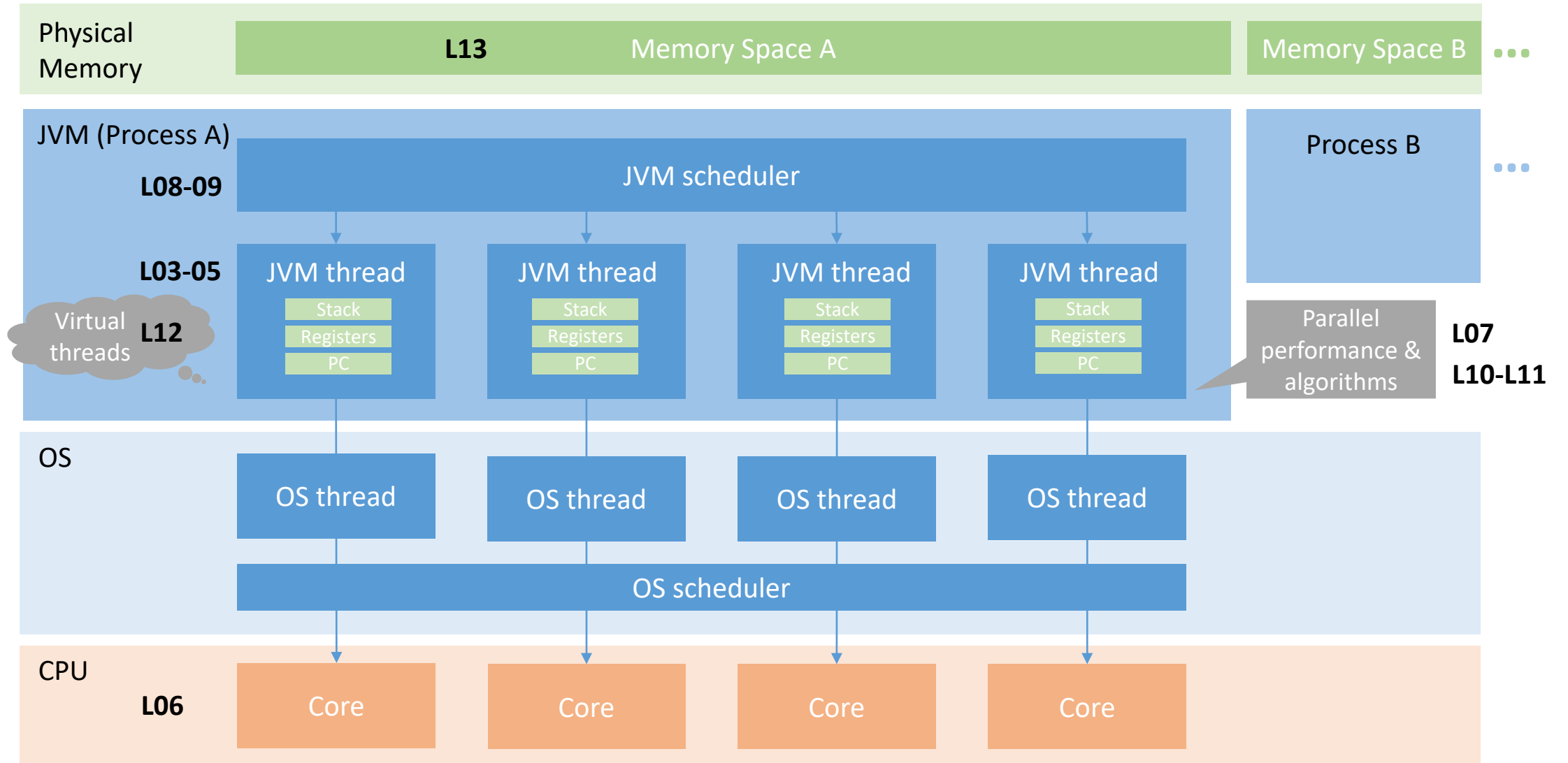


# Parallel Programming

Parallel Architectures: Parallelism on the Hardware Level

# Big Picture (Part I)



# Parallel vs. Concurrent (Recap)

In practice, these terms are often used interchangeably

Key concerns:

**Parallelism:** Use extra resources to solve a problem faster

**Concurrency:** Correctly and efficiently manage access to shared resources

# Parallel and Concurrent vs. Distributed (Preview)

Common assumption for parallel and concurrent:

- one “system”

Distributed computing:

- Physical separation, administrative separation, different domains, multiple systems

Reliability / Fault tolerance is a *big* issue in distributed computing

- Also for parallel computing
- Some of the approaches developed for distributed systems may find their way into parallel systems.

# Motivation for material to come

Get some high-level intuition about:

- Architectural challenges & choices
- Why architects have turned to multicores

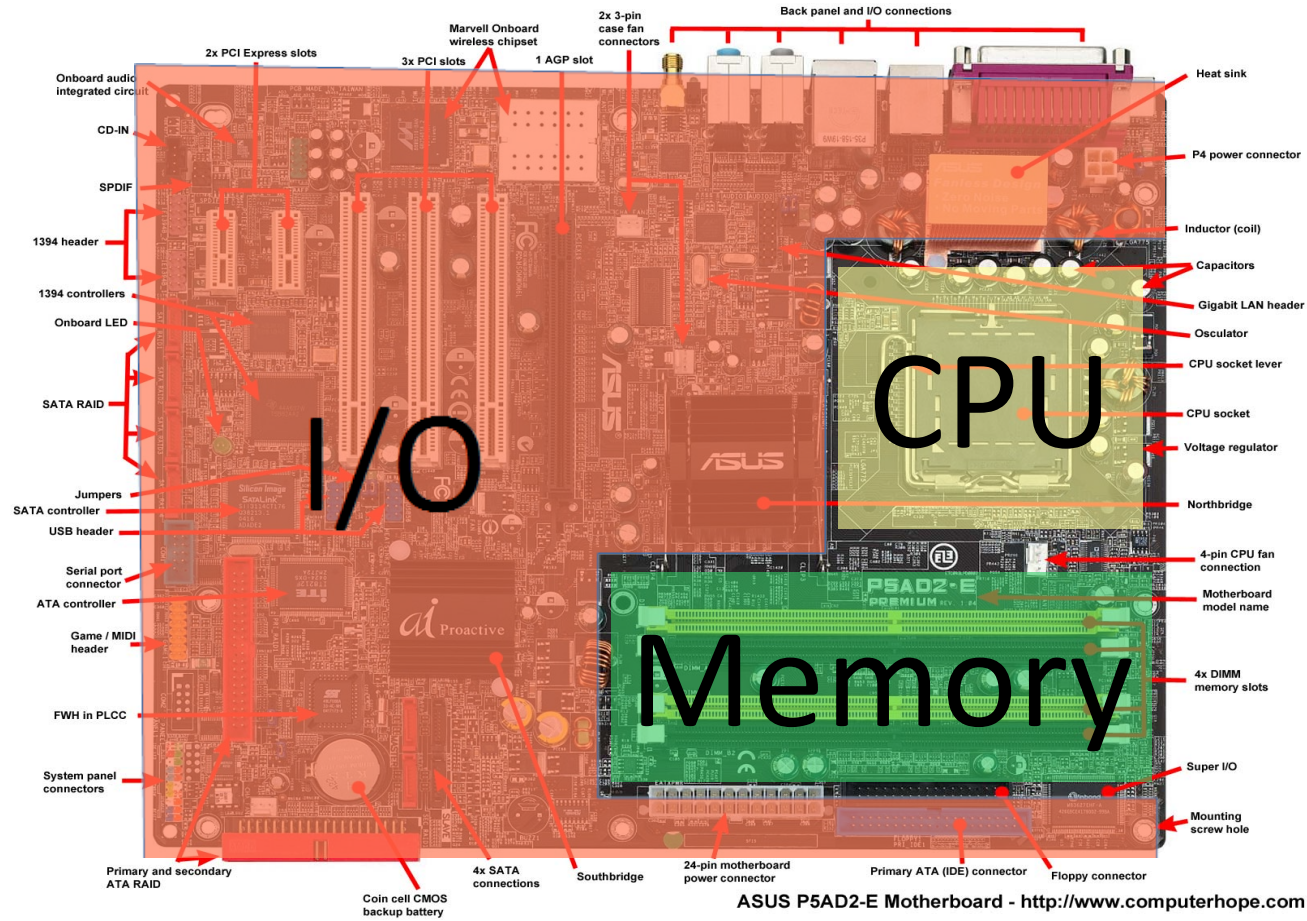
Useful for parallel programming

- Due to performance implications (caches, locality)
- Some challenges & choices transfer to software

# Today's computers: different appearances ...



... similar from the inside



# Basic principles of today's computers

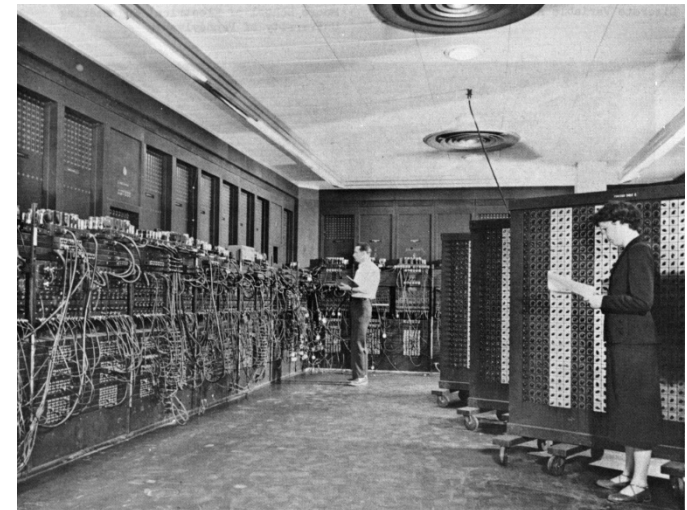
Based on the **Von Neumann architecture** (or **Princeton arch.**):  
program data and program instructions in the same memory

Wasn't always like this: see ENIAC, the first general purpose  
(1945, Turing-complete) computer; used **Harvard arch.**

Von Neumann arch. is simpler:  
one address space (data and code), one bus



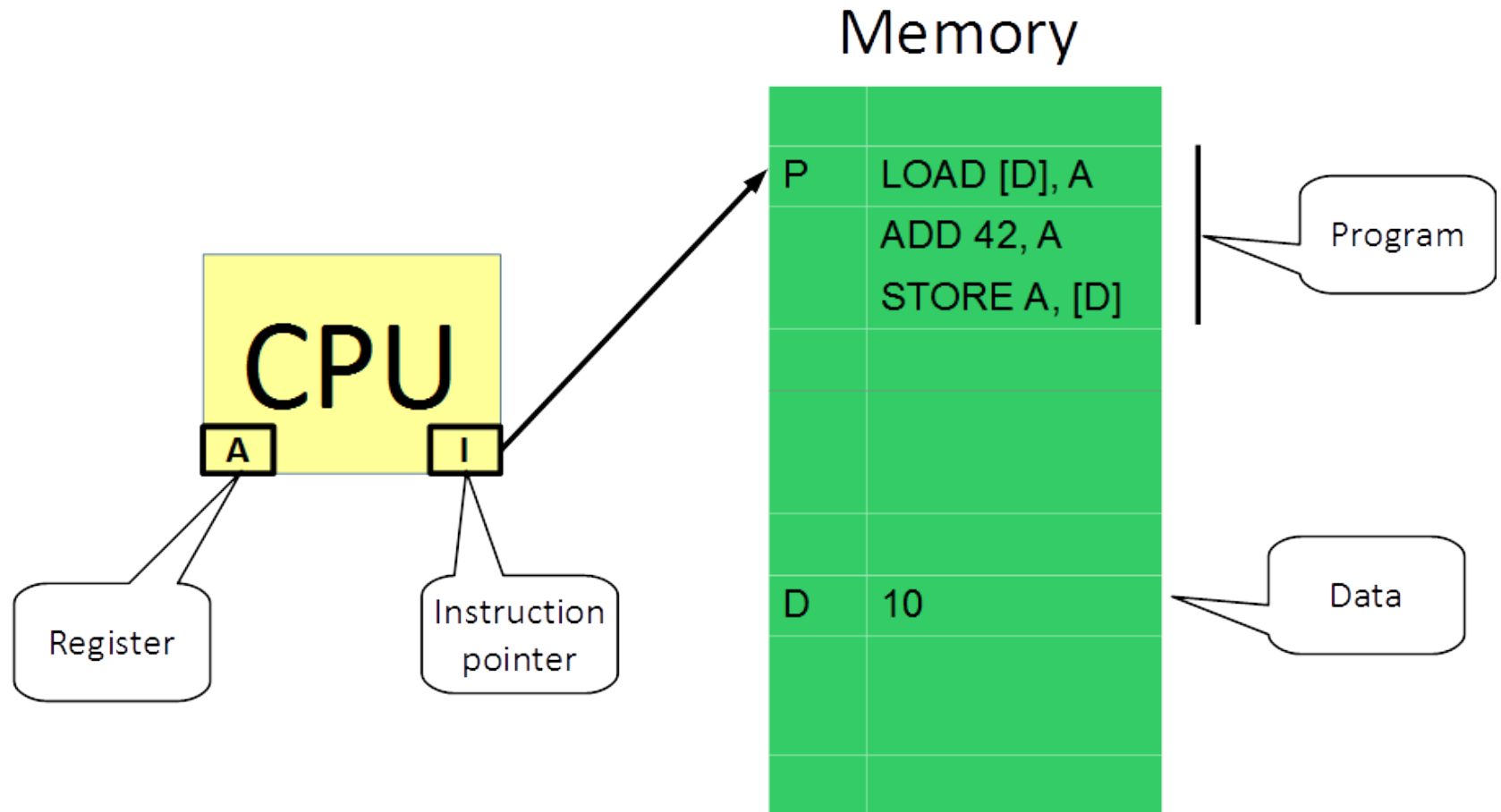
John von Neumann  
(1903-1957).





Von Neumann architecture “matches” imperative programming languages (PL) such as Java: statement executes, then another statement, then a 3<sup>rd</sup> statement, etc...

Have imperative languages been designed in some sense to “match” hardware rather than human thinking?



---

# Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus  
IBM Research Laboratory, San Jose

---



Turing award 1977  
[Paper 1978](#)

John Backus (IBM Research), Turing award winner 1977

co-inventor of Fortran, 1<sup>st</sup> high level imperative programming language  
co-inventor BNF (Backus-Naur Form), used to define formal languages

# CPU Caches

CPUs grew faster

Memories grew bigger

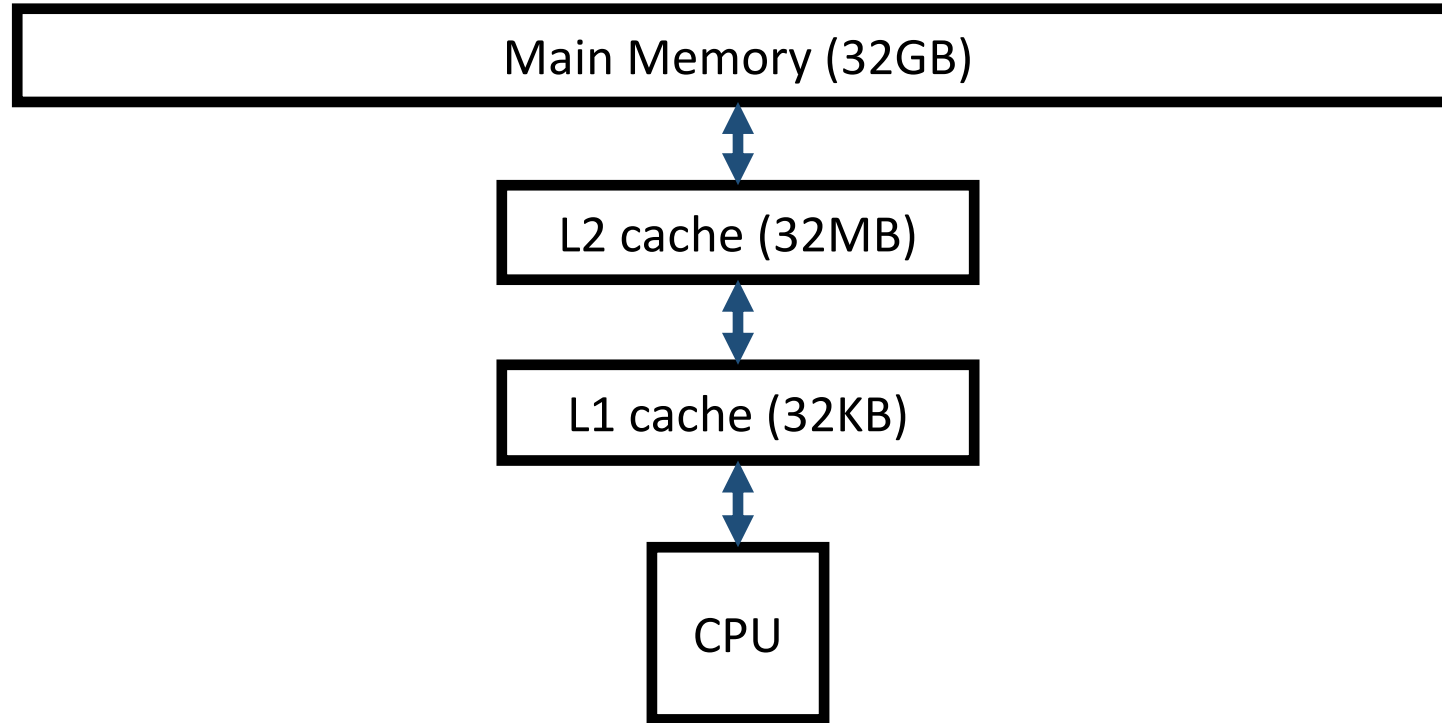
Accessing memory became slower than accessing CPU registers

Locality:

- Data locality/locality of reference: related storage locations (spatial) are often accessed shortly after each other (temporal)
- (Modularity/Encapsulation: reason locally, e.g. one thread at a time)



# CPUs and Memory Hierarchies

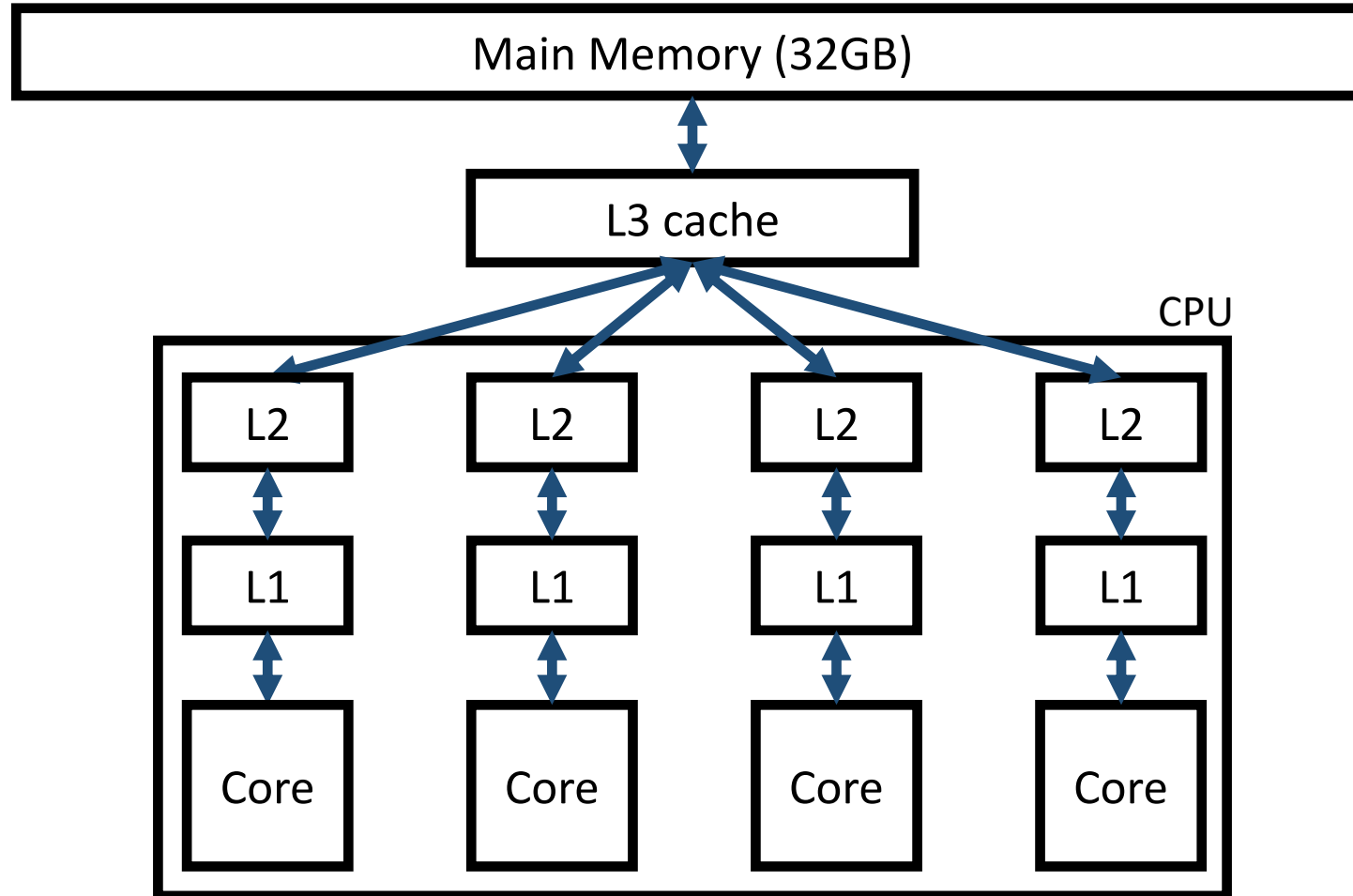


CPU reads/writes values from/to main memory, to compute with them ...

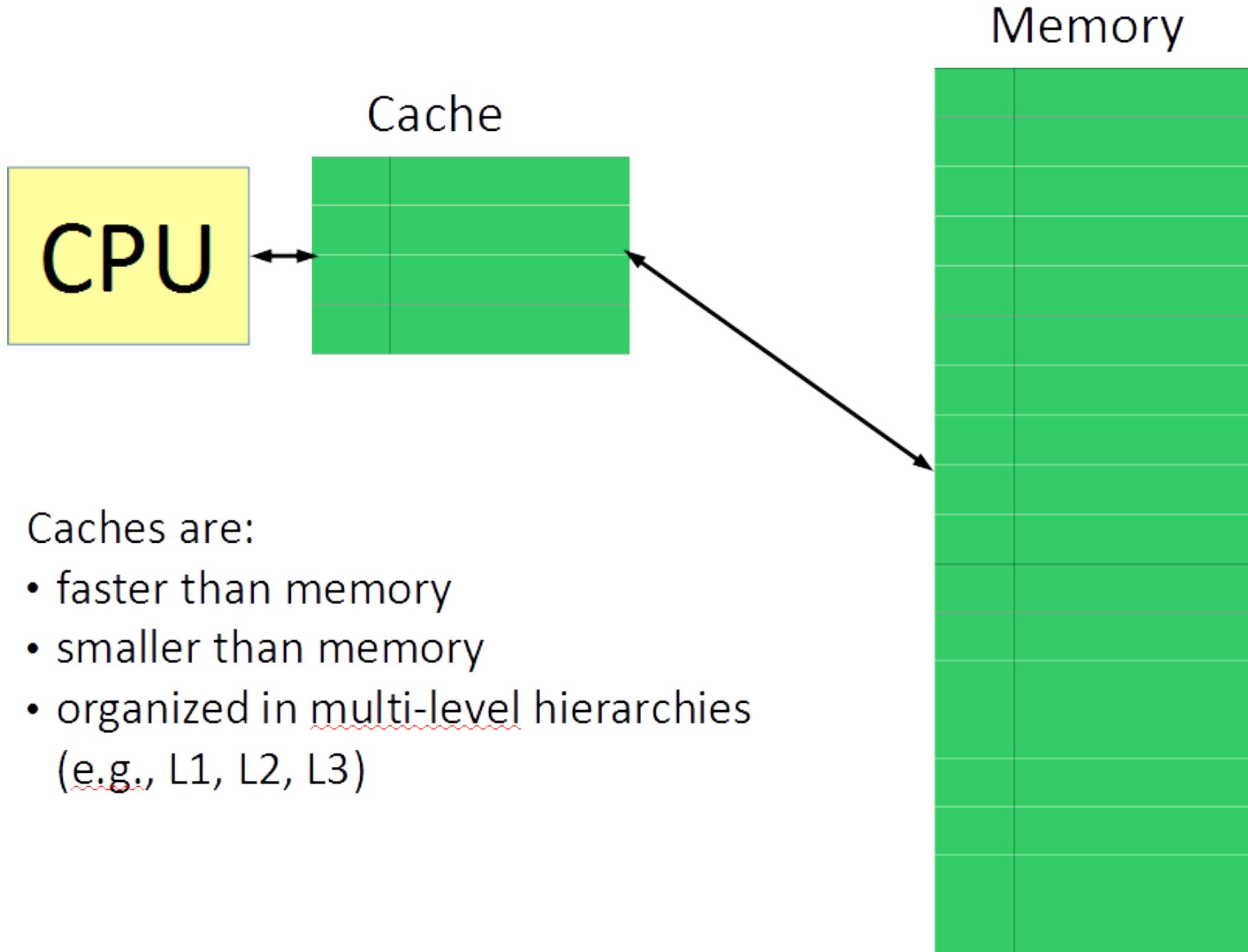
... with a hierarchy of *memory caches* in between

*Faster* memory is more expensive, hence smaller: L1 is 5x faster than L2, which is 30x faster than main memory, which is 350x faster than disk

# CPUs and Memory Hierarchies



Multi-core CPUs have caches per core → more complicated hierarchies



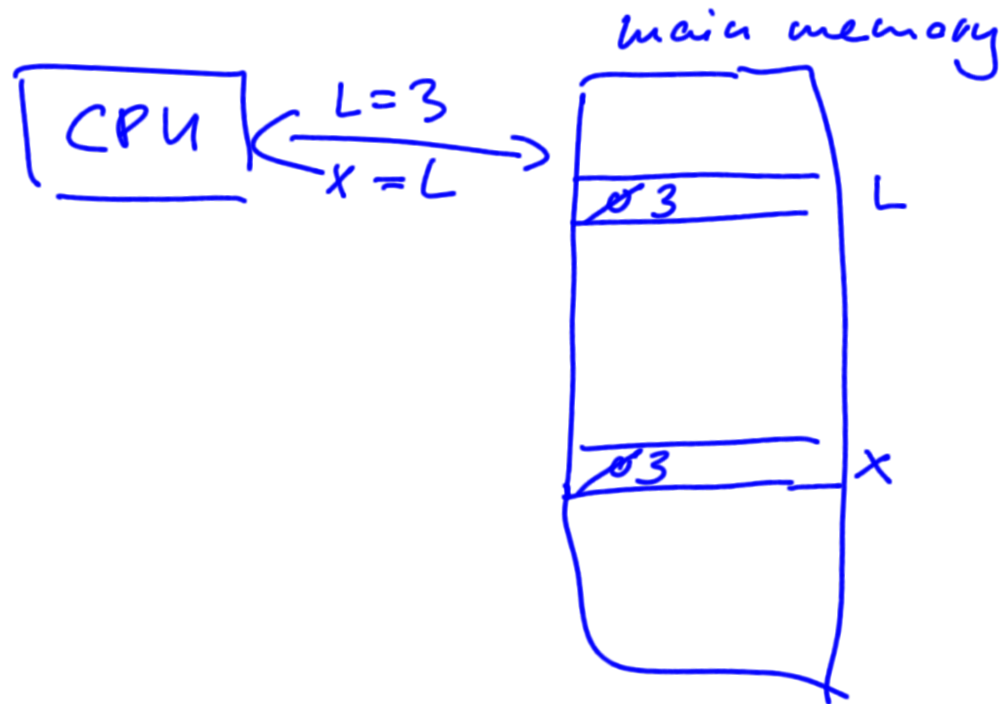
Caches are:

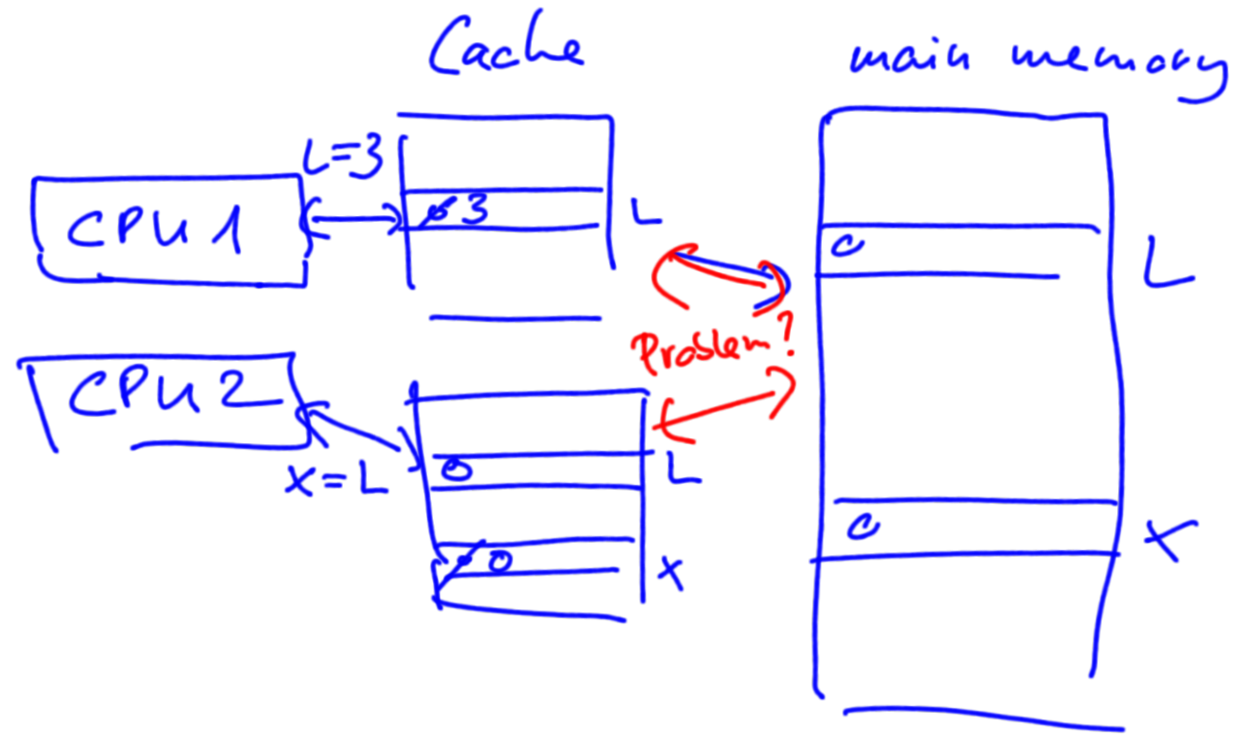
- faster than memory
- smaller than memory
- organized in multi-level hierarchies (e.g., L1, L2, L3)

EXAMPLE:  $1 L = 3$  ||  $S^1$   
 $S$  ||  $4 X = L$

$L$ : shared value

$X$ : thread local variable



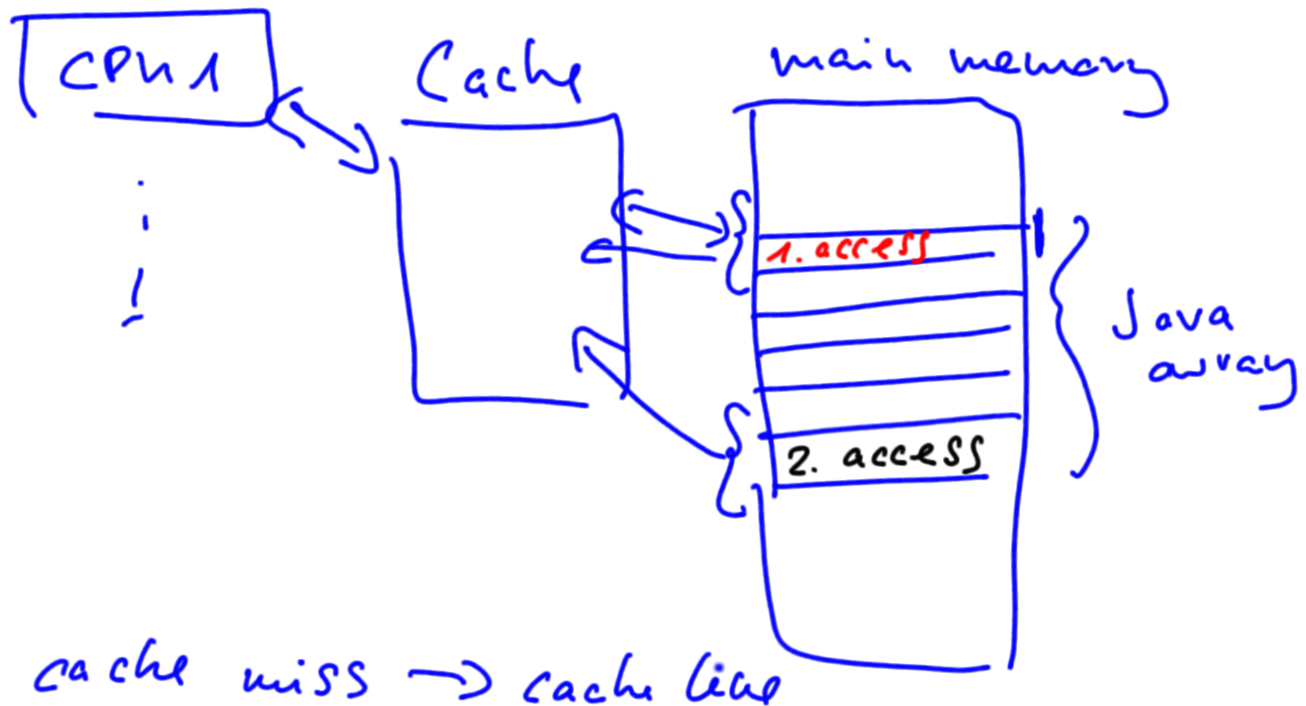


cache coherence protocol

↳ BUT: CPU may postpone writes / prefetches  $\Rightarrow$  optimizations

$\Rightarrow$  memory barriers  $\Rightarrow$  Java: sync





*Code example: 01\_cache\_effects*

How can we make computations faster (on hardware level)?

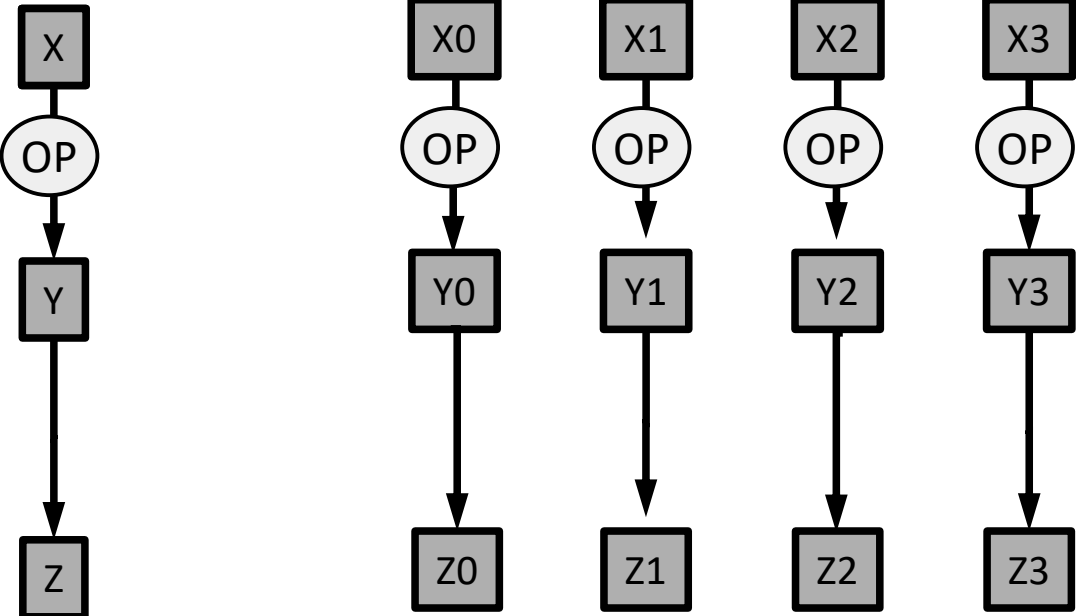
## **Parallel Execution**

I.e., additional execution units (that are actually used)

# 3 approaches to apply parallelism to improve sequential processor performance

- *Vectorization*: Exposed to developers
- *Instruction Level Parallelism (ILP)*: Inside CPU
- *Pipelining*: Also internal, but transfers to software

# Vectorization



Single Instruction (OP),  
applied to Multiple Data  
=  
SIMD

# Example: adding vectors X and Y

Step 1: load (mem->registers)

Step 2: Operation

Step 3: store (registers->mem)

Standard way: 1-at-a-time


Vectorized way: N-at-a-time

X	$X_0$
	$X_1$
	$X_2$
	$X_3$
Y	$Y_0$
	$Y_1$
	$Y_2$
	$Y_3$

# VECTORISE

a 

+

b 

=

c 

embarrassingly parallel

↳ supported by hardware vectorization

seq.  
for (int i=0; i < 16; ++i)

$$c[i] = a[i] + b[i]$$

loop unrolling

seq.

for (int i=0; i < 16; i+=4) {

$$c[i+0] = a[i+0] + b[i+0];$$

$$c[i+1] = \dots$$

⋮

$$c[i+3] = \dots$$

}

## ASSEMBLER PSEUDOCODE

r1 = VLOAD a, i, i+3

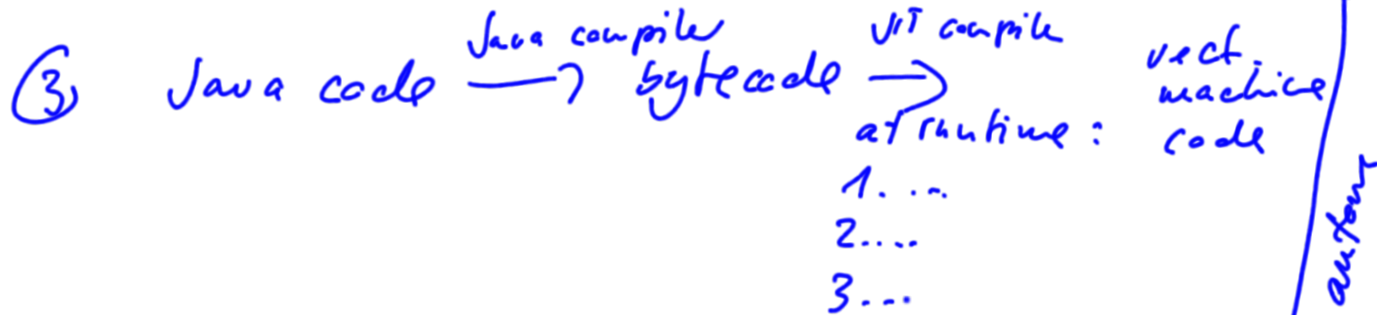
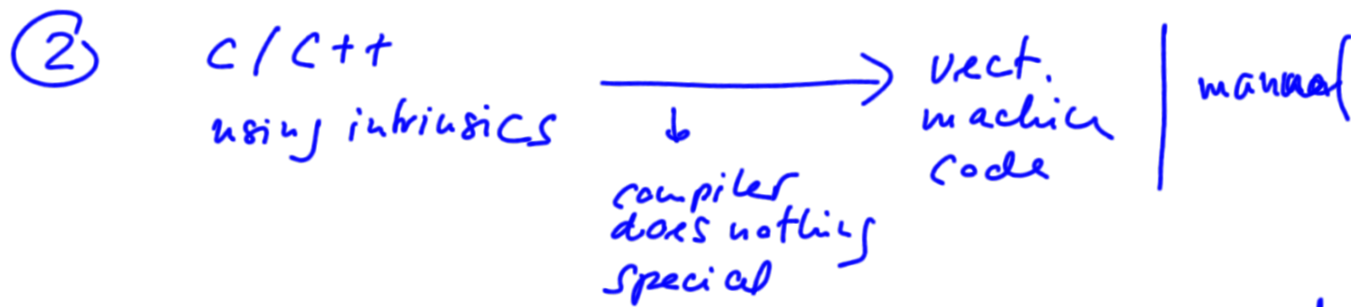
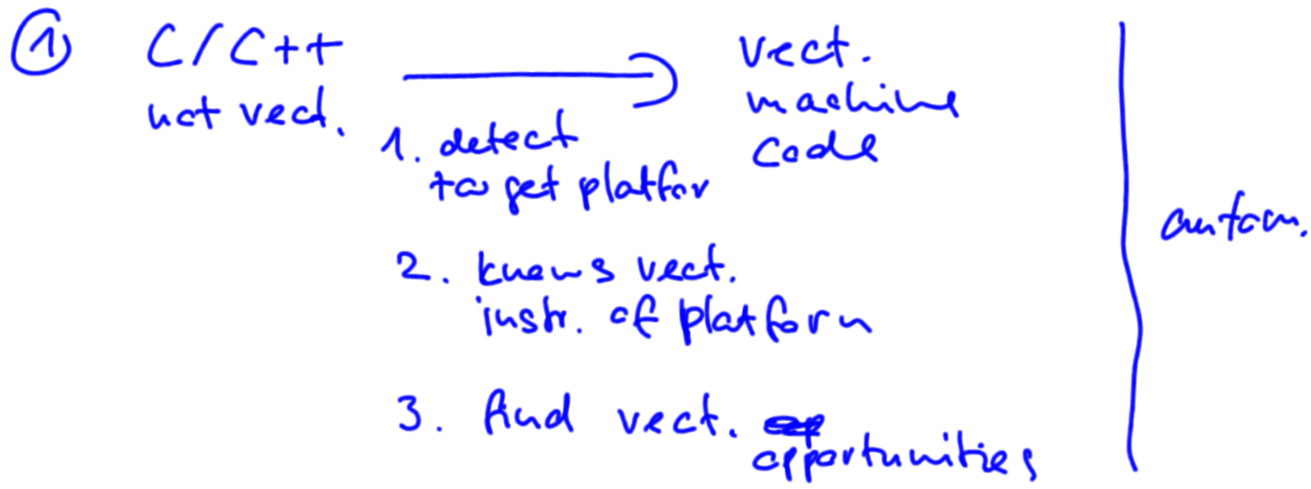
r2 = VLOAD b, i, i+3

r3 = VADD r1, r2

VSTORE = c, i, r3

*Code example: 02\_gcc\_vectorize*





# 3 approaches to apply parallelism to improve sequential processor performance

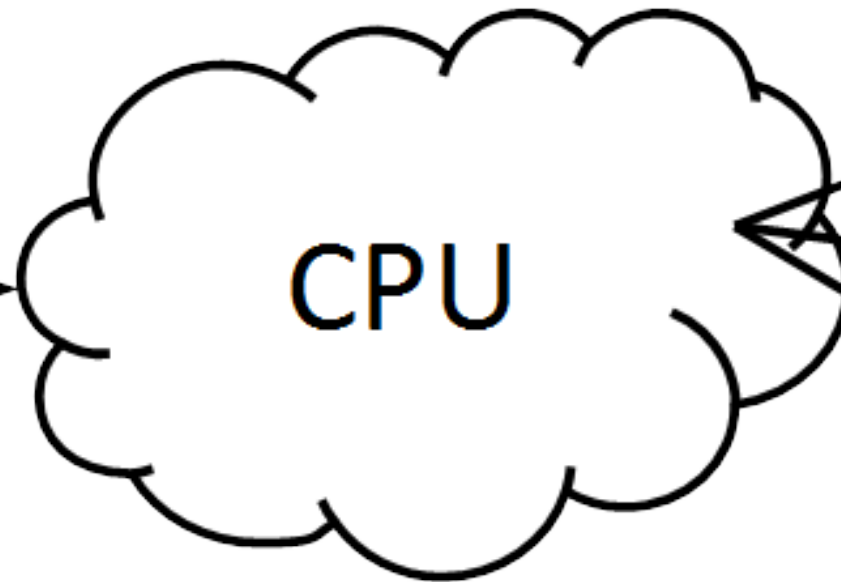
- Vectorization
- Instruction Level Parallelism (ILP)
- Pipelining

# Modern CPUs exploit Instruction Level Parallelism (ILP)

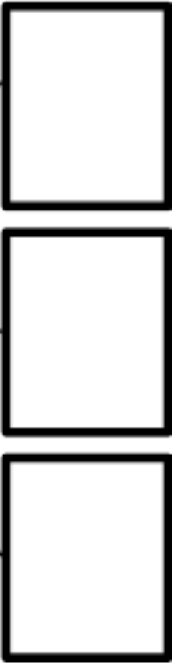
Instruction stream  
(sequential program)

```
LOAD [D], X
ADD 42, X
ADD 23, Y
ADD 54, Z
ADD 10, W
ADD Y, X
ADD W, Z
ADD Z, X
STORE X, [D]
```

Appears  
as if program  
was executed  
sequentially



Parallel execution,  
(multiple hardware units)



# ILP: a very simple example

Consider the following program:

```
1: e = a + b    // this and the one below are independent
2: f = c + d    // can execute these 2 instructions in parallel
3: m = e * f    // this one depends on results above, so has to wait
```

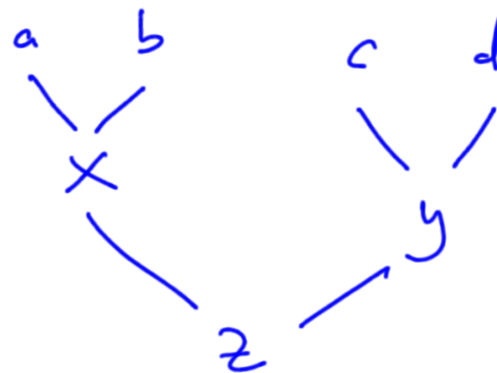
Independent, if

- different register names
- different memory addresses

# ILP

$$x = a + b; \quad y = c + d; \quad z = x \cdot y$$

1) determine dependencies



2) parallelize across  
CPU execution units

( $\neq$  cores)

$$(x = a + b \parallel y = c + d);$$
$$z = x \cdot y$$

prefetching  $\Rightarrow$  increase opportunities for ILP

- 1) speculative execution
- 2) reordering

*Code example: 03\_reordering*

# Instruction Level Parallelism (ILP)

- Enable ILP: Superscalar CPUs
  - Multiple instructions per cycle / multiple functional units
- Increase opportunities for ILP:
  - Speculative execution
    - Predict results to continue execution
  - Out-of-Order (OoO) execution
    - Potentially change execution order of instructions
    - As long as the programmer observes the sequential program order
  - Pipelining → next

# 3 approaches to apply parallelism to improve sequential processor performance

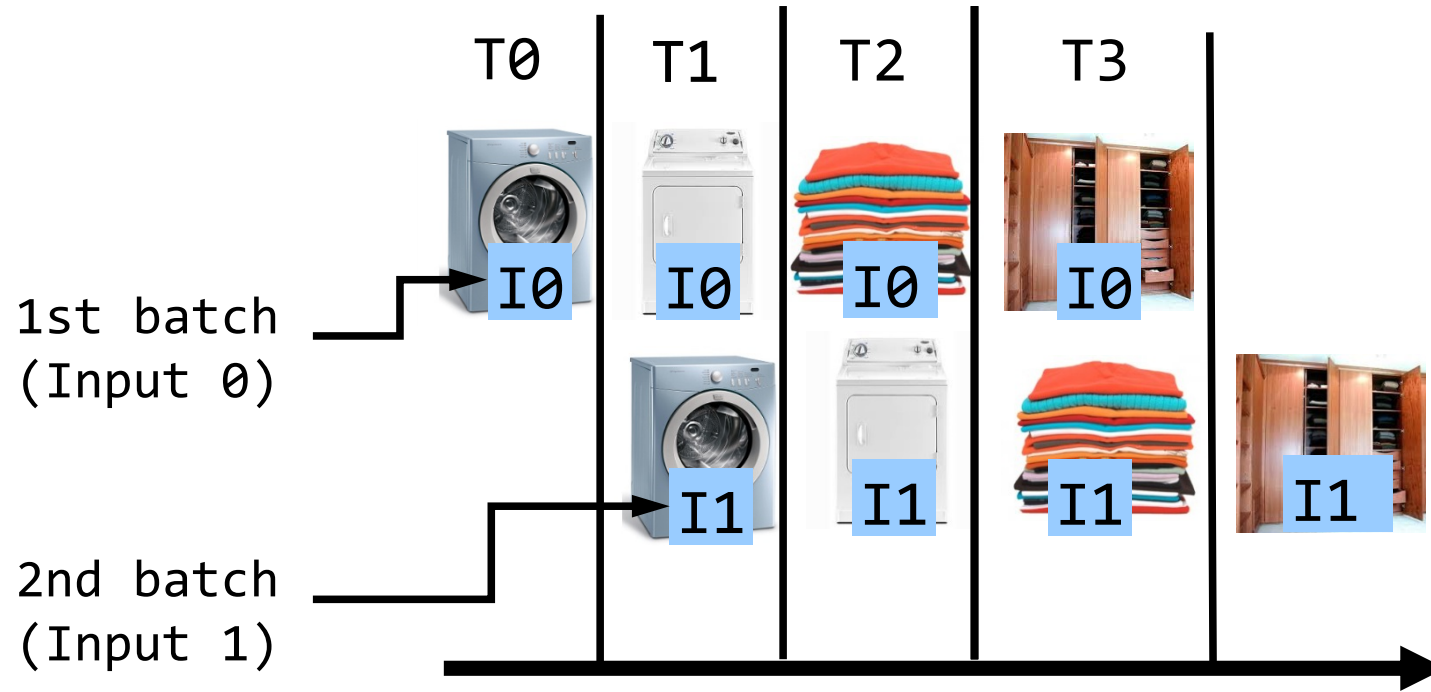
- Vectorization
- Instruction Level Parallelism (ILP)
- **Pipelining**



# Washing clothes



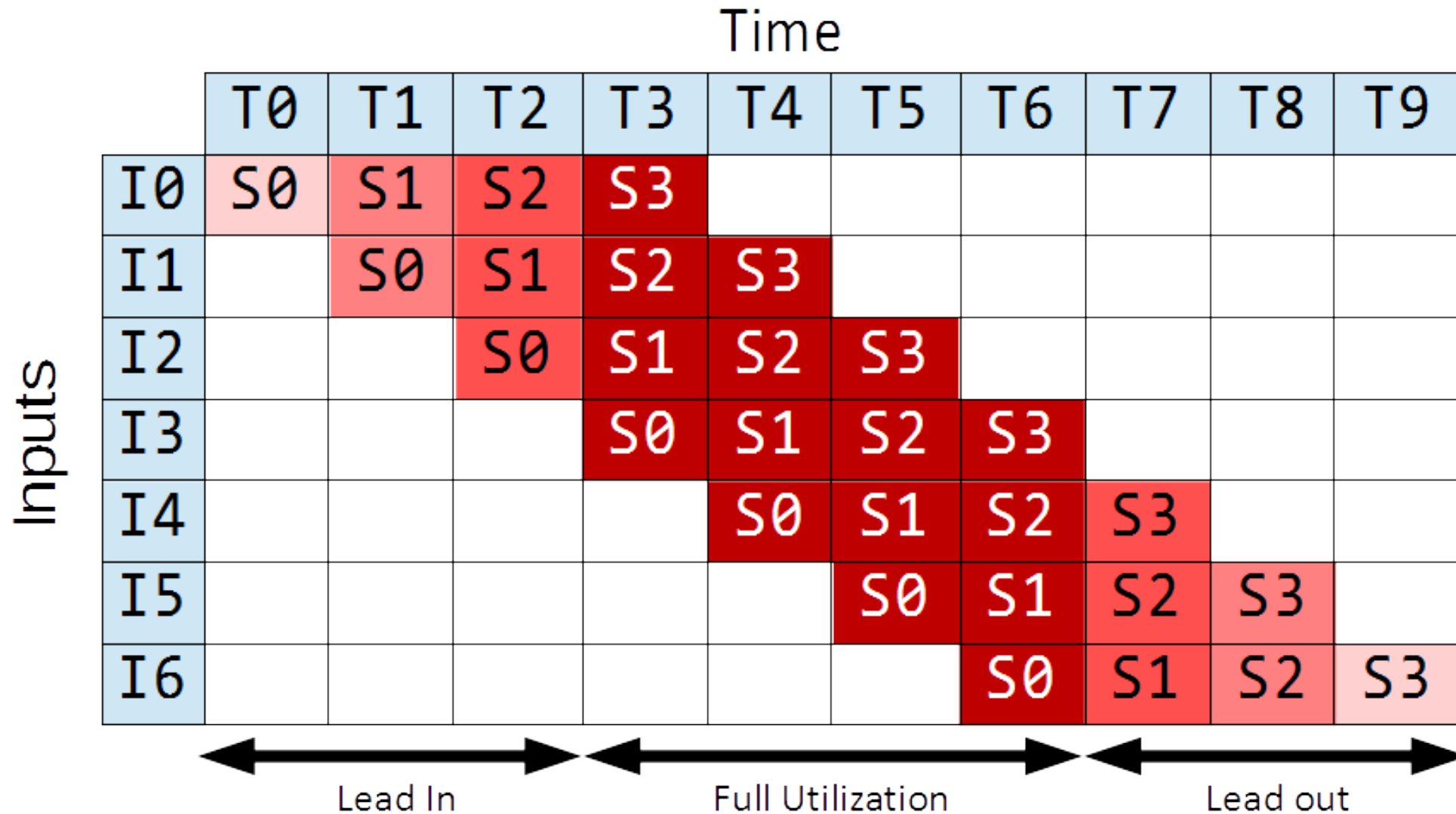
# Washing clothes – Pipeline



- Additional functional units
- Multiple “inputs” (stream)

# Balanced Pipeline

balanced = all steps require same time



# Pipeline Characteristics/Metrics

- **Throughput** = amount of work that can be done by a system in a given period of time
- **Latency** = time needed to perform a given computation (e.g., a CPU instruction)

More exists, e.g. bandwidth (amount of work done in parallel)

# Throughput

- Throughput = amount of work that can be done by a system in a given period of time
- In CPUs: # of instructions completed per second
- Larger is better

$$\text{Throughput bound} = \frac{1}{\max(\textit{computationtime}(\textit{stages}))}$$

(ignoring lead-in and lead-out time in pipeline with large number of states; cannot do better than this)

# Latency

- Latency = time to perform a computation (e.g., a CPU instruction)
- In CPU: time required to execute a single instructions in the pipeline
- Lower is better

Latency bound =  $\#stages \cdot \max(\textit{computationtime}(stages))$

- Pipeline latency only *constant over time if pipeline balanced*: sum of execution times of each stage

# Washing clothes – Unbalanced Pipeline



Takes 5 seconds. We use “w” for Washer next.



Takes 10 seconds. We use “d” for Dryer next.



Takes 5 seconds. We use “f” for Folding next.



Takes 10 seconds. We use “c” for Closet next.

# Designing a pipeline: 1<sup>st</sup> Attempt (lets consider 5 washing loads)

Time (s) Load #	0	5	10	15	20	25	30	35	40	45	50	55	60	65	70
Load 1	w	d	d	f	c	c									
Load 2		w	_	d	d	f	c	c							
Load 3			w	_	_	d	d	f	c	c					
Load 4				w	_	_	_	d	d	f	c	c			
Load 5					w	_	_	_	_	d	d	f	c	c	

The total time for all 5 loads is **70 seconds**.

This pipeline can work, however it cannot bound the latency of a Load as it keeps growing. If we want to bound this latency, one approach is to make each stage take as much time as the longest one, thus balancing it. In our example, the longest time is 10 seconds, so we can do the following:



# Make Pipeline balanced by increasing time for each stage to match longest stage



Now takes 10 seconds.



Takes 10 seconds, as before.



Now takes 10 seconds.



Takes 10 seconds, as before.

# Designing a pipeline: 2<sup>nd</sup> Attempt

Time (s) Load #	0	10	20	30	40	50	60	70	80	90	100	110	60	65	70	75	80	85	90
Load 1	w	d	f	c															
Load 2		w	d	f	c														
Load 3			w	d	f	c													
Load 4				w	d	f	c												
Load 5					w	d	f	c											

This pipeline is a bit wasteful, but the latency is bound at **40 seconds** for each Load. Throughput here is about 1 load / 10 seconds, so about 6 loads / minute.

So now we have the total time for all 5 loads at **80 seconds**, higher than before.

Can we somehow get a bound on latency while improving the time/throughput?

# Step 1: make the pipeline from 1<sup>st</sup> attempt a bit more fine-grained:



Like in the 1<sup>st</sup> attempt, this takes 5 seconds.



Lets have 2 dryers working in a row. The first dryer is referred to as **d1** and takes 4 seconds, the second as **d2** and takes 6 sec.



Like in the 1<sup>st</sup> attempt, it takes 5 seconds.



Lets have 2 closets working in a row. The first closet is referred to as **c1** and takes 4 seconds, the second as **c2** and takes 6 sec.

Step 2: and also, like in the 2<sup>nd</sup> pipeline, make each stage take as much time as the longest stage does from Step 1 [this is 6 seconds due to d2 and c2]



It now takes 6 seconds.



Each of d1 and d2 dryers take 6 seconds.



Now takes 6 seconds.



Each of c1 and c2 closets now take 6 seconds.

# Designing a pipeline: 3<sup>rd</sup> Attempt (lets consider 5 washing loads)

Time (s)	0	6	12	18	24	30	36	42	48	54	60	110	60	65	70	75	80	85	90
Load #																			
Load 1	w	d1	d2	f	c1	c2													
Load 2		w	d1	d2	f	c1	c2												
Load 3			w	d1	d2	f	c1	c2											
Load 4				w	d1	d2	f	c1	c2										
Load 5					w	d1	d2	f	c1	c2									

The bound on latency for each load is now:  $6 * 6 = 36$  seconds.

The throughput is approximately:  $1 \text{ load} / 6 \text{ seconds} = \sim 10 \text{ loads} / \text{minute}$ .

The total time for all 5 loads is **60 seconds**.

# Throughput vs. Latency

Throughput optimization may increase the latency: in our 3<sup>rd</sup> pipeline attempt, we split the dryers into 2, but it could be that the split of 'd' into d1 and d2 leads to higher times for d1 and d2 than 4 and 6.

Pipelining typically adds constant time overhead between individual stages (synchronization, communication)

⇒ Infinitely small pipeline steps not practical

⇒ Time it takes to get one complete task through the pipeline may take longer than with a serial implementation

# CPU Pipeline (Classical RISC)

(remaining slides not exam relevant)

Instr. Fetch

Instr. Decode

Execution

Mem. access

Writeback

Multiple stages (CPU functional units)

- Each instruction takes 5 time units (cycles)
- 1 instr. / cycle (not always possible, though)

Parallelism (multiple hardware functional units)

Leads to faster execution of sequential programs

Actual pipelines potentially much more complicated

# CPU Pipeline (Classical RISC)

Instr. Fetch

Instr. Decode

Execution

Mem. access

Writeback

- Fetches next instruction from memory (CPU's instruction pointer)
- May prefetch additional instructions (ILP, speculative execution)



# CPU Pipeline (Classical RISC)

Instr. Fetch

Instr. Decode

Execution

Mem. access

Writeback

Prepares instructions for execution, e.g.

- Decodes bit sequence 01101... into ADD A1 A2
- “Understands” registers denoted by A1/A2

(RISC instructions either transfer data between memory and CPU registers, or compute on registers)

# CPU Pipeline (Classical RISC)

Instr. Fetch

Instr. Decode

Execution

Mem. access

Writeback

Executes decoded instruction (in CPU, no data transfer yet)

# CPU Pipeline (Classical RISC)

Instr. Fetch

Instr. Decode

Execution

Mem. access

Writeback

Exchange data between CPU and memory  
(if needed, depending on executed instruction)

# CPU Pipeline (Classical RISC)

Instr. Fetch

Instr. Decode

Execution

Mem. access

Writeback

Updates registers

(if needed, depending on executed instruction; also special registers such as flags)

# For a long time...

- CPU architects improved sequential execution by exploiting Moore's law and ILP
- more transistors → used for add. CPU pipeline stages → more performance
- sequential programs were becoming exponentially faster with each new CPUs
  - Most programmers did not worry about performance
  - They waited for the next CPU generation



# But architects hit walls

- power (dissipation) wall
  - faster CPU → consumes more energy → expensive to cool
- memory wall
  - CPUs faster than memory access
- ILP wall
  - Limits in inherent program's ILP, complexity



**no longer affordable to  
increase sequential CPU performance**

# Multicore processors

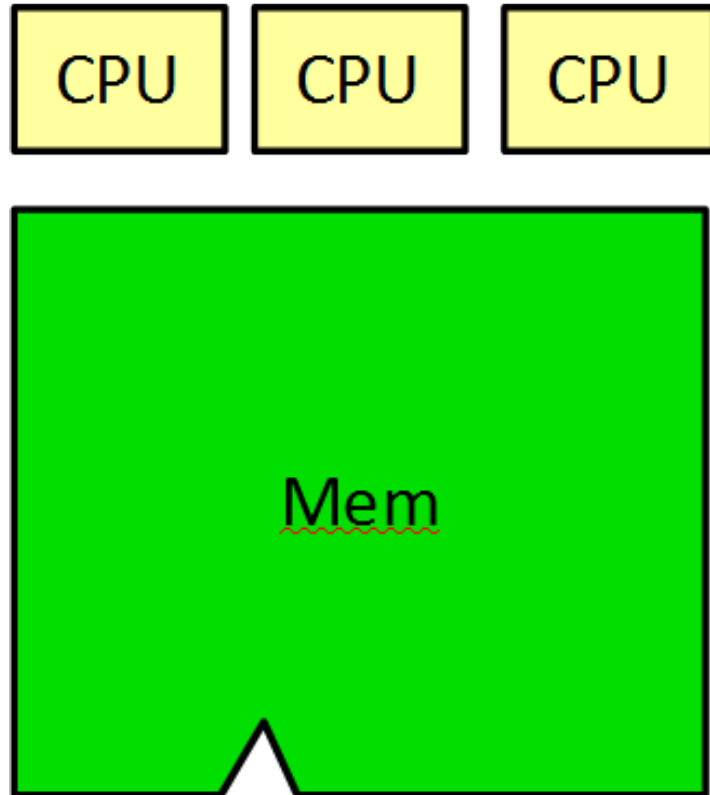
- Use transistors to add cores: “externalize” parallelization to developers ...
- ... instead of improving sequential performance: “internalize” parallelization to hardware designers
- **Expose parallelism to software**
- **Implication: programmers need to write parallel programs to take advantage of new hardware**
  - Past: parallel programming was performed by select few
  - Now (since 2008): ETH teaches parallel programming in the first year → programmers need to worry about (parallel) performance

# Parallel Architectures



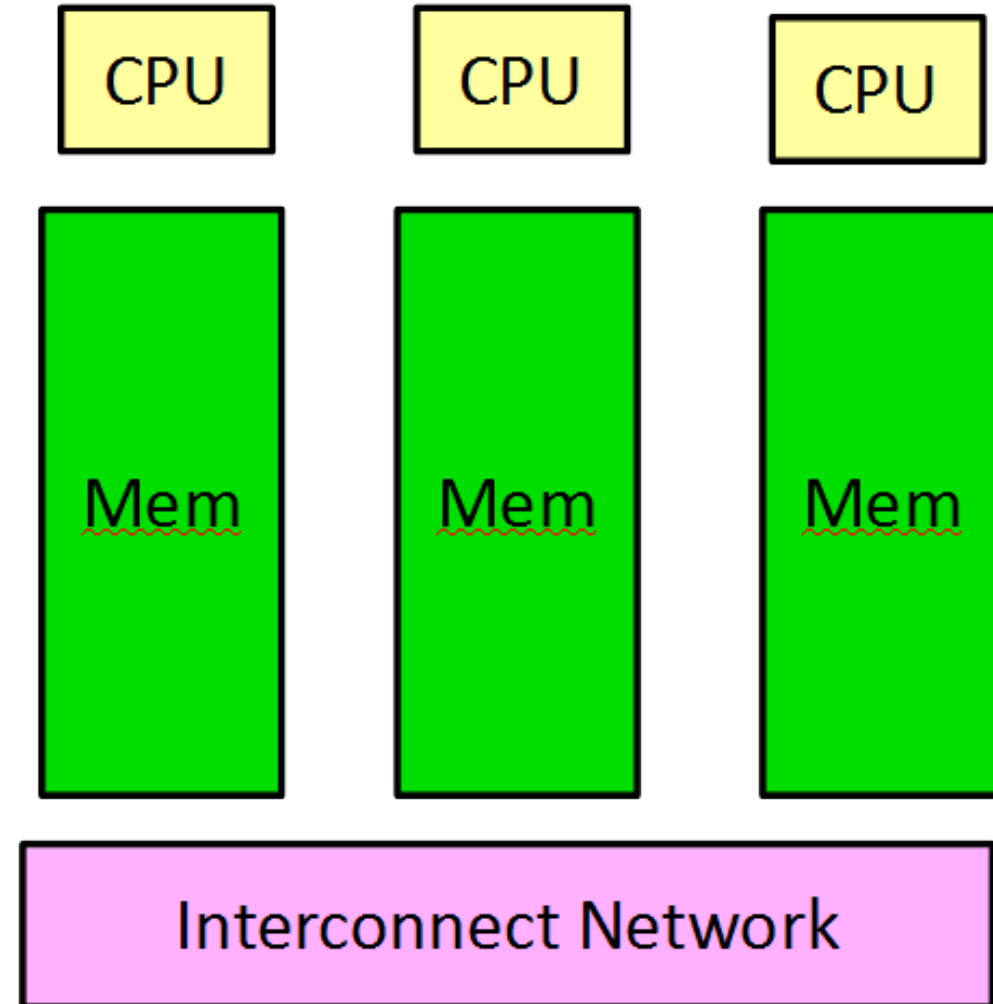
# Shared / Distributed memory architectures

Shared Memory



Main focus  
of course

Distributed Memory



# Shared memory architectures

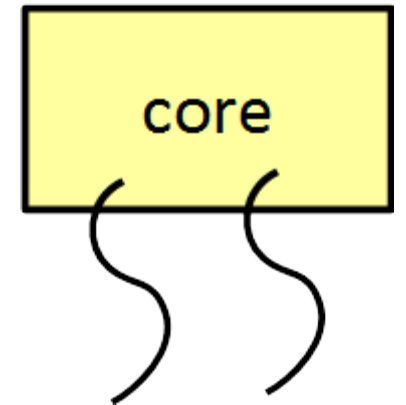
- Simultaneous Multithreading (Hyper-Threading)
- Multicores
- Symmetric Multiprocessor System
- Non-Uniform Memory Access



less  
resource  
sharing at  
hardware  
level

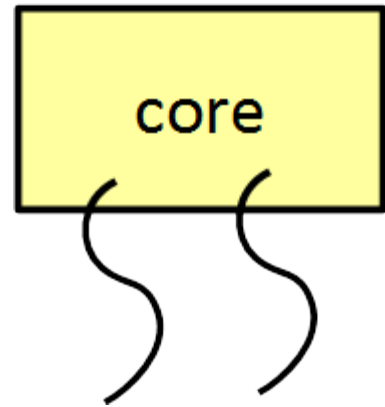
# Simultaneous Multithreading

- Single *physical* core, but multiple *logical/virtual* cores (to OS)
- Certain pipeline steps (e.g. decode) duplicated
- Multiple instruction streams (called threads)
- Between ILP and multicore
  - ILP: multiple units for one instr. stream
  - SMT: multiple units for multiple instr. streams
  - Multicore: completely duplicated cores
- Limited parallel performance, but can increase pipeline utilization if CPU would otherwise have to wait for memory (CPU stalling)



# Intel's Hyper-threading

- First mainstream attempt to expose parallelism to developers
- Push users to structure their software in parallel units
- Motivation for developers: potentially gain performance (not “only” reactivity)



System Information

File Edit View Help

System Summary

- Hardware Resources
- Components
- Software Environment

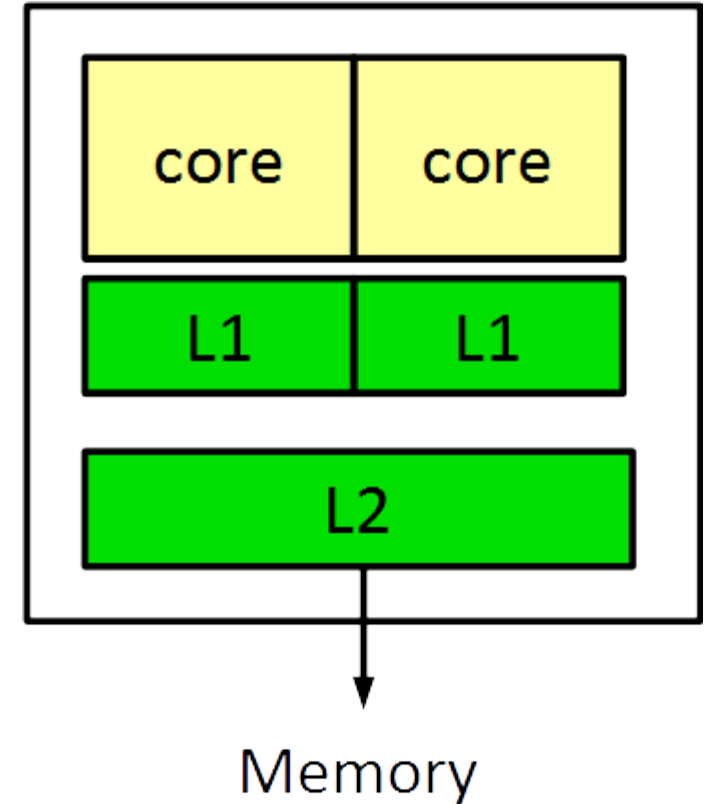
Item	Value
OS Name	Microsoft Windows 10 Education
Version	10.0.17134 Build 17134
Other OS Description	Not Available
OS Manufacturer	Microsoft Corporation
System Name	MALTE
System Manufacturer	
System Model	
System Type	x64-based PC
System SKU	
Processor	Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz, 3401 Mhz, 4 Core(s), 8 Logical Processor(s)
BIOS Version/Date	Intel Corp. BAP6710H.86A.0072.2011.0927.1425, 27-Sep-11
SMBIOS Version	2.7
Embedded Controller Version	255.255
BIOS Mode	Legacy
BaseBoard Manufacturer	Intel Corporation
BaseBoard Model	Not Available

Find what:

Search selected category only  Search category names only

# Multicores

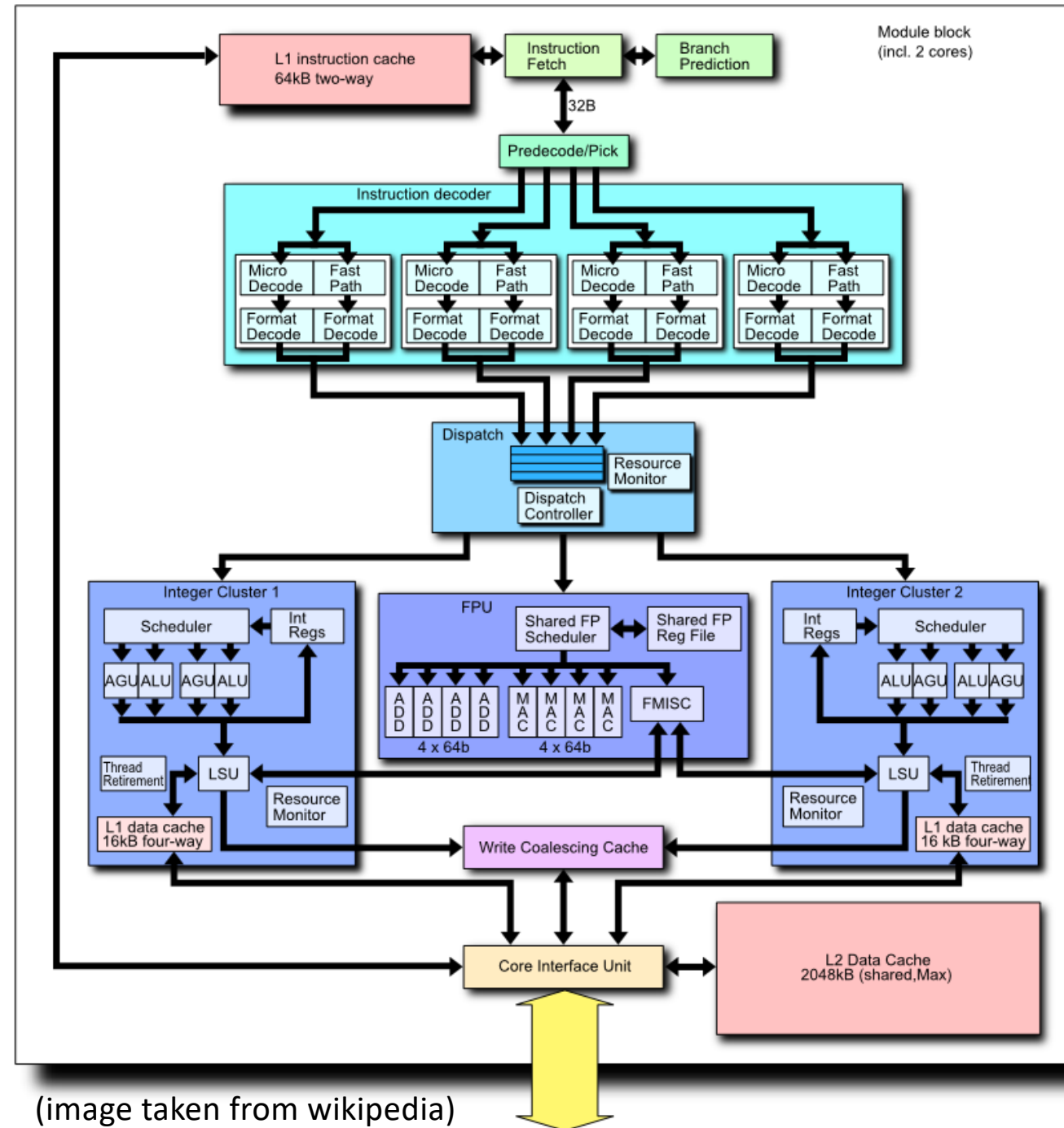
- Single chip, multiple cores
- Dual-, quad-, octa-...
- Each core has its own hardware units
  - Computations in parallel perform well
- Might share part of the cache hierarchy



# AMD Bulldozer (2011)

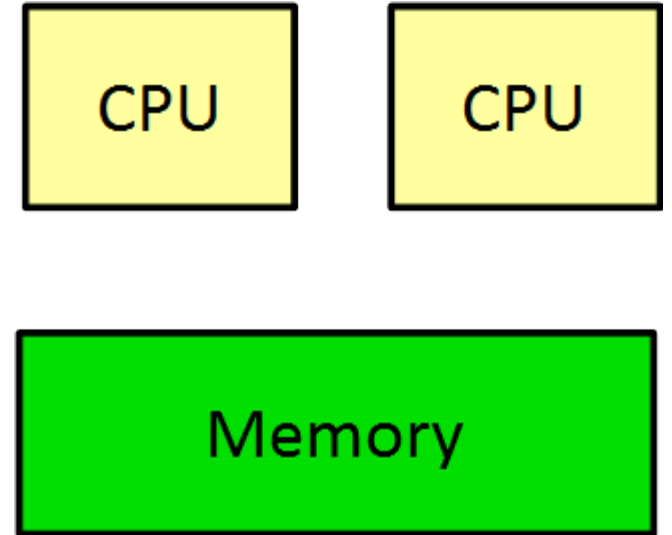
Between multicore and simultaneous multithreading  
→ hybrid design

- 2x cores integer
- 1x core floating point



# Symmetric Multi Processing

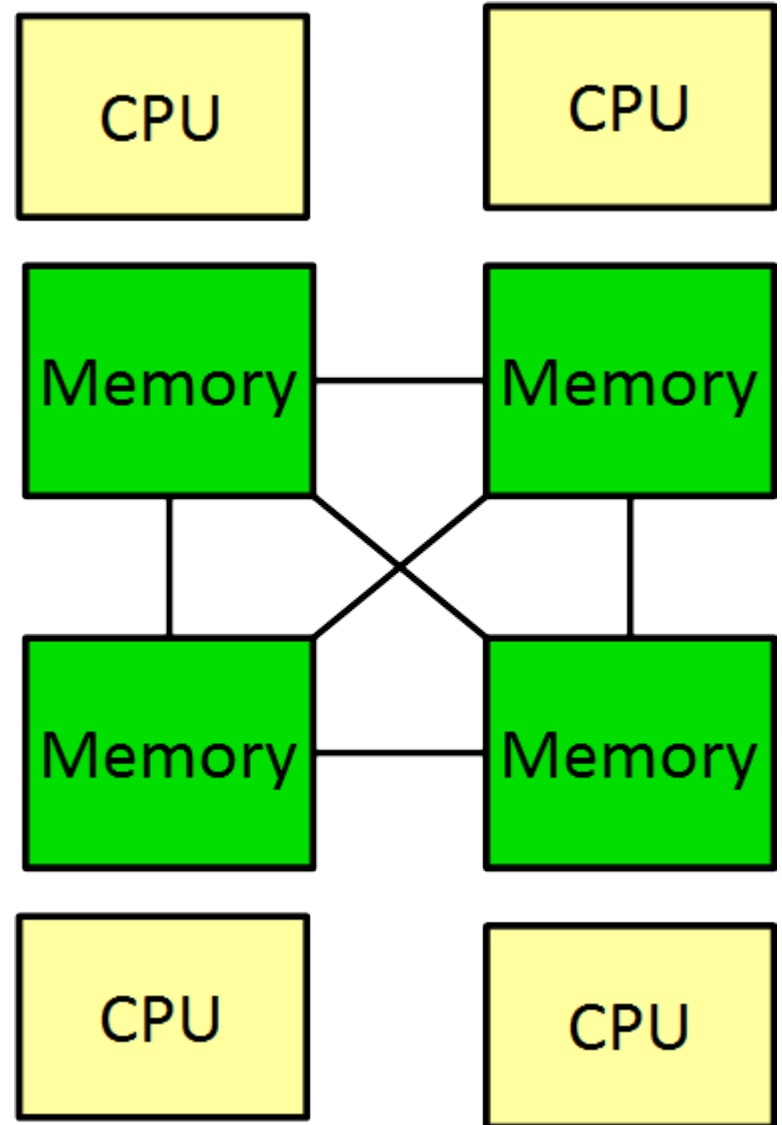
- Multiple chips (CPUs) on the same system
- CPUs share main memory (same cost to access memory for each CPU)
- Communication through main memory
- CPUs still have caches, could have multiple cores





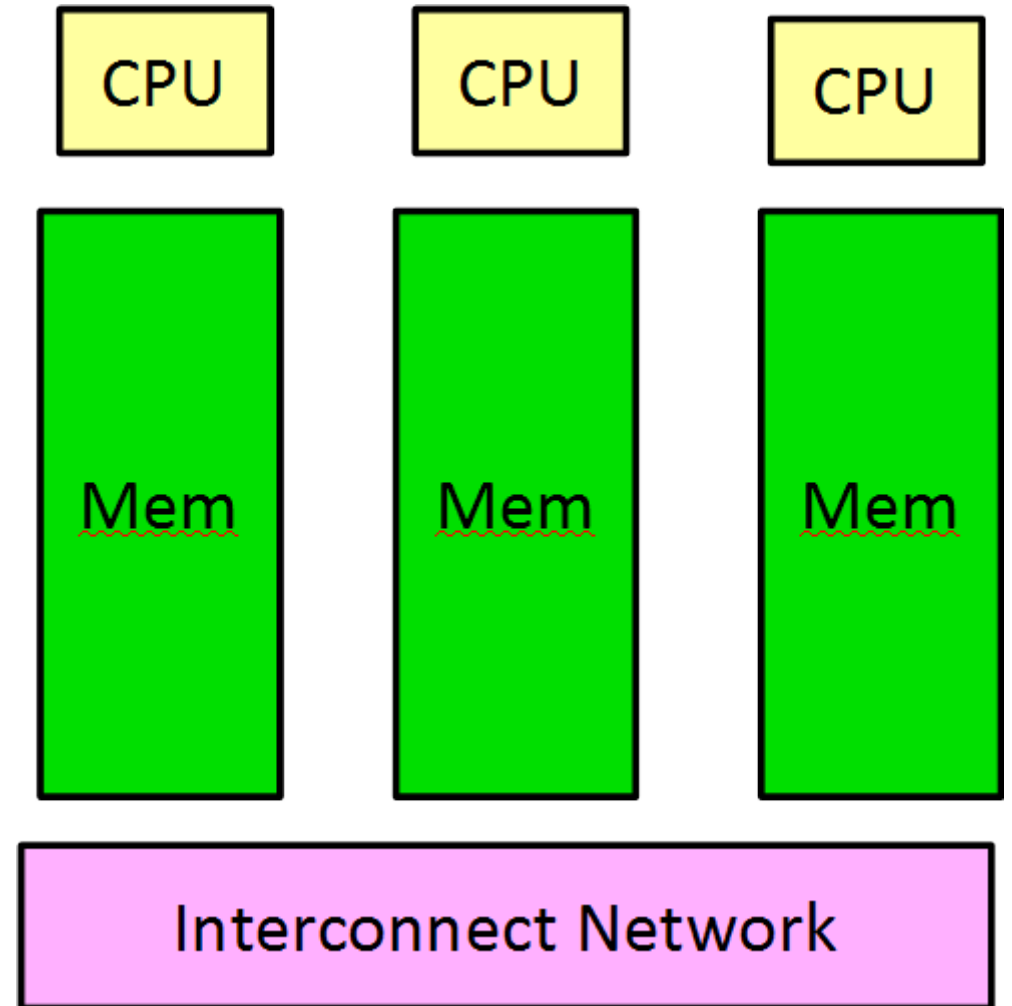
# Non-Uniform Memory Access

- Main memory is distributed
- Accessing local/remote memory is faster/slower
- Shared memory interface



# Distributed Memory

- Clusters, Data Warehouses
- Large-scale machines
  - See top500
- Message Passing
  - MPI
  - ...



# Shared vs Distributed Memory

- The categorization is about the *native programming/ communication interface* the system provides
  - Shared: directly access memory locations (x.f) [implicit communication]
  - Distributed: send messages (e.g. over network) to access/exchange data [explicit communication]
- Shared memory systems still need to exchange messages between processors (synchronize caches)
- It is possible to program (via suitable abstractions)
  - shared memory systems as distributed memory systems
  - distributed memory systems as shared memory systems

# Summary

- Parallelism is used to improve performance (at all levels)
- Architects cannot improve sequential CPU performance anymore
- “Multicore era” → programmers need to write parallel programs
- Shared vs distributed memory architectures

# Further reading material (for the interested)

