Some details on Modern Architectures

Luca Tornatore, I.N.A.F.

2025 INAF Course on HPC

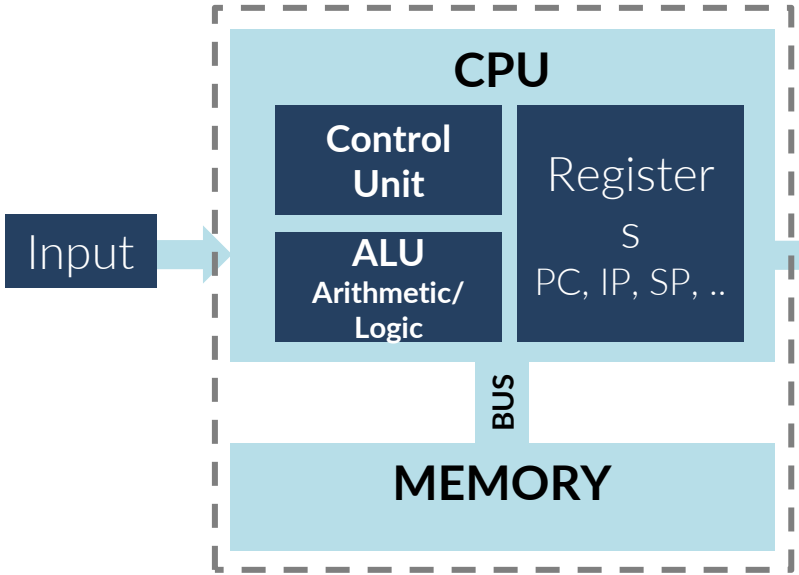
September, 22nd - 26th, OACT, Catania

This lecture presents, *in very condensed way*, the fundamental traits of the **single-core** modern architecture, in their historical developments and roots.

The aim is to convince you that knowing it is important in order to write efficient codes.

Once upon a time..

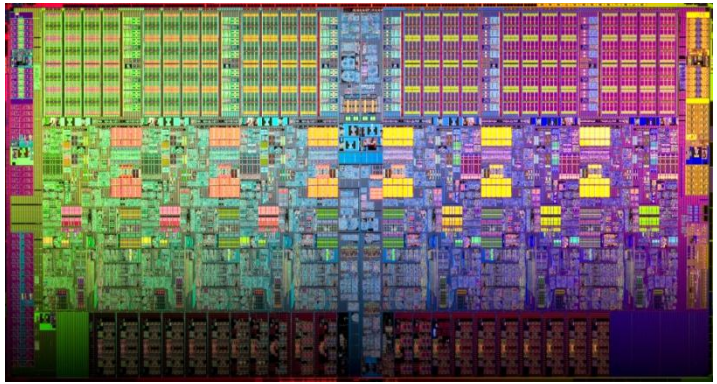
The computer architecture was much simpler than today. Well, if you really think that the wonderful abstraction known as *Von Neumann architecture*, which brought order into chaos, is “simple”



In the **Von Neumann architecture** (still taught as the fundamental model)

- **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

Look back in time

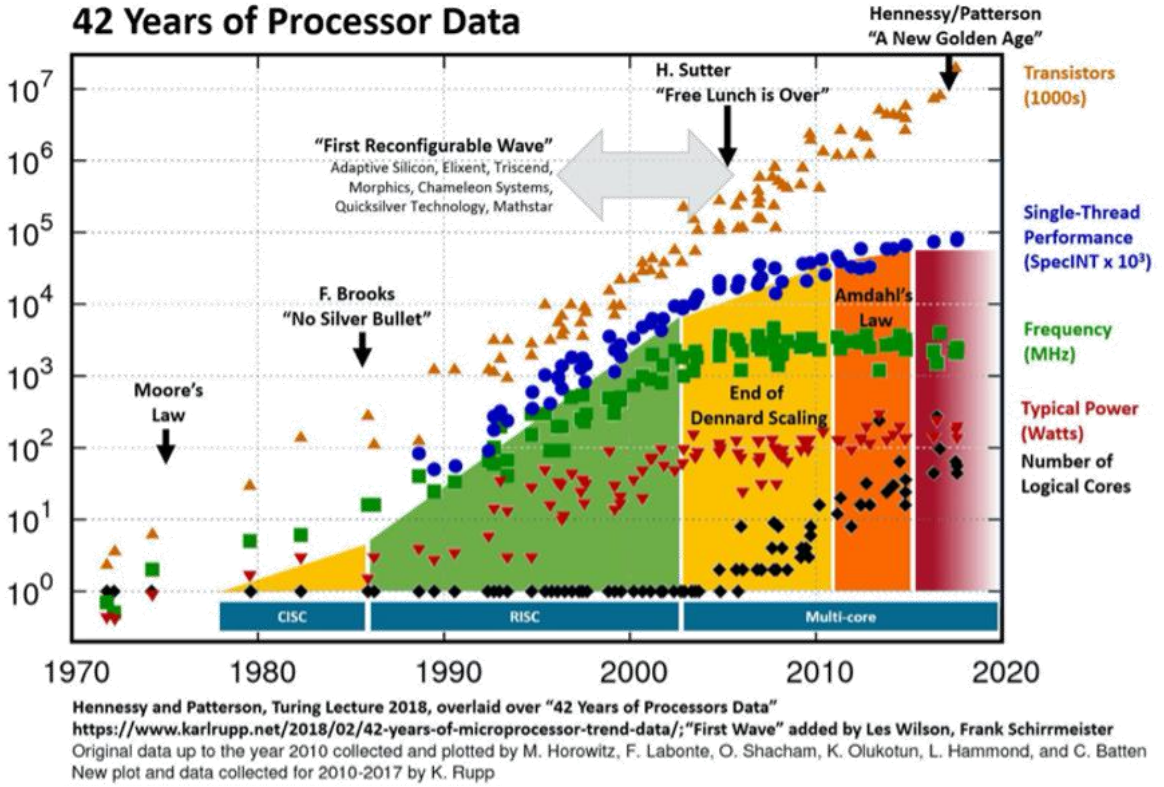
The so-called “Moore’s law” has been there since more than 50 years:
every 18 months the number of transistor per unit ares doubles (*), *at the same manufactoring cost*.

meaning that every 18 months you could buy a commodity CPUs with 2× logics than the previous generation, at the same cost.

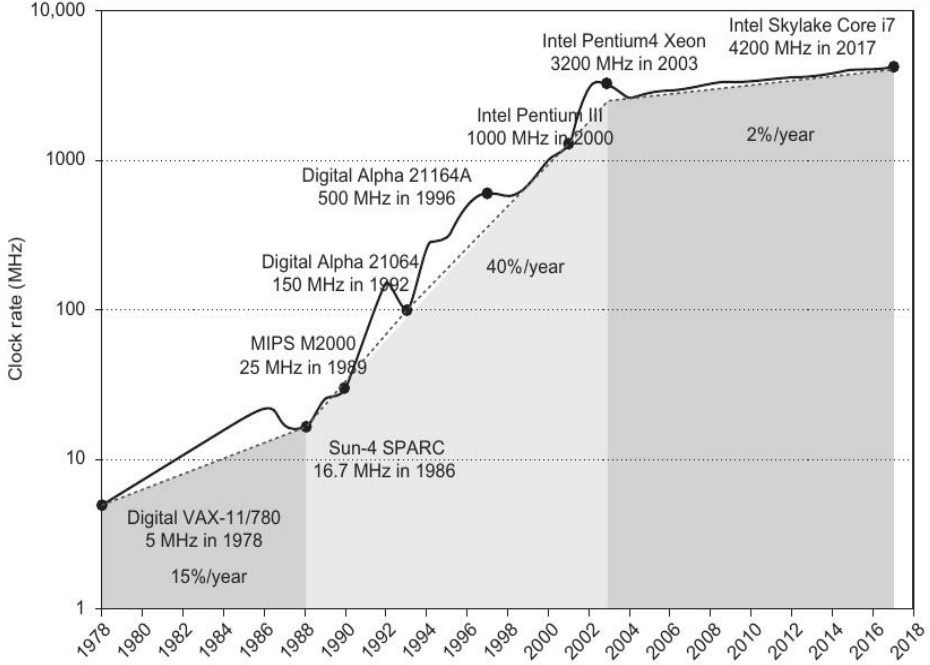
Basically that means more complex circuits, and enhanced capabilities and *higher frequencies*: all together, the code execution was faster.
That has been called the “free lunch era”.

However, every growth comes to and end, especially the exponential ones.

(*)*Note: there have been even faster growths in other fields, for instance in data storage density.*



Increase of clock frequency



From Hernessy & Patterson, Computer Architecture - A quantitative approach, 6th Ed, 2017

The power required per transistor is

$$C \times V^2 \times f$$

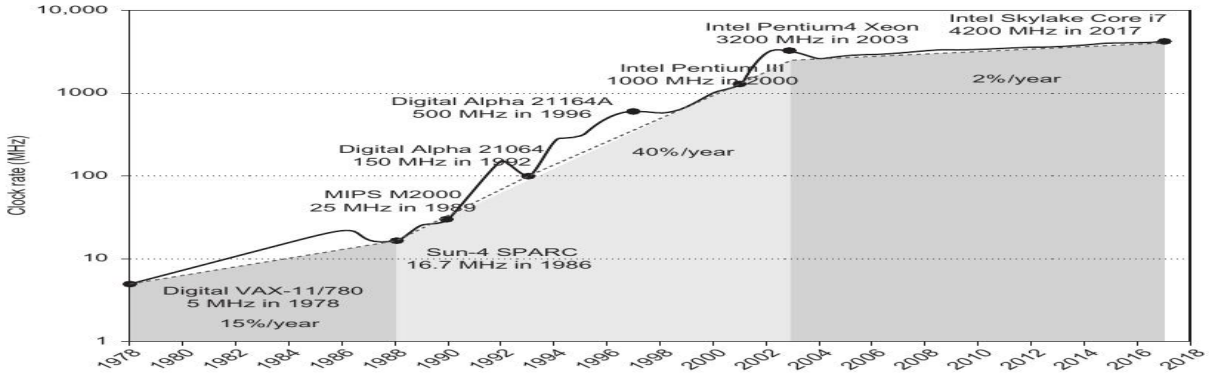
Roughly the capacitance and the voltage of transistor shrinks with the feature size, whereas the scaling is much more complicated for the wires.

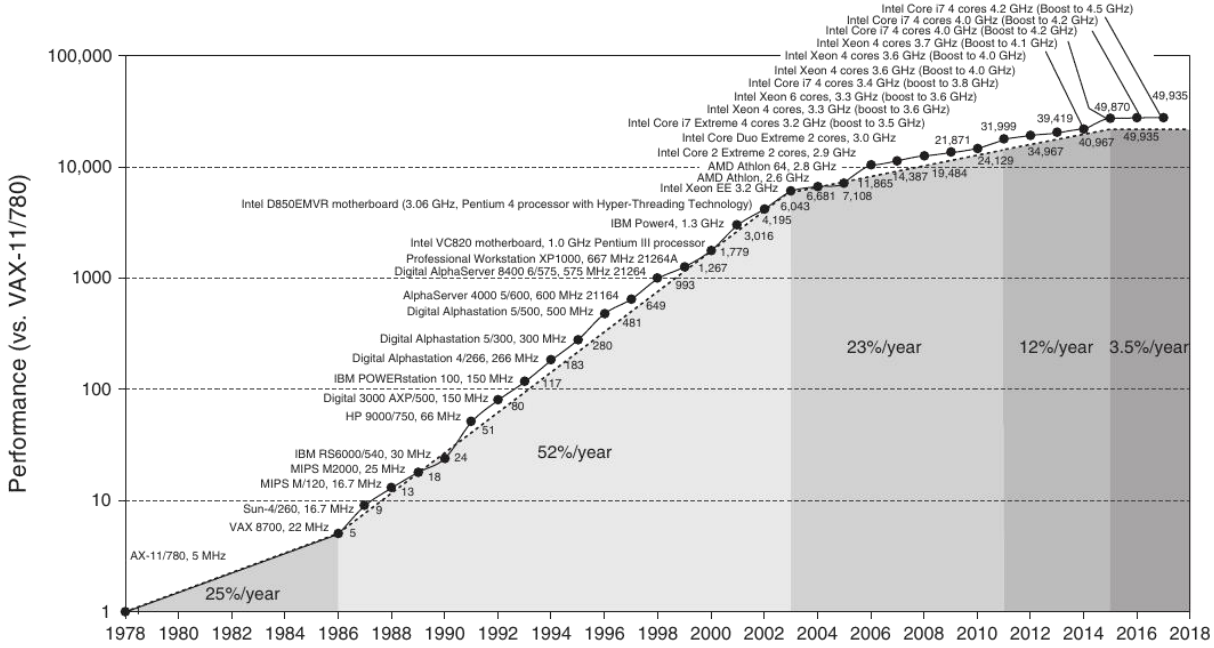
That is basically addressed as the breaking of the Dennard's scaling.

Overall, a typical CPU got from ~2W power to ~100W which is at the limit of the air cooling capacity.

Relating the clock growth and the performance growth

(“performance” here is andicator drawn from some standard arithmetic test)





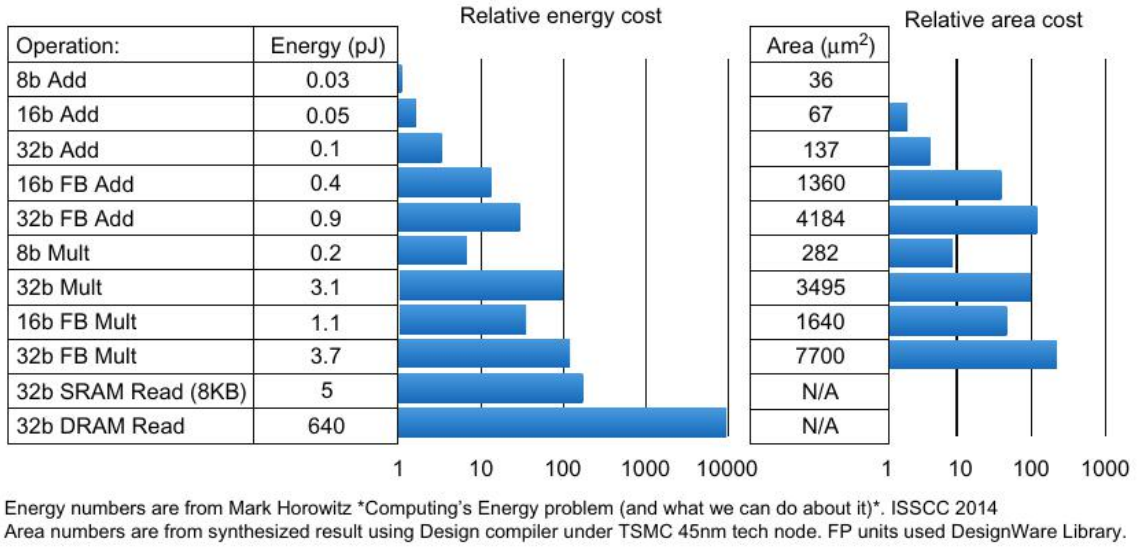
From Hennessy & Patterson, Computer Architecture - A quantitative approach, 6th Ed, 2017

The energy and power are a major design issue

Primary strategies to cope with the *power wall*:

1. Turn off inactive circuits (*dark silicon*)
2. Downscale Voltage and Frequency for both cores and DRAM
3. Thermal Power Design, or design for typical case
4. Overclocking for a short period of time and possibly for just a fraction of the chip

Static power, $P_{static} \propto Current_{static} \times V$, is also an important factor in the power consumption. It increases with the number of transistor while the current leakage increases as the feature size decreases and could amount to ~25% of the total power consumption.



Operation:	Energy (pJ)	Area (µm²)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FB Add	0.4	1360
32b FB Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FB Mult	1.1	1640
32b FB Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A

Energy numbers are from Mark Horowitz *Computing's Energy problem (and what we can do about it)*. ISSCC 2014
Area numbers are from synthesized result using Design compiler under TSMC 45nm tech node. FP units used DesignWare Library.

From Hernessy & Patterson, Computer Architecture - A quantitative approach, 6th Ed, 2017

The “free lunch”

During a couple of decays, we have relied on the Moore’s law and on the Dennard’s scaling to have

1) more *capable* CPUs

2) *faster* CPUs (increasing the frequency, the CPU can execute the same ops at a faster pace)

The burden of performance was basically on the hardware side.

Then, something happened.

The breaking of Dennard's scaling

The **Dennard's scaling** explained to physicists:

Dennard (1974) described **constant-field scaling**. If you shrink every linear dimension by a factor $\kappa>1$ (so new length $L'=L/\kappa$) and also scale supply and threshold voltages by $1/\kappa$, while upping doping appropriately, then:

- Electric fields stay roughly constant (since $E \sim V/L$ and both scale by $1/\kappa$).
- Capacitance per gate scales like $C \propto \varepsilon A/t \sim 1/\kappa$.
- Delay scales down ($\tau \sim RC \propto 1/\kappa$), so frequency goes up ($f \propto \kappa$).
- Dynamic power per gate $P_{dyn} \sim CV^2 f \propto (1/\kappa)(1/\kappa^2)(\kappa) = 1/\kappa^2$.
- Transistor density rises as κ^2.
  Multiplying density by per-gate power gives power density ≈ constant.

In a motto: **smaller & faster at constant heat.**

The breaking of Dennard's scaling

In a motto: **smaller & faster at constant heat.**

Wow.

The breaking of Dennard's scaling

Why **Dennard's scaling broke in the mid of 2000s**

1. **Voltage stopped scaling.**

   You can't keep dropping V_{DD} once V_T (threshold) stops tracking it. The subthreshold slope at room temp bottoms near ~60 mV/dec for an ideal MOSFET; pushing V_T lower blows up subthreshold leakage. With V_{DD} stuck ~1 V, the key energy term CV^2 ceased shrinking.

2. **Leakage currents exploded.**

   As channels thinned and oxides approached a few atomic layers, you got gate tunneling, stronger DIBL (drain-induced barrier lowering), and overall exponential leakage growth. Dynamic power stopped being the only game; static power became a first-order term.

3. **Velocity saturation and mobility limits.**

   At nanometer fields, carriers hit velocity-saturation; you don't get the ideal $1/\kappa$ delay.

4. **Interconnect became a bottleneck.**

   Wires didn't follow device scaling: RC delays worsened as cross-sections shrank; resistivity effectively increases with surface/grain-boundary scattering. Moving charge across the chip became slower and costlier than switching a gate.

5. **Thermal reality.**

   With V pinned but clocks still rising, **power density** rose. That's the **power wall**: chips could no more dissipate the heat, and would have melt at increased frequencies.

The breaking of Dennard's scaling

The impact of the **End of Dennard's scaling.**

A. Frequency plateau
Stuck at ~2GHz since ~20yrs (the so called *"boost frequency"* works for a limited time and for a subset of cores).

B. Multi-core
If you can't push further a single one, put many at work.
$\longrightarrow$ multi-core $\longrightarrow$ wider vector registers $\longrightarrow$ heteregeneous-cores (GPUs/FPGAs/Vector)

C. Device innovations.
In *Geometry* (non-planer, 3D stacking, HBM, to reduce the joule/bit) and *doping*

D. ALGORITHMIC LEVEL
Lots of things. That is exactly the scope of this course.
$\longrightarrow$ the burden of performance has been significantly moved to the programmers' shoulders

A cartoon

50s - 70s	80s - 90s	90s - 2000s	mid 2000s	last 20yrs	next
• Architectures were “simple” • Memory performance was close to cpu • Frequency increase was driving the performance gain	The larger number of transistors/cm^2 permitted more sophisticated logics: • Superscalarity → many functional units executing in a clock • out-of-order execution (*hint: Tomasulo’s algorithm*) • **ILP** starts to be power-hungry	The epoch of **MEMORY WALL**. The CPU ops beats DRAM latenbcy by orders of magnitude. **Note: DRAM GHz buy you bandwidth, the latency is still a problem** • Deep cache hierarchy • TLBs • HW prefetchers • branch predictors • SIMD	The epoch of **POWER WALL** and **MULTICORE.** Many simpler cores instead of a unique monster. TLP brings in issues: • cache coherency • memory models • NUMA	The Cambrian epoch. • More and wider **vectors**! • **Heterogeneous** devices, and **Massively parellel** devices (GPUS, TPUs) • SIMD on CPUs, SIMT on GPUs • HBM, high-speed interconnects (NVlink, Infinity,..), DPUs	**Who knows ?** • **Proximity computation** (compute near data / in memory: Neuromorphic devices?) • **domain-specific** devices and ISA ? • … ?

A small vocabulary

Functional Units	A hardware block inside a CPU core that executes a specific class of operations—e.g., INT/FP math, mem load/store ops, branch for control flow, vector/SIMD for lane-wise ops	NUMA	**N**on-**U**niform **M**emory **A**ccess *we'll see in detail*
out-of-order execution	A CPU strategy that lets micro-ops run as soon as their inputs are ready, instead of strictly following the original program order	cache coherency	How you propagate changes in the content of variables among the caches of different cores
ILP / TLP	**I**nstruction / **T**hread **L**evel **P**arallelism	vector registers	Wide CPU Registers that can operate in chunks, performing the same op on all the chunks
Cache, TLB, Pipelines	*we'll see in the next slides*	HBM	**H**igh-**B**andwidth **M**emory; very expensive RAM built to deliver high bandwidth at the cost of larger latencies
Prefetcher	The logic that fetches data and instructions fomr DRAM before their usage, based on predictions	NVlink	A custom NVIDIA's hardware connection to/among GPUs both on the motherboard and btw nodes
SIMD / SIMT	**S**ingle **I**nstructions **M**ultiple **D**ata/**T**hreads	Infinity Fabric	A high-speed (~100Gb per lane) interconnect among nodes
Branch Predictor	The logic that heuristically predicts whether a conditional will evaluate either true or false *we'll see in the next slides*	Neuromorphic	*I could not fit it here; experimental, qualitatively radically different approach to computation.*

A cartoon

Performance no more dragged by frequency increase

DRAM lags behind:

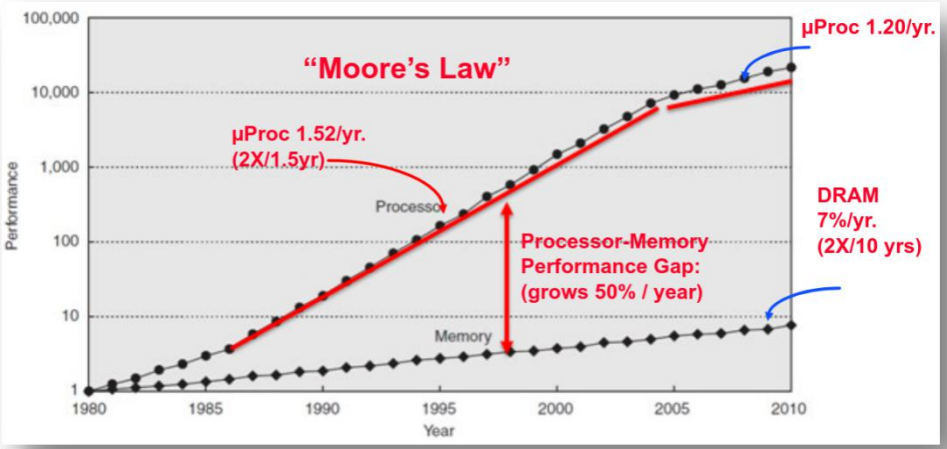
latency vs bandwidth

At a glance

CPUs become faster than memory	Accessing the central DRAM becomes a major bottleneck due to a mucgh larger latency than CPU. If the memory access is not carefully designed the CPU just waits for the data most of the time (*data starvation*). A memory hierarchy is introduced to reduce the impact of this gap, and that introduces the key concept of *data locality*.
CPUs become super-scalar and acquire out-of-order capacilities	A modern core has more than one functional unit that can perform a class of operations (Arithmetic-logic, memory access, I/O, ...). That means that more than one operation can be performed in the same clock, if the code is suited to allow that, which is referred as Instruction-Level-Parallelism (ILP).
Operations are broken down into smaller stages and pipelined	The expected throughput is larger, at the condition that the pipelines are always as full as possible and do not stall due to data and control hazards (we'll see that later)
Branch predictors are an important part of the fron-end	Branches play a major role in causing stalls in the pipelines and in the execution flow. Dealing with this is a key factor to increase the code's performance.
CPUs acquire vector capabilities	Special registers in the CPU have the capacity of performing the same operation on multiple data (SIMD - Same Instruction Multiple Data). That is referred as Data-Level-Parallelism. Not all the loops are vectorizable, that depends on a number of factors.

The Memory

Early 90s: CPU becomes 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 may 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 frequency increased in the last year up to 3-4GHz

However, that alleviate the problem only partially

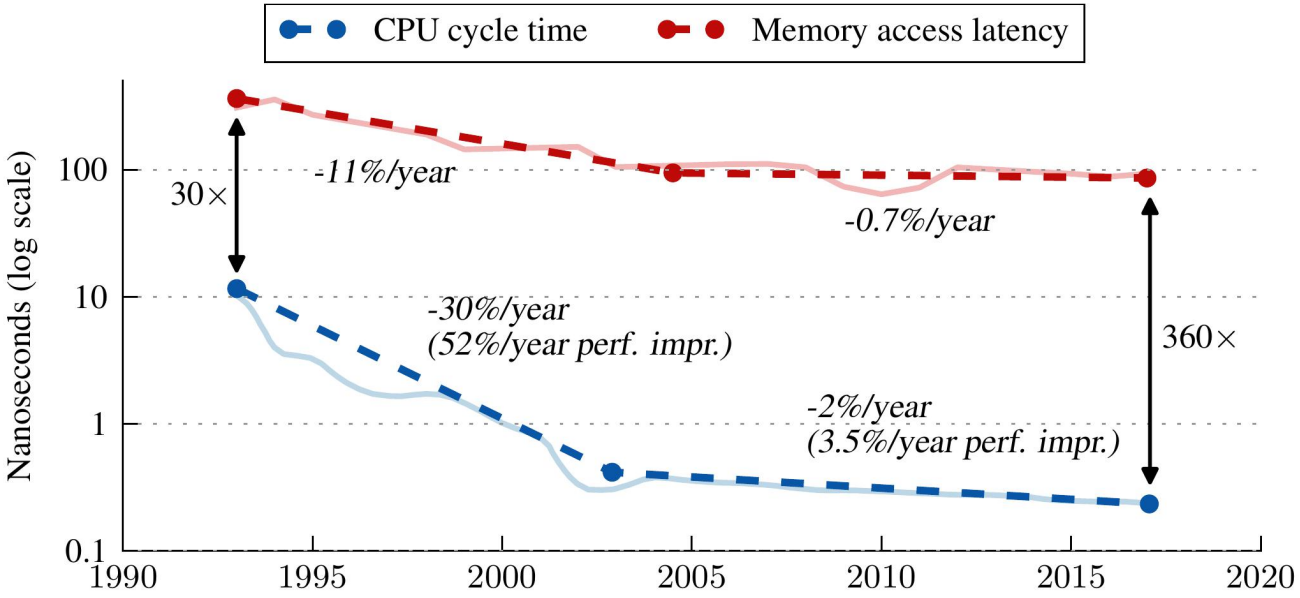


Figure 2.4: Increasing discrepancy between main memory access latency and CPU cycle time, for last 25 years. Recently the downtrend in CPU cycle time decreased to 2%, while single processor performance improves at 3.5% per year [52]. The decrement in memory access latency is less than 1%.

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

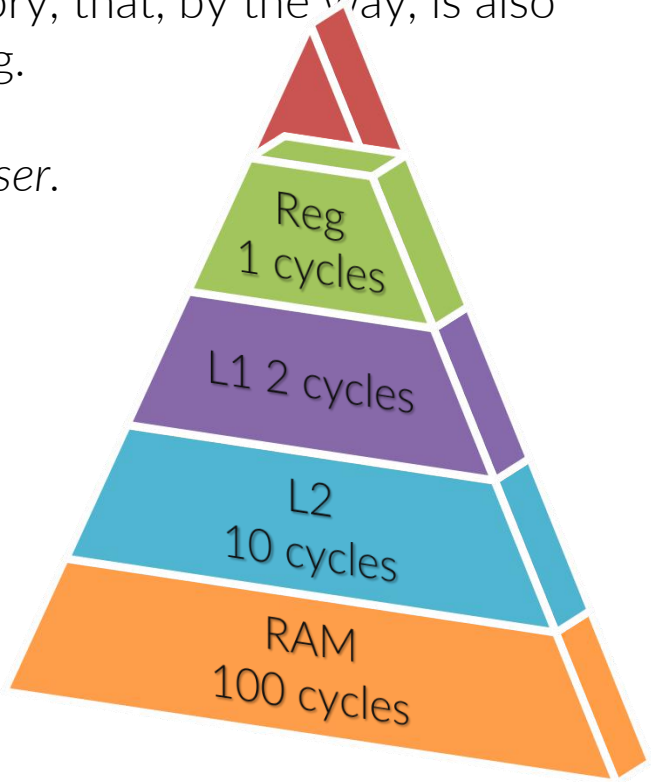
The cache memory

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.



Just a quick curiosity: the origin of the word “*cache*” is linked to winery.

While you have the bulk of your precious wines in the basement cellar, it may be uncomfortable to get back and forth while you are dining with friends..

Just a quick curiosity: the origin of the word "*cache*" is linked to winery.

That's why you are backing up with your ***cache*,** which you have pre-emptively filled with the most suited(*) wines !

(*) *Then, if one of your guests asks for a different wine, you will have to get back to your cellar… Which is exactly the point we'll arrive at.*

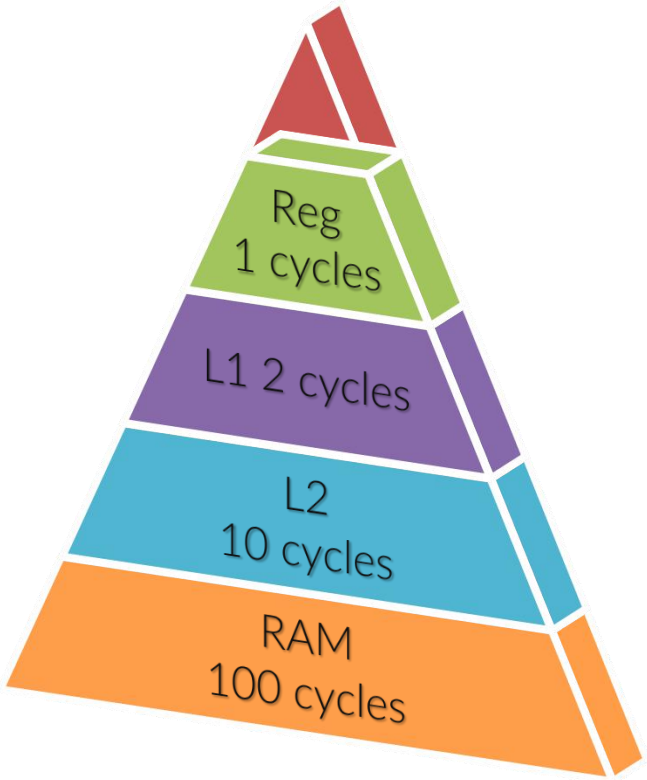
The cache memory

L1 cache + RAM

$L_{1cost} + Miss_1 \times RAM_{cost}$

- 100% L1 hit → 2 cycles
- 99% L1 hit → 3 cycles
- 97% L1 hit → 5 cycles

} 50% to 150% slower



The cache memory

L1 cache + RAM

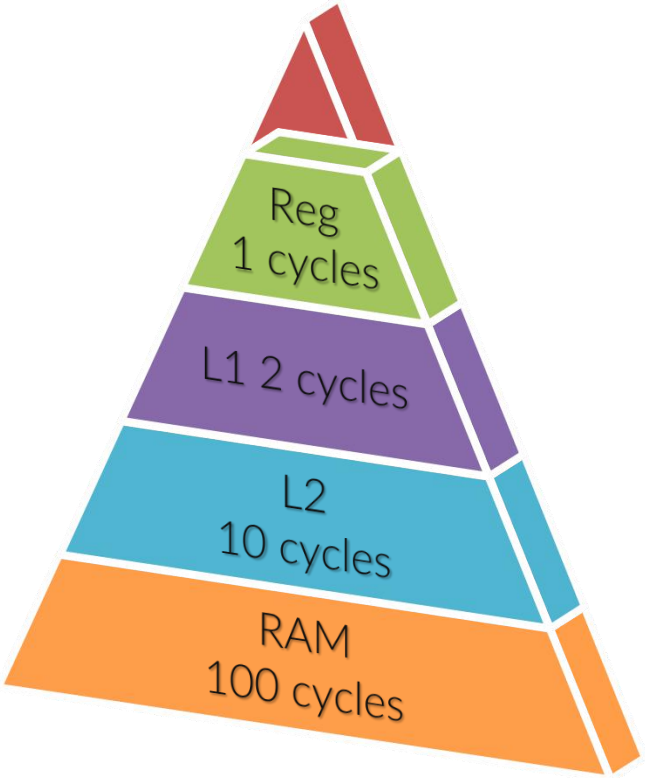
$$L_{1cost} + Miss_1 \times RAM_{cost}$$

- 100% L1 hit → 2 cycles
- 99% L1 hit → 3 cycles
- 97% L1 hit → 5 cycles

L1 cache + L2 cache + RAM

$$L_{1cost} + Miss_1 \times (L2_{cost} + Miss_2 \times RAM_{cost})$$

- 100% L1 hit → 2 cycles
- 99% L1 hit, 100% L2 hit → 2.1cycles
- 97% L1 hit, 100% L2 hit → 2.3 cycles
- 90% L1 hit, 97% L2 hit → 3.3 cycles



The cache memory

L1 cache + RAM

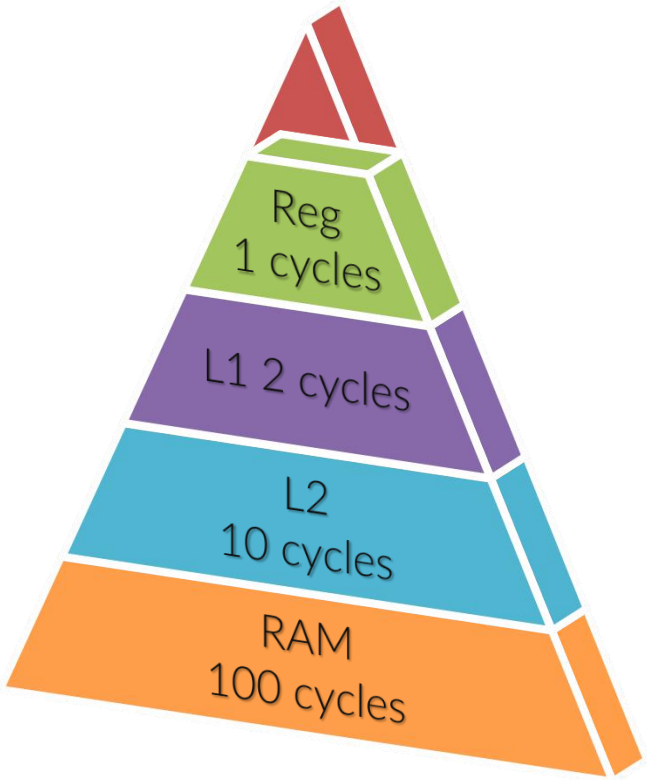
$$L_{1cost} + Miss_1 \times RAM_{cost}$$

- 100% L1 hit → 2 cycles
- 99% L1 hit → 3 cycles
- 97% L1 hit → 5 cycles

L1 cache + L2 cache + RAM

$$L_{1cost} + Miss_1 \times (L2_{cost} + Miss_2 \times RAM_{cost})$$

- 100% L1 hit → 2 cycles
- 99% L1 hit, 100% L2 hit → 2.1cycles
- 97% L1 hit, 100% L2 hit → 2.3 cycles
- 90% L1 hit, 97% L2 hit → 3.3 cycles



The principle of locality

We are quite naturally led to the “principle of locality”:

Data are defined “local” when they reside in a “small”portion of the address space that is accessed in some “short” period of time

→ local data are likely to be in the cache

Temporal locality if an address is referenced, it is likely to be referenced again soon

Spatial locality if an address is referenced, close addresses are likely to be referenced soon

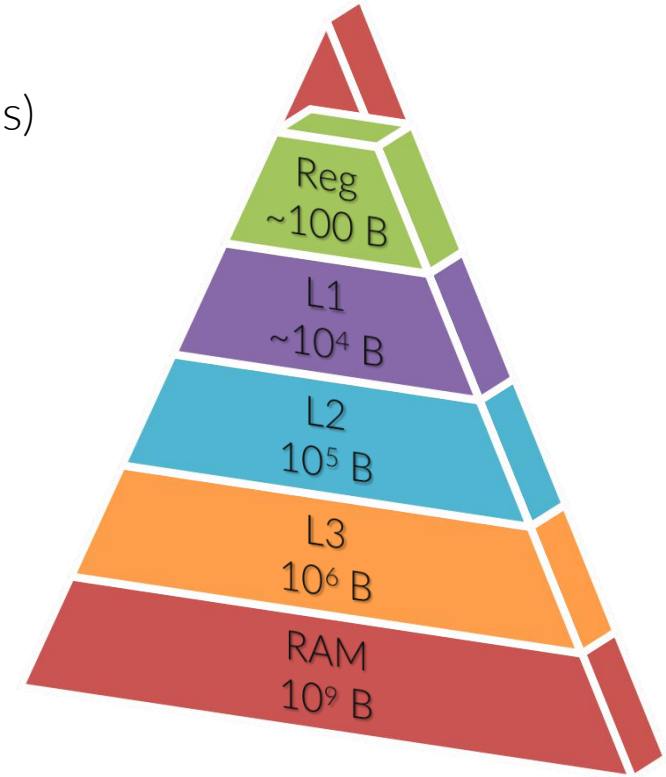
Cache mapping

Question:
The RAM contains ~10^9 bytes, while L1 contains ~10^4 bytes (32KB for data and 32KB for instructions)

So, how do you map the RAM in to a given level of cache, for instance L1, in an effective way?
The main problems are:

- *Where* to map an address
- *What if* the location in L1 is already occupied?

…we'll see that in the next lecture



Cache recap in two slides

3C for the foes

- **Compulsory misses**
  Unavoidable misses when data are read for the first time
- **Capacity misses**
  - Not enough space to hold all data
  - Too much data accessed in between successive use
- **Conflict misses**
  Cache trashing due to data mapping to same cache lines

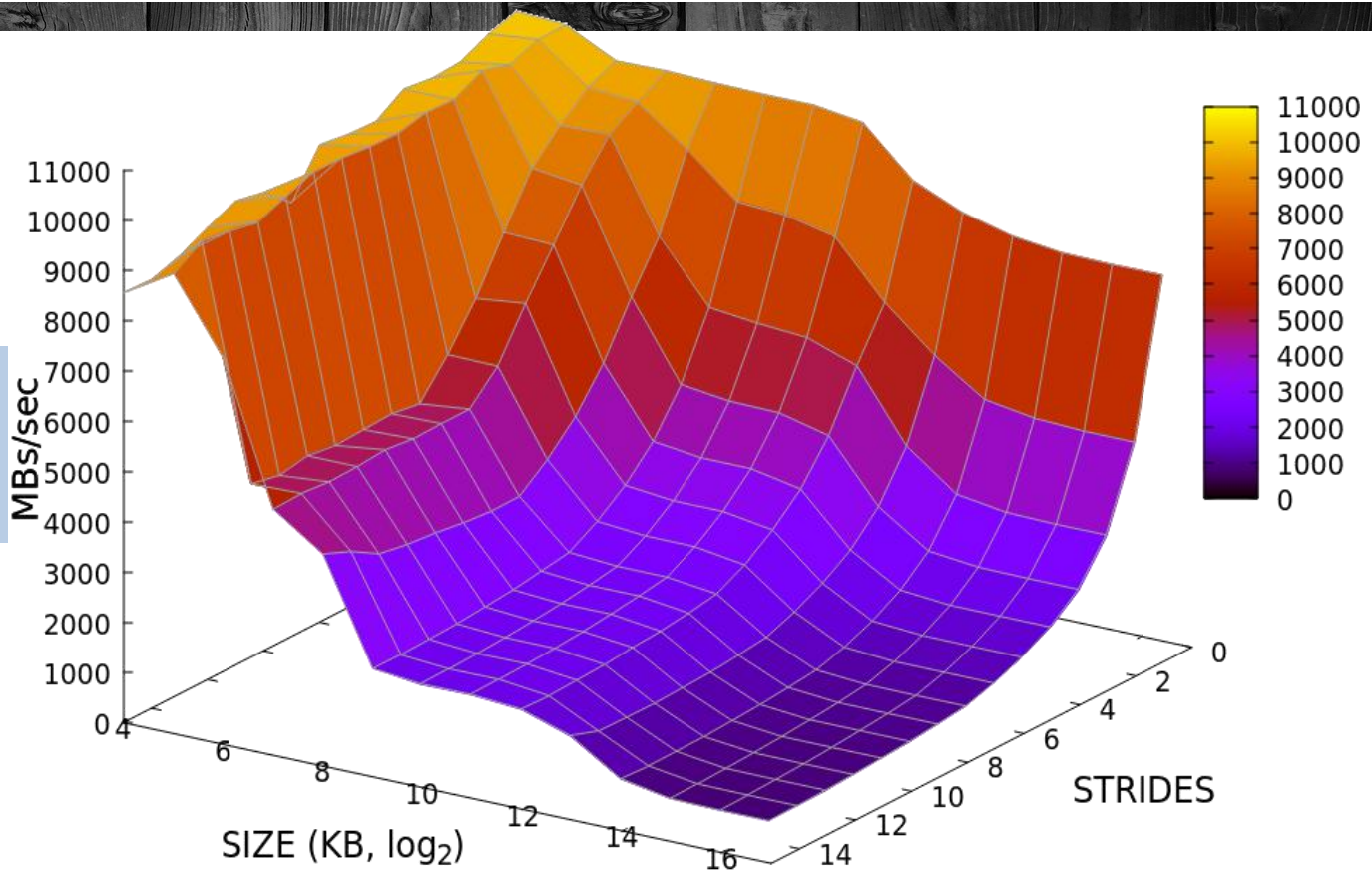
Cache recap in two slides

3R for the friends

- **Rearrange (code & data)**
  Design layout to improve temporal & spatial locality
- **Reduce (size)**
  - Smaller data size – smaller chunks accessed
  - Fewer instructions
- **Reuse (cache lines)**
  Increase spatial & temporal locality – keep resident data for more operations

Preview: The memory access pattern

The result is..
the memory mountain



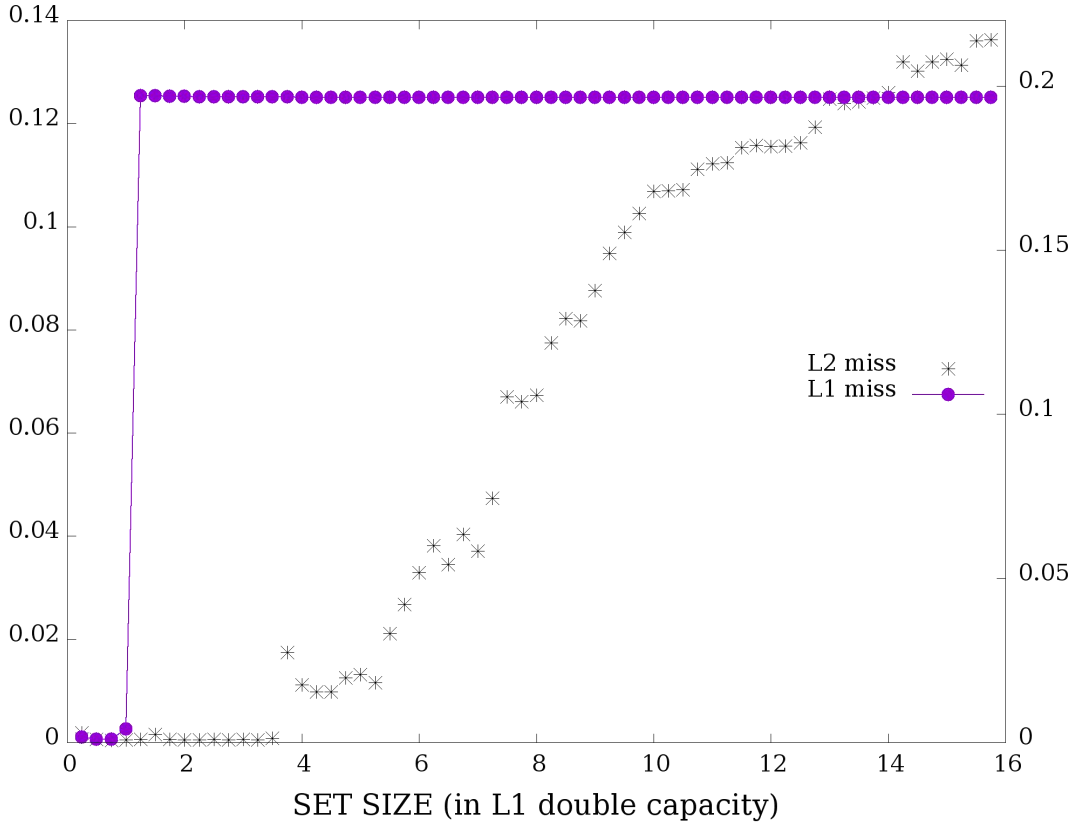
Just a preview;
more details in the next lecture

Preview: The cache-miss signature

Let's find out our cache size

```
for (j=0; j < size; j++)
   array[j] =
     2.3*array[j]+1.2;
```

Just a preview;
more details in the next lecture

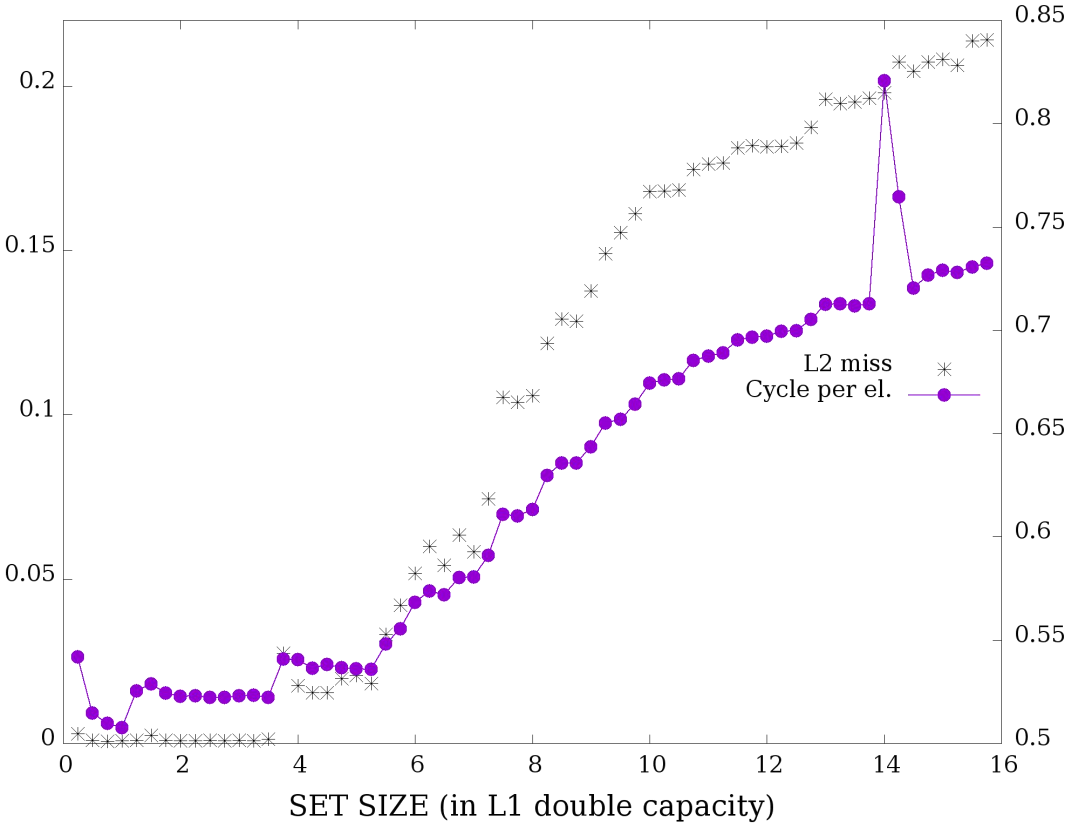


Preview: The cache-miss signature

And the effect on cycles-per-operation metrics

```
for (j=0; j<size; j++)
   array[j] =
     2.3*array[j]+1.2;
```

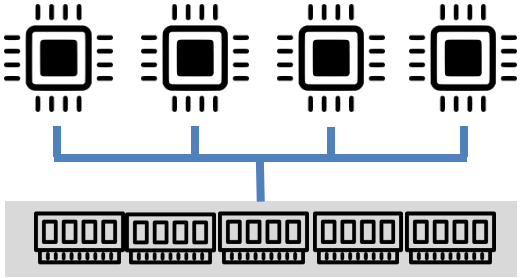
Just a preview;
more details in the next lecture



A coherency problem

You already know the difference between

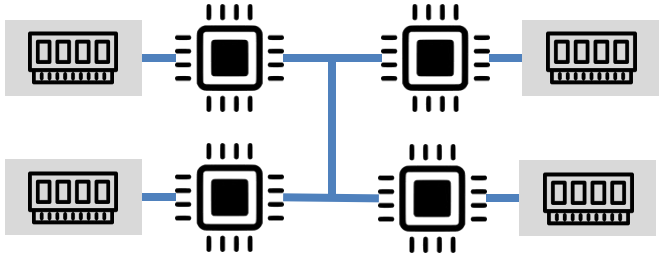
SMP



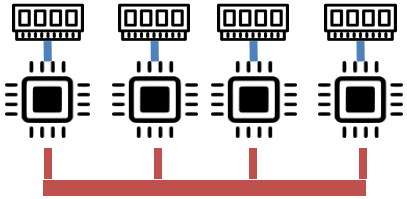
CPUs

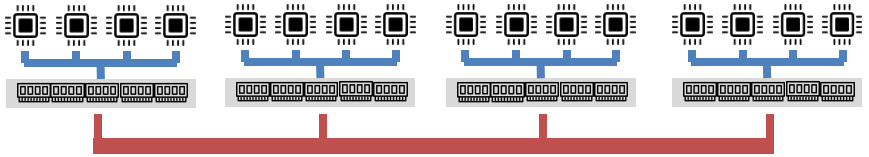
BUS

Memory



Disitributed NUMA

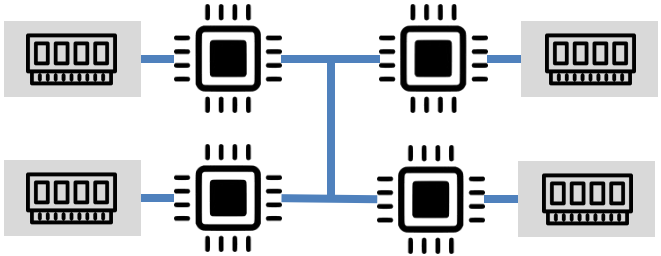




A coherency problem

Consider an SMP node, with tens of sockets interconnected by a bus

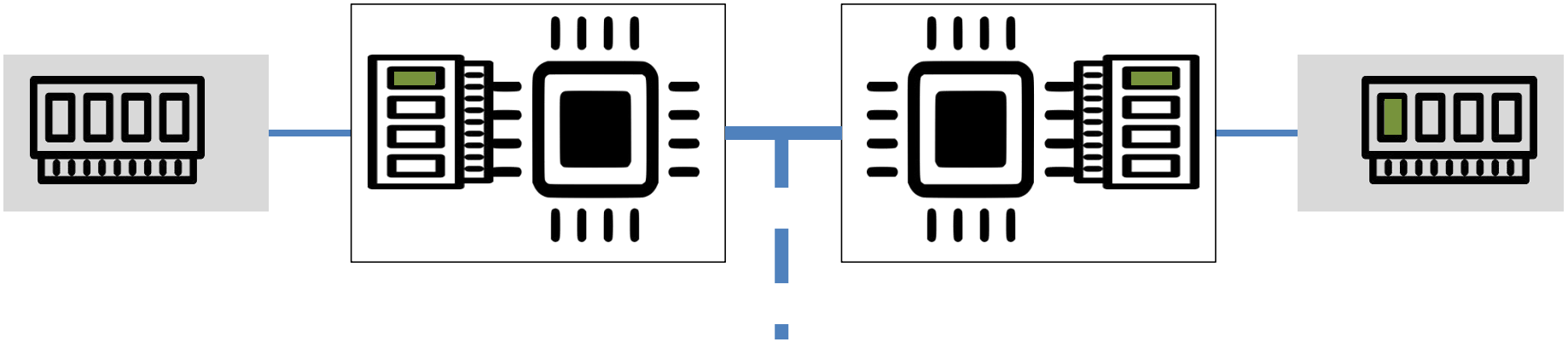
*(remind: a **bus** is a "collective" interconnect in which the messages are broadcasted and everyone is listening for a message dedicated to itself)*



The memory is *shared*, and everybody sees the whole amount of RAM

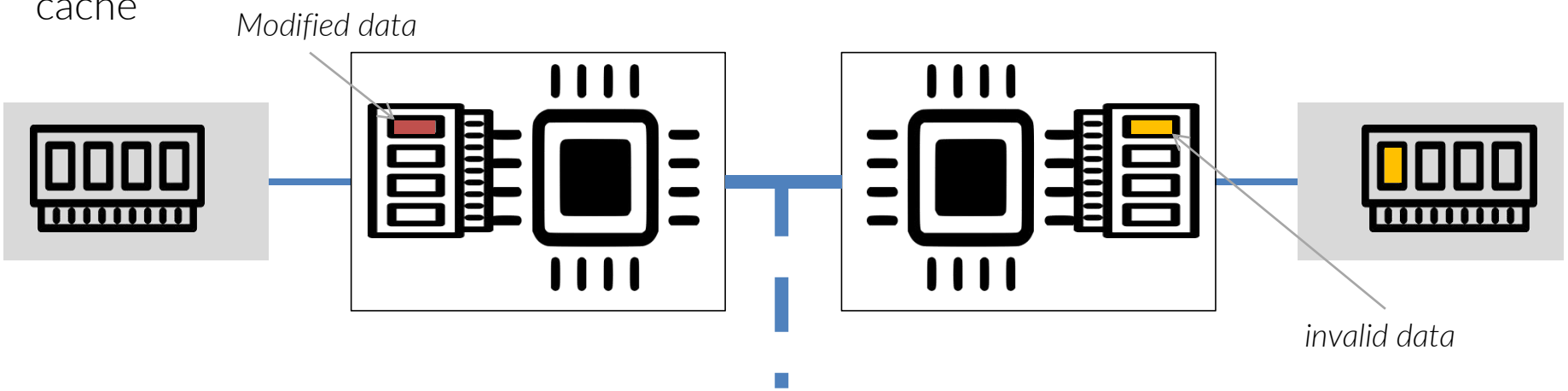
A coherency problem

Let's say that the CPUs have caches and some data are loaded in more than one cache



A coherency problem

Let's say that the CPUs have caches and some data are loaded in more than one cache



What happens to all the caches and "actual" data in memory when one CPU modifies the data?
That is called *cache coherency* and the overwhelming difficulty and cost to manage it on too large SMP nodes is the main limit to their size.

Early 90s: CPU becomes faster than memory

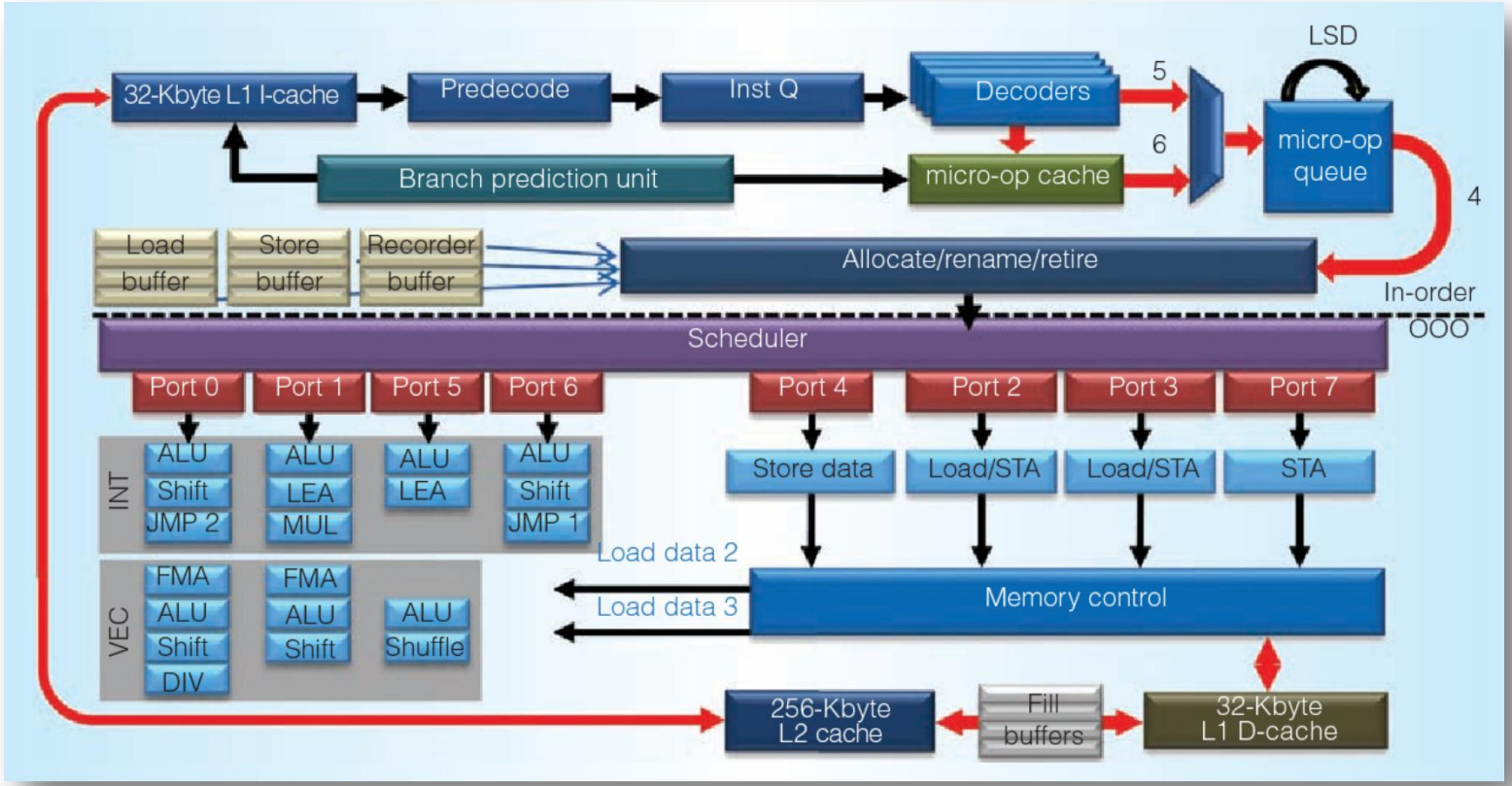
So, after having introduced the hierarchy of cache memories, you have to deal with a strong memory hierarchy and the fact that your CPU is much faster than the central memory.

Even if you are very good in designing the data model and the workflow of your code, that complexity may result in a real performance disaster: for instance because your CPU is stuck doing nothing while waiting for some data or for some operation to end.

This leads us to some more improvements.

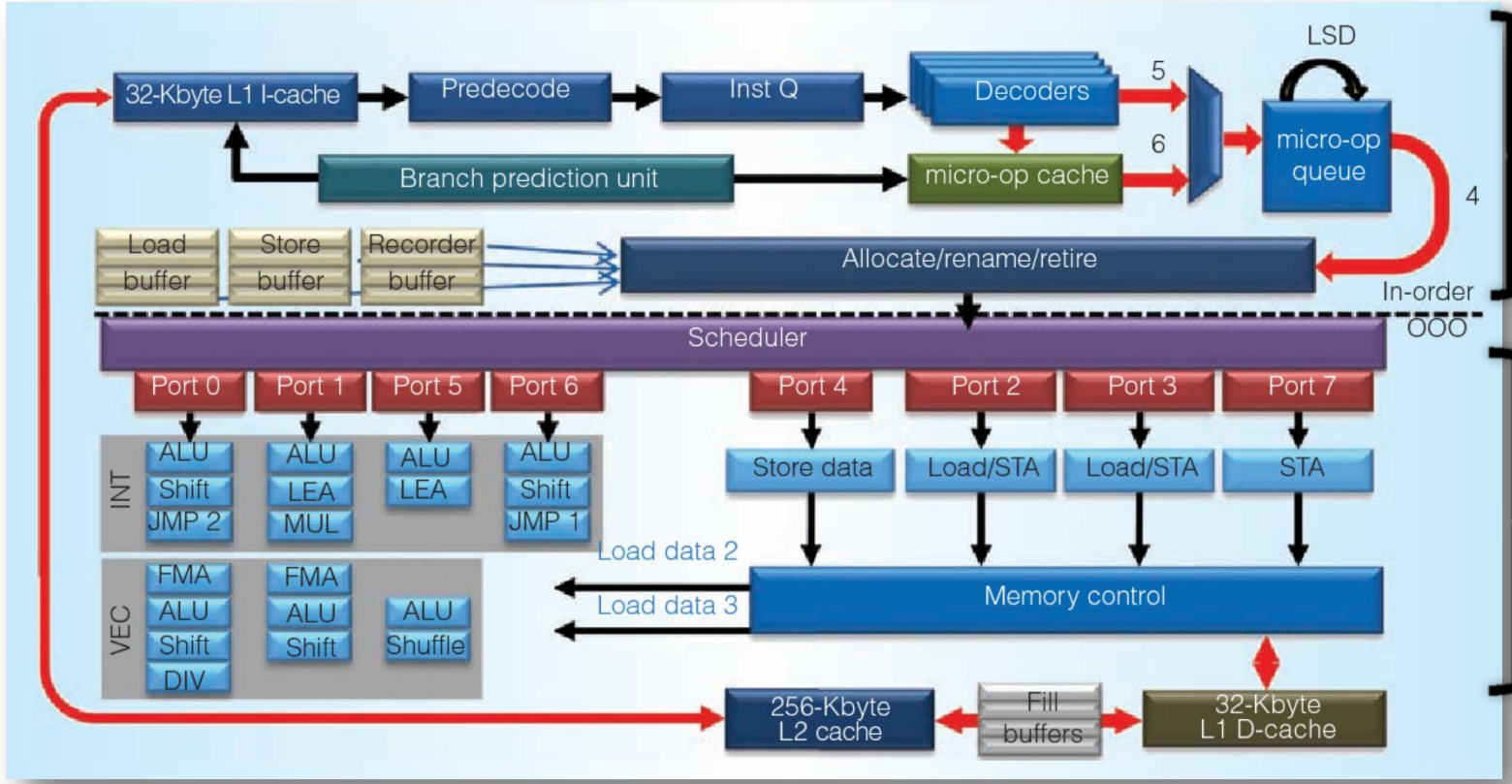
OoO, ILP, Pipelines, Vectors

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



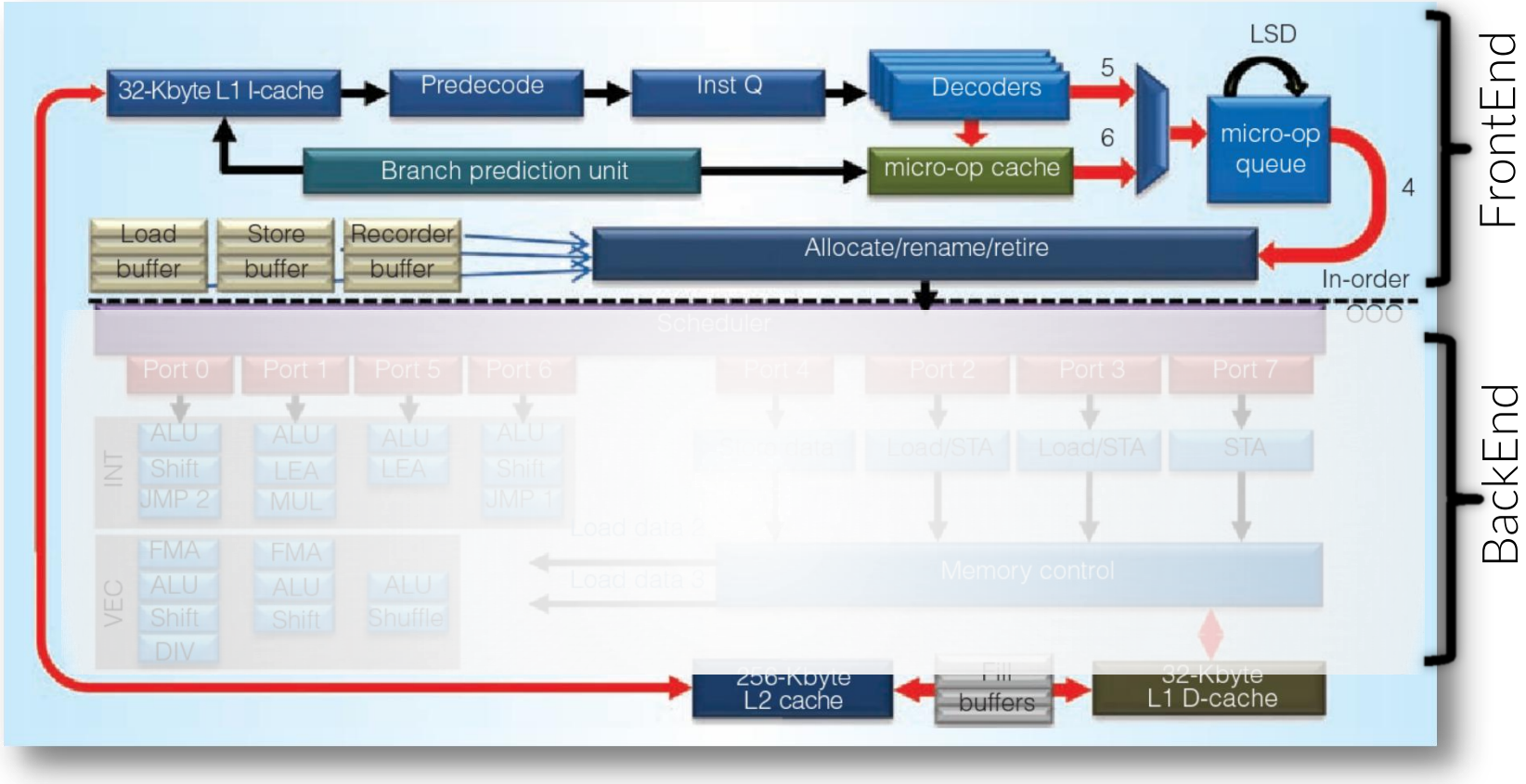
6th generation SkyLake micro-arch.

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.



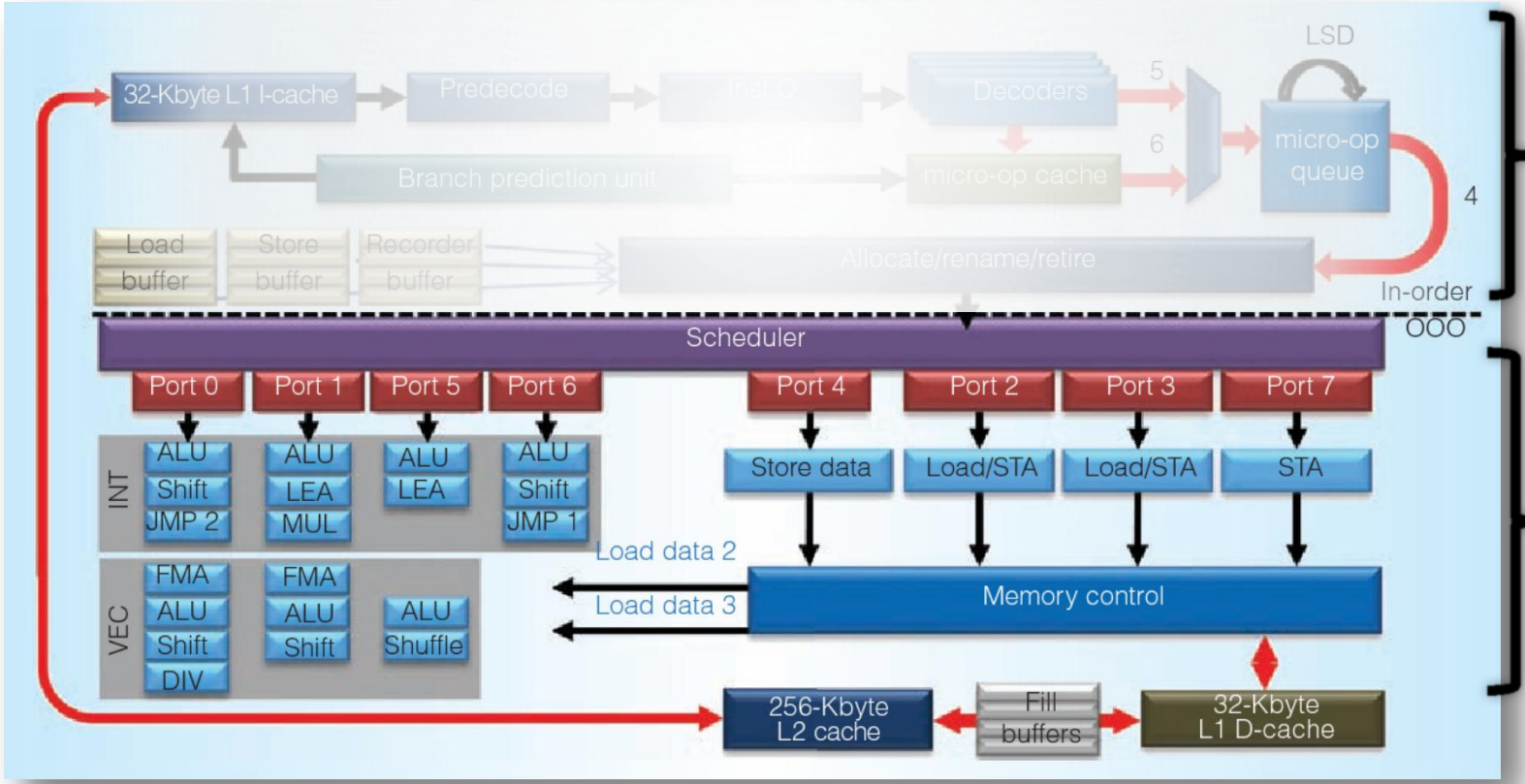
6th generation SkyLake micro-arch.

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.

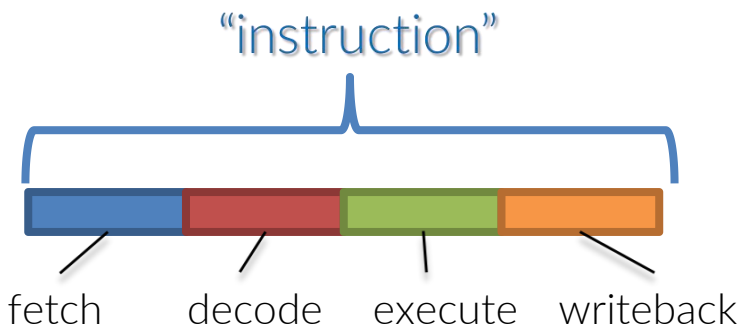


6th generation SkyLake micro-arch.

Pipelines

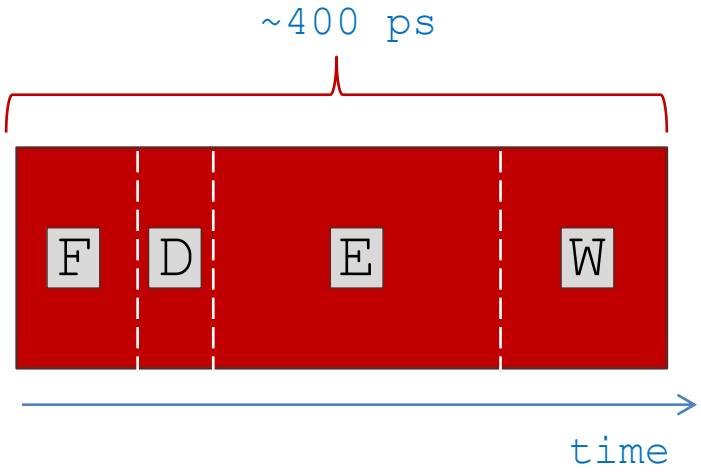
It would be obvious to think that an “instruction” is a kind of *atomic* operation that the CPU perform as a whole. Indeed that was true until the mid of the 80s.

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

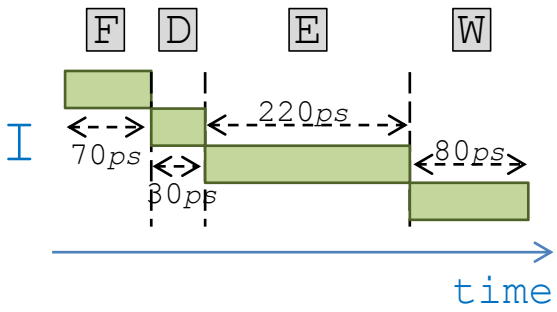


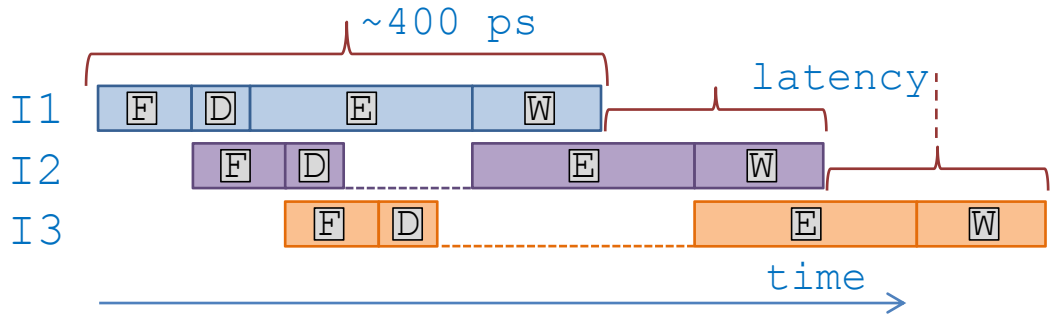
If all the four stages take ~400ps, we then would obtain a **throughput** of 2.5GIPS (giga-instructions per second).
400ps is also the total time required to get a result from an instruction, and so it is the **latency** of the instruction.

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



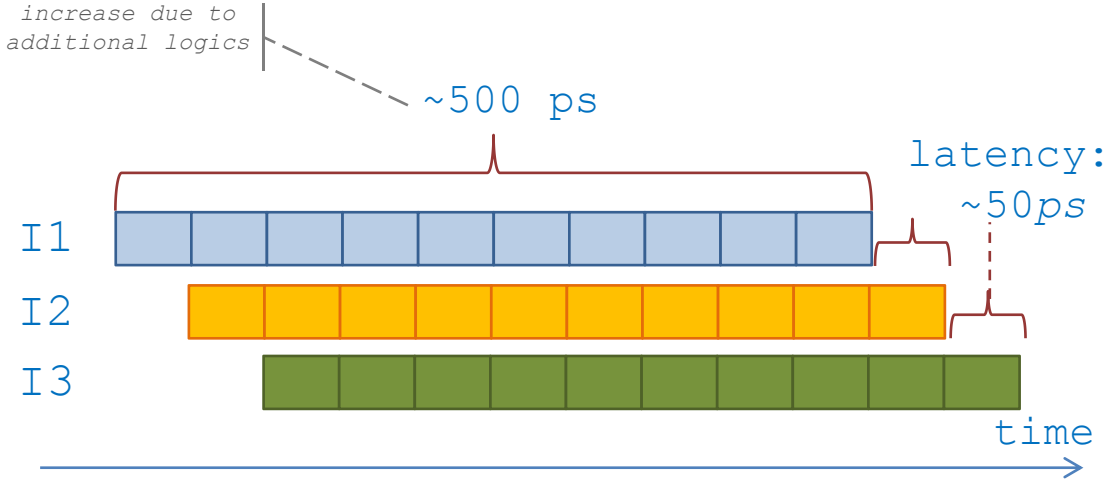


If many independent logical units exist to perform each step, they could operate subsequently on **different instructions:**

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. **20GIPS**

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

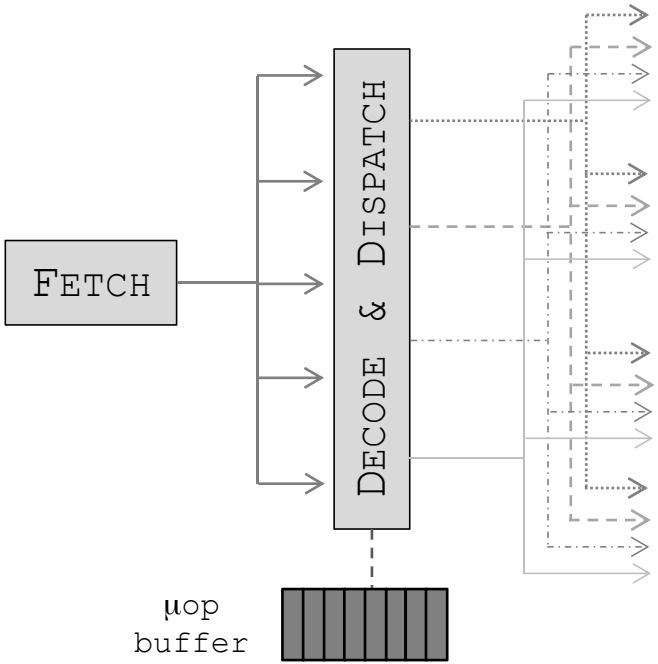
Pipelines are about throughput

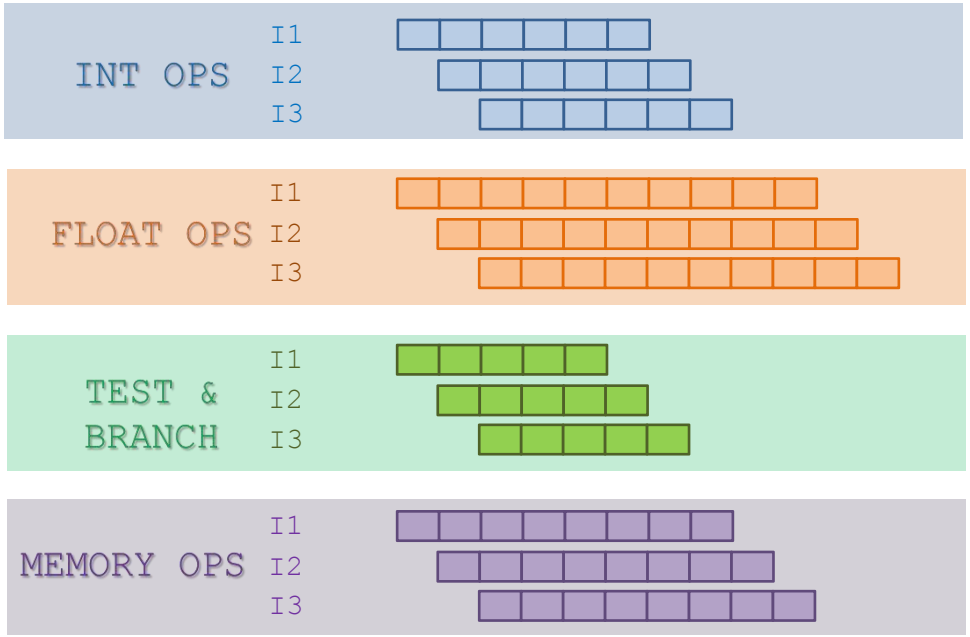
Pipelining is then about increasing the throughput of a system, and as we have seen it could be really effective.

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:

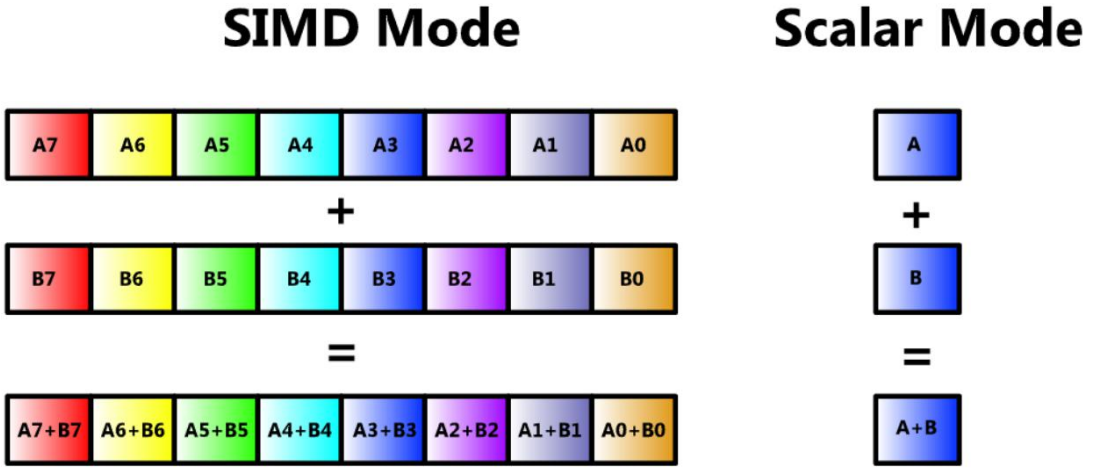
Multiple pipelines





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*	

Branches

We’ll treat this separately

Many Cores

That is the focus of the course on OpenMP in the next days

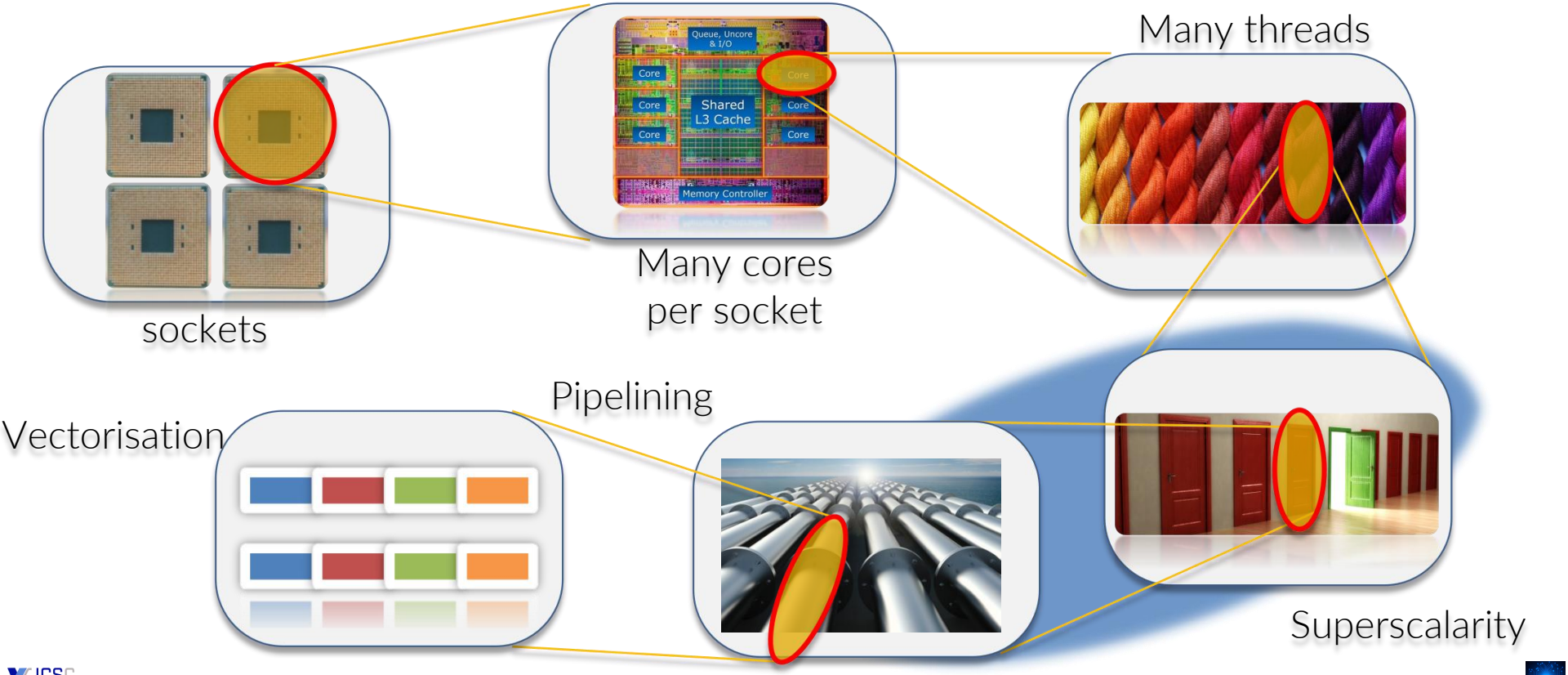
Many Nodes

That is the focus of the course on MPI that we will organise in the future

Heterogeneous devices

That is the focus of the course on GPUs that we will organise in the future

A final sketch



A final sketch

Data-Level Parallelism

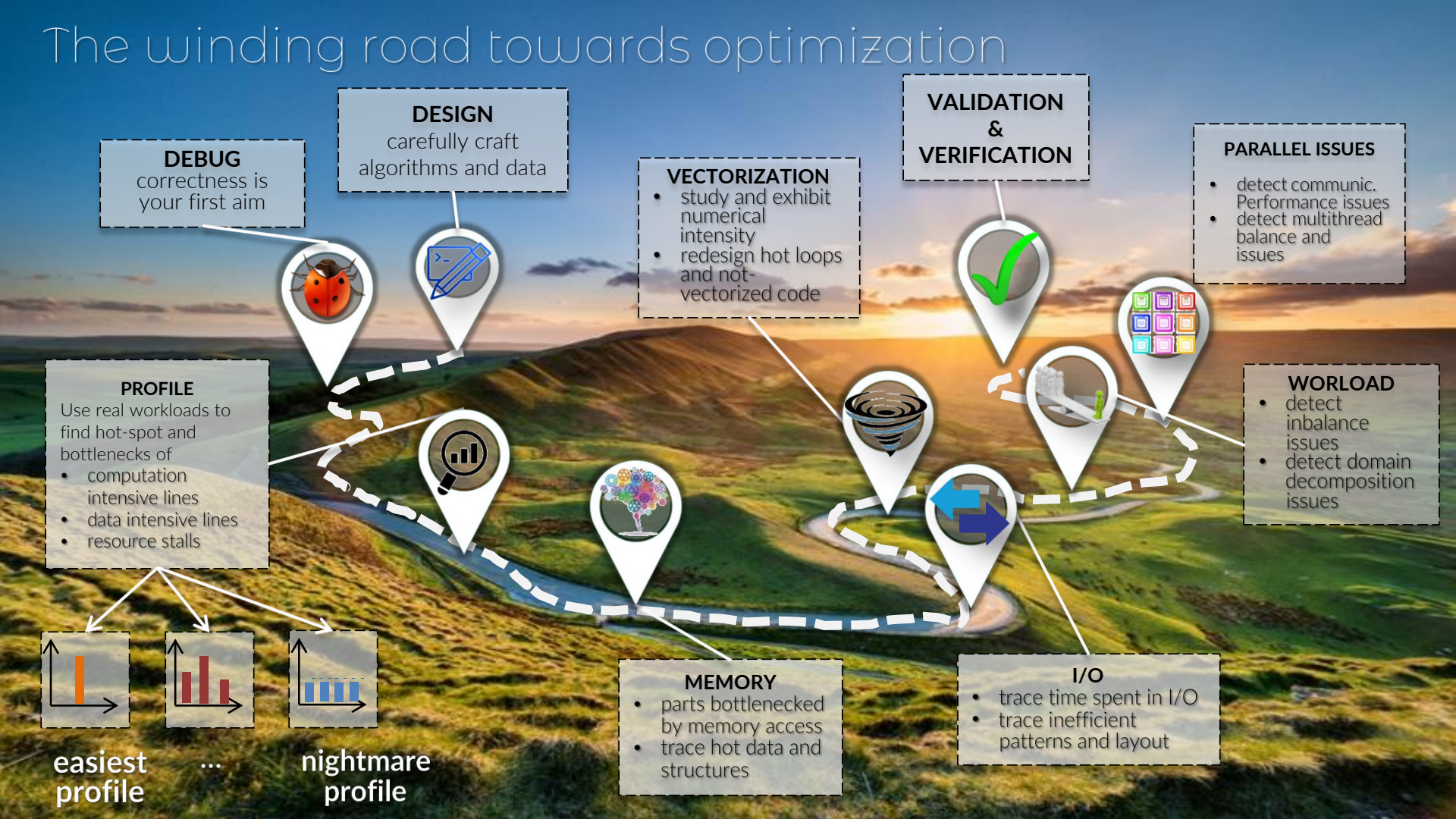
Instruction-Level Parallelism

Thread-Level Parallelism

The parallellism levels in a computational node.
We'll get through each of them separately.

Thread-Level parallelism will be covered in the OpenMP lectures

The winding road towards optimization



that's all, have fun