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
```

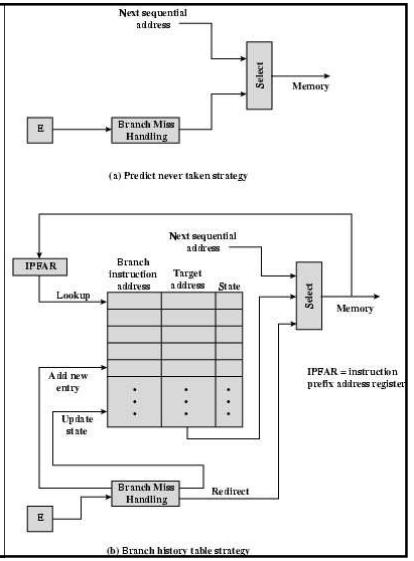
- **Predizione con 1 bit**: si predice il comportamento osservato l'ultima volta
  - dopo la prima esecuzione del ciclo, in uscita dal ciclo, il bit assegnato a BNZ ricorderà che il salto **non è stato preso**, così che, quando si rientra nel ciclo si avrà un primo errore per la prima iterazione del ciclo (che invece è preso), le successive predizioni saranno giuste, tranne l'ultima, quando si esce dal ciclo: in totale **2 errori**
- **Predizione con 2 bit**: vedi lucido precedente
  - dopo la prima esecuzione del ciclo, si commette **un solo errore** di predizione all'uscita del ciclo

Problemi di bit *taken/not taken* :

- quando si decide di saltare, bisogna aspettare la decodifica dell'indirizzo destinazione prima di poter prelevare l'istruzione destinazione
- si può anticipare il prelievo a patto di salvare opportune info nel *branch target buffer* o *branch history table*

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



Salto ritardato (delayed branch)

Idea base: utilizzare gli stadi inattivi a causa dello stallo per fare del lavoro utile

Delayed branch:

- La CPU esegue **sempre** l'istruzione che segue il salto e solo dopo altera, se necessario, la sequenza di esecuzione delle istruzioni
- L'istruzione che segue quella di salto si dice essere posta nel *branch delay slot*
- Il **compilatore** cerca di allocare nel *branch delay slot* una istruzione "opportuna"

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 si prende il salto →  MOVE #10,R1  R1 ← 10
                                       ----------
                                  TAR  ----------
```

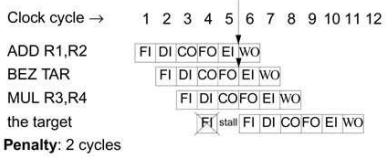
codice ottimizzato dal compilatore

```
        SUB   #1,R2
        ADD   R1,R2
        BEZ   TAR
        MUL   R3,R4     ← istruzione eseguita in ogni caso: si trova nel branch delay slot !!
        MOVE  #10,R1    ← istruzione eseguita solo se si prende il salto
        ----------
TAR     ----------
```

Salto ritardato (delayed branch)

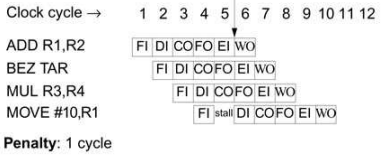
Salto preso

condizione ed indirizzo di salto conosciuti qui

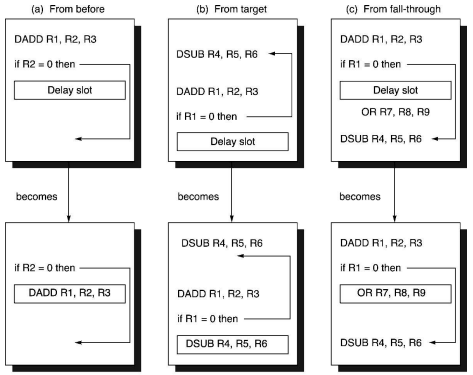


Salto non preso

condizione di salto conosciuta qui



Salto ritardato (delayed branch)



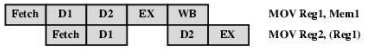
b) e c) legali solo se R4 e R7 sono registri temporanei il cui contenuto può essere "sporcato" senza cambiare la semantica del programma

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
  - Independente 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
  - Se l'istruzione corrente aggiorna la memoria, pone il valore calcolato in cache e nei buffer di scrittura del bus

80486 Instruction Pipeline: esempi

(a) No Data Load Delay in the Pipeline



(b) Pointer Load Delay

(c) Branch Instruction Timing

Esercizio: Dipendenze

Dipendenza dai dati : per la quale l'istruzione *j* dipende dall'istruzione *i* se *i* produce, direttamente o transitivamente (ossia tramite una o più istruzioni intermedie) un risultato richiesto da *j*.

Dipendenza dal controllo : la quale determina l'ordinamento di una istruzione rispetto ad un salto condizionale, così che essa esegua solo quando dovuto rispetto all'esecuzione del salto.

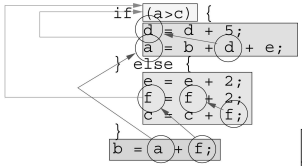
Dipendenza dai nomi : la quale ha luogo allorchè due istruzioni, tra le quali non vi sia flusso di dati, usano lo stesso registro o la stessa locazione di memoria.

Si illustrino tutte le dipendenze presenti nel seguente frammento di programma in linguaggio C, assumendo che il programma non faccia riferimento ad altri dati, che tutti i valori siano definiti prima dell'uso, e che soltanto `b` e `c` siano usati successivamente alle istruzioni date:

```
if (a>c) {
    d = d + 5;
    a = b + d + e; }
else {
    e = e + 2;
    f = f + 2;
    c = c + f; }
b = a + f;
```


Soluzione

A
B | 1
2
C | 1
2
3
D



dipendenze		
dal controllo	**dai dati**	**dai nomi**
B → A	B.2 → B.1	
C → A	C.3 → C.2	
	D → C.2,B.1 (via B.2)	

legenda:

dipendenza dal controllo: X dipende da Y

dipendenza dai dati: X dipende da Y

Esercizio: valutazione delle prestazioni

- Si considerino le seguenti statistiche:
  - 15% delle istruzioni sono di salto condizionale
  - 1% delle istruzioni sono di salto incondizionale
  - Il 60% delle istruzioni di salto condizionale hanno la condizione soddisfatta (prese)
- …ed una pipeline a 4 stadi (IF, ID, EI, WO) per cui:
  - i salti incondizionati sono risolti (identificazione salto e calcolo indirizzo target) alla fine del secondo stadio (ID)
  - i salti condizionati sono risolti (identificazione salto, calcolo indirizzo target e calcolo condizione) alla fine del terzo stadio (EI)
  - il primo stadio (IF) è indipendente dagli altri
- inoltre si assuma che non ci siano altre istruzioni che possano mandare in stallo la pipeline e che non sia implementato alcun meccanismo di trattamento dei salti

Domanda:

calcolare quanto più veloce, a regime, sarebbe la pipeline senza gli stalli introdotti dai salti

Aiuto: fattore di velocizzazione di una pipeline a k stadi, a regime, in funzione del numero di stalli:

$$S_k = \frac{1}{1+cicli\_stallo}\, k$$