Il Pipelining

Cos'è il Pipelining?

Occorrono parecchie ore per montare un'automobile; come può una fabbrica di automobili produrre un'automobile ogni pochi minuti? Usando una catena di montaggio con molte stazioni: ogni stazione aggiunge una parte o due al telaio in alcuni minuti e quindi trasmette il telaio alla stazione seguente. Possono passare diverse ore prima che un telaio passi attraverso un intera catena di montaggio ma questo non è importante: il rendimento della linea di produzione è una nuova automobile ogni pochi minuti.

Riprendendo il ciclo completo della rete di governo, vediamo che vengono eseguite in sequenza le operazioni :

- Fetch Instruction
- Decode instruction
- Read Operand (fetch dei dati dai registri, costruzione dell'effettivo indirizzo di memoria)
- Execute instruction
- Write Operand (trasferimento dei dati da/dalla memoria, scrittura dati sui registri, etc.)

Per ogni fase occorre uno o più cicli di clock. Come fa una CPU moderna ad eseguire un'istruzione ogni ciclo di clock? Usando una *pipeline*.
La *pipeline* è composta da un certo numero di fasi ed ogni fase effettua un'operazione sull'istruzione per ogni ciclo di clock. Le istruzioni attraversano la pipeline così come avviene per le automobili in una catena di montaggio.
Ogni fase della pipeline deve ultimare i relativi lavori in un ciclo di clock: in questo modo il periodo di clock è determinato dalla fase più lunga. Per minimizzare il periodo di clock il lavoro dovrebbe essere equilibrato così tutte le fasi completino le loro operazioni in tempi quasi uguali.
Per fissare le idee, prendiamo come modello una delle istruzioni più lunghe, ovvero quella relativa al caricamento di un valore dalla memoria con un indirizzamento indicizzato (esempio: Ry ← Mem[Rx+100]): occorre eseguire le seguenti operazioni:

1. fetch dell'istruzione dalla memoria usando il PC come indirizzo;
2. incrementare il PC per puntare all'istruzione successiva;
3. decodificare l'istruzione;
4. leggere il valore dal registro indice (Rx);
5. calcolare la somma del campo immediato e del valore del registro indice per trovare l'indirizzo di memoria effettivo (Rx+100).
6. leggere l'operando dalla memoria usando l'indirizzo di memoria effettivo;
7. scrivere il valore letto dalla memoria nel registro destinazione (Ry).

Le azioni 1 e 2 possono essere realizzate simultaneamente: il valore corrente del PC è usato per prelevare l'istruzione dalla memoria mentre si sta calcolando il nuovo valore del PC;
Le azioni 3 e 4 possono anche essere realizzate simultaneamente: il registro indice può essere letto mentre si sta decodificando il codice operativo.

L'attuale tecnologia permette di:

• Leggere un istruzione dalla cache-memory,
• Leggere un registro da un file-register
• Calcolare un indirizzo effettivo di memoria
• Leggere un operando una cache-memory
• Scrivere un valore in un registro

in tempi quasi uguali: questo consente di usare una pipeline a cinque-fasi secondo il seguente schema:

A Five-Stage Pipeline

IF	ID	EX	MEM	WB
Fetch dell'istruzione dalla memoria usando il PC come indirizzo; Incrementa il PC per puntare all'istruzione successiva;	Decodifica l'istruzione; Legge il valore dal registro indice;.	Esegue l'operazione nell'ALU o calcola l'indirizzo di memoria effettivo; Test la condizione di branch;	Legge o scrive l'operando in memoria; Modifica il PC in caso di salto o branch vero.	Scrive il risultato dell'ALU o l'operando da memoria nel registro di destinazione;

Nota 1:

La fase di ID preleva sempre i valore da un registro indice senza considerare il codice operativo: in realtà occorre prelevare, da una posizione fissa del codice operativo, l'identificatore del registro indice. Questo comporta il caricamento inutile del registro indice nel caso l'istruzione non lo usi ma, se si aspettasse la decodifica, si avrebbe un rallentamento qualora si dovesse caricare il valore da uno dei registri indice; eventuali istruzioni che non hanno bisogno dei registri indice ignoreranno i valori caricati.

Nota 2:

Per un jump o un branch vero, il PC viene cambiato dalla fase di MEM. Nel frattempo, però, già le tre istruzioni seguenti sono state prelevate e caricate nella pipeline. Questo è un tipico caso di "*control-hazard*" che verrà discusso in seguito.

Diagrammi Temporali della Pipeline

Il flusso delle istruzioni attraverso la pipeline può essere indicato con un diagramma temporale ovvero con la seguente tabella:

	IF	ID	EX	MEM	WB
T1	Istruzione i				
T2	istruzione i+1	istruzione i			
T3	istruzione i+2	istruzione i+1	istruzione i		
T4	istruzione i+3	istruzione i+2	istruzione i+1	istruzione i	
T5	istruzione i+4	istruzione i+3	istruzione i+2	istruzione i+1	istruzione i
T6		istruzione i+4	istruzione i+3	istruzione i+2	istruzione i+1
T7			istruzione i+4	istruzione i+3	istruzione i+2
T8				istruzione i+4	istruzione i+3
T9					istruzione i+4

Pipeline Latches

Affinché ogni fase della pipeline possa operare indipendentemente dalle altre, è necessario immagazzinare temporaneamente le istruzioni, gli operandi, i registri indice, ecc., in opportuni "buffer" o "latches".

IF stage	**IF/ID latch**	ID stage	**ID/EX latch**	EX stage	**EX/MEM latch**	MEM stage	**MEM/WB latch**	WB stage

Ogni latch della pipeline viene caricato con le informazioni dalla fase sulla relativa parte di sinistra: il latch conserva le informazioni durante il ciclo di clock seguente mentre la fase sulla relativa destra le usa.

Le seguenti tabelle sono un esempio dei campi di informazioni da conservare per ogni latch della pipeline.

IF/ID Latch

Nome	Descrizione
NPC	Prossimo valore del PC
IR	Istruzione

ID/EX Latch

Nome	Descrizione
NPC	Prossimo valore del PC
IR	Istruzione
A	Valore Registro indice
Imm	Valore immediato

EX/MEM Latch

Nome	Descrizione
cond	Bit-flag per branch preso
ALUO	Uscita ALU
B	Indirizzo di memoria calcolato
IR	Istruzione

MEM/WB Latch

Nome	Descrizione
LMD	Operando in Memoria
ALUO	Uscita ALU
IR	Istruzione

Come si aumentano le prestazioni tramite la pipeline?

Ignorando le "pipeline hazards" (discusse più avanti) la pipeline esegue un'istruzione ogni ciclo di clock (se il numero di clock per istruzione CPI = 1). Il periodo di clock è determinato dalla fase più lunga. Supponiamo che:

- la fase IF richieda 10 nS,
- la fase di ID richieda 8 nS,
- la fase EX richieda 10 nS,
- la fase di MEM richieda 10 nS
- la fase di WB richieda 7 nS.

La fase più lunga richiede 10 nS; la pipeline aggiunge un ulteriore overhead di 1 nanosecondo a causa dei latch della pipeline e dell'obliquità della rampa di clock.
Il periodo di clock sarà quindi di 11 nS ovvero la frequenza di clock sarà di 90,909 MHz. La pipeline eseguirà le istruzioni ad una velocità di 90,909 MIPS (se CPI = 1).

Senza pipeline, il periodo di clock può essere ridotto a 10 nS (la frequenza di clock è quindi di 100 MHz). L'istruzione successiva non viene prelevata prima che sia completa l'istruzione corrente. Ogni istruzione di *load da memoria* richiede cinque cicli di clock; supponiamo che le altre istruzioni richiedano soltanto quattro cicli di clock.
Supponiamo inoltre che il 40% delle istruzioni siano *load da memoria*. Il CPI sarà quindi:
(40%) * (5 cicli) + (60%) * (4 ciclo) = 4,4 cicli/istruzione ovvero le istruzioni vengono eseguite ad una velocità di (100) / (di MHz 4,4) = 22,727 MIPS.
Se la situazione è la stessa per qualsiasi programma possiamo dire che la pipeline migliora la prestazione della CPU di un fattore di (90,909 MIPS) / (22,727 MIPS) = **4,0**

Un altro modo di procedere senza pipeline consiste nel prendere come riferimento il periodo di clock relativo all'istruzione più lunga. Un'istruzione di *load da memoria* richiede 10 + 8 + 10 + 10 + 7 = 45 nS ovvero il periodo di clock è di 45 nS e la frequenza di clock è di 22,222 MHz. Ogni istruzione è eseguita in un ciclo di clock ovvero CPI = 1,0 le istruzioni vengono eseguite alla velocità di 22,222 MIPS. In questo caso la pipeline migliorerà la prestazione della CPU da fattore di (90,909 MIPS) / (22,222 MIPS) = **4,09**

In definitiva, l'uso del pipelining migliora le prestazioni del CPU da un fattore di 4,0 o di 4,09 a seconda della strategia usata. Il calcolo effettuato, comunque, non tiene conto dei *pipeline hazards* per cui il miglioramento reale delle prestazioni sarà più basso.

Pipeline hazards (Conflitti potenziali nella pipeline)

Si possono distinguere tre classi di *pipeline hazards*:

1. **Strutturali:** (*Structural-hazard*) si presentano quando due o i più fasi della pipeline provano a realizzare delle azioni in conflitto sulla stessa risorsa di hardware.
2. **Sui dati:** (*Data-hazard*) si possono presentare quando l'ordine temporale delle read/write su una variabile è permutato.

3. **Di controllo:** (*Control-hazard*) si presentano quando le istruzioni modificano il normale incremento del PC.

Un *hazard* può essere corretto bloccando l'istruzione critica nella fase della pipeline per uno o più cicli di clock fino a che non esiste più conflitto e quindi è possibile trattare l'istruzione.
Supponiamo che un'istruzione *i* si blocca nella fase *k* della pipeline.

- Il blocco viene effettuata lasciando immutato il latch della pipeline precedente la fase *k*; il latch conserva l'istruzione *i* invece di cambiare all'istruzione i+1.
- Le istruzioni che seguono l'istruzione i (i+1, i+2, ecc.) nelle fasi precedenti della pipeline (k-1, k-2, ecc.) devono essere bloccate cosicché tutti i latches precedenti della pipeline (ed il PC) rimangono invariati. Tutte le istruzioni seguenti sono bloccate piuttosto che continuare ad attraversare la pipeline.
- Le istruzioni che precedono *i* (i-1, i-2, ecc.) nelle fasi successive della pipeline (k+1, k+2, ecc.) continuare il loro normale flusso attraverso la pipeline.

In pratica si crea una discontinuità (una bolla) fra le istruzioni *i-1* ed *i*. La bolla attraversa le fasi k+1, k+2, ecc., della pipeline e non deve cambiare alcun registro, posizione di memoria, ecc. Il latch successivo la fase *k* della pipeline viene quindi caricato con un'istruzione **nop** (No Operation) cosicché la bolla non fa nulla.

Come esempio di blocco, consideriamo il seguente frammento di codice:

1. R6 ← Mem[R1+100]
2. R1 ← R2+R3
3. R4 ← R1-R5
4. R3 → Mem[R2+10]

Questo codice presenta un *data-hazard* tra l'istruzione 2 e 3; l'istruzione 3 deve essere bloccata nella fase ID fino a quando la fase WB ha finito di caricare il registro R1 con il risultato della somma. Possiamo tracciare il seguente diagramma temporale:

	IF	**ID**	**EX**	**MEM**	**WB**
T1	R6←Mem[R1+100]				
T2	R1 ← R2+R3	R6←Mem[R1+100]			
T3	R4 ← R1-R5	R1 ← R2+R3	R6←Mem[R1+100]		
T4	*R3→ Mem[R2+10]**	*R4← R1-R5 (bloc.)*	R1 ← R2+R3	R6←Mem[R1+100]	
T5	*R3→ Mem[R2+10]**	*R4← R1-R5 (bloc.)*	*Bolla 1*	R1 ← R2+R3	R6←Mem[R1+100]
T6	*R3→ Mem[R2+10]**	*R4← R1-R5 (bloc.)*	*Bolla 2*	*Bolla 1*	R1 ← R2+R3
T7	R3→ Mem[R2+10]	R4 ← R1-R5	*Bolla 3*	*Bolla 2*	*Bolla 1*
T8		R3→ Mem[R2+10]	R4 ← R1-R5	*Bolla 3*	*Bolla 2*
T9			R3→ Mem[R2+10]	R4 ← R1-R5	*Bolla 3*
T10				R3→ Mem[R2+10]	R4 ← R1-R5
T11					R3→ Mem[R2+10]

Al clock T4 il *data-hazard* viene rilevato dalla fase **ID** cosicché le istruzioni R4 ← R1-R5 e R3 → Mem[R2+10] rimangono bloccate nelle fasi **ID** e **SE** (inibendo la scrittura dei latch **IF/ID** e del PC) e una *nop* (*Bolla 1*), essere inserita nel latch **ID/EX**. La bolla 1 entra nella fase **EX** fase al clock T5.
Nei periodi T5 e T6 il *data-hazard* ancora esiste così le fasi **IF** e **ID** devono ancora rimanere bloccate e le *Bolla 2* e *Bolla 3* vengono inserite nella fase **EX**.
Al T6 la fase di **WB** memorizza il risultato nel registro R1 che elimina *data-hazard* e quindi le istruzioni precedentemente bloccate possono continuare ad attraversare la pipeline durante i periodi da T7 a T11.

Nell'esempio precedente il *data-hazard* crea 3 cicli di blocco (e 3 di bolla) ovvero le 4 istruzioni richiedono 7 clicli di clock ciclo anziché i 4 previsti. Il CPI per questa sequenza di codice a 4 istruzioni diventa 7 / 4 = 1,75 anziché 1. L'aumento di performance della pipeline di 4,0 (o di 4,09) che avevamo calcolato precedentemente scende a 4,0/1,75 = 2,29 (o 4,09 / 1,75 = 2,34) durante l'esecuzione di queste 4 istruzioni

Il calcolo del fattore di miglioramento delle prestazioni dovuto alla pipeline, e quindi il CPI effettivo, dovrebbe essere basato su grandi sequenze di codice nel seguente modo:

CPI = (numero di cicli di clock) / (numero di istruzioni)

dove:

(numero di cicli di clock) = (numero di istruzioni) + (numero di cicli bloccati)

per cui:

CPI = 1 + (numero di cicli bloccati) / (numero di istruzioni).

Calcoliamo ad esempio il CPI nel caso che il 15% delle istruzioni di un lungo programma provochino un *hazard* ed ogni *hazard* causi 3 cicli bloccati:

CPI = 1 + (15%) * (3) = 1,45

Il fattore di miglioramento scende quindi a: 4,0/1,45 = 2,76

Conclusioni

Da questa breve trattazione possiamo concludere che, comunque, l'utilizzo del pipelining migliora le prestazioni della singola CPU, tendendo a portale il CPI ad un valore unitario, senza mai raggiungerlo.
Il pipelining ha messo in evidenza i potenziali conflitti *(hazard)* che si vengono a creare, quando si intende trasformare il ciclo sequenziale della Rete di Governo in un **processo parallelo**: tali potenziali conflitti devono essere risolti ed occorre introdurre nuove tecniche per poterli risolverli al meglio, riducendo al minimo l'uso di *nop*.

La strada successiva è quella di superare la barriera unitaria del CPI, introducendo più *moduli di elaborazione* identici (ad esempio più di una ALU) in modo da poter trattare più di una istruzione per ciclo di clok.
In questo modo si verrebbero a realizzare situazioni effettive di *parallelismo* che introdurrebbero ulteriori potenziali conflitti, ad esempio nel caso di sequenza di istruzioni *dipendenti*, da trattare opportunamente.
Le architetture che prevedono di superare la barriera di 1 CPI vengono dette **scalari** identificando in questo termine la possibilità (teorica) di raddoppiare il numero di istruzioni eseguibili per ciclo di clock semplicemente raddoppiando i necessari moduli di elaborazione. Per far questo, dovrà essere introdotta una fase ulteriore nel ciclo della RG che *smisti* le istruzioni decodificate alle unità di elaborazione attualmente libere evitando potenziali conflitti.

I moderni studi sulle architetture degli elaboratori sono indirizzati appunto verso questa direzione ovvero quella di riuscire ad utilizzare al meglio le unità di elaborazione multiple che è possibile implementare in un singolo chip grazie all'aumento di densità di transistor (aumento del 50% ogni 18 mesi) e di dimensioni del singolo chip (aumento del 10% ogni 18 mesi), resi possibili dal progresso tecnologico nella costruzione dei circuiti integrati.