

Some details on Modern Architectures

Luca Tornatore - I.N.A.F.



“Foundation of HPC” course



DATA SCIENCE &
SCIENTIFIC COMPUTING
2020-2021 @ Università di Trieste

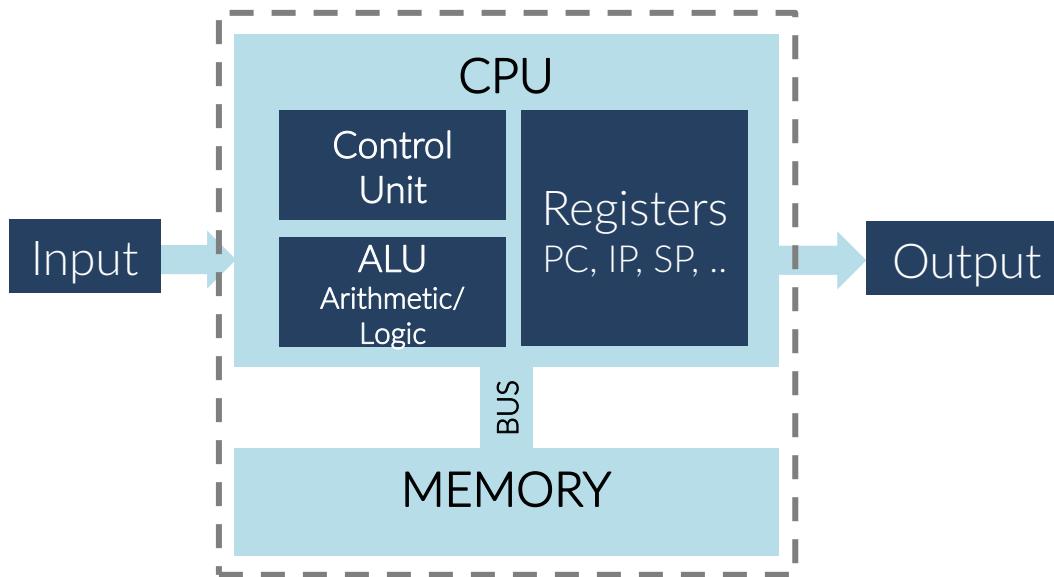


This lecture presents the fundamental traits of the **single-core** modern architecture, with some attention to its historical developments and roots.



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.

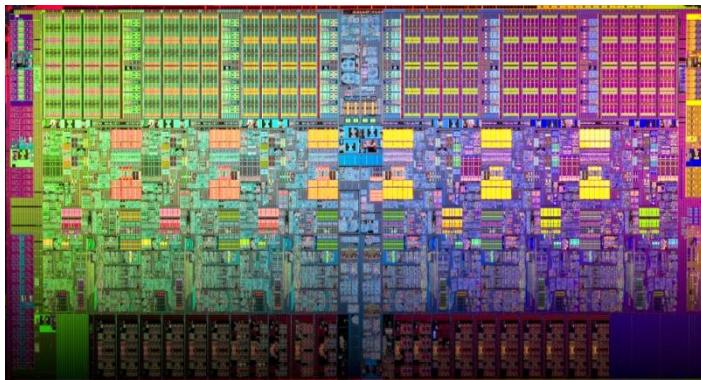


In the **Von Neumann architecture** (still taught as the fundamental model)

- **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

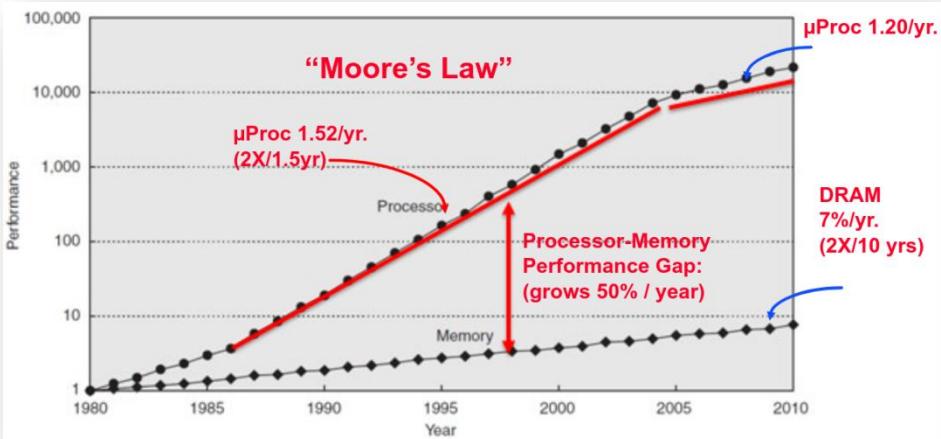
(*) "cost" is in the CPU-cycles currency

- 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 μ ops 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 μ op or reading an internal register

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



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 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 ?



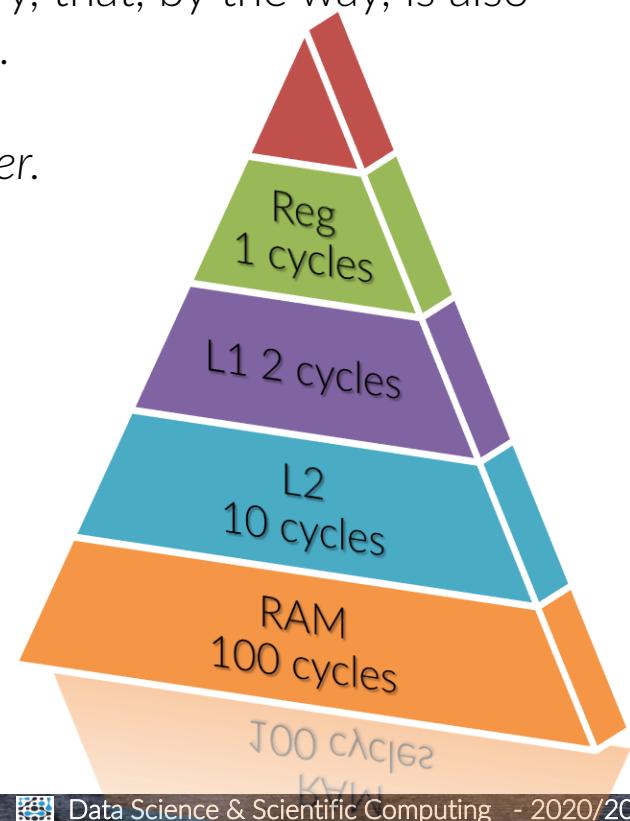
| The cache memory

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.



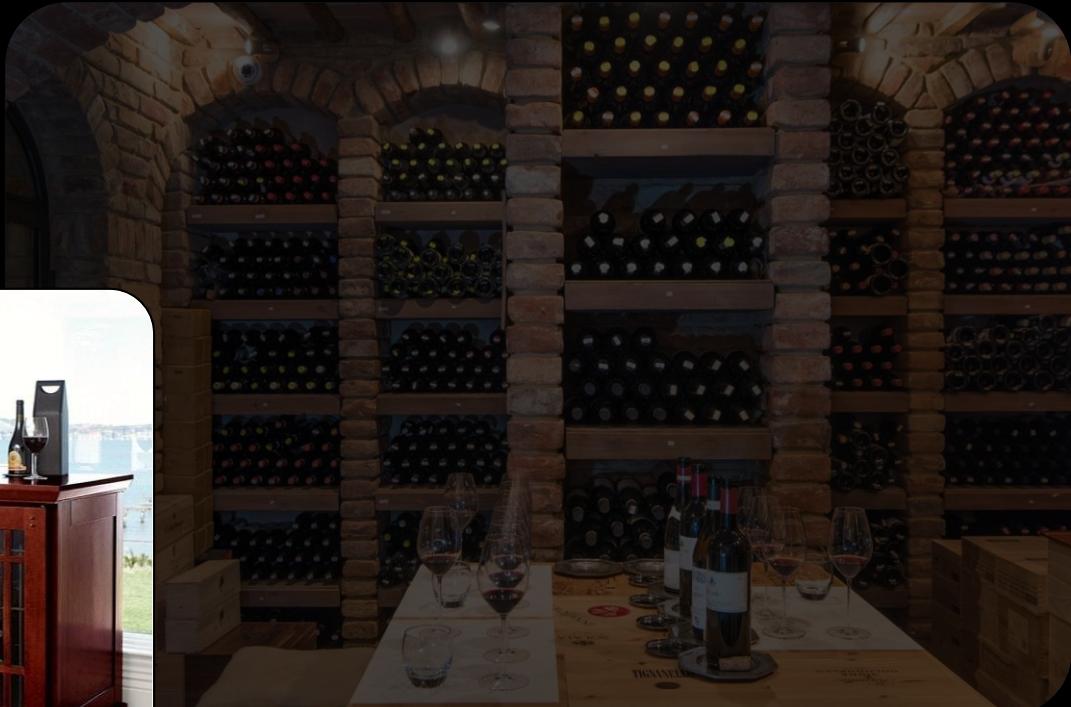
Just a quick curiosity: the origin of the word “cache” is linked to winery.

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 pre-emptively you filled with the most suited wines !



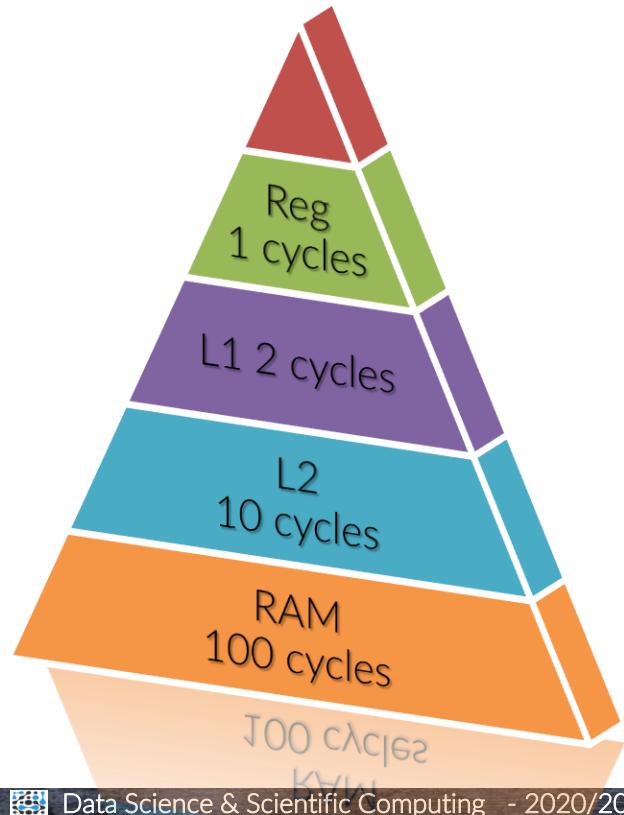


| The cache memory

L1 cache + RAM

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

- 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

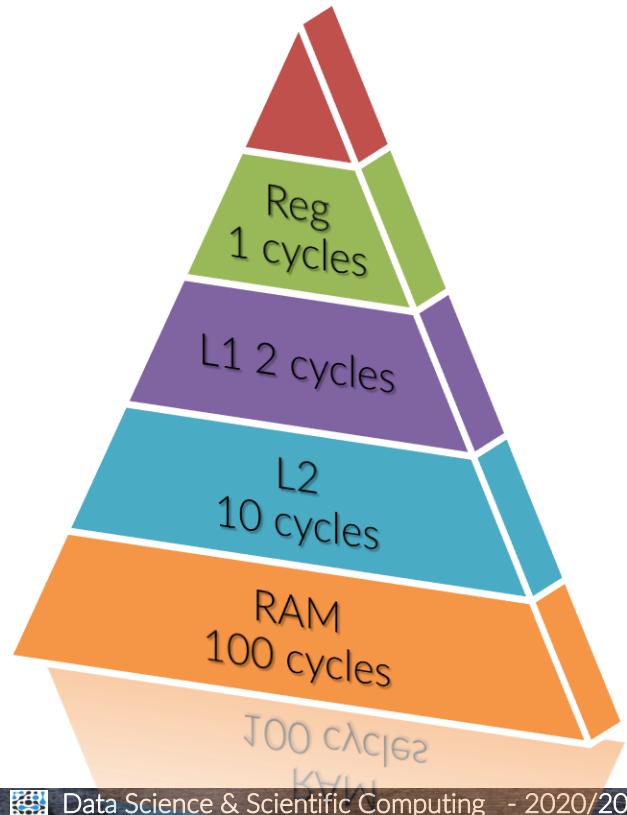
$$L_{1\text{cost}} + \text{Miss}_1 \times RAM_{\text{cost}}$$

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

L1 cache + L2 cache + RAM

$$L_{1\text{cost}} + \text{Miss}_1 \times (L2_{\text{cost}} + \text{Miss}_2 \times RAM_{\text{cost}})$$

- 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

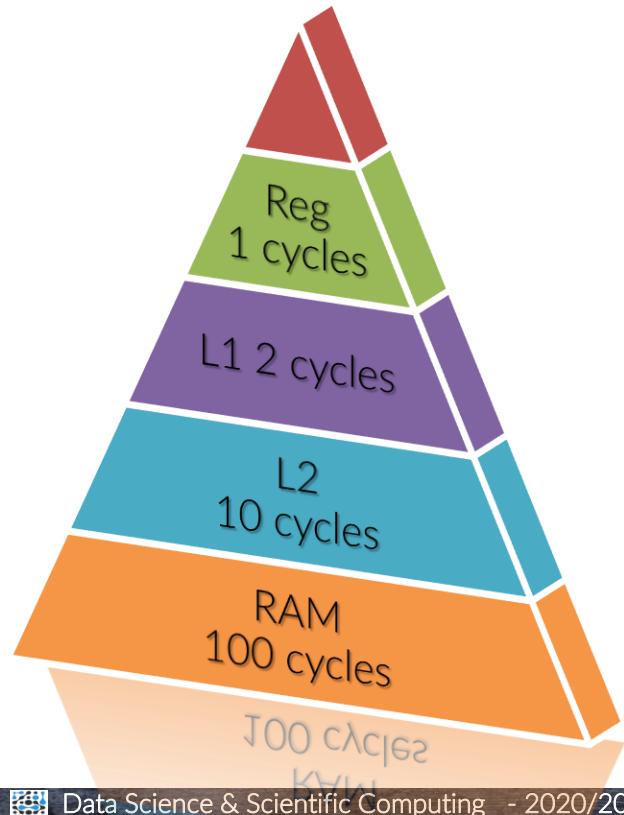
$$L_{1\text{cost}} + \text{Miss}_1 \times RAM_{\text{cost}}$$

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

L1 cache + L2 cache + RAM

$$L_{1\text{cost}} + \text{Miss}_1 \times (L2_{\text{cost}} + \text{Miss}_2 \times RAM_{\text{cost}})$$

- 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:

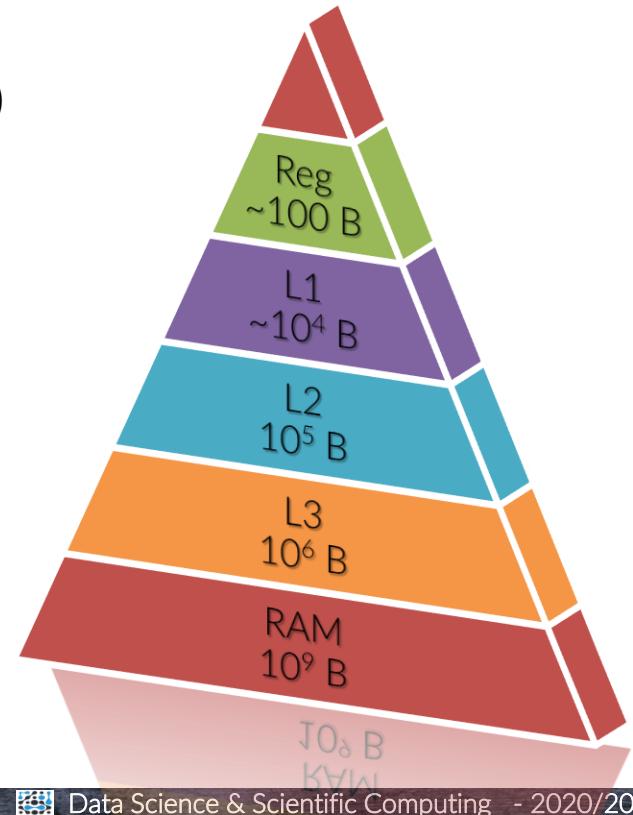
The RAM contains $\sim 10^9$ bytes, while L1 contains $\sim 10^4$ bytes (32KB for data and 32KB for instructions)

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

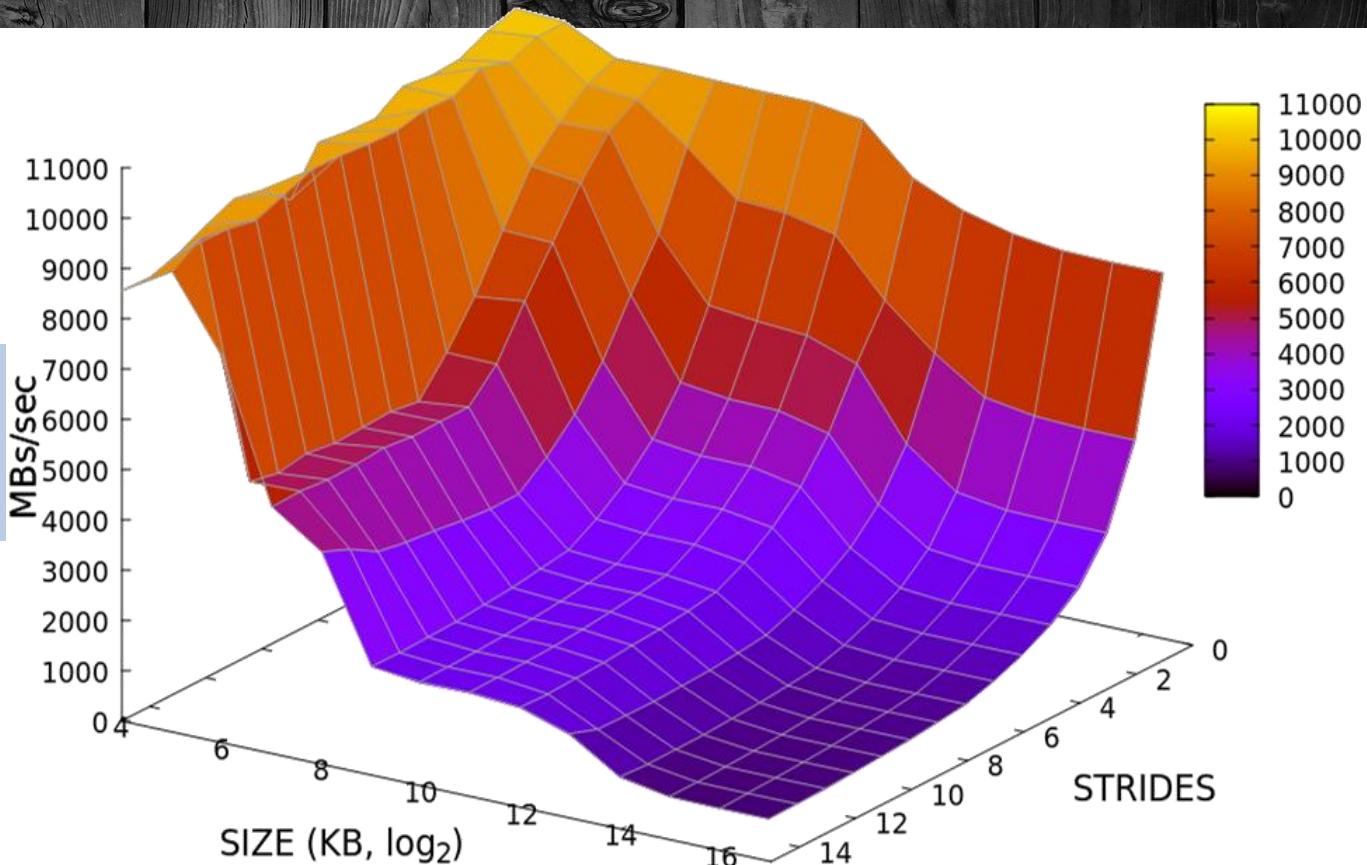
3R for the
friends

- ▶ 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

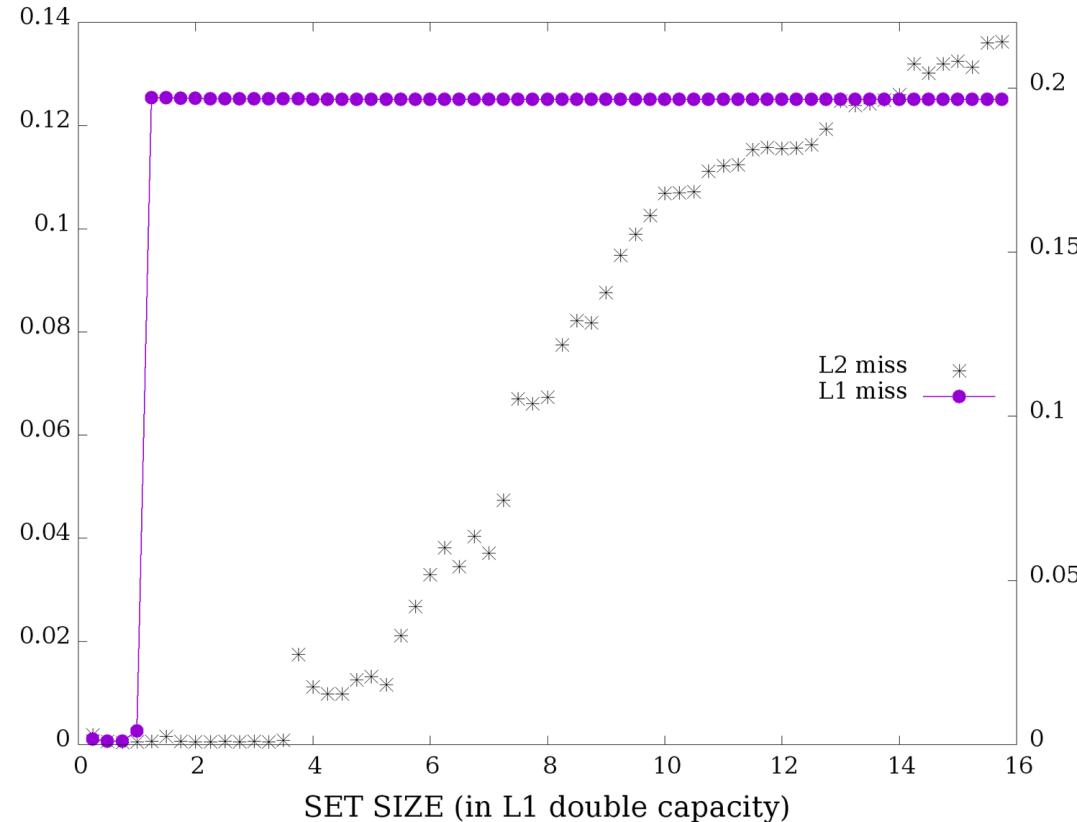


Preview: The cache-miss signature

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



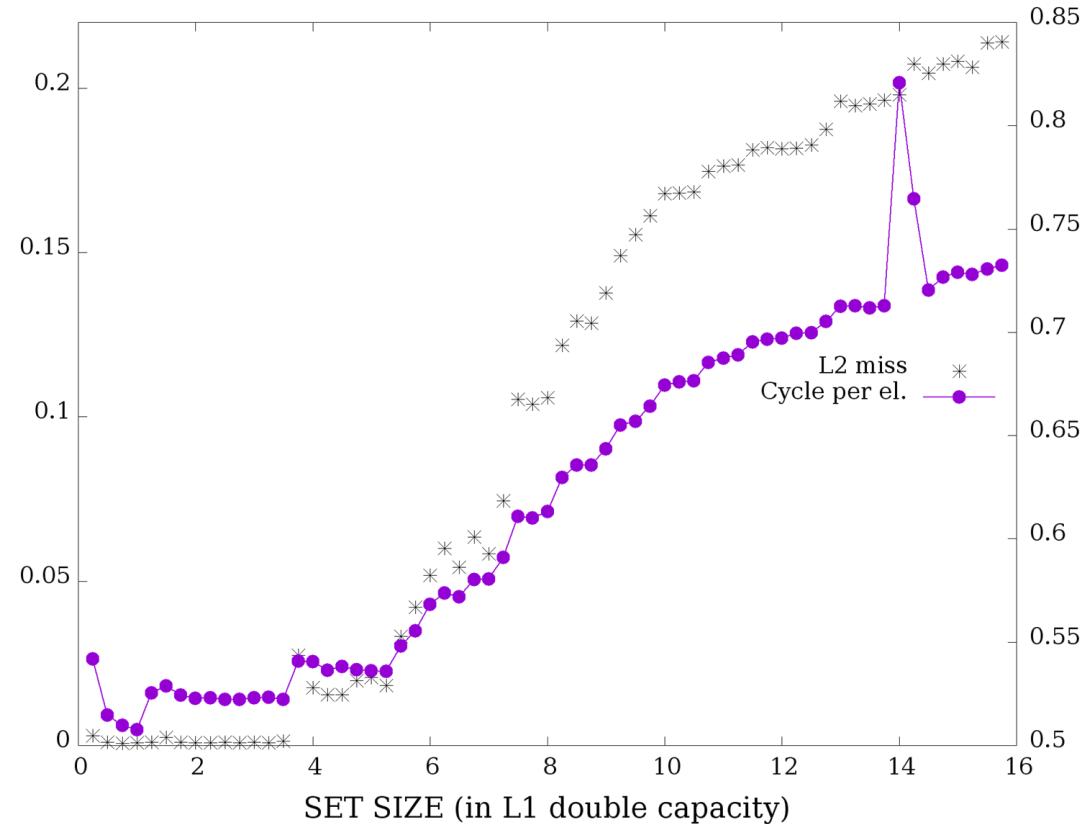


Preview: The cache-miss signature

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

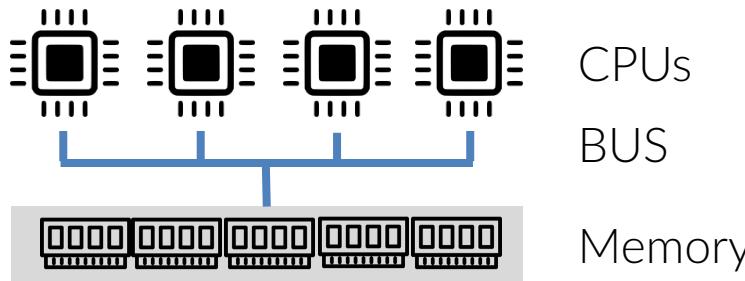




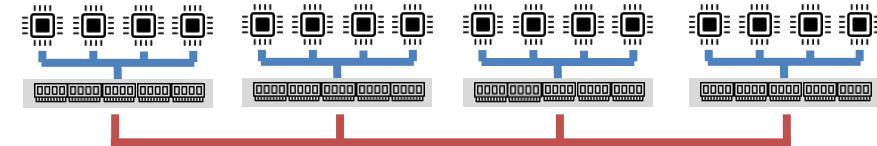
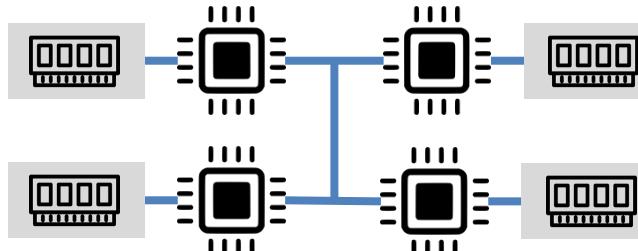
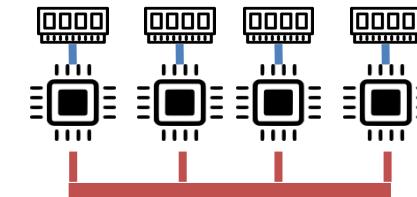
A coherency problem

You already know the difference between

SMP



Distributed NUMA

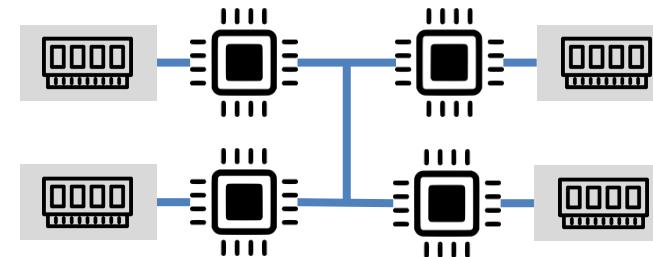




A coherency problem

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)

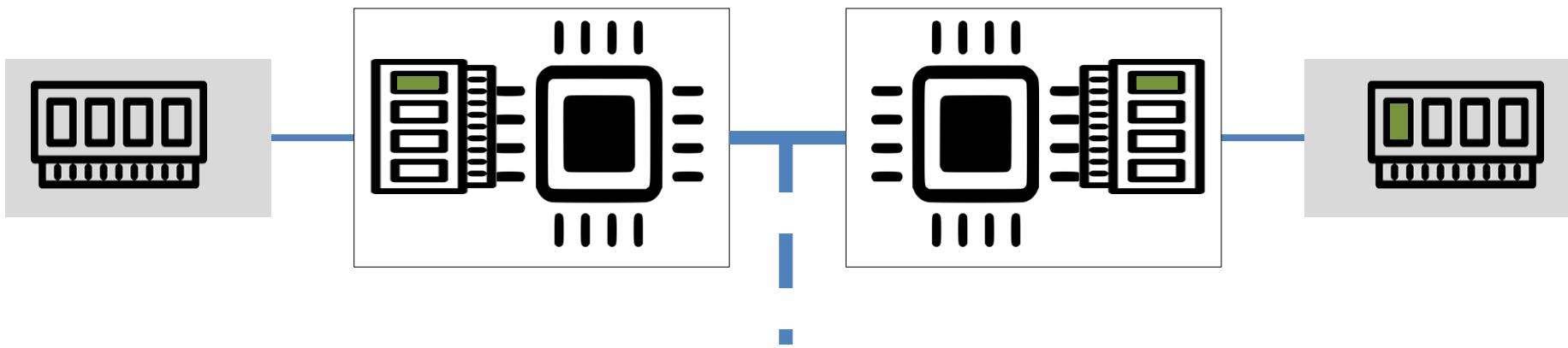


The memory is *shared*, and everybody sees the whole amount of RAM



A coherency problem

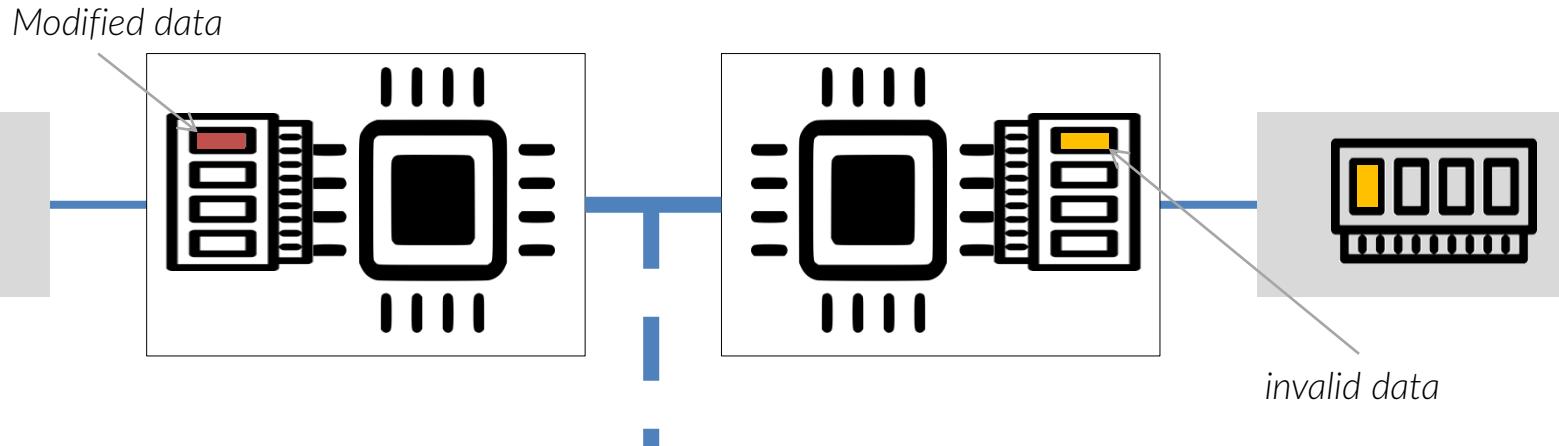
Let's say that the CPUs have caches and some data are loaded in more than one cache





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 get faster than memory

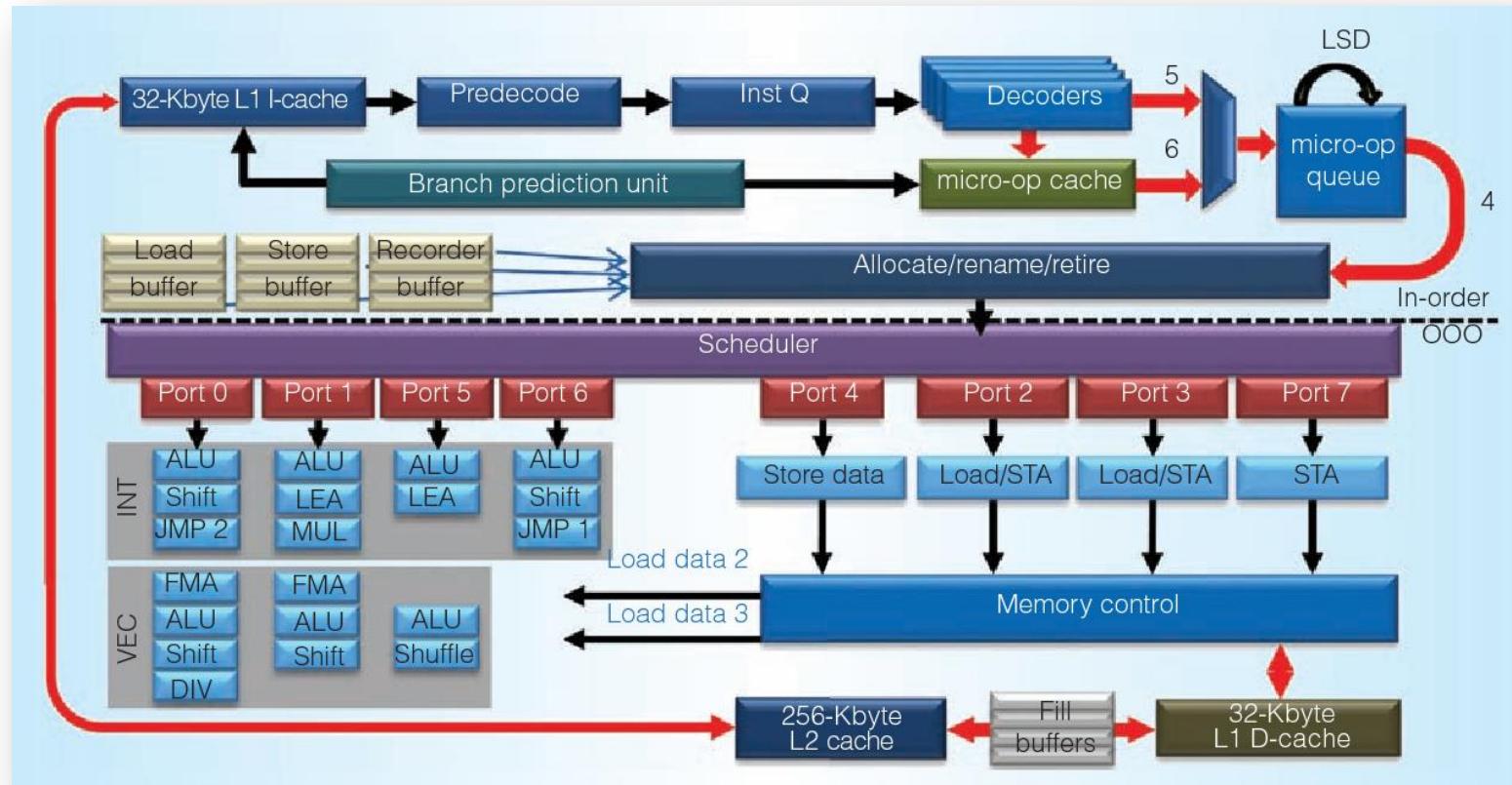
So, after having introduced the hierarchy of cache memories, you have to deal with a strong memory hierarchy and the fact that your CPU is much faster than the central memory.

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.

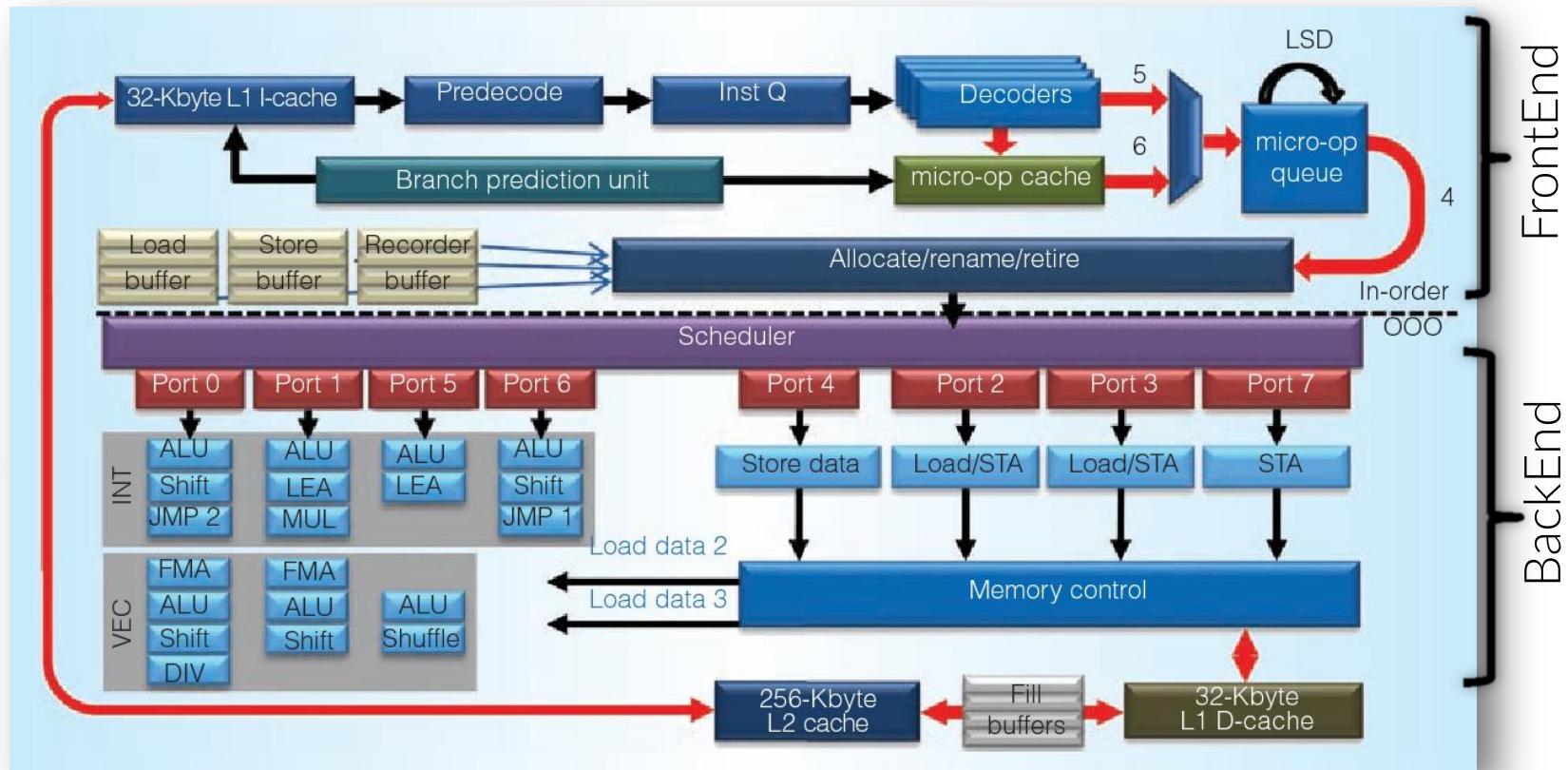
This leads us to some more improvements.



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

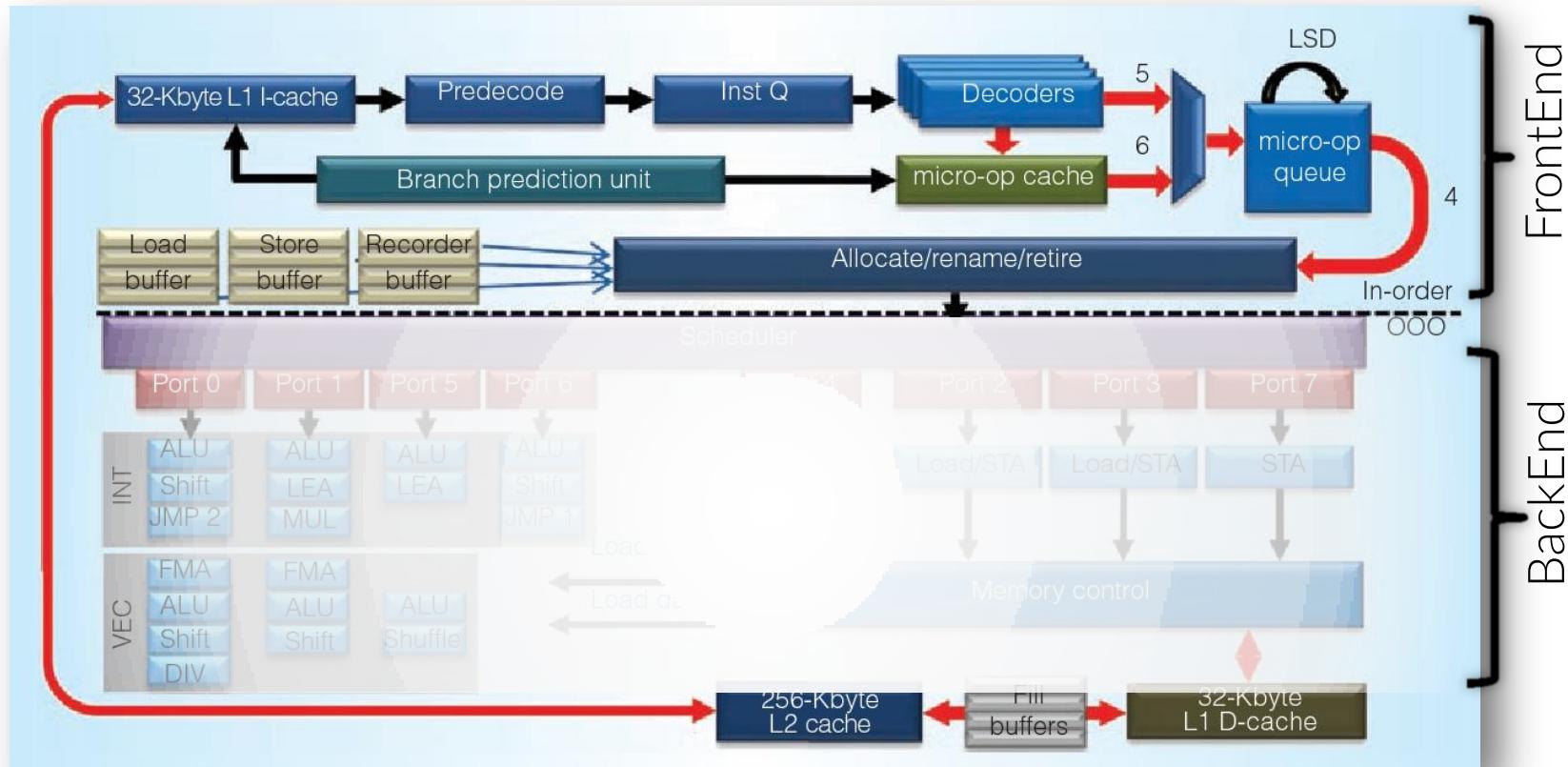


More than 1 port is available to execute CPU instructions, although different units have different specializations (ALU, LEA, SHIFT, FMA, ...) : that is **superscalar capacity**, i.e. the capacity of executing more than 1 instructions per cycle.



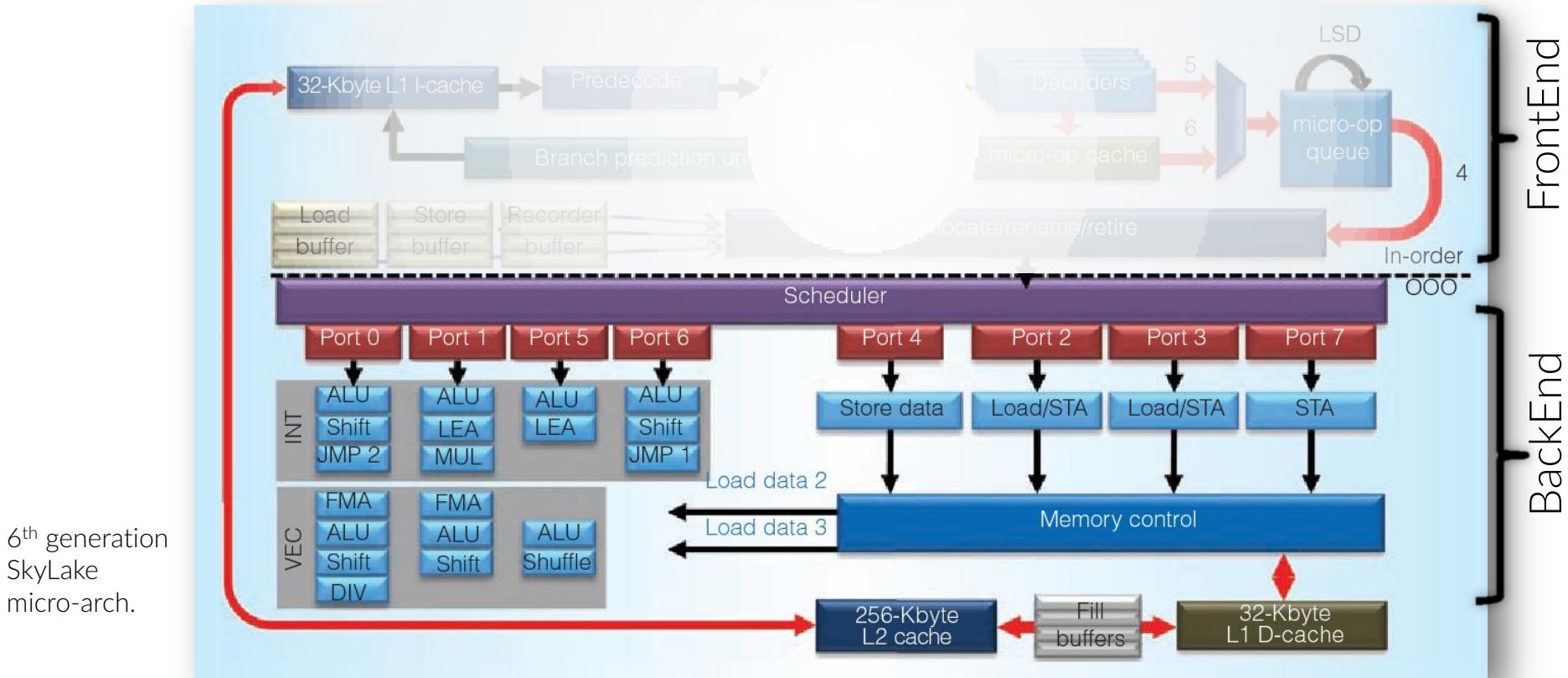
6th generation
SkyLake
micro-arch.

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.



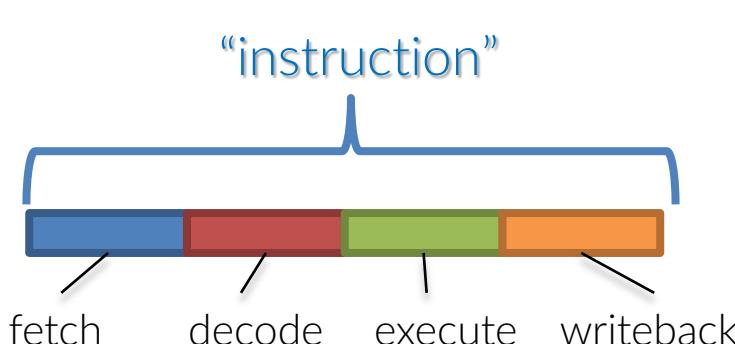
6th generation
SkyLake
micro-arch.



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.

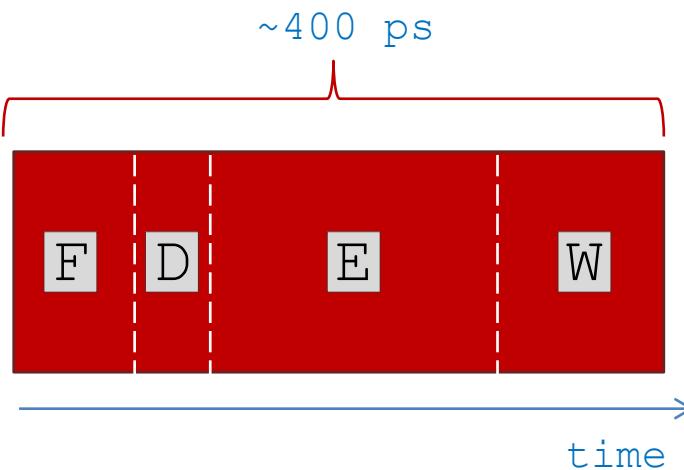
If you think carefully about it, it is easy to understand that actually an “instruction” involves at least the following *independent* steps:



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



Pipelines



If all the four stages take $\sim 400\text{ps}$, we then would obtain a **throughput** of 2.5GIPS (giga-instructions per second).

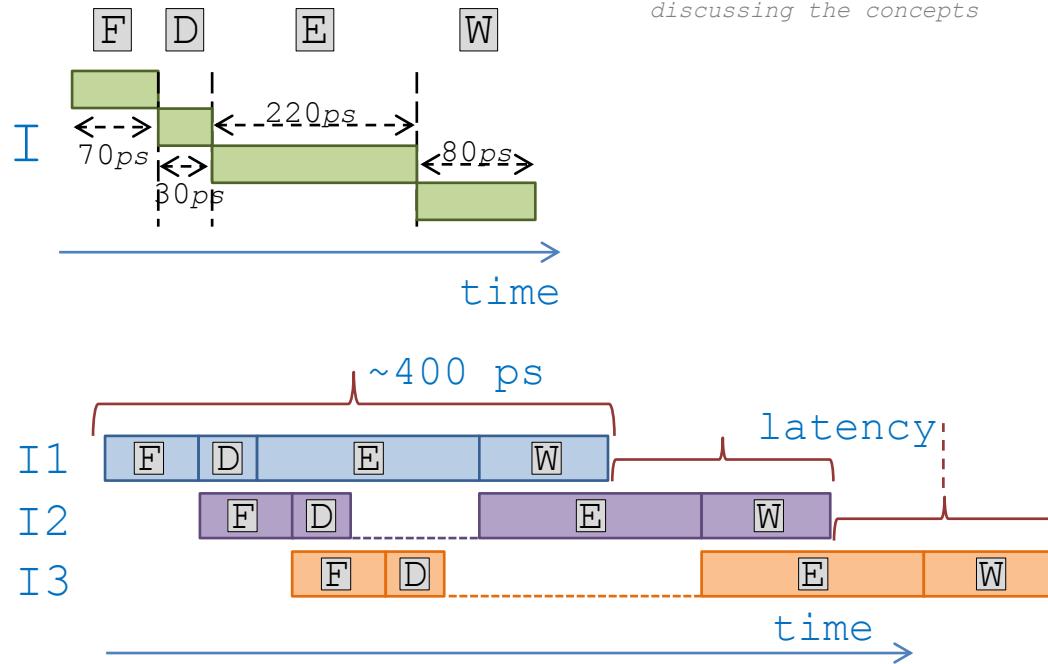
400ps is also the total time required to get a result from an instruction, and so it is the **latency** of the instruction.

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:

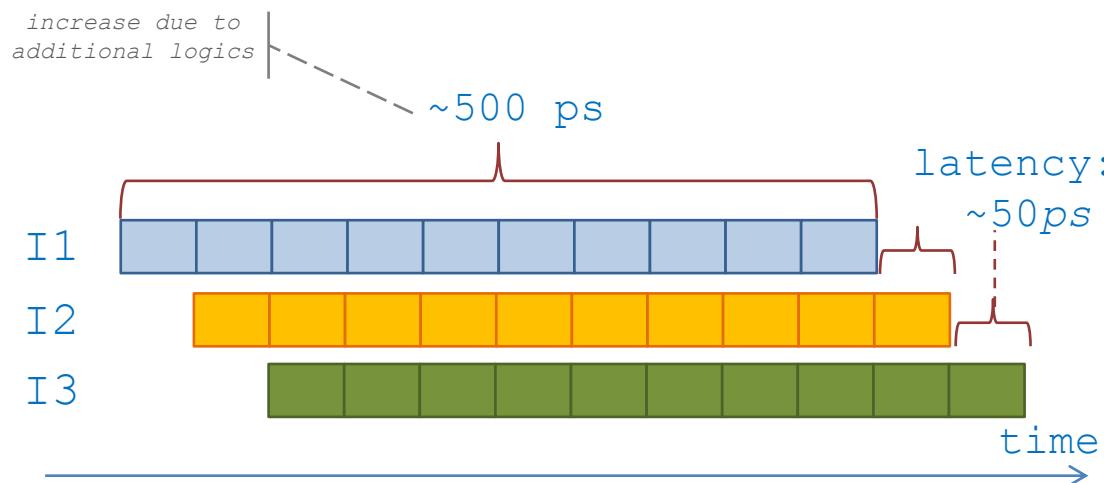
If the stage delays are not uniform, the throughput is limited by the latency $F + (D+E) - (F+D) = E \sim 220\text{ps}$, which means we have a throughput of **~4.5GIPS** just because of logic units separation.



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 50ps, 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.

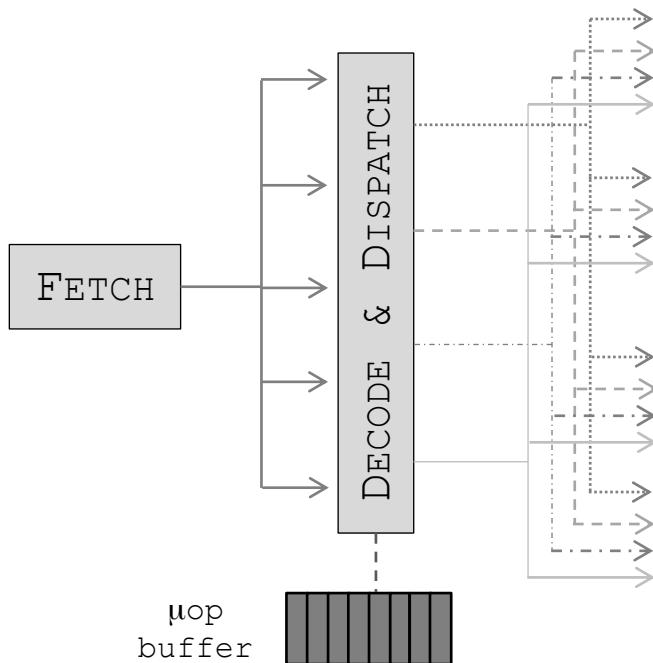
However, there is more that can be done working on the **execution stage** that, of course, encompasses a large number of different *functional units*, performing a **different and independent tasks**.

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



Pipelines

Multiple pipelines

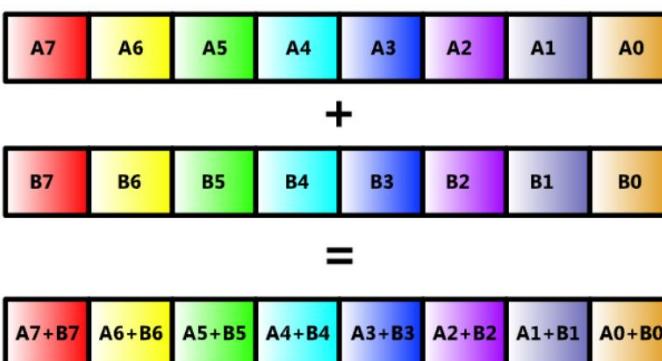




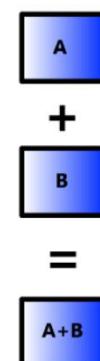
Late 90s: vector capabilities become mainstream

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

SIMD Mode



Scalar Mode



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)

<code>_m128</code>	Float	Float	Float	Float	4x 32-bit float
<code>_m128d</code>	Double		Double		2x 64-bit double
<code>_m128i</code>	B	B	B	B	16x 8-bit byte
<code>_m128i</code>	short	short	short	short	8x 16-bit short
<code>_m128i</code>	int		int	int	4x 32bit integer
<code>_m128i</code>	long long		long long		2x 64bit long
<code>_m128i</code>	doublequadword				1x 128-bit quad

AVX Data Types (16 YMM Registers)

<code>_mm256</code>	Float	Float	Float	Float	Float	Float	Float	Float	8x 32-bit float
<code>_mm256d</code>	Double		Double		Double		Double		4x 64-bit double
<code>_mm256i</code>	<i>256-bit Integer registers. It behaves similarly to <code>_m128i</code>. Out of scope in AVX, useful on AVX2</i>								



Early 00s: multi-core becomes mainstream

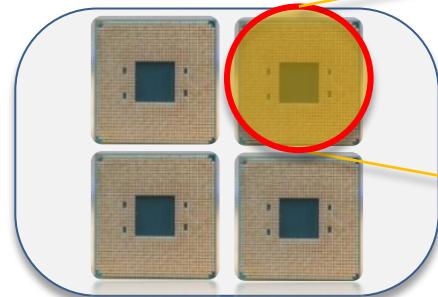
Late 00s : accelerators become mainstream

Early 10s: many-cores + heterogeneous computation
becomes mainstream

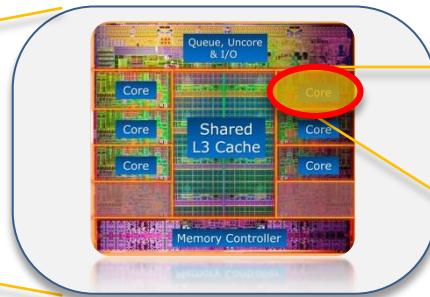
That is the focus of the course on OpenMP/MPI



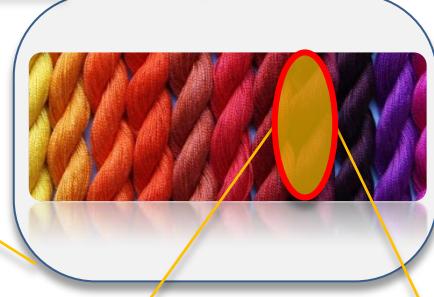
A final sketch



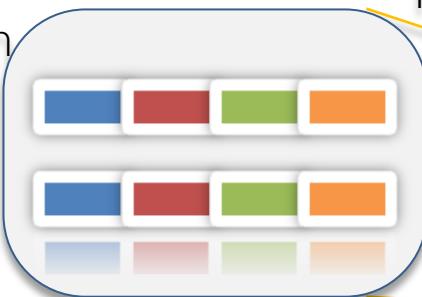
sockets



Many cores
per socket



Many threads



Vectorisation

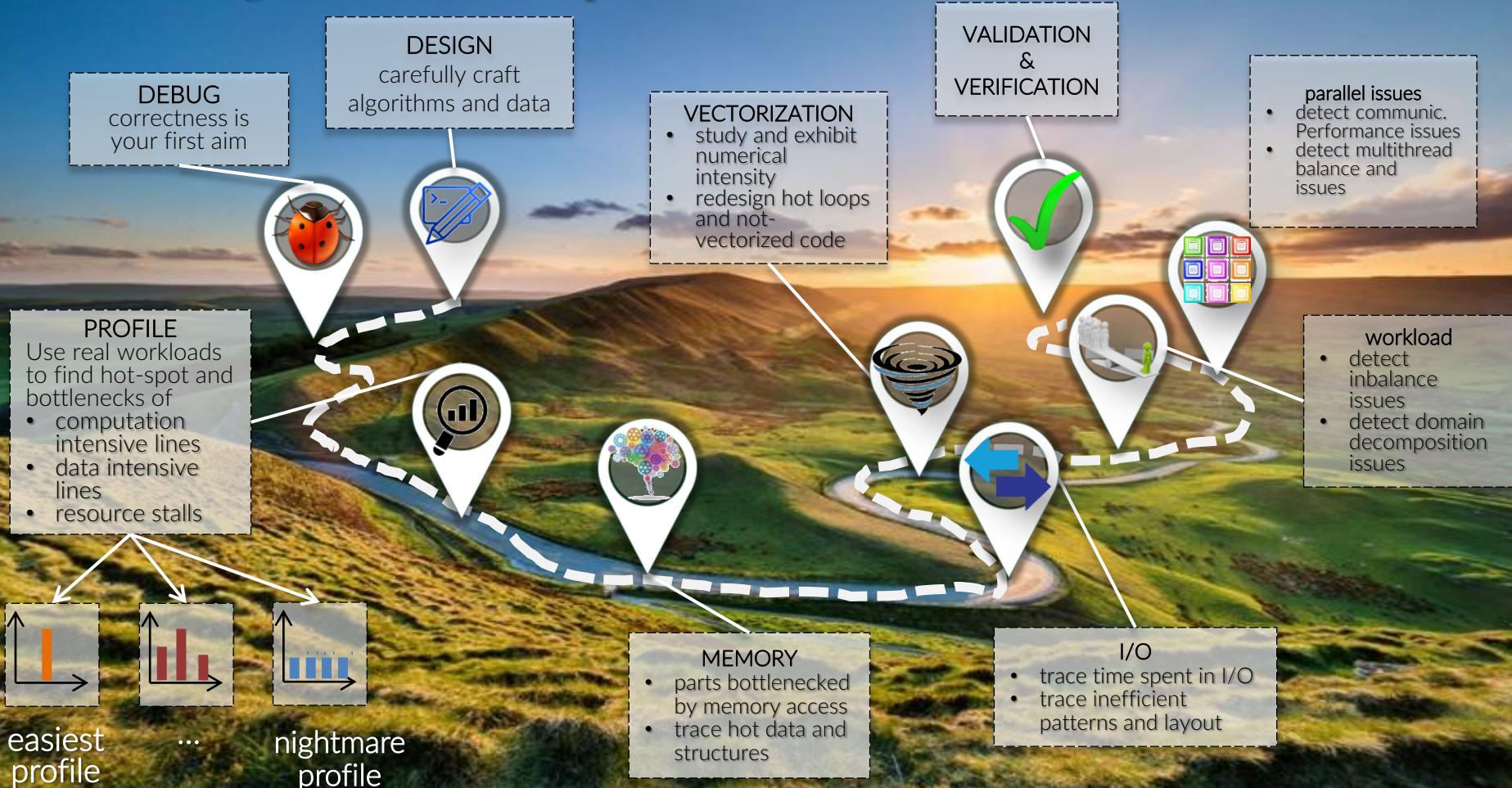


Pipelining



Superscalarity

The winding road towards optimization



that's all, have fun

"So long
and thanks
for all the fish"