La CPU pipeline

Alberto Borghese, Federico Pedersini
Dipartimento di Informatica
Università degli Studi di Milano

Principio intuitivo della pipe-line

- Anna, Bruno, Carla e Dario devono fare il bucato.

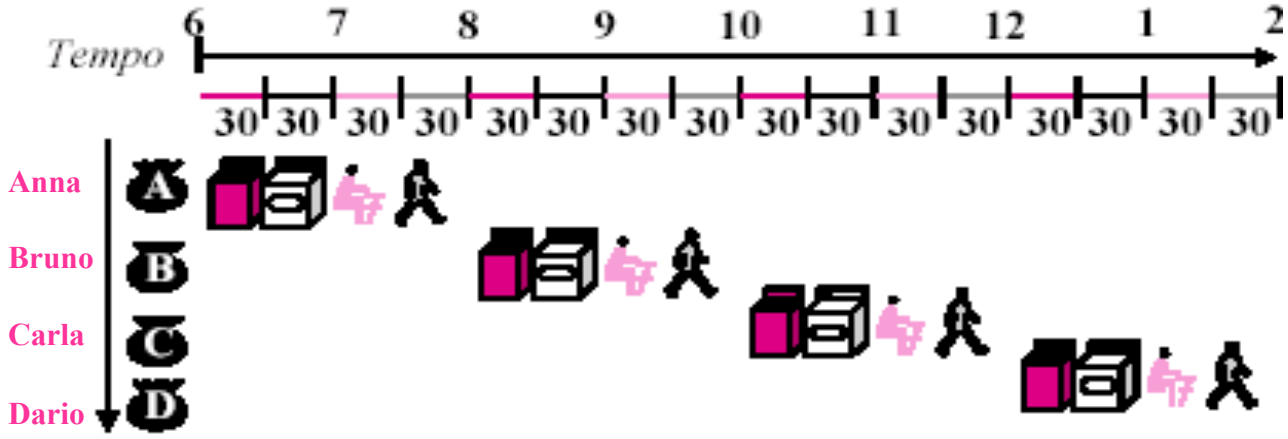
  Ciascuno deve:

  1. lavare,
  2. asciugare,
  3. stirare
  4. mettere via

  un carico di biancheria

- La lavatrice richiede: **30 minuti**
- L'asciugatrice: **30 minuti**
- Stirare richiede: **30 minuti**
- Piegare/mettere via: **30 minuti**

La lavanderia *sequenziale*



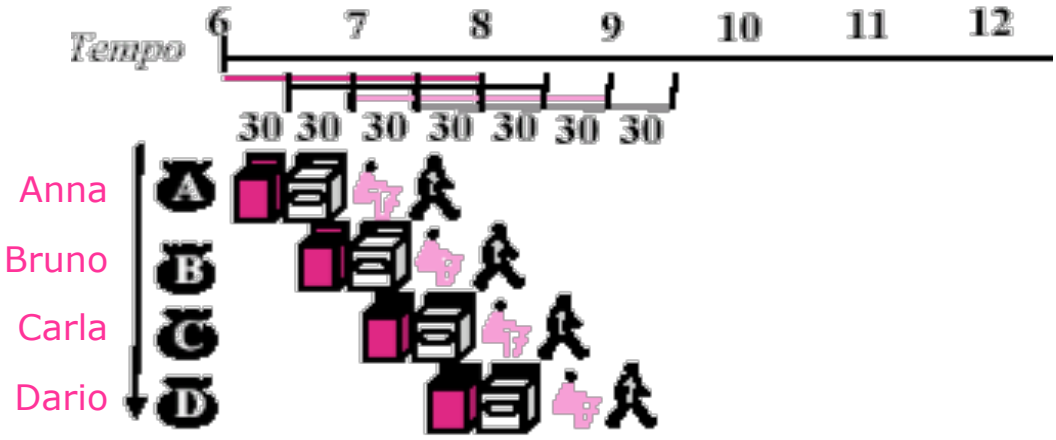
Approccio sequenziale:

- le singole operazioni vengono svolte una alla volta

  Tempo totale = TempoCicloCompleto * N.Utenti =
  = (TempoSingolaFase * N.Fasi) * N.Utenti

- Tempo totale richiesto: **0,5 ore × 4** fasi **× 4** utenti **= 8 ore**
- Ogni nuovo "lavoro" richiede **2 ore** in più

La lavanderia *pipe-line*



Approccio pipeline:

- le singole operazioni vengono svolte **in parallelo**
  - **T_totale = T_SingolaFase * (N_Fasi + N_Utenti – 1)**
- Tempo totale richiesto: **3,5 ore**
- Ogni nuovo "lavoro" richiede **0,5 ore** in più: (***0,5 ore = 2 ore / 4 stadi***)

Osservazioni sulla pipeline

Caratteristiche:

- Il tempo di ciascuna operazione elementare non viene ridotto.
- Gli stadi della pipeline lavorano in contemporanea perché utilizzano unità funzionali differenti.
- Le unità funzionali lavorano sequenzialmente (in passi successivi) su istruzioni successive

Il ***throughput aumenta di*** *un* ***fattore*** *pari al* ***numero di stadi***
(***throughput*** = quantità di lavoro / tempo)

Prestazioni:

- a regime (= quando tutti gli stadi di pipeline lavorano) il **throughput** aumenta di ***N* volte**, dove ***N*** è il numero di stadi

Vincoli:

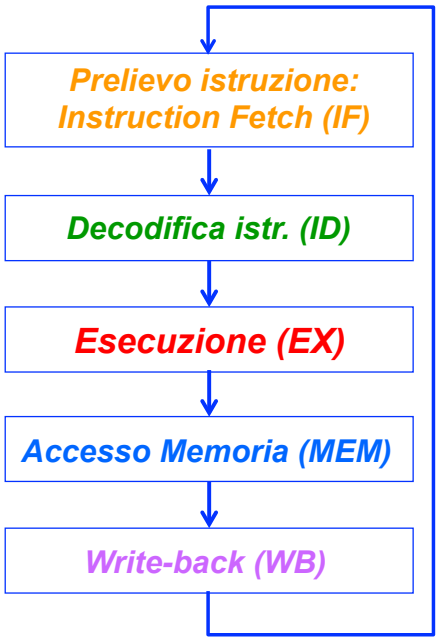
- Tutti gli stadi devono essere avere la **stessa durata**
- Tutti i "lavori" devono utilizzare **tutti gli stadi** in sequenza

Ciclo di esecuzione: istruzioni MIPS

MIPS è un'architettura pipeline

MIPS32: *pipeline a* ***5 stadi***

1. **IF**: Prelievo istruzione (Instruction Fetch)
2. **ID**: Decodifica istruzione (+ lettura RF)
3. **EX**: Esecuzione
4. **MEM**: Accesso a memoria (Read/Write)
5. **WB**: Scrittura del register file



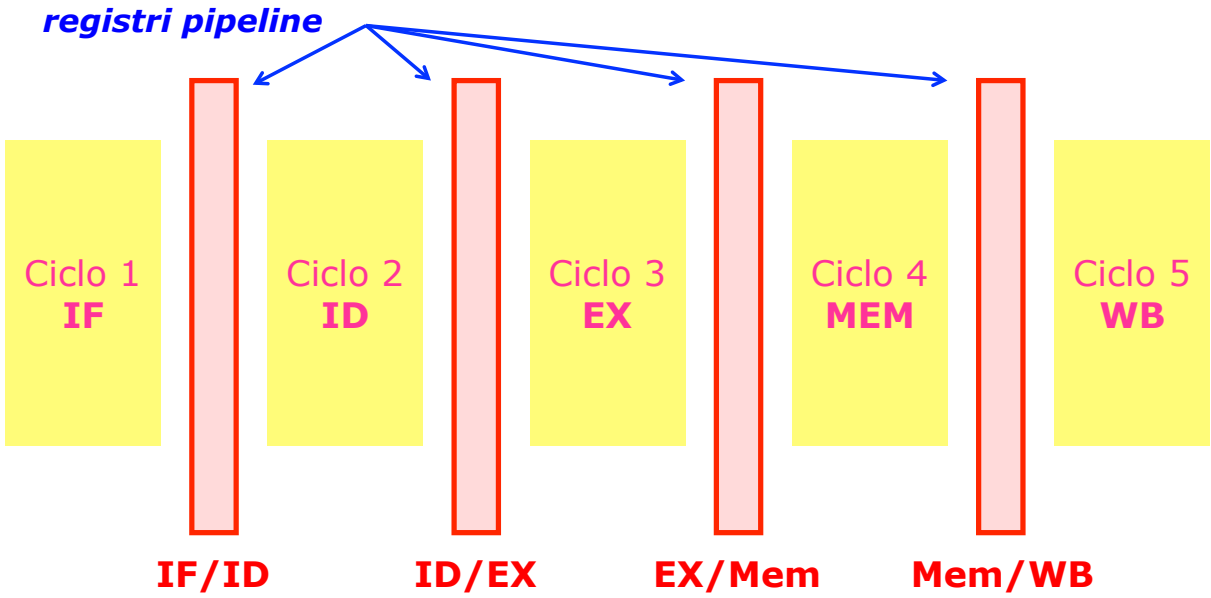
Tutte le istruzioni devono attraversare **tutti i 5 stadi, in ordine**

- Tutte le istruzioni durano 5 fasi (cicli clock)
- Nelle fasi "inutili", l'istruzione è inattiva

I 5 stadi della pipeline

Tra fasi contigue sono posti i **registri di pipeline**

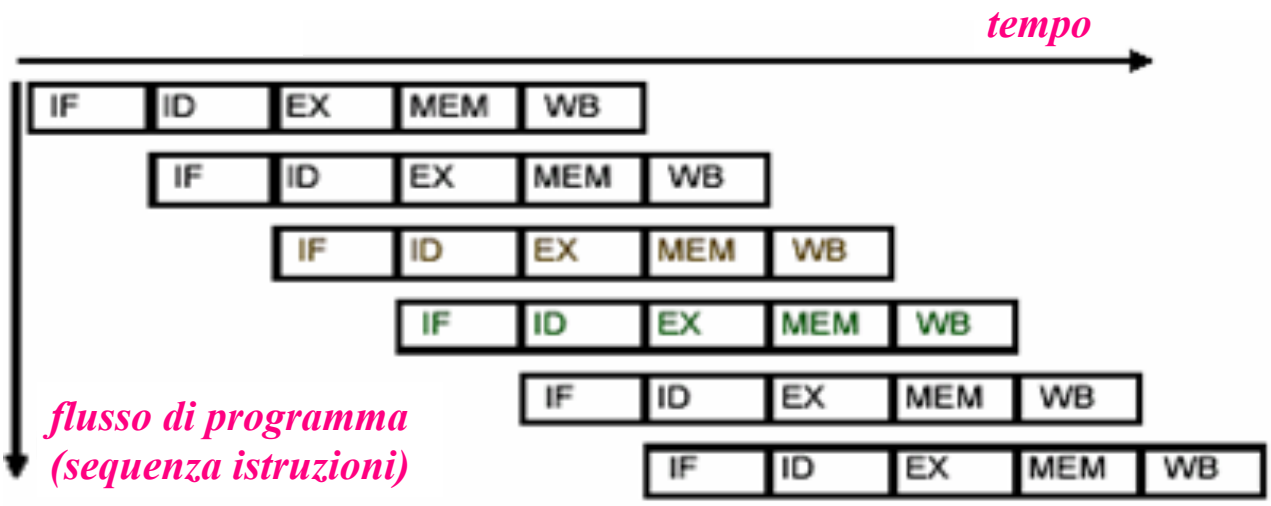
- Ogni stadio inizia il suo lavoro quando il clock fa scattare i registri di pipeline e trasferisce in quello stadio l'elaborazione dello stadio precedente



Rappresentazione del funzionamento di CPU pipeline

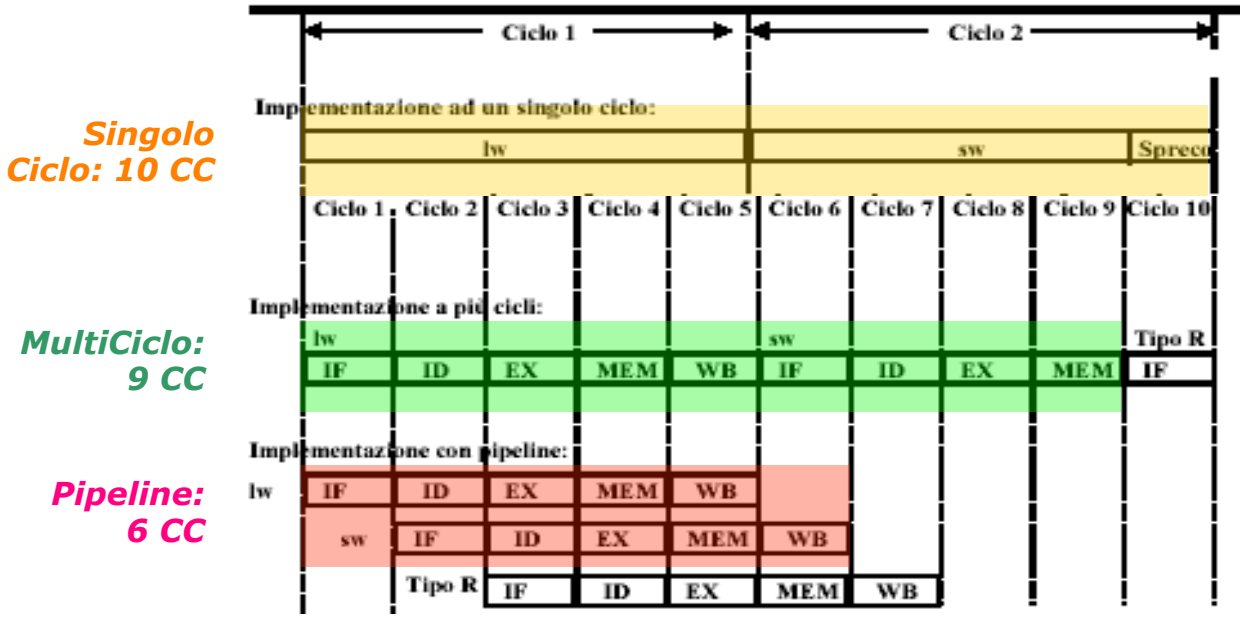
Funzionamento pipeline: rappresentazione convenzionale:

- Ascisse **X:** **tempo** (cicli di clock)
- Ordinate **Y:** **flusso di programma** (sequenza istruzioni)



I vantaggi della pipeline

Vantaggi pipeline: a regime, 1 istruzione/CC



Miglioramento massimo: riduzione del tempo/istruzione di un fattore pari al **numero di stadi** della pipe-line

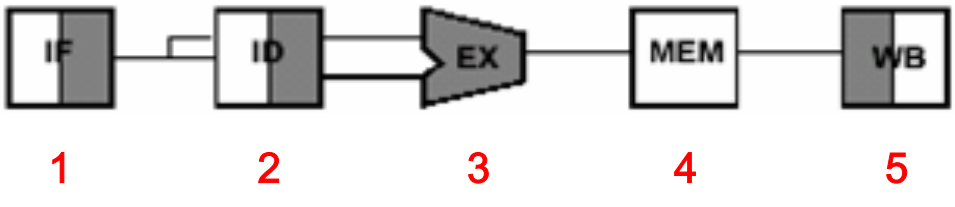
Rappresentazione grafica della pipeline

Rappresentazione grafica *del funzionamento pipeline.*

utile per individuare le risorse impegnate ad ogni stadio

- Rettangoli significano memoria,
  trapezio significa elaborazione (ALU)
- Componenti bianchi → non utilizzati ; grigi → in utilizzo
- I rettangoli grigi a destra indicano lettura (di memoria/registro)
  grigi a sinistra indicano scrittura

Esempio: `add $s0, $t0, $t1`



CPU pipeline: criticità strutturali

Quali unità funzionali (ALU, RF, Mem) vengono utilizzate in ciascuna fase?

PROBLEMA: se istruzioni diverse (in fasi diverse) necessitano contemporaneamente di una stessa risorsa, la risorsa va duplicata!

Tale problema è detto: **criticità strutturale**

istruzioni → fase ↓	Istruzioni A/L Tipo R	Istruz. di accesso a memoria (`lw`,`sw`)	Salto condizionato	Salto non condizionato
Fetch	IR ← **Memory**[PC] ; PC ← PC + 4			
Decodifica / Prelievo dati	A ← Reg[IR[25-21]] B ← Reg[IR[20-16]] ALUout ← PC + (sign_ext(IR[15-0]) << 2)			
Esecuzione	ALUOut ← A **oper** B	ALUOut ← A + sign_ext(IR[15-0])	If (A==B) then PC ← ALUOut	PC ← PC[31-28] || IR[25-0]<<2
`lw/sw`: mem tipo R: WB	---	**`lw`**: MDR←**Memory**[ALUOut] **`sw`**: **Memory**[ALUOut] ← B	---	---
`lw/sw`: WB	Reg(IR[15-11])← ALUOut	**`lw`**: Reg[IR[20-16]] ← MDR	---	---

Criticità **(hazards)**

Esistono tre tipi di Criticità (Hazards):

- **Strutturali**
  - Dovrei utilizzare la stessa unità funzionale due volte nello stesso passo.
  - Le unità funzionali non sono in grado di supportare le istruzioni (nelle diverse fasi) che devono essere eseguite in un determinato ciclo di clock.
- **di Dato**
  - Devo eseguire un'istruzione in cui uno degli operandi è il risultato dell'esecuzione di un'istruzione precedente.
  - `lw  $t0, 16($s1)`
  - `add $t1, $t0, $s0`
- **di Controllo/salto**
  - Devo prendere una decisione (sull'istruzione successiva) prima che l'esecuzione dell'istruzione sia terminata (e.g. **`branch`**) → generazione di situazioni di **criticità (hazard)**

Utilizzo unità funzionali della CPU

Quali unità funzionali (ALU, RF, Mem) vengono utilizzate in ciascuna fase?

Φ	Passo esecuzione	ALU	Memoria	Register File
1	**Fase fetch**	Sì (PC+4)	Sì (istr.)	No
2	**Decodifica**	No / Sì (salto)	No	Sì (lettura)
3	**Exec I – `beq`**	Sì (test)	No	No
	Exec I – `jump`	No	No	No
	Exec I – `tipo R`	Sì (operazione)	No	No
	Exec I – `sw`	Sì (indirizzo)	No	No
	Exec I – `lw`	Sì (indirizzo)	No	No
4	**Exec II – `tipo R`**	No	No	Sì (scrittura)
	Exec II – `sw`	No	Sì (dati)	No
	Exec II – `lw`	No	Sì (dati)	No
5	**Exec III – `lw`**	No	No	Sì (scrittura)

Pipeline per l'istruzione: `lw`

Esempio – consideriamo due istruzioni **`lw`** consecutive:

(senza anticipazione branch in ID)

```
0x300: lw $t0, 4($s0)
0x304: lw $t1, 8($s0)
```

Passo esecuzione	ALU	Memoria	Register File
IF (Fase fetch)	Sì	Sì	No
ID (Decodifica)	No	No	Sì
EX (Esecuzione)	Sì	No	No
MEM (Accesso memoria)	No	Sì	No
WB (riscrittura)	No	No	Sì

Tempo (CC) → ↓ Istruzioni	t_0	t_1	t_2	t_3	t_4	t_5
...						
`lw $t0, 4($s0)`	Mem, ALU	RF	ALU	Mem	RF	
`lw $t1, 8($s0)`		Mem, ALU	RF	ALU	Mem	RF
...						

***OK**: nessuna criticità strutturale.*

Pipeline per l'istruzione: `lw`

Esempio – consideriamo due istruzioni **`lw`** consecutive:

(**con** anticipazione branch in ID)

```
0x300: lw $t0, 4($s0)
0x304: lw $t1, 8($s0)
```

Passo esecuzione	ALU	Memoria	Register File
IF (Fase fetch)	Sì	Sì	No
ID (Decodifica)	**Sì**	No	Sì
EX (Esecuzione)	Sì	No	No
MEM (Accesso memoria)	No	Sì	No
WB (riscrittura)	No	No	Sì

Tempo (CC) → / ↓ Istruzioni	t_0	t_1	t_2	t_3	t_4	t_5
...						
`lw $t0, 4($s0)`	Mem, ALU	RF, **ALU**	ALU	Mem	RF	
`lw $t1, 8($s0)`		Mem, **ALU**	RF	ALU	Mem	RF
...						

Criticità strutturale!

CPU pipeline: criticità strutturali

Possibile approccio alla soluzione: provare a modificare la posizione delle singole micro-operazioni di un'istruzione:

Es: calcolo indirizzo di **`beq`** a fase 3 ➔ mi servono **2 ALU allo stadio 3!**

istruzioni → / fase ↓	Istruzioni A/L Tipo R	Istruz. di accesso a memoria (**`lw,sw`**)	Salto condizionato	Salto non condizionato
Fetch	IR ← **Memory**[PC] ; PC ← PC + 4			
Decodifica / Prelievo dati	A ← Reg[IR[25-21]] B ← Reg[IR[20-16]]			
Esecuzione	ALUOut ← A **oper** B	ALUOut ← A + sign_ext(IR[15-0])	ALUout ← PC + (sign_ext(IR[15-0]) << 2) If (A==B) then PC ← ALUOut	PC ← PC[31-28] || IR[25-0]<<2
`lw/sw`: mem tipo R: WB	---	**`lw`**: MDR←**Memory**[ALUOut] **`sw`**: **Memory**[ALUOut] ← B	---	---
`lw/sw`: WB	Reg(IR[15-11])← ALUOut	**`lw`**:Reg[IR[20-16]] ← MDR	---	---

Criticità strutturali: esec. istruzioni `lw`

Esempio: istruzioni consecutive `lw, branch, add, lw, ...`

`lw`: passo esecuzione	ALU			Memoria		Register File
	ALU	PC	branch	Dati	Istruzioni	
IF (Fase fetch)	No	Sì	No	No	Sì	No
ID (Decodifica)	Sì	No	No	No	No	Sì (READ)
EX (Esecuzione)	Sì	No	No	No	No	No
MEM (Acc memoria)	No	No	No	Sì	No	No
WB (riscrittura)	No	No	No	No	No	Sì (WRITE)

`lw $t0, 8($t2)`	ALU (PC) Mem	RF read	ALU (addr.)	Mem	RF write		
`beq $1, $0, +16`		ALU (PC) Mem	RF read	ALU addr. ALU zero	—	—	
`add $1, $2, $0`			ALU (PC) Mem	RF read	ALU	—	RF write
`lw $t1, 4($t2)`				ALU (PC) Mem	RF read	ALU	Mem

Presenza di criticità strutturali su: ALU, MEMORIA

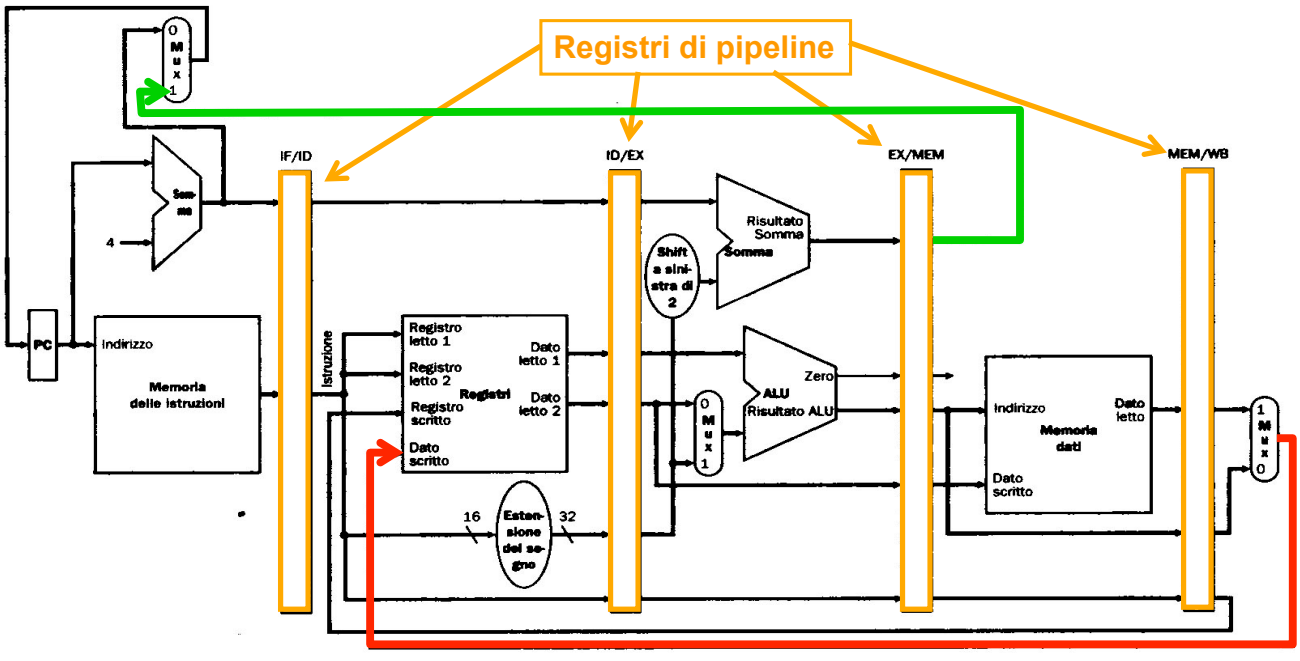
Nessun problema con **RF**: **RF-write** e **RF-read** possibili contemporaneamente.

Soluzione: replicazione delle risorse: 3 ALU, 2 MEMORIE

ALU / ALU (PC) / ALU (address) Memoria DATI / Memoria ISTRUZIONI

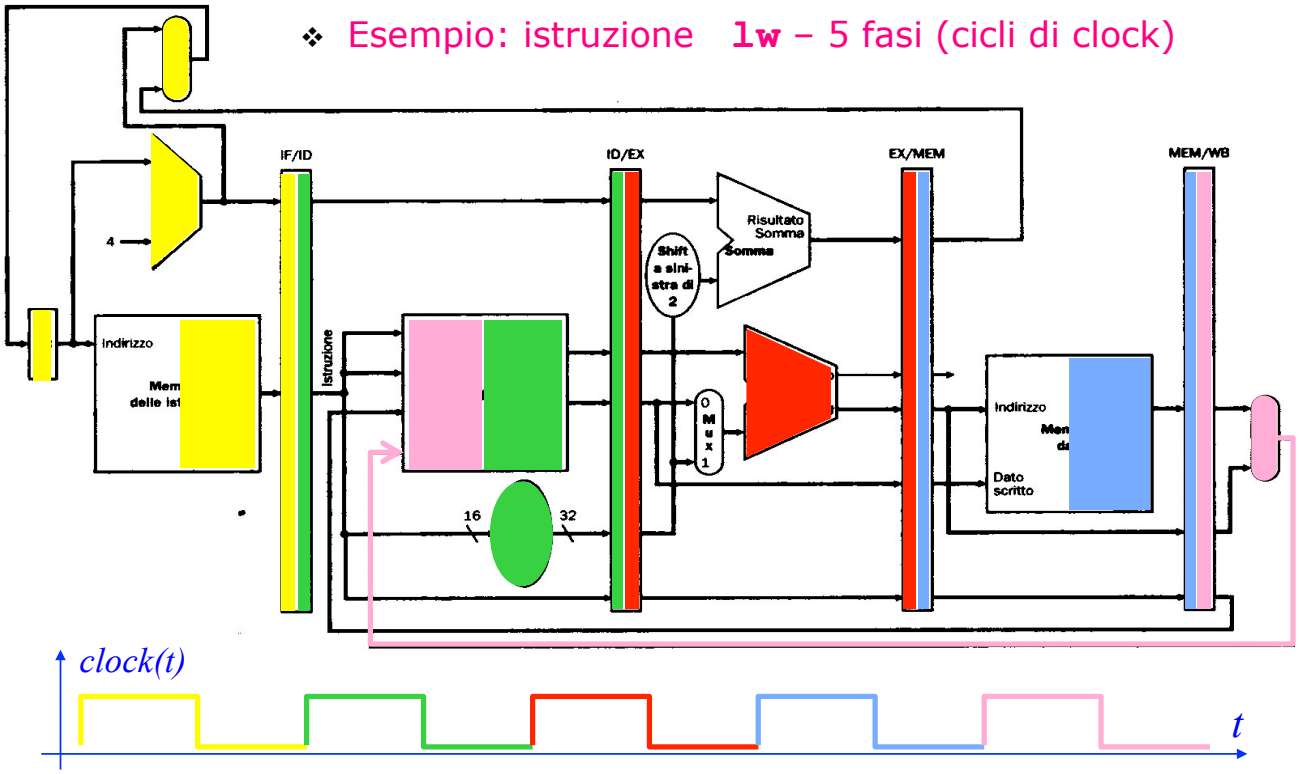
CPU pipeline – soluzione criticità strutturali

- Duplicazione **memoria** (**Dati** / **Istruzioni**)
- Triplicazione **ALU** (**ALU**, **ALU-PC**, **ALU-address**)



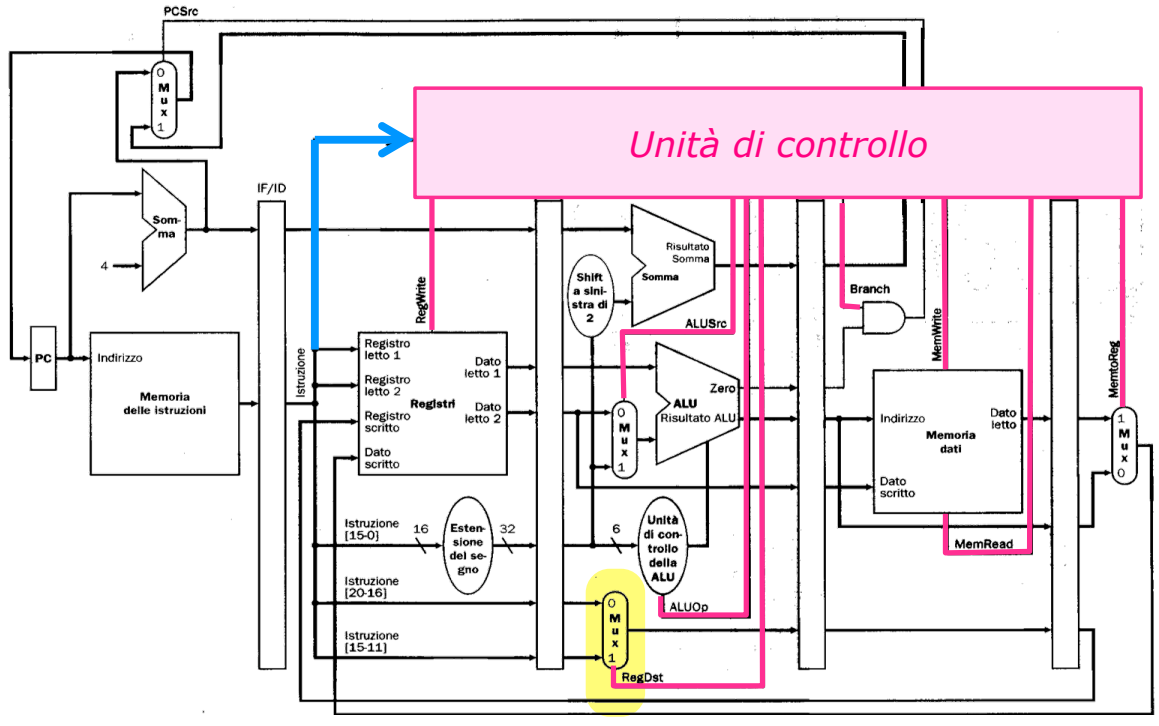
Funzionamento CPU pipeline

- Esempio: istruzione **`lw`** – 5 fasi (cicli di clock)



CPU Pipeline – schema completo

CPU pipeline – schema completato, con unità di controllo



Segnali di controllo – CPU pipeline

CPU pipeline: segnali di controllo (selezione e comando)

Segnale	Effetto quando è = 0	Effetto quando è = 1
RegDst	Il numero del registro destinazione proviene dal campo `rt` (bit 20-16)	Il numero del registro destinazione proviene dal campo `rd` (bit 15-11)
ALUSrc	Il secondo operando della ALU proviene dalla seconda uscita in lettura del RF	Il secondo operando della ALU è la versione estesa (con segno) del campo offset
ALUOp	Selezione dell'operazione effettuata dalla ALU principale	
Branch	Il valore del PC viene sostituito dall'uscita del sommatore che calcola **PC + 4** (condizionato all'uscita di ALU)	Il valore del PC viene sostituito dall'uscita del sommatore che calcola l'**indirizzo di salto** (condizionato all'uscita di ALU)
MemtoReg	Il valore inviato all'ingresso Dato al RF proviene dalla ALU	Il valore inviato all'ingresso DatoScritto al RF proviene dalla memoria
MemRead	Nessuno	Il contenuto della cella di memoria dati indirizzata dal MAR è posto nel MDR
MemWrite	Nessuno	Il contenuto in ingresso al MDR viene memorizzato nella cella il cui indirizzo è caricato nel MAR
RegWrite	Nessuno	Nel registro specificato a #RegWrite viene scritto il valore presente all'ingresso DatoScritto

PC (PCWrite) e **registri di pipeline** ➔ scrittura ad **ogni ciclo di clock**

CPU pipeline: progetto unità di controllo

Progetto **Unità di Controllo** della CPU pipeline:

Specifiche di funzionamento:

- Ogni istruzione necessita dei propri segnali di controllo
- Ogni segnale di controllo è utilizzato in una sola fase
- Segnali per una delle tre fasi: **EX**, **MEM** o **WB**

SOLUZIONE:

una volta generati i segnali di controllo di un' istruzione...

- **nella fase di DECODIFICA (ID)**

...li posso "trasportare" attraverso la pipeline insieme all' istruzione stessa

- **nelle fasi successive (EX, MEM, WB)**

Osservazioni

Classificazione dei segnali di controllo in base alla fase nella quale sono attivi:

Nelle fasi di fetch (**IF**) e decodifica (**ID**) non esistono segnali di controllo

nella fase **EX:** **RegDst, ALUOp, ALUSrc**

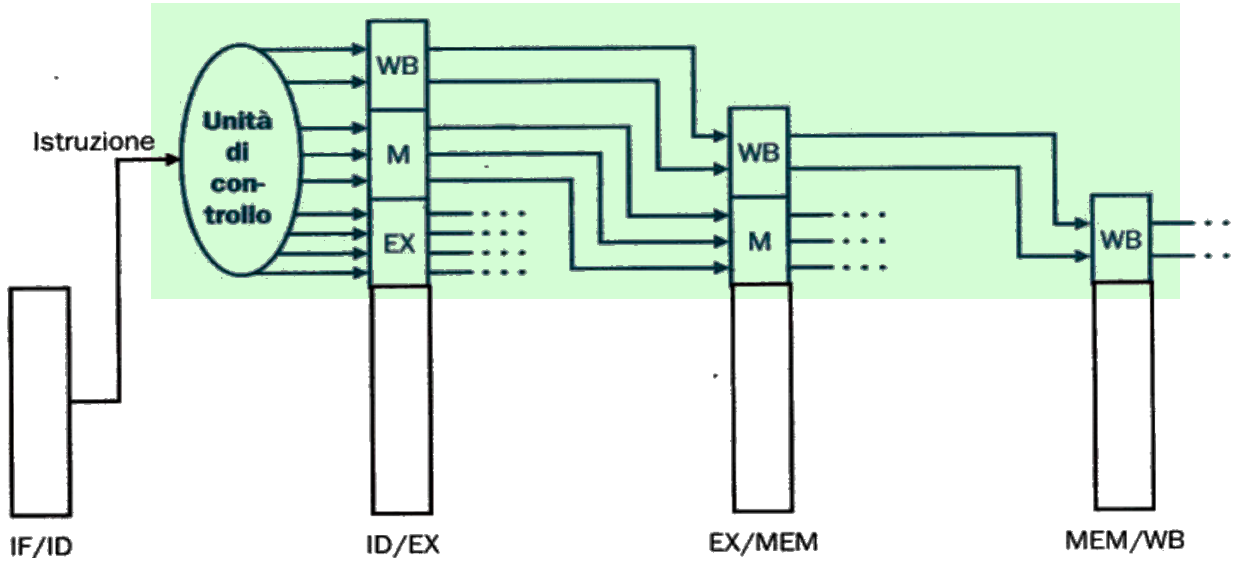
nella fase **MEM:** **MemRead, MemWrite, Branch**

nella fase **WB:** **RegWrite, MemToReg**

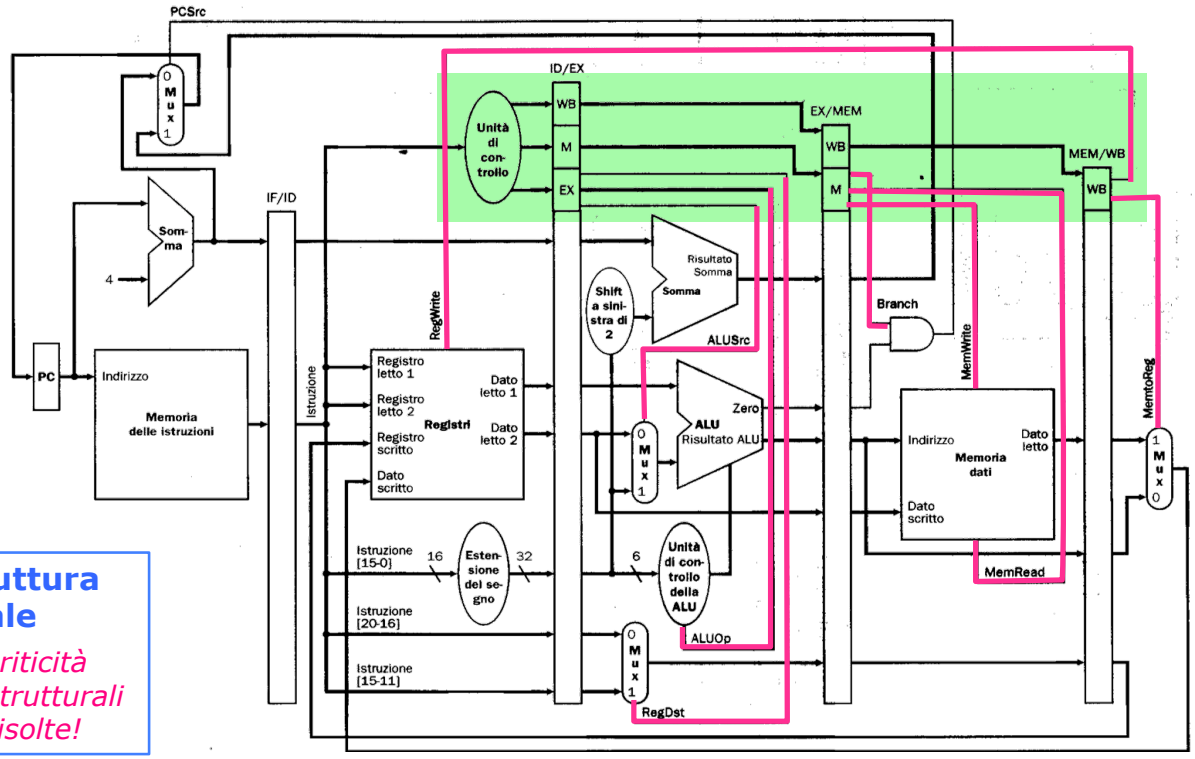
Fase →	Exec				Memory			WB	
Segnali → ↓ Istruz.	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem-Read	Mem-Write	Reg-Write	MemTo-Reg
Tipo R	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	x	0	0	1	0	0	1	0	x
beq	x	0	1	0	1	0	0	0	x

Generazione dei segnali di controllo

- I segnali di controllo vengono generati nello stadio di decodifica e poi propagati agli stadi successivi.



UC per CPU Pipeline



Struttura finale

- *criticità strutturali risolte!*

Criticità **(hazards)**

- **Criticità strutturali**
  - Necessità di una stessa unità funzionale per più compiti nello stesso istante (ciclo di clock).
  - Risolte con la duplicazione (suddivisione) delle unità funzionali
  - **3 ALU, 2 memorie**, come nella CPU singolo ciclo → **criticità risolte!**
- **Criticità di dato:**

  devo eseguire un'istruzione in cui uno degli operandi è il risultato dell'esecuzione di un'istruzione precedente.

```
lw  $t0, 16($s1)
add $t1, $t0, $s0
```

- **Criticità di controllo/salto**
  - Devo prendere una decisione (sull'istruzione successiva) prima che l'esecuzione dell'istruzione sia terminata (e.g. **`branch`**) → generazione di situazioni di **criticità (hazard)**

Stallo della pipeline

Le criticità possono anche costringere ad uno STALLO:

Se, ad un certo istante di clock, l'istruzione successiva non può essere eseguita,

la pipeline deve far aspettare l'istruzione ➔ una delle istruzioni si "ferma ad aspettare" ➔ **l'istruzione deve andare "in stallo"**

- Nel flusso della pipeline si generano i **buchi** o **bolle** (o ***bubbles***)

Tempo → ↓ Istruzioni	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
…..								
`lw $t0, 8($s0)`	FF (Mem, ALU)	DEC (RF)	EXEC (ALU)	MEM (Mem)	WB (RF)	`$t0` disponibile		
`add $t1, $t0, $s0`		FF	DEC	Buco (EXEC)	Buco (MEM)	Buco (WB)		
`add $t1, $t0, $s0`			Buco	Buco	Buco	Buco	Buco	
`add $t1, $t0, $s0`	devo aspettare `$t0` !			Buco	Buco	Buco	Buco	Buco
`add $t1, $t0, $s0`					FF	DEC	EXEC	MEM

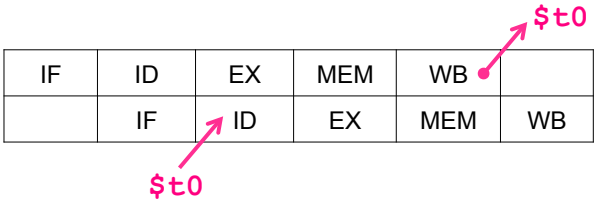
Criticità nella CPU

Criticità di dati: devo eseguire un'istruzione in cui uno degli operandi è il risultato di un'istruzione precedente.

```
Es:   lw  $t0, 16($s1)
      add $t1, $t0, $s0
```

- `$t0` viene scritto nella fase WB di `lw`...
- ...ma `$t0` viene già letto nella fase ID di `add`!
- **WB** porta informazioni a ritroso → possibile causa di problemi.

```
lw $t0, 16($s1)
add $t1,$t0,$s0
```



Soluzioni possibili:

- SW: aspettare ad eseguire **add**
- SW: riorganizzazione automatica del codice (compilatore).
- **HW: propagazione *(forwarding)***

Sommario

Caso I:

- Criticità nei dati causate da istruzioni **A/L**

Es: `add $9, $8, $16`

`add $20, $9, $0`

Esempio: *criticità di dati – istruzioni A/L*

- Il dato in **$2** diviene disponibile nel RF nella fase **WB** della **sub**.
  - Non è ancora pronto quando viene effettuata la decodifica della **and** e della **or** successive.
- Hazards: tra **sub** e **and** e tra **sub** e **or**
  - Con le frecce sono indicate le dipendenze, in rosso gli hazard

`sub $2, $1, $3`	IF	ID	EX $1–$3	MEM	**WB s→$2**				
`and $12, $2, $5`		IF	ID	EX $2 **and** $5	MEM	WB s→$12			
`or $13, $6, $2`			IF	ID	EX $6 **or** $2	MEM	WB (s→$13)		
`add $14, $2, $2`				IF	ID	EX $2+$2	MEM	WB s→$14	
`sw $15, 100($s2)`					IF	ID	EX $2+100	MEM $15→Mem	WB

Soluzione *SW*: *riorganizzazione del codice*

Soluzione SW:

1. Inserimento **nop** →

```
sub $2, $1, $3
nop
nop
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($s2)
```

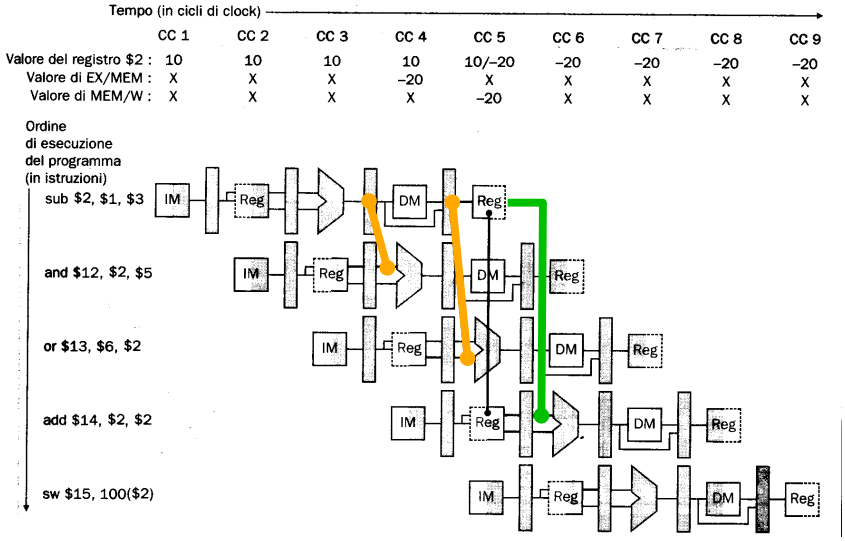
Spreco di 2 cicli di clock, ma ottengo che:
la fase **ID** dell'istruzione: **`and $12, $2, $5`**
coincide con la fase di **WB** della: **`sub $2, $1, $3`**

➔ situazione troppo frequente perché la soluzione sia accettabile.

Soluzione *HW*: *forwarding*

Soluzione HW: Forwarding

- Si invia il **risultato dell'istruzione precedente** al passo intermedio **dell'istruzione attuale** (indietro nellla pipeline).
- Il contenuto dei registri **EX/MEM** e **MEM/WB** anticipa il contenuto del Register File.



Soluzione HW: forwarding

`sub $2, $1, $3`	IF	ID	EX $1–$3	MEM	**WB s→$2**				
`and $12, $2, $5`		IF	ID	EX **$2 and** $5	MEM	WB s→$12			
`or $13, $6, $2`			IF	ID	EX $6 **or** $2	MEM	WB (s→$13)		
`add $14, $2, $2`				IF	ID	EX $2+$2	MEM	WB s→$14	
`sw $15, 100($s2)`					IF	ID	EX $2+100	MEM $15→Mem	WB

FORWARDING da **EX/MEM:**

- Prendo il risultato della sottrazione all'uscita del registro **EX/MEM** (per istruzioni A/L, lo stadio MEM è di attesa): `$1-$3` è già contenuto nel registro EX/MEM e si legge all'uscita
- Sostituisco il primo operando della **`and`** successiva (`$2`) con il risultato della sottrazione eseguita in precedenza, senza attendere la fase di **WB**

Hazard tra `sub` e `or`: forwarding

`sub $2, $1, $3`	IF	ID	EX $1–$3	MEM	**WB s→$2**				
`and $12, $2, $5`		IF	ID	EX $2 **and** $5	MEM	WB s→$12			
`or $13, $6, $2`			IF	ID	EX $6 **or $2**	MEM	WB (s→$13)		
`add $14, $2, $2`				IF	ID	EX $2+$2	MEM	WB s→$14	
`sw $15, 100($s2)`					IF	ID	EX $2+100	MEM $15→Mem	WB

FORWARDING da **MEM/WB:**

- Prendo il risultato della sottrazione (`$1-$3`) dall'uscita del registro **MEM/WB**.
- Sovrascrivo il secondo operando della **`or`** successiva (`$2`) con il risultato della sottrazione eseguita in precedenza, senza attendere la fase di **WB**.

Forwarding e contenuto del registro **ID/EX**

Datapath delle operazioni di **forwarding:**

Osservazioni:

- Nel normale funzionamento, il registro **ID/EX** contiene quanto letto dal Register File.
- Quando abbiamo un forwarding, quello che viene letto dal registro **ID/EX** nella fase di esecuzione deve essere **sostituito** da quanto letto dal registro **EX/MEM** o **MEM/WB**.
- Nel registro **EX/MEM** è contenuto il risultato dell'operazione eseguita **1 ciclo** di clock prima
- Nel registro **MEM/WB** è contenuto il risultato dell'operazione eseguita **2 cicli** di clock prima

➔ ***Trasferimento dati dagli stadi EX/MEM e MEM/WB allo stadio ID/EX***

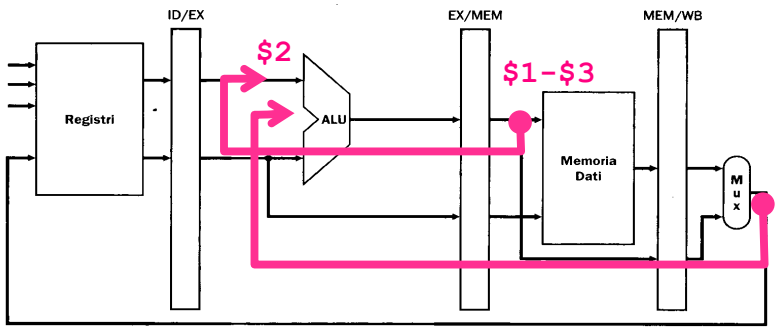
Hazard nei dati, *progetto soluzione HW*

Datapath:
collegamenti per trasferire i dati tra gli stadi di pipeline

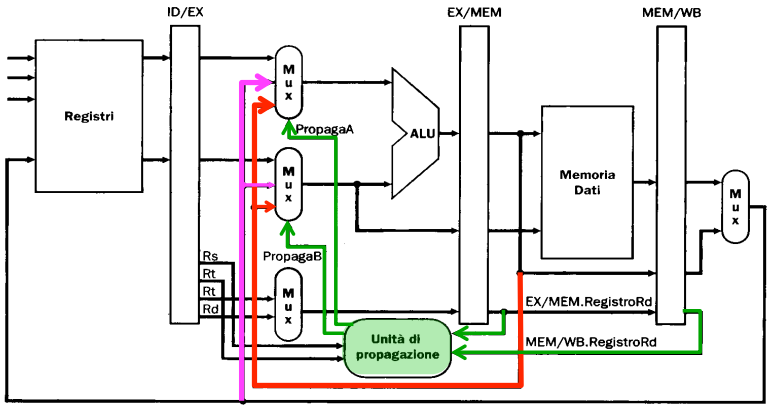
da EX/MEM.**ALUout**
a ID/EX.**ALUin**

da MEM/WB.**MEMout**
a ID/EX.**ALUin**

Controllore:
selezione opportuna operandi
↓
Unità di Propagazione (Forwarding Unit)



a. Senza propagazione



Controllo MUX ingresso alla ALU

- Nuovi segnali di controllo: PropagaA, PropagaB
  - generati dall' unità di propagazione

Controllo Multiplexer	Registro Sorgente	Funzione
PropagaA = 00	**ID/EX**	Il primo operando della ALU proviene dal Register File
PropagaA = 10	**EX/MEM**	Il primo operando della ALU è propagato dal risultato della ALU per l'istruzione precedente.
PropagaA = 01	**MEM/WB**	Il primo operando della ALU è propagato dalla memoria o da un'altra istruzione precedente.
PropagaB = 00	**ID/EX**	Il secondo operando della ALU proviene dal Register File
PropagaB = 10	**EX/MEM**	Il secondo operando della ALU è propagato dal risultato della ALU per l'istruzione precedente.
PropagaB = 01	**MEM/WB**	Il secondo operando della ALU è propagato dalla memoria o da un'altra istruzione precedente.

Soluzione architetturale per **AND**

Hazard su AND:

- Occorre implementare la seguente funzione logica nella fase ID dell'esecuzione dell'istruzione:

```
MEM:  sub $2, $1, $3     (nel registro: EX/MEM)
EXE:  and $12, $2, $5    (nel registro: ID/EX)
```

if ($rd_{EX/MEM} == rs_{ID/EX}$)
then "**collega**": $ALUout_{EX/MEM} \rightarrow ALU_A$ → **PropagaA = 10**

if ($rd_{EX/MEM} == rt_{ID/EX}$)
then "**collega**": $ALUout_{EX/MEM} \rightarrow ALU_B$ → **PropagaB = 10**

- $rd_{MEM} = rs_{EX} = \$2$ è il contenuto del registro destinazione dell'istruzione precedente la `and` (`sub $2, $1, $3`)
- all'istante ***t*** è contenuto nel registro EX/MEM

Soluzione architetturale per OR

Hazard su OR:

- Occorre implementare la seguente funzione logica nella fase ID dell'esecuzione dell'istruzione:

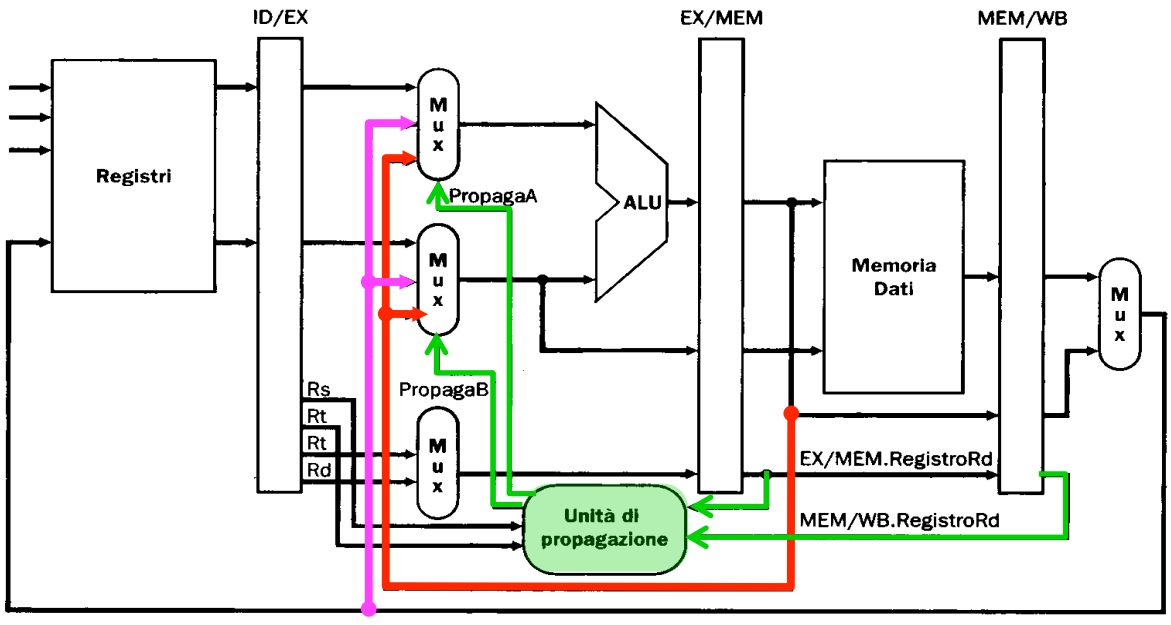
```
WB:   sub $2, $1, $3     (nel registro: MEM/WB)
MEM:  ...                (nel registro: EX/MEM)
EXE:  or $13, $6, $2     (nel registro: ID/EX)
```

if ($rd_{MEM/WB}$ == $rs_{ID/EX}$)
then "**collega**" $ALUout_{MEM/WB} \rightarrow ALU_A$ → **PropagaA = 01**

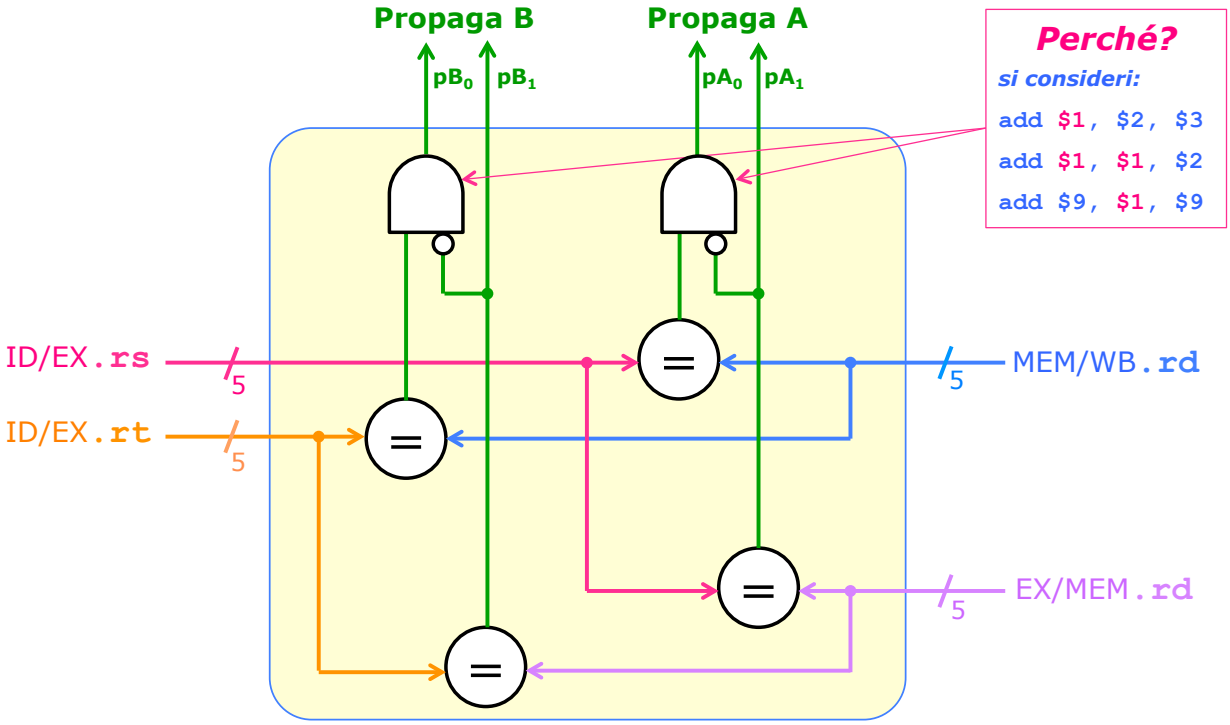
if ($rd_{MEM/WB}$ == $rt_{ID/EX}$)
then "**collega**" $ALUout_{MEM/WB} \rightarrow ALU_B$ → **PropagaB = 01**

- $rd_{WB} = rt_{EX} = \$2$ è il contenuto del registro destinazione di 2 istruzioni precedenti la `or: (sub $2, $1, $3)`
- all'istante ***t*** è contenuto nel registro MEM/WB

CPU con forwarding: schema circuitale



Forwarding Unit: progetto circuitale



Data Hazard: soluzioni

Sintesi: soluzione del problema della
criticità di dato (Data Hazard) per istruzioni A/L

Software:

- Inserire una **`nop`**
- Buona scrittura del codice

  il programmatore deve conoscere la macchina per scrivere un buon codice!
- Compilatore efficiente (che riordina il codice per minimizzare hazards)

Hardware:

- FORWARDING

  architettura che rende disponibile i dati appena pronti alla fase di esecuzione.

Caso II:

- Criticità nei dati causate da istruzioni: **lw**

```
Es:     lw $10, 0($8)
        addi $4, $10, +16
```


Diagramma delle dipendenze:

- in verde le dipendenze gestite regolarmente
- in rosso le criticità (hazards) di dato

- Il dato corretto è pronto nella **lw** alla fine della MEM
  - mentre nel caso di istruzione tipo R era pronto alla fine di EX

`lw $2, 40($3)`	IF	ID	EX $3+40	MEM	WB s→ $2				
`and $9,$2,$5`		IF	ID	EX $2 and $5	MEM	WB s→$t2			
`or $10,$6,$2`			IF	ID	EX $6 or $2	MEM	WB s→$10		
`add $11,$2,$2`				IF	ID	EX $2 +$2	MEM	WB s → $11	
`sw $15, 100($2)`					IF	ID	EX $2+100	MEM $15→Mem	WB

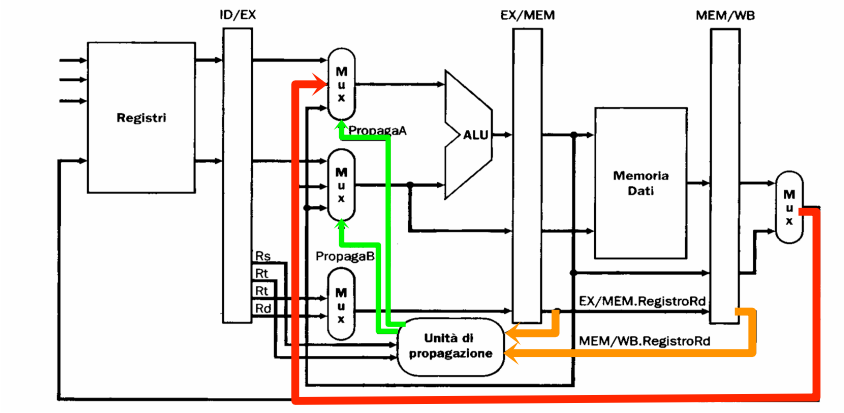
Hazard sui dati: `lw` – forwarding

`lw $2, 40($3)`	IF	ID	EX $3+40	MEM	WB s→ $2			
`and $9, $2, $5`		IF	ID	EX $2 and $5	MEM	WB s→$9		
`or $10, $6, $2`			IF	ID	EX $6 or $2	MEM	WB s→$10	
`add $15, $2, $2`				IF	ID	EX $2 +$2	MEM	WB s → $15

- Il dato corretto per **`$s2`** è pronto nella **`lw`** solamente alla fine della **MEM**, all'uscita del registro **MEM/WB**
- Rilevo la criticità su **`or`** alla fine della sua fase di **ID**.
  In questo caso il dato corretto si trova all'inizio della fase WB della **`lw`**
  - Posso risolvere la criticità con il **FORWARDING**
- Rilevo la criticità su **`and`** alla fine della sua fase di **ID**.
  In questo caso il dato corretto non è ancora stato prodotto dalla **`lw`**

➔ NON posso risolvere la criticità → **STALLO**

Soluzione criticità `or` con forwarding

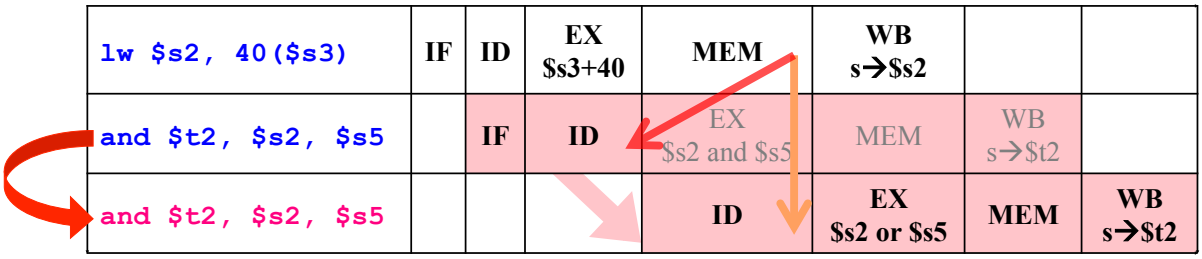


Risolvo la criticità per la **`or`** mediante propagazione (forwarding)

- **Datapath:** non cambia rispetto al caso precedente
- **UC** (unità di propagazione): prelievo dalla fase **WB**

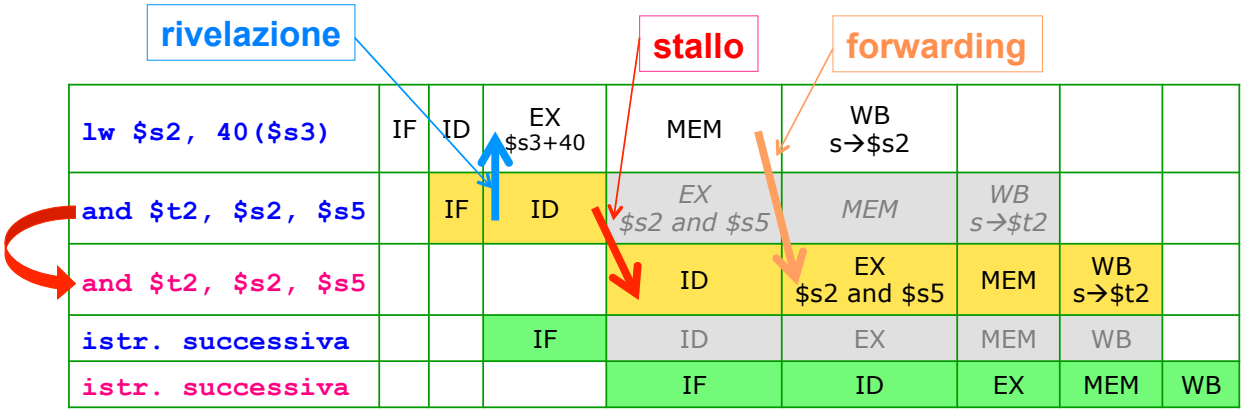
```
if (RD_{t-2} == RS_t)
        then "collega":        ALUout_WB con ALU_A
if (RD_{t-2} == RT_t)
        then "collega":        ALUout_WB con ALU_B
```

Hazard sui dati – `lw`: **stallo**



- Il dato corretto per `$s2`:
  è pronto nella `lw` solamente alla fine della fase **MEM**,
  ed è perciò utilizzabile solamente a partire dall'inizio della fase di **WB**
- **Soluzione: 1 ciclo di STALLO + FORWARDING**
  - ➔ **Stallo della pipeline!**
    - Devo bloccare l'esecuzione della `and` e ripeterla un ciclo dopo
      - solo quando è possibile utilizzare il valore corretto del registro `$s2`

Rivelazione della criticità su `lw`



Devo rivelare la criticità prima possibile, in modo da mettere in stallo in tempo la pipeline

Il primo momento possibile è:

- stadio di decodifica (ID) dell'istruzione **`and`**
- l'istruzione **`lw (t–1)`** è già in stadio di esecuzione (EX)

Rivelazione della criticità su `lw`

IF *[(ID/EX.MemRead)]*
AND
{[IF/ID.reg_rt == ID/EX.reg_rt] ***OR***
[IF/ID.reg_rs == ID/EX.reg_rt] }

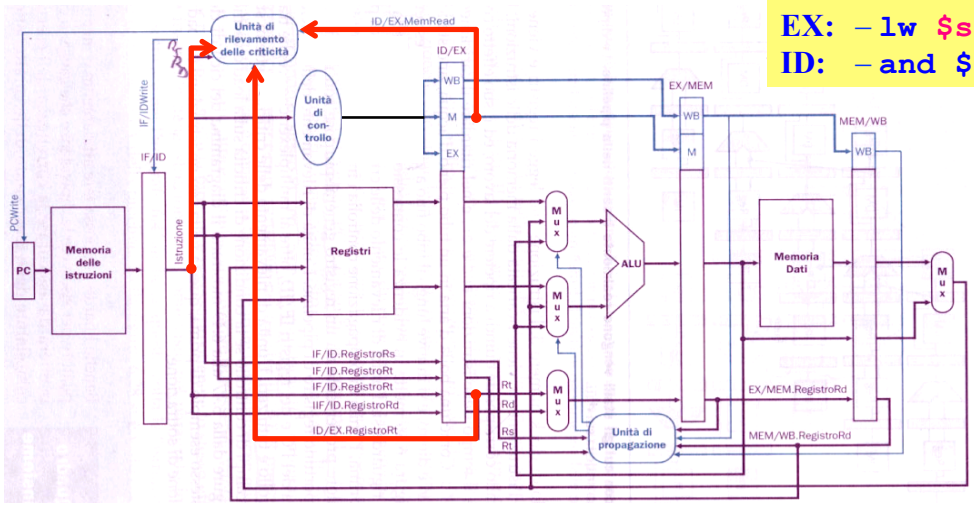
`lw` in fase di EX

registro destinazione di `lw`

`rd, rt`

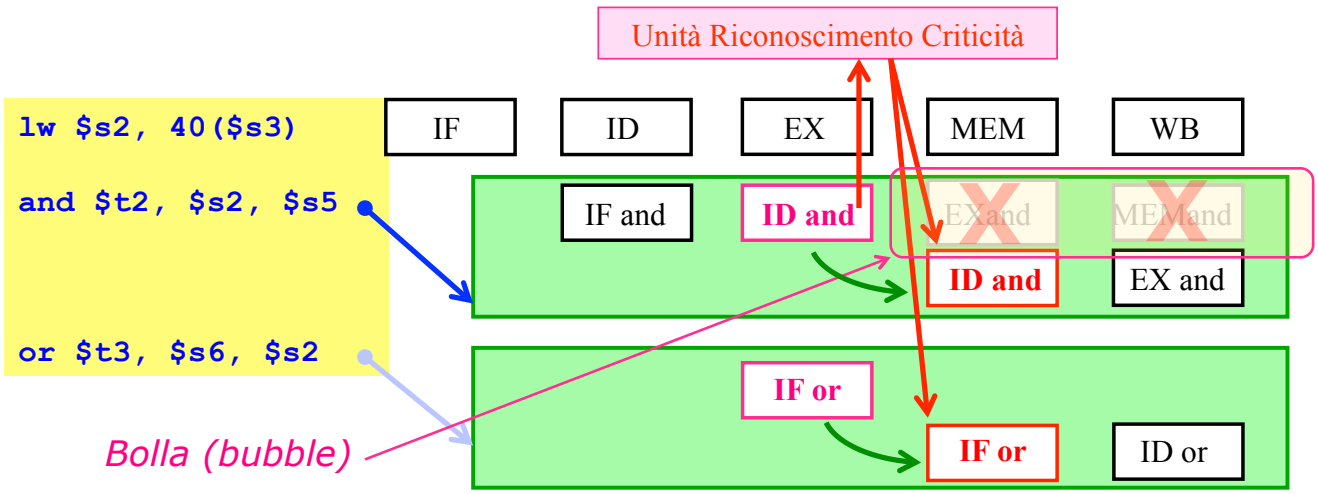
THEN *"**metti in stallo** lo **stadio ID** (e quindi anche **IF**)"*

```
EX: – lw $s2, 40($s3)
ID: – and $t2, $s2, $s5
```

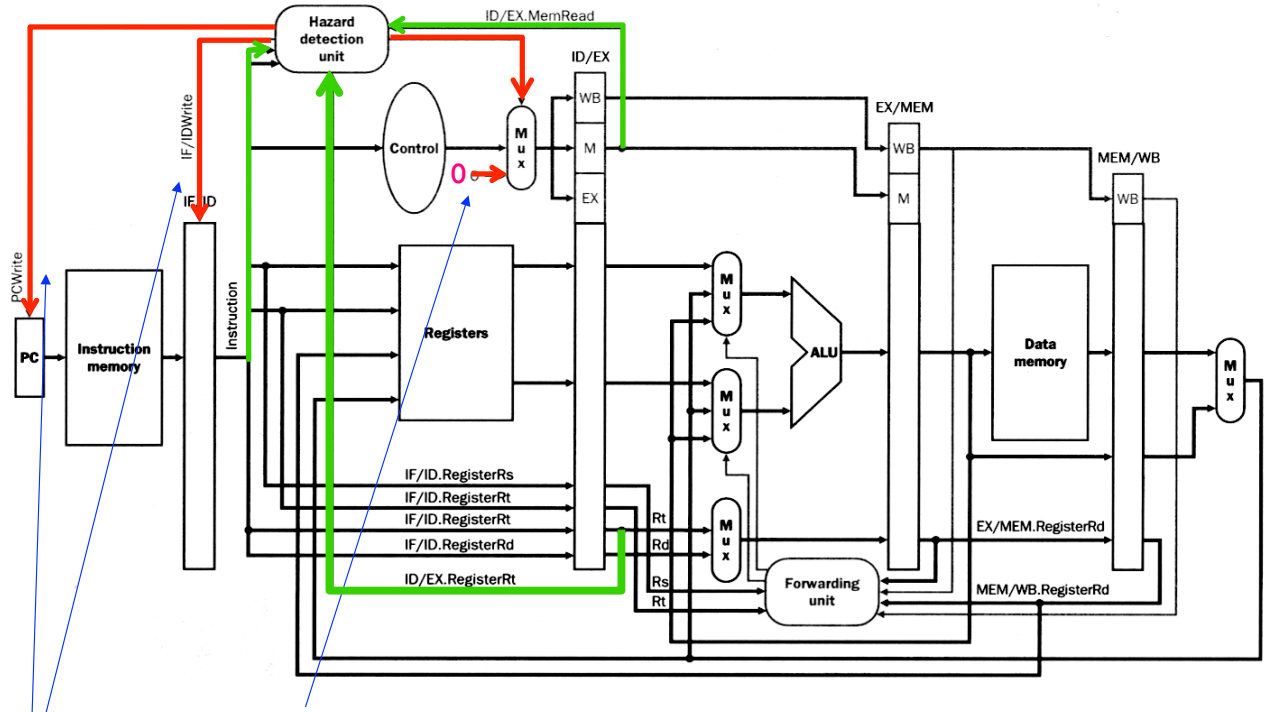


Stallo della pipeline

- **Implementazione dello STALLO** (generazione di una "**BUBBLE**")
  1. **Annullare i segnali di controllo dell'istruzione** in **ID (ID/EX)**
     ➔ segnali di controllo inattivi ➔ l'istruzione diventa ininfluente
  2. **Ripetere la decodifica (ID) e la lettura (IF) dell'istruzione successiva**
     - Si ripetono le fasi di **fetch(t–1)** e **decodifica(t–2)**



Gestione dello stallo



- **Annullamento** dei segnali di controllo associati → formazione della "bolla"
- **Inibizione** della **scrittura** del **PC** e del registro **IF/ID**

Criticità ***(hazards)***

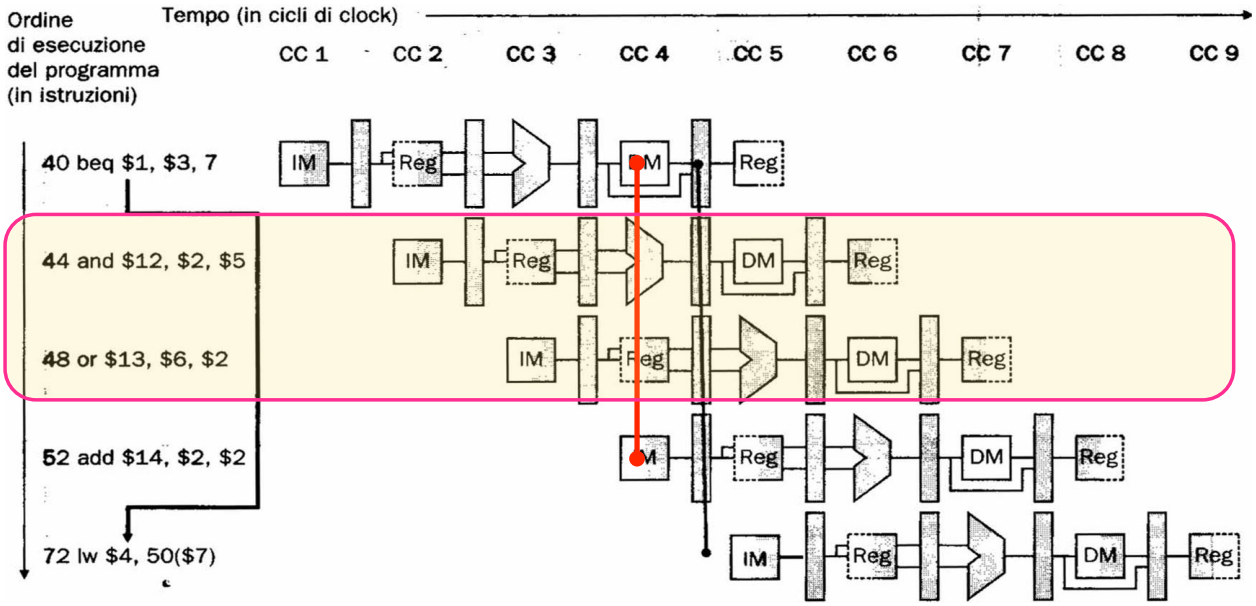
Esistono tre tipi di criticità (Hazards):

- **Strutturali**
  - Dovrei utilizzare la stessa unità funzionale due volte nello stesso passo.
  - Le unità funzionali non sono in grado di supportare le istruzioni (nelle diverse fasi) che devono essere eseguite in un determinato ciclo di clock.
- **di Dato**
  - Devo eseguire un'istruzione in cui uno degli operandi è il risultato dell'esecuzione di un'istruzione precedente.
  - `lw  $t0, 16($s1)`
  - `add $t1, $t0, $s0`
- **di Controllo/salto**
  - Devo prendere una decisione (sull'istruzione successiva) prima che l'esecuzione dell'istruzione sia terminata (e.g. **`branch`**) → generazione di situazioni di **criticità (hazard)**

Hazard: istruzione `lw` dopo `branch`

Criticità di salto (Control Hazard) – istruzioni: `branch/jump`

- *quando valuto la condizione, le istruzioni successive, **che forse (branch) / che sicuramente (jump) non dovevo eseguire**, sono già in pipeline!*



Criticità di salto (control hazard)

Criticità di salto

- **L'indirizzo di salto** eventuale viene calcolato nella **fase di EX** e viene memorizzato nel **registro EX/MEM**
- Il Program Counter può venire aggiornato al più durante la fase di MEM: in questo istante può iniziare il FETCH dell'istruzione all'indirizzo di salto.

  ➔ **ho DUE istruzioni da eliminare.**

`sub $s2,$s1,$s3`	IF	ID	EX $1–$3	MEM	WB ris.➔ $s2			
`beq $t2,$t6,24`		IF	ID	EX zero IF $t2=$t6	MEM	WB		
`or $t7,$s6,$s7`			IF	ID	EX $s6 or $s7	MEM	WB ris.➔ $t7	
`add $t4,$s8,$s8`				IF	ID	EX $s8+$s8	MEM	WB ris.➔ $t4
`add $t0,$t1,$t2`					IF	ID	EX	MEM

istruzione all'indirizzo di salto

Soluzioni alla criticità nel controllo

Soluzioni possibili...

...per **aggirare** la criticità

- Riordinamento del codice – **delayed branch**

(soluzione software)

...per **minimizzare le conseguenze** della criticità

- Modifiche strutturali della CPU per **anticipazione dei salti**
- Minimizzazione delle bolle in pipeline dovute ai branch: **Branch Prediction Table**

(soluzioni hardware)

Soluzioni: Delayed branch

Delayed branch: *decisione ritardata*

- ci si affida al compilatore

❖ Si aggiunge un: **"branch delay slot"**

- l' istruzione successiva ad un salto condizionato viene sempre eseguita
- il compilatore/assemblatore (o l'utente) sposta le istruzioni macchina, mettendo dopo l'istruzione di salto un'istruzione che andrebbe comunque eseguita, indipendentemente dal salto.

Delayed branch: esempio

```
      add $s4, $t0, $t1
      beq $s5, $s6, salto
      add $s0, $s0, $s1
salto:
      add $t5, $t4, $t3
      add $t6, $t7, $t7
      lw $a0, 0($t6)
```


```
      add $s4, $t0, $t1
      beq $s5, $s6, salto
      add $t5, $t4, $t3
      add $t6, $t7, $t7
      add $s0, $s0, $s1
salto:
      lw $a0, 0($t6)
```

Osservazioni:

- Le istruzioni `add $t5, $t4, $t3` e `add $t6, $t7, $t7` devono essere comunque eseguite.
- Il salto (se richiesto) avviene dopo.

Vincoli:

- L'istruzione, se presa prima, non deve avere **dipendenze** con il salto
- Se presa dopo, deve essere comunque eseguita

Gestione con modifiche strutturali della CPU

Soluzioni hardware:

1. Anticipazione dei salti:

- anticipazione della valutazione della condizione di branch
- **valutazione: da fase EXE (elimino 2 istruzioni)...**
- **...a fase DEC (elimino 1 istr.)**

2. Guessing (Branch Prediction):

- la pipeline cerca di indovinare se il salto debba essere eseguito
  - Es: si comporta come se il salto non dovesse essere eseguito
- Se "indovino": non ho vuotato la pipeline (50% dei casi)
- Se "sbaglio": devo scartare le istruzioni in corso

- Modifiche architetturali per scartare istruzioni già in corso

1. Anticipazione del salto:

Identificare l'hazard durante la fase **ID** di esecuzione della branch, anziché nella fase **EX**.

➔ necessario scartare **una sola** istruzione, anziché due!

`sub $s2,$s1,$s3`	IF	ID	EX $s1−$s3	MEM	WB ris.➔ $s2			
`beq $t2,$t5, 24`		IF	ID	EX zero IF $t2=$t5	MEM	WB		
`or $t7,$s6,$s7`			IF	ID	EX $s6 **or** $s7	MEM	WB ris.➔ $t7	
`add $t0,$t1,$t2`				IF	ID	EX $s8+$s8	MEM	WB ris.➔ $t4

Anticipazione del branch – datapath:

Nello stadio ID:

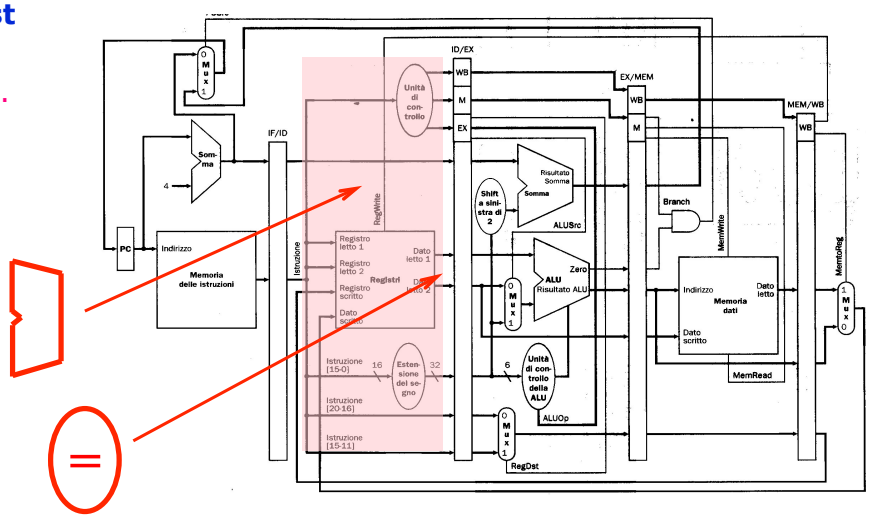
- HW per calcolare l'**indirizzo di salto**

➔ Una **ALU** per il calcolo dell'indirizzo di salto

- HW per valutare il **test**

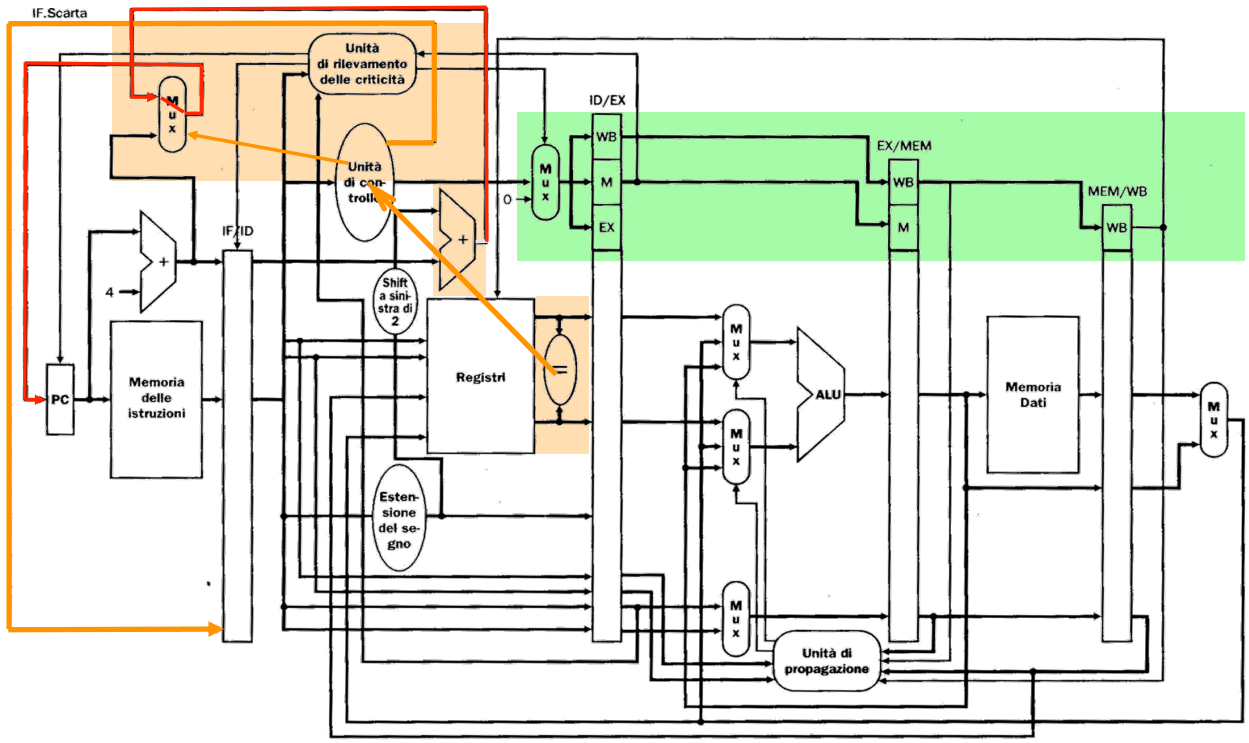
➔ un **comparatore** all'uscita del Register File.

- modifica della **Unità di Controllo**.



***ALU** calcolo indirizzo di salto*
+
Comparatore per confronto `rs` *ed* `rt`

CPU pipeline con gestione degli hazard



Ottimizzazione Criticità di controllo

2. Branch Prediction:

- Faccio una predizione della necessità di effettuare il salto, tentando di indovinare.
- Se indovino, non dovrò fermare la pipeline:
  ➔ più indovino, più vado veloce

Mi servono strutture hardware e strategie per la
predizione ottima del salto

Branch Prediction Buffer:

- Buffer di memoria in cui, associata all'indirizzo di salto, viene memorizzata l'informazione se debba essere eseguito o meno.

Intel Pentium IV: Branch Prediction Buffer di 4 kByte in CPU

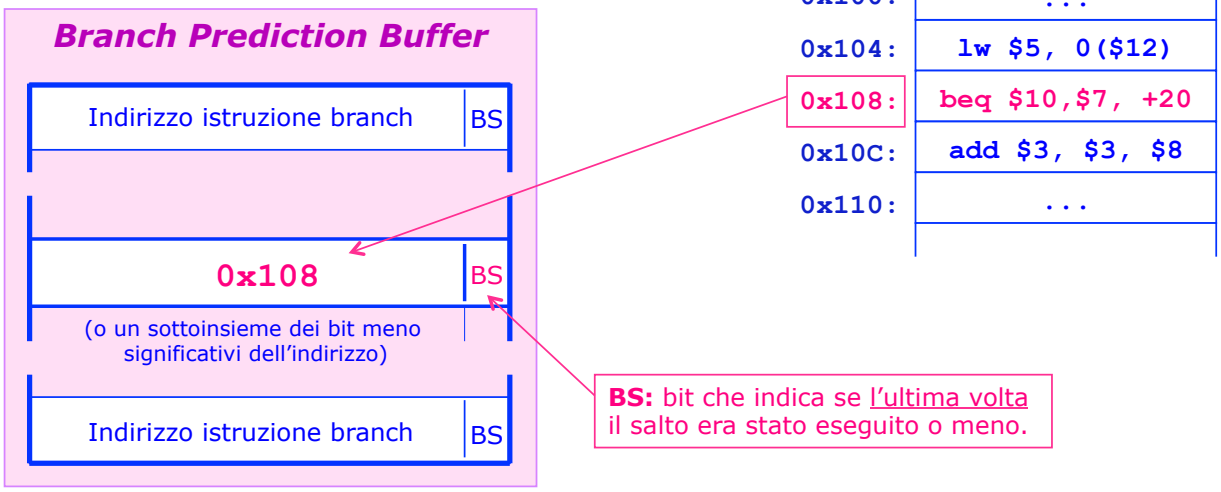
Branch Prediction Buffer

Branch Prediction Buffer – struttura: è una memoria associativa

La CPU deve disporre delle seguenti informazioni:

- Identificazione dell'istruzione di branch → indirizzo
- Previsione → 1 bit (salto/non salto)

➔ **BPB risiede nella CPU**

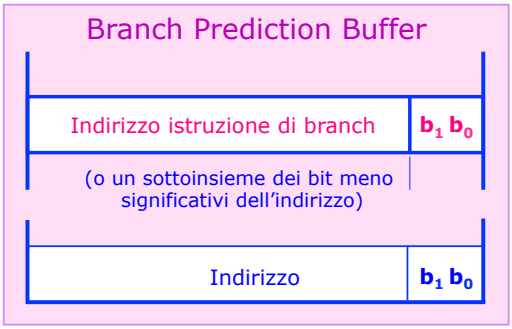


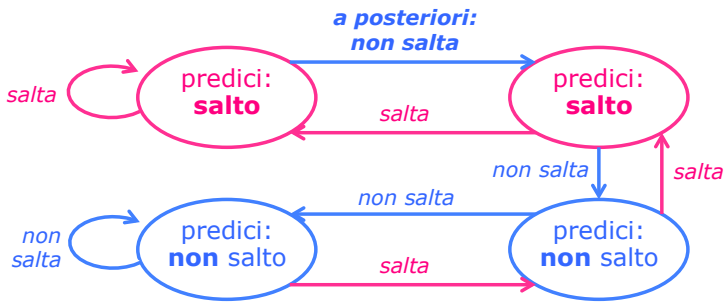
Branch Prediction: 2-bit Prediction

Svantaggio funzionamento BPB a 1 bit: *2 cicli FOR annidati*

- Nel ciclo FOR interno, **BPB** sbaglia 2 volte:
  - alla fine (per uscire) – inevitabile
  - all'inizio (a causa del FOR precedente)

- **Schema di predizione a 2 bit** (*2-bit Prediction Scheme*):
  - non sbaglia all'inizio
  - Costo: 2 bit per la predizione anziché 1 nel BPB





Come scartare le istruzioni

In ogni caso, nelle criticità di salto devo **eliminare istruzioni**:

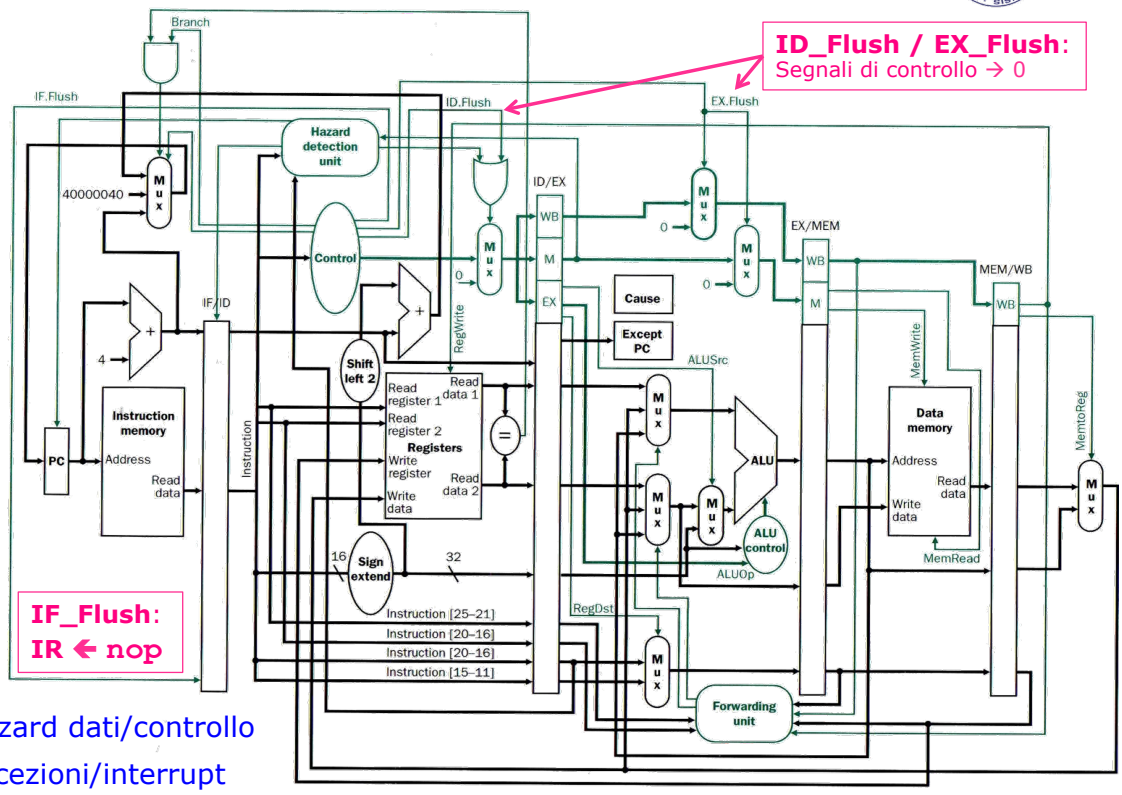
Modifica architettura CPU:
introduzione del **meccanismo di eliminazione istruzioni**

- È diverso dallo stallo di un'istruzione per criticità dati, nel quale l'istruzione viene ritardata, ma non eliminata.

Dipende dallo stadio in cui si trova l'istruzione da scartare.
Potrebbe trovarsi in: **IF**, **ID**, **EXE**:

- ELIMINAZIONE in fase di Fetch (IF):
  sovrascrivo IF/ID con l'istruzione `nop`
  (riesco così a gestire anche criticità generate da `jump`)
- ELIMINAZIONE in fase di Decodifica (ID):
  metto a ZERO i segnali di controllo nel registro di pipeline **ID/EX**
- ELIMINAZIONE in fase di Esecuzione (EX):
  metto a ZERO i segnali di controllo nel registro di pipeline **EX/MEM**

CPU pipeline: struttura finale



CPU completa:

Gestione di hazard dati/controllo

Gestione di eccezioni/interrupt

Tecnologia delle CPU pipeline moderne

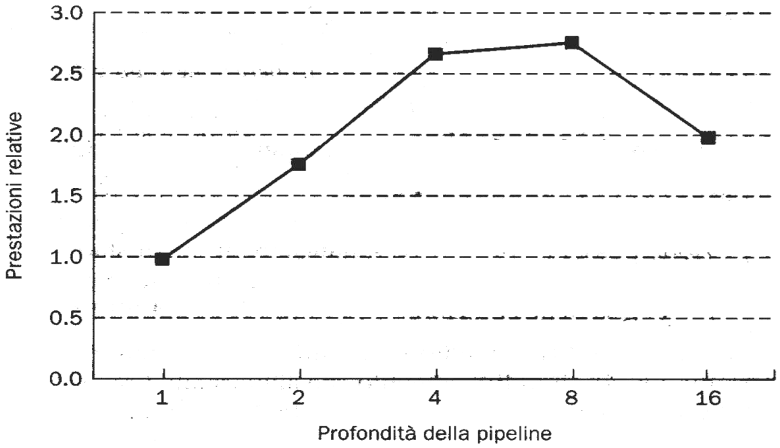
- Super-pipelining
- Architetture superscalari
  - scheduling statico
  - scheduling dinamico

Pipeline più spinte

- **Superpipelining**: pipeline più lunga
  - dato che il fattore di accelerazione delle pipelines è il numero di stadi
- **Architetture superscalari** (statica/dinamica):
  più di una istruzione viene iniziata (scheduled) nello stesso ciclo di clock
  - **Scheduling statico**: parallelismo definito a priori dal programmatore o dal compilatore (IA-64)
  - **Scheduling dinamico**: parallelismo deciso automaticamente dalla CPU. L'hardware cerca di individuare le istruzioni pronte per essere eseguite (MIPS, Pentium 4)
    - è possibile l'esecuzione in ordine diverso da quello dato.
    - è possibile avere un'unica istruzione per più dati: architettura ***SIMD:*** *Single Instruction – Multiple Data*

      Esempio: Intel Pentium IV: tempi medi di esecuzione della ALU sono metà del periodo di clock – fino a 126 istruzioni contemporanee

Superpipeline



Pipelines più lunghe:

teoricamente guadagno in velocità proporzionale al **n. di stadi**

Es: Digital DEC Alpha: 5-7 stadi, Intel/AMD: oltre 20 stadi

Problemi:

- Criticità sui dati: stalli più frequenti.
- Criticità sul controllo: numero maggiore di stadi di cui annullare l'esecuzione.
- Maggior numero di registri → maggiori dimensioni chip → il clock non si riduce con il numero degli stadi

Schedulazione statica: *IA-64*

Scheduling statico: parallelismo tra istruzioni **definito a priori** dall'utente (programmatore e/o compilatore)

Intel IA-64: Instruction-level parallelism (ILP)

EPIC: Explicitly Parallel Instruction Computer
– parallelismo implementato esplicitamente

- Le istruzioni sono raggruppate in "bundles" eseguiti in parallelo
  - 128 registri accedibili a finestre (in parallelo)
  - Istruzione: 128 bit = 5 bit: template field + 3 x 41 bit: 3 istruzioni in parallelo

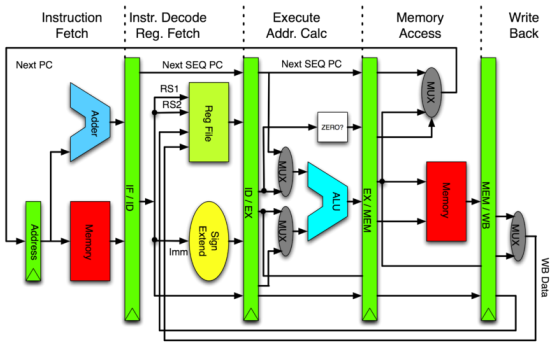
Predication: trasformazione di branch in predicati:

```
If (p) then (statement 1) else (statement 2)
                ▼
   (p)   ➔ statement 1     # statements 1 e 2 possono
   (~p)  ➔ statement 2     # essere parallelizzati!
```

Pipeline Superscalari: **MIPS**

Architetture surperscalari statiche

ottengo più di 1 istruzione per ciclo di clock: **N > 1 istruzione/CC**



Famiglia MIPS:

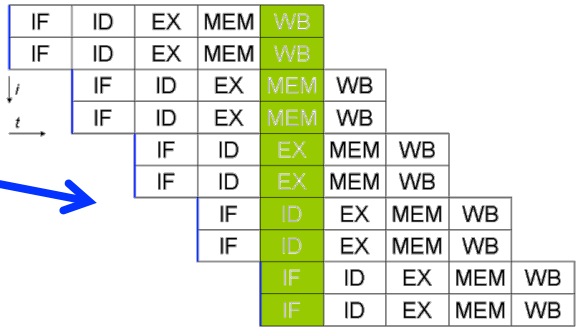
MIPS R2000: pipeline classica

R3000: parziale raddoppio

- **Register File** a **doppia banda**:
  - **4** porte di **lettura**,
  - **2** porte di **scrittura**
- **2 ALU**: general purpose, base+offset

R8000: pipeline **superscalare**

- Replicate le unità funzionali per eseguire più istruzioni in parallelo



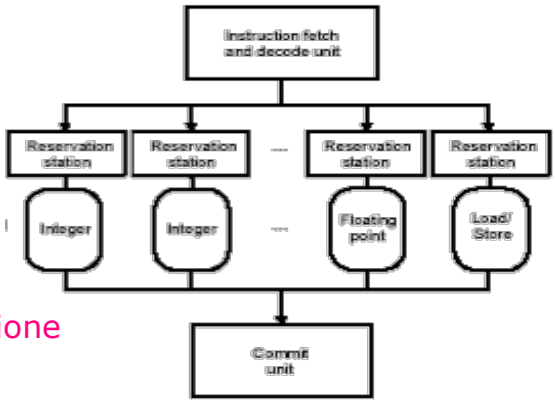
Schedulazione dinamica:

Schedulazione Dinamica:

Fetch: Prelievo ed invio alla coda istruzioni

Decodifica: Conversione in microistruzioni

- Valutazione criticità (dipendenze/branch)



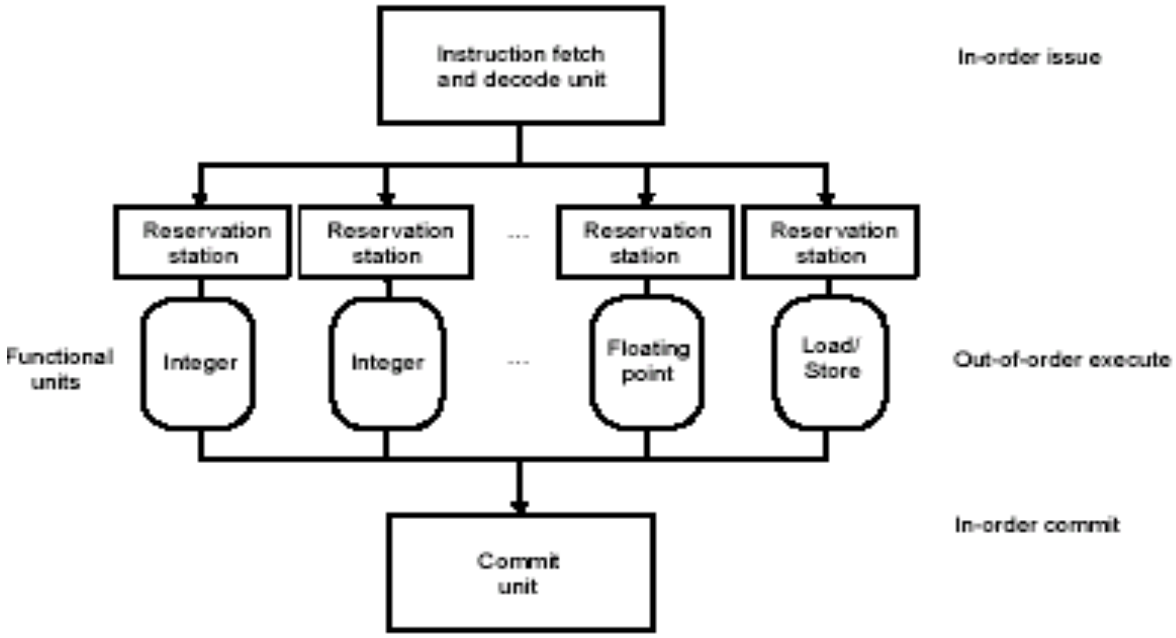
Esecuzione:
Le istruzioni vengono alla stazione di prenotazione corrispondente all'unità funzionale richiesta (smistamento)

- Un riferimento all'istruzione viene inserito nel buffer di riordino.
- Esecuzione vera e propria (calcolo)
- Eventuali criticità portano a far attendere alcune istruzioni

Commit Unit:
Riordino istruzioni in modo che terminino in modo sequenziale

- Può fornire in uscita più istruzioni in un ciclo di clock.
- Commit Unit tiene traccia di tutte le istruzioni pendenti.

Pipeline con schedulazione dinamica



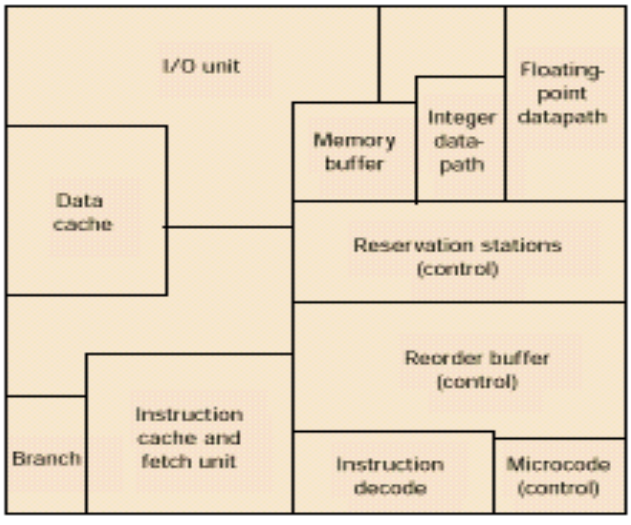
- Esistono cammini paralleli per le diverse fasi dell'istruzione (Exec/Mem)
- Durante lo stallo eseguo istruzioni successive → **supero gli stalli**

Pipeline: processori INTEL

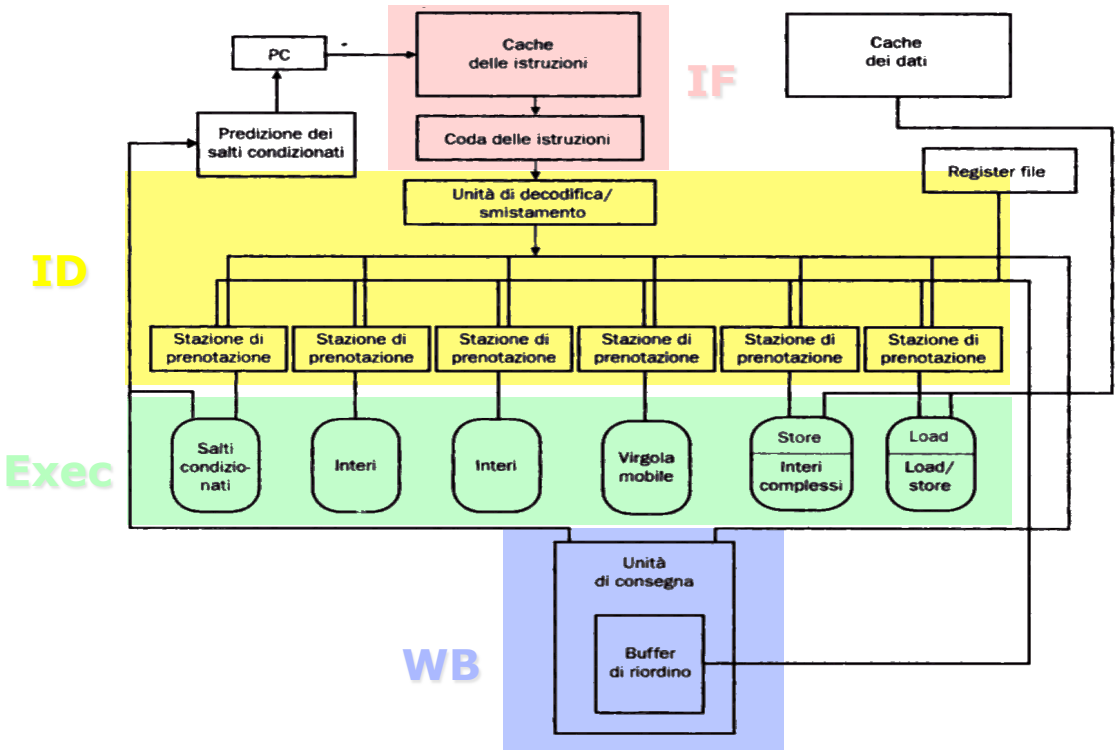
Architetture superscalari di INTEL:

- Intel x86, dal ***Pentium***: esecuzione "speculativa":
  - scheduling dinamico
  - predizione dei salti
- Dal ***Pentium IV***:
  - architettura super-scalare con scheduling dinamico
  - "super-pipeline"

Pentium – aree funzionali sul chip



Scheduling dinamico: PentiumPro



Pipeline PentiumPro: funzionamento

Pipeline **Pentium Pro**

- **Fetch**: Prelievo di 16 byte dalla MI ed invio alla coda istruzioni
- **Decodifica**:
  Conversione in microistruzioni di lunghezza fissa: **72 bit**
  - In questa fase vengono valutate le branch (su un certo numero di istruzioni) attraverso una **Branch Prediction Table** di 512 elementi
  - Recupero operandi ed assegnazione registri replicati (rename buffer)
- **Esecuzione**: Le istruzioni vengono inviate alla stazione di prenotazione corrispondente all'unità funzionale richiesta (smistamento)
  - Un riferimento all'istruzione viene inserito nel buffer di riordino.
  - Esecuzione vera e propria (calcolo)
- **Riconsegna:**
  Riordino istruzioni in modo che terminino in modo sequenziale
  - L'unità di consegna tiene traccia di tutte le istruzioni pendenti all'interno del buffer di riordino.
  - Può fornire in uscita più istruzioni in un ciclo di clock.

Esecuzione dell'istruzione

- Per considerare un'istruzione serve spazio nel buffer di riordino e nella stazione di prenotazione della pipeline richiesta
- La CPU contiene registri duplicati
  - con più istruzioni in esecuzione, il registro destinazione potrebbe essere occupato → scrivo su un registro ausiliario: **rename registers/buffers**
  - in attesa che l'unità di consegna dia il permesso di scrivere i registri effettivi
- L'istruzione viene eseguita quando:
  - nella stazione di prenotazione ci sono **tutti gli operandi**,
  - l'unità funzionale è **libera**.
- L'unità di consegna decide quando l'istruzione è terminata → definitiva
  - Se la predizione di un **salto è scorretta**, vengono scartate le istruzioni sbagliate dalle stazioni di prenotazione e dai buffer di riordino e liberati i registri interni

LA CPU del **Pentium IV**

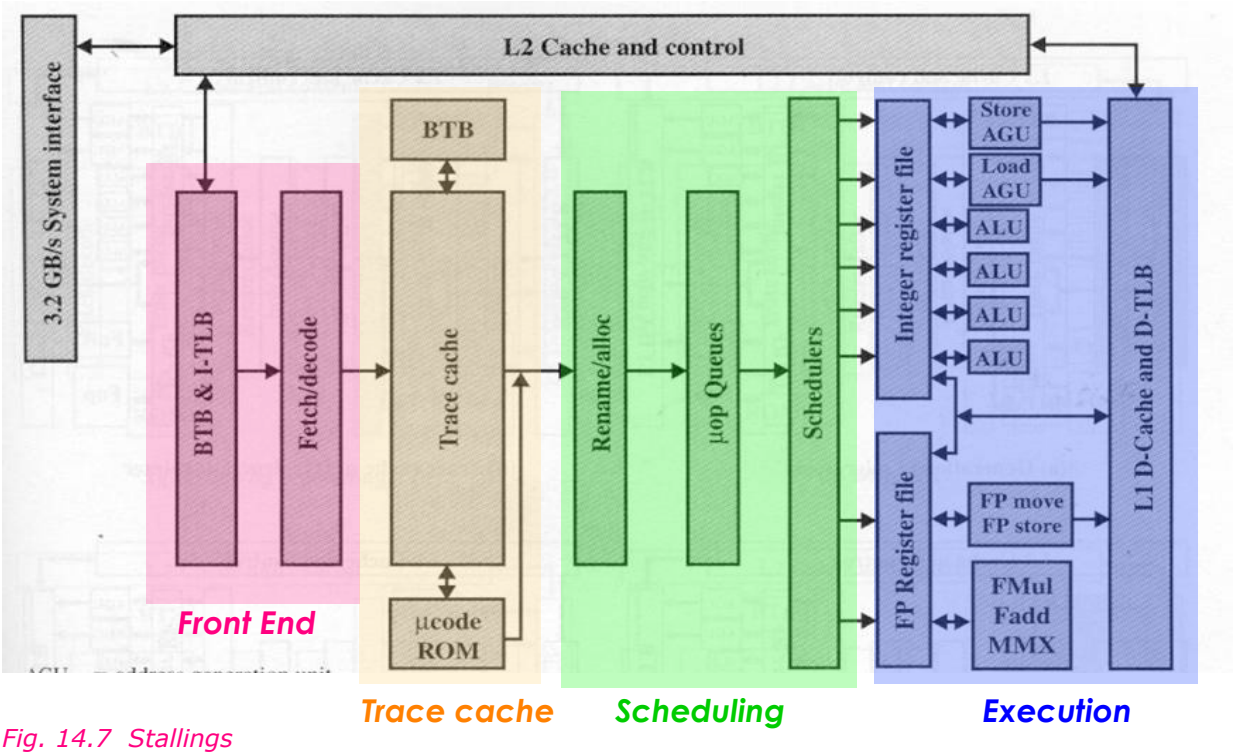


Fig. 14.7 Stallings

Pipeline del **Pentium IV**

Pipeline Pentium IV

- **Fetch** delle istruzioni della Memoria (Cache)
- **Decodifica:** Ciascuna istruzione viene tradotta in una o più microistruzioni di lunghezza fissa (RISC)
- Il processore esegue le micro-istruzioni in una pipeline superscalare a schedulazione dinamica
- Il processore restituisce ai registri il risultato dell'esecuzione delle micro-istruzioni relative alla stessa istruzione
  - rispettando l'ordine di esecuzione delle microistruzioni

Il Pentium 4 trasforma un **ISA CISC** in un **ISA interno RISC** costituito dalle micro-operazioni

Pipeline del Pentium 4: **Front end**

Front End:

- Le **istruzioni** passano dalla cache primaria al buffer **L2** (64 byte)
  - Qui vengono gestiti casi di "missing" ed i trasferimenti da cache
- In parallelo i **dati** vengono caricati da cache al buffer dati **L1**
  - Nel trasferimento da L1 ed L2 agiscono Branch Target Buffer (BTB) e Translation Look-aside Buffer (TLB) che gestiscono le predizioni sulle branch
- Inizia la fase **Fetch** e **Decodifica** che legge il numero di byte pari alla lunghezza dell'istruzione
- Il **decoder** traduce l'istruzione in micro-operazioni (da 1 a 4):
  - **Microistruzione: 118 bit, appartenenti ad un'ISA RISC.**

Pipeline del Pentium 4: **Trace cache**

Trace cache: riorganizzazione delle istruzioni in decodifica, per ottimizzare il flusso di esecuzione

- Funzioni svolte:
  - **Riordino locale del codice,** in modo da minimizzare le criticità
    - la pipeline ha 20 stadi.
  - Invio alla fase di esecuzione (Allocate e Renaming).
  - **Istruzioni complesse** (> 4 microistruzioni) vengono create in questa fase con una macchina a stati finiti
    - Implementata con una ROM + memoria

Pipeline del Pentium IV: **Esecuzione**

- Elemento centrale è lo **Scheduler** (Reorder Buffer)

  È un **buffer circolare** – può contenere fino a **126 micro-istruzioni**
  - Contiene le micro-istruzioni con il loro stato, la presenza o assenza dei dati, per tutta la durata dell'esecuzione.
  - Alloca i registri (traducendo i registri simbolici dell'Assembly in uno dei 128 registri di pipeline).
  - Avvia alla coda di esecuzione l'istruzione (coda per le **`lw/sw`** e coda per le altre istruzioni).
  - Lo scheduling avviene prendendo la prima istruzione che può essere eseguita nella coda. Dagli scheduler si può passare alle ALU in modi diversi.
- La fase di esecuzione scrive i risultati nei registri o nella cache L1
- C'è una parte dedicata ai flag che vengono poi inviati alla **BTB** per predire correttamente i salti.