

Appunti di Calcolatori Elettronici

Unali Andrea, Baki di Pirri

2025

Indice

1	Introduzione	5
1.1	Diverse Architetture	5
1.2	Evoluzione delle Architetture	7
1.3	Legge di Moore e Nathan	8
1.4	Analisi delle Prestazioni	9
1.5	Note a Margine	i
2	Organizzazione dei Sistemi di calcolo	11
2.1	Struttura di un computer	11
2.2	Il BUS	12
2.3	Tecnologie per BUS esterni	13
2.4	La CPU	14
2.5	Principi di Progettazione e Parallelismo	17
2.6	La Memoria Centrale e di Massa	21
2.7	Dispositivi di Input/Output	26
2.8	Modalità di I/O	27
2.9	Note a Margine	i
3	Funzionamento del BUS	31
3.1	Approfondimento e Gestione del BUS	31
3.2	Protocollo e Arbitraggio del BUS	33
3.3	Temporizzazione del BUS	37
4	Algebra e logica Booleana	43
4.1	Sistemi Numerici	43
4.2	Alfabeto Binario	44
4.3	Rappresentazione di Numeri con Segno	44
4.4	Rappresentazione in Virgola Mobile (IEEE 754)	45
4.5	Algebra Booleana	46
4.6	Porte Logiche e Reti Combinatorie	47
5	Livello Logico Digitale	49
5.1	Architettura a Livelli dei Calcolatori	49
5.2	Reti Logiche e Reti Combinatorie	50
5.3	Componenti Elettronici	50
5.4	Reti Logiche Sequenziali	53
5.5	Elementi di Memoria	56
5.6	Operazioni Aritmetiche Binarie	61
5.7	Operazioni Fondamentali in Floating Point	63
5.8	ALU (Arithmetic Logic Unit)	64

6 Memorie e Cache	71
6.1 Introduzione alle Memorie	71
6.2 Memorie a Semiconduttore	73
6.3 Temporizzazione delle Memorie	77
6.4 Principio di Località e Gerarchie	78
6.5 Memoria cache	80
7 Microarchitetture Avanzate	105
7.1 Introduzione alle microarchitetture	105
7.2 Pipelining: principi e struttura	106
7.3 Pipeline MIPS	109
7.4 Diagrammi Temporali	113
7.5 Hazard della Pipeline	115

Capitolo 1

Introduzione

In questo capitolo vedremo la struttura generale delle architetture, e le varie tipologie. Citando alcune delle leggi fondamentali che hanno caratterizzato l'evoluzione dei calcolatori negli ultimi decenni.

1.1 Diverse Architetture

Tanto per iniziare vediamo sbrigativamente una serie di tipologie di calcolatori che possiamo trovare ai giorni nostri:

- **RFID:** Appartengono alla categoria **usa e getta**, sono tipicamente senza batteria. Nel concreto stiamo parlando di quei minuscoli processori che troviamo sulle carte di credito o all'interno dei libri della biblioteca.

Di fatto vengono utilizzati per il *riconoscimento di qualcosa*.

Contengono al loro interno un **numero a 128 bit** e quando **ricevono un segnale radio trasmettono il proprio numero**.

- **Microcontrollori:** Piccoli computer inclusi in vari dispositivi tipicamente connessi alla rete; li troviamo all'interno delle automobili, elettrodomestici, ecc. Sono tipicamente dotati di:

- una **CPU**;
- una piccola **Memoria**;
- qualche **dispositivo di I/O**.

- **Game computers:** Sono normali computer ma con le seguenti caratteristiche:

- Software Limitato;
- Sbilanciati verso il calcolo grafico (GPU);
- Sistemi chiusi (non estendibili).

- **Smartphone:** Sono normali computer, con interfaccia (ormai quasi tutti) touchscreen, e possiedono tipicamente un sistema di riconoscimento biometrico. Usano un sistema operativo diverso, studiato *ad hoc* per l'uso mobile e le limitate risorse Hardware.

- **Tablet:** Quasi considerabili dei computer portatili, sono però caratterizzati da:

- Sistema operativo mobile;
- CPU potenti;

- Memorie ridotte;
- Interfaccia touchscreen.

• **Workstation:**

- Diversi TB di disco;
- Diversi GB di memoria;
- Rete locale o web server.

• **COW (Cluster of Workstation):**

- Sistema multiprocessore ad *accoppiamento lasco*¹;
- In sostanza tante workstation che comunicano tra loro.

• **Mainframe:**

- Centinaia di terminali connessi;
- Costi di parecchi milioni di euro.

1.1.1 Differenza tra CPU e GPU

- La **CPU**: è un processore di tipo generale, utilizzabile per gli usi più comuni e in grado di far girare qualunque tipo di software (tralasciando problemi di compatibilità).
- La **GPU**: è un processore specializzato, nasce per un uso specifico. In particolare è orientato al calcolo ai fini grafici, quindi immagini e video, mediante operazioni tra matrici. Non è specializzata (e quindi limitata) quanto un ASIC o un FPGA.

1.1.2 Architetture multilivello



Figura 1.1: Schema dell'architettura multilivello

Vedremo poi nel dettaglio la **Struttura di un computer** per analizzare le sue componenti interne e il loro funzionamento.

¹Vedi approfondimento nella sezione delle note a margine a fine capitolo: 1.5.1

Per progettare una macchina/CPU non posso mettere insieme i componenti necessari, raggruppare dei transistor e collegarli tra loro. È *necessario astrarre il problema in più livelli; nello specifico ogni livello può usufruire di strumenti e risorse messi a disposizione dai livelli sottostanti.*

Adesso quando si progetta un'architettura multilivello si tiene anche conto dell'importante uso di parallelismo, sia di task che di componenti. Per evitare sprechi (livelli di parallelismo inutilmente alti) si fa affidamento alla **Legge di Amdahl**².

1.1.3 Come si sono evolute le architetture?

8 Grandi idee che hanno permesso di inseguire le prestazioni

1. Legge di Moore;
2. Astrazione per semplificare il design;
3. Rendere l'utilizzo comune veloce;
4. Aumento del Parallelismo dei task e dell'hardware;
5. Aumento della Prediction (prevedere le prossime istruzioni, i tempi di esecuzione);
6. Aumento del Pipelining (catena di montaggio);
7. Gerarchia delle memorie (Cache → RAM → ROM);
8. Ridondanza.

Nota Bene

Queste idee e non solo hanno contrassegnato l'andamento dell'**Evoluzione delle architetture nel tempo**.

1.2 Evoluzione delle Architetture

Nelle slide vengono presentati una serie di esempi di microprocessori Intel per mostrare come sono progredite le tecnologie nel tempo.

Ciò che ci importa però è capire quali sono stati i fattori chiave che hanno contraddistinto una generazione rispetto a un'altra, o un processore di fascia media rispetto a uno di fascia bassa o alta:

1. Aumento dei Transistor;
2. A parità di Transistor, aumento dei core.

1.2.1 Come aumentare i core? (Il Binning)

Nel momento in cui creo una serie di microchip sopra il *wafer* di silicio ci saranno:

- chip completamente non funzionanti;
- chip parzialmente difettosi;
- chip perfettamente funzionanti.

²Vedi approfondimento nella sezione delle note a margine a fine capitolo: 1.5.2

I primi di essi andranno sprecati, mentre quelli **perfettamente funzionanti** saranno destinati a essere la fascia top di gamma della linea di mercato (ad esempio per Intel gli i9).

I chip **parzialmente difettosi**, invece, vengono classificati in base al numero di core difettosi. Se parto da 10 core e di questi 3 sono difettosi avrò un Intel i7; avrò un i5 invece con 5 core difettosi.

Ovviamente i core difettosi vengono disabilitati via hardware/software.

1.2.2 Intel vs ARM

- Negli anni '80 **Intel** ha investito su architetture di tipo **CISC**.
- Nello stesso periodo la **Acorn** ha investito maggiormente su architetture di tipo **RISC**. Questa compagnia negli anni si è suddivisa in più aziende e in particolare è arrivata ai giorni nostri la **Arm Holdings**.

Nota Bene

I processori **ARM** sono open source (è possibile visionare online la struttura e il set di istruzioni). Li troviamo all'interno di praticamente qualsiasi cellulare o dispositivo mobile, e sempre di più anche nei PC o laptop portatili. Questo grazie alla loro particolare **efficienza energetica**.

1.3 Legge di Moore e Nathan

1.3.1 Legge di Moore

Definizione 1.1: Legge di Moore

Il numero dei Transistor (componenti) su un circuito integrato raddoppierà ogni anno.

Dal 1970 fino agli anni 2000 la legge è stata quasi rispettata: il numero di transistor su singolo chip è **raddoppiato ogni 18 mesi**.

Cosa comporta l'aumento dei transistor:

- ↑ **Prestazioni** (vedi **Analisi delle Prestazioni**);
- ↓ Consumo di potenza;
- ↑ Affidabilità;
- ≈ Prezzo costante (dipendente dai prezzi di vendita; aumenta la scala produttiva).

1.3.2 Legge di Nathan

Definizione 1.2: Legge di Nathan

Il software è come un gas: riempie sempre completamente qualsiasi contenitore in cui lo si metta.

La capacità di calcolo che siamo in grado di offrire tramite l'hardware si rincorrerà sempre con la complessità del software, generando così un **circolo virtuoso**.

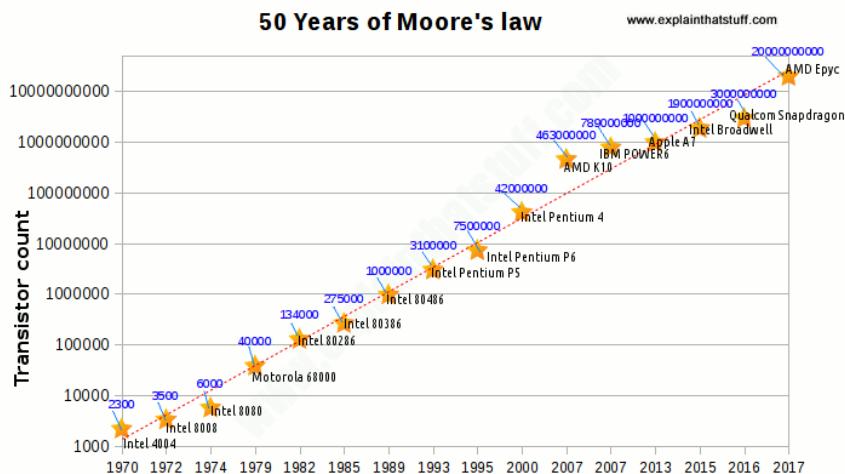


Figura 1.2: Crescita esponenziale dei transistor secondo la Legge di Moore

1.4 Analisi delle Prestazioni

Abbiamo detto che negli ultimi anni le prestazioni sono cresciute del 50% all'anno, fino al 2010, e del 22% dal 2010 in poi.

Ma come si fa a determinare lo stato attuale e l'avanzamento delle prestazioni?

- **CPI:** Indica il numero di operazioni per ciclo di clock.
- **GigaFlops:** Indica quante istruzioni per unità di tempo.
- **I/O Operations:** Quanto velocemente eseguo le operazioni di Input Output.
- **Machine Instructions:** Quante istruzioni macchina sono necessarie per eseguire un'operazione.

1.5 Note a Margine

1.5.1 Accoppiamento Lasco

Significa che i processori del sistema multiprocessore sono **poco integrati** tra loro:

- Ogni processore ha **memoria e risorse proprie**
- La comunicazione avviene tramite **reti o canali di interconnessione relativamente lenti** (es. LAN), non attraverso una memoria condivisa veloce.
- Ogni nodo può anche eseguire il proprio sistema operativo.

1.5.2 Legge di Amdhal

La Legge di Amdahl permette di determinare i potenziali guadagni in termini di prestazioni ottenuti dall'aggiunta di ulteriori core, nel caso di applicazioni che contengano sia componenti seriali (quindi non parallelizzabili) sia componenti parallele.

$$\text{speedup} \leq \frac{1}{S + \frac{1-S}{N}}$$

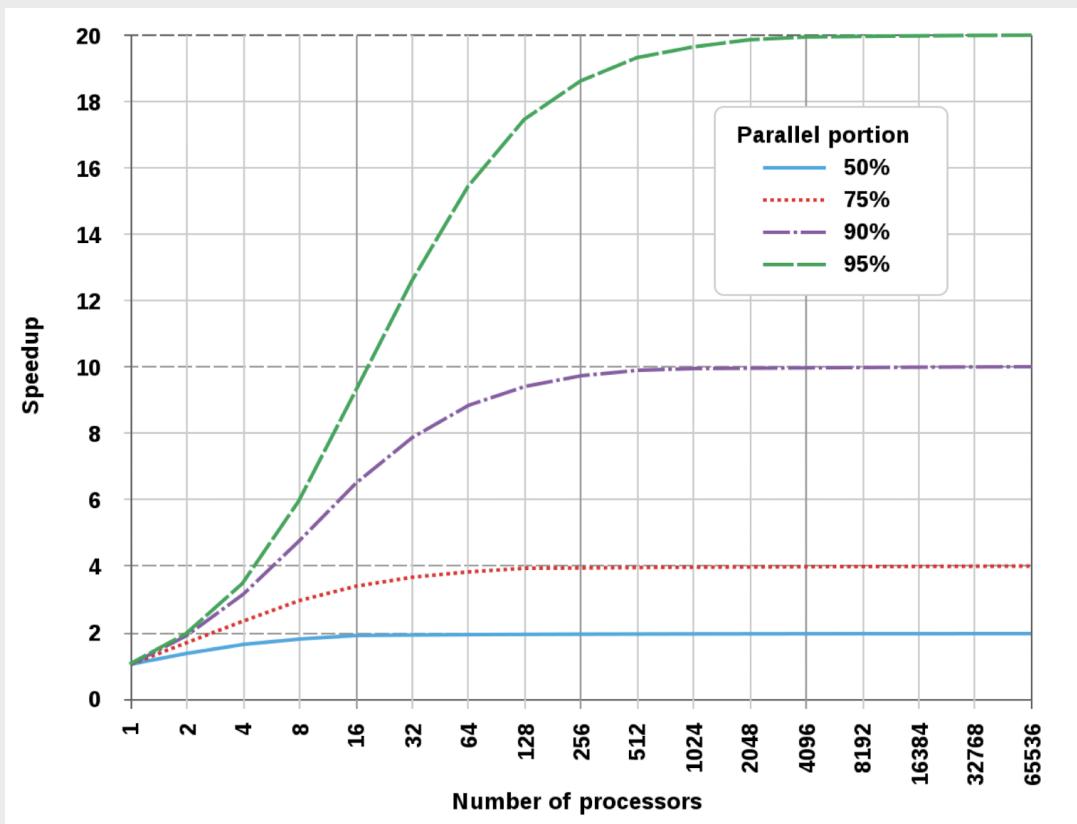


Figura 1.3: Grafico della legge di Amdahl

L'immagine illustra la Legge di Amdahl, che descrive il limite massimo di accelerazione ottenibile aumentando il numero di processori per un dato task. I grafici mostrano come lo **speedup** (accelerazione del calcolo) cresce con il numero di processori per diverse percentuali di codice parallelizzabile (50%, 75%, 90%, 95%). Si nota che, anche con un elevata **parallelizzabilità**, il guadagno tende ad assestarsi, evidenziando che oltre un certo punto aggiungere più processori non porta benefici significativi. Questo principio è fondamentale nella progettazione di sistemi paralleli e nell'ottimizzazione delle prestazioni computazionali.

Capitolo 2

Organizzazione dei Sistemi di calcolo

2.1 Struttura di un computer

La base della struttura è costituita dalla **Scheda Madre** sulla quale troviamo:

- La **CPU**, il cervello del PC;
- Il **BUS** (vedi cap. 2), costituito da una serie di piste su circuito stampato;
 - Spesso sono presenti più BUS secondo diversi standard.
- Le **Schede di I/O**.

2.1.1 Definizione di Computer

Definizione 2.1: Computer

Macchina organizzata, composta da una serie di dispositivi elettronici che può risolvere problemi eseguendo delle istruzioni da un insieme predefinito.

2.1.2 Strutture di Interconnessione

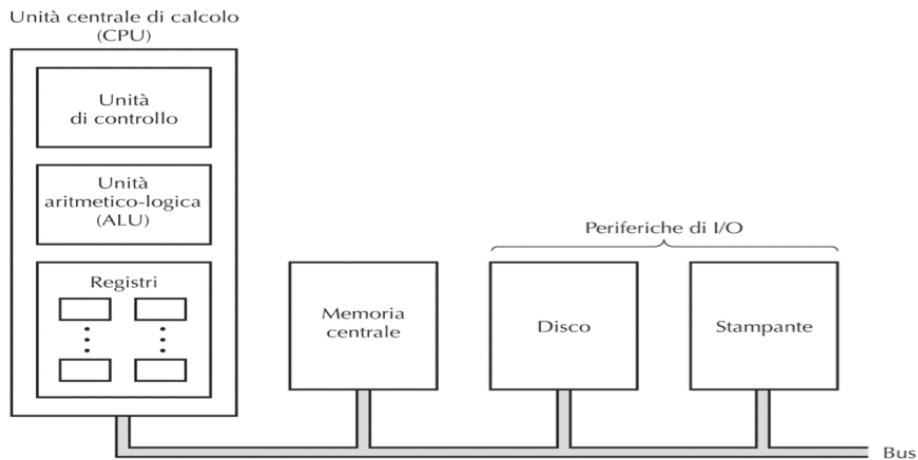


Figura 2.1: Schema delle interconnessioni

L'insieme dei componenti all'interno di un computer necessita di comunicare tra loro, e per farlo usano dei "canali" ad hoc chiamati **strutture di interconnessione**.

Le caratteristiche della *struttura di interconnessione* dipendono dal tipo di dato che deve trasferire. I **Moduli I/O** trasferiranno tipi di dato diversi rispetto alla **CPU** o alle **Memorie**.

La struttura di interconnessione più importante è l'**Interconnessione a BUS**.

2.2 Il BUS

Il BUS è una struttura di interconnessione basata sulla condivisione; è composto da **Linee** che collegano 2 o più moduli/dispositivi.

Nota Bene

È importante dire **LINEE**: non sono fili, cavi o collegamenti magici, sono linee stampate su circuito (se proprio non ti piace "linee" puoi dire **Piste**).

Solo un dispositivo alla volta può essere abilitato a usare il BUS. A tal proposito abbiamo 2 tipi di trasmissioni a BUS:

- Trasmissioni a Bus Seriali;
- Trasmissioni a Bus Parallele.

(*Dopo vedremo la differenza!*)

In un moderno calcolatore coesistono più BUS, ma quello che si occupa di collegare i moduli più importanti (**CPU**, **Memorie**, **Schede I/O**) si chiama **BUS di Sistema** (o, in inglese, *System BUS*).

2.2.1 Classificazione dei BUS

In un moderno calcolatore abbiamo:

- **BUS interni**: sono confinati all'interno di un singolo modulo;
- **BUS esterni**: collegano più moduli.

Nota Bene

È ovvio che questa definizione lascia il tempo che trova, dato che è molto dipendente dalla definizione di "singolo modulo". Le memorie sono un singolo modulo anche se composte fisicamente da blocchi diversi? E i bus che collegano le memorie come li consideriamo?

Cerchiamo di capire quando stiamo parlando di BUS interni/esterni in base al contesto.

Vedremo un approfondimento nella nota **BUS - Approfondimento**.

Come si valutano le prestazioni di un BUS?

1. Velocità di trasferimento;
2. Numero di bit (la larghezza del BUS).

Inoltre, le prestazioni sono estremamente dipendenti dal tipo di circuiteria di controllo (generalmente basati sul **tri-state buffer**).

2.2.2 Schema Logico di un BUS

Abbiamo appena descritto il BUS come un insieme di piste; vediamo ora che diverse piste possono essere di tipi diversi in base al tipo di dato che trasmettono.

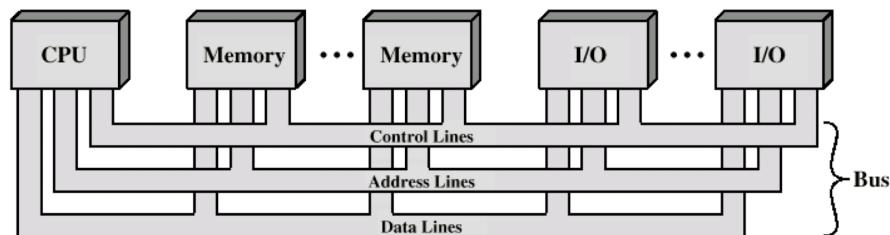


Figura 2.2: Schema logico delle linee del BUS

Come è possibile vedere nell'immagine (Fig. 2.2), un singolo BUS può essere composto da 3 linee principali o sotto-BUS:

Control BUS

Si occupa di trasmettere bit di controllo come:

- Lo stato di una risorsa (Read/Write);
- Il segnale di Clock;
- Richieste di Interrupt.

Address Bus

Ci dice la sorgente e la destinazione del dato trasmesso. La sua ampiezza determina lo spazio di memoria indirizzabile esplicitamente.

- Esempio: con un Address BUS di 16 bit sarà possibile indirizzare 64K spazi di memoria (2^{16}).

Data BUS

La linea che si occupa di trasmettere dati o istruzioni. Può essere di diverse ampiezze, ad esempio:

- 8 bit
- 16 bit
- 32 bit
- 64 bit

2.3 Tecnologie per BUS esterni

La maggior parte dei **BUS esterni** è realizzata tramite collegamenti elettrici:

- Schede di BUS, con piste di collegamento e connettori montati sulla scheda;

- Cavi elettrici flessibili con connettori;
- Alcuni BUS, ad altissime prestazioni, sono realizzati in fibra ottica (FiberChannel);
- Alcuni BUS si basano sull'etere (onde radio, Bluetooth).

2.4 La CPU

La CPU è colui che si occupa di gestire l'intera struttura del sistema ed è responsabile di tutte le **operazioni aritmetiche e logiche (ALU)** sui dati contenuti nei registri.

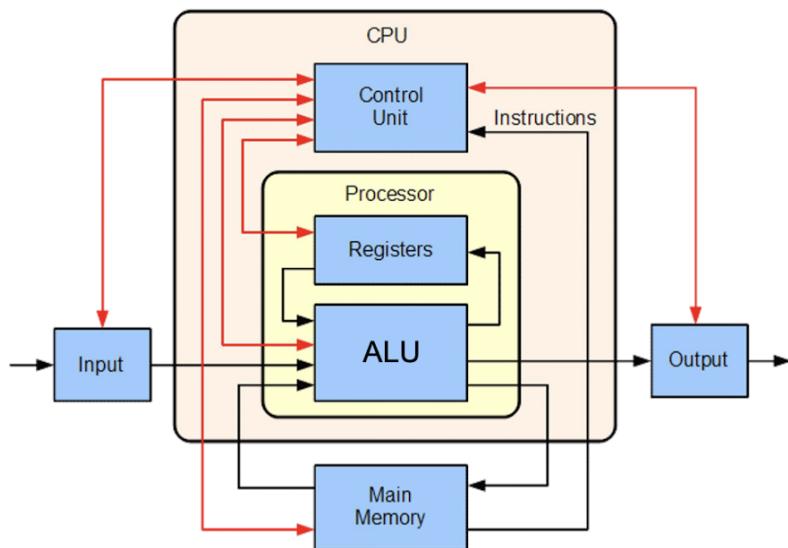


Figura 2.3: Struttura interna della CPU

Un **ciclo elementare** da prendere come esempio può essere:

- Due operandi sono inviati alla ALU;
- La ALU li somma;
- Il risultato viene memorizzato nella memoria.

Perché parliamo di ciclo? Nello specifico ci stiamo riferendo al **Ciclo Fetch-decode-execute**.

2.4.1 Differenza tra Esecuzione e Interpretazione

Esecuzione

In questo caso abbiamo un **Set prestabilito di istruzioni** che vengono direttamente eseguite dall'hardware.

- È un approccio complesso;
- Le istruzioni possibili sono limitate;
- L'esecuzione è molto più efficiente.

Interpretazione

In questo caso l'hardware esegue poche istruzioni elementari dette **micro-istruzioni**. Le istruzioni che usiamo nel nostro progetto vengono scomposte da un **decoder** in tante microistruzioni.

Questo paradigma permette:

- Repertorio di istruzioni più esteso;
- Hardware più compatto (meno dispendio economico);
- Più flessibilità di progetto.

Nota Bene

Possiamo vedere due tipi diversi di architetture basati su questi concetti nella sezione dedicata alle **Architetture CISC e RISC**.

2.4.2 Architetture CISC (Complex Instruction Set Computer)

- **Interpretazione** tramite micropogramma.
- Repertorio esteso (alcune centinaia di istruzioni).
- Istruzioni anche su memoria.
- Molti cicli di macchina per istruzione.

Architetture RISC (Reduced Instruction Set Computer)

- **Esecuzione diretta**.
- Repertorio ristretto (alcune decine di istruzioni).
- Istruzioni prevalentemente su registri.
- Un'istruzione per ciclo di macchina (del data path).

Nota Bene

Le architetture RISC sono caratterizzate dalla disponibilità di **grandi quantità di registri**. Al contrario, le architetture CISC dispongono di registri limitati.

Tabella Confronto CISC vs RISC

CISC	RISC
Enfasi sull'Hardware	Enfasi sul Software
Istruzioni complesse (molti cicli)	Istruzioni semplici (1 ciclo)
Memoria-a-Memoria	Registro-a-Registro (Load/Store)
Pochi registri	Molti registri

Tabella 2.1: Differenze principali tra CISC e RISC

2.4.3 Microprogrammazione

L'hardware può eseguire microistruzioni:

- **Trasferimenti tra registri;**
- **Trasferimenti da e per la memoria;**
- **Operazioni della ALU su registri.**

Ciascuna istruzione viene scomposta in una sequenza di microistruzioni. L'unità di controllo della CPU esegue un microprogramma per effettuare l'interpretazione delle istruzioni macchina.

Il microprogramma è contenuto in una memoria ROM sul chip del processore.

Vantaggi:

- Progetto strutturato;
- Semplice correggere errori;
- Facile aggiungere nuove istruzioni.

2.4.4 Ciclo Fetch-Decode-Execute

Per eseguire ogni singola istruzione la CPU deve seguire una serie di precisi STEP; l'insieme di questi è appunto chiamato **Ciclo Fetch-Decode-Execute**.

Vediamoli nel dettaglio:

1. **(FETCH)** Carica l'istruzione dalla memoria al registro *Instruction Register* (IR).
2. Incrementa il *Program Counter* (PC).
3. **(DECODE)** Decodifica l'istruzione.
4. Se l'operazione usa un dato in memoria, ne calcola l'indirizzo.
5. Carica l'operando in un registro (qualsiasi).
6. **(EXECUTE)** Esegue l'istruzione.
7. **(STORE)** Memorizza il risultato.
8. Siamo pronti a eseguire la prossima istruzione e torniamo allo step 1.

Nota Bene

Gli accessi alla memoria sono effettuati **sempre al passo 1, e non sempre ai passi 4, 5 e 7.**

```

public class Interp {
    static int PC;           // il program counter contiene l'indirizzo dell'istruzione successiva
    static int AC;           // l'accumulatore, un registro per i calcoli aritmetici
    static int instr;         // un registro che contiene l'istruzione corrente
    static int instr-type;   // il tipo d'istruzione (opcode)
    static int data-loc;     // l'indirizzo del dato, o -1 se non c'è
    static int data;          // contiene l'operando corrente
    static boolean run-bit = true; // un bit che può essere impostato a 0 per arrestare la macchina

    public static void interpret(int memory[ ], int starting-address) {
        // Questa procedura interpreta programmi per una semplice macchina le cui istruzioni hanno
        // un solo operando in memoria. La macchina ha un registro AC (accumulatore), utilizzato per
        // i calcoli aritmetici. L'istruzione ADD, per esempio, somma un intero memorizzato in memoria al registro AC.
        // L'interprete continua finché il bit di esecuzione non viene impostato a 0 dall'istruzione HALT.
        // Lo stato di un processo eseguito su questa macchina è costituito dalla memoria, dal
        // contatore di programma, dal bit di esecuzione e dal registro AC. I parametri di input consistono
        // nell'immagine della memoria e nell'indirizzo di partenza.

        PC = starting-address;
        while (run-bit) {
            instr = memory[PC];           // preleva l'istruzione successiva e la memorizza in instr
            PC = PC + 1;                 // incrementa il program counter
            instr-type = get-instr-type(instr); // determina il tipo d'istruzione
            data-loc = find-data(instr, instr-type); // trova la posizione del dato (-1 se non c'è)
            if (data-loc >= 0)           // se "data-loc" vale -1, allora non c'è nessun operando
                data = memory[data-loc] // preleva il dato
            execute(instr-type, data);   // esegue l'istruzione
        }
    }

    private static int get-instr-type(int addr) { ... }
    private static int find-data(int instr, int type) { ... }
    private static void execute(int type, int data) { ... }
}

```

Figura 2.4: Diagramma del ciclo Fetch-Decode-Execute

2.5 Principi di Progettazione e Parallelismo

Attualmente possiamo distinguere 5 diversi tipi di paradigmi da perseguire in fase di progettazione:

1. Eseguire tutte le istruzioni direttamente dall'hardware

- Esecuzione diretta per le istruzioni più comuni.
- Interpretazione solo per le istruzioni più complesse e meno comuni.

2. Massimizzare la frequenza di emissione delle istruzioni

- Determina il valore dei MIPS (Millions Instructions per Second).
- Ruolo importante del parallelismo.

3. Semplificare la decodifica delle istruzioni

- Rendere le istruzioni regolari, di lunghezza fissa, pochi campi.
- In questo modo la decodifica avrà meno variabili, ergo più semplice.

4. Limitare i riferimenti alla memoria (solo LOAD e STORE)

- L'accesso alla memoria richiede tempo.
- Tutte le operazioni dovrebbero operare su registri.

- Operazioni sovrapponibili agli accessi alla memoria.

5. Aumentare il numero di registri disponibili

- Limitare l'accesso alla memoria richiesta dalle operazioni.
- Disporre sempre di registri liberi per le operazioni.

2.5.1 Parallelismo

Ad oggi il paradigma migliore (e più diffuso) per massimizzare le prestazioni è il parallelismo; nel nostro caso ne abbiamo di 2 tipi:

1. Parallelismo a livello di Istruzione

- Diverse istruzioni o porzioni di esse vengono eseguite contemporaneamente.

2. Parallelismo a livello di processore

- Diversi processori lavorano congiuntamente sullo stesso problema.
- Fattori di parallelismo molto elevati.
- Diversi tipi di interconnessione e cooperazione.

Uno dei più importanti metodi per implementare il parallelismo è il **Pipelining** (che vedremo meglio in seguito). Vediamo adesso altri tipi di parallelismo, meno importanti rispetto alla pipeline ma pur sempre degni di nota.

* Un altro paradigma molto utilizzato sono le **Architetture Multiscalari**.¹

Architetture Superscalari

In queste architetture si avviano più istruzioni (4-6 contemporaneamente). Ognuna di esse con una propria **Pipeline**.

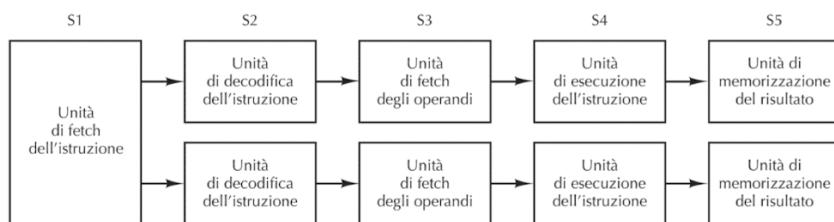


Figura 2.5: Architettura superscalare con pipeline multiple

Ovviamente abbiamo dei problemi a fronte di questo paradigma, principalmente 2:

- **Interdipendenza tra istruzioni:** diverse istruzioni possono richiedere stesse risorse.
- **Nessuna istruzione può dipendere da una eseguita in parallelo:** ad esempio gli operandi di un'istruzione non possono essere il risultato di un'altra.

È possibile avere una versione della stessa architettura ma con singola pipeline, dotata di unità funzionali multiple.

Viene duplicato solo lo stadio più lento.

¹Vedi approfondimento nella sezione delle note a margine a fine capitolo: 2.9.1

- Lo stadio S3 deve poter lanciare istruzioni ad una frequenza superiore all'esecuzione dello stadio S4.
- La CPU contiene diverse unità funzionali indipendenti.
- Architettura adottata nei processori Intel Core.

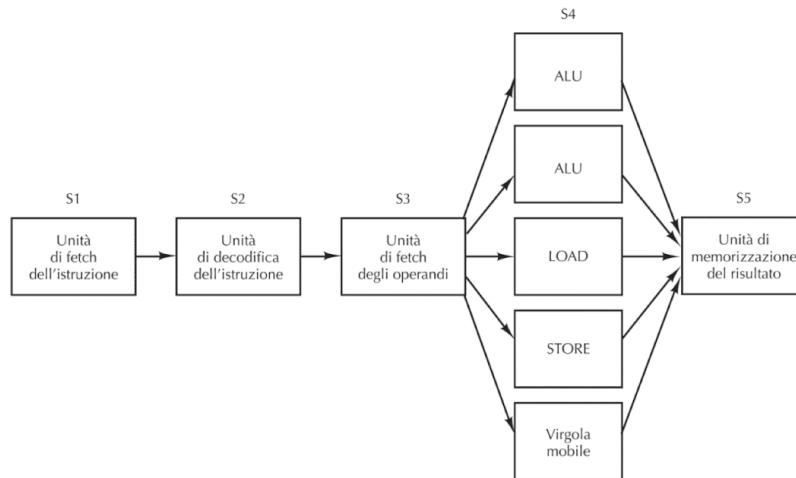


Figura 2.6: Architettura superscalare a singola pipeline con unità multiple

2.5.2 La Pipeline

Nota Bene

Ricordiamo che parliamo di Hardware e non Software. *La pipeline stessa è un insieme di componenti hardware* (ne abbiamo una per core).

Questo tipo di approccio permette di eseguire più operazioni contemporaneamente. Vediamo i punti chiave:

1. Ogni istruzione deve essere spezzettata in più *parti* (nello specifico 5).
2. Per ogni *parte* dell'istruzione abbiamo un supporto Hardware, quindi un modulo dedicato per svolgere solo quello specifico compito.
3. L'obiettivo è tenere occupati più moduli possibili contemporaneamente; appena se ne libera uno deve essere richiesto dall'istruzione successiva.

Vediamo uno schema di funzionamento:

L'istruzione (o i bit di essa) non passa da un componente all'altro; viene divisa nei moduli della pipeline che si attivano in maniera sequenziale (gestiti da un controllo) solo perché la *fase n* necessita della *fase n - 1*.

- S_1, S_2, \dots, S_5 sono i moduli sopracitati, ma possiamo anche interpretarli come stati in cui si possono trovare le istruzioni in un certo istante al fine di comprendere meglio questo schema.
- Come possiamo vedere all'istante di tempo 1 abbiamo una sola istruzione (chiamata 1 essendo la prima) e si trova nello stato 1.

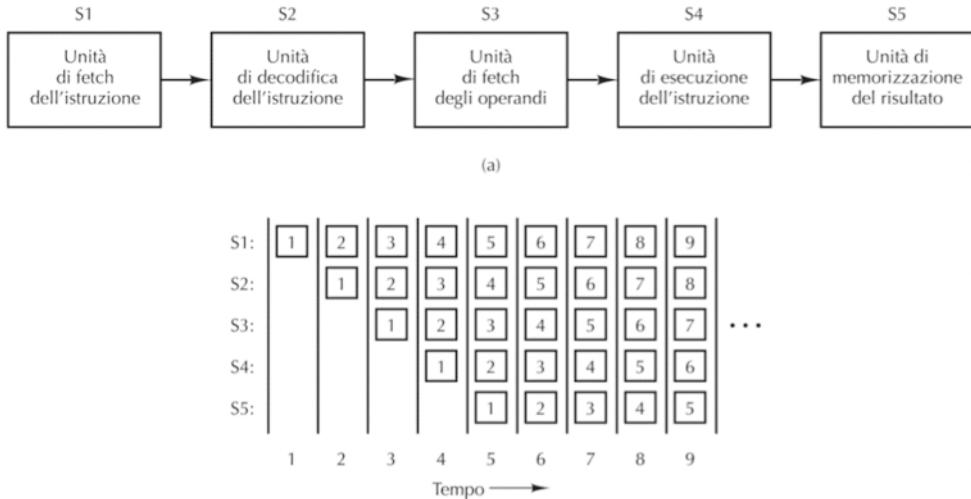


Figura 2.7: Schema temporale della Pipeline

- Notiamo che fino allo stato 5 l'istruzione 1 è passata per tutti gli stati, liberando dietro di sé spazio per le istruzioni successive.
- All'istante di tempo 6 l'istruzione 1 abbandona il nostro schema dato che ha terminato il suo ciclo di vita.
- Fanno la stessa cosa le istruzioni successive negli istanti che seguono.

Vantaggi e Svantaggi

Vantaggi:

- Notiamo che dall'istante 5 in poi abbiamo un'istruzione portata a termine per ogni colpo di clock (OTTIMO!).
- Il fattore di guadagno è pari al numero di stati della pipeline.

Svantaggi:

- Le istruzioni che richiedono **salti** possono complicare molto la pipeline o rallentarla, dato che non si può più garantire la sequenzialità degli stadi delle istruzioni ed è complesso prevederla.
- Le istruzioni devono ovviamente essere **indipendenti**; in caso contrario si genera uno **stallo** (sarà necessario attendere il termine di un'istruzione prima di eseguirne un'altra, il che va *contro il concetto stesso della pipeline*).

Tempi e Metriche

Sia n = numero di stadi.

- **Ciclo di clock:** $T \cdot \text{nsec}$
- **Frequenza:** $10^9 / T \text{ Hz}$
- **Latenza:** $n \cdot T$
- **Larghezza di banda:** Numero di istruzioni eseguite al secondo.

Parallelismo mediante Multithreading

Più Processi vengono eseguiti in maniera concorrente. Ho 2 opzioni:

- **Grana fine:** Ogni processo ha a disposizione un quanto di tempo fisso per poi cedere il posto al Thread successivo.
 - In questo modo non ho stalli ma effettuo molti *context switch*, che è un'operazione pesante per il sistema.
- **Grana grossa:** Cambio processo quando quest'ultimo si interrompe (perché ha finito o ha una fase di stallo), o se raggiunge un quanto di tempo massimo.
 - Effettuo pochi context switch ma presenta *stalli* dato che devo attendere un'operazione *nop* per sapere quando passare al prossimo processo.

Il Thread che è attualmente in esecuzione esegue 2 istruzioni per ciclo fino a quando non raggiunge uno stallo; in quel caso si passa al Thread successivo.

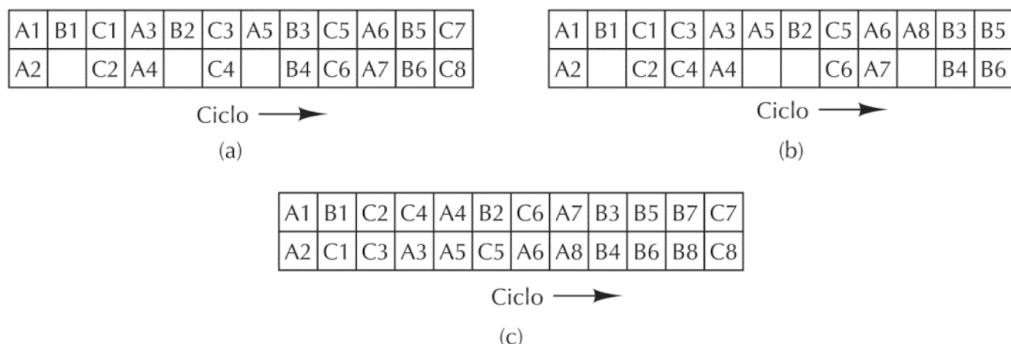


Figura 2.8: Gestione degli stalli nel multithreading

Nota Bene

L'obiettivo è quello di tenere la CPU perennemente impegnata.

Parallelismo a livello della CPU

Per avere guadagni fino a 2 ordini di grandezza è necessario utilizzare più processori simultaneamente, anche in questo caso abbiamo 2 vie:

Parallelismo dei dati (SIMD) Le stesse istruzioni vengono eseguite contemporaneamente da due processori distinti ma su dati diversi.²

Parallelismo dei task (MIMD) Diversi processori eseguono diversi task.

2.6 La Memoria Centrale e di Massa

La **memoria centrale** è quella che chiamiamo tipicamente RAM; è dove operano i processi e il sistema operativo stesso. Difatti contiene sia dati che programmi.

²Vedi Approfondimento nella sezione a fine capitolo delle note a margine: 2.9.2

- **Cella:** minima unità di memoria indirizzabile.
- **Word:** insieme di celle consecutive (insieme K di Byte).
- **Indirizzo:** nome identificativo della cella.
 - **Spazio di indirizzamento:** una volta scelto quanti bit usare per generare indirizzi abbiamo 2^m indirizzi possibili (m numero di bit), quindi 2^m celle indirizzabili.

2.6.1 Endianness: Organizzazione dei Byte

Abbiamo 2 modi per salvare le parole in memoria:

- **LittleEndian:** Byte più significativo a destra.
 - Il più utilizzato (incentivato da Intel).
- **BigEndian:** Byte più significativo a sinistra.
 - Più comodo per la lettura di stringhe (promosso da IBM).

2.6.2 Tipi di schede di memoria

SIMM (Single Inline Memory Module)

Vecchi moduli di memoria, usati negli anni '90.

- Hanno **72 piedini** e un bus di **32 bit**.
- Poiché i processori Pentium avevano un bus a 64 bit, servivano **due moduli SIMM in coppia** per funzionare.
- Capacità tipica: da pochi MB fino a \sim 128 MB per modulo.

DIMM (Double Inline Memory Module)

Successori dei SIMM, ancora oggi il formato standard nei PC desktop.

- Hanno **120 o 240 piedini** e un bus di **64 bit**, quindi **basta un solo modulo** per lavorare con CPU a 64 bit.
- Capacità iniziali di poche centinaia di MB, oggi fino a decine di GB per modulo.
- Ogni modulo è composto da più chip di memoria (es. 8 chip = 1 word per volta).

SO-DIMM (Small Outline DIMM)

Versione compatta dei DIMM.

- Usata soprattutto nei **notebook**, mini-PC e dispositivi compatti.
- Più corta (67 mm circa) rispetto ai DIMM standard (133 mm).

DDR (Double Data Rate, DDR2...DDR5)

Tecnologia che permette di **trasferire dati sia sul fronte di salita che di discesa del clock**, raddoppiando la velocità rispetto alla SDRAM classica.

- Ogni nuova generazione (DDR → DDR2 → ... → DDR5) aumenta frequenza, banda passante e riduce consumo.
- Introdotto anche un sistema di **pipeline** per ottimizzare lettura e scrittura.

Bit di parità / ECC (Error Correction Code)

Alcuni moduli hanno un bit di parità o un sistema ECC: servono per rilevare (parità) o correggere (ECC) errori di memoria.

- Utilizzati soprattutto nei **server** e nei sistemi critici; nei PC consumer spesso assenti.
³.

2.6.3 Gerarchie della memoria

Andando verso il basso nella piramide:

- **Scende la capacità;**
- **Aumenta il tempo di accesso;**
- **Diminuisce il costo per bit.**

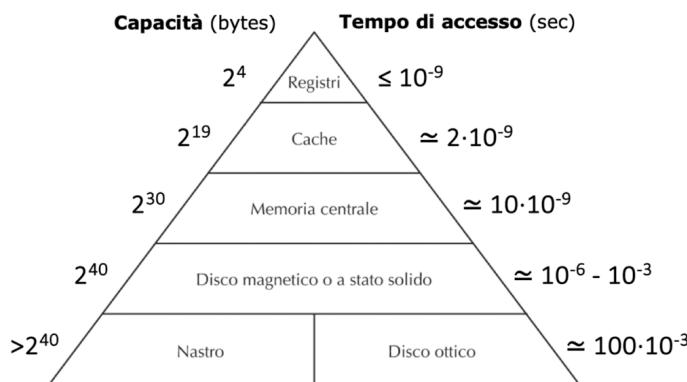


Figura 2.9: Piramide delle gerarchie di memoria

Nota Bene

I registri sono l'unico livello a diretto contatto con la CPU. Esiste un forte **Performance Gap** tra processi e memorie creato negli anni.^a

^aVedi approfondimento nella sezione a fine capitolo delle note a margine: 2.9.4

2.6.4 Memoria CACHE

Come possiamo vedere dallo schema delle gerarchie, prima della nostra Memoria centrale troviamo un altro stadio: la **Cache**. Ovviamente i registri non li consideriamo neanche, dato che sono a diretto contatto con la CPU (sono interni al processore e non è possibile cambiarli o aumentarli).

L'esistenza della cache è dovuta al fatto che la **Memoria centrale è sempre più lenta della CPU**. La cache opera alla stessa velocità del processore in modo da compensare i ritardi della memoria centrale; contiene le ultime locazioni di memoria a cui è stato eseguito l'accesso.

In questo modo quando la CPU richiede un indirizzo passa prima per la Cache: se è presente si evita un accesso alla memoria, altrimenti si prosegue con il percorso *tradizionale*.

- **Cache hit** = il dato che cerchi è già nella cache → velocissimo.
- **Cache miss** = il dato non c'è nella cache → tocca prenderlo dalla memoria lenta.

Quando leggi la **stessa parola k volte di fila**:

1. Solo la prima volta vai in memoria (miss).
2. Le altre $k - 1$ volte la trovi in cache (hit).

³Vedi approfondimento nella sezione a fine capitolo: 2.9.3

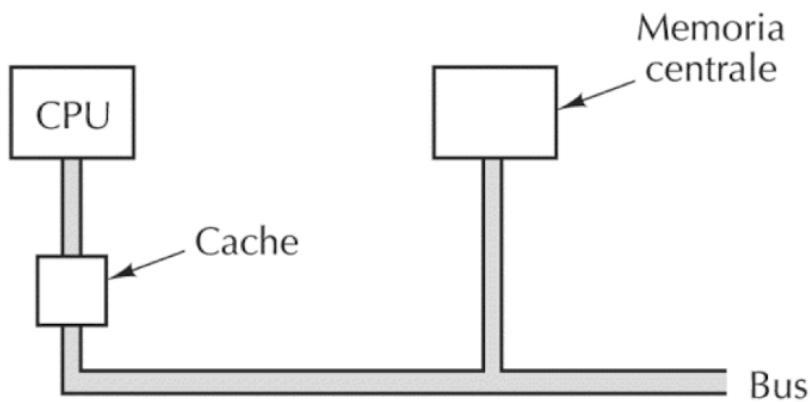


Figura 2.10: Schema di interazione CPU - Cache - Memoria

Analisi Prestazionale

Formula del cache hit ratio (quanto spesso trovo il dato in cache):

$$H = \frac{k-1}{k}$$

Tempo medio di accesso: Siano:

- m = tempo per leggere dalla memoria (lento);
- c = tempo per leggere dalla cache (veloce).

Il tempo medio di accesso A è:

$$A = c + (1 - H)m$$

Nota Bene

Significato: Parti dal tempo della cache c , poi aggiungi una “penalità” per le volte che non trovi il dato $(1 - H)$ e devi andare in memoria (m).

2.6.5 Memorie di Massa

Hard Disk

- **Dimensioni:** < 10cm (1.8'', 2.5'', 3.5''), densità $\sim 25\text{Gb/cm}^2$.
- **Struttura:**
 - Tracce concentriche divise in settori (dati, preambolo, spazio tra settori, ECC).
 - Velocità di rotazione: 5.400 - 10.800 RPM.
 - Velocità di trasferimento: 150 MB/sec.
- **Tempi:**
 - T_{seek} : 5-10ms (spostamento testine).
 - T_{lat} : 3-6ms (latenza rotazionale).
 - $T_{acc} = T_{seek} + T_{lat}$.

Controller per Hard Disk

- **Tipi:** IDE, E-IDE, Serial ATA, SCSI (fino a 640 MB/sec).
- **Evoluzione:**
 - **IDE:** limite 504 MB, 4MB/sec.
 - **E-IDE:** LBA a 28 bit (128GB), 17MB/sec.
 - **ATA-3/ATAPI-5/6:** fino a 100MB/sec, LBA a 48 bit (128PB).
 - **SATA:** > 500MB/sec, tensioni più basse.

RAID (Redundant Array of Inexpensive Disks)

Obiettivi: Parallelismo, ridondanza, resistenza ai guasti.

- **RAID 0:** Striping senza ridondanza (migliora velocità, peggiora MTBF).
- **RAID 1:** Mirroring (duplicazione dati, ottimo per lettura e fault tolerance).
- **RAID 2/3:** Striping a bit/byte con parità (sincronizzazione dischi, overhead elevato).
- **RAID 4/5:** Striping a blocchi con parità distribuita (RAID 5 elimina collo di bottiglia del disco di parità).

Dischi Ottici

- **Struttura:**
 - CD/CD-ROM: traccia a spirale con pit/land, blocchi da 2KB.
 - CD-R: strati di lacca, oro, dye, policarbonato.
 - DVD: doppio lato/doppio strato con semiriflettenti.
- **Tecnologia:** Laser a infrarossi per lettura/scrittura.

SSD (Solid State Drive)

- **Tecnologia:** Hot-carrier injection (stato transistor fisso).
- **Vantaggi:**
 - Tempi di accesso nulli ($T_{seek} = 0$), > 200MB/sec.
 - Adatti a dispositivi mobili.
- **Svantaggi:**
 - Costo elevato (~1€/GB vs 1 cent/GB HDD).
 - MTBF limitato (~100.000 cicli di scrittura).

2.7 Dispositivi di Input/Output

Iniziamo a definire cosa sono i dispositivi di Input/Output: sono tutti i dispositivi che possiamo vedere/toccare/usare concretamente quando abbiamo a che fare con un calcolatore (ad esempio tastiere, schermo, mouse e tanto altro).

Vengono connessi al BUS tramite una circuiteria chiamata **device controller**. Non possono essere direttamente collegati al BUS di sistema per diversi motivi, ma i 2 principali sono:

1. Sono di diversi ordini di grandezza più lenti rispetto a tutti gli altri apparecchi che usufruiscono del bus (potrebbero rallentare drasticamente l'intero sistema).
2. Ogni dispositivo potrebbe richiedere un tipo di dato diverso; non è detto che sia facilmente compatibile con il nostro BUS.

2.7.1 Moduli Input/Output

Si occupano di fare da ponte tra le interfacce e il BUS di sistema (corrispondono a quelli che abbiamo precedentemente chiamato *device controller*).

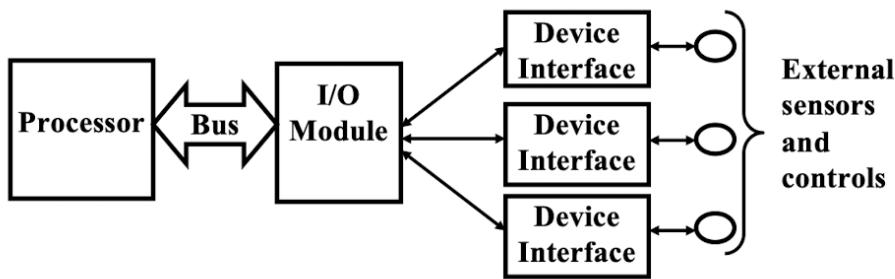


Figura 2.11: Il modulo I/O come ponte verso il Bus

Struttura Logica

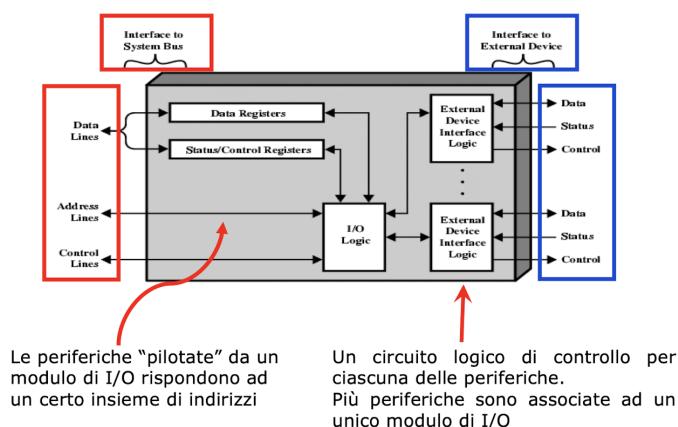


Figura 2.12: Struttura logica di un modulo I/O

2.8 Modalità di I/O

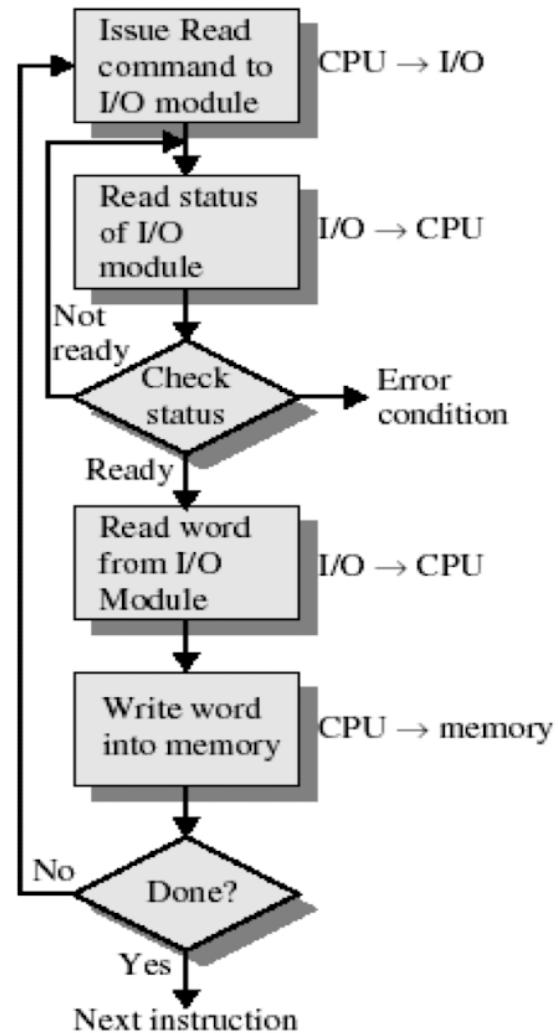
1. I/O programmato

In questo caso la CPU si occupa di controllare la totalità delle operazioni, mediante l'esecuzione diretta di istruzioni di I/O.

In questa immagine possiamo vedere il flusso di lavoro nel caso di I/O programmato.

Tutte le istruzioni del flusso vengono eseguite dalla CPU, e devono essere esplicitamente nelle istruzioni del programma.

Si può notare che al termine abbiamo un ciclo di **polling** per verificare se abbiamo o no terminato l'esecuzione del programma.



2. I/O pilotato da Interrupt

La CPU in questo caso non necessita di Polling, quindi non ha più cicli di attesa. Verifica se è stato emesso un interrupt al termine di ogni istruzione (lo fa all'interno del ciclo stesso, questo non genera quindi sovraccarichi inutili di lavoro). In caso positivo dovrà effettuare un **context switch**.

1. La CPU sta eseguendo un processo e si trova all'istruzione x .
2. **Arriva un interrupt.**
3. La CPU emette un *Acknowledge* per segnalare la corretta rilevazione dell'interrupt e dell'imminente gestione di esso.

4. Esegue le istruzioni contenute nel **IV** (Interrupt Vector, contenente le istruzioni richieste dall'interrupt).
5. Terminate le istruzioni del IV riprende con l'esecuzione del processo interrotto dall'istruzione $x + 1$.

3. Accesso diretto alla memoria - DMA

I moderni dispositivi sono dotati di moduli DMA che “bypassano” la CPU in fase di esecuzione delle istruzioni.

1. Il DMA invia un interrupt alla CPU per richiedere *scrittura* o *lettura*.
2. La CPU interrompe il suo flusso di lavoro per inviare **dei segnali di controllo** al DMA.
3. La CPU riprende il suo flusso e intanto il DMA si occupa del trasferimento in maniera indipendente.
4. Al termine del trasferimento il DMA invia nuovamente un interrupt per segnalare la fine del lavoro.

Segnali di controllo:

- **Locazione Iniziale:** cella in memoria da cui partire per ricevere/trasferire dati.
- **Numero di parole:** quante parole trasferire (immagino che in alcuni casi, pensa a una keyboard, sia indefinito).
- **Indirizzo:** indirizzo che identifica il modulo I/O.

2.8.1 Interfacce I/O

1. **Parallela:** Trasmissione di più ‘bit’ alla volta.
2. **Seriale:** Trasmissione di un singolo ‘bit’ alla volta.
 - (a) **Point-To-Point:** due dispositivi che hanno un loro canale dedicato per la comunicazione e il trasferimento dati.
 - (b) **Multipoint:** Canali condivisi tra più dispositivi.

2.8.2 USB (Universal Serial Bus)

Definizione 2.2: USB

Bus Universale di I/O che permette il trasferimento tra moduli a diverse velocità di trasmissione.

Il trasferimento avviene mediante **frame**; ne possiamo avere di 4 tipi:

1. **Controllo:** Per configurare e interrogare dispositivi.
2. **Isocrono:** Per interrogazione sincrona di dispositivi *real-time*, hanno un ritardo prevedibile.
3. **Bulk:** Trasferimento massiccio di dati (grandi dimensioni).
4. **Interrupt:** Simula Interrupt includendo tutte le richieste emesse in un intervallo di tempo.

Esistono quattro tipi di pacchetti:

- **Token:** Inviato dal Root Hub al dispositivo.
 - **SOF** (Start of Frame);
 - **IN/OUT** (richiesta di lettura/scrittura dati);
 - **setUp** (configurazione).
- **Dati:** 8bit sync; 8bit packet type(id); payload fino a 64 byte; 16bit CRC.
- **Handshake:** ACK/NACK (acknowledge o errore).
- **Speciali:** (vedi foto).

USB PID bytes					
Type	PID value (msb-first)	Transmitted byte (lsb-first)	Name	Description	
Reserved	0000	0000 1111			
Token	1000	0001 1110	SPLIT	High-bandwidth (USB 2.0) split transaction	
	0100	0010 1101	PING	Check if endpoint can accept data (USB 2.0)	
Special	1100	0011 1100	PRE	Low-bandwidth USB preamble	
			ERR	Split transaction error (USB 2.0)	
Handshake	0010	0100 1011	ACK	Data packet accepted	
	1010	0101 1010	NAK	Data packet not accepted; please retransmit	
	0110	0110 1001	NYET	Data not ready yet (USB 2.0)	
	1110	0111 1000	STALL	Transfer impossible; do error recovery	
	0001	1000 0111	OUT	Address for host-to-device transfer	
Token	1001	1001 0110	IN	Address for device-to-host transfer	
	0101	1010 0101	SOF	Start of frame marker (sent each ms)	
	1101	1011 0100	SETUP	Address for host-to-device control transfer	
	0011	1100 0011	DATA0	Even-numbered data packet	
Data	1011	1101 0010	DATA1	Odd-numbered data packet	
	0111	1110 0001	DATA2	Data packet for high-bandwidth isochronous transfer (USB 2.0)	
	1111	1111 0000	MDATA	Data packet for high-bandwidth isochronous transfer (USB 2.0)	

Time (msec) →

Figura 2.13: Struttura dei pacchetti USB

2.9 Note a Margine

2.9.1 Architetture Multiscalare

In questo tipo di Architetture abbiamo più processori o più computer che lavorano in maniera indipendente.

- **CPU multiple:** memoria condivisa
- **Computer multipli:** memoria separata.

Anche nel caso delle CPU multiple abbiamo diversi paradigmi applicabili. Il problema principale in questo caso è la gestione della memoria (in realtà anche la gestione dei thread ma non lo vediamo)

- **Memoria condivisa:** Tutte le CPU lavorano nello stesso spazio di memoria, il *BUS fa da collo di bottiglia*.
- **Memoria locale(+ condivisa):** non ho più il problema del BUS ma la memoria è a 2 livelli quindi *aumenta l'overhead generale*.

2.9.2 SIMD

Le architetture SIMD sono caratterizzate da pochi pilastri concettuali ma abbastanza restrittivi e che lasciano poco spazio alla libera interpretazione.

1. Qualsiasi processo può leggere o scrivere tutta la memoria.
2. Due o più processori comunicano leggendo o scrivendo in un'opportuna cella di memoria.
3. Di norma prevedono un'elevata capacità di interazione tra i processori (**tightly couple**)
4. C'è una sola copia del sistema operativo in esecuzione.

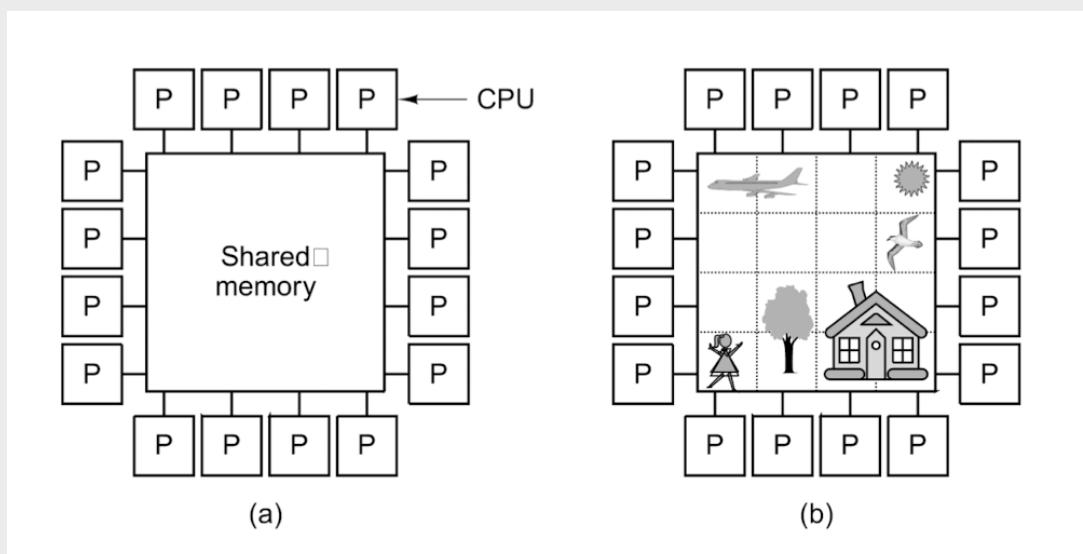


Figura 2.14: Schema dell'architettura SIMD

2.9.3 Algoritmi di rilevamento dell'errore - Hamming

Se voglio correggere o rilevare errori la regola aurea è questa:

Definizione 2.3: Hamming

Per correggere o rilevare errori sono necessari bit di rindondanza, più sono i bit di rindondanza più errori potrò rilevare/correggere.

- **Parità:** il numero di bit a 1 deve essere pari, in caso contrario so che ho un errore
 - funziona solo se voglio rilevare un singolo errore, (2 errori ad esempio si annullano)
- **Sottoinsiemi:** consiste nell'usare tanti bit per rappresentare pochi valori
- **Distanza di Hamming:** numero di bit diversi tra 2 configurazioni valide.

2.9.4 Performance Gap

Il Performance Gap è la marcata **disparità tecnologica** che si è creata tra la velocità di elaborazione dei processori e i tempi di accesso della memoria centrale

- **Tassi di crescita differenti:** Dagli anni '80, le prestazioni delle CPU sono aumentate di circa il 25% all'anno, mentre il tempo di accesso delle memorie DRAM è migliorato solo del 7% annuo.
- **Il collo di bottiglia:** Poiché la memoria è molto più lenta del processore, quest'ultimo finisce spesso per "aspettare" i dati, rallentando l'intero sistema.
- **Inadeguatezza del modello tradizionale:** Questa differenza ha reso il modello classico CPU-Memoria insufficiente per le esigenze moderne.

Per colmare questo divario, è stato necessario introdurre un livello intermedio molto veloce tra CPU e DRAM: la memoria Cache. La cache serve proprio a "nascondere" la lentezza della memoria principale sfruttando il principio di località

Capitolo 3

Funzionamento del BUS

3.1 Approfondimento e Gestione del BUS

In questo capitolo andiamo ad approfondire il BUS: in particolare vediamo come funziona e come va gestito, quindi quali sono i problemi da sopperire e quali sono le strategie utilizzate.

Purtroppo in questa prima nota introduttiva (sì, introduttiva anche se si tratta di un capitolo di approfondimento) dovrò un po' ripetermi al fine di aggiungere dettagli e smussare qualche definizione data frettolosamente in precedenza.

Iniziando dalle basi, possiamo constatare che:

Definizione 3.1: Il Calcolatore e il BUS

Un calcolatore elettronico può essere sinteticamente visto come un insieme di unità funzionali interconnesse tra loro mediante un BUS.

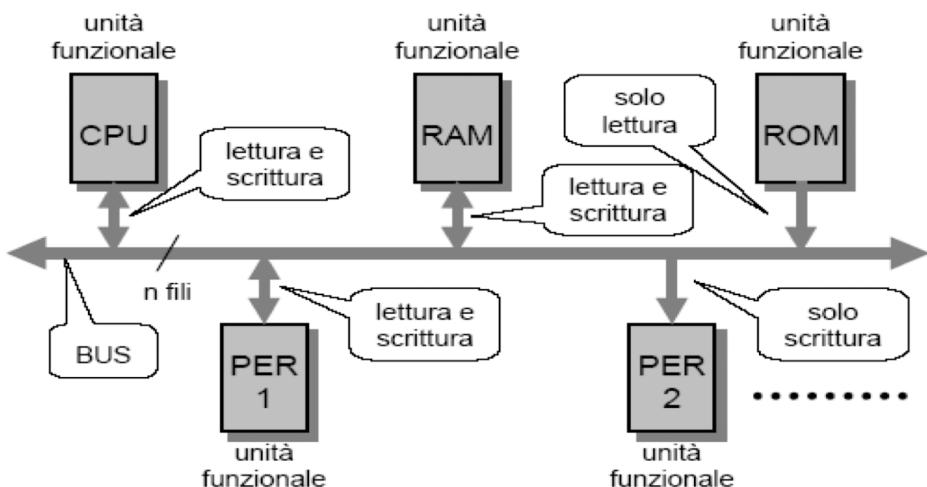


Figura 3.1: Unità funzionali connesse tramite BUS

Tutti i BUS hanno una propria velocità:

- I BUS **sincroni** sono scanditi dal clock.
- I BUS **asincroni** dipendono dalla frequenza della corrente che li attraversa, che viene a sua volta influenzata dalla circuiteria a monte e a valle del BUS.

3.1.1 Classificazione e Caratteristiche

In un moderno calcolatore esistono:

- **BUS interni (data path)**: confinati all'interno di una singola unità funzionale, collegano i blocchi funzionali contenuti nell'unità.
- **BUS esterni**: si estendono all'esterno dell'unità funzionale e la collegano alle altre unità funzionali. I BUS esterni del calcolatore sono solitamente standardizzati.

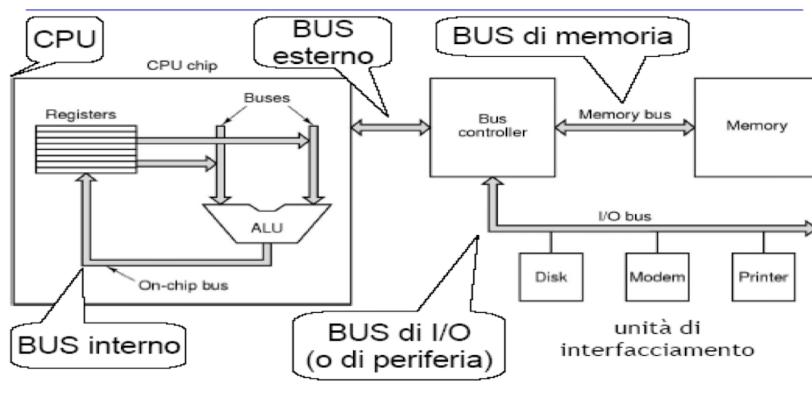


Figura 3.2: Differenza tra BUS interni ed esterni

Nota Bene

Qua sembrerebbe la stessa definizione discussa nella nota precedente, ma c'è un dettaglio che cambia le cose: ossia nominare i BUS interni come **datapath**. In elettronica, come già sappiamo, il datapath è l'insieme di tutti i componenti che svolgono calcoli ma non mutano il comportamento del sistema.

Specifiche Tecniche

- **Larghezza del BUS** = numero di linee.
- **Linee indirizzo**: determinano la dimensione dello spazio (di memoria) indirizzabile.
 - Con n bit di indirizzo ho 2^n locazioni.
- Ampliando le linee dati aumenta la velocità di trasmissione (**banda di trasferimento**).
- **Condivisione** di più segnali sulla stessa linea (*Multiplexing*) per diminuire i costi.
 - Il **Multiplexing** consiste nell'utilizzo delle stesse linee per diversi tipi di segnali a seconda della situazione; questo permette un grosso risparmio.
 - **Problema**: al crescere della velocità del bus aumenta il **BUS skew** (differenza nella velocità di propagazione dei segnali su linee diverse).

Nota Bene

Da adesso in poi adotteremo questa convenzione:

- Segnale basso = 0
- Segnale alto = 1

3.1.2 Terminologia dei Segnali

Per evitare confusione si parla di:

Segnale asserito Quando assume il valore (alto o basso) che provoca l'azione.

Segnale negato Altrimenti.

Si adotta la seguente notazione:

- S : segnale che è asserito alto.
- \bar{S} : segnale che è asserito basso.
- $S\#$: segnale che è asserito basso in contesto Intel.

Passiamo alle cose interessanti andando a parlare dell'**Arbitraggio del BUS**, per vedere come viene gestito il suo utilizzo, e la **Temporizzazione del BUS** per vederne il funzionamento.

3.2 Protocollo e Arbitraggio del BUS

La comunicazione tramite BUS viene regolata mediante un **protocollo di BUS**. Per potersi collegare al BUS tutte le unità possiedono un BUS driver in grado di svolgere le seguenti operazioni:

1. Collegarsi e scollegarsi elettricamente al BUS (tramite collegamento a tre stati o di tipo *open collector*);
2. Amplificare opportunamente i segnali da trasmettere/ricevere sul/dal BUS.

A questo scopo vengono sfruttati i **Dispositivi a 3 stati (Tri-state Buffer)** (vedi capitolo Memorie).

Come già sappiamo, **in qualunque istante una e una sola unità funzionale può avere il controllo del BUS**.

- Chiamiamo l'unità che possiede il controllo **MASTER**.
- Tutte le altre unità le chiamiamo **SLAVE**.
- Ogni BUS deve avere un MASTER per funzionare.

L'unità che possiede il ruolo del MASTER decide quali operazioni fare su quale unità e in quale istante di tempo. **Soltanente il ruolo del MASTER è ricoperto dalla CPU**.

Nota Bene

Ma non è sempre così: è possibile cedere temporaneamente il ruolo di MASTER tra le varie unità, e questo va gestito tramite qualche meccanismo chiamato **arbitraggio**.

Abbiamo 2 tipi di meccanismi principali:

- Arbitraggio **centralizzato**;
- Arbitraggio **distribuito**.

3.2.1 Arbitraggio Centralizzato

Per l'arbitraggio centralizzato (o **Daisy Chain**) abbiamo 3 regole fondamentali:

1. È presente un'unità funzionale dedita all'arbitraggio (per questo è detto centralizzato).
2. Alcune linee del BUS sono dicate a trasmettere le richieste delle unità funzionali all'arbitro.
3. È l'arbitro che realizza la cessione del controllo del BUS (cessione del ruolo MASTER).

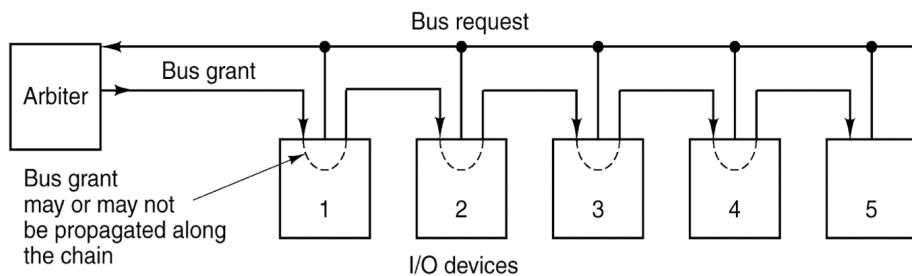


Figura 3.3: Schema Arbitraggio Centralizzato (Daisy Chain)

Come possiamo vedere, abbiamo 2 linee per il controllo del BUS:

- Una da destra verso sinistra usata dalle unità funzionali per richiedere il ruolo di MASTER;
- Una da sinistra a destra che emette l'arbitro per assegnare il ruolo alla prima unità funzionale che incontra che ha fatto richiesta.

Nota Bene

Questo meccanismo avvantaggia le unità funzionali fisicamente più vicine all'arbitro; in questo modo introduce implicitamente una logica di priorità.

Funzionamento Daisy Chain:

1. Ingresso grant asserito: lo propago agli altri componenti, a meno che abbia intenzione di chiedere il controllo del BUS.
2. Solo se ricevo un grant asserito sono in grado di effettuare una richiesta.
3. Il segnale busy ci dice se il BUS è disponibile (questo deve essere non asserito per permettere una richiesta).

Arbitro centralizzato con richieste multiple

È possibile rendere più flessibile questo meccanismo aumentando il numero di linee request (richiesta del bus) e grant (conferma da parte dell'arbitro), assegnando una scala di priorità alle varie linee.

Se abbiamo più linee che chiedono contemporaneamente il BUS, l'arbitro darà il grant alla linea con priorità più alta che ha effettuato la richiesta. (Ha senso se linee di richiesta diverse sono collegate a dispositivi diversi).

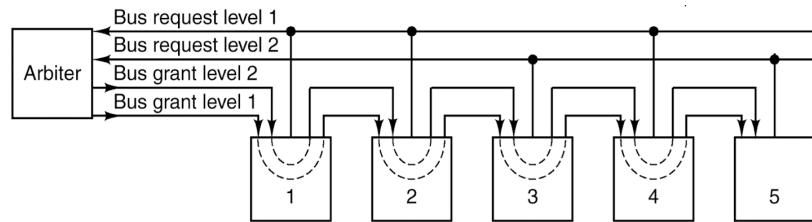


Figura 3.4: Arbitraggio centralizzato con livelli di priorità

BUS Acknowledge (Handshaking)

È possibile migliorare ulteriormente questo meccanismo introducendo il concetto di **accettazione**.

Flusso operativo:

1. Un'unità chiede all'arbitro il controllo del BUS.
2. L'arbitro concede il permesso: l'unità attiva la linea **BUS acknowledge** e disattiva la sua richiesta.
3. L'arbitro, vedendo l'acknowledge, disattiva la linea di conferma.
4. A questo punto l'unità ha il controllo del BUS; le linee di richiesta e conferma sono libere, quindi un'altra unità può fare richiesta (che rimane in attesa).
5. Quando la prima unità rilascia il BUS disattivando l'acknowledge, l'arbitro può confermare la richiesta alla prossima unità.

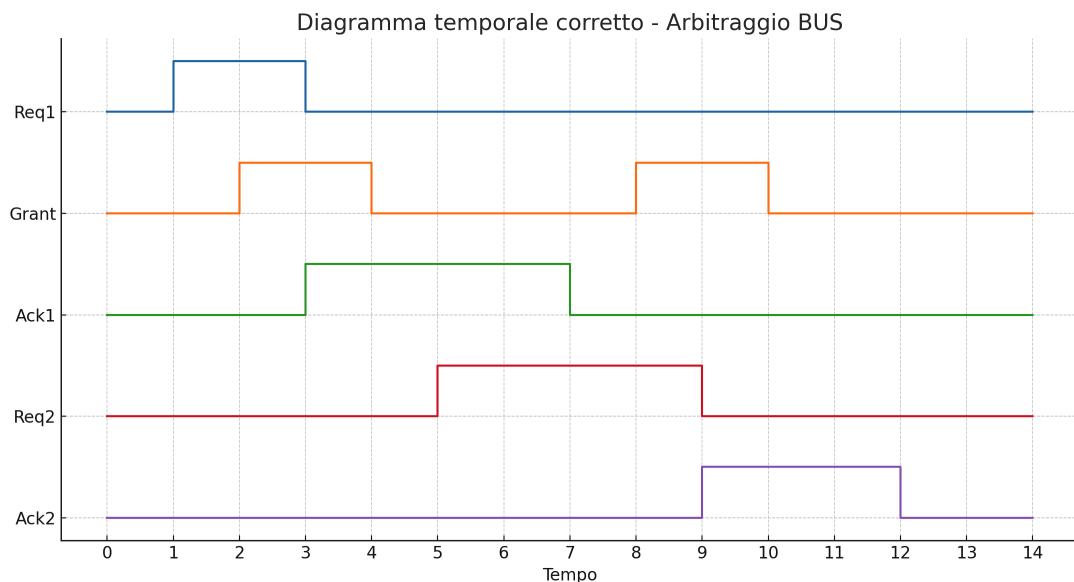


Figura 3.5: Diagramma temporale del BUS Acknowledge

3.2.2 Arbitraggio Distribuito

Anche qui abbiamo delle regole che scandiscono la logica di questo meccanismo:

1. Non abbiamo un arbitro che regola le richieste, ma ogni unità ha la sua linea per emettere la richiesta.

2. Le unità hanno diverse priorità fissate a priori.
3. Ogni unità può attivare **solo la propria linea di richiesta** del BUS.

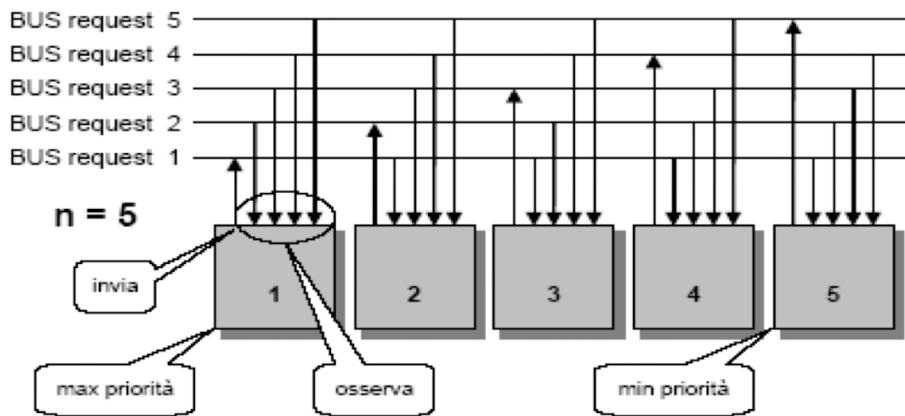


Figura 3.6: Schema Arbitraggio Distribuito

Come funziona: Tutte le unità possono *parlare* alle altre mediante la propria linea e allo stesso tempo possono *ascoltare* le altre unità. Ogni unità, per diventare master, si assicura che unità di priorità superiore alla propria non abbiano fatto richiesta.

Nota Bene

È lampante il problema principale di questo meccanismo: il numero di linee di controllo cresce con il numero di unità funzionali (non scalabile).

Arbitraggio distribuito a 3 Linee

Per migliorare l'arbitraggio distribuito è possibile implementarlo usando solo 3 Linee:

- **Impegno del BUS (BUS busy):** questa linea viene mantenuta attiva dall'unità che detiene il controllo del BUS.
- **Conferma del BUS (BUS grant):** è la linea di conferma, collegata da un dispositivo all'altro in *daisy chain*.
- **Richiesta del BUS (BUS request):** l'unità attiva questa linea per richiedere il BUS.

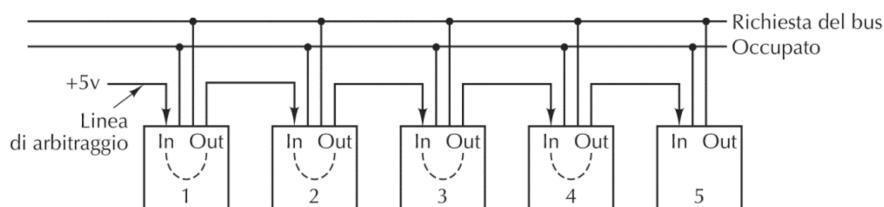


Figura 3.7: Arbitraggio distribuito ottimizzato a 3 linee

3.3 Temporizzazione del BUS

Le attività di un calcolatore si scandiscono in **cicli di BUS**, un’operazione per ciclo. I MASTER governano le operazioni; ricordiamo che gli SLAVE non possono dare inizio a un’operazione autonomamente.

Abbiamo sostanzialmente 2 tipi di operazioni:

- **Lettura:** un dato viene trasferito da uno SLAVE al MASTER.
- **Scrittura:** un dato viene trasferito dal MASTER a uno SLAVE.

Nota Bene

Sottolineo che la direzione del dato è sempre decisa dal MASTER.

Il susseguirsi degli eventi sopra descritti va coordinato e per farlo abbiamo 2 metodi:

- **Temporizzazione Sincrona;**
- **Temporizzazione Asincrona.**

3.3.1 Temporizzazione Sincrona

Questa modalità è la più semplice e anche la più usata; a discapito di ciò è però meno flessibile.

Vediamo cosa la caratterizza:

- Eventi determinati da un clock. BUS cycle corrispondono a Clock Cycle.
- Tutti gli eventi sono sincronizzati sui fronti del clock.
- Molti eventi durano più cicli di clock.

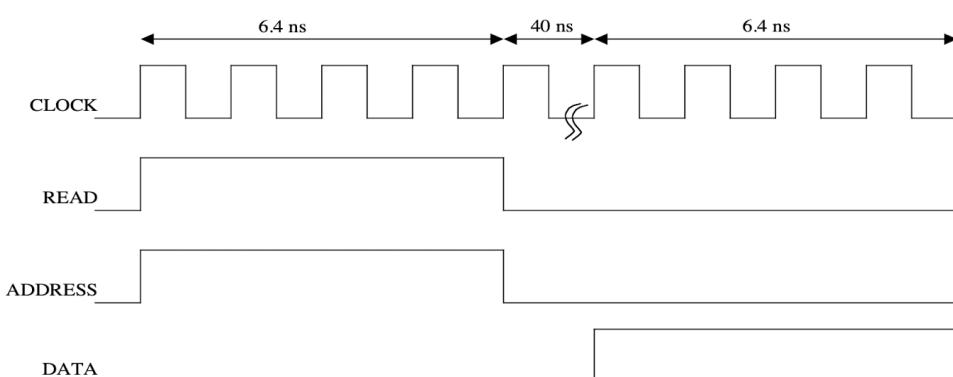


Figura 3.8: Temporizzazione Sincrona: eventi guidati dal clock

3.3.2 Temporizzazione Asincrona

Questa modalità consente a dispositivi di diversa velocità di condividere efficacemente il BUS, è quindi più flessibile. Un evento dipende dagli eventi precedenti (Handshaking).

Vediamo da cosa è caratterizzata:

- Non esiste un segnale di clock comune tra tutti.
- Le transizioni si sincronizzano in base agli eventi.
- Il BUS include dei segnali di controllo che insieme formano il **protocollo di sincronizzazione del BUS**.
- Lo stato delle operazioni avanza in corrispondenza degli eventi.

Lettura asincrona di una Parola

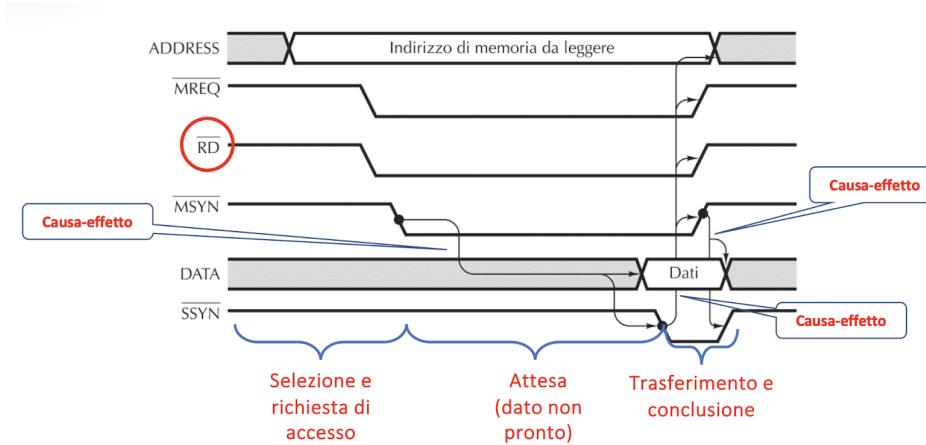


Figura 3.9: Protocollo di lettura asincrona

1. Il **Master invia l'indirizzo** sul BUS indirizzi, attiva **MREQ**, **RD** e, dopo averli stabiliti, attiva **MSYN**.
2. Lo **Slave (memoria) esegue l'operazione** nel minor tempo possibile e fornisce la parola sul BUS dati; poi attiva **SSYN**.
3. Il **Master legge la parola** dal BUS dati, toglie l'indirizzo e disattiva **MREQ** e **RD**; poi disattiva anche **MSYN**.
4. Infine, lo Slave disattiva **SSYN**.

Nota Bene

Tutte le operazioni sono scandite dall'attivazione e la disattivazione di **MSYN** e **SSYN**.

3.3.3 Lettura Sincrona con ritardi (Wait States)

Questo meccanismo serve a sincronizzare un Master veloce (come la CPU) con uno Slave più lento (come la memoria). Dato che la memoria è troppo lenta per fornire i dati in un ciclo di clock standard, il sistema introduce un "Ciclo di stato di attesa" (Wait State), in questo caso il ciclo T2.

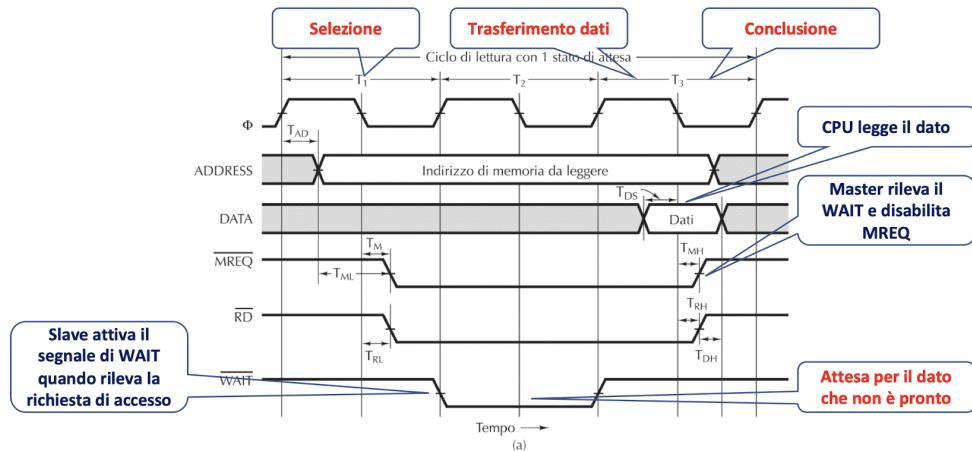


Figura 3.10: Lettura sincrona con inserimento di Wait State

Analisi dei 3 cicli

Ciclo T1 (Selezione):

- Il Master (CPU) avvia l'operazione.
- Mette l'indirizzo della memoria che vuole leggere sul bus ADDRESS.
- Dopo che l'indirizzo si è stabilitizzato, sul *fronte di discesa* del clock (metà ciclo T1), il Master attiva (abbassa) i segnali MREQ (richiesta di memoria) e RD (richiesta di lettura).

Ciclo T2 (Attesa, Wait):

- Lo Slave (memoria) ha rilevato la richiesta ($\overline{\text{MREQ}}$ e $\overline{\text{RD}}$ bassi) ma sa di non essere abbastanza veloce per fornire il dato (richiede 40 ns, più del tempo disponibile).
- Per guadagnare tempo, sul *fronte di salita* del clock (inizio ciclo T2), lo Slave attiva (abbassa) il segnale WAIT.
- Il Master rileva $\overline{\text{WAIT}}$ attivo e capisce che deve attendere. Invece di leggere i dati alla fine di T2, estende l'operazione di un altro ciclo.

Ciclo T3 (Trasferimento dati e Conclusione):

- All'inizio di T3, lo Slave ha finalmente il dato pronto.
- Sul *fronte di salita* del clock (inizio ciclo T3), lo Slave disattiva (alza) il segnale WAIT (per dire "OK, sono pronto") e contemporaneamente mette il dato richiesto sul bus DATA.
- Sul *fronte di discesa* del clock (metà ciclo T3), il Master legge il dato presente sul bus DATA.
- Subito dopo aver letto, il Master conclude l'operazione: disattiva (alza) $\overline{\text{MREQ}}$ e $\overline{\text{RD}}$ e rimuove l'indirizzo dal bus.

Tipi di ritardi possibili

3.3.4 Lettura Sincrona a Blocchi

Più parole vengono fornite in parallelo; nello specifico siamo in grado di trasferire lungo il BUS una parola per ciclo. Nella lettura tradizionale necessitiamo sempre di 3 cicli per parola.

Simbolo	Parametro	Min	Max	Unità
T_{AD}	Ritardo dell'output dell'indirizzo		4	nsec
T_{ML}	Indirizzo stabile prima di \overline{MREQ}		2	nsec
T_M	Ritardo di \overline{MREQ} rispetto al fronte di discesa di Φ in T_1		3	nsec
T_{RL}	Ritardo di RD rispetto al fronte di discesa di Φ in T_1		3	nsec
T_{DS}	Tempo di impostaz. dei dati prima del fronte di discesa di Φ	2		nsec
T_{MH}	Ritardo di \overline{MREQ} rispetto al fronte di discesa di Φ in T_3		3	nsec
T_{RH}	Ritardo di RD rispetto al fronte di discesa di Φ in T_3		3	nsec
T_{DH}	Tempo di mantenimento dei dati dopo la negazione di RD	0		nsec

Figura 3.11: Esempi di ritardi nel BUS

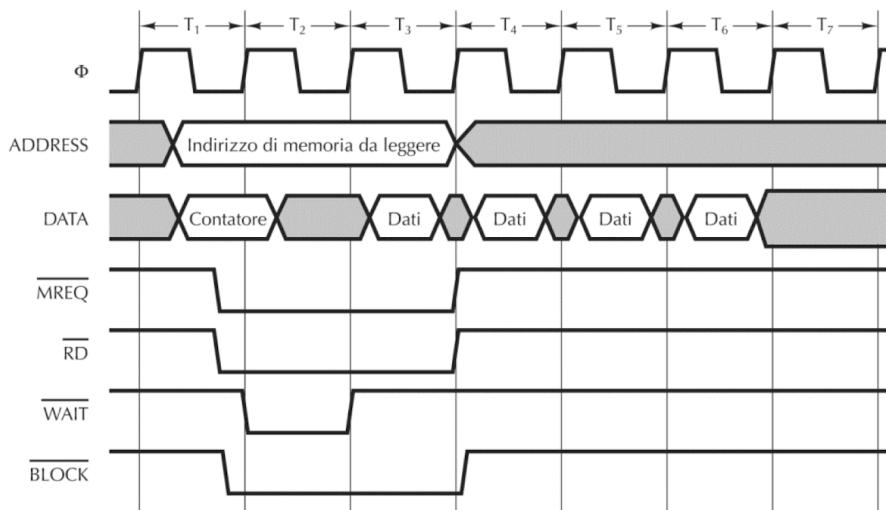


Figura 3.12: Confronto lettura tradizionale vs lettura a blocchi

Grazie a questo paradigma necessitiamo dei primi 2 cicli preparatori e successivamente un ciclo per parola.

Nota Bene

Questo metodo è molto vantaggioso al crescere del numero di parole.

Esempio trasferimento di 4 parole:

- **Lettura a blocchi:** 6 cicli = 2 (prep) + 4 (dati)
- **Lettura tradizionale:** 12 cicli = 3×4

Riepilogo e Conclusioni

Il **BUS sincrono** è più semplice da implementare e da gestire. Introduce *sprechi di tempo* (operazioni che richiederebbero meno di un ciclo o un numero frazionato di essi).

Il **BUS asincrono** è più efficiente ma ha in generale una complessità più alta.

Si usano principalmente bus con **temporizzazione sincrona**, nonostante sia più lenta. Questo perché la maggior parte dei dispositivi (CPU, memorie, etc.) sono nativamente sincroni; ciò significa che per permettere l'utilizzo con bus asincroni servirebbe una rete sequenziale *ad hoc* usata come interfaccia tra il bus e il dispositivo, aumentando particolarmente la complessità del sistema (in particolare la progettazione di esso).

Capitolo 4

Algebra e logica Booleana

4.1 Sistemi Numerici

Definizione 4.1: Sistema Numerico

Un sistema numerico è definito da:

- Un insieme di simboli (cifre);
- Un insieme di regole per costruire i numeri.

I sistemi possono essere **posizionali** o **non posizionali** (come il sistema romano).

4.1.1 Formula Generale dei Sistemi Posizionali

Il valore V di un numero può essere espresso come:

$$V(n) = \sum_{i=-m}^{n-1} d_i \times r^i$$

Dove:

- d_i è il simbolo in posizione i -esima;
- r è la base del sistema;
- m = numero di cifre della parte frazionaria;
- n = numero di cifre della parte intera;
- $0 \leq d_i < r$.

4.1.2 Principali Sistemi Numerici

- **Sistema binario:** $d_i \in \{0, 1\}$, $r = 2$.
- **Sistema ottale:** $d_i \in \{0, \dots, 7\}$, $r = 8$.
- **Sistema esadecimale:** $d_i \in \{0, \dots, 9, A, B, C, D, E, F\}$, $r = 16$.

4.2 Alfabeto Binario

- **BIT (binary digit)**: una cifra binaria (0 o 1).
- **BYTE**: insieme di 8 bit.
- **WORD (parola)**: insieme di più byte (la dimensione dipende dall'architettura).
- Un numero binario è una **sequenza di bit**.

4.3 Rappresentazione di Numeri con Segno

Per rappresentare interi relativi con n bit, si dimezza l'intervallo dei valori assoluti. Esistono diverse rappresentazioni.

4.3.1 Modulo e Segno

- 1 bit per il segno (0: positivo, 1: negativo).
- $n - 1$ bit per il modulo.
- **Intervallo**: $[-(2^{n-1} - 1), + (2^{n-1} - 1)]$.
- **Svantaggio**: doppia rappresentazione dello zero (+0 e -0).

4.3.2 Complemento a 1

- **Positivi**: rappresentazione posizionale normale (iniziano per 0).
- **Negativi**: complemento bit a bit del corrispondente positivo (iniziano per 1).
- **Intervallo**: $[-(2^{n-1} - 1), + (2^{n-1} - 1)]$.
- **Svantaggio**: doppia rappresentazione dello zero.

4.3.3 Complemento a 2 (CP2)

- **Positivi**: stessa rappresentazione del complemento a 1.
- **Negativi**: complemento a 1 + 1.
- **Metodo pratico**: partendo da destra, lasciare invariati tutti i bit fino al primo "1" compreso, poi complementare i restanti a sinistra.
- **Intervallo**: $[-2^{n-1}, +2^{n-1} - 1]$.
- **Vantaggi**: intervallo più esteso verso i negativi, una sola rappresentazione dello zero.

4.3.4 Eccesso 2^{n-1} (Bias)

- I numeri sono rappresentati come somma tra il numero dato e un Bias 2^{n-1} .
- Positivi iniziano per 1, negativi per 0.
- Si ottiene da CP2 complementando il bit più significativo.
- **Intervallo**: $[-2^{n-1}, +2^{n-1} - 1]$.

Metodo	Zero unico?	Range Simmetrico?	Minimo
Modulo e Segno	No	Sì	$-(2^{n-1} - 1)$
Complemento a 1	No	Sì	$-(2^{n-1} - 1)$
Complemento a 2	Sì	No (più negativi)	-2^{n-1}
Eccesso K	Sì	No	-2^{n-1}

Tabella 4.1: Confronto rappresentazioni intere

4.4 Rappresentazione in Virgola Mobile (IEEE 754)

4.4.1 Standard IEEE 754 (1985)

Singola Precisione (32 bit)

S (1 bit)	W - Esponente (8 bit)	Y - Mantissa (23 bit)
Segno	Eccesso 127	Parte Frazionaria

Doppia Precisione (64 bit)

- 1 bit segno + 11 bit esponente + 52 bit mantissa.

4.4.2 Numeri Normalizzati

- Esponente rappresentato in **eccesso 127** (singola precisione).
- Range esponente: $-126 \leq e \leq 127$.
- Mantissa: solo parte frazionaria (bit 1 prima della virgola è implicito: 1.YYYY...).
- Condizione: $1 \leq m < 2$.
- Intervallo valori: $[2^{-126}, \sim 2^{128}]$.

4.4.3 Casi Speciali

- **Zero:** Mantissa = 0, Esponente = 0.
- **Overflow (Infinito):** Mantissa = 0, Esponente = tutti 1 (255).
- **NaN (Not a Number):** Mantissa $\neq 0$, Esponente = tutti 1.
- **Numero denormalizzato:** Mantissa $\neq 0$, Esponente = 0.

Nota Bene

Servono a gestire situazioni di **underflow** graduale. In questo caso l'esponente è 0 e il bit implicito della mantissa diventa 0 (0.YYYY...). Intervallo: $[2^{-149}, \sim 2^{-126}]$.

4.4.4 Operazioni ed Errori

Per la **moltiplicazione**:

1. Moltiplicare le mantisse;
2. Sommare algebricamente gli esponenti;
3. Normalizzare la mantissa se necessario;
4. Riaggiustare l'esponente.

Errori di Rappresentazione

- **Errore assoluto:** $e_A = n - n'$
- **Errore relativo:** $e_R = \frac{e_A}{n} = \frac{n-n'}{n}$

Con mantissa normalizzata, l'errore relativo massimo è costante. La **densità** dei numeri rappresentabili non è uniforme (è maggiore vicino all'origine).

4.5 Algebra Booleana

Definizione 4.2: Algebra Booleana

Sistema algebrico definito da:

1. Variabili binarie (True/False, 1/0);
2. Operatori logici;
3. Tabelle di verità;
4. Relazione di equivalenza “=” tra espressioni.

4.5.1 Operatori Logici

Fondamentali:

- **AND** ($P \cdot Q$): vale 1 sse entrambi gli operandi sono 1.
- **OR** ($P + Q$): vale 1 se almeno un operando è 1.
- **NOT** (\bar{P} o P'): inverte il valore.

Aggiuntivi:

- **XOR** ($P \oplus Q$): vale 1 quando gli operandi sono diversi.
- **NAND** ($\overline{P \cdot Q} = \bar{P} + \bar{Q}$): complemento dell'AND.
- **NOR** ($\overline{P + Q} = \bar{P} \cdot \bar{Q}$): complemento dell'OR.

4.5.2 Forme Canoniche

- **Somma di Prodotti (SoP):**
 - Configurazioni che producono uscita = 1 sono chiamate **mintermini**.
 - Esempio: $F = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + A\bar{B}\bar{C}$.
- **Prodotto di Somme (PoS):**
 - Configurazioni che producono uscita = 0 sono chiamate **maxtermini**.
 - Esempio: $F = (A + B) \cdot (A + \bar{B}) \cdot (\bar{A} + B)$.

Si passa da una forma all'altra applicando il **teorema di De Morgan**:

$$\overline{A \cdot B} = \bar{A} + \bar{B} \quad \text{e} \quad \overline{A + B} = \bar{A} \cdot \bar{B}$$

4.6 Porte Logiche e Reti Combinatorie

4.6.1 Porte Logiche

Circuiti elettronici (realizzati tramite transistor) che implementano operatori booleani. È tecnologicamente più facile realizzare porte NAND e NOR rispetto a AND e OR.

4.6.2 Reti Logiche

Caratterizzate da n variabili di ingresso e m variabili di uscita.

- **Reti Combinatorie:** Senza memoria. L'uscita dipende **solo** dagli ingressi attuali.
- **Reti Sequenziali:** Con memoria. L'uscita dipende dagli ingressi e dallo stato precedente.

4.6.3 Analisi e Sintesi di Reti Combinatorie

Livelli di una Rete: Porte i cui ingressi ricevono segnali nello stesso istante.

- Ritardo totale = $L \times \Delta t$ (dove L è il numero di livelli).
- Obiettivo: minimizzare i livelli per ridurre il ritardo.

Sintesi Minima: Dai requisiti si implementa la funzione con il minimo numero di porte.

- Obiettivo: ridurre spazio su chip e costi.

Metodi di Sintesi Minima

1. **Semplificazione algebrica:** uso delle relazioni notevoli.
2. **Mappe di Karnaugh:** rappresentazione grafica su mappa di 2^n celle.
3. **Metodo di Quine-McKluskey:** confronto sistematico dei termini (algoritmico).

Capitolo 5

Livello Logico Digitale

5.1 Architettura a Livelli dei Calcolatori

Per semplificare la progettazione e rendere più facile la manutenzione scegliamo di costruire i calcolatori in maniera modulare, *a strati*. In questo modo abbiamo diversi *livelli*: dal **Livello 0** (Hardware) al **Livello n** (interfaccia con cui interagisce l'utente).

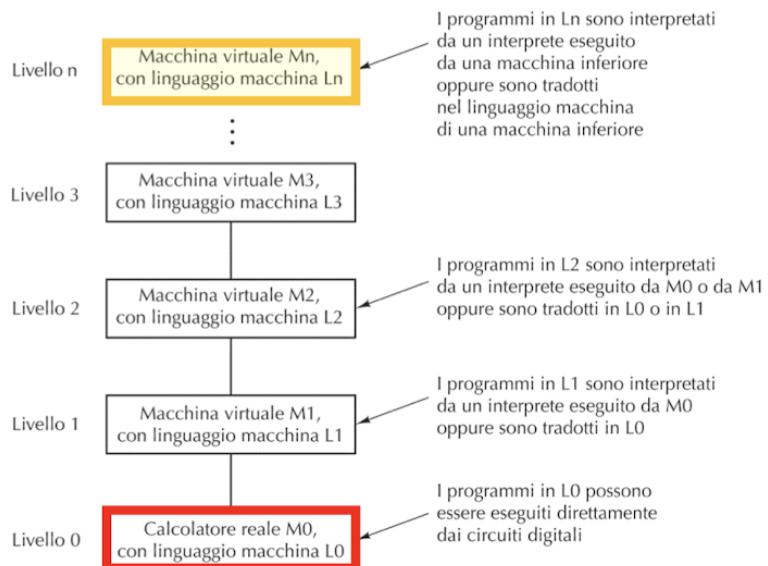


Figura 5.1: Gerarchia dei livelli di un calcolatore

- Più siamo in basso di Livello, più è facile la realizzazione Hardware e più è complesso programmare.
- Più siamo in alto di Livello, più è facile programmare ma da un certo punto in poi diventa impossibile la realizzazione Hardware.
- Il **Livello 2** è il più basso al quale un utente può arrivare.
- Tipicamente si programma al **Livello 5**.

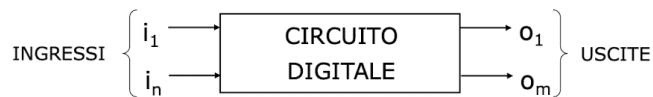


Figura 5.2: Rappresentazione di segnali digitali

5.1.1 Livello 0: Circuiti Logici Digitali

In questo livello troviamo Circuiti elettronici i cui ingressi e uscite assumono solo 2 valori (0-1).

Unendo tanti di questi blocchi generiamo **Reti logiche combinatorie**.

5.2 Reti Logiche e Reti Combinatorie

Una **rete logica** è un insieme di blocchi funzionali realizzati mediante porte logiche ed elementi di memoria.

Abbiamo 2 tipi di reti Logiche:

- Con memoria: **Reti Sequenziali**.
- Senza memoria: **Reti Logiche Combinatorie**.

Definizione 5.1: Rete Combinatoria

È caratterizzata dal fatto che in un qualsiasi istante i valori in uscita della rete combinatoria dipendono solo ed esclusivamente dai valori in ingresso (mai da valori precedenti).

D'altra parte, una **Rete Sequenziale** può assumere uno *stato* ed esso influenza i valori in uscita della rete. Significa che, a parità di ingressi, se la rete si trova allo stato S_1 o S_2 avremo in uscita valori diversi.

5.3 Componenti Elettronici

I circuiti integrati sono chip di silicio che racchiudono al loro interno moltissime porte logiche. A seconda della complessità si parla di SSI, MSI, LSI o VLSI. Oggi i processori moderni sono esempi di VLSI.

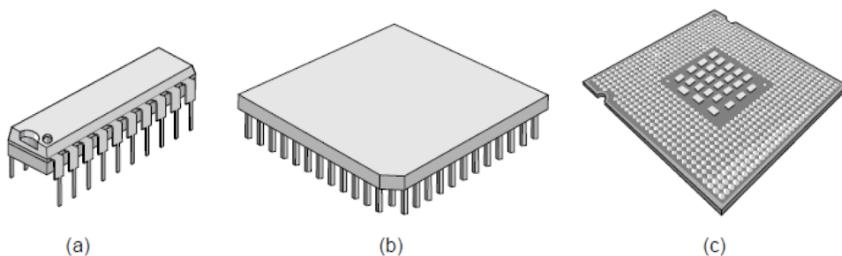


Figura 5.3: Esempi di package e wafer di silicio

Esistono diversi livelli di integrazione:

- **SSI** (Small Scale): poche porte (1 – 10).
- **MSI** (Medium Scale): decine di porte (10 – 100).
- **LSI** (Large Scale): centinaia/migliaia (10^2 – 10^5).
- **VLSI** (Very Large Scale): centinaia di migliaia o milioni ($> 10^5$) → processori moderni.

Questi componenti hanno tempi di commutazione molto rapidi (nanosecondi) e si presentano in diversi **contenitori fisici**:

1. **DIP** (Dual Inline Package): i classici chip rettangolari a piedini.
2. **PGA** (Pin Grid Array): piedini sul fondo.
3. **LGA** (Land Grid Array): contatti piatti usati ad esempio nelle CPU moderne.

5.3.1 Comparatore

Il comparatore è un circuito logico che confronta due stringhe di bit sfruttando l'operatore XOR. Restituisce un'uscita che segnala se i due valori sono uguali o diversi.

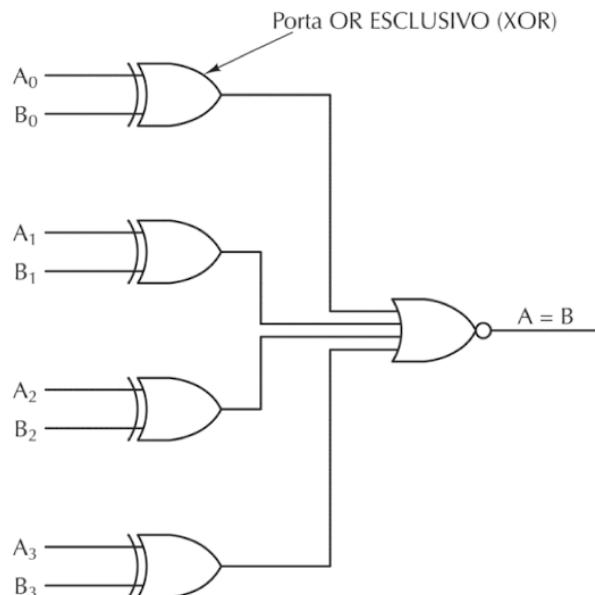


Figura 5.4: Schema logico di un comparatore

Si basa sull'operatore **XOR** (OR Esclusivo):

- Se due bit sono uguali → l'uscita è 0.
- Se due bit sono diversi → l'uscita è 1.

In pratica, confronta bit a bit due numeri binari. L'uscita indica se i due valori sono **uguali** o **diversi**.

5.3.2 Multiplexer (MUX)

Il Multiplexer è un componente logico che ci permette di scegliere quale segnale ascoltare in base ai bit di controllo (selettori).

Facciamo un esempio per capire meglio:

1. Immaginiamo di avere due segnali (A, B) e di voler scegliere se ascoltare uno o l'altro.
2. Tramite il MUX ci basterà un bit di controllo (S_0) per scegliere quale segnale ascoltare.
3. Ad esempio $S_0 = 0$ per ascoltare A e $S_0 = 1$ per ascoltare B .

S	A	B	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

	AB	00	01	11	10
S	0			1	1
	1		1	1	

$$Y = \bar{S}A + SB$$

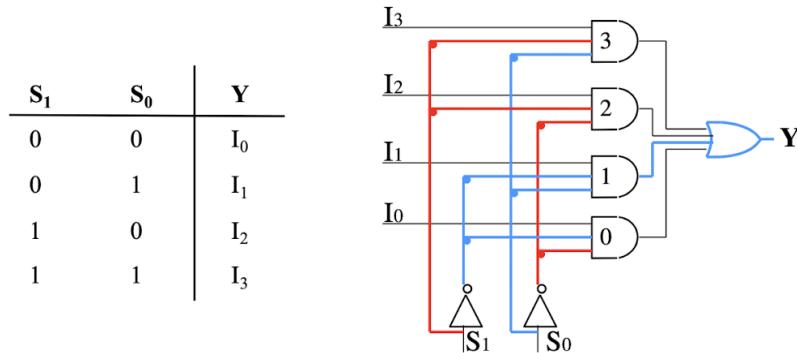
Figura 5.5: Multiplexer a 2 ingressi

Per realizzare un MUX a 1 bit (di controllo) ci basterà:

1. Scrivere la tabella di verità.
2. Trovare la formula minima tramite le **Leggi di Karnaugh**.
3. Disegnare il circuito della equazione ricavata.

Multiplexer a 2 bit (4-to-1)

Dati due bit di controllo S_1 ed S_0 , per selezionare l'uscita Y è sufficiente effettuare l'**AND** di ogni ingresso con la configurazione di selezione corrispondente e porre tutte le uscite in **OR**.



$$Y = \bar{S}_0 \bar{S}_1 I_0 + S_0 \bar{S}_1 I_1 + \bar{S}_0 S_1 I_2 + S_0 S_1 I_3$$

Figura 5.6: Struttura interna di un MUX 4-a-1

Utilizzo del Multiplexer

Un **multiplexer (MUX)** non serve solo a selezionare quale segnale mandare in uscita tra più sorgenti, ma può anche essere utilizzato per condividere una stessa linea di trasmissione o scegliere un comando/controllo in base a un codice.

Inoltre, grazie alla sua struttura, con un MUX a n bit è possibile implementare qualsiasi funzione booleana di n variabili, semplicemente cablando i suoi ingressi a 0 o 1 a seconda della presenza dei relativi **mintermini** nella forma canonica della funzione.

5.3.3 Decoder

Possiamo vederlo un po' come l'inverso del Multiplexer, dato che è caratterizzato da N ingressi e 2^N uscite. Ciò significa che se avremo 2 ingressi troveremo 4 uscite (2^2), se invece abbiamo 3 ingressi troveremo 8 uscite (2^3).

Perché ho detto che può essere visto come l'inverso del multiplexer? Perché in uscita troveremo alto il segnale corrispondente al valore in binario degli ingressi.

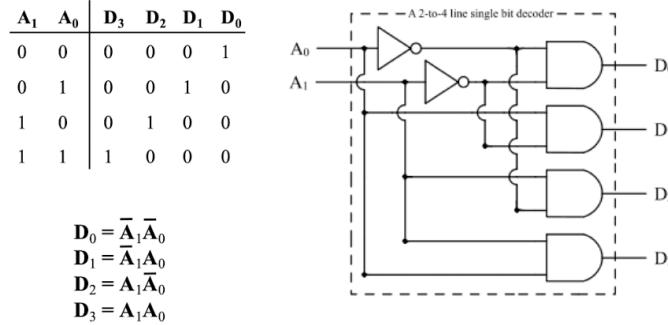


Figura 5.7: Decoder 2-a-4

Nell'esempio sopra, i segnali A sono i 2 ingressi *selettori* mentre i segnali D sono le uscite.

5.4 Reti Logiche Sequenziali

5.4.1 Definizione e Caratteristiche

Le reti sequenziali, proprio come le **Reti logiche combinatorie**, sono caratterizzate da **N ingressi e M uscite**. A differenza di esse, però, i valori dei segnali in uscita dipendono dagli ingressi assunti precedentemente.

Per far ciò hanno ovviamente bisogno di qualche **Elemento di memoria**, e dato che la memoria è limitata possono "ricordare" solo un numero finito di combinazioni di ingressi precedenti.

Esempio di retroazione (Latch): Questa rete sequenziale realizza la sua memoria tramite retroazione; così è implementato l'elemento di memoria più semplice, il **latch**.

Caratteristiche Fondamentali

Una rete sequenziale è definita da:

1. Un **insieme degli stati di ingresso**: l'insieme di configurazioni delle variabili booleane d'ingresso.
2. Un **insieme degli stati interni**: ogni stato corrisponde ad una possibile configurazione passata della rete.
3. Un **insieme degli stati di uscita**: l'insieme di configurazioni delle variabili booleane di uscita.
4. Un **diagramma degli stati**: descrive, in funzione delle variabili di ingresso e degli stati interni, ad un dato istante, lo stato all'istante successivo e le relative configurazioni degli stati di uscita.

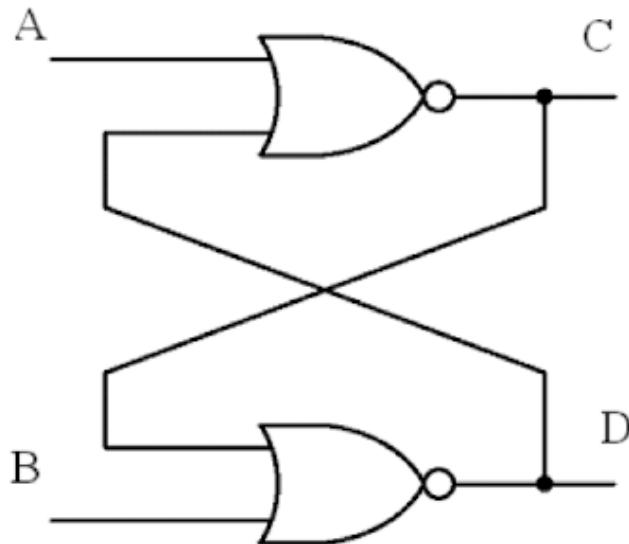


Figura 5.8: Elemento di memoria semplice tramite retroazione

Nota Bene

Questa definizione coincide con quella di **automa a stati finiti**.

5.4.2 Diagramma degli stati: Mealy vs Moore

Entrambi i diagrammi sono grafi orientati, dove i nodi rappresentano lo stato della rete e gli archi la possibilità di passare da uno stato all'altro in funzione degli ingressi.

La differenza sostanziale è:

- **Moore:** i valori in uscita della rete sono assegnati in base allo **stato**.
- **Mealy:** i valori in uscita dipendono dalla combinazione di **ingressi e stato**.

Esempio Pratico: Ho implementato una semplice rete logica che conta da 0 a 2 finché in ingresso ha 1 e ricomincia; se invece in ingresso ha 0 torna allo step precedente del conteggio (quando è a 0 rimane a zero fin tanto che in ingresso ha valore 0).

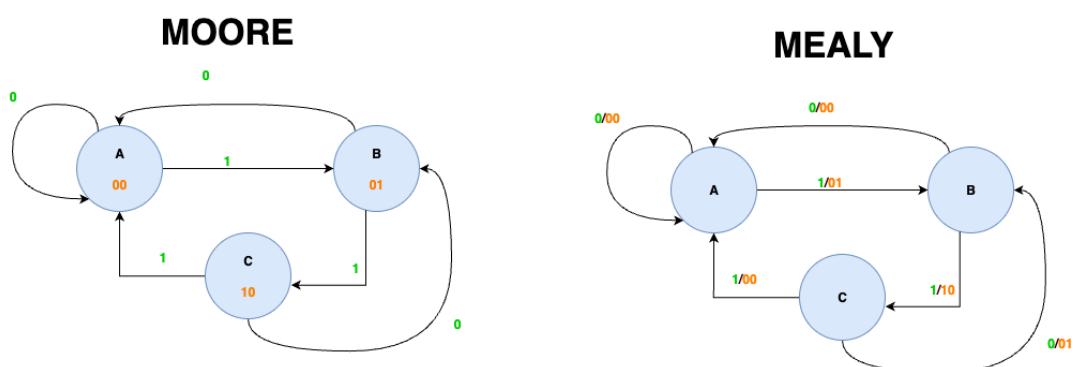


Figura 5.9: Confronto Diagrammi Mealy e Moore (Verde: Ingressi, Arancio: Uscite, Celeste: Stati)

Differenze Implementative

- **Mealy:** Ha sempre 2 reti combinatorie, una per il calcolo del prossimo stato e una per le uscite.
- **Moore:** Ha sempre una rete combinatoria per il calcolo del prossimo stato.
 - Dai **Flip Flop** osservo lo stato presente, e da quei bit posso ricavare le uscite *anche con un semplice decoder*.

5.4.3 Tabella di flusso e Tabella delle transizioni

- La **Tabella di flusso** descrive le transizioni della rete sequenziale in forma tabellare: Ingresso, Stato Presente → Stato Successivo, Uscita.
 - È l'equivalente della rappresentazione con **diagramma degli stati**.
- La **Tabella delle transizioni** sostituisce i nomi simbolici degli stati (S_0, S_1, \dots) con le **configurazioni binarie** che li rappresentano.
 - Serve per l'implementazione pratica (per esempio nei circuiti con latch/flip-flop).

Esempio Tabellare

Tabella 5.1: Tabella di Flusso

Stato Pres.	X=0	X=1
S_0	$S_0 / 00$	$S_1 / 10$
S_1	$S_1 / 10$	$S_2 / 01$
S_2	$S_2 / 01$	$S_3 / 11$
S_3	$S_3 / 11$	$S_0 / 00$

Tabella 5.2: Tabella Transizioni (Binario)

Stato Pres.	X=0	X=1
00	00 / 00	01 / 10
01	01 / 10	10 / 01
10	10 / 01	11 / 11
11	11 / 11	00 / 00

5.4.4 Architettura di una Rete Logica Sequenziale

In generale, una rete sequenziale è composta da una parte combinatoria e una parte di memoria (Flip-Flop).

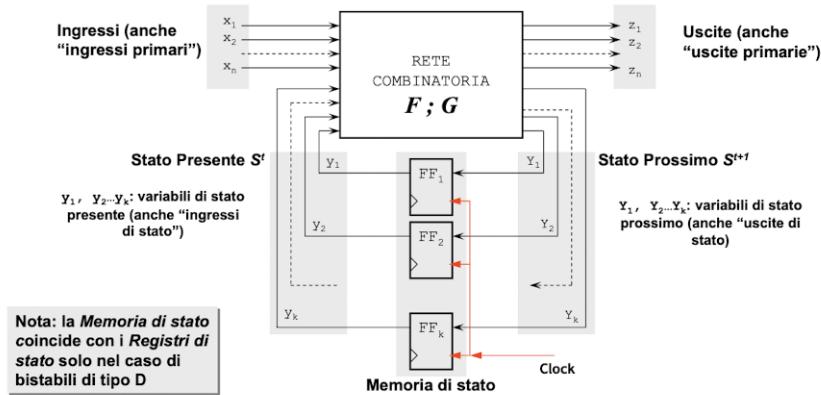


Figura 5.10: Schema a blocchi di una rete sequenziale

In caso di una macchina sincrona avrà anche un segnale di clock (tipicamente indicato con CLK) che *temporizza* le uscite e i cambi di stato.

Definizione 5.2: Temporizzazione

Con il termine **temporizza** si intende che uscite e stati cambiano il valore allo "scoccare" del colpo di clock (fronte di salita o discesa).

5.5 Elementi di Memoria

Gli elementi di memoria servono a memorizzare gli stati passati della rete. Più elementi di memoria abbiamo (a 1 bit), più stati è possibile memorizzare combinandoli.

Abbiamo 2 tipi di elementi di memoria:

1. Elementi di memoria di tipo **Sincroni** o **Asincroni** (i **Latch**).
2. Elementi di memoria di tipo **Master-Slave** (i **Flip-Flop**).

5.5.1 Latch

Il principio base che accomuna tutti i tipi di Latch è quello della **retroazione**, ossia *usare un precedente risultato come operando per calcolare il prossimo valore in uscita*. Adesso vediamo diversi tipi di Latch in ordine di complessità.

Latch Set-Reset (SR)

È un latch asincrono (privo di clock) ed è il latch più semplice che possiamo implementare.

Il modo più facile per capire la logica secondo me è:

1. Se $R = 0$ abbiamo S in uscita.
2. Se $R = 1$ abbiamo sempre 0 in uscita.

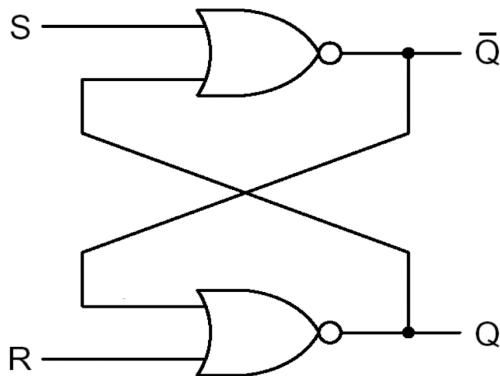


Figura 5.11: Circuito Latch SR

S	R	$Q(t)$	$Q(t+\tau)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	-
1	1	1	-

Figura 5.12: Tabella di verità SR

3. Se $R = S = 0$ abbiamo $Q(t)$, che sarebbe l'uscita all'istante precedente (la tabella di verità classica non mostra il tempo, ma qui è fondamentale capire lo stato "hold").
4. Non posso mai avere $S = R = 1$ (Stato proibito).

Latch SR Sincrono

Vediamo adesso cosa succede se proviamo a rendere sincrono il circuito appena visto, collegando quindi un clock alla logica.

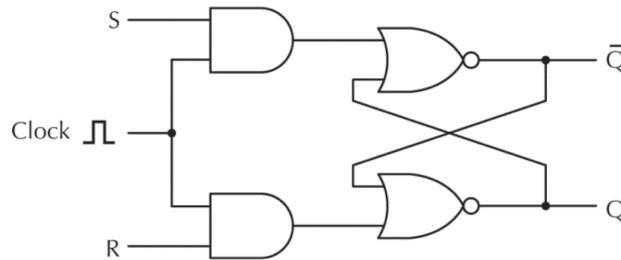


Figura 5.13: Latch SR Sincrono (con Clock)

La logica rimane identica; l'unica differenza è che le uscite cambiano solo in corrispondenza di fronti di salita (o di discesa) del clock.

Implementazione Pratica Per implementare un Latch di tipo SR necessito di:

- 2 Transistor.
- 2 Interruttori (non possono mai essere entrambi aperti).

Latch JK

Per quanto riguarda la logica, è identico a un latch SR con la differenza che permette in entrata due valori uguali a 1. In questo caso particolare (con i due ingressi uguali a 1) avremo in uscita il risultato dell'istante precedente **invertito** (Toggle).

Vediamo che per realizzare il circuito ci basta collegare alle 2 AND iniziali l'uscita del Latch (retroazione incrociata).

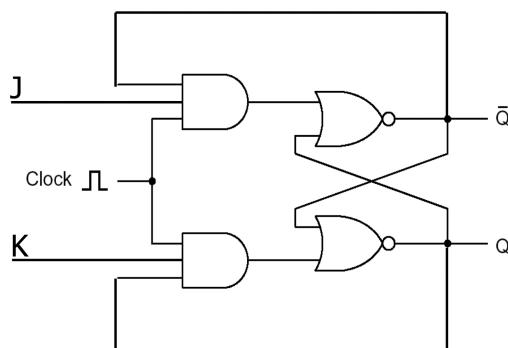


Figura 5.14: Circuito Latch JK

J	K	Q(t)	Q(t+τ)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Inversione dell'uscita

Figura 5.15: Tabella di verità JK

Altri tipi di Latch

Abbiamo ovviamente altri tipi di Latch (come il tipo D e il tipo T). Ecco uno schema riassuntivo:

- **Latch D (Delay)**
 - ✓ latch JK per cui $\mathbf{J} = \mathbf{K} = \mathbf{D}$
 - ✓ l'effetto è di ritardare il valore dell'ingresso \mathbf{D} sull'uscita \mathbf{Q} di un ciclo di clock

D	Q(t)	Q(t+τ)
0	0	0
0	1	0
1	0	1
1	1	1

- **Latch T (Trigger)**
 - ✓ latch JK per cui $\mathbf{J} = \mathbf{K} = \mathbf{T}$
 - ✓ l'effetto è di invertire il valore dell'uscita \mathbf{Q} se l'ingresso \mathbf{T} vale 1

T	Q(t)	Q(t+τ)
0	0	0
0	1	1
1	0	1
1	1	0

Figura 5.16: Altri tipi di Latch (D e T)

5.5.2 Generatore di Impulsi

Un generatore di impulsi è un piccolo circuito logico utile a simulare un clock. Il concetto di base è quello di generare un ritardo (nello specifico sfruttiamo il ritardo di un **inverter**) per creare dei segnali intermittenti.

Nel diagramma in figura l'inverter induce un ritardo Δ che permette alla rete di cambiare il proprio stato. La durata Δ dell'impulso equivale al ritardo dell'inverter.

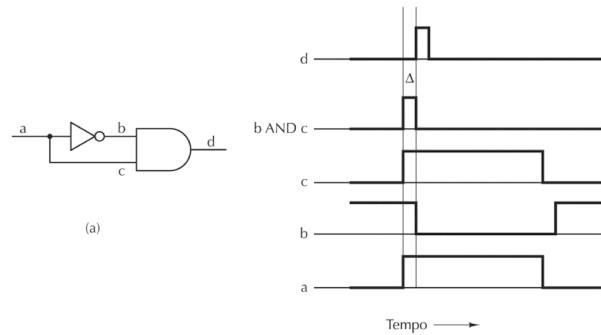


Figura 5.17: Generatore di impulsi tramite inverter

Simboli circuituali dei Latch

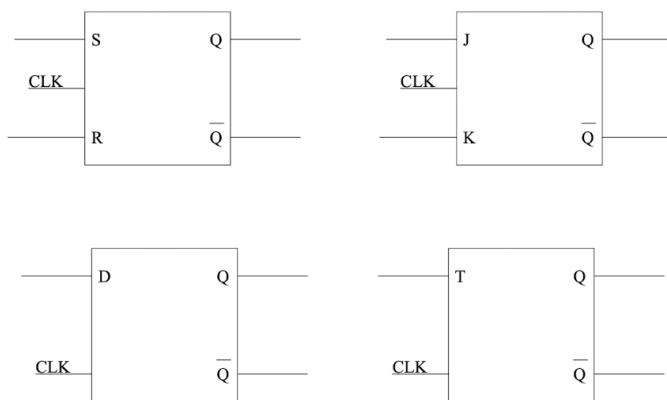


Figura 5.18: Schema Riassuntivo dei Latch

5.5.3 Elementi di Memoria Master-Slave (Flip-Flop)

Concetti Fondamentali

- Cosa sono gli elementi di memoria?** Gli elementi di memoria sono circuiti che **salvano 1 bit** (0 oppure 1). A differenza della **logica combinatoria** (dove l'uscita dipende solo dagli ingressi in quel momento), qui l'uscita **dipende anche dal passato** (mantengono informazione).
- Il problema del "Latch Trasparente"** Un Latch è semplice, ma è **trasparente**: quando l'ingresso cambia, l'uscita può cambiare subito fintanto che il segnale di abilitazione è attivo. Questo può creare errori perché non c'è un momento preciso in cui "bloccare" il valore.

La soluzione: Il Master-Slave

Un **Flip-Flop Master-Slave** è formato da **due latch** messi in cascata:

- **Master** = primo latch.
- **Slave** = secondo latch.

Funzionano alternati, grazie alla negazione del segnale di **clock**:

- Quando il **clock** è **ALTO** → il Master legge l'ingresso, lo Slave è bloccato (mantiene il vecchio valore).
- Quando il **clock** è **BASSO** → il Master si blocca (mantiene il valore letto) e lo Slave prende il valore dal Master (aggiorna l'uscita).

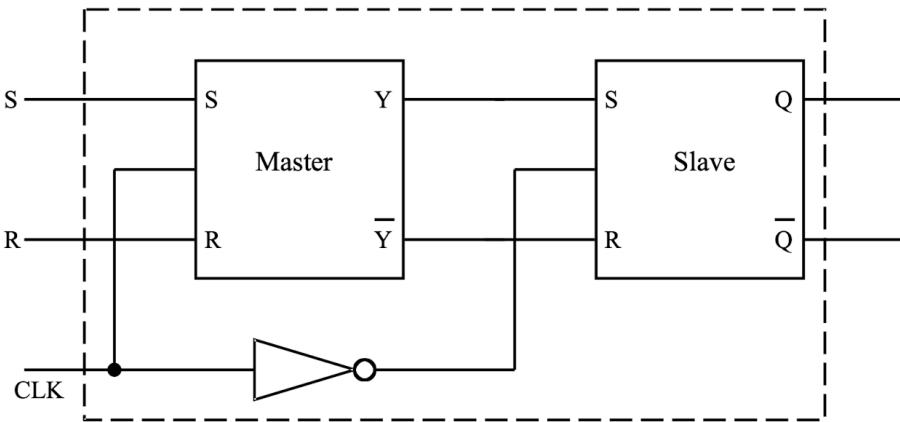


Figura 5.19: Logica interna Master-Slave

Nota Bene

L'uscita cambia solo in corrispondenza del fronte di clock (discesa, in questo esempio), non in modo "sporco".

Vantaggi:

- Uscita stabile e sincronizzata col **clock**.
- Niente più effetto "trasparenza" del latch.
- È la base di **registri, contatori, memorie digitali**.

Simboli circuituali dei Flip-Flop

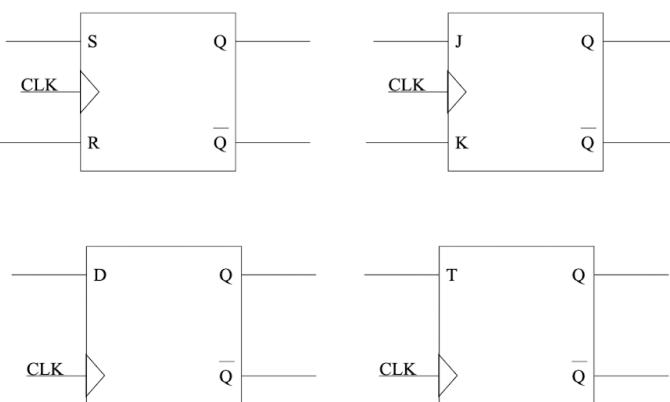


Figura 5.20: Rappresentazione schematica dei Flip-Flop

5.5.4 Analisi e Sintesi di una Rete Sequenziale

Analisi Consiste nel passaggio dal circuito alla sua descrizione.

- Si descrive mediante tabelle e diagrammi.

Sintesi Consiste nel passaggio dalla descrizione delle funzionalità alla realizzazione del circuito.

- Tipicamente questa fase ha in ingresso una descrizione (anche qualitativa) del comportamento che il circuito deve avere, e si progetta a step la circuiteria.
- Flusso: Diagrammi → Tabelle → Equazioni → Circuito.

5.6 Operazioni Aritmetiche Binarie

5.6.1 Somma Binaria

- La somma segue le regole standard del sistema binario.
- È possibile che si verifichi **Overflow** quando servono più bit di quelli disponibili:
 - Il risultato diventa errato.
 - Il segno può risultare discordante da quello degli addendi (se hanno lo stesso segno).
- I numeri negativi sono rappresentati in **complemento a 2**.

5.6.2 Sottrazione Binaria

- Si effettua sottrazione sommando il minuendo con il **complemento a 2** del sottraendo.
- Sono necessari solo:
 - Circuiti per l'addizione;
 - Circuiti per la complementazione.
- Valgono le stesse condizioni di **overflow** della somma.

Schema di un Sommatore

- Può utilizzare registri dedicati per il risultato.
- **Segnali di controllo:**
 - **OF:** Overflow bit.
 - **SW:** Switch bit (indica se eseguire addizione o sottrazione).

5.6.3 Moltiplicazione Binaria

Algoritmo base:

1. Generare i prodotti parziali (uno per ogni bit del moltiplicatore).
2. Se il bit del moltiplicatore è 0 → prodotto parziale = 0.
3. Se il bit del moltiplicatore è 1 → prodotto parziale = moltiplicando.
4. Ogni prodotto parziale viene **shiftato a sinistra** di una posizione rispetto al precedente.
5. Si sommano i prodotti parziali per ottenere il risultato.

Problema con numeri negativi: L'algoritmo base non funziona correttamente con i numeri negativi.

- **Soluzione 1:** Convertire i fattori in positivi, applicare l'algoritmo e negare il risultato se i segni sono discordi.
- **Soluzione 2:** Usare l'**Algoritmo di Booth (1952)**.
 - Più rapido.
 - Non richiede la negazione del risultato a posteriori.

5.6.4 Divisione Binaria

- Operazione più complessa della moltiplicazione.
- Si può eseguire tramite **sottrazioni ripetute**.

Algoritmo base (numeri interi senza segno):

1. Allineare a sinistra dividendo e divisore.
2. Confrontare: se la parte alta del dividendo \geq divisore:
 - Sottrarre il divisore.
 - Inserire 1 nel quoziente.
3. Altrimenti (se dividendo $<$ divisore):
 - Inserire 0 nel quoziente.
4. Shiftare il divisore di un bit a destra.
5. Ripetere fino a che il divisore non può più scalare.
6. La differenza finale è il **resto**.

5.6.5 Confronto operazioni aritmetiche

Operazione	Metodo principale	Note operative	Problemi principali
Somma	Addizione bit a bit con propagazione del riporto	Stesse regole dell'aritmetica binaria classica	Overflow se servono più bit di quelli disponibili
Sottrazione	Somma del minuendo con il complemento a 2 del sottraendo	Richiede solo addizionatore e circuiti per complemento a 2	Stessi problemi di overflow della somma
Moltiplicazione	Generazione di prodotti parziali e somma, con shift a sinistra	Se bit moltiplicatore = 1 \rightarrow prodotto parziale = moltiplicando	Con numeri negativi serve conversione o Algoritmo di Booth
Divisione	Sottrazioni ripetute con shift del divisore verso destra	Quoziente costruito progressivamente (1 o 0)	Operazione più lenta e complessa rispetto alla moltiplicazione

Tabella 5.3: Riepilogo delle operazioni aritmetiche

5.7 Operazioni Fondamentali in Floating Point

- **Addizione/Sottrazione:** si eseguono in 4 passi:
 1. Controllo se uno dei due operandi è nullo.
 2. **Allineamento delle mantisse** (correggendo gli esponenti).
 3. Esecuzione dell'operazione sulle mantisse.
 4. **Normalizzazione** del risultato (se necessario incrementando l'esponente).
- **Moltiplicazione/Divisione:**
 - Sono più semplici: agiscono separatamente su esponenti e mantisse.
 - **Esponenti** → somma o differenza.
 - **Mantisse** → moltiplicazione o divisione.

5.7.1 Esempio di somma Floating Point

- Dati due numeri FP, se gli esponenti sono diversi si porta il più piccolo al valore del maggiore.
- Le mantisse vengono sommate (includendo il bit隐含的).
- Se c'è un **riporto finale**, si incrementa l'esponente e si normalizza la mantissa.

5.7.2 Floating Point Unit (FPU)

- **Funzione:** eseguire operazioni su numeri in virgola mobile.
- **Struttura:** composta da due unità aritmetiche in virgola fissa:
 - Una per la **mantissa** (somma, sottrazione, moltiplicazione, divisione).
 - Una per l'**esponente** (solo somma algebrica e confronto).

Il confronto degli esponenti può essere fatto tramite **sottrazione**.

Procedura per la somma FP:

1. Si calcola la differenza degli esponenti.
2. Il numero con esponente minore viene shiftato a destra sulla mantissa.
3. Un contatore decrementa fino ad annullare la differenza.
4. Poi si procede all'operazione sulla mantissa.

5.7.3 Tabella Riassuntiva Operazioni FP

Operazione	Passi principali	Note
Addizione	1. Confronto esponenti 2. Allineamento mantisse (shift della più piccola) 3. Somma delle mantisse 4. Normalizzazione (eventuale incremento esponente)	Operazione più complessa per la necessità di allineare e normalizzare
Sottrazione	Stessi passi dell'addizione, ma operazione di differenza tra mantisse	Richiede attenzione ai segni degli operandi
Moltiplicazione	1. Somma degli esponenti 2. Moltiplicazione delle mantisse 3. Normalizzazione	Più semplice: esponenti e mantisse trattati separatamente
Divisione	1. Differenza degli esponenti 2. Divisione delle mantisse 3. Normalizzazione	Analogamente più semplici rispetto ad addizione/-sottrazione

Tabella 5.4: Riepilogo delle operazioni in virgola mobile

5.8 ALU (Arithmetic Logic Unit)

È il componente che si occupa di eseguire le operazioni aritmetiche all'interno del nostro calcolatore. Per essere tale ha bisogno dei seguenti requisiti:

- Registri:** per conservare gli operandi e i risultati parziali.
- Bit di controllo:** per indicare il tipo di operazione voluta.
- MUX:** per interpretare i bit di controllo.
- Rete combinatoria:** per effettuare le operazioni.

[Image of ALU Arithmetic Logic Unit block diagram]

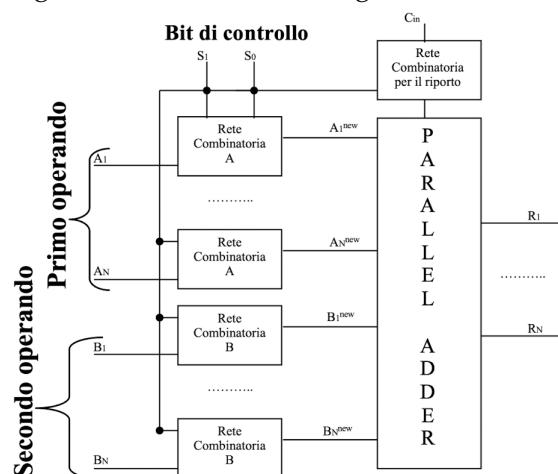


Figura 5.21: Schema concettuale di una ALU

Nota Bene

È inoltre indispensabile avere un blocco in grado di eseguire **operazioni in virgola Mobile**, una **FPU (Floating Point Unit)**.

5.8.1 Sommatori di un modulo ALU

Half-Adder

Il tipo di sommatore più elementare. Dati 2 bit in ingresso restituisce la loro somma e l'eventuale riporto.

[Image of half adder logic circuit diagram]

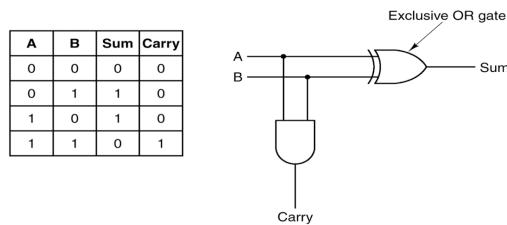


Figura 5.22: Circuito Half-Adder

- **Ingressi:** 2 bit (i 2 operandi da sommare).
- **Uscite:** 2 bit (somma dei due operandi e eventuale riporto).
- **Somma:** Ottenuta mediante XOR dei due ingressi ($A \oplus B$).
- **Riporto:** Ottenuto mediante AND dei due ingressi ($A \cdot B$).

Full-Adder

Il passo successivo a livello di implementazione. Dati 3 bit in ingresso (operandi da sommare e eventuale riporto precedente) restituisce la somma e un eventuale riporto.

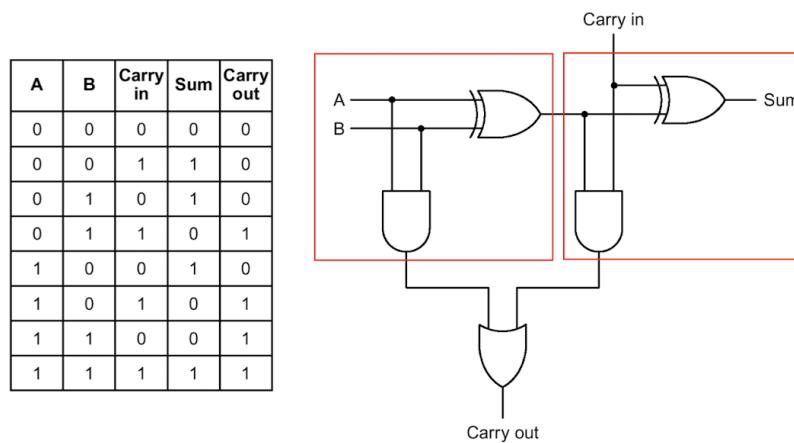


Figura 5.23: Circuito Full-Adder composto da due Half-Adder

Come possiamo vedere dall'immagine è sostanzialmente composto da 2 Half-Adder:

1. Il primo (da sinistra) esegue la somma dei due operandi in ingresso e restituisce somma parziale e riporto.
2. Il secondo somma il risultato del primo Half-Adder e l'eventuale riporto in ingresso (C_{in}), generando la somma totale.

Nota Bene

I due riporti parziali vengono messi in OR: basta che uno dei due Half-Adder abbia overflow per generare un riporto. Per la natura dello XOR, non potremo mai avere riporto in entrambi gli Half-Adder contemporaneamente.

5.8.2 ALU a 1 Bit

Questa rappresenta il blocco fondamentale per costruire ALU più complesse a 32 o 64 bit.

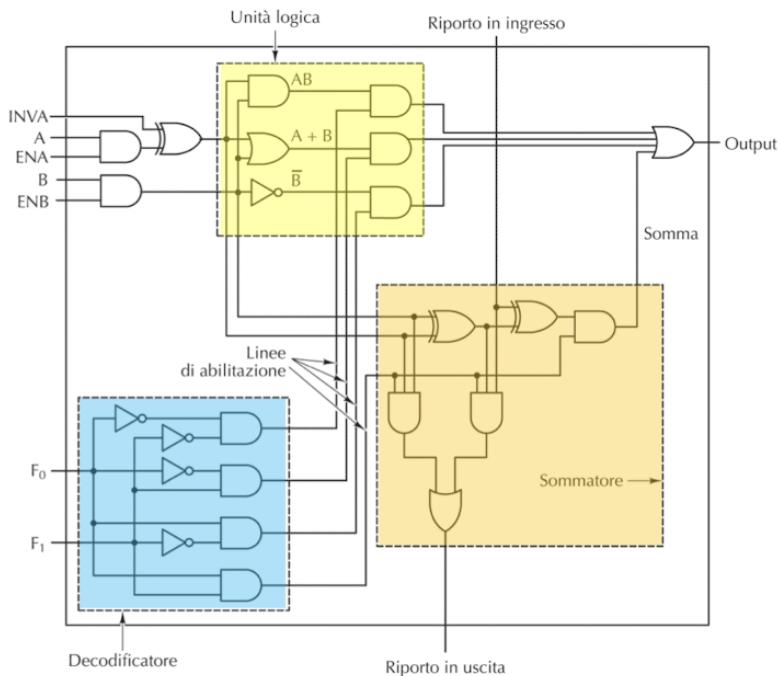


Figura 5.24: Architettura interna di una ALU a 1 bit

Componenti Principali

1. Unità Logica (in alto a sinistra)

- **Funzione:** Esegue operazioni logiche tra i bit A e B.
- **Operazioni:** AND ($A \cdot B$), OR ($A + B$), NOT B (\bar{B}).
- **Controllo:** Le linee INVA, A, ENA, B, ENB determinano quale operazione viene eseguita.

2. Decodificatore (in basso a sinistra - area blu)

- **Funzione:** Interpreta i segnali di controllo F_0 e F_1 .
- **Scopo:** Determina quale operazione deve essere eseguita.

3. Sommatore Full-Adder (in basso a destra - area gialla)

- **Funzione:** Esegue operazioni aritmetiche (addizione/sottrazione).
- **Riporti:** C_{in} dal bit precedente, C_{out} verso il successivo.

4. Multiplexer di Output • Funzione: Seleziona se inviare in output il risultato dell'unità logica o del sommatore, basandosi sui segnali del decodificatore.

Funzionamento: I bit A e B vengono inseriti come input. I segnali F_0 e F_1 specificano l'operazione; il decodificatore attiva l'unità appropriata e il MUX seleziona il risultato finale.

5.8.3 ALU a N Bit

Per ottenere ALU a N Bit ci basterà mettere in sequenza tante ALU a 1 Bit, passando da una all'altra gli eventuali riporti delle operazioni.

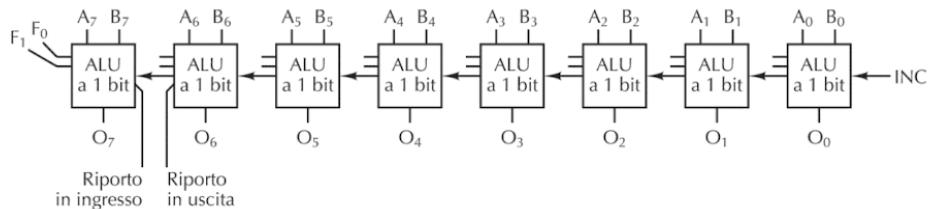


Figura 5.25: ALU a N bit (Ripple Carry)

Nota Bene

Ogni stadio deve aspettare che lo stadio precedente termini l'operazione e gli fornisca il riporto prima di poter eseguire le sue. Questo genera un ritardo che cresce linearmente con il numero di bit ($O(n)$): terribile per un modulo Hardware.

La soluzione è inserire delle **Ottimizzazioni nei sommatori**.

5.8.4 Ottimizzazioni nei Sommatori

Carry Look-Ahead Adder (CLA)

Problema del Parallel Adder (Ripple Carry Adder): il tempo di somma cresce linearmente con il numero di bit n . **Soluzione:** calcolare i riporti in **parallelo** usando le relazioni tra bit di ingresso.

Definizioni:

- g_i (**Generate**): $A_i \cdot B_i \rightarrow$ il bit genera sempre un riporto.
- p_i (**Propagate**): $A_i \oplus B_i \rightarrow$ il riporto viene propagato.
- **Formula ricorsiva del riporto:**

$$C_{i+1} = g_i + (p_i \cdot C_i)$$

Vantaggi: I riporti non dipendono più dalla propagazione sequenziale ma vengono calcolati da una **rete combinatoria**. Ritardo complessivo = ritardo della rete CLA + singolo Full Adder.
Svantaggio: circuito più complesso all'aumentare dei bit.

5.8.5 Carry Save Adder (CSA)

Utilizzato quando si devono sommare **più di due operandi** (es. nelle moltiplicazioni). **Idea:** evitare la propagazione dei riporti a ogni somma parziale.

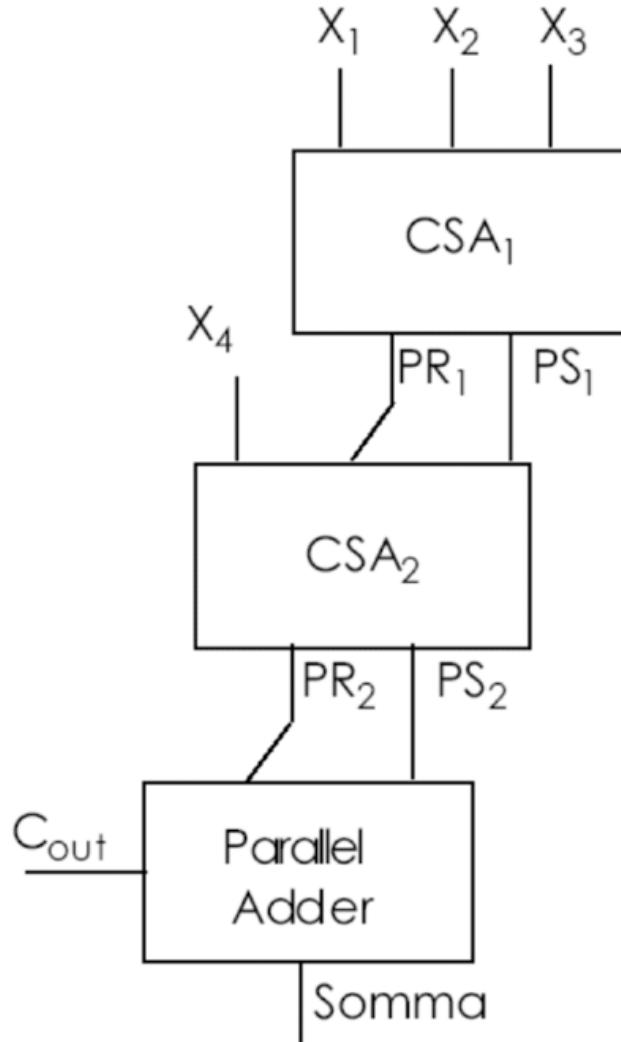


Figura 5.26: Principio del Carry Save Adder

Principio: Dati 3 addendi X, Y, Z , la somma produce due uscite:

1. **Pre-somma (S^*):** somma parziale senza riporti.
2. **Pre-riporto (C^*):** riporti generati, shiftati di un bit a sinistra.

La somma finale è $S = S^* + (C^* \ll 1)$.

Caratteristiche:

- Ogni CSA è formato da n Full Adder in parallelo.
- Il risultato finale si ottiene con un Parallel Adder solo all'ultimo stadio.
- **Vantaggio:** riduzione drastica del ritardo (nessuna attesa del riporto in cascata tra gli stadi intermedi).

5.8.6 Confronto Sommatori

Sommatore	Caratteristica principale	Pro/Contro
Ripple Carry Adder	Propagazione sequenziale del ripporto	(+) Semplice (-) Ritardo $O(n)$
Carry Look-Ahead	Calcolo dei riporti in parallelo tramite rete combinatoria	(+) Molto veloce (-) Circuito complesso
Carry Save Adder	Somma di più operandi senza propagazione immediata	(+) Ideale per moltiplicazioni (-) Serve un adder finale

Tabella 5.5: Confronto tra le tipologie di sommatori

Capitolo 6

Memorie e Cache

6.1 Introduzione alle Memorie

In questo capitolo andremo a studiare alcune **Reti logiche sequenziali** note e l'organizzazione fisica della memoria.

6.1.1 Dispositivi a 3 Stati (Tri-state Buffer)

È un dispositivo di controllo usato per la connessione al BUS e in generale per tutte le comunicazioni bidirezionali. Ha un segnale in input che rappresenta il dato e un secondo segnale in input che rappresenta invece il segnale di controllo. In base al segnale di controllo permette o meno il passaggio del dato (stato di alta impedenza).

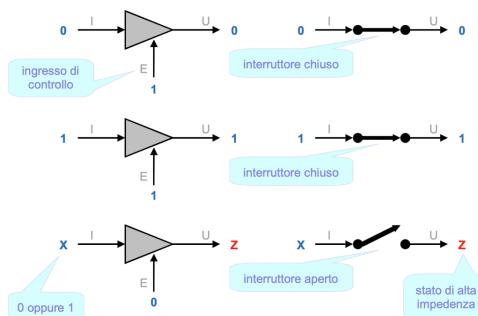


Figura 6.1: Simbolo del Tri-state Buffer

Nota Bene

Il canale dati deve essere bidirezionale, per permettere il passaggio da slave a master (lettura) e da master a slave (scrittura). Questo è fondamentale nel caso di utilizzo del BUS.

6.1.2 Chip di Memoria

Un **chip di memoria** è visto come una **matrice di celle** organizzate in **righe e colonne**. Per selezionare l'operazione da eseguire e le celle interessate usiamo 3 **segnali di controllo**:

1. **CS (Chip Select)** → abilita/disabilita il chip.
2. **OE (Output Enable)** → abilita l'uscita dei dati (lettura).

- 3. **WE (Write Enable)** → stabilisce se scrivere o leggere.



Figura 6.2: Organizzazione interna di un chip di memoria

Memorie Dinamiche (DRAM)

Nelle memorie dinamiche, per ridurre il numero di piedini, l'indirizzo viene fornito in due parti (multiplexing degli indirizzi):

- **RAS (Row Address Strobe)** → seleziona la riga.
- **CAS (Column Address Strobe)** → seleziona la colonna.

In questo modo con n linee di indirizzo si possono indirizzare 2^n celle, ma usando RAS/-CAS lo stesso bus indirizzi può essere sfruttato due volte (prima per la riga, poi per la colonna), dimezzando i pin necessari per l'indirizzamento.

6.1.3 Banchi di Memoria

Se vogliamo ottenere memorie più grandi sarà necessario avere **più chip in parallelo**, formando così un **banco di memoria**.

Ogni chip fornisce solo **una parte della parola**:

- Esempio: se un chip è largo **1 bit**, servono n chip in parallelo per memorizzare una parola di n bit.

Nota Bene

Gli **indirizzi** sono condivisi da tutti i chip del banco → in questo modo la stessa posizione viene selezionata contemporaneamente in ogni chip.

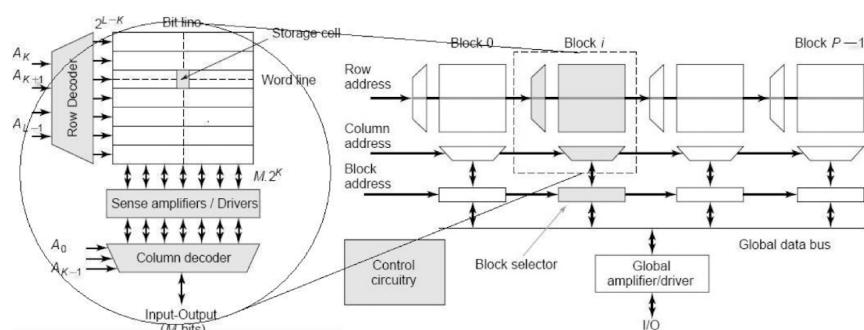


Figura 6.3: Collegamento in parallelo per formare un banco

Se le parole sono più grandi della dimensione delle righe del chip di memoria usato, sarà necessario collegare tra loro i chip.

Nell'immagine a fianco possiamo notare un esempio di organizzazione nel caso in cui la parola sia da 128 bit.

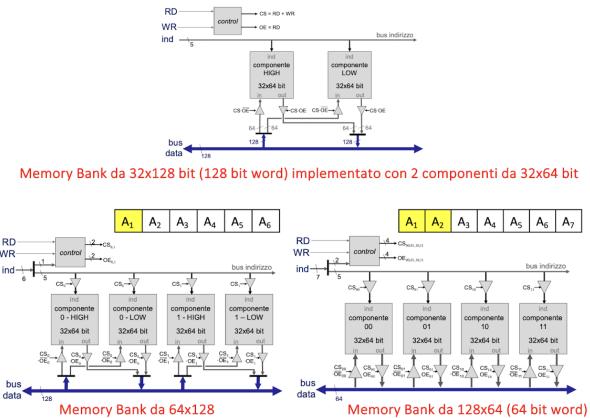


Figura 6.4: Banco di memoria a 128 bit

Segnali di controllo nei banchi

I **segnali di controllo** possono essere sia **unificati** (in modo da attivare tutti i chip insieme), o **differenziati** (in modo da dividere il banco di memoria in più blocchi).

Esempi di segnali:

- **RD**: segnale univoco che ci dice se siamo in fase di scrittura o lettura.
- **CS**: (*Chip Select*) ci dice se siamo autorizzati ad accedere a quell'area di memoria.
- **OE**: (*Output Enable*) ci dice se siamo autorizzati alla scrittura sul BUS.

A seconda dei bisogni posso usare banchi di memoria diversi, che possono variare per:

- **Memorie più larghe** (aumentando i bit per parola).
- **Memorie più grandi** (aumentando il numero di parole).
- **Memorie più veloci**.

6.2 Memorie a Semiconduttore

Di seguito una tabella riassuntiva delle principali tecnologie di memoria:

6.2.1 RAM (Random Access Memory)

Quando parliamo della “memoria del computer” ci riferiamo spesso alla sua memoria RAM. Ai nostri giorni i computer hanno dai 4GB fino a 128 GB (fascia alta), ma non c’è un limite teorico massimo: con budget e spazio hardware sufficienti è possibile aumentare di molto questi numeri.

La **RAM** è una *memoria volatile*: significa che una volta terminato il suo compito (o tolta l’alimentazione) non conserva i dati al suo interno. A compenso di ciò è molto veloce; la sua velocità è fondamentale per le prestazioni del sistema. La lettura e la scrittura avvengono mediante segnali elettrici.

Tipo	Categoria	Cancellabilità	Scrittura	Permanenza
RAM (Random Access Memory)	Scrittura / Lettura	Elettrica, Livello di Byte	Elettrico	Volatile
ROM (Read Only Memory)	Sola Lettura	Non Possibile	Maschere (in fabbrica)	Non Volatile
PROM (Programmable ROM)	Sola Lettura	Non Possibile	Elettrico	Non Volatile
EPROM (Erasable PROM)	Principalmente Lettura	Luce UV, Livello Chip	Elettrico	Non Volatile
EEPROM (Electrically EPROM)	Principalmente Lettura	Elettrica, Livello di Byte	Elettrico	Non Volatile
Flash	Principalmente Lettura	Elettrica, Livello di Blocco	Elettrico	Non Volatile

Tabella 6.1: Confronto tecnologie di memoria

Tipologie di RAM

- **DRAM (RAM Dinamica):** I dati sono memorizzati tramite l'iniezione di cariche elettriche in condensatori; la lettura avviene rilevando la presenza di cariche.
 - Richiede un'operazione periodica di ripristino delle cariche (*refresh*).
- **SRAM (RAM Statica):** È realizzata mediante Flip-Flop tradizionali.
 - Più costosa della DRAM.
 - Mantiene il contenuto fino a che viene alimentata (non serve refresh).

6.2.2 ROM (Read Only Memory)

La memoria ROM rappresenta lo spazio di archiviazione base del sistema (es. BIOS/Bootloader). Nasce come memoria in **sola lettura, non riscrivibile** e ha il vantaggio che i contenuti sono persistenti.

(+) **Vantaggi:** I dati sono reperibili direttamente nello spazio di indirizzamento della memoria principale.

(-) **Svantaggi:**

- Costo fisso elevato per la fabbricazione (scrittura tramite maschere litografiche).
- Non sono ammessi errori, altrimenti la ROM è da buttare.

PROM (Programmable ROM)

È una ROM riprogrammabile (riscrivibile) anche in fasi successive alla fabbricazione, ma rimane **non volatile**. È più flessibile di una normale ROM ma per essere scritta necessita di Hardware dedicato (*programmatore di ROM*).

Nota Bene

Una volta scritta, non può essere cancellata (o la cancellazione è complessa e dipende dal tipo specifico).

6.2.3 Memorie Read Mostly

Sono memorie **non volatili riprogrammabili** ma con un importante squilibrio nell'utilizzo verso la lettura.

EPROM (Erasable PROM)

È una PROM cancellabile. Ad ogni scrittura si devono riportare tutte le celle allo stato iniziale mediante esposizione a raggi ultravioletti (UV).

- Ogni cancellazione dura circa 20 min.
- La cancellazione avviene a livello dell'intero Chip.

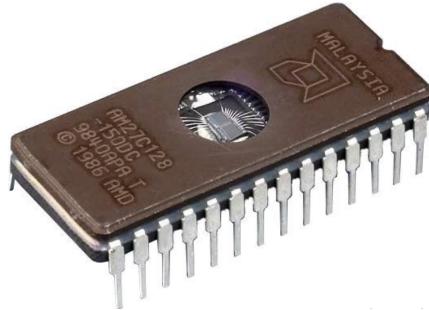


Figura 6.5: Chip EPROM con finestra al quarzo per raggi UV

EEPROM (Electrically Erasable PROM)

Sono memorie non volatili che mantengono i dati anche senza alimentazione.

- Si possono cancellare e riscrivere **elettricamente**.
- Cancellazione a livello di **Byte** (più precisa delle EPROM).
- Usate per configurazioni, BIOS moderni, firmware.

6.2.4 Memorie Flash

Sono memorie **non volatili** a semiconduttore che derivano dalle EEPROM, ma sono più veloci e convenienti. Sono organizzate a **blocchi**.

Caratteristiche Principali

- **Lettura:** veloce (simile a DRAM). Posso indirizzare interi blocchi o specifiche righe.
- **Scrittura:** più lenta, richiede la cancellazione preventiva di un intero **blocco**.
- **Cancellazione:** avviene per blocchi (molto più grande della singola cella → limite rispetto alle EEPROM).
- **Durata:** numero limitato di *cicli di scrittura/cancellazione* (usura delle celle).

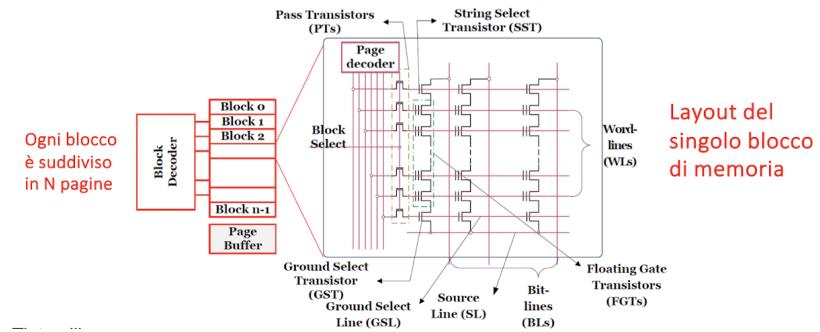


Figura 6.6: Organizzazione a blocchi della memoria Flash

Tipi di Flash

NOR Flash Accesso diretto a livello di byte. Maggiore velocità in lettura (usata per esecuzione codice/firmware). Più costosa e meno densa.

NAND Flash Organizzata a blocchi con accesso sequenziale. Maggiore densità, più economica. Usata per archiviazione di massa (SSD, USB, SD).

Ciclo di vita delle memorie Flash

A differenza delle EEPROM (aggiornabili a byte), nelle Flash:

1. La cancellazione avviene tramite segnali elettrici.
2. L'aggiornamento deve avvenire a livello superiore (Blocco o Pagina).

Differenza tra Block-Based e Page-Based:

- **Operazioni su Pagina (Scrittura):** La *pagina* è l'unità logica più piccola per la programmazione (scrittura).
- **Operazioni su Blocco (Cancellazione):** La cancellazione (*erase*) deve avvenire sull'unità più grande, il *blocco*.

Nota Bene

Poiché non posso sovrascrivere un singolo byte senza cancellare il blocco: se modifco un dato in una pagina, devo **cancellare l'intero blocco** e riscriverlo. Questo fa sì che ogni piccolo aggiornamento consumi un intero Ciclo P/E (Program/Erase) del blocco.

La durata del componente dipende quindi dalla dimensione del blocco. Per mitigare l'usura si utilizza il **Wear Leveling**.

Definizione 6.1: Wear Leveling

Tecnica che distribuisce uniformemente le scritture su tutti i blocchi fisici della memoria, evitando che alcuni blocchi vengano scritti/cancellati troppo spesso e si rompano prematuramente.

Prestazioni

- Tempo di accesso in lettura: 10 – 50 ns.
- Tempo di scrittura: 200 – 500 μ s (molto più lento).
- Tempo di cancellazione blocco: anche millisecondi.

6.3 Temporizzazione delle Memorie

Adesso andiamo a vedere la *temporizzazione delle memorie*, in particolare quali sono i parametri (tempi) che caratterizzano il lavoro delle memorie. Sono aspetti che incidono significativamente sulla velocità delle memorie, da cui possiamo ricavare i loro tempi di risposta.

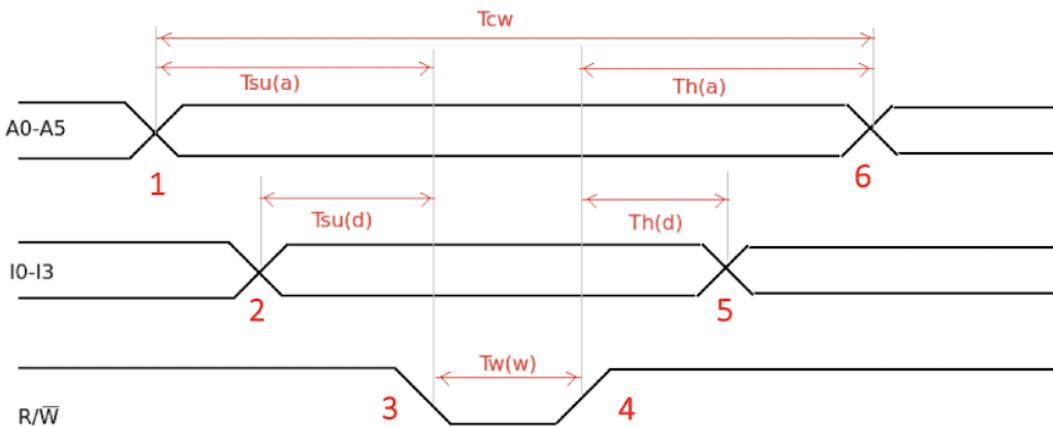


Figura 6.7: Diagramma temporale del ciclo di scrittura

Analizziamo la sequenza temporale (riferimento Fig. 6.7):

1. **Imposizione dell'indirizzo del dato:** all'istante 1 incontriamo per la prima volta l'indirizzo del dato richiesto. In questo momento **iniziano** i seguenti tempi:
 - **TCW (Time Cycle Write):** tempo minimo di scrittura, ossia il tempo minimo che deve passare tra due operazioni di scrittura.
 - **TSU(a) (Time Set-Up Address):** tempo di set-up, rappresenta il tempo minimo in cui il segnale indirizzo deve rimanere stabile per essere considerato valido.
2. **Fornitura del dato:** all'istante 2 viene fornito il dato da scrivere in memoria. In questo istante **inizia**:
 - **TSU(d) (Time Set-Up Data):** tempo di set-up, rappresenta il tempo minimo in cui il segnale dati deve rimanere stabile per essere considerato valido.
3. **Impulso di scrittura:** in questo istante si presenta il primo segnale che avvia la fase di scrittura in memoria. A questo punto **terminano** i tempi TSU, e **inizia**:
 - **TW(w) (Time Width Write):** rappresenta la durata minima dell'impulso di scrittura.
4. **Fine comando di scrittura:** in questo istante termina l'impulso di scrittura, e con esso **termina il tempo TW(w)**. In questo istante **iniziano** i tempi di mantenimento:
 - **TH(a) (Time Hold Address):** tempo di hold, rappresenta il tempo in cui l'indirizzo deve rimanere stabile *a seguito* dell'operazione.

- **TH(d)** (*Time Hold Data*): tempo di hold, rappresenta il tempo in cui il dato deve rimanere stabile *a seguito* dell'operazione.
- Variazione del dato:** in questo istante termina la necessità di mantenere stabile il dato scritto; **termina il tempo TH(d)**.
 - Cambio di Indirizzo:** questo istante rappresenta la fine del ciclo di scrittura e con esso **termina il tempo TH(a)**, permettendo un conseguente cambio di indirizzo per il ciclo successivo.

Nota Bene

Per non confondersi:

- **Set-Up (T_{SU})**: Per quanto tempo il segnale deve essere stabile **PRIMA** del fronte di clock/impulso.
- **Hold (T_H)**: Per quanto tempo il segnale deve rimanere stabile **DOPPO** il fronte di clock/impulso.

6.4 Principio di Località e Gerarchie

Definizione 6.2: Principio di Località

Si è osservato che, in un breve lasso di tempo, la CPU dovrà accedere a determinati gruppi (*cluster*) di dati ed istruzioni.

Il concetto si distingue in due tipologie:

- **Località spaziale**: I prossimi dati e istruzioni da utilizzare probabilmente si troveranno in locazioni **contigue** a quelli attualmente in uso (cluster di locazioni).
 - Ciò avviene perché i programmi hanno un flusso di esecuzione tipicamente sequenziale e spesso usano strutture dati sequenziali (come vettori o matrici).
- **Località temporale**: Dati e istruzioni usati di recente è probabile che vengano riutilizzati entro breve tempo.
 - Ad esempio, istruzioni ripetute all'interno di cicli (*loop*).

6.4.1 Osservazioni fondamentali

Il principio di località si basa quindi su questi fatti empirici:

1. I programmi hanno un flusso di esecuzione tipicamente sequenziale.
2. Si hanno tante chiamate a procedure nidificate in sequenza, una dietro l'altra.
3. Nei programmi si hanno spesso gruppi di istruzioni che vengono ripetuti per un certo tempo (*loop*).
4. Nei programmi si utilizzano spesso strutture dati sequenziali.

6.4.2 Esempio: Gerarchia a 2 Livelli (M_1 e M_2)

Immaginiamo una gerarchia composta da due livelli:

- **Livello M_1 (Cache):** Più veloce e costoso.
 - Capacità: 1.000 parole.
 - Tempo di accesso (T_1): **0,1 μ s**.
- **Livello M_2 (Memoria Principale/DRAM):** Più lento ed economico.
 - Capacità: 100.000 parole.
 - Tempo di accesso (T_2): **1 μ s**.

Funzionamento: Tutte le informazioni presenti in M_1 sono anche in M_2 , ma non viceversa. Quando i dati non vengono trovati in M_1 (*Cache Miss*), si accede a M_2 , ma si copia in M_1 un intero **blocco di dati** contenente anche le locazioni contigue, sfruttando la Località Spaziale.

Calcolo del Tempo Medio di Accesso (T_s)

Proviamo a calcolare l'efficacia del sistema:

1. Ipotizziamo che il **95%** delle volte troviamo il dato richiesto nella cache (*Cache Hit*).
2. Il restante **5%** avremo un *Cache Miss*.
3. Quindi il rapporto di successo è $H = 0.95$.

La formula per il tempo medio è:

$$T_s = H \cdot T_1 + (1 - H) \cdot (T_1 + T_2)$$

Dove:

- $H \cdot T_1$: è il tempo speso quando l'accesso è un successo (Hit).
- $(1 - H)$: è la probabilità che l'accesso fallisca (Miss).
- $(T_1 + T_2)$: è il tempo speso per cercare in M_1 (fallendo) e poi accedere a M_2 , copiando il dato.

Sostituendo i valori dell'esempio:

$$T_s = (0.95 \cdot 0,1 \mu\text{s}) + (0.05 \cdot (0,1 \mu\text{s} + 1 \mu\text{s}))$$

$$T_s = 0.095 \mu\text{s} + (0.05 \cdot 1.1 \mu\text{s})$$

$$T_s = 0.095 \mu\text{s} + 0.055 \mu\text{s}$$

$$T_s = \mathbf{0.15 \mu\text{s}}$$

Risultato: Il tempo medio effettivo di accesso ($0,15 \mu\text{s}$) si avvicina molto al tempo di accesso della memoria più veloce ($0,1 \mu\text{s}$), dimostrando l'efficacia della gerarchia.

Nota Bene

Sfruttando questo principio, la gerarchia di memoria organizza i dati in modo che i *cluster* che servono in un dato momento si trovino ai livelli più alti e veloci della gerarchia.

6.5 Memoria cache

Nel contesto della gerarchia di memoria, la Cache nasce per **risolvere un problema fondamentale**: il **divario di velocità** tra la CPU (velocissima) e la Memoria Principale (più lenta).

La Cache agisce come una memoria di "primo livello": è piccola, costosa e molto veloce, e il suo scopo è contenere una copia delle porzioni di memoria che il processore usa più frequentemente, per averle subito a disposizione. Solitamente tra la CPU e la RAM ci sono tre livelli di cache, una più capiente dell'altra: minore è la capienza della Cache, maggiore è la velocità.

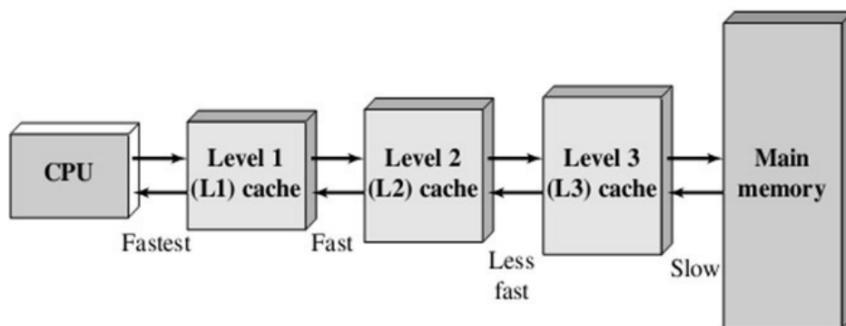


Figura 6.8: collegamento tra CPU e memoria primaria

6.5.1 Struttura della memoria primaria

Tuttavia, per capire come la Cache gestisce queste copie, dobbiamo analizzare **come avviene il trasferimento dei dati**. Come si può vedere in figura 6.10, mentre la CPU richiede singole parole (Word Transfer), la Cache e la Memoria Principale dialogano scambiandosi interi blocchi di dati (Block Transfer). La Cache, quindi, non carica singoli byte, ma "pacchetti" interi. Di conseguenza, per comprendere il funzionamento della Cache, è necessario prima definire formalmente **come è strutturata e suddivisa la Memoria Primaria**.

Analizzando la struttura della memoria primaria, vediamo che questa è costituita da 2^n parole indirizzabili, dove n è il numero di bit utilizzati per gli indirizzi. La logica di base è la **suddivisione in blocchi**:

- La memoria viene immaginata come divisa in blocchi di dimensione fissa, ognuno contenente k parole (words);
- Matematicamente, il numero totale di blocchi (M) che abbiamo nella memoria è dato dalla capacità totale divisa per la grandezza del singolo blocco ($2^n / k$).



Figura 6.9: struttura di un indirizzo di memoria

Questa suddivisione in blocchi si riflette direttamente su come la macchina interpreta l'indirizzo di memoria. Quando la CPU genera un indirizzo di n bit, questo non viene letto come un numero unico, ma viene diviso logicamente in due parti per individuare il dato all'interno della struttura a blocchi:

- **Block Frame (X bit):** I bit più significativi (la parte sinistra dell'indirizzo) ci dicono in quale blocco si trova il dato. È l'indirizzo del "contenitore";
- **Offset (Y bit):** I bit meno significativi (la parte destra) ci dicono la posizione esatta della parola all'interno di quel blocco.

In sintesi, gli X bit portano la Cache a trovare il blocco giusto, e gli Y bit selezionano la parola specifica richiesta dalla CPU.

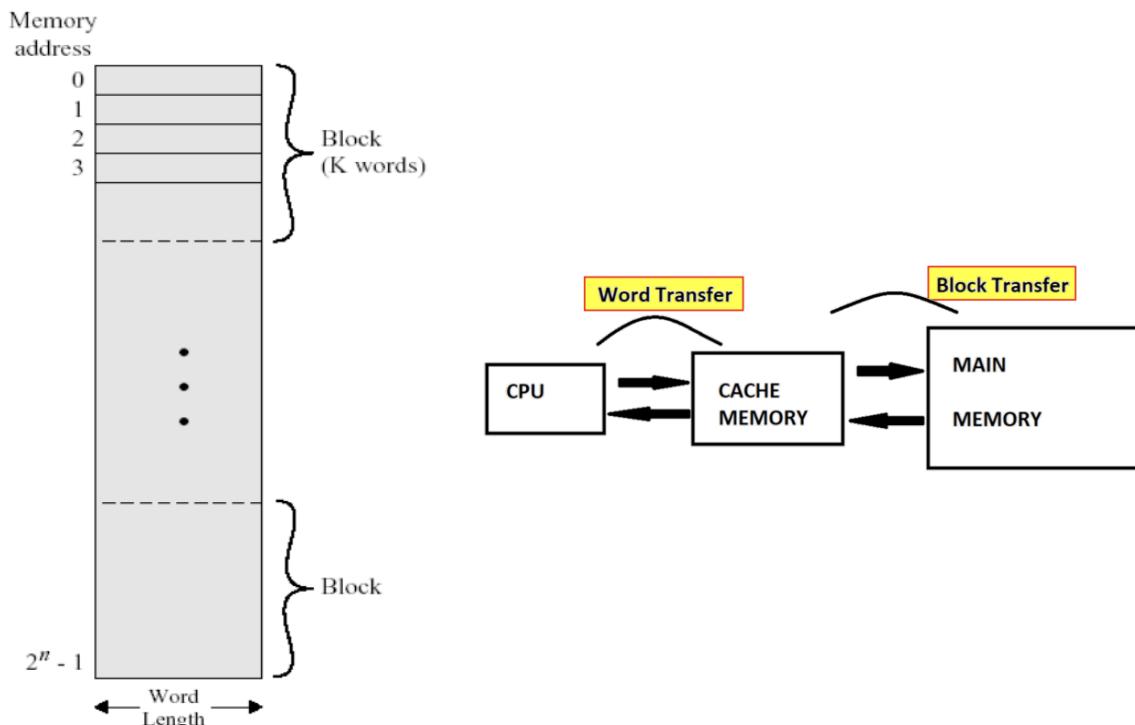


Figura 6.10: Trasferimento dei dati tra CPU, Cache e Memoria Principale

6.5.2 Struttura della cache

Abbiamo detto che la memoria principale è divisa in blocchi. La Cache, invece, è **strutturata in Slot** (o Linee/Righe), come in figura 6.11. Ogni slot può contenere un blocco della memoria principale (RAM) composto da k parole, le quali contengono l'informazione.

Si presenta però un problema fondamentale: la memoria principale è enorme, mentre la Cache è piccola. Abbiamo molti più blocchi in RAM che slot in Cache. Quindi, molti **blocchi diversi dovranno competere** per lo stesso slot nella cache.

Per fare in modo che la CPU sappia se il blocco contenuto in uno slot è quello che effettivamente sta cercando (e non un altro finito per caso in quella posizione) entra in gioco il concetto di **TAG** (l'etichetta): ogni slot della cache non contiene solo i dati, ma ha un'etichetta che indica *"il blocco numero X della RAM"*.

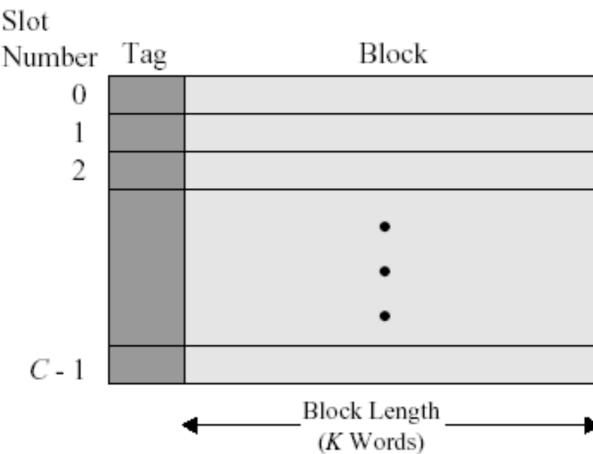


Figura 6.11: struttura della memoria cache

Nota Bene

Proprio perché il **TAG** indica un determinato blocco della RAM, viene utilizzato per l'indirizzamento della Cache.

Dato che ci sono più blocchi in RAM che slot in Cache, un singolo slot non può essere associato in modo univoco ad un unico blocco della Memoria Primaria, infatti il TAG permette di **identificare, per ogni istante**, quali blocchi della memoria principale sono presenti nelle slot della Cache.

Funzionamento della Cache

Quando la CPU cerca un dato, richiede l'accesso o ad una parola di memoria inviando l'indirizzo corrispondente della RAM, poi si controlla se la parola è presente in Cache (confrontando i Tag):

- Si parla di **Cache hit** quando il blocco **viene trovato** in Cache: in questo caso si accede alla parola direttamente ottenendo una lettura delle informazioni molto veloce;
- Si parla di **Cache miss** quando il blocco **non è presente** in Cache: in questo caso si copia l'intero blocco a cui appartiene la parola cercata dalla Memoria Principale (RAM) alla Cache, per poi trasferire la parola dalla Cache alla CPU.

In questo modo viene sfruttato il **principio di località** visto al capitolo 6.4.

Nella maggior parte degli accessi del processore alla memoria, la parola cercata dovrebbe essere già in cache.

6.5.3 Indirizzamento della Cache

Per quanto riguarda l'indirizzamento della memoria Cache bisogna aprire una parentesi importante. La CPU nei moderni calcolatori supporta la gestione dell'indirizzamento della memoria in modalità di **memoria virtuale**.

Questo meccanismo è fondamentale per **disaccoppiare la vista del programma** dalla memoria fisica reale. Se due processi decidessero di scrivere i loro dati nello stesso indirizzo della Memoria Primaria, andrebbero in conflitto.

Tramite la "*memoria virtuale*" viene permesso ai processi di utilizzare degli indirizzi "finti" (indirizzi logici), **indipendenti** sia da quelli che sono **presenti** nella memoria RAM che dalla memoria fisica effettivamente disponibile.

Ovviamente, per far funzionare questo meccanismo è necessario che gli indirizzi logici vengano poi **tradotti** in **indirizzi fisici** realmente **esistenti all'interno della RAM**: in questo modo due processi possono anche condividere lo stesso indirizzo logico, che verrà poi tradotto in un differente indirizzo fisico per ciascun processo.

L'utilizzo di questo meccanismo garantisce due vantaggi principali:

1. **Isolamento**: le applicazioni non rischiano di sovrascrivere dati di altri processi, poiché ognuna vive nel proprio spazio virtuale;
2. **Standardizzazione**: i programmati non devono preoccuparsi di dove il programma verrà caricato fisicamente nella RAM.

Il "*prezzo da pagare*" per questa comodità è la complessità hardware, ovvero, l'aggiunta di un componente che si occupa della traduzione il quale prende il nome di **Memory Management Unit** (MMU). Il suo compito è **intercettare ogni richiesta della CPU** e trasformare l'indirizzo virtuale (logico) generato dal processo, nel corrispondente indirizzo fisico reale dove si trova il dato nella Memoria Primaria.

Questa architettura pone una scelta progettuale critica, ovvero quella di scegliere dove posizionare la Cache rispetto alla MMU. Esistono due modalità:

Cache Logica (o virtuale)

In questa configurazione la Cache è posta tra la CPU e la MMU.

La Cache memorizza i dati usando direttamente gli indirizzi virtuali, e la CPU accede alla Cache subito, senza passare per la MMU.

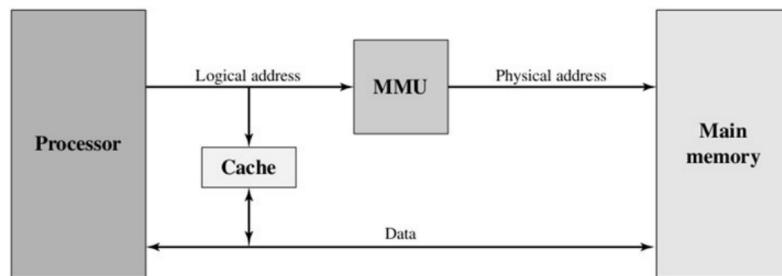


Figura 6.12: Cache logica

Il principale **vantaggio** di questa implementazione è che il **tempo di accesso è molto ridotto** poiché non bisogna attendere la traduzione dell'indirizzo.

Allo stesso tempo però, nasce una **problematica**.

Tutti i processi **condividono lo stesso spazio di indirizzamento virtuale**, e quando avviene il context switch (il salvataggio dell'applicazione per passare ad un'altra) non si vuole rischiare che il nuovo processo ottenga lo stesso indirizzo logico di un processo già presente in Cache poiché si otterrebbero due processi con lo stesso indirizzo logico che però devono fare riferimento a due indirizzi fisici differenti, il che causerebbe conflitti.

Per evitare questi conflitti, ad **ogni context switch**, la Cache deve essere **svuotata completamente**, il che comporta perdita di tempo.

La situazione sarebbe differente se si potessero distinguere i processi in Cache. Infatti, alternativamente si possono utilizzare dei **bit aggiuntivi** nel tag per indicare a quale spazio di indirizzamento virtuale si sta facendo riferimento.

Cache fisica

In quest'altra configurazione la Cache è posta tra la MMU e la Memoria Primaria. La cache memorizza i dati usando gli indirizzi fisici. La CPU deve prima passare per la MMU per tradurre l'indirizzo, e solo dopo accede alla cache.

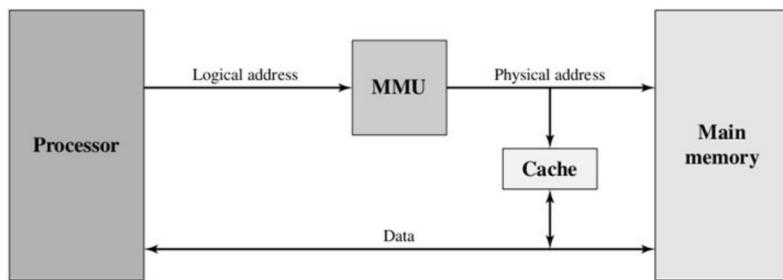


Figura 6.13: Cache fisica

In questo caso non c'è il rischio che si creino delle ambiguità, ogni indirizzo è univoco perché viene inserito nella Cache facendo riferimento direttamente alla Memoria primaria.

D'altra parte, lo **svantaggio principale** è che utilizzare una Cache fisica **comporta lentezza**, poiché ogni singola richiesta deve prima passare attraverso la traduzione della MMU.

6.5.4 Dimensione della Cache

Nel momento in cui si va a scegliere la dimensione della Cache, si cerca il **miglior compromesso** fra **costo e velocità**:

- **Cache piccola:** permette di mantenere sia i **costi** che i tempi di accesso alla Cache stessa bassi ("*cercare un dato in una stanza piccola è più rapido che cercarlo in un magazzino enorme*");
- **Cache grande:** il tempo impiegato per cercare al suo interno aumenta leggermente, ma si ha una probabilità maggiore di trovare il dato poiché sono presenti più blocchi di memoria primaria massimizzando così l'hit rate; dunque aumenta la **velocità** media percepita dal sistema.

Da questa distinzione si può dedurre che il tempo richiesto per cercare un blocco aumenta con la dimensione della Cache, e allo stesso tempo aumenta l'hit rate.

In sostanza la **dimensione ottimale** della cache dipende da quanto ogni processo rispetta il Principio di Località (capitolo 6.4), ma non esiste un valore ottimale assoluto.

6.5.5 Mappatura della Cache

Definizione 6.3: Mappare una Cache

La "*mappatura di una Cache*" è una tecnica che permette di decidere dove posizionare un blocco di RAM appena arrivato nella Cache.

La mappatura della Cache rappresenta un **procedimento complesso** poiché, come precedentemente introdotto al capitolo 6.5.2, sono disponibili molti più blocchi nella memoria primaria che slot all'interno della cache.

Per effettuare l'operazione di mappatura possono essere utilizzati **tre metodi differenti**:

- Diretto;
- Associativo;
- Associativo su insiemi

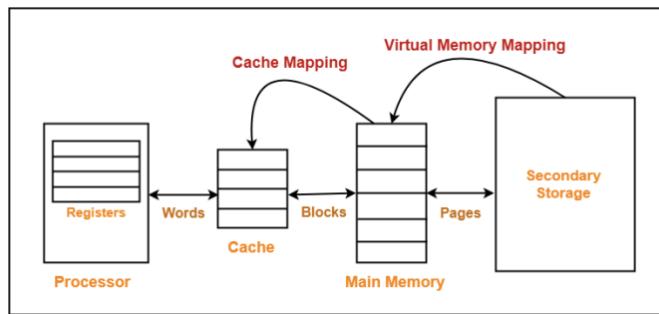


Figura 6.14: Schema generale del processo di mappatura

Mappatura diretta della Cache

Utilizzando la tecnica di mappatura diretta, distinguiamo due punti di vista differenti:

- **Punto di vista del blocco:** ogni blocco della memoria primaria può occupare solo uno slot specifico della Cache, poiché la sua posizione è vincolata dallo slot/line dell'indirizzo del blocco di memoria.

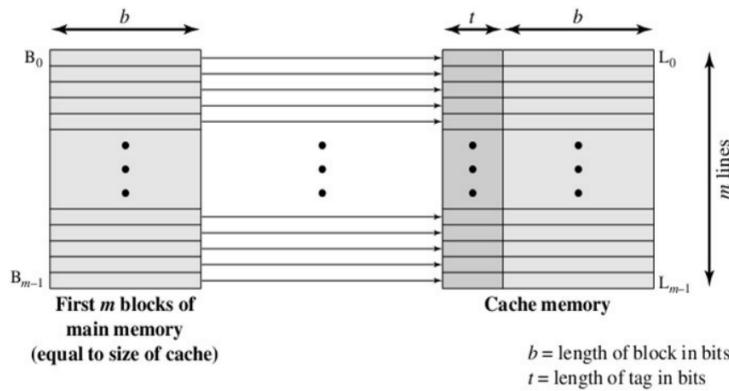


Figura 6.15: mappatura Cache 1:1

Questo vincolo è ottenibile tramite una formula matematica: se i è l'indirizzo della linea della cache, j è l'indirizzo del blocco di memoria ed m il numero di linee della cache, l'indirizzo di un blocco di memoria primaria in cache è: $i = j \bmod m$.

Nota Bene

L'operazione del modulo (\bmod) serve per fare in modo di ottenere sempre un indice **valido** compreso tra 0 e $m - 1$.

Esempio pratico:

Ipotizziamo una Cache con 10 posti ($m = 10$).

Blocco 5 \rightarrow 5 ($\bmod 10$) = 5 (va allo slot 5 della Cache).

- **Punto di vista dello slot:** cambiando punto di vista, diversi blocchi della memoria primaria possono essere mappati sullo stesso slot, poiché sono vere anche le seguenti affermazioni (sempre considerando $m = 10$).

Blocco 15 \rightarrow 15 ($\bmod 10$) = 5 (va allo slot 5 della Cache).

Blocco 25 \rightarrow 25 ($\bmod 10$) = 5 (va allo slot 5 della Cache).

Questo significa che i blocchi distanti **esattamente m posizioni** nella RAM **competeranno** per lo stesso slot nella cache (capitolo 6.5.2). Come si può vedere in figura 6.16, sono

rappresentati più blocchi della RAM associati ad un singolo slot di Cache:

$$i_1 = |00001| = |01001| = |10001| = |11001| = 001$$

$$i_5 = |00101| = |01101| = |10101| = |11101| = 101$$

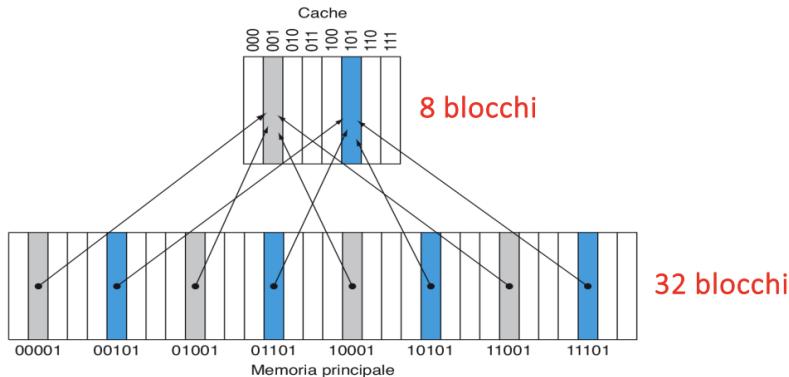


Figura 6.16: mappatura Cache n:1

Per implementare questo sistema, l'indirizzo di memoria (di n bit) viene tagliato in 4 pezzi:

- **word (w)**: sono i bit meno significativi (più a destra), e identificano la singola word all'interno del blocco;
- **byte (b)**: indicano quale singolo byte prendere all'interno della word;
- Gli s bit (dove $s = t + r$) costituiscono l'indirizzo del blocco, sono i bit più significativi e vengono ulteriormente suddivisi in:
 - **slot/line (r)**: Indicano in quale riga della cache andare a guardare (l'indice della tabella);
 - **tag (t)**: permettono di riconoscere il blocco di memoria primaria associato a quella linea di Cache.

Dunque, $n = t + r + w + b$ sono i bit totali di indirizzamento della memoria.

TAG (t bits)	SLOT/LINE (r bits)	WORD (w bits)	BYTE (b bits)
-----------------	-----------------------	------------------	------------------

Figura 6.17: struttura dell'indirizzo di memoria

Le formule per il dimensionamento degli indirizzi nella mappatura diretta sono le seguenti:

- Lunghezza indirizzo: $(s + w)$ bits;
- Parole indirizzabili: $2^{(s+w)}$ words;
- Dimensione blocco = dimensione linea: 2^w words;
- Numero di blocchi nella memoria primaria: $\frac{2^{(s+w)}}{2^w} = 2^s$;
- Numero di linee/slot della Cache: $2^r = m$;
- Dimensione della Cache: $2^{(r+w)}$ words;
- Dimensione del tag: $(s - r)$ bits.

Esercizio d'esempio:

Si ha una Cache da 64KB con linee (slot) di Cache da 4 Byte. Invece, la memoria primaria è da 16MB e contiene blocchi da 4Byte ciascuno in cui le word sono da 1 byte.

Obiettivo: Trovare quanto è lunga la striscia totale (n bit), quanti bit servono per la word (w bit), quanti bit servono per gli indici/slot della Cache (r bit) e infine quanti bit servono per il tag (t bit).

Nota Bene

Il campo "byte" non viene preso in considerazione poiché l'esercizio specifica una condizione semplificata, ovvero "*word da 1 byte*", in questo caso i campi "word" e "byte" sono la stessa cosa.

- **Passo 1: calcolare la lunghezza totale dell'indirizzo (n)**

La RAM è da 16MB: 16 sono 2^4 bit, mentre un MB sono 2^{20} bit $\Rightarrow 16MB = 2^4 \cdot 2^{20} = 2^{24}$ Byte. Ciò vuol dire che per ogni indirizzo sono necessari 24 bit = n .

- **Passo 2: calcolare i bit per la word (w)**

Dato che i blocchi in ram sono grandi 4 Byte ciascuno, e abbiamo word grandi 1 Byte, ogni blocco è composto da 4 word: per indirizzare 4 word servono solamente 2 bit.

- **Passo 3: calcolare i bit per gli indici/slot (r)**

Per capire quanti bit servono per r indici bisogna prima capire quanti slot ha la Cache.

Per trovare il numero di slot dobbiamo dividere la grandezza totale della Cache (64KB) per la grandezza del singolo slot (4 Byte): $64KB = 2^6 \cdot 2^{10}$, mentre 4 Byte = 2^2 .

Il numero di slot è quindi dato da: $2^{16}/2^2 = 2^{14} = 16.384$ linee/slot di Cache.

Dal calcolo si vede che, per indirizzare queste 16.384 linee, servono proprio 14 bit.

- **Passo 4: calcolare i bit per il tag (t)**

Il tag è semplicemente "tutto quello che avanza": $24 - 2 - 14 = 8$ bit

Linea di Cache	Indirizzo di inizio del blocco di memoria primaria allocato alla linea di cache
0	000000, 010000,.....,FF0000
1	000004, 010004,.....,FF0004
.	
.	
$m - 1$	00FFFC, 01FFFC,.....,FFFFFC

TAG (8 bits)	SLOT/LINE (14 bits)	WORD (2 bits)
--------------	---------------------	---------------

Figura 6.18: struttura della Cache

Ovviamente, il calcolatore lavora con una **logica operativa** differente per arrivare allo stesso risultato. Normalmente la CPU fornisce l'indirizzo di memoria (n): se l'architettura è stata creata per avere word da 1 byte e blocchi di RAM da 4 word, il calcolatore saprà che serviranno 2 bit per le word.

A questo punto il calcolatore per trovare le informazioni mancanti utilizza le **operazioni di shift** (come mostrato in figura 6.19), infatti, far scorrere l'indirizzo a destra di w posizioni equivale a dividere per 2^w . In questo modo si trova l'**indirizzo del blocco** ($s = t + r$, che è tutto il resto a sinistra), "tagliando" quei w bit (2 nel caso dell'esempio precedente).

Esempio: Se l'indirizzo è 11101 (binario per 29) e la word sono 2 bit:

- Taglio gli ultimi due (01): resta 111 (binario per 7);

- $29/4 = 7$ (con resto 1);
- Quindi: Il blocco è il 7, la word è il resto (1).

Lo stesso procedimento viene effettuato partendo dall'indirizzo del blocco (s) appena calcolato per separare il tag dall'indice di linea. Il tag viene trovato scorrendo l'indirizzo del blocco a destra di r posizioni (divisione per 2^r): quello che rimane è il tag, mentre il resto della divisione è l'indirizzo di linea della Cache.

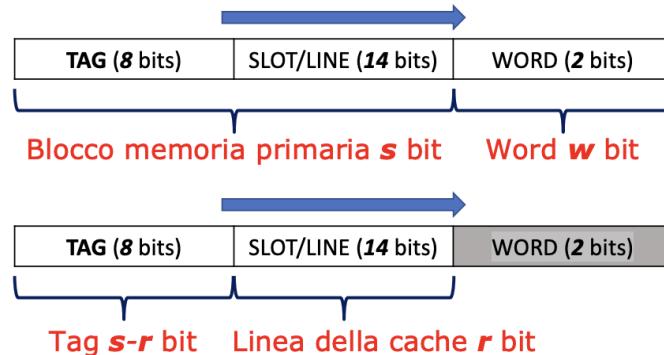


Figura 6.19: shift a destra (divisione)

La figura 6.20 mostra il principio della **mappatura diretta**: ogni blocco di memoria ha un unico posto fisso (linea) nella cache dove può andare. Le frecce mostrano che blocchi diversi della RAM (se hanno lo stesso finale di indirizzo) competono per quella stessa unica linea, creando potenziali conflitti.

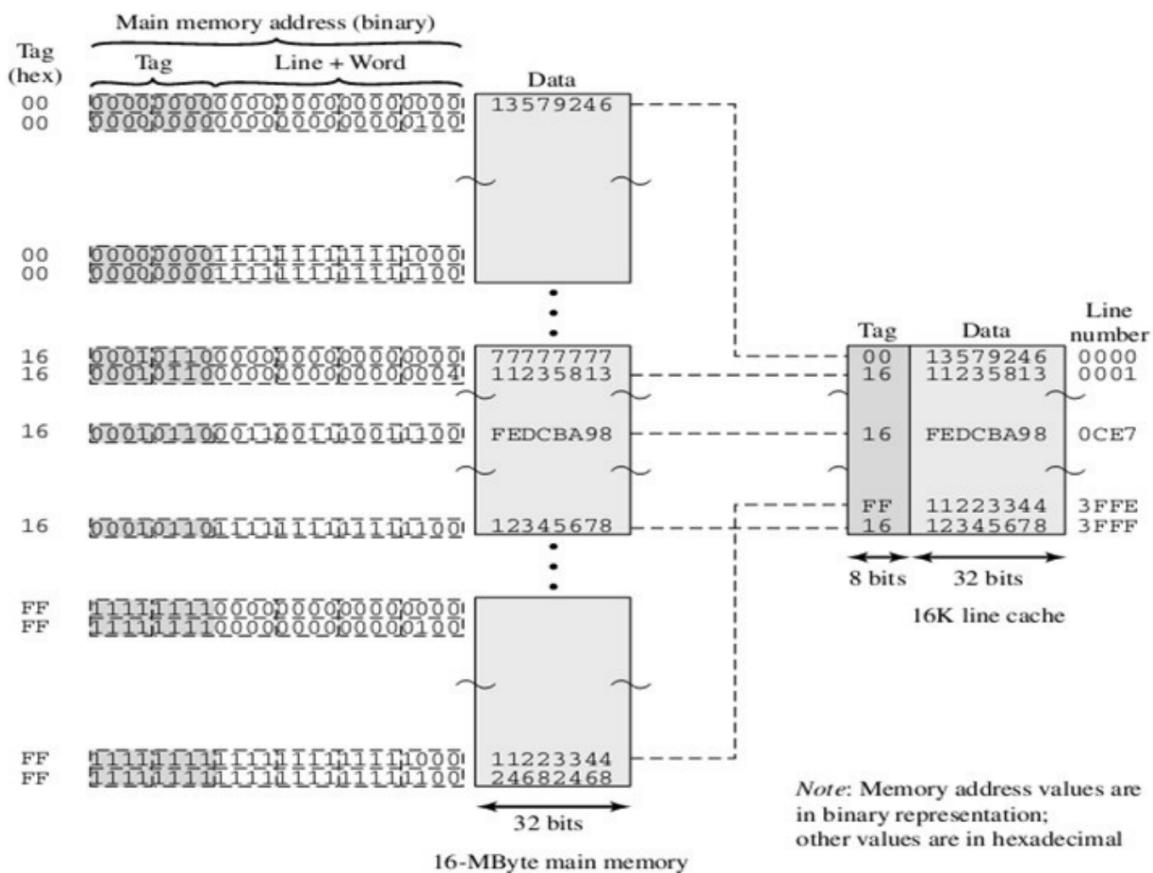


Figura 6.20: rappresentazione della mappatura diretta

Invece, nella figura 6.21, viene mostrato cosa succede fisicamente dentro la CPU quando arriva l'indirizzo (Tag 8, Line 14, Word 2):

1. L'indirizzo arriva dall'alto (indirizzo a 24 bit);
2. I fili si separano:
 - **Centro (r bit = 14 bit)**: questo filo va al selettore delle righe, dicendo alla Cache "apri lo slot numero X";
 - **Sinistra ($s - r$ bit = 8 bit)**: questi sono i bit del Tag. Vanno dentro una scatolaletta chiamata Compare (Comparatore).
3. Il confronto:
 - Dallo slot selezionato esce il tag memorizzato;
 - Il **comparatore** controlla se il tag appena letto dalla Cache è uguale al tag dell'indirizzo cercato.
4. Esito:
 - Se sono uguali → Cache hit. Il dato esce verso la CPU;
 - Se sono diversi → Cache miss. Bisogna andare in RAM a cercare il dato.

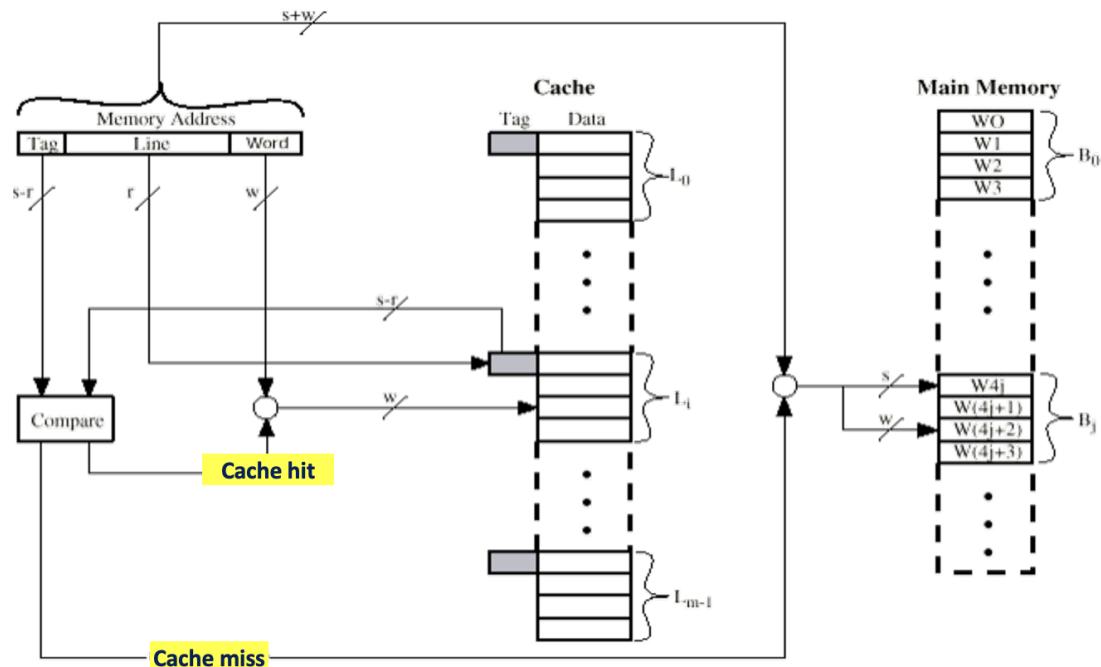


Figura 6.21: circuito hardware per mappatura diretta

Ovviamente, utilizzare una mappatura di questo tipo comporta dei vantaggi e degli svantaggi:

- **Vantaggi**: questa mappatura è semplice e poco costosa da implementare, poiché è necessario un solo comparatore;
- **Svantaggi**: si ha un'**associazione fissa** (rigida) di una linea di Cache ad un blocco di memoria, inoltre se due blocchi usati spesso finiscono nello stesso slot, la Cache continua a sovrascriversi (fenomeno del **thrashing**) riducendo notevolmente l'**hit ratio**.

Mappatura associativa della Cache

Utilizzando una mappatura associativa della Cache **viene rimosso il vincolo** che costringeva ad avere il blocco di memoria primaria in una determinata linea/slot di Cache.

Per poter ottenere questo tipo di implementazione, l'indirizzo di memoria primaria non ha più il campo "line" (perché non c'è una riga fissa), ed è formato solamente dal **tag** e **word**.

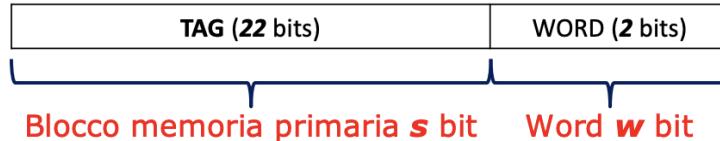


Figura 6.22: struttura dell'indirizzo di memoria

A questo punto, se il dato può essere ovunque, la **ricerca in Cache diventa più complessa**, poiché invece che effettuare un solo confronto (Tag Cache con Tag indirizzo), è necessario confrontare il Tag dell'indirizzo del blocco di memoria con **tutti i Tag** di linea della Cache in parallelo.

Come si può vedere in figura 6.23 l'indirizzo ("memory address") non contiene più tre campi, ma solamente due (tag e word):

1. **Indirizzo in ingresso:** arriva sempre dall'alto, ma con meno campi;
2. **Ingresso nel comparatore:**
 - Dall'alto entra il Tag dell'indirizzo generato dalla CPU;
 - Da destra entrano simultaneamente tutti i Tag memorizzati nelle varie linee della cache (le frecce che partono dai rettangoli grigi "Tag" di ogni linea e vanno verso sinistra nel comparatore);
 - Il circuito non seleziona una riga specifica prima. Invece, porta tutti i tag presenti in cache dentro il comparatore per confrontarli contemporaneamente con il tag cercato.

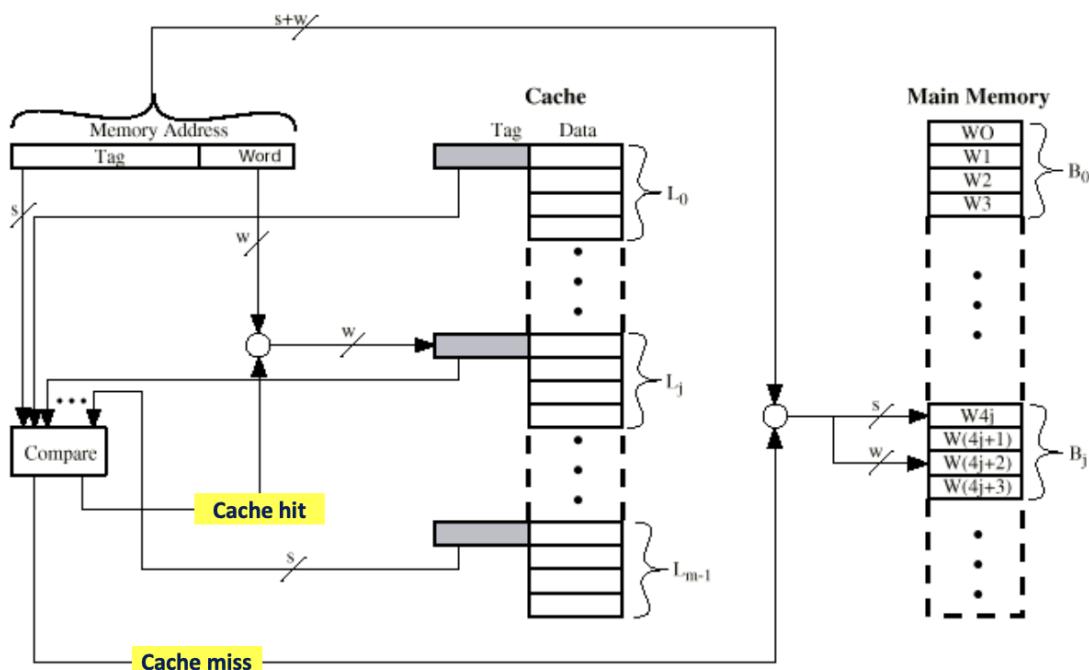


Figura 6.23: circuito hardware per mappatura associativa

3. La Logica di Confronto (Ricerca Parallelia): Il blocco "Compare" effettua un controllo simultaneo su tutte le linee, e poiché non esiste un indice, l'hardware deve cercare ovunque nello stesso istante (questo richiede una circuiteria più complessa e costosa rispetto alla mappatura diretta);

4. L'Esito (Hit/Miss e Selezione del Dato):

- Se c'è corrispondenza il comparatore attiva il segnale Cache Hit e abilita l'uscita della specifica linea che ha dato esito positivo;
- Solo a questo punto interviene il campo Word (w) dell'indirizzo (il filo che scende dritto al centro). Una volta individuato il blocco giusto, questi bit servono a selezionare la singola parola desiderata all'interno del blocco di dati uscito.

Anche in questo caso, per ottenere il tag dall'indirizzo di memoria, si utilizza un'**operazione di shift a destra** (divisione). Considerando di avere un indirizzo da 24 bit di cui 22 sono dedicati al tag (t bit) e 2 alla word (w bit) (figura 6.24), si effetta una divisione per 2^w . In questo modo i bit rimanenti faranno riferimento al tag dell'indirizzo di memoria, mentre il resto della divisione saranno i bit utilizzati per la word.

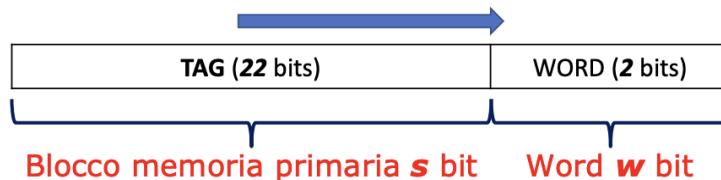


Figura 6.24: ottenimento del tag, shift a destra

L'immagine in figura 6.25 mostra che nella mappatura associativa l'indirizzo viene scisso solo in Tag e Word. Il Tag (molto più lungo rispetto alla mappatura diretta) viene salvato, insieme al dato, in una posizione qualsiasi della cache. L'esempio mostra visivamente che il blocco 16339C viene identificato dal Tag 058CE7 e "parcheggiato" arbitrariamente alla linea 0001 (1).

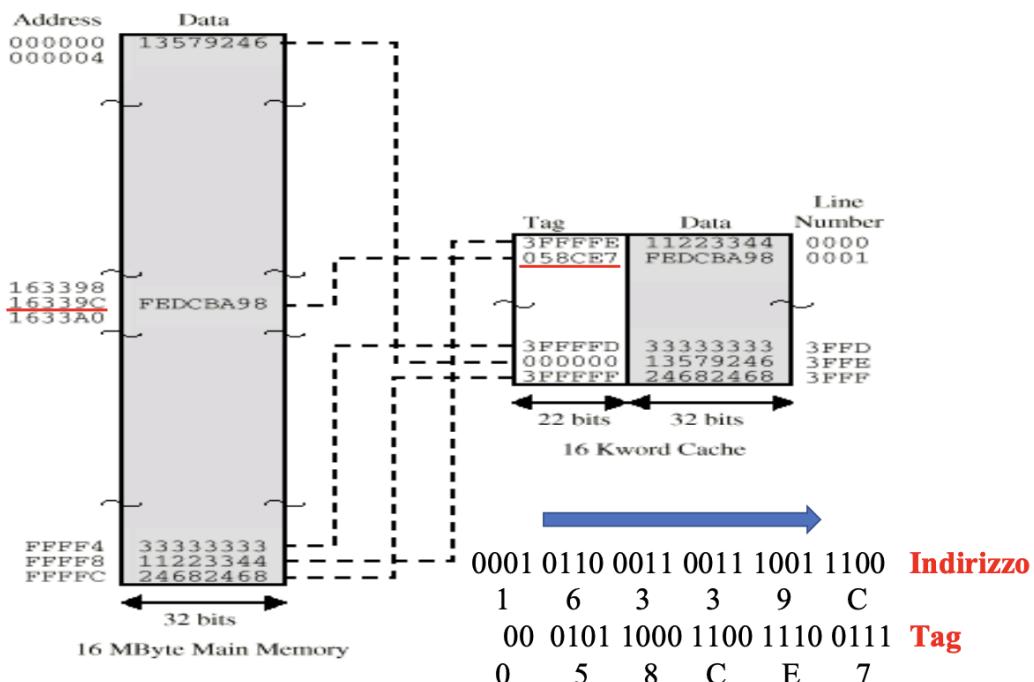


Figura 6.25: rappresentazione della mappatura associativa

Anche la mappatura associativa porta con se alcuni vantaggi e svantaggi:

- **Vantaggi:** si ottiene una grande flessibilità nell'allocazione dei blocchi nella Cache;
- **Svantaggi:** aumenta la complessità del circuito richiesto (comparatore) per confrontare i tag di tutte le linee della cache, in parallelo.

Mappatura associativa su insiemi della Cache

La mappatura associativa su insiemi **rappresenta un compromesso** tra i due metodi precedenti: non si è obbligati ad allocare un blocco di memoria in una sola linea/slot di Cache e non è nemmeno necessario confrontare tutti i campi tag della Cache.

La Cache viene **divisa** in un numero v di **insiemi (Sets)**, ognuno dei quali è **composto** da k **linee di Cache**. Ad esempio: "Il blocco B può essere mappato in una qualsiasi linea del set i".

Per la mappatura viene quindi utilizzata una **logica ibrida**:

- **Come la mappatura diretta:** il blocco di RAM deve andare in uno specifico Insieme (calcolato con il modulo);
- **Come la mappatura associativa:** una volta arrivato nell'insieme giusto, può occupare una qualsiasi delle linee disponibili in quell'insieme (le "vie").

Nota Bene

Questo significa che vi è un'**associazione rigida** blocco-insieme, ma all'interno dello stesso insieme un blocco può trovare differenti allocazioni (linee/slot).

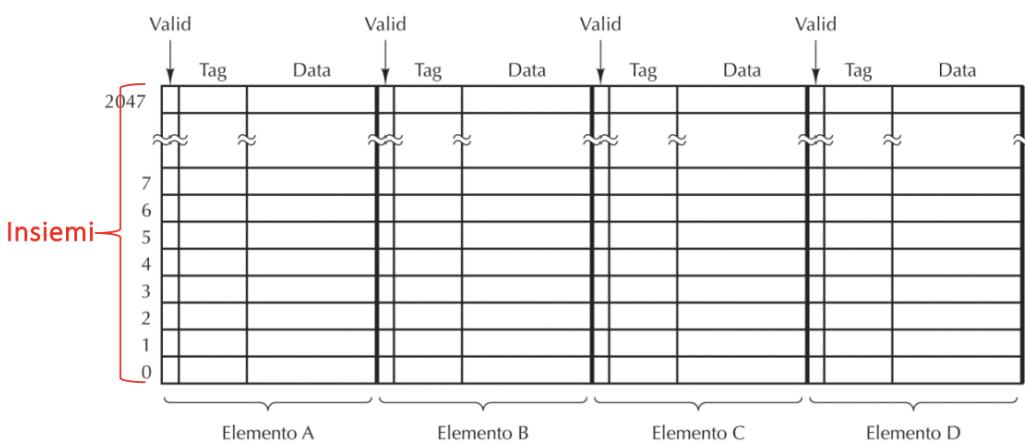


Figura 6.26: mappatura associativa su insiemi a quattro vie

Nota Bene

Una Cache associativa a N vie indica che ogni insieme (Set) è composta da N linee di Cache.

In figura 6.26 ogni riga è da considerare un Set, mentre gli elementi (A, B, ...) sono le linee all'interno del Set.

Una **situazione tipica** è una Cache con 2 linee/slot per ogni insieme (Set), in questo modo si hanno due possibili mappature associative: così facendo, un blocco di memoria primaria può essere posizionato in una delle due linee del Set.

Ovviamente, anche in questo caso, l'indirizzo di memoria viene modificato in base alla logica utilizzata per il sistema di mappatura. Rimangono i campi **tag** e **word**, ma se ne aggiunge un terzo, "Set", il quale:

1. Identifica la porzione di Cache su cui procedere;
2. Tramite l'utilizzo dei comparatori, viene confrontato il campo **tag** per verificare se il blocco di memoria richiesto sia in quel Set.

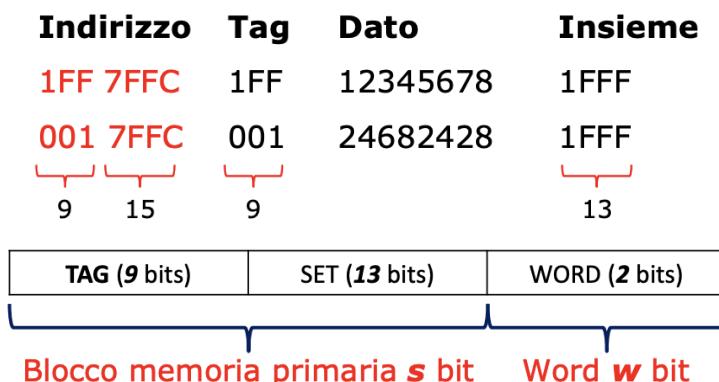


Figura 6.27: struttura dell'indirizzo di memoria

In figura 6.28 si può vedere il circuito ibrido che unisce la velocità della mappatura diretta con la flessibilità della mappatura associativa.

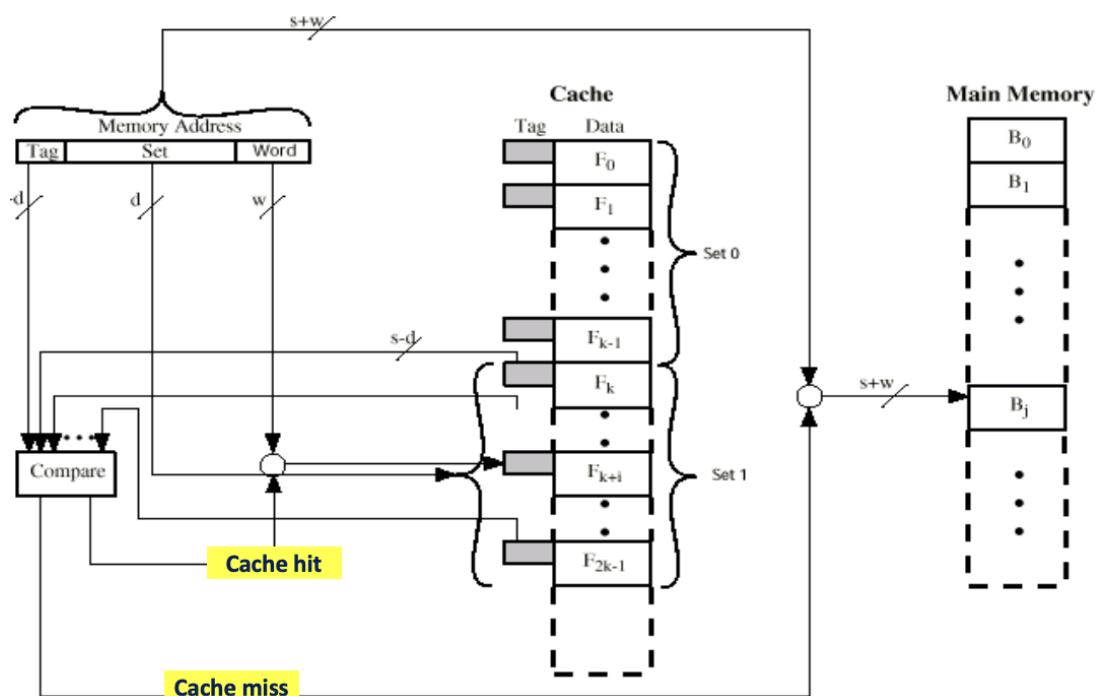


Figura 6.28: circuito hardware per la mappatura associativa su insiemi

1. **L'indirizzo (il ritorno dell'indice):** guardando in alto a sinistra si può vedere come l'indirizzo è tornato ad essere suddiviso in tre parti, ma con la differenza che il campo centrale rappresenta i bit utilizzati per il Set, al posto dei bit che venivano utilizzati per l'indirizzamento della linea di Cache;

2. **La selezione del Set:** seguendo il filo che scende dal campo **Set** si vede che

- Questo filo funziona esattamente come nella mappatura diretta, infatti punta dritto a un gruppo specifico di righe;
- Nell'immagine, il filo seleziona il "Set 1";
- **Non seleziona una singola riga**, ma seleziona l'intero pacchetto (Set). Se è una cache a 2 vie, seleziona 2 righe insieme; se a 4 vie, ne seleziona 4, e così via...

3. **Il confronto ristretto:** focalizzandosi sul blocco "compare" si vede che

- Dall'alto arriva il tag cercato dalla CPU;
- Da destra arrivano i tag **solo** dalle linee contenute nel Set 1 (le frecce che escono da F_k, \dots, F_{2k-1}).
- Nella mappatura diretta si effettuava un solo confronto, nella mappatura associativa confrontava con tutta la Cache mentre **in questo caso si confronta il tag cercato con i k tag presenti dentro il Set selezionato.**

4. **L'esito:**

- Il comparatore controlla parallelamente solo quelle poche vie (ad esempio 4);
- Se trova corrispondenza all'interno di quel determinato Set si attiva il **Cache hit**;
- A quel punto, il multiplexer (il cerchietto) lascia passare il dato corretto e il campo Word seleziona il pezzetto finale.

Dai capitoli precedenti abbiamo visto come il calcolatore, per ricavare i vari campi dell'indirizzo di memoria, utilizzi le operazioni di shift verso destra (le quali equivalgono ad una divisione). Conoscendone ormai il funzionamento (continuando a ipotizzare un indirizzo di memoria a 24 bit), possiamo dire che :

- Per trovare l'indirizzo del blocco di memoria (composto da bit del tag + bit del Set) viene effettuato uno shift verso destra di w bit (divisione per 2^w);
- Invece, per trovare i bit del tag si effettua un shift verso destra di r bit (divisione per 2^r). Il risultato della divisione sono gli $s - r$ bit del tag, mentre il resto sono gli r bit che identificano il Set.

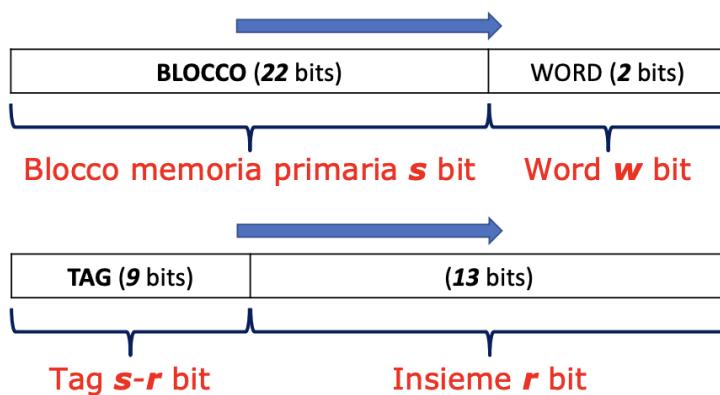


Figura 6.29: struttura dell'indirizzo di memoria

Infine, nell'immagine in figura 6.30 viene mostrata una Cache associativa a 2 vie (ogni riga ha 2 spazi). Si vede chiaramente che l'indirizzo seleziona un Set specifico (es. 0000), ma all'interno di quel Set il dato può finire in una qualsiasi delle due vie disponibili. Questo permette a due blocchi diversi (es. Tag 000 e Tag 001) di convivere nello stesso Set 0000 senza buttarsi fuori a vicenda (cosa che invece succedeva nella mappatura diretta).

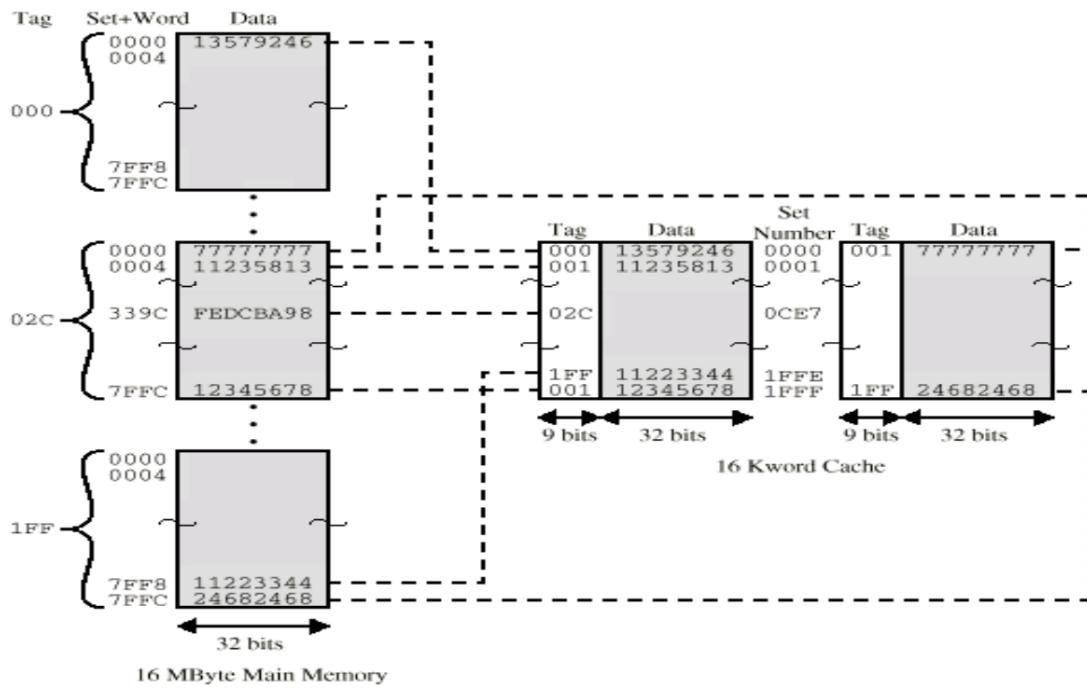


Figura 6.30: rappresentazione della mappatura associativa su insiemi (a 2 vie)

Considerazioni finali sulle tecniche di mappatura

Il grafico in figura 6.31 mette in confronto tra di loro le varie metodologie utilizzate per la mappatura della Cache. Tramite di esso è possibile comprendere alcune sfumature importanti:

- **Rendimenti decrescenti:** il grafico mostra che passare da "Direct" a "2-way" (associativa a 2 vie) riduce drasticamente i Cache Misses, ma andare oltre (4-way, 8-way) riduce i miss di pochissimo. Questo significa che aumentare troppo la complessità hardware (più comparatori) non vale più la pena rispetto al guadagno minimo di prestazioni;
- **Il fattore dimensione:** guardando la parte destra del grafico (16M), si nota come le barre si assomigliano molto di più rispetto alla parte sinistra (512k). Ciò significa che, se la cache è molto grande, la differenza tra mappatura diretta e associativa diventa trascurabile (perché è talmente grande che i conflitti avvengono raramente comunque).

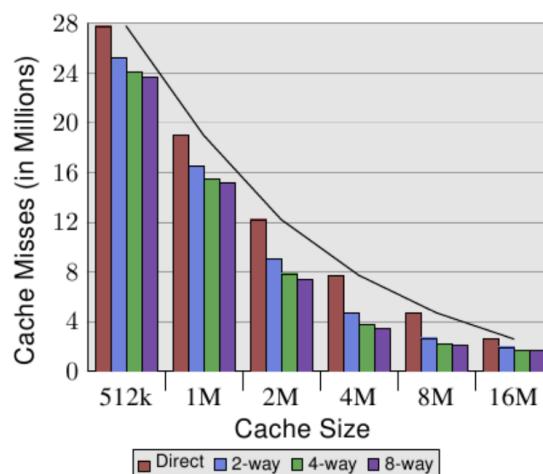


Figura 6.31: grafico comparativo delle mappature

6.5.6 Algoritmi di rimpiazzamento

Nel contesto delle memorie Cache, un argomento importante da tenere in considerazione è quello degli **algoritmi di rimpiazzamento**, che cercano di rispondere alla domanda: "Quando la cache è piena (o l'insieme è pieno) e avviene un Cache miss, chi butta fuori per essere sostituito con il blocco richiesto?".

Dunque, vengono utilizzati nel caso in cui la Cache sia **totalmente occupata**: se un ulteriore blocco deve essere caricato **se ne sceglie un altro da eliminare dalla Cache**.

Metodo diretto

Come è stato spiegato al capitolo 6.5, l'utilizzo di una mappatura diretta è limitante poiché **la scelta è obbligata**: ogni blocco può essere sistemato solo in una determinata linea di Cache che viene indicata all'interno dell'indirizzo di memoria.

Ciò significa che **si rimpiazza il blocco presente nella linea di Cache corrispondente con il nuovo blocco in ingresso** alla Cache.

Metodo associativo e associativo su insiemi

Se viene utilizzata una tecnologia di mappatura basata su metodo associativo o associativo su insiemi, la **politica di rimpiazzamento** permette **maggior flessibilità**.

Infatti, grazie alla possibilità di inserimento del blocco in qualsiasi linea di Cache, queste map-pature permettono di utilizzare 4 differenti algoritmi che vengono **implementati direttamente su hardware** per ottenere una **maggior velocità**:

- **Last Recently Used (LRU)**: questo algoritmo viene utilizzato principalmente con una mappatura associativa su insiemi (ad esempio, con un Set da 2 linee di Cache).

All'interno di ogni Set viene rimpiazzato il blocco nella linea **non utilizzata da più tempo**, tramite l'utilizzo di un bit aggiuntivo, infatti:

- Ogni linea del set presenta un bit che può essere settato a 0/1;
- La linea da rimpiazzare sarà quella con il bit a 0. Una volta rimpiazzata il bit torna ad 1 per indicare che il blocco è stato sostituito da poco tempo.

Viene considerato il metodo più **efficiente** proprio grazie al **principio di località temporale** (capitolo 6.4). Statisticamente, se un programma ha usato un dato poco fa, è molto probabile che lo userà ancora a breve. Al contrario, se un dato non viene toccato da molto tempo, probabilmente non serve più. LRU è l'approssimazione più vicina all'algoritmo Ottimo (che scarterebbe il dato che verrà usato "più in là nel futuro").

Problematica dell'algoritmo LRU: l'algoritmo LRU si basa sull'assunzione che "Se un dato è stato usato ORA, sarà riusato PRESTO (Recenza)". Tuttavia, viene completamente ignorata la **frequenza di utilizzo**.

Quando viene letto un file enorme (ad esempio un film in 4K, più grande della Cache) ogni blocco del file è nuovo, appena usato, e l'algoritmo LRU lo considera importantissimo mettendolo all'inizio della lista. A causa di questo vengono spinti fuori dalla Cache i dati che erano lì in modo stabile. Successivamente, se i nuovi dati non vengono più acceduti, si finisce con l'aver riempito la Cache di dati "inutili" (usati una sola volta) perdendo i dati "utili" (usati spesso).

- **First In First Out (FIFO)**: si rimpiazza il blocco presente in Cache da più tempo;
- **Last Frequently Used**: si rimpiazza il blocco utilizzato meno di frequente;

- **Random replacement:** come dice il nome, viene effettuata una scelta a caso;
- **Low Inter-reference Recency Set (LIRS):** è un algoritmo avanzato che cerca di risolvere i limiti di LRU (come quando una scansione di un file enorme cancella dati utili dalla cache). LIRS risolve questo problema misurando non solo **quando è stato usato** il dato l'ultima volta (Recency), ma **quanto tempo passa tra due utilizzi successivi** (Inter-Reference Recency). Per implementare questo algoritmo i dati vengono divisi in **due classi**:
 - **HIR (High IRR):** dati "freddi" o usati sporadicamente, candidati all'eliminazione. Se un dato viene usato una volta sola (il film), non ha una "seconda volta", quindi LIRS capisce che non è stabile e lo tiene nella zona "fredda" (HIR), pronto a essere scartato subito;
 - **LIR (Low IRR):** dati "caldi" e stabili, da mantenere. Per entrare nella zona "protetta" (LIR), un dato deve dimostrare di essere stato richiesto almeno due volte in un arco di tempo breve;

Il grafico in figura 6.32 mostra il confronto delle prestazioni (Hit Ratio) al variare della dimensione della Cache. LIRS (linea blu) ha un Hit Ratio molto più alto di LRU (linea rossa) e si avvicina molto all'ideale (OPT).

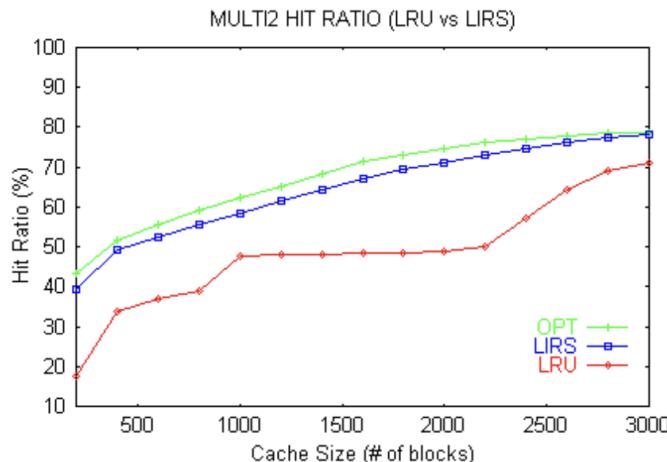


Figura 6.32: confronto tra algoritmo LIRS e LRU

- **OPT (Verde):** L'algoritmo ideale, impossibile nella realtà perché si dovrebbe sapere quale programma non verrà mai utilizzato (un po come prevedere il futuro);
- **LRU (Rosso):** algoritmo classico
- **LIRS (Blu):** algoritmo avanzato.

6.5.7 Politiche di scrittura

Una volta deciso quale blocco rimpiazzare, sorge un problema fondamentale: **se il blocco in Cache è stato modificato dalla CPU, è diverso da quello in RAM**. Non basta semplicemente sovrascriverlo con il nuovo dato in arrivo, altrimenti le modifiche andrebbero perse.

Nota Bene

Negli algoritmi di rimpiazzamento, la regola fondamentale è che **non si deve mai sovrascrivere un blocco presente in cache** (se è stato modificato dalla CPU) **prima di aver aggiornato la memoria principale**.

Il problema sorge perché la memoria primaria non è accessibile solo dalla CPU, ma anche da altri attori. Dunque, nasce un problema legato alla **coerenza della memoria**:

- **Multiprocessore:** Possono accedere alla stessa memoria primaria **più processori**, ciascuno dotato della propria Cache locale. Se la CPU A modifica un dato nella sua Cache, la CPU B potrebbe leggere dalla RAM il dato vecchio.
- **DMA (Direct Memory Access):** Alcune periferiche (schede video, dischi, schede di rete) hanno **l'accesso diretto alla memoria principale**.

Se una periferica DMA (o un altro processore) modifica un dato direttamente in RAM mentre la CPU possiede una copia di quel dato in Cache, la copia in Cache diventa obsoleta ("stale"). Per **garantire la coerenza**, il sistema hardware deve essere in grado di rilevare queste modifiche esterne e **invalidare** immediatamente il blocco corrispondente nella Cache della CPU, costringendola a rileggerlo aggiornato dalla memoria principale.

Vengono quindi implementate delle **politiche di scrittura** che si occupano di far fronte a questi problemi per gestire al meglio le operazioni di scrittura e soprattutto per **capire quando aggiornare la RAM**:

- **Politica Write Through:** utilizzando una metodologia di questo tipo, i dati vengono scritti contemporaneamente sia sulla Cache che sulla RAM.
In questo modo, i dati saranno **sempre sicuri ed allineati**, ma verrà generato **molto traffico** verso la RAM con possibilità di un collo di bottiglia;
- **Politica Write Back:** questa metodologia prevede che ogni linea di Cache abbia un bit aggiuntivo ("dirty bit"), che indica se sia necessario aggiornare il corrispondente blocco in memoria. Inizialmente la **scrittura** viene effettuata solo all'interno della Cache.
Si aggiorna un blocco in memoria principale, solo se il bit vale 1, quando tale blocco va rimpiazzato in Cache.
Ovviamente questa procedura risulta molto più veloce, ma di contro la gestione diventa più complessa.

6.5.8 Dimensione della linea

Un altro parametro fondamentale nella progettazione di una Cache è la **dimensione del blocco** (o linea). La scelta della grandezza non è arbitraria, ma deve rispettare un delicato equilibrio per massimizzare le prestazioni.

- **Linee più grandi:** Aumentare la dimensione della linea aiuta a sfruttare meglio il **princípio di località spaziale**. Se la CPU richiede una parola, è molto probabile che richieda anche le parole adiacenti; se la linea è grande, queste vengono caricate tutte insieme in un colpo solo, aumentando gli Hit futuri.
- **Linee troppo grandi:** Tuttavia, esagerare porta a due effetti negativi (contro-producenti):
 1. **Pochi blocchi disponibili:** Se la cache ha una dimensione fissa, fare linee enormi significa averne poche. Questo aumenta la probabilità di conflitti (più blocchi di RAM competono per quelle poche linee), costringendo a rimpiazzamenti frequenti.
 2. **Aumento del tempo di trasferimento:** Caricare una linea gigante dalla memoria principale richiede più tempo (Miss Penalty più alto).
 3. **Perdita di località:** Se la linea è troppo lunga, le parole finali potrebbero essere così distanti da quelle iniziali da non essere mai richieste dal programma, sprecando spazio prezioso.

In conclusione, il **valore ottimale** dipende dal tipo di processo in esecuzione, ma i valori tipici riscontrati nelle architetture reali variano **da 2 a 8 parole** (word) per linea.

6.5.9 Uso di più Cache

L'evoluzione tecnologica ha portato a spostare la Cache dalla scheda madre direttamente all'interno del chip del processore, principalmente per le seguenti motivazioni:

1. **Bisogno di velocità:** si vuole che la Cache sia dentro il chip del processore (**livello interno**) perché così i segnali elettrici devono fare pochissima strada. **Meno distanza percorsa** dai segnali elettrici implica **più velocità di accesso** ai dati nella Cache;
2. **Il problema dello spazio:** lo spazio fisico sul silicio del processore è limitatissimo e costoso. Non è possibile inserire una Cache gigantesca, altrimenti il chip diventerebbe enorme, costoso e scalderebbe troppo;
3. **Compromesso:** quindi la Cache interna (L1) deve essere per forza piccola (per starci fisicamente) e velocissima;
4. **La conseguenza (I livelli):** ma con la Cache L1 piccola, si riempirebbe subito. Se si riempie, si avranno tanti "Cache miss". Per evitare di andare fino alla "lenta" RAM ogni volta che la L1 è piena, si aggiunge una seconda Cache (L2), che è un po' più grande e un po' più lenta, la quale viene posizionata "appena fuori" dal core o in una zona meno pregiata del chip. Successivamente una L3 ancora più grande aggiunta nel **livello esterno**, che contribuisce ad **aumentare le prestazioni** del sistema.

Nota Bene

Dunque, la **creazione di una gerarchia** di questo tipo serve per **bilanciare velocità e costi**.

Cache Unificata vs Separata (Split Cache)

Nei sistemi a singola cache o nel primo livello (L1) dei sistemi moderni, si adotta spesso un'architettura che prevede la separazione fisica della cache in due parti:

- **I-Cache (Instruction Cache):** Dedicata esclusivamente al salvataggio delle istruzioni del programma.
- **D-Cache (Data Cache):** Dedicata esclusivamente ai dati su cui il programma lavora.

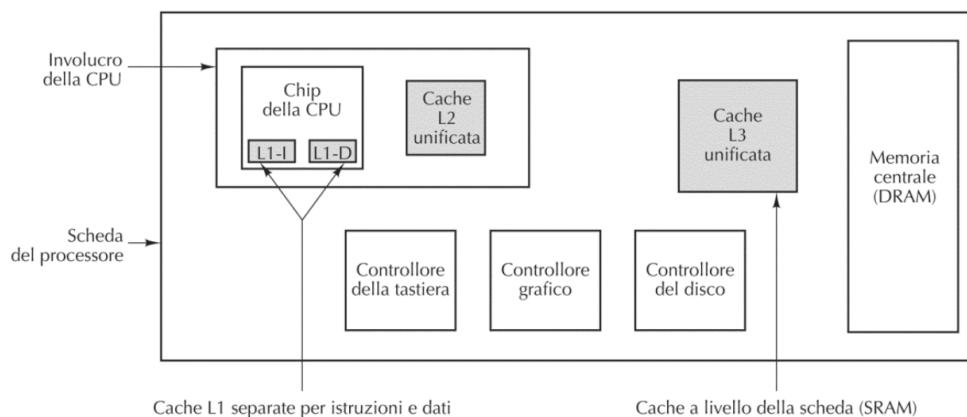


Figura 6.33: Sistema con tre livelli di Cache

Questa separazione è fondamentale nelle architetture con pipeline, poiché permette alla CPU di prelevare un'istruzione e un dato **nello stesso ciclo di clock**, raddoppiando di fatto la banda passante verso la memoria.

Organizzazione in sistemi Multi-Core

Come mostrato in figura 6.34, nei moderni processori (ad esempio, un Intel Core i7), la gerarchia della cache assume una struttura specifica per supportare più Core:

- **Cache Private (Dedicated):** La L1 e la L2 sono spesso dedicate al **singolo Core**. Questo garantisce che ogni processore abbia i suoi dati "caldi" immediatamente disponibili senza interferenze dagli altri.
- **Cache Condivisa (Shared):** La L3 (molto più grande, ad esempio "Shared L3 Cache" nell'immagine) è invece condivisa tra **tutti i Core**. Essa agisce come punto di incontro e scambio dati tra i processori, oltre a fungere da "Victim Cache" per raccogliere i dati scartati dalle L2 private.

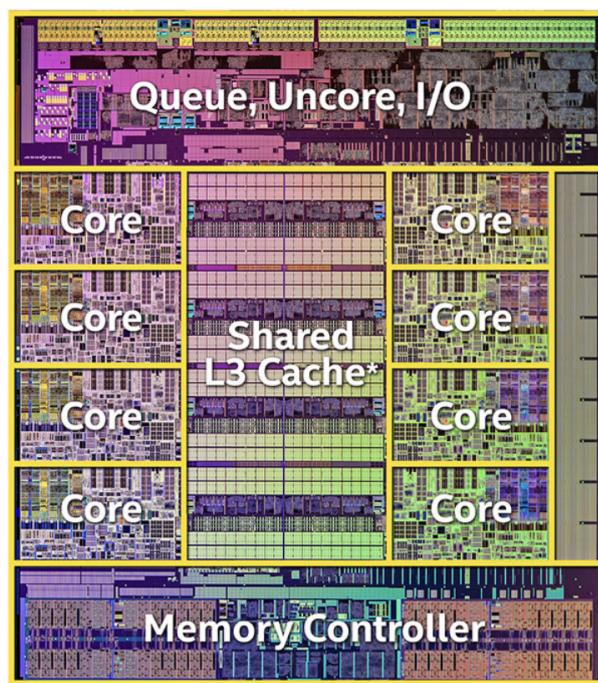


Figura 6.34: Layout reale (die shot) e schema a blocchi di un Intel Core i7

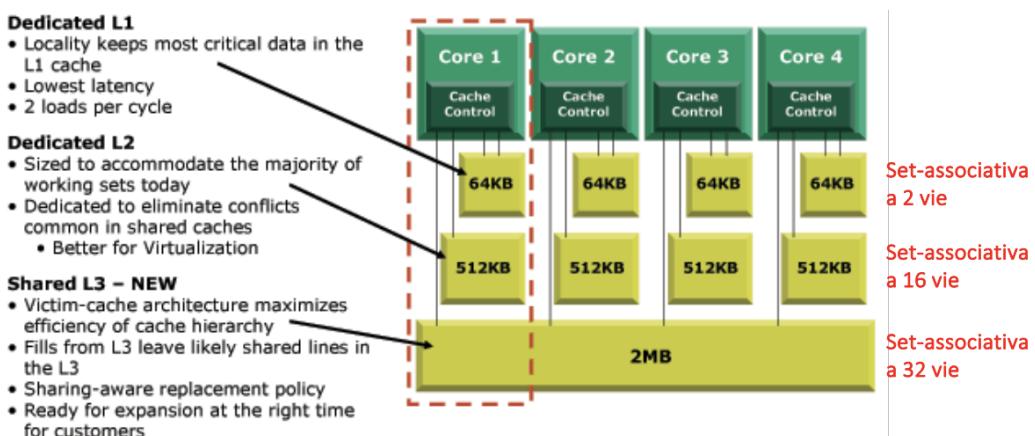


Figura 6.35: Posizionamento logico dei livelli della Cache

6.5.10 Tempo di acceso e hit ratio con più Cache

Dopo aver descritto l'architettura a più livelli, è fondamentale analizzare matematicamente come questa influenzi il **Tempo Medio di Accesso** alla memoria. L'obiettivo è dimostrare che l'aggiunta di cache riduce drasticamente il tempo medio, avvicinandolo a quello della cache più veloce.

Tempo medio con un solo livello

Considerando un sistema semplice con Memoria Primaria e un solo livello di Cache, il tempo medio di accesso (T_s) è dato dalla somma del tempo di accesso alla cache più l'eventuale penalità in caso di Miss:

$$T_s = T_1 + (1 - H)T_2$$

Dove:

- T_1 : Tempo di accesso alla Cache (molto basso);
- T_2 : Tempo di accesso alla Memoria Primaria (alto);
- H (**Hit Ratio**): Percentuale di successi (probabilità che il dato sia in cache);
- $(1 - H)$ (**Miss Rate**): Percentuale di insuccessi.

Il grafico n figura 6.36 mostra chiaramente che all'aumentare della dimensione della cache e sfruttando una **Strong Locality** (Princípio di Località), il valore di H tende a 1, rendendo il termine $(1 - H)T_2$ trascurabile. Di conseguenza, il tempo medio T_s si avvicina a T_1 .

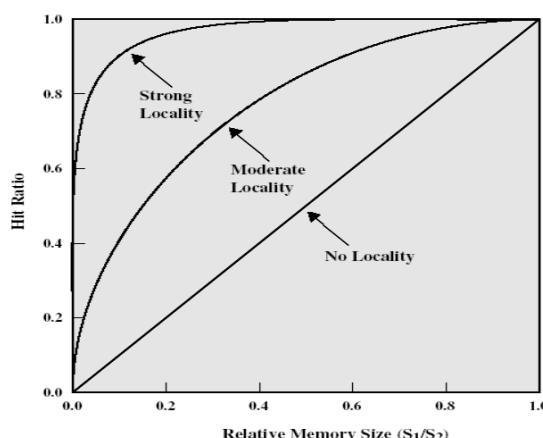


Figura 6.36: Hit ratio vs. Memory size

Gerarchia a più livelli (Multi-level)

In una gerarchia multilivello (L_1, L_2, L_3, \dots), i dati contenuti al livello i sono un sottoinsieme dei dati contenuti al livello inferiore $i + 1$. È necessario definire due tipi di Hit Ratio:

- **Hit Ratio Globale (H_i)**: La probabilità di trovare un dato entro il livello i (frequenza di successi cumulativa).
- **Hit Ratio Condizionale ($H_{i|i-1}$)**: La probabilità di trovare una parola al livello i , *dato che non è stata trovata* al livello precedente ($i - 1$).

La formula esplicita per ricavare l'Hit Ratio condizionale, basata sulla probabilità condizionata, è la seguente:

$$H_{i|i-1} = \frac{P(i \cap \overline{i-1})}{P(\overline{i-1})} = \frac{H_i - H_{i-1}}{1 - H_{i-1}}$$

Questa relazione ci dice che la probabilità di successo nel livello corrente dipende dai successi globali meno quelli già ottenuti nei livelli precedenti, normalizzati sulla probabilità di aver fallito nel livello precedente.

Calcolo del Tempo Medio per gerarchie a 2 e 3 livelli

Analizziamo ora come si compone il tempo totale (T) espandendo la formula passo dopo passo. Indichiamo con t_i il tempo medio di accesso al livello i .

- **Gerarchia a due livelli:** nel caso semplice (L1 + RAM), il tempo totale è dato dalla somma del tempo in caso di successo in L1 (H_0) più il tempo in caso di insuccesso ($1 - H_0$), il quale richiede di accedere al livello successivo (t_1) sommato al tempo già perso (t_0):

$$T = H_0 t_0 + (1 - H_0)(t_1 + t_0)$$

- **Gerarchia a tre livelli:** aggiungendo un livello, la formula si espande. Dobbiamo considerare tre scenari: successo in L1, successo in L2 (dato miss in L1), e miss in entrambi (accesso a L3/RAM). La formula estesa è:

$$\begin{aligned} T &= H_0 t_0 + (H_1 - H_0)(t_1 + t_0) + (1 - H_1)(t_2 + t_1 + t_0) \\ &= t_0 + (1 - H_0)t_1 + (1 - H_1)t_2 \end{aligned}$$

Dove:

- $H_0 t_0$: Tempo speso se trovo il dato subito in L1;
- $(H_1 - H_0)(t_1 + t_0)$: Tempo speso se il dato è in L2 ma non in L1 (pago t_1 e t_0);
- $(1 - H_1)(t_2 + t_1 + t_0)$: Tempo speso se devo scendere fino al terzo livello (pago tutti i tempi di accesso).

Semplificazione e Formula Generale

La forma semplificata del tempo di accesso della gerarchia a tre livelli, permette di generalizzare il calcolo per induzione.

$$T = t_0 + (1 - H_0)t_1 + (1 - H_1)t_2$$

Per una **gerarchia a N livelli** (da 0 a N-1), la formula generale è:

$$T = t_0 + \sum_{i=1}^{N-1} (1 - H_{i-1})t_i$$

Questa formula finale dimostra che il tempo totale è dato dal tempo base del primo livello (t_0) più le penalità dei livelli successivi (t_i), ponderate per la probabilità di Miss del livello precedente ($1 - H_{i-1}$).

6.5.11 Esempio Reale: Evoluzione dell'architettura Intel

Per comprendere come le tecnologie di Caching siano evolute nel tempo, è utile analizzare la storia dei processori Intel. Questo percorso mostra il passaggio da sistemi senza cache a sistemi multi-livello complessi.

Evoluzione storica

- **Intel 80386:** Non supportava alcuna cache interna.
- **Intel 80486:** Introduce per la prima volta una cache interna da 8KB. Era una cache **unificata** (Dati + Istruzioni insieme) in cui ogni linea poteva contenere un blocco da 16 byte ed era organizzata col metodo associativo su insiemi con 4 linee per set.
- **Pentium (tutte le versioni):** Introduce l'architettura a **doppia cache interna** (Split Cache), una dedicata ai dati e una alle istruzioni, per migliorare il parallelismo.

Analisi del Pentium 4

Il processore Pentium 4 rappresenta un ottimo esempio di applicazione delle tecniche avanzate di gerarchia della memoria (L1 + L2).

1. Cache Interne (L1):

- **Dimensione:** 8KB (separate per Dati e Istruzioni);
- **Dimensione linea:** 64 byte;
- **Mappatura:** Associativa su insiemi a 4 vie (4-way).

2. Cache Esterna (L2):

- **Dimensione:** 256KB (molto più grande della L1);
- **Dimensione linea:** 128 byte (linee più lunghe per sfruttare meglio la località spaziale);
- **Mappatura:** Associativa su insiemi a 8 vie (8-way, per ridurre i conflitti);
- **Connessione:** È collegata a entrambe le cache interne (Dati e Istruzioni).

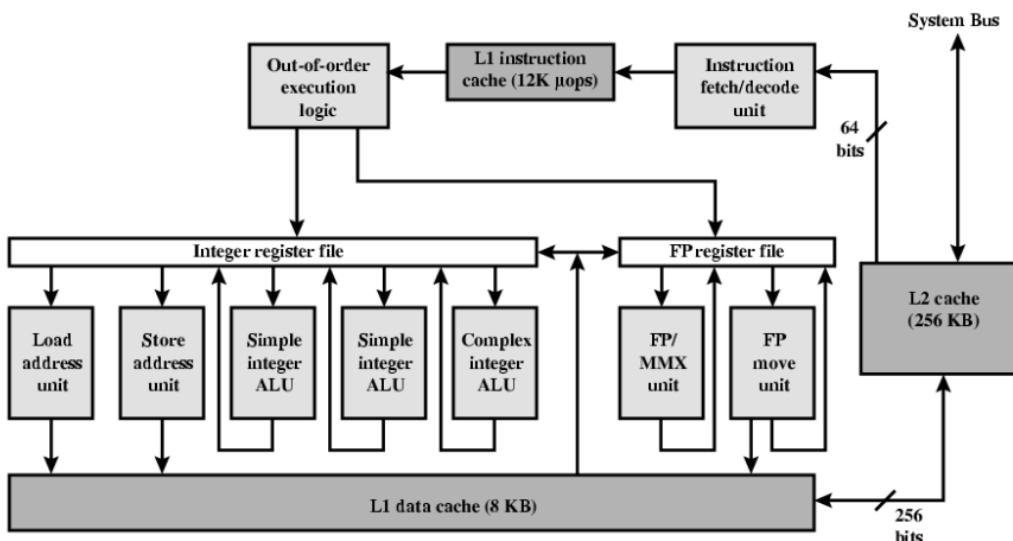


Figura 6.37: Schema a blocchi della gerarchia di memoria nel Pentium 4

Facendo riferimento alla figura 6.37, il testo descrittivo illustra il flusso dati tra le unità di elaborazione e le cache:

- **Fetch/Decode Unit:** Preleva le istruzioni in ordine dalla L2, le decodifica in una serie di **micro-operazioni** elementari e salva i risultati nella **L1 Instruction Cache**;

- **Out-of-order execution logic:** È il componente che pianifica l'esecuzione. Non segue rigidamente l'ordine sequenziale delle istruzioni, ma organizza le micro-operazioni in base alla **disponibilità dei dati** e delle risorse (data dependencies). Se un'istruzione è pronta per essere eseguita (ha tutti i dati), viene processata subito, anche se nell'ordine originale del programma si trovava dopo un'istruzione che è momentaneamente in attesa;
- **Execution Units:** Eseguono le operazioni prelevando i dati necessari dalla **L1 Data Cache**. Questo conferma l'architettura "Split Cache": mentre una parte della CPU preleva istruzioni dalla L1-Istr, un'altra parte elabora dati dalla L1-Dati contemporaneamente;
- **Memory Subsystem:** Questa unità include le cache L2 e L3 e il bus di sistema. Viene utilizzata per accedere alla memoria principale (RAM) quando si verificano dei Cache Miss (ovvero quando i dati non si trovano né in L1 né in L2).

Capitolo 7

Microarchitetture Avanzate

7.1 Introduzione alle microarchitetture

Definizione 7.1: Microarchitettura

Una **Microarchitettura** descrive l'organizzazione a livello hardware e l'implementazione interna di una CPU, definendo il modo in cui le istruzioni vengono effettivamente eseguite e manipolate.

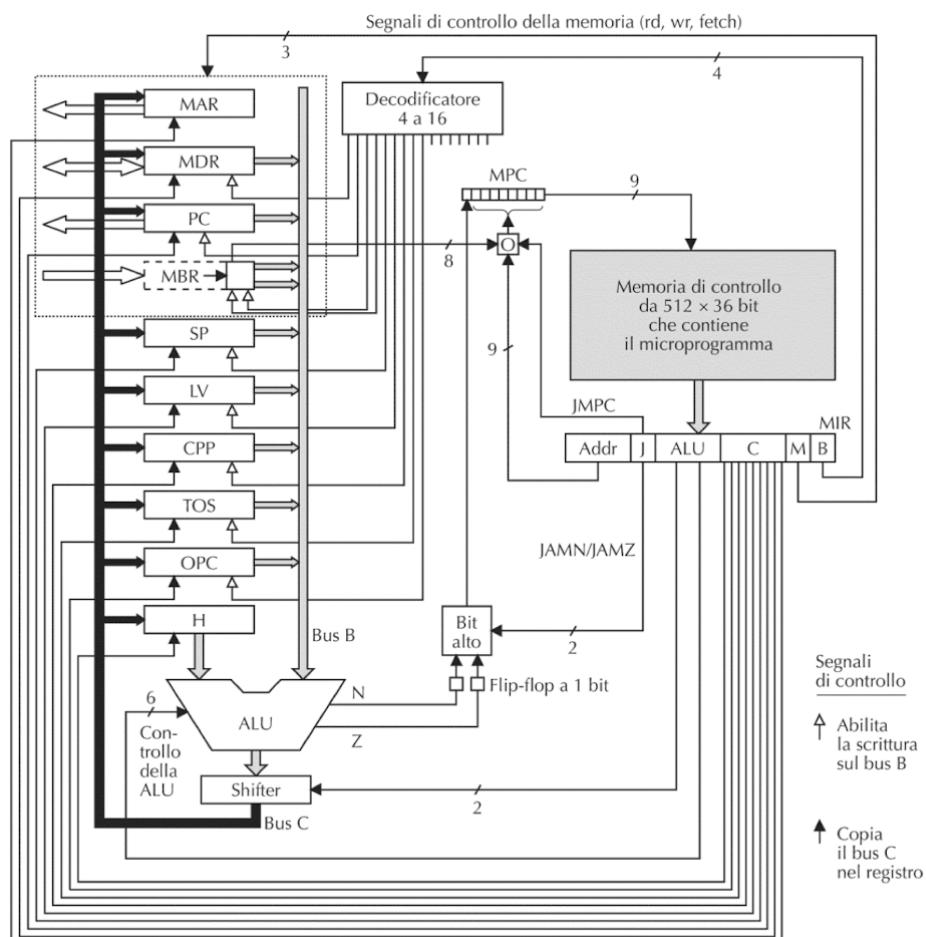


Figura 7.1: Schema concettuale della Microarchitettura

7.1.1 Trade-Off Fondamentale

Nella progettazione esistono due verità contrapposte:

- Macchine semplici non sono veloci.
- Macchine veloci non sono semplici.

Nota Bene

Le Architetture **MIC1** e **MIC2** sono macchine molto semplici (rispetto a ciò che si trova normalmente in commercio o nella letteratura scientifica) che hanno messo da parte il lato prestazionale per agevolare la didattica.

7.1.2 Prestazioni

Per migliorare le prestazioni di una macchina, l'obiettivo principale è **ridurre la lunghezza del percorso di esecuzione**.

Ciò può essere ottenuto tramite diverse azioni, come:

- Unire il ciclo di interpretazione alla fine della sequenza di microcodice.
- Passare da un'architettura a due BUS ad una a tre BUS (per parallelizzare i trasferimenti).
- Far prelevare le istruzioni dalla memoria da un'unità specializzata (*Instruction Fetch Unit*).

7.2 Pipelining: principi e struttura

7.2.1 Introduzione e esempi Intuitivi

Come abbiamo già accennato in precedenza la pipeline è una tecnica di implementazione hardware che migliora le prestazioni del processore sfruttando il parallelismo a livello di istruzione. Invece di eseguire un'istruzione completa prima di passare alla successiva, la pipeline divide il processo di esecuzione in più stadi distinti (gestiti da hardware dedicato). Il concetto chiave: Questa suddivisione permette di **sovraporre l'esecuzione di più istruzioni contemporaneamente**. Mentre un'istruzione sta usando l'unità di calcolo (Execute), la successiva può essere decodificata (Decode) e quella dopo ancora prelevata dalla memoria (Fetch). L'obiettivo ideale è riuscire a completare un'istruzione a ogni ciclo di clock ($CPI \approx 1$), aumentando drasticamente la larghezza di banda (throughput) del processore.

Nota Bene

Un classico esempio, citato sia in vari libri che nelle nostre slide, è quello della lavandaia. Secondo me è perfetto per comprendere concretamente i vantaggi che comporta l'utilizzo di una pipeline all'interno di un qualsiasi sistema.

Metafora della lavandaia

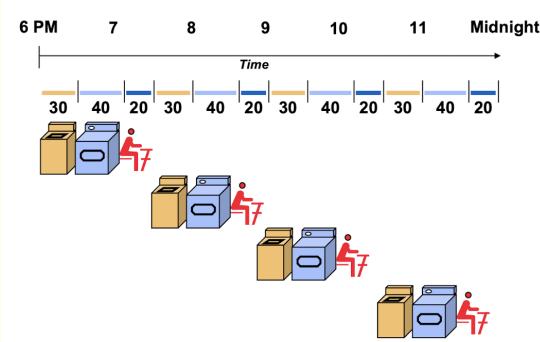
Immaginiamo di dover gestire quattro carichi di bucato e di avere a disposizione tre risorse funzionali distinte, ciascuna con un tempo di esecuzione specifico:

1. Lavatrice: impiega 30 minuti.
2. Asciugatrice: impiega 40 minuti.
3. Piegatura (Folding): impiega 20 minuti.

Il tempo totale per completare un singolo carico, dalla lavatrice alla piegatura, è quindi di 90 minuti ($30 + 40 + 20$).

L'Approccio Sequenziale (Single-Cycle)

In un sistema non ottimizzato (sequenziale), l'operatore attenderebbe il completamento totale del primo carico (lavaggio, asciugatura e piegatura) prima di inserire il secondo carico nella lavatrice. In questo scenario, ogni carico richiede 90 minuti e la risorsa "Lavatrice" rimane inattiva mentre l'asciugatrice e l'operatore addetto alla piega lavorano.



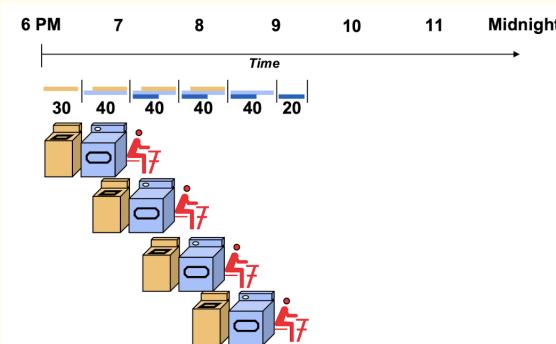
- Risultato: $4 \text{ carichi} \times 90 \text{ minuti} = 6 \text{ ore totali per completare il lavoro.}$

L'Approccio Pipelined

La tecnica del pipelining introduce il concetto di sovrapposizione temporale (overlap). Non appena il primo carico termina il lavaggio e passa nell'asciugatrice, la lavatrice si libera. Invece di lasciarla vuota, inseriamo immediatamente il secondo carico.

- Alle 6:00 PM iniziamo il primo carico.
- Alle 6:30, il carico 1 passa all'asciugatrice e il carico 2 entra in lavatrice.
- Alle 7:10, il carico 1 passa alla piegatura, il carico 2 passa all'asciugatrice e il carico 3 entra in lavatrice.

In questo scenario, lavatrice, asciugatrice e operatore lavorano contemporaneamente su carichi diversi.



- Risultato: Il tempo totale scende drasticamente a 3.5 ore.

Dall'analisi di questa metafora emergono quattro principi tecnici che si applicano direttamente ai microprocessori:

1. **Latenza vs. Throughput:** Il pipelining non riduce il tempo di esecuzione del singolo task. Il primo carico impiega sempre 90 minuti per essere pronto (30+40+20). Ciò che migliora è il throughput, ovvero la quantità di lavoro completato nell'unità di tempo complessiva.
2. **Il Collo di Bottiglia (Bottleneck):** La velocità massima della pipeline è limitata dallo stadio più lento. Nel nostro esempio, l'asciugatrice richiede 40 minuti. Anche se la lavatrice ne impiega solo 30, non può procedere più velocemente di un carico ogni 40 minuti, altrimenti i panni bagnati si accumulerebbero davanti all'asciugatrice occupata. In una CPU, la frequenza di clock è dettata dallo stadio hardware più lento.
3. **Fasi di Riempimento e Svuotamento:** La pipeline non lavora a pieno regime istantaneamente. Esiste un tempo iniziale per "riempire" la pipeline (Filling) in cui solo alcune unità sono attive, e un tempo finale per "svuotarla" (Draining/Emptying). Il guadagno prestazionale massimo si ottiene solo quando la pipeline è a regime (Full).
4. **Dipendenza dalle Risorse:** Affinché il sistema funzioni, ogni stadio (lavaggio, asciugatura, piega) deve essere indipendente. Se l'asciugatrice e la lavatrice condividessero lo stesso motore elettrico, non potrebbero funzionare contemporaneamente, creando un conflitto strutturale (Structural Hazard)

7.2.2 Pipeline a 5 stadi (IF, ID, EX, MEM, WB)

Introduciamo adesso le vere fasi che compongono una pipeline (didattica) all'interno di un processore.

Gli step fondamentali sono 5:

1. **IF → Fetch:** Il processore preleva l'istruzione dalla memoria e incrementa il Program Counter
2. **ID → Decode:** L'unità di controllo decodifica l'istruzione e legge gli operandi dai registri sorgente
3. **EX → EXecute:** La ALU esegue l'operazione aritmetica o calcola l'indirizzo di memoria necessario
4. **MEM → MEMory:** Se l'istruzione lo richiede, si accede alla memoria dati per leggere o scrivere un valore
5. **WB → Write Back:** Il risultato dell'elaborazione o il dato letto dalla memoria viene scritto nel registro di destinazione

È importante capire quanto (e se) la nostra pipeline sta influenzando in modo positivo i nostri processi. Per determinarlo c'è un parametro fondamentale chiamato **CPI** (Avarage Cycle Per Instructions), misura il numero medio di cicli di clock necessari al processore per completare una singola istruzione.

Confronto Prestazionale: Perché la Pipeline vince?

La seguente tabella riassume perfettamente come l'architettura Pipelined riesca a ottenere "il meglio dei due mondi" rispetto agli approcci precedenti.

Single Cycle Datapath	CPI = 1	Long Cycle Time
Multi-cycle Datapath	CPI ≈ 4	Short Cycle Time
Pipelined Datapath	CPI ≈ 1	Short Cycle Time

Analizziamo nel dettaglio cosa significano questi dati:

Single Cycle Datapath (Ciclo Singolo)

Ogni istruzione viene eseguita interamente in un solo colpo ($CPI = 1$).

Il problema? Per fare tutto in un unico ciclo, il periodo di clock deve essere lunghissimo (deve accomodare l'istruzione più lenta), rallentando drasticamente la frequenza della CPU.

Multi-cycle Datapath (Multiciclo)

L'istruzione viene spezzata in passi più piccoli (fetch, decode, etc.). Questo permette di avere un tempo di ciclo molto breve (alta frequenza).

Il problema? Servono molti cicli per completare una singola istruzione (in media ≈ 4), rendendo l'esecuzione totale lenta.

Pipelined Datapath (Pipeline)

Grazie alla sovrapposizione temporale (overlap), una volta che la pipeline è a regime, esce un'istruzione completata ad ogni ciclo di clock ($CPI \approx 1$).

Il vantaggio: Mantiene il tempo di ciclo breve del multiciclo, ma recupera l'efficienza del ciclo singolo.

L'obiettivo è quello di tenere un CPI tendente a uno che significherebbe avere un PipeLining che termina un'istruzione per ciclo, quindi *Hardware perennemente impegnato*. Ma **non sempre è possibile ottenere questo risultato**, i fattori determinanti sono principalmente 2:

- **Controlli di flusso:** Immaginiamo a funzioni come if o while, a seconda dei parametri in ingresso cambiano l'istruzione successiva che dobbiamo eseguire, questo comlica il lavoro della pipeline. (più avanti vedremo sia dei problemi reali con vere microistruzioni che le varie strategie di "soluzione" o contentimento del problema)
- **Istruzioni all'interno del codice:** alcune istruzioni non necessitano di 5 stadi, potrebbero terminare il loro *lavoro* la terza o quarta fase.

Instruzione	Step					Descrizione
beq	IF	ID	EX	X	X	Branch (Salto). Non accede alla memoria dati e non scrive nei registri, quindi salta MEM e WB.
R-type	IF	ID	EX	X	WB	Operazioni aritmetiche (add, sub). Lavorano solo sui registri, quindi saltano l'accesso in memoria (MEM).
sw	IF	ID	EX	MEM	X	Store Word. Scrive un dato in memoria. Non deve restituire nulla al Register File, quindi salta WB.
lw	IF	ID	EX	MEM	WB	Load Word. È l'istruzione più lunga: usa tutte le fasi perché legge dalla memoria e scrive nel registro.

7.3 Pipeline MIPS

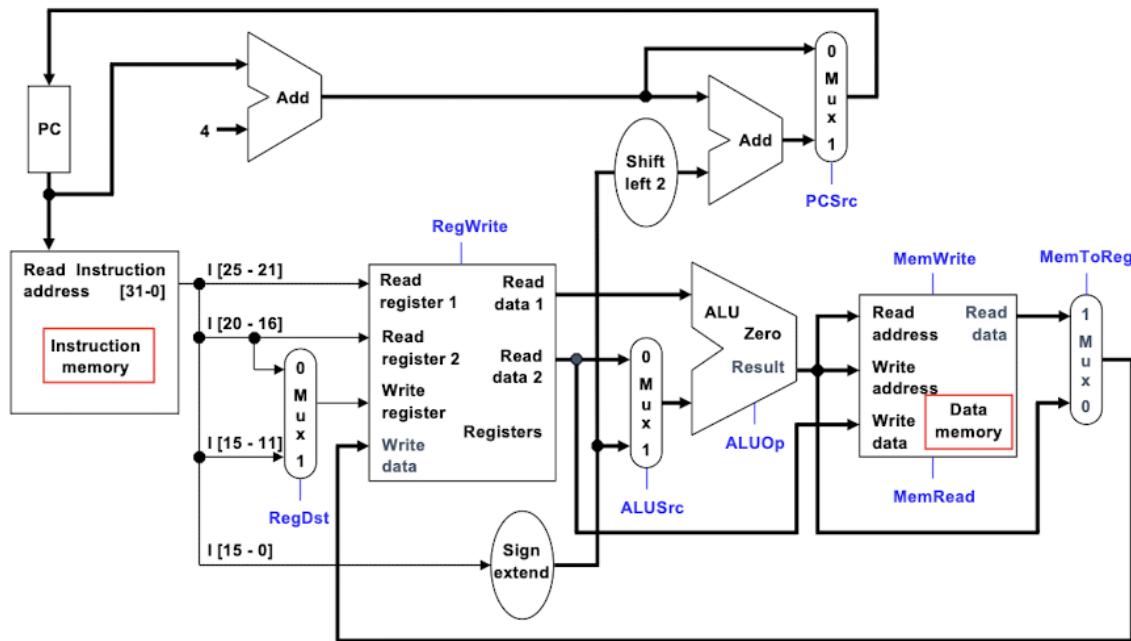
In questa sezione andiamo a vedere una vera pipeline come funziona a livello Hardware. Vediamo nello specifico la **MIPS** è il classico esempio di architettura utilizzata per illustrare i principi del RISC (Reduced Instruction Set Computer), in contrapposizione alle architetture CISC più complesse. La sua importanza didattica e industriale deriva dalla sua regolarità e semplicità, che permettono un'implementazione efficiente della pipeline. Rappresenta un punto di svolta nella progettazione dei processori moderni.

Definizione 7.2: MIPS

Un'architettura **load/store** (solo le istruzioni di caricamento e memorizzazione accedono alla memoria) con istruzioni a lunghezza fissa di 32 bit. È progettata per massimizzare il parallelismo a livello di istruzione. Il suo set di istruzioni ridotto e regolare facilita la decodifica hardware e permette di ottimizzare il percorso dei dati (*datapath*) per l'esecuzione in pipeline, mirando a un CPI (*Cycles Per Instruction*) vicino a 1

7.3.1 Composizione Hardware

Dal punto di vista hardware, il datapath MIPS è composto da diverse unità funzionali che operano in sequenza. S



Sebbene la pipeline introduca registri intermedi per separare gli stadi, i componenti logici fondamentali rimangono gli stessi:

- 1. Program Counter (PC):** Un registro che contiene l'indirizzo dell'istruzione corrente.
- 2. Memoria Istruzioni (Instruction Memory):** Dove viene prelevata l'istruzione da eseguire.
- 3. Banco dei Registri (RegWrite):** Un insieme di registri a 32 bit per la lettura degli operandi e la scrittura dei risultati.
- 4. Unità di calcolo (ALU):** Il cuore di calcolo, usato per operazioni aritmetiche, logiche e per il calcolo degli indirizzi di memoria.
- 5. Memoria Dati (DataMemory):** Unità di memoria separata (o cache distinta) per le operazioni di lettura (load) e scrittura (store).
- 6. Multiplexer (MemToReg):** Componenti essenziali per indirizzare il flusso dei dati (es. scegliere se l'input della ALU proviene da un registro o da un campo immediato dell'istruzione).

7.3.2 Le fasi della Pipeline MIPS

In questo modello (Single-Cycle), tutte le 5 fasi vengono schiacciate in un unico ciclo di clock. Sulla carta sembra ottimo ($CPI = 1$), ma c'è una fregatura enorme: il periodo di clock deve essere abbastanza lungo da farci stare l'istruzione più lenta di tutte (ovvero la *Load Word*, $1w$, che deve farsi tutto il giro dell'hardware).

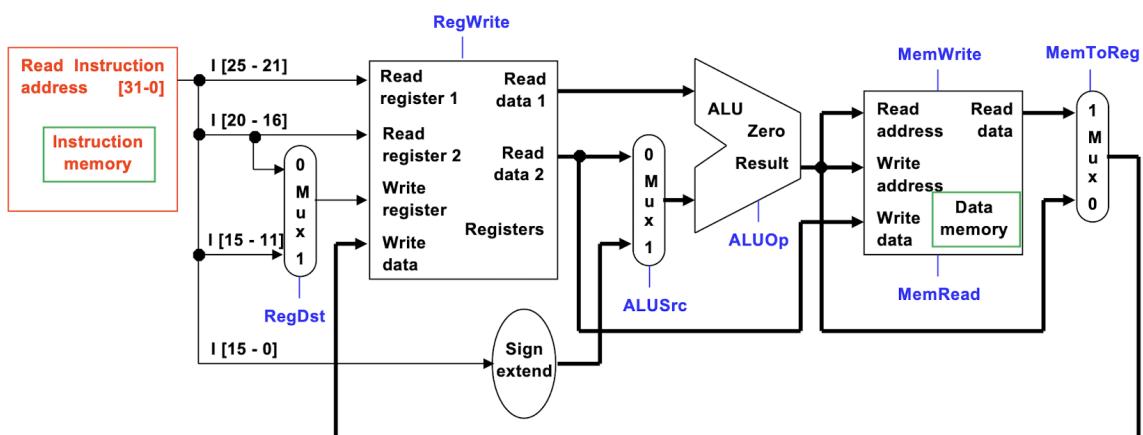
Il risultato è un'inefficienza assurda. Componenti costosi come la ALU o la memoria passano gran parte del tempo a "girarsi i pollici" in attesa che il ciclo finisca. Esempio pratico: se la $1w$ impiega 8ns, il clock dev'essere settato a 8ns per **tutte** le istruzioni, anche per quelle molto più veloci che finirebbero prima.

Nota Bene

È come quando si cammina in gruppo: la velocità del gruppo dipende da quello che cammina più lento

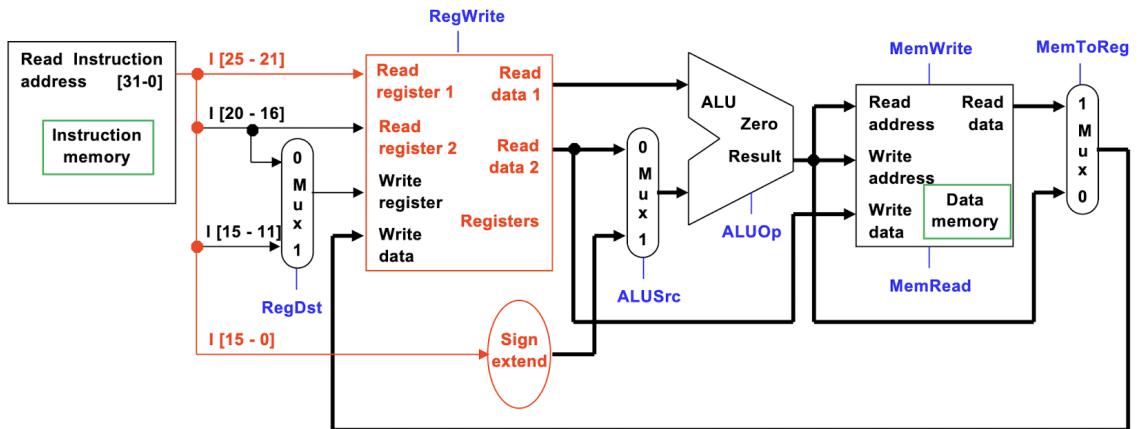
Per risolvere il problema, il MIPS spezza tutto in 5 fasi distinte. Questo è il trucco che permette all'hardware di fare 5 cose diverse nello stesso momento. Vediamole una per una:

1. Instruction Fetch (IF):



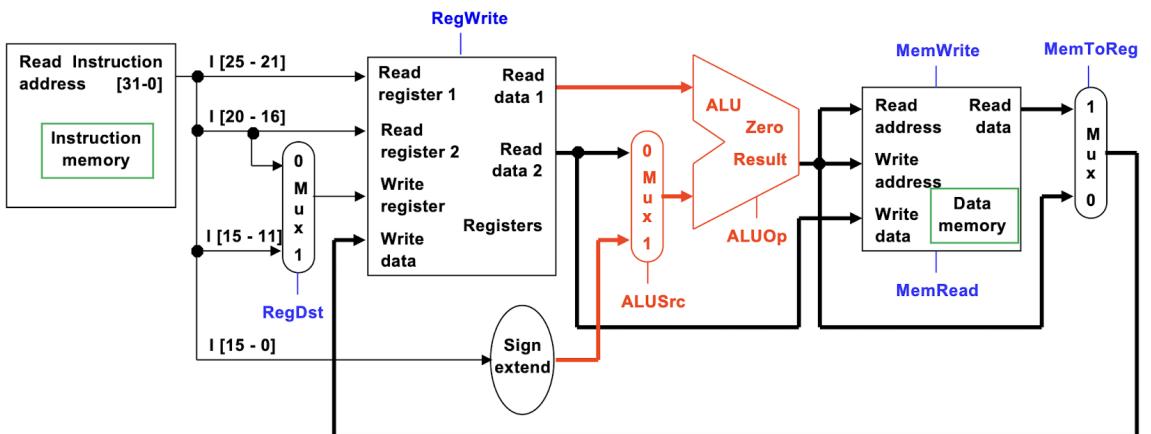
Qui il processore usa il *Program Counter* (PC) per andare a pescare l'istruzione dalla memoria. Nel frattempo, un sommatore dedicato si porta avanti e incrementa il PC di 4 ($PC + 4$), così l'indirizzo per l'istruzione successiva è già pronto.

2. Instruction Decode (ID):



L'unità di controllo decodifica l'istruzione per capire cosa deve fare. In parallelo, vengono letti i valori dai registri. Ah, se c'è una costante immediata (16 bit), viene subito estesa di segno a 32 bit per renderla compatibile con le operazioni successive.

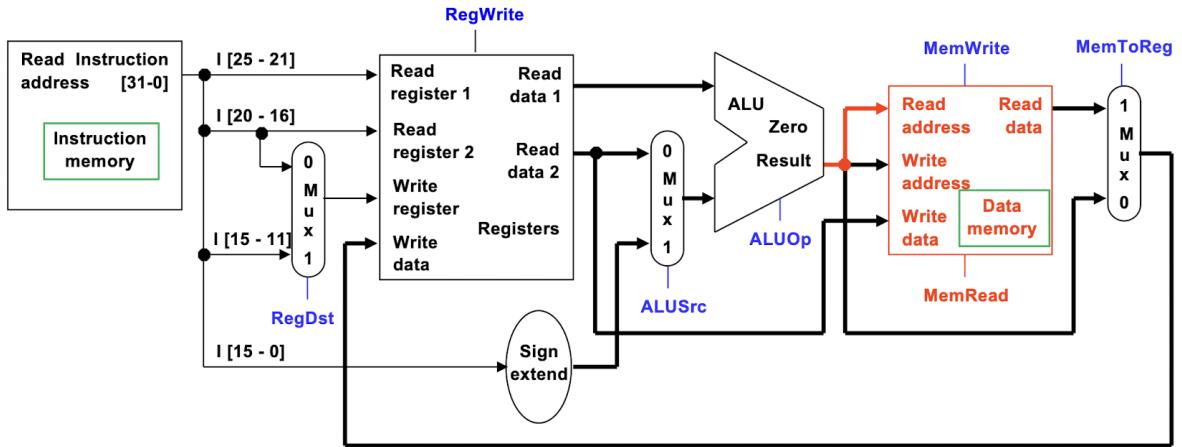
3. Execute (EX):



Qui entra in gioco la ALU e si fanno i calcoli veri. Cosa calcola? Dipende dall'istruzione:

- **R-Type:** fa i calcoli aritmetici/logici sui valori letti dai registri.
- **Load/Store:** somma il registro base + l'offset per calcolare l'indirizzo di memoria effettivo.
- **Branch:** calcola l'indirizzo di destinazione del salto.

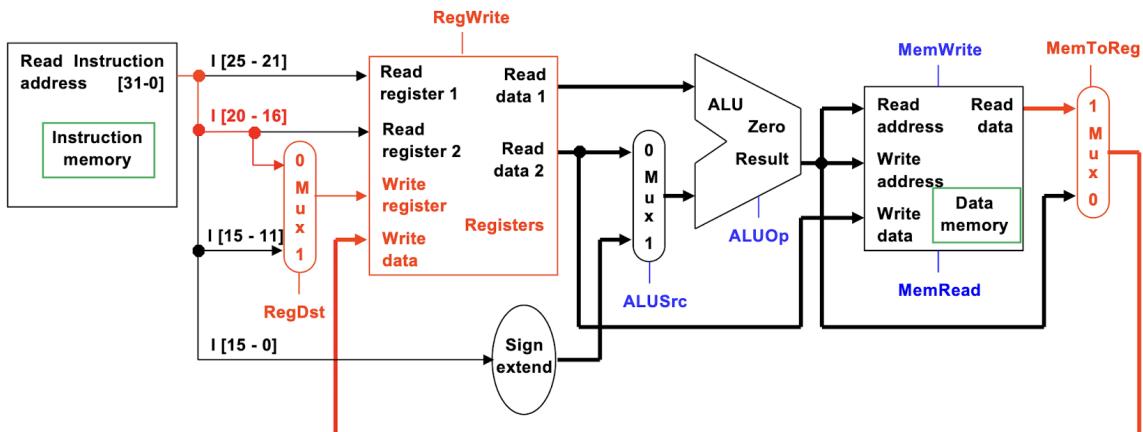
4. Memory Access (MEM):



Accesso alla memoria dati (attenzione: *dati*, non istruzioni). Si usa solo per caricare (lw) o salvare (sw).

- **Load:** legge il dato dall'indirizzo calcolato in fase EX.
- **Store:** scrive il dato all'indirizzo calcolato.
- **R-type:** non usano la memoria, quindi attraversano questo stadio senza fare nulla (idle).

5. Write Back (WB):



Per finire. Il risultato viene scritto nel registro di destinazione del Register File. Il dato può arrivare dalla ALU (se abbiamo fatto calcoli) o dalla memoria (se era una lw).

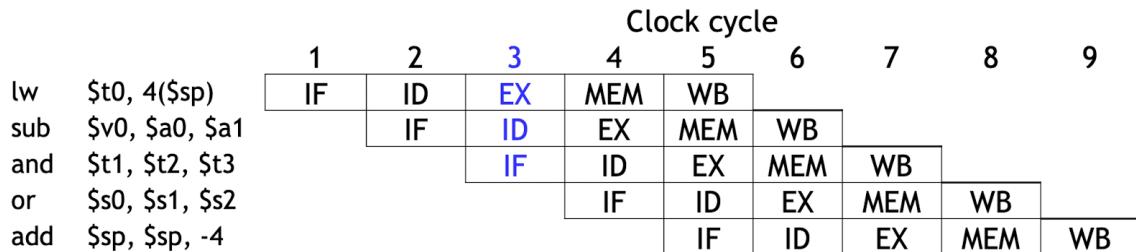
Nota Bene

Tutto questo funziona se le istruzioni sono indipendenti l'una dall'altra. In caso contrario, si verificano quelli che vengono tipicamente chiamati **Data Hazard**. È un argomento che approfondiremo meglio tra poco.

7.4 Diagrammi Temporali

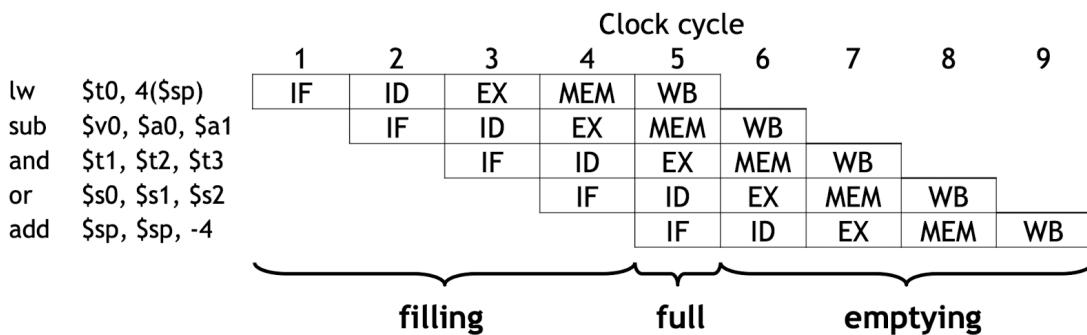
In questa breve sezione si mostrano i diagrammi temporali della pipeline. Non aggiungono assolutamente niente di nuovo alle informazioni che già sono state acquisite fino ad ora riguardo

la pipeline, il loro unico scopo è quello di mostrare in maniera semplice come sta procedendo il flusso di lavoro.



- l'asse verticale scandisce la sequenza di istruzioni (in ordine dall'alto verso il basso)
 - l'asse orizzontale il passare del tempo, scandito per cicli di clock
 - Ogni istruzione è divisa nei 5 stadi che abbiamo visto nel paragrafo 7.3.2 . Poiché ogni istruzione inizia un ciclo dopo la precedente, si crea una forma caratteristica "a gradini"

7.4.1 Terminologia



1. La prima fase di riempimento si dice **filling**, parte dall'inserimento della prima istruzione e termina alla quinta, momento in cui tutti i 5 gli stadi sono occupati
 2. La seconda fase in cui la pipeline lavora a pieno regime si dice **full**, questa fase è *grasso che cola*, qui troviamo il vero guadagno dovuto alla pipeline. Nel nostro caso sfortunato avviene per un solo ciclo.
 3. L'ultima fase in cui la pipeline si svuota è chiamata *Emptying* sono i cicli finali (da 6 a 9). Non vengono immesse nuove istruzioni, ma si completano quelle rimaste "in volo" fino a svuotare la macchina.

7.5 Hazard della Pipeline

Vengono chiamate **Hazard** o conflitti tutte le situazioni in cui la pipeline non è libera di lavorare a pieno regime escluse ovviamente le fasi filling e emptying, sono come delle casistiche anomale (anche se come vedremo non sono per niente rare) che interrompono o complicano il lavoro. La maniera più semplice per capirle è elencarle direttamente.

7.5.1 Tipi di Hazard

- **Structural Hazards (Conflitti Strutturali):** Qui il problema è l'hardware: non ce la fa fisicamente a gestire quella combinazione di istruzioni nello stesso ciclo. Succede ad esempio quando due istruzioni provano a usare la stessa risorsa (tipo un'unica porta di memoria per dati e istruzioni) nello stesso istante.
- **Data Hazards (Conflitti sui Dati):** Si hanno quando un'istruzione ha bisogno di un dato che non è ancora pronto perché l'istruzione precedente lo sta ancora calcolando ed è ancora "in volo" nella pipeline. Si dividono in tre tipi:
 - **RAW (Read After Write):** È il tipo di conflitto più comune e rappresenta una *dipendenza vera* sui dati.

In pratica, l'istruzione successiva (J) è un po' troppo "impaziente" e tenta di leggere un registro prima che l'istruzione precedente (I) abbia finito di scriverci dentro il nuovo valore. L'esempio classico è una add che salva il risultato nel registro \$1, seguita subito da una sub che usa proprio \$1 come input. Nella pipeline, la sub proverà a leggere \$1 già nella sua fase di decodifica (ID), ma in quel momento la add sta ancora lavorando (EX o MEM). Risultato: la sub legge un valore vecchio e il calcolo va in malora.

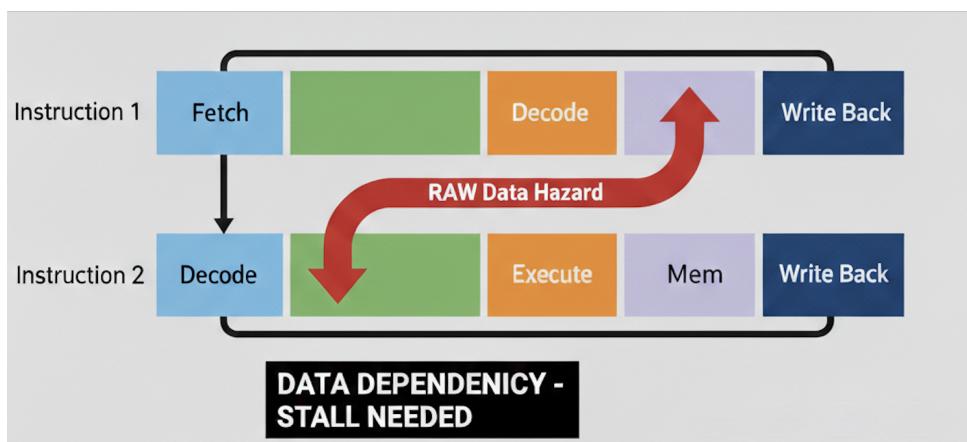


Figura 7.2: Schema concettuale raw hazard

Nel contesto MIPS questo accade spessissimo. La buona notizia è che si risolve quasi sempre con il **Forwarding** (senza fermare la macchina). La cattiva notizia è che c'è un'eccezione: il **Load/Use Hazard**. Quando il dato deve essere pescato dalla memoria (Load), ci mette fisicamente troppo tempo per essere "girato" subito all'istruzione dopo. In quel caso specifico, il forwarding non basta e siamo obbligati a inserire uno **stall**.

- **WAR (Write After Read):** Questo conflitto è noto come **anti-dipendenza**.

Il problema nasce quando l'istruzione successiva (J) scrive su un registro *prima* che l'istruzione precedente (I) abbia fatto in tempo a leggerlo come input. Se succedesse, l'istruzione I finirebbe per leggere il valore "nuovo" (appena scritto da J) invece di quello originale che le serviva, sballando il calcolo.

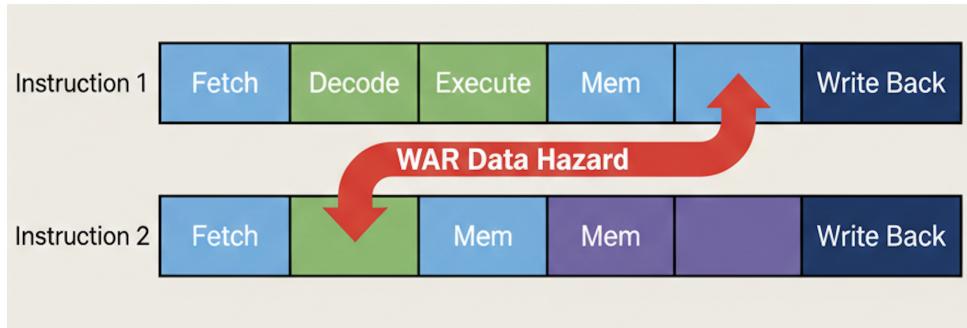


Figura 7.3: Schema concettuale war hazard

La buona notizia è che **nella pipeline MIPS a 5 stadi questo hazard non può verificarsi**. È una questione di architettura: le letture avvengono sempre all'inizio (stadio 2, ID), mentre le scritture avvengono solo alla fine (stadio 5, WB). Poiché l'istruzione I parte prima di J, avrà già letto i suoi dati molto prima che J arrivi al traguardo per sovrascriverli.

- **WAW (Write After Write):** Questo conflitto è chiamato **dipendenza di output**. Il problema qui è l'ordine finale delle cose: l'istruzione successiva (J) riesce a scrivere sul registro di destinazione *prima* che lo faccia l'istruzione precedente (I). Se succede, il risultato finale è sbagliato: il registro rimarrà con il valore scritto da I (che ormai è "vecchio") sovrascrivendo quello di J (che era l'aggiornamento più recente). Le scritture sono avvenute nell'ordine inverso.

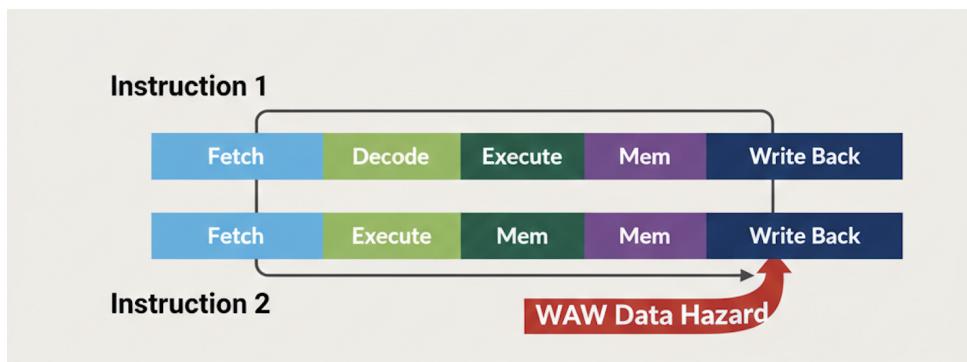


Figura 7.4: Schema concettuale waw hazard

Anche in questo caso, **nella pipeline MIPS standard a 5 stadi siamo salvi**. Dato che tutte le istruzioni impiegano esattamente 5 cicli e scrivono il risultato solo alla fine (in fase WB), l'ordine cronologico è sempre rispettato. Questo hazard diventa però un problema reale in architetture più complesse o superscalari, dove istruzioni diverse hanno durate diverse (es. operazioni floating point) e possono finire fuori ordine.

- **Control Hazards (Conflitti di Controllo):** Il problema dei salti (branch). Il processore deve decidere quale istruzione caricare (Fetch) *subito*, ma la decisione vera (se saltare o no) viene presa solo dopo (in EX). Se indovina male, carica istruzioni inutili.

7.5.2 Tecniche risolutive

Ci sono diverse tecniche per gestire gli Hazard che abbiamo visto, in generale anche in questo caso vale la solita regola aurea per cui soluzioni più semplici sono meno efficaci rispetto a soluzioni più complesse.

Soluzioni Hardware

Nelle soluzioni Hardware il processore implementa unità logiche dedicate che monitorano costantemente il flusso delle istruzioni. Vediamo adesso 3 strategie Hardware:

1. Tecnica degli Stalli:

Questa tecnica è gestita interamente dall'hardware tramite una **Hazard Detection Unit**. Quando il processore capisce che i dati non sono pronti e nemmeno il *Forwarding* può salvarci, l'unica soluzione è mettere tutto in pausa. La pipeline viene "congelata" inserendo una **bolla** (bubble): in pratica si genera internamente un'istruzione **NOP** (No Operation) che scorre nella pipeline senza fare nulla, creando un ritardo intenzionale.



Figura 7.5: Schema concettuale strategia basata su Stalli

Ma quando serve davvero? Il forwarding risolve quasi tutto, ma fallisce nel caso del **Load/Use Hazard**. Immagina una lw seguita subito da una add che usa quel dato. Il problema è fisico: la lw legge il dato dalla memoria solo alla fine del 4° stadio (MEM), ma la add lo vorrebbe già all'inizio del suo 3° stadio (EX). Non possiamo mandare un dato "indietro nel tempo". Ecco perché in questo specifico caso l'hardware **deve** inserire uno stall di 1 ciclo.

```
L1: lw $s0, 4($t1)      # $s0 <- M [4 + $t1]
L2: add $s5, $s0, $s1  # 1° operand depends from L1
```

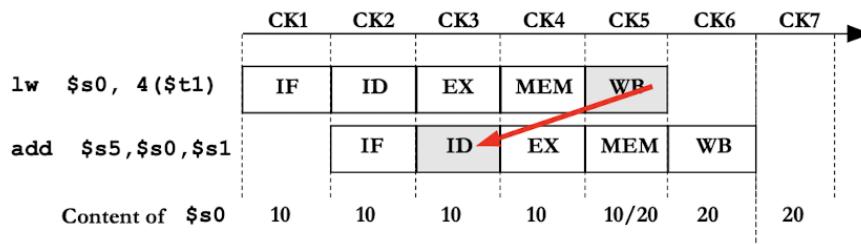


Figura 7.6: Schema concettuale del caso Load/Use

Il "Cervello": L'Hazard Detection Unit

Questa unità lavora nello stadio di decodifica (ID) e fa il vigile urbano monitorando i registri. La sua logica è semplice ma ferrea: controlla se l'istruzione che si trova nello stadio successivo (EX) è una lettura di memoria (MemRead attivo, quindi una 1w) e se il registro dove andrà a scrivere corrisponde a uno di quelli che l'istruzione corrente sta cercando di leggere. In "codice", la condizione che fa scattare l'allarme è questa:

```
if (ID/EX.MemRead and ((ID/EX.Rt == IF/ID.Rs) or (ID/EX.Rt == IF/ID.Rt)))
```

Cosa succede durante lo stall

Se la condizione sopra è vera, l'unità tira il freno a mano agendo su tre fronti contemporaneamente:

- (a) **Blocca il PC e il registro IF/ID:** Disabilita la scrittura su questi componenti. Così facendo, non viene pescata nessuna nuova istruzione e quella che era in decodifica non avanza, rimanendo lì per essere riletta al ciclo dopo.
- (b) **Inserisce la Bolla:** Forza a 0 tutti i segnali di controllo diretti verso lo stadio EX (tramite un multiplexer). Questo trasforma l'istruzione "fantasma" in una NOP, che non scriverà nulla né in memoria né nei registri.

Nota Bene

L'inserimento di stalli, seppur necessario per la correttezza, fa alzare il CPI medio sopra l'1 ideale, riducendo le prestazioni globali perché la CPU spreca cicli senza completare lavoro utile.

2. Forwarding:

Il Forwarding (o Bypassing) è una tecnica hardware intelligente per risolvere i Data Hazards (RAW) evitando il più possibile gli stalli. L'idea di base è semplice: il dato calcolato esiste dentro il processore (nei registri temporanei di pipeline) molto prima di essere scritto ufficialmente nel Register File (fase WB). Perché aspettare fino al ciclo 5 se ce l'abbiamo già? Per farlo, modifichiamo l'hardware aggiungendo dei **Multiplexer (MUX)** davanti alla ALU. Questi permettono di ignorare il Register File e prendere l'operando direttamente dagli stadi successivi. A gestire il traffico c'è la **Forwarding Unit**, un'unità logica che spia costantemente i numeri dei registri per capire se c'è un conflitto e deviare i dati corretti verso la ALU.

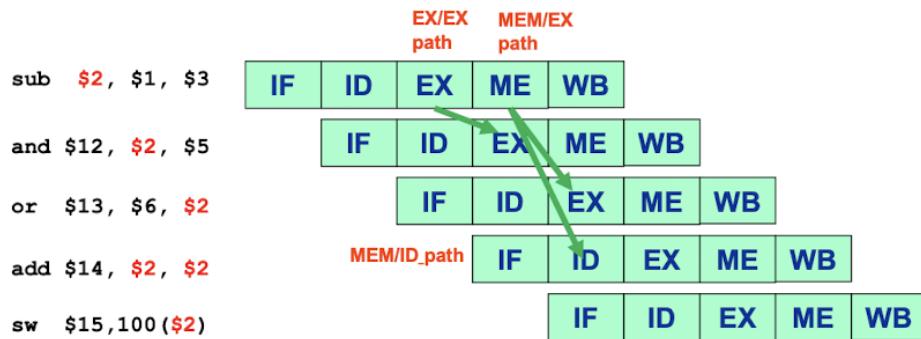


Figura 7.7: Schema concettuale strategia di Forwarding

I percorsi e la logica di attivazione

Esistono due "scorciatoie" principali. Se il conflitto è con l'istruzione appena passata (N-1), usiamo il percorso ****EX/EX****: prendiamo il risultato dal registro EX/MEM e lo giriamo subito alla ALU. Se il conflitto è con quella ancora prima (N-2), usiamo il percorso ****MEM/EX****, pescando il dato dal registro MEM/WB. La Forwarding Unit attiva questi percorsi solo se si verificano tre condizioni sacre: l'istruzione precedente sta effettivamente scrivendo ('RegWrite' attivo), non sta scrivendo sul registro \$0 (che è intoccabile), e ovviamente il registro di destinazione coincide con uno dei nostri operandi sorgente.

L'unico limite (No viaggi nel tempo) (Load/Use)

Il Forwarding è potente, ma non magico. Esiste un caso specifico, il ****Load/Use Hazard****, dove fallisce. Se un'istruzione 'lw' carica un dato, questo è disponibile fisicamente solo alla fine dello stadio MEM (ciclo 4). Se l'istruzione subito dopo ('add') lo vuole usare, ne ha bisogno all'inizio del ciclo 4 (fase EX). Non possiamo mandare indietro nel tempo un dato che non è ancora stato letto! In questo unico scenario, siamo costretti a inserire ****uno stallo di 1 ciclo****, per poi usare il forwarding subito dopo. *Piccola nota:* il caso Load/Store (carico e salvo subito) invece funziona liscio senza stalli, perché il dato serve allo stadio MEM e non EX.

3. Branch Prediction:

La Branch Prediction è sostanzialmente la "sfera di cristallo" dell'hardware. Il suo scopo è indovinare se un salto verrà preso o no (e dove andrà) *prima* che l'istruzione venga eseguita, così la pipeline non deve fermarsi ad aspettare. Il modo più semplice per farlo è la ****Predizione Statica****: si decide una regola fissa a priori (spesso lo fa il compilatore) e non si cambia mai. Le strategie classiche sono:

- **Always Not Taken:** Si assume che il salto non avvenga mai. Funziona bene per il codice sequenziale, ma con i loop è un disastro (precisione bassa, $\sim 30\%$).
- **Always Taken:** Si assume che si salti sempre (meglio, $\sim 60\%$).
- **BTFN (Backward Taken, Forward Not Taken):** La più intelligente. Si basa sulla logica del codice: se il salto va indietro (loop) si assume PRESO; se va avanti (spesso errori o *if* rari) si assume NON PRESO.

L'Hardware intelligente: La Predizione Dinamica

Qui le cose si fanno interessanti: il processore usa la memoria per imparare dal passato. Tutto inizia col ****Branch Target Buffer (BTB)****, una cache che si ricorda gli indirizzi dei salti già visti: se l'istruzione è lì, sappiamo subito dove andare senza fare calcoli. Ma *come* decidiamo se saltare? Usiamo la storia.

- **Contatori a 1 bit:** Memorizzano solo l'ultima scelta. Sono "volubili": basta un errore per fargli cambiare idea (nei loop sbagliano sempre due volte).
- **Contatori a 2 bit (Saturating):** La vera svolta. Usano 4 stati (da *Strongly Taken* a *Strongly Not Taken*). Per cambiare previsione devono sbagliare *due volte* di fila. Questa "isteresi" li rende molto più stabili nei cicli.
- **Predittori Avanzati:** I modelli *Two-Level* guardano non solo la storia del singolo salto, ma la correlazione con quelli vicini (pattern globali). I *Tournament Predictors* sono ancora più evoluti: usano più predittori diversi e un "arbitro" sceglie di volta in volta quale dei due sta indovinando meglio.

Il prezzo dell'errore: Misprediction e Flush

Cosa succede se la sfera di cristallo sbaglia la predizione (*Misprediction*)? Purtroppo bisogna pagare il conto. L'hardware deve eseguire un **“Pipeline Flush”**: tutte le istruzioni che erano state caricate sulla fiducia (la strada sbagliata, o *wrong path*) devono essere cancellate istantaneamente. La pipeline viene svuotata e si riparte dall'indirizzo corretto. Più la pipeline è profonda (ha tanti stadi), più cicli di lavoro abbiamo buttato via e maggiore sarà il calo di prestazioni (penalty).

Nota Bene

Nelle slide del Prof viene dedicato molto più spazio alla **Branch Prediction**. Io l'ho solo accennata, quindi se volete approfondire i vari tipi vi tocca guardarveli sulle slide originali.