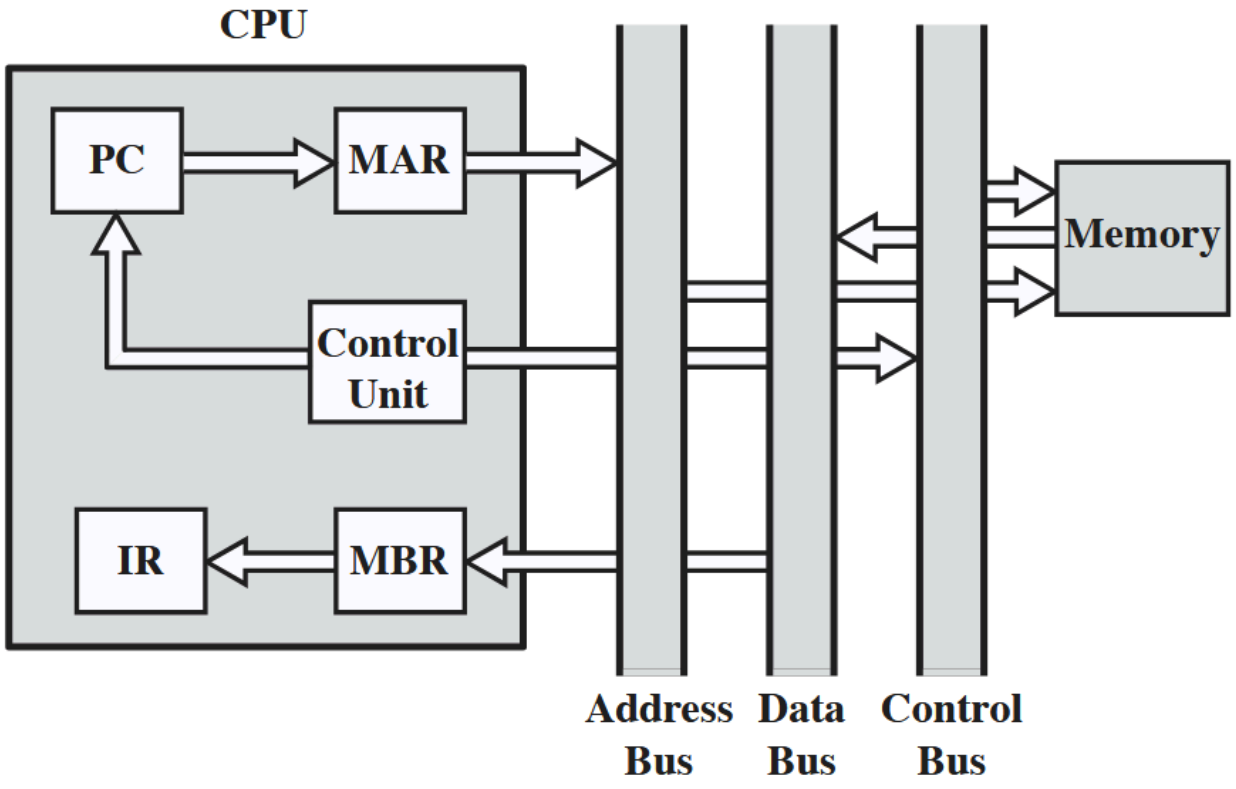
06 - PIPELINE

FASI DI ESECUZIONE DELLE ISTRUZIONI

Sappiamo che il *ciclo di esecuzione* di un'istruzione è composto da 2 fasi principali: **fetch** ed **execute**.

FETCH

1. Il *valore* di **PC** (prossima istruzione) viene copiato in **MAR**.
2. L'indirizzo viene emesso sul **bus indirizzi**.
3. La **control unit** richiede una *lettura in memoria*.
4. Il risultato della lettura viene inviato al **bus dati** e copiato in **MBR** che lo salva in **IR**.
5. Contemporaneamente, **PC** viene *incrementato*.



EXECUTE

Tale fase *dipende dal tipo di istruzione* e può includere:

- *Lettura/scrittura* in memoria.
- *Input/Output*.
- Trasferimento dati fra *registri*.
- Operazioni svolte dalla *ALU*.

PREFECTCH

I **circuiti** logici dedicati alle fasi di *fetch* ed *execute* sono **distinti**, per cui non è necessario attendere il termine dell'esecuzione dell'istruzione precedente per "cominciare" la successiva:
si può *prelevare* l'istruzione *successiva* **durante** l'*esecuzione* dell'*istruzione corrente*.
Tale operazione si chiama **INSTRUCTION PREFETCH**.

LIMITI DELL'APPROCCIO A DUE FASI

Il *prefetch* **NON** raddoppia le prestazioni, principalmente a causa di due limiti:

1. I *branch condizionati* possono rendere **vano** il prefetch: se la prossima istruzione dipende dal risultato di quella corrente, non possiamo sapere quale prelevare.
2. La fase di *fetch* è tipicamente **più breve** dell'*esecuzione*.

PIPELINE

Possiamo aggiungere *più fasi*, scomponendo execute in fasi di *durata simile*, per migliorare le prestazioni.
Questo approccio è detto **PIPELINE**.

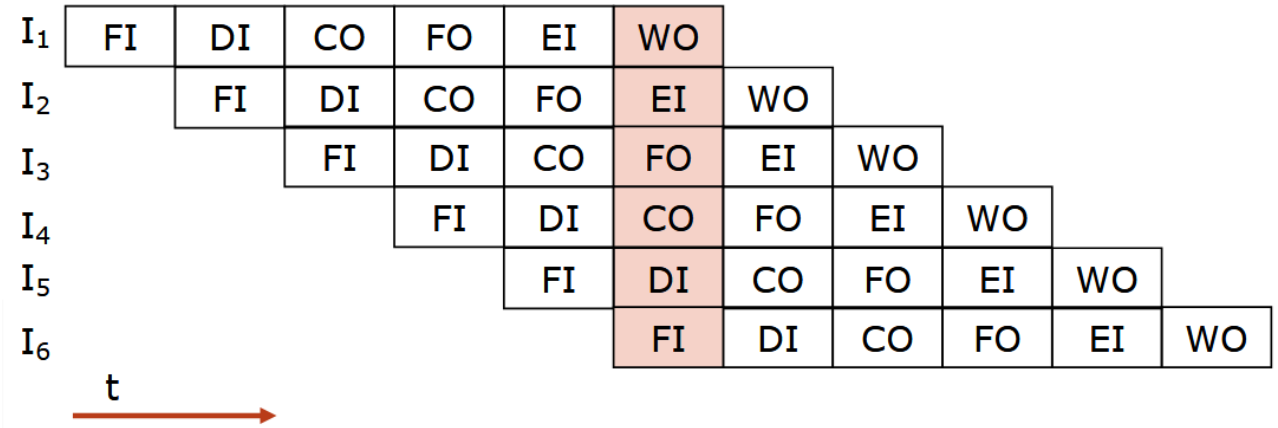
APPROCCIO A 6 FASI

Possiamo scomporre la parte di *execute* in *5 fasi* di *durata simile*, per un totale di **6 FASI**:

1. **FETCH INSTRUCTION (FI)**
   Legge la prossima istruzione.

2. **DECODE INSTRUCTION (DI)**
   Determina l'opcode e il tipo di operandi.
3. **CALCULATE OPERANDS (CO)**
   Gestisce il tipo di indirizzamento e calcola l'effettivo indirizzo degli operandi.
4. **FETCH OPERANDS (FO)**
   Recupera gli operandi da memoria se necessario.
5. **EXECUTE INSTRUCTION (EI)**
   Esecuzione effettiva dell'istruzione.
6. **WRITE OPERAND (WO)**
   Scrive il risultato in memoria o in un registro.

Anche in questo caso, se le fasi sono eseguite da *circuiti logici distinti*, possono essere tutte attive contemporaneamente (per istruzioni diverse).



Il tempo necessario ad eseguire la *prima istruzione* è detto **TRANSITORIO**: la *pipeline* si sta *riempiendo*.
Notiamo che **dopo** il *transitorio* viene *completata* un'istruzione a *ogni stadio*.

Possiamo anche introdurre *nuovi registri* per memorizzare i *risultati parziali* di ogni stadio.

TEMPO DI ESECUZIONE E SPEEDUP

Possiamo calcolare il tempo di esecuzione di N istruzioni con e senza implementazione della pipeline per comprenderne il vantaggio.

Chiamiamo t il tempo di esecuzione (medio) di una fase e k il numero di fasi in cui le operazioni sono scomposte:

- **SENZA** pipeline: $t_1 = N \cdot k \cdot t$
- **CON** pipeline a k fasi: $t_k = (k + (N - 1)) \cdot t$

Ricordando che:

- Transitorio: primi k cicli con pipeline in riempimento.
- Regime: un'istruzione completata a ogni fase (caso ideale).

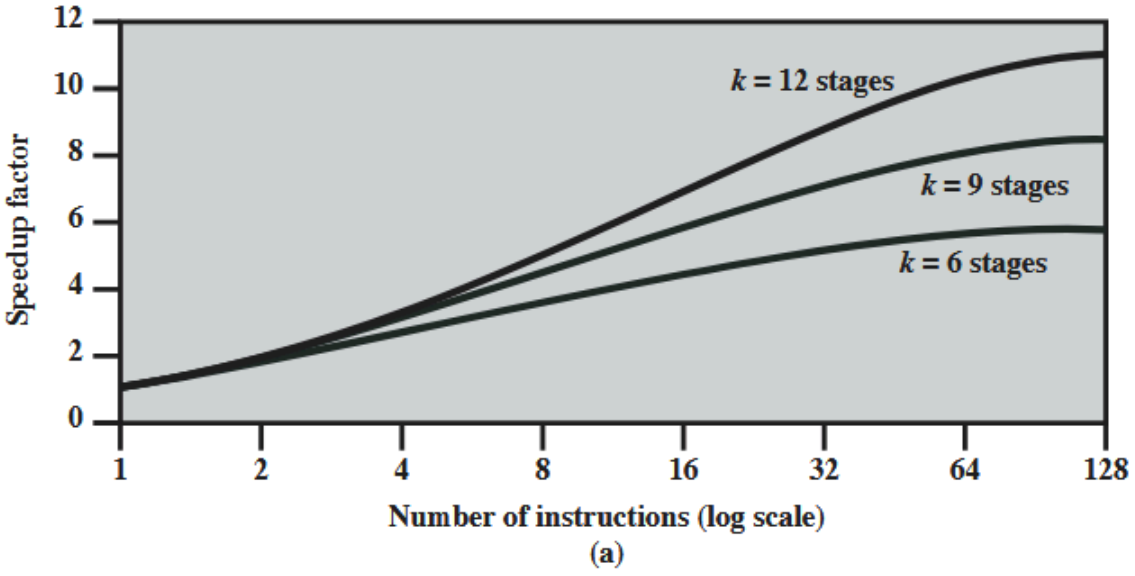
FATTORE DI SPEEDUP

Indica *quanto più veloce* l'uso di una pipeline rende l'esecuzione di un programma rispetto al caso in cui la pipeline è assente.

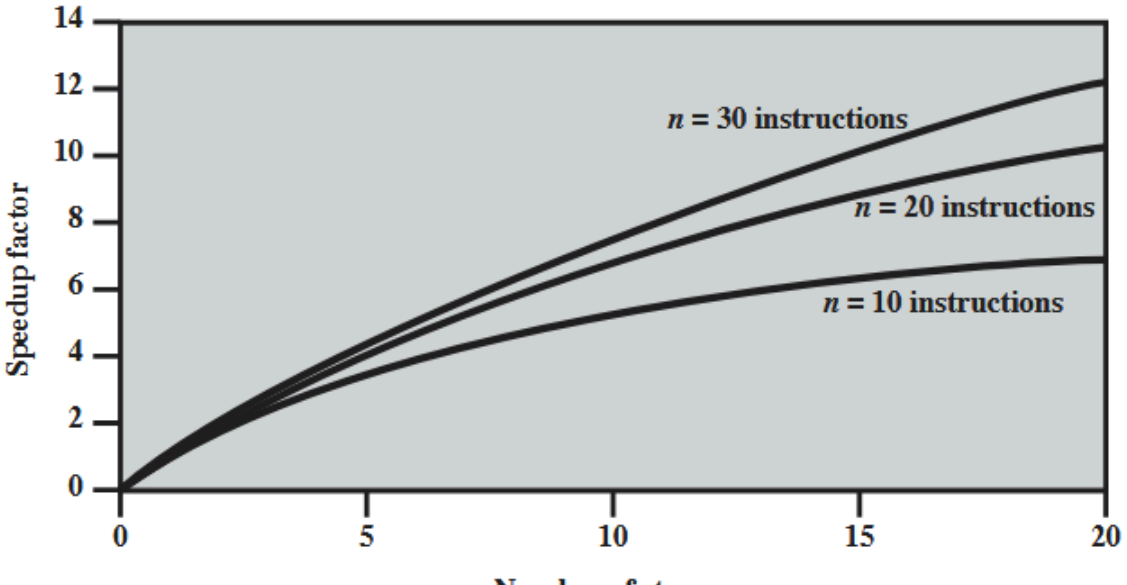
$$S_k = \frac{T_1}{T_k} = \frac{N \cdot k \cdot t}{k \cdot t + (N-1) \cdot t} = \frac{N \cdot k}{k + N - 1}$$

Per un numero di istruzioni tendente a infinito abbiamo:

$$S_k = \lim_{N \to \infty} \frac{N \cdot k}{k + N - 1} = k$$



(a)



(b)

INCEPPAMENTO PIPELINE

Il *fattore di speedup* S_k è un valore **teorico**, raggiunto solo se il pipeline opera a regime in modo che a ogni tempo di ciclo *avvii* e *termini* una *nuova istruzione*.

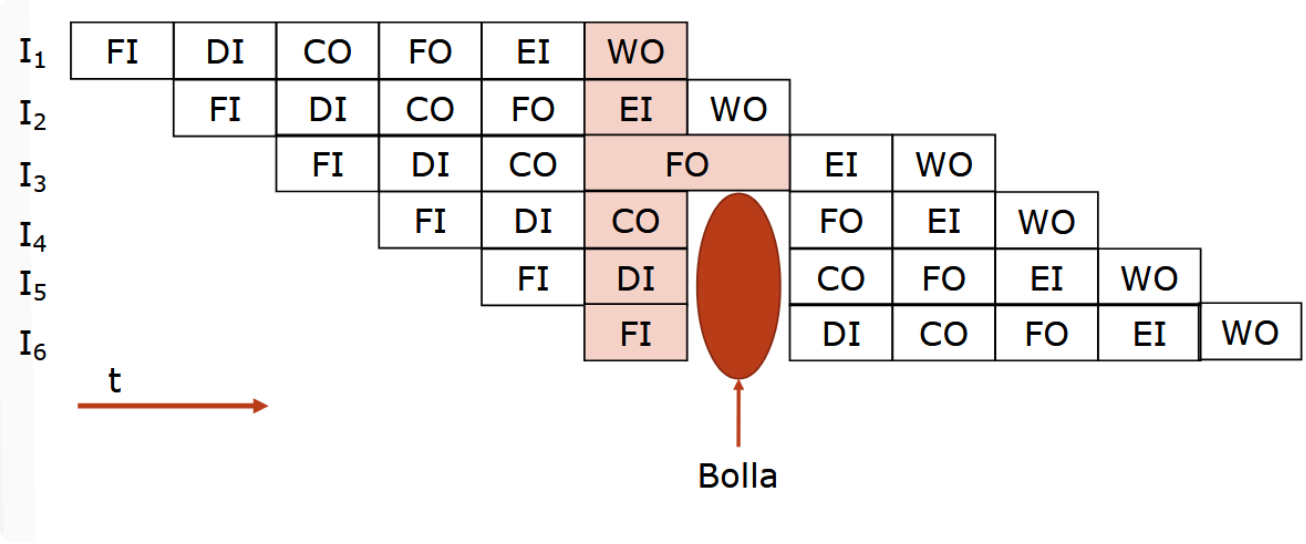
In realtà il pipeline può **incepparsi** (*pipeline stall/bubble*) per problemi dovuti a:

- Accessi alla memoria (*cache miss*).
- Conflitti sulle risorse (*resource hazard*).
- Dipendenze tra dati (*data hazard*).
- Salti condizionati (*branch hazard*).

Chiaramente, gli stalli *riducono* il fattore di speedup.

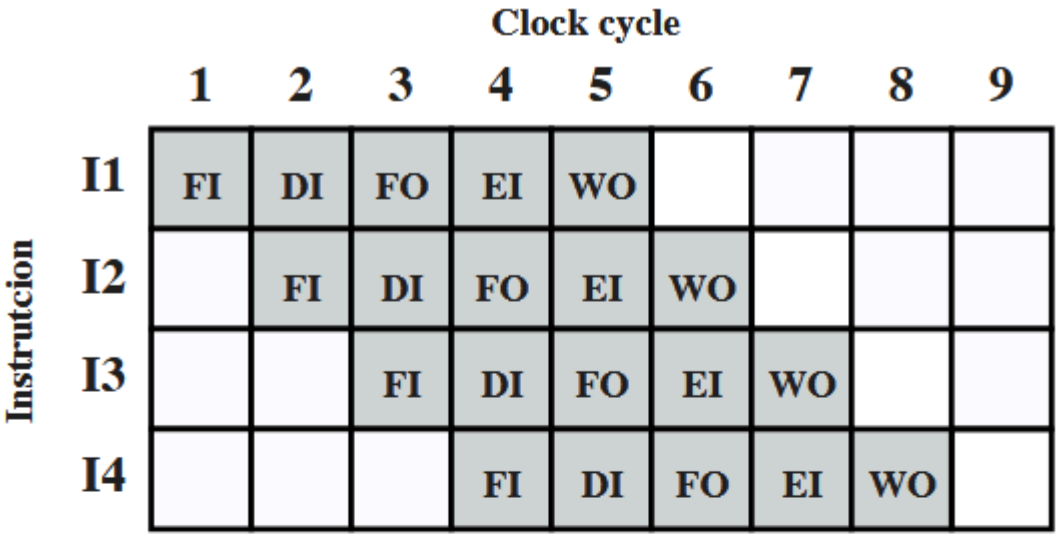
CACHE MISS

Le fasi che *accedono alla memoria* hanno una *durata pari alle altre* solo se gli accessi si *risolvono in cache hit*. In caso di *cache miss* l'operazione può richiedere 2 o 3 cicli, ritardando l'esecuzione delle istruzioni.

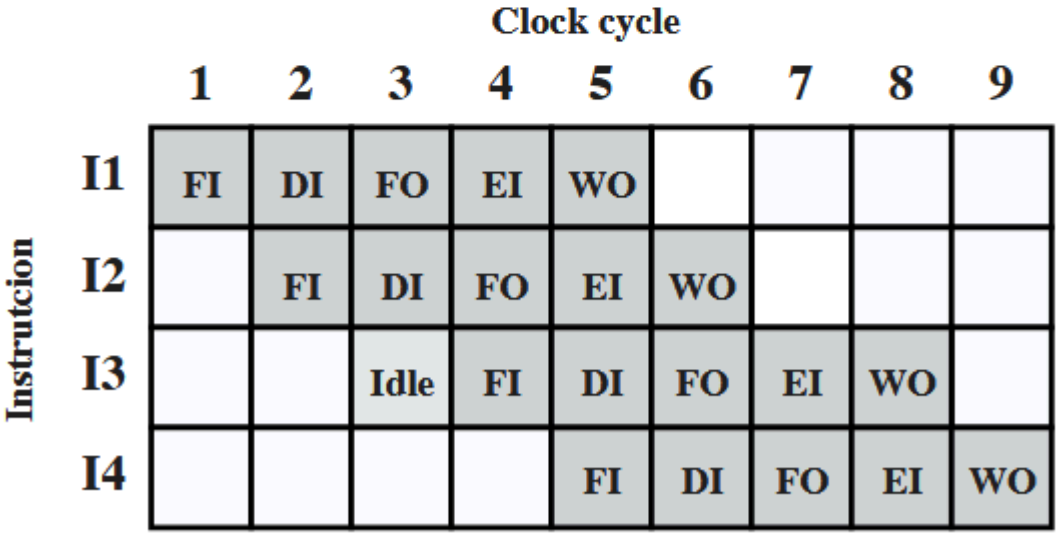


RESOURCE HAZARD

Si verifica quando *due fasi* richiedono la *stessa risorsa* (per esempio richiedono contemporaneamente di accedere alla memoria).



(a) Five-stage pipeline, ideal case



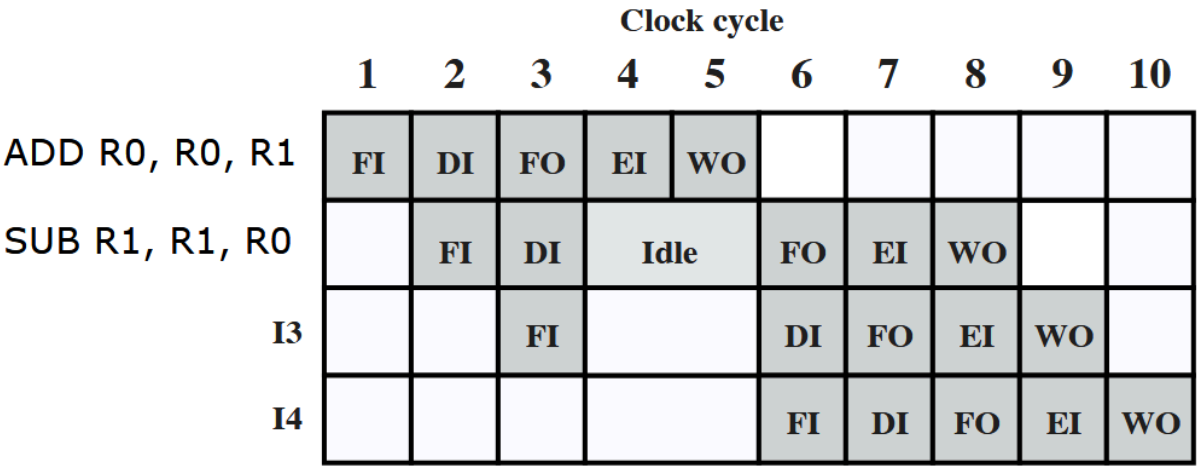
(b) I1 source operand in memory

Qui nell'esempio al terzo ciclo sia I1 che I3 dovrebbero accedere in memoria.

DATA HAZARD

Si verifica quando gli *operandi* di un'istruzione sono i *risultati* di una precedente *istruzione non ancora completata*.

L'esecuzione non può proseguire finchè il risultato non è pronto e si ha uno stallo:



I *data hazard* possono essere *evitati* dal *compilatore* con un **riordino delle istruzioni**. Anche quando non eliminabili, le conseguenze negative possono comunque essere ridotte dal processore con la tecnica del **data forwarding** (*bypass*): i risultati prodotti dall'ALU sono inoltrati allo stadio successivo del pipeline in contemporanea alla loro memorizzazione.

CONTROL HAZARD

Detto anche *branch hazard*, si verifica nelle istruzioni di salto condizionato quando (prima di verificare se la condizione di salto è verificata), il pipeline viene *alimentato* con l'*istruzione* della *diramazione* che **NON** sarà *intrapresa*.

In tal caso è necessario:

- *Svuotare* il pipeline.
- *Annullare* gli *effetti* delle istruzioni relative al branch non effettuato.
- *Rialimentare* il pipeline con le istruzioni della diramazione corretta.

Esistono tuttavia dei rimedi contro questa tipologia di hazard:

1. **MULTIPLE STREAMS**
   Se sono disponibili almeno *due pipeline distinte*, si caricano entrambi i rami e quando il risultato è noto si scarta il ramo sbagliato.
2. **DELAYED BRANCH**
   Prevede che il processore *esegua comunque* un'*altra istruzione prima del branch*.
   Il compilatore può riordinare le istruzioni in modo da collocarne una (da eseguire in ogni caso) dopo ogni istruzione di salto. Se una tale istruzione non viene trovata si inserisce una **NOP**, che comporta un ritardo ma almeno evita l'avvio di istruzioni non corrette.
3. **BRANCH PREDICTION**
   Si può provare a *prevedere* quale possa essere il ramo corretto da intraprendere.
   Esistono branch prediction:

- *Statici*: la scelta *non* dipende dalla storia di esecuzione (sempre branch o mai branch, in base all'opcode).
- *Dinamici*: la scelta dipende dalla storia di esecuzione in base a *bit di stato* e ad una *branch history table*.
  La **BHT** è una *cache* contenente (su *CAM*) gli *indirizzi* delle *istruzioni di salto* e lo *stato della previsione*: il PC viene confrontato con tali indirizzi; se presente si valuta lo stato della previsione, se assente si inserisce una nuova riga nella tabella. Lo stato della previsione viene aggiornato quando la condizione viene valutata.
  NOTA: Il *Branch Target Buffer* (*BTB*) è una *BHT* in cui ogni riga include *anche l'indirizzo del branch*.