

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.



Salto condizionato

salto
condizionato
a Istruzione 15

Instruction 2

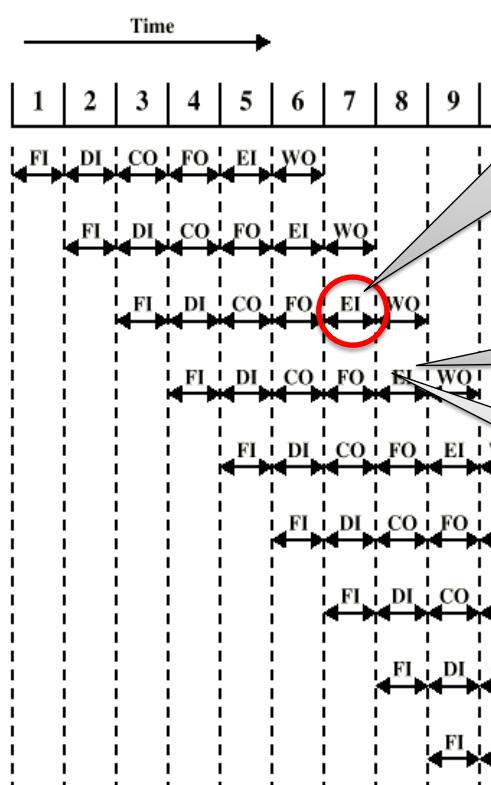
Instruction 4

Instruction 5

Instruction 7

Instruction 8

Instruction 9

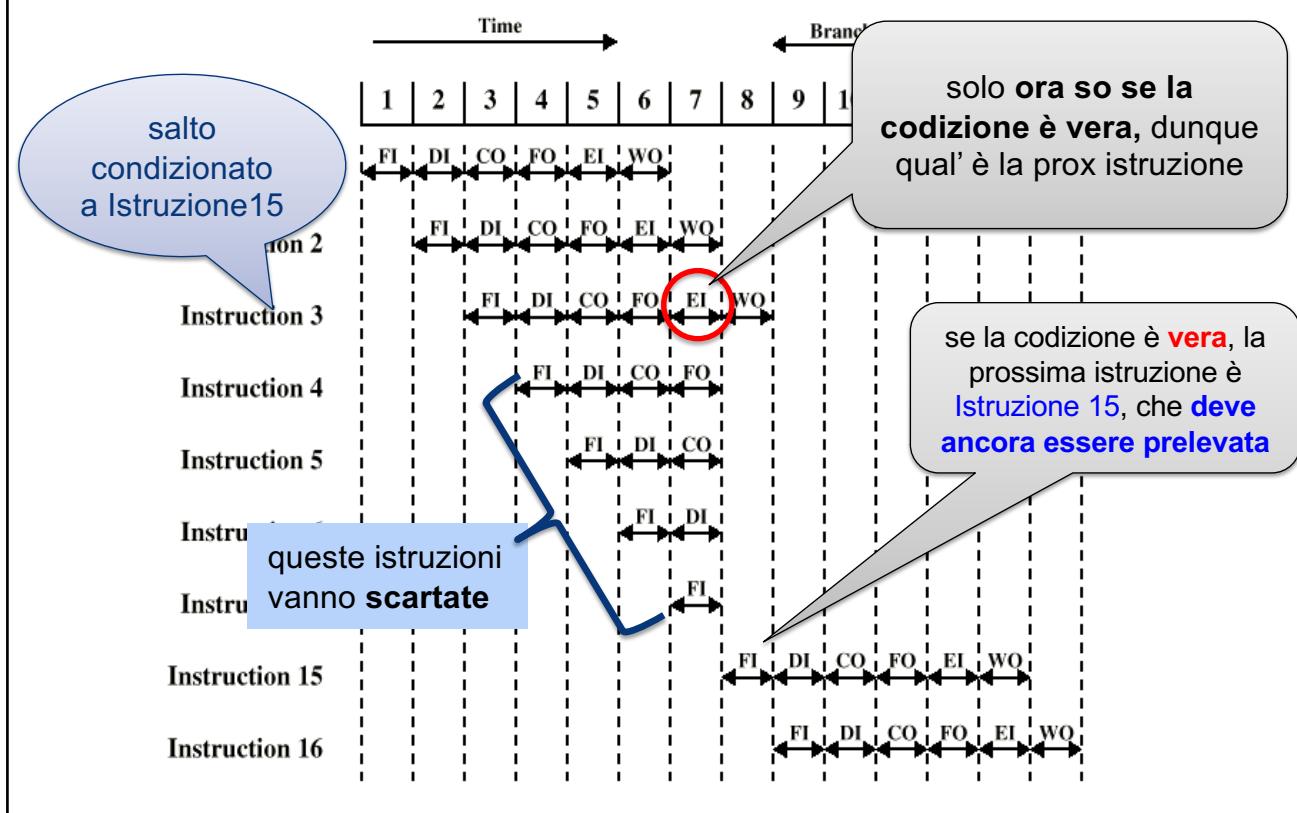


solo ora so se la codizione è vera, dunque qual' è la prox istruzione

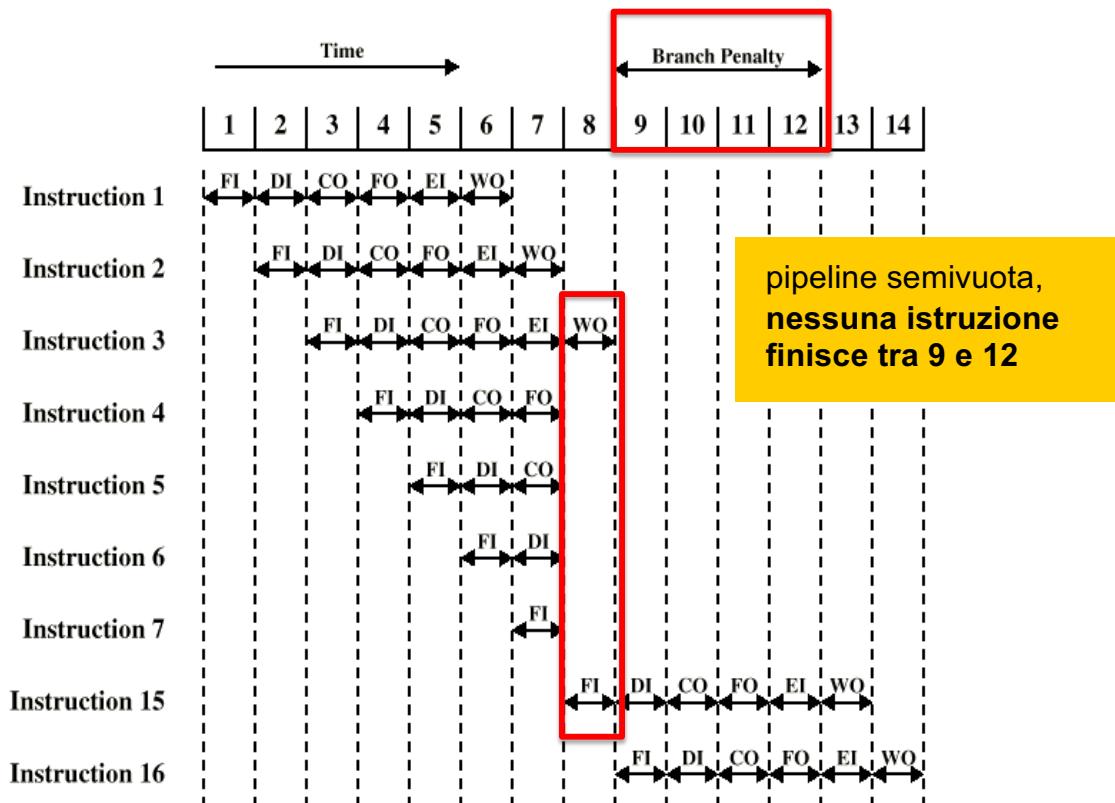
se la codizione è falsa, il prossimo EI è di Istruzione 4

se la codizione è vera, la prossima istruzione è Istruzione 15, che deve ancora essere prelevata

Salto condizionato



Salto condizionato



Esempio: salto condizionato

ADD R1 R2
BEZ TARGET
istr
...
target

FI	DI	CO	FO	EI	WO			
	FI	DI	CO	FO	EI	WO		
		FI						

qui si sa che forse c'è un salto
finché non si sa se/dove saltare
si mette pipeline in stallo

qui si sa se è vera la condizione di salto

qui si conosce l'indirizzo a cui saltare

ADD R1 R2
BEZ TARGET
istr
...
target

FI	DI	CO	FO	EI	WO			
	FI	DI	CO	FO	EI	WO		
		X						

supponiamo **condizione vera**:
si scarta istruzione pre-fetched
e si ricomincia con istruzione **target**

qui si sa se è vera la condizione di salto

qui si conosce l'indirizzo a cui saltare

Esempio: salto condizionato

ADD R1 R2
BEZ TARGET
istr
...
target

	FI	DI	CO	FO	EI	WO			
		FI	DI	CO	FO	EI	WO		
			FI	#		DI	CO	FO	EI

qui si sa se è vera la condizione di salto

supponiamo **condizione falsa**:
riprendo dopo lo stalllo con
l'istruzione pre-fetched

qui si conosce l'indirizzo a cui saltare

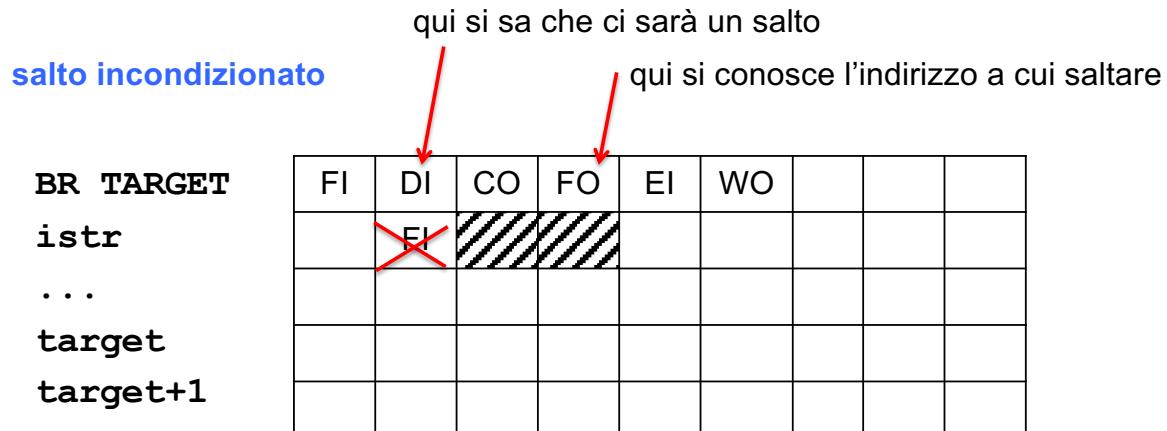
salto incondizionato
BR TARGET
istr
...
target
target+1

	FI	DI	CO	FO	EI	WO			

qui si sa che ci sarà un salto

qui si conosce l'indirizzo a cui saltare

Esempio



ormai ha prelevato l'istruzione errata,
quindi mette in stallo e scarta **istr**

Esempio



ormai ha prelevato l'istruzione errata
quindi mette in stallo e scarta **istr**
ricomincia con istruzione **target**

dipendenza dai controlli

- Uno dei maggiori problemi della progettazione della pipeline è **assicurare un flusso regolare di istruzioni**
 - violato da salti condizionati, salti non condizionati, *chiamate e ritorni da procedure*
 - se la fase fetch ha caricato un'istruzione errata, va scartata
 - queste istruzioni sono circa il 30% del totale medio di un programma

Soluzioni:

- **mettere in stallo** la pipeline finché non si è calcolato l'indirizzo della prossima istruzione. **semplice ma inefficiente**
- individuare le istruzioni critiche e aggiungere un'apposita **logica di controllo**. si **complica il compilatore e hardware** specifico

Soluzioni per salti condizionati

1. flussi multipli (*multiple streams*)

- replica la prima parte della pipeline, EI esclusa, per entrambi i rami possibili

```
istr i → se cond
    istruzione n
else
    istruzione i+1      inserisce nella pipeline sia
                        istruzione n che istruzione i+1
```

brute-force

- conflitti di accesso alle risorse tra i due stream
- se istruzione n (o i+1) contiene un salto aggiunge ulteriori stream

Soluzioni per salti condizionati

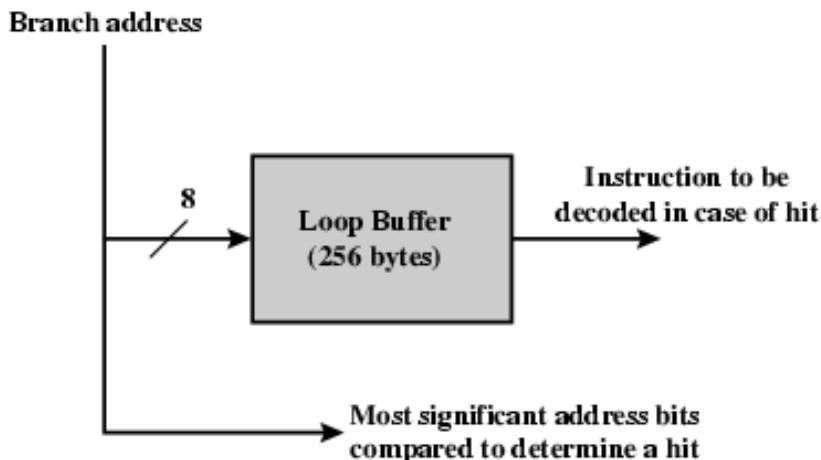
1. flussi multipli (*multiple streams*)
 - replica la prima parte della pipeline, EI esclusa, per entrambi i rami possibili
2. prefetch anche dell'istruzione target
 - anticipa il fetch dell'istruzione target **oltre a** quella successiva al salto
 - se il salto è preso, trova l'istruzione già caricata
 - in ogni caso una parte della pipeline deve essere scartata

Soluzioni per salti condizionati

1. flussi multipli (*multiple streams*)
2. prefetch anche dell'istruzione target
3. buffer circolare (*loop buffer*)
 - è una memoria piccola e molto veloce che **mantiene le ultime n istruzioni prelevate**
 - in caso di salto l'hardware **controlla se l'istruzione target è tra quelle già dentro il buffer**, così da evitare il fetch
 - utile **in caso di loop**, specie se il buffer contiene tutte le istruzioni nel loop, così vengono prelevate dalla memoria una sola volta
 - può essere **accoppiato al pre-fetch**: riempio il buffer con un pò di istruzioni sequenzialmente successive alla corrente. Per molti if-then-else i due rami sono istruzioni vicine, quindi probabilmente entrambe già nel buffer

Buffer circolare (senza prefetch)

- buffer senza prefetch, capienza 256 bytes, indirizzato a byte
- dato l'*indirizzo target* di salto/branch, controllo se c'è nel buffer:
 - gli **8 bit meno significativi** sono usati come **indice nel buffer**
 - gli altri **bit più significativi** si usano per controllare **se la destinazione del salto sta già nel buffer**



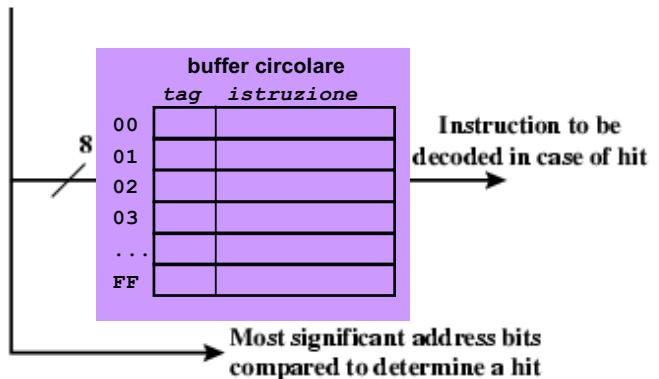
Buffer circolare (senza prefetch)

Memoria

...

```
0AF700 SUB CX,CX
0AF701 MOV AX,01AF90
0AF702 CMP AX,000543
0AF703 BNZ 0AF701
...
0AF7FF ADD AX,CX
0AF800 SUB BX,CX
0AF801 CMP BX,AX
0AF802 BNZ 0AF800
0AF803 INC BX
...
```

Branch address



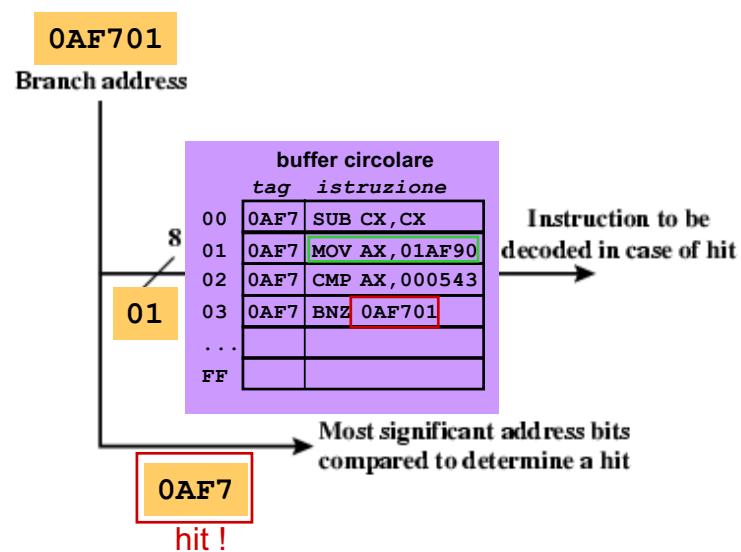
Buffer circolare (senza prefetch)

Memoria

```

...
0AF700 SUB CX,CX
0AF701 MOV AX,01AF90
0AF702 CMP AX,000543
0AF703 BNZ 0AF701
...
0AF7FF ADD AX,CX
0AF800 SUB BX,CX
0AF801 CMP BX,AX
0AF802 BNZ 0AF800
0AF803 INC BX
...

```



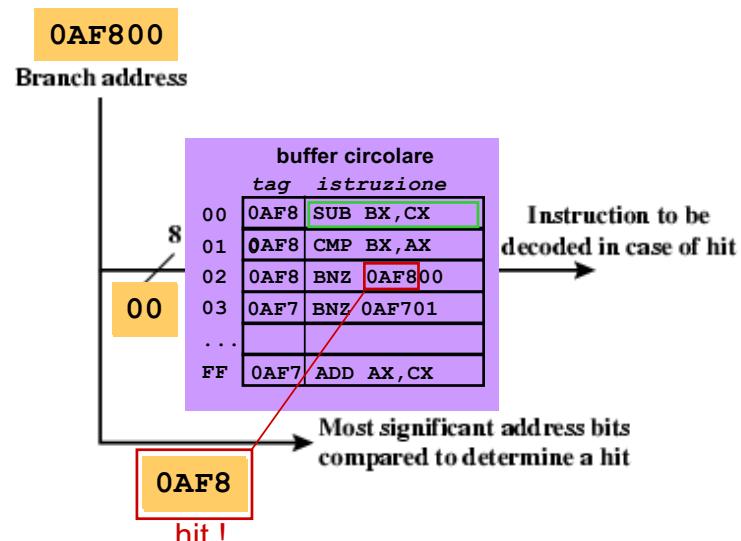
Buffer circolare (senza prefetch)

Memoria

```

...
0AF700 SUB CX,CX
0AF701 MOV AX,01AF90
0AF702 CMP AX,000543
0AF703 BNZ 0AF701
...
0AF7FF ADD AX,CX
0AF800 SUB BX,CX
0AF801 CMP BX,AX
0AF802 BNZ 0AF800
0AF803 INC BX
...

```



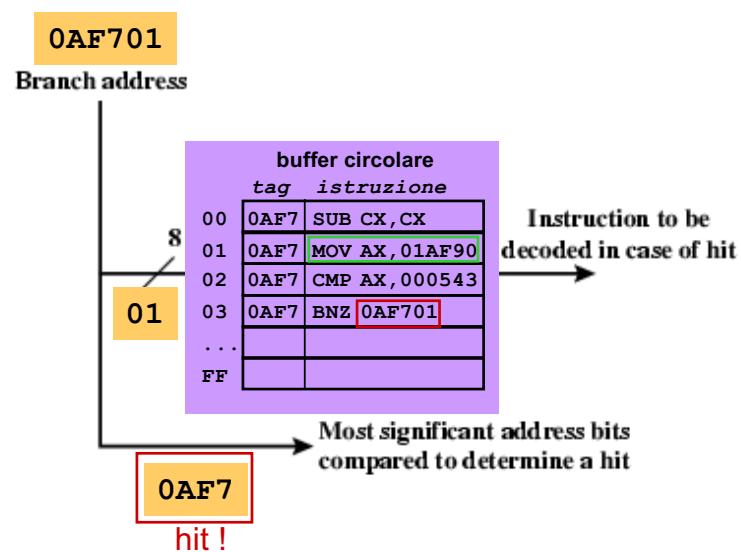
Buffer circolare (senza prefetch)

Memoria

```

...
0AF700 SUB CX,CX
0AF701 MOV AX,01AF90
0AF702 CMP AX,000543
0AF703 BNZ 0AF701
...
0AF7FF ADD AX,CX
0AF800 SUB BX,CX
0AF801 CMP BX,AX
0AF802 BNZ 0AF800
0AF803 INC BX
...

```



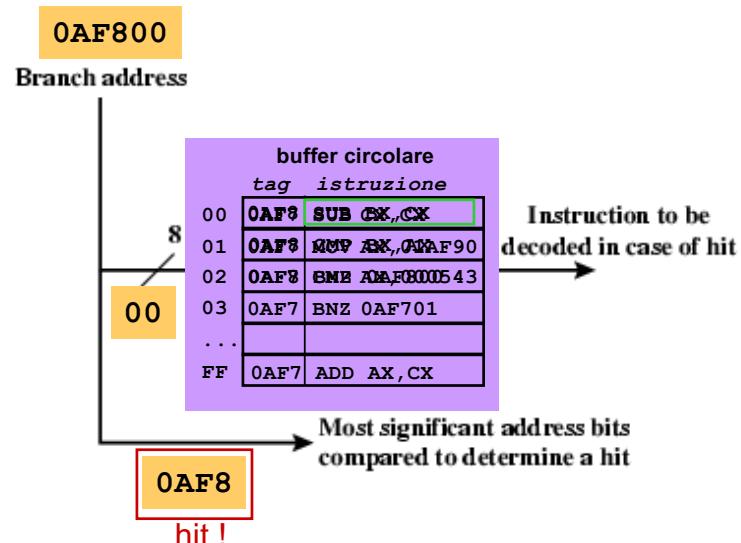
Buffer circolare (senza prefetch)

Memoria

```

...
0AF700 SUB CX,CX
0AF701 MOV AX,01AF90
0AF702 CMP AX,000543
0AF703 BNZ 0AF701
...
0AF7FF ADD AX,CX
0AF800 SUB BX,CX
0AF801 CMP BX,AX
0AF802 BNZ 0AF800
0AF803 INC BX
...

```



Soluzioni per salti condizionati

1. flussi multipli (*multiple streams*)
2. prefetch dell'istruzione target
3. buffer circolare (*loop buffer*)

4. predizione dei salti

- cerco di predire se il salto sarà intrapreso o no

Varie possibilità:

- previsione di saltare **sempre**
 - previsione di non saltare **mai** (*molto usato*)
 - 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
- ad ogni istruzione di salto condizionato associo **1 (o 2) bit** per ricordare la storia recente dell'istruzione, i.e. **se l'ultima (e la penultima) volta il salto è stato preso**
- bit memorizzati in una locazione temporanea ad accesso molto veloce

Approcci dinamici di predizione

associo 1 bit ad ogni istruzione di salto

- ricorda come è andata l'ultima volta, **predico di comportarsi nello stesso modo**
- se bit è **1** predico di **saltare**
- se bit è **0** predico di **non saltare**
- se ho **sbagliato** predizione **inverto il bit**

**a regime:
2 errori per ciclo**

esempio:

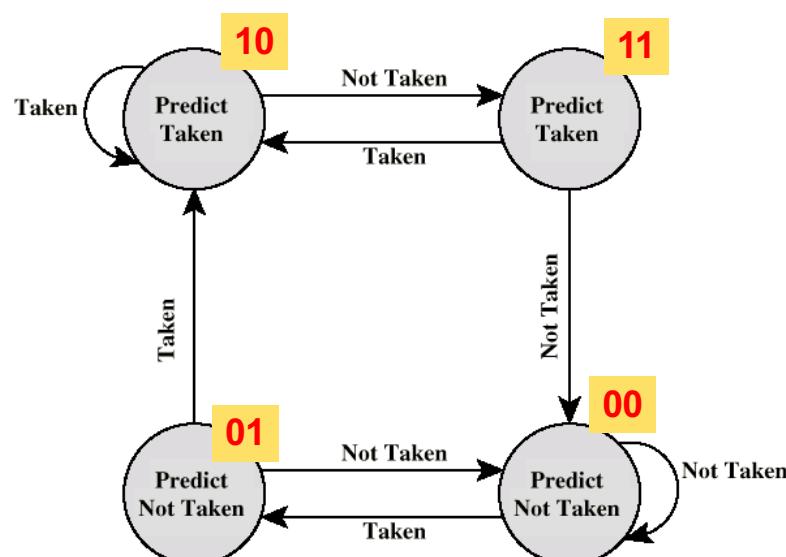
.....
LOOP:

.....
.....
BNZ LOOP

- dopo la prima esecuzione del ciclo, **in uscita dal ciclo**, il bit assegnato a BNZ LOOP è **0** perché il salto **non è stato preso**
- quando si rientra nello stesso ciclo,
 - si avrà **un errore alla prima iterazione** (il bit era a 0, invece prendo il salto)
 - le successive predizioni saranno giuste (l'entrata ha portato il bit a 1)
 - quando **si esce dal ciclo si fa un ulteriore errore** di predizione (e si rimette il bit a 0)

Predizione dinamica con 2 bit

- 2 bit per ricordare come è andata la predizione degli ultimi due salti
- per invertire la predizione ci vogliono **2 errori consecutivi**
- in questo modo a regime fa un solo errore per ciclo

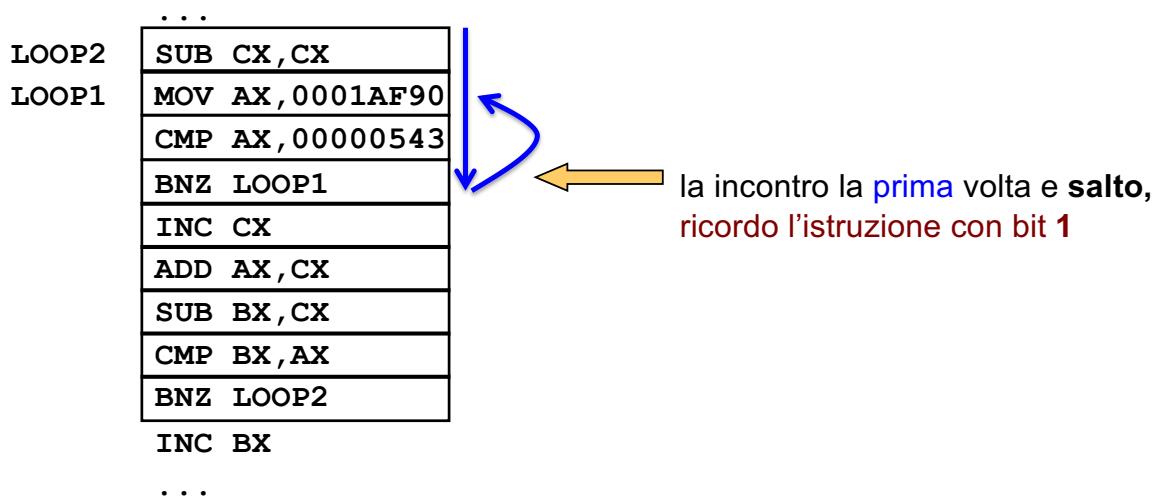


Predizione dinamica 1/2 bit

- per **ogni istruzione di salto** condizionato uso **1/2 bit**
 - per ricordare se l'ultima volta che ho eseguito *quella stessa istruzione* il salto è stato fatto o no
- **se incontro di nuovo** quell'istruzione **e** l'ultima volta **aveva provocato il salto**
 - **allora** predico che salterà, quindi **carico la pipeline** con le istruzioni **a partire dalla destinazione** del salto
 - se ho fatto la scelta sbagliata, le istruzioni caricate vengono eliminate

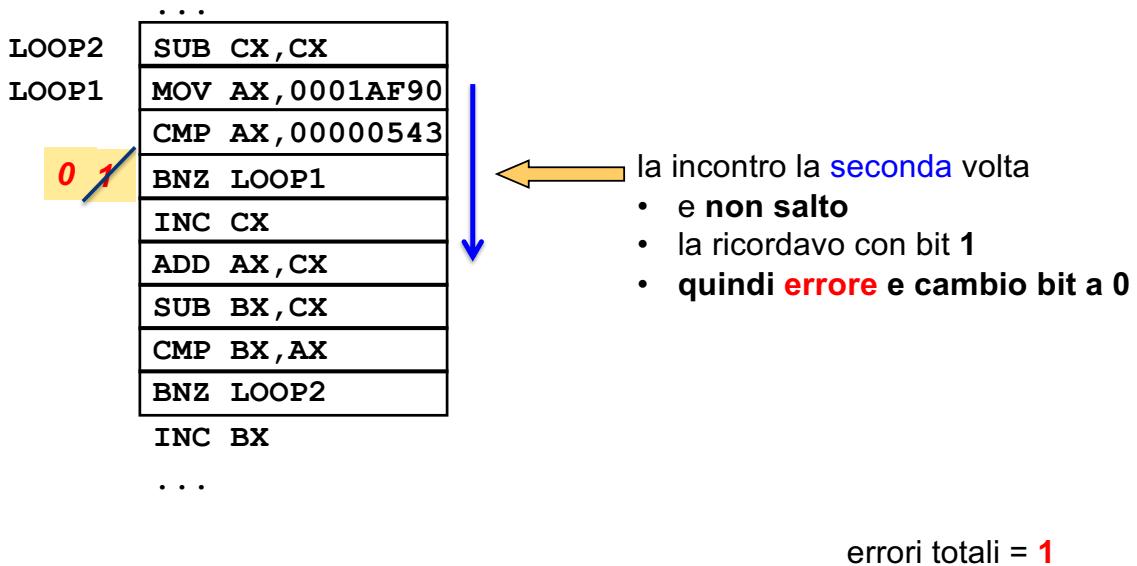
Predizione dinamica 1 bit

due cicli innestati, supponiamo che per entrambi si iteri una sola volta



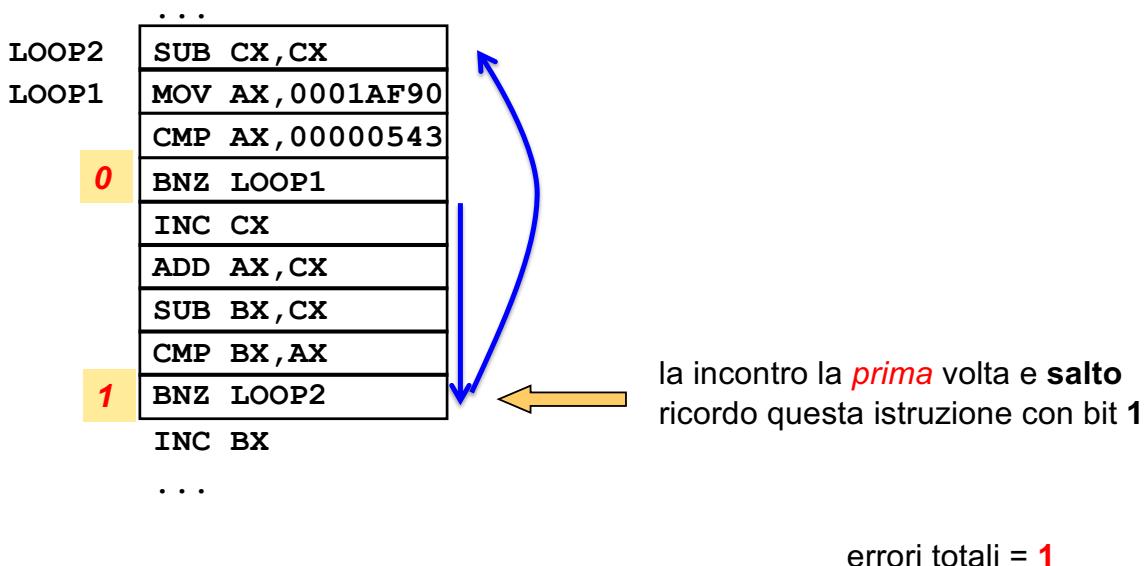
Predizione dinamica 1 bit

due cicli innestati, supponiamo che per entrambi si iteri una sola volta



Predizione dinamica 1 bit

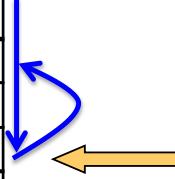
due cicli innestati, supponiamo che per entrambi si iteri una sola volta



Predizione dinamica 1 bit

due cicli innestati, supponiamo che per entrambi si iteri una sola volta

```
...
LOOP2    SUB CX,CX
         MOV AX,0001AF90
         CMP AX,00000543
         BNZ LOOP1
         INC CX
         ADD AX,CX
         SUB BX,CX
         CMP BX,AX
         BNZ LOOP2
         INC BX
         ...
1 / 0
```



la incontro la **terza** volta

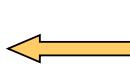
- e **salto**
- la ricordavo con bit **0**
- quindi **errore** e cambio bit a **1**

errori totali = **2**

Predizione dinamica 1 bit

due cicli innestati, supponiamo che per entrambi si iteri una sola volta

```
...
LOOP2    SUB CX,CX
         MOV AX,0001AF90
         CMP AX,00000543
         BNZ LOOP1
         INC CX
         ADD AX,CX
         SUB BX,CX
         CMP BX,AX
         BNZ LOOP2
         INC BX
         ...
0 / 1
```



la incontro la **quarta** volta

- e **non salto**
- la ricordavo con bit **1**
- quindi **errore** e cambio bit **0**

errori totali = **3**

Predizione dinamica 1 bit

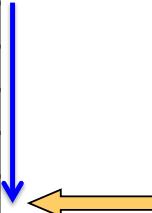
due cicli innestati, supponiamo che per entrambi si iteri una sola volta

```
...
LOOP2    SUB CX,CX
         MOV AX,0001AF90
         CMP AX,00000543
         BNZ LOOP1
         INC CX
         ADD AX,CX
         SUB BX,CX
         CMP BX,AX
         BNZ LOOP2
         INC BX
         ...
         ...
```

0 1

0

0



la incontro la **seconda** volta

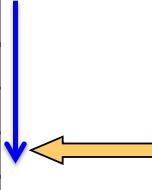
- e **non salto**
- la ricordavo con bit 1
- quindi **errore** e cambio bit a 0

errori totali = 4

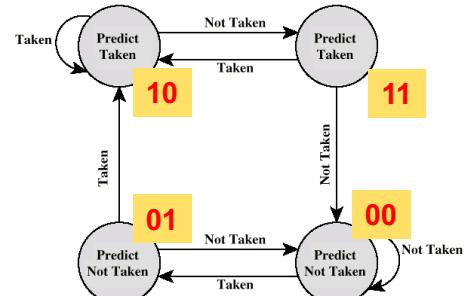
Predizione dinamica 2 bit

errori totali = 0

```
...
LOOP2    SUB CX,CX
         MOV AX,0001AF90
         CMP AX,00000543
         BNZ LOOP1
         INC CX
         ADD AX,CX
         SUB BX,CX
         CMP BX,AX
         BNZ LOOP2
         INC BX
         ...
         ...
```



la incontro la **prima** volta e **salto**
la ricordo con bit **10**



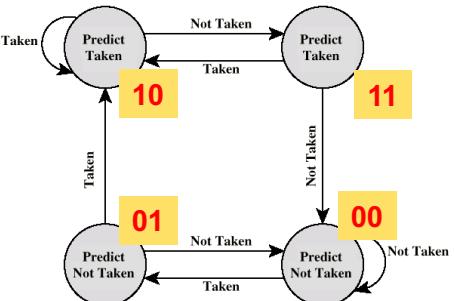
Predizione dinamica 2 bit

errori totali = 1

	...
LOOP2	SUB CX, CX
LOOP1	MOV AX, 0001AF90
	CMP AX, 00000543
11 10	BNZ LOOP1
	INC CX
	ADD AX, CX
	SUB BX, CX
	CMP BX, AX
	BNZ LOOP2
	INC BX

la incontro la seconda volta

- predice salto e **non salto**
- la ricordavo con bit **10**
- **quindi errore e cambio in 11**
(ma resta stessa predizione)

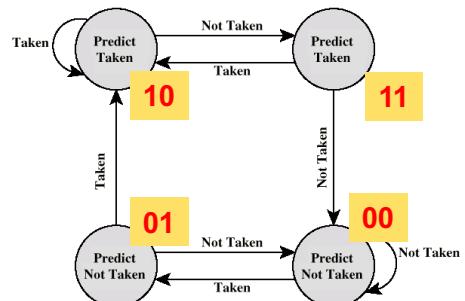


Predizione dinamica 2 bit

errori totali = 1

	...
11	SUB CX,CX
	MOV AX,0001AF90
	CMP AX,00000543
	BNZ LOOP1
	INC CX
	ADD AX,CX
	SUB BX,CX
10	CMP BX,AX
	BNZ LOOP2
	INC BX

la incontro la prima volta e **salto**
la ricordo con bit **10**



Predizione dinamica 2 bit

errori totali = 1 non è errore in più

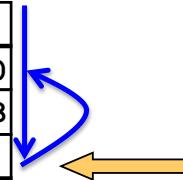
```

...
LOOP2
    SUB CX,CX
    MOV AX,0001AF90
    CMP AX,00000543
    BNZ LOOP1
    INC CX
    ADD AX,CX
    SUB BX,CX
    CMP BX,AX
    BNZ LOOP2
    INC BX
...

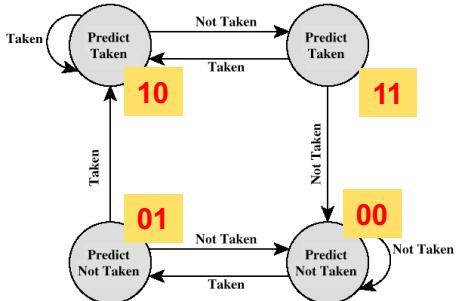
```

10 / 11

10



- la incontro la terza volta
- predice salto e **salto**
 - la ricordavo con bit 11
 - **quindi non errore di predizione**
 - ma cambio bit a 10



Predizione dinamica 2 bit

errori totali = 2

```

...
LOOP2
    SUB CX,CX
    MOV AX,0001AF90
    CMP AX,00000543
    BNZ LOOP1
    INC CX
    ADD AX,CX
    SUB BX,CX
    CMP BX,AX
    BNZ LOOP2
    INC BX
...

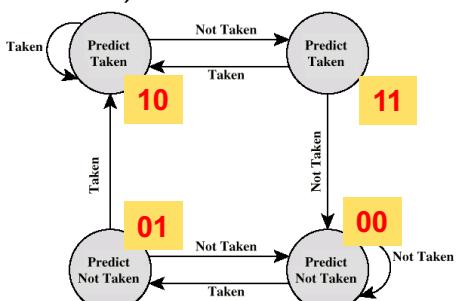
```

11 / 10

10



- la incontro la quarta volta
- predice salto e **non salto**
 - la ricordavo con bit 10
 - **quindi errore di predizione e cambio bit a 11 (ma stessa predizione)**



Predizione dinamica 2 bit

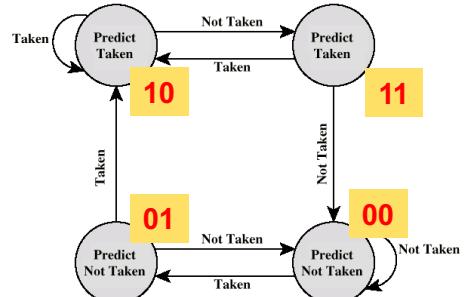
errori totali = 3

```
...
LOOP2    SUB CX,CX
         MOV AX,0001AF90
         CMP AX,00000543
         BNZ LOOP1
         INC CX
         ADD AX,CX
         SUB BX,CX
         CMP BX,AX
         BNZ LOOP2
         INC BX
         ...
11      10
```



la incontro la seconda volta

- predice salto e **non salto**
- la ricordavo con bit 10
- **quindi errore di predizione e cambio bit a 11**



Predizione dinamica 1/2 bit

- per **ogni istruzione di salto condizionato** uso **1/2 bit**
 - per ricordare se l'ultima volta che ho eseguito *quella stessa istruzione* il salto è stato fatto o no
- **se incontro di nuovo** quell'istruzione **e** l'ultima volta **aveva provocato il salto**
 - **allora** predico che salterà, quindi **carico la pipeline** con le istruzioni **a partire dalla destinazione** del salto
 - se ho fatto la scelta sbagliata, le istruzioni caricate vengono eliminate

Soluzioni per salti condizionati

1. flussi multipli (*multiple streams*)
2. prefetch dell'istruzione target
3. buffer circolare (*loop buffer*)
- 4. predizione dei salti**
 - cerco di predire se il salto sarà intrapreso o no

Varie possibilità:

 - previsione di saltare **sempre**
 - previsione di non saltare **mai** (*molto usato*)
 - previsione in base al codice operativo

• bit *taken/not taken*

• tabella della storia dei salti

} *approcci statici*

} *approcci dinamici*

Predizione dinamica

buffer di predizione dei salti

(*branch prediction buffer* o [branch history table](#))

- piccola memoria associata allo stadio fetch della pipeline
- ogni riga della tabella è costituita da 3 elementi:
 1. indirizzo istruzione salto,
 2. i bit di predizione
 3. l'indirizzo destinazione del salto (o l'istruzione destinazione stessa), così quando la predizione è di saltare non devo attendere che si ri-decodifichi il target del salto (se la **previsione è errata** dovrò eliminare le istruzioni errate e caricare quelle corrette)

Predizione dinamica 1/2 bit

buffer di predizione dei salti

(*branch prediction buffer op branch history table*)

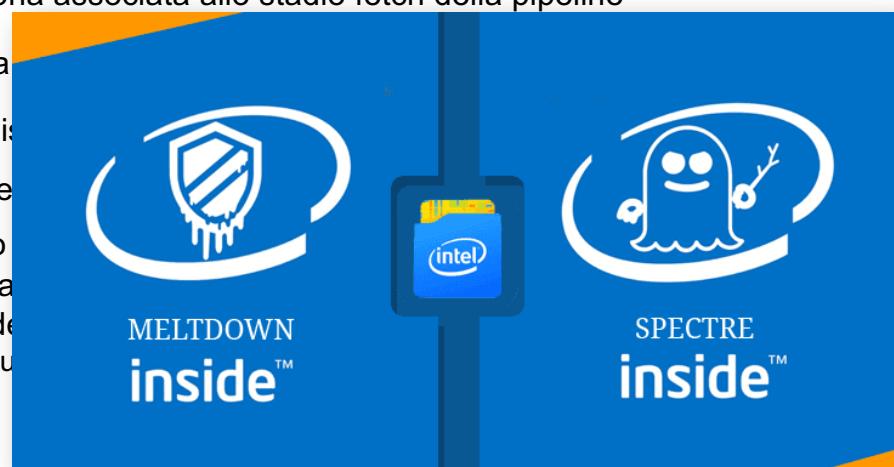
- piccola memoria associata allo stadio fetch della pipeline
- ogni riga della tabella è costituita da 3 elementi:
 1. indirizzo istruzione salto,
 2. i bit di predizione
 3. l'indirizzo destinazione del salto (o l'istruzione destinazione stessa), così quando la predizione è di saltare non devo attendere che si ri-decodifichi il target del salto (se la **previsione è errata** dovrò eliminare le istruzioni errate e caricare quelle corrette)

Predizione dinamica

buffer di predizione dei salti

(*branch prediction buffer o branch history table*)

- piccola memoria associata allo stadio fetch della pipeline
- ogni riga della tabella è costituita da 3 elementi:
 1. indirizzo istruzione salto,
 2. i bit di predizione
 3. l'indirizzo destinazione del salto (o l'istruzione destinazione stessa), così quando la predizione è di saltare non devo attendere che si ri-decodifichi il target del salto (se la **previsione è errata** dovrò eliminare le istruzioni errate e caricare quelle corrette)

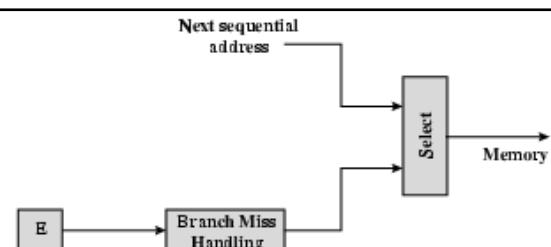


Predizione dinamica 1/2 bit

buffer di predizione dei salti

(branch prediction buffer op *branch history table*)

- questa tabella contiene l'indirizzo dell'istruzione di salto, i bit di predizione (1 o 2), il target del salto
- piccola memoria indicizzata attraverso i bit meno significativi dell'*indirizzo* dell'istruzione di salto
- si usa **nella fase FI**, che conosce l'indirizzo dell'istruzione di salto
- se la **previsione è di saltare**, le istruzioni successive saranno quelle dal target in poi
 - devo attendere che sia calcolato, in fase DI opp FO
 - dopo che l'ho calcolato posso ricordarlo nella *branch history table*
- se la **previsione è errata** devo eliminare le istruzioni errate e caricare quelle corrette



(a) Predict never taken strategy

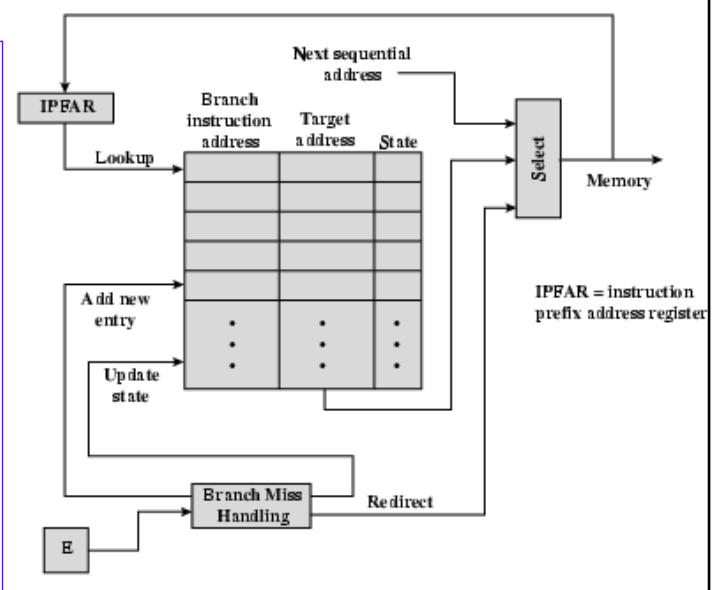


tabella della storia dei salti:

- 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

Soluzioni per salti condizionati

1. flussi multipli (*multiple streams*)
2. prefetch dell'istruzione target
3. buffer circolare (*loop buffer*)
4. predizione dei salti
5. **salto ritardato** (*delayed branch*)
 - Finché non si sa se ci sarà o no il salto (l'istruzione è in pipeline), invece di restare in stallo si può eseguire un'istruzione che non dipende dal salto
 - istruzione successiva al salto: **branch delay slot**
 - Il compilatore cerca di allocare nel *branch delay slot* una istruzione “**opportuna**” (magari inutile ma non dannosa)
 - la CPU esegue **sempre** l'istruzione del *branch delay slot* e solo dopo altera, se necessario, la sequenza di esecuzione delle istruzioni

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 salta	→ MOVE #10,R1 R1 ← 10

	TAR -----

codice ottimizzato dal compilatore

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

TAR	-----

senza ottimizzazione

qui si conosce
condizione e indirizzo del salto

MUL R3, R4
SUB #1 R2
ADD R1, R2
BEZ TAR
MOVE #10, R1
...
TAR ...

	1	2	3	4	5	6	7	8	9	10	11	12	13
MUL R3, R4	FI	DI	CO	FO	EI	WO							
SUB #1 R2		FI	DI	CO	FO	EI	WO						
ADD R1, R2			FI	DI	CO	FO	EI	WO					
BEZ TAR				FI	DI	CO	FO	EI	WO				
MOVE #10, R1					FI								
...													
TAR ...													

qui si sa che è un salto condizionato
quindi inserisco bubble finché non si conosce la condizione

senza ottimizzazione

qui si conosce
condizione e indirizzo del salto

MUL R3, R4
SUB #1 R2
ADD R1, R2
BEZ TAR
MOVE #10, R1
...
TAR ...

	1	2	3	4	5	6	7	8	9	10	11	12	13
MUL R3, R4	FI	DI	CO	FO	EI	WO							
SUB #1 R2		FI	DI	CO	FO	EI	WO						
ADD R1, R2			FI	DI	CO	FO	EI	WO					
BEZ TAR				FI	DI	CO	FO	EI	WO				
MOVE #10, R1					FI					DI	CO	FO	EI WO
...										FI	DI	CO	FO EI WO
TAR ...													

- se **non si salta**
 - continuo con MOVE
 - termino in 12 con **2** cicli di stall
- se **si salta**
 - scarto MOVE e inizio con TAR
 - termino in 13 con **3** cicli persi (uno inutile + 2 stalli)

con delayed branch

qui si conosce
condizione e indirizzo del salto

	1	2	3	4	5	6	7	8	9	10	11	12
SUB #1, R2	FI	DI	CO	FO	EI	WO						
ADD R1, R2		FI	DI	CO	FO	EI	WO					
BEZ TAR			FI	DI	CO	FO	EI	WO				
MUL R3,R4				FI	DI	CO	FO	EI	WO			
MOVE #10, R1					FI							
...												
TAR ...												

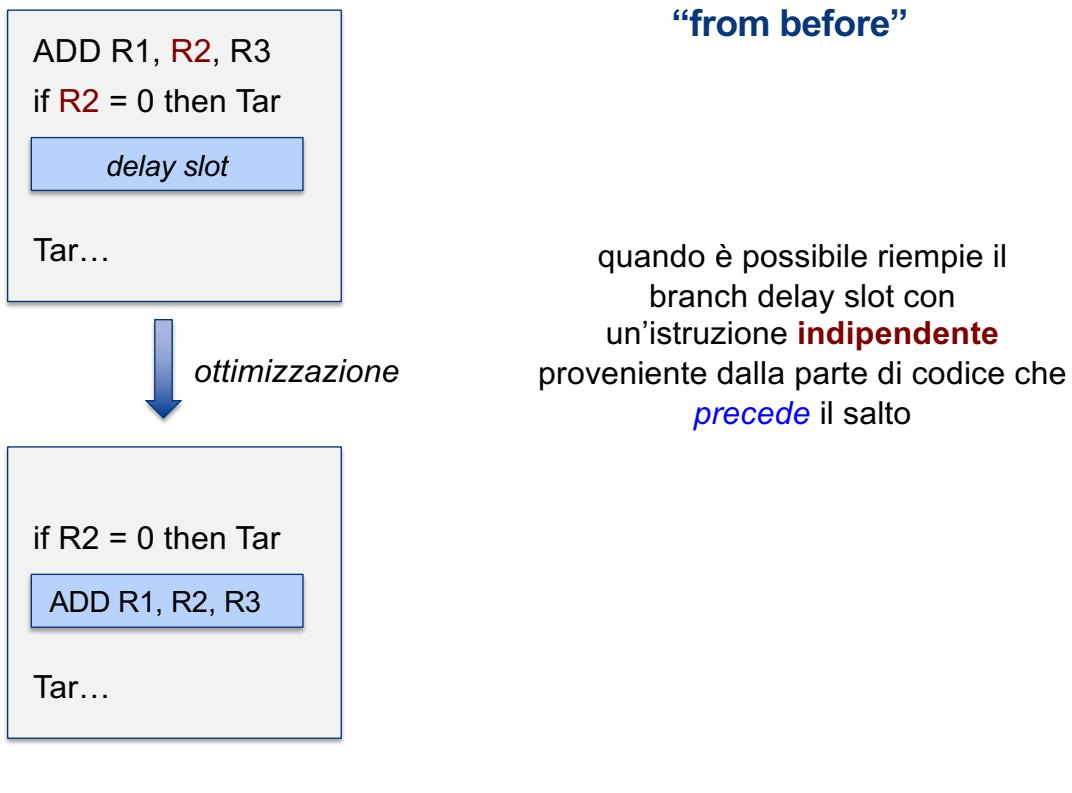
con delayed branch

qui si conosce
condizione e indirizzo del salto

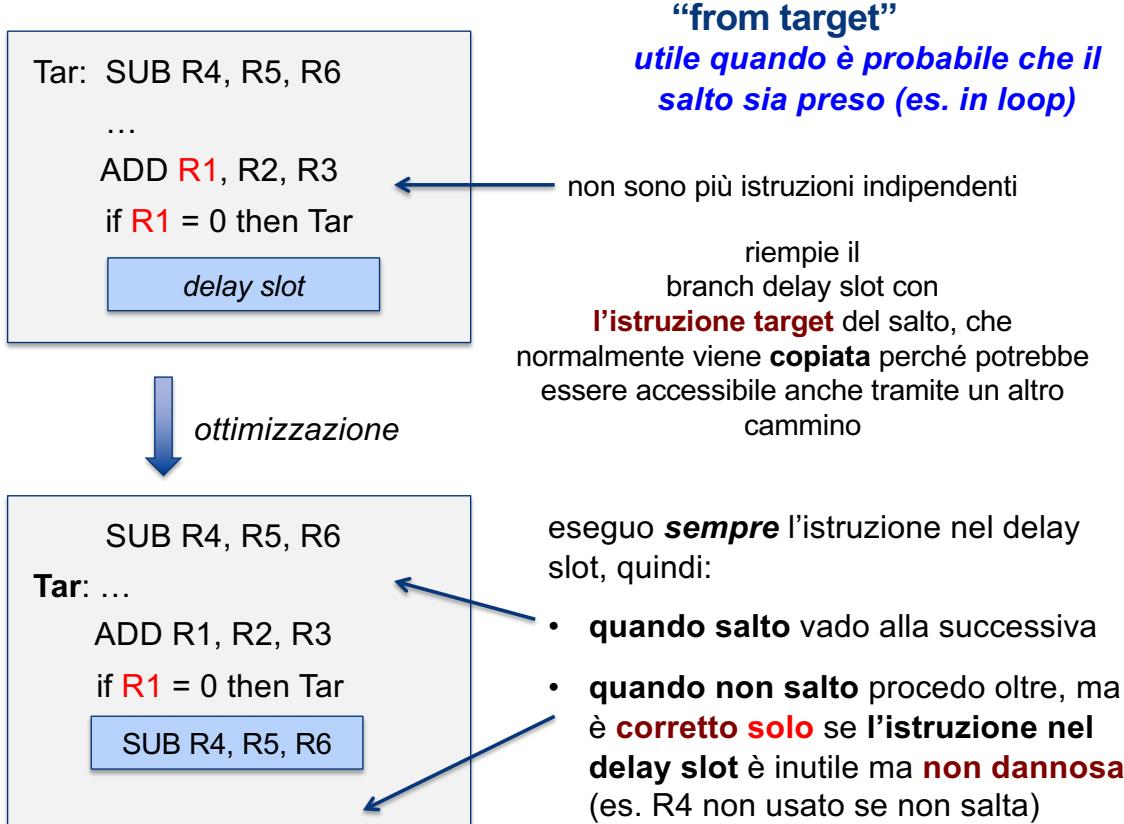
	1	2	3	4	5	6	7	8	9	10	11	12
SUB #1, R2	FI	DI	CO	FO	EI	WO						
ADD R1, R2		FI	DI	CO	FO	EI	WO					
BEZ TAR			FI	DI	CO	FO	EI	WO				
MUL R3,R4				FI	DI	CO	FO	EI	WO			
MOVE #10, R1					X			DI	CO	FO	EI	WO
...												
TAR ...							FI	DI	CO	FO	EI	WO

- se **non si salta**
 - continuo con MOVE
 - termino in 11 con **1** ciclo di stallo
- se **si salta**
 - scarto MOVE e inizio con TAR
 - termino in 12 con **2** cicli persi (uno inutile e 1 stallo)

Salto ritardato (delayed branch)



Salto ritardato (delayed branch)

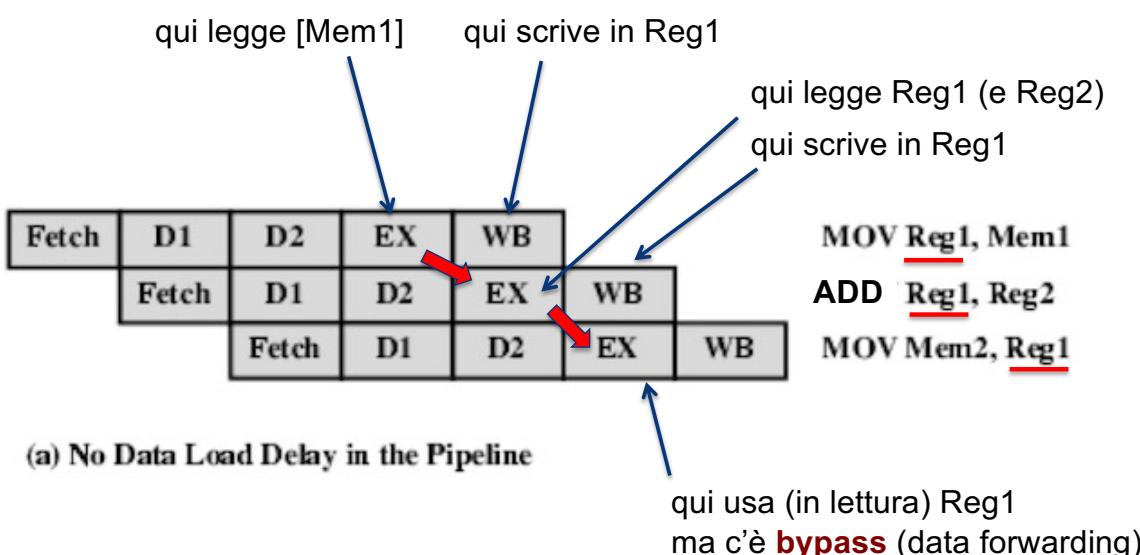


Intel 80486 Pipelining

- 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 spiazzamento)
- Decodifica 2 (D2)
 - Espande i codici operativi in segnali di controllo per l’ALU
 - Calcola gli indirizzi in memoria per i modi di indirizzamento più complessi
- Esecuzione (EX)
 - Operazioni ALU, accesso alla cache (memoria).
- 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

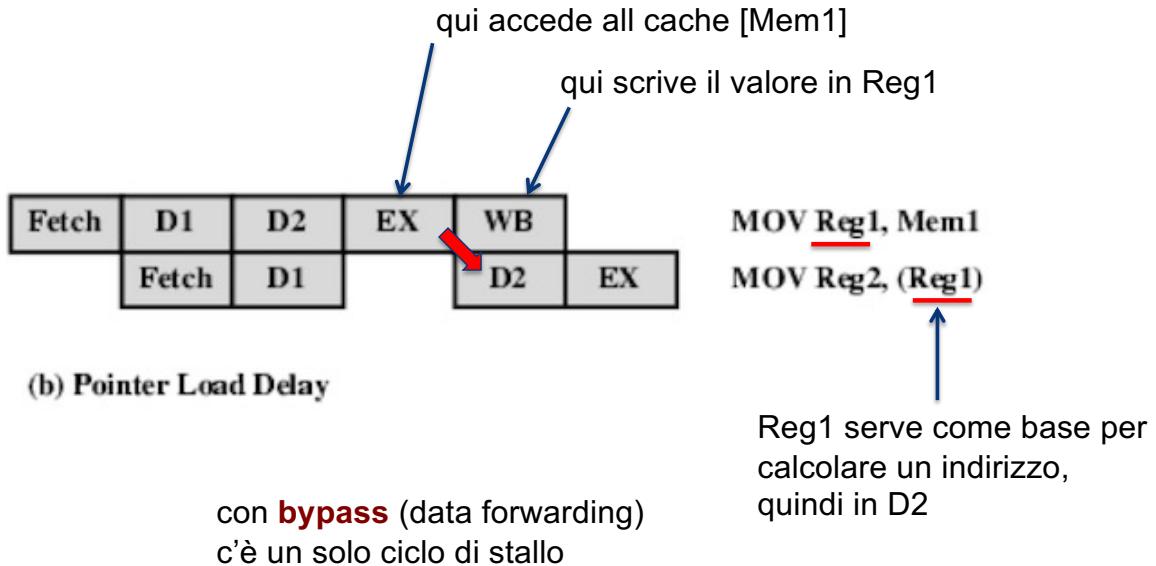
80486 Instruction Pipeline: esempi

accessi consecutivi allo stesso dato non introducono ritardi



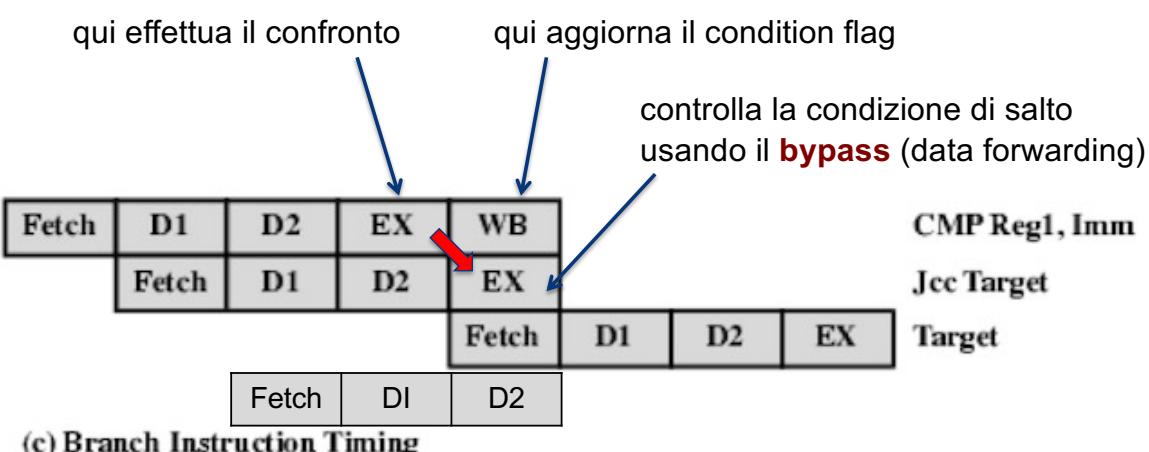
80486 Instruction Pipeline: esempi

ritardo per valori usati per calcolare un indirizzo



80486 Instruction Pipeline: esempi

salto condizionato. Assumiamo venga eseguito



in parallelo fa uno “**speculative fetch**”
dell’istruzione target
(in aggiunta a quello già iniziato per
l’istruzione sequenziale, che sarà scartata).