POLITECNICO DI MILANO

Parallelism in wonderland: are you ready to see how deep the rabbit hole goes?

Pipelining

Ver. Jan 14, 2014

Marco D. Santambrogio: marco.santambrogio@polimi.it
Simone Campanoni: xan@eecs.harvard.edu

Outline

- Processors and Instruction Sets
- Review of pipelining
- MIPS
  - Reduced Instruction Set of MIPS™ Processor
  - Implementation of MIPS Processor Pipeline

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 they cannot directly come from the memory.
  Dedicated instructions are necessary to:
  - ***load*** data from memory to registers
  - ***store*** data from registers to memory
- **Pipeline Architecture:**

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

A Typical RISC ISA

- 32-bit fixed format instruction (3 formats)
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store:
  base + displacement
  - no indirection
- Simple branch conditions
- Delayed branch

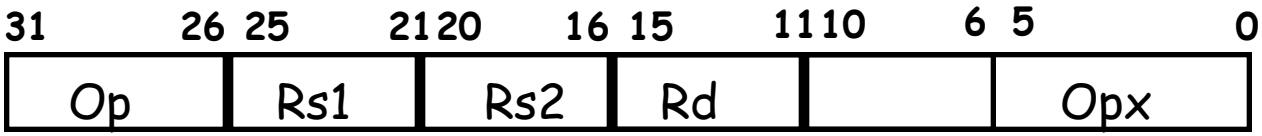
- Example: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3

Approaching an ISA

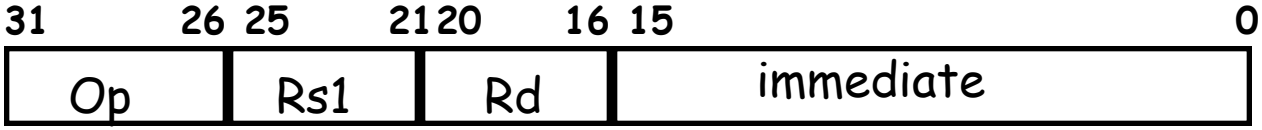
- Instruction Set Architecture
  - Defines set of operations, instruction format, hardware supported data types, named storage, addressing modes, sequencing
- Meaning of each instruction is described by RTL on *architected registers* and memory
- Given technology constraints assemble adequate datapath
  - Architected storage mapped to actual storage
  - Function units to do all the required operations
  - Possible additional storage (eg. MAR, MBR, …)
  - Interconnect to move information among regs and FUs
- Map each instruction to sequence of RTLs
- Collate sequences into symbolic controller state transition diagram (STD)
- Implement controller

Example: MIPS

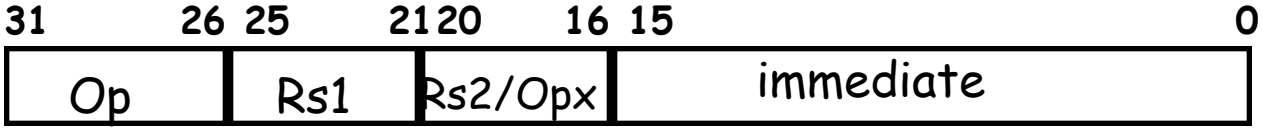
Register-Register



Register-Immediate

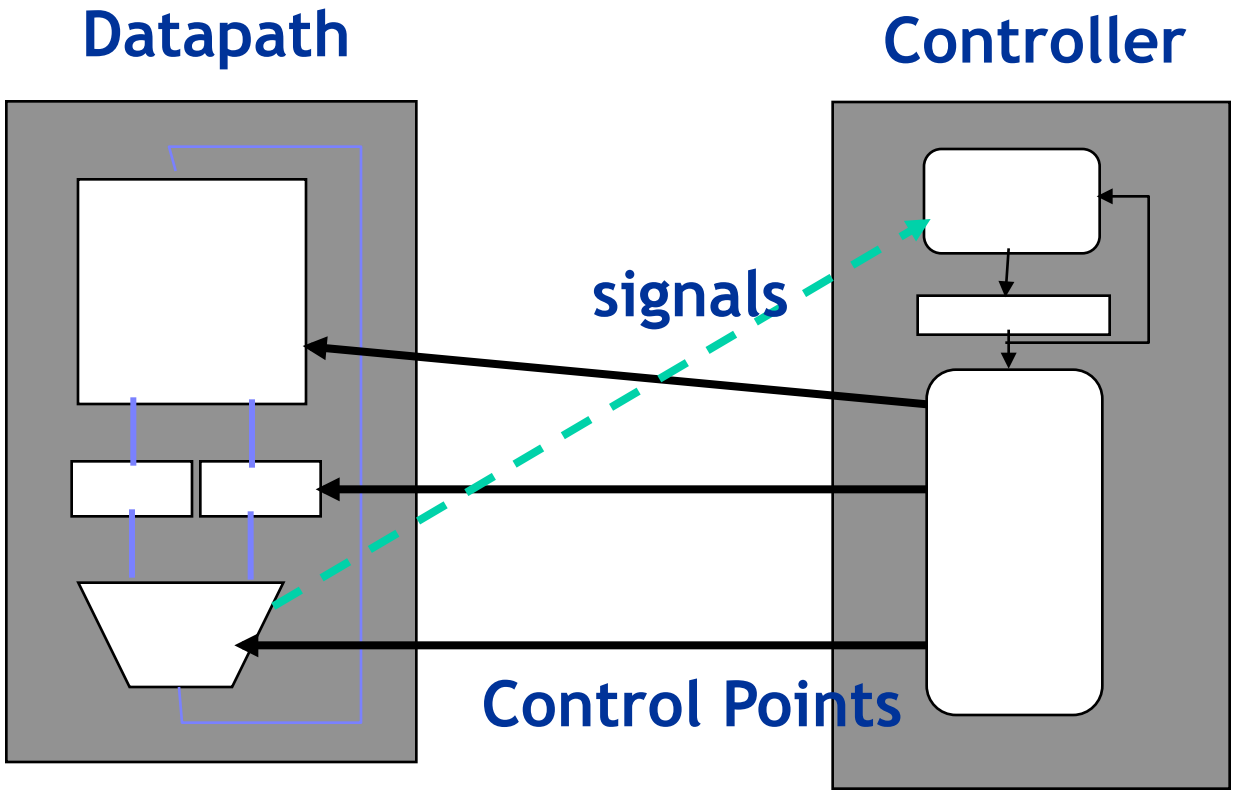


Branch



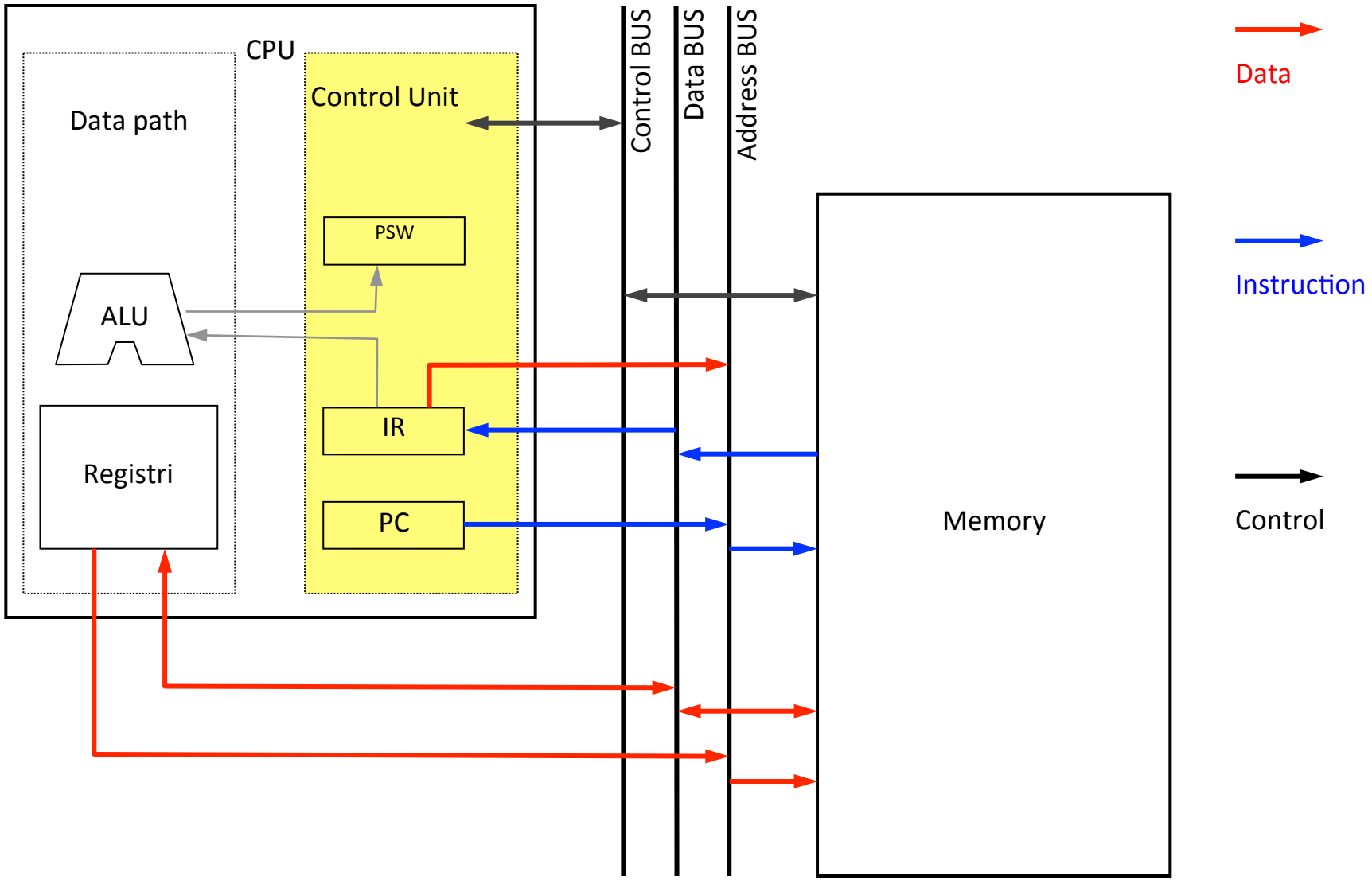
Jump / Call

Datapath vs Control

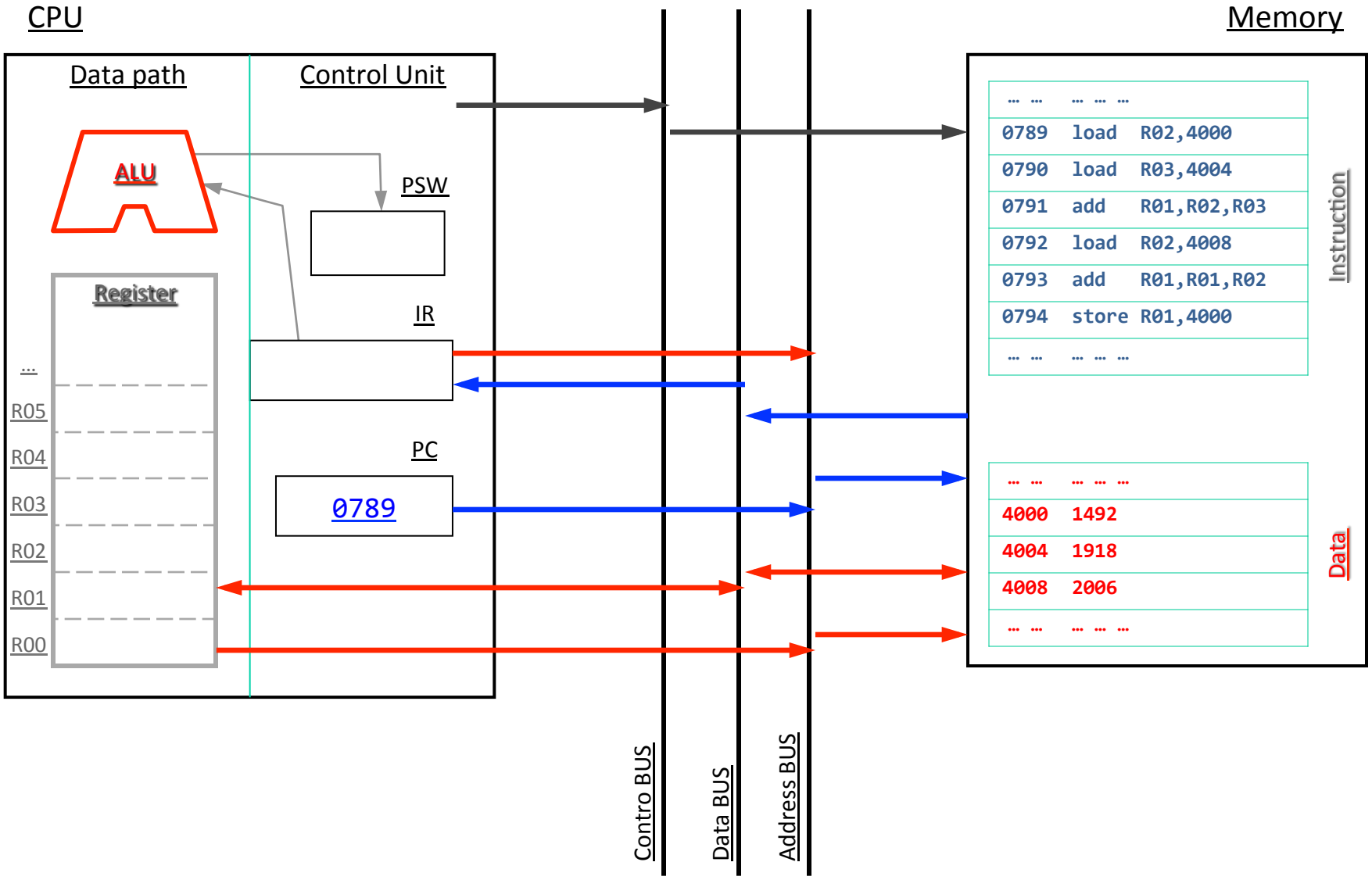


- Datapath: Storage, FU, interconnect sufficient to perform the desired functions
  - Inputs are Control Points
  - Outputs are signals
- Controller: State machine to orchestrate operation on the data path
  - Based on desired function and signals

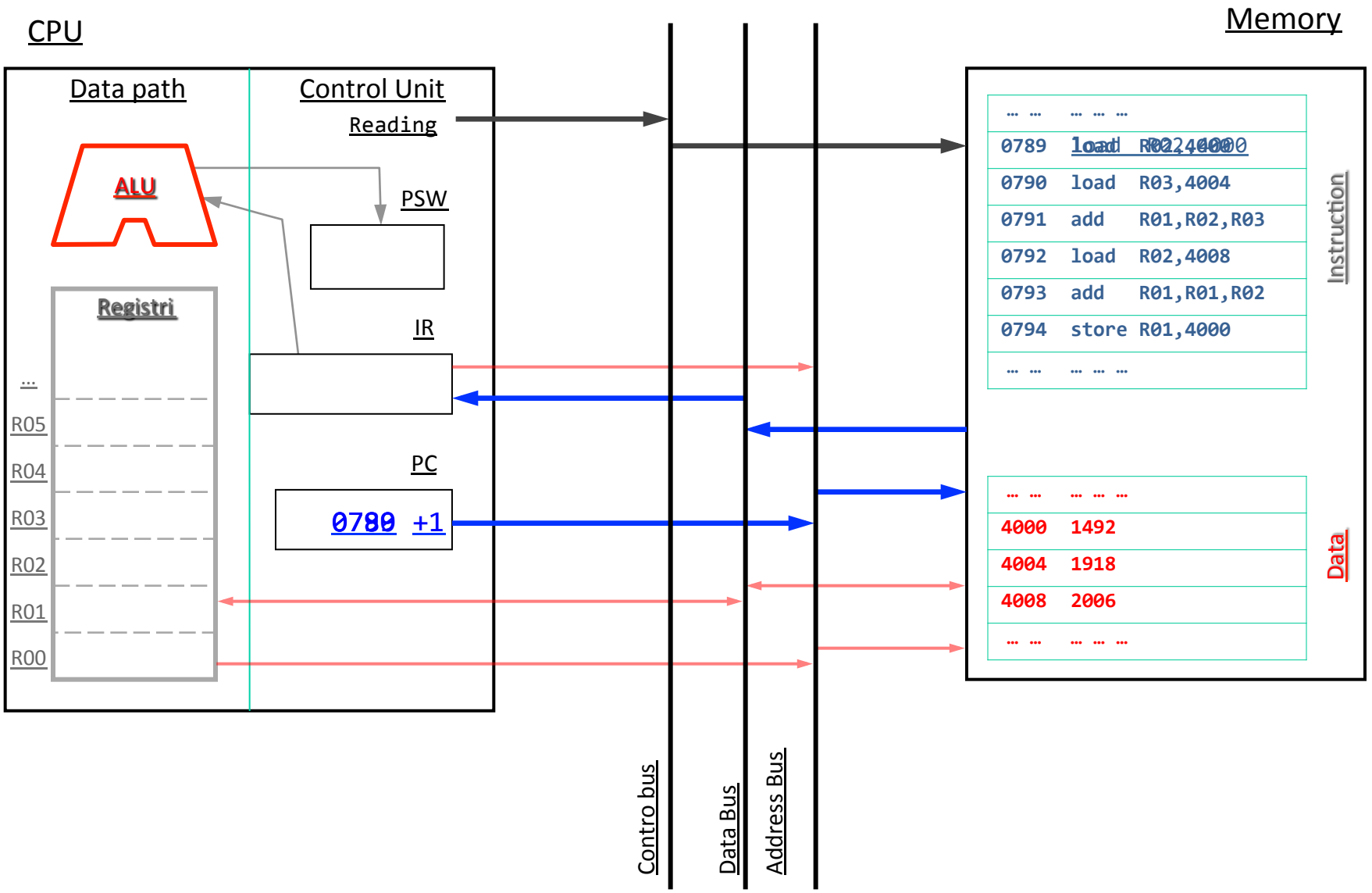
Datapath vs Control



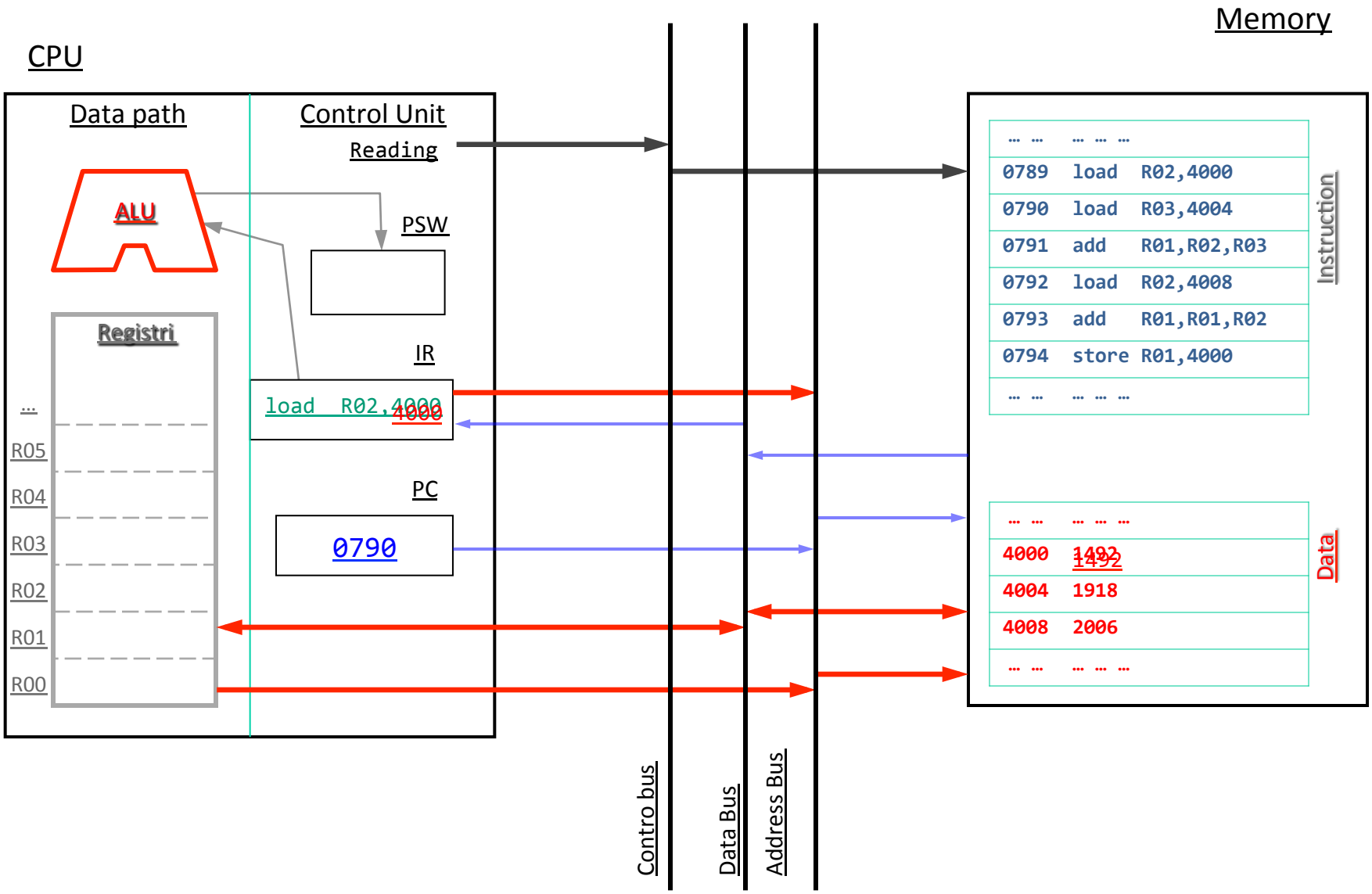
Starting scenario



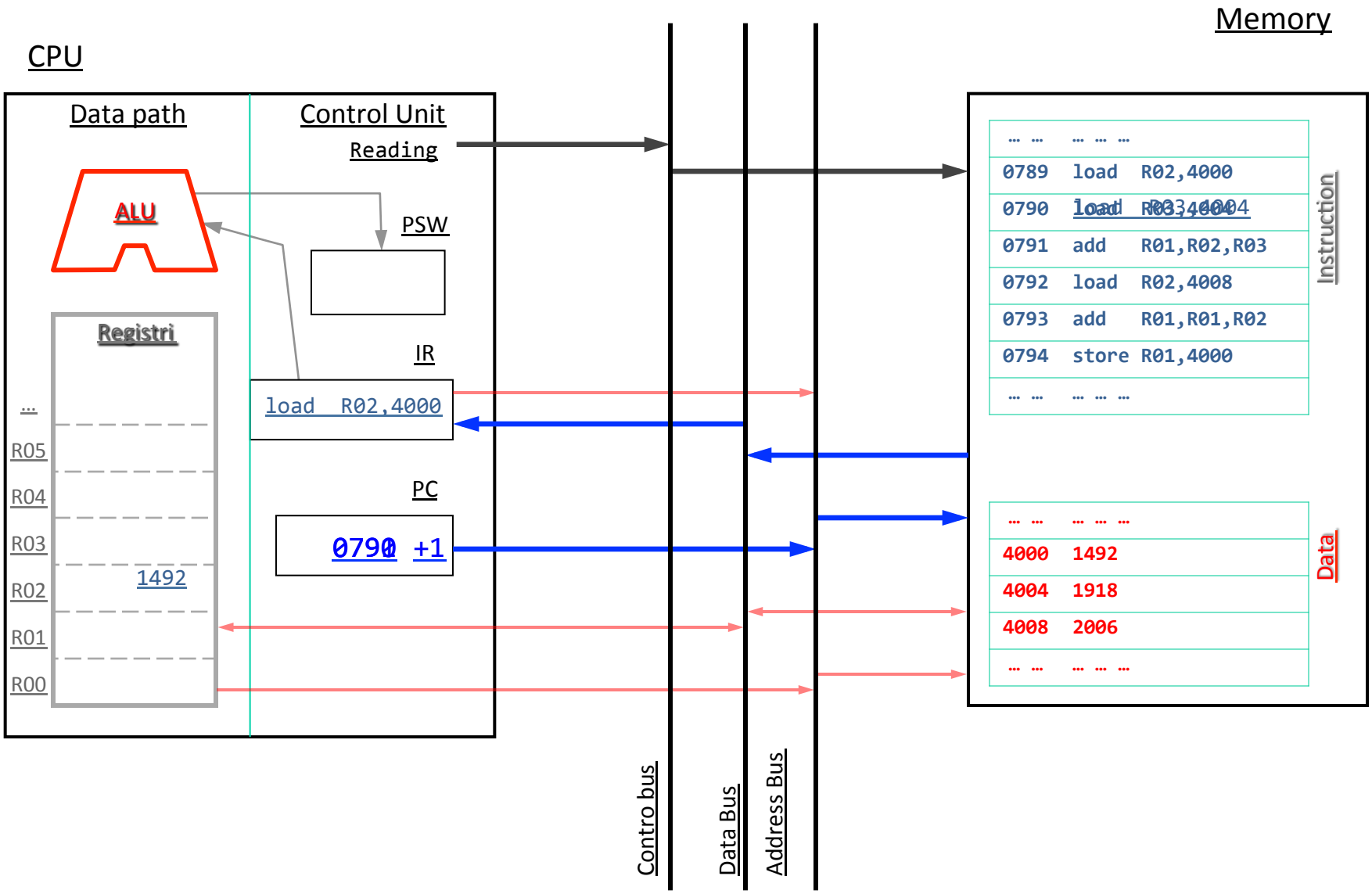
Read Instruction 0789



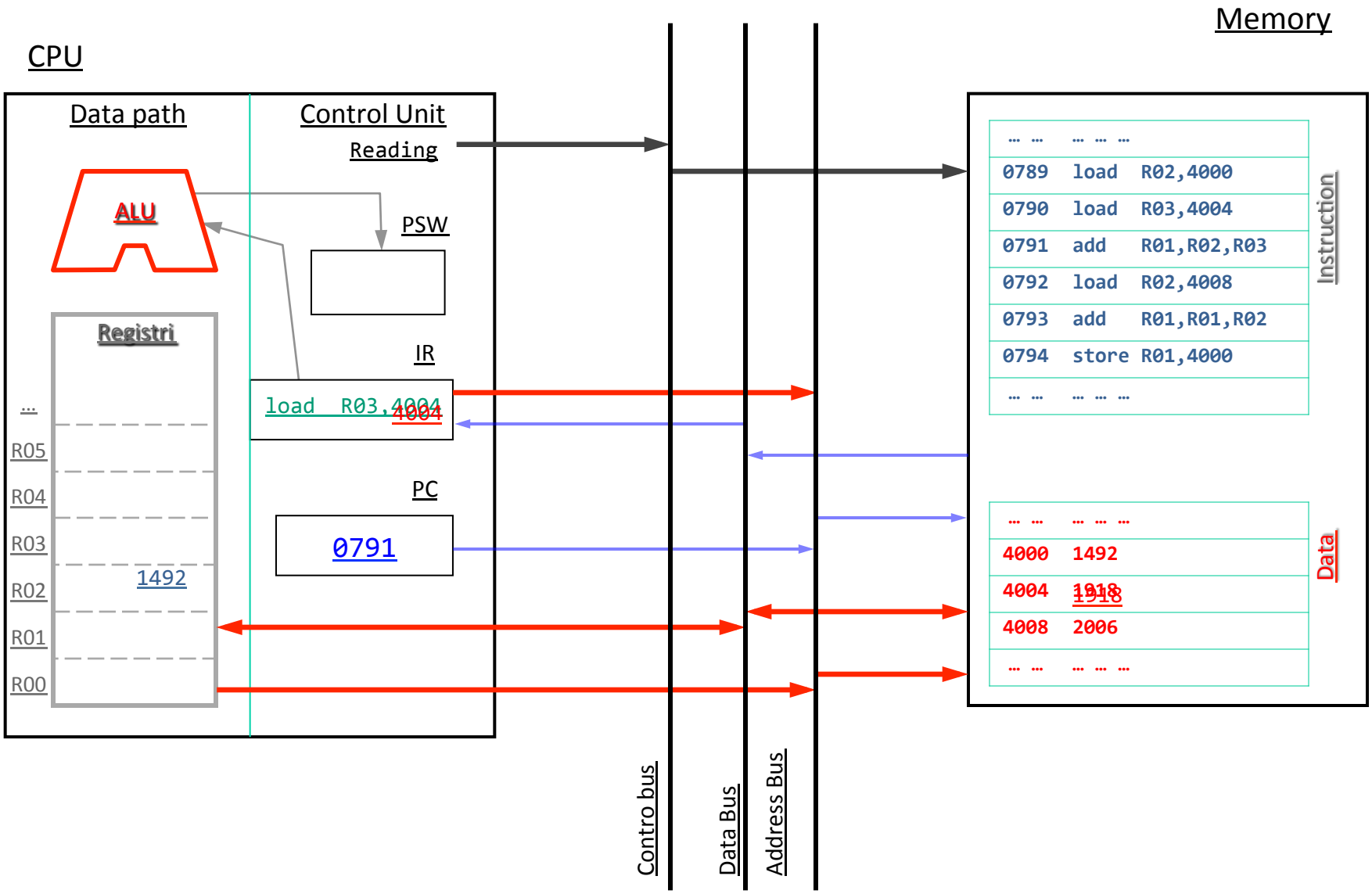
Exe Instruction 0789



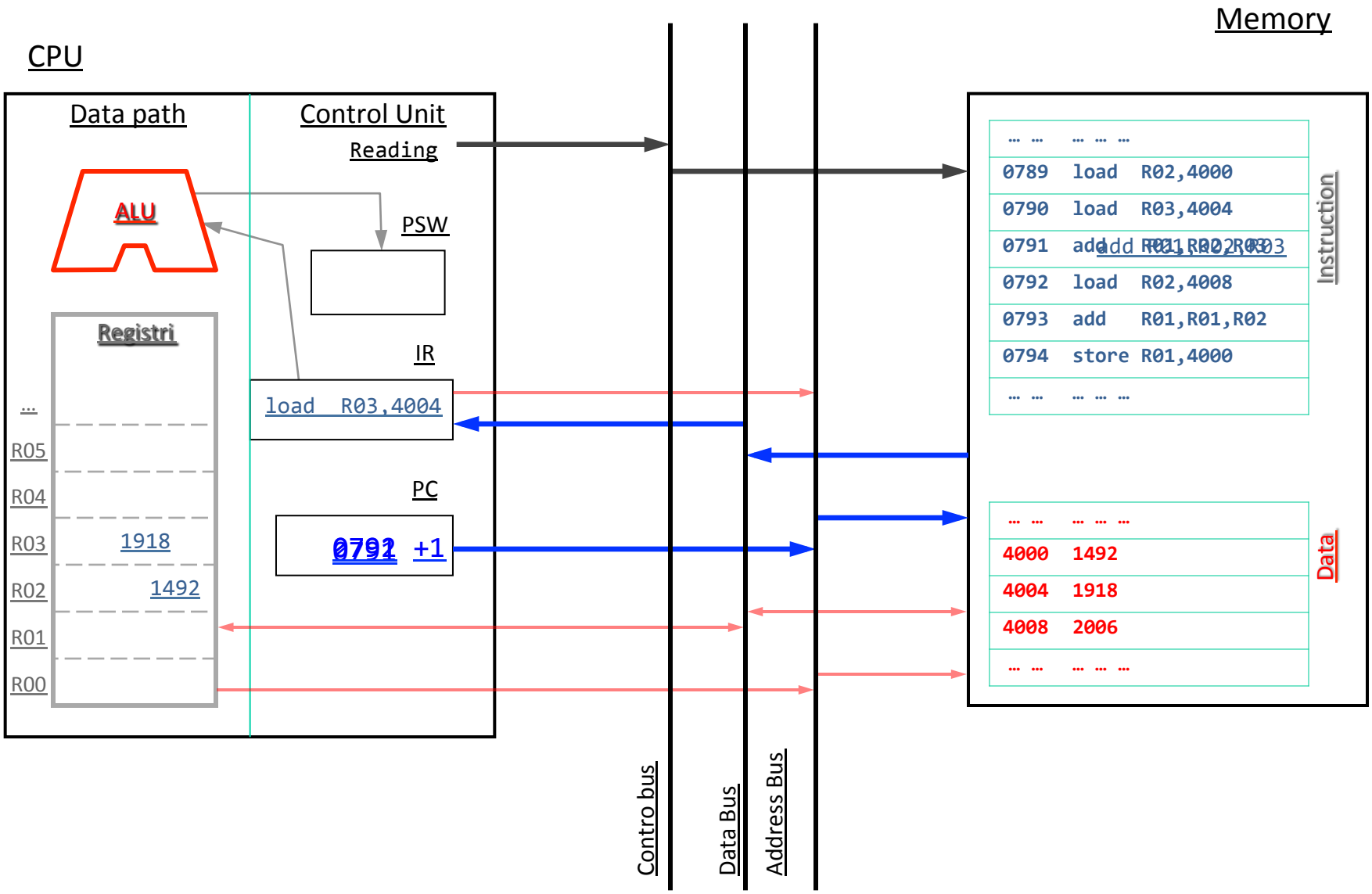
Read instruction 0790



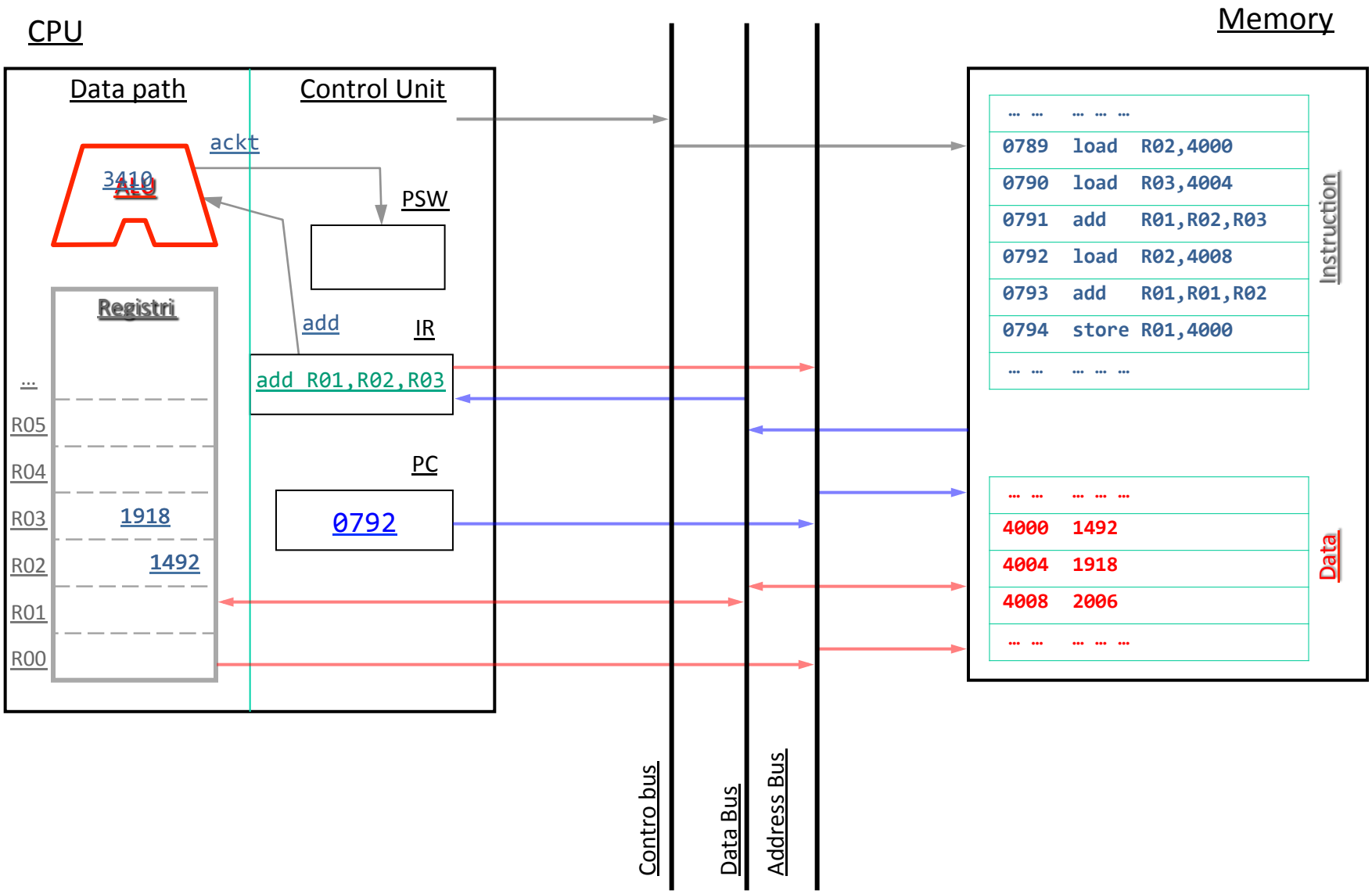
Exe Instruction 0790



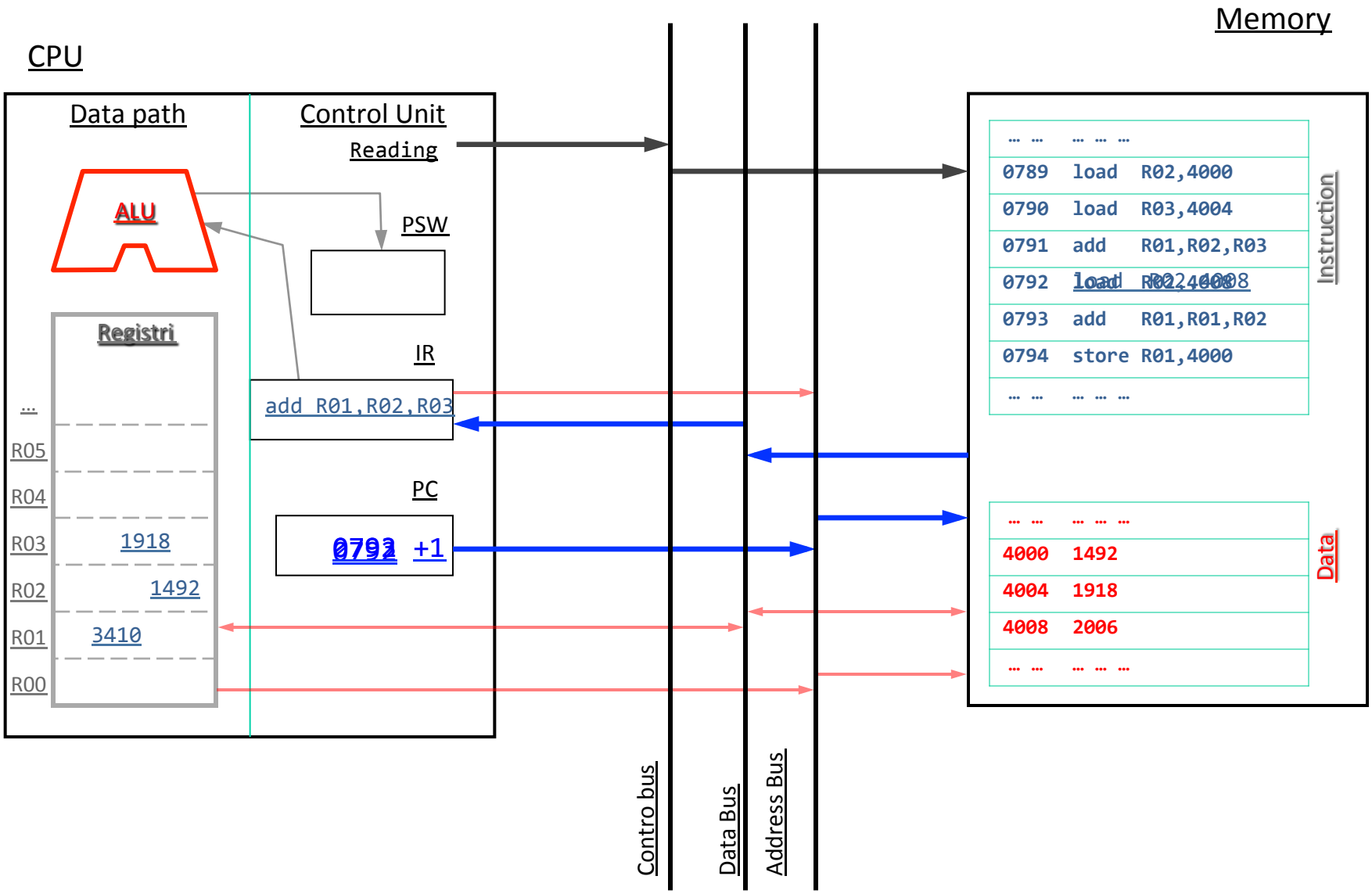
Read Instruction 0791



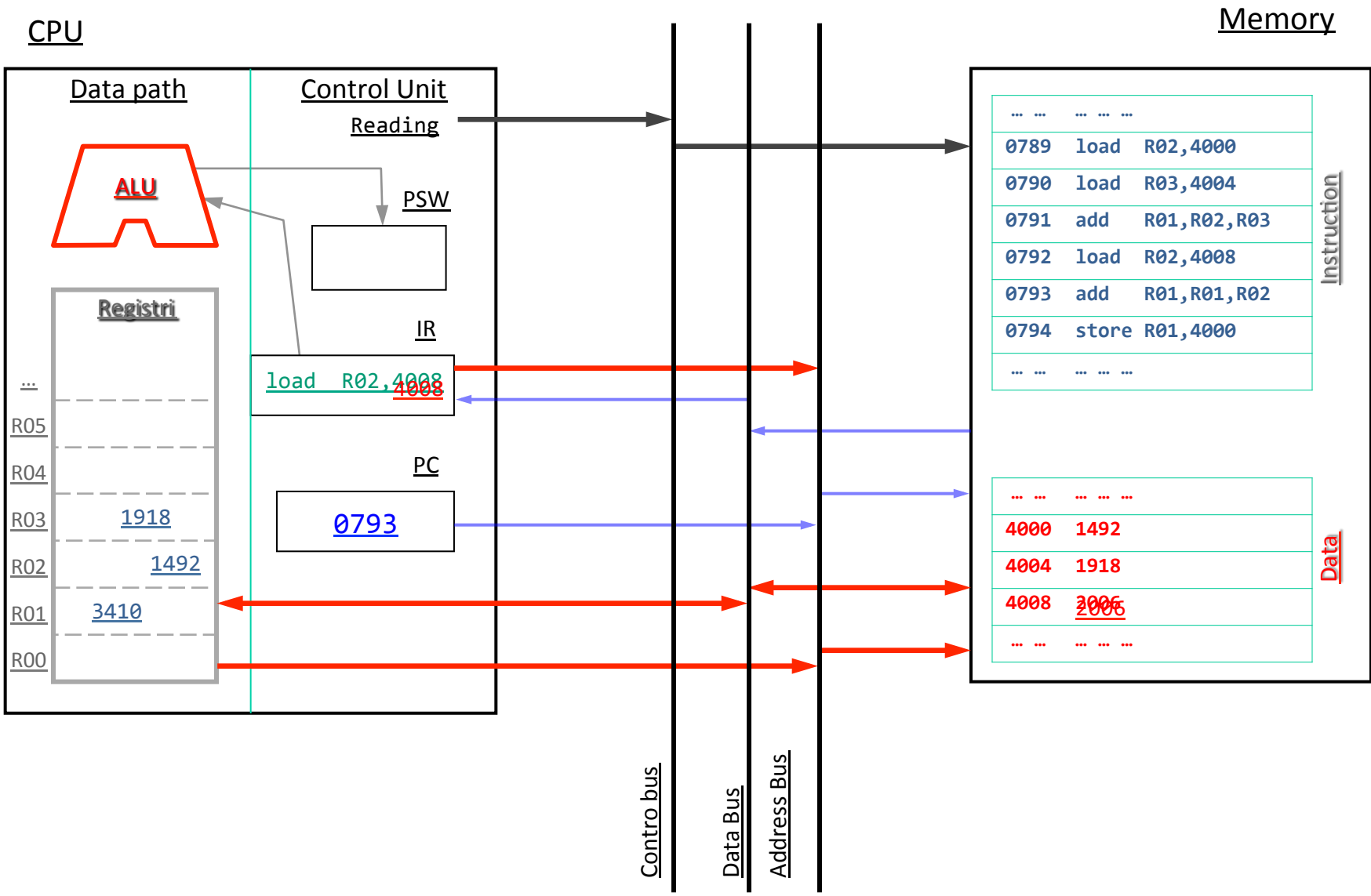
Exe Instruction 0791



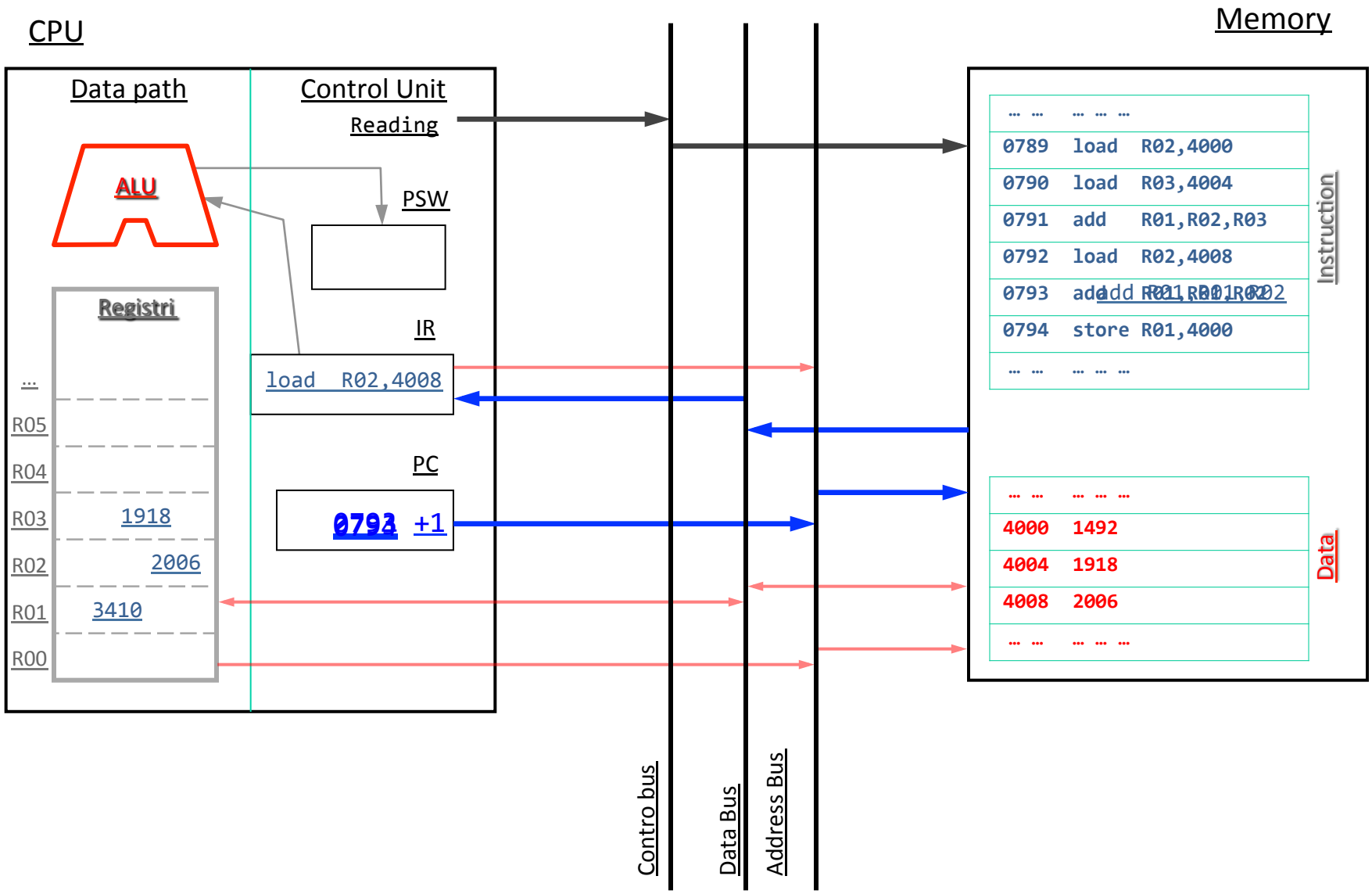
Read Instruction 0792



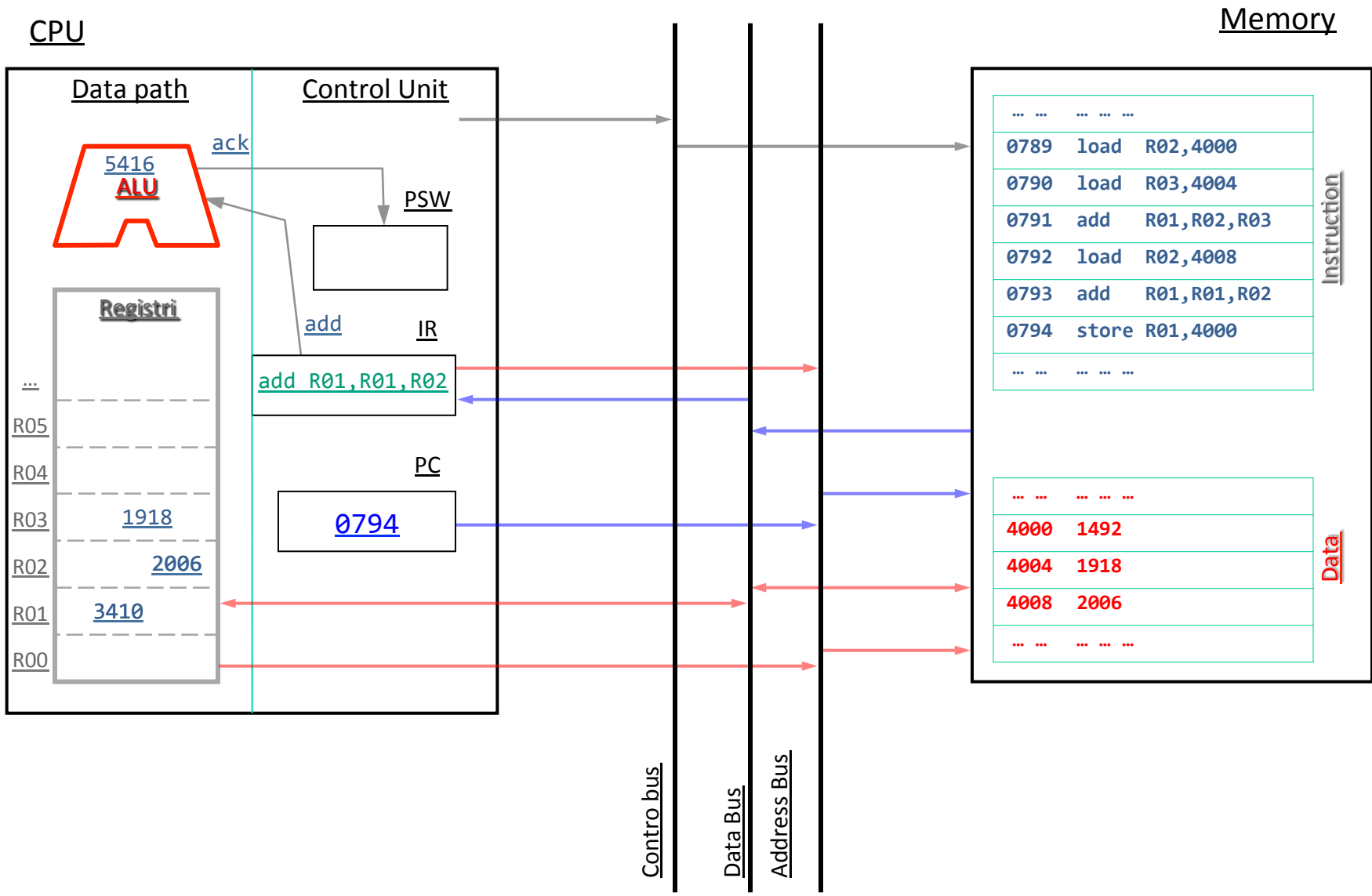
Exe Instruction 0792



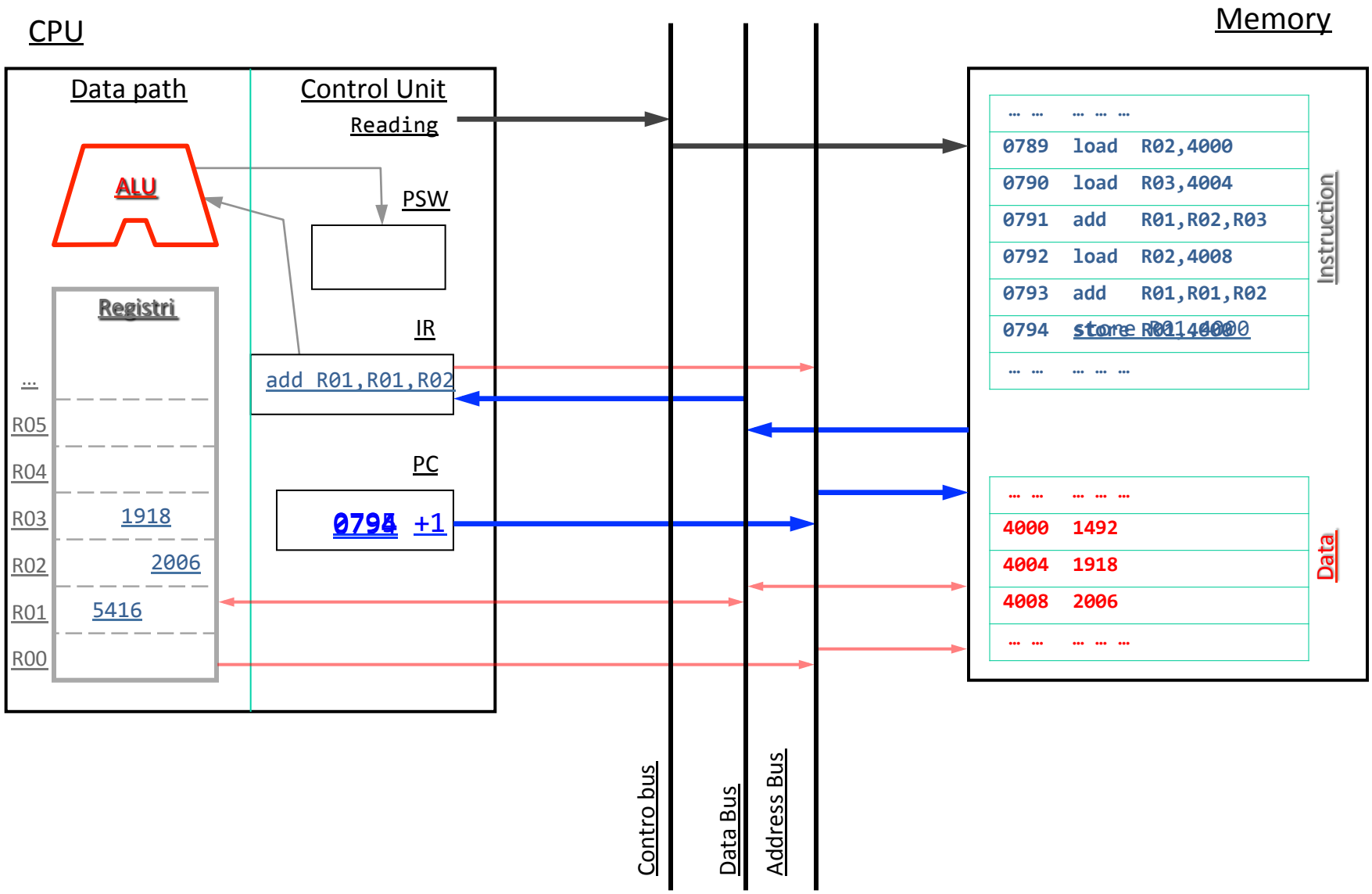
Read Instruction 0793



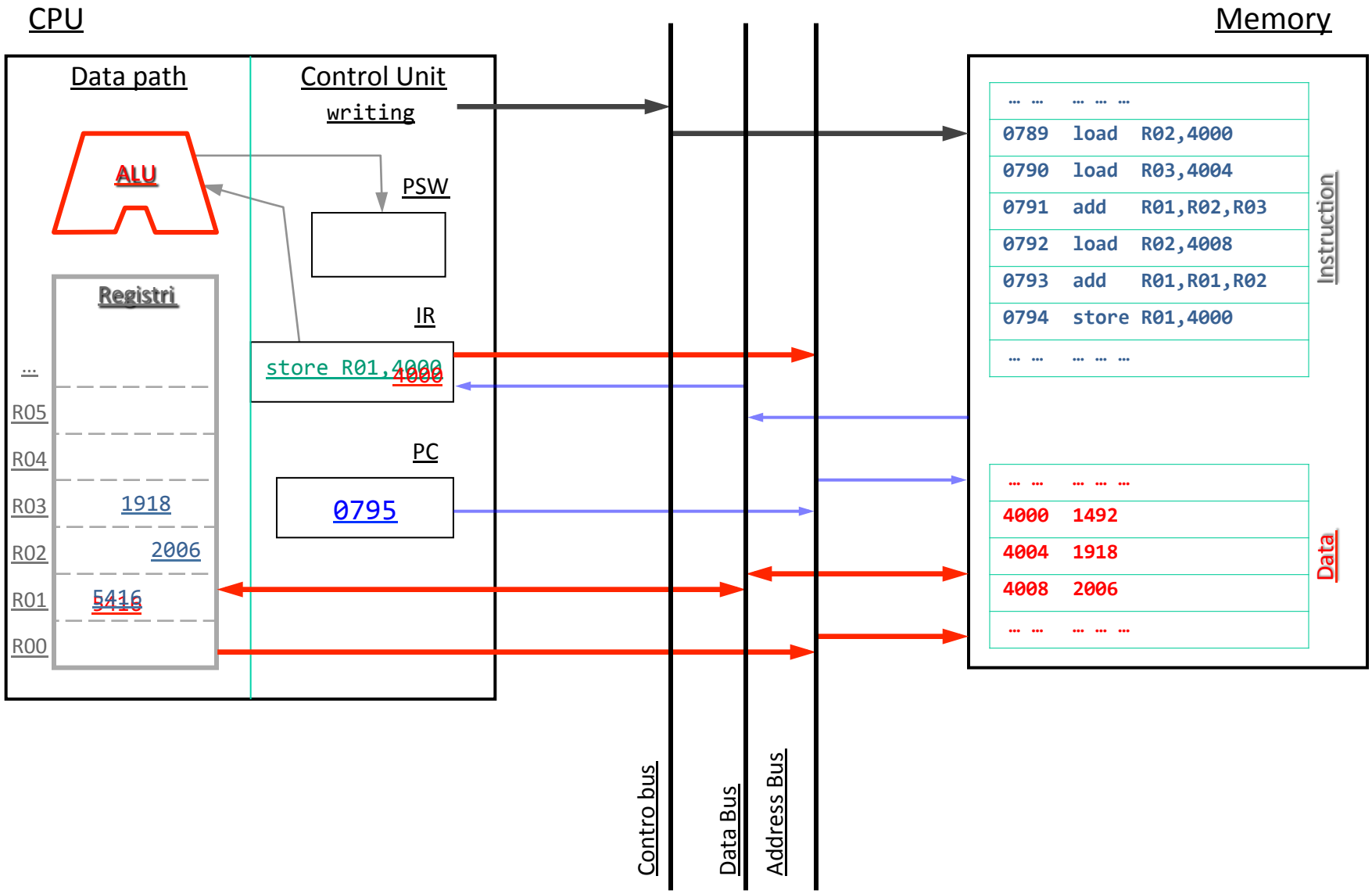
Exe Instruction 0793



Read Instruction 0794



Exe Instruction 0794



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

Every instruction in the MIPS subset can be implemented in **at most 5 clock cycles** as follows:

- **Instruction Fetch Cycle:**
  - 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).
- **Instruction Decode and Register Read Cycle**
  - 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

- **Execution Cycle**

  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

- **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 yielded true.
- **Write-Back Cycle (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.

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

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

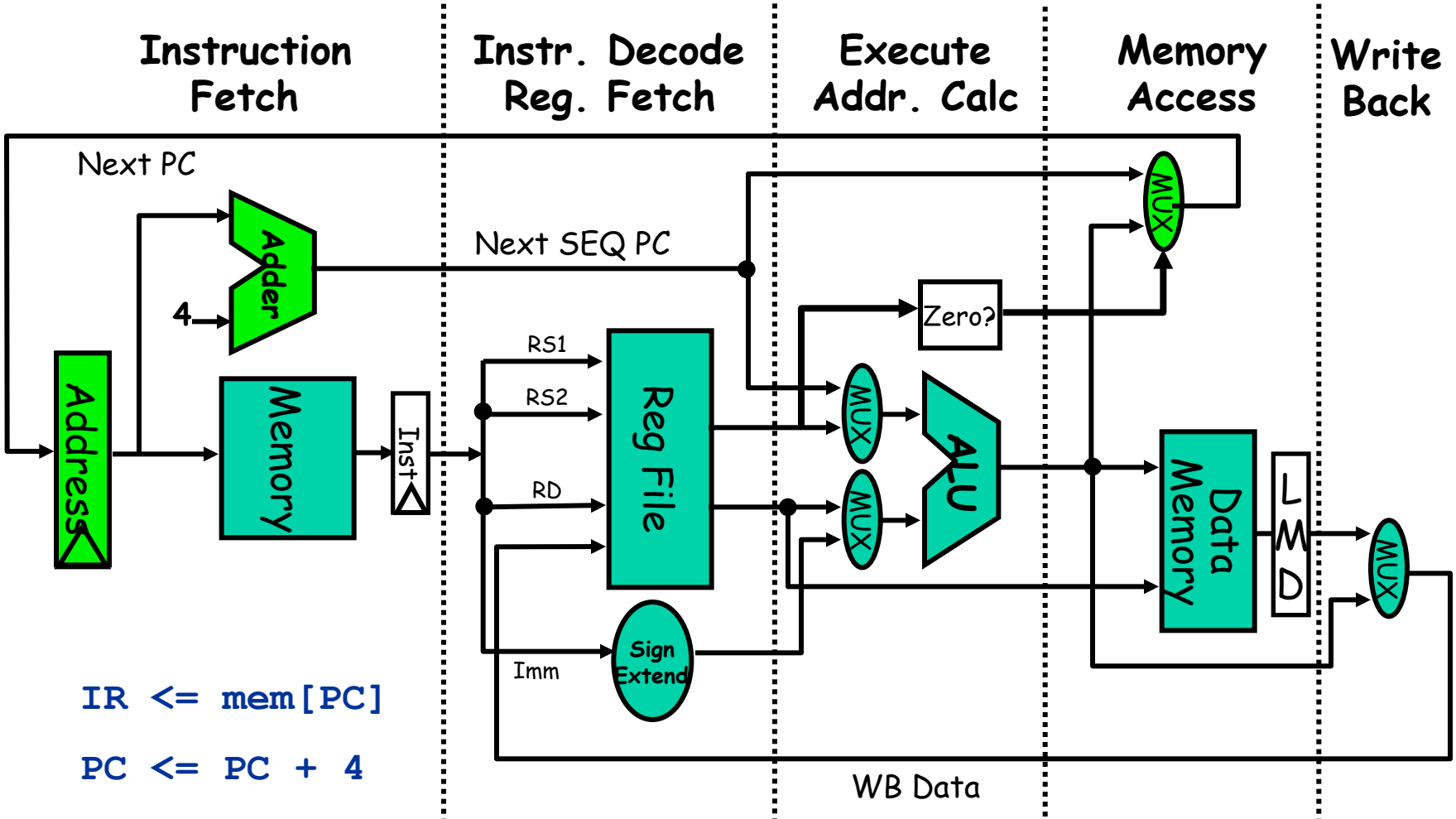
Store Instructions: `sw $x,offset($y)`

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

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

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

MIPS Data path



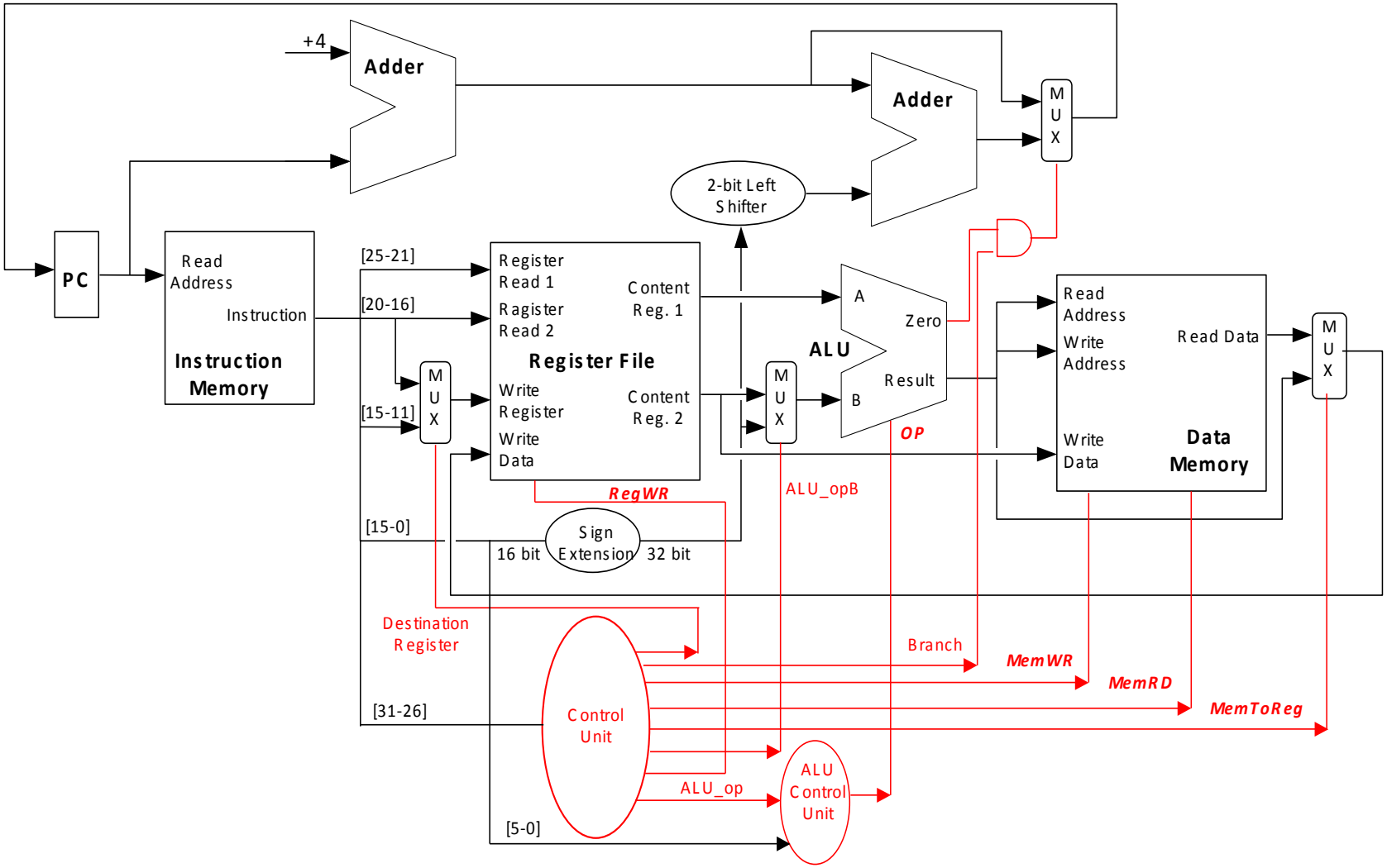
Instructions Latency

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 of MIPS

- The length of the clock cycle is defined by the critical path given by the load instruction: ***T = 8 ns (f = 125 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 different inputs based on the configuration of control lines.

Implementation of MIPS data path with Control Unit



Multi-cycle Implementation

- The instruction execution is distributed on multiple cycles (5 cycles for MIPS)
- The basic cycle is smaller
  (2 ns ⇒ 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.

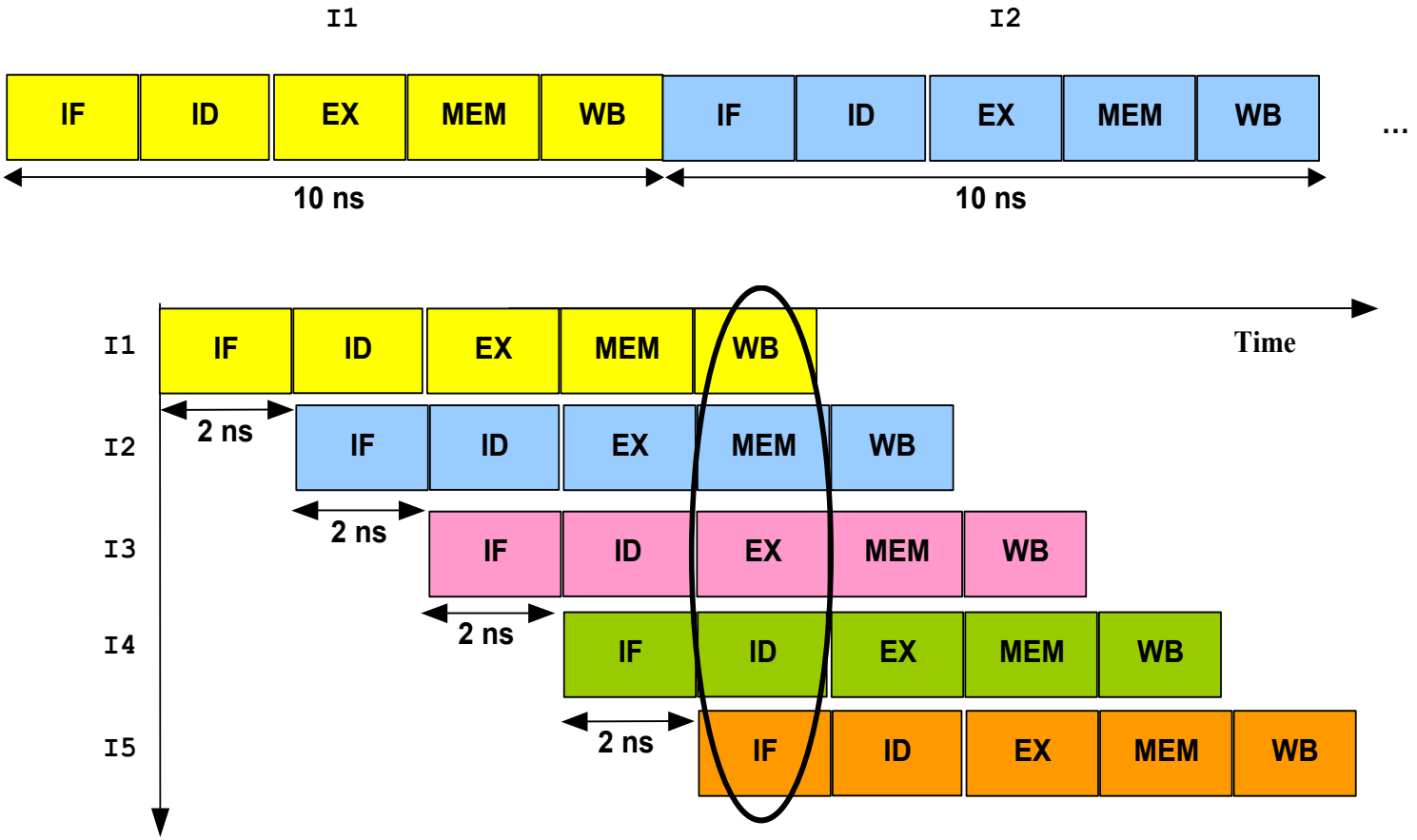
Pipelining

- 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 **(pipelines 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.

Pipelining

- Advantage: technique transparent for the programmer.
- Technique similar to a assembly line: a new car exits from the assembly line in the time necessary to complete one of the phases.
- An assembly line does not reduce the time necessary to complete a car, but increases the number of cars produced simultaneously and the frequency to complete cars.

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 *8 ns* and the pipelined *CPU2* with 5 stages of 2 ns :
  - The **latency** (total execution time) of each instruction is worsened: from *8 ns* to *10 ns*
  - The **throughput** (number of instructions completed in the time unit) is improved of **4 times:**
    (1 instruction completed each *8 ns)* vs.
    (1 instruction completed each *2 ns)*

Performance Improvement

- Ideal case (asymptotically): If we consider the multi-cycle unpipelined *CPU3* composed of 5 cycles of *2 ns* and the pipelined *CPU2* with 5 stages of 2 ns :
  - The **latency** (total execution time) of each instruction is not varied *(10 ns)*
  - The **throughput** (number of instructions completed in the time unit) is improved of ***5* times:**
    (1 instruction completed every *10 ns) vs.*
    (1 instruction completed every *2 ns)*

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`

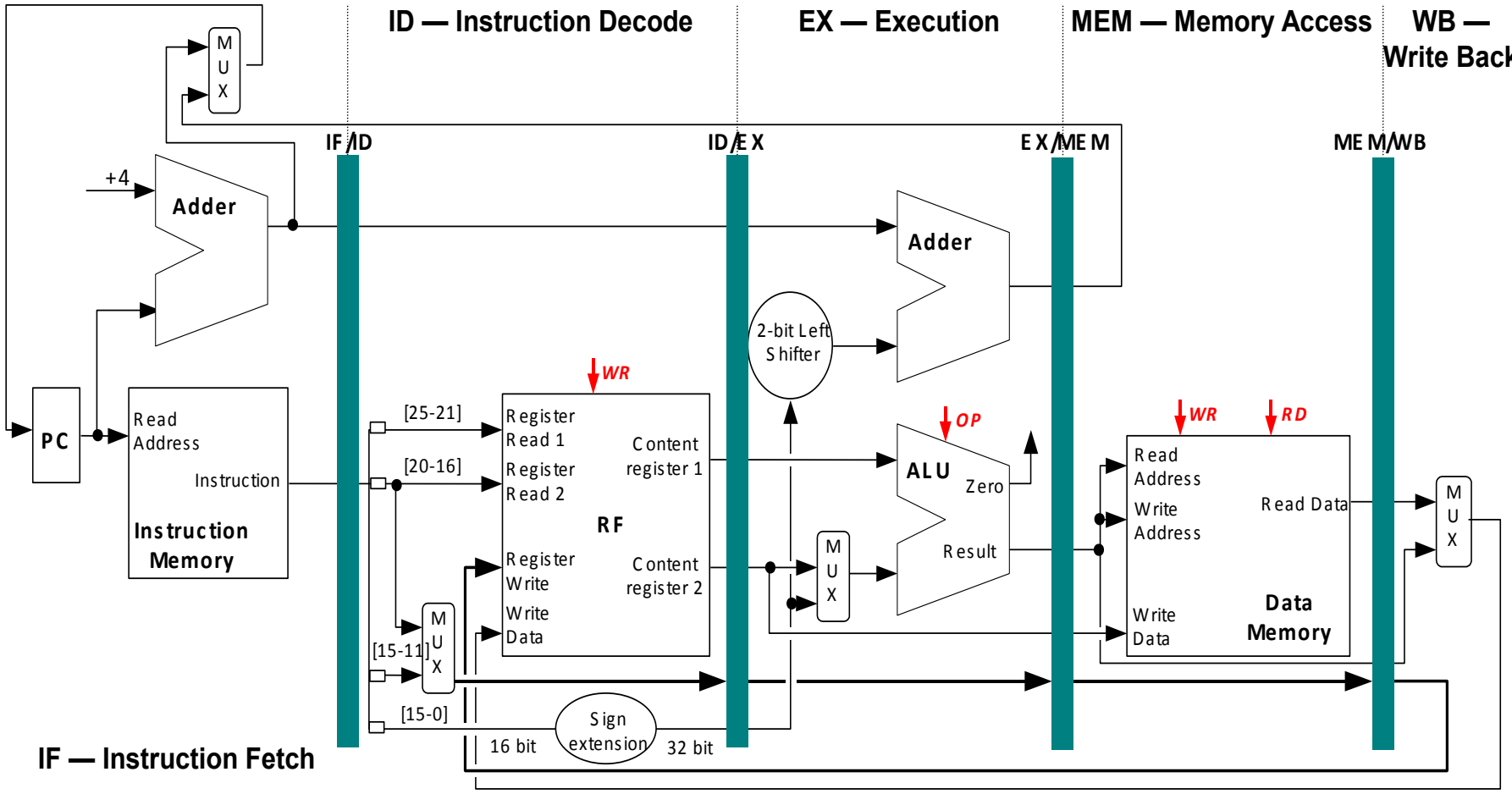
Store Instructions: `sw $x,offset($y)`

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

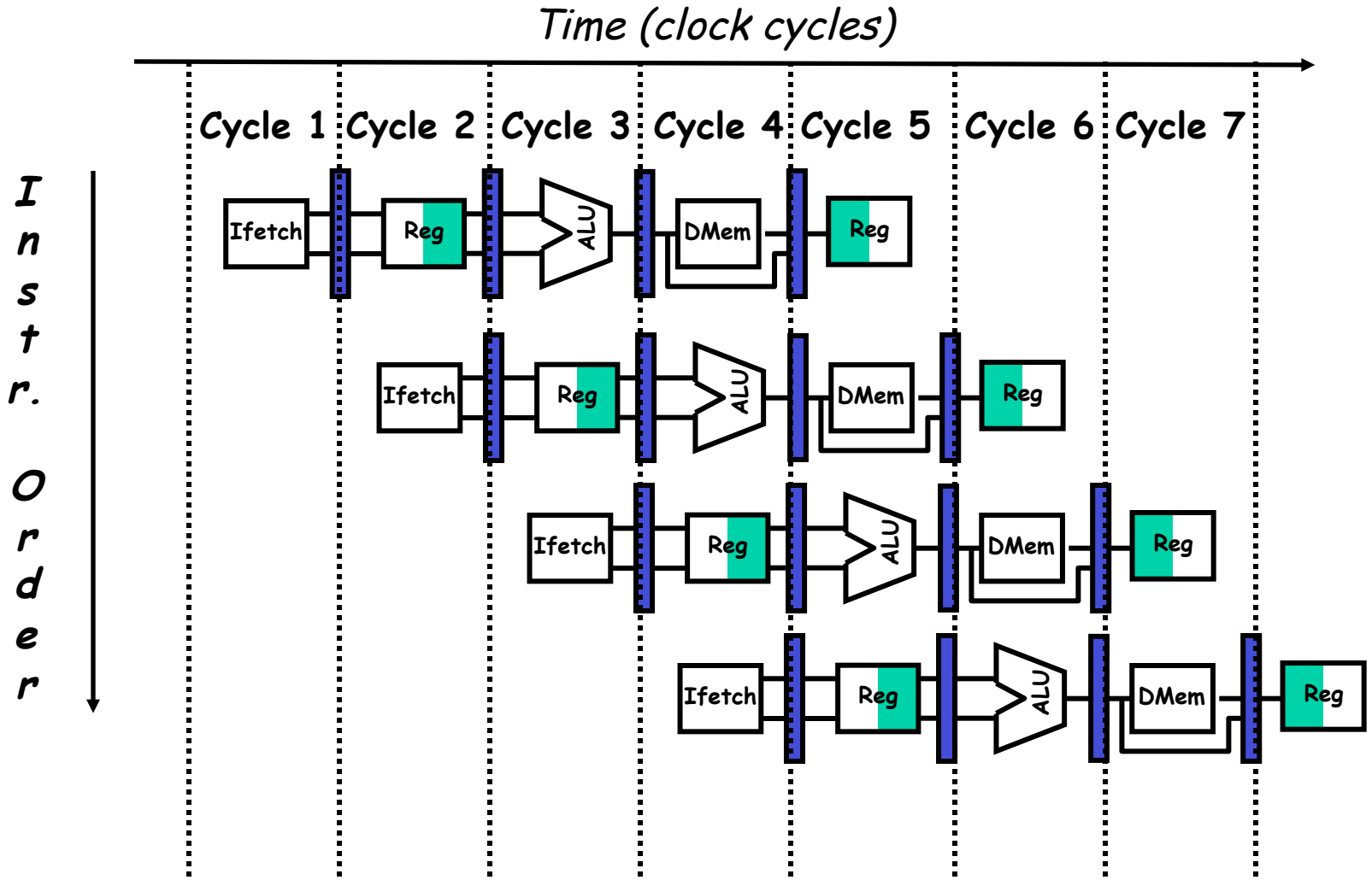
Conditional Branches: `beq $x,$y,offset`

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: Optimized Pipeline

- Register File used in 2 stages: Read access during ID and write access during WB
  - What happens if read a[illegible] register in [illegible]

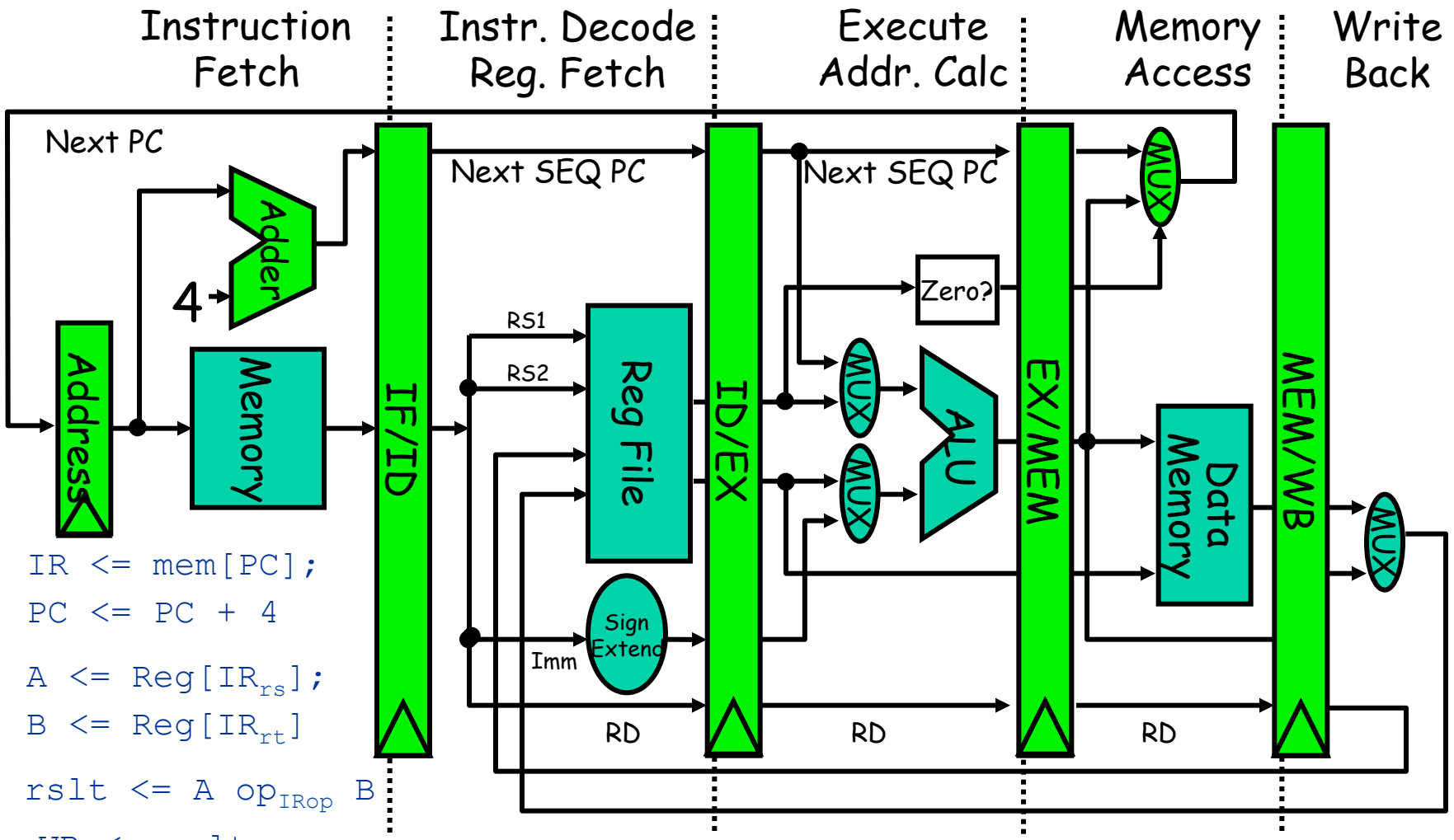
From now on, this is the Pipeline we are going to use

[illegible] the same [illegible] cycle?

[illegible]ecessary to insert one stall

5 Steps of MIPS Datapath

Figure A.3, Page A-9



Data stationary control

local decode for each instruction phase / pipeline stage

Questions