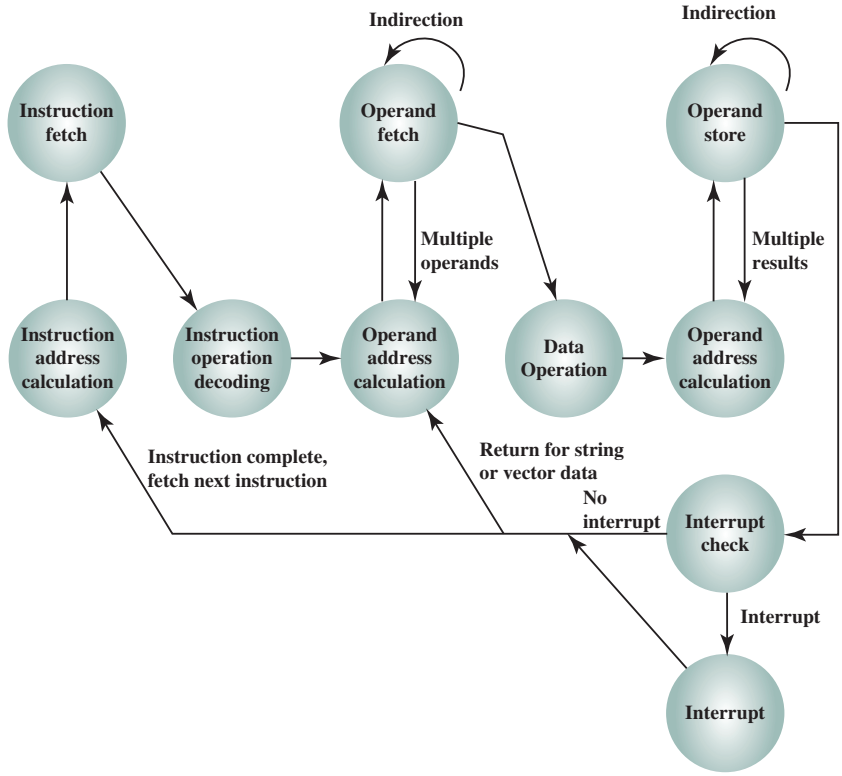
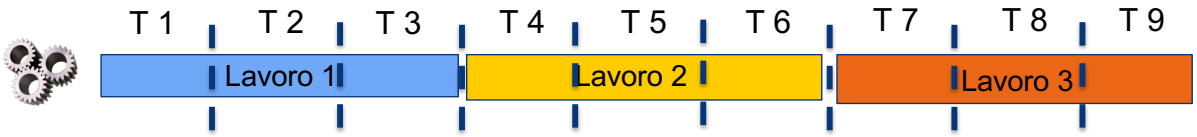
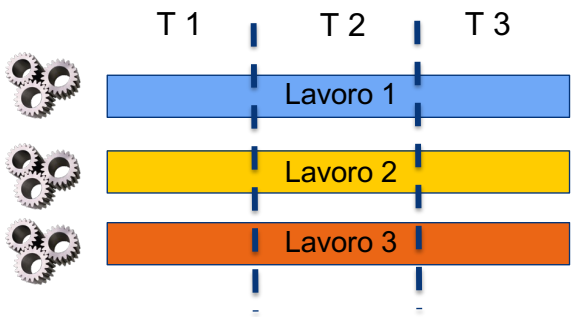
Fetch/Execute: ancora più in dettaglio



Parallelismo

- eseguire più attività **contemporanemante**
- 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

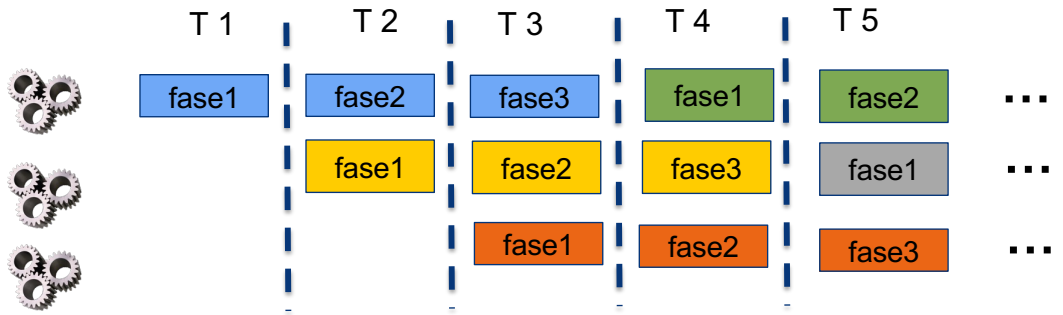
se c'è ***dipendenza funzionale***
tra un lavoro e il successivo?

Parallelismo

L 1
L 2
L 3
L4
L 5

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:

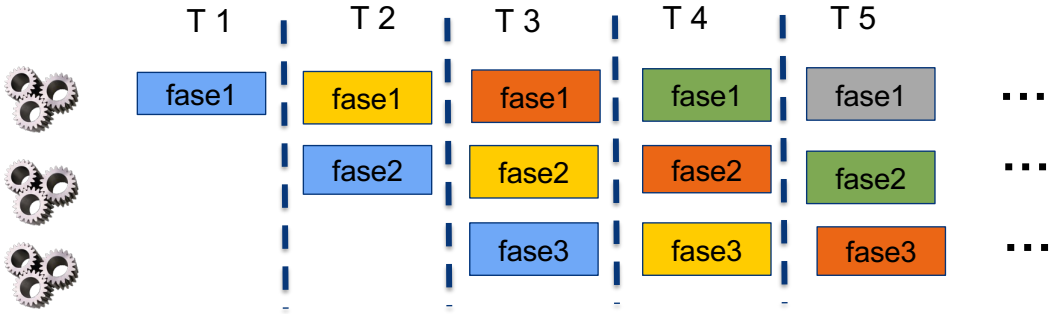
- ognuno esegue un lavoro completo
- **a regime** ha lo stesso throughput del parallelismo totale
- ognuno ha le risorse necessarie per ogni fase: *sistema totalmente replicato*

Parallelismo

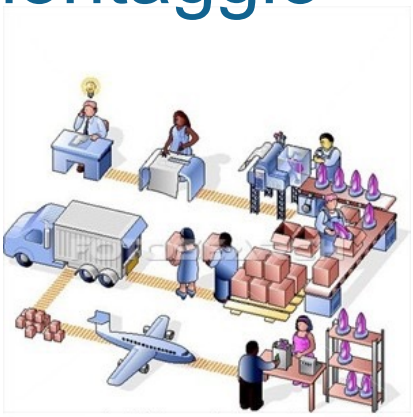
L 1
L 2
L 3
L4
L 5

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

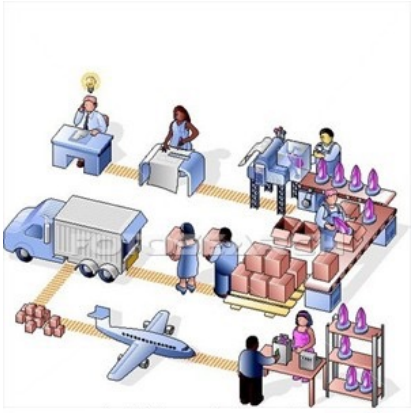


Pipeline – catena di montaggio

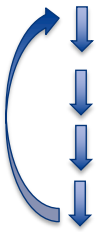


- si decompone un lavoro in fasi successive
  - un prodotto deve passare **una fase dopo l'altra**
  - ogni fase è realizzata da un ***diverso*** **operatore**
  - **nello stesso istante**
    - prodotti diversi sono in fasi diverse (**parallelismo**)
  - **l'istante successivo**
    - ogni fase ripete 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

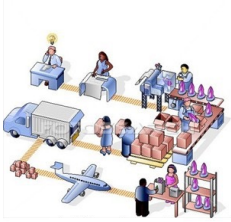


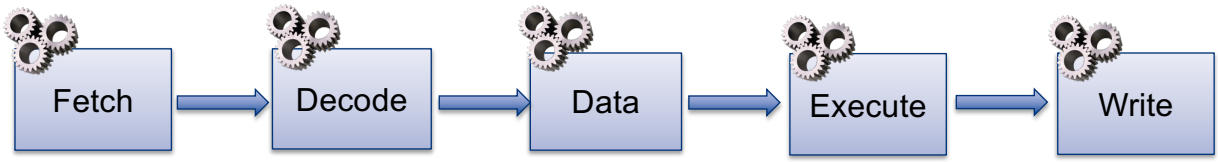
diverse **fasi del ciclo esecutivo** di un' istruzione



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

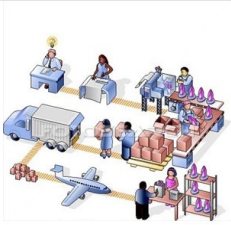
Pipeline

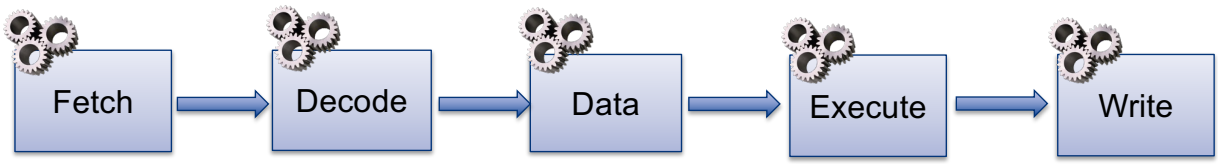




- nello stesso istante:
  - ***istruzioni*** diverse sono in fasi diverse
- l'istante successivo
  - ogni fase ripete lo stesso lavoro ***sull'istruzione*** successiva
  - ogni ***istruzione*** avanza alla fase successiva

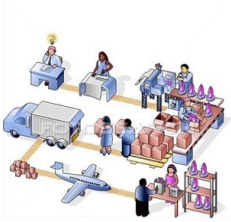
Pipeline

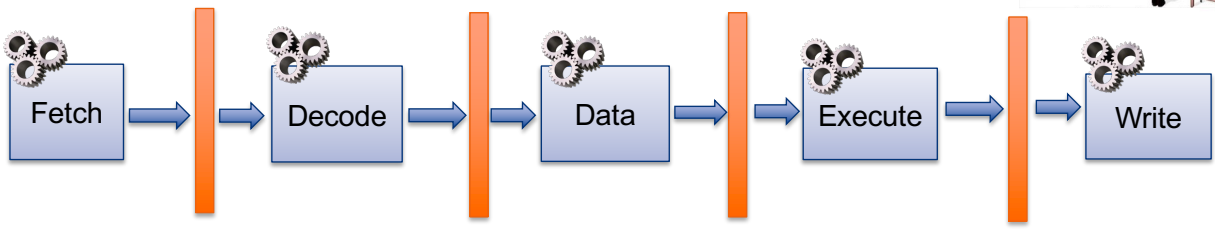




- ogni fase è realizzata da una ***diversa unità funzionale della CPU***
- operatori/fasi diverse usano risorse diverse evitando conflitti (se possibile)

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?

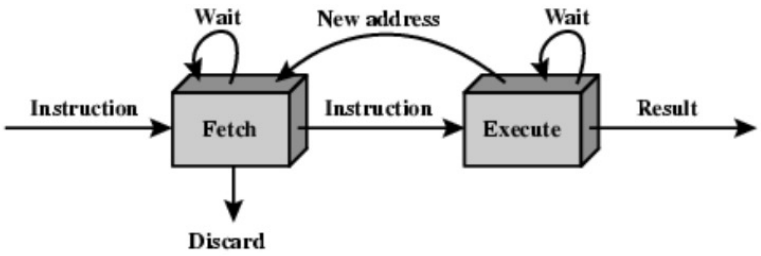
il prefetch non raddoppia le prestazioni:

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

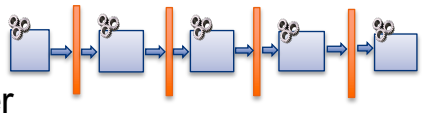
Miglioramento delle prestazioni?

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, perche' 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 pipline

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**

- **fetch** **(FI)** lettura dell'istruzione
- **decodifica** **(DI)** decodifica dell'istruzione

Fetch Data

- **esecuzione** **(EI)** esecuzione dell'istruzione
- **scrittura** **(WO)** scrittura del risultato in memoria

Pipeline – evoluzione ideale

Per aumentare le prestazioni bisogna

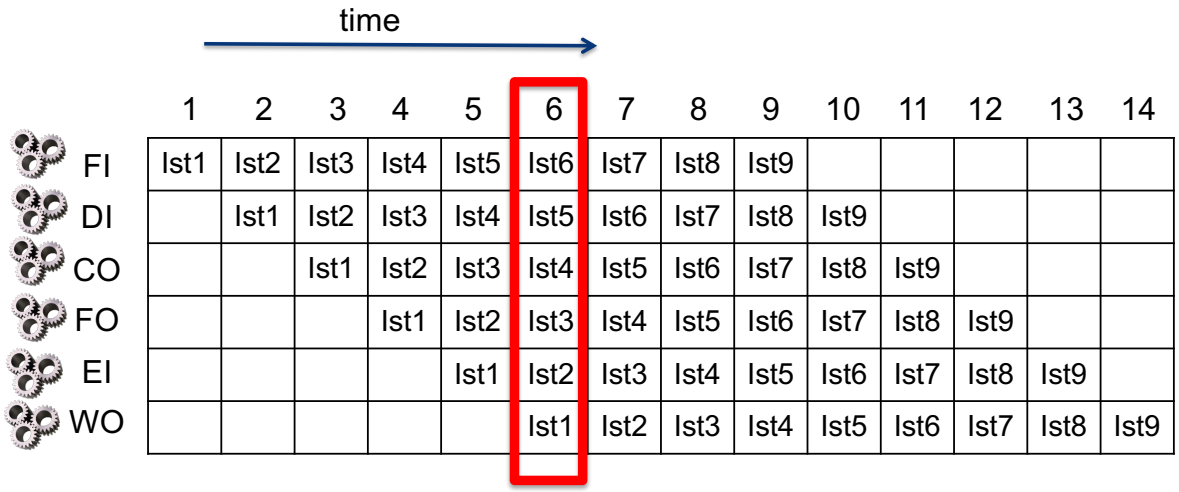
- decomporre il lavoro in un **maggior numero di fasi**
- cercare di rendere le fasi più **indipendenti** e con una **durata simile**

- **fetch** **(FI)** lettura dell'istruzione
- **decodifica** **(DI)** decodifica dell'istruzione
- **calcolo ind. op.** **(CO)** calcolo indirizzo effettivo operandi

Fetch Data

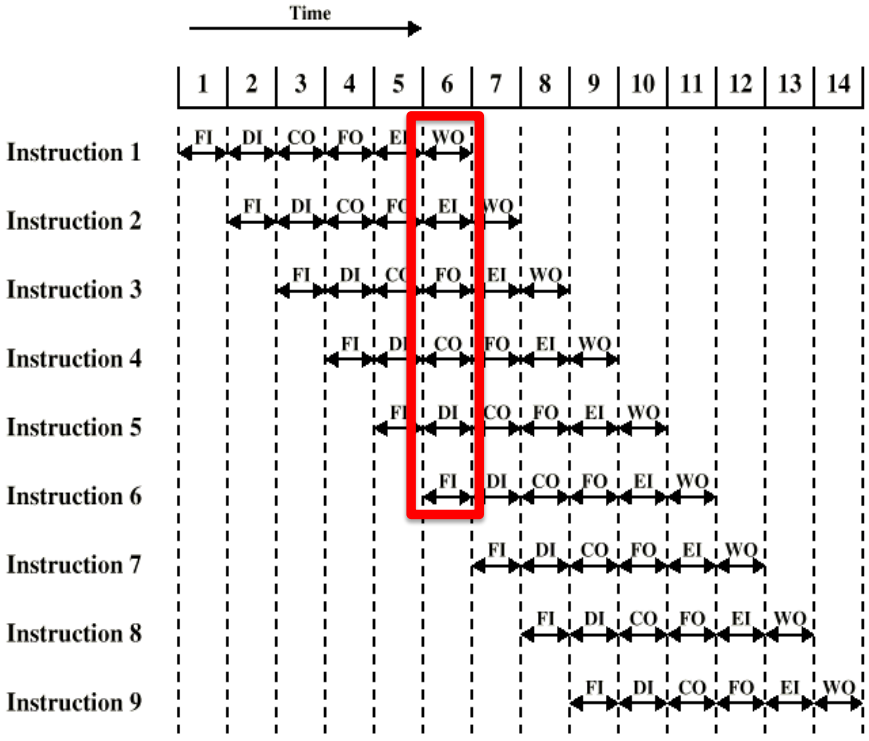
- **fetch operandi** **(FO)** lettura degli operandi in memoria
- **esecuzione** **(EI)** esecuzione dell'istruzione
- **scrittura** **(WO)** scrittura del risultato in memoria

Pipeline – evoluzione ideale



esegue 9 istruzioni
in 14 unità di tempo
invece di 9x6=54

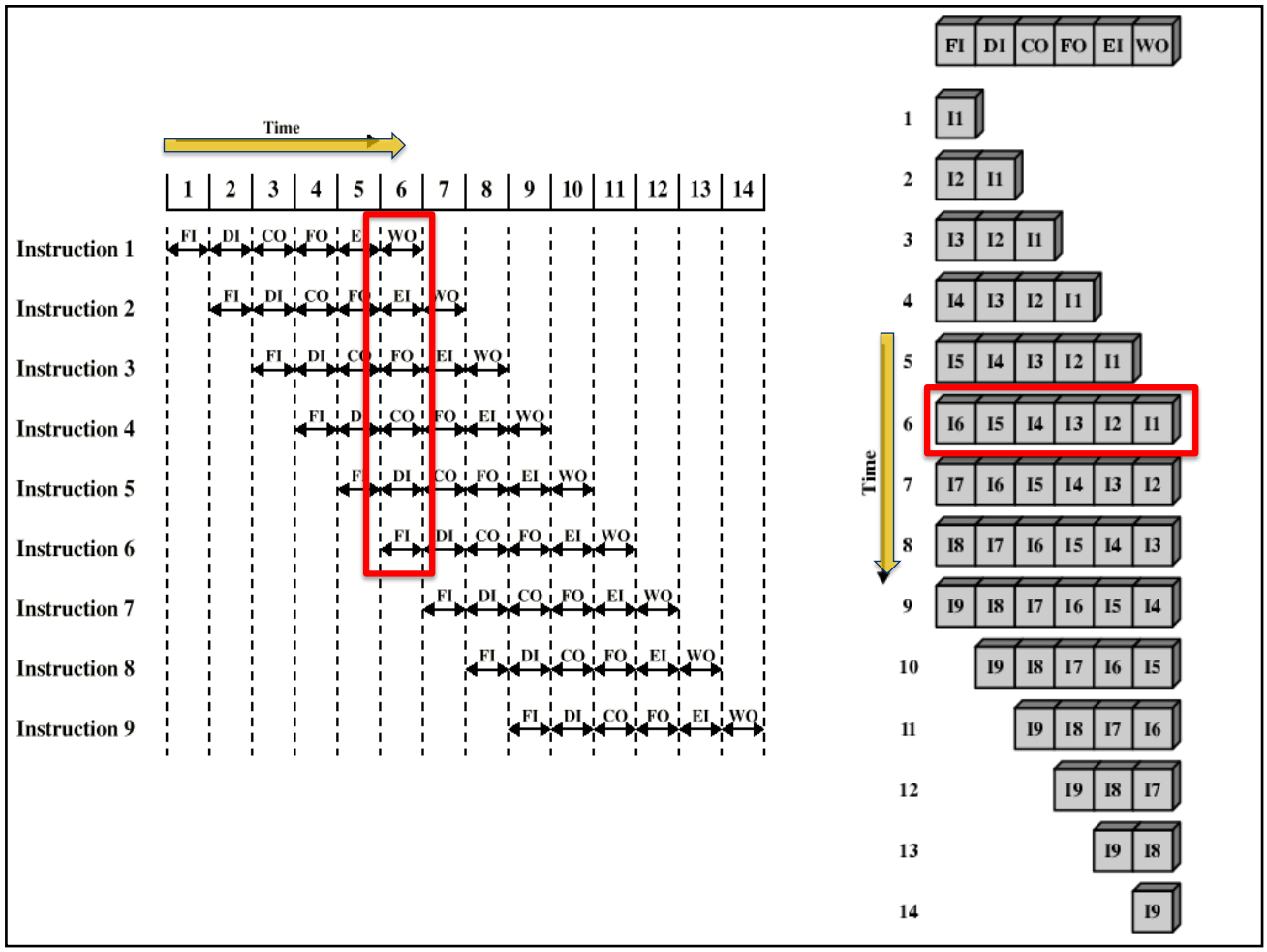
Pipeline – evoluzione ideale



esegue 9 istruzioni
in 14 unità di tempo
invece di 9x6=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:

$$\tau = \max_i [\tau_i] + d = \tau_m + d \qquad 1 \le i \le k$$

k: numero di stadi nella pipeline

$\max_i [\tau_i]$: massimo ritardo di stadio (ritardo dello stadio più oneroso)

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

$\tau_m >> d$

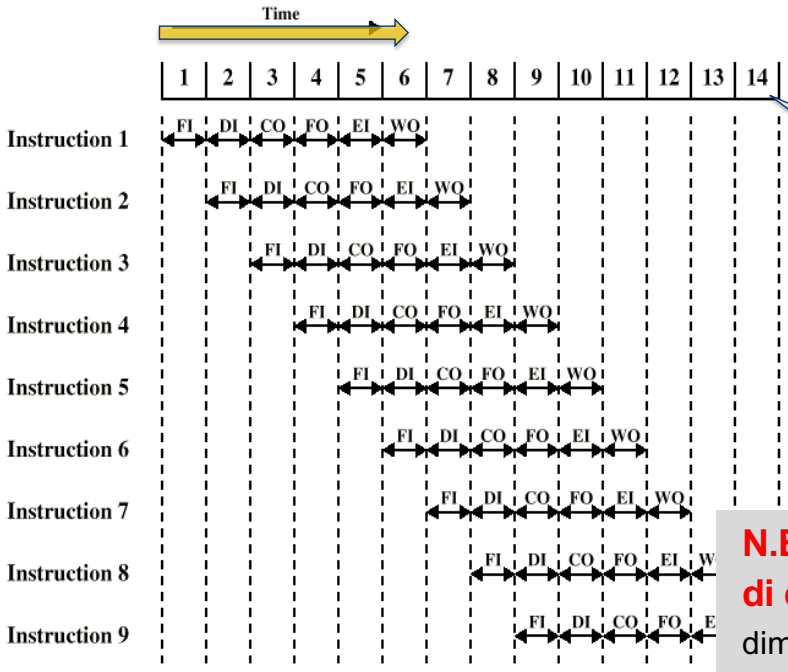
Pipeline performance ideali

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

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

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)



9 istruzioni
ognuna in 6 unità di tempo

14 = [6+ (9-1)]

invece di 6 * 9=54

N.B. non diminuisce il tempo di esecuzione di 1 istruzione!
diminuisce il tempo di esecuzione di **n** istruzioni:

aumenta il troughput

Pipeline performance ideali

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

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

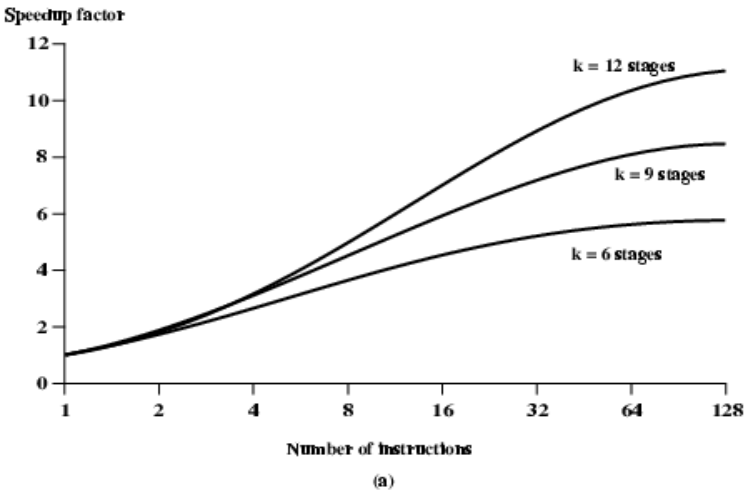
Speedup (fattore di velocizzazione)

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

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

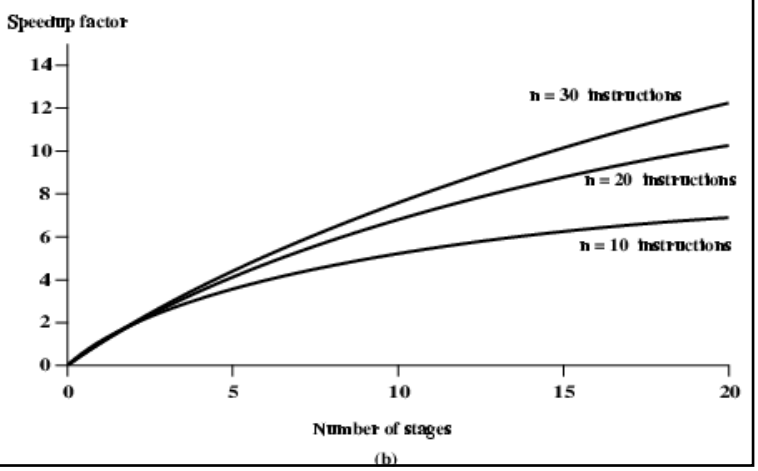
Speedup

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

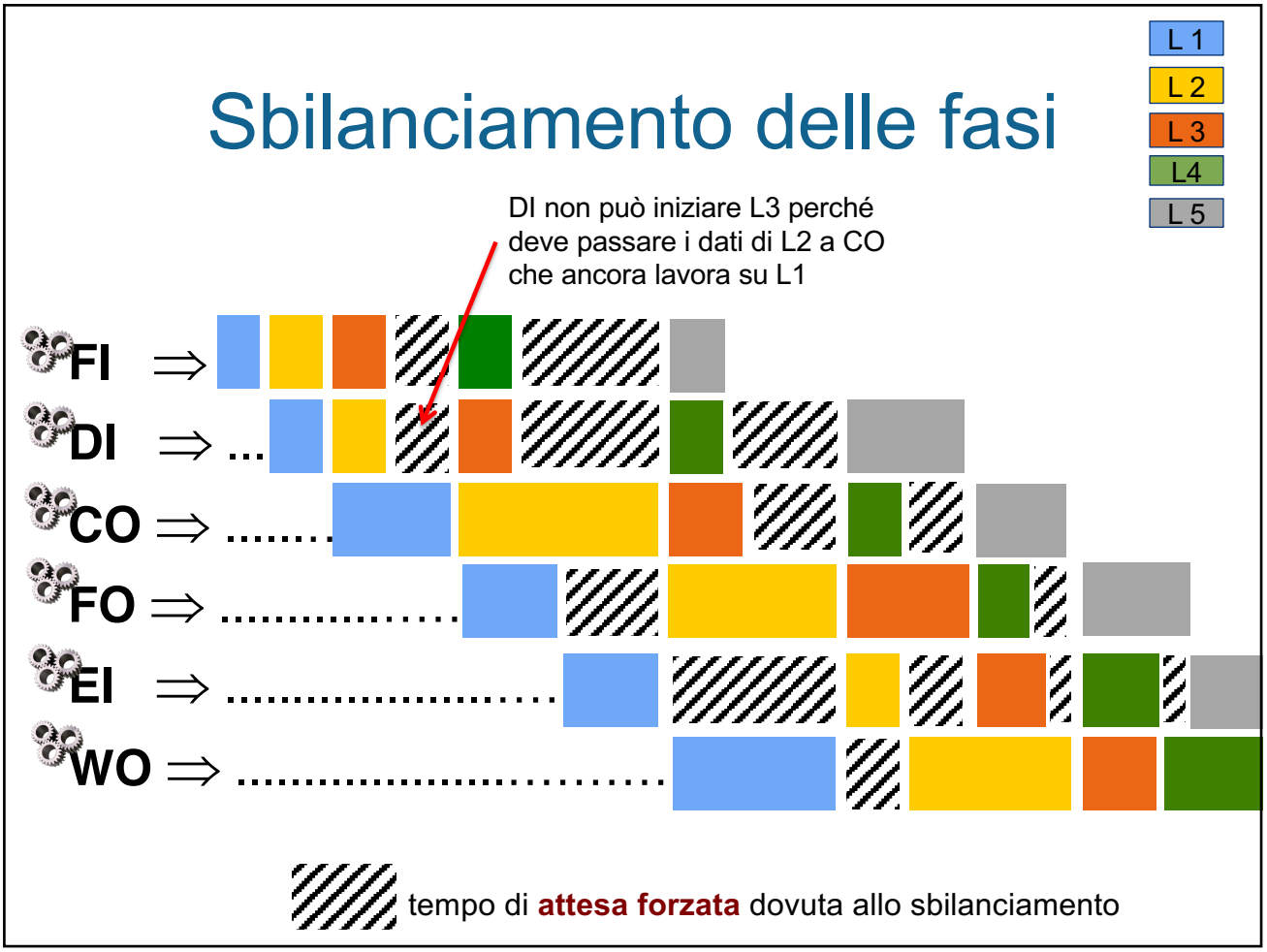
1. **sbilanciamento delle fasi**
   - durate diverse per fase e per istruzione
2. problemi **strutturali** (*structural hazards*)
   - due fasi competono per usare la stessa risorsa, es. memoria in FI, FO, WO
3. dipendenza dai **dati** (*data hazards*)
   - un'istruzione dipende dal risultato di un'istruzione 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

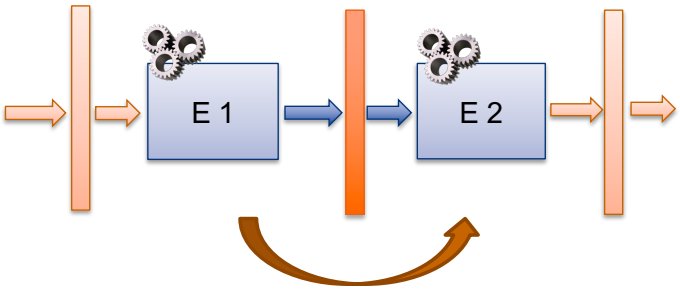
- Non tutte le fasi richiedono lo stesso tempo di esecuzione

  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

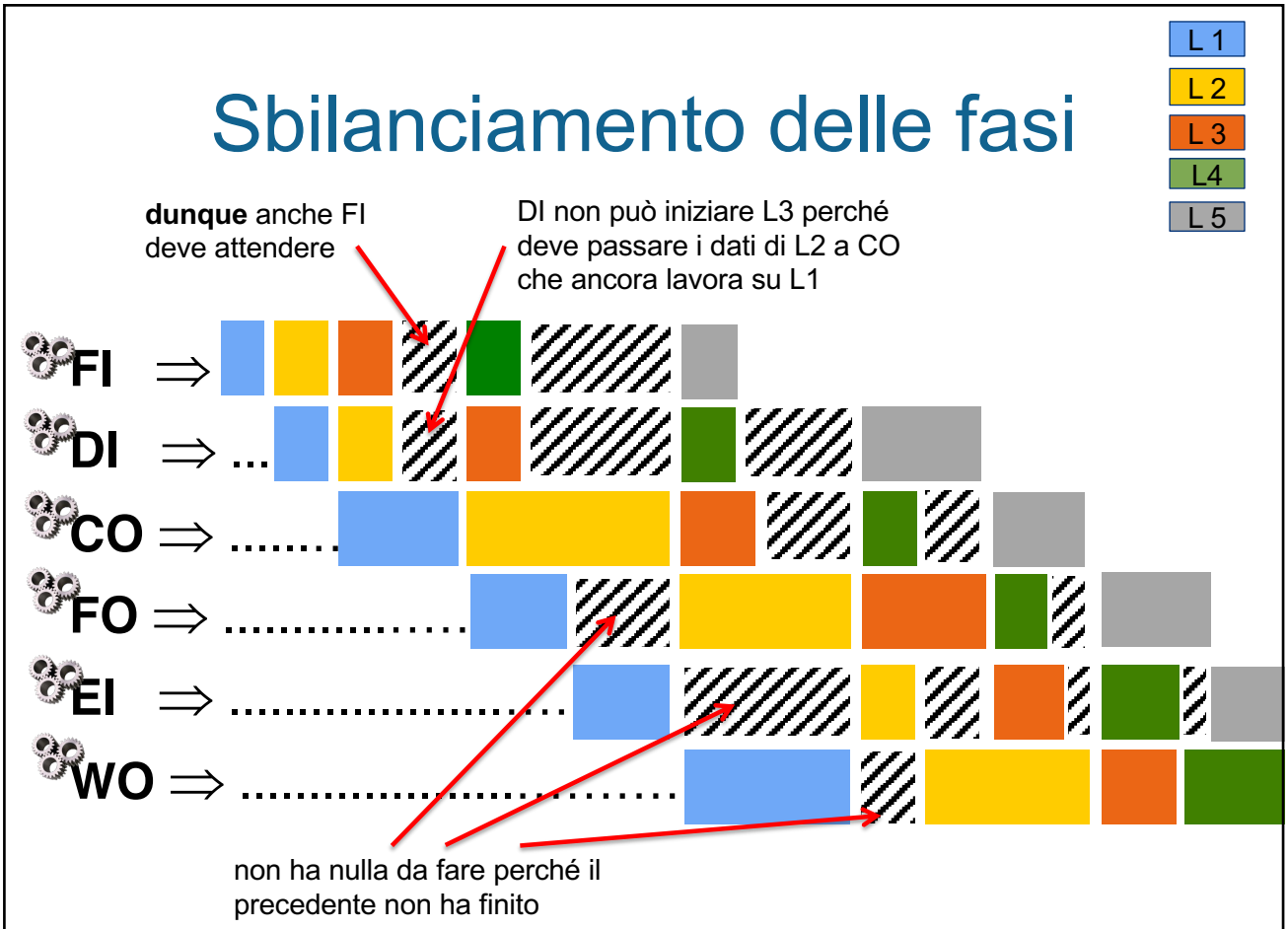


passare i dati significa che

- l'esecutore E1 mentre lavora **scrive** sul registro intermendio
- 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

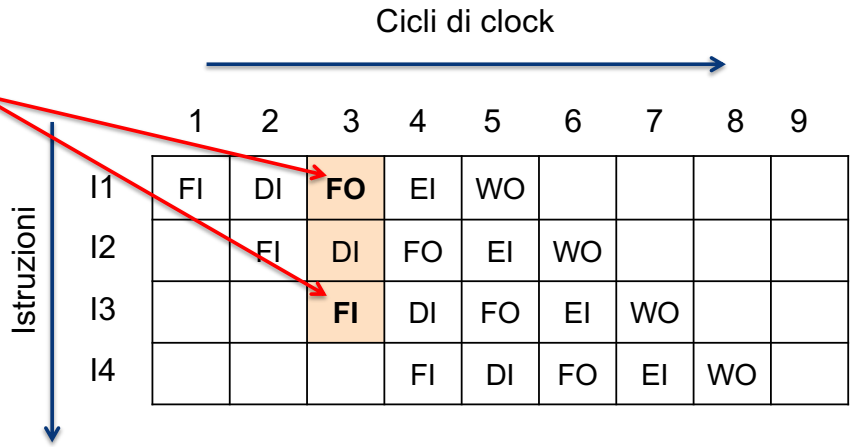
Possibili soluzioni:

- 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**

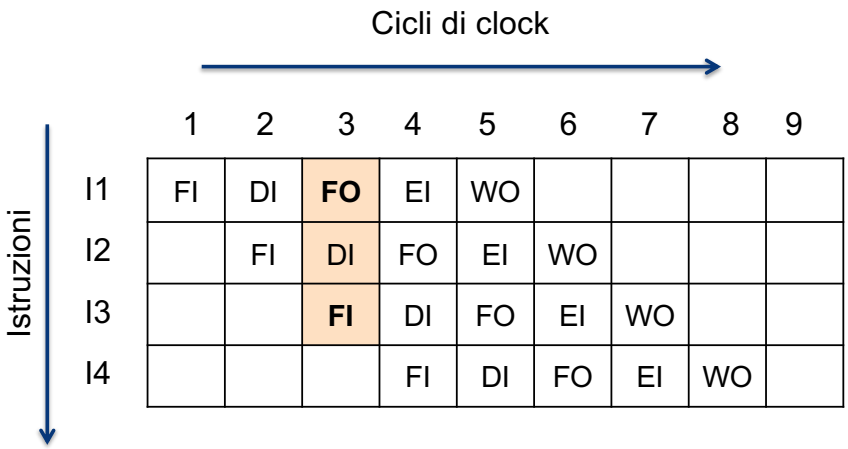
Problemi strutturali

- 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

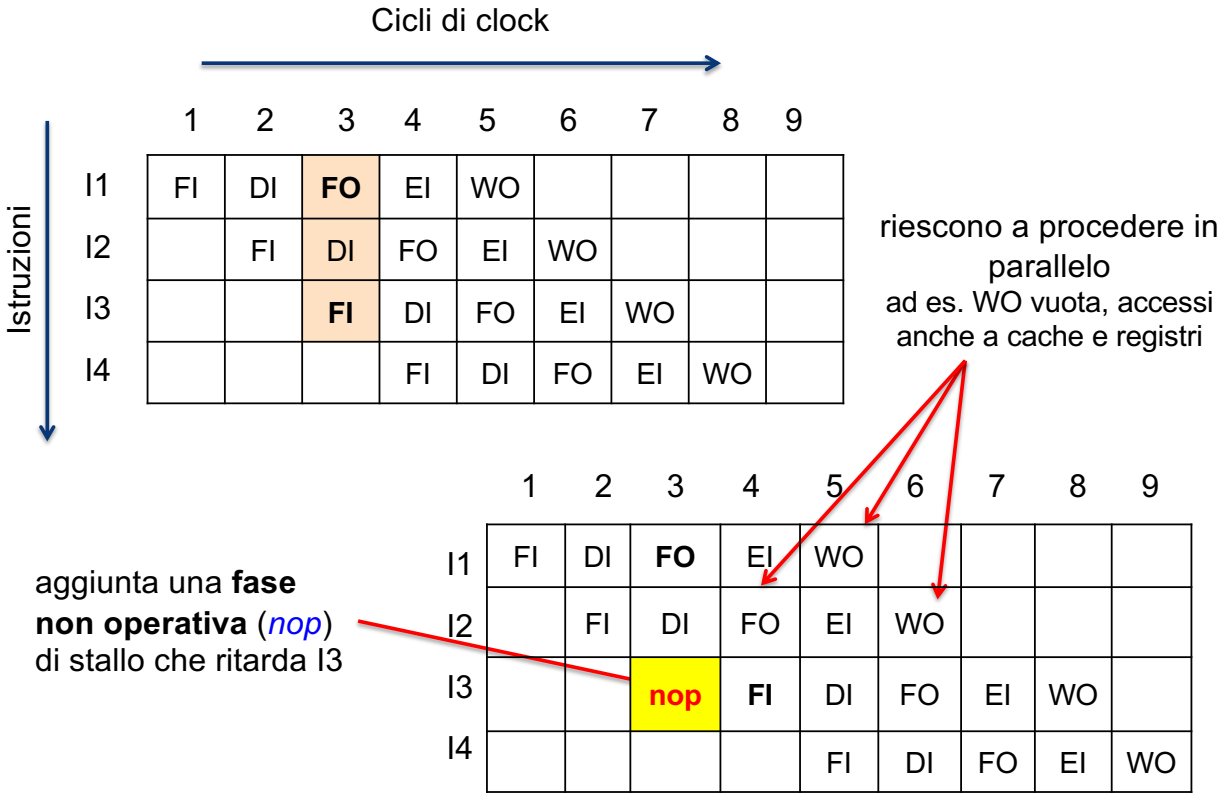


Problemi strutturali



aggiunta una **fase non operativa** (*nop*) di stallo che ritarda I3

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

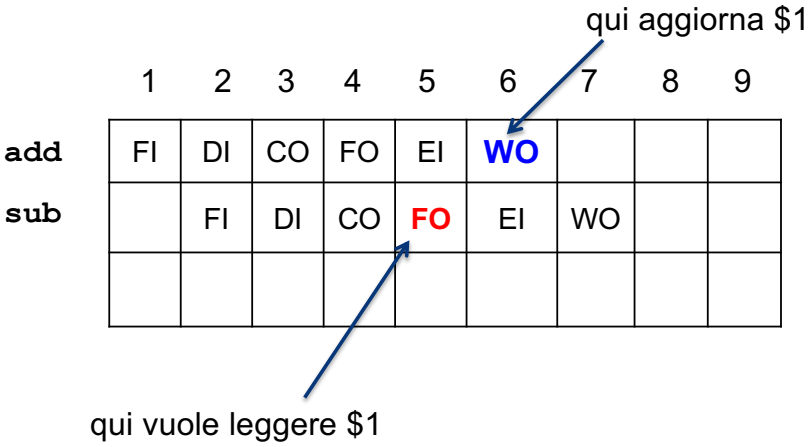
- un dato modificato nell'esecuzione dell'istruzione ***corrente*** può dover essere utilizzato dalla fase **FO** dell'istruzione ***successiva***

```
add $1, $2, $3      R1 <- [R2] + [R3]
sub $4, $1, $5      R4 <- [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 <- [R2] + [R3]
sub $4, $1, $5      R4 <- [R1] - [R5]
```



Dipendenza dai dati

```
add $1, $2, $3
sub $4, $1, $5
```

R1 <– [R2] + [R3]

R4 <– [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		

due cicli di stallo

Dipendenza dai dati

```
add $1, $2, $3
sub $4, $1, $5
```

R1 <– [R2] + [R3]

R4 <– [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	W O		
Istr3			FI	DI			CO	FO	EI	WO	
Istr4				FI			DI	CO	FO	EI	

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

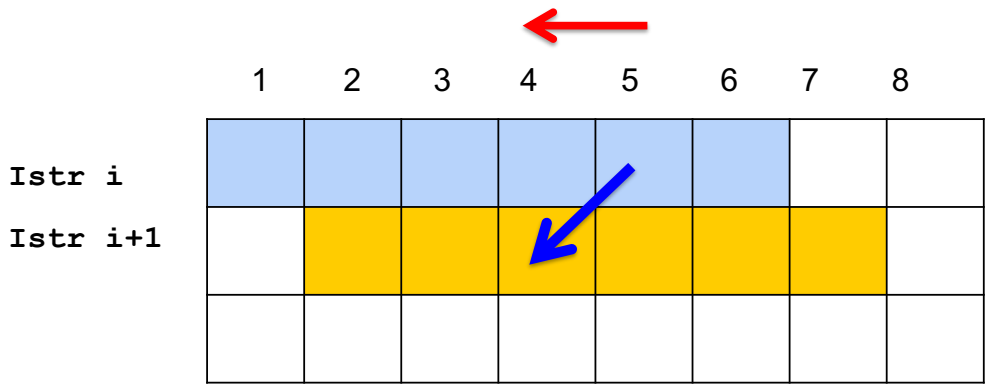
Data hazards

```
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

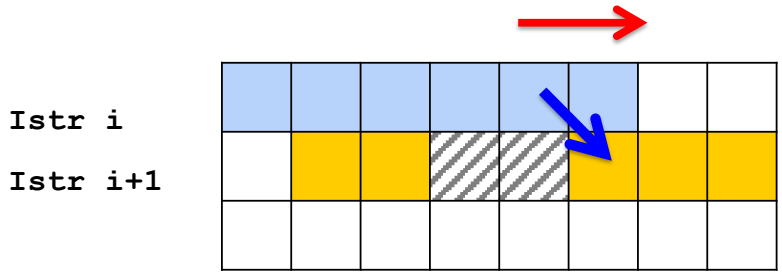
l’istruzione successiva ha bisogno dei dati **prima** che la precedente li abbia prodotti



- 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

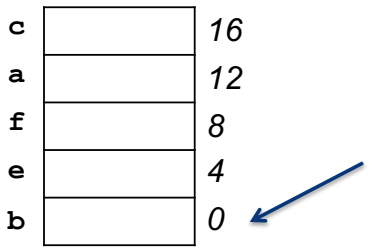
1. Introduzione di fasi non operative (**nop**-stallo)
2. propagazione in avanti del dato richiesto (**data forwarding**)
   - dipende da architettura di pipeline e implementazione istruzioni
3. **riordino delle istruzioni**

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)



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
```

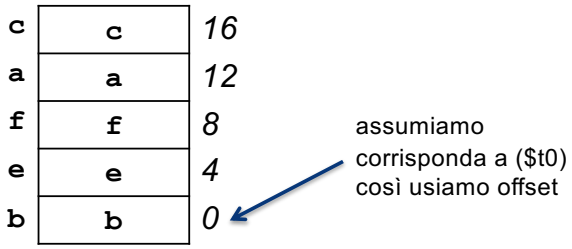
```
lw  $1  0  ($t0)
lw  $2  4  ($t0)
add $3  $1  $2
sw  $3  12 ($t0)
lw  $4  8  ($t0)
add $5  $1  $4
sw  $5  16 ($t0)
```

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)




```
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 definite la pipeline, qualche problema può risoversi 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`

```
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)
```

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