1 Lezione del 05-05-25

1.1 Architettura interna del processore

Vediamo più nel dettaglio dell'architettura interna dei processori Intel x86.

Per velocizzare l'operazione del processo la via principale potrebbe essere quello di aumentare le prestazioni dei componenti, cioè dei transistor, che lo compongono. Si ha che questo approccio però non è scalabile all'infinito, in quanto negli ultimi anni si è raggiunto un *plateu* delle prestazioni.

La soluzione che vediamo è quindi **architetturale**, e consiste nell'uso di una **pipeline** particolare per l'esecuzione delle istruzioni. Un normale ciclo di esecuzione di un'istruzione si svolge come:

Prelievo istruzione | Decodifica | Prelievo operandi | Esecuzione | Scrittura

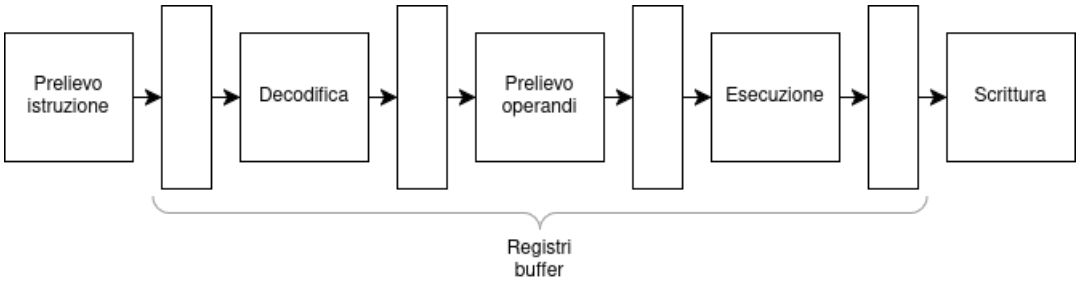
1.1.1 Pipeline

Se ognuna di queste fasi è svolta da una certa circuiteria, possiamo far passare in parallelo ogni istruzione da ogni circuiteria, cioè avere che l'istruzione gestita ad ogni istante temporale sucessivo t_0, t_1, ... è:

	Prelievo istruzione	Decodifica	Prelievo operandi	Esecuzione	Scrittura
t_0	i				
t_1	$i+1$	i			
t_2	$i+2$	$i+1$	i		
t_3	$i+3$	$i+2$	$i+1$	i	
t_4	$i+4$	$i+3$	$i+2$	$i+1$	i

Questo approccio chiaramente non modifica il tempo necessario ad eseguire una istruzione (e anzi vedremo lo aumenta un po'), ma di contro permette di aumentare la frequenza delle istruzioni eseguite, per l'esattezza di un fattore pari al numero di fasi in cui si divide l'esecuzione (qui 5).

Realizzaremo infatti questo tipo di struttura frapponendo fra ogni blocco funzionale un registro, che campiona sul rising edge del clock, rallentando leggermente la velocità della pipeline per accomodare il tempo di setup dei registri. Inoltre, il periodo del clock dovrà essere determinato dal più lungo dei percorsi (in termini temporali) Δ_1, Δ_2, ..., Δ_5 fra un registro e un altro, cioè dall'elemento più lento della pipeline. Si avrà quindi una configurazione del tipo:



Di base, questa configurazione risulterà comunque un'accelerazione del clock. Infatti, se l'intera pipeline richiedeva prima un tempo Δ, ci aspettiamo che ogni componente in cui

la dividiamo richieda un tempo nell'ordine di $\sim \frac{\Delta}{5}$, ed esattamente $\frac{\Delta}{5}$ se ogni circuiteria ha lo stesso tempo di attraversamento, per cui il clock può essere accelerato di un fattore di 5.

Il bottleneck è però chiaro per la fase di esecuzione, che potrebbe andare dalla somma naturale alla divisione in virgola mobile, con evidenti differenze in tempo di esecuzione.

Inoltre, la stessa fase di prelievo potrebbe variare in requisiti temporali per via del tipo di codifica delle istruzioni, a lunghezza variabile, adottata dai processori Intel x86 (instruction set **CISC**, *Complex Instruction Set Computer*, contro gli instruction set **RISC**, *Reduced Instruction Set Computer*, adottati da ARM).

Ci troveremo quindi di fronte a situazioni dove, con una sola circuiteria di prelievo, non si può sapere quando un'istruzione è veramente finita prima di decodificarla, e quindi non si può procedere con una nuova fase di prelievo.

Chiaramente, tutto questo procedimento è semplificato per i processori ad architettura RISC, in quanto la dimensione delle istruzioni è standardizzata. Inoltre, si ottengono vantaggi nella gestione della pipeline eliminando la possibilità di avere operandi in memoria: si dedicano istruzioni dedicate alla lettura/scrittura in memoria da registri, cioe le `LOAD` e `STORE`, più semplici da gestire.

1.1.2 Alee

Altri problemi sono dati dalle **alee**, cioè legati ai salti condizionali (*alee di controllo*), o a casi dove istruzioni hanno bisogno di risultati di istruzioni ancora in pipeline (*alee di dati*), se non a casi dove la pipeline non permette in primo luogo l'esecuzione successiva di due istruzioni (*alee strutturali*).

- Questi problemi possono sempre essere risolti dall'introduzione di **bolle**: si modificano i registri intermedi perché possano conservare il loro stato, e in caso una certa rete di controllo rilevi situazioni a rischio di alee, si introducono nella pipeline *bolle*, cioè si lasciano stadi di elaborazione vuoti, o se vogliamo si introducono istruzioni `nop`, a effetto nullo (che chiaramente rappresentano throughput spreacto). I registri che alimentano gli stadi rimasti in attesa manterranno quindi una copia dell'istruzione allo scorso ciclo di clock, e invieranno invece avanti istruzioni nulle. In questo modo si torna effettivamente al processore prima della pipeline.

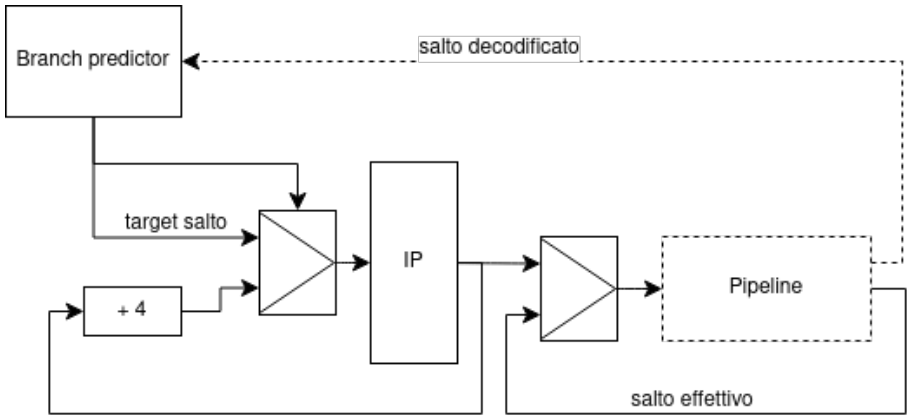
- Un'altra soluzione per alee dati e alee strutturali può essere quella di dotare l'ultima fase della pipeline di una linea di **bypass**, che porti il risultato a termine esecuzione a fronte della fase di esecuzione: un istruzione che richiede un operando non ancora scritto dall'istruzione precedente può ottenerlo direttamente da questa, attraverso il bypass.

- Per la gestione delle alee di controllo possiamo sfruttare la cosiddetta *esecuzione predittiva*, in particolare **branch prediction**: si fa un ipotesi sul risultato dell'istruzione di salto condizionale, e si riempie la pipeline con istruzioni che provengono dalla regione corrispondente di programma. Al momento della fase di esecuzione dell'istruzione di salto, si capisce quindi se l'ipotesi si è avverata o meno, e si procede ripulendola completamente introducendo bolle (in caso di *miss*, pagando un prezzo pari al numero di fasi della pipeline) o non facendo nulla (in caso di *hit*).

  Esistono politiche leggermente diverse in caso di salti **diretti** (con indirizzo noto) o **indiretti** (ad indirizzi calcolati).

- **Salti diretti**: in questo caso si fa una predizione **statica**, cioè si sceglie sostanzialmente a caso fra gli esiti del salto. Esistono comunque alcune euristiche che possiamo usare: nel caso di salti *all'indietro* ci si aspetta di entrare in un loop, e quindi si assume che il salto verrà eseguito; di contro per salti *in avanti* l'ipotesi è meno forte e ci si aspetta che il salto non verrà eseguito.

Abbiamo quindi a grandi linee la struttura funzionale:

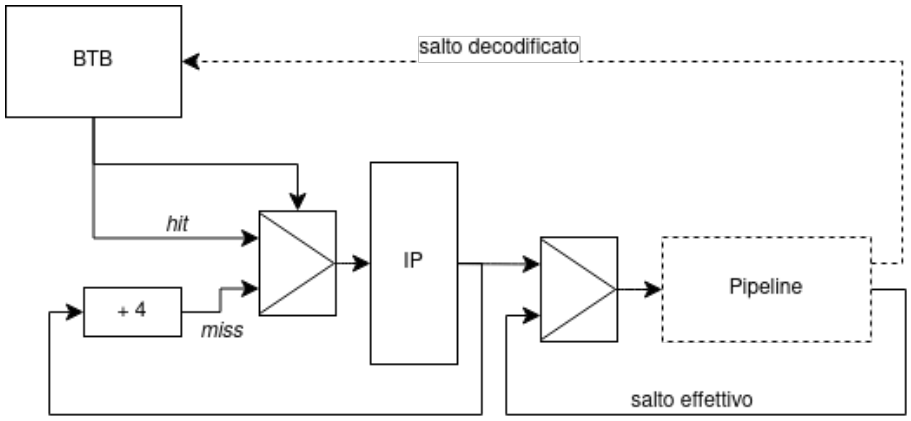


dove una qualche rete di branch prediction valuta le regole appena valutate per ogni istruzione di salto decodificata dalla pipeline, e invia al primo stadio della pipeline stessa la prossima istruzione che ne ricava. Notiamo che il $+4$ corrisponde al calcolo dell'indirizzo della prossima istruzione sequenziale: si assume che l'architettura abbia istruzioni RISC fisse a 4 byte (vedremo che nelle architetture moderne la semplificazione non è inopportuna).

– **Salti indiretti:** Per questi ci si aspetta che i salti si comporteranno come si sono comportati in precedenza, cioè si fa una predizione **dinamica**. Questo compito è associato ad una componente detta **BTB**, *Branch Target Buffer*, che si occupa di capire se un salto verrà effettuato o meno sulla base della sua storia precedente, e a quale locazione si salterà, sempre basandosi sulla storia precedente, nel caso di salti indiretti.

  Questi circuiti assumono essenzialmente la forma di cache, e visto che il loro insuccesso (*miss*) risulta solo in un annullamento di un ciclo di pipeline, non si preoccupano di gestire le collisioni.

  In questo caso la struttura funzionale è del tipo:



  cioè sostanzialmente analoga alla precedente ma dove l'ipotesi sul prossimo salto viene fatta dall'hit/miss della cache BTB.

1.1.3 Architettura del Pentium Pro

Nei processori di oggi, la gestione della pipeline è effettivamente quella che Intel sviluppa dal Pentium Pro del 1995 (che ha continuato ad evolvere fino ad oggi, se non per una deviazione che fu esplorata nel Pentium 4).

In questa architettura, il processore si occupa di tradurre intermante le istruzioni CISC in istruzioni RISC, e quindi a gestire la pipeline con sole istruzioni RISC. Vorremo quindi rimuovere le fasi di prelievo e decodifica dalla pipeline vista finora, per porle esternamente come parte del ciclo di traduzione da CISC a RISC, e aspettarci che la pipeline vera e propria si veda arrivare istruzioni già decodificate.

Notiamo infine che spesso nemmeno l'esecuzione sequenziale del codice non è necessaria. Prendendo ad esempio il frammento di codice:

```
for(int i = 0; i < 1000; i++) {
  a[i] = v1[i] * v2[i];
}
```

potremmo "srotolarlo" in:

```
a[0] = v1[0] * v2[0];
a[1] = v1[1] * v2[1];
// ...
a[999] = v1[999] * v2[999];
```

1.1.4 Esecuzione asincrona

Vediamo che nessuna di queste istruzioni dipende dalle altre, ergo l'ordine in cui vengono eseguite non è importante. In questo caso il miss di cache per alcune di queste può tradursi semplicemente in un ritardo nella loro esecuzione, mentre altre che invece si trovano in cache (magari relative ad indirizzi successivi) possono essere eseguite da subito, cioè l'esecuzione asincrona è non solo possibile, ma può risultare anche utile.

1.1.5 Esecuzione predittiva

Un'ultima tecnica che il processore moderno può usare per accelerare l'esecuzione delle istruzioni è eseguire istruzioni prima che questo sia necessario, eventualmente scartando i risultati nel caso risultino inutili.