La CPU pipeline

A. Borghese, F. Pedersini

Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano

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

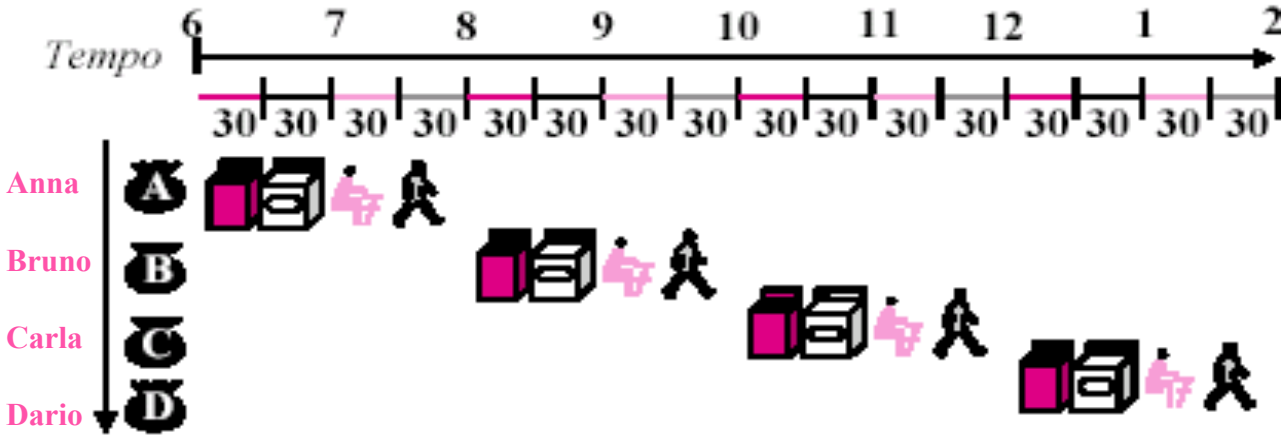
  Ciascuno deve:

  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**



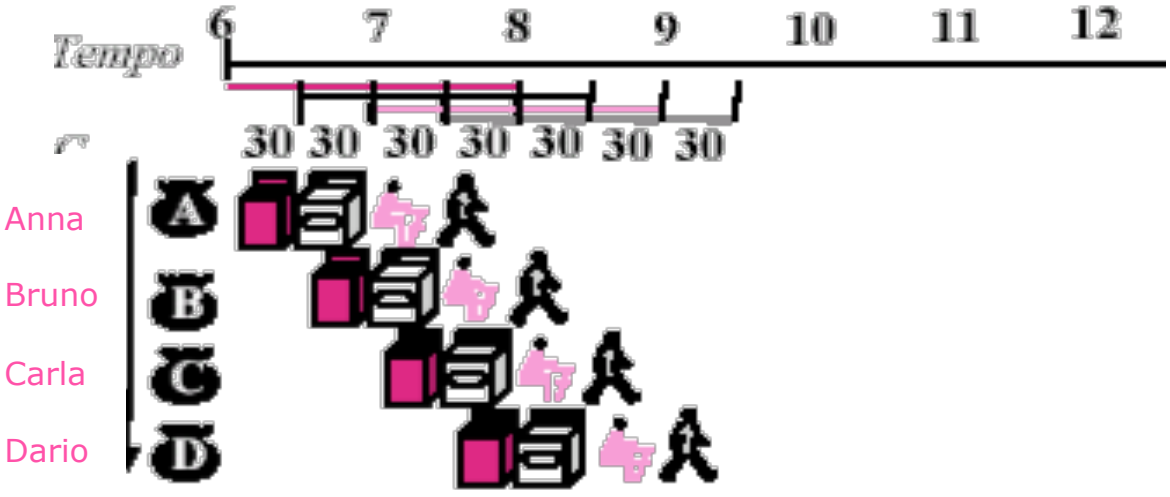
- le singole operazioni vengono svolte una alla volta

  T_totale = T_CicloCompleto * N_CICLI =

  = (T_SingolaFase * N_Fasi) * N_CICLI

- Tempo totale richiesto: **8 ore**

La lavanderia **pipe-line**



- le singole operazioni vengono svolte **in parallelo**
  - **T_totale = T_SingolaFase * (N_Fasi + N_CICLI – 1)**
- Tempo totale richiesto: **3,5 ore**

Osservazioni sulla pipe-line

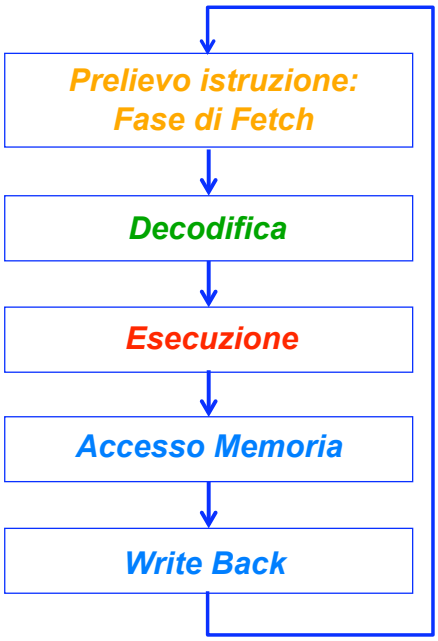
- Il tempo di ciascuna operazione elementare non viene ridotto.
- Gli stadi della pipe-line lavorano in contemporanea perché utilizzano unità funzionali differenti.
- Le unità funzionali lavorano sequenzialmente (in passi successivi) su istruzioni successive

  ➔ Aumenta il ***"Throughput"***

  throughput = quantità di lavoro / tempo

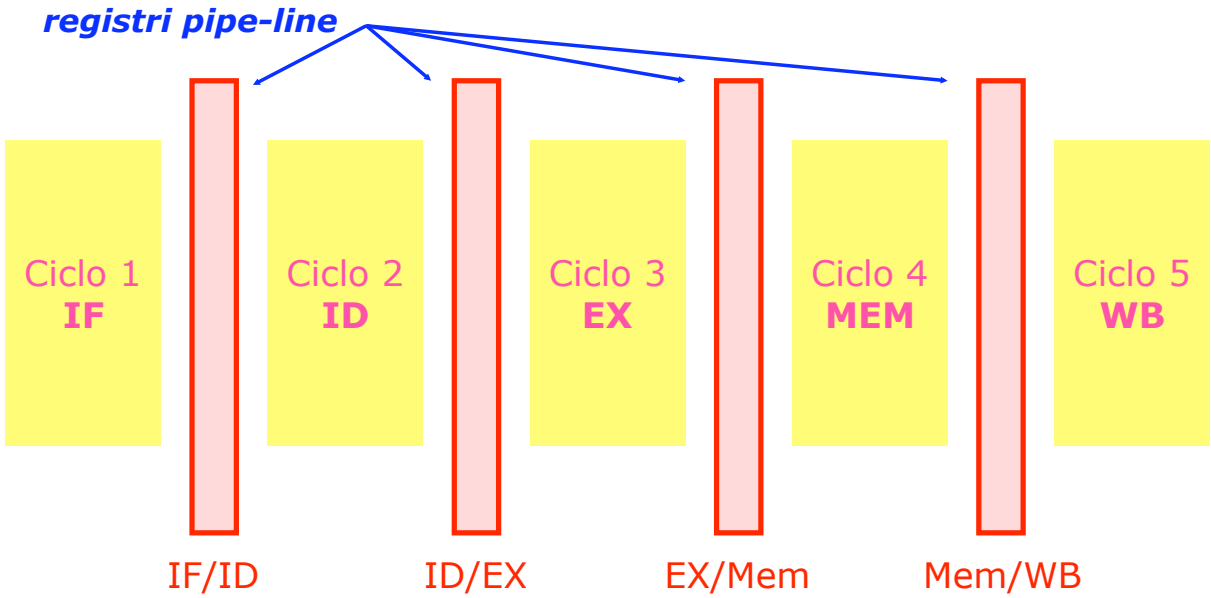
Ciclo di esecuzione: istruzioni MIPS

- MIPS è un'architettura pipeline
  - le istruzioni richiedono 5 passi
- **MIPS:** 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



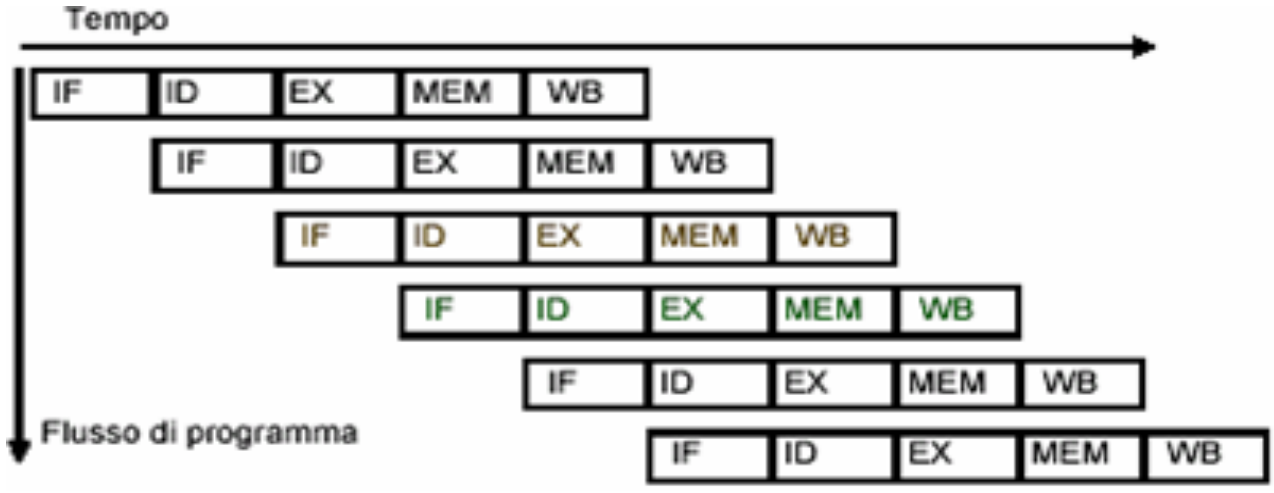
I 5 stadi della pipeline

- Tra cicli contigui sono posti i **registri di pipe-line**
  - Uno stadio inizia il suo lavoro quando il clock va basso e trasferisce in quello stadio l'elaborazione effettuata dallo stadio precedente

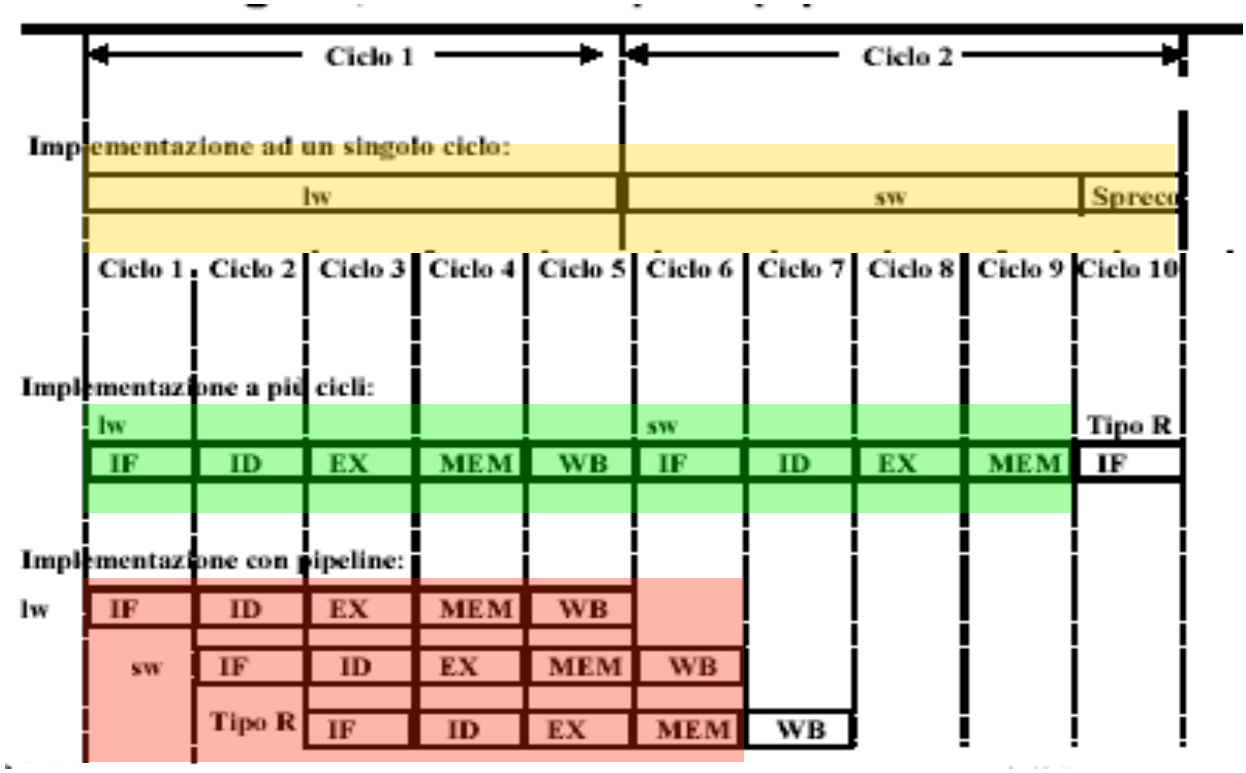


Rappresentazione della pipeline

- Rappresentazione convenzionale:
  - Ascisse (X): tempo
  - Ordinate (Y): Flusso di programma



I vantaggi della pipeline



Miglioramento delle prestazioni

- Il miglioramento massimo è una riduzione del tempo di un fattore pari al numero di stadi della pipe-line
  - Nell'esempio precedente (2 istruzioni `lw`, 2ns ad istruzione), il tempo richiesto con la pipe-line è di **12 ns** contro i **20 ns** senza pipe-line.
  - Il miglioramento teorico prevedrebbe **4ns**
- Il *Throughput* migliora comunque!
  - Miglioramento relativo al lavoro globale (con pipe-line senza stalli)

Utilizzo unità funzionali della CPU

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

Istruzioni → passo ↓	Istruzioni tipo R	Istruzioni di accesso a memoria	Salto condizionato	Salto non condizionato
1. Fetch	IR = Memory[PC] ; PC = PC + 4			
2. Decodifica	A = Reg[IR[25-21]] ; B = Reg[IR[20-16]] ALUout = PC + (sign_ext(IR[15-0]) << 2)			
3. 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
4. Mem: acc. R: WB	Reg([IR[15-11]] = ALUOut	**lw**: MDR=Mem[ALUOut] **sw**: Mem[ALUOut] = B		
5. Mem: WB		**lw**: Reg[IR[20-16]] = MDR		

- **Criticità strutturale:** se istruzioni diverse (in fasi diverse) necessitano contemporaneamente di una stessa risorsa, la risorsa va duplicata, oppure... si aspetta!

Utilizzo unità funzionali della CPU

Φ	Passo esecuzione	ALU	Memoria	Register File
1	Fase fetch	Yes (PC+4)	Yes	No
2	Decodifica	Yes (salto)	No	Yes
3	Exec I – beq	Yes (test)	No	No
	Exec I – jump	No	No	No
	Exec I – tipo R	Yes (salto)	No	No
	Exec I – sw	Yes (indirizzo)	No	No
	Exec I – lw	Yes (indirizzo)	No	No
4	Exec II – tipo R	Yes (op)	No	Yes
	Exec II – sw	No	Yes	No
	Exec II – lw	No	Yes	No
5	Exec III – lw	No	No	Yes

Criticità strutturali: esec. istruzione `lw`

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

`lw $t0, 8($t2)`	ALU (PC) Mem	RF	ALU	Mem	RF		
`lw $t1, 12($t2)`		ALU (PC) Mem	RF	ALU	Mem	RF	
`lw $t1, 16($t2)`			ALU (PC) Mem	RF	ALU	Mem	RF
`beq $1, $0, +16`				ALU-PC Mem	ALU (addr.)	...	...

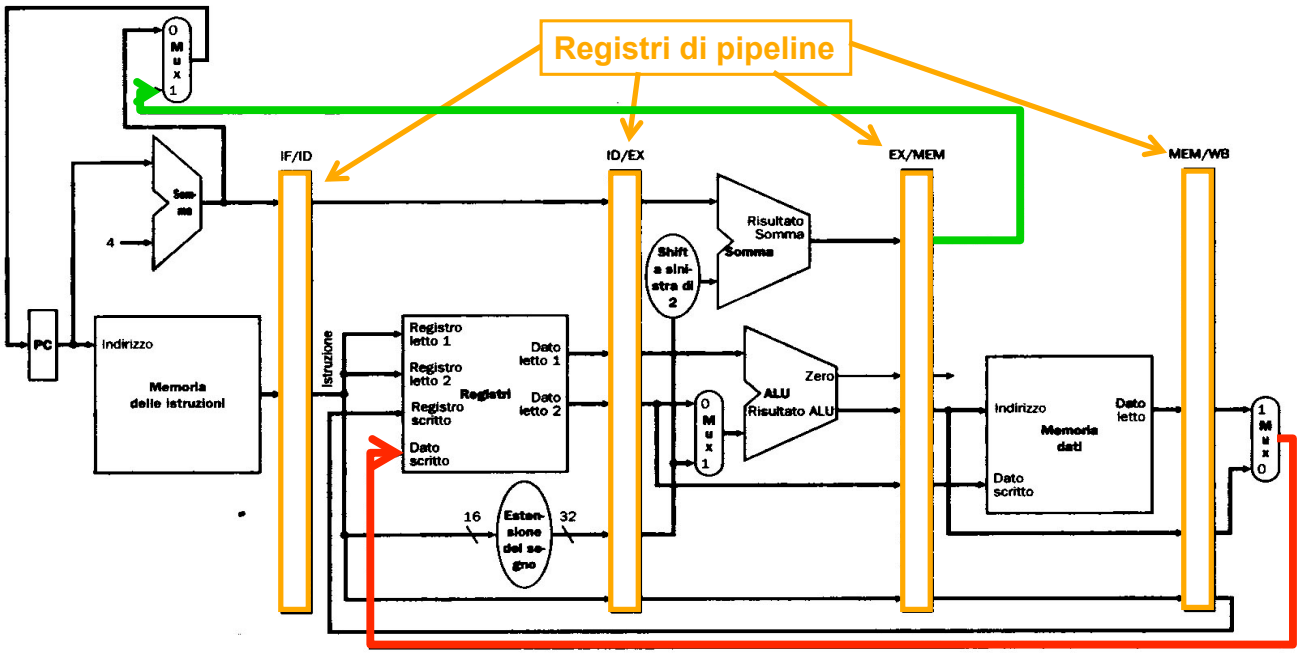
Presenza di **criticità strutturali: ALU, MEMORIA**

- Soluzione: **replicazione** delle risorse: **3 ALU, 2 MEMORIE**

  ALU / ALU (PC) / ALU (address) **Mem. DATI / Mem. ISTRUZIONI**

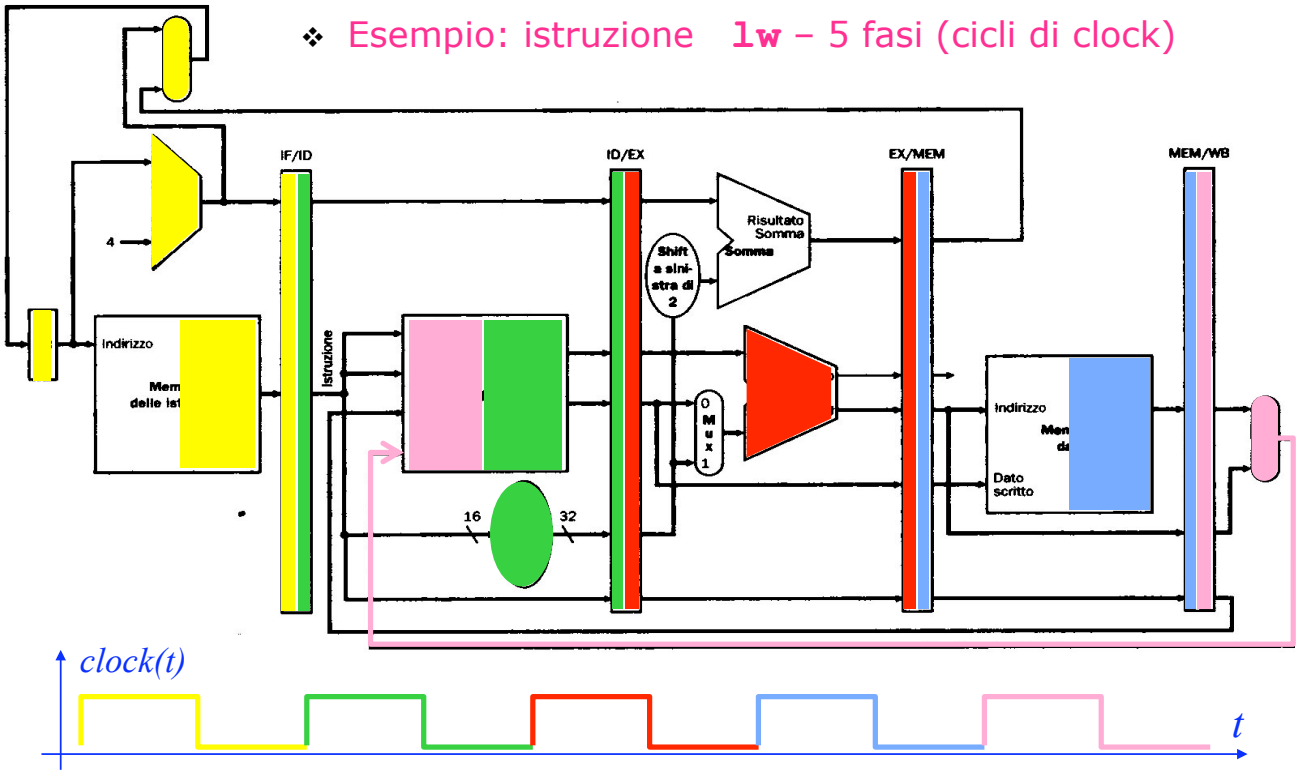
Struttura CPU pipeline

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



Funzionamento CPU pipeline

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



Segnali di controllo – CPU pipeline

Segnale	Effetto quando è negato (0)	Effetto quando è affermato (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
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

- Scrittura di PC e dei registri di pipeline avviene ad ogni fronte di clock

CPU pipeline: progetto unità di controllo

Unità di controllo – CPU pipeline:

Ipotesi:

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

Soluzione:

- una volta generati i segnali di controllo di un'istruzione...
  - nella fase di decodifica
- ...li trasporto attraverso la pipeline insieme all'istruzione stessa
  - nelle fasi successive (EX, MEM, WB)

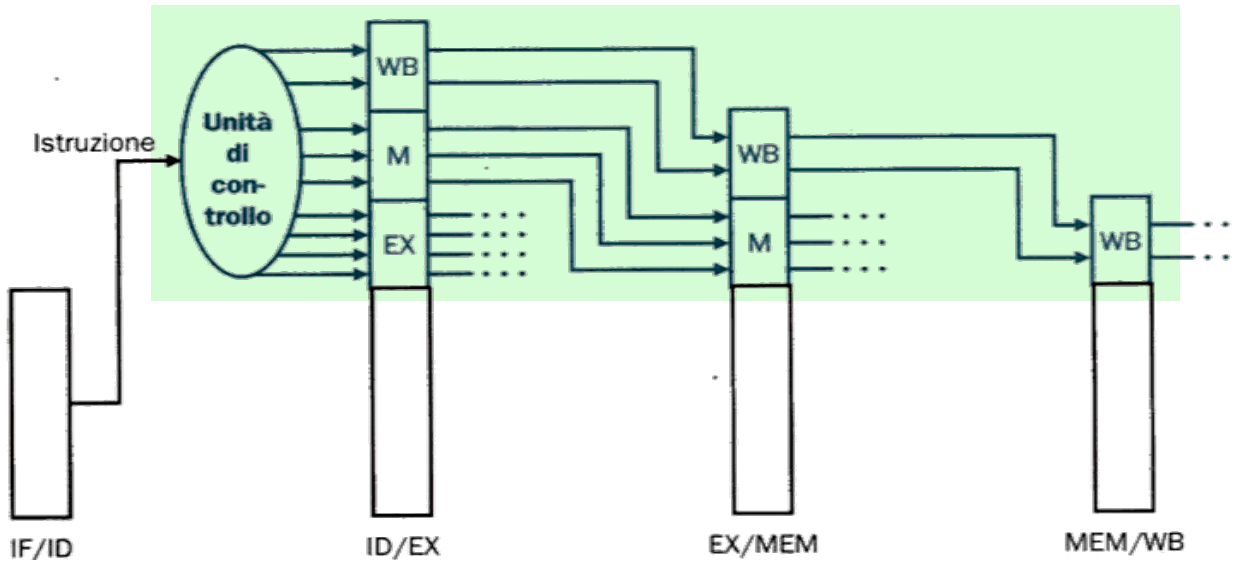
Osservazioni

- I segnali di controllo particolari (legati alle diverse istruzioni) si possono così raggruppare:
  - Nelle fasi di fetch e decodifica non esistono segnali di controllo
  - Il contenuto di **`rt`** ed il numero di **`rd`** vengono portati attraverso i vari stadi

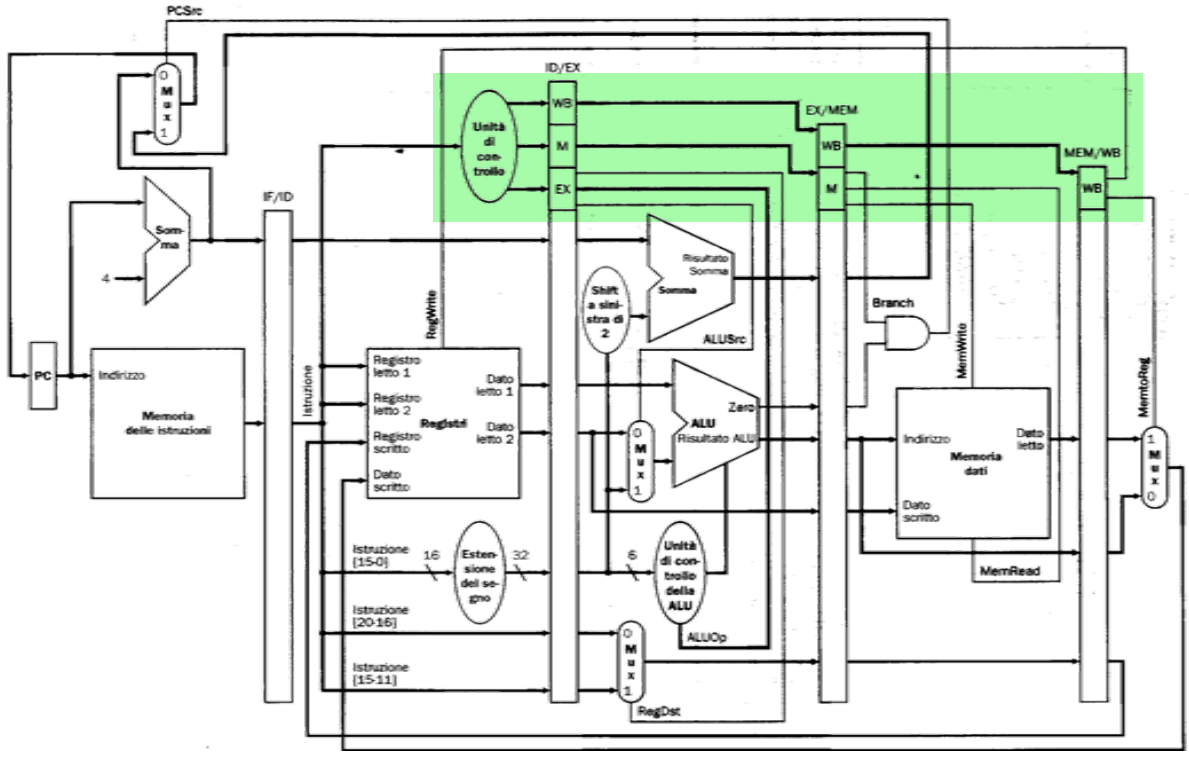
Fase →	Exec				Memory			WB	
Segnali → ↓ Istruz.	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem-Read	Mem-Write	Reg-Write	MemTo-Reg
`Tipo R`	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	x	0	0	1	0	0	1	0	x
`beq`	x	0	1	0	1	0	0	0	x

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



Considerazioni

- La tecnica di progettazione mediante **pipeline** sfrutta il **parallelismo** tra **fasi diverse** di **istruzioni consecutive**.
- Non aumenta la velocità di esecuzione della singola istruzione, ma del lavoro nel suo complesso (**throughput**)
  - Aumento di prestazioni massimo = n. di stadi di pipeline
- L'impossibilità di iniziare ad eseguire una fase di istruzione determina uno stallo della pipeline
- Criticità (hazards) in una pipeline sono suddivisibili in:
  - strutturali,
  - di controllo,
  - sui dati.
- La soluzione a tutte le criticità potrebbe essere … aspettare!

Criticità (hazards) – Sintesi

- **Criticità strutturali**
  - **Necessità della stessa unità funzionale più 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.
- **Criticità di Dato**
  - Dovrei eseguire un'istruzione in cui uno dei dati è il risultato dell'esecuzione di un'istruzione precedente.
  - `lw  $t0, 16($s1)`
  - `add $t1, $t0, $s0`

Esempio: criticità di dati – istruzioni A/L tipo R

`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

- Il dato in **$2** diviene disponibile nel RF nella fase **WB** della **sub**.
- Non è ancora pronto durante 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

Soluzione (software): riorganizzazione del codice

Soluzioni:

- IDEA: 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**, in modo che la fase **ID** dell'istruzione:
  `and $12, $2, $5`
  vada a coincidere con la fase di **WB** della:
  `sub $2, $1, $3`

- Situazione troppo frequente perché la soluzione sia accettabile.

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

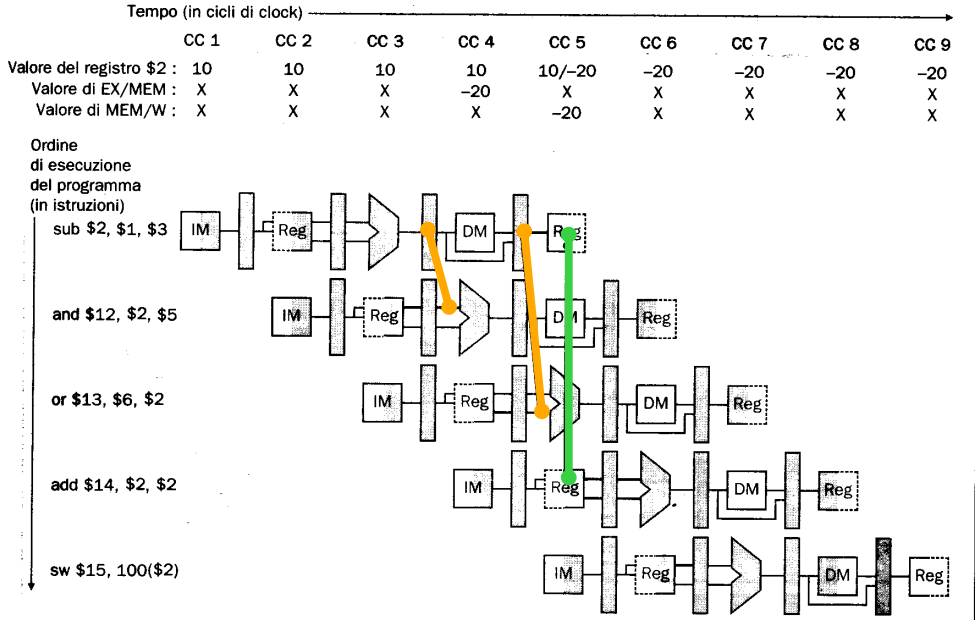
- Prendo il risultato della sottrazione all'inizio dello stadio **MEM**: `$1-$3` è già contenuto nel registro EX/MEM e si trova all'uscita nella fase bassa del clock.
- Sostituisco il primo operando della **`and`** successiva (**`$2`**) con il risultato della sottrazione (contenuto nel registro EX/MEM)
- Non attendo 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

- Prendo il risultato della sottrazione (**`$1-$3`**) all'inizio dello stadio **WB**, presente all'uscita del registro **MEM/WB** nella fase bassa del clock.
- Sovrascrivo il primo operando della **`and`** successiva (**`$2`**) con il risultato della sottrazione (presente all'uscita del registro MEM/WB), senza attendere la fase di **WB**.

Hazard dati: **Forwarding**



Forwarding: si invia il **risultato dell'istruzione precedente** ad un **passo intermedio dell'istruzione attuale** (in pipe).

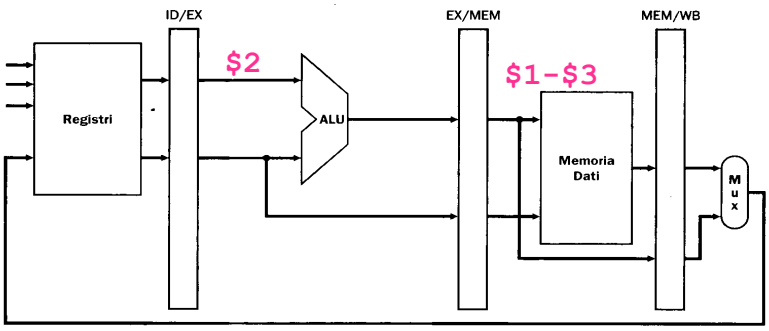
- Il contenuto dei registri **EX/MEM** e **MEM/WB** anticipa il contenuto del Register File.

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

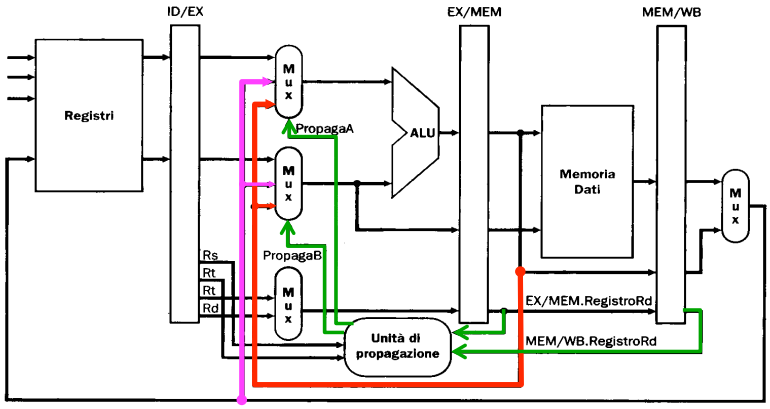
- Nel normale funzionamento, il registro **ID/EX** contiene quanto letto dal Register File.
- Quando abbiamo forwarding, quello che viene letto dal registro **ID/EX** nella fase di esecuzione deve essere sovrascritto da quanto letto dal registro **EX/MEM** o **MEM/WB**.
- Nel registro **EX/MEM** è contenuto il risultato dell'operazione eseguita all'istante precedente.
- Nel registro **MEM/WB** è contenuto il risultato dell'operazione eseguita 2 istanti precedenti.

Hazard nei dati, **progetto soluzione HW**

- Datapath:
  - collegamenti da **ALU:out** ad ALU:in
  - e da MEM:out a ALU:in
- Controllore:
  - Unità di propagazione



a. Senza propagazione



Soluzione architetturale per **AND**

Hazard su AND:

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

```
t-1:   sub $2, $1, $3
t:     and $12, $2, $5
```

IF (RD_{t-1} == RS_t)
then "collega": RD_{t-1} → RS_t
IF (RD_{t-1} == RT_t)
then "collega": RD_{t-1} → RT_t

- RD_{t-1} (`sub $2, $1, $3`) è il contenuto del registro destinazione dell'istruzione precedente la `and` contenuto nel registro EX/MEM

Soluzione architetturale per **OR**

Hazard su OR:

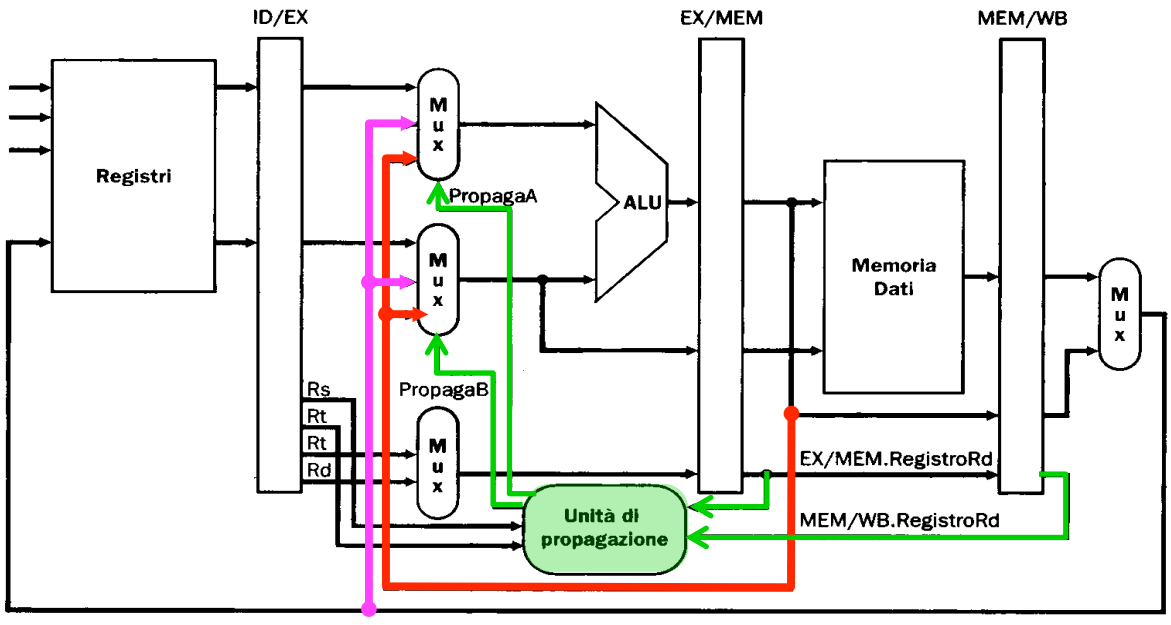
- Occorre implementare la seguente funzione logica nella fase ID dell'esecuzione dell'istruzione:

```
t-2:   sub $2, $1, $3
t-1:   ...
t:     or $13, $6, $2
```

> if (RD_{t-2} = RS_t)
> then "collega" RD_{t-2} con RS_t
> if (RD_{t-2} = RT_t)
> then "collega" RD_{t-2} con RT_t

- RD_{t+2} = RD_{sub} = $2
  è il contenuto del registro destinazione di
  2 istruzioni precedenti la `or: (sub $2, $1, $3)`
  che è copiato nella fase MEM.

CPU con forwarding: schema circuitale



Controllo MUX ingresso alla ALU

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

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

Data Hazard: soluzioni

Soluzioni DATA HAZARD:

SW: Buona scrittura del codice

- il programmatore deve conoscere la macchina per scrivere un buon codice!

- Compilatore efficiente
  - che riordini il codice

HW: FORWARDING: Architettura che renda disponibile i dati appena pronti alla fase di esecuzione.

- Inserire una **nop**
- Accettare uno stallo
  - non sempre si può evitare…

Sommario

- Criticità nei dati – istruzione `lw`
- Criticità nei branch – Branch hazard

Hazard sui dati: `lw`

- 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

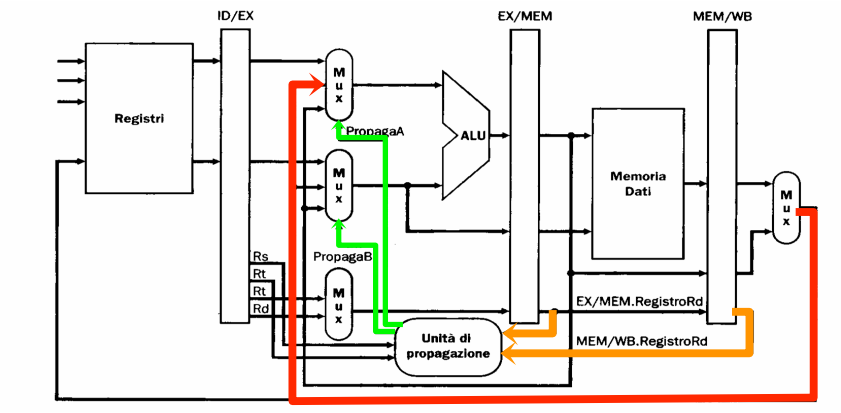
`lw $s2, 40($s3)`	IF	ID	EX $3+40	MEM	WB s → $t2				
`and $t2,$s2,$s5`		IF	ID	EX $2 and $5	MEM	WB s→$t2			
`or $t3,$s6,$s2`			IF	ID	EX $6 or $2	MEM	WB s→$t3		
`add $t4,$s2,$s2`				IF	ID	EX $2 +$2	MEM	WB s → $t4	
`sw $t5, 100($s2)`					IF	ID	EX $2+100	MEM $t5→Mem	WB

Hazard sui dati: `lw` – forwarding

`lw $s2, 40($s3)`	IF	ID	EX $3+40	MEM	WB s→ $2			
`and $t2, $s2, $s5`		IF	ID	EX $2 and $5	MEM	WB s→$t2		
`or $t3, $s6, $s2`			IF	ID	EX $6 or $2	MEM	WB s→$t3	
`add $t4, $s2, $s2`				IF	ID	EX $2 +$2	MEM	WB s → $t4

- Il dato corretto per `$s2` è pronto nella `lw` solamente alla fine della **MEM**
  - utilizzabile solamente dall'inizio della fase di **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

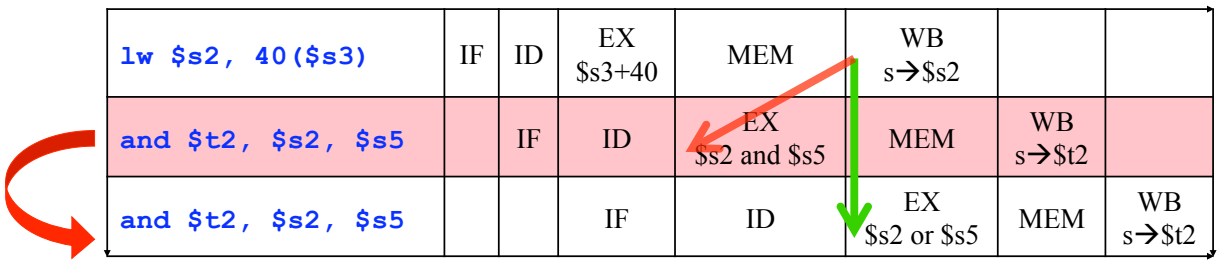


Risolvo la criticità per la `or` mediante propagazione (forwarding)

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

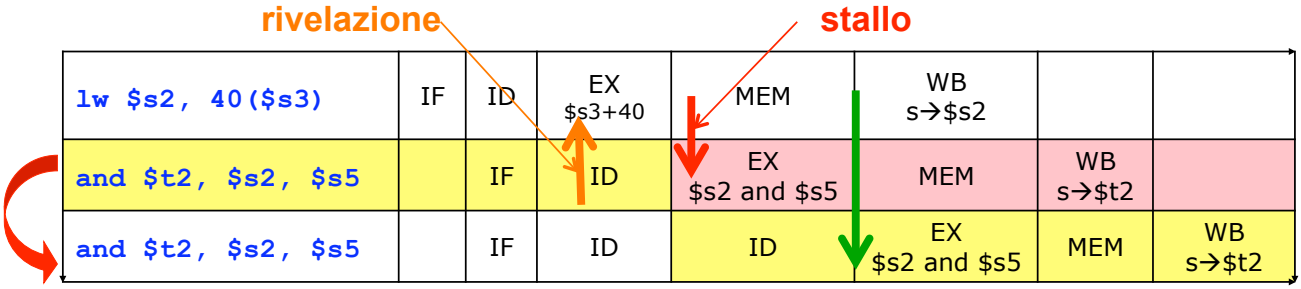
```
IF (RD_{t-2} == RS_t)
        then "collega":        RD_{t-2} con RS_t
IF (RD_{t-2} == RT_t)
        then "collega":        RD_{t-2} con RT_t
```

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`
  - Istruzione `lw` (t–1) in stadio di esecuzione (EX)

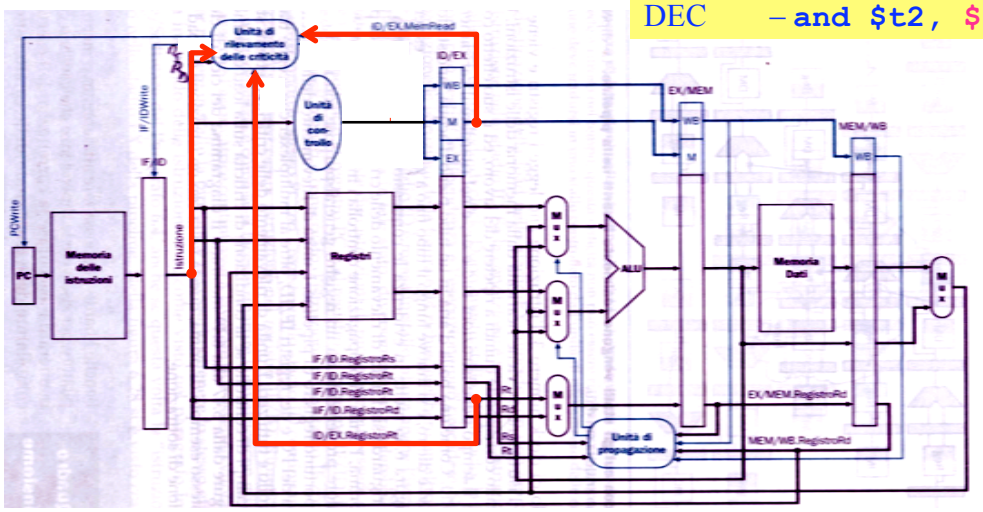
Rivelazione della criticità su `lw`

IF [(ID/EX.MemRead)] → `lw` in fase di EX
AND
{[(IF/ID.RegistroRT) == ID/EX.RegistroRD] OR
[(IF/ID.RegistroRS) == ID/EX.RegistroRD]}

*THEN "Metti in **stallo** la pipeline"*

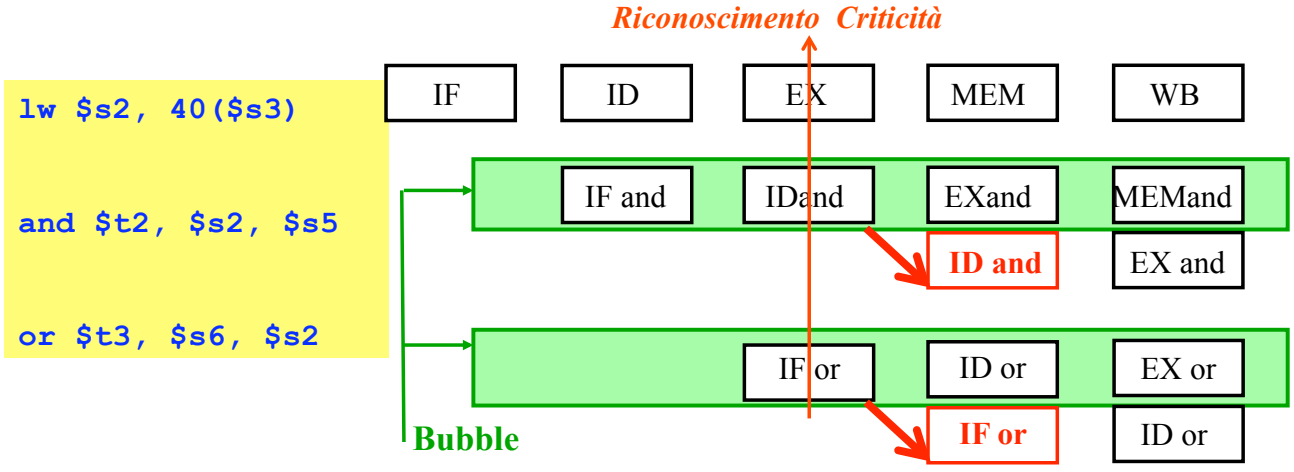
`rd, rt`

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

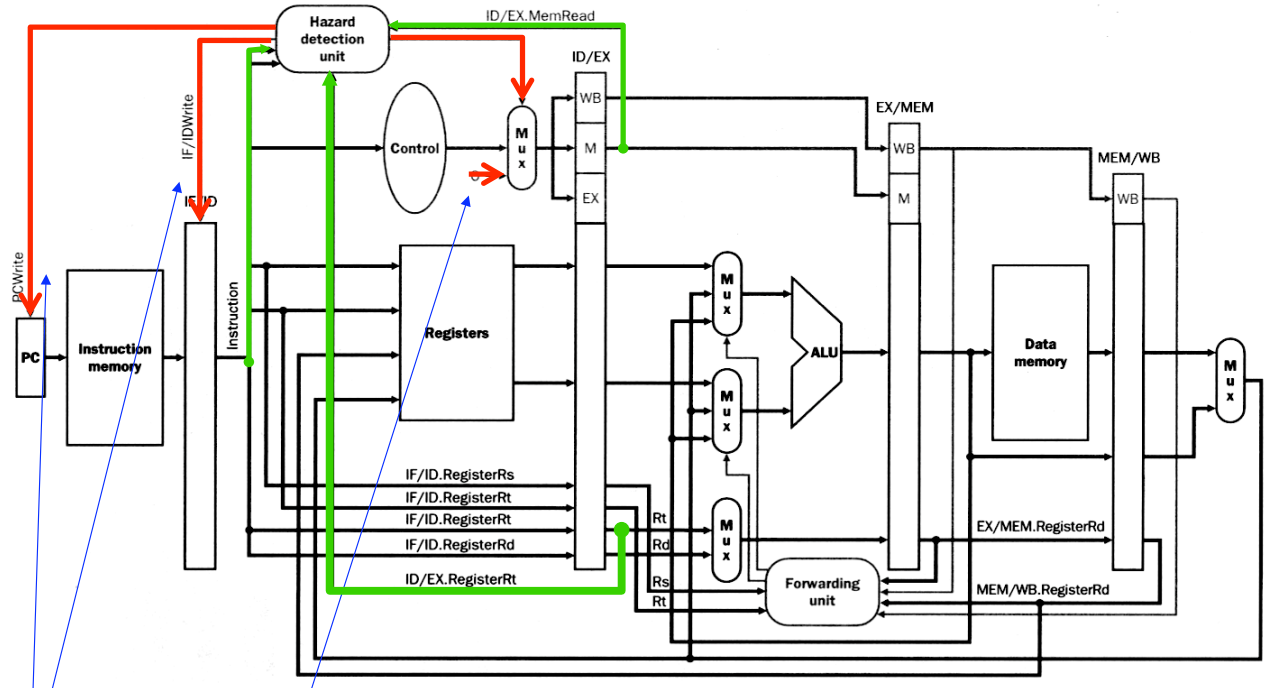


Stallo della pipeline

- **Implementazione dello STALLO** (generazione di una "**BUBBLE**")
  1. **Annullare i segnali di controllo**
     generati nella fase ID per l'esecuzione dell'istruzione successiva alla `lw`
     - Segnali di controllo inattivi ➔ non succede niente
  2. **Ripetere la lettura e la decodifica delle 2 istruzioni successive**
     - Si ripetono le fasi di **fetch(t–1)** e **decodifica(t–2)**



Gestione dello stallo



- **Annullamento** dei segnali di controllo associati → formaz. BUBBLE
- **Disabilitazione** della scrittura del **PC** e del registro **IF/ID**

Sommario

- Criticità nei dati – istruzione `lw`
- Criticità di salto: Branch hazard

Criticità di salto (control hazard)

- Criticità di salto:
  - L'indirizzo di salto eventuale è pronto al termine della fase di EX
  - Il Program Counter può venire aggiornato 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

Soluzioni alla criticità nel controllo

- Soluzioni possibili...
- ...per aggirare la criticità,
  - Riordinamento del codice – **Delayed branch**
- ...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: 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
  - contiamo sul compilatore/assemblatore per mettere dopo l'istruzione di salto una istruzione che andrebbe comunque eseguita indipendente-mente dal salto

Delayed branch

Delayed branch: esempio

```
      add $s4, $t0, $t1
      beq $s5, $s6, salto
      add $s0, $s0, $s1
salto:
      add $t5, $t4, $t3
      add $t6, $t7, $t7
```



```
      add $s4, $t0, $t1
      beq $s5, $s6, salto
      add $t5, $t4, $t3
      add $s0, $s0, $s1
salto:
      add $t6, $t7, $t7
```

- L'istruzione `add $t5, $t4, $t3` deve essere comunque eseguita.
- Il salto (se richiesto) avviene dopo.

Gestione con modifiche strutturali della CPU

- Anticipazione dei salti:
  - diminuire il numero di passi necessari per valutare la branch
  - da 3 (fase MEM) a 1 (fase DEC)
- Guessing (branch prediction buffer):
  - 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

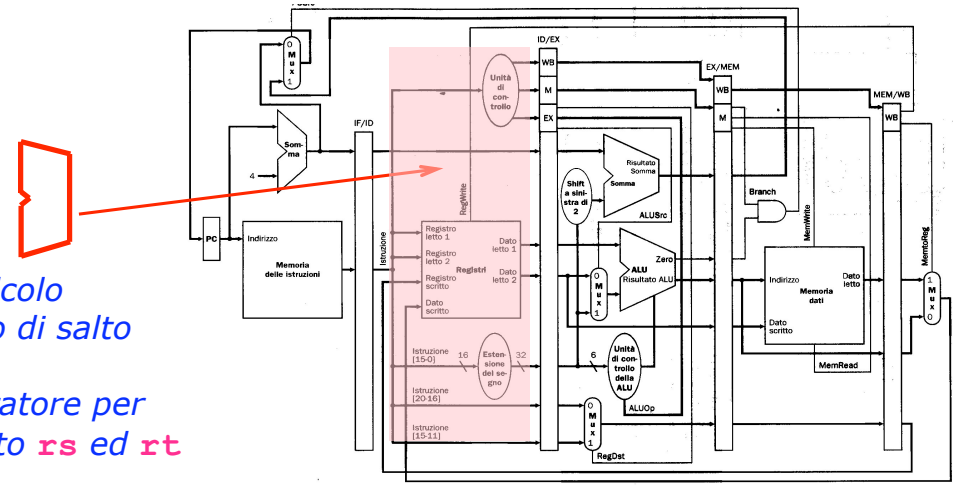
  - Modifiche architetturali per scartare istruzioni già in corso

Modifica della CPU

- Anticipazione del salto:
  - Identificare l'hazard durante la fase **ID** di esecuzione della branch.
  - Necessario scartare **una sola** istruzione.

`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 $t4,$s8,$s8`				IF	ID	EX $s8+$s8	MEM	WB ris.➔ $t4
`add $t0,$t1,$t2`					IF	ID	EX	MEM

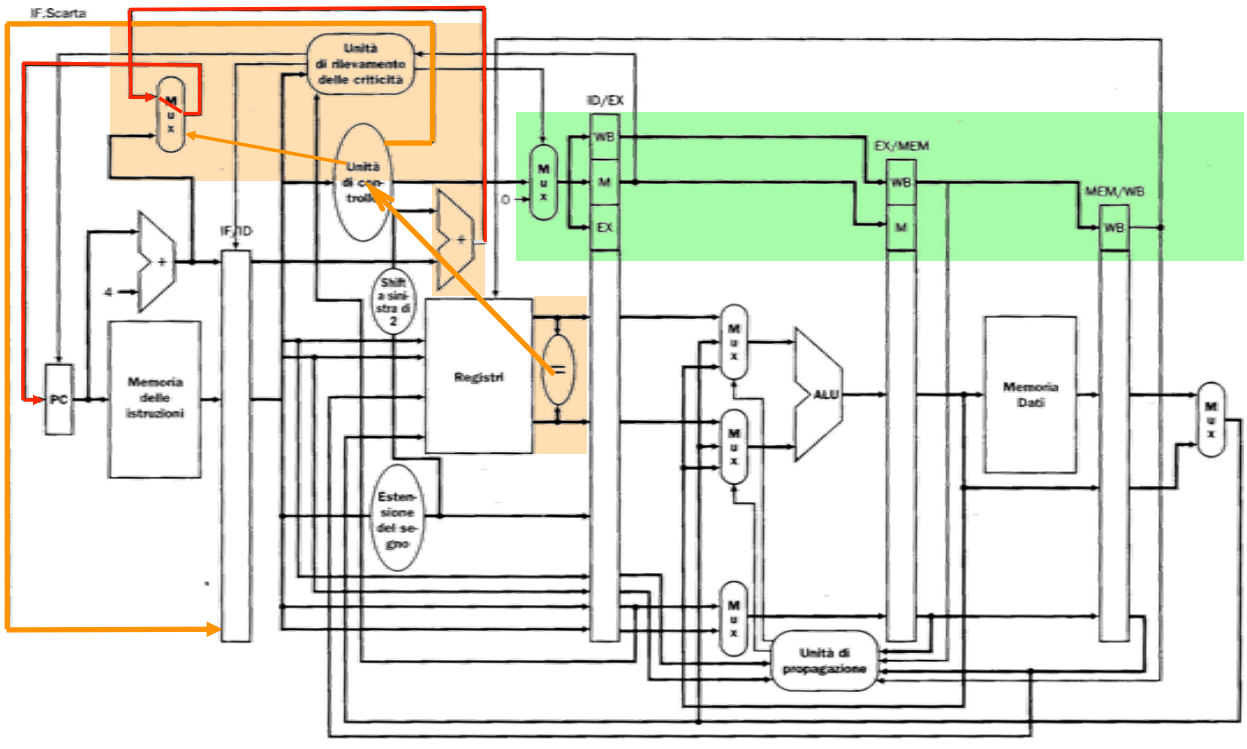
Modifica **HW**: **Anticipazione dei salti**



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

- Anticipazione della valutazione della branch:
  - Modifica della CPU nella gestione dei salti: anticipazione del calcolo dell'indirizzo di salto.
  - HW addizionale:
    - un **comparatore** all'uscita del **Register File**.
    - Modifica dell'**Unità di Controllo**.

CPU pipeline con gestione degli hazard



Ottimizzazione Criticità di controllo

Branch prediction:

- *La CPU predice (tentando di indovinare) la necessità di effettuare il salto*
- *Se indovina, non dovrà fermare la pipeline*

Branch Prediction Buffer:

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

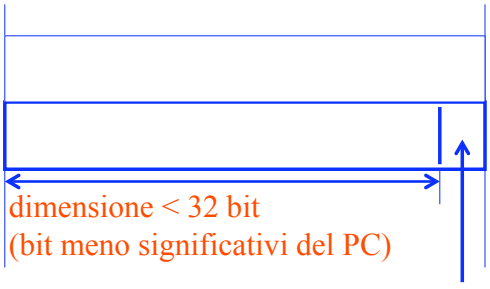
  Intel Pentium IV: Branch Prediction Buffer con capacità di 4 kB

- Algoritmi e strutture per la predizione ottima del salto
  - *Bastano strategie di predizione molto semplici*
    es: ripetizione dell'ultima scelta

Branch Prediction buffer

- Esempio: previsione relativa ad una **beq**
  (con gli stessi bit meno significativi del PC)
- Suppongo di non dovere saltare → procedo in sequenza

  PC* = PC+4: **add**
- Se la previsione è sbagliata, devo:
  - annullare la **add**
  - saltare a **SALTA**.



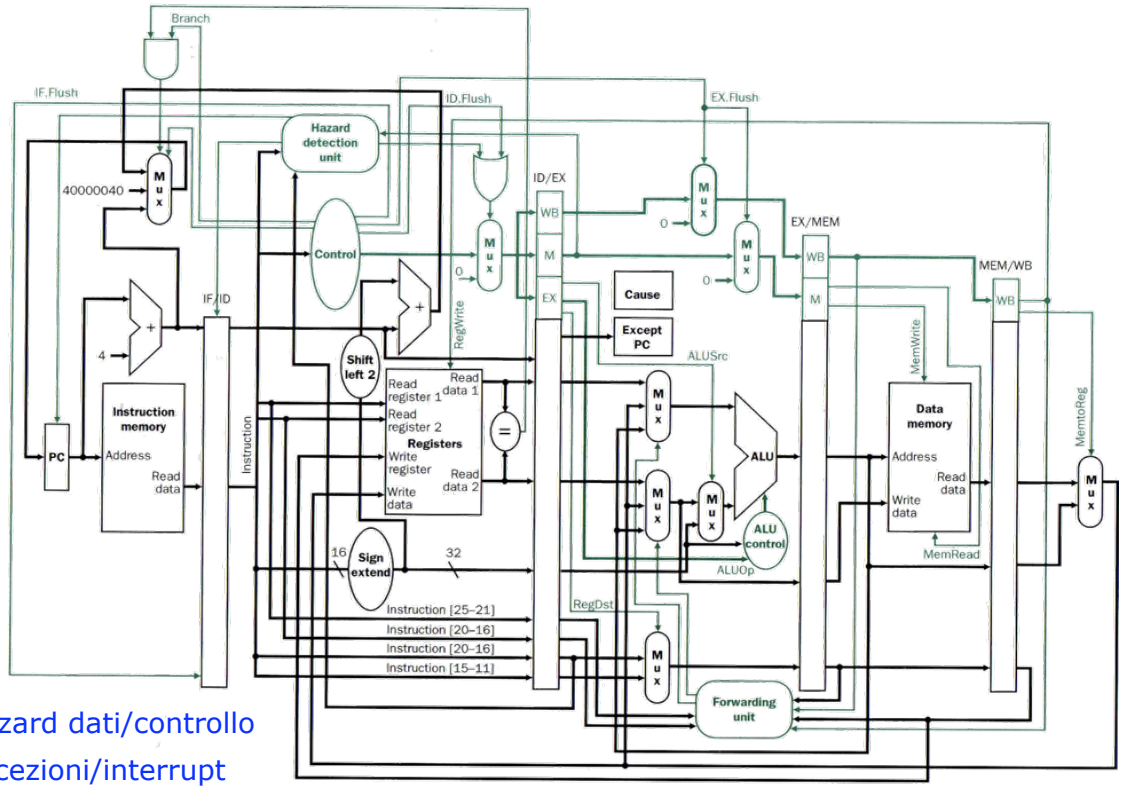
Bit che indica se l'ultima volta il salto era stato eseguito o meno.

```
        beq $t0, $t1, SALTA
        add $s0, $s1, $s2
        sub $s3, $s4, $s5
        or   $s6 $s7, $s8
SALTA:  and $t2, $t3, $t4
```

Come scartare le istruzioni

- Modifica architettura CPU: introduzione del meccanismo di eliminazione istruzioni:
  - Dipende dallo stadio in cui si trova l'istruzione da scartare:
- In fase di fetch:
  - sovrascrivo IF/ID con l'istruzione **nop**
- In fase esecuzione in un altro stadio:
  - metto a ZERO i segnali di controllo nel registro di pipeline di quello stadio.

CPU pipeline: struttura finale



CPU completa:

Gestione di hazard dati/controllo

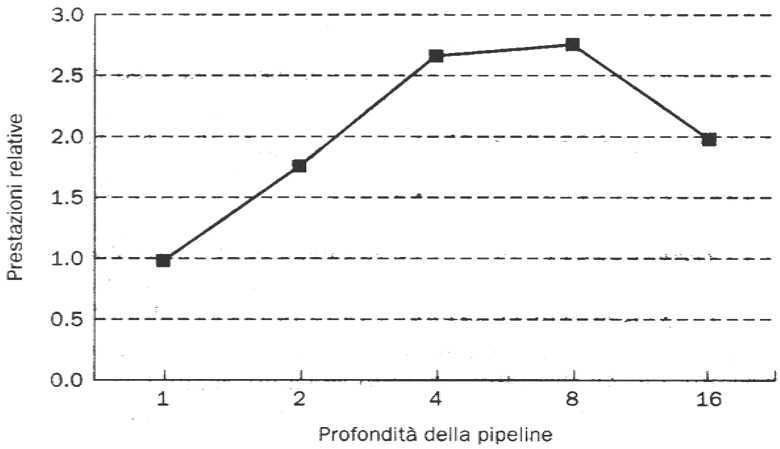
Gestione di eccezioni/interrupt

Le CPU moderne

- Le CPU moderne adottano strutture pipeline più sofisticate:
- **Superpipelining**: pipelines più lunghe (più stadi)
- Architettura **Superscalare**:
  più di una istruzione viene iniziata nello stesso ciclo di pipeline
  - **Scheduling statico**:
    parallelismo definito a priori dal programmatore o dal compilatore (IA-64)
  - **Scheduling dinamico**:
    parallelismo deciso automaticamente dalla CPU (MIPS, IA-32).
    - 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*

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

Scheduling statico: *(IA-64)*

Scheduling statico: parallelismo definito a priori dall'utente (programmatore o compilatore)

IA-64: Instruction-level parallelism (ILP) implementato esplicitamente

- **EPIC**: Explicitly Parallel Instruction Computer

- 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!
```

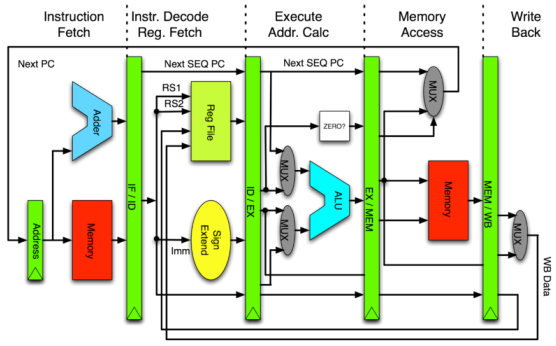

Pipeline Superscalari: *MIPS*

Famiglia MIPS:

R2000: pipeline **classica**

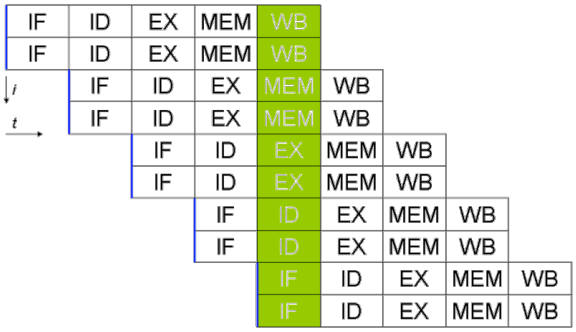
R3000: pipeline **doppia**

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



R8000: pipeline **superscalare**

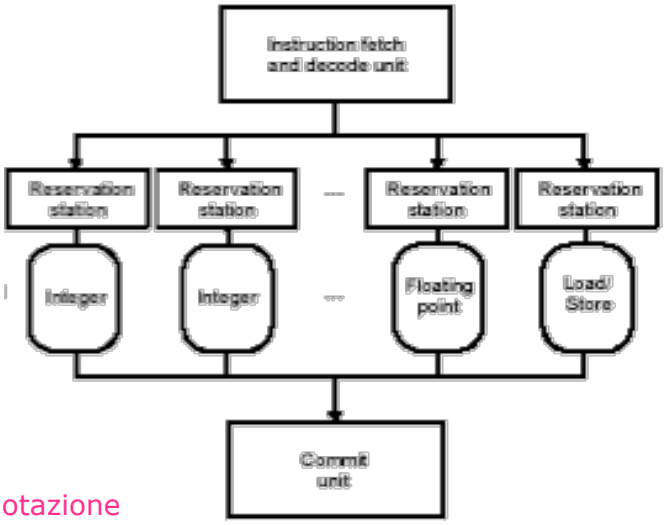
- Replicate le unità funzionali per eseguire più istruzioni in parallelo
- Ottengo più di 1 istruzione per ciclo di clock: **N > 1 instr. / CC**



Schedulazione dinamica:

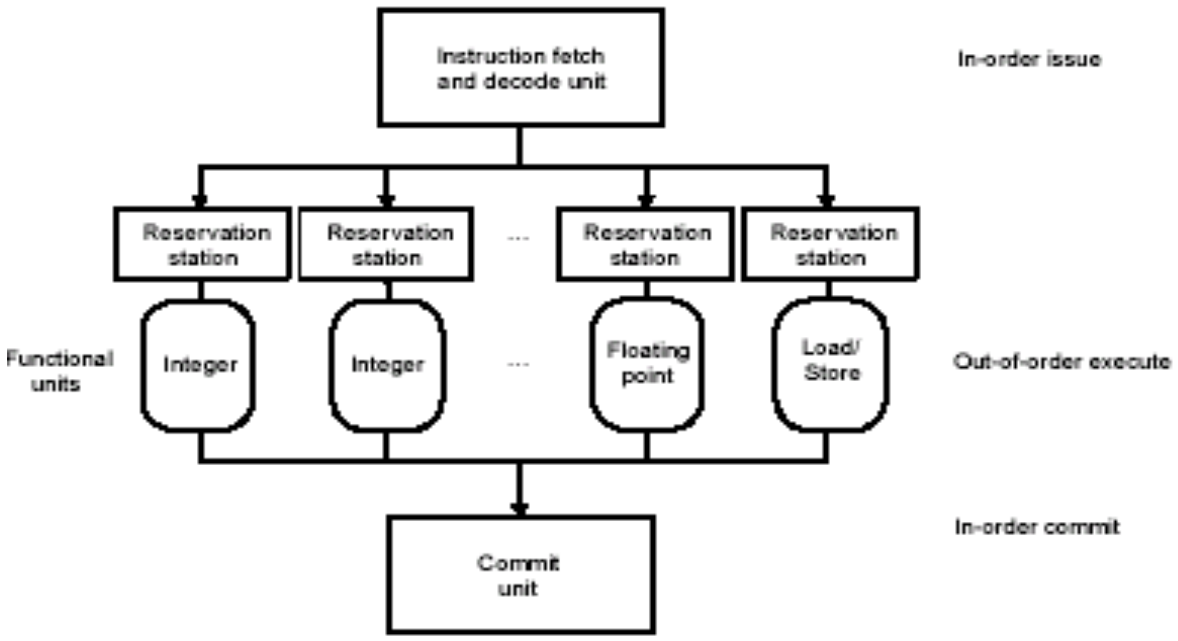
Schedulazione Dinamica:

- **Fetch**:
  Prelievo ed invio alla coda istruzioni
- **Decodifica**:
  Conversione in microistruzioni
  - Valutazione criticità (branch)



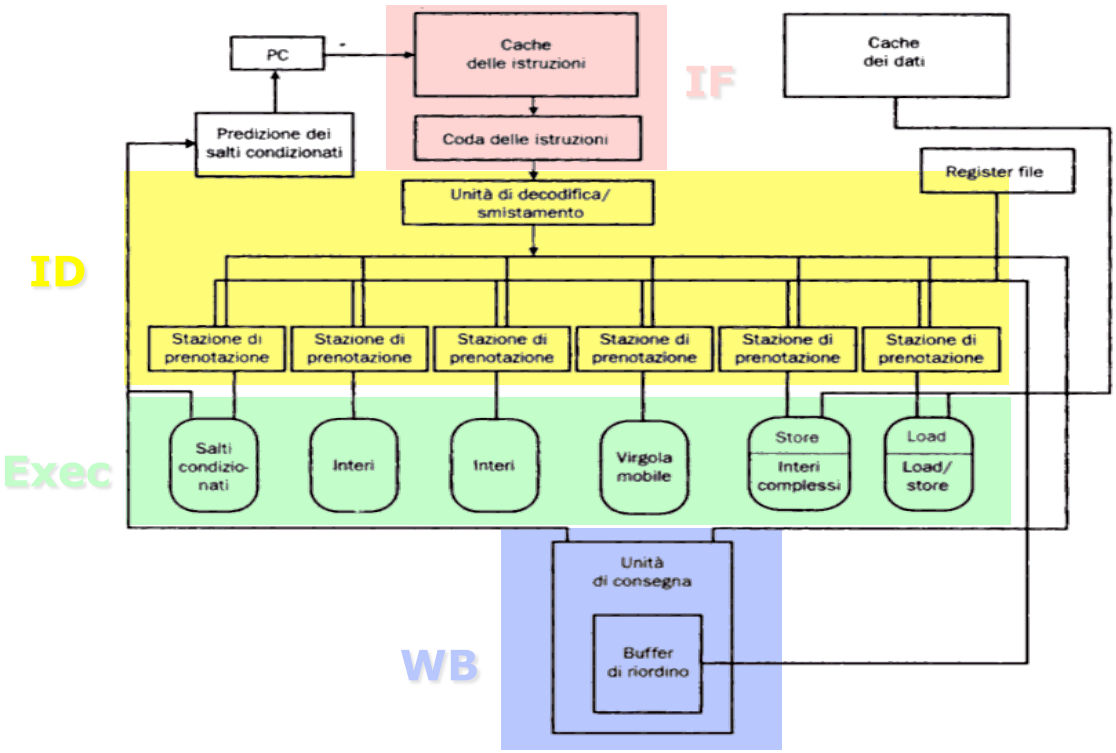
- **Esecuzione**:
  Le istruzioni arrivano alla stazione di prenotazione corrispondente all'unità funzionale richiesta (smistamento)
  - Esecuzione vera e propria (calcolo)
  - Eventuali criticità portano a far attendere alcune istruzioni
- **Commit Unit:**
  Riordino istruzioni per rispettare l'ordine sequenziale
  - Può completare più istruzioni in un ciclo di clock.

Pipeline con schedulazione dinamica



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

Scheduling dinamico: PentiumPro



Pipeline PentiumPro: funzionamento

- **Fetch**
  Prelievo di 16 byte dalla MI ed invio alla coda istruzioni
- **Decodifica**
  Conversione in microistruzioni di lunghezza fisssa: **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 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 delle 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
  - 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 4

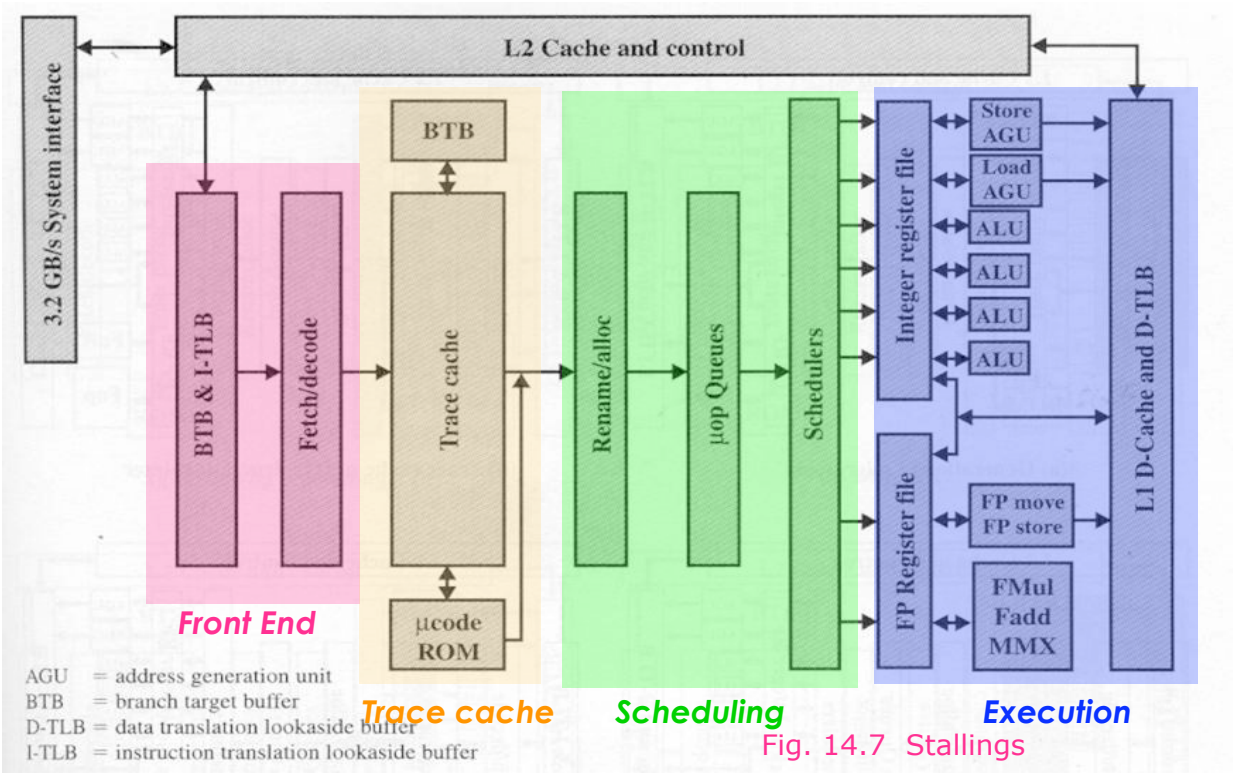


Fig. 14.7 Stallings

Pipeline del **Pentium 4**

- **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**

- 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 e look-aside buffer 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 4: **Esecuzione**

- Elemento centrale è il **Reorder Buffer** o **Scheduler**
  - 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.