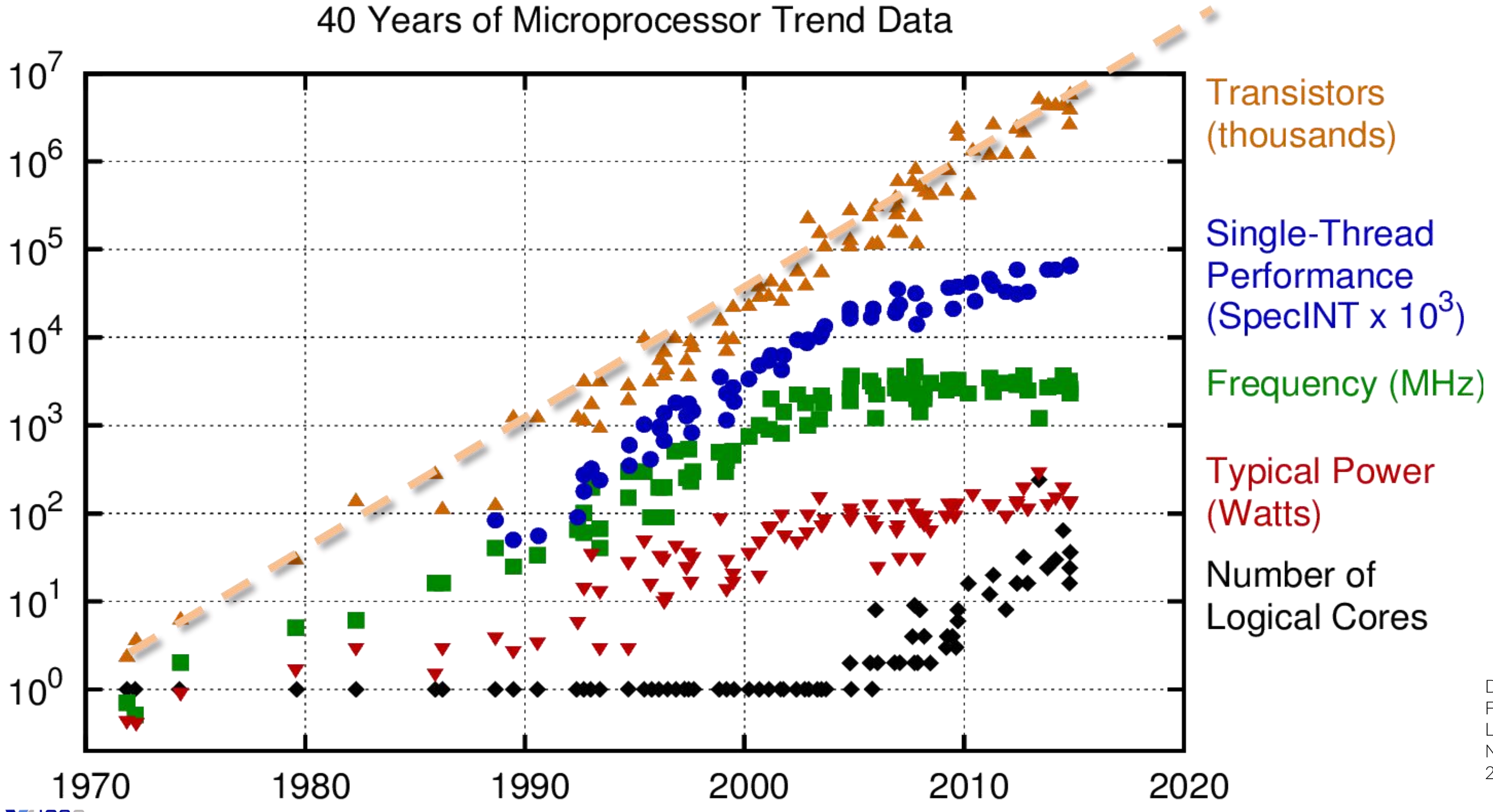
Introduction to Modern CPU architecture

Luca Tornatore, I.N.A.F.

2024 INAF Course on HPC

June, 24th - July 5th, IRA, Bologna

Moore's law, is it over ?

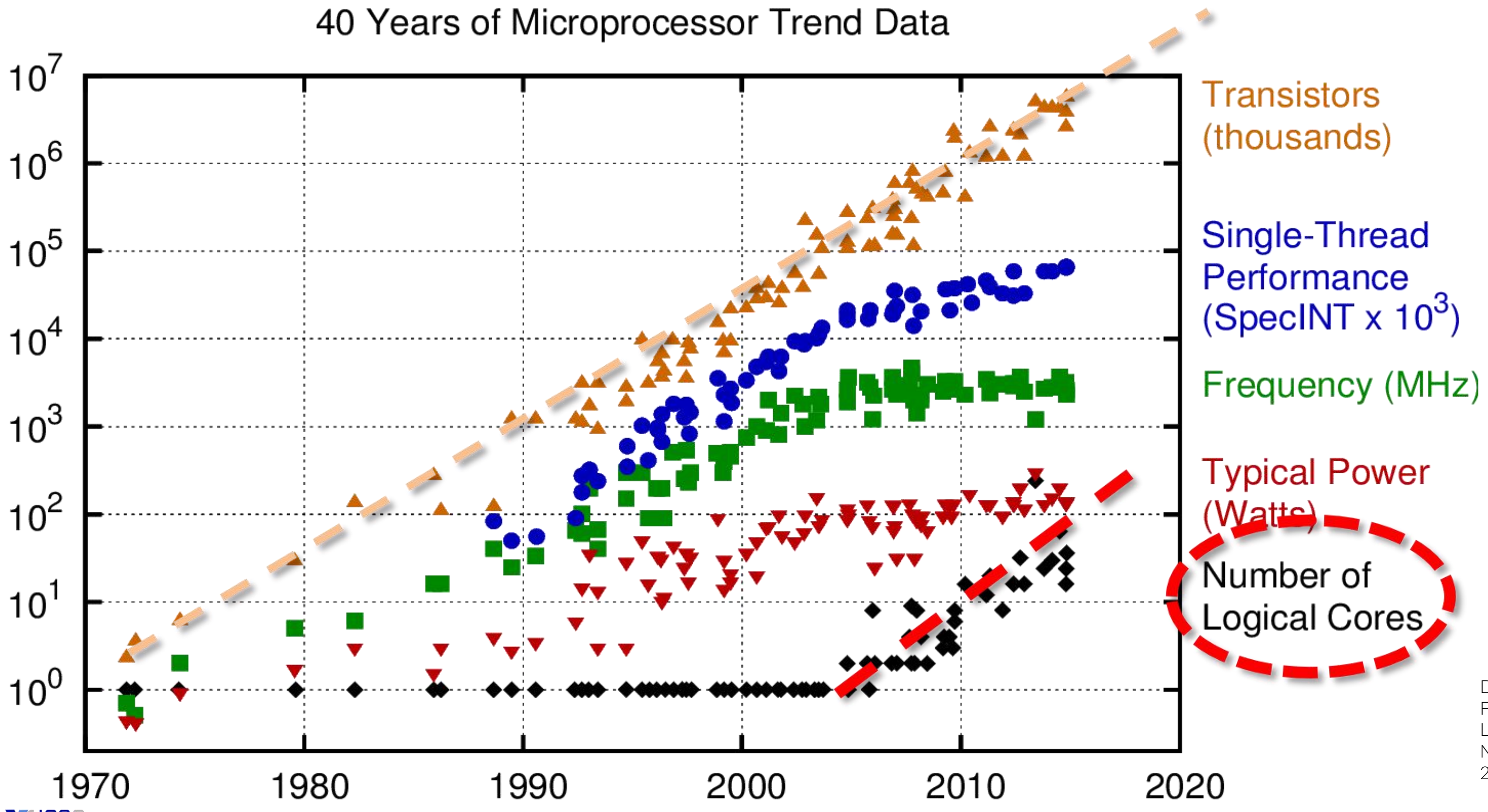


« every 18 months the density of transistors will double *at the same manufacturing cost* »

It is still in place, it seems

Data collected up to 2010 by M.Horowitz, F. Labonte, O. Schacham, K. Olukotun, L.Hammond and C. Batlen.
New data up to 2015 collected by 2010-2015 by K. Rupp.

Moore's law, is it over ?



« every 18 months the density of transistors will double *at the same manufacturing cost* »

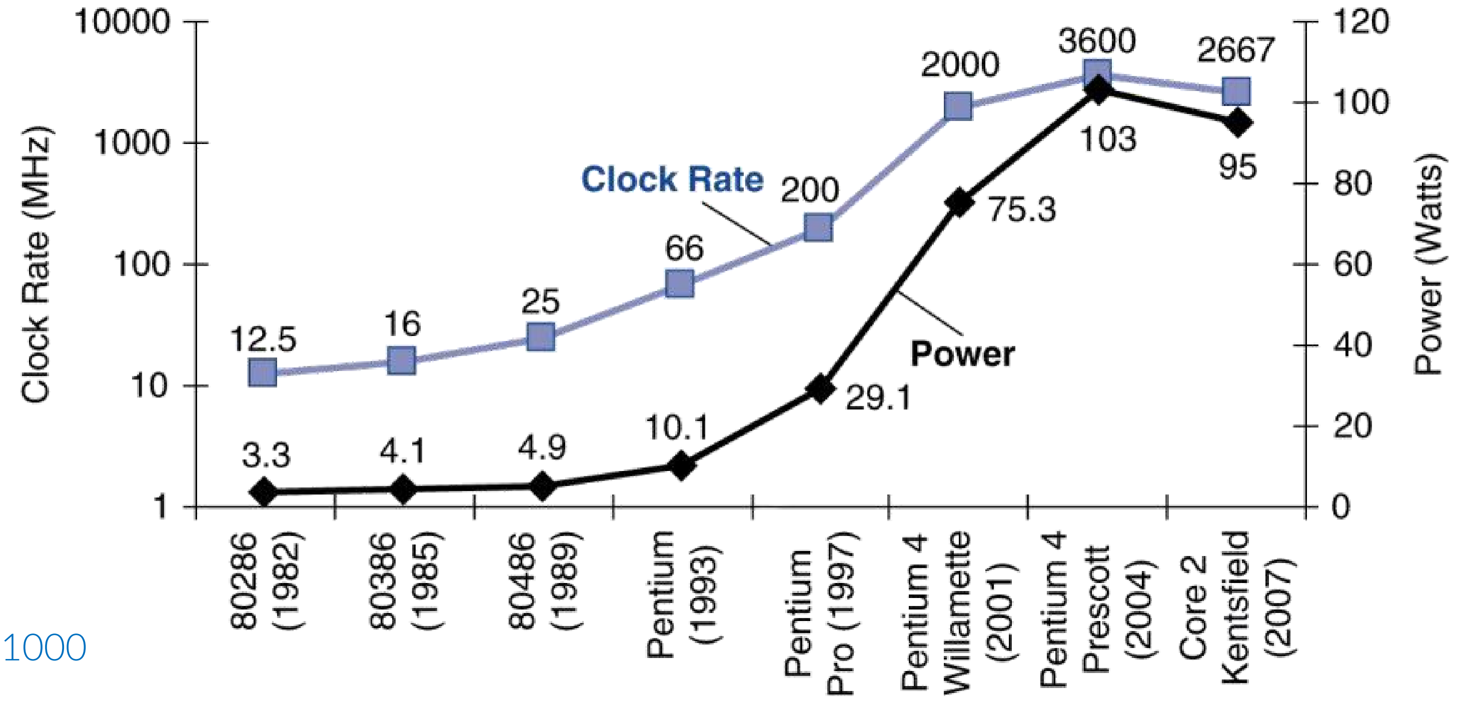
It is still in place, it seems,

with a fundamental factor

Data collected up to 2010 by M.Horowitz, F. Labonte, O. Schacham, K. Olukotun, L.Hammond and C. Batlen.
New data up to 2015 collected by 2010-2015 by K. Rupp.

Why there is no more "free lunch"?

« **free lunch** » is the fact that you do not act on your code; you just wait 18 months and it will run faster



Power consumption:

$$P \propto C \cdot V^2 \cdot f$$

$\times \sim 30$

$5V \rightarrow 1V$

$\times \sim 1000$

The transition from the “free lunch”

Until half of the ‘00s, engineers succeeded in gaining performance by essentially 3 ways:

1. Increasing **clock** speed
2. Optimizing **execution**
3. Enlarging/improving **cache**

The transition from the “free lunch”

Until half of the ‘00s, engineers succeeded in gaining performance by essentially 3 ways:

1. Increasing **clock** speed ⟶
2. Optimizing **execution**
3. Enlarging/improving **cache**

You get more cycles per unit time; more or less that means doing the same bunch of instructions faster

The transition from the "free lunch"

Until half of the '00s, engineers
gaining performance by essent

1. Increasing **clock** speed
2. Optimizing **execution** →
3. Enlarging/improving **cache**

- More powerful instructions
- Pipelining
- Branch predictions
- Out-of-order execution
- Vector instructions

Under enormous pressure, CPUs manufacturers risked (and did) to break the semantic of your code. Or introduce horrible bugs..
(*have you heard about **Meltdown** and **Spectre** ?*)

The transition from the “free lunch”

Until half of the ‘00s, engineers succeeded in gaining performance by essentially 3 ways:

1. Increasing **clock** speed
2. Optimizing **execution**
3. Enlarging/improving **cache** $\rightarrow$ More on that later..

Why there is no more "free lunch"?

Applications no longer get more performance for free without significant re-design, since $\gtrsim$15 years

Since 15 years, the gain in performance is essentially due to **fundamentally different factors**:

1. Multi-core + **Multi-threads**
2. Enlarging/improving **cache**
3. Hyperthreading (*smaller contribution*)

Why there is no more "free lunch"?

For instance:

2 Cores at 3GHz are basically 1 Core at 6GHz.. ?

False

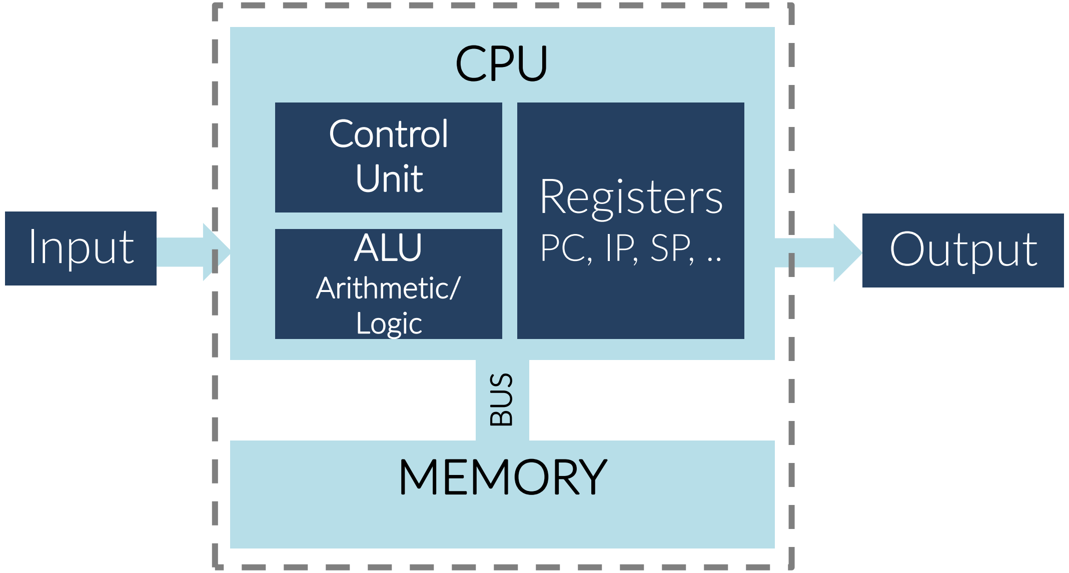
- × Cores coordination for cache-coherence
- × Threads coordination
- × Memory access
- × Increased algorithmic complexity

Since 15 years, the gain in performance is essentially due to **fundamentally different factors**:

1. Multi-core + **Multi-threads**
2. Enlarging/improving **cache**
3. Hyperthreading (*smaller contribution*)

Once upon a time..

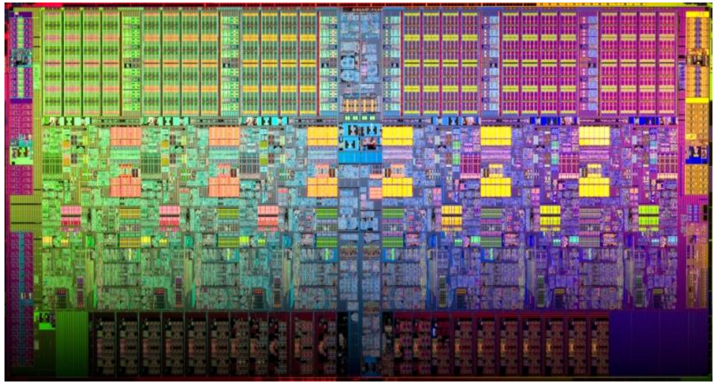
.. The computer architecture was much simpler than today. In the lecture “The Free lunch is over”, we have seen the reasons why it was so – or, better, why nowadays it is much more complex.



In the **Von Neumann architecture**

- there is only 1 processing unit
- 1 instructions is executed at a time
- memory is “flat”:
  - access on any location has always the same cost
  - access to memory has the same cost than op execution

While today…



NOT *an extreme example:* 6-cores Intel Xeon e5600

(∗) "cost" is in terms of the CPU-cycles currency

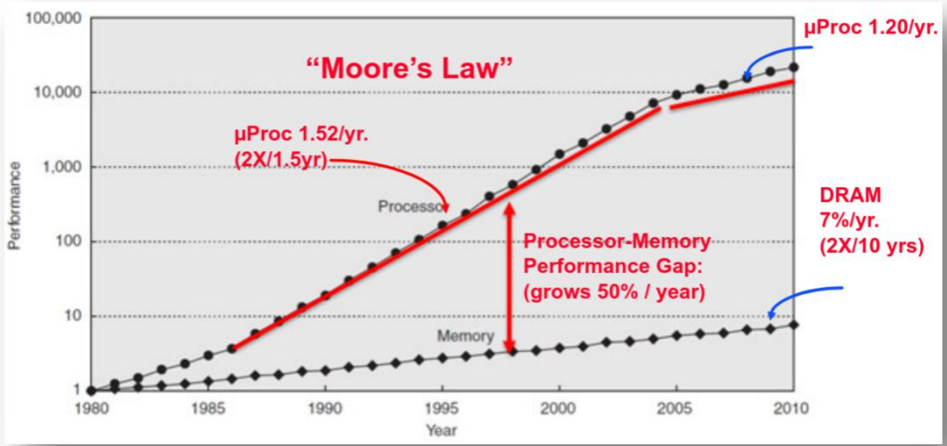
- there are *many* processing units
- many instructions can be executed at a time
- many data can be processed at a time
- "instructions" are internally broken down in many simpler μops that are pipelined
  - different instructions could have quite different cost(∗)
- memory is strongly *not* "flat":
  - there is a strong memory hierarchy
  - access memory can have *very* different costs(∗) depending on the location
  - accessing RAM is *way* more costly than performing a μop or reading an internal register

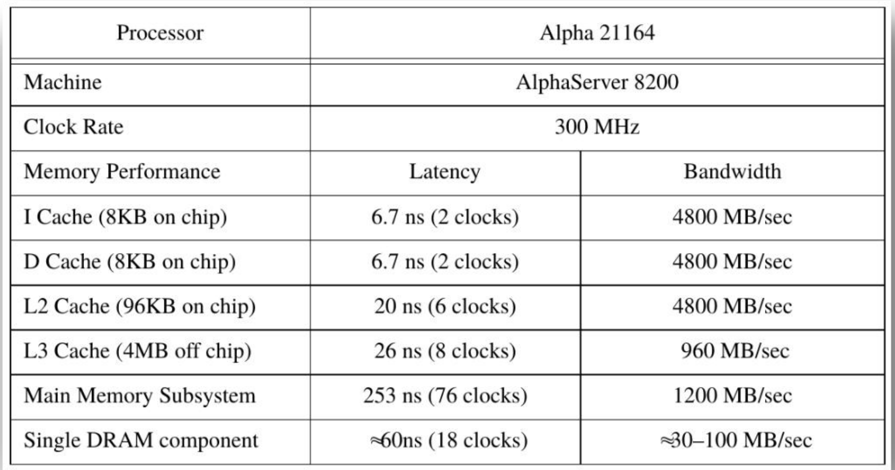
In the next slides we'll go through all this features in chronological order, as they developed in time

Some of the steps in between

- The memory hierarchy
- Superscalability and out-of-order execution
- Pipelining
- Vector capabilities

Early 90s: CPU get faster than memory





Processor	Alpha 21164	
Machine	AlphaServer 8200	
Clock Rate	300 MHz	
Memory Performance	Latency	Bandwidth
I Cache (8KB on chip)	6.7 ns (2 clocks)	4800 MB/sec
D Cache (8KB on chip)	6.7 ns (2 clocks)	4800 MB/sec
L2 Cache (96KB on chip)	20 ns (6 clocks)	4800 MB/sec
L3 Cache (4MB off chip)	26 ns (8 clocks)	960 MB/sec
Main Memory Subsystem	253 ns (76 clocks)	1200 MB/sec
Single DRAM component	≈60ns (18 clocks)	≈30–100 MB/sec

Taken from a 1997 paper

The CPU had to spend more time waiting for data coming from RAM than executing ops.
That is part of the so called "memory wall".
What is the solution ?

A note about today

You certainly know that **DDRAM memory catched up: frequency has increased in the last year up to ≥ 3-4GHz**

However, that alleviate the problem only partially

Figure taken from: "Memory Bandwidth and Latency in HPC: System Requirements and Performance Impact"

Ph.D disserattion thesis, 2019, Dep.t of Computing Architectures, UPC

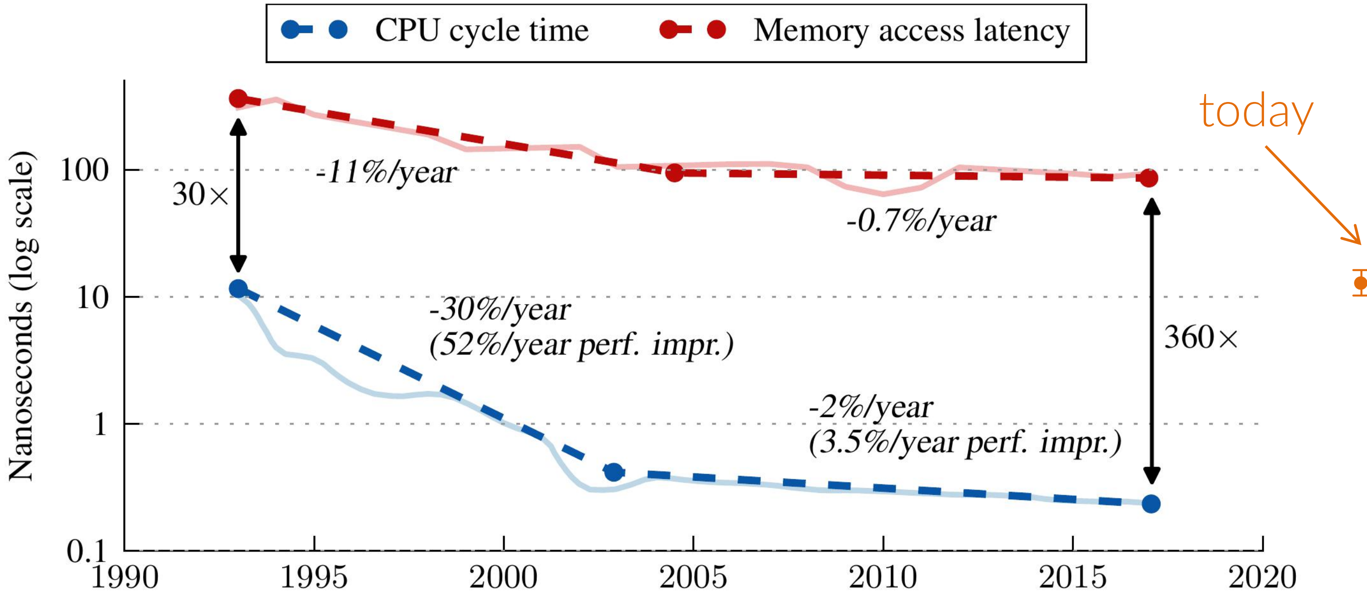


Figure 2.4: Increasing discrepancy between main memory access latency and CPU cycle time, for last 25 years. Recently the downtrend in CPU cycle time decreased to 2%, while single processor performance improves at 3.5% per year [52]. The decrement in memory access latency is less than 1%.

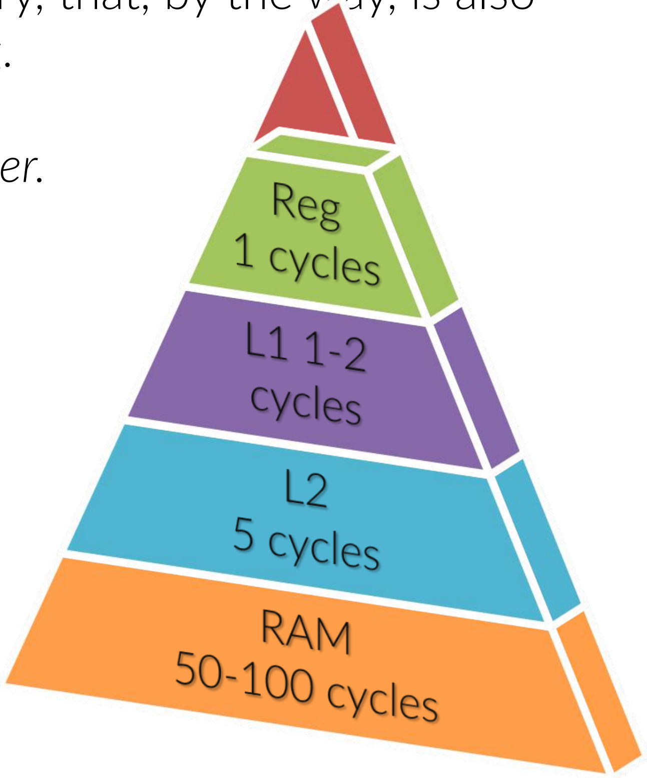
The cache memory

The solution is to equip your CPU with a *faster* memory, that, by the way, is also *way* more expensive and *way* more energy consuming.

Furthermore, to be faster it ought to be *extremely closer*.
All in all, the new memory that will be called ***cache***,
will be much *smaller* than RAM.

The cache itself has a hierarchy:

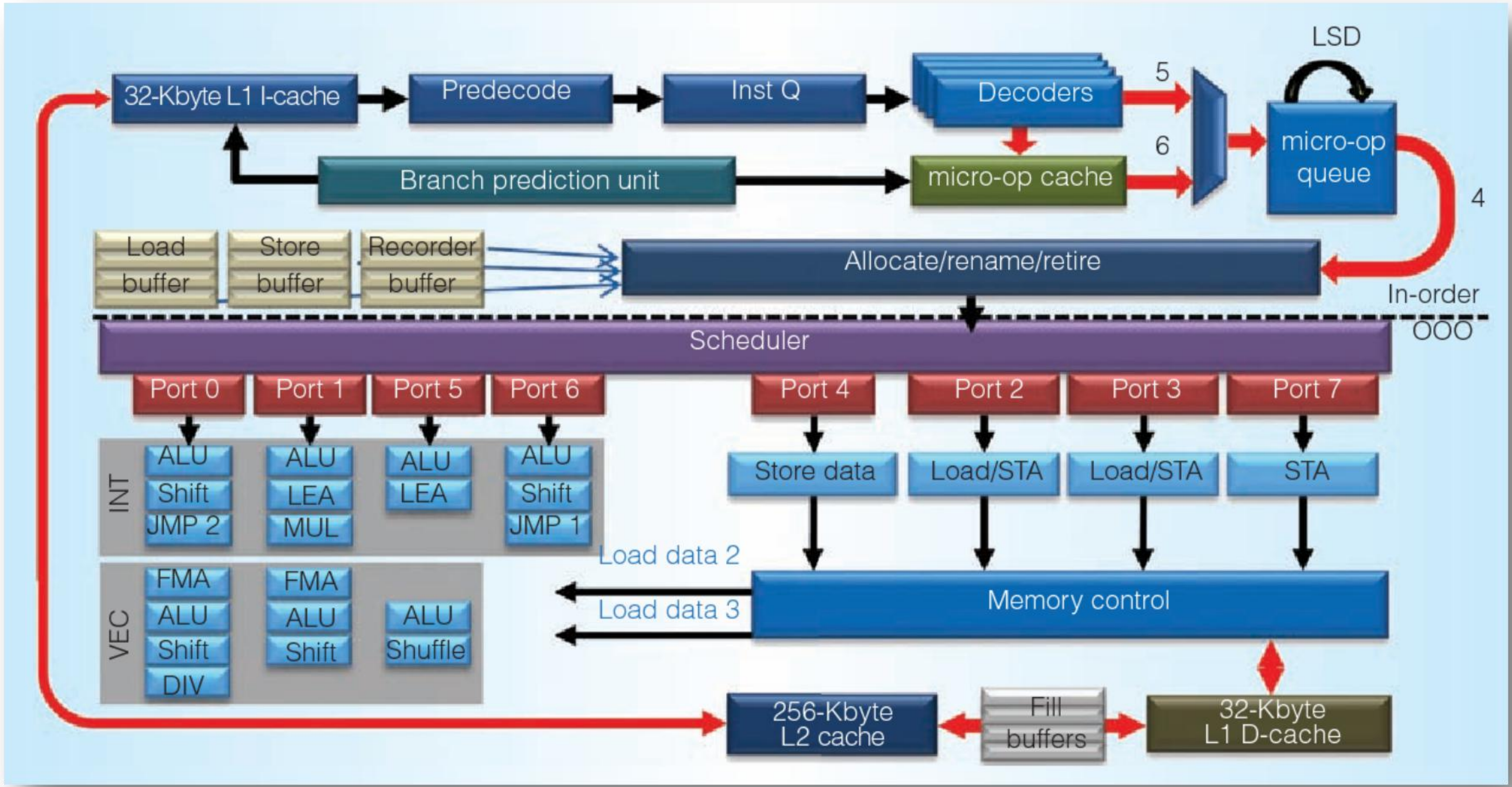
- *Level-I* is inside each core.
- *Level-II* is also inside the core, or
  may be shared by more cores.
- *Level-III* is inside the CPU, shared by many cores.



Some of the steps in between

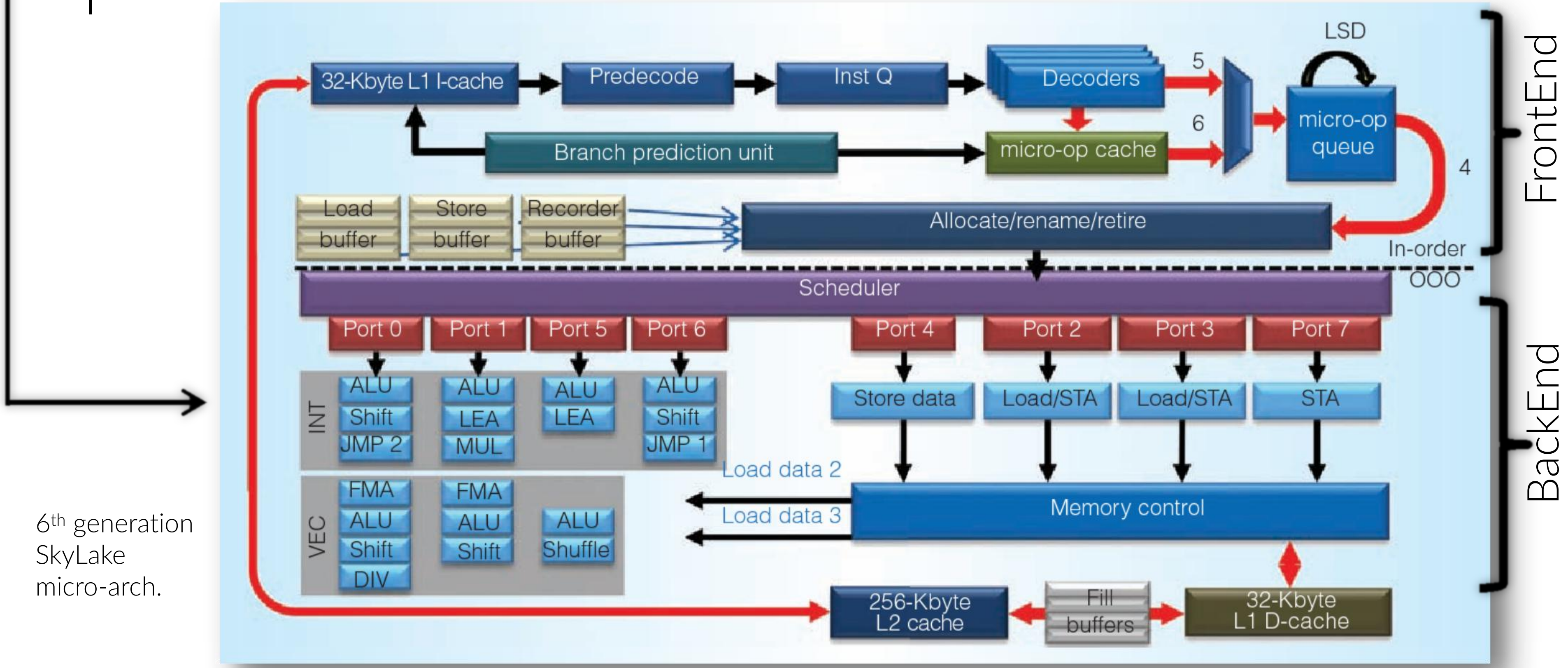
- The memory hierarchy
- Superscalability and out-of-order execution
- Pipelining
- Vector capabilities

Mid 90s: superscalar and out-of-order CPUs



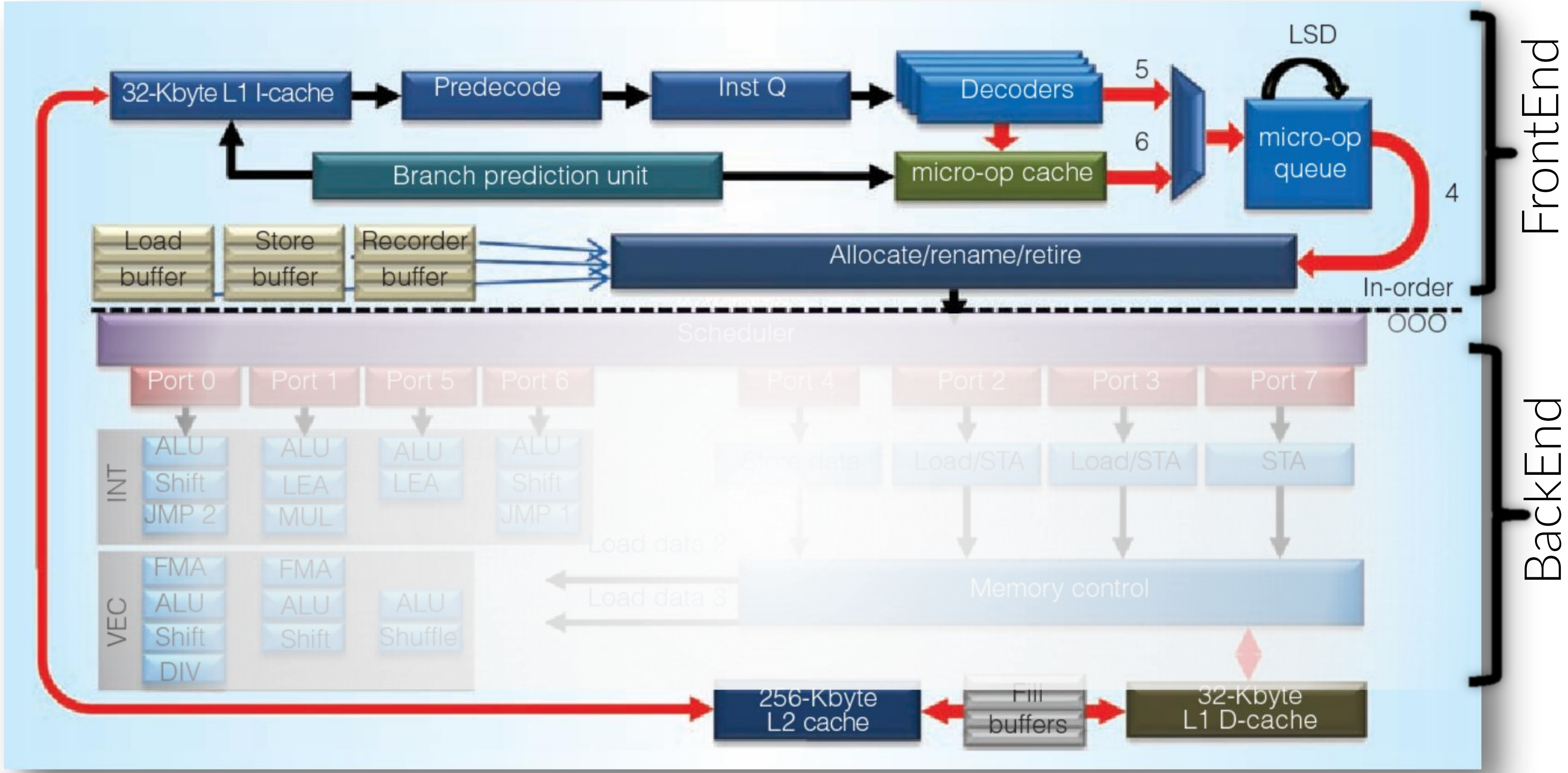
6th generation SkyLake micro-arch.

More than 1 *port* is available to execute CPU instructions, although different units have different specializations (ALU, LEA, SHIFT, FMA, …) : that is superscalar capacity, i.e. the capacity of executing more than 1 instructions per cycle.



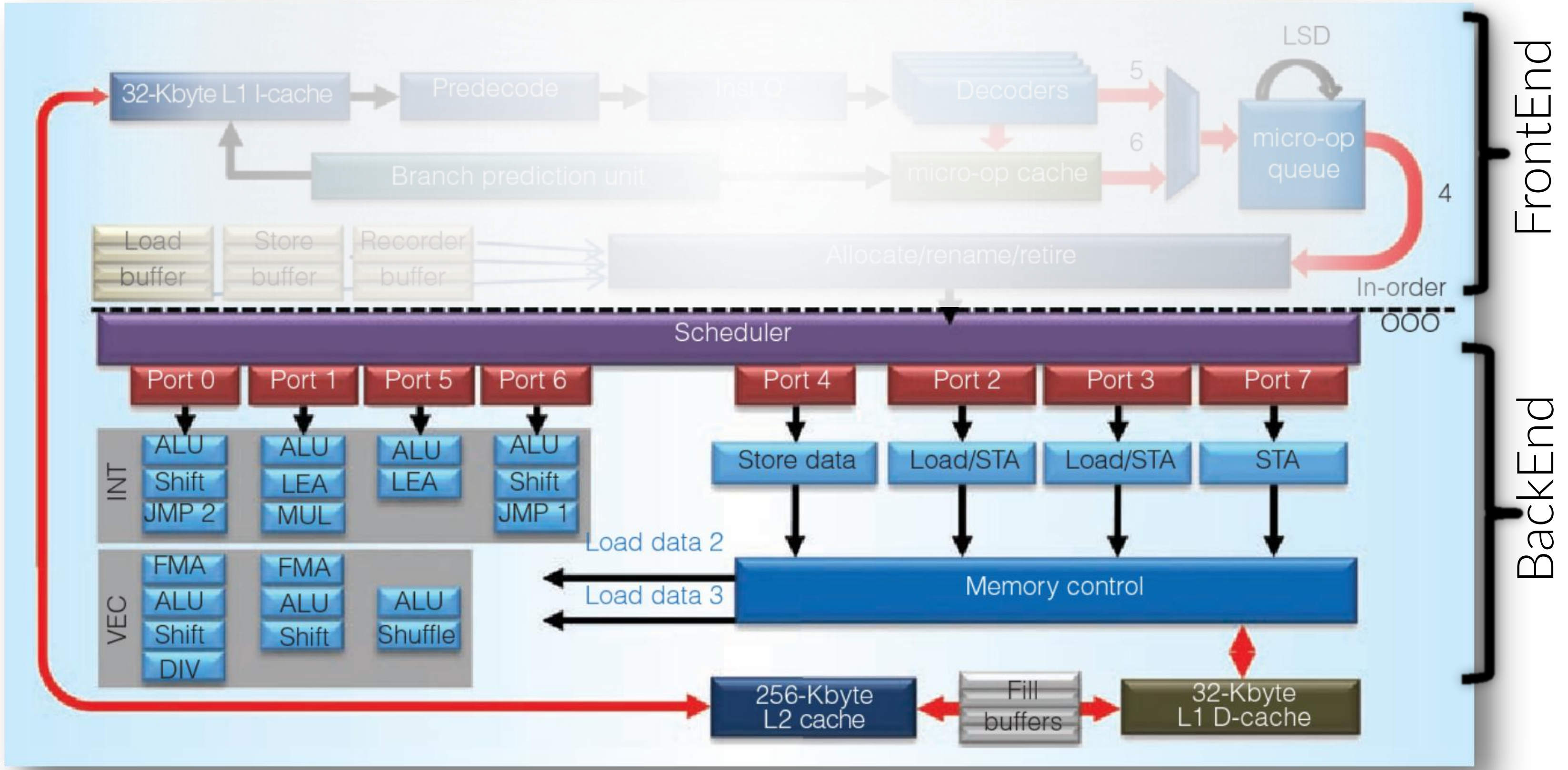
6th generation SkyLake micro-arch.

The *Front-End* basically fetches instructions and the data they operate on from instruction and data caches, decodes instructions, predicts branches and dispatches the instructions to different ports



6th generation SkyLake micro-arch.

The *Back-End* is responsible for the actual instructions execution and for the back-writing of results in memory locations. It is responsible also for orchestrating out-of-order ops execution depending on their instructions/data dependencies.



6th generation SkyLake micro-arch.

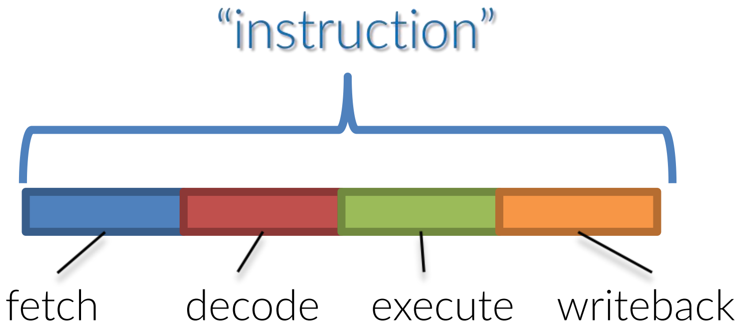
Some of the steps in between

- The memory hierarchy
- Superscalability and out-of-order execution
- Pipelining
- Vector capabilities

Pipelines

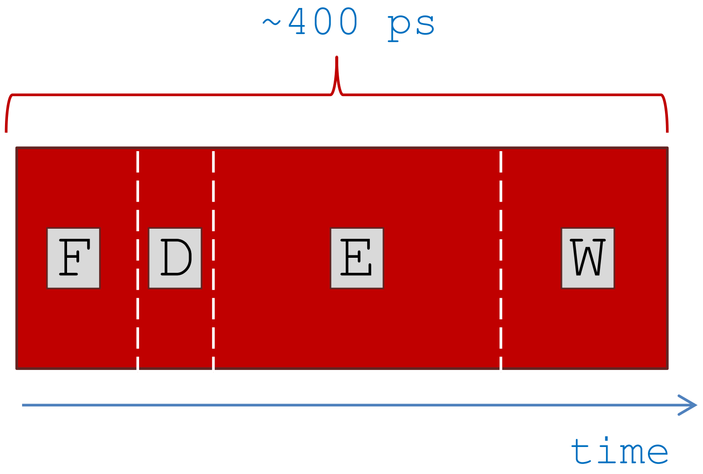
It would be obvious to think that an "instruction" is a kind of *atomic* operation that the CPU perform as a whole. Indeed that was true until the mid of the 80s.

If you think carefully about it, it is easy to understand that actually an "instruction" involves at least the following ***independent*** steps:



1. ***Fetching***
   it must be recalled from memory/Icache
2. ***Deconding:***
   it must be "understood and interpreted"
3. ***Execution***
4. ***Writeback :***
   the result must be accounted in memory ()

Pipelines

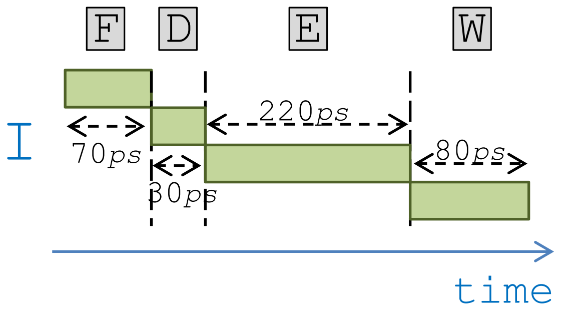


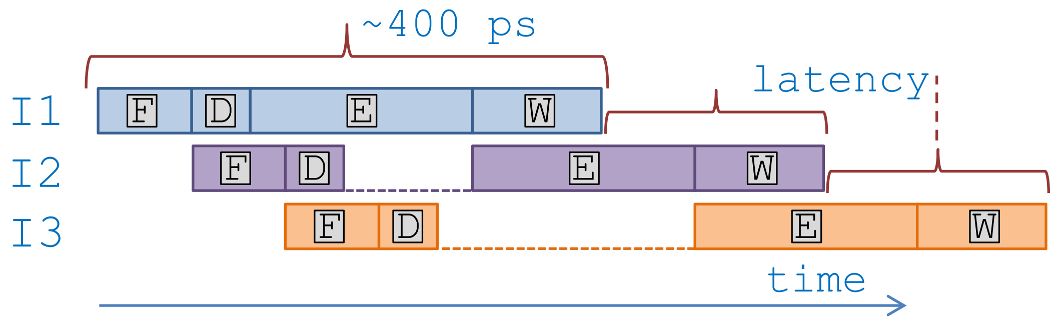
If all the four stages take ~400ps, we then would obtain a **throughput** of 2.5GIPS (giga-instructions per second).
400ps is also the total time required to get a result from an instruction, and so it is the **latency** of the instruction.

However, if we were able to “detach” the four stages, we could organize things differently, like in a car-building chain, or even at the mensa of the university.

Pipelines

Note: all the timing estimates are hypothetical for the purpose of discussing the concepts



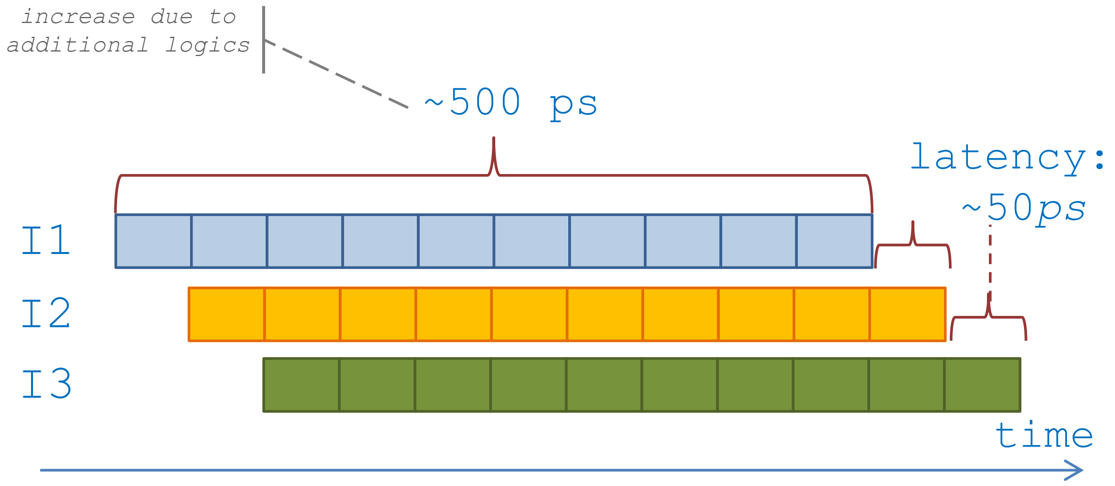


If many independent logical units exist to perform each step, they could operate subsequently on **different instructions**:

If the stage delays are not uniform, the throughput is limited by the latency `F+(D+E)-(F+D) = E` ~ 220*ps*, which means we have a throughput of ~**4.5GIPS** just because of logic units separation.

Pipelines

Therefore, introducing the instructions pipelining, we can increase the **throughput** of our system by a large factor.
However, the efficiency of the pipelines is limited by its longest stage: the better option would be to have all equal stages, for instance further subdividing each stage – especially the most demanding ones $^{(*)}$.



Now the throughput of our system has increased to 1 instruction retired every 50*ps*, i.e. **20**GIPS

$^{(*)}$ this is called *superpipelining*: modern CPUs may have 10-20 stages per pipeline.

Pipelines are about throughput

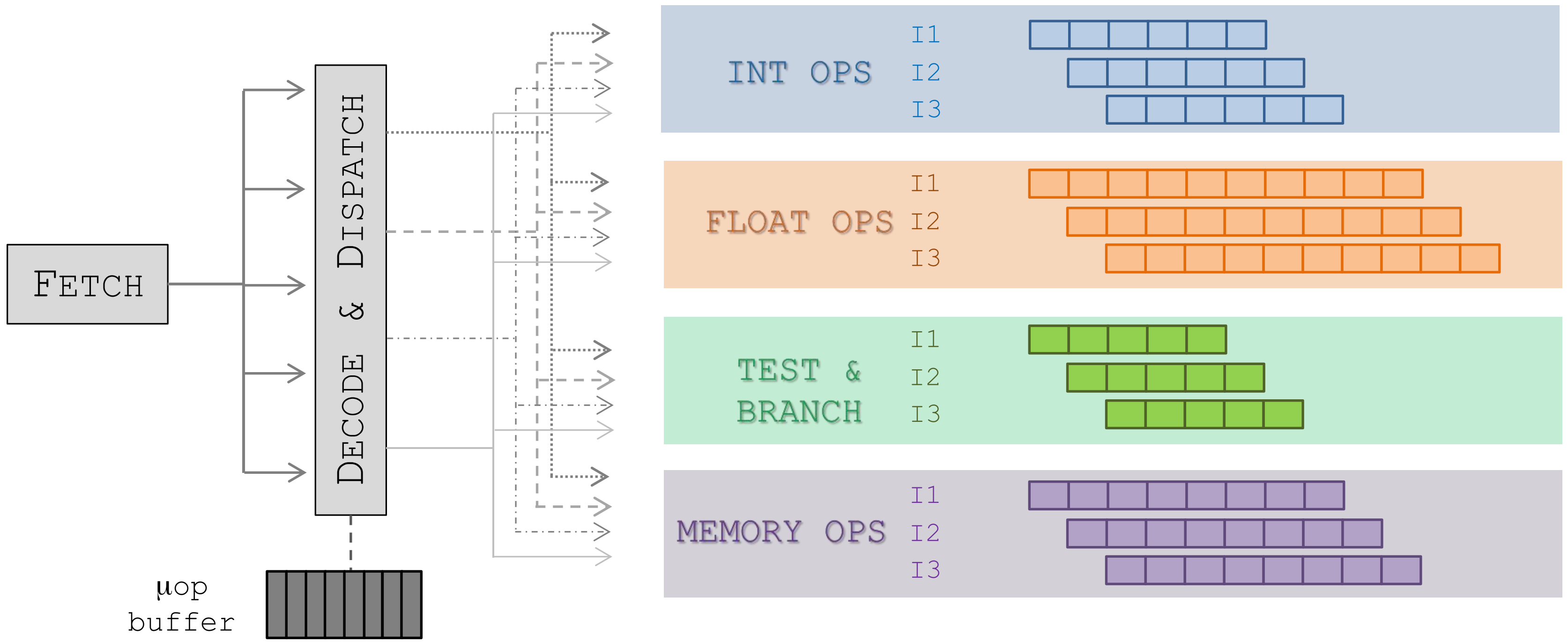
Pipelining is then about increasing the throughput of a system, and as we have seen it could be really effective.

However, there is more that can be done working on the **execution stage** that, of course, encompasses a large number of different *functional units*, performing a **different and independent tasks**.

With an enhancement of the decode/dispatch stage, we can address multiple pipelines that can be active at the same time:

DISCLAIMER: *of course things are more complicated than depicted here, and we do not get* 10^{10} *GIPS.*

Multiple pipelines

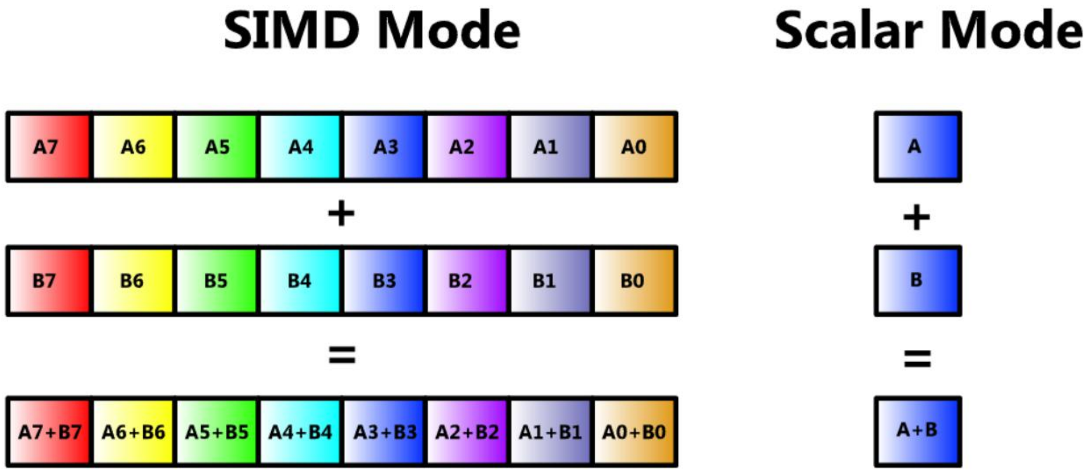


Some of the steps in between

- The memory hierarchy
- Superscalability and out-of-order execution
- Pipelining
- Vector capabilities

Late 90s: vector capabilities become mainstream

https://software.intel.com/sites/default/files/m/d/4/1/d/8/Intro_to_Intel_AVX.pdf

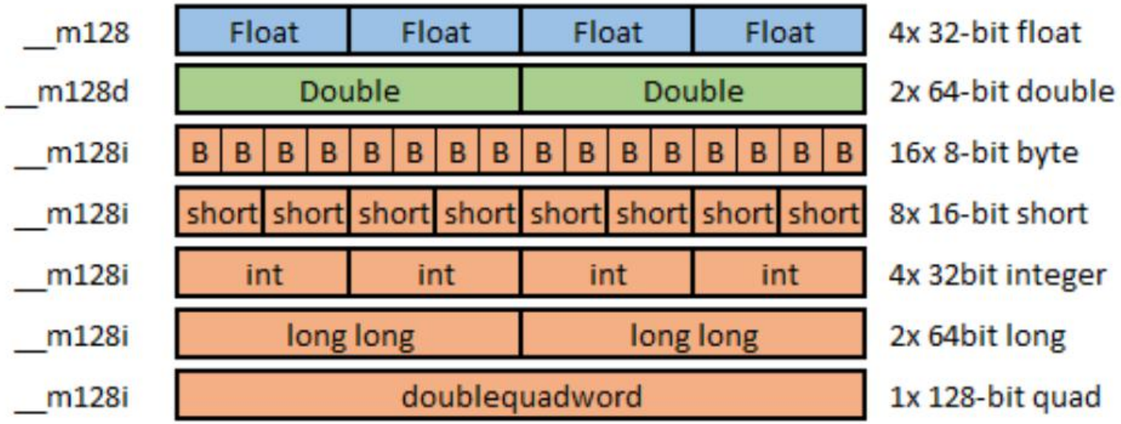


Vector registers are large special registers in the CPU that can be considered subdivided in smaller independent chunks over which the same operation can be performed:

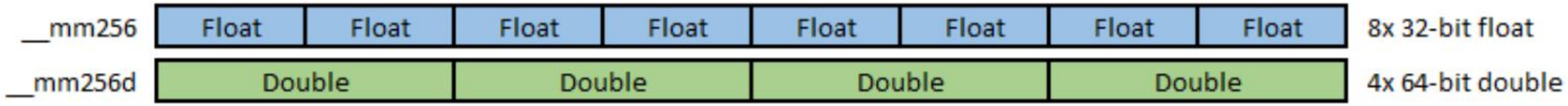
SIMD: Single Instruction
Multiple Data

Vector registers size

SSE Data Types (16 XMM Registers)


__m128 Float Float Float Float 4x 32-bit float
__m128d Double Double 2x 64-bit double
__m128i B B B B B B B B B B B B B B B B 16x 8-bit byte
__m128i short short short short short short short short 8x 16-bit short
__m128i int int int int 4x 32bit integer
__m128i long long long long 2x 64bit long
__m128i doublequadword 1x 128-bit quad

AVX Data Types (16 YMM Registers)


__mm256 Float Float Float Float Float Float Float Float 8x 32-bit float
__mm256d Double Double Double Double 4x 64-bit double

__mm256i *256-bit Integer registers. It behaves similarly to __m128i.Out of scope in AVX, useful on AVX2*

Vector registers size

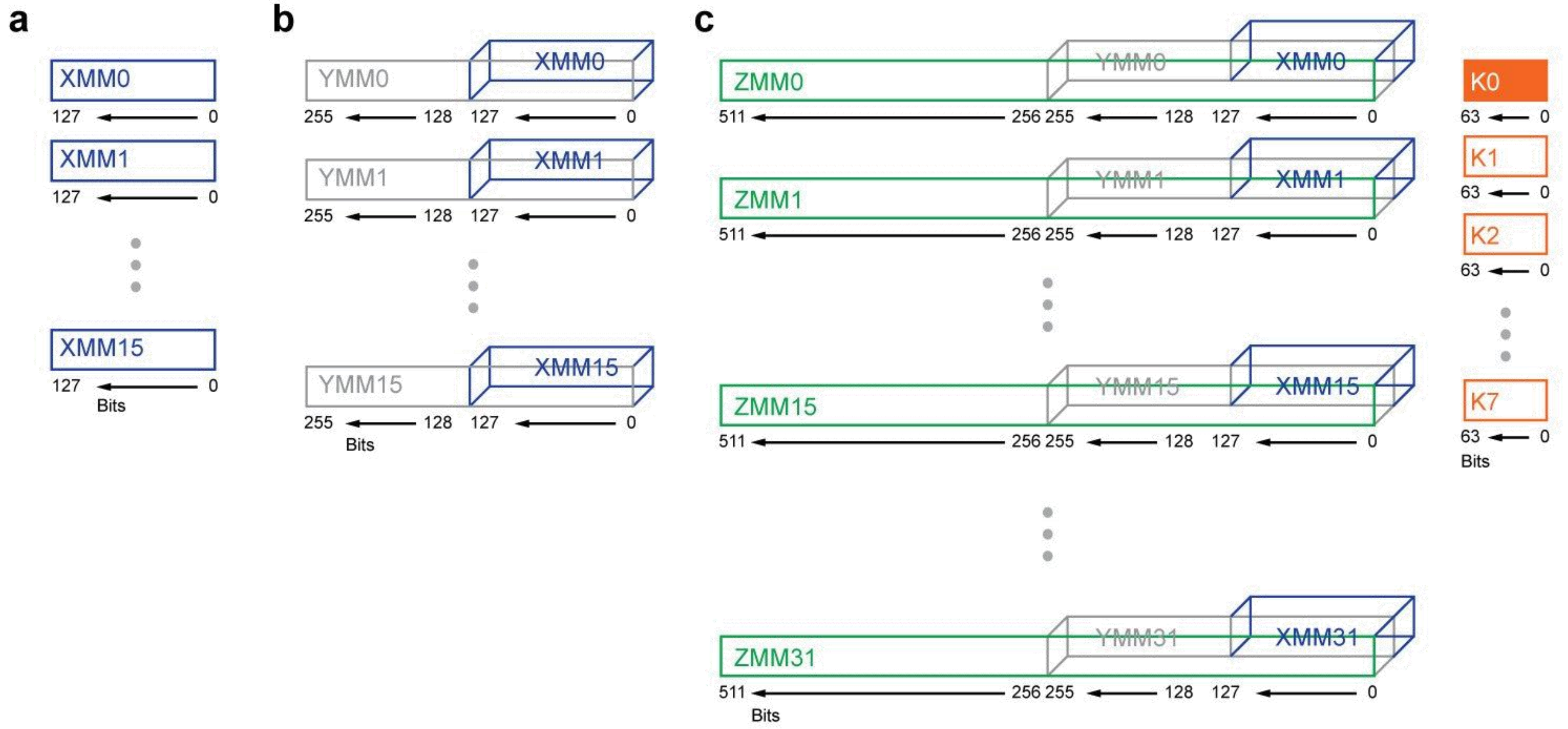
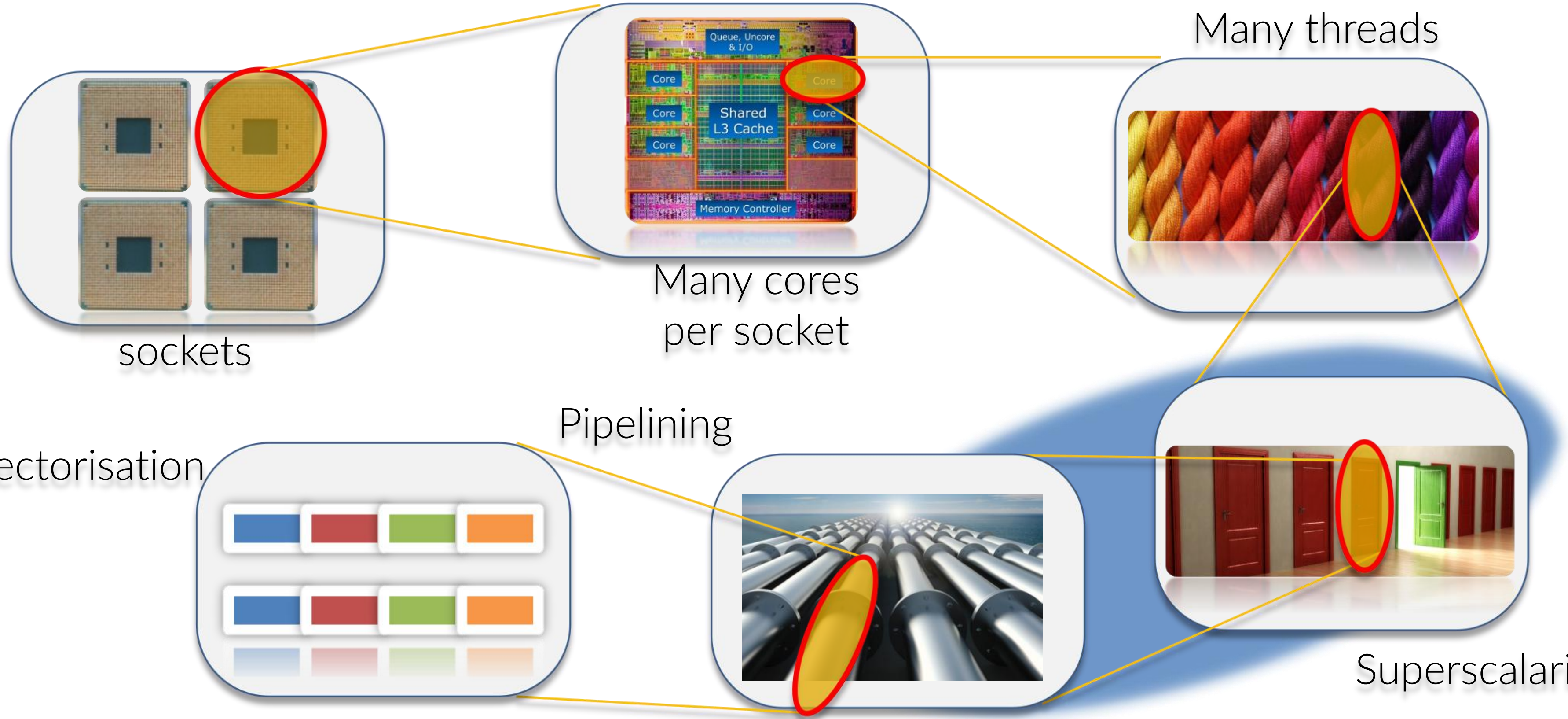


Figure by Marcus D. R. Klarqvist, taken from X

A final sketch



Many threads

sockets

Many cores per socket

Pipelining

Vectorisation

Superscalarity

that's all, have fun