



POLITECNICO
MILANO 1863

DIPARTIMENTO DI ELETTRONICA
INFORMAZIONE E BIOINGEGNERIA

Advanced Computer Architectures

MIPS Pipelining

A.Y. 2024/2025 | Christian Pilato (christian.pilato@polimi.it)

Main characteristics of MIPS architecture

RISC (Reduced Instruction Set Computer) Architecture based on the concept of executing only simple instructions in a reduced basic cycle to optimize the performance of CISC CPUs

LOAD/STORE Architecture

- **ALU operands come from the CPU general purpose registers and cannot directly come from the memory**
- **Dedicated instructions are necessary to:**
 - **load data from memory to registers**
 - **store data from registers to memory**

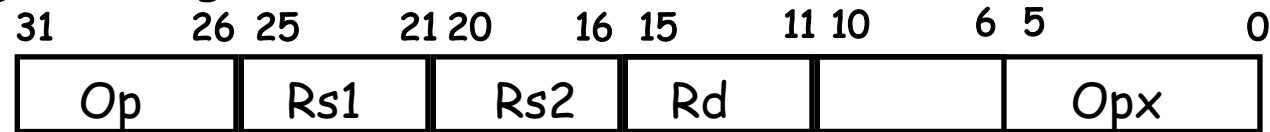
Pipelining

- **Performance optimization technique based on the overlapping of the execution of multiple instructions derived from a sequential execution flow**

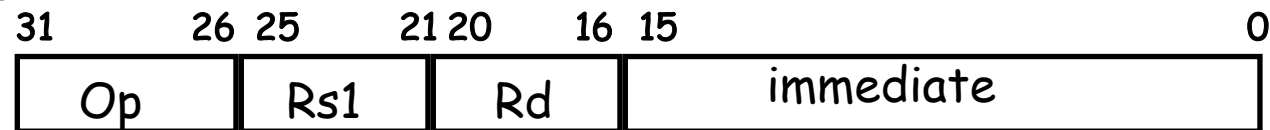
MIPS ISA example

The **Instruction Set Architecture** defines the set of operations, instruction format, hardware-supported data types, named storage, addressing modes, sequencing

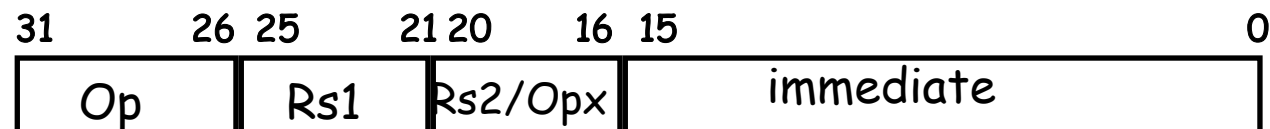
Register-Register



Register-Immediate



Branch



Jump / Call



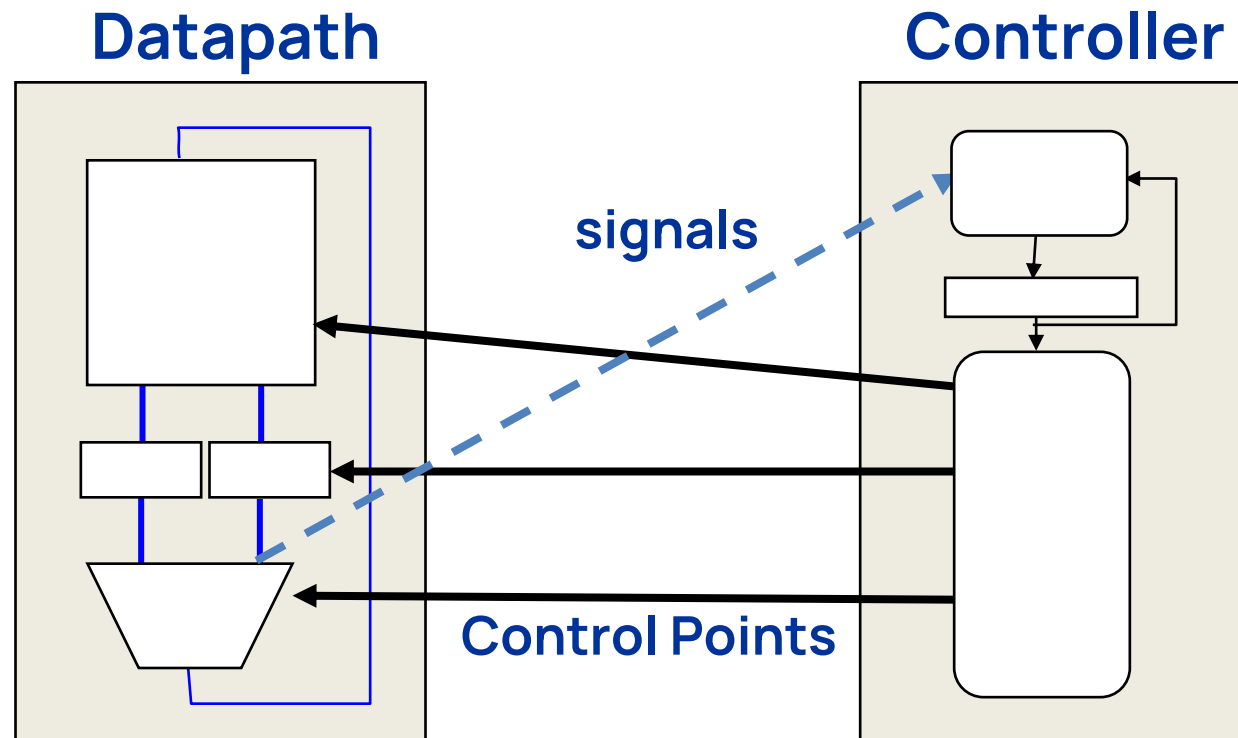
Data vs Control

Datapath: Storage, FU, interconnections sufficient to perform the desired functions

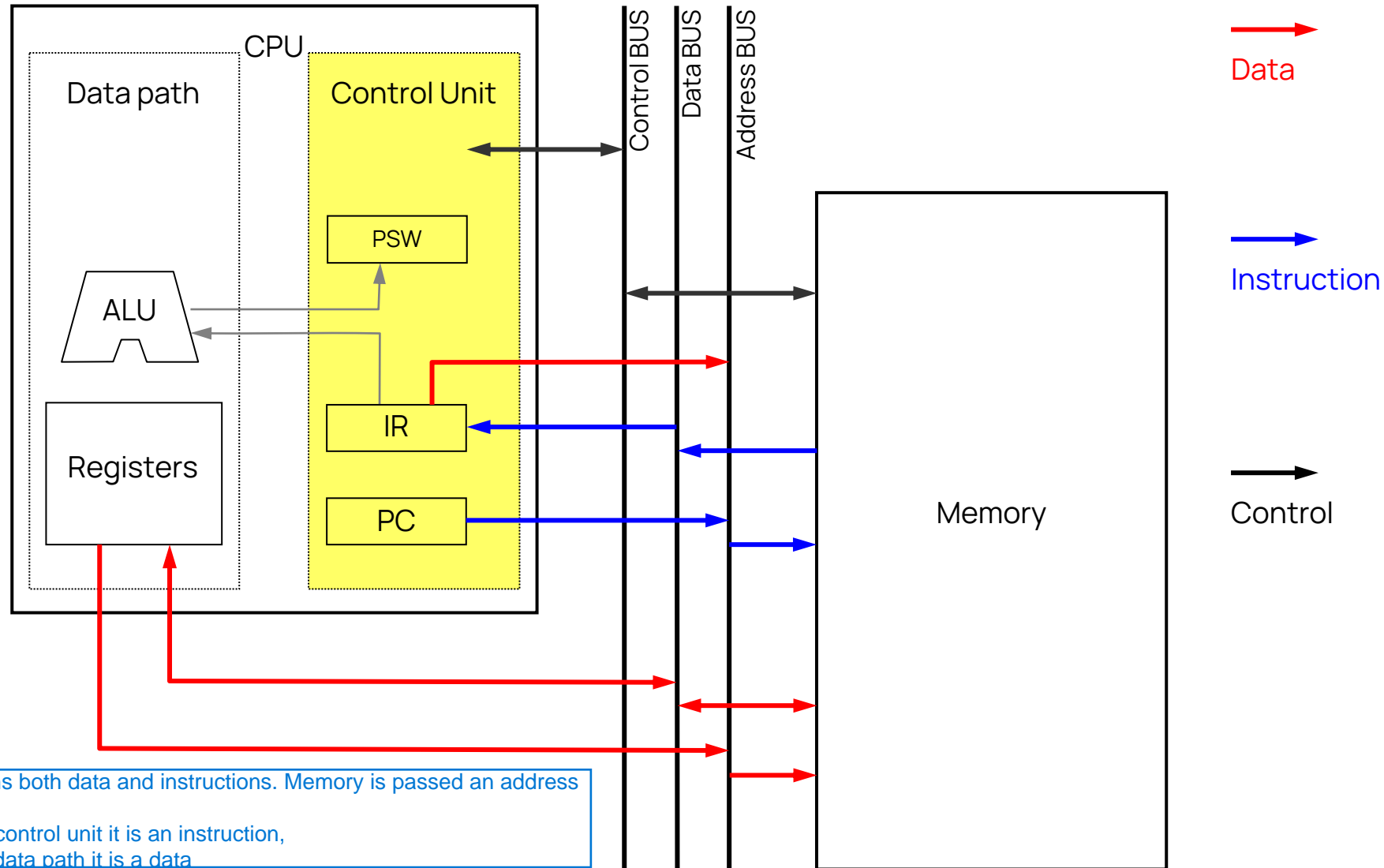
Inputs are Control Points, Outputs are signals

Controller: State machine to orchestrate operations in the data path

Based on the desired function and signals



Computing infrastructure



The memory contains both data and instructions. Memory is passed an address in both cases and:
If it comes from the control unit it is an instruction,
if it comes from the data path it is a data

The program...

...

```
k=b+c+d;
```

...



Breaking down performance

A **program** is broken into **instructions**

- Hardware is aware of instructions, not programs

At a lower level, hardware breaks **instructions** into **clock cycles**

Lower-level state machines change state every cycle

For **example**:

500 MHz P-III runs 500M cycles/sec, 1 cycle = 2 ns

2 GHz P-IV runs 2G cycles/sec, 1 cycle = 0,5 ns

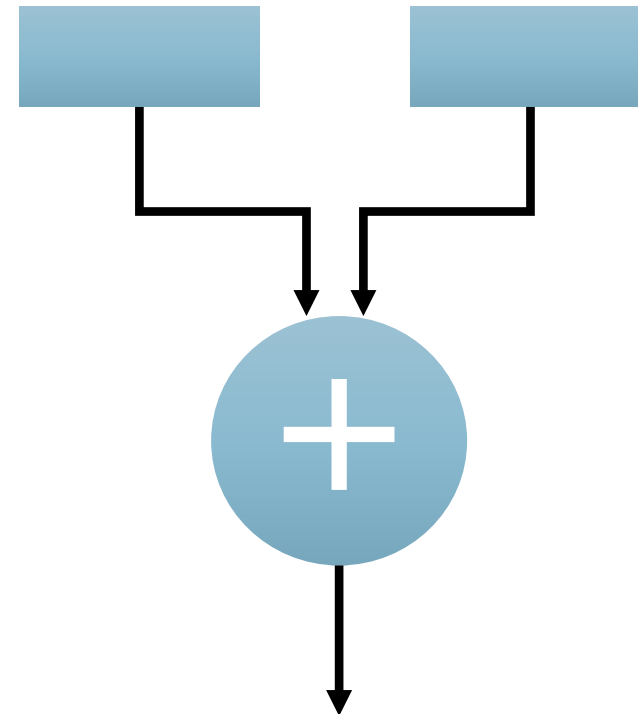
The program and the ISA

Every instruction must be converted into its **three-address code equivalent**

...

$k = b + c + d;$

...



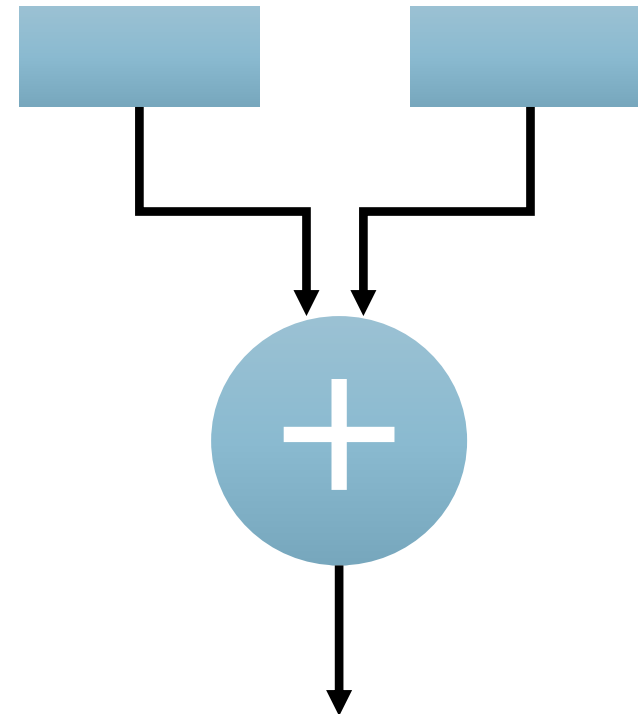
The program and the ISA

...

$k = b + c;$

$k = k + d;$

...



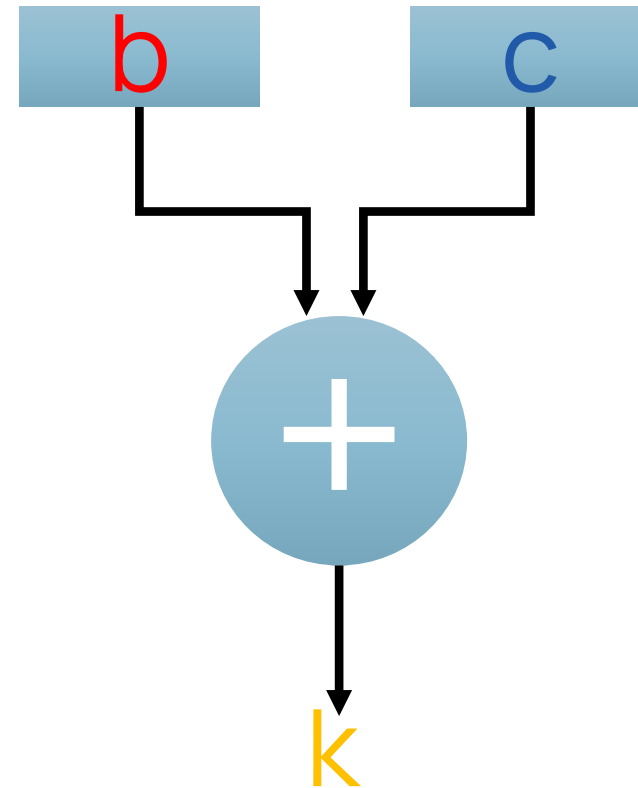
The program and the ISA

...

$k = b + c;$

$k = k + d;$

...



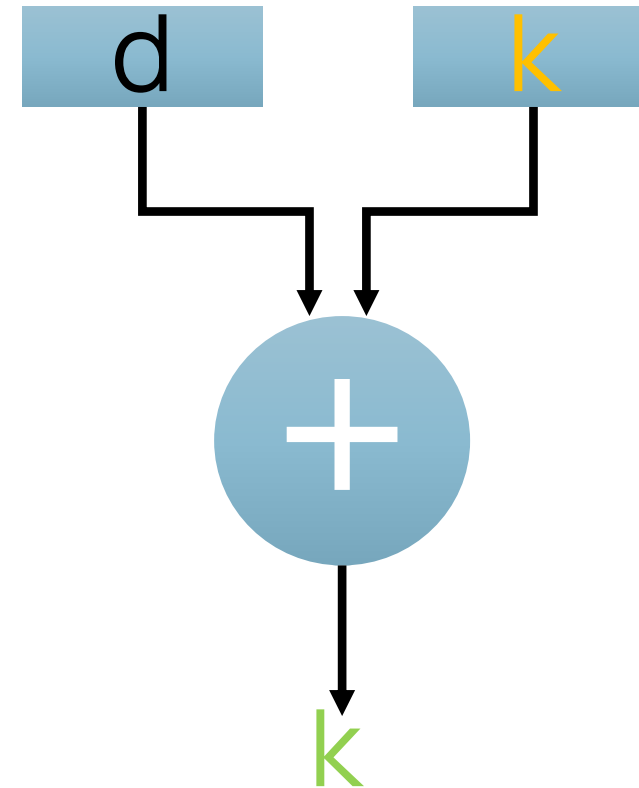
The program and the ISA

...

$k = b + c;$

$k = k + d;$

...





The Instructions and the ISA

Reduced instruction set of MIPS processor

- **ALU instructions:**

```
add $s1, $s2, $s3      # $s1 ← $s2 + $s3
addi $s1, $s1, 4        # $s1 ← $s1 + 4
```

- **Load/store instructions:**

```
lw $s1, offset ($s2)   # $s1 ← M[$s2+offset]
sw $s1, offset ($s2)   # M[$s2+offset] ← $s1
```

- **Branch instructions** to control the control flow of the program:

- Conditional branches: the branch is taken only if the condition is satisfied. Examples: `beq` (branch on equal) and `bne` (branch on not equal)

```
beq $s1, $s2, L1        # go to L1 if ($s1 == $s2)
bne $s1, $s2, L1        # go to L1 if ($s1 != $s2)
```

- Unconditional jumps: the branch is always taken

Examples: `j` (jump) and `jr` (jump register)

```
j    L1                # go to L1
jr   $s1                # go to add. contained in $s1
```

Execution of MIPS Instructions

ALU Instructions: **op \$x, \$y, \$z**

Instr. Fetch & PC Increm.	Read of Source Regs. \$y and \$z	ALU OP (\$y op \$z)	Write Back of Destinat. Reg. \$x
------------------------------	-------------------------------------	--------------------------	-------------------------------------

Load Instructions: **l w \$x, of f set (\$y)**

Instr. Fetch & PC Increm.	Read of Base Reg. \$y	ALU Op. (\$y+of f set)	Read Mem. M \$y+of f set)	Write Back of Destinat. Reg. \$x
------------------------------	--------------------------	-----------------------------	-------------------------------	-------------------------------------

Store Instructions: **sw \$x, of f set (\$y)**

Instr. Fetch & PC Increm.	Read of Base Reg. \$y & Source \$x	ALU Op. (\$y+of f set)	Write Mem. M \$y+of f set)
------------------------------	---------------------------------------	-----------------------------	--------------------------------

Conditional Branch: **beq \$x, \$y, of f set**

Instr. Fetch & PC Increm.	Read of Source Regs. \$x and \$y	ALU Op. (\$x- \$y) & (PC+4+of f set)	Write PC
------------------------------	-------------------------------------	--	-------------

Execution of MIPS Instructions

Every instruction in the MIPS subset can be implemented in at most **five clock cycles** as follows:

1. Instruction **Fetch** (from memory)

- Send the content of Program Counter register to Instruction Memory and fetch the current instruction from Instruction Memory
- Update the PC to the next sequential address by adding 4 to the PC (since each instruction is 4 bytes) [prepare for the next instruction in memory (which will be the next one to be executed unless there is a jump)]

2. Instruction **Decode** and **Register Read**

- Decode the current instruction (fixed-field decoding) and read from the Register File of one or two registers corresponding to the registers specified in the instruction fields.
- Sign-extension of the offset field of the instruction in case it is needed.

Execution of MIPS instructions

3. Execution

The ALU operates on the operands prepared in the previous cycle depending on the instruction type:

- Register-Register ALU Instructions:
 - ALU executes the specified operation on the operands read from the RF
- Register-Immediate ALU Instructions:
 - ALU executes the specified operation on the first operand read from the RF and the sign-extended immediate operand
- Memory Reference:
 - ALU adds the base register and the offset to calculate the effective address.
- Conditional branches:
 - Compare the two registers read from RF and compute the possible branch target address by adding the sign-extended offset to the incremented PC.

Execution of MIPS instructions

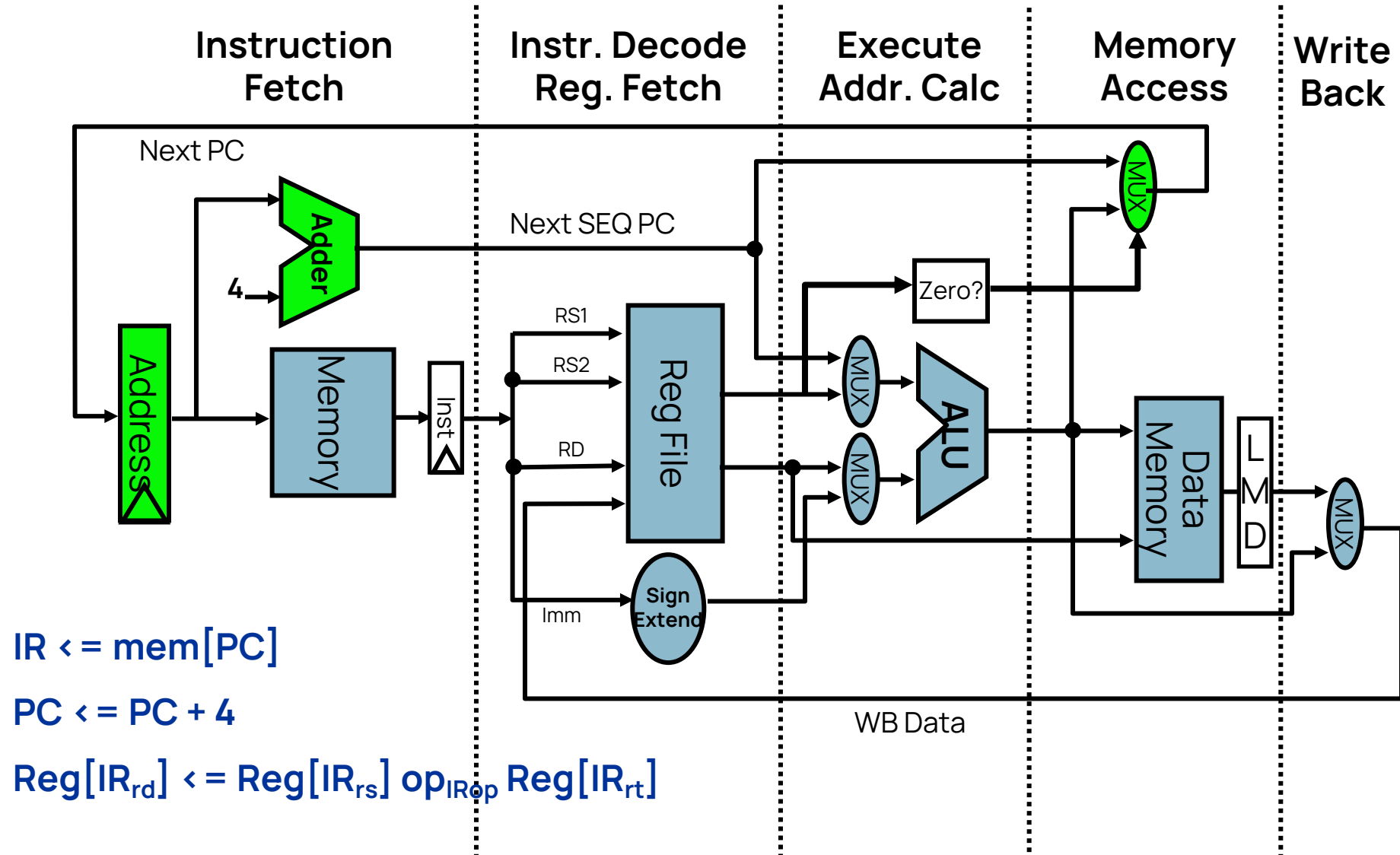
4. Memory Access (ME)

- Load instructions require a read access to the Data Memory using the effective address
- Store instructions require a write access to the Data Memory using the effective address to write the data from the source register read from the RF
- Conditional branches can update the content of the PC with the branch target address if the conditional test yields true

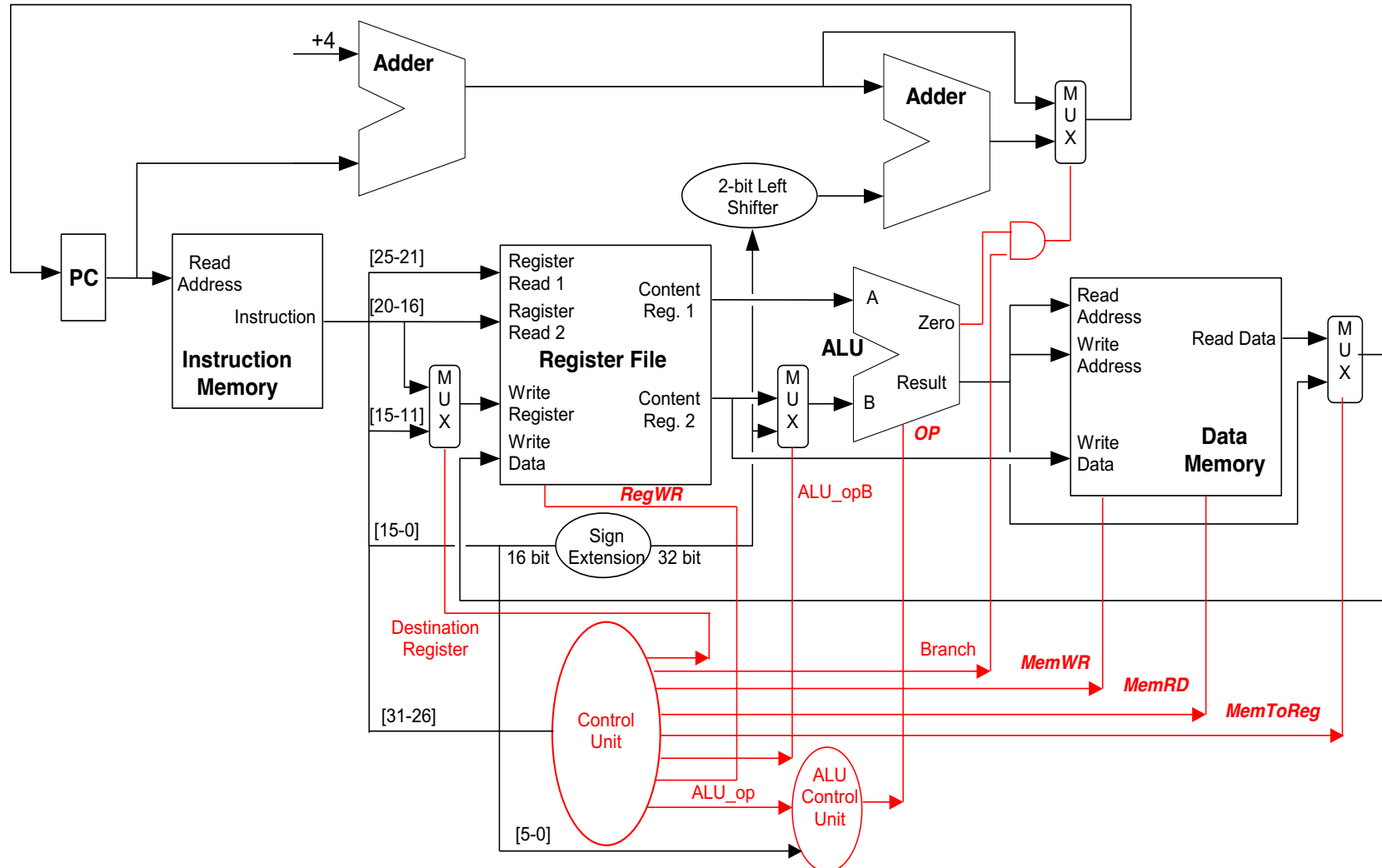
5. Write-Back (WB)

- Load instructions write the data read from memory in the destination register of the RF
- ALU instructions write the ALU results into the destination register of the RF

MIPS Data path



Implementation of MIPS data path with **Control Unit**



Instructions Latency

Instruction Type	Instruct. Mem.	Register Read	ALU Op.	Data Memory	Write Back	Total Latency
ALU Instr.	2	1	2	0	1	6 ns
Load	2	1	2	2	1	8 ns
Store	2	1	2	2	0	7 ns
Cond. Branch	2	1	2	0	0	5 ns
Jump	2	0	0	0	0	2 ns

Single-cycle implementation

The length of the clock cycle is defined by the **critical path given by the load instruction**:

$$T = 8 \text{ ns} \text{ (} f = 125 \text{ MHz)}$$

- We **assume each instruction is executed in a single clock cycle**
 - Each module must be used once in a clock cycle
 - The modules used more than once in a cycle must be duplicated
- We need an Instruction Memory separated from the Data Memory
- Some modules must be duplicated, while other modules must be shared from different instruction flows
- To share a module between two different instructions, we need a multiplexer to enable multiple inputs to a module and select one of the different inputs based on the configuration of control lines

Multi-cycle implementation

The instruction execution is distributed on multiple cycles (5 cycles for MIPS)

The basic cycle is smaller (2 ns \Rightarrow instruction latency = 10 ns)

Implementation of multi-cycle CPU:

- Each phase of the instruction execution requires a clock cycle
- Each module can be used more than once per instruction in different clock cycles
 - possible sharing of modules
- We need internal registers to store the values to be used in the next clock cycles

Pipelining

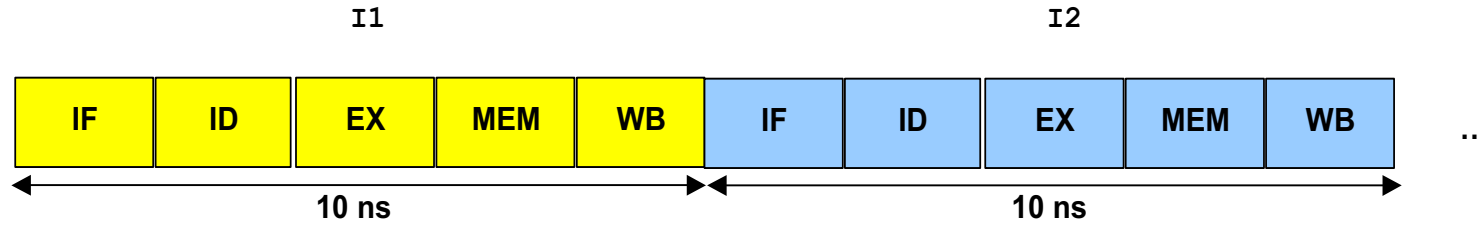
Performance optimization technique based on the overlap of the execution of multiple instructions deriving from a sequential execution flow

Pipelining exploits the parallelism among instructions in a sequential instruction stream

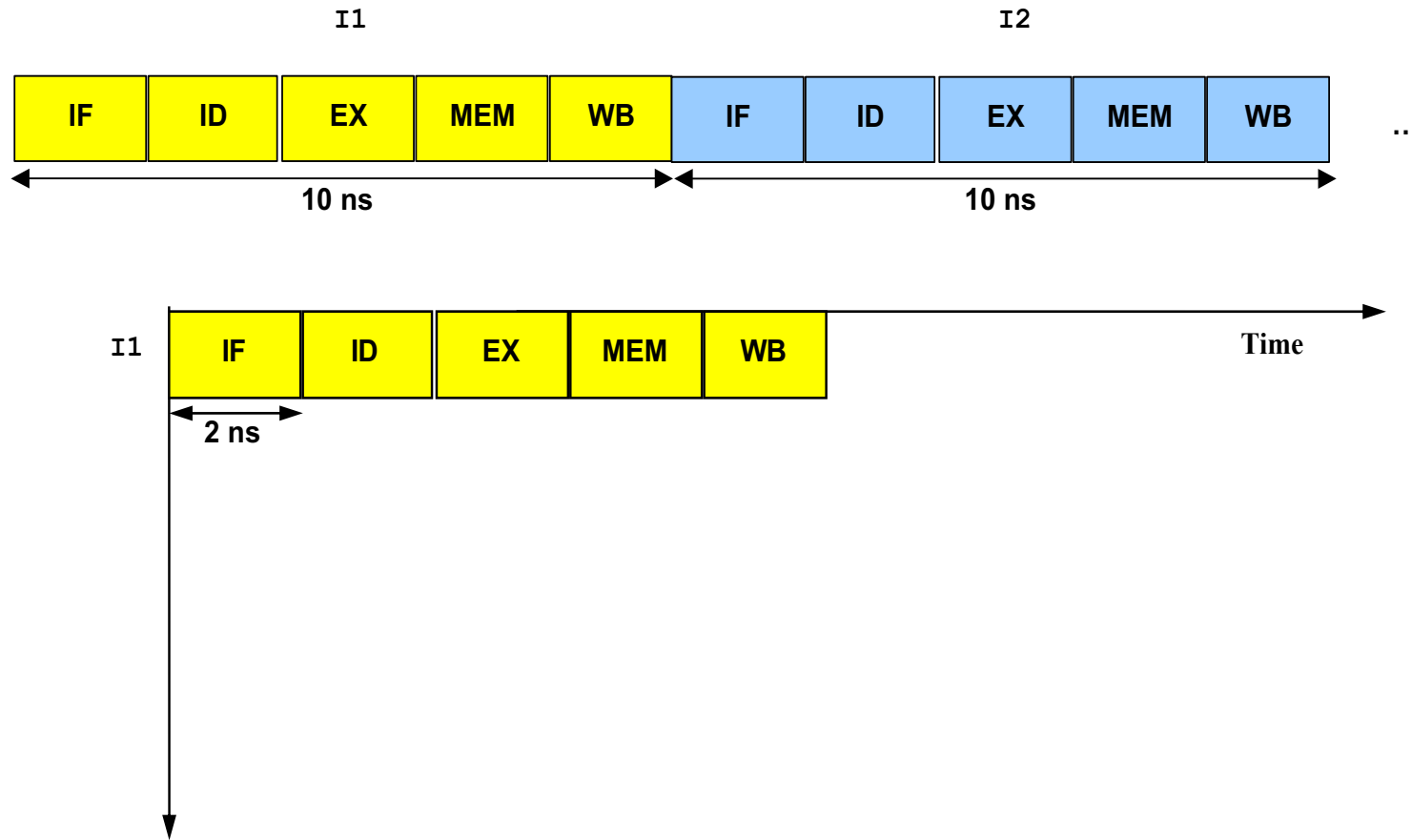
Basic idea:

- The execution of an instruction is divided into different phases (**pipeline stages**), requiring a fraction of the time necessary to complete the instruction
- The stages are connected one to the next to form the pipeline: instructions enter in the pipeline at one end, progress through the stages, and exit from the other end, as in an assembly line

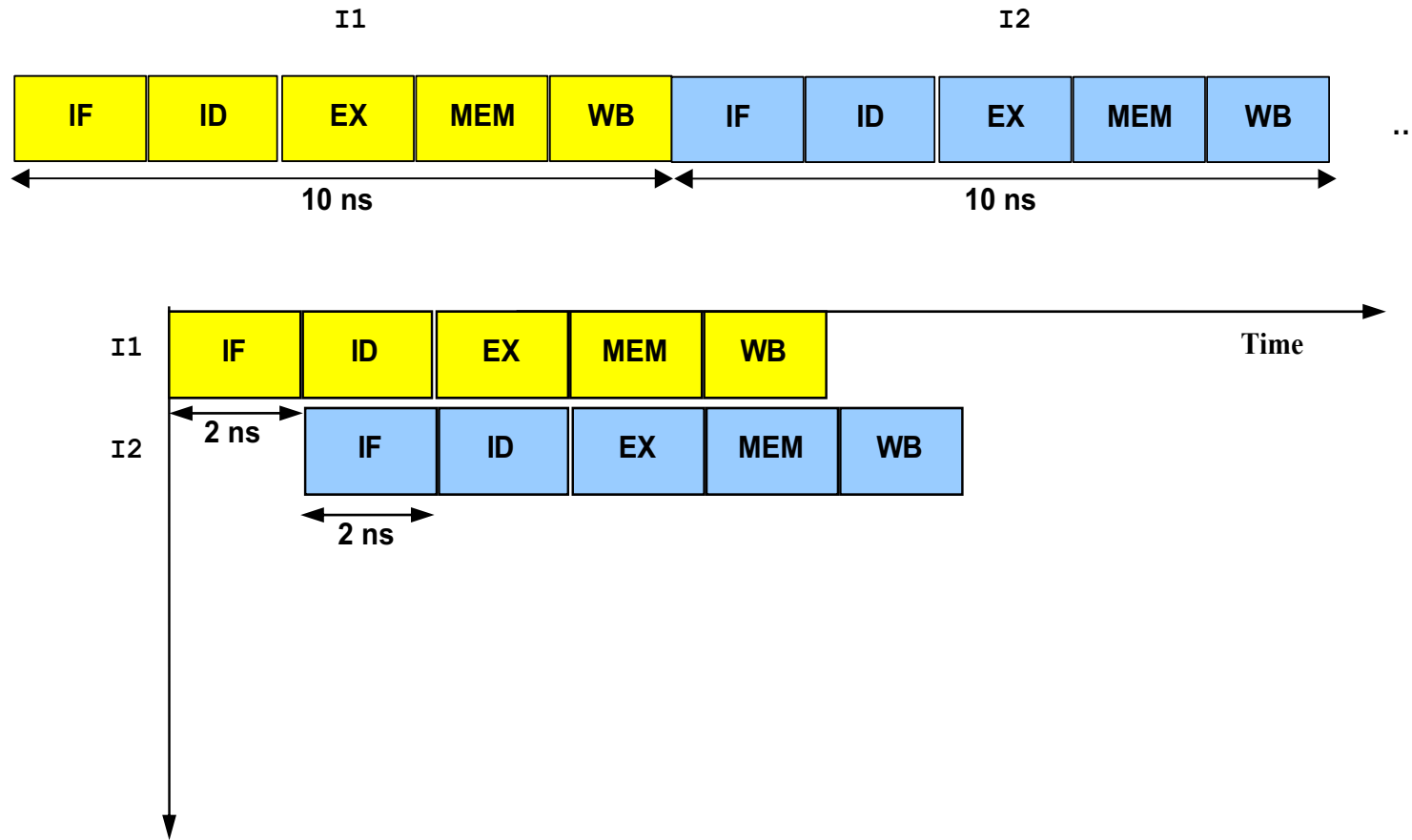
Sequential vs. Pipelining Execution



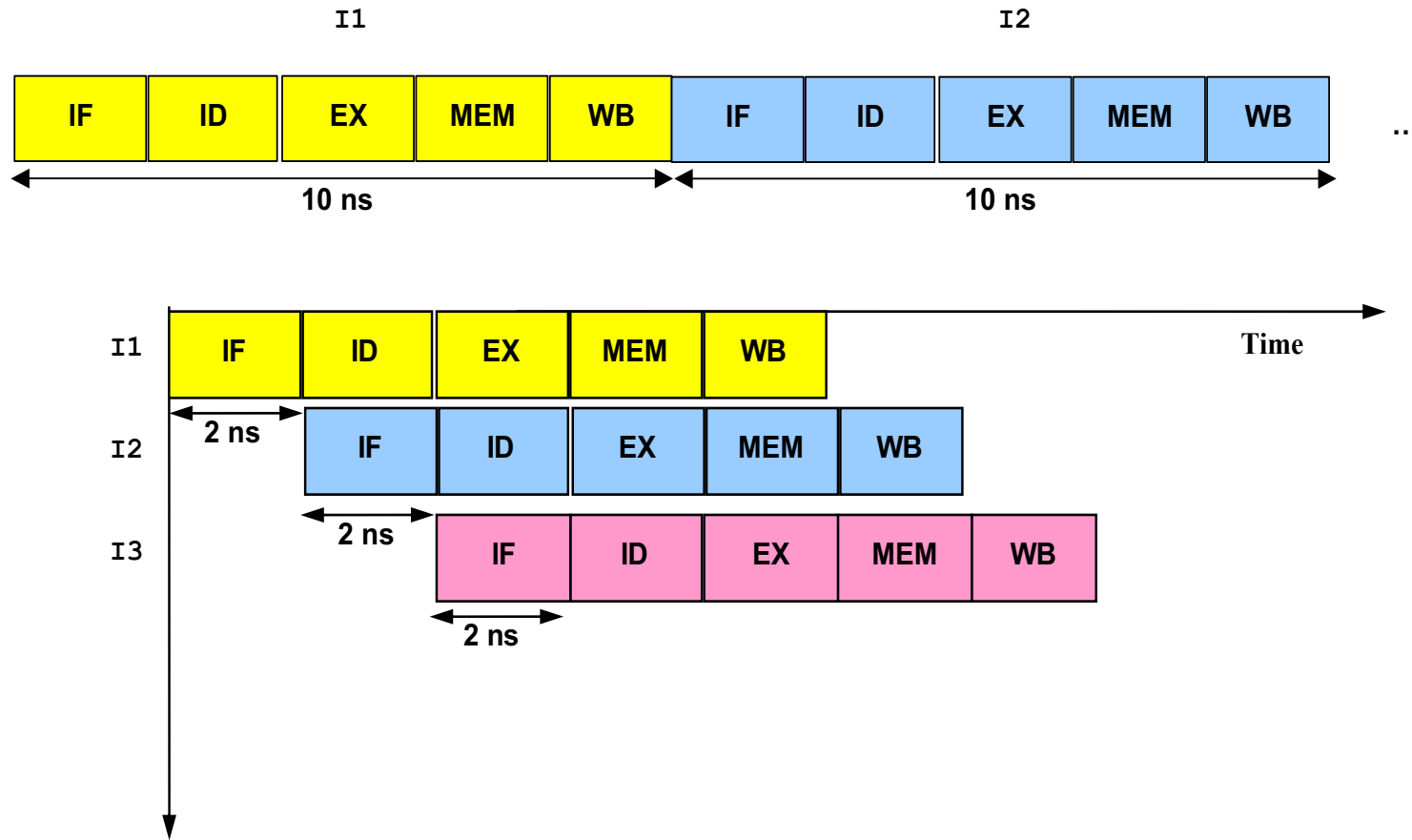
Sequential vs. Pipelining Execution



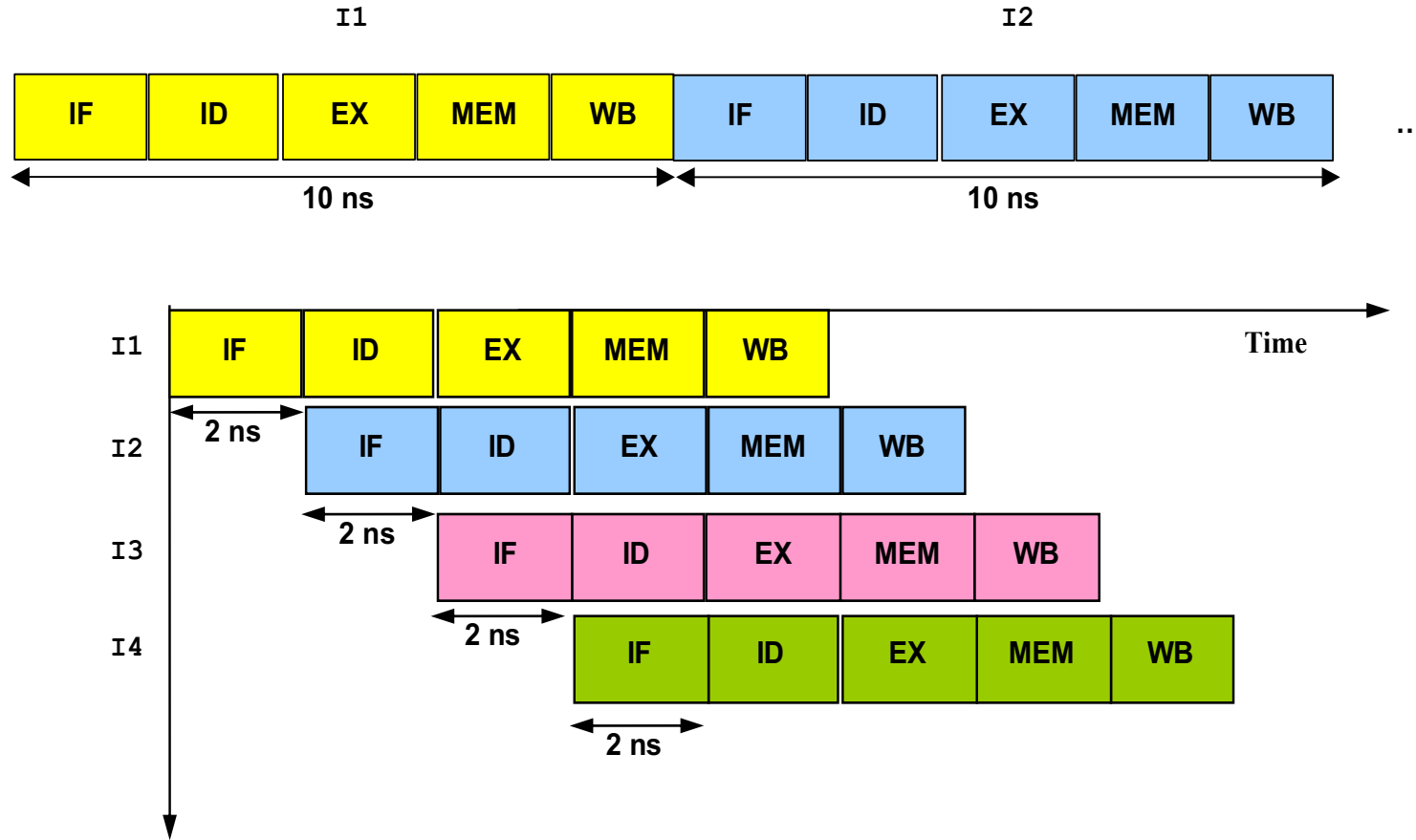
Sequential vs. Pipelining Execution



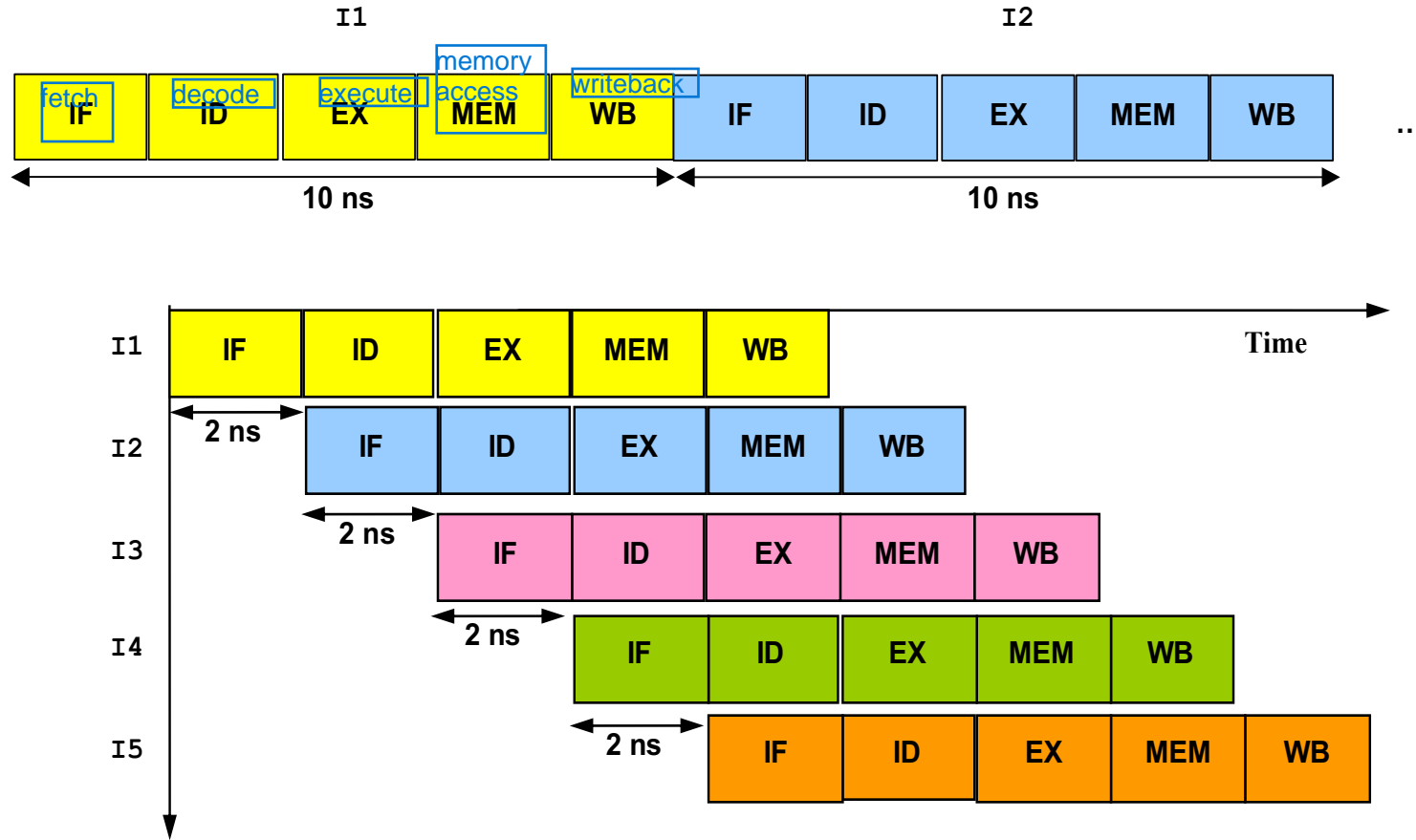
Sequential vs. Pipelining Execution



Sequential vs. Pipelining Execution



Sequential vs. Pipelining Execution



Pipelining

The time to advance the instruction of one stage in the pipeline corresponds to a clock cycle

The pipeline stages must be synchronized: the duration of a clock cycle is defined by the time requested by the slower stage of the pipeline (i.e., 2 ns)

The goal is to balance the length of each pipeline stage

If the stages are perfectly balanced, the **ideal speedup** due to pipelining is equal to the number of pipeline stages

Performance improvement

Ideal case (asymptotically):

If we consider the **single-cycle unpipelined CPU1** with clock cycle of **8ns** and the **pipelined CPU2** with 5 stages of **2ns**:

- The **latency** (total execution time) of each instruction **is worsened**: from **8ns** to **10ns**
- The **throughput** (number of instructions completed in the time unit) **is improved** of 4 times:
1 instruction completed each 8ns vs. **1 instruction completed each 2ns**

Performance improvement

Ideal case (asymptotically):

If we consider the multi-cycle unpipelined CPU3 composed of 5 cycles of 2ns and the pipelined CPU2 with 5 stages of 2ns:

- The latency (total execution time) of each instruction is not varied (10ns)
- The throughput (number of instructions completed in the time unit) is improved by 5 times:

1 instruction completed every 10ns vs. 1 instruction completed every 2ns

Pipeline Execution of MIPS Instructions

IF Instruction Fetch	ID Instruction Decode	EX Execution	ME Memory Access	WB Write Back
-------------------------	--------------------------	-----------------	---------------------	------------------

ALU Instructions: `op $x, $y, $z`

Instr. Fetch & PC Increm.	Read of Source Regs. \$y and \$z	ALU Op. (\$y op \$z)		Write Back Destinat. Reg. \$x
------------------------------	-------------------------------------	-------------------------	--	----------------------------------

Load Instructions: `lw $x, offset($y)`

Instr. Fetch & PC Increm.	Read of Base Reg. \$y	ALU Op. (\$y+offset)	Read Mem. M(\$y+offset)	Write Back Destinat. Reg. \$x
------------------------------	--------------------------	-------------------------	----------------------------	----------------------------------

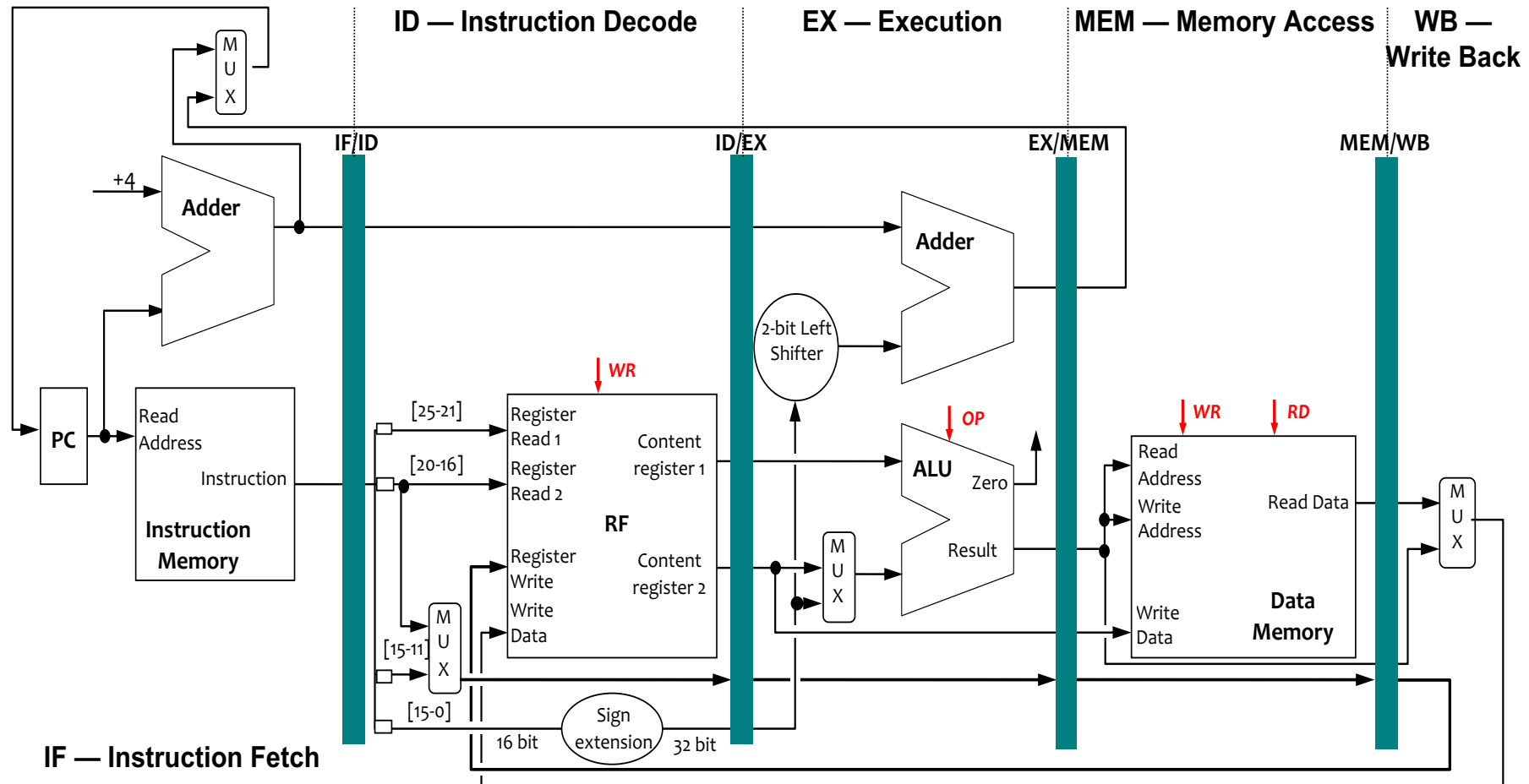
Store Instructions: `sw $x, offset($y)`

Instr. Fetch & PC Increm.	Read of Base Reg. \$y & Source \$x	ALU Op. (\$y+offset)	Write Mem. M(\$y+offset)	
------------------------------	---------------------------------------	-------------------------	-----------------------------	--

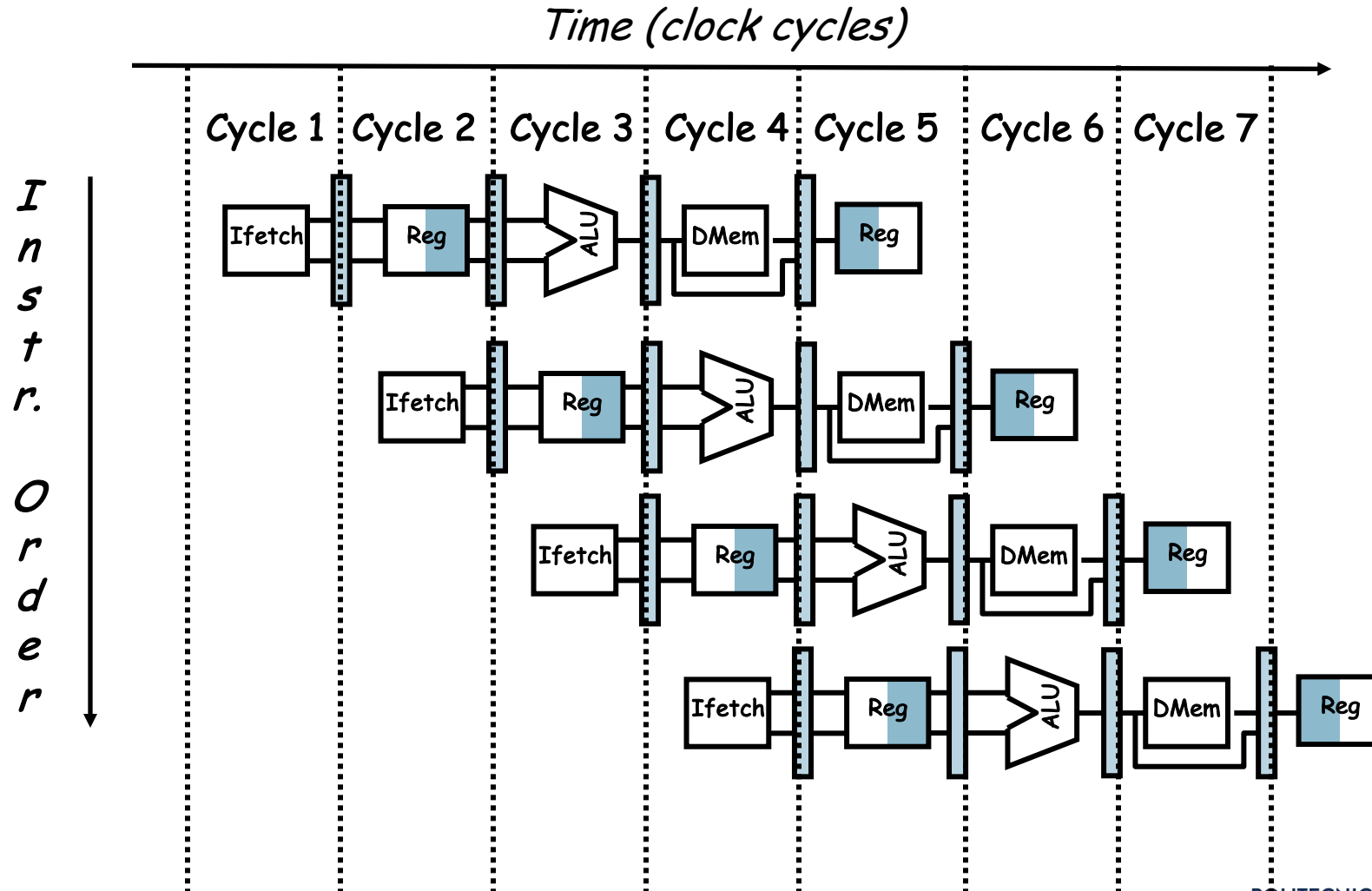
Conditional Branches: `beq $x, $y, offset`

Instr. Fetch & PC Increm.	Read of Source Regs. \$x and \$y	ALU Op. (\$x-\$y) & (PC+4+offset)	Write PC	
------------------------------	-------------------------------------	--------------------------------------	-------------	--

Implementation of MIPS pipeline



Visualizing Pipelining



Note: a possible issue

Register File used in 2 stages: Read access during ID and write access during WB

What happens if **read** and **write** refer to the same register in the same clock cycle?

It is necessary to insert one stall

Issue as an Opportunity

Register File used in 2 stages: Read access during ID and write access during WB

What happens if **read** and **write** refer to the same register in the same clock cycle?

It is necessary to insert one stall

Optimized Pipeline: the RF read occurs in the second half of the clock cycle and the RF write in the first half of the clock cycle

What happens if **read** and **write** refer to the same register in the same clock cycle?

Now, it is not necessary to insert the stall anymore

**From now on,
this is the Pipeline
we are going to use**

The Problem of Hazards

A hazard is created whenever there is a dependence between instructions, and instructions are close enough that the overlap caused by pipelining would change the order of access to the operands involved in the dependence

Hazards prevent the next instruction in the pipeline from executing during its designated clock cycle

Hazards reduce the performance from the ideal speedup gained by pipelining

Example

```
I:  add  r1, r2, r3  
J:  sub  r4, r1, r3
```

Example

I: add **r1**, r2, r3
J: sub r4, **r1**, r3

First instruction is writing r1 and second one is writing r1.

Three Classes of Hazards

Structural Hazards: Attempt to use the same resource from different instructions simultaneously

Example: Single memory for instructions and data

Control Hazards: Attempt to make a decision on the next instruction to execute before the condition is evaluated

Example: Conditional branch execution

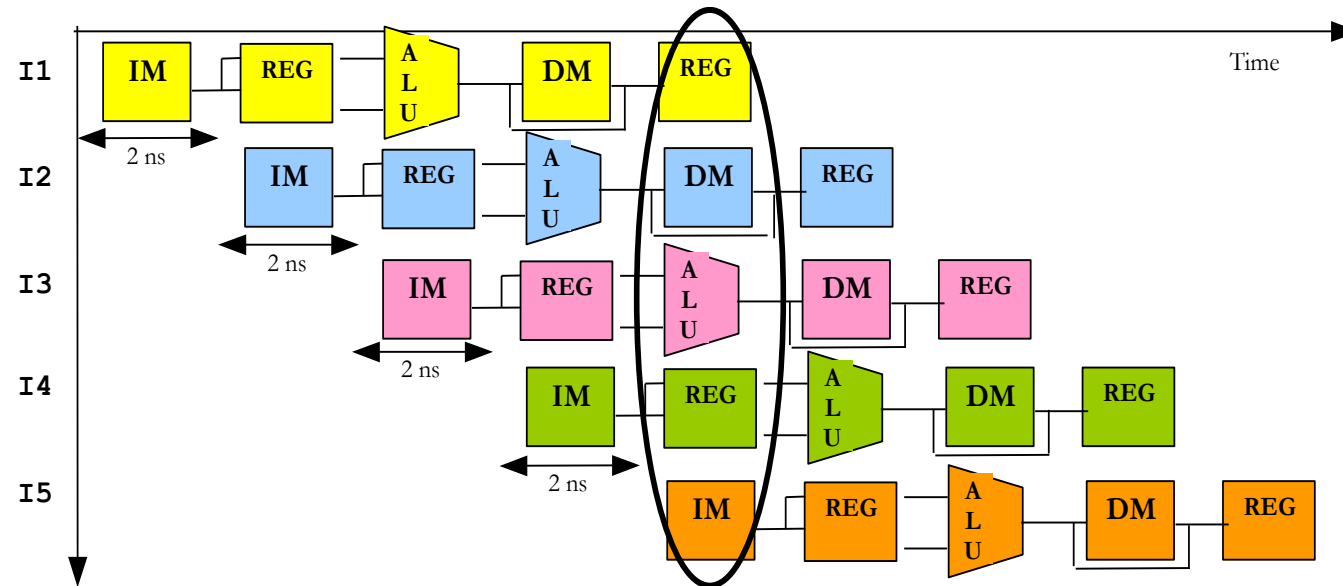
Data Hazards: Attempt to use a result before it is ready [\(the example in the previous slide\)](#)

Example: Instruction depending on a result of a previous instruction still in the pipeline

Structural Hazards

No structural hazards in MIPS architecture:

- Instruction Memory separated from Data Memory
- Register File used in the same clock cycle: Read access by an instruction and write access by another instruction



Data Hazards

If the instructions executed in the pipeline are dependent, data hazards can arise when instructions are too close

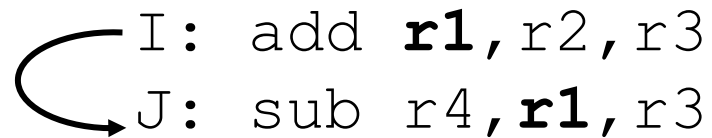
Example:

sub	\$2 , \$1, \$3	# Reg. \$2 written by sub
and	\$12, \$2 , \$5	# 1° operand (\$2) depends on sub
or	\$13, \$6, \$2	# 2° operand (\$2) depend on sub
add	\$14, \$2 , \$2	# 1° (\$2) & 2° (\$2) depend on sub
sw	\$15, 100 (\$2)	# Base reg. (\$2) depends on sub

Types of Data Hazards

Read After Write (RAW)

- Instr_j tries to read operand before Instr_i writes it
- Caused by a “**Dependence**” (in compiler nomenclature). This hazard results from an actual need for communication

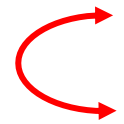


```
I: add r1, r2, r3
J: sub r4, r1, r3
```

Types of Data Hazards

Write After Write (WAW)

- Instruction $n+1$ tries to write a destination operand before it has been written by the previous instruction n
- Called an “output dependence”




```
I: sub r1, r4, r3
J: add r1, r2, r3
K: mul r6, r1, r7
```

- This also results from the reuse of the name “**r1**”.
- NOTE: **Can't happen in MIPS 5** stage pipeline because all instructions take 5 stages, and writes are always in stage 5

Types of Data Hazards

Write After Read (WAR)

- Instruction n+1 tries to write a destination operand before it has been read from the previous instruction n (which can read a wrong value)
- Called an “**anti-dependence**”



```
I: sub r4,r1,r3  
J: add r1,r2,r3  
K: mul r6,r1,r7
```

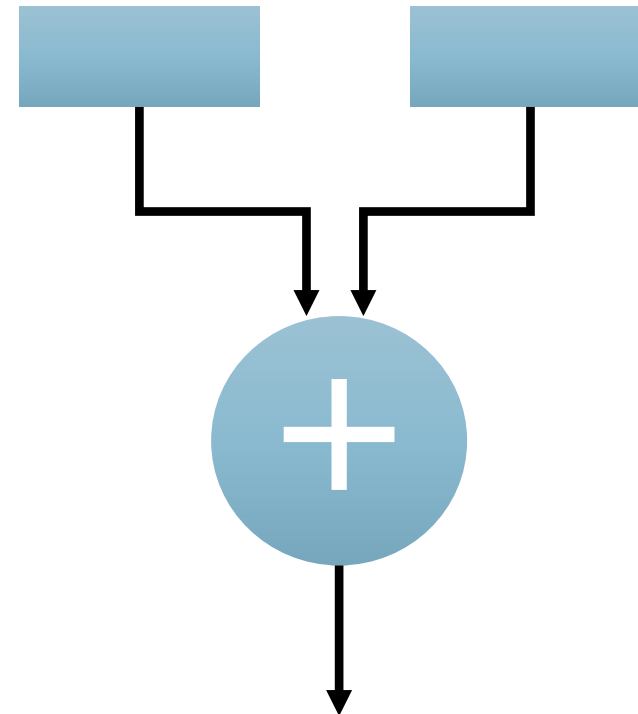
- This also results from the reuse of the name “**r1**”.
- NOTE: **Can't happen in MIPS 5** stage pipeline because all instructions take five stages, and reads are always in stage 2 and writes are always in stage 5

We will see WAR and WAW in more complicated pipelines

Appendix: Example

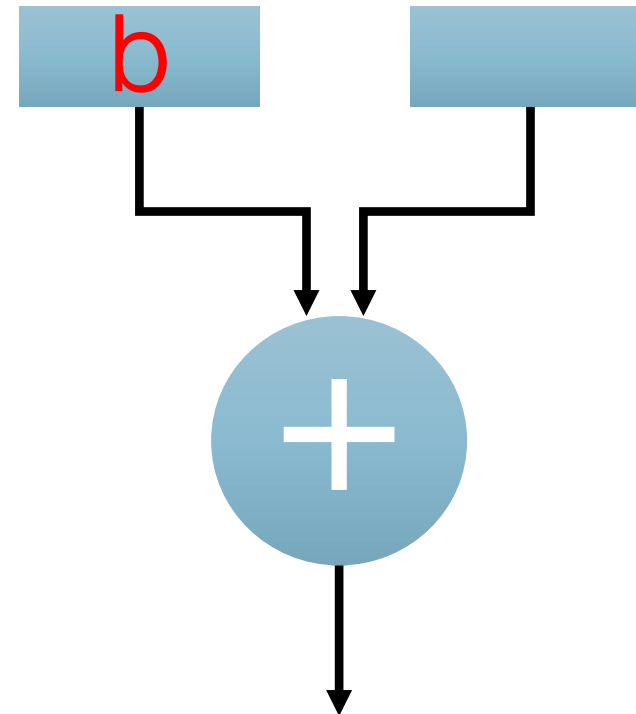
The instructions and the ISA

...
0789	load	R02,4000		
0790	load	R03,4004		
0791	add	R01,R02,R03		
0792	load	R02,4008		
0793	add	R01,R01,R02		
0794	store	R01,4000		
...



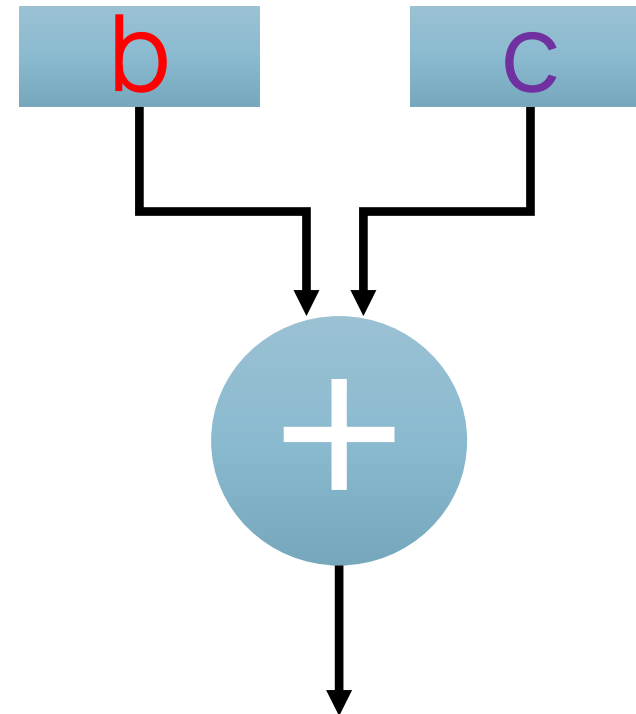
The instructions and the ISA

...
0789	load	R02,4000		
0790	load	R03,4004		
0791	add	R01,R02,R03		
0792	load	R02,4008		
0793	add	R01,R01,R02		
0794	store	R01,4000		
...



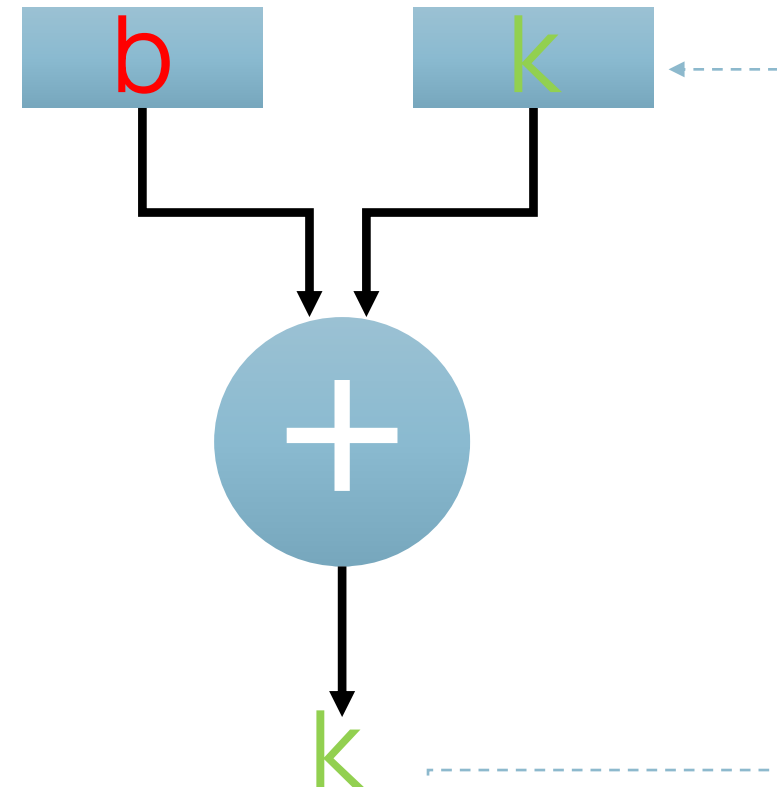
The instructions and the ISA

...
0789	load	R02,4000		
0790	load	R03,4004		
0791	add	R01,R02,R03		
0792	load	R02,4008		
0793	add	R01,R01,R02		
0794	store	R01,4000		
...



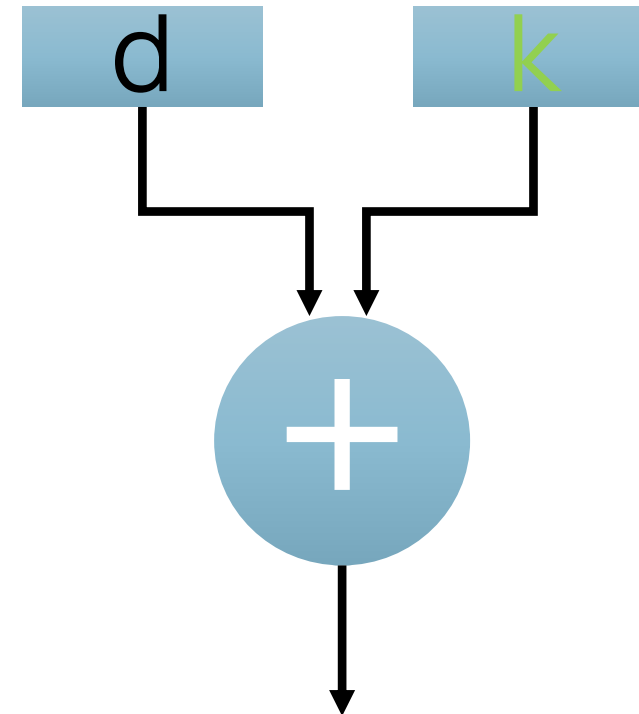
The instructions and the ISA

...
0789	load	R02,4000		
0790	load	R03,4004		
0791	add	R01,R02,R03		
0792	load	R02,4008		
0793	add	R01,R01,R02		
0794	store	R01,4000		
...



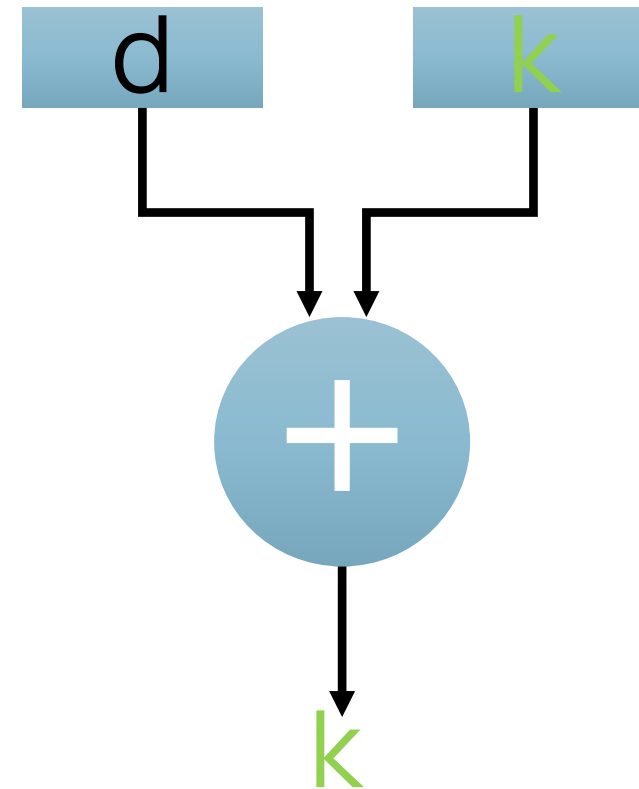
The instructions and the ISA

...
0789	load	R02,4000		
0790	load	R03,4004		
0791	add	R01,R02,R03		
0792	load	R02,4008		
0793	add	R01,R01,R02		
0794	store	R01,4000		
...

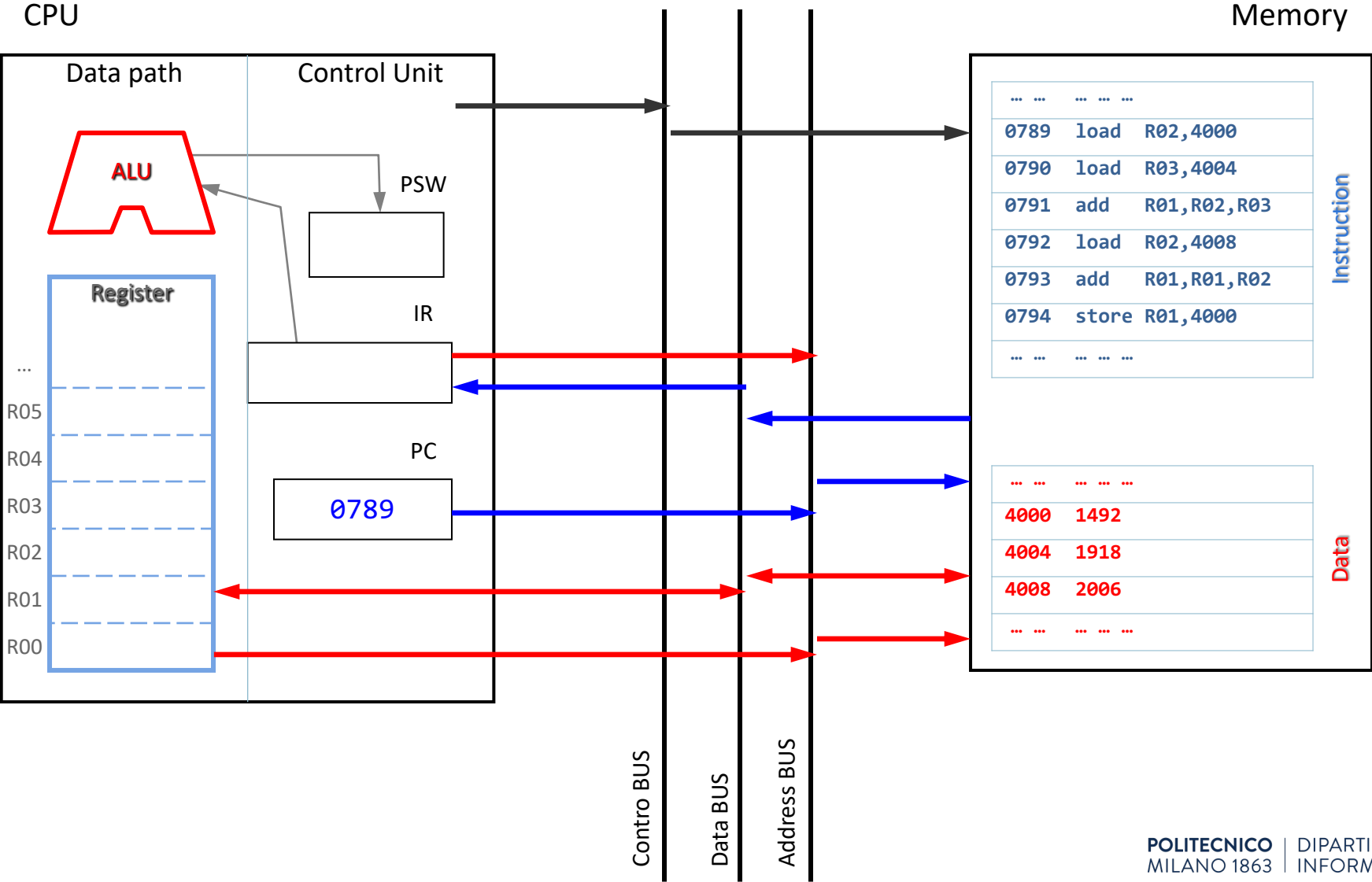


The Instructions and the ISA

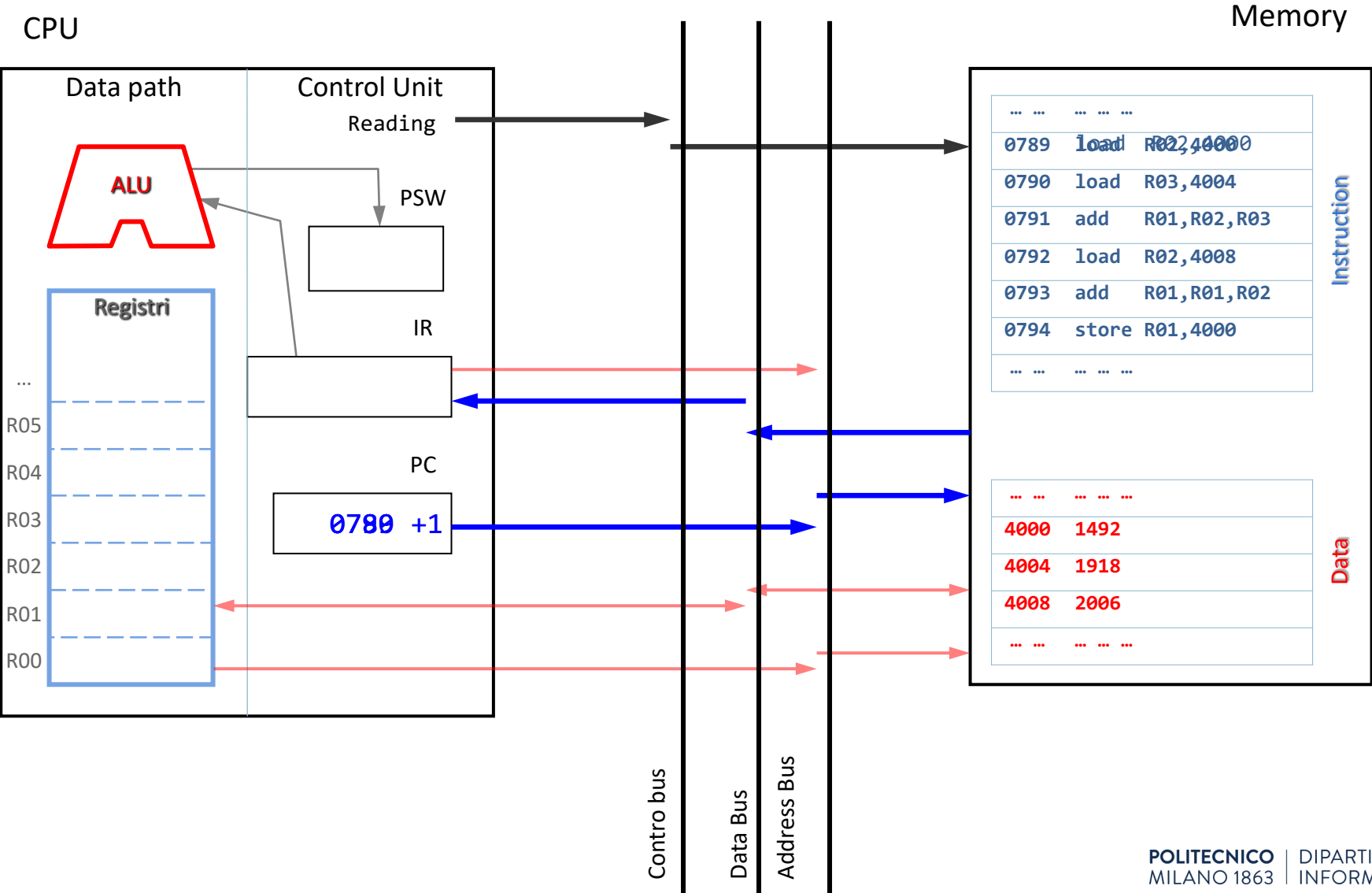
...
0789	load	R02,4000		
0790	load	R03,4004		
0791	add	R01,R02,R03		
0792	load	R02,4008		
0793	add	R01,R01,R02		
0794	store	R01,4000		
...



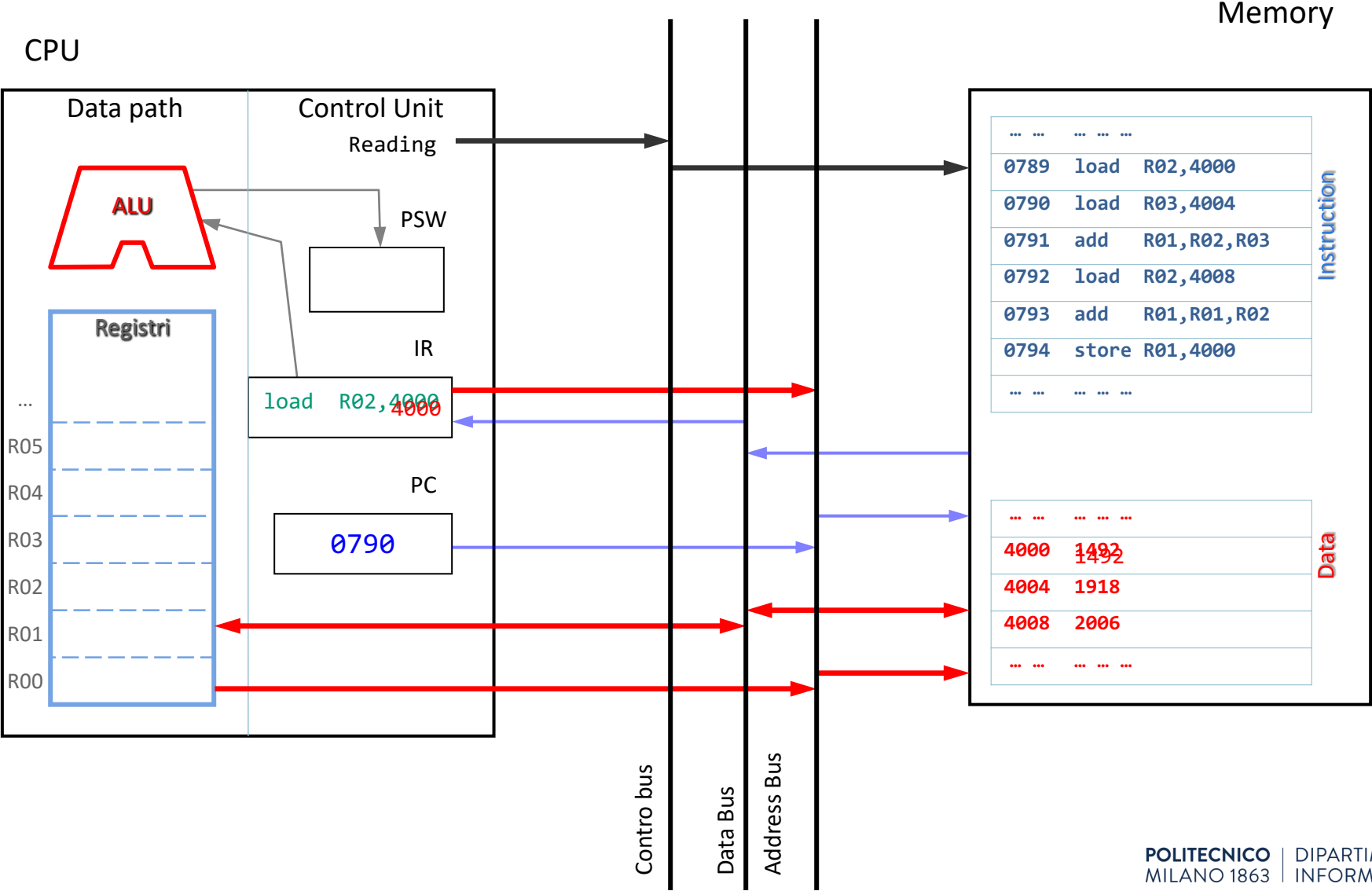
Starting scenario



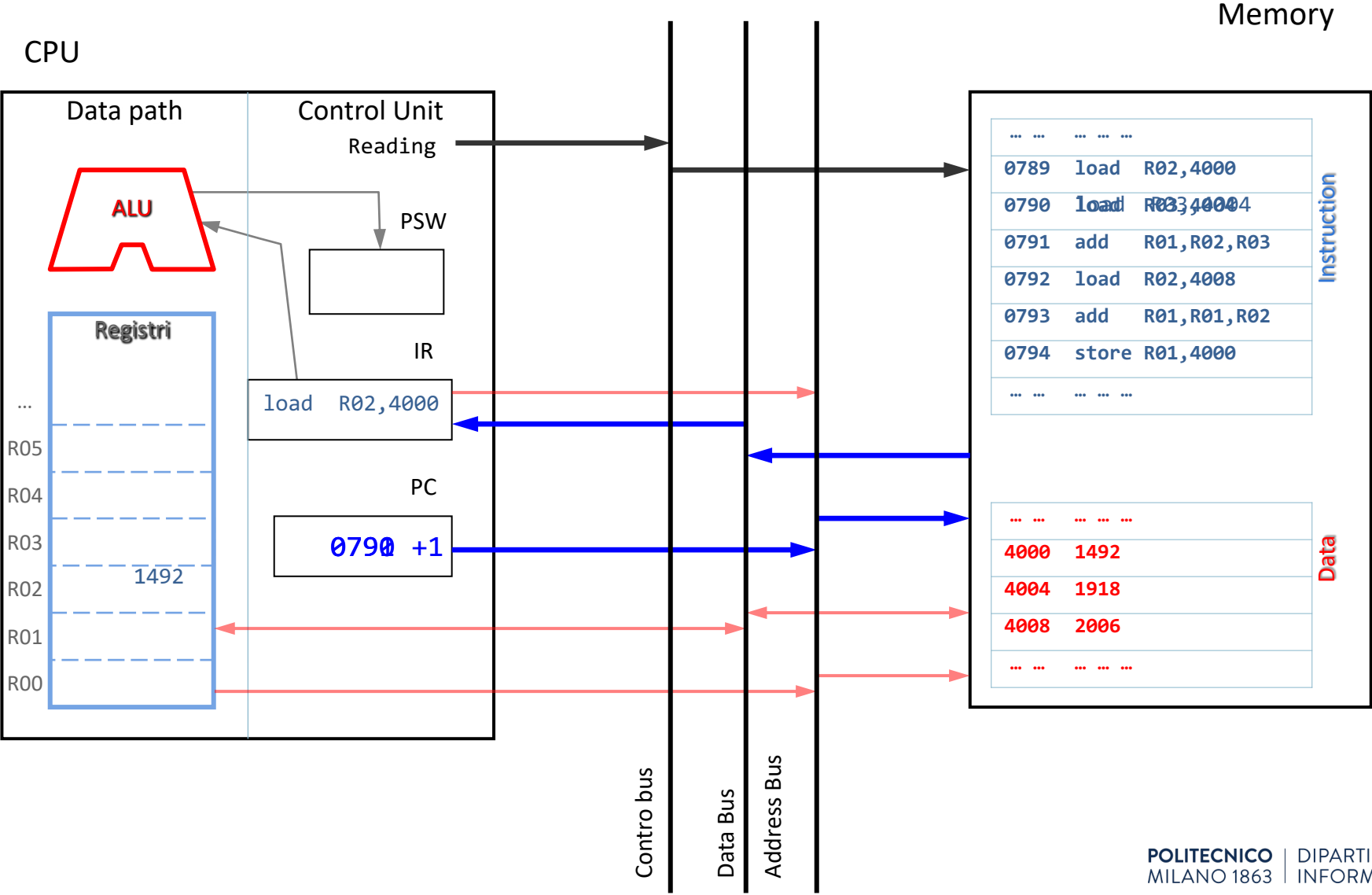
Read Instruction 0789



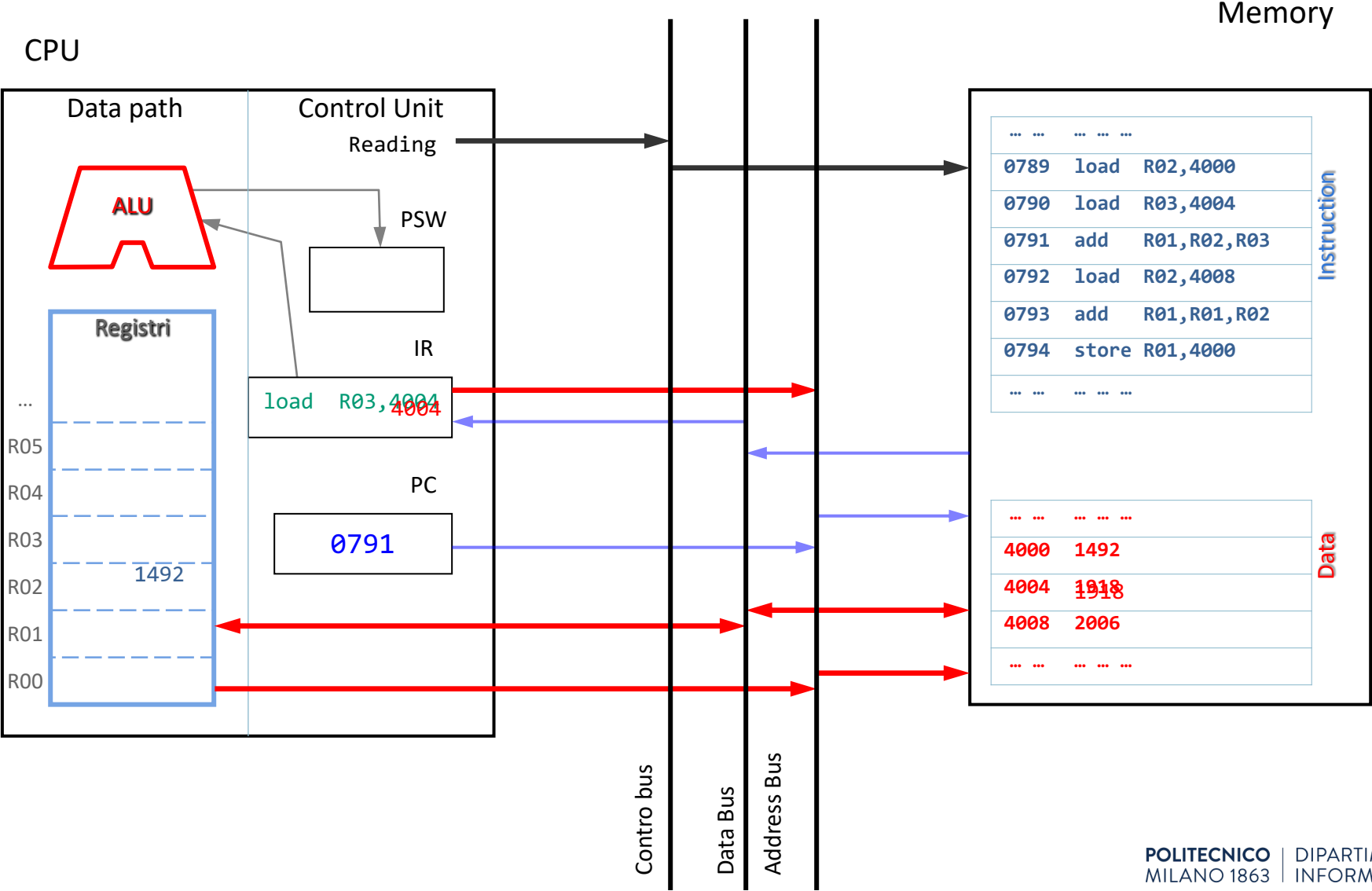
Exe Instruction 0789



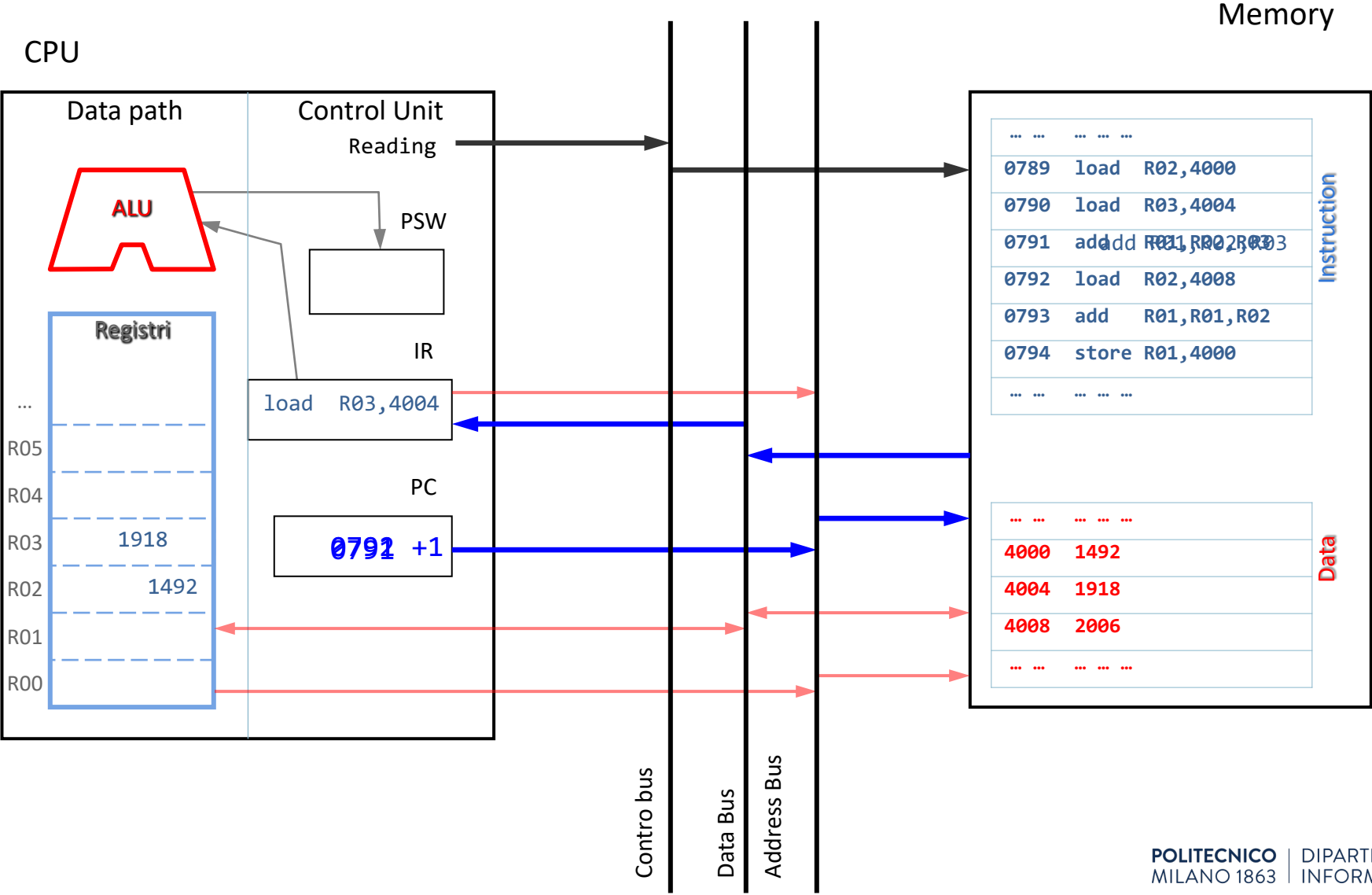
Read instruction 0790



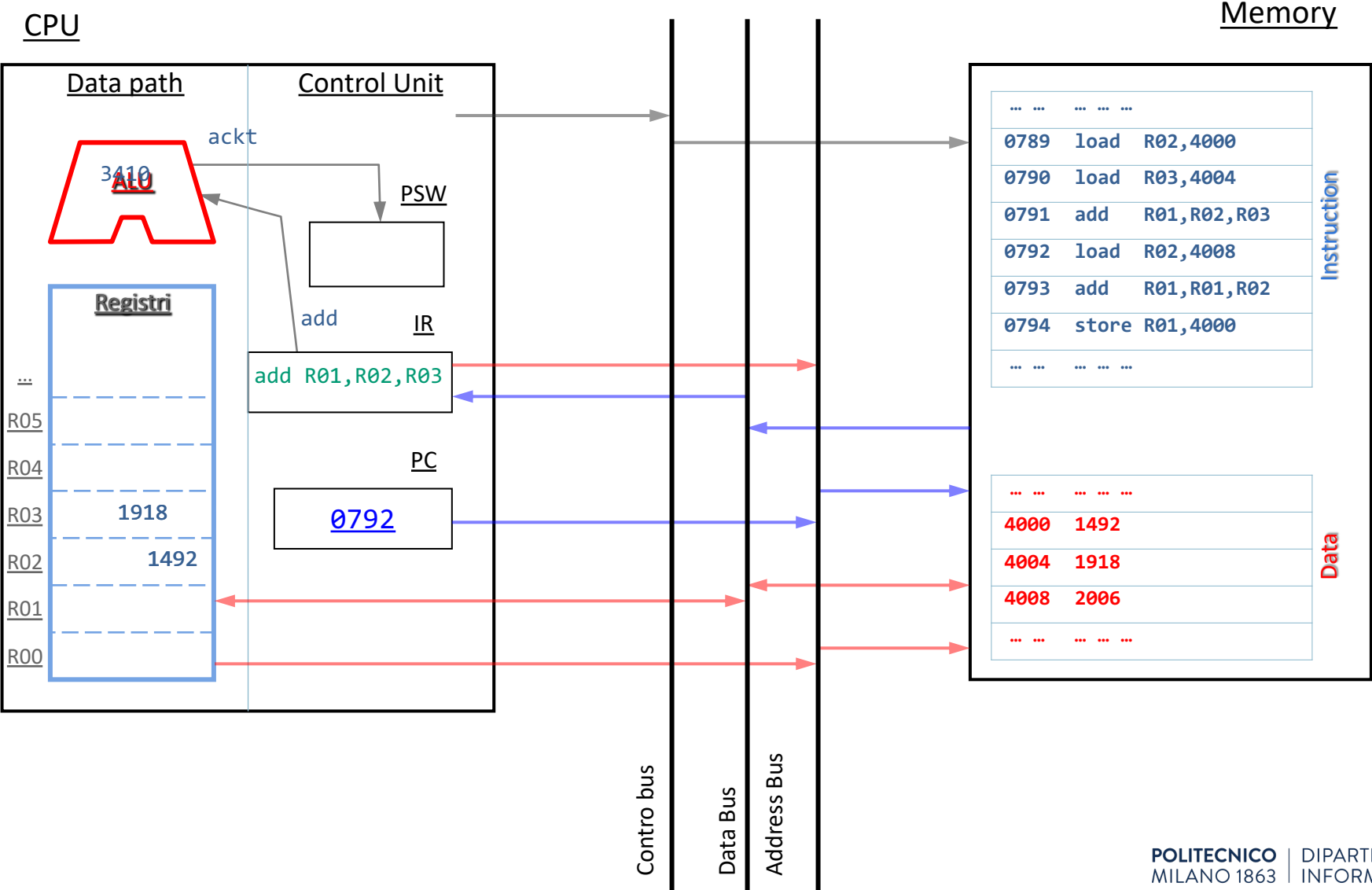
Exe Instruction 0790



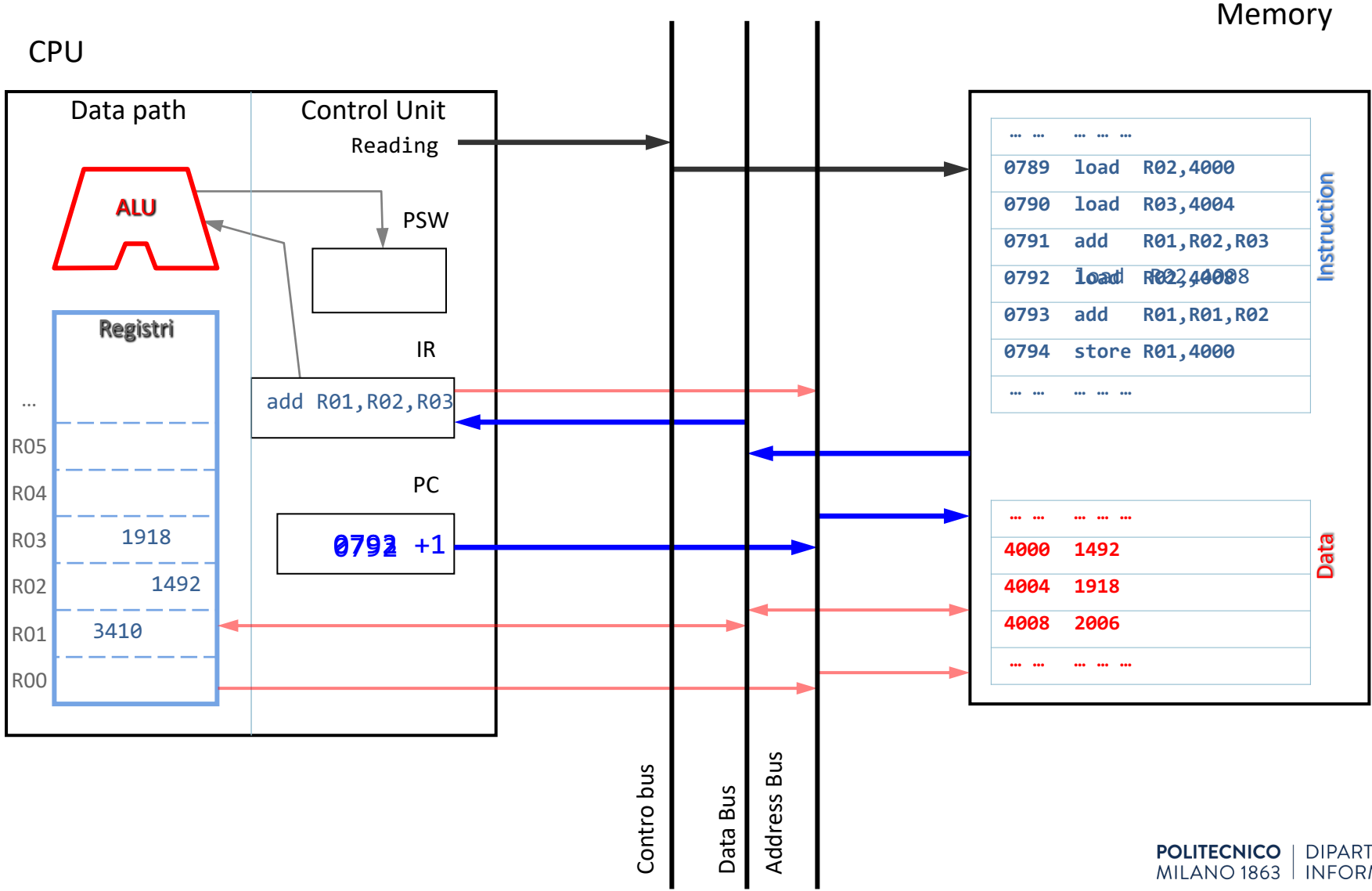
Read Instruction 0791



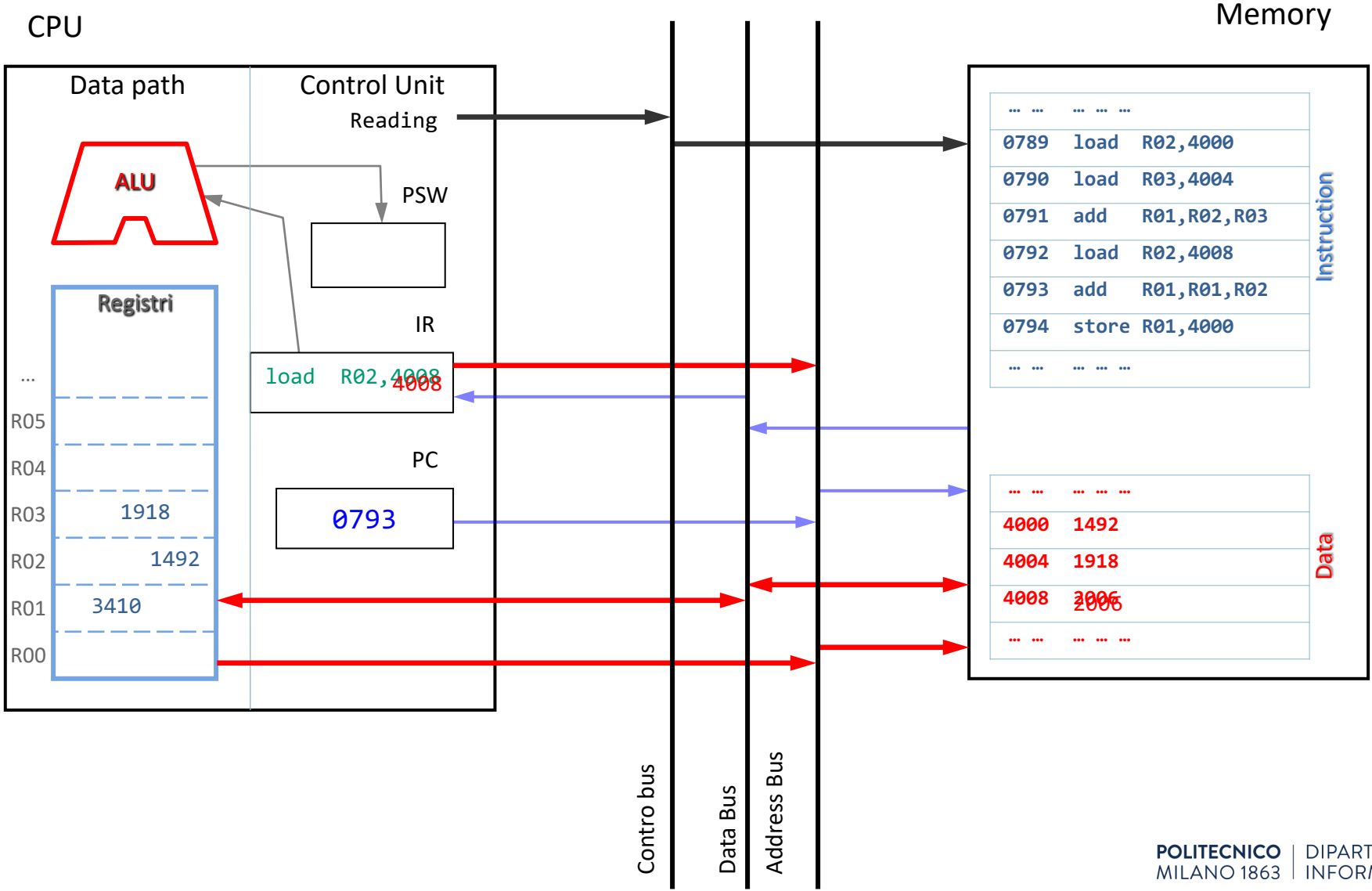
Exe Instruction 0791



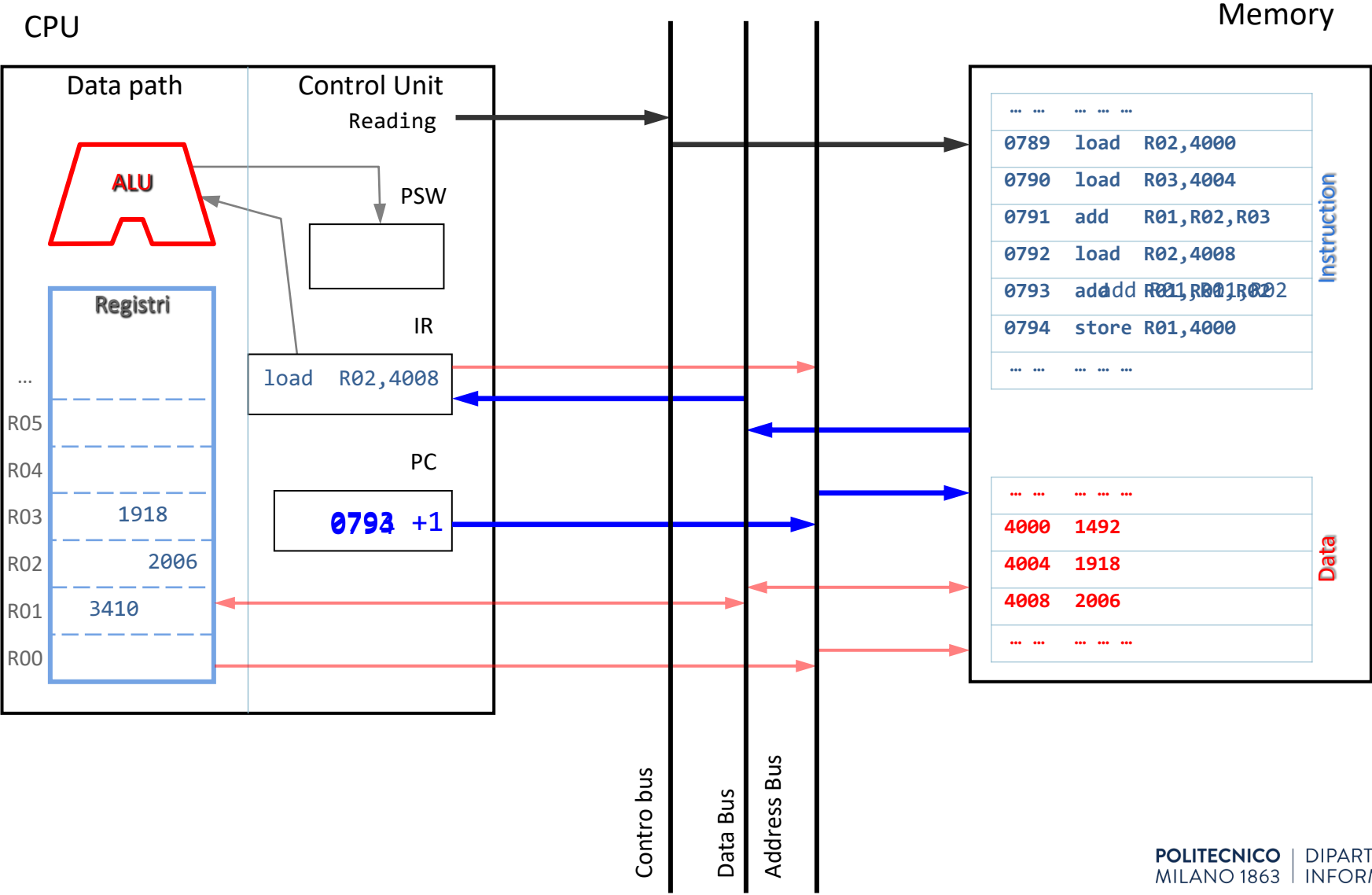
Read Instruction 0792



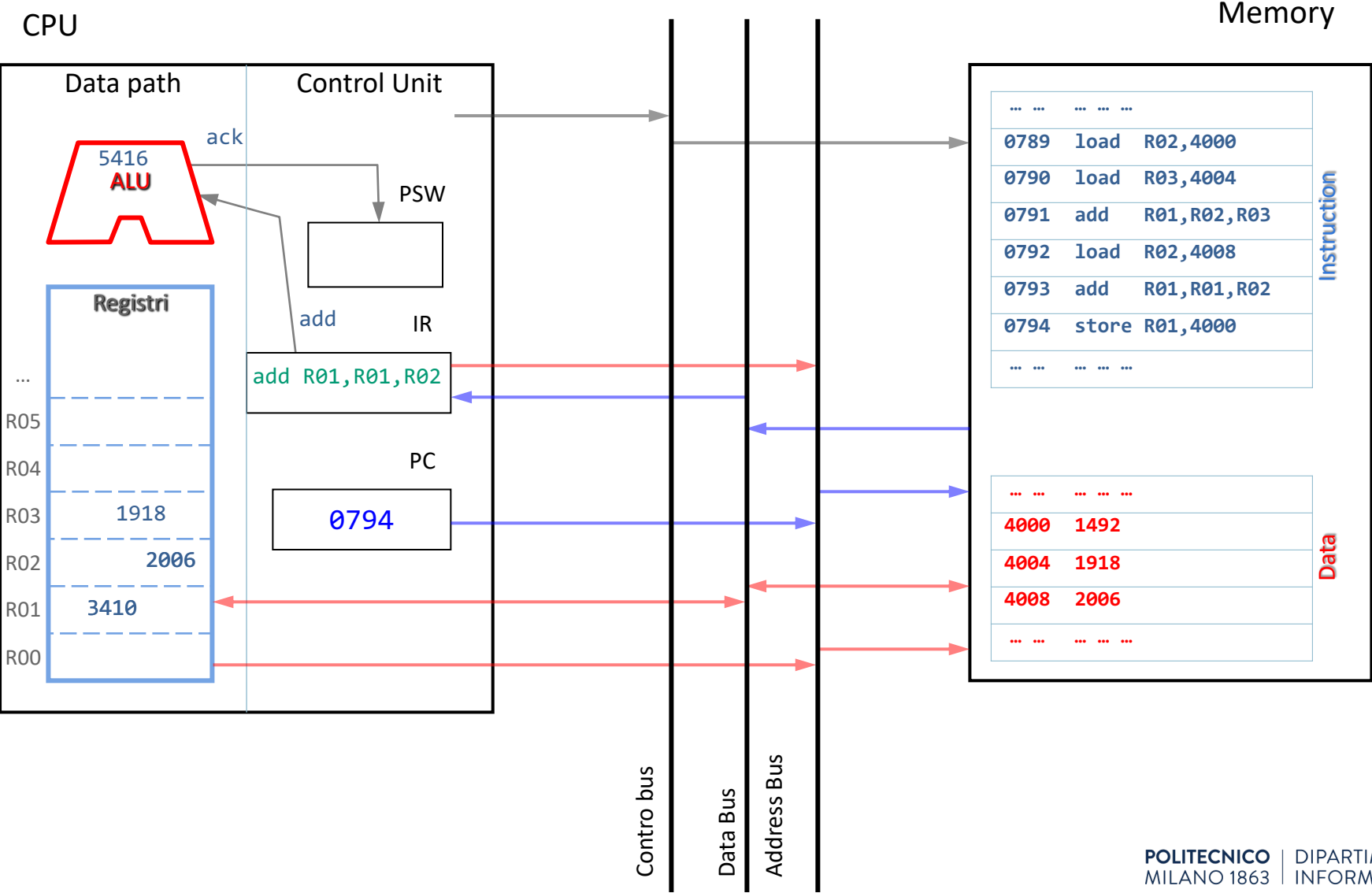
Exe Instruction 0792



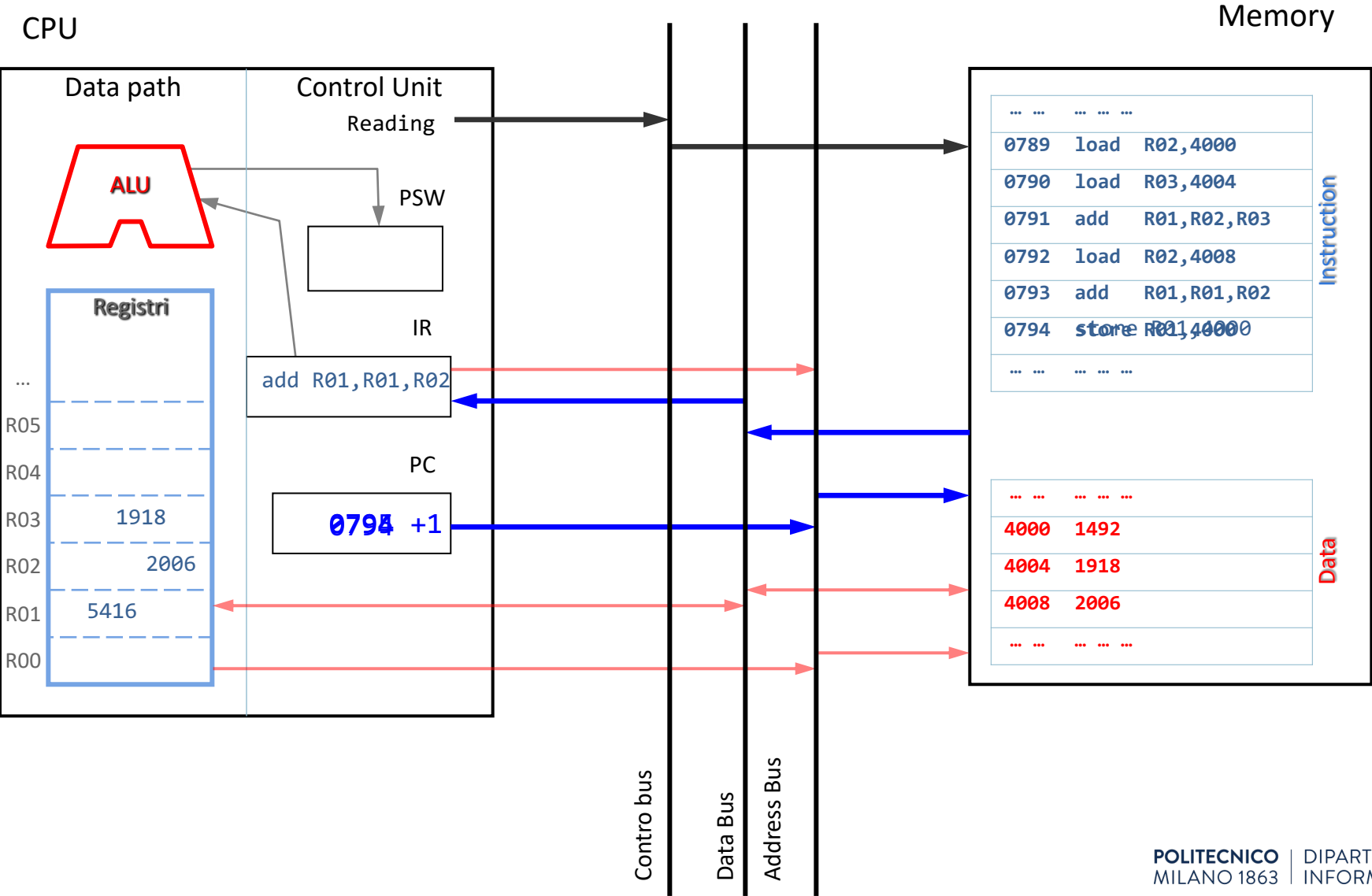
Read Instruction 0793



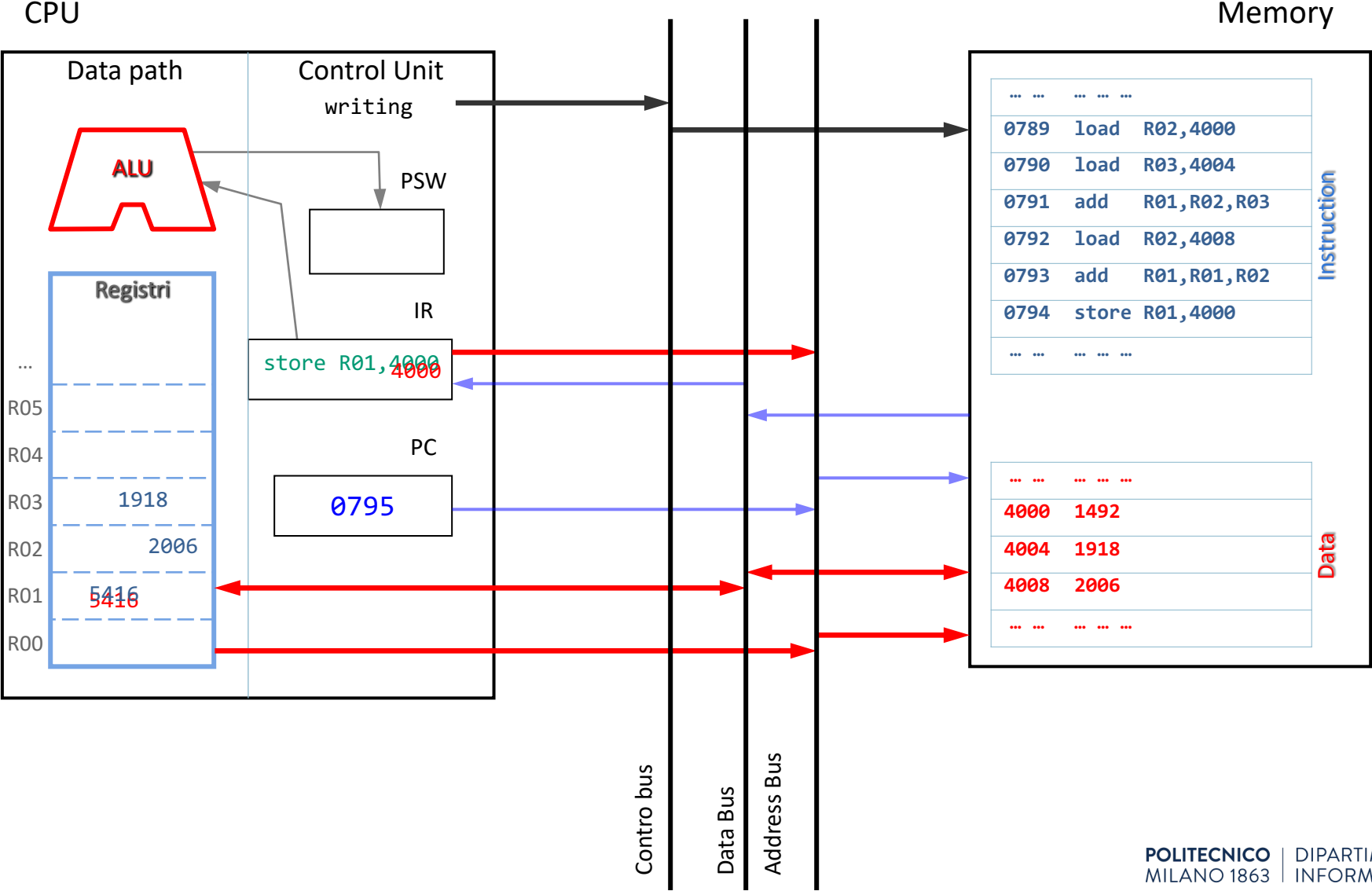
Exe Instruction 0793



Read Instruction 0794



Exe Instruction 0794





Questions?