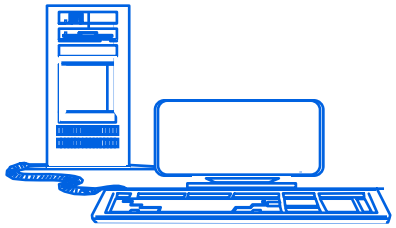
Computer **Architectures**

DLX ISA: Pipelined Implementation



The *Pipelining* Principle

Pipelining **is nowadays the main basic technique deployed to "speed-up" a CPU.**

The key idea for pipelining is general, and is currently applied to several industry fields (productions lines, oil pipelines, …)

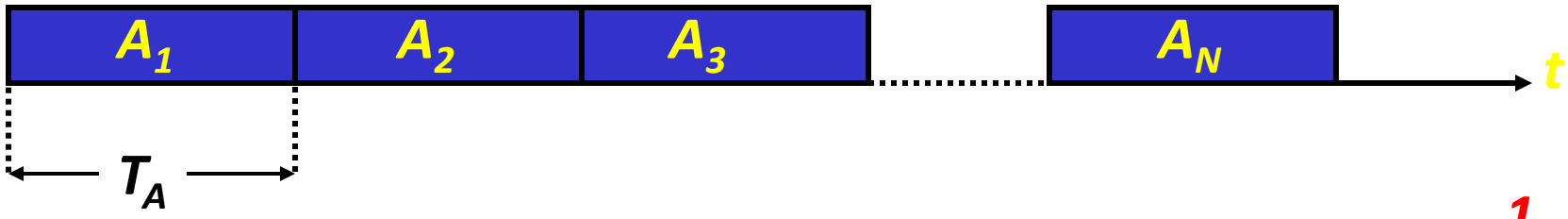
A system *S*, has to execute N times a task *A*:

Latency **: time occurring between the beginning and the end of task A (*T_A*).**

Throughput **: frequency at which each task is completed.**

The *Pipelining* Principle

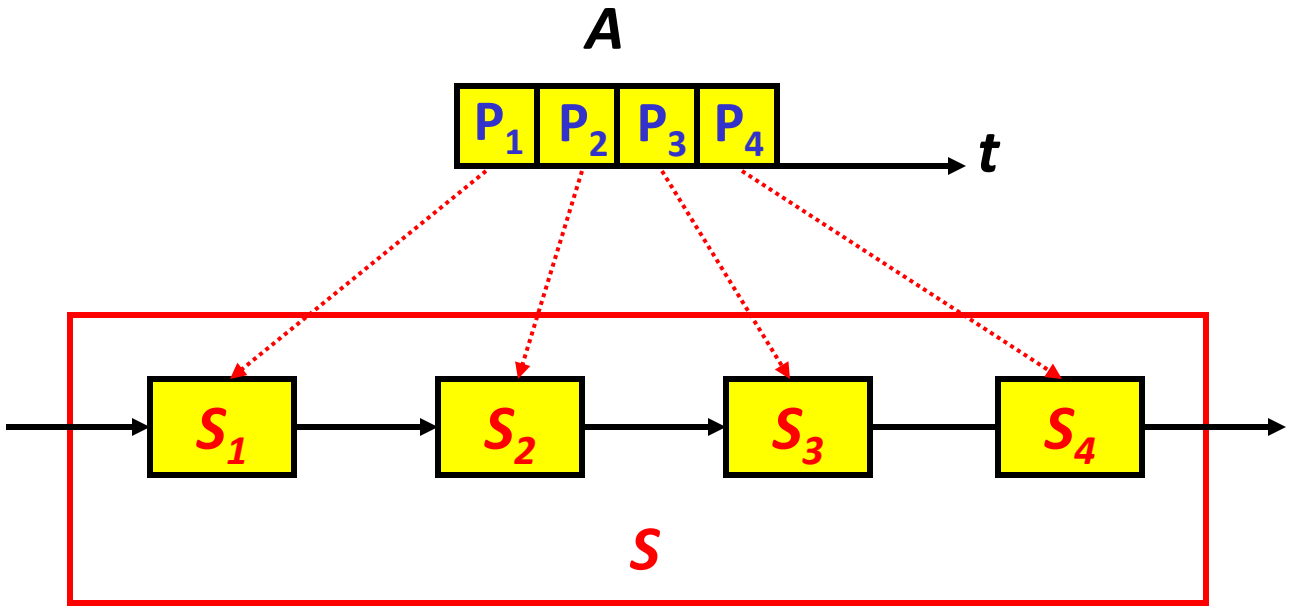
1) Sequential System



Latency (execution time of a single instruction) = T_A

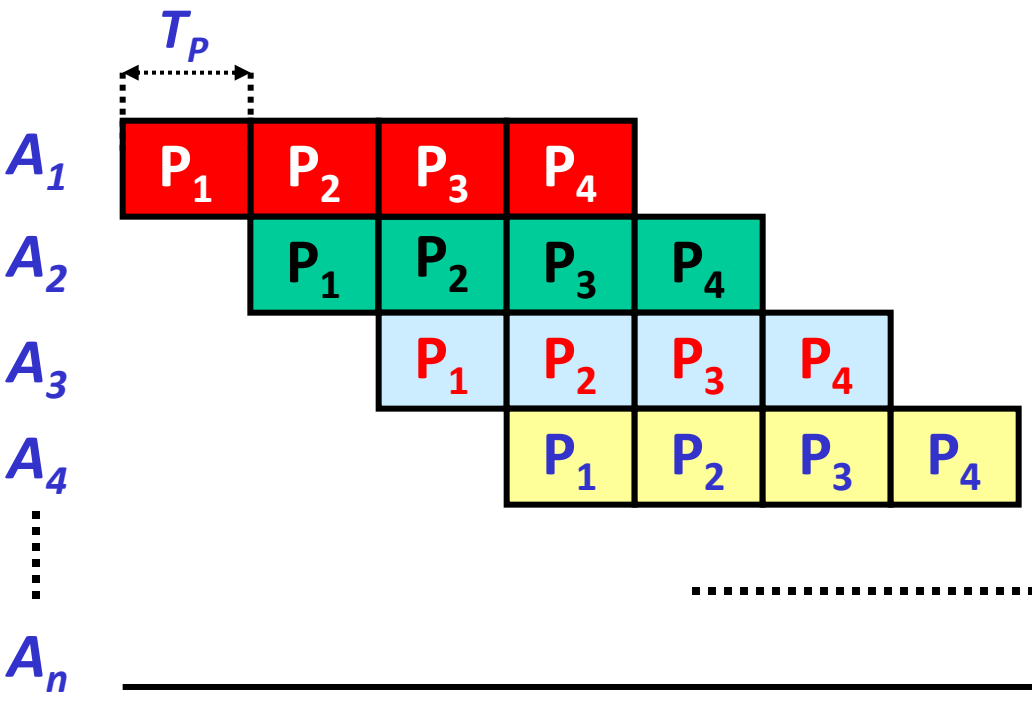
$$Throughput(1) = \frac{1}{T_A}$$

2) Pipelined System



S_i: pipeline stage

The *Pipelining* Principle

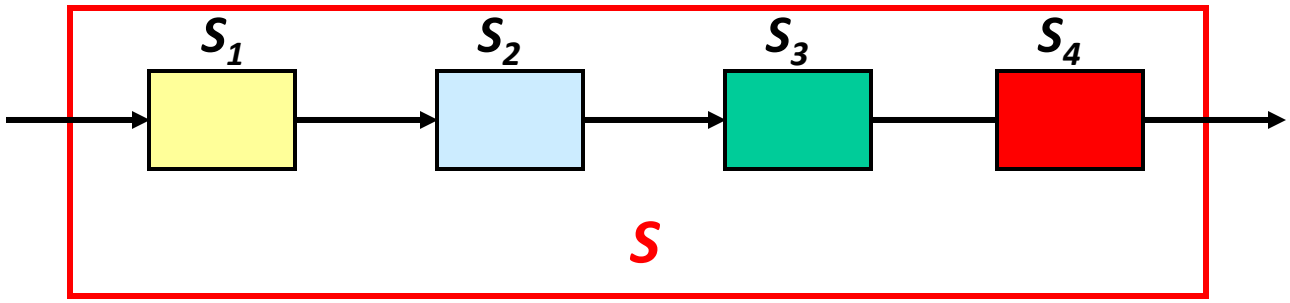


T_P : pipeline cycle

$Latency(2) = 4 * T_P = T_A$

$$Throughput(2) \cong \frac{1}{T_P} = \frac{4}{T_A}$$

$$= 4 * Throughput(1)$$



The *Pipelining* Principle (2)

- Pipelining does not decrease the amount of time needed for carrying out each single task:

Latency(2) = Latency(1)

- Pipelining, instead, increases the Throughput, by multiplying it of a factor **K** equal to the number of stages of the pipeline:

Throughput(2) = K * Throughput(1)

- This yields a reduction, by the same factor K, of the ***total execution time*** of a sequence of N tasks (T_N):

$$T_N = \frac{N}{Throughput} \Rightarrow T_N(1) = \frac{N}{Throughput(1)}, T_N(2) = \frac{N}{Throughput(2)}$$

$$\Rightarrow Speedup(2\ vs\ 1) = \frac{T_N(1)}{T_N(2)} = \frac{Throughput(2)}{Throughput(1)} = K$$

The *Pipelining* Principle (2)

- Ideal case:

$$T_P = T_{Pi} = \frac{T_A}{K} \Rightarrow \text{perfectly balanced pipeline} \Rightarrow \text{Speedup} = K$$

- Real case:

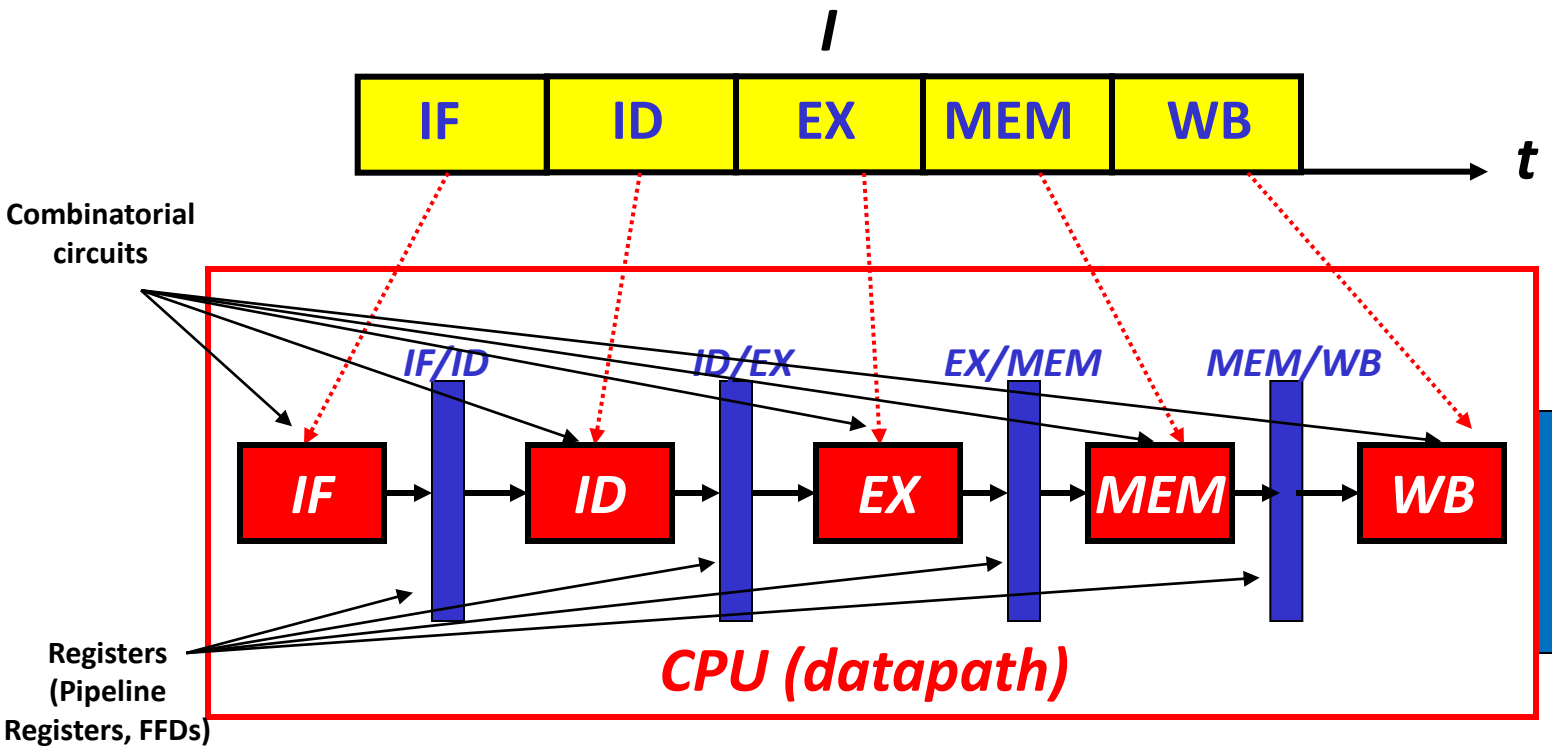
$$T_P = max(T_{P1}, T_{P2}, .., T_{PK}) \Rightarrow \text{(slightly) unbalanced pipeline} \Rightarrow \text{Speedup} < K$$

- Example:
  - T_A = 20 t (t: time unit)
  - T_{P1} = 5t, T_{P2} = 5t, T_{P3} = 6t, T_{P4} = 4t T_P = 6t

$$\Rightarrow Speedup(2\ vs\ 1) = \frac{T_A}{T_P} = \frac{20t}{6t} = (<4)$$

Pipelining in a CPU (*DLX*)

Tasks: $A_1, A_2, A_3 \ldots A_N$ → *Instructions:* $I_1, I_2, I_3 \ldots I_N$



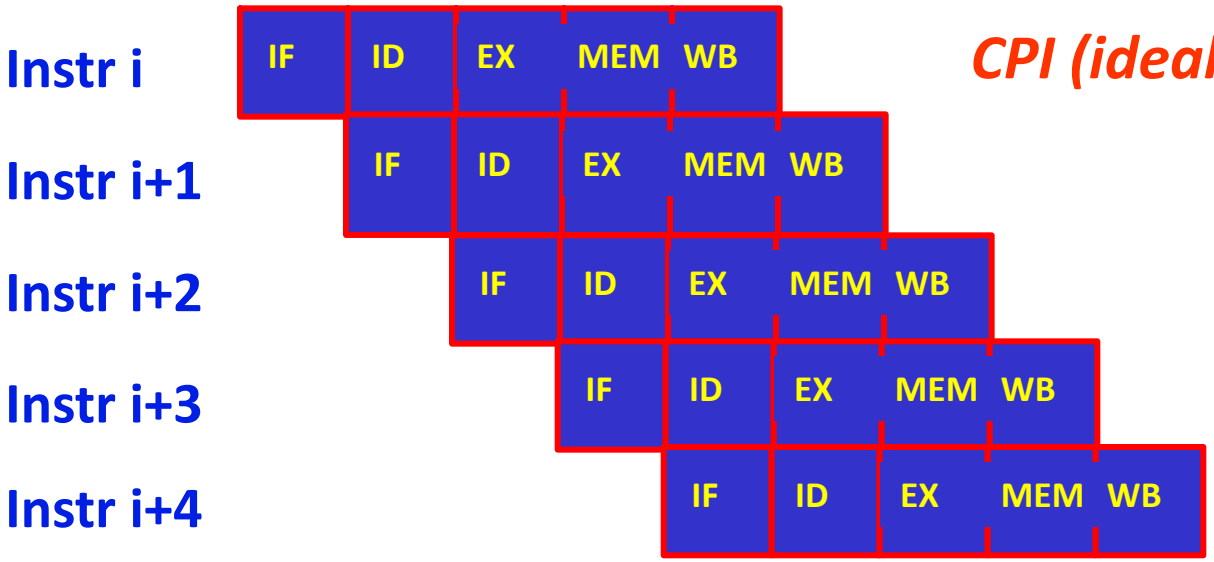
N.B. this architecture is COMPLETELY different from the sequential one

Pipeline Cycle → *Clock Cycle* → *Delay of the slowest stage*

↓

CPI=1 (ideally !)

Pipeline in the *DLX*

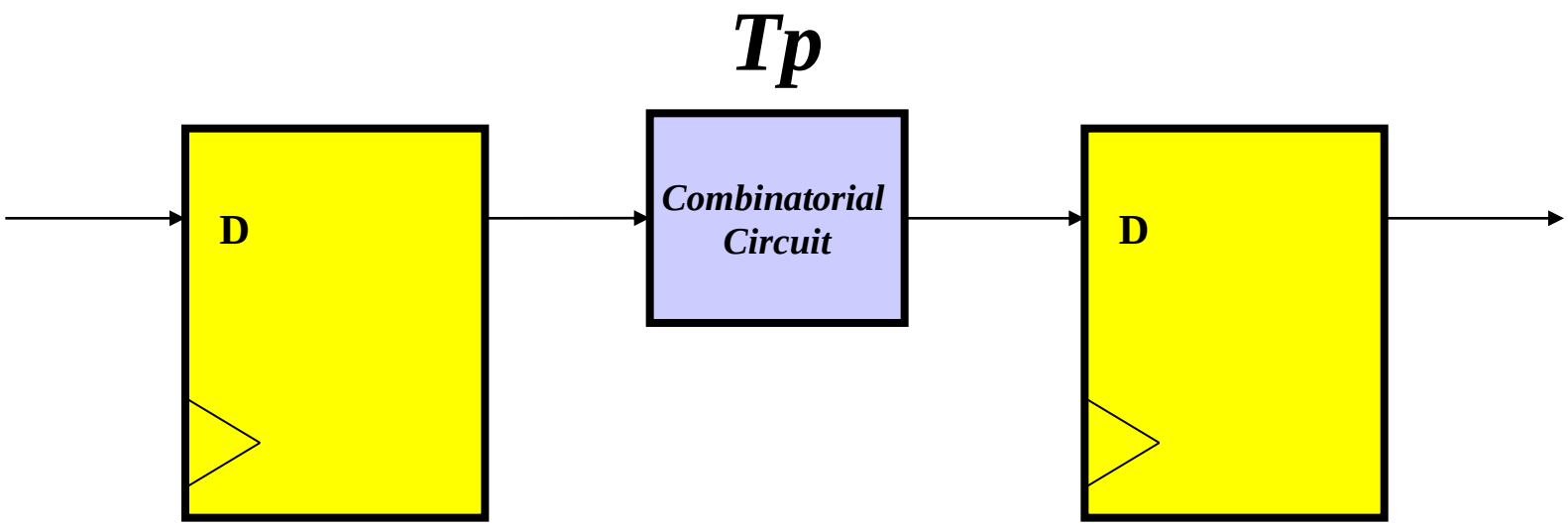


CPI (ideally) = 1

Overhead introduced by the Pipeline Registers:

$$T_{clk} = T_d + T_P + T_{su}$$

- T_{clk}: *Clock Cycle*
- T_d: *Delay of the Input stage register*
- T_P: *Delay of the slowest combinatorial stage*
- T_{su}: *Set-up of the output stage register*



Delay of the Input stage register

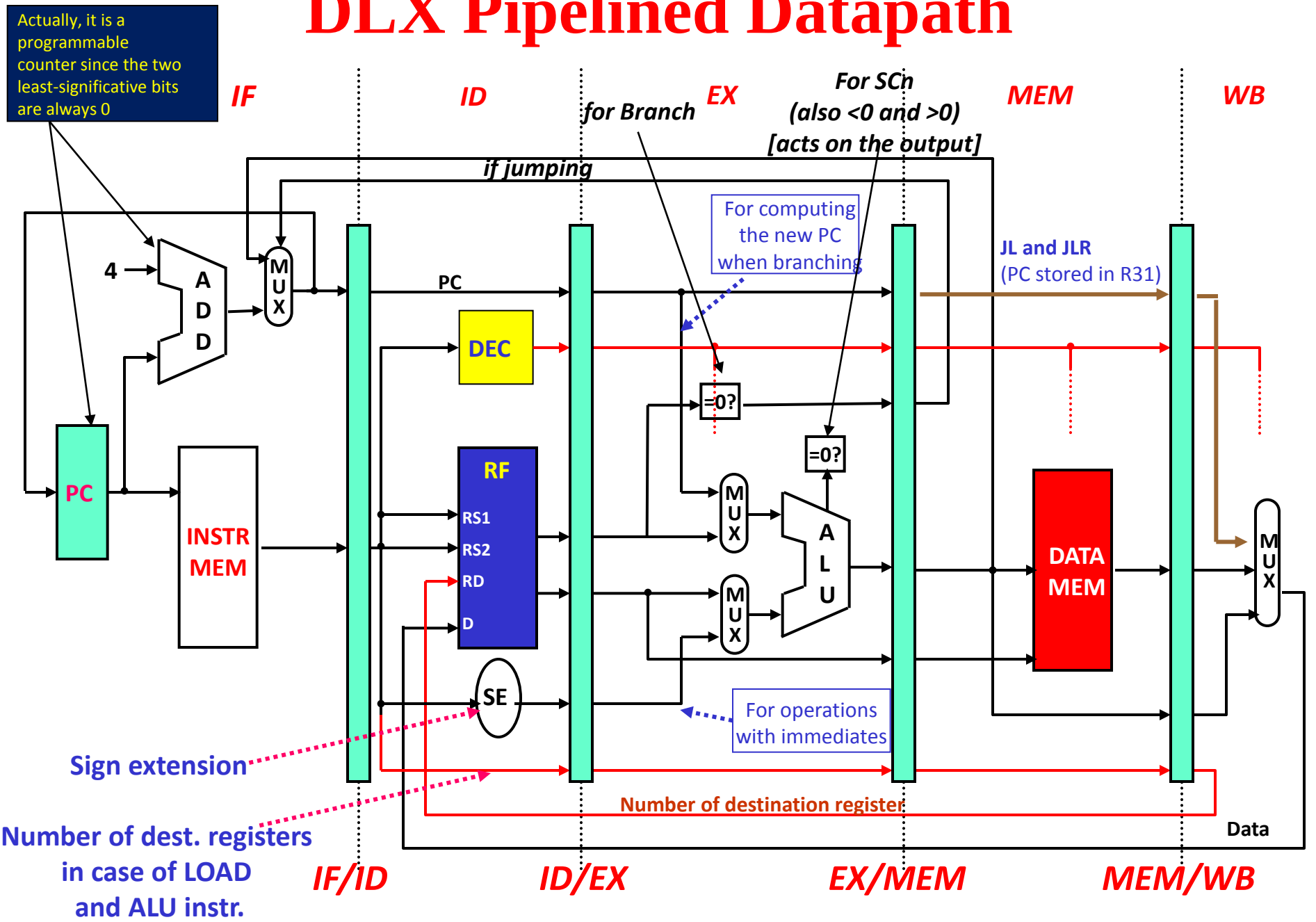
Delay of the slowest combinatorial stage

Set-up of the output stage register

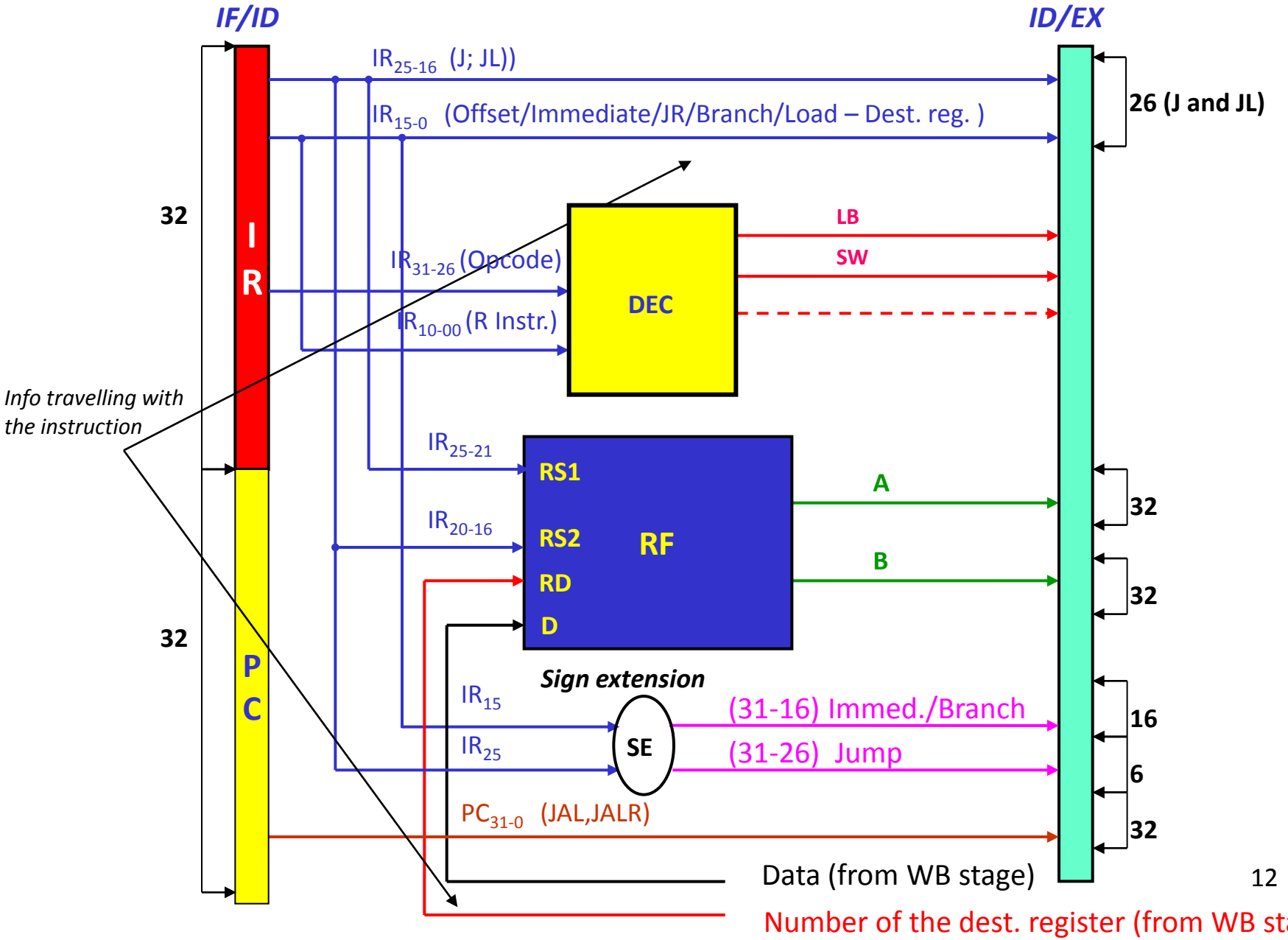
Requirements for implementation of the pipeline

- Each stage has to be active during each clock cycle.
- The PC has to be incremented in the IF stage (instead of ID).
- An ADDER has to be introduced (PC <-- PC+4 – PC <-PC+1) in the IF stage. Since instructions are aligned, a 30 bit register (counter) is incremented each clock cycle (2 ls bits are always 0).
- Two MDRs are required (that will be referred to as LMDR e SMDR) to handle the situation where a LOAD is immediately followed by a STORE (WB-MEM overlapping – two data waiting to be written (one in memory, the other one in RF) are overlapping.
- At every clock cycle, it has to be possible to execute 2 memory accesses (IF, MEM): Instruction Memory (IM) and Data Memory (DM): *"Harvard" Architecture*
- The CPU clock is determined by the slowest stage: IM, DM have to be cache memories (on-chip)
- Pipeline Registers store both data and control information (the Control Unit is "*distributed*" among the pipeline stages)

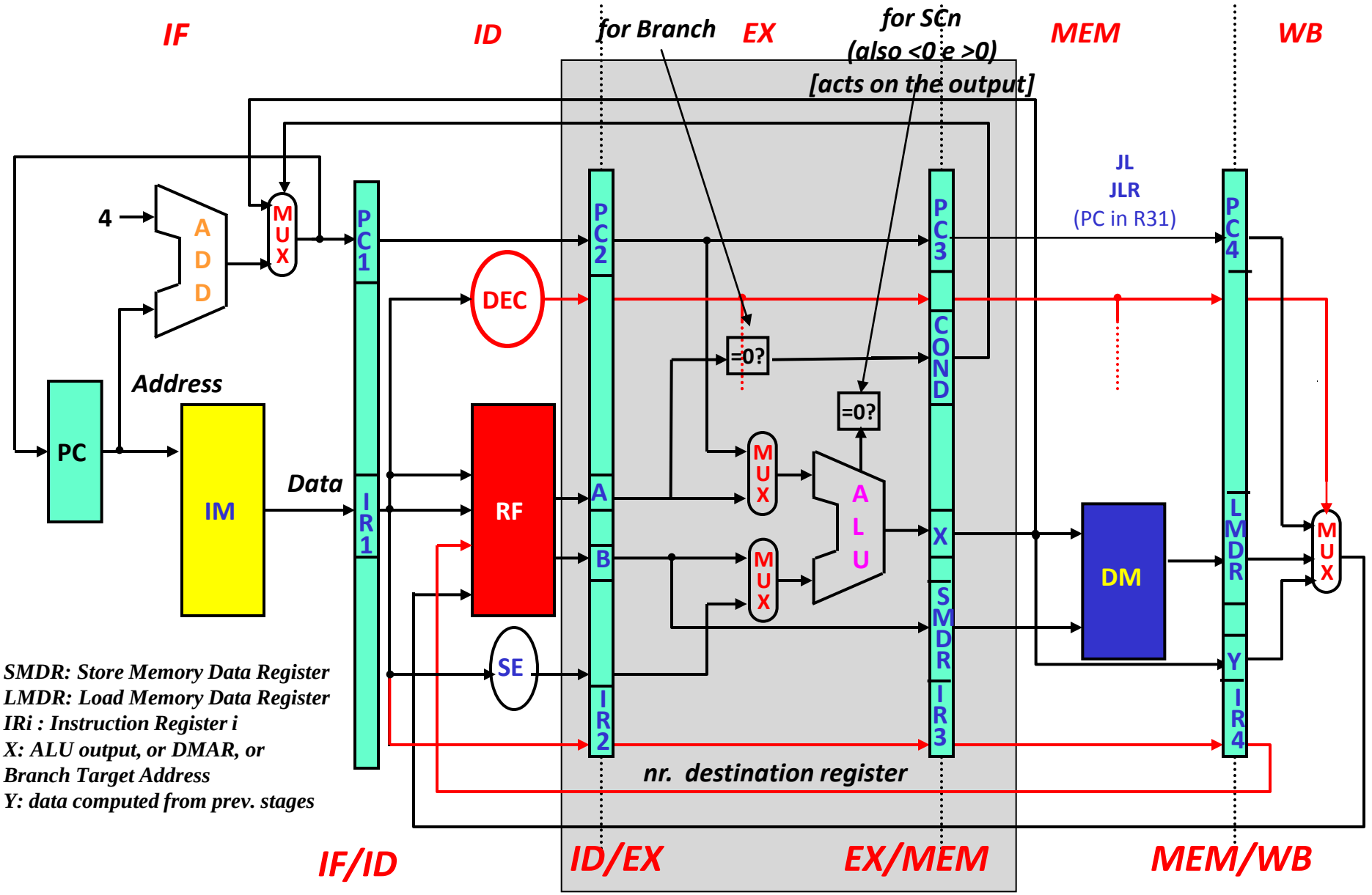
DLX Pipelined Datapath



ID stage


IF/ID
ID/EX
IR25-16 (J; JL))
IR15-0 (Offset/Immediate/JR/Branch/Load – Dest. reg.)
26 (J and JL)
32
I
R
IR31-26 (Opcode)
IR10-00 (R Instr.)
DEC
LB
SW
Info travelling with
the instruction
IR25-21
IR20-16
RS1
RS2
RD
D
RF
A
B
32
32
32
32
P
C
Sign extension
IR15
IR25
SE
(31-16) Immed./Branch
(31-26) Jump
16
6
32
PC31-0 (JAL,JALR)
Data (from WB stage)
Number of the dest. register (from WB stage)

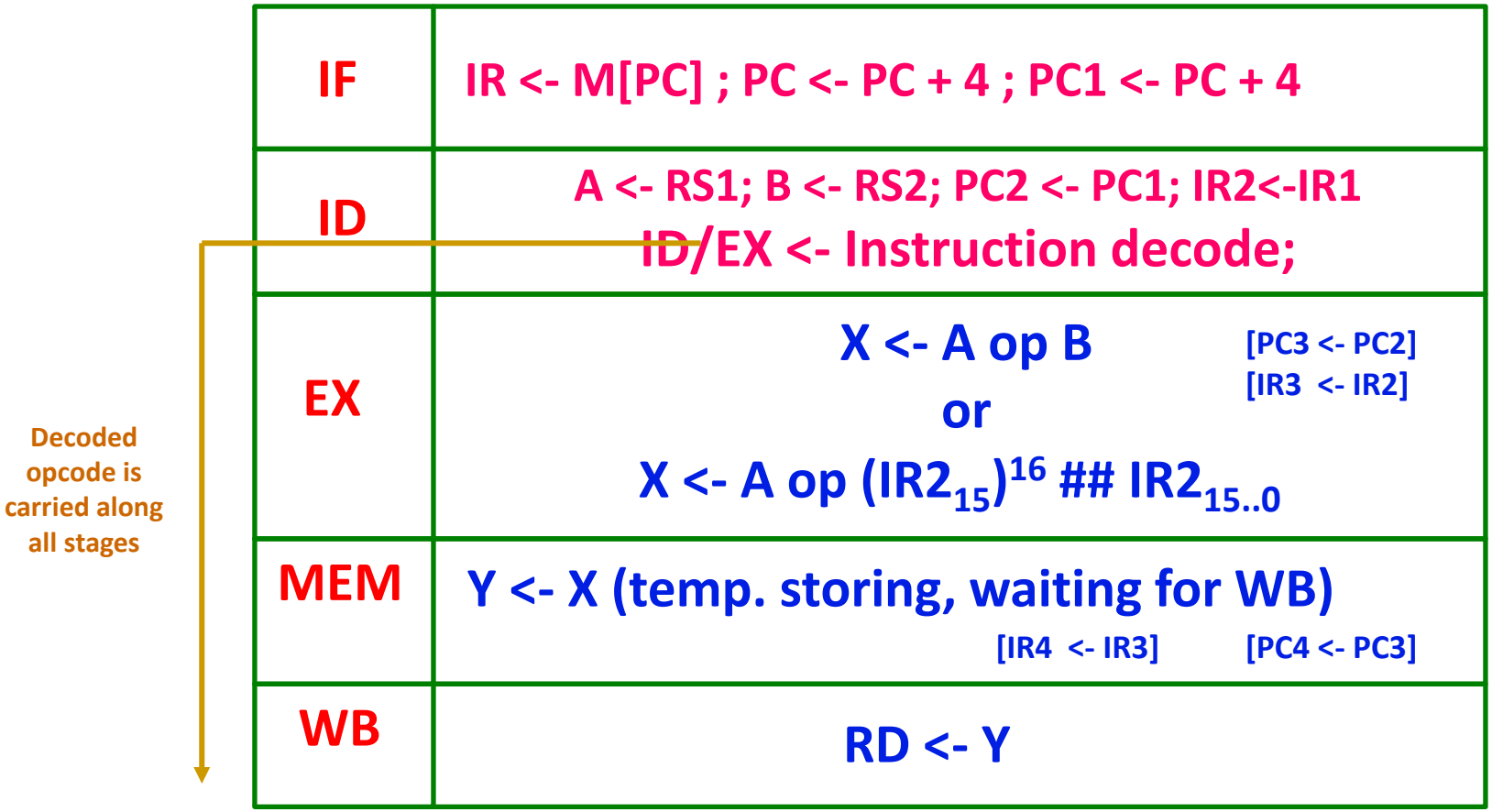
DLX Pipelined Datapath



SMDR: Store Memory Data Register
LMDR: Load Memory Data Register
IRi : Instruction Register i
X: ALU output, or DMAR, or Branch Target Address
Y: data computed from prev. stages

Pipelined execution of an "ALU" instruction

NOTE: for these instructions, RS2/RD need to be carried along the pipeline and up to the WB stage

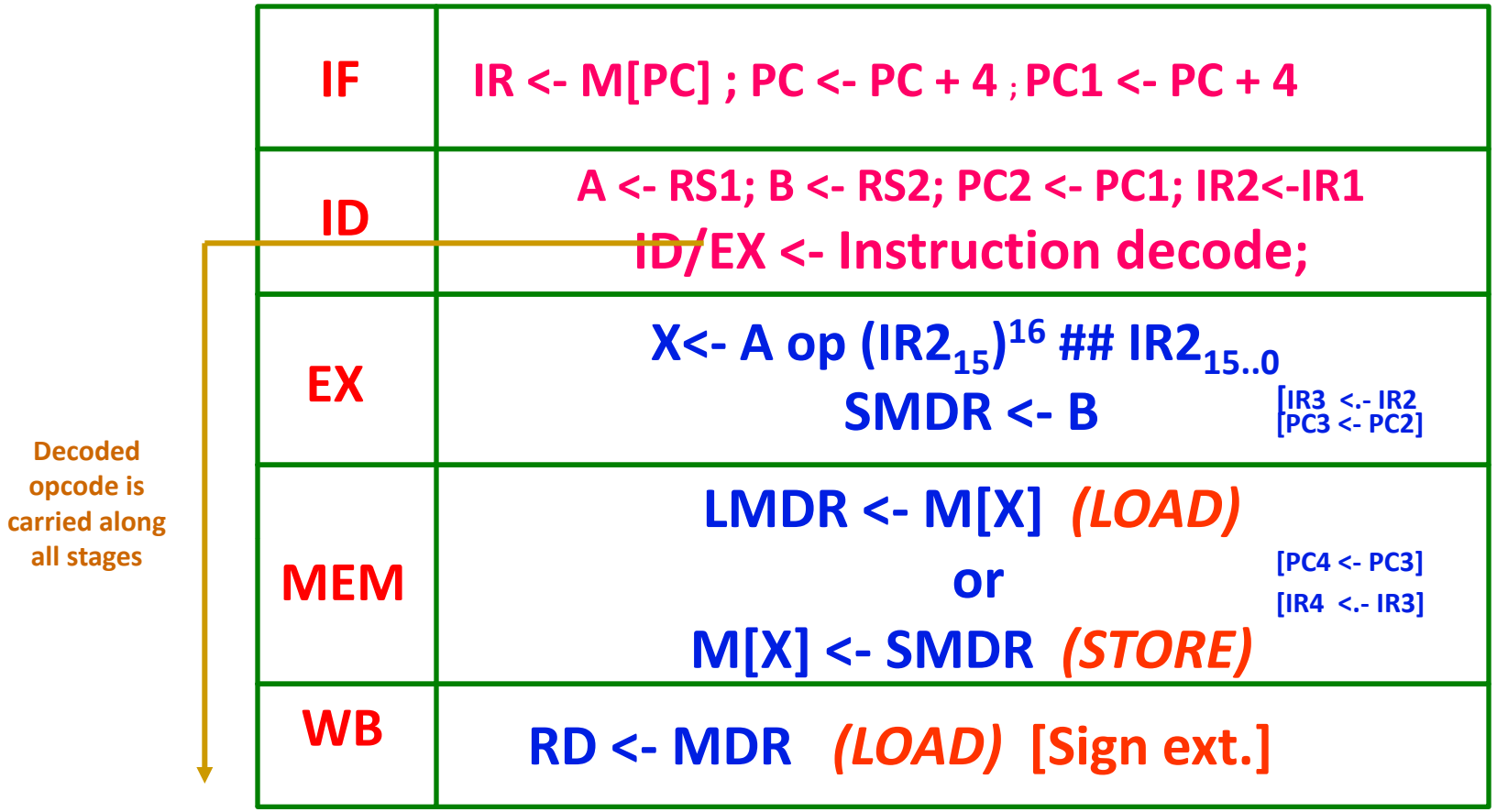


NOTE:
IRi bits that are not needed for all instructions are dropped during successive stages.

From a stage to the next one, those bits that are needed for all instructions are kept

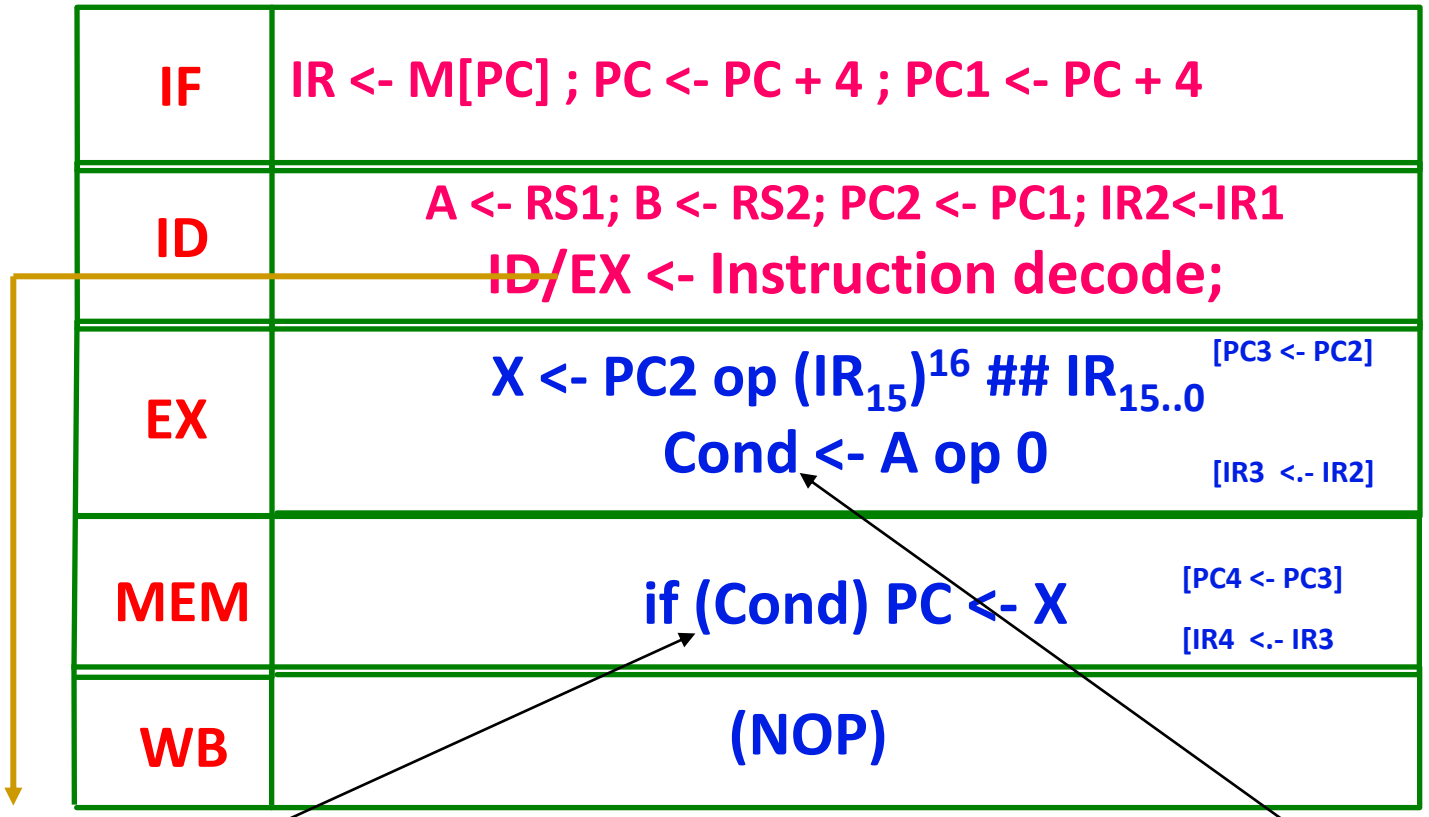
X : "ALUOUTPUT" (in EX/MEM), Y : "ALUOUTPUT1"

Pipelined execution of a "MEM" instruction

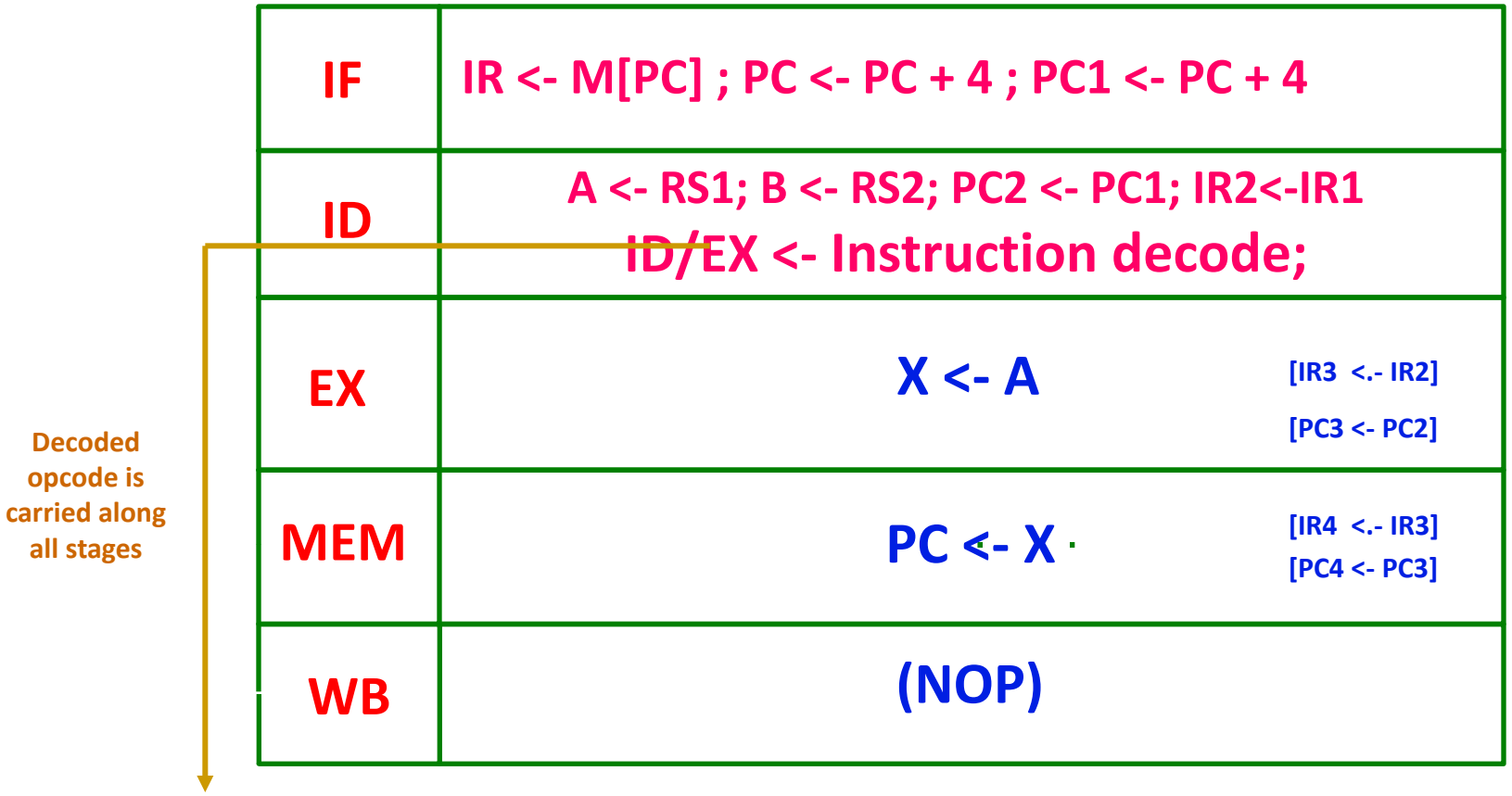


X : "DMAR (Data Memory Address Registrer)"

Pipelined execution of a "BRANCH" instruction (normally after a SCn instruction)

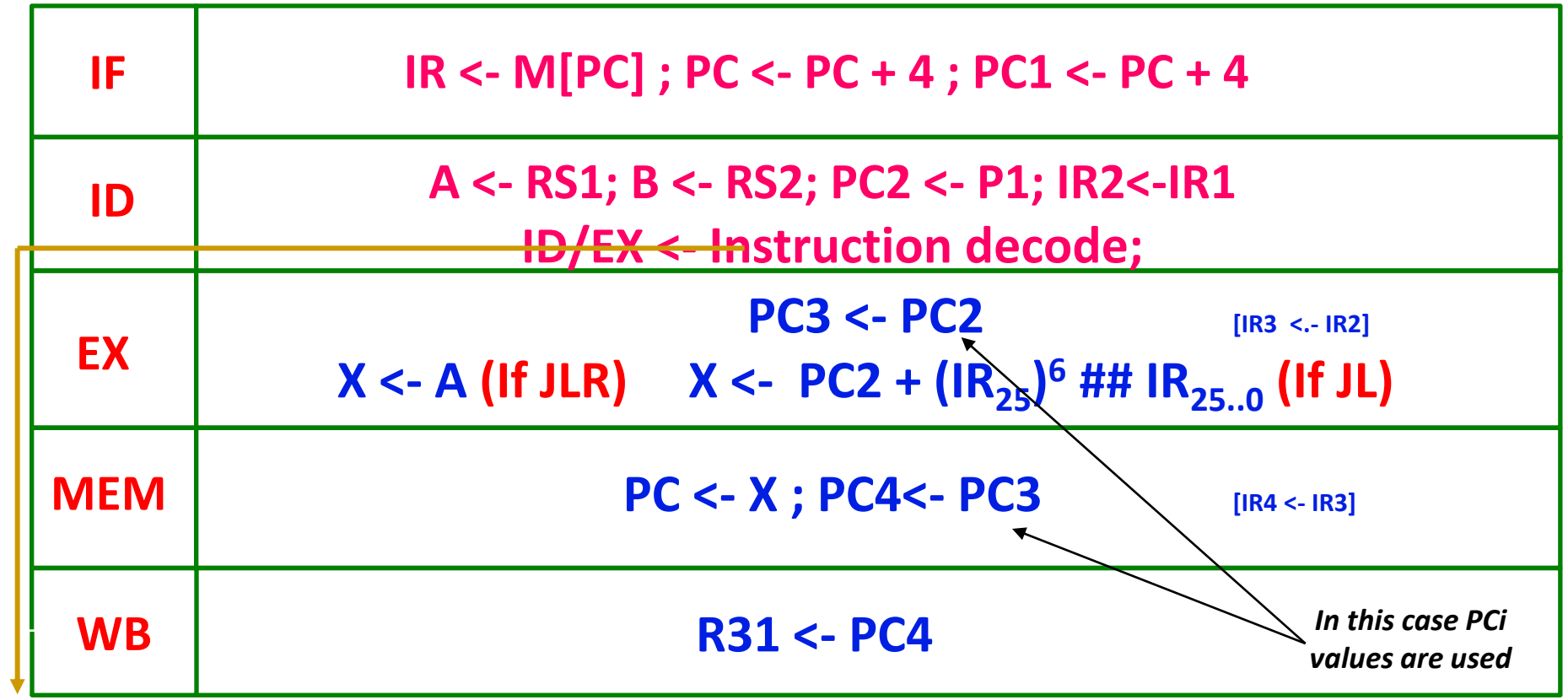


Pipelined execution of a "JR" instruction



What would the stage sequence be for a J instruction?

Pipelined execution of a "JL or JLR" instruction



NOTE: Writing on R31 can NOT be done on-the-fly since it could overlap with another register write operation

What would be the sequence in case of SCn (ex SLT R1,R2,R3) ?

IF	IR <- M[PC] ; PC <- PC + 4 ; PC1 <- PC + 4
ID	A <- RS1; B <- RS2; PC2 <- P1; IR2<-IR1 ID/EX <- Instruction decode;
EX	?
MEM	?
WB	?

Pipeline hazards

A *"Hazard"* occurs when, in a specific clock cycle, an instruction currently flowing through a pipeline stage can not be executed in the same clock cycle.

- *Structural Hazards* – The same resource is used by two different pipeline stages: the instructions currently in those stages can not be executed simultaneously.

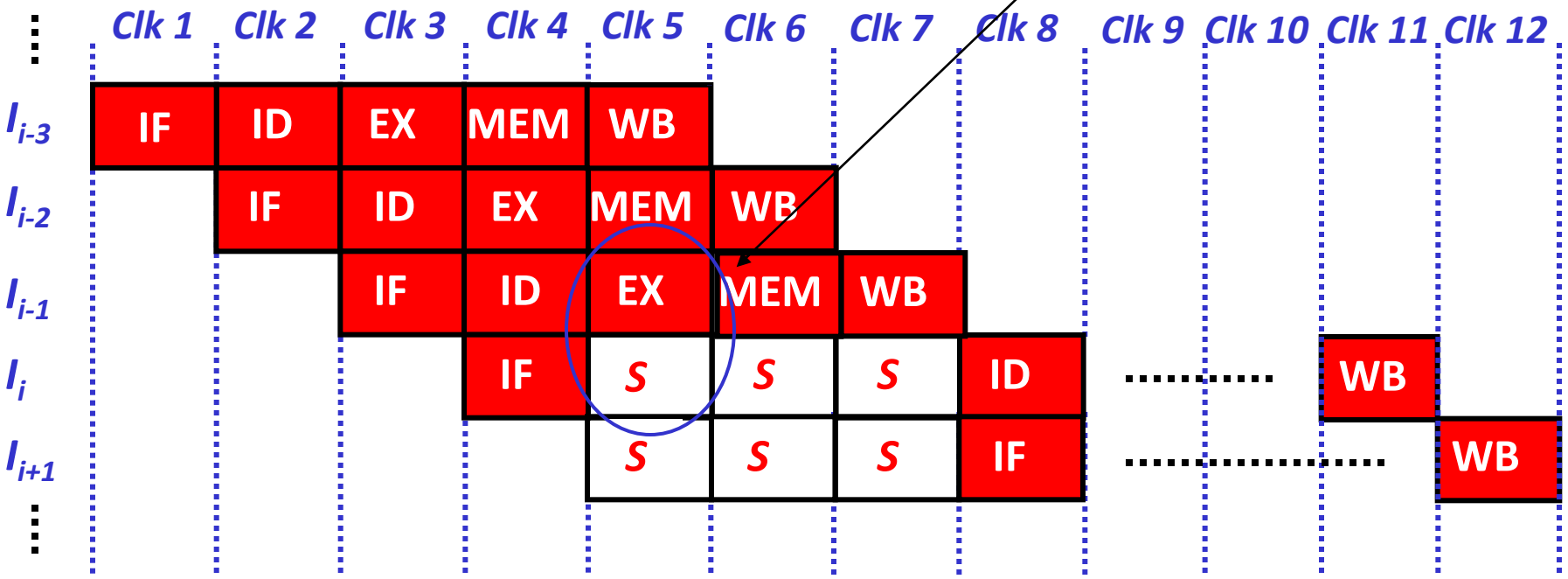
- *Data Hazards* – they are due to instruction *dependencies*. For example, an instruction that needs to read a register not yet written by a previous instruction (*Rear After Write - RAW)*.

- *Control Hazards* – The instructions that follow a branch *depend* from the branch result (*taken/not taken*).

The instruction that can not be executed has to be stopped ("pipeline stall" or "pipeline bubbling"), together with all the following instructions, while the previous instructions can proceed normally (so as to eliminate the hazard).

Hazards and stalls

The consequence of a data hazard: if instruction I_i needs thre result of instruction I_{i-1} (data are read in the ID stage), it has to wait until after WB of I_{i-1}



Stall: the clock signal for I_i, I_{i+1}, .. is stopped for three cycles

$T_5 = 8 * CLK = (5 + 3) * CLK$

$T_5 = 5 * (1 + 3/5) * CLK$

ideal CPI

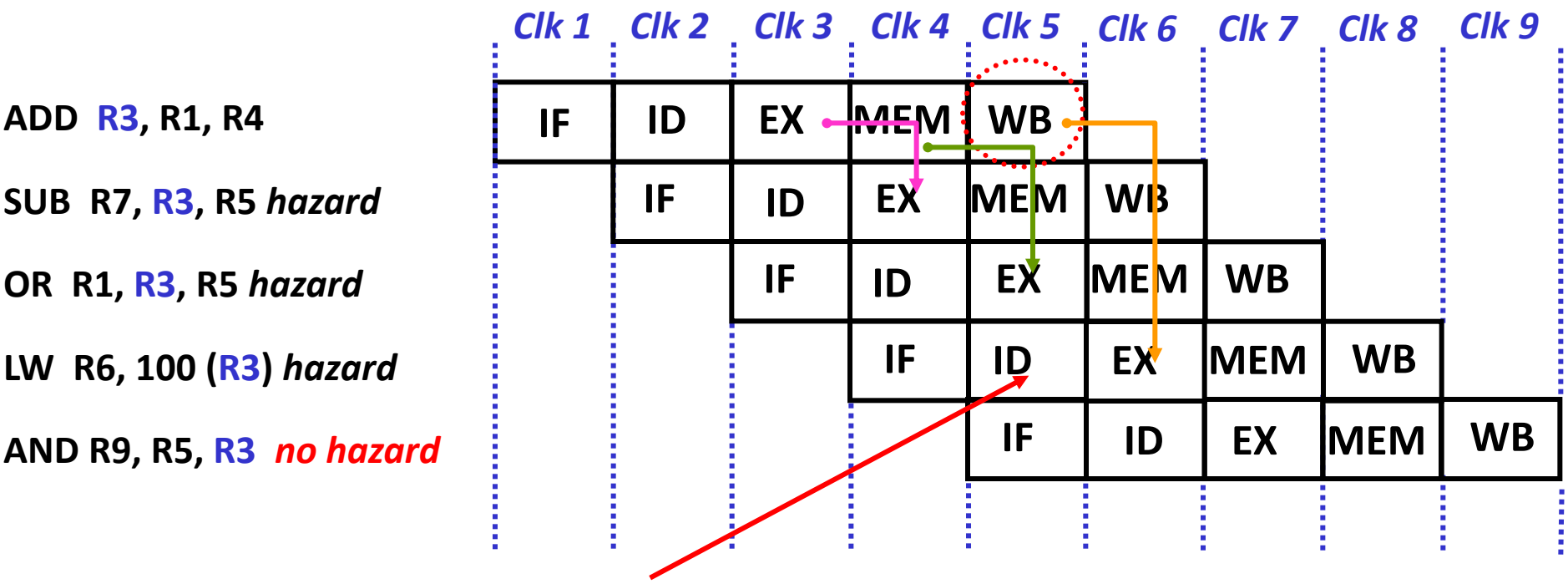
Stalls per Instruction

$T_N = N * 1 * CLK$

$T_N = N * (1 + S) * CLK$

effective CPI

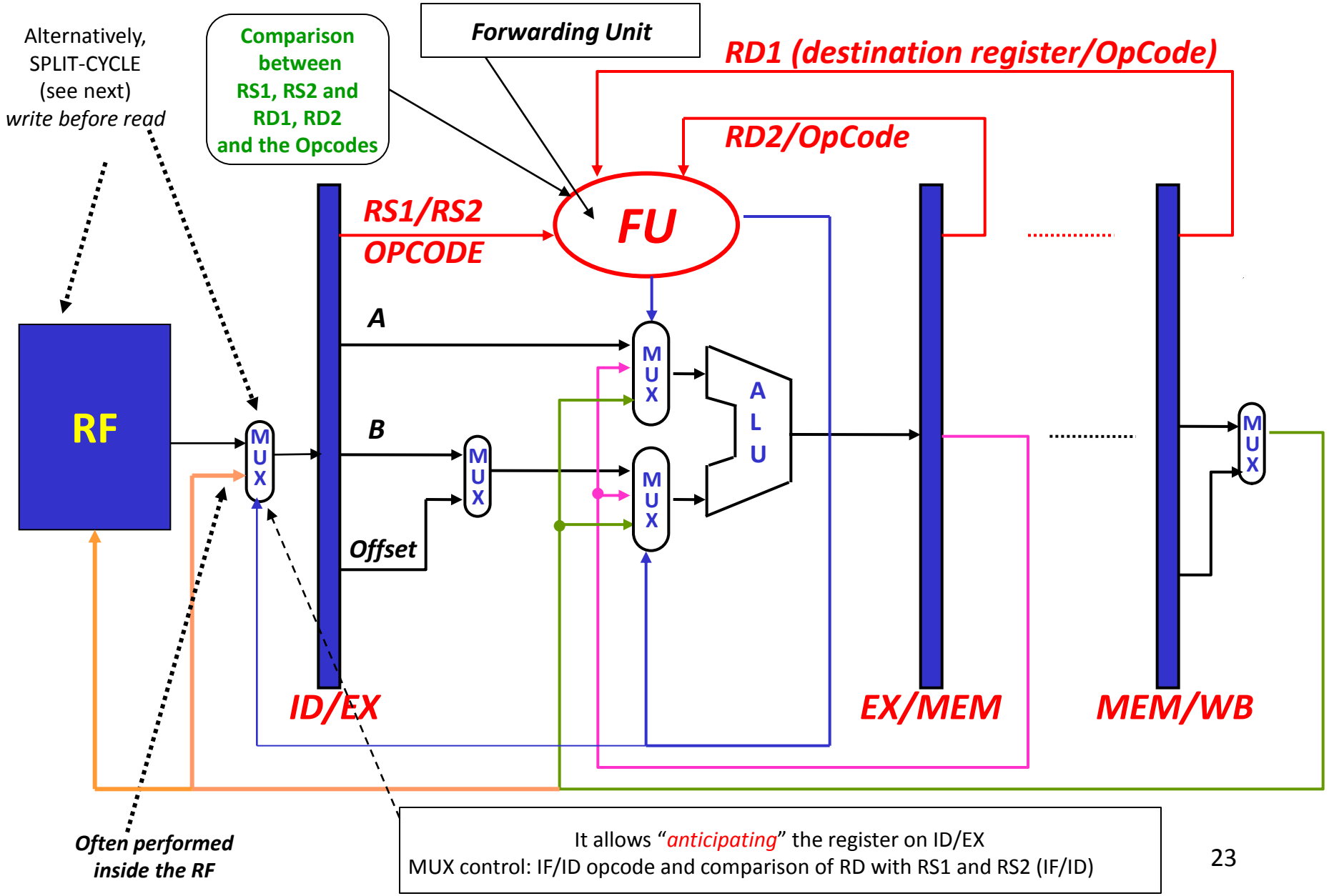
Forwarding



Forwarding **allows eliminating almost all RAW hazards of the DLX pipeline *without stalling* the pipeline.**

(NOTE: in the DLX, registers are *modified only* in WB)

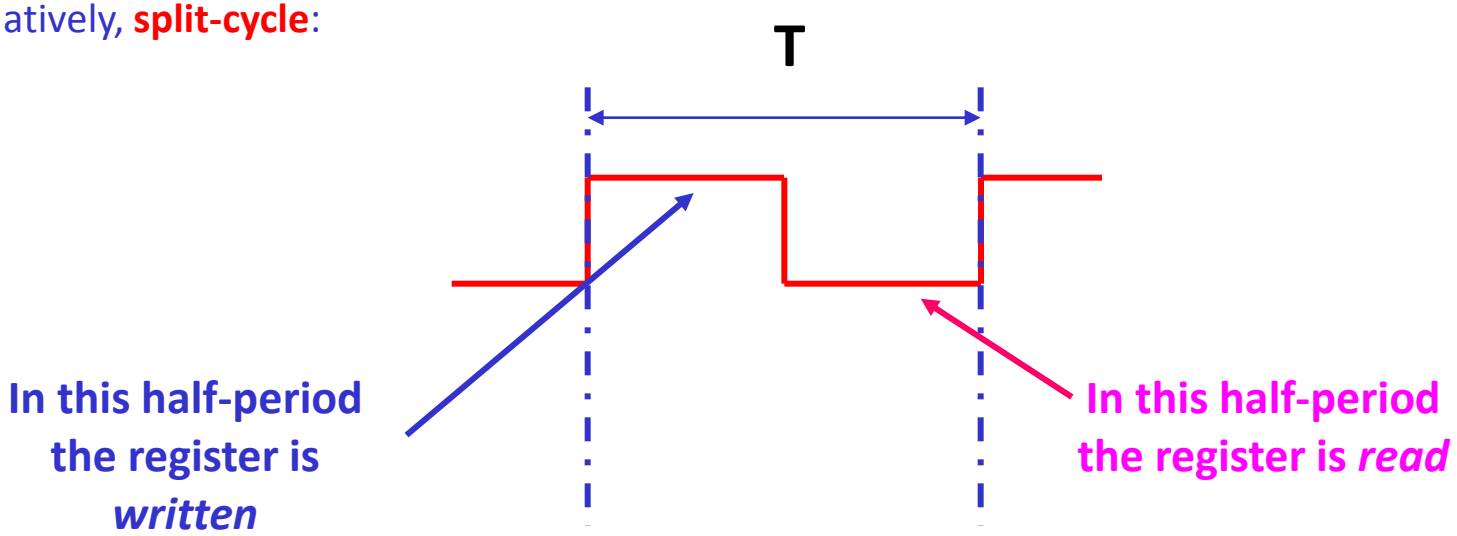
Forwarding implementation



It allows "*anticipating*" the register on ID/EX
MUX control: IF/ID opcode and comparison of RD with RS1 and RS2 (IF/ID)

Forwarding Unit

- Within the Forwarding Unit, the opcodes of the instructions in the EX, MEM and WB stages are decoded.
- If the instruction in the EX stage needs a register value (either A or B i.e. an ALU instruction, NOT a J or Branch instruction) the opcodes of the instructions in the MEM and WB stages are examined. If they require a register update, the number of the involved register is compared with the register numbers of the instruction in the EX stage. If there is a match then the corresponding data is forwarded to the EX stage, thus replacing the data read from the register file
- The bypass MUXes (inputs of the ID/EX barrier) are needed because a fetched instruction can require the contents of registers whose numbers can match that of the instruction in the WB stage (if it must store a register value). In this case data must be read from the MEM/WB barrier instead from the register file.
- Alternatively, **split-cycle**:



Data hazard due to LOAD instructions

LW R1,32(R6)	IF	ID	EX	MEM	WB
ADD R4,R1,R7		IF	ID	EX	MEM
SUB R5,R1,R8			IF	ID	EX
AND R6,R1,R7				IF	ID

NOTE: the datum required by the ADD is available only *at the end* of the MEM stage. The hazard can not be eliminated by means of forwarding (unless there is an additional input in the MUXs between memory and ALU and everything is done in the same clock cycle – delays, there is a memory access in between which is already slow by itself!)

As a matter of fact, the clock signal is not generated. The clock block is propagated along the pipeline one stage at a time.

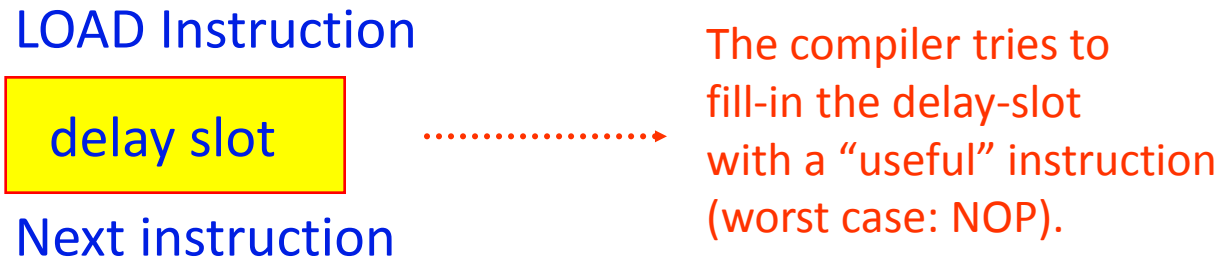
The pipeline needs to be stalled

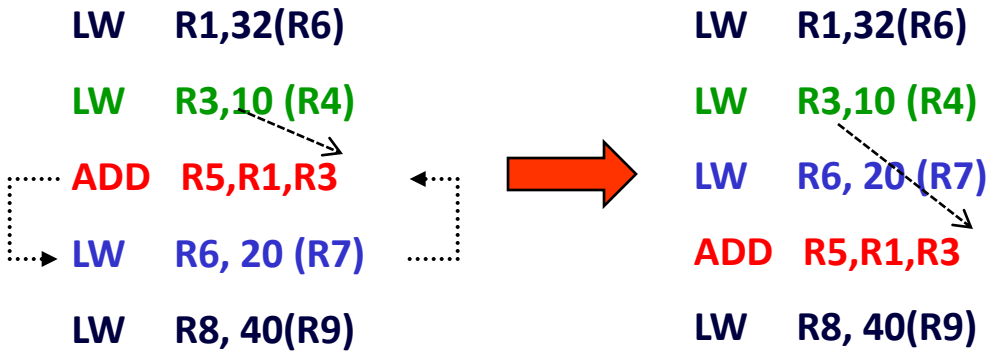
From the end of this stage onwards: standard forwarding MEM->EX

LW R1,32(R6)	IF	ID	EX	MEM	WB	
ADD R4,R1,R7		IF	ID	*S*	EX	MEM
SUB R5,R1,R8			IF	*S*	ID	EX
AND R6,R1,R7				*S*	IF	ID

Delayed load

In many RISC CPUs, the hazard associated with the LOAD instruction is not handled by the hardware through pipeline stalling, instead it is handled via software by the compiler (*delayed load*):





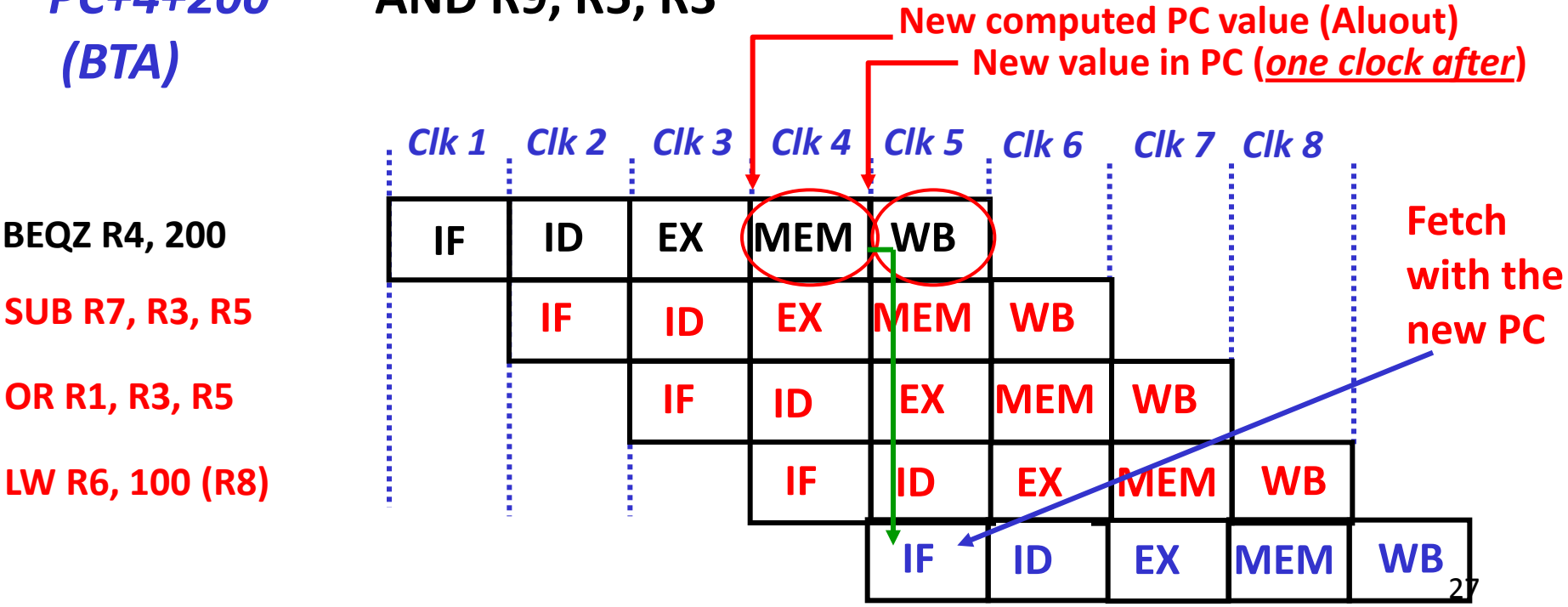
Control Hazards

PC	BEQZ R4, 200
PC+4	SUB R7, R3, R5
PC+8	OR R1, R3, R5
PC+12	LW R6, 100 (R8)
⋮	⋮
PC+4+200 (BTA)	AND R9, R5, R3

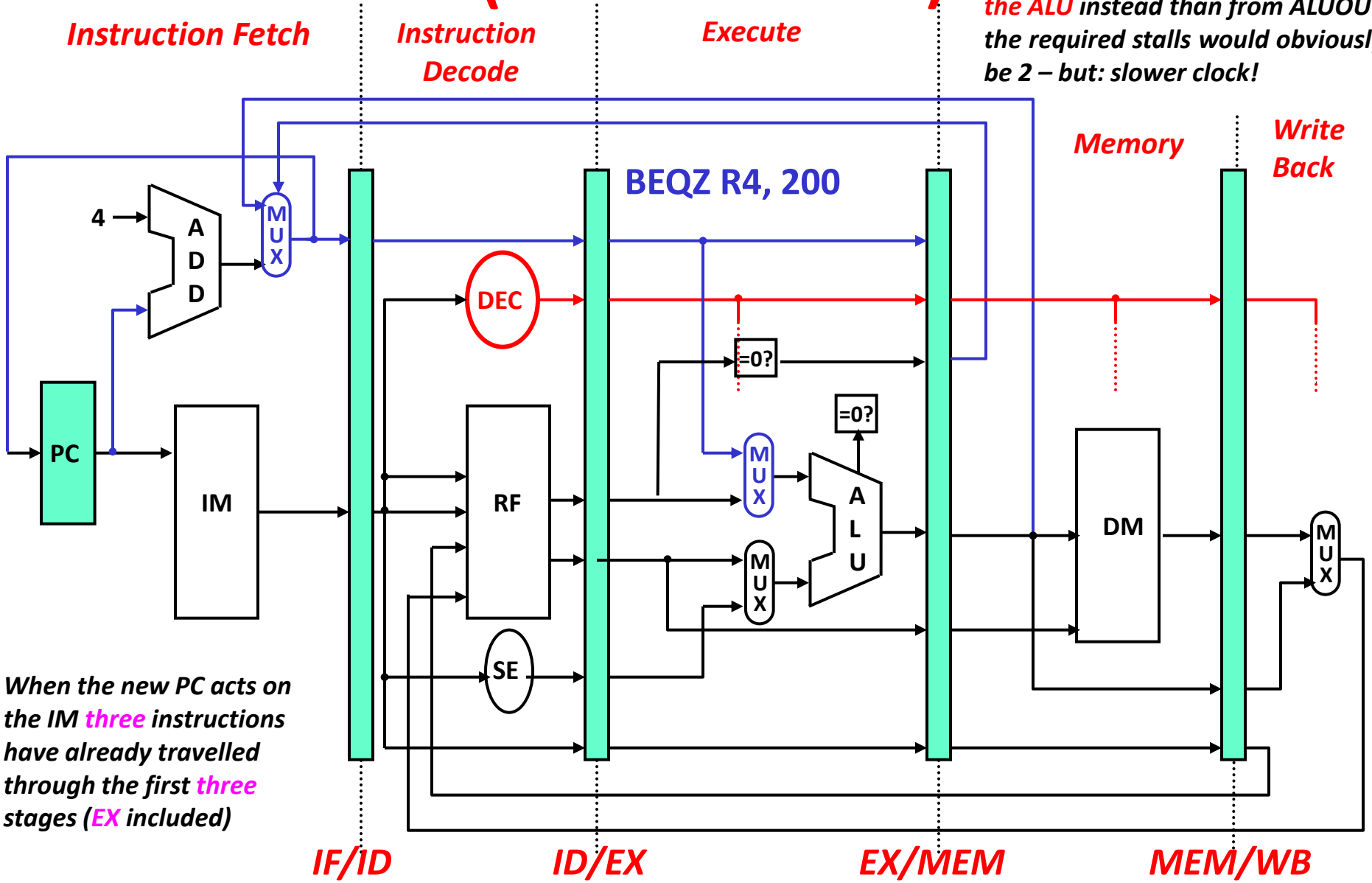
Next Instruction Address

R4 = 0 : ***Branch Target Address*** (taken)

R4 ≠ 0 : *PC+4* (not taken)



DLX Pipelined Datapath (Branch or JMP)

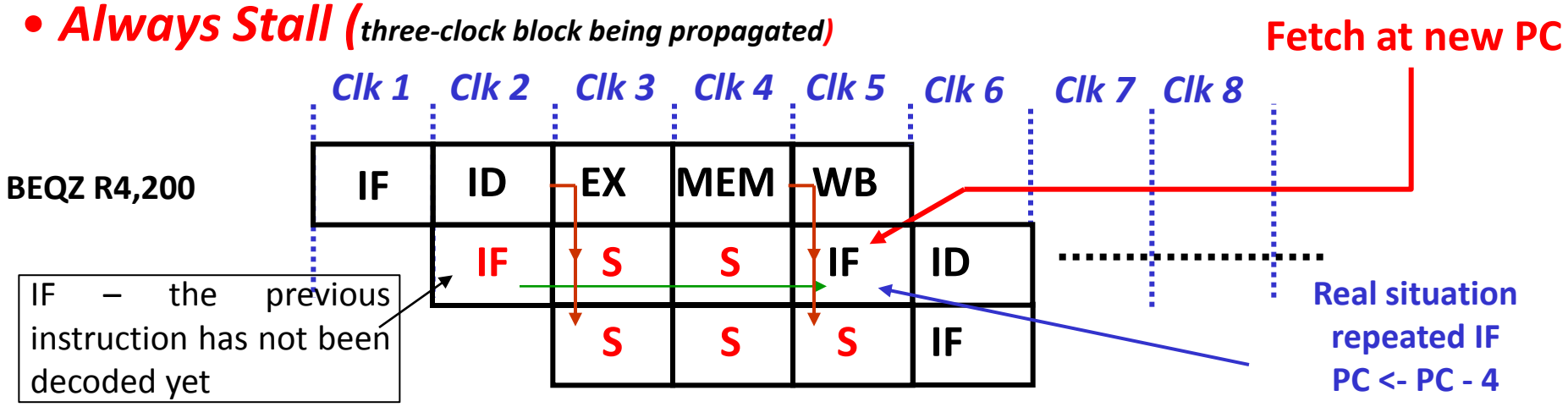


NOTE if the feedback signal of the new PC was taken directly from the ALU instead than from ALUOUT the required stalls would obviously be 2 – but: slower clock!

When the new PC acts on the IM three instructions have already travelled through the first three stages (EX included)

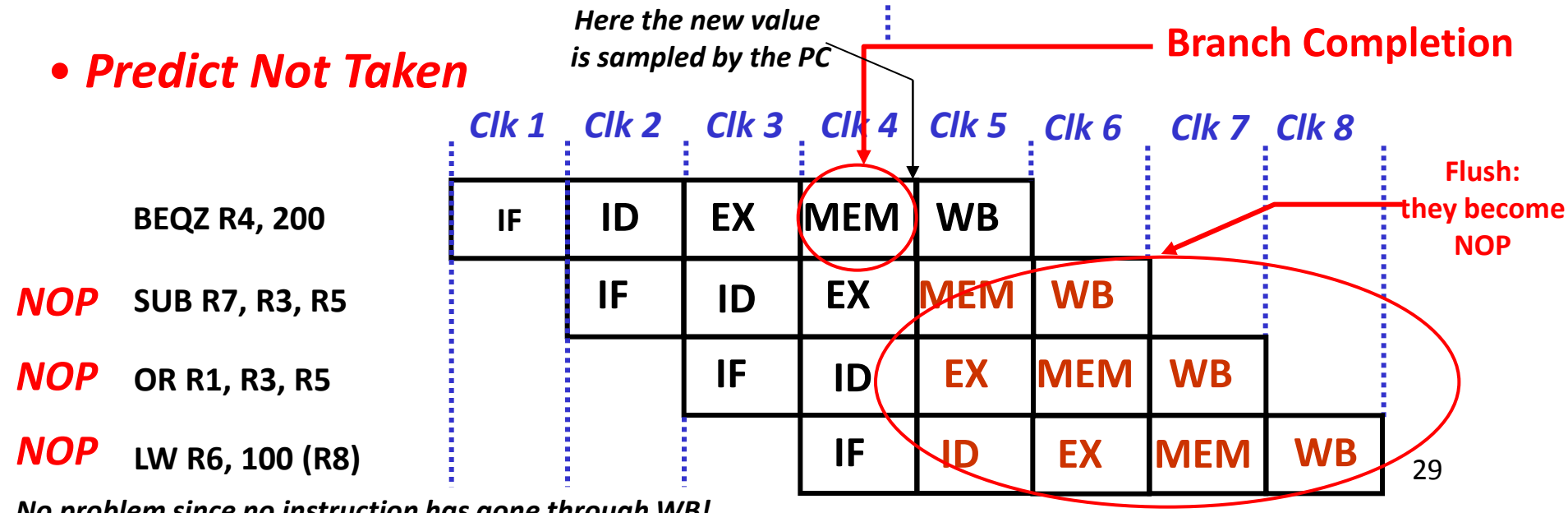
Handling the Control Hazards

- ***Always Stall (****three-clock block being propagated****)***



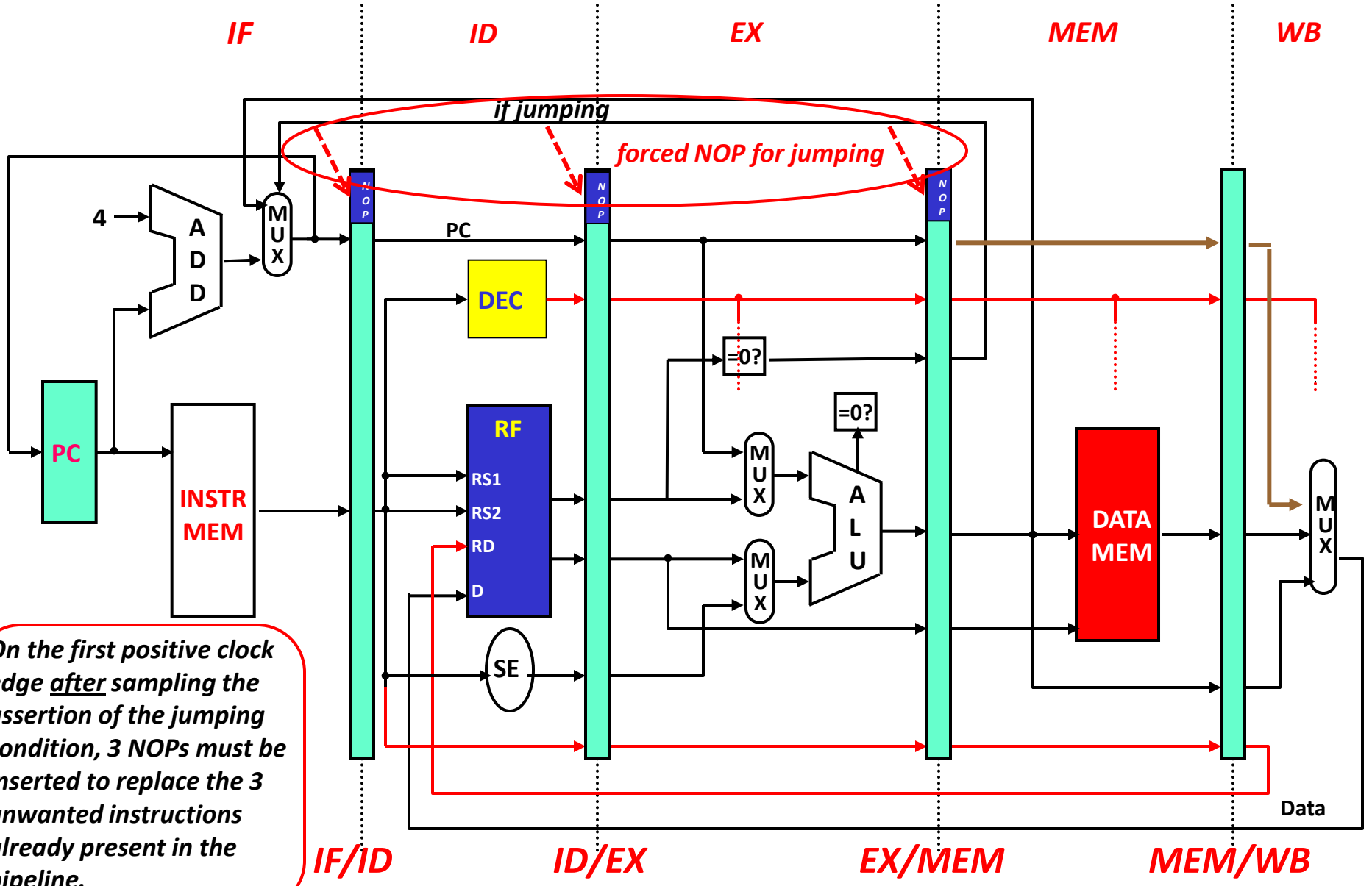
Hyp.: Branch Freq.= 25 % ⟶ $CPI = (1 + S) = (1 + 3 * 0.25) = 1.75$

- ***Predict Not Taken***



No problem since no instruction has gone through WB!

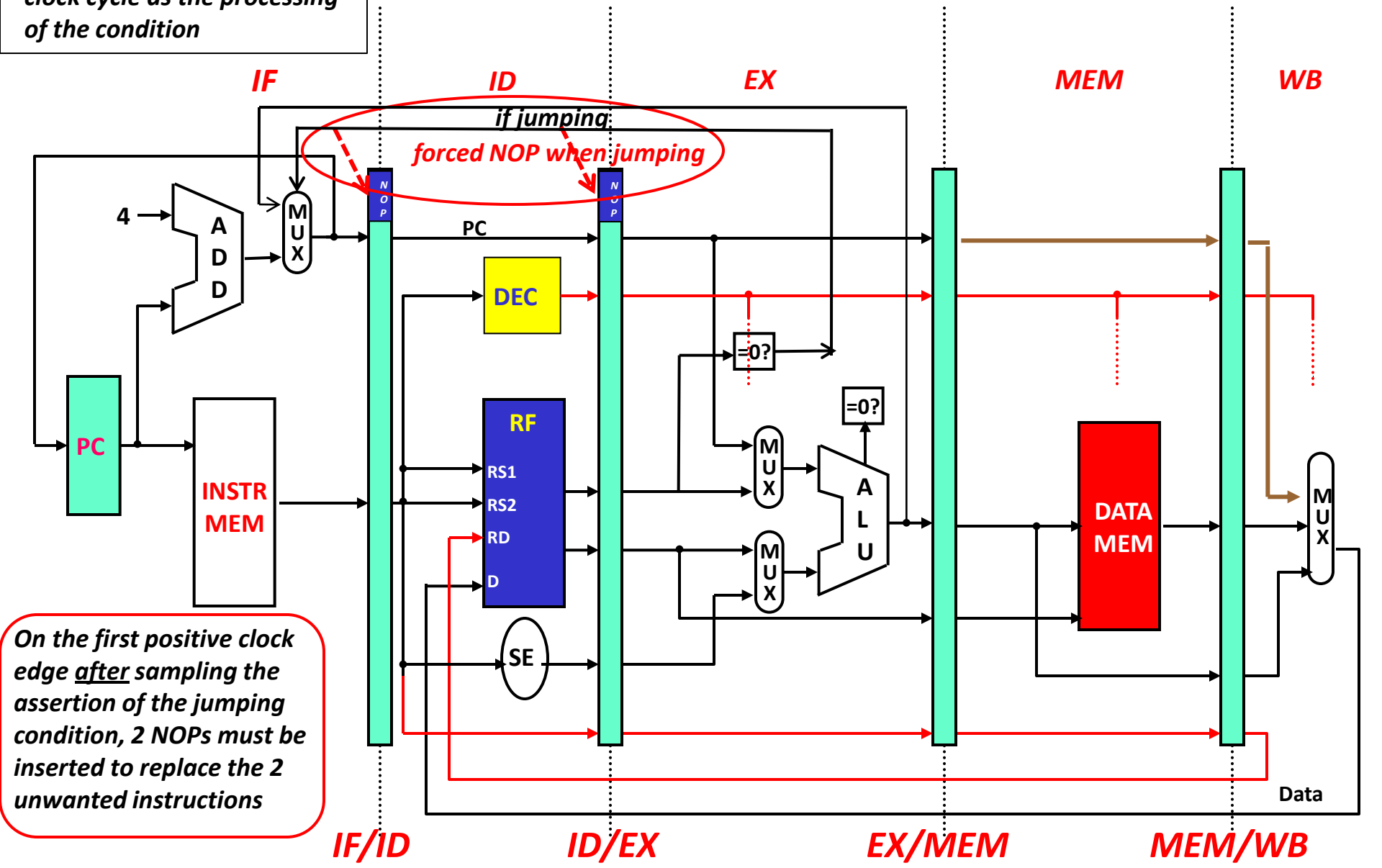
Stalls with jumps (1/3)



On the first positive clock edge <u>after</u> sampling the assertion of the jumping condition, 3 NOPs must be inserted to replace the 3 unwanted instructions already present in the pipeline.

NOTE in this case the jump condition and the new PC are sent to the MUX in the same clock cycle as the processing of the condition

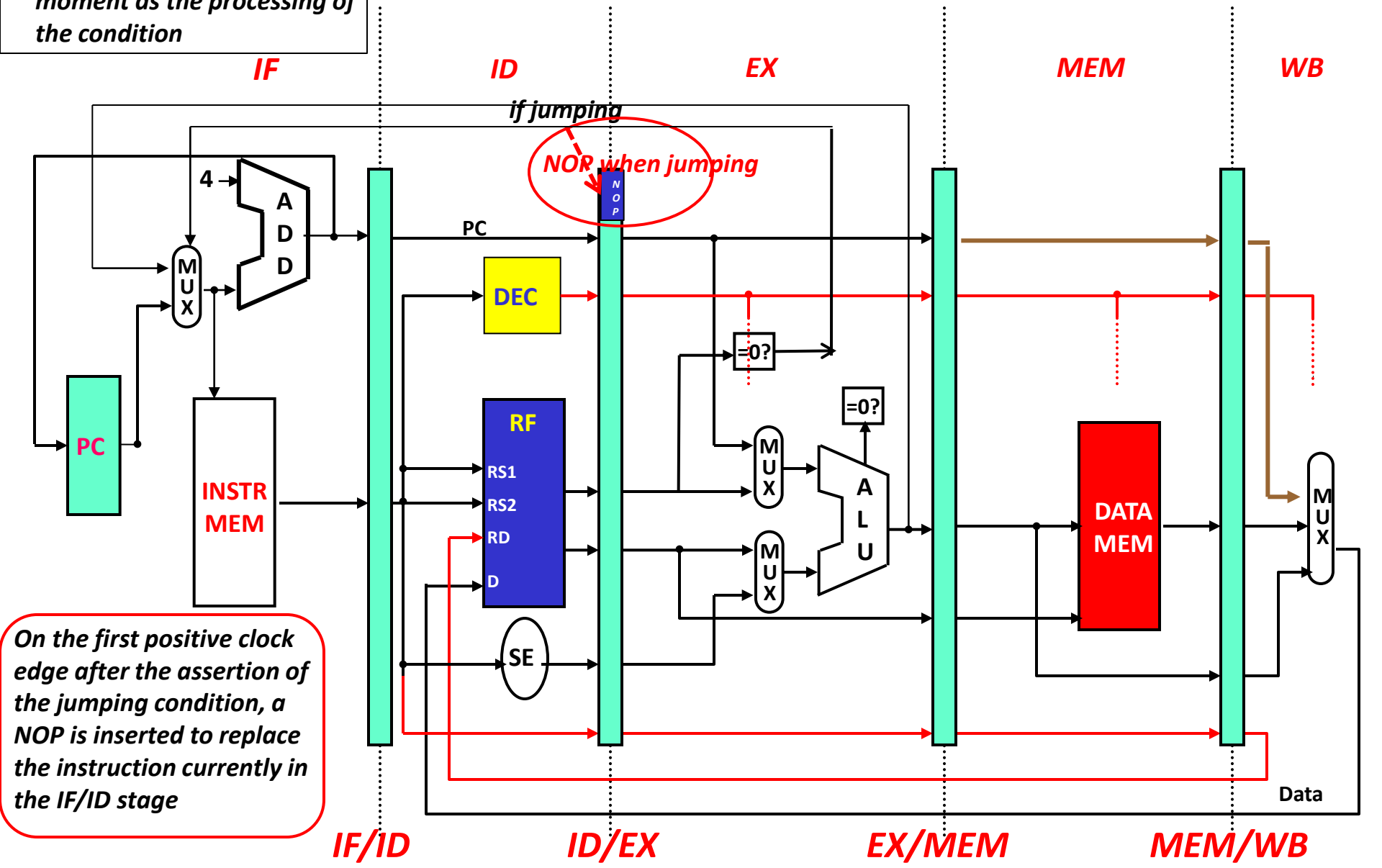
Stalls when jumping(2/3)



On the first positive clock edge after sampling the assertion of the jumping condition, 2 NOPs must be inserted to replace the 2 unwanted instructions

NOTE In this case the jumping condition and the new PC control the MUX in the same moment as the processing of the condition

Stalls when jumping (3/3)

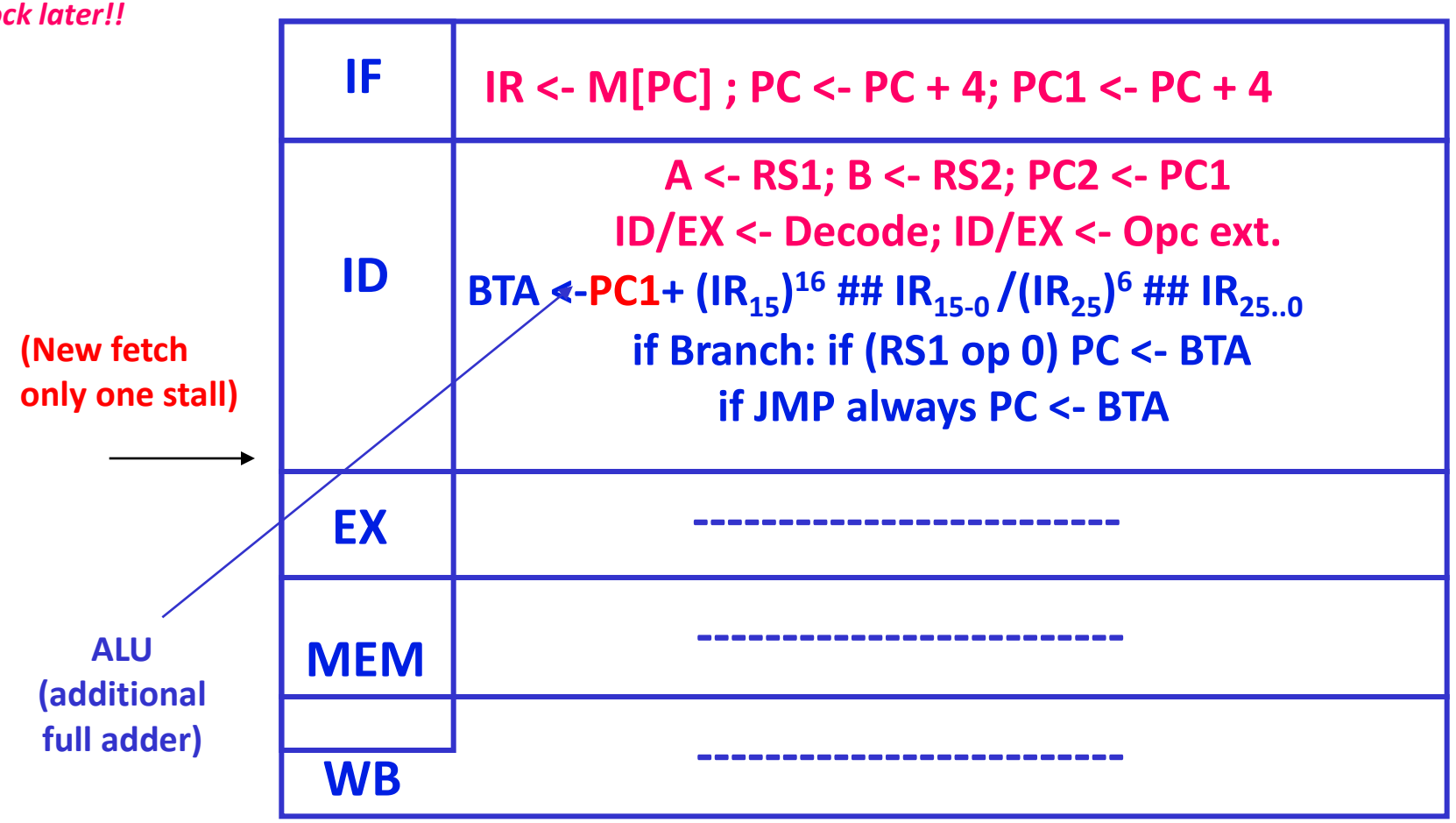


On the first positive clock edge after the assertion of the jumping condition, a NOP is inserted to replace the instruction currently in the IF/ID stage

NOTE here there is only one "stall" since the new value is inserted in the PC on the positive clock edge that ends the ID stage while, in the previous case, it was inserted after the MEM stage, that is, two clock later!!

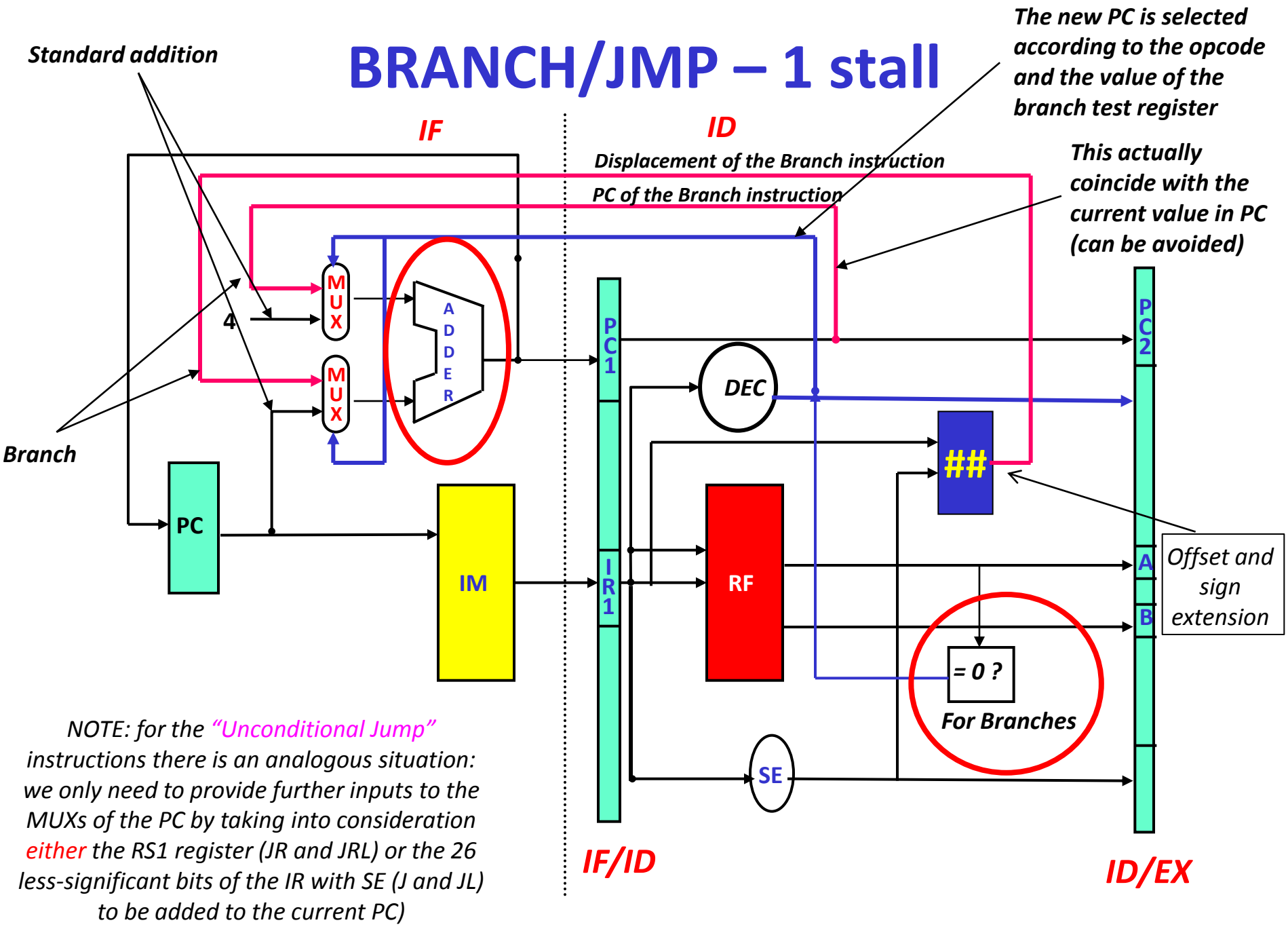
To reduce the number of stalls

Independent ALU for BRANCH/JMP



N.B. The full adder is separated from the adder "+4" (this means it overlaps with the addition required to compute the next instruction!), otherwise the same adder has to be used together with some multiplexers (so to select whether to add 4 or the offset, and whether to use PC or PC1)

BRANCH/JMP – 1 stall



NOTE: for the "Unconditional Jump" instructions there is an analogous situation: we only need to provide further inputs to the MUXs of the PC by taking into consideration either the RS1 register (JR and JRL) or the 26 less-significant bits of the IR with SE (J and JL) to be added to the current PC)

Delayed branch

Similarly to the LOAD case, with several RISC CPUs the hazard associated with BRANCH instructions is handled via SW by the compiler (*delayed branch*):

BRANCH instruction

delay slot

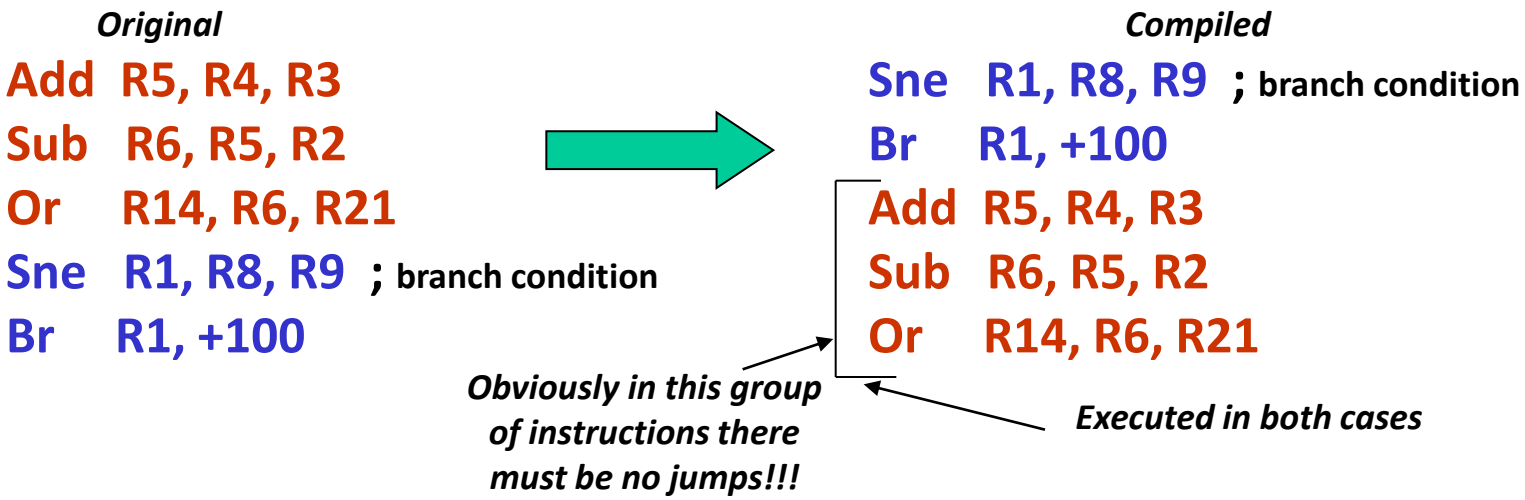
delay slot

delay slot

Next instruction

The compiler tries to
fill-in the delay-slots
with “useful” instructions
(worst case: NOP).

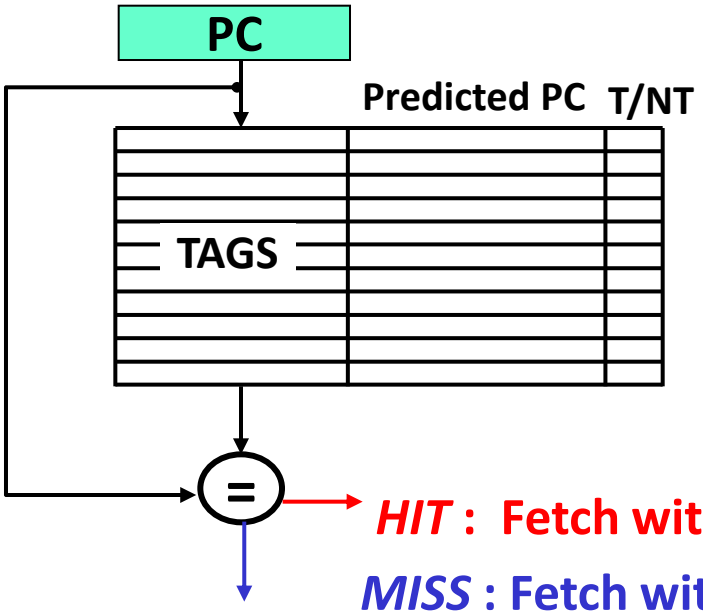
Delayed branch/jump



Instead of one or more "postponed" instructions, the compiler inserts NOPs in case no suitable instructions are available

Handling the Control Hazards

Dynamic Prediction: **Branch Target Buffer -> no stall (almost..)**



N.B. Here the branch slot is selected *during* the IF clock cycle that loads IR1 in IF/ID

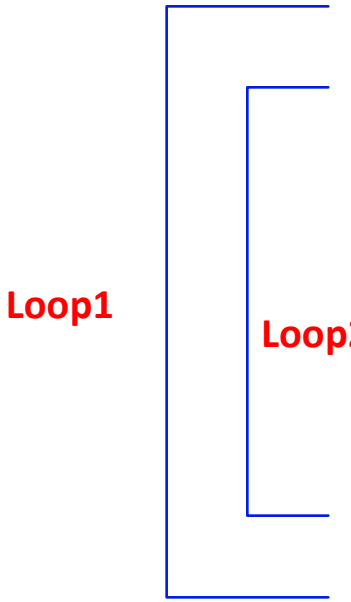
HIT **: Fetch with predicted PC**

MISS **: Fetch with PC + 4**

Correct prediction : no stall

Wrong prediction : 1-3 stalls (correct fetch in ID or EX, see before)

Prediction Buffer: the simplest implementation uses a single bit that indicates what happened when the last branch occurred.



When exiting loop2, the prediction fails (branch predicted as taken but actually it is untaken), then it fails again when it predicts as untaken whilst entering once again loop2

In case of predominance of one prediction, when the opposite situation occurs we have *two* consecutive errors.

Hence, usually *two* bits are used for branch prediction: