

---

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 2</b>
1.1	Introduzione Tassonomia delle architetture — 3 • Alcuni concetti fondamentali — 3	2
1.2	Una semplice macchina RISC MIPS - Versione monociclo — 5 • Banco dei registri — 7 • Una semplice Control Unit — 8 • L'esecuzione di un'istruzione — 9	4
1.3	Dal monociclo al multiciclo MIPS - Versione multiciclo — 9 • Control Unit multiciclo — 11 • Macchine a stati finiti e Microprogrammi — 12 • Complex Instruction Set Computer (CISC) — 12 • Reduced Instruction Set Computer (RISC) — 13	9
1.4	Pipeline L'Architettura MIPS Pipelined — 14 • Problemi della Pipeline — 18 • Multiple Pipeline — 20 • Scheduling della Pipeline — 21	14
<b>CAPITOLO 2</b>	<b>INSTRUCTION LEVEL PARALLELISM (ILP)</b>	<b>PAGINA 24</b>
2.1	Introduzione Aumentare la Frequenza del Clock della CPU — 24 • Multiple Issue — 25	24
2.2	ILP Dinamico Scheduling Dinamico, Branch Prediction e Speculazione Hardware — 26 • I Problemi di Fondo — 27 • L'Approccio di Tomasulo — 28 • Multiple Issue con ILP Dinamico — 35	26
2.3	ILP Statico Scheduling Statico e Loop Unrolling — 38 • Multiple Issue Statico — 41 • IA-64: Itanium — 43	38
<b>CAPITOLO 3</b>	<b>CACHING</b>	<b>PAGINA 47</b>
3.1	Funzionamento di Base di una Cache Cache Direct-Mapped — 49 • Il supporto della RAM — 50 • Cache Set-Associative — 51	47
3.2	Ridurre i Cache Miss Cache a Più Livelli — 53 • Ulteriore Riduzione — 54	53
3.3	Tecnologie	54
<b>CAPITOLO 4</b>	<b>ARCHITETTURE PARALLELE</b>	<b>PAGINA 57</b>
4.1	I Problemi del Parallelismo Esplicito	57
4.2	Multi-Threading Introduzione — 59 • Tipi di Multi-Threading — 60 • Simultaneous Multi-Threading — 61	59
4.3	Multiprocessori a Memoria Condivisa Introduzione — 63 • UMA — 65 • NUMA — 70 • Sincronizzazione tra Processi — 72	63
<b>CAPITOLO 5</b>	<b>QUANTUM COMPUTING</b>	<b>PAGINA 76</b>

---

<b>CAPITOLO 6</b>	<b>GPU</b>	<b>PAGINA 78</b>
6.1	Che cosa è una GPU? Passaggio da GPU a CPU — 79	78
6.2	Gerarchia di Flynn SISD — 79 • SIMD — 80 • MIMD — 80 • SIMT — 80	79
6.3	Elaboratori Moderni Schema Generale — 80 • Architettura di una GPU — 81	80
6.4	Programmare una GPU: CUDA Decomposizione di un Problema — 83 • Il Paradigma CUDA — 84	83
6.5	I Thread Architettura per il Multithreading — 85 • L'Architettura del Multiprocessore — 86 • SIMT, WARP e WARP Scheduler — 87 • Blocchi, Griglie e Sincronizzazione — 87	85
6.6	Quale codice può girare su una GPU? Map: VectorAdd — 88 • Reduce — 89 • Copiare da Global Memory a Shared Memory — 90	88





# 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".

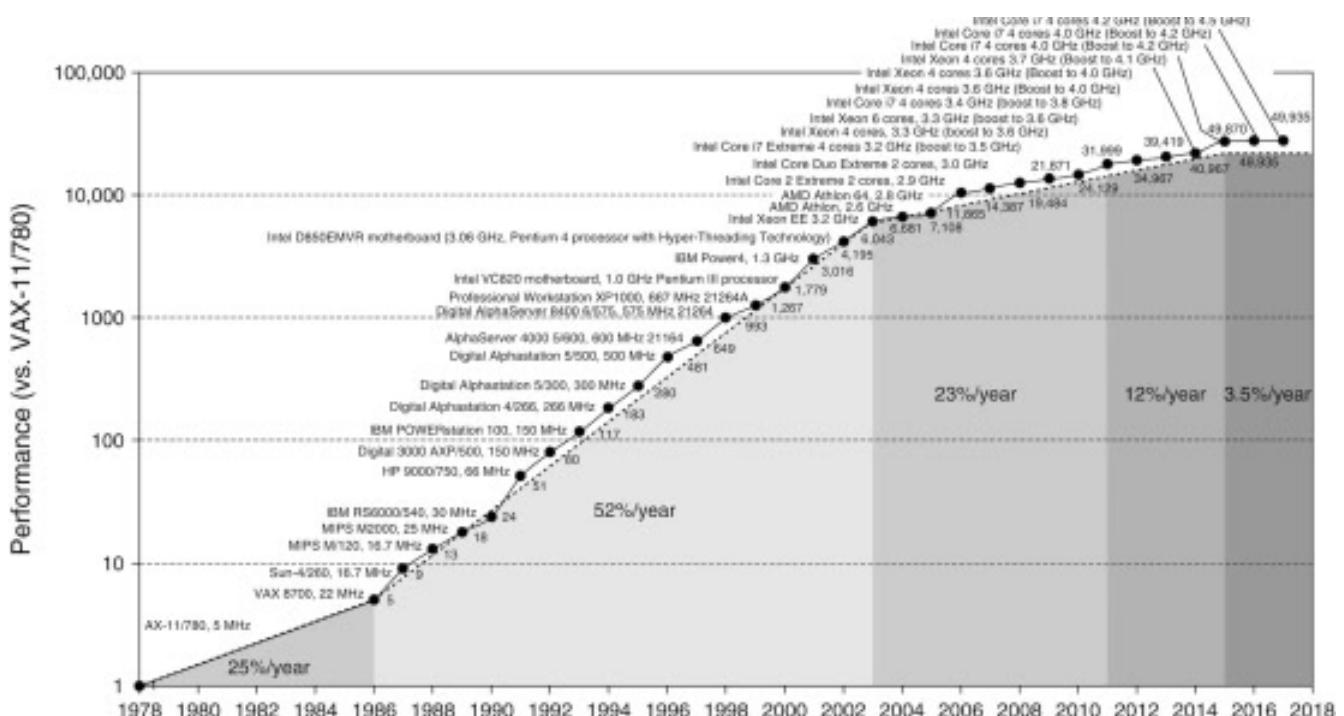


Figure 1.1: Nel 1986 ha inizio la "Rivoluzione RISC", mentre all'inizio degli anni 2000 si inizia a sfruttare l'idea di avere più "core".

### 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.

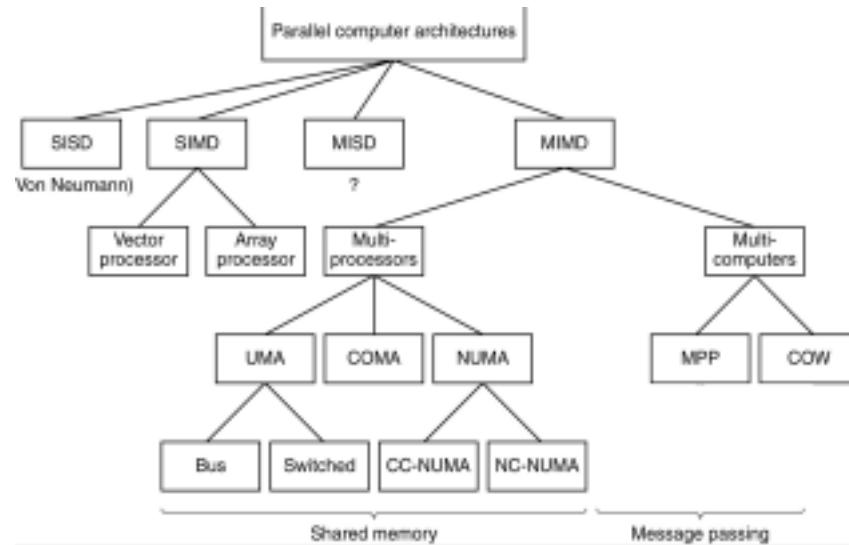


Figure 1.2: La Tassonomia di Flynn

### 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.

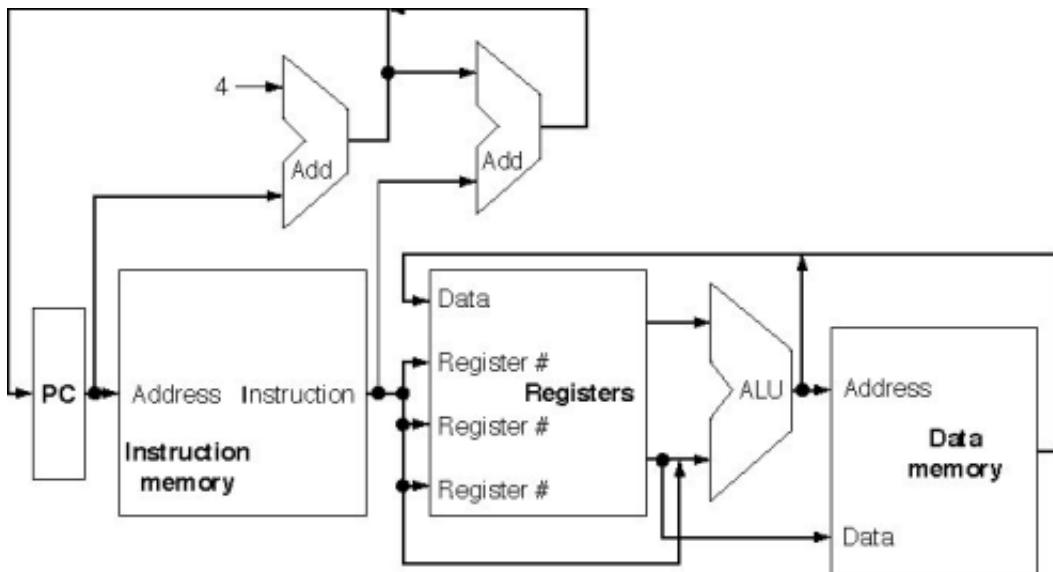


Figure 1.4: Schema ad alto livello del datapath MIPS

**Note:-**

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

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

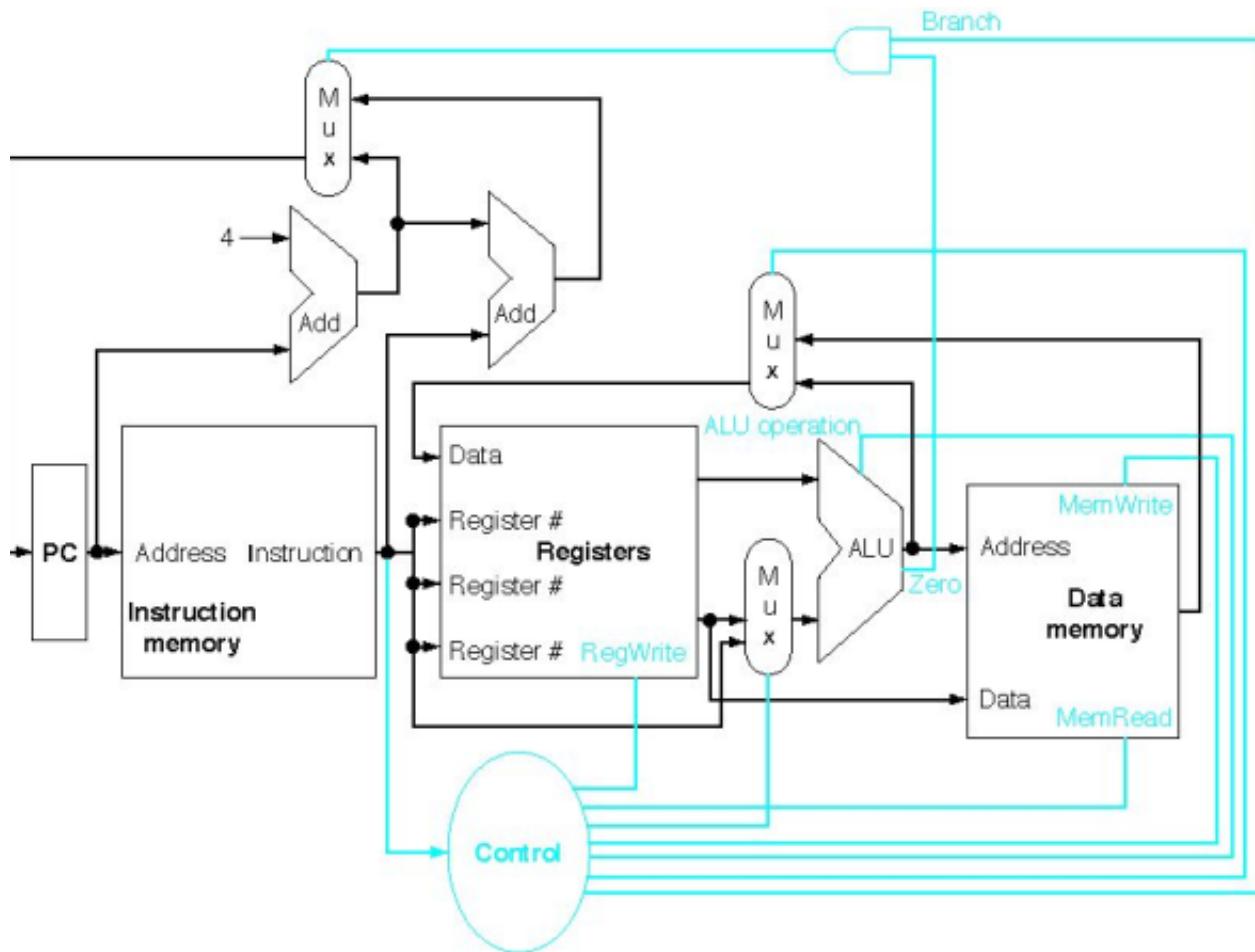


Figure 1.5: Versione modificata del MIPS con una Control Unit

**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:-**

Solitamente 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*.

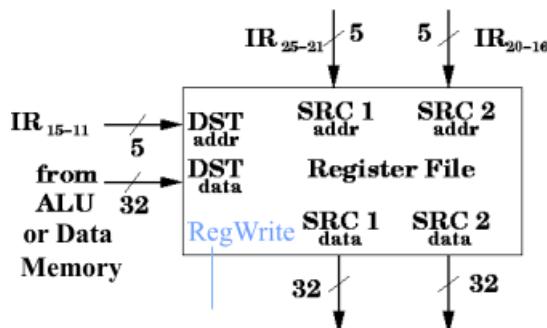


Figure 1.6: Operazione di scrittura

**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*.

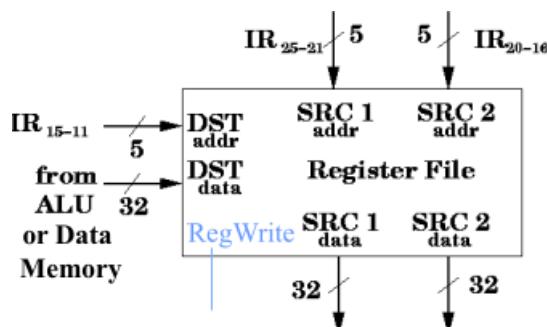


Figure 1.7: Operazione di lettura

### 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.

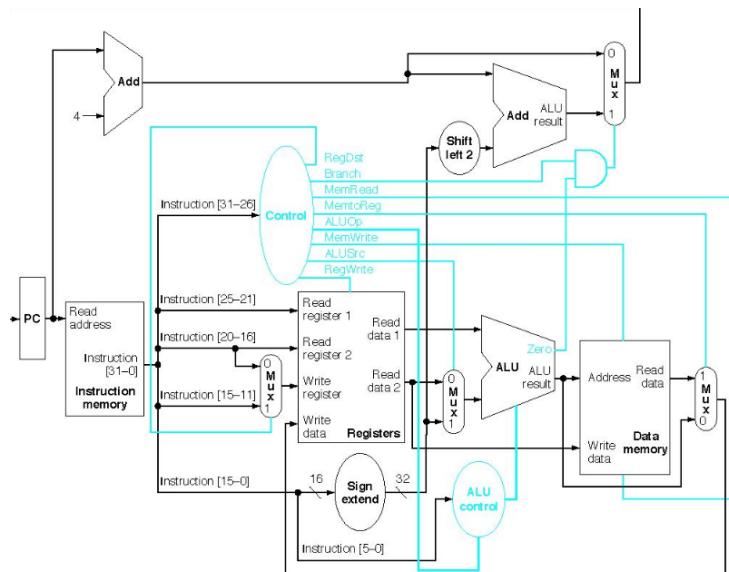


Figure 1.8: Versione modificata del MIPS la ALU Control

### 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

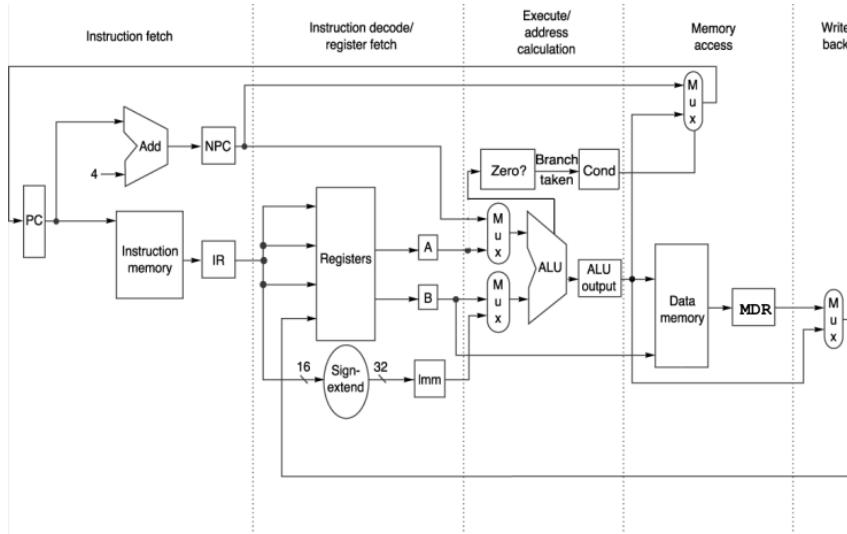


Figure 1.9: Il Datapath suddiviso nelle 5 fasi standard

**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.

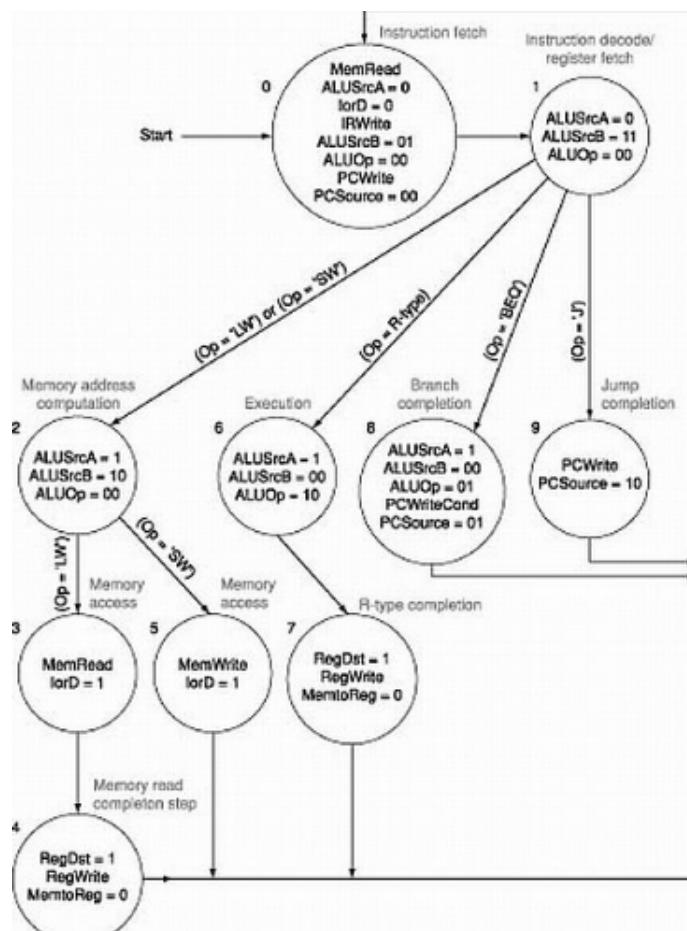


Figure 1.10: Unità di Controllo multiciclo

### 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.

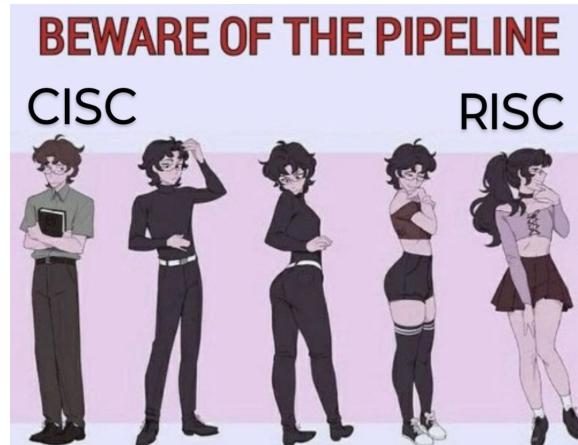


Figure 1.11: The virgin CISC vs. The chad RISC

#### 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.

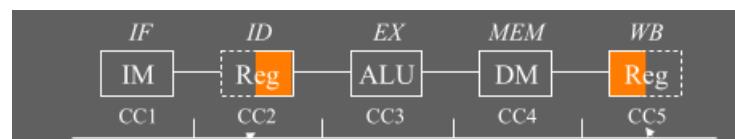


Figure 1.13: Utilizzo del banco dei registri.

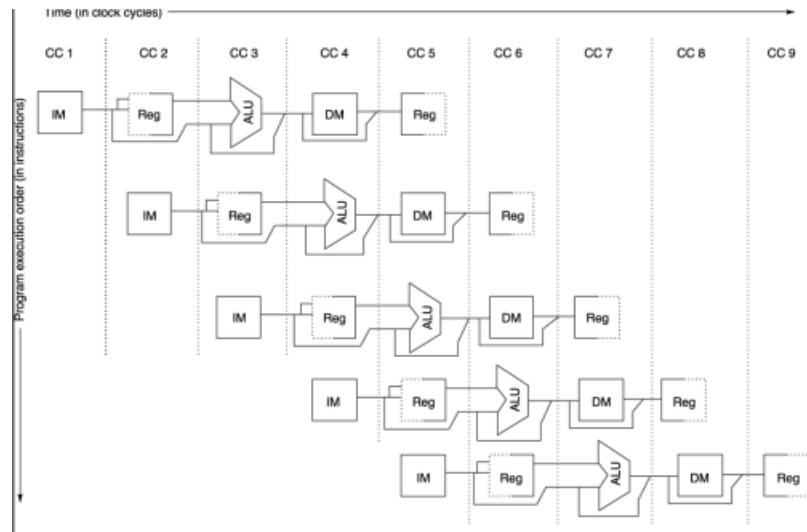


Figure 1.14: MIPS come serie di datapath traslati nel tempo.

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.

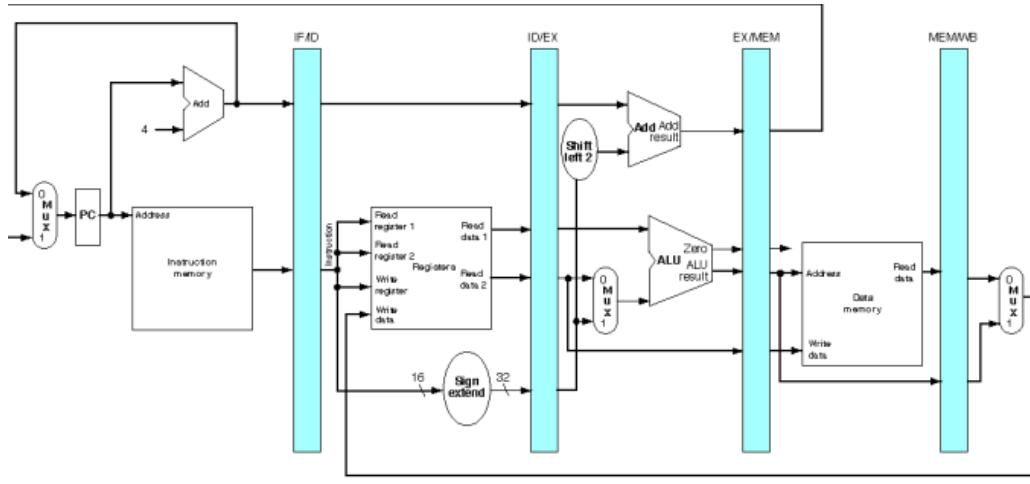


Figure 1.15: Registri della pipeline (in azzurro).

## 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;

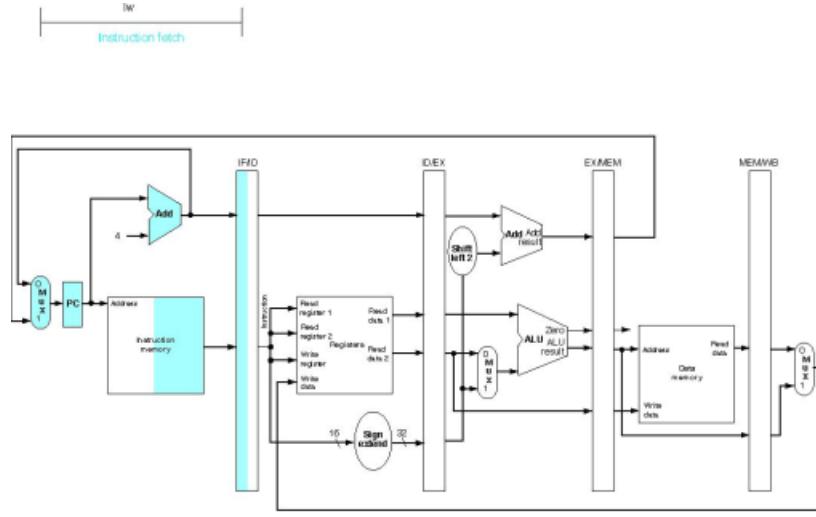


Figure 1.16: IF.

- **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;

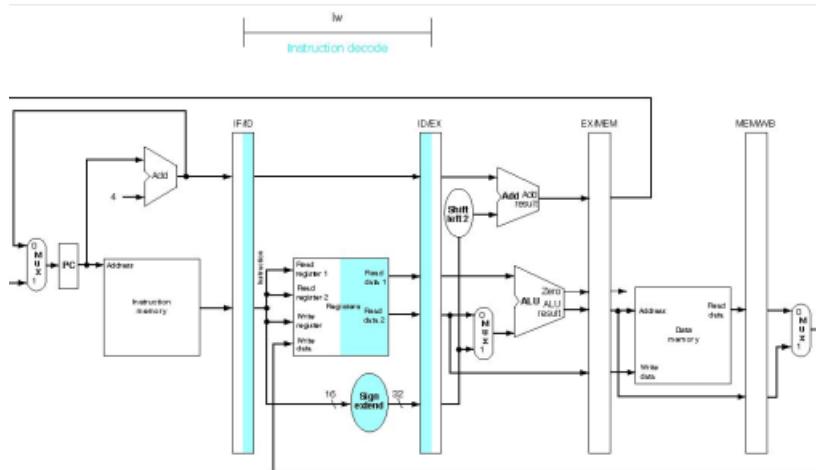


Figure 1.17: ID.

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

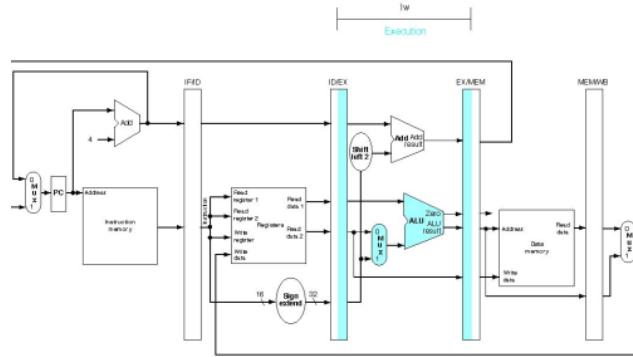


Figure 1.18: EX.

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

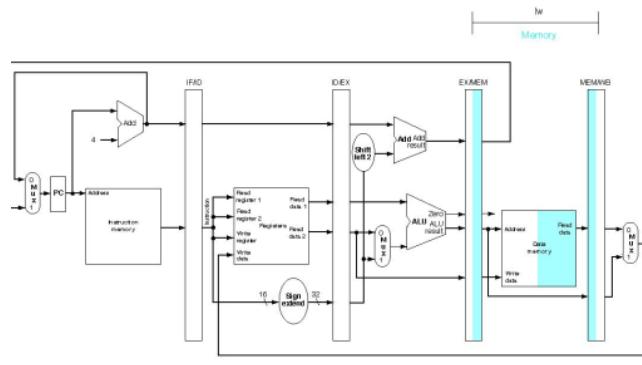


Figure 1.19: MEM.

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

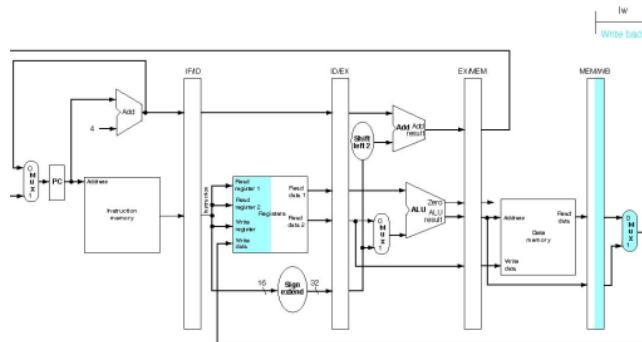


Figure 1.20: WB.

### 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.

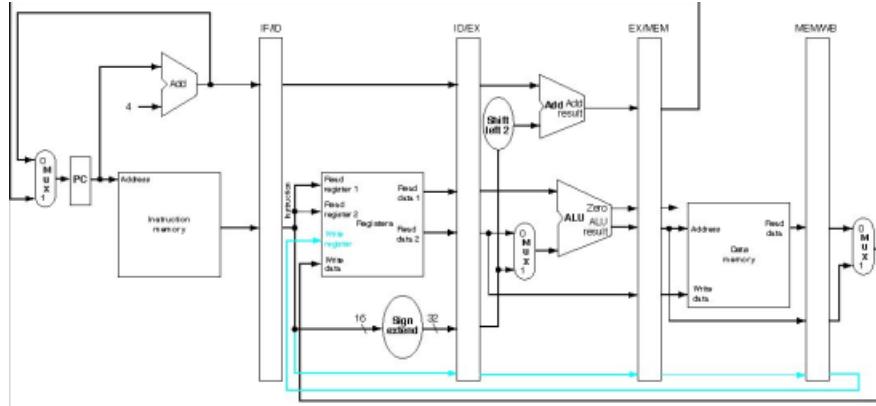


Figure 1.21: Datapath corretto.

#### 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.

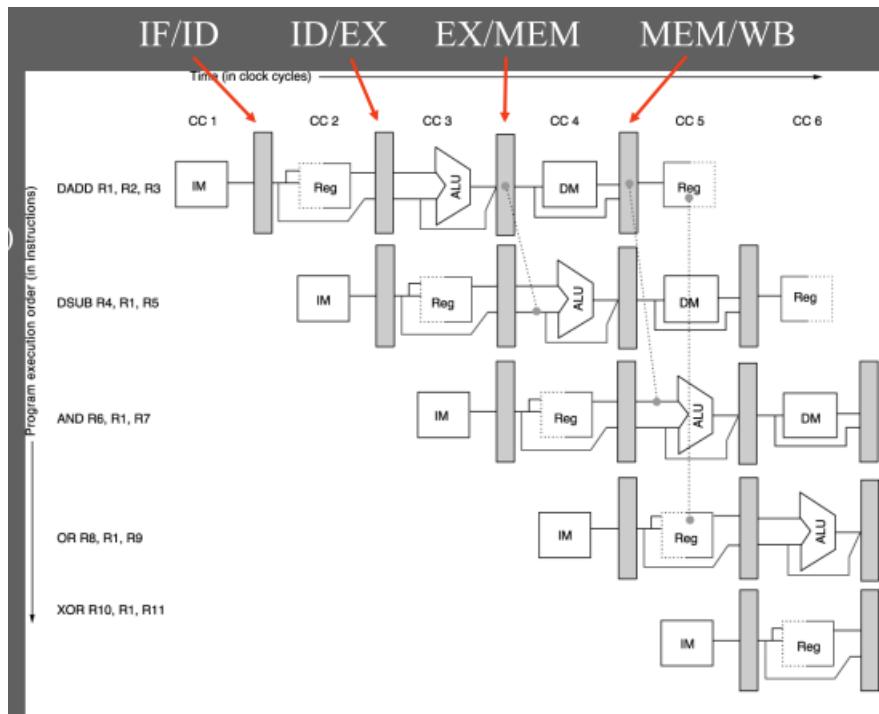


Figure 1.22: Forwarding.

**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.

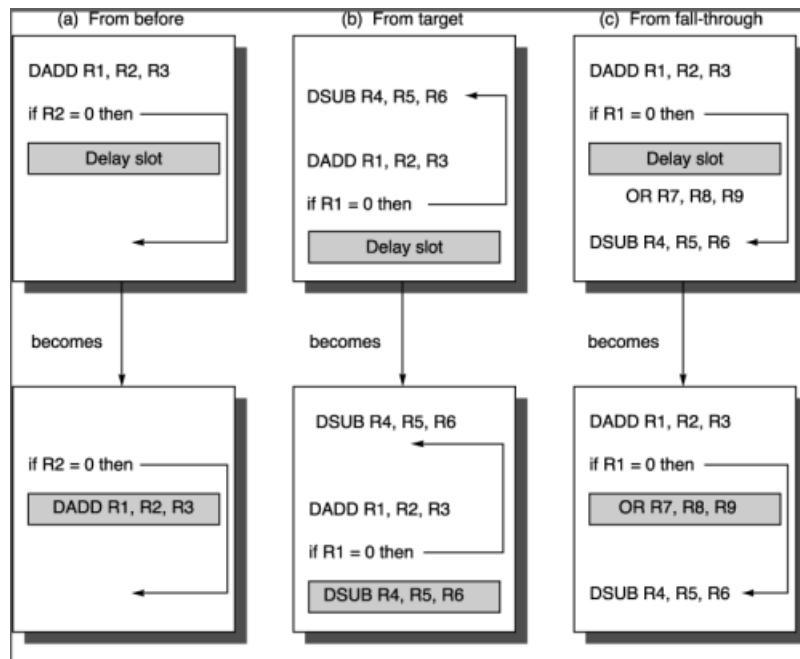


Figure 1.23: Delayed Branch.

### 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.

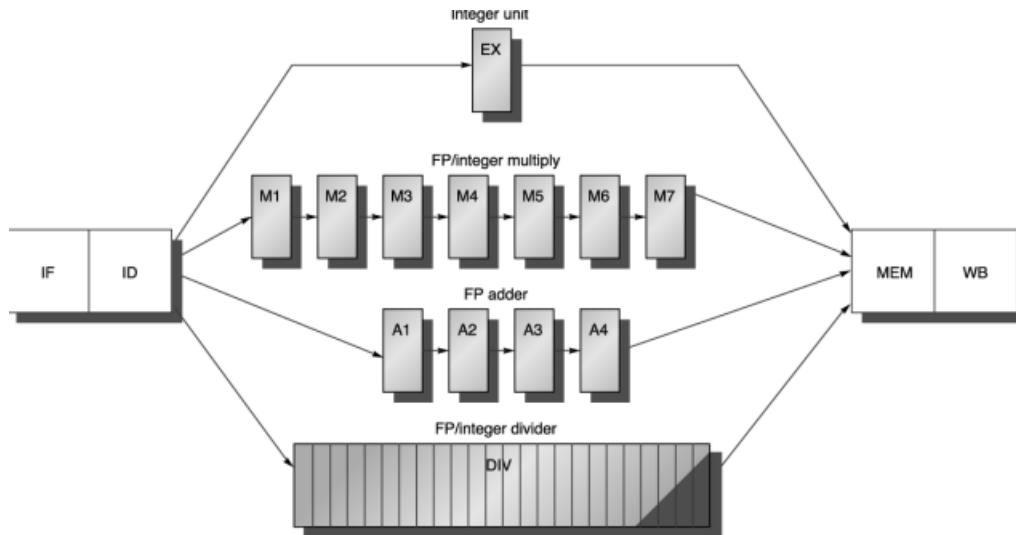


Figure 1.24: Pipeline con più unità funzionali.

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.

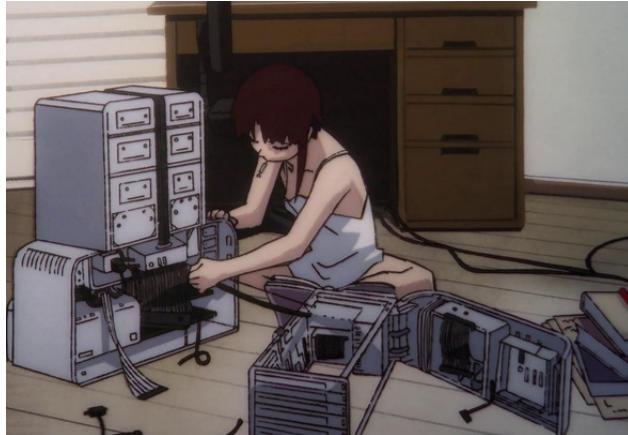


Figure 2.1: Io che faccio overclock del case.

### 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)
DADD R10, 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.

```
MUL R7, R3, R6
PRINT R7
LOAD R7, #100(R3)
```

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

```
DIV  F0, F2, F4
ADD F6, F0, F8
SUB F8, F10, F14
MUL F6, F10, F12
```

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.

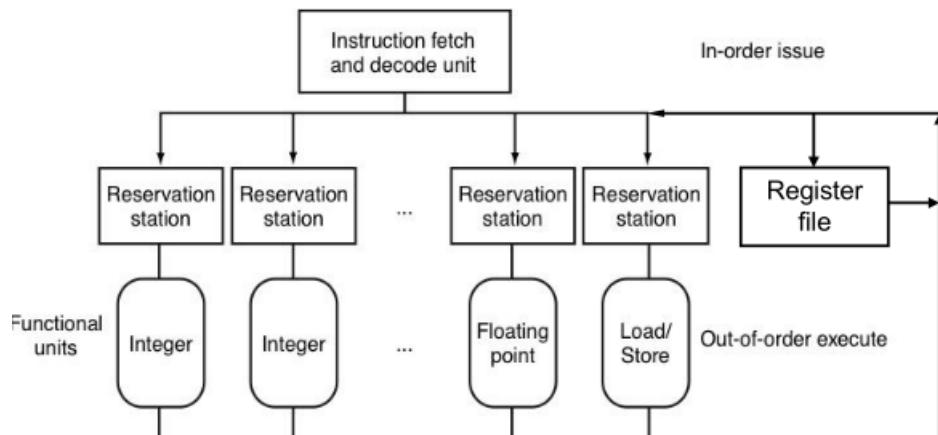


Figure 2.3: Lo schema di Tomasulo.

**Corollario 2.2.4 Reservation Station**

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

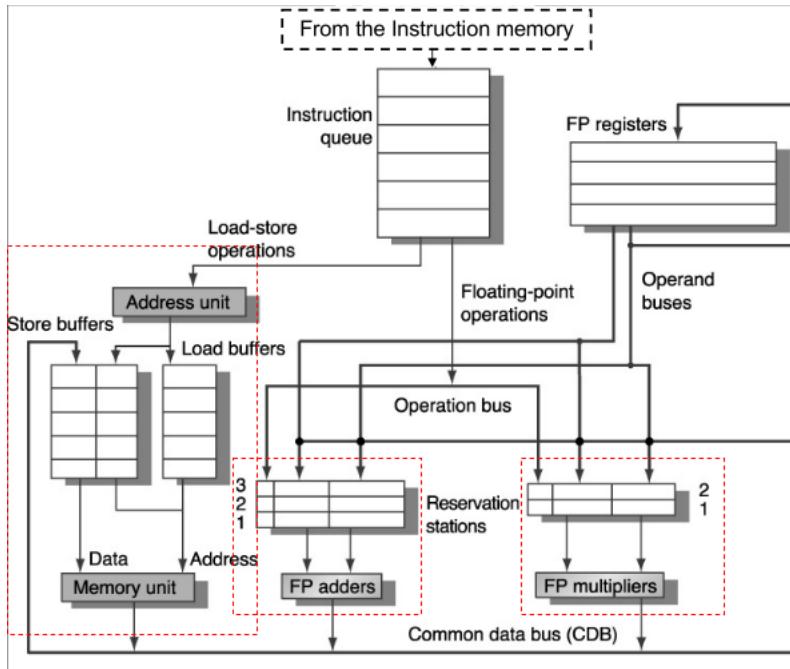


Figure 2.4: Sezione dello schema di Tomasulo che si occupa delle operazioni floating point.

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.

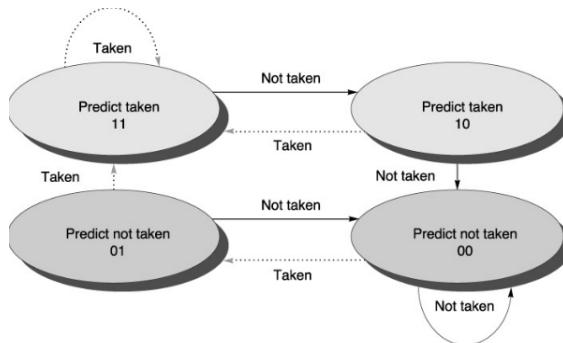


Figure 2.5: Schema di predizione a due bit.

##### 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.

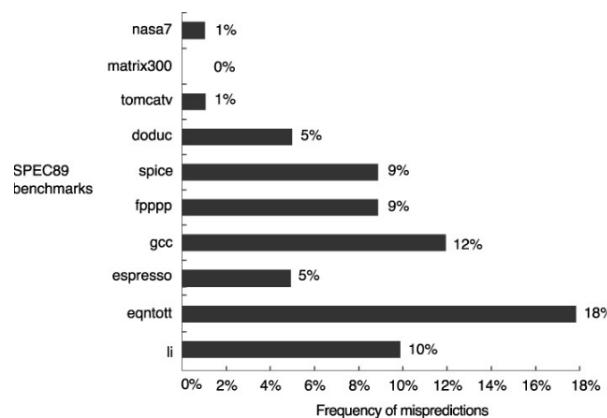


Figure 2.6: Percentuale di errori di predizione per banche prediction buffer a 2 bit e 4096 entry.

**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.

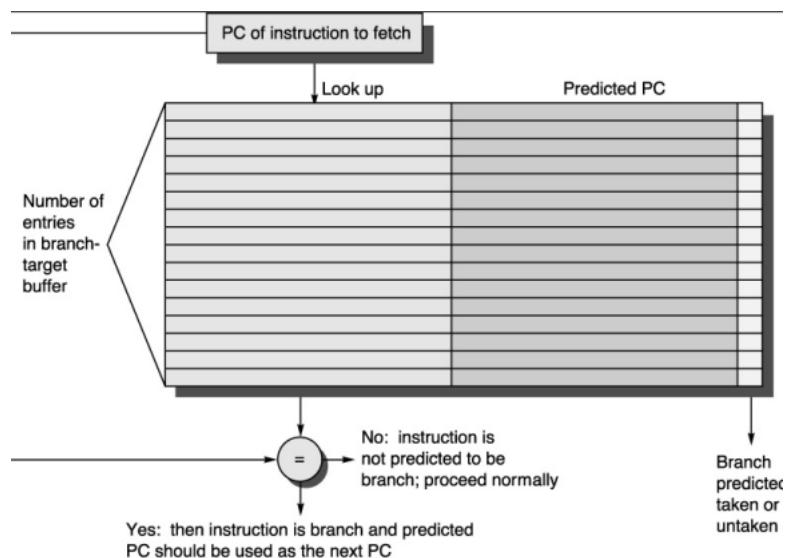


Figure 2.7: Branch Prediction Buffer.

**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.

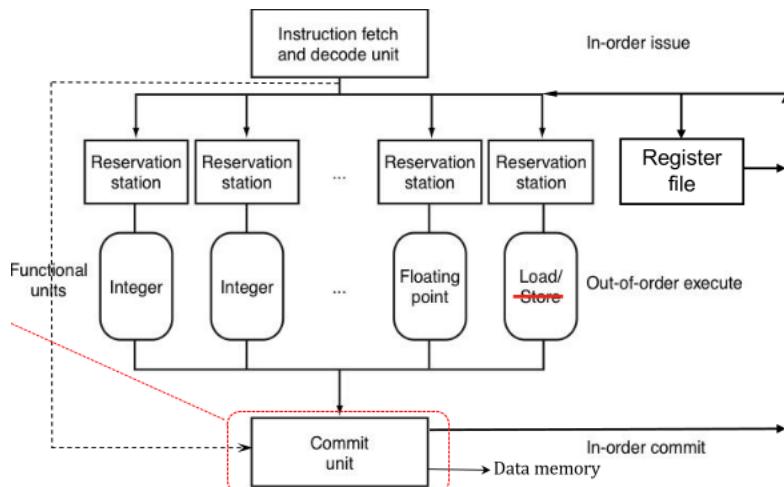


Figure 2.8: Speculazione hardware.

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.

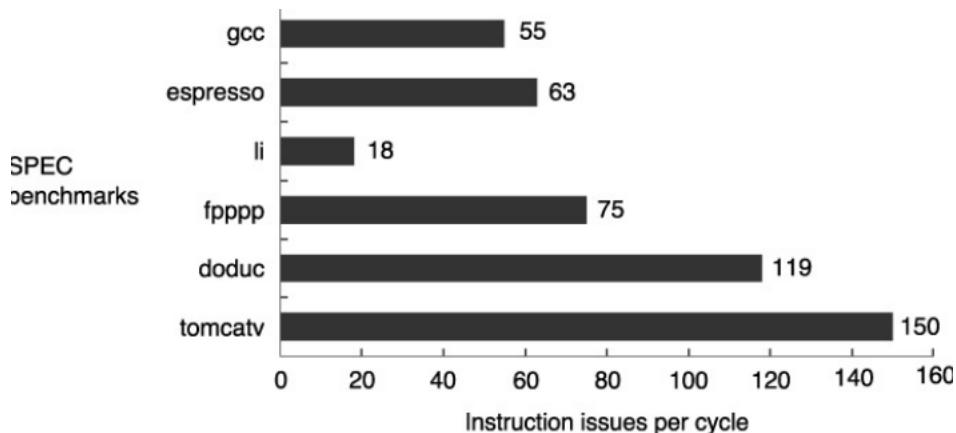


Figure 2.9: Alcuni Benchmark.

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.

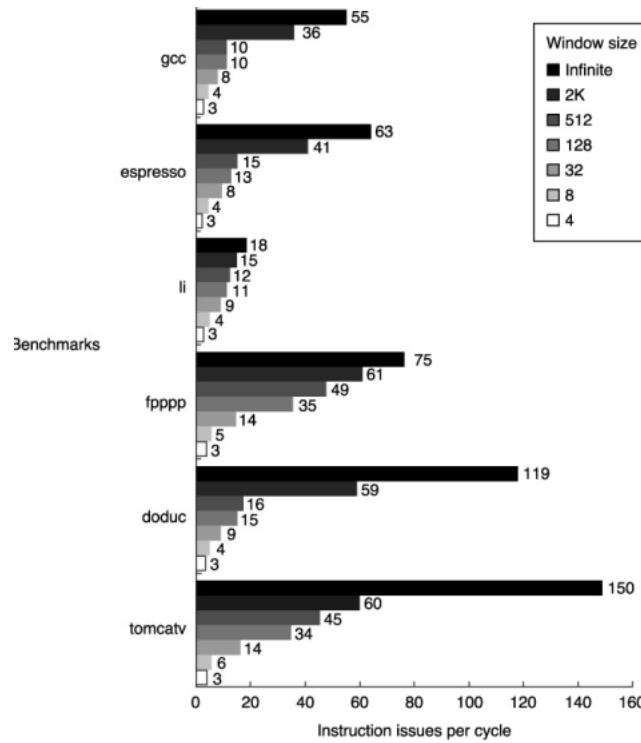


Figure 2.10: Alcuni Benchmark con la limitazione sul multiple issue.

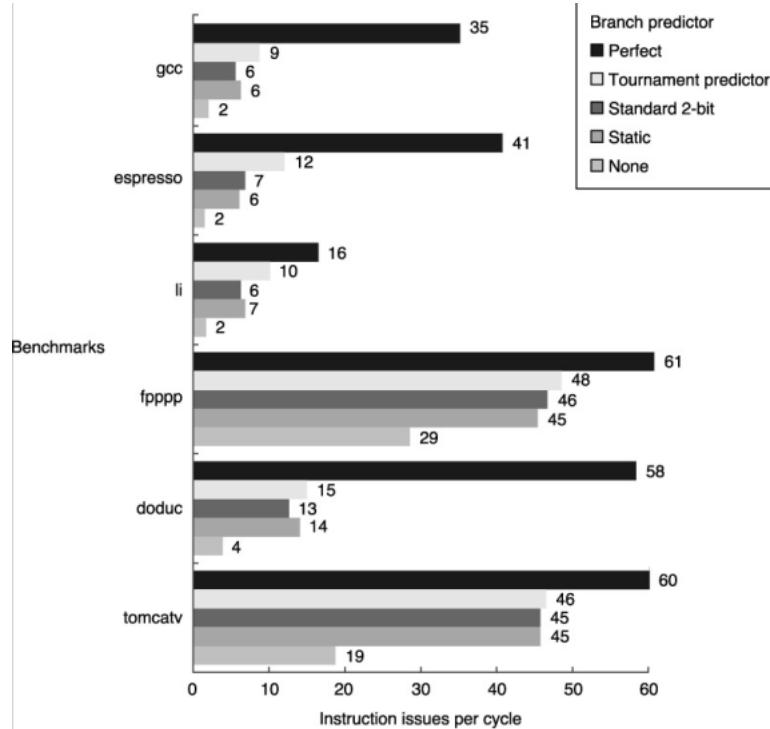


Figure 2.11: Alcuni Benchmark con la limitazione sul multiple issue (con 2k istruzioni) e sulla branch prediction.

**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 confliggono nell'uso delle unità funzionalità.
- Non confliggono 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*.

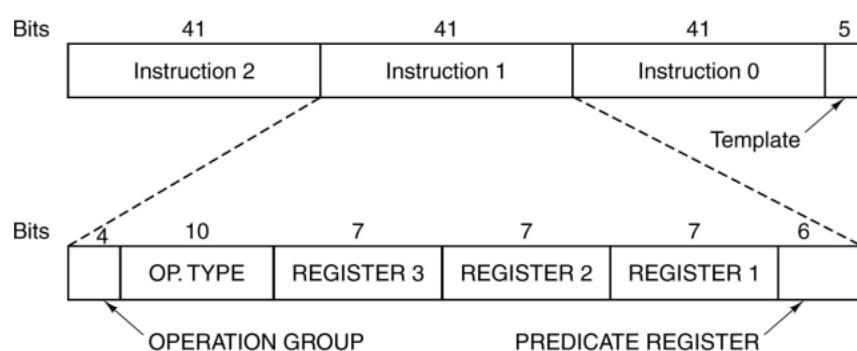


Figure 2.19: Organizzazione di un bundle.

**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.

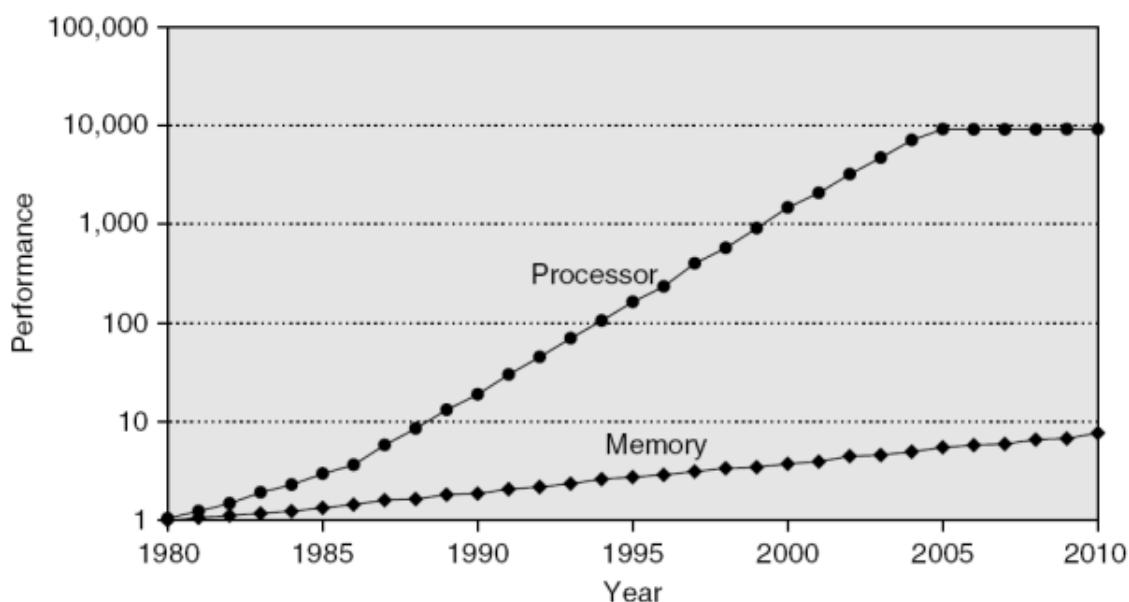


Figure 3.1: Andamento nel tempo.

**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).

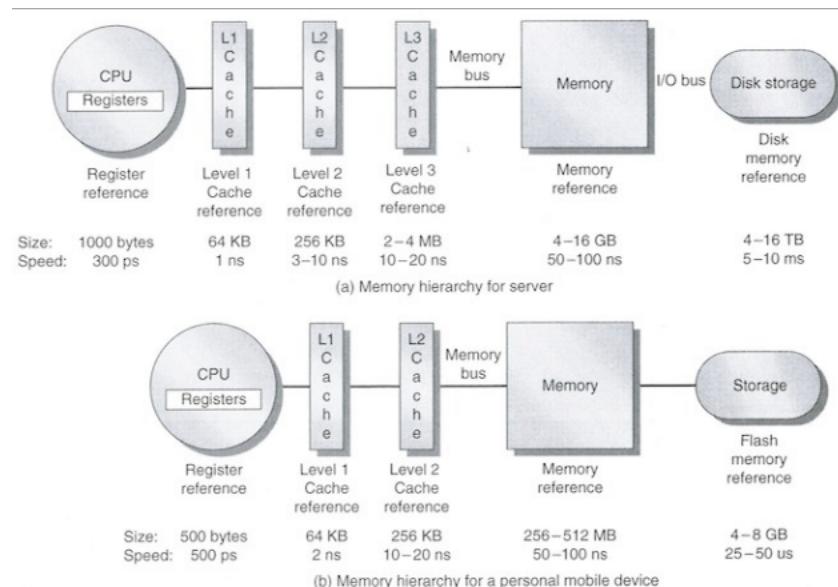


Figure 3.2: Gerarchia di memorie.

#### 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.

**Corollario 3.1.5 Penalità da fallimento**

Il tempo necessario a recuperare un dato in RAM quando si verifica un cache miss.

**La durata della penalità da fallimento è dovuta a:**

- Inviare l'indirizzo della linea mancante alla RAM.
- Accedere alla RAM per recuperare la linea.
- Inviare alla cache la linea recuperata.

### 3.1.1 Cache Direct-Mapped

**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).

**I 32 bit di cui è composto l'indirizzo sono suddivisi in:**

- **TAG:** 16 bit più significativi dell'indirizzo generato dalla CPU, usati per distinguere fra loro le linee di RAM che fanno capo alla stessa entry in cache.
- **CACHE ENTRY:** gli 11 bit che indicano in quale entry della cache cercare la linea che contiene il dato/istruzione indirizzato.
- **WORD:** i 3 bit con cui distinguiamo fra loro le 8 word contenute in una linea da 32 byte.
- **BYTE:** i 2 bit che possono servire a individuare i singoli byte di cui è composta ciascuna word.

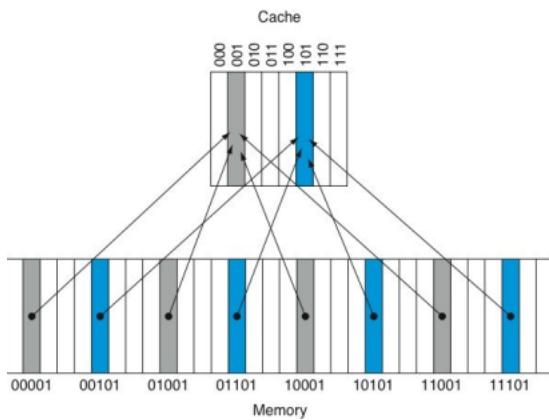


Figure 3.3: Linee di cache.

**Note:-**

Le prestazioni di una cache dipendono, oltre che dal numero di linee di RAM che è in grado di memorizzare, anche dalla dimensione scelta per le linee: linee più grandi riducono i cache miss, ma linee troppo grandi rallentano nel caso debbano essere rimosse dalla cache per cache miss.

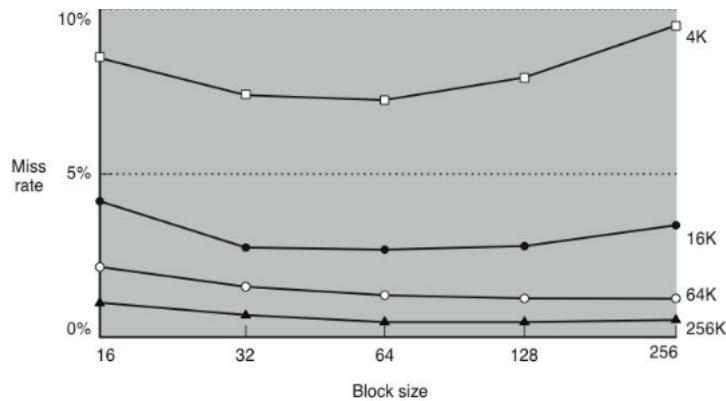


Figure 3.4: Confronto sulla dimensione delle linee du cache.

### 3.1.2 Il supporto della RAM

**Definizione 3.1.4: Coerenza della Cache**

Una linea in RAM e la corrispondente copia in cache sono sempre identiche.

**Corollario 3.1.6 Write-Through**

Le scritture vengono sempre eseguite sia in cache che in RAM in modo che i dati presenti nelle due memorie siano sempre consistenti tra di loro.

**Note:-**

Lo schema Write-Through non offre buone prestazioni perché richiede l'accesso, per ogni scrittura in cache, anche l'accesso alla RAM (il ché consuma molti cicli di clock).

**Corollario 3.1.7 Write-Back**

Il valore viene modificato solo nella cache. La modifica viene propagata in RAM (o ai livelli successivi di cache) solo se il dato stesso dev'essere rimpiazzato da un altro con cui condivide lo stesso tag.

Si consideri un sistema a 32 bit (e quindi con un bus sulla scheda madre a 32 bit). Se si suppone che un accesso alla RAM richieda 15 cicli di clock per leggere una word da 32 bit, per servire un cache miss con linee da 16 byte si impiegheranno 65 cicli di clock:

- 1 ciclo di clock per inviare l'indirizzo della linea mancante alla RAM.
- 4 x 15 cicli di clock per leggere l'intera linea.
- 4 x 1 cicli di clock per inviare l'intera linea alla cache.

Si può cercare di migliorare la situazione:

- Allargando la banda passante della RAM, ossia la quantità di dati che possiamo prelevare dalla RAM con un unico accesso.
- Allargando il bus in modo da poter trasportare in un unico ciclo di clock l'intera linea prelevata dalla RAM.

**Note:-**

Aumentare l'ampiezza del bus è costoso e non troppo efficiente. Per cui si sceglie di allargare la banda passante della RAM.

**Definizione 3.1.5: Memoria Interlacciata (Interleaved Memory)**

SI organizza la RAM in banchi in modo da poter leggere o scrivere più word (anziché una sola) in parallelo. Se una linea è formata da  $n$  word la linea viene distribuita su  $n$  banchi di RAM, ciascuno dei quali memorizza una delle  $n$  word della linea. Gli altri  $n$  banchi possono essere letti in parallelo estraendo così, in una sola lettura, tutte le word di cui è composta una linea.

**Note:-**

Si risparmia sul tempo necessario a prelevare un'intera linea dalla RAM. Applicando questo schema all'esempio precedente si scende da 65 a 20 cicli di clock.

**3.1.3 Cache Set-Associative**

Una caratteristica fondamentale delle cache Direct-Mapped, è che una determinata linea della RAM viene memorizzata sempre nella stessa entry della cache. Inoltre più linee di RAM fanno riferimento alla stessa entry della cache, dato che questa è meno capiente della RAM.

**Definizione 3.1.6: Cache Set-Associative a n vie**

Una cache Set-Associative a  $n$  vie è suddivisa in più insiemi ciascuno formato da  $n$  entry (2, 4, 6, 8) e può contenere  $n$  linee. Ogni linea della RAM corrisponde a uno e un solo insieme e quella linea può essere memorizzata in una qualsiasi delle  $n$  entry dell'insieme. In una cache Set-Associative l'insieme che contiene una linea è stabilito da: (numero della linea in RAM) modulo (numero di insiemi della cache).

**Note:-**

Quasi sempre nei computer sono presenti queste cache.

**Note:-**

Nella cache direct mapped, la linea 12 può andare solo nella entry numero 4. Nella cache set-associativa la linea 12 può andare nella entry 0 o nella entry 1 dell'insieme zero.

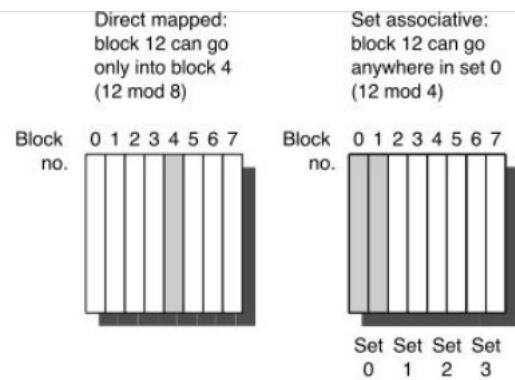


Figure 3.5: Confronto tra Direct-Mapped e Set-Associative.

In una cache set-associativa quindi, una volta individuato l’insieme che può contenere una certa linea, occorre ancora verificare se una delle entry di quell’insieme contiene la linea indirizzata (il che viene fatto con una ricerca associativa su tutti gli elementi dell’insieme). Se una linea indirizzata manca, va portata in cache, e inserita in una qualsiasi entry (è meglio scegliere una entry libera). Se invece tutte le entry dell’insieme sono occupate allora si sovrascrive una delle linee dell’insieme. Di solito si sacrifica quella indirizzata meno di recente, con rimpiazzamento LRU (Least Recent Used).

**Note:-**

Le cache Set-Associative sono più sofisticate, quindi richiedono apposito hardware, ma hanno prestazioni maggiori.

**Corollario 3.1.8 Cache completamente associativa**

In generale, una cache associativa in cui c’è un solo unico grande insieme, e una linea può essere allocata in una qualsiasi entry della cache, prende il nome di cache completamente associativa (fully associative cache).

**Note:-**

Sono ancora più efficienti, ma molto più costose (esistono solo a livello teorico).

Associatività	Frequenza dei fallimenti
1 via (direct mapped)	10.3%
2 vie	8.6%
4 vie	8.3%
8 vie	8.1%

Figure 3.6: Confronto sul numero di vie.

**Note:-**

Servire un cache miss è molto costoso, se si hanno più vie c’è un miglioramento evidente.

**Osservazioni 3.1.3**

Le cache hanno un effetto positivo in base a quanto:

- Riescono a diminuire il tempo per servire un cache miss.

- Riescono a ridurre il numero di cache miss.

## 3.2 Ridurre i Cache Miss

### 3.2.1 Cache a Più Livelli

Per ridurre ulteriormente il numero di cache miss si possono introdurre più livelli di cache. Così facendo si riesce tra l'altro a raggiungere un accettabile trade-off tra l'avere una cache veloce quanto la CPU (ma costosa, e quindi piccola) e una cache sufficientemente grande da contenere una buona parte della RAM (più lenta, ma più economica).

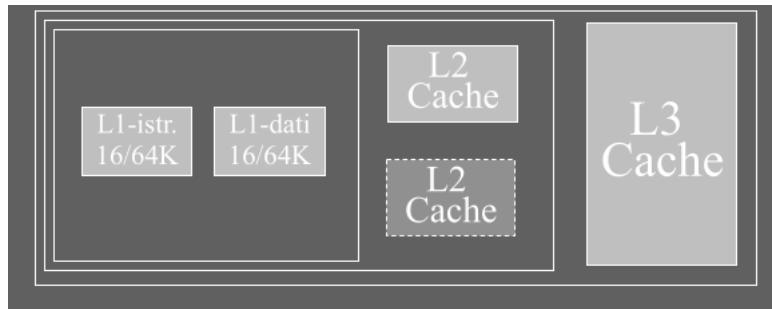


Figure 3.7: I vari livelli di cache, normalmente  $L1 \leq L2 \leq L3$ .

**Note:-**

Nei moderni multi-core, ogni core ha a disposizione L1 ed L2 private, mentre L3 è condivisa. L1 ed L2 risiedono sullo stesso chip del processore, mentre L3 può essere:

- Sullo stesso chip.
- Separato nello stesso package.
- Sulla motherboard.

**In un sistema con almeno due livelli di cache:**

1. La cache di primo livello deve avere un tempo di accesso simile alla velocità della CPU per non rallentare l'esecuzione in caso di *cache hit*.
2. La cache di secondo livello deve avere una dimensione maggiore della cache di primo livello per ridurre il numero di *cache miss* rispetto alla cache di primo livello.
3. Lo stesso ragionamento vale per la cache di terzo livello rispetto a quella di secondo.

Cache	tempo di accesso	dimensioni
L1	1÷4 clock cycles	16÷64 Kbytes
L2	8÷15 clock cycles	256÷1024 Kbytes
L3	25÷50 clock cycles	2÷10 Mbytes

Figure 3.8: I dati sulle dimensioni sono un po' outdated tbh.

### 3.2.2 Ulteriore Riduzione

Un'altro modo per ridurre il numero di cache miss è quello di rendere l'utilizzo della cache il più efficace possibile, per esempio con una cache Set-Associative, ma anche aumentando le dimensioni delle linee o della cache stessa.

Normalmente, gli algoritmi sono valutati solo a livello teorico, in base al numero di operazioni da compiere per risolvere un problema, rispetto alla "dimensione" del problema. A livello pratico tuttavia, il modo in cui un computer funziona può influenzare enormemente le effettive prestazioni di un programma. La presenza della cache è uno degli aspetti pratici, nell'esecuzione dei programmi, che può influenzarne enormemente i tempi di esecuzione. Un algoritmo che sembra più efficiente di altri a livello teorico, può in realtà fornire prestazioni peggiori se fatto girare su un sistema che usa un sistema di cache. Un esempio è il Radix Sort, un algoritmo di ordinamento che, per array molto grandi (milioni di elementi), fornisce prestazioni migliori del Quick Sort in termini del numero di operazioni richieste per l'ordinamento.

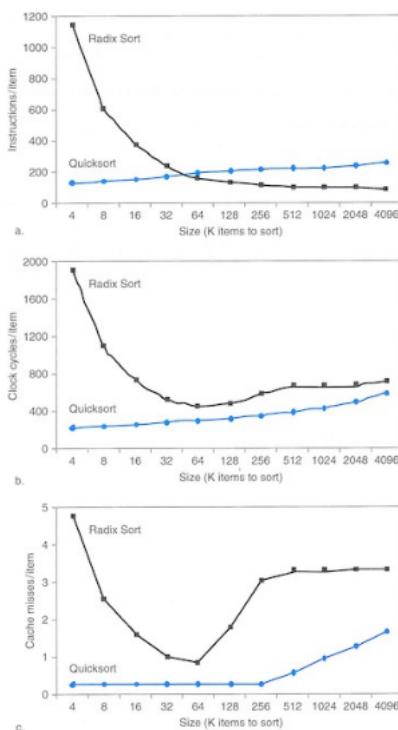


Figure 3.9: Prestazioni di quick sort e radix sort a confronto.

**Note:-**

Il radix sort genera più cache miss e quindi a livello pratico è peggiore del quick sort.

## 3.3 Tecnologie

**Definizione 3.3.1: DIMM**

DIMM: Dual Inline Memory Modules. I chip di DRAM sono normalmente montati su piccole schede contenenti da 4 a 16 DRAM organizzate in modo da memorizzare i dati in blocchi di 8 bytes (+ ECC).

**Definizione 3.3.2: SDRAM**

SDRAM: Synchronous DRAM. Sono DRAM dotate di un clock, in modo da potersi sincronizzare col clock del controller della RAM, e ottimizzare le prestazioni.

**Definizione 3.3.3: DDR**

DDR: Double Data Rate. Sono SDRAM in cui è possibile trasferire dati sia sul fronte discendente che su quello ascendente del clock, in questo modo raddoppiando le prestazioni rispetto a normali DRAM.

**Corollario 3.3.1 DDR n**

DDR2, DDR3, DDR4... L'acronimo DDR indica ormai comunemente una sequenza di standard, in cui viene man mano aumentato il clock della DRAM, e quindi la quantità di dati trasferiti per secondo

**Note:-**

Viene anche diminuito il voltaggio di funzionamento, in modo da diminuire i consumi. DDR=2,5 volt; DDR2=1,8 volt; DDR3=1,5 volts; DDR4=1,2 volts.

Standard	Clock rate (MHz)	M transfers per second	DRAM name	MB/sec /DIMM	DIMM name
DDR	133	266	DDR266	2128	PC2100
DDR	150	300	DDR300	2400	PC2400
DDR	200	400	DDR400	3200	PC3200
DDR2	266	533	DDR2-533	4264	PC4300
DDR2	333	667	DDR2-667	5336	PC5300
DDR2	400	800	DDR2-800	6400	PC6400
DDR3	533	1066	DDR3-1066	8528	PC8500
DDR3	666	1333	DDR3-1333	10,664	PC10700
DDR3	800	1600	DDR3-1600	12,800	PC12800
DDR4	1066–1600	2133–3200	DDR4-3200	17,056–25,600	PC25600

Figure 3.10: Prestazioni delle DRAM nel 2010.

**Definizione 3.3.4: GDRAM**

GDRAM: Graphics (Synchronous) DRAM. Sono particolari tipi di DDR (e infatti sono anche denominate GDDR) adatte a gestire le alte larghezze di banda richieste dai processori grafici:

- Trasferiscono più bit in parallelo rispetto alle DRAM.
- Sono saldate sulla GPU per diminuire la dispersione del segnale e aumentare la frequenza di clock.

**Definizione 3.3.5: FLASH Memory**

FLASH Memory. Sono particolari tipi di EEPROM (Electronically Erasable Programmable Read-Only Memory): possono non solo essere lette, ma anche riscritte. Tuttavia, la riscrittura avviene a livello di interi blocchi di dati, ossia anche i dati non modificati del blocco vanno riscritti.

**Le memorie FLASH sono:**

- Statiche, quindi mantengono i dati anche se non alimentate.
- Hanno un numero di cicli di riscrittura limitato a circa 100k per blocco.
- Meno costose delle SDRAM, ma più costose degli Hard Disk.
- In lettura poco più lente delle SDRAM, ma 1000 volte più veloci degli Hard Disk. Ma in scrittura le prestazioni peggiorano molto.



# 4

## Architetture Parallele

Per aumentare ancora di più le prestazioni, se è già stato applicato tutto ciò descritto in precedenza, si deve ricorrere a un'architettura dotata di più unità computazionali in modo da avere un *parallelismo esplicito*.

**Note:-**

Preso in considerazione il fatto che i programmi debbano andare riscritti per girare su più core.

### 4.1 I Problemi del Parallelismo Esplicito

Tuttavia, (eccetto che in casi molto molto particolari) l'incremento di prestazioni (il cosiddetto *speed-up*) che si può ottenere usando più CPU su cui far girare in parallelo i vari programmi è meno che lineare rispetto al numero di CPU disponibili. Principalmente perché i programmi che girano in parallelo dovranno sincronizzarsi tra loro. Consideriamo il comando "gcc main.c function1.c function2.c -o output". Supponiamo che, lanciando il programma su una macchina monoprocessoresso ci vogliano 7 secondi:

- 3 secondi per compilare main.c
- 2 secondi per compilare function1.c
- 1 secondo per compilare function2.c
- 1 secondo per linkare gli oggetti.

Con 3 CPU a disposizione i tre sorgenti possono essere compilati in parallelo, ma l'operazione di linking può essere eseguita solo dopo che tutti e tre gli oggetti sono stati generati (quindi dopo 3 secondi). In questo modo il tempo si riduce da 7 a 4 secondi con uno speed-up di 1.75 pur avendo usato 3 processori.

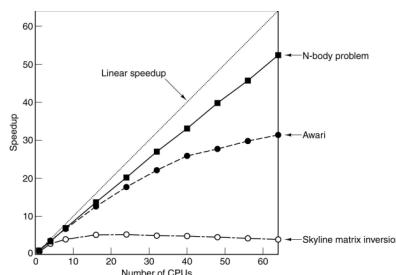


Figure 4.1: Speed-up di alcuni problemi computazionali.

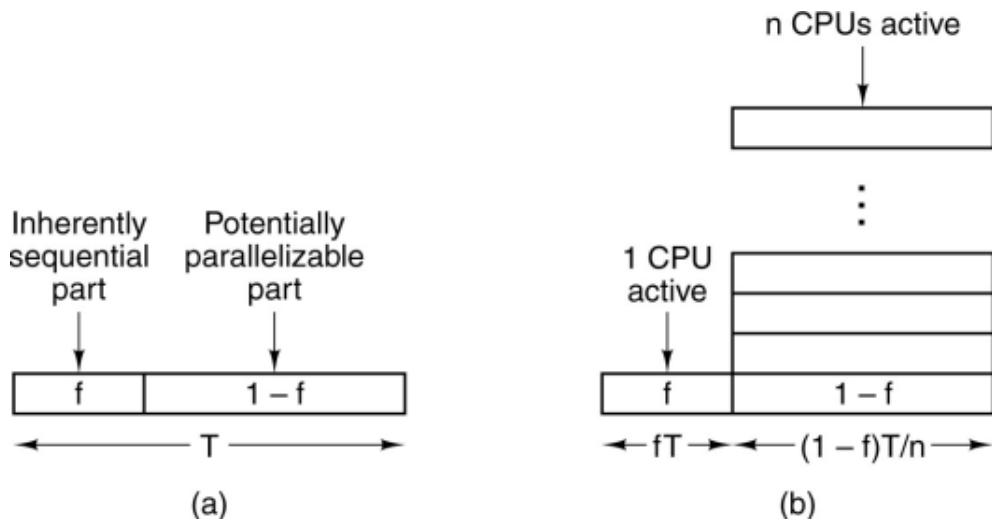


Figure 4.2: Sia  $P$  un programma che gira in tempo  $T$  su una CPU, con  $f$  = la frazione di  $T$  dovuta a codice sequenziale, e  $(1-f)$  la frazione di  $T$  dovuta a codice parallelizzabile.

**Note:-**

Il tempo di esecuzione dovuto alla parte parallelizzabile, passa da  $(1-f)T$  a  $(1-f)T/n$  se sono disponibili  $n$  processori. Questa è nota come *legge di Amdahl*

$$\begin{aligned} \text{speed-up} &= \frac{fT + (1 - f)T}{fT + (1 - f)T/n} = \frac{n[fT + (1 - f)T]}{nfT + (1 - f)T} \\ &= \frac{nT}{T(1 + nf - f)} = \frac{n}{1 + (n-1)f} = \text{legge di Amdahl} \end{aligned}$$

Figure 4.3: La legge di Amdahl.

- Se si vuole ottenere uno speed-up perfetto, pari al numero di CPU usate,  $f$  deve valere 0, ma ciò è impossibile.
- Spesso il miglioramento è sub-lineare.

I problemi del parallelismo esplicito sono generalmente due:

1. La quantità limitata di parallelismo nel codice dei programmi (problema software).
2. Gli elevati costi delle comunicazioni tra processori e memoria (problema hardware).

**Note:-**

Per molte applicazioni avere uno speed-up, seppur limitato, è comunque accettabile, perché:

- La presenza di più processori aumenta l'affidabilità del sistema.
- Servizi che per loro natura sono forniti su ampia scala geografica, devono essere implementati con una architettura distribuita. Se il sistema fosse centralizzato in un unico nodo, l'accesso di tutte le richieste a quest'unico nodo costituirebbe probabilmente un collo di bottiglia in grado di rallentare enormemente il servizio fornito.

Ci sono tre tipi di architetture parallele esplicite:

- Multi-threading (a).
- Sistemi a memoria condivisa (b, c).
- Sistemi a memoria distribuita (d, e).

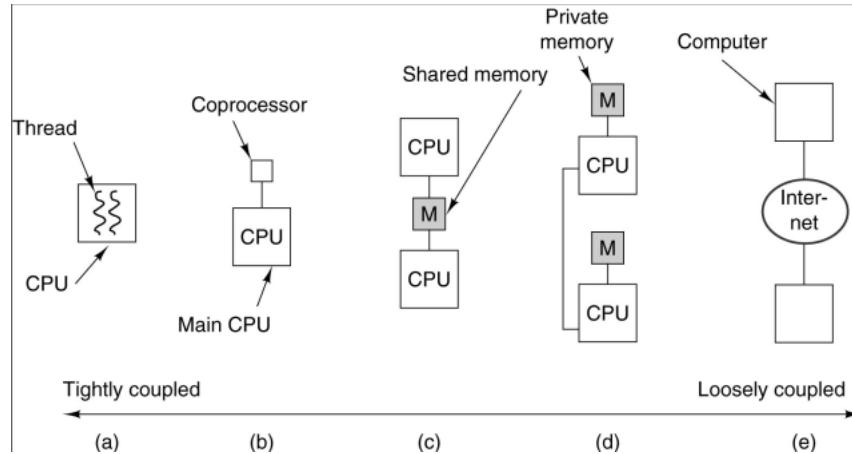


Figure 4.4: Vari tipi di architetture esplicitamente parallele.

## 4.2 Multi-Threading

### 4.2.1 Introduzione

#### Definizione 4.2.1: Architettura Multi-Threaded

Una CPU multi-threaded è una architettura parallela un po' particolare, in quanto il multi-threading è realizzato con un'unica CPU (sempre single core), ma porta il programmatore a concepire e sviluppare le sue applicazioni come formate da un insieme di programmi che possono essere eseguiti in parallelo: i thread.

#### Note:-

Se questi programmi vengono fatti girare su una CPU che supporta il multi-threading ne sfrutteranno le caratteristiche architetturali.

L'idea del multi-threading nasce dalla constatazione di un problema di fondo presente in qualsiasi CPU pipelined: un cache miss produce una "lunga" attesa necessaria per recuperare l'informazione mancante in RAM. Se non c'è un'altra istruzione indipendente da poter eseguire, o se non è implementato lo scheduling dinamico della pipeline, la pipeline va in stall. Una soluzione per non sprecare inutilmente cicli di clock in attesa del dato mancante è il multithreading: permettere alla CPU di gestire più peer-thread allo stesso tempo: se un thread è bloccato la CPU ha ancora la possibilità di eseguire istruzioni di un altro thread, in modo da tenere le varie unità funzionali comunque occupate.

#### Osservazioni 4.2.1

- Per implementare il multithreading, la CPU deve poter gestire lo stato della computazione di ogni singolo thread.
- Ci deve essere un Program Counter (PC) e un set di registri separato per ciascun thread.
- Il thread switch deve essere più efficiente del process switch.

### 4.2.2 Tipi di Multi-Threading

Esistono due tecniche di base per il multi-threading:

- *Fine-grained multi-threading*.
- *Coarse-grained multi-threading*.

#### Definizione 4.2.2: Fine-grained Multi-Threading

Lo switching tra i vari peer-thread avviene ad ogni istruzione, indipendentemente dal fatto che l'istruzione del thread in esecuzione abbia generato un cache miss.

Lo *scheduling* tra i vari peer-thread avviene secondo una politica round robin e la CPU deve essere in grado di effettuare lo switch praticamente senza overhead (che sarebbe inaccettabile).

#### Note:-

Se vi è un numero sufficiente di peer-thread, è possibile che ve ne sia sempre almeno uno non in stall, e la CPU può essere mantenuta sempre attiva.

#### Osservazioni 4.2.2

- Lo stalling della CPU può anche essere dovuto a una dipendenza o a un branch, poiché l'ILP dinamico non può sempre evitarlo.
- Invece, con il Fine-grained multithreading, in un'architettura pipelined, se:
  - La pipeline ha  $k$  stage.
  - Ci sono almeno  $k$  peer-thread da eseguire.
  - La CPU esegue lo switch tra thread a ogni ciclo di clock.
- Non ci può essere mai più di un'istruzione per thread nella pipeline in un dato istante, le dipendenze sui dati vengono risolte automaticamente e la pipeline non va mai in stall (se un cache miss può essere gestito in un numero di cicli di clock inferiore o uguale al numero di thread in esecuzione).

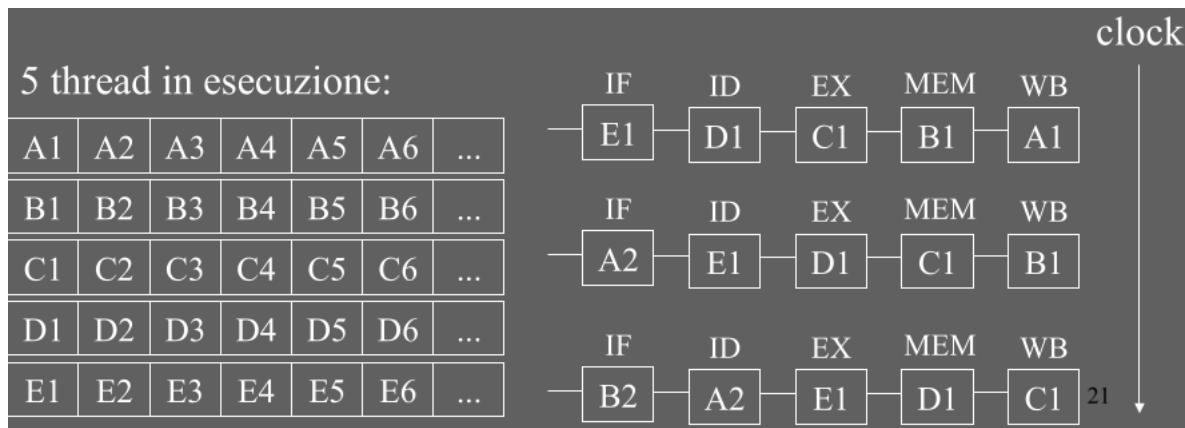


Figure 4.5: Fine-grained multithreading in una CPU con pipeline a 5 stadi.

#### Note:-

Lo scheduling Fine-grained fa sì che un thread venga rallentato anche quando potrebbe proseguire l'esecuzione perché non sta generando alcuno stall (per via del fatto che vada eseguito un context switch a ogni istruzione). Inoltre potrebbero esserci meno thread che stage della pipeline, per cui non è facile mantenere la CPU sempre occupata.

**Definizione 4.2.3: Coarse-grained Multi-Threading**

Lo switch avviene solo quando il thread in esecuzione genera uno stall, provocando così lo spreco di un ciclo di clock.

Viene effettuato lo switch a un altro thread. Quando anche questo genererà uno stall verrà schedulato un terzo thread (o eventualmente si tornerà al primo) e così via.

**Note:-**

Questo approccio spreca potenzialmente più cicli di clock rispetto al Fine-grained, perché comunque lo switch avviene solo se si è verificato uno stall. Ma se ci sono pochi thread attivi questi possono essere sufficienti per tenere la CPU occupata.

**Fine-grained vs. Coarse-grained:**

- Non è pensabile che lo switch tra thread possa avvenire senza perdite di tempo.
- Se le istruzioni dei vari thread non generano stall molto frequentemente uno scheduling Coarse-grained è più vantaggioso (perché il Fine-grained, ogni ciclo di clock, paga un overhead limitato, ma non nullo).
- Fine-grained → Sistema operativo time-sharing.
- Coarse-grained → Sistema operativo multitasking.

**Definizione 4.2.4: Medium-grained Multi-Threading**

Una via di mezzo tra il fine- e il coarse- grained multithreading consiste nell'eseguire lo switch tra thread solo quando quello in esecuzione sta per eseguire una istruzione che potrebbe generare uno stall di lunga durata, come ad esempio una load (che potrebbe richiedere un dato non in cache), o un branch (perché anche l'istruzione nel caso il branch venisse preso potrebbe non trovarsi nelle cache L1, L2, L3).

L'istruzione viene avviata, ma il processore esegue in ogni caso lo switch a un altro thread.

**Note:-**

In questo modo non si spreca neanche il ciclo di clock dovuto all'eventuale stall dell'istruzione eseguita (come accade nel Coarse-grained).

**Domanda 4.1**

Ma come si fa a sapere a quale thread appartiene una qualsiasi istruzione nella pipeline?

- Nel caso del Fine-grained Multi-Threading l'unico modo è di attaccare un *thread identifier* a ogni istruzione. Per esempio l'ID univoco associato a quel thread all'interno dell'insieme di peer-thread a cui appartiene.
- Nel caso del Coarse-grained Multi-Threading si può adottare la stessa soluzione oppure si può anche svuotare la pipeline a ogni thread switch (in questo modo le istruzioni di un solo thread alla volta sono nella pipeline e quello sa sempre a quale thread appartengono).

**Note:-**

Ciò ha senso solo se lo switch avviene a intervalli molto maggiori del tempo necessario a svuotare la pipeline. Inoltre, le istruzioni dei vari thread, devono essere, per quanto possibile, tutte contemporaneamente nella cache delle istruzioni, altrimenti ogni context switch tra thread produce un cache miss e si perde qualsiasi vantaggio nell'utilizzare i thread.

**4.2.3 Simultaneous Multi-Threading**

Le moderne *architetture superscalari*, multiple issue e a scheduling dinamico della pipeline danno la possibilità di sfruttare contemporaneamente il parallelismo insito nelle istruzioni di un programma (ILP) con il parallelismo insito in un insieme di peer-threads: il Thread Level Parallelism (TLP).

$$\text{ILP} + \text{TLP} = \text{SMT} \text{ (Simultaneous Multi-Threading)}$$

**Note:-**

Le ragioni per implementare SMT risiedono nell'osservazione che le moderne CPU multiple issue hanno più unità funzionali di quante siamo mediamente sfruttabili dal singolo thread in esecuzione.  
Sfruttando il register renaming e lo scheduling dinamico istruzioni appartenenti a thread diversi possono essere eseguite insieme.

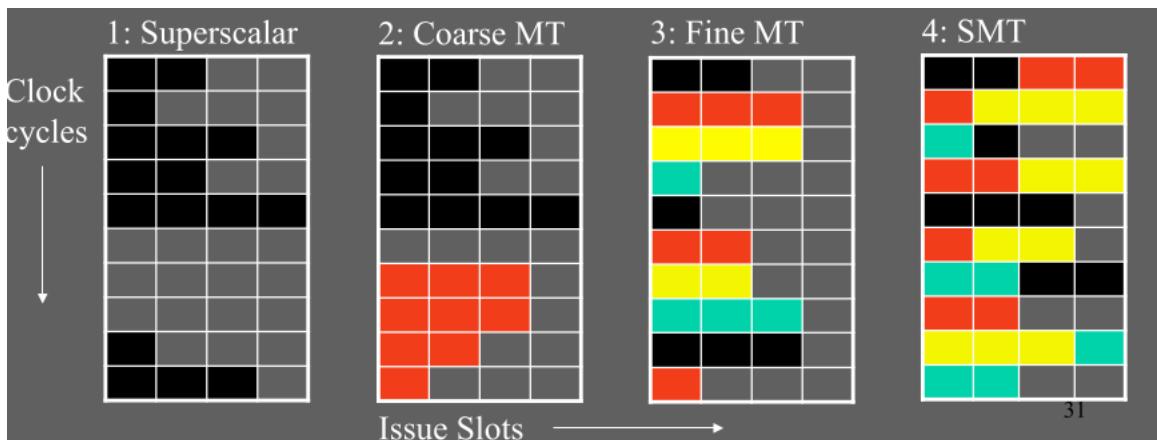


Figure 4.6: Confronto tra processori.

**Osservazioni 4.2.3**

- Anche nel SMT non si riesce sempre ad avviare il massimo numero possibile di istruzioni per ciclo di clock a causa del numero limitato di unità funzionali e di stazioni di prenotazione disponibili, della capacità della cache di istruzioni di fornire le istruzioni dei vari thread e del numero di thread presenti.
- SMT può essere adeguatamente supportato solo se sono disponibili registri rinominabili in abbondanza.
- Nel caso di una CPU che supporti la ridenominazione sarà importante avere un ROB distinto per ogni thread in modo da gestire in maniera indipendente i commit.

**Il Multi-Threading Intel****Definizione 4.2.5: Hyperthreading**

Viene supportata l'esecuzione di due thread in modalità SMT.

**Osservazioni 4.2.4**

- Dal punto di vista del Sistema Operativo, una CPU multithreaded è vista come due CPU che condividono cache e RAM: dunque due applicazioni che condividono lo stesso spazio di indirizzamento, possono essere eseguite in parallelo nei due thread.
- Siccome due thread possono usare contemporaneamente la CPU, occorre adottare delle strategie per fare in modo che entrambi i thread possano utilizzare le varie risorse della CPU.

**Ci sono 4 strategie ideate da Intel:**

- **Risorse duplicate:** quelle necessarie a gestire l'esecuzione dei due thread (due program counter e due banchi di registri separati).

- *Risorse partizionate equamente tra i due thread:* la coda delle microistruzioni e il ROB.
- *Risorse condivise dinamicamente:* un thread può occupare quante linee di cache vuole, eventualmente rallentando l'altro thread.
- *Risorse condivise dinamicamente:* lo scheduler che invia le uops alle varie stazioni di prenotazione.

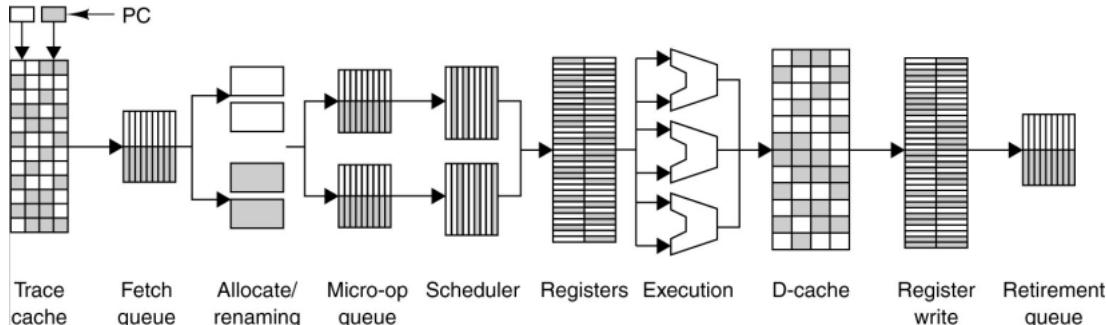


Figure 4.7: Risorse in una CPU Intel.

#### Brevi note storiche:

- Intel abbandona la tecnologia Hyperthreading nel passaggio ai nuovi processori dual core (non supportavano il multithreading).
- L'Hyperthreading è stato reintrodotto a partire dal 2008 con l'architettura Nehalem.
- IBM Power 7 (2010) implementò SMT con 4 thread contemporaneamente attivi.
- I processori UltraSPARK, da T2 in poi, supportano il Fine-grained multithreading con 8 thread per core. L'UltraSPARK T1 solo 4 thread per core.

## 4.3 Multiprocessori a Memoria Condivisa

### 4.3.1 Introduzione

#### Definizione 4.3.1: Multiprocessore

Un *multiprocessore* è un'architettura con più CPU che condividono la stessa memoria primaria.

In un sistema multiprocessore tutti i processi che girano sulle varie CPU condividono un unico spazio di indirizzamento logico, mappato su una memoria fisica che può però anche essere distribuita tra i vari banchi di memoria in modi diversi. Ogni processo può leggere e scrivere un dato in memoria specificando un indirizzo usato da LOAD e STORE, e la comunicazione tra processi avviene attraverso la memoria condivisa.

#### Note:-

È responsabilità dell'hardware del sistema fare in modo che tutte le CPU possano vedere e usare la stessa memoria principale.

#### Osservazioni 4.3.1

- Siccome tutte le CPU vedono lo stesso spazio di indirizzamento, è sufficiente una copia del sistema operativo.
- Quando un processo termina o va in wait per qualche ragione, il S.O. può cercare nella coda di ready

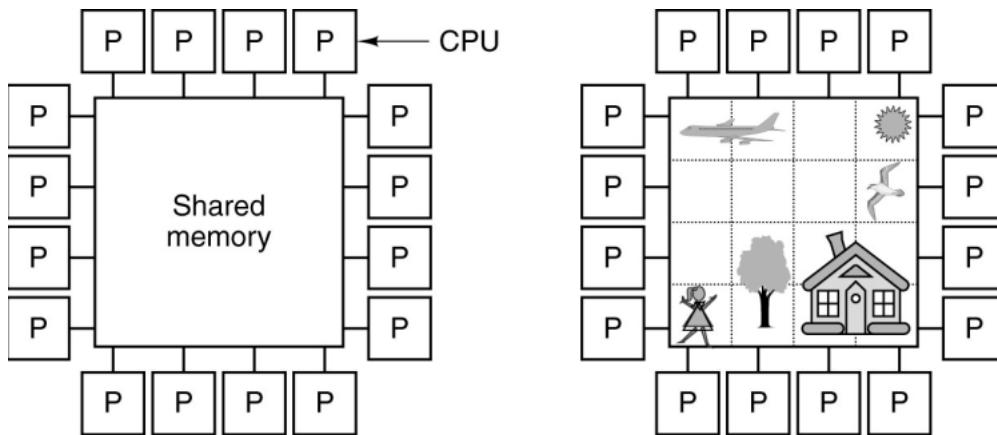


Figure 4.8: Schema di base di un multiprocessore.

un altro processo a cui dare la CPU idle.

- Al contrario, nei sistemi a memoria non condivisa, ogni CPU deve far girare la propria copia del sistema operativo, e i processi possono comunicare solo attraverso lo scambio di messaggi.
- Il problema di fondo dei sistemi multiprocessore a memoria condivisa è la memoria stessa, che è difficile da far funzionare in maniera efficiente quanto più è alto il numero dei processori coinvolti.

#### Domanda 4.2

Come funzionano i sistemi multiprocessore?

- Tutti i moderni SO (Windows, Solaris, Linux, MacOS) prevedono in particolare la cosiddetta multielaborazione simmetrica (symmetric multiprocessing, SMP), in cui (tralasciando un po' di cose) uno scheduler gira su ciascun processore.
- I processi “ready to run” possono essere inseriti in un'unica coda, vista da ciascun scheduler, oppure vi può essere una coda “ready to run” separata per ciascun scheduler/processore.
- Quando lo scheduler di un processore si attiva, sceglie uno dei processi “ready to run” e lo manda in esecuzione sul proprio processore.
- Un aspetto importante dei sistemi multiprocessore è il bilanciamento del carico.
- Non ha infatti senso avere un sistema con più CPU se poi i vari processi da eseguire non sono distribuiti più o meno omogeneamente tra i vari processori.
- Nel caso di un'unica coda ready to run, il bilanciamento del carico è solitamente automatico: quando un processore è inattivo, il suo scheduler prenderà un processo dalla coda comune e lo manderà in esecuzione su quel processore.
- I SO moderni predisposti per l'SMP usano però spesso una coda separata per ciascun processore in modo da non doversi sincronizzare per accedere all'unica coda di ready quando devono inserire o prelevare un processo dalla coda. Esiste allora un esplicito meccanismo di bilanciamento del carico che può prendere un processo in attesa sulla coda di un processore sovraccarico e spostarlo nella coda di un processore scarico<sup>1</sup>.

<sup>1</sup>Per esempio, in Linux SMP meccanismo di bilanciamento del carico si attiva ogni 200 millisecondi, e ogni qualvolta si svuota la coda di un processore.

### 4.3.2 UMA

Esistono sostanzialmente due classi di architetture multiprocessore:

- **UMA**.
- **NUMA**.

**Note:-**

Una terza classe, detta **COMA** (Cache Only Memory Architecture), sebbene teoricamente promettente non è stata mai implementata in maniera soddisfacente.

**Definizione 4.3.2: UMA**

Uniform Memory Access (UMA) è un tipo di architettura in cui tutti i processori condividono un'unica memoria primaria centralizzata e quindi ogni CPU ha lo stesso tempo di accesso alla memoria.

**Note:-**

Questi sistemi vengono anche chiamati SMP (Symmetric Shared-Memory Multiprocessor)

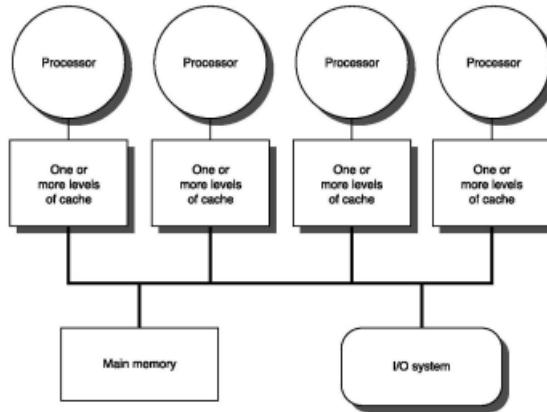


Figure 4.9: Schema UMA.

**Osservazioni 4.3.2**

- I sistemi multiprocessore UMA costituiscono il primo tipo di sistemi multiprocessore, e nella sua forma più semplice un sistema UMA prevede un unico bus su cui si affacciano almeno due CPU e una memoria (condivisa da tutti i processori).
- Quando una CPU vuole leggere una locazione di memoria verifica prima che il bus sia libero, invia la richiesta al modulo di interfaccia della memoria e attende sul bus che arrivi il valore richiesto.
- La memoria condivisa può però facilmente diventare un collo di bottiglia per le prestazioni del sistema, visto che tutti i processori devono sincronizzarsi sull'uso di un singolo bus e memoria.

**Note:-**

I moderni processori multi-core sono semplici architetture UMA in cui la prima cache condivisa da tutti i core (solitamente L3) costituisce un canale di comunicazione più efficiente della RAM: se un core modifica un dato un altro core potrà vedere la modifica in una linea di L3 anziché dover arrivare fino alla linea di RAM corrispondente.

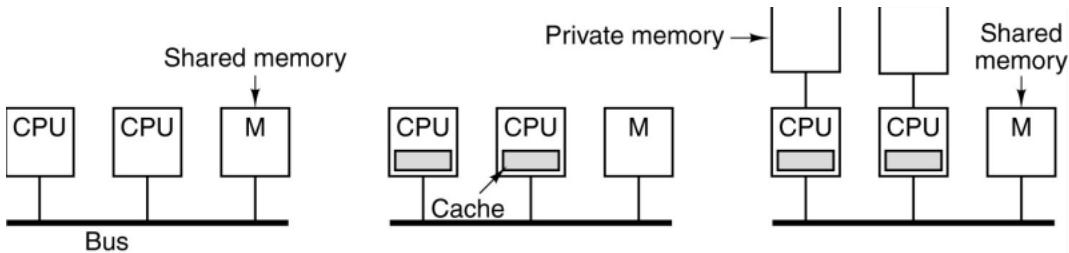


Figure 4.10: Ogni processore dispone di una sua memoria privata.

**Corollario 4.3.1 Coerenza della Cache**

La visione che ogni processore ha della memoria primaria passa attraverso la propria cache, e due processori possono vedere valori diversi per la stessa locazione.

**Note:-**

Sono stati proposti vari *protocolli di coerenza della cache*, e tutti hanno come scopo quello di evitare che versioni differenti della stessa linea di RAM possano essere contemporaneamente presenti in due o più cache (un problema noto come *false sharing*).

**Definizione 4.3.3: Snooping del Bus**

A livello hardware il controller di ogni cache è in grado di monitorare, sul bus, tutte le richieste di RAM che provengono dalle altre CPU.

**Corollario 4.3.2 Write Through**

Una cache 1:

- *Read miss*: il cache controller della CPU preleva la linea mancante dalla RAM e la mette nella cache. Ulteriori letture dello stesso dato avverranno nella cache (quindi, le successive letture saranno dei read hit).
- *Write miss*: il dato modificato viene direttamente scritto in RAM: la linea contenente il dato non viene prima caricata nella cache locale.
- *Write hit*: la cache line viene aggiornata e l'aggiornamento viene anche propagato alla RAM.

Nel frattempo, un'ipotetica cache 2:

- *Read miss*: la cache 2 vede cache 1 prelevare una linea dalla memoria, ma non fa nulla (in caso di read hit la cache 2 non se ne accorge nemmeno).
- *Write miss/hit*: cache 2 verifica se ha una copia del dato modificato: in caso negativo, non fa nulla. Se però il dato è presente, la linea che lo contiene viene marcata come invalida nella cache 2.

**Osservazioni 4.3.3**

- Poiché tutte le cache sorvegliano tutte le operazioni compiute dalle altre cache, quando in una cache viene modificato un dato, la modifica viene fatta nella cache stessa (se c'è), in RAM, e in più la "vecchia" linea viene marcata come invalida da tutte le altre cache.
- In questo modo nessuna cache può contenere dati inconsistenti rispetto alle altre cache.
- Il problema fondamentale dei protocolli *write through based* è l'inefficienza perché ogni operazione di write viene propagata alla RAM e il bus di comunicazione può diventare un collo di bottiglia.

- Per limitare questo problema non tutte le write vengono immediatamente propagate in RAM, per cui un bit nella cache viene settato per indicare che la linea nella cache è aggiornata rispetto a quella nella RAM.

#### Definizione 4.3.4: Protocollo MESI

Il protocollo di tipo write back più diffuso, usato dai processori moderni. Ogni entry può essere in uno di 4 diversi stati:

- *Invalid*: l'entry nella cache non contiene dati validi.
- *Shared*: più cache possono contenere la linea e la memoria RAM è aggiornata.
- *Exclusive*: nessun'altra cache contiene la linea e la memoria cache è aggiornata.
- *Modified*: la linea è valida, la memoria RAM contiene un valore vecchio, non esistono altre copie.

#### Nel protocollo MESI:

- La prima volta che una linea viene letta nella cache di CPU 1 viene marcata E: exclusive, poiché è l'unica cache a contenere quella linea.
- Successive letture del dato da parte della stessa CPU useranno la copia in cache, e non impegnano il bus.
- Se un'altra CPU legge la stessa linea, lo snooper della prima CPU se ne accorge e annuncia sul bus che anche la prima CPU ha una copia della linea. Entrambe le entry nelle cache vengono marcate S: Shared.
- Successive letture del dato da parte della prima CPU o della seconda CPU avverranno nelle rispettive cache, e non useranno il BUS.
- Se la seconda CPU modifica una linea marcata S, invia sul bus un segnale di invalidazione della linea, in modo che le altre CPU possano invalidarla nella loro cache. La linea viene marcata M: modified, e non viene scritta in RAM.
- Notiamo che se la linea era marcata E, il segnale di avviso alle altre CPU non è necessario, perché non esistono altre copie della linea in altre cache.

#### Domanda 4.3

Cosa succede se una terza CPU tenta di leggere la stessa linea?

- Lo snooper della seconda CPU se ne accorge, sa di possedere l'unica copia valida della linea, per cui invia un segnale su bus per avvertire la terza CPU di aspettare, mentre la copia valida viene usata per aggiornare memoria.
- A fine aggiornamento la terza CPU può prelevare la linea richiesta, e nelle due cache la linea viene marcata S, shared.
- Se a questo punto la seconda CPU rimodifica la linea, nella sua cache, invierà di nuovo un segnale di invalidazione sul bus, e tutte le altre copie vengono marcate come I: invalid. La linea nella cache della seconda CPU è di nuovo marcata M: modified.
- Se la prima CPU cerca di scrivere un dato nella linea, la seconda CPU vede il tentativo di write e invia un segnale sul bus per dire alla prima CPU di aspettare mentre aggiorna la linea in memoria. Alla fine, la seconda CPU marca la propria copia della linea come invalida, perché sa che un'altra CPU sta per modificarla.
- Se si sta usando una politica *write-allocate*, la linea sarà caricata nella cache della prima CPU e marcata M. In caso contrario, la write ha effetto direttamente in RAM, e la linea continua a non essere in nessuna cache.

**Osservazioni 4.3.4**

- Anche usando un protocollo come il MESI, l'uso di un bus singolo su cui si affacciano tutti i processori che comunicano con la RAM limita la dimensione di sistemi multiprocessore UMA ad un massimo che di solito si indica in 32 CPU.
- Per andare al di là di questo limite, è necessario usare un diverso sistema di interconnessione tra le CPU e la RAM. Lo schema più semplice per connettere n CPU a k memorie è a commutatori incrociati (*crossbar switch*), un sistema simile a quello usato per decenni nelle centrali telefoniche.

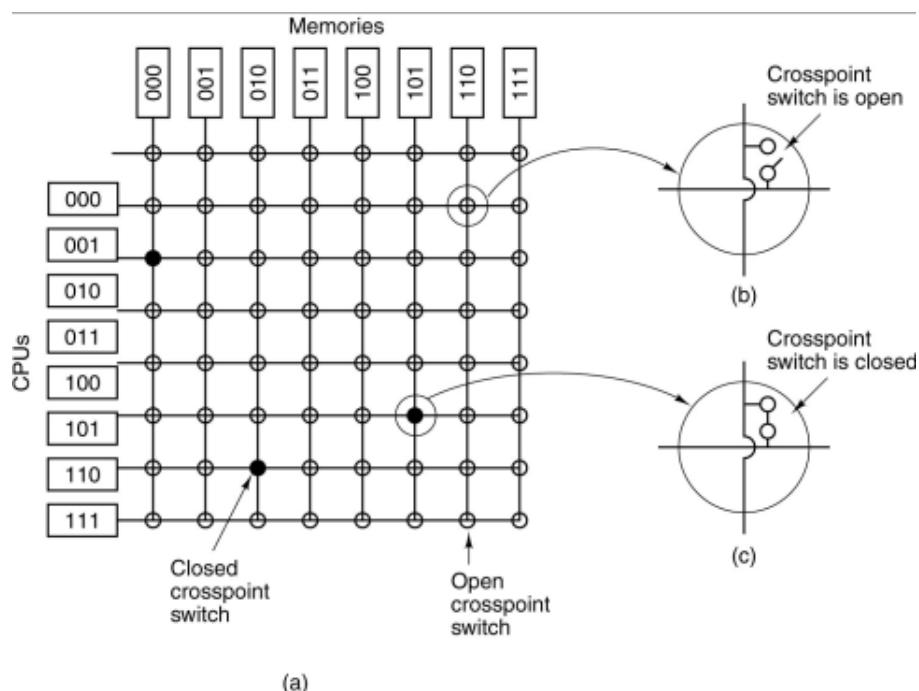


Figure 4.11: Nell'esempio, tre switch sono chiusi, e connettono le coppie CPU- memoria (001-000), (101-101) e (110-010).

**Definizione 4.3.5: UMA a Crossbar Switch**

È possibile configurare gli switch in modo che ciascuna CPU possa connettersi a ciascun banco di memoria. Il numero di switch necessari per realizzare questo schema però cresce quadraticamente col numero di CPU (memorie) coinvolte:  $n$  CPU e  $4n$  memorie richiedono  $n^2$  switch. La cosa è accettabile per sistemi di media grandezza (vari sistemi multiprocessore della Sun usano questo schema), ma certamente non è usabile in un sistema con 256 CPU (sarebbero necessari  $256^2$  switch).

**Note:-**

Quando il numero di CPU è alto, si può usare un sistema basato su switch bidirezionali con due ingressi e due uscite: in questi switch ciascun ingresso può essere rediretto su ciascuna uscita.

**Nei sistemi UMA che usano switch bidirezionali i messaggi scambiati tra CPU e memoria sono fatti di quattro parti:**

- *Module*: quale memoria usare, quale CPU richiede il dato.
- *Address*: specifica un indirizzo all'interno del modulo di memoria.

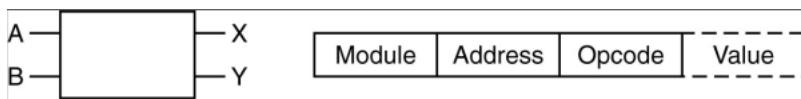


Figure 4.12: Sistema di switch bidirezionali.

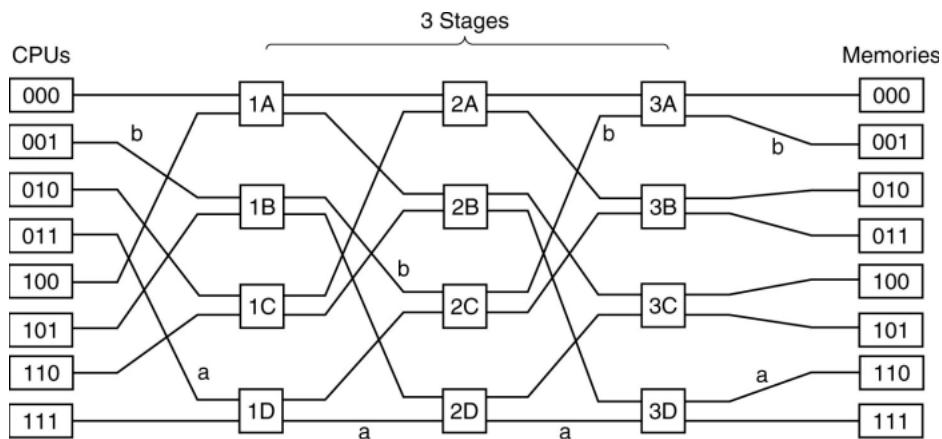


Figure 4.13: Switch bi-???

- **Opcode**: l'operazione da eseguire.
- **Value** (opzionale): il valore da scrivere nel caso di write.

**Note:-**

Lo switch può essere programmato in modo da analizzare il Module e determinare su quale output instradare il messaggio.

Figure 4.14: Gli switch  $2 \times 2$  possono essere usati in molti modi per costruire reti a commutazione a più stadi. Un semplice esempio è il modello di rete omega.

- Nell'esempio, 8 CPU sono connesse a 8 memorie, usando in tutto 12 switch in tre stadi. In generale,  $n$  CPU e  $n$  memorie richiedono  $\log_2 n$  stadi, con  $\frac{n}{2}$  switch per stadio, per un totale di  $(\frac{n}{2})\log_2 n$  switch: molto meglio che nel caso dei crossbar switch ( $n^2$ ).
- La CPU 011 vuole leggere un dato nel modulo di RAM 110. La CPU invia una READ allo switch 1D con Module = 110 - 011.
- Lo switch preleva il bit più significativo (quello più a sinistra) e lo usa per l'instradamento: 0 instrada la richiesta sull'output superiore, 1 instrada la richiesta sull'output inferiore.
- Lo switch 2D si comporta allo stesso modo: analizza il secondo bit più significativo (quello centrale) e instrada la richiesta verso 3D.
- Infine, il bit meno significativo viene usato per l'ultimo instradamento, verso il modulo 110 (percorso a nella figura).
- A questo punto, il dato letto deve essere reinstradato alla CPU 011: viene usato il suo "indirizzo", leggendo però i bit da destra verso sinistra.

- Allo stesso tempo, la CPU 001 vuole eseguire una WRITE nel modulo 001. Avviene un processo simile a quello visto (percorso b nella figura). Siccome i percorsi a e b non usano gli stessi switch, le due richieste possono procedere in parallelo.

**Note:-**

Al contrario di quello che accade con le reti che usano crossbar switch, le reti omega sono *reti bloccanti*: non tutte le sequenze di richieste possono essere servite contemporaneamente.

### 4.3.3 NUMA

I sistemi UMA a bus singolo sono limitati dal numero di processori, e per connettere più processori è necessario dell'hardware comunque costoso. Allo stato attuale, non è conveniente costruire sistemi UMA con più di qualche centinaio di processori. Per costruire sistemi più grandi è necessario accettare un compromesso: che non tutti i moduli di memoria abbiano lo stesso tempo di accesso rispetto a ciascuna CPU.

**Definizione 4.3.6: NUMA**

Non Uniform Memory Access (NUMA) è un tipo di architettura in cui la memoria fisica è distribuita tra le varie CPU e quindi i tempi di accesso ai dati variano a seconda che siano nella RAM locale o in una remota.

**Note:-**

Questi sistemi vengono anche chiamati DSM (Distributed Shared-Memory).

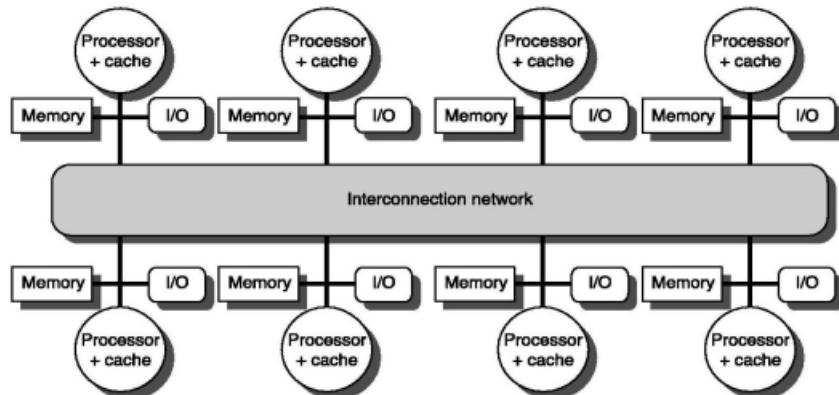


Figure 4.15: Schema NUMA.

### Osservazioni 4.3.5

- Come per i UMA, nei sistemi NUMA tutte le CPU vedono lo stesso spazio di indirizzamento ma, ogni processore è dotato di una propria memoria locale, vista anche da tutti gli altri processori.
- Al contrario dei sistemi UMA quindi, nei sistemi NUMA l'accesso ai moduli di memoria locale è più veloce dell'accesso ai moduli di memoria remota.
- Come conseguenza, i programmi scritti per sistemi UMA possono comunque girare senza dover apportare alcun cambiamento su macchine NUMA, possibilmente con prestazioni diverse a causa dei diversi tempi di accesso ai vari moduli remoti di RAM.
- Poiché le macchine NUMA hanno uno spazio di indirizzamento logico unico visto da tutte le CPU, mentre la memoria fisica è in realtà suddivisa tra i vari processori, emerge il concetto di memoria locale e remota.

**Esistono due tipi di sistemi NUMA:**

- *Non-Caching NUMA (NC-NUMA)*.
- *Cache-Coherent NUMA (CC-NUMA)*.

#### Definizione 4.3.7: NC-NUMA

In un sistema NC-NUMA i processori non hanno cache locale. Ogni accesso alla memoria è gestito da una MMU modificata, che controlla se la richiesta è diretta alla memoria locale o a un modulo remoto, nel qual caso la richiesta viene instradata al nodo che contiene il dato richiesto.

#### Note:-

I programmi che usano dati remoti risulteranno più lenti che se i dati fossero memorizzati nella memoria locale.

#### Osservazioni 4.3.6

- Nei sistemi NC-NUMA il problema della coerenza della cache è automaticamente risolto perché non c'è nessuna forma di caching: ogni dato della memoria è presente esattamente in una locazione ben precisa.
- Rimane il problema dell'inefficienza dell'accesso alla memoria remota. Per questo, le macchine NC-NUMA possono usare software elaborato per spostare le pagine di memoria da un modulo all'altro in modo da minimizzare gli accessi remoti e massimizzare le prestazioni.
- Un page scanner può attivarsi ogni pochi secondi, esaminare le statistiche sull'indirizzamento della memoria, e spostare le pagine da un modulo all'altro per cercare di migliorare le prestazioni.
- Nei sistemi NC-NUMA, ogni processore può avere anche una memoria locale privata e una cache, e solo i dati privati del processore (ossia quelli nella memoria locale privata) possono risiedere nella cache. Questa soluzione aumenta ovviamente le prestazioni di ciascun processore.

#### Definizione 4.3.8: CC-NUMA

Aggiungere il caching diminuisce ovviamente i tempi di accesso ai dati remoti, ma introduce il problema della coerenza della cache. Un modo di garantire la coerenza sarebbe ovviamente lo snooping sul bus di sistema, ma questa tecnica diventa troppo inefficiente oltre un certo numero di CPU, ed è comunque troppo difficile da implementare nei sistemi che non usano un bus comune di interconnessione.

L'approccio più usato per costruire sistemi CC-NUMA con molte CPU assicurando la coerenza della cache è noto come schema o protocollo *directory-based* (multiprocessor). L'idea di base è di associare ad ogni nodo del sistema una directory per le linee della sua RAM: un database che regista in quale cache di quale nodo si trova ogni linea, e qual è il suo stato.

#### Osservazioni 4.3.7

- Quando viene indirizzata una linea di RAM, la directory del nodo a cui quella linea appartiene viene interrogata per sapere se la linea si trova in qualche cache e se questa sia stata modificata rispetto alla copia in RAM.
- Poiché una directory viene interrogata ogni volta che una istruzione accede la corrispondente memoria, deve essere implementata con un hardware molto veloce, per esempio una memoria associativa.
- Come esempio di protocollo directory based, consideriamo un sistema con 256 nodi, ognuno formato da una CPU e una RAM locale da 16 MB. Ci sono in tutto  $232 = 4$  GB di RAM, e ogni nodo contiene 218 linee da 64 byte ciascuna ( $218 \times 26 = 224 = 16$  MB). Lo spazio di indirizzamento è unico, col nodo zero che contiene la memoria con indirizzi da 0 a 16 MB, il nodo 1 la memoria con indirizzi da 16 a 32 MB, e così via. Un indirizzo fisico è quindi scritto su 32 bit (gli 8 bit più significativi indirizzano di fatto il numero del nodo a cui appartiene il banco di RAM che contiene il dato indirizzato), i successivi

18 bit indicano la linea indirizzata all'interno del banco da 16 MB e i restanti 6 bit meno significativi indirizzano il byte all'interno della linea).

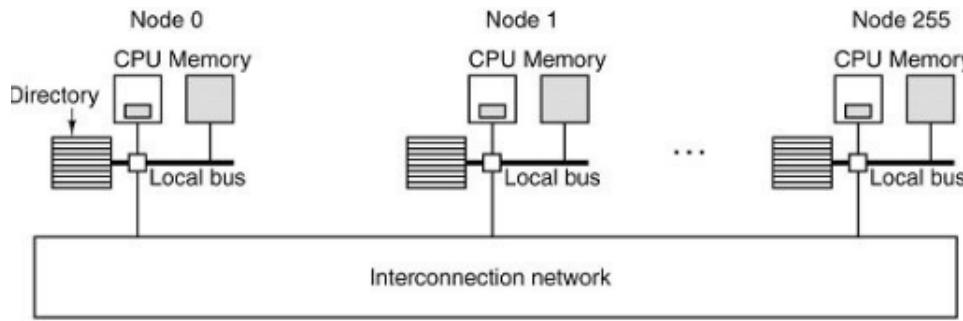


Figure 4.16: Esempio di CC-NUMA.

In un sistema reale, l'architettura **directory-based**:

- Le linee di RAM possono essere contemporaneamente nella cache di più nodi.
- Si tiene traccia del fatto che una cache line sia stata modificata o meno e si possono limitare le comunicazioni tra CPU e memorie.
- Se una cache line non è stata modificata la linea originale in RAM è ancora valida e una read da una CPU remota per quellls line può essere soddisfatta dalla RAM stessa, senza dover andare a recuperare la linea della cache che ne contiene una copia.

#### 4.3.4 Sincronizzazione tra Processi

In un sistema monoprocesso, i vari processi si sincronizzano fra loro usando opportune system call o costrutti del linguaggio in uso: semafori, regioni critiche condizionali, monitors. Questi meccanismi di sincronizzazione sono costruiti a partire da opportune primitive di sincronizzazione hardware: spesso una istruzione macchina non interrompibile in grado di prelevare e modificare un valore, o di scambiare il contenuto di un registro e una cella di memoria. In un sistema multiprocesso abbiamo bisogno di primitive di sincronizzazione simili: i processi vedono un unico spazio di indirizzamento e la sincronizzazione deve avvenire sfruttando questo spazio comune, e non meccanismi a scambio di messaggi.

```

Shared var int lock = 0;           // lock inizialmente accessibile
int v = 1;
repeat
    while (v == 1) do exch(v, lock); //entry section
                                //critical section
                                // qui dentro lock = 1
    lock = 0;                   // exit section
                                // altro codice non in mutua esclusione // il lock è di nuovo libero
until false;

```

Figure 4.17: Spin lock, il processo cicla sulla variabile di lock fino a quando non la trova libera.

**Note:-**

In questo caso è presente un problema di **attesa attiva** per cui i test della variabile sprecano il quanto di tempo dei processi.

Tuttavia, in un ambiente in cui i processi che usano una variabile condivisa per sincronizzarsi possono girare su CPU diverse, l'atomicità delle istruzioni di sincronizzazione non è sufficiente.

#### Domanda 4.4

Su un processore, l'istruzione atomica sarà eseguita senza interruzioni, ma che succede sugli altri processori? Avrebbe senso disabilitare tutte le operazioni sulla memoria dal momento in cui una primitiva di sincronizzazione viene avviata a quando ha finito di modificare la variabile di sincronizzazione?

**Risposta:** Funzionerebbe, ma questo danneggerebbe le operazioni delle CPU non coinvolte nella sincronizzazione. La soluzione adottata in molti processori usa una coppia di istruzioni macchina eseguite una dopo l'altra. La prima istruzione cerca di scrivere in uno dei registri della CPU il valore della variabile condivisa. La seconda istruzione cerca di modificare la variabile condivisa, e restituisce un valore da cui si può capire se la coppia di istruzioni è stata eseguita in modo atomico, il che in un sistema multiprocessore, significa:

- Nessun altro processo attivo su qualsiasi processore ha modificato il valore della variabile usata per la sincronizzazione prima della terminazione della seconda istruzione della coppia.
- Durante l'esecuzione delle due istruzioni non si è verificato alcun context switch nel processore che le ha eseguite.

#### Esempio 4.3.1 (Sincronizzazione)

Sia  $[0(R1)]$  il valore della cella di memoria di indirizzo  $0(R1)$ , usata come variabile condivisa di sincronizzazione.

1) <b>LL R2, 0(R1)</b>	<b>// Load linked: scrive <math>[0(R1)]</math> in R2</b>
2) <b>SC R3, 0(R1)</b>	<b>// Store conditional: scrive il contenuto // di R3 in <math>0(R1)</math></b>

L'esecuzione delle due istruzioni rispetto a  $0(R1)$  è vincolata a ciò che accade tra l'esecuzione delle due istruzioni:

- Se  $0(R1)$  viene modificata da un altro processo prima della terminazione della SC, la SC fallisce, ossia:  $0(R1)$  non viene modificata dalla SC e viene scritto 0 in R3. Se invece la SC non fallisce, allora: R3 viene copiato in  $0(R1)$  e in R3 viene scritto 1.
- Analogamente, se la CPU su cui la coppia di istruzioni viene eseguita esegue un context switch tra le due istruzioni, la SC fallisce, con gli stessi effetti del punto 1.

Una exchange “atomica” tra R4 e  $0(R1)$  in un sistema multiprocessore (simile ad uno spin lock):

retry: OR R3, R4, R0	<b>// copy value of R4 in R3</b>
<b>LL R2, 0(R1)</b>	<b>// load linked: copy <math>[0(R1)]</math> in R2</b>
<b>SC R3, 0(R1)</b>	<b>// try to store value of R3 in <math>0(R1)</math></b>
BEQZ R3, retry	<b>// try again if SC failed</b>
MOV R4, R2	<b>// now put loaded value in R4</b>

Quando la MOV viene eseguita, R4 e  $0(R1)$  sono stati scambiati in modo “atomico”, e si è certi che il contenuto di  $0(R1)$  non è stato modificato da altri processi prima del completamento della exchange. Chiamiamo EXCH l'operazione eseguita da questo codice.

Una volta in possesso di un'operazione atomica EXCH, la si può usare per implementare spin locks: accessi ad una sezione critica che ogni processore cerca di acquisire ciclando sulla variabile di lock, che controlla appunto l'accesso mutuamente esclusivo.

Se le CPU non sono dotate di cache, la variabile di lock viene ovviamente tenuta in memoria: un processore cerca di acquisire il lock con una exchange atomica, e verifica se il lock è libero. Se le CPU sono dotate di cache allora è presente un meccanismo per mantenere la coerenza della cache, e ogni processore coinvolto nella sincronizzazione userà la copia della variabile di lock che ha portato nella propria cache. Questo rende il meccanismo dello spin lock più efficiente, perché tutte le operazioni dei vari processori che cercano di acquisire il lock operano sulle rispettive cache, in modo più efficiente. Il codice dello spin lock va però modificato: ogni processore esegue una read sulla copia in cache del lock, fino a quando non vede che il lock è libero. A questo punto tenta di acquisire il lock con la exchange atomica.



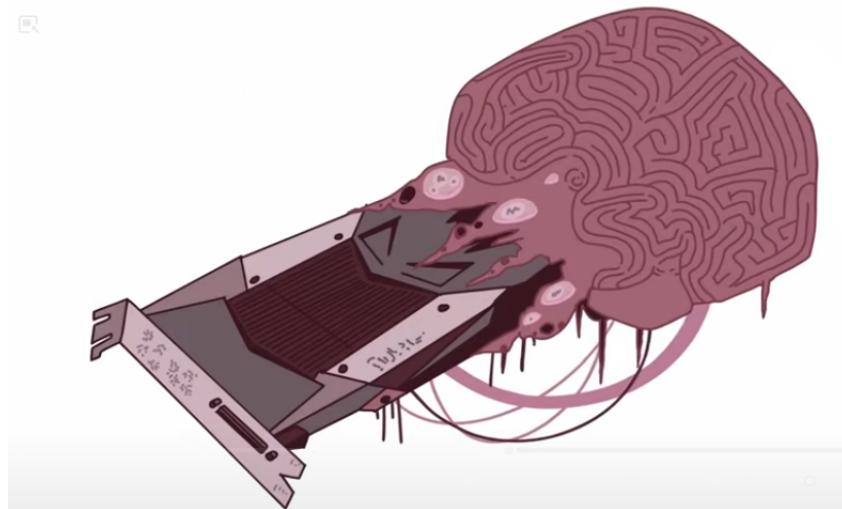
# 5

Quantum Computing



# 6

## GPU



### 6.1 Che cosa è una GPU?

La **GPU** (*Graphics Processing Unit*) è un processore altamente parallelo e multithread ottimizzato per l'elaborazione visuale. Nei PC moderni viene aggiunta alla CPU per creare un sistema *eterogeneo*. Benché la GPU sia il processore più potente all'interno di un normale PC, questa non può però essere da sola in quanto l'architettura che implementa l'elevato grado di parallelismo necessita di essere affiancata a una CPU che esegua al massimo delle prestazioni le operazioni sequenziali. A oggi le GPU non sono limitate all'elaborazione visuale, la consistente potenza di calcolo parallelo le ha rese appetibili per innumerevoli altre applicazioni (e. g. il deep learning). Inizialmente si effettuava il **GPGPU** (*General Purpose GPU*), ovvero l'utilizzo delle API della pipeline grafica per l'elaborazione di compiti non grafici. Attualmente viene definito **GPU Computing**, da quando **CUDA** è stato rilasciato, in quanto si è abbandonato l'utilizzo della pipeline grafica bensì si utilizza un linguaggio di programmazione parallelo.

**Definizione 6.1.1: CUDA**

CUDA (Compute Unified Device Architecture) è il modello di programmazione parallela scalabile e una piattaforma software per la GPU. Questo permette di programmare direttamente in un linguaggio simil C/C++.

**Note:-**

CUDA ha un approccio SIMD (o SIMT), quindi il programmatore scrive il programma per un thread e questo viene istanziato ed eseguito da molti thread in parallelo sui molteplici processori della GPU.

**6.1.1 Passaggio da GPU a CPU**

Le GPU eliminano tutto quello che è necessario per il parallelismo delle istruzioni e lasciano al programmatore la responsabilità di parallelizzare i suoi problemi (questo non è sempre possibile per cui GPU e CPU consistono in un'architettura ibrida). Inoltre sono presenti due problemi:

- *Divergence Memory Wall*.
- *Dark Silicon*.

**Definizione 6.1.2: Divergence Memory Wall**

Negli anni le differenze di velocità tra CPU e memoria principale sono diventate troppo importanti per essere ignorate. Le GPU eliminano questo gap cercando di ridurre al minimo l'interazione con la memoria principale.

**Definizione 6.1.3: Dark Silicon**

In una CPU possono stare attivi solo il 15-20% dei transistor totali contemporaneamente, altrimenti si rischia di bruciare l'intera CPU, per questo problema vengono pensate CPU che ottengono prestazioni meno elevate ma che magari permettono un'attivazione maggiore dei transistor presenti (CPU desktop e server). Le GPU in parte permettono di ridurre anche questo problema grazie al loro alto grado di parallelismo che permette di sfruttare più transistor contemporaneamente.

**6.2 Gerarchia di Flynn**

I modelli illustrati dalla gerarchia di Flynn sono fondamentali per capire come il calcolo parallelo viene implementato in diversi contesti hardware. Ogni modello ha i suoi punti di forza e debolezza e viene utilizzato in base alle necessità dell'applicazione.

**6.2.1 SISD****Definizione 6.2.1: SISD (Single Instruction Single Data)**

Modello più semplice, solitamente utilizzato nei tradizionali processori single core. In un'architettura SISD una *singola unità di controllo* gestisce una *singola istruzione* alla volta e questa istruzione opera su un *singolo dato*.

**Note:-**

Un normale processore di un personal computer che esegue un codice seriale. Se un programma richiede l'aggiunta di due numeri, il processore SISD eseguirà l'operazione in un singolo flusso, elaborando un'istruzione alla volta.

### 6.2.2 SIMD

#### Definizione 6.2.2: SIMD (Single Instruction Multiple Data)

In un'architettura SIMD, una singola istruzione viene eseguita contemporaneamente su *molti dati*. Questo è utile per operazioni che devono essere applicate in parallelo su grandi set di dati, come il calcolo vettoriale o l'elaborazione grafica.

#### Note:-

Le GPU sono un classico esempio di SIMD. Quando una GPU rendirizza un'immagine lo stesso shader (istruzione) può essere applicato a molti dati (pixel).

### 6.2.3 MIMD

#### Definizione 6.2.3: MIMD (Multiple Instruction Multiple Data)

In un'architettura MIMD *più istruzioni* vengono eseguite contemporaneamente su *molti dati diversi*. Questo modello è più complesso e versatile perché permette a diversi processori di eseguire diverse istruzioni su diversi dati indipendentemente.

#### Note:-

I moderni processori multicore in un computer sono esempi di MIMD. Ogni core può eseguire un thread separato, con istruzioni diverse che lavorano su dati diversi. Questo modello è utile per eseguire diversi programmi o processi simultaneamente.

### 6.2.4 SIMT

#### Definizione 6.2.4: SIMT (Single Instruction Multiple Thread)

È un modello ibrido che combina aspetti di SIMD e MIMD. Utilizzato prevalentemente nelle GPU moderne, consente a *molti thread* di eseguire istruzioni su dati diversi. Tuttavia, a differenza del SIMD puro, i thread in un blocco SIMT possono divergere<sup>a</sup>.

<sup>a</sup>Prendere percorsi diversi nel flusso di controllo del programma.

#### Note:-

Le GPU NVIDIA utilizzano il modello SIMT. Ogni *WARP*<sup>a</sup> in una GPU NVIDIA esegue la stessa istruzione, ma su dati diversi. Se alcuni thread di un WARP devono eseguire un'istruzione condizionale (e. g. IF), possono divergere e successivamente riconvergere, ma sono ancora coordinati dal modello SIMT.

<sup>a</sup>Gruppo di thread.

## 6.3 Elaboratori Moderni

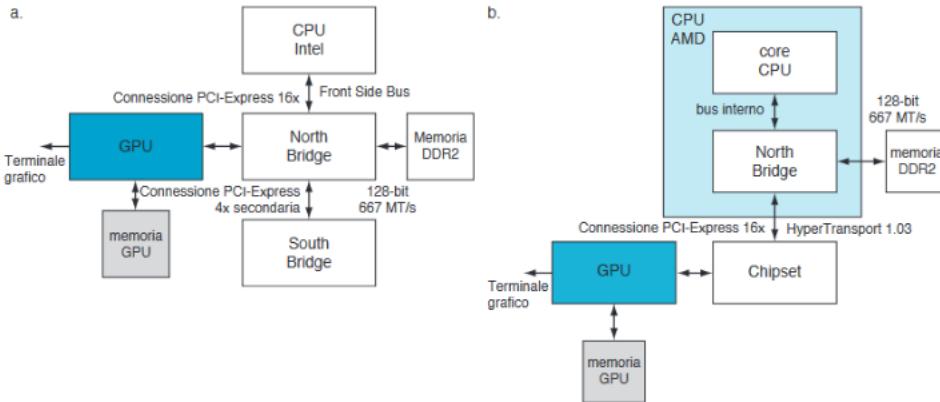
### 6.3.1 Schema Generale

L'architettura di un sistema di calcolo eterogeneo composto da CPU e GPU può essere descritta ad alto livello prendendo in considerazione due caratteristiche principali:

- Il numero di sottosistemi funzionali.
- Il tipo di tecnologia di interconnessione.

#### Note:-

Il *north bridge* contiene le interfacce a banda larga che connettono la CPU, la memoria e il PCI.



**Figura C.2.2.** PC contemporanei con CPU Intel e AMD. Si veda il Capitolo 6 per una descrizione dei componenti e delle interconnessioni riportate in figura.

Figure 6.1: Schema di un elaboratore moderno.

Nella figura a è riportato lo schema INTEL dove la GPU è connessa mediante un collegamento **PCI-EXPRESS** 2.0 a 16 vie che fornisce una velocità di trasferimento di picco di 16 GB/s (8 in ogni direzione). Nella figura b è riportato lo schema AMD nel quale la GPU è connessa al chipset, anche in questo caso tramite PCI-EXPRESS. In entrambi i casi GPU e CPU possono accedere alla memoria l'una dell'altra, seppure con una larghezza di banda minore che nel caso di accesso diretto.

Le GPU possono accedere alla propria memoria locale e alla memoria fisica di sistema della CPU utilizzando indirizzi virtuali che vengono tradotti da un'unità di calcolo dedicata di gestione della memoria (MMU, Memory Management Unit) a bordo della GPU. Sarà il kernel del sistema operativo a gestire la tabella delle pagine della GPU: si può accedere a una pagina fisica della memoria di sistema mediante transazioni sulla connessione PCI-EXPRESS coerenti o non-coerenti, a seconda del valore di un attributo specificato nella tabella delle pagine della GPU. La CPU può accedere alla memoria locale della GPU attraverso un intervallo di indirizzi nello spazio di indirizzamento del bus PCI-EXPRESS.

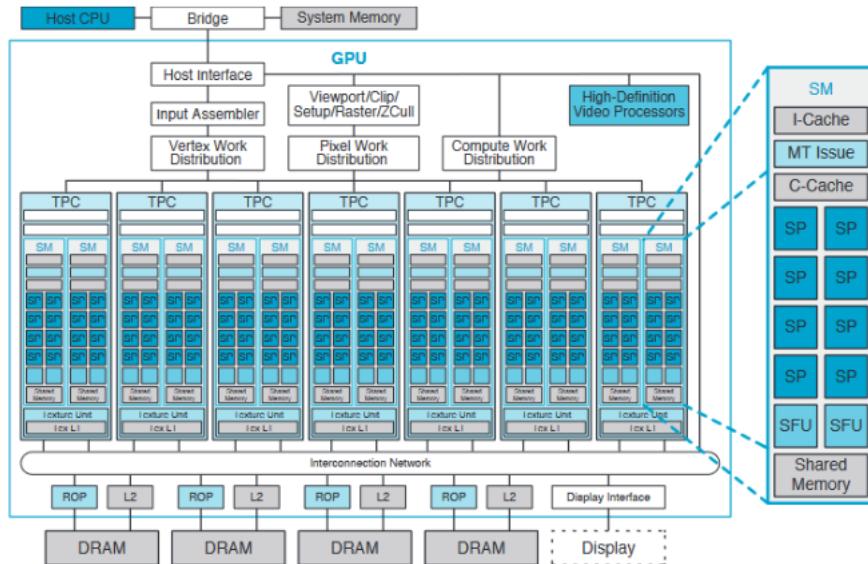
### 6.3.2 Architettura di una GPU

Le architetture di GPU unificate sono basate su una schiera di molti processori programmabili che unificano l'elaborazione effettuata dagli shader dei vertici, della geometria e dei pixel e il calcolo generico sugli stessi processi, a differenza delle GPU precedenti che avevano processori separati dedicati a ciascun tipo di elaborazione. L'insieme dei processori programmabili è strettamente integrato con alcuni processori che hanno una funzione prefissata, come il filtraggio della tessitura, la rasterizzazione, le operazioni sulla memoria video, l'anti-aliasing, la compressione, la decomposizione, la visualizzazione, la decodifica video e l'elaborazione video ad alta definizione.

Rispetto alle CPU multicore, le GPU a più core presentano una differente filosofia di progetto dell'architettura, che è focalizzata sull'esecuzione di molti thread paralleli su più processori in modo efficiente. Utilizzando molti core più semplici e ottimizzando il parallelismo sui dati tra gruppi di thread, una frazione maggiore di transistor per ogni chip è dedicata all'elaborazione e non alla memoria cache interna o funzioni accessorie.

Una schiera di processori unificati di una GPU contiene molti processori core, tipicamente organizzati in multiprocessori multithread. In figura 6.2 è mostrata una GPU contenente un'insieme di 112 processori core a flusso continuo (SP, Streaming Processor) organizzati in 14 multiprocessori multithread a flusso continuo (SM, Streaming Multiprocessor). Ogni core SP è altamente multithread, essendo in grado di gestire in hardware 96 thread concorrenti e il loro stato. I processori sono connessi a 4 partizioni di memoria DRAM a 64 bit attraverso una rete di interconnessione. Ogni SM contiene 8 core SP, 2 unità per le istruzioni multithread e una memoria condivisa. Questa è l'architettura di base implementata nella scheda **GeForce 8800 di NVIDIA**.

L'architettura a schiera di processori consente di creare configurazioni di GPU, modificando il numero dei



**Figura C.2.5. Architettura di base di una GPU unificata.** Esempio di GPU contenente 112 processori core a flusso continuo (SP) organizzati in 14 multiprocessori a flusso continuo (SM); i core sono fortemente multithread. Lo schema riproduce l'architettura Tesla di base di una GeForce 8800 NVIDIA. I processori sono connessi a quattro partizioni di memoria DRAM a 64 bit attraverso una rete di interconnessione. Ogni SM contiene otto core SP, due unità per funzioni speciali (SFU), memorie cache per le istruzioni e per le costanti, un'unità per istruzioni multithread e una memoria condivisa.

Figure 6.2: Architettura di una GPU.

multiprocessori e il numero di partizioni di memoria. Il numero dei processori e il numero delle memorie può variare, al fine di progettare sistemi basati su GPU che siano bilanciati per differenti prestazioni e destinati a differenti segmenti di mercato.

Il sottosistema di memoria è il componente principale per le prestazioni del sistema grafico. I sistemi di memoria delle GPU devono possedere le seguenti caratteristiche:

- **Aampiezza Elevata:** è presente un elevato numero di piedini per trasferire i dati tra la GPU e i suoi dispositivi di memoria e che la struttura a matrice della memoria stessa è composta da molti chip di DRAM in modo da poter sfruttare tutta la larghezza del bus dati.
- **Elevata velocità:** sono implementate tecniche aggressive di segnalazione per massimizzare la velocità di trasferimento dei dati (bit/s) per ogni piedino del chip.
- Le GPU cercano di utilizzare ogni ciclo di clock disponibile per trasferire dati da o verso la matrice della memoria. Per raggiungere questo obiettivo nelle GPU non si cerca di minimizzare la latenza del sistema di memoria: un throughput elevato (efficienza di utilizzo) e una latenza bassa sono intrinsecamente in conflitto.
- Vengono impiegate tecniche di compressione, sia con perdita di informazione (eventualità di cui il programmatore deve essere ben consapevole) sia senza perdita, le quali vengono applicate in maniera opportunistica e risultano trasparenti alle altre applicazioni.
- Cache e strutture che fondono le richieste di trasferimento vengono utilizzate per ridurre la quantità di traffico richiesto fuori dal chip e per garantire che i cicli di clock spesi per il trasferimento dei dati vengano sfruttati nella maniera più completa possibile.

### Domanda 6.1

Ma come viene gestita la memoria?

- **Memoria Globale:** risiede nella DRAM esterna e non è locale a nessuno degli SM, poiché è pensata per la comunicazione tra i diversi CTA (blocchi di thread) di griglie diverse.

- **Memoria Condivisa:** è dedicata a ciascun CTA, è visibile soltanto ai thread che appartengono a quel CTA e alloca spazio di memoria solo dall'istante in cui il CTA viene creato fino all'istante in cui termina. Per questi motivi spesso risiede sul chip del SM per limitare al minimo i tempi di interazione.

- **Memoria Locale:** questa è dedicata a ciascun thread, è una memoria privata visibile soltanto al singolo thread. Dal punto di vista architetturale è più grande dell'insieme dei registri di thread. Per permettere di allocare vaste aree di memoria si usa la DRAM esterna.

**Note:-**

La memoria globale e locale, essendo che risiedono esternamente al chip, si prestano ad avere la propria cache nel chip.

## 6.4 Programmare una GPU: CUDA

La programmazione di una GPU differisce in maniera sostanziale dalla programmazione degli altri multiprocessori. I modelli di programmazione delle GPU e i programmi applicativi vengono progettati per coprire una vasta gamma di gradi di parallelismo. Il motivo per cui è necessario disporre di molti core ed eseguire svariati thread in parallelo è rappresentato dalla grafica in tempo reale (solo un sistema di questo tipo permette il rendering di scene 3D complesse ad alta risoluzione). Analogamente il modello di programmazione CUDA permette a generiche applicazioni di calcolo parallelo di sfruttare grandi quantità di thread paralleli e funzionare su un numero qualsiasi di processore core in modo trasparente all'applicazione. Il programmatore scrive il codice per un singolo thread, ma la GPU lancia un numero elevato di istanze del thread in parallelo.

### 6.4.1 Decomposizione di un Problema

Per mappare in modo efficace problemi di calcolo di grandi dimensioni su un'architettura altamente parallela, il programmatore o il compilatore devono scomporre il problema in tanti problemi più piccoli che possono essere risolti in parallelo. Per esempio: il programmatore può partizionare un grande vettore di dati contenente il risultato in blocchi e poi partizionare ulteriormente ogni blocco in elementi più piccoli, così che i blocchi del risultato possano essere determinati indipendentemente e in parallelo e gli elementi di ogni blocco possano essere elaborati in parallelo.

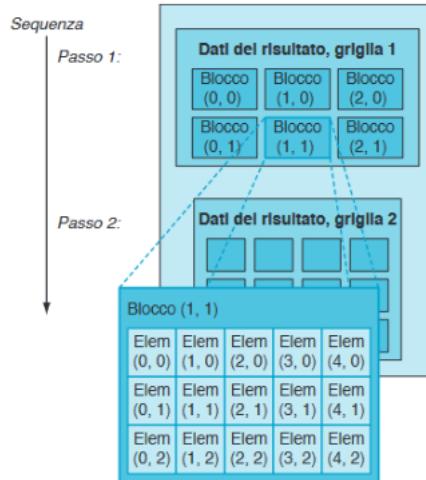


Figure 6.3: Scomposizione dei dati in una griglia di blocchi di elementi che vengono calcolati in parallelo.

## 6.4.2 Il Paradigma CUDA

### Definizione 6.4.1: CUDA: Programmazione Scalabile e Parallela

CUDA estende i linguaggi C e C++ per sfruttare il parallelismo. Questo fornisce 3 astrazioni fondamentali che permettono una strutturazione chiara del parallelismo:

- *Gerarchia di gruppi.*
- *Memorie condivise.*
- *Sincronizzazione a barriera.*

#### Note:-

Le astrazioni guidano il programmatore a suddividere il problema in spezzoni indipendenti e poi ulteriormente in spezzoni più piccoli.

Il programmatore scrive un programma sequenziale che richiama kernel paralleli, i quali possono contenere semplici funzioni o interi programmi. Un kernel viene eseguito in parallelo su un insieme di thread paralleli. Il programmatore organizza questi thread in una gerarchia di blocchi di thread e griglie di blocchi di thread. Un *blocco di thread* è un insieme di thread concorrenti che possono collaborare tra loro mediante sincronizzazione a barriera e accesso condiviso allo spazio di memoria privato del blocco. Una *griglia* è un insieme di blocchi di thread concorrenti che possono essere eseguiti ciascuno in modo indipendente, quindi in parallelo. Quando deve essere lanciato in esecuzione un kernel, il programmatore specifica il numero di thread per blocco e il numero di blocchi costituenti la griglia. A ciascun thread viene assegnato un numero identificativo univoco all'interno del blocco di thread che lo contiene, e a ciascun blocco di thread un numero di blocco univoco all'interno della sua griglia.

Il testo di un *kernel CUDA* è semplicemente una funzione C scritta per un thread sequenziale. È quindi in genere molto semplice da scrivere, soprattutto rispetto al codice parallelo per le operazioni vettoriali. Il parallelismo è determinato in modo chiaro ed esplicito specificando le dimensioni della griglia di elaborazione e il numero dei suoi blocchi di thread quando il programma kernel viene lanciato in esecuzione. L'esecuzione parallela e la gestione dei thread sono automatiche.

I thread di un blocco vengono eseguiti in modo concorrente e possono venire sincronizzati in corrispondenza di una *barriera di sincronizzazione* chiamando la primitiva *syncthreads()*. Questo garantisce che nessun thread del blocco possa proseguire l'esecuzione fino a quando tutti i thread dello stesso blocco non avranno raggiunto la barriera. Dopo averla superata, è garantito che questi thread vedranno i dati che i thread hanno scritto in memoria prima di aver raggiunto la barriera. I thread all'interno di uno stesso blocco, quindi, possono comunicare tra loro scrivendo e leggendo nella memoria condivisa del blocco in corrispondenza della barriera di sincronizzazione. Poiché i thread di un blocco possono condividere la memoria e sincronizzarsi mediante le barriere, essi risiederanno fisicamente nello stesso processore o multiprocessore. Tuttavia, il numero di blocchi di thread può essere significativamente maggiore del numero di processori e fornisce al programmatore totale flessibilità sul grado di granularità che ritiene più conveniente. La virtualizzazione in thread e blocchi di thread permette di scomporre il problema in maniera intuitiva, poiché il numero dei blocchi può essere determinato più dalla dimensione dei dati da elaborare che dal numero di processori del sistema.

Per gestire questa virtualizzazione degli elementi di elaborazione e garantire la scalabilità su più processori, CUDA richiede che i blocchi di thread possano essere eseguiti in modo indipendente: si deve poter eseguire i blocchi in un ordine qualsiasi, in parallelo o in sequenza. Blocchi differenti non hanno possibilità di comunicazione diretta, benché essi possano coordinare le loro attività utilizzando operazioni atomiche di memoria sulla memoria globale visibile da tutti i thread. Questo requisito di indipendenza consente di mandare in esecuzione blocchi di thread in un qualsiasi ordine su un qualsiasi numero di processori rendendo il modello CUDA scalabile su un numero arbitrario di core.

Ogni thread dispone di una *memoria locale* privata. Ogni blocco dispone di una *memoria condivisa* visibile a tutti i thread del blocco e tutti i thread hanno anche accesso alla *memoria globale*.

Cuda ha uno stile molto simile al modello SIMD: il parallelismo è espresso in modo esplicito e ogni kernel viene eseguito con un numero prefissato di thread. Tuttavia, CUDA è più flessibile in quanto il numero di thread

e per blocco e griglia varia per ogni kernel.

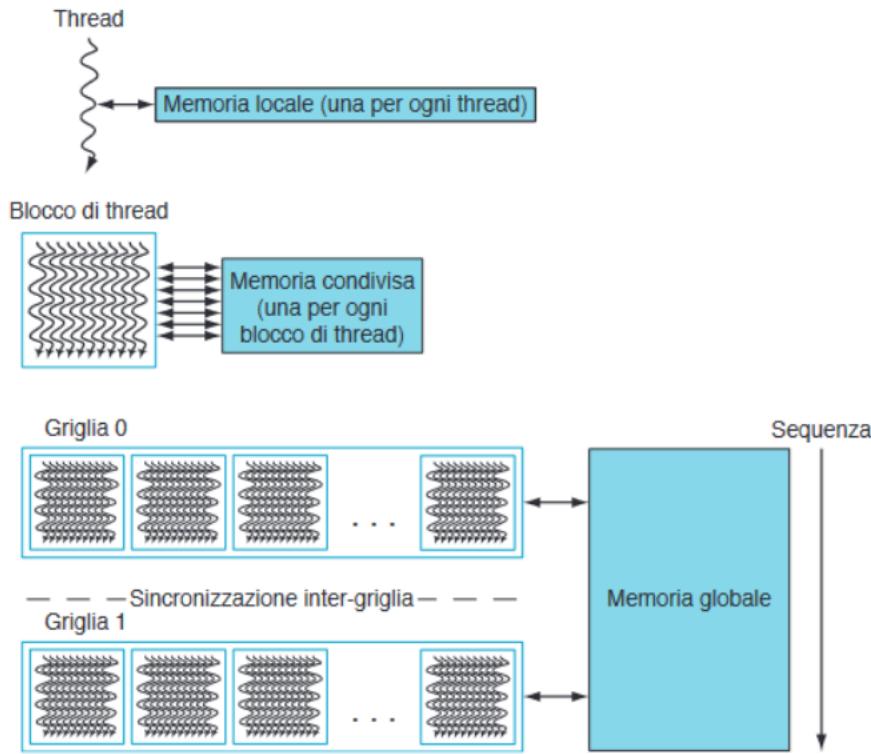


Figure 6.4: I livelli di granularità innestati con le memorie corrispondenti.

## 6.5 I Thread

### 6.5.1 Architettura per il Multithreading

Le GPU sono multiprocessori composti da multiprocessori, dove ognuno di questi è fortemente multithread. L'alto livello di multithreading è richiesto per vari motivi:

- Nascondere la latenza del caricamento dei dati dalla memoria e della tessitura della RAM.
- Supportare i modelli di programmazione degli shader grafici paralleli a grana fine.
- Supportare i modelli di programmazione per il calcolo parallelo a grana fine.
- Rendere virtuali i processi fisici come thread e blocchi di thread per fornire una scalabilità trasparente.
- Semplificare il modello di programmazione parallela riducendolo alla scrittura di un programma sequenziale per un singolo thread.

La latenza del caricamento dei dati e della tessitura della memoria può richiedere centinaia di cicli di clock del processore, poiché le GPU sono tipicamente dotate di cache a flusso continuo di piccole dimensioni. L'organizzazione multithreading aiuta a riempire i tempi di latenza con altre elaborazioni: mentre un thread attende il completamento del caricamento dei dati dalla memoria o del prelievo di una tessitura, il processore può eseguire un altro thread. I modelli di programmazione parallela a grana fine generano migliaia di thread indipendenti, i quali possono mantenere impegnati i processori nonostante le lunghe latenze di memoria viste dal singolo thread. I programmi di grafica e di calcolo istanziano molti thread paralleli rispettivamente per generare immagini complesse e per calcolare matrici di grandi dimensioni che contengono il risultato.

Per supportare il modello di programmazione ogni thread della GPU dispone dei propri *registri privati*, di *memoria privata* dedicata al thread, di un registro *program counter* e di un *registro dello stato di esecuzione del thread*. Proprio per questo è in grado di eseguire un frammento di codice in maniera indipendente.

Per eseguire in modo efficiente centinaia di thread leggeri e concorrenti, il multiprocessore della GPU implementa il multithreading in hardware, ossia gestisce ed esegue centinaia di thread concorrenti in hardware, senza sovraccarichi per la pianificazione dell'esecuzione.

### 6.5.2 L'Architettura del Multiprocessore

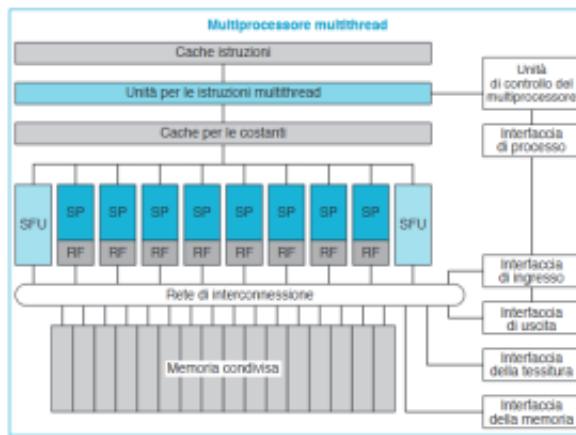


Figura C.4.1. Multiprocessore multithread contenente otto processori core scalari (SP). Gli otto core SP dispongono ciascuno di un grosso insieme di registri multithread (RF) e condividono una cache istruzioni, un'unità di lancio di istruzioni multithread, una cache per le costanti, due unità per funzioni speciali (SFU), una rete di interconnessione e una memoria condivisa a bandi multipli.

Figure 6.5: SM.

L'SM riportato contiene otto SP, ognuno dei quali equipaggiato con tanti registri multithread.

Ogni core SP contiene unità aritmetiche scalari intere e in virgola mobile che eseguono la maggior parte delle istruzioni. Il processore SP implementa il multithreading in hardware ed è in grado di gestire fino a 64 thread. La pipeline di ogni SP esegue un'istruzione scalare per ogni thread per ogni ciclo di clock, la cui frequenza può variare tra 1.2 GHz e 1.6 GHz, a seconda del modello. Ogni processore SP possiede un insieme di registri (RF) di grosse dimensioni, costituito da 1024 registri a 32 bit di utilizzo generale. Il processore SP può eseguire in modo concorrente molti thread che impiegano pochi registri, oppure un numero minore di thread che necessitano di più registri.

I programmi CUDA compilati necessitano spesso di 32 registri per thread, limitando così ogni SP a 32 thread e imponendo che un programma kernel abbia al massimo 256 thread per ogni blocco.



Figure 6.6: SM di NVIDIA Ampere Architecture.

### 6.5.3 SIMT, WARP e WARP Scheduler

Per gestire ed eseguire in maniera efficiente centinaia di thread che svolgono una grande quantità di programmi diversi, il multiprocessore adotta un'architettura a singola istruzione e thread multipli (**SIMT**). Essa crea, gestisce e pianifica l'esecuzione ed esegue thread concorrenti in gruppi di thread paralleli denominati **WARP**. Il multiprocessore d'esempio utilizza una dimensione di WARP di 32 thread ed esegue 4 thread in ciascuno degli 8 core SP, in 4 cicli di clock.

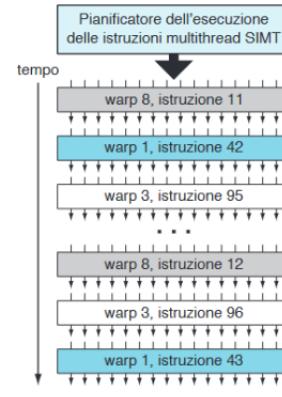
I singoli thread paralleli che compongono un WARP sono dello stesso tipo e partono contemporaneamente dallo stesso indirizzo di codice, ma poi sono liberi di procedere e seguire le biforcati del codice in modo indipendente. Ogni volta che viene lanciata in esecuzione un'istruzione, l'unità istruzioni multithread SIMT seleziona un WARP che è pronto per eseguire l'istruzione successiva e invia quell'istruzione ai thread attivi di quel WARP. Un'istruzione SIMT viene diffusa in modo sincrono sui thread paralleli attivi del WARP.

L'architettura di un processore SIMT è simile a quella dei processori a singola istruzione a corsi di elaborazione multiple di dati. La differenza sta nel fatto che l'architettura SIMT applica un'istruzione a thread multipli indipendenti in parallelo e non semplicemente a corsie multiple di dati.

Un processore SIMT ottiene piena efficienza e massime prestazioni quando tutti i thread di un WARP seguono lo stesso flusso di esecuzione. Se alcuni thread divergono a causa di un salto condizionato dipendente dai dati, l'esecuzione diventa sequenziale per ognuna delle due diramazioni possibili del codice e, quando tutti i flussi di esecuzione giungono a conclusione, i thread si ricongiungono a un unico flusso. Un codice che contiene una biforcazione del tipo IF ELSE in due flussi di esecuzione di uguale lunghezza ha un'efficienza del 50%. Per gestire thread indipendenti che divergono e convergono, il multiprocessore utilizza uno stack di sincronizzazione delle diramazioni. WARP differenti vengono eseguiti in maniera indipendente fra loro alla massima velocità possibile, anche se stanno eseguendo flussi di codice comuni o disgiunti. Ne consegue che le GPU SIMT sono decisamente più efficienti e flessibili in relazione ai frammenti di codice divergenti rispetto alle GPU delle precedenti generazioni, poiché i loro WARP sono molto più sottili se paragonati all'ampiezza delle istruzioni SIMD.

Perché il codice sia corretto, il programmatore può sostanzialmente ignorare gli attributi dell'esecuzione SIMT dei WARP ; tuttavia, si può ottenere un incremento sostanziale delle prestazioni avendo cura che il codice raramente abbia bisogno di divergere all'interno di un WARP.

Un blocco di thread contiene uno o più WARP. L'architettura SIMT condivide l'unità di prelievo e distribuzione delle istruzioni in modo efficiente tra i thread paralleli di uno stesso WARP, ma richiede che tutti i thread del WARP siano attivi per ottenere la massima prestazione. Tale multiprocessore unificato pianifica l'esecuzione ed esegue molteplici tipi di WARP in modo concorrente, permettendo quindi di eseguire WARP di vertice e di pixel. Lo scheduler dei WARP lavora a una frequenza minore di clock del processore, perché deve riempire le corsie di 4 thread per ogni processore core; durante ogni ciclo di pianificazione, esso seleziona un WARP per l'esecuzione di un'istruzione SIMT. Le istruzioni di un WARP vengono lanciate in esecuzione come 4 gruppi di 8 thread, generando i risultati su 4 cicli di clock del processore. Il pianificatore (scheduler) deve selezionare un WARP ogni 4 cicli di clock in modo da lanciare in esecuzione un'istruzione per ciascun ciclo di clock per ogni thread, il che equivale a un IPC = 1,0 per ogni processore core. Dato che i WARP sono indipendenti, le uniche dipendenze possono nascere tra le istruzioni sequenziali di uno stesso WARP. Lo scheduler utilizza una tabella sulla quale annotare le dipendenze tra i registri, per identificare i WARP nei quali i thread attivi sono pronti a eseguire un'istruzione. Questo rende prioritari tutti i WARP pronti e seleziona quindi per il lancio in esecuzione il WARP con la priorità più alta. Il calcolo della priorità deve tener conto del tipo di WARP, del tipo di istruzione, nonché del desiderio di equità nei confronti di tutti i WARP attivi.



**Figura C.4.2. Pianificazione dell'esecuzione di warp multithread SIMT.**  
Il pianificatore (scheduler) seleziona un warp pronto per l'esecuzione e lancia in esecuzione un'istruzione in modo sincrono sui thread paralleli che costituiscono il warp. Poiché i warp sono indipendenti, il pianificatore può scegliere ogni volta un warp differente.

Figure 6.7: Scheduler WARP.

### 6.5.4 Blocchi, Griglie e Sincronizzazioni

L'unità di controllo del multiprocessore e quella delle istruzioni gestiscono i thread e i blocchi di thread. L'unità di controllo accetta richieste di elaborazione e dati in ingresso e arbitra l'accesso alle risorse condivise, comprese

l'unità della tessitura, il cammino di accesso alla memoria e i collegamenti di I/O. L'unità di controllo alloca anche un WARP libero e i registri per i thread del WARP e inizia l'esecuzione di WARP sul multiprocessore. Ogni programma dichiara il numero di registri per thread di cui ha bisogno; l'unità di controllo fa partire un WARP solo quando è in grado di allocare il numero di registri richiesto. Quando tutti i thread di WARP hanno terminato l'esecuzione, l'unità di controllo distribuisce i risultati e libera i registri e le risorse dal WARP. L'unità di controllo crea *insiemi di thread cooperativi* (CTA, Cooperative Thread Array), i quali implementano blocchi di thread CUDA sotto forma di uno o più WARP di thread paralleli. Essa crea un CTA quando può produrre tutti i WARP del CTA e allocare tutte le risorse condivise utilizzate dal CTA. L'unità di controllo si accorge quando tutti i thread di un CTA hanno terminato e libera quindi le risorse condivise utilizzate dal CTA e dai suoi WARP.

La *sincronizzazione veloce a barriera* permette ai programmi CUDA di comunicare frequentemente attraverso la memoria condivisa e la memoria globale, semplicemente chiamando la funzione *syncthread()* come parte di ogni passo di comunicazione tra thread. Raggruppando i thread in un WARP SIMT di 32 thread si riduce la complessità della sincronizzazione di un fattore 32. I thread attendono a una barriera all'interno dello scheduler dei thread SIMT, in modo da non utilizzare cicli di processore durante l'attesa. Quando un thread esegue un'istruzione *bar.sync*, incrementa il contatore del numero di thread arrivati alla barriera e lo scheduler registra il thread in attesa. Quando tutti i thread del CTA sono arrivati alla barriera, il conteggio corrisponderà al numero di thread e lo scheduler libererà tutti i thread in attesa, facendo riprendere la loro esecuzione.

**Note:-**

I core LOAD e STORE sono dedicati alla lettura/scrittura sulla memoria globale, sono core dedicati al migliorare l'efficienza delle operazioni riguardanti la memoria.

## 6.6 Quale codice può girare su una GPU?

Le GPU sono componenti pensati per l'elaborazione visuale. In generale però, al giorno d'oggi, grazie al paradigma CUDA possono essere utilizzate più facilmente per l'elaborazione di qualsiasi problema che necessiti di un'elevata parallelizzazione durante l'esecuzione. In generale quindi svolgono molto facilmente elaborazioni che, se eseguite sequenzialmente, risulterebbero eccessivamente complesse, mentre non sono ottimali per programmi che di base non permettono la scomposizione del problema in sottoproblemi da parallelizzare. I casi d'uso più emblematici sono:

- *Map* (*AppyAll*): applica una funzione specificata a ogni elemento di una lista (o di un altro iterabile) e restituisce una nuova lista con i risultati. In pratica trasforma i dati. In altre parole è semplicemente la capacità di applicare una funzione a  $n$  item di una lista, dove l'ordine degli item è indifferente.
- *Reduce*: aggrega o combina gli elementi di una lista in un unico risultato utilizzando una funzione binaria specificata (somma, prodotto, etc.). Riduce i dati a un singolo valore. Può essere associativa e commutativa.

**Note:-**

Queste due operazioni permettono alle GPU di essere più efficienti nei problemi scomponibili.

### 6.6.1 Map: VectorAdd

Nell'ipotesi in cui volessimo creare una semplice funzione che somma due vettori ( $C = A + B$ ), ci risulta subito chiara la potenza del paradigma MAP.

```

1 /**
2 * Computes the vector addition of A and B into C. The 3 vectors have the same
3 * number of elements numElements.
4 */
5 void vectorAdd(const float *A, const float *B, float *C, int numElements){
6     for (int i = 0; i < numElements; ++i) {
7         C[i] = A[i] + B[i] + 0.0f;
8     }
9 }
```

Listing 6.1: Implementazione sequenziale

```

1 /**
2 * Computes the vector addition of A and B into C. The 3 vectors have the same
3 * number of elements numElements.
4 */
5 __global__ void vectorAdd(const float *A, const float *B, float *C, int numElements) {
6     int i = blockDim.x * blockIdx.x + threadIdx.x;
7     if (i < numElements) {
8         C[i] = A[i] + B[i] + 0.0f;
9     }
10 }
```

Listing 6.2: Implementazione CUDA

È evidente che il passaggio da CPU a GPU permetta l'eliminazione del ciclo per effettuare la somma dei vettori. La somma di ogni elemento dei vettori è lasciata a un singolo thread. Solo quando tutti i thread hanno terminato, quindi si ha raggiunto la barriera di sincronizzazione, si ha raggiunto il risultato.

### 6.6.2 Reduce

Dato un vettore vogliamo ottenere la somma di tutti i suoi elementi.

```

1 /*
2 * Parallel sum reduction using shared memory
3 */
4 void reduce(const int *input, int *output, unsigned int n) {
5     // Inizializzazione dell'output a zero
6     output[0] = 0;
7
8     // Riduzione seriale
9     for (unsigned int i = 0; i < n; ++i) {
10         output[0] += input[i];
11     }
12 }
```

Listing 6.3: Implementazione sequenziale

```

1 /*
2 * Parallel sum reduction using shared memory
3 */
4 __global__ void reduce(T *g_idata, T *g_odata, unsigned int n) {
5     // Handle to thread block group
6     cg::thread_block cta = cg::this_thread_block();
7     T *sdata = SharedMemory<T>();
8
9     // load shared mem
10    unsigned int tid = threadIdx.x;
11    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
12
13    // uso la shared memory come una cache programmata della global memory
14    sdata[tid] = (i < n) ? g_idata[i] : 0;
15
16    //barriera di sincronizzazione, aspetto che tutti i thread arrivino a questo punto.
17    cg::sync(cta);
18
19    // do reduction in shared mem
20    for (unsigned int s = 1; s < blockDim.x; s *= 2) {
21        // modulo arithmetic is slow!
22        if ((tid % (2 * s)) == 0) {
```

```

23     sdata[tid] += sdata[tid + s];
24 }
25 cg::sync(cta);
26 }
27
28 // write result for this block to global mem
29 if (tid == 0) g_odata[blockIdx.x] = sdata[0];
30 }
```

Listing 6.4: Implementazione CUDA

A questo punto sono chiare due cose:

1. Il codice scritto per una GPU risulta più corposo e di difficile interpretazione, inoltre è il programmatore a doverlo rendere efficiente e controllare lo stato di esecuzione (shared memory e barriera di sincronizzazione).
2. Si ha un aumento notevole di prestazioni da  $O(n)$  a  $O(\log n)$  per lo stesso problema.

### 6.6.3 Copiare da Global Memory a Shared Memory

Per copiare da global a shared memory si assegna una variabile che è in shared memory a una che è in global memory. Per esempio nella reduce:

```
1 sdata[tid] = (i < n) ? g_idata[i] : 0;
```

**Note:-**

Copiare le variabili in shared memory è utile in quanto la shared memory è molto più veloce (1 ciclo per accedervi rispetto ai +400 cicli per accedere alla global memory). Importarsi in questo modo le variabili è come se si andasse a creare una *cache programmabile*, in cui decide il programmatore cosa avvicinare.

