativity
8 sets, 1 block each



2-way associativity
4 sets, 2 blocks each

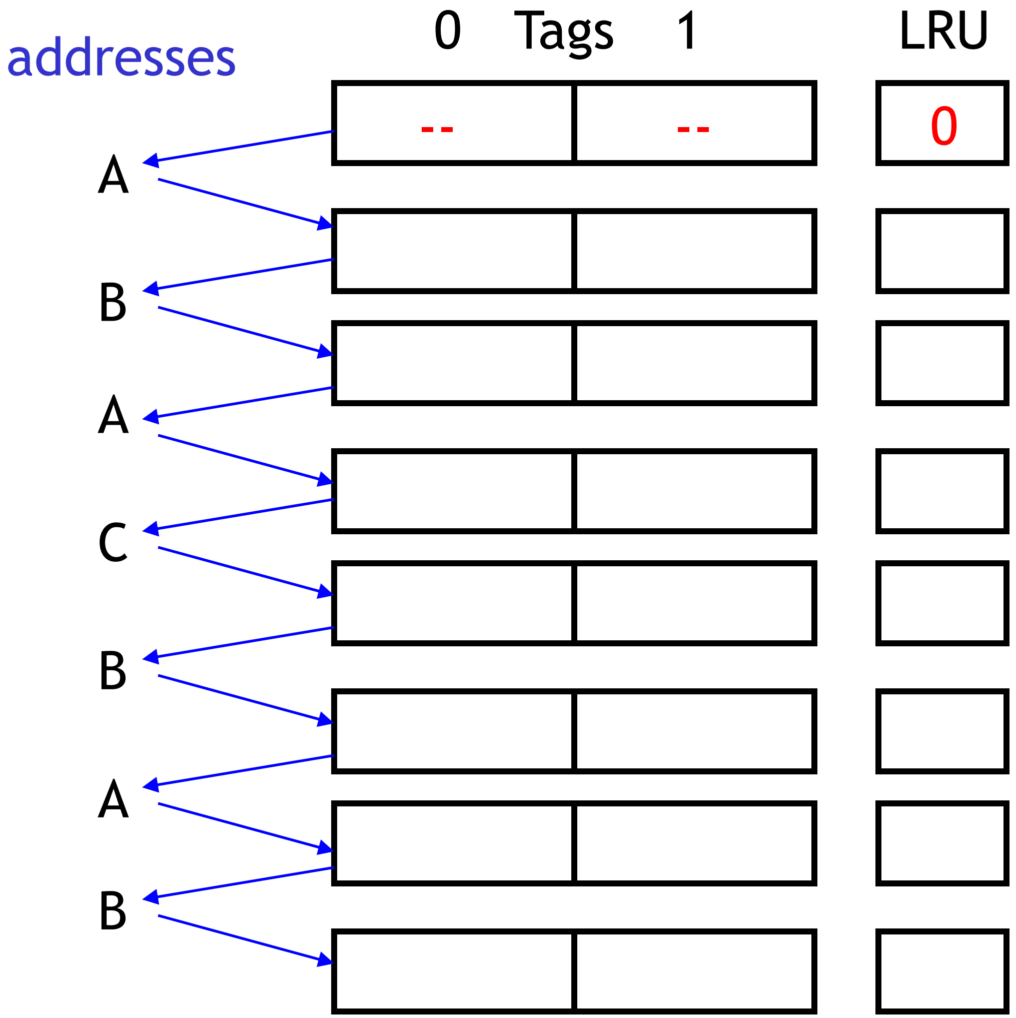


4-way associativity
2 sets, 4 blocks each



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



Set associative caches are a general idea

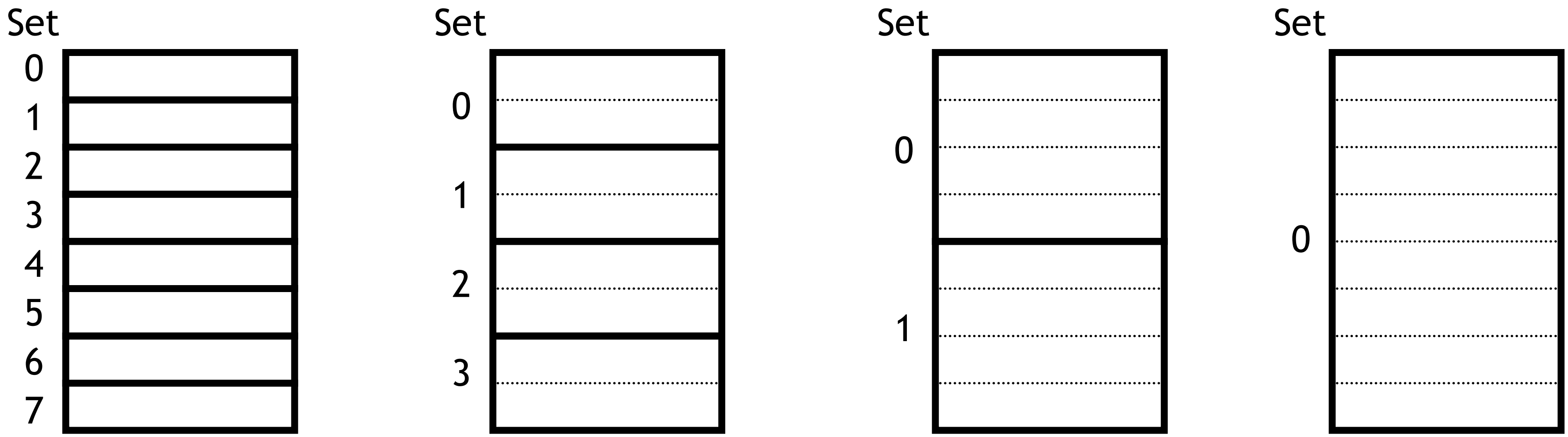
- By now you may have noticed the 1-way set associative cache is the same as a direct-mapped cache.
- Similarly, if a cache has 2^k blocks, a 2^k-way set associative cache would be the same as a fully-associative cache.

1-way
8 sets,
1 block each

2-way
4 sets,
2 blocks each

4-way
2 sets,
4 blocks each

8-way
1 set,
8 blocks

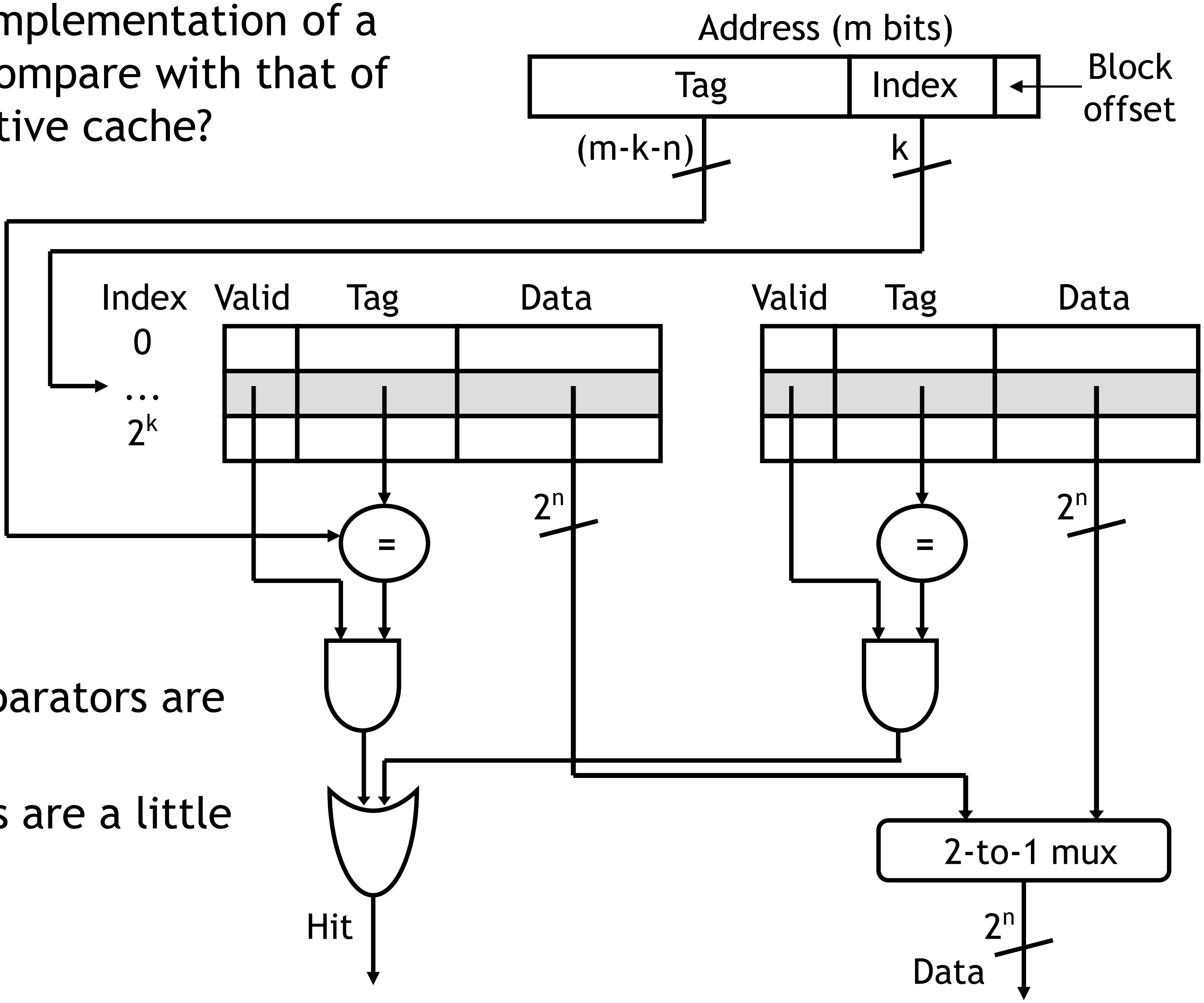


direct mapped

fully associative

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.