

# 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



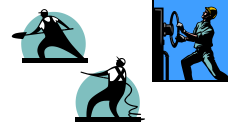
- Soluzione (ideale) a parallelismo totale  
 $E1 \Rightarrow [fase1.A] [fase2.A] [fase3.A] \mid [fase1.D] \dots$   
 $E2 \Rightarrow [fase1.B] [fase2.B] [fase3.B] \mid [fase1.E] \dots$   
 $E3 \Rightarrow [fase1.C] [fase2.C] [fase3.C] \mid [fase1.F] \dots$
- $N$  esecutori svolgono un lavoro ogni  $3T/N$  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]

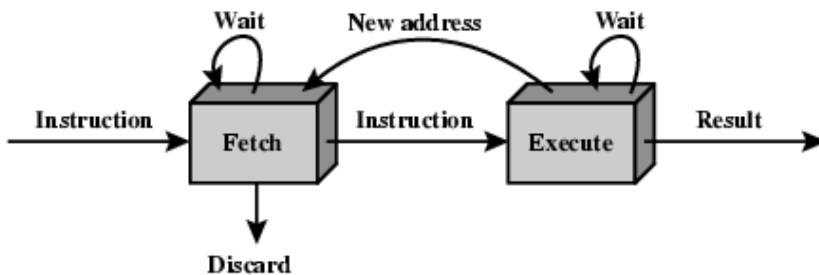
E3 ⇒ ..... [fase3] [fase3] [fase3]

- 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



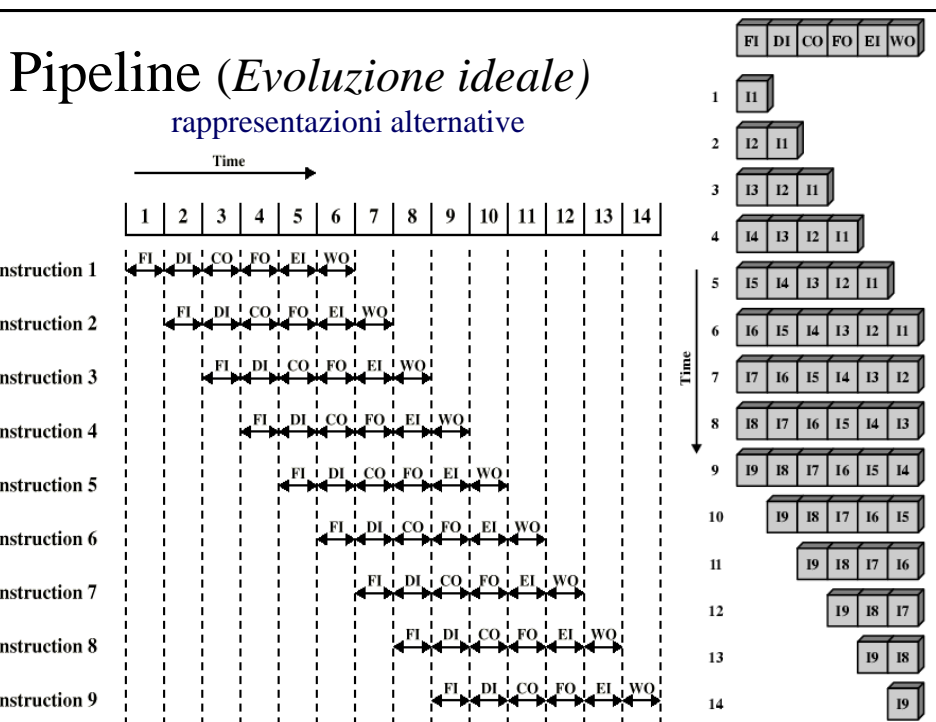
(b) Expanded view

# Pipeline

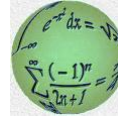
## *Decomposizione in fasi*



- 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

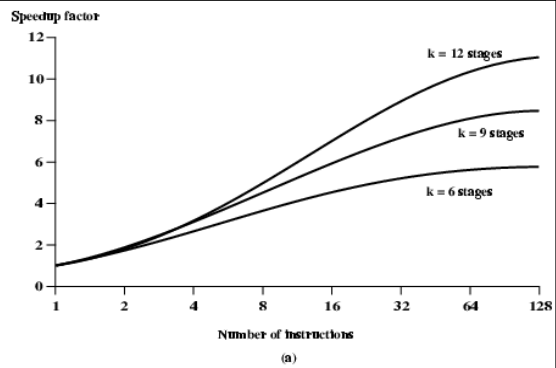
$$T_k = [k + (n-1)]\tau$$

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:

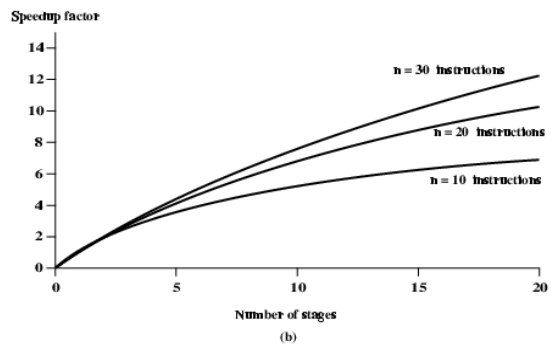
$$S_k = \frac{T_1}{T_k} = \frac{nk\tau}{[k + (n-1)]\tau} = \frac{nk}{[k + (n-1)]}$$

# Speedup

Calcolato in funzione  
del numero di istruzioni



Calcolato in funzione  
del numero di stadi



## 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

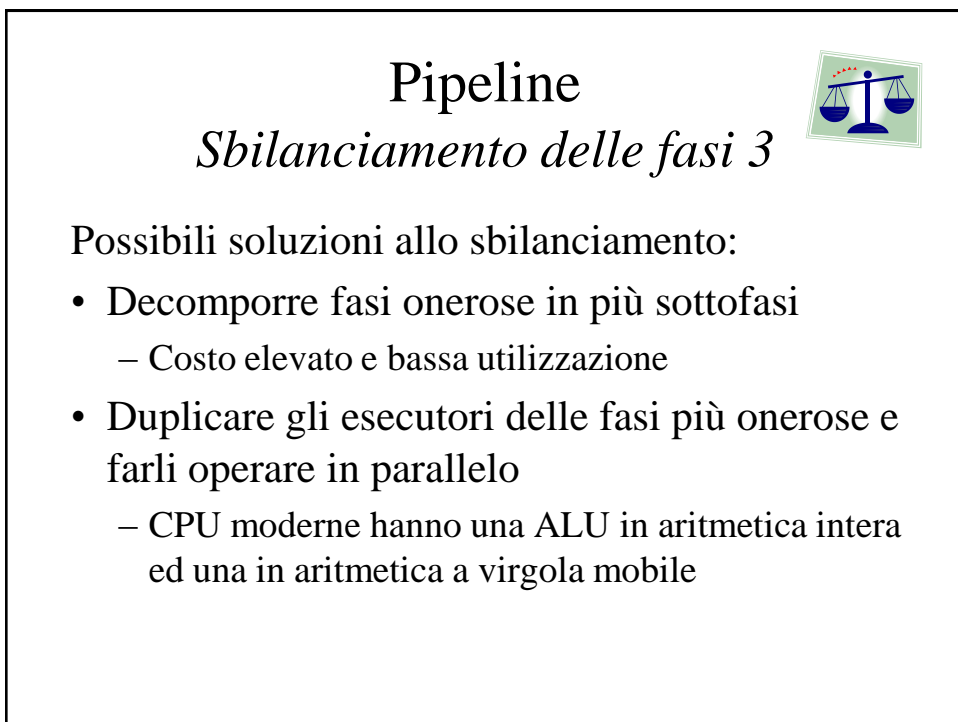
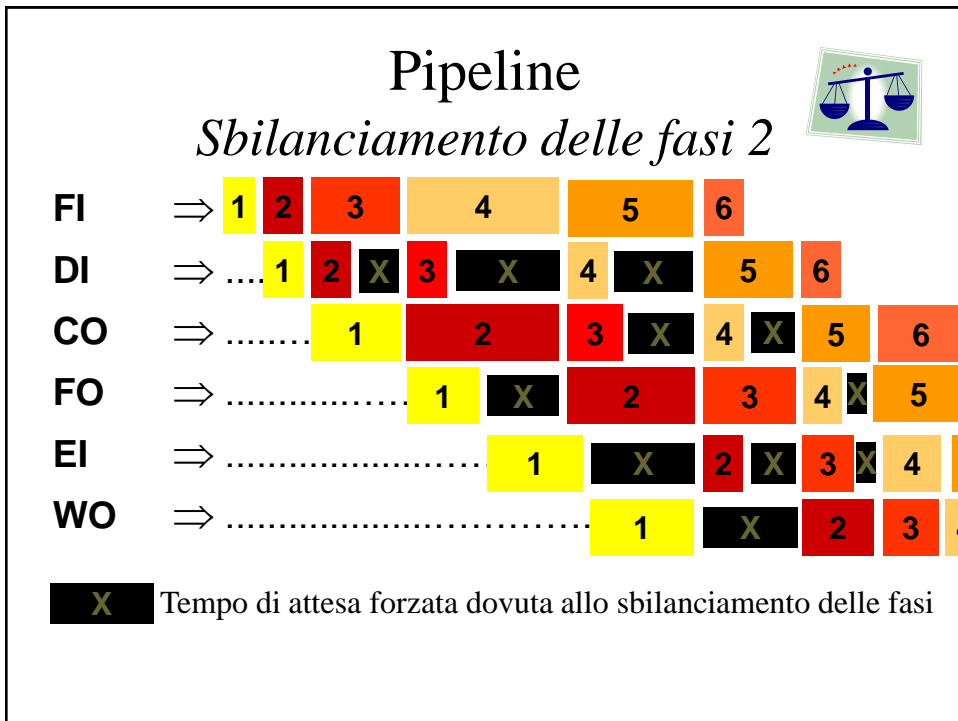
- Istruzioni che causano una violazione di sequenzialità (p.es.: salti condizionali) invalidano il principio del *pipelining* sequenziale

## 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

## *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)

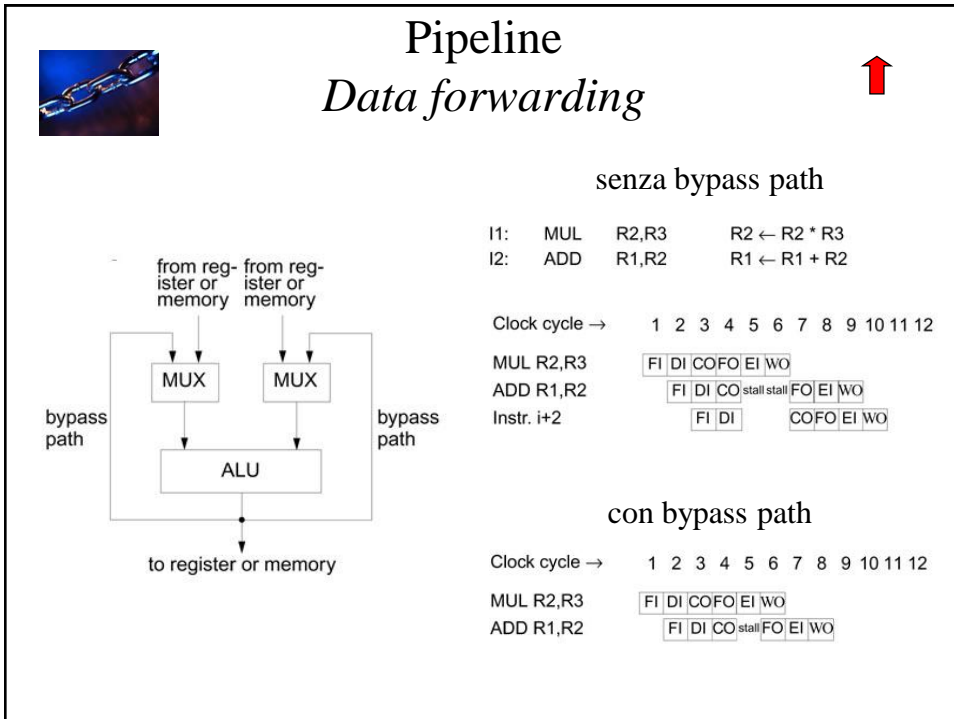


## Pipeline


### *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 *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

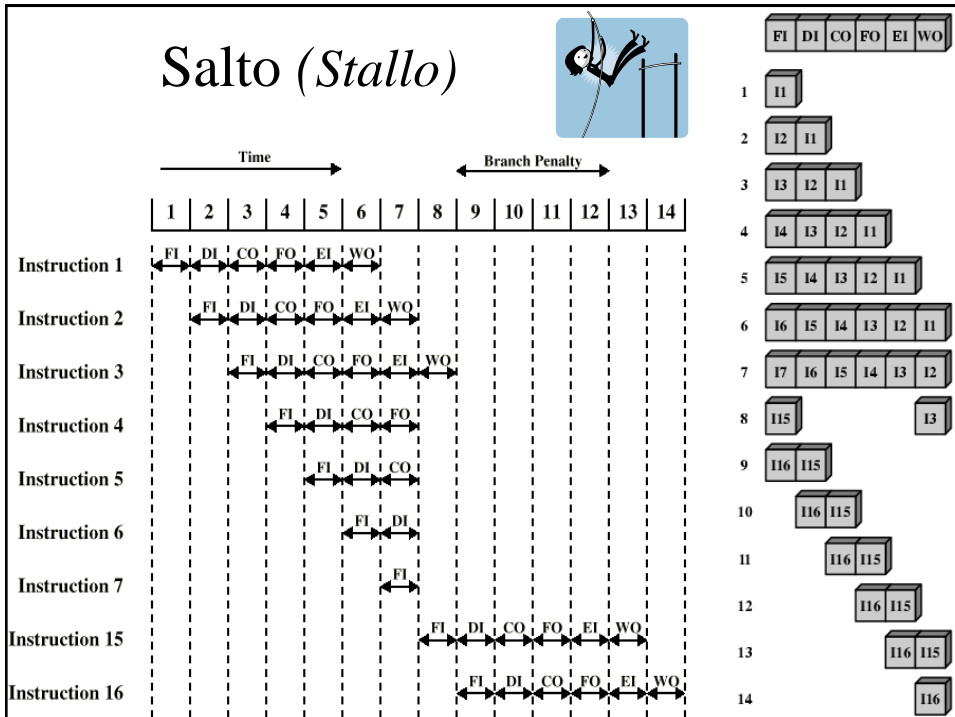
## Dipendenza dai controlli



### 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

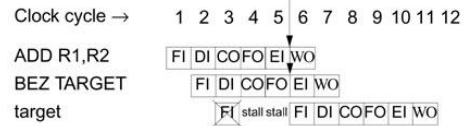


inizialmente caricata istruzione successiva a BR TARGET, poi eliminata quando si riconosce il salto incondizionato alla fine della fase DI

## Salto condizionato

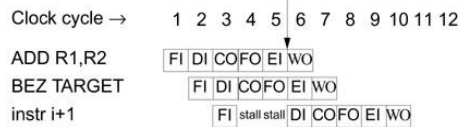
### Salto preso

condizione ed indirizzo di salto conosciuti qui

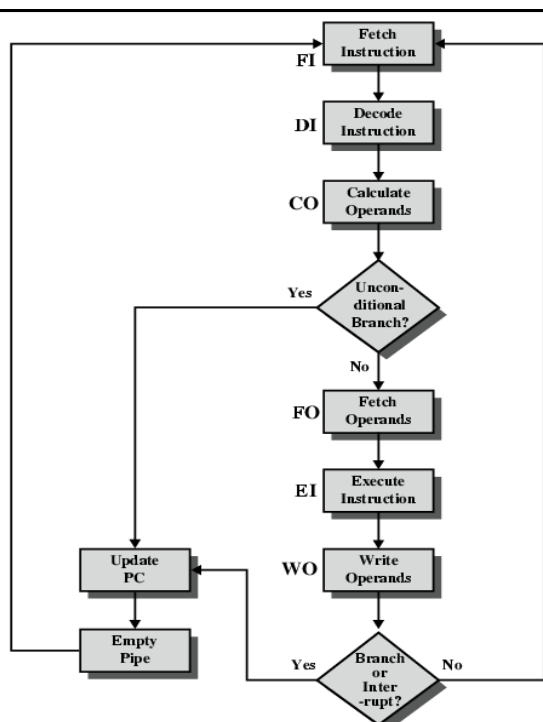


### Salto non preso

condizione di salto conosciuta qui



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



## Pipeline

### *Dipendenza dai controlli*



Alcune soluzioni per salti condizionati

- flussi multipli (*multiple streams*) →
- prelievo anticipato della destinazione (*prefetch branch target*) →
- buffer circolare (*loop buffer*) →
- predizione del salto (*branch prediction*) →
- salto ritardato (*delayed branch*) →



## Pipeline

### *Dipendenza dai controlli*





**Flussi multipli:** replicare le parti iniziali della pipeline, una che contenga l'istruzione successiva a quella corrente di salto (nel caso il salto non avvenga), e l'altra l'istruzione destinazione (*target*) del salto (nel caso in cui il salto avvenga)

Problemi di questa soluzione:

- conflitti nell'accesso alle risorse (registri, memoria, ALU,...) da parte delle 2 pipeline
- presenza di salti condizionali in sequenza che entrano nelle 2 pipeline prima che si sia risolta la condizione del primo salto condizionale (occorrerebbero 2 pipeline aggiuntive per ogni ulteriore salto condizionale...)

# Pipeline


## *Dipendenza dai controlli*

**Prelievo anticipato della destinazione:** quando si incontra un salto condizionato si effettua il fetch anticipato della istruzione di destinazione del salto in modo da trovarla già caricata nel caso in cui il salto debba avvenire.

Problemi di questa soluzione:

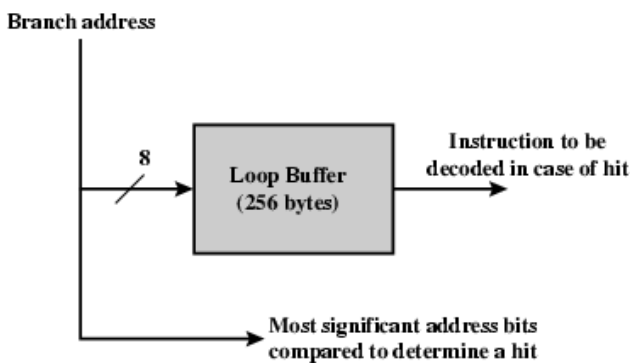
- non evita l'eventuale svuotamento della pipeline con conseguente perdita di prestazioni



# Pipeline

## *Dipendenza dai controlli*

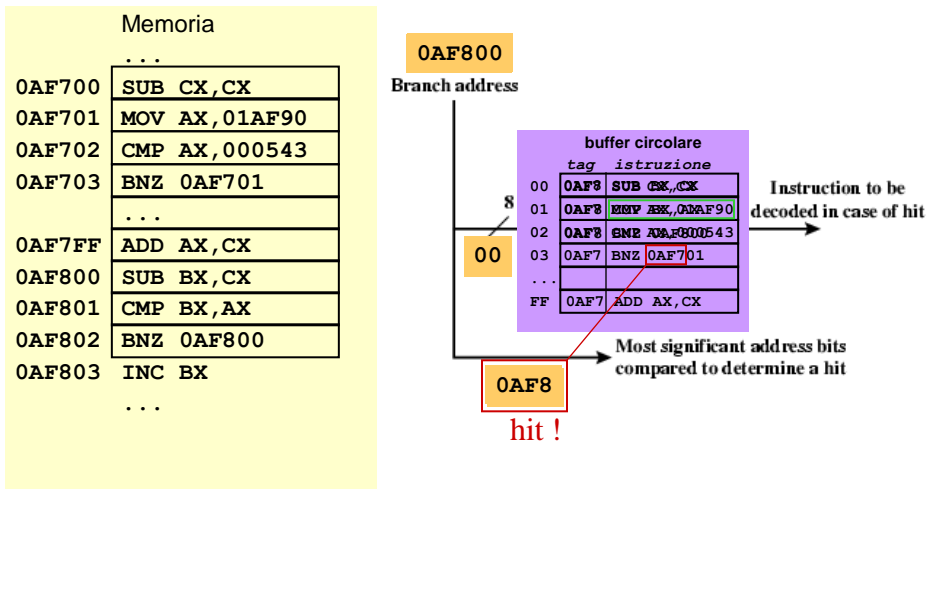
**Buffer circolare:** si utilizza una memoria piccola e molto veloce (il buffer circolare) dove mantenere le ultime  $n$  istruzioni prelevate. In caso di salto, si controlla se l'istruzione destinazione è già presente nel buffer, così da evitare il fetch della stessa.



```

graph LR
    BA[Branch address] -- 8-bit bus --> LB[Loop Buffer 256 bytes]
    LB --> I[Instruction to be decoded in case of hit]
    BA --> C[Most significant address bits compared to determine a hit]
  
```

## Buffer circolare (senza prefetch)



## Pipeline



### Dipendenza dai controlli

**Buffer circolare:** si utilizza una memoria piccola e molto veloce (il buffer circolare) dove mantenere le ultime  $n$  istruzioni prelevate. In caso di salto, si controlla se l'istruzione destinazione è già presente nel buffer, così da evitare il fetch della stessa.

#### Vantaggi:

- anticipando il fetch, alcune delle istruzioni successive a quella corrente saranno già presenti nel buffer e se non si ha salto non ci sarà bisogno di caricarle dalla memoria
- se si salta in avanti di poche istruzioni (vedi trattamento del costrutto IF-THEN-ELSE), l'istruzione destinazione sarà già presente nel buffer
- se il salto condizionale realizza un ciclo le cui istruzioni possono essere tutte contenute nel buffer, non c'è bisogno di effettuare fetch ripetuti delle stesse istruzioni





# Pipeline

## Dipendenza dai controlli

**Predizione dei salti:** si cerca di prevedere se il salto sarà intrapreso oppure no.

Varie possibilità:

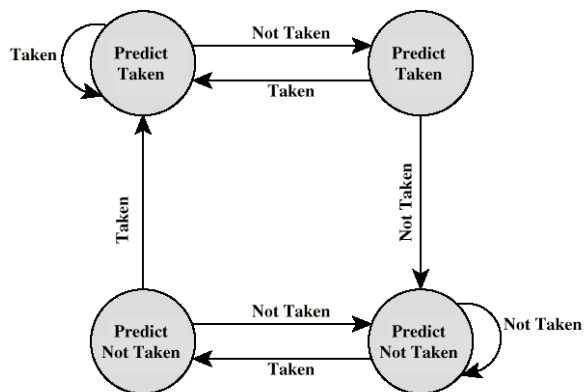
- previsione di saltare sempre
  - previsione di non saltare mai
  - previsione in base al codice operativo
- } *approcci statici*
- bit *taken/not taken*
  - tabella della storia dei salti
- } *approcci dinamici*



**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.

**bit *taken/not taken* :**

- ad ogni istruzione di salto condizionato si associano uno o più bit che codificano la storia recente.
- bit memorizzati non in memoria centrale ma in una locazione temporanea ad accesso molto veloce



esempio con 2 bit

**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



esempio:

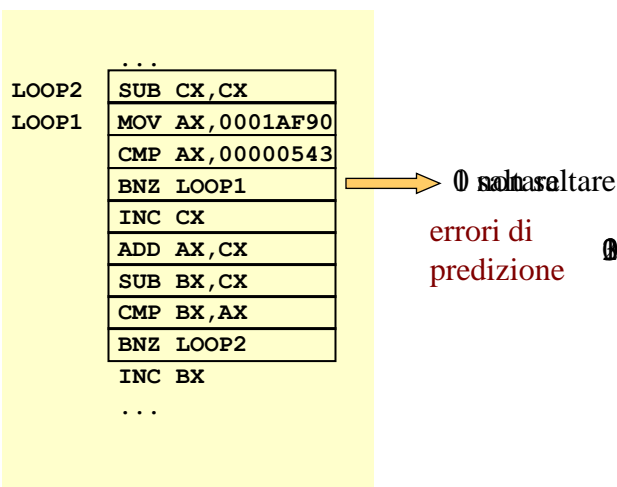
```

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

```

- **Predizione con 1 bit:** si predice il comportamento osservato l'ultima volta
  - dopo la prima esecuzione del ciclo, in uscita dal ciclo, il bit assegnato a BNZ ricorderà che il salto **non è stato preso**, così che, quando si rientra nel ciclo si avrà un primo errore per la prima iterazione del ciclo (che invece è preso), le successive predizioni saranno giuste, tranne l'ultima, quando si esce dal ciclo: in totale **2 errori**
- **Predizione con 2 bit:** vedi lucido precedente
  - dopo la prima esecuzione del ciclo, si commette **un solo errore** di predizione all'uscita del ciclo

## Predizione dinamica 1 bit



# Predizione dinamica 2 bit

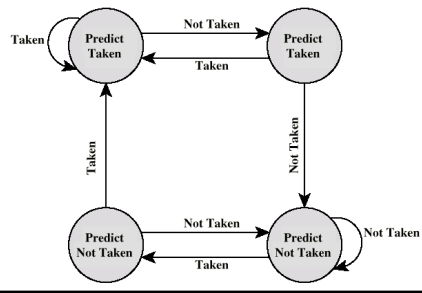
```

...
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
...

```

errori di predizione 0

10 saltare

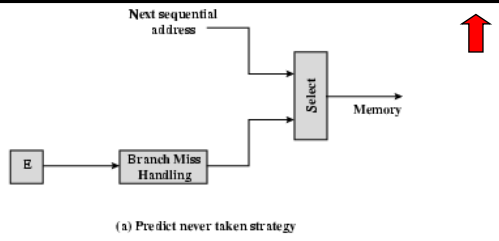


## Problemi di bit taken/not taken :

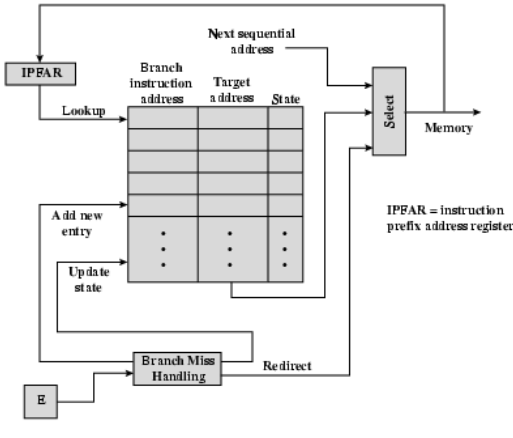
- quando si decide di saltare, bisogna aspettare la decodifica dell'indirizzo destinazione prima di poter prelevare l'istruzione destinazione
- si può anticipare il prelievo a patto di salvare opportune info nel *branch target buffer* o *branch history table*

### tabella della storia dei salti:

- piccola memoria associata allo stadio fetch della pipeline
- ogni riga della tabella è costituita da 3 elementi:
  1. indirizzo istruzione salto,
  2. l'indirizzo destinazione del salto (o l'istruzione destinazione stessa),
  3. alcuni bit di storia che descrivono lo stato dell'uso dell'istruzione



(a) Predict never taken strategy



(b) Branch history table strategy

## Salto ritardato (delayed branch)

**Idea base:** utilizzare gli stadi inattivi a causa dello stallo per fare del lavoro utile

Delayed branch:

- La CPU esegue **sempre** l'istruzione che segue il salto e solo dopo altera, se necessario, la sequenza di esecuzione delle istruzioni
- L'istruzione che segue quella di salto si dice essere posta nel *branch delay slot*
- Il **compilatore** cerca di allocare nel *branch delay slot* una istruzione "opportuna"



## Salto ritardato (delayed branch)

**codice scritto dal programmatore**

istruzione indipendente dalle altre	→	MUL R3,R4	R3 ← R3*R4
		SUB #1,R2	R2 ← R2-1
		ADD R1,R2	R1 ← R1+R2
		BEZ TAR	branch if zero
istruzione eseguita solo se non si prende il salto	→	MOVE #10,R1	R1 ← 10
		TAR	-----

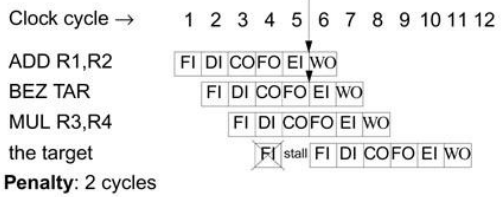
**codice ottimizzato dal compilatore**

SUB #1,R2	istruzione eseguita in ogni caso: si trova nel <i>branch delay slot</i> !!
ADD R1,R2	
BEZ TAR	
MUL R3,R4	
MOVE #10,R1	
-----	istruzione eseguita solo se non si prende il salto
TAR	

## Salto ritardato (delayed branch)

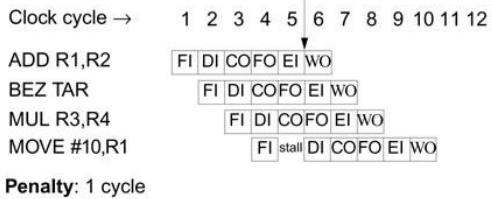
### Salto preso

condizione ed indirizzo di salto conosciuti qui

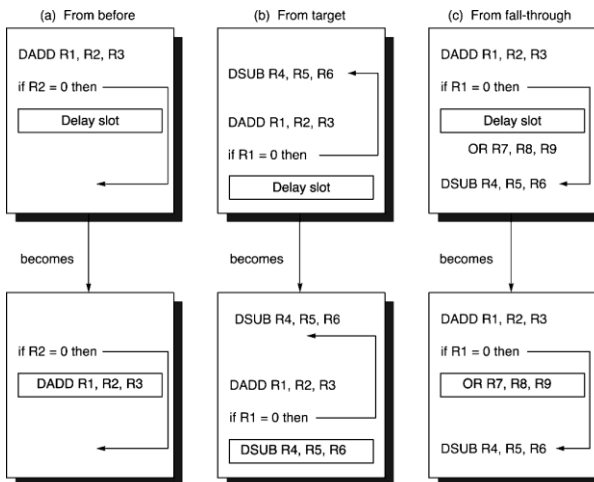


### Salto non preso

condizione di salto conosciuta qui



## Salto ritardato (delayed branch)

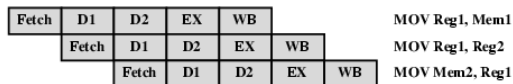


b) e c) legali solo se R4 e R7 sono registri temporanei il cui contenuto può essere “sporcat” senza cambiare la semantica del programma

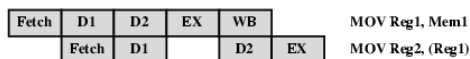
## Intel 80486 Pipelining

- 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
  - Provvede a controllare i calcoli per i modi di indirizzamento più complessi
- Esecuzione (EX)
  - Operazioni ALU, accesso alla cache (memoria), aggiornamento registri
- 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



(a) No Data Load Delay in the Pipeline



(b) Pointer Load Delay



(c) Branch Instruction Timing