

CPSC 313: Computer Hardware and Operating Systems

Unit 3: Caching Cachelines and Performance



Administration

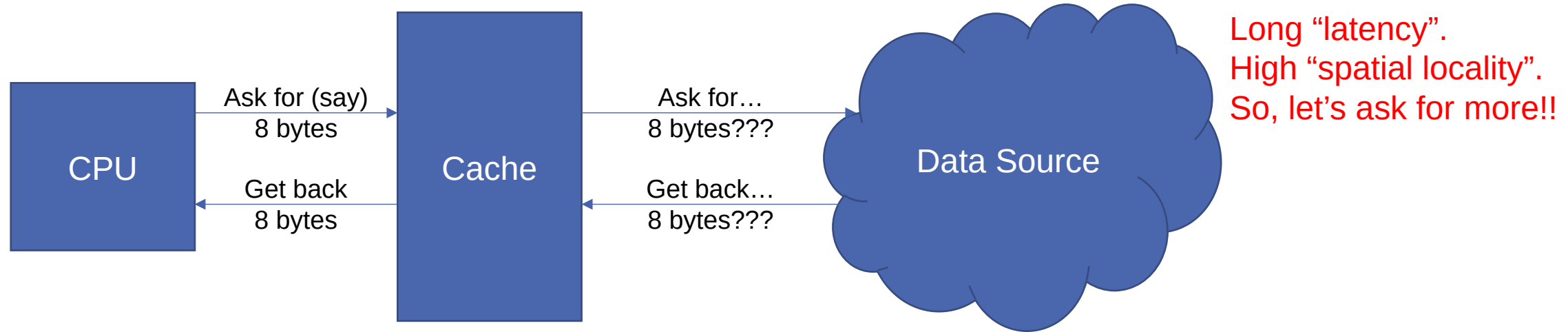
- Lab 5 is due next Sunday
 - Start **now** if you haven't already started.
- Tutorial #5 is this week.



Today

- Learning Objectives
 - Define
 - Cache line
 - Direct-mapped cache
 - Compute hit/miss rates
 - Compute cache performance
- Reading
 - 6.4

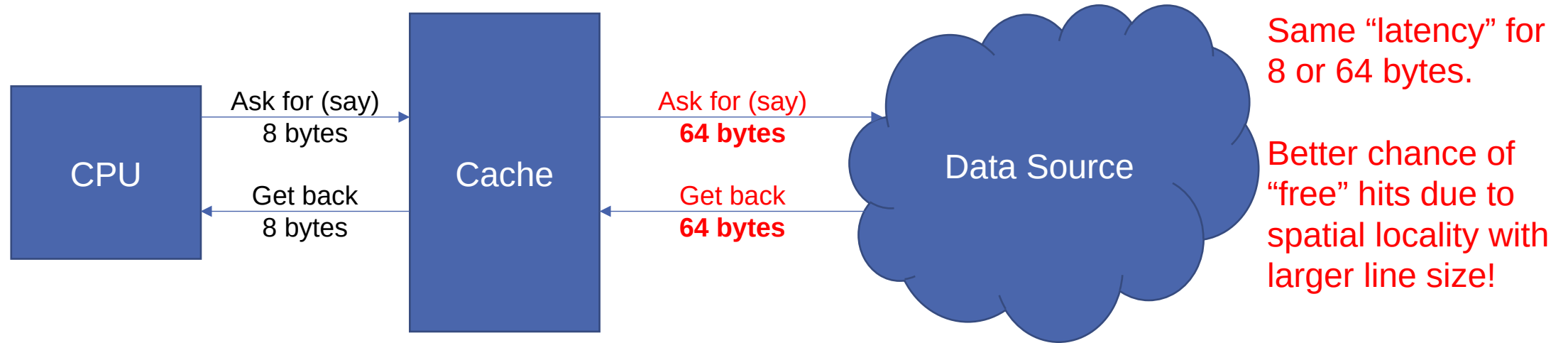
In what unit do I move things into a cache?



Spatial Locality Definition

Having accessed some item, the tendency to access other items that are near that item in the near future.

In what unit do I move things into a cache?



- **Block size:** the unit in which data is stored in a cache.
 - 64 (Intel) or 128 (Apple) bytes (block size called **cache line size** in HW caches)
 - File system caches: 4 KB
 - Object caches: size of the object (SW!)
- You should assume that this is always a power of 2 size in HW.

Assume persistent devices are at least 4KB

Cache line (block) size tradeoffs

- Large cache lines can be good:
 - If your data exhibit excellent spatial locality.
 - Imagine:
 - read 1 byte and take a miss, BUT automatically load an entire KB!
 - If your application accesses each of those bytes, one at a time:
1 miss and 1023 hits!

Cache line (block) size tradeoffs

- Large cache lines can be bad:
 - If your data does **not** exhibit great spatial locality.
 - Imagine:
 - read 1 byte and take a miss, BUT automatically load an entire KB!
 - If your application accesses only one of those bytes and then does the same for the next 1024 KBs: 1 miss each, and...
 - We waste 1023 bytes that could have been used otherwise, each time!
 - A smaller cache line size might or might not take less time to read (latency).
 - Your cache would be able to contain more (different) cache lines!

Cache line (block) size summary

- In general, smaller the closer you are to the processor.
- Powers of 2 in hardware caches
- Persistent storage uses 4 KB blocks
 - we call them “blocks” not “cache lines”...
 - and sometimes even “pages” !

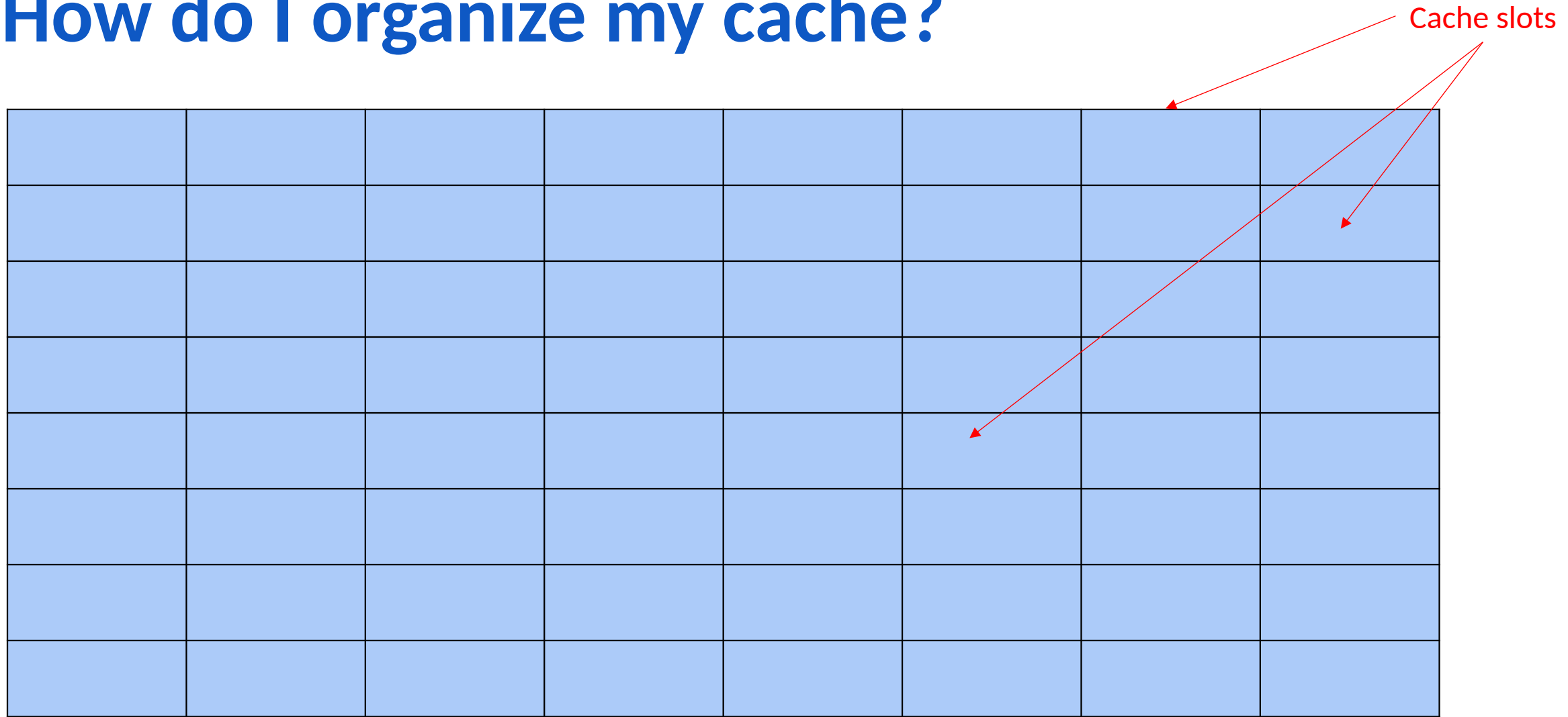
Cache line (block) size summary

- Suppose cache line size = 8 bytes.
- We divide memory into chunks of 8 bytes each.

0x00	00000000 to 00000111	00001000 to 00001111	00010000 to 00010111	00011000 to 00011111
0x20	00100000 to 00100111	00101000 to 00101111	00110000 to 00110111	00111000 to 00111111
0x40	01000000 to 01000111	01001000 to 01001111	01010000 to 01010111	01011000 to 01011111

...

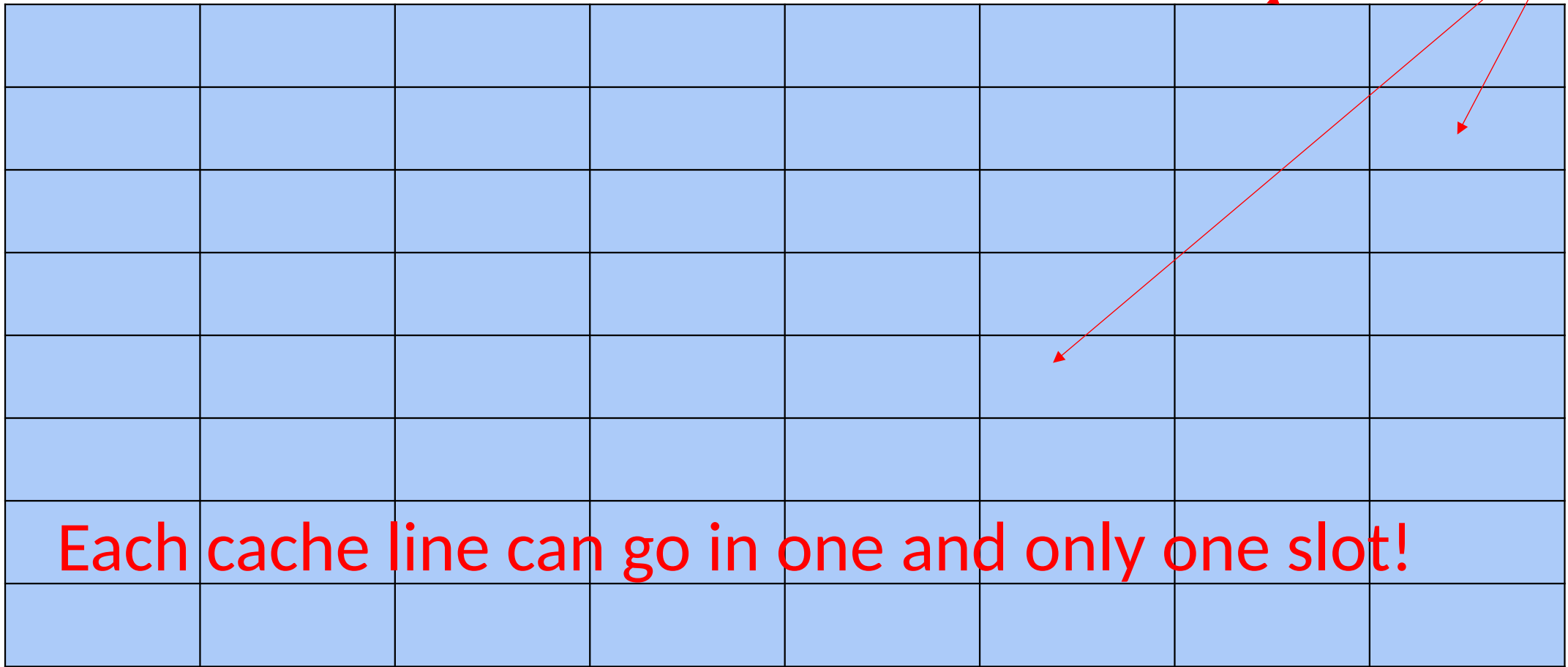
How do I organize my cache?



A slot holds a cache line (and metadata)

Direct Mapped:

A cache line can only be placed in 1 slot.



Each cache line can go in one and only one slot!

Deciding where to place a cache line

- Problem: Small set of locations in which to store a large set of objects.
- What is a general approach to this?

Deciding where to place a cache line

- Problem: Small set of locations in which to store a large set of objects.
- What is a general approach to this?
 - Hashing

Deciding where to place a cache line

- Problem: Small set of locations in which to store a large set of objects.
- What is a general approach to this?
 - Hashing
 - And we typically use actual hashing when we are implementing a cache in software (e.g., a file system cache or a database cache).
- Hardware constraints:
 - use a very very simple hash function, “mod by power-of-2 table size”. That’s trivial to implement in hardware!

Deciding where to place a cache line

0x00	00000000 to 00000111	00001000 to 00001111	00010000 to 00010111	00011000 to 00011111
0x20	00100000 to 00100111	00101000 to 00101111	00110000 to 00110111	00111000 to 00111111
0x40	01000000 to 01000111	01001000 to 01001111	01010000 to 01010111	01011000 to 01011111

- ..
- Suppose our cache has 4 locations :
Red (00), Green (01), Blue (10), Yellow (11)
 - We place successive memory blocks in these locations in order

Deciding where to place a cache line

0x00	00000000 to 00000111	00001000 to 00001111	00010000 to 00010111	00011000 to 00011111
0x20	00100000 to 00100111	00101000 to 00101111	00110000 to 00110111	00111000 to 00111111
0x40	01000000 to 01000111	01001000 to 01001111	01010000 to 01010111	01011000 to 01011111

- ..
- Suppose our cache has 4 locations :
Red (00), Green (01), Blue (10), Yellow (11)
 - We place successive memory blocks in these locations in order

Deciding where to place a cache line

0x00	00000000 to 00000111	00001000 to 00001111	00010000 to 00010111	00011000 to 00011111
0x20	00100000 to 00100111	00101000 to 00101111	00110000 to 00110111	00111000 to 00111111
0x40	01000000 to 01000111	01001000 to 01001111	01010000 to 01010111	01011000 to 01011111

- ..
- Suppose our cache has 4 locations :
Red (00), Green (01), Blue (10), Yellow (11)
 - We place successive memory blocks in these locations in order

Deciding where to place a cache line

0x00	00000000 to 00000111	00001000 to 00001111	00010000 to 00010111	00011000 to 00011111
0x20	00100000 to 00100111	00101000 to 00101111	00110000 to 00110111	00111000 to 00111111
0x40	01000000 to 01000111	01001000 to 01001111	01010000 to 01010111	01011000 to 01011111

- ..
- Suppose our cache has 4 locations :
Red (00), Green (01), Blue (10), Yellow (11)
 - We place successive memory blocks in these locations in order

Deciding where to place a cache line

0x00	00000000 to 00000111	00001000 to 00001111	00010000 to 00010111	00011000 to 00011111
0x20	00100000 to 00100111	00101000 to 00101111	00110000 to 00110111	00111000 to 00111111
0x40	01000000 to 01000111	01001000 to 01001111	01010000 to 01010111	01011000 to 01011111

- ..
- Suppose our cache has 4 locations :
Red (00), Green (01), Blue (10), Yellow (11)
 - We place successive memory blocks in these locations in order

Deciding where to place a cache line

0x00	00000000 to 00000111	00001000 to 00001111	00010000 to 00010111	00011000 to 00011111
0x20	00100000 to 00100111	00101000 to 00101111	00110000 to 00110111	00111000 to 00111111
0x40	01000000 to 01000111	01001000 to 01001111	01010000 to 01010111	01011000 to 01011111

- ..
- Suppose our cache has 4 locations :
Red (00), Green (01), Blue (10), Yellow (11)
 - We place successive memory blocks in these locations in order

Deciding where to place a cache line

0x00	00000000 to 00000111	00001000 to 00001111	00010000 to 00010111	00011000 to 00011111
0x20	00100000 to 00100111	00101000 to 00101111	00110000 to 00110111	00111000 to 00111111
0x40	01000000 to 01000111	01001000 to 01001111	01010000 to 01010111	01011000 to 01011111

- ..
- Suppose our cache has 4 locations :
Red (00), Green (01), Blue (10), Yellow (11)
 - We place successive memory blocks in these locations in order

Deciding where to place a cache line

0x00	00000000 to 00000111	00001000 to 00001111	00010000 to 00010111	00011000 to 00011111
0x20	00100000 to 00100111	00101000 to 00101111	00110000 to 00110111	00111000 to 00111111
0x40	01000000 to 01000111	01001000 to 01001111	01010000 to 01010111	01011000 to 01011111

- ..
- Suppose our cache has 4 locations :
Red (00), Green (01), Blue (10), Yellow (11)
 - We place successive memory blocks in these locations in order

Deciding where to place a cache line

0x00	00000000 to 00000111	00001000 to 00001111	00010000 to 00010111	00011000 to 00011111
0x20	00100000 to 00100111	00101000 to 00101111	00110000 to 00110111	00111000 to 00111111
0x40	01000000 to 01000111	01001000 to 01001111	01010000 to 01010111	01011000 to 01011111

- ..
- Suppose our cache has 4 locations :
Red (00), Green (01), Blue (10), Yellow (11)
 - We place successive memory blocks in these locations in order

Deciding where to place a cache line

0x00	00000000 to 00000111	00001000 to 00001111	00010000 to 00010111	00011000 to 00011111
0x20	00100000 to 00100111	00101000 to 00101111	00110000 to 00110111	00111000 to 00111111
0x40	01000000 to 01000111	01001000 to 01001111	01010000 to 01010111	01011000 to 01011111

- ..
- Suppose our cache has 4 locations :
Red (00), Green (01), Blue (10), Yellow (11)
 - We place successive memory blocks in these locations in order

Deciding where to place a cache line

0x00	00000000 to 00000111	00001000 to 00001111	00010000 to 00010111	00011000 to 00011111
0x20	00100000 to 00100111	00101000 to 00101111	00110000 to 00110111	00111000 to 00111111
0x40	01000000 to 01000111	01001000 to 01001111	01010000 to 01010111	01011000 to 01011111

- ..
- Suppose our cache has 4 locations :
Red (00), Green (01), Blue (10), Yellow (11)
 - We place successive memory blocks in these locations in order

Deciding where to place a cache line

0x00	00000000 to 00000111	00001000 to 00001111	00010000 to 00010111	00011000 to 00011111
0x20	00100000 to 00100111	00101000 to 00101111	00110000 to 00110111	00111000 to 00111111
0x40	01000000 to 01000111	01001000 to 01001111	01010000 to 01010111	01011000 to 01011111

- ..
- Suppose our cache has 4 locations :
Red (00), Green (01), Blue (10), Yellow (11)
 - We place successive memory blocks in these locations in order

Deciding where to place a cache line

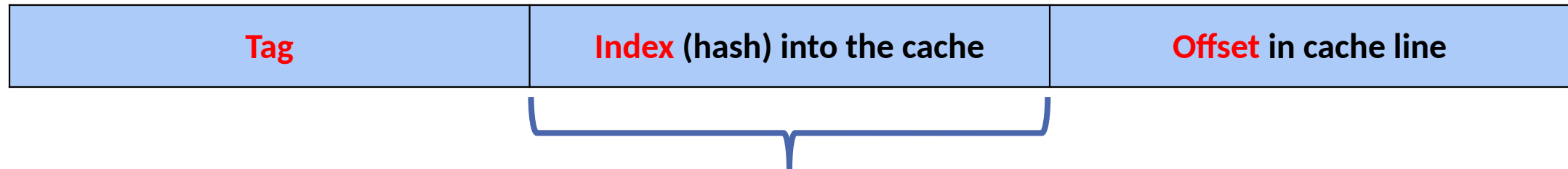
0x00	00000000 to 00000111	00001000 to 00001111	00010000 to 00010111	00011000 to 00011111
0x20	00100000 to 00100111	00101000 to 00101111	00110000 to 00110111	00111000 to 00111111
0x40	01000000 to 01000111	01001000 to 01001111	01010000 to 01010111	01011000 to 01011111

..

- What do the addresses of the red blocks have in common?
- What about the addresses of the green blocks? The blue blocks? The yellow blocks?

From Addresses to Cache lines: Index

Address



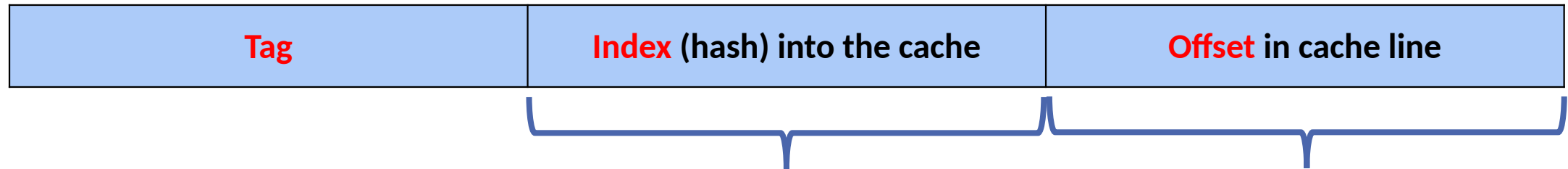
The number of bits in this part are a function of the # of entries* in the cache

If the cache holds 128 cache lines, then I need to index 128 slots (0-127) ; that requires 7 bits

* Warning: Right now, we're going to explain this assuming that every address can be placed in one and only one place in the cache; as you'll see in subsequent material, this is not always the case; when we do that, we'll come back to this.

From Addresses to Cache lines: Offset

Address



The number of bits in this part are a function of the size* of the cache

The number of bits in this part are a function of the cache line size

- If my cache line size is 16 bytes, then addresses of the form $0x???????0 - 0x???????F$ will all be in the same cache line.
- Therefore: if a cache line is 2^m bytes, the bottom m bits, do **not** distinguish between cache lines.
- Instead, they tell you where in the cache line you find a particular address.

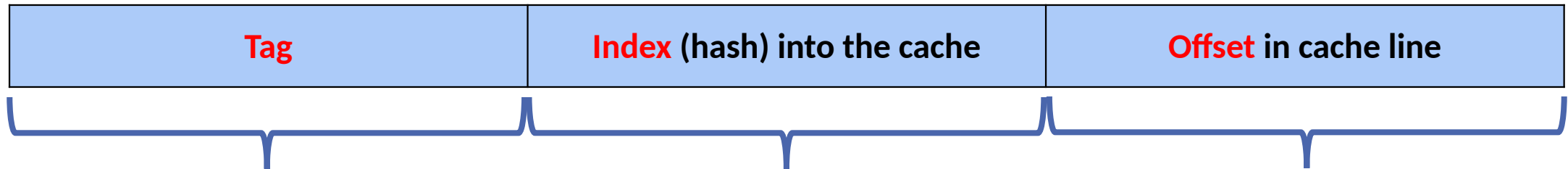
Offset Example: 16 byte cache line

Offset in cache line

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Slot 0																
Slot 1																

From Addresses to Cache lines: tag

Address



The rest of the bits are tag:
they disambiguate addresses
that have the same index.

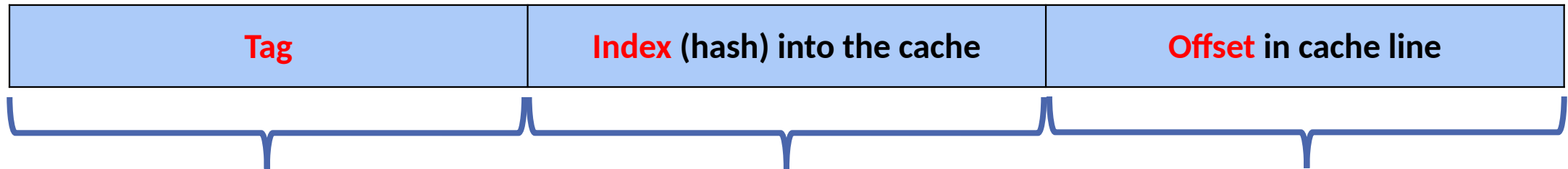
The number of bits in this part are a
function of the size* of the cache

The number of bits in this part are a
function of the cache line size

* Warning: Right now, we're going to explain this assuming that every address can be placed in one and only one place in the cache; as you'll see in subsequent material, this is not always the case; when we do that, we'll come back to this.

From Addresses to Cache lines: hash table analogy

Address



The rest of the bits are tag:
they disambiguate addresses
that have the same index.

The number of bits in this part are a
function of the size* of the cache

The number of bits in this part are a
function of the cache line size

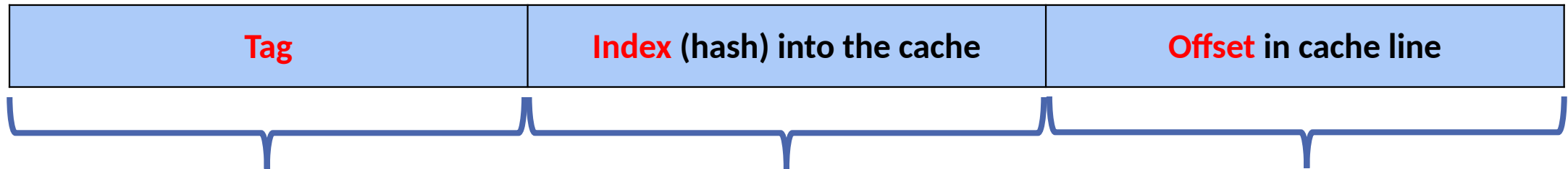
In a hash table:

- An entry had to store **both** a key and a value. **Why?** (Arrays don't do that!)
- What is the **key** in a cache line? (Hint: it's on this page.)
- What is the **value** in a cache line? (Hint: it's **not** on this page!)

* Warning: Right now, we're going to explain this assuming that every address can be placed in one and only one place in the cache; as you'll see in subsequent material, this is not always the case; when we do that, we'll come back to this.

From Addresses to Cache lines: hash table analogy

Address



The rest of the bits are tag:
they disambiguate addresses
that have the same index.

The number of bits in this part are a
function of the size* of the cache

The number of bits in this part are a
function of the cache line size

In a hash table:

- An entry had to store **both** a key and a value
- What is the **key** in a cache line?
- What is the **value** in a cache line?

* Warning: Right now, we're going to explain this assuming that every address can be placed in one and only one place in the cache; as you'll see in subsequent material, this is not always the case; when we do that, we'll come back to this.

Address 2: 0xFA

Address 3: 0xCE

Cache Example: A Teeny Tiny cache

Our cache has:

- 8-byte cache lines
- 4 slots
- Total size: 32 bytes

Let's say that addresses are only 8 bits (one byte)!

Address 1: 0xBE = 1011 1110

		0	1	2	3	4	5	6	7
Slot 0									
Slot 1									
Slot 2									
Slot 3									

Address 2: 0xFA

Address 3: 0xCE

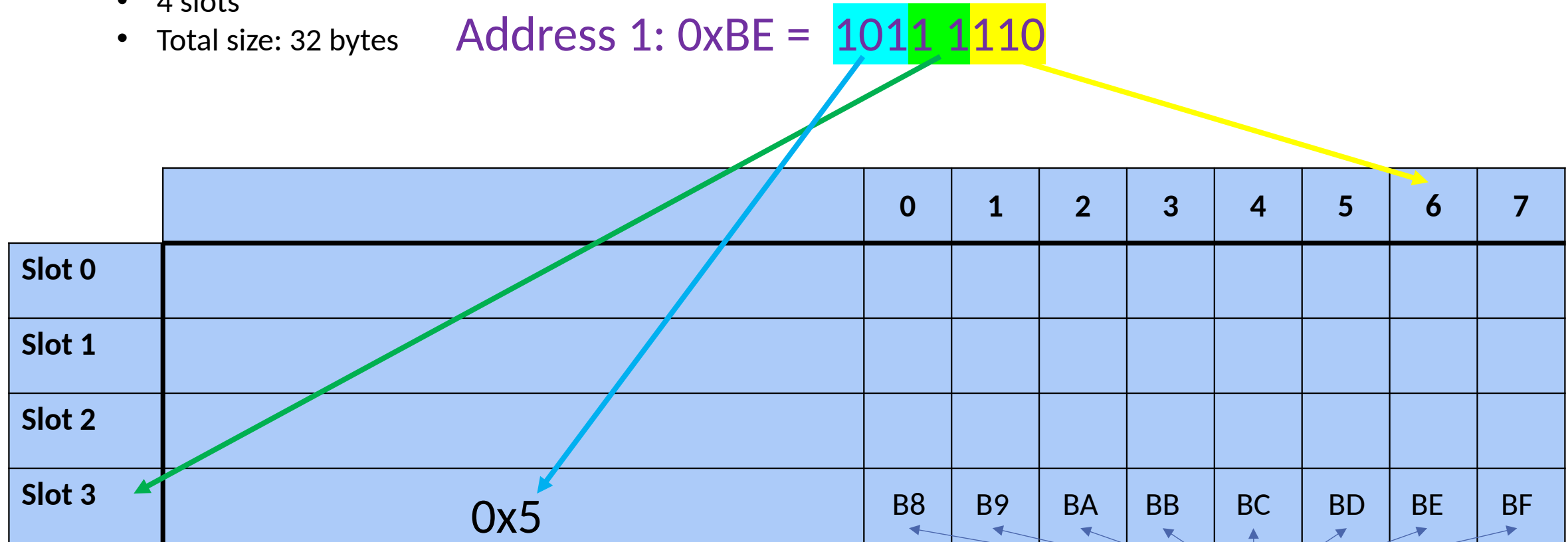
Cache Example: A Teeny Tiny cache

Our cache has:

- 8-byte cache lines
- 4 slots
- Total size: 32 bytes

Let's say that addresses are only 8 bits (one byte)!

Address 1: 0xBE = 1011 1110



Cache Example: A Teeny Tiny cache

Address 3: 0xCE

Our cache has:

- 8-byte cache lines
- 4 slots
- Total size: 32 bytes

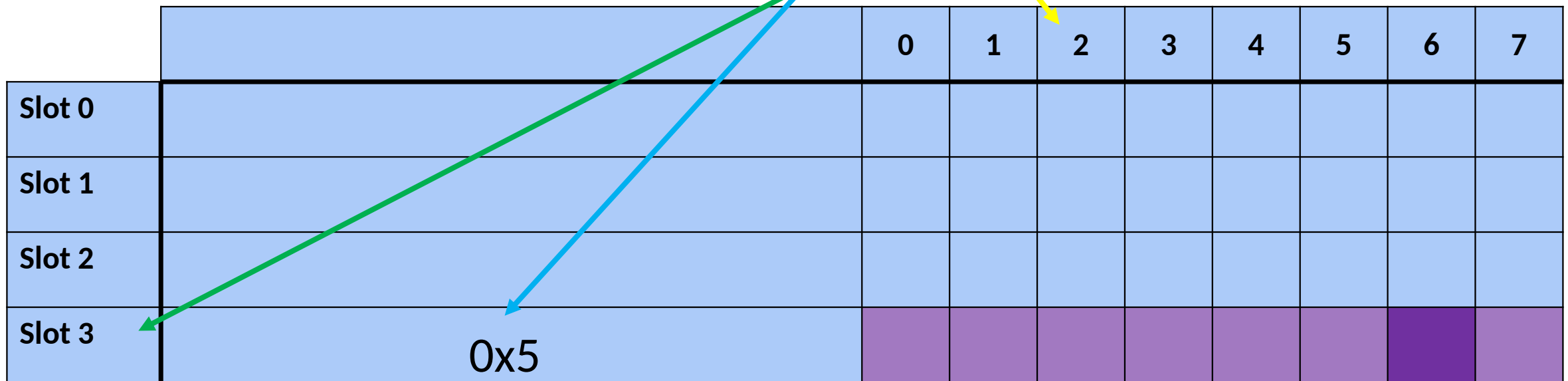
Let's say that addresses are only 8 bits (one byte)!

Address 1: 0xBE = 1011 1110

Address 2: 0xFA = 1111 1010

This is a conflict!

		0	1	2	3	4	5	6	7
Slot 0									
Slot 1									
Slot 2									
Slot 3	0x5								



Cache Example: A Teeny Tiny cache

Address 3: 0xCE

Our cache has:

- 8-byte cache lines
- 4 slots
- Total size: 32 bytes

Let's say that addresses are only 8 bits (one byte)!

Address 1: 0xBE = 1011 1110

Address 2: 0xFA = 1111 1010

This is a conflict!

		0	1	2	3	4	5	6	7
Slot 0									
Slot 1									
Slot 2									
Slot 3	0x7								

Cache Example: A Teeny Tiny cache

Address 3: 0xCE

Our cache has:

- 8-byte cache lines
- 4 slots
- Total size: 32 bytes

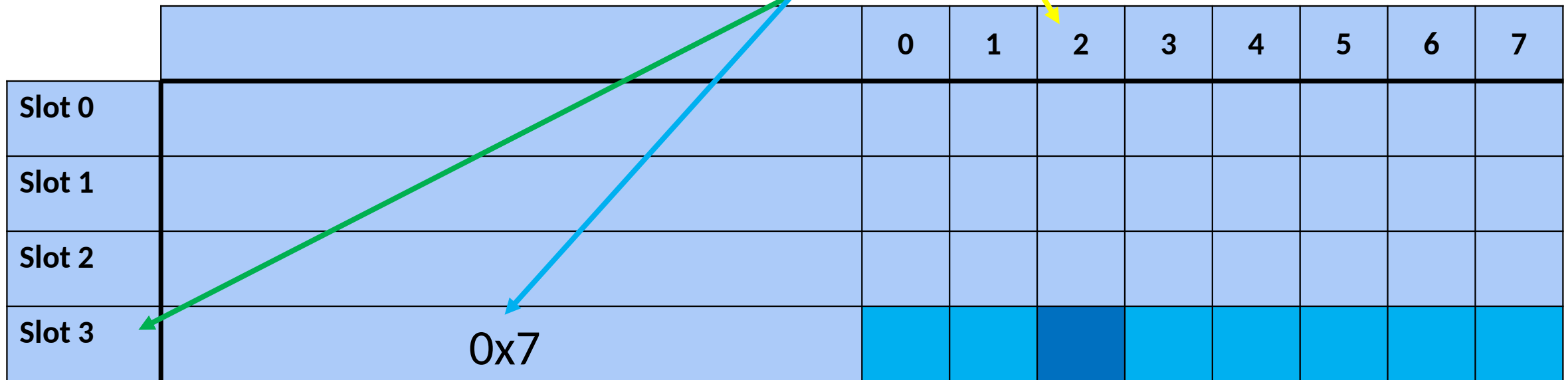
Let's say that addresses are only 8 bits (one byte)!

Address 1: 0xBE = 1011 1110

Address 2: 0xFA = 1111 1010

This is a conflict!

		0	1	2	3	4	5	6	7
Slot 0									
Slot 1									
Slot 2									
Slot 3	0x7								



Cache Example: A Teeny Tiny cache

Our cache has:

- 8-byte cache lines
- 4 slots
- Total size: 32 bytes

Let's say that addresses are only 8 bits (one byte)!

Address 1: 0xBE = 1011 1110

Address 2: 0xFA = 1111 1010

Address 3: 0xCE = 1100 1110

		0	1	2	3	4	5	6	7
Slot 0									
Slot 1	0x6								
Slot 2									
Slot 3	0x7								

Cache Example: A Teeny Tiny cache

Our cache has:

- 8-byte cache lines
- 4 slots
- Total size: 32 bytes

Let's say that addresses are only 8 bits (one byte)!

Address 1: 0xBE = 1011 1110

Address 2: 0xFA = 1111 1010

Address 3: 0xCE = 1100 1110

		0	1	2	3	4	5	6	7
Slot 0									
Slot 1	0x6								
Slot 2									
Slot 3	0x7								

Evaluating a Cache

- Hits are much better than misses!
- We measure the efficiency of a cache in terms of its **cache hit rate**:
 - $\# \text{ cache hits} / \# \text{ cache accesses}$
 - $\# \text{ cache hits} / (\# \text{ cache hits} + \# \text{ cache misses})$
- Example:
 - I access my cache 1000 times and have 400 hits.
 - My cache hit rate is $400/1000 = 40\%$ (aka 0.40 hit rate)
- Good performance requires high cache hit rates.

Computing Average Access Time

- Let's say that it takes 1 time unit to access your cache and 100 time units for a miss (including accessing the data source).
 - 10% hit rate: $90\% * 100 + 10\% * 1 = 90.1$ time units/access
 - 50% hit rate: $50\% * 100 + 50\% * 1 = 50.5$ time units/access
 - 90% hit rate: $10\% * 100 + 90\% * 1 = 10.9$ time units/access
 - 99% hit rate: $1\% * 100 + 99\% * 1 = 1.99$ time units/access
- The ratio between access time for the cache and the data source dictates just how good your hit rate needs to be to obtain good performance from a cache.

Counting hits

temporal locality



- If you touch the same item more than once within a short period, you get a hit, but there is another way to get a hit.
- Think about the fact that your cache is organized in **cache lines** ...
- Consider our teeny tiny cache.
 - Let's say that I access 8 bytes in order starting at the address 0x00.
 - How many misses will I get?
 - How many hits will I get?

Now it is your turn!

