Struttura e Funzione del Processore

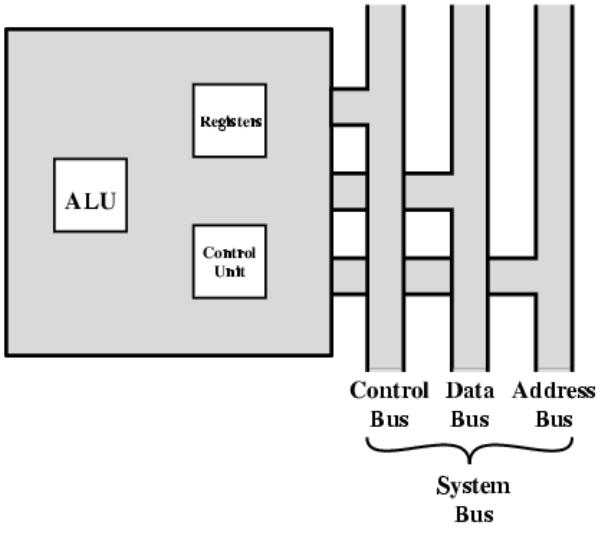
Capitolo 12

Struttura CPU

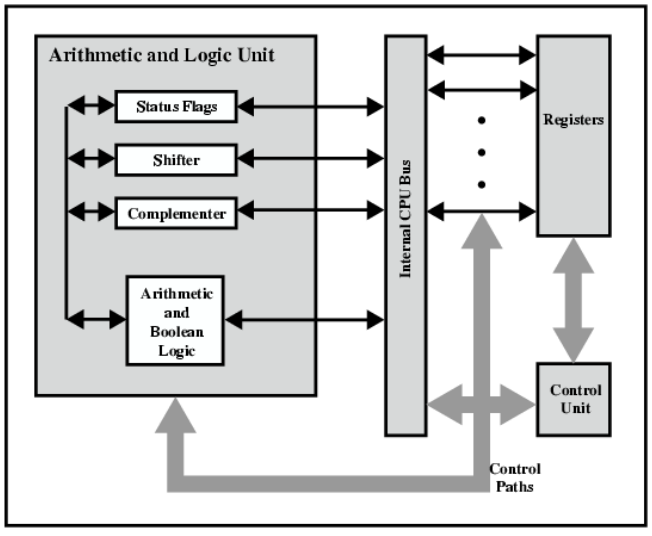
Compiti CPU:

- Prelevare istruzioni 
- Interpretare istruzioni 
- Prelevare dati 
- Elaborare dati 
- Scrivere (memorizzare) dati 

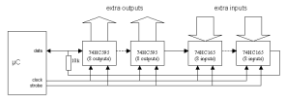
CPU con bus di sistema



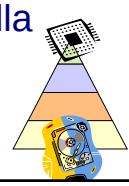
Struttura interna CPU



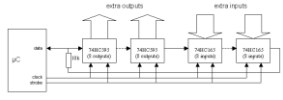
Registri



- CPU ha bisogno di uno “spazio di lavoro” dove memorizzare i dati
- Questo “spazio di lavoro” è costituito dai **registri**
- Numero e funzioni svolte dai registri varia a seconda dell’impianto progettuale della CPU
- Scelta progettuale molto importante
- I registri costituiscono il livello più alto della così detta “Gerarchia della memoria”



Registri



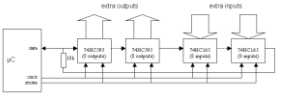
- Registri utente
  - usati dal “programmatore” per memorizzare internamente alla CPU i dati da elaborare
- Registri di controllo e di stato
  - usati dall’unità di controllo per monitorare le operazioni della CPU
  - usati dai programmi del Sistema Operativo (SO) per controllare l’esecuzione dei programmi

“programmatore”

1. Umano che programma in assembler
2. Compilatore che genera codice assembler a partire da un programma scritto in un linguaggio ad alto livello (C, C++, Java,…)

Ricordarsi che un programma in assembler è trasformato in codice macchina dall’assemblatore (+ linker) che trasforma il codice mnemonico delle istruzioni in codice macchina

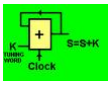
Registri visibili all’utente: registri utente



- Ad uso generale
- Per la memorizzazione di dati
- Per la memorizzazione di indirizzi
- Per la memorizzazione di codici di condizione

Registri ad uso generale

- Possono essere veramente ad uso generale
- …oppure dedicati a funzioni particolari
- Possono essere usati per contenere dati o indirizzi
- Dati
  - Ad esempio: accumulatore



- Indirizzi
  - Ad esempio: indirizzo base di un segmento di memoria

Segmento di memoria

- La memoria principale può essere organizzata, dal punto di vista logico (cioè concettuale), come un insieme di segmenti o spazi di indirizzamento multipli:
  - "visibili" al "programmatore", che riferisce logicamente una locazione di memoria riferendo il segmento e la posizione della locazione all'interno del segmento:
    es. segmento 4, locazione 1024
  - come supporto a questa "visione" della memoria, occorre poter indicare **dove**, all'interno della memoria fisica, inizia il segmento (*base*) e la sua **lunghezza** (*limite*)
    es. il segmento 4 ha base = $00EF9445_{hex}$ e limite = 4MB
  - quindi occorrono dei registri dove memorizzare tali informazioni

Registri ad uso generale

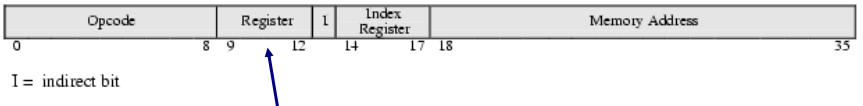
- Registri veramente ad uso generale
  - Aumentano la flessibilità e le opzioni disponibili al programmatore “a basso livello”
  - Aumentano la dimensione dell’istruzione e la sua complessità (perché ?)
- Registri specializzati
  - Istruzioni più piccole e più veloci
  - Meno flessibili

Perché aumenta dimensione e complessità ?

- Facciamo l’esempio di istruzioni a formato fisso:

PDP-10

Campo non necessario se non avessi registri generali: *formato semplificato !*



I = indirect bit

4 bit ➡ 2^4 = 16 registri generali
Se si avessero 32 registri generali sarebbero necessari 5 bit di indirizzamento

Formato più lungo! 37 bit ⇦

Quanti registri generali?

- Tipicamente tra 8 e 32
- Meno di 8 = più riferimenti (accessi) alla memoria principale (perché ?)
- Più di 32 non riducono i riferimenti alla memoria ed occupano molto spazio nella CPU
- Nelle architetture RISC tipicamente si hanno più di 32 registri generali

Perché più accessi ?

ESEMPIO: supponiamo di dover calcolare:

```
mem[4] = mem[0]+mem[1]+mem[2]+mem[3]
mem[5] = mem[0]*mem[1]*mem[2]*mem[3]
mem[6] = mem[5]-mem[4]
```

```
 4 registri
R0 ← mem[0];
R1 ← mem[1];
R2 ← mem[2];
R3 ← mem[3];
R0 ← R0+R1;
R0 ← R0+R2;
R0 ← R0+R3;
R1 ← R1*R2;
R1 ← R1*R3;
R2 ← mem[0];
R1 ← R1*R2;
R0 ← R1-R0;
```

7 op
5 mem

```
 6 registri
R0 ← mem[0];
R1 ← mem[1];
R2 ← mem[2];
R3 ← mem[3];
R4 ← R0+R1;
R4 ← R2+R4;
R4 ← R3+R4;
R5 ← R0*R1;
R5 ← R2*R5;
R5 ← R3*R5;
R0 ← R5-R4;
```

7 op
4 mem

Quanto lunghi (in bit) ?

- Abbastanza grandi da contenere un indirizzo della memoria principale
- Abbastanza grandi da contenere una “full word”
- E’ spesso possibile combinare due registri dati in modo da ottenerne uno di dimensione doppia
  - Es.: programmazione in C
    - double int a;
    - long int a;

Registri per la memorizzazione di Codici di Condizione

- Insiemi di bit individuali
  - es. Il risultato dell’ultima operazione era zero
- Possono essere letti (implicitamente) da programma
  - es. “Jump if zero” (salta se zero)
- Non possono (tipicamente) essere impostati da programma

Registri di Controllo e di Stato

- Program Counter (PC)
- Instruction Register (IR)
- Memory Address Register (MAR)
- Memory Buffer Register (MBR)

Program Status Word

- Un insieme di bit

	0	0	0	0	0	0	0	0	Value after reset
PSW	CY	AC	F0	RS1	RS0	OV		P	Bit name
	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	

- Include Codici di Condizione
  - Segno dell'ultimo risultato
  - Zero
  - Riporto
  - Uguale
  - Overflow
  - Abilitazione/disabilitazione Interrupt
  - Supervisore

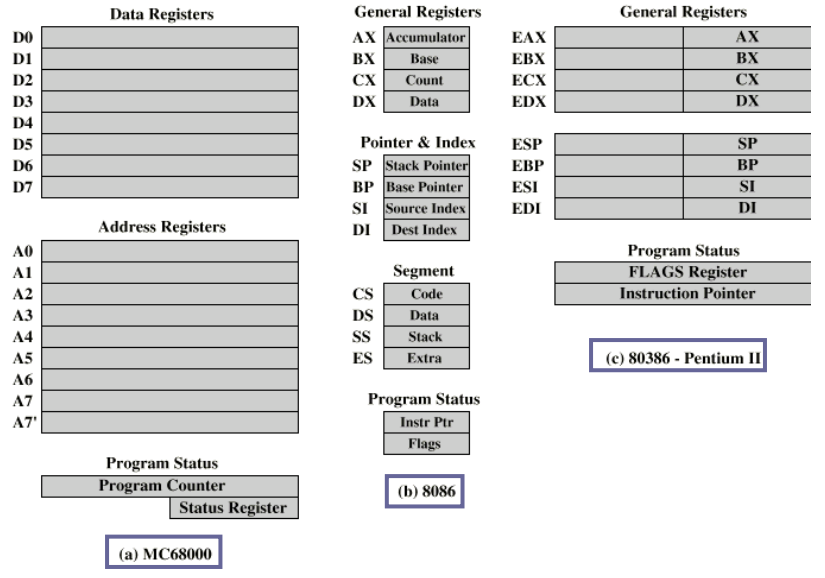
Modo Supervisore

- Permette al sistema operativo di utilizzare le procedure del Kernel, che agiscono su componenti critiche del sistema
- In particolare permette l'esecuzione di istruzioni "privilegiate"
- Disponibile esclusivamente al sistema operativo
- Non disponibile all'utente programmatore
- Lo studierete in dettaglio nel corso di Sistemi Operativi

Altri registri

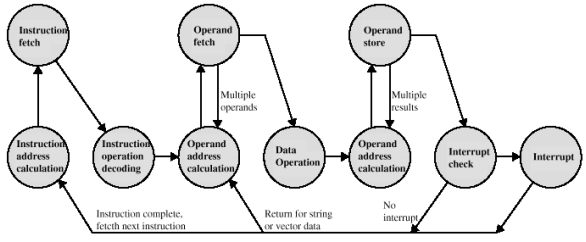
- Ci possono essere registri che puntano a:
  - Process control blocks (sistemi operativi)
  - Interrupt Vectors (sistemi operativi)
  - Tabella delle pagine della memoria virtuale
- La progettazione della CPU e quella dei sistemi operativi sono strettamente correlate

Esempi di Organizzazione di Registri



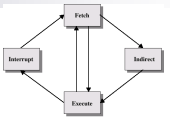
Ciclo esecutivo delle istruzioni: Fetch/Execute

- Lo avete già visto



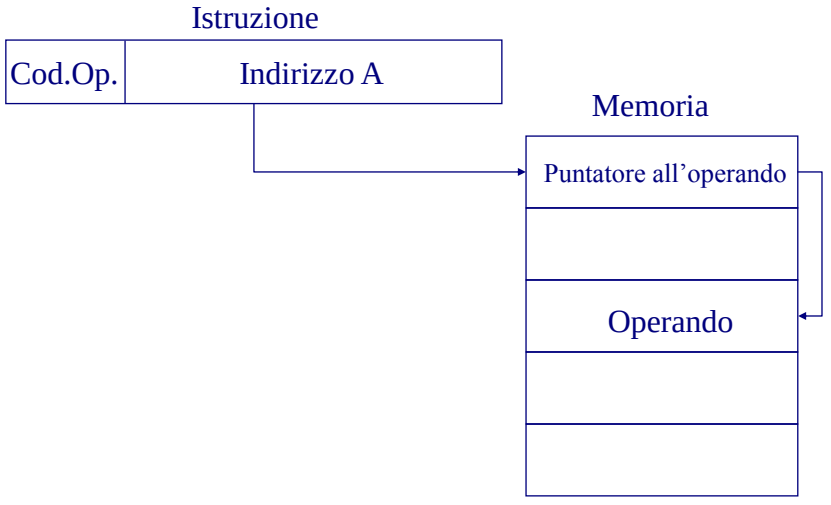
- Stallings, Capitolo 3
- Ne vediamo una versione revisionata

Indirettezza

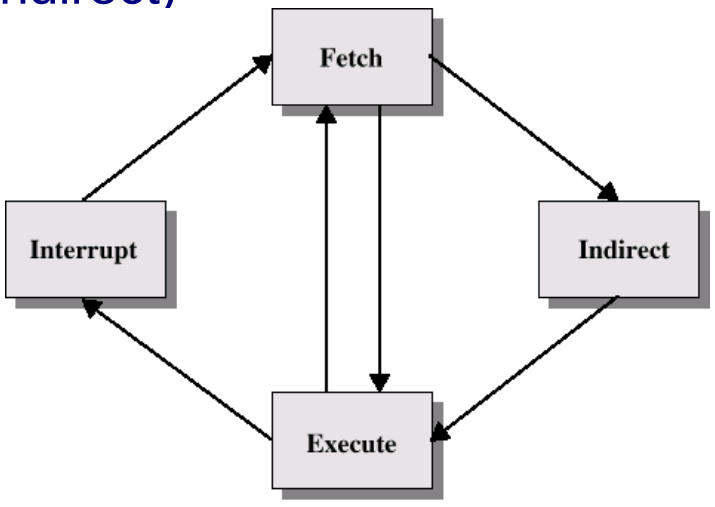


- Per recuperare gli operandi di una istruzione può essere necessario accedere alla memoria
- La modalità di indirizzamento indiretto per specificare la locazione in memoria degli operandi richiede più accessi in memoria
- L'indirettezza si può considerare come un sottociclo del ciclo fetch/execute

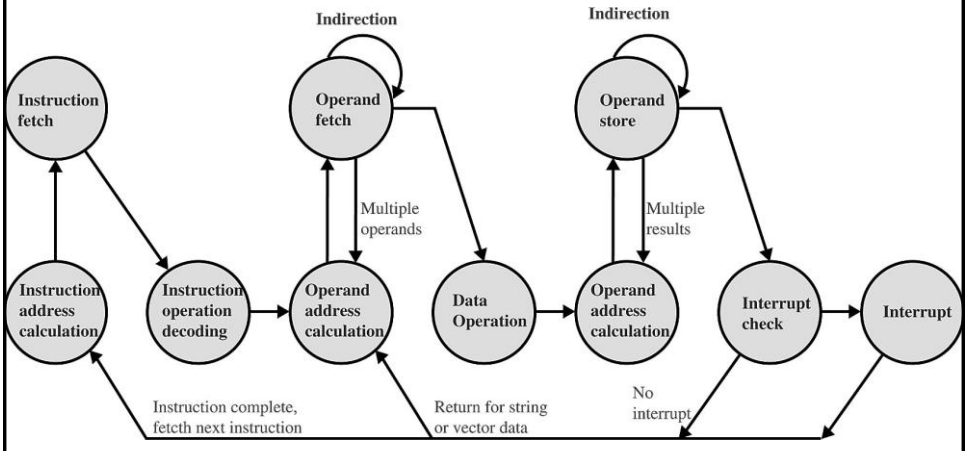
Indirizzamento indiretto



Introduzione della Indirettezza (indirect)



Fetch/Execute: ancora più in dettaglio

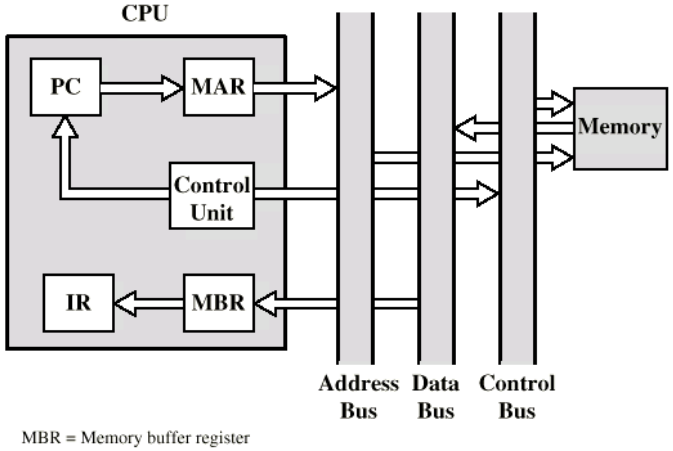


Flusso dei dati (Instruction Fetch)

Dipende dalla architettura della CPU, in generale:

- Fetch
  - PC contiene l'indirizzo della istruzione successiva
  - Tale indirizzo viene spostato in MAR
  - L'indirizzo viene emesso sul bus degli indirizzi
  - La unità di controllo richiede una lettura in memoria principale
  - Il risultato della lettura in memoria principale viene inviato nel bus dati, copiato in MBR, ed infine in IR
  - Contemporaneamente il PC viene incrementato

Flusso dei dati (Diagramma di Fetch)

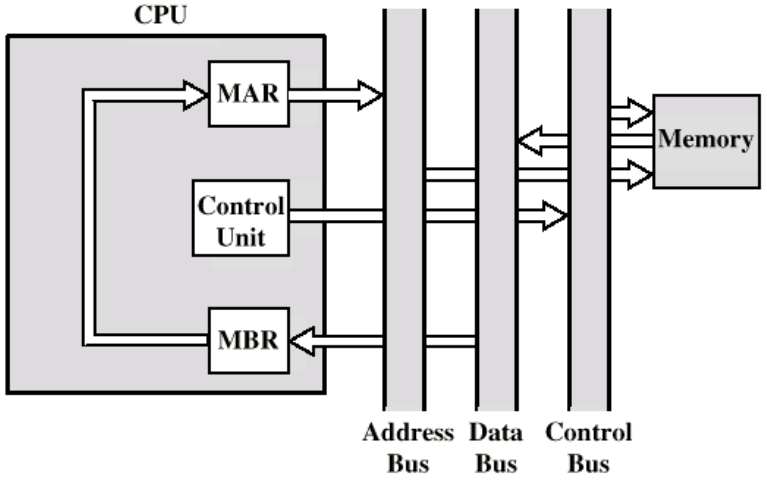


MBR = Memory buffer register
MAR = Memory address register
IR = Instruction register
PC = Program counter

Flusso dei dati (Data Fetch)

- IR è esaminato
- Se il codice operativo della istruzione richiede un indirizzamento indiretto, si esegue il ciclo di indirettezza
  - gli N bit più a destra di MBR vengono trasferiti nel MAR
  - L'unità di controllo richiede la lettura dalla memoria principale
  - Il risultato della lettura (indirizzo dell'operando) viene trasferito in MBR

Flusso dei dati (Diagramma di Indirettezza)



Flusso dei dati (Execute)

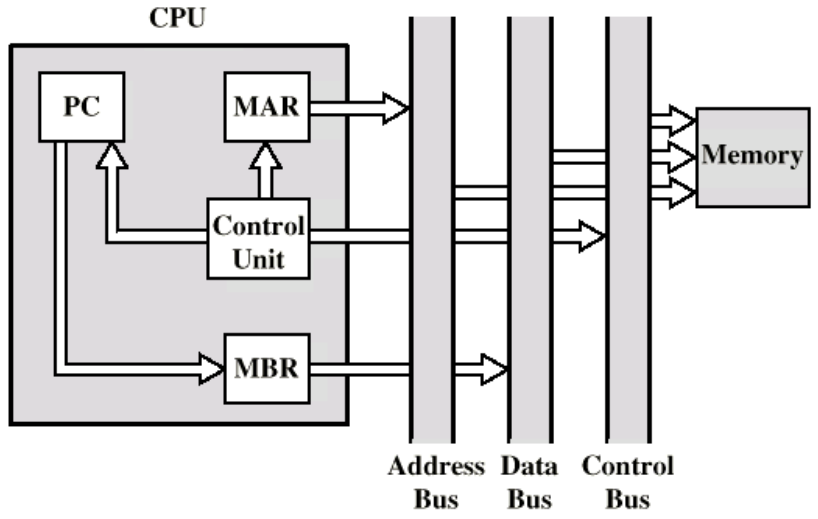
- Può assumere molte forme
- Dipende dalla istruzione da eseguire
- Può includere
  - lettura/scrittura della memoria
  - Input/Output
  - Trasferimento di dati fra registri e/o in registri
  - Operazioni della ALU

Flusso dei dati (Interrupt)

Semplice e prevedibile:

- Contenuto corrente del PC deve essere salvato per permettere il ripristino della esecuzione dopo la gestione dell'interruzione
  - Contenuto PC copiato in MBR
  - Indirizzo di locazione di memoria speciale (es. stack pointer) caricato in MAR
  - Contenuto di MBR scritto in memoria
- PC caricato con l'indirizzo della prima istruzione della routine di gestione della interruzione
- Fetch della istruzione puntata da PC

Flusso dei dati (Diagram. di Interrupt)



Prefetch

- La fase di prelievo della istruzione accede alla memoria principale
- La fase di esecuzione di solito non accede alla memoria principale
- Si può prelevare l'istruzione successiva durante l'esecuzione della istruzione corrente
- Questa operazione si chiama "instruction prefetch"

Prefetch

(a) Simplified view

Miglioramento delle prestazioni

- Il prefetch non raddoppia le prestazioni:
  - L'esecuzione di istruzioni jump o branch possono rendere vano il prefetch (perché ?)
  - La fase di prelievo è tipicamente più breve della fase di esecuzione
    - Prefetch di più istruzioni ?
- Aggiungere più fasi per migliorare le prestazioni

Il prefetch può essere inutile perché…

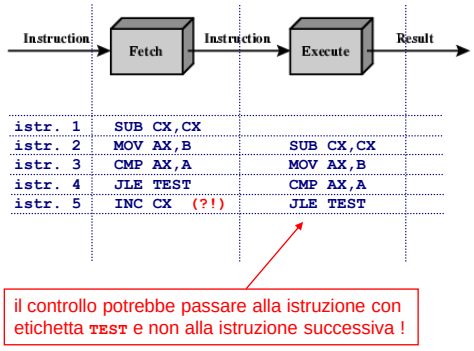
Per il seguente costrutto

```
if (A > B) then
```

un compilatore potrebbe generare il seguente codice 80x86

```
      SUB CX,CX; CX ← 0
      MOV AX,B ; AX ← mem[B]
      CMP AX,A ; paragona [AX] con mem[A]
      JLE TEST ; salta se A ≤ B
      INC CX   ; CX ← CX+1
TEST  JCXZ OUT ; salta se [CX] è 0
THEN

OUT
```



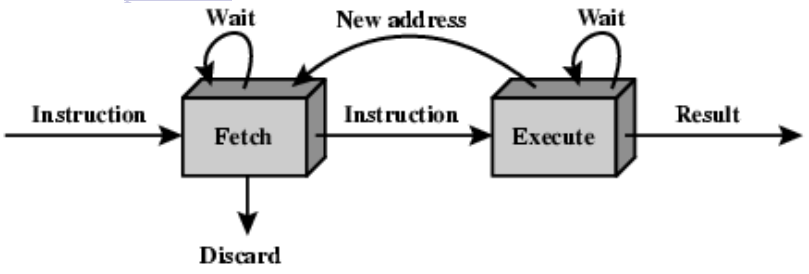
il controllo potrebbe passare alla istruzione con etichetta `TEST` e non alla istruzione successiva !

… si deve caricare una istruzione diversa dalla successiva !

Prefetch

(a) Simplified view

perché ?

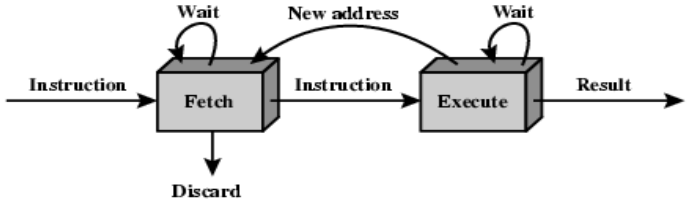


(b) Expanded view

perché…

su processore 286

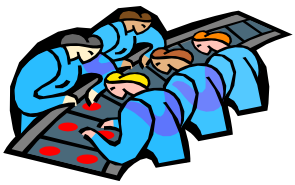
	istruzione	cicli di clock impiegati dall'istr.
istr. 1	SUB CX,CX	2
istr. 2	MOV AX,B	5
istr. 3	CMP AX,A	6
istr. 4	JLE TEST	3 se non salta,>7 altrimenti
istr. 5	INC CX	2
istr. 6	JCXZ	4 se non salta,>8 altrimenti



(b) Expanded view

Evoluzione delle architetture
Evoluzione strutturale

- **Parallelismo**
  - Se un lavoro non può essere svolto più velocemente da una sola persona (unità), allora conviene **decomporlo** in parti che possano essere eseguite da più persone (unità) **contemporaneamente**
    - **Catena di montaggio**



Pipeline
Generalità 1

- Ipotizziamo che per svolgere un dato lavoro **L** si debbano eseguire tre fasi distinte e sequenziali

  $$\mathbf{L} \Rightarrow [\mathit{fase1}]\ [\mathit{fase2}]\ [\mathit{fase3}]$$

- Se ogni fase richiede **T** unità di tempo, un unico esecutore svolge un lavoro **L** ogni **3T** unità di tempo
- Per ridurre i tempi di produzione si possono utilizzare **più esecutori**

Pipeline
Generalità 2

- Soluzione (ideale) a parallelismo totale

  E1 ⇒ [fase1.A] [fase2.A] [fase3.A] | [fase1.D] …

  E2 ⇒ [fase1.B] [fase2.B] [fase3.B] | [fase1.E] …

  E3 ⇒ [fase1.C] [fase2.C] [fase3.C] | [fase1.F] …

- **N** esecutori svolgono un lavoro ogni **3T/N** unità di tempo
- Il problema è **come** preservare la **dipendenza funzionale** nell'esecuzione (di fasi) dei 'lavorì **A**, **B**, **C**, **D**, **E**, **F**, …

Pipeline
Generalità 3

- Soluzione **pipeline** ad **esecutori generici**

E1 ⇒ [fase1] [fase2] [fase3] [fase1] [fase2]

E2 ⇒ ..... [fase1] [fase2] [fase3] [fase1]

E3 ⇒ ..... [fase1] [fase2] [fase3]

- Ogni esecutore esegue un ciclo di lavoro **completo** (*sistema totalmente replicato*)
- A regime, **N** esecutori svolgono un lavoro **L** ogni **3T/N** unità di tempo rispettandone la sequenza

Pipeline
Generalità 4

- Soluzione **pipeline** ad **esecutori specializzati**

E1 ⇒ [fase1] [fase1] [fase1] [fase1] [fase1]

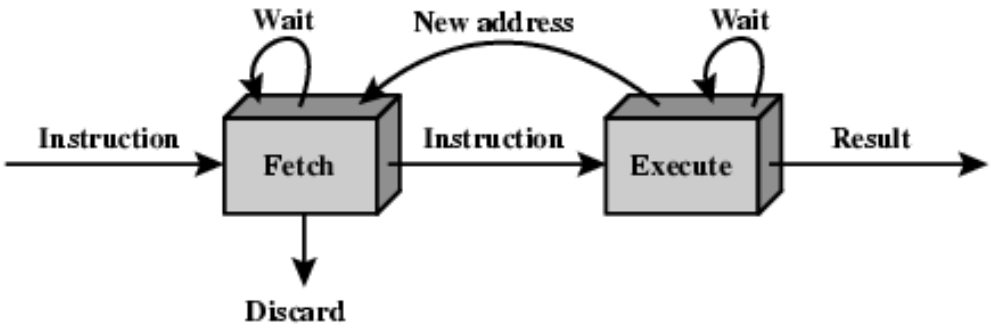
E2 ⇒ ..... [fase2] [fase2] [fase2] [fase2]

E3 ⇒ ..... [fase3] [fase3] [fase3]

- Ogni esecutore svolge sempre e solo la **stessa** fase di lavoro
- Soluzione più efficace in termini di **uso di risorse** (**3T/N** lavori con **N/3** risorse)

Prefetch come pipeline a due stadi

(a) Simplified view

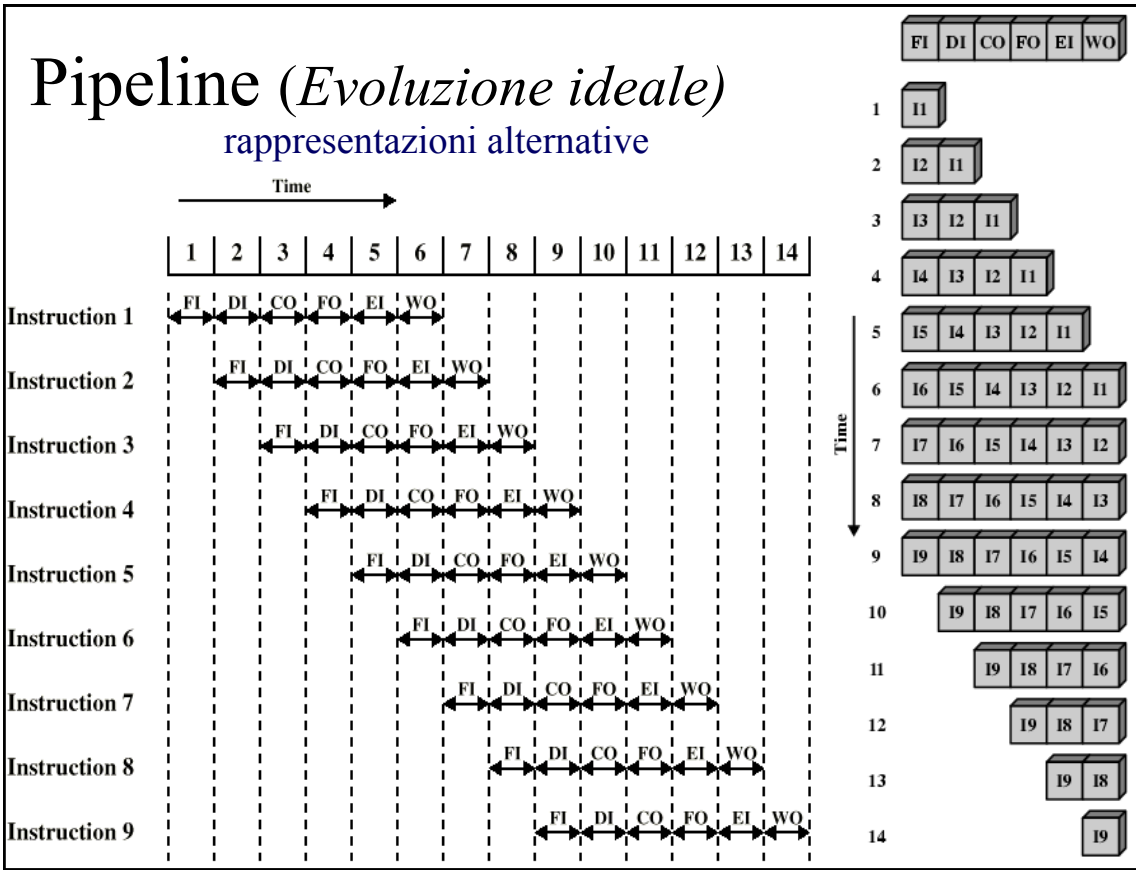


(b) Expanded view

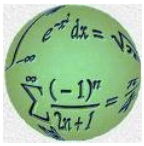
Pipeline
Decomposizione in fasi

- L'esecuzione di una generica istruzione può essere suddivisa nelle seguenti fasi:
  - **fetch** (FI) lettura dell'istruzione
  - **decodifica** (DI) decodifica dell'istruzione
  - **calcolo ind. op.** (CO) calcolo indirizzo effettivo operandi
  - **fetch operandi** (FO) lettura degli operandi in memoria
  - **esecuzione** (EI) esecuzione dell'istruzione
  - **scrittura** (WO) scrittura del risultato in memoria

Pipeline (*Evoluzione ideale)*
rappresentazioni alternative



Pipeline prestazioni ideali



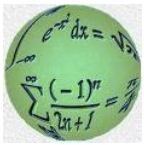
Le prestazioni ideali di una pipeline si possono calcolare matematicamente come segue

- Sia τ il tempo di ciclo di una pipeline necessario per far avanzare di uno stadio le istruzioni attraverso una pipeline. Questo può essere determinato come segue:

$$\tau = \max_i [\tau_i] + d = \tau_m + d \quad 1 \le i \le k$$

  - τ_m = massimo ritardo di stadio (ritardo dello stadio più oneroso)
  - k = numero di stadi nella pipeline
  - d = ritardo di commutazione di un registro, richiesto per l'avanzamento di segnali e dati da uno stadio al successivo

Pipeline prestazioni ideali



Poiché $\tau_m >> d$, il tempo totale T_k richiesto da una pipeline con k stadi per eseguire n istruzioni (senza considerare salti ed in prima approssimazione) è dato da

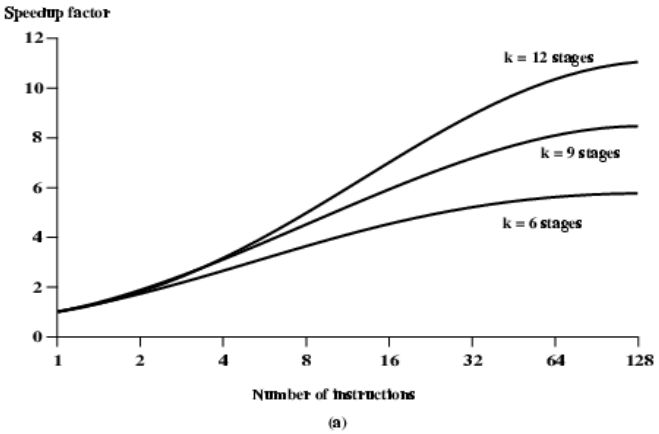
$$T_k = [k + (n\text{-}1)]\tau$$

in quanto occorrono k cicli per completare l'esecuzione della prima istruzione e n-1 per le restanti istruzioni, e quindi il *fattore di velocizzazione* (speedup) di una pipeline a k stadi è dato da:

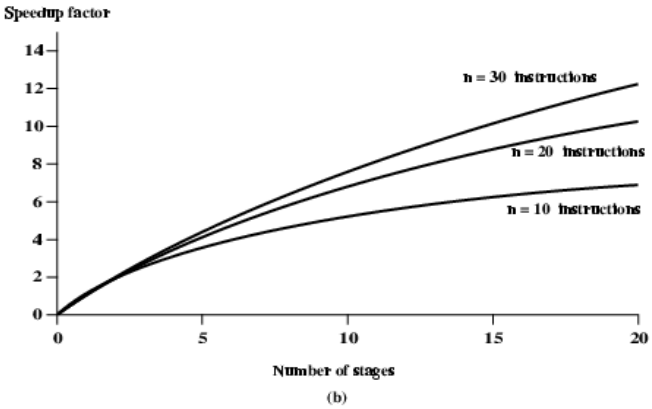
$$S_k = \frac{T_1}{T_k} = \frac{nk\tau}{[k+(n-1)]\tau} = \frac{nk}{[k+(n-1)]}$$

Speedup

Calcolato in funzione del numero di istruzioni



Calcolato in funzione del numero di stadi



Pipeline Problemi 1

- Vari fenomeni pregiudicano il raggiungimento del massimo di parallelismo teorico (**stallo**)
  - **Sbilanciamento delle fasi**
    - Durata diversa per fase e per istruzione
  - **Problemi strutturali**
    - La sovrapposizione totale di tutte le (fasi di) istruzioni causa conflitti di accesso a risorse limitate e condivise (ad esempio la memoria per gli stadi FI, FO, WO)

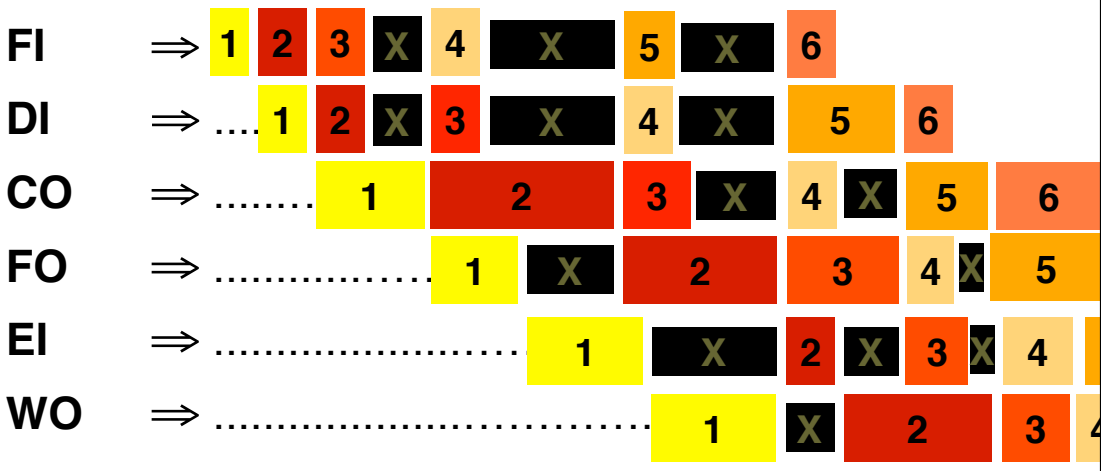
Pipeline
Problemi 2

- **Dipendenza dai dati**
  - L'operazione successiva dipende dai risultati dell'operazione precedente
- **Dipendenza dai controlli**
  - Istruzioni che causano una violazione di sequenzialità (p.es.: salti condizionali) invalidano il principio del *pipelining* sequenziale

Pipeline
Sbilanciamento delle fasi 1

- La suddivisione in fasi va fatta in base all'istruzione più onerosa
- Non tutte le istruzioni richiedono le stesse fasi e le stesse risorse
- Non tutte le fasi richiedono lo stesso tempo di esecuzione
  - P.es.: lettura di un operando tramite registro rispetto ad una mediante indirizzamento indiretto

Pipeline
Sbilanciamento delle fasi 2



X Tempo di attesa forzata dovuta allo sbilanciamento delle fasi

Pipeline
Sbilanciamento delle fasi 3

Possibili soluzioni allo sbilanciamento:

- Decomporre fasi onerose in più sottofasi
  - Costo elevato e bassa utilizzazione
- Duplicare gli esecutori delle fasi più onerose e farli operare in parallelo
  - CPU moderne hanno una ALU in aritmetica intera ed una in aritmetica a virgola mobile

Pipeline
Problemi strutturali

Problemi

- Maggiori risorse interne (**severità bassa**): l'evoluzione tecnologica ha spesso permesso di duplicarle (es. registri)

- Colli di bottiglia (**severità alta**): l'accesso alle risorse esterne, p.es.: memoria, è molto costoso e molto frequente (anche 3 accessi per ciclo di clock)

Soluzioni

- Suddividere le memorie (accessi paralleli: introdurre una memoria cache per le istruzioni e una per i dati)
- Introdurre fasi non operative (***nop***)

Pipeline
Dipendenza dai dati 1

- Un dato modificato nella fase **EI** dell'istruzione corrente può dover essere utilizzato dalla fase **FO** dell'istruzione successiva

  INC [0123]

  CMP [0123], AL

Ci sono altri tipi di dipendenze ?

Dipendenze

Si consideri la sequenza

```
istruzione i
istruzione j
```

Esempio visto: “lettura dopo scrittura” (ReadAfterWrite)

– ***j*** leggere prima che ***i*** abbia scritto

Altro caso: “scrittura dopo scrittura” (WriteAfterWrite)

– ***j*** scrive prima che ***i*** abbia scritto

Altro caso: “scrittura dopo lettura” (WriteAfterRead)

– ***j*** scrive prima che ***i*** abbia letto (caso raro in pipeline)

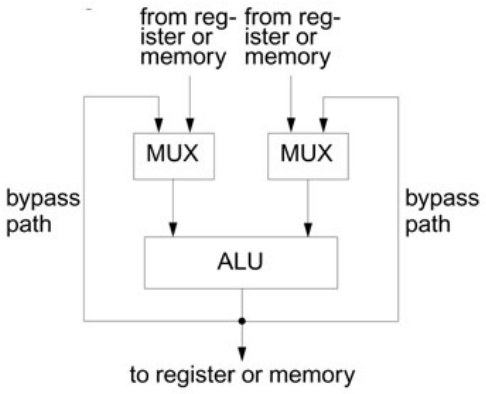
Pipeline

Dipendenza dai dati 2

Soluzioni

- Introduzione di fasi non operative (***nop***)
- Individuazione del rischio e prelievo del dato direttamente all’uscita dell’ALU (**data forwarding**) ➡
- Risoluzione a livello di compilatore **(vedremo esempi per l’architettura MIPS)**
- Riordino delle istruzioni (**pipeline scheduling**)

Pipeline
Data forwarding



senza bypass path

```
I1:   MUL   R2,R3     R2 ← R2 * R3
I2:   ADD   R1,R2     R1 ← R1 + R2
```

Clock cycle →	1	2	3	4	5	6	7	8	9	10	11	12
MUL R2,R3	FI	DI	CO	FO	EI	WO						
ADD R1,R2		FI	DI	CO	stall	stall	FO	EI	WO			
Instr. i+2			FI	DI			CO	FO	EI	WO		

con bypass path

Clock cycle →	1	2	3	4	5	6	7	8	9	10	11	12
MUL R2,R3	FI	DI	CO	FO	EI	WO						
ADD R1,R2		FI	DI	CO	stall	FO	EI	WO				

Pipeline
Dipendenza dai controlli

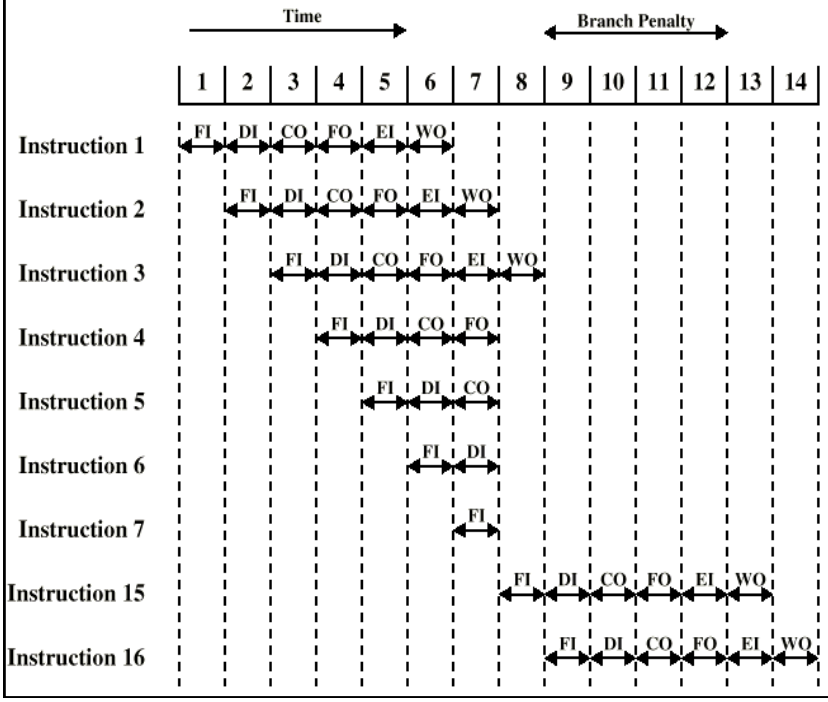
- Tutte le istruzioni che modificano il PC (salti condizionati e non, chiamate a e ritorni da procedure, interruzioni) invalidano la pipeline
- La fase **fetch** successiva carica l'istruzione seguente, che può *non essere* quella giusta
- Tali istruzioni sono circa il 30% del totale medio di un programma

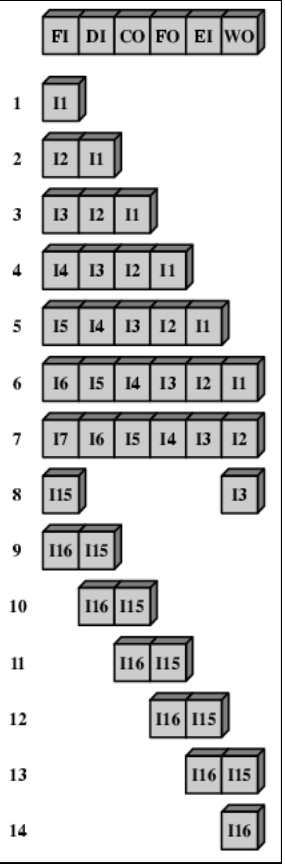
Pipeline
Dipendenza dai controlli

Soluzioni

- Mettere in **stallo** la pipeline fino a quando non si è calcolato l'indirizzo della prossima istruzione
  - Pessima efficienza, massima semplicità
- Individuare le istruzioni critiche per anticiparne l'esecuzione, eventualmente mediante apposita logica di controllo
  - Compilazione complessa, hardware specifico

Salto *(Stallo)*



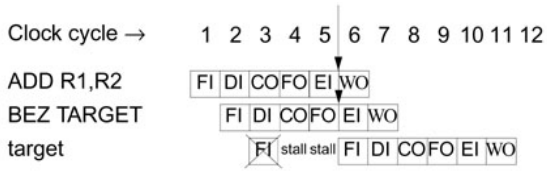


Salto *(Stallo)*

Salto incondizionato

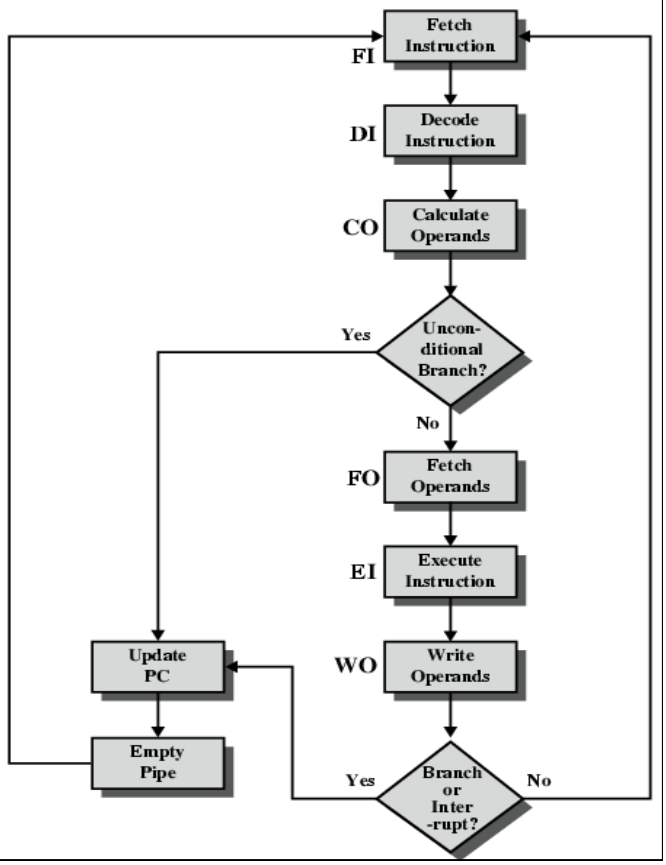
Salto condizionato

Salto preso



Salto non preso

Funzionamento pipeline a 6 stadi con trattamento dei salti ed interrupt tramite svuotamento



Pipeline
Dipendenza dai controlli

Alcune soluzioni per salti condizionati

- flussi multipli (multiple streams) ➡
- prelievo anticipato della destinazione (prefetch branch target) ➡
- buffer circolare (loop buffer) ➡
- predizione del salto (branch prediction) ➡
- salto ritardato (delayed branch) ➡

Pipeline
Dipendenza dai controlli

Flussi multipli: replicare le parti iniziali della pipeline, una che contenga l'istruzione successiva a quella corrente di salto (nel caso il salto non avvenga), e l'altra l'istruzione destinazione (*target*) del salto (nel caso in cui il salto avvenga)

Problemi di questa soluzione:

- conflitti nell'accesso alle risorse (registri, memoria, ALU,…) da parte delle 2 pipeline
- presenza di salti condizionali in sequenza che entrano nelle 2 pipeline prima che si sia risolta la condizione del primo salto condizionale (occorrerebbero 2 pipeline aggiuntive per ogni ulteriore salto condizionale…)

Pipeline
Dipendenza dai controlli

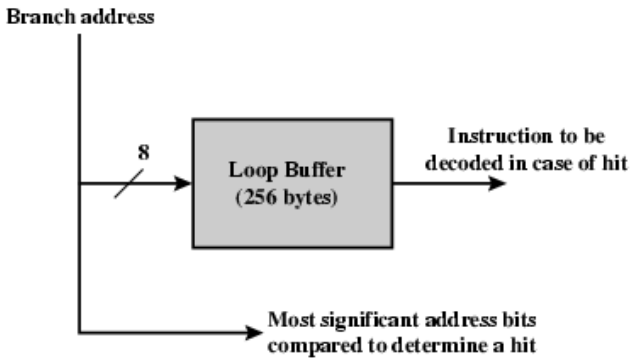
Prelievo anticipato della destinazione: quando si incontra un salto condizionato si effettua il fetch anticipato della istruzione di destinazione del salto in modo da trovarla già caricata nel caso in cui il salto debba avvenire.

Problemi di questa soluzione:

- non evita l'eventuale svuotamento della pipeline con conseguente perdita di prestazioni

Pipeline
Dipendenza dai controlli

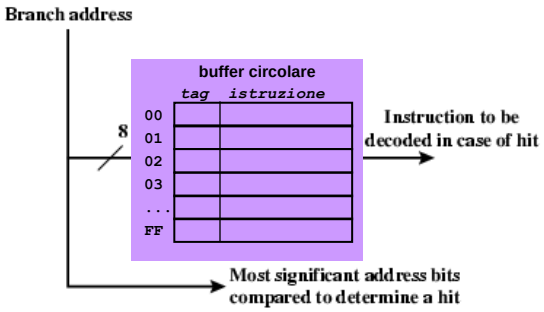
Buffer circolare: si utilizza una memoria piccola e molto veloce (il buffer circolare) dove mantenere le ultime *n* istruzioni prelevate. In caso di salto, si controlla se l'istruzione destinazione è già presente nel buffer, così da evitare il fetch della stessa.



Buffer circolare (senza prefetch)

Memoria

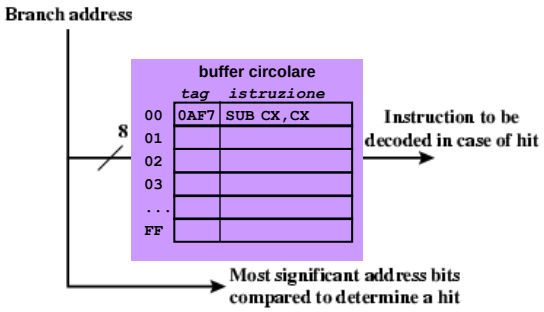
```
        ...
0AF700  SUB CX,CX
0AF701  MOV AX,01AF90
0AF702  CMP AX,000543
0AF703  BNZ 0AF701
        ...
0AF7FF  ADD AX,CX
0AF800  SUB BX,CX
0AF801  CMP BX,AX
0AF802  BNZ 0AF800
0AF803  INC BX
        ...
```



Buffer circolare (senza prefetch)

Memoria

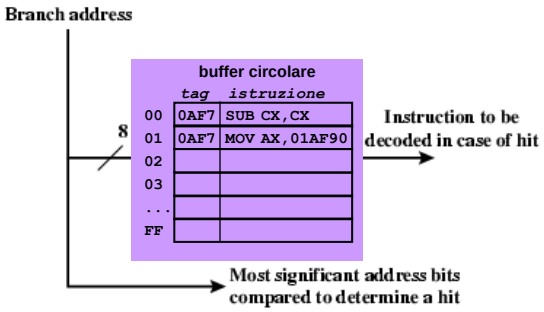
```
        ...
0AF700  SUB CX,CX
0AF701  MOV AX,01AF90
0AF702  CMP AX,000543
0AF703  BNZ 0AF701
        ...
0AF7FF  ADD AX,CX
0AF800  SUB BX,CX
0AF801  CMP BX,AX
0AF802  BNZ 0AF800
0AF803  INC BX
        ...
```



Buffer circolare (senza prefetch)

Memoria

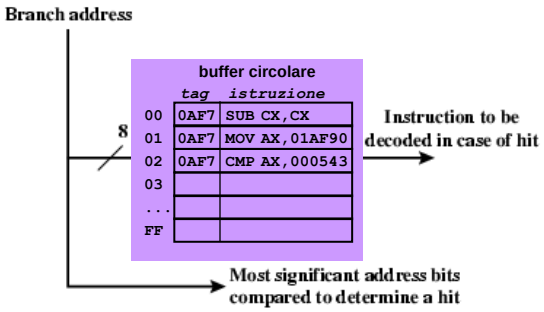
```
        ...
0AF700  SUB CX,CX
0AF701  MOV AX,01AF90
0AF702  CMP AX,000543
0AF703  BNZ 0AF701
        ...
0AF7FF  ADD AX,CX
0AF800  SUB BX,CX
0AF801  CMP BX,AX
0AF802  BNZ 0AF800
0AF803  INC BX
        ...
```



Buffer circolare (senza prefetch)

Memoria

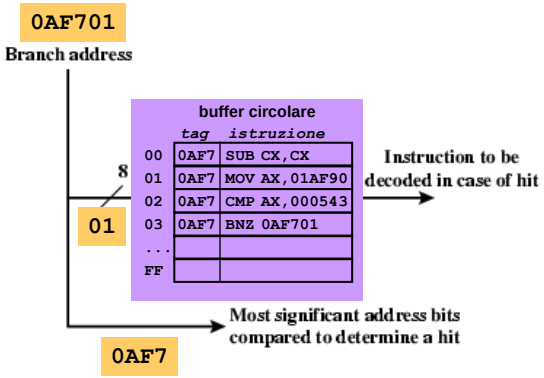
```
        ...
0AF700  SUB CX,CX
0AF701  MOV AX,01AF90
0AF702  CMP AX,000543
0AF703  BNZ 0AF701
        ...
0AF7FF  ADD AX,CX
0AF800  SUB BX,CX
0AF801  CMP BX,AX
0AF802  BNZ 0AF800
0AF803  INC BX
        ...
```



Buffer circolare (senza prefetch)

Memoria

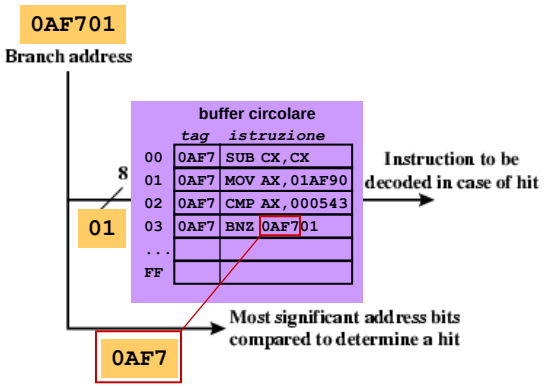
```
          ...
0AF700    SUB  CX,CX
0AF701    MOV  AX,01AF90
0AF702    CMP  AX,000543
0AF703    BNZ  0AF701
          ...
0AF7FF    ADD  AX,CX
0AF800    SUB  BX,CX
0AF801    CMP  BX,AX
0AF802    BNZ  0AF800
0AF803    INC  BX
          ...
```



Buffer circolare (senza prefetch)

Memoria

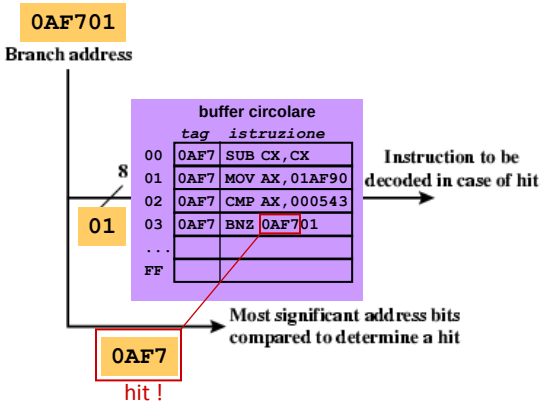
```
          ...
0AF700    SUB  CX,CX
0AF701    MOV  AX,01AF90
0AF702    CMP  AX,000543
0AF703    BNZ  0AF701
          ...
0AF7FF    ADD  AX,CX
0AF800    SUB  BX,CX
0AF801    CMP  BX,AX
0AF802    BNZ  0AF800
0AF803    INC  BX
          ...
```



Buffer circolare (senza prefetch)

Memoria

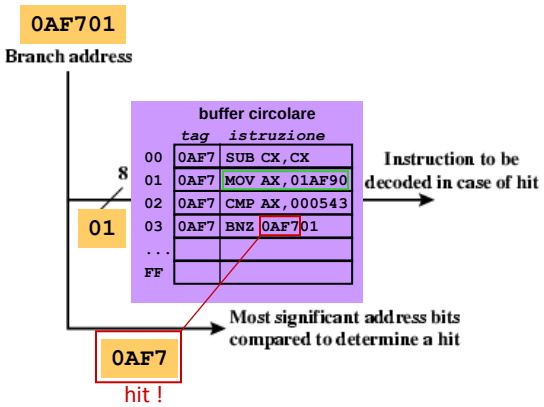
```
          ...
0AF700    SUB  CX,CX
0AF701    MOV  AX,01AF90
0AF702    CMP  AX,000543
0AF703    BNZ  0AF701
          ...
0AF7FF    ADD  AX,CX
0AF800    SUB  BX,CX
0AF801    CMP  BX,AX
0AF802    BNZ  0AF800
0AF803    INC  BX
          ...
```



Buffer circolare (senza prefetch)

Memoria

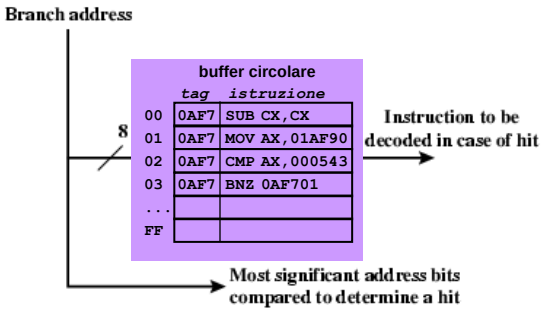
```
          ...
0AF700    SUB  CX,CX
0AF701    MOV  AX,01AF90
0AF702    CMP  AX,000543
0AF703    BNZ  0AF701
          ...
0AF7FF    ADD  AX,CX
0AF800    SUB  BX,CX
0AF801    CMP  BX,AX
0AF802    BNZ  0AF800
0AF803    INC  BX
          ...
```



Buffer circolare (senza prefetch)

Memoria

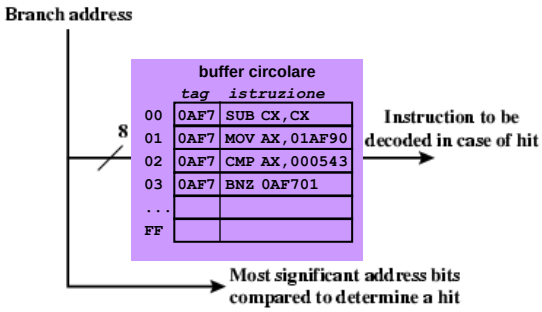
```
        ...
0AF700  SUB CX,CX
0AF701  MOV AX,01AF90
0AF702  CMP AX,000543
0AF703  BNZ 0AF701
        ...
0AF7FF  ADD AX,CX
0AF800  SUB BX,CX
0AF801  CMP BX,AX
0AF802  BNZ 0AF800
0AF803  INC BX
        ...
```



Buffer circolare (senza prefetch)

Memoria

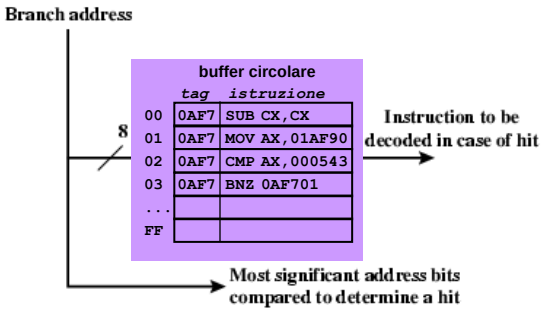
```
        ...
0AF700  SUB CX,CX
0AF701  MOV AX,01AF90
0AF702  CMP AX,000543
0AF703  BNZ 0AF701
        ...
0AF7FF  ADD AX,CX
0AF800  SUB BX,CX
0AF801  CMP BX,AX
0AF802  BNZ 0AF800
0AF803  INC BX
        ...
```



Buffer circolare (senza prefetch)

Memoria

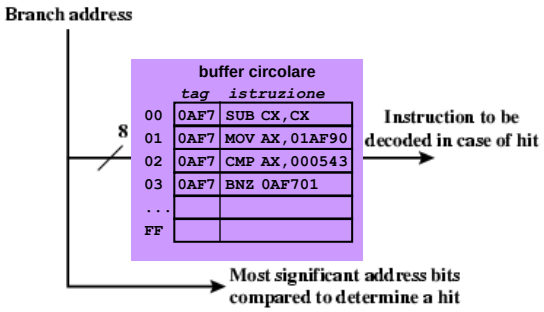
```
          ...
0AF700    SUB CX,CX
0AF701    MOV AX,01AF90
0AF702    CMP AX,000543
0AF703    BNZ 0AF701
          ...
0AF7FF    ADD AX,CX
0AF800    SUB BX,CX
0AF801    CMP BX,AX
0AF802    BNZ 0AF800
0AF803    INC BX
          ...
```



Buffer circolare (senza prefetch)

Memoria

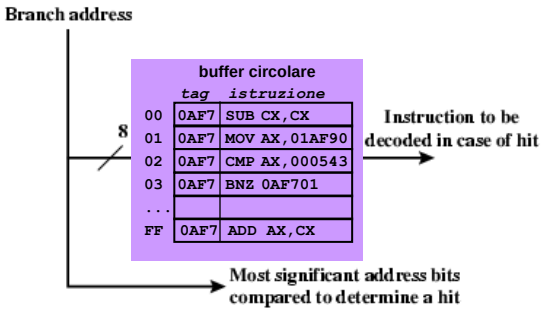
```
          ...
0AF700    SUB CX,CX
0AF701    MOV AX,01AF90
0AF702    CMP AX,000543
0AF703    BNZ 0AF701
          ...
0AF7FF    ADD AX,CX
0AF800    SUB BX,CX
0AF801    CMP BX,AX
0AF802    BNZ 0AF800
0AF803    INC BX
          ...
```



Buffer circolare (senza prefetch)

Memoria

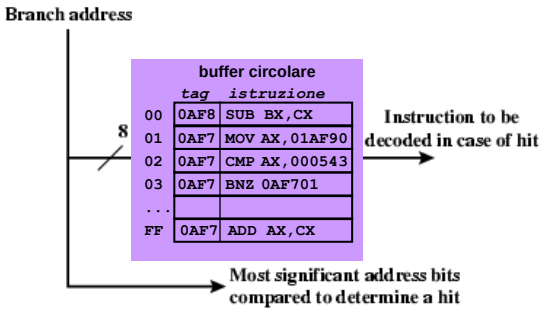
```
        ...
0AF700  SUB CX,CX
0AF701  MOV AX,01AF90
0AF702  CMP AX,000543
0AF703  BNZ 0AF701
        ...
0AF7FF  ADD AX,CX
0AF800  SUB BX,CX
0AF801  CMP BX,AX
0AF802  BNZ 0AF800
0AF803  INC BX
        ...
```



Buffer circolare (senza prefetch)

Memoria

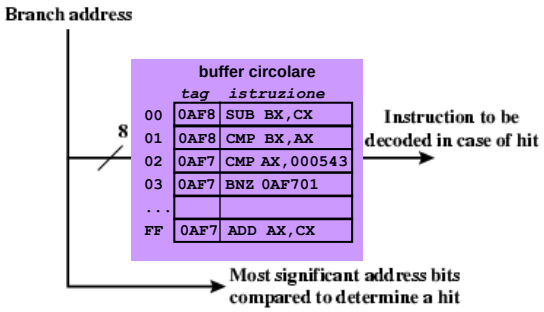
```
        ...
0AF700  SUB CX,CX
0AF701  MOV AX,01AF90
0AF702  CMP AX,000543
0AF703  BNZ 0AF701
        ...
0AF7FF  ADD AX,CX
0AF800  SUB BX,CX
0AF801  CMP BX,AX
0AF802  BNZ 0AF800
0AF803  INC BX
        ...
```



Buffer circolare (senza prefetch)

Memoria

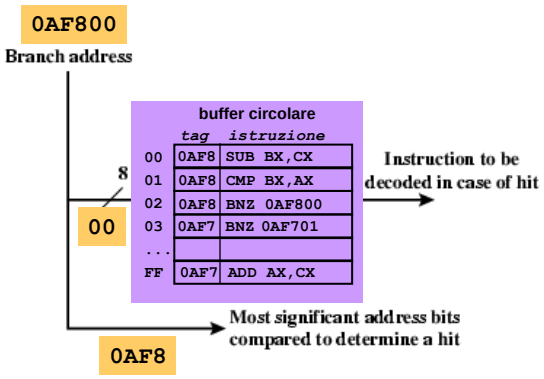
```
          ...
0AF700  SUB CX,CX
0AF701  MOV AX,01AF90
0AF702  CMP AX,000543
0AF703  BNZ 0AF701
          ...
0AF7FF  ADD AX,CX
0AF800  SUB BX,CX
0AF801  CMP BX,AX
0AF802  BNZ 0AF800
0AF803  INC BX
          ...
```



Buffer circolare (senza prefetch)

Memoria

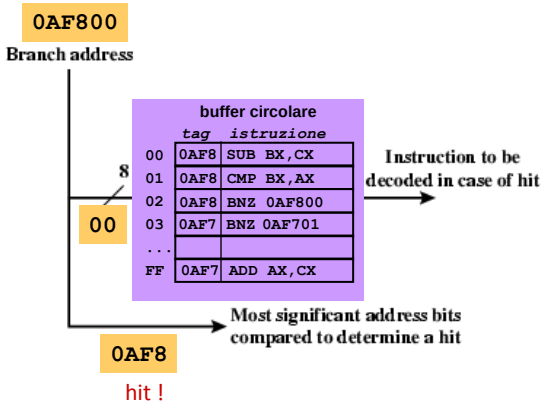
```
          ...
0AF700  SUB CX,CX
0AF701  MOV AX,01AF90
0AF702  CMP AX,000543
0AF703  BNZ 0AF701
          ...
0AF7FF  ADD AX,CX
0AF800  SUB BX,CX
0AF801  CMP BX,AX
0AF802  BNZ 0AF800
0AF803  INC BX
          ...
```



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
         ...
```



Pipeline

Dipendenza dai controlli

Buffer circolare: si utilizza una memoria piccola e molto veloce (il buffer circolare) dove mantenere le ultime *n* istruzioni prelevate. In caso di salto, si controlla se l'istruzione destinazione è già presente nel buffer, così da evitare il fetch della stessa.

Vantaggi:

- anticipando il fetch, alcune delle istruzioni successive a quella corrente saranno già presenti nel buffer e se non si ha salto non ci sarà bisogno di caricarle dalla memoria
- se si salta in avanti di poche istruzioni (vedi trattamento del costrutto IF-THEN-ELSE), l'istruzione destinazione sarà già presente nel buffer
- se il salto condizionale realizza un ciclo le cui istruzioni possono essere tutte contenute nel buffer, non c'è bisogno di effettuare fetch ripetuti delle stesse istruzioni

Pipeline
Dipendenza dai controlli

Predizione dei salti: si cerca di prevedere se il salto sarà intrapreso oppure no.

Varie possibilità:

- previsione di saltare sempre
- previsone di non saltare mai
- previsione in base al codice operativo

} *approcci statici*

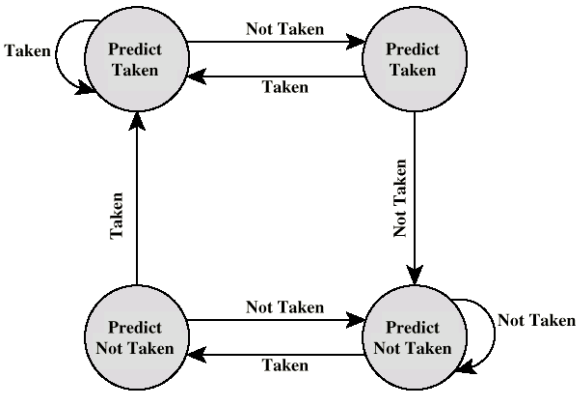
- bit *taken/not taken*
- tabella della storia dei salti

} *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.

bit *taken/not taken* :

- ad ogni istruzione di salto condizionato si associano uno o più bit che codificano la storia recente.
- bit memorizzati non in memoria centrale ma in una locazione temporanea ad accesso molto veloce



esempio con 2 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
```

- **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

Predizione dinamica 1 bit

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

1 saltare

errori di predizione 0

Predizione dinamica 1 bit

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

1 saltare

errori di predizione 0

Predizione dinamica 1 bit

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

1 saltare

errori di predizione 0

Predizione dinamica 1 bit

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

1 saltare

errori di predizione

0

Predizione dinamica 1 bit

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

1 saltare

errori di predizione

0

Predizione dinamica 1 bit

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

1 saltare

errori di predizione

0

Predizione dinamica 1 bit

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

1 saltare

errori di predizione

0

Predizione dinamica 1 bit

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

1 saltare

errori di predizione 0

Predizione dinamica 1 bit

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

0 non saltare

errori di predizione 1

Predizione dinamica 1 bit

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

0 non saltare

errori di predizione 1

Predizione dinamica 1 bit

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

0 non saltare

errori di predizione 1

Predizione dinamica 1 bit

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

0 non saltare

errori di predizione 1

Predizione dinamica 1 bit

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

0 non saltare

errori di predizione 1

Predizione dinamica 1 bit

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

0 non saltare

errori di predizione 1

Predizione dinamica 1 bit

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

0 non saltare

errori di predizione 1

Predizione dinamica 1 bit

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

0 non saltare

errori di predizione 1

Predizione dinamica 1 bit

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

0 non saltare

errori di predizione 1

Predizione dinamica 1 bit

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

1 saltare

errori di predizione 2

Predizione dinamica 1 bit

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

1 saltare

errori di predizione 2

Predizione dinamica 1 bit

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

1 saltare

errori di predizione 2

Predizione dinamica 1 bit

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

0 non saltare

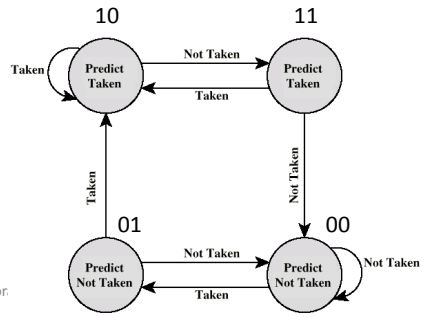
errori di predizione 3

Predizione dinamica 2 bit

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

errori di predizione 0

10 saltare

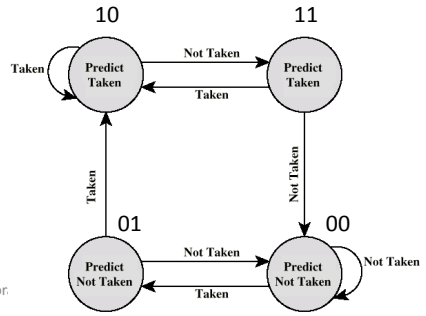


Predizione dinamica 2 bit

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

errori di predizione 0

10 saltare

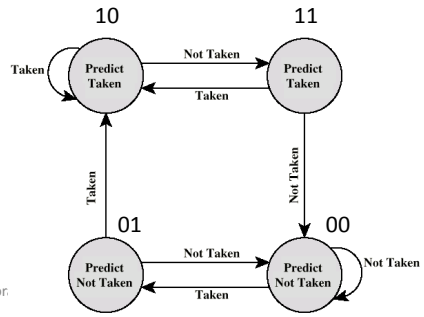


Predizione dinamica 2 bit

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

errori di predizione 0

10 saltare

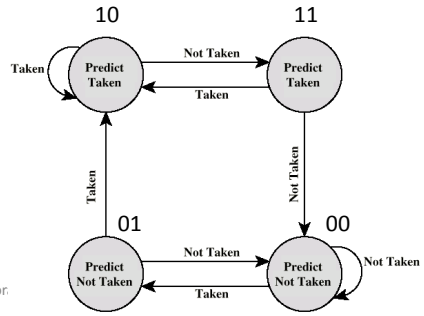


Predizione dinamica 2 bit

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

errori di predizione 0

10 saltare

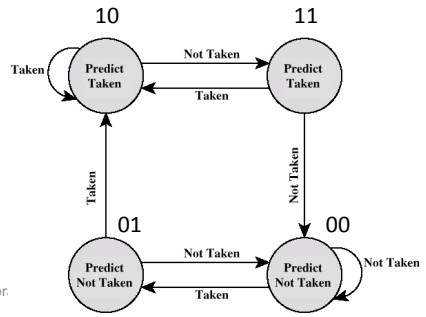


Predizione dinamica 2 bit

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

errori di predizione 0

10 saltare

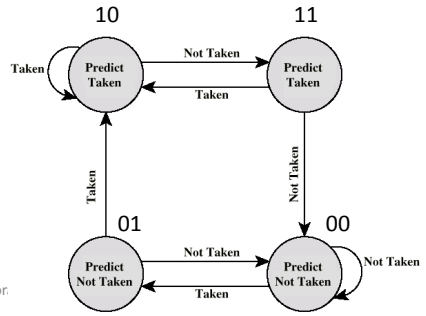


Predizione dinamica 2 bit

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

errori di predizione 0

10 saltare

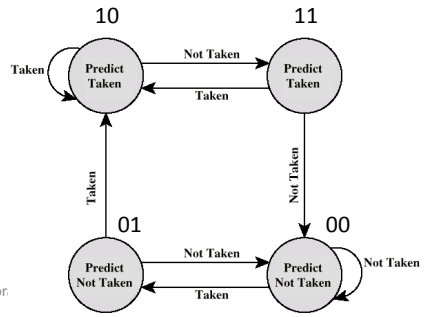


Predizione dinamica 2 bit

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

errori di predizione 0

10 saltare

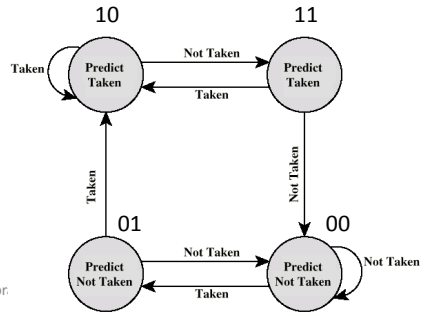


Predizione dinamica 2 bit

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

errori di predizione 0

10 saltare

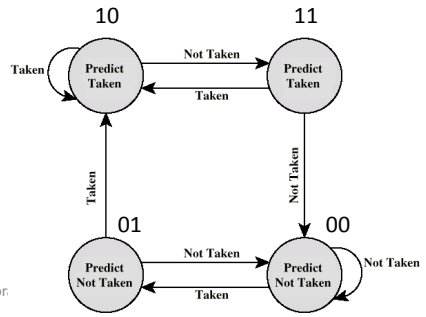


Predizione dinamica 2 bit

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

errori di predizione 1

11 saltare

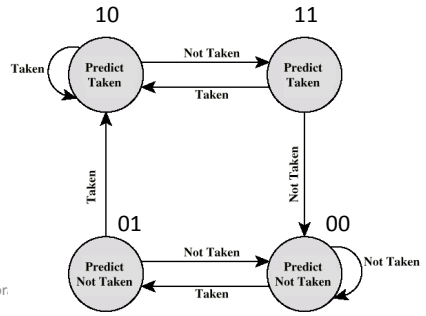


Predizione dinamica 2 bit

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

errori di predizione 1

11 saltare

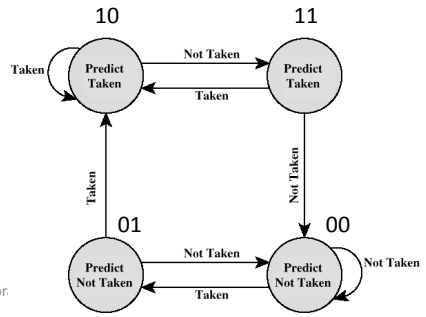


Predizione dinamica 2 bit

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

errori di predizione 1

11 saltare

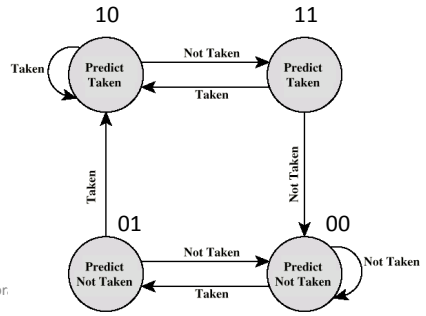


Predizione dinamica 2 bit

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

errori di predizione 1

11 saltare

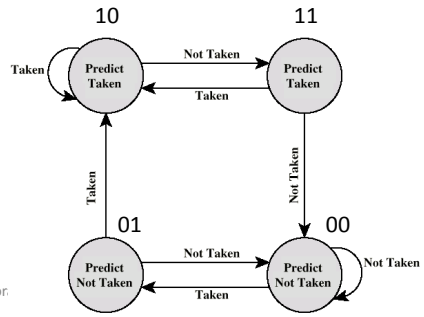


Predizione dinamica 2 bit

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

errori di predizione 1

11 saltare

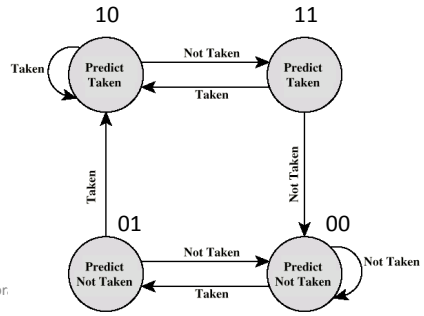


Predizione dinamica 2 bit

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

errori di predizione 1

11 saltare

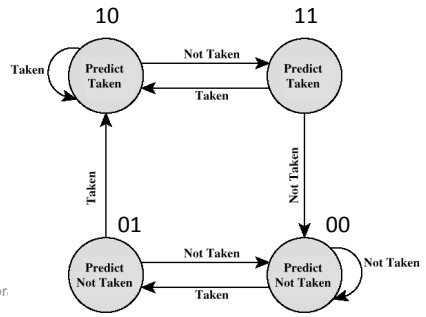


Predizione dinamica 2 bit

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

errori di predizione 1

11 saltare

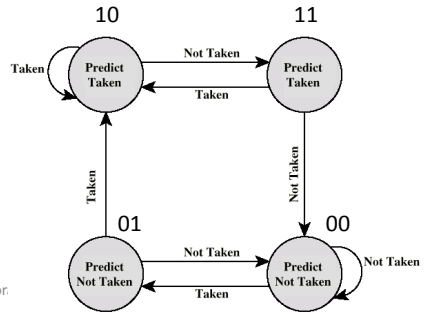


Predizione dinamica 2 bit

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

errori di predizione 1

11 saltare

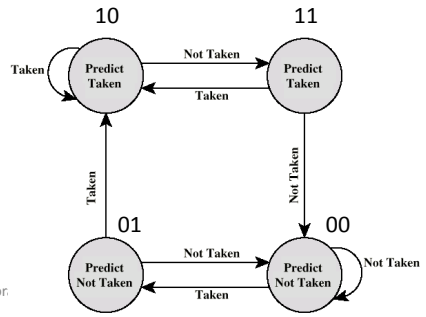


Predizione dinamica 2 bit

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

errori di predizione 1

11 saltare

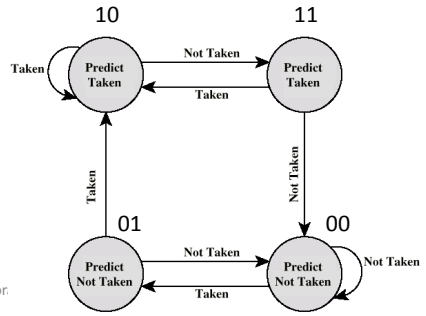


Predizione dinamica 2 bit

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

errori di predizione 1

10 saltare

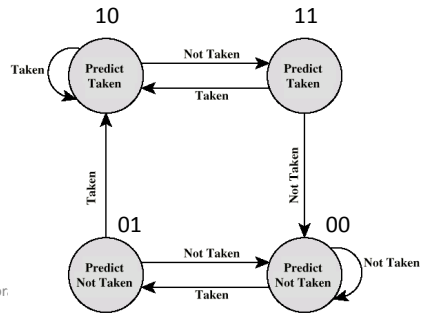


Predizione dinamica 2 bit

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

errori di predizione 1

10 saltare

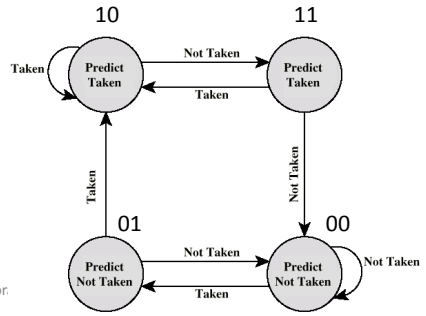


Predizione dinamica 2 bit

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

errori di predizione 1

10 saltare

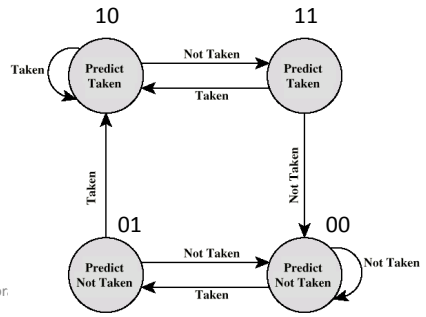


Predizione dinamica 2 bit

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

errori di predizione 2

11 saltare

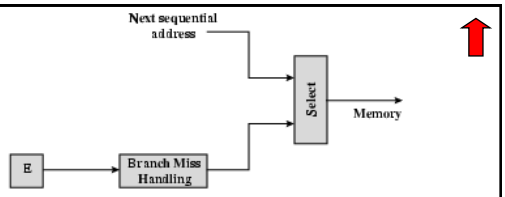


Problemi di bit *taken/not taken* :

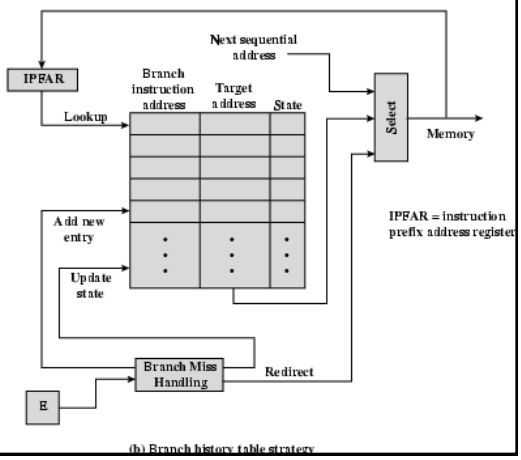
- quando si decide di saltare, bisogna aspettare la decodifica dell'indirizzo destinazione prima di poter prelevare l'istruzione destinazione
- si può anticipare il prelievo a patto di salvare opportune info nel *branch target buffer* o *branch history table*

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



(a) Predict never taken strategy



(b) Branch history table strategy

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
MOVE #10,R1   R1 ← 10
------------
TAR ------------
```

istruzione eseguita solo se non si prende il salto →

codice ottimizzato dal compilatore

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

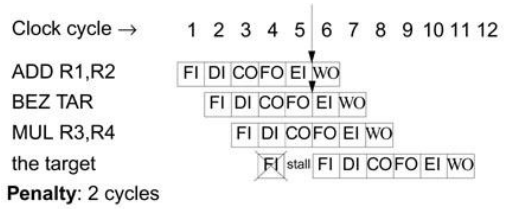
istruzione eseguita in ogni caso: si trova nel *branch delay slot* !!

istruzione eseguita solo se non si prende il salto

Salto ritardato (delayed branch)

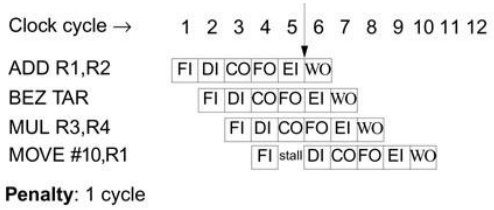
Salto preso

condizione ed indirizzo di salto conosciuti qui

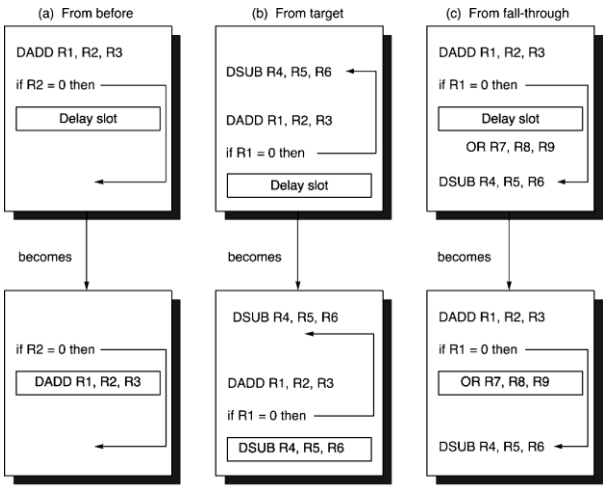


Salto non preso

condizione di salto conosciuta qui



Salto ritardato (delayed branch)

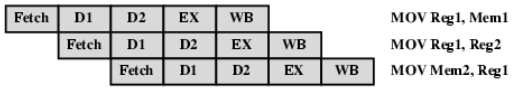


b) e c) legali solo se R4 e R7 sono registri temporanei il cui contenuto può essere “sporcato” senza cambiare la semantica del programma

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
  - 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
  - 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