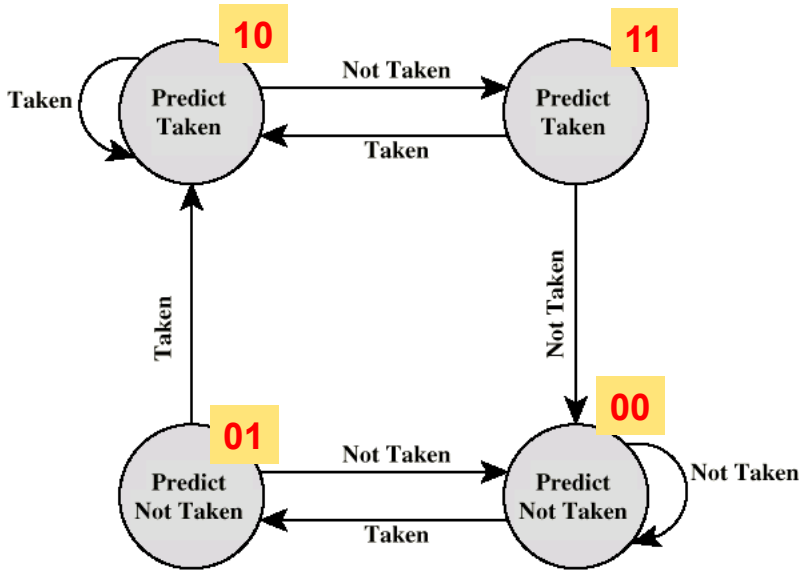
Predizione dinamica con 2 bit

- 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



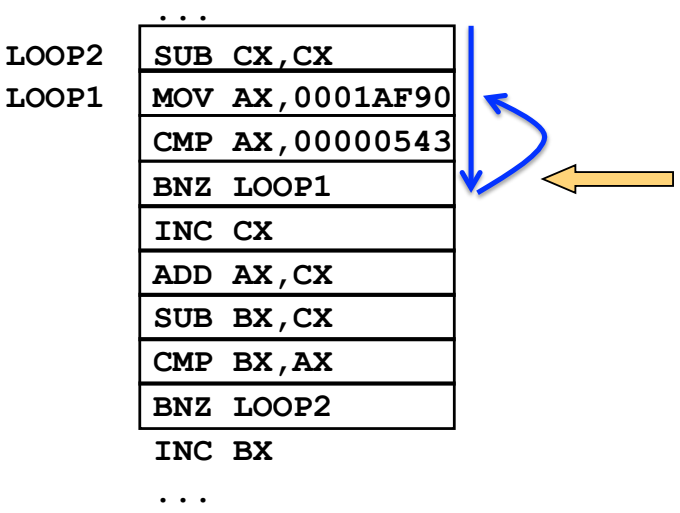
Predizione dinamica 1/2 bit

- 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** 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

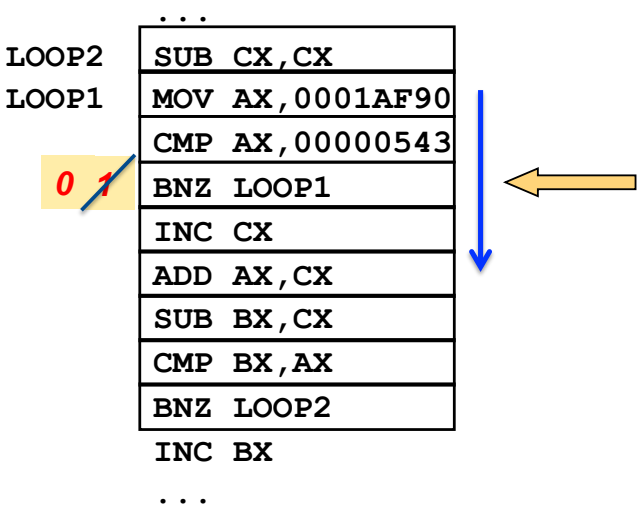
errori totali = 0



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

Predizione dinamica 1 bit

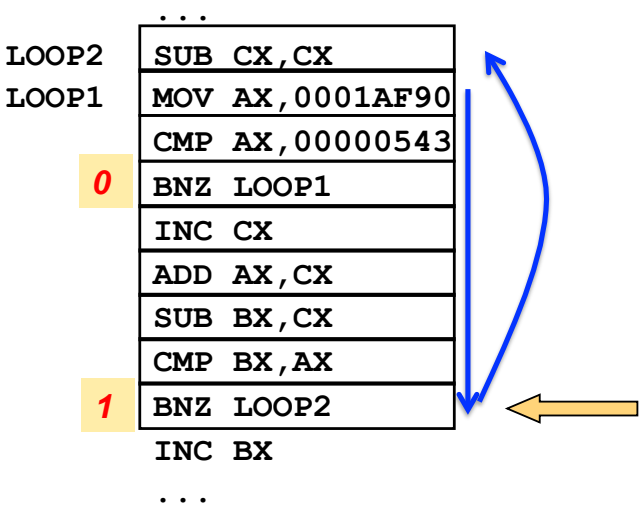
errori totali = **1**



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

Predizione dinamica 1 bit

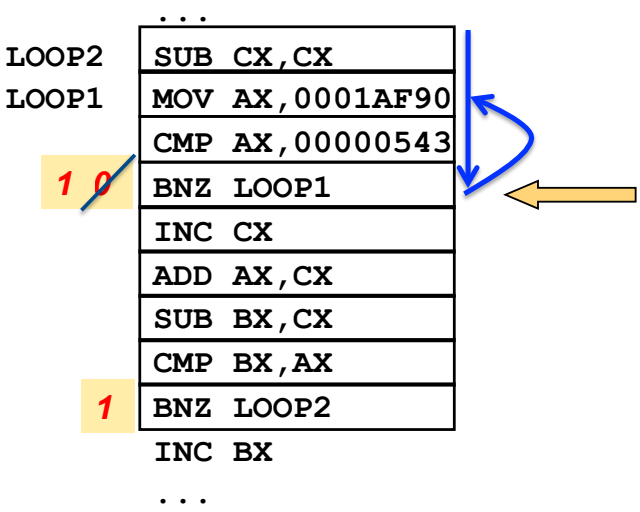
errori totali = **1**



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

Predizione dinamica 1 bit

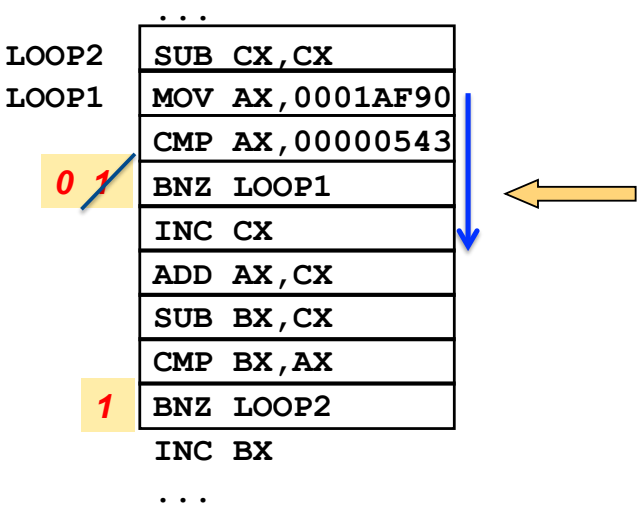
errori totali = **2**



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

Predizione dinamica 1 bit

errori totali = **3**

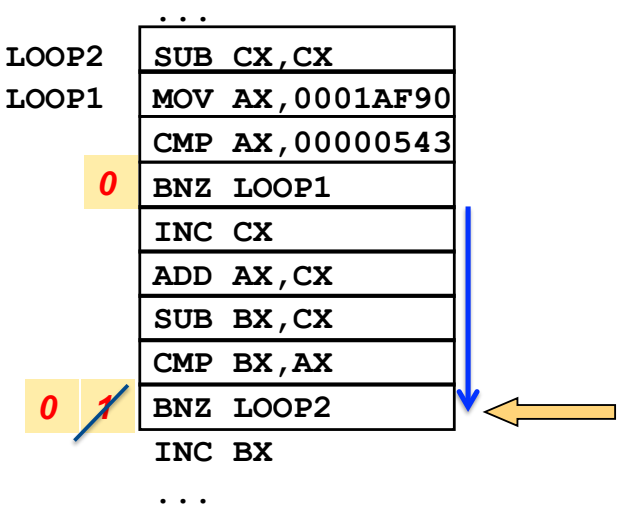


la incontro la quarta volta
- e **non salto**
- la ricordavo con bit 1
- **quindi errore e cambio bit**

Predizione dinamica 1 bit

errori totali = 4

2 errori per ciclo



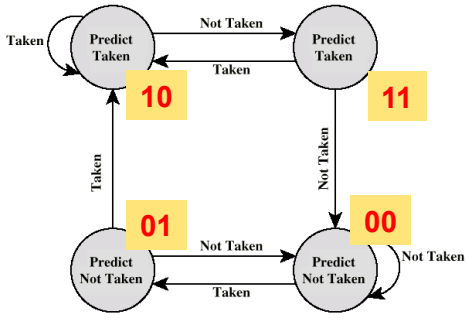
la incontro la seconda volta
- e **non salto**
- la ricordavo con bit 1
- **quindi errore e cambio bit**

Predizione dinamica 2 bit

errori totali = 0

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

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

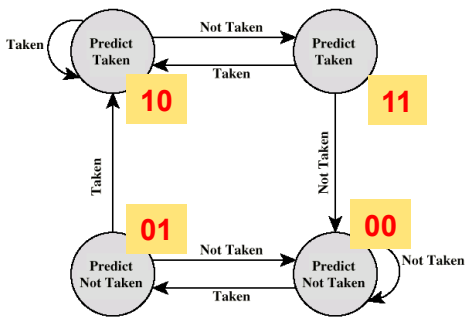


Predizione dinamica 2 bit

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
- e **non salto**
- la ricordavo con bit **10**
- **quindi errore e cambio in 11**

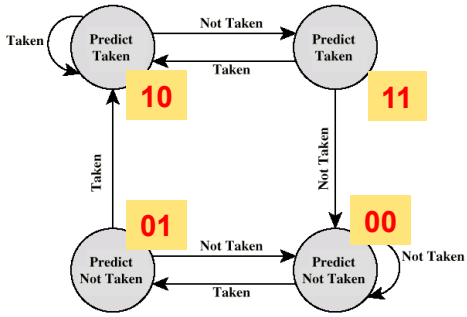


Predizione dinamica 2 bit

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
10      BNZ  LOOP2
        INC  BX
        ...
```

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

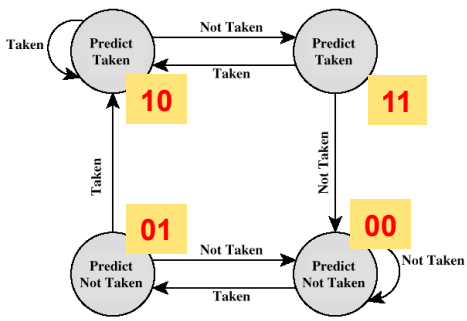


Predizione dinamica 2 bit

errori totali = 1 non è errore in più

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

la incontro la terza volta
- e **salto**
- la ricordavo con bit **11**
- **quindi non errore**
- **ma cambio bit**

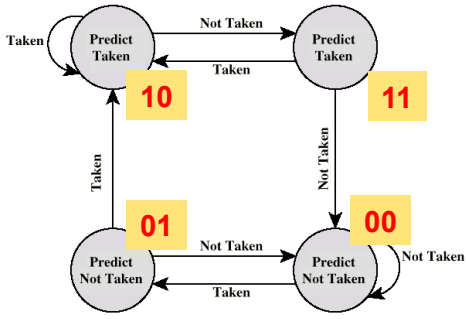


Predizione dinamica 2 bit

errori totali = **2**

```
        ...
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
10      BNZ  LOOP2
        INC  BX
        ...
```

la incontro la quarta volta
- e **non salto**
- la ricordavo con bit 10
- **quindi errore**
- **e cambio bit**

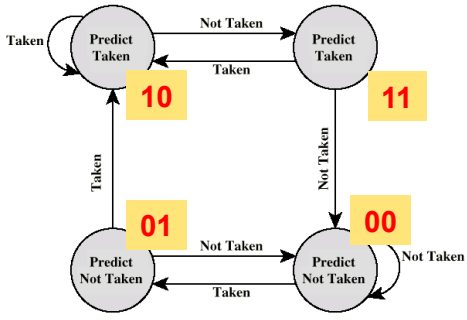


Predizione dinamica 2 bit

errori totali = 3

```
        ...
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
11 10   BNZ  LOOP2
        INC  BX
        ...
```

la incontro la seconda volta
- e **non salto**
- la ricordavo con bit 10
- **quindi errore e cambio bit**

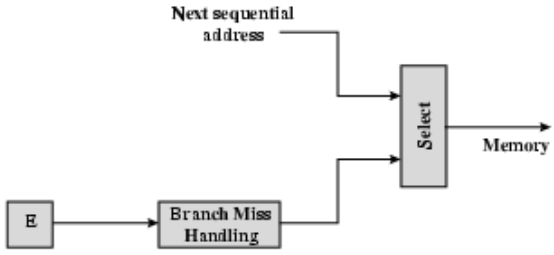


Predizione dinamica 1/2 bit

buffer di predizione dei salti

(*branch prediction buffer op branch history table*)

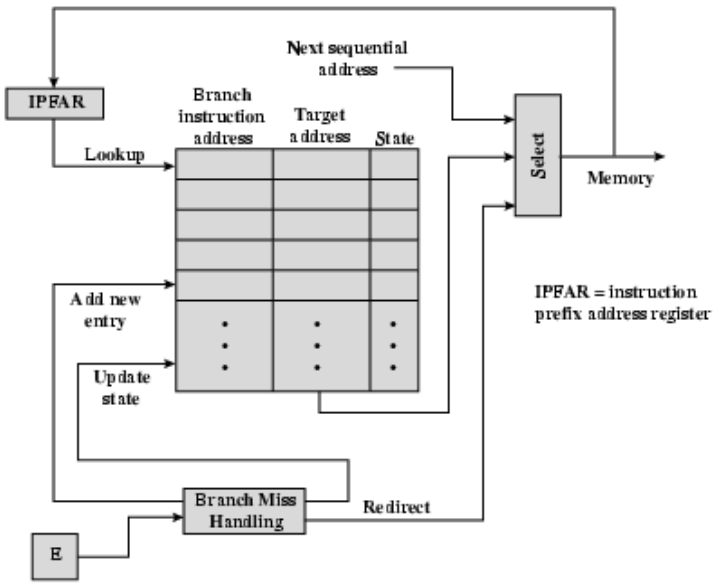
- questa tabella contiene i bit di predizione (1 opp 2)
- piccola memoria indicizzata attraverso i bit meno significativi dell'*indirizzo* dell'istruzione di salto
- si usa **nella fase FI**, che conosce l'indirizzo dell'istruzione di salto
- se la **previsione è di saltare**, le istruzioni successive 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



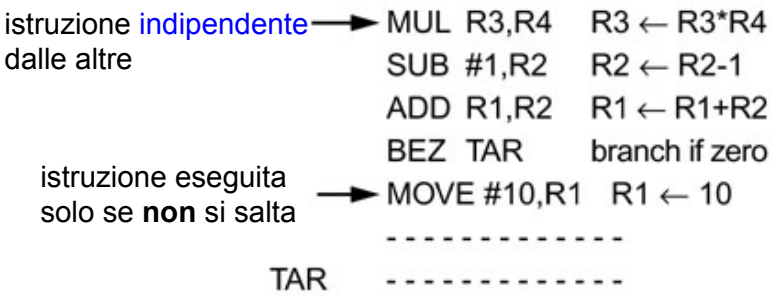
(b) Branch history table strategy

Soluzioni per salti condizionati

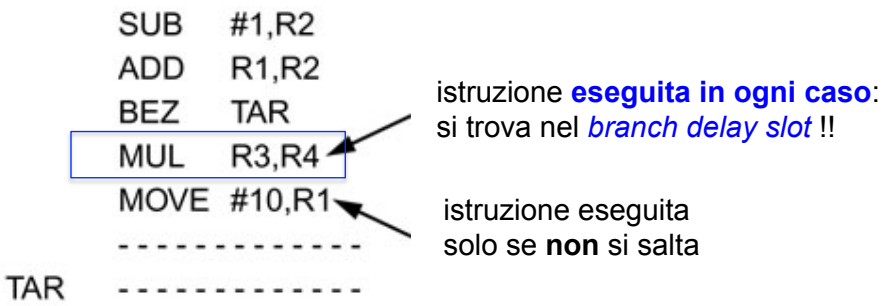
1. flussi multipli (*multiple streams*)
2. prefetch dell'istruzione target
3. buffer circolare *(loop buffer)*
4. predizione dei salti
5. **salto ritardato** *(delayed branch)*
   - finche' non si sa se ci sarà o no il salto (l'istruzione è in pipeline), invece di restare in stallo si può **eseguire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

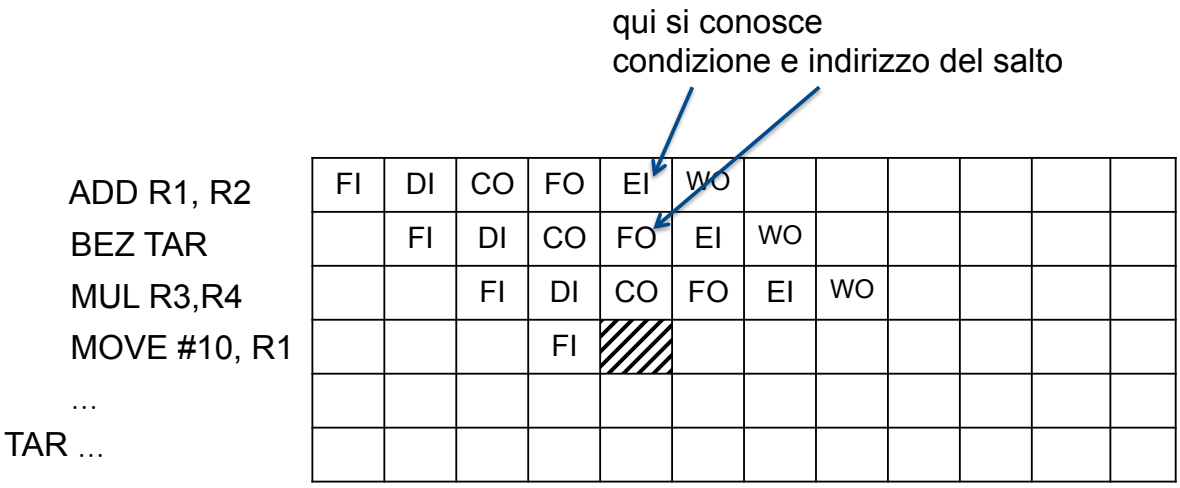
Salto ritardato (delayed branch)

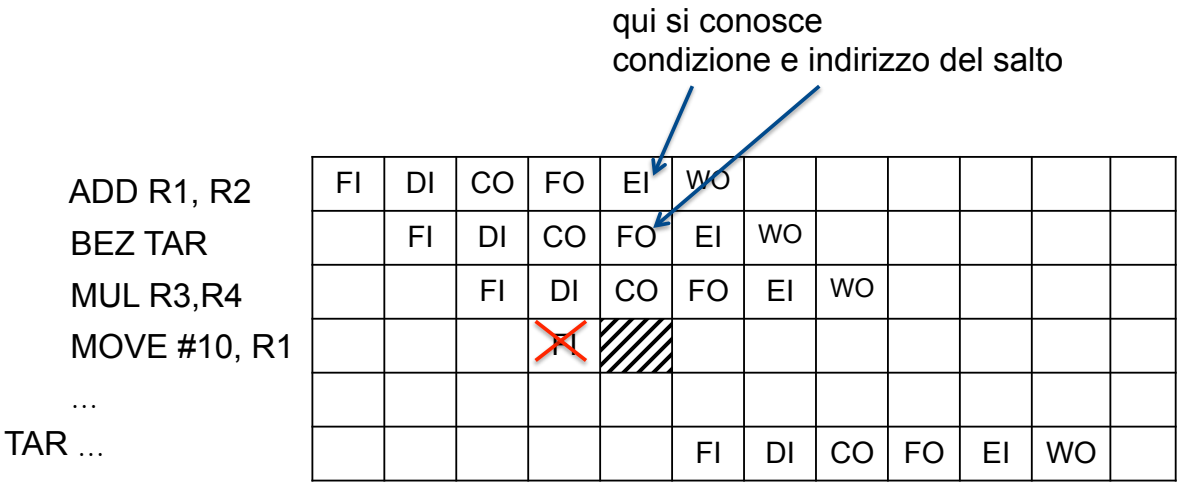
codice scritto dal programmatore



codice ottimizzato dal compilatore

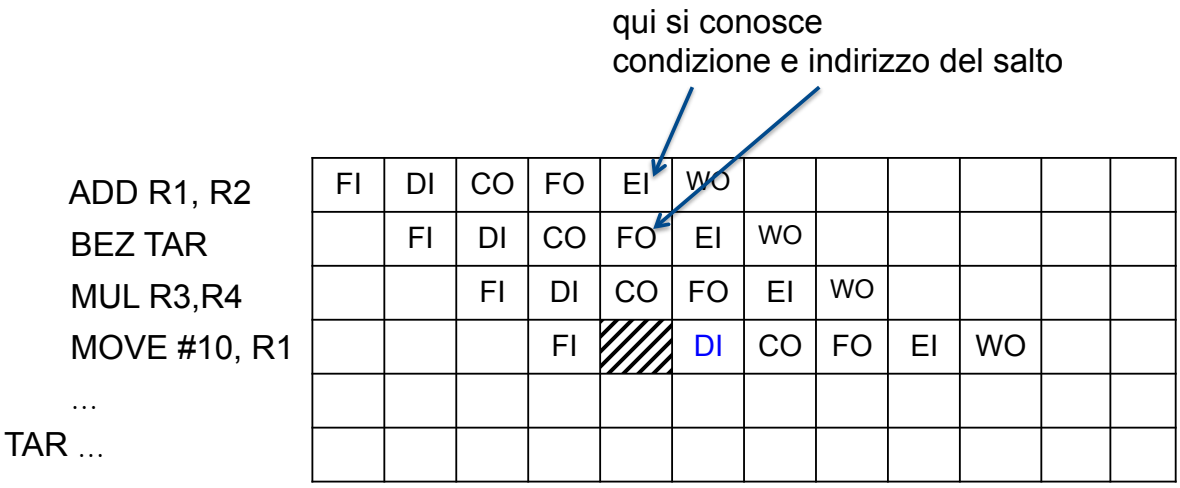






se la **condizione è vera** e prende il salto
scarta l'istruzione errata e riprende da TAR

penalità 2 cicli: 1 fetch inutile e 1 stallo



se la **condizione è falsa** e non prende il salto
continua con l'istruzione prefetched

penalità 1 ciclo di stallo

Salto ritardato (delayed branch)

ADD R1, R2, R3
if R2 = 0 then Tar
delay slot
Tar…

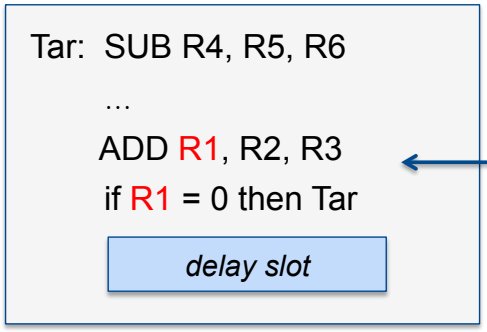
ottimizzazione

if R2 = 0 then Tar
ADD R1, R2, R3
Tar…

"from before"

quando è possibile riempie il branch delay slot con un'istruzione **indipendente** proveniente dalla parte di codice che *precede* il salto

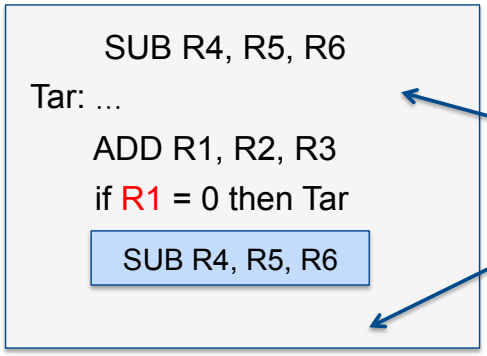
Salto ritardato (delayed branch)



"from target"
utile quando è probabile che il salto sia preso (es. in loop)

non sono più istruzioni indipendenti

riempie il branch delay slot con **l'istruzione target** del salto, che normalmente viene **copiata** perche' potrebbe essere accessibile anche tramite un altro cammino



eseguo ***sempre*** l'istruzione nel delay slot, quindi:

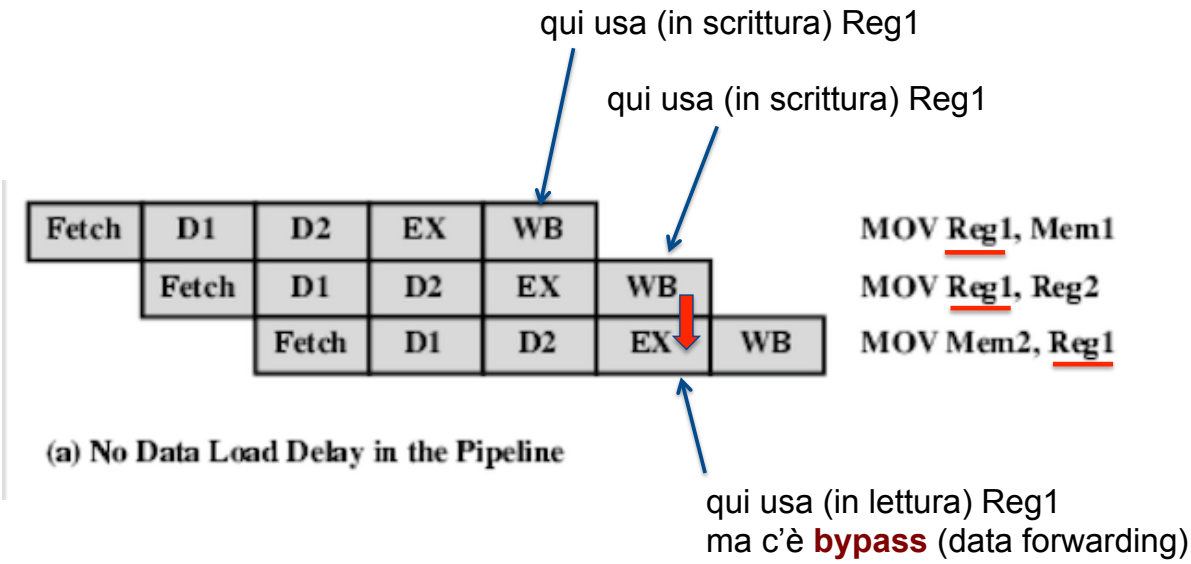
- **quando salto** vado alla successiva
- **quando non salto** procedo oltre, ma è **corretto** solo se **l'istruzione nel delay slot** è inutile ma **non dannosa** (es. R4 non usato se non salta)

Intel 80486 Pipelining

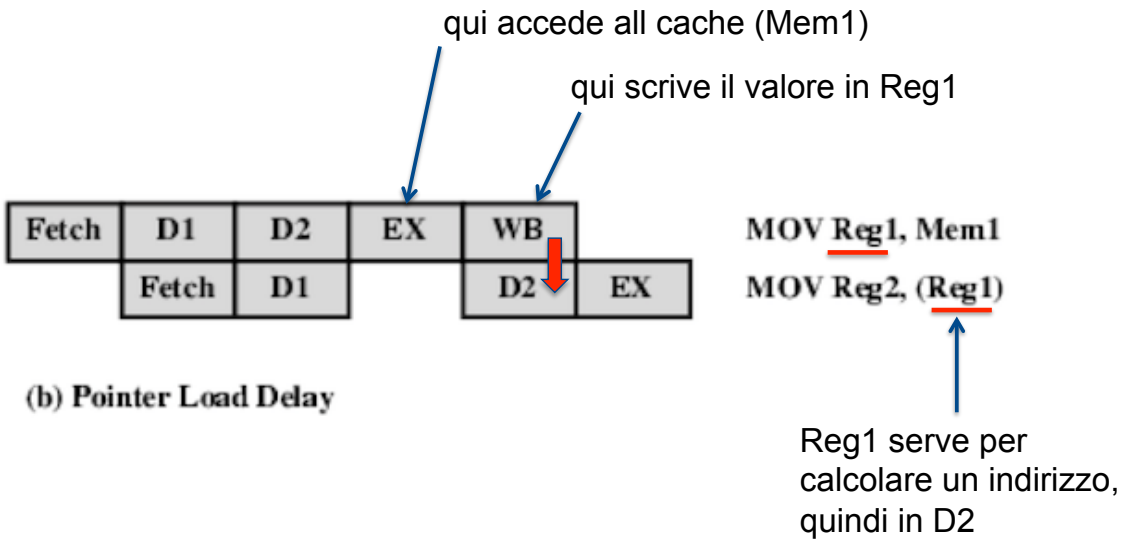
- Fetch
  - Istruzioni prelevate dalla cache o memoria esterna
  - Poste in uno dei due buffer di prefetch da 16 byte
  - Carica dati nuovi appena quelli vecchi sono "consumati"
  - Poiché le istruzioni sono a lunghezza variabile (1-11 byte), in media carica 5 istruzioni per ogni caricamento da 16 byte
  - 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
  - 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 (es. storia dei salti)
  - 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 alla memoria non introducono ritardi



80486 Instruction Pipeline: esempi



con **bypass** (data forwarding)
c'è un solo ciclo di stallo

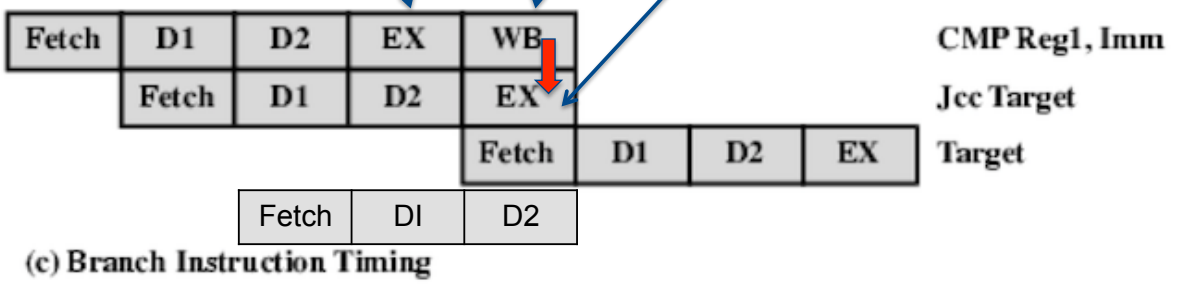
80486 Instruction Pipeline: esempi

salto condizionato. Assumiano venga eseguito

qui effettua il confronto

qui aggiorna il condition flag

controlla la condizione di salto usando il **bypass** (data forwarding)



(c) Branch Instruction Timing

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