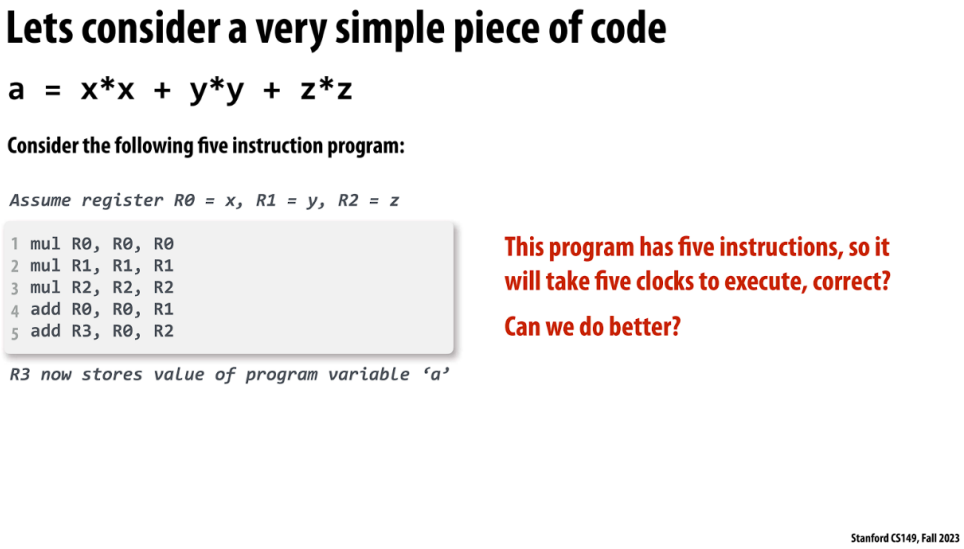
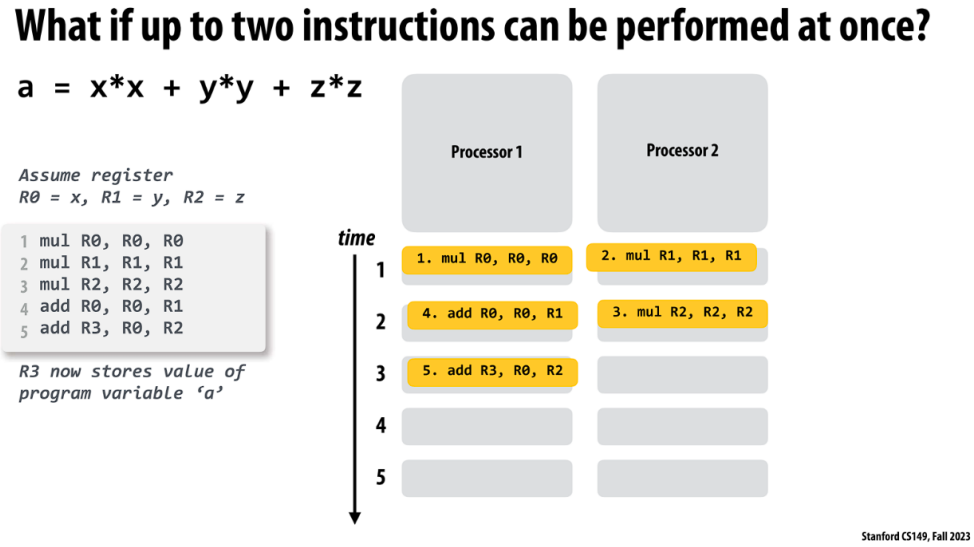
HIGH PERFORMANCE COMPUTING

Done by:
Yosef Alsheikh qasem



In this slide, we're dealing with a straightforward code snippet for a simple calculation.

The assembly code provided to discover that it consists of five instructions, indicating that it will take five units of time to execute on a single core processor. However, the question arises: How much can we improve performance by utilizing multiple cores?



Let's optimize a simple program (a = x * x + y * y + z * z) for a dual-core processor.

To do this effectively, we need to understand how instruction dependencies affect execution speed.

Dependencies exist when an instruction's output is required as input for another instruction.

In our example, the first three instructions (multiplications) are independent.

Their inputs (x, y, and z) are available from the start, and their outputs don't affect each other.

Dual cores can execute independent instructions simultaneously.

Here, instructions 1 and 2 can run on separate cores in the first clock cycle, potentially halving the execution time compared to a single core.

Instruction 4 (addition) depends on both outputs from instructions 1 and 2.

This creates a bottleneck. It can only run after both 1 and 2 finish (cycle 1).

Instruction 3 (another multiplication) is independent of instructions 1, 2, and 4, It can theoretically run in cycle 1 alongside instructions 1 and 2 on a separate core.

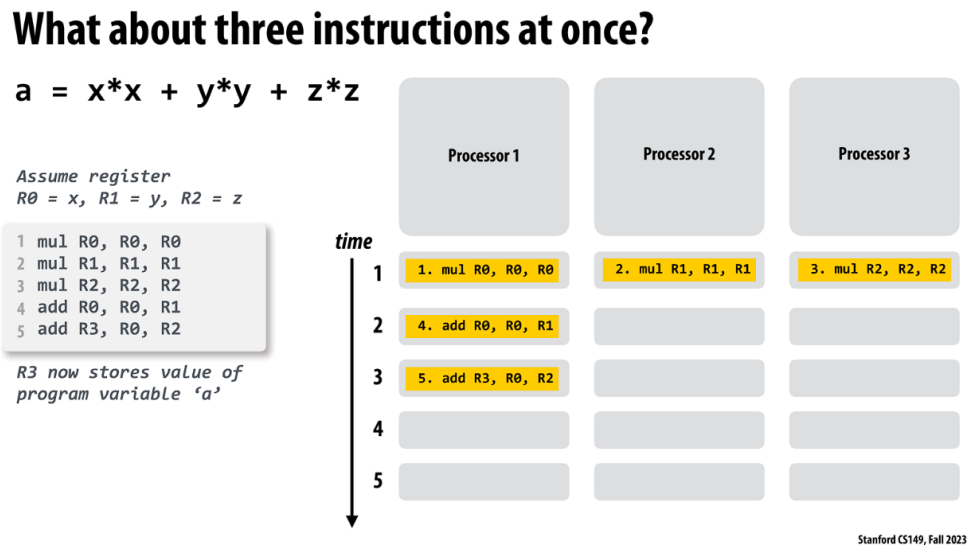
By exploiting these dependencies, we can schedule instructions efficiently:

Instructions 1 and 2 run in cycle 1.

Instruction 3 and Instruction 4 (dependent on 1 and 2) runs in cycle 2.

Instruction 5 (dependent on 3 and 4) runs in cycle 3.

This approach reduces the total execution time from 5 cycles (single core) to 3 cycles (dual core).



We've optimized our code for dual cores, achieving 3 cycles from 5, But what about using 3 cores? Will it squeeze even more performance?

The answer, in this specific case, is no.

Remember the dependencies? Instruction 4 relies on the outputs of 1 and 2, forcing it to wait until cycle 2.

Similarly, instruction 5 depends on both 3 and 4, delaying it until cycle 3.

Even with 3 cores, instruction 4 remains dependent and can't start earlier. Likewise for instruction 5. While core 3 sits idle waiting for instruction 4, we've gained no performance boost compared to dual cores.

This example highlights that:

Just throwing more cores at a problem doesn't guarantee faster execution.

Instruction dependencies control the potential for parallelism.

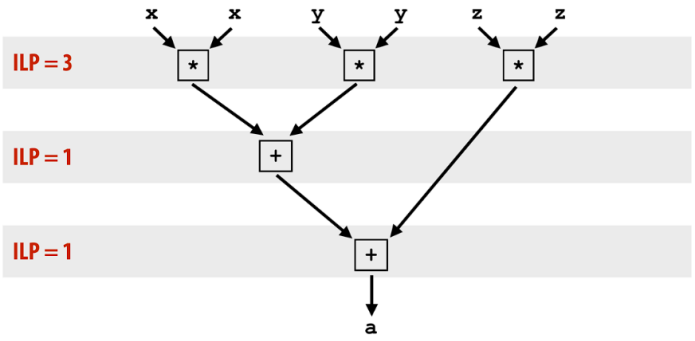
Understanding these dependencies is key to optimizing code for multi-core processors.

If we had more independent instructions to distribute across the 3 cores, we could see a significant performance improvement, the key lies in the code itself and how effectively it can be parallelized.

Instruction level parallelism (ILP) example

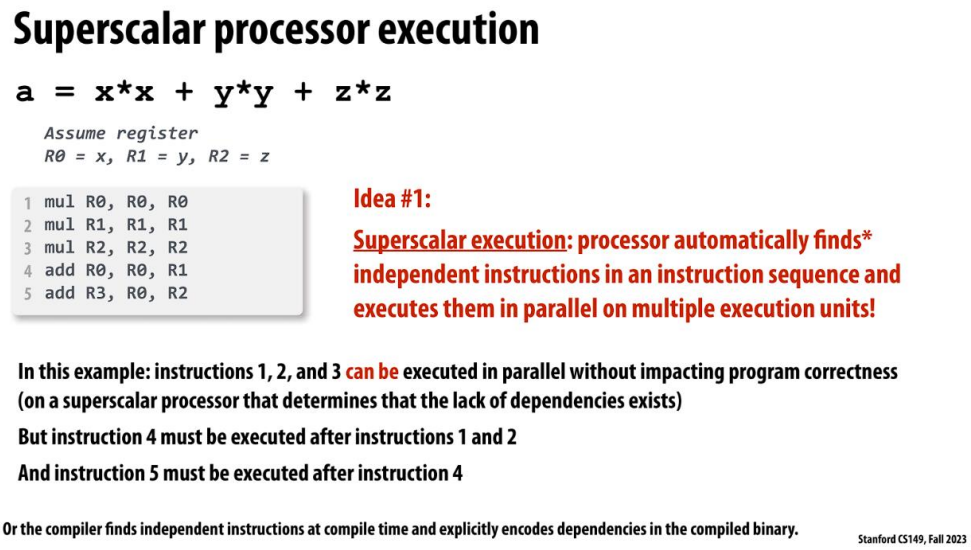
- ILP = 3

```
a = x*x + y*y + z*z
```



Stanford CS149, Fall 2023

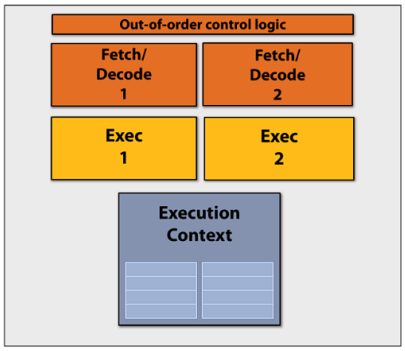
This slide presents a clear explanation of the dependencies between instructions, making it easy to understand.



In superscalar execution, also known as instruction-level parallelism, processors execute instructions by identifying and exploiting dependencies. This allows the processor to execute code out of order, based on the dependencies among instructions and execution elements.

Superscalar processor

This processor can decode and execute up to two instructions per clock



Stanford CS149, Fall 2023

Traditional processors handle instructions one at a time, creating a bottleneck, superscalar execution breaks free from this limitation by introducing multiple processing units.

the components you see in the image:

Control Unit (1): directs the overall flow of instructions.

Fetch/Decode Units (2): Each can grab and interpret an instruction simultaneously, doubling the potential for parallelism compared to a single unit.

Execution Units (2): Each execution unit can carry out an instruction independently, allowing truly parallel processing.

Shared Execution Context (1): This acts as a common ground where instructions can exchange information if needed. While instructions are executed concurrently, they can still access and share data within this context.

With two fetch/decode units and two execution units, this processor can potentially fetch, decode, and execute two instructions at the same time, as long as they're independent, This significantly boosts performance compared to a single-core processor.

Key to Efficiency is Instruction Dependencies But there's a catch. Not all instructions are created equal. Some rely on the outputs of others before they can proceed, one instruction can't move forward until its other finished.

The processor considers these dependencies to ensure smooth execution.

Aside:
Old Intel Pentium 4 CPU

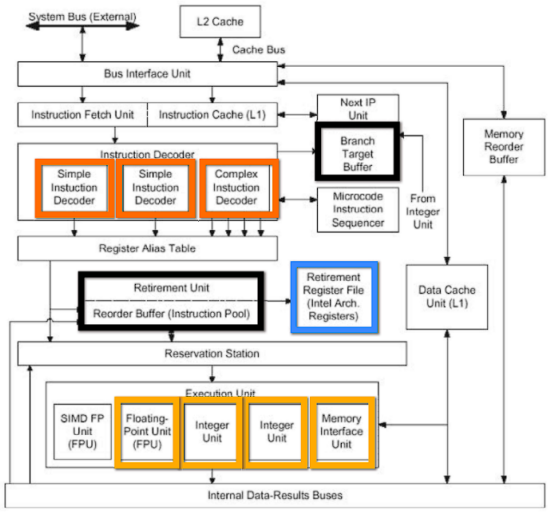
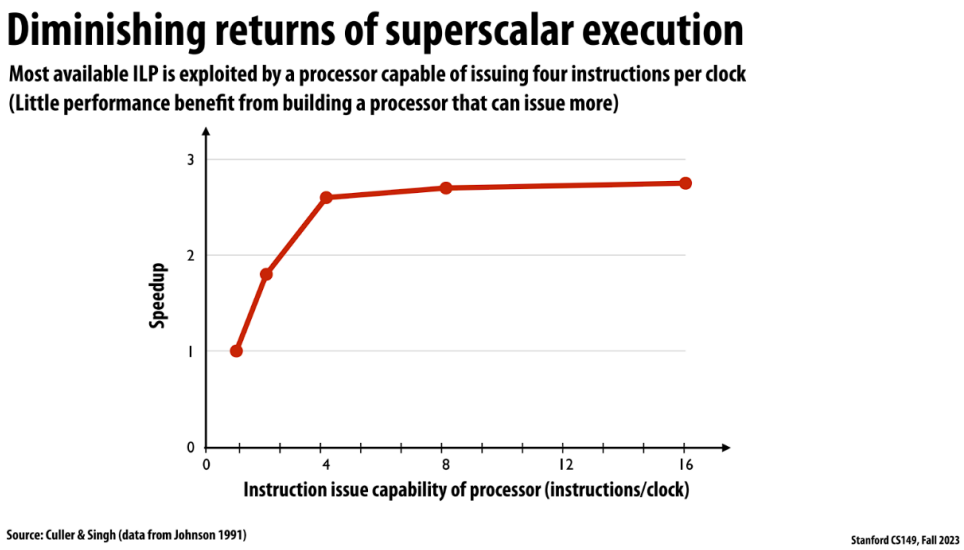


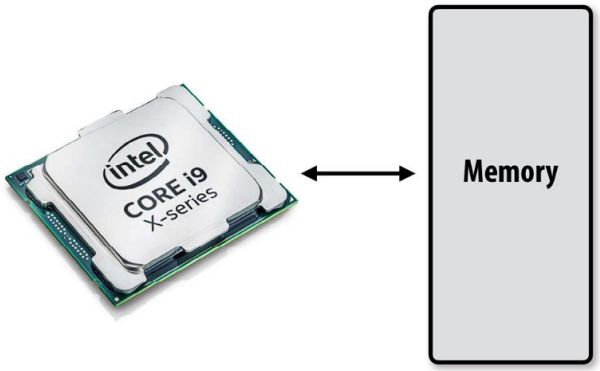
Image credit: http://ixbtlabs.com/articles/pentium4/index.html



Instruction-level parallelism relies on the ability to identify and execute multiple instructions simultaneously, Research indicates that utilizing more than four execution units yields only little performance benefits.

This limitation arises from the difficulty of consistently finding a sufficient number of instructions to execute concurrently.

What is memory?



Achieving efficient processing almost always comes down to accessing data efficiently.

A program's memory address space

- A computer's memory is organized as an array of bytes

- Each byte is identified by its "address" in memory (its position in this array)
  (We'll assume memory is byte-addressable)

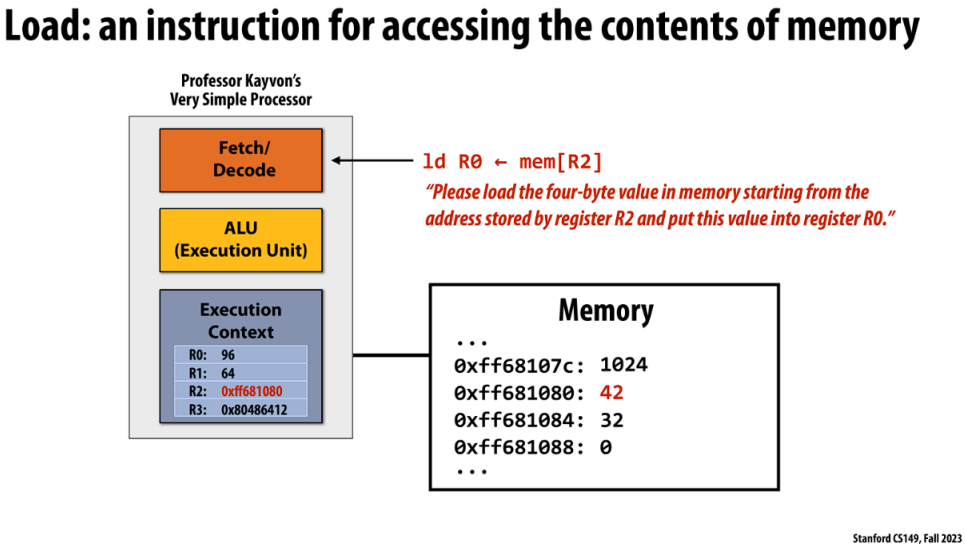
  "The byte stored at address 0x8 has the value 32."

  "The byte stored at address 0x10 (16) has the value 128."

  In the illustration on the right, the program's memory address space is 32 bytes in size (so valid addresses range from 0x0 to 0x1F)

Address	Value
0x0	16
0x1	255
0x2	14
0x3	0
0x4	0
0x5	0
0x6	6
0x7	0
0x8	32
0x9	48
0xA	255
0xB	255
0xC	255
0xD	0
0xE	0
0xF	0
0x10	128
⋮	⋮
0x1F	0

The memory is an array of bytes(each word 8 bits) such that each byte identified by an address .



Consider a load instruction, denoted as "ld", where the processor aims to retrieve a value from memory, This value is then stored in the destination register, R0. The memory address accessed is specified by the content of register R2.

ld R0←mem[R2]

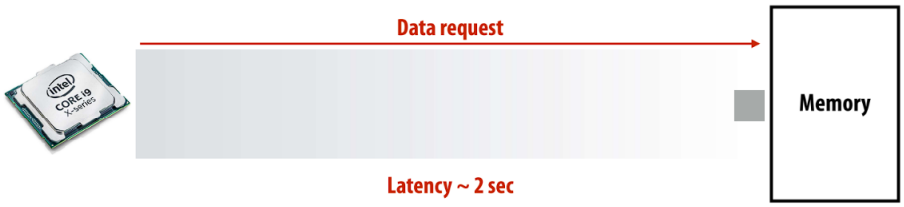
ld: the load instruction

R0: destination register

Mem[R2] : accessing the memory its address is R2.

Terminology

- **Memory access latency**
  - The amount of time it takes the memory system to provide data to the processor
  - Example: 100 clock cycles, 100 nsec



Memory access latency: The time it takes for the processor to access data from memory.

Why Latency Matters?

Faster memory access latency means the processor can get data quicker, leading to faster program execution and improved responsiveness.

Slower memory access latency creates bottlenecks, causing the processor to wait for data, hindering performance.

Stalls

- A processor "stalls" (can't make progress) when it cannot run the next instruction in an instruction stream because future instructions depend on a previous instruction that is not yet complete.

- Accessing memory is a major source of stalls

```
ld r0 mem[r2]
ld r1 mem[r3]
add r0, r0, r1
```

Dependency: cannot execute 'add' instruction until data from mem[r2] and mem[r3] have been loaded from memory

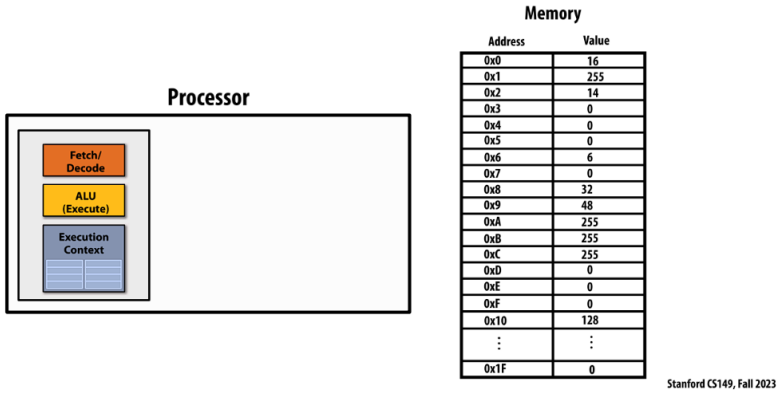
- Memory access times ~ 100's of cycles
  - Memory "access time" is a measure of latency

Stalls occur when the processor must wait because the next instruction in the instruction stream depends on a previous instruction that has not yet finished executing.

Memory access often causes significant stalls because it is considerably slower than using registers or performing arithmetic operations. For instance, fetching data from memory (memory access) may take 100 units of time, while an addition operation might only take 1 unit of time.

What are caches?

- Recall memory is just an array of values
- And a processor has instructions for moving data from memory into registers (load) and storing data from registers into memory (store)



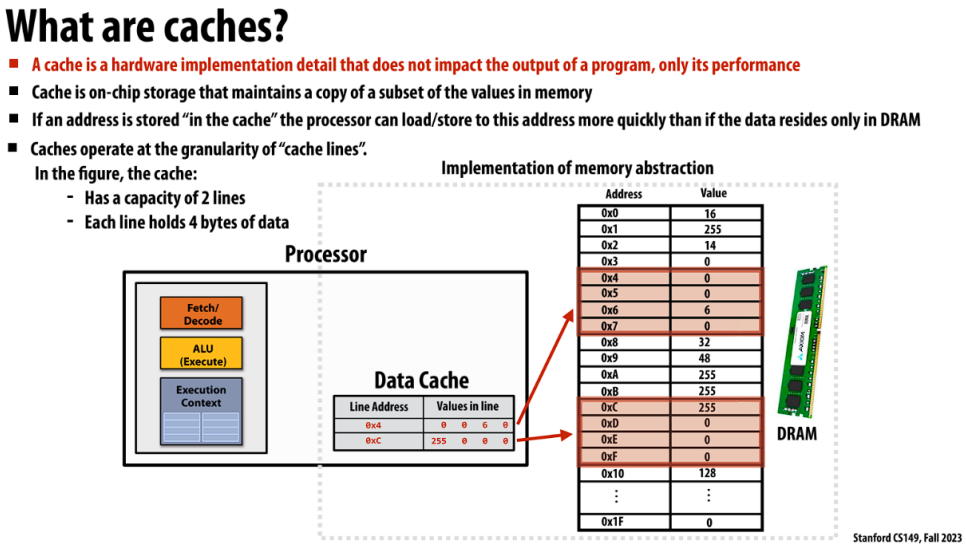
Stanford CS149, Fall 2023

Memory: A Data Array for the Processor

Think of memory as a giant filing cabinet with numbered drawers.

Each drawer holds a piece of data (like a byte).

The processor uses instructions like "load" and "store" to access data in specific drawers (memory addresses).



Cache: A Speedy Shortcut for Data Access

Imagine your computer's memory is like a large library.

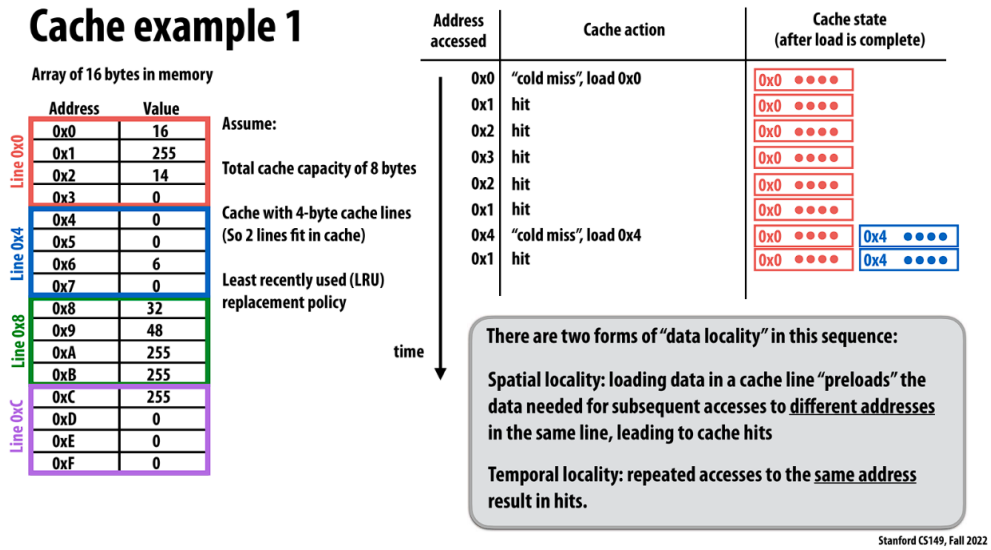
The cache is like a small desk next to your chair where you keep frequently used reference books.

Just like the desk, the cache is a smaller but much faster memory than the main memory.

The processor can access data from the cache much quicker than from the main memory.

The cache holds copies of the most recently used data from the main memory.

Even though the cache isn't perfect (it doesn't store everything), it significantly speeds up data access for the processor, improving overall performance.



In our example, we have a memory consisting of 16 bytes, with a cache featuring 2 lines, each capable of holding 8 bytes.

Here, a "line" refers to a block of data, with each line being represented by every 4 bytes in a row, numbered 0, 4, 8, and c.

Our cache implements the least recently used (LRU) algorithm for data replacement, this means that when adding a new line to the cache and the maximum capacity is reached, the least recently accessed line is removed, based on the assumption that the most recently accessed data is likely to be accessed again soon.

There are two forms of locality discussed in the slide.

Now, let's delve into the hit and miss operations in the table.

Initially, when the processor attempts to access the memory address 0x0, it results in a cold miss since the address is not included in the cache.

Consequently, we add it to the cache without needing to delete any lines.

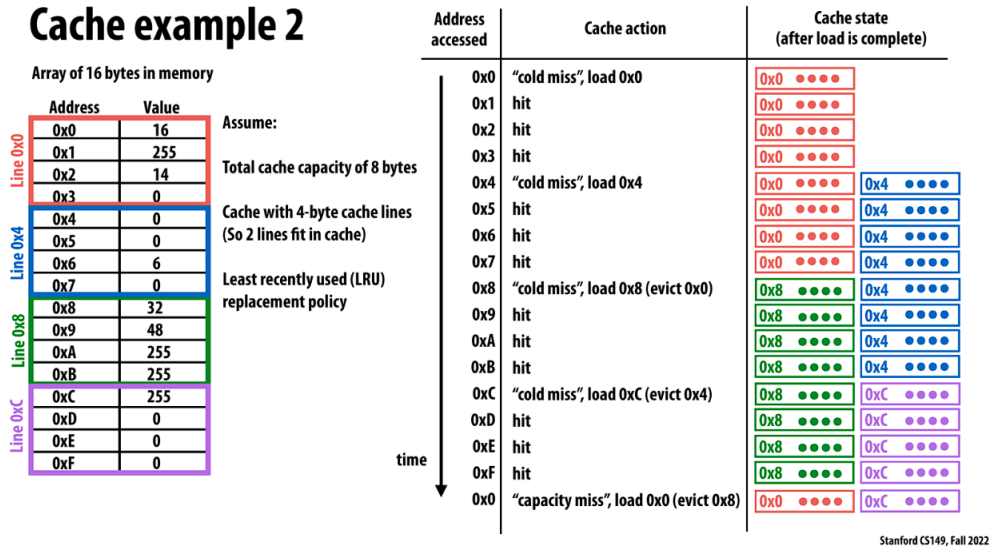
Subsequently, the processor accesses memory locations 0x1, 0x2, 0x3, 0x2, and 0x1 in succession, all of which are found in the cache, resulting in five consecutive hits, and the data is retrieved directly from the cache.

However, when the processor tries to access 0x4, it results in another cold miss, necessitating the addition of the 0x4 line to the cache.

With the cache now containing lines 0x0 and 0x4, the processor attempts to access 0x1, since 0x1 is already present in the cache alongside 0x0, it results in a hit.

In scenarios where the user needs to access, for instance, 0x9, the cache's limited capacity of 2 lines requires the removal of one line to accommodate the new data. I

n this case, line 0x4 is deleted from the cache to make room for 0x8.



we'll continue using the same cache and memory configuration, skipping the initial 8 operations as previously explained, we start with operation number 9, where the processor seeks to access memory address 0x8, However, the cache currently only holds lines 0x0 and 0x4, leading to a miss, Consequently, to add 0x8 line in the cache, the least recently used line (LRU), which is 0x0 in this case, is deleted.

With the cache now containing lines 0x4 and 0x8, subsequent accesses to 0x9, 0xA, and 0xB result in hits since they are all located within the 0x8 line.

However, when the processor attempts to access 0xC, another miss occurs, there is need to delete the

LRU line, which is 0x4, Therefore, 0xC is added to the cache.

The cache now holds lines 0x8 and 0xC. When the processor requests access to 0xD, 0xE, and 0xF, all the accesses result in hits since they are contained within the 0xC line.

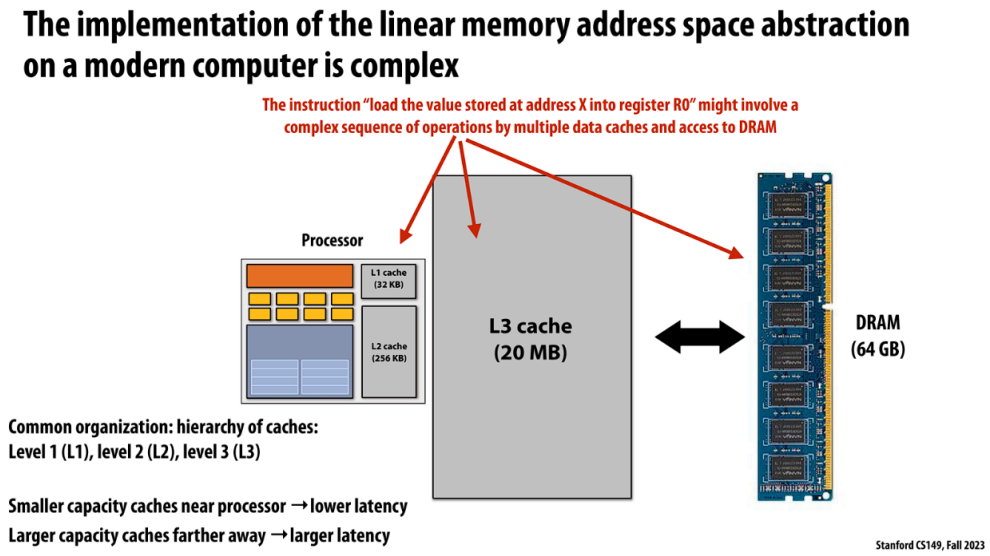
Finally, when the processor seeks access to 0x0, the cache delete the 0x8 line to make room for 0x0, completing the cache operations.

Caches reduce length of stalls (reduce memory access latency)

- Processors run efficiently when they access data that is resident in caches
- Caches reduce memory access latency when processors accesses data that they have recently accessed! *

* Caches also provide high bandwidth data transfer

Stalls primarily occur due to memory access latency. However, utilizing caches can significantly reduce access latency, decreasing the duration of stalls.



To achieve a better performance we have 3 levels of caches:

Level 1: the smallest in the size but fastest access time.

Level 2: faster access time than L3 but slower than L1, and larger capacity than L1 while smaller capacity than L3.

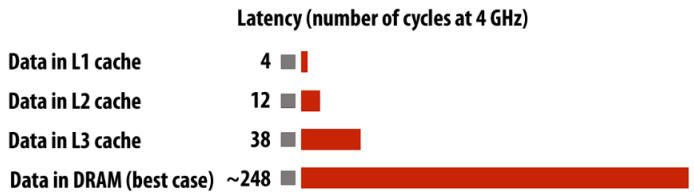
Level 3: the largest capacity with the slowest access time.

As a result:

larger capacity→ higher latency and vice versa

Data access times

(Kaby Lake CPU)



The example shows time latencies for both memory and caches.

Summary

- **Today, single-thread-of-control performance is improving very slowly**
  - To run programs significantly faster, programs must utilize multiple processing elements or specialized processing hardware
  - Which means you need to know how to reason about and write parallel and efficient code
- **Writing parallel programs can be challenging**
  - Requires problem partitioning, communication, synchronization
  - Knowledge of machine characteristics is important
  - In particular, understanding data movement!
- **I suspect you will find that modern computers have tremendously more processing power than you might realize, if you just use it efficiently!**