CSE 351
Section 7

Caches Intro - Autumn 2020

Caches

Cache Motivation

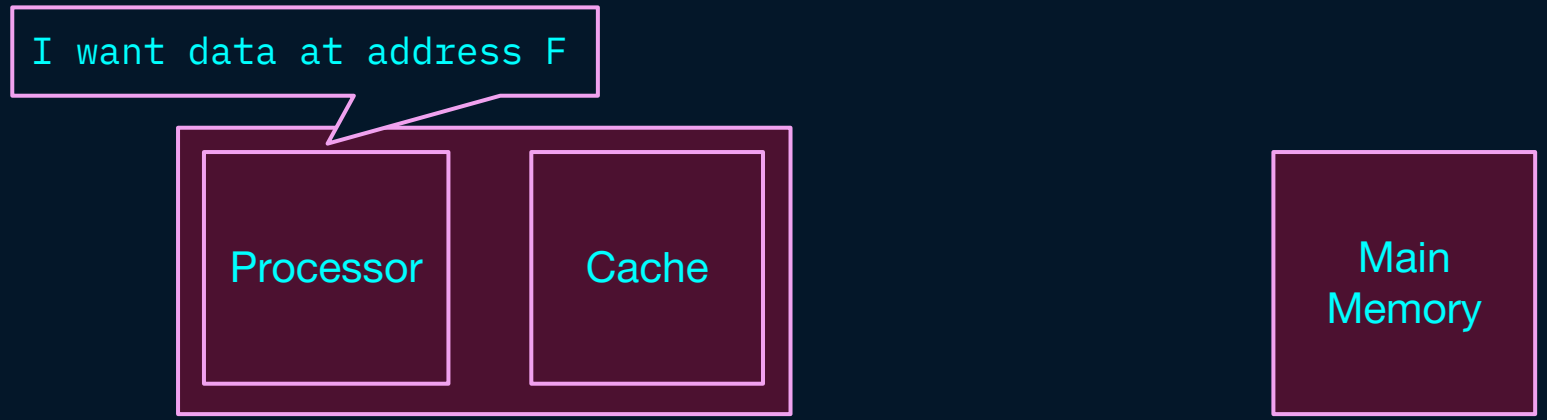
Going all the way to memory is expensive! What if we had an intermediate place where we could store data closer to the processor?

- **Temporal locality**
  - If you used some data, you will probably use it again
- **Spatial locality**
  - If you used some data, you will probably use data that is close to it also

Can you think of any code examples of the above?

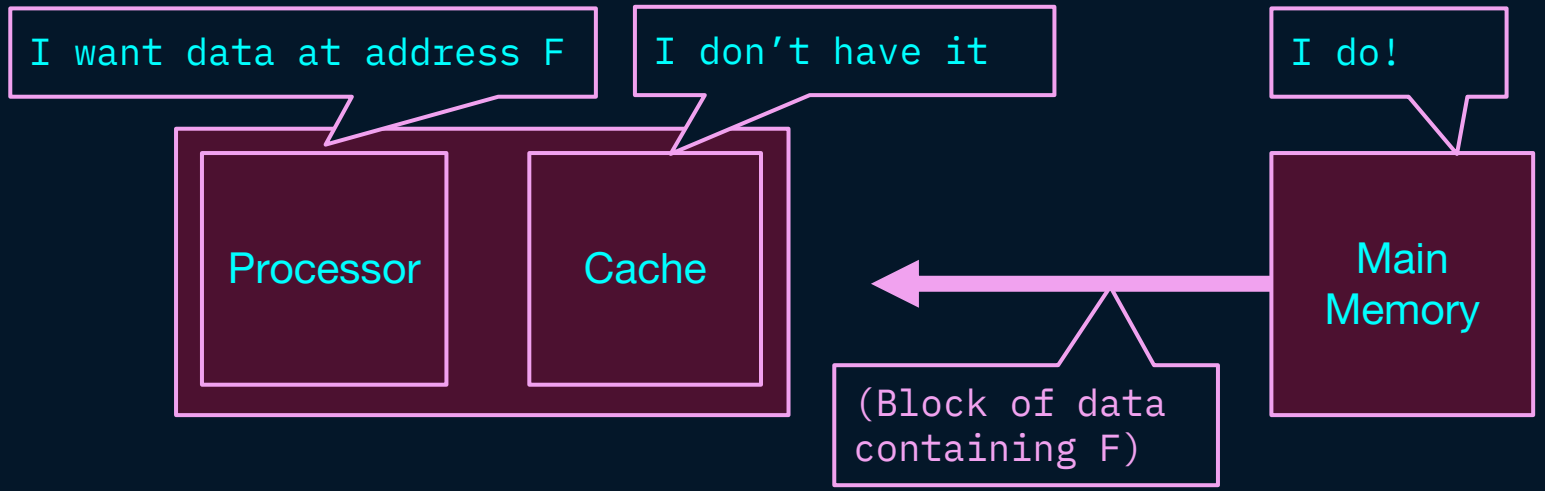
Cache Review

The cache stores a subset of main memory with much faster access time! It is located much closer to the processor, often on the same chip. When we access memory, we check the cache(s) first.



Cache Review

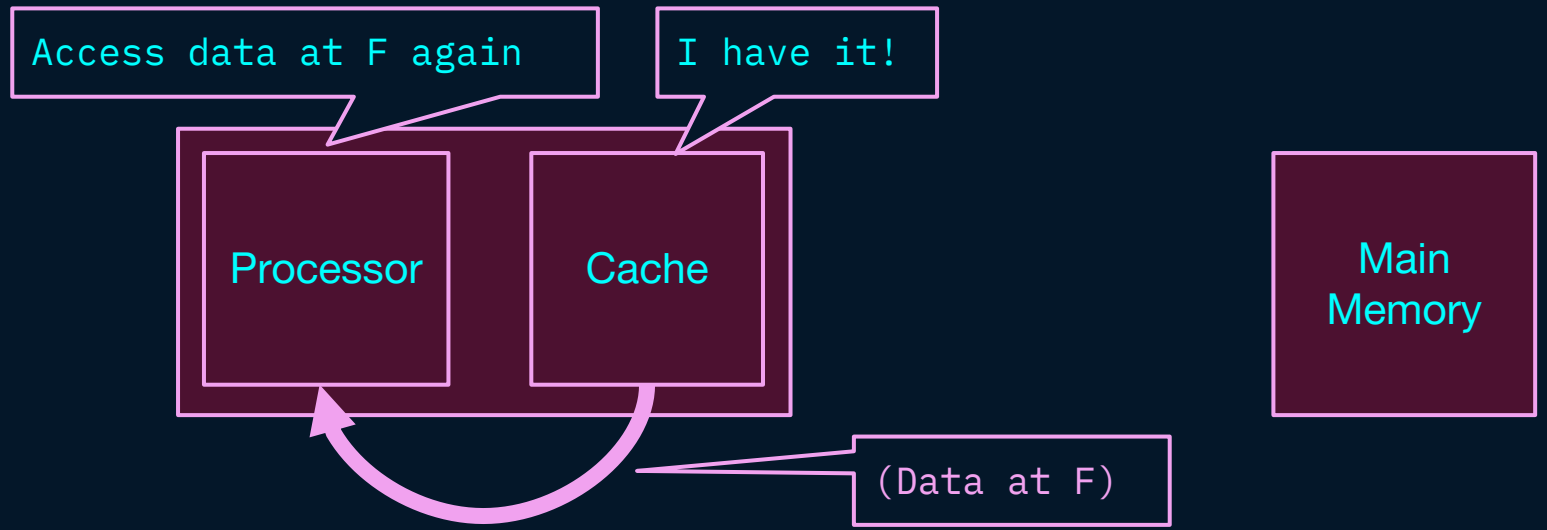
If the data we want isn't in the cache, that's a *cache miss*. We have to go to main memory, and then we'll save that data in the cache. By transferring entire blocks of data at a time, we take advantage of spatial locality.



Cache Review

If the data we want is in the cache and valid, that's a *cache hit*.

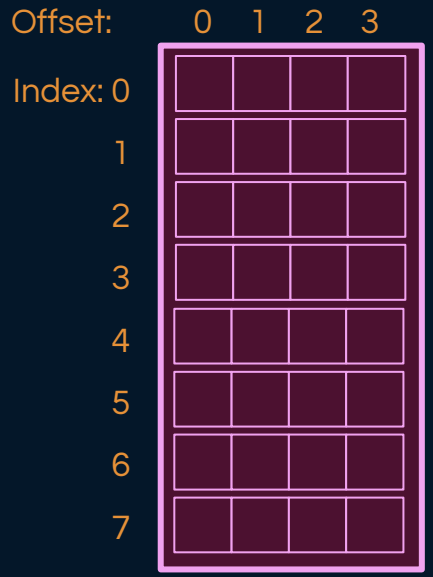
We don't go to memory, which saves us a lot of time!



Cache Organization

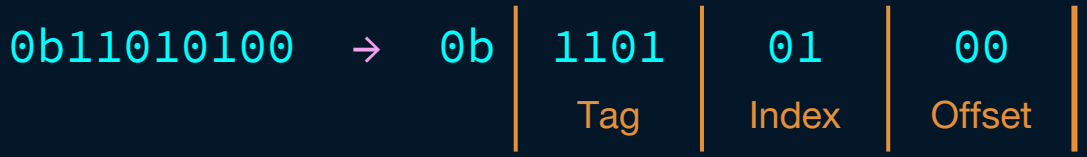
- Cache blocks (or lines) are a fixed size.
- The cache has “slots” for blocks of data that are *indexed*.
- We access individual bytes in a block with as an *offset* from the first byte in the block.
- The number of blocks in the cache times the number of bytes in a block gives us the size of the cache.

Cache with 8 blocks of 4 bytes each:



Tag, Index, Offset!

To determine where each address maps to in the cache, we break it up into:

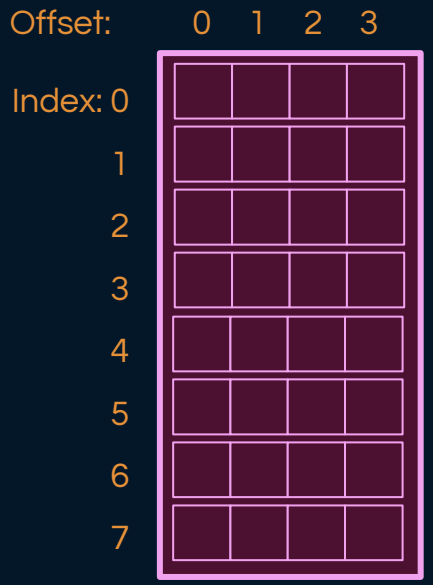


- The **tag** is used to distinguish blocks which map to the same location. It is stored along with the block data and a "valid bit" which indicates whether the data is current and ready to be used.
- The **index** tells us the "slot" in which the data at this address goes.
- The **offset** is tells us how far into the *block* our address is.

This can be thought of kind of like a modulo hashing function; the address gets mapped to a location in the cache based on its index bits.

Cache Parameters

Symbol	Meaning
K	Block Size
C	Cache Size
E	Associativity
m	Address Width
k	# Offset Bits = $\log_2(K)$
s	# Index Bits = $\log_2(C / K)$
t	# Tag Bits = m - k - s



8 blocks of 4 bytes:

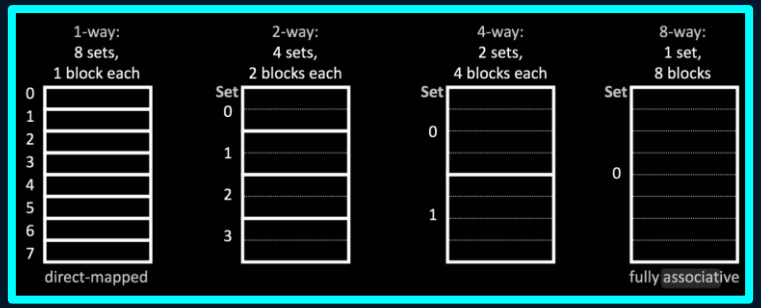
K = 4 bytes k = 2

C = 32 bytes s = 3

Direct-mapped → E = 1

Coming Soon: Associativity

A cache has associativity > 1 if each index may correspond to > 1 block:



We'll be talking more about associativity on Friday and next week.

Today our focus will be on *direct-mapped* caches (E = 1), meaning that each cache index corresponds to exactly one block and vice-versa.

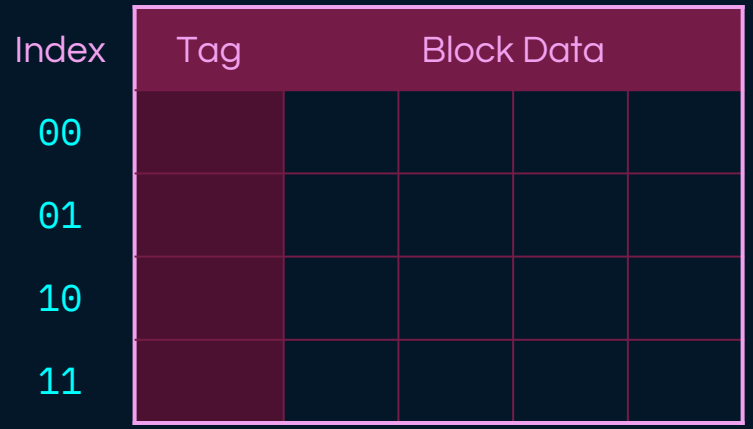
The final page of the worksheet has practice problems with associative caches.

Example

16 B capacity cache, 4 B block size, direct-mapped, 8 bit address length.

What's the TIO address breakdown? How can we visualize this cache?

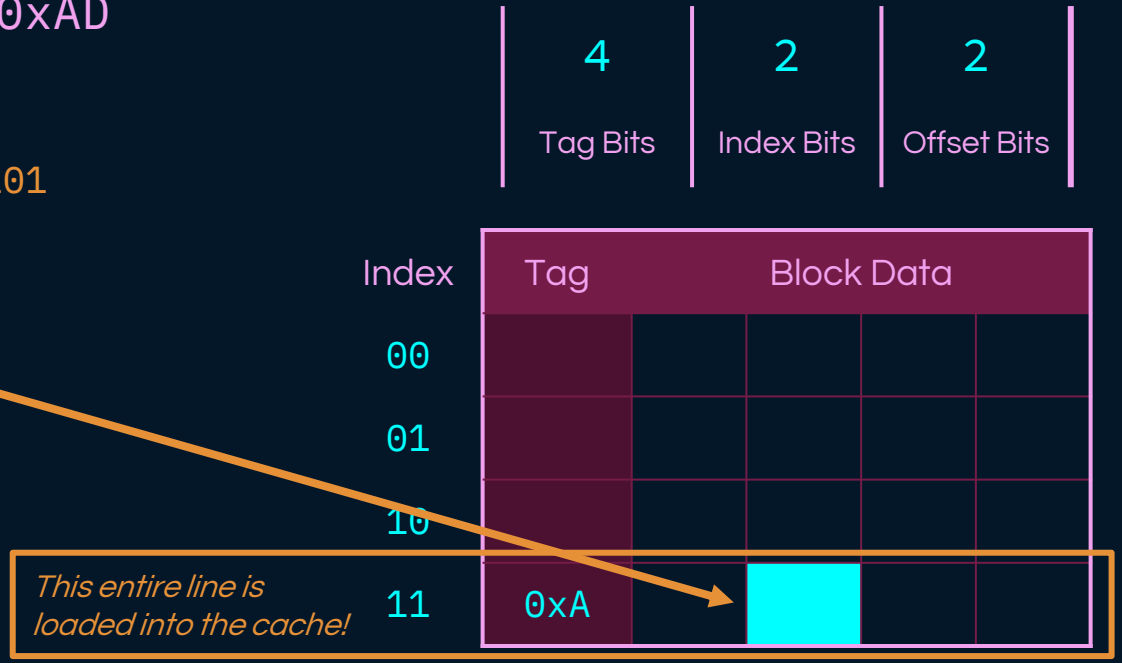
- #bits for offset: 2 (4 B in a block; $\log_2(4) = 2$)
- #bits for index: 2 (4 blocks in cache, direct-mapped)
- #bits for tag: 4 (remaining bits of address)



Example

Read 1 byte from address `0xAD`

1. Translate to Binary:
   a. `0xAD = 0b 1010 1101`
2. Split into TIO
   a. Tag = `1010`
   b. Index = `11`
   c. Offset = `01`



Code Analysis

Miss Rate

The cache is mostly invisible to programmers. But we can still make some optimizations by keeping it in mind!

The *miss rate* is the ratio of cache misses to total memory accesses. If we can analyze when cache misses occur, we may be able to make our code more cache-friendly and improve performance.

Average memory access time (AMAT) = (Hit Time) + (Miss Penalty) * (Miss Rate)

What's the Miss Rate?

- First loop
  - Note array starts at beginning of a block
  - First access misses (cold cache), loads array[0] through array[3] into cache
    - array[1] through array[3] are hits
  - Miss on array[4], load [4] through [7]
    - array[5] through array[7] are hits
  - 8 accesses, 2 misses
- Second loop
  - Entire array is still in the cache!
  - 8 accesses, 0 misses
- Overall miss rate
  - 16 accesses, 2 misses
  - 2 / 16 = **12.5%**

```
char val = 0;

for (int i = 0; i < 8; i++)
    val += array[i];

for (i = 0; i < 8; i++)
    val ^= array[i];
```

`array` is a **`char`** array of size 8.
Its address is 0x600000, and the cache starts cold.
`val` is stored in a register.

Cache Parameters

C = 256 bytes K = 4 bytes

Cache State

After execution of both loops:

Index	Valid	Block Offset			
		00	**01**	**10**	**11**
0	1	array[0]	array[1]	array[2]	array[3]
1	1	array[4]	array[5]	array[6]	array[7]
2	0	?	?	?	?
...			...		

Cache Simulator

Cache Simulator!

Link:

https://courses.cs.washington.edu/courses/cse351/cachesim/

Worksheet:

https://us.edstem.org/courses/2402/lessons/5419/slides/31721

We haven't covered all of its features in class yet, but the cache simulator can be a helpful tool for reasoning through cache problems and mechanisms, particularly on homework and in lab 4.