# Computer architecture and software optimization procedures

Cecilia Jarne

cecilia.jarne@unq.edu.ar

Twitter: @ceciliajarne

CONICET

Universidad Nacional de Quilmes

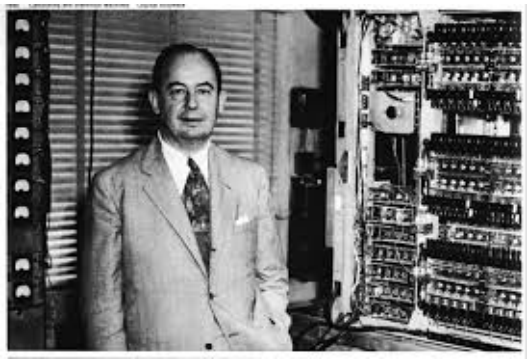
WTPC workshop en técnicas de programación científica

# To start

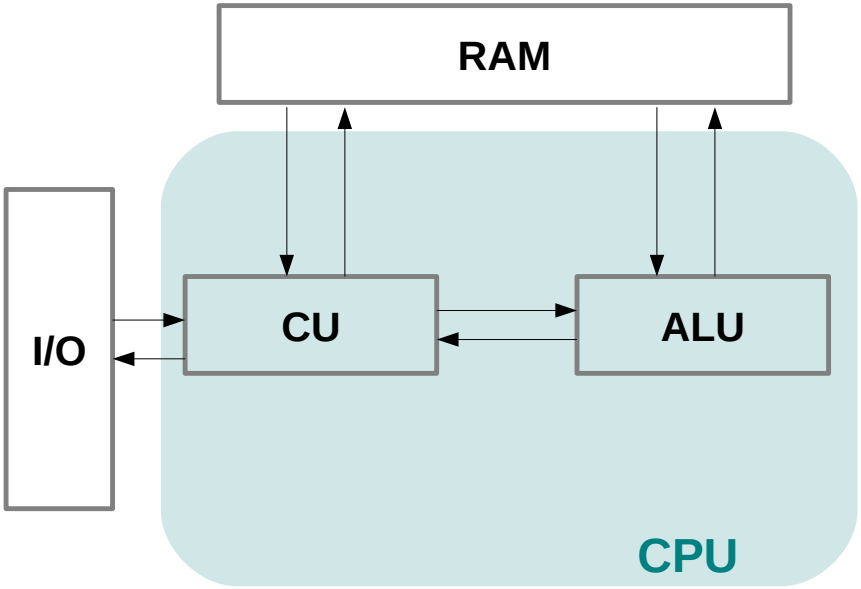
What is this talk about?

- Basic knowledge of computer architecture. .
- Memory and Performance
- Algorithms.

# von Neumann Model

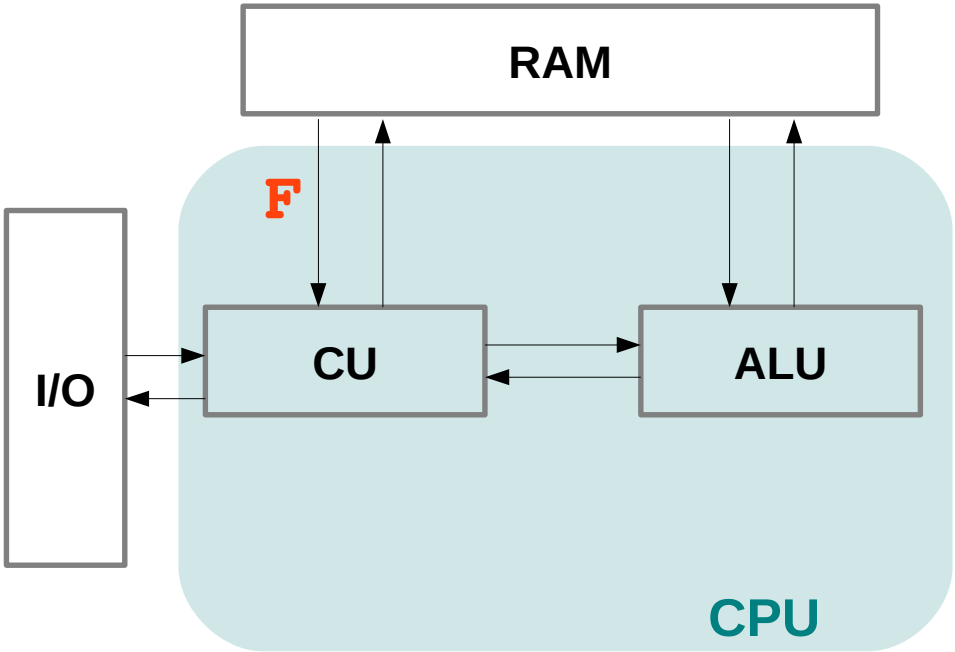


J. Von Neumann and the IAS computer, 1952.



# von Neumann Model

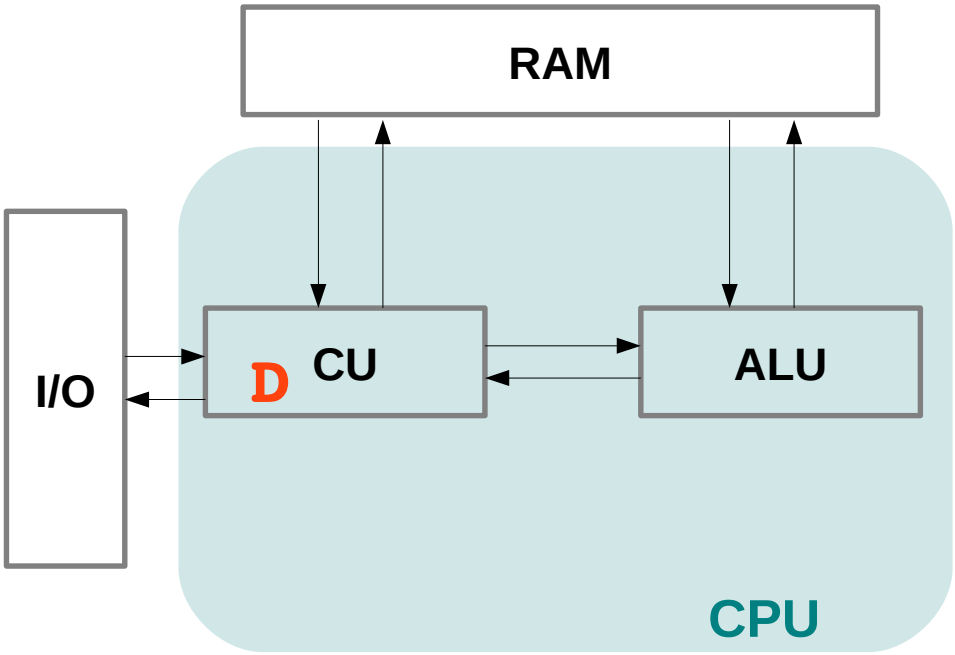
1. FETCH: read next instruction.




0001 1101 0011 0110 1001 **?**

# von Neumann Model

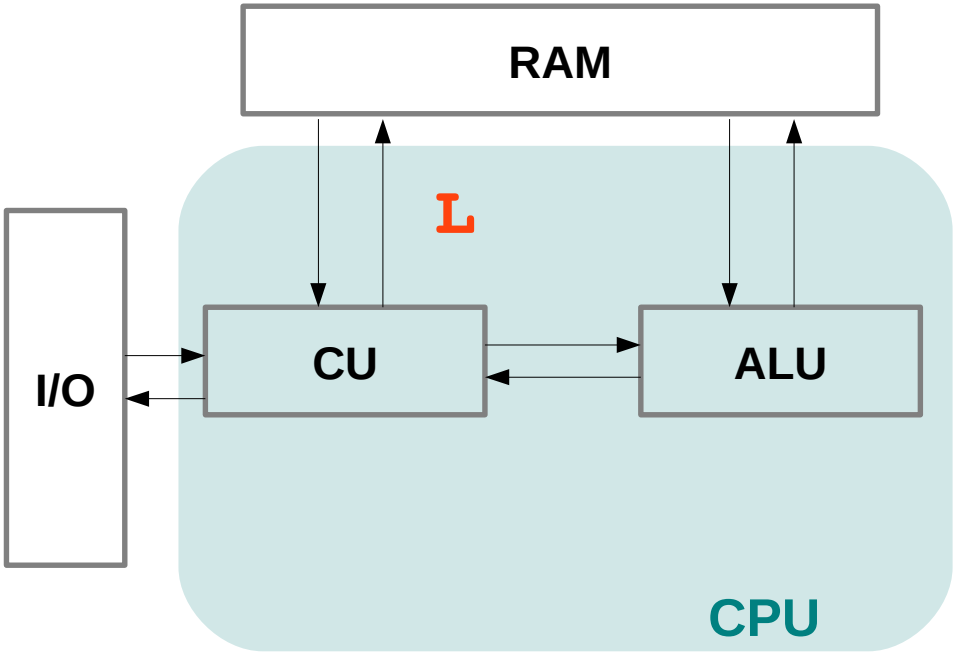
2. DECODE: decodes the instruction.




add r1, 0xF21E !

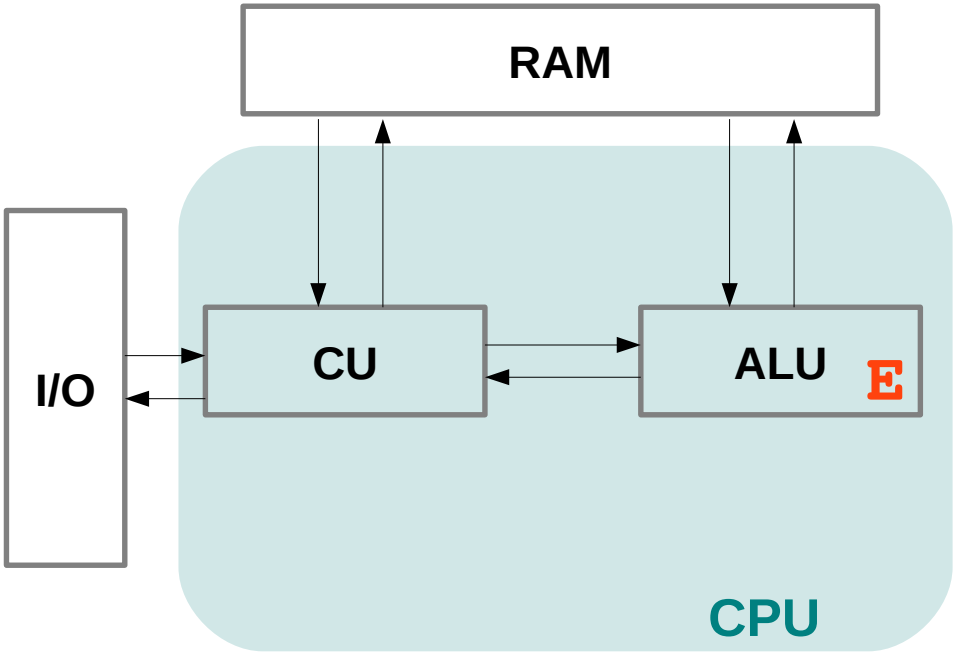
# von Neumann Model

3. LOAD: to ask for the instruction.



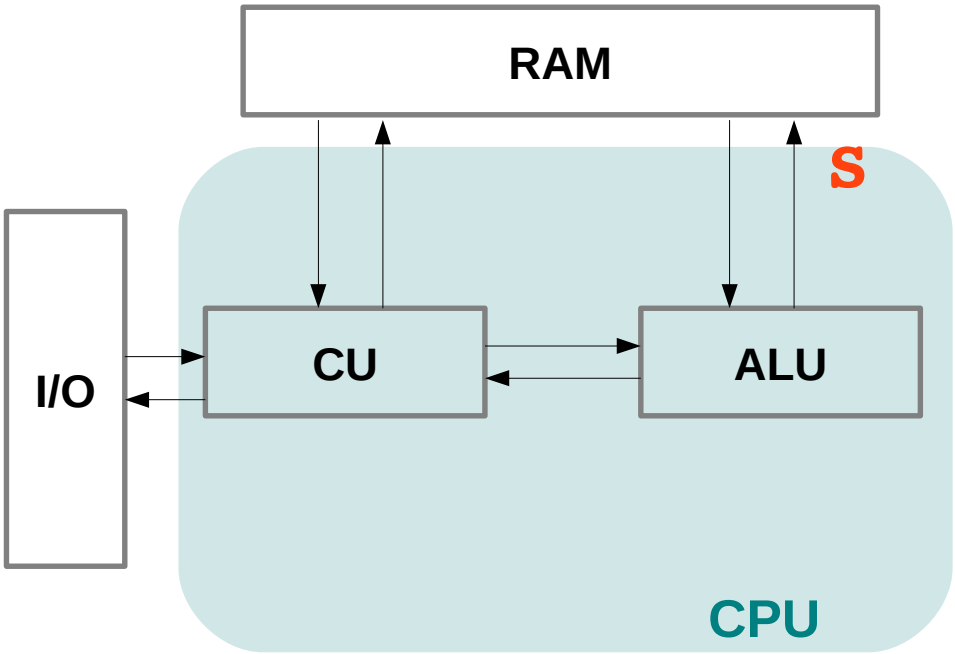

$r2 \leftarrow 0xF21E$

4. EXECUTE: execute the operation.



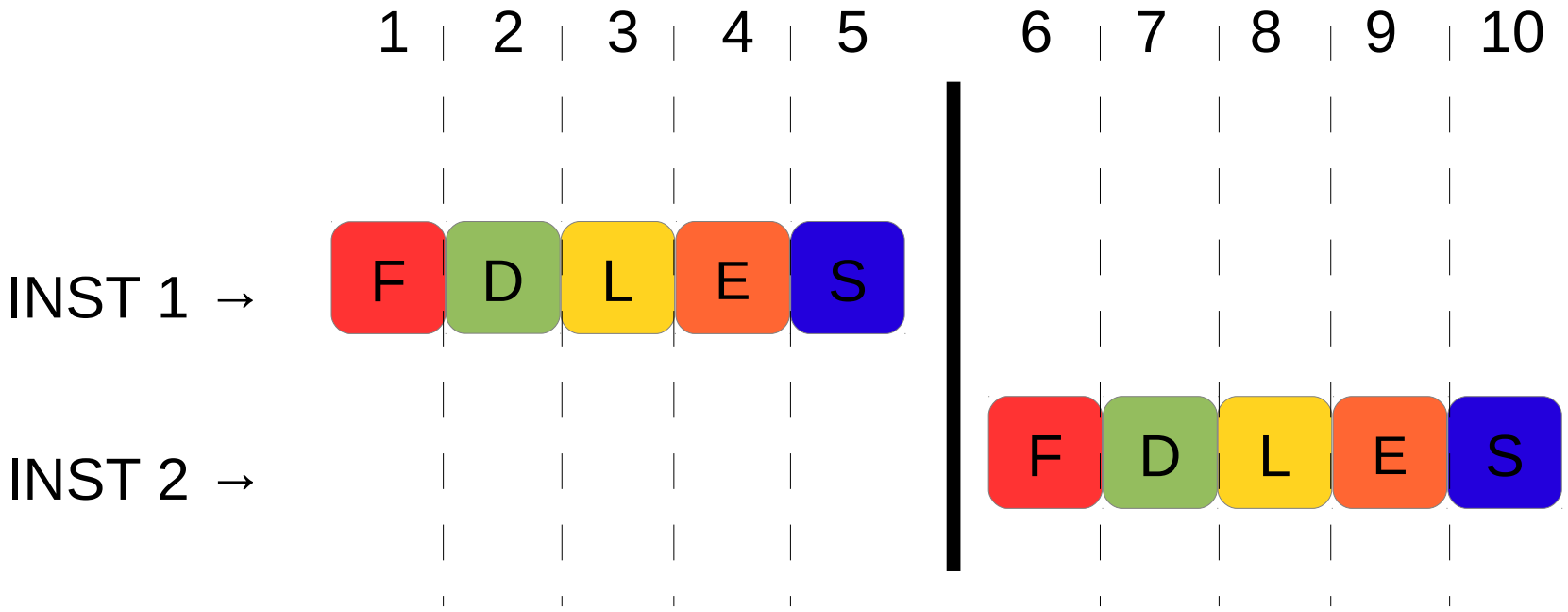

$$r1 \leftarrow r1 + r2$$

5. STORE: saves the result in memory.

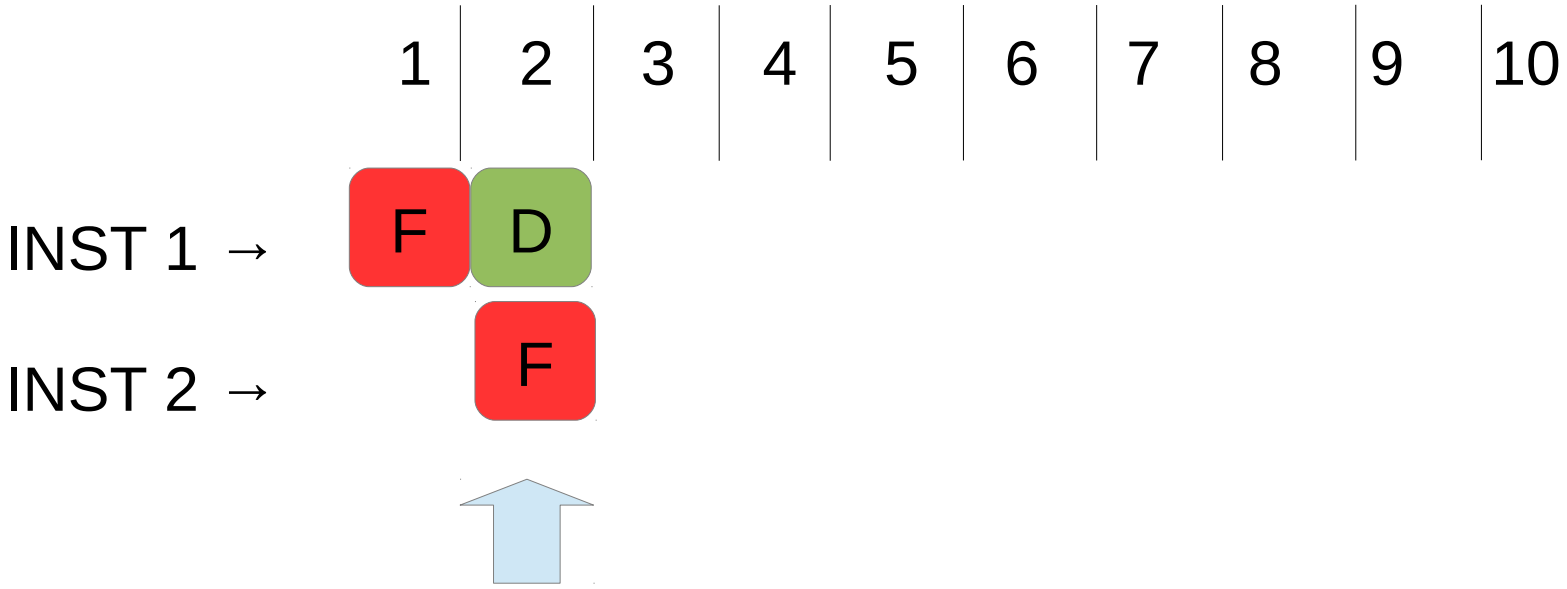



$$RAM \leftarrow r1$$

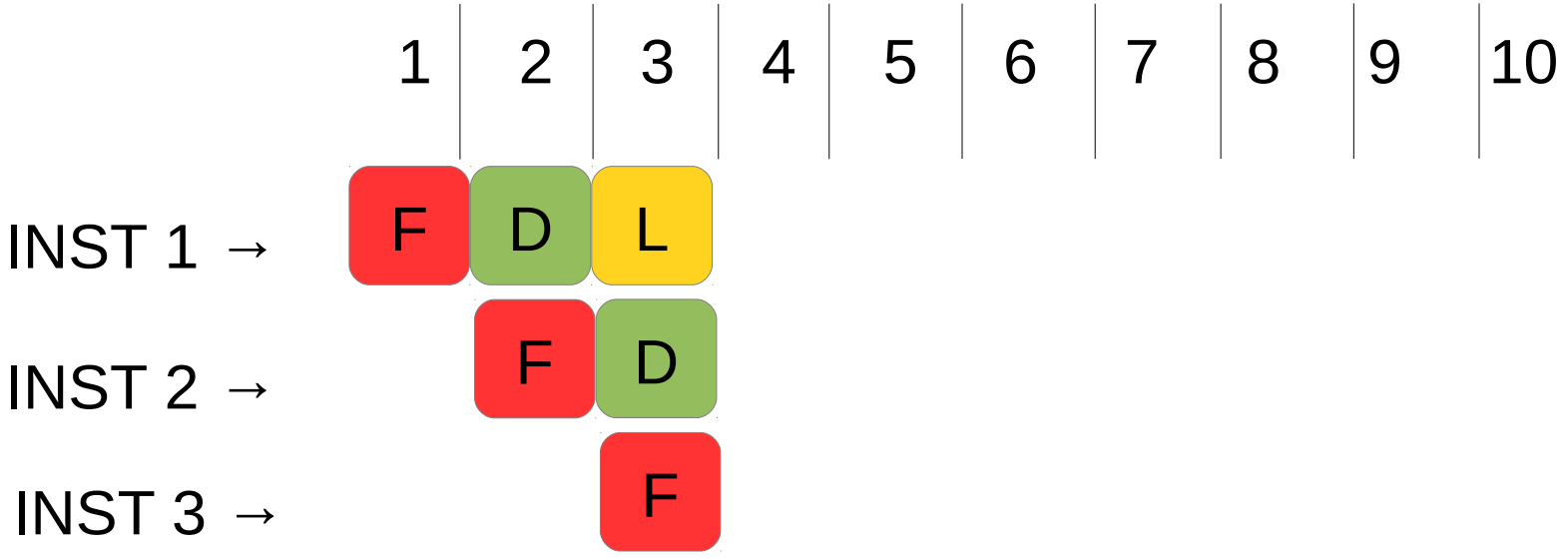
# Sequential processing



# Pipelining

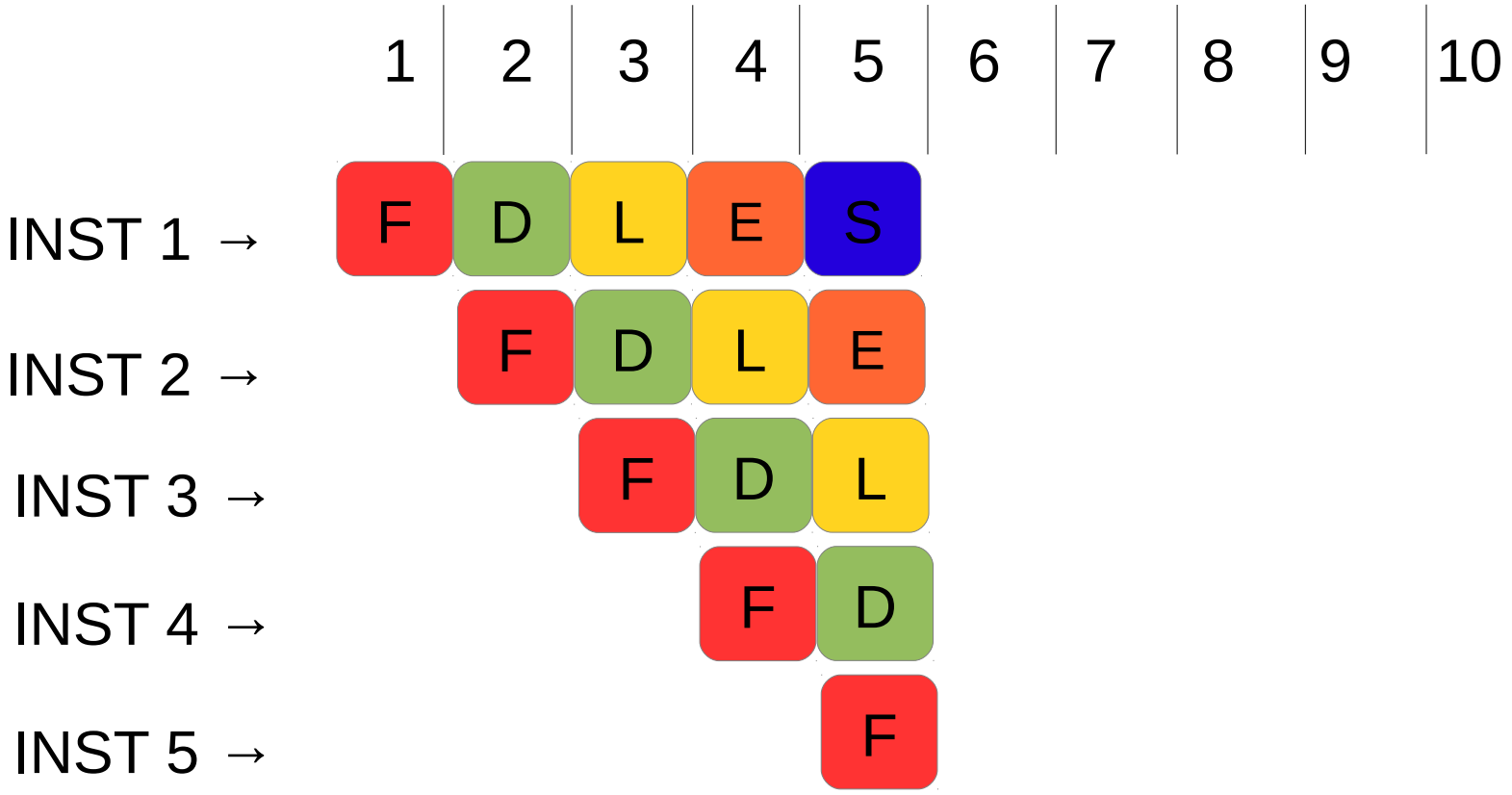


# Pipelining

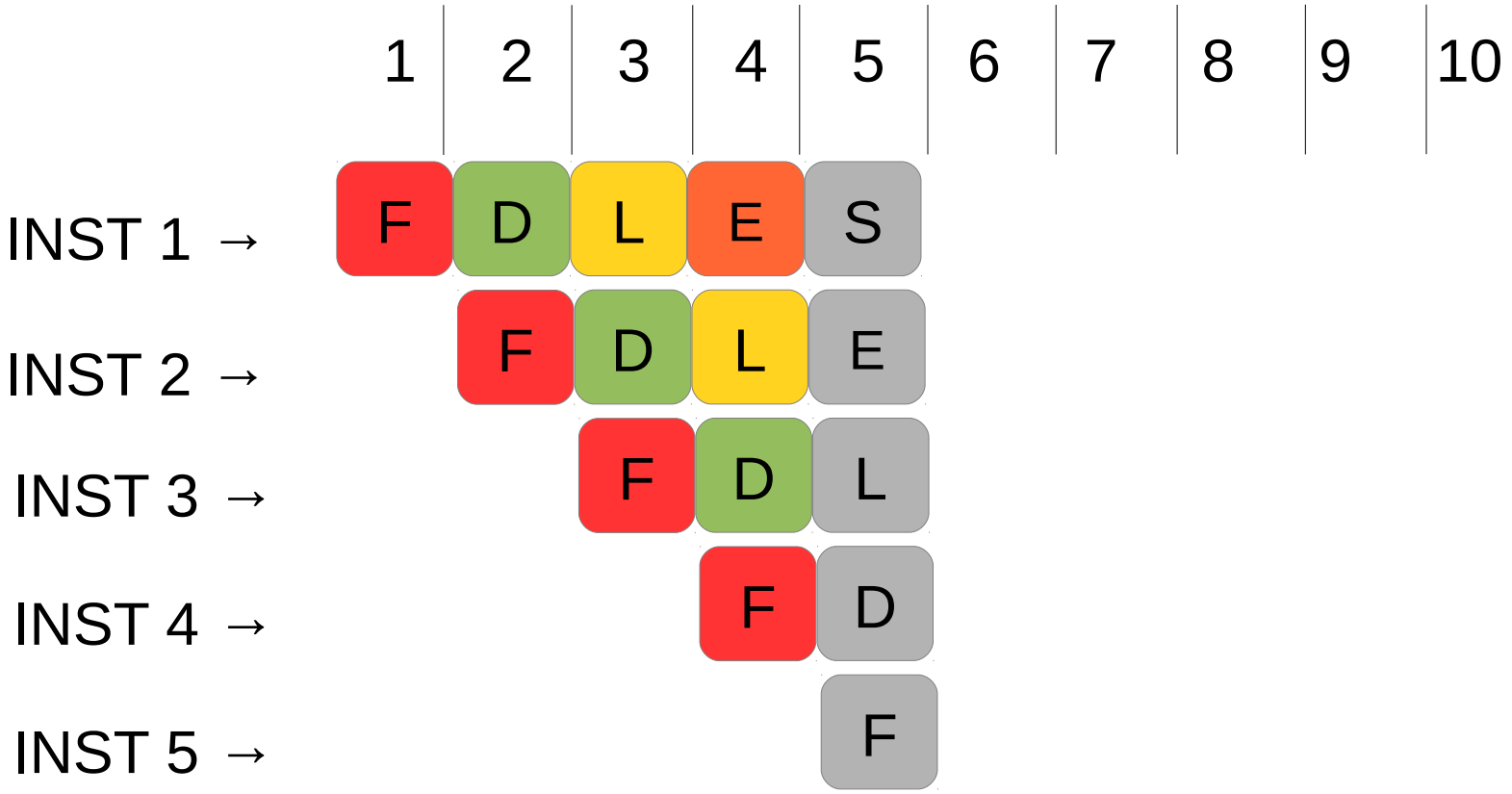


# Pipelining

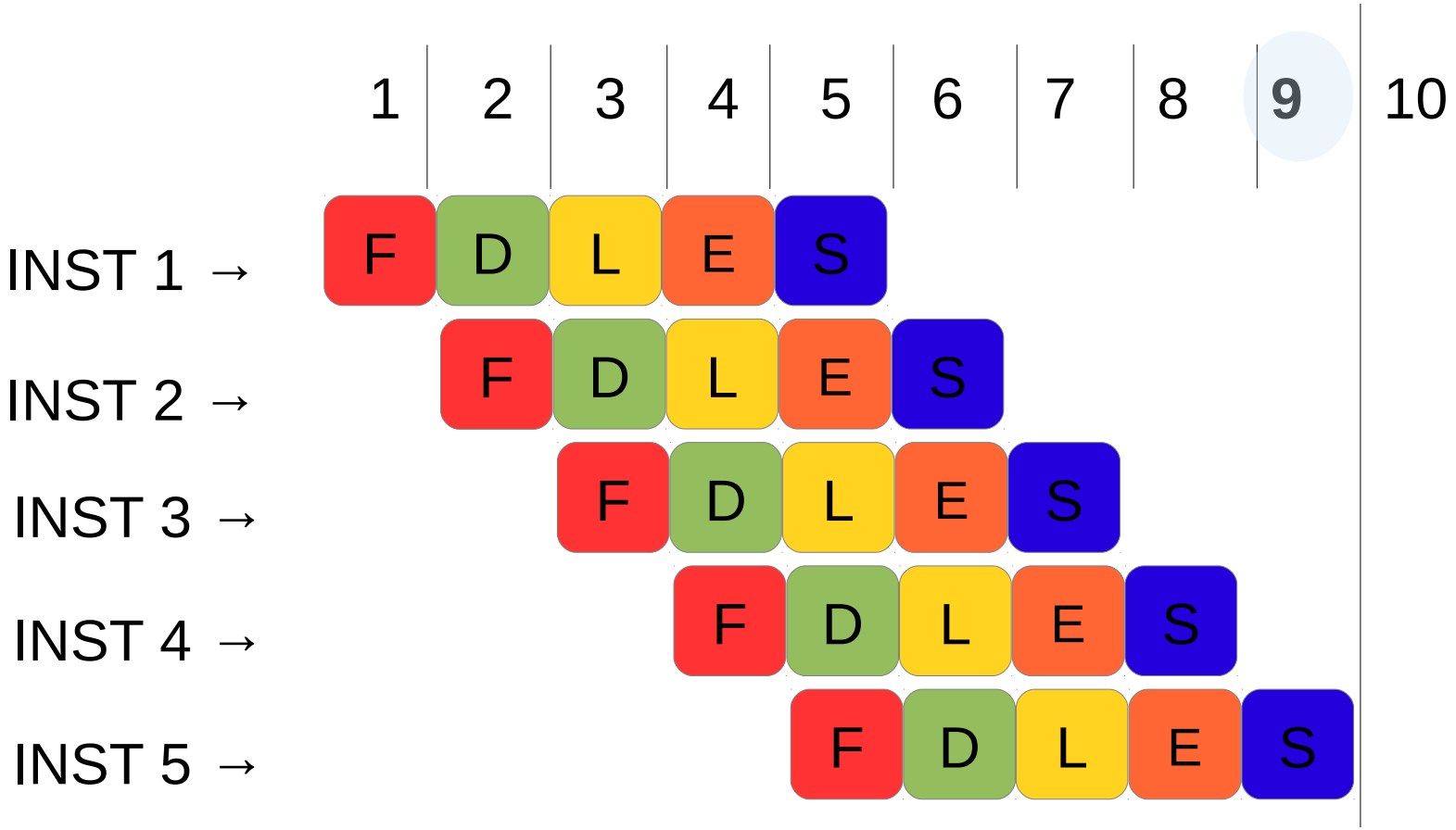
# Pipelining


1
2
3
4
5
6
7
8
9
10
INST 1 →
F D L E S
INST 2 →
F D L E
INST 3 →
F D L
INST 4 →
F D
INST 5 →
F

# Pipelining


1
2
3
4
5
6
7
8
9
10
INST 1 →
F D L E S
INST 2 →
F D L E
INST 3 →
F D L
INST 4 →
F D
INST 5 →
F

# Pipelining



# Pipelining

Let's see an example

$$z = a * b + c * d$$

But actually:

$$z1 = a * b$$

$$z2 = c * d$$

$$z = z1 + z2$$

**C1.** load in $R0$
**C2.** load b in $R1$
**C3.** $R2 = R0 * R1$
Load c in $R3$
**C4.** Load d en $R4$
**C5.** $R5 = R3 * R4$
Inst. next operation
**C6.** $R6 = R2 + R5$
Inst. next operation
**C7.** save $R6$ in $z$
Inst. next operation

# Pipelining

Let's see an example

$$z = a * b + c * d$$

But actually:

$$z1 = a * b$$
$$z2 = c * d$$
$$z = z1 + z2s$$

We only save 1 step of 8!

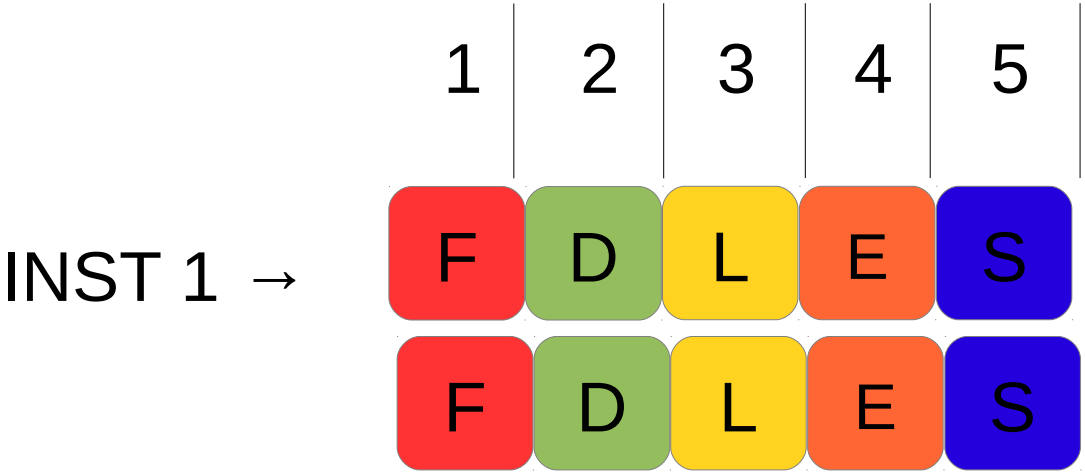
+

3 more if the next operation is independent.

**C1.** load a in $R0$
**C2.** load b in $R1$
**C3.** $R2 = R0 * R1$
load c in $R3$
**C4.** load d in $R4$
**C5.** $R5 = R3 * R4$
Inst. next operation
**C6.** $R6 = R2 + R5$
Inst. next operation
**C7.** save $R6$ en z
Inst. next operation

**Problem: dependencies!**

# Superscaling



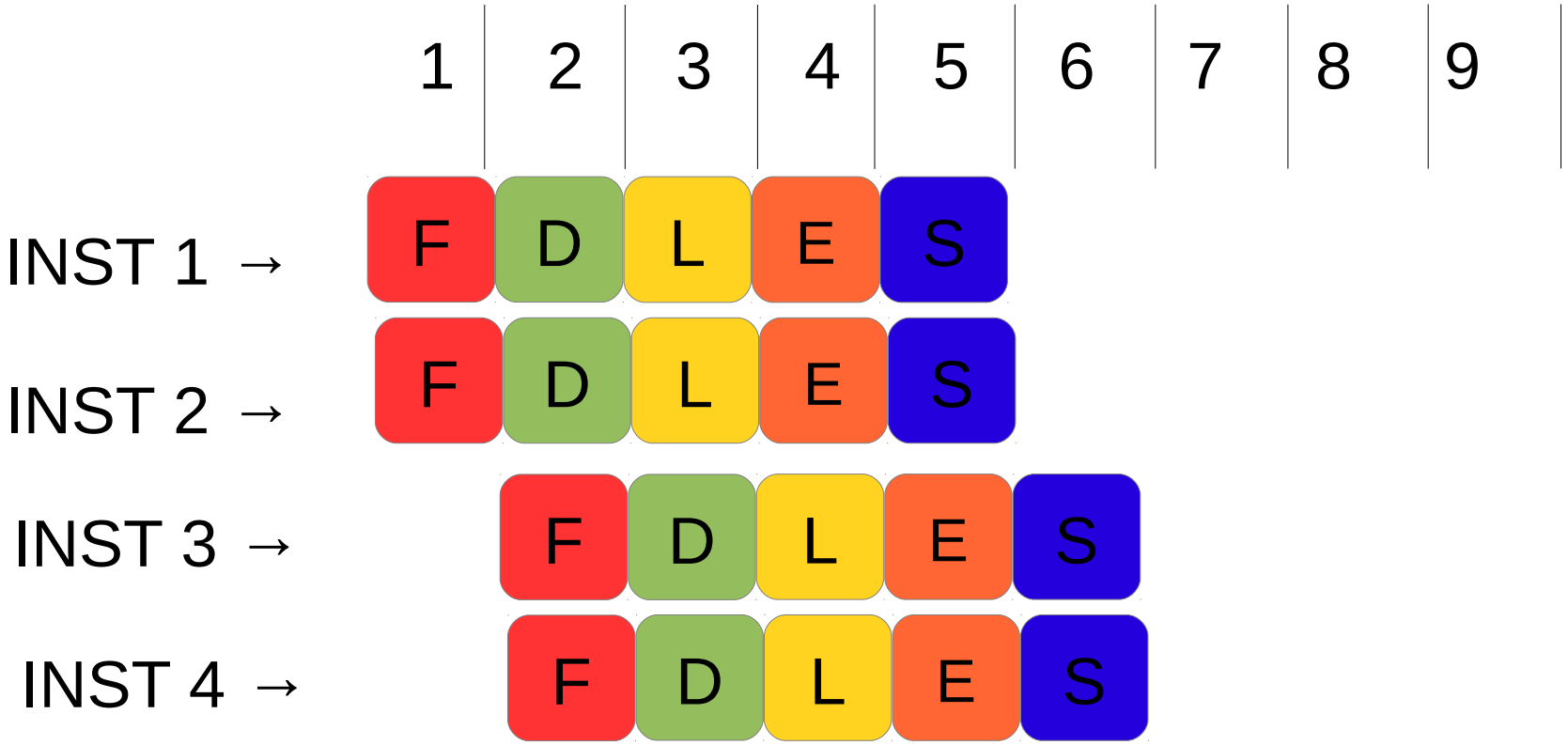

+ of one instruction per cycle
inst. independent

# Superscaling

Be aware, still scalar!

- Parallelism at the instructional level.
- Many instructions are executed simultaneously, usually combined with pipelining.

# Pipelining + Superscaling



# Pipelining + Superscaling

Let's see an example

$$z = a * b + c * d$$

Actually:

$$z1 = a * b$$

$$z2 = c * d$$

$$z = z1 + z2$$

**C1.** load a in $R0$

load b in $R1$

**C2.** $R2 = R0 * R1$

load c en $R3$

load d en $R4$

**C3.** $R5 = R3 * R4$

Inst. next operation

**C4.** $R6 = R2 + R5$

Inst. next operation

**C5.** load $R6$ in z

Inst. next operation

# Pipelining + Superscaling

Let's see an example

$$z = a * b + c * d$$

Actually:

$$z1 = a * b$$

$$z2 = c * d$$

$$z = z1 + z2$$

We save 3 steps from 8!

+

3 more if the next operation is independent

**C1.** load a in $R0$

load b in $R1$

**C2.** $R2 = R0 * R1$

load c in $R3$

load d in $R4$

**C3.** $R5 = R3 * R4$

Inst. next operation

**C4.** $R6 = R2 + R5$

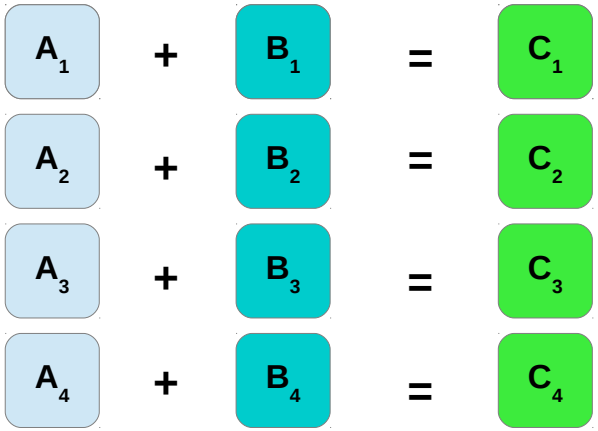
Inst. next operation

**C5.** almacenar $R6$ en z

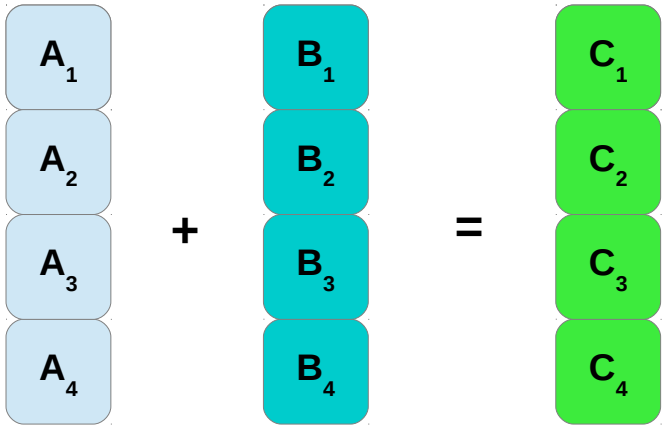
Inst. next operation

# Scalar operations and vector operations

Scalar registers




Vectorial registers



# Vectorial CPU

- Able to execute mathematical operations on multiple data simultaneously. (vector registers)
- In general, it is additional to superscalar pipelining.
- Special vectorial instructions(SSE,AVX,etc).

# Scalar operations: Pipelining + Superscaling

```
1 for (i=0;i<length;i++) \{
2 z[i]=a[i]*b[i]+c[i]*d[i]; \}
```

1. a[0] in R0
   b[0] in R1
2. R2=R0*R1
   c[0] in R3
   D[0] in R4
3. R5=R3*R4
   a[1] in R0
   b[1] in R1
4. R6=R2+R5
   c[1] in R3
   d[1] in R4
5. R6 in z[0]
   R2=R0*R1
   R5=R3*R4
   a[2] in R0
   b[2] in R1

Repeat steps 4-5 according to the index

# Vectorial operations: Pipelining + Superscaling

```
for (i=0;i¡length;i++){
        z[i]=a[i]*b[i]+c[i]*d[i];}
```

Vector registers can store and operate in parallel:

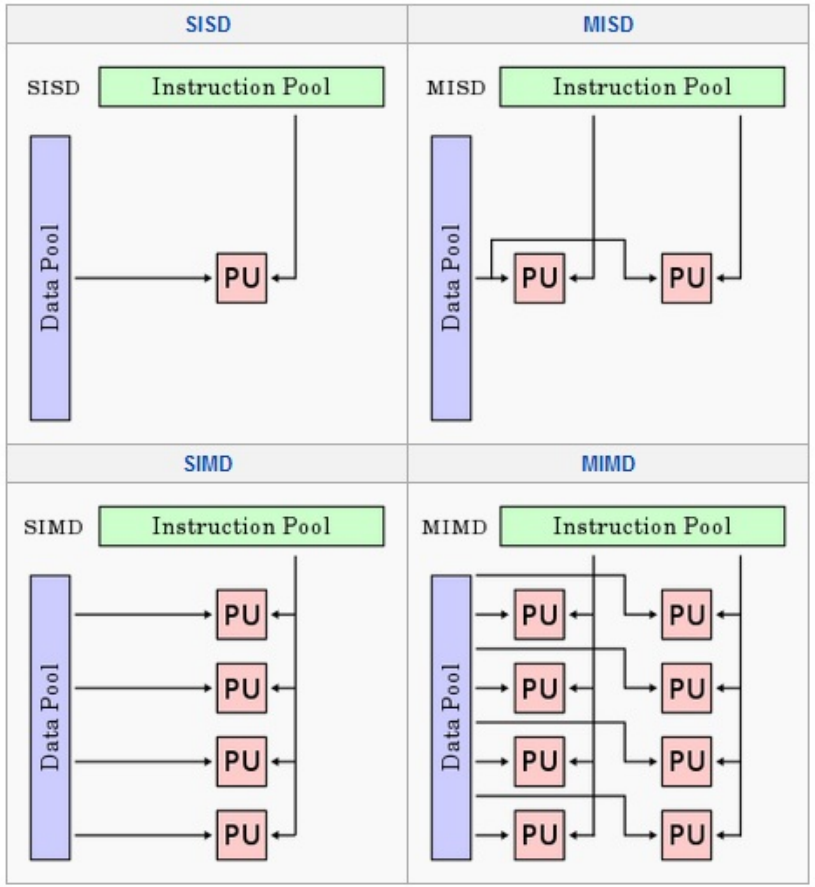
```
for (i=0;i¡length;i+=2){
        z[i]=a[i]*b[i]+c[i]*d[i];
        z[i+1]=a[i+1]*b[i+1]+c[i+1]*d[i+1];}
```

# Classification of architectures

| | | Instrucciones | |
|---|---|---|---|
| | | SI | MI |
| Datos | SD | SISD | (MISD) |
| | MD | SIMD | MIMD |

Taxonomía de Flynn (1966)

S=single, M=multi, I=Instrucción, D=Datos



# How to measure performance

Unit of measurement → FLOPs
**Peak Performance** (theoretical):
Estimation of CPU performance when working at full speed.

**Benchmark Performance**:
Specific tools are used to measure the peak of real" performance.

**Real Performance**:
Measurement made with the program that I want to run.

# How to measure the performance: Benchmark

*Linpack*:package that solves a system of dense matrices (with random values) of double precision (64 bits) and measures the performance of the system.(http://www.netlib.org/benchmark/hpl/)

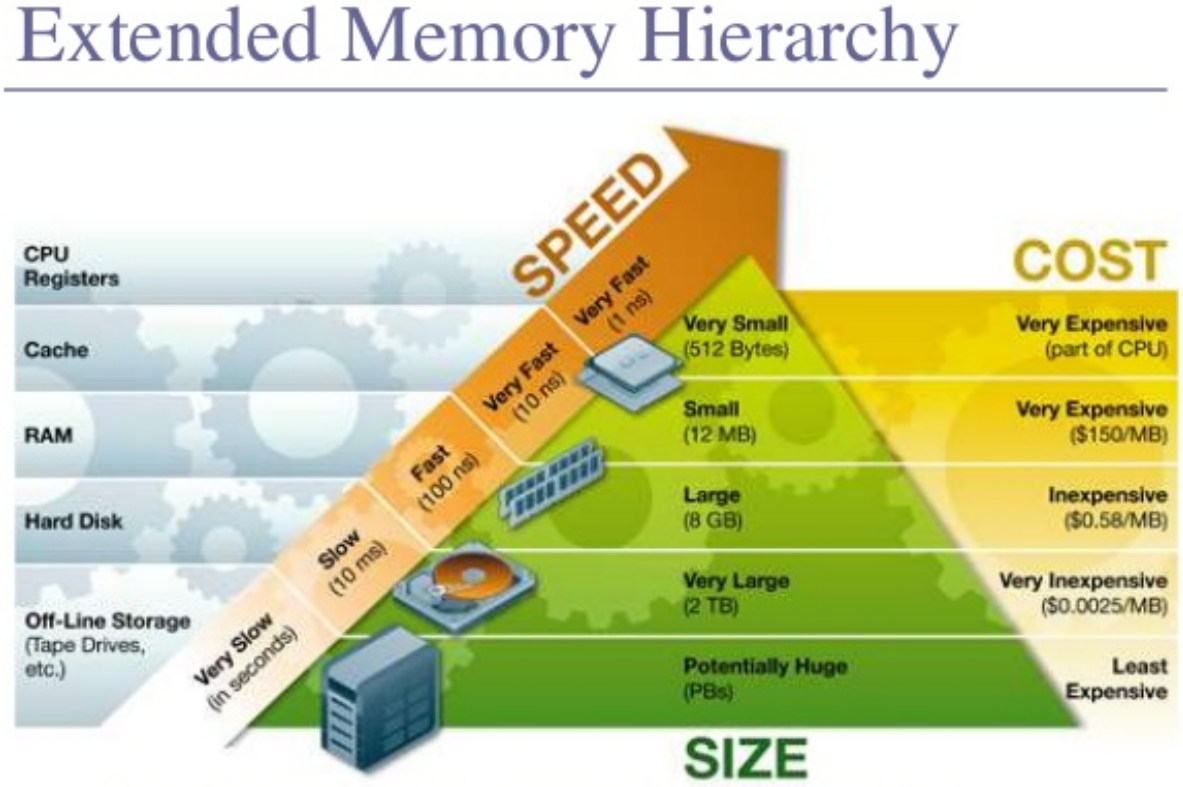
| | |
|---|---|
| OS | Linux 6.2 RedHat (Kernel 2.2.14) |
| C compiler | gcc (egcs-2.91.66 egcs-1.1.2 release) |
| C flags | -fomit-frame-pointer -O3 -funroll-loops |
| MPI | MPIch 1.2.1 |
| BLAS | ATLAS (Version 3.0 beta) |
| Comments | 09 / 00 |

Performance (Gflops) w.r.t Problem size on 4 nodes.

| **GRID** | **2000** | **5000** | **8000** | **10000** |
|---|---|---|---|---|
| **1 x 4** | 1.28 | 1.73 | 1.89 | 1.95 |
| **2 x 2** | 1.17 | 1.68 | 1.88 | 1.93 |
| **4 x 1** | 0.81 | 1.43 | 1.70 | 1.80 |

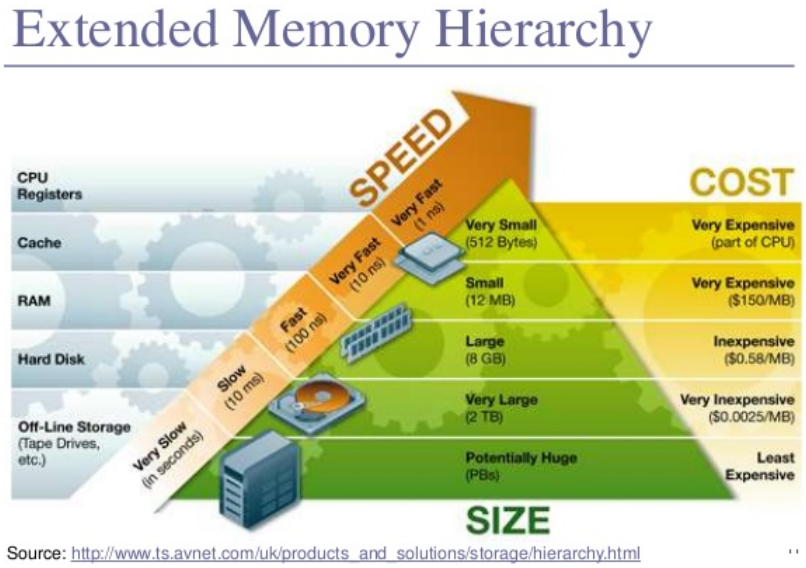
4 AMD Athlon K7 500 Mhz (256 Mb) - (2x) 100 Mbs Switched
2 NICs per node (channel bonding)

# Memory hierarchy




Source: http://www.ts.avnet.com/uk/products_and_solutions/storage/hierarchy.html

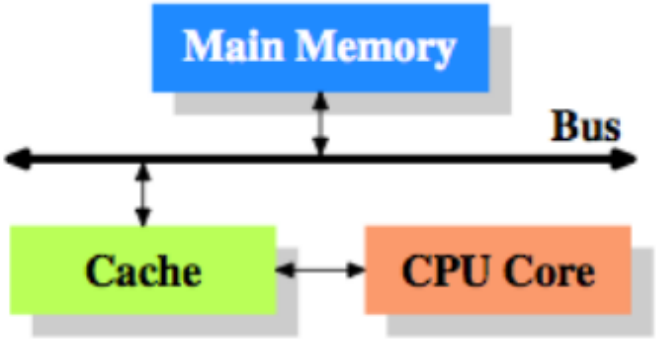
# Memory hierarchy

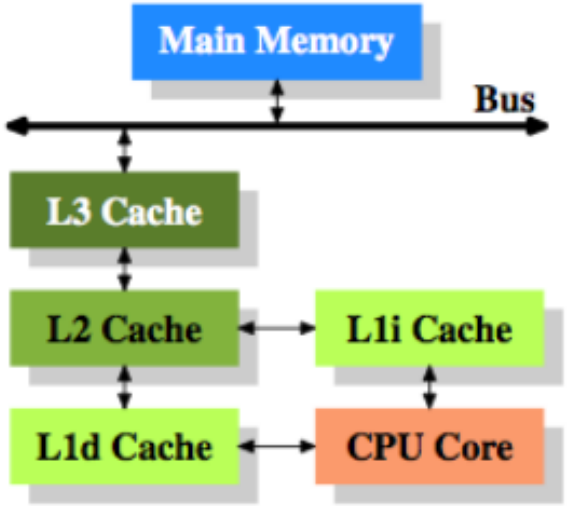



*Registers*: Wired in the Processor.
*Cache*:Fast memory, close to the processor and expensive.
*Principal Memory*: RAM, slow and cheap.
*Hard disk*: very slow and very cheap.

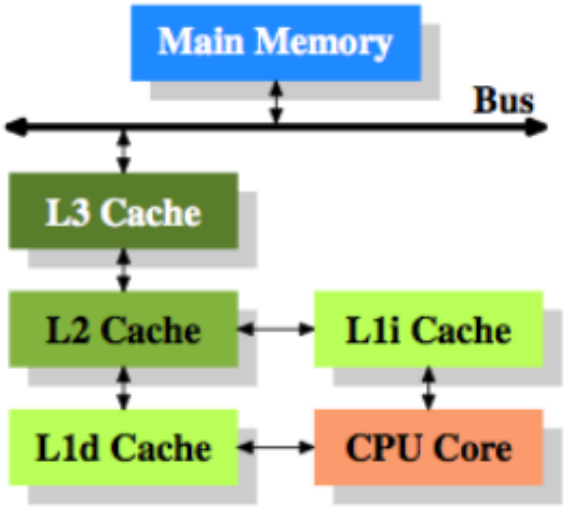
# Cache memory

Higher cost.
Higher speed.
Lower capacity.
Data is transferred to cache in blocks of a certain size.



# Cache memory




```
maria@maria-UX21E: ~
maria@maria-UX21E:~$ lscpu
Arquitectura:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Orden de bytes:        Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Hilo(s) por núcleo:    2
Núcleo(s) por zócalo:2
Socket(s):             1
Nodo(s) NUMA:          1
ID del vendedor:       GenuineIntel
Familia de CPU:        6
Modelo:                42
Stepping:              7
CPU MHz:               800.000
BogoMIPS:              3190.05
Virtualización:        VT-x
caché L1d:             32K
caché L1i:             32K
caché L2:              256K
caché L3:              3072K
NUMA node0 CPU(s):     0-3
maria@maria-UX21E:~$
```

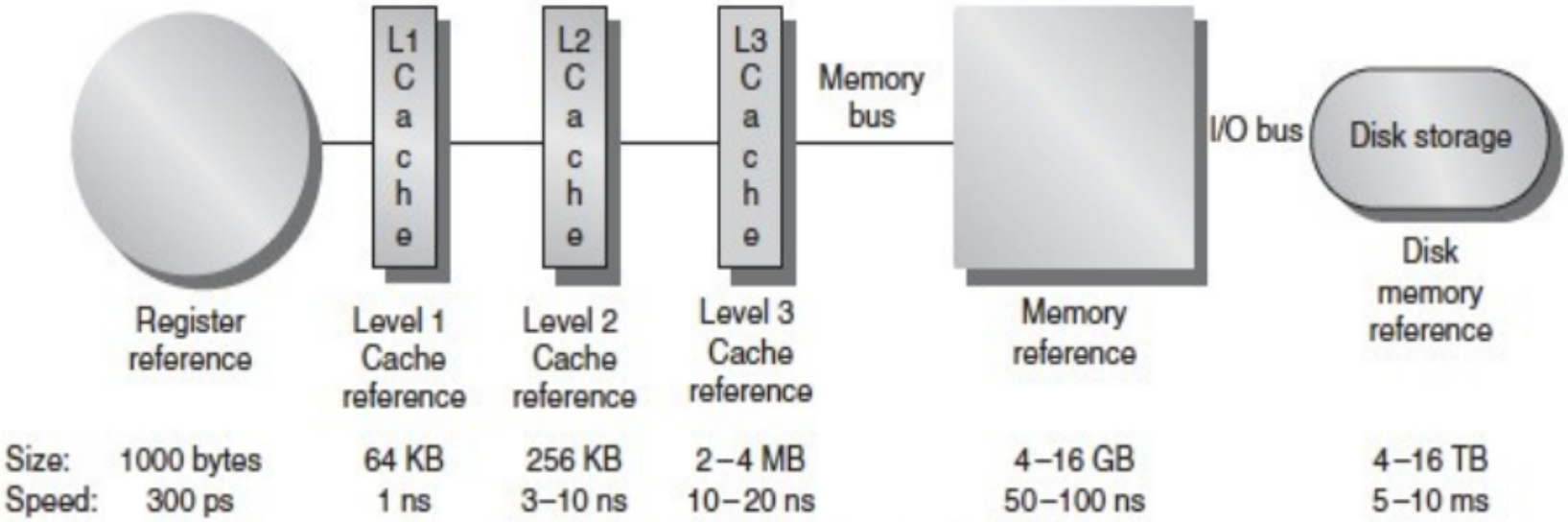
# Cache memory

Principle of locality:

- **Spatial**
  For example, go over a vector with a loop.
- **Temporal**
  For example, a call to a function within a loop.

Each time a LOAD / STRORE operation is performed, it can occur:
**cache miss** o **cache hit**

# Cache memory



# Memory hierarchy

How to compare speeds in a more legible way?

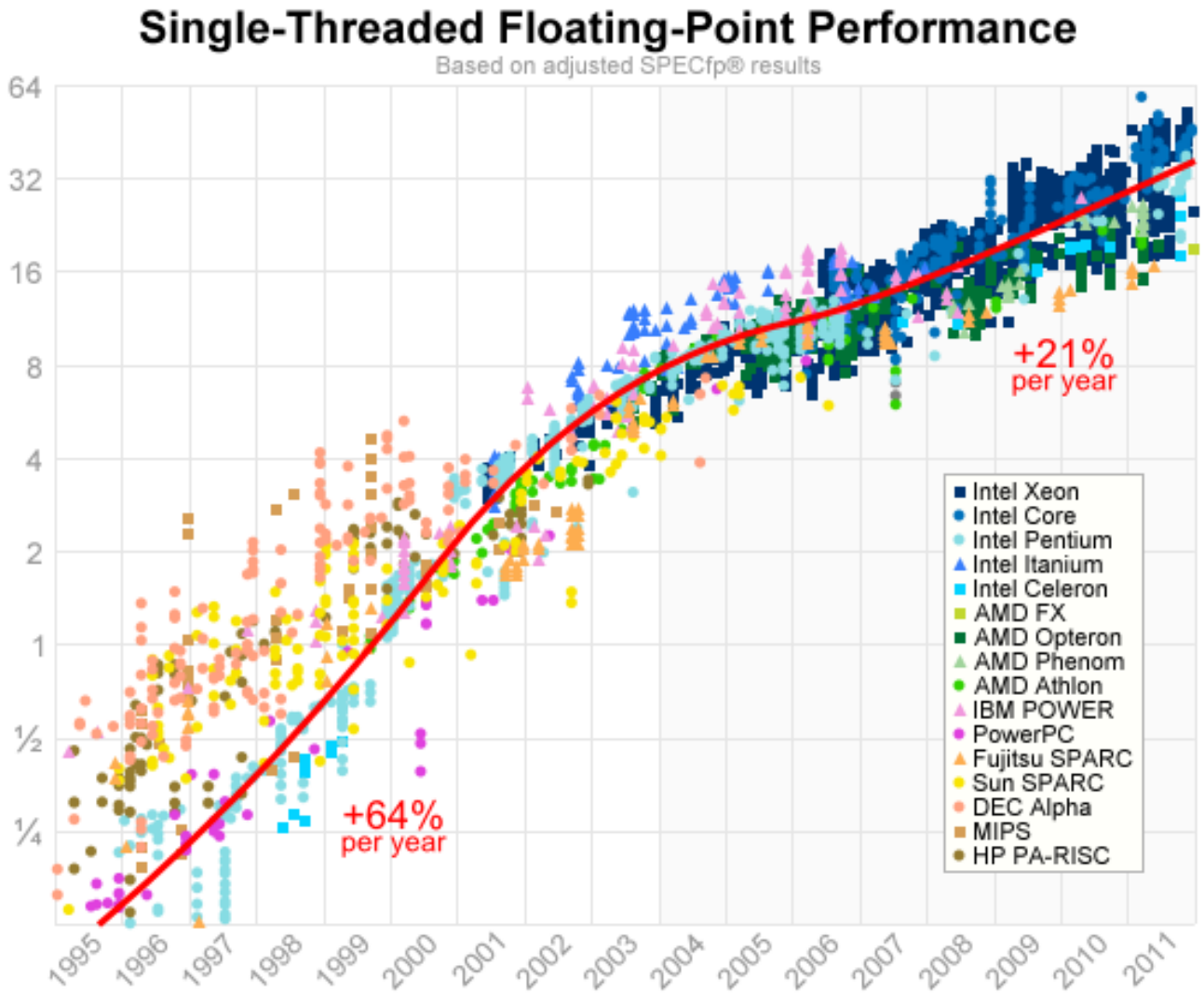
| | | |
|---|---|---|
| **1 CPU cycle** | **0.3 ns** | **1 s** |
| Level 1 cache access | 0.9 ns | 3 s |
| Level 2 cache access | 2.8 ns | 9 s |
| Level 3 cache access | 12.9 ns | 43 s |
| Main memory access | 120 ns | 6 min |
| Solid-state disk I/O | 50-150 µs | 2-6 days |
| **Rotational disk I/O** | **1-10 ms** | **1-12 months** |
| Internet: SF to NYC | 40 ms | 4 years |
| Internet: SF to UK | 81 ms | 8 years |
| Internet: SF to Australia | 183 ms | 19 years |
| OS virtualization reboot | 4 s | 423 years |
| SCSI command time-out | 30 s | 3000 years |
| Hardware virtualization reboot | 40 s | 4000 years |
| Physical system reboot | 5 m | 32 millenia |

# Memory hierarchy

All this for a single processor!

Currently all CPUs have more than one processor.

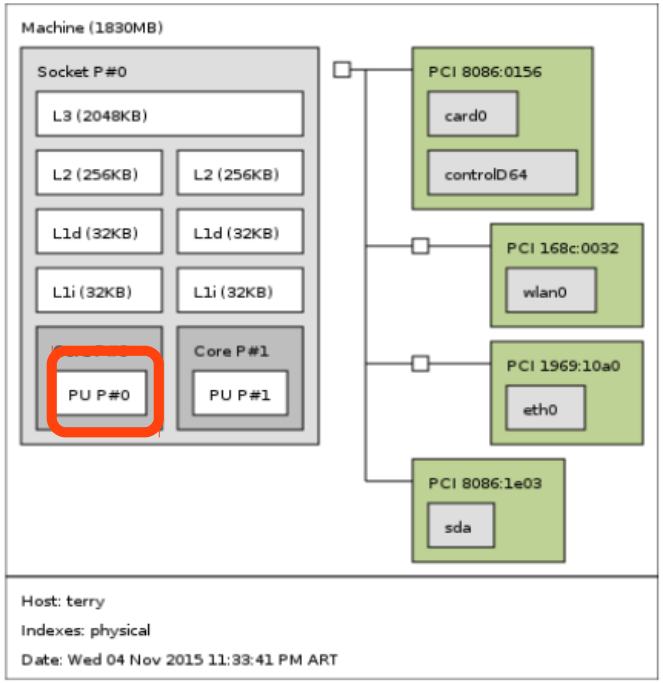
# Multiprocessors



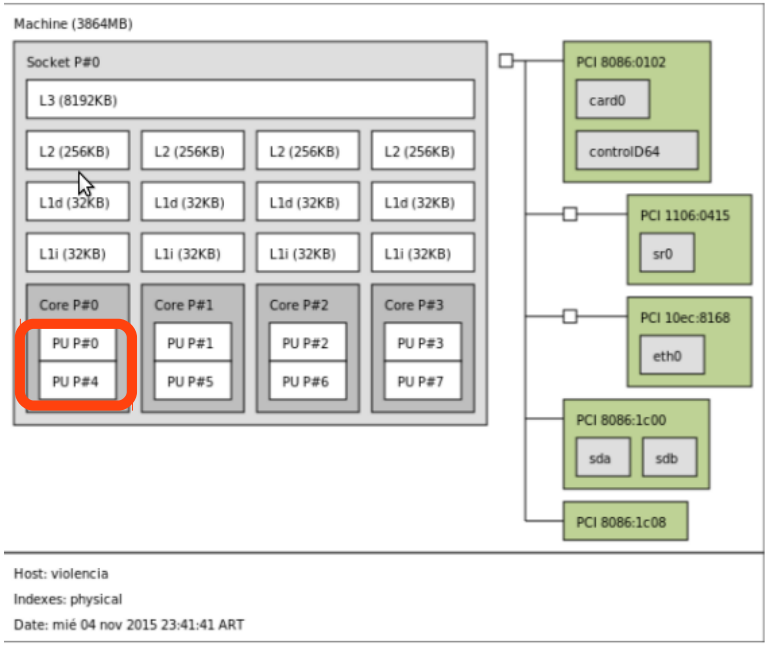

Moore's Law?

# Multiprocessors

How really is:




Intel(R) Celeron(R) CPU 1007U




Intel(R) Core(TM) i7-2600 CPU

Basically von Neumann

# The process of writing scientific software:

We code without much planning
(How hard can it be ???)
↓
The first version that "works"(correct results ?!)

# The process of writing scientific software:

But when the problem grows ...

We just noticed that they are not **optimal solutions**

# What do we optimize?

We must think about achieving a trade-off between:

Development time, Debugging, Validation, portability, etc.
And of course the execution time.

Important: CPU time is more cheap that human time!

# What do we optimize?

Surely someone already solved part of the problem!

**We are not going to invent the wheel.**

# Use optimized libraries

Some scientific libraries (some already mentioned)

- LAPACK — Linear Algebra PACKage (http://www.netlib.org/lapack/)
- GSL - GNU Scientific Library (https://www.gnu.org/software/gsl/)
- C++ Boost (http://www.boost.org/)
- Scipy (https://www.scipy.org/)
- Numpy (http://www.numpy.org/)
- Pandas (http://pandas.pydata.org/)

# What else can we do?

LEARN Google Skils!
(Evaluate what exists)
Surely we will get something of better quality than if we try to write it ourselves from scratch.

And
if we have to program, CHOOSE A PROGRAMMING LANGUAGE
(Keep in mind what exists, know your strengths and weaknesses)

# What else can we optimize?

- Use approximations when possible.
- Develop more algorithms efficient.
- Use data structures appropriate.
- Get faster hardware.
- Use or write software optimized for the hardware that is owned. For example: take advantage of libraries and optimized, BLAS (www.netlib.org/blas), LAPACK (www.netlib.org/lapack),etc.
- Parallelize.

# Optimization

- Regarding **algorithms**.
- Regarding **data structures**.
- Regarding **hardware**.

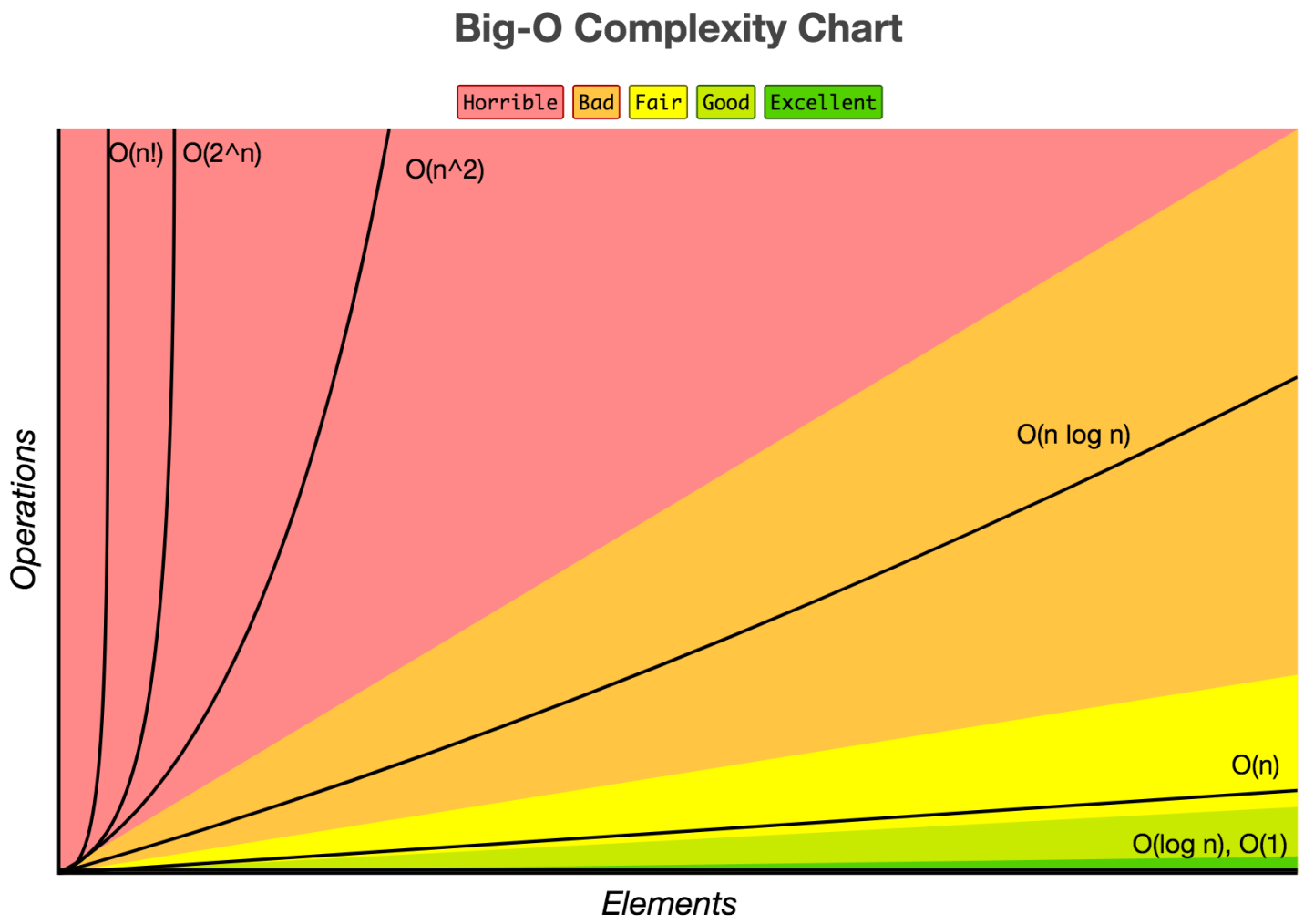
# How to choose the right algorithm?

- **Theoretical evaluation:**.
  Complexity in time and storage. (Analyze how involves the program as the size of the entrance grows)

- **Practical evaluation:**
  Take measurements (profiling)

# Algorithms: theoretical evaluation

Asymptotic complexity and large 'O' notation: Estimate how the time will grow of execution as the size of the entry.



# Algorithms: theoretical evaluation

| n | O(1) | O(log(n)) | O(n) | O(n log(n)) | $O(n^2)$ | $O(n^3)$ |
|---|---|---|---|---|---|---|
| 10 | const | 3 | 10 | 33 | 100 | 1000 |
| 1000 | const | 10 | 1000 | 9966 | 1E+06 | 1E+09 |
| 100000 | const | 17 | 100000 | 1660964 | 1E+10 | 1E+15 |
| 1000000 | const | 20 | 1000000 | 19931569 | 1E+12 | 1E+18 |

$$O(1)<O(\log n)<O(n)<O(n^2)<O(n^3)\ldots<O(c^n)$$

# Example: Sorting algorithms

## Array Sorting Algorithms

| Algorithm | Time Complexity | | | Space Complexity |
|---|---|---|---|---|
| | Best | Average | Worst | Worst |
| Quicksort | Ω(n log(n)) | Θ(n log(n)) | O(n^2) | O(log(n)) |
| Mergesort | Ω(n log(n)) | Θ(n log(n)) | O(n log(n)) | O(n) |
| Timsort | Ω(n) | Θ(n log(n)) | O(n log(n)) | O(n) |
| Heapsort | Ω(n log(n)) | Θ(n log(n)) | O(n log(n)) | O(1) |
| Bubble Sort | Ω(n) | Θ(n^2) | O(n^2) | O(1) |
| Insertion Sort | Ω(n) | Θ(n^2) | O(n^2) | O(1) |
| Selection Sort | Ω(n^2) | Θ(n^2) | O(n^2) | O(1) |
| Tree Sort | Ω(n log(n)) | Θ(n log(n)) | O(n^2) | O(n) |
| Shell Sort | Ω(n log(n)) | Θ(n(log(n))^2) | O(n(log(n))^2) | O(1) |
| Bucket Sort | Ω(n+k) | Θ(n+k) | O(n^2) | O(n) |
| Radix Sort | Ω(nk) | Θ(nk) | O(nk) | O(n+k) |
| Counting Sort | Ω(n+k) | Θ(n+k) | O(n+k) | O(k) |
| Cubesort | Ω(n) | Θ(n log(n)) | O(n log(n)) | O(n) |

WTPC workshop en técnicas de programación científica

# Example: Sorting algorithms

5 Sorting Algorithms in 6 minutes.
http://youtube/kPRA0W1kECg

# How much the choice of a good algorithm matters

Algorithm for the Fourier Transformation.

Cost TDF $O(N^2) \rightarrow$ Cost FFT $O(N \log_2 N)$

*J. W. Cooley,J. W. Tukey (1965)*

What is really beneficial?

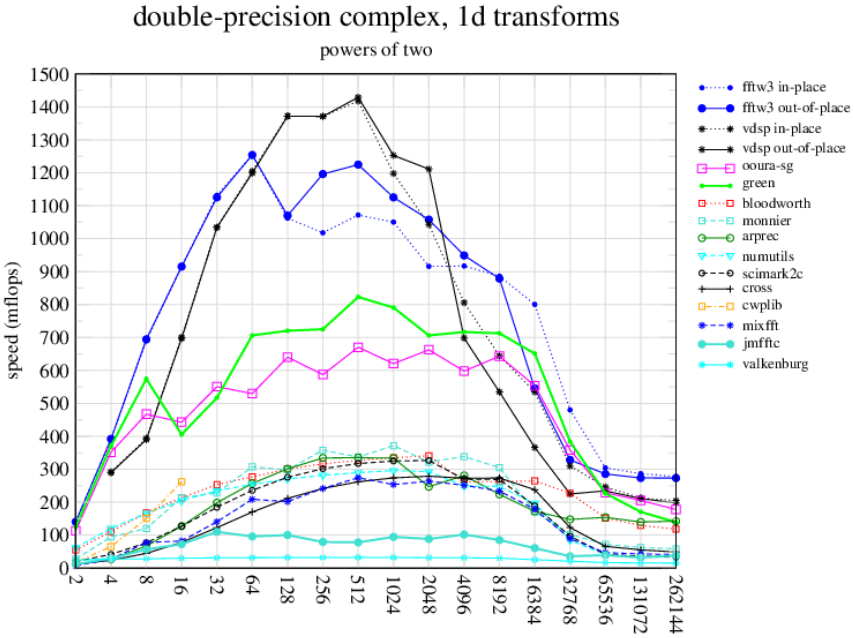
| **N** | 1000 | $10^6$ | $10^9$ |
|---|---|---|---|
| $O(N^2)$ | $10^6$ | $10^{12}$ | $10^{18}$ |
| $O(N \log_2 N)$ | $10^4$ | $20 \times 10^6$ | $30 \times 10^9$ |

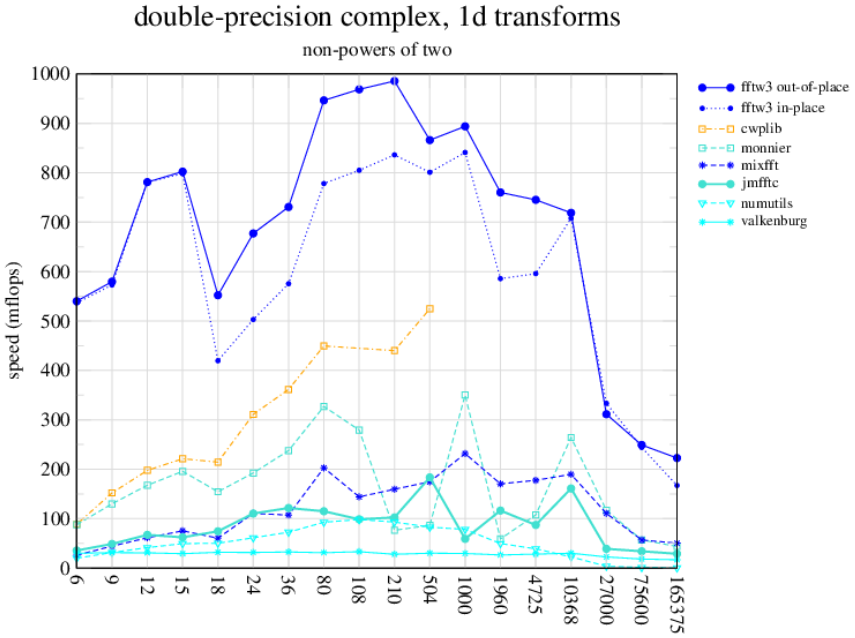
Suppose that each operation takes 1 *ns*

$10^{18}$ ns 31.2 years

$30 \times 10^9$ ns 30 sec

# How much the choice of a good algorithm matters?

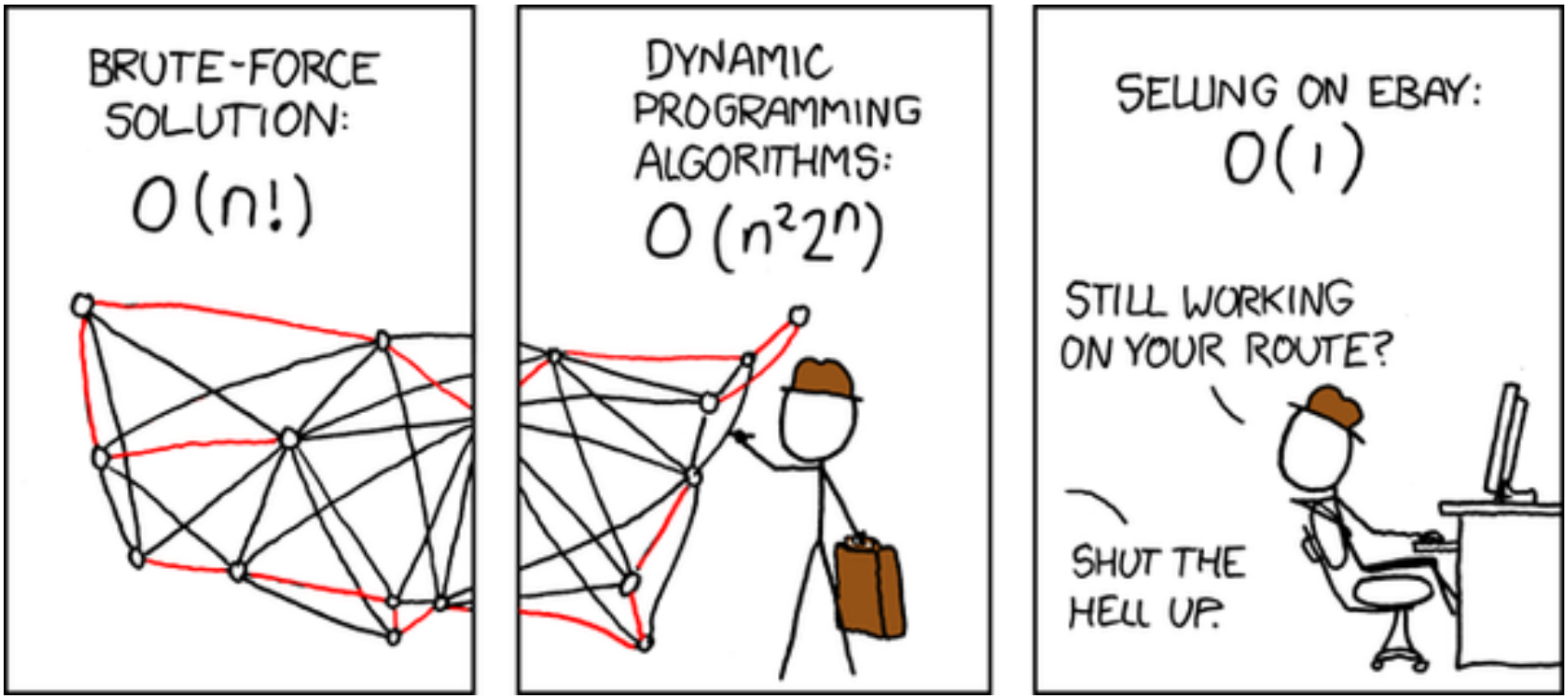



http://www.fftw.org/speed/G4-1.06GHz-macosx/

# How much the choice of a good algorithm matters?

Finding a better algorithm is better than optimizing an algorithm!

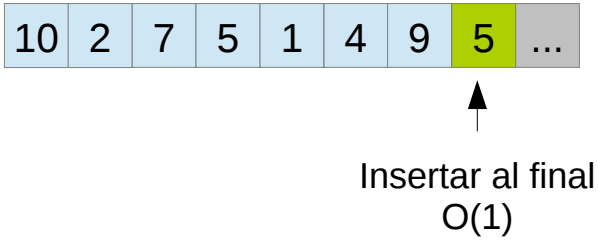


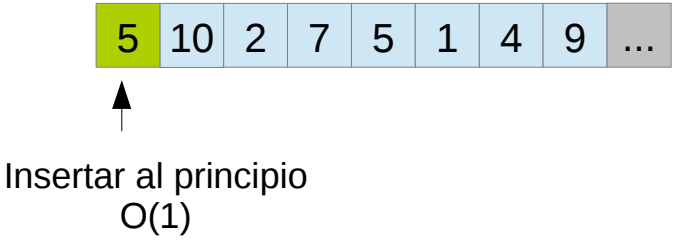
# Optimization with respect to data structures

- Store information in an organized and structured way and not as simple data.
- They can be seen as a collection of data that are characterized by their organization and the operations that are defined in them.
- Objective: to have easy access and data management.

# Sequential containers

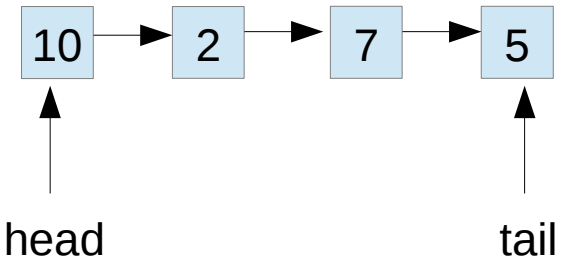
## Arrays



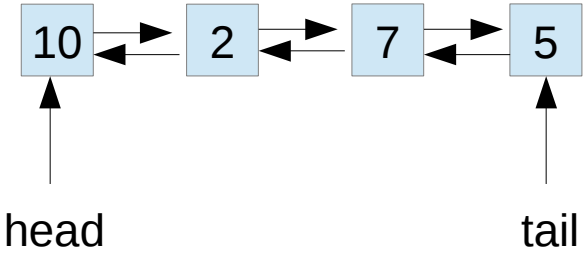




## Linked lists

Single linked-list

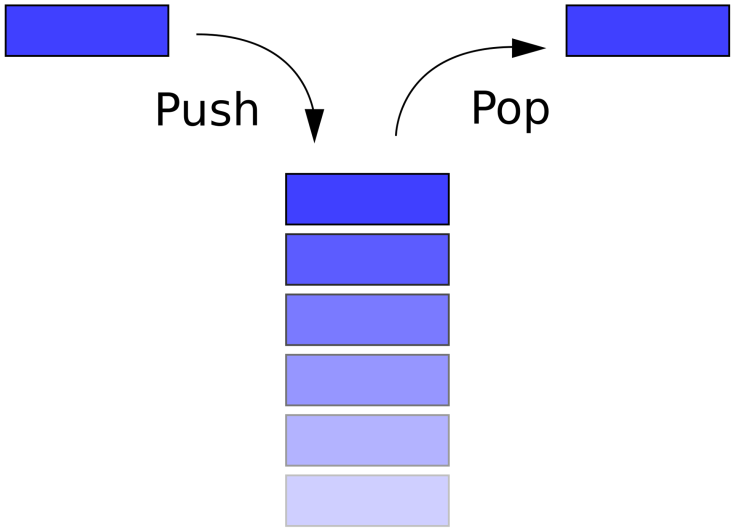



Double linked-list

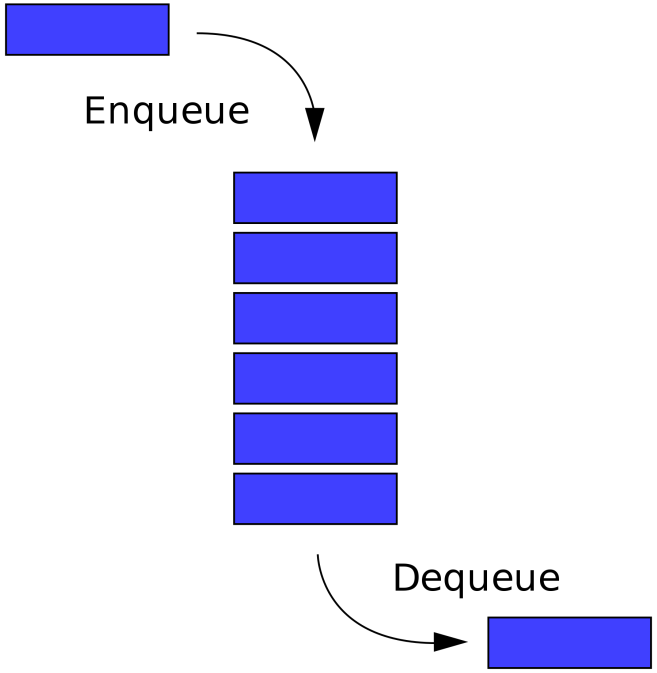


# Sequential containers

queue or LIFO (last in first out)

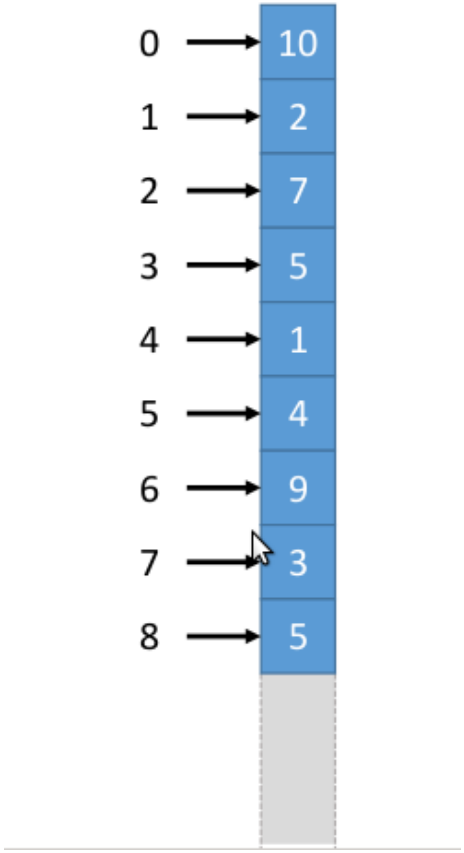



Row o FIFO (first in first out)

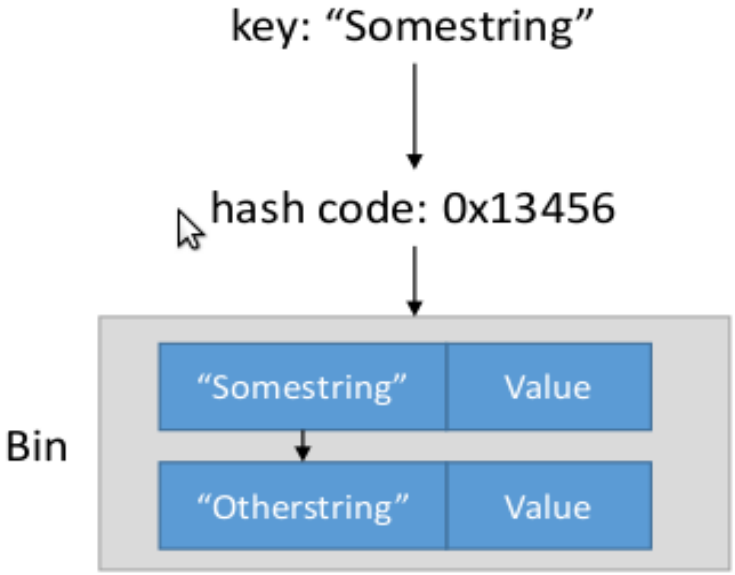


# Associative containers

Arrays




Hash maps




+ dictionaries, hash maps, sets

# Regarding hardware optimization

How to make our calculations faster?

- Write and read faster (I/O veloc).
- Turn the handle faster (¿ clock veloc).
- Improve technology(better CPU).
- Limit access to non-uniform and multi-core memory.
- Multi-core architectures. Parallelazing.
- Compilers that allow optimization to take advantage of the CPU (pipelining, vector and superscalar units , etc)

WTPC workshop en técnicas de programación científica

# Regarding hardware optimization

The type of operations matters !!!

- + - *
- / % sqrt()
- Transcendental functions
- Pow(x,y) x,y real

- Op. faster,0.5x-1x
- Op. average speed,5x-10x
- Op. slowers , 20x-50x
- Op. expensive, very slow,+100x

Pipelining → fast operations.
To use for instance BLAS-LAPACK (álgebra lineal, son + y *)

# Conclusions

- Look carefully the trade off between time to use and development time.
- Scientific libraries with a lot of examples.
- Consider Possibility of re-use and improve existing code.

WTPC workshop en técnicas de programación científica