E' una tecnica implementativa dove più istruzioni sono interfogliate nell'esecuzione.
Diverse unita, dette **pipe stage**, completano diverse parti di diverse istruzioni in parallelo
Il **throughput** di un processore con **pipeline** è il numero di istruzioni che escono dalla pipeline per unità di tempo.

Stati di pipeline

- **IF - Instruction Fetch:** Preleva l'istruzione dalla memoria dove risiede il codice
- **ID - Instruction decode/register fetch:** Decodifica l'istruzione/prendere i registri
- **EX - Execution/effective address:** Esegue l'istruzione/calcolo dell'indirizzo reale
- **MEM - Memory access/branch completion:** Accesso alla memoria/completare i branch
- **WB - Write Back:** Per alcune istruzioni(load), scrive il valore sul registro

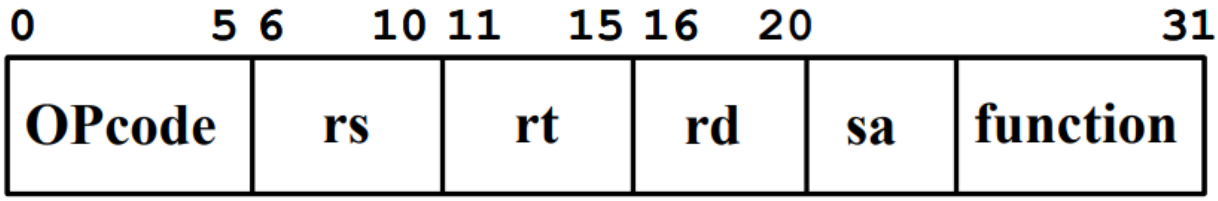
Instruction Fetch Cycle

IR = MEM[PC]
NPC = PC + 4
dove **PC** è il program counter e **NPC** è New Program Counter

Instruction decode/Register fetch cycle

Decodifica l'istruzione e ne identifica i vari *parametri*,effettuando il *forwarding* verso lo stadio successivo.

$$A \leftarrow Regs[IR_{6..10}]$$
$$B \leftarrow Regs[IR_{11..15}]$$
$$Imm \leftarrow ((IR]_{16})^{16}\#\#IR_{16..31})$$



Nell'ultima espressione, la prima parentesi rappresenta l'estensione del segno.
Esempio: 1000 0000 0000 0000 corrisponde a -32768, dunque se lo vogliamo portare a 32 bit,

dobbiamo replicare il MSB 16 volte, facendolo diventare 1111 1111 1111 1111 1000 000000000000

Execution/Effective address cycle

- **Memory reference**

```
ALUOutput ← A + Imm;
```

- **Register-Register ALU instruction**

```
ALUOutput ← A op B;
```

- **Register-Immediate ALU instruction**

```
ALUOutput ← A op Imm;
```

- **Branch**

```
ALUOutput ← NPC + Imm;
Cond ← (A op 0);
```

Memory reference calcola gli indirizzi, ad esempio [R1]+Imm che restituisce un indirizzo di memoria.
In Branch, l'output dell'ALU è il prossimo indirizzo dato dalla somma di NPC + Imm e andrà nel MUX che lo convoglierà al PC. Cond controlla il MUX per decidere se il salto deve essere effettuato o no.

Memory Access/Branch completion cycle

Memory Access/Branch Completion Cycle

- **Memory reference**

```
LMD ← Mem[ALUOutput] or
Mem[ALUOutput] ← B;
```

- **Branch**

```
if (cond) PC ← ALUOutput else PC ← NPC;
```

LMD sta per Load Memory Data. Il valore preso dalla memoria viene passato in un ulteriore registro(LMD) che poi viene riscritto(Write-back) nel registro di destinazione.

Write back cycle

- **Register-Register ALU instruction**

$$Regs[IR_{16..20}] \leftarrow ALUOutput;$$

- **Register-Immediate ALU instruction**

$$Regs[IR_{11..15}] \leftarrow ALUOutput;$$

- **Load instruction**

$$Regs[IR_{11..15}] \leftarrow LMD;$$

Stadio che non viene utilizzato da tutte le istruzioni(ad esempio la Store).
Il numero del registro, se c'è l'immediate nell'istruzione, risiede nei bit 11..15 dell'istruzione,altrimenti bit 16..20

Behaviour and Optimizations

Dunque tutte le istruzioni, tranne le **store** e i **branch**, impiegano **5** cicli di clock.
Si potrebbe ottimizzare riducendo il **CPI(Clock per instruction):** le operazione dell'**ALU** possono essere completate durante il ciclo di **MEM.**

Basic Pipelining

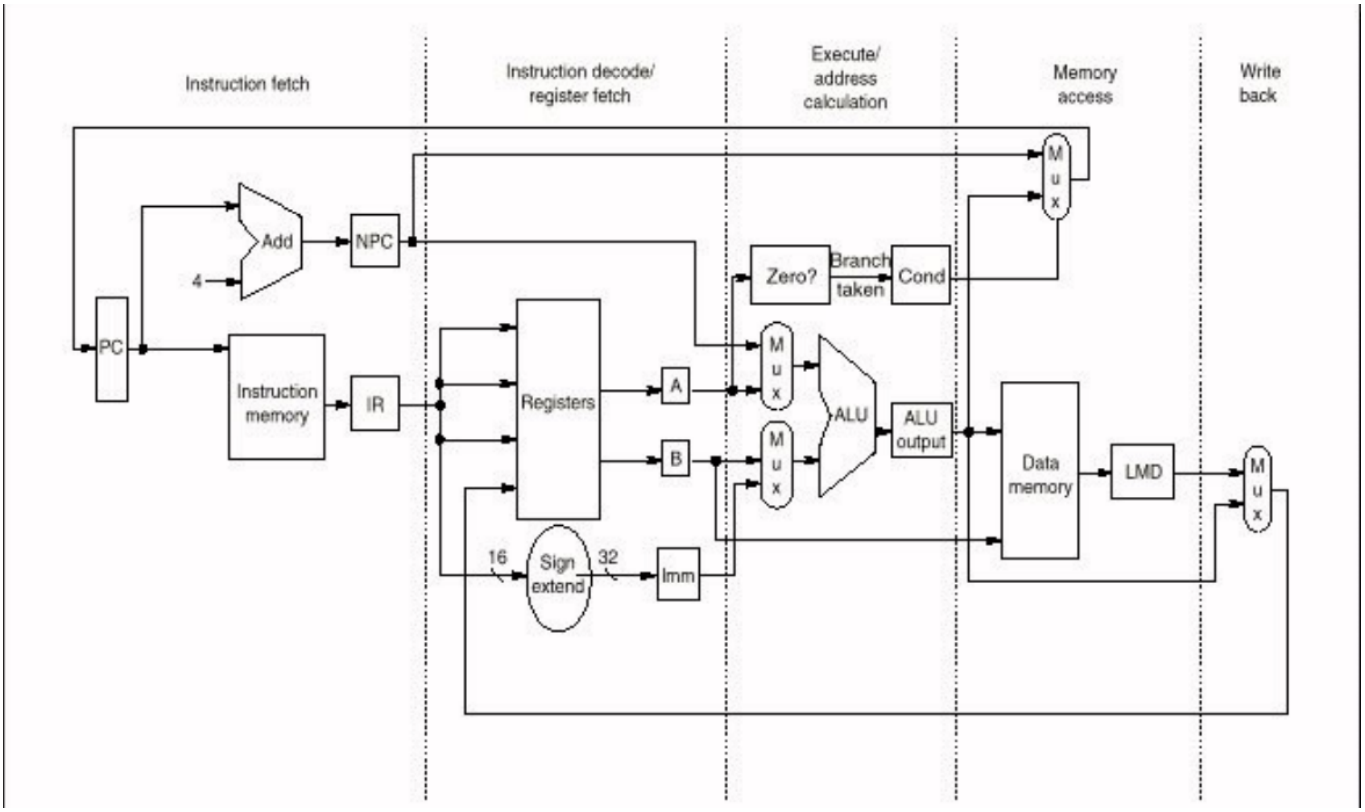
Ad ogni ciclo di clock viene fatta partire una istruzione.
Diverse risorse lavorano su diverse istruzioni allo stesso tempo.
Inoltre, ad ogni ciclo di clock, ogni risorsa deve essere usata per un solo scopo, dunque:

- **istruzioni separate** corrispondono a dati nella **memoria separati**
- il **register file** viene usato in due stadi: **lettura** in **ID** e per **scrittura** in **WB.**
- **PC** deve essere incrementato in **IF**

Performance

Il **pipelining** incrementa ovviamente il **throughput.** Tuttavia l'esecuzione singola delle istruzioni viene resa più lenta, a causa degli overhead(registri di pipeline e **cycle skew**) introdotti dalla pipeline.

Datapath



Esempio

Consider the unpipelined processor, and suppose that

- **The clock cycle is 1 ns**
- **ALU operations and branches require 4 cycles**
- **Memory operations require 5 cycles**
- **The relative frequency of these operations is 40%, 20%, and 40%, respectively.**

The average instruction execution time is

$$\text{Clock cycle} \times \text{average CPI}$$
$$= 1\text{ ns} \times ((0.4 + 0.2) \times 4 + 0.4 \times 5)$$
$$= 1\text{ ns} \times 4.4$$
$$= 4.4\text{ ns}$$

40% ALU, 20% Branch e 40% operazioni di memoria. Con 1ns abbiamo una potenza di 1GHz

Esempio con pipeline

Example (II)

Suppose that moving to the pipelined architecture slows down the clock of the slowest stage by 20%.

The average instruction execution time is therefore 1.2 ns.

The speedup introduced by pipelining is

$$\text{speedup} = 4.4\text{ ns} / 1.2\text{ ns} = 3.7\text{ times}$$

Abbiamo un incremento di 3.7 volte a fronte di una riduzione di performance della CPU(da 1GHz a 0.8 GHz)!!

Pipeline hazards

Sono situazioni che impediscono all'istruzione di venir eseguita durante il suo ciclo di clock.
Ci sono diversi tipi di **hazards(rischi):**

- **data hazards:** un istruzione dipende dai risultati dell'istruzione precedente
- **control hazards:** dipende dal pipelining dei branch e da istruzioni che modificano il **PC**
- **structural hazards:** conflitti di risorse

Stalls

Un modo per risolvere questi **hazards** è quello di stallare la **pipeline**, ovvero bloccare l'istruzione per uno o più cicli.

Structural Hazards

Capita quando una unità di pipeline non può eseguire tutte le sue task assegnate durante un dato ciclo.
Ad esempio:

- la pipeline possiede soltanto un **register-file write port** ma più istruzioni durante lo stesso ciclo scrivono
- più istruzioni accedono alla memoria contemporaneamente(durante IF e durante EX ad esempio).
  Sono dunque legati alla struttura dell'hardware e dunque si risolve introducendo appunto **hardware aggiuntivo**

Example

Load structural hazards happen when two instructions contemporarily try to make a memory access to a single-port memory.

Assume that:

- **40% of instructions make access to memory**
- **the machine with structural hazard has a clock 1.05 times faster than the one without.**

How much faster is the machine without structural hazard?

Solution

For the machine without structural hazard:

Average Instruction Time = CPI × clock cycle time

For the machine with structural hazard:

$$\text{Average Instruction Time} = \text{CPI} \times \text{Clock Cycle Time}$$

$$= (1 + 0.4 \times 1) \times \frac{\text{Clock Cycle Time}_{\text{ideal}}}{1.05}$$

$$= 1.3 \times \text{Clock Cycle Time}_{\text{ideal}}$$

Data hazards

L'interfogliamento dell'esecuzione delle istruzioni, attraverso il pipelining, cambia l'ordine d'accesso di lettura/scrittura agli operandi.

```
ADD     R1, R2, R3
SUB     R4, R1, R5
AND     R6, R1, R7
OR      R8, R1, R9
XOR     R10, R1, R11
```

In ADD, il valore di R1 viene scritto alla fine del ciclo di WB,tuttavia la SUB legge R1 in ID, ovvero due cicli prima che R1 sia disponibile con il valore giusto

Per evitare questi **hazards** si possono usare gli **stalli** oppure implementando delle tecniche di **forwarding(o bypass).**

Interrupt effects

Se succedono degli interrupt durante l'esecuzione di codice, dove sta avvenendo un **data-hazard.** Ciò può causare un comportamento *non deterministico.*

Forwarding

Un hardware dedicato capisce quando un'operazione dell'ALU precedente deve scrivere il valore del registro nell'operazione di ALU corrente. Bypassa dunque lo stadio di **WB.**
Questo forwarding avviene dai vari **pipeline register** agli input dell'ALU.

Pipeline registers

Contengono varie cose.
Ad esempio in ID/EX, ci sono gli operandi(A,B e IMM,NPC etc), in EX/MEM lo Zero Flag, l'output dell'ALU e così' via.

Cause

Si crea quando c'è una dipendenza tra istruzioni abbastanza vicine tra loro da interfogliarsi durante il pipelining.
Può succedere per **register operands** e **memory operands(load/store)**

Stalls per data hazard

Non tutti i data hazard possono essere risolti tramite **forwarding**, ad esempio
LD R1, 0(R2)
SUB R4, R1, R5
R1 diventa disponibile all'inizio del quinto ciclo di clock/fine del quarto ciclo di clock, tuttavia la seconda istruzione ne ha bisogno proprio durante il quarto ciclo di clock.

Implementazione del controllo e Load Interlock Detection

Ad ogni ciclo di clock, lo stadio di **decode** deve capire se l'istruzione attuale soffre di **hazard** rispetto alle istruzioni precedenti.
Se viene scoperto un data hazard, viene attivato il cammino di forwarding giusto oppure stallare.

Introduzione di uno stallo

Durante uno stadio in **ID**, introdurre uno stallo in **EX** si può fare:

- mettendo tutti 0 nel registro ID/EX
- mantenere il valore corrente nel registro IF/ID

Control hazards

Son dovuti ai **branch** che possono cambiare il **PC** dopo che l'istruzioni successiva è già stata presa(**fetch**).
Nel caso dei salti condizionati la decisioni se il **PC** deve essere modificato(**branch taken**) oppure no(**branch untaken**) può essere fatta dopo

Soluzione base

Una soluzione semplice è quella di **stallare(freezing/flushing)** la pipeline appena si trova(**ID stage**) un'istruzione di **branch:**

- decidendo fin da prima se bisogna **prendere** il **branch** oppure no
- **calcolare prima** il nuovo **PC**

Modifiche alla pipeline

La **comparison unit** viene spostata dallo stadio **MEM** allo stadio **ID** e viene aggiunto un **ADDER** sempre nello stadio **ID** per calcolare prima il nuovo **PC** e decidere prima se prendere il salto.

Predict untaken

Si assume che il branch non sia stato preso.

Predict taken

Si assume che il branch sia stato preso.
Bisogna sapere tuttavia a priori dove saltare.

Compilatore

Il compilatore ovviamente può massimizzare il codice per far sì che la CPU faccia la prediction giusta.
Ad esempio il ciclo **for** è più suitable per predict untaken mentre **while** per predict taken.

Delayed branch slot

Si basa sul spostare istruzioni che vengono sempre eseguire dopo il salto, così in caso di **predict untaken**, anche se prendo il branch, non perdo cicli di clock.