

## Prefetch

- La fase di prelievo della istruzione accede alla memoria principale
- La fase di esecuzione di solito non accede alla memoria principale
- Si può prelevare l'istruzione successiva durante l'esecuzione della istruzione corrente
- Questa operazione si chiama "instruction prefetch"

## Prefetch



(a) Simplified view



## Miglioramento delle prestazioni

- Il prefetch non raddoppia le prestazioni:
  - L'esecuzione di istruzioni jump o branch possono rendere vano il prefetch (**perché?**)
  - La fase di prelievo è tipicamente più breve della fase di esecuzione
    - Prefetch di più istruzioni ?
- Aggiungere più fasi per migliorare le prestazioni

## Il prefetch può essere inutile perché...

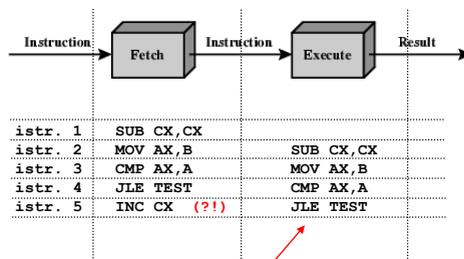


Per il seguente costrutto

```
if (A > B) then
```

un compilatore potrebbe generare il seguente codice 80x86

```
SUB CX,CX; CX ← 0
MOV AX,B ; AX ← mem[B]
CMP AX,A ; paragona [AX] con mem[A]
JLE TEST ; salta se A ≤ B
INC CX ; CX ← CX+1
TEST JCXZ OUT ; salta se [CX] è 0
THEN
OUT
```



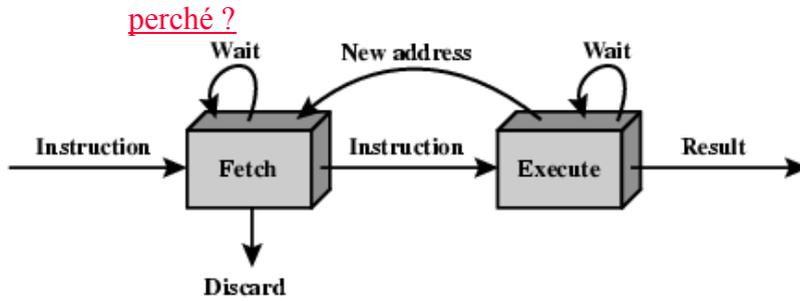
il controllo potrebbe passare alla istruzione con etichetta `TEST` e non alla istruzione successiva !

... si deve caricare una istruzione diversa dalla successiva !

# Prefetch



(a) Simplified view



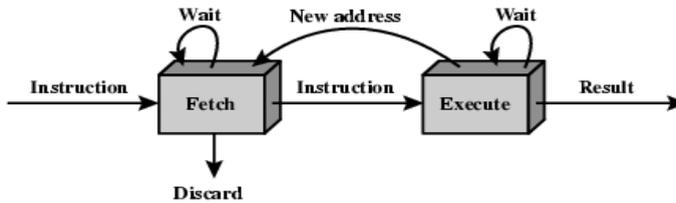
(b) Expanded view

## perché...



su processore 286

	istruzione	cicli di clock impiegati dall'istr.
istr. 1	SUB CX, CX	2
istr. 2	MOV AX, B	5
istr. 3	CMP AX, A	6
istr. 4	JLE TEST	3 se non salta, >7 altrimenti
istr. 5	INC CX	2
istr. 6	JCZX	4 se non salta, >8 altrimenti



(b) Expanded view

## Evoluzione delle architetture

### *Evoluzione strutturale*

- **Parallelismo**

- Se un lavoro non può essere svolto più velocemente da una sola persona (unità), allora conviene **decomporlo** in parti che possano essere eseguite da più persone (unità) **contemporaneamente**



- **Catena di montaggio**



## Pipeline

### *Generalità 1*



- Ipotizziamo che per svolgere un dato lavoro **L** si debbano eseguire tre fasi distinte e sequenziali  

$$L \Rightarrow [fase1] [fase2] [fase3]$$
- Se ogni fase richiede **T** unità di tempo, un unico esecutore svolge un lavoro **L** ogni **3T** unità di tempo
- Per ridurre i tempi di produzione si possono utilizzare **più esecutori**

## Pipeline Generalità 2



- Soluzione (ideale) a parallelismo totale  
 $E1 \Rightarrow [fase1.A] [fase2.A] [fase3.A] \mid [fase1.D] \dots$   
 $E2 \Rightarrow [fase1.B] [fase2.B] [fase3.B] \mid [fase1.E] \dots$   
 $E3 \Rightarrow [fase1.C] [fase2.C] [fase3.C] \mid [fase1.F] \dots$
- $N$  esecutori svolgono un lavoro ogni  $3T/N$  unità di tempo
- Il problema è **come** preservare la **dipendenza funzionale** nell'esecuzione (di fasi) dei 'lavori' **A, B, C, D, E, F, ...**

## Pipeline Generalità 3



- Soluzione **pipeline** ad **esecutori generici**  
 $E1 \Rightarrow [fase1] [fase2] [fase3] [fase1] [fase2]$   
 $E2 \Rightarrow \dots [fase1] [fase2] [fase3] [fase1]$   
 $E3 \Rightarrow \dots [fase1] [fase2] [fase3]$
- Ogni esecutore esegue un ciclo di lavoro **completo** (*sistema totalmente replicato*)
- A regime,  $N$  esecutori svolgono un lavoro  $L$  ogni  $3T/N$  unità di tempo rispettandone la sequenza

# Pipeline Generalità 4



- Soluzione **pipeline** ad **esecutori specializzati**

E1 ⇒ [fase1] [fase1] [fase1] [fase1] [fase1]

E2 ⇒ ..... [fase2] [fase2] [fase2] [fase2]

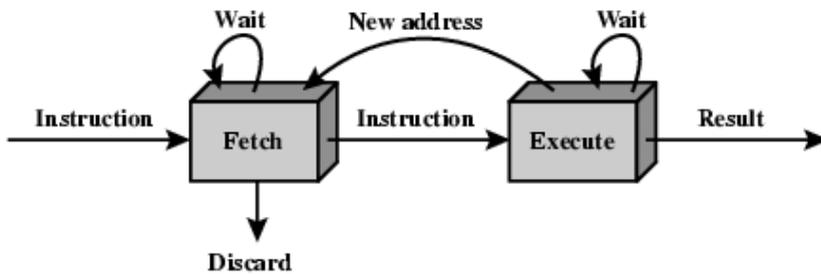
E3 ⇒ ..... [fase3] [fase3] [fase3]

- Ogni esecutore svolge sempre e solo la **stessa** fase di lavoro
- Soluzione più efficace in termini di **uso di risorse** (**3T/N** lavori con **N/3** risorse)

## Prefetch come pipeline a due stadi



(a) Simplified view



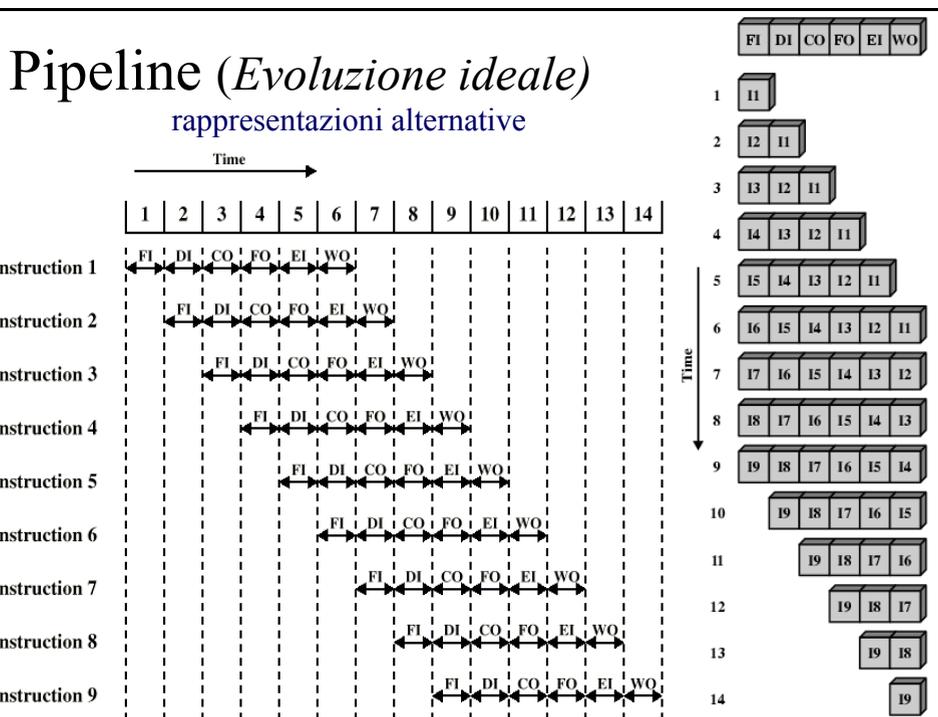
(b) Expanded view

# Pipeline

## *Decomposizione in fasi*



- L'esecuzione di una generica istruzione può essere suddivisa nelle seguenti fasi:
  - **fetch (FI)** lettura dell'istruzione
  - **decodifica (DI)** decodifica dell'istruzione
  - **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 prestazioni ideali



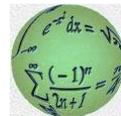
Le prestazioni ideali di una pipeline si possono calcolare matematicamente come segue

- Sia  $\tau$  il tempo di ciclo di una pipeline necessario per far avanzare di uno stadio le istruzioni attraverso una pipeline. Questo può essere determinato come segue:

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

- $\tau_m$  = massimo ritardo di stadio (ritardo dello stadio più oneroso)
- $k$  = numero di stadi nella pipeline
- $d$  = ritardo di commutazione di un registro, richiesto per l'avanzamento di segnali e dati da uno stadio al successivo

# Pipeline prestazioni ideali



Poiché  $\tau_m \gg d$ , il tempo totale  $T_k$  richiesto da una pipeline con  $k$  stadi per eseguire  $n$  istruzioni (senza considerare salti ed in prima approssimazione) è dato da

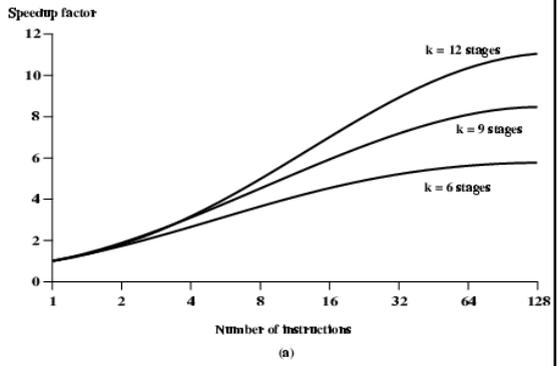
$$T_k = [k + (n-1)]\tau$$

in quanto occorrono  $k$  cicli per completare l'esecuzione della prima istruzione e  $n-1$  per le restanti istruzioni, e quindi il *fattore di velocizzazione* (speedup) di una pipeline a  $k$  stadi è dato da:

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

# Speedup

Calcolato in funzione del numero di istruzioni



Calcolato in funzione del numero di stadi

