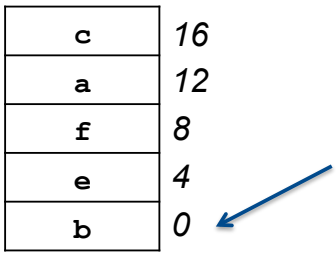
riordino delle istruzioni

programma C con 5 variabili che stanno in memoria

```
a = b + e;
c = b + f;
```

memoria indirizzata al byte (1 word=4 byte)



assumiamo corrisponda a ($t0) così usiamo offset

compilatore produce il codice assembler
- associando i registri alle variabili del programma
- e trasferendo i dati tra la memoria e i registri

```
b - $1    e - $2    a - $3
f - $4    c -$5
```

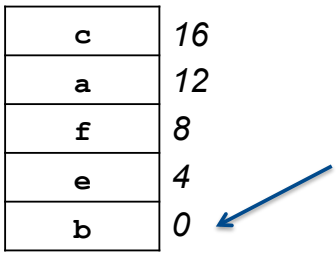
```
lw   $1   0   ($t0)
lw   $2   4   ($t0)
add  $3   $1   $2
sw   $3   12  ($t0)
lw   $4   8   ($t0)
add  $5   $1   $4
sw   $5   16  ($t0)
```

riordino delle istruzioni

programma C con 5 variabili che stanno in memoria

```
a = b + e;
c = b + f;
```

memoria indirizzata al byte (1 word=4 byte)



assumiamo corrisponda a ($t0) così usiamo offset

```
lw   $1   0   ($t0)
lw   $2   4   ($t0)
add  $3   $1   $2
sw   $3   12  ($t0)
lw   $4   8   ($t0)
add  $5   $1   $4
sw   $5   16  ($t0)
```

tutte dipendenze Read after Write

quindi servono degli **stalli**

(e a seconda di come è fatta pipeline con ***data forwarding*** si eliminano i problemi tra `add` e `sw`)

riordino delle istruzioni

programma C con 5 variabili
che stanno in memoria

```
a = b + e;
c = b + f;
```

riordinando le istruzioni si sono eliminate anche le dipendenze `lw - add`

```
lw   $1   0   ($t0)
lw   $2   4   ($t0)
add  $3   $1   $2
sw   $3   12  ($t0)
lw   $4   8   ($t0)
add  $5   $1   $4
sw   $5   16  ($t0)
```

```
lw   $1   0   ($t0)
lw   $2   4   ($t0)
lw   $4   8   ($t0)
add  $3   $1   $2
sw   $3   12  ($t0)
add  $5   $1   $4
sw   $5   16  ($t0)
```

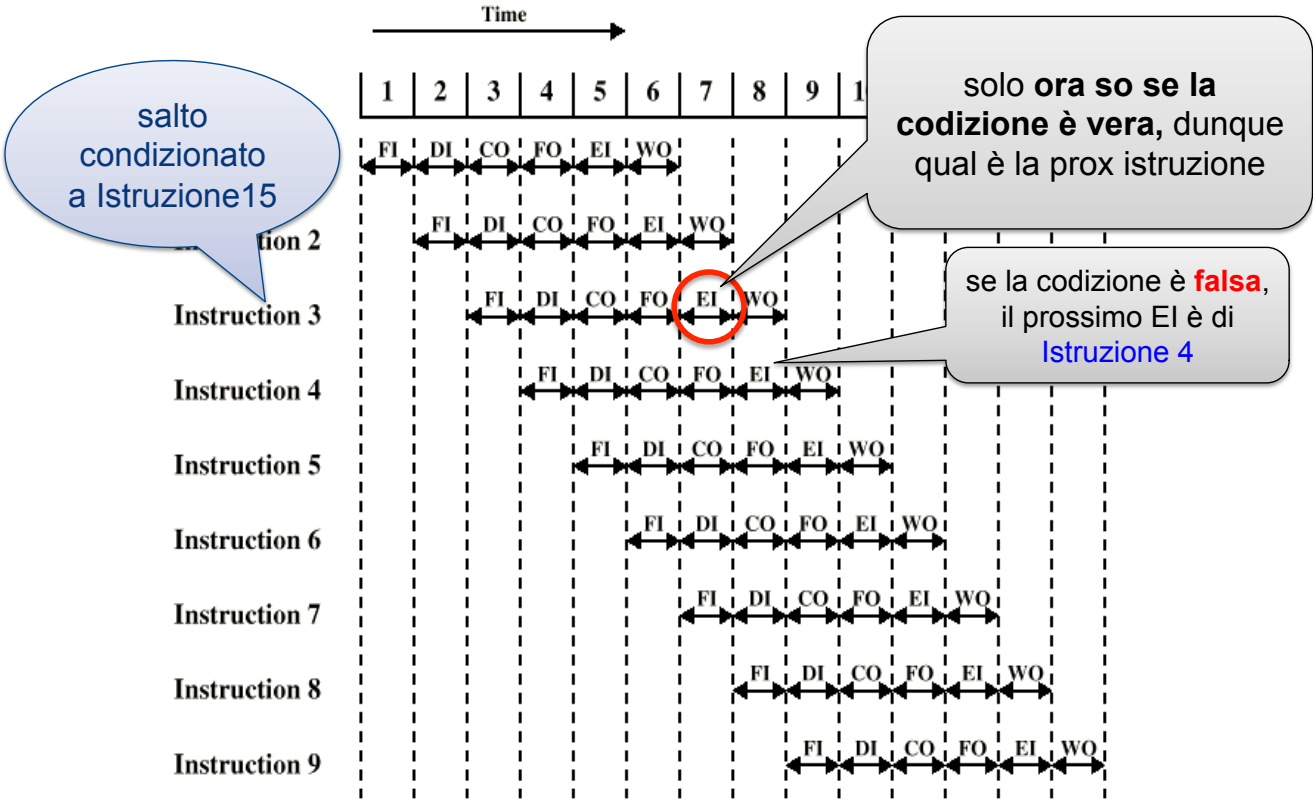
pipeline hazards - criticità

- varie situazioni in cui l'istruzione successiva non può essere eseguita nel ciclo di clock immediatamente successivo (**stallo** – *pipeline bubble*)

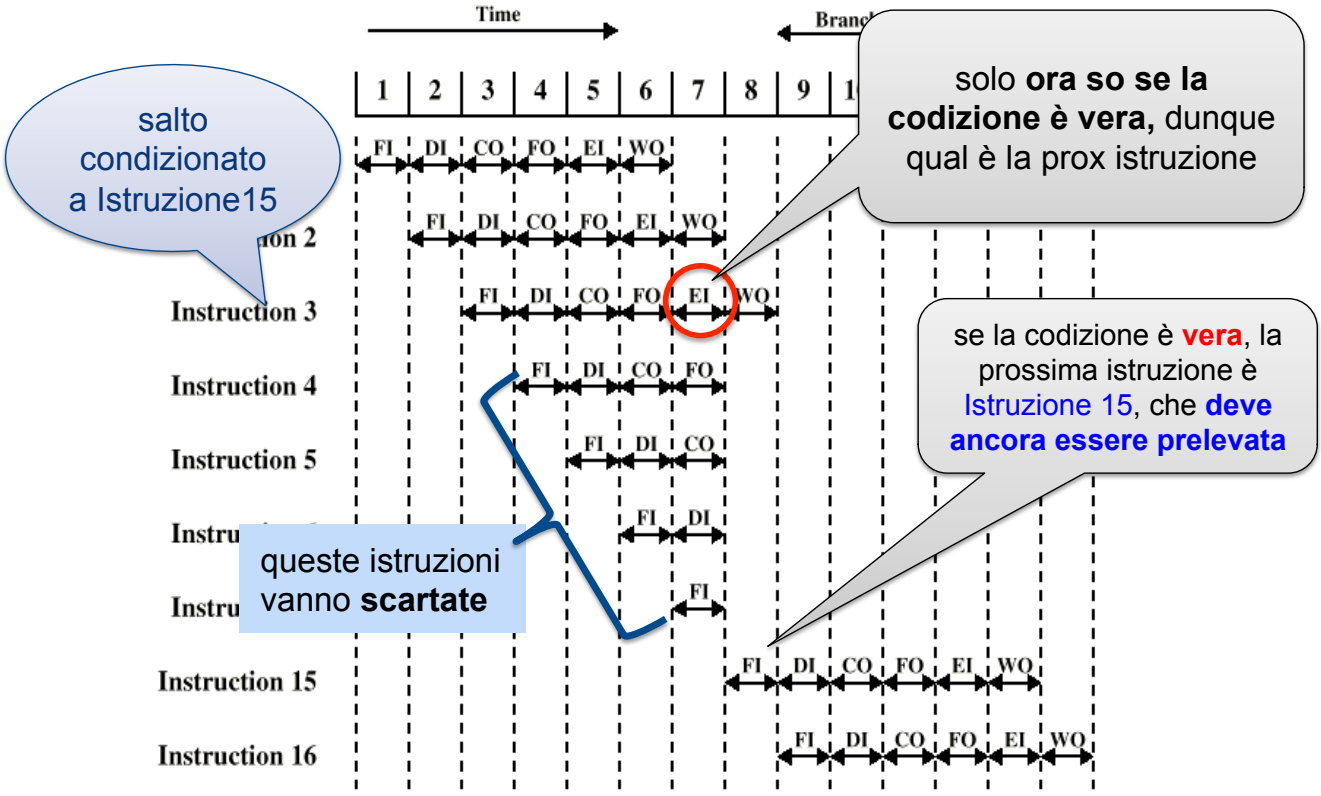
  non si raggiunge il parallelismo massimo

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

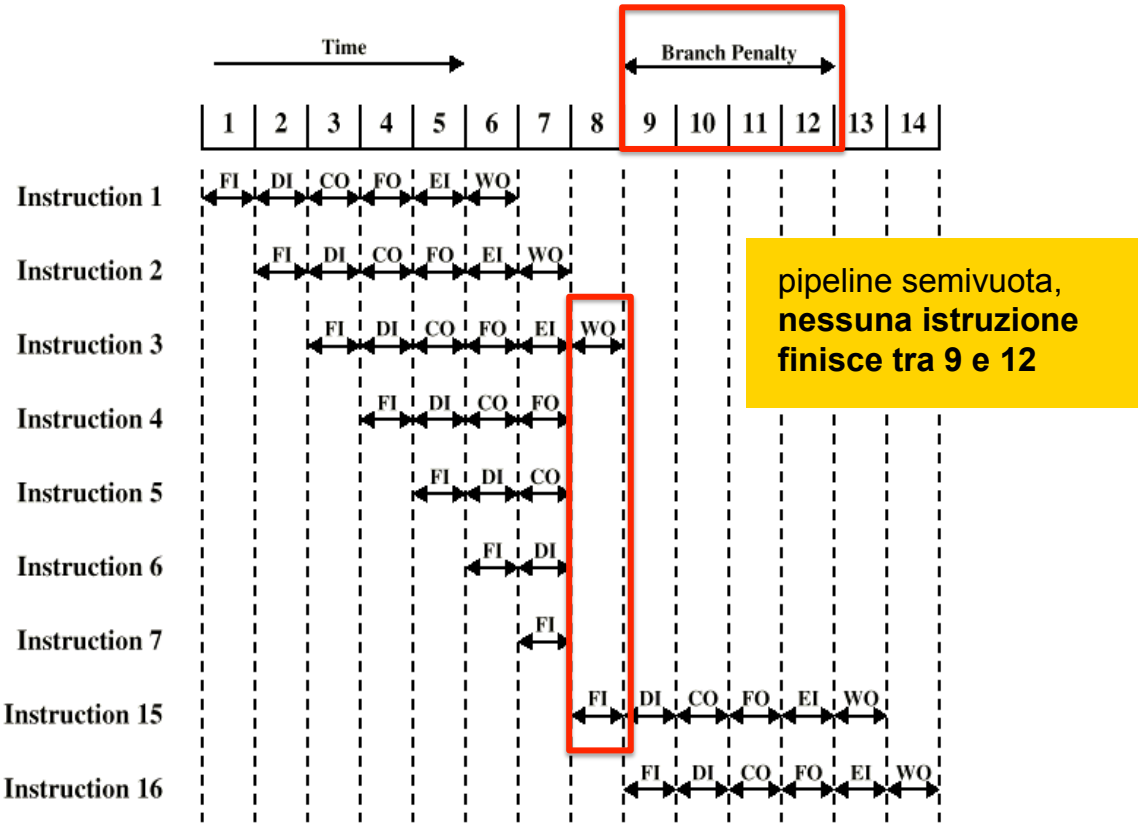
Salto condizionato



Salto condizionato



Salto condizionato



pipeline semivuota, **nessuna istruzione finisce tra 9 e 12**

Esempi

qui si sa se è vera la condizione di salto

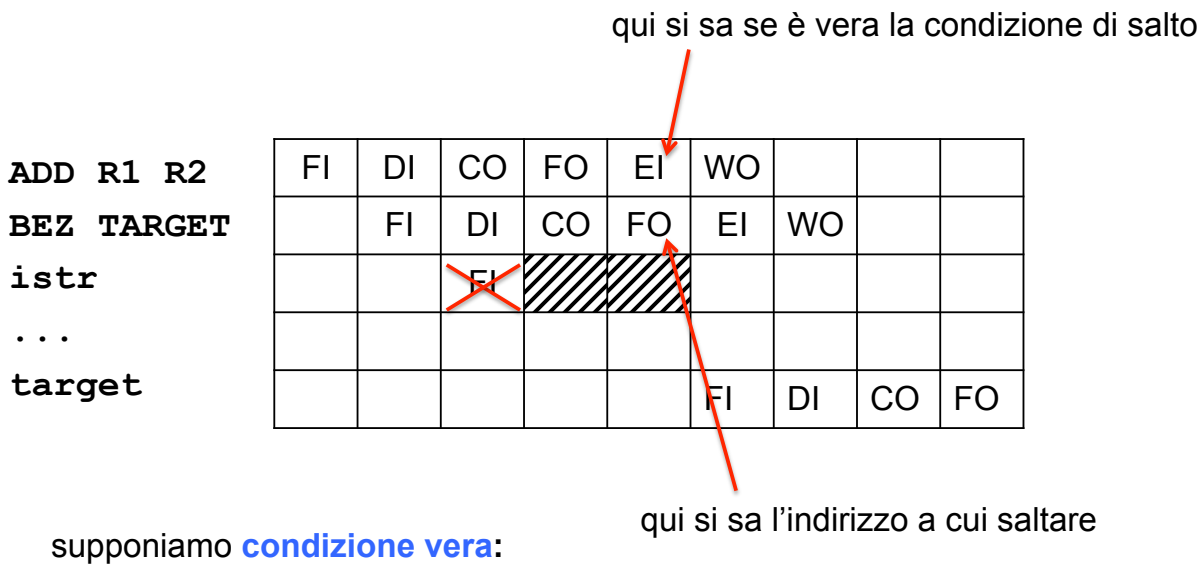
`ADD R1 R2`	FI	DI	CO	FO	EI	WO			
`BEZ TARGET`		FI	DI	CO	FO	EI	WO		
`istr`			FI						
`...`									
`target`									

qui sa che forse c'è un salto

qui si sa l'indirizzo a cui saltare

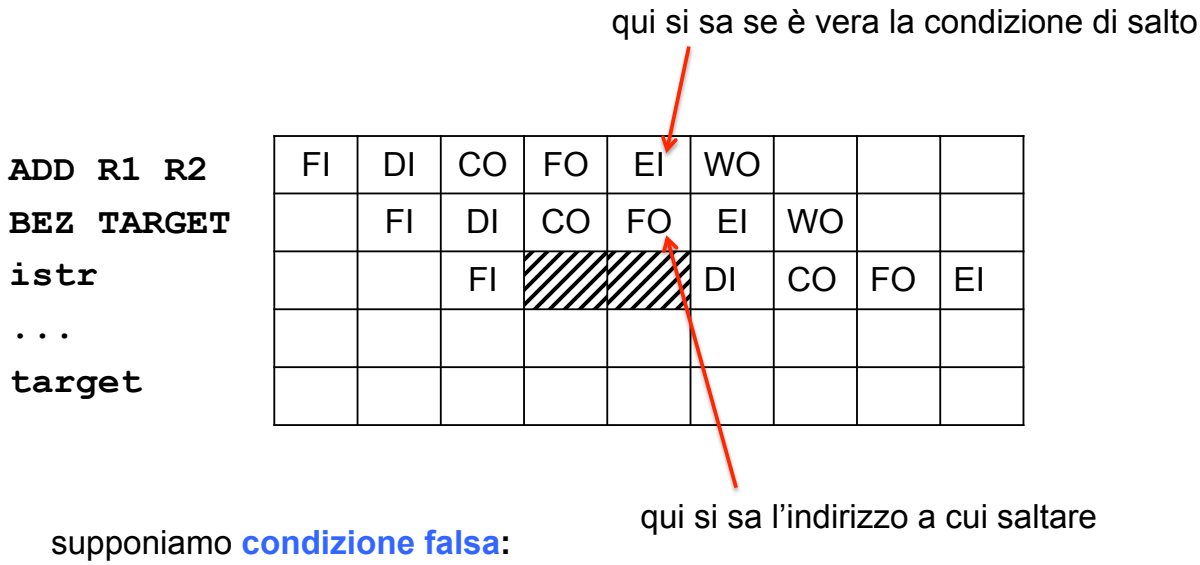
finche' non si sa se/dove saltare
si mette pipeline in stallo

Esempi



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

Esempi



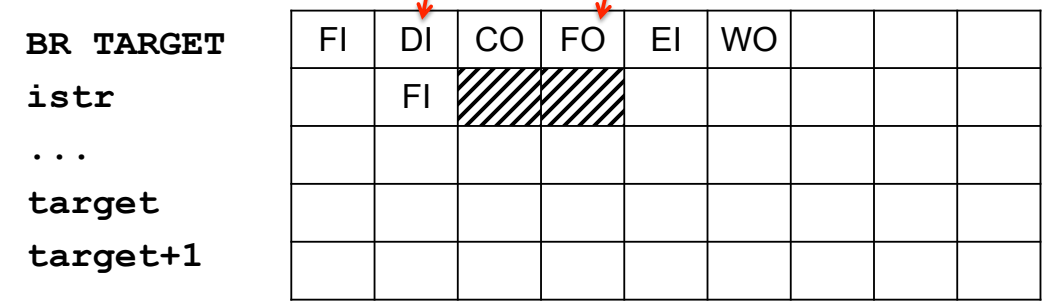
supponiamo **condizione falsa**:
riprendo dopo lo stallo con
l'istruzione pre-fetched

Esempi

qui si sa che ci sarà un salto

salto incondizionato

qui si sa l'indirizzo a cui saltare



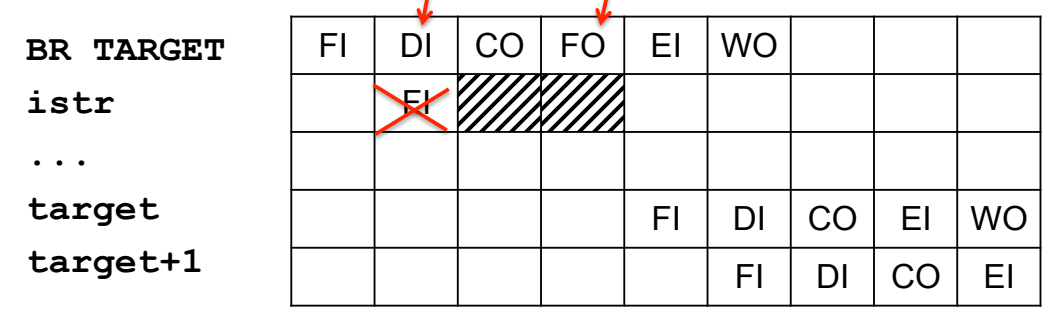
ormai ha prelevato l'istruzione errata
intanto mette in stallo

Esempi

qui si sa che ci sarà un salto

salto incondizionato

qui si sa l'indirizzo a cui saltare



ormai ha prelevato l'istruzione errata
intanto mette in stallo

scarta **`istr`** e 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, che va scartata
  - queste istruzioni sono circa il 30% del totale medio di un programma

Soluzioni:

- **mettere in stallo** la pipeline finche' 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

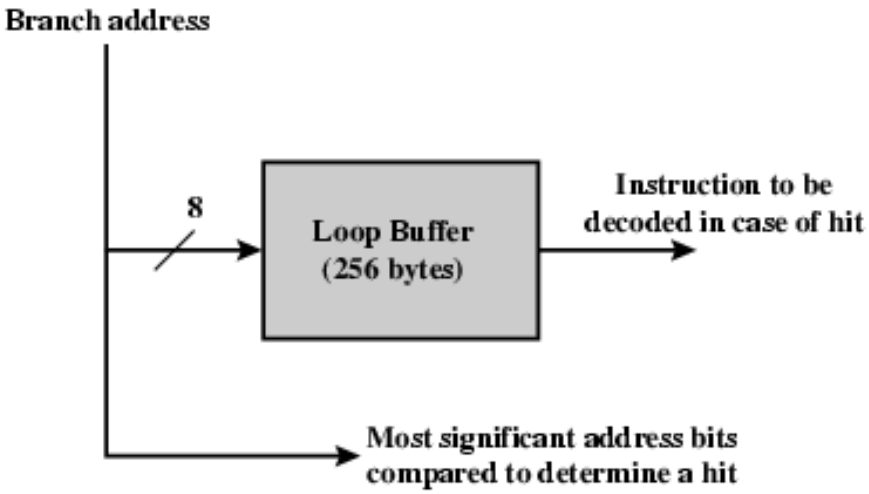
2. prefetch dell'istruzione target
   - anticipa il fetch dell'istruzione target oltre a quella successiva al salto
   - se il salto è preso, trova l'istruzione già caricata
   - in ogni caso una parte della pipeline deve essere scartata

Soluzioni per salti condizionati

1. flussi multipli (*multiple streams*)
2. prefetch dell'istruzione target
3. buffer circolare *(loop buffer)*
   - è una memoria piccola e 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 po' 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)

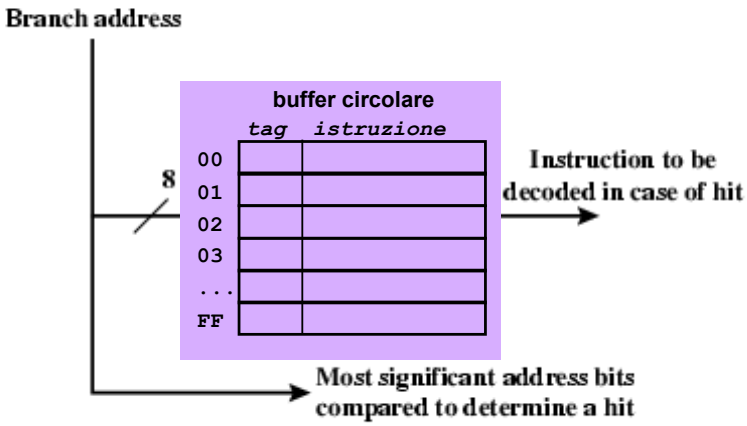
- buffer senza prefetch
- gli **8 bit meno significativi** sono usati come **indice nel buffer**
- gli altri **bit più significativi** si usano per controllare **se** la destinazione del salto sta **già nel buffer**



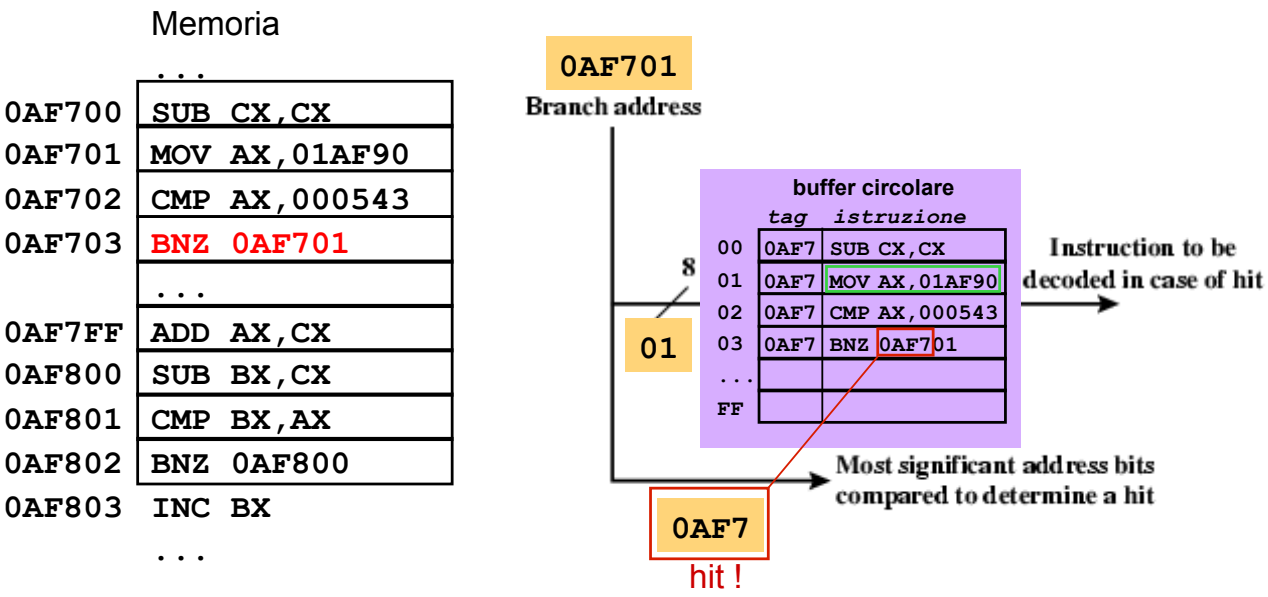
Buffer circolare (senza prefetch)

Memoria

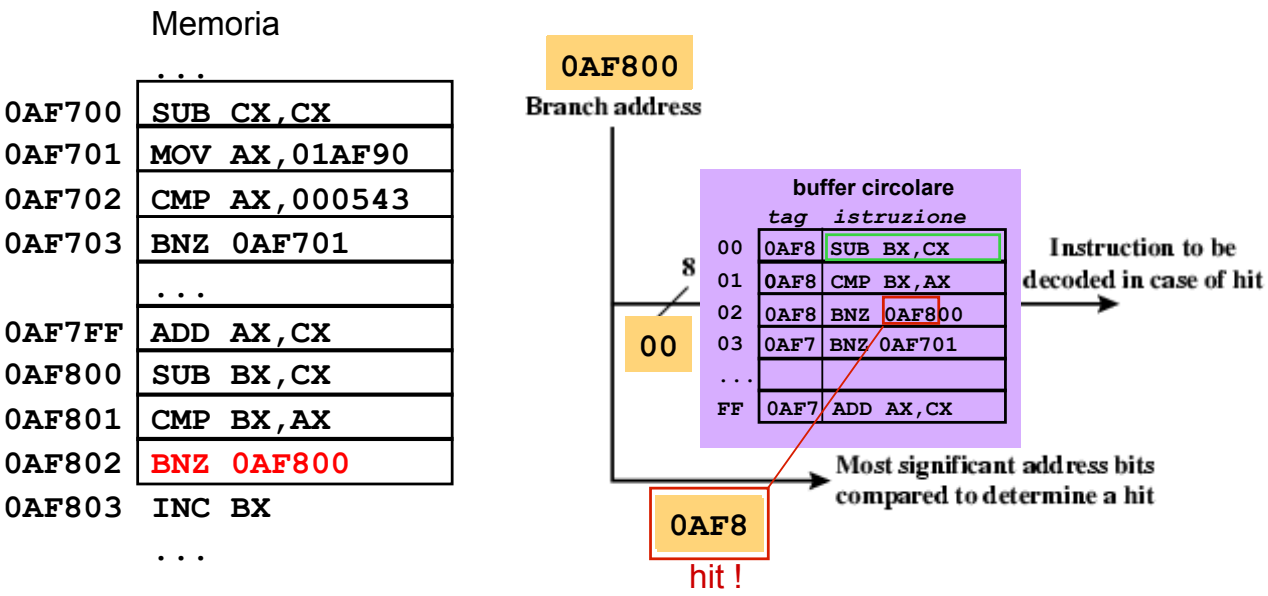
```
          ...
0AF700    SUB  CX,CX
0AF701    MOV  AX,01AF90
0AF702    CMP  AX,000543
0AF703    BNZ  0AF701
          ...
0AF7FF    ADD  AX,CX
0AF800    SUB  BX,CX
0AF801    CMP  BX,AX
0AF802    BNZ  0AF800
0AF803    INC  BX
          ...
```



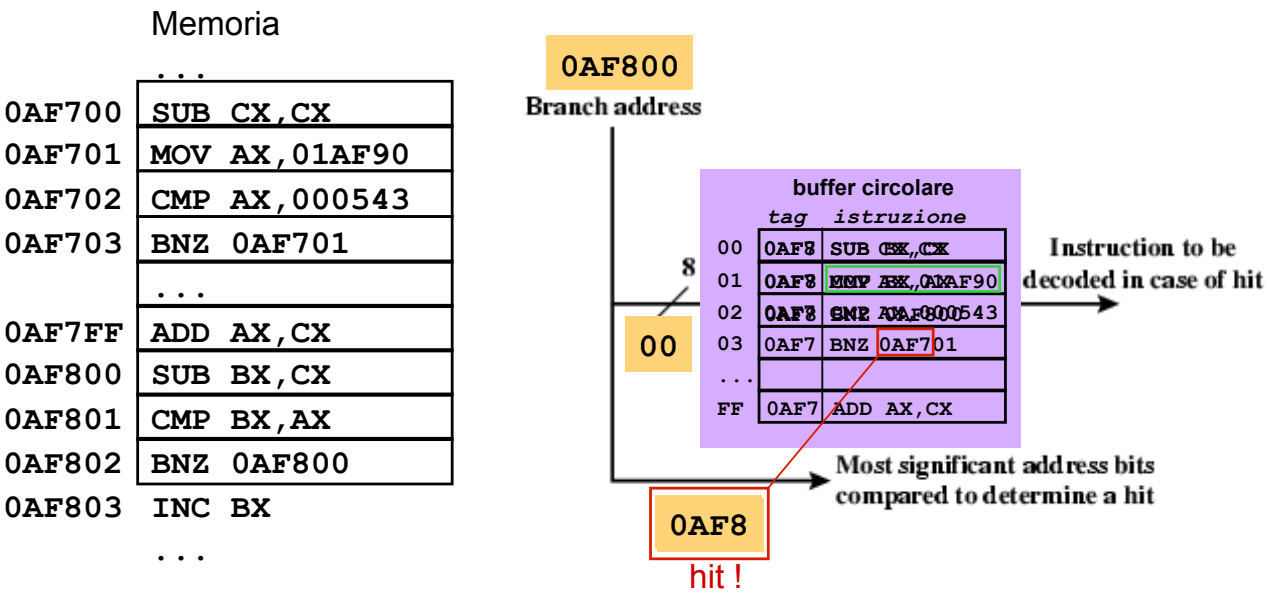
Buffer circolare (senza prefetch)



Buffer circolare (senza prefetch)



Buffer circolare (senza prefetch)



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**
   - previsone 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

esempio:

```
        ……
LOOP:   ……
        ……
        ……
        BNZ   LOOP
```

associo 1 bit all'istruzione di salto
- se bit è **1** predico di saltare
- se bit è **0** predico di non saltare
- se ho sbagliato predizione inverto il 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
```

associo 1 bit all'istruzione di salto
- se bit è 1 predico di saltare
- se bit è 0 predico di non saltare
- se ho sbagliato predizione inverto il bit

- bit di predizione a **1**
- entro in LOOP
- n iterazioni - ***n predizioni corrette***
- esco dal ciclo - ***predizione errata*** – metto bit a **0**
- al prossimo ciclo trovo bit a 0
- alla prima iterazione – ***predizione errata*** – metto bit a **1**
- m-1 iterazioni - ***predizioni corrette***
- esco dal ciclo – ***predizione errata*** – metto bit a **0**

a regime:
2 errori per ciclo

Predizione dinamica con 2 bit

- un solo errore per ciclo
- 2 bit per ricordare come è andata la predizione degli ultimi due salti
- per invertire la predizione ci vogliono 2 errori