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

  $$\mathbf{L} \Rightarrow [fase1]\ [fase2]\ [fase3]$$

- 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 ⇒ [fase1.**A**] [fase2.**A**] [fase3.**A**] | [fase1.**D**] …

  E2 ⇒ [fase1.**B**] [fase2.**B**] [fase3.**B**] | [fase1.**E**] …

  E3 ⇒ [fase1.**C**] [fase2.**C**] [fase3.**C**] | [fase1.**F**] …
- **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**

- 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]

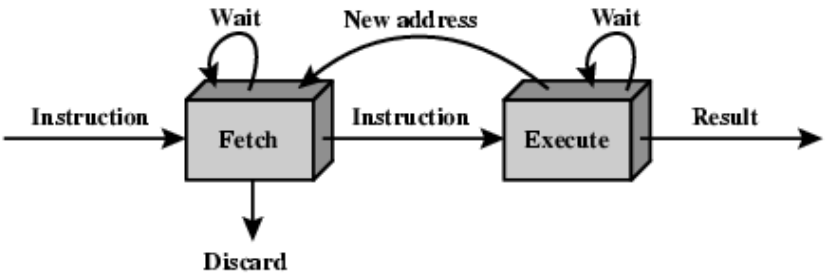
E2 ⇒ ..... [fase2] [fase2] [fase2] [fase2]

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



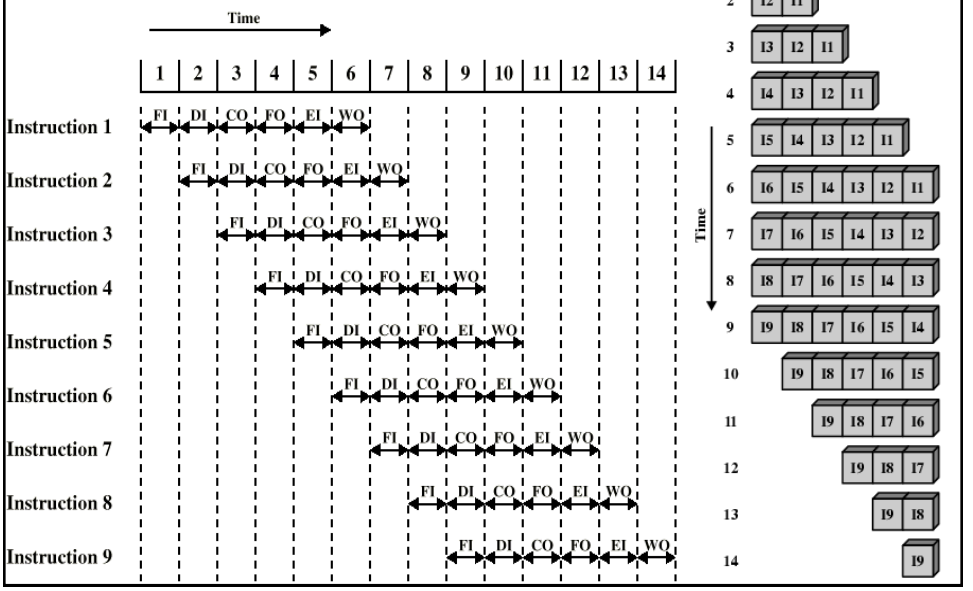
(b) Expanded view

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 (*Evoluzione ideale)*
rappresentazioni alternative



Pipeline
prestazioni ideali

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

- Sia τ 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:

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

  - τ_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 $\tau_m >> d$, 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

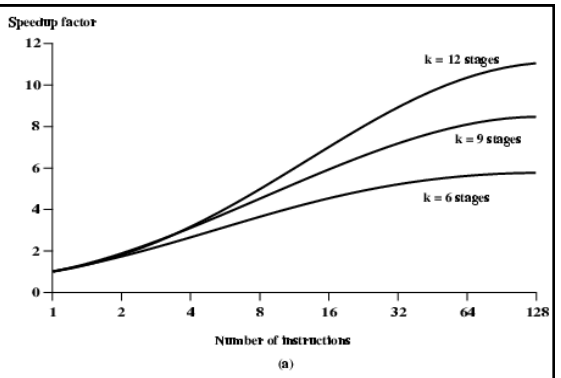
$$T_k = [k + (n\text{-}1)]\tau$$

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:

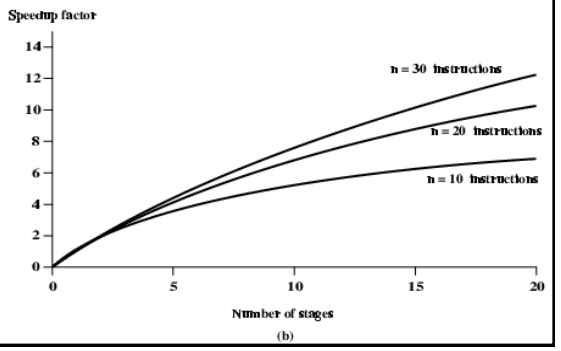
$$S_k = \frac{T_1}{T_k} = \frac{nk\tau}{[k+(n-1)]\tau} = \frac{nk}{[k+(n-1)]}$$

Speedup

Calcolato in funzione del numero di istruzioni



Calcolato in funzione del numero di stadi



Pipeline
Problemi 1

- Vari fenomeni pregiudicano il raggiungimento del massimo di parallelismo teorico (**stallo**)
  - **Sbilanciamento delle fasi**
    - Durata diversa per fase e per istruzione
  - **Problemi strutturali**
    - La sovrapposizione totale di tutte le (fasi di) istruzioni causa conflitti di accesso a risorse limitate e condivise (ad esempio la memoria per gli stadi FI, FO, WO)

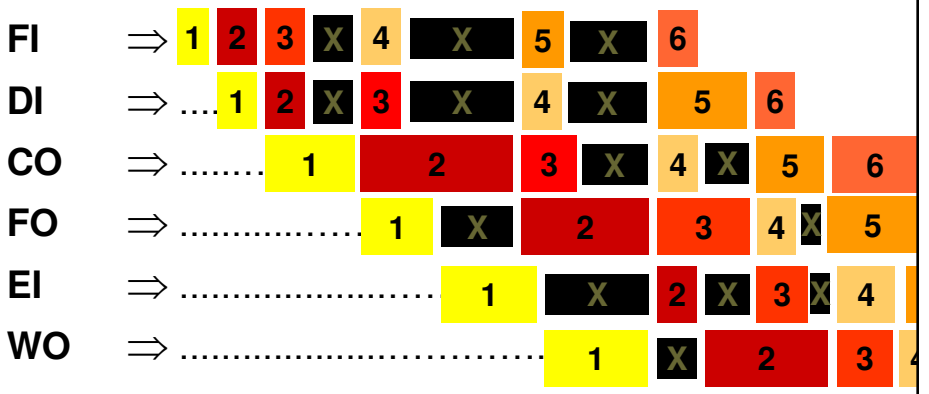
Pipeline
Problemi 2

- **Dipendenza dai dati**
  - L'operazione successiva dipende dai risultati dell'operazione precedente
- **Dipendenza dai controlli**
  - Istruzioni che causano una violazione di sequenzialità (p.es.: salti condizionali) invalidano il principio del *pipelining* sequenziale

Pipeline
Sbilanciamento delle fasi 1

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

Pipeline
Sbilanciamento delle fasi 2



X Tempo di attesa forzata dovuta allo sbilanciamento delle fasi

Pipeline
Sbilanciamento delle fasi 3

Possibili soluzioni allo sbilanciamento:

- 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

Pipeline
Problemi strutturali

Problemi

- Maggiori risorse interne (**severità bassa**): l'evoluzione tecnologica ha spesso permesso di duplicarle (es. registri)
- Colli di bottiglia (**severità alta**): l'accesso alle risorse esterne, p.es.: memoria, è molto costoso e molto frequente (anche 3 accessi per ciclo di clock)

Soluzioni

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

Pipeline
Dipendenza dai dati 1

- Un dato modificato nella fase **EI** dell'istruzione corrente può dover essere utilizzato dalla fase **FO** dell'istruzione successiva

INC [0123]
CMP [0123], AL

Ci sono altri tipi di dipendenze ?

Dipendenze

Si consideri la sequenza

```
istruzione i
istruzione j
```

Esempio visto: “lettura dopo scrittura” (ReadAfterWrite)

– ***j*** leggere prima che ***i*** abbia scritto

Altro caso: “scrittura dopo scrittura” (WriteAfterWrite)

– ***j*** scrive prima che ***i*** abbia scritto

Altro caso: “scrittura dopo lettura” (WriteAfterRead)

– ***j*** scrive prima che ***i*** abbia letto (caso raro in pipeline)

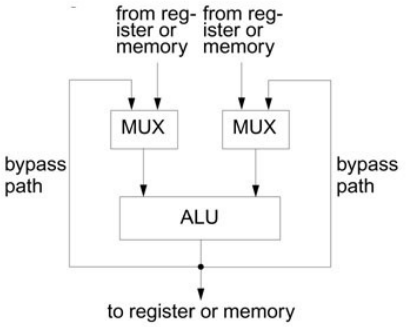
Pipeline

Dipendenza dai dati 2

Soluzioni

- Introduzione di fasi non operative (***nop***)
- Individuazione del rischio e prelievo del dato direttamente all’uscita dell’ALU (**data forwarding**) ➡
- Risoluzione a livello di compilatore **(vedremo esempi per l’architettura MIPS)**
- Riordino delle istruzioni (**pipeline scheduling**)

Pipeline
Data forwarding



senza bypass path

```
I1:   MUL   R2,R3    R2 ← R2 * R3
I2:   ADD   R1,R2    R1 ← R1 + R2
```

Clock cycle →	1	2	3	4	5	6	7	8	9	10	11	12
MUL R2,R3	FI	DI	CO	FO	EI	WO						
ADD R1,R2		FI	DI	CO	stall	stall	FO	EI	WO			
Instr. i+2			FI	DI			CO	FO	EI	WO		

con bypass path

Clock cycle →	1	2	3	4	5	6	7	8	9	10	11	12
MUL R2,R3	FI	DI	CO	FO	EI	WO						
ADD R1,R2		FI	DI	CO	stall	FO	EI	WO				

Pipeline
Dipendenza dai controlli

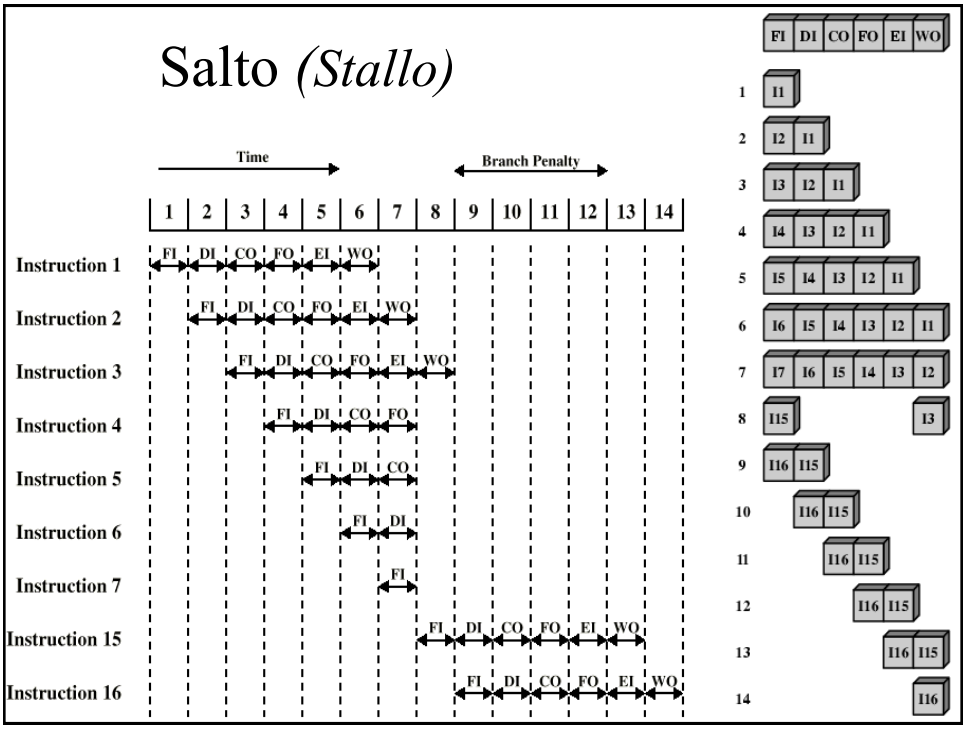
- Tutte le istruzioni che modificano il PC (salti condizionati e non, chiamate a e ritorni da procedure, interruzioni) invalidano la pipeline
- La fase **fetch** successiva carica l'istruzione seguente, che può *non essere* quella giusta
- Tali istruzioni sono circa il 30% del totale medio di un programma

Pipeline
Dipendenza dai controlli

Soluzioni

- Mettere in **stallo** la pipeline fino a quando non si è calcolato l'indirizzo della prossima istruzione
  - Pessima efficienza, massima semplicità
- Individuare le istruzioni critiche per anticiparne l'esecuzione, eventualmente mediante apposita logica di controllo
  - Compilazione complessa, hardware specifico

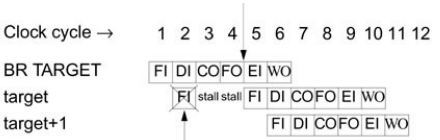
Salto *(Stallo)*



Salto *(Stallo)*

Salto incondizionato

indirizzo di salto conosciuto qui

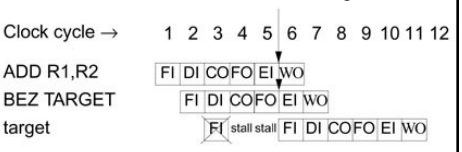


inizialmente caricata istruzione successiva a BR TARGET, poi eliminata quando si riconosce il salto incondizionato alla fine della fase DI

Salto condizionato

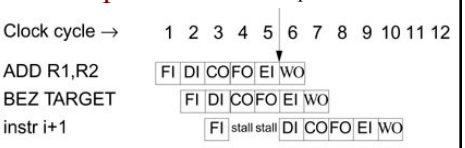
Salto preso

condizione ed indirizzo di salto conosciuti qui



Salto non preso

condizione di salto conosciuta qui



Funzionamento pipeline a 6 stadi con trattamento dei salti ed interrupt tramite svuotamento