ARC: Computer Architecture

tanguy.risset@insa-lyon.fr
Lab CITI, INSA de Lyon
Version du March 16, 2023

Tanguy Risset

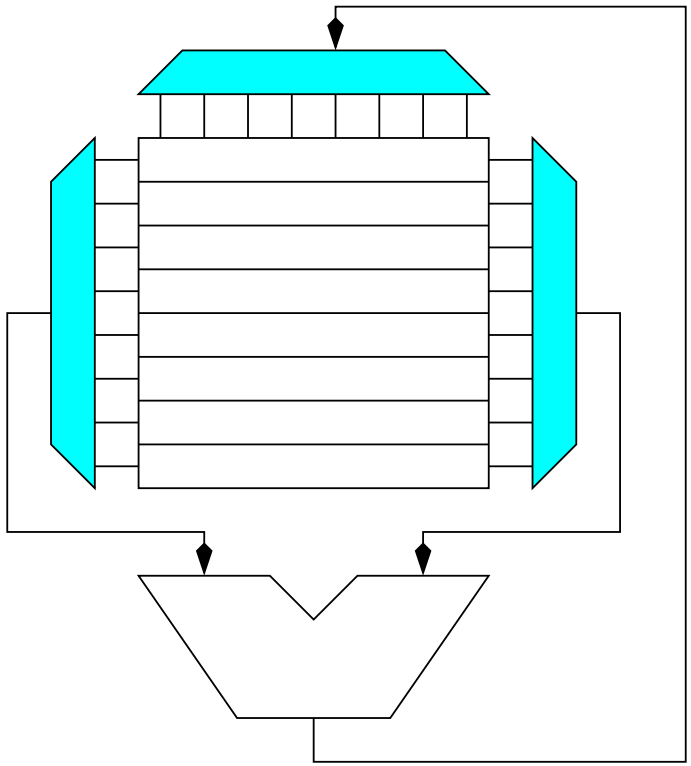
March 16, 2023

Table of Contents

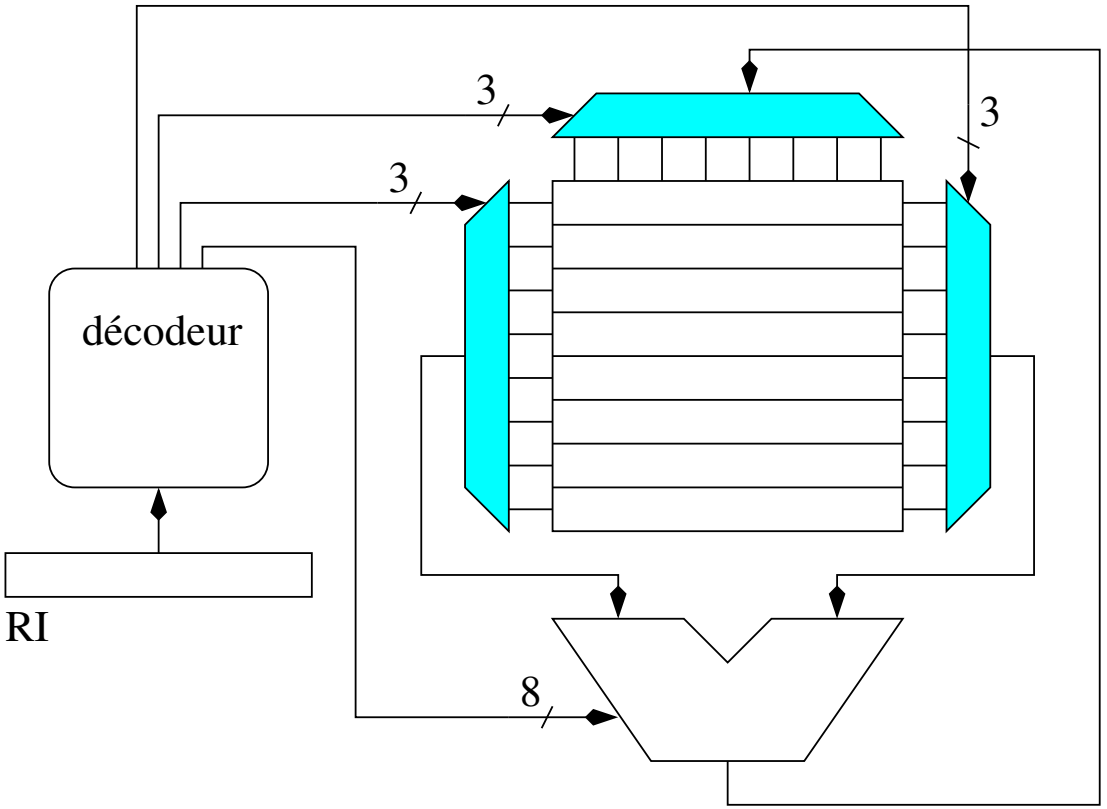
Toward a Von Neumann computer

- What you have seen:
  - How to build an ALU with transistors (combinatorial circuit)
  - How to build a controller with transistor (automate)
  - Basic Von Newmann behavior:
    - Registers
    - ALU (or datapath)
    - control unit (automaton)
    - Memory (and bus)
- Lets recall the basic Von Neumann behavior

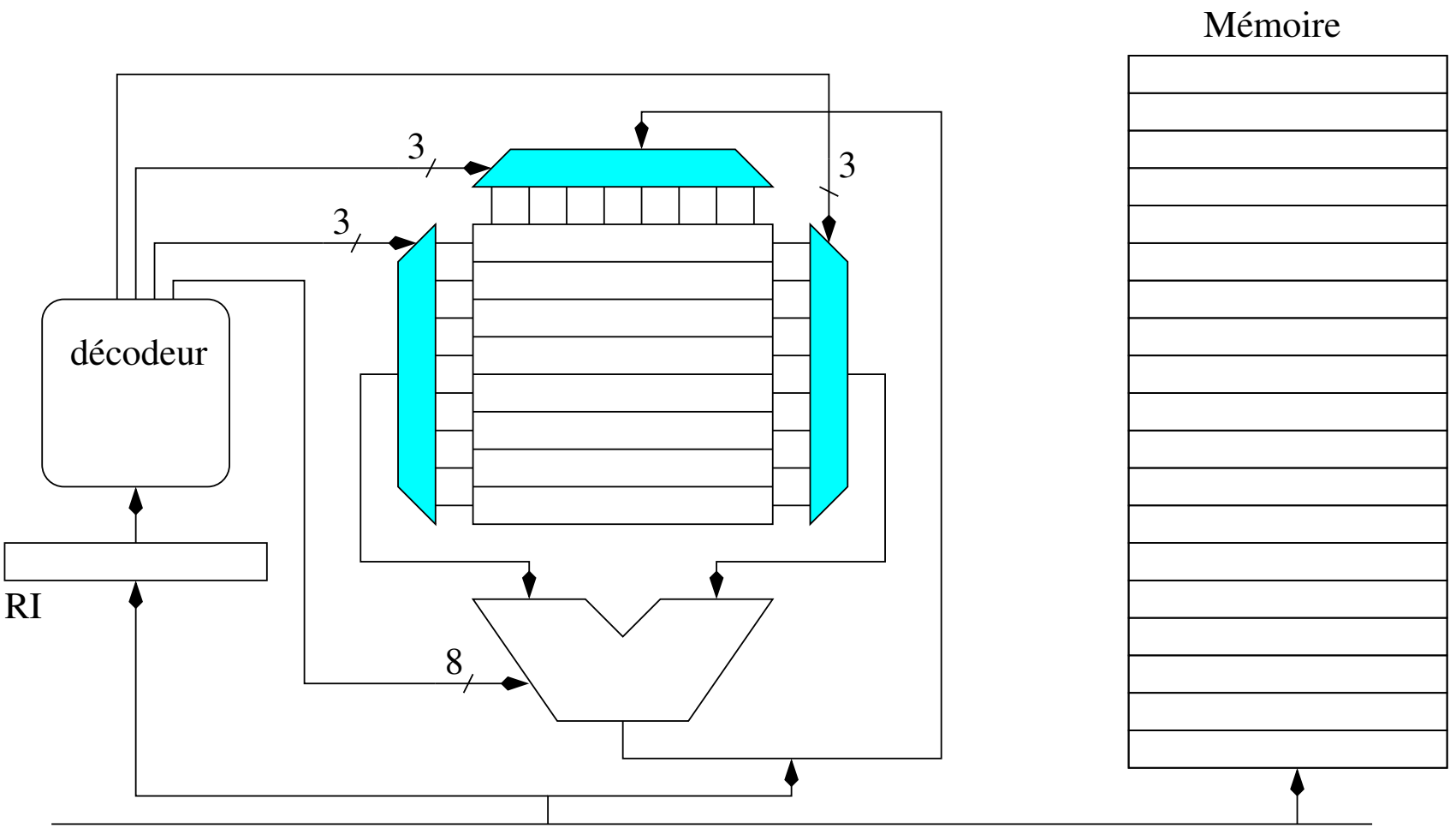
Program execution on a Processor (8 general purpose registers)



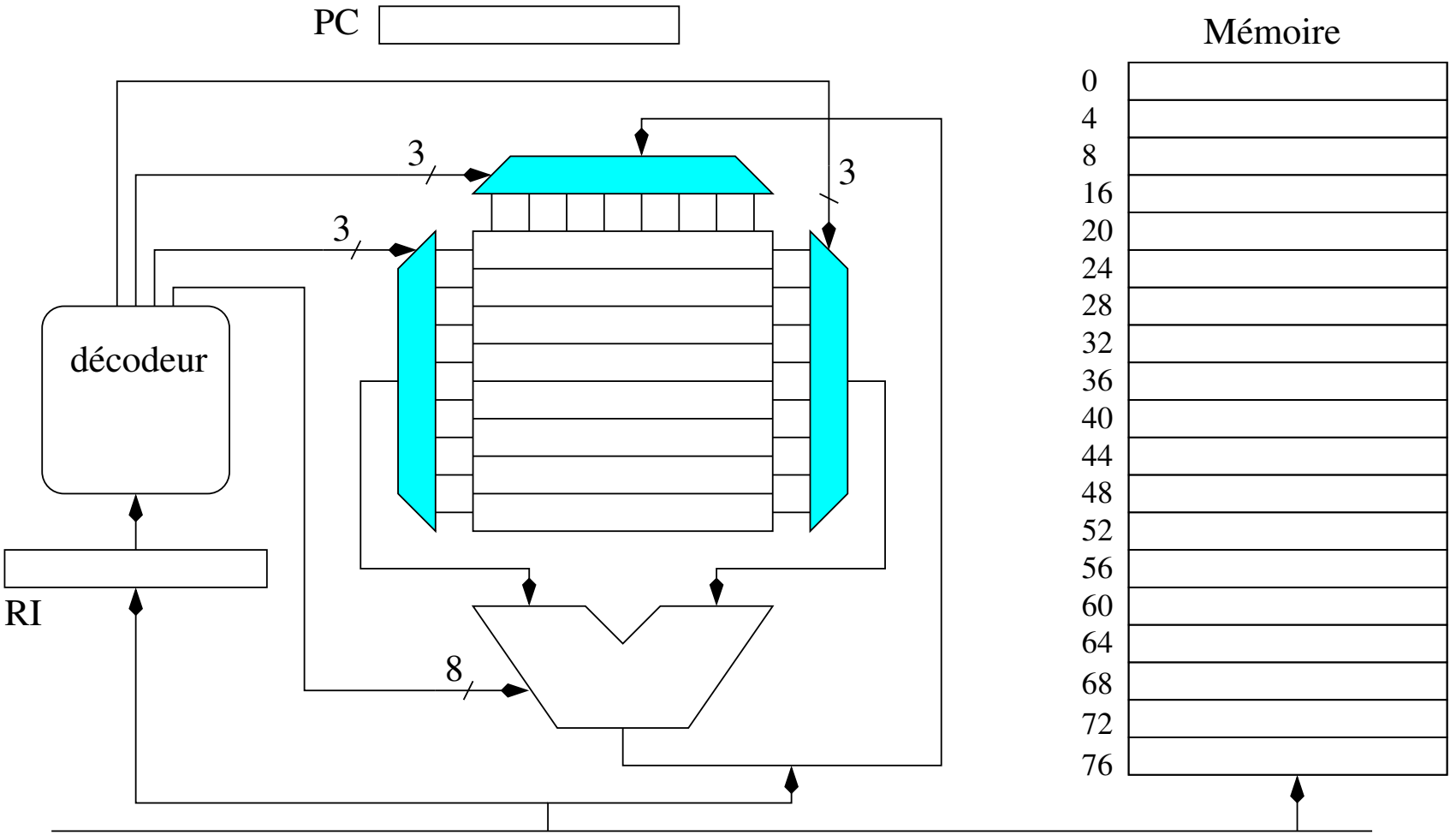
Program execution on a Processor (8 general purpose registers)



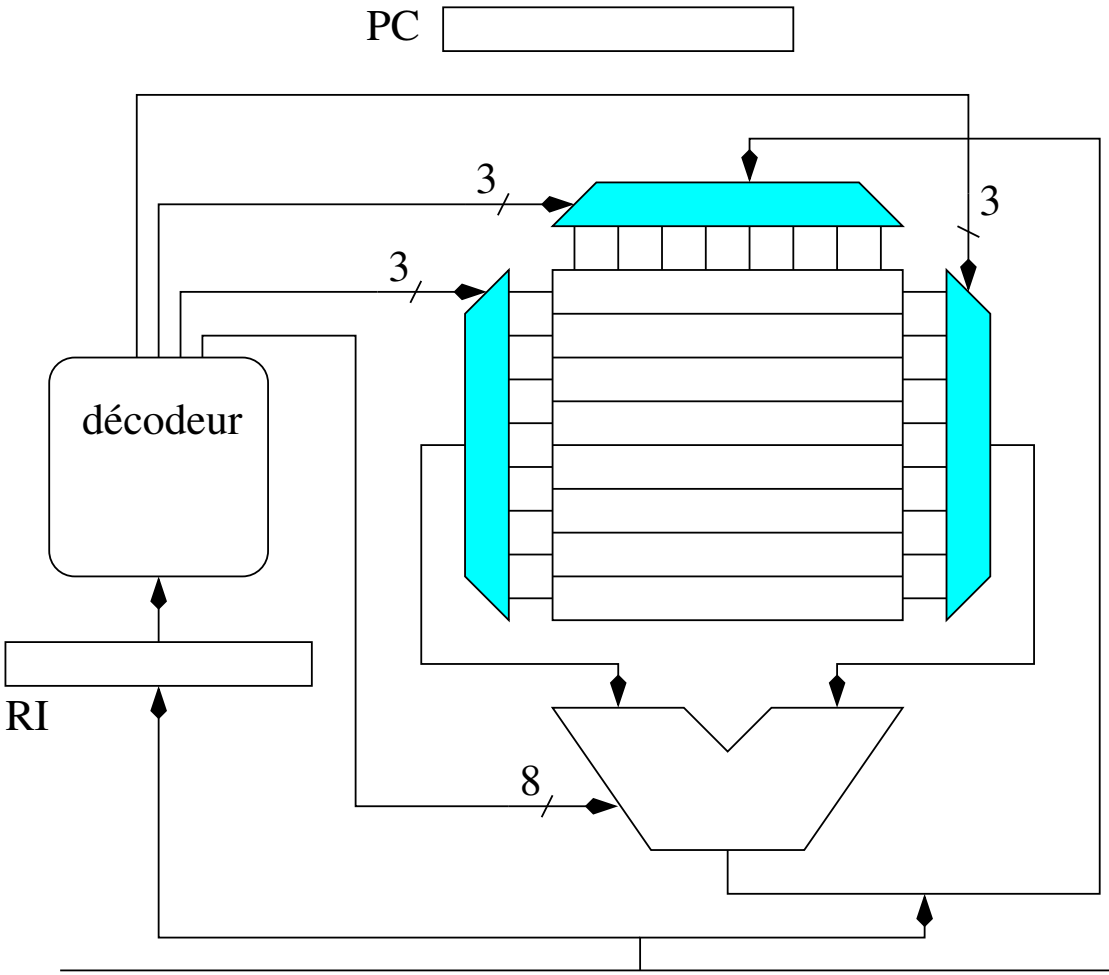
Program execution on a Processor (8 general purpose registers)

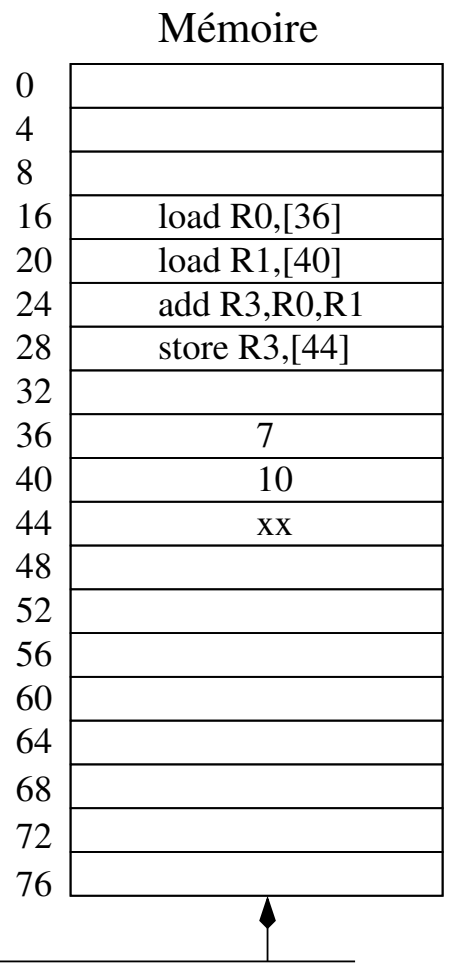


Program execution on a Processor (8 general purpose registers)

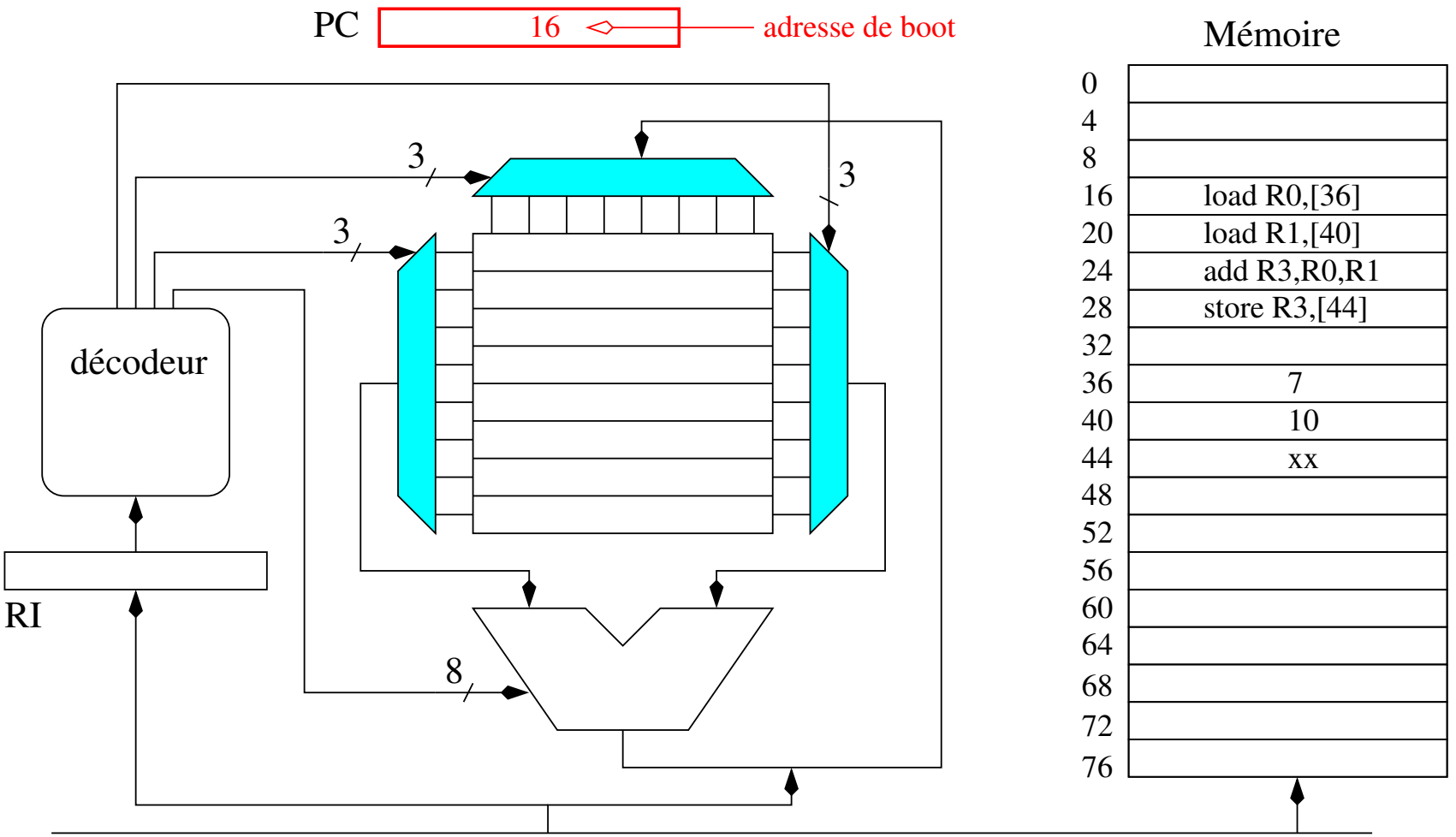


Program execution on a Processor (8 general purpose registers)

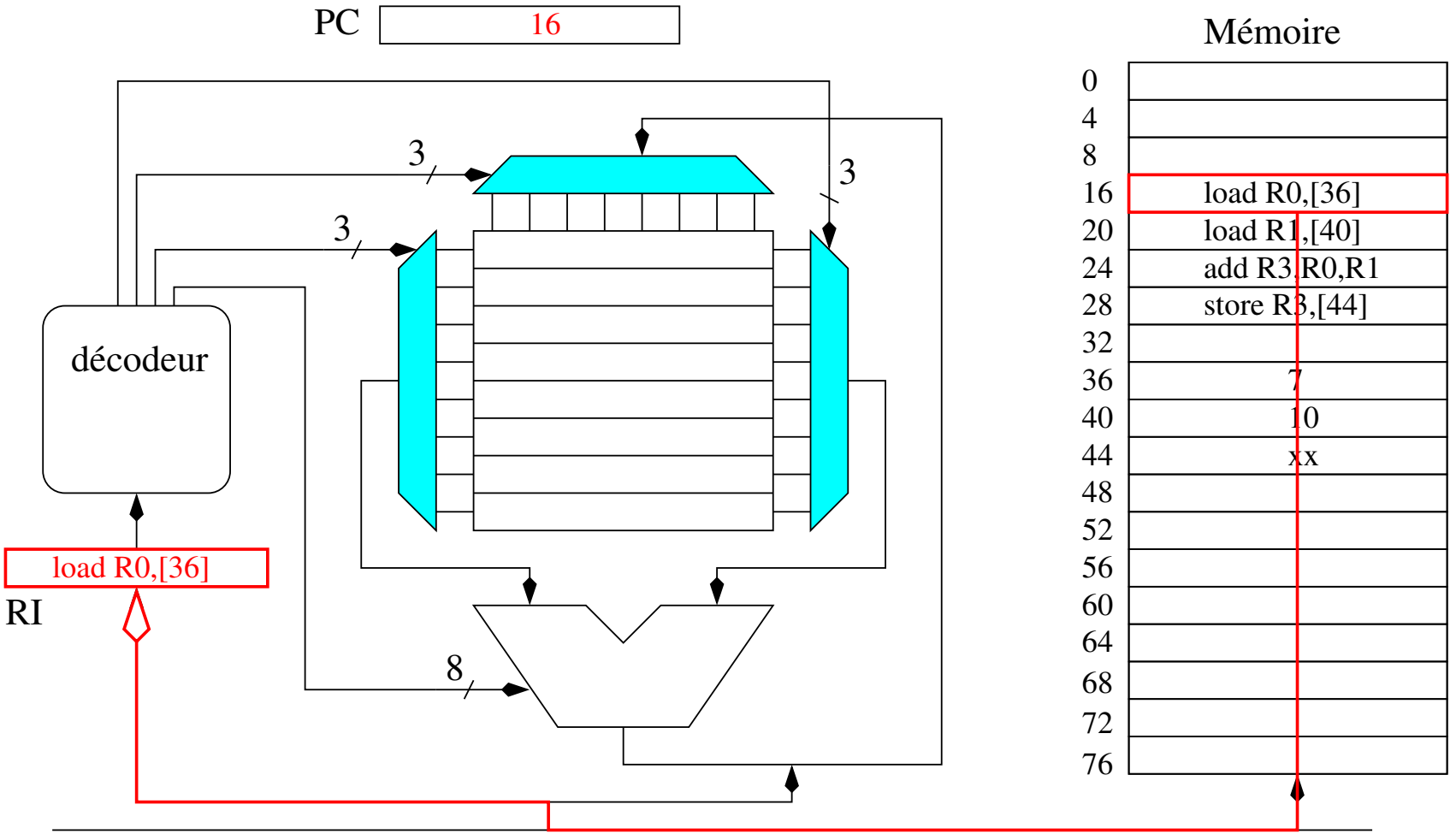




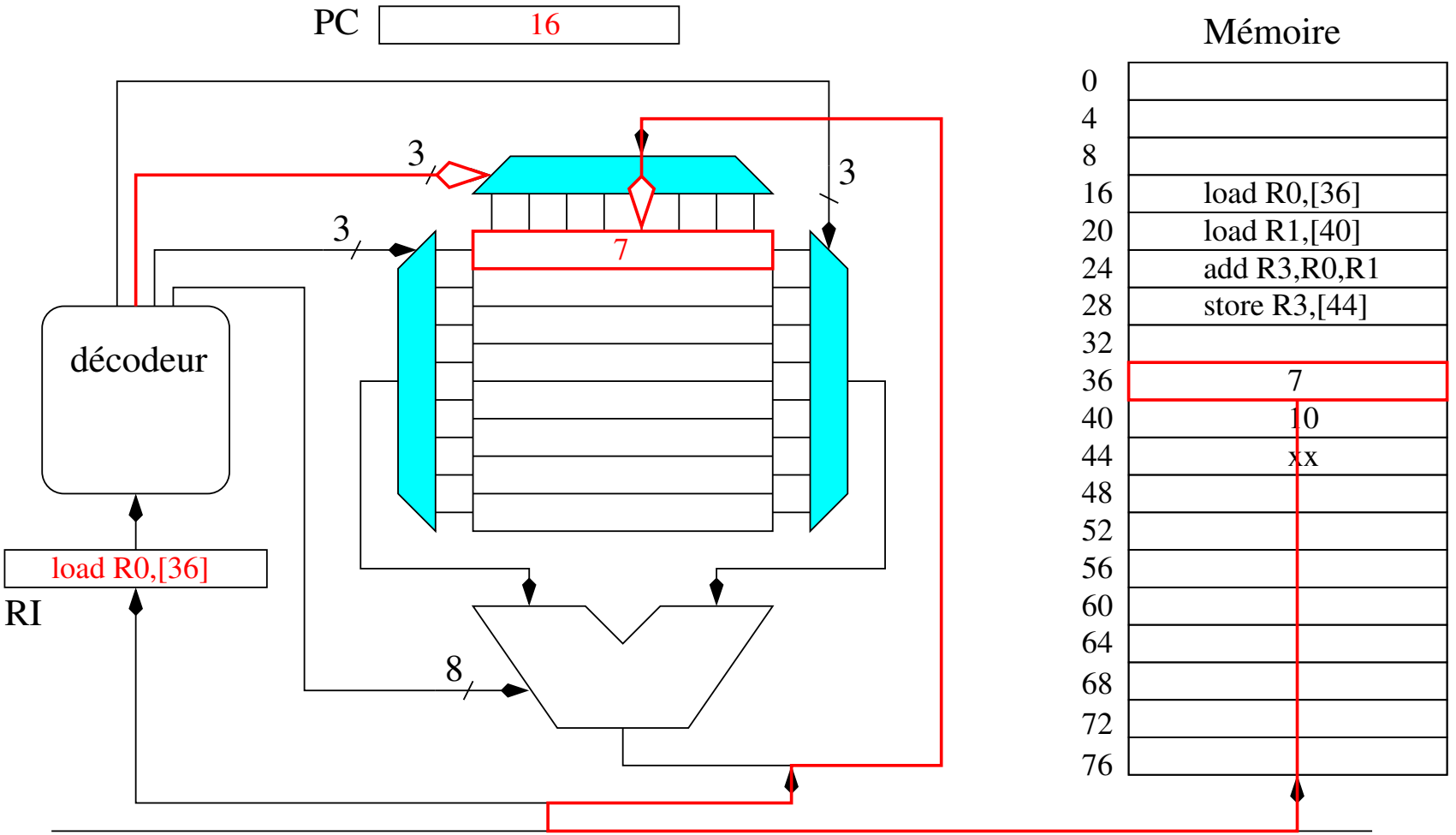
Program execution on a Processor (8 general purpose registers)



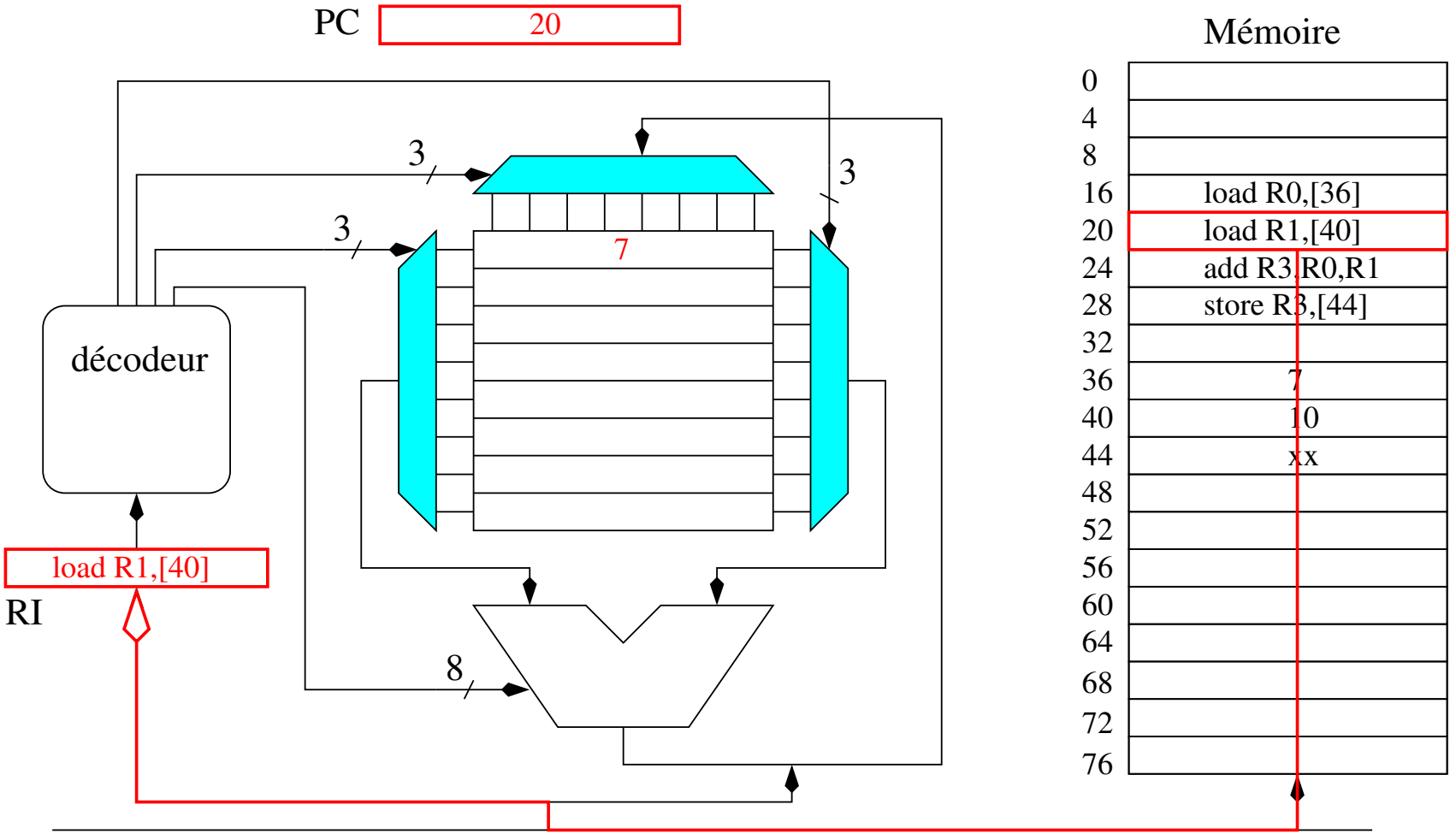
Program execution on a Processor (8 general purpose registers)



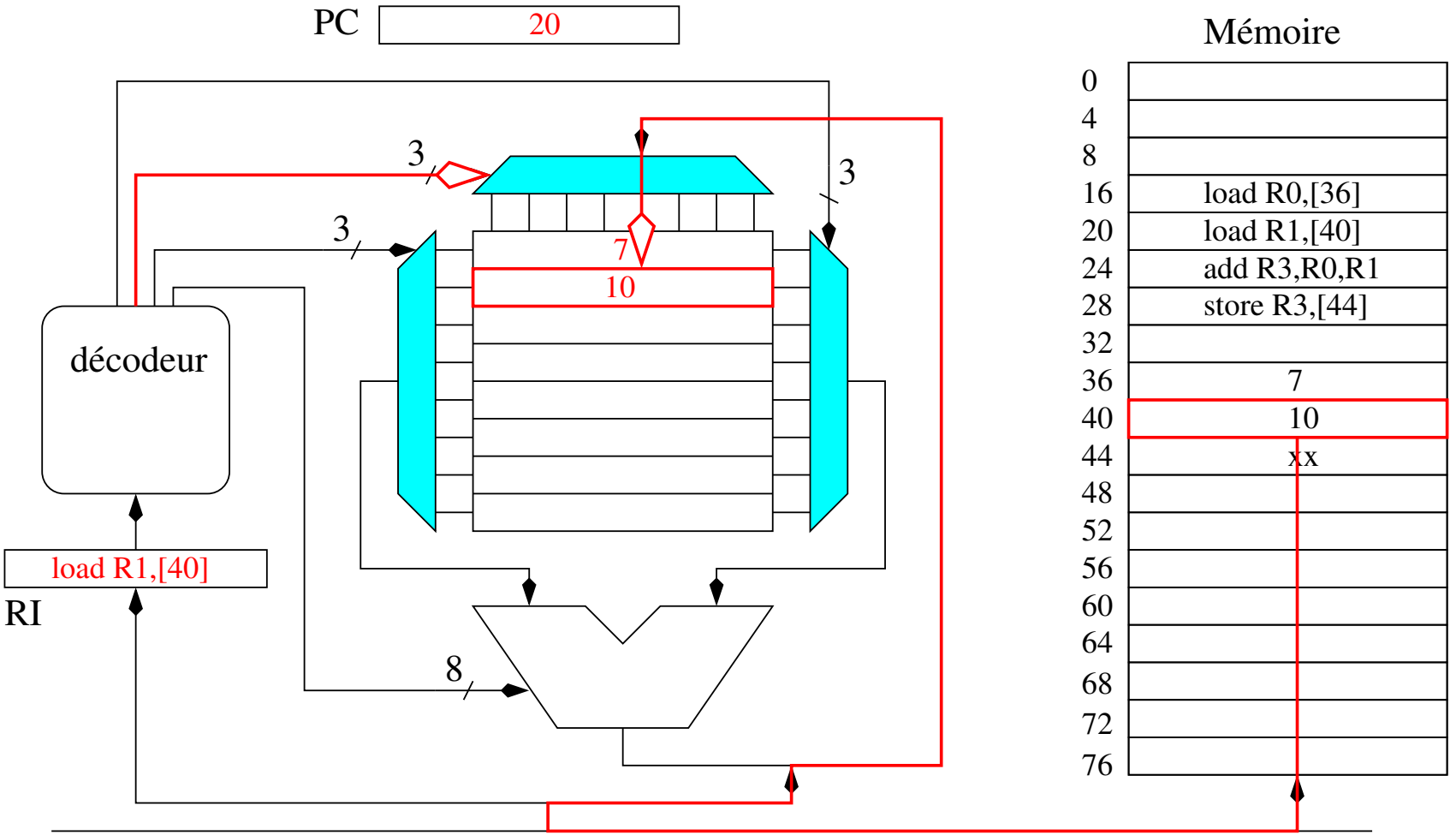
Program execution on a Processor (8 general purpose registers)



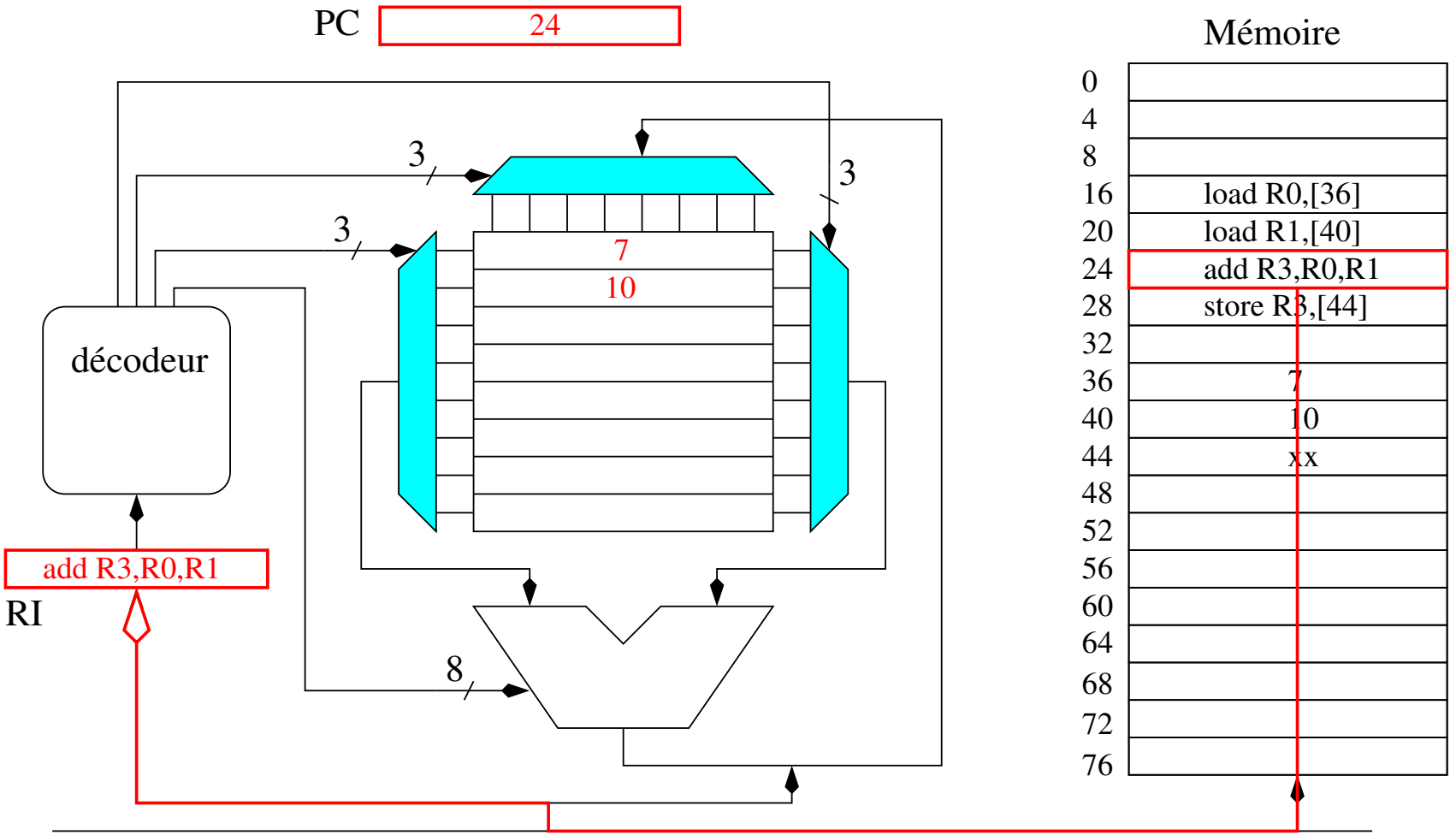
Program execution on a Processor (8 general purpose registers)



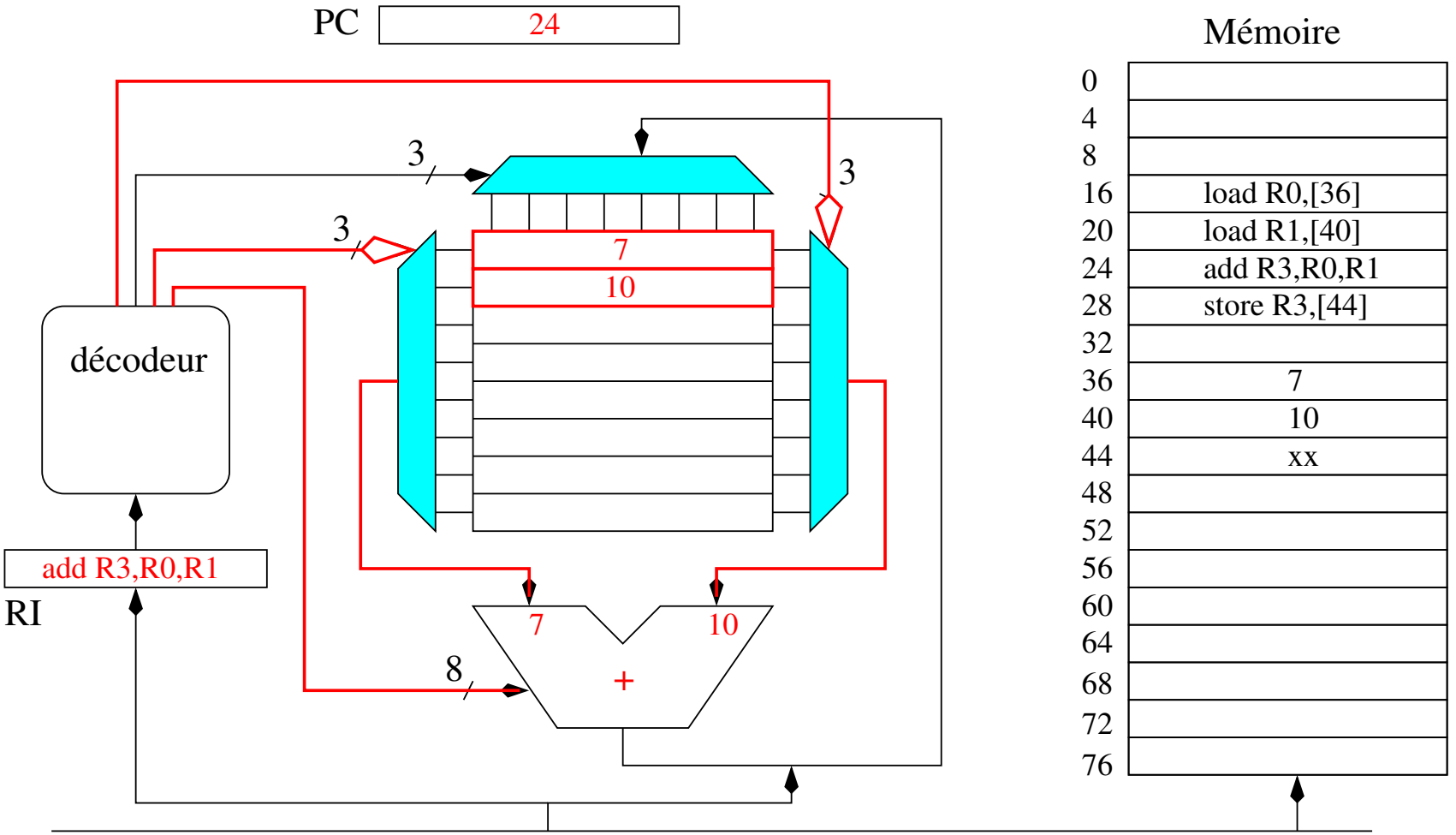
Program execution on a Processor (8 general purpose registers)



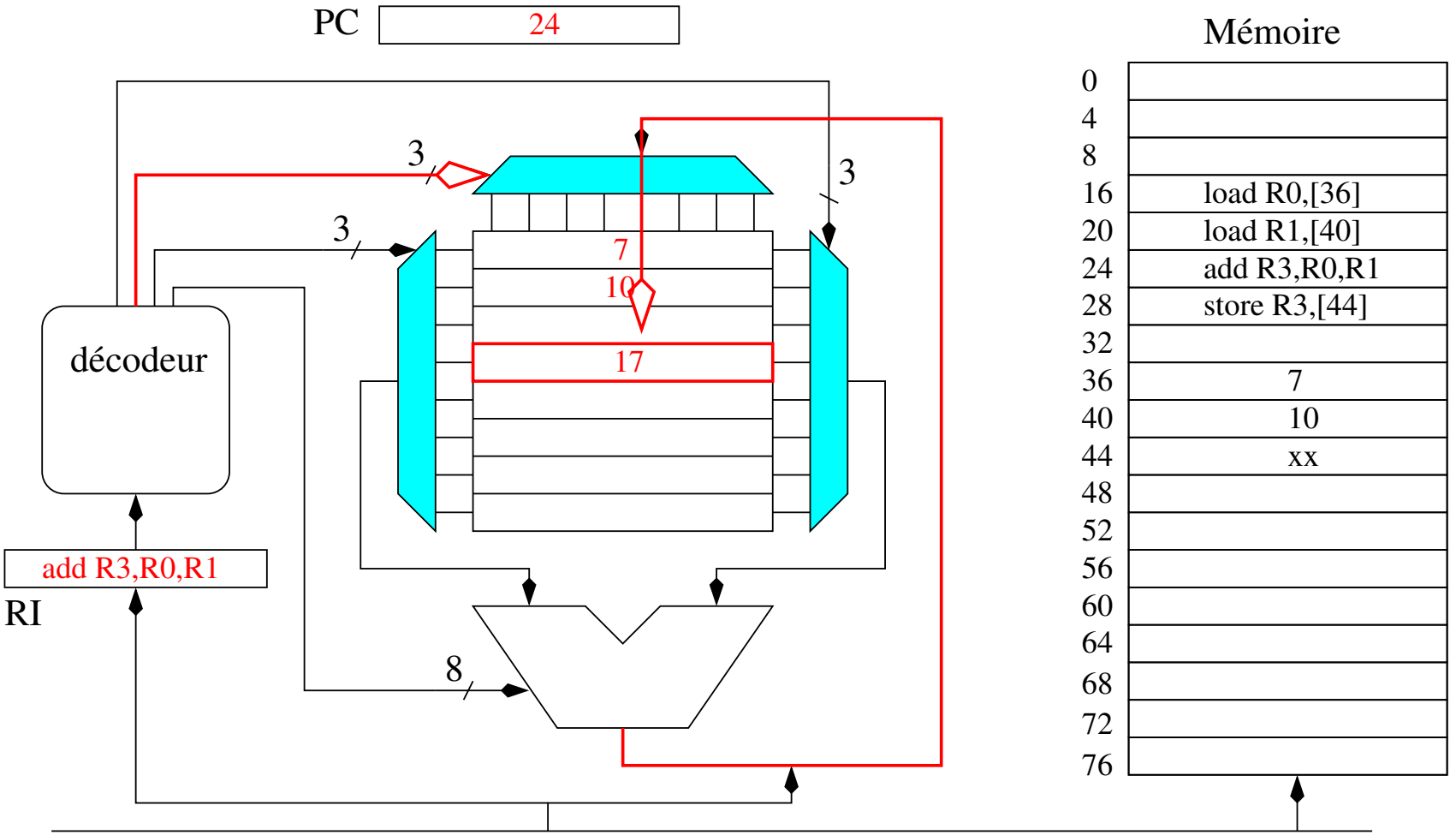
Program execution on a Processor (8 general purpose registers)



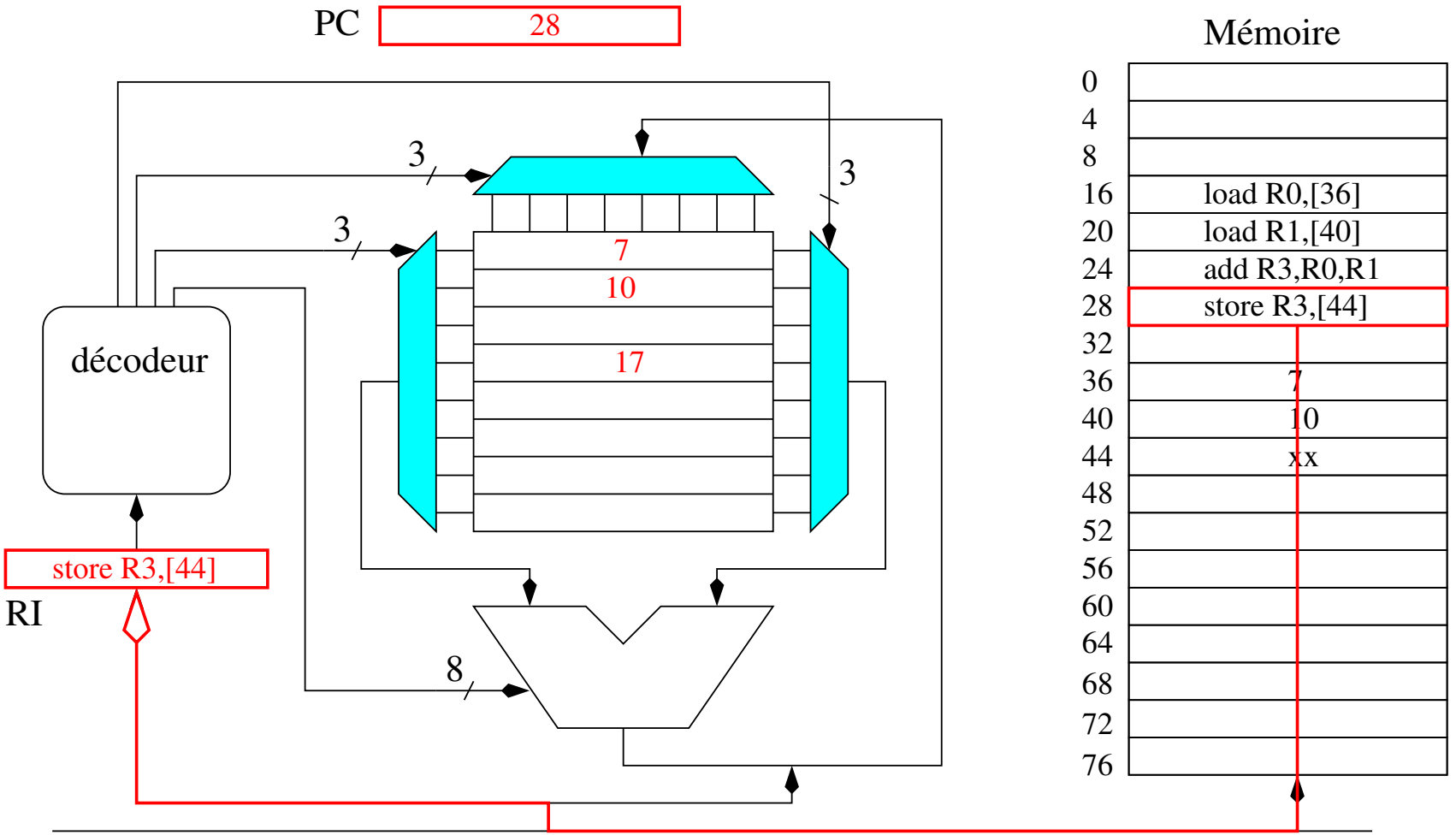
Program execution on a Processor (8 general purpose registers)



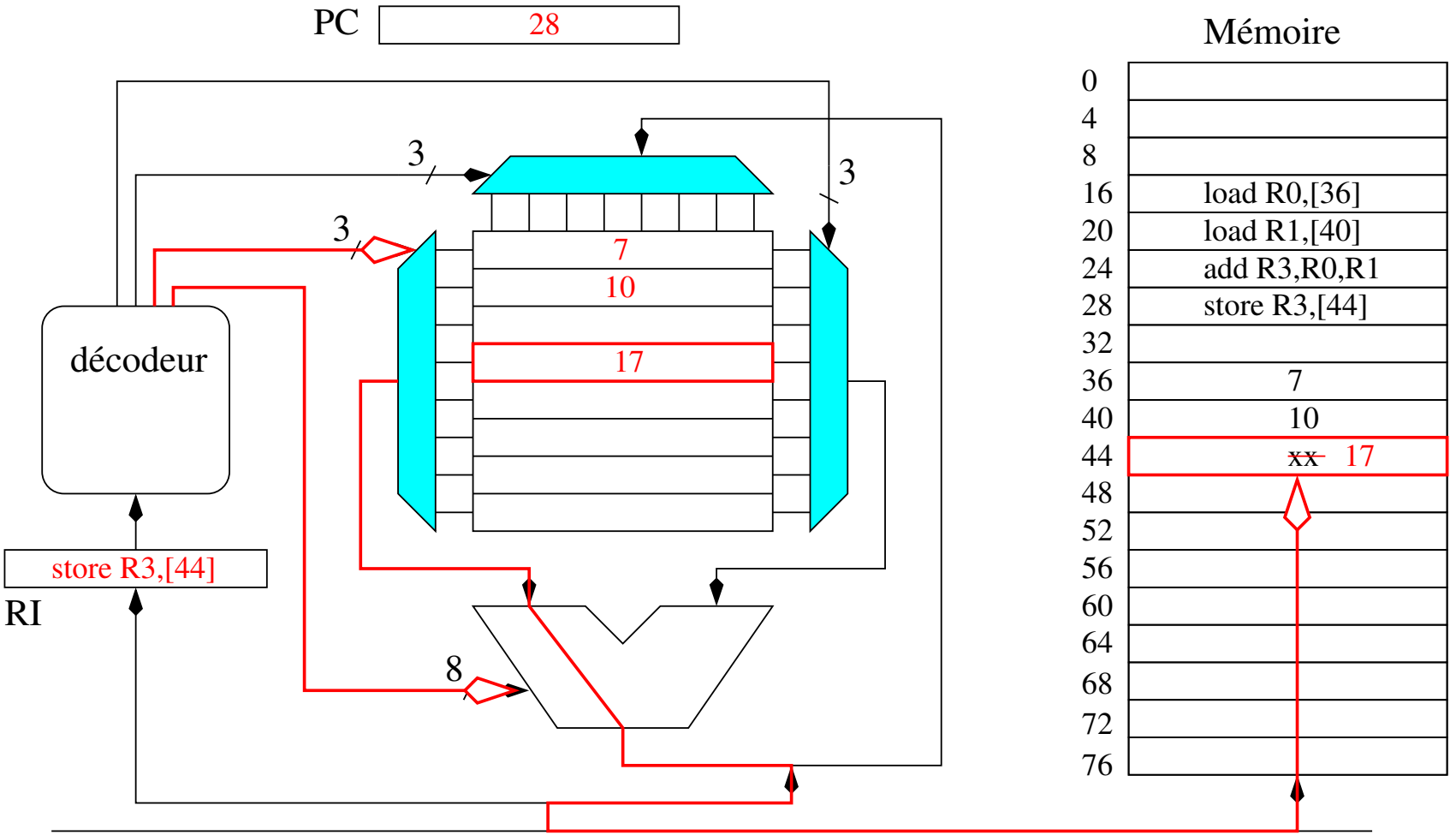
Program execution on a Processor (8 general purpose registers)



Program execution on a Processor (8 general purpose registers)



Program execution on a Processor (8 general purpose registers)



- Let's study in more details the instruction execution:
  - instruction execution cycle (Von Neumann cycle)
  - Instruction pipeline
  - ISA definition
  - RISC instruction set

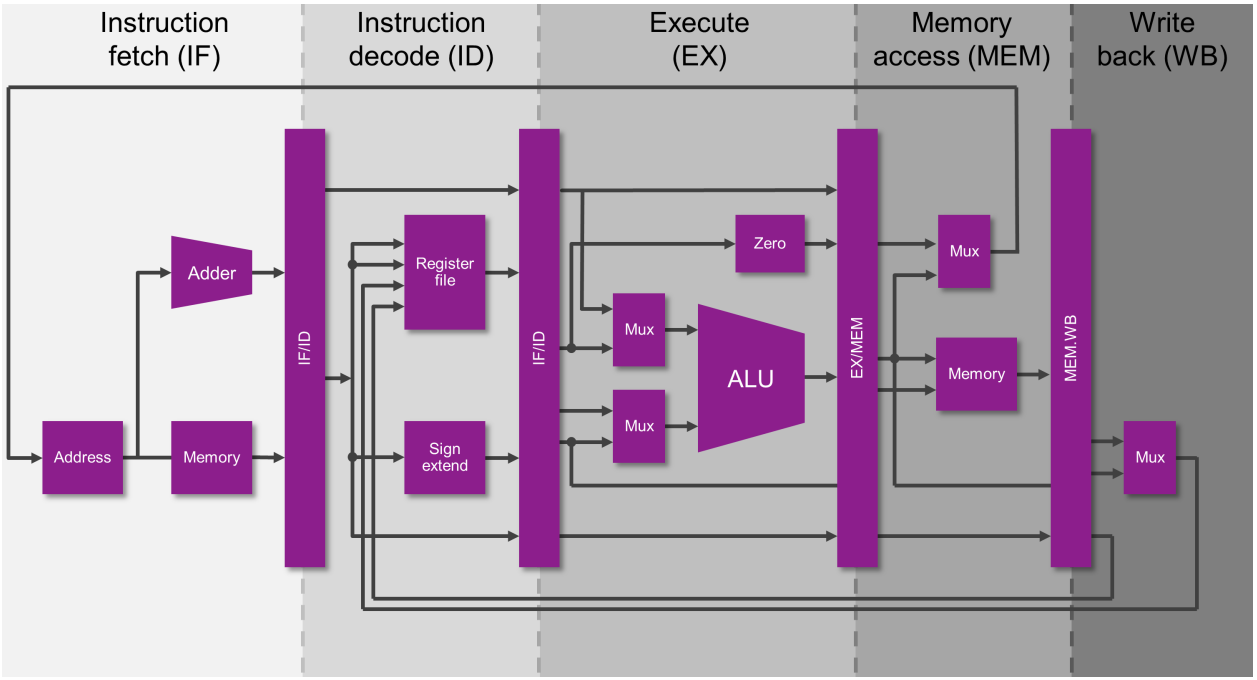
Table of Contents

The "Von Neumann cycle"

- The so-called Von Neumann cycle is simply the decomposition of the execution of an instruction in several independent stages.
- The number of stages depend on the processor, usually 5 stages are commonly used as example:
  - **Instruction Fetch** (IF)
    - Reads the instruction from memory (at address $PC) and write it in $IR.
  - **Instruction Decode** (ID)
    - computes what needs to be computed before execution: jump address destination, access to register, etc.
  - **Execute** (EX)
    - executes the instruction: ALU computation if needed
  - **Memory Access** (MEM)
    - Loads (or stores) data from memory if needed
  - **Write Back** (WB)
    - Writes the result into the register file if needed

The MIPS example

- The RISC paradigm was invented by Berkeley and popularized by Henessy and Patterson in the book on MIPS
- MIPS stands for *Microprocessor without Interlocked Pipeline Stages* and propose and architecture to execute each stage independently



from MIPS website `https://www.mips.com/`

Christian Wolf's slides

- Use Christian Wolf slides for explaining MIPS instruction pipeline
- Here

example of MIPS pipeline CPU architecture

- Taken from Henessy/patterson book

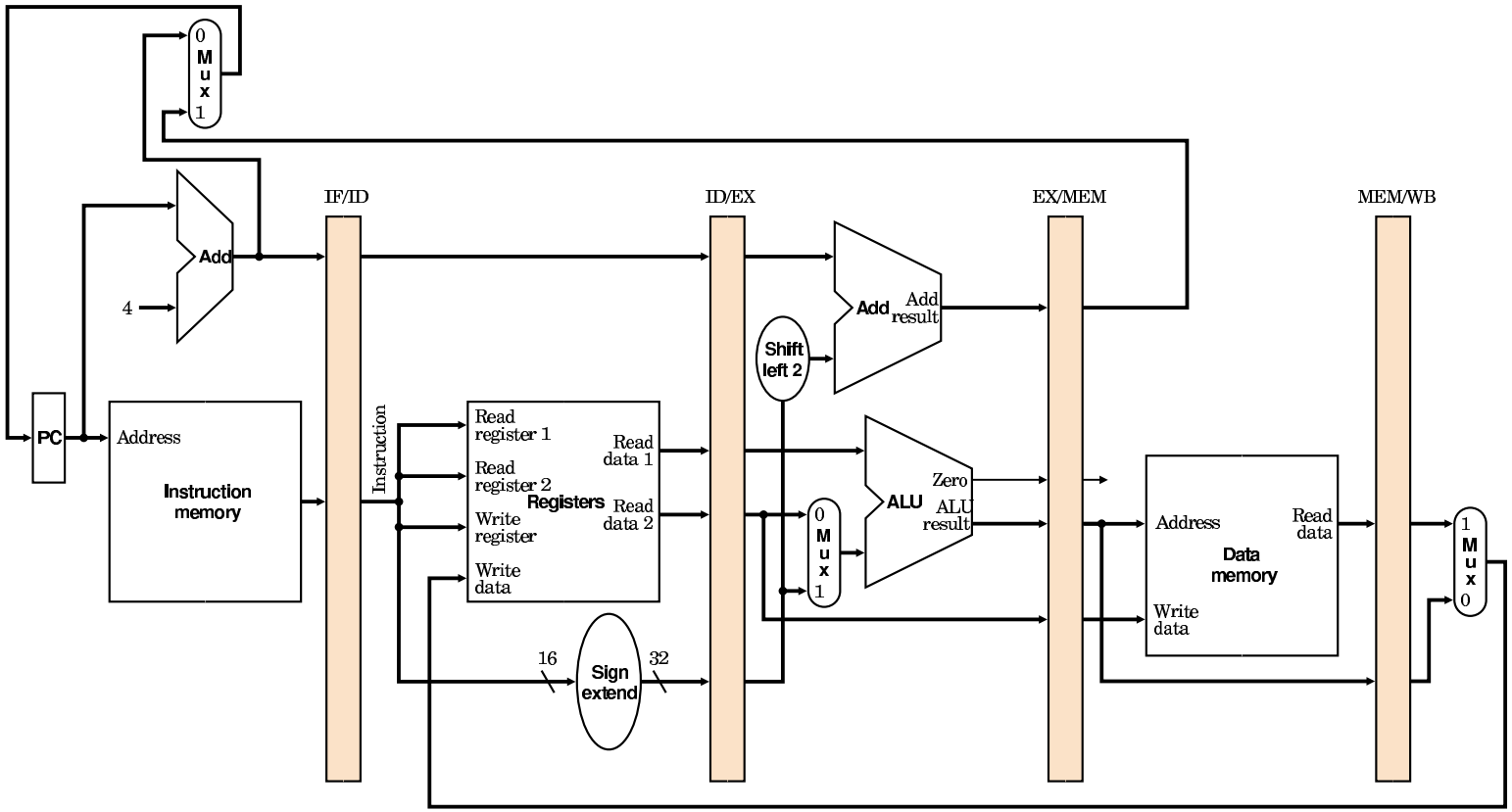
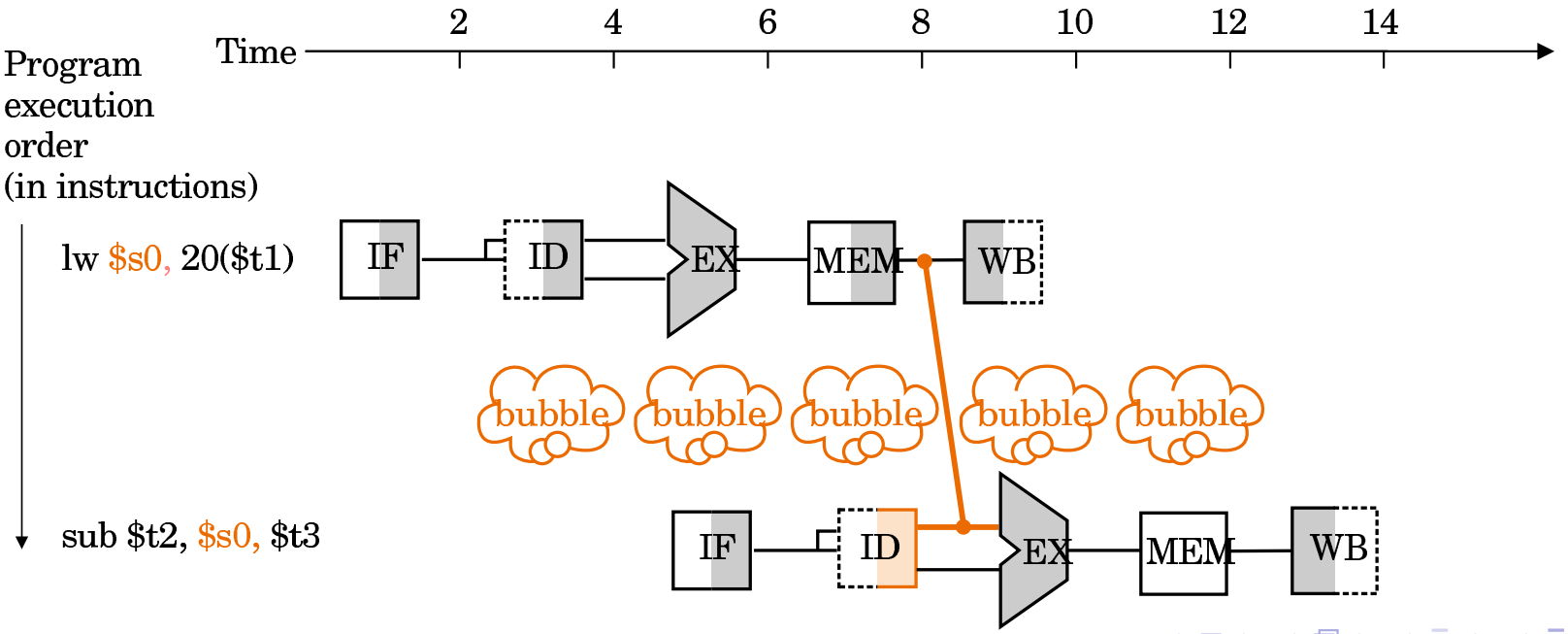


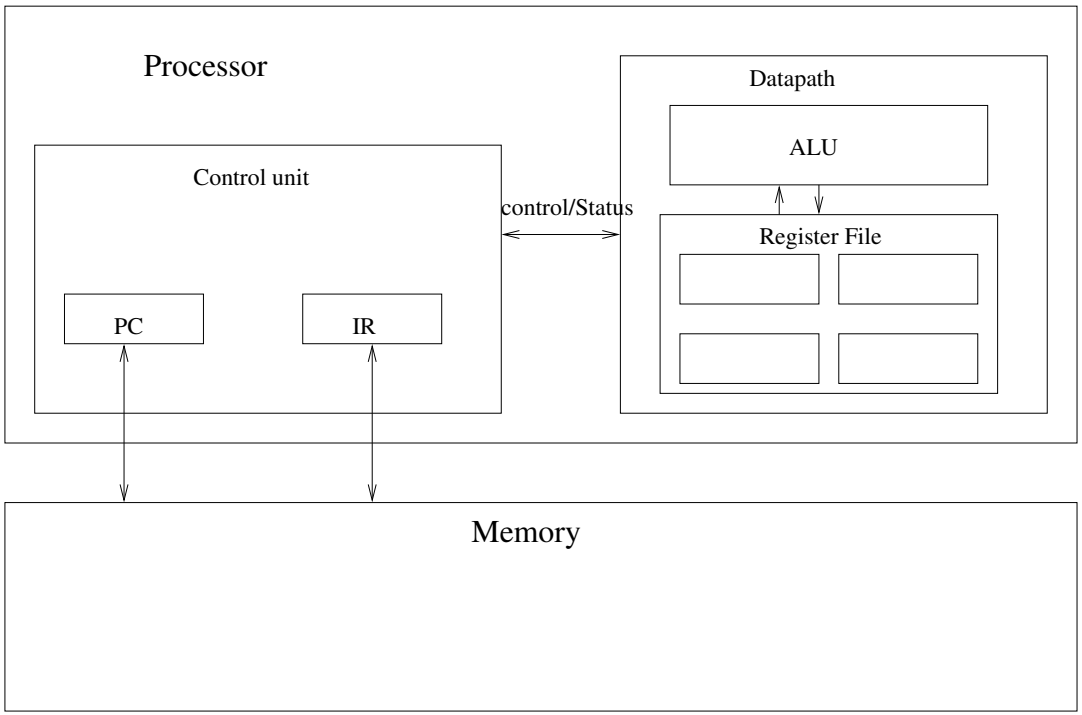
Illustration of bubble on MIPS

- When next instruction cannot be fetched directly (because it need the result of previous instruction for instance) it creates a "bubble"
- For instance: an addition using a register that was just loaded
- The value of the register will be available after the MEM stage of first instruction, hence we can delay on only on cycle, provided there is a *shortcut*.



Another illustration of instruction pipeline

- Go back to our previous representation of the processor and memory:
  - Von Neumann computer= Memory + CPU
  - CPU= = control Unit + Datapath
  - Datapath= ALU + Register file



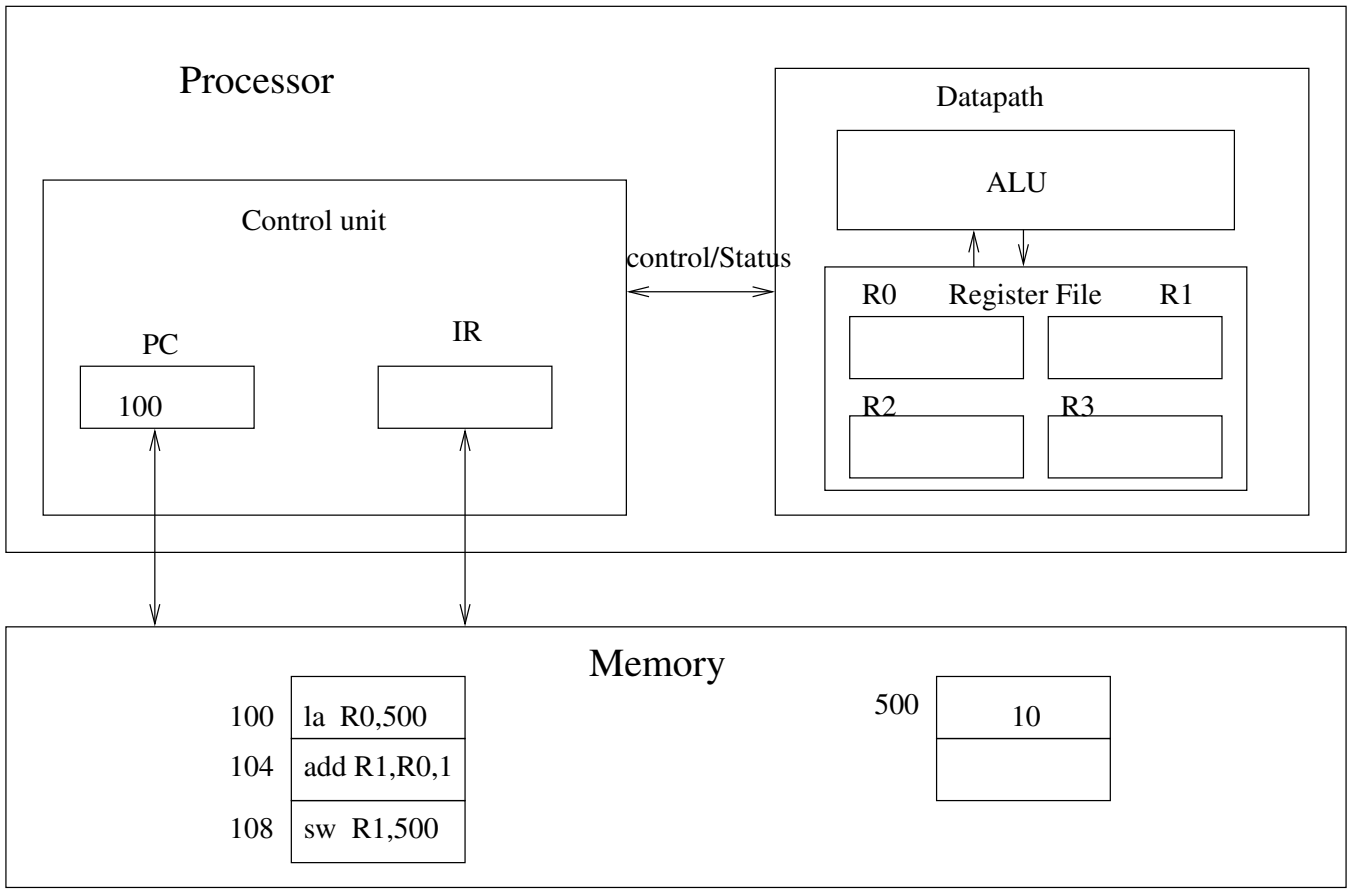
A pipeline example from MIPS

- Execute the sequence of assemby instruction:
  - load value at address 500 in register R0
  - Add 1 to R0 and put result in R1
  - store value of Register R1 at address 500
- (Think of `i=i+1`)
- Code:

```
la R0,500
add R1, R0, 1
sw  R1,500
```

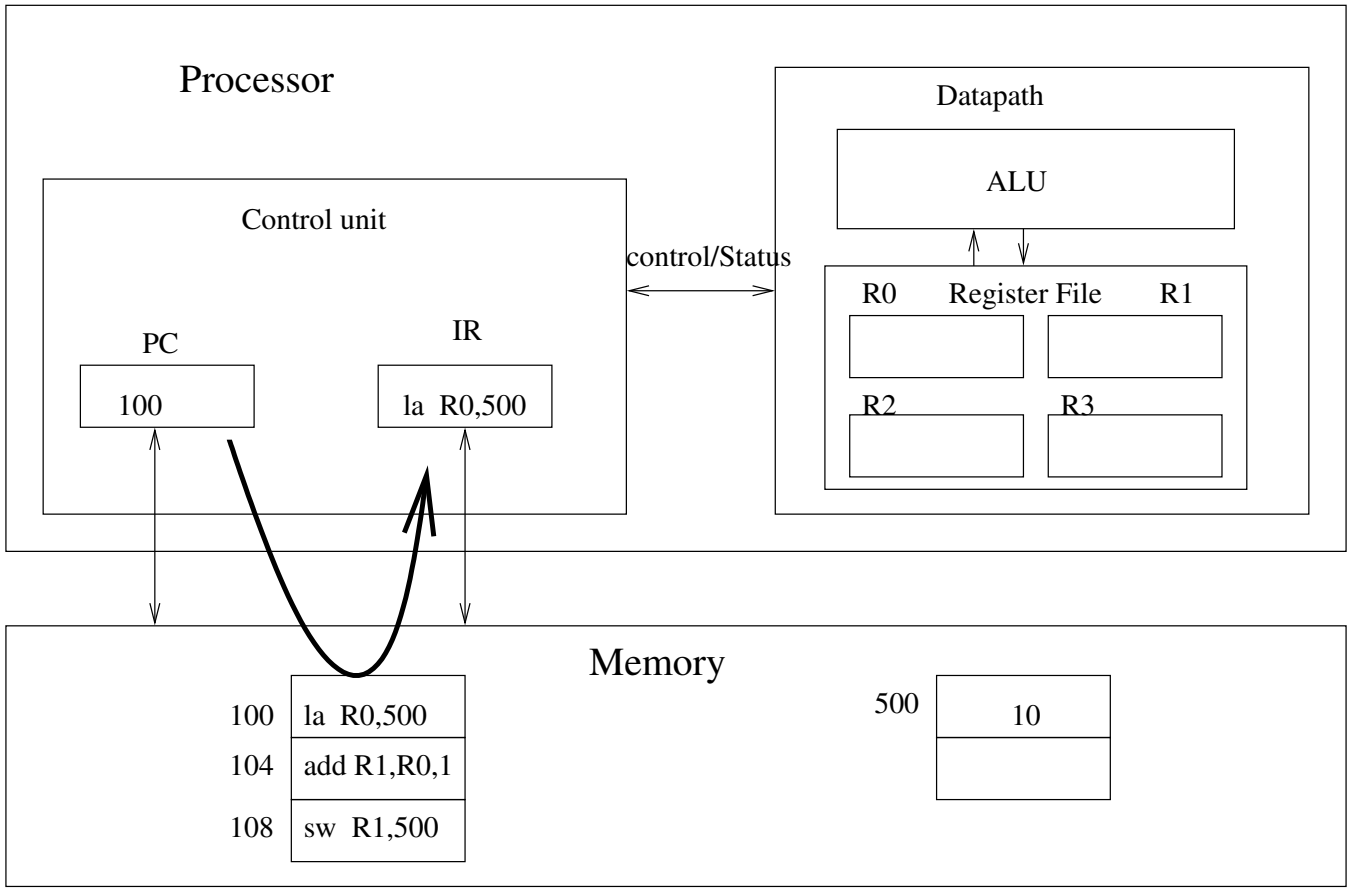
First possible execution: without pipeline

- Before execution starts, $PC contains the address of the first instruction: 100



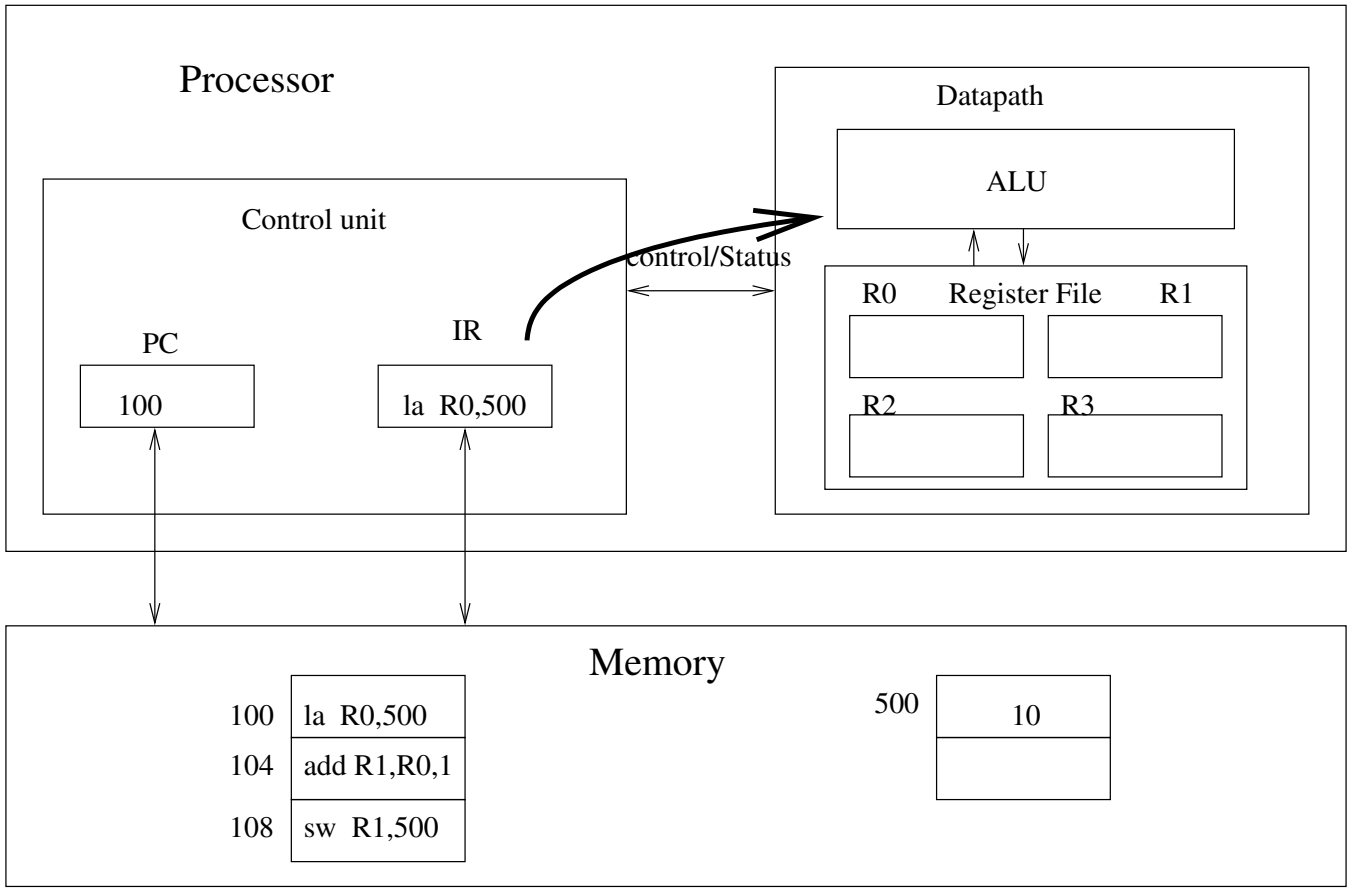
cycle 1

- Instruction Fetch



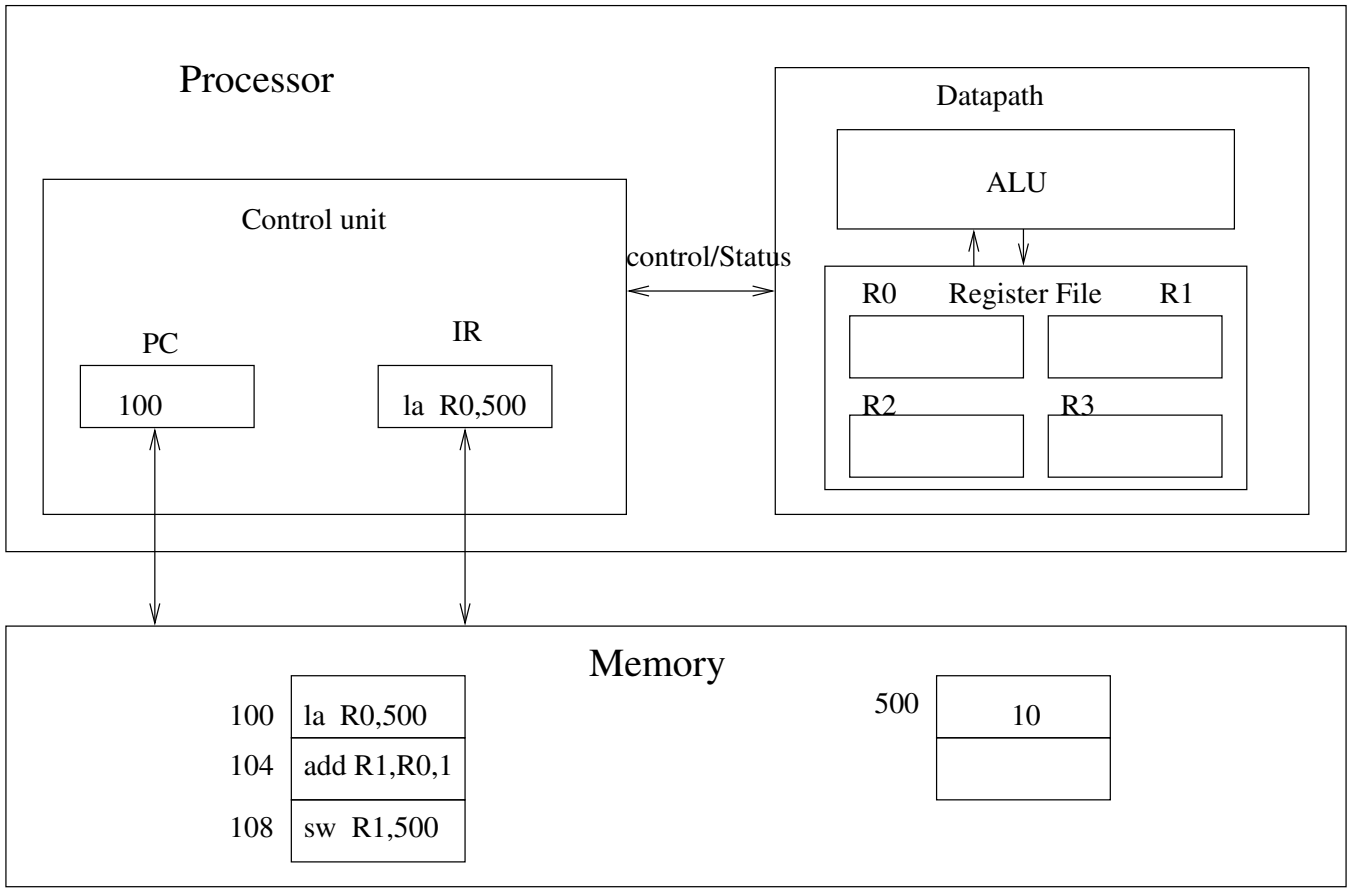
cycle 2

- Instruction Decode



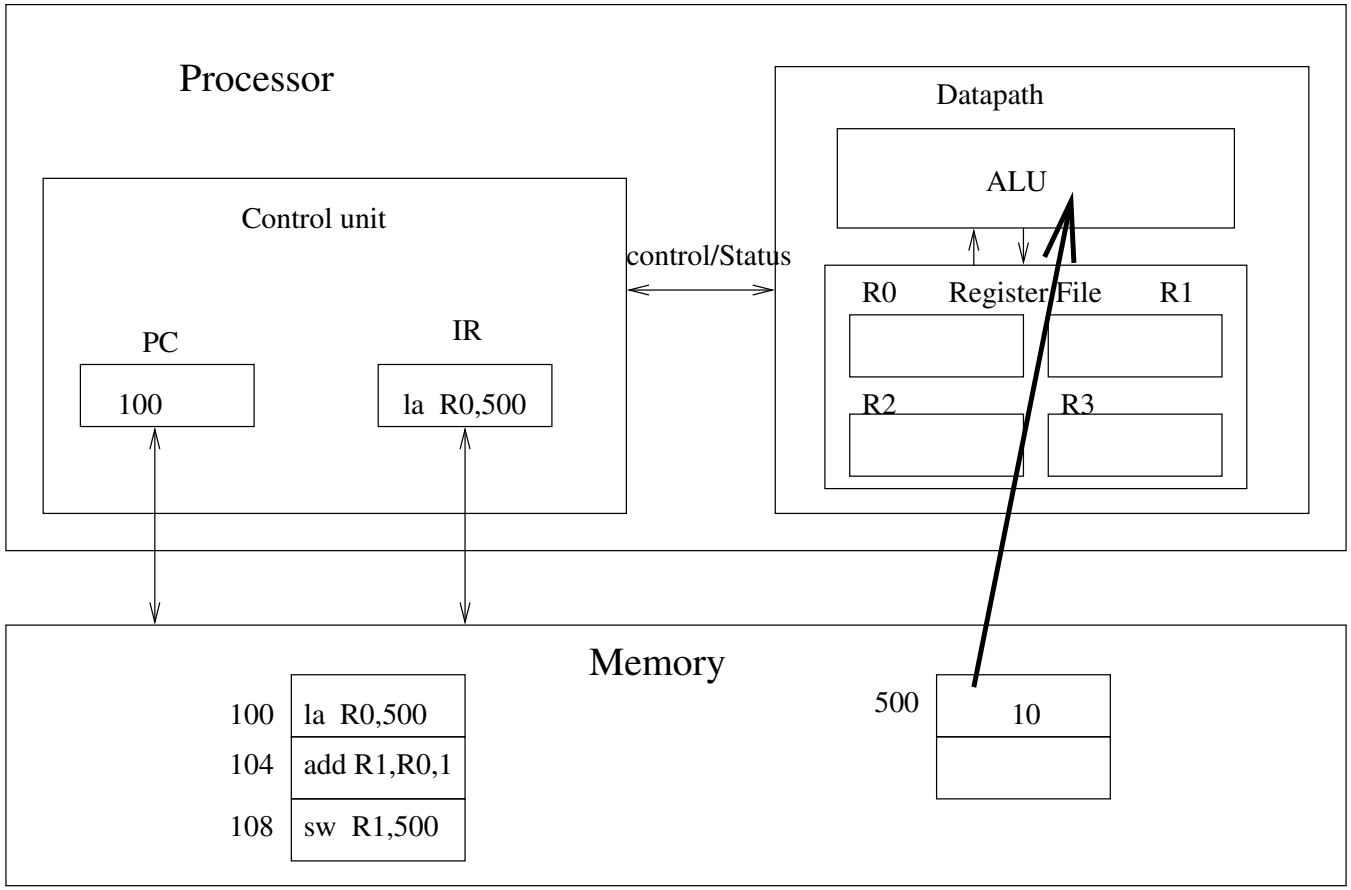
cycle 3

- Execute (nothing for load)



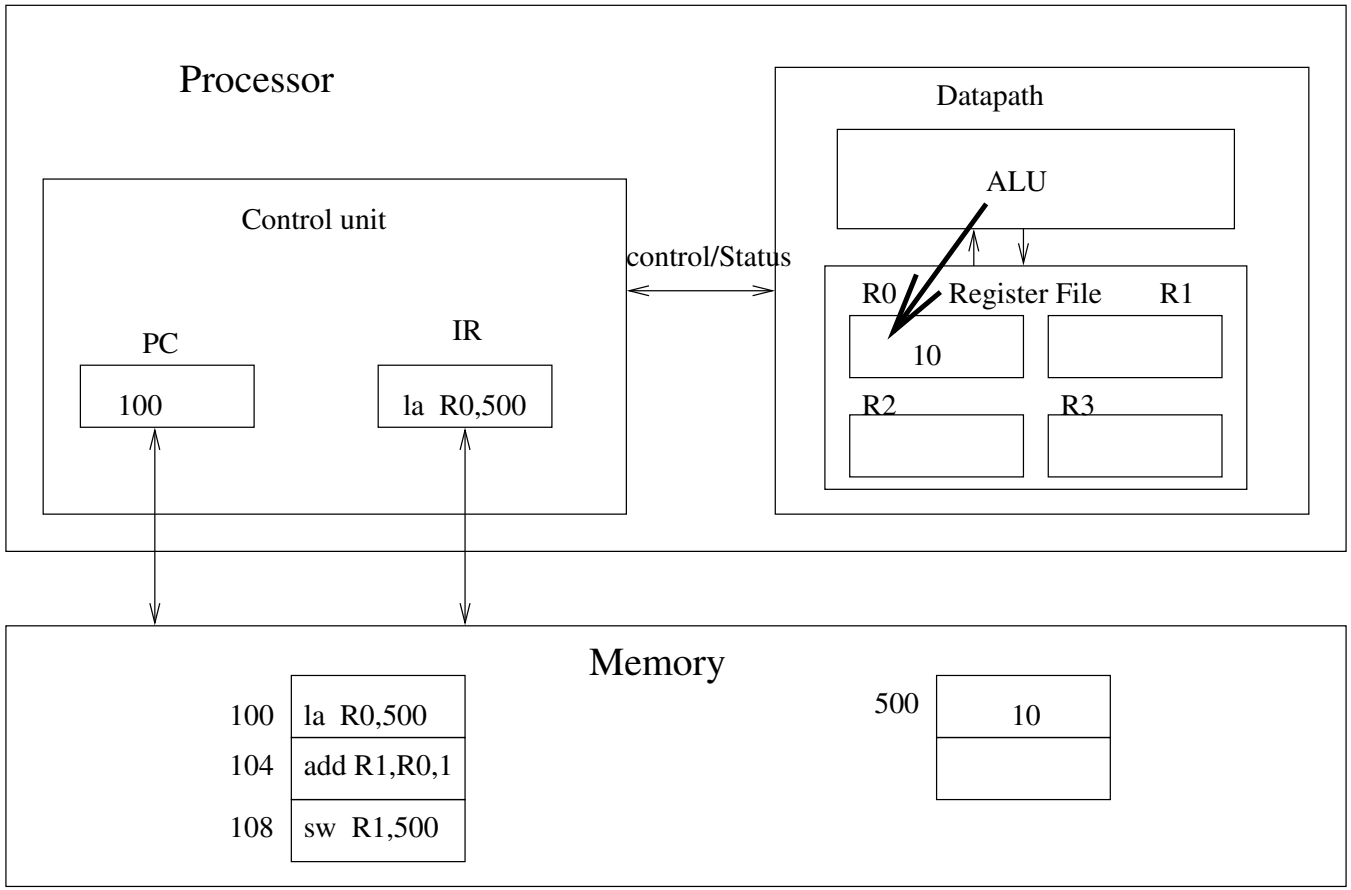
cycle 4

- Memory access



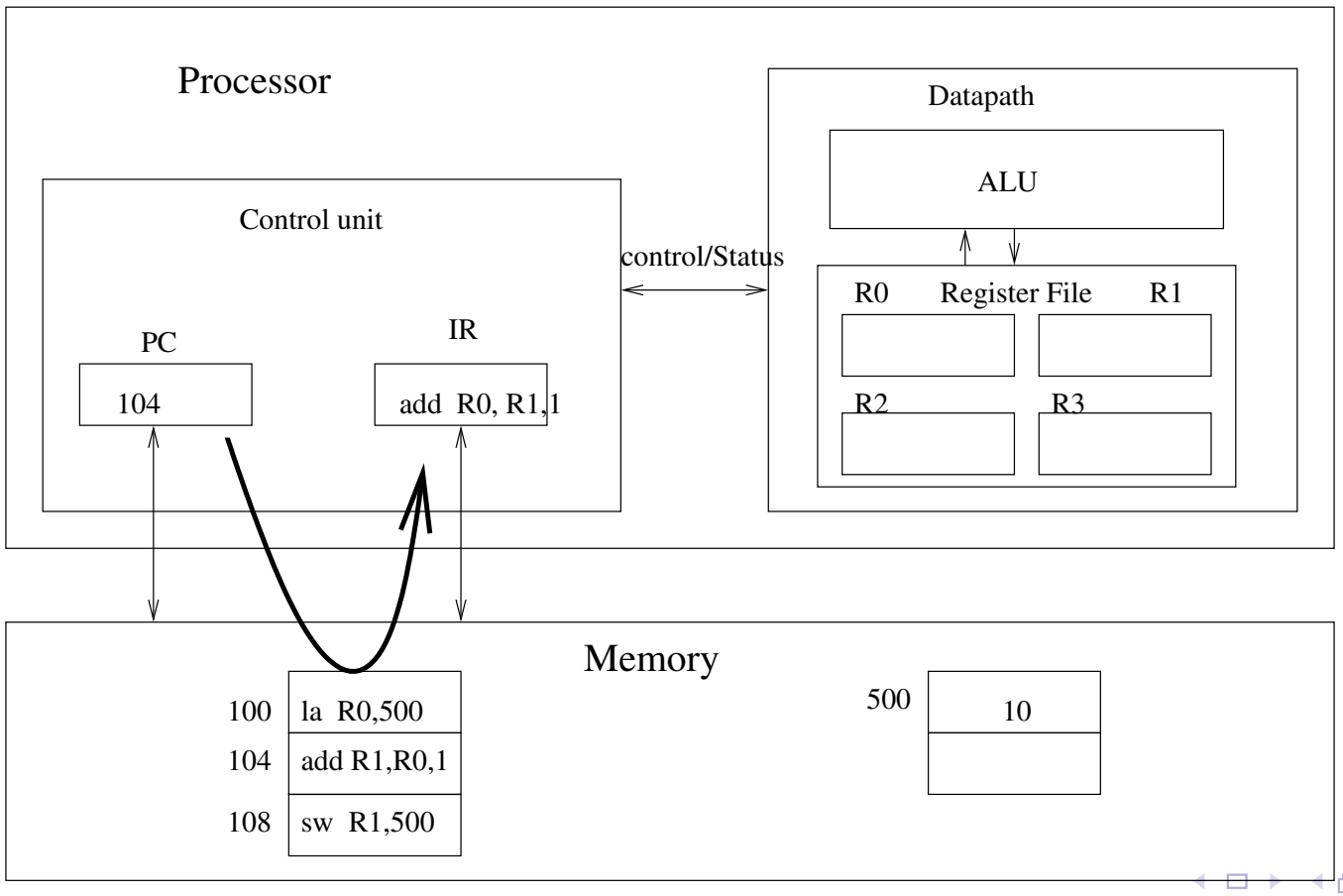
cycle 5

- Write Back



cycle 6

- increment $PC
- Fetch next instruction
- etc. etc.

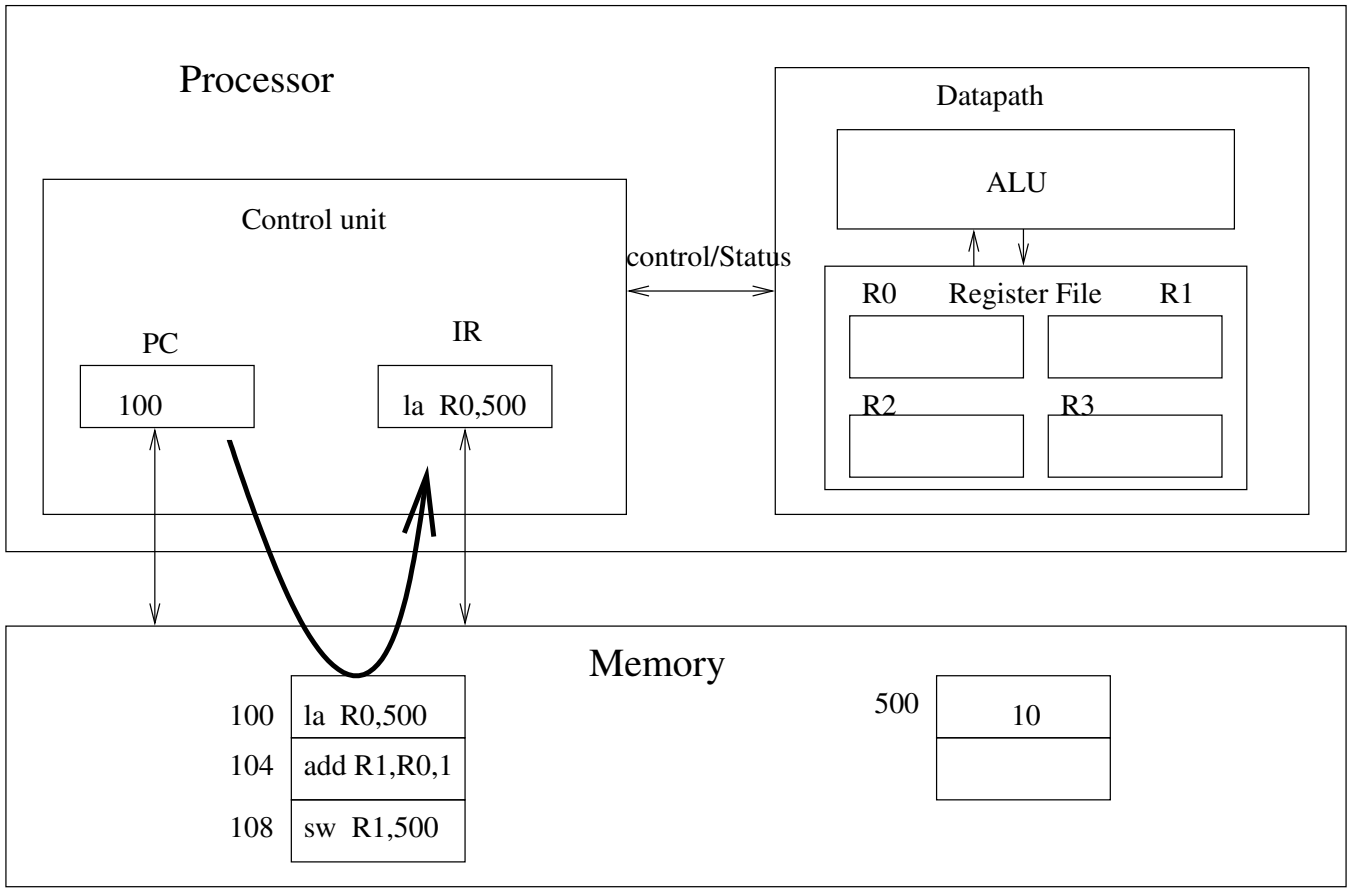


Counting CPI for non-pipelined architecture

- CPI= Cycle per instruction
- 5 cycles for executing on instruction
- $\Rightarrow$ 15 cycles for 3 instructions.

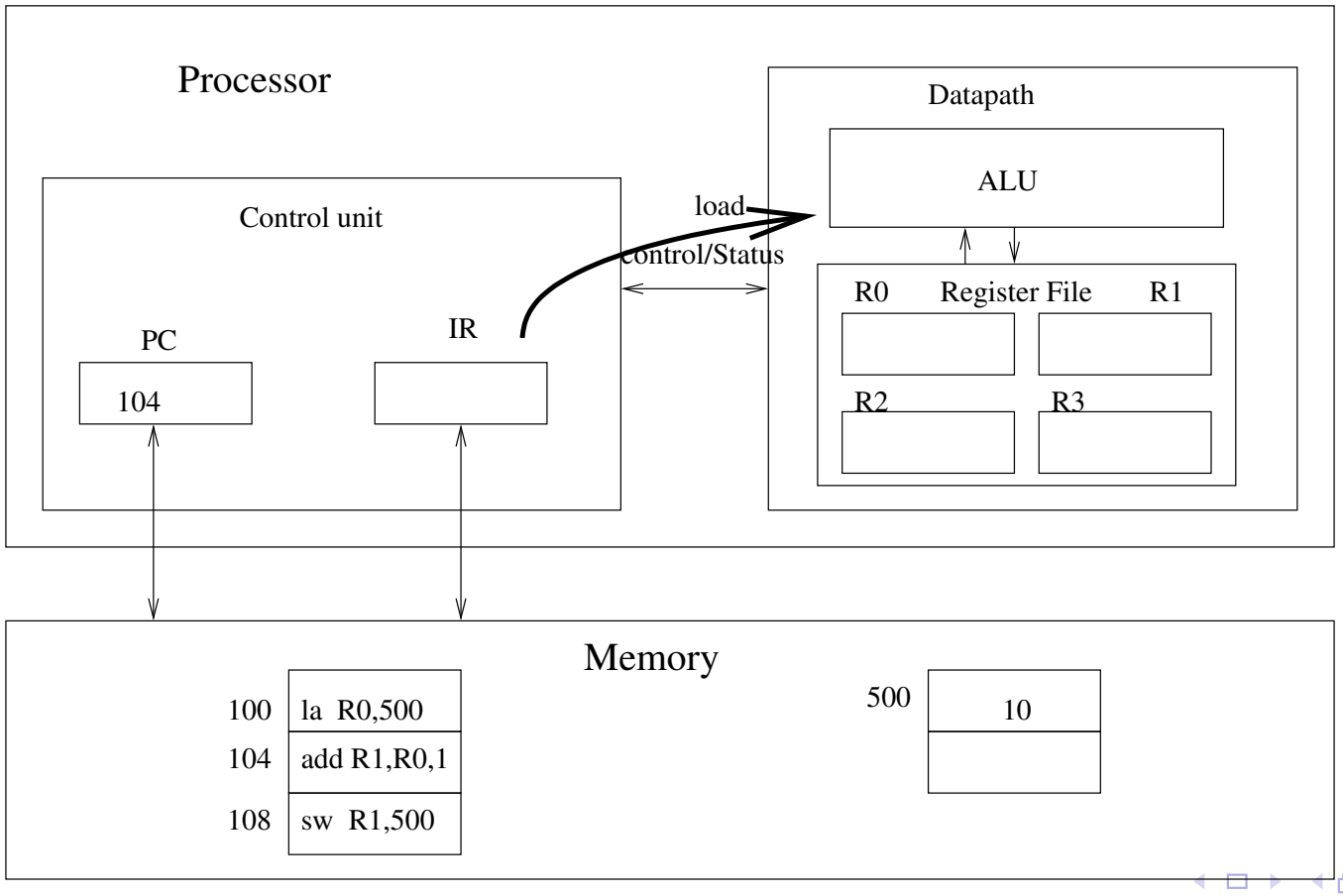
Example of pipelined execution

- Instruction Fetch (for 'load' instruction)



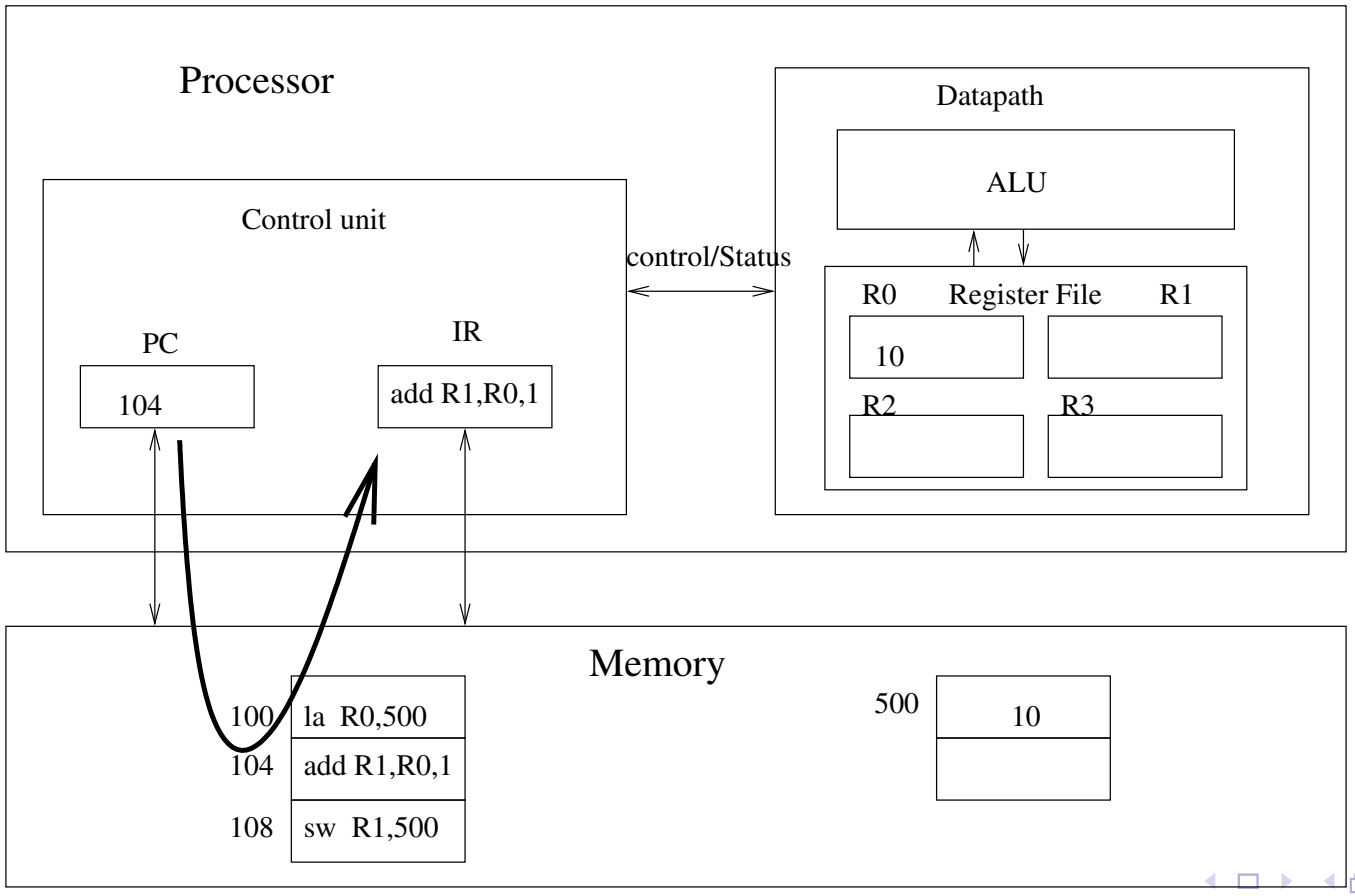
cycle 2

- Instruction Decode (for load)
- Instruction Fetch (for 'nothing' because of a bubble: instruction 'add' delayed)



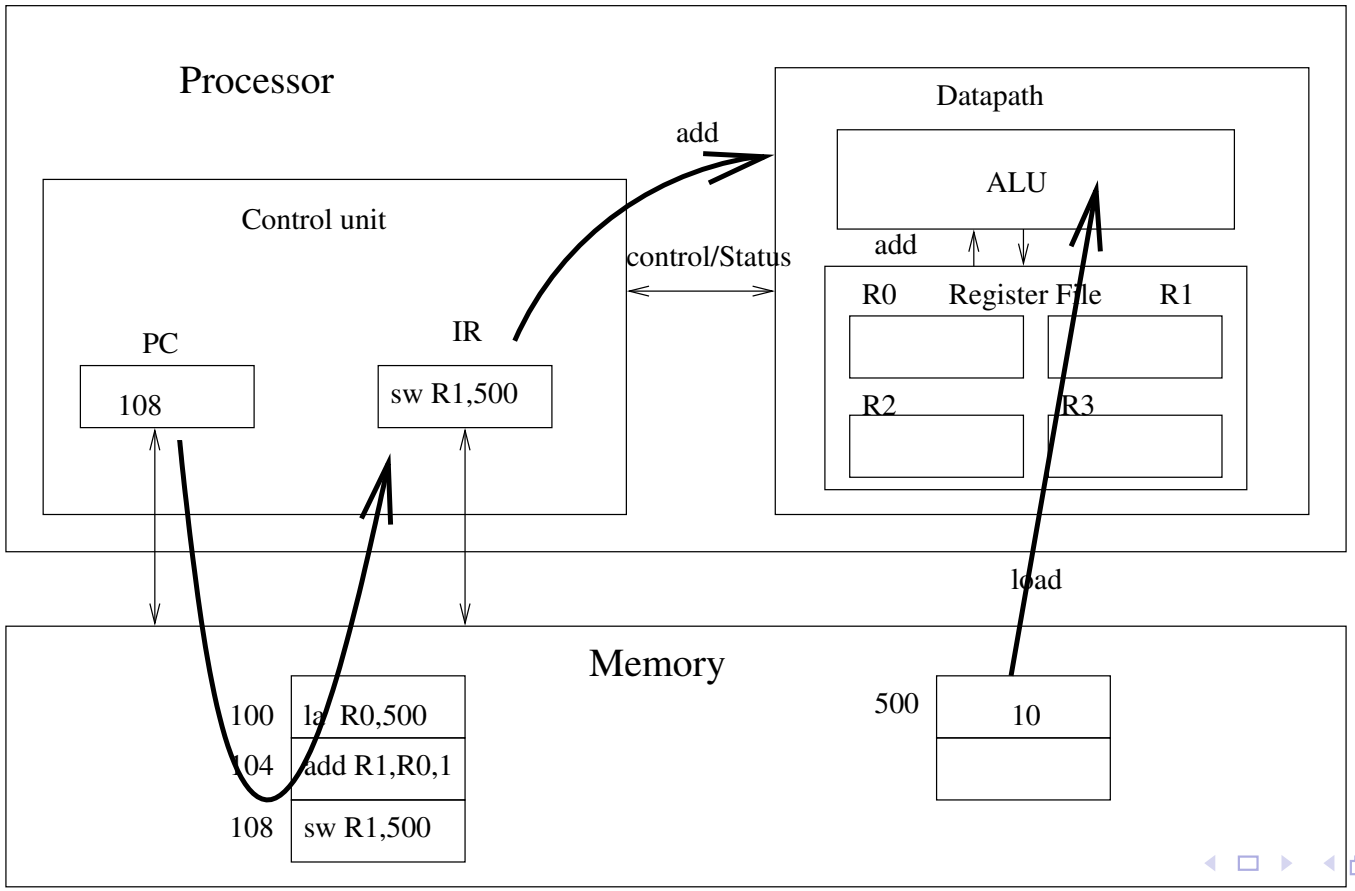
cycle 3

- Execute (for load: nothing to do)
- Instruction Decode (for 'nothing')
- Instruction fetch (for 'add')



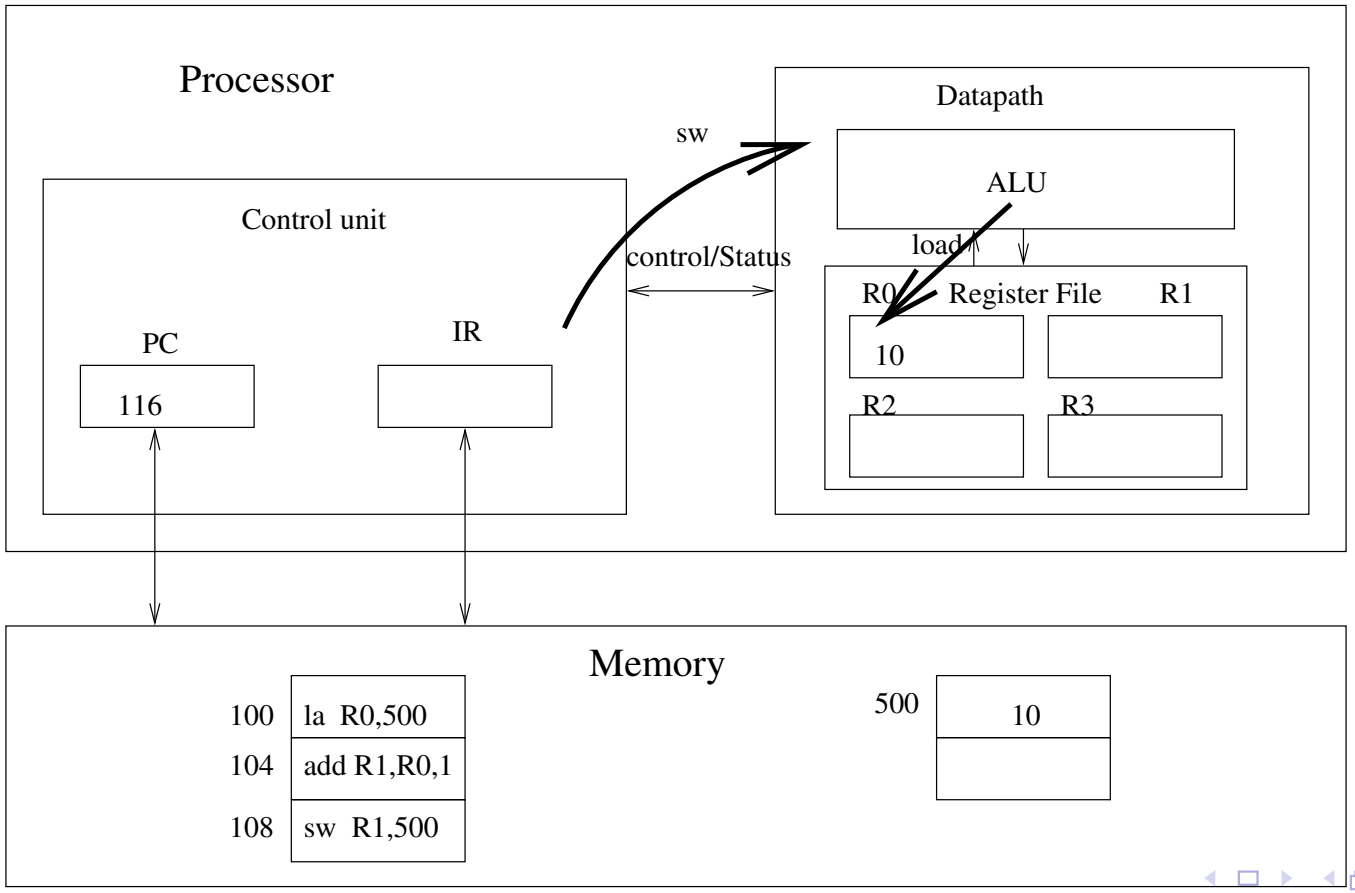
cycle 4

- Memory access (for load)
- Execute (for 'nothing')
- Instruction Decode (for add)
- Instruction fetch (for store)



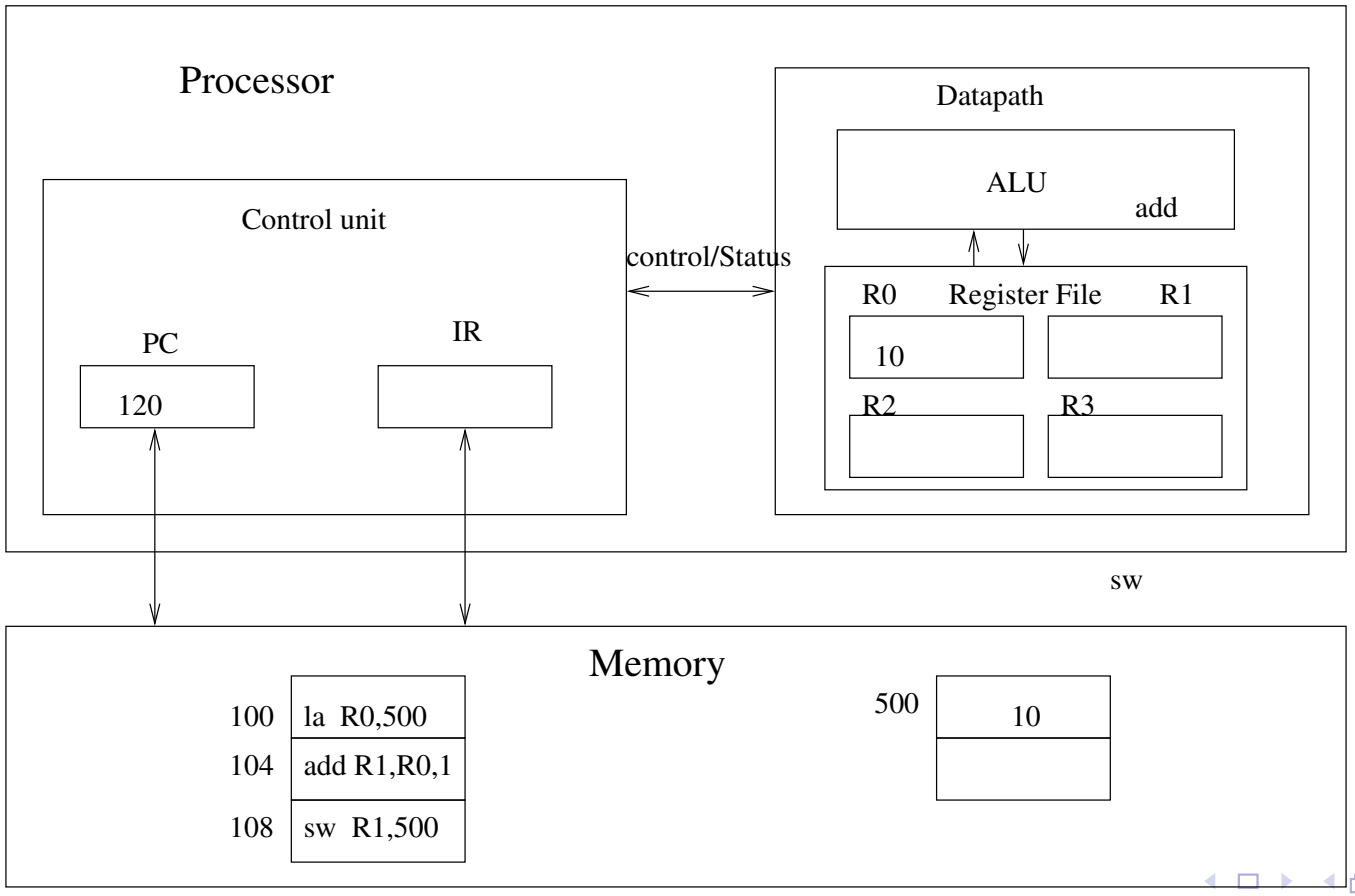
cycle 5

- Write Back (instruction load)
- Memory access (for 'nothing')
- Execute (instruction add: bypass)
- Instruction Decode store



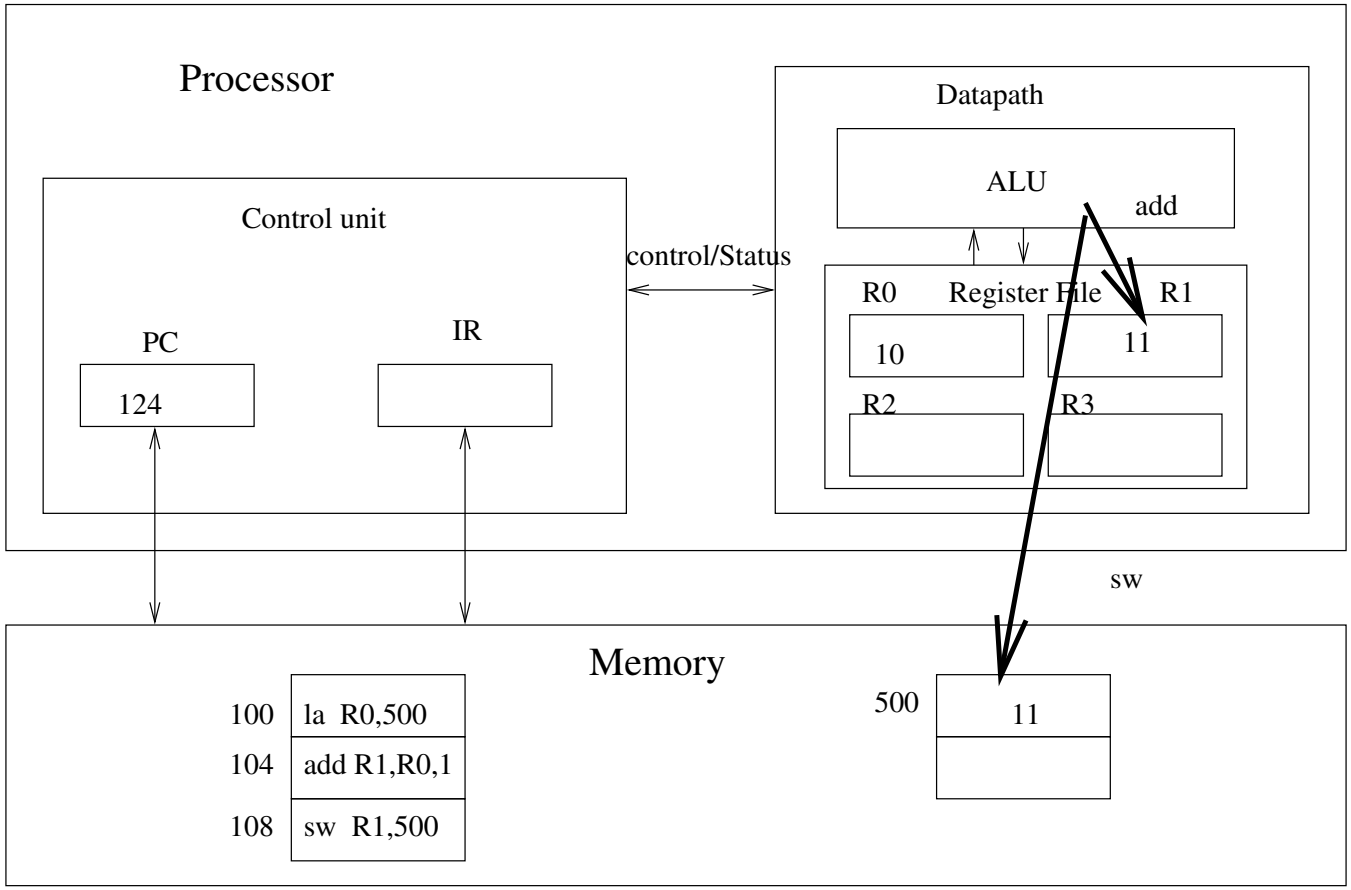
cycle 6

- Write Back (for 'nothing')
- Memory access (instruction add, nothing to do)
- Execute (instruction store: nothing to do)



cycle 7

- Write Back (instruction add)
- Memory access (instruction store: bypass)



Counting CPI for both architectures

- Non-pipelined architecture:
  - 5 cycles for one instruction
  - $\Rightarrow$ 15 cycles for 3 instructions.
- Pipelined architecture:
  - 5 cycles for one instruction
  - 8 cycles for 3 instructions.
  - $\Rightarrow$ without bubbles, one instruction per cycle
  - A 'jump' instruction interrupt the pipeline (need to wait for the address decoding to fetch next instruction) $\Rightarrow$ *pipeline stall*
  - Some ISA allow to use these *delay slots*: one or two instruction *after* the jump are executed before the jump occurs.

Table of Contents

Set Architecture Statement (ISA)

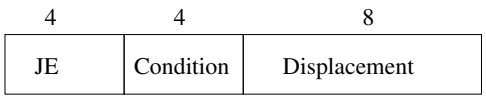
- The *instruction set* (Set Architecture statement: ISA) is of paramount importance
  - It determines the basic instructions executed by the CPU.
  - It's a balance between the hardware complexity of the CPU and the ability to express the required actions
  - It is represented in a symbolic way: the assembly code/language (ex: ADD R1,R2)
  - The tool that translates symbolic assembly code in binary code (i.e. machine code) is also called the **assembler**
- Two types of ISA:
  - CISC: Complex Instruction Set Computer
  - RISC: Reduce Instruction Set Computer

CISC: Complex Instruction Set Computer

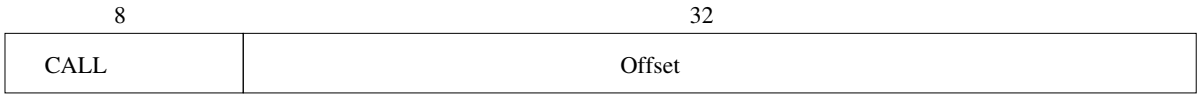
- An instruction can code several elementary operations
  - Ex: a load, an add and a store (in memory operations)
  - Ex: computer a linear interpolation of several values in memory
- Need a mode complex hardware (specifically hardware accelerators)
- High variability in size and execution time for different instructions
- Produce a more compact code but more complex to generate
- Vax, Motorola 68000, Intel x86/Pentium

Example: instructions ISA of Pentium

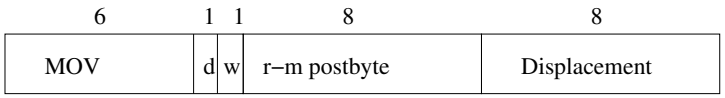
JE EIP + displacement



Call

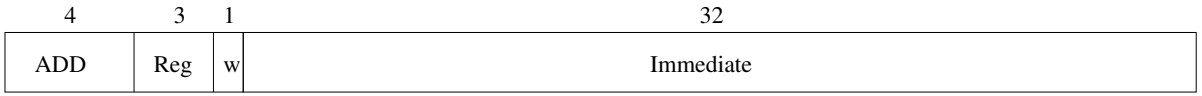


Mov $EBX, [EDI+displacement]



Push ESI

Add $EAX, Immediate



Test $EDX, Immediate

RISC: Reduced Instruction Set Computer

- Small simple instructions, all having the same size, and (almost) the same execution time.
- no complex instruction
- Clock speed increase with pipelining (between 3 and 7 pipeline stages)
- Code simpler to generate but less compact
- Every modern processor use this paradigm: SPARC, MIPS, ARM, PowerPC, etc.

Example: instructions of MSP430 ISA

1 operand instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	opcode			B/W	Ad		Dest reg.			

relative Jumps

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	condition			PC offset (10 bits)									

2 operands instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode			Dest reg.				Ad	B/W	As			Dest reg.			

Examples:

- `PUSB.B R4`
- `JNE -56`
- `ADD.W R4,R4`

Exemple of Pentium ISA

- Write a simple C program `toto.c`
- Type `gcc -S toto.c` and get the `toto.s` file
- you can also use the *compiler explorer*: `https://gcc.godbolt.org/`
-

```
main() {
  int i=17;
  i=i+42;
  printf("%d\n", i);
}
```

$\Rightarrow$

```
(... instructions ...)
movl $17, -4(%rbp)
addl $42, -4(%rbp)
(...  printf params... )
call printf
(... instrucitons ...)
```

Disassemby

- compile the assembly code: `gcc toto.s -o toto`
- disassemble with `objdump`:
  `objdump -d toto`

```
Adresses  Instructions binaires    Assembleur
(...)
 40052c: 55                        push   %rbp
 40052d: 48 89 e5                  mov    %rsp,%rbp
 400530: 48 83 ec 10               sub    $0x10,%rsp
 400534: c7 45 fc 11 00 00 00      movl   $0x11,-0x4(%rbp)
 40053b: 83 45 fc 2a               addl   $0x2a,-0x4(%rbp)
 40053f: 8b 45 fc                  mov    -0x4(%rbp),%eax
 400542: 89 c6                     mov    %eax,%esi
% (...)
```

common properties of ISA

- An ISA first defines the types of data on which the processors can compute (32 bit memory addresses, integer of various sizes, etc.)
- Then it contains various types of instructions:
  - Computation instructions (add, sub, or, and, ...), with various number of operands
  - Memory addessing instructions (load, store)
  - stack management instructions (push, pop)
  - Flow control instructions (jumps)
  - subroutine calls

ISA computation instruction

- Instruction dedicated to computations: arithmetics, logical, shift etc.
- Computation instruction operate on registers (if 16 registers are available, 4 bits are sufficient to identify them).
- let's take the example of addition
  - Possible mnemonic for addition:
    `add R0 R1` $\rightarrow$ `R3`
  - in general `op Rx, Ry` $\rightarrow$ `Rd`, where op can be `add`, `mul`, `sub`, `or`, `and` etc.
  - Operands can be values instead of registers. In general the instruction name changes: `addi R0, #4` $\rightarrow$ `R3`.
  - How to code the instruction in binary:
    - 3 registers to name $\Rightarrow 3 \times 4 = 12$ bits
    - A given number of bits to code the operation (8bits: 256 operations)
    - Some bits left for coding constants if needed

Number of operand

We have defined a so-called three-operands (*code trois addresses*) addition since the three operands used. There are other solutions :

- 2-operands instructions: `op Rd, Rx` $\rightarrow$ `Rd` (*assembleur à deux addresses*). One of the operands is overwritten.
- 1-operand instructions: same idea, but Rd is fixed and implicit. It's usually called the accumulator.
- Stack based instructions (0 operand): the processor has a stack (like calculators HP), and an add statement takes its two operands to the top of the stack and stores the result.

Memory addressing operation

- Basic read/write:
  - `Read [R2]` $\rightarrow$ `R5`
    reads the content of address contained in R2, place the result in R5
  - conversely: `Write R3` $\rightarrow$ `[R6]`
  - With a constant: `Read [#46]` $\rightarrow$ `R5`
- indirect addressing:
  - `Read [R2+R3]` $\rightarrow$ `R5`
    ((R5$\leftarrow$ [R2+R3])
- with auto-increment
  - `Read [R2+]` $\rightarrow$ `R5`
    (R5$\leftarrow$ [R2]; R2=R2+1)

Stack management instruction

- High-level languages use a lot of Stacks (last in, first out, used for managing the calls of procedures).
- The stack is stored in memory
- A specific register is used to access the top of stack: $SP for Stack Pointer
- instruction `Push R1` pushes R1 on stack i.e.:
  `Write R1` $\rightarrow$ `[SP]`
  `Add SP, 1` $\rightarrow$ `SP`
- SP+1 means "add to SP the size of the word" (32 or 64 bits)
- Instruction `Pop R1` will do the opposite: `Read [SP]` $\rightarrow$ `R1`
  `Sub SP, 1` $\rightarrow$ `SP`

Flow control instructions

- The main flow control instruction (to implement loops and ifs) are the jumps:
  - Relative Jumps
    - `GoForward #123` (jump to current address+123)
    - `GoBackward #123`
    - Warning: less than 32 bits for the constant!
  - Absolute jumps
    - `Jump #1234` (jump to address 1234)
  - Conditional jumps
    - `GoForward #123 IfPositive` (i.e. if last operation's result was positive)
    - necessitates the presence of flags
    - Usually at four flags: C (carry), N (negative), Z (Zero), V (overflow)
    - In some ISA, flags are replaced by predicates (any condition can be used as flag).

Subroutine Calls

- The call (i.e. jump at the first instruction) of a procedure is done with the instruction `call`
  `call Label`
- The label is a symbolic way to indicate where is stored the code of the procedure
- The `call` instruction performs a `jump` and keep tracks of the current address in order to be able to come back here after the execution of the instruction `Return` (usually use the stack for that)
- The `Return` instruction does not returns values (such as the result of functions),
- Results and parameters of functions are transmitted either on the stack or in specific register. This is defined by the ABI (Application Binary Interface).
- The ABI allow functions produced from different compiler to call themselves (i.e. library)

Interrupts (ISR for interrupt service routine)

- The instruction flow can be interrupted at any time by an interrupt:
  - Packet arrived on network card
  - Sound card required more samples to play
  - a character has been stroked on the keyboard
  - etc.
- Interrupt are almost equivalent of function calls:
  - They interrupt the current instruction flow to execute the interrupt handler
  - Then they continue the execution where is has been stopped
  - Ideally they have no impact on the function being executed, hence all registers are saved on the stack before jumping to interrupt handler
- Interrupts and Interrupts handler will be studied in more details in CRO course

Other ISA instructions

- Change mode instructions
  - modes interrupt, user, supervisor etc.
- Synchronization instructions
  - for multi-core systems

ISA Example: MSP430

- MSP430 ISA is an instruction set for micro-controller very low consumption (smart cards, RFID tags).
- 16-bit RISC instruction set with 16-bit registers .
- There are instructions of one and two operands.
- If these operands are registers, the instruction holds in one word of 16 bits.
- The operands can also be a memory box whose address (on 16 bits) is coded in one of the words following the instruction. In this case the instructions are 32 or 48 bits.
- Some registers are special: the $PC is R0,
- the flags are in the register R1, and the register R2 can take different values useful constants (0, 1, -1 ...).
- the number of cycles that each instruction takes is exactly equal to the number of memory accesses that it makes.

Example: instructions of MSP430 ISA

1 operand instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	opcode			B/W	Ad		Dest reg.			

relative Jumps

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	condition			PC offset (10 bits)									

2 operands instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode			Dest reg.				Ad	B/W	As			Dest reg.			

Exemples:

- `PUSB.B R4`
- `JNE -56`
- `ADD.W R4,R4`

ISA example: ARM

- ARM ISA is a 32-bit RISC instruction set with 16 registers found among other things in all mobile phones.
- All the instructions are coded in exactly one memory word (32 bits).
- The instructions are 4 operands: the fourth is an offset that can be applied to the second operand.
- The second operand, and the offset can be constants. For example, `add R1, R0, R0, LSL #4` calculates in `R1` the multiplication of `R0` by 17, without using the multiplier (slow, and sometimes otherwise absent).
- For embedded systems, ARM produced the THUMB ISA (instructions with 2 operands on 16 bits)
- In recent ARM system, instruction of 16 and 32 bits can be mixed