Small Scale Parallel Programming
Introduction to Parallel Computing

Salvatore Filippone

salvatore.filippone@cranfield.ac.uk

High Performance Computer Architectures and Parallelism

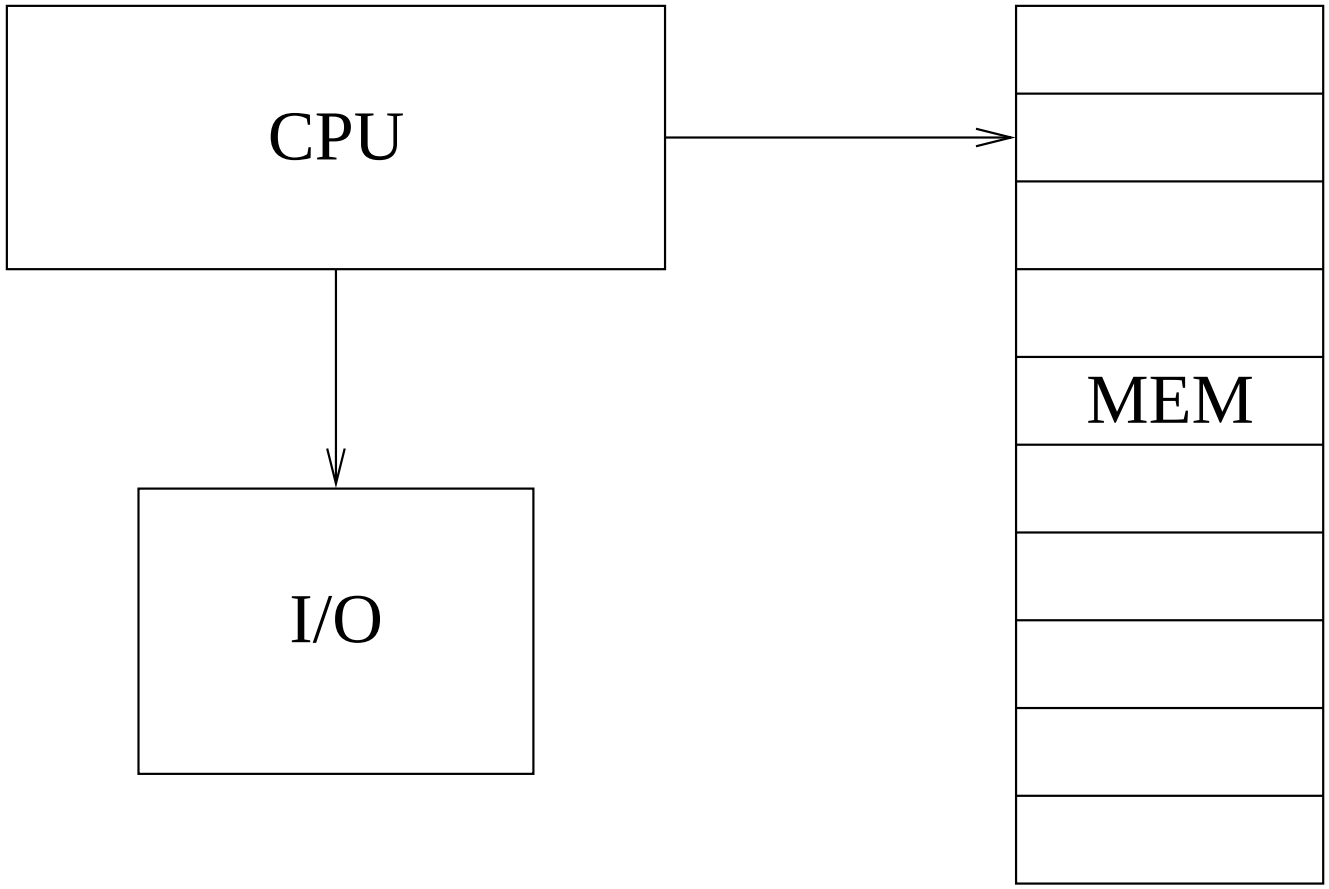
Computer Architecture

Organization and interaction of the various components making up a computer

- Nomenclature originally introduced in the 60s (IBM 360)
- Basic idea: Von Neumann architecture;
- Evolution over time:
  - "Traditional" systems;
  - "Pipelined" computers;
  - Vector CPUs ;
  - Microprocessors;
  - RISC (Reduced Instruction Set Computer) CPUs;
  - SMPs (Symmetric MultiProcessor);
  - MPPs (Massively Parallel Processor).
  - Multi-core computers
  - GPU (Graphical Processing Units) accelerators;

CPU: Central Processing Unit.

Von Neumann Architecture



Instruction execution is defined on a sequence of discrete time steps

Von Neumann Architecture

Basic principles:

1. Memory is a set of storage cells, identified by an *address* (a number).
2. CPU has a set of *registers*, holding data from memory;
3. Arithmetic instructions are only executed in the CPU (RISC: only on data already in the registers);
4. Special registers are used to control the execution sequence;
5. Memory holds the *instructions* making up a program;
6. Memory *also* holds *data* on which the instructions work.

Note: the ability to interpret the contents of memory as *either* data or instruction is what makes compilers possible! Some areas of memory may be *tagged* as being code, but there is nothing *intrinsic* to the memory contents per se.

Von Neumann/RISC Architecture

Instruction classes:

Data Movement: `LW R1,D(R2)` `SW R1,D(R2)` `MOV.D`

Arithmetics: `ADD R3,R2,R1` `SLT`

Floating-point: `MUL.D F3,F2,F1`

Control: `BEQ` `BNE` `J` `JALR`

Von Neumann/RISC Architecture

Example:

```
c = a + b;
```

```
LW R1,DA(R7)        R1 ← MEM[DA+[R7]]
LW R2,DB(R7)        R2 ← MEM[DB+[R7]]
ADD R3,R1,R2        R3 ← [R1]+[R2]
SW R3,DC(R7)        MEM[DC+[R7]] ← [R3]
```

with `[ ]` we denote access to contents (of a register or of a memory location)

Von Neumann/RISC Architecture

Write Access

causes the data sent by the CPU to be written into the memory location specified by the address. Typically, a write access will only occur when a STORE instruction is executed by the CPU. Write accesses to different addresses will go to different parts of the memory.

Read Access

causes the data residing at the specified address to be returned to the CPU. Typically, a read access will only occur when a LOAD instruction is executed by the CPU. Read accesses to different addresses will read data from different parts of the memory.

Processor history

1. Accumulator (1948-1956)
2. Stack Processors (1960s)
3. General-Purpose Register processors (1956-)
4. Microprocessors (1970s-)
5. Vector Processors (1970-)
6. RISC (1980-)
7. Multi-core
8. Many-core (GPUs)

Intel x86 is a peculiar hybrid, integer general-purpose, floating-point stack, apparently CISC, but really a RISC on the inside, “classic” SSE is almost but not quite a vector coprocessor, but Skylake is a vector processor. . .

Memory Hierarchy

1. Evolution of processor vs. memory technology: differential speed;
2. Paging and Virtual Memory;
3. Cache memories;

Essential to proper design of software.

Memory Hierarchy

Facts of (memory) life:

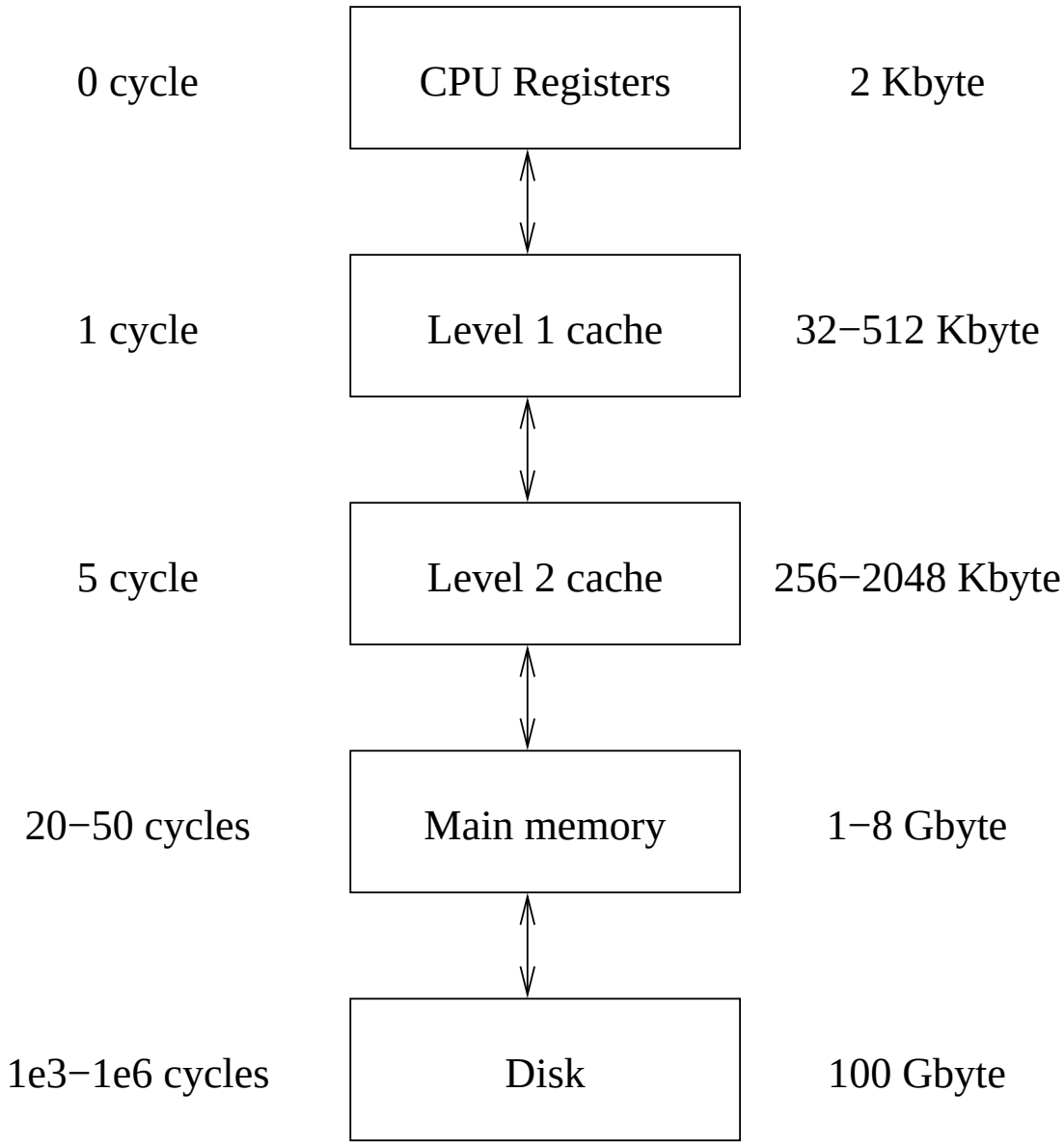
Memory can be

- Large;
- Fast;
- Cheap;

but *you can choose only two*;

Computer architects and programmers want all three anyway!
Hence the concept of *Memory Hierarchy*

Memory Hierarchy



Memory Hierarchy

Why does it work?

1. Memory access is latency bound, but data are transferred in blocks;
2. Programming *locality* principles:
   - Temporal locality: if a program accesses a memory location now it will likely access it again in the near future;
   - Spatial locality: if a program accesses a memory location now, it will likely access nearby locations as well.

Assume we have two levels:

Cache: Uniform cost: 1 cycle;

Main: 20 cycles for start of blocks of 32 items, with items after the first being made available at full speed.

Average cost of sequential access of 32 items:

$$T_{avg} = \frac{20 \times 1 + 1 \times 31}{32} = 1.59$$

Performance Metrics

Most important:

Time to complete a task;

However, there are different viewpoints:

User: Time to response (single task);

Administrator: Throughput (multiple tasks);

Elapsed time: "wall clock"

CPU time: time spent in the CPU on account of a specific task

What is the program with which you perform the measurements?

Performance Metrics

Benchmark: a program in some ways "representative" of actual usage

1. Real applications;
2. Modified (simplified) Real applications (aka MiniApps);
3. Kernels;
4. Toys;
5. Synthetic Benchmarks;

Units of measure: time, MIPS (million instructions per second), MFLOPS (million floating-point operations per second).
Example: LINPACK & friends (see http://www.top500.org)

Performance Metrics

Fundamental equation of computer performance:

$$CPU = IC \times CPI \times Ct$$

- *IC* Instruction count: algorithm, language–compiler, instruction set;
- *CPI* Cycles per Instruction: instruction set, organization of the CPU;
- *Ct* Cycle time: organization of the CPU, semiconductor technology;

Concentrating on CPU time for now (i.e. looking at the CPU internals only).

Moore's Law

Number of components in an integrated circuit doubles every 18-24 months

. . . and all other performance measures will follow. This has been true from 1970 for more than 30 years, entailing a performance improvement of 2^{20}!
Since 2008 we are at a density plateau

Problems: Power dissipation, components costs, memory costs.

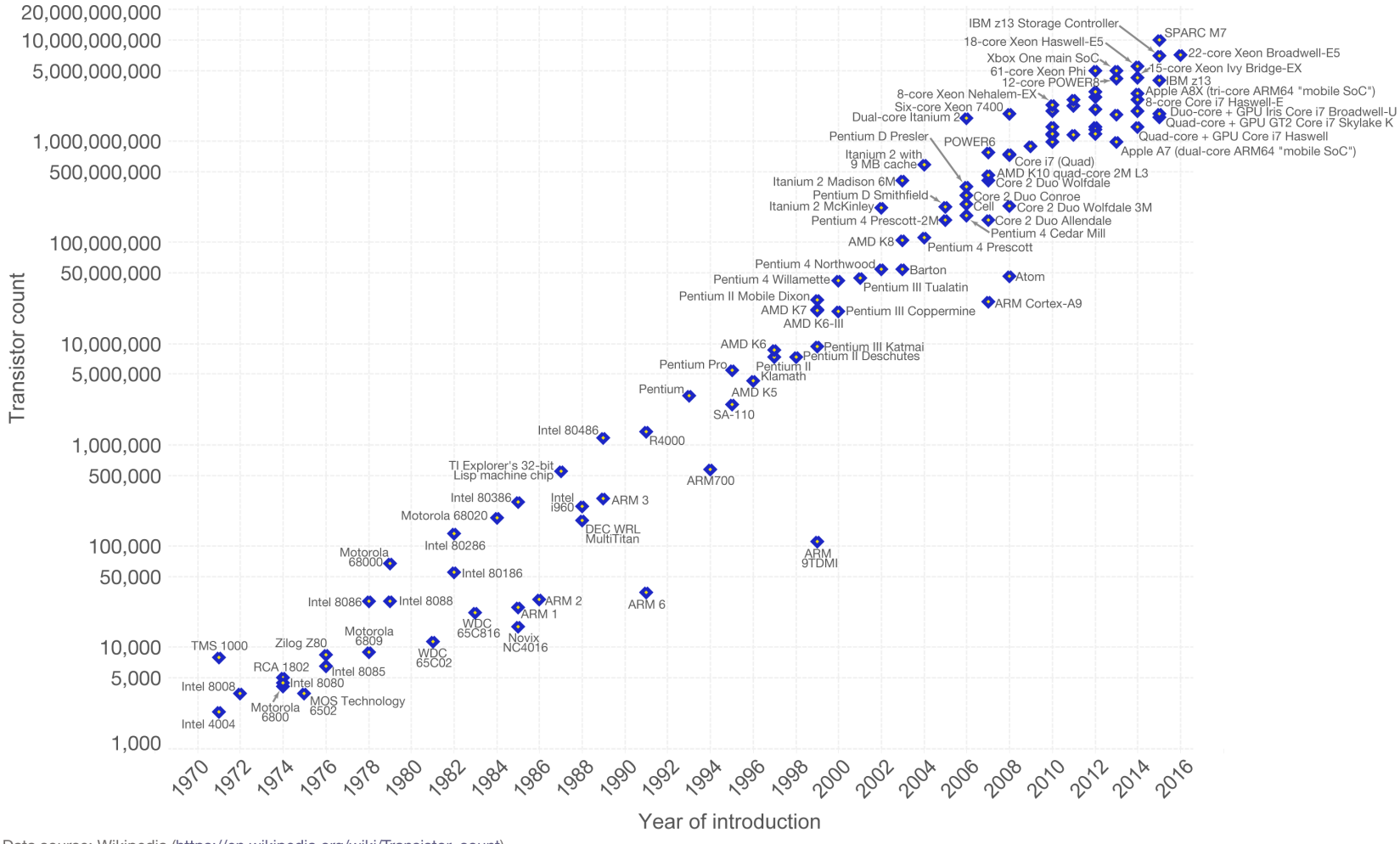
Solution: Use parallelism

Moore's law

Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

Our World in Data

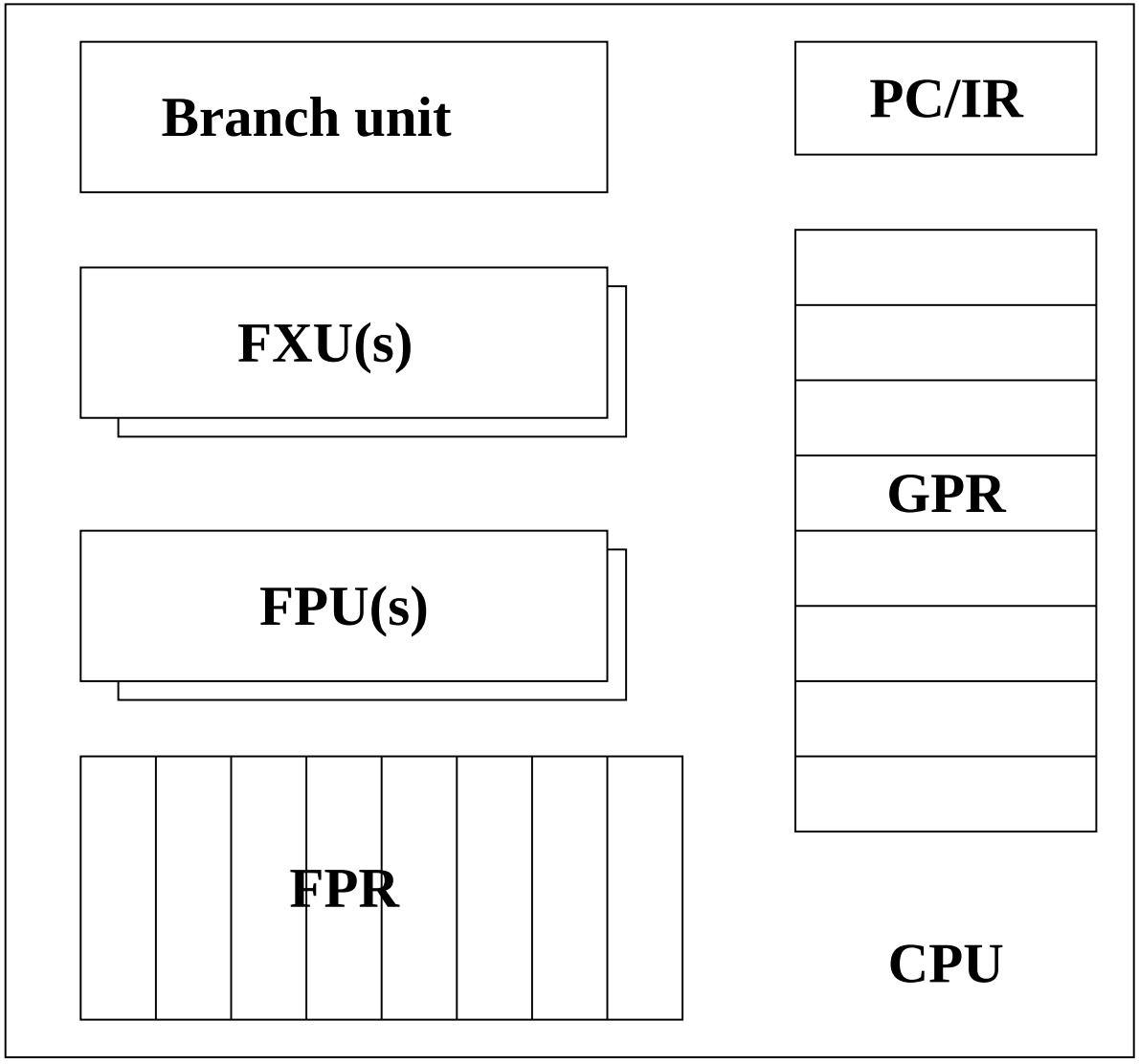
Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

from Wikipedia, `https://en.wikipedia.org/wiki/Moore%27s_law`

CPU Internals: the Execution Cycle



CPU Internals: the Execution Cycle

`LW R1,DA(R2)`

IF	Instruction fetch: `IR` $\leftarrow$ `MEM[PC]`
ID	Instruction decode: recognize LW
EXE	Execute: address buffer `AB` $\leftarrow$ `DA+[R2]`
MEM	Memory: data buffer `DB` $\leftarrow$ `MEM[AB]`
WB	Write back: `R1` $\leftarrow$ `[DB]`

Buffer: additional internal register, not addressable explicitly by the programmer.

What is a machine cycle?

One instruction per cycle: means that the machine cycle is the time to complete all five of the above, e.g. 5 ns, with $CPI = 1$;

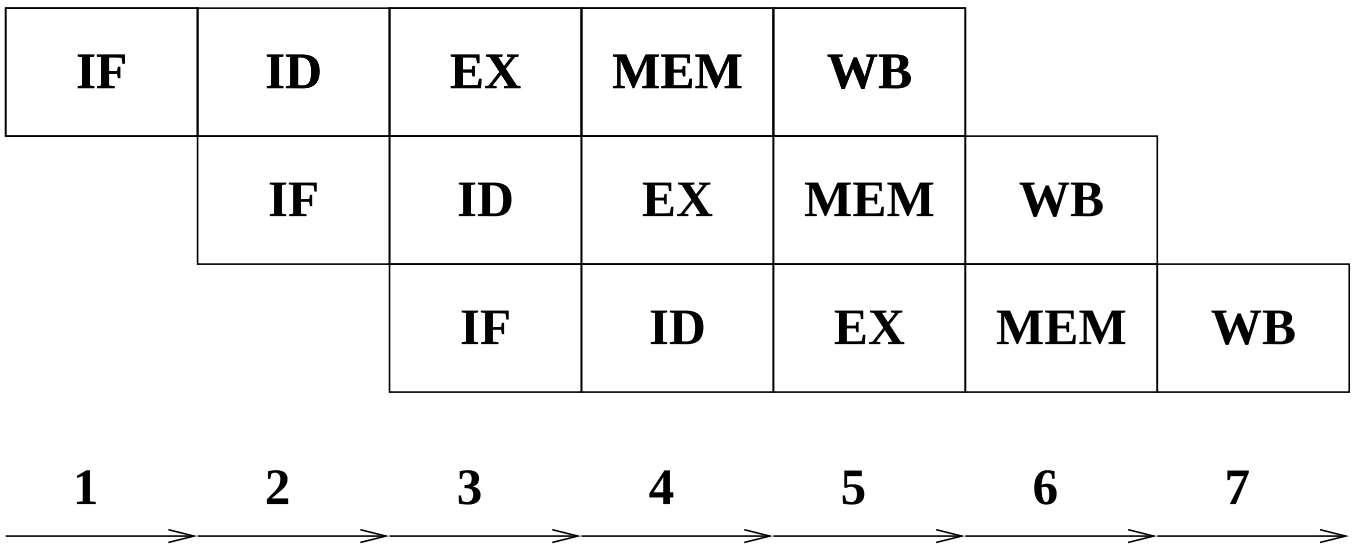
One phase per cycle: e.g. 1 ns, but then we have $CPI = 5$.

Pipelining

Can we have our cake and eat it too?

Definition (Pipelining)

A technique to reduce *CPI* by increasing throughput



Average instruction time over n instructions:

$$T_{avg} = \frac{T_{tot}}{n} = \frac{T_s + n}{n} \underset{n \to \infty}{\to} 1$$

Pipelining

Fundamental issue:

How do we provide a stream of mutually independent instructions?

A Simple Execution Sequence

1. `LW R1,DA(R7)` `R1 ← MEM[DA+[R7]]`
2. `LW R2,DB(R7)` `R2 ← MEM[DB+[R7]]`
3. `ADD R3,R1,R2` `R3 ← [R1]+[R2]`
4. `SW R3,DC(R7)` `MEM[DC+[R7]] ← [R3]`

instruction 3 has to wait for completion of 1 and 2, thus generating a "bubble" in the pipeline.

Solution strategies in hardware:

- Vector processors;
- RISC Processors (and system);

Solutions depend on *BOTH* programmer and compiler technology.