Architettura Degli Elaboratori

Corso di Laurea in Informatica - Università degli Studi di Verona

Autore: Federico Brutti

Federico Brutti
federico.brutti@studenti.univr.it

Indice

6 | Codifica dell'Informazione

13 | Strategie di Ottimizzazione

15 | Progettazione Digitale

52 | Linguaggio HDL - Verilog

53 | Linguaggio Assembly

Forse dovrei ricordarvi il discorso dei CFU...

Codifica dell'Informazione

1.1 Elaboratori e codifica dell'informazione

Informatica è un acronimo che sta per *informazione automatica* ed il suo scopo è la risoluzione dei problemi mediante **algoritmi**[1]. Per quanto riguarda la qualità di esecuzione, esistono elaboratori più e meno efficienti, ma il nostro scopo principale è la creazione dei primi e ottimizzarli quanto più possibile. Esistono due tipi di macchine con le quali è possibile lavorare:

- **Sistemi Embedded**; composte da solo hardware, capaci di eseguire un singolo algoritmo.
- **Sistemi General Purpose**; composte dal connubio hardware-software, sono capaci di eseguire diversi algoritmi.

Nel corso di Programmazione hai potuto vedere il passaggio da algoritmo a software, ma qui ci concentreremo sulla comunicazione algoritmo ad hardware, che chiamiamo **Sintesi Logica**. In ogni caso, il sistema operativo lavora con il **linguaggio macchina**[2], il quale deve essere tradotto per essere leggibile da noi comuni mortali.

Come prima cosa, le informazioni vengono recepite dalla macchina per poi essere codificate ed inviate al sistema operativo; **Input**. Il tutto viene poi decodificato per far sì che si possa leggere; **Output**.

Figura 1.1: Ciclo di elaborazione informazioni

Gli elaboratori hanno risorse limitate e non è quindi possibile inserire infinite informazioni, ma il problema viene arginato dividendo in tanti pezzi l'informazione registrata, *Campionamento*, ed approssimando quanto possibile ad un numero che la macchina può leggere *Discretizzazione*. Vale qui il **Teorema di Shannon**; data una funzione in un intervallo di campionamento e discretizzazione, questi algoritmi garantiscono la presenza di un errore.

Le informazioni vengono ricevute ovviamente in un determinato arco di tempo, che viene misurato in **Hertz**.[3]

Passiamo ora invece alla *codifica*. Questo processo consiste nell'assegnazione di un codice binario ad ogni frammento di informazione entro un certo numero di *bits*. Supponiamo di avere 12M

[1] Insiemi di istruzioni non ambigue che risolvono un problema.

[2] Detto anche codice oggetto; si tratta di una sequenza di "0" ed "1" con un significato specifico per la macchina.

[3] Generalmente l'unità di misura dei processi. $1\text{Hz} = 1\text{ms}$

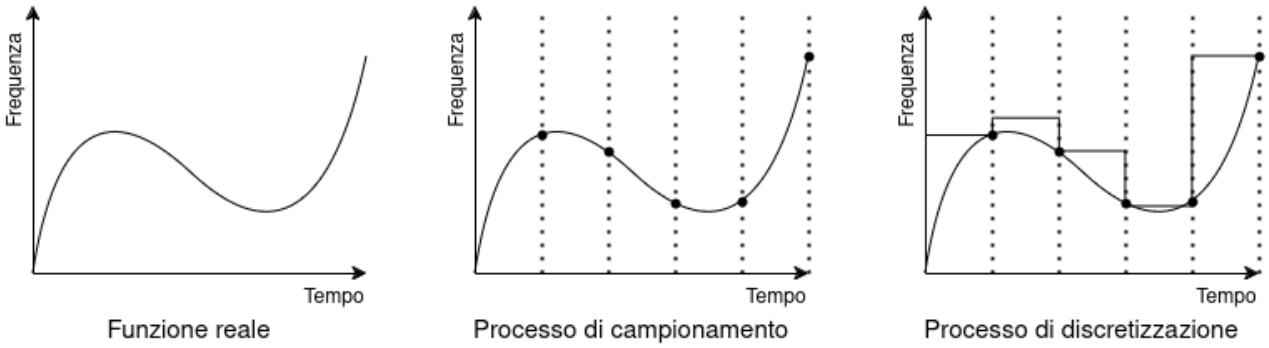


Figura 1.2: Processo di ricezione delle informazioni

unità di informazioni e dobbiamo trovare un numero di bit adeguato per poterle contenere tutte.

Il ragionamento va basato sulla potenza del 2. Abbiamo bisogno di 4b perché $2^4 = 16$ e sono quindi abbastanza per contenere le 12M.

Si possono scegliere diversi tipi di codifica a seconda delle caratteristiche del calcolatore, di cui due più importanti:

- *Velocità di elaborazione*; La frequenza di campionamento produce risultati ad una certa frequenza in Hz. Se la codifica richiedesse più tempo si perderebbero dati e quindi deve essere sempre maggiore o uguale alla frequenza di campionamento.

- *Facilità di calcolo*; Su alcune architetture è reso più semplice l'elaborazione delle informazioni.

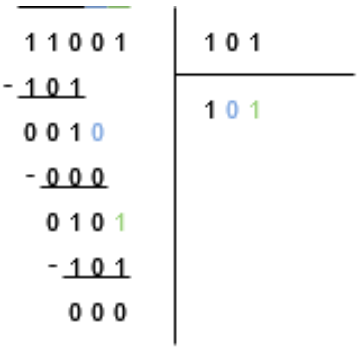
Ricorda sempre: **Bits are bits**; questo significa che (nei circuiti combinatori) due sequenze di bit identiche saranno comprese come la stessa informazione, quindi è necessario fare attenzione a come codificare; inoltre, il bit più a sinitra è detto più significativo, mentre l'opposto è il meno significativo. Infine, è possibile codificare pressoché qualunque cosa, come immagini, musica, e video, ma i dati più importanti rimangono i numeri e i caratteri; i quali sono codificati mediante il codice **ASCII** dove i primi 127b sono comuni a tutte le lingue.

1.2 Operazioni in base 2

Il nostro campo di lavoro è, come avrai capito, la base 2; nella quale valgono le solite quattro operazioni elementari, che adesso vedremo una per una nella loro forma binaria. Addizione, sottrazione e moltiplicazione sono intuitive e non dovrebbero portare problemi.

```
  - Addizione -              - Sottrazione -              - Moltiplicazione -
0   0       0              0   0       0                    0   0       0
0   1       1              0   1       1 (Carry-in)         0   1       0
1   0       1              1   0       1                    1   0       0
1   1       0 (Carry-out)  1   1       0                    1   1       1
```

La divisione è tuttavia un altro paio di maniche; può risultare contro-intuitiva a causa dell'utilizzo della sottrazione. Vediamo con un esempio: $\frac{11001}{101}$, ovvero $\frac{25}{5}$.



La logica dietro la divisione binaria è poter sottrarre il divisore al dividendo nel caso in cui questo "stia dentro" al primo. Poco chiaro? lascia che ti guidi.

Abbassa 110 e sottraigli 101 perché il divisore ci sta una volta. Otterrai 001, al quale dovrai aggiungere la cifra successiva del dividendo e scriverai "1" come prima cifra del risultato.

Osserva che 10 non ci sta in 101, quindi non gli si può sottrarre nulla e scriverai "0" come seconda cifra del risultato, procedendo ad abbassare l'ultima cifra del dividendo ed ottenere il numero 101 che, guarda un pò, è uguale al divisore e quindi ci sta dentro. $101-101=000$, è una divisione intera senza resto. Scrivi "1" come ultima cifra del risultato e hai finito.

Passiamo alle ultime due operazioni, utili per la *codifica in virgola mobile*, la quale vedremo nelle prossime sezioni, e per velocizzare moltiplicazione e divisione:

- **Shift Left**; Aggiunge uno zero alla fine del numero (Sposta tutte le cifre a sinistra di una posizione). $1101 \times \texttt{ShiftL} = 11010$.
- **Shift Right**; Sposta le cifre a destra ed aggiunge uno 0 a sinistra. $1101 \times \texttt{ShiftR} = 0110$.

1.3 Codifica dei numeri

In questa sezione vedremo come codificare i numeri e le particolarità di ogni algoritmo che svolge tale funzione. Tieni a mente che non esiste una codifica superiore alle altre ed ognuna di loro ha un suo perché. Inoltre, la pool di domande da esame comprende ognuna di queste codifiche quindi ti tocca imparartele lo stesso. Tutti questi algoritmi lavorano su cifre binarie e sono basati sulla *Notazione posizionale*[4].

1.3.1 Codifica standard

La codifica standard è un algoritmo utile per lavorare con numeri interi positivi. Bisogna inizialmente prendere la potenza del 2 più grande che si avvicina al numero da codificare, ma che non lo supera, per poi sottrarla all'altro. Ripetere fin quando il numero iniziale non è "0". Vediamo un esempio:

```
Numero da codificare: 683

683 - 512 = 171             Valore 1 in posizione bit 9
171 - 128 = 43              Valore 1 in posizione bit 7
43 - 32 = 11                Valore 1 in posizione bit 5
11 - 8 = 3                  Valore 1 in posizione bit 3
3 - 2 = 1                   Valore 1 in posizione bit 1
1 - 1 = 0                   Valore 1 in posizione bit 0

Codifica standard di 683 = 1010101011
```

[4]Il valore di un numero è dato dalla posizione delle sue cifre.

1.3.2 Codifica in modulo e segno

La codifica in modulo e segno non è molto dissimile dalla precedente; infatti l'unica differenza è l'utilizzo di un bit in più nella parte più significativa per marcare il segno positivo "0" o negativo "1".

```
Numeri da codificare: 227, -227

227 - 128 = 99              Valore 1 in posizione bit 7
99 - 64 = 35                Valore 1 in posizione bit 6
35 - 32 = 3                 Valore 1 in posizione bit 5
3 - 2 = 1                   Valore 1 in posizione bit 1
1 - 1 = 0                   Valore 1 in posizione bit 0

Ottenuta la codifica standard; 11100011 aggiungiamo il bit del segno:
227 = 011100011, -227 = 111100011
```

1.3.3 Codifica in virgola fissa

La codifica in virgola fissa è un algoritmo capace di tradurre i numeri razionali considerando separatamente parte intera e decimale. Abbiamo la fortuna che la virgola rimane nella stessa posizione della base 10.

in primo luogo bisogna codificare la parte intera come una normale codifica in modulo (eventualmente anche in segno), mentre per trovare quella decimale bisogna moltiplicare per 2 il numero. Se il risultato è maggiore o uguale a 1, si scrive 1 e si verifica la stessa condizione per la parte decimale risultante. Di norma è specificato quanti bit di precisione deve avere la parte decimale, perché spesso troverai numeri periodici (dove trovi cifre decimali uguali in verifica) e andresti avanti all'infinito.

Se ti vuoi male e vuoi verificare la correttezza del tuo risultato decimale, puoi sommare tutte le potenze negative di 2 e vedere cosa ti esce. Molto probabilmente, un risultato approssimato.

```
Numero da codificare: 56,83 in 4b di precisione

Parte intera: 56                    Parte decimale: 0,83
56 - 32 = 24    1 in bit 5          0,83 × 2 = 1,66     1 in bit -1
24 - 16 = 8     1 in bit 4          0,66 × 2 = 1,32     1 in bit -2
8 - 8 = 0       1 in bit 3          0,32 × 2 = 0,64     0 in bit -3
                                    0,64 × 2 = 1,28     1 in bit -4
Risultato: 111000,1101
```

1.3.4 Codifica in virgola mobile

La codifica in virgola mobile o *Floating Point* consente di ottenere numeri particolarmente grandi e piccoli. Risulta utile quindi per avvicinarsi al concetto di numero reale.

Un numero in floating point si esprime nella formula di *Notazione scientifica*[5] e si divide in tre parti a cui è associato un numero specifico di bits da un totale di 32 (float) oppure 64 (double); le quali sono:

[5] $N = \pm Mant \times Base^{\pm exp}$

- *Segno*; 1b
- *Esponente*; 8b, oppure 9b in doppia precisione.
- *Mantissa*; 23b, oppure 54b in doppia precisione.

Prima di poter lavorare sul numero è necessario **normalizzarlo**[6] attraverso le operazioni di shifting viste prima. Dipendentemente da quante posizioni sono modificate, sarà necessario sommare, se shiftR o sottrarre, se shiftL, tal numero all'esponente. Ottenuto l'esponente, bisogna sommargli +127[7].Inoltre, nella codifica della mantissa la cifra intera non è mai scritta perché è sempre la stessa e si può omettere. Vediamo ora i due tipi di esercizi proposti:

```
Ex. 1: Si codifichi in virgola mobile il numero -30,375

    1. Convertiamo in binario il numero con la codifica in virgola fissa:
    Parte intera: 30 = 11110                Parte decimale: 0,375 = 011
    30 - 16 = 14                            0,375 × 2 = 0,75
    14 - 8 = 6                              0,75 × 2 = 1,5
    6 - 4 = 2                               0,5 × 2 = 1
    2 - 2 = 0

    Codifica in virgola fissa: 11110,011

    2. Procediamo con la normalizzazione:
    11110,011 / 1000 = 1,1110011 × 2^4      Sommeremo 4 all'esponente.

    La mantissa sarà inoltre: 1110011...0.

    3. Troviamo l'esponente:
    Non c'è un esponente nel numero richiesto, quindi: 4 + 127 = 131.

    131 - 128 = 3       1 in bit 7
    3 - 2 = 1           1 in bit 1
    1 - 1 = 0           1 in bit 0

    131 = 10000011 - Esponente trovato!

    4. Ricomponiamo il tutto

    - Segno: 1
    - Esponente: 10000011
    - Mantissa: 1110011...0

    La codifica in virgola mobile di -30,375 è: 1 10000011 1110011...0
```

[6]Si intende portarlo in una forma dove rimane una singola cifra intera.

[7]Questa operazione si dice *Eccesso 127* ed è necessaria per codificare l'esponente nello standard IEEE754.

```
Ex 2: Quale decimale è la codifica in virgola mobile 0100011001000110...0?

    1. Dividiamo nelle varie parti i bits
    - Segno: 0
    - Esponente: 10001100
    - Mantissa: 1000110...0

    2. Otteniamo l'esponente decimale
    128 + 8 + 4 = 140
    140 - 127 = 13 - Esponente trovato!

    3. Ricaviamo la mantissa
    Considera che ora stai lavorando con cifre decimali, quindi le potenze
    del 2 dove sta il valore 1 saranno negative. In questo caso notiamo
    che si trovano nelle posizioni -1, -5 e -6, quindi:

    2^-1 + 2^-5 + 2^-6 = 0,547 - Valore della mantissa trovato!

    4. Ricostruiamo il decimale
    Il segno è positivo, ricorda di sommare 1 alla mantissa trovata e
    moltiplica ad essa l'esponente. Hai finito.

    1 × 1,547 × 2^13
```

Ci sono infine alcuni casi di cifre particolari a cui fare attenzione:

- $+0$; Tutte le cifre a 0.
- -0; Tutte le cifre a 0, tranne quella del segno.
- $+\infty$; Esponente massimo, il resto a 0.
- $-\infty$; Esponente massimo e bit segno a 1, il resto a 0.
- *Not a Number*; Qualunque numero superi gli infiniti.
- 2; Tutte le cifre a 0, tranne il bit più significativo dell'esponente.
- 2^{-145}; Tutte le cifre a 0 tranne il bit meno significativo. Si tratta del numero più piccolo ottenibile.

1.3.5 Codifica in complemento a 1 e a 2

Parleremo solamente della codifica in complemento a 2 in quanto è un singolo passaggio in più rispetto a quella ad 1 ed è anche più utile rispetto alla prima.

Il suo scopo principale è dividere a metà il totale delle codifiche ottenibili da $2^n\mathtt{b}$, rendendo più semplice ottenere numeri lunghi. Vediamo il procedimento:

```
Diciamo di avere a disposizione 4 bits, quindi 16 combinazioni diverse.
Per ottenere il complemento ad 1 basta invertire tutte le cifre, mentre
per il complemento a 2 bisognerà poi sommare 1 ad ogni combinazione.
```

```
Per esempio: codificare -3 in complemento a 2 con 4b di precisione.

3 = 0011 -> 1100 in compl. ad 1
1100 + 1 = 1101 in compl. a 2

Qui sotto lascio una piccola lista per esercitazione.

Codifica normale              Compl. ad 1              Compl. a 2
0000 = 0                      1111 = -8                1111 = -1
0001 = 1                      1110 = -7                1110 = -2
0010 = 2                      1101 = -6                1101 = -3
0011 = 3                      1100 = -5                1100 = -4
0100 = 4                      1011 = -4                1011 = -5
0101 = 5                      1010 = -3                1010 = -6
0110 = 6                      1001 = -2                1001 = -7
0111 = 7                      1000 = -1                1000 = -8
```

1.3.6 Codifica in esadecimale

Soon!

Strategie di Ottimizzazione

2.1 Introduzione

La domanda principale di questo capitolo è "Come realizzare un sistema digitale?" Ebbene, è necessario un modello apposito che consentirà di rappresentare appropriatamente la sua struttura.

Per far ciò useremo l'algebra di Boole; uno spazio ad n dimensioni misurate in base all'alfabeto che voglio dare allo spazio. Qui sono presenti solo due valori: 0 e 1. Secondo Boole, se si definisce una funzione che genera valori in un altro spazio, generalmente scritta $f(B^n) -> B^m$, questa potrà essere rappresentata tramite gli operatori elementari; in pratica ci puoi fare qualunque cosa.

Utilizzando questo spazio è possibile passare da una scrittura ambigua ad una formale, chiara per quelli che saranno i nostri scopi; la rappresentazione dei sistemi avviene infatti tramite le tabelle di verità, che mostrano le funzioni booleane «Inserire immagine»

Nella tabella di verità viene definito un onset[1] ed un offset[2]. Ed i singoli elementi dell'onset si dicono *mintermini*, mentre quelli dell'offset sono *maxtermini*. La loro unione è complementare, poiché rappresentano tutto lo spazio usato dalla funzione.

Per mettere in relazione i bit con gli operatori si utilizza le seguenti espressioni: $m_3 = a \times b$, $m_0 = !a \times !b$. In merito, definiamo **letterale** una qualunque coppia variabile, Valore ed è l'unità di misura usata per definire la complessità di un circuito. Infine, la funzione in output si scrive attraverso una somma di prodotti o somma di min/maxtermini, per esempio: $O = abc + !ac + b!c$ e avremo una complessità di 7 letterali.

[1] Insieme dei punti dello spazio di ingresso dove la funzione vale 1
[2] Insieme dei punti dello spazio di ingresso dove la funzione vale 0

2.2 Realizzazione di porte logiche

2.3 Minimizzazione a due livelli

2.3.1 Mappa di Karnaugh

2.3.2 Algoritmo di Quine Mc-Cluskey

2.4 Funzioni parzialmente specificate

2.5 Sintesi combinatoria multilivello

2.6 Mapping tecnologico

2.7 Dispositivi programmabili

2.7.1 PLA - Programmable Logic Array

2.7.2 FPGA - Field Programmable Gate Array

2.7.3 SoC - System On Chip

Progettazione Digitale

3.1 Circuiti sequenziali

3.1.1 FSM - Finite State Machines

3.2 Sintesi delle funzioni λ e Δ, assegnamento stati

3.3 Minimizzazione degli stati

3.4 Datapath

3.4.1 Componenti del datapath

3.5 Modello FSMD - Finite State Machine with Datapath

3.6 Derivazione FSMD da algoritmo

3.7 Modello dispositivi programmabili

Architettura del Calcolatore

4.1 Modello di Von Neumann e Unità funzionali del calcolatore

Ebbene, sei ufficialmente sopravvissuto/a a metà del corso; rinunciare ora sarebbe disdicevole, non trovi? Minchiate a parte, parliamo del modello base di letteralmente ogni architettura di calcolatori: il Modello di Von Neumann.

Si tratta di un'architettura costituita da tre componenti interconnesse da un dispositivo chiamato **BUS**[1]:

- Processore; la Central Processing Unit, esegue le istruzioni.
- Memoria; composta da registri, salva le istruzioni e i dati.
- Dispositivi di I/O; periferiche come tastiere o altoparlanti.

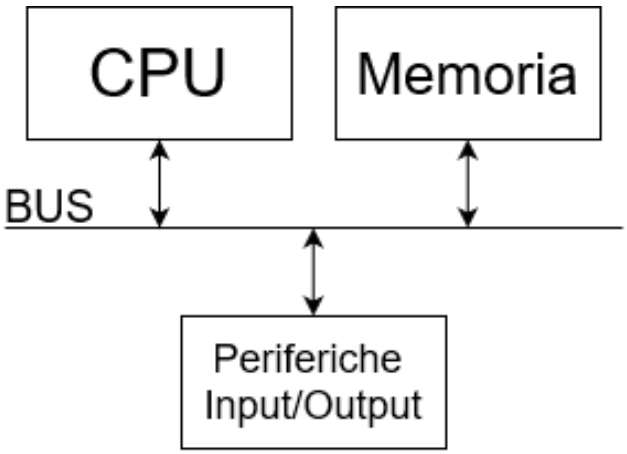


Figura 4.1: Modello Di Von Neumann

Un calcolatore è costituito dalle seguenti parti principali, indipendenti l'una dall'altra:

- Unità di ingresso ed uscita (I/O).
- Arithmetic Logic Unit (ALU).
- Control Unit, che compone processore e memoria.

Queste componenti lavorano sotto la supervisione ed il controllo della **Control Unit**; l'**Input Unit** riceve informazioni in forma codificata da operatori o periferiche come tastiere e vengono utilizzate dalla **ALU** per effettuare operazioni, eventualmente salvando in **memoria** i risultati. Quanto eseguito viene infine inviato all'**Output Unit** che ritornerà quanto eseguito.

[1]Flusso di bits che rende possibile la comunicazione fra le varie componenti

Le informazioni manipolate sono categorizzate in **istruzioni** e **dati**. Le prime sono i comandi dati alla macchina direttamente interpretabili da essa, mentre i secondi sono le informazioni che vengono manipolate.

Una lista di istruzioni compone il **programma**, il quale, se in esecuzione, si troverà *sempre* in memoria, a meno che non venga dato un comando di interruzione.

Il nostro ambiente di lavoro sarà la CPU Intel 80x86, la quale ha la particolarità di utilizzare lo stesso linguaggio del microprocessore: **Assembly**. L'insieme che compone le istruzioni scritte in tale lingua leggibili dal microprocessore si dice **Instruction Set Architecture**, (ISA), ed è letto da un **assemblatore**, il quale provvederà a tradurlo in codice oggetto.
Ma come è fatta una CPU? Andiamo a vedere nello specifico ogni componente. Consiglio di studiare attentamente la figura 4.2, così almeno ti fai un'idea.

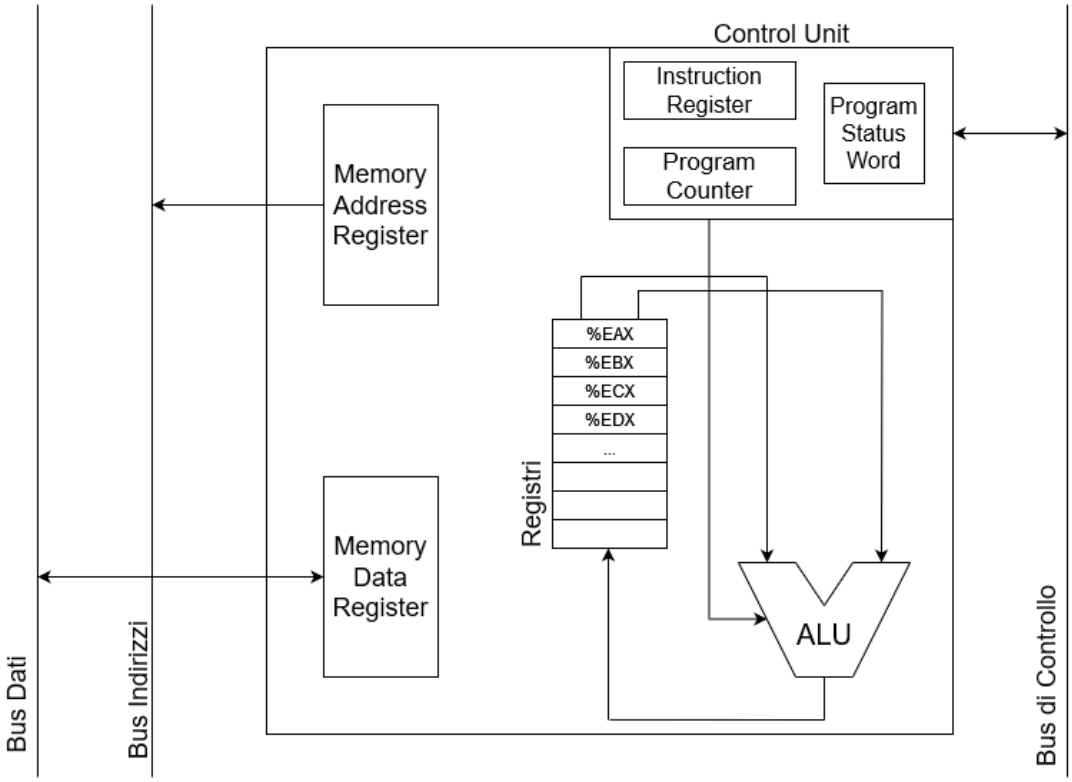


Figura 4.2: Architettura della CPU

- Componenti generali:

- *Bus Dati*; Consente di trasportare i dati fra le varie componenti.
- *Bus Indirizzi*; Comunica gli indirizzi di memoria delle informazioni.
- *Bus di Controllo*; Invia i segnali di controllo fra le varie componenti.
- *Memory Address Register*; Tiene in memoria e fornisce gli indirizzi dei dati da manipolare.
- *Memory Data Register*; Salva temporaneamente i dati da o per la CPU
- *Registri*; Salvano le informazioni elaborate.
- *ALU*; Esegue operazioni logico-matematiche con i vari dati.

- Componenti della Control Unit:

- *Instruction Register*; Contiene le istruzioni da eseguire.
- *Program Counter*; Contiene gli indirizzi delle operazioni da eseguire.
- *Program Status Word*; Insieme di flags che, in stretta collaborazione con la ALU, indicano lo stato dei diversi risultati di operazioni matematiche. Si modifica ad ogni singola operazione.

Tutto molto bello, ma come funziona? Ci è possibile descrivere tale processo mediante una FSM a tre stati, chiamati **Fetch**, **Decode** ed **Execution**.

1. *Fetch*; Fase di ricezione delle informazioni. Ottiene il necessario dall'MDR e lo passa all'IR mediante il Bus Dati. Il Program Counter aumenterà di 1 ad ogni istruzione.
2. *Decode*; Fase di decodifica delle istruzioni. Mediante l'ISA dell'architettura, indirizza opportunamente i dati per farli elaborare.
3. *Execution*; Esecuzione delle istruzioni decodificate.

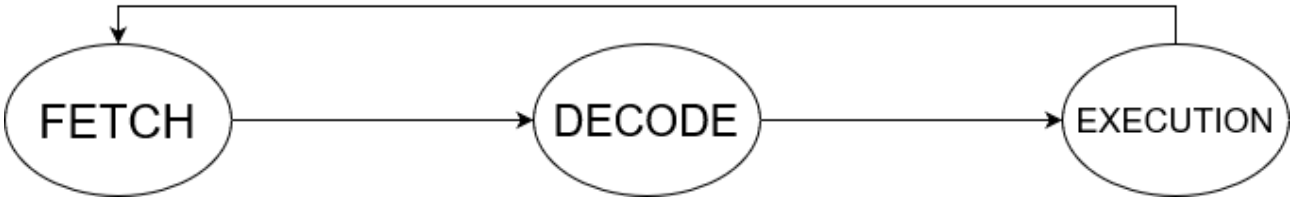


Figura 4.3: Macchina a stati della CPU

Un'ultima cosa di cui tener conto è la struttura in bit delle istruzioni dell'ISA, poiché ricorda che tutto ciò che legge la macchina sono segnali di assenza/presenza. Queste sono divise in tre parti:

- *OPcode*; Il codice operazione. Ci fa capire quante istruzioni possono essere registrate, ma soprattutto quale sta venendo effettuata.
- *Source Address*; L'indirizzo dal quale ottenere le informazioni.
- *Destination Address*; L'indirizzo nel quale verranno salvate le informazioni.

Abbiamo parlato molto di indirizzi finora e credo basti questo per capire quanto sia fondamentale questo concetto. Infatti per lavorare sui dati è necessario conoscere il loro indirizzo, ed è per questo che abbiamo ben *cinque* metodi di indirizzamento diversi:

- *Indirizzamento a registro* - [MOVL %EAX, %EBX]

  Sposta il contenuto del registro EAX in EBX.

- *Indirizzamento immediato* - [MOVL $8, %ECX]

  Sposta il valore 8 nel registro ECX.

- *Indirizzamento assoluto* - [MOVL DATO, %EDX]

  Non dissimile dalla dichiarazione di una variabile in C, richiede una lettura in memoria.

- *Indirizzamento indiretto a registro* - [MOVL (%EAX), %EBX]

  Interpreta il contenuto di EAX come un indirizzo e viene messo in EBX.

- *Indirizzamento indiretto a registro con spezzamento* - [MOVL $8(%EBX), %ECX]

  Somma il valore 8 al contenuto di EBX inteso come indirizzo, per poi salvarlo in ECX.

Questa parte del programma è fondamentale, perché se manca la comprensione del funzionamento della CPU, non sarà possibile comprendere appieno tutto ciò che concernerà la suddetta componente; ovvero tutto. La CPU fa tutto.

4.2 CPU - Central Processing Unit

Vediamo ora i due tipi di CPU che potremmo incontrare nel corso del nostro lavoro: le **CPU cablate** e le **CPU microprogrammate**. La loro differenza principale è che la prima si tratta di un circuito sequenziale generante segnali di controllo, mentre la seconda è un'unità contenente microistruzioni nella memoria di controllo per generare i relativi segnali... di controllo. Vediamole quindi nel dettaglio.

4.2.1 CPU Cablata

Il circuito della CPU cablata è creato mediante l'utilizzo di componenti elettroniche; ciò significa che nel caso in cui l'ISA dovesse essere modificato, lo stesso destino colpirà il cablaggio, rendendo questo modello *molto poco* flessibile ai cambiamenti. Tuttavia, il loro vantaggio è la velocità di elaborazione nel loro specifico compito e capirai di conseguenza che per i sistemi embedded sono perfette.

Le CPU cablate sono dotate di un **Bus Interno** con la capacità di trasportare dati da massimo 4Byte. Notare inoltre la presenza del **Tri-State Buffer**[2], il cui scopo è interrompere il flusso del segnale quando il Program Counter segna 0.

Il registro Y sottostante ad esso è necessario per selezionare l'operazione della ALU, ricevente i dati da elaborare nello stesso momento. Questi ultimi saranno salvati nel registro Z e mediante il Bus interno, spediti nel registro occorrente.

Il compito delle altre componenti è invariato da quello già trattato nello scorso paragrafo e non voglio in alcun modo ripetermi.

4.2.2 CPU Microprogrammata

Questa architettura non è rinomata per la sua velocità, tuttavia ha la capacità di svolgere compiti diversi. Come si suol dire; conoscitore di molti, maestro di nessuno. Il punto di forza della CPU microprogrammata è la sua *semplicità*, che rende facile non solo la sua progettazione, ma anche l'apporto di eventuali modifiche.

Il modello utilizza una *parola di controllo*, riferita ad una parola binaria. Si tratta dei classici insiemi di bits e vengono utilizzate per specificare le **micro-operazioni**.

[2]Nella figura 4.4, il cerchio con una croce al suo interno

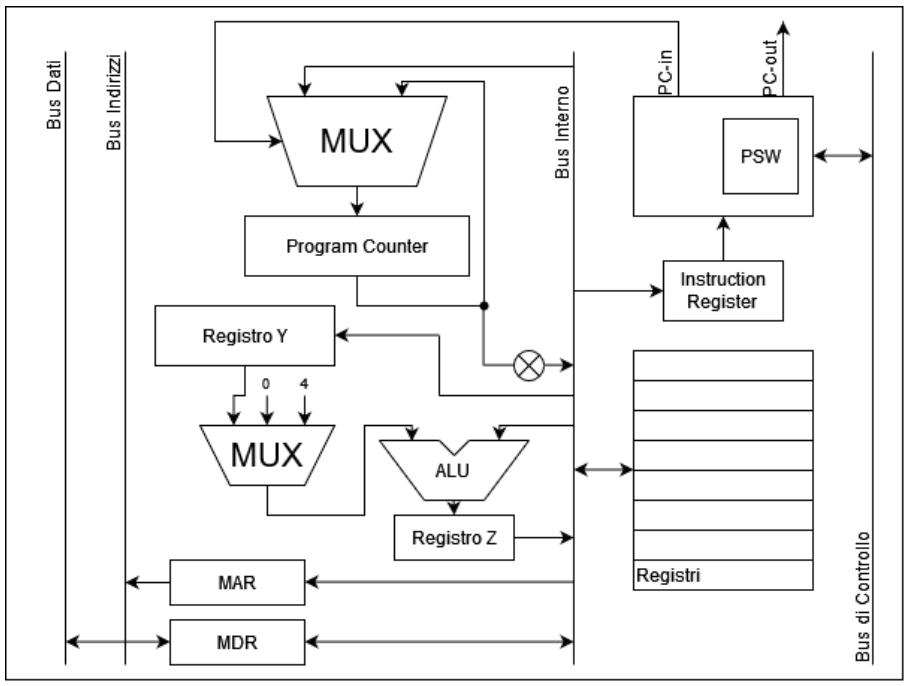


Figura 4.4: Modello di CPU cablata

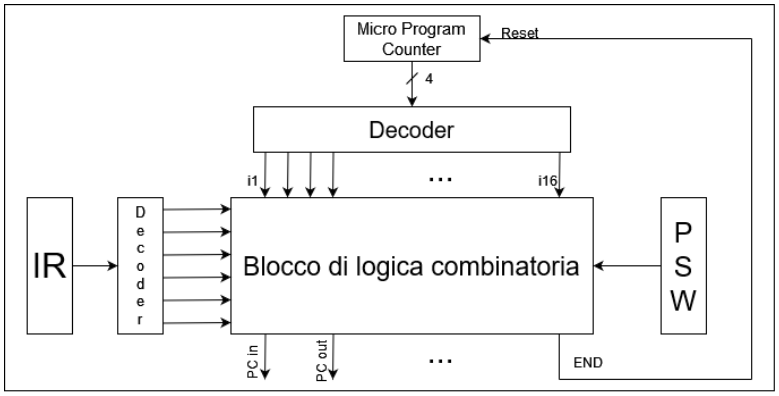


Figura 4.5: Modello di CPU microprogrammata

4.2.3 Microistruzioni CPU

Tempo di arrivare alla pratica delle micro-istruzioni. Gli esercizi consistono nella scrittura di ogni singola componente attiva assieme al compito che viene eseguito in funzione di una determinata architettura. In questo caso useremo la CPU cablata.

Le istruzioni sono lette sequenzialmente, ma ogni riga si esegue nello stesso ciclo di clock. Ciò significa che non importa l'ordine delle operazioni di una riga, ma bisogna stare attenti al loro insieme in tale posizione. Faccio davvero prima a farti vedere con ben quattro esempi.

1. **Indirizzamento a registro**: MOVL %EAX, %EBX

```
PC-in, MAR-in, READ, ADD, Z-in
WMFC, Z-out, PC-in
MDR-out, IR-in
EAX-out, EBX-in, END
```

Abbiamo scritto in totale quattro righe, dove le prime tre sono universali e fanno parte della fase di *Fetch* (per brevità indicherò queste linee con la parola "FETCH"), mentre l'ultima prevede la fusione di *Decode* ed *Execution* nello stesso ciclo di clock. Una riga corrisponde ad un clock tick, di conseguenza avremo CPI = 4.
In ordine, le istruzioni significano quanto segue:

1. PC incrementato di 1, MAR riceve i dati, lettura dei dati, istruzione "+" della ALU e salva il risultato nel registro Z.
2. Aspetta la fine della funzione in memoria[3], fai uscire i dati da Z (per poterli trasferire col bus interno) e PC incrementato di 1.
3. Invia i dati dall'MDR all'IR.
4. Invia il contenuto di EAX in EBX e termina il processo.

2. Indirizzamento indiretto a registro: MOVL (%EAX), %EBX

```
"FETCH"
EAX-out, MAR-in, READ
WMFC
MDR-out, EBX-in, END
```

Le due righe dopo FETCH appartengono alla fase di *Decode*, necessaria perché stiamo considerando il contenuto del primo registro come un indirizzo ed è quindi richiesto l'intervento del MAR per far sì che la macchina trovi il dato. Capirai che l'ultima riga rappresenta l'esecuzione. In tutto abbiamo CPI = 6.
Il significato di ogni singola istruzione dopo il fetch è:

1. Invia il contenuto di EAX al MAR ed effettua una lettura del dato in tale indirizzo.
2. Aspetta che finisca di leggere.
3. Invia quanto trovato dall'MDR al registro EBX e termina il processo.

3. Indirizzamento immediato MOVL $4, %ECX

```
"FETCH"
OFFSET_IR-out, Y-in
ECX-out, SELECT_Y, ADD, Z-in
Z-out, ECX-in, END
```

Qui non serve alcuna decodifica perché abbiamo un valore specifico, di conseguenza *Execution* viene subito dopo "FETCH". Spero tu abbia capito come contare i CPI perché non mi ripeterò più da ora.
Le istruzioni significano:

1. Prendi il valore dell'offset dell'IR ed inseriscilo nel registro Y.
2. Invia il contenuto di ECX alla ALU e usa come secondo operando il contenuto del registro Y. Effettuane la somma ed inserisci il risultato nel registro Z.

[3]WMFC, Wait Memory Function Complete è necessario a bloccare il clock della CPU nel caso servisse più tempo per effettuare un'operazione ed ottenerne i dati.

3. Invia il contenuto di Z al registro ECX e termina il processo.

4. Istruzione di salto (check se 0) alla fine: JZ END

```
"FETCH"
if zero == 0 -> END, OFFSET_REG-out, Y-in
PC-out, SELECT_Y, ADD, Z-in
Z-out, PC-in, END
```

Parto dal presupposto che questa istruzione è una carognata. La CPU se potesse ti sputerebbe in un occhio solo per aver scritto st'aborto. Detto questo, nemmeno qua è richiesta una decodifica. Il significato è tale:

1. Controllo se la parola "zero" equivale allo "0" e se questa uguaglianza è vera, termina il processo. Per questo controllo, invia l'offset del registro ad Y.

2. Invia il conteggio di PC alla ALU e sommalo al contenuto di Y, per poi salvare tutto in Z.

3. Invia il contenuto di Z al PC per spostarlo alla posizione desiderata e termina il processo.

Come dici? È un botto di roba? Ti farà sapere piacere che le microistruzioni compongono *sicuramente* un esercizio dell'esame, quindi richiamo all'attenzione la frase associata al libro per motivarti.

4.3 Metodi di input/output, segnale interrupt

I dispositivi di Input ed Output **I/O**, detti anche *periferiche*, consentono di effettuare uno scambio di dati dalla persona alla macchina e viceversa. Dei classici esempi sono tastiere, mouse ed altoparlanti.

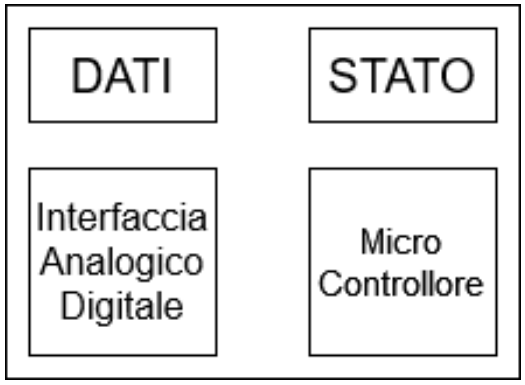


Figura 4.6: Dispositivo di Input/Output

Sono capaci di codificare l'informazione e mandarla al sistema mediante l'utilizzo di due registri da 1B l'uno. Osserviamo le loro componenti:

- **Micro-Controllore**; Piccola CPU dedicata al dispositivo. Supervisiona e controlla qualunque cosa si faccia.

- **Registro Dati**; Dove sono salvate tutte le codifiche della periferica. L'input è ricevuto attivamente e viene tradotto dall'interfaccia.
- **Registro Stato**; Effettua una funzione analoga al PSW ed esattamente come lui, ogni bit ha significato.
- **Interfaccia Analogico-Digitale**; Componente che traduce da segnale analogico a segnale digitale.

I dispositivi I/O sono **Memory Mapped**; ciò significa che *nella macchina intera* esiste un intervallo di indirizzi riservato a loro a cui rispondono i registri Dati e Stato. Nel caso in cui si provasse a far accedere la memoria in quei registri, la CPU si rifiuterebbe. L'unico modo per entrarvi è utilizzare gli accessi da *SuperUser* o *Admin*, dipendentemente dal sistema operativo che si usa.

Un esempio di istruzione effettuabile con gli indirizzi delle periferiche è il seguente:

```
CMPL $0, %EAX          #Controlla l'attivazione di Stato
JE TESTKEY
MOVL INDDATAKEY, %EBX  #Prende il valore ASCII dall'indirizzo del tasto
MOVL %EBX, INDC
```

Con queste linee di istruzioni stiamo effettuando una **SuperVisor Call**[4]. Ciò apre varie possibilità di personalizzazione, come la modifica degli output dei tasti in una tastiera. Se si vuole invece solo gestire le periferiche, si utilizza la tecnica del **Polling**[5], implicando che la periferica abbia una potenza simile se non uguale a quella della CPU. In caso contrario, verrebbero persi dati in corso d'opera.

Parliamo ora invece di **Interrupt**; un segnale asincrono che interrompe il lavoro della CPU, ricordando un neonato che piange ed il genitore che se ne prende cura.

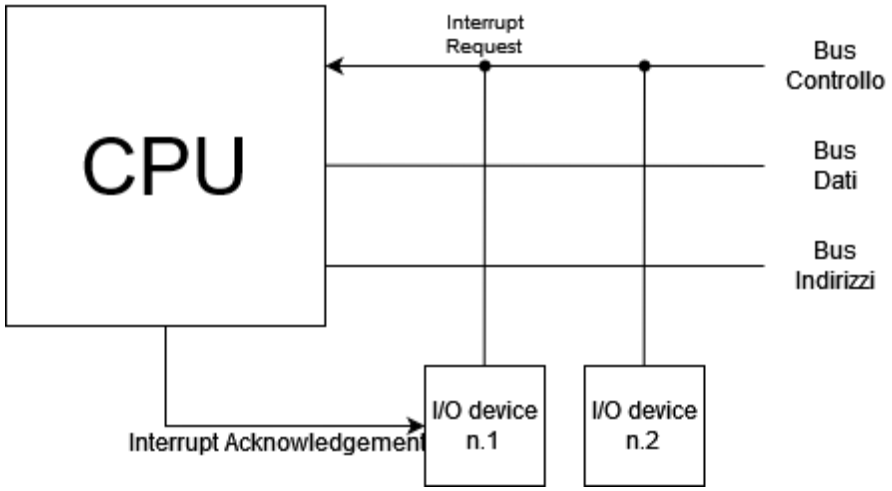


Figura 4.7: Funzionamento dell'Interrupt

Osservando la figura possiamo notare due elementi a noi nuovi:

[4]SVC, Chiamata per passare il controllo delle operazioni al sistema operativo

[5]Verifica ciclica dei dispositivi I/O mediante testing dei bits di bus di ogni periferica, seguita da un'interazione R/W.

- *Interrupt Request*; Un segnale posto sul Bus di Controllo normalmente posto a 1. Se è richiesto un interrupt, il valore diventa 0.
- *Interrupt Acknowledgement*; Segnale che conferma quale dispositivo ha richiesto un interrupt.

Mediante il secondo segnale, la CPU controlla sequenzialmente ogni dispositivo per capire quale ha abbassato il flag. Una volta trovato, chiama l'*Interrupt Service Routine* ad esso associato. Il suo compito è salvare le modifiche fatte a PC e PSW per poi interrompere il programma; infine interverrà il microprocessore per scaricare quanto appena salvato e tornare allo stato precedente. Tutto questo processo di controllo è parte integrante del **Device Driver**[6].

Un'ultima cosa da tenere in conto è che gli interrupt signals *non si sovrappongono*[7] e sono creati per gestire tempi umani. Se c'è un'alta frequenza di questi segnali è necessario trovare un sistema diverso. Ma tu guarda, è proprio la soluzione esplicitata nel capitolo seguente, che fortuna.

4.4 Direct Memory Access, BUS e arbitraggio

Iniziamo dando una visione più vasta del problema; è nostro volere trasportare una grande quantità di dati utilizzando il sistema appena visto con l'interrupt signal. Nel modello di Von Neumann avremo di conseguenza il seguente processo:

1. I dispositivi I/O ricevono l'input e lo inviano alla CPU.
2. Ricevuti i dati, si attiverà l'interrupt e si inizierà a lavorare con quanto ottenuto.
3. I dati elaborati sono salvati in memoria.

Non è di difficile comprensione il fatto che se l'interrupt signal è gestito in questo modo avremo uno spreco di risorse non indifferente ed è esattamente per questo che è stato ideato il concetto di **DMA**[8]. La CPU programma un dispositivo I/O, per far sì che questo possa accedere direttamente alla memoria senza passare da essa. Sarà inoltre in grado di eseguire operazioni di lettura e scrittura, grazie al suo micro-controllore e i suoi registri.

Il senso di tutto ciò è voler affaticare il dispositivo piuttosto che la CPU con lo scopo di risparmiare energia. Infatti, una volta finito il lavoro della periferica con DMA, il microprocessore della macchina dovrà ricevere un singolo interrupt. Tuttavia, ciò fa sorgere un ulteriore problema: più CPU vogliono accedere allo stesso BUS e se questo avesse luogo, i dati potrebbero venire persi o sfalsati. La soluzione sta in una gestione del BUS tramite un **arbitro**, il cui ruolo è tipicamente vestito dalla CPU principale. Sarà lei a scegliere chi e quando potrà accedere ed usare il BUS. Lo schema non è dissimile dal lavoro che esegue il singolo interrupt signal.
Osserviamo ora nel dettaglio una lista delle componenti ed i segnali nuovi presenti nella figura 4.8.

[6]Programma con lo scopo di ottimizzare e ridurre gli sforzi della CPU legati al funzionamento di una periferica.

[7]Tutte quelle volte che hai continuato a premere ESC o l'icona per chiudere la finestra hai sprecato tempo. In ogni caso era meglio aprire Task Manager.

[8]Direct Memory Access - Accesso diretto alla memoria

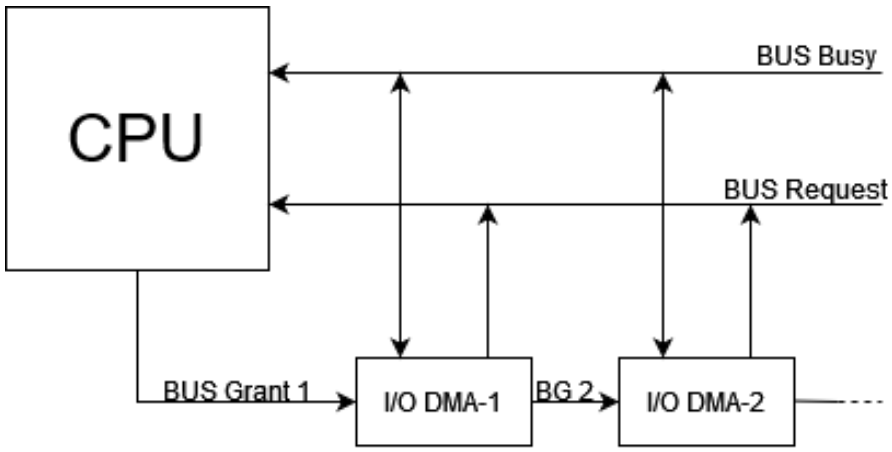


Figura 4.8: Arbitraggio BUS

- *BUS Busy*; Segnale che indica se il BUS è attualmente utilizzato da una componente.
- *BUS Request*; Segnale normalmente posto a valore 0, similmente all'interrupt, è una richiesta fatta alla CPU per poter utilizzare il BUS.
- *BUS Grant*; Segnale di concessione di utilizzo BUS una volta terminato il precedente lavoro[9].

Prima di andare nel dettaglio è necessario familiarizzare con due termini: **Master** e **Slave**, le entità sulle quali si basa il funzionamento del BUS. Il primo è l'iniziatore dell'operazione, mentre il secondo ne risponde. Esistono due protocolli di operazione che renderanno rispettivamente un BUS *sincrono*, dipendente da clock o *asincrono*, da esso indipendente. Negli schemi userò degli esagoni per rendere la scrittura più compatta. Se le linee si incrociano avremo un cambio di valore, mentre se sulla stessa riga v'è una linea sola è sinonimo di *alta impedenza*[10]. Vediamo meglio i due protocolli insieme ad un particolare funzionamento.

- BUS Sincrono
Le operazioni effettuabili sono le classiche lettura e scrittura.
La prima vede il Master nel fronte di salita ricevere il dato per poi farlo leggere, produrre ed inviare al BUS Dati dallo Slave nel fronte di discesa.
La seconda invece ha le medesime condizioni iniziali, ma lo Slave agirà prima per scrivere il dato.

- BUS dal funzionamento Multiciclo Questo utilizzo del BUS sincrono è quello che viene generalmente più utilizzato. Crea un ambiente relativamente solido per effettuare operazioni con la sicurezza di poter gestire ritardi o fallimenti, grazie ad un segnale aggiuntivo detto *Pronto*, che si attiva quando l'operazione è stata ultimata. Sceglieremo i cicli minimi che diranno quale coppia M/S è la migliore.

- BUS Asincrono
Qui abbiamo un doppio riscontro detto **Hand-Shaking** e due segnali MasterReady e SlaveReady. Di per sé non è complesso, ma aiuta molto guardare la figura durante la lettura.

[9]La CPU trasmette questo segnale in ordine fissato, da sinistra a destra. Per ottimizzare i tempi, è intelligente porre per prime le componenti più utilizzate dal sistema

[10]La parte non può agire poiché non riceve segnale

Diciamo di voler effettuare una lettura a tempo i, del quale importa meno di zero poiché siamo liberi da vincoli di clock. Deve ricevere prima il Master, che attiverà il segnale MasterReady e una volta ottenuti i dati, li invierà allo Slave che rimanderà tutto al capo come precedentemente visto. Finito di leggere, una volta confermato che le operazioni sono state completate tramite l'abbassamento di MasterReady, si potrà chiudere l'operazione, liberare il BUS e prepararsi per un'altra.

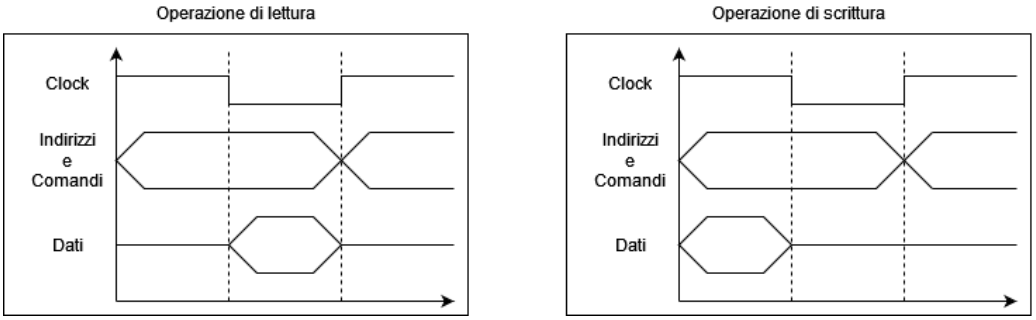


Figura 4.9: Operazioni in un BUS Sincrono

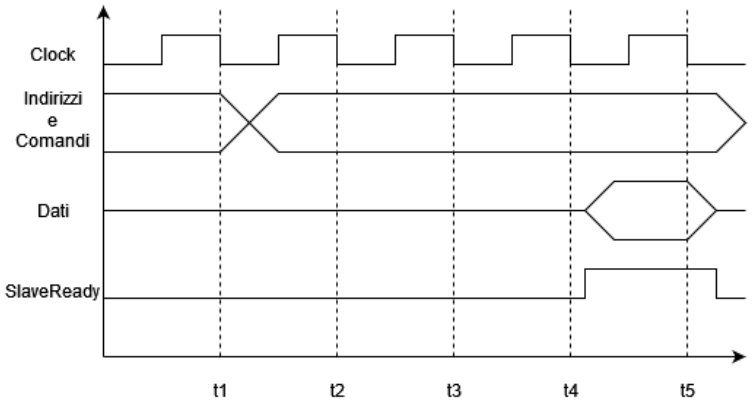


Figura 4.10: Funzionamento BUS Multiciclo

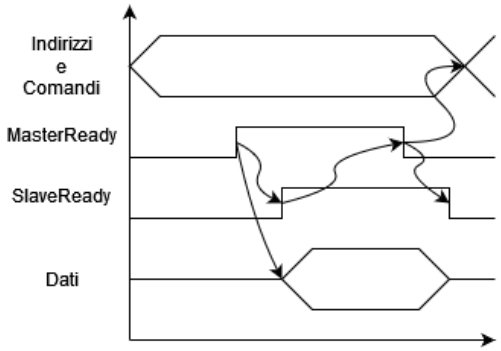


Figura 4.11: Lettura in un BUS Asincrono

Questi concetti sono fondamentali in quanto alla base degli elaboratori contemporanei sta il *Multitasking*, che è reso possibile grazie a quanto visto finora.

4.5 Stati di un processo

Se le istruzioni possono appartenere ad un solo processo com'è possibile che io riesca a procrastinare lo studio con ben quattro pagine di Youtube aperte? Questa è una buona domanda. Oltre a richiamarti perché devi studiare ti parlo del **Time Sharing**, un meccanismo di condivisione del runtime fra le varie operazioni.

Fondamentalmente si divide l'intervallo di tempo reale in vari sottointervalli di egual misura, la cui dimensione è 1Quanto, che corrisponde ad 1ms.

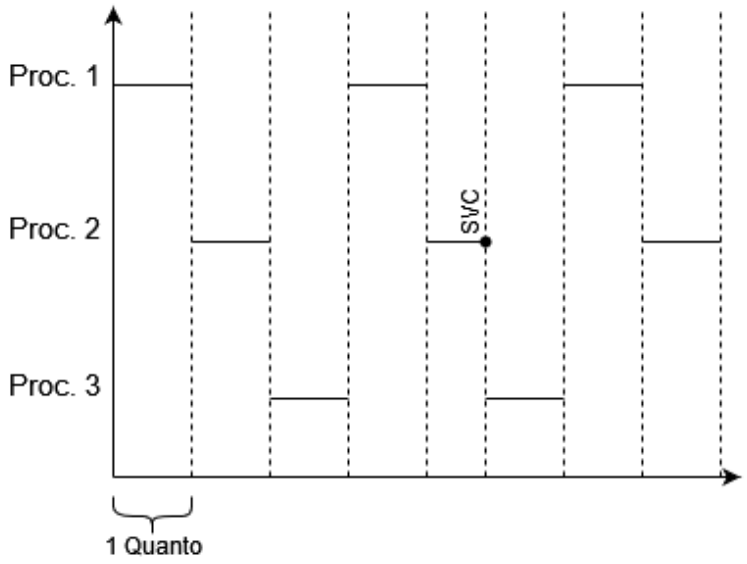


Figura 4.12: Grafico temporale di Time Sharing

Il funzionamento è il seguente:

Comincia il primo processo che ha a disposizione 1Q di tempo per lavorare; quando questo termina, si passa al processo successivo e così via fino al loro termine. Tuttavia, nell'eventualità di una *SVC*[11] il processo viene interrotto fino alla comunicazione dell'interrupt. In sostanza, la macchina non sta elaborando più processi allo stesso tempo, bensì uno alla volta ma ad una tale velocità che ciò risulta impercettibile.

Tutto questo bel meccanismo è gestito dal **Kernel** del sistema operativo, il quale vede i progressi di un processo in base allo stato in cui si trovano. Osserviamo i quattro stati della macchina.

- *ExecutionSystem*; Dato dalla CPU, è lo stato dove ha libero accesso.
- *ExecutionUser*; Dato dalla CPU, qui ha accesso limitato per far agire l'utente in caso di una SVC[12].
- *Waiting*; Stato dove si spostano i processi non terminati mentre la macchina ne esegue altri.
- *Ready*; Stato dove si trovano i processi interrotti ora pronti per essere eseguiti. Si basa sulla regola LIFO[13].

[11] Supervisor Call, richiamo del sistema operativo.
[12] Nell'ISA Intel 80x86 le SVC svolgono anche la funzione di interrupt signals e si indicano con "INT".
[13] Last-In, First-Out.

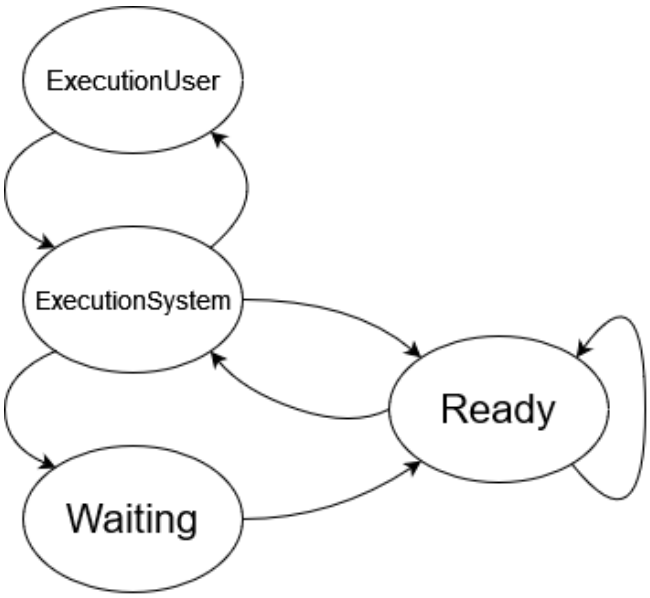


Figura 4.13: FSM degli stati di un processo

Mentre la CPU conta i cicli di clock dedicati agli interrupt signals, il sistema operativo può intervenire sui processi grazie allo **Scheduler**, il cui compito è decidere se dare ulteriore tempo ad un processo per interromperlo o terminarlo in base al tempo utilizzato in un quanto prima della SVC. Se è meno della metà, il sistema operativo aspetterà il prossimo ciclo di clock, altrimenti lo sposta in E-System. Questo meccanismo è detto **Preemption**.

Se uno scheduler lavora in tempo reale, i processi da lui gestiti si diranno *corretti* e produrranno un risultato giusto nell'intervallo di tempo giusto. In merito diremo inoltre:

- *Soft RealTime*; Se il processo è stato ultimato sforando di poco l'intervallo di tempo a disposizione.
- *Hard RealTime*; Se il processo è stato ultimato entro l'intervallo di tempo a disposizione.

Ogni singolo processo ha poi un suo descrittore che fa parte di una struttura dati del sistema operativo le cui parti sono: ProcessIDm, Proprietà, Stato della CPU[14], Cache e FileID. Questo tipo di strutture è salvato dallo scheduler e si dice **Context Switch**. Tuttavia, essendo che necessita del tempo, tutte le istanze in cui esso avviene sono tempi persi.

Quanto visto finora crea l'illusione per la macchina di avere una CPU per ogni singolo processo, idem per i dispositivi di I/O.

4.6 Pila e gestione del segnale interrupt

Avrai con ogni probabilità sentito parlare del termine "**Stack**". Conoscere il suo funzionamento è *fondamentale* per comprendere appieno come i programmi vengono trattati dalla macchina. Si tratta di una zona di memoria con due caratteristiche:

- *Ristretta*; Vengono selezionati rispettivamente un indirizzo di fine ed uno di inizio per delimitare lo spazio apposito per il programma. I loro valori sono salvati in due registri posti all'inizio e la fine di questa zona di memoria.

[14]Il salvataggio dello state nella memoria.

- *Riservata*; Ciò è necessario perché se altri processi dovessero accedere ad una zona di memoria già usata, si sfalserebbero o sovrascriverebbero i dati.

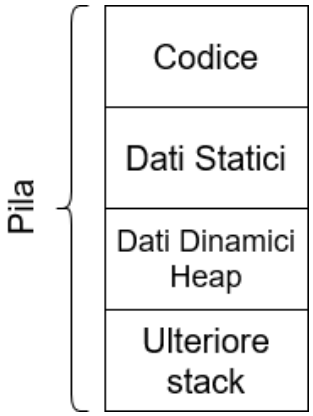


Figura 4.14: Divisione in parti della Pila

La Pila è divisa propriamente in quattro parti quando questa è ristretta per un programma:

- *Codice*; Registri per il salvataggio del codice scritto.
- *Dati Statici*; Registri per il salvataggio di costanti simboliche.
- *Dati Dinamici o Heap*; Registri per la memoria temporanea.
- *Ulteriore stack*; Il resto della pila non utilizzato per il nostro programma.

Tutto molto bello, ma che funzione svolge? È capace di allocare le variabili locali e passare parametri a funzioni, ed è proprio per quest'ultima abilità che ti voglio concentrato. Mediante l'utilizzo di *funzioni* è possibile ottimizzare l'utilizzo della memoria, in quanto lo spazio eventualmente creato per queste "vive" fino al loro termine, rendendolo riutilizzabile.

Detto ciò, esistono tre azioni *illegali* che portano direttamente ad errori fatali:

- Tentativo di accesso a zone di memoria al di fuori dello spazio creato per il dato processo; Se ciò accade, interviene il sistema operativo mediante una routine simil-interrupt per fermare il programma.
- Esecuzione di istruzioni che il processore non è abilitato ad effettuare; risulterà in un errore di Interrupt Service Routine.
- Lettura di sequenze di bits non integrate nell'ISA, di conseguenza irriconoscibili per la macchina; Avrà luogo una **TRAP**. Il microprocessore effettuerà una SVC per attivare l'ISR e fermare il processo. L'eventuale report sarà condiviso mediante registri.

Vediamo ora il comportamento della stack con un semplice programma in C.

```
void main(){

    int a, b, c;
    c = foo(a, b);
}
int foo(int d, int e){

    int f;
    ...
    return f;
}
```

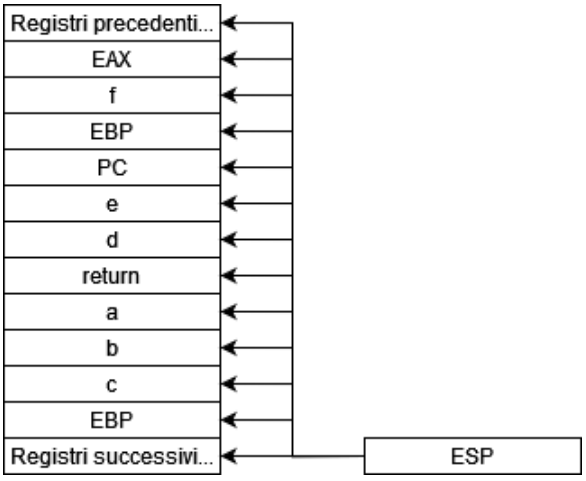


Figura 4.15: Posizionamento dei dati in Stack

Come puoi vedere, le variabili della funzione sono posizionate in registri precedenti a quelle presenti nel main; questo significa che lo spazio si crea all'indietro, ovvero con la regola **LIFO**[15]. Solitamente aiuta immaginare che la stack sia una pila di libri, dalla quale puoi rimuovere con facilità solamente l'ultimo impilato.

Questo meccanismo è ottenuto posizionando il Program Counter in cima alle celle di memoria allocate per il main, insieme ad una *necessaria* zona del main dove ritornare il valore elaborato, per evitare che si perda. Infine, terminata la funzione, lo spazio che è stato utilizzato deve essere liberato dall'utente.

Anche il sistema operativo ha una propria stack, dove sono presenti PSW e PC per effettuare le routines quando richiesto. Nel momento in cui termina, i registri nominati vengono scaricati.

Se non ricordi il significato dei registri qua presenti, invito a tornare indietro alla sezione apposita. Non c'è vergogna nel ripasso; non puoi sapere una cosa che credi di sapere.

[15]Last-in, First-out, ovvero Ultimo dentro, Primo fuori.

La Memoria

5.1 Tipi di memorie RAM - Random Access Memory

Tempo di conoscere il funzionamento di ciò che non si ha mai abbastanza: la **Random Access Memory**. Si dice ad accesso casuale perché il tempo di accesso ai registri è indipendente dalla distanza percorsa dai segnali. Immaginala; è composta da varie celle di bits organizzate e distanziate opportunamente entro un certo numero di bit, rendendole *indirizzabili*[1]; queste sono collegate orizzontalmente dalle **Word Lines** e verticalmente dalle **Bit Lines**, le quali sono collegate ad un circuito apposito per effettuare operazioni di lettura e scrittura.

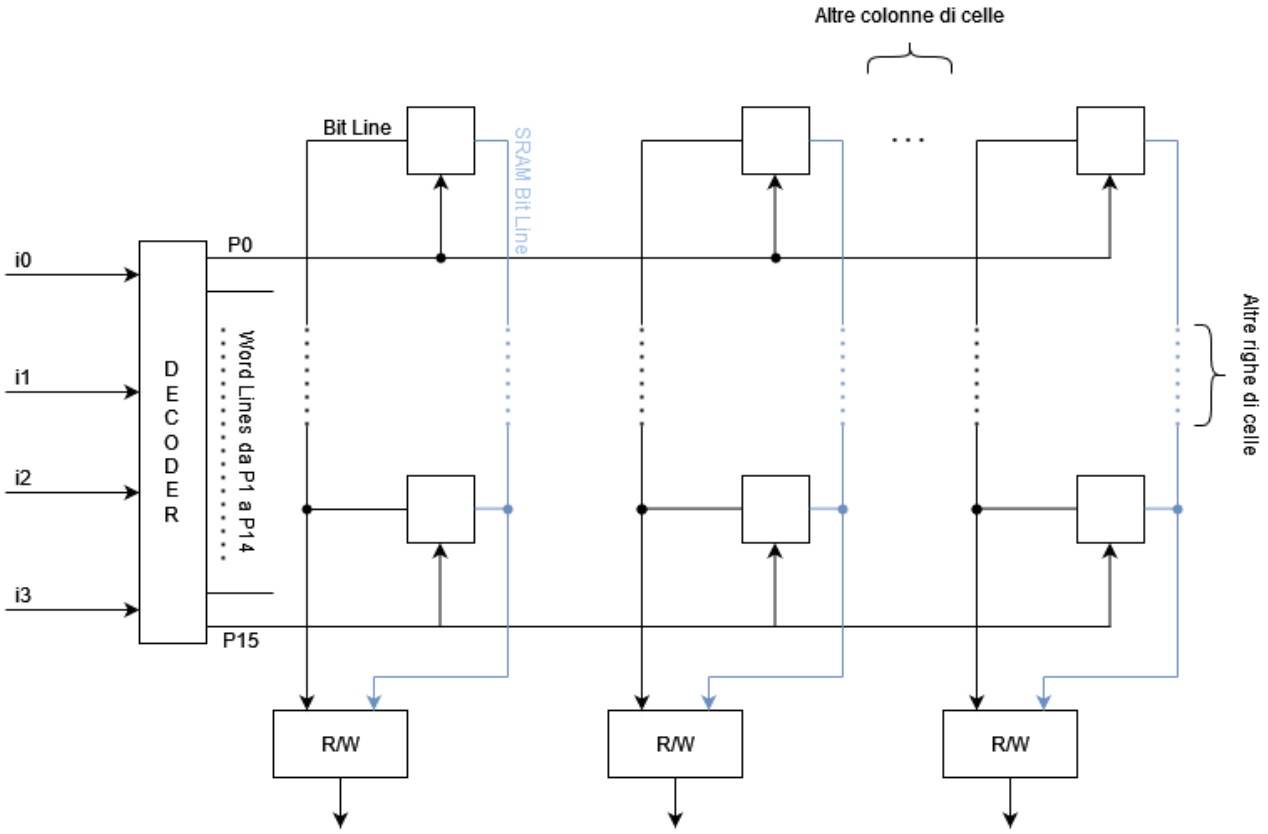


Figura 5.1: Organizzazione della memoria

Diciamo per esempio di avere 2^4 indirizzi, quindi quattro entrate nel decoder e sedici uscite, ovvero le Word Lines. Noi vogliamo poter attivare una singola cella fra tutte quelle presenti in memoria.

La Linea di Parola andrà a scegliere la cella corretta, la quale invierà un segnale ai circuiti R/W che produrranno in output il bit di interesse[2].

[1] Per accedere alla memoria è necessario sapere dove essa si trova, quindi conoscere il suo indirizzo.

[2] Questo segnale è di forma booleana, dove 0 significa scrittura, 1 lettura

Una caratteristica aggiuntiva del modello visionabile nella figura 5.1 è la presenza di un'ulteriore Bit Line apposita per la S-RAM, che vedremo meglio fra poco; ha lo scopo di amplificare la differenza nel segnale output di R/W[3]. Questo circuito è connesso direttamente alla scheda madre tramite pins dorati, il cui numero dipende dai seguenti tre tipi di segnale:

- *Di indirizzo*; Dipendono da quanta memoria è possibile indirizzare.
- *Di dato*; Dipendono dall'indirizzabilità.
- *Di controllo*; Fanno parte dei circuiti R/W e si chiamano **controlSignal** e **chipSelect**

Per esempio, se avessimo 4 indirizzi, 8 segnali di dato, 1 segnale di R/W ed 1 segnale di chipSelect avremmo un totale di 14 segnali, di conseguenza quattordici pins. Orsù, non indugiamo e andiamo a studiare nel dettaglio i vari tipi di memoria.

5.1.1 Static RAM

La memoria ad accesso casuale statica, **S-RAM**, è di tipo *volatile*[4] ed è composta da due semiconduttori di tecnologia CMOS. Sono presenti nel circuito i segnali *bit vero* e *bit falso*, che per comodità chiamerò b e !b; i quali vengono inviati tramite le apposite linee.

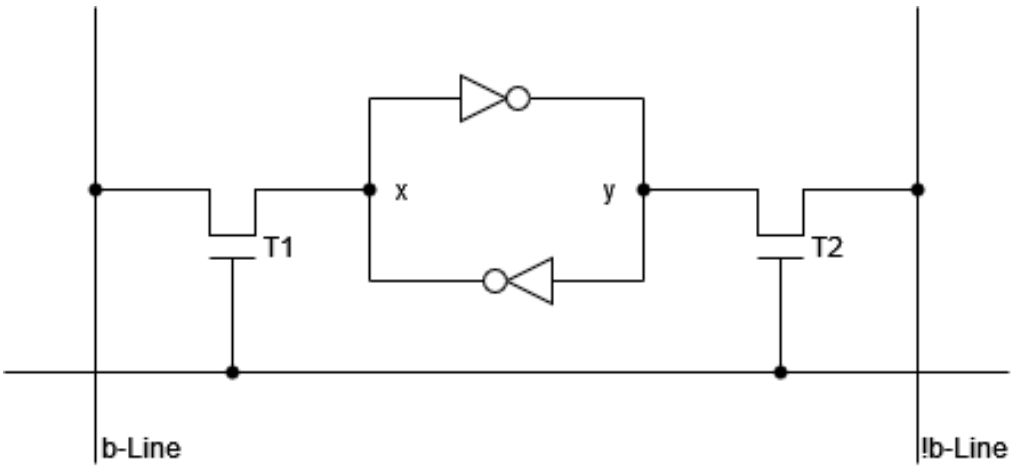


Figura 5.2: Modello di Static RAM

L'operazione da compiere è selezionata tramite due transistors. Nel caso in cui la Word Line abbia il valore 0, b e !b saranno isolati; se invece ha il valore 1, x otterrà il segnale b e y !b.

Per quanto veloce questa operazione possa essere, richiede un costo energetico importante, per questo che sono usate in quantità ridotte e posizionate puramente nella CPU. Vediamo ora un esempio del funzionamento della S-RAM, un probabile esercizio da esame:

Ci è assegnata una matrice 32x32 in bits, per un totale di 1024 bits. Noi vogliamo avere 10b di indirizzo ed 1b di dato.

Una prima cosa che possiamo dedurre sono le entrate del decoder; siccome la matrice riceve 32 segnali equivalenti a 2^5b, avremo 5 entrate. Questo gruppo consente di puntare ad una riga, ma è necessario trovare un metodo per selezionare la cella precisa.

[3]Si effettua una sottrazione. Se il circuito legge $1 - 0 < 0$, out = 1; altrimenti se $1 - 0 > 0$, out = 0

[4]Capace di mantenere il dato solamente se alimentata dall'elettricità.

Per far ciò si utilizza un blocco di logica di multiplexing il quale riceve il segnale di R/W e, date le altre 32 entrate, avrà un segnale di controllo a 5b, ovvero quelli rimanenti di indirizzo. Il blocco darà in output il bit preciso che è stato selezionato.

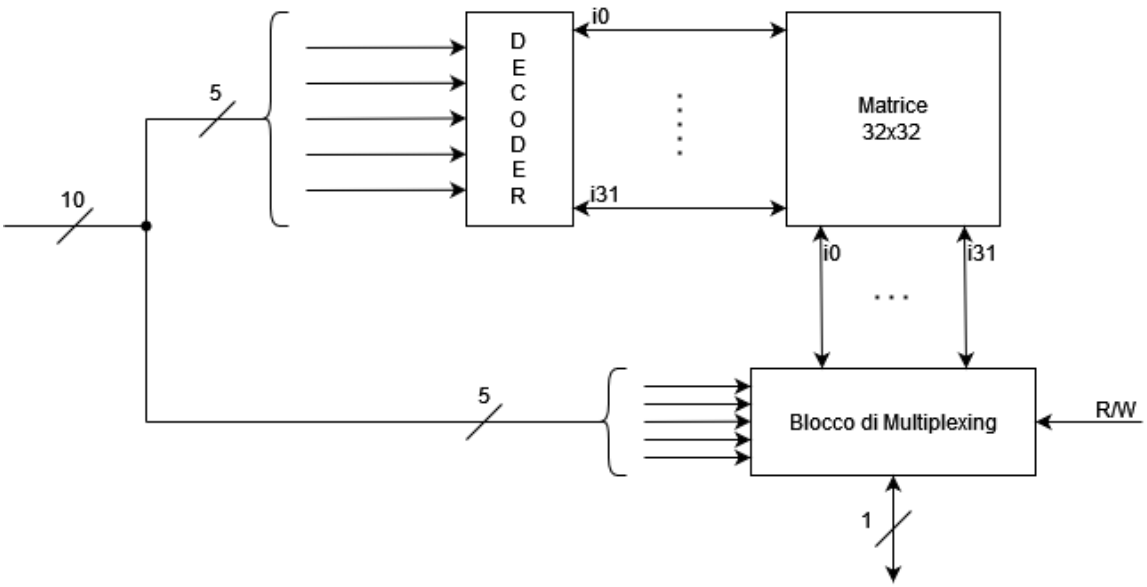


Figura 5.3: Funzionamento della S-RAM

5.1.2 Dynamic RAM

Nella memoria ad accesso casuale dinamica non è possibile memorizzare indefinitamente il contenuto di una cella poiché la memoria effettua un *refresh* periodico che libera lo spazio, indipendentemente dal fatto che una cella sia piena o meno. Dispone di un *condensatore*, un accumulatore di carica elettrica costante rispetto alla propria capacità; al suo interno si trova dell'isolante elettrico, il quale rilascia il segnale ad ogni refresh.

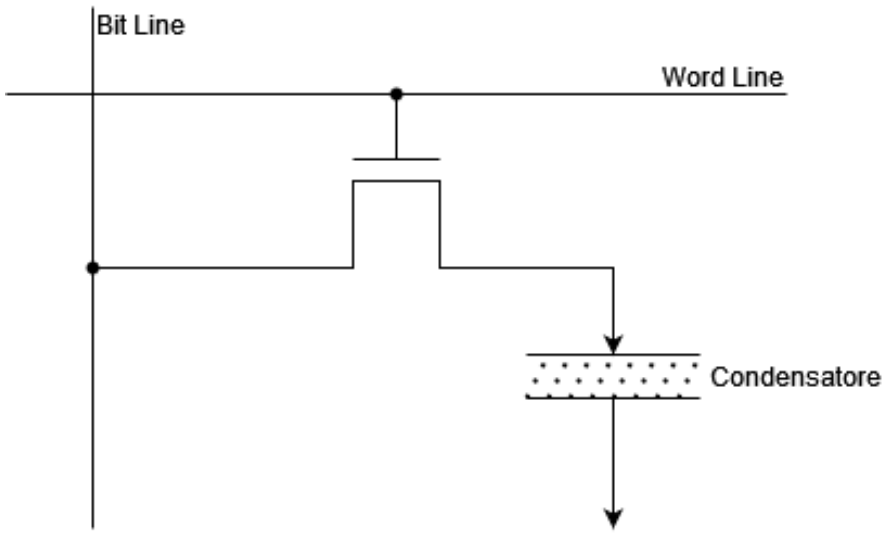


Figura 5.4: Modello di Dynamic RAM

La D-RAM risulta più lenta della S-RAM perché si scarica anche se isolata e di conseguenza in caso di necessità deve essere nuovamente riempita, specie perché è nostra intenzione preservare il contenuto il più possibile. Anche la sola operazione di lettura è distruttiva, perché se si vuole accedere al contenuto è necessario liberarlo. Essendo poi una memoria più compatta e capiente, è molto probabile interfacciarsi con dimensioni di bit molto grandi con di conseguenza molti più segnali. Vediamo un esempio del funzionamento:

Diciamo che un refresh avviene ogni 64ms, il tempo minimo per accedere ad una riga è 5ms e tutte le righe saranno refreshate ogni 8192 cicli di rinfresco. Possiamo ricavare i *millisecondi necessari per effettuare un refresh* con $8192 \times 0,05 = 0,41\text{ms}$ ed il *costo del refresh* con $\frac{0,41}{64} = 0,0064\text{ms}$.

La nostra matrice questa volta ha $16384 = 2^{14}$ righe per $2048 = 2^{11}$ bytes; avremo 14b in entrata nel decoder e 11b di selezione nella logica di multiplexing. I bits dell'indirizzo totali saranno $14+11 = 25$ ed i pins da utilizzare saranno la somma fra tutti i segnali: $25+8+2 = 35$. Il fatto è che questi ultimi sono tanti e richiederebbero costi importanti, ragion per cui è stata ideata la tecnica di **strobing** dell'indirizzo, un piccolo sacrificio della velocità per consentire un risparmio in oro.

Sostanzialmente si inviano le due parti dei bits di indirizzo in istanze diverse; questa strategia è resa possibile grazie a due registri che consentono il passaggio dei dati solo in presenza del proprio segnale, chiamato **RAS**[5] per le righe e **CAS**[6] per le colonne.

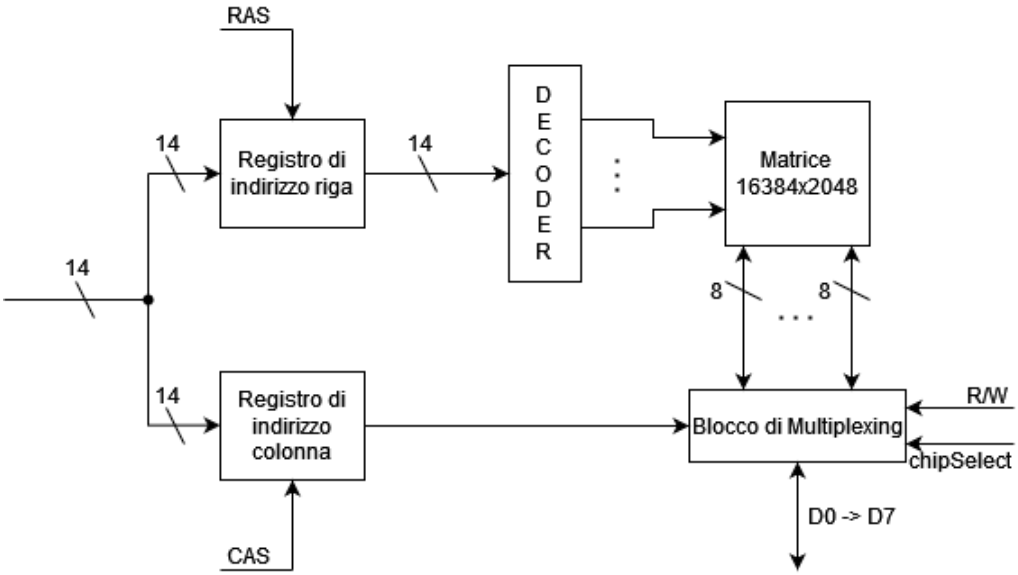


Figura 5.5: Funzionamento della D-RAM

Esistono inoltre le **SD-RAM**[7], le quali svolgono la medesima funzione ma sono governate da un clock. La particolarità di questa memoria è la sua capacità di velocizzare i processi in caso di accessi consecutivi; ciò sinergizza con il *Principio di Località*, il quale vedremo meglio in seguito nella gerearchia di memoria, ed è per questo che le SD-RAM vengono posizionate nella CPU, risparmiando anche sul costo dei pins.

[5]Row Address Strobe
[6]Column Address Strobe
[7]Synchronous Dynamic RAM - RAM Dinamiche Sincrone

5.1.3 Banchi di memoria

Un ultimo modello molto importante è rappresentato dai **Banchi di memoria**; insiemi integrati di memoria collegati fra loro per condividere Bit Line e Word Line. Ci si può trovare in una situazione dove risulta comodo modificare a piacimento bits ed indirizzi e questo modello si presta particolarmente bene.

L'architettura presenta i soliti bits di indirizzo per selezionare la riga e quelli inviati al decoder per effettuare il chipSelect e selezionare la cella esatta. Eseguita la loro istruzione, comunicheranno quanto elaborato mediante un BUS apposito che collega tutte le celle di una colonna. Vediamo ora un esempio pratico:

Diciamo di avere componenti da 512K indirizzi a 8b, ma per nostra comodità vogliamo creare un banco di memoria a 32b per 2M indirizzi. Come fare?

Iniziamo col capire di quante celle abbiamo bisogno per ogni riga; sappiamo che $8 \times 4 = 32$, quindi serviranno quattro celle da 512K per 8b, e questa è già una parte. Noi però necessitiamo di 2M indirizzi.

Se ricordi le grandezze in dati, sai che $1M = 1024K$ e che quindi $2M = 2048K$. Avremo bisogno di quattro righe totali.

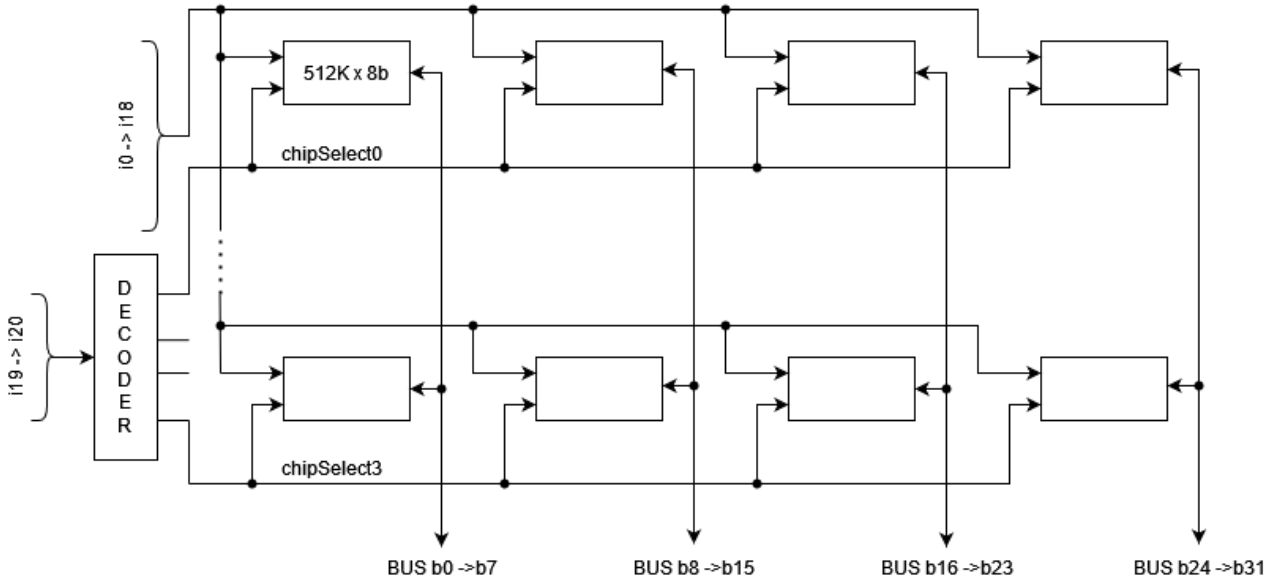


Figura 5.6: Banco Di Memoria

5.2 Caratteristiche delle memorie e relativa gerarchia

Andiamo ora ad osservare ulteriori tipologie di memoria insieme alle loro caratteristiche. Oltre a registri, cache e RAM, abbiamo:

- Solid State Disk

Questo tipo di memoria è quello utilizzato oggigiorno in qualunque computer o console per garantire la massima velocità in relazione alla propria struttura. Appartiene all'insieme delle memorie *flash*, usate anche per la creazione di schede SIM e chiavette USB, che agiscono come estensioni del BUS, dal quale ovviamente ricevono i dati.

Sono divise in blocchi capaci di contenere un determinato numero di parole e le informazioni sono scritte in modo circolare; vediamolo nel dettaglio:

Il blocco con le informazioni viene interamente copiato in un *buffer* che scriverà i dati sulla RAM, la quale invierà tutto ai blocchi della memoria flash. Quando uno finisce lo spazio, si passa al successivo.

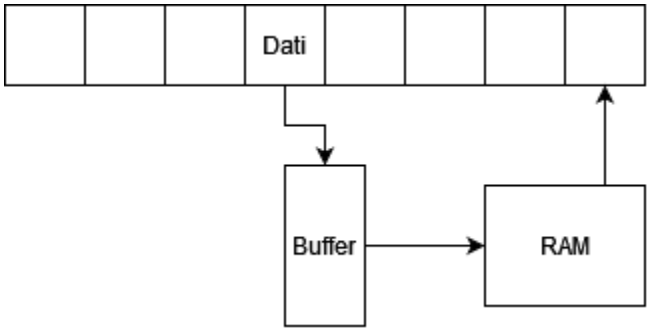


Figura 5.7: Memoria Flash

Questo processo è reso possibile dai **droganti**; elementi che muovendosi nella memoria sono capaci di modificare la struttura interna del silicio. Dove questo però consente la scrittura, fa sorgere un nuovo problema: l'**isteresi**.

Per effettuare modifiche al materiale è necessario rimuoverne dei pezzi, poco a poco; questo significa che più questa cosa avviene, più la memoria risulterà inaffidabile, fino ad arrivare ad un punto dove sarà inutilizzabile. Dal primo utilizzo, se il dispositivo non è difettoso, vedrà la sua probabilità di rottura diminuire fino ad arrivare al punto **MTBF**[8], dove da lì in poi aumenta esponenzialmente.

Abbiamo inoltre a disposizione nel nostro arsenale una legge particolarmente utile per i nostri scopi di ottimizzazione hardware: Il principio di località, ramificato nei seguenti due concetti:

- *Località spaziale*; Quando una cella di memoria è utilizzata è altamente probabile che vengano usate anche quelle ad essa vicine.
- *Località temporale*; Quando una cella di memoria è utilizzata è altamente probabile che venga usata nuovamente di lì a poco.

- Hard Disk

Questo è un tipo di memoria molto importante, che sta venendo tuttavia sostituito col tempo a causa di alcune sue caratteristiche negative; come sono fatti?

L'architettura presenta un albero che ruota ad una certa velocità costante e sposta allo stesso tempo un piatto dalla superficie magnetica composta da tanti piccoli magnetini separati in diverse aree, magnetizzati in modo diverso e posizionati in tal modo da consentire la manipolazione di dati. Gli Hard disks sono poi divisi in tracce che comunicano i loro rispettivi punti di inizio e di fine. Queste sono a loro volta separate in blocchi equamente distanziati.

Questa era la loro struttura, ma come funzionano? Le operazioni di lettura e scrittura avvengono mediante forze magnetiche. Sopra il piatto è presente un braccio con una spira che, se elettrificata, ha la capacità di emettere un campo magnetico. Nel caso della lettura passa solamente sopra ai magnetini coi dati desiderati, mentre per la scrittura è necessario il suo campo magnetico poiché è necessario cambiare l'orientamento dei magnetini toccati.

[8]Mean Time Between Failure - Tempo medio fra fallimenti

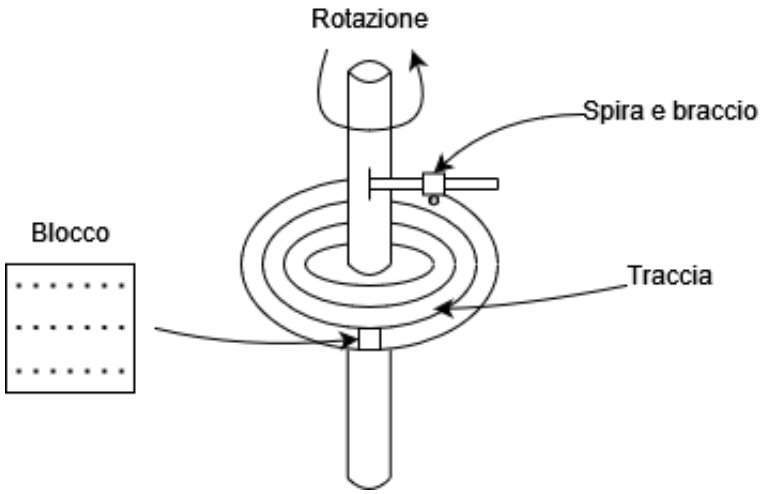


Figura 5.8: Architettura di un Hard Disk

La spira non deve per nessuna ragione toccare il disco perché è stata creata per stare a debita distanza. Se questa dovesse restringersi o allontanarsi non sarebbe più possibile effettuare le operazioni di lettura e scrittura e quindi usare i dati contenuti nella memoria.

Prima di un'operazione, tuttavia, agisce un buffer il cui tempo di lavoro totale è dato dalla seguente equazione: $T_{Totale} = T_{RicercaMagnete} + T_{Rotazione} + T_{LetturaBlocco}$. Una volta terminato il suo lavoro sarà possibile effettuare l'operazione richiesta. Bisogna infine farsi domande sull'algoritmo per l'organizzazione dei files.

Partiamo dal presupposto che i files sono composti da una catena di blocchi; la **FAT**[9] propria del sistema operativo riceve in input il nome del file ed userà un puntatore per spostarsi sulla catena che costituisce il primo allo scopo di leggerla tutta; finita questa operazione verrà tutto scritto sul disco nella catena dei blocchi liberi. Invece nel caso in cui si cancellassero dei files, i loro blocchi verranno inseriti nella suddetta catena per poter essere eventualmente sovrascritti. Fai attenzione a ciò che salvi, poiché la sovrascrittura non implica la completa cancellazione delle modifiche fisiche della memoria.

- Compact Disk e Digital Versatile Disk

La nostra penultima memoria da studiare è il supporto ottico. Come funzionano?

I CD sono unità di memoria nate per il salvataggio di brani musicali i cui dati sono salvati sotto forma di codice binario in una singola traccia a spirale. Le informazioni vengono lette da un laser che viene riflettuto in un apposito lettore. Per direzionare il raggio viene utilizzato uno specchietto che rende possibile coprire ogni zona del disco ottico. Allo scopo di effettuare l'operazione di scrittura, invece, si utilizza un laser più potente in grado di creare buchi nel disco, i quali verranno eventualmente letti tramite il laser normale.

Sorse poi la domanda: "Come migliorare ulteriormente questa tecnologia per salvare ancora più dati?"; la soluzione si rivelò in gel e luci per il laser di colori diversi. I dischi sono cosparsi del suddetto materiale e si scrive su di essi con luce blu, con frequenza maggiore, e rossa, con frequenza minore. Lasciando il colore nello strato in modalità diverse, il laser a luce bianca potrà leggere quanto scritto e distinguere fra le due scritture. Con questa idea nacquero i **DVD-ROM**[10] ed i **DVD-RW**[11].

[9] File Allocation Table - Tabella di Allocazione dei File.
[10] DVD Read Only Memory - DVD sola lettura
[11] DVD Rewritable - DVD Riscrivibile

- Digital Audio Tape
Tecnologia di vita breve partorita da Sony. Raramente sono ancora utilizzati per effettuare dei backup nonostante il loro funzionamento fastidiosamente *sequenziale* per le operazioni di lettura e scrittura. Facciamo direttamente un esempio; mettiamo caso di avere un nastro scritto al 95% e riteniamo necessario leggere o scrivere in una posizione precedente. Qui sarà necessario riavvolgere tutta la memoria fino al punto desiderato. Non è molto efficiente.

Lascio ora una tabella riassuntiva di quanto visto, aggiungendo ulteriori caratteristiche.

Memoria	Tecnologia	Accesso	Velocità [s]	Dimensioni [B]	Costo $[\frac{€}{B}]$
Registri	CMOS, elett., volatile	Datapath	10^{-9}	10^{4}	10^{-3}
Cache	Statica, elett., volatile	Associativo	10^{-8}	10^{7}	10^{-6}
RAM	Dinam., elett., volatile	Casuale	10^{-7}	10^{10}	10^{-8}
SSD	Elett., flash, non vol.	A blocchi	10^{-5}	10^{12}	10^{-10}
HD	Magn., mecc., non vol.	Diretto	10^{-3}	10^{13}	10^{-11}
CD - DVD	Ottica	Diretto	10^{-3}	10^{10} o 10^{13}	10^{-9}
DAT	Magnetica	Sequenziale	10^{0}	10^{9} o 10^{11}	10^{-9}

Tabella 5.1: Memorie a confronto ordinate in base alla velocità

5.3 Memoria Cache

Parliamo ora della **Memoria Cache**, dalla velocità assurda. Nella gerarchia di memoria è seconda solamente ai registri, si trova nella CPU e se le informazioni sono in essa presenti, hai la garanzia che si eseguano in un solo ciclo di clock, ottenendo il tuo tanto bramato 1CPI.

Da grandi velocità derivano grandi responsabilità ed è per questo che nella cache stanno dati ed istruzioni importanti se non fondamentali. Sappiamo che la cache è meno capiente della memoria RAM, creando i concetti di:

- Cache Hit; L'informazione è presente in cache e viene elaborata in 1CPI.
- Cache Miss; L'informazione non è presente in cache e la pagina deve essere recuperata dalla RAM e messa nella prima memoria. Alla CPU saranno fornite le parole.

Una cosa che bisogna tenere a mente è la dimensione della singola parola: 1Byte = 8bit, dalla quale è possibile ottenere la dimensione in bit degli indirizzi di memoria. Diciamo di avere 4GB di RAM, dovrai usare la seguente formula: $\frac{4GB}{1B} = 4G$, ovvero 4GigaParole.

Nella definizione delle pagine è possibile decidere la grandezza, caldamente consigliata essere una potenza del 2. Diciamo che una pagina ha il valore di 4KB. Dividila per la dimensione della RAM e della cache per ottenere quante pagine possono essere contenute nelle due memorie. La formula è simile a quella di prima: $\frac{4GB}{4KB} = 1M$ in RAM, $\frac{4MB}{4KB} = 1K$ in cache.

Quanti bits mi serviranno per indirizzare una pagina? $4 \times 2^{10} = 2^{12}$. La risposta è 12. Temo sia ora di imparare le misure in bit, se così tu non abbia già fatto.

Le pagine presenti nella cache si copiano mediante un indirizzamento dalla stessa e possiamo vedere il suo indirizzo come se fosse diviso in bit. In questo caso parliamo di 22bits totali, dove 12 servono, come visto poco fa, per indirizzare la pagina, mentre i restanti 10 per la parola.

Di indirizzamenti ne esistono di tre tipi: diretto, associativo e set-associativo, il più usato. Approfondiamoli uno per volta.

- Indirizzamento diretto[12] $\frac{\text{N.PagineRAM}}{\text{N.PagineCache}} = \mathfrak{x}[\mathfrak{b}]$
In questo metodo abbiamo che ogni singola pagina della RAM ha una sola posizione in cui andare nelle pagine cache. Noi vogliamo sapere in che parte della cache il segnale va a finire.

Qui si rende necessario dividere i 32bits della RAM in tre parti: 10bits per indirizzare la parola, 12bits per indirizzare la pagina e gli ultimi 10bits per l'etichetta della pagina.

Riceviamo i bits dal MAR per poi metterli nella RAM, dopodiché, si effettuerà un check nel vettore sovramenzionato. Se la pagina corrisponde a quella ottenuta dalla RAM avremo un cache hit, metteremo insieme bits di pagina coi bits di parola e accederemo alla cache. Se la pagina è diversa avremo un cache miss ed il microprocessore andrà in RAM per prendere la pagina richiesta e sostituire quella precedente.

- Indirizzamento associativo[13]
Con questo indirizzamento non abbiamo il lusso di sapere dove va una data pagina, perché verranno controllate tutte fin quando non si troverà quella corrispondente (immaginati i cache miss se ti serve l'ultima pagina). Vengono tenuti i 10bits per la parola, ma i restanti 22bits saranno usati per l'etichetta. Una volta trovata, si darà l'indirizzo in cache.

- Indirizzamento set-associativo $\frac{\text{N.PagineCache}}{\text{N.PagineInSet}} = \mathfrak{totSets}$
Questo metodo è una sinergia fra i due appena visti. Fondamentalmente ci è possibile raggruppare delle pagine in degli insiemi, sets, per poter effettuare il controllo delle etichette direttamente in tal gruppo. Nella divisione dei bits dell'indirizzo in RAM, oltre ai 10bits per la parola, al posto della pagina, avremo, dipendentemente da quanti insiemi sono presenti in memoria, un numero $\mathfrak{n}$bits per i sets ed i rimanenti verranno usati per l'etichetta.

Tutto molto bello ma c'è un ultimo problema: mettiamo caso di avere un cache miss e di non avere più spazio in tal memoria. Bisogna trovare tale **algoritmo di sostituzione** per scaricare delle pagine ed inserire quelle richieste. Ne esistono due e riguardano rispettivamente lo scarico della pagina più vecchia, **LRU**, Least Recently Used, o la più recente, **MRU**, Most Recently Used, e funzionano grazie alla presenza di un contatore in ogni pagina che aumenta ad ogni ciclo di clock. La comodità dell'algoritmo dipende dallo scopo della macchina.

5.4 Memoria Virtuale

Iniziamo ora un ulteriore approfondimento sulla memoria presentando due processi utili per i nostri scopi:

- Rilocazione; Strategia che permette di usare la stessa RAM senza sovrascrivere i registri
- Paginazione; Strategia che permette a più processi di utilizzare efficientemente la stessa RAM

Questi due processi ci consentono di dare l'impressione al sistema operativo di avere a disposizione tutta la ram dell'architettura, ovvero di **virtualizzare la memoria**.

[12] Vedi figura in merito; equivale ad un singolo indirizzamento set-associativo
[13] Corrisponde ad 1set-associativo, quindi come se tutte le etichette fossero un unico set.

Partiamo da un semplice presupposto; le varie istruzioni dei codici devono avere un indirizzo per poter andare nelle zone di memoria della Stack, Heap etc. e questo viene loro assegnato in ordine crescente dall'assemblatore o dal compilatore, dipendentemente dal linguaggio nel quale si sta scrivendo.

Le parole verranno inserite nello spazio di memoria appositamente creato per il programma, quindi avremo una parola in i_1, un'altra in i_2 e così via fino all'indirizzo massimo. Tuttavia al momento non sai dove questo intervallo di indirizzi verrà posizionato.

Qui entra in gioco la **Rilocazione**; trasforma gli indirizzi logici in indirizzi fisici, i quali verranno caricati sul BUS indirizzi. Si tratta di un processo necessario in quanto per poter accedere ai dati serve l'indirizzo fisico. Abbiamo due modi per effettuare la rilocazione: statica e dinamica.

- Rilocazione statica

Questa strategia è usata puramente per i sistemi la cui struttura è statica, quindi per i sistemi embedded.

Diciamo che un programma deve partire dall'indirizzo 1438; a questo numero saranno sommati tutti gli indirizzi logici dei dati del codice, così da non sovrascrivere i registri precedenti. Eventualmente, se sono presenti delle condizioni if o cicli, il codice si biforcherà dipendentemente dalla loro quantità. Se non fosse per il compilatore o l'assemblatore, in quanto sono loro a trovare e segnare le parti dei bits che rappresentano l'indirizzo, questo algoritmo non avrebbe mai visto la luce.

- Rilocazione dinamica

Questa è la strategia utilizzata nei sistemi programmabili, quindi quella di nostro maggiore interesse.

Serve a ottenere un indirizzo fisico da uno logico ed entrare nel MAR. Per fare ciò abbiamo due registri *base* e *limit* che contengono rispettivamente la prima e l'ultima zona di memoria creata per il programma. Questi lavorano insieme ad un circuito di *testing* che controlla se la somma fra indirizzo logico e base è compresa nella zona di memoria apposita. Se lo è, crea l'indirizzo fisico e lo invia al MAR, altrimenti si ha un errore di Segmentation Fault.

C'è tuttavia un problema in questi metodi. Immagina di voler caricare tre diversi programmi: P_1, P_2, P_3, i quali verranno posizionati in ordine nelle rispettive zone di memoria.

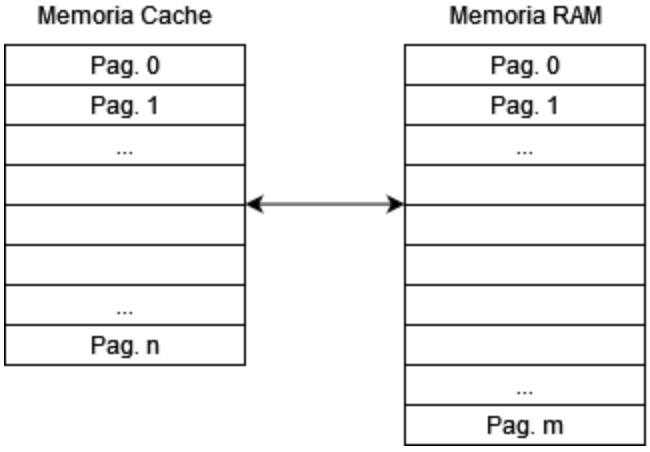


Figura 5.9: Comunicazione fra RAM e cache

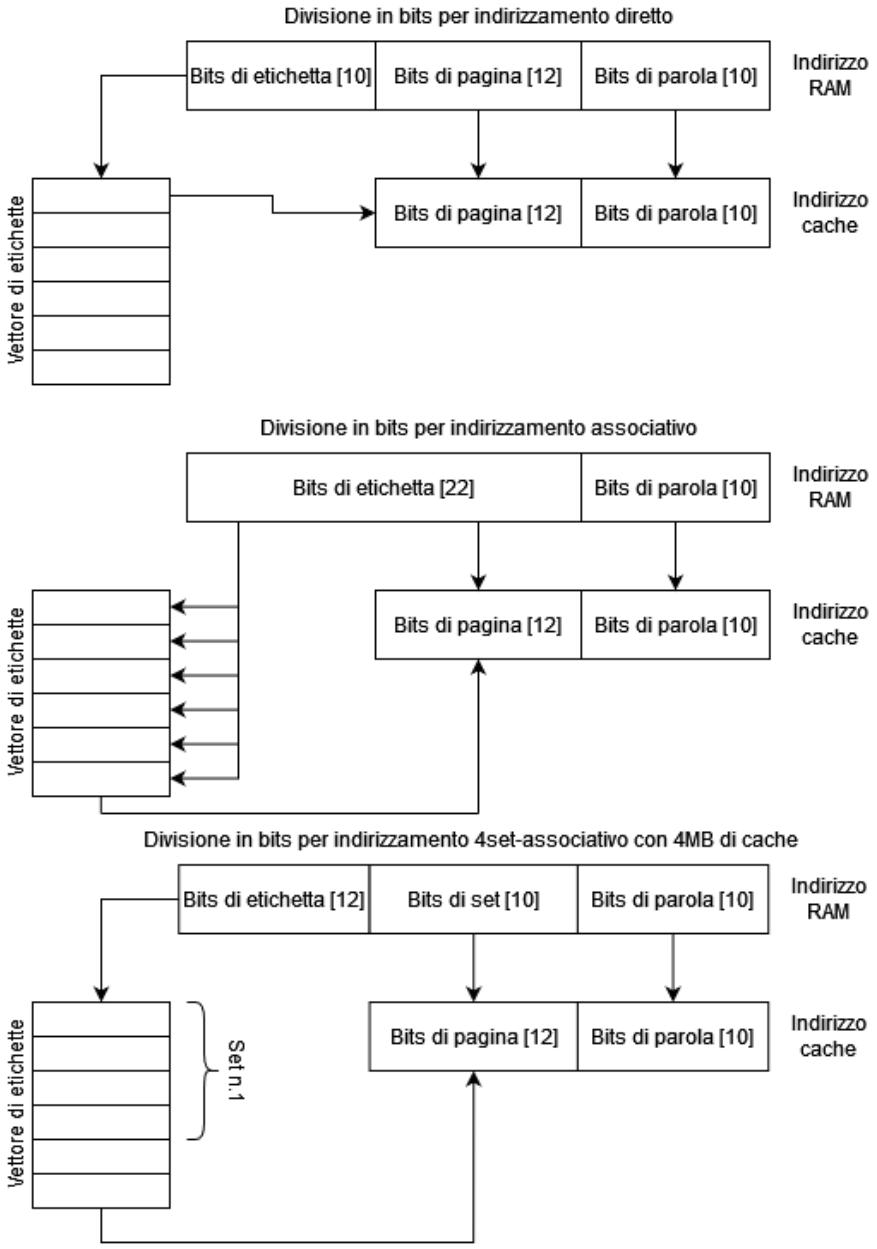


Figura 5.10: Divisione in bit per i metodi di indirizzamento

Diciamo che P_1 richiede una malloc, ma non è possibile donare altro spazio poiché è coperto da P_2 e non è possibile spostare i processi, oppure mettiamo caso che P_2 finisca il suo lavoro e devi inserire P_4 ma non ha abbastanza spazio.
Come puoi vedere, è un bordello. Ragion per cui hanno ideato la **Paginazione**; l'assegnazione ad ogni processo di un numero n di pagine indipendentemente dalla loro posizione fisica. Ciò risolve il problema alla radice perché potrei aggiungere pagine quando richieste.

Ma come funziona? Abbiamo la solita divisione dei 32bits nella memoria, dove 10 rappresentano la parola e i restanti 22 l'indirizzo logico che poi verrà calcolato nell'indirizzo fisico. Noi potremmo assegnare a questi bit di indirizzo una pagina logica e una fisica, mantenendo lo stesso procedimento visto nello scorso capitolo. Il twist è la presenza della **Tabella delle pagine**; una matrice tanto grande quanto le pagine del processo, dove sono scritte in ordine crescente tutte le pagine logiche insieme alla loro rispettiva pagina fisica.

Questa matrice è contenuta in un circuito chiamato **MMU**; Memory Management Unit, presente nella CPU e guidato dal sistema operativo. Riceve i dati dalla tabella con la possibilità di ampliarla quando richiesto. Può inoltre riconoscere se le informazioni ricevute sono utili o meno, ed eventualmente scartare le seconde.

Questo circuito è necessario per due motivi:

1. Se questo compito di ricerca fosse effettuato dal sistema operativo, si richiederebbe moltissimo tempo.

2. Sinergizza con il principio di località, rendendo il processo efficiente.

Una curiosa domanda ora: quanto è grande la tabella delle pagine? Comprenderà certamente l'insieme delle pagine necessarie per il funzionamento del programma. Chiamiamo tale famiglia il **Working Set**, basato sul principio di località.

Capirai che ogni processo avrà il proprio working set, ma una cosa meno ovvia è come si tratta del giusto equilibrio fra tempo di esecuzione ed il numero di pagine minimo utilizzabile, allo scopo di garantire le prestazioni migliori.

Rimane solo da vedere il gran sotterfugio: la **Memoria Virtuale**. Per capirla iniziamo a ragionare dal working set. Come precedentemente menzionato, ogni processo ha il suo working set che lavora indipendentemente dagli altri e noi dobbiamo convincere il sistema operativo che è in grado di usare tutta la memoria fisica.

Posso prendere un pezzo di SSD (o HD, che dir si voglia) detto **Swap** e donarlo al sistema operativo. Se una pagina del set viene usata poco, (controllo eseguito mediante contatori nella tabella delle pagine incrementati ad ogni ciclo di clock) verrà spostata in questo swap per liberare spazio in RAM. Chiariamo che questo processo non butta via le pagine perché sempre nella tabella è presente un'area apposita che dice se una pagina si trova o meno nello swap.

Nel caso in cui serva una pagina presente nello swap, la si scambierà sempre la pagina più vecchia in ram. Questo algoritmo non è tuttavia onnipotente, perché dando troppo spazio allo swap potresti inchiodare il sistema. Caution is advised.

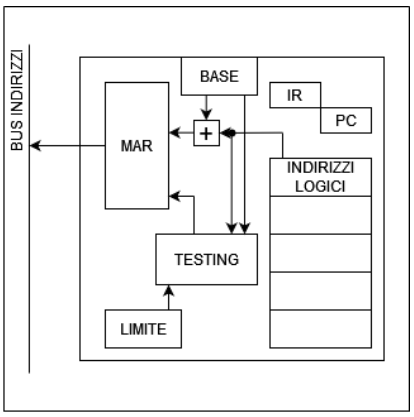


Figura 5.11: Circuito per la rilocazione

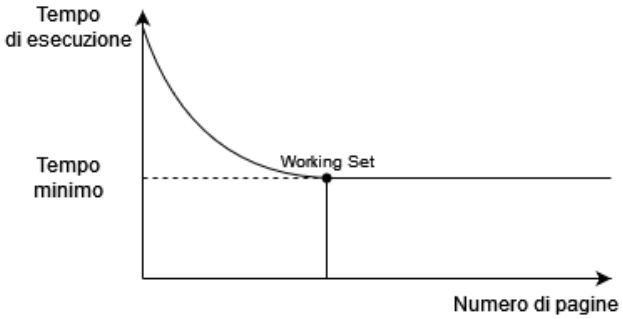


Figura 5.12: Come ottenere il working set

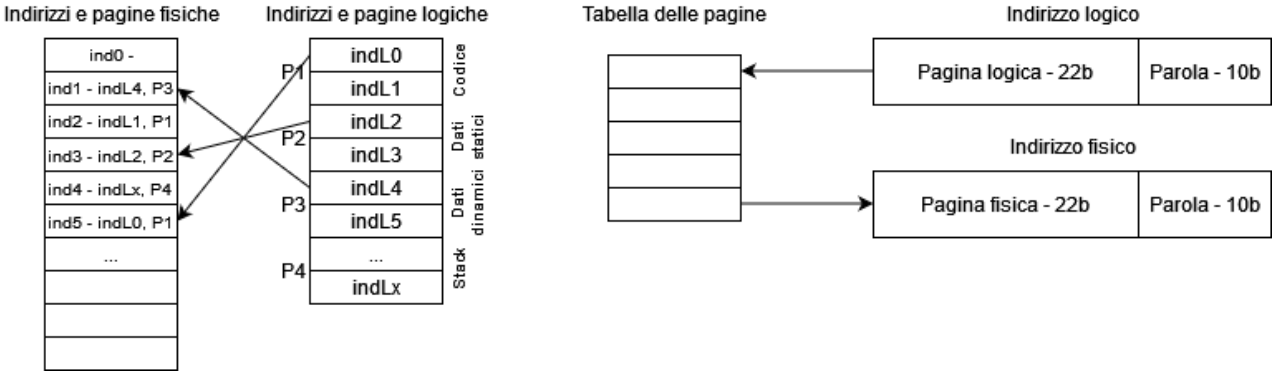


Figura 5.13: Percorso di paginazione

5.5 Pipelining

Finora abbiamo considerato la processione dei dati come una FSM a quattro stati, ma è possibile per i nostri scopi dividere l'ultimo stato di execution fra *esecuzione istruzione* e *salvataggio risultato* e chiameremo quest'ultimo stato **WriteBack**.

Immaginiamo di avere un circuito semplice, dal leggero costo in runtime: un sommatore. Ti guiderò in ogni istruzione così fai anche un ripasso.

Nello stato fetch riceviamo le istruzioni dai registri o dalla memoria del calcolatore. Abbiamo un'architettura ottimizzata con un circuito di somma esterno e registri flip-flop; di conseguenza avremo un accesso alla memoria in un ciclo di clock.

La fase di decode ha il compito di decodificare le istruzioni mediante le informazioni che riceve da registri o memoria. Queste possono richiedere un accesso a memoria o meno. Dipendentemente da quanti sono avremo il nostro ciclo di memoria in un ciclo di clock.

Siamo arrivati all'esecuzione. Non richiede accessi a memoria e nell'immaginario di un'architettura ottimizzata con ben tre bus per la ALU, si riesce ad effettuare il tutto in un ciclo di clock.

L'ultima fase di WriteBack richiede un accesso in memoria solamente quando deve salvare i dati, facendo un accesso a memoria. Se non accede, nulla è modificato. Serve in ogni caso un ciclo di clock.

	Ciclo di Memoria	Ciclo di clock
Fetch	1	1
Decode	da 0 a 1	1
Execution	0	1
WriteBack	da 0 a 1	1
Totale	da 1 a 2	4

Tabella 5.2: Costo in accessi a memoria/cicli di clock dei quattro stati

Puoi osservare nella tabella riassuntiva che il numero minimo di cicli ottenibile è $1 + 4 = 5$, 6 nel peggiore dei casi. Questo è ottenibile solamente con un ISA richiedente un solo accesso a memoria, ma ahimè, non è neanche lontanamente vicino al nostro amato 1CPI.

Possiamo ulteriormente ottimizzare minimizzare la memoria scegliendo quale operazione fra decode e writeBack avrà il diritto di accedervi.[14] Conviene inoltre dividere i bits della memoria cache in due parti: una per i dati e l'altra per le istruzioni, rendendo possibile farle lavorare in contemporanea a due mansioni diverse.

Non è ancora abbastanza; per questo introduciamo la strategia di **pipeline**; una strategia portentosa che vede i quattro stati messi in fila come una catena di montaggio. Questi sono separati da dei banchi di registri che salveranno quanto svolto per darlo allo stato seguente.

Questo sistema consente alle quattro istruzioni di lavorare indipendentemente l'una dall'altra[15], riducendo drasticamente i cicli di clock necessari per elaborare un programma. Osserviamo l'idea per il suo funzionamento:

1. L'istruzione di *fetch* prende dai registri il valore del PC e dalla cache istruzioni il compito che dovrà eseguire. Incrementerà di uno il PC e passerà nel banco il valore di quest'ultimo insieme a quello dell'IR.

2. L'istruzione di *decode* riceve le informazioni dal banco precedente e dipendentemente dalla modalità di indirizzamento, prenderà il codice dalla cache o dai registri. Metterà infine nel suo banco gli operandi ed il valore di PSW.

3. L'istruzione di *execution* riceverà gli operandi e ne calcolerà il risultato. Aggiornerà infine i registri e metterà il tutto nel suo banco insieme al PSW aggiornato.

4. L'istruzione *WriteBack* infine salverà questi risultati nella cache dati o nei registri, dipendentemente da dove è richiesto.

E questa era la teoria; passando alla pratica si presentano due problemi:

- Abbiamo una situazione dove ogni stato sarà molto più complesso, siccome le quattro operazioni agiscono indipendentemente e nello stesso ciclo di clock.

- Le *bolle*; delle istruzioni che non possono essere eseguite nello stesso ciclo di clock a causa di una dipendenza condizionale o ciclica.

Osserva bene la tabella considerando la singola funzione di ogni istruzione. Al quarto ciclo di clock si vuole sottrarre il valore 1 al contenuto del registro %EBX, ma questo non è stato ancora

[14] Non possono avere entrambe accesso alla memoria, devi per forza sceglierne una.
[15] Nella maggior parte dei casi è così. L'eccezione è quando si hanno cicli e condizioni.

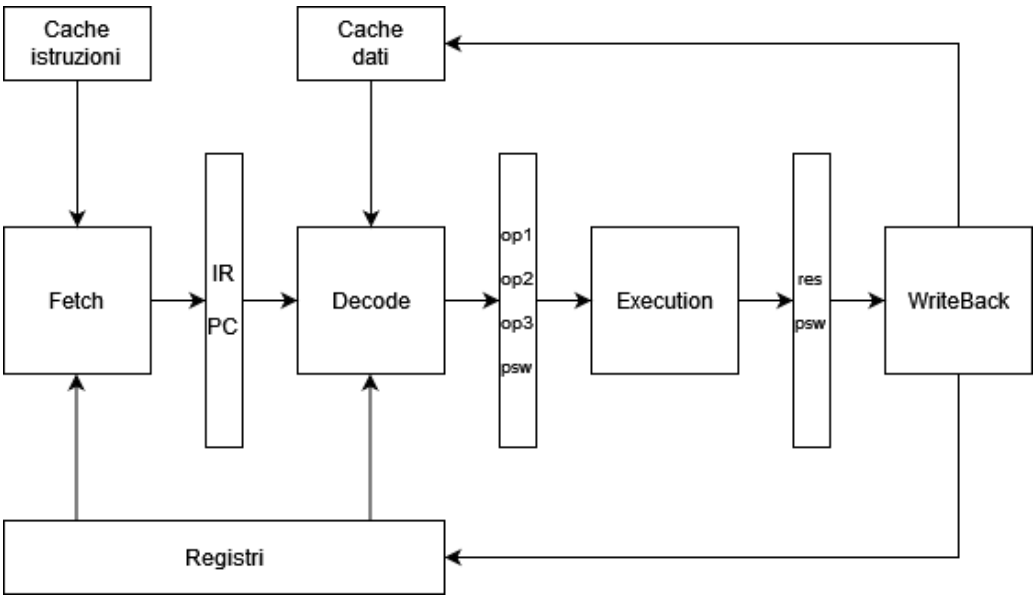


Figura 5.14: Circuito di pipeline

preparato da WriteBack, quindi sarà necessario aspettare un ciclo di clock per effettuare poi il decoding.

Da questo puoi notare come un registro non sia pronto quando viene effettuata la WriteBack, bensì a quello successivo.

Esistono inoltre delle strategie per la riduzione delle bolle; per le condizioni si allontanano le istruzioni che dipendono le une dalle altre, mentre per i cicli ci si affida ad un riconoscimento di pattern tramite le *jump predictions*[16].

In questo caso, se il risultato del salto è predetto correttamente, si potrà effettuare il fetch della prima istruzione al nono ciclo di clock, ottenendo un CPI medio di $\frac{12-4}{6} = 1,35$CPI, che si avvicina molto al nostro obiettivo.

Per chiarire, infine, la preparazione è del tutto innocua in quanto non va a modificare alcun registro.

INIT:	1	2	3	4	5	6	7	8	9	10	11	12	13	14
movl %eax, %ebx	F	D	E	WB								F	D	E
addl $4, %ecx		F	D	E	WB									
subl $1, %ebx			F	D	D	E	WB							
cmpl %eax, %ecx				NOP	F	D	E	WB						
JZ INIT						F	D	D	D	E	WB			
Altre istruzioni												F	D	E

Figura 5.15: Istruzioni pipeline

[16]La CPU osserva il primo risultato della condizione e implica che questo avvenga anche alla prossima richiesta. Si prepara di conseguenza per l'output che si aspetta.

Ulteriori Architetture

6.1 Architettura LC-3

Il microprocessore **Little Computer 3** è un'architettura CPU basata sul modello di Von Neumann della quale studieremo struttura, caratteristiche e ISA. **- Struttura - Divisione in**

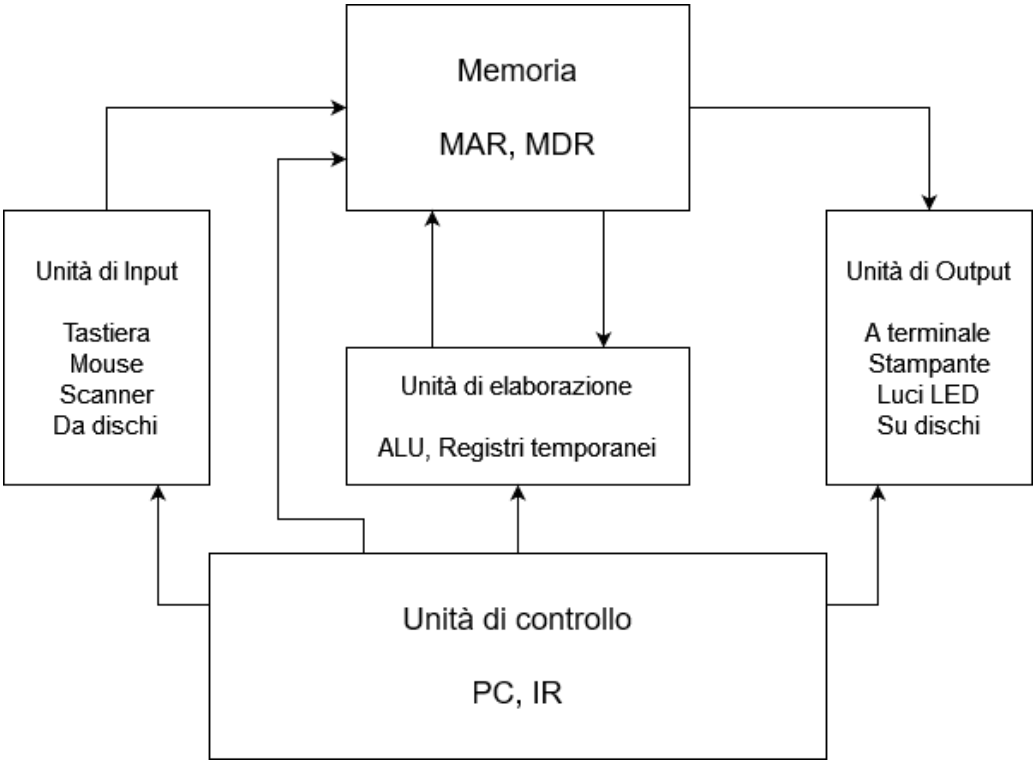


Figura 6.1: Modello di architettura LC-3

bit delle istruzioni e stati di processo ISA LC-3

6.2 Modello CISC e RISC

My dear Reader, che ne pensi di una sezione relax dove ti racconto una storia al posto delle solite spiegazioni? Davvero, zero immagini, zero presura e mettiamoci comodi.

Parliamo di com'è nato il modello di architettura sul quale stai con ogni probabilità leggendo questo PDF: il **RISC**[1]; figlio del connubio di tutto ciò che abbiamo studiato finora. Si tratta dell'architettura con l'attuale migliore ottimizzazione del *tempo impiegato dalla CPU*[2] per elaborare le informazioni.

[1]Reduced Instruction Set Computer - Elaboratore dall'insieme di istruzioni ridotto.

[2]Più precisamente, vale l'equazione: $\mathsf{T}_{\mathsf{CPU}} = \mathsf{Tot}_{\mathsf{Istruzioni}} \times \mathsf{CPI}_{\mathsf{Medio}} \times \frac{1}{\mathsf{Freq}_{\mathsf{Clock}}}$.

Ma partiamo dal principio: Giusto neanche cinquant'anni fa, il mondo ha visto una grandissima opportunità di mercato nei microprocessori, e di conseguenza molte aziende, come Intel, hanno provato a prendere parte alla corsa per fare i grossi soldi.

L'intenzione è creare il microprocessore più efficiente possibile per sbaragliare la concorrenza. Come fare? La prima idea diffusa su vasta scala fu quella di *aumentare la frequenza dei cicli di clock*. Essendo che la frequenza del clock è data dal Datapath, il quale ha un cammino critico, si pensò di utilizzare delle pipelines per ottimizzare le istruzioni dell'ISA.

Fatto sta che il funzionamento della pipeline dipende dalla quantità ed il tipo di istruzioni presenti nell'insieme e siamo inoltre limitati dalla potenza delle componenti fisiche. Agiremo, ordunque, sul valore totale delle istruzioni nell'ISA, modificandole e rendendole molto più complesse e compatte. Abbiamo ottenuto un valore molto minore di direttive, ma avremo come conseguenza un datapath che rispecchia la complessità di quanto rielaborato e quindi un cammino critico devastante, per non parlare della pipeline che deve costantemente aspettare il lavoro finito dello stato precedente. Doesn't sound that efficient now, does it?

Qui entrarono in gioco due professori dell'Università di Stanford; coloro che ebbero il coraggio e l'astuzia di pensare fuori dal pensiero comune, creando l'architettura sulla quale si basano gli elaboratori contemporanei: *John Hennessy* ed il suo collaboratore *David Patterson*.

"E se fossero le istruzioni a dover essere ridotte per ottenere prestazioni migliori?" Fu la fatidica domanda che smosse il tutto. Chiesero ad altri ricercatori di creare dei programmi in diversi linguaggi e notarono una particolarità fra tutti i codici: erano scritti in modo *semplice* indipendentemente dalla complessità del linguaggio. Come diretta conseguenza, anche le istruzioni in Assembly erano semplici.

Con questa filosofia i due riuscirono a creare il DLX, il primo vero dispositivo RISC e con i dati alla mano, nonostante i soliti dubbi dal mercato, riuscirono a dimostrare di aver creato un microprocessore nettamente migliore a quelli delle altre aziende, che decisero di chiamare **CISC**[3]. I dispositivi RISC, oltre ad avere poche istruzioni, hanno il vantaggio di avere pochi metodi di indirizzamento. Questa semplicità aiuta il funzionamento della pipeline, la quale sarà nettamente più veloce, ottenendo non solo un numero molto minore di CPI medio, ma anche una frequenza di clock maggiore rispetto alle altre architetture.
Annusata l'opportunità per fare soldi, le aziende iniziarono a cimentarsi nella filosofia RISC, ottenendo risultati sulla potenza in pochissimo tempo:

- **Intel** fu come al solito la prima e creò il *MIPS*, con una frequenza di 100MHz.
- **Motorola** seguì subito con la creazione di *PowerPC*, da 200MHz.
- **DEC** poi creò *Alpha* dai sorprendenti 600MHz.
- Il **Stanford University Network** infine creò la *SPARC* su FPGA, da 300MHz.

Nella foga per l'ottimizzazione, gli ingegneri a Intel crearono il microprocessore P6, che per quanto ottimale potesse risultare al tempo, aveva un ISA completamente diverso da quello utilizzato da loro finora (e tuttora); Intel 80x86.

Si dovette creare un chip più grande con lo scopo di, mediante una fase di *Pre-Fetch*, tradurre l'ISA diverso nell'80x86. Chiamarono questa architettura *Pentium*.

Attualmente nessuno di questi microprocessori è più in uso, ma immagino tu già lo sappia, dato che abbiamo ben superato il GigaHz di frequenza.

[3]Complex Instruction Set Computer - Elaboratore dall'insieme di istruzioni complesso.

Figura 6.2: John Hennessy e David Patterson

6.3 Architetture parallele

Allora, è nostro volere ottimizzare *even further beyond*, quindi bisogna capire cosa vuol dire veramente migliorare le prestazioni di un calcolatore.

Da un punto di vista dell'utente medio si può pensare a ridurre il tempo totale di esecuzione della CPU, che non è necessariamente sbagliato, ma è un ragionamento superficiale. Si ritiene invece più importante aumentare il numero di processi eseguibili in un determinato arco di tempo ed è questo ragionamento che ha portato allo sviluppo delle **Architetture Superscalari**, modelli con la capacità di elaborare istruzioni con un CPI medio minore di 1, ma come raggiungere questo risultato?

Il primo microprocessore vincente fu il *PowerPC* di Motorola, il quale utilizzava una componente di pre-fetch che inviava poi i dati da elaborare a *due pipelines* da quattro stati l'una. La prima avrebbe svolto operazioni solamente con interi, l'altra con numeri in virgola mobile. Queste due pipelines funzionano *in parallelo*, quindi dove normalmente avremmo un CPI = 2, data la "doppia velocità", abbiamo il CPI = 0,5. Inoltre abbiamo la possibilità di donare loro un clock proprio per gestire al meglio le operazioni ed aumentare conseguentemente ancora di più la frequenza di clock. Se far ciò migliora così tanto le prestazioni, perché non inserire sempre più pipelines che lavorano parallelamente? Ti invito a richiamare alla memoria la *dipendenza delle istruzioni*. Ulteriori pipelines equivalgono ad ulteriori condizioni e più ce ne sono, più (se necessario, ma di norma lo è) bisognerà attendere la fine della fase precedente, rallentando l'intero processo e peggiorando le prestazioni. Che fare, ordunque?

Un primo tentativo di risoluzione si ebbe con le architetture di microprocessori **VLIW**[4], il cui scopo è ottenere un livello di astrazione maggiore per le istruzioni, le quali saranno quindi composte di più operazioni, aumentando la dimensione dei banchi di memoria presenti fra la pipeline.

Di una diversa linea di pensiero furono i creatori dei **Calcolatori Vettoriali**; i quali pensarono di utilizzare, insieme alla componente di pre-Fetch e ad un banco di memoria, molte più CPU che lavorano in parallelo. Per esempio, se bisogna eseguire 1000 somme normalmente uti-

[4]Very Long Instruction Word - Parola d'istruzione molto lunga.

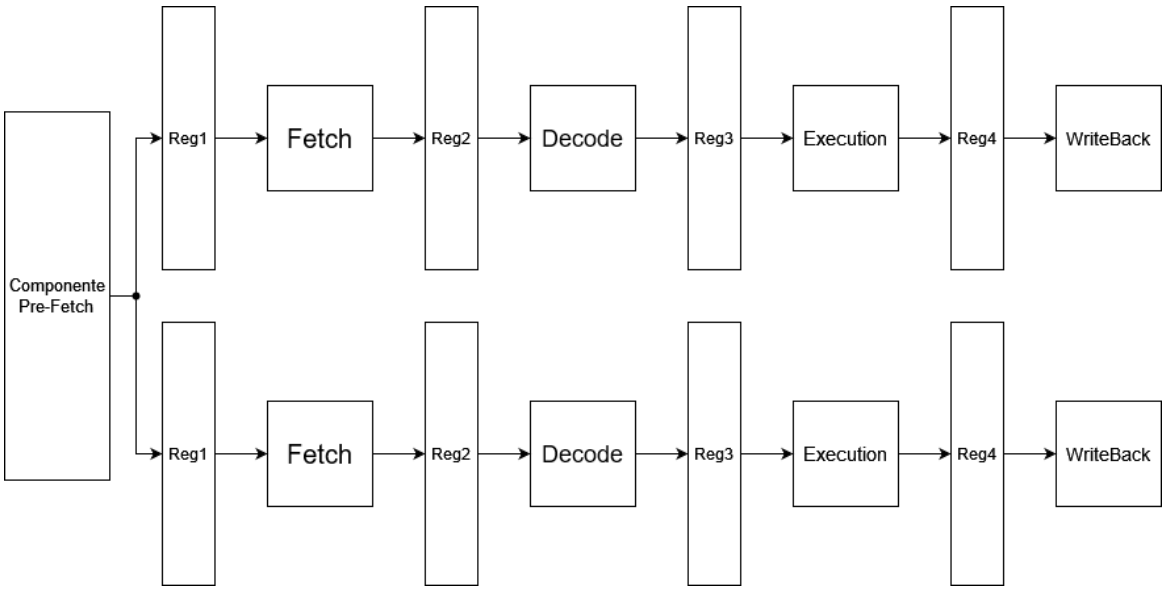


Figura 6.3: Funzionamento dual-pipeline di PowerPC

lizzeresti 1000 cicli di clock, ma se inseriamo 500 circuiti di somma, ne utilizzeremo solamente 2; un miglioramento spaventoso. Tuttavia, per collegare le mini-CPU era necessario utilizzare una *matrice di interconnessione*, che causava una sovrabbondanza di collegamenti.

Un altro problema è presentato dalla **Legge di Amdahl**, la quale afferma: "*Il miglioramento delle prestazioni di un sistema che si può ottenere ottimizzando una certa parte del sistema è limitato dalla frazione di tempo in cui tale parte è effettivamente utilizzata*".
In merito introduco il concetto di **Speed-up**, che ha origine proprio grazie a questa legge. Si tratta in genere di un valore che misura le prestazioni di un elaboratore in funzione di un determinato programma; in particolare ci interessa la *Latenza*[5] per poter eventualmente paragonare i tempi di due architetture effettuando un rapporto fra di loro.
Visto questo, immagina un calcolatore vettoriale che cerca di svolgere qualsiasi compito diverso da operazioni aritmetiche. Non sembra molto efficiente.

Ipotizziamo ora di non volere un parallelismo per ogni singolo programma, bensì di lavorare su un parallelismo fra i processi e quindi avere più CPU in una singola architettura. Il problema fondamentale di questa idea è che ognuna di loro necessita la propria cache e nell'elaborazione dei dati avremo delle copie esatte di quanto elaborato in ogni CPU, quindi un problema di *Coerenza di cache*.

Per la gestione di questo problema vennero ideati gli **Algoritmi di Snooping**, che vedono la presenza in ogni microprocessore di una tabella contenente le posizioni di $\mathfrak{n}$ dati, dopo che questi sono passati per il BUS. Non sono tuttavia soluzioni perfette, poiché il troppo snooping peggiora le prestazioni, senza contare che abbiamo un singolo BUS dove passano le informazioni[6].

Poi venne il nuovo millennio e cambiò tutto con la tecnologia **Network On Chip**; la realizzazione di una matrice di interconnessione sul silicio stesso, che consentì di inserire più CPU in una singola architettura. Questi processori vengono chiamati *Cores* e questo miracolo diede vita

[5] $L = \frac{T}{W}$, dove T è il tempo e W il totale del lavoro eseguito per la task.
[6] Se non ti sembra aver senso invito la revisione della sezione dedicata all'arbitraggio del BUS.

ai processori **Multicore**, inseriti nei dispositivi come quello che stai utilizzando ora. Il grande vantaggio è avere una cache consistente poiché le connessioni della matrice annichiliscono le attese per accedervi.

Ma non è finita qui; venne data nuova vita ai calcolatori vettoriali per la necessità della rapida elaborazione delle grafiche a video. Si tratta di un compito particolarmente complesso basato su calcoli computazionali; il punto di forza di tali architetture. Oggi sono conosciuti sotto il nome di GPU[7] ed il capo supremo del loro mercato è NVidia.

Col passare del tempo questi modelli si svilupparono e videro la luce le GPGPU[8], utilizzate per richieste grafiche particolarmente esigenti, come un qualunque gioco PS5 realistico. Si presta inoltre molto bene per la creazione di reti neurali, quindi per la creazione e lo sviluppo di intelligenze artificiali.

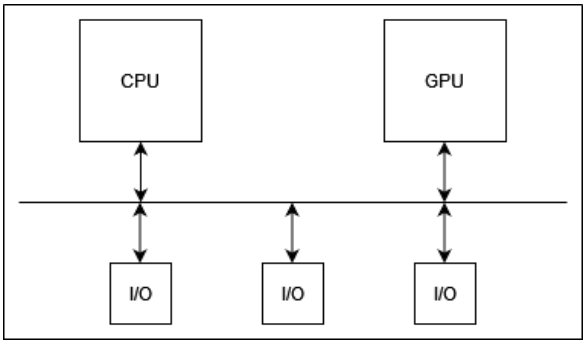


Figura 6.4: Architettura di Von Neumann contemporanea

[7]Graphic Processing Unit - Unità di elaborazione grafica.
[8]General Purpose GPU

SIS - Sequential Interactive Synthesis

7.1 Introduzione a SIS

7.2 Sintesi combinatoria esatta

7.3 Sintesi combinatoria approssimata multilivello

7.4 Modellazione di FSM

7.5 Modellazione di FSMD

Linguaggio HDL - Verilog

8.1 Introduzione a Verilog

8.2 Modellazione in Verilog

8.3 Modellazione di FSM

8.4 Modellazione di FSMD

Linguaggio Assembly

9.1 Introduzione al linguaggio Assembly

Quando collocate nella memoria, le istruzioni macchina hanno una forma binaria, illeggibile dalla persona; per questo ci si affida alla forma simbolica dell'istruzione, chiamata **Assembly Language**. Si tratta di un linguaggio comune a tutti gli elaboratori, usato come base per creare gli ISA propri della macchina.

Consente di accedere ai registri della CPU, Scrivere un codice ottimizzato per una specifica architettura di microprocessore e di ottimizzare sezioni dei programmi a livello hardware. Esistono due tipi di convenzioni o regole:

- Quelle che definiscono la forma simbolica del linguaggio che formerà il **Programma Sorgente**.
- Quelle che definiscono la forma numerica che formerà il **Programma Oggetto** e il passaggio all'altro tipo.

Il compito di controllare la correttezza sintattica del codice è affidato all'**Assembler**, il quale eventualmente genererà la forma numerica corrispondente a quanto scritto sotto forma di file con *estensione ".o"*. L'ISA utilizzato per gli esercizi è **AT&T**, la cui unica differenza dal più comune **Intel 80x86** è l'utilizzo di un % prima del nome dei registri.

- ISA AT&T:

  ADDL %EAX, %EBX

- ISA INTEL 80X86:

  ADD EAX, EBX

Utilizzeremo registri dalla dimensione base di 16b. Scrivendo "E" all'inizio del loro nome diventeranno *extended registers*, dalla dimensione di 32b. Si dividono in:

- Generici:

- *AX*, Accumulation Register; Accumulatore di operazioni aritmetiche e contenitore del risultato.
- *BX*, Base Register; Per le operazioni di indirizzamento alla memoria.
- *CX*, Counter Register; Usato per contare, per esempio, l'indice nei cicli.
- *DX*, Data Register; Usato nelle operazioni di I/O, divisioni e moltiplicazioni.

- Di segmento:

- *CS*, Code Segment; Punta alla zona di memoria che contiene il codice e fa accedere all'istruzione successiva. Non può essere modificato.

- *DS*, Data Segment; Punta alla zona di memoria che contiene i dati.
- *ES*, Extra Segment: Spesso usato come registro ausiliario.
- *SS*, Stack Segment; Punta alla zona di memoria dove risiede la stack.

- Puntatore:

- *SP*, Stack Pointer; Punta alla cima della stack. Viene modificato dalle istruzioni PUSH e POP[1].
- *BP*, Base Pointer; Punta alla base della porzione di stack gestita in quel punto dal codice.
- *IP*, Instruction Pointer; Punta alla prossima istruzione da eseguire.

- Indice:

- *SI*, Source Index; Punta alla stringa/Vettore sorgente.
- *DI*, Destination Index; Punta alla stringa/vettore destinazione.
- *FLAGS*; Memorizza lo stato corrente del processore. Ogni bit fornisce un'informazione diversa.

Infine vediamo le modalità di indirizzamento:

- *A registro*, [%EAX, %EBX]

  L'operando è contenuto in un registro il cui nome è specificato nell'istruzione.

- *Diretto o Assoluto*, [(%EAX), %EBX]

  L'operando è contenuto in una locazione di memoria e l'indirizzo di quest'ultimo è specificato nell'istruzione.

9.2 Istruzioni e sintassi

9.2.1 Stringhe e numeri

9.3 Debugging e Makefile

Il debugging è una parte fondamentale per il controllo di eventuali problemi nel codice che è stato scritto. Quello che sarà utilizzato è **GDB**, il debugger di GNU.
È necessario assemblare il programma con le seguenti linee di comando da terminale:

```
# Per creare il codice oggetto coi flag di debug
> as -gstabs -o file.o file.s

# Linking del file oggetto ad un nuovo file eseuibile
> ld -o file file.o
```

[1] Rispettivamente inserimento ed estrazione di un dato dalla stack

Seguono comandi utili generali per GDB. È possibile utilizzarli anche per i programmi C se è necessario:

```
# Run del debugger
> gdb nomeEseguibile

# Cambio visualizzazione. Utile per osservare il workflow e lo stato dei
  registri. Dopo aver dato il comando, premere invio per cambiare visuale.
> lay next

# Setta un breakpoint al punto indicato
> break nomeFunzione / numeroRiga

# Passa alla prossima istruzione
> next / nexti

# Va avanti fino al prossimo breakpoint
> continue

# Refresha la visuale. Si bugga spesso.
> ref
```

Adesso andiamo invece a vedere ulteriori comandi specifici per i programmi Assembly:

```
# Stampa a video i valori contenuti nei vari registri.
> info registers

# Stampa il valore nel registro in binario, decimale, hex.
> p/t $eax
> p/d $eax
> p/x $eax

# Trova l'indirizzo della variabile e stamoa 4B
> x/4b &variabile

# Stampa il valore di una variabile.
> print nomeVariabile

# Stampa il valore di un registro come stringa.
> x/s $eax
```

Tuttavia, attenzione. Assemblare un progamma con i flag di debug peggiora le prestazioni totali; inoltre comporta problemi di sicurezza perché consentirebbe di leggere il codice sorgente tramite debugger.

9.4 Relazione ISA-FSMD su LC3

9.5 Funzioni e passaggio di parametri

Prima di parlare di funzioni è necessario chiarire il funzionamento dei due tipi di strutture di dato:

- *LIFO*; Last In, First Out, si tratta delle dinamiche governanti la stack. Immagina un secchio nel quale stai ponendo dei libri, noterai sicuramente che ti è possibile rimuovere solamente l'ultimo inserito.
- *FIFO*; First In, First Out, facilmente comprensibile immaginando un tubo con una bocca che si chiude quando inserisci qualcosa ed un culo sempre aperto. Sarà possibile prelevare solo gli elementi in fondo alla pila.

Nei programmi Assembly è possibile utilizzare la stack mediante due comandi:

```
# Mette il valore contenuto in eax nella stack. [32b]
pushl %eax

# Preleva l'ultimo valore inserito in stack e lo inserisce in eax. [32b]
popl %eax
```

Per il funzionamento della stack, ogni volta in cui si pusha qualche valore su di essa, il registro %esp si decrementa, perché il valore più alto è alla base, mentre in cima sta 0.

Conoscere questa dinamica è importante, perché consente un maggiore spazio di manovra nel salvataggio dei dati. Inoltre, possiamo capire come dare parametri da terminale quando si attiva un programma. La linea di comando conta come una stringa ed è quella il path dell'eseguibile. Tutti gli indirizzi dei parametri (stringhe) dati vengono impilati sulla stack, insieme al loro numero totale, che starà sempre in cima ad essi.

Per poterli utilizzare bisogna usare il comando popl due volte per rimuovere prima il totale dei parametri, e poi prelevare il primo valore e salvarlo su qualche registro. Il comando dal terminale è infatti letto dall'assembler da destra a sinistra.

Una cosa utile è inoltre la possibilità di ottenere un valore dalla stack in qualunque posizione senza che esso venga prelevato, utilizzando la modalità di *Indirizzamento più spiazzamento*.

```
# Salva in ecx il valore in stack alla posizione +8 rispetto ad esp
  (vista come posizione 0).
movl 8(%esp), %ecx
```

Passiamo alle **funzioni**; è realisticamente impensabile voler scrivere tutto un programma in un singolo file, sia per motivi di lettura, che di debugging e controllo. Divideremo quindi il progetto in vari sottoprogrammi. Ciò è possibile tramite la seguente scrittura:

```
# Nel file main, si invoca la funzione.
call nomeFile

    # Nel file nomeFile
        # Dichiarazione della funzione
        .type nomeFunzione, @function
```

```
    .
    # Blocco di istruzioni
    .
# Ritorno al file chiamante la funzione
ret
```

Con queste conoscenze sei ora pronto a lavorare al progetto richiesto. Se non lo consegnerai entro i termini, perderai ogni cosa e avrai sprecato tempo.

9.6 Confronto con il linguaggio C