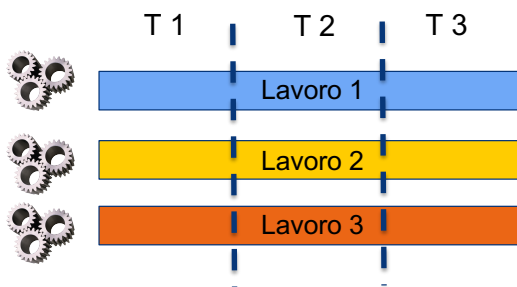
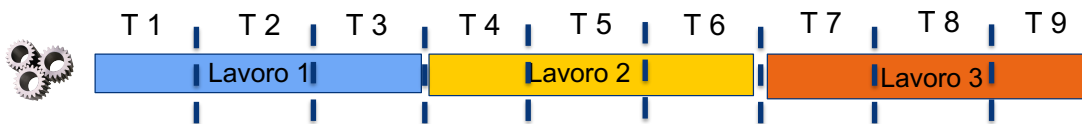


Parallelismo

- eseguire più attività **contemporaneamente**
- il lavoro viene **completato in meno tempo**



in 3 unità di Tempo

- 1 esecutore termina 1 solo lavoro
- 3 esecutori terminano 3 lavori

Parallelismo totale

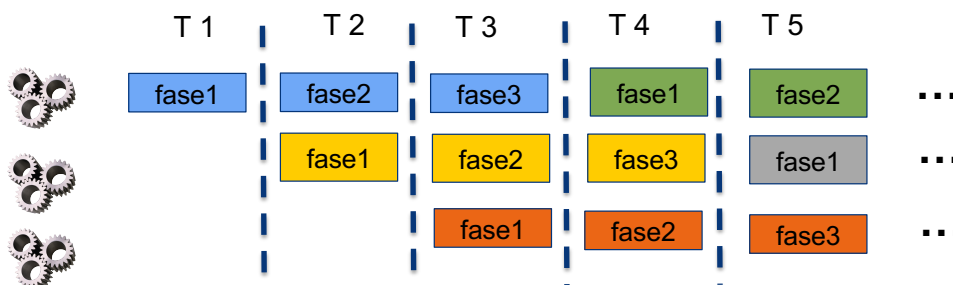
MA

se c'è **dipendenza funzionale**
tra un lavoro e il successivo?

Parallelismo

Dipendenza funzionale tra lavori successivi

- ogni lavoro è diviso in 3 fasi successive
- la fase 1 di *Lavoro i* deve essere eseguita dopo la fase 1 del precedente *Lavoro i-1*



Esecutori generici:

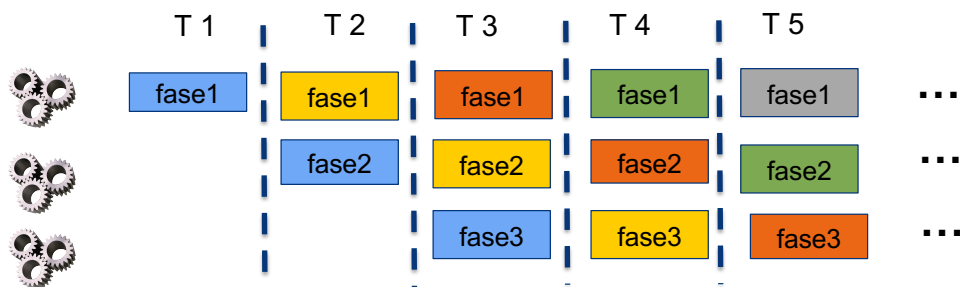
- ognuno esegue un lavoro completo
- **a regime** ha lo stesso throughput del parallelismo totale
- ognuno ha le risorse necessarie per ogni fase: *sistema totalmente replicato*

Parallelismo



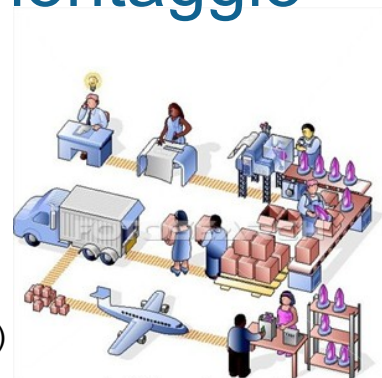
Esecutori specializzati

- ogni esecutore svolge sempre la stessa fase di ognuno dei lavori
- ogni esecutore **ha solo le risorse per eseguire quella fase**
- ogni lavoro passa da un esecutore all'altro
- **a regime** ha lo stesso throughput del parallelismo totale, ma usando meno risorse



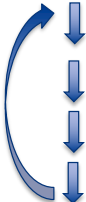
Pipeline – catena di montaggio

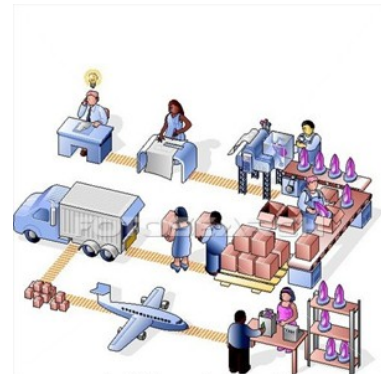
- si **decompone** un lavoro in **fasi successive**
 - un prodotto deve passare **una fase dopo l'altra**
 - ogni fase è realizzata da un **diverso operatore**
 - **nello stesso istante**
 - prodotti diversi sono in fasi diverse (**parallelismo**)
 - **l'istante successivo**
 - ogni fase ripete lo stesso lavoro sul prodotto successivo,
 - ogni lavoro avanza alla fase successiva
 - operatori/fasi diverse usano risorse diverse evitando conflitti (se possibile)



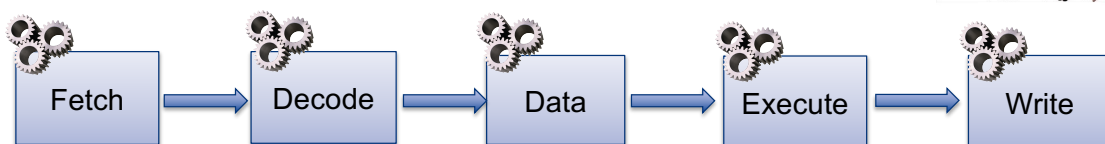
Pipeline – catena di montaggio

diverse **fasi del ciclo esecutivo** di un'istruzione

- 
- prelevare istruzioni (*fetch instruction*)
 - interpretare istruzioni (*decode instruction*)
 - prelevare dati (*fetch data*)
 - elaborare dati (*execute instruction*)
 - memorizzare dati (*write data*)

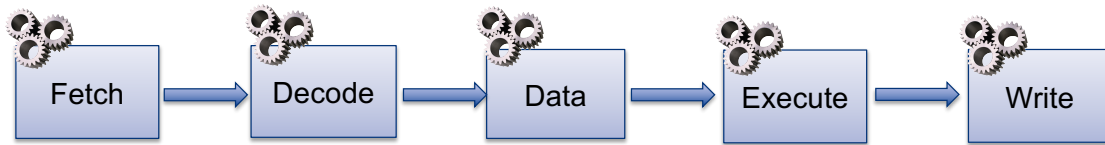
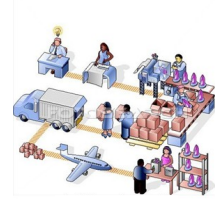


Pipeline



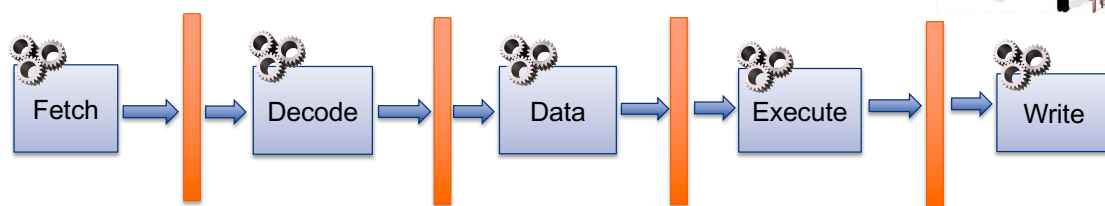
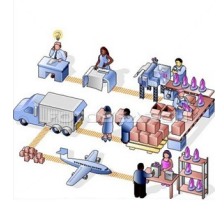
- **nello stesso istante:**
 - **istruzioni** diverse sono in fasi diverse
- **l'istante successivo**
 - ogni fase ripete lo stesso lavoro **sull'istruzione** successiva
 - ogni **istruzione** avanza alla fase successiva

Pipeline



- ogni fase è realizzata da una ***diversa unità funzionale della CPU***
- operatori/fasi diverse usano risorse diverse evitando conflitti (se possibile)

Pipeline



- ogni fase è realizzata da una ***diversa unità funzionale della CPU***
- operatori/fasi diverse usano risorse diverse evitando conflitti (se possibile)
- tra due fasi successive si inseriscono dei **buffer (registri)** su cui si scrivono/leggono **dati temporanei** utili alla fase successiva

Miglioramento delle prestazioni?

il prefetch non raddoppia le prestazioni:

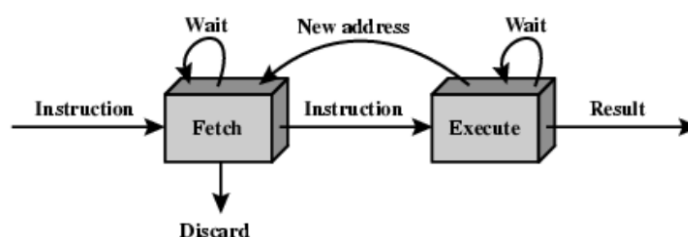
- la fase di **fetch è più breve**, ma prima di poter iniziare il fetch successivo **deve attendere** che termini anche la fase di esecuzione



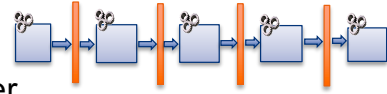
Miglioramento delle prestazioni?

il prefetch non raddoppia le prestazioni:

- la fase di **fetch è più breve**, ma prima di poter iniziare il fetch successivo **deve attendere** che termini anche la fase di esecuzione
- se viene eseguito un **jump o branch**, la prossima istruzione da eseguire **non è quella che è appena stata prelevata**:
 - la fase di **fetch deve attendere** che la fase **execute** le fornisca l'indirizzo a cui prelevare l'istruzione
 - la successiva fase di **execute deve attendere** che sia prelevata l'istruzione, perché quella pre-fetched non era valida



Miglioramento delle prestazioni?



- la suddivisione in fasi **aggiunge overhead** per
 - spostare i dati nei buffer tra una fase e l'altra
 - per gestire il cambiamento di fase
- questo overhead potrebbe essere significativo quando:
 - istruzioni successive **dipendono logicamente** da quelle precedenti,
 - quando ci sono **salti**,
 - quando ci sono **conflitti negli accessi alla memoria/registri**
- la gestione logica e l'overhead aumentano con l'aumentare del numero di fasi della pipeline

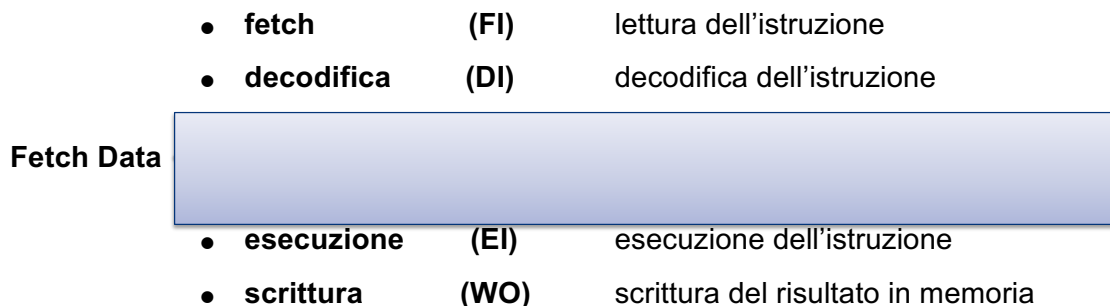
progettazione accurata per ottenere
risultati ottimali con una **complessità** ragionevole



Pipeline – evoluzione ideale

Per aumentare le prestazioni bisogna

- decomporre il lavoro in un **maggior numero di fasi**
- cercare di rendere le fasi più **indipendenti** e con una **durata simile**









Pipeline – evoluzione ideale

Per aumentare le prestazioni bisogna

- decomporre il lavoro in un **maggior numero di fasi**
- cercare di rendere le fasi più **indipendenti** e con una **durata simile**

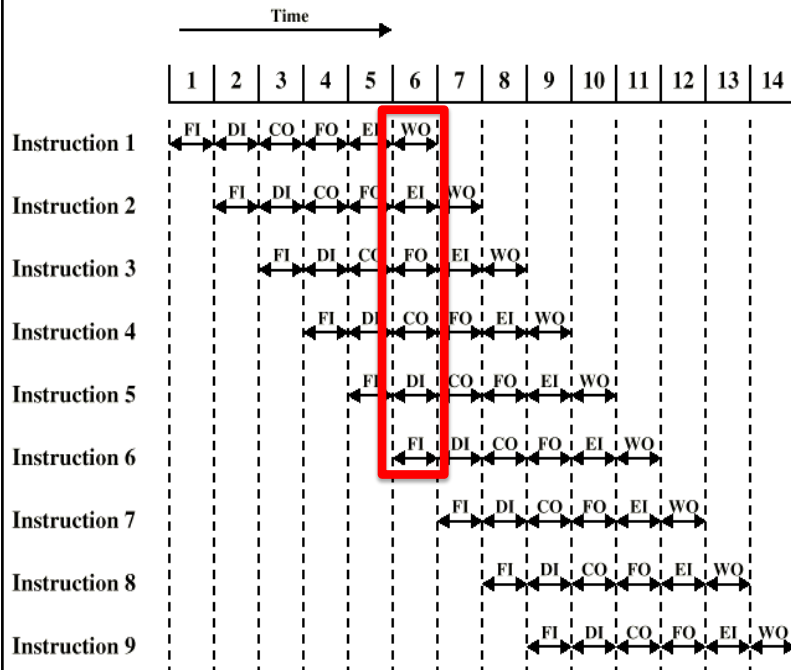
	•	fetch	(FI)	lettura dell'istruzione
	•	decodifica	(DI)	decodifica dell'istruzione
Fetch Data	•	calcolo ind. op. (CO)		calcolo indirizzo effettivo operandi
	•	fetch operandi (FO)		lettura degli operandi in memoria
	•	esecuzione	(EI)	esecuzione dell'istruzione
	•	scrittura	(WO)	scrittura del risultato in memoria

Pipeline – evoluzione ideale

		time →													
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
 FI		Ist1	Ist2	Ist3	Ist4	Ist5	Ist6	Ist7	Ist8	Ist9					
 DI			Ist1	Ist2	Ist3	Ist4	Ist5	Ist6	Ist7	Ist8	Ist9				
 CO				Ist1	Ist2	Ist3	Ist4	Ist5	Ist6	Ist7	Ist8	Ist9			
 FO					Ist1	Ist2	Ist3	Ist4	Ist5	Ist6	Ist7	Ist8	Ist9		
 EI						Ist1	Ist2	Ist3	Ist4	Ist5	Ist6	Ist7	Ist8	Ist9	
 WO							Ist1	Ist2	Ist3	Ist4	Ist5	Ist6	Ist7	Ist8	Ist9

esegue 9 istruzioni
in 14 unità di tempo
invece di $9 \times 6 = 54$

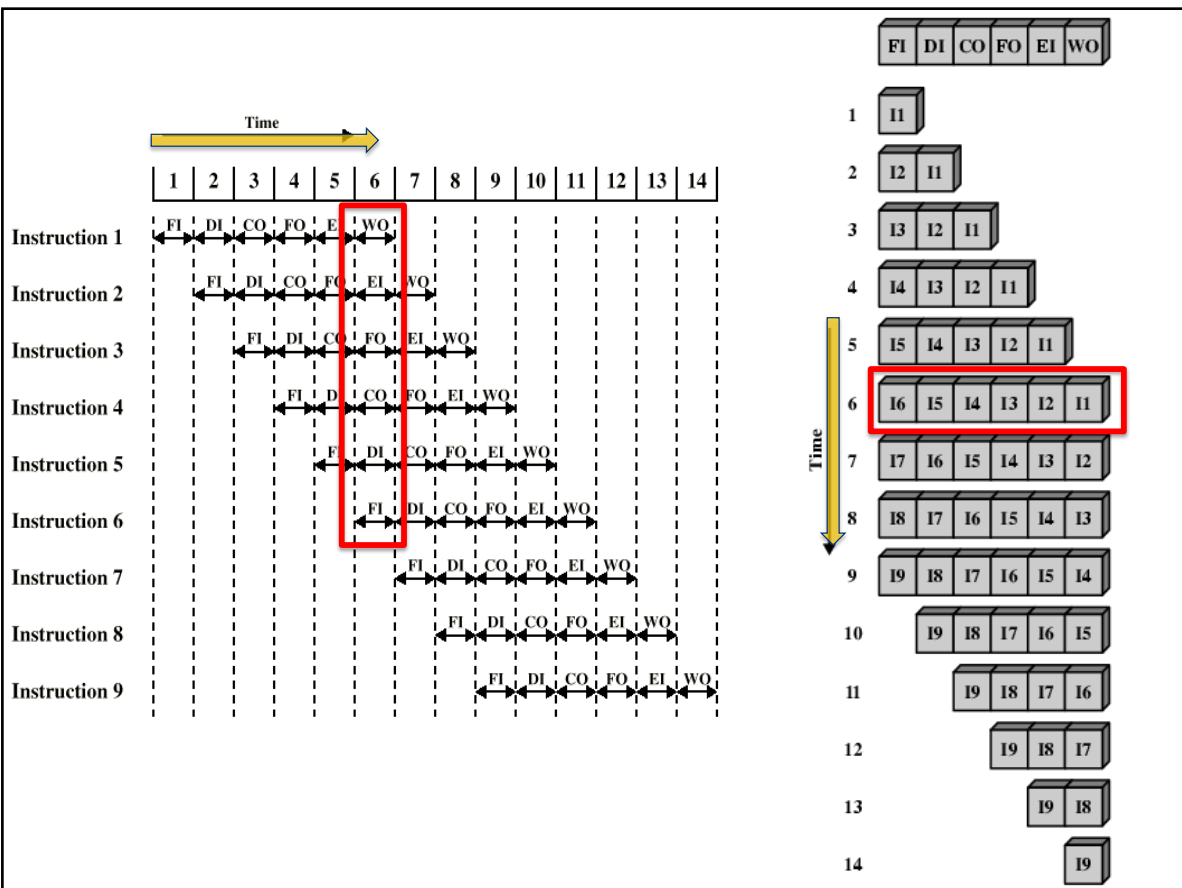
Pipeline – evoluzione ideale



esegue 9 istruzioni
in 14 unità di tempo
invece di $9 \times 6 = 54$

Assunzioni:

- ogni fase ha durata uguale
- ogni istruzione passa per tutte le fasi (e.g. LOAD non necessita WO)
- FI, FO, WO possono accedere alla memoria parallelamente senza fare conflitti
- non ci sono salti, né interrupt, né dipendenze



Pipeline performance

- Sia τ il **tempo di ciclo** di una pipeline
 - cioè il tempo necessario per far **avanzare di uno stadio/fase** le istruzioni attraverso una pipeline
 - può essere determinato come segue:

$$\tau = \max_i [\tau_i] + d = \tau_m + d \quad 1 \leq i \leq k$$

massimo ritardo di stadio (ritardo dello stadio più oneroso)

ritardo di commutazione di un registro, richiesto per l'avanzamento di segnali e dati da uno stadio al successivo

numero di stadi nella pipeline

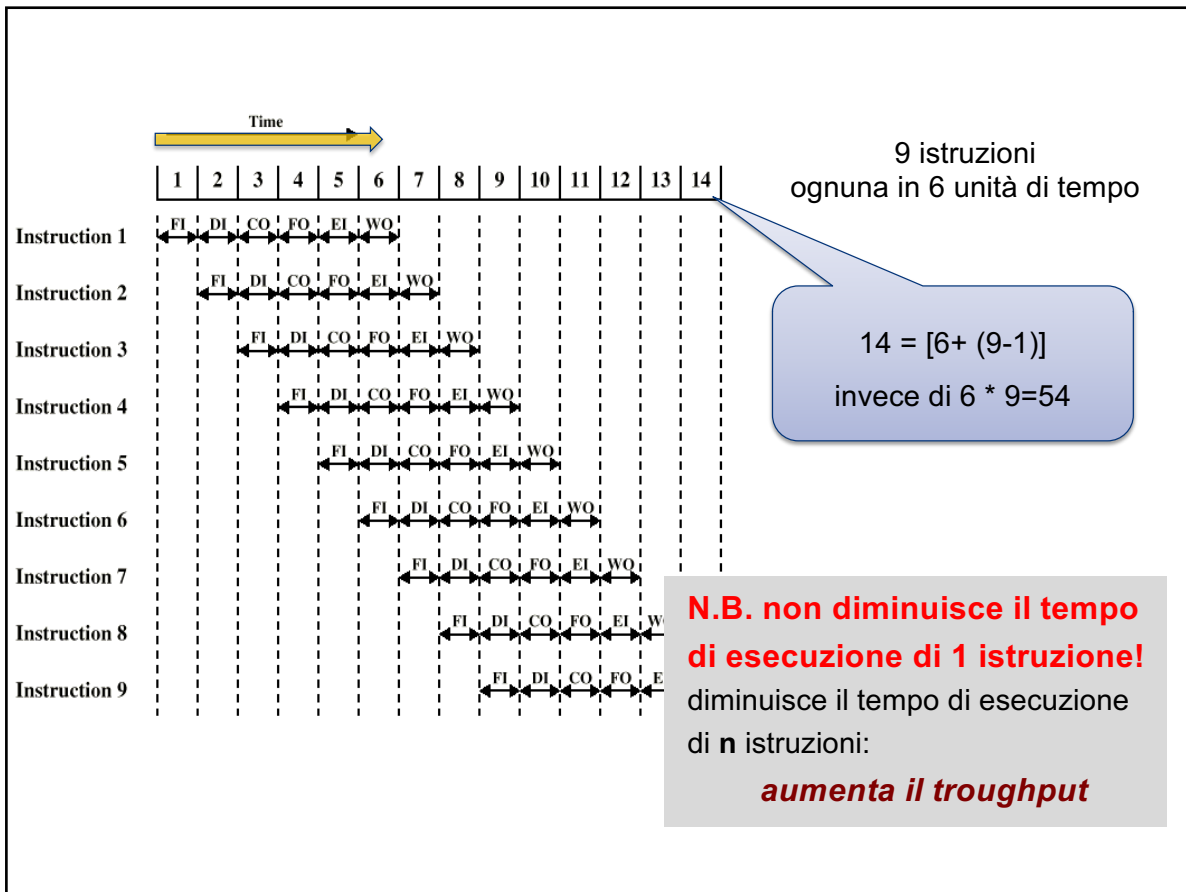
$\tau_m \gg d$

Pipeline performance ideali

Tempo totale richiesto da una pipeline con k stadi per eseguire n istruzioni (approssimazione e assumendo no salti)

$$T_k = [k + (n-1)] \tau$$

Infatti in k cicli si completa la prima istruzione
in altri $n-1$ cicli si completano le altre $n-1$ istruzioni (ogni istruzione finisce la sua pipeline 1 ciclo dopo la precedente)



Pipeline performance ideali

Tempo totale richiesto da una pipeline con k stadi per eseguire n istruzioni (approssimazione e assumendo no salti)

$$T_k = [k + (n-1)] \tau$$

Speedup (fattore di velocizzazione)

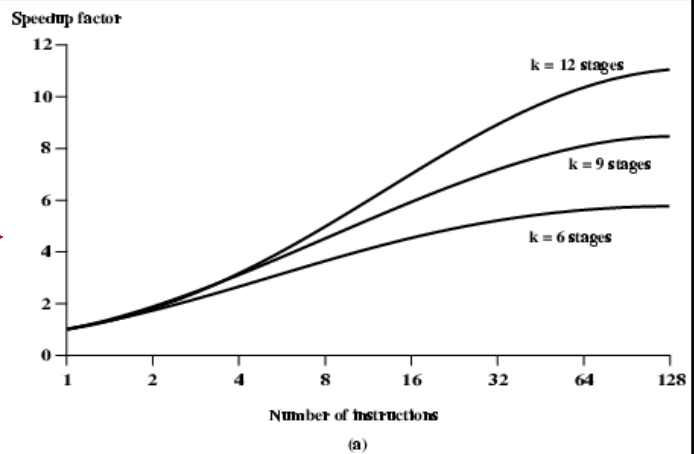
n istruzioni **senza** pipeline,
cioè 1 stadio di durata $k \tau$

$$S_k = \frac{T_1}{T_k} = \frac{nk\tau}{[k + (n-1)]\tau} = \frac{nk}{[k + (n-1)]}$$

Speedup

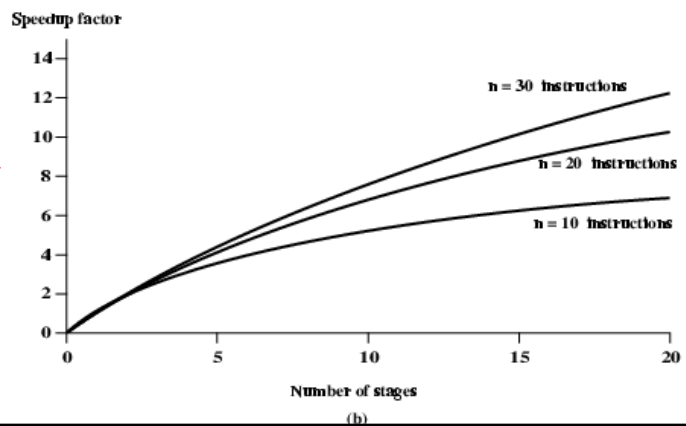
Calcolato in funzione
del numero di istruzioni

al crescere del numero di istruzioni
l'incremento di velocità si avvicina
al numero di stadi



Calcolato in funzione
del numero di stadi

pipeline con più stadi aumentano il
throughput, MA aggiungono
overhead e criticità (es. con salti)



pipeline hazards - criticità

- varie situazioni in cui l'istruzione successiva non può essere eseguita nel ciclo di clock immediatamente successivo (**stallo** – *pipeline bubble*)
non si raggiunge il parallelismo massimo

1. **sbilanciamento delle fasi**

- durate diverse per fase e per istruzione

2. problemi **strutturali** (*structural hazards*)

- due fasi competono per usare la stessa risorsa, es. memoria in FI, FO, WO

3. dipendenza dai **dati** (*data hazards*)

- un'istruzione dipende dal risultato di un'istruzione precedente ancora in pipeline

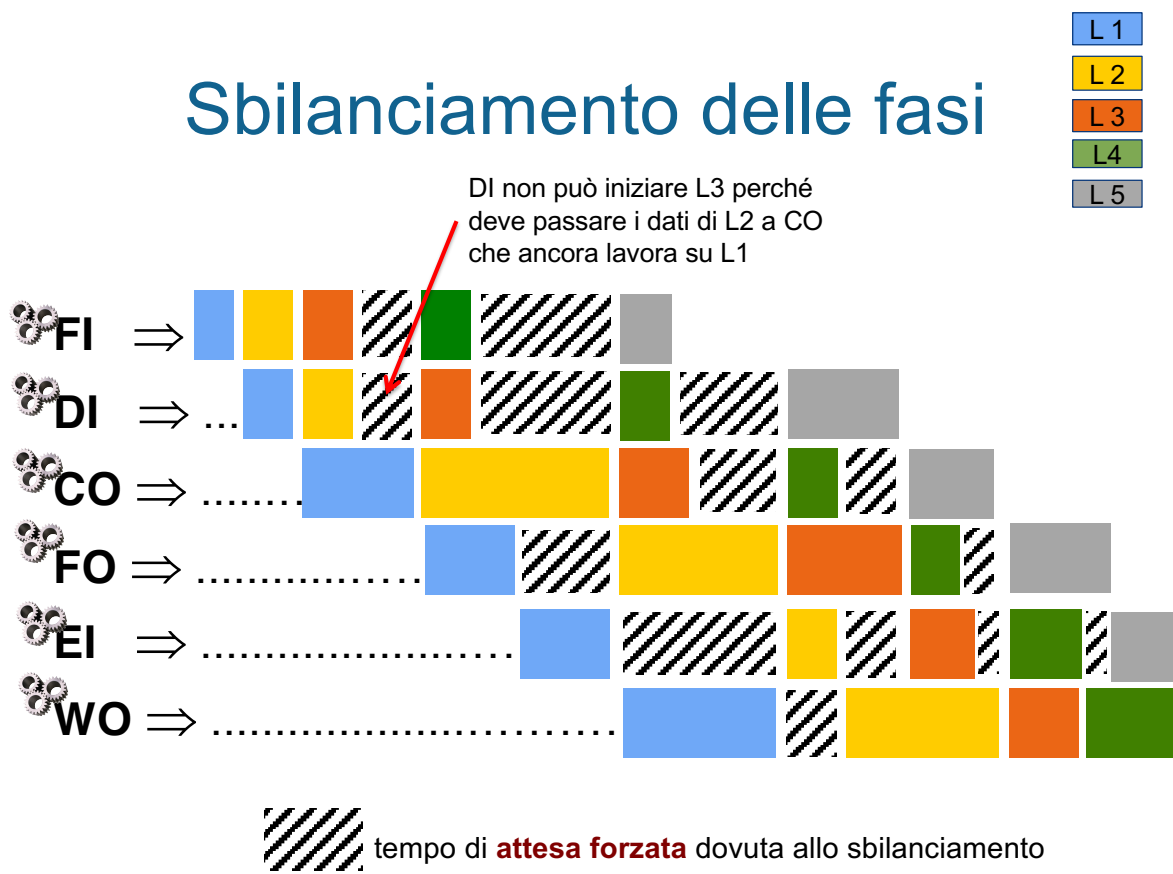
4. dipendenza dal **controllo** (*control hazards*)

- istruzioni che alterano la sequenzialità, es. salti (condizionati o no), chiamate e ritorni da procedure, interruzioni

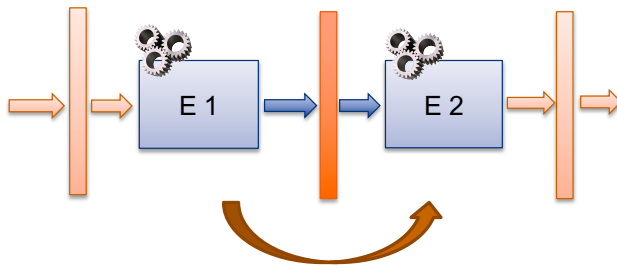
Sbilanciamento delle fasi

- Non tutte le fasi richiedono lo stesso tempo di esecuzione
es.: lettura di un operando tramite registro rispetto ad una mediante indirizzamento indiretto
- La suddivisione in fasi va fatta in base all'istruzione più onerosa
- Non tutte le istruzioni richiedono le stesse fasi e le stesse risorse

Sbilanciamento delle fasi



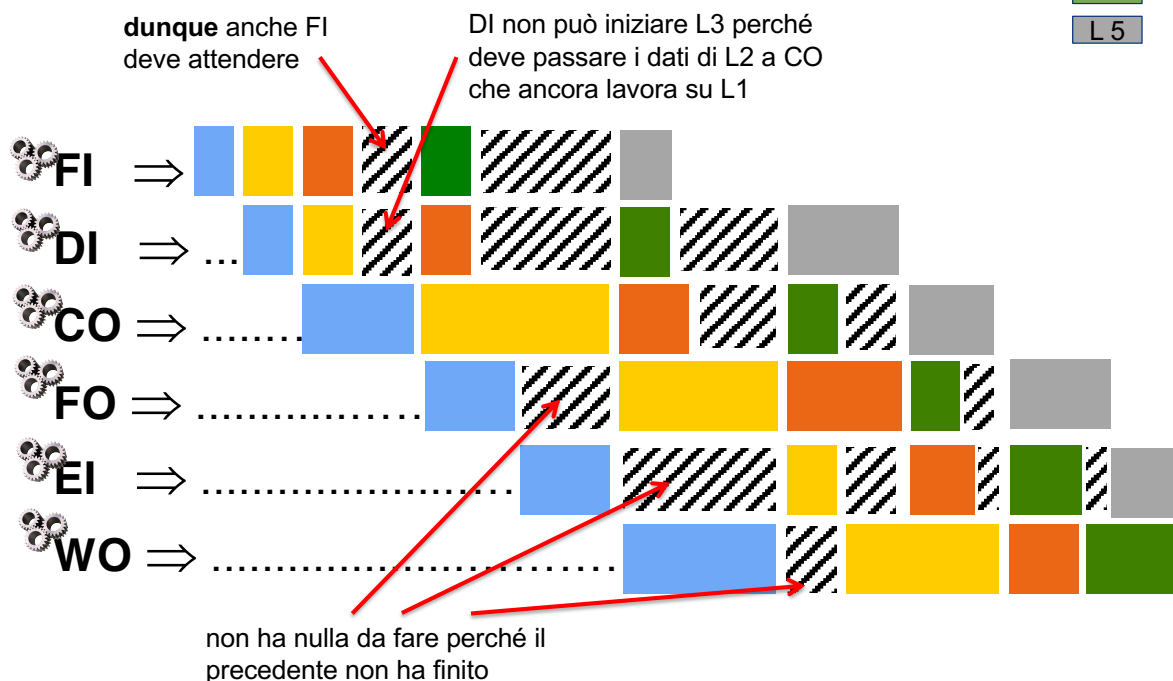
Sbilanciamento delle fasi



passare i dati significa che

- l'esecutore E1 mentre lavora **scrive** sul registro intermedio
- l'esecutore E2 nel ciclo successivo **leggerà** questi dati
- se E1 comincia il lavoro successivo prima che anche E2 cominci il lavoro successivo, allora E1 può **sovrascrivere** i dati nel registro prima che E2 li abbia letti

Sbilanciamento delle fasi



Sbilanciamento delle fasi



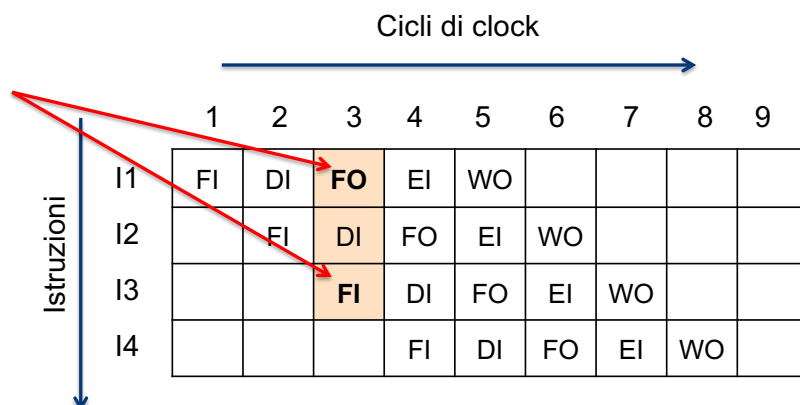
Possibili soluzioni:

- Decomporre fasi onerose in più sottofasi
 - Costo elevato e bassa utilizzazione
- Duplicare gli esecutori delle fasi più onerose e farli operare in parallelo
 - CPU moderne hanno una **ALU in aritmetica intera** ed una in **aritmetica a virgola mobile**

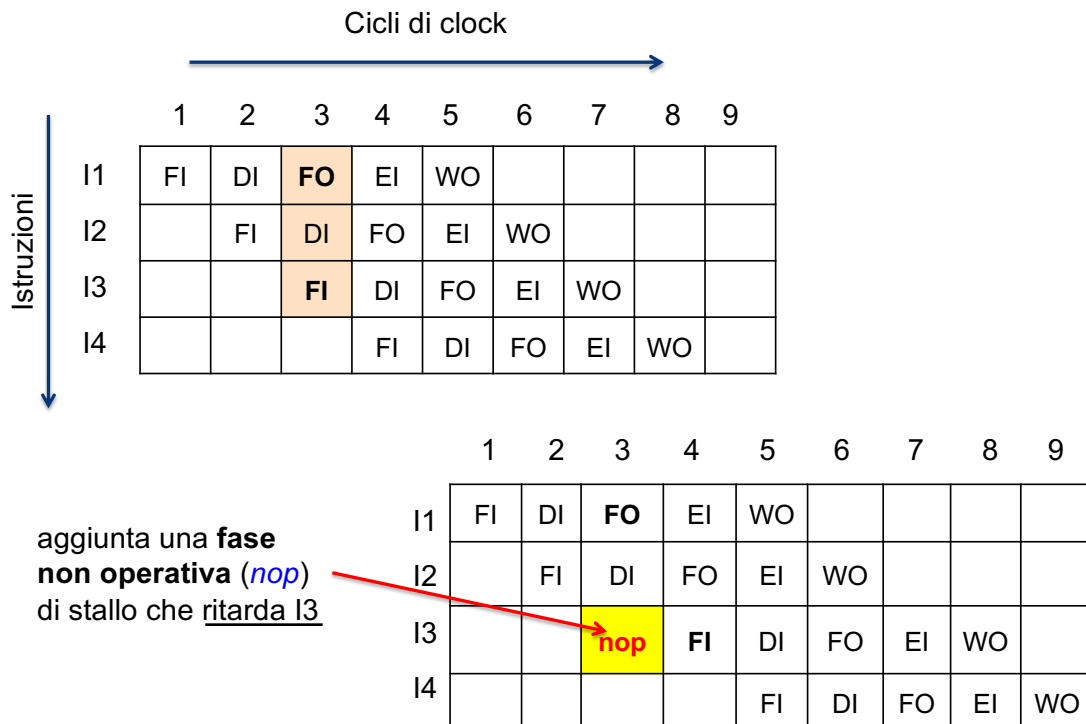
Problemi strutturali

- due (o più) istruzioni già nella pipeline (i.e., l'esecuzione di due o più fasi) richiedono di **accedere ad una stessa risorsa nello stesso ciclo di clock**
- **quindi** gli accessi devono avvenire in **sequenza** e **non in parallelo**

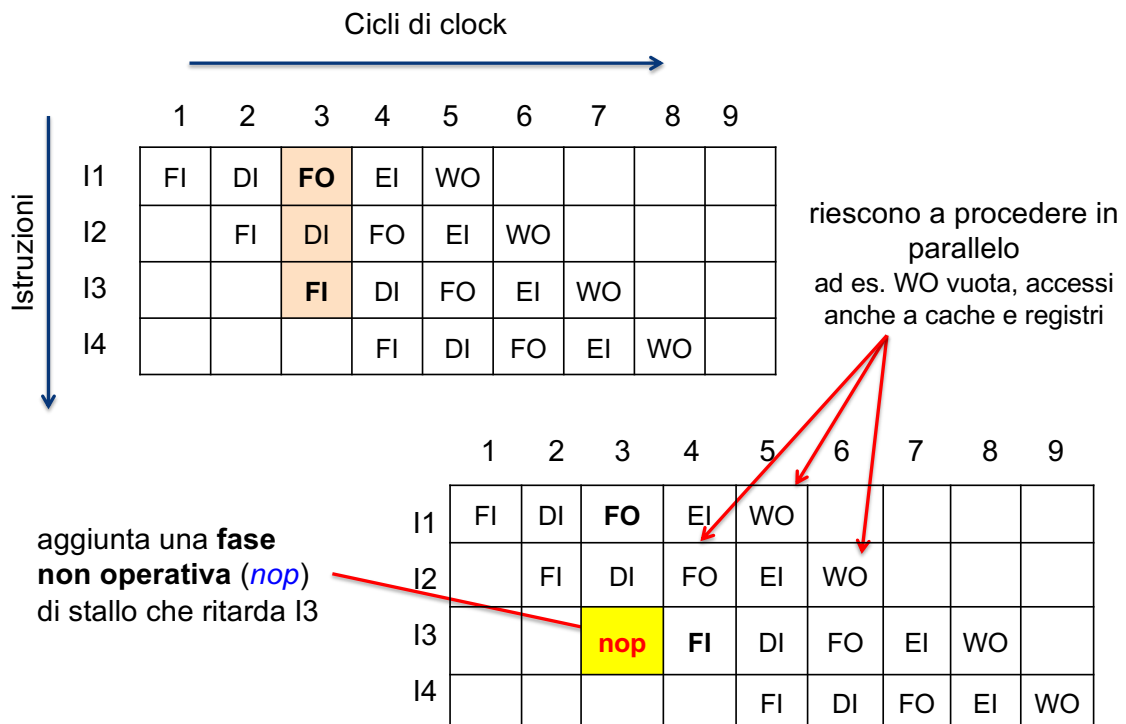
se l'operando di I1
è in memoria,
non si può fare FI
di I3 in parallelo



Problemi strutturali



Problemi strutturali



Problemi strutturali

- due (o più) istruzioni già nella pipeline (i.e., l'esecuzione di due o più fasi) richiedono di **accedere ad una stessa risorsa nello stesso ciclo di clock**
- **quindi** gli accessi devono avvenire in **sequenza** e **non in parallelo**
- es. FI, FO, WO potrebbero dover accedere alla memoria principale (perché i dati non risiedono nella cache o nei registri)

Soluzioni:

- introdurre fasi non operative (nop)
- **suddividere le memorie** permettendo accessi paralleli: una memoria cache per le **istruzioni** e una per i **dati**

Dipendenza dai dati

- una fase non può essere eseguita in un certo ciclo di clock perché i **dati** di cui ha bisogno **non sono ancora disponibili**
 - deve attendere che termini l'elaborazione di un'altra fase
- un dato modificato nell'esecuzione dell'istruzione **corrente** può dover essere utilizzato dalla fase **FO** dell'istruzione **successiva**

add **\$1**, \$2, \$3 $R1 \leftarrow [R2] + [R3]$

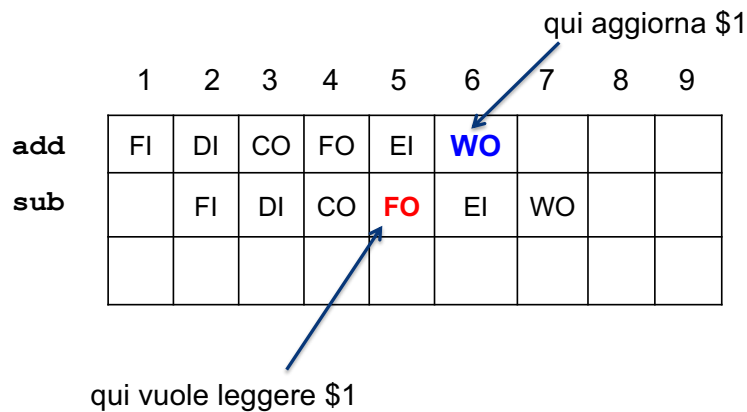
sub \$4, **\$1**, \$5 $R4 \leftarrow [R1] - [R5]$

**la seconda istruzione dipende dal risultato della prima,
che si trova ancora all'interno della pipeline!**

Dipendenza dai dati

add \$1, \$2, \$3 $R1 \leftarrow [R2] + [R3]$

sub \$4, \$1, \$5 $R4 \leftarrow [R1] - [R5]$



Dipendenza dai dati

add \$1, \$2, \$3 $R1 \leftarrow [R2] + [R3]$

sub \$4, \$1, \$5 $R4 \leftarrow [R1] - [R5]$



due cicli di stallo

Dipendenza dai dati

add \$1, \$2, \$3 $R1 \leftarrow [R2] + [R3]$

sub \$4, \$1, \$5 $R4 \leftarrow [R1] - [R5]$

	1	2	3	4	5	6	7	8	9	10
add	FI	DI	CO	FO	EI	WO				
sub		FI	DI	CO	nop	nop	FO	EI	WO	
Istr3			FI	DI				CO	FO	EI
Istr4				FI				DI	CO	FO

due cicli di stallo per **tutte** le istruzioni

Data hazards

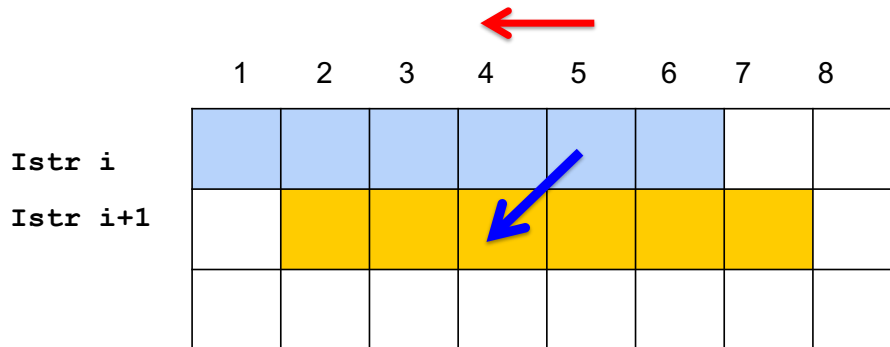
istruzione i

istruzione $i+1$

- **Read after Write** : “lettura dopo scrittura” (esempio di prima)
 - $i+1$ legge **prima** che i abbia scritto
- **Write after Write** : “scrittura dopo scrittura”
 - $i+1$ scrive **prima** che i abbia scritto
- **Write after Read**: “scrittura dopo lettura”
 - $i+1$ scrive **prima** che i abbia letto (caso raro in pipeline)

Data hazards

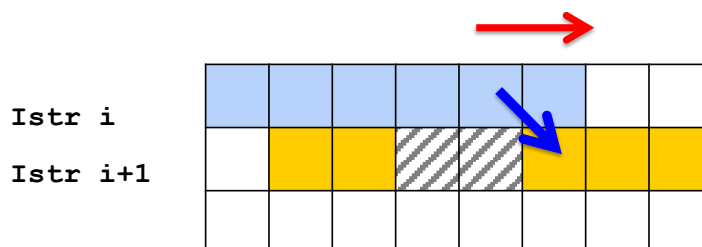
l'istruzione successiva ha bisogno dei dati **prima** che la precedente li abbia prodotti



- dipende dall'architettura della pipeline: da come sono definiti i suoi stadi e come sono implementate le istruzioni

Dipendenza dai dati - Soluzioni

1. Introduzione di fasi non operative (**nop**-stallo)



2. propagazione in avanti del dato richiesto (**data forwarding** – bypassing)

Data forwarding

ma il valore di \$1 si conosce
già all'uscita della ALU

qui aggiorna \$1

add \$1 \$2 \$3

sub \$4 \$1 \$5

FI	DI	CO	FO	EI	WO		
	FI	DI	CO	FO	EI	WO	

1 solo ciclo di stallo

FI	DI	CO	FO	EI	WO		
	FI	DI	CO	nop	FO	EI	WO

un circuito
riconosce la dipendenza e
propaga in avanti l'output
della ALU

Dipendenza dai dati - Soluzioni

1. Introduzione di fasi non operative (**nop**-stallo)
2. propagazione in avanti del dato richiesto (**data forwarding**)
 - dipende da architettura di pipeline e implementazione istruzioni
3. **riordino delle istruzioni**

riordino delle istruzioni

programma C con 5 variabili
che si riferiscono a indirizzi di memoria

```
a = b + e;
c = b + f;
```

memoria indirizzata al byte (1 word=4 byte)

c		16
a		12
f		8
e		4
b		0

assumiamo
corrisponda a (\$t0)
così usiamo offset

compilatore produce il codice assembler

- associando i registri alle variabili del programma
- e trasferendo i dati tra la memoria e i registri

b - \$1 e - \$2 a - \$3
f - \$4 c - \$5

```
lw  $1  0  ($t0)
lw  $2  4  ($t0)
add $3  $1  $2
sw  $3  12 ($t0)
lw  $4  8  ($t0)
add $5  $1  $4
sw  $5  16 ($t0)
```

riordino delle istruzioni

programma C con 5 variabili
che si riferiscono a indirizzi di memoria

```
a = b + e;
c = b + f;
```

memoria indirizzata al byte (1 word=4 byte)

c	c	16
a	a	12
f	f	8
e	e	4
b	b	0

assumiamo
corrisponda a (\$t0)
così usiamo offset

```
lw  $1  0  ($t0)
lw  $2  4  ($t0)
add $3  $1  $2
sw  $3  12 ($t0)
lw  $4  8  ($t0)
add $5  $1  $4
sw  $5  16 ($t0)
```

tutte dipendenze **Read after Write**

quindi servono degli **stalli**

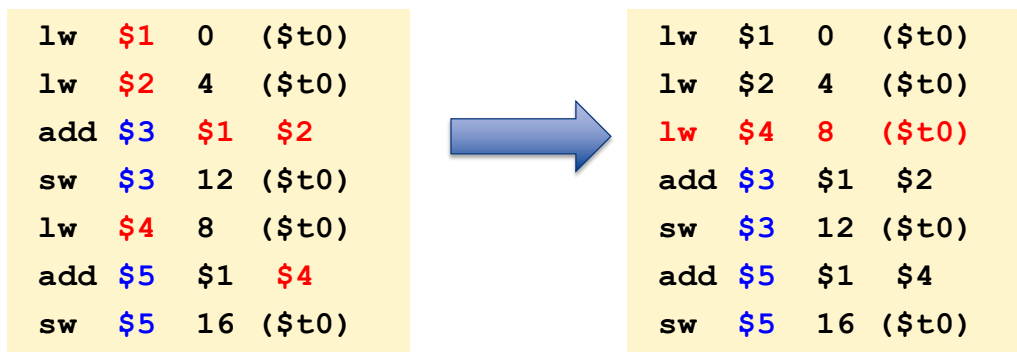
(a seconda di come è definita la pipeline, qualche problema può risolversi con *data forwarding*)

riordino delle istruzioni

programma C con 5 variabili
che si riferiscono a indirizzi di memoria

```
a = b + e;  
c = b + f;
```

riordinando le istruzioni si sono “ridotte”
le dipendenze **lw** - **add**



pipeline hazards - criticità

- varie situazioni in cui l'istruzione successiva non può essere eseguita nel ciclo di clock immediatamente successivo (**stallo** – *pipeline bubble*)
non si raggiunge il parallelismo massimo

1. sbilanciamento delle fasi

- durate diverse per fase e per istruzione

2. problemi **strutturali** (*structural hazards*)

- due fasi competono per usare la stessa risorsa, es. memoria in FI, FO, WO

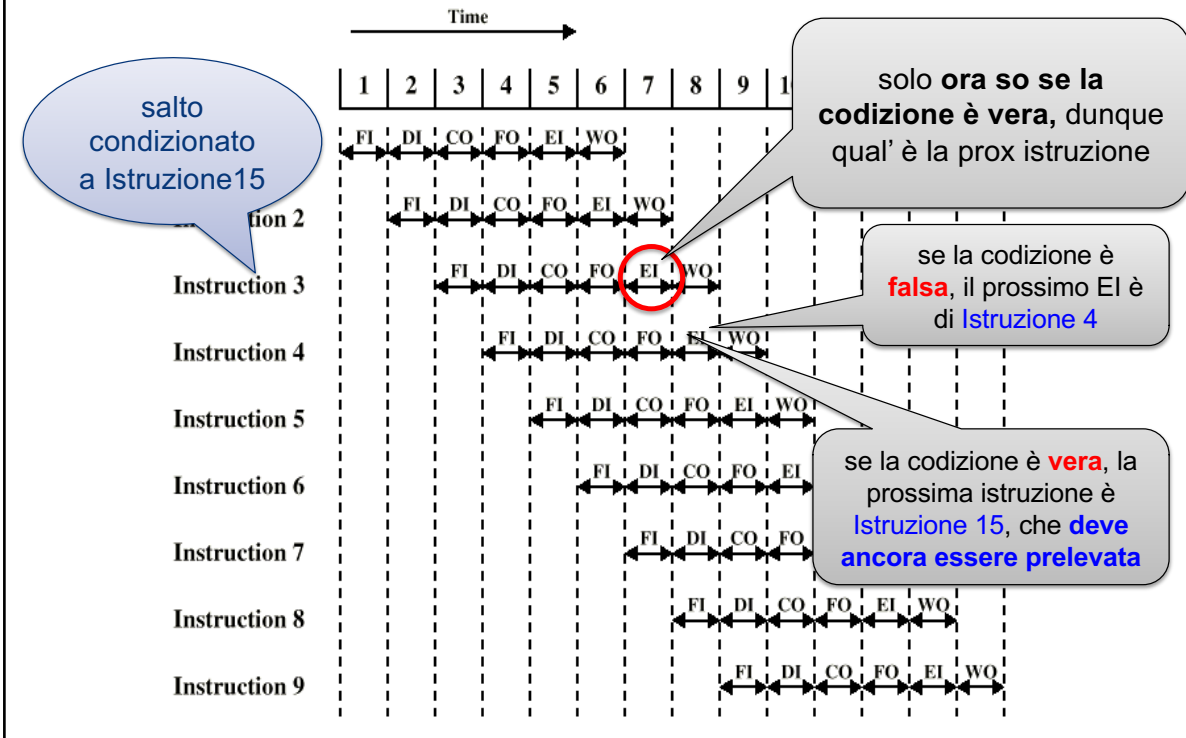
3. dipendenza dai **dati** (*data hazards*)

- un'istruzione dipende dal risultato di un'istruzione precedente ancora in pipeline

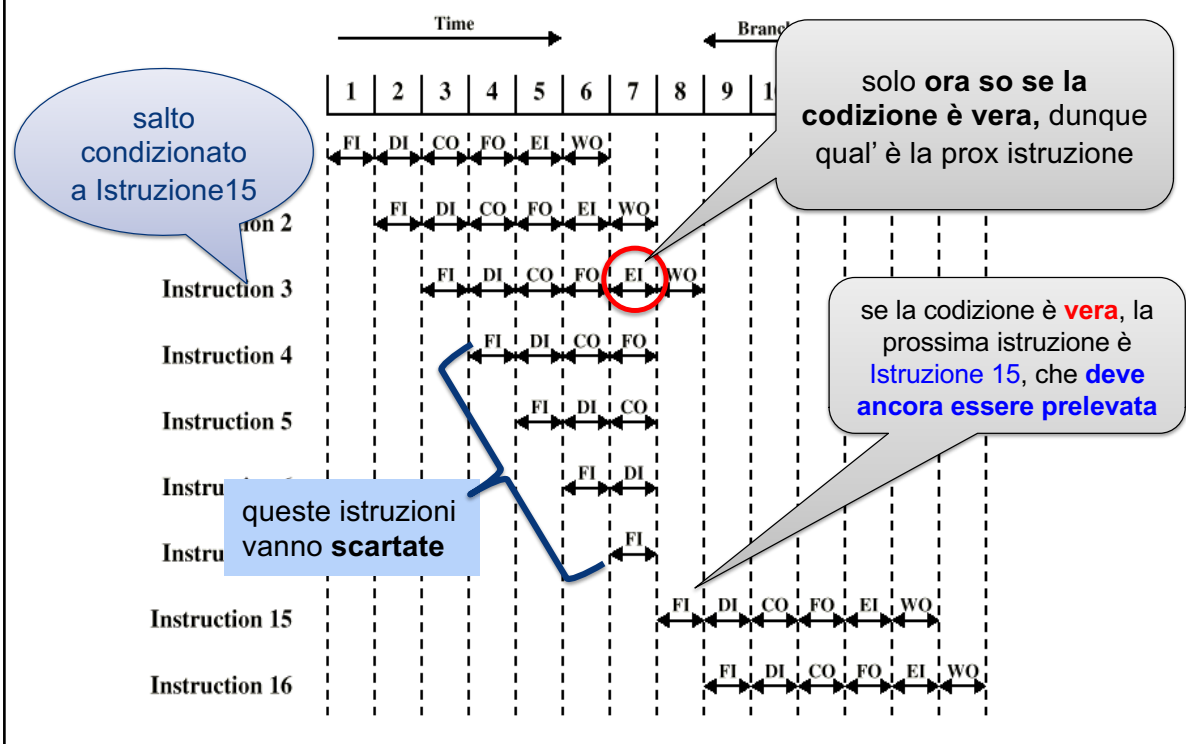
4. dipendenza dal **controllo** (*control hazards*)

- istruzioni che alterano la sequenzialità, es. salti (condizionati o no), chiamate e ritorni da procedure, interruzioni.

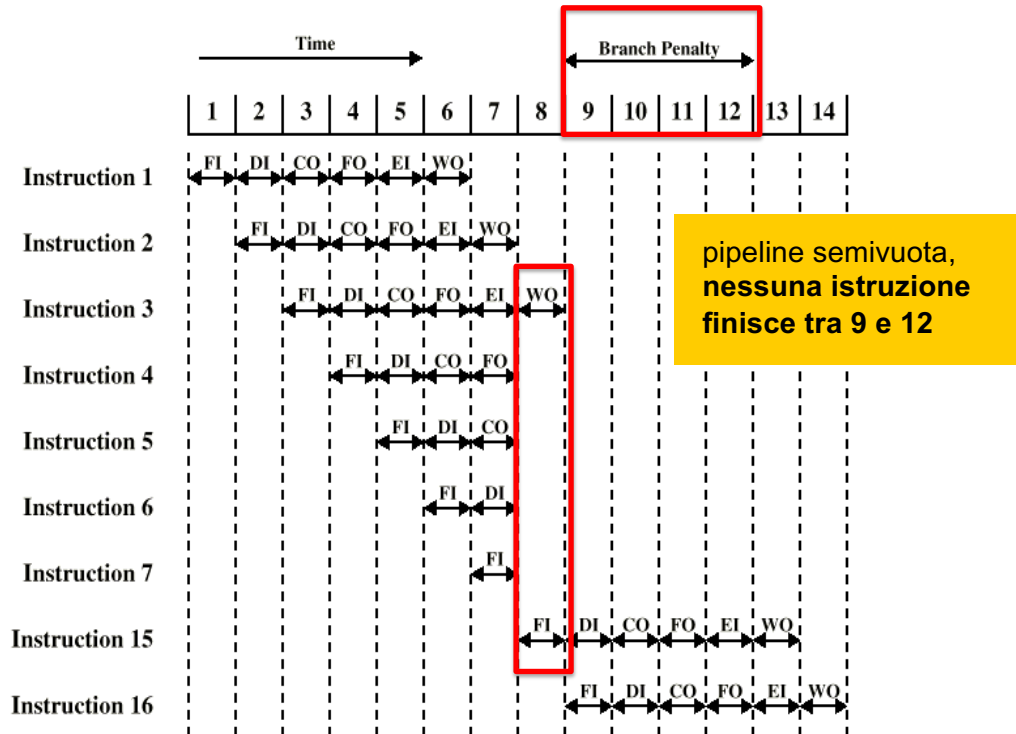
Salto condizionato



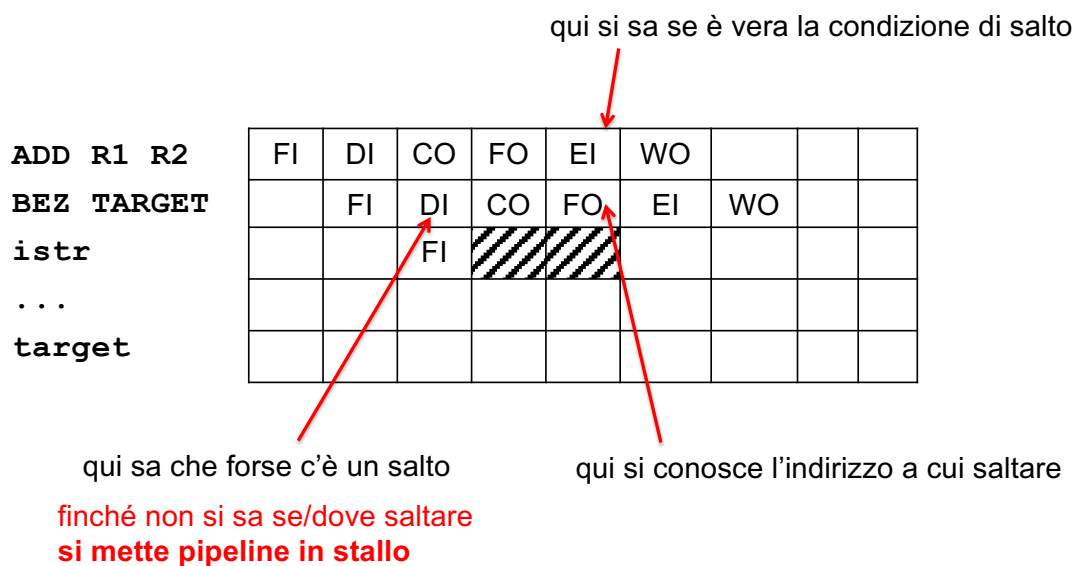
Salto condizionato



Salto condizionato



Esempio: salto condizionato



Esempio: salto condizionato

qui si sa se è vera la condizione di salto

ADD R1 R2
BEZ TARGET
istr
...
target

FI	DI	CO	FO	EI	WO			
	FI	DI	CO	FO	EI	WO		
		FI						
					FI	DI	CO	FO

qui si conosce l'indirizzo a cui saltare

supponiamo **condizione vera**:
si scarta istruzione pre-fetched
e si ricomincia con istruzione **target**

Esempio: salto condizionato

qui si sa se è vera la condizione di salto

ADD R1 R2
BEZ TARGET
istr
...
target

FI	DI	CO	FO	EI	WO			
	FI	DI	CO	FO	EI	WO		
		FI			DI	CO	FO	EI

qui si conosce l'indirizzo a cui saltare

supponiamo **condizione falsa**:
ripreso dopo lo stallo con
l'istruzione pre-fetched

Esempio

salto incondizionato

qui si sa che ci sarà un salto

qui si conosce l'indirizzo a cui saltare

BR TARGET
istr
...
target
target+1

FI	DI	CO	FO	EI	WO			

Esempi

salto incondizionato

qui si sa che ci sarà un salto

qui si conosce l'indirizzo a cui saltare

BR TARGET
istr
...
target
target+1

FI	DI	CO	FO	EI	WO			
	EI	/ / / / /						

ormai ha prelevato l'istruzione errata,
quindi mette in stallo e scarta **istr**

Esempi

salto incondizionato qui si sa che ci sarà un salto qui si conosce l'indirizzo a cui saltare

BR	TARGET	FI	DI	CO	FO	EI	WO			
istr		FI								
...										
target						FI	DI	CO	EI	WO
target+1							FI	DI	CO	EI

ormai ha prelevato l'istruzione errata
quindi mette in stallo e scarta **istr**
ricomincia con istruzione **target**

dipendenza dai controlli

- Uno dei maggiori problemi della progettazione della pipeline è **assicurare** un **flusso regolare di istruzioni**
 - violato da salti condizionati, salti non condizionati, *chiamate e ritorni da procedure*
 - se la fase fetch ha caricato un'istruzione errata, va scartata
 - queste istruzioni sono circa il 30% del totale medio di un programma

Soluzioni:

- **mettere in stallo** la pipeline finché non si è calcolato l'indirizzo della prossima istruzione. **semplice ma inefficiente**
- individuare le istruzioni critiche e aggiungere un'apposita **logica di controllo**. si **complica il compilatore e hardware** specifico

Soluzioni per salti condizionati

1. flussi multipli (*multiple streams*)

- replica la prima parte della pipeline, EI esclusa, per entrambi i rami possibili

```
istr i → se cond  
          istruzione n  
    else  
          istruzione i+1
```

inserisce nella pipeline sia
istruzione n che istruzione i+1

brute-force

- conflitti di accesso alle risorse tra i due stream
- se istruzione n (o i+1) contiene un salto aggiunge ulteriori stream

Soluzioni per salti condizionati

1. flussi multipli (*multiple streams*)

- replica la prima parte della pipeline, EI esclusa, per entrambi i rami possibili

2. prefetch anche dell'istruzione target

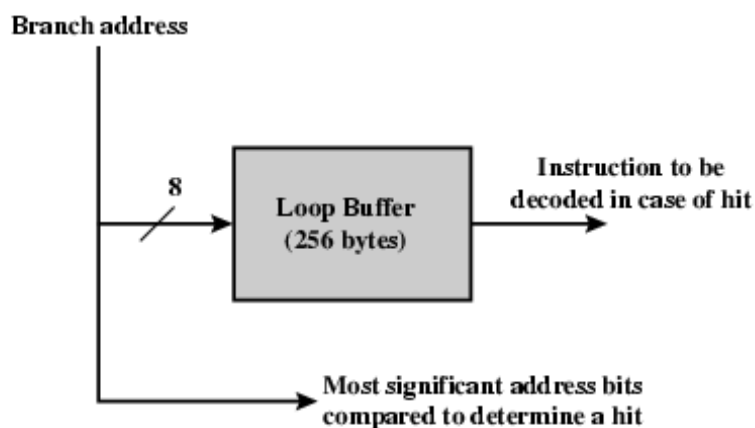
- anticipa il fetch dell'istruzione target oltre a quella successiva al salto
- se il salto è preso, trova l'istruzione già caricata
- in ogni caso una parte della pipeline deve essere scartata

Soluzioni per salti condizionati

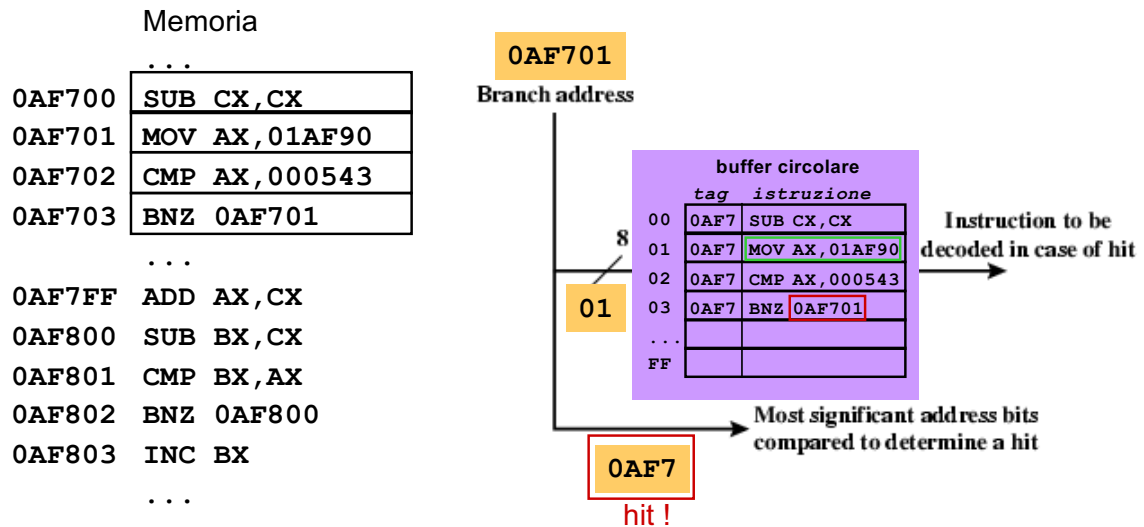
1. flussi multipli (*multiple streams*)
2. prefetch anche dell'istruzione target
3. buffer circolare (*loop buffer*)
 - è una memoria piccola e molto veloce che **mantiene le ultime n istruzioni prelevate**
 - in caso di salto l'hardware **controlla se l'istruzione target è tra quelle già dentro il buffer**, così da evitare il fetch
 - utile **in caso di loop**, specie se il buffer contiene tutte le istruzioni nel loop, così vengono prelevate dalla memoria una sola volta
 - può essere **accoppiato al pre-fetch**: riempio il buffer con un pò di istruzioni sequenzialmente successive alla corrente. Per molti if-then-else i due rami sono istruzioni vicine, quindi probabilmente entrambe già nel buffer

Buffer circolare (senza prefetch)

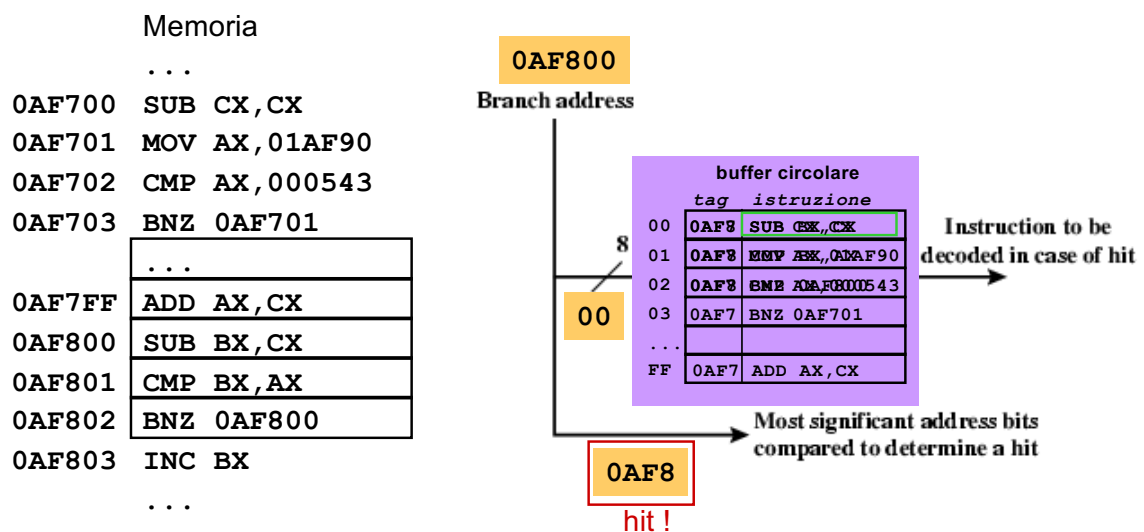
- buffer senza prefetch, capienza 256 bytes, indirizzato a byte
- dato l'*indirizzo* target di salto/branch, controllo se c'è nel buffer:
 - gli **8 bit meno significativi** sono usati come **indice nel buffer**
 - gli altri **bit più significativi** si usano per controllare **se** la destinazione del salto sta **già nel buffer**



Buffer circolare (senza prefetch)



Buffer circolare (senza prefetch)



Soluzioni per salti condizionati

1. flussi multipli (*multiple streams*)
2. prefetch dell'istruzione target
3. buffer circolare (*loop buffer*)

4. predizione dei salti

- cerco di predire se il salto sarà intrapreso o no

Varie possibilità:

- | | | |
|--|---|--------------------------|
| <ul style="list-style-type: none">• previsione di saltare sempre• previsione di non saltare mai (<i>molto usato</i>)• previsione in base al codice operativo | } | <i>approcci statici</i> |
| <ul style="list-style-type: none">• bit <i>taken/not taken</i>• tabella della storia dei salti | } | <i>approcci dinamici</i> |

Approcci dinamici di predizione

- cercano di migliorare la qualità della predizione sul salto **memorizzando la storia delle istruzioni di salto condizionato** di uno specifico programma
- ad ogni istruzione di salto condizionato associo **1 (o 2) bit** per ricordare la storia recente dell'istruzione, i.e. **se l'ultima (e la penultima) volta il salto è stato preso**
- bit memorizzati in una locazione temporanea ad accesso molto veloce

Approcci dinamici di predizione

associo 1 bit ad ogni istruzione di salto

- ricorda come è andata l'ultima volta, **predico di comportarsi nello stesso modo**
- se bit è **1** predico di **saltare**
- se bit è **0** predico di **non saltare**
- se ho **sbagliato** predizione **inverto il bit**

**a regime:
2 errori per ciclo**

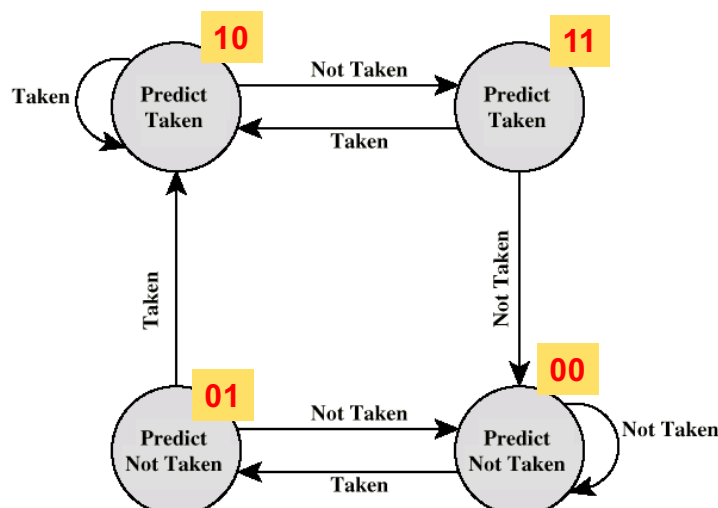
esempio:

```
.....  
LOOP: .....  
.....  
.....  
BNZ  LOOP
```

- dopo la prima esecuzione del ciclo, **in uscita dal ciclo**, il bit assegnato a BNZ LOOP è **0** perché il salto **non è stato preso**
- quando si rientra nello stesso ciclo,
 - si avrà **un errore alla prima iterazione** (il bit era a 0, invece prendo il salto)
 - le successive predizioni saranno giuste (l'entrata ha portato il bit a 1)
 - quando **si esce dal ciclo si fa un ulteriore errore** di predizione (e si rimette il bit a 0)

Predizione dinamica con 2 bit

- 2 bit per ricordare come è andata la predizione degli ultimi due salti
- per invertire la predizione ci vogliono **2 errori consecutivi**
- in questo modo a regime fa un solo errore per ciclo

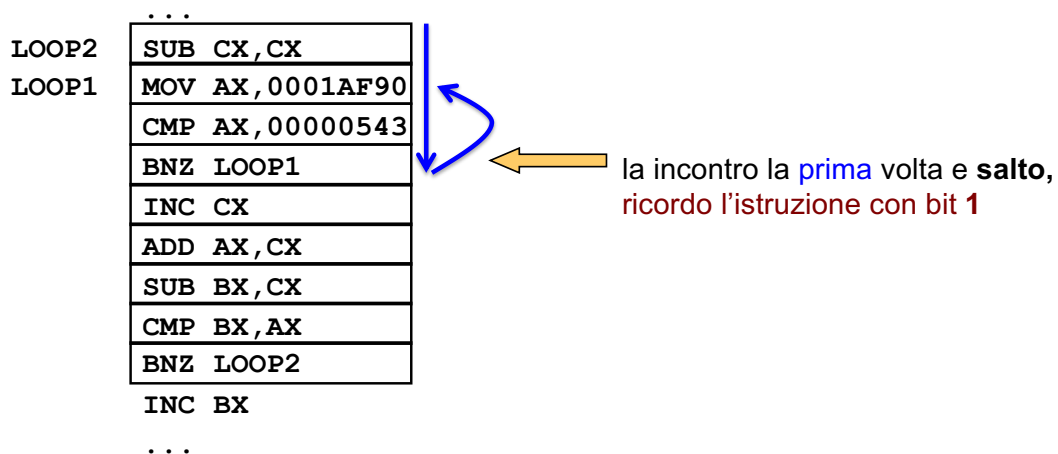


Predizione dinamica 1/2 bit

- per **ogni istruzione di salto** condizionato uso **1/2 bit**
 - per ricordare se l'ultima volta che ho eseguito *quella stessa istruzione* il salto è stato fatto o no
- **se incontro di nuovo** quell'istruzione e l'ultima volta **aveva provocato il salto**
 - **allora** predico che salterà, quindi **carico la pipeline** con le istruzioni **a partire dalla destinazione** del salto
 - se ho fatto la scelta sbagliata, le istruzioni caricate vengono eliminate

Predizione dinamica 1 bit

due cicli innestati, supponiamo che per entrambi si iteri una sola volta



Predizione dinamica 1 bit

due cicli innestati, supponiamo che per entrambi si iteri una sola volta

	...
LOOP2	SUB CX,CX
LOOP1	MOV AX,0001AF90
	CMP AX,00000543
0 1	BNZ LOOP1
	INC CX
	ADD AX,CX
	SUB BX,CX
	CMP BX,AX
	BNZ LOOP2
	INC BX
	...

- la incontro la **seconda** volta
- e **non salto**
 - la ricordavo con bit **1**
 - quindi **errore** e cambio bit a 0

errori totali = **1**

Predizione dinamica 1 bit

due cicli innestati, supponiamo che per entrambi si iteri una sola volta

	...
LOOP2	SUB CX,CX
LOOP1	MOV AX,0001AF90
	CMP AX,00000543
0	BNZ LOOP1
	INC CX
	ADD AX,CX
	SUB BX,CX
	CMP BX,AX
1	BNZ LOOP2
	INC BX
	...

- la incontro la **prima** volta e **salto**
ricordo questa istruzione con bit **1**

errori totali = **1**

Predizione dinamica 1 bit

due cicli innestati, supponiamo che per entrambi si iteri una sola volta

	...
LOOP2	SUB CX,CX
LOOP1	MOV AX,0001AF90
	CMP AX,00000543
1 0	BNZ LOOP1
	INC CX
	ADD AX,CX
	SUB BX,CX
	CMP BX,AX
1	BNZ LOOP2
	INC BX
	...

la incontro la **terza** volta

- e **salto**
- la ricordavo con bit **0**
- quindi **errore** e cambio bit a **1**

errori totali = **2**

Predizione dinamica 1 bit

due cicli innestati, supponiamo che per entrambi si iteri una sola volta

	...
LOOP2	SUB CX,CX
LOOP1	MOV AX,0001AF90
	CMP AX,00000543
0 1	BNZ LOOP1
	INC CX
	ADD AX,CX
	SUB BX,CX
	CMP BX,AX
1	BNZ LOOP2
	INC BX
	...

la incontro la **quarta** volta

- e **non salto**
- la ricordavo con bit **1**
- quindi **errore** e cambio bit **0**

errori totali = **3**

Predizione dinamica 1 bit

due cicli innestati, supponiamo che per entrambi si iteri una sola volta

...	
LOOP2	SUB CX,CX
LOOP1	MOV AX,0001AF90
	CMP AX,00000543
0	BNZ LOOP1
	INC CX
	ADD AX,CX
	SUB BX,CX
	CMP BX,AX
0 1	BNZ LOOP2
	INC BX
...	

la incontro la **seconda** volta

- e **non salto**
- la ricordavo con bit 1
- quindi **errore** e cambio bit a 0

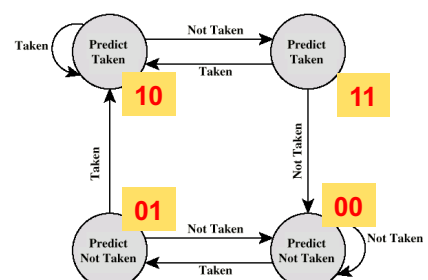
errori totali = **4**

Predizione dinamica 2 bit

errori totali = 0

...	
LOOP2	SUB CX,CX
LOOP1	MOV AX,0001AF90
	CMP AX,00000543
	BNZ LOOP1
	INC CX
	ADD AX,CX
	SUB BX,CX
	CMP BX,AX
	BNZ LOOP2
	INC BX
...	

la incontro la prima volta e **salto**
la ricordo con bit **10**

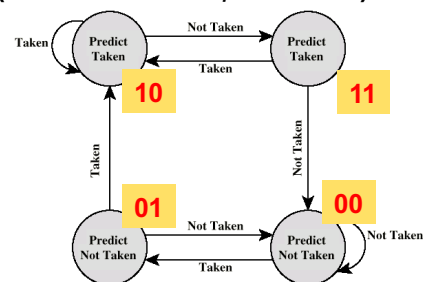


Predizione dinamica 2 bit

errori totali = 1

...	
LOOP2	SUB CX,CX
LOOP1	MOV AX,0001AF90
	CMP AX,00000543
	11 10 BNZ LOOP1
	INC CX
	ADD AX,CX
	SUB BX,CX
	CMP BX,AX
	BNZ LOOP2
	INC BX
...	

- la incontro la seconda volta
- predice salto e **non salto**
 - la ricordavo con bit **10**
 - **quindi errore e cambio in 11** (ma resta stessa predizione)

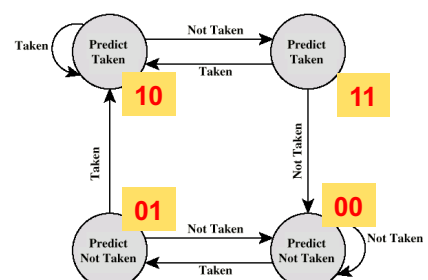


Predizione dinamica 2 bit

errori totali = 1

...	
LOOP2	SUB CX,CX
LOOP1	MOV AX,0001AF90
	CMP AX,00000543
	11 BNZ LOOP1
	INC CX
	ADD AX,CX
	SUB BX,CX
	CMP BX,AX
	10 BNZ LOOP2
	INC BX
...	

- la incontro la prima volta e **salto**
la ricordo con bit **10**



Predizione dinamica 2 bit

errori totali = 1 non è errore in più

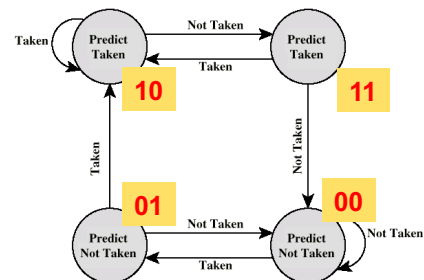
...	
LOOP2	SUB CX,CX
LOOP1	MOV AX,0001AF90
	CMP AX,00000543
	BNZ LOOP1
	INC CX
	ADD AX,CX
	SUB BX,CX
	CMP BX,AX
	BNZ LOOP2
	INC BX
...	

10 11

10

la incontro la terza volta

- predice salto e **salto**
- la ricordavo con bit 11
- **quindi non errore di predizione**
- **ma cambio bit a 10**



Predizione dinamica 2 bit

errori totali = 2

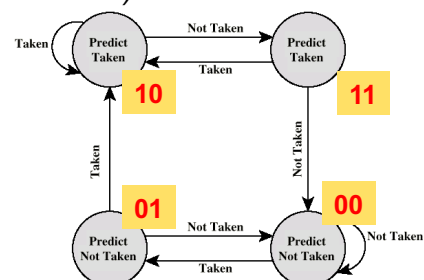
...	
LOOP2	SUB CX,CX
LOOP1	MOV AX,0001AF90
	CMP AX,00000543
	BNZ LOOP1
	INC CX
	ADD AX,CX
	SUB BX,CX
	CMP BX,AX
	BNZ LOOP2
	INC BX
...	

11 10

10

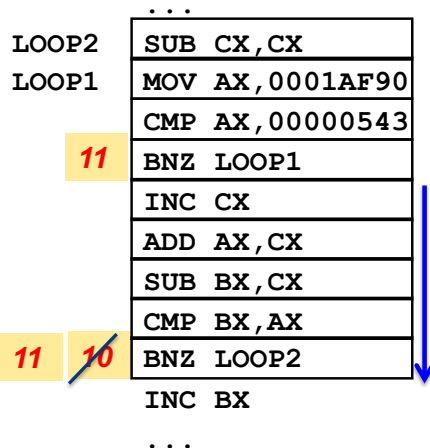
la incontro la quarta volta

- predice salto e **non salto**
- la ricordavo con bit 10
- **quindi errore di predizione e cambio bit a 11 (ma stessa predizione)**

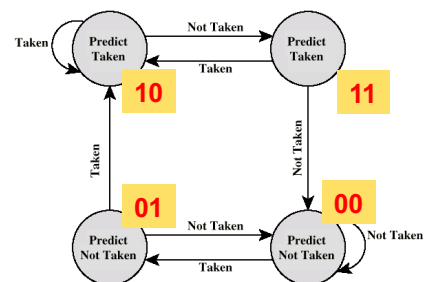


Predizione dinamica 2 bit

errori totali = 3



- la incontro la seconda volta
- predice salto e **non salto**
 - la ricordavo con bit 10
 - **quindi errore di predizione e cambio bit a 11**



Predizione dinamica

buffer di predizione dei salti

(branch prediction buffer o *branch history table*)

- piccola memoria associata allo stadio fetch della pipeline
- ogni riga della tabella è costituita da 3 elementi:
 1. indirizzo istruzione salto,
 2. i bit di predizione
 3. l'indirizzo destinazione del salto (o l'istruzione destinazione stessa), così quando la predizione è di saltare non devo attendere che si ri-decodifichi il target del salto (se la **previsione è errata** dovrò eliminare le istruzioni errate e caricare quelle corrette)

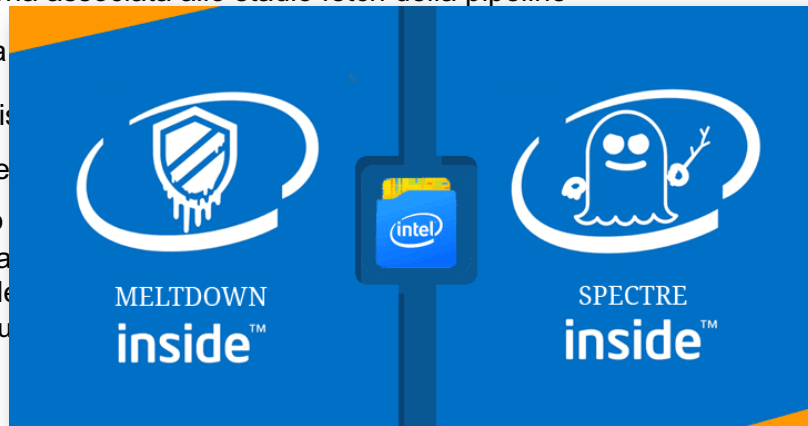
Predizione dinamica

buffer di predizione dei salti

(*branch prediction buffer* o *branch history table*)

- piccola memoria associata allo stadio fetch della pipeline
- ogni riga della

1. indirizzo is
2. i bit di pre
3. l'indirizzo
quando la
il target de
caricare qu



Soluzioni per salti condizionati

1. flussi multipli (*multiple streams*)
2. prefetch dell'istruzione target
3. buffer circolare (*loop buffer*)
4. predizione dei salti
5. **salto ritardato** (*delayed branch*)
 - **Finché non si sa** se ci sarà o no il salto (l'istruzione è in pipeline), invece di restare in stallo si può **eseguire un'istruzione che non dipende dal salto**
 - istruzione successiva al salto: **branch delay slot**
 - Il **compilatore** cerca di allocare nel *branch delay slot* una istruzione "**opportuna**" (magari inutile ma non dannosa)
 - la CPU esegue **sempre** l'istruzione del *branch delay slot* e solo dopo altera, se necessario, la sequenza di esecuzione delle istruzioni

Salto ritardato (delayed branch)

codice scritto dal programmatore

istruzione indipendente dalle altre	→	MUL R3,R4	$R3 \leftarrow R3 * R4$
		SUB #1,R2	$R2 \leftarrow R2 - 1$
		ADD R1,R2	$R1 \leftarrow R1 + R2$
		BEZ TAR	branch if zero
istruzione eseguita solo se non si salta	→	MOVE #10,R1	$R1 \leftarrow 10$

	TAR	-----	

codice ottimizzato dal compilatore

SUB #1,R2
ADD R1,R2
BEZ TAR
MUL R3,R4
MOVE #10,R1

TAR -----

istruzione **eseguita in ogni caso**:
si trova nel *branch delay slot* !!

istruzione eseguita
solo se **non** si salta

senza ottimizzazione

senza ottimizzazione

qui si conosce
condizione e indirizzo del salto

	1	2	3	4	5	6	7	8	9	10	11	12	13
MUL R3, R4	FI	DI	CO	FO	EI	WO							
SUB #1 R2		FI	DI	CO	FO	EI	WO						
ADD R1, R2			FI	DI	CO	FO	EI	WO					
BEZ TAR				FI	DI	CO	FO	EI	WO				
MOVE #10, R1					FI								
...													
TAR ...													

qui si sa che è un salto condizionato
quindi inserisco bubble finché non si conosce la condizione

senza ottimizzazione

qui si conosce
condizione e indirizzo del salto

[illegible]

- se **non si salta**
 - continuo con MOVE
 - termino in 12 con **2** cicli di stallo
- se **si salta**
 - scarto MOVE e inizio con TAR
 - termino in 13 con **3** cicli persi (uno inutile + 2 stalli)

con delayed branch

qui si conosce
condizione e indirizzo del salto

[illegible]

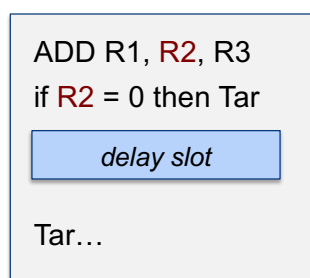
con delayed branch

qui si conosce
condizione e indirizzo del salto

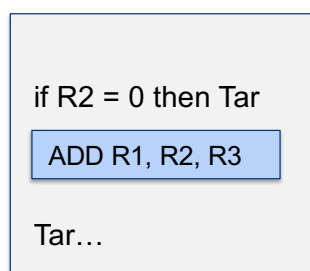
	1	2	3	4	5	6	7	8	9	10	11	12
SUB #1, R2	FI	DI	CO	FO	EI	WO						
ADD R1, R2		FI	DI	CO	FO	EI	WO					
BEZ TAR			FI	DI	CO	FO	EI	WO				
MUL R3,R4				FI	DI	CO	FO	EI	WO			
MOVE #10, R1					FI			DI	CO	FO	EI	WO
...												
TAR ...							FI	DI	CO	FO	EI	WO

- se **non si salta**
 - continuo con MOVE
 - termino in 11 con **1** ciclo di stallo
- se **si salta**
 - scarto MOVE e inizio con TAR
 - termino in 12 con **2** cicli persi (uno inutile e 1 stallo)

Salto ritardato (delayed branch)



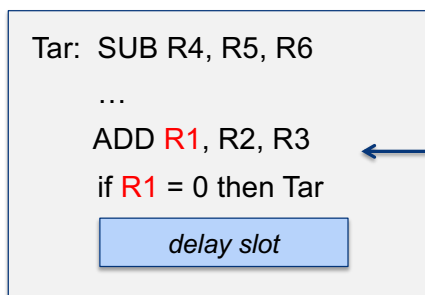
ottimizzazione



“from before”

quando è possibile riempie il
branch delay slot con
un'istruzione **indipendente**
proveniente dalla parte di codice che
precede il salto

Salto ritardato (delayed branch)

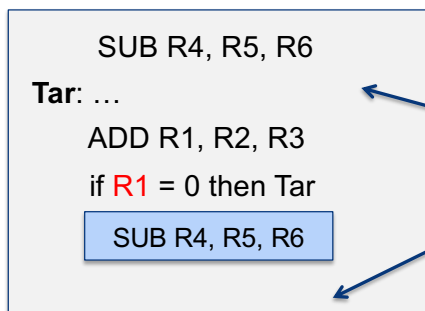


“from target”
utile quando è probabile che il salto sia preso (es. in loop)

non sono più istruzioni indipendenti

riempie il branch delay slot con l'istruzione target del salto, che normalmente viene copiata perché potrebbe essere accessibile anche tramite un altro cammino

ottimizzazione



eseguo **sempre** l'istruzione nel delay slot, quindi:

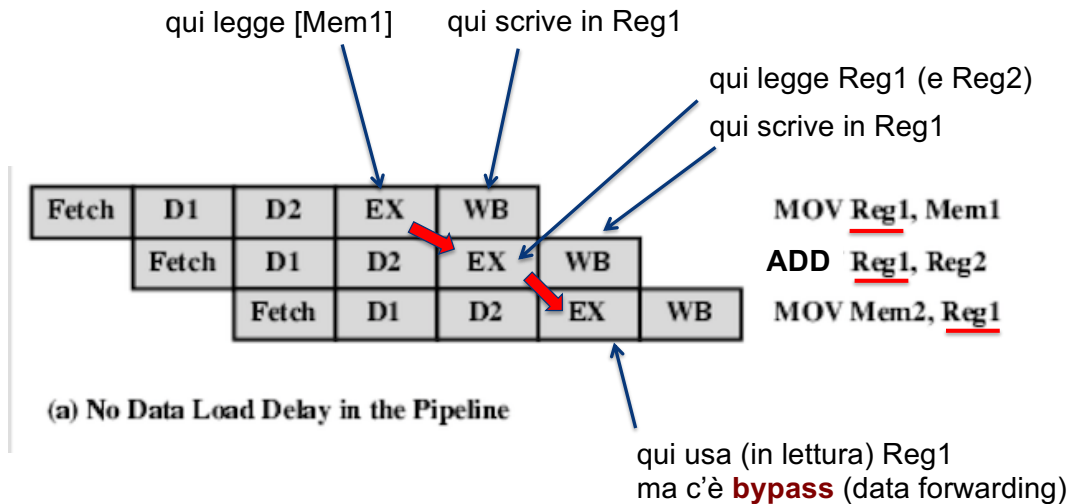
- quando salto vado alla successiva
- quando non salto procedo oltre, ma è **corretto solo** se l'istruzione nel delay slot è inutile ma **non dannosa** (es. R4 non usato se non salta)

Intel 80486 Pipelining

- Fetch
 - Istruzioni prelevate dalla cache o memoria esterna
 - Poste in uno dei due buffer di prefetch da 16 byte
 - Carica dati nuovi appena quelli vecchi sono “consumati”
 - Poiché le istruzioni sono a lunghezza variabile (1-11 byte), in media carica 5 istruzioni per ogni caricamento da 16 byte
 - Indipendente dagli altri stadi per mantenere i buffer pieni
- Decodifica 1 (D1)
 - Decodifica codice operativo e modi di indirizzamento
 - Le informazioni di sopra sono codificate (al più) nei primi 3 byte di ogni istruzione
 - Se necessario, indica allo stadio D2 di trattare i byte restanti (dati immediati e spiazamento)
- Decodifica 2 (D2)
 - Espande i codici operativi in segnali di controllo per l'ALU
 - Calcola gli indirizzi in memoria per i modi di indirizzamento più complessi
- Esecuzione (EX)
 - Operazioni ALU, accesso alla cache (memoria).
- Retroscrittura (WB)
 - Se richiesto, aggiorna i registri e i flag di stato modificati in EX
 - Se l'istruzione corrente aggiorna la memoria, pone il valore calcolato in cache e nei buffer di scrittura del bus

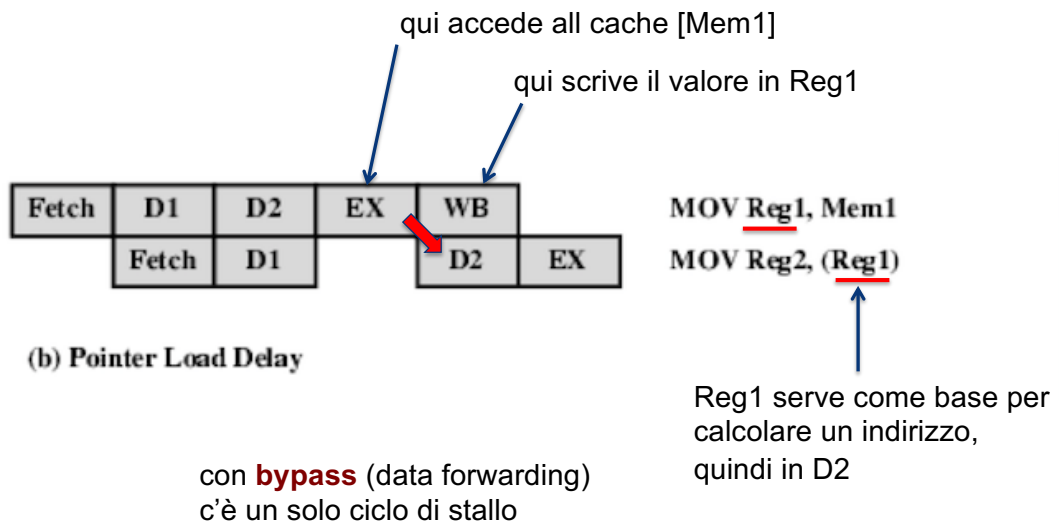
80486 Instruction Pipeline: esempi

accessi consecutivi allo stesso dato non introducono ritardi



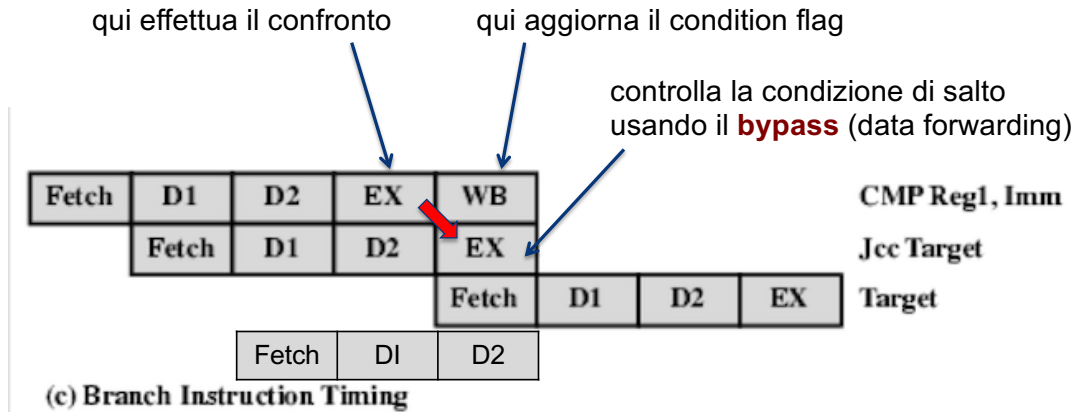
80486 Instruction Pipeline: esempi

ritardo per valori usati per calcolare un indirizzo



80486 Instruction Pipeline: esempi

salto condizionato. Assumiamo venga eseguito



in parallelo fa uno “**speculative fetch**” dell’istruzione target
(in aggiunta a quello già iniziato per l’istruzione sequenziale, che sarà scartata).