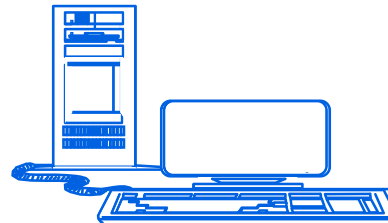
DLX computer

Electronic Computers M

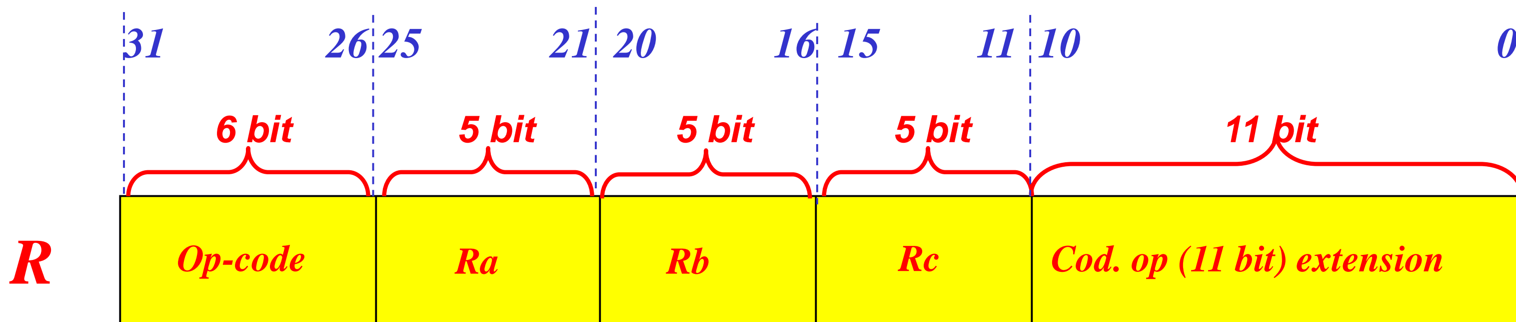


RISC architectures

- *RISC vs CISC (Reduced Instruction Set Computer vs Complex Instruction Set Computer*
- *In CISC architectures the 10% of the instructions are used in 90% of cases*
- *Waste of silicon*
- *Bottleneck: the bus*
- *Mid '80s a new architecture: RISC*
- *Solution: reduction of instruction number and complexity (fewer simpler machine instructions)*
- *Fixed instruction format (simpler instruction decoders)*
- *Simpler control logic network increasing the number of on-chip registers*
- *Reduction of bus/memory accesses*
- *Increase of machine instructions needed for a job which is (in many cases) more than compensated (in term of time) by the reduction of bus accesses*
- *CISC and RISC are each one the best solution in different application fields*
- *Nowadays coexistence of both architectures in the same processor: analysis at the end of the course*
- *A simplified RISC architecture: DLX (implemented as real processor in the '80s as R4000)*

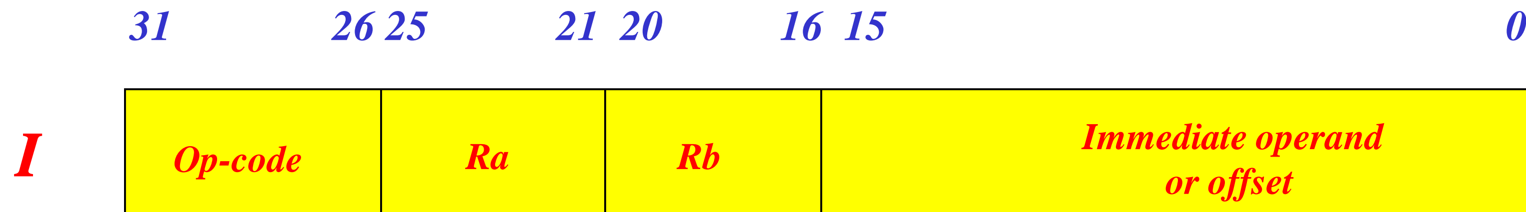
DLX (fixed) instruction format

R



Arithmetic or logic instructions; i.e. Ra ← Rb op Rc or Set Conditions between registers Branch instructions

I



Data transfer (Load, Store), conditional Branch , JR and JALR (Control transfer via register), Set Condition e ALU with immediate operator. In Load and ALU instructions Ra=destination, in the Store Ra=source. -- Rb content as ALU value for the immediate instructions - Branch instructions

J

Direct, unconditional control transfer(J e JAL)

DLX non floating-point instructions

(31x32bit registers R31…R1 - R0=0 fixed - Ra and Rb any of the 32 registers)

Data Transfer

LW	Ra, offset(Rb)
LB	Ra, offset(Rb)
LBU	Ra, offset(Rb)
LHU	Ra, offset(Rb)
LH	Ra, offset(Rb)
SW	Ra, offset(Rb)
SH	Ra, offset(Rb)
SB	Ra, offset(Rb)
LHI	Ra, value

Arithmetic/Logic

ADD	Ra,Rb,Rc
ADDI	Ra,Rb,value
ADDU	Ra,Rb,Rc
ADDUI	Ra,Rb, value
SUB	Ra,Rb,Rc
SUBI	Ra,Rb,value
SUBU	Ra,Rb,Rc
SUBUI	Ra,Rb, value
DIV	Ra,Rb,Rc
DIVI	Ra,Rb,value
MULU	Ra,Rb,Rc
MULI	Ra,Rb, value
SLL	Ra ,Rb,Rc
SLLI	Ra,Rb;value
SHR	Ra,Rb.Rc
SHRI	Ra,Rb,value
SLA	Ra,Rb,Rc
SLAI	Ra,Rb,value
OR	Ra,Rb,Rc
ORI	Ra,Rb,value
XOR	Ra,Rb,Rc
XORI	Ra,Rb,value
AND	Ra,Rb,Rc
ANDI	Ra,Rb,value

Control

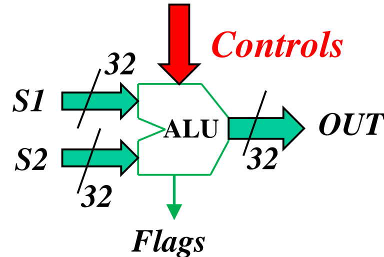
SETx	Ra,Rb,Rc
SETIx	Ra,Rb,value
BEQZ	Ra, offset (- - - +[PC])
BNEQZ	Ra, offset (- - - +[PC])
J	offset
JR	Ra
JL	offset (- - - +[PC])
JLR	Ra

N.B.
Postfix x (set condition) can be LT, GT, LE, GE, EQ, NE
JL (via or non via register) -> Jump and link saving PC in R31
Offset is a value within the instruction
Postfix I means «immediate» (value within the instruction)
PostfixA means «arithmetic» (sign extension)
Postfix U means «unsigned»
Value is the immediate within the instruction

No STACK registers

DLX ALU operations

Two inputs data One output data plus flags



S1 , S2 : ALU inputs (32 bit)

S1 + S2
S1 – S2
S1 and S2
S1 or S2
S1 exor S2
Left Shift S1 of S2 positions
Right Shift S1 of S2 positions
Arithmetic Right Shift S1 of S2 positions
S1
S2
0
1

Output Flags

Zero
Negative sign

ALU is a combinatorial circuit !!!

Sequential DLX

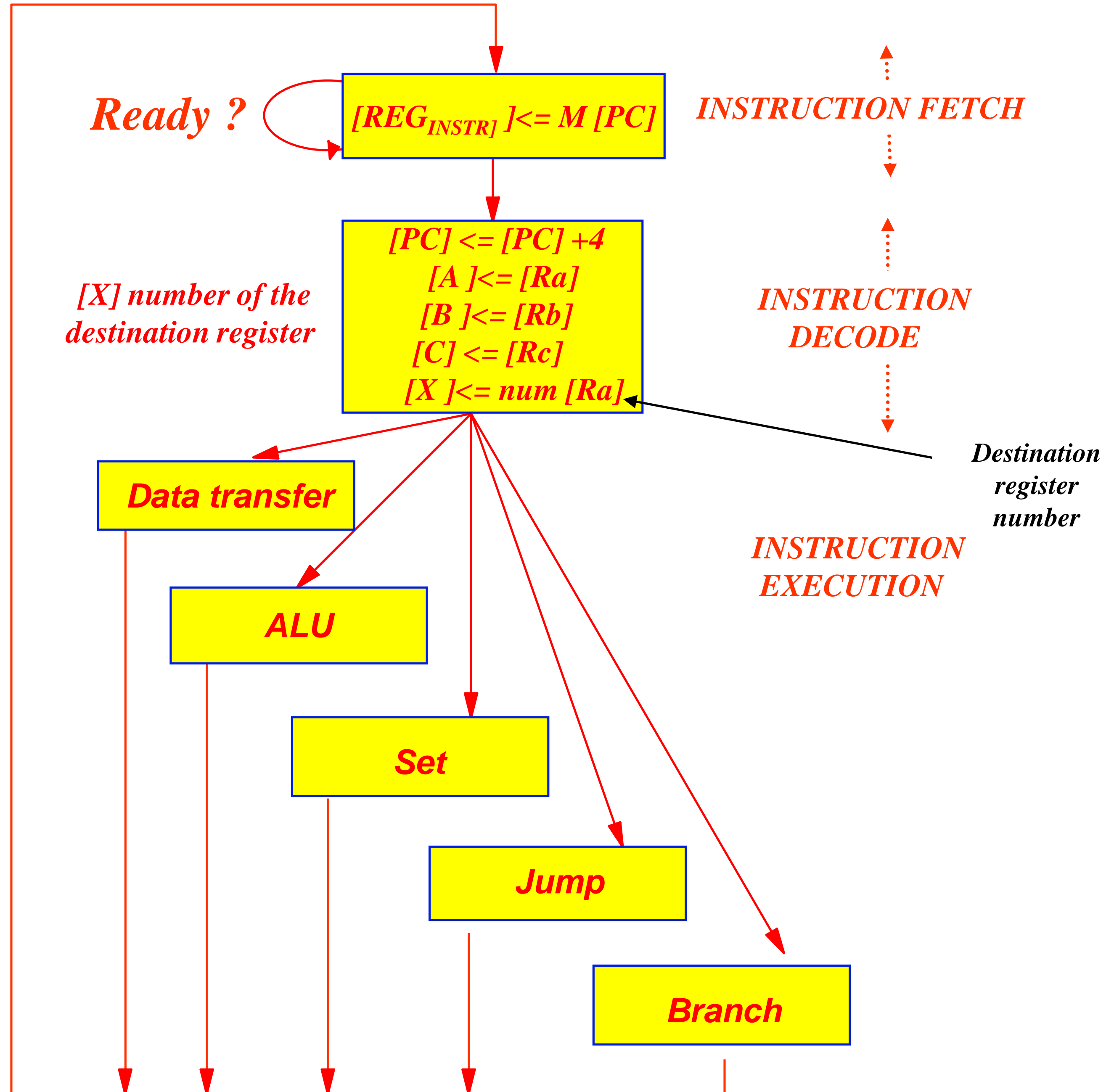
Abstract

instruction execution

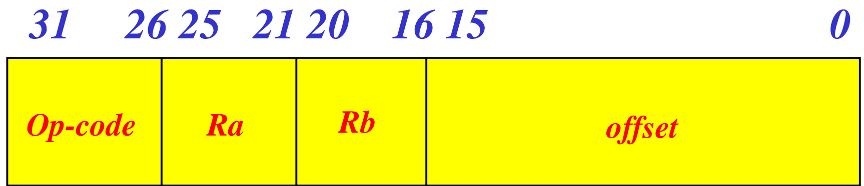
PC is the Program Counter
REG_{instr} is the register where the new fetched instruction is stored.

*A and B are two scratchpad internal registers, All these registers are **unknown** to the programmer*

This is a synchronous state diagram



Example: LB
(LOAD BYTE format I)



LB Ra, offset(Rb)

$Instr_{15.0.}$ is the instruction offset
Address is always 32 bit

31 MBbit ↔ **0 LSbit**

Sign extension !!

Example

$M[Addr]_{7..0} = A7_H \Rightarrow (10100111)_b$

Sign extended address <= $FFFFFFA7_H$

$I_{NSTR} \Leftarrow M[PC]$

$[PC] \Leftarrow [PC] + 4$
$[A] \Leftarrow [Ra]$
$[B] \Leftarrow [Rb]$
$[C] \Leftarrow [Rc]$
$[X] \Leftarrow num [Ra]$

Instruction bit 15 (sign) is left extended 16 times

LOAD

=> JOIN operator

$Byte\ Addr. \Leftarrow [B] + (Instr_{15})^{16}\ \#\#\ Instr_{15..0}$

Byte address compute

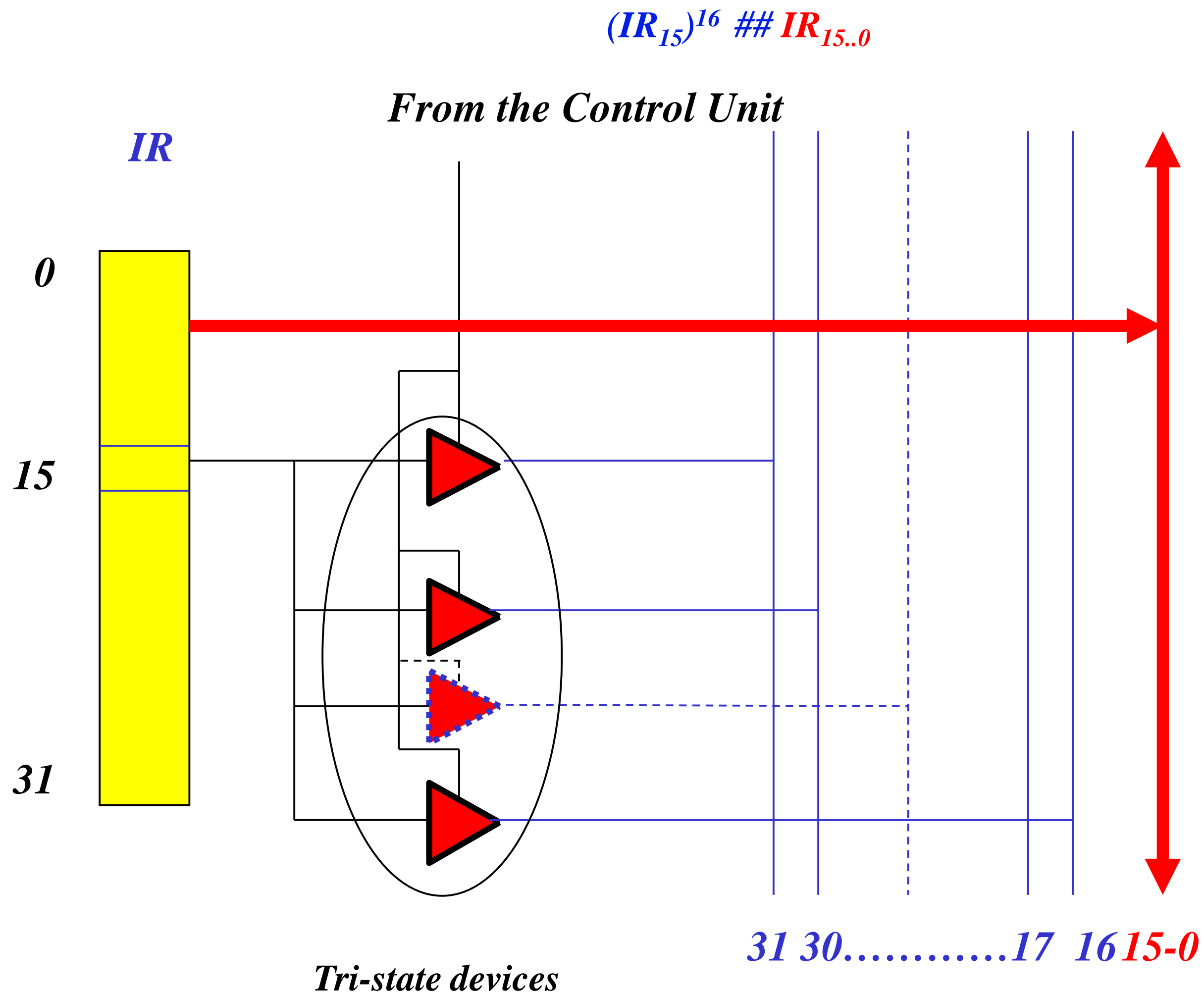
Sign extension

$[Ra] \Leftarrow (M[Addr.]_7)^{24}\ \#\#\ M[Addr.]_{7..0}$

Byte in register

Next Instruction

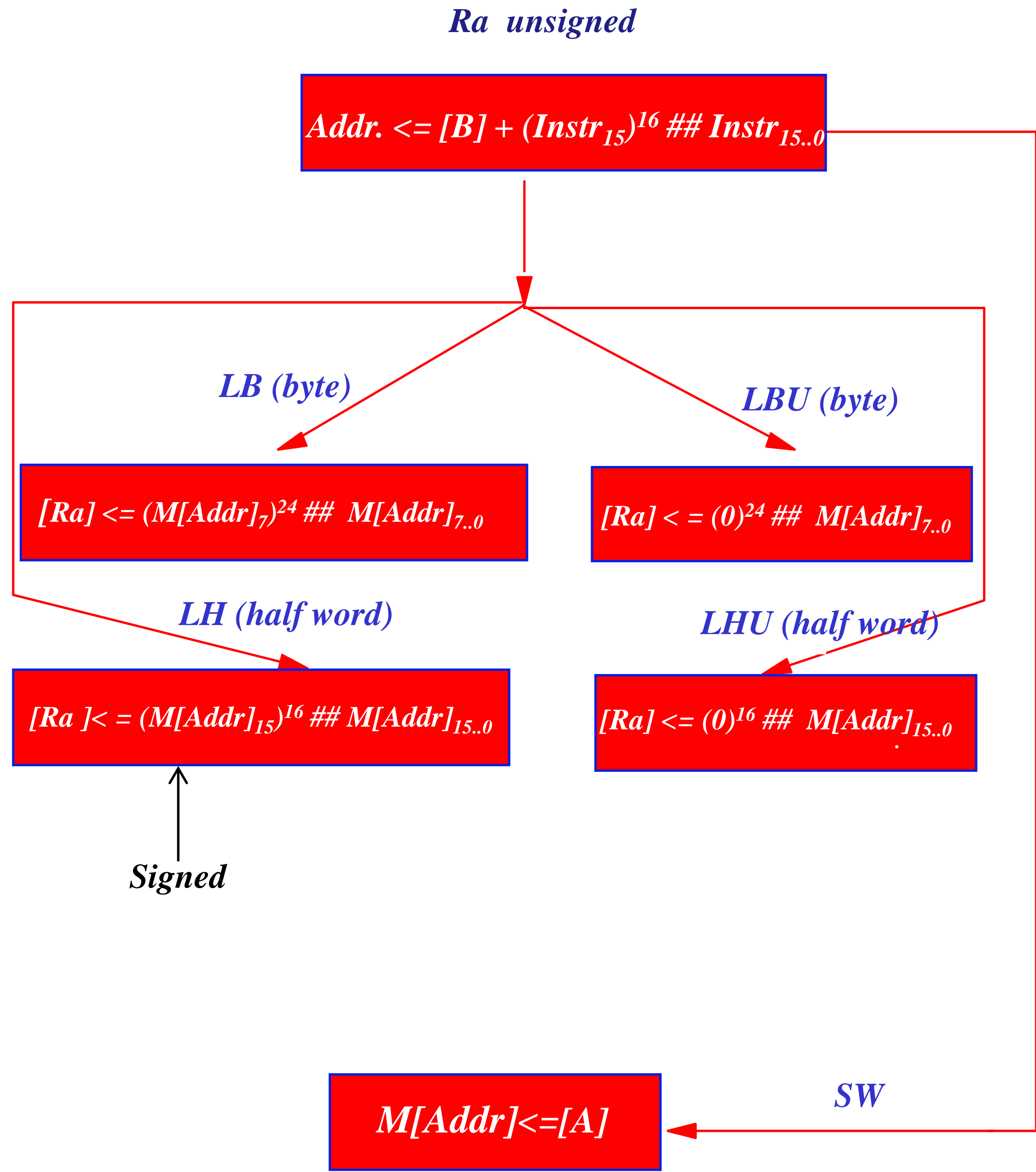
Sign extension - example with IR



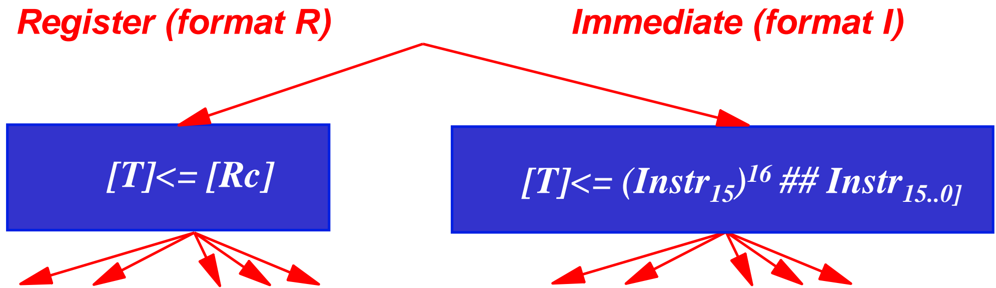
Data transfer Instructions (R format)

Examples

LW	*Ra, offset(Rb)*	
LB	*Ra, offset(Rb)*	
LBU	*Ra, offset(Rb)*	*unsigned*
LHU	*Ra, offset(Rb)*	*unsigned*
SW	*Ra, offset(Rb)*	

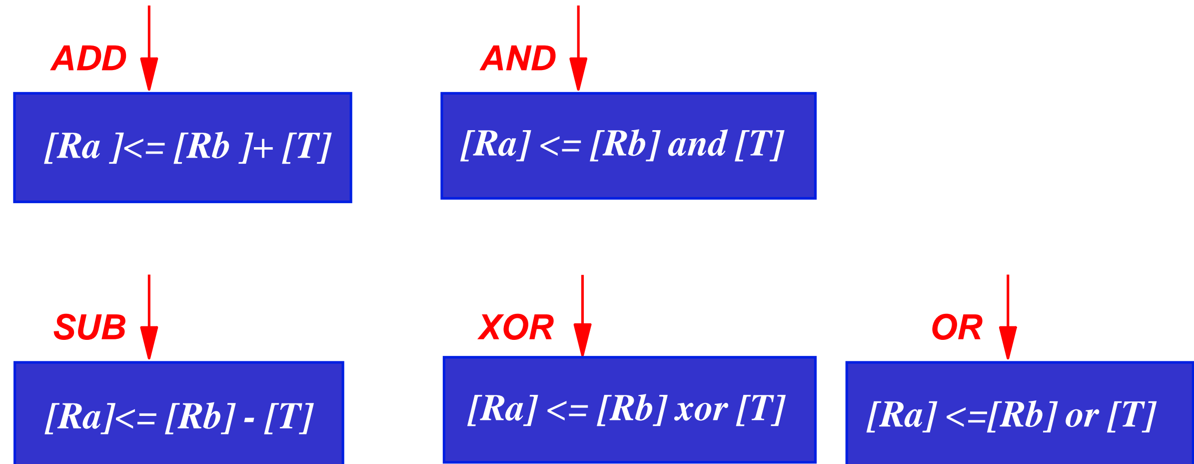


ALUinstructions examples (I format)



(T is a hidden register unknown to the programmer storing temporary data)

Register content signed if arithmetic operations

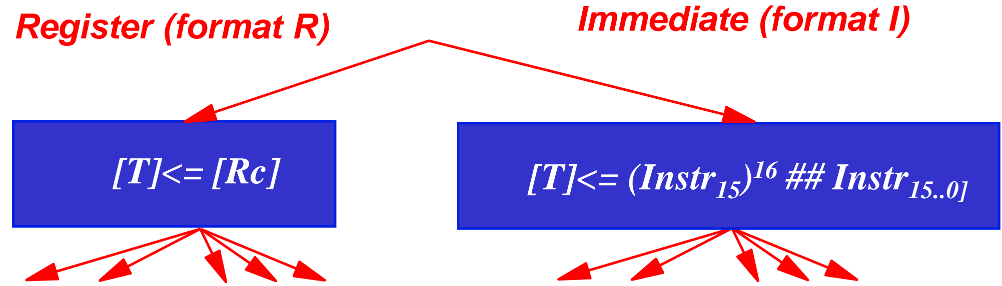


ADD Ra,Rb,Rc
ADDI Ra,Rb,value
ADDU Ra,Rb,Rc
ADDUI Ra,Rb, value
……………………….

The same scheme for the shift etc.
A and B generic registers (Ra, Rb)

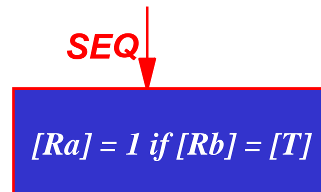
SET instructions
(*see branch*)

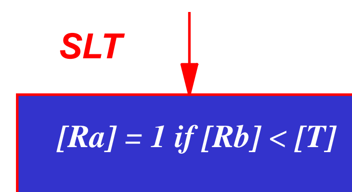
ex. SLT Ra,Rb,Rc
Set Ra=1 if Rb is less than Rc
otherwise Ra=0

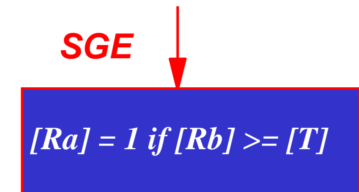


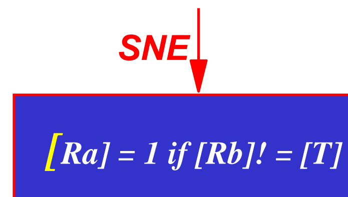
Register content as signed

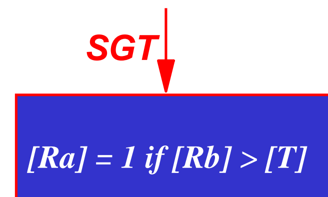
(T is a hidden register unknown to the programmer storing temporary data)

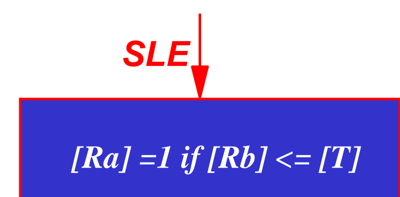




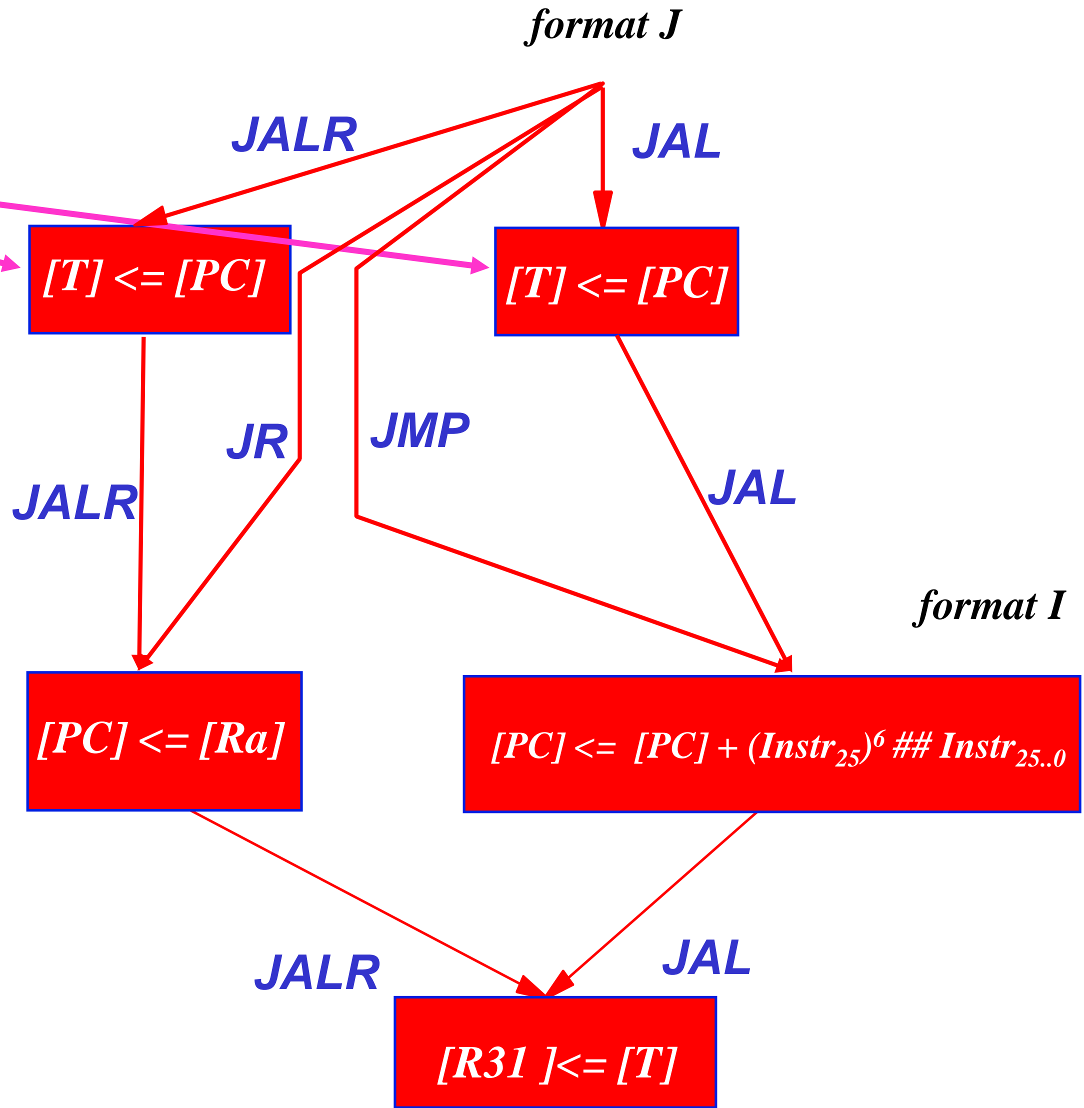








For saving [PC] in R31

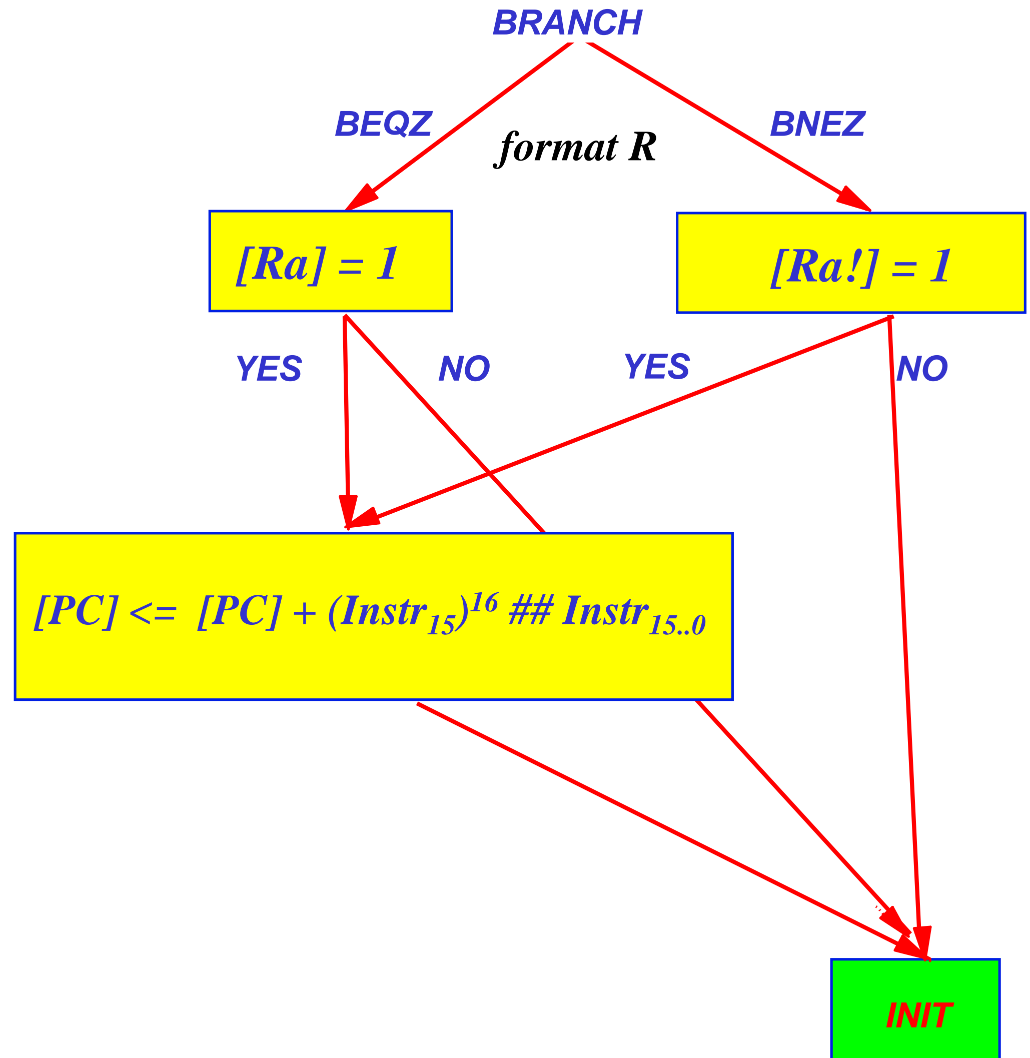


JUMP Instructions

J	*offset*	*(jump address)*
JR	*Ra*	*(jump register)*
JL	*offset*	*(jump and link address)*
JLR	*Ra*	*(jump and link register)*

Branch Instructions

Ex. BNEQZ R5, 100
Jump to PC+100 if R5 not equal 0

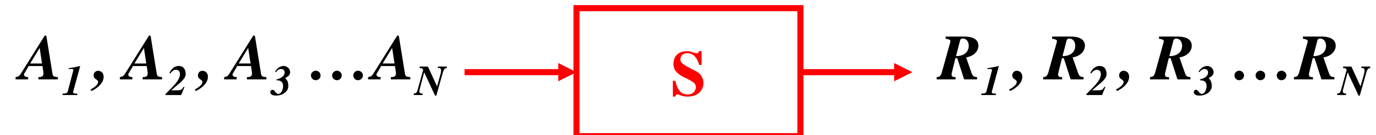


The *Pipelining* Principle

Pipelining is the main basic technique used for "*speeding-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* must operate N times on a task A_i producing result R_i :

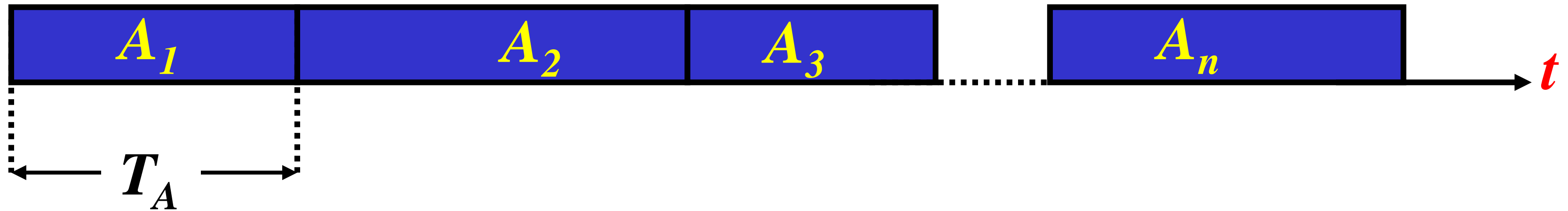


Latency : time occurring between the beginning and the end of task A (T_A).

Throughput : frequency of each task completion

The *Pipelining* Principle

1) Sequential System - A new instruction starts when the previous instruction is finished



A_n n-th instruction - Latency (execution time of a single instruction) = T_{An}

Different execution times

2) Pipelined System (instruction are subdivided in stages – each stage during one n_{th} – 1/4 in this example - of the entire instruction time)

Successive instructions stages *overlap*

A

P_1 P_2 P_3 P_4 *t*

S_1 S_2 S_3 S_4

S

S_i: pipeline stage

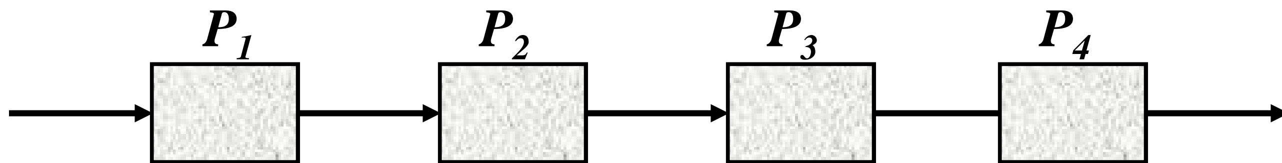
The *Pipelining* Principle

T_P: *pipeline cycle (ideally one clock)*

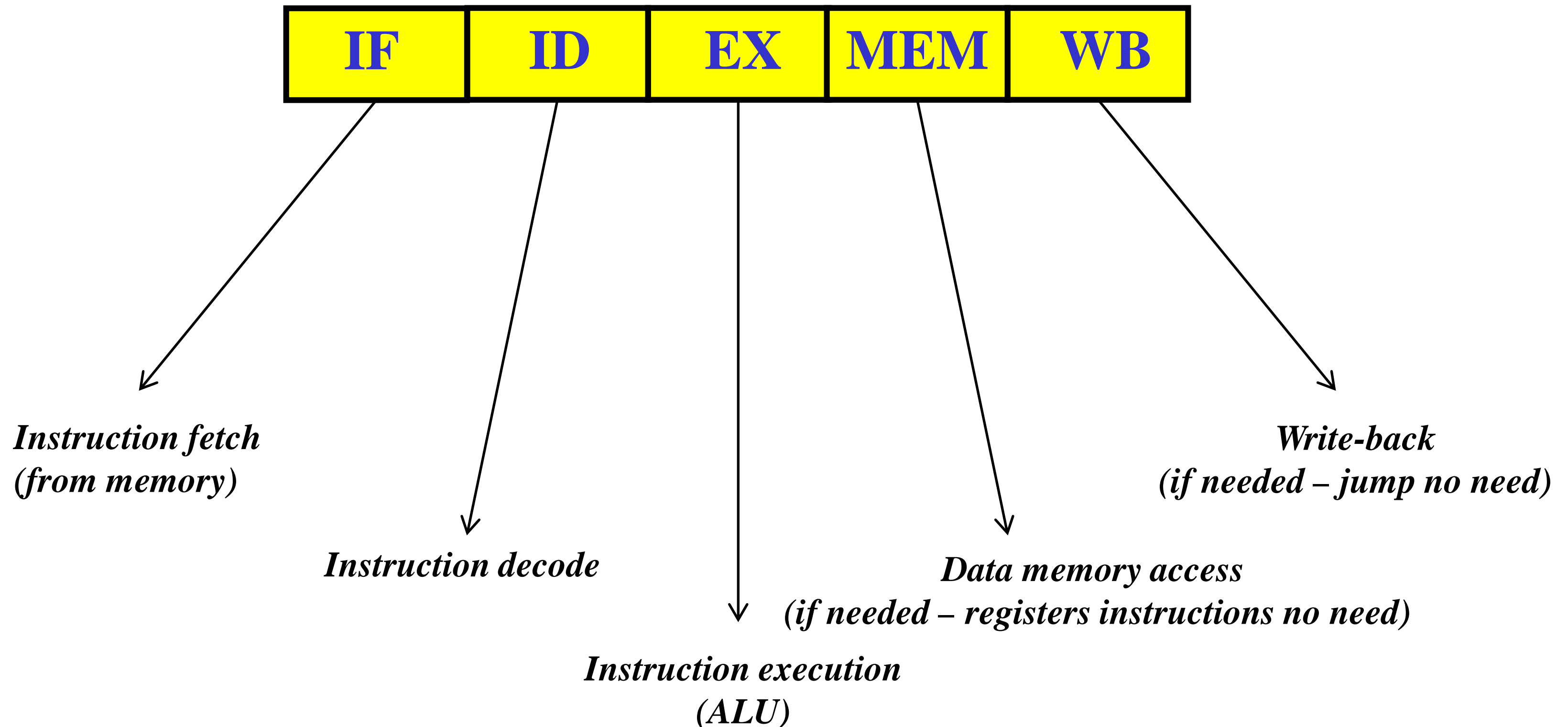
For each cycle one instruction terminates

In figure A1 terminates at t_x

Next cycle A2 terminates at t_y *etc.*



Typical instruction stages

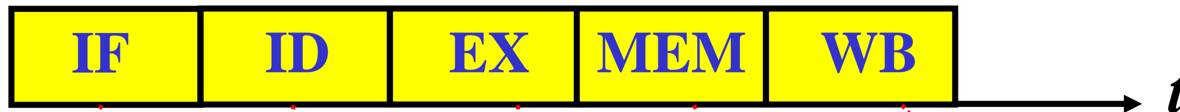


N.B. The execution time (latency) of all instructions must be the same, for maintaining the results order. Some stages are not used for some instructions (the stage is a NOP for them) – i.e. the stage MEM for register operations)

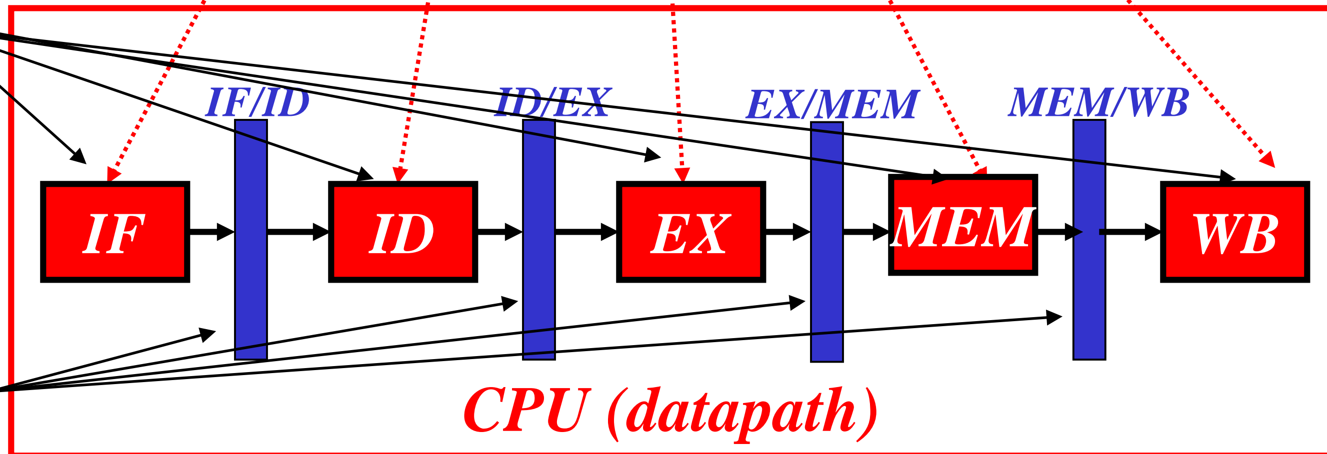
Pipelining of a CPU (*DLX*)

Instruction sequence: $I_1, I_2, I_3 \ldots I_N$

Instruction j



Combinatorial circuits



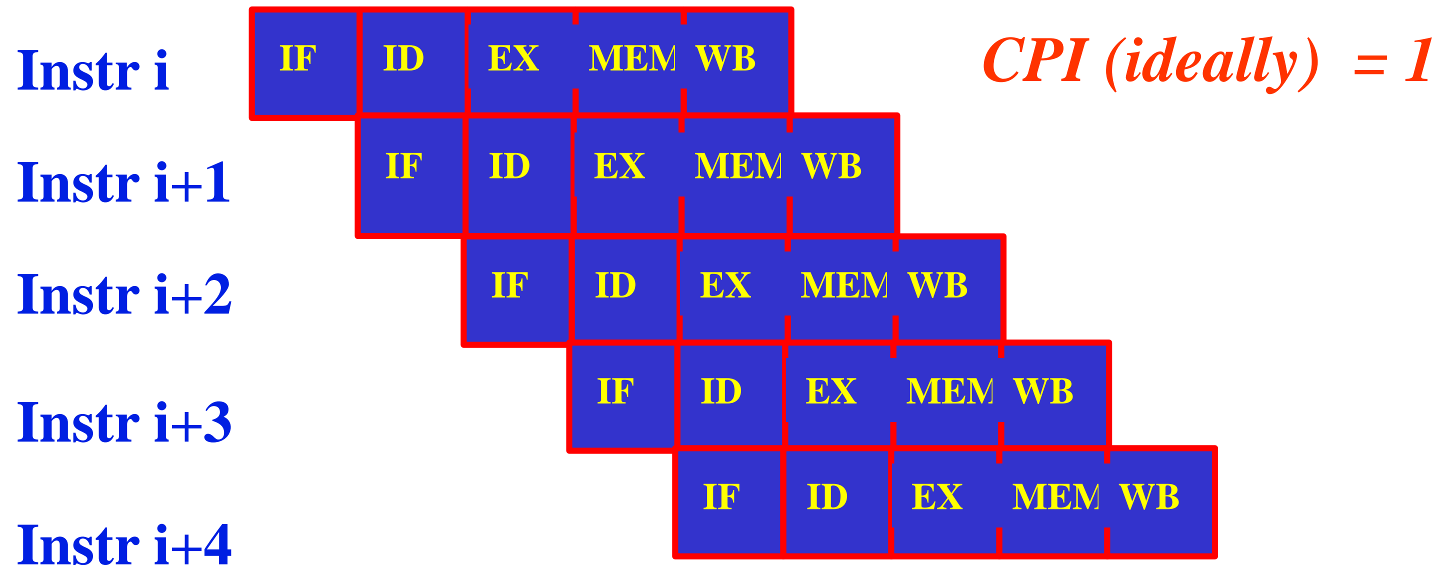
Registers (Pipeline Registers D FF)

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

↓

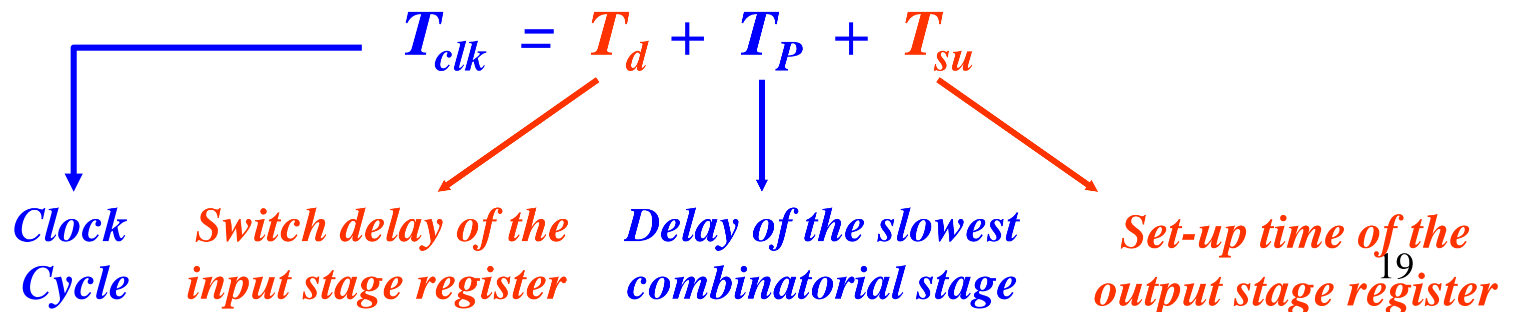
ClockPerInstruction (CPI)=1 (ideally !)

DLX Pipeline

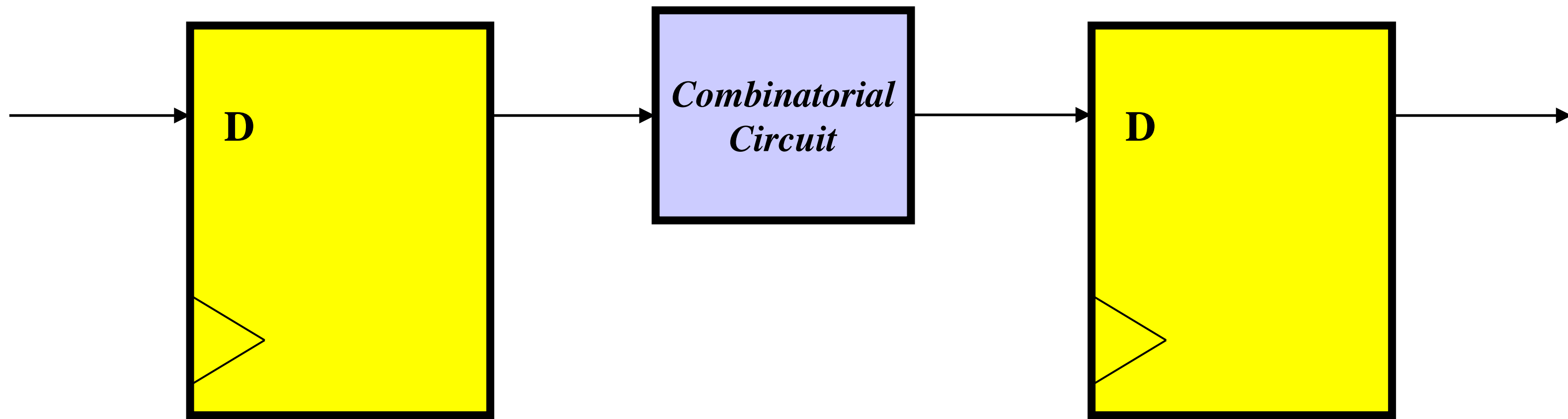


CPI (ideally) = 1

Overhead introduced by the Pipeline Registers:



Tp



Switch delay of the input stage register

Delay of the slowest combinatorial stage

Set-up time of the output stage register

Pipeline implementation requirements

- **Each stage is active at each clock cycle.**

- **The PC is incremented in the IF stage.**

- **An ADDER should be introduced (PC <=PC+4 – one instruction is 4 bytes) in the IF stage. But instructions are aligned (each one ends to an address multiple of the instruction length in bytes) and therefore a 30 bit only register (a programmable counter for jumps) is used, incremented by 1 each clock cycle**

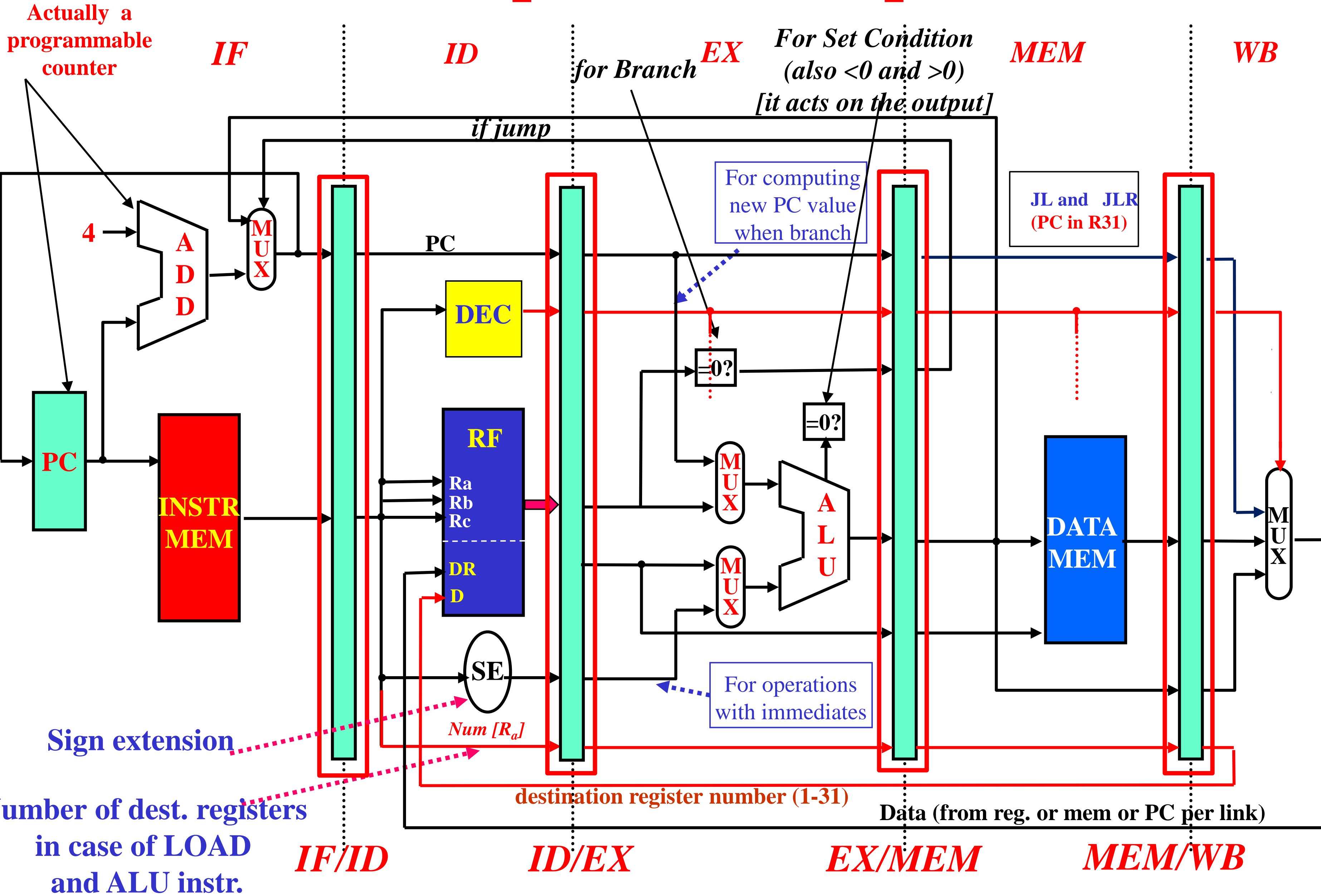
- **Two Memory Data Registers are required (referred to as LMDR e SMDR). In fact when a LOAD is immediately followed by a STORE there is a WB/MEM stages overlap – two data waiting therefore to be written (one onto the memory, the other onto a register of the RF).**

- **Each clock cycle 2 memory accesses must be possibly executed (IF, MEM): Instruction Memory (IM) and Data Memory (DM):** ***"Harvard" Architecture***

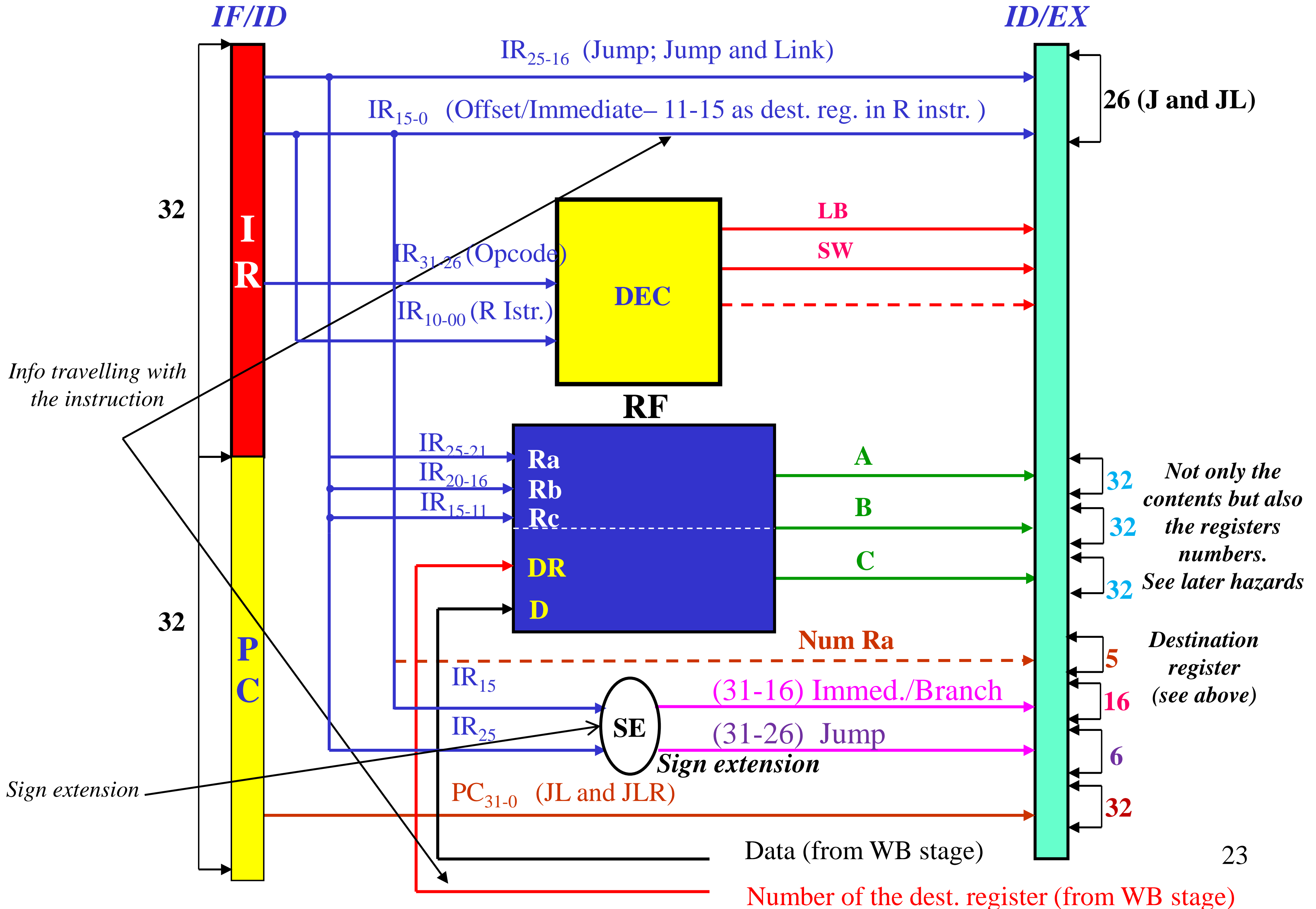
- **The CPU clock is determined by the slowest stage**

- **Pipeline Registers store both data and control information ("*distributed*" control unit)**

DLX Pipelined Datapath

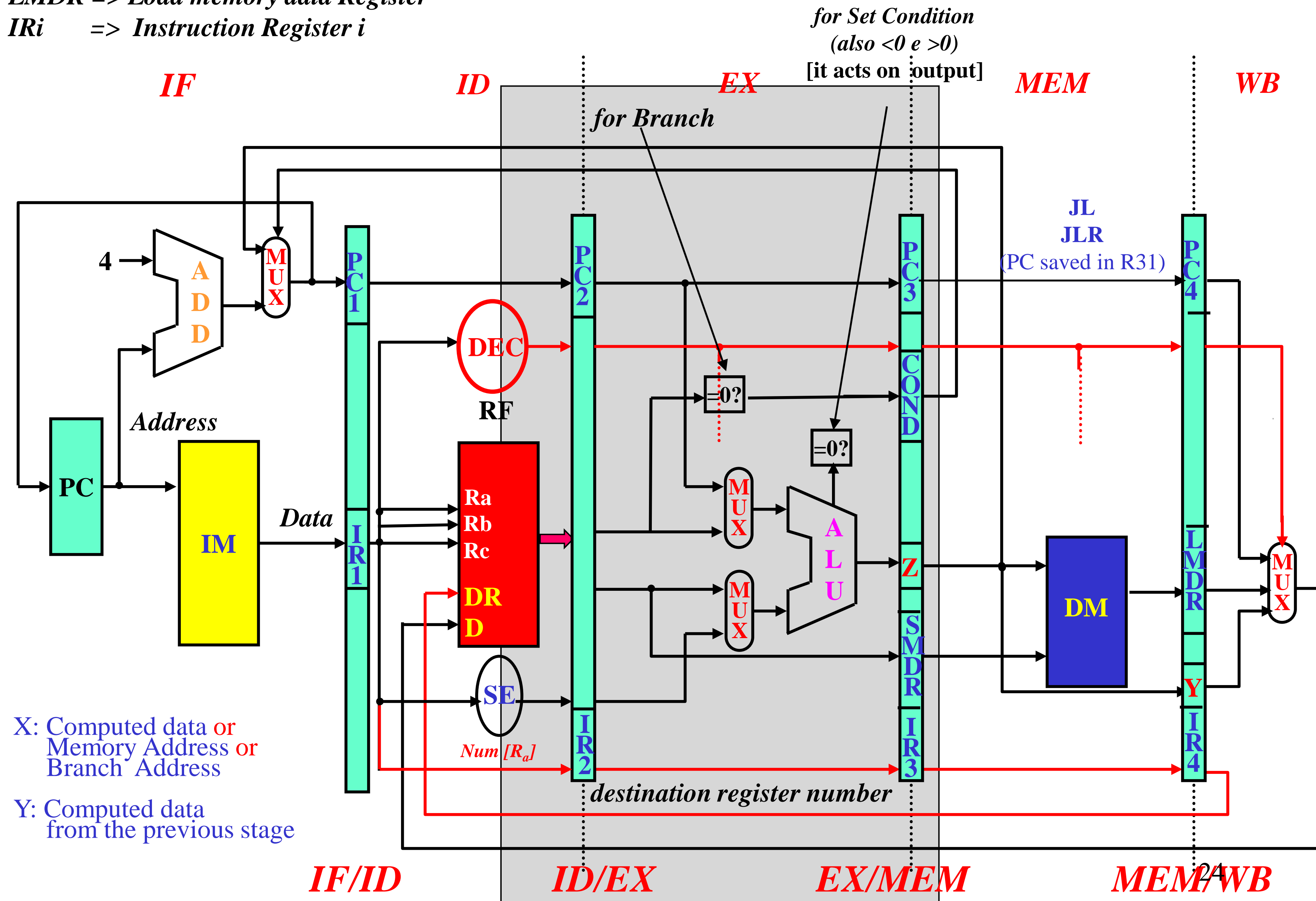


ID stage (N.B. stage layout different from previous slide!)



SMDR => Store Memory Data Register
LMDR => Load memory data Register
IRi => Instruction Register i

DLX Pipelined Datapath



Pipelined execution of an "ALU" instruction
The result of each stage is sampled at the end of its cycle

Stage	Operations
IF	IR <= M[PC] ; PC <= PC + 4 ; PC1 <= PC + 4
ID	A <= Ra; B <= Rb;C<=Rc PC2 <= PC1; IR2<=IR1 ID/EX <= Instruction decode; [X]<= num[Ra]
EX	Z<= A op B [PC3 <= PC2] [IR3 <= IR2] or $Z <= A\ op\ [(IR2_{15})^{16}\ \#\#\ IR2_{15..0}]$
MEM	Y <= Z (temporary storage for WB) [IR4 <.= IR3] [PC4 <= PC3]
WB	Ra <= Y

Decoded opcode travels through all stages

NOTE: *IRi bits which are dropped stage by stage when no more needed for all instructions.*

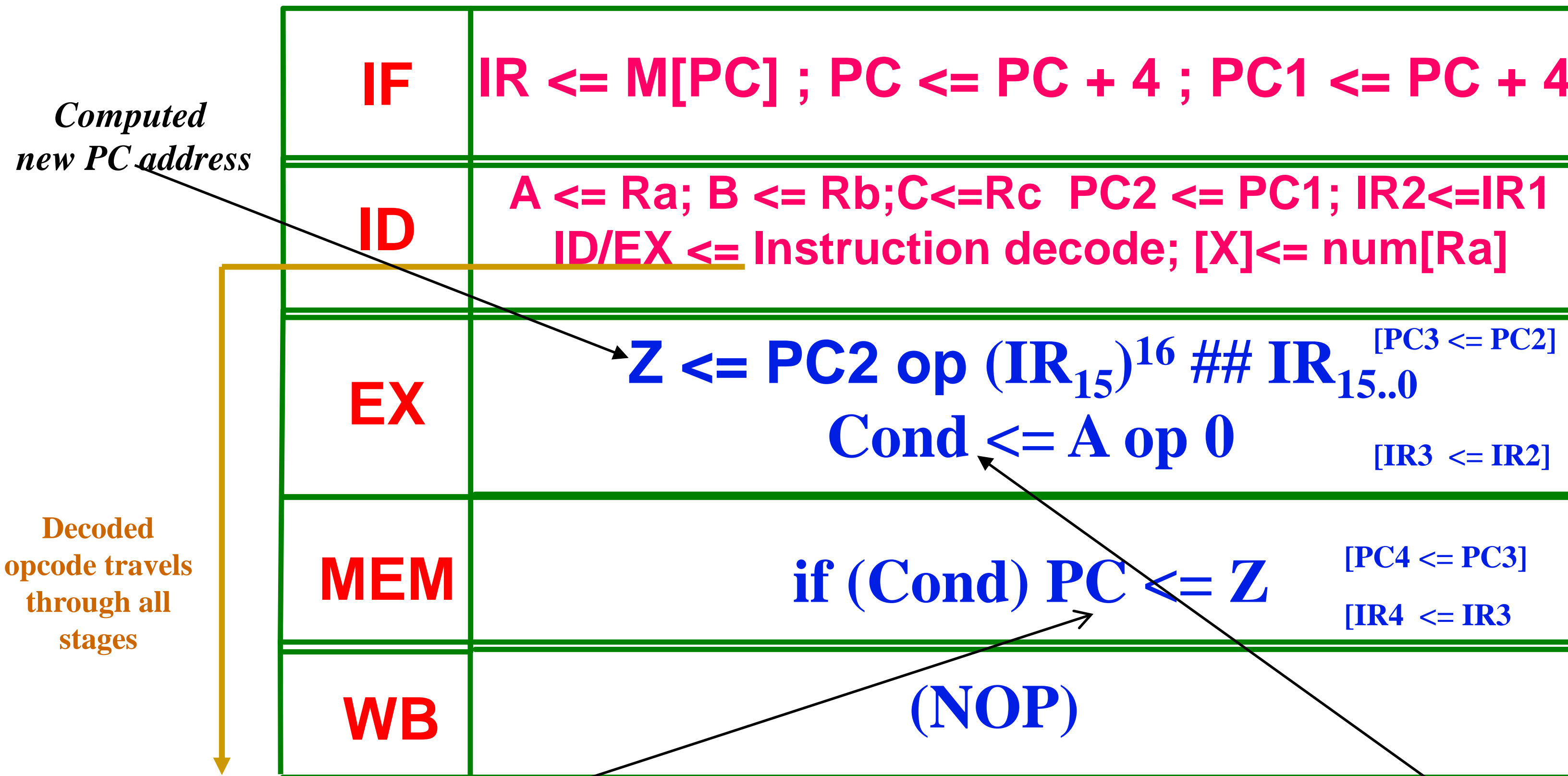
Why ?

JAL, JALR!!

Pipelined execution of a "MEM" instruction

Stage	Operations
IF	IR <= M[PC] ; PC <= PC + 4 ; PC1 <= PC + 4
ID	A <= Ra; B <= Rb;C<=Rc PC2 <= PC1; IR2<=IR1 ID/EX <= Instruction decode; [X]<= num[Ra]
EX	MAR <= B op $(IR2_{15})^{16}$ ## $IR2_{15..0}$ SMDR <= A [IR3 <= IR2 [PC3 <= PC2]
MEM	LMDR <= M[MAR] *(if LOAD)* or [PC4 <= PC3] [IR4 <= IR3] M[MAR] <= SMDR *(if STORE)*
WB	Ra <= MDR *(if LOAD)* [Sign ext.]

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

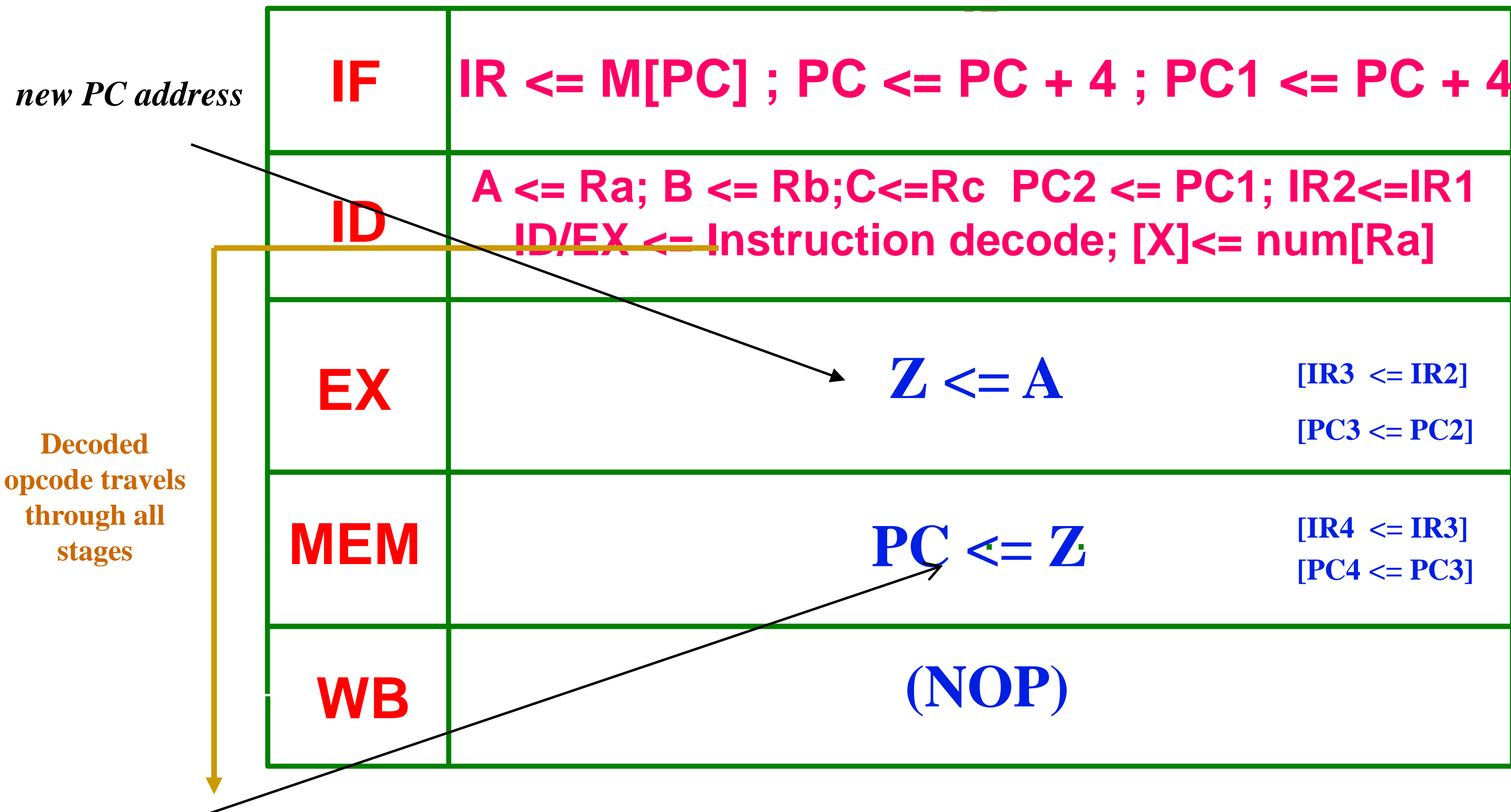


X : "BTA (BRANCH TARGET ADDRESS)"

New value in PC at the end of this cycle.
When Branch is taken 3 new unwanted instructions have already started

Branch on Reg A value (0/1)

Pipelined execution of a "JR" instruction



Which would be the stage sequence for a J instruction?

Pipelined execution of a "JL or JLR" instruction

IF	IR <= M[PC] ; PC <= PC + 4 ; PC1 <= PC + 4
ID	A <= Ra; B <= Rb;C<=Rc PC2 <= PC1; IR2<=IR1 ID/EX <= Instruction decode; [X]<= num[Ra]
EX	PC3 <= PC2 [IR3 <= IR2] Z <= A (If JLR) $Z \leftarrow PC2 + (IR_{25})^6 \ \#\# \ IR_{25..0}$ (If JL)
MEM	PC <= Z ; PC4<= PC3 [IR4 <= IR3]
WB	R31 <= PC4

Decoded opcode through all stages

In this case PCi values are used

NOTE: Write on R31 CANNOT be performed on-the fly since it could overlap with another register write

New value in PC in this interval .
When Jump executed 3 new unwanted instructions are already started

Which 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 <= Ra; B <= Rb;C<=Rc PC2 <= PC1; IR2<=IR1 ID/EX <= Instruction decode; [X]<= num[Ra]
EX	?
MEM	?
WB	?

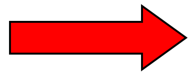
Pipeline Hazards

A *"Hazard"* occurs when during a clock cycle an instruction currently in a pipeline stage *can't 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't be executed simultaneously.**

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

- ***Control Hazards*** **– Instructions following a branch *depend* from the branch result (*taken/not taken*).**



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

Hazards and stalls

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

	Clk 1	Clk 2	Clk 3	Clk 4	Clk 5	Clk 6	Clk 7	Clk 8	Clk 9	Clk 10	Clk 11	Clk 12
⋮												
I_{i-3}	IF	ID	EX	MEM	WB							
I_{i-2}		IF	ID	EX	MEM	WB						
I_{i-1}			IF	ID	EX	MEM	WB					
I_i				IF	S	S	S	ID	....	....	WB	
I_{i+1}					S	S	S	IF	....	....	....	WB
⋮												

Stall: the clock signal for I_i, I_{i+1} …etc. is blocked for three periods

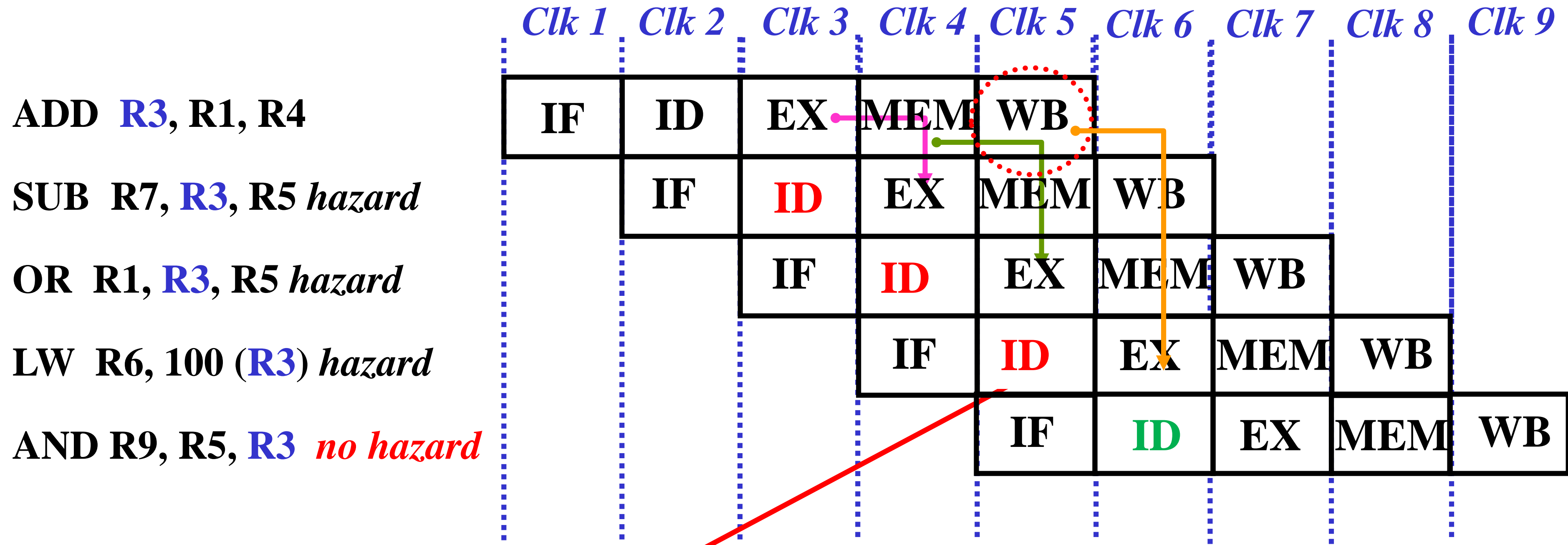
$$T_i = 8 * CLK = (5 + 3) * CLK$$

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

Normally the three stalled instructions are transformed in NOPs to avoid clock blocking

Instruction stalls

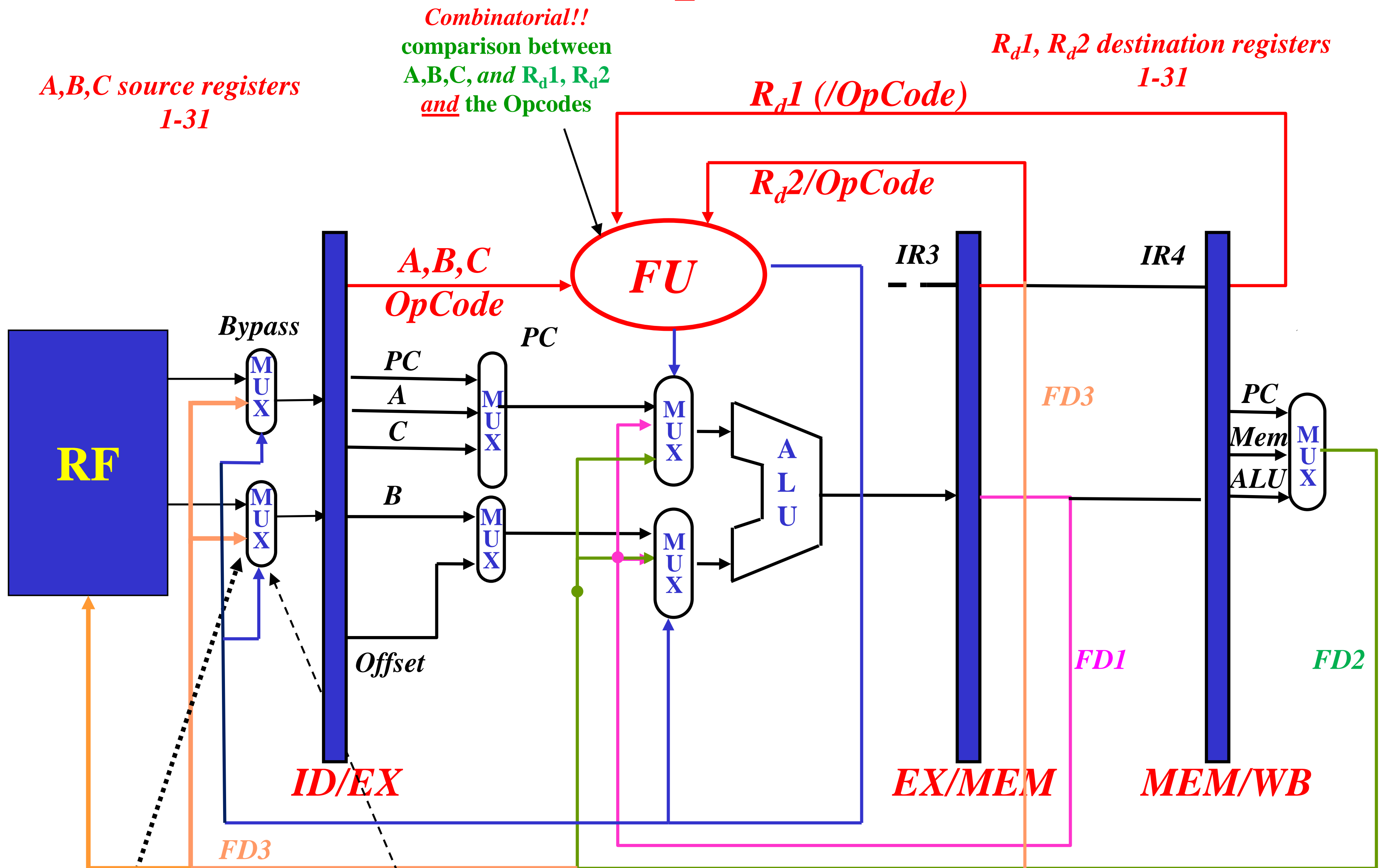
Forwarding



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

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

Forward implementation

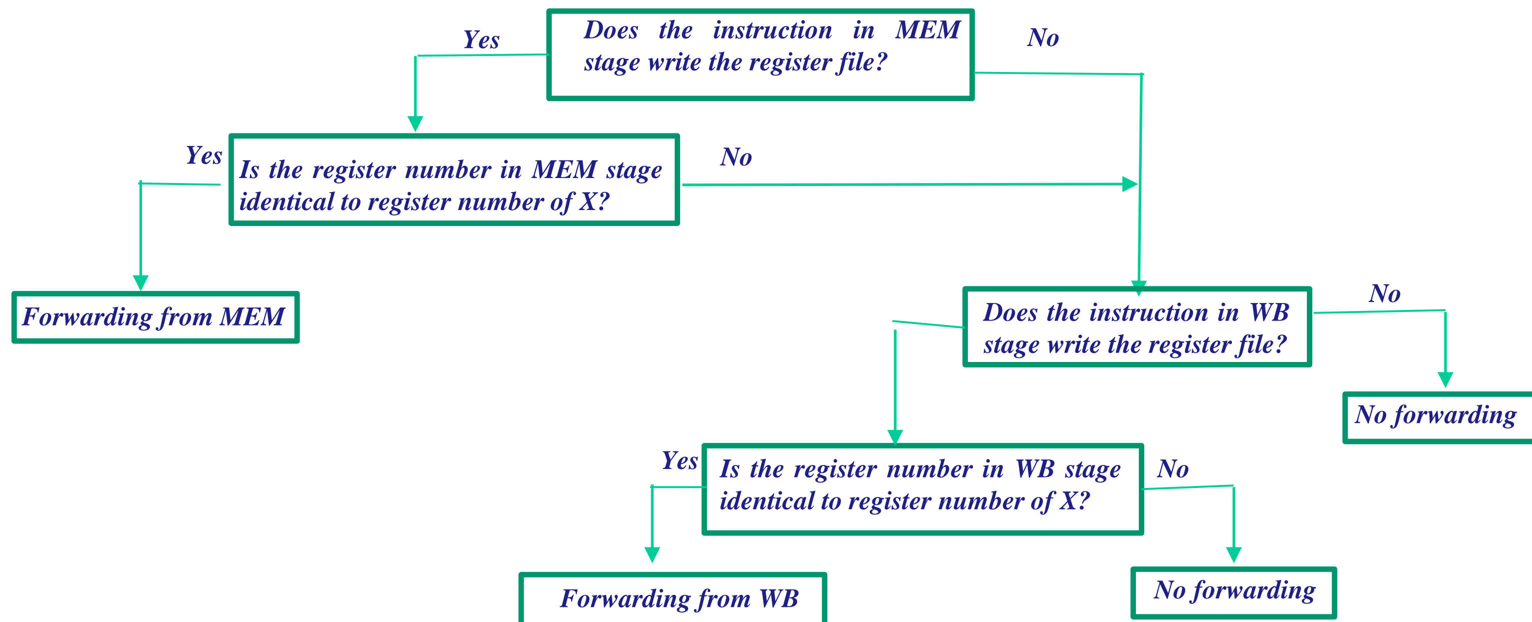


Often performed inside the RF

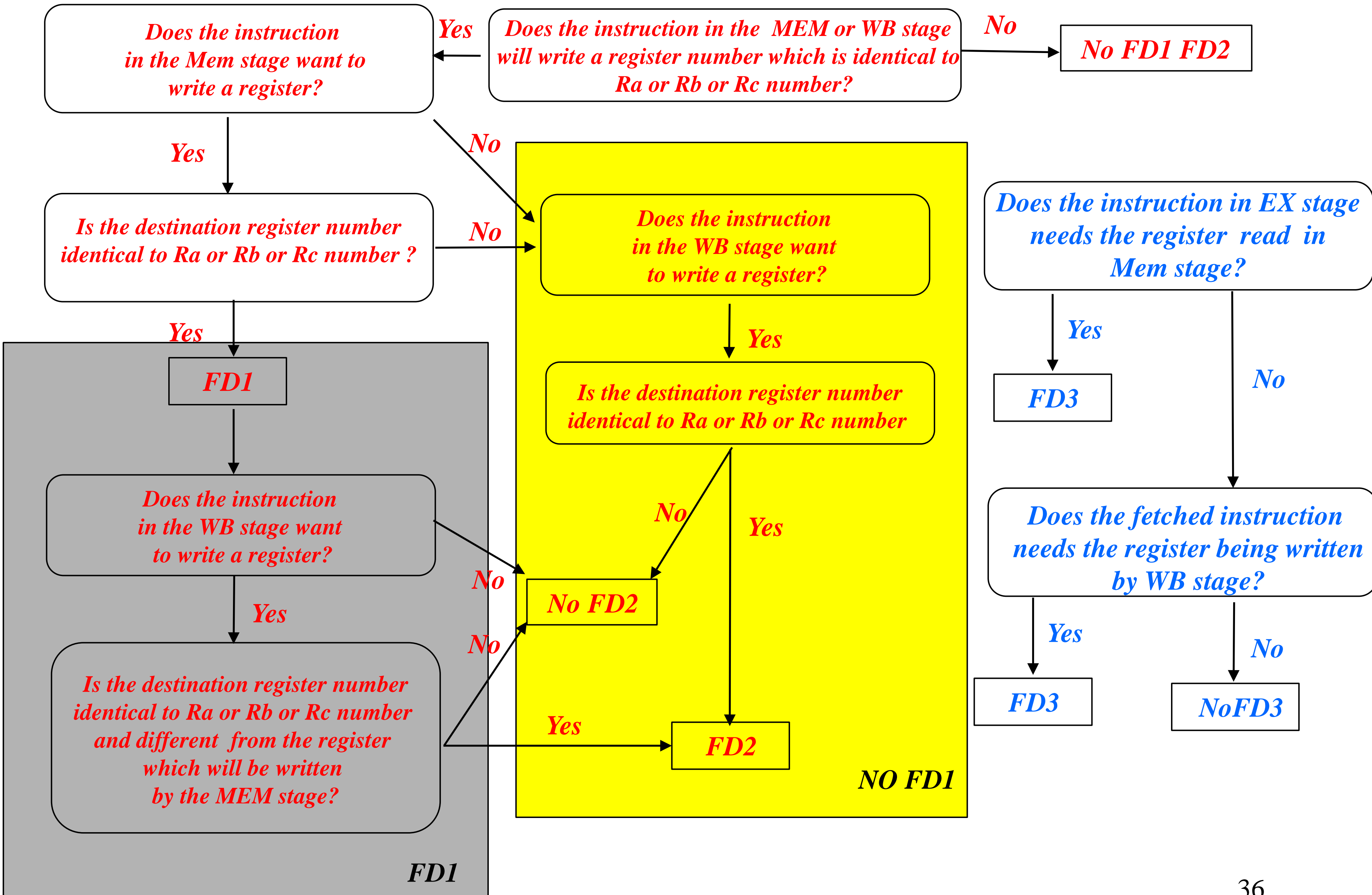
It allows "*the anticipation*" of the register on ID/EX
MUX control: IF opcode and comparison of RD with Ra, Rb and Rc numbers

Forward (simplified)

Let's call X and Y the values of registers Ra, Rb, Rc extracted from the register file. The following combinatorial procedure applies separately for each of these registers.

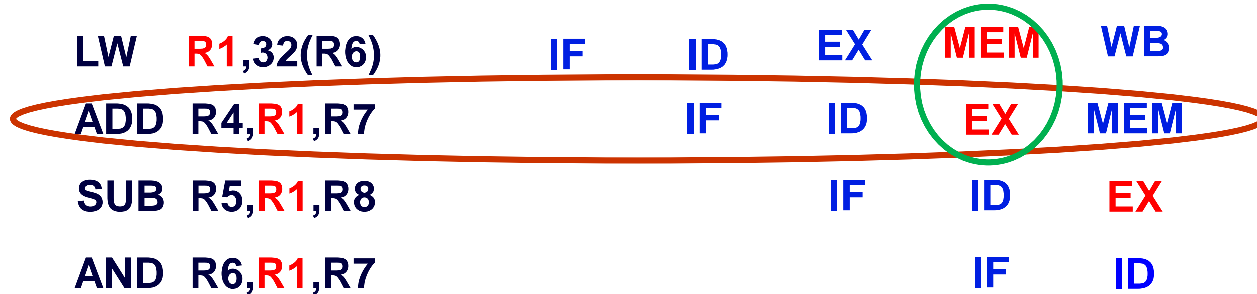


Forward Unit implementation

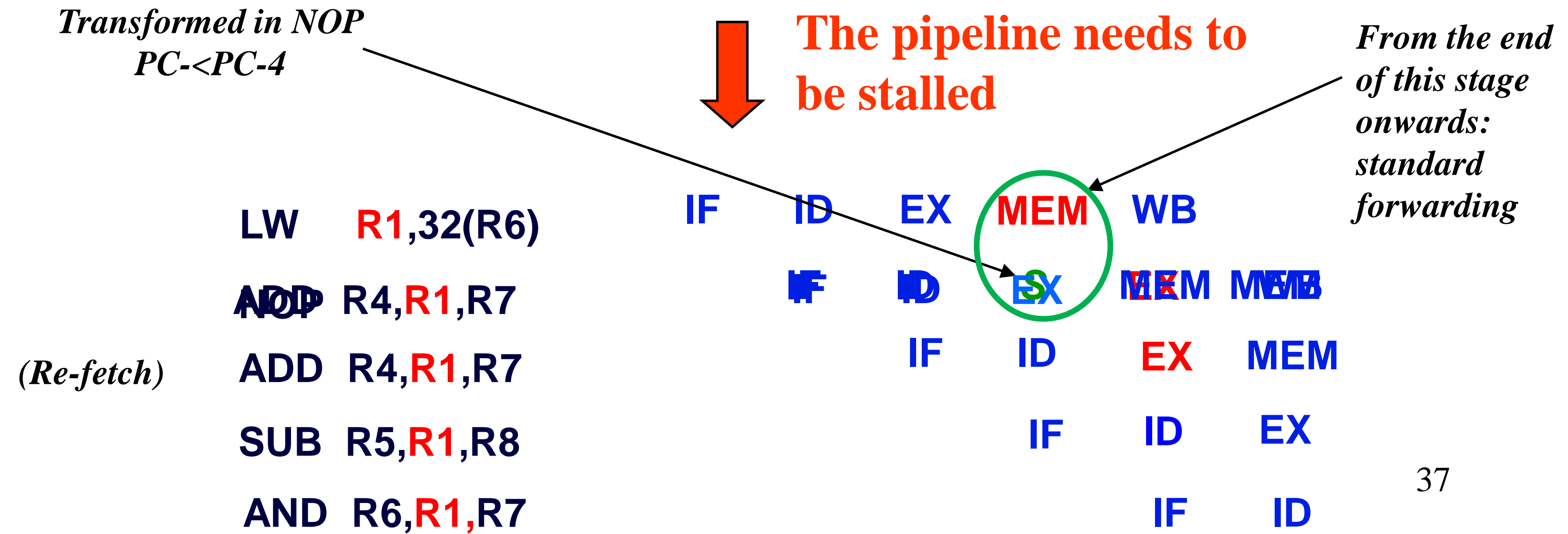


This slide must be viewed using its .PSM version

Data hazard due to LOAD instructions

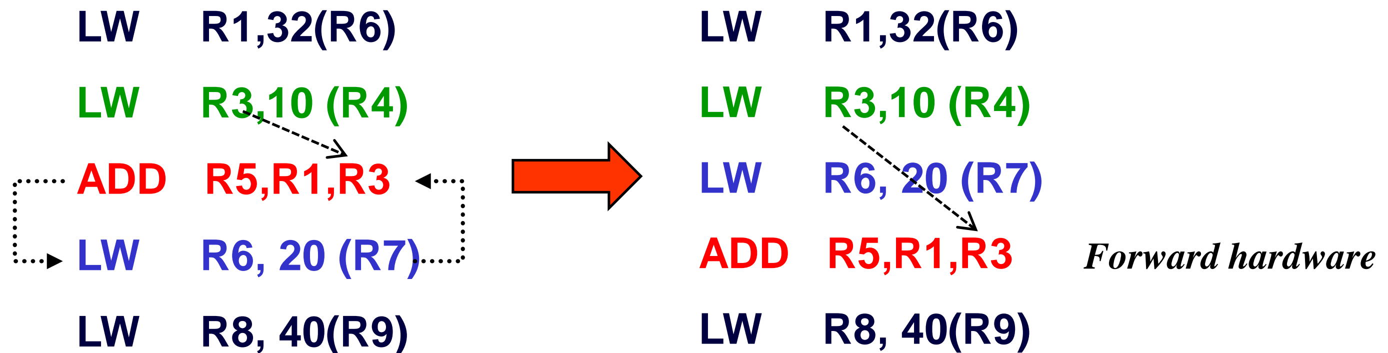


NOTE: the data required by the ADD is available only *at the end* of MEM stage. This hazard cannot be eliminated by forwarding (unless there is an additional input in the MUXs between memory and ALU – delays!)



Delayed load

In many RISC CPUs, the special hazard associated with the LOAD instruction (which would in any case lead to a stall) is not handled by stalling the pipeline but by software through the compiler (*delayed load*). In this example R3 is needed by the ADD instruction *while* it is read from the memory [instruction LW R3, 10(R4)]. Please notice that in any case a hardware forward netwotk is required



LOAD Instruction

delay slot

Next instruction

The compiler tries to fill the delay-slot with a "useful" instruction (worst case: NOP).

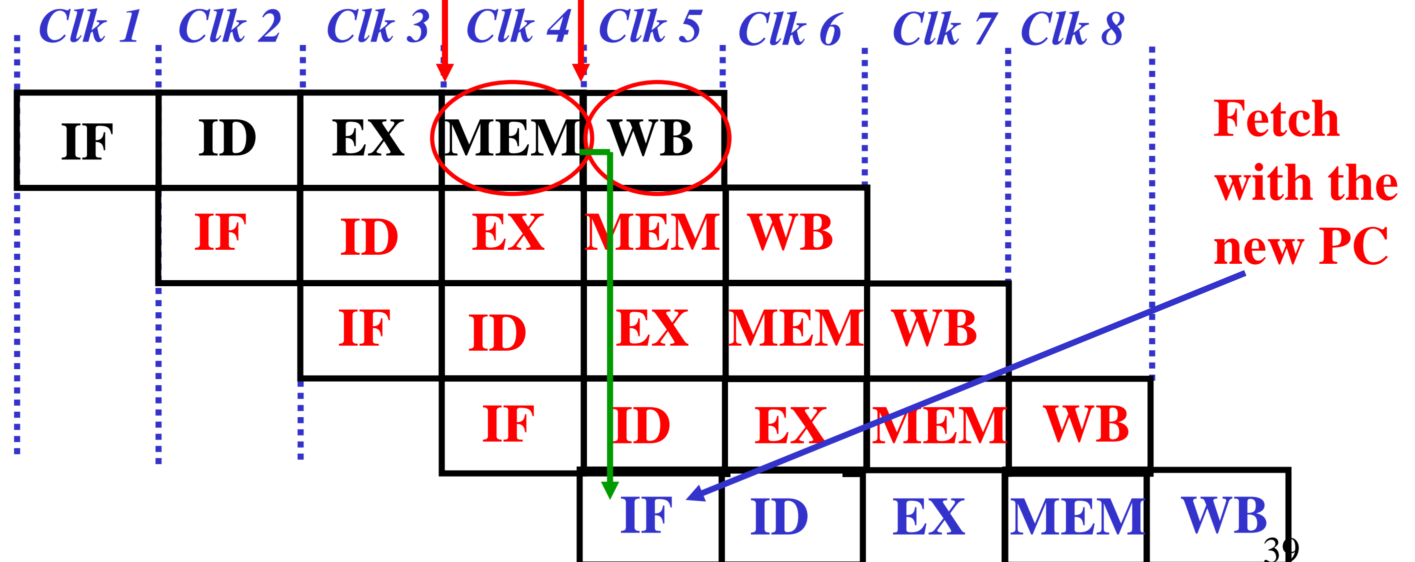
Control Hazards

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 InstructionAddress

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

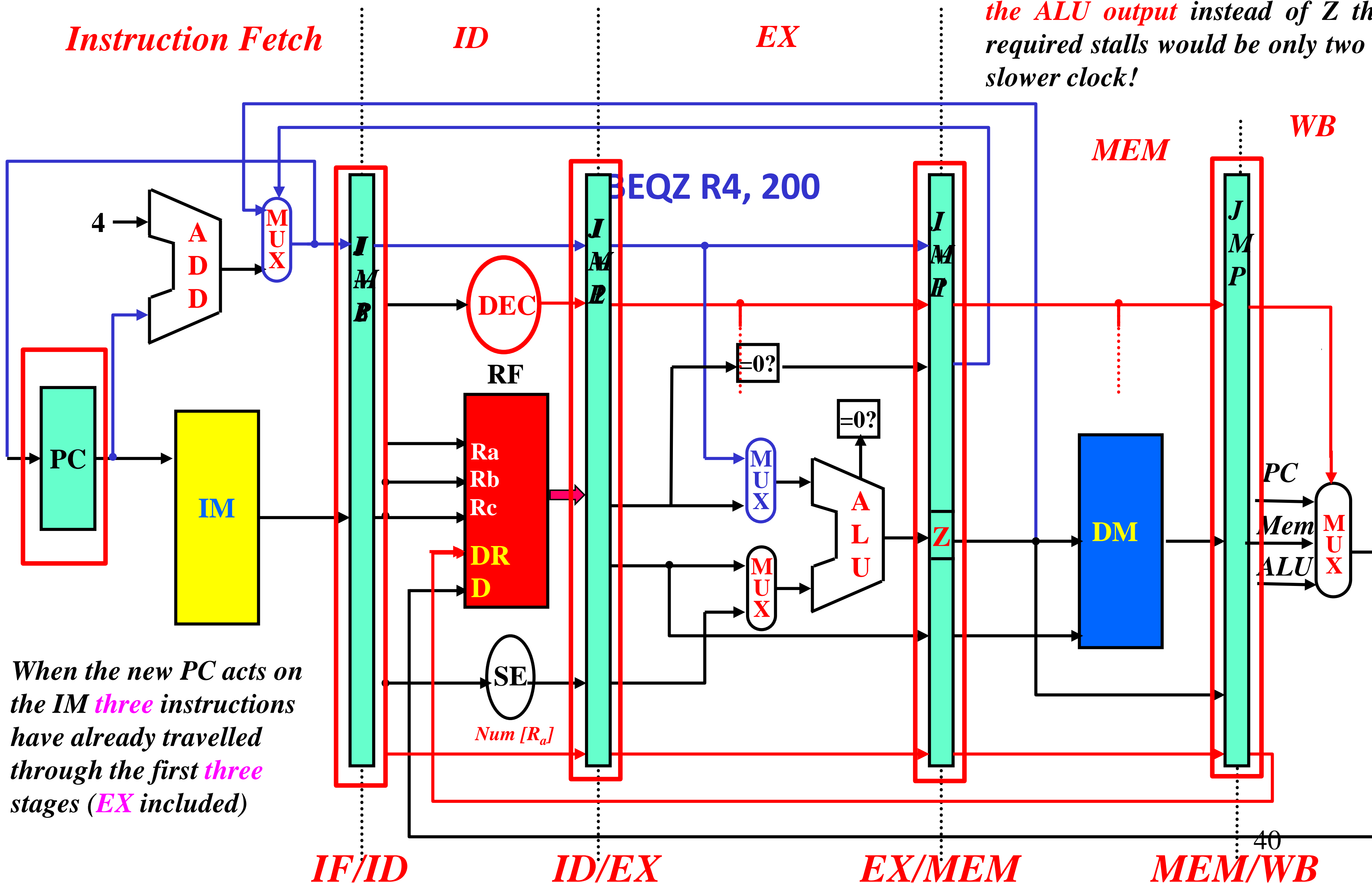
R4 ≠ 0 : *PC+4* (not taken)



DLX Branch or JMP

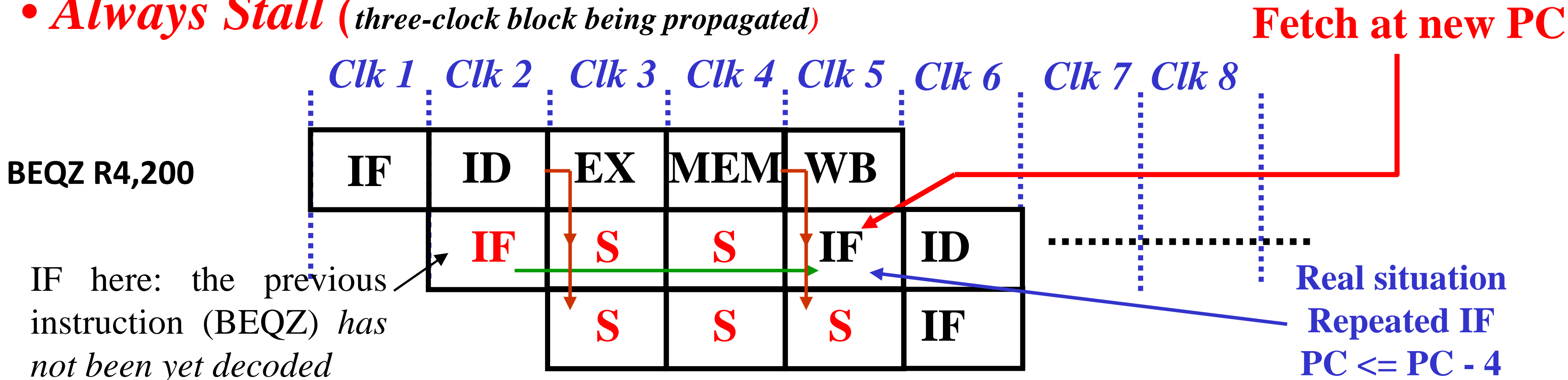
Detailed dapath slide:
See DLX Pipelined Datapath
Here we assume that the JMP instruction is the Ith instruction

NOTE if the feedback signal of the new PC were output directly from the ALU output instead of Z the required stalls would be only two – slower clock!



Handling the Control Hazards

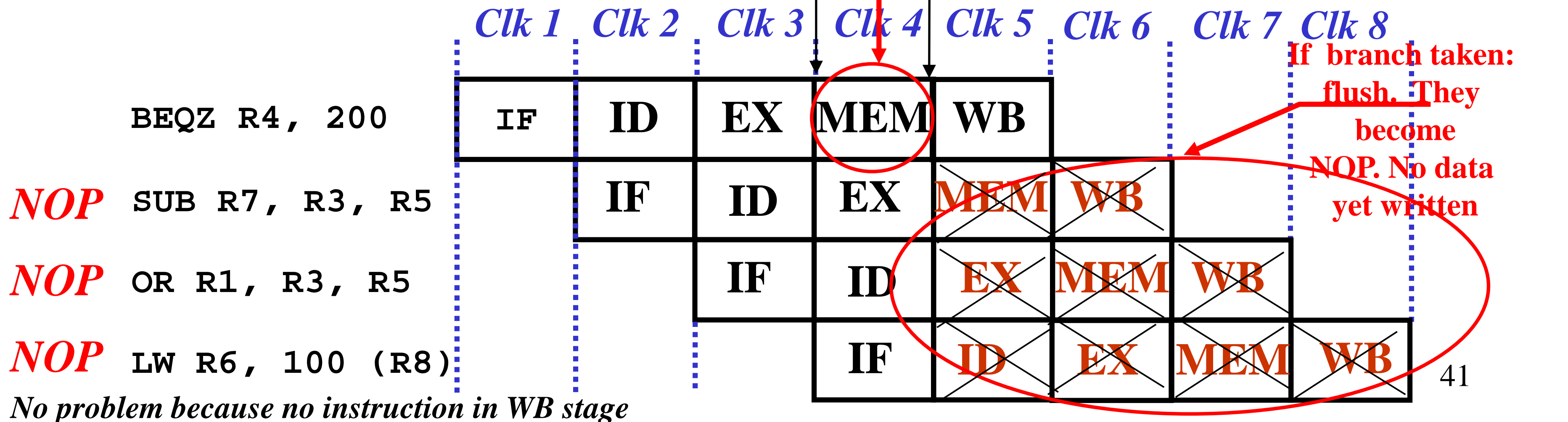
- *Always Stall* (*three-clock block being propagated*)



Here the new value of PC has been computed

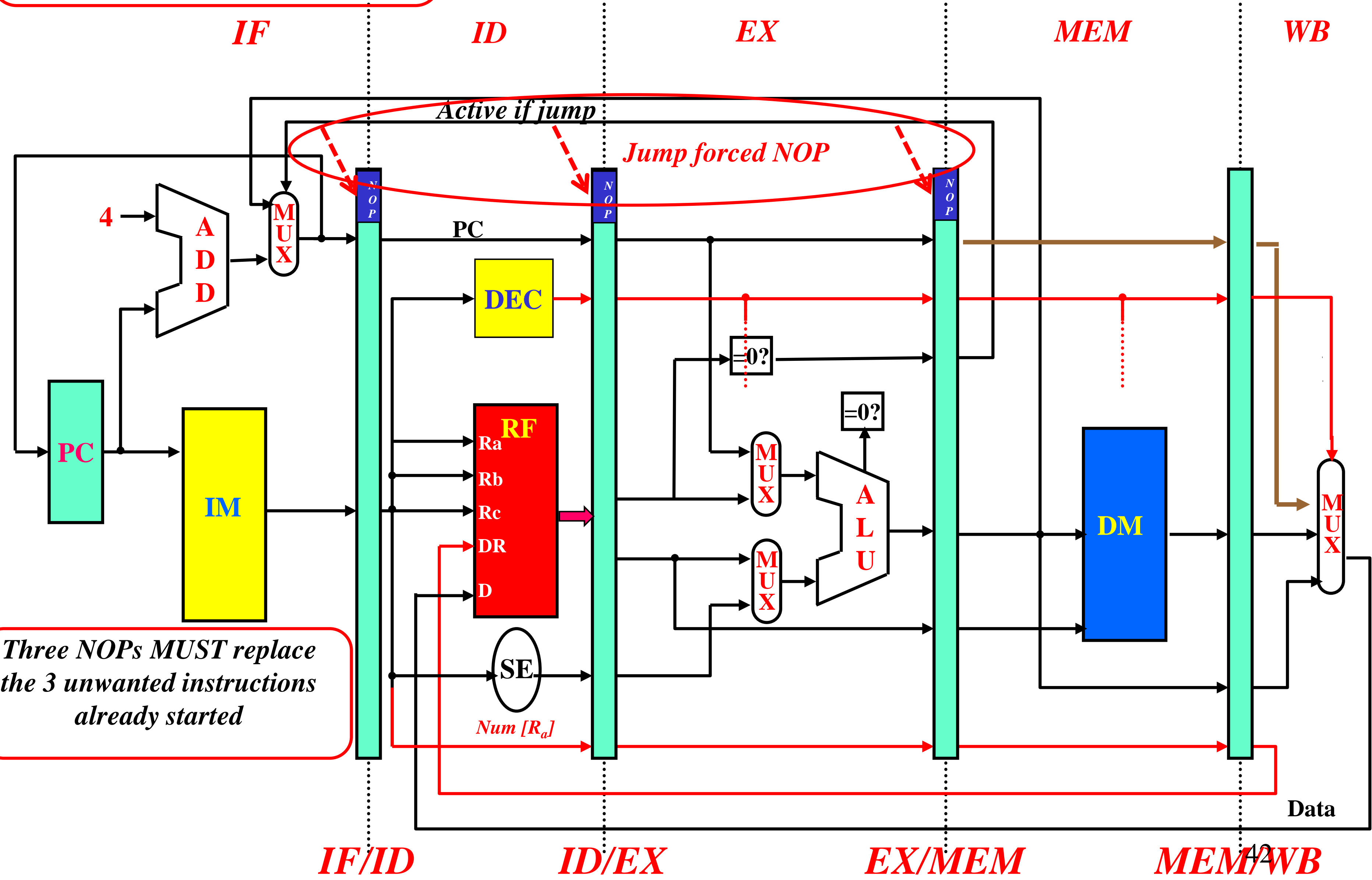
Here the new value is sampled by the PC

- *Predict Not Taken*



When the Branch Target Address is clocked into the PC three unwanted instructions are already in IF/ID, ID/EX and EX/MEM

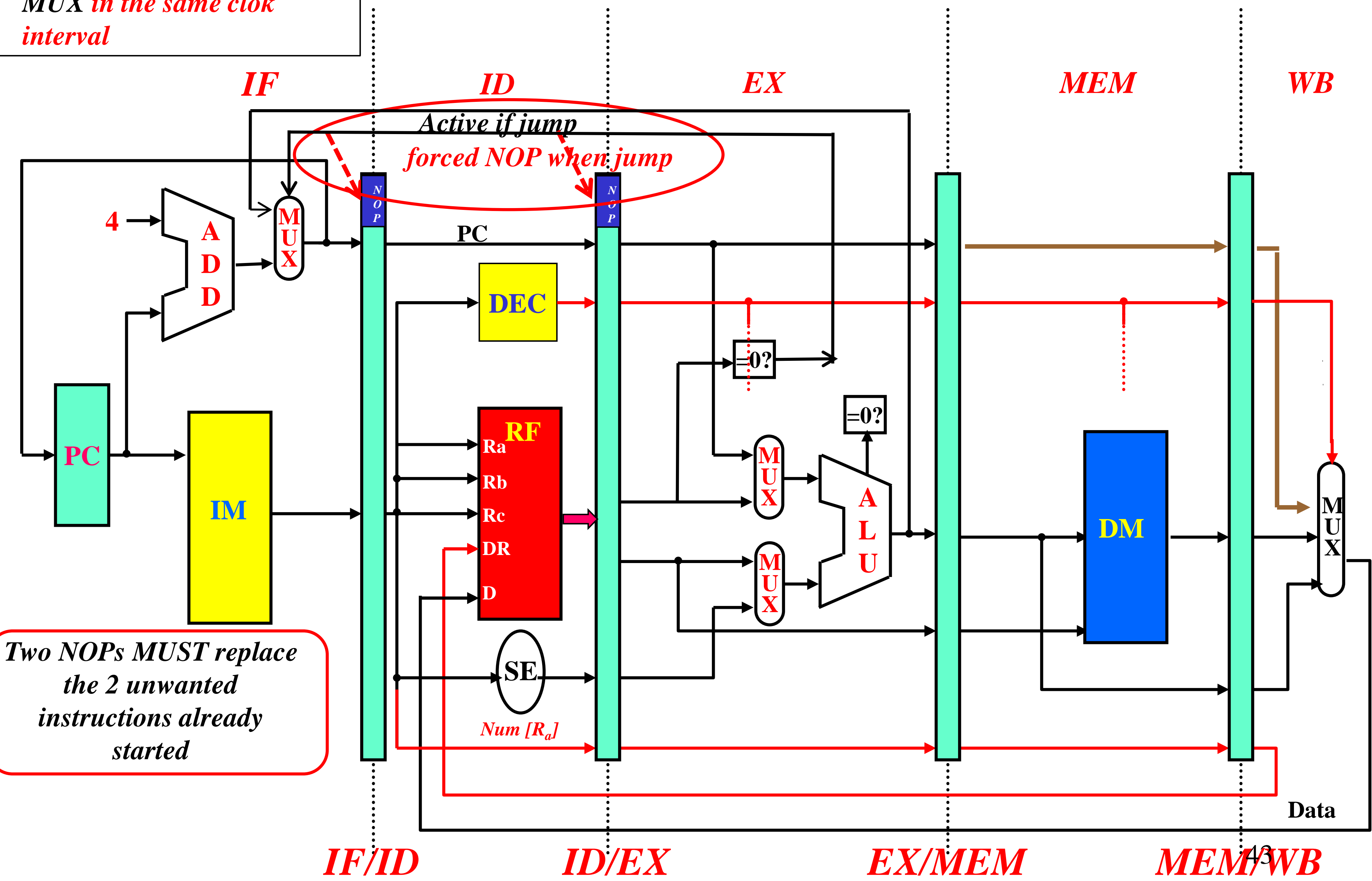
Stalls with jumps (1/3)



Three NOPs MUST replace the 3 unwanted instructions already started

Stalls with jump (2/3)

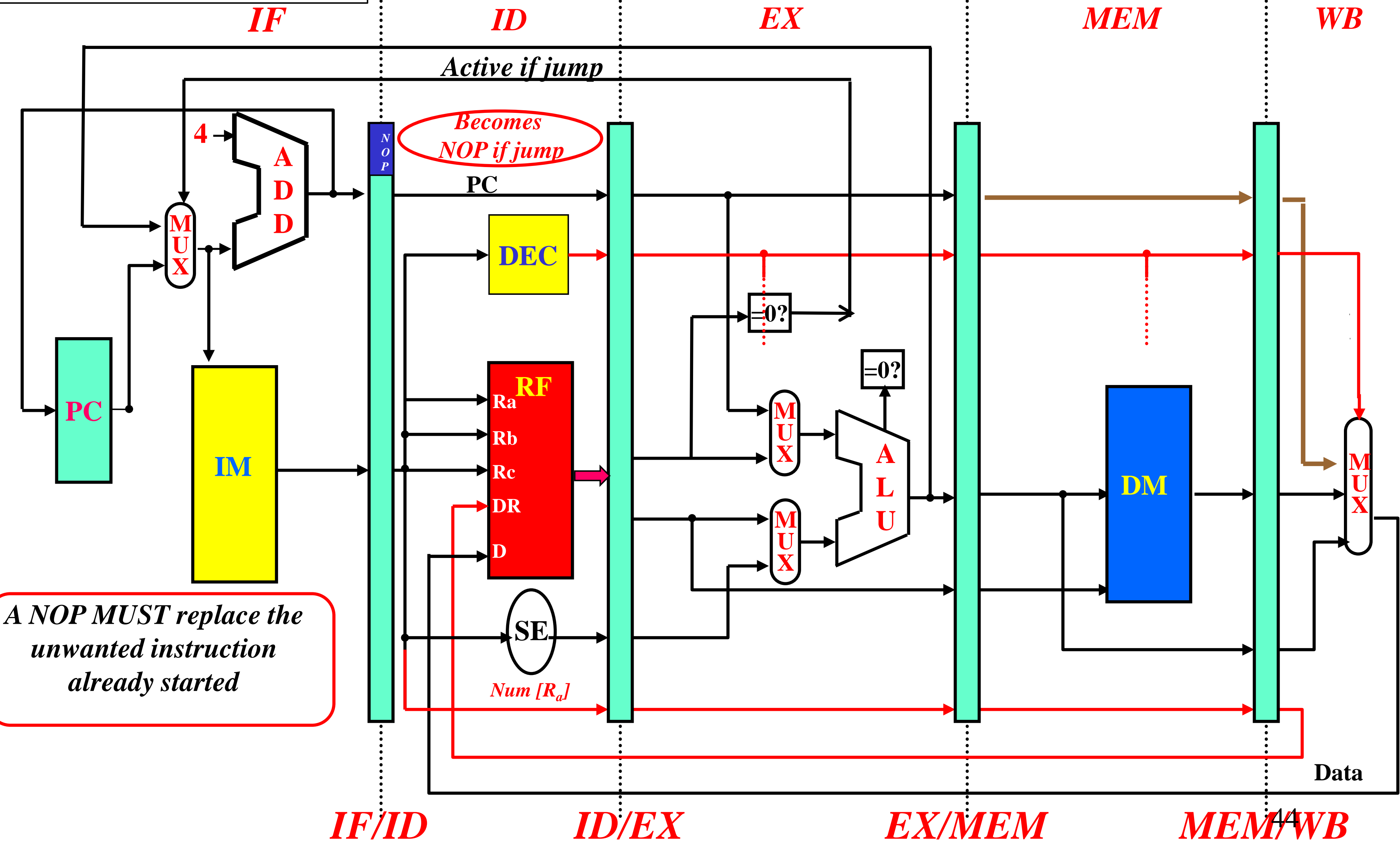
NOTE in this case the jump condition detection and the new PC value are input to the MUX in the same clok interval



NOTE In this case the jump condition and the new PC act on the MUX in the same period when the condition is detected

Stalls with jump (3/3)

Very slow clock solution !



A NOP MUST replace the unwanted instruction already started

Delayed branch

Similarly to the LOAD case. In several RISC CPUs the BRANCH instructions hazard is handled by SW through the compiler (*delayed branch*):

BRANCH instruction

delay slot
delay slot
delay slot

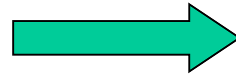
Next instruction

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

Delayed branch/jump

Original

Add R5, R4, R3
Sub R6, R5, R2
Or R14, R6, R21
Sne R1, R8, R9 ;
Br R1, +100



branch condition

Compiled

Sne R1, R8, R9 ; branch condition
Br R1, +100
Add R5, R4, R3
Sub R6, R5, R2
Or R14, R6, R21

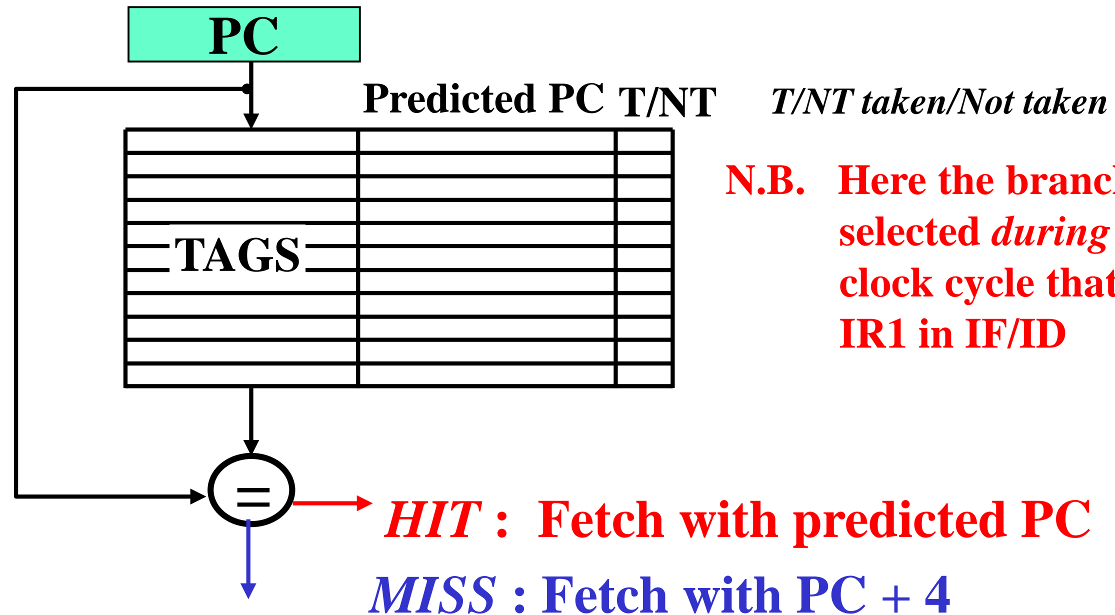
Obviously in this instructions group there must be no jumps!!!

Executed in both cases

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

Handling the Control Hazards

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



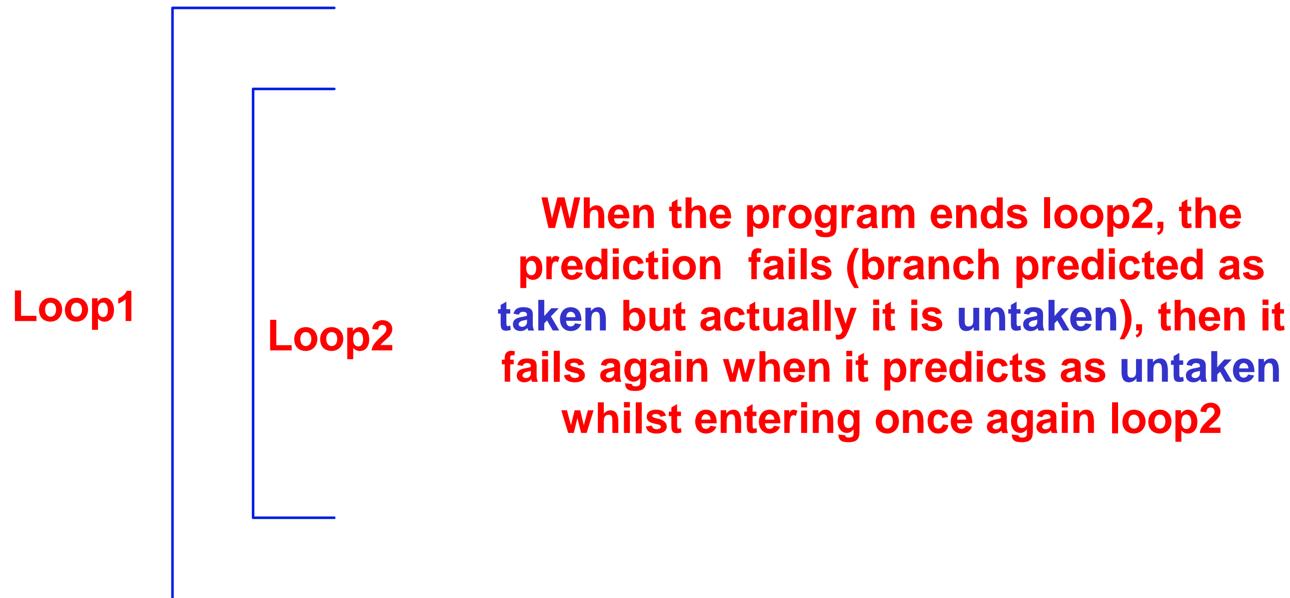
T/NT taken/Not taken

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

Correct prediction : no stalls

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 last branch occurred.



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

Usually *two* bits.