Course on: “Advanced Computer Architectures”

Pipelining: Basic Concepts

Prof. Cristina Silvano
Politecnico di Milano
email: cristina.silvano@polimi.it

Reduced Instruction Set of RISC-V Processor

- **ALU instructions:**

```
add rd, rs1, rs2          # $rd ← $rs1 + $rs2
addi rd, rs1, 4           # $rd ← rs1 + 4
```

- **Load/store instructions:**

```
ld rd,  offset (rs1)      # rd ← M[rs1+offset]
sd rs2, offset (rs1)      # M[rs1+offset] ← rs2
```

- **Branch instructions** to control the instruction flow:
  - **Conditional branches:** the branch is taken only if the condition is true.
    Examples: **`beq`** (*branch on equal*) and **`bne`** (*branch on not equal*)

```
beq rs1, rs2, L1   # go to L1 if (rs1 == rs2)
bne rs1, rs2, L1   # go to L1 if (rs1 != rs2)
```

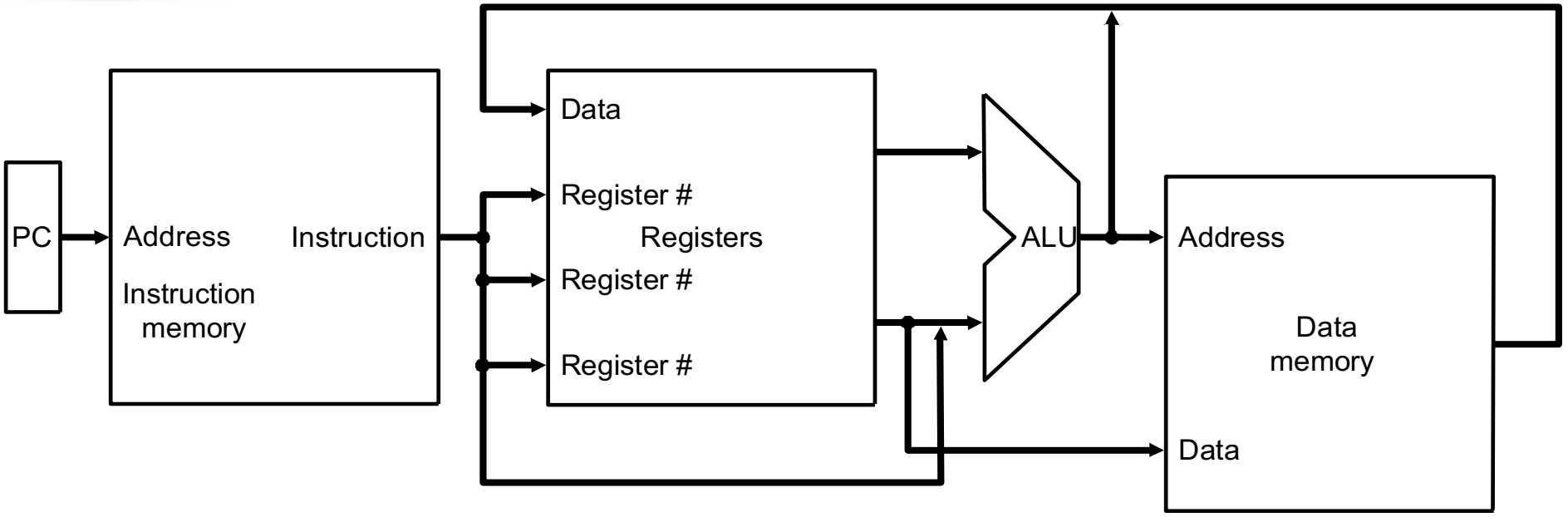
  - **Unconditional jumps:** the branch is always taken.
    Examples: **`j`** *(jump)* and **`jr`** *(jump register)*

```
j   L1             # go to L1
jr  ra             # go to add. contained in ra
```

Phases of execution of RISC-V Instructions

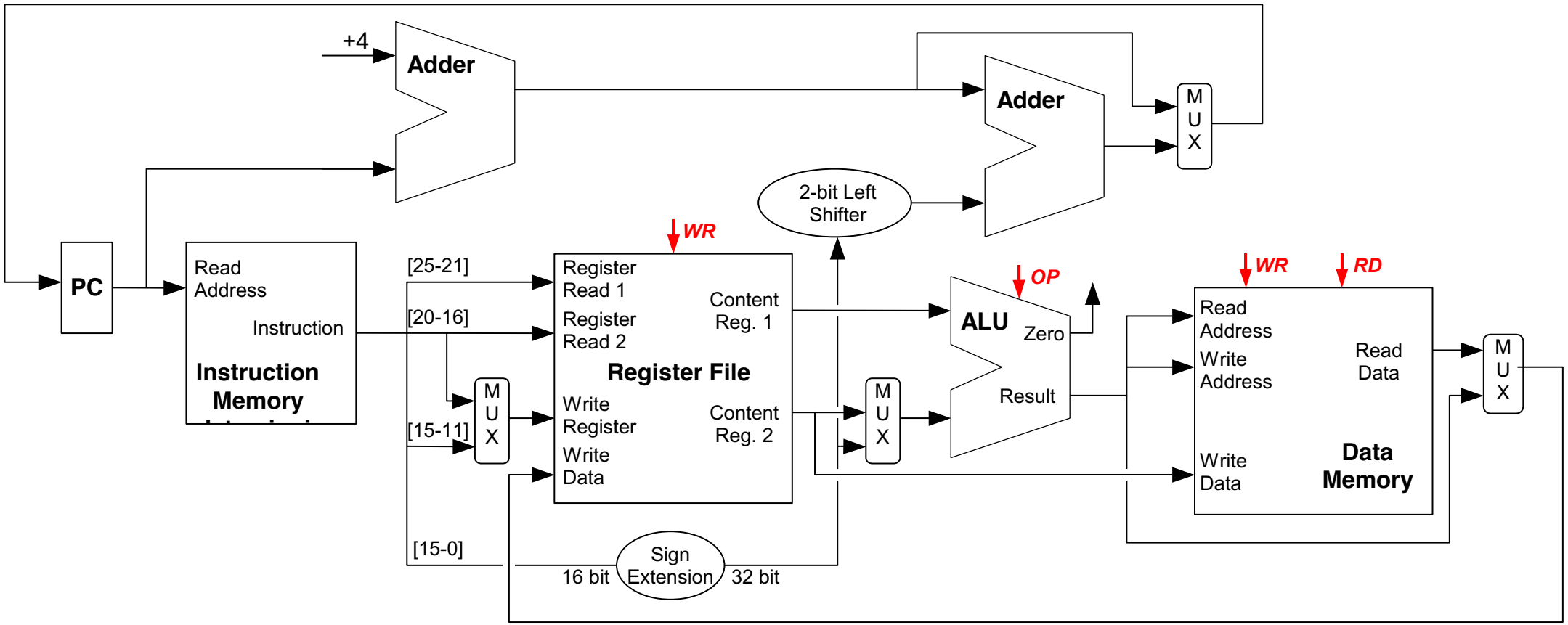
IF Instruction Fetch	ID Instruction Decode	EX Execution	ME Memory Access	WB Write Back

Basic Implementation of RISC-V data path



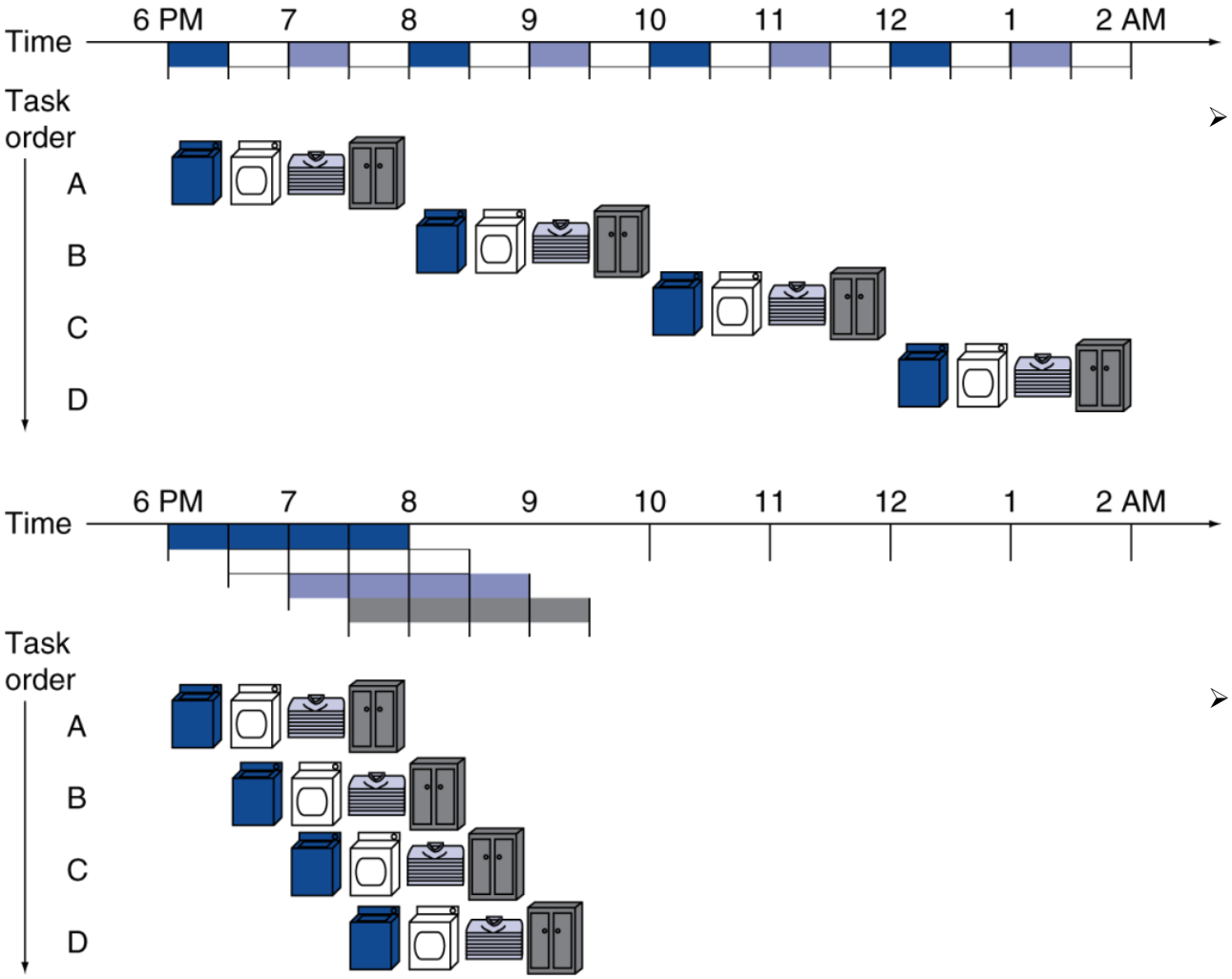
- **Instruction Memory** (read-only memory) separated from **Data Memory**
- 32 General-Purpose Registers organized in a **Register File** (RF) with 2 read ports and 1 write port.

Implementation of RISC-V data path



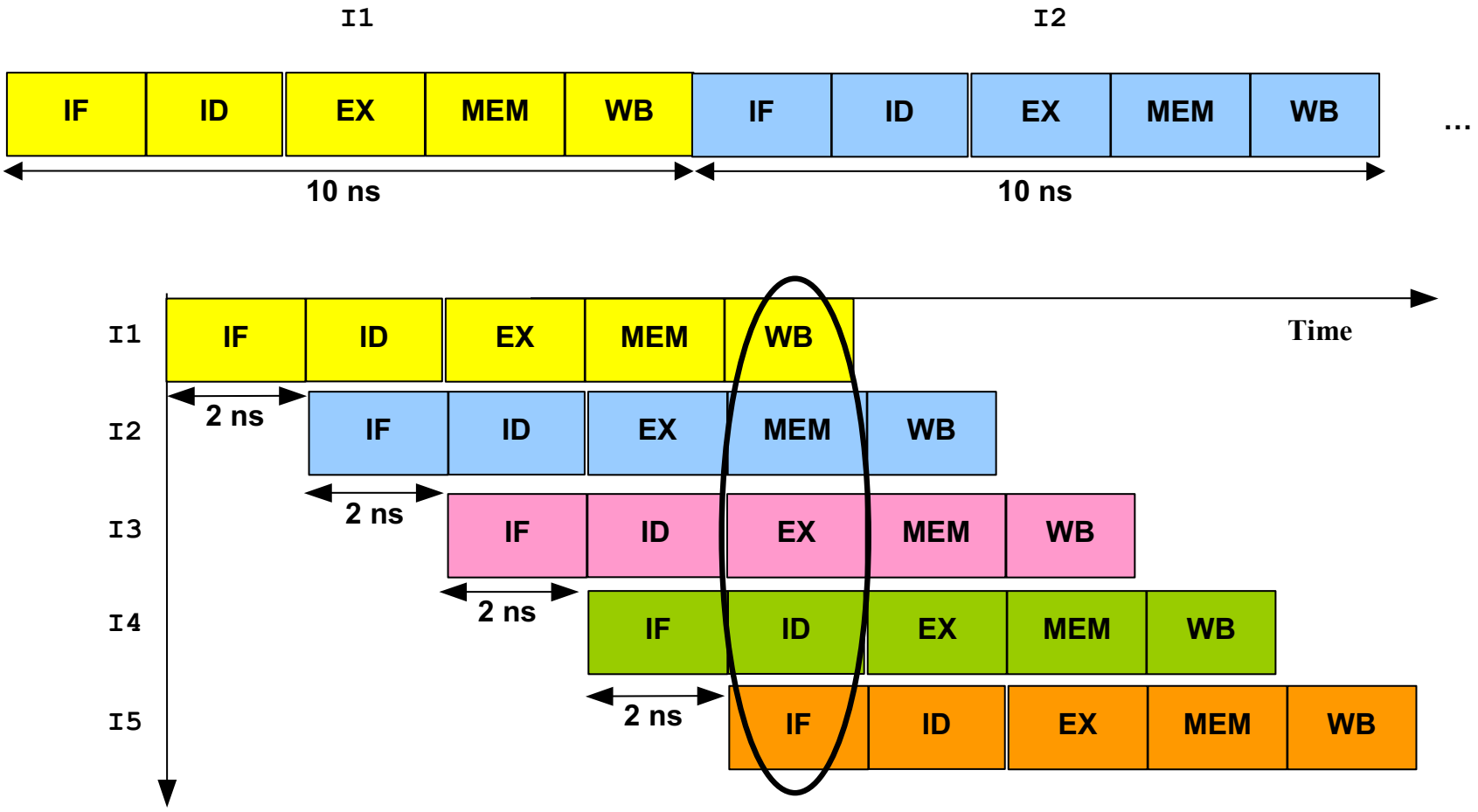
RISC-V PIPELINING

Pipelining Analogy



- Sequential laundry jobs over time
- Pipelined laundry: overlapping execution of stages to improve throughput: number of jobs executed per hour

Sequential vs. Pipelining Execution



Pipelining

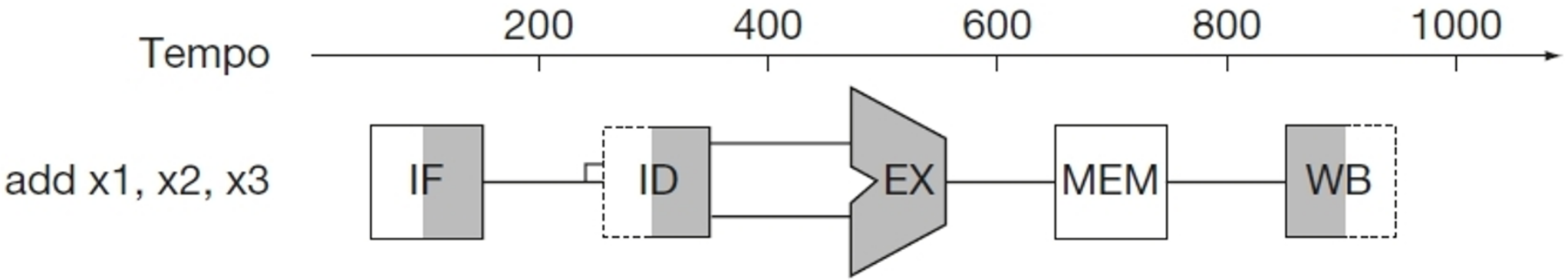
- The time to advance the instruction of one stage in the pipeline corresponds to a clock cycle.
- The pipeline stages must be **synchronized:** the duration of a clock cycle is defined by the time requested by the slower stage of the pipeline *(i.e. 2 ns)*.
- The goal is to **balance** the length of 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 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)*

Pipelined Execution of RISC-V Instructions

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



Pipeline Execution of RISC-V 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`

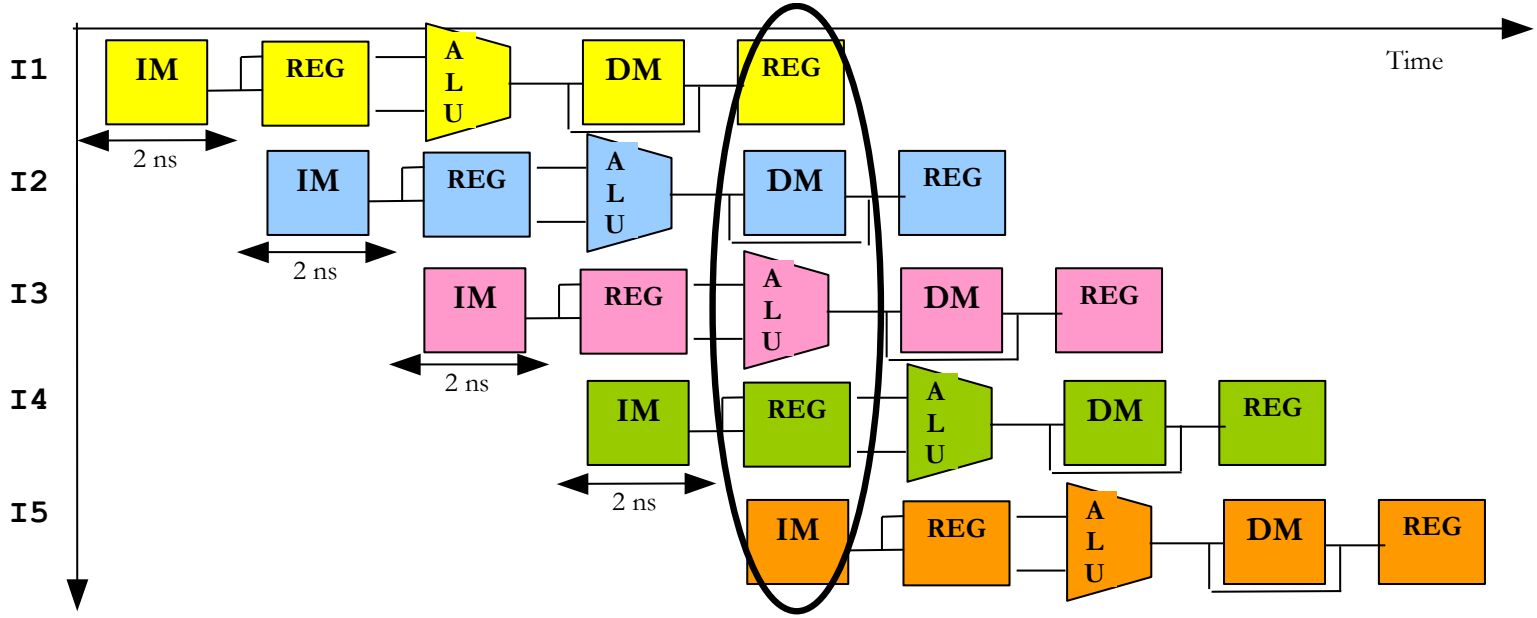
Store Instructions: `sw $x,offset($y)`

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	

Resources used during the pipeline execution



IM = Instruction Memory

REG = Register File

DM = Data Memory

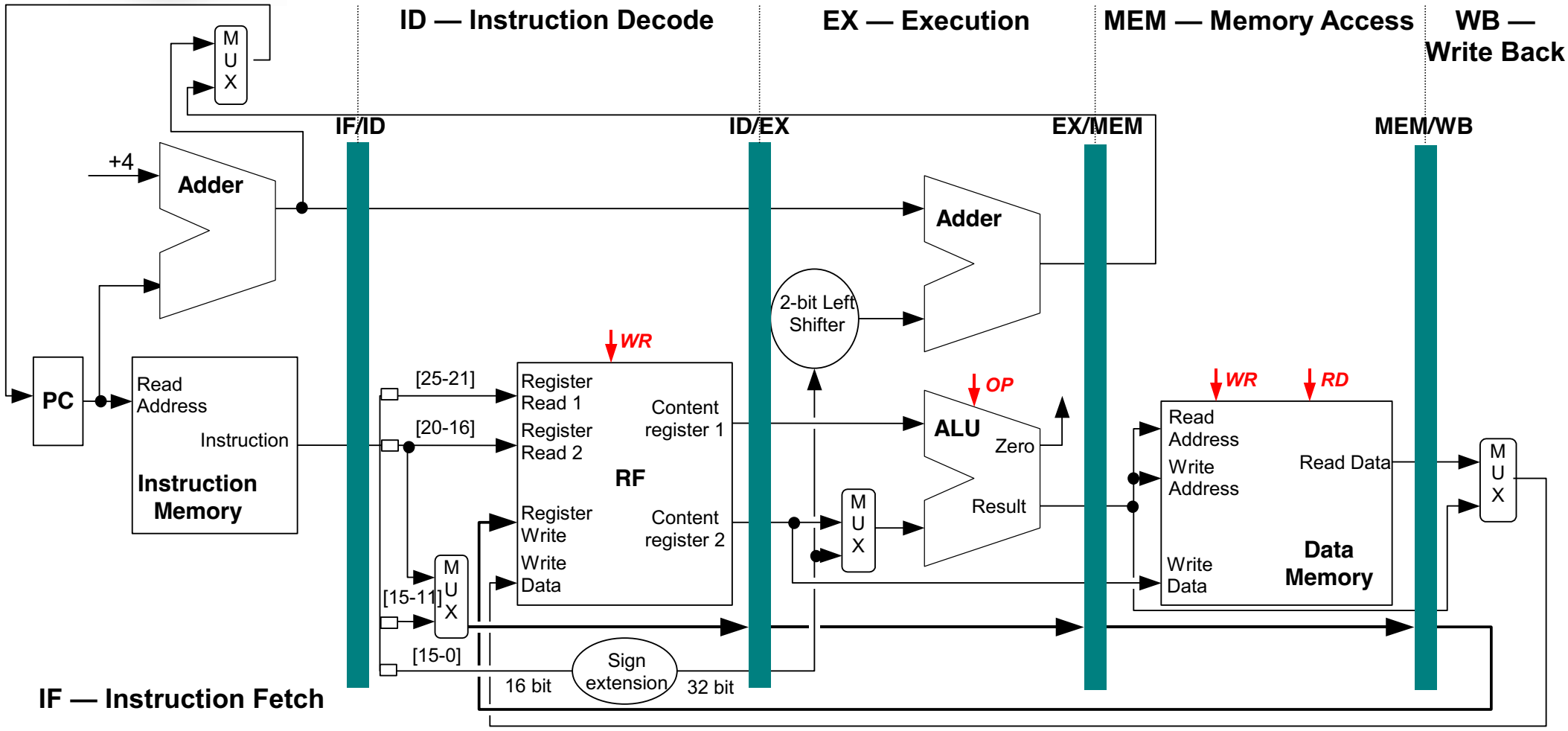
Implementation of RISC-V pipeline

- The division of the execution of each instruction in 5 stages implies that in each clock cycle there are 5 instructions in execution.

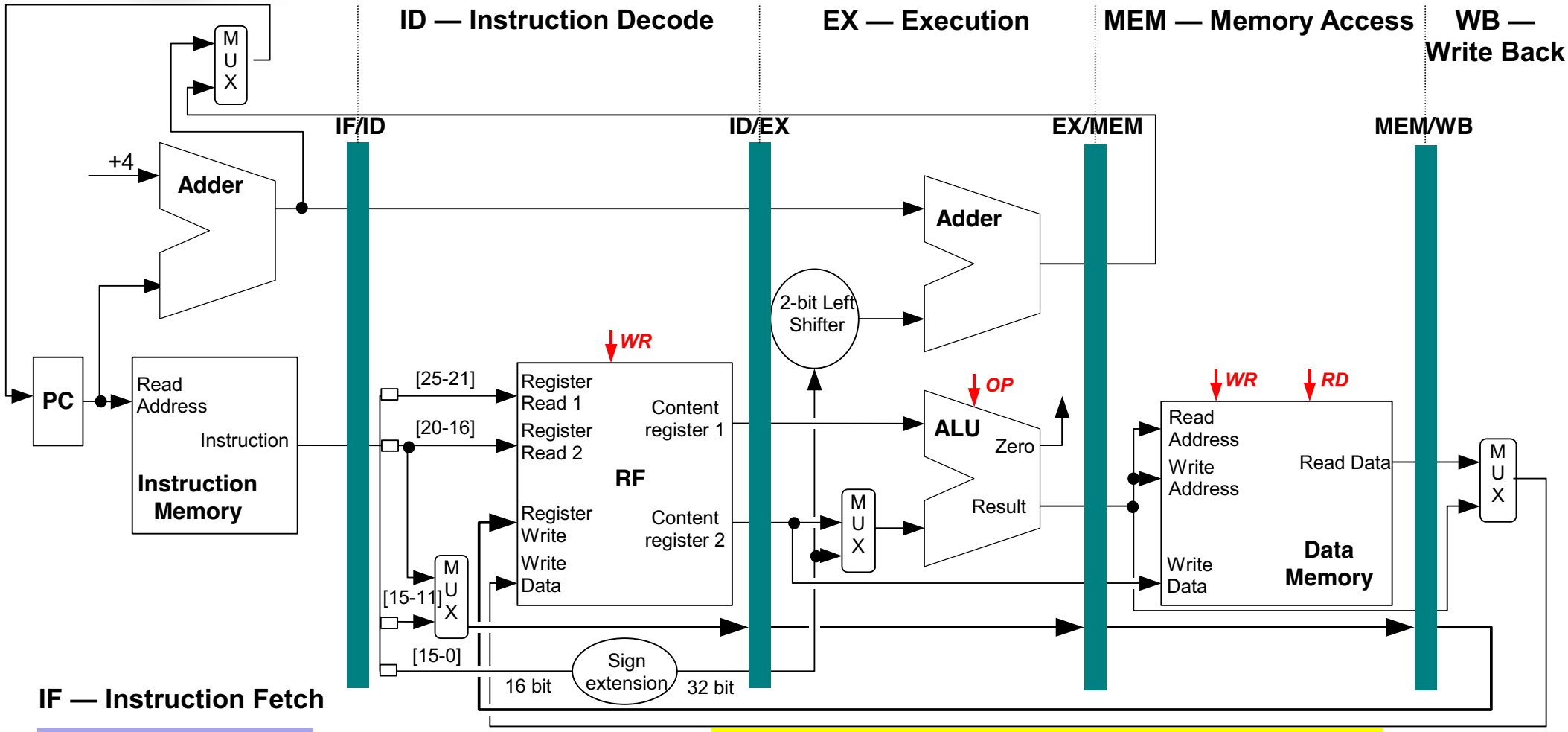
  $\Rightarrow$ the implementation of pipelined *CPU* with 5 stages must be composed of 5 modules corresponding to 5 execution stages

  $\Rightarrow$ we need **pipeline registers** to separate the different stages.

Structure of RISC-V pipeline



Structure of RISC-V pipeline

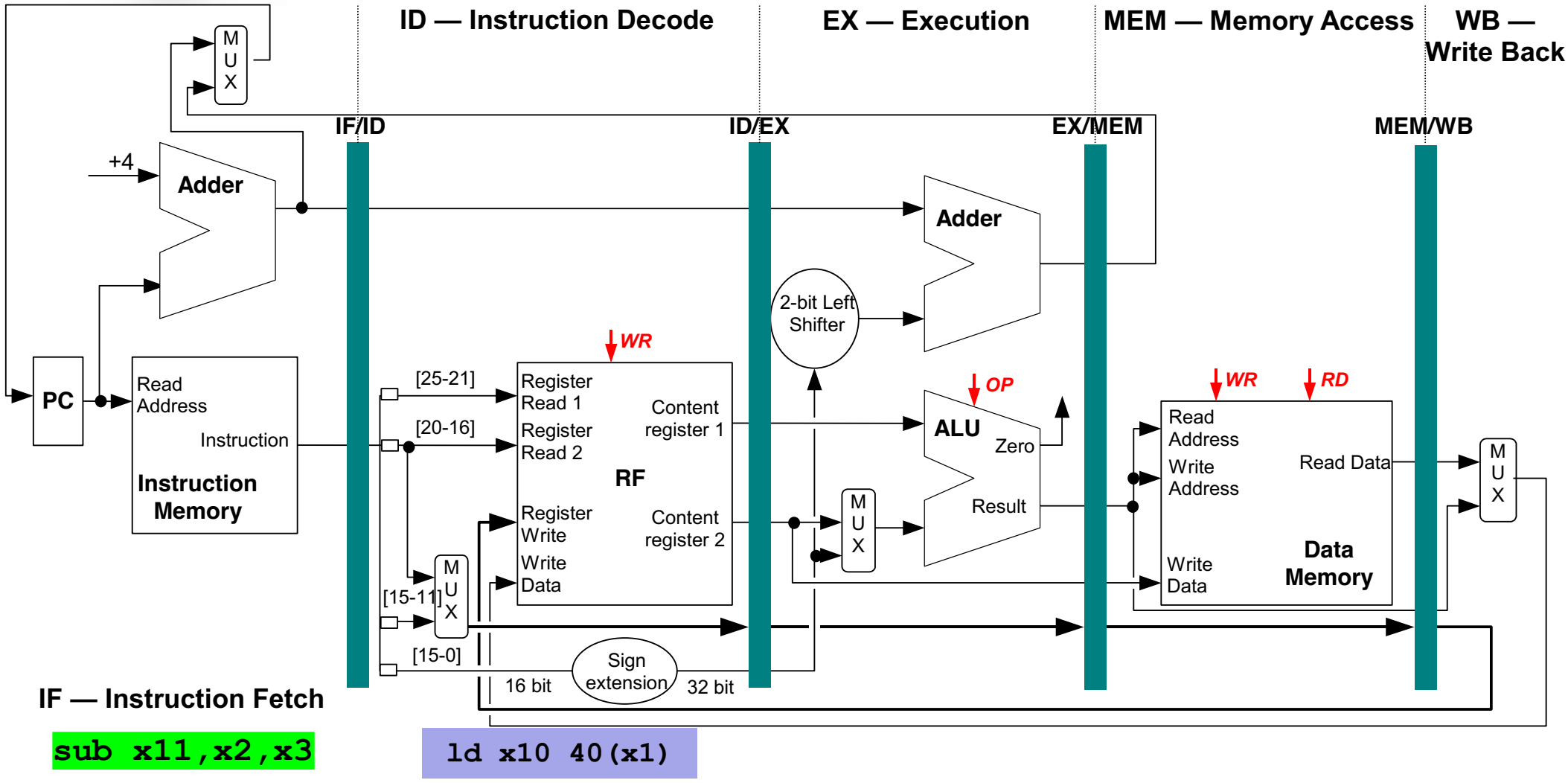


IF — Instruction Fetch

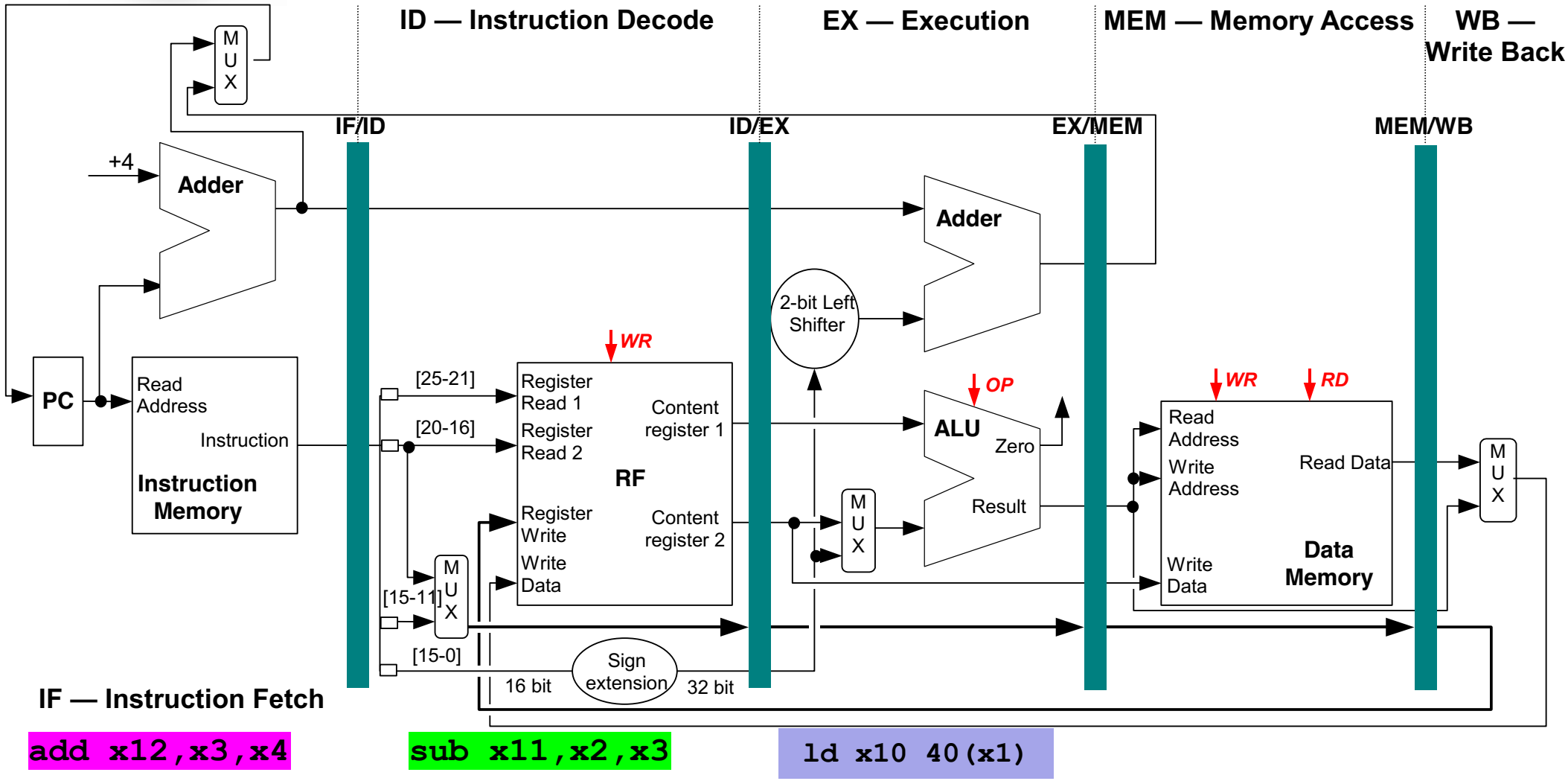
```
ld x10 40(x1)
```

An instructions flows along pipeline stages with its own data stored in the interstage registers

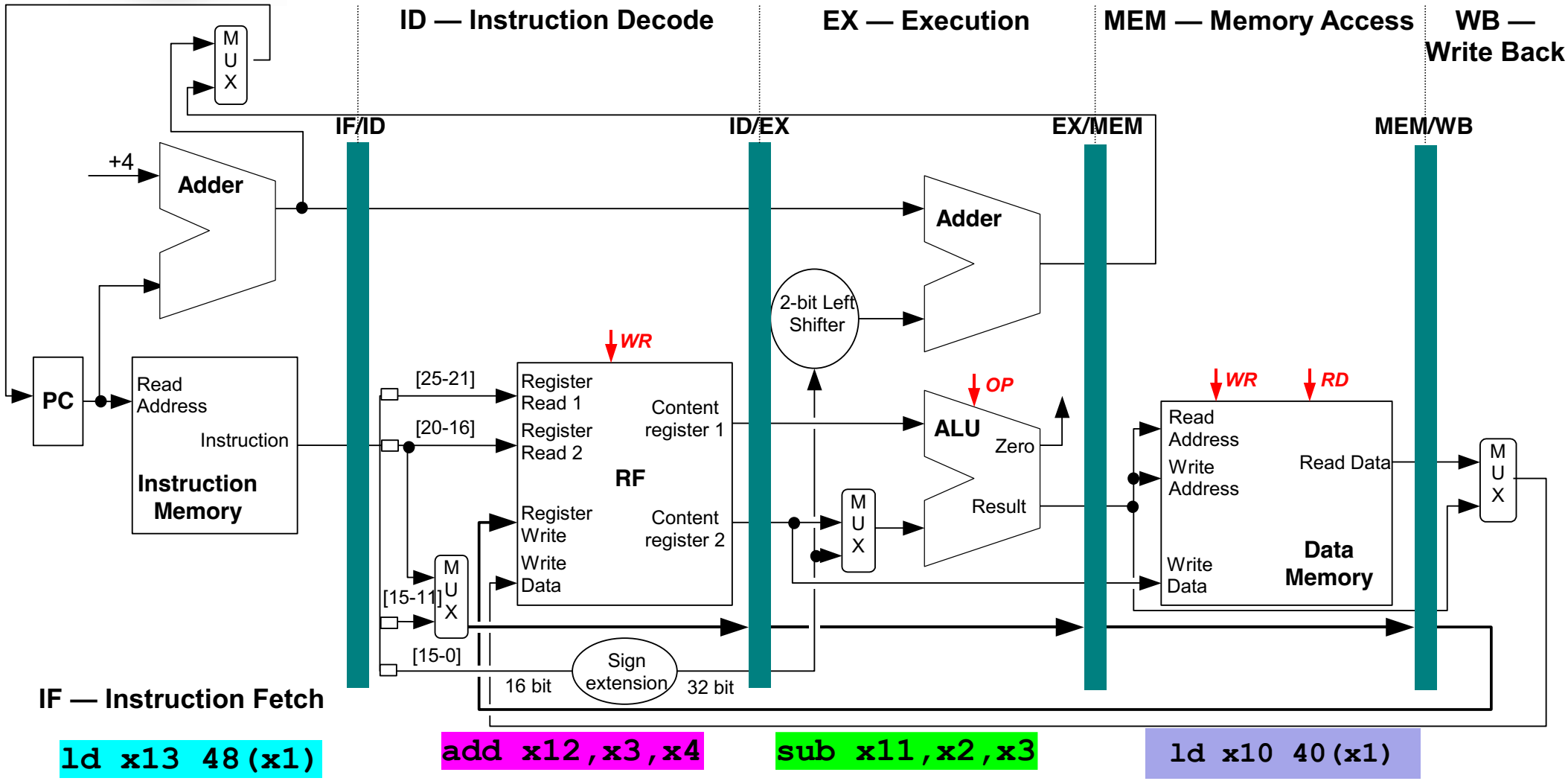
Structure of RISC-V pipeline



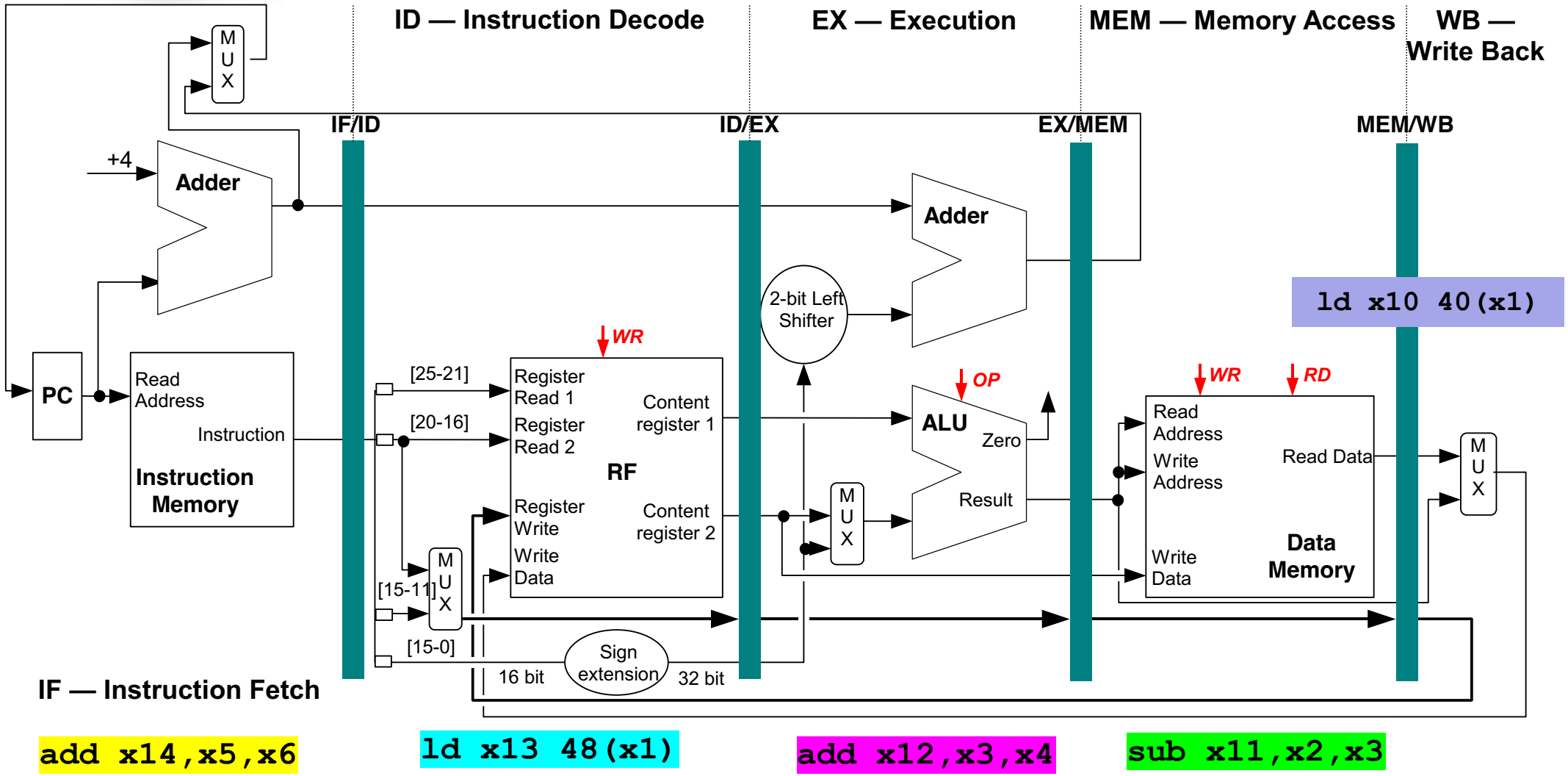
Structure of RISC-V pipeline



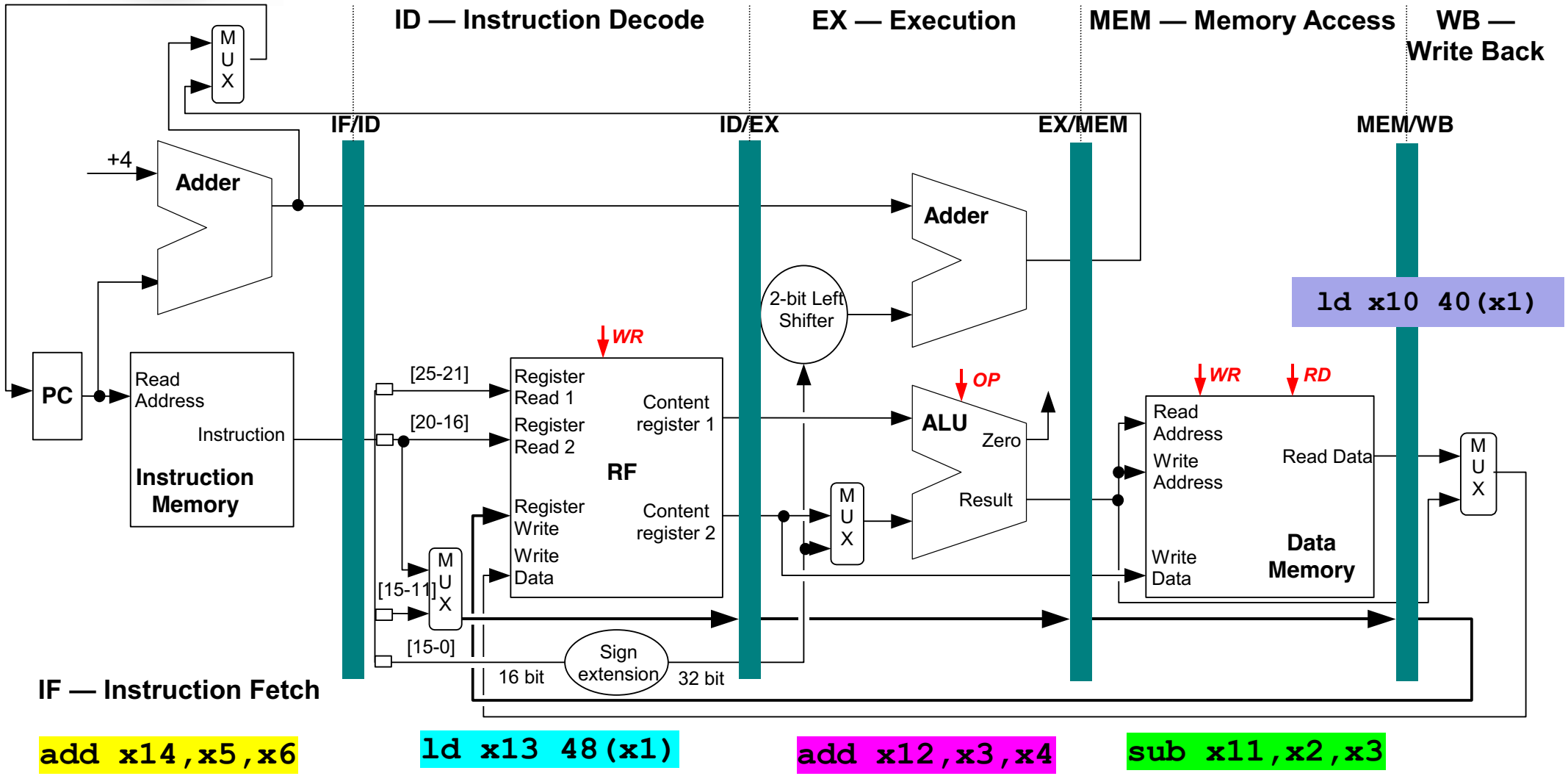
Structure of RISC-V pipeline

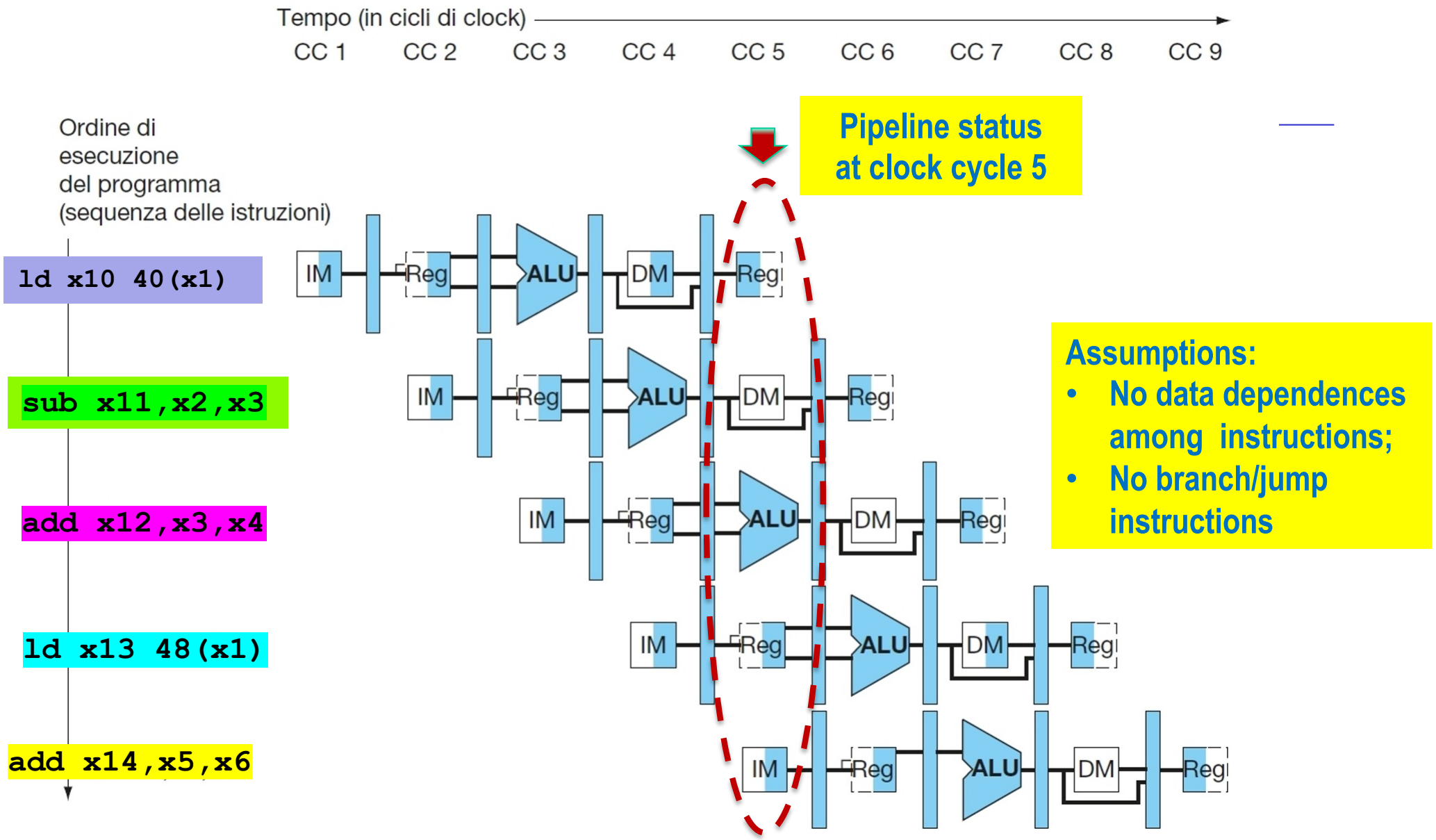


Structure of RISC-V pipeline



- In a clock cycle there are in execution 5 different instructions!
- Data stored in interstage registers correspond to different instructions!




Tempo (in cicli di clock)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
Ordine di esecuzione del programma (sequenza delle istruzioni)
ld x10 40(x1)
sub x11,x2,x3
add x12,x3,x4
ld x13 48(x1)
add x14,x5,x6
IM
Reg
ALU
DM
Reg
Pipeline status at clock cycle 5
Assumptions:
• No data dependences among instructions;
• No branch/jump instructions

The Problem of Pipeline Hazards

The Problem of Pipeline Hazards

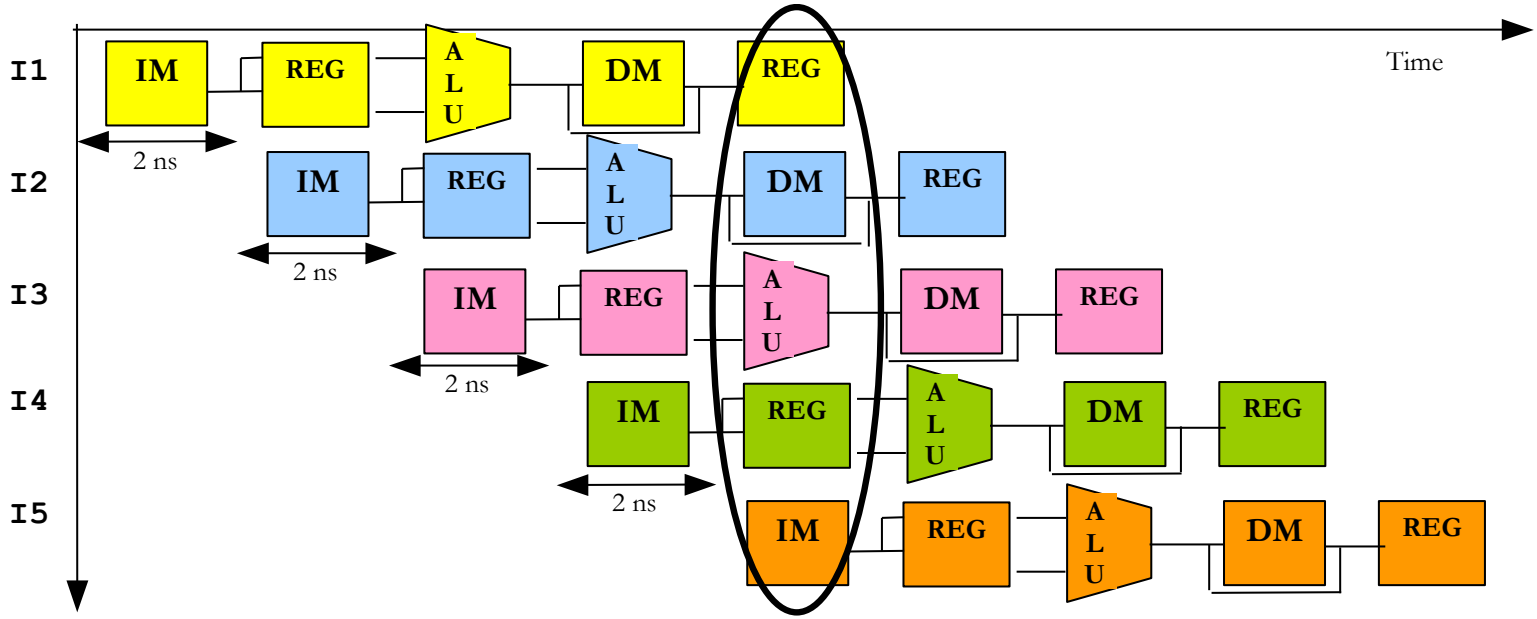
- A **hazard (conflict)** is created whenever there is a **dependence** between instructions, and instructions are close enough that the overlap caused by pipelining would change the order of access to the operands involved in the dependence.
- Hazards imposes the next instruction in the pipeline to be executed later than during its designated clock cycle.
- Hazards reduce the performance from the ideal speedup gained by pipelining.

Three Classes of Hazards

1) **Structural Hazards:** Attempt to use the same resource from different instructions simultaneously
   - Example: Single memory for instructions and data
2) **Data Hazards:** Attempt to use a result before it is ready
   - Example: Instruction depending on a result of a previous instruction still in the pipeline
3) **Control Hazards:** Attempt to make a decision on the next instruction to execute before the condition is evaluated
   - Example: Conditional branch execution
   - ***Control hazards will be studied in the next lessons***

Structural Hazards

- **No structural hazards in RISC-V 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: Analysis of dependencies

- If the instructions executed in the pipeline are **dependent to each other**, data hazards can arise when instructions are too close

- Example:

```
sub  x2, x1, x3 # reg. x2 written by sub
and x12, x2, x5 # 1° operand (x2) depends on sub
or x13, x6, x2  # 2° operand (x2) depend on sub
add x14, x2, x2 # 1° (x2) & 2° (x2) depend on sub
sw x15,100(x2)  # base reg. (x2) depends on sub
```

Data Hazards: RAW

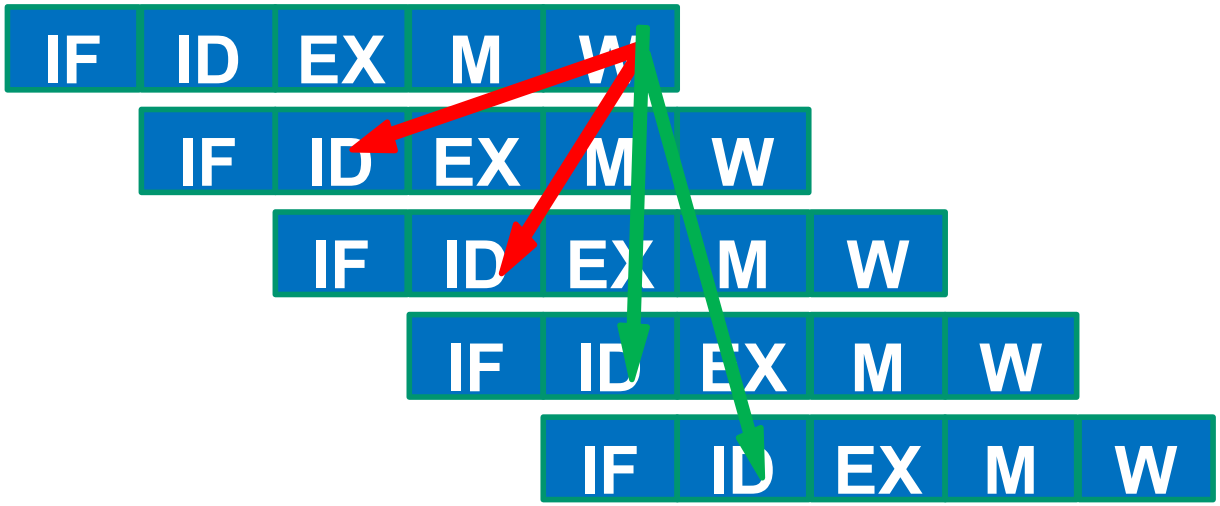
- Data hazards are **true data dependencies** and generate **RAW hazards** in the pipelines.

- **RAW (READ AFTER WRITE) hazard:** instruction *n+1* tries to read a source operand before the previous instruction *n* has written its value in the RF. For example:

```
sub x2, x1, x3      # reg. x2 is written by sub
and x12, x2, x5     # 1° op. (x2) depends on sub
```

Data Hazards: Example

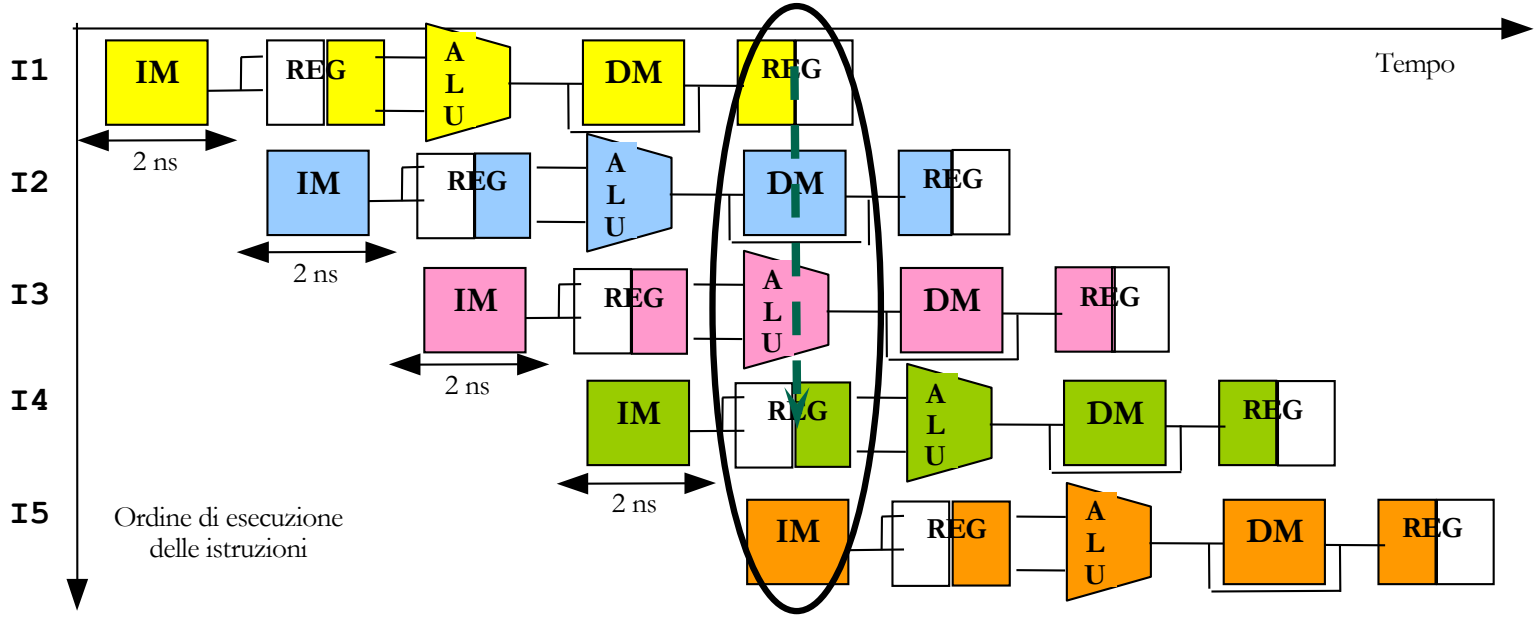
```
sub  x2, x1, x3
and x12, x2, x5
or  x13, x6, x2
add x14, x2, x2
sw  x15,100(x2)
```



RISC-V Optimized Pipeline

- Register File used in 2 stages: Read access during ID and write access during WB
- **Optimized Pipeline: we assume the RF read occurs in the second half of clock cycle and the RF write in the first half of clock cycle**
- *What happens if read and write refer to the same register in the same clock cycle?*
  - It is **not** necessary to insert any stall

Resources Used in the Optimized Pipeline

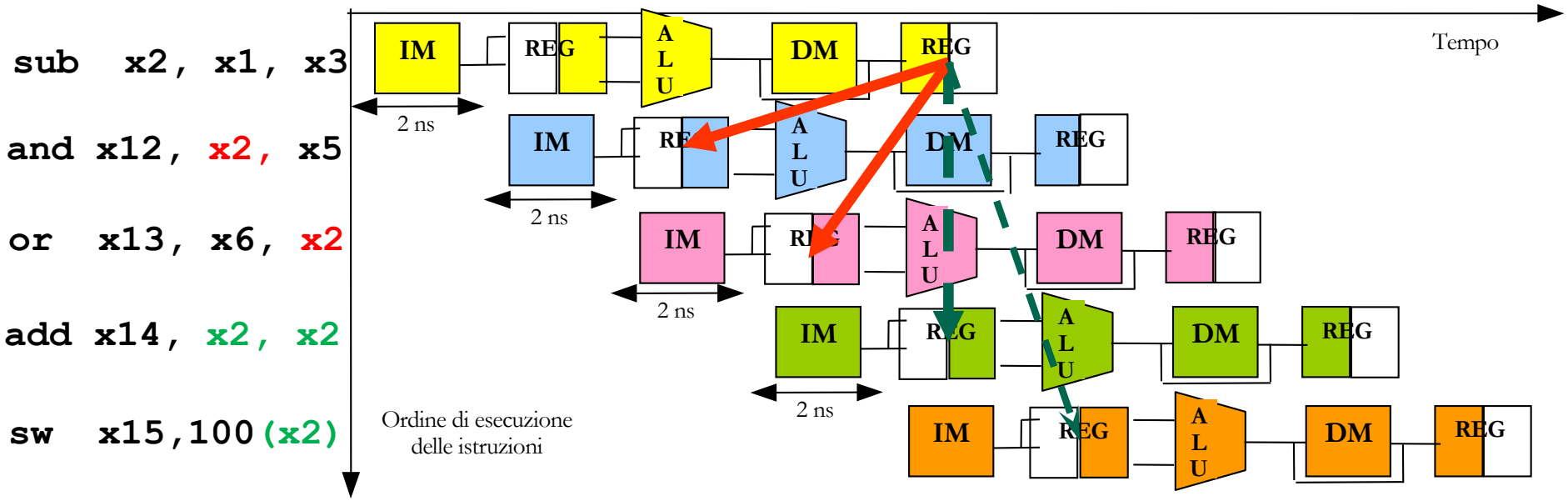


IM = Instruction Memory

REG = Register File

DM = Data Memory

Data Hazards in the Optimized Pipeline: Example



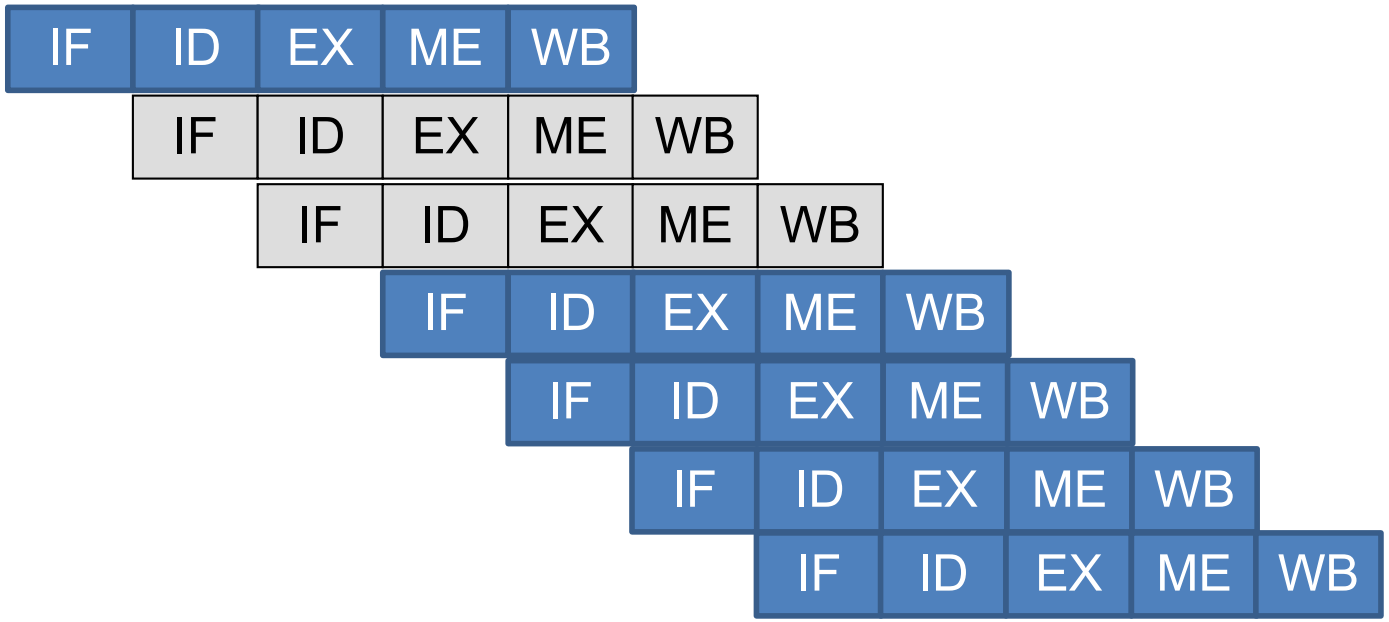
Data Hazards: Possible Solutions

- **Compilation Techniques *(static-time techniques):***
  - a) **Insertion of `nop`** (*no operation)* instructions
  - b) **Instructions scheduling** to avoid that correlating instructions are too close
    - The compiler tries to insert independent instructions among correlating instructions, otherwise insert **`nops`**.

- **Hardware Techniques *(runtime techniques):***
  - c) **Insertion of *stalls*** *or "bubbles"* in the pipeline
  - d) **Data forwarding** or bypassing

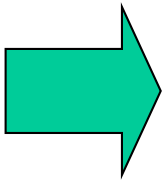
a) Insertion of `nops`: Example

```
sub  x2, x1, x3
nop
nop
and x12, x2, x5
or  x13, x6, x2
add x14, x2, x2
sd  x15,100(x2)
```



b) Scheduling: Example

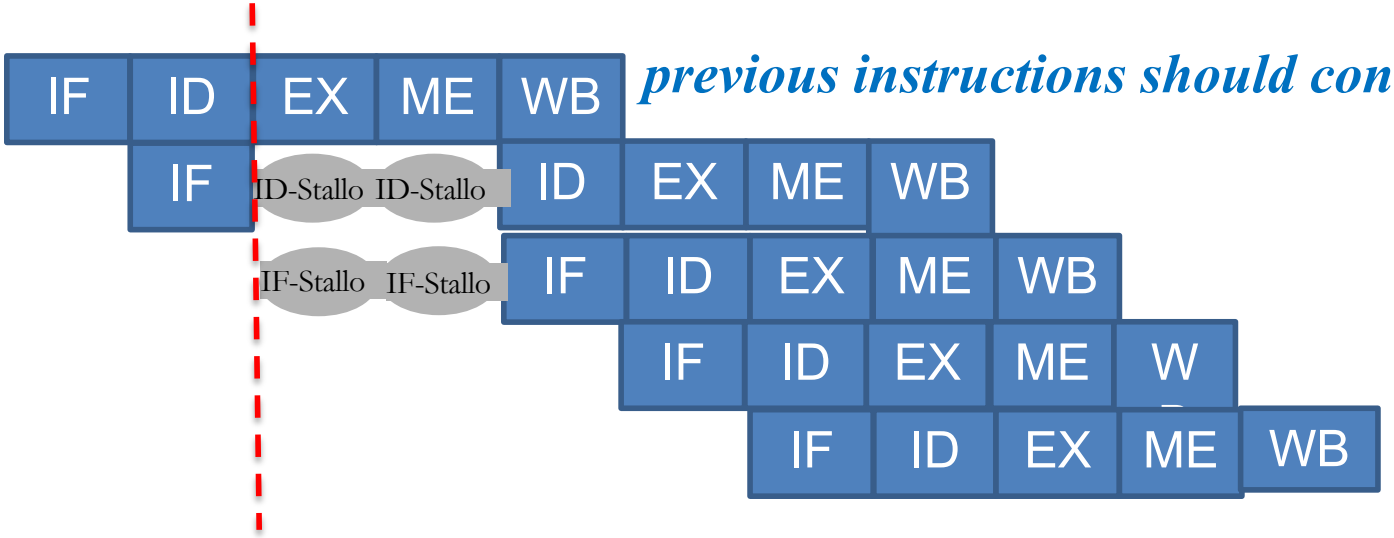
```
sub   x2, x1, x3
and   x12, x2, x5
or    x13, x6, x2
add   x14, x2, x2
sd    x15,100(x2)
add   x4, x10, x11
and   x7, x8, x9
ld    x16, 100(x18)
Ld    x17, 200(x19)
```



```
sub   x2, x1, x3
add   x4, x10, x11
and   x7, x8, x9
and   x12, x2, x5
or    x13, x6, x2
add   x14, x2, x2
sd    x15,100(x2)
ld    x16, 100(x18)
ld    x17, 200(x19)
```

c) Insertion of Stalls: Example

```
sub  x2, x1, x3
and x12, x2, x5
or  x13, x6, x2
add x14, x2, x2
sd  x15,100(x2)
```

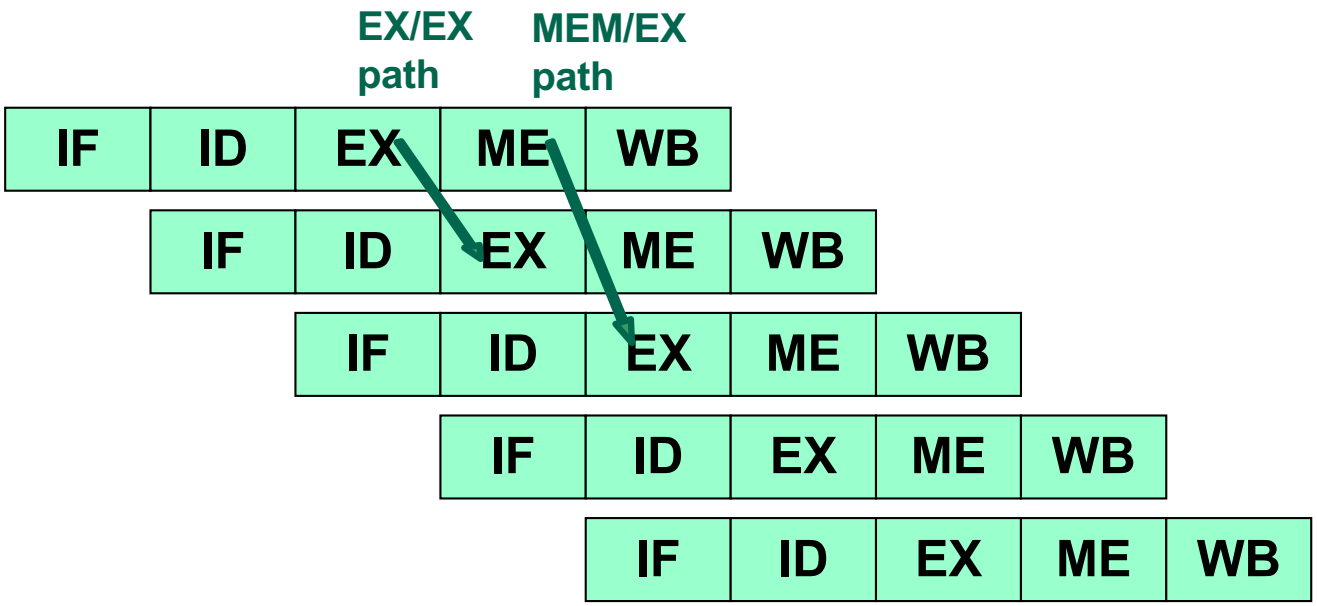


d) Forwarding

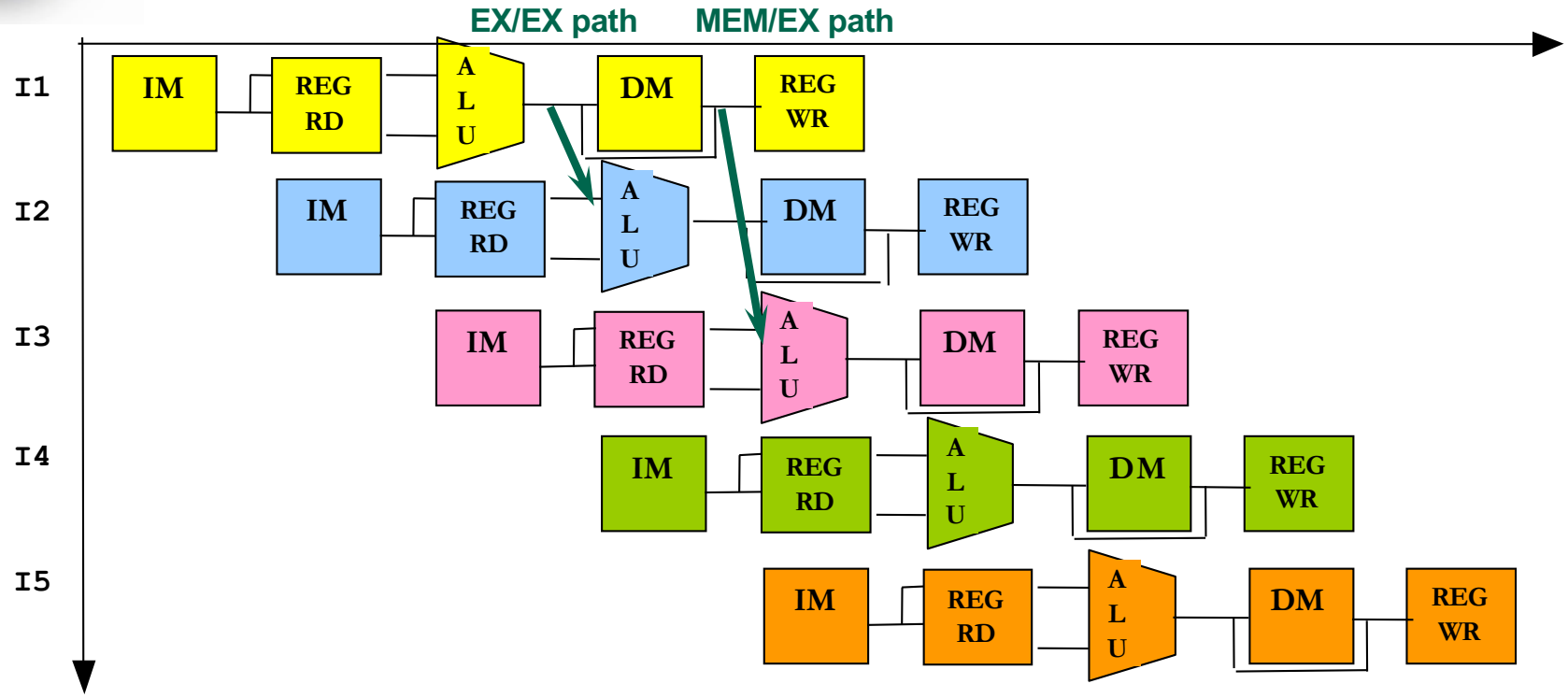
- Data forwarding uses temporary results stored in the pipeline registers instead of waiting for the write back of results in the RF.
- We need to add **new paths and multiplexers** at the inputs of ALU to fetch inputs from pipeline registers to avoid the insertion of stalls in the pipeline.

Forwarding: Example

```
sub  x2, x1, x3
and x12, x2, x5
or  x13, x6, x2
add x14, x2, x2
sd  x15,100(x2)
```

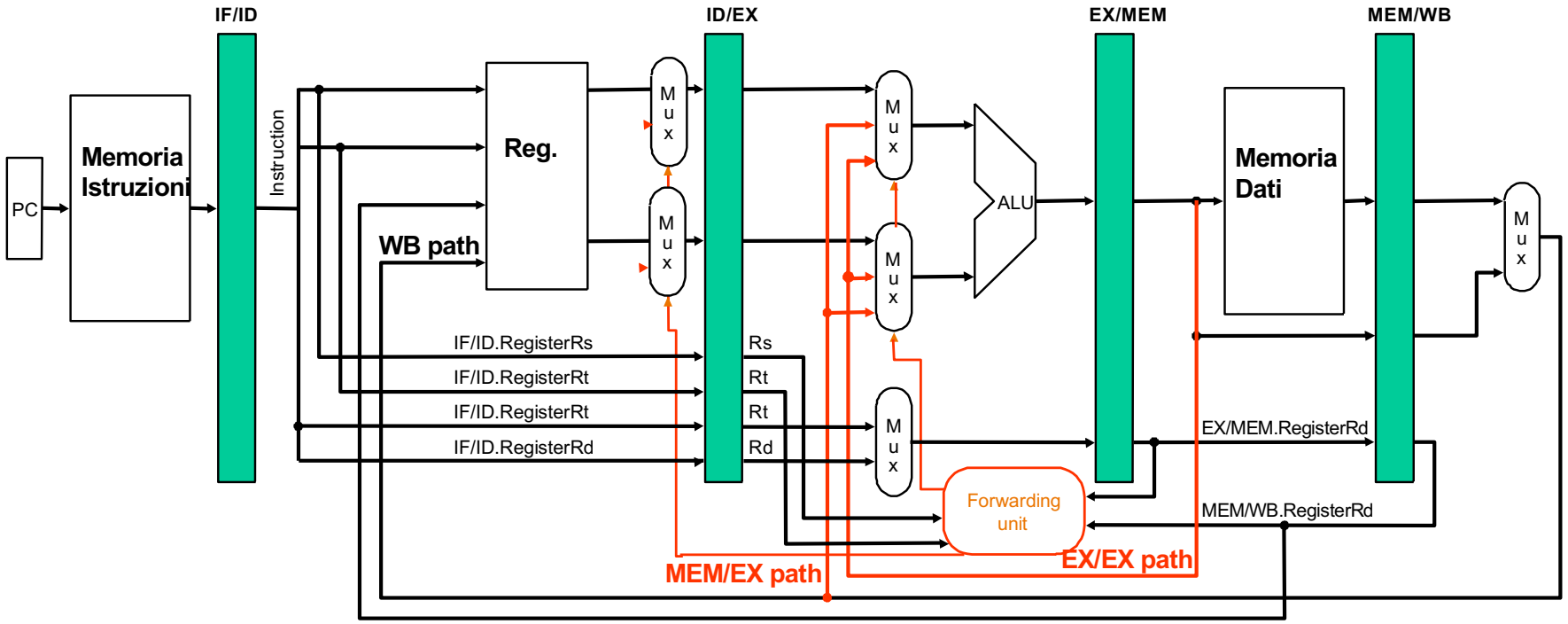


Forwarding Paths



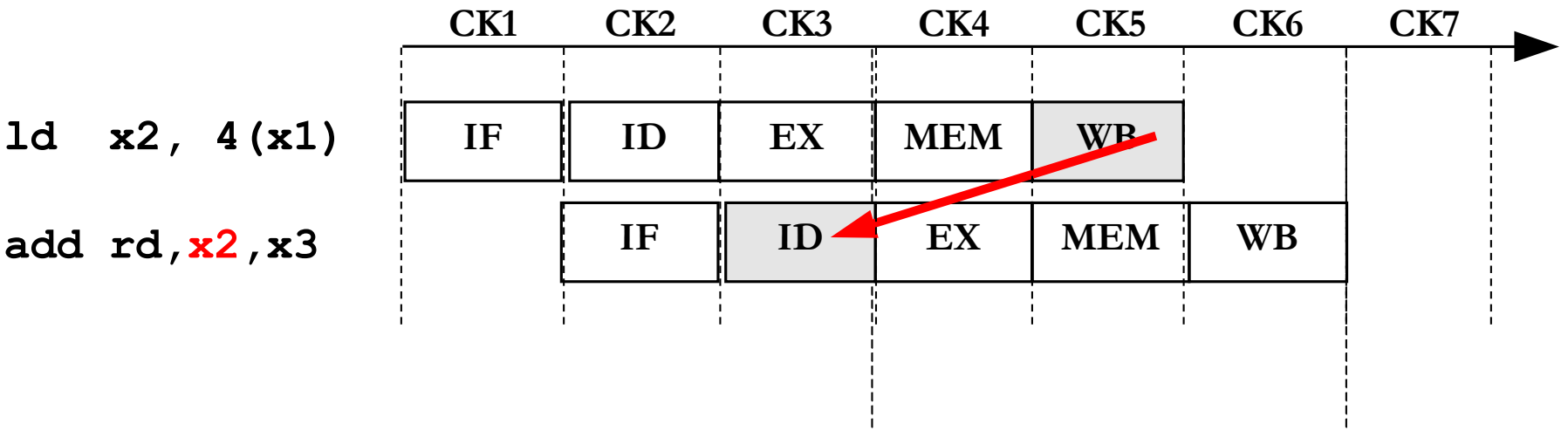
- Two data forwarding paths:
  - EX/EX path
  - MEM/EX path

Implementation of RISC-V with Forwarding Unit



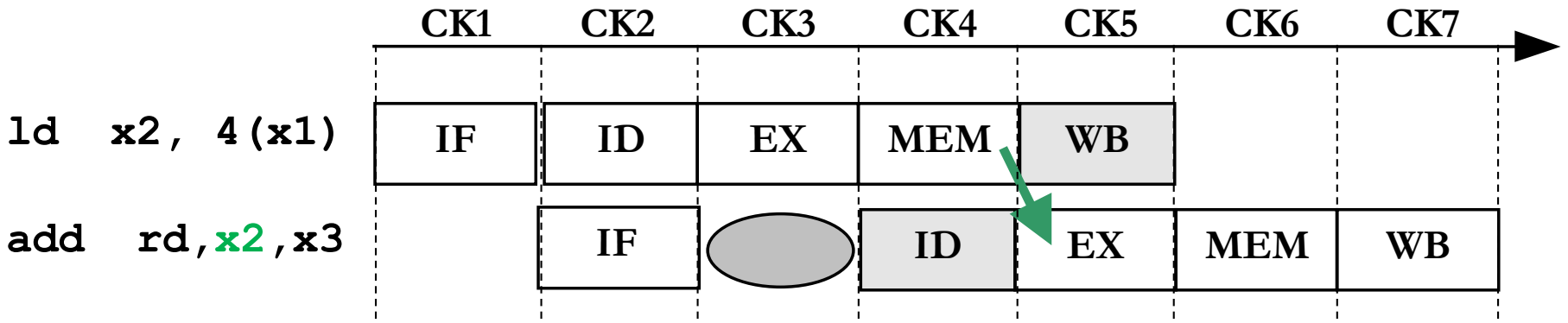
Data Hazards: Load/Use Hazard

```
L1: ld x2, 4(x1)        # x2 <- M [4 + x1]
L2: add rd, x2, x3      # 1° operand x2 depends from L1
```



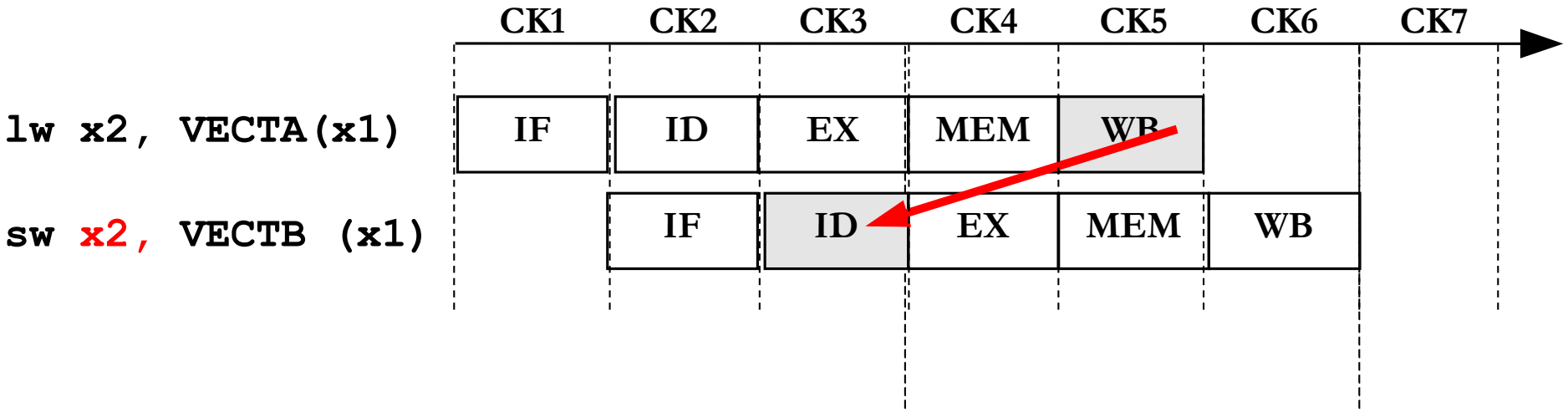
Data Hazards: Load/Use Hazard

- With forwarding using the MEM/EX path: 1 stall needed



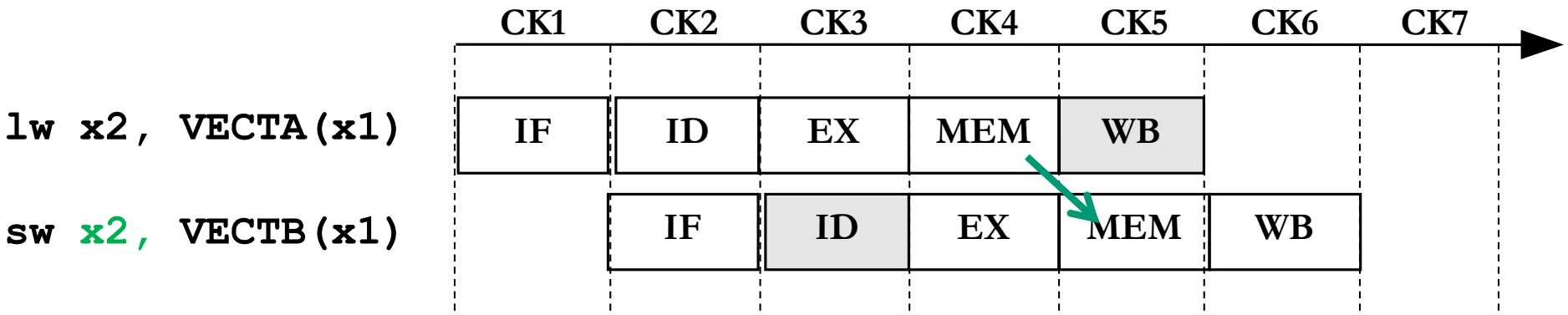
Data Hazards: Load/Store Hazard

```
L1: lw x2, VECTA(x1)   # x2 <- M [VECTA + x1]
L2: sw x2, VECTB(x1)   # M [VECTB + x1] <- x2
```

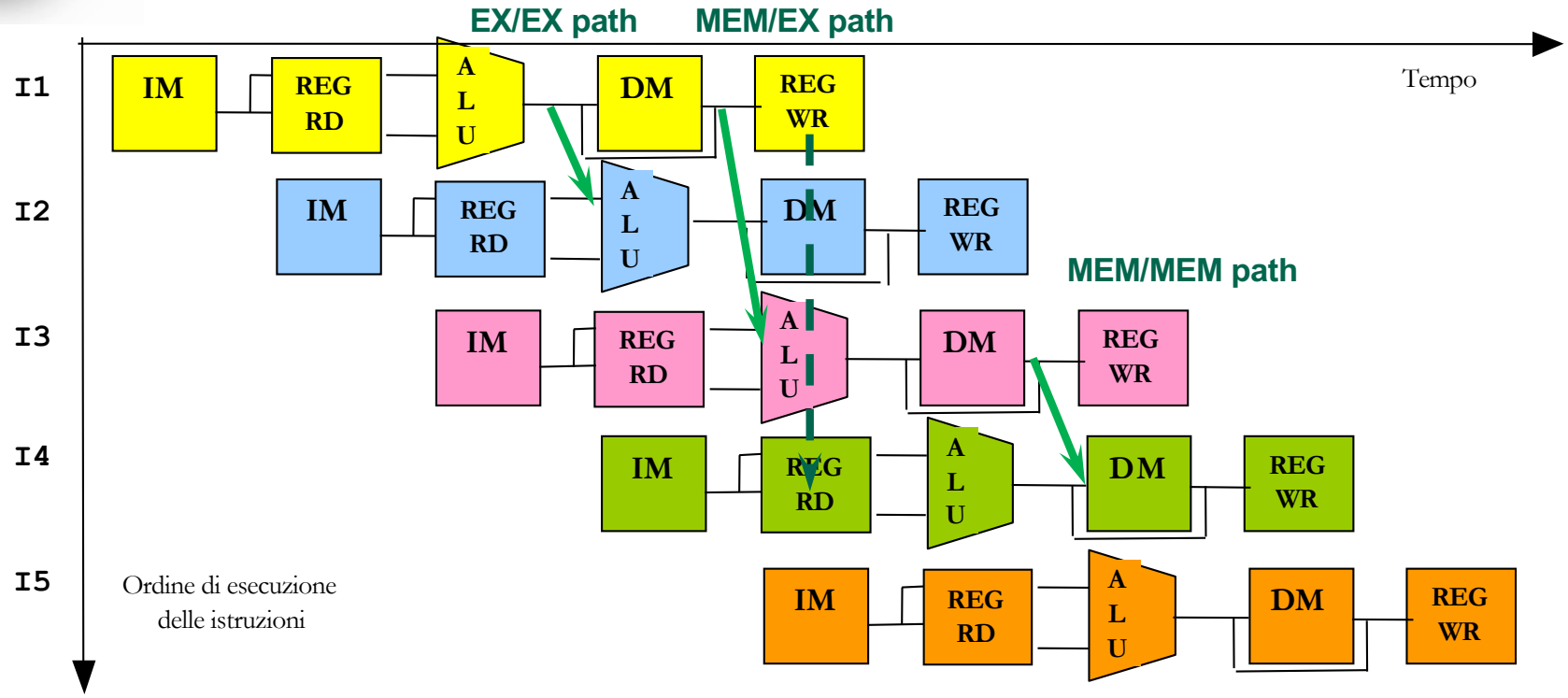


Data Hazards: Load/Store Hazard

- With forwarding by introducing the **MEM/MEM path: hazard solved**



Forwarding Paths



- Three data forwarding paths:
  - EX/EX path
  - MEM/EX path
  - MEM/MEM path (for LOAD/STOREs)

Performance evaluation in pipelining

Performance Metrics

IC = Instruction Count

CPI = Clocks Per Instruction

IPC = Instructions Per Clock = 1 / CPI

Clock Cycles = IC + # Stall Cycles + 4

CPI = # Clock Cycles / IC = (IC + # Stall Cycles + 4) / IC

MIPS = f_{clock} / (CPI * 10^{6})

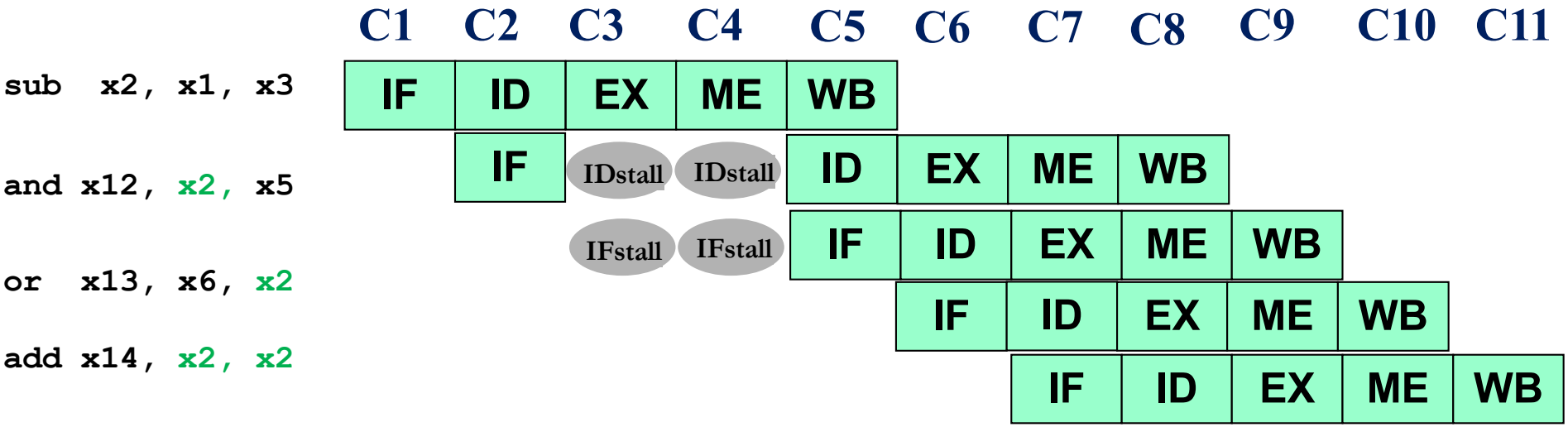
Example

IC = 5

Clock Cycles = IC + # Stall Cycles + 4 = 5 + 2 + 4 = 11

CPI = # Clock Cycles / IC = 11 / 5 = 2.2

MIPS = f_{clock} / (CPI * 10^6) = 500 MHz / 2.2 * 10^6 = 227



Performance Metrics (2)

- Let us consider **n** iterations of a loop composed of **m** instructions per iteration requiring **k** stalls per iteration

$IC_{per\_iter} = m$

$\#\ Clock\ Cycles_{per\ iter} = IC_{per\_iter} + \#\ Stall\ Cycles_{per\_iter} + 4$

$CPI_{per\_iter} = (IC_{per\ iter} + \#\ Stall\ Cycles_{per\_iter} + 4) / IC_{per\_iter}$

$= (m + k + 4) / m$

$MIPS_{per\_iter} = f_{clock} / (CPI_{per\_iter} * 10^{6})$

Asymptotic Performance Metrics

- Let us consider **n** iterations of a loop composed of **m** instructions per iteration requiring **k** stalls per iteration

IC_{AS} = Instruction Count $_{AS}$ = m * n

Clock Cycles = IC $_{AS}$ + # Stall $Cycles_{AS}$ + 4

CPI $_{AS}$ = lim $_{n \to \infty}$ (IC $_{AS}$ + # Stall $Cycles_{AS}$ + 4) / IC $_{AS}$

= lim $_{n \to \infty}$ (m *n + k * n + 4) / (m * n)

= (m + k) / m

MIPS $_{AS}$ = f_{clock} / (CPI_{AS}* 10^{6})

Performance Issues in Pipelining

- The **ideal CPI** on a pipelined processor would be ***1,*** but stalls cause the pipeline performance to degrade form the ideal performance, so we have:

Ave. CPI Pipe = Ideal CPI + Pipe Stall Cycles per Instruction
= 1 + Pipe Stall Cycles per Instruction

- **Pipeline Stall Cycles per Instruction** are due to: Structural Hazards + Data Hazards + Control Hazards + Memory Stalls ***(we will see in the next lessons)***

Reference

- **Appendix A** of the textbook (MIPS Processor):

  J. Hennessey, D. Patterson,

  "Computer Architecture: A Quantitative Approach"
  4th Edition, Morgan-Kaufmann Publishers.