Scuola Superiore Sant'Anna
di Studi Universitari e di Perfezionamento

Università di Pisa

Master Program (Laurea Magistrale) in Computer Science and Networking

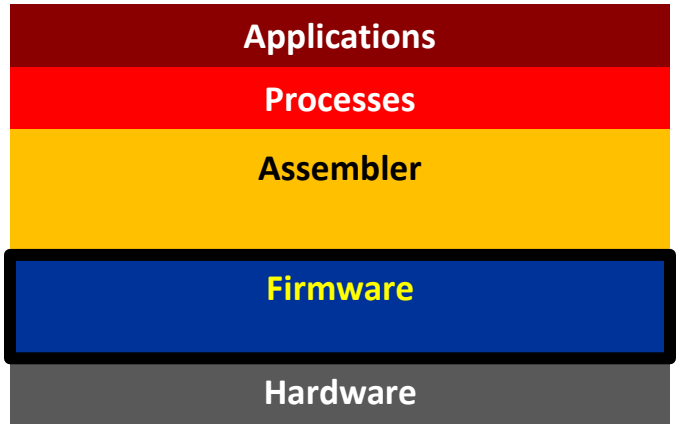
High Performance Computing Systems and Enabling Platforms

Marco Vanneschi

1. Prerequisites Revisited

1.3. Assembler level, CPU architecture, and I/O

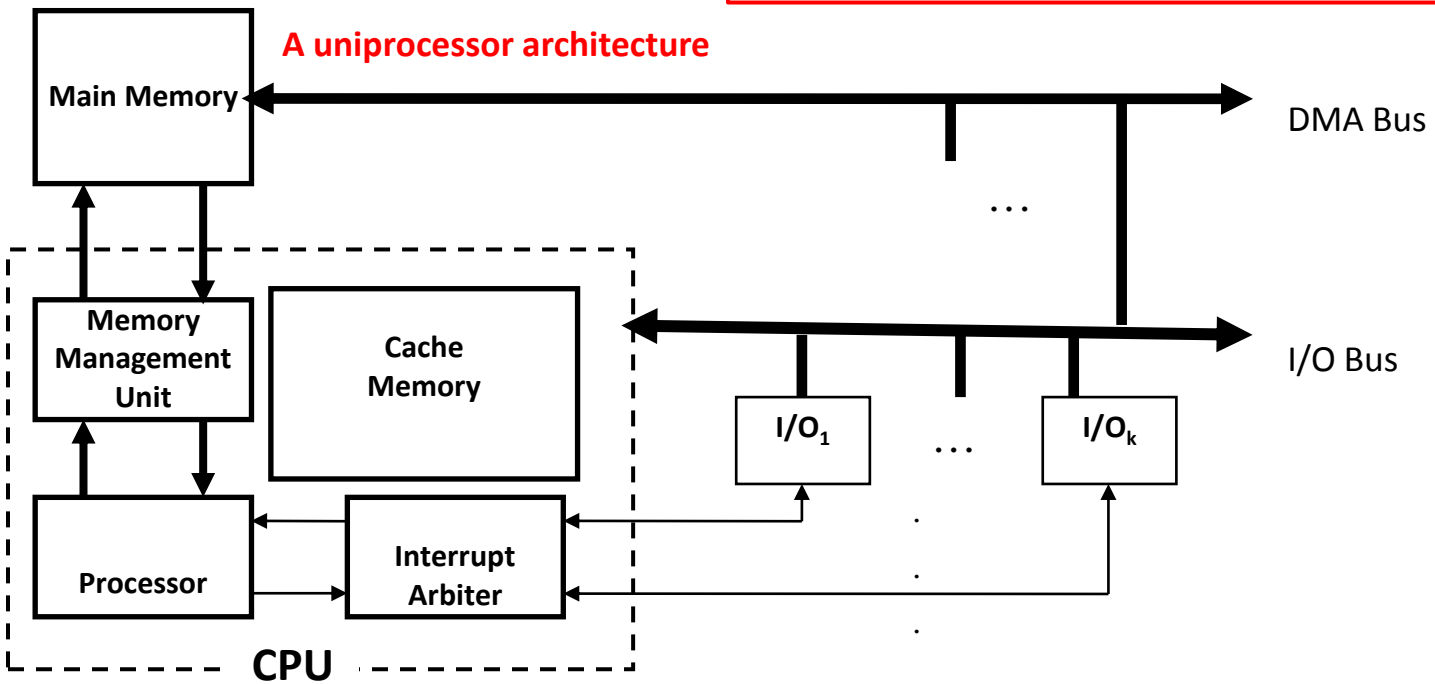
Uniprocessor architecture



The firmware interpretation of any assembly language instructions has a **decentralized organization**, even for simple uniprocessor systems.

The interpreter functionalities are **partitioned** among the various processing units (belonging to CPU, M, I/O subsystems), *each unit having its own Control Part and its own microprogram.*

Cooperation of units through explicit message passing .



Assembly level characteristics

- RISC (Reduced Instriction Set Computer) vs CISC (Complex Instriction Set Computer)
  - "The MIPS/Power PC *vs* the x86 worlds"

RISC:

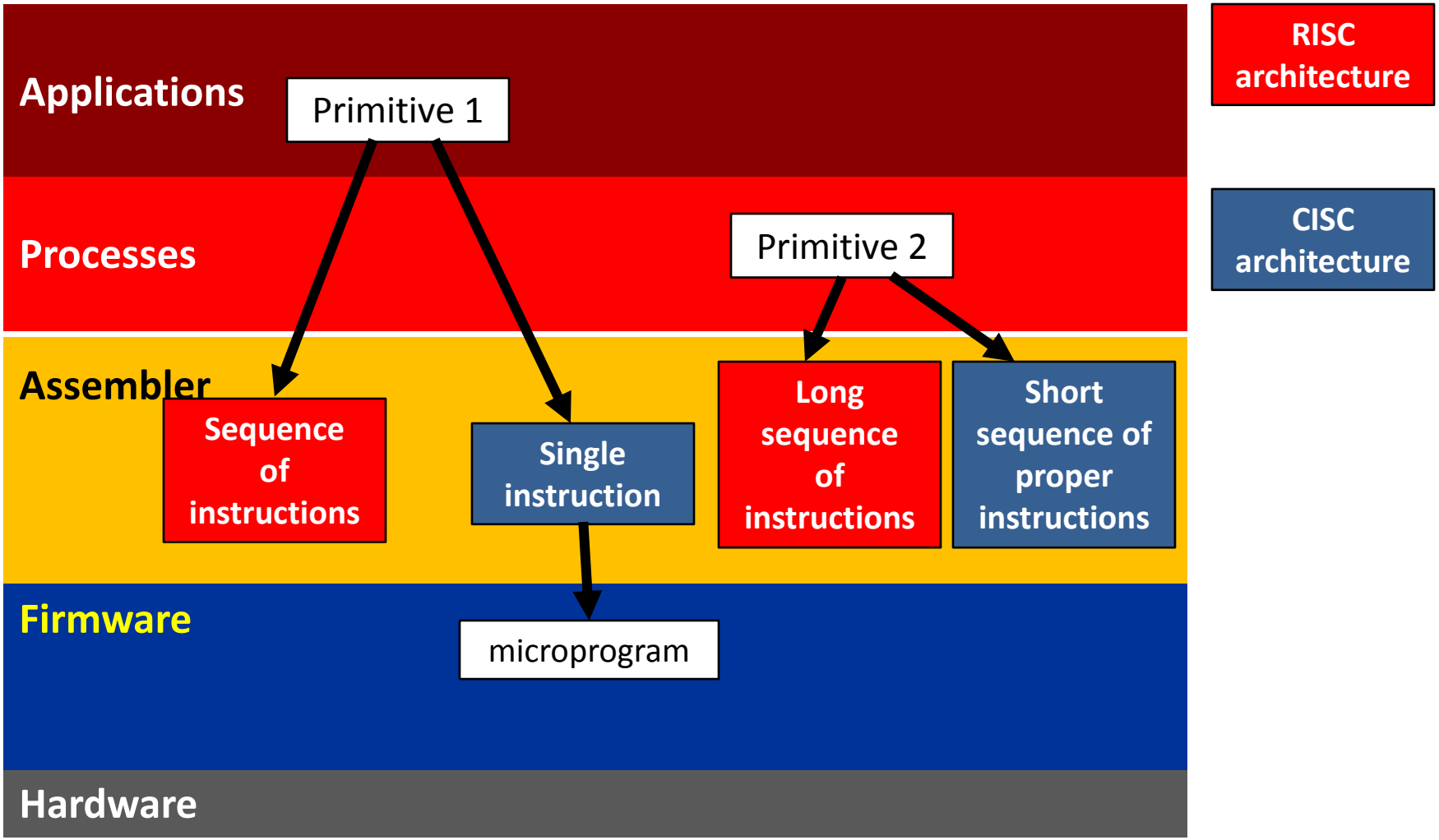
- basic elementary arithmetic operations on integers,
- simple addressing modes,
- intensive use of General Registers, LOAD/STORE architecture
- complex functionalities are implemented at the assembler level by (long) sequences of simple instructions
- rationale: powerful optimizations are (feasible and) made much easier:
  - Firmware architecture
  - Code compilation for Caching and Instruction Level Parallelism

Assembly level characteristics

CISC:

- includes complex istructions for arithmetic operations (reals, and more),
- complex nested addressing modes,
- instructions corresponding to (parts of) systems primitives and services
  - process cooperation and management
  - memory management
  - …
  - graphics,
  - …
  - networking
  - ….

RISC and CISC approaches



Test: to be submitted and discussed

Basically, CISC implements at the firmware level what RISC implements at the assembler level.

Assume:

- No Instruction Level Parallelism
- Same technology: clock cycle, memory access time, etc

- *Why* the completion time could be reduced when implementing the same functionality at the firmware level ("in hardware" … !) compared to the implementation at the assembler level ("in software" … !)?
- *Under which conditions* is the ratio of completion times significant? Which *order of magnitude* ?

- **Exercise**: answer these questions in a non trivial way, i.e. exploting the concepts and features of the levels and level structuring.
- **Write the answer** (1-2 pages)*, submit the work to me, and discuss it at Question Time.*

➢ **Goal: to highlight background problems, weaknesses, … to gain aptitude.**

Performance parameters

Average SERVICE TIME per instruction

$$\forall i = 1,\ldots,r: \quad T_i = k_i \cdot \tau$$

Average service time of *individual* instructions

$$\forall i = 1,\ldots,r: \quad p_i \mid \sum_{i=1}^{r} p_i = 1$$

Probability *MIX* (occurrence frequencies of instructions) for a given APPLICATION AREA

$$T = \sum_{i=1}^{r} p_i \cdot T_i = CPI \cdot \tau$$

$$CPI = \sum_{i=1}^{r} p_i \cdot CPI_i$$

Global average SERVICE TIME per instruction
CPI = average number of **C**lock cycles **P**er **I**nstruction

$$\wp = \frac{1}{T}$$

Performance = average number of executed instructions per second (MIPS, GIPS, TIPS, …) = *processing bandwidth* of the system at the assembler-firmware levels

Performance parameters

Completion time of a program

- **Benchmark** approach to performance evalutation (e.g., **SPEC** benchmarks)
- **Comparison of distinct machines,** differing at the assembler (and firmware) level
- On the contary: the **Performance** parameters is meaningful *only for the comparison of different firmware implementations of the* **same** *assembler machine*.

$$T_c \approx m \cdot T = \frac{m}{\wp}$$

m = average number of executed instructions

This simple formula holds rigorously for uniprocessors without Instruction Level Parallelism

It holds with *good approximation* for Instruction Level Parallelism (*m = stream length*),
or for parallel architectures.

RISC *vs* CISC

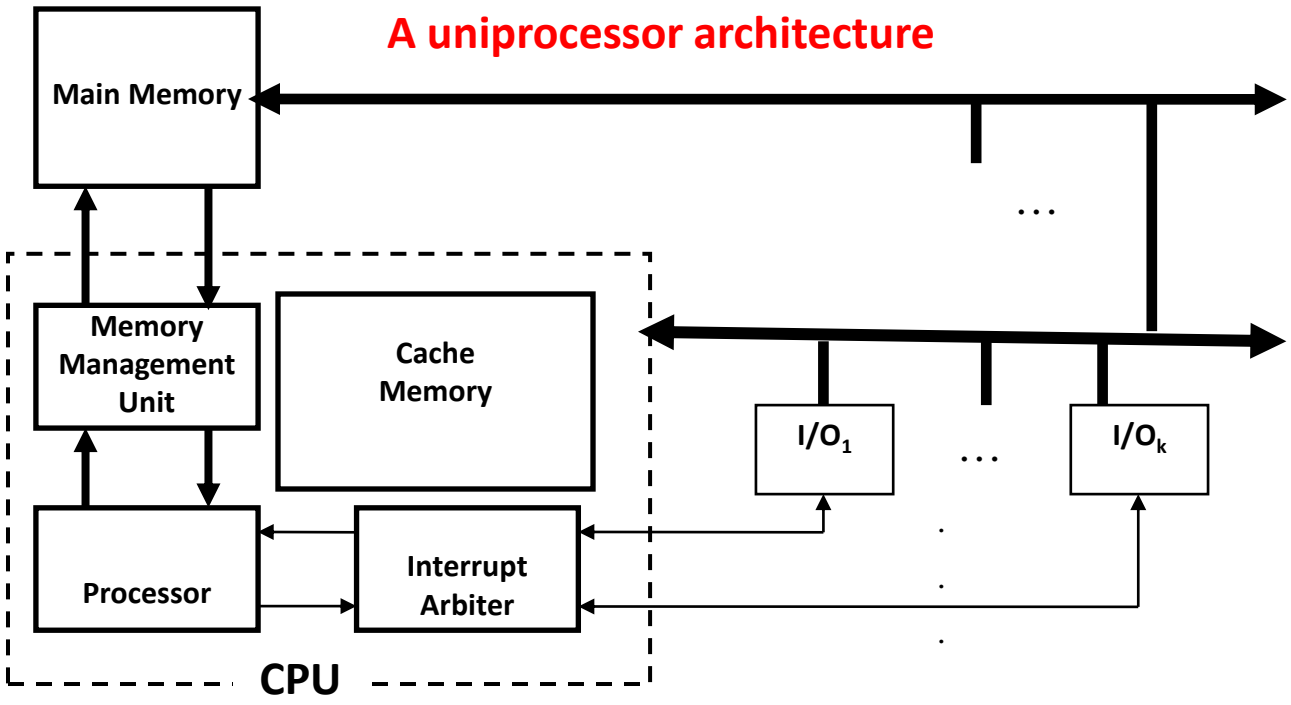
$$T_c \approx m \cdot T = \frac{m}{\wp}$$

- RISC decreases T and increases m
- CISC increases T and decreases m
- *Where is the best trade-off?*
- For **Instruction Level Parallelism** machines: RISC optimizations may be able to achieve a *T decrease* greater than the *m increase*
  - *cases in which the **execution bandwidth** has greater impact than the **latency***
- Where **execution latency** dominates, CISC achieves lower completion time
  - *e.g. process cooperation and management, services, special functionalities, …*

A possible trade-off

- Basically a **RISC** machine with the addition of **specialized co-processors or functional units** for complex tasks
  - Graphic parallel co-processors
  - Vectorized functional units
  - Network/Communication co-processors
  - …
- Concept: **modularity** can lead to **optimizations**
  - *separation of concerns* and its effectiveness
- Of course, a CISC machine with "*very* optimized" compilers can achieve the same/better results
  - A matter of **COMPLEXITY** and **COSTS / INVESTMENTS**

Modularity and optimization of complexity / efficiency ratio



In turn, an advanced I/O unit can be a parallel architecture (vector / SIMD co-processor / GPU / …), or a powerful multiprocessor-based network interface, or ….

For interprocess communication and management: *communication co-processors*

In turn, the Processor can be implemented as a (large) collection of parallel, pipelined units

E.g. pipelined hardware implementation of arithmetic operations on reals, or vector instructions

A “didactic RISC”

- See Patterson – Hennessy: **MIPS** assembler machine

- In the courses of Computer Architecture in Pisa: **D-RISC**
  - a didactic version (on paper) of MIPS with few, inessential simplifications
  - useful for explaining concepts and techniques in computer architecture and compiler optimizations
  - includes many features for code optimizations

D-RISC in a nutshell

- 64 General Registers: RG[64]
- Any instruction represented in a single 32-bit word
- All arithmetic-logical and branch/jump instructions operate on RG
  - **Integer** arithmetic only
  - Target addresses of branch/jump; **relative to Program Counter** , or RG contents
- Memory accesses can be done by LOAD and STORE instructions only
  - **Logical addresses** are generated, i.e. CPU "sees" the Virtual Memory of running process
  - Logical address computed as the sum of two GR contents (**Base** + **Index** mode)
- Input/output: Memory Mapped I/O through LOAD/STORE
  - No special instructions
- Special instructions for
  - Interrupt control: mask, enable/disable
  - Process termination
  - Context switching: minimal support for MMU
  - Caching optimizations: annotations for prefetching, reuse, de-allocation, memory re-write
  - Indivisible sequences of memory accesses and annotations (multiprocessor application)

Example of compilation

```
int A[N], B[N];
    for (i = 0; i < N; i++)
        A[i] =  A[i] + B[i];
```

```
LOOP:   LOAD   RA, Ri, Ra
        LOAD   RB, Ri, Rb
        ADD    Ra, Rb, Rc
        STORE  RA, Ri, Rc
        INCR   Ri
        IF <   Ri, RN, LOOP
        END
```

- **R** = keyword denoting "register address"
- For clarity of examples, register addresses are indicated by symbols (e.g. RA = R27, then base address of A is the content RG[27])

REGISTER ALLOCATIONS and INITIALIZATIONS at COMPILE TIME:

- RA, RB: addresses of RG initialized at the base address of A, B
- Ri: address of RG containing variable i, initialized at 0
- RN: address of RG initialized at constant N
- Ra, Rb, Rc: addresses of RG containing temporaries (not initialized)

Virtual Memory denoted by **VM**
Program Counter denoted by **IC**

"NORMAL" SEMANTICS of SOME D-RISC INSTRUCTIONS:

LOAD RA, Ri, Ra ::
RG[Ra] = MV[RG[RA] + RG[Ri]], IC = IC + 1

ADD Ra, Rb, Rc ::
RG[Rc] = RG[Ra] + RG[Rb], IC = IC + 1

IF < Ri, RN, LOOP ::
if RG[Ri] < RG[RN] *then* IC = IC + offset(LOOP) *else* IC = IC + 1

Compilation rules

compile(**if** C **then** B) ≡

```
        IF (not C)   CONTINUE
        compile(B)
CONTINUE: …
```

compile(**if** C **then** B1 **else** B2) ≡

```
        IF (C)  THEN
        compile(B2)
        GOTO CONTINUE
THEN:   compile(B1)
CONTINUE: …
```

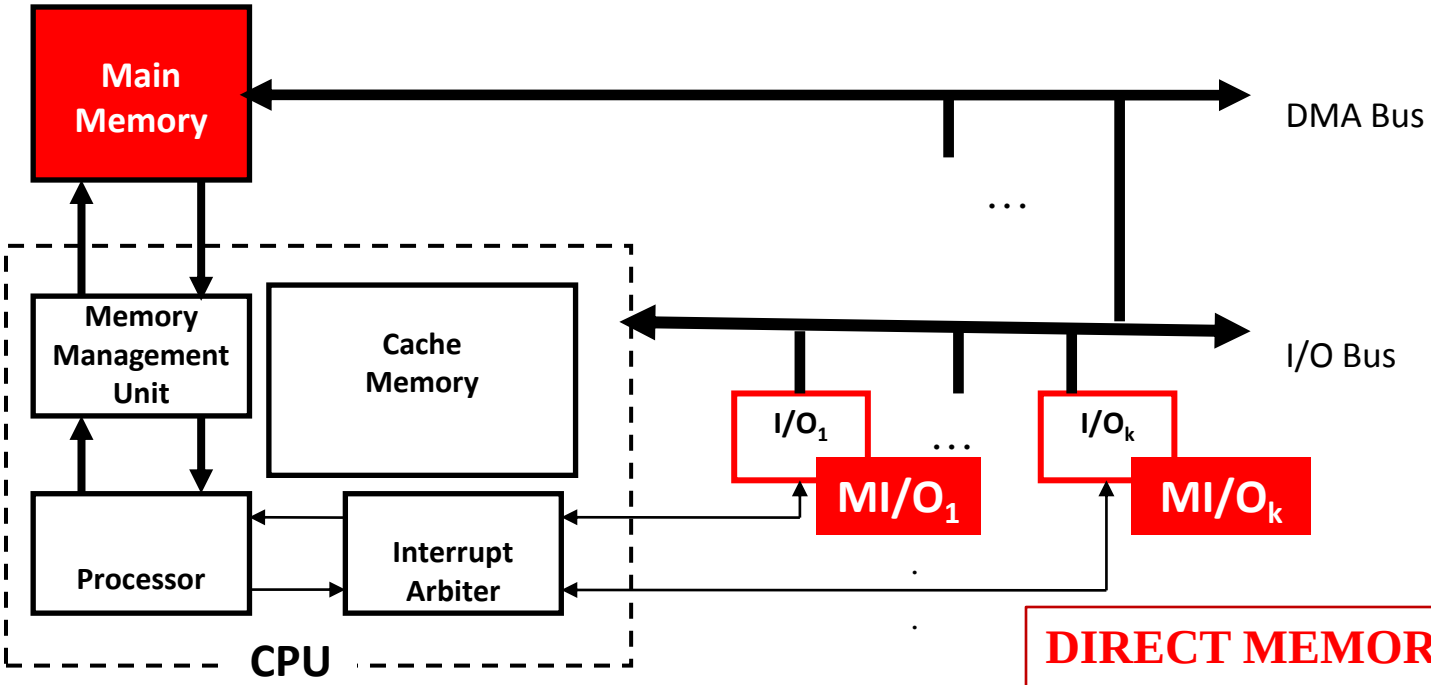
compile(**while** C **do** B) ≡

```
LOOP:   IF (not C)   CONTINUE
        compile(B)
        GOTO LOOP
CONTINUE: …
```

compile(**do** B **while** C) ≡

```
LOOP:   compile(B)
        IF (C)   LOOP
CONTINUE: …
```

I/O transfers



MEMORY MAPPED I/O:

- **Internal Memories of I/O units** are extensions of the Main Memory
- Fully transpartent to the process: LOAD and STORE contain logical addresses
- Compiler annotations drive the physical memory allocation

DIRECT MEMORY ACCESS:

- I/O units can access the Main Memory

SHARED MEMORY:

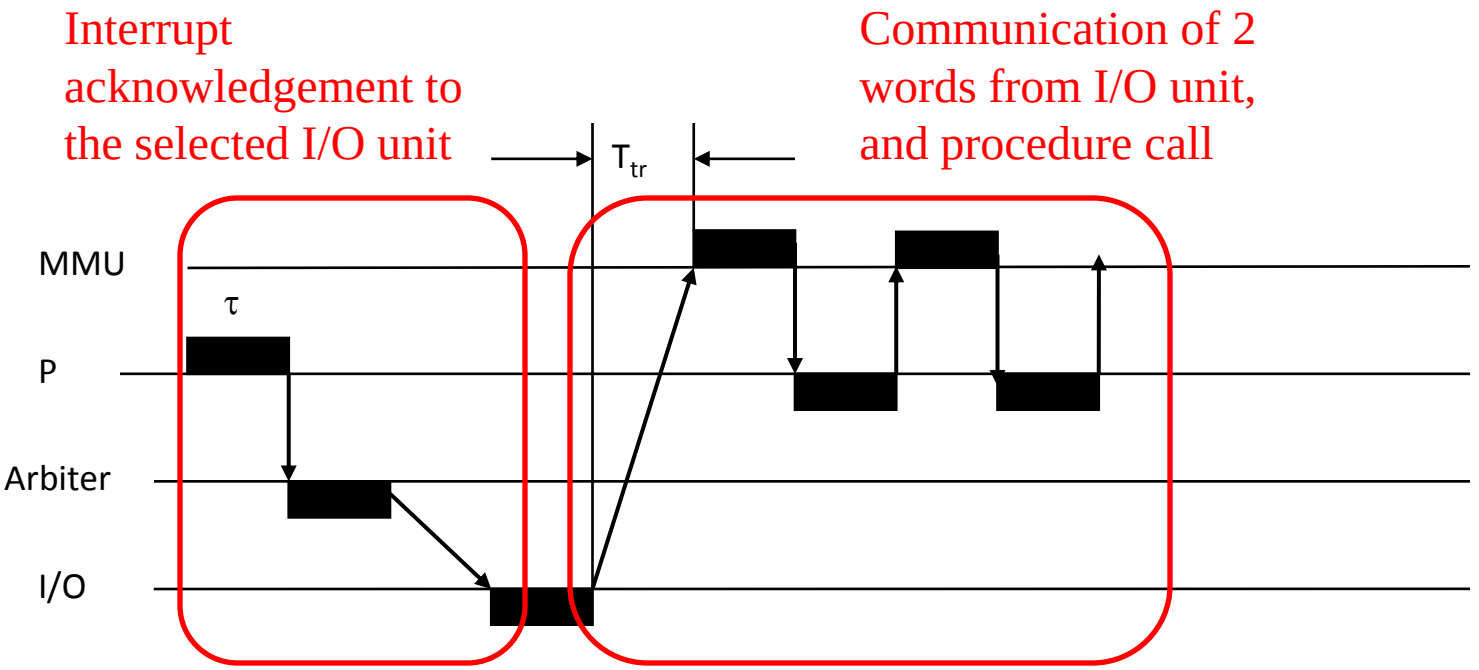
- **Both** techniques support sharing of memory between **CPU-processes** and **I/O-processes** (I/O Memory and/or Main Memory)

Interrupt handling

- Signaling **asynchronous events** (asynchronous wrt the CPU process) from the I/O subsystem
- In the microprogram of every instruction:
  - **Firmware phase**: Test of interrupt signal and call of a proper assembler procedure (HANDLER)
  - **HANDLER**: specific treatment of the event signaled by the I/O unit

- Instead, **exceptions** are **synchronous** events (e.g. memory fault, protection violation, arithmetic error, and so on), i.e. generated by the running process
  - Similar technique for Exception Handling (Firmware phase + Handler)

Firmware phase of interrupt handling



- I/O unit identifier is not associated to the interrupt signal
- Once the interrupt has been acknowledged by P, the I/O unit sends a message to CPU consisting of 2 words (via I/O Bus):
  - An event identifier
  - A parameter