

# Tools and Techniques, Lecture 3

## **Performance**

- Algorithmic complexity
- Memory access patterns
- Benchmarking



LUNDS UNIVERSITET



CERN  
School of Computing

CERN School of Computing 2025, Lund  
Sten Åstrand, Lund University

# Why?



Image credit: Giulio Eulisse, "Benchmarking and Profiling" lecture, CSC2024

# How does it scale?

ALICE  
Event Processing  
Nodes Cluster

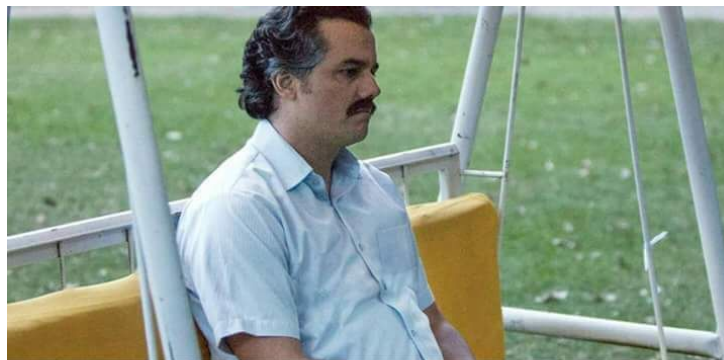
24640 CPU cores  
2800 GPUs



Image credit: Giulio Eulisse, "Benchmarking and Profiling" lecture, CSC2024



# How do you scale?



*Man making Python plots,  
color photograph*



Image from imgflip.com

# Algorithmic complexity

Theoretical metric for estimating resource consumption (in our case runtime - time complexity).

*“The time complexity of an algorithm represents the number of steps it has to take to complete.”*

– Complexity Theory. Brilliant.org. Retrieved 13:58, June 18, 2025, from <https://brilliant.org/wiki/complexity-theory/>

Also useful: Devopedia. 2022. "Algorithmic Complexity." Version 8, February 19. Accessed 2024-06-25. <https://devopedia.org/algorithmic-complexity>

# Big O notation



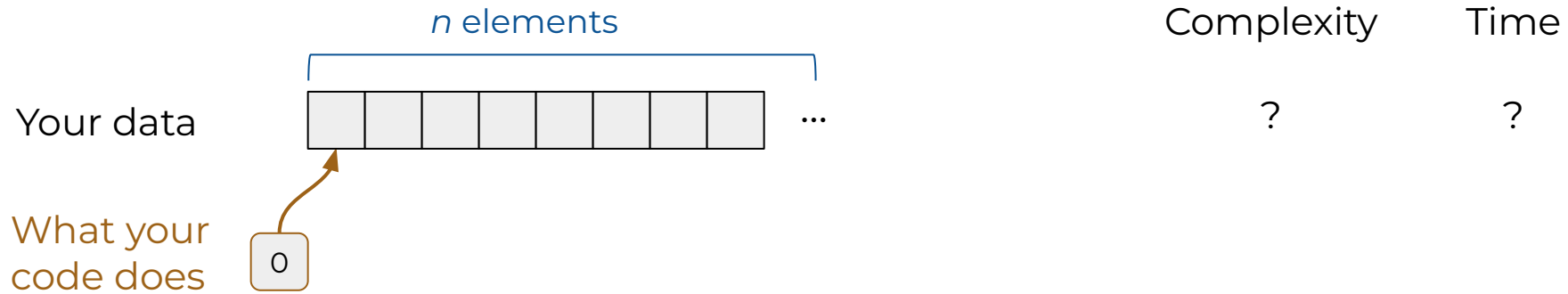
⇒ algorithm is on the order of  $f(n)$  or  $O(f(n))$

Example: if an algorithm's runtime grows **linearly with the input size**, the **algorithm is  $O(n)$**

Strictly:  $O(f(n)) \Leftrightarrow$  “asymptotically bounded by  $f(n)$  up to some constant”  
(see Bachmann-Landau notation)

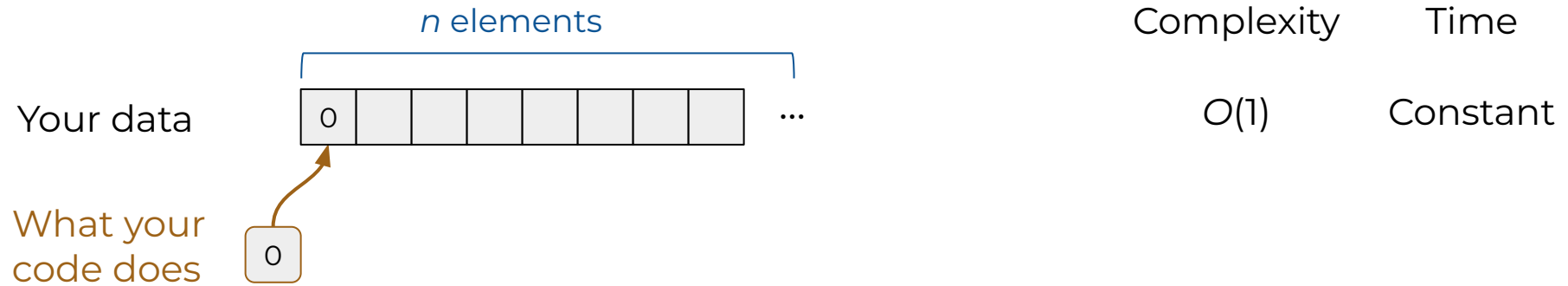
Big O Notation. Brilliant.org. Retrieved 13:54, June 18, 2025, from <https://brilliant.org/wiki/big-o-notation/>

# Common Big O's and examples thereof



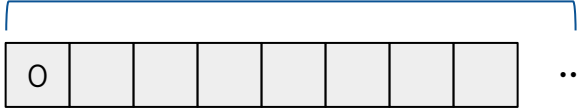
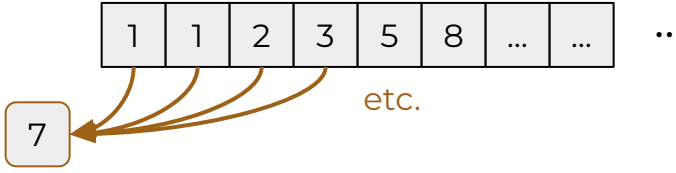
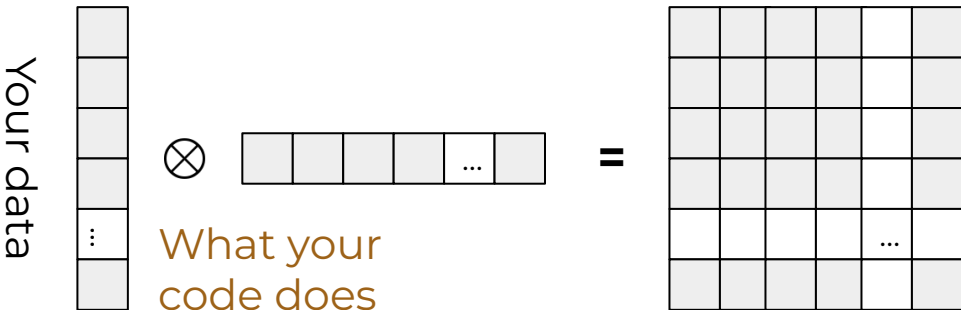


# Common Big O's and examples thereof








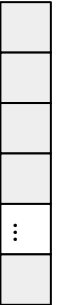
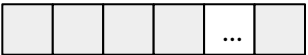
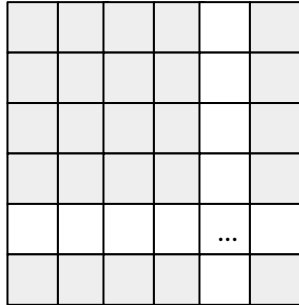
# Common Big O's and examples thereof

	$n$ elements	Complexity	Time
Your data		$O(1)$	Constant
Your data What your code does		?	?
Your data What your code does		?	?


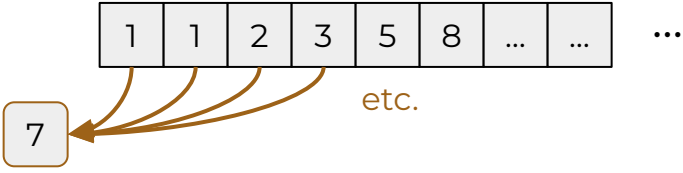
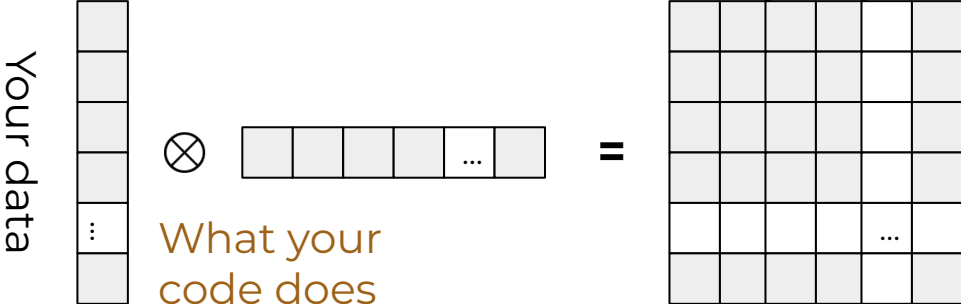
# Common Big O's and examples thereof

	$n$ elements	Complexity	Time
Your data		$O(1)$	Constant

Your data			
What your code does		$O(n)$	Linear

Your data		$\otimes$ 	=		?	?
	What your code does					

# Common Big O's and examples thereof

	$n$ elements	Complexity	Time
Your data		$O(1)$	Constant
Your data What your code does		$O(n)$	Linear
Your data What your code does		$O(n^2)$	Quadratic

# Common Big O's and examples thereof

**$O(n)$**  - linear time (i.e. input size doubles = runtime doubles)

- Radix sort

**$O(n^2)$**  - quadratic time (input size doubles = runtimes quadruples)

- “For loop in a for loop”
- Outer product of two  $n$ -length vectors  $\Rightarrow n$ -by- $n$  matrix
- Selection sort, insertion sort

# Common Big O's and examples thereof

**$O(n)$**  - linear time (i.e. input size doubles = runtime doubles)

- Radix sort

aktshually  $O(k \cdot n)$

**$O(n^2)$**  - quadratic time (input size doubles = runtimes quadruples)

- “For loop in a for loop”
- Outer product of two  $n$ -length vectors  $\Rightarrow n$ -by- $n$  matrix
- Selection sort, insertion sort

# Common Big O's and examples thereof, cont.

**$O(\log_2 n)$**  - logarithmic time

- Binary search of a sorted array

**$O(n \log_2 n)$**  - log-linear or “linearithmic” time  
(close to  $O(n)$  for “reasonably sized”  $n$ )

- Many sorting algorithms, e.g. merge sort, heap sort

**$O(n^3)$**  - cubic time

- “For loop in a for loop in a for loop”
- “Schoolbook” matrix multiplication



# Common Big O's and examples thereof, cont.

**$O(\log_2 n)$**  - logarithmic time

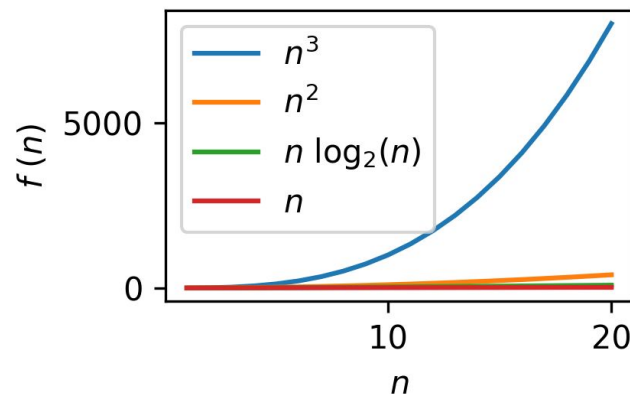
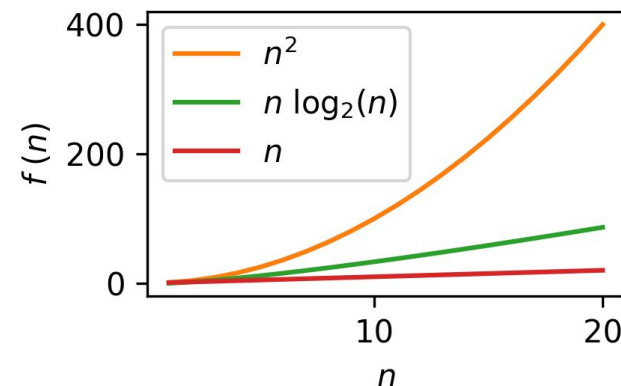
- Binary search of a sorted array

**$O(n \log_2 n)$**  - log-linear or “linearithmic” time  
(close to  $O(n)$  for “reasonably sized”  $n$ )

- Many sorting algorithms, e.g. merge sort, heap sort

**$O(n^3)$**  - cubic time

- “For loop in a for loop in a for loop”
- “Schoolbook” matrix multiplication





# Two really Big O's (to avoid if possible)

**$O(2^n)$**  and  **$O(n!)$**  - exponential time and factorial time

- Brute-force search, combinatorics, NP-hard problems
- Finding prime numbers



# Matrix multiplication

- “Schoolbook” matrix multiplication  $O(n^3)$
- The Strassen algorithm (1969)  $O(n^{2.805})$
- State of the art (2023)  $O(n^{2.373})$

For 4 x 4 matrices:

Strassen algorithm: 49 multiplications

Google Deepmind AlphaEvolve algorithm: 48 multiplications

AlphaEvolve: A Gemini-powered coding agent for designing advanced algorithms

<https://deepmind.google/discover/blog/alphaevolve-a-gemini-powered-coding-agent-for-designing-advanced-algorithms/>

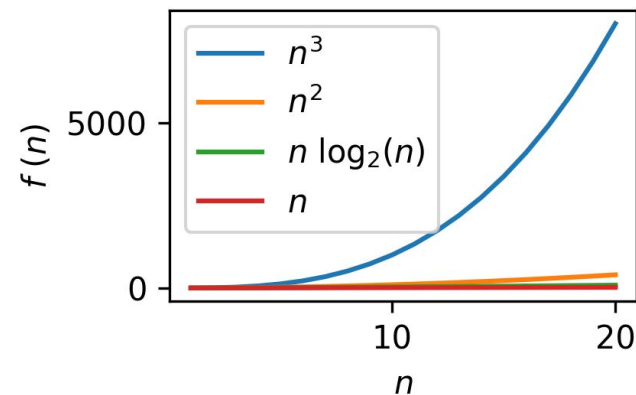
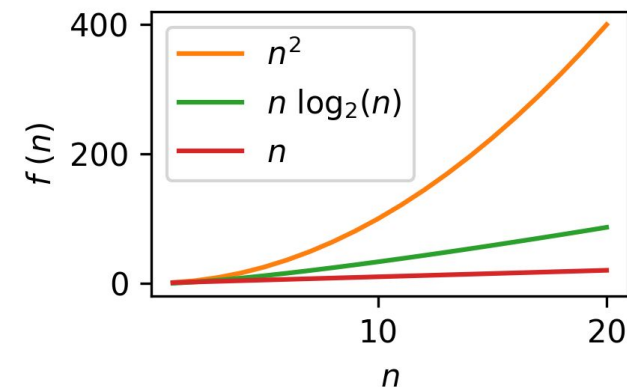


# Algorithmic complexity - conclusion

```
int binary_search(const std::vector<int>& v, int target) {  
    int left = 0, right = v.size() - 1;  
  
    while (left <= right) {  
        int mid = left + (right - left) / 2;  
        if (v[mid] == target)  
            return mid;  
        else if (v[mid] < target)  
            left = mid + 1;  
        else  
            right = mid - 1;  
    }  
  
    return -1; // not found  
}
```

# Common code patterns

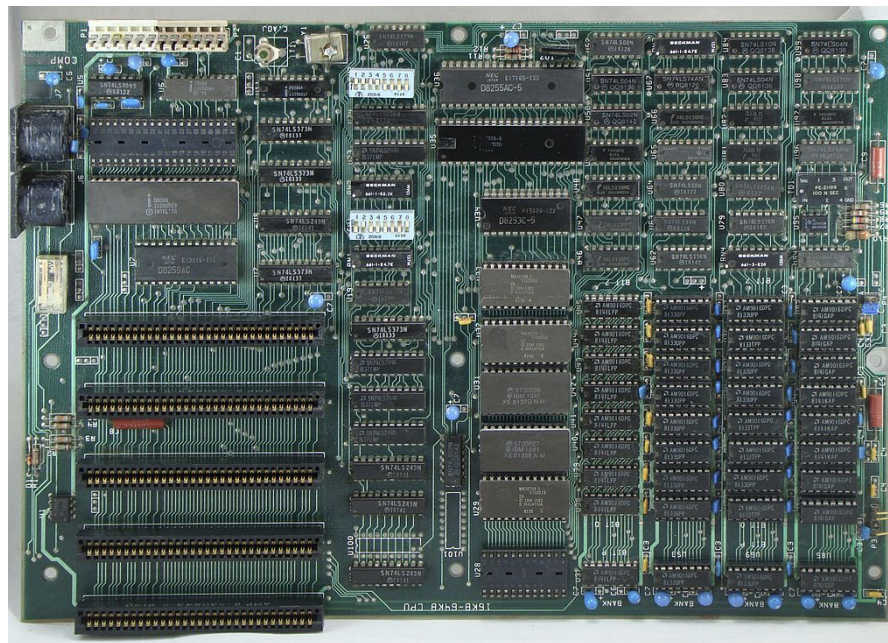
- For loop in a for loop
  - Solve differently?
  - Combine loops?
- Searching through data
  - Consider sorting first?
  - Use look-up table?
- Recomputing values
  - Compute once, keep in a table?



Pit stop



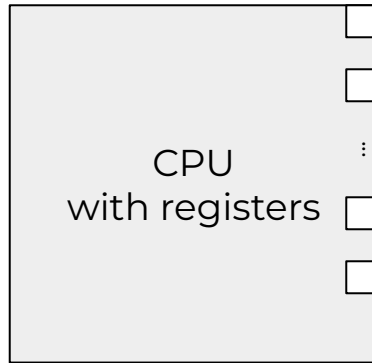
# Memory access patterns



An IBM Personal Computer Model 5150 motherboard, 1981 - the first "PC". Image credit: user GermanX on Wikimedia Commons, under license [Attribution-ShareAlike 2.5 Generic](https://creativecommons.org/licenses/by-sa/2.5/)

# CPU-to-memory layout

CPU registers



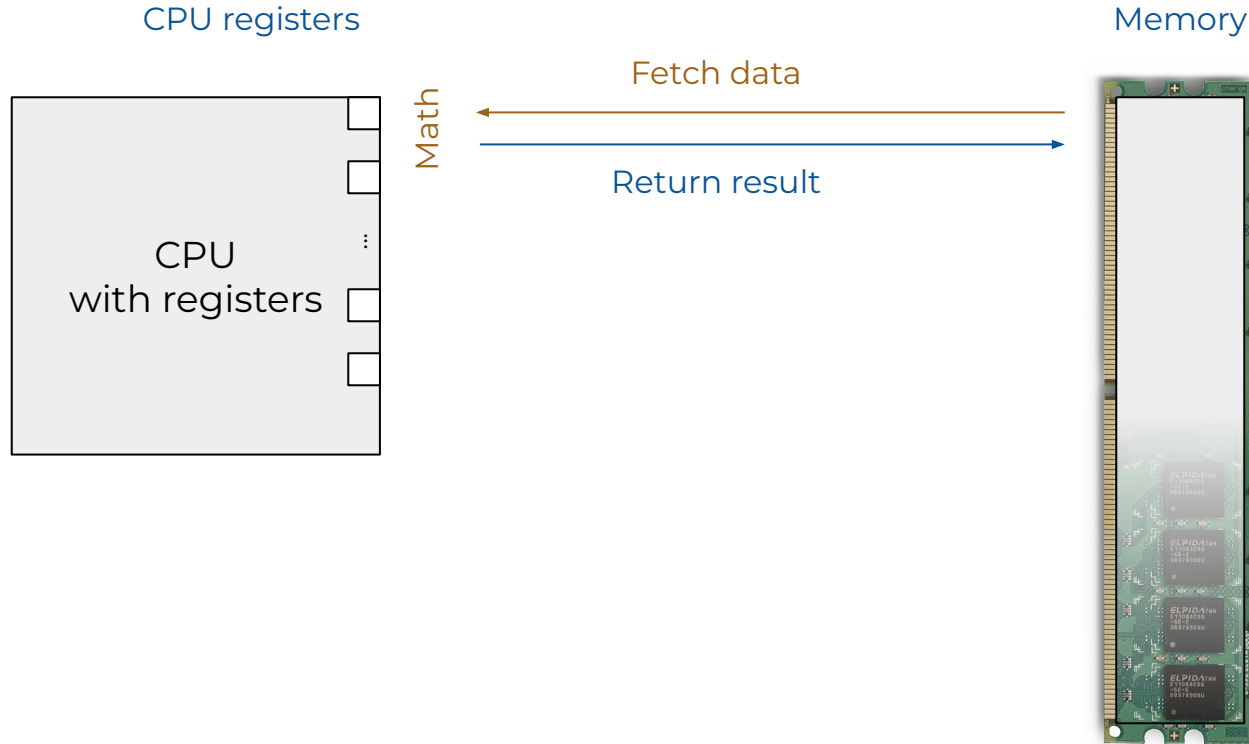
Memory



RAM image credit: user an-d on Wikipedia,  
under [Attribution-ShareAlike 3.0 Unported](#)

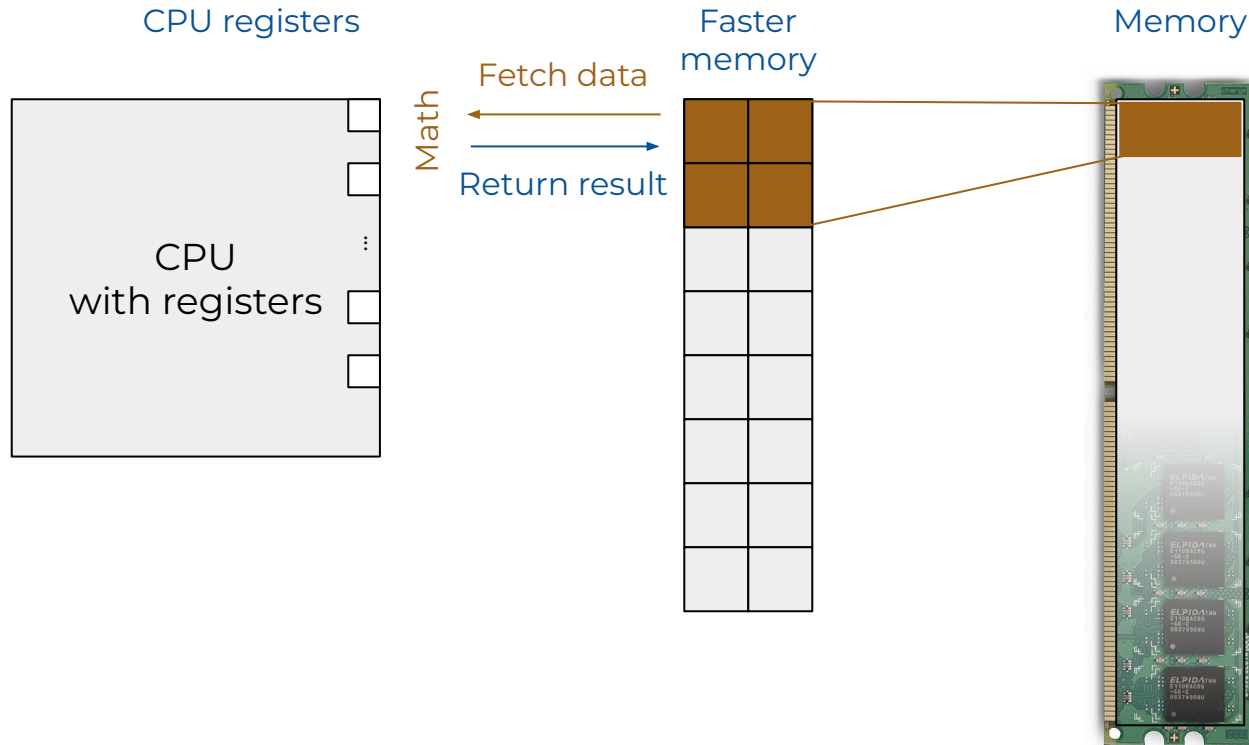


# CPU-to-memory layout



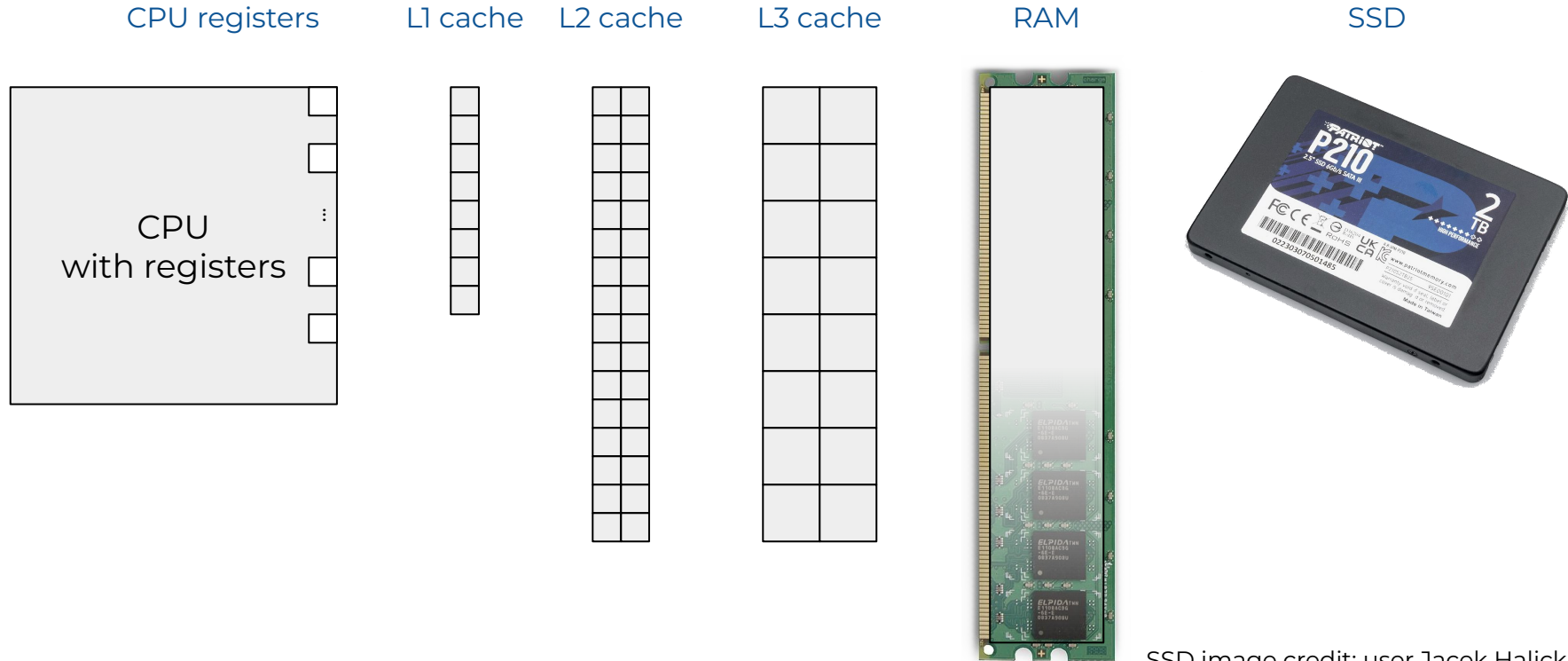
RAM image credit: user an-d on Wikipedia,  
under [Attribution-ShareAlike 3.0 Unported](#)

# CPU-to-memory layout



RAM image credit: user an-d on Wikipedia,  
under [Attribution-ShareAlike 3.0 Unported](#)

# CPU-to-memory layout

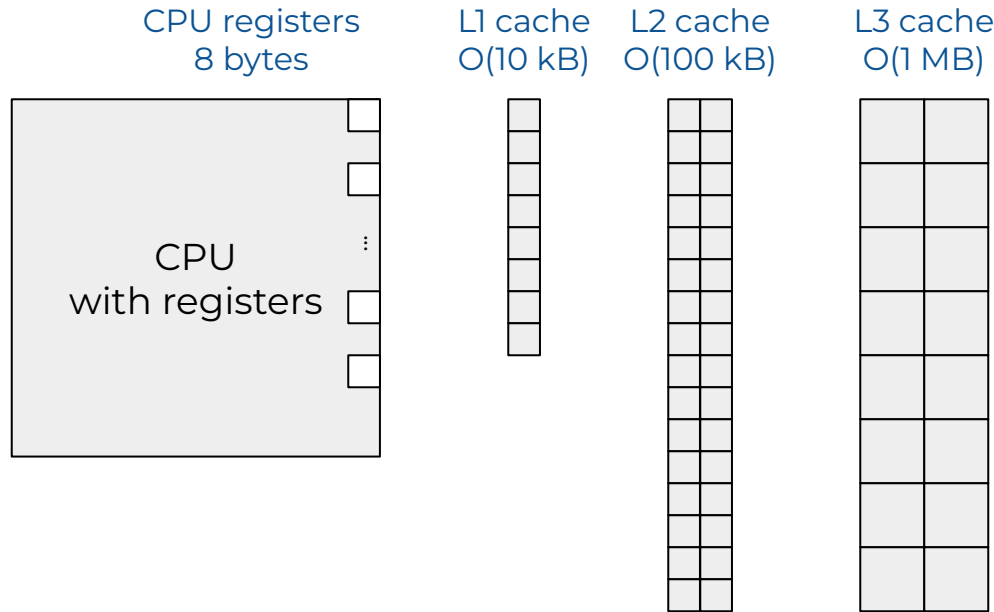


RAM image credit: user an-d on Wikipedia,  
under [Attribution-ShareAlike 3.0 Unported](#)

SSD image credit: user Jacek Halicki  
on Wikipedia, under  
[Attribution-ShareAlike 4.0](#)

# CPU-to-memory layout

Typical sizes of different memory hardware



RAM  
O(10 GB)



SSD  
O(1 TB)



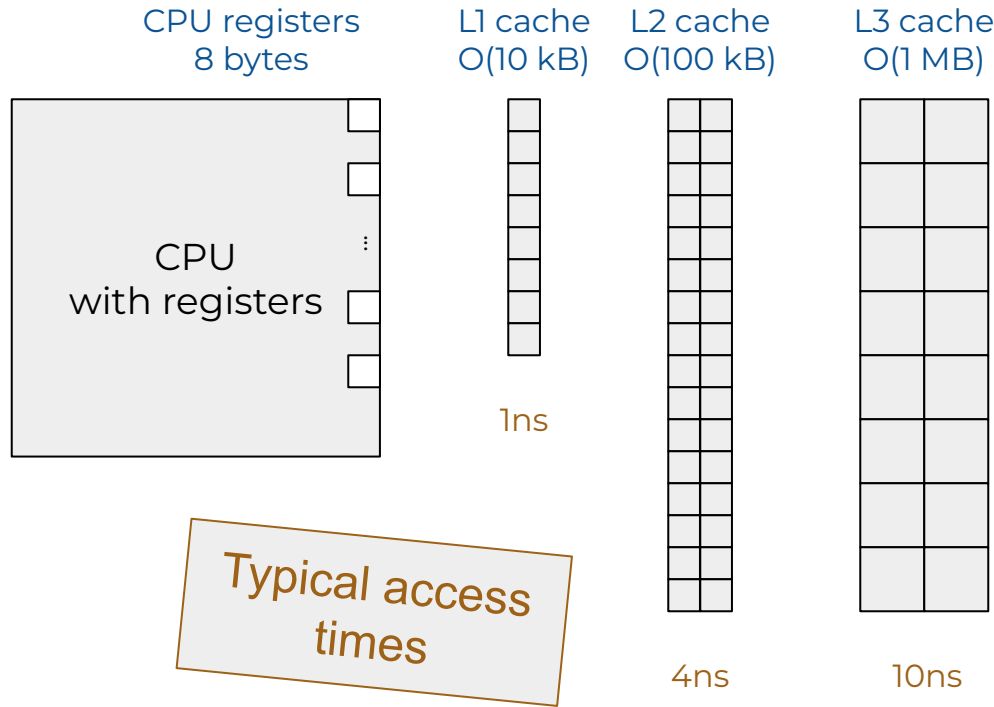
RAM image credit: user an-d on Wikipedia,  
under [Attribution-ShareAlike 3.0 Unported](#)

SSD image credit: user Jacek Halicki  
on Wikipedia, under  
[Attribution-ShareAlike 4.0](#)



# CPU-to-memory layout

Typical sizes of different memory hardware

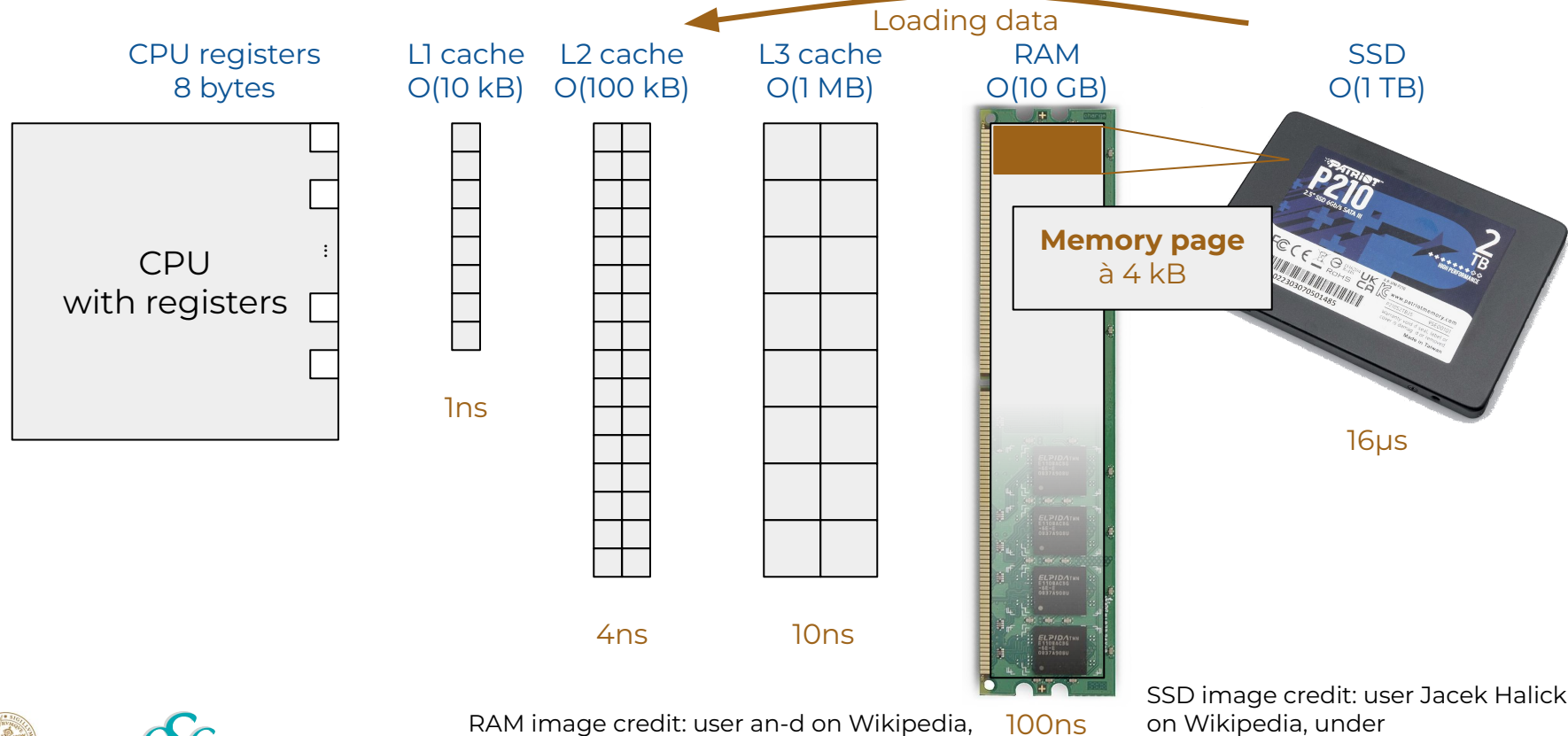


Typical access times

RAM image credit: user an-d on Wikipedia, under [Attribution-ShareAlike 3.0 Unported](#)

SSD image credit: user Jacek Halicki on Wikipedia, under [Attribution-ShareAlike 4.0](#)

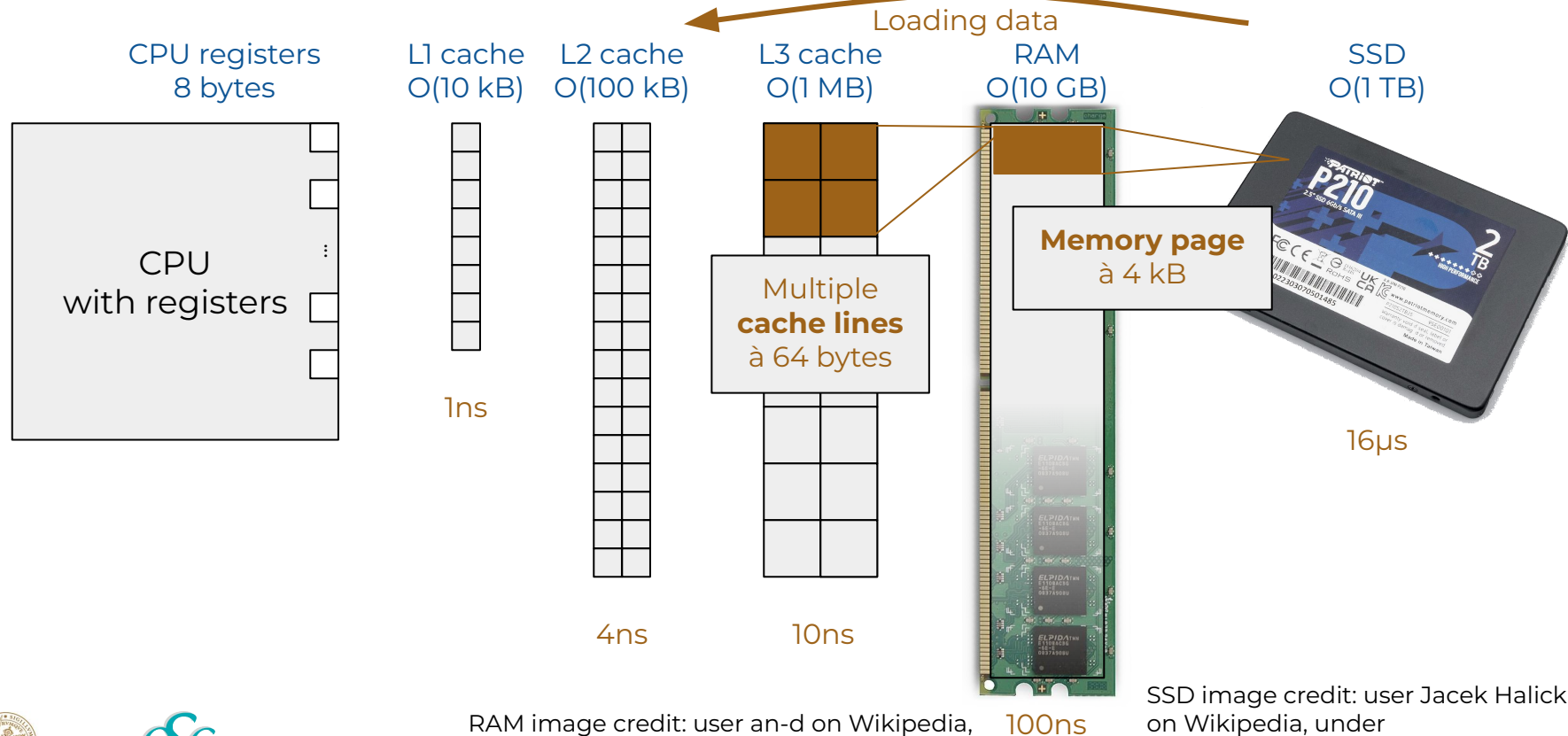
# Block-based memory access, the “how”



RAM image credit: user an-d on Wikipedia,  
under [Attribution-ShareAlike 3.0 Unported](#)

SSD image credit: user Jacek Halicki  
on Wikipedia, under  
[Attribution-ShareAlike 4.0](#)

# Block-based memory access, the “how”

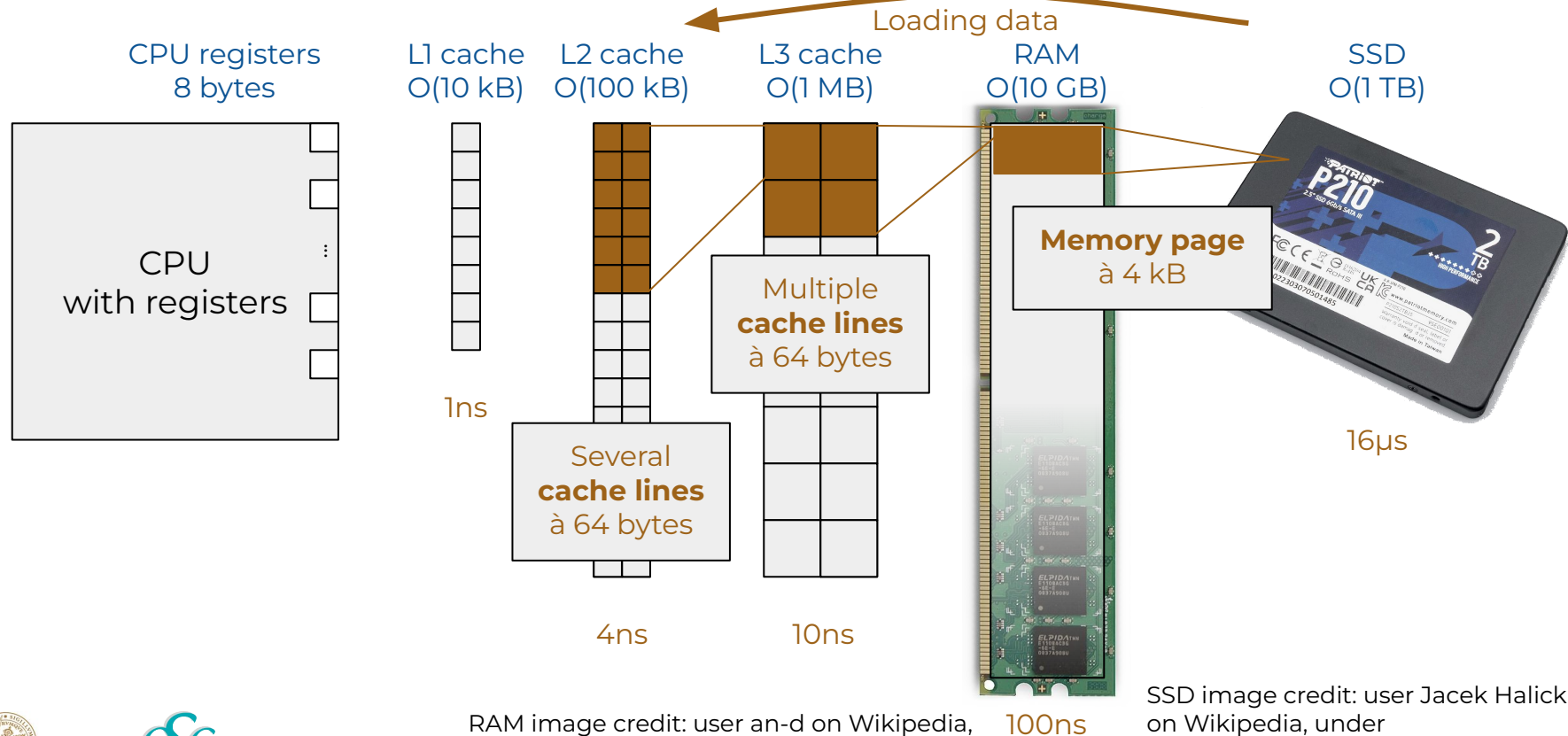


RAM image credit: user an-d on Wikipedia,  
under [Attribution-ShareAlike 3.0 Unported](#)

SSD image credit: user Jacek Halicki  
on Wikipedia, under  
[Attribution-ShareAlike 4.0](#)



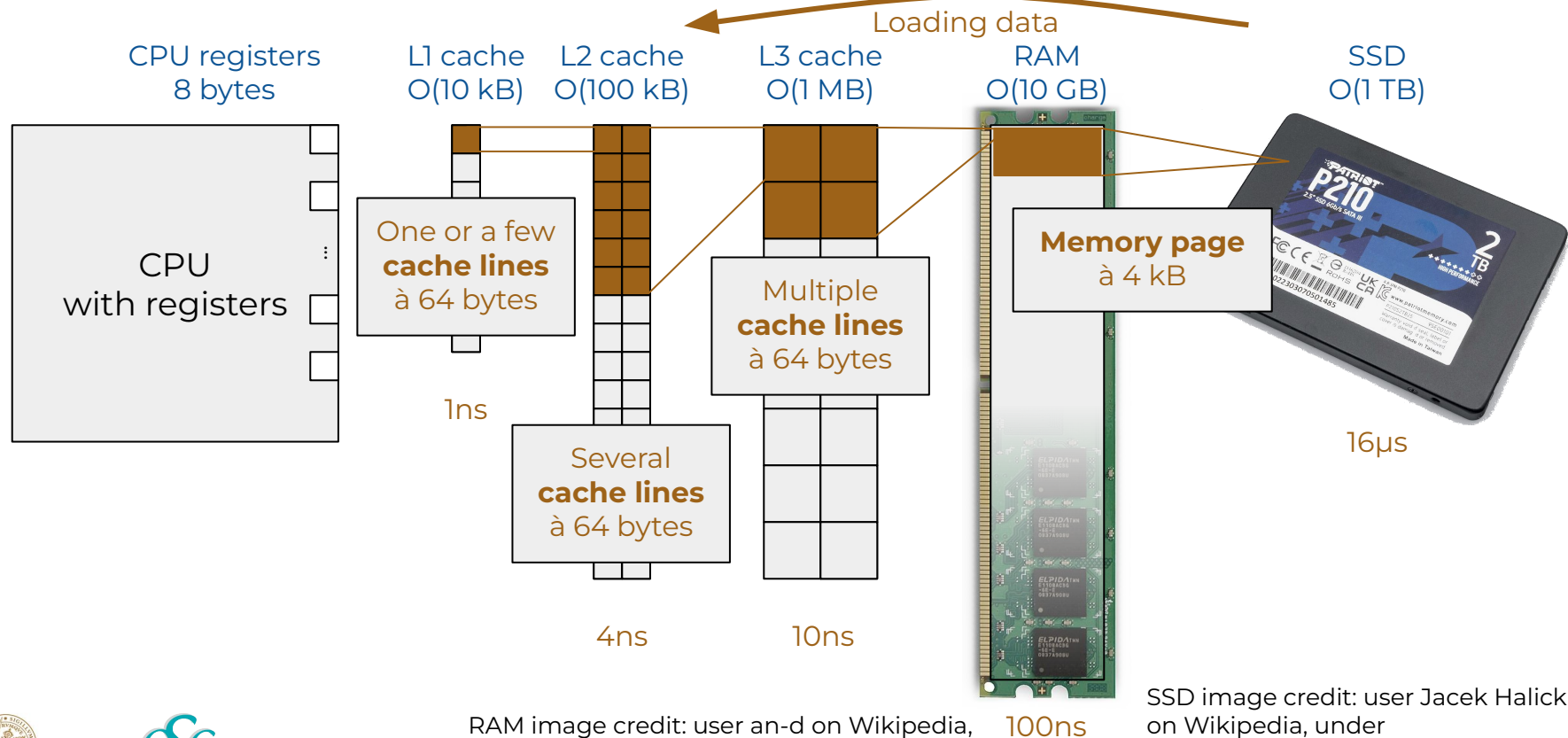
# Block-based memory access, the “how”



RAM image credit: user an-d on Wikipedia,  
under [Attribution-ShareAlike 3.0 Unported](#)

SSD image credit: user Jacek Halicki  
on Wikipedia, under  
[Attribution-ShareAlike 4.0](#)

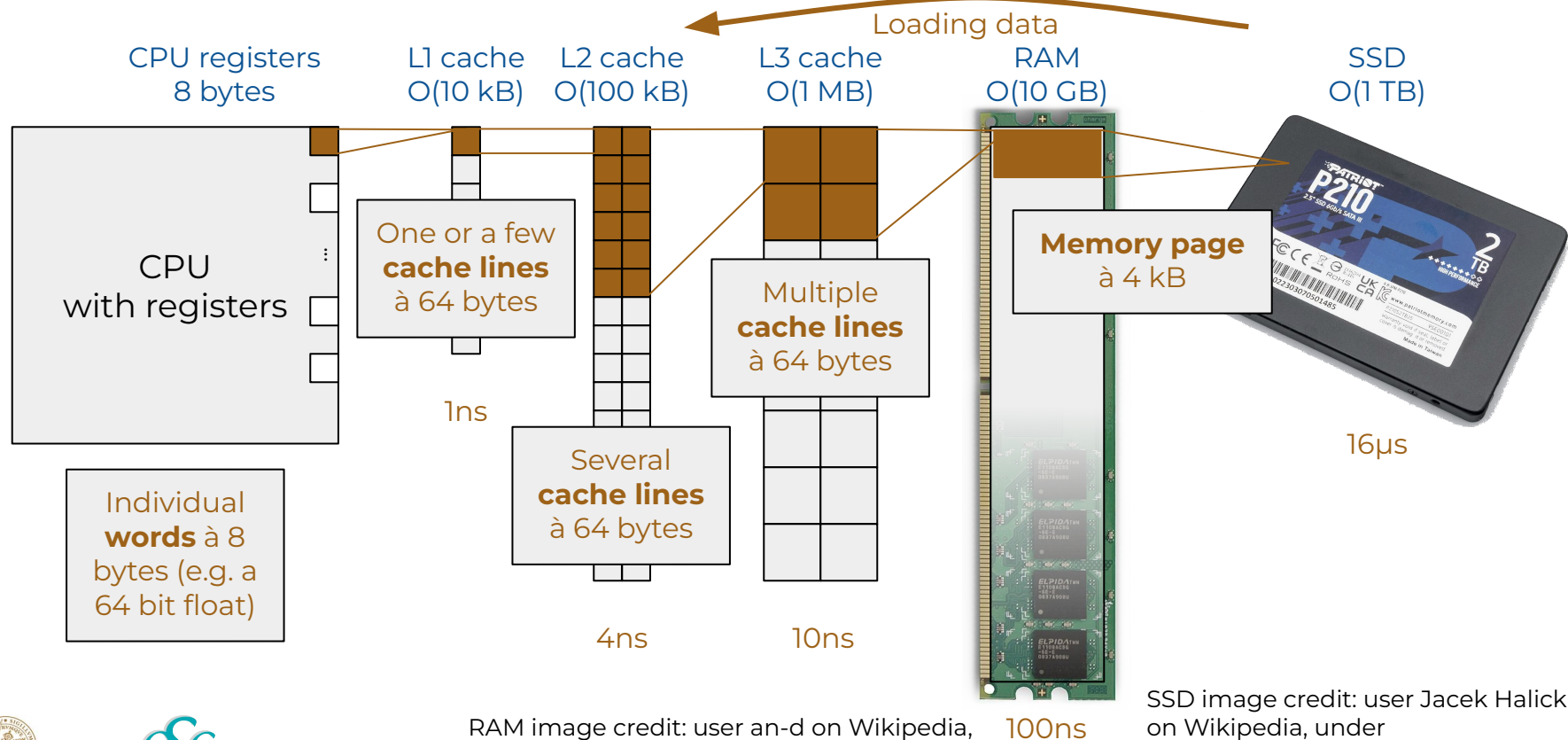
# Block-based memory access, the “how”



RAM image credit: user an-d on Wikipedia,  
under [Attribution-ShareAlike 3.0 Unported](#)

SSD image credit: user Jacek Halicki  
on Wikipedia, under  
[Attribution-ShareAlike 4.0](#)

# Block-based memory access, the “how”



RAM image credit: user an-d on Wikipedia,  
under [Attribution-ShareAlike 3.0 Unported](#)

SSD image credit: user Jacek Halicki  
on Wikipedia, under  
[Attribution-ShareAlike 4.0](#)

# Block-based memory access, the “why”

“**Principle of locality**” or “**data locality**” - data access often happens on many elements close to each other.

If you read  
this...



You will  
likely then  
read those

# Block-based memory access, the “why”

“**Principle of locality**” or “**data locality**” - data access often happens on many elements close to each other.

If you read  
this...



You will  
likely then  
read those

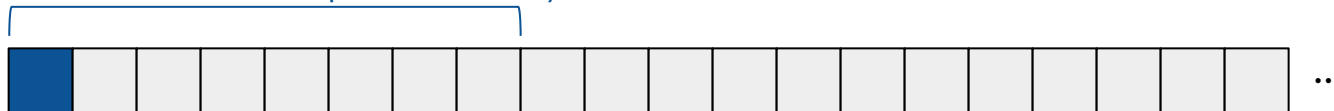
- **space locality**
- **time locality**

**Pre-fetching:** loading data that is likely to be needed soon

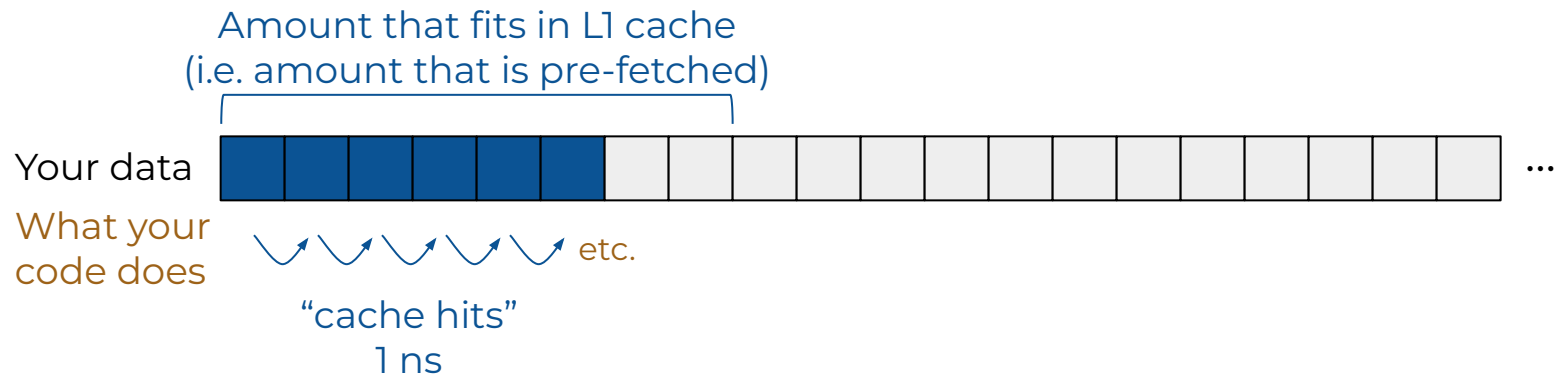
# Memory access patterns

Amount that fits in L1 cache  
(i.e. amount that is pre-fetched)

Your data

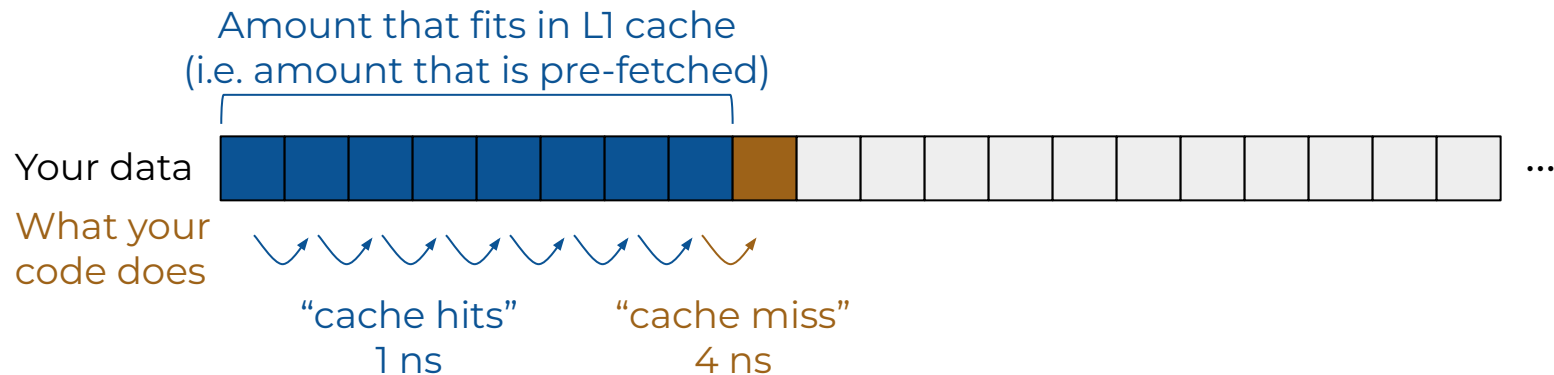


# Memory access patterns



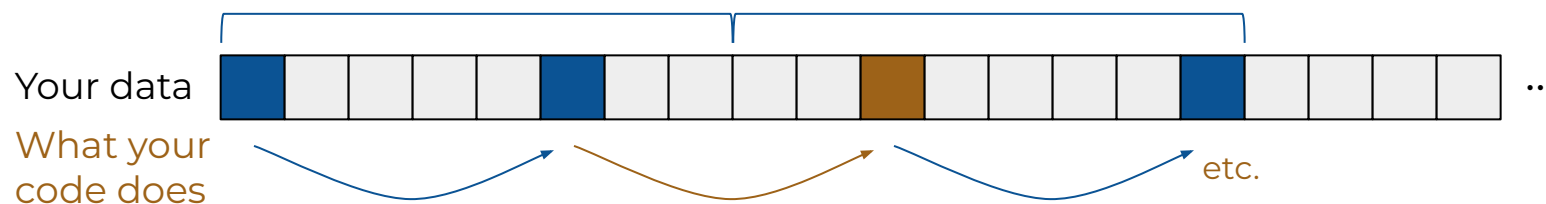


# Memory access patterns

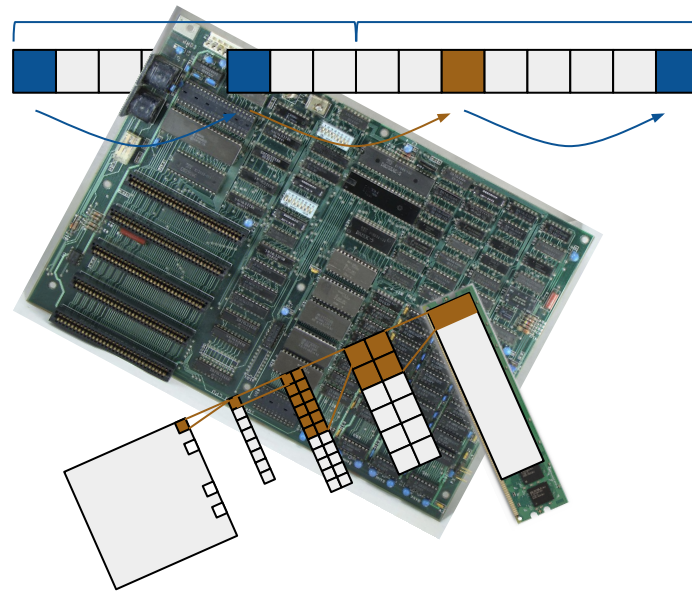


# Memory access patterns

Amount that fits in L1 cache  
(i.e. amount that is pre-fetched)



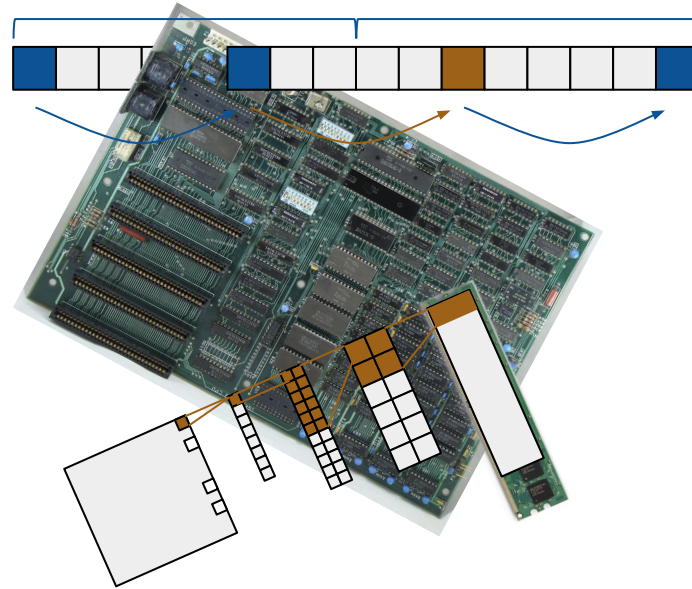
# Memory access patterns - conclusions



## Pit stop 2



# Benchmarking



# Benchmarking, pitfalls of

...but measuring is hard.

- Compiler optimizations
- Programs running in parallel  $\Rightarrow$  variance
- CPUs boost clock frequencies
- ...

Takeaway: healthy scepticism and reasonable expectations



# Linux **time** command

```
int main() {  
    int temp;  
    for (int i = 0; i < 1000000000; i++) {  
        temp = i;  
    }  
    return temp;  
}
```

```
$ g++ -O3 time_example.cpp -o time_example  
$ time ./time_example
```

```
real    0m0.009s  
user    0m0.002s  
sys     0m0.002s
```

- **real:** total time until your program finished
- **user:** time spent executing your program
- **sys:** time the system spent on behalf of your program

Note: blunt tool



# Linux **time** command

```
int main() {
    int temp;
    for (int i = 0; i < 1000000000; i++) {
        temp = i;
    }
    return temp;
}
```



[godbolt.org](https://godbolt.org) Compiler Explorer:

```
main:
    mov     eax, 999999999
    ret
```

```
$ g++ -O3 time_example.cpp -o time_example
$ time ./time_example
```

```
real    0m0.009s
user    0m0.002s
sys     0m0.002s
```

- **real:** total time until your program finished
- **user:** time spent executing your program
- **sys:** time the system spent on behalf of your program

Note: blunt tool





# Compiler pitfalls

```
int main() {  
    int size = 1000;  
    int index = 0;  
    for (int i = 0; i < 1000000000; i++) {  
        index = i % size;  
        // ... vector operations  
    }  
}
```

standard string-to-int conversion

```
int main() {  
    int size = std::stoi(string_from_user);  
    int index = 0;  
    for (int i = 0; i < 1000000000; i++) {  
        index = i % size;  
        // ... vector operations  
    }  
}
```

# Compiler pitfalls

```
int main() {  
    int size = 1000;  
    int index = 0;  
    for (int i = 0; i < 1000000000; i++) {  
        index = i % size;  
        // ... vector operations  
    }  
}
```

Array size: 1000  
Steps taken: 1000000000  
Time per step: 1.06009 ns

standard string-to-int conversion

```
int main() {  
    int size = std::stoi(string_from_user);  
    int index = 0;  
    for (int i = 0; i < 1000000000; i++) {  
        index = i % size;  
        // ... vector operations  
    }  
}
```

Array size: 1000  
Steps taken: 1000000000  
Time per step: 2.02596 ns

⇒ loop 2x slower

# Compiler pitfalls

```
int main() {
    int size = 1000;
    int index = 0;
    for (int i = 0; i < 1000000000; i++) {
        index = i % size;
        // ... vector operations
    }
}
```

```
movabs rsi, 2361183241434822607
```

```
.L4:
mov rdx, rcx
shr rdx
mov rax, rdx
mul rsi
mov rax, rcx
shr rdx, 4
imul rdx, rdx, 250
sub rax, rdx
movsx rax, DWORD PTR [rbp+0+rax*4]
add ebx, DWORD PTR [rbp+0+rax*4]
```

standard string-to-int  
conversion

```
int main() {
    int size = std::stoi(string_from_user);
    int index = 0;
    for (int i = 0; i < 1000000000; i++) {
        index = i % size;
        // ... vector operations
    }
}
```

```
mov rax, rcx
cqo
idiv rbp
movsx rax, DWORD PTR [rbx+rdx*4]
add esi, DWORD PTR [rbx+rax*4]
mov r12d, esi
```

⇒ loop 2x slower

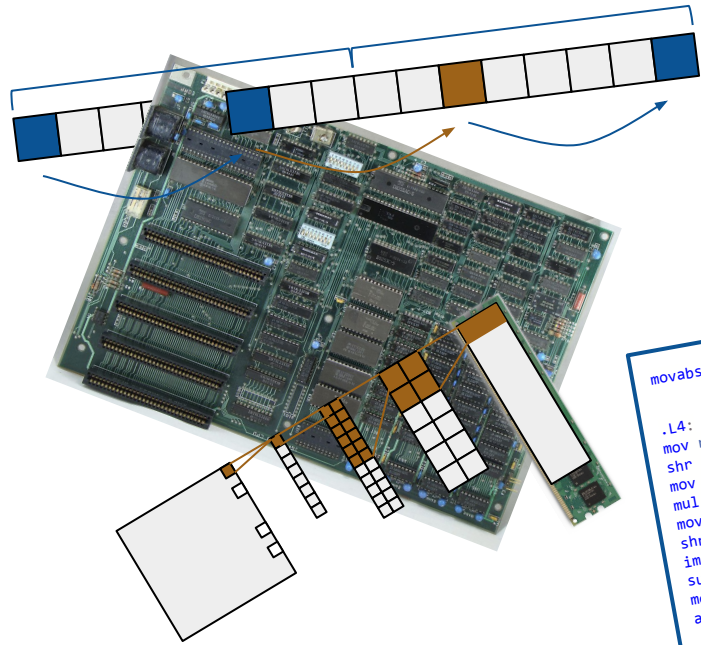
# Benchmarking tools

Some more sophisticated benchmarking tools:

- C/C++
  - Catch2
  - Google Benchmark
- Rust
  - bench
- Python
  - pytest-benchmark
  - timeit
- General purpose
  - hyperfine
  - **perf** (Linux)



# Benchmarking - conclusions



- Compiler optimizations
- Programs running in parallel  $\Rightarrow$  variance
- CPUs boost clock frequencies
- ...

```

movabs rsi, 236118324143
.L4:
mov rdx, rcx
shr rdx
mov rax, rdx
mul rsi
mov rax, rcx
shr rdx, 4
imul rdx, rdx, 250
sub rax, rdx
movsx rax, DWORD PTR [rbp+0+rax*4]
add ebx, DWORD PTR [rbp+0+rax*4]

mov rax, rcx
cqo
idiv rbp
movsx rax, DWORD PTR [rbx+rdx*4]
add esi, DWORD PTR [rbx+rax*4]
mov r12d, esi
  
```

**Healthy scepticism and reasonable expectations**

## Conclusions - conclusions

# Tools and Techniques, Lecture 3

# Performance

- Algorithmic complexity
- Memory access patterns
- Benchmarking

<https://github.com/Stoneandbeach/CSC2025>



# Backup



# Catch2

```
#include <catch2/catch_test_macros.hpp>
#include <catch2/benchmark/catch_benchmark.hpp>

int arithmetic_sum(int upper) {
    volatile int sum = 0;
    for (int i = 1; i <= upper; i++) {
        sum += i;
    }
    return sum;
}

TEST_CASE("Benchmarking Basics") {
    BENCHMARK("Arithmetic sum 1 to 100") {
        return arithmetic_sum(100);
    };
}
```



# Catch2

```
#include <catch2/catch_test_macros.hpp>
#include <catch2/benchmark/catch_benchmark.hpp>

int arithmetic_sum(int upper) {
    volatile int sum = 0;
    for (int i = 1; i <= upper; i++) {
        sum += i;
    }
    return sum;
}

TEST_CASE("Benchmarking Basics") {
    BENCHMARK("Arithmetic sum 1 to 100") {
        return arithmetic_sum(100);
    };
}
```

including Catch2  
library

function to  
benchmark

defining a set of tests  
or benchmarks

configuring a  
benchmark

# Catch2

```
#include <catch2/catch_test_macros.hpp>
#include <catch2/benchmark/catch_benchmark.hpp>

int arithmetic_sum(int upper) {
    volatile int sum = 0;
    for (int i = 1; i <= upper; i++) {
        sum += i;
    }
    return sum;
}

TEST_CASE("Benchmarking Basics") {
    BENCHMARK("Arithmetic sum 1 to 100") {
        return arithmetic_sum(100);
    };
}
```

including Catch2  
library

function to  
benchmark

defining a set of tests  
or benchmarks

configuring a  
benchmark

# Catch2 output

```
-----
Benchmarking Basics
-----
```

```
/eos/user/k/kaastran/schools/CSC2025/prep/exercises/src/ex0.0_benchmarking_basics.cpp:15
.....
```

benchmark name	samples	iterations	est run time
	mean	low mean	high mean
	std dev	low std dev	high std dev
Arithmetic sum 1 to 100	100	518	3.4706 ms
	38.5609 ns	36.9933 ns	40.8075 ns
	9.47688 ns	7.24546 ns	12.492 ns

```
=====
test cases: 1 | 1 passed
assertions: - none -
```