Dipendenza dai dati - Soluzioni

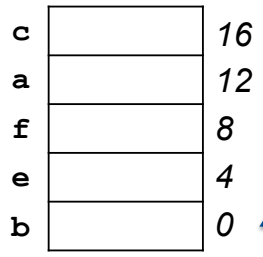
1. Introduzione di fasi non operative (**nop**-stallo)
2. propagazione in avanti del dato richiesto (**data forwarding**)
   – dipende da architettura di pipeline e implementazione istruzioni
3. **riordino delle istruzioni**

riordino delle istruzioni

programma C con 5 variabili
che si riferiscono a indirizzi di 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
```

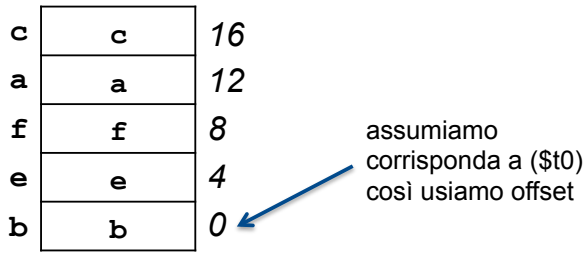
```
lw   $1   0   ($t0)
lw   $2   4   ($t0)
add  $3   $1   $2
sw   $3   12  ($t0)
lw   $4   8   ($t0)
add  $5   $1   $4
sw   $5   16  ($t0)
```

riordino delle istruzioni

programma C con 5 variabili
che si riferiscono a indirizzi di memoria

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

memoria indirizzata al byte (1 word=4 byte)




```
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**

(a seconda di come è fatta pipeline qualche problema può risoversi con *data forwarding*)

riordino delle istruzioni

programma C con 5 variabili
che si riferiscono a indirizzi di memoria

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

riordinando le istruzioni si sono "ridotte"
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)
```

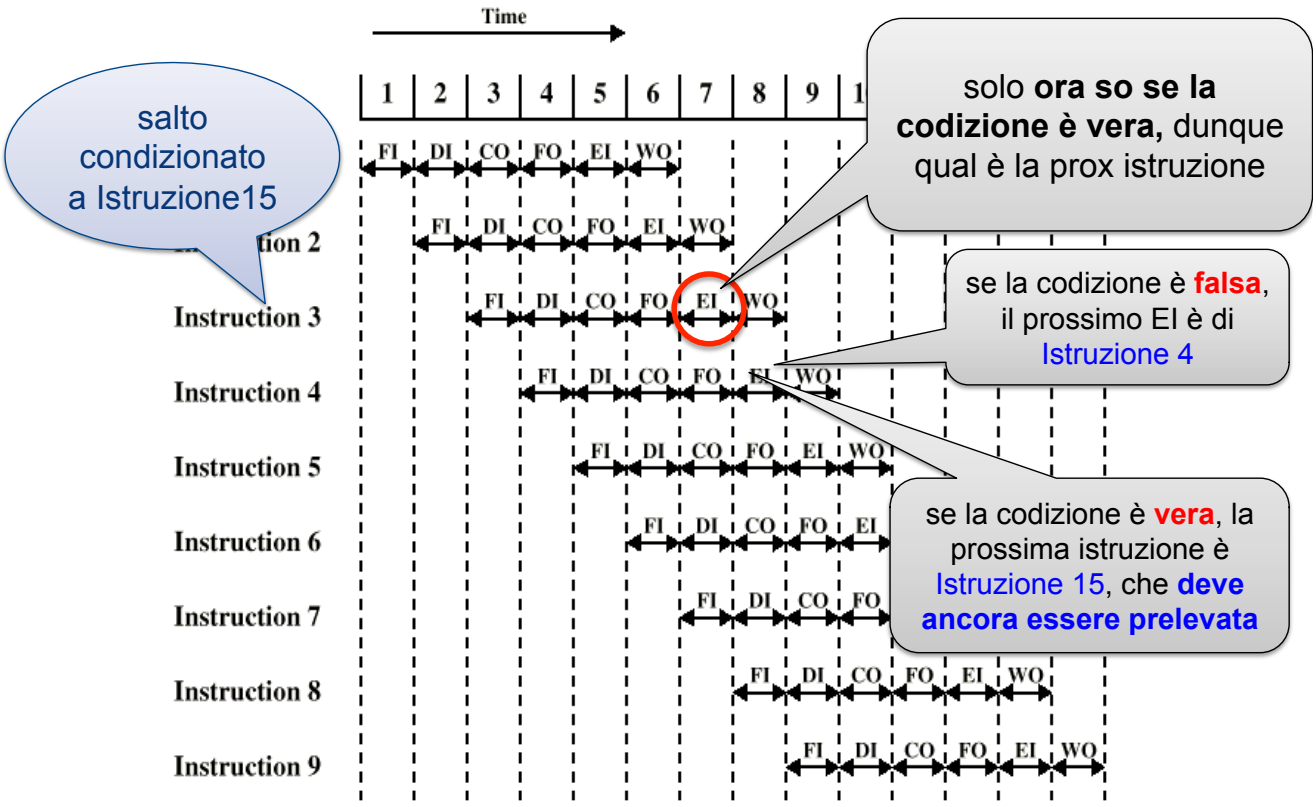
pipeline hazards - criticità

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

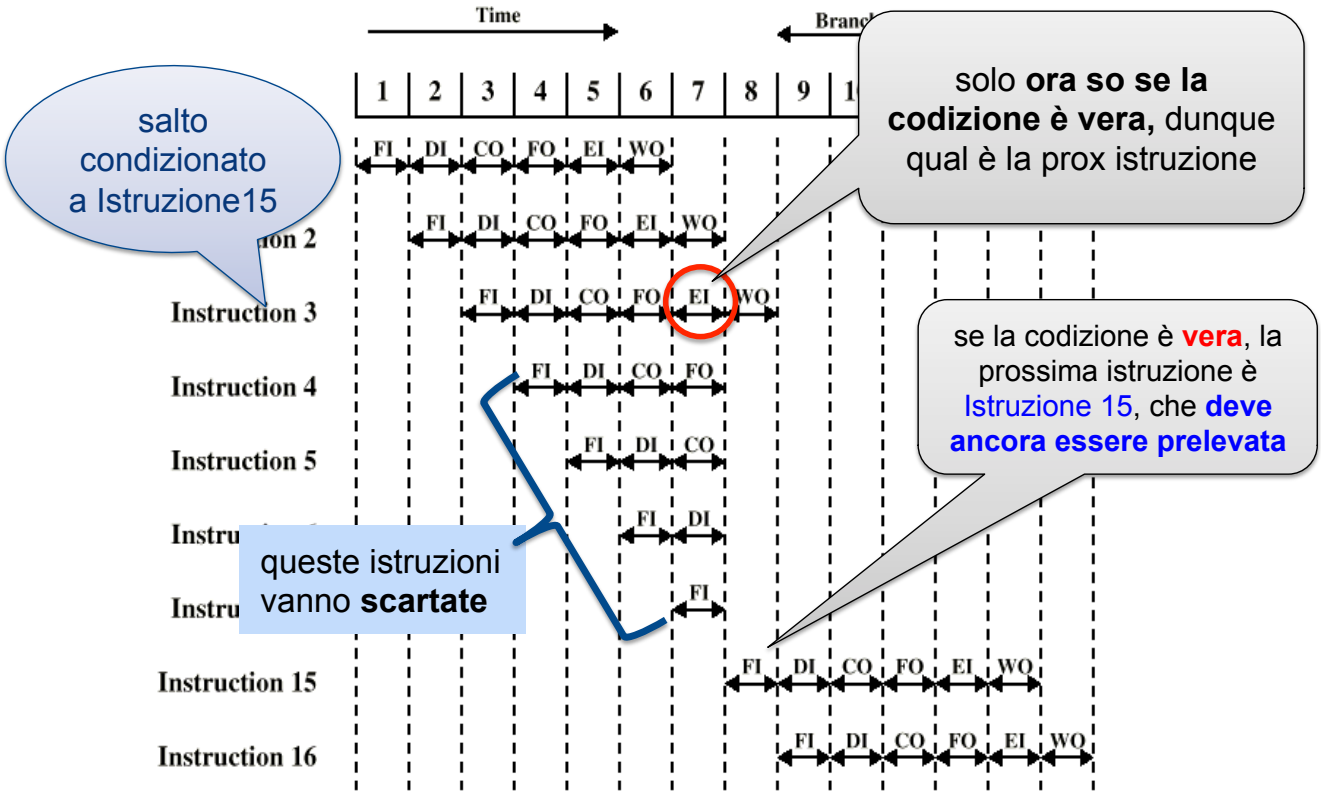
  non si raggiunge il parallelismo massimo

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 o no), chiamate e ritorni da procedure, interruzioni.

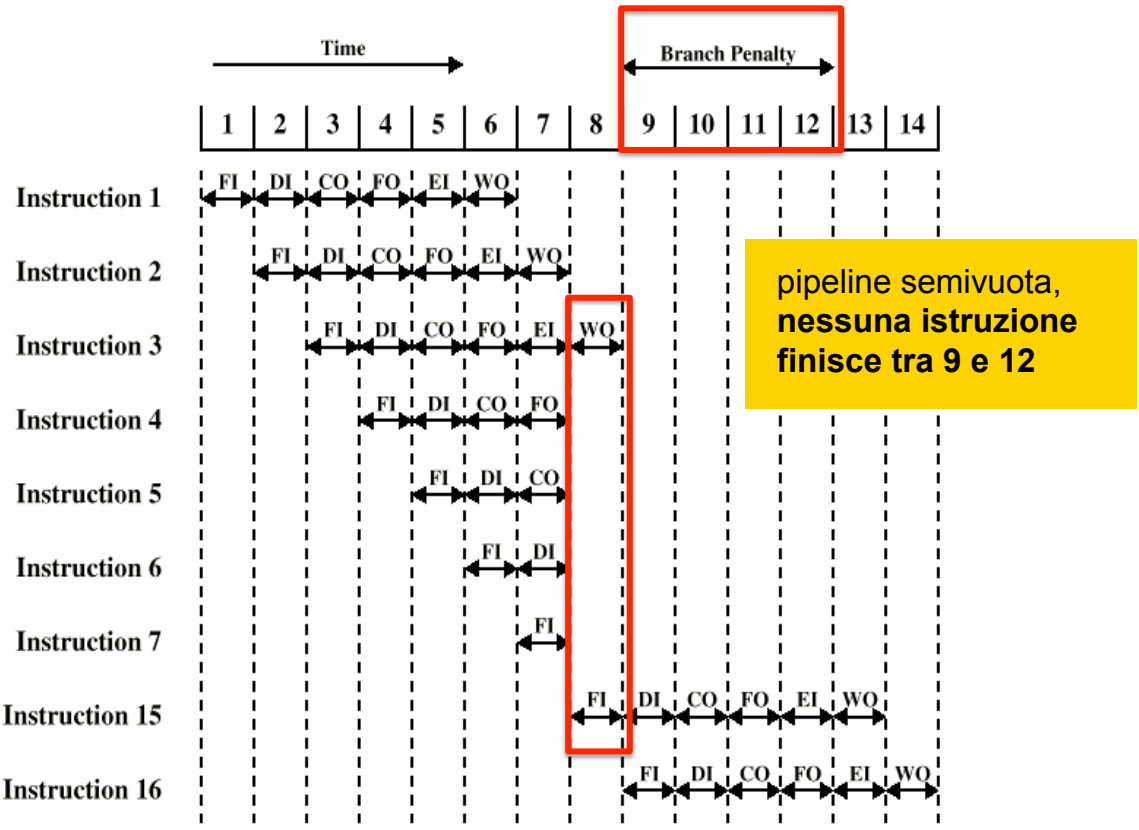
Salto condizionato



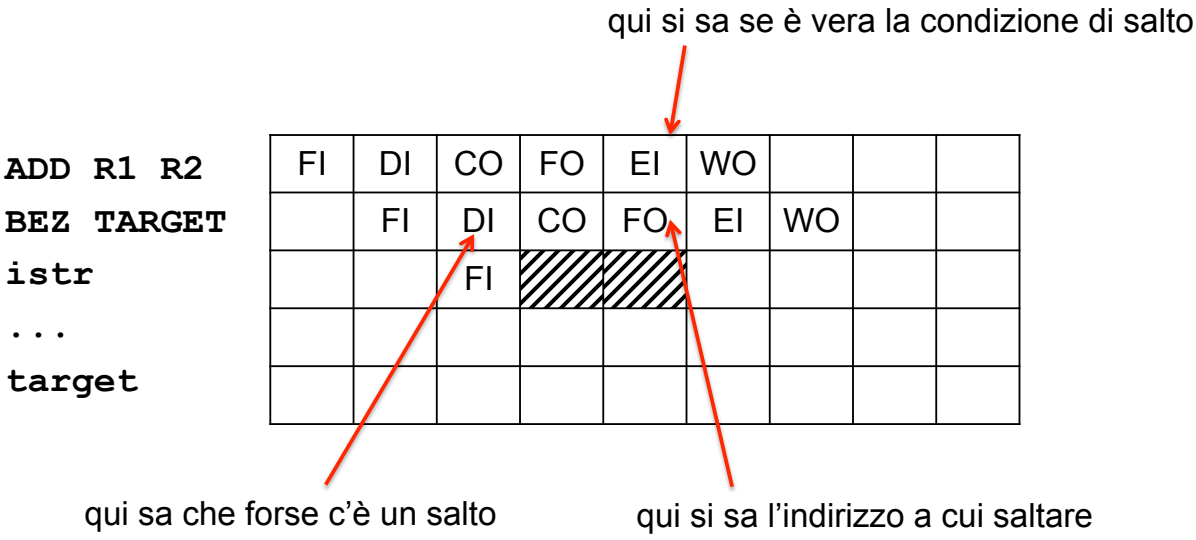
Salto condizionato



Salto condizionato

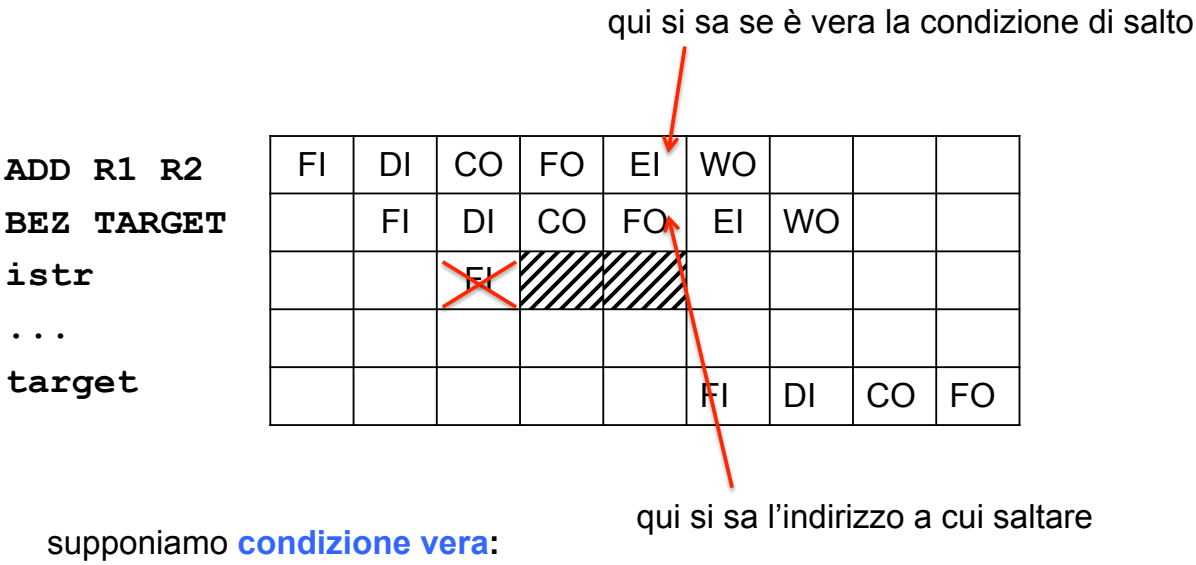


Esempi



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

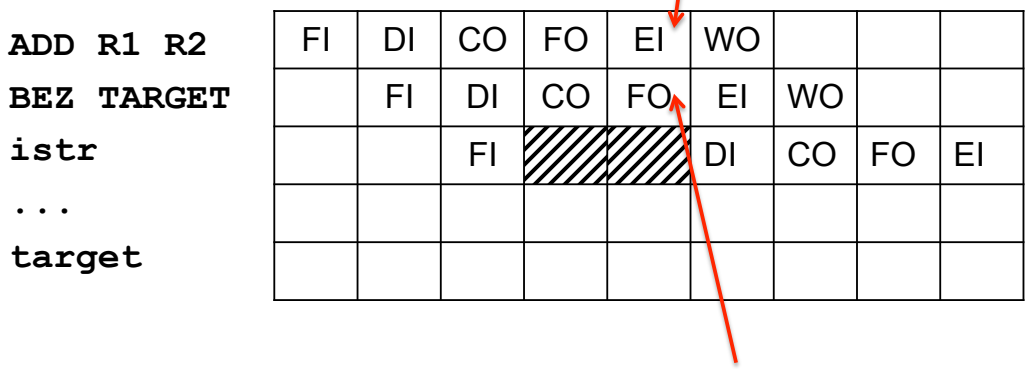
Esempi



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

Esempi

qui si sa se è vera la condizione di salto



qui si sa l'indirizzo a cui saltare

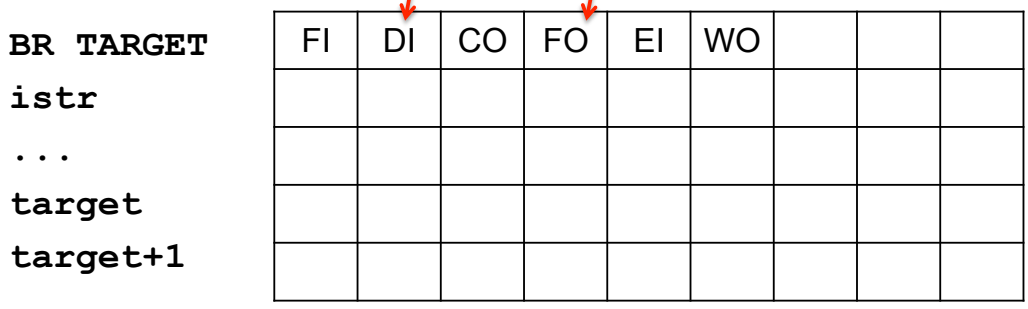
supponiamo **condizione falsa**:
riprendo dopo lo stallo con
l'istruzione pre-fetched

Esempi

salto incondizionato

qui si sa che ci sarà un salto

qui si sa l'indirizzo a cui saltare

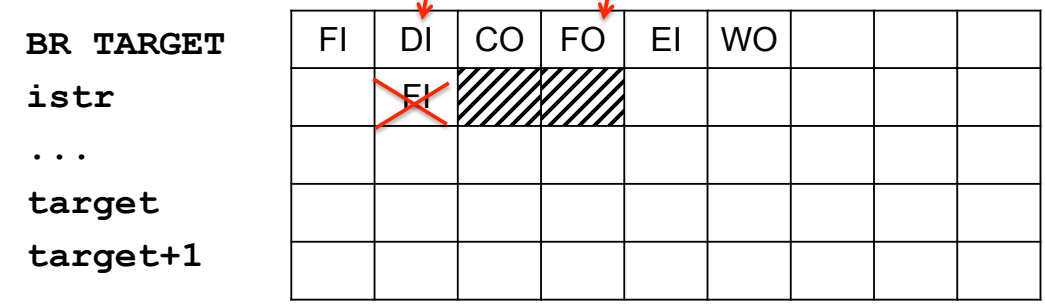


Esempi

qui si sa che ci sarà un salto

salto incondizionato

qui si sa l'indirizzo a cui saltare



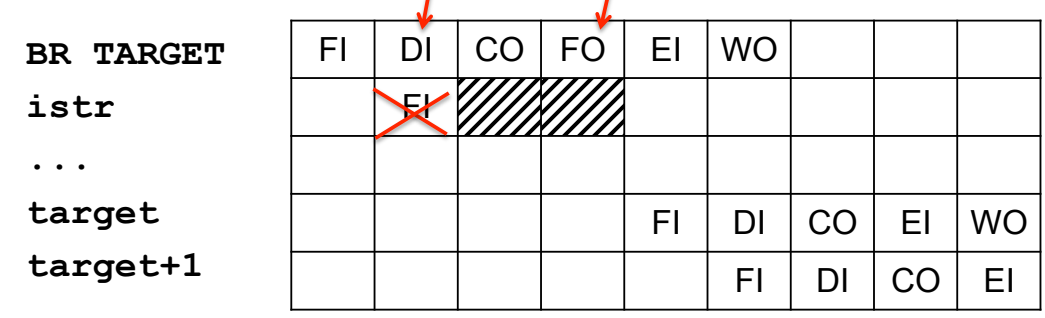
ormai ha prelevato l'istruzione errata,
quindi mette in stallo e scarta **`istr`**

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
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, 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

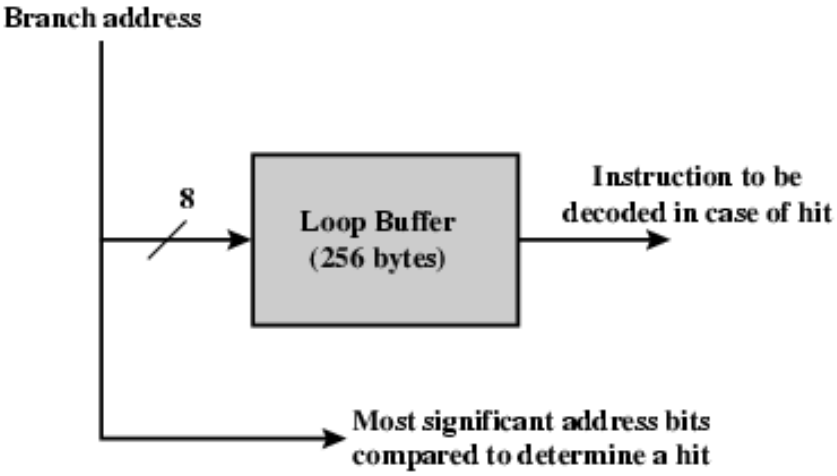
2. prefetch dell'istruzione target
   - anticipa il fetch dell'istruzione target oltre a quella successiva al salto
   - se il salto è preso, trova l'istruzione già caricata
   - in ogni caso una parte della pipeline deve essere scartata

Soluzioni per salti condizionati

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

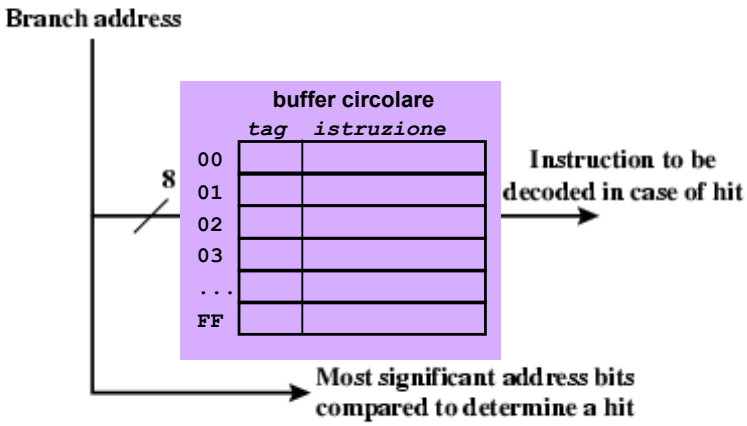
- buffer senza prefetch, capienza 256 bytes, indirizzato a byte
- dato l'*indirizzo* target di salto/branch, controllo se c'è nel buffer:
  - gli **8 bit meno significativi** sono usati come **indice nel buffer**
  - gli altri **bit più significativi** si usano per controllare **se** la destinazione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



Buffer circolare (senza prefetch)

Memoria

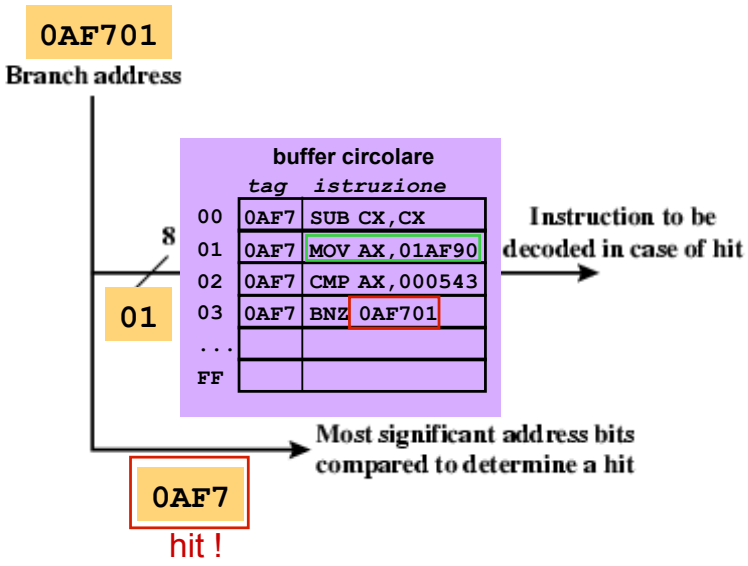
```
        ...
0AF700  SUB CX,CX
0AF701  MOV AX,01AF90
0AF702  CMP AX,000543
0AF703  BNZ 0AF701
        ...
0AF7FF  ADD AX,CX
0AF800  SUB BX,CX
0AF801  CMP BX,AX
0AF802  BNZ 0AF800
0AF803  INC BX
        ...
```



Buffer circolare (senza prefetch)

Memoria

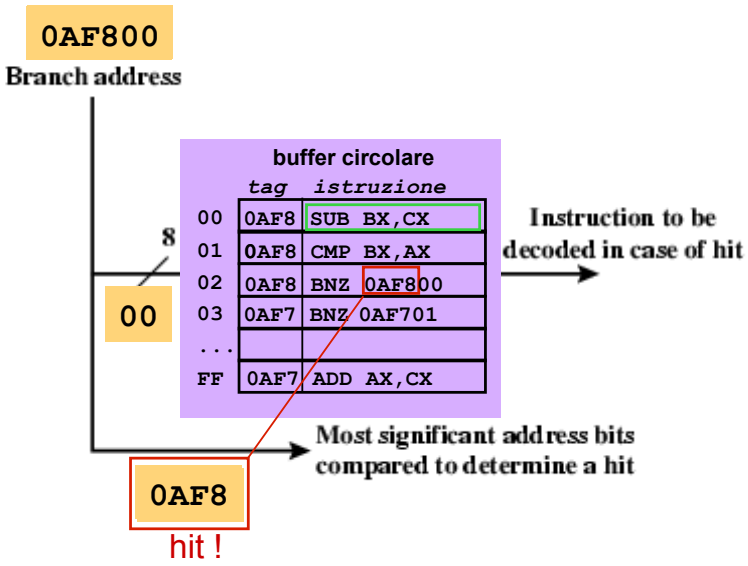
```
          ...
0AF700    SUB  CX,CX
0AF701    MOV  AX,01AF90
0AF702    CMP  AX,000543
0AF703    BNZ  0AF701
          ...
0AF7FF    ADD  AX,CX
0AF800    SUB  BX,CX
0AF801    CMP  BX,AX
0AF802    BNZ  0AF800
0AF803    INC  BX
          ...
```



Buffer circolare (senza prefetch)

Memoria

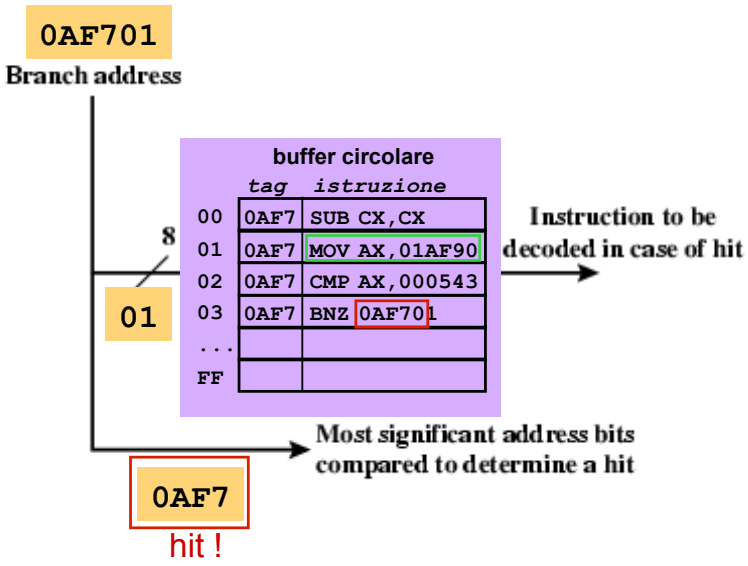
```
          ...
0AF700    SUB  CX,CX
0AF701    MOV  AX,01AF90
0AF702    CMP  AX,000543
0AF703    BNZ  0AF701
          ...
0AF7FF    ADD  AX,CX
0AF800    SUB  BX,CX
0AF801    CMP  BX,AX
0AF802    BNZ  0AF800
0AF803    INC  BX
          ...
```



Buffer circolare (senza prefetch)

Memoria

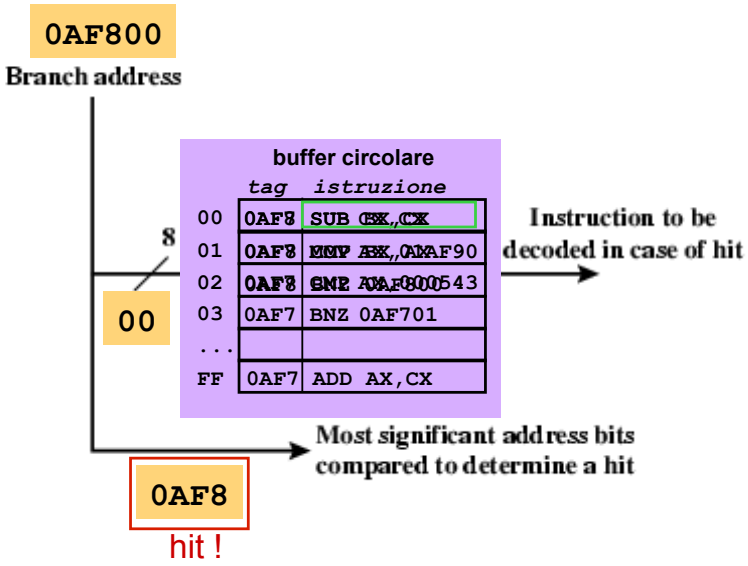
```
          ...
0AF700    SUB  CX,CX
0AF701    MOV  AX,01AF90
0AF702    CMP  AX,000543
0AF703    BNZ  0AF701
          ...
0AF7FF    ADD  AX,CX
0AF800    SUB  BX,CX
0AF801    CMP  BX,AX
0AF802    BNZ  0AF800
0AF803    INC  BX
          ...
```



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**
   - 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

- 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

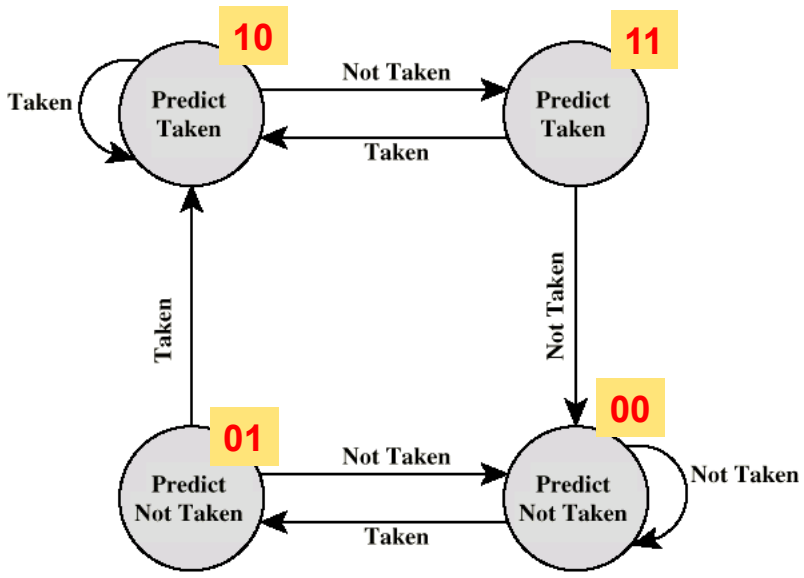
esempio:

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

- dopo la prima esecuzione del ciclo, **in uscita dal ciclo**, il bit assegnato a BNZ LOOP è **0** perche' 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



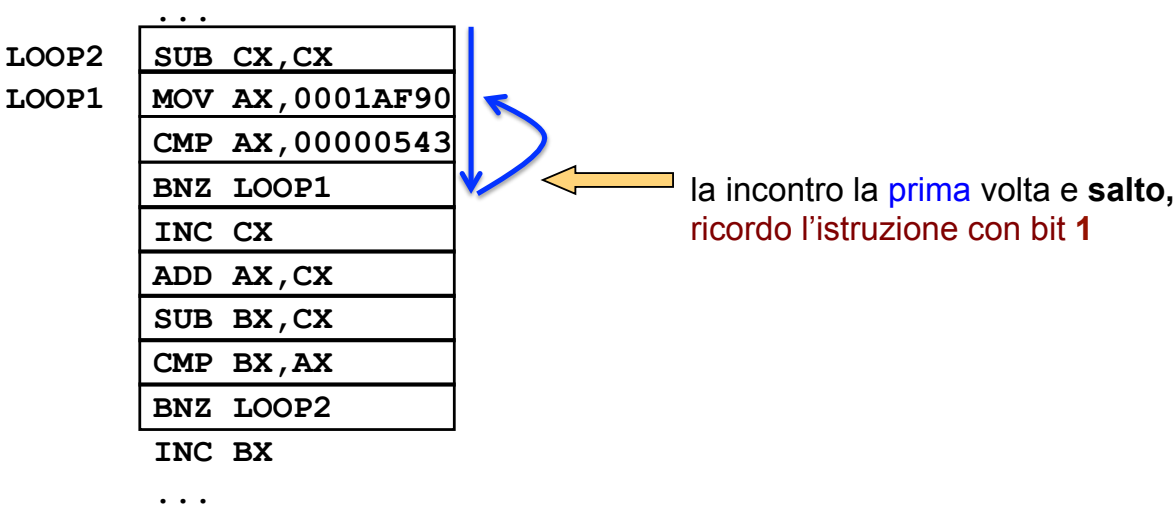
Predizione dinamica 1/2 bit

- 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

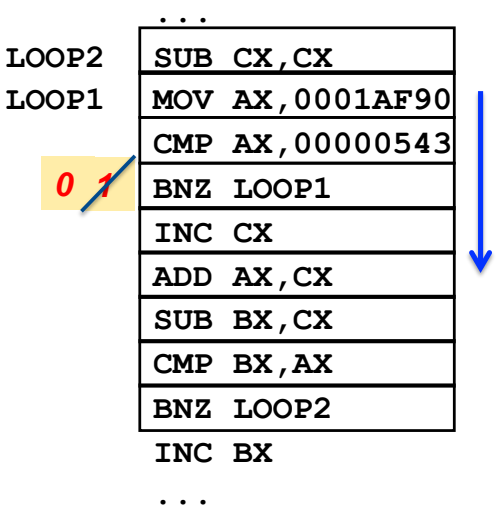
Predizione dinamica 1 bit

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 <u>una sola volta</u>

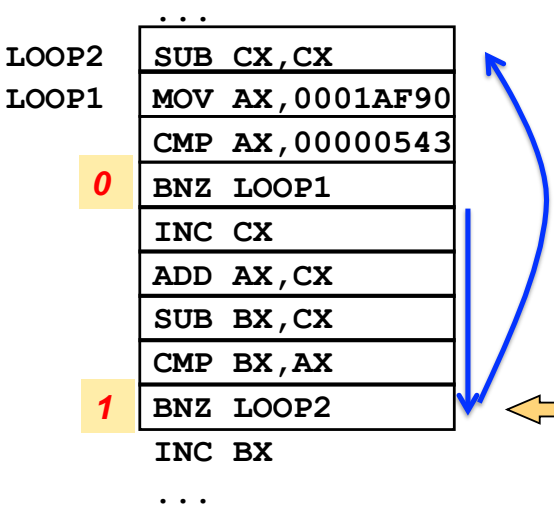


la incontro la seconda volta
- e **non salto**
- la ricordavo con bit **1**
- **quindi errore e cambio bit a 0**

errori totali = **1**

Predizione dinamica 1 bit

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

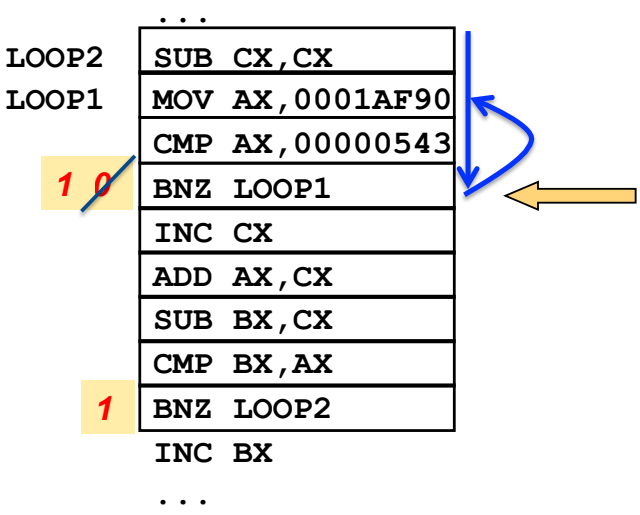


la incontro la *prima* volta e **salto**
ricordo questa istruzione con bit **1**

errori totali = **1**

Predizione dinamica 1 bit

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

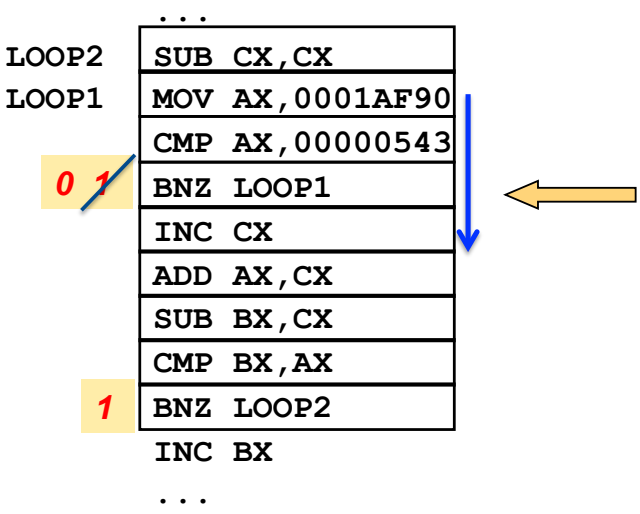


la incontro la terza volta
- e **salto**
- la ricordavo con bit **0**
- **quindi errore e cambio bit a 1**

errori totali = **2**

Predizione dinamica 1 bit

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

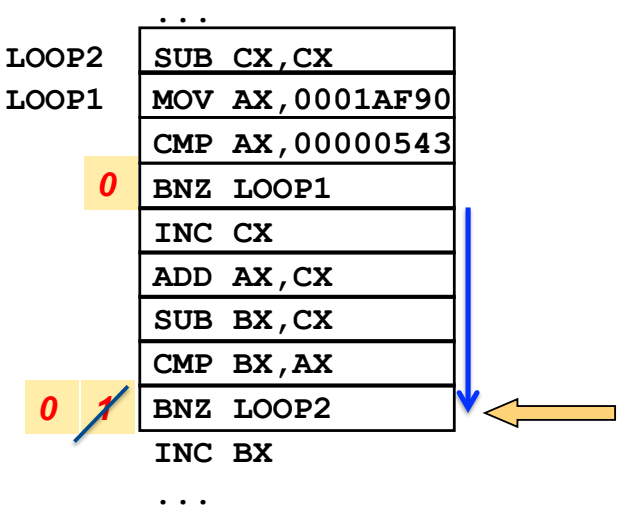


la incontro la quarta volta
- e **non salto**
- la ricordavo con bit 1
- **quindi errore e cambio bit 0**

errori totali = **3**

Predizione dinamica 1 bit

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

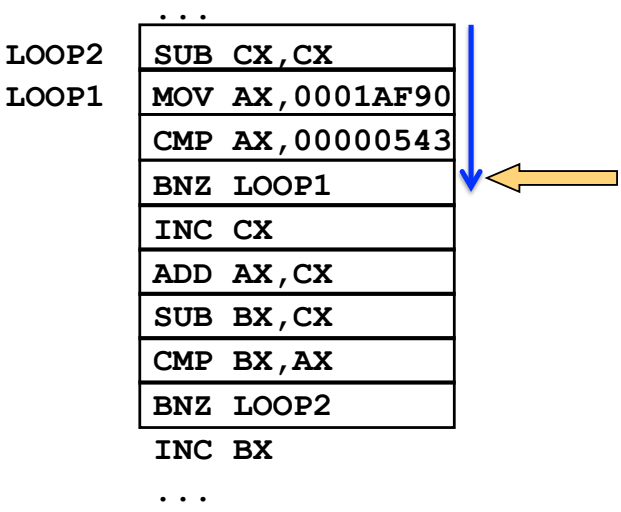


la incontro la seconda volta
- e **non salto**
- la ricordavo con bit 1
- **quindi errore e cambio bit a 0**

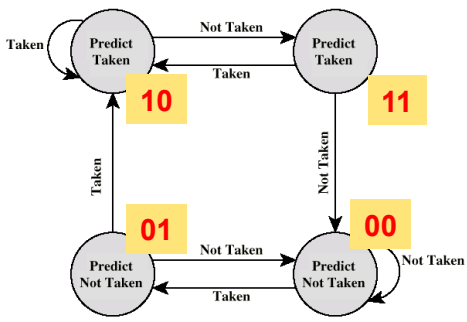
errori totali = **4**

Predizione dinamica 2 bit

errori totali = 0



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

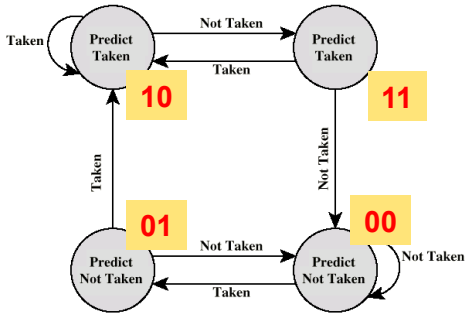


Predizione dinamica 2 bit

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
        ...
```

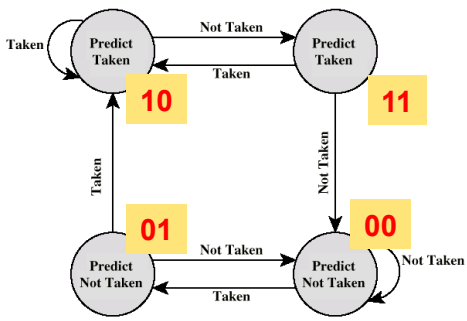
la incontro la seconda volta
- predice salto e **non salto**
- la ricordavo con bit **10**
- **quindi errore e cambio in 11** (*ma resta stessa predizione*)



Predizione dinamica 2 bit

errori totali = **1**

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



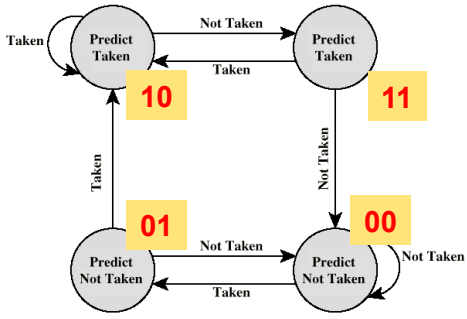
Predizione dinamica 2 bit

errori totali = 1 non è errore in più

```
        ...
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
        ...
```

~~10~~ ~~11~~

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



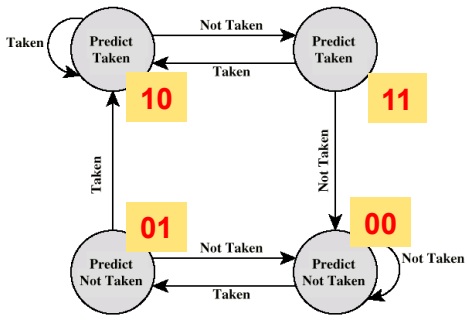
Predizione dinamica 2 bit

errori totali = **2**

```
        ...
LOOP2   SUB  CX,CX
LOOP1   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 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 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 predizone 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)