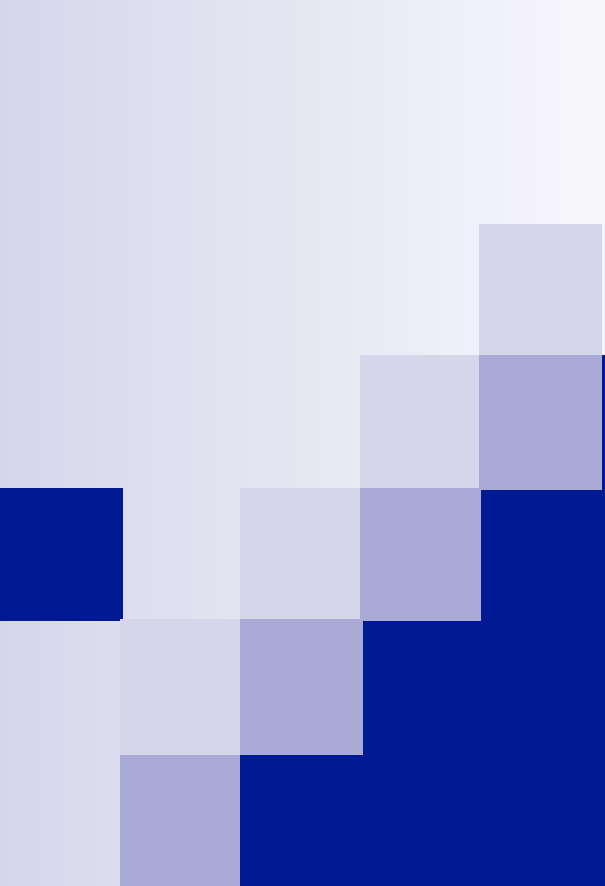




The Memory Bottleneck

**ICS332
Operating Systems**

Henri Casanova (henric@hawaii.edu)

RAM is “slow”

- Often programs are slow because the memory is slow
- Accessing a register is very fast
 - e.g., a 4GHz CPU can update a register in 0.25 nanosecond (1 cycle)
- Accessing the memory can take ~50 ns
- **What does the CPU do while it's waiting for the memory to give it data?**

RAM is “slow”

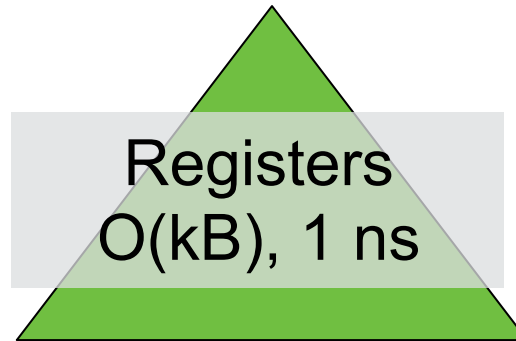
- Often programs are slow because **the memory is slow**
- Accessing a register is very fast
 - e.g., a 4GHz CPU can update a register in 0.25 nanosecond (1 cycle)
- Accessing the memory can take ~50 ns
- **What does the CPU do while it's waiting for the memory to give it data?**
- **NOTHING!!** (yes, this is a problem)
- This is the famous “Von-Neumann Bottleneck”
- Many techniques have been developed to address this problem



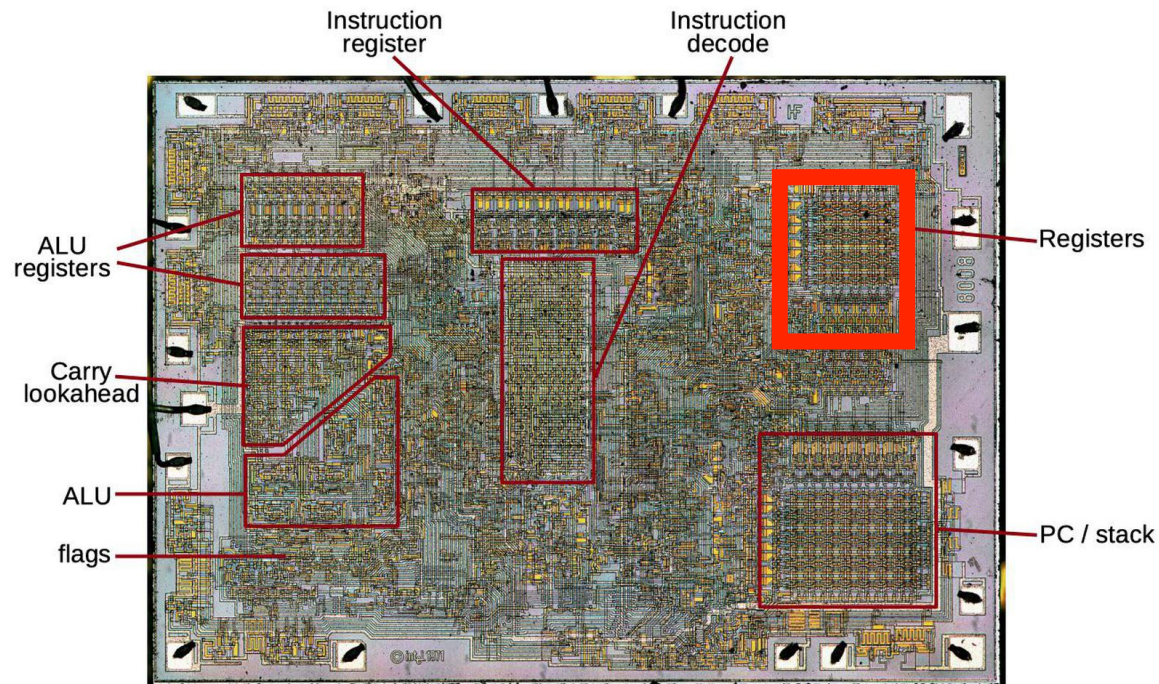
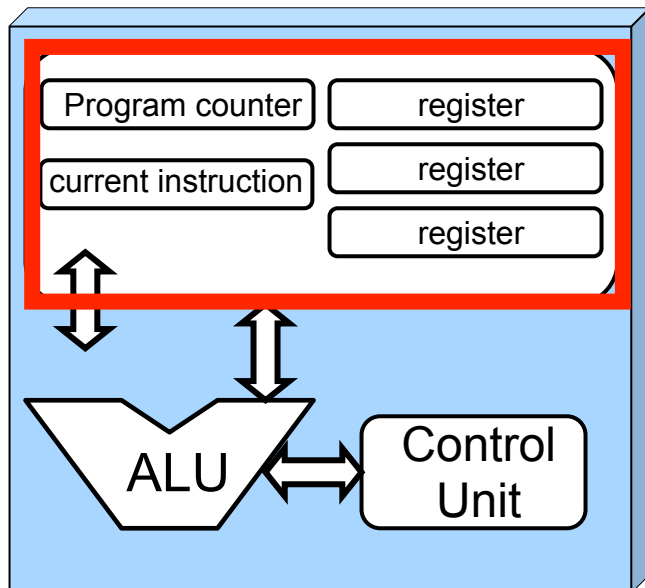
The Memory Hierarchy

- We would like a gigantic and fast memory
- Could we just build the memory just as gazillions of registers?
- No!!! Cost/physics make it impossible
- Instead, we play a trick to provide the illusion of a fast memory
- This trick is called the **memory hierarchy**

The Memory Hierarchy



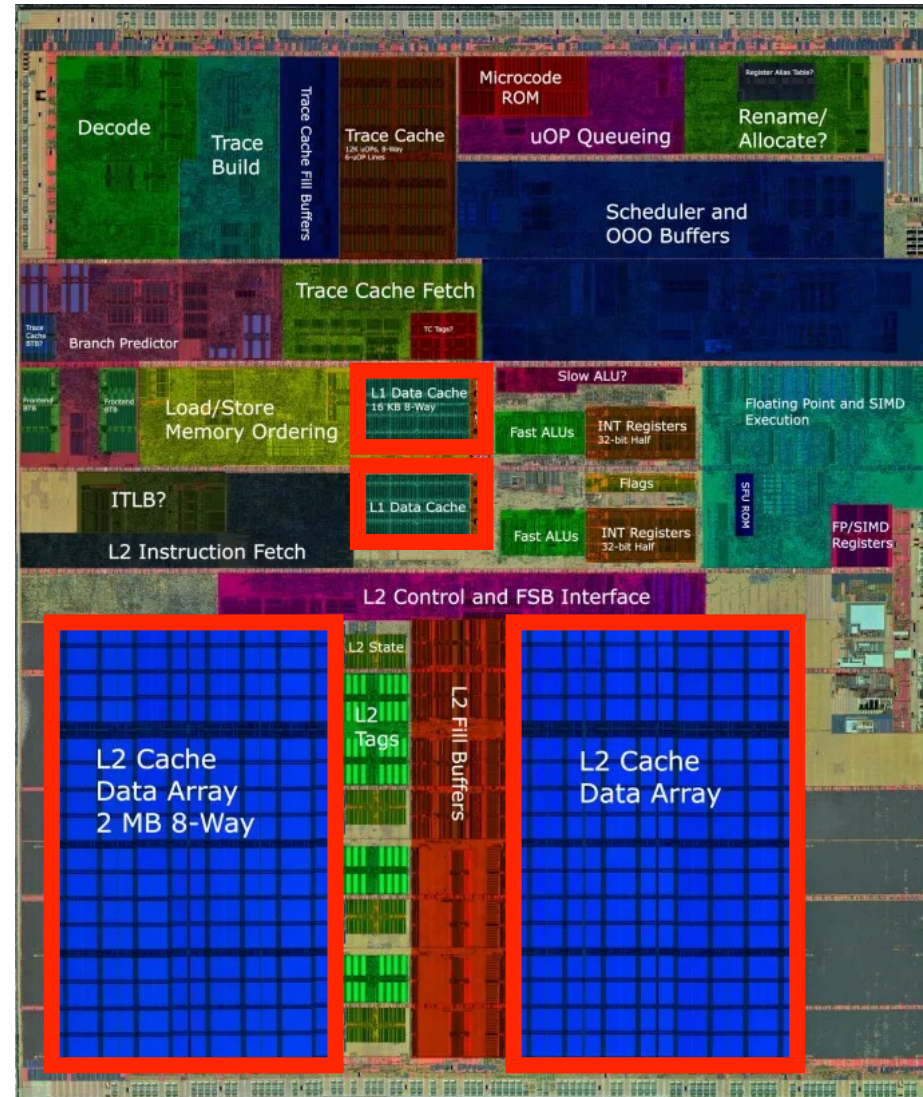
Intel 8088



The Memory Hierarchy

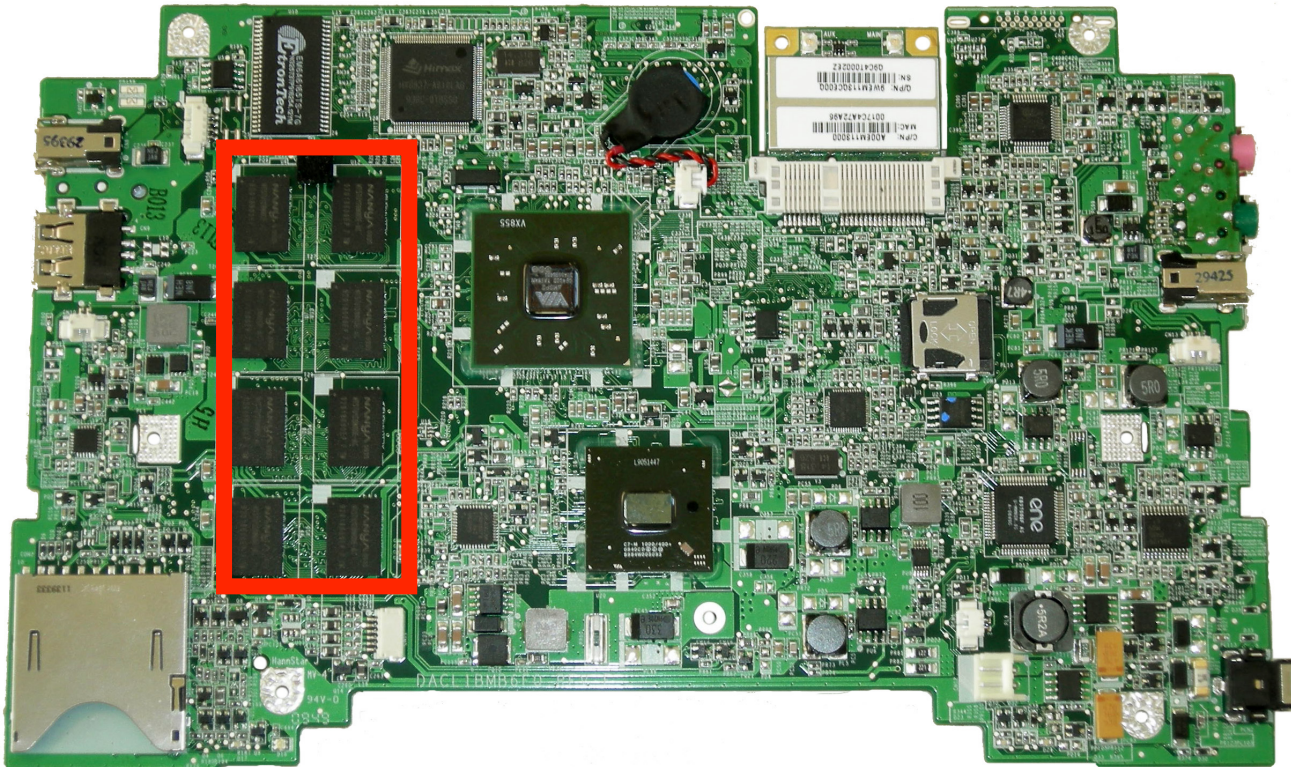
Caches
O(MB), 1-50 ns

Intel
Netburst
Processor



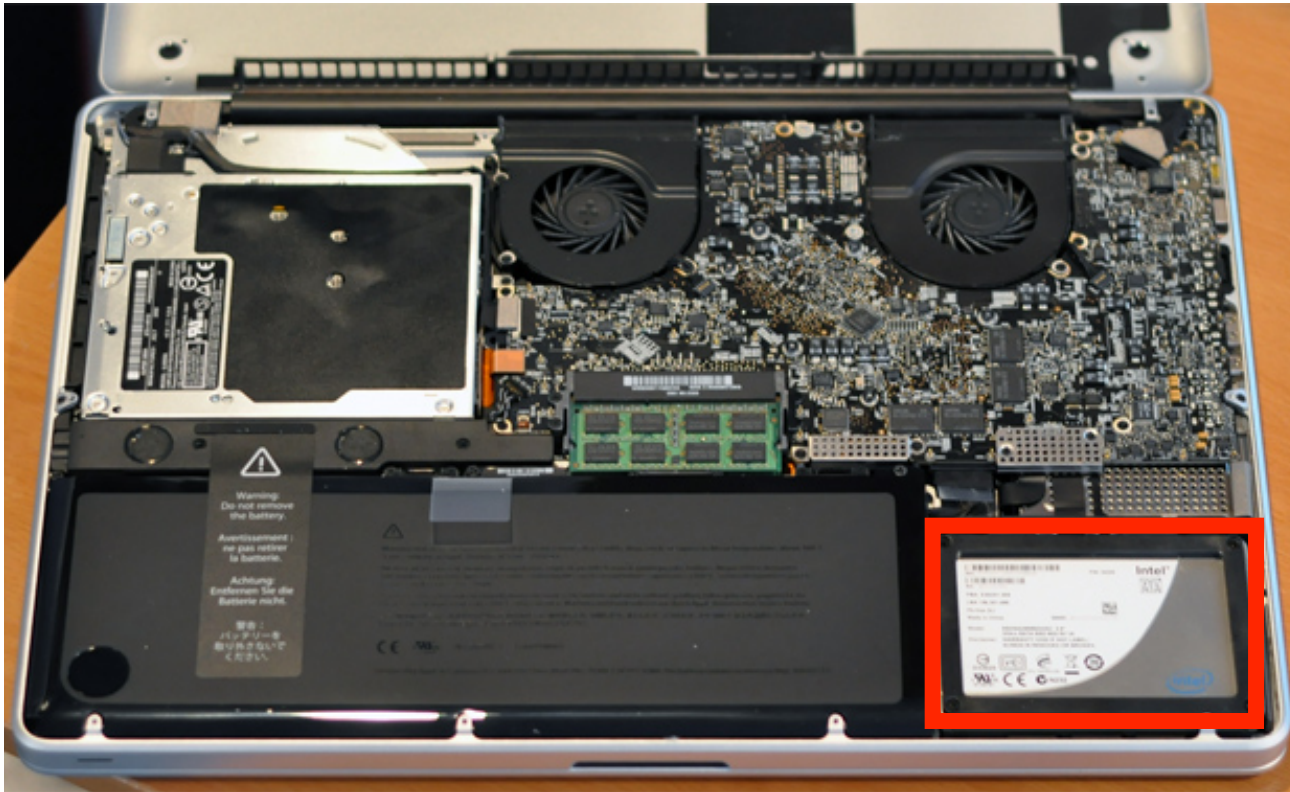
The Memory Hierarchy

Memory
 $O(\text{GB})$, $\sim 100 \text{ ns}$



Dell laptop
motherboard

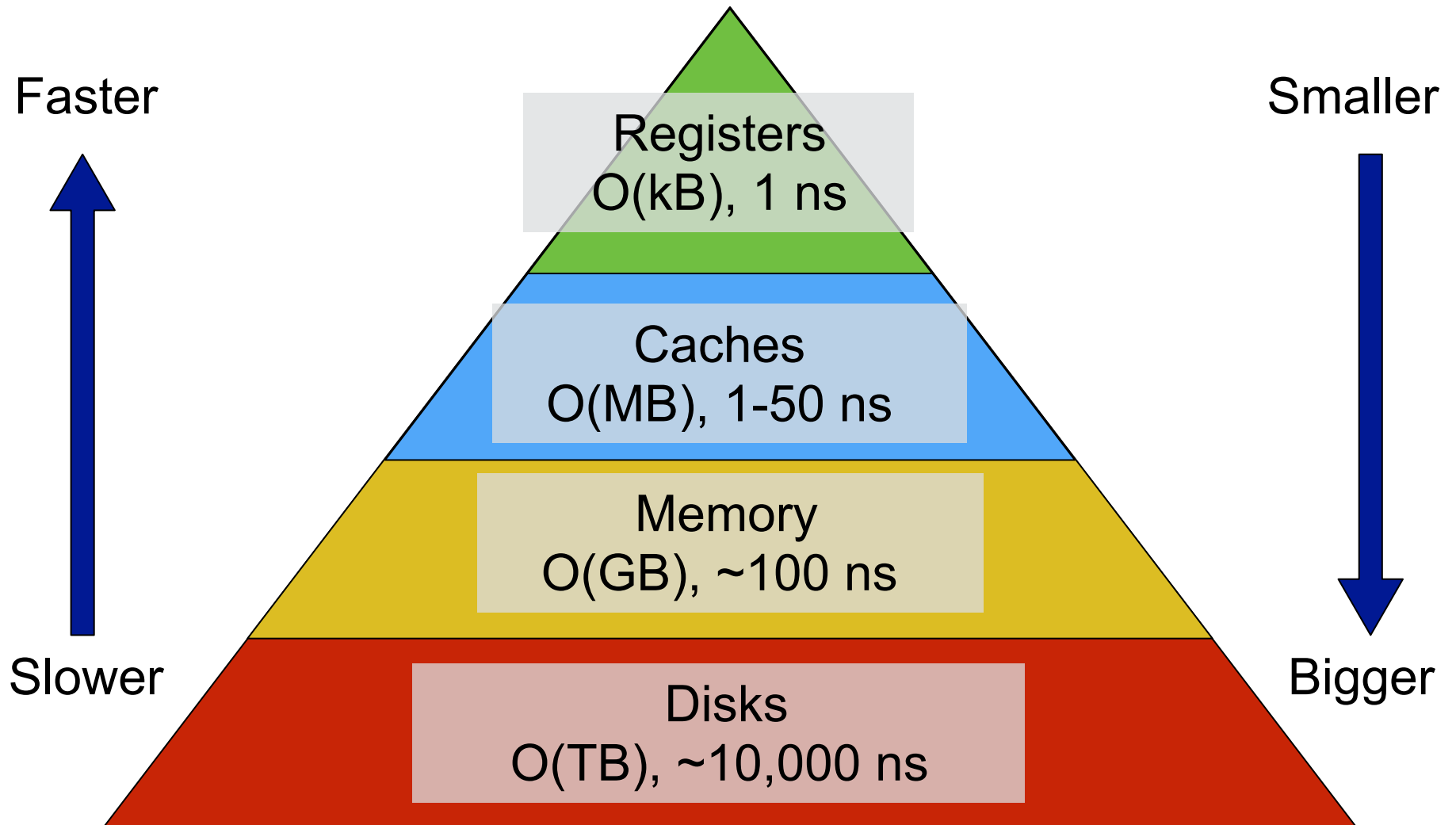
The Memory Hierarchy



Apple
laptop

Disks
 $O(\text{TB}), \sim 10,000 \text{ ns}$

The Memory Hierarchy





The Memory Hierarchy in a Nutshell

- When a program accesses a byte in memory
 - It checks whether the byte is in cache, and if so, it just gets it (and puts it in a register)
 - Otherwise, the byte value is brought from the (slow) memory into the (fast) cache
 - The values **around the byte** are also brought into the cache
- This can happen at all levels
 - Each level of the hierarchy serves as a “cache” for the level below it



The Memory Hierarchy: Analogy

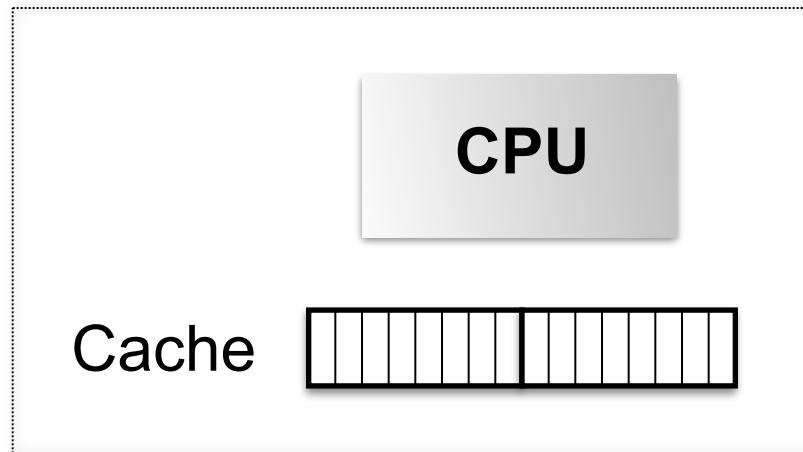
- To write a paper at your desk at home you need a reference book from the library
- You go to the library and find the book on a shelf, noticing that the books around it are on the same topic! You can...
 - **Option #1:** Leave the book at the library and go to the library each time you need one reference
 - **Option #2:** Take only the one book and reuse it at will... but if it makes a reference to another book on the same topic you'll have to go back to the library
 - **Option #3:** Take the one book and the books around it and put them on your desk... and if the reference makes a reference to another book, maybe you'll have the referred book right there
- Option #3 above is: “your desk is a cache for the library”
 - The set of books you grabbed is called a “cache line” or “memory line”

Misses and Hits

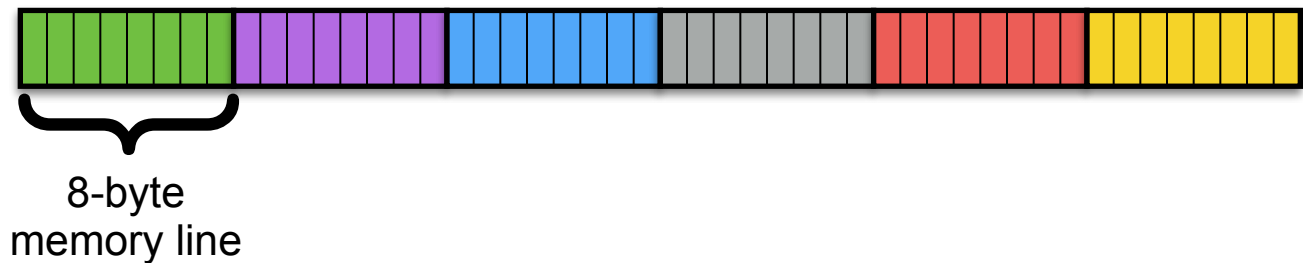
- **Cache hit:** the processor references an address, and the data at that address is in cache
 - The good case
 - You hope for most of your references to be hits
- **Cache miss:** the processor references an address, and the data at that address is not in cache
 - The bad case, which takes much more time
 - **A memory line is brought into the cache**
 - The bytes you need and some bytes around it
 - So that next time, all those bytes will be in cache
- Let's see this on a picture...

Cache/Memory Lines

Processor

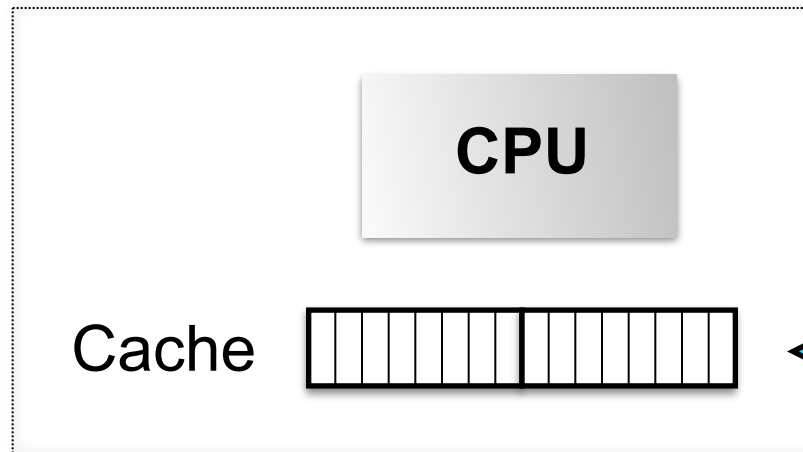


Memory



Cache/Memory Lines

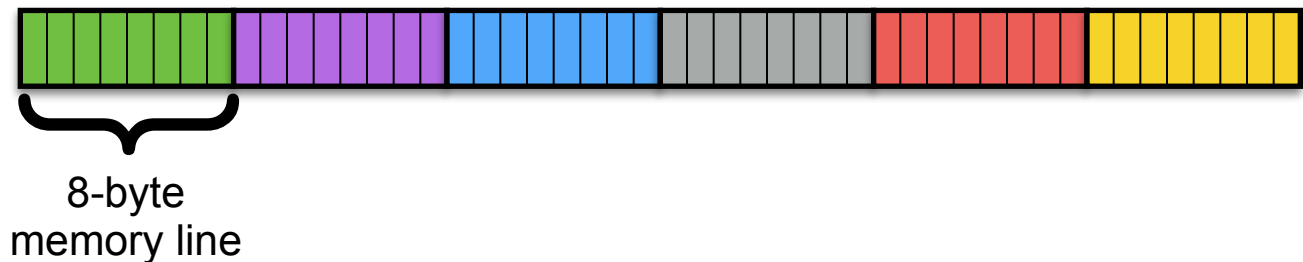
Processor



Cache space for
2 memory lines

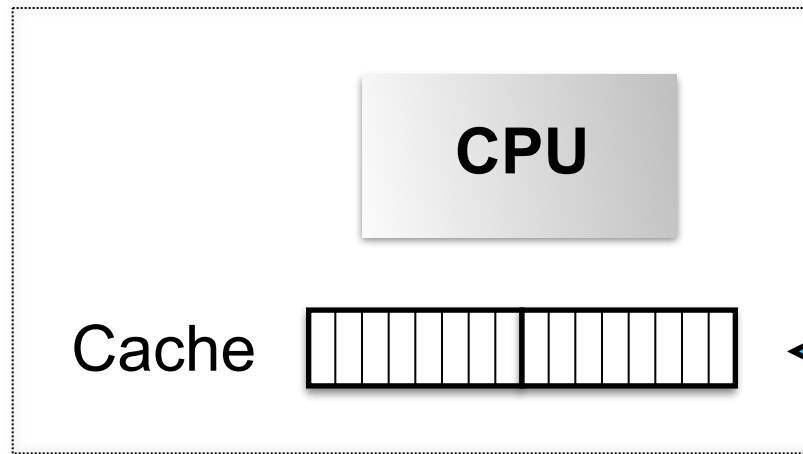
Array that fits in 6
memory lines

Memory



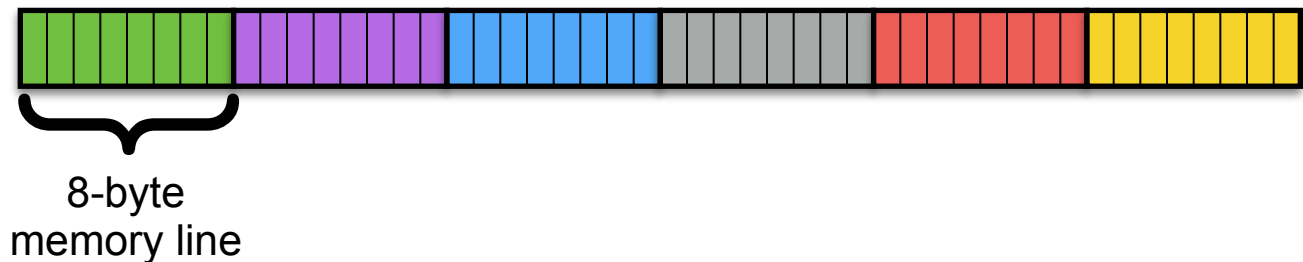
Cache/Memory Lines

Processor



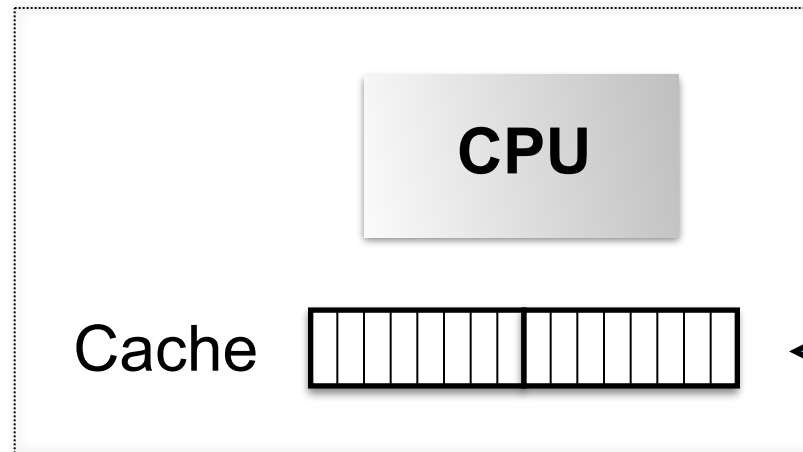
Program says:
"I want byte at
address 20"

Memory



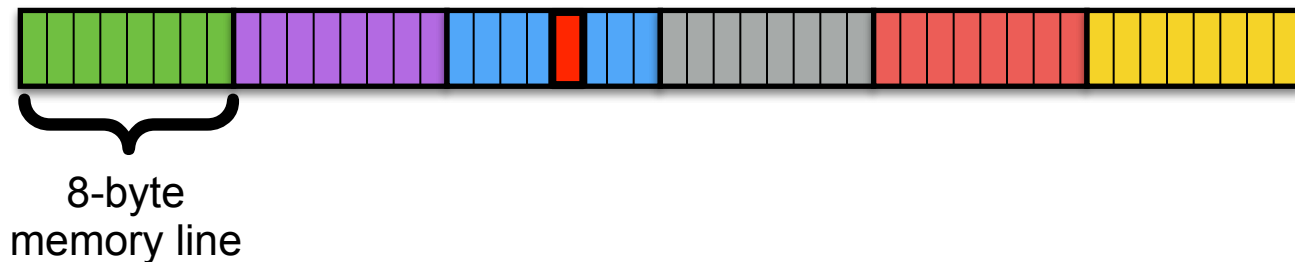
Cache/Memory Lines

Processor



Program says:
"I want byte at
address 20"

Memory



Cache/Memory Lines

Processor

CPU

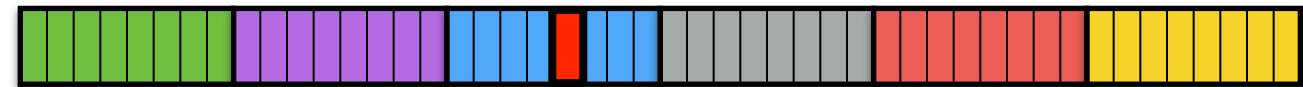
Cache

Program says:
"I want byte at
address 20"

cache
miss

Bring whole line from RAM to Cache

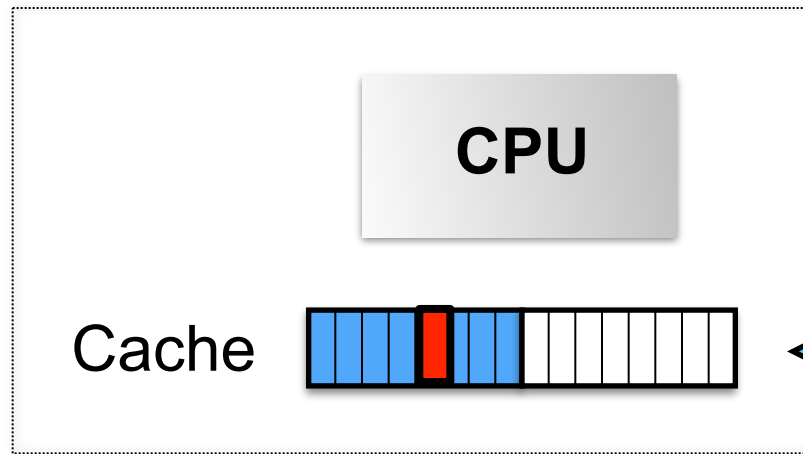
Memory



8-byte
memory line

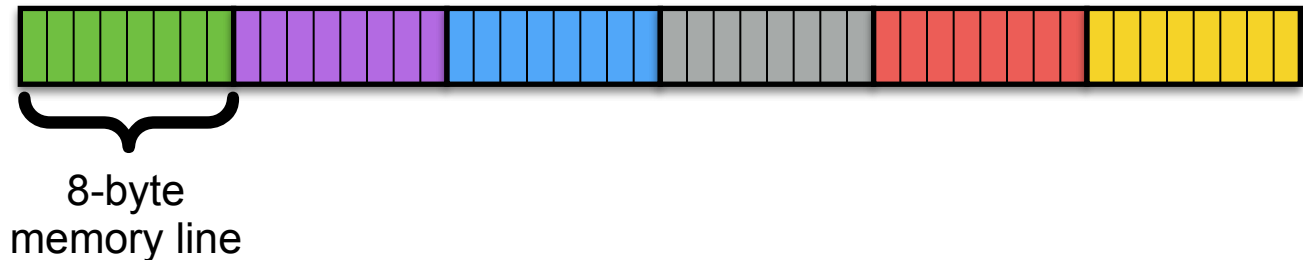
Cache/Memory Lines

Processor



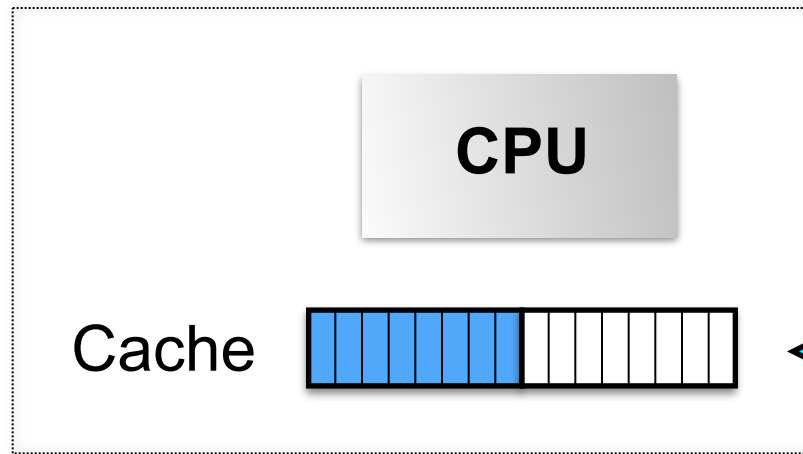
Program says:
"Great, now I can
access it"

Memory



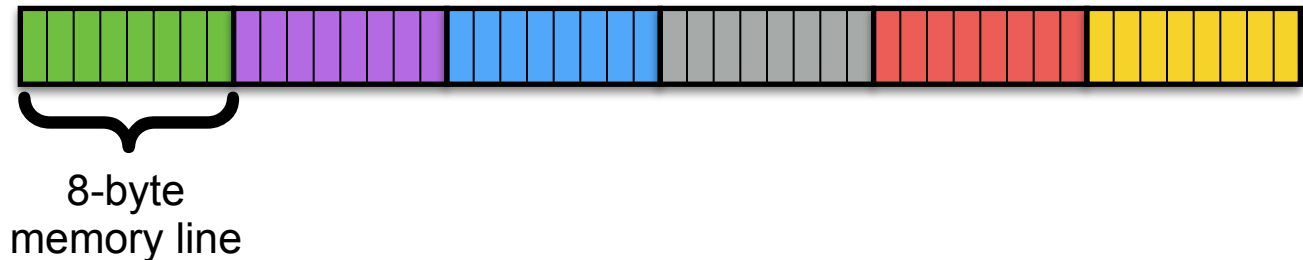
Cache/Memory Lines

Processor



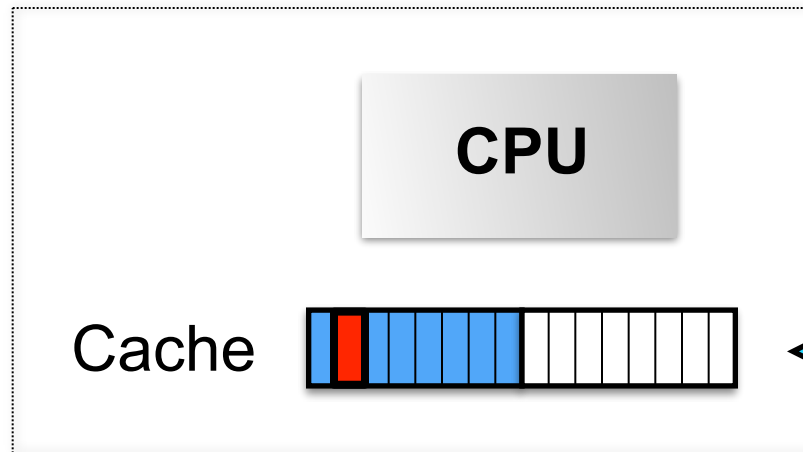
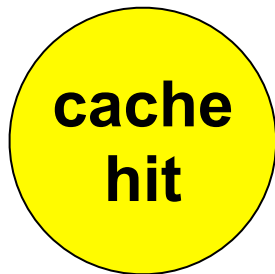
Program says:
"I want to access
by at address 17"

Memory



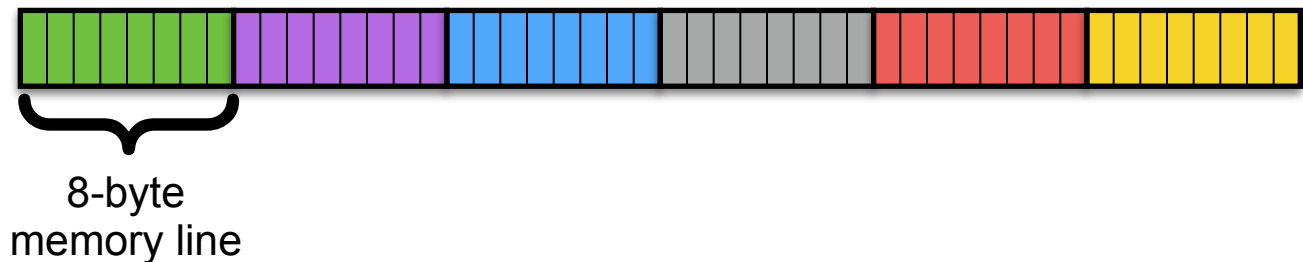
Cache/Memory Lines

Processor



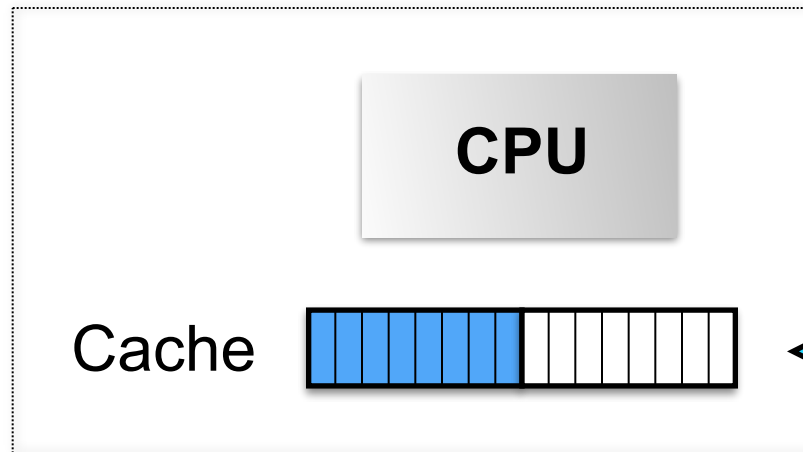
Program says:
"Great! It's
already in cache"

Memory



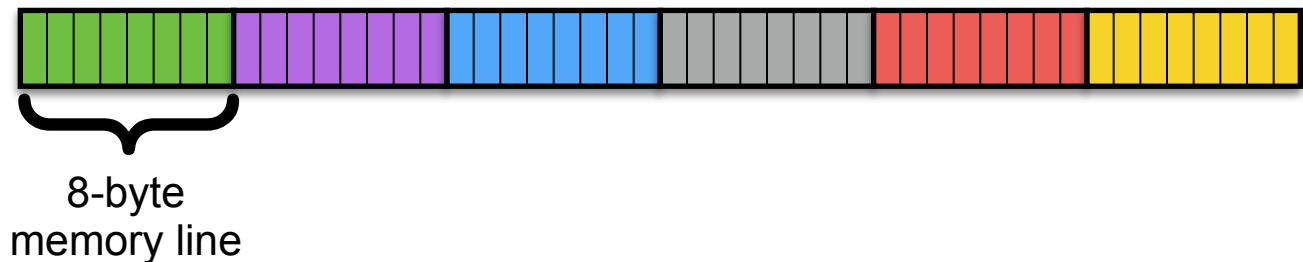
Cache/Memory Lines

Processor



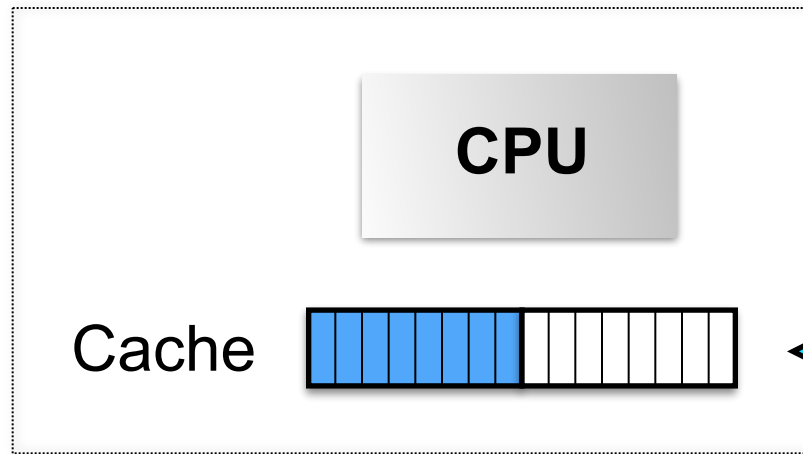
Program says:
"I want byte at
address 5"

Memory



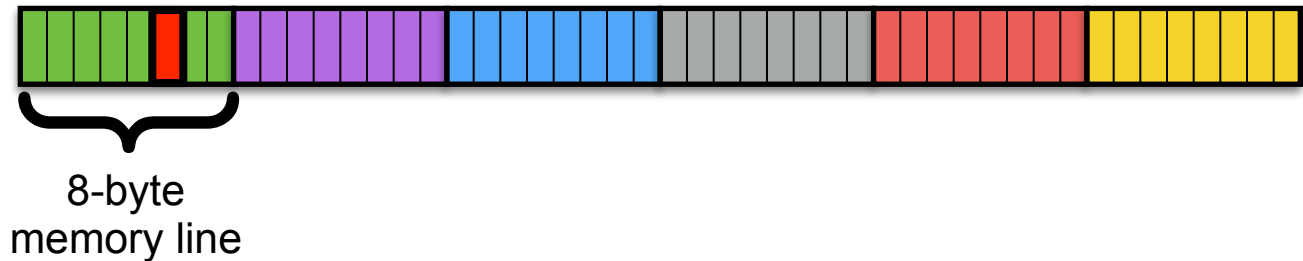
Cache/Memory Lines

Processor



Program says:
"I want byte at
address 5"

Memory



Cache/Memory Lines

Processor

CPU

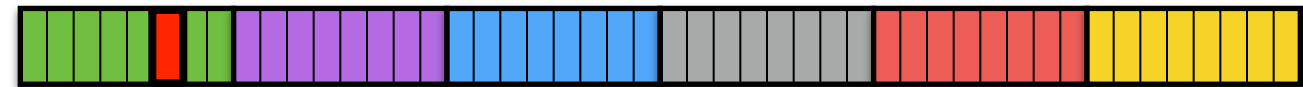
Cache

Program says:
"I want byte at
address 5"

cache
miss

Bring cache line from RAM to Cache

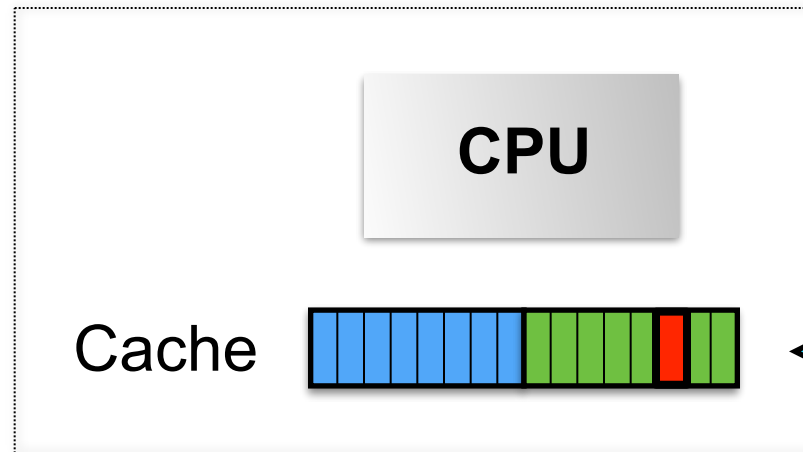
Memory



8-byte
memory line

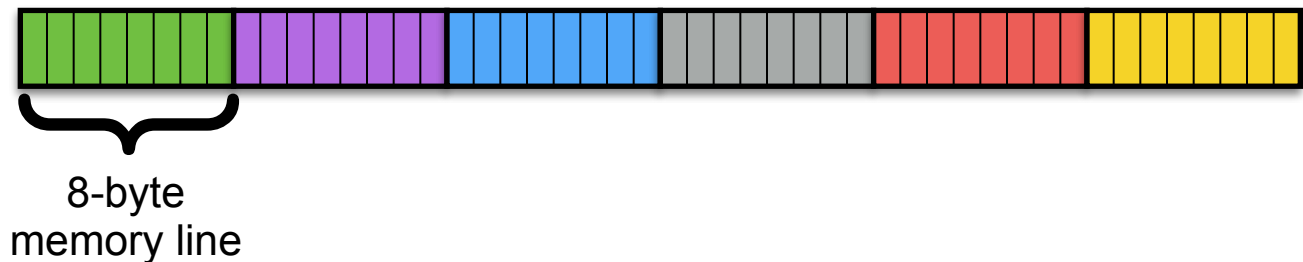
Cache/Memory Lines

Processor



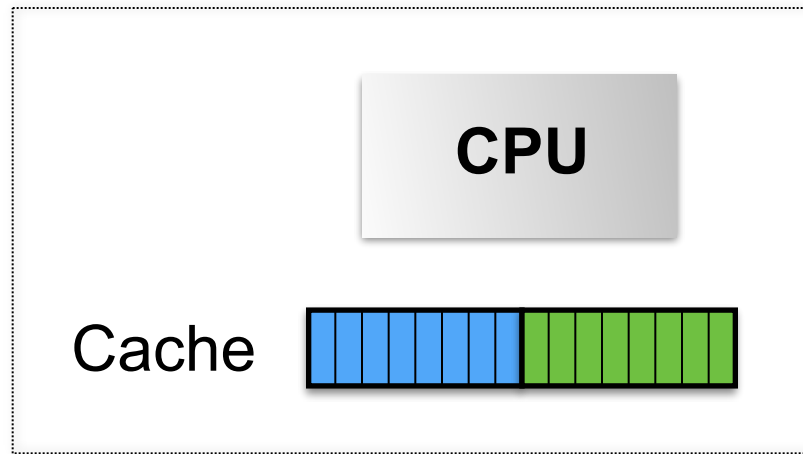
Program says:
"Great, now I can
access it"

Memory



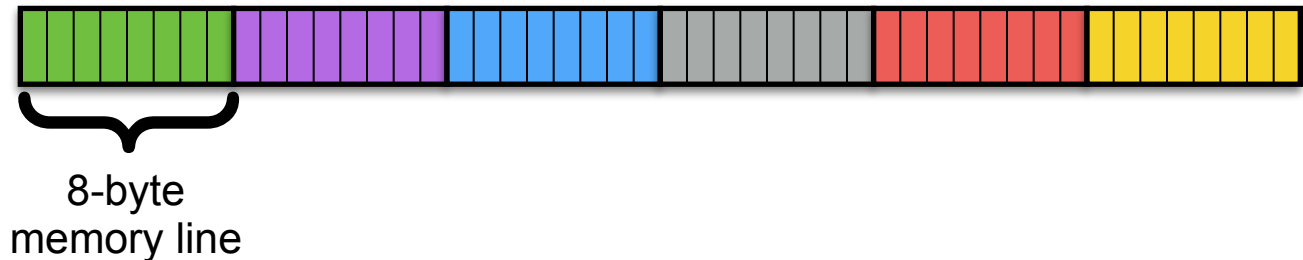
Cache/Memory Lines

Processor



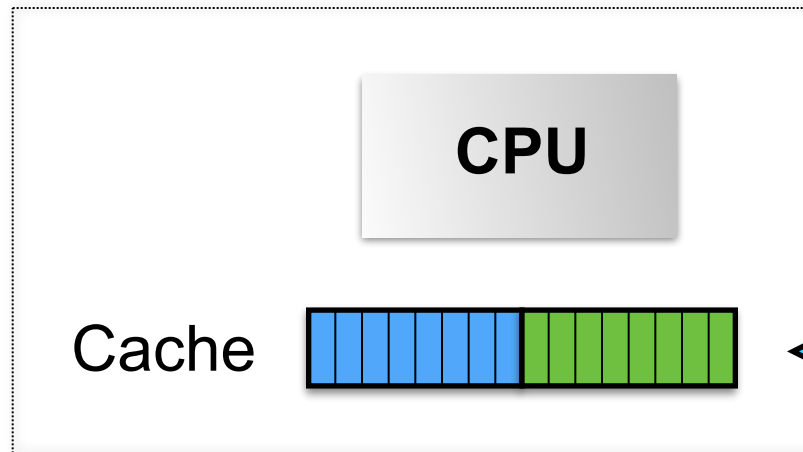
And now, the cache is full!

Memory



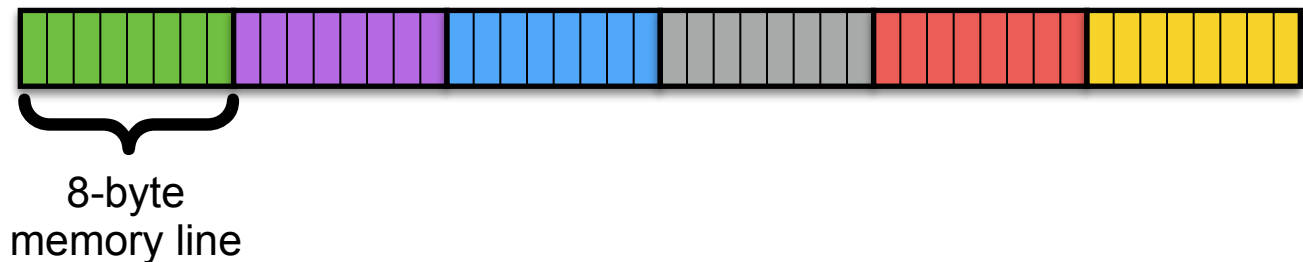
Cache/Memory Lines

Processor



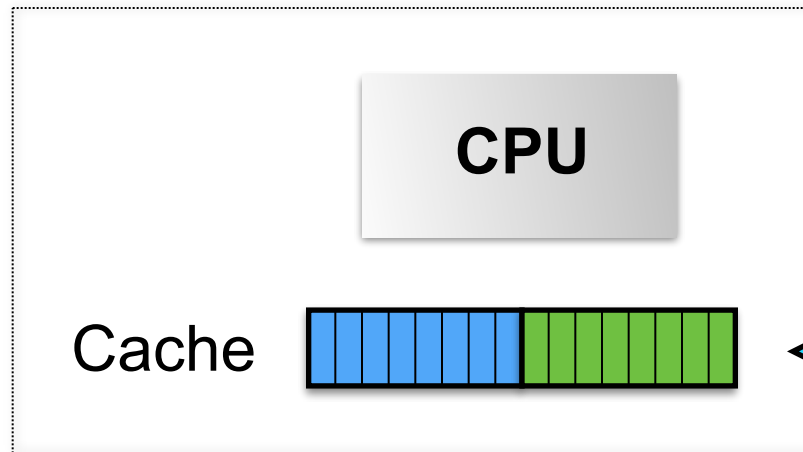
Program says:
"I want byte at
address 43"

Memory



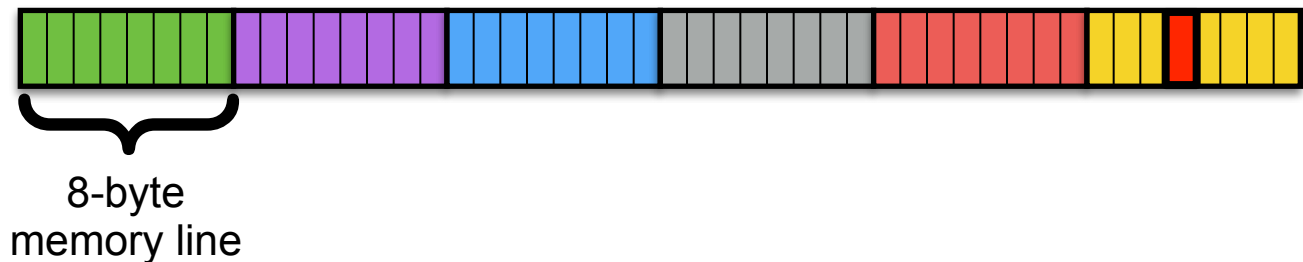
Cache/Memory Lines

Processor



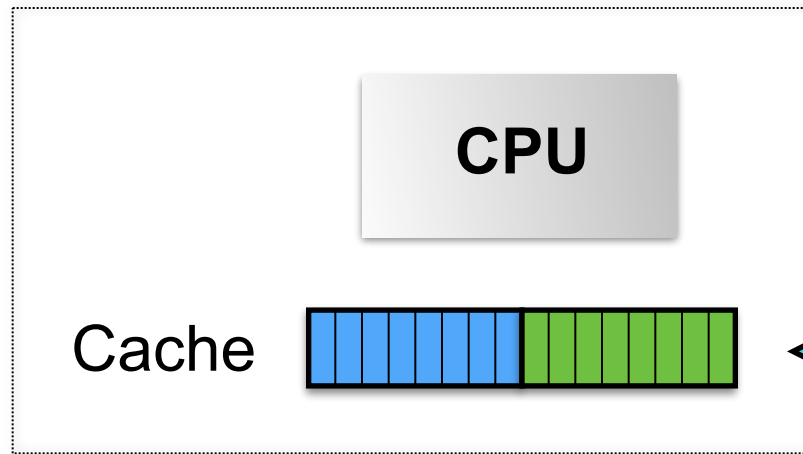
Program says:
"I want byte at
address 43"

Memory



Cache/Memory Lines

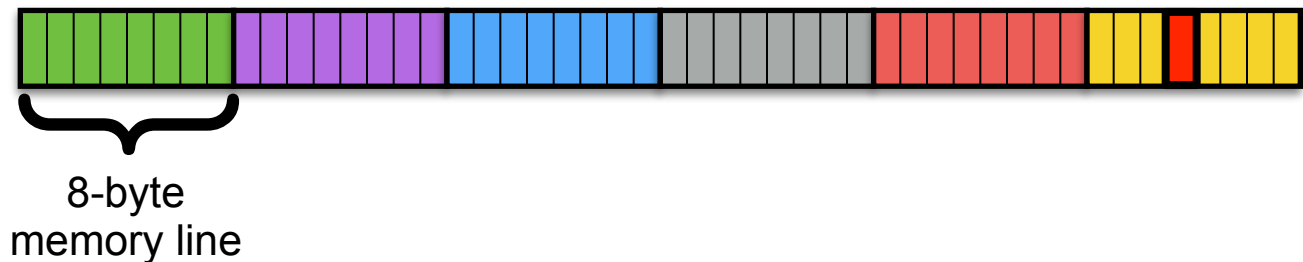
Processor



Program says:
"I want byte at
address 43"

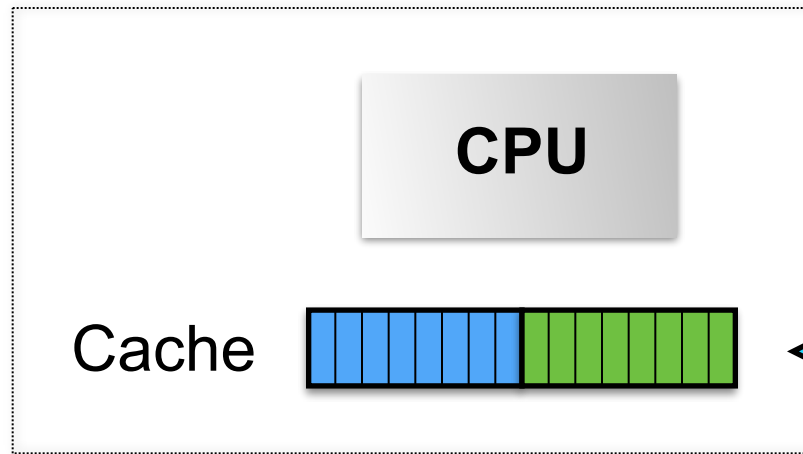
We need to "evict" a memory line from the cache to create space (say the blue one)

Memory



Cache/Memory Lines

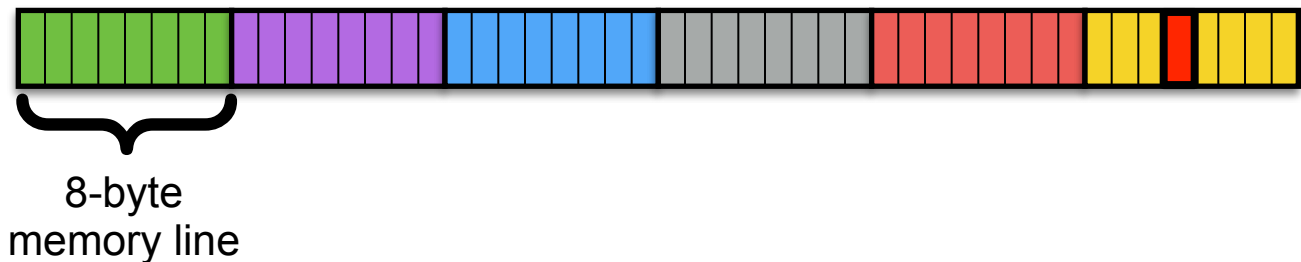
Processor



Program says:
"I want byte at
address 43"

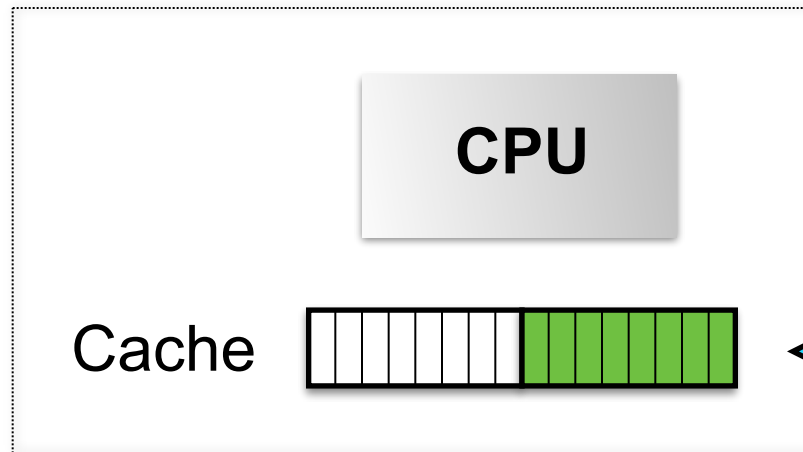
Let's say we evict the Least Recently Used (LRU) line from the cache (blue one)

Memory



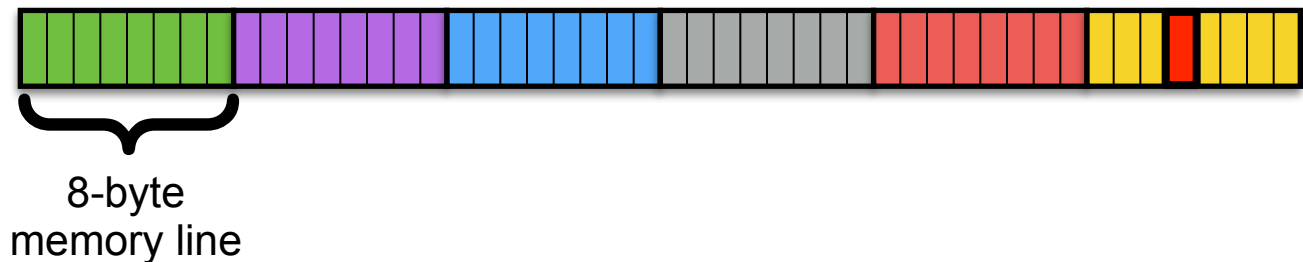
Cache/Memory Lines

Processor



Program says:
"I want byte at
address 43"

Memory



Cache/Memory Lines

Processor

CPU

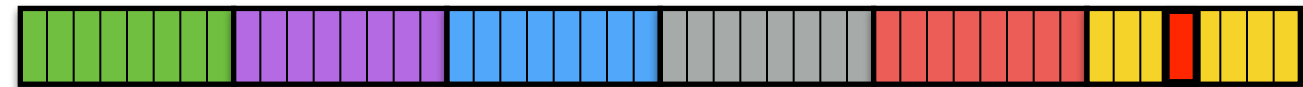
Cache

Program says:
"I want byte at
address 43"

cache
miss

Bring cache line from RAM to Cache

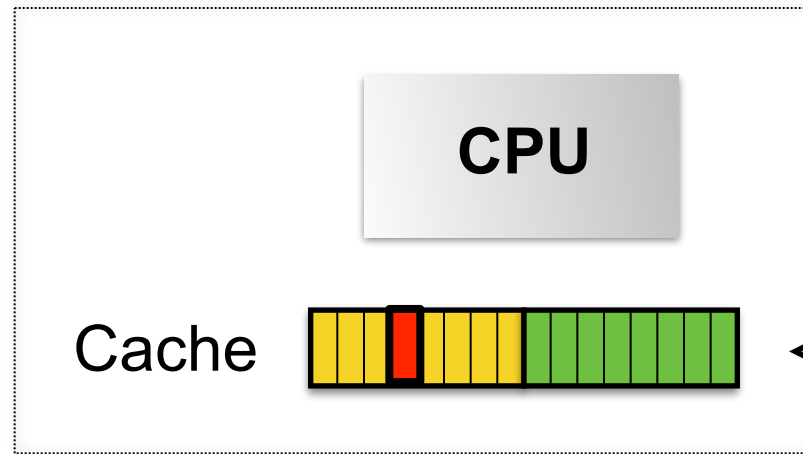
Memory



8-byte
memory line

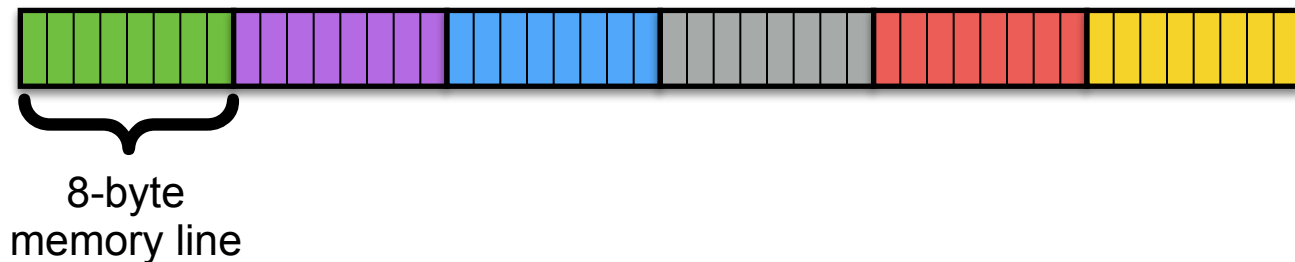
Cache/Memory Lines

Processor



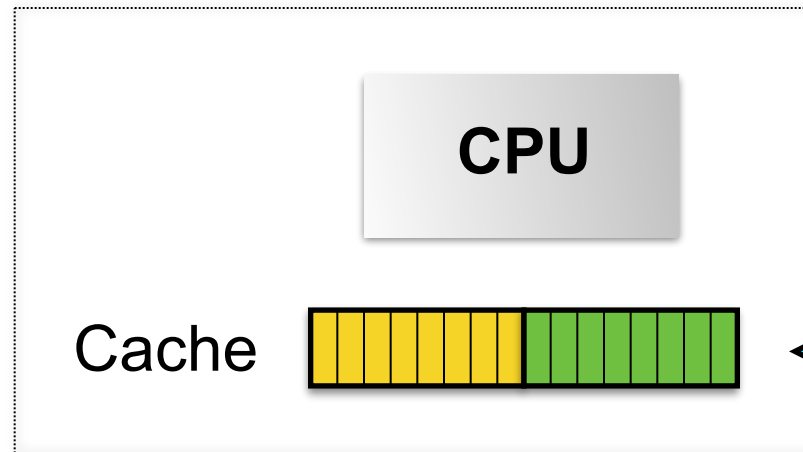
Program says:
"Great, now I can
access it"

Memory



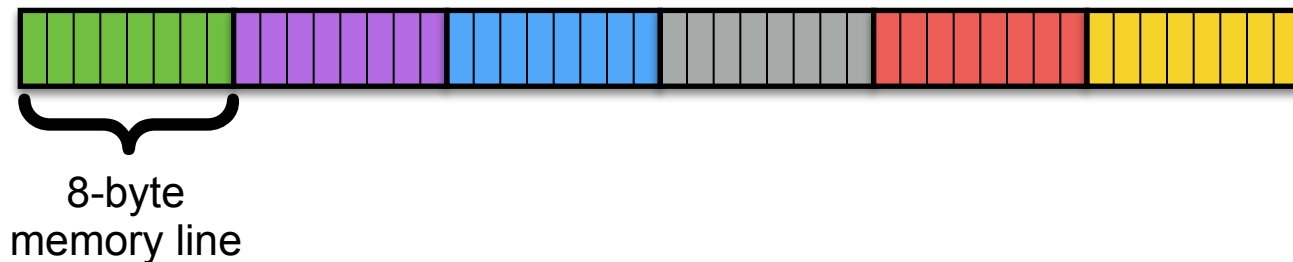
Cache/Memory Lines

Processor



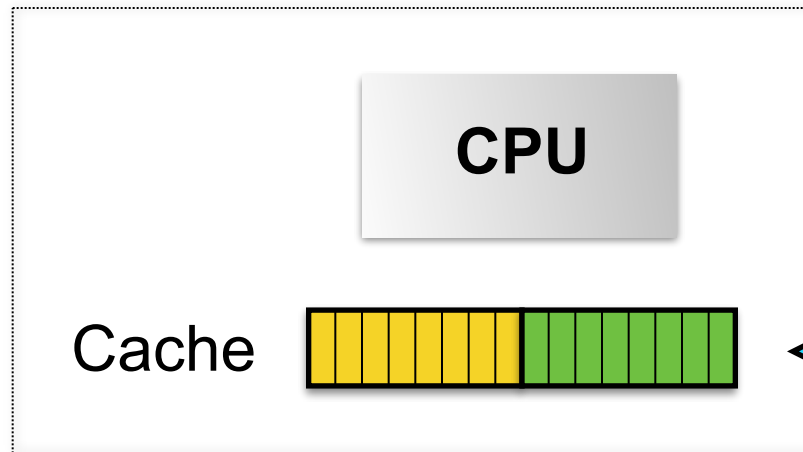
Program says:
"I want byte at
address 12"

Memory



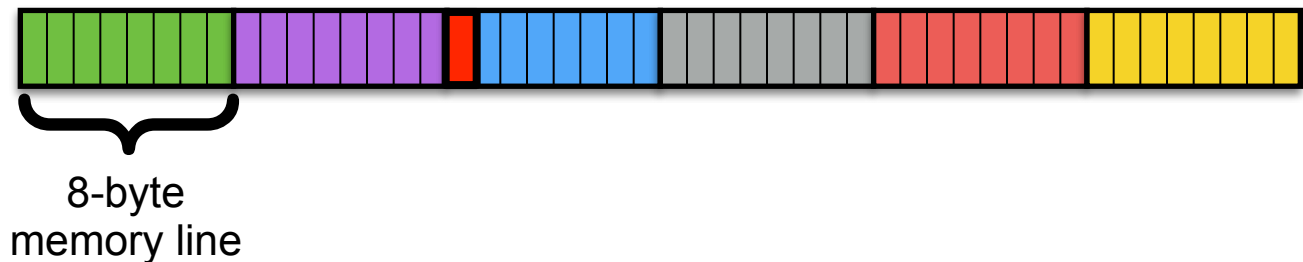
Cache/Memory Lines

Processor



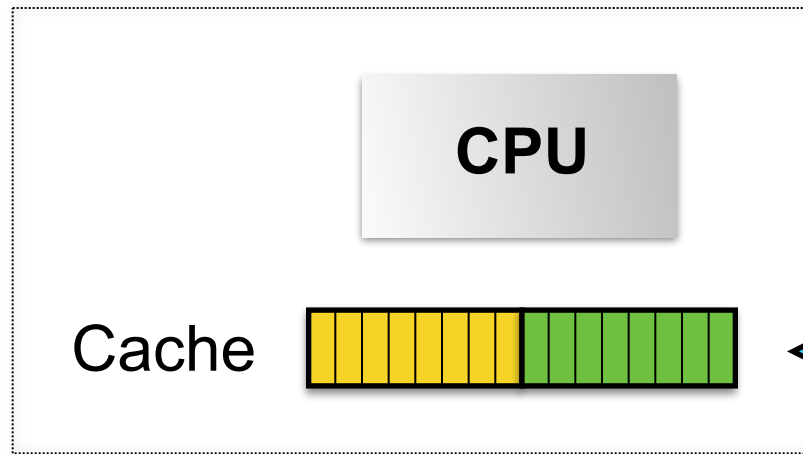
Program says:
"I want byte at
address 12"

Memory



Cache/Memory Lines

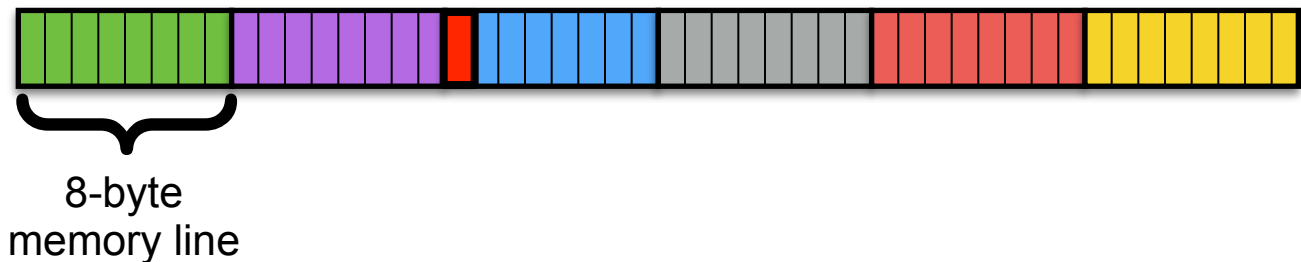
Processor



Program says:
"I want byte at
address 12"

We had the blue line in cache, but evicted it, so now we'll incur another cache miss...

Memory





All this Happens in Hardware

- All cache management is done in hardware
- The OS or the programmer can't influence how the cache works
- Real hardware is more complex than what we saw in the previous slides
 - Several levels of cache (the “hierarchy”)
 - What happens on a write? (update only the cache or both the cache and the memory?)
 - Which cache lines should be evicted?
 - What happens with multiple cores?
 - See a Computer Architecture course
- But regardless, why does it all work?



Locality in your Programs

- The memory hierarchy is useful because of “locality”
- **Temporal locality:** a memory location that was referenced in the past is likely to be referenced again
 - If you reference a byte, you’ll reference it again soon (think of updating a counter)
- **Spatial locality:** a memory location next to one that was referenced in the past is likely to be referenced in the near future
 - If you reference a byte, you’ll soon reference a byte close to it (think of going through an array)



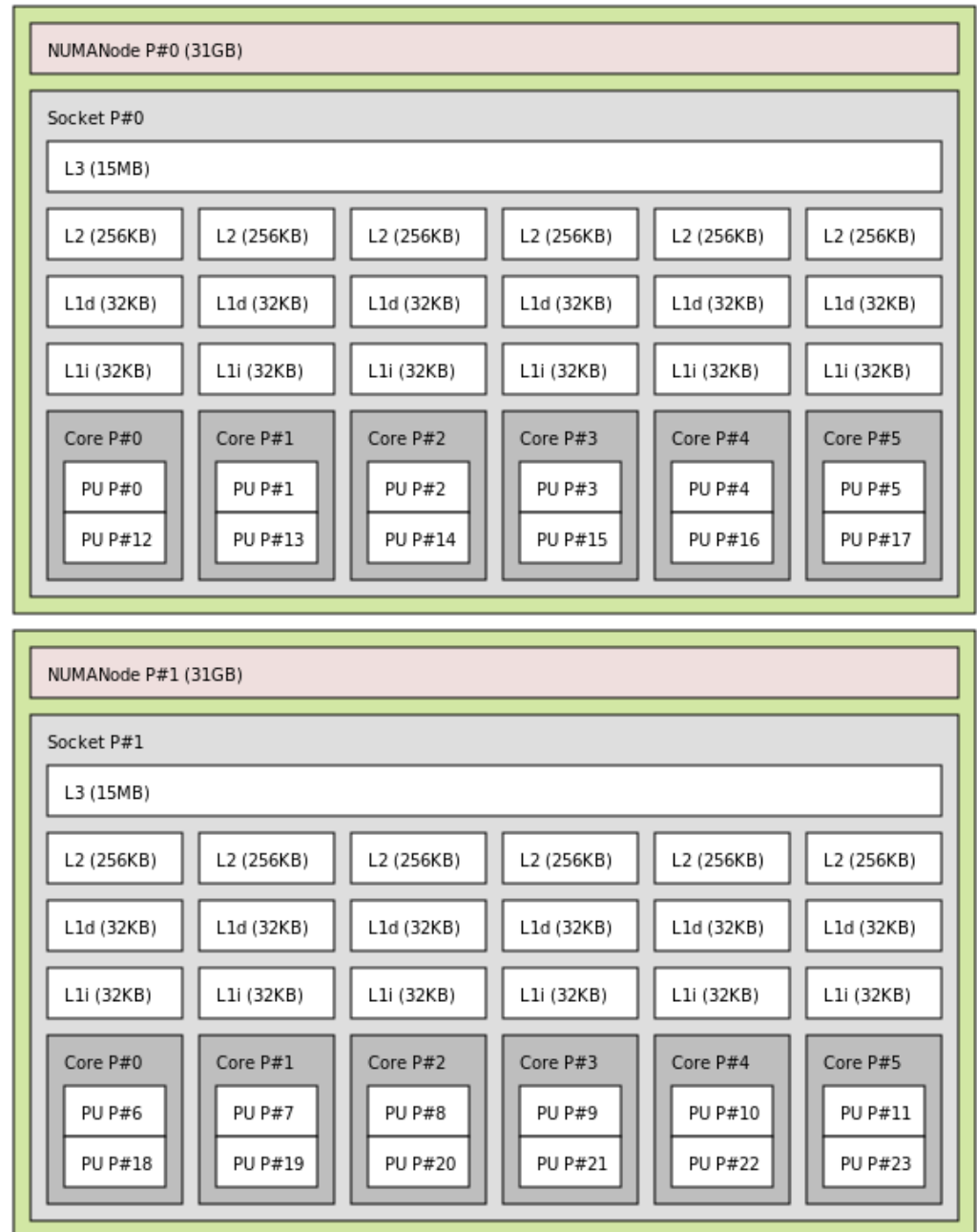
Locality for the developer

- In general, all useful programs have some locality
- But programming for best locality is a well-known challenge (see ICS432, ICS433, ICS621)
- This means we can write a program with horrible locality just to see how bad it is

A real system

Picture obtained with lstopo on my Linux server (sudo apt-get install hwloc)

Machine (63GB)



Direct Memory Access (DMA)

- Often, one has to copy large chunks of data to/from RAM from/to some peripheral device (graphics card, network card, sound card, disk)
- In the pure Von-Neumann model, the CPU has to be involved for each copy operation
- The problem is that memory copies take a long time (even with caches), and the CPU spends its life twiddling its thumbs while the copies are taking place
- It would be better to have copies occur independently so that the CPU can do something useful while the memory copies are taking place
- This is called **Direct Memory Access**
 - The “let’s not have the CPU do this” is a common theme

Direct Memory Access (DMA)

- DMA is used on all modern computers
 - e.g., the M1 chip on my laptop has a (pretty fancy) DMA controller
- How DMA works (without getting into details):
 - The CPU simply tells the DMA controller to initiate a RAM copy
 - When the copy is complete the DMA controller tells the CPU “it’s done” by generating an interrupt (more on interrupts very soon)
 - In the meantime, the CPU was free to do whatever



Direct Memory Access (DMA)

- To perform data transfers the DMA controller uses the memory bus
- In the meantime, the code executed by the CPU likely also uses the memory bus
- Therefore, they can interfere with each other
- There are several ways in which this interference can be managed (give priority to DMA, to CPU, weight usage, ...)
 - See a Computer Architecture course
- In general, using DMA leads to much better performance anyway and (good) software should use it as often as possible



Conclusion

- This concludes a lightning fast review/overview of computer architecture
- You don't need to be computer architecture experts for this course
- But since the OS is in charge of interacting with the hardware, you need to know these basics
- And many of the principles behind what we've talked about in this module are reused in software by the OS (and programs in general)