

---

ANNO ACCADEMICO 2024/2025

---

## Architettura degli Elaboratori II

---

Teoria

Altair's Notes



**UNIVERSITÀ  
DI TORINO**

---

DIPARTIMENTO DI INFORMATICA

---



<b>CAPITOLO 1</b>	<b>CONCETTI DI BASE</b>	<b>PAGINA 5</b>
1.1	Introduzione Tassonomia delle architetture — 6 • Alcuni concetti fondamentali — 6	5
1.2	Una semplice macchina RISC MIPS - Versione monociclo — 8 • Banco dei registri — 10 • Una semplice Control Unit — 11 • L'esecuzione di un'istruzione — 12	7
1.3	Dal monociclo al multiciclo MIPS - Versione multiciclo — 12 • Control Unit multiciclo — 14 • Macchine a stati finiti e Microprogrammi — 15 • Complex Instruction Set Computer (CISC) — 15 • Reduced Instruction Set Computer (RISC) — 16	12
1.4	Pipeline L'Architettura MIPS Pipelined — 17 • Problemi della Pipeline — 21 • Multiple Pipeline — 23 • Scheduling della Pipeline — 24	17
<b>CAPITOLO 2</b>	<b>INSTRUCTION LEVEL PARALLELISM (ILP)</b>	<b>PAGINA 27</b>
2.1	Introduzione Aumentare la Frequenza del Clock della CPU — 27 • Multiple Issue — 28	27
2.2	ILP Dinamico Scheduling Dinamico, Branch Prediction e Speculazione Hardware — 29 • I Problemi di Fondo — 30 • L'Approccio di Tomasulo — 31 • Multiple Issue con ILP Dinamico — 38	29
2.3	ILP Statico Scheduling Statico e Loop Unrolling — 41 • Multiple Issue Statico — 44 • IA-64: Itanium — 46	41
<b>CAPITOLO 3</b>	<b>CACHING</b>	<b>PAGINA 50</b>
3.1	Funzionamento di Base di una Cache	50
<b>CAPITOLO 4</b>	<b>ARCHITETTURE PARALLELE</b>	<b>PAGINA 54</b>
<b>CAPITOLO 5</b>	<b>QUANTUM COMPUTING</b>	<b>PAGINA 56</b>
<b>CAPITOLO 6</b>	<b>GPU</b>	<b>PAGINA 58</b>



# Premessa

## Licenza

Questi appunti sono rilasciati sotto licenza Creative Commons Attribuzione 4.0 Internazionale (per maggiori informazioni consultare il link: <https://creativecommons.org/licenses/by/4.0/>).



## Formato utilizzato

Box di "Concetto sbagliato":

### Concetto sbagliato 0.1: Testo del concetto sbagliato

Testo contenente il concetto giusto.

Box di "Corollario":

### Corollario 0.0.1 Nome del corollario

Testo del corollario. Per corollario si intende una definizione minore, legata a un'altra definizione.

Box di "Definizione":

### Definizione 0.0.1: Nome delle definizioni

Testo della definizione.

Box di "Domanda":

### Domanda 0.1

Testo della domanda. Le domande sono spesso utilizzate per far riflettere sulle definizioni o sui concetti.

Box di "Esempio":

### Esempio 0.0.1 (Nome dell'esempio)

Testo dell'esempio. Gli esempi sono tratti dalle slides del corso.

**Box di "Note":**

**Note:-**

Testo della nota. Le note sono spesso utilizzate per chiarire concetti o per dare informazioni aggiuntive.

**Box di "Osservazioni":**

**Osservazioni 0.0.1**

Testo delle osservazioni. Le osservazioni sono spesso utilizzate per chiarire concetti o per dare informazioni aggiuntive. A differenza delle note le osservazioni sono più specifiche.



# 1

## Concetti di Base

### 1.1 Introduzione

In questo corso verrà studiata l'architettura interna e il funzionamento dei processori moderni (con riferimento a cache e RAM).

**Note:-**

Lo scopo del corso è quello di spiegare il passaggio al multi-core, subito dopo la "Rivoluzione RISC".

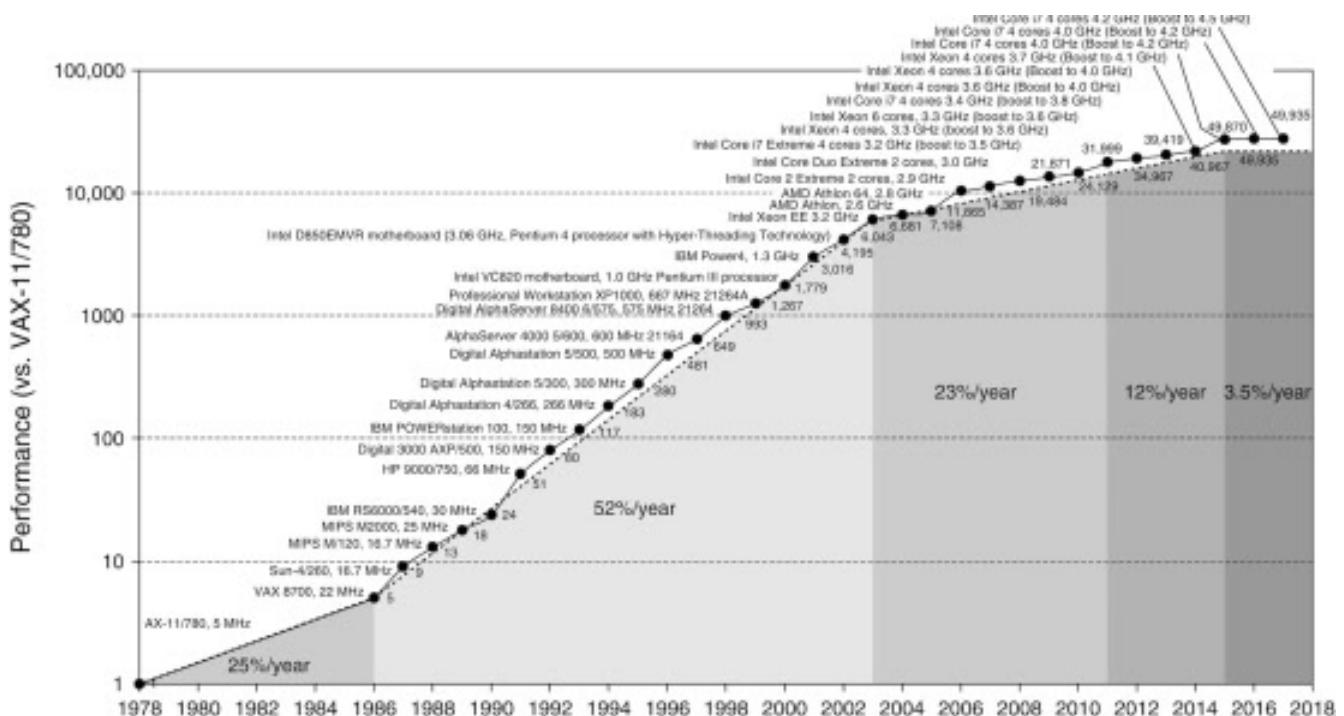


Figure 1.1: Nel 1986 ha inizio la "Rivoluzione RISC", mentre all'inizio degli anni 2000 si inizia a sfruttare l'idea di avere più "core".

### 1.1.1 Tassonomia delle architetture

Il contenuto del corso può essere descritto dalla "Tassonomia di Flynn".

#### Definizione 1.1.1: Tassonomia di Flynn

La Tassonomia di Flynn organizza i vari tipi di processori in base a determinate caratteristiche che verranno approfondite in questo corso.

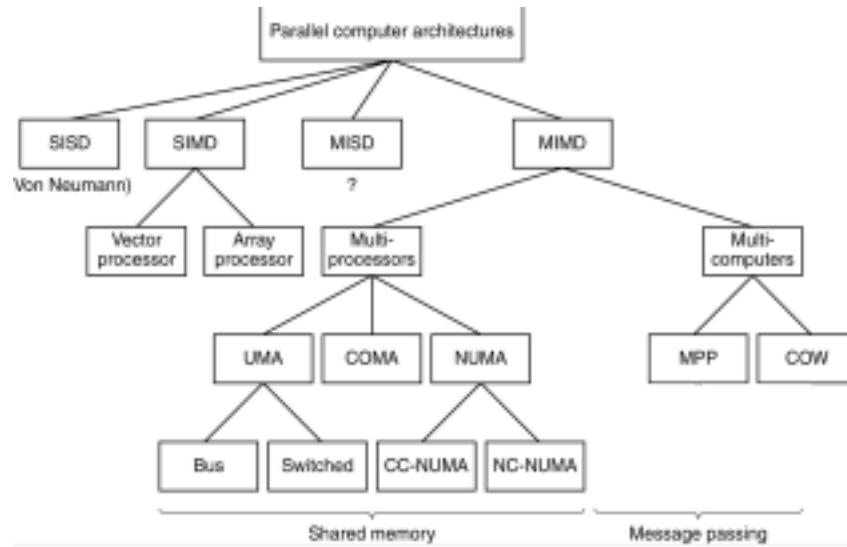


Figure 1.2: La Tassonomia di Flynn

### 1.1.2 Alcuni concetti fondamentali

#### Definizione 1.1.2: Microarchitettura

L'architettura interna di un processore: com'è fatto a partire dal suo datapath.

#### Corollario 1.1.1 Datapath

Il percorso che compiono le istruzioni all'interno del processore per venire eseguite.

#### Note:-

Diversi tipi d'istruzioni percorrono diverse parti del datapath per venire eseguite.

#### Definizione 1.1.3: ISA

L'Instruction Set Architettura (ISA) è l'insieme d'istruzioni macchina di un processore.

#### Note:-

Due processori possono avere lo stesso ISA, ma microarchitetture diverse (e.g. AMD e Intel).

## 1.2 Una semplice macchina RISC

### Domanda 1.1

Qual è la differenza tra un processore a 32 bit e un processore a 64 bit?

**Risposta:** il processore a 64 bit manipola in maniera naturale informazione scritta con 64 bit e il processore a 32 bit manipola in maniera naturale informazione scritta con 32 bit.

**Caratteristiche fondamentali dell'architettura RISC:**

- ⇒ le istruzioni hanno tutte la stessa lunghezza (o a 32 bit o a 64 bit);
- ⇒ le istruzioni sono semplici;
- ⇒ la Control Unit è semplice (poche porte logiche, quindi frequenze di clock più elevate).

### Note:-

Ciò che verrà descritto in questa sezione è una versione semplificata di MIPS, la prima macchina RISC. Si considerano 32 registri a 32 bit e si ignorano le operazioni floating point.

- Una generica istruzione MIPS ha il seguente formato:

op	reg_1	reg_2	reg_dest	shift	funct
6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

- op: tipo di operazione di base richiesta
- reg\_1: registro sorgente 1
- reg\_2: registro sorgente 2
- reg\_dest: registro destinazione
- shift: per le eventuali operazioni di shift sui registri
- funct: specifica la variante dell'operazione op richiesta.

Figure 1.3: Istruzione MIPS

### Definizione 1.2.1: Istruzioni di tipo-R

Le istruzioni di tipo-R usano due registri e restituiscono il risultato a un terzo registro. La convenzione prevede che il campo OP sia 0. L'operazione specifica si trova nel campo func.

### Note:-

Soltamente si usa la lettera D quando si parla di dati interi (DADD, DSUB, etc.), F per i floating point.

### Definizione 1.2.2: Istruzioni di tipo-I

Le istruzioni di tipo-I usano un valore immediato. La convenzione prevede che il campo op sia 8.

### Definizione 1.2.3: LOAD e STORE

La LOAD carica in un registro un valore che si trova in memoria ( $op = 35$ ). La STORE salva in memoria il valore di un registro ( $op = 43$ ).

### Definizione 1.2.4: Salti condizionati (BRANCH)

Salta solo se si verifica una determinata condizione ( $op = 5$ ).

### Definizione 1.2.5: Salti incondizionati (JUMP)

Salta sempre ( $op = 4$ ).

## 1.2.1 MIPS - Versione monociclo

Generalmente i primi due passi di ogni istruzione sono:

1. Usa il Program Counter (PC) per prelevare dalla "memoria d'istruzioni"<sup>1</sup> la prossima istruzione da eseguire;
2. Decodifica l'istruzione e contemporaneamente legge i registri.

#### Note:-

I passi successivi dipendono dal tipo d'istruzione (tutte usano la ALU).

- ⇒ LOAD e STORE accedono alla memoria dati e nel caso di LOAD viene aggiornato un registro;
- ⇒ le istruzioni logico-aritmetiche aggiornano un registro;
- ⇒ le istruzioni di salto possono alterare il valore di PC.

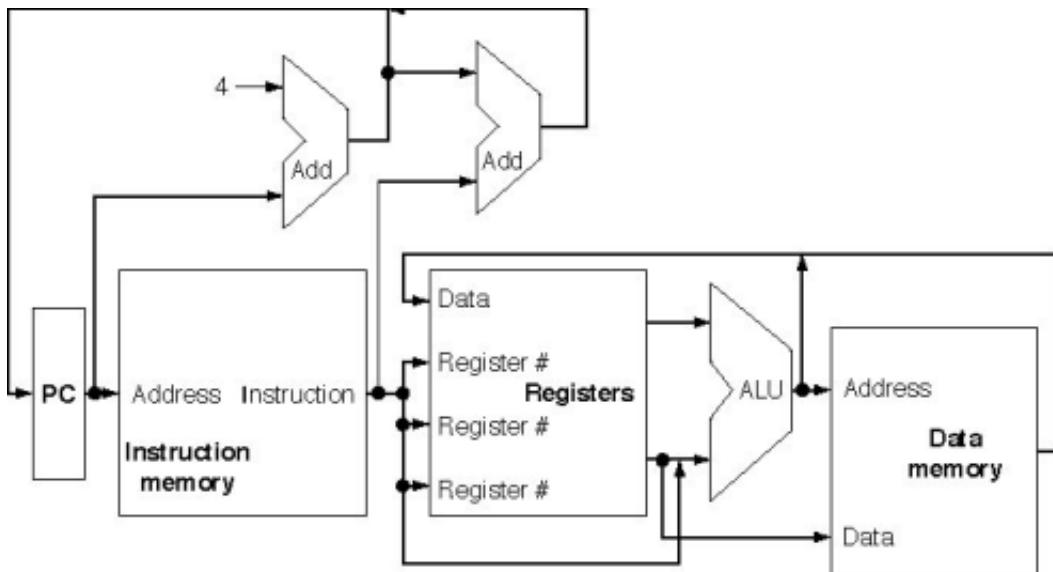


Figure 1.4: Schema ad alto livello del datapath MIPS

#### Note:-

Il fluire delle informazioni nel datapath deve essere controllato da una "Control Unit".

<sup>1</sup>Cache di primo livello.

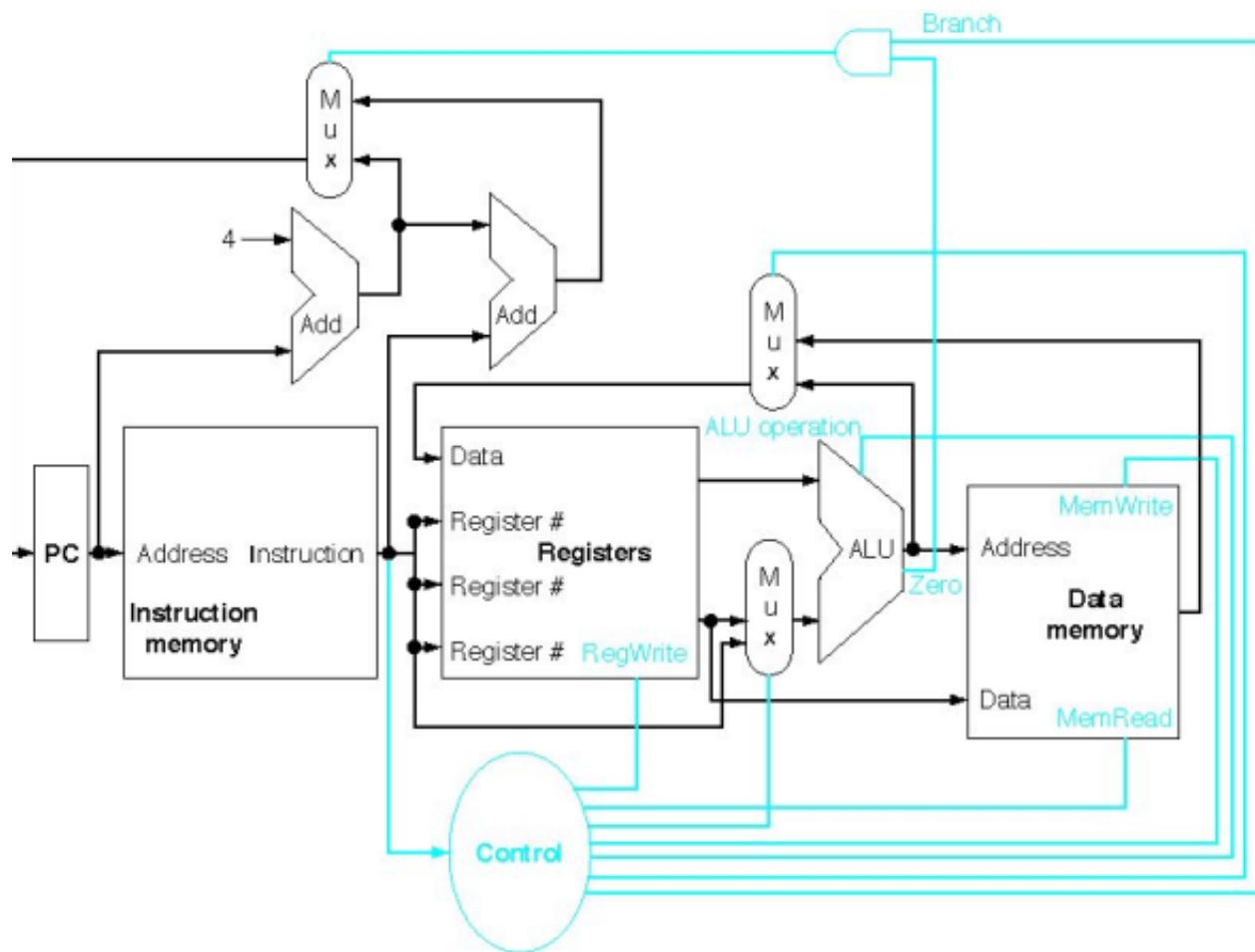


Figure 1.5: Versione modificata del MIPS con una Control Unit

**Note:-**

L'esecuzione di ciascuna istruzione può avvenire in un unico ciclo di clock, purché sia stato adeguatamente dimensionato.

Per capire come funziona il datapath di una macchina monociclo si osserva che esso è composto da due tipi di elementi logici:

- ⇒ gli elementi di *stato*;
- ⇒ gli elementi di tipo *combinatorio*.

**Definizione 1.2.6: Elementi di stato**

Gli elementi di stato sono quelli in grado di memorizzare uno *stato* (e.g. flip flop, registri e memorie). Un elemento di stato possiede almeno 2 ingressi e un'uscita. Gli ingressi richiedono:

- ⇒ il valore da scrivere nell'elemento;
- ⇒ il clock per determinare quando scriverlo.

Il dato presente in uscita è sempre quello scritto in un ciclo di clock precedente.

**Note:-**

Soltanamente esiste un terzo ingresso "di controllo" che stabilisce se l'elemento di stato può effettivamente memorizzare l'input.

**Definizione 1.2.7: Elementi combinatori**

Gli elementi combinatori sono quelli in cui le uscite dipendono solamente dai valori d'ingresso in un dato istante (e.g. ALU e Multiplexer).

## 1.2.2 Banco dei registri

**Definizione 1.2.8: Banco dei registri**

Nelle immagini precedenti i registri della CPU sono rappresentati da un'unità funzionale detta *register file* (o banco dei registri). Essa è un'unità di memoria molto piccola e veloce.

**Note:-**

Si può accedere a ognuno dei 32 registri (da 0 a 31) specificando il suo indirizzo. Ogni registro può essere letto o scritto.

**Operazione di scrittura:** quando il segnale di controllo (RegWrite) è a 1, il valore proveniente dalla ALU o dalla Data Memory e presente in input in *DST data* viene memorizzato nel registro di destinazione specificato da *DST addr*.

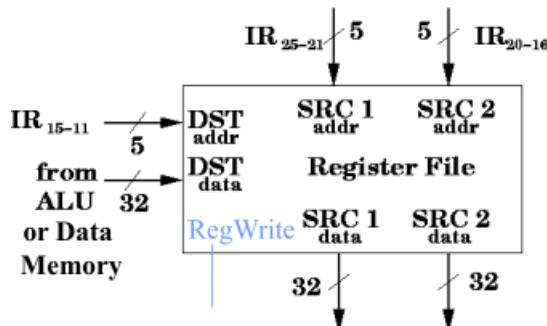


Figure 1.6: Operazione di scrittura

**Operazione di lettura:** le letture sono immediate. In qualunque momento alle uscite *SRC1 data* e *SRC2 data* è presente il contenuto dei registri i cui numeri sono specificati da *SRC1 addr* e *SRC2 addr*.

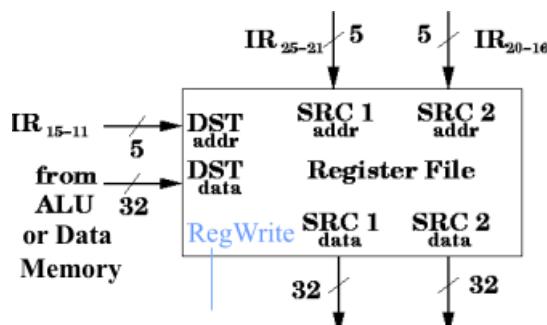


Figure 1.7: Operazione di lettura

### 1.2.3 Una semplice Control Unit

#### Definizione 1.2.9: Control Unit

La Control Unit riceve in input i 6 bit del campo op dell'istruzione e deve generare in output i segnali per comandare:

- ⇒ la scrittura dei registri;
- ⇒ la lettura/scrittura della memoria dati (MemRead/MemWrite);
- ⇒ i Multiplexer che selezionano gli input da usare;
- ⇒ la ALU (ALUOp) che deve eseguire ciascuna operazione aritmetico-logica appropriata per la specifica istruzione in esecuzione.

#### Corollario 1.2.1 Segnale ALUOp

Il segnale ALUOp dipende:

- ⇒ dal tipo d'istruzione in esecuzione, specificato nel campo op;
- ⇒ dalla specifica operazione da eseguire, determinata dal campo func.

#### Esempio 1.2.1 (Istruzioni di tipo-R)

- ⇒ Se  $op == \text{load} \parallel op == \text{store}$  allora la ALU deve eseguire una *somma*;
- ⇒ se  $op == \text{beq}$  allora la ALU deve eseguire una *sottrazione* (per controllare se il risultato è 0);
- ⇒ Se  $op == \text{tipo-R}$  allora è il campo *func* a stabilire l'operazione che deve eseguire la ALU.

#### Definizione 1.2.10: ALU Control

Parte della Control Unit che indica le operazioni da eseguire.

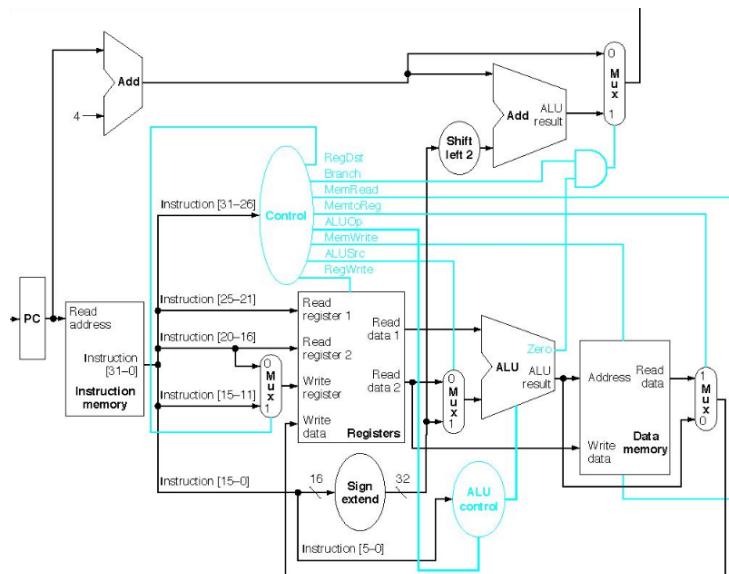


Figure 1.8: Versione modificata del MIPS la ALU Control

### 1.2.4 L'esecuzione di un'istruzione

1. Il contenuto del PC viene utilizzato per indirizzare la instruction memory e produrre in output l'istruzione da eseguire;
2. Il campo *op* dell'istruzione viene mandato in input alla Control Unit, mentre i campi *reg\_1* e *reg\_2* vengono usati per indirizzare il register file;
3. La CU produce in output i nove segnali di controllo relativi a una operazione di tipo-R, e in particolare ALUOp=10, che, dati in input alla ALU control insieme al campo *func* dell'istruzione stabiliscono l'effettiva operazione di tipo-R che deve eseguire la ALU;
4. Nel frattempo, il register file produce in output i valori dei due registri *reg\_1* e *reg\_2*, mentre il segnale ALUSrc=0 stabilisce che il secondo input della ALU deve provenire dal secondo output del register file;
5. La ALU produce in output il risultato della computazione, che attraverso il segnale *MemtoReg* viene presentato in input al register file (Write data).

**Note:-**

Tutti questi passi devono essere eseguiti nello stesso ciclo di clock.

### 1.3 Dal monociclo al multiciclo

Le macchine monociclo sono estremamente inefficienti perché portano a sprecare cicli di clock nel caso di operazioni semplici. Per trasformare una macchina da monociclo in multiciclo si scomponete l'esecuzione di ciascuna istruzione in un insieme di passi ognuno dei quali eseguibile in un singolo ciclo di clock (per cui ogni passo deve richiedere più o meno lo stesso tempo di esecuzione, perché la durata del clock va commisurata alla durata del passo più lungo).

**Note:-**

Le parole "passo", "fase", "stadio" e "stage" sono intercambiabili.

#### 1.3.1 MIPS - Versione multiciclo

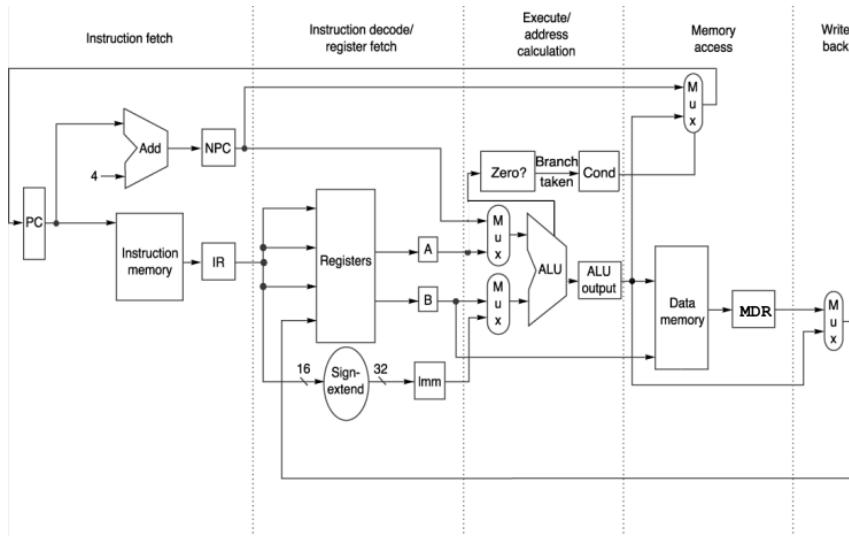


Figure 1.9: Il Datapath suddiviso nelle 5 fasi standard

**Note:-**

Instruction Memory e Data Memory corrispondono alla cache di primo livello (L1).

1. *Instruction Fetch*: il PC indirizza le istruzioni (IR) e contemporaneamente incrementa il PC di 4 (salvato in NPC che verrà instradato alla ALU per eventuali istruzioni di salto). Da notare che la ALU non viene utilizzata quindi si potrebbe pensare di usarla al posto dell'adder, ma quando si introdurrà il concetto di *pipeline* si vedrà che la ALU sarà occupata dalla fase di esecuzione di un'altra istruzione;
2. *Instruction Decode / Register Fetch*: vengono sempre prelevati i valori dei 2 registri di destinazione (anche se l'istruzione non è di tipo-R) e messi in 2 *registri nascosti* A e B. Inoltre il registro Imm viene memorizzato nell'eventualità che sia un'istruzione di tipo-I. Oltre a ciò si memorizza un eventuale salto (usando un adder apposta);
3. *Execute / Address Calculation*: la ALU fa i suoi calcoli in base al tipo d'istruzione;
4. *Memory Access*: nei casi di load e store si accede alla memoria. Per la load si preleva il risultato della fase di esecuzione e viene salvato nel registro MDR. Se si sta eseguendo un'istruzione di tipo-R o di tipo-I questa fase viene saltata;
5. *Write Back*: il risultato viene scritto sul registro di destinazione. Non è necessaria per le store.

**Note:-**

Non necessariamente l'esecuzione di un'istruzione racchiude tutte le fasi. Ogni fase avviene in un ciclo di clock.

**Definizione 1.3.1: Registro "nascosto"**

I risultati di ciascuna fase sono memorizzati da registri "nascosti" in cui viene salvato il risultato dell'output di una fase in modo che diventi l'output della fase successiva. Non sono indirizzabili e servono solo a velocizzare l'esecuzione delle istruzioni.

**Domanda 1.2**

Quali fasi sono coinvolte nell'esecuzione di una "load"?

**Risposta:** tutte.

**Domanda 1.3**

Quali fasi sono coinvolte nell'esecuzione di un'istruzione di tipo-R?

**Risposta:** 4, perché non si deve accedere alla memoria.

**Domanda 1.4**

Quali fasi sono coinvolte nell'esecuzione di una "store"?

**Risposta:** 4, perché non esiste la frase di Write Back.

**Domanda 1.5**

Quali fasi sono coinvolte nell'esecuzione di un'istruzione di salto?

**Risposta:** 3, Instruction Fetch, Instruction Decode e Address Calculation.

**Note:-**

Tutto questo è più efficiente della versione monociclo perché ogni fase usa circa  $\frac{1}{5}$  del ciclo di clock e non sempre si devono attraversare tutte le fasi.

### 1.3.2 Control Unit multiciclo

#### Definizione 1.3.2: Control Unit - multiciclo

Ogni fase sarà descritta da una tabella di verità i cui input sono:

- ⇒ il tipo d'istruzione in esecuzione;
- ⇒ la fase corrente nella sequenza di esecuzione dell'istruzione.

L'output invece sarà:

- ⇒ l'insieme di segnali da asserire in quella fase;
- ⇒ la fase successiva nella sequenza di esecuzione dell'esecuzione.

#### Note:-

Gli output della Control Unit sono una descrizione informale di una macchina di Moore.

#### Corollario 1.3.1 Macchina di Moore

La macchina di Moore è un automa a stati finiti in cui:

- ⇒ a ogni stato è associato un output che dipende solo da quello stato (i segnali da asserire);
- ⇒ la transizione nello stato successivo dipende solo dallo stato corrente e dall'input.

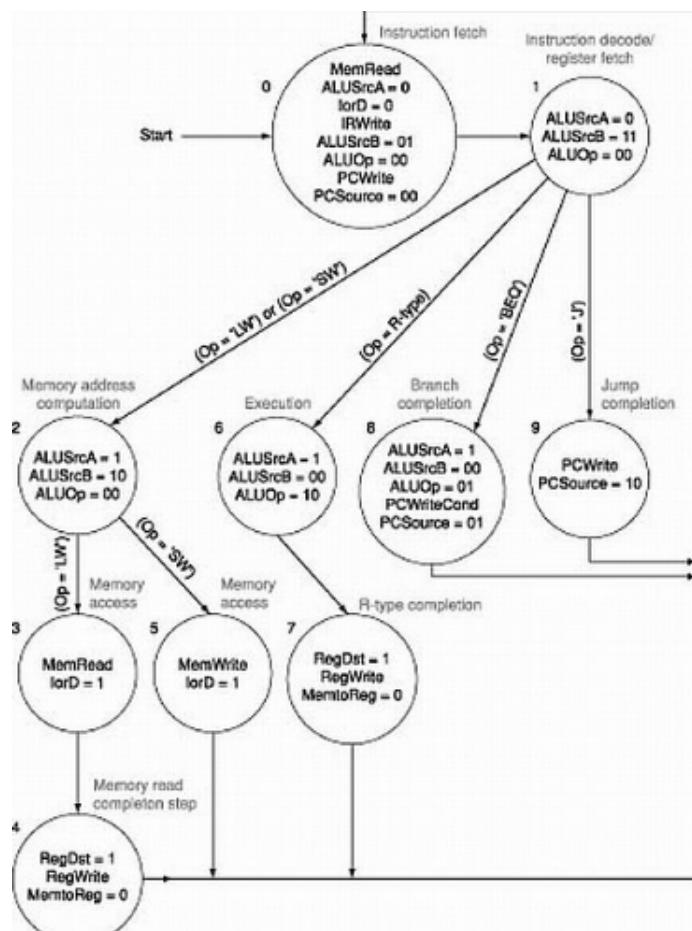


Figure 1.10: Unità di Controllo multiciclo

### 1.3.3 Macchine a stati finiti e Microprogrammi

#### Domanda 1.6

Che differenza c'è tra un datapath controllato da un DFA e uno controllato da un Microprogramma?

**Risposta:** nessuna, è solo un modo di descrivere l'informazione.

#### Definizione 1.3.3: Microprogrammazione

Tecnica per descrivere il funzionamento della Control Unit mediante una rappresentazione simbolica del controllo in forma di microistruzioni indirizzate da un micro Program Counter.

#### Note:-

Un microprogramma è solamente una rappresentazione testuale di una macchina di Moore, ogni microistruzione corrisponde a uno stato e il micro PC rappresenta il registro degli stati.

#### Domanda 1.7

Perché usare una rappresentazione o un'altra?

**Risposta:** è una questione di convenienza in base alla complessità della funzione (dipende sia dalle istruzioni ISA che dalla loro scomposizione).

#### Note:-

Per motivi storici le prime Control Unit furono descritte tramite microprogrammazione.

### 1.3.4 Complex Instruction Set Computer (CISC)

Negli anni '60 e '70 l'instruction set diventa molto grande e, a posteriori, sarà chiamato CISC. In alcuni casi si raggiungevano centinaia d'istruzioni.

#### Definizione 1.3.4: Completa ortogonalità

Ogni argomento di ogni istruzione poteva indirizzare la memoria usando una qualsiasi modalità d'indirizzamento possibile<sup>a</sup>.

<sup>a</sup>Visto in "Storia dell'informatica".

#### Note:-

Questa caratteristica permetteva molta flessibilità, ma al tempo stesso rendeva la Control Unit complicata e molto lenta.

#### Domanda 1.8

Perché si utilizzava un ISA così complesso?

1. All'epoca l'accesso alla RAM erano molto più elevati di quelli dell'accesso alla ROM che conteneva i microprogrammi per le istruzioni;
2. Non esistevano CPU dotate di cache (introdotte solo nel 1968 e divennero di uso comune solo dopo molti anni);
3. Per semplificare il lavoro del programmatore;
4. La RAM era poca e costosa, istruzioni più espansive generano eseguibili più corti;
5. I microprogramma permettevano di aggiungere istruzioni all'instruction set.

### 1.3.5 Reduced Instruction Set Computer (RISC)

Tra la fine degli anni '70 e l'inizio degli anni 80' la concezione delle architetture inizia a mutare. Nel 1980, D. Patterson e C. Sequin progettano una CPU la cui Control Unit non è descritta da un microprogramma e coniano i termini CISC e RISC. Il loro progetto si evolverà nelle architetture SPARC (Scalable Processor ARChitecture), usate dalla SUN Microsystem.

Contemporaneamente, J. Hennessy lavora a un'architettura simile per ottimizzare il *pipeline*: MIPS (Microprocessor *without* Interlocked Pipelines Stages), visto precedentemente.

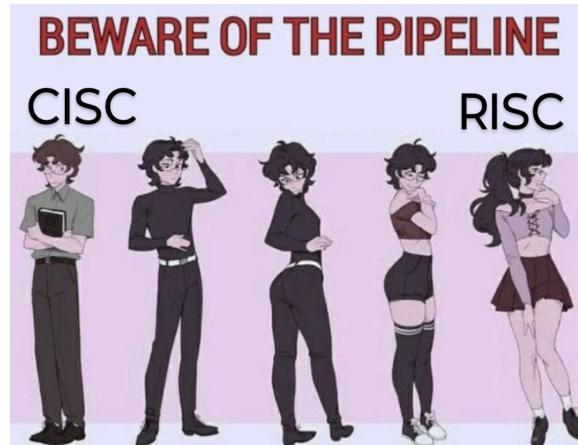


Figure 1.11: The virgin CISC vs. The chad RISC

#### Domanda 1.9

Qual è l'idea alla base delle architetture RISC?

1. La CPU esegue un numero limitato d'istruzioni macchina semplici (che richiedono datapath più corti e una Control Unit semplice) e quindi avere un ciclo di clock più corto;
2. L'accesso alla RAM va limitato il più possibile;
3. Le istruzioni devono principalmente usare registri come argomenti.

#### In dettaglio:

- ⇒ Rinunciare a un sofisticato livello di microcodice;
- ⇒ Definire istruzioni di lunghezza fissa e facili da decodificare;
- ⇒ La memoria RAM deve essere indirizzata solo da operazioni di LOAD e STORE;
- ⇒ Avere molti registri;
- ⇒ Sfruttare la pipeline.

#### Note:-

Questo contribuì al successo dei processori RISC che portò alla scomparsa dei CISC.

#### Definizione 1.3.5: CPI

Il Clock cycles Per Instruction (CPI) è il numero di cicli di clock necessari per eseguire un'istruzione. Indica la velocità alla quale una CPU è in grado di sforzare le istruzioni che esegue.

## 1.4 Pipeline

### 1.4.1 L'Architettura MIPS Pipelined

Passando da monociclo a multiciclo si può aumentare l'efficienza, riducendo lo spreco di tempo. Ma c'è un'altra ragione: si può pensare di sovrapporre, parzialmente, istruzioni consecutive. Ciò non sarebbe possibile in una macchina monociclo.

In una macchina multiciclo mentre una certa fase di un'istruzione occupa una certa parte del datapath le altre porzioni di datapath sono libere.

istr. number	1	2	3	4	5	6	7	8	9
istr. i	IF	ID	EX	MEM	WB				
istr. i+1		IF	ID	EX	MEM	WB			
istr. i+2			IF	ID	EX	MEM	WB		
istr. i+3				IF	ID	EX	MEM	WB	
istr. i+4					IF	ID	EX	MEM	WB

Figure 1.12: Rappresentazione di una pipeline.

**Note:-**

A regime tutte le fasi sono occupate da un'istruzione diversa.

#### Osservazioni 1.4.1 Accorgimenti

- Si deve evitare di usare le stesse risorse per compiere contemporaneamente operazioni diverse. Per questo motivo spesso si usano più ALU;
- Questo è ancora più importante se si introduce il multiple issue, ossia la possibilità di eseguire istruzioni indipendenti di uno stesso programma;
- In alcuni casi ciò è possibile. Per esempio il banco dei registri può essere usato 2 volte in un ciclo di clock in fasi differenti (nell'immagine sopra cerchiati in giallo), uno nella prima parte del ciclo di clock e l'altro nella seconda.

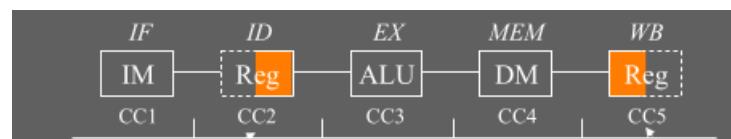


Figure 1.13: Utilizzo del banco dei registri.

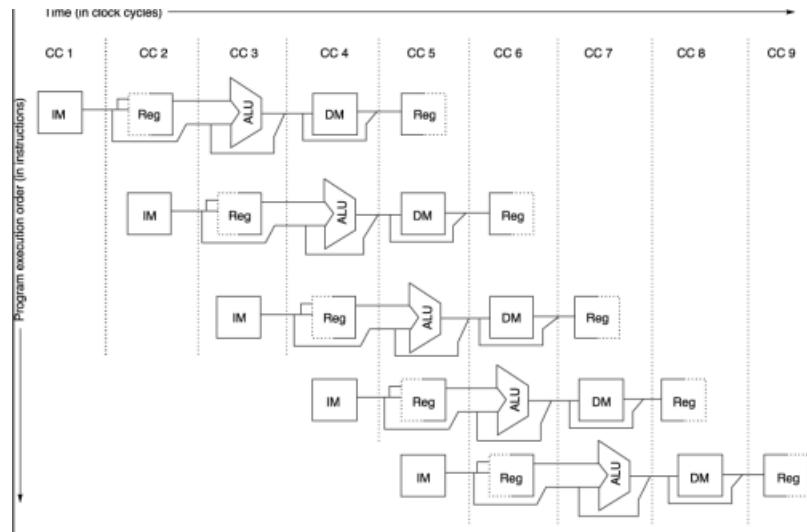


Figure 1.14: MIPS come serie di datapath traslati nel tempo.

Vedendo questo schema si possono fare due osservazioni:

- Si deve ipotizzare l'esistenza di memorie (cache) diverse per dati (DM) e istruzioni (IM) in modo da non avere conflitti tra IF e MEM;
- Il PC deve essere incrementato a ogni ciclo di clock nella fase IF.

#### Definizione 1.4.1: Registri della Pipeline

In un'implementazione reale ogni stage della pipeline è separato e collegato allo stage successivo da opportuni registri della pipeline<sup>a</sup>. Alla fine di un ciclo di clock il risultato viene memorizzato in uno di questi registri.

<sup>a</sup>Un po' come una catena di montaggio.

#### Note:-

Questi registri servono anche a trasportare dati d'istruzioni non adiacenti.

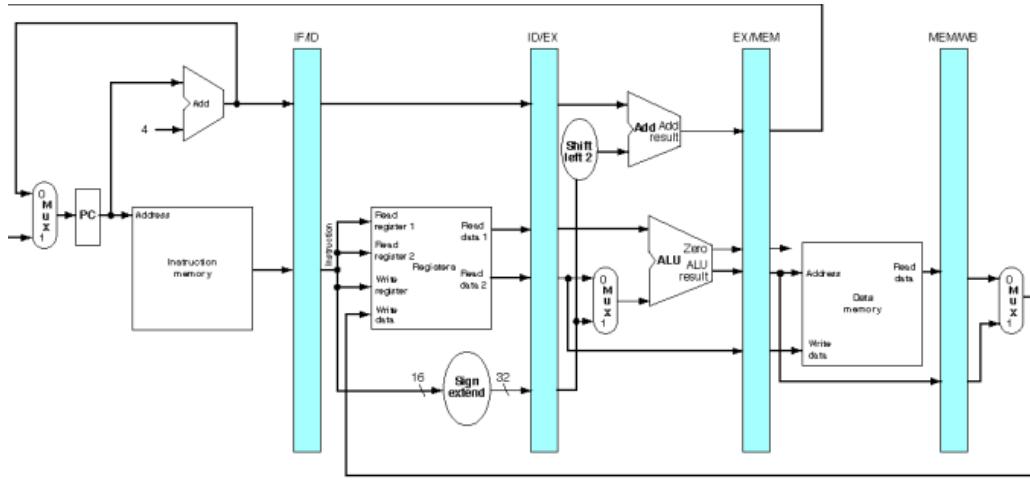


Figure 1.15: Registri della pipeline (in azzurro).

## Simulazione di una LOAD

- **Instruction Fetch:** l'istruzione viene letta dalla Instruction Memory indirizzata dal PC e viene posta nel registro IF/ID. Il PC viene incrementato di 4 e salvato in IF/ID;

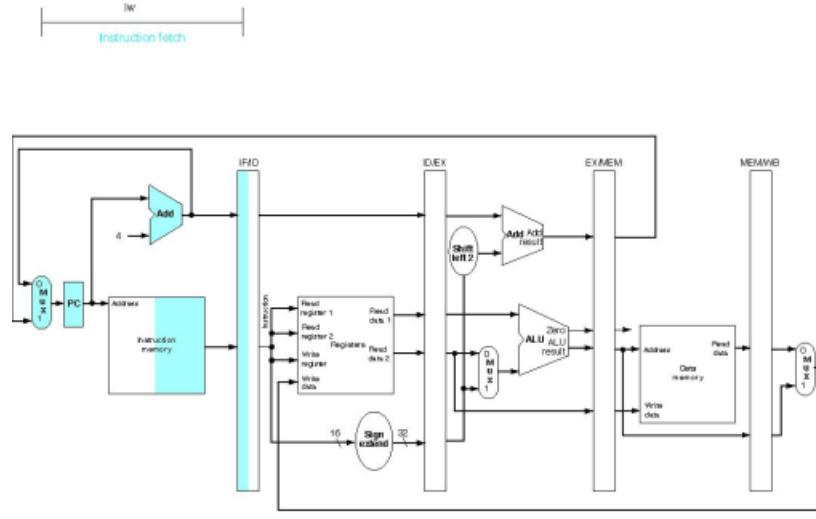


Figure 1.16: IF.

- **Instruction Decode:** il registro IF/ID viene letto per indirizzare il register file. Vengono letti i due registri, indipendentemente dal fatto che se ne userà uno solo oppure entrambi. I 16 bit del valore immediato vengono convertiti a valore 32 bit, e i 3 valori vengono scritti in ID/EX insieme al valore incrementato del PC;

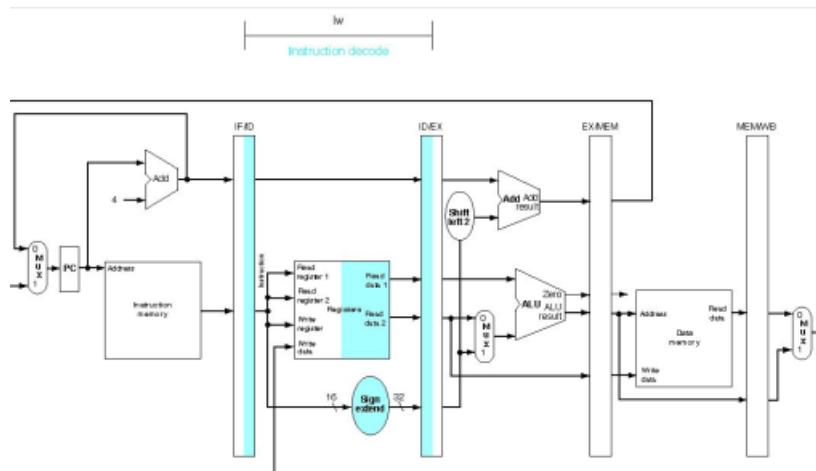


Figure 1.17: ID.

- **EXecute:** la load legge da ID/EX il contenuto del registro 1 e il valore immediato, li somma attraverso l'ALU e scrive il risultato in EX/MEM;

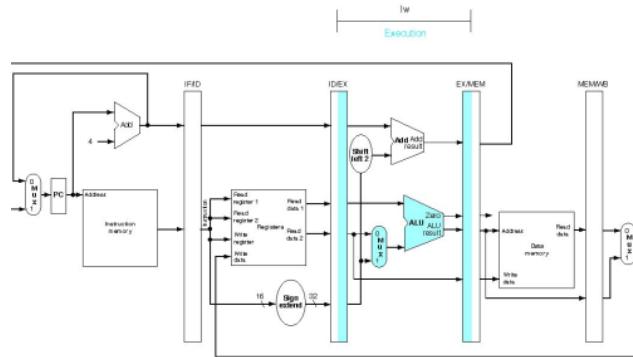


Figure 1.18: EX.

- **MEMory:** avviene l'accesso alla memoria dati, indirizzata dal valore letto nel registro EX/MEM. Il dato letto viene scritto in MEM/WB;

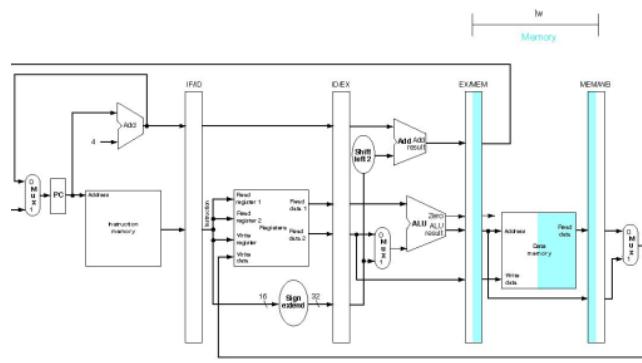


Figure 1.19: MEM.

- **Write Back:** viene scritto il registro di destinazione della load con il valore prelevato dalla memoria dati.

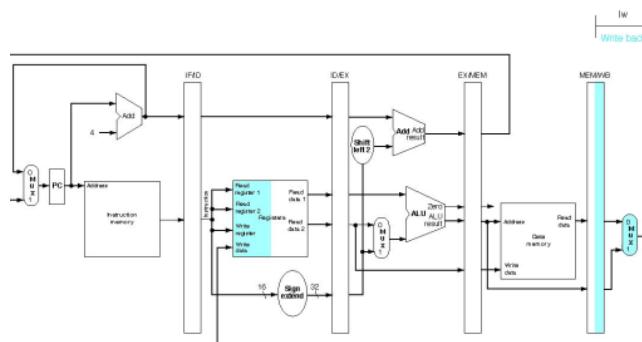


Figure 1.20: WB.

**Domanda 1.10**

Dov'è il numero del registro da modificare?

**Risposta:** è andato perso (sovraffatto). Per evitare ciò bisogna far sì che il numero passi attraverso tutti i registri della pipeline fino a WB.

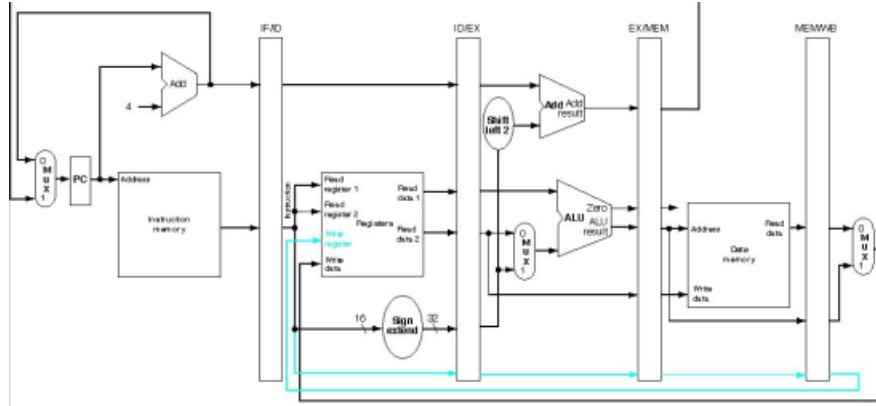


Figure 1.21: Datapath corretto.

**Note:-**

La presenza di una pipeline rende l'esecuzione di un'istruzione leggermente più lenta di una versione senza, ma i programmi girano più velocemente.

### 1.4.2 Problemi della Pipeline

Purtroppo la pipeline presenta alcuni problemi che ne limitano la produttività:

- **Problemi Strutturali:** alcune combinazioni d'istruzioni non possono essere eseguite simultaneamente (e.g. se si ha una sola ALU non la si può usare nello stesso ciclo di clock per fare più cose diverse);
- **Problemi sui Dati:** se un'istruzione A ha bisogno del risultato di un'istruzione B precedente che non ha ancora terminato;
- **Problemi di Controllo:** quando vengono eseguiti salti che possono cambiare il valore di PC.

**Definizione 1.4.2: Stall**

Se si verifica uno di questi problemi è necessario fermare la pipeline (stall). Se un'istruzione generica I genera un problema:

- le istruzioni avviate prima dell'istruzione I possono proseguire fino a essere completate;
- le istruzioni avviate dopo I devono essere fermate, fino a che non viene risolto il problema dell'istruzione I.

### Problemi Strutturali

**Definizione 1.4.3: Problemi Strutturali**

In generale i problemi strutturali si verificano perché, per ragioni di complessità o di costi di produzione, alcune risorse hardware all'interno del datapath non sono duplicate.

**Note:-**

Una cache L1 unica per dati e istruzioni genererebbe molti problemi strutturali, per questo solitamente ve ne sono 2 separate.

## Problemi sui Dati

**Definizione 1.4.4: Problemi sui Dati**

I problemi sui dati sono causati dal fatto che le istruzioni di un programma non sono scollegate tra di loro e alcune devono usare output generati da istruzioni precedenti.

**Corollario 1.4.1 Forwarding**

Il forwarding è una tecnica per bypassare questi problemi. L'idea è quella di rendere disponibile un risultato il prima possibile, per esempio in una ADD il valore di output è generato prima della fase di WB. Quindi si può usare il valore memorizzato in EX/MEM dell'istruzione interessata.

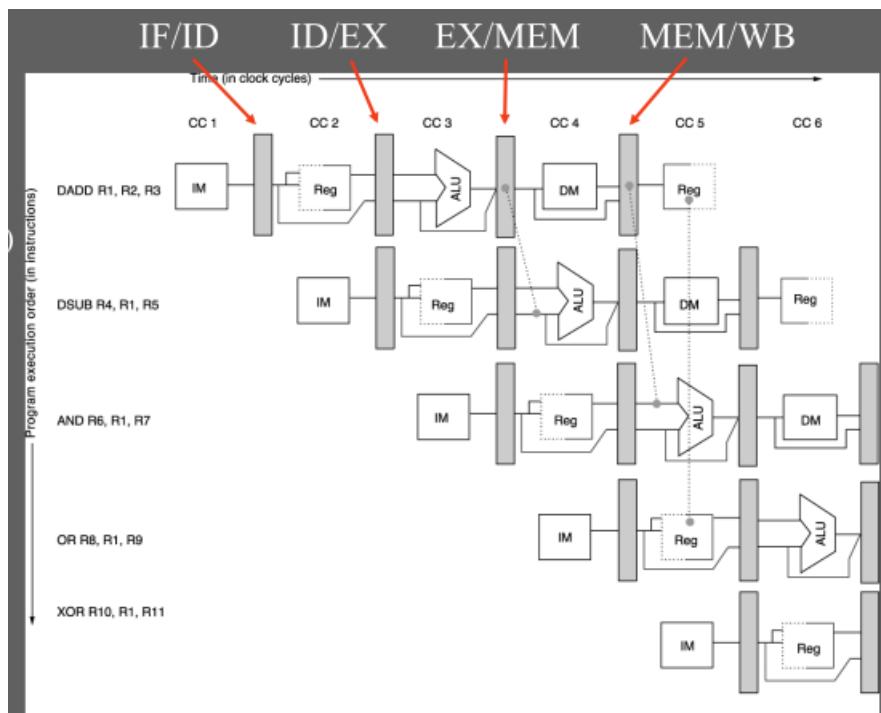


Figure 1.22: Forwarding.

## Problemi di Controllo

**Definizione 1.4.5: Problemi di Controllo**

I problemi di controllo sono dovuti al fatto che, nel caso d'istruzioni di salto, il PC può cambiare.

**Note:-**

Si sprecano cicli di clock quando si salta. Per mitigare questo problema si usa una predizione statica: si dà per scontato che i salti in indietro vengano sempre presi e i salti in avanti no.

**Corollario 1.4.2 Delayed Branch**

Il Delayed Branch consiste nello spostare dopo un branch una istruzione I che è comunque necessario eseguire indipendentemente dall'esito. In questo modo si dà tempo al datapath di valutare se il salto debba essere eseguito o meno.

**Note:-**

Questa tecnica richiede l'intervento del compilatore e la CPU deve essere progettata apposta.

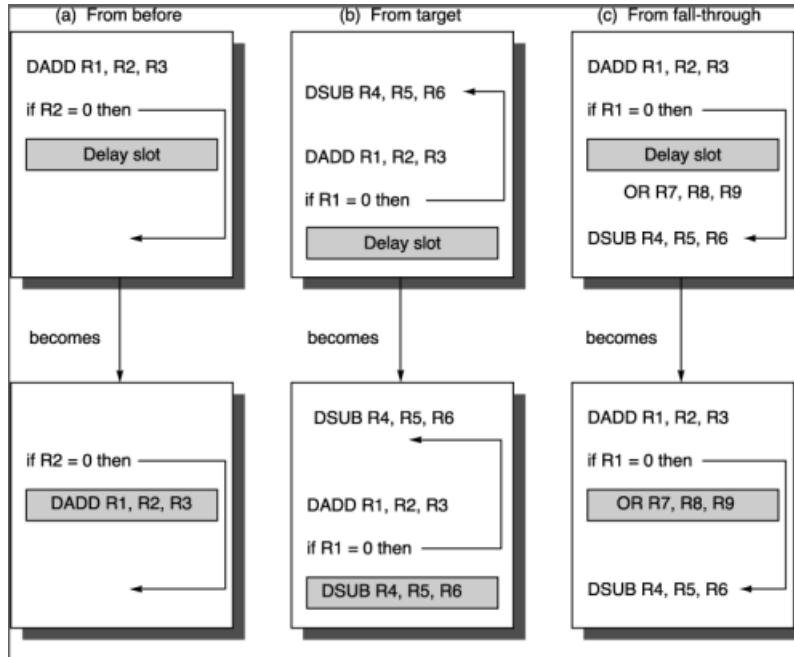


Figure 1.23: Delayed Branch.

**1.4.3 Multiple Pipeline****Domanda 1.11**

Se una pipeline è efficiente, perché non usarne 2?

Bisogna che:

- Si riescano a prelevare due istruzioni dalla instruction memory.
- Le due istruzioni non generino conflitti.
- Le due istruzioni sono indipendenti.

Un'architettura a due pipeline era adottata dal primo pentium (80586). Una pipeline  $u$  poteva eseguire qualsiasi istruzione macchina, mentre la pipeline  $v$  solo istruzioni intere.

Le istruzioni venivano eseguite *in-order*, ossia nell'ordine in cui comparivano nell'eseguibile e alcune regole stabilivano se erano compatibili. Esistevano specifici compilatori per il pentium in grado di generare codice con un alto numero di istruzioni compatibili.

**Note:-**

Ovviamente è possibile avere più di due pipeline in parallelo.

Tuttavia alcune fasi di execute sono molto lunghe rispetto alla semplice addizione tra interi:

- Somma/sottrazione di numeri floating point.
- Moltiplicazione/divisione tra numeri interi.
- Moltiplicazione/divisione tra numeri floating point.

**Note:-**

Si potrebbe adottare un ciclo di clock sufficientemente lungo per le operazioni più lente, ma ciò penalizzerebbe le operazioni più veloci. Conviene suddividere la execute in più fasi.

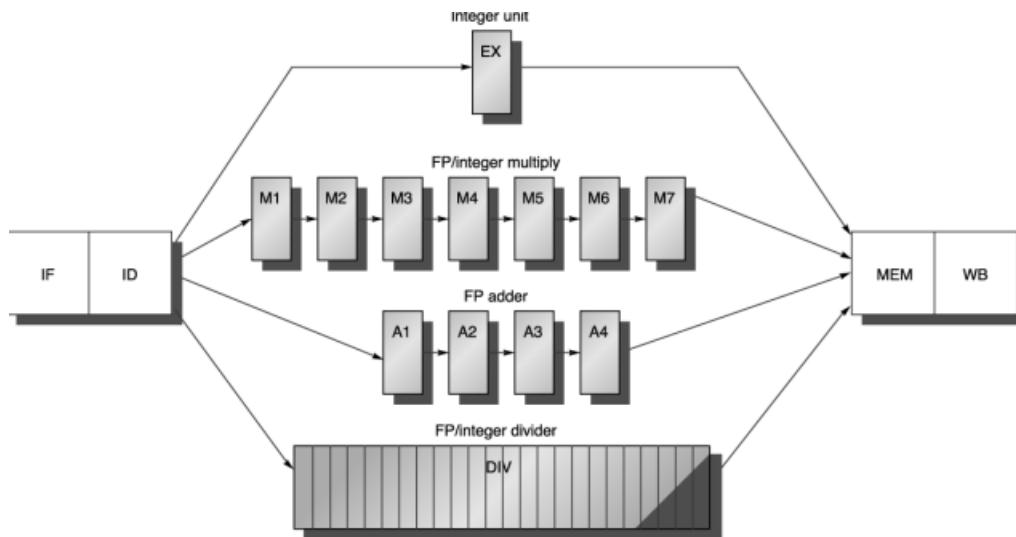


Figure 1.24: Pipeline con più unità funzionali.

In questo schema si vede:

1. Il numero di stage della pipeline dipende dalla complessità dell'operazione.
2. Ci possono essere più istruzioni in fase di execute contemporaneamente.

#### 1.4.4 Scheduling della Pipeline

Fin'ora si è sempre assunta l'esecuzione delle istruzioni in-order, però se si verifica un problema strutturale o sui dati la pipeline viene temporaneamente fermata. Questo tipo di pipeline è detto *schedulata staticamente*.

Le CPU moderne implementano una qualche forma di *scheduling dinamico*, per limitare gli stall sulla pipeline.

**Definizione 1.4.6: IPC**

*Instruction Per Clock (cycle)*: il numero medio di istruzioni eseguite per ciclo di clock.

**Note:-**

Si calcola facendo girare un benchmark di N istruzioni in C cicli di clock.

$$IPC = N/C$$

**Definizione 1.4.7: CPI**

*Clock Per Instruction* ossia il numero medio di cicli di clock necessari per eseguire un'istruzione.

$$CPI = 1/IPC$$



# 2

## Instruction Level Parallelism (ILP)

### 2.1 Introduzione

#### Definizione 2.1.1: Instruction Level Parallelism (ILP)

I processori che tratteremo sono pipelined e superscalari:

1. Eseguono le istruzioni in pipeline.
2. Avviano all'esecuzione in parallelo più istruzioni per ciclo di clock.

#### Per implementare ILP:

1. Deve essere disponibile un numero sufficiente di unità funzionali.
2. Deve essere possibile prelevare dalla instruction memory più istruzioni e dalla data memory più operandi (le cache servono a facilitare questo).
3. Deve essere possibile indirizzare in parallelo più registri della CPU e deve essere possibile leggere/scrivere i registri usati dalle diverse istruzioni in esecuzione nello stesso ciclo di clock.

#### 2.1.1 Aumentare la Frequenza del Clock della CPU

Ciò significa un ciclo di clock più corto con una divisione in un maggiore numero di fasi. Se aumenta il numero di fasi (*profondità*) allora ci saranno più istruzioni in esecuzione contemporaneamente.

##### Note:-

Questa relazione tra numero di fasi e ciclo di clock fu pesantemente sfruttata nel pentium IV in cui si sfioravano i 4 GHz con pipeline di quasi 30 stadi.

#### Tuttavia non è possibile sfruttare all'infinito questa tecnica perché:

1. Maggiore è il numero di fasi, maggiore è la complessità della pipeline e quindi la sua control unit.
2. Frequenze di clock maggiori producono interferenze tra le piste, consumi e conseguenti problemi di dissipazione del calore.

**Definizione 2.1.2: Overclocking**

Il progettista di una CPU non tara il ciclo di clock sulla durata esatta del tempo necessario all'impulso elettrico per attraversare una parte del datapath, ma lo rende un po' più lungo. Su quella differenza gli smanettoni possono "giocare" per aumentare le prestazioni della CPU.

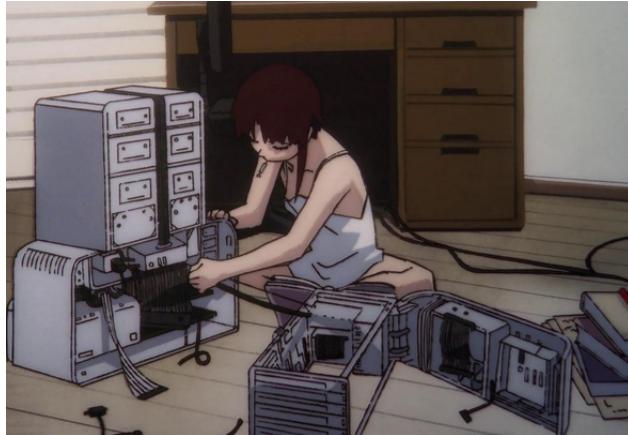


Figure 2.1: Io che faccio overclock del case.

**2.1.2 Multiple Issue****Definizione 2.1.3: Multiple Issue**

Il Multiple Issue consiste nell'aumentare il numero di istruzioni eseguite in parallelo a ogni ciclo di clock. Le architetture che implementano un multiple issue dinamico vengono dette *superscalari*.

**Note:-**

Si può vedere il multiple issue come più pipeline che eseguono istruzioni in parallelo.

**Osservazioni 2.1.1**

- In un'architettura pipelined senza multiple issue, in assenza di stall, il CPI è uguale a 1.
- Introducendo il multiple issue il CPI diventa minore di 1 (più istruzioni per ciclo di clock).
- Tuttavia, nel caso reale, anche implementando multiple issue si ha un CPI maggiore di 1 per via dei problemi strutturali sui dati e sul controllo.

Per implementare il multiple issue è necessario determinare quali e quante istruzioni possono essere avviate all'esecuzione in un dato ciclo di clock. La ricerca è effettuata tra le istruzioni *in attesa* di essere eseguite e ci sono limiti a quante istruzioni possono essere analizzate contemporaneamente. Una volta individuate vengono impacchettate in un *issue packet* e avviate all'esecuzione nello stesso *issue slot* (ciclo di clock). I processori multiple issue si possono dividere in due categorie a seconda di come e quando vengono risolti questi problemi:

- **Multiple Issue statico:** è il compilatore, a livello software, a decidere quali istruzioni mandare in esecuzione in parallelo. Quando il processore preleva dalla Instruction Memory un pacchetto di istruzioni sa già che potrà eseguirle in parallelo. Il numero di istruzioni per pacchetto è stabilito a priori, nella fase di progettazione del processore (adottato principalmente da processori embedded).
- **Multiple Issue dinamico:** è la CPU stessa che analizza, a runtime, le istruzioni e decide quali mandare in esecuzione in parallelo nello stesso ciclo di clock. C'è un limite al numero massimo di istruzioni analizzabile, di solito 3 o 4 (adottato principalmente da processori moderni dei PC).

**Note:-**

ILP dinamico e ILP statico non sono interamente distinti. I processori di una categoria adottano sempre anche qualche tecnica dell'altra.

## 2.2 ILP Dinamico

Per avvicinare una pipeline alle sue prestazioni ideali si utilizzano tre tecniche:

1. *Scheduling dinamico della pipeline*.
2. *Branch prediction*.
3. *Speculazione hardware*.

### 2.2.1 Scheduling Dinamico, Branch Prediction e Speculazione Hardware

Consideriamo questo programma e supponiamo che 100(R2) non si trovi nella cache.

LD      R4, 100(R2)
DADDR10, R4, R8
DSUB R12, R8, R1

L'esecuzione della DADD dipende dalla LD, ma in una pipeline le istruzioni procedono una dopo l'altra. Però la DSUB rimane bloccata anche se non sta aspettando alcun valore dalla LD e dalla DADD

**Definizione 2.2.1: Scheduling dinamico della pipeline**

Nello scheduling dinamico della pipeline l'ordine con cui le istruzioni vengono avviate alla fase EX può essere diverso dall'ordine in cui sono state prelevate dalla memoria di istruzioni.

**Note:-**

Nell'esempio precedente la DADD deve aspettare la LD, ma la DSUB può essere eseguita indipendentemente.

**Corollario 2.2.1 Out-of-order**

Lo scheduling dinamico della pipeline permette sia l'esecuzione out-of-order che il completamento out-of-order. Le istruzioni possono eseguire la fase WB in un ordine diverso da quello in cui compaiono nel programma.

In un sistema che implementa ILP statico è il compilatore ad accorgersi di una situazione di potenziale stallo e genera un codice oggetto in cui la DSUB è posta prima della LD.

**Definizione 2.2.2: Branch Prediction Dinamica**

Per ogni salto condizionato si memorizza l'esito della sua esecuzione. Se lo stesso salto viene eseguito di nuovo si utilizza il risultato precedente per fare una predizione.

**Note:-**

Utile con i cicli, soprattutto se vengono eseguiti molte volte.

**Definizione 2.2.3: Speculazione Hardware**

Estensione della branch prediction: si presuppone che le istruzioni vengano eseguite dopo un salto. Se la predizione è corretta si è fatto del lavoro in anticipo, altrimenti si devono cancellare gli effetti di questa speculazione.

## 2.2.2 I Problemi di Fondo

Le istruzioni dipendono l'una dall'altre.

### Definizione 2.2.4: True Data Dependence

Le istruzioni hanno bisogno di argomenti, ma quegli argomenti possono essere il risultato di altre istruzioni.

### Corollario 2.2.2 Istruzioni Indipendenti

Due istruzioni sono *indipendenti* tra loro se possono essere eseguite simultaneamente e/o in qualsiasi ordine a condizione che ci siano risorse sufficienti.

### Corollario 2.2.3 Istruzioni Dipendenti

Due istruzioni sono *dipendenti* se non possono essere eseguite in modo sovrapposto e quindi devono essere eseguite in ordine.

**Le istruzioni devono riutilizzare i registri.**

Dato che il numero di registri è limitato alcuni registri devono essere riutilizzati terminata la loro funzione.



Figure 2.2: Name Dependence.

#### Note:-

In questo caso non c'è un passaggio di valori tra la PRINT e la LOAD, ma finché la PRINT non è stata completata il registro R7 non può essere riutilizzato.

### Definizione 2.2.5: Name Dependence

Stessi registri vengono utilizzati da istruzione che altrimenti sarebbero indipendenti tra di loro.

Data l'istruzione  $i$  che precede l'istruzione  $j$  si possono avere:

- *Antidipendenza* se  $i$  legge in un registro che  $j$  deve scrivere. L'istruzione  $i$  deve aver tempo di leggere il registro prima che venga sovrascritto da  $j$ , altrimenti legge un valore sbagliato.
- *Dipendenza in output* se  $i$  e  $j$  scrivono nello stesso registro. Il valore finale deve essere quello di  $j$ .

### Esempio 2.2.1



Tra ADD e SUB si ha un antidipendenza, mentre tra ADD e MUL si ha una dipendenza in output.

**Osservazioni 2.2.1**

- La dipendenza sui nomi non è una vera dipendenza perché non ci sono valori trasmessi tra le istruzioni.
- Le istruzioni coinvolte in una dipendenza sui nomi potrebbero essere eseguite in parallelo se il nome del registro usato venisse cambiato.
- La ridenominazione può essere fatta staticamente dal compilatore o dinamicamente dalla CPU mentre esegue le istruzioni.

**Esempio 2.2.2**

DIV	F0, F2, F4
ADD	<b>F6</b> , F0, <b>F8</b>
SUB	<b>F9</b> , F10, F14
MUL	<b>F11</b> , F10, F12

**Note:-**

Una tecnica alternativa è quella di utilizzare registri aggiuntivi *nascosti*.

**2.2.3 L'Approccio di Tomasulo**

Nel 1967, Robert Tomasulo (ricercatore dell'IBM), sviluppò una tecnica per lo scheduling dinamico della pipeline:

- Per minimizzare le dipendenze sui dati tiene traccia di quando gli operandi delle istruzioni sono disponibili, indipendentemente dall'ordine in cui le istruzioni sono entrate nella CPU.
- Per minimizzare le dipendenze sui nomi un insieme di registri interni alla CPU, invisibili a livello ISA viene usato per implementare la ridenominazione dei registri.

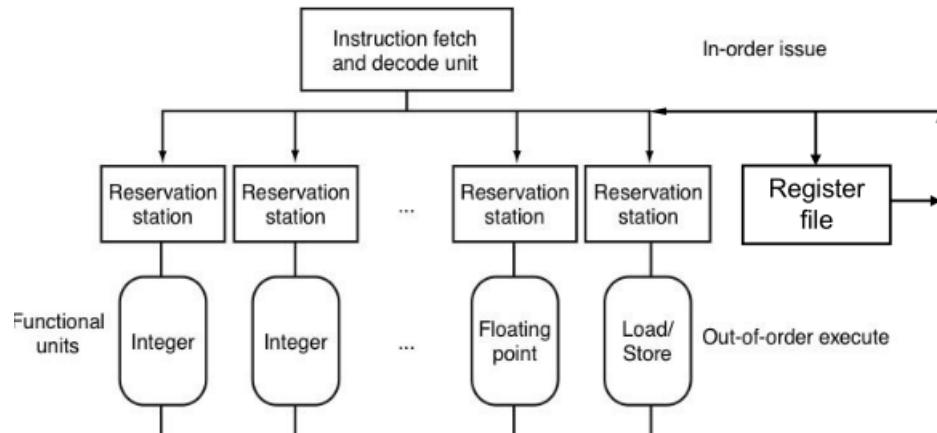


Figure 2.3: Lo schema di Tomasulo.

**Corollario 2.2.4 Reservation Station**

Le *stazioni di prenotazione* servono da stallo per le istruzioni che verranno eseguite quando gli operandi saranno disponibili.

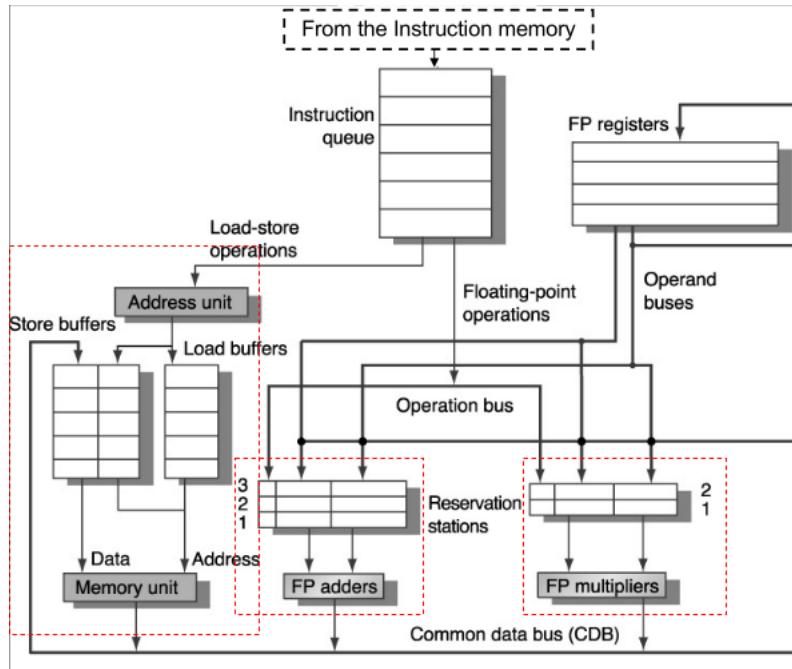


Figure 2.4: Sezione dello schema di Tomasulo che si occupa delle operazioni floating point.

Ogni stazione di prenotazione è fatta da una o più entry. Quando un'entry contiene un'istruzione, per ogni operando dell'istruzione l'entry memorizza anche:

- Se già disponibile: il valore dell'operando stesso.
- Se l'operando non è ancora stato calcolato: l'identificativo della entry della stazione di prenotazione che contiene l'istruzione che dovrà produrre il valore dell'operando mancante.

#### Corollario 2.2.5 Common Data Bus

Il common data bus permette di trasferire in parallelo il risultato prodotto in output da una unità funzionale a tutte le stazioni di prenotazione che lo stanno aspettando e al register file.

Nello schema di Tomasulo l'esecuzione di un'istruzione è divisa in tre macropassi:

1. **Issue:** prelievo, decodifica e inserimento dell'istruzione in una entry della stazione di prenotazione associata all'unità funzionale che dovrà eseguire quell'istruzione. Se non ci sono entry vuote disponibili si ha stall della pipeline. Se gli operandi dell'istruzione sono disponibili nel register file o in qualche altra stazione di prenotazione vengono prelevati e mandati nella entry in cui è stata messa l'istruzione. Per ogni operando che manca, nella entry dell'istruzione viene scritto l'identificativo della entry/stazione di prenotazione in cui è presente l'istruzione che dovrà produrre quell'operando.

La logica di controllo della CPU ha dovuto tenere conto delle istruzioni prelevate in precedenza e già instradate verso le varie stazioni di prenotazione e da cui l'istruzione corrente potrebbe dipendere.

2. **Execute:** l'istruzione si trova in una entry di una stazione di prenotazione. Può essere inoltrata all'unità funzionale associata quando i suoi operandi sono disponibili. Quando un operando viene prodotto come risultato dell'esecuzione di un'altra istruzione tramite il CDB viene inviato al register file e a tutte le entry in cui sono presenti le istruzioni che lo stanno aspettando. Quando tutti gli operandi sono disponibili l'istruzione può essere inoltrata all'unità funzionale che esegue la fase EX. Se più istruzioni sono contemporaneamente pronte a una determinata unità funzionale vengono eseguite una dopo l'altra in pipeline.

Le istruzioni LOAD e STORE, che usano la data memory, vengono inserite nelle entry di specifiche stazioni di prenotazione chiamate load/store buffer. Prima viene calcolato l'indirizzo di memoria a cui operare

e l'indirizzo è memorizzato nella entry della LOAD/STORE relativa. A questo punto le LOAD possono prelevare il dato nella DM che tramite il CDB verrà distribuito in tutte le entry che lo attendono. Le istruzioni di STORE possono dover ancora attendere il valore da depositare in DM.

Per gestire eventuali dipendenze sui dati e sui nomi per gli indirizzi in RAM l'ordine di esecuzione di LOAD e STORE sottostà ad alcuni vincoli aggiuntivi.

3. **Write Result:** quando termina la fase di execute il risultato viene scritto sul CDB e da qui inoltrato al register file e a tutte le entry delle stazioni di prenotazione che stanno attendendo quel risultato. Le istruzioni di STORE scrivono nella memoria dati in questa fase, quando sia l'indirizzo che il dato da mandare in memoria siano disponibili. Le LOAD prelevano il dato dalla RAM e, tramite il CDB, lo scrivono nel registro di destinazione e in qualsiasi entry che lo stia aspettando.

**Note:-**

Per implementare questo meccanismo si ha bisogno di hardware molto sofisticato.

### Osservazioni 2.2.2

- I registri interni di cui sono datte le entry delle stazioni di prenotazione svolgono il compito dei registri temporanei e vengono usati per implementare la rinominazione dei registri.
- Un'istruzione può essere eseguita appena i suoi operandi diventano disponibili.
- Se due istruzioni indipendenti devono usare la stessa unità funzionale non possono essere eseguite in parallelo.

Per far funzionare questo meccanismo ogni entry di ogni stazione di prenotazione è suddivisa in:

- Op: l'operazione da eseguire sugli operandi.
- Qj, Qk: le entry delle stazioni che produrranno il risultato atteso da Op. Uno zero indica che l'operando è già presente in Vj o Vk.
- Vj, Vk: il valore dei due operandi.
- A: presente solo nei load/store buffer, contiene prima il valore immediato per la LOAD o STORE e, dopo che è stato calcolato, l'indirizzo effettivo in RAM in cui leggere/scrivere il dato.
- Busy: indica che la stazione è attualmente in uso.

Ogni registro del file dei registri ha associato un campo:

- Qi: l'entry della stazione di prenotazione che deve produrre l'istruzione il cui risultato dovrà andare in quel registro.
- Se Qi vale 0 non c'è alcuna istruzione che sta calcolando un valore che deve andare in quel registro.

### Osservazioni 2.2.3

Lo schema di Tomasulo ha due caratteristiche fondamentali:

1. L'accesso agli operandi avviene in maniera distribuita:

- Quando più istruzioni stanno aspettando un operando A per passare alla fase EX, non appena A è disponibile tutte le istruzioni possono essere avviate, perché A viene distribuito a tutte mediante il CDB.
- Se si prelevasse A da un registro del register file, ogni unità funzionale dovrebbe accedere sequenzialmente al registro R che contiene A, e nel frattempo nessuna istruzione potrebbe sovrascrivere R.

2. Antidipendenze e dipendenze in output vengono risolte.

**Note:-**

Lo schema di Tomasulo è particolarmente efficace nella gestione di dipendenze sui dati e i nomi nei cicli.

**Definizione 2.2.6: Srotolamento Dinamico**

A regime sono in esecuzione le istruzioni appartenenti a più iterazioni successive del ciclo.

**Note:-**

LOAD e STORE possono essere eseguite indipendentemente se utilizzano registri diversi. Altrimenti:

- Se la STORE è eseguita prima della LOAD si verifica una dipendenza sui dati.
- Se la STORE è eseguita dopo la LOAD si verifica un'antidipendenza.

**Osservazioni 2.2.4**

- Implementare ILP dinamico richiede hardware complesso e costoso insieme a una logica di controllo molto sofisticata.
- Il CDB è implementato con hardware complesso.
- Una pipeline schedulata dinamicamente può fornire prestazioni molto elevate purché i salti vengano predetti in modo accurato.

**Branch Prediction**

Come si è accennato in precedenza si utilizza una predizione statica: se è giusta non si sprecano cicli di clock. In caso di errore la pipeline va svuotata, mediante opportuni segnali alla CU, di tutte le istruzioni nella pipeline successive al branch che non dovevano essere eseguite. La branch prediction dinamica funziona perché spesso le istruzioni nei branch vengono eseguite più volte.

**Definizione 2.2.7: Branch Prediction Buffer**

Una memoria con  $2^n$  entry indirizzate dagli ultimi n bit meno significativi dell'indirizzo di un'istruzione di branch. Ogni entry del buffer memorizza un *bit di predizione* che indica se la volta precedente in cui è stato eseguito quel branch il salto è stato preso o no.

**Note:-**

Il buffer si comporta come una cache di informazioni sulle istruzioni del branch.

**Domanda 2.1**

Cosa succede se il bit di predizione dice che il salto va fatto?

**Domanda 2.2**

Cosa succede se la predizione che diceva di saltare è sbagliata?

**Domanda 2.3**

Le informazioni in una certa entry del buffer sono relative a quello specifico branch che si sta eseguendo?

**Osservazioni 2.2.5**

- Questa forma di predizione può produrre errori.
- Se si considera un ciclo che deve eseguire 10 volte la decima predizione sarà sbagliata.

- Però se il programma rientrerà nel futuro in quello stesso ciclo si avrà di nuovo una predizione sbagliata.

### Definizione 2.2.8: Local 2 bit predictor

Uno *schema di predizione a due bit* consiste nell'aspettare che una predizione sia sbagliata due volte prima di modificarla.

#### Note:-

Può essere implementato con un automa a stati finiti.

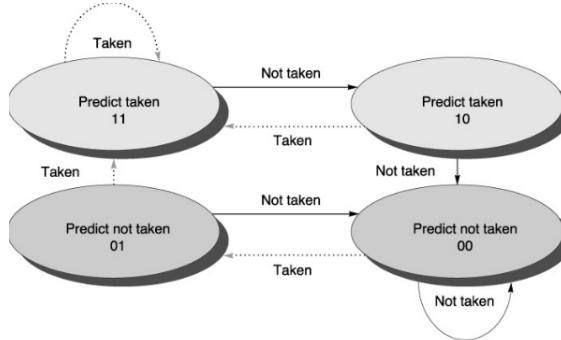


Figure 2.5: Schema di predizione a due bit.

#### Note:-

Questo schema è più complicato da implementare, ma funziona bene se il rapporto tra salti effettuati e non effettuati è molto sbilanciato.

Sistemi a più di 2 bit non aumentano di molto l'efficienza.

### Osservazioni 2.2.6

- L'efficacia di questo schema dipende dal numero di entry nella memoria associativa che contiene i bit di predizione per le istruzioni di branch.
- Solitamente vengono usate cache da 4096 entry, sufficienti per la maggior parte delle istruzioni.

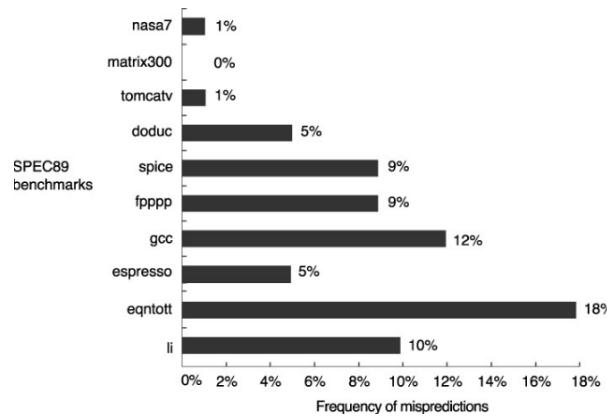


Figure 2.6: Percentuale di errori di predizione per banche prediction buffer a 2 bit e 4096 entry.

**Note:-**

I processori moderni usano varianti di questa tecnica.

**Definizione 2.2.9: Schema a Predittori Correlati**

Uno *schema a predittori correlati* combina i predittori a due bit di due salti consecutivi, combinando così la storia locale di un salto con il comportamento dei salti circostanti.

**Definizione 2.2.10: Schema a torneo**

Uno *schema a torneo* utilizza due predittori diversi per ciascun salto (uno a un bit, l'altro a due bit) e viene usato ogni volta quello che si è comportato meglio la volta precedente.

**Note:-**

Se il predittore dice che il salto va effettuato la CPU non può immediatamente iniziare la fase di fetch dell'istruzione puntata dal salto perché il suo indirizzo non è ancora noto e va calcolato ( $PC + offset$  specificato nell'istruzione di branch).

**Definizione 2.2.11: Branch Prediction Cache**

Un *Branch Prediction Cache* (o Branch Prediction Buffer), per ogni controllo del branch, memorizza anche l'indirizzo a cui trasferire il controllo se il salto viene predetto come eseguito. Il valore  $PC + offset$  viene calcolato e messo nel buffer solo la prima volta che un branch viene eseguito.

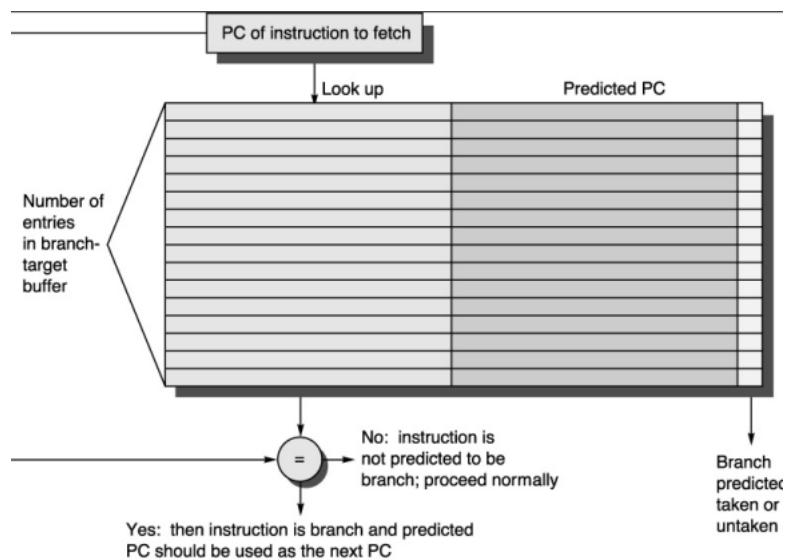


Figure 2.7: Branch Prediction Buffer.

**Speculazione Hardware****Definizione 2.2.12: Speculazione Hardware**

La *speculazione hardware* è la tecnica utilizzata nell'ILP dinamico per gestire e sfruttare vantaggiosamente situazioni in cui un dato non è presente in cache, ma deve essere recuperato dalla memoria primaria.

**Corollario 2.2.6 Istruzioni Speculative**

Le istruzioni controllate dal branch vengono eseguite come se la predizione sul branch fosse corretta.

**Note:-**

La speculazione hardware fa consumare corrente in più.

**Corollario 2.2.7 Unità di Commit**

Un insieme di entry interne alla CPU chiamato *reorder buffer* (ROB) in cui vengono parcheggiate le istruzioni eseguite speculativamente, in attesa di sapere se dovessero essere effettivamente eseguite.

Quando il risultato di un'istruzione eseguita speculativamente è disponibile, viene inserito nel ROB e associato alla entry che contiene l'istruzione che lo ha prodotto. Le entry del ROB fungono da supporto alla ridefinizione dei registri. Quando la CPU sa che un'istruzione nel ROB doveva effettivamente essere eseguita ne esegue il commit: la toglie dal ROB e permette che il registro di destinazione dell'istruzione venga aggiornato. Se invece l'istruzione non doveva essere eseguita viene semplicemente rimossa da ROB.

**Note:-**

L'esecuzione delle istruzioni può avvenire out-of-order, ma il commit deve avvenire nell'ordine in cui le istruzioni sono entrate nella CPU. Questo diminuisce la quantità di lavoro.

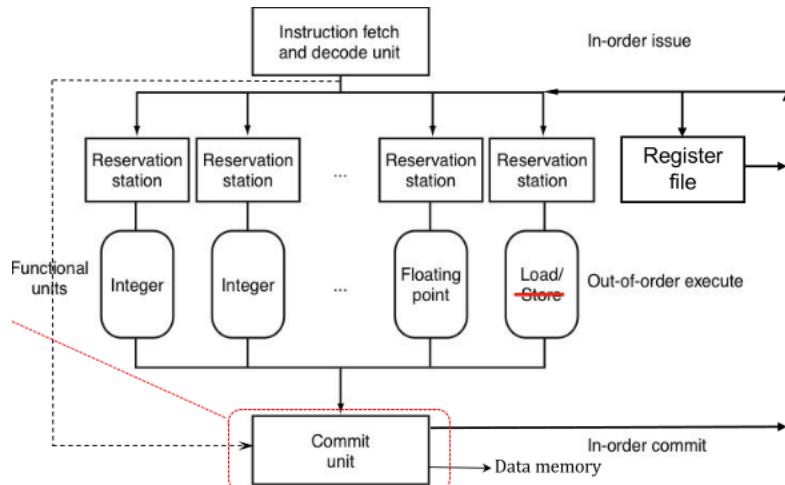


Figure 2.8: Speculazione hardware.

**Il ROB è composto da:**

- *Il tipo di istruzione:*
  - *Branch*, che non produce un risultato.
  - *Store*, che scrive in RAM.
  - *ALU o Load*, che scrivono in un registro.
- *Il campo di destinazione:* ossia il numero del registro o l'indirizzo della locazione di memoria modificati dall'istruzione, se questa riuscirà a passare la fase di commit.
- *Il campo valore:* memorizza temporaneamente il risultato dell'esecuzione dell'istruzione fino al commit.
- *Il campo ready:* indica che l'istruzione ha terminato l'esecuzione e il contenuto del campo valore è valido.

**Con la speculazione si hanno 4 macropassi:**

1. *Issue:* dopo le fasi IF e ID, l'istruzione I viene avviata ad una entry di una stazione di prenotazione. *I viene anche inserita in fondo al ROB, facendo scalare verso la cima tutte le istruzioni già presenti nel ROB.* Se disponibili, gli operandi di I vengono inviati alla entry che contiene I, prelevandoli da uno dei registri, da un'altra stazione di prenotazione, o da una entry del ROB.

*Il numero della entry del ROB che contiene I viene scritto nella stazione di prenotazione di I (così alla fine della fase EX di I, il risultato di I potrà essere inviato a quella entry del ROB).*

2. **Execute:** quando gli operandi sono tutti disponibili l'istruzione I viene inoltrata all'Unità Funzionale corrispondente.
3. **Write Result:** quando il risultato di I è pronto, viene scritto sul CDB, e da qui viene inoltrato ad ogni stazione di prenotazione che lo stava aspettando (ma, notate, non nel register file o in RAM).  
*Il risultato di I viene anche inoltrato verso il ROB e scritto nel campo valore della entry che contiene una copia dell'istruzione I* (il numero della entry del ROB da usare è quello associato ad I durante la fase Issue).
4. **Commit:** quando una istruzione nel ROB raggiunge la cima della coda (perché altre istruzioni sono state inserite in fondo alla coda), il commit può avvenire. Se:
  - L'istruzione non è un branch, il contenuto del campo valore è trasferito nel registro o nella locazione di RAM opportuni. L'istruzione viene rimossa dal ROB.
  - L'istruzione è un branch con predizione sbagliata, tutto il ROB viene svuotato, e la computazione è riavviata dall'istruzione corretta.

#### 2.2.4 Multiple Issue con ILP Dinamico

Se alle tecniche già accennate sull'ILP dinamico si aggiunge il multiple issue si ha la descrizione della maggior parte dei processori moderni (single core). Il numero di istruzioni lanciate in parallelo dal multiple issue è anch'esso dinamico: varia a ogni ciclo di clock.

**Il multiple issue richiede:**

- Un numero sufficiente di unità funzionali per l'esecuzione di più istruzioni in parallelo. Per esempio, più ALU, più unità di moltiplicazione intera / Floating Point, e così via.
- Possibilità di prelevare dalla Instruction Memory più istruzioni, e dalla Data Memory più operandi, per ciclo di clock.
- Possibilità di indirizzare in parallelo più registri della CPU, e deve essere possibile leggere e/o scrivere i registri usati da diverse istruzioni in esecuzione nello stesso ciclo di clock.

**Note:-**

Un Processore con tutte queste caratteristiche è detto processore superscalare.

**Funzionamento di un Processore Superscalare:**

1. Preleva dalla IM (cache di primo livello) N istruzioni per ciclo di clock, dove N è il numero di istruzioni che la IM riesce a fornire in parallelo.
2. Le istruzioni vengono messe in una Instruction queue (IQ) per poter essere analizzate dalla logica della CU che deve controllare la presenza di eventuali dipendenze.
3. Una volta analizzate le istruzioni nella IQ, vengono avviate all'esecuzione, cioè instradate verso le stazioni di prenotazione, alcune delle istruzioni indipendenti, liberando così spazio nella Instruction Queue.
4. Al successivo ciclo di clock viene prelevato dalla IM un altro gruppo N di istruzioni.
5. A regime quindi, la IQ tende riempirsi di istruzioni, e se ad un certo ciclo di clock M istruzioni vengono avviate all'esecuzione, al massimo  $M \leq N$  altre istruzioni possono essere prelevate dalla IM al successivo ciclo di clock.
6. Se la IQ è piena, e a causa delle dipendenze nessuna istruzione è stata inviata alla fase EXECUTE al ciclo precedente, nessuna istruzione potrà essere prelevata dalla IM al ciclo successivo.

**Osservazioni 2.2.7**

- A regime, la CPU deve controllare la presenza di dipendenze tra qualche decina di istruzioni, il che può richiedere migliaia di confronti incrociati, che devono essere fatti in uno o due cicli di clock.
- Se è implementata la speculazione hardware la CPU deve anche essere in grado di eseguire il commit di più istruzioni nel ROB per ciclo di clock, che altrimenti diviene il collo di bottiglia del sistema.

**Domanda 2.4**

Perché ILP dinamico funziona?

- I miss cache non sono prevedibili staticamente, e l'ILP dinamico può parzialmente nasconderli eseguendo altre istruzioni, mentre una istruzione attende dalla RAM il dato mancante in cache.
- I branch non sono prevedibili con accuratezza in modo statico e il BP dinamico e la speculazione aumentano la probabilità di riuscire a sbrigare del lavoro utile in anticipo rispetto al momento in cui si conosce l'esito dell'esecuzione delle istruzioni di branch.
- L'ILP statico funziona bene solo su una specifica architettura. Invece, con l'ILP dinamico i programmi possono essere eseguiti su architetture diverse (purché con ISA compatibili) per numero di unità funzionali, numero di registri rinominabili, numero di stage della pipeline, tipo di predizione dei branch (processori Intel e AMD).

**Limiti Teorici dell'ILP Dinamico**

Tutte le limitazioni, a eccezione delle true data dependence, possono essere eliminate se si ha hardware sufficientemente potente. Si possono fare le seguenti assunzioni:

- *Register Renaming*: la CPU ha un numero infinito di registri rinominabili.
- *Branch Prediction*: perfetta.
- *Memory-Address alias analysis*: tutti gli indirizzi di RAM sono noti, per cui si possono sempre evitare le dipendenze sui nomi in RAM.
- *Multiple Issue*: illimitato.
- *Cache Memory*: non si verificano miss.

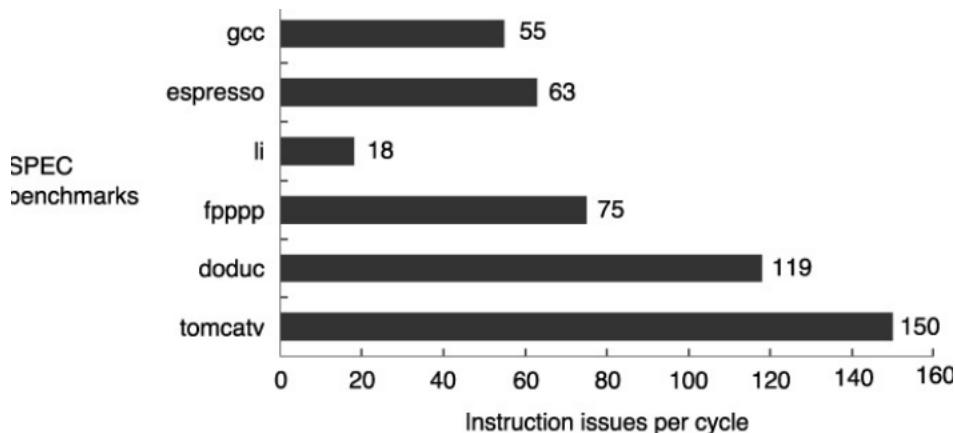


Figure 2.9: Alcuni Benchmark.

Ritornando a un processore più realistico si limita il numero massimo di istruzioni consecutive che possono essere contemporaneamente prese in considerazione per cercare dipendenze sui dati.

Successivamente possiamo introdurre la possibilità di errore nella branch prediction.

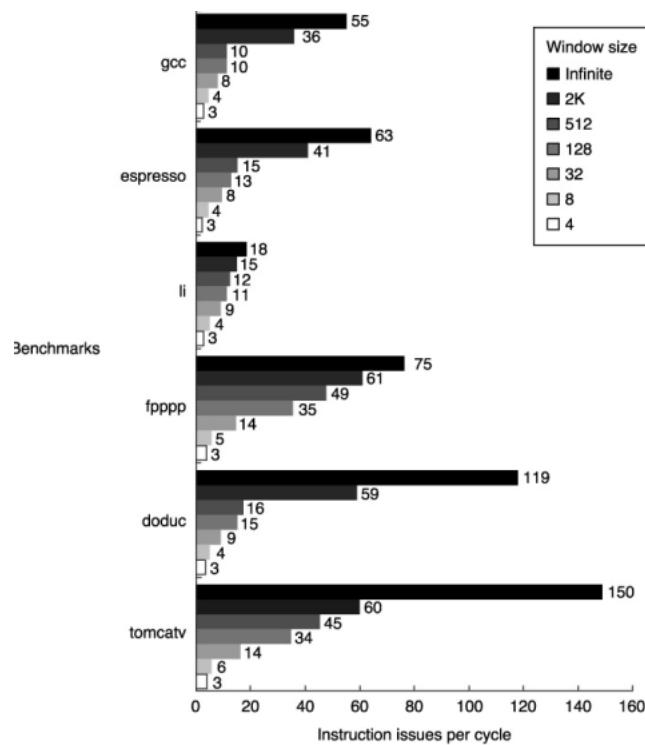


Figure 2.10: Alcuni Benchmark con la limitazione sul multiple issue.

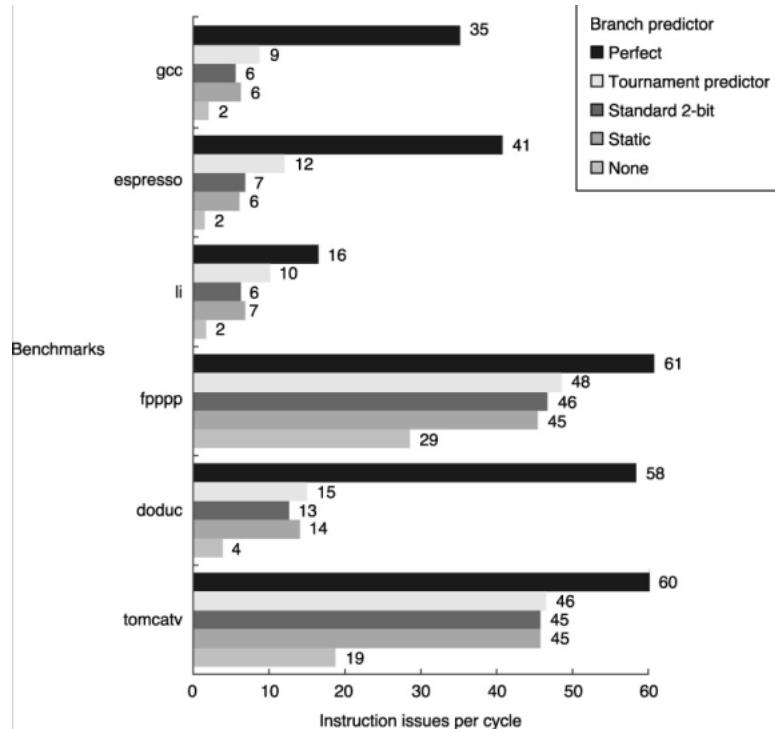


Figure 2.11: Alcuni Benchmark con la limitazione sul multiple issue (con 2k istruzioni) e sulla branch prediction.

**Note:-**

Aggiungendo anche tutte le altre limitazioni le performance calano drasticamente. I progettisti sono ormai convinti che i processori moderni abbiano già da alcuni anni raggiunto il massimo livello possibile di sfruttamento

dell'ILP, e che ulteriori migliorie possano venire solo da avanzamenti tecnologici.

## 2.3 ILP Statico

Il compilatore può riordinare le istruzioni macchina che vengono generate per migliorare lo sfruttamento della pipeline, il tutto ciò prima che il programma vada in esecuzione. L'idea alla base dell'ILP statico è quella di tenere il più possibile occupata la pipeline generando sequenze di istruzioni indipendenti o che non generino stall. Il compilatore separa due istruzioni, A e B (con B che dipende da A), di un numero di cicli di clock sufficiente perché il risultato prodotto da A sia disponibile a B. Per fare ciò il compilatore deve conoscere la CPU su cui sta lavorando.

**Note:-**

ILP statico è importante nei processori embedded perché devono consumare poco.

**Prendiamo in considerazione un compilatore che genera codice per una pipeline a 5 stage:**

- La CPU implementa la tecnica del delayed branch.
- I branch vengono eseguiti ogni due cicli di clock.
- Perché l'istruzione B possa usare il risultato di A occorre attendere:

(A) instruction producing result	(B) instruction using result	latency (clock cycles)
FP ALU op	another FP ALU op	3
FP ALU op	Store double	2
double ALU op	branch	1
Load double	FP ALU op	1 7

### 2.3.1 Scheduling Statico e Loop Unrolling

Consideriamo un for che somma uno scalare a un vettore di 1000 elementi:

```
for ( i = 1000; i > 0; i = i-1 )
    x[i] = x[i] + s;
```

LOOP: LD F0, 0 (R1)	// F0 = array element
FADD F4, F0, F2	// scalar is in F2
SD F4, 0 (R1)	// store result
DADD R1, R1, #-8	// pointer to array is in R1 (DW)
BNE R1, R2, LOOP	// suppose R2 precomputed 8

			<u>clock cycle issued</u>
LOOP:	LD	F0, 0 (R1)	1
	stall		2
	FADD	F4, F0, F2	3
	stall		4
	stall		5
	SD	F4, 0 (R1)	6
	DADD	R1, R1, #-8	7
	stall		8
	BNE	R1, R2, LOOP	9
	stall		10
			<i>No branch prediction</i>

Figure 2.12: Simulazione senza alcuno scheduling e senza branch prediction.

### Definizione 2.3.1: Pipeline Scheduling

Riordino delle istruzioni in modo da minimizzare gli stall della pipeline.

			<u>clock cycle issued</u>
LOOP:	LD	F0, 0 (R1)	1
	DADD	R1, R1, #-8	2 // one stall “filled”
	FADD	F4, F0, F2	3
	stall		4
	BNE	R1, R2, LOOP	5 // delayed branch
	SD	F4, 8 (R1)	6 // altered & interchanged with DADD

Figure 2.13: Simulazione con pipeline scheduling statico.

### Note:-

I compilatori che generano questo tipo di codice sono progettati per architetture specifiche.

### Domanda 2.5

Quali modifiche sono state fatte?

- Con la DADD spostata in seconda posizione, si sono eliminati gli stall (e quindi i ritardi) tra LD e FADD e tra DADD e BNE.
- Spostando la SD dopo la BNE si elimina lo stall di un ciclo di clock dovuto alla BNE e si riduce di uno lo stall tra FADD e SD.
- Siccome SD e DADD sono state scambiate, l'offset nella SD deve ora essere 8 anziché 0.
- Questa forma di scheduling è possibile solo se il compilatore conosce la latenza in cicli di clock di ogni istruzione che genera, ossia se conosce i dettagli interni dell'architettura per cui genera codice.

### Definizione 2.3.2: Loop Unrolling Statico

Le istruzioni di più iterazioni consecutive vengono fuse in un unico macrociclo gestito da un unico controllo di terminazione.

```

LD    F0, 0 (R1)
FADD F4, F0, F2
SD    F4, 0 (R1)          // drop DADD & BNE
LD    F6, -8 (R1)
FADD F8, F6, F2
SD    F8, -8 (R1)          // drop DADD & BNE
LD    F10, -16 (R1)
FADD F12, F10, F2
SD    F12, -16 (R1)        // drop DADD & BNE
LD    F14, -24 (R1)
FADD F16, F14, F2
SD    F16, -24 (R1)
DADD R1, R1, #-32
BNE   R1, R2, LOOP

```

Figure 2.14: Loop Unrolling Statico con 4 iterazioni del ciclo.

**Note:-**

Si possono unire Scheduling Statico e Loop Unrolling.

LD F0, 0 (R1)	ecco come ora le istruzioni
LD F6, -8 (R1)	srotolate (loop unrolling)
LD F10, -16 (R1)	possono essere riordinate
LD F14, -24 (R1)	(pipeline scheduling) per
FADD F4, F0, F2	eliminare tutti gli stall presenti
FADD F8, F6, F2	
FADD F12, F10, F2	
FADD F16, F14, F2	
SD F4, 0 (R1)	lo spostamento di
SD F8, -8 (R1)	queste
DADD R1, R1, #-32	istruzioni serve per
SD F12, 16 (R1)	eliminare
BNE R1, R2, LOOP	gli ultimi due stall
SD F16, 8 (R1)	// 8 - 32 = -24 (?)

Figure 2.15: Loop Unrolling e Scheduling Statico.

**Note:-**

Si ha un miglioramento enorme, ma con scarsa portabilità.

Inoltre vi sono diversi problemi:

- Quanti nomi di registri diversi si possono usare nel loop unrolling? (ossia quanti ne sono disponibili a livello ISA?) Se si esauriscono tutti i registri, può essere necessario usare la RAM come deposito temporaneo, il che è molto inefficiente.
- Come si possono gestire i cicli con un numero di iterazioni non noto a priori (per esempio, un while-do o un repeat-until)?
- Il loop-unrolling genera codice più lungo di quello originale. Quindi aumenta la probabilità di cache miss nella Instruction Memory, eliminando parte dei vantaggi dell'unrolling.

### 2.3.2 Multiple Issue Statico

Consideriamo un MIPS in grado di eseguire un'operazione intera e una floating point.

loop:	int. instruction	FP instruction	clock cycle
	LD F0, 0 (R1)		1
	LD F6, -8 (R1)		2
	LD F10, -16 (R1)	FADD F4, F0, F2	3
	LD F14, -24 (R1)	FADD F8, F6, F2	4
	LD F18, -32 (R1)	FADD F12, F10, F2	5
	SD F4, 0 (R1)	FADD F16, F14, F2	6
	SD F8, -8 (R1)	FADD F20, F18, F2	7
	SD F12, -16 (R1)		8
	DADD R1, R1, #-40		9
	SD F16, 16 (R1)		10
	BNE R1, R2, Loop		11
	SD F20, 8 (R1)		12

Figure 2.16: Esecuzione con multiple issue statico.

#### Definizione 2.3.3: Multiple Issue Statico

Nel Multiple Issue statico il compilatore produce già dei pacchetti contenenti un numero prefissato di istruzioni indipendenti. Però alcune istruzioni possono essere no-op perché il compilatore non è riuscito a trovare abbastanza istruzioni indipendenti.

#### Corollario 2.3.1 Issue Packets

I pacchetti generati dal compilatore riordinano le istruzioni, srotolando i cicli e minimizzando le dipendenze.

#### Note:-

Questo riduce il lavoro della CPU.

Questo approccio prende il nome di VLIW (Very Long Instruction Word).

pack et / n. clock	memory ref. 1	memory ref. 2	FP op. 1	FP op. 2	integer / branch op.
1	LD F0,0(R1)	LD F6,-8(R1)	<i>no-op</i>	<i>no-op</i>	<i>no-op</i>
2	LD F10,-16(R1)	LD F14,-24(R1)	<i>no-op</i>	<i>no-op</i>	<i>no-op</i>
3	LD F18,-32(R1)	LD F22,-40(R1)	FADD F4,F0,F2	FADD F8,F6,F2	<i>no-op</i>
4	LD F26,-48(R1)	<i>no-op</i>	FADD F12,F10,F2	FADD F16,F14,F2	<i>no-op</i>
5	<i>no-op</i>	<i>no-op</i>	FADD F20,F18,F2	FADD F24,F22,F2	<i>no-op</i>
6	SD F4,0(R1)	SD F8,-8(R1)	FADD F28,F26,F2	<i>no-op</i>	<i>no-op</i>
7	SD F12,-16(R1)	SD F16,-24(R1)	<i>no-op</i>	<i>no-op</i>	DADD R1,R1,#-56
8	SD F20,24(R1)	SD F24,16(R1)	<i>no-op</i>	<i>no-op</i>	<i>no-op</i>
9	SD F28,8(R1)	<i>no-op</i>	<i>no-op</i>	<i>no-op</i>	BNE R1,R2, <sub>26</sub> ,Loop

Figure 2.17: Esecuzione con approccio VLIW.

**Osservazioni 2.3.1**

- Non tutti gli slot di ogni pacchetto sono occupati, e vengono riempiti da no-op. A seconda del codice e del tipo di architettura, il compilatore non riesce sempre a sfruttare tutti gli slot di ogni pacchetto. Nel nostro esempio circa il 40% degli slot non viene sfruttato.
- Ci vogliono 9 cicli di clock per eseguire 7 iterazioni del for, con una media di  $9/7 = 1.29$  cicli di clock per eseguire una iterazione del for originale.
- Una macchina con architettura diversa (per esempio, diverse U.F. o diverse latenze per istruzione), fornirebbe prestazioni diverse eseguendo lo stesso codice.

**Tecniche più sofisticate:**

- Static branch prediction:** assume che i branch abbiano sempre un certo tipo di comportamento, o analizza il codice per cercare di dedurne il comportamento.
- Loop Level Parallelism:** tecniche per evidenziare il parallelismo in iterazioni successive di un loop, quando i vari cicli non sono indipendenti fra loro.
- Symbolic loop unrolling:** il loop non viene “srotolato”, ma si cerca di mettere in ogni pacchetto istruzioni fra loro indipendenti, anche appartenenti a cicli diversi.
- Global code scheduling:** cerca di compattare codice proveniente da diversi basic block (quindi istruzioni separate da varie istruzioni condizionali, inclusi i cicli) in sequenze di istruzioni indipendenti.

**Istruzioni Predicative****Definizione 2.3.4: Istruzioni Predicative**

Le *Istruzioni Predicative* (o condizionali) servono per eliminare alcune istruzioni non facilmente prevedibili. Quindi gli if non vengono presi in considerazione perché eseguiti una volta sola. Se la condizione è vera il resto dell’istruzione viene eseguito, ma se è falsa l’istruzione diventa una no-op.

**Note:-**

Questa tecnica viene implementata in qualche variante, statica o dinamica, nei processori moderni.

### Osservazioni 2.3.2

- Le semplici move condizionali non permettono però di eliminare i branch che controllano blocchi di istruzioni diverse dalle move.
- Per questo, alcune architetture supportano la full predication: tutte le istruzioni possono essere controllate da un predicato.
- Per funzionare correttamente, queste architetture hanno bisogno di opportuni registri predicativi (ciascuno formato da un solo bit), che memorizzano il risultato di test effettuati da istruzioni precedenti, in modo che il valore possa essere usato per controllare l'esecuzione di istruzioni successive (che quindi divengono predicative).
- In generale, le istruzioni predicative sono particolarmente utili per implementare piccoli if-then-else eliminando branch difficilmente predicibili e quindi per semplificare le tecniche di ILP statico più sofisticate.

Tuttavia queste istruzioni sono limitate da:

- Le istruzioni predicative poi annullate hanno comunque utilizzato risorse della CPU, limitando l'esecuzione di altre istruzioni.
- Combinazioni complesse di branch non sono facilmente convertibili in istruzioni condizionali.
- Le istruzioni condizionali possono essere più lente delle corrispondenti non condizionali.

**Note:-**

Molti processori moderni basati sull'ILP dinamico, inclusi i core i3-5-7 della Intel supportano la move condizionale.

### 2.3.3 IA-64: Itanium

Alla fine degli anni '90, in previsione di raggiungere il limite delle istruzioni a 32 bit, l'intel si è mossa in due direzioni:

- Sviluppo dell'ISA e della microarchitettura *Intel 64*.
- Dal 2000 insieme alla Hewlett-Packard, sviluppò un ISA e una microarchitettura completamente nuova, nota come *IA-64*.

**Definizione 2.3.5: Itanium**

Gli Itanium sono i primi processore della Intel completamente RISC, basati su IA-64, adottano VLIW (rinominato dall'Intel in EPIC).

**Caratteristiche dell'Itanium:**

- 128 registri general-purpose a 64 bit.
- 128 registri floating point a 82 bit.
- 64 registri predicativi da 1 bit ciascuno.
- 128 registri speciali usati per vari scopi.
- 3 livelli di cache.

**Note:-**

Dei 128 registri general-purpose i primi 32 sono sempre disponibili, mentre gli altri sono usati come stack di registri.

Exec. unit	Instr. type	description	example instr.
I-unit	A	Integer ALU	add, sub, and, or, compare
	I	non-ALU integer	int./ multimedia shift, bit test, moves
M-unit	A	Integer ALU	add, sub, and, or, compare
	M	Memory access	load / store for integer and FP registers
F-unit	F	Floating Point	floating point Instructions
B-unit	B	Branches	cond. branches, call, loop branches
L+X	L+X	Extended	extended immediate, stop, no-ops

Figure 2.18: Unità funzionali dell'Itanium.

**Corollario 2.3.2 Instruction Group**

Il compilatore di una architettura IA-64 organizza le istruzioni macchina del programma che sta compilando come sequenza di instruction groups.

**Osservazioni 2.3.3**

Le istruzioni di un instruction group:

- Non conffiggono nell'uso delle unità funzionalità.
- Non conffiggono nell'uso dei registri.
- Non hanno dipendenze sui dati e sui nomi.

La CPU può schedulare come vuole questi instruction group, ma devono essere eseguiti in sequenza e nell'ordine in cui sono stati prodotti dal compilatore. Il compilatore usa anche un'istruzione speciale detta *stop* che separa gruppi di istruzioni.

**Corollario 2.3.3 Bundle**

In fase di esecuzione le istruzioni vengono prelevate dal processore in blocchi di lunghezza fissa detti *bundle*.

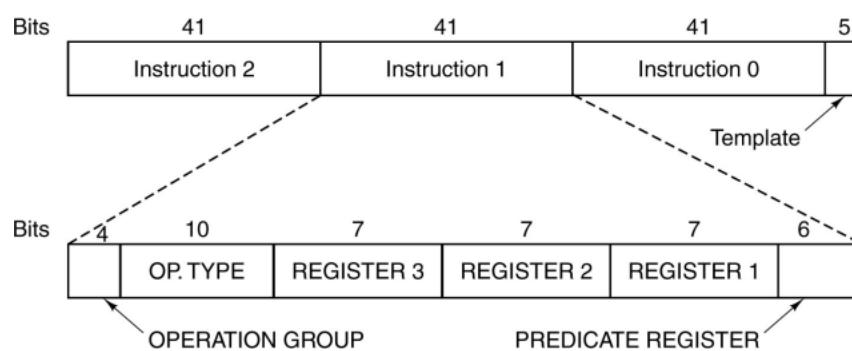


Figure 2.19: Organizzazione di un bundle.

#### Corollario 2.3.4 Advanced Load

L'advanced load (o LDA) è una forma limitata di speculazione hardware che utilizza le load speculative. Quando viene eseguita una LDA viene creata una nuova entry in una tavola detta **ALAT** contenente:

- Il numero del registro di destinazione della load.
- l'indirizzo della locazione di memoria usata dalla load (noto solo a run time).

**Note:-**

Una advanced load è una load che è stata speculativamente spostata prima di una store da cui potrebbe potenzialmente dipendere.

#### Osservazioni 2.3.4

- Quando viene eseguita una store viene fatto anche un controllo associativo sull'ALAT.
- Prima che una qualsiasi istruzione usi il registro caricato da una LDA, l'entry della ALAT contenente quel registro viene controllata.
- Se la entry non è valida, la load viene rieseguita.



# 3

## Caching

### 3.1 Funzionamento di Base di una Cache

La memoria principale è sempre stata più lenta di una CPU, questo gap si è ampliato ulteriormente nel tempo.

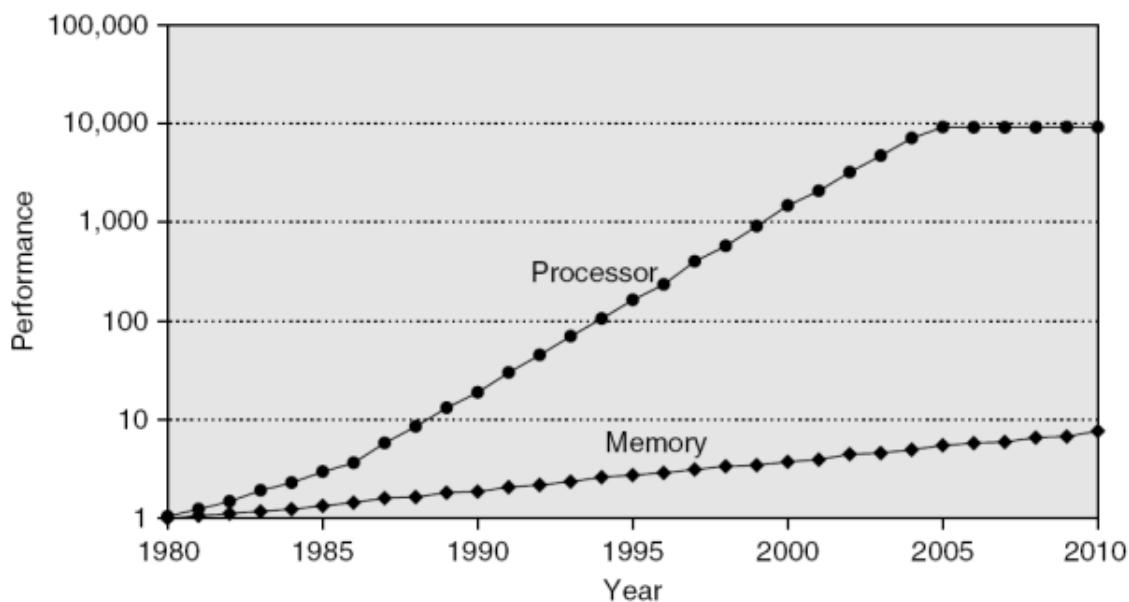


Figure 3.1: Andamento nel tempo.

**Note:-**

Inoltre la situazione peggiora ancora se si considerano anche i processori multi-core.

#### Osservazioni 3.1.1

- Il termine SRAM (Static RAM) indica la tecnologia di base con cui sono costruite le RAM usate nei vari livelli di cache del processore.
- Le SRAM usano un circuito di transistor che devono essere sempre alimentati.

- Le DRAM usano un condensatore la cui carica indica se un bit vale 0 o 1. Ogni pochi millisecondi viene fatto un refresh.
- Le SRAM sono più veloci, ma consumano di più, sono più complesse e consumano di più.
- La cache L1 è ottimizzata rispetto alla velocità di accesso, mentre L2 e L3 rispetto alla capienza di memorizzazione.

### Il concetto di caching:

- I registri fanno da cache per la cache hardware.
- La cache hardware fa da cache per la RAM.
- La RAM fa da cache per l'Hard disk (memoria virtuale).
- L'hard disk fa da cache per supporti magnetici più lenti.

#### Definizione 3.1.1: Gerarchi di cache

- L1: due cache, una per la data memory l'altra per la instruction memory.
- L2 e L3 (non in tutti).

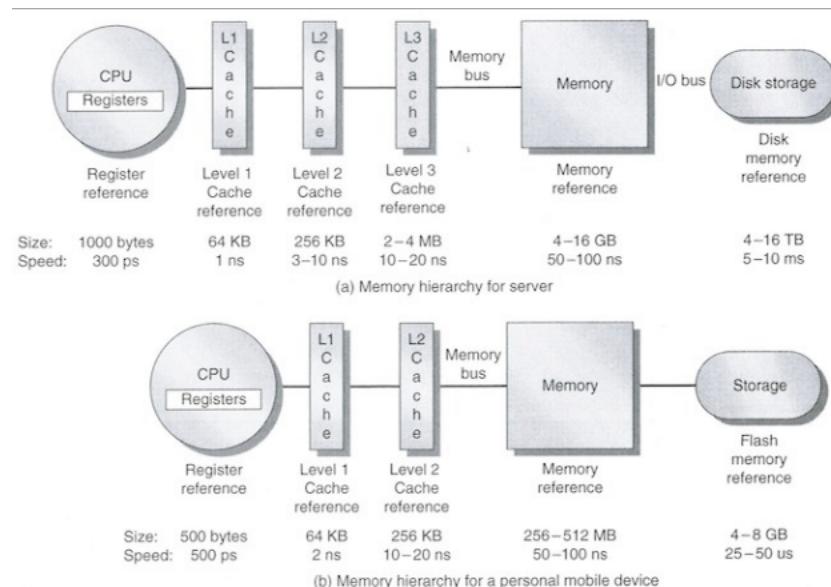


Figure 3.2: Gerarchia di memorie.

#### Corollario 3.1.1 Località Spaziale

Area di memoria con indirizzi simili a quelli appena usati, saranno a loro volta usate nell'immediato futuro.

#### Corollario 3.1.2 Località Temporale

Locazioni accedute di recente verranno di nuovo accedute nell'immediato futuro.

#### Definizione 3.1.2: Linee

Per permettere l'interazione tra memoria cache e RAM, entrambe sono suddivise in blocchi di dimensione fissa dette linee di cache e linee di RAM rispettivamente.

**Osservazioni 3.1.2**

- Ogni linea è identificata dal suo indirizzo in RAM. L'indirizzo in RAM del primo byte appartiene a quella linea.
- Il numero della linea dipende dagli  $m$  bit più significativi.

**Corollario 3.1.3 Cache Hit**

Se viene indirizzata una word che si trova nella cache si ha un cache hit: tutto procede normalmente in quanto la cache è in grado di lavorare alla stessa velocità degli altri elementi del datapath.

**Corollario 3.1.4 Cache Miss**

Se viene indirizzata una word che non si trova nella cache si ha un cache miss: il dato viene prelevato dalla RAM e se necessario una linea della cache viene rimossa per fare spazio a quella mancante.

**Definizione 3.1.3: Cache Direct-Mapped**

Le cache Direct-Mapped (in italiano: a indirizzamento diretto) sono il tipo di cache più semplice. Una cache Direct-Mapped è formata da  $2^k$  entry numerate consecutivamente.

Ogni entry memorizza 32 o 64 byte consecutivi e ha associate due informazioni:

- Un *bit di validità* che dice se quella entry contiene dell'informazione significativa.
- Un campo *TAG* che identifica univocamente la linea contenuta in quella entry della cache rispetto a tutte le linee della RAM.

**Note:-**

In una cache direct-mapped ogni linea della RAM viene memorizzata in una entry ben precisa della cache. Per stabilire in quale entry della cache cercare una linea che contiene il dato o l'istruzione indirizzati dalla CPU si usa l'operazione: (numero della linea in RAM) modulo (numero di entry nella cache).

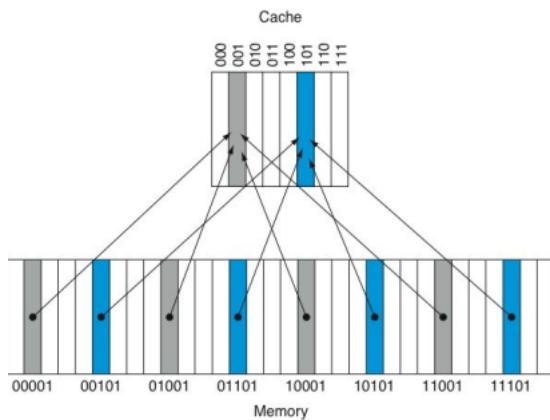


Figure 3.3: Linee di cache.



# 4

## Architetture Parallele



# 5

Quantum Computing



6  
GPU

