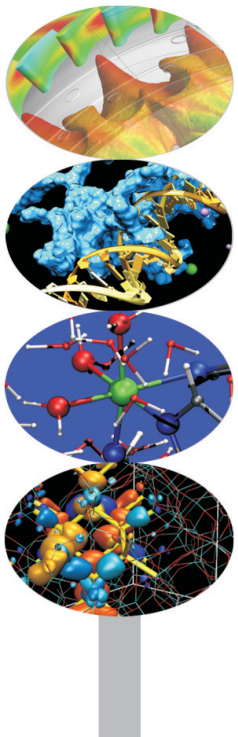
CINECA SCAI
SuperComputing Applications and Innovation

SAPIENZA
UNIVERSITÀ DI ROMA

Programmazione Avanzata

Vittorio Ruggiero

(v.ruggiero@cineca.it)



Roma, Marzo 2017

CINECA

Outline

Pipeline

CPU: internal parallelism?

- CPU are entirely parallel
  - pipelining
  - superscalar execution
  - units SIMD MMX, SSE, SSE2, SSE3, SSE4, AVX
- To achieve performances comparable to the peak performance:
  - give a large amount of instructions
  - give the operands of the instructions

The pipeline

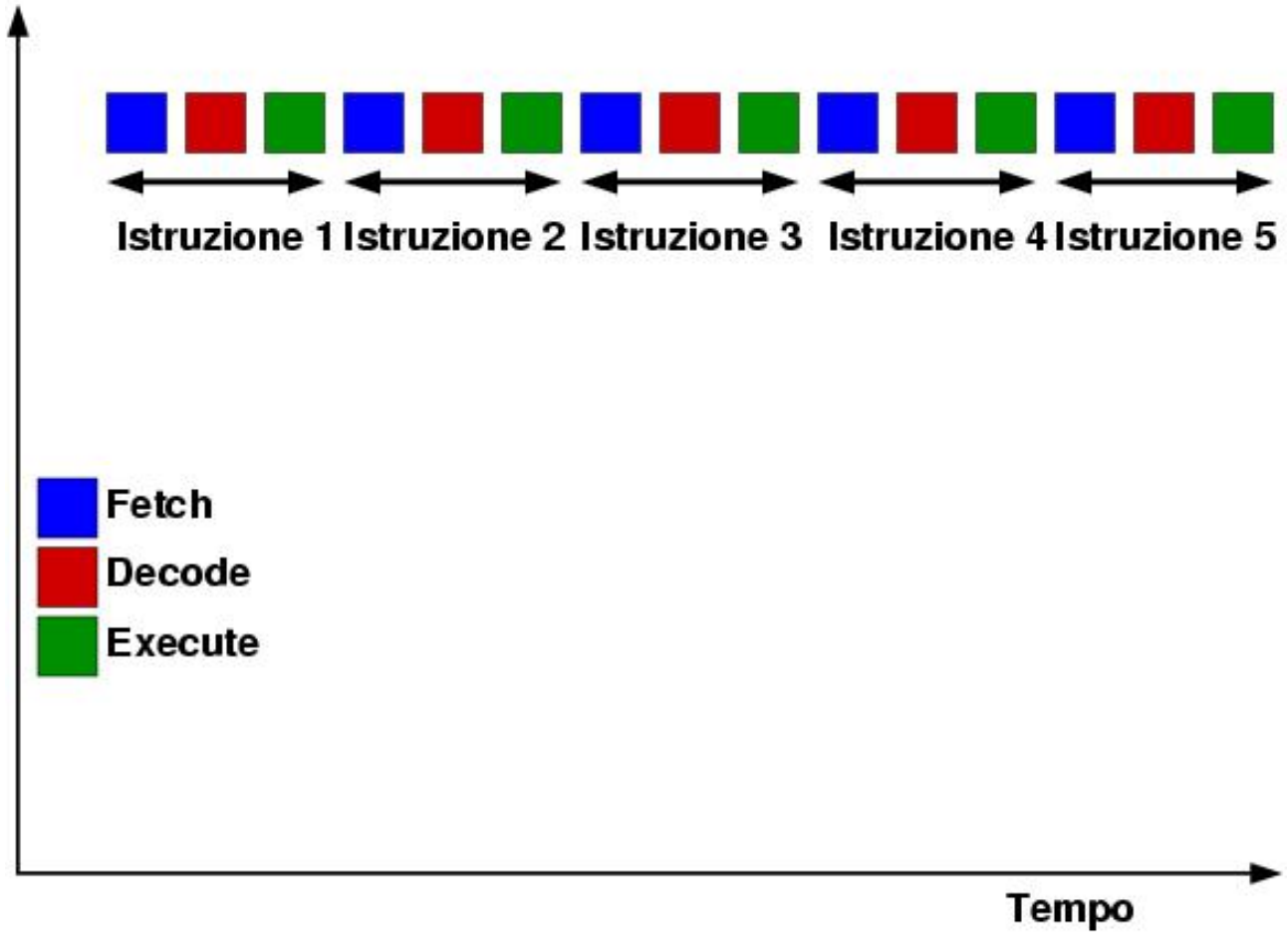
- Pipeline, channel or tube for carrying oil
- An operation is split in independent stages and different stages are executed simultaneously
  - fetch (get, catch) gets the instruction from memory and the pointer of Program Counter is increased to point to the next instruction
  - decode instruction gets interpreted
  - execute send messages which represent commands for execution
- Parallelism with different operation stages
- Processors significantly exploit pipelining to increase the computing rate

CPU cycle

- The time to move an instruction one step through the pipeline is called a machine cycle
- CPI (clock Cycles Per Instruction)
  - the number of clock cycles needed to execute an instruction
  - varies for different instructions
  - its inverse is IPC (Instructions Per Cycle)

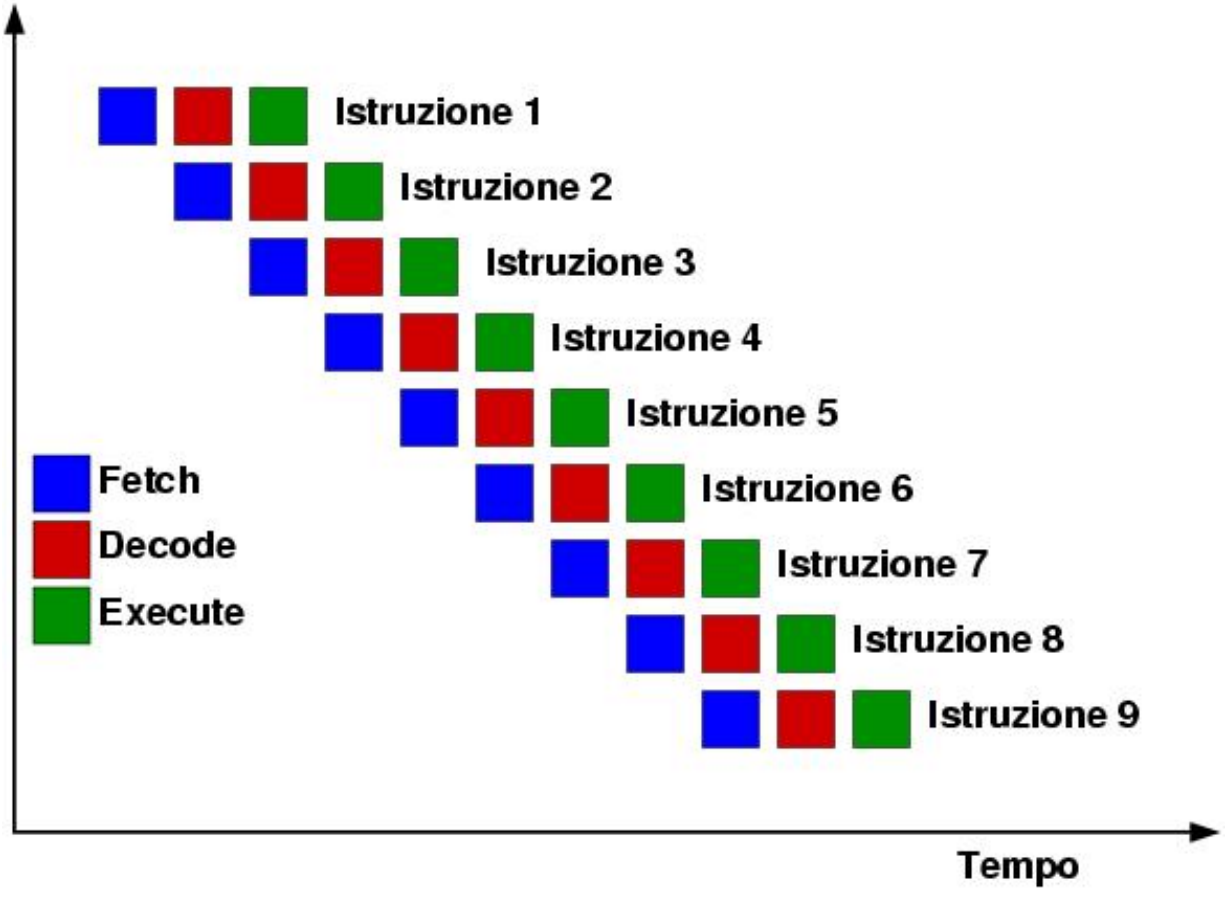
NON pipelined computing units

- Each instruction is completed after three cycles



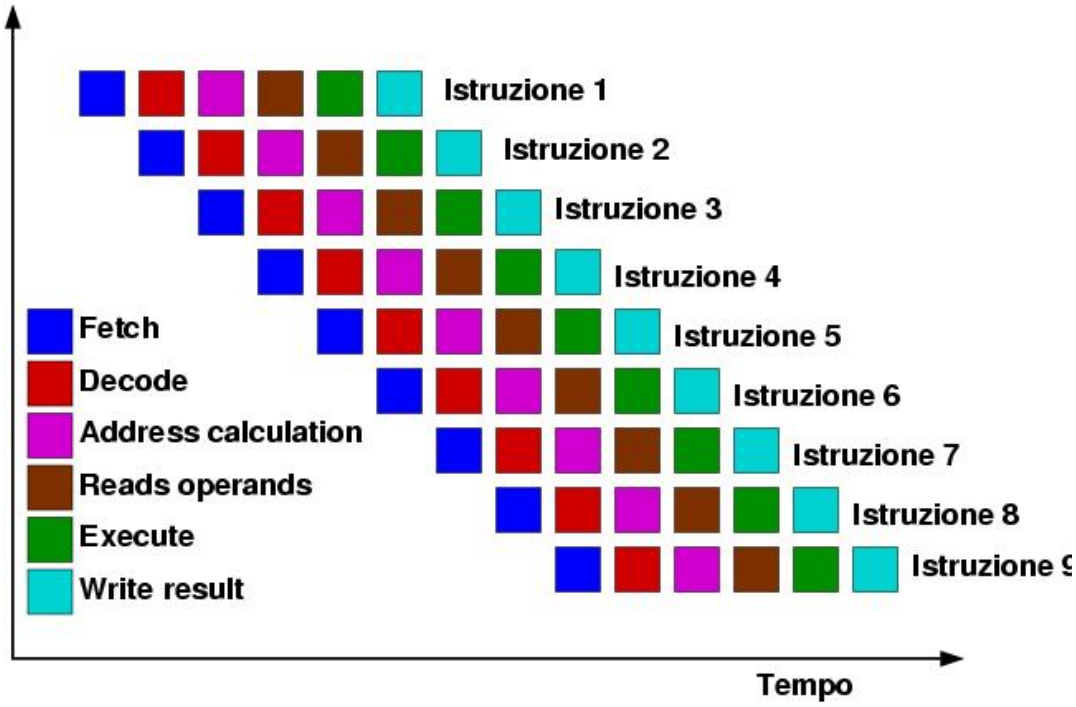
Pipelined units

- After 3 clock cycles, the pipeline is full
- A result per cycle when the pipeline is completely filled
- To fill it 3 independent instructions are needed (including the operands)



Superpipelined computing units

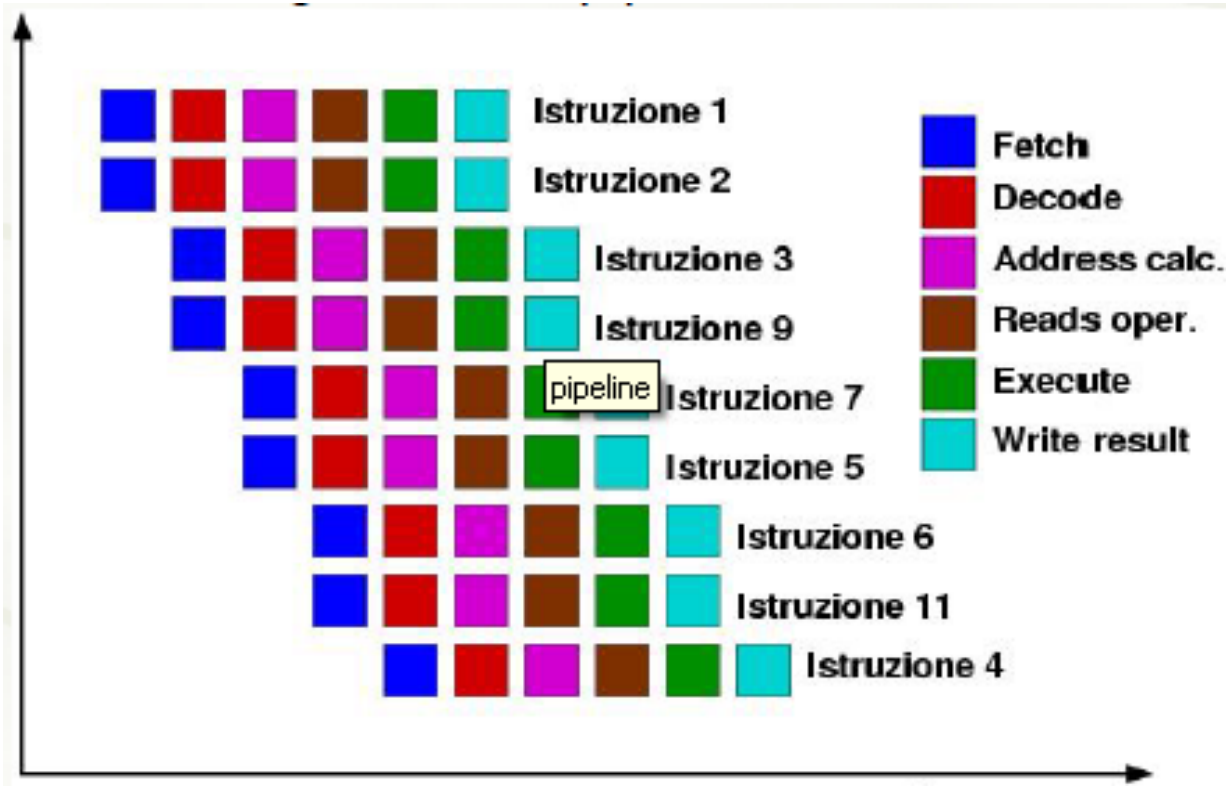
- After 6 clock cycles, the pipeline is full
- A result per cycle when the pipeline is completely filled
- To fill it 6 independent instructions are needed (including the operands)
- It is possible to halve the clock rate, i.e. doubling the frequency



Out of order execution

- Dynamically reorder the instructions
  - move up instructions having operands which are available
  - postpone instructions having operands still not available
  - reorder reads/write from/into memory
  - always considering the free functional units
- Exploit significantly:
  - register renaming (physical vs architectural registers)
  - branch prediction
  - combination of multiple read and write from/to memory
- Crucial to get high performance on present CPUs
- The code should not hide the reordering possibilities

Out of order execution



Superscalar execution

- CPUs have different independent units
  - functional differentiation
  - functional replication
- Independent operations are executed at the same time
  - integer operations
  - floating point operations
  - skipping memory
  - memory accesses
- Instruction Parallelism
- Hiding latencies
- Processors exploit superscalarity to increase the computing power for a fixed clock rate

How to exploit internal parallelism?

- Main issues:
  - minimize dependency among instructions
  - handle conditional statements (if and loop)?
  - provide all the required data
- Who has to modify the code?
  - CPU? $\rightarrow$ yes, if possible, OOO and branch prediction
  - compiler? $\rightarrow$ yes, is possible, understanding the semantics
  - user? $\rightarrow$ yes, for the most complex cases
- Strategies
  - loop unrolling $\rightarrow$ unroll the loop
  - loop merging $\rightarrow$ merge loops into a single loop
  - loop splitting $\rightarrow$ decompose complex loops
  - function inlining $\rightarrow$ avoid breaking instruction flow

Loop unrolling

- Repeat the body of a loop k times and go through the loop with a step length k
- k is called the unrolling factor

```
do j = 1, nj                    -> do j = 1, nj
do i = 1, ni                    -> do i = 1, ni, 2
 a(i,j)=a(i,j)+c*b(i,j)         ->  a(i  ,j)=a(i  ,j)+c*b(i  ,j)
                                ->  a(i+1,j)=a(i+1,j)+c*b(i+1,j)
```

  - The unrolled version of the loop has increased code size, but in turn, will execute fewer overhead instructions.
  - The same number of operations, but the loop index is incremented half of the times
  - The performance of this loop depends upon both the trace cache and L1 cache state.
  - In general the unrolled version runs faster because fewer overhead instrunctions are executed.
  - It is not valid when data dependences exist.

Reduction & Unroll


```
    do j = i, nj ! normal case 1)
       do i = i, ni
          somma = somma + a(i,j)
       end do
    end do
.......
    do j = i, nj !reduction to 4 elements.. 2)
       do i = i, ni, 4
          somma_1 = somma_1 + a(i+0,j)
          somma_2 = somma_2 + a(i+1,j)
          somma_3 = somma_3 + a(i+2,j)
          somma_4 = somma_4 + a(i+3,j)
       end do
    end do
    somma = somma_1 + somma_2 + somma_3 + somma_4
f77 -native -O2 (-O4)
time 1) ---> 4.49785 (2.94240)
time 2) ---> 3.54803 (2.75964)
```

What inhibits loop unrolling?

- Conditional jumps (`if` ...)
- Calls to intrinsic functions and library (sin, exp, ...)
- I/0 operations in the loop

Compiler options

- Can I know how compiler works?

Compiler options

- Can I know how compiler works?
- See reference documentation for the compiler.

Compiler options

- Can I know how compiler works?
- See reference documentation for the compiler.
- Use, for example, the intel compiler with flag **-qopt-report**.