

# Pipeline

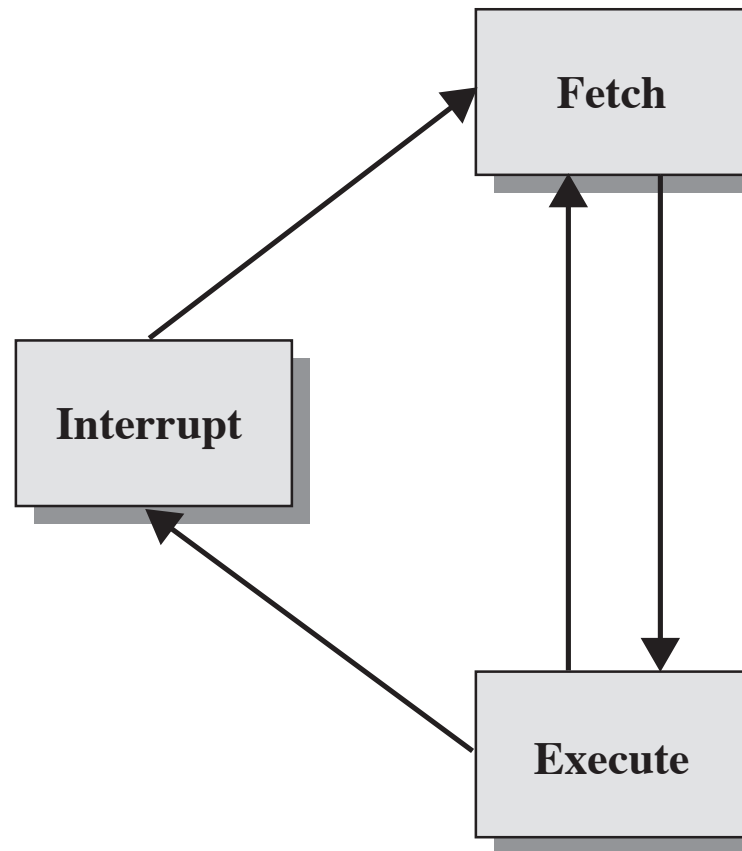
**Argomento:**

- Prefetch
- Pipeline
- Hazards
- Branch tables

**Materiale di studio:**

- Capitolo 14, sezione 14.1-14.6

# Ciclo di esecuzione delle istruzioni: Fetch/Execute/Interrupt

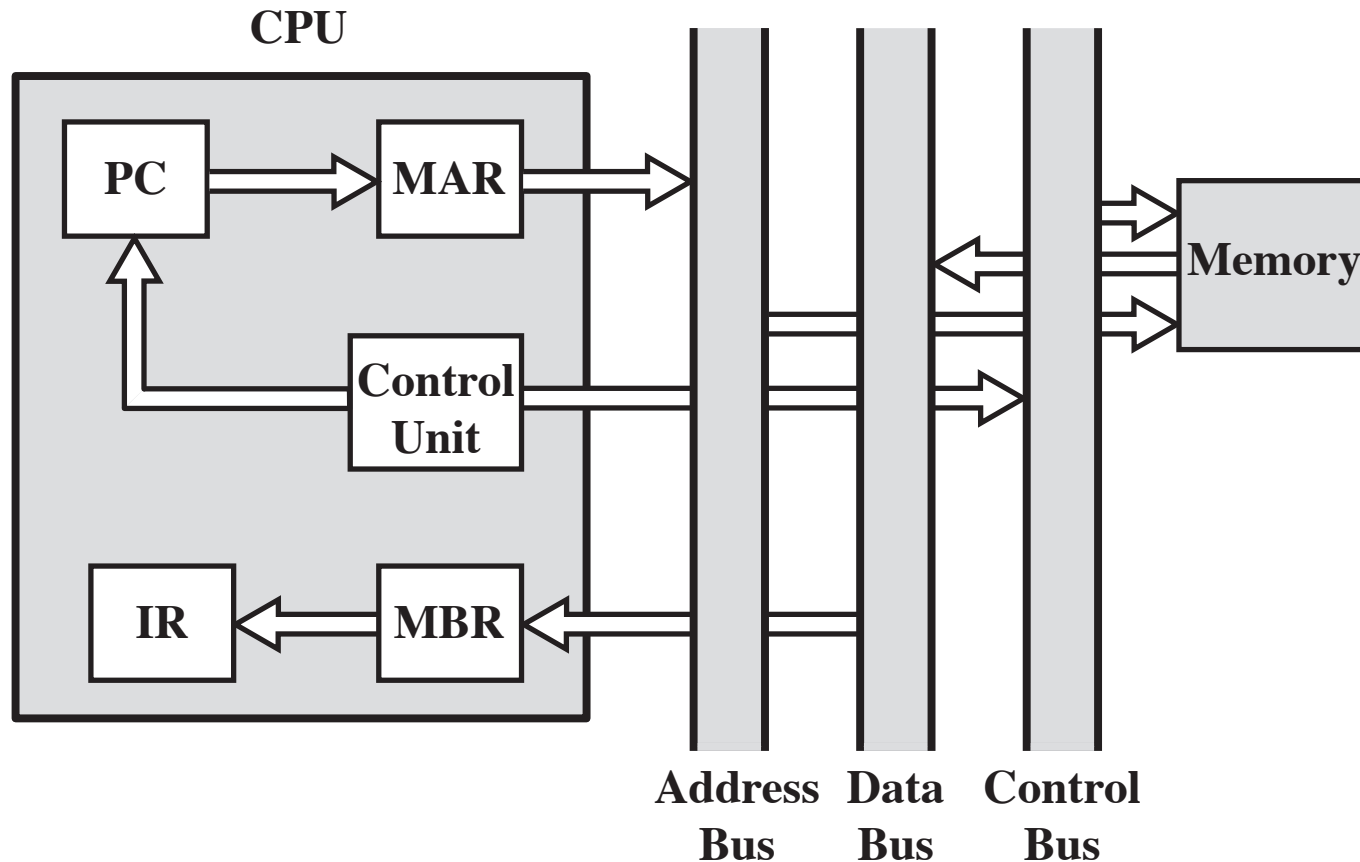


# Instruction fetch

## **Fetch:**

1. PC contiene l'indirizzo della istruzione successiva;
2. Il valore in PC viene spostato in MAR;
3. L'indirizzo viene emesso sul bus degli indirizzi;
4. La unità di controllo richiede una lettura in memoria principale;
5. Il risultato della lettura in memoria principale viene inviato nel bus dati, copiato in MBR, e infine in IR;
6. Contemporaneamente alle istruzioni 2-5, il PC viene incrementato.

# Diagramma di fetch

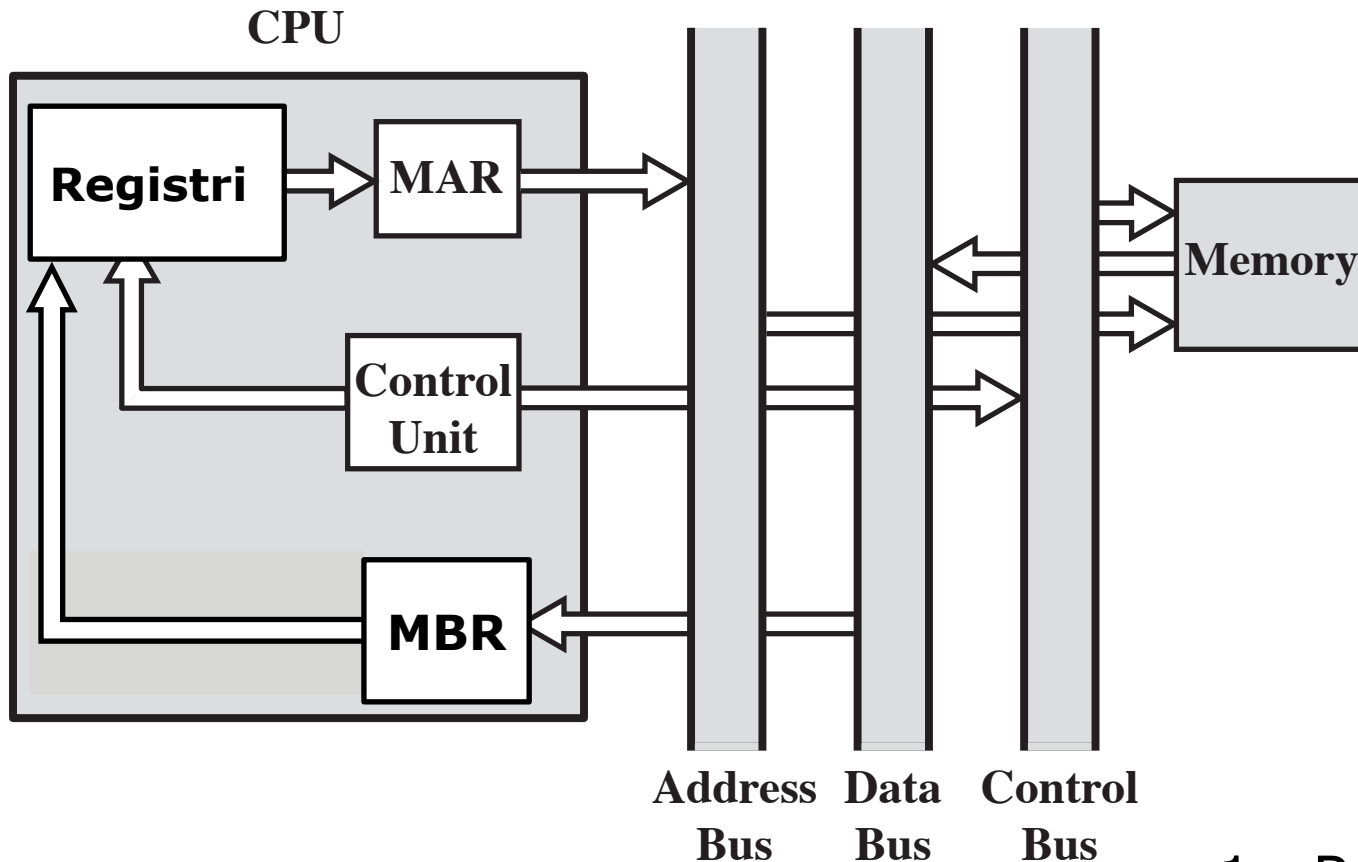


MBR = Memory buffer register  
MAR = Memory address register  
IR = Instruction register  
PC = Program counter

# Execute

- Dipende dalla istruzione da eseguire
- Può includere
  - Lettura/scrittura della memoria
  - Input/output
  - Trasferimento di dati fra registri e/o in registri
  - Operazioni della ALU

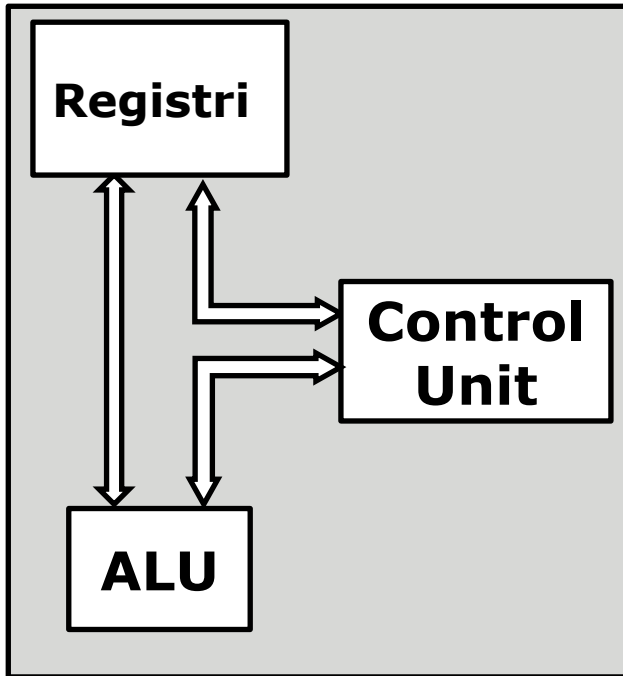
# Diagramma di execute: LDR R0, [R1]



MBR = Memory buffer register  
MAR = Memory address register  
IR = Instruction register  
PC = Program counter

1. R1 → MAR
2. MAR → memoria
3. Memoria → MBR
4. MBR → R0

# Diagramma di execute: ADD R0, R1, R2



1. R1 → ALU
2. R2 → ALU
3. Somma in ALU
4. Risultato → R0

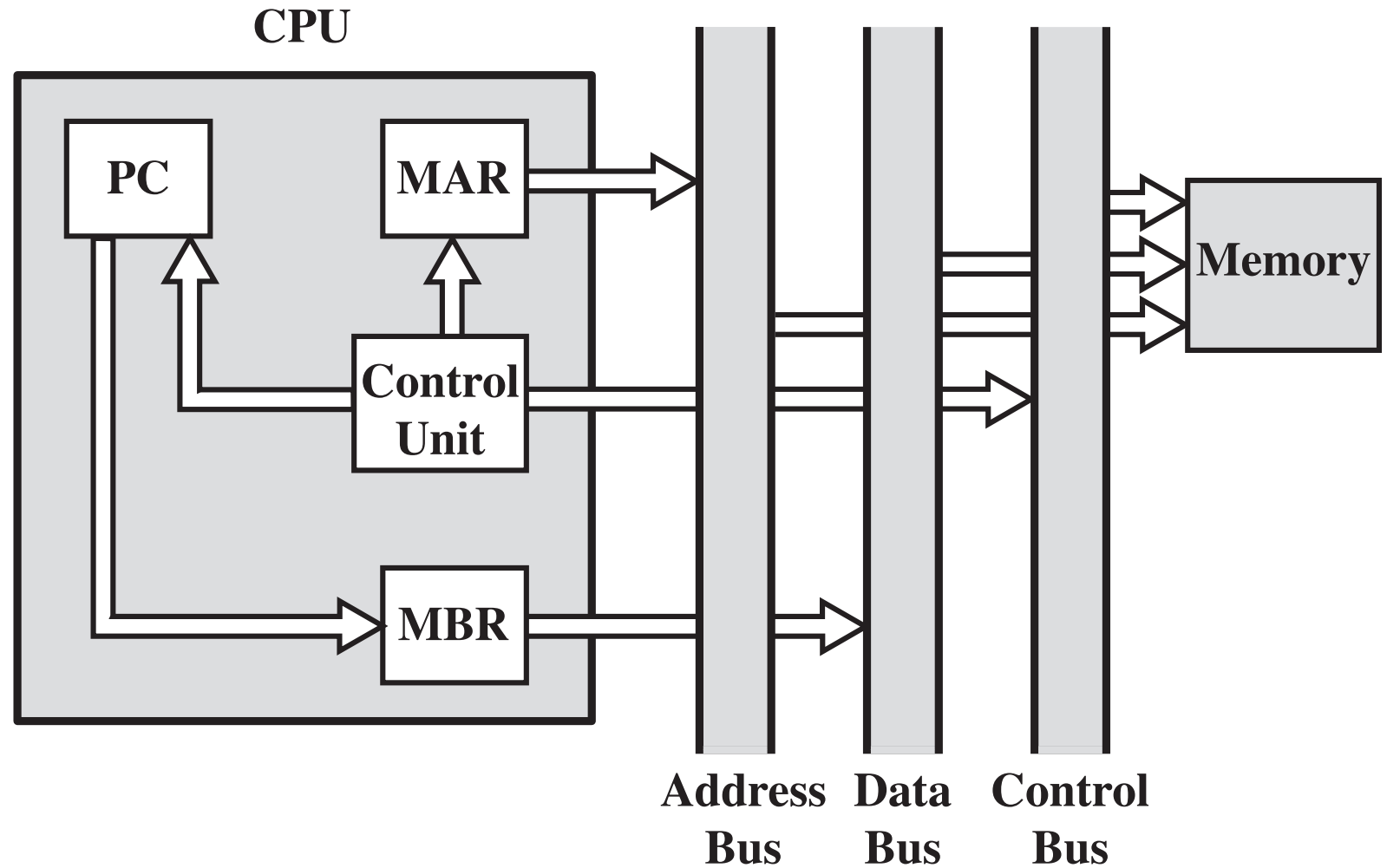
# Interrupt

## **Interrupt:**

1. Il contenuto corrente del PC viene salvato per permettere il ripristino della esecuzione dopo la gestione dell'interruzione:
  - a. Contenuto di PC copiato in MBR
  - b. Indirizzo di locazione dove salvare PC (es. stack pointer) copiato in MAR
  - c. Contenuto di MBR scritto in memoria
2. L'indirizzo della prima istruzione della routine di servizio della interruzione viene caricato in PC;
3. Fetch della istruzione puntata da PC.



# Diagramma di interrupt



# Prefetch

- Esecuzione semplice:
  - Per ogni istruzione, si esegue prima il fetch e poi l'execute
  - Quando termina l'esecuzione di un'istruzione, si procede con il fetch della successiva istruzione.
- Ad ogni istante una solo delle due fasi è attiva
  - I circuiti logici associati alle due fasi sono distinti
- Si può prelevare l'istruzione successiva durante l'esecuzione dell'istruzione corrente
  - Questa operazione si chiama **instruction prefetch**

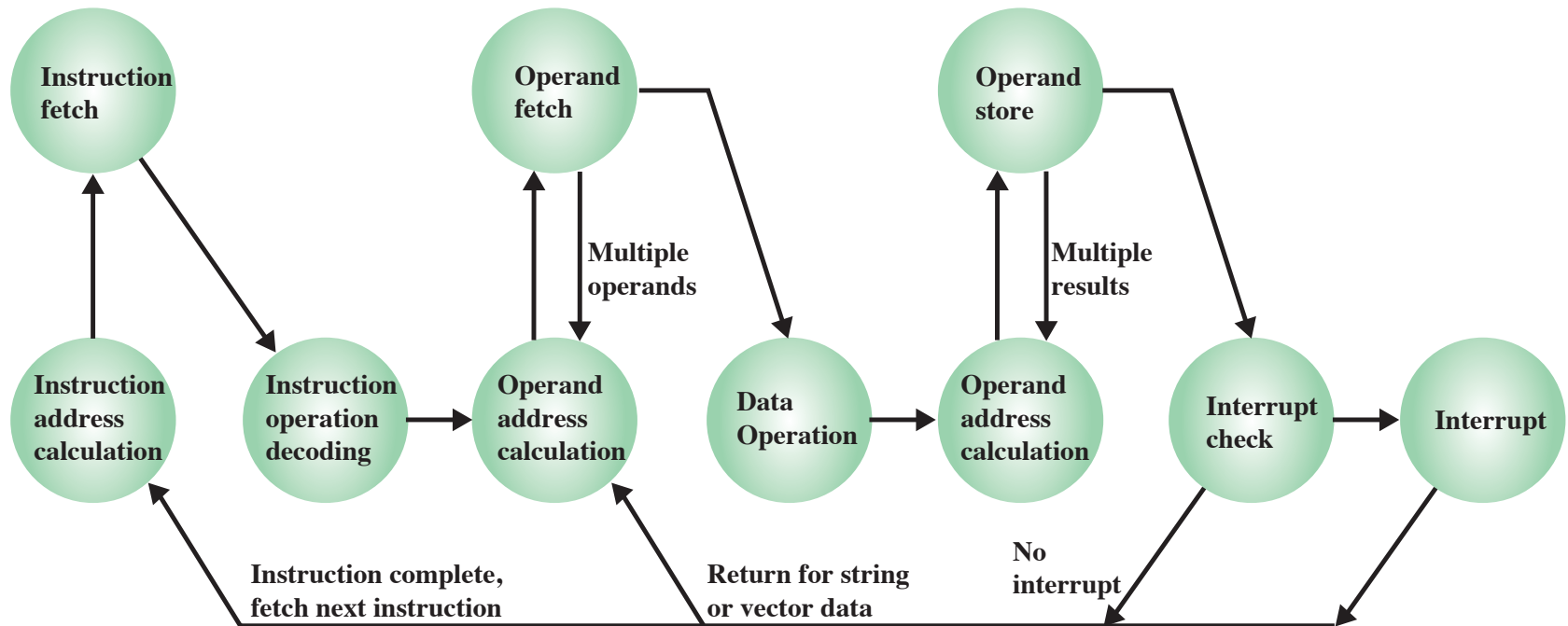
# Esecuzione con prefetch



# Limite dell'approccio a 2 fasi

- Il prefetch non raddoppia le prestazioni:
  - L'esecuzione di istruzioni branch condizionato possono rendere vano il prefetch: se la prossima istruzione dipende dal risultato dell'istruzione in esecuzione, non possiamo effettuare il prefetch
  - La fase di prelievo è tipicamente più breve della fase di esecuzione
    - Conviene scomporre execute in più fasi di **durata simile**.
- Aggiungere più fasi per migliorare le prestazioni → **Pipeline**

# Ciclo di esecuzione in dettaglio



# Scomposizione in 6 fasi

## 1. Fetch instruction (FI)

- Leggi la prossima istruzione

## 2. Decode instruction (DI)

- Determina l'opcode e il tipo di operandi

## 3. Calculate operands (CO)

- Calcola l'indirizzo effettivo di ogni operando
- Gestione degli indirizzamenti: con offset, diretto, indiretto di registro/memoria,...

## 4. Fetch operands (FO)

- Recupera gli operandi dalla memoria
- Operandi nei registri non richiedono il fetch

## 5. Execute instruction (EI)

- Esegui l'istruzione indicata

## 6. Write operand (WO)

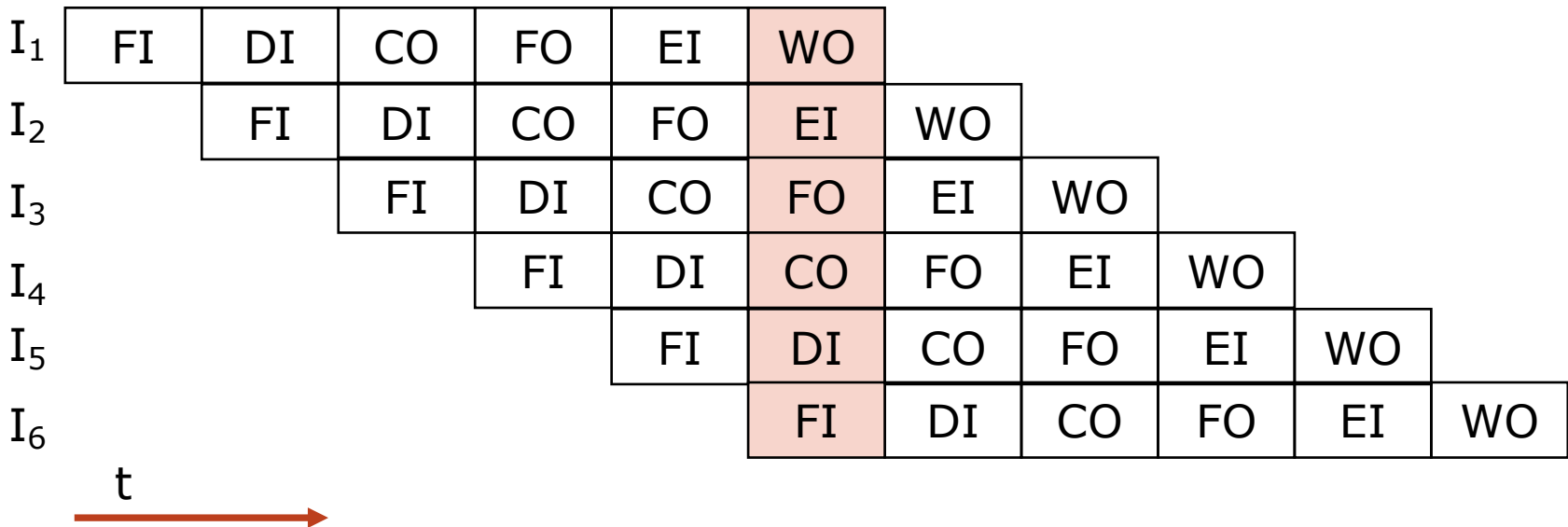
- Scrivi il risultato in memoria o in un registro

# Scomposizione in 6 fasi



# Pipelining

Se le fasi sono eseguite da sezioni indipendenti di **hardware**, possono essere tutte contemporaneamente attive (su istruzioni diverse)

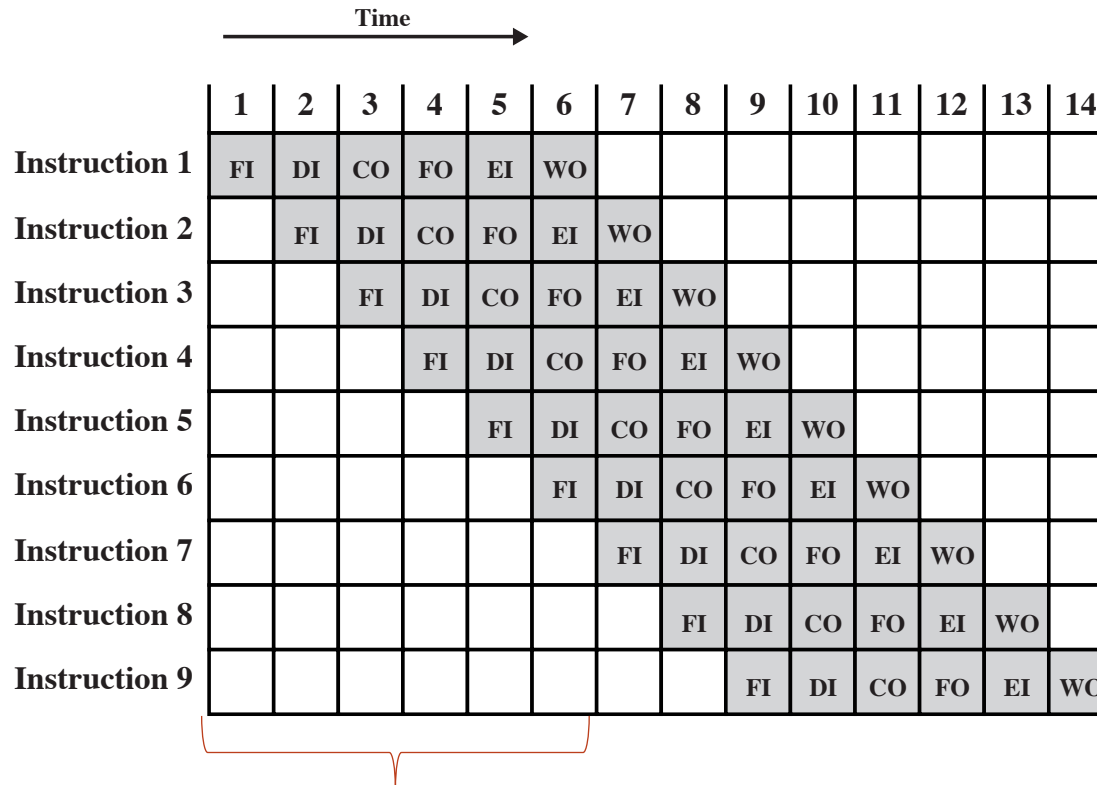


Dopo il transitorio: istruzione completata ad ogni stadio



# Diagramma temporale

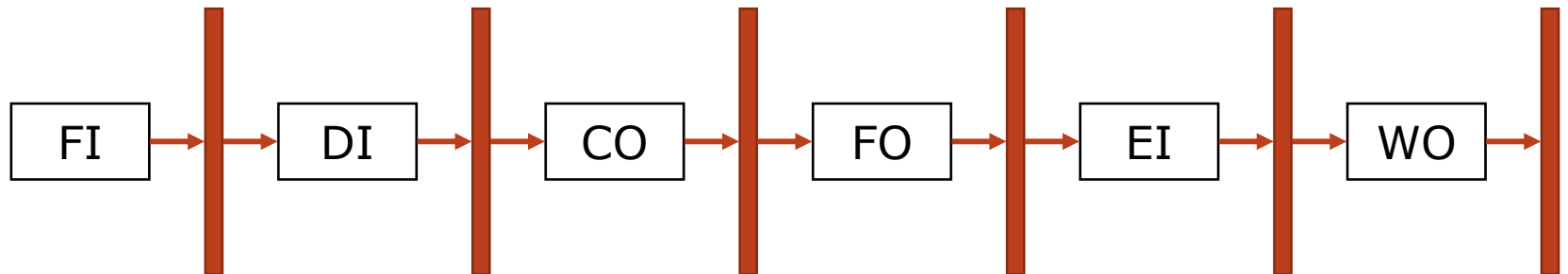
Dopo un transitorio iniziale (in cui si esegue la prima istruzione), viene terminata l'esecuzione di **un'istruzione al termine di ogni fase**



**Transitorio**

# Hardware in più

Nuovi registri fra ogni stadio del pipeline per memorizzare i risultati parziali di ogni ciclo e separare le fasi.



# Tempo minimo di avanzamento

$k$  = numero di fasi (6 nell'esempio precedente)

$t_i$  = tempo di esecuzione della fase  $i$ , con  $1 \leq i \leq k$

$d$  = tempo per trasmettere un segnale da una fase alla successiva

**Tempo di ciclo:  $t = \max_i [t_i] + d$**

Tempo di ciclo è il **tempo minimo** per avanzare un'istruzione da una fase alla successiva

# Tempo di esecuzione

Tempo totale per eseguire N operazioni:

- **Senza pipeline:**  $t_1 = N \cdot k \cdot t$
- **Con pipeline a k fasi:**  $t_k = (k + (N - 1)) \cdot t$
- **Transitorio:** primi k cicli (con pipeline vuota)
- **Regime:** un'istruzione eseguita ad ogni ciclo (caso ideale)

# Fattore di speed-up

**Fattore di speedup  $S_k$** : indica quanto più veloce l'uso di un pipeline rende l'esecuzione di un programma (ovvero di quanto viene ridotto il tempo di esecuzione), rispetto al caso in cui il pipeline sia assente:

$$S_k = \frac{T_1}{T_k} = \frac{N \cdot k \cdot t}{k \cdot t + (N - 1) \cdot t} = \frac{N \cdot k}{k + N - 1}$$

Con  $N \rightarrow +\infty$ , lo speedup tende a  $k$

$$S_k = \lim_{N \rightarrow +\infty} \frac{N \cdot k}{k + N - 1} = k$$

# Esempio

- Si consideri una pipeline con  $k = 4$  fasi e tempo di ciclo  $t = 20\text{ns}$  (assumiamo che il tempo di trasmissione sia nullo,  $d=0$ ).

- Il tempo richiesto per l'esecuzione di un programma di  $N$  istruzioni con la tecnica del pipelining è pari a:

$$T_4 = 80 + 20 \cdot (N - 1) \text{ ns}$$

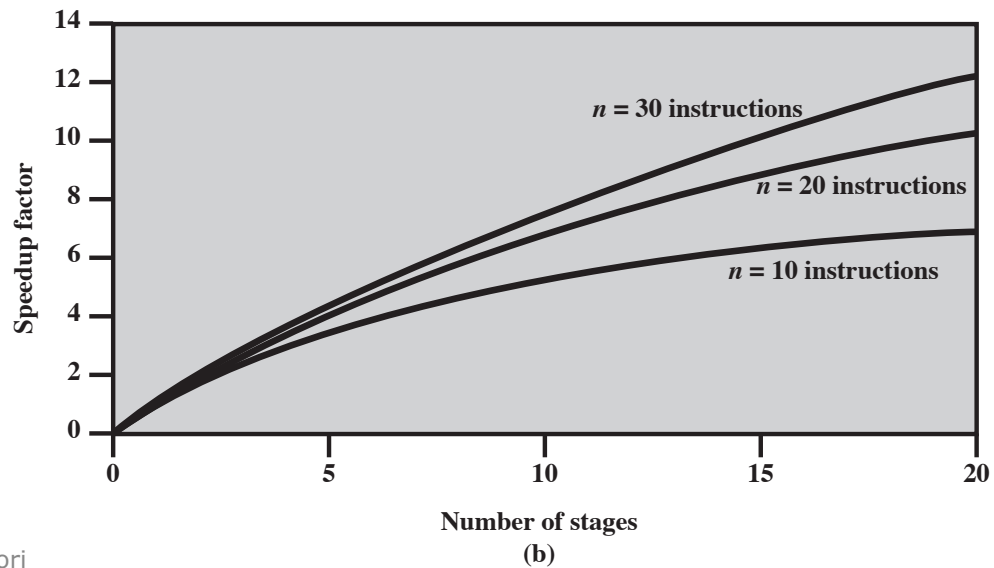
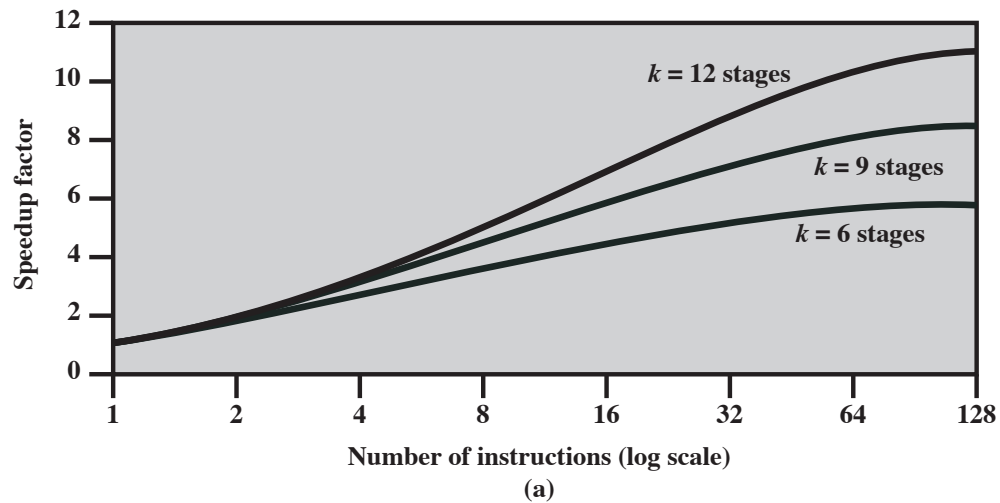
- Senza pipeline il tempo necessario sarebbe:

$$T_1 = 80 \cdot N \text{ ns}$$

- Speedup:  $S_k = T_1/T_4 \rightarrow 4$ .

- Per  $N$  abbastanza grande, il pipeline riduce il tempo di esecuzione di un fattore 4.

# Speed-up



# Inceppamento del pipeline

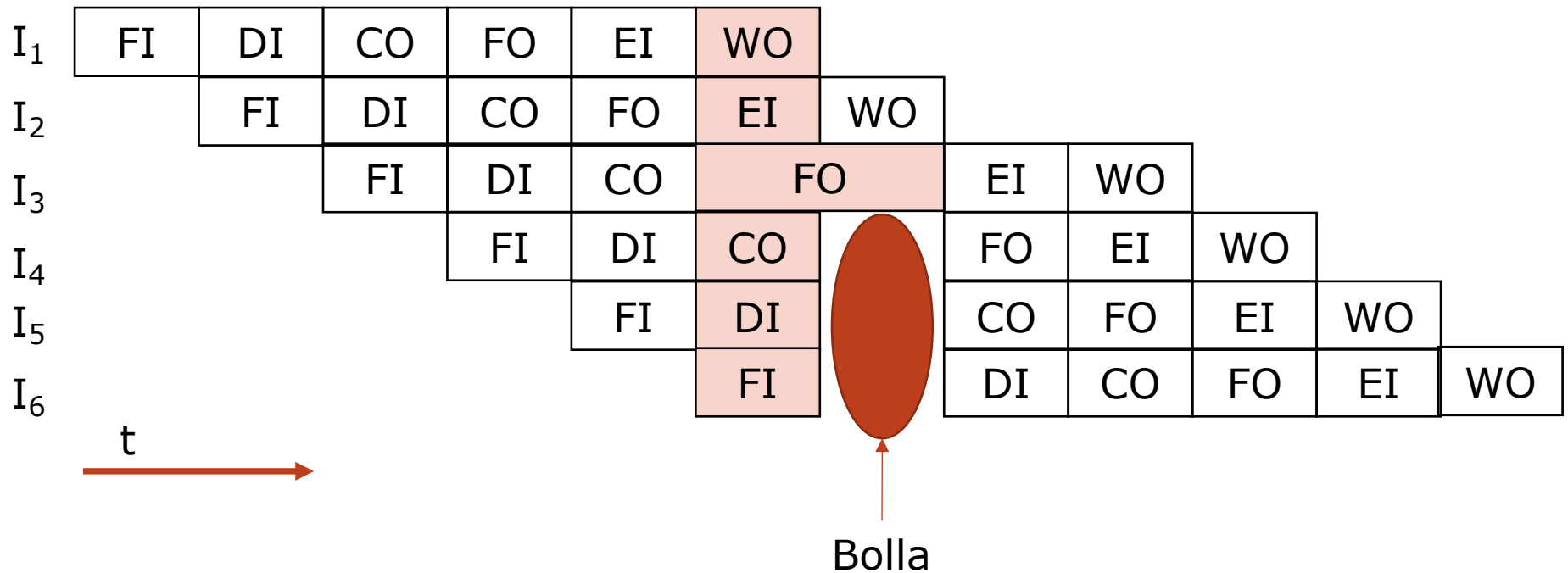
- Il fattore di speedup  $S_k$  è un valore teorico raggiunto solo se il pipeline opera, a regime, avviando sempre, ad ogni tempo di ciclo, una nuova istruzione (e completandone una).
- In realtà il pipeline può incepparsi (**pipeline stall/bubble**) per problemi dovuti a:
  - accessi alla memoria (cache miss)
  - conflitti sulle risorse (resource hazard)
  - dipendenze tra dati (data hazard),
  - salto condizionati (branch hazard);
- Gli stalli riducono il fattore di speedup.



# Cache miss

- Le fasi che accedono alla memoria (fetch, lettura/scrittura operandi) hanno una durata pari alle altre fasi (1 ciclo), **solo se gli accessi alla memoria si risolvono nella cache** (cache hit).
- In caso di cache miss, l'operazione può richiedere 2 o 3 cicli.
- Di conseguenza il pipeline si **inceppa** e l'esecuzione delle istruzioni viene ritardata.

# Cache miss



# Resource hazards

**Resource hazards:** quando due fasi richiedono la stessa risorsa

E.g. quando due fasi richiedono la memoria RAM contemporaneamente

- FO di I1 e FI di I3 accedono alla memoria nello stesso istante
- Fase FI di I3 entra in stallo

	Clock cycle								
	1	2	3	4	5	6	7	8	9
Istruzione	I1	FI	DI	FO	EI	WO			
	I2		FI	DI	FO	EI	WO		
	I3			FI	DI	FO	EI	WO	
	I4				FI	DI	FO	EI	WO

(a) Five-stage pipeline, ideal case

Istruzione	Clock cycle								
	1	2	3	4	5	6	7	8	9
	I1	FI	DI	FO	EI	WO			
	I2		FI	DI	FO	EI	WO		
	I3			Idle	FI	DI	FO	EI	WO
I4					FI	DI	FO	EI	WO

(b) I1 source operand in memory

# Data hazard

- **Data hazard**: quando gli operandi di una istruzione sono i risultati di un'istruzione precedente ancora da terminare.
- L'esecuzione dell'istruzione non può procedere finché il risultato non è pronto e l'esecuzione subisce un ritardo (pipeline stall).

# Data hazard: esempio

- La seconda istruzione utilizza il risultato della prima istruzione
- Il risultato della prima istruzione sarà pronto solo nella fase WO, ma viene richiesto nella fase FO della seconda istruzione

Clock cycle


	1	2	3	4	5	6	7	8	9	10
ADD R0, R0, R1	FI	DI	FO	EI	WO					
SUB R1, R1, R0		FI	DI	Idle		FO	EI	WO		
I3			FI			DI	FO	EI	WO	
I4						FI	DI	FO	EI	WO

# Rimedi contro i data hazard

- I data hazard possono essere evitati dal compilatore con un **riordino delle istruzioni** (per eseguire altre istruzioni prima di quella cui servono i dati);
- Le conseguenze negative di un data hazard possono essere ridotte dal processore, con la tecnica del **by-pass** (detta anche data-forwarding): i risultati prodotti dall'ALU vengono inoltrati allo stadio successivo del pipeline, in contemporanea alla (e senza attendere la) loro memorizzazione:

# Esempio con gcc

```
1 int f1(int a, int b, int c){  
2     int d = a * b;  
3     int e = d * b;  
4     int f = a * c;  
5     int g = e - f;  
6     return g;  
7 }  
8
```

ARM GCC 8.5.0 (linux)  -O1

A     + 

```
1 f1:  
2     str    lr, [sp, #-4]!  
3     mov    ip, r0  
4     mul    r3, r1, r0  
5     mul    lr, r3, r1  
6     mul    r0, r2, ip  
7     sub    r0, lr, r0  
8     ldr    pc, [sp], #4
```

ARM GCC 8.5.0 (linux)  -O2

A     + 

```
1 f1:  
2     mov    ip, r0  
3     mul    r3, r1, r0  
4     mul    r0, r2, ip  
5     mul    r2, r3, r1  
6     sub    r0, r2, r0  
7     bx     lr
```

# By-pass

		Clock cycle									
		1	2	3	4	5	6	7	8	9	10
ADD R0, R0, R1		FI	DI	FO	EI	WO					
SUB R1, R1, R0			FI	DI	Idle	FO	EI	WO			
I3			FI		DI	FO	EI	WO			
I4					FI	DI	FO	EI	WO		



# Control Hazard

- **Control hazard (o branch hazard)**: si verifica nelle istruzioni di salto condizionato, quando (prima di conoscere se il salto verrà effettuato) il pipeline viene alimentato con le istruzioni della diramazione che non sarà intrapresa.
- In tal caso è necessario:
  1. svuotare il pipeline,
  2. annullare gli effetti delle istruzioni che ne hanno percorso indebitamente alcune fasi,
  3. rialimentare il pipeline con le istruzioni dell'altra diramazione.

# Control hazard (2)

- L'istruzione 3 è un branch condizionato verso l'istruzione 15.
- Finché la fase EI dell'istruzione 3 non viene eseguita non è noto quale sarà la prossima istruzione da eseguire e vengono caricate le istruzioni successive.

	Time →							← Branch Penalty						
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO							
Instruction 5					FI	DI	CO							
Instruction 6						FI	DI							
Instruction 7							FI							
Instruction 15								FI	DI	CO	FO	EI	WO	
Instruction 16									FI	DI	CO	FO	EI	WO

# Rimedi contro i control hazard

- Multiple streams
- Delayed branch
- Branch prediction

# Multiple streams (o esecuzione speculativa)

- Se sono disponibili due pipeline distinte (i.e., **stream**), si caricano entrambi i rami del branch condizionato.
- Quando il risultato è noto, si scarta il ramo sbagliato.
- Limiti:
  - Possibili conflitti sull'utilizzo di registri e memoria
  - Ci potrebbero essere altri branch condizionati prima che il branch iniziale venga risolto.

# Delayed branch

- La tecnica prevede che il **processore esegua comunque un'ulteriore istruzione** (successiva a quella di salto) prima di intraprendere il salto:
  - il compilatore può riordinare le istruzioni in modo da collocarne una (da eseguire comunque) dopo ogni istruzione di salto;
  - se non riesce a trovare una istruzione di questo tipo, il compilatore inserisce, dopo l'istruzione di salto, una NOP (comporta un ritardo, ma evita di avviare l'esecuzione di istruzioni che non devono essere eseguite).

# Branch prediction

**Branch prediction:** si predice quale possa essere il ramo da intraprendere.

- Se la scelta è corretta: nessun rallentamento!
- Se la scelta è sbagliata: bisogna svuotare la pipeline e ripartire con il ramo corretto.

# Tipi di branch prediction

**Statici:** la scelta non dipende dalla storia di esecuzione

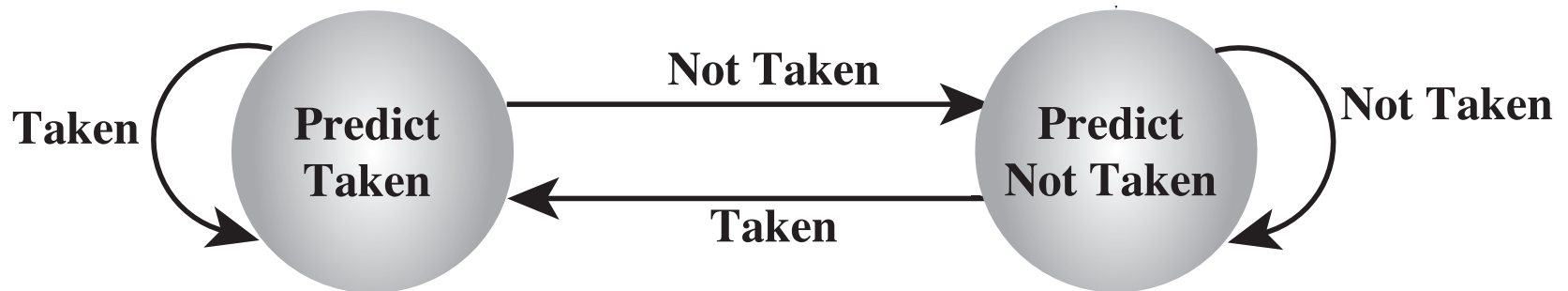
- Sempre "esegui branch"
- Sempre "non eseguire branch"
- La scelta "esegui/non esegui" dipende dell'opcode

**Dinamici:** la scelta dipende dalla storia di esecuzione

- La scelta "intrapreso/non intrapreso" dipende da alcuni bit di stato
- Branch History Table

# Predittore dinamico con 1 bit di stato

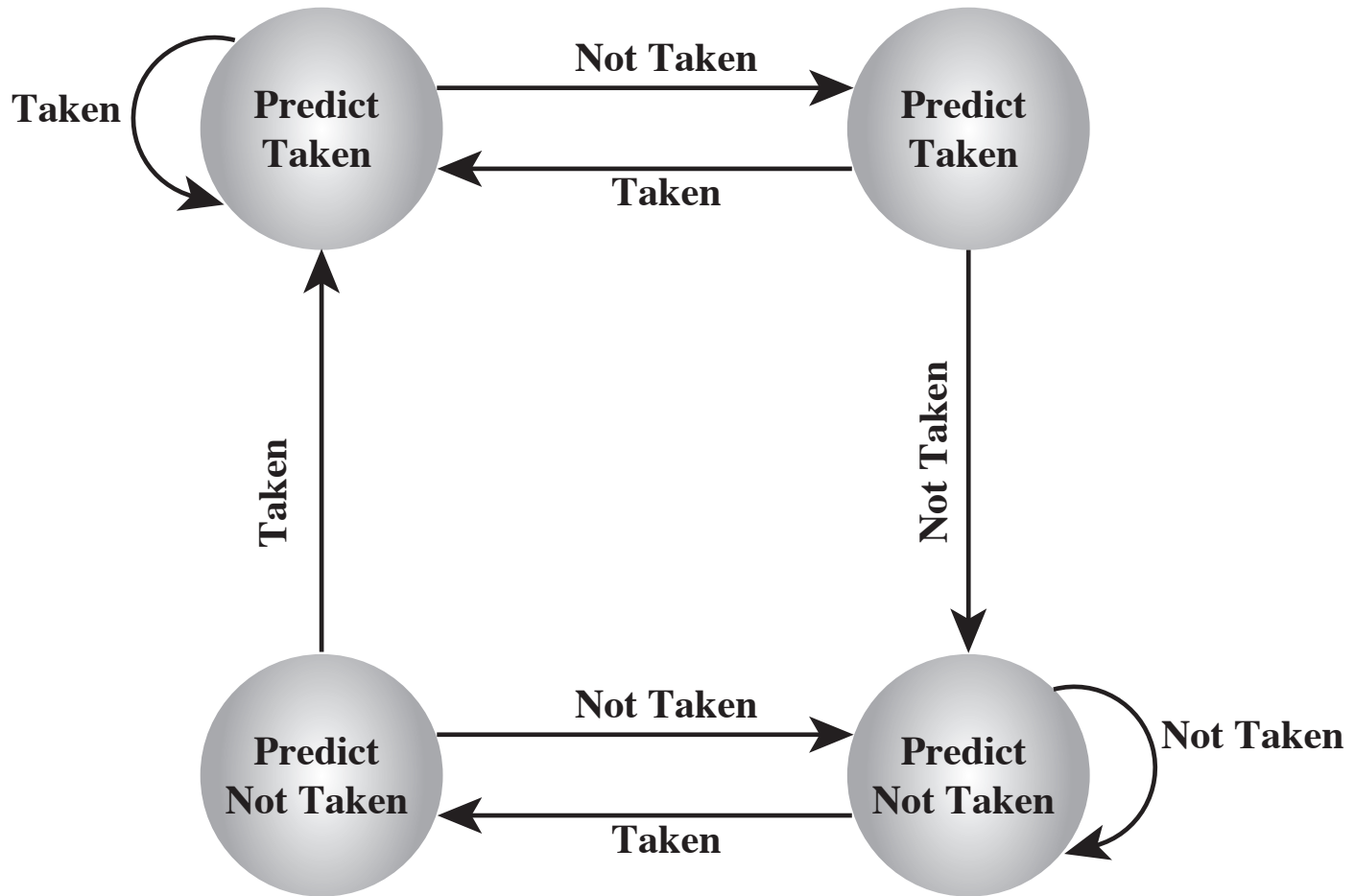
- I predittori dinamici servono in caso di ripetute esecuzioni di un branch condizionato (e.g., for/while loop)
- Si associa uno stato ad ogni branch condizionato
- Se lo stato è di 1 bit, il predittore di un branch si ricorda solo il risultato dell'ultima esecuzione





# Predittore dinamico con 2 bit di stato

- Il passaggio tra una predizione all'altra è più stabile



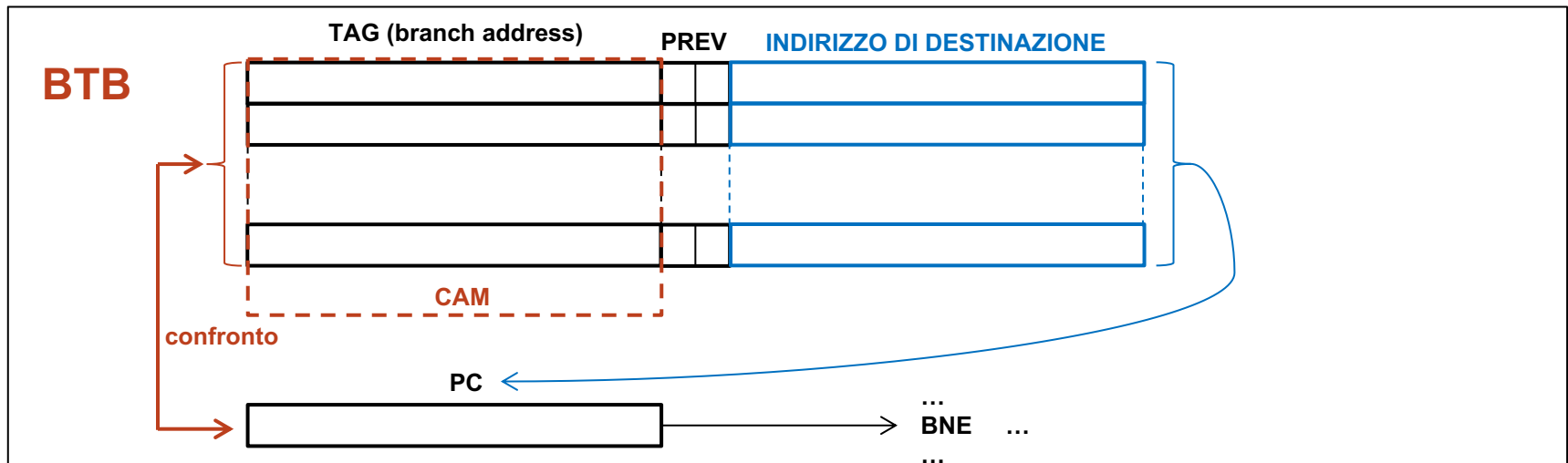
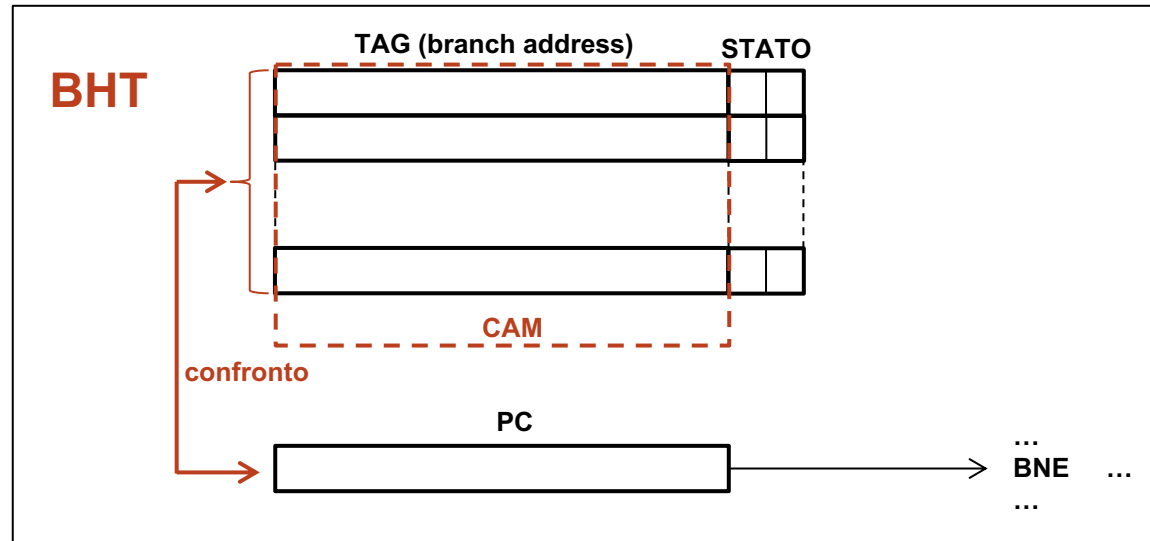
# Branch history table

- Lo stato di un branch condizionato viene salvato nella **Branch History Table** (BHT)
- La BHT è una cache contenente (su CAM) gli indirizzi delle istruzioni di salto e lo stato di predizione
  1. il PC viene confrontato con gli indirizzi nella CAM;
  2. Se presente, la predizione viene fatta con lo stato di predizione
  3. Se assente, si inserisce una nuova riga nella tabella
  4. Quando il risultato del branch è noto, si aggiorna lo stato di predizione.

# Branch target buffer

- Il BHT non ha informazioni sull'indirizzo del branch condizionato:
  - Quindi l'istruzione del branch va parzialmente eseguita per calcolare l'indirizzo
- Il **Branch Target Buffer** (BTB) è un BHT in cui ogni riga include anche l'indirizzo del branch.

# Branch History Table / Target Buffer



# Uso del BTB

