
# High Performance and Quantum Computing

Anno Accademico
2025/26

**Rocco Lo Russo**
**Agostino D'Amora**

# Contents

# Introduzione

In questo documento vengono raccolti appunti del corso *High Performance and Quantum Computing* tenuto nell'anno 2025-26 dal professor Cilardo presso l'università di Napoli Federico II, Dipartimento Ingegneria Elettrica e Tecnologie dell'Informazione.
È assolutamente scoraggiata la vendita di questi appunti, che possono in qualsiasi momento essere scaricati gratuitamente da github. Allo stessa repository è possibile in qualsiasi momento contruibuire seguendo le istruzioni esposte nel README.
Per la parte iniziale di ricapitolazione, sono stati utilizzati gli appunti raccolti durante il corso di Architettura e Progetto dei Calcolatori tenuto nell'anno 2025-26 dal professor Mazzocca presso l'università Federico II di Napoli (a cui è possibile accedere sia per consultare che per contribuire, nelle stesse modalità, su github), i libri *Computer Organization and Desing Patterns* (Hannessy-Patterson) e degli stessi autori *Computer Architecture (a quantitative approach)*.

# Chapter 1

# Architetture superscalari

## 1.1 Richiami Pipelining

Il pipelining è una tecnica di implementazione di processori dove diverse istruzioni si sovrappongono durante l'esecuzione. Questa tecnica, chiave nella progettazione dei moderni processori, trae vantaggio dal parallelismo che esiste intrinsecamente tra le azione necessarie per l'esecuzione di un'istruzione. Nella pipeline, ogni passo necessario all'esecuzione di un'istruzione è svolto da un'unità funzionale autonoma, denominata *pipe segment* o *pipe stage*. Ogni unità è collegata alla successiva, e il throughput di una pipeline è determinato dal *tasso di attraversamento*, ovvero dalla frequenza con la quale un'istruzione esce dalla pipeline. Il tempo richiesto da un'istruzione per procedere allo stage successivo è denominato *processor cycle*, e dato che tutti gli stage devono necessariamente essere pronti a procedere allo stesso tempo, la lunghezza di un processor cycle è determinato dal tempo richiesto dallo stage più lento. In un computer il processor cycle è quasi sempre un colpo di clock.
Se il tempo di ogni stage è perfettamente bilanciato, chiamando $T_i$ il tempo per ogni istruzione sul processore pipelined, $T_{np}$ il tempo per ogni istruzione su un processore non pipelined ed $n$ il numero di stages della pipe, allora vale la relazione: $T_i = \frac{T_{np}}{n}$.
Sotto queste condizioni ideali, lo speedup vale proprio $n$. Tuttavia, il tempo di ogni stage non è mai perfettamente bilanciato, e in più c'è da considerare il tempo di overhead introdotto dalla complessità del sistema. Una semplice pipline a cinque stadi presenta i seguenti blocchi funzionali:

- **Instruction fetch**: Viene letto il PC per prelevare l'indirizzo della prossima istruzione; Si accede al registro Instruction Memory e si carica l'istruzione; Si calcola il prossimo valore del PC (+4).
- **Instruction Decode**: L'istruzione viene decodificata, ovvero l'unità di controllo capisce di che tipo di istruzione si tratta; si leggono i valori dei registri sorgente; si verifica l'estensione del segno.
- **Execute**: In questa fase avviene la vera elaborazione.
  - *istruzione aritmetico-logica* $\rightarrow$ l'ALU esegue l'operazione;
  - *istruzione load/store* $\rightarrow$ si calcola l'indirizzo effettivo (base + offset);
  - *istruzione branch* $\rightarrow$ si calcola la condizione e il nuovo PC;

- **Memory Access**: Operazioni di lettura/scrittura in memoria;
- **Write Back**: Scrittura nei registri del processore. Non tutte le istruzioni scrivono un registro.

Da questa presentazione emergono tre osservazioni: innanzitutto, è necessario separare memorie di istruzione e memorie dati, attraverso l'implementazione di diverse caches. Questa soluzione permette di evitare conflitti durante le fasi di IF e MEM; Il register file è usato in due stages, ovvero in ID e WB. Quindi, è necessario in un solo colpo di clock performare due letture e una scrittura, e questo si ottiene effettuando la scrittura nella prima parte di ciclo di clock e le lettura nella seconda parte; infine, è necessario considerare un circuito hardware che incrementi automaticamente il PC nella fase di IF, e un circuito che calcoli un eventuale indirizzo di branch durante lo stadio ID. La condizione del salto viene valutata in fase EXE, con una parte della ALU che confronta due registri.
Per fare in modo che l'output di uno stage non interferisca con lo stage successivo, vengono introdotti dei registri tra uno stage e l'altro, in modo che alla fine di un ciclo di clock, tutti i risultati di uno stage sono conservati in un registro che viene usato con input allo stage successivo al prossimo ciclo di clock. I registri svolgono un ruolo chiave anche nel trasporto di eventuali risultati intermedi dove stage di origine e destinazione non sono immediatamente adiacenti.

Il pipelining incrementa il throughput di istruzioni del processore, ovvero il numero di istruzioni completate per unità di tempo, ma non riduce il tempo di esecuzione di una singola istruzione. Al più infatti il tempo di esecuzione di ogni singola istruzione aumenta a causa dell'overhead introdotto nel controllo della pipeline. Questo fatto pone un limite importante sulla profondità pratica di una pipeline. In particolare, lo sbilanciamento tra i tempi di attraversamento di ciascuno stage implica una riduzione di perfomance in quanto il clock può essere soltanto veloce quanto il tempo richiesto dallo stage più lento; inoltre, l'overhead introdotto dalla pipeline è causato dalla combinazione di fattori quali il ritardo dei registri della pipeline e il *clock skew*. Il tempo di setup dei registri della pipeline è il tempo necessario affinchè l'input che viene dato ad un registro si stabilizzi prima che il clock attivi una scrittura. Il clock skew è il massimo ritardo che segna la ricezione del colpo di clock tra qualsiasi coppia di registri.

## 1.2 Pipeline Hazards

Vi sono situazioni, denominate *hazards*, nelle quali viene impedito alla prossima istruzione nel flusso di istruzioni della pipeline di venire eseguita nel ciclo di clock designato.

Distinguiamo: hazards **strutturali**, che sorgono a causa di conflitto tra risorse hardware, ovvero quando l'hardware non riesce a supportare tutte le possibili combinazioni di istruzioni simultaneamente in esecuzioni sovrapposte; hazards **sui dati**, che sorgono quando un'istruzione dipende dal risultato di un'istruzione precedente; hazards **di controllo**, che sorgono in casi di istruzioni di branch e in generale di istruzioni che modificano il PC. Spesso, per risolvere un hazard è necessario fermare la pipeline, ovvero forzare una *stall*. Nello situazione in cui un'istruzione viene fermata, tutte le istruzioni successive a questa vengono fermate, mentre quelle precedenti continuano il

loro flusso di attraversamento della pipe. Gli hazards strutturali sono poco frequenti in quanto riguardano unità funzionali hardware (come l'unità di divisione a virgola mobile) non utilizzate frequentemente, perchè sia i programmatori che i compilatori sono a conoscenza di questo tipo di hazards. Ci concentreremo principalmente sui data hazards e sui control hazards.

### 1.2.1 Hazards sui dati

La sovrapposizione di istruzioni introdotta dalla pipeline implica un cambiamento dell'ordine di accessi in lettura/scrittura agli operandi rispetto al modello sequenziale di Von Neumann. Un primo passo risolutivo è già stato presentato: quando si cerca di leggere o scrivere da uno stesso registro contemporaneamente, molte architetture prevedono che la scrittura avvenga nella prima metà del ciclo di clock mentre la lettura nella seconda metà. Questo risolve l'hazard quando la scrittura del registro avviene durante lo stesso ciclo di clock della lettura.
In linea di massima, il dato prodotto dall'istruzione da cui dipendono le successive è già pronto durante la fase di execute, quindi basterebbe semplicemente propagare il dato appena è possibile alle unità a cui serve prima che sia disponibile per la lettura dal file register. Ci occuperemo per semplicità solo del caso di propagazione in una fase EX, che può essere la propagazione di un risultato di una operazione ALU o il calcolo di un indirizzo effettivo. Questo significa che quando un'istruzione prova ad usare un registro nella sua fase EX che un'istruzione precedente deve ancora scrivere in fase di WB, può servirsi del valore anticipatamente come input controllato alla ALU.
Un caso in cui il forwarding non può risolvere l'hazard è quando un'istruzione prova a leggere un registro immediatamente dopo un'istruzione di *load* dalla memoria. Il dato risulterebbe ancora in fase di recupero dalla memoria, quindi è necessaria un'unità funzionale che stalli la pipeline. In generale, il compialtore, che è a conoscenza della struttura del processore, in casi di hazard detection di questo tipo si preoccupa di anticipare delle istruzioni indipendenti utilizzando la proprietà commutativa e associativa. Lo stallo è inevitabile quando non è possibile inserire dopo l'istruzione di accesso in memoria un blocco di istruzioni indipendenti. È possibile tuttavia un approccio, denominato **internal forwarding**, che permette di ibernare temporaneamente le istruzioni bloccate, e non stallare la pipeline. Bloccando istruzioni e caricandone altre, tuttavia può accadere che più istruzioni cerchino di operare sullo stesso registro, quindi occorre utilizzare una tecnica di indirizzamento indiretto che permette ai registri fisici del processore (*forwarding registers*) di puntare a dei registri in memoria che contengono valori precedenti assunti dal registro. Le istruzioni vengono ibernate in un'apposita area della memoria, la *ibernation table*.

### 1.2.2 Hazards di controllo

Quando sopraggiunge un istruzione di salto, riusciamo a captare che è così solo nella fase di esecuzione dell'istruzione, mentre la pipe ha continuato a caricare le istruzioni in modo sequenziale, che si troveranno rispettivamente nella fase di IF e ID e non avranno quindi ancora modificato lo stato del processore e della memoria. Nel caso in cui il salto non deve essere eseguito, allora la pipe continuerà a funzionare normalmente, mentre se il salto deve essere eseguito, le istruzioni caricate dovranno essere eliminate dalla

pipe (*pipe flush*) e dovrà essere caricata l'istruzione a cui punta il salto e le successive. Il ritardo che segue quest'ultimo caso è detto *branch penalty*. L'obiettivo è quello di confinare la gestione dei salti nelle fasi IF e ID, perchè in quelle fasi le istruzioni non hanno ancora modificato lo stato del processore e della memoria, e quindi la rete di controllo hardware deve riguardare solo queste due fasi. In generale, il problema è risolvibile fondamentalmente mediante due approcci: l'approccio conservativo e l'approccio ottimistico (*branch prediction*). Nel caso dell'**approccio conservativo**, quando il processore interpreta, durante la fase ID, un'istruzione come istruzione di salto, ferma la pipe e disabilita la propagazione dell'istruzione che si trova erroneamente nella fase IF, determina l'istruzione a cui saltare in fase EX e la preleva. Si torna insomma al modello sequenziale di Von Neumann. Il conto è salato se consideriamo che le istruzioni di salto costituiscono il 25% di un programma, e infatti ne consegue un notevole spreco delle risorse di parallelismo fornite dalla pipe. Questo approccio è *leggermente* migliorabile attraverso l'hardware, anticipando la decisione inerente al salto alla fase di ID, in base alla condizione di salto. Ad esempio l'istruzione `JNZ <LABEL>` controlla se il flag Z del SR è alto, e in tal caso non occorre saltare e quindi l'istruzione che si è prelevata nel frattempo è giusta. Molti ritardi possono essere evitati dal programmatore o dal compilatore: il programmatore può contribuire al buon funzionamento del sistema, scrivendo le istruzioni in un ordine tale da minimizzare le probabilità di stallo della pipe. Nel caso di un costrutto if-then-else, ad esempio, conviene inserire nel ramo then l'alternativa più probabile. Il compilatore, da parte sua, può operare vari accorgimenti; ad esempio, siccome un ciclo for è sempre tradotto in un if-then-else, esso deve inserire l'alternativa più probabile nel ramo then. In fase di compilazione è possibile evitare l'approccio conservativo. Se immediatamente prima del salto c'è un'istruzione con operandi da cui non dipende l'esito della scelta di saltare o meno, è possibile in fase di compilazione inserire l'istruzione indipendente dopo il branch, in modo da sfruttare lo slot di tempo in cui l'istruzione viene caricata ed entra nella pipe, e quindi non c'è bisogno di pipe flush. Qualora non sia possibile invertire una istruzione if con quella che la precede, il compilatore potrebbe decidere di mettere subito dopo la if una **nop**, istruzione che non ha alcun effetto e quindi non è mai errato inserirla nella pipe. In questo modo si può operare senza considerare alcun tipo di approccio. Possiamo aggiungere che se stiamo considerando un'istruzione di salto in un processore CISC con una condizione di salto elaborata, allora il numero di nop che bisogna inserire a seguito dell'istruzione di salto risulta essere superiore a 1, in quanto ricordiamo che la pipe per i CISC è più lunga e la valutazione della condizione coinvolge più fasi oltre le prime due, di caricamento e decodifica. Una soluzione meno conservativa a quella appena presentata è di utilizzare la **branch-prediction** $\rightarrow$ approccio ottimistico.
La branch prediction è una tecnica che cerca di prevedere a quale ramo un' istruzione di salto condizionato possa saltare. Consideriamo il classico esempio:

```
1 for (int i = 0; i < N; i++){
2     for (int j = 0; j < N; j++){
3         op...
4     }
5 }
```

Il controllo prevede l'esecuzione del ramo else una sola volta (guardando il for interno) ogni N passi. I modi per prevedere il branch possono essere vari, e possono essere

descritti mediante degli appositi automi a stati finiti. Un primo approccio molto basilare è quello di andare a cambiare il branch da caricare successivamente ad ogni errore di decisione e quindi eseguire le operazioni descritte dall'automa [1.1].

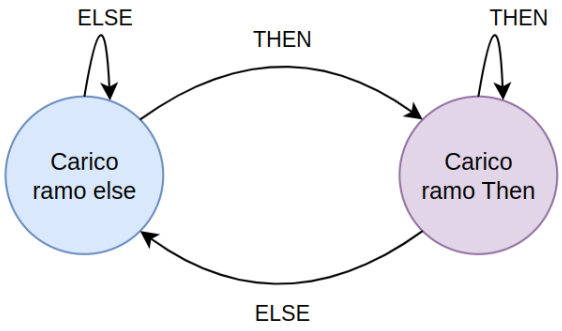


Figure 1.1: Automa della branch prediction base

Questa soluzione non è ottima poichè per quel singolo errore che avviene ad ogni N interazioni la pipeline dovrà assorbire 2 penalties. Per evitare tale condizione, e quindi rendere la persistenza più forte, consideriamo un automa a 4 stati che permette di rendere la condizione di "cambio del branch" più solida, poichè solo in caso di due errori successivi, allora cambio l'indirizzo di salto effettivo. L'automa a cui facciamo riferimento è [1.2].

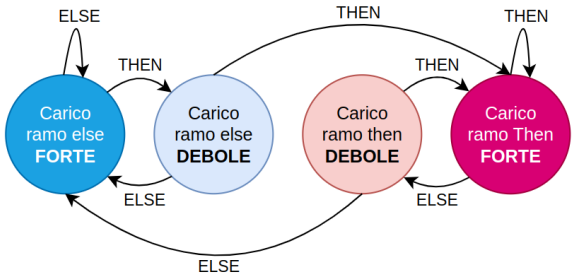


Figure 1.2: Automa della branch prediction avanzato

Per implementare la predizione a livello hardware, il processore utilizza una **tabella di predizione** dei salti. Una possibile struttura prevede i campi (Indirizzo dell'istruzione di salto, indirizzo di destinazione, 2 bit per codificare lo stato).
La tabella viene aggiornata in parallelo durante la fase di Execute (EX), ovvero nel momento in cui il processore verifica se la predizione è corretta. Se c'è stato un errore, si aggiorna lo stato dell'automa (e la corrispondente voce nella tabella). Tale struttura è memorizzata in un apposita area di memoria, detta **Branch History Table** (BHT).

Con l'introduzione del sistema a pipeline e del meccanismo dell'**internal forwarding**, che altera la sequenzialità delle istruzioni eseguite dal processore, nasce il problema della **gestione delle interruzioni**. Infatti può capitare che alcune istruzioni siano completate in un ordine diverso da quello in cui sono state avviate. In tal caso è possibile che si generino delle *interruzioni non esatte*, cioè situazioni in cui un' istruzione provoca un' eccezione prima che tutte quelle che la precedono siano completate. Ovviamente, per

la corretta gestione delle eccezioni stesse, l'obiettivo che ci si pone è quello di creare un sistema che sia in grado di arrestare opportunamente la pipe in modo da completare le istruzioni che precedono quella che ha generato l'interruzione senza avviare quelle che seguono, così da ottenere un comportamento che rispetti quello descritto dal modello di Von Neumann che viene definito di *gestione precisa* delle interruzioni. Le **interruzioni esterne** sono più facili da gestire: infatti la interrupt service routine relativa all'interruzione sopraggiunta verrà inserita nella pipeline come una normale istruzione, dopo che tutte le istruzioni precedenti verranno correttamente terminate, comprese quelle ibernate. Più critico è il caso delle **interruzioni interne**. Infatti questo genere di interruzioni possono essere causate da diverse istruzioni contemporaneamente all'interno della pipe in diversi stages. Non è dunque garantito il comportamento sequenziale di Von Neumann. Un esempio abbastanza lampante di conflitto è il caso in cui nella stessa pipe un'istruzione cerca di effettuare un accesso illegale in memoria (eccezione scatenata nella fase MEM) mentre un'altra istruzione, successiva, tenta una divisione per zero (eccezione scatenata nella fase ID).
Alcuni processori rinunciano del tutto ad avere interruzioni precise. In questo caso è compito del software assicurare che nella stessa pipeline non ci siano istruzioni le cui possibili eccezioni possano causare questo tipo di conflitto. Considerando il tasso di istruzioni che generano eccezione in un generico codice, questa soluzione può essere più o meno accettabile. Altri processori invece riescono a ricostruire interruzioni precise, utilizzando tecniche di tipo conservativo o ottimistico. Nelle tecniche di tipo **conservativo**, il processore fa procedere nella pipe un'istruzione finché è sicuro che non possa essere fonte di interruzione, e solo a questo punto ne avvia un'altra. Il parallelismo intrinseco della pipe è dunque limitato, a vantaggio della gestione precisa delle eccezioni.
Nel caso di tecniche di tipo **ottimistico**, vengono inserite comunque istruzioni nella pipeline indipendentemente dalla possibilità di generare o meno eccezioni. Se emergono conflitti il sistema deve essere in grado di effettuare un *rollback*, e quindi è necessaria un'immagine dello stato del sistema. Il rollback può essere effettuato mediante due tecniche:

- **Check point**: Vengono effettuate periodici snapshot dello stato dei registri del processore, compreso il PC, conservate in una opportuna area di memoria. Nel momento in cui si verifica un'eccezione, vengono ricopiati i valori memorizzati nei registri del processore e si riprende l'esecuzione *sequenziale* del programma, quindi un'istruzione per volta, a partire dall'ultimo valore di PC riportato nello snapshot. Occorre in questo caso scegliere con attenzione la frequenza di cattura dello snapshot.

- **History Buffer**: Ogni istruzione eseguita viene memorizzata in un'area di memoria insieme ai valori dei propri operandi. Ogni istruzione memorizzata può essere cancelalta dal buffer solo quando tutte quelle che la precedono si sono concluse senza generare interruzione. È possibile definire in questo modo una sequenza di operazioni UNDO da effettuare nel caso un'istruzione ibernata precedentemente causi un'eccezione. Questa tecnica è anche utile per garantire la gestione precisa delle interruzioni, facendo in modo che un'eccezione non venga gestita se prima non sono terminate correttamente (quindi senza causare eccezioni) le istruzioni ibernate.

## 1.3 Problemi nelle architetture superscalari

Per migliorare le prestazioni di un processore, si può pensare di realizzare un'architettura che presenti più pipe che eseguono diverse istruzioni in parallelo. Questo tipo di architettura viene definito **superscalare**. Questa soluzione introduce diverse problematiche. Una prima problematica riguarda il prelievo delle istruzioni in memoria. Le istruzioni vengono prelevate in maniera sequenziale dalla memoria condivisa, quindi non è possibile che due pipeline inseriscano contemporaneamente in esecuzione due istruzioni diverse. Se le pipeline sono completamente disgiunte, ovvero non condividono alcuna unità funzionale, allora l'unico altro problema è quello relativo alla *branch prediction*. I dispositivi per la predizione del salto devono essere, in questo caso, duplicati. Invece se le pipe condividono alcuni stages, la cooperazione e la competizione per le risorse condivise deve essere gestita in maniera più complessa via hardware. Quindi non c'è una vera e propria alimentazione in parallelo. Questa condizione di funzionamento, unita alla considerazione che le pipe condividono alcune unità funzionali, ci consente di affermare che il sistema nel suo complesso può essere visto come costituito da una sola pipe con più unità specializzate e con altre che sono in comune. Infatti è facile notare che nelle architetture superscalari ciascuna pipe è specializzata nell'esecuzione di particolari operazioni. Inoltre, per distribuire le istruzioni tra le pipe è possibile pensare di introdurre a monte della fase di EX di ciascuna pipe (dopo le fasi IF e ID) un'unità hardware, detta **Dispatch**, che analizza più istruzioni per volta e decide quali inserire in quale pipe. Ogni fase di ciascuna pipe, in generale, può essere utilizzata da più pipe per più cicli, quindi risulta particolarmente interessante l'idea di affidare al Dispatch anche il compito di riconoscere le caratteristiche di ciascuna operazione per evitare collisioni tra le pipe. Per le sue caratteristiche, la fase di dispatch non può essere realizzata in software, poiché deve essere molto veloce ed efficiente, quindi deve essere necessariamente realizzata in hardware. Strumento fondamentale sia per il compilatore che per il Dispatcher è il *collision vector*, foriero di informazioni utili riguardo la compatibilità temporale di più operazioni che necessitano di condividere la stessa unità funzionale. Per ulteriori dettagli, consultare gli appunti di APC.