Calcolatori Elettronici L-A

ISA DLX: Implementazione Tramite Struttura "Pipelined"

Principio del Pipelining (1)

Il pipelining è oggi la principale tecnica di base impiegata per rendere "veloce" una CPU

L'idea alla base del pipeling è generale, e trova applicazione in molteplici settori dell'industria (linee di produzione, oleodotti …)

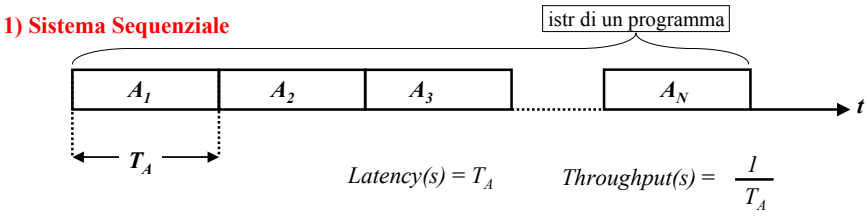
Un sistema, *S*, deve eseguire *N* volte un'attività *A*:

Def.: *Latency*: (o latenza [sec]) T_A tempo che intercorre fra l'inizio ed il completamento dell'attività *A*.

Def.: *Throughput* : (produttività [Hz]) frequenza con cui vengono completate le attività ($\frac{1}{T_A}$)

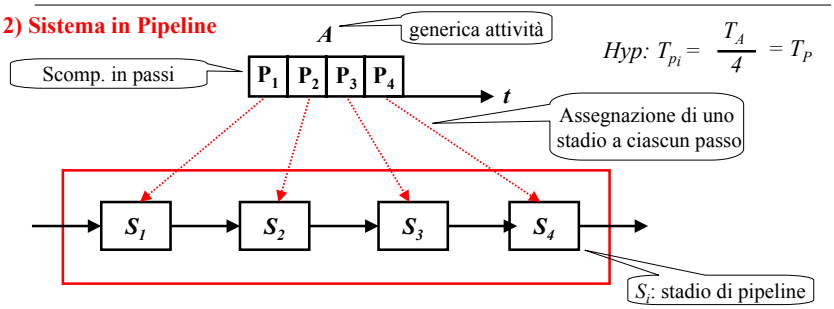
Principio del Pipelining (2)

1) Sistema Sequenziale



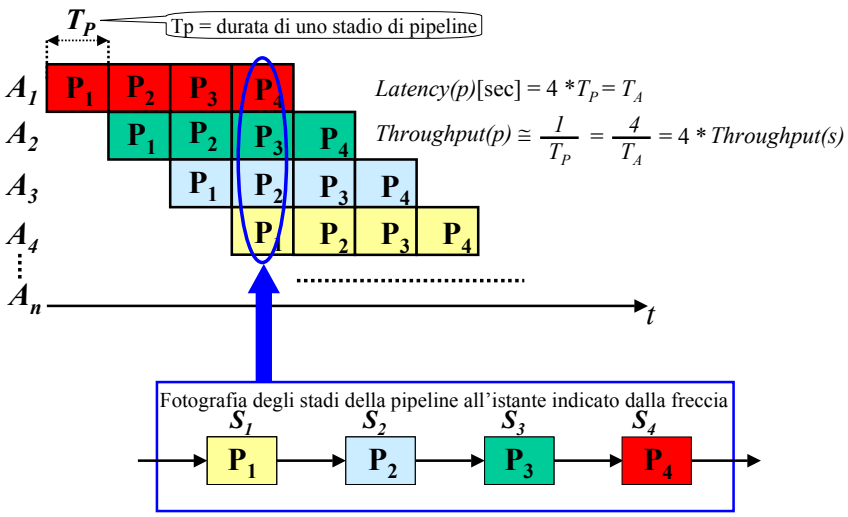
$Latency(s) = T_A$ $\quad Throughput(s) = \frac{1}{T_A}$

2) Sistema in Pipeline



$Hyp: T_{pi} = \frac{T_A}{4} = T_P$

Principio del Pipelining (3)



$Latency(p)[\text{sec}] = 4 * T_P = T_A$

$Throughput(p) \cong \frac{1}{T_P} = \frac{4}{T_A} = 4 * Throughput(s)$

Principio del Pipelining (4)

Il pipelining *non riduce il tempo necessario al completamento di una singola attività:*

$$Latency(p)\ [\text{sec}] = Latency(s)$$

Il pipelining *incrementa il Throughput*, moltiplicandolo per un fattore pari al numero degli stadi (K):

$$Throughput(p)\ [1/\text{sec}] = K * Throughput(s)$$

Ciò porta ad una riduzione dello stesso fattore *del tempo totale di esecuzione* di una sequenza di N attività $T_N = [\text{sec}]$:

$$T_N = \frac{N}{Throughput} \rightarrow T_N(s) = \frac{N}{Throughput(s)}\ ,\quad T_N(p) = \frac{N}{Throughput(p)}$$

$$\Rightarrow Speedup\ (p) = \frac{T_N(s)}{T_N(p)} = \frac{Throughput(p)}{Throughput(s)} = K$$

DLX pipelined 5

Bilanciamento di una Pipeline

- Caso Ideale

$$T_P = T_{Pi} = \frac{T_A}{K} \Rightarrow \text{Pipeline perfettamente bilanciata} \Rightarrow Speedup = K$$

- Caso Reale

$$T_P = max\ (T_{P1}, T_{P2} \ldots\ldots T_{PK}\ ) \Rightarrow \text{Bilanciatamento imperfetto} \Rightarrow Speedup < K$$

- Un Esempio

$T_A = 20\ t$ (t : unità di tempo, es.: cicli di clock)

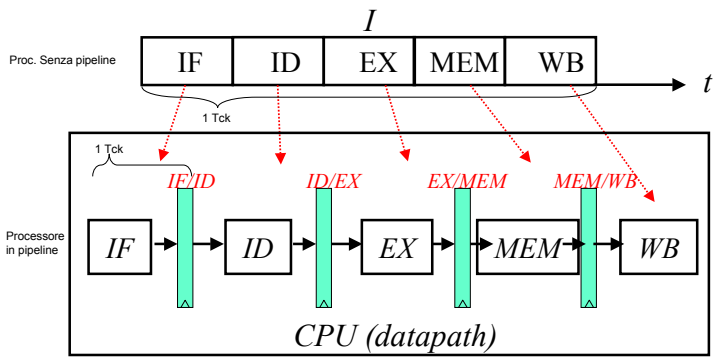
$T_{P1} = 5\ t\ ,\ T_{P2} = 5\ t\ ,\ T_{P3} = 6\ t\ ,\ T_{P4} = 4\ t \Rightarrow T_P = 6\ t$ (è il più lento)

$$\Rightarrow Speedup\ (p) = \frac{T_A}{T_P} = \frac{20\ t}{6\ t} \cong 3.33\ (< 4\ )$$

DLX pipelined 6

Pipelining in una CPU: DLX

Attività: $A_1, A_2, A_3 \ldots A_N$ → *Istruzioni:* $I_1, I_2, I_3 \ldots I_N$

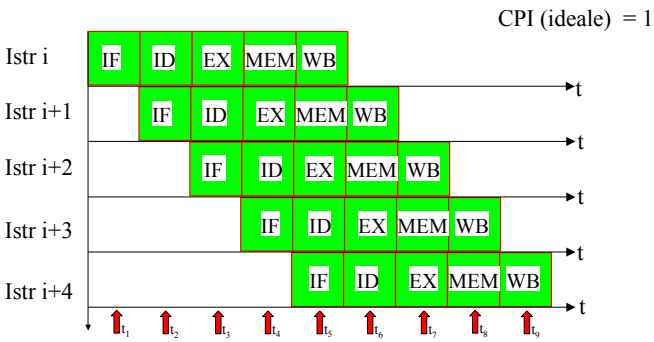


Pipeline Cycle → *Clock Cycle* → *Ritardo dello stadio più lento*

CPI=1 (idealmente !)

DLX pipelined 7

Evoluzione della pipeline del DLX



Carico aggiuntivo introdotto dai Pipeline Registers:

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

- T_{clk} — *Clock Cycle*
- T_d — *Ritardo registro a monte*
- T_P — *Ritardo stadio più lento*
- T_{su} — *Set-up registro a valle*

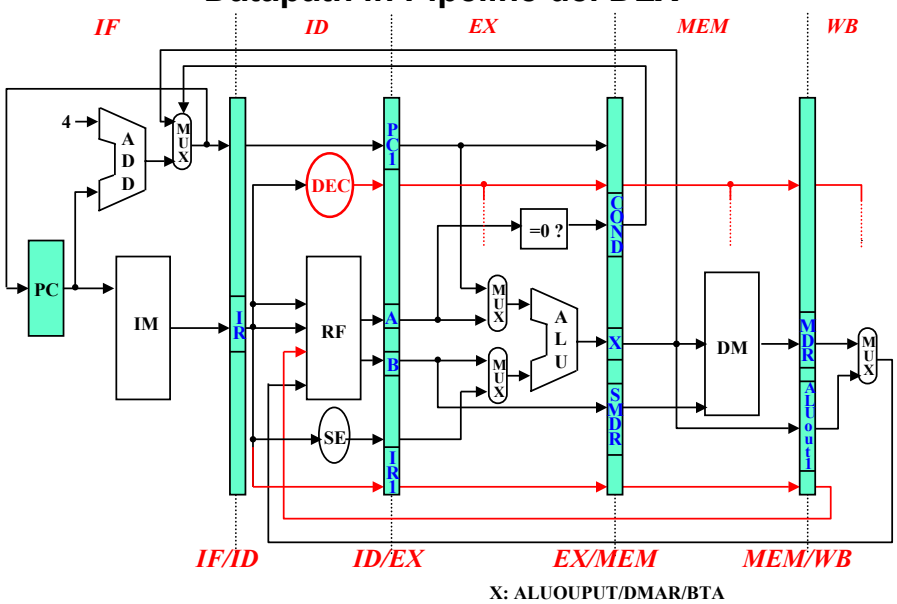
DLX pipelined 8

Requisiti per l'implementazione in pipeline

- 1) Ogni stadio deve essere attivo in ogni ciclo di clock.
- 2) E' necessario incrementare il PC in IF (invece che in ID).
- 3) E' necessario introdurre un ADDER (PC <-- PC+4) nello stadio IF.
- 4) Sono necessari due MDR (che chiameremo MDR e SMDR) per gestire il caso di una LOAD seguita immediatamente da una STORE (sovrapposizione di WB su registro e MEM su ram).
- 5) In ogni ciclo di clock devono poter essere eseguiti 2 accessi alla memoria (IF, MEM) riferiti a Instruction Memory (IM) e Data Memory (DM) (architettura *"Harvard")*.
- 6) Il clock della CPU è determinato dallo stadio più lento: IM, DM devono essere delle memorie cache (on-chip)!
- 7) I Pipeline Registers trasportano sia dati sia informazioni di controllo (il controller è "distribuito" fra gli stadi della pipeline).

DLX pipelined 9

Datapath in Pipeline del DLX



X: ALUOUPUT/DMAR/BTA

DLX pipelined 10

Esecuzione in pipeline di un'istruzione "ALU"

IF	IR←M[PC] ; PC←PC+4
ID	A←RS1; B←RS2; PC1←PC; IR1←IR (per gli stadi successivi)
EX	ALUOUTPUT←A *op* B oppure ALUOUTPUT←A *op* $IR_{0..15}$ ## $(IR_{15})^{16}$
MEM	ALUout1←ALUOUTPUT ("parcheggio" in attesa di WB)
WB	RD←ALUout1

DLX pipelined 11

Esecuzione in pipeline di un'istruzione "MEM" (Load o Store)

IF	IR ←M[PC] ; PC←PC+4
ID	A←RS1; B←RS2; PC1←PC;IR1←IR
EX	DMAR←A *op* $IR_{0..15}$ ## $(IR_{15})^{16}$ SMDR←B
MEM	MDR←M[DMAR] ***(LOAD)*** oppure M[DMAR]←SMDR ***(STORE)***
WB	RD←MDR ***(LOAD)***

DLX pipelined 12

Esecuzione in pipeline di un'istruzione "BRANCH"

IF	IR ← M[PC]; PC ← PC+4
ID	A ← RS1; B ← RS2; PC1 ← PC; IR1 ← IR
EX	BTA ← PC1 + $IR_{0..15}$ ## $(IR_{15})^{16}$ Cond ← A *op* 0
MEM	if (Cond) PC ← BTA
WB	- - - - - -

BTA = BRANCH TARGET ADDRESS
indirizzo del salto (calcolato in ID)

Alee nelle Pipeline

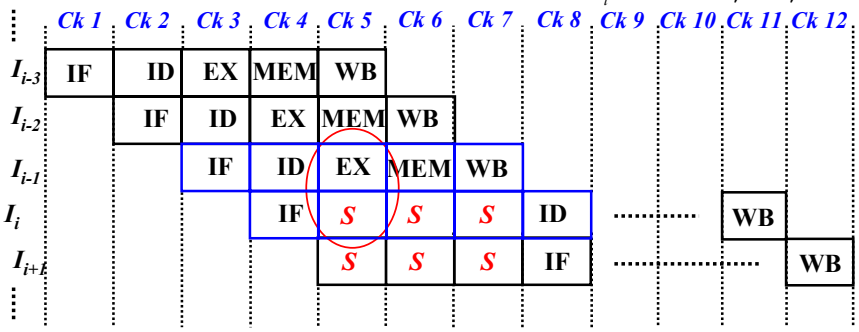
Si verifica una situazione di "*Alea*" (h*azard*) quando in un determinato ciclo di clock un'istruzione presente in uno stadio della pipeline non può essere eseguita in quel clock.

- *Alee Strutturali* - Una risorsa è condivisa fra due stadi della pipeline: le istruzioni correntemente in tali stadi non possono essere eseguite simultaneamente.
- *Alee di Dato* – Sono dovute a *dipendenze* fra le istruzioni. Ad esempio un istruzione tenta di leggere un registro prima che l'istruzione precedente l'abbia scritto (*RAW)*.
- *Alee di Controllo* – Le istruzioni che seguono un branch *dipendono* dal risultato del branch (*taken/not taken*).

L'istruzione che non può essere eseguita viene bloccata ("stallo della pipeline"), insieme a tutte quelle che la seguono, mentre le istruzioni che la precedono avanzano normalmente (così da rimuovere la causa dell'alea). Lintroduzione degli stalli è una possibile soluzione al problema delle alee

Alee e Stalli

Caso di un'alea di dato RAW nel DLX: I_{i-1} = `ADD R3, R1, R4`
I_i = `SUB R7, R3, R5`



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

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

CPI ideale — *Stalli per istruzione*

generalizz.

$T_N = N * 1 * CK$

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

CPI effettivo

Impatto degli stalli sullo Speedup

s: non-pipelined
p: pipelined

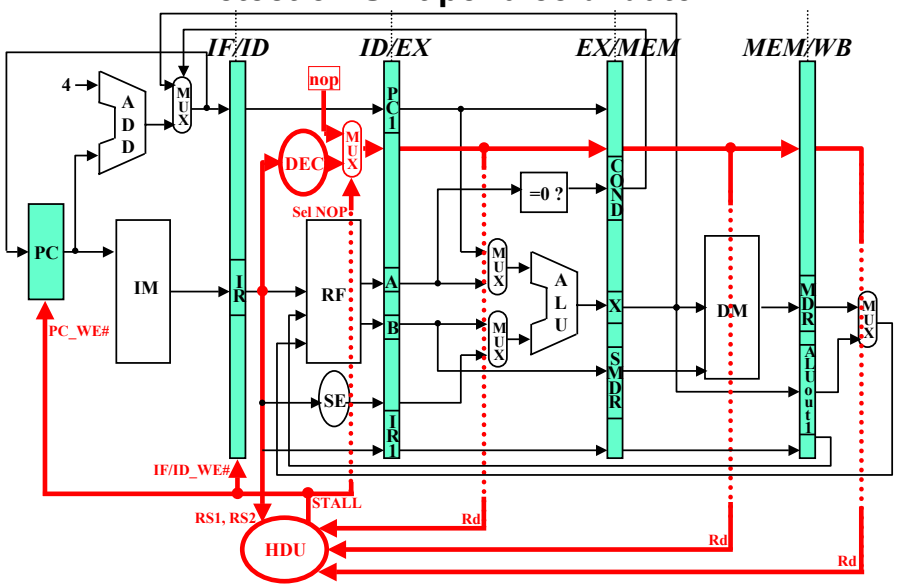
$$Speedup = \frac{T_N(s)}{T_N(p)} = \frac{N * CPI(s) * Ck(s)}{N * CPI(p) * Ck(p)} = \frac{CPI(s)}{1+S} * \frac{Ck(s)}{Ck(p)}$$

Ipotizzando che i clock siano identici: $Speedup = \frac{CPI(s)}{1+S}$

Ipotizzando che nell'implementazione sequenziale *tutte* le istruzioni impieghino un numero di clock pari al numero degli stadi della pipeline (indicato con K):

$$Speedup = \frac{K}{1+S}$$

Circuito per l'introduzione degli stalli: Hazard Detection Unit per alee di dato



Forwarding

```
ADD R3, R1, R4
SUB R7, R3, R5
OR  R1, R3, R5
LW  R6, 100 (R3)
AND R9, R5, R3
```

	Ck 1	Ck 2	Ck 3	Ck 4	Ck 5	Ck 6	Ck 7	Ck 8	Ck 9
ADD R3, R1, R4	IF	ID	EX	MEM	WB				
SUB R7, R3, R5		IF	ID	EX	MEM	WB			
OR R1, R3, R5			IF	ID	EX	MEM	WB		
LW R6, 100 (R3)				IF	ID	EX	MEM	WB	
AND R9, R5, R3					IF	ID	EX	MEM	WB

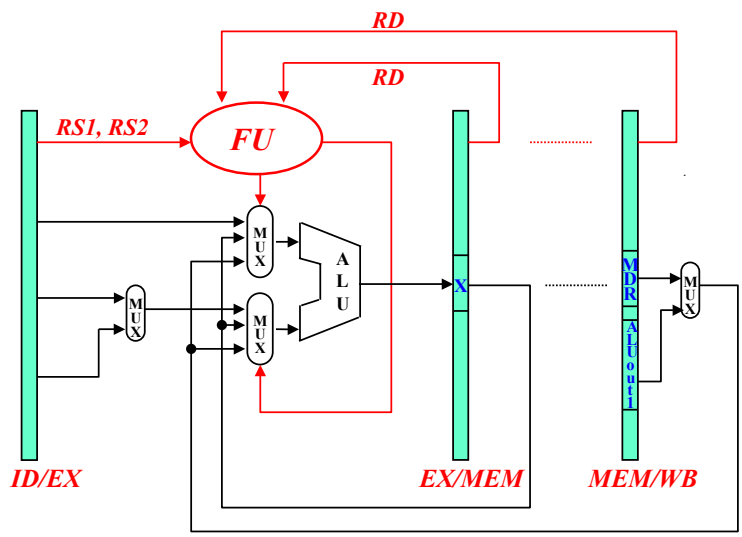
Se il RF ritorna il nuovo valore in caso di RD/WR sullo stesso registro non c'è alea fra la AND e la LW.

Il *forwarding* consente di eliminare quasi tutte le alee di tipo RAW della pipeline del DLX *senza stallare* la pipeline.

Esercizio

• Si disegni lo schema logico del Register File del DLX nell'ipotesi che esso debba restituire il nuovo valore in caso di lettura/scrittura simultanea sullo stesso registro.

Implementazione del Forwarding



Alea di dato dovuta alle istruzioni di LOAD

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	

N.B. il dato richiesto dalla ADD è presente solo alla fine di MEM. L'Alea non può essere eliminata con il Forwarding!

E' necessario stallare la pipeline

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

DLX pipelined 21

Delayed load

In diverse CPU RISC l'alea associata alla LOAD non è gestita in HW stallando la pipeline ma è gestita via SW dal compilatore (*delayed load*):

Istruzione LOAD

delay slot

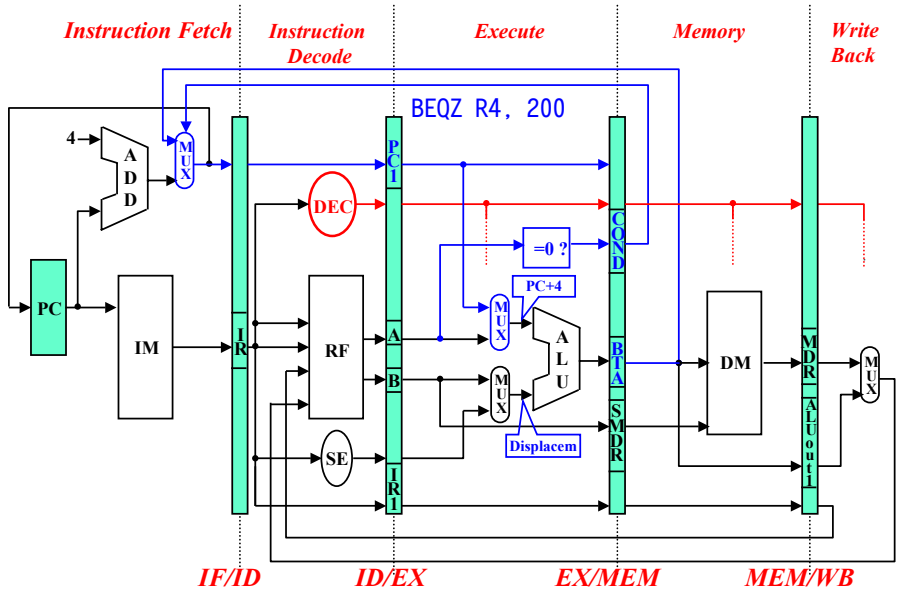
Istruzione Successiva

Il compilatore cerca di riempire il delay-slot con un'istruzione "utile" (caso peggiore: NOP).

```
LW   R1,32(R6)          LW   R1,32(R6)
LW   R3,10 (R4)         LW   R3,10 (R4)
ADD  R5,R1,R3           LW   R6, 20 (R7)
LW   R6, 20 (R7)        ADD  R5,R1,R3
LW   R8, 40(R9)         LW   R8, 40(R9)
```

DLX pipelined 22

Datapath in Pipeline del DLX (3)



Alee di Controllo

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 : *BTA* (taken)

R4 ≠ 0 : *PC+4* (not taken)

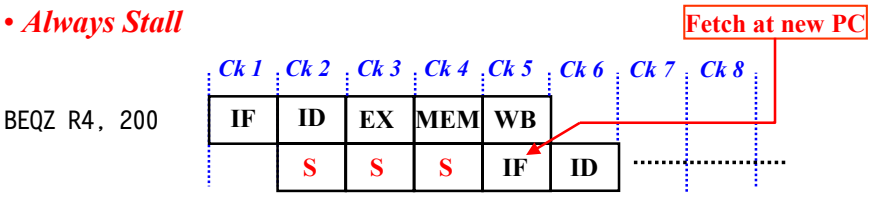
Branch Completion

Fetch at new PC

	Clk 1	Clk 2	Clk 3	Clk 4	Clk 5	Clk 6	Clk 7	Clk 8
BEQZ R4, 200	IF	ID	EX	MEM	WB			
SUB R7,R3,R5		IF	ID	EX	MEM	WB		
OR R1,R3 ,R5			IF	ID	EX	MEM	WB	
LW R6,100,(R8)				IF	ID	EX	MEM	WB

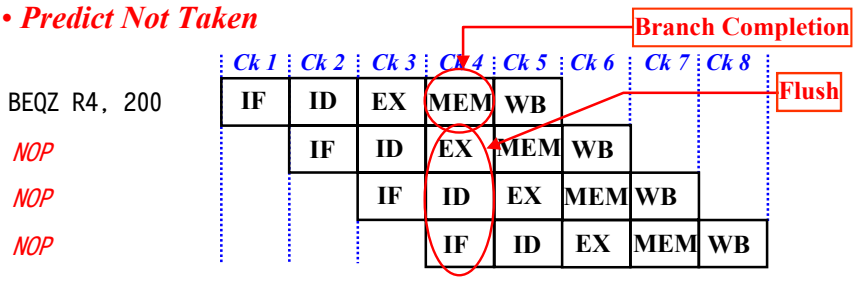
Gestione delle Alee di Controllo (1)

• *Always Stall*



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

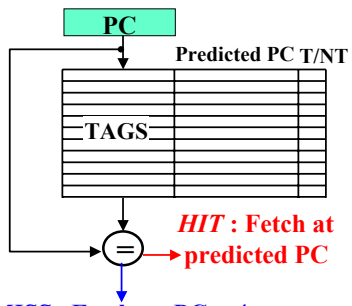
• *Predict Not Taken*



Gestione delle Alee di Controllo (2)

Hyp.: Branch = 25 %
Taken = 65 % → *CPI = (1 + S) = (1 + 3 * 0.65 * 0.25) = 1.48*

• ***Dynamic Prediction:*** **Branch Target Buffer**



HIT Predizione Corretta : 0 stalli
Predizione Errata : (3 +1) stalli
MISS Predict Not Taken
Taken: (3+1) stalli
Not Taken: 1 stallo

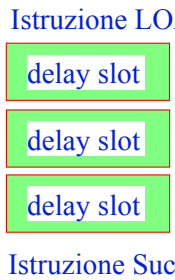
H: Hit Rate, *E*: % di Predizioni Errate
B: % di Branch
T/NT: % di Branch Taken/Not Taken

S = ((H * E * 4 + (1-H) * (T * 4 + NT * 1)) * B

S = ((0.9 * 0.1 * 4 + 0.1 * (0.65 * 4 + 0.35* 1)) * 0.25 = 0.16 CPI=(1 +S) = 1.16

Delayed branch

• **Similmente al caso della LOAD, in diverse CPU RISC l'alea associata alle istruzioni di BRANCH è gestita via SW dal compilatore (*delayed branch*):**



Il compilatore cerca
di riempire i delay-slot
con istruzioni "utili"
(caso peggiore: NOP).

• **Introducendo opportune modifiche al datapath è possibile anticipare l'esecuzione del branch fino a ridurre la penalità ad un solo clock (un solo delay slot, come nel caso della LOAD).**