Pipeline

Parallelismo

Il principio della pipeline permette di eseguire più attività contemporaneamente, completando il lavoro in meno tempo. In generale, con la pipeline otteniamo un parallelismo totale, eseguendo tante attività allo stesso tempo. Tuttavia, possono esistere dipendenze funzionali tra le singole attività.

Dipendenza funzionale tra lavori successivi

In una pipeline di dati, la dipendenza funzionale si riferisce alla relazione tra le diverse fasi o lavori dell'elaborazione, dove l'output di un lavoro diventa l'input del successivo. Questa connessione sequenziale crea una catena di dipendenze che influenza l'intero processo di elaborazione dei dati.

- Ogni lavoro è suddiviso in 3 fasi successive
- La fase 1 di *Lavoro i* deve essere eseguita dopo la fase 1 del precedente *Lavoro i-1*

La pipeline può avere degli **esecutori generici**, ognuno di questi esegue un lavoro completo, a regime hanno lo stesso **throughput** del parallelismo totale, inoltre, ognuno ha le risorse necessarie per svolgere ogni fase.

La pipeline, altrimenti, può avere degli **esecutori specializzati**, dove ogni esecutore svolge sempre la stessa fase di ognuno dei lavori ed solo le risorse per eseguire quella fase, ogni lavoro passa da un esecutore all'altro, a regime ha lo stesso **throughput** del parallelismo totale, ma usando meno risorse.

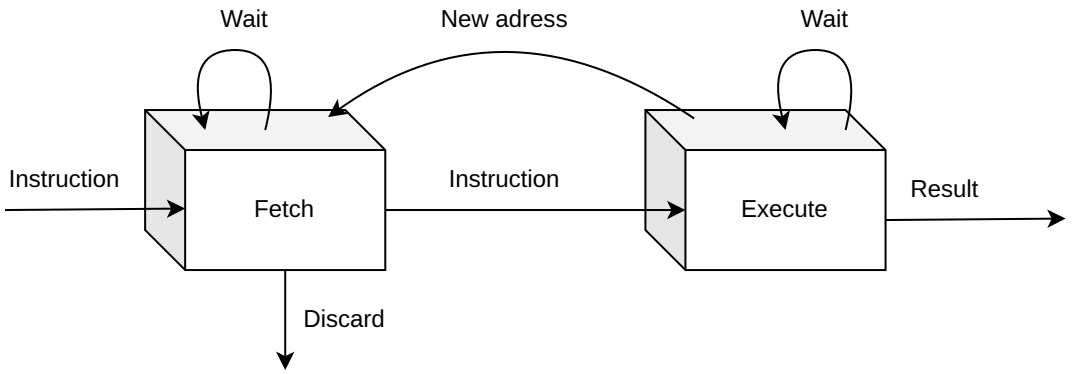
Catena di montaggio

La pipeline prevede di scomporre un lavoro in fasi successive. Un prodotto deve passare una fase dopo l'altra e ogni fase è realizzata da un diverso operatore. Nello stesso istante, istruzioni diverse sono in fasi diverse e, nell'istante successivo, ogni fase ripete lo stesso lavoro sull'istruzione che avanza alla fase successiva (**parallelismo**). Ogni fase è realizzata da una diversa unità funzionale della CPU ed operatori/fasi diverse usano risorse diverse per evitare conflitti. Tra due fasi successive si inseriscono dei **buffer** (registri) su cui si scrivono/leggono dati temporanei utili alla fase successiva.

Miglioramento prestazioni

Il ***prefetch*** non raddoppia le prestazioni, difatti la fase di fetch è più breve, ma prima di poter iniziare il fetch successivo deve attendere che termini anche la fase di esecuzione. Nel caso in cui si esegue un **jumo o branch**, la prossima istruzione da eseguire non è quella che è appena stata prelevata:

- La fase di fetch deve attendere che la fase execute le fornisca l'indirizzo a cui prelevare l'istruzione
- La fase successiva di execute deve attendere che sia prelevata l'istruzione, perchè quella pre-fetched non era valida



La suddivisione in fasi aggiunge overhead per spostare i date nei buffer tra una fase e l'altra e per gestire il cambiamento di fase. L'overhead può essere significativo quando:

- Le istruzioni successive dipendono logicamente da quelle precedenti
- Quando ci sono salti
- Quando ci sono conflitti negli accessi alle memoria/registri

La gestione logica e l'overhead aumentano con l'aumentare del numero di fasi della pipeline, è quindi necessaria una progettazione accurata per ottenere risultati ottimali con una complessità ragionevole.

Evoluzione ideale

- 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

Per aumentare le prestazioni bisogna decomporre il lavoro in un maggior numero di fasi, cercare di rendere le fasi più indipendenti e con una durata simile.
La tabella rappresenta l'evoluzione ideale di una pipeline in cui 6 istruzioni vengono eseguite in 11 cicli di clock, invece che di 6x6, quindi 36 cicli

	1	2	3	4	5	6	7	8	9	10	11
FI	Ist1	Ist2	Ist3	Ist4	Ist5	Ist6					
DI		Ist1	Ist2	Ist3	Ist4	Ist5	Ist6				
CO			Ist1	Ist2	Ist3	Ist4	Ist5	Ist6			
FO				Ist1	Ist2	Ist3	Ist4	Ist5	Ist6		
EI					Ist1	Ist2	Ist3	Ist4	Ist5	Ist6	
WO						Ist1	Ist2	Ist3	Ist4	Ist5	Ist6

Il tempo è rappresentato sull'asse orrizzontale, da 1 a 14. Sull'asse verticale sono presenti i 6 stadi di pipeline. La pipeline mostra come le istruzioni vengono eseguite in modo sovrapposto.

- Ogni istruzione attraversa sequenzialmente tutti gli stadi
- Ogni stadio richiede un ciclo di clock
- Una nuova istruzione inizia ogni ciclo di clock
- Il tempo 6 mostra come in un determinato momento ci siano più istruzioni in esecuzione simultanea

Nel tempo 6 si nota come sia il momento di massimo parallelismo, ossia c'è la massima efficienza e il migliore utilizzo delle risorse hardware. Per un'evoluzione ideale della pipeline è necessario fare delle supposizioni:

- Ogni fase deve avere durata uguale
- Ogni istruzione passa per tutte le fasi
- FI, FO, WO possono accedere alla memoria parallelamente senza creare conflitti
- Non ci devono essere salti, interrupt e dipendenze
  La performance della pipeline viene descritta da: il tempo di un ciclo (τ), ossia il tempo necessario per far avanzare di uno stadio/fase le istruzioni attraverso una pipeline, si può determinare tramite la formula:

$$\tau = \max_i[\tau_i] + \mathrm{d} = \tau_m + \mathrm{d} \qquad 1 \leq i \leq k$$

$\max_i$ = stadio i più oneroso
τ_i = tempo di ciclo i
d = ritardo di comunicazione di un registro, richesto per l'avanzamento di segnali e dati da uno stadio al successivo
τ_m = massimo ritardo di stadio
k = numero di stadi nella pipeline

Le performance ideali di una pipeline sono:

- **Tempo totale** richiesto da una pipeline con k stadi per eseguire n istruzioni

$$\mathrm{T}_k = [k + (n-1)] \cdot \tau$$

Infatti in k cicli si completa la prima istruzione, in altri $n-1$ cicli si completano le altre $n-1$ istruzioni (ogni istruzione finisce la sua pipeline un ciclo dopo la precedente)

- **Speedup** (fattore di velocizzazione), rappresenta il rapporto tra il tempo di esecuzione di un'operazione senza pipeline e il tempo di esecuzione della stessa operazione con l'utilizzo della pipeline. In termini matematici:

$$\mathrm{S}_k = \frac{\mathrm{T}_1}{\mathrm{T}_k} = \frac{nk\tau}{[k+(n-1)]\tau} = \frac{nk}{[k+(n-1)]}$$

Pipeline hazards

Ci sono varie situazione critiche in cui l'istruzione successiva non può essere eseguita nel ciclo di clock immediatamente successivo (***stallo***, ossia uno stadio in cui non viene eseguito nessun lavoro utile per mancanza di dati da uno stadio precedente), non si raggiunge mai il parallelismo massimo.

Sbilanciamento delle fasi

Si verifica quando si hanno durate diverse per fase e per istruzione: non tutte le fasi richiedono lo stesso tempo di esecuzione. Di conseguenza la suddivisione in fasi è in base all'istruzione più onerosa, inoltre non tutte le istruzioni richiedono le stesse fasi e risorse.

Di conseguenza, quando uno stadio richiede più tempo degli altri, gli stadi successivi devono attendere il completamento dello stadio più lento, si possono formare delle bolle nella pipeline e il throughput complessivo viene limitato dallo stadio più lento.
Le possibili soluzioni possono essere:

- Decomporre le fasi onerose in più sottofasi, ciò comporta ad un costo elevato e bassa utilizzazione
- Duplicare gli esecutori delle fasi più onerose e farli operare in paralllelo, le CPU moderne hanno una ALU in aritmetica intera e una in virgola mobile

Problemi strutturali

Si verfica quando due o più istruzioni già nella pipeline richiedo di accedere ad una stessa risorsa nello stesso ciclo di clock; gli accessi devono avvenire, quindi, in sequenza e non in parallelo. Per esempio FI, FO, WO potrebbero dover accedere alla memoria principale (perchè i dati non si trovano né nella cache né nei registri).
Possibili soluzioni:

- Introdurrre fasi non operative (stalli), per ritardare l'istruzione successiva che deve accedere alla memoria
- Suddividere le memorie permettendo accessi paralleli, una memoria cache per le istruzioni e una per i dati

Dipendenza dei dati

Si verifica quando, un'istruzione dipende dal risultato di un'istruzione precedente, la quale si trova ancora all'interno della pipeline. Quindi, una fase non può essere eseguita in un certo tempo di clock, perchè i dati di cui ha bisogno non sono ancora disponibili; deve attendere che termini l'esecuzione di un'altra fase. Un dato modificato nell'esecuzione dell'istruzione corrente può dover essere utilizzato dalla fase di FO dell'istruzione successiva.
Ci sono tre tipi di conflitto:

```
        istruzione i
        istruzione i+1
```

1. ***Read after Write***: per esempio viene letto il valore `i+1` prima che `i` venga scritto
2. ***Write after Write***: per esempio viene scritto prima il valore `i+1` prima di `i`

3. ***Write after Read***: per esempio `i+1` viene scritto prima che `i` venga letto (caso raro)

Possibili soluzioni sono:

- L'utilizzo di fasi non operative (***nop***-stallo)
- Propagazione in avanti del dato richiesto (***data forwording***-bypassing): si basa sul fatto che il dato si può conoscere già all'uscita dall'ALU, quindi prima che venga memorizzato, di conseguenza si può intercettare tale date ed eliminare degli stalli
- Riordino delle istruzioni

Dipendenza dal controllo

Uno dei maggiori problemi della progettazione della pipeline è assicurare un flusso regolare di istruzioni. Tale problema, si verifica quando sono presenti istruzioni che alterano la sequenza di esecuzione (salti condizionati o no, chiamate e ritorni da procedure, interruzioni), oppure se la fase di fetch ha caricato un'istruzione errata che va scartata. Queste istruzioni occupano circa il 30% del totale medio di un programma.
Le soluzioni sono:

- mettere in stallo la pipeline finchè non si è calcolato l'indirizzo della prossima istruzione, metodo semplice ma inefficiente
- individuare le istruzioni critiche ed aggiungere un'apposita logica di controllo, si complica il compilatore e l'hardware

Salto condizionato

Il salto condizionato è un'istruzione che altera il flusso sequenziale di esecuzione del programma in base ad una condizione.
Possibili soluzioni:

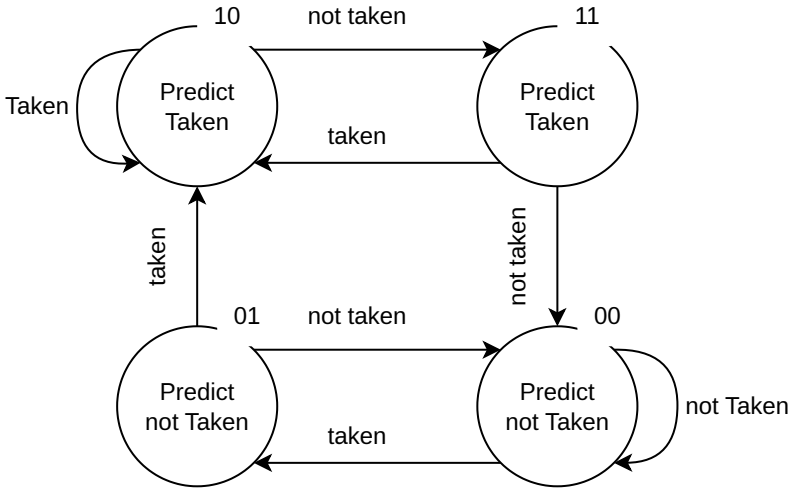
1. Flussi multipli (*multiple streams*): replica la prima parte della pipeline, EI esclusa, per entrambi i rami possibili
2. Prefetch anche dell'istruzione target:
    - anticipa il fetch dell'istruzione target oltre a quella successiva al salto
    - se il salto è preso, trova l'istruzione già caricata
    - in ogni caso una parte della pipeline deve essere scartata
3. Buffer circolare (*loop buffer*):
    - si tratta di una memoria piccola e veloce che mantiene le ultime n istruzioni prelevate
    - in caso di salto l'hardware controlla se l'istruzione target è tra quelle già dentro il buffer, così da evitare il fetch
    - utile in caso di loop specie se il buffer contiene tutte le istruzioni nel loop, così vengono prelevate dalla memoria una sola volta

- può essere accoppiato al prefetch: riempio il buffer con un po di istruzioni sequenzialmente successive alla corrente. Per molti *if-then-else* i due rami sono istruzioni vicine, quindi probabilmente entrambe già nel buffer
- dato l'indirizzo target di salto/branch controllo se presente nel buffer, gli 8 bit meno significativi sono usati come indice, i restanti servono per controllare se la destinazione del salto è nel buffer

4. Predizione dei salti, cerco di predirre se il salto sarà intrapreso o no
   - Approccio **statico**:
     - prevedo di saltare sempre
     - prevedo di non saltare mai
     - prevedo in base al codice operativo
   - Approccio **dinamico**:
     - Bit taken/not taken
     - tabella della storia dei salti

Gli approcci dinamici tentano di migliorare la qualità della predizione sul salto, memorizzando la storia delle istruzioni di salto condizionato di uno specifico programma. Ad ogni istruzione di salto condizionato associo 1 o 2 bit per ricordare la storia recente dell'istruzione. I bit sono memorizzati in una locazione temporanea ad accesso molto veloce.

- Il salto condizionato a 1 bit è una delle tecniche più semplici utilizzate nella predizione dei salti. Viene utilizzato 1 solo bit per memorizzare la previsione per ciascuna istruzione di salto condizionato, 0 dice che il salto non sarà preso, 1 dice che il salto sarà preso. Dopo ogni esecuzione effettiva del salto, il bit viene aggiornato, se il salto è stato preso,il bit viene impostato a 1, se il salto non è stato preso, il bit viene impostato a 0 (il valore del bit viene invertito). A pieno regime, fa 2 errori per ciclo.
- Il salto condizionato a 2 bit è una tecnica più avanzata rispetto alla precedente. Il predittore utilizza 2 bit per ogni istruzione di salto, permettendo quattro stati. Per invertire la predizione servono 2 errori consecutivi e in questo modo a regime fa un solo errore per ciclo



La predizione dinamica usa il buffer di predizione dei salti, una piccola memoria associata allo stadio di fetch della pipeline. Ogni riga della tabella è costituita da 3 elementi:

- L'indirizzo dell'istruzione salto
- I bit di predizione
- L'indirizzo destinazione del salto, così quando la predizione è di saltare non devo attendere che si ri-decodifichi il target del salto

Da questa predizione nei salti dipendono due vulnerabilità nei processori: ***Spectre*** e ***Meltdown***, ossia vulnerabilità di sicurezza molto pericolose che permettono a soggetti malintenzionati di aggirare le protezioni di sicurezza del sistema presenti nei dispositivi recenti dotati di CPU. Sfruttando il duo, è possibile leggere la memoria di sistema protetta, ottenendo l'accesso ad informazioni sensibili.
Specter e *Meltdown* sono esempi di attacchi di ***esecuzione transitoria***, che si basano su diffetti di progettazione hardware nell'implementazione dell'esecuzione speculativa, del pipelining delle istruzioni e dell'esecuzione fuori ordine nelle CPU.
Gli attachi ***Spectre*** hanno 3 fasi:

- La fase di setup, dove si distrae il processore per fare una previsione speculativa errata
- Il processore esegue speculativamente le istruzioni dal contesto di destinazione in un canale nascosto dalla microarchitettura
- I dati sensibili vengono recuperati, questo è possibile temporizzando gli accessi agli indirizzi di memoria nella cache della CPU

Anche gli attacchi ***Meltdown*** si svolgono in 3 fasi:

- Il contenuto di memoria scelto dall'attaccante viene caricato in un registro
- Un'istruzione transitoria accede ad una linea della cache in base al contenuto segreto del registro
- L'aggressore utilizza Flush+Reload per determinare la linea di cache a cui si accede e quindi il segreto memoriazzato nella posizione di memoria scelta inizialmente

5. Salto ritardato (*delayed branch*):
    - l'istruzione resta in pipeline, finchè non si sa se ci sarà o no il salto, invece di restare in stallo
    - istruzione successiva al salto: *branch delay slot*
    - il compilatore cerca di allocare nel *branch delay slot* un'istruzione opportuna
    - la CPU esegue sempre prima l'istruzione del *branch delay slot* e solo dopo altera, se necessario, la sequenza di esecuzione delle istruzioni

Ci sono 2 tipi:

- ***From before***: riempio il *branch delay slot* con una istruzione indipendente precedente al salto. Es:

```
ADD R1, R2, R3
if R2 = 0 then Target
```

```
    ADD R1, R2, R3   --> branch delay slot
Target..
```

- ***From target***: riempio il *branch delay slot* con l'istruzione target del salto. Non è più una istruzione independente. Utile nel caso il salto sia molto probabile. Se il salto avviene allora l'istruzione target viene eseguita di sicuro essendo nel delay slot e poi si continua con l'istruzione successiva al target. Se il salto non avviene il target viene eseguito, ma ignorato e si continua con la normale esecuzione. Es:

```
Target: SUB R4, R5, R6
ADD R1, R2, R3
if R1 = 0 then Target
    SUB R4, R5, R6  --> branch delay slot
```