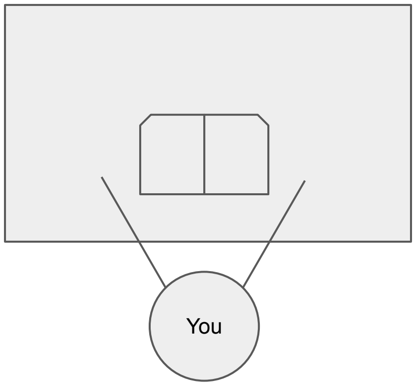
Optimizations for a single thread

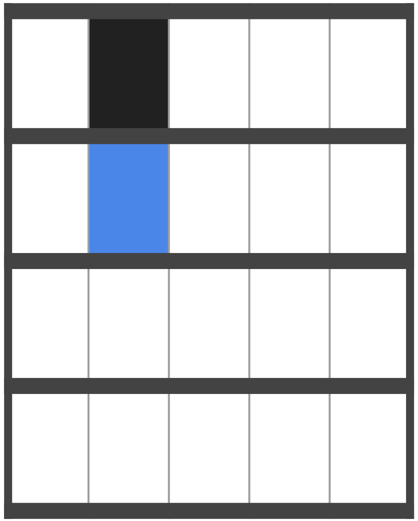
Only after you've benchmarked

Imagine being at a library

You want to read about a certain topic.

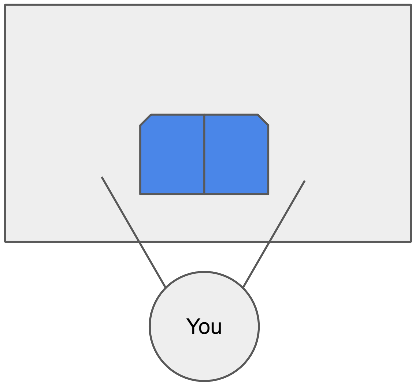
You are at a table, reading books.

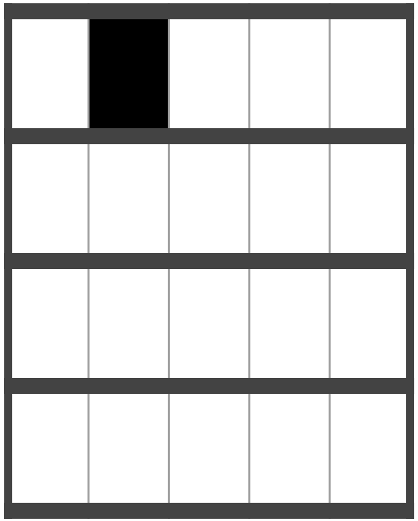




Imagine being at a library

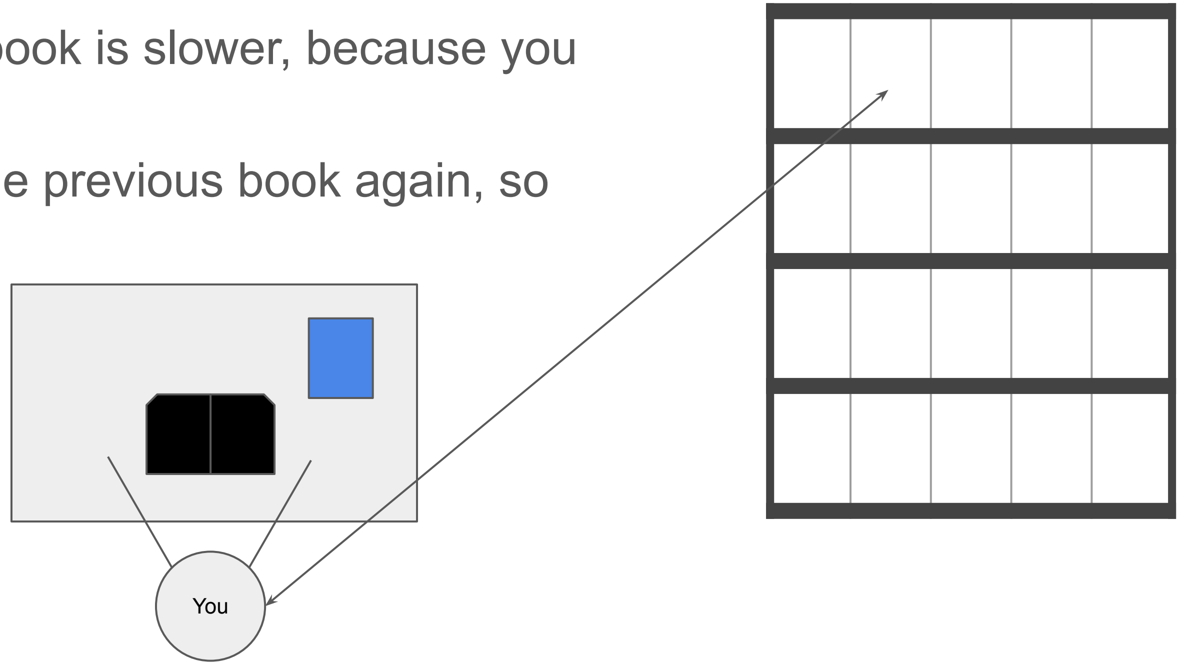
- You take a book and start to read it
- When you read a word, it is fast to read the next one, if it is on the same page





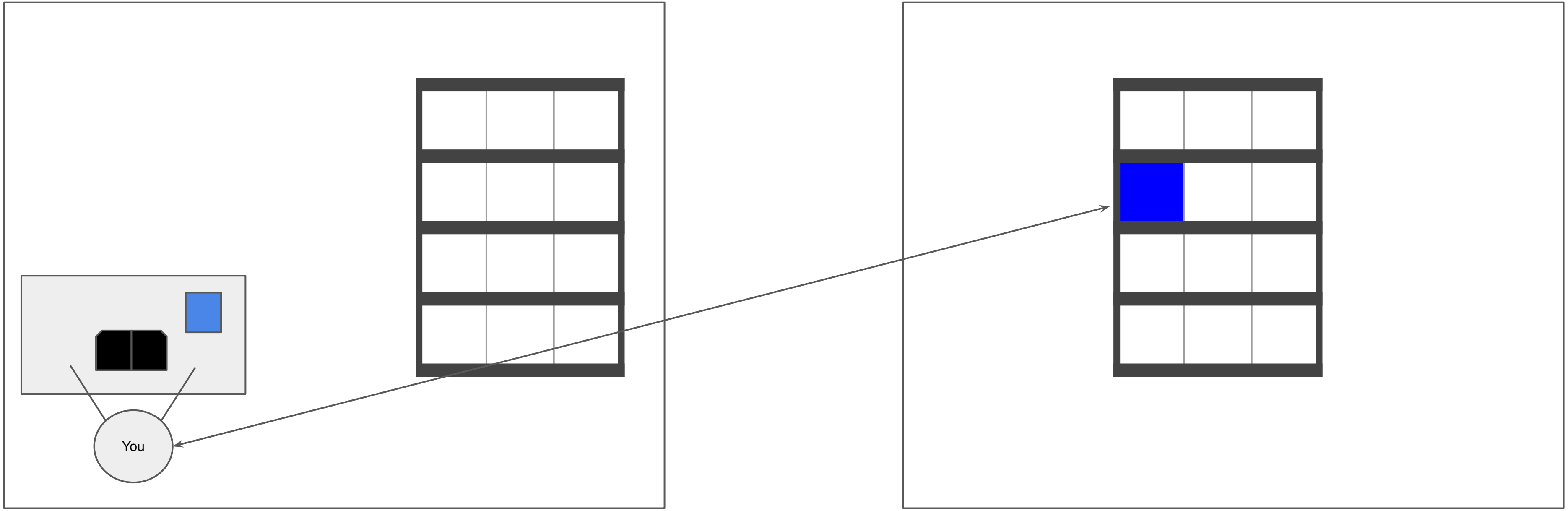
Imagine being at a library

- Starting to read another book is slower, because you have to fetch it first
- You might want to read the previous book again, so keep it close to you

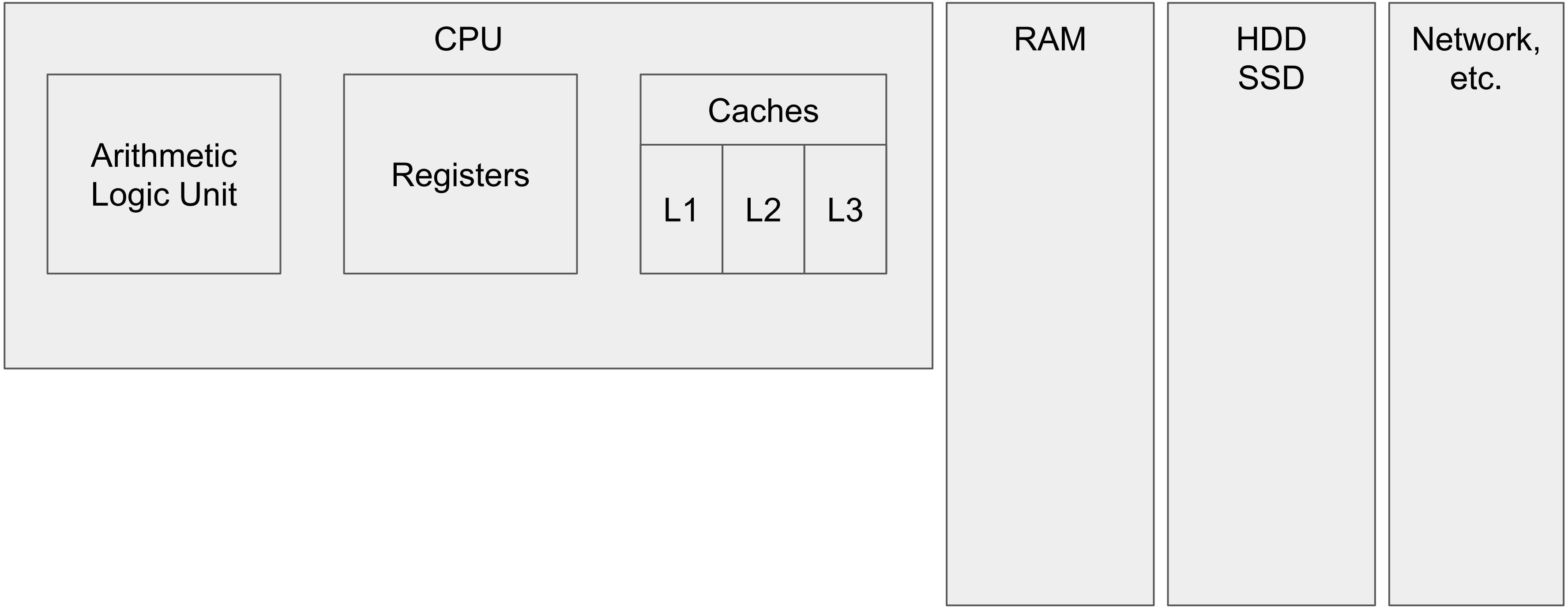


Imagine being at a library

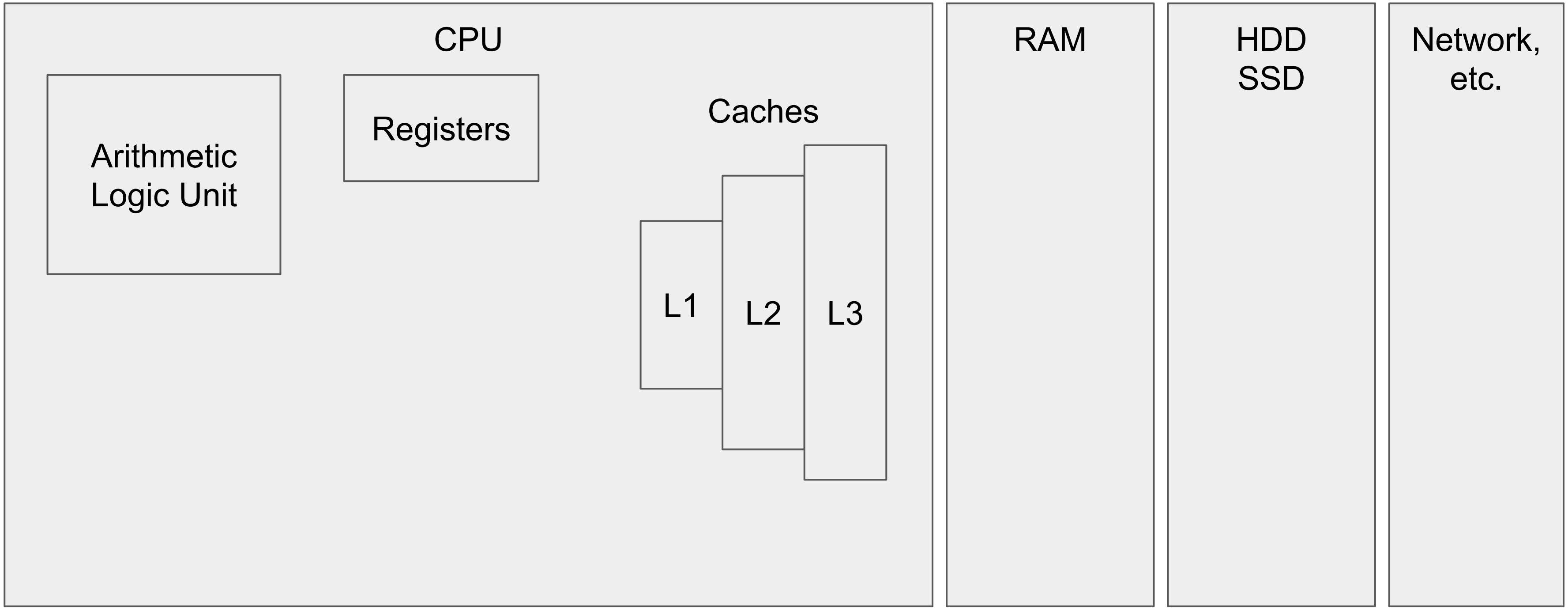
- Fetching a book from another library will take days or weeks



Very simple model of a computer



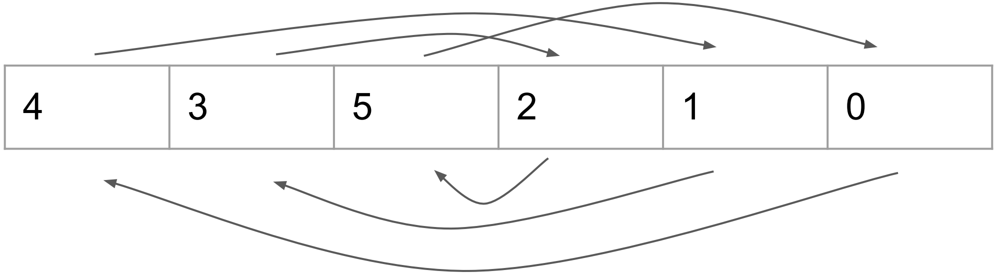
Very simple model of a computer



Effect of cache misses

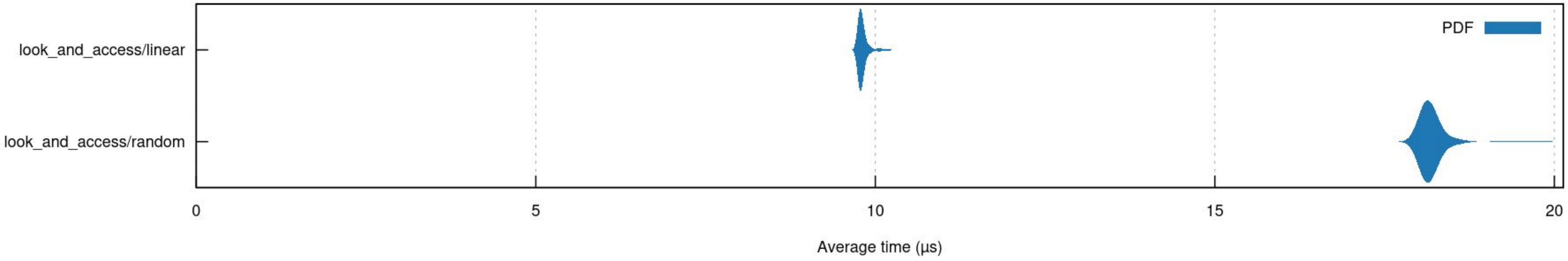
```
pub fn look_and_access(vector: &[usize]) {
    let initial_element = vector[0];
    let mut element = initial_element;

    loop {
        element = vector[element];
        if element == initial_element {
            break;
        }
    }
}
```

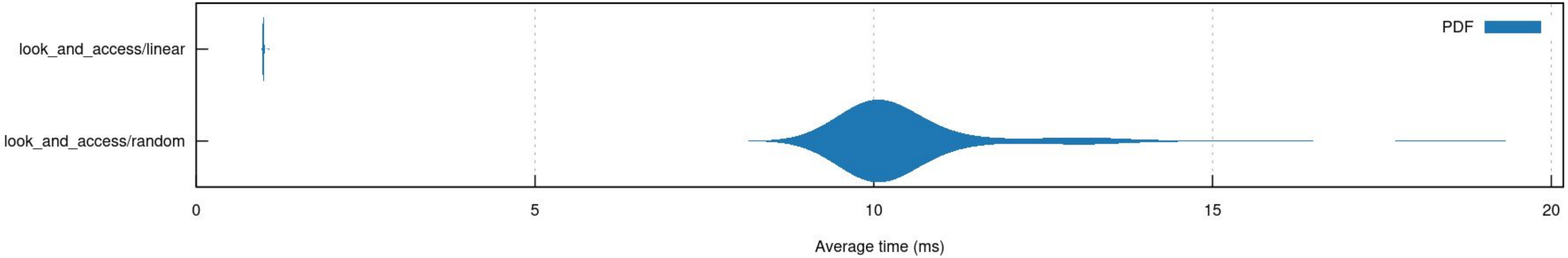


Effect of cache misses

10 000 elements



1 000 000 elements



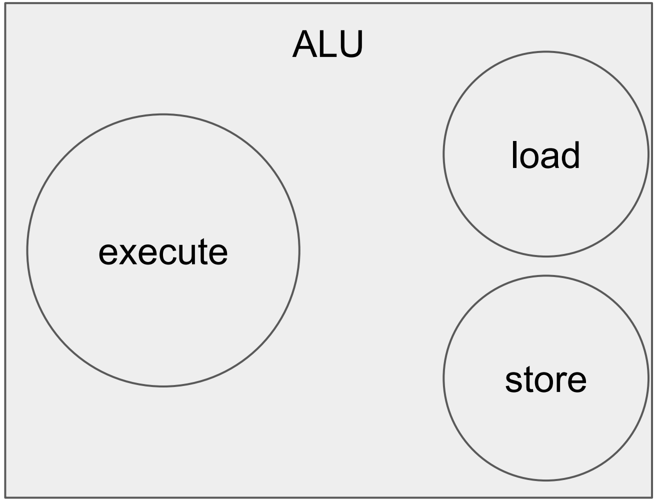
Take home message

- keep the data is local as possible
  - avoid indirections
  - if you store a matrix, think about the order (column vs lines)
  - replace hash tables by vectors for small batches of data
  - …

A CPU is actually a pipeline

An ALU consists of multiple parts.

One instruction will “visit” these parts one by one.

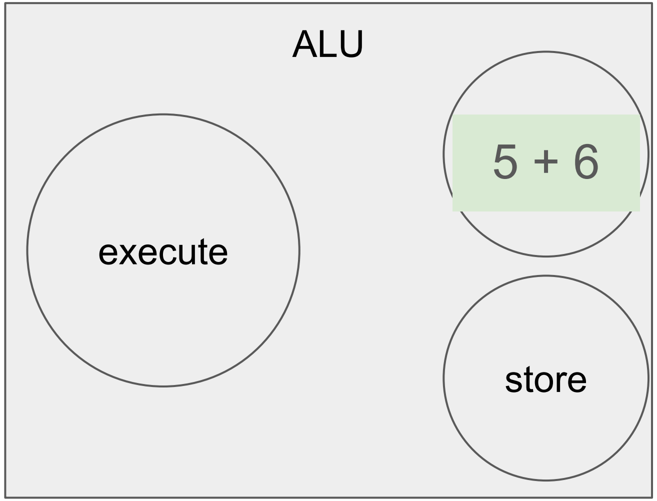


c = a + b

A CPU is actually a pipeline

An ALU consists of multiple parts.

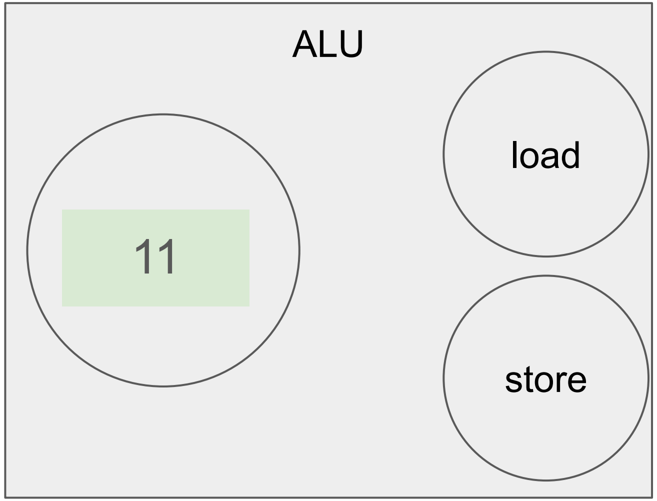
One instruction will “visit” these parts one by one.



A CPU is actually a pipeline

An ALU consists of multiple parts.

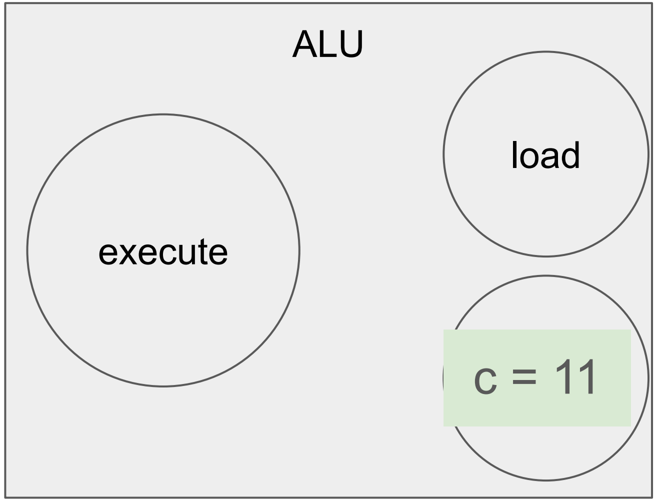
One instruction will “visit” these parts one by one.



A CPU is actually a pipeline

An ALU consists of multiple parts.

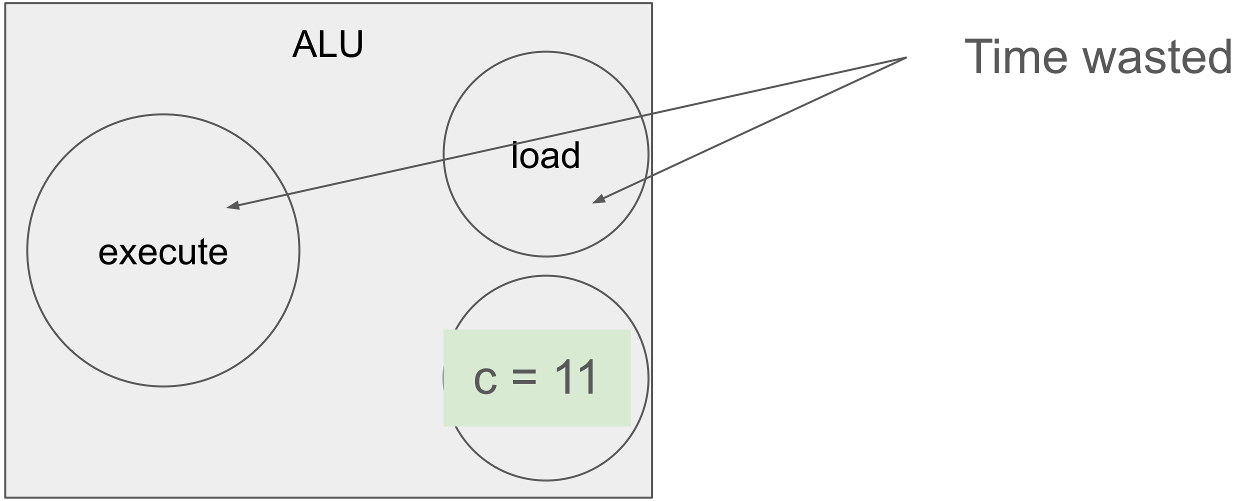
One instruction will “visit” these parts one by one.



A CPU is actually a pipeline

An ALU consists of multiple parts.

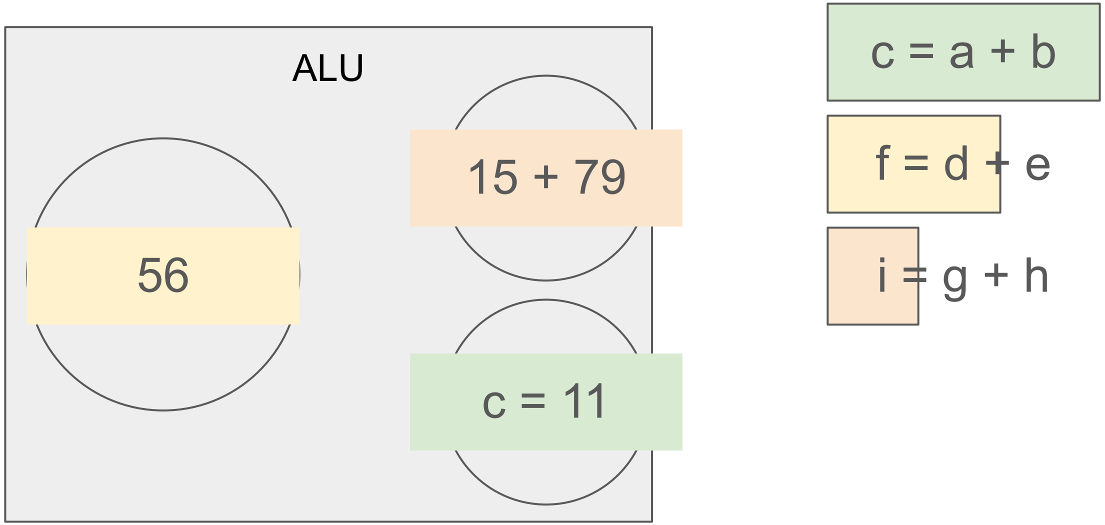
One instruction will “visit” these parts one by one.



A CPU is actually a pipeline

An ALU consists of multiple parts.

One instruction will “visit” these parts one by one.



A CPU is actually a pipeline - what happens on an if ?

```
c = a + b

if c < 5 {

    i = g + h

}
```

A CPU is actually a pipeline - what happens on an if ?

```
c = a + b
if c < 5 {
    i = g + h
}
```

Executed without checking the condition

A CPU is actually a pipeline - what happens on an if ?

```
c = a + b
if c < 5 {
    i = g + h
}
```

Executed without checking the condition

- if condition is true, you win
- else, you rollback

A CPU is actually a pipeline - what happens on an if ?

```
c = a + b
if c < 5 {
    i = g + h

}
```

Executed without checking the condition

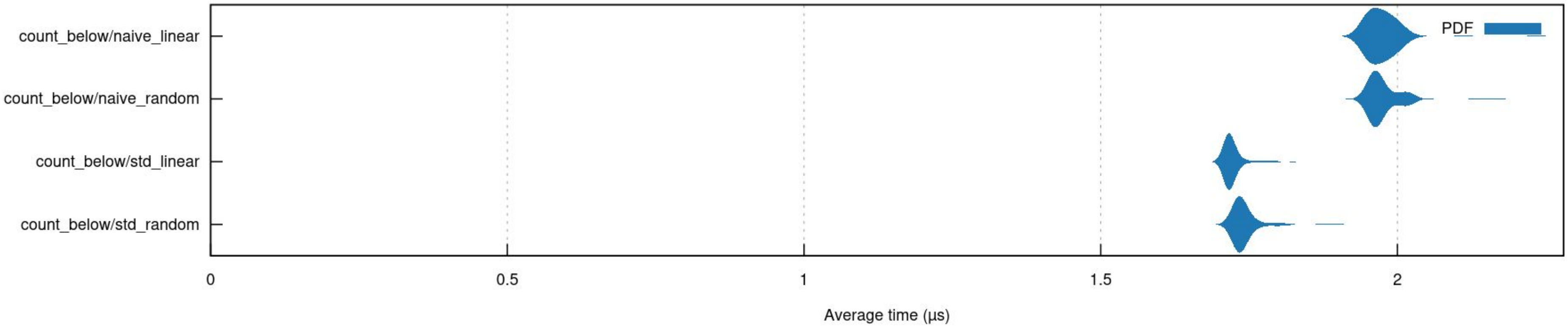
- if condition is true, you win
- else, you rollback

Predict which branch to take based on previous iterations

Effect of if branch

```rust
pub fn count_below_naive(vector: &[usize], n: usize) -> usize {
    let mut count = 0;
    for element in vector {
        if *element < n {
            count += 1;
        }
    }
    count
}
```

```rust
pub fn count_below_std(vector: &[usize], n: usize) -> usize {
    vector.iter().filter(|x| **x < n).count()
}
```



Effect of if branch

```
// Branch prediction hint. This is currently only available on nightly but it
// consistently improves performance by 10-15%.
#[cfg(not(feature = "nightly"))]
use core::convert::identity as likely;
#[cfg(feature = "nightly")]
use core::intrinsics::likely;
// likely has no effect for the usual compiler
// but on the nightly one, it may help the compiler

pub fn count_below_naive(vector: &[usize], n: usize) -> usize {
   let mut count = 0;
   for element in vector {
       if likely(*element < n) {
           // code here will be faster, e.g. in cache
       } else {
           // code here will be slower to execute
       }
   count
}
```

Take home message

- keep the data is local as possible
- avoid ifs
  - These two are equivalent:
    - `if *element < n { count +=1;}`
    - `count += (*element < 1) as usize;`

Let’s sum two slices

```
pub fn sum_two_vec(a: &[u8], b: &[u8]) -> Vec<u8> {
    let mut c = Vec::with_capacity(a.len());
    for i in 0..a.len() {
        c[i] = a[i] + b[i];
    }
    c
}
```

hidden if branch here

sum of u8, but my computer can manipulate more bits at a time

Same function in SIMD

```
#[cfg(target_arch = "x86_64")]
use std::arch::x86_64::{__m256i,
    _mm256_adds_epu8,
    _mm256_loadu_si256,
    _mm256_storeu_si256
};

#[cfg(target_arch = "x86_64")]
pub fn sum_two_vec(a: &[u8], b: &[u8]) -> Vec<u8> {
  // now we need to check the len manually
  // as the compiler will not do it for us
  assert_eq!(a.len(), b.len());

  let len = a.len();
  let mut c = vec![0u8; len];

  // 256 bits / 8 = 32 bytes per __m256i
  let chunks = len / 32;
  let remainder = len % 32;

  let a_ptr = a.as_ptr();
  let b_ptr = b.as_ptr();
  let c_ptr = c.as_mut_ptr();
```

```
  // Safety:
  // We're in a x86-64 architecture (+ more conditions)
  unsafe {
    for i in 0..chunks {
      // cast into 256 bit registers
      let a_chunk =
        _mm256_loadu_si256(a_ptr.add(i * 32) as *const __m256i);
      let b_chunk =
        _mm256_loadu_si256(b_ptr.add(i * 32) as *const __m256i);
      // saturated addition
      let sum = _mm256_adds_epu8(a_chunk, b_chunk);
      // store the result
      _mm256_storeu_si256(c_ptr.add(i * 32) as *mut __m256i, sum);
    }
  }

  // Handle the remainder with scalar code
  let offset = chunks * 32;
   for i in 0..remainder {
       c[offset + i] = a[offset + i].wrapping_add(b[offset + i]);
   }
   c
}
```