

UNIVERSITÀ DEGLI STUDI DI VERONA

CORSO DI LAUREA IN INFORMATICA

Architettura degli Elaboratori

Federico Brutti
federico.brutti@studenti.univr.it

*"Bits are bits, oramai lo sapete... o forse dovrei ricordarvi il discorso sui
CFU?" - Franco F.*

Contents

1	Codifica dell'Informazione	4
1.1	Sistemi ed elaborazione dati	4
1.2	Codice Binario e operazioni in base 2	6
1.3	Codifica dei numeri	7
1.3.1	Codifica in modulo e segno	8
1.3.2	Codifica in virgola fissa	8
1.3.3	Codifica in virgola mobile	9
1.3.4	Codifica in complemento a 1 e a 2	11
1.3.5	Codifica in esadecimale	12
2	Circuiti e Ottimizzazione	13
2.1	Realizzazione di porte logiche	14
2.2	Minimizzazione a due livelli	15
2.2.1	Mappa di Karnaugh	17
2.2.2	Algoritmo di Quine Mc-Cluskey	17
2.2.3	Funzioni parzialmente specificate	18
2.3	Dispositivi programmabili	18
2.4	Sintesi combinatoria multilivello	19
2.5	Mapping tecnologico	19
3	Progettazione Digitale	21
3.1	Circuiti sequenziali	21
3.1.1	FSM - Finite State Machines	21
3.2	Sintesi delle funzioni λ e δ , codifica degli stati	22
3.3	Minimizzazione degli stati	23
3.4	Datapath e componenti	24
3.5	Modello FSMD - Finite State Machine with Datapath	25
3.6	Derivazione FSMD da algoritmo	25
3.7	Modello dispositivo programmabile	26

4	Architetture dei Calcolatori	27
4.1	Modello di Von Neumann e Unità funzionali del calcolatore	27
4.2	Architettura CPU RISC-V	30
4.3	Metodi di I/O, Segnale Interrupt	35
4.4	Direct Memory Access, BUS e Arbitraggio	36
4.5	Stati di un processo	40
4.6	Pila e gestione Interrupt	42
4.7	Device driver	43
4.8	Tipi di Memoria RAM	45
4.9	Caratteristiche e gerarchia delle memorie	48
4.10	Memoria Cache, Paginata e Virtuale	48
4.11	Pipelining	53
4.12	Modello CISC e RISC	55
4.13	Architetture parallele	57
5	SIS e Verilog	61
5.1	Introduzione a SIS	61
5.2	Sintesi combinatoria esatta	61
5.3	Sintesi combinatoria approssimata multilivello	61
5.4	Modellazione di FSM	61
5.5	Modellazione di FSMD	61
5.6	Introduzione a Verilog	61
5.7	Modellazione in Verilog	61
5.8	Modellazione di FSM	61
5.9	Modellazione di FSMD	61
6	Il linguaggio Assembly	62
6.1	Introduzione ad Assembly	62
6.2	Istruzioni e Sintassi	64
6.3	Debugging e Makefile	65
6.4	Funzioni e passaggio di parametri	67
6.5	Assembly e C	68

Chapter 1

Codifica dell'Informazione

1.1 Sistemi ed elaborazione dati

Cominciamo col dire che lo scopo dell'informatica è la risoluzione dei problemi attraverso insiemi di istruzioni non ambigue, ovvero gli **algoritmi**. In questa materia ci occuperemo di studiare la progettazione ed ottimizzazione di sistemi digitali tramite programmi di **sintesi logica**¹ e rivolgeremo in seguito l'attenzione al linguaggio Assembly, per una corretta comprensione delle funzionalità di un'architettura. Ad oggi esistono due macro-categorie di architetture:

- **Sistemi embedded:** Macchine composte puramente da hardware, capaci di eseguire un solo algoritmo.
- **Sistemi general purpose:** Macchine composte dal connubio hardware-software, capaci di eseguire diversi algoritmi.

Generalmente, ogni sistema operativo lavora con il **linguaggio macchina** o codice oggetto; si tratta di una sequenza di "0" ed "1" con un significato specifico per la macchina e per essere leggibile dalle persone è necessaria una traduzione. Poniamoci quindi la domanda: "In che modo è possibile passare informazioni dal mondo reale ad un computer?"; stiamo parlando di un processo di due passi:

- **Input:** Le informazioni vengono prima recepite dalla macchina per la loro codifica e poi inviate al sistema operativo per l'elaborazione.
- **Output:** L'elaborato viene decodificato per risultare leggibile alle persone e poi mostrato all'utente.

¹Comunicazione da algoritmo a hardware.

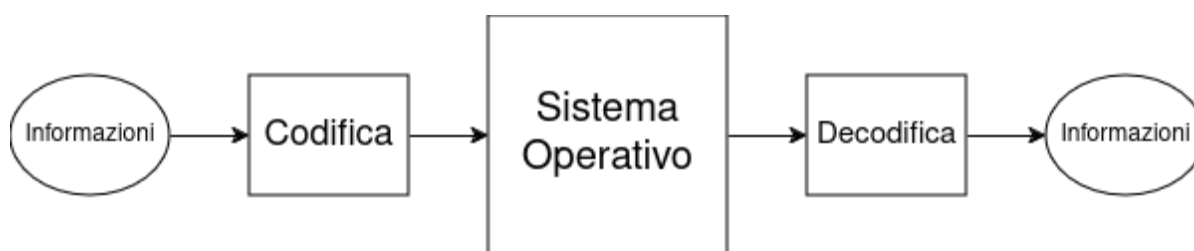


Figure 1.1: Ciclo di elaborazione informazioni

Questo vale come discorso generale; nello specifico è giusto chiarire che, avendo risorse limitate, non è possibile dare in input infinite informazioni. La soluzione a questo problema si ottiene con il seguente algoritmo:

1. **Campionamento:** Divisione in intervalli dell'informazione registrata.
2. **Discretizzazione:** Approssimazione degli stessi quanto possibile ad un numero leggibile dalla macchina.

Le informazioni sono recepite in un determinato arco di tempo, il quale viene misurato in **Hertz** ($1Hz = 1ms$). Per ogni elaborazione vale inoltre il seguente teorema:

Teorema 1. Teorema di Shannon

Data una funzione in un intervallo di campionamento e discretizzazione, è garantita la presenza di un errore.

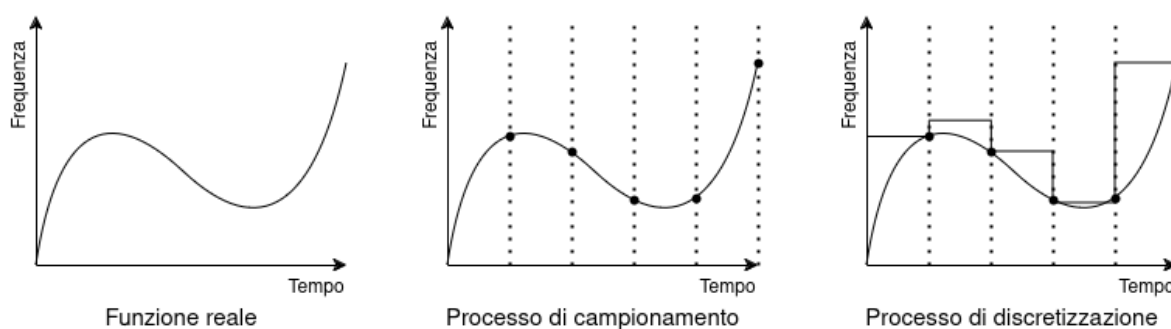


Figure 1.2: Processo di ricezione delle informazioni

Ma in che modo vengono codificate le informazioni? Si tratta di un processo che vede l'assegnazione di un codice binario ad ogni frammento di informazione con un certo numero di *bits*. Con "bit" intendiamo il numero totale di cifre binarie usate per un dato.

Esempio 1. Calcolo del numero di bits necessari per salvare informazioni

Supponiamo di avere $12M$ unità di dati da registrare; dobbiamo ragionare attraverso le potenze di 2 ed ottenere il valore più piccolo che sia maggiore o uguale al numero di dati da registrare.

In questo caso sarà $2^4 = 16$ e l'esponente sarà il numero di bits necessari. Per fare ordine:

- Da registrare: $12M = 12 \times 1000000 = 12000000$
- Potenza corretta: $2^4 = 16$
- Numero di bits necessari: $2^4 \implies 4b$

La codifica non è un procedimento sempre uguale; talvolta risulta necessario modificarlo in base alle specifiche del calcolatore, tra le quali notiamo in particolare la **facilità di calcolo**, che riguarda la semplicità delle istruzioni, e la **velocità di elaborazione**, la quale influenza la codifica. Se quest'ultima richiedesse più tempo rispetto alla frequenza di campionamento si perderebbero dei dati e per questo deve essere sempre maggiore o uguale al valore del campionamento.

Per il momento prenderemo in considerazione i **circuiti combinatori**, dove ad ogni codifica binaria è associata un'informazione e sequenze identiche verranno elaborate come la stessa informazione. In questo caso: **bits are bits**.

1.2 Codice Binario e operazioni in base 2

Il codice binario è fondamentalmente la "lingua" in cui è scritto il linguaggio macchina; come detto prima è una sequenza di 0 e 1 che rappresenta un'informazione. In particolare, il bit all'estremità sinistra è detto **più significativo**, mentre quello al lato opposto è il **meno significativo**.

È grazie alla codifica binaria che è possibile elaborare informazioni come immagini, musica e video, ma soprattutto numeri e caratteri, i quali hanno il codice **ASCII** con uno standard che vede i primi $127b$ comuni a tutte le lingue.

Ma per quale motivo stiamo considerando solo le potenze del 2? Procediamo a fare un collegamento: il codice binario ha solo *due* cifre utilizzabili, quindi bisognerà ragionare in loro funzione. Noterai, per esempio, che per $2b \implies 2^2 = 4$ puoi esprimere quattro combinazioni di numeri diverse senza ripetizione; estendendo il ragionamento a valori più alti avrai capito il funzionamento.

Lavorare in una base diversa dal 10 non comporta modifiche nella logica; infatti sono presenti tutte le operazioni elementari, come segue:

- 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 può risultare contro-intuitiva a causa dell'utilizzo della sottrazione. Vediamo con un esempio: $\frac{11001}{101} = \frac{25}{5}$.

1 1 0 0 1	1 0 1
- 1 0 1	
0 0 1 0	1 0 1
- 0 0 0	
0 1 0 1	
- 1 0 1	
0 0 0	

Bisogna poter sottrarre il divisore al dividendo quando questo "sta dentro" al primo.

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.

Esistono anche altre 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). $1101 \times SL = 11010$.
- **Shift Right**; Sposta le cifre a destra ed aggiunge uno 0 a sinistra. $1101 \times SR = 0110$.

1.3 Codifica dei numeri

In questa sezione vedremo come codificare i numeri e le particolarità di ogni algoritmo che svolge tale funzione. Si lavora con cifre binarie con la regola della *Notazione posizionale*, dove il valore di un numero è dato dalla posizione delle sue cifre. Iniziamo con la **codifica standard**.

Questo è un algoritmo utile per lavorare con numeri interi positivi. Bisogna 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".

Esempio 2. Codifica del numero 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.

1.3.1 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 ulteriore bit nella parte più significativa per marcare il segno positivo "0" o negativo "1".

Esempio 3. Codifica del numero -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:
 $227 = 011100011 \implies -227 = 111100011.$

1.3.2 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 virgola rimane nella stessa posizione della base 10.

in primo luogo bisogna codificare la parte intera come una normale codifica in modulo e 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.

Esempio 4. Codifica del numero 56,83 in 3b 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

Risultato: 111000,110

1.3.3 Codifica in virgola mobile

La codifica in virgola mobile o *Floating Point* consente di ottenere numeri particolarmente grandi e piccoli. Risulta utile per avvicinarsi al concetto di numero reale. Un tale valore si esprime nella formula di **Notazione scientifica**:

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

Si divide quindi in tre parti a cui è associato un numero specifico di bits da un totale di 32b (float) oppure 64b (double); le quali sono:

- *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**, ovvero portarlo in una forma dove rimane una singola cifra intera attraverso le operazioni di shifting a destra o sinistra.

Dipendentemente da quante posizioni sono modificate, sarà necessario sommare, se *SR* o sottrarre, se *SL*, tal numero all'esponente. Una volta ottenuto, bisogna sommargli $+127^2$ e hai fatto.

Notare che nella codifica della mantissa la cifra intera non è mai scritta perché è sempre la stessa e si può omettere.

Esempio 5. Codifica del numero -30,375 in virgola mobile

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	

²Questa operazione si chiama **Eccesso 127** ed è necessaria per codificare l'esponente nello standard IEEE754.

$$2 - 2 = 0$$

Codifica in virgola fissa: 11110,011

2. Procediamo con la normalizzazione:

$$11110,011 / 1000 = 1,1110011 \times 2^4 \quad \text{Sommeremo 4 all'esponente.}$$

La mantissa sarà: 1110011...0.

3. Troviamo l'esponente:

Non c'è un esponente nel numero richiesto, quindi: $1 \times 4 + 127 = 131$.

$$131 - 128 = 3 \quad 1 \text{ in bit } 7$$

$$3 - 2 = 1 \quad 1 \text{ in bit } 1$$

$$1 - 1 = 0 \quad 1 \text{ in bit } 0$$

$$131 = 10000011 - \text{Esponente trovato!}$$

4. Ricomponiamo il tutto

- Segno: 1

- Esponente: 10000011

- Mantissa: 1110011...0

La codifica in virgola mobile di $-30,375$ è: 1100000111110011...0

Esempio 6. Trasformazione in decimale di 0100011001000110...0

1. Dividiamo nelle varie parti i bits

- Segno: 0

- Esponente: 10001100

- Mantissa: 1000110...0

2. Otteniamo l'esponente decimale

$$128 + 8 + 4 = 140$$

$$140 - 127 = 13 - \text{Esponente trovato!}$$

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:

$$2^{-1} + 2^{-5} + 2^{-6} = 0,547 \text{ - Valore della mantissa trovato!}$$

4. Ricostruiamo il decimale

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

Risultato: $1 \times 1,547 \times 2^{13} = 1,547 \times 2^{13}$

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

- +0; Tutte le cifre sono 0.
- -0; Tutte le cifre sono 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 sono 0, tranne il bit più significativo dell'esponente.
- 2^{-145} ; Tutte le cifre sono 0 tranne il bit meno significativo. Si tratta del numero più piccolo ottenibile.

1.3.4 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 all'altra.

Il suo scopo principale è dividere a metà il totale delle codifiche ottenibili da 2^nb , rendendo più semplice ottenere numeri lunghi. Supponiamo 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.

Esempio 7. Codifica di -3 in complemento a 2 con 4b di precisione

$$\begin{aligned} 3 &= 0011 \rightarrow 1100 \text{ in compl. ad 1} \\ 1100 + 1 &= 1101 \text{ in compl. a 2} \end{aligned}$$

Segue una lista indicativa di tutte le codifiche in precisione 4b per il complemento ad 1 e 2:

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

1.3.5 Codifica in esadecimale

Soon!

Chapter 2

Circuiti e Ottimizzazione

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 come in base binaria: 0 e 1.

Secondo Boole, se si definisce una funzione che genera valori in un altro spazio, generalmente scritta $f(B^n) \rightarrow B^m$, 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.

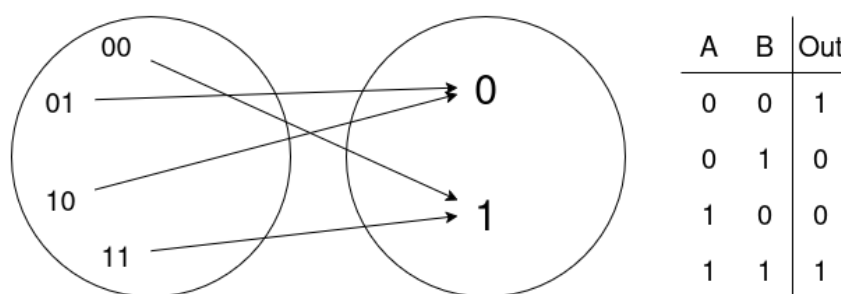


Figure 2.1: Funzione booleana XNOR

Nella tabella di verità vengono definiti:

- **Onset**: L'insieme dei punti dello spazio di ingresso dove la funzione vale 1. Gli elementi si dicono **mintermini**.

- **Offset:** L'insieme dei punti dello spazio di ingresso dove la funzione vale 0. Gli elementi si dicono **maxtermini**.

L'unione di questi due insiemi è complementare, poiché rappresentano tutto lo spazio usato dalla funzione. Inoltre, per mettere in relazione i bit con gli operatori si utilizzano le seguenti espressioni:

$$m_3 = a \times b, m_0 = !a \times !b$$

In tal 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.

2.1 Realizzazione di porte logiche

Le porte logiche e di conseguenza qualunque circuito elettronico sono governati dal flusso di elettricità gestito dai **transistors**; interruttori comandati. Nella tecnologia **CMOS** (Complementary Metal Oxide Semiconductor) sono implementati solo due tipi:

- **Interruttore N:** Se riceve corrente, conduce l'elettricità.
- **Interruttore P:** Se riceve corrente, ferma il flusso.

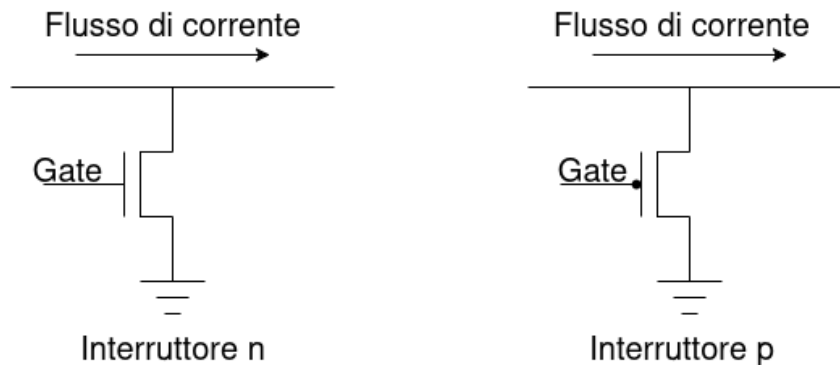


Figure 2.2: Tipi di transistors

Per usare termini corretti, quando un transistor è chiuso e scorre la corrente si dice che è in **conduzione**, mentre se è aperto e la corrente è bloccata diciamo che c'è un segnale di **interdizione**.

Di base i circuiti vanno letti da sinistra a destra, e grazie ai due interruttori appena visti è possibile creare le porte logiche elementari **AND**, **OR**, **NOT**, **NAND**, **NOR**, **XOR**, **XNOR**. Adesso abbiamo tutti gli strumenti per la creazione di un circuito elettronico. Ciò non significa tuttavia che possiamo buttarci senza cognizione di causa; ragion per cui bisognerà ragionare sui costi e le parti effettivamente necessarie al processo di realizzazione. Il nostro scopo da adesso diventa l'**ottimizzazione** dei nostri progetti.

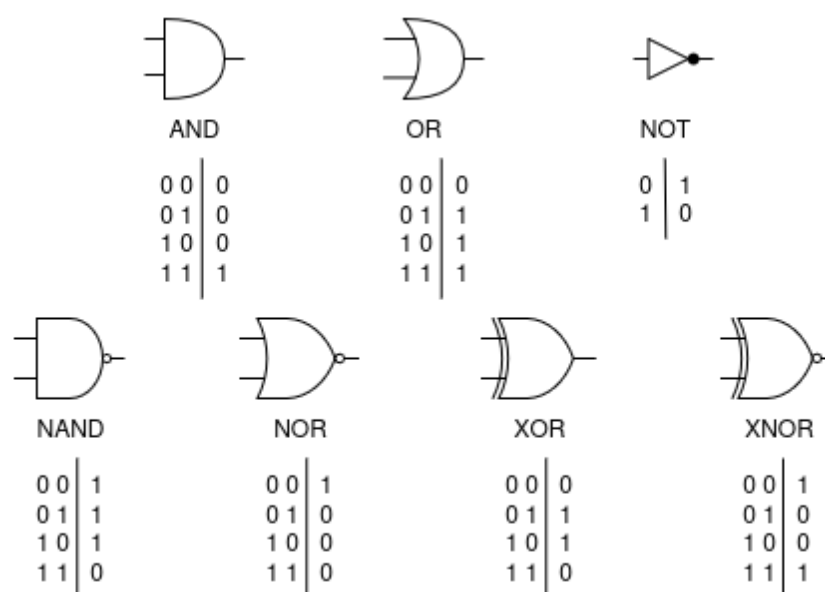


Figure 2.3: Porte logiche elementari

2.2 Minimizzazione a due livelli

Non esiste un solo modo per ottimizzare un circuito, bensì più tecniche, tutte con la loro ragion d'essere in base ai nostri scopi. Una prima semplice regola, per esempio è l'**assorbimento**, da utilizzare assieme all'algebra di Boole. Bisogna innanzitutto tenere a mente queste proprietà:

- Identità: $1 \times x = x$, $0 + x = x$
- Elemento nullo: $0 \times x = 0$, $1 + x = 1$
- Idempotenza: $x \times x = x$, $x + x = x$

- Inverso: $x \times \bar{x} = 0$, $x + \bar{x} = 1$

Valgono anche le proprietà associative, commutative e distributive. Procediamo col dare la definizione della regola:

Definizione 1. Regola dell'assorbimento

Sia una funzione booleana scritta in somma di prodotti; se due di questi sono a distanza di Hamming 1, è possibile sommare le parti inverse ed ottenere il valore 1, riducendoli letterali e quindi la complessità della funzione.

$$x(x + y) = x, x + (xy) = x$$

Esempio 8. Si ottimizzi la seguente funzione in somma di prodotti:

$$O = \overline{xy}z + \overline{xy}z + x\overline{y}z + x\overline{y}z + xyz$$

Utilizziamo l'assorbimento; per prima cosa bisogna vedere se sono presenti termini a distanza di Hamming 1 per semplificarli; ripetere il processo fin quando non è più possibile.

$$\begin{aligned}
 O &= \overline{xy}z + \overline{xy}z + x\overline{y}z + x\overline{y}z + xyz \\
 &= \overline{xy}(\bar{z} + z) + \overline{y}z(x + \bar{x}) + \overline{y}z(x + \bar{x}) + x\overline{y}(\bar{z} + z) + xz(\overline{y} + y) \\
 &= \overline{xy} + \overline{y}z + \overline{y}z + x\overline{y} + xz \\
 &= \overline{y} + \overline{y} + xz \\
 &= \overline{y} + xz
 \end{aligned} \tag{2.1}$$

In parole molto povere, bisogna vedere uno per uno tutti i termini che presentano un solo bit di differenza fra di loro, ovvero a **distanza di Hamming 1**, e rimuovere le due parti inverse, poiché equivarranno ad 1. Nello svolgimento dell'ottimizzazione, chiameremo

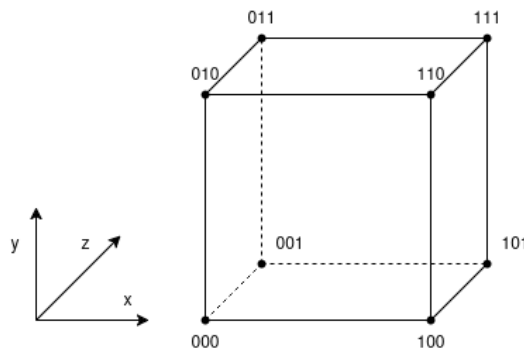


Figure 2.4: Cubo delle distanze fra codifiche

i termini che non è più possibile ridurre come **implicanti primi**; dove è vero che li vogliamo, è possibile che più implicanti primi coprano lo stesso mintermine, creando una ripetizione inutile. Ci è quindi necessario cercare gli **implicanti primi essenziali**, ovvero un implicante che tocca un mintermine non raggiunto dagli altri.

Subentra di conseguenza la questione della **copertura minima**, ovvero il problema di trovare il numero minimo di implicanti necessari per coprire ogni mintermine; per lavorare useremo i due metodi nelle prossime sezioni.

2.2.1 Mappa di Karnaugh

Da me soprannominato "Il Sarrus dell'ottimizzazione", è un algoritmo utilizzabile con funzioni $f(B^3)$ o massimo per $f(B^4)$. Servirà prendere il cubo delle distanze visto prima ed immaginare di piallarlo. Usciranno le celle corrispondenti agli spigoli del solido.

Una particolarità è che il cubo deve essere visto come se fosse impossibile andare out of bounds; quindi una volta superato un lato estremo, ci si trova subito dopo in quello opposto, esattamente come in Pacman. Le celle adiacenti sono quelle a distanza

$\begin{smallmatrix} x & y \\ z \end{smallmatrix}$	00	01	11	10
0	1	0	0	1
1	1	0	1	1

Figure 2.5: Mappa di Karnaugh

di Hamming 1 e la funzione ottimizzata è quella di prima, con risultato $O = \bar{y} + xz$.

2.2.2 Algoritmo di Quine Mc-Cluskey

Trattasi del Laplace di questa materia, sarà l'algoritmo che useremo per l'ottimizzazione dei circuiti combinatori. Può essere utilizzato con ogni tipo di funzione, ma aumenta di complessità in base a quante entrate presenta. Generalmente, segue i passaggi:

1. Partendo dalla tabella di verità completa, prendere i suoi mintermini e ordinarli in base al numero di 1 contenuti nella combinazione di ingresso.
2. Confrontare ogni configurazione con tutte quelle del gruppo successivo. Se presenta distanza di Hamming a valore 1 (più semplicemente, se le due combinazioni di

entrate sono uguali tranne per un input), inserire un **don't care** dove gli inputs differiscono.

3. Ottenuta la tabella ridotta, ripetere il punto 2 fin quando non è più possibile ridurre i mintermini. Fatto ciò, abbiamo ottenuto gli implicant primari.
4. Creare una seconda tabella indicando sulle y i mintermini e sulle x le entrate. Lo scopo è trovare quei mintermini sufficienti per coprire tutti gli inputs, saranno gli implicant primari essenziali.
5. Scrivere la soluzione in somma di prodotti. Finito.

Metodo decisamente macchinoso, che tuttavia permette di ottenere tramite semplici passaggi la funzione ottima. Segue immagine per evidenziare il caso base.

2.2.3 Funzioni parzialmente specificate

Quine Mc-Cluskey è versatile. È possibile utilizzarlo anche con funzioni, le quali hanno combinazioni risultanti come don't cares. L'algoritmo non cambia molto, bisognerà solamente considerare il DC-set come altri mintermini per poi ignorarlo nella tabella finale.

2.3 Dispositivi programmabili

I dispositivi programmabili sono componenti hardware dotati di porte **PROM**, programmable read only memory. La loro implementazione comporta alcune conseguenze:

- Essendo il circuito programmabile, è modificabile anche dopo la sua produzione, quindi, in caso di necessità, si risparmierà sui materiali.
- Eventuali bug presenti nel circuito potranno essere facilmente risolti grazie a questa flessibilità.
- La logica di programmazione copre necessariamente spazio nell'area; verrà sempre utilizzata, richiedendo energia.
- Risulta tendenzialmente più lento rispetto ad un dispositivo dedicato.

Una prima evoluzione dei circuiti PROM sono i **PLA**, i programmable logic array, con due livelli di porte logiche. Col tempo si evolsero in **CPLD**, complex programmable logic devices, un'interconnessione di più circuiti PLA su una scheda di silicio. Dopodiché nacquero gli **FPGA**, field programmable gate array, che sono i dispositivi programmabili odierni. Sono composti da vari CPLD. Infine abbiamo gli **SoC**, system on chip, l'evoluzione più recente utilizzata su vasta scala. Si tratta di una FPGA controllata da una CPU.

2.4 Sintesi combinatoria multilivello

Finora è stato visto come sintetizzare circuiti combinatori ad un massimo di due livelli; è tuttavia possibile espandere il concetto ad n livelli, accettando un compromesso di ottenere soluzioni approssimate, tramite tecniche euristiche. In tal merito, introduciamo il concetto di **ritardo**, il valore tempo impiegato per ottenere gli outputs da un circuito, il quale è dato dal **cammino critico**, il percorso più lungo che la corrente attraversa dall'entrata all'uscita. Le tecniche sovramenzionate procedono su due passi principali:

1. Ottimizzazione eseguita ignorando i vincoli implementativi, come quelli imposti dalla libreria tecnologica o al fanin/fanout.
2. Raffinamento del precedente risultato in base ai vincoli implementativi.

Non sarà un risultato ottimo come lo darebbe la fattorizzazione, tuttavia riduce il carico computazionale. Per lavorarci, introduciamo i relativi **modelli di rappresentazione** e le **trasformazioni** con le quali si andrà ad operare.

La struttura di un circuito può essere rappresentata tramite una **rete logica**, una interconnessione di nodi associati a funzioni booleane ad una sola uscita, l'ultime chiamate **funzioni scalari**. Questa collega i comportamenti delle funzioni dei nodi e viene rappresentata tramite **grafi orientati aciclici**, "DAG" $G(V, E)$, dove;

- V è l'insieme dei nodi, diviso in V^I (nodi d'ingresso), V^O , (nodi d'uscita) V^G , (nodi interni a cui è associata una funzione scalare)
- E è l'insieme dei lati.
- Ad ogni nodo si associa una variabile temporanea.

Bisognerà quindi capire su quale parte operare per ottimizzare; il ritardo, riducendo i tempi di esecuzione, o l'area, riducendo il carico computazionale? Tenzialmente si cerca di trovare un equilibrio fra i due e gli algoritmi o **scripts** usati si dividono in algoritmi di **ottimizzazione**, che riducono il livello di complessità, e di **ristrutturazione**, i quali restaurano il circuito ad uno stato più complesso ma meno carico. Si useranno esclusivamente strumenti automatici.

2.5 Mapping tecnologico

Il **mapping tecnologico** è il problema di implementazione dei circuiti sequenziali mediante le porte logiche di una data libreria tecnologica.

Una volta completato il processo di minimizzazione tramite gli scripts, si otterrà un risultato in somma di prodotti, rappresentativo della minima realizzazione teorica.

Ci occuperemo ora di mapparlo su una libreria tecnologica con lo scopo di renderlo realizzabile per la specifica architettura.

L'algoritmo alla base del processo è detto **tree-mapping**, una dispiegazione dei nodi come se fosse un grafico ad albero. L'unico limite risulta, quindi, la necessità di non avere nodi predecessori. Il circuito con la rispettiva libreria, ambo convertiti in grafici albero, potranno poi essere scritti sulle seguenti architetture:

- **ASIC**: Il cui processo è diviso in due parti:
 1. **Placing**: Posizionamento delle celle nella scheda.
 2. **Routing**: Collegamento delle celle fra loro.
- **FPGA**: Il cui processo di placing corrisponde alla programmazione delle celle, le quali hanno egual dimensione e capacità.

Chapter 3

Progettazione Digitale

3.1 Circuiti sequenziali

Nella progettazione dei sistemi digitali ci sono due principali classi di sistemi:

- **Circuiti combinatori:** Il valore delle uscite dipende interamente dalla combinazione di inputs.
- **Circuiti sequenziali:** Il valore delle uscite dipende dalla combinazione di input attuale e dalle precedenti ricevute.

Semplicemente, quando si parla di sistema sequenziale, abbiamo la possibilità di salvare in memoria uno **stato**, la compressione di tutto ciò che il sistema ha eseguito fino a quel momento. La loro necessità è presto detta, vedendo solo come un ciclo distrugge i circuiti combinatori.

Esistono anche due sottoinsiemi dei circuiti sequenziali, ovvero gli **asincroni**, indipendenti dalla variabile del tempo **clock**, e i **sincroni**, dalla quale sono dipendenti per un corretto funzionamento.

3.1.1 FSM - Finite State Machines

Le **macchine a stati finiti** stanno alla base dell'informatica tutta; hanno lo scopo di salvare e mostrare l'evoluzione del programma fra gli stati in base ad una combinazione di ingresso e generandone una di uscita. La sua funzione è data da:

$$M = \langle S, I, O, \delta, \lambda, s \rangle$$

Dove le parti sono:

- S : Insieme degli stati, necessariamente finito e non vuoto.

- I : Alfabeto di ingresso, $|I| = 2^n b$.
- O : Alfabeto di uscita, $|O| = 2^m b$.
- δ : Funzione allo stato prossimo. Riceve stato e ingresso correnti per ritornare il successivo. $\delta = S \times I \rightarrow S$.
- λ : Funzione di uscita; la sua definizione dipende dal tipo di FSM usata:
 - **Macchina di Mealy**: Genera l'uscita in base a stato ed input correnti. $\lambda = S \times I \rightarrow O$.
 - **Macchina di Moore**: Genera l'uscita in base allo stato corrente. $\lambda = S \rightarrow O$.
- s : Stato iniziale, probabile che non venga specificato.

Le FSM si possono rappresentare mediante i seguenti due costrutti:

State Transition Table

Per ogni coppia [STATO/INPUT] si indica lo stato prossimo e l'uscita. Nelle colonne saranno inseriti i simboli di ingresso, mentre nelle righe lo stato corrente. Le intersezioni dipendono dal tipo di macchina scelta.

$$M = \langle \{A, B, C\}, \{0, 1\}, \delta, \lambda \rangle$$

State Transition Graph

Costrutto matematico costituito da archi e nodi collegati. I nodi rappresentano gli stati, mentre gli archi lo spostamento da uno stato all'altro. Sono basati sulle state transition tables e vanno interpretati come dei percorsi.

Lo scopo di questi costrutti è la formalizzazione di concetti a partire dal linguaggio naturale, rendendoli eseguibili in modo non ambiguo dalle macchine che si andranno a progettare. Partendo quindi dalle specifiche si può definire il corretto *modus operandi*.

Esempio 9. FSM

La seguente macchina ha 1b di entrata e 1b di uscita; se nelle entrate si legge una sequenza pari di 0 seguita da una sequenza dispari di 1, l'output ritornerà 1. L'output sarà 0 quando il numero diventa pari o si riceve uno 0.

3.2 Sintesi delle funzioni λ e δ , codifica degli stati

L'argomento di maggiore importanza per il momento sono i circuiti sequenziali sincroni, nello specifico, quindi, macchine a stati finiti dipendenti da un clock. L'immagine a

sinistra rappresenta il **Modello di Huffman** e mostra come gli outputs dipendano sia da logica combinatoria che da clock.

Per far sì che la macchina funzioni è necessario distinguere gli stati ad essa appartenenti; ciò si fa attraverso una loro codifica, la cui stringa risultante è salvata nei **registri**.

Per attuare questo processo bisogna assegnare un valore binario ad ogni stato. La formula generale è:

$$\log_2 n, \text{ con } n \text{ il valore totale degli stati.}$$

Un altro metodo di codifica consiste nell'utilizzo della **1-hot**, dove la posizione dell'1 e non il valore binario indica il valore dello stato. La codifica degli stati risulta utile per una maggiore leggibilità dei grafici utilizzati prima.

3.3 Minimizzazione degli stati

L'utilità della minimizzazione è sempre la stessa; la riduzione delle risorse necessarie atte al funzionamento della macchina. Sebbene i casi di studio rimangano gli stessi, ovvero macchine completamente e non completamente specificate, qui è necessario che nella FSM esista almeno una relazione di **equivalenza fra stati**. Deve quindi avere:

- **Riflessività:** $S_i \sim S_i$, ogni stato è equivalente a sé stesso.
- **Simmetricità:** $S_i \sim S_j \implies S_j \sim S_i$, ovvero proprietà commutativa.
- **Transitività:** $S_i \sim S_j \wedge S_j \sim S_k \implies S_i \sim S_k$.

Parlando in termini più concreti, due macchine a stati dovranno avere entrate e uscite siano equivalenti per essere chiamate tali. Avremo quindi la seguente logica per due funzioni di output:

$$\lambda(S_i, I_\alpha) = \lambda(S_j, I_\alpha)$$

Con questa premessa possiamo definire una macchina **minima** quando non esistono più coppie di stati equivalenti. Come conseguenza diretta, la macchina risulta essere anche unica.

L'algoritmo di ottimizzazione è quello di **Paull-Unger** e si dà come scopo la ricerca di stati equivalenti con conseguente assorbimento. Presenta alcuni casi di studio:

Esempio 10. Algoritmo di Paull-Unger

Osserva attentamente la STT; noterai che alcuni stati, in base all'entrata, si sposteranno su un medesimo luogo o torneranno una stessa uscita. Se entrambi combaciano, significa che gli stati sono equivalenti, mentre se solo gli outputs sono uguali, l'equivalenza dipenderà dagli stati prossimi in cui si muoveranno quelli correnti.

Osserviamo che gli stati correnti A, B presentano lo stesso output, ma stati prossimi diversi. Solamente se B e C sono equivalenti si potrà considerare tale anche A . In questo caso non è così.

Gli stati equivalenti ora vengono rinominati con una **classe di equivalenza**, ha lo scopo di essere il "portavoce" di tutti loro. Poi si continua a vedere se esistono ulteriori equivalenze. Se sì, ripetere il processo, altrimenti si è ottenuta la macchina minima.

Esempio 11. Algoritmo di Paull-Unger, alta complessità

Data la seguente macchina, è chiaro che procedere con il precedente algoritmo sia tedioso e possa portare a errori. Per semplificare il tutto, si raggrupperanno gli stati da esaminare in **cricche** in base alle loro dipendenze. Per esempio, l'equivalenza $B \sim E$ dipende da quella $D \sim F$. Nel grafico sottostante si collegheranno tutte le dipendenze.

Ottenute tutte le cricche, sarà possibile raggruppare gli stati collegati in classi di equivalenza, per poi procedere con l'algoritmo avendo dati meno complessi.

Esempio 12. Funzioni parzialmente specificate

Il raggruppamento in cricche qui è necessario; bisogna trovare quella massima (che raggruppa più stati possibile con meno condizioni) e darla per vera, per poi confrontarla con gli stati restanti mediante Paull-Unger.

3.4 Datapath e componenti

Il **Datapath** è l'unità di elaborazione composta da un'aggregazione di componenti elementari. Principalmente esistono due tipi di componenti:

- **Unità di memoria**, usati per la memorizzazione dei dati.
- **Unità funzionali**, usate per l'elaborazione dei dati.

Per la scrittura di un datapath è fortemente consigliato di definire le sue funzioni prima di iniziare i lavori, per poi procedere con la creazione delle componenti elementari, ed infine il loro collegamento. Adesso andiamo a vedere le varie componenti di base:

- **Registri**: Unità per il salvataggio di dati, si differenziano per le modalità di salvataggio in base al clock, ovvero definiamo **rising-edge** un registro che si modifica quando riceve un clock positivo, **falling-edge** se lo fa quando il clock è negativo e **asincrono** quando lo fa a prescindere da esso. Ne esistono di tre tipi:
 - **Parallelo/parallelo**: Riceve bits in input e si aggiorna ricevendo il segnale di clock, per poi dare in output il dato salvato dal ciclo seguente.
 - **Seriale/seriale**: Riceve un bit alla volta e posiziona l'ultimo ricevuto nella posizione più significativa. Ritorna in output un bit alla volta la posizione meno significativa.

- **Parallelo/seriale:** Svolge entrambe le funzioni appena viste, la scelta è in base ad un segnale aggiuntivo.
- **Multiplexer:** Riceve più entrate da n bits, sceglie quale trasmettere in output in base ad un segnale di controllo.
- **Demultiplexer:** Riceve un'entrata da n bits e decide a quale ramo di output spedirla in base ad un segnale di controllo.
- **Decoder:** Modifica la dimensione in bits del segnale ricevuto in input.
- **Shifter:** Riceve un valore in input ed effettua shiftL o shiftR in base ad un segnale di controllo.
- **Unità logiche:** Tutte le porte logiche viste in precedenza, il cui funzionamento dipende dalla relativa tabella di verità.
- **Unità aritmetiche:** Tutte le porte che eseguono le quattro operazioni elementari in base 2.
- **Operatori di confronto:** Tutte le porte di confronto valori.

Un esempio di datapath completo è la ALU, **Arithmetic logic Unit**, che consente di effettuare somma, sottrazione, comparazione e riportare il risultato.

3.5 Modello FSMD - Finite State Machine with Datapath

Una FSMD è una macchina sincrona, il connubio fra le due macrocomponenti viste prima e sta per **Macchina a stati finiti con unità di elaborazione**. Si compone di due parti; quella di controllo e quella di elaborazione, le quali comunicano mediante segnali di stato e controllo.

Abbiamo già discusso della progettazione delle sue macrocomponenti, quindi ora bisogna capire quale creare prima e che tipo di segnali di controllo implementare. Non esiste propriamente una risposta corretta, ma bisogna adattarsi a ciò che viene più comodo fare.

3.6 Derivazione FSMD da algoritmo

È possibile ottenere un modello FSMD attraverso specifici algoritmi per una **sintesi ad alto livello**, portando ad una soluzione buona. Questi svolgono i seguenti passaggi:

1. **Scheduling:** Organizzazione dei cicli di clock nei quali avvengono le operazioni.
2. **Allocation:** Organizzazione delle risorse della macchina in base allo scheduling

In alternativa, un altro modo per ottenere una FSM, seppur più macchinoso e meno automatico, è scrivere un algoritmo in un linguaggio di programmazione diverso oppure in pseudocodice, con lo scopo di capire quali componenti siano necessarie all'adempimento del compito.

3.7 Modello dispositivo programmabile

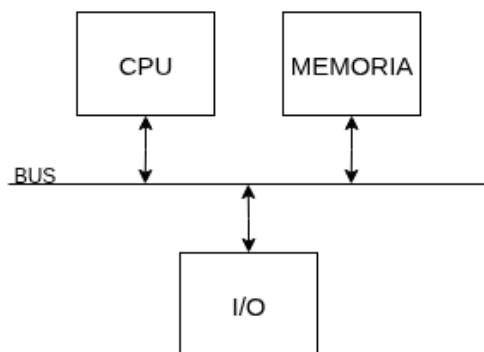
Ogni macchina vista finora è considerato un sistema embedded, il quale svolge, appunto, un solo compito. Tuttavia è possibile progettare anche dispositivi programmabili, che svolgono diversi algoritmi in base a determinate entrate. Un esempio è una piccola calcolatrice capace di eseguire le quattro operazioni.

Chapter 4

Architetture dei Calcolatori

4.1 Modello di Von Neumann e Unità funzionali del calcolatore

Il **Modello di Von Neumann** è un'architettura composta da tre componenti principali interconnesse mediante un dispositivo detto **BUS**¹. Si tratta del modello sul quale si basano tutte le architetture dei calcolatori. Le sue parti sono:



- **Processore:** Atto all'elaborazione dei dati.
- **Memoria:** Atta al salvataggio dei dati.
- **Dispositivi I/O:** Periferiche varie come microfoni o tastiere.

Più nello specifico, un calcolatore elabora i dati grazie all'ausilio delle seguenti tre componenti, ognuna indipendente dall'altra:

- **I/O:** Unità di ingresso ed uscita.
- **ALU:** Unità aritmetico-logica.
- **CU:** Control Unit, che compone processore e memoria.

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

Le prime due lavorano sotto supervisione e controllo della **CU**. L'unità di input riceve informazioni in forma codificata da operatori o periferiche, le quali saranno elaborate dalla **ALU** per effettuare calcoli, eventualmente salvando in memoria i risultati. Quanto eseguito verrà inviato all'unità di output, la quale ritornerà il tutto.

Le informazioni manipolate sono categorizzate in **istruzioni** e **dati**; le prime sono 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 potrà svolgere algoritmi elaborando i dati. Se in esecuzione, si troverà sempre in memoria, a meno che non venga dato un comando di interruzione.

L'ambiente di lavoro del corso sarà la CPU Intel 80x86, con 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 **ISA**, Instruction Set Architecture, ed è letto da un **assemblatore**, il quale provvederà a tradurlo in codice oggetto. Analizziamo ora le microcomponenti della CPU non ancora viste:

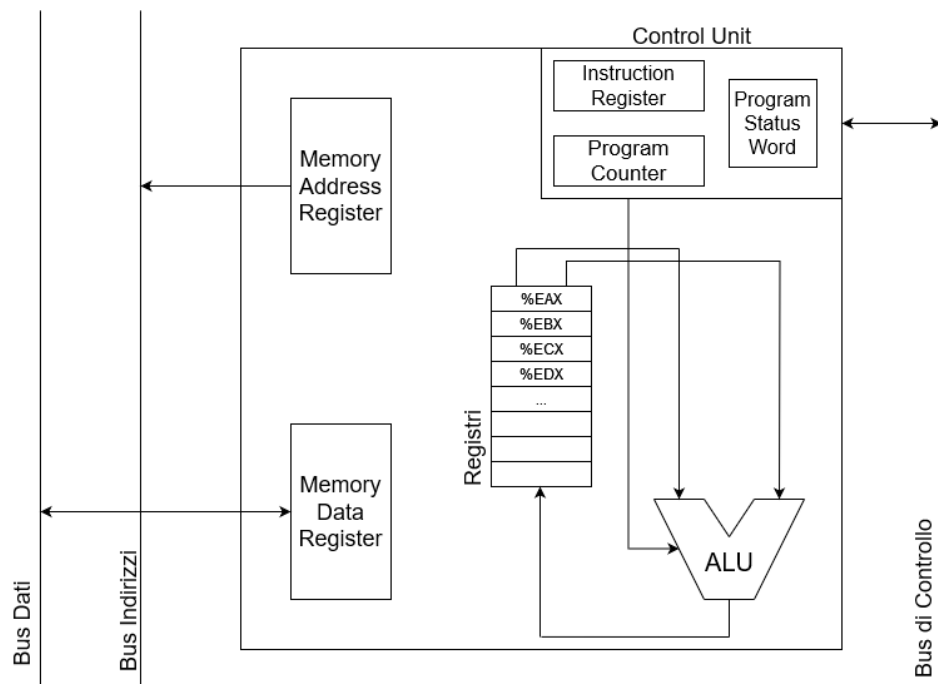


Figure 4.1: Architettura di una CPU

- **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.
- **Componenti della control unit:**
- **Instruction Register:** Contiene gli identificativi delle istruzioni.
 - **Program Counter:** Contiene gli indirizzi delle stesse.
 - **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.

Ora che sappiamo da cosa è composta, è il momento di chiederci come funziona questa CPU. Ci è possibile descrivere tale processo mediante una FSM a tre stati, chiamati **Fetch**, **Decode** ed **Execution**.

Il primo rappresenta la ricezione delle informazioni. Ottiene i dati dall'**MDR**, memory data register, per poi passare tutto all'**IR**, instruction register con il bus dati. Nel frattempo, il program counter aumenterà di 1 ad ogni istruzione ricevuta.

Il secondo stato è la fase di decodifica delle istruzioni; tramite l'ISA della macchina, si indirizzano opportunamente i dati che verranno poi elaborati.

Il terzo ed ultimo stato è infatti quello dell'esecuzione delle istruzioni decodificate.

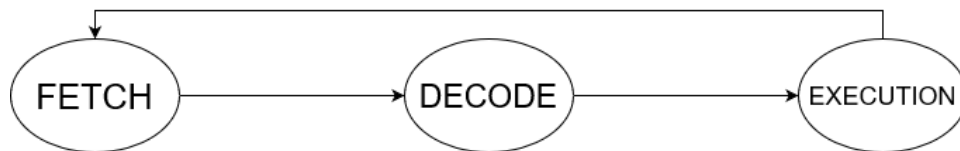


Figure 4.2: Macchina a stati della CPU

L'ISA, come si può facilmente dedurre, non è uguale per ogni singolo calcolatore, tuttavia presenta sempre lo stesso scheletro, composto da:

- **OPcode:** Il codice operazione. Comunica 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.

La modalità di scambio e trasmissione dati si chiama **indirizzamento**, i cui tipi sono discussi più nello specifico nella sezione relativa ad Assembly.

4.2 Architettura CPU RISC-V

Come precedentemente menzionato, le CPU organizzano il lavoro del calcolatore, e più nello specifico ciò avviene grazie ad un ciclo di **fetch**, **decode** ed **execution**. Segue i passi:

1. Il program counter si modifica per puntare all'istruzione successiva.
2. Viene determinato il tipo dell'istruzione letta.
3. Se l'istruzione usa una word in memoria, si determina dove essa si trovi.
4. Se necessario, si carica la word in un registro della CPU.
5. Esecuzione dell'istruzione.
6. Ripeti il passo 1.

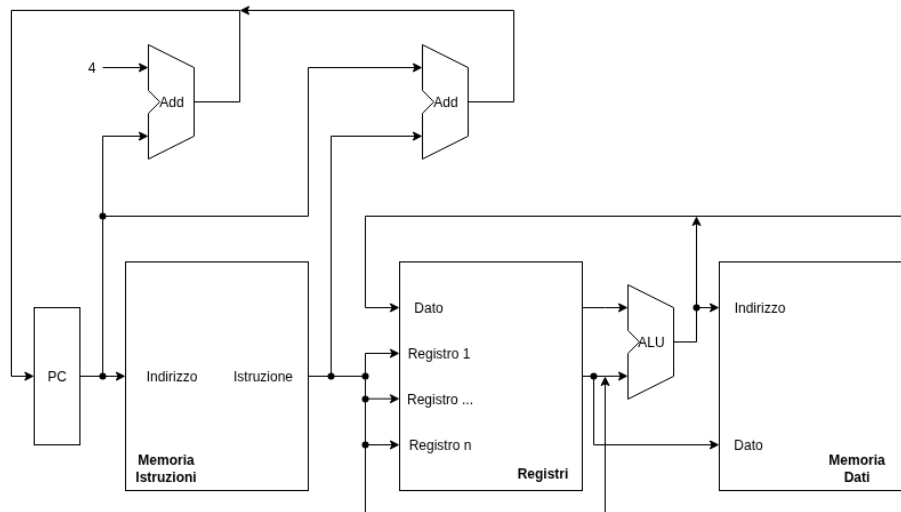


Figure 4.3: Architettura semplificata della control unit

A partire da questo disegno, è necessario andare più nel dettaglio con alcune componenti. Per esempio, è importante notare i due adder in alto. Questi consentono di spostarsi fra le varie istruzioni; infatti il primo all'estrema sinistra è quello che aggiorna il program counter sommandogli $4B$, spostandolo all'istruzione successiva, mentre il secondo adder è utile per le istruzioni di salto. Ottenuto il nuovo indirizzo, il program counter verrà aggiornato.

Bisogna inoltre tenere a mente che i dati da passare alla ALU sono contenuti nei **registri**; in base al segnale dato dalla control unit, si effettuerà un'operazione specifica. Detto ciò, possiamo elaborare sul ciclo di elaborazione informazioni:

- **Fetch**, la selezione della parola corrispondente all'istruzione da eseguire. Qui il program counter fornisce l'indirizzo di memoria in cui si trova la prossima istruzione. Questo valore è ottenuto da una memoria read only chiamata **memoria delle istruzioni**.
- **Decode**, la decodifica dell'istruzione in codice oggetto. In primo luogo la ALU decodifica le istruzioni ricevute in base al valore del program counter e, se necessario, verranno caricati gli operandi dalla **memoria dei dati**.
- **Execution**, l'esecuzione dell'istruzione. Qui si possono effettuare processi diversi, come:
 - Eseguire un calcolo con la ALU.
 - Elaborare il contenuto degli operandi e determinare un indirizzo di memoria.
 - Eseguire un confronto per effettuare dei salti.

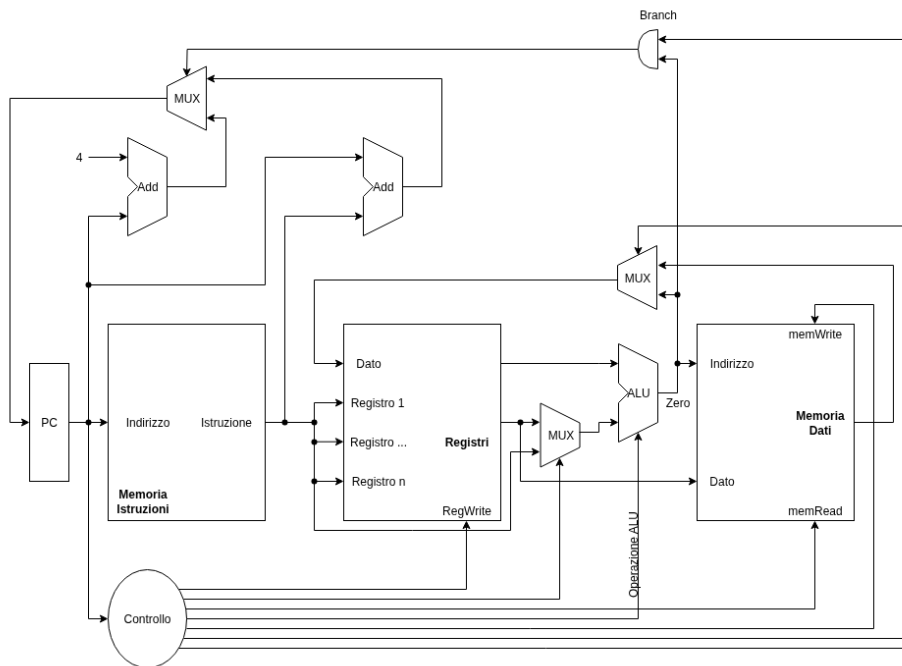


Figure 4.4: Architettura della control unit

Notare adesso nella nuova figura con aggiunta la **control unit** che i segnali dei due adder vanno in un mux, per capire quale utilizzare in base al segnale branch, ovvero di salto, usato come controllo. Quest'ultimo è dato da una porta AND che prende:

- Il risultato della condizione per il jump (zero).
- Il segnale di richiesta di jump (uscito dalla CU).

Se e solo se entrambi i bits sono veri, si va a saltare all'indirizzo richiesto, ed in tal caso il primo parametro è dato dal PC, mentre il secondo è un pezzo corrispondente al path saltato.

Un altro compito della control unit è l'invio di segnali **memoryRead**, **memoryWrite** e **registerWrite**, i quali consentono di leggere e scrivere valori in memoria.

Notare inoltre il multiplexer posto fra la ALU ed il blocco centrale; è necessario per scegliere il luogo da dove prendere l'operando. È possibile ottenerlo dalla memoria oppure direttamente dall'input.

Adesso invece andiamo a vedere nello specifico il comportamento del datapath in base all'istruzione data. Ne abbiamo di quattro tipi:

- **Type-R**: Istruzioni aritmetico-logiche.

I registri **rs1** e **rs2** presentano il numero dei registri sorgenti ed **rd** contiene il numero del registro di destinazione. L'operazione da eseguire sta nei campi **func3** e **func7** ed è letto dalla control unit per comunicare i segnali adatti alla ALU.

- **Type-L**: Istruzioni di caricamento. Necessita di memoria dati e componente per estensione del segno.

Si attiva con il segnale **memRead** a valore vero. Qui **rs1** è il registro base il cui contenuto è sommato al campo immediato di 12b per ottenere l'indirizzo del dato in memoria. Il campo **rd** è il registro destinazione per il valore letto.

- **Type-S**: Istruzioni di salvataggio. Necessita di memoria dati e componente per estensione del segno.

Si attiva con il segnale **memWrite** a valore vero. Qui **rs1** è il registro base il cui contenuto è sommato al campo immediato di 12b per ottenere l'indirizzo del dato in memoria. Il campo **rs2** è il registro sorgente il cui valore è poi copiato in memoria.

- **Type-SB**: Istruzioni di salto.

rs1 e **rs2** sono confrontati. Il campo immediato di 12b è preso, il suo bit di segno viene esteso, shiftato a sinistra di una posizione e sommato al program counter per calcolare l'indirizzo di destinazione del salto.

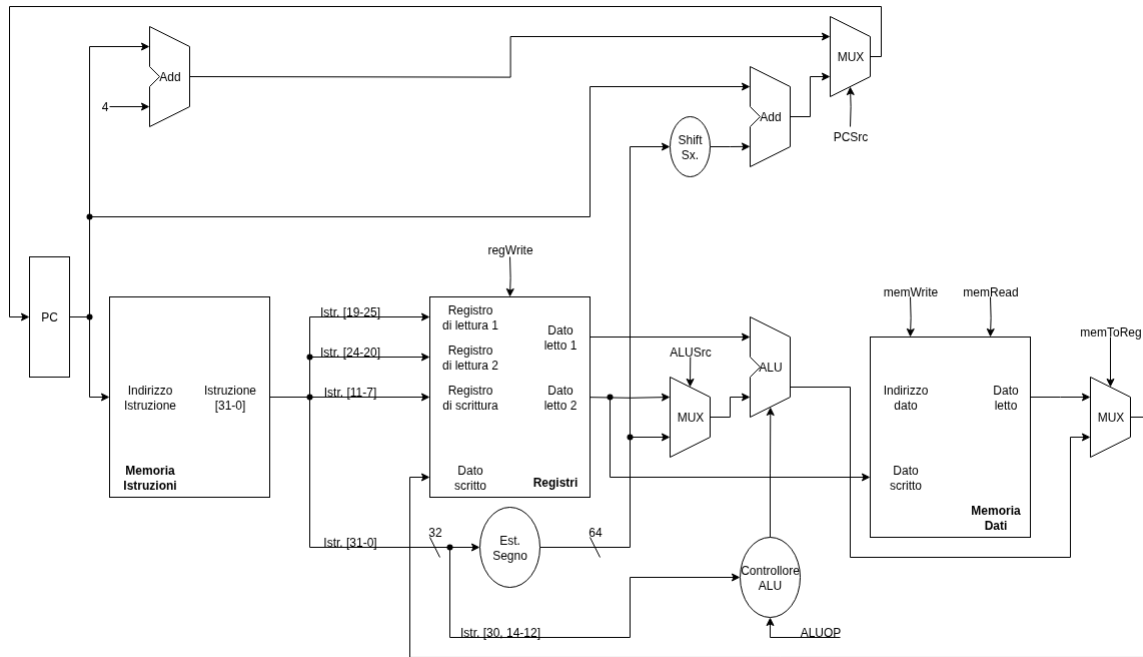


Figure 4.5: Architettura della control unit RISC-V

Esempio 13. Istruzione $[sub\ x4,\ x5,\ x6]$

1. Il PC dà l'indirizzo dell'istruzione e viene incrementato di 4B.
2. L'istruzione è decodificata, viene riconosciuto il tipo-R. Ciò è indicato dal Codop, che è inviato alla CU.
3. In base al Codop, la CU imposta i segnali: **regWrite** = 1, perché scriverà in $x4$, **ALUSrc** = 0, perché gli operandi vengono dai registri, **memWrite** = 0, **memRead** = 0, **memToReg** = 0, perché non c'è accesso alla memoria.
4. La CU legge **func3**=0b000 e **func7**=0b0100000, determinando che l'operazione è una sottrazione.
5. La ALU calcola $x4 = x5 - x6$.
6. Se il risultato è 0, il segnale **Zero** sarà vero, ed il primo è scritto nel register file.
7. Adesso $x4$ contiene il valore $x5 - x6$.

Esempio 14. Istruzione $[and\ x7,\ x8,\ x9]$

1. *PC dà l'indirizzo dell'istruzione e si incrementa di 4B.*
2. *Il decoder la riconosce come Type-R.*
3. *La CU imposta: $\text{regWrite} = 1$, $\text{ALUSrc} = 0$, $\text{memWrite} = 0$, $\text{memRead} = 0$, $\text{memToReg} = 0$.*
4. *La ALU ottiene **func3=0b111** e **func7=0b0000000**, selezionando l'operazione *logicalAND*. Esegue quindi $x7 = x8 \wedge x9$.*
5. *Scrive il risultato nel register file.*
6. *$x7$ contiene il risultato di $x8 \wedge x9$.*

Esempio 15. Istruzione [*lw* $x4$, 16($x5$)]

1. *PC dà l'indirizzo dell'istruzione e avanza di 4B.*
2. *L'istruzione decodificata è type-L, LOAD.*
3. *$x5$ viene letto dal register file per ottenere l'indirizzo base.*
4. *L'offset 16 viene estratto e inviato alla ALU, che calcola l'indirizzo effettivo eseguendo $x5 + 16$.*
5. *L'indirizzo risultante è inviato alla memoria dati, che restituisce il valore ivi memorizzato.*
6. *Il valore letto va nel mux che riceve anche $\text{memToReg}=1$ e $\text{regWrite}=1$.*
7. *Il valore è ora scritto in $x4$, che conterrà infatti il valore memorizzato all'indirizzo $x5 + 16$.*

Esempio 16. Inserisci esempio istruzione di salto.

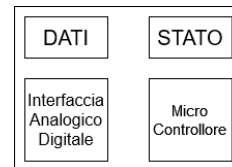
Un'ultima particolarità di cui tener conto è il quantitativo di bits usati per un'operazione. Usiamo load come esempio; avremo:

- **lw**: Load word.
- **lh**: Load half-word.
- **lb**: Load byte.
- **l*u**: Load unsigned, con $* \in \{w, h, b\}$

4.3 Metodi di I/O, Segnale Interrupt

Lo scopo dei dispositivi Input/Output, detti anche **periferiche**, è quello di effettuare uno scambio di dati più naturale fra persona e macchina. Alcuni esempi di queste architetture sono tastiera, mouse o altoparlanti. 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.

Ottenendo questi accessi, è possibile effettuare una **Supervisor Call**². 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**³, 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.

²SVC, Chiamata per passare il controllo delle operazioni al sistema operativo

³Verifica ciclica dei dispositivi I/O mediante testing dei bits di bus di ogni periferica, seguita da un'interazione Read/Write.

Parliamo ora invece di **Interrupt**; un segnale asincrono che interrompe il lavoro della CPU. Quando il suo flag è disattivato, è richiesto l'intervento dell'unità di controllo per riattivarlo. Nello specifico, l'algoritmo si realizza tramite due elementi; **interrupt**

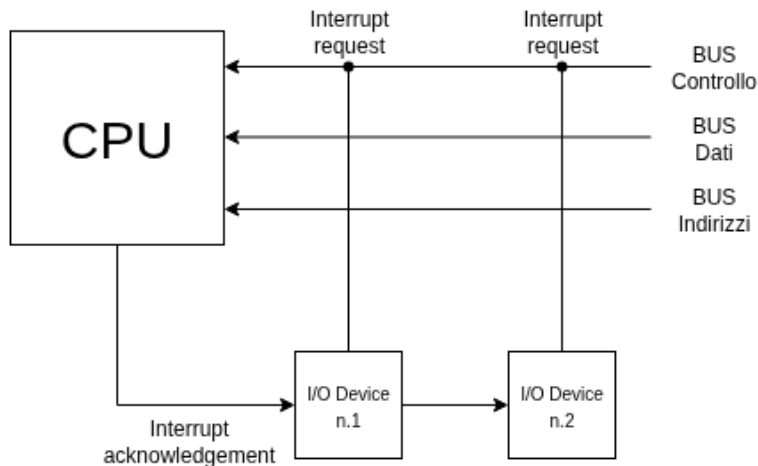


Figure 4.6: Funzionamento dell'Interrupt

request, dato dal dispositivo, aggiorna il valore di interrupt a 0, ed **interrupt acknowledgement**, un segnale proveniente dalla CPU per confermare quale dispositivo ha richiesto l'interruzione. È grazie a quest'ultimo segnale che l'unità di controllo è capace di controllare sequenzialmente le richieste di ogni dispositivo. Una volta trovato il chiamante, chiamerà la **interrupt service routine** ad esso associato.

Il compito della ISV 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. Notare inoltre che le richieste di interrupt non si sovrappongono e sono state create per gestire tempi umani, quindi non si otterranno ulteriori richieste finché la prima non sarà risolta.

La ISV è poi parte integrante del **Device driver**, un programma con lo scopo di ottimizzare e ridurre gli sforzi della CPU legati al funzionamento di una periferica. Sarà discusso più nel dettaglio nella sezione apposita.

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, l'interrupt signal sarà posto a 0 e si inizierà a lavorare con quanto ottenuto.
3. I dati elaborati sono salvati in memoria.

Capiamo subito il fatto che se l'interrupt signal è gestito in questo modo avremo uno spreco di risorse non indifferente; una soluzione al problema è ottenuta tramite il **Direct Memory Access**.

La CPU programmerà 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 relativi registri.

Il vantaggio sta nella divisione del lavoro fra CPU e periferiche, risparmiando energia della CPU; 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 potrebbero voler accedere allo stesso BUS e se questo avesse luogo, si potrebbero perdere dati. 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. Nell'immagine è possibile notare la presenza di segnali nuovi:

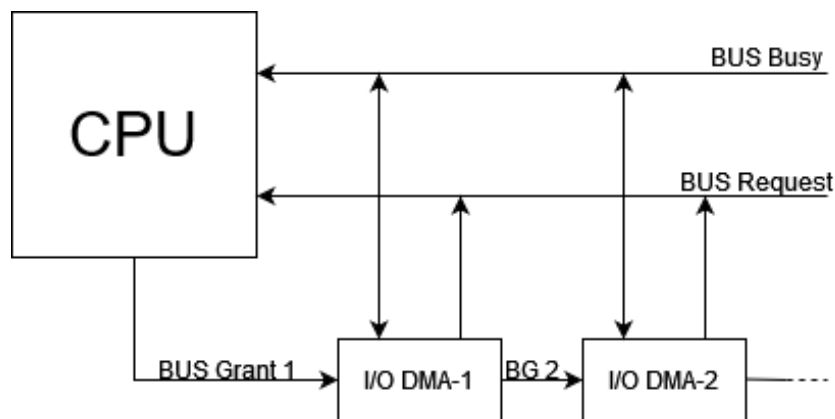


Figure 4.7: Arbitraggio dei BUS

- **BUS Busy:** Segnale che indica se il BUS è attualmente utilizzato da una componente.
- **BUS Request:** Segnale che indica una richiesta fatta alla CPU per poter utilizzare il BUS.

- **BUS Grant:** Segnale di concessione di utilizzo BUS una volta terminato il precedente lavoro⁴.

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 colui che inizia l'operazione, mentre il secondo ne risponde. Esistono due protocolli di operazione che renderanno rispettivamente un BUS sincrono o asincrono.

Negli schemi seguenti verranno usati 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**⁵.

Esempio 17. *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.

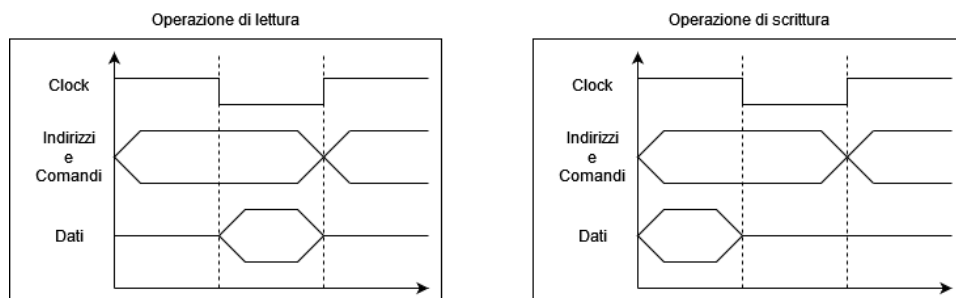


Figure 4.8: Operazioni in un BUS Sincrono

Esempio 18. *BUS 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**, il quale si attiva quando l'operazione è stata ultimata. Sceglieremo i cicli minimi che diranno quale coppia Master/Slave è la migliore.*

⁴La 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

⁵La parte non può agire poiché non riceve segnale

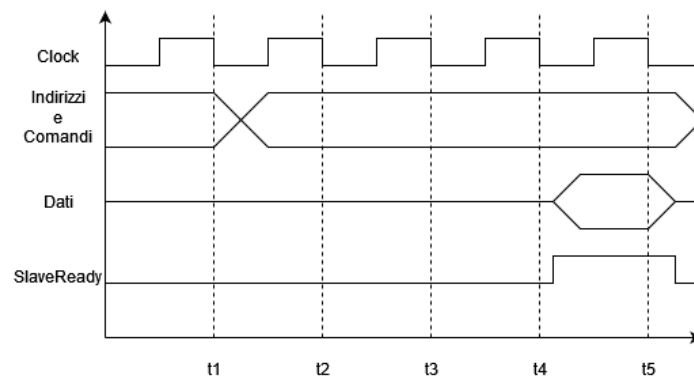


Figure 4.9: Funzionamento BUS Multiciclo

Esempio 19. 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.

Diciamo di voler effettuare una lettura a tempo generico i ; per prima cosa il Master dovrà ricevere il segnale del comando, il quale attiverà **MasterReady**. Ottenuti i dati, li invierà allo Slave che rimanderà tutto al primo come precedentemente visto. Finito di leggere ed 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 una prossima istruzione.

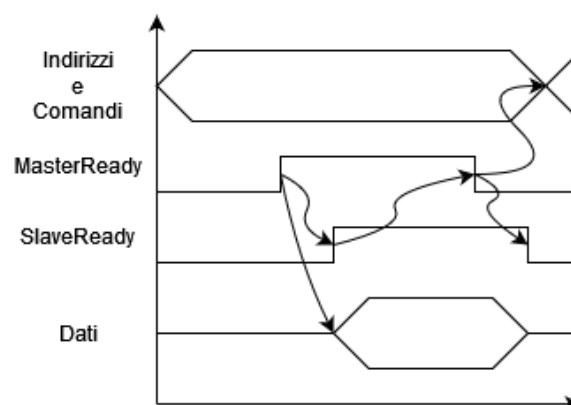


Figure 4.10: Lettura in un BUS Asincrono

4.5 Stati di un processo

I concetti di direct memory acces e arbitraggio BUS sono fondamentali in quanto alla base degli elaboratori contemporanei sta il multitasking, che è reso possibile grazie a quanto visto finora. Più nello specifico, andremo a parlare ora di **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 un **Quanto**, che corrisponde ad $1ms$. Fondamentalmente, comincia il primo processo

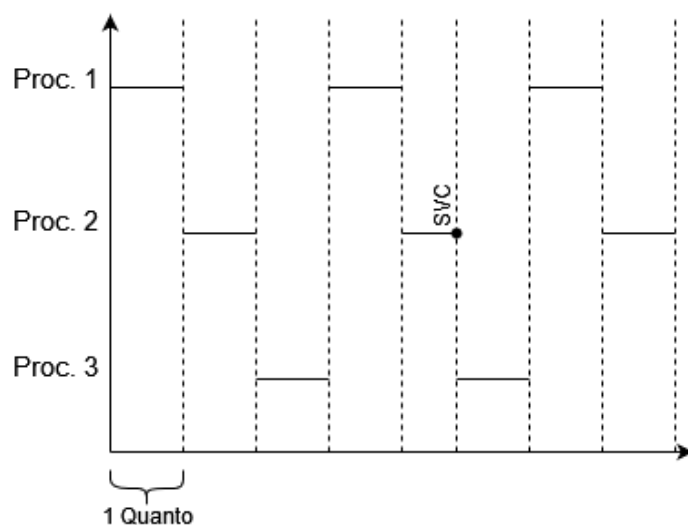
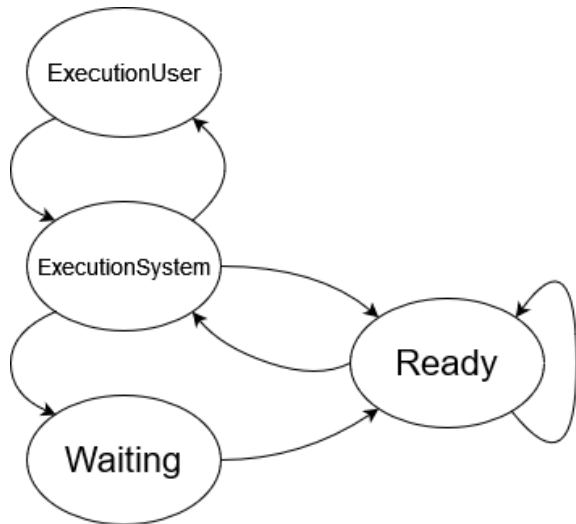


Figure 4.11: Grafico temporale del Time Sharing

che ha a disposizione $1Q$ di tempo per lavorare e quando finisce questo arco di tempo, si passa al processo successivo e così via fino al loro termine. Tuttavia, nell'eventualità di una SVC, 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. L'algoritmo è gestito dal **Kernel** del sistema operativo, il quale vede i progressi di un processo in base allo stato in cui si trovano, ovvero i seguenti:



- **ExecutionSystem:** Dato dalla CPU, è lo stato dove il sistema operativo ha libero accesso.
- **ExecutionUser:** Dato dalla CPU, qui il sistema operativo ha accesso limitato per consentire all'utente di agire in caso di una SVC^a.
- **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.

^aNell'ISA Intel 80x86 le SVC svolgono anche la funzione di interrupt signals e si indicano con "int".

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: ProcessID, Proprietà, Stato della CPU⁶, 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.

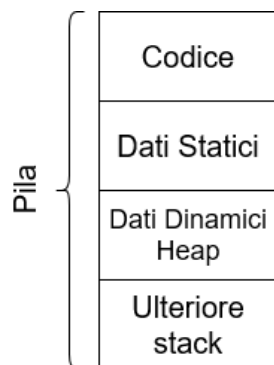
⁶Il salvataggio dello state nella memoria.

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

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.

La stack è capace di allocare le variabili locali e passare parametri a funzioni. Proprio grazie a queste ultime 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.

Mettiamo di avere un semplice programma in C, abilitato all'utilizzo di una funzione; in tal caso avremo che gli eventuali parametri da essa ricevuti saranno posti in cima alla pila, su registri antecedenti quelli della funzione main. Questo perché la stack è gestita con la regola **LIFO**, ovvero last-in, first-out.

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.

4.7 Device driver

Gli elaboratori odierni hanno una caratteristica comune: necessitano di componenti software che compongono il sistema operativo, e che quindi gestiscano il flusso di input/output. In tal merito, nei dispositivi I/O andiamo a distinguere due livelli:

- **Unità di controllo**, hardware.
- **Device driver**, software.

È grazie al connubio fra queste due parti che possiamo avere un'astrazione tale da consentirci di programmare senza tener conto delle routine a basso livello del sistema operativo. Abbiamo già parlato di come ogni dispositivo I/O utilizzi il meccanismo dell'interrupt; tuttavia questo è legato alla frequenza con la quale avvengono le varie chiamate. Si presenta quindi il problema di gestire più device dalle diverse frequenze.

La soluzione ha nome UNIX: rimandare ogni dispositivo I/O al concetto di file, indipendentemente dall'hardware della macchina. Ciò consente di poterci leggere, scrivere, oppure di fare operazioni di controllo input-output. In parole povere, il **device driver** è un file con un algoritmo.

Andando ora a parlare più nello specifico dei sistemi operativi, di norma sono costruiti secondo il modello **onion skin**, dove ogni strato più all'esterno arricchisce quello interno. Si compone di:

- **Hardware**, autoesplicativo.
- **Gestore dei processi**: Consente di gestire più processi grazie al meccanismo di time sharing.
- **Gestore della memoria**: Divide la memoria in tanti sottoinsiemi, ognuno dei quali è dedicato ad un singolo processo. Queste zone di memoria sono inviolabili.

- **Device driver:** Astrae la CPU del dispositivo I/O nel quale è inserita, facendolo vedere come un file. Dà inoltre l'impressione ad ogni processo di avere un dispositivo dedicato.
- **File system:** Gestisce i dispositivi visti sotto forma di files.
- **Gestore rete:** Comunica e gestisce la connessione con la rete internet.
- **Interfaccia utente,** autoesplicativo.
- **Software applicativo:** I programmi.

Dove questo schema rende bene un'idea della struttura dei sistemi operativi, è leggermente datato, perché oggi, per migliori prestazioni, è tutto scritto nel kernel e gestito da appositi moduli:

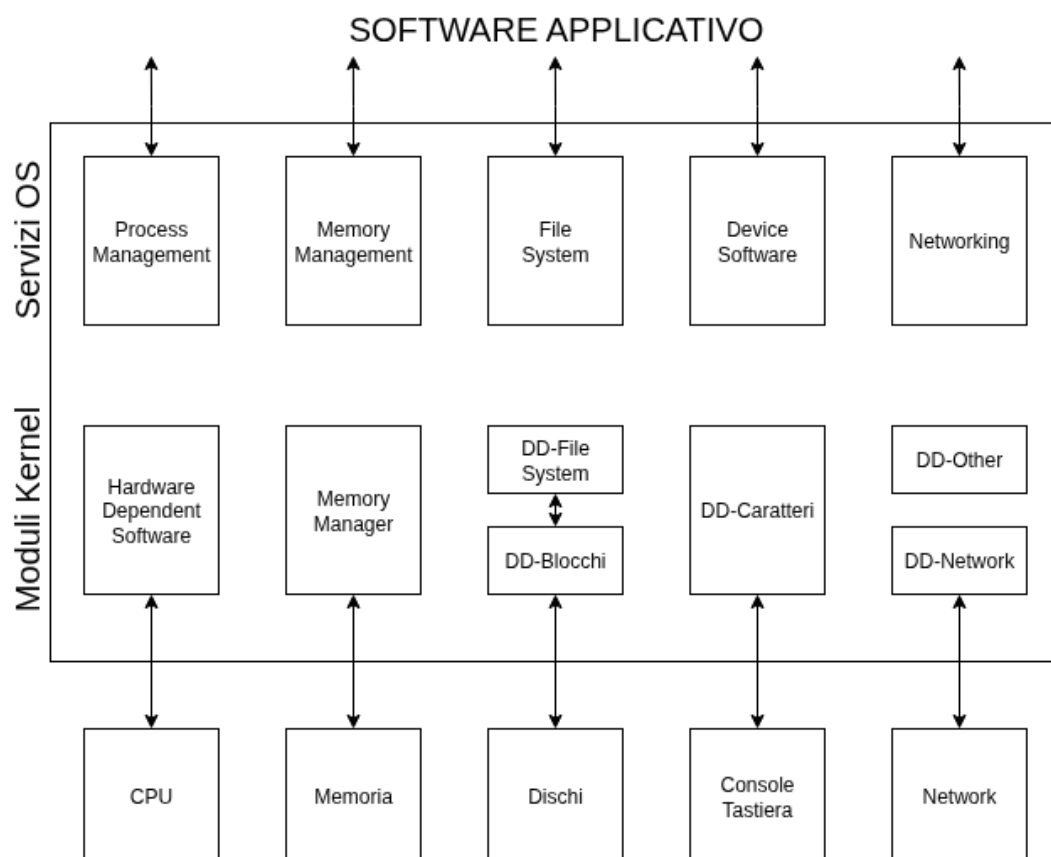


Figure 4.12: Kernel moderno

Come puoi vedere abbiamo i seguenti moduli del kernel:

- **Hardware Dependent Software:** Algoritmi che lavorano a stretto contatto con l'hardware e che ne sono dipendenti.
- **Gestore della memoria:** Permette che solo lo spazio di memoria scrivibile sia segnato da un processo. Gestisce anche la memoria virtuale.
- **Device Driver**, il quale è diviso in ulteriori blocchi:
 - **DD-Caratteri:** Trasferisce caratteri.
 - **DD-Blocchi:** Trasferisce minimo blocchi.
 - **DD-File System:** Gestisce il file system.
 - **DD-Network:** Trasferisce dati in forma di messaggio, ovvero da un intestatario ad una destinazione.
 - **DD-Other:** Usato per ulteriori tipi di dispositivi I/O, come le chiavi USB.

E i relativi servizi del sistema operativo, costruiti in base ai moduli precedenti ed utilizzati con il software applicativo:

- **Device Software:** Gli algoritmi del sistema operativo a livello più alto.
- **Networking:** Gestione della rete.
- **File System:** Gestione dei files.
- **Memory Management:** Gestione della memoria, per esempio come con malloc.
- **Process Management:** Gestisce e consente di creare più processi allo stesso tempo, per esempio come con fork.

4.8 Tipi di Memoria RAM

Parliamo di **Random Access Memory**. Si dice tale perché il tempo di accesso ai registri è indipendente dalla distanza percorsa dai segnali. Si compone di varie celle di bits organizzate e distanziate opportunamente entro un certo numero di bit, rendendole **indirizzabili**⁷. Di fondamentale importanza sono le **WordLines** e le **BitLines**, le quali consentono di individuare un singolo bit di una singola parola. Le linee sono comuni ad ogni tipo di RAM, ma ne vedremo solo i primi due principali: **static RAM** e **dynamic RAM**.

⁷Per accedere alla memoria è necessario sapere dove essa si trova, quindi conoscere il suo indirizzo.

Cominciamo dal primo tipo; la memoria ad accesso casuale statica è generalmente più prestante, nonostante richieda più transistor rispetto alla seconda. Presenta una word-Line orizzontale e due bitLine verticali, queste ultime collegate ad un circuito chiamato **Level Sense**, che è capace di scegliere l'operazione read/write in base ad un segnale dato dal bus controllo⁸. Il risultato è poi spedito al bus dati.

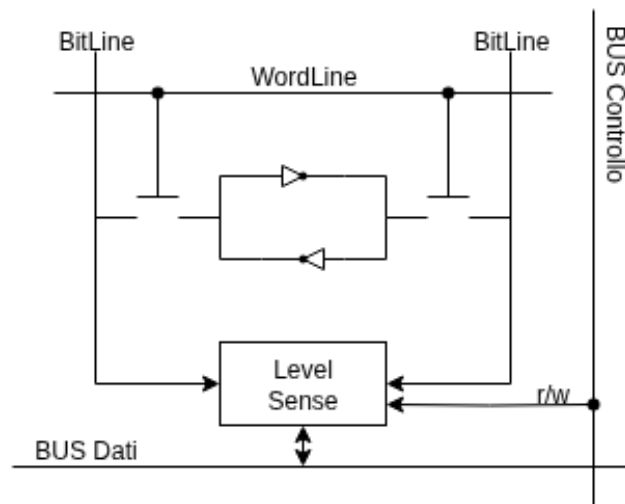


Figure 4.13: Singola cella di S-RAM

Se la singola parola è selezionata, la wordLine avrà valore 1, in alternativa sarà falsa; inoltre, è possibile modificare la parola e non il singolo bit specifico. Per essere precisi, avviene:

- Se si vuole memorizzare 1, sarà presente nella bitLine di sinistra, con 0 nella destra.
- Se si vuole memorizzare 0, sarà presente nella bitLine di sinistra, con 1 nella destra.

Notare che le parole sono composte da $32b$, quindi, il circuito Level Sense è collegato ad uno specifico bit del bus dati; di conseguenza ci saranno tanti circuiti minimi di S-RAM quanti sono i bit del bus dati. In questo caso, 32. Se poi level sense decide di leggere, guarda alla bitLine sinistra e capisce la parte destra in base al valore della prima. Metterà poi il valore sul bus dati. Se scrive, invece, prende il bit dal bus dati e lo riporta in output.

Proviamo adesso ad estendere il concetto e gestirlo su più bit. Conseguo capire che la RAM è un vettore di parole. Supponiamo ora di avere una matrice $1024b \times 32b$, quindi 1024 parole e righe totali con 32 colonne di bits.

⁸Se riceve valore 0, scrive, se 1, legge.

Naturalmente, i level sense devono essere collegati a bus controllo e bus dati, ma abbiamo ora un'aggiunta direttamente dal bus indirizzi: **ChipSelect**. Se vale 1, farà lavorare il level sense scelto, disattivando gli altri.

Per trovare il totale dei bit del bus indirizzi è necessario usare le potenze del 2 per trovare il numero che corrisponde alle righe. Nel nostro caso, $2^{10} = 1024$, quindi avremo 10b di indirizzi. Questi sono poi collegati tramite un decoder che andrà a generare le nostre 1024 wordlines. Selezionando infine un singolo bit dei 1024 si arriverà la singola riga e tutte le altre saranno a 0. Espandiamo le dimensioni e diamoci il compito di creare

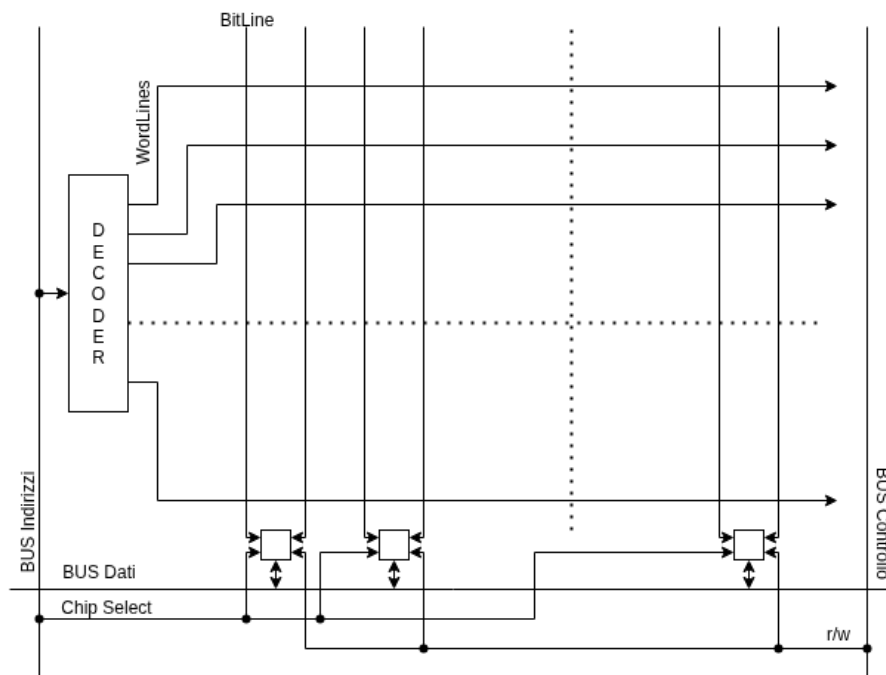


Figure 4.14: Esempio di S-RAM

una matrice di dimensione $1Mb \times 64b$. Avremo:

- Totale bits bus indirizzi: $1Mb = 2^{20} \Rightarrow 20b$
- Totale wordLines: $1Mb = 1024Kb$.

Iniziamo dicendo che le parole sono fatte da massimo $32b$, quindi per gestirne 64 avremo bisogno di più coppie di circuiti S-RAM. Saranno necessarie coppie di circuiti che abbiamo creato prima; il primo elemento gestirà i bit più significativi, mentre il secondo si occuperà degli altri.

Quindi per selezionare il bit corretto si attiveranno le wordLines dei decoder se è necessario un bit meno significativo [0-9], in alternativa verrà usato chipSelect, uscente da un altro decoder supplementare, per prendere i bit più significativi [10-19].

Nella **D-RAM** 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. Questa memoria si compone da una sola word-Line e una bitLine. La cella da $1b$ presenta un transistor collegato ad 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.

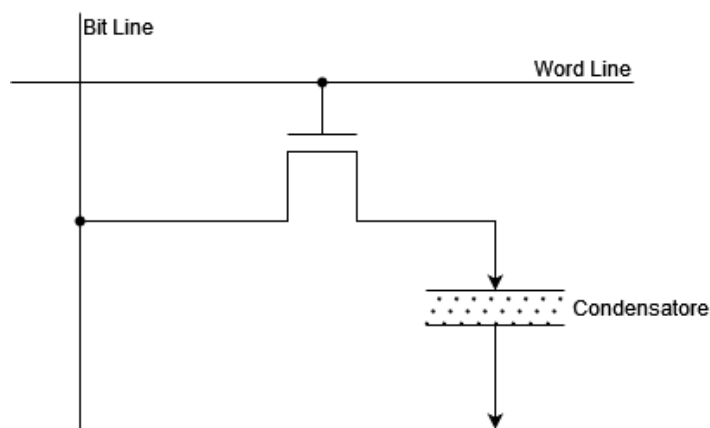


Figure 4.15: Singola cella di Dynamic RAM

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. 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.

4.9 Caratteristiche e gerarchia delle memorie

4.10 Memoria Cache, Paginata e Virtuale

La **memoria cache** è una componente installata direttamente nel microprocessore, non visibile al software e completamente gestita dall'hardware. Il suo scopo è la memorizzazione dei dati più recentemente usati dalla memoria principale ed è capace di velocizzarne gli accessi, aumentando le prestazioni del sistema. Nello specifico, quando un dato si trova all'interno di questa memoria, è preso in un solo ciclo di clock.

La memoria cache è relativamente più piccola della ram; ne consegue che non è possibile accedere ad ogni dato a velocità istantanea. Definiamo infatti:

- **Cache hit:** Quando il dato richiesto è attualmente salvato nella memoria cache.
- **Cache miss:** Quando il dato richiesto non si trova nella cache e deve essere pescato.

Prima di parlare dell'algoritmo di spostamento dati da ram a cache, è necessario chiarire alcune dinamiche e compiere delle operazioni preliminari per assicurarsi delle dimensioni delle componenti prese in esame. Inoltre, le memorie cache e ram non trasferiscono una singola parola, bensì copieranno un'intera pagina, che è un insieme di queste ultime. Ciò avviene poiché per il principio di località, gli accessi successivi si trovano nelle parole successive presenti nella stessa pagina, velocizzando le operazioni. Supponiamo ora di avere $4GB$ di ram e $4MB$ di cache, con pagine grandi $1KB$, allora:

- Dimensione indirizzi ram: $4GB = 2^2 \times 2^{30} = 2^{32} \rightarrow 32b$.
- Dimensione indirizzi cache: $4MB = 2^2 \times 2^{20} = 2^{22} \rightarrow 22b$.
- Dimensione pagine: $1KB = 1024b = 2^{10} \rightarrow 10b$.
- Divisione in pagine della ram: $\frac{4GB}{1KB} = 4M = 2^2 \times 2^{20} \rightarrow 22b$.
- Divisione in pagine della cache: $\frac{4MB}{1KB} = 4K = 2^2 \times 2^{10} \rightarrow 12b$.

Quindi abbiamo già capito la struttura degli indirizzi sia per ram che per cache. Nella prima si necessitano $22b$ per differenziare ogni pagina, mentre per la seconda, $12b$. I bit restanti saranno usati per definire la parola.

Questo era un procedimento generale, tuttavia, l'algoritmo di indirizzamento presenta tre tipi diversi:

- **Diretto:** Necessita di uno spazio apposito dedicato al vettore delle etichette, grande quanto il numero di pagine della cache.

Si pone il compito di dare ad ogni singola pagina della ram una sola posizione in cui andare nelle pagine cache. Si rende quindi necessario dividere i bit della ram in modo diverso:

$10b$ per la parola, $12b$ per la pagina e $10b$ per l'etichetta della pagina.

- **Associativo:** Con questo algoritmo non siamo certi di dove vada una specifica pagina, perché verranno controllate tutte fin quando non si troverà quella corrispondente, come un ciclo for che fa break solo quando l'indice è arrivato ad un certo valore.

Qui vengono tenuti i $10b$ per la parola, mentre i restanti $22b$ della ram sono usati per l'etichetta. Una volta trovata, si darà l'indirizzo in cache.

- **Set-Associativo:** Sinergia fra i due metodi precedenti. Ci è possibile raggruppare delle pagine in insiemi per poter effettuare il controllo delle etichette direttamente in tal gruppo.

Nella divisione dei bits dell'indirizzo in ram, oltre ai 10b per la parola, al posto della pagina avremo, dipendentemente da quanti insiemi sono presenti in memoria, un numero n di bits per i set, mentre per l'etichetta saranno usati i rimanenti.

Questo è il funzionamento in condizioni ideali, ma ora bisogna considerare l'eventualità di avere una memoria cache piena e di aver effettuato un cache miss. Con che modalità si sceglie quali dati tenere e quali rimuovere per fare spazio agli altri? Questo dipende dallo scopo della macchina, ma esistono due **algoritmi di sostituzione** principali e riguardano rispettivamente lo scarico della pagina più vecchia, **LRU**, Least Recently Used, o la più recente, **MRU**, Most Recently Used.

Questi algoritmi funzionano grazie alla presenza di un contatore in ogni pagina, incrementato ad ogni ciclo di clock.

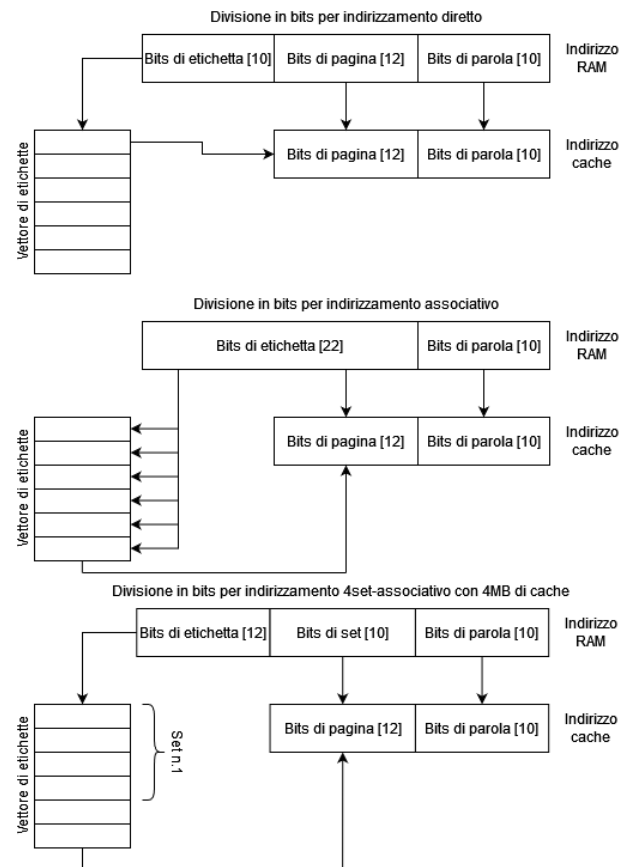


Figure 4.16: Divisione in bit per i metodi di indirizzamento

Negli elaboratori odierni, la memoria centrale fisica non è grande quanto lo spazio di indirizzamento gestibile da parte del processore, ragion per cui, come fatto nelle sezioni precedenti, è necessario dare alla macchina l'impressione di avere a disposizione tutta la memoria per ogni singolo programma; parliamo quindi di **memoria virtuale**. Per questo scopo, introduciamo gli algoritmi di:

- **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.

Partiamo da un semplice presupposto; il processore, per poter lavorare con dei dati o prelevare un'istruzione, genera un indirizzo binario chiamato **indirizzo virtuale**, il quale sarà tradotto in **indirizzo fisico** tramite un algoritmo svolto da sistema operativo ed unità funzionali.

Se l'indirizzo logico fa riferimento ad una parte di spazio di codice in memoria primaria, si otterrà accesso immediato, in alternativa dovrà essere ripescato dal disco fisso e posizionato in cache tramite apposito algoritmo di sostituzione. Il processo di traduzione degli indirizzi è effettuato mediante la **rilocazione**, per poi inviare il risultato al bus indirizzi. Ci sono due modalità:

- **Rilocazione statica**, usata per i sistemi embedded.

Supponiamo che un programma abbia inizio all'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 cicli o costrutti condizionali, il codice si biforcherà dipendentemente dalla loro quantità. Ripetere fino a completa scrittura del programma.

- **Rilocazione dinamica**, apposita per i dispositivi programmabili.

Utilizza due registri, **base** e **limit**, i quali contengono rispettivamente la prima e l'ultima zona di memoria creata per il programma. Questi lavorano insieme ad un circuito che controlla se la somma fra indirizzo logico e base è compresa nella zona di memoria appositata. 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. Supponiamo di voler caricare tre diversi programmi: P_1 , P_2 , P_3 , i quali verranno posizionati in ordine nelle rispettive zone di memoria.

P_1 richiede una malloc, ma non è possibile donare altro spazio poiché la zona ad esso successiva è usata 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 è presente spazio sufficiente per

contenerlo tutto. Ne consegue che è necessario ideare un algoritmo per la gestione dello spazio: la **paginazione**. Si tratta dell'assegnazione ad ogni processo di un numero n pagine indipendentemente dalla loro posizione fisica. Ciò risolve il problema alla radice perché consente di usare zone di memoria non necessariamente adiacenti.

Ma come funziona? Abbiamo la solita divisione dei $32b$ nella memoria RAM, dove $10b$ rappresentano la parola e i restanti $22b$ l'indirizzo logico che poi verrà tradotto in indirizzo fisico. Noi potremmo assegnare a questi bit di indirizzo una pagina logica e una fisica, mantenendo lo stesso procedimento visto nella scorsa sezione. Ciò avviene grazie alla 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 scartarle. Questa unità gestionale è necessaria per due motivi: non appesantire il sistema operativo e sfruttare il principio di località a proprio vantaggio, migliorando le prestazioni. Tuttavia, quanto è grande la tabella delle pagine? Comprenderà certamente l'insieme delle pagine necessarie per il funzionamento del programma, detto **Working Set**.

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.

Arriviamo al dunque; come viene creata ed utilizzata la memoria virtuale? Partiamo dal working set. Come precedentemente menzionato, ogni processo ha il proprio che lavora in solitaria e bisogna convincere il sistema operativo che ha a disposizione tutta la RAM per ogni processo.

Si prende quindi una sezione di memoria su disco, detto **Swap** e donarlo al sistema operativo. Se una pagina del set viene usata poco⁹, verrà spostata in questo swap per liberare spazio in RAM. Notare che l'algoritmo 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 in esso presente, sovrascriverà sempre la pagina più vecchia in ram. È necessario trovare anche qui un compromesso, poiché se allo swap è dato troppo spazio, potrebbero risultare svariati problemi nell'esecuzione dei programmi.

⁹Questo è un controllo eseguito mediante contatori nella tabella delle pagine. Sono incrementati ad ogni ciclo di clock.

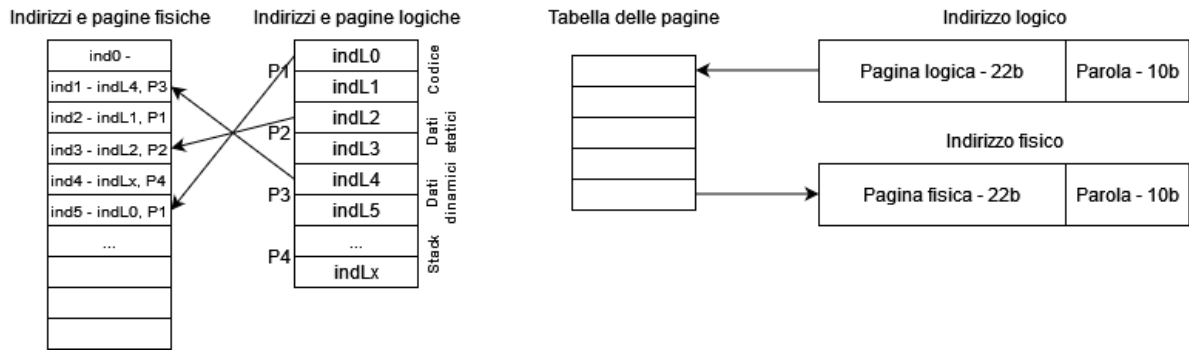


Figure 4.17: Percorso di paginazione

4.11 Pipelining

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

Immaginiamo ora di avere un circuito semplice di somma, dal leggero costo in runtime e con un'architettura ottimizzata con un circuito esterno e registri flip-flop. Avremo:

1. **Fetch:** Ricezione istruzioni in $1CPI$.
2. **Decode:** Decodifica istruzioni con eventuali accessi in memoria, dipendentemente dalla loro necessità si useranno $0CPI$ se non acceduto, $1CPI$ altrimenti.
3. **Execution:** Esecuzione istruzioni in $1CPI$.
4. **WriteBack:** Salvataggio risultato quando richiesto. Usa $1CPI$.

	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

Table 4.1: Costo in accessi a memoria/cicli di clock dei quattro stati

Puoi osservare nella tabella 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 possiamo ottimizzare ulteriormente.

È possibile minimizzare la memoria scegliendo quale operazione fra decode e write-Back avrà il diritto di accedervi. 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.

Possiamo andare oltre. Introduciamo quindi la strategia di **pipelining**; la quale 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'altra¹⁰, riducendo drasticamente i cicli di clock necessari per elaborare un programma. Generalmente, segue questi passi:

1. **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. **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 del PSW.
3. **Execution** riceverà gli operandi e ne calcolerà il risultato. Aggiungerà infine i registri e metterà il tutto nel suo banco insieme al PSW aggiornato.
4. **WriteBack** infine salverà questi risultati nella cache dati o nei registri, dipendentemente da dove è richiesto.

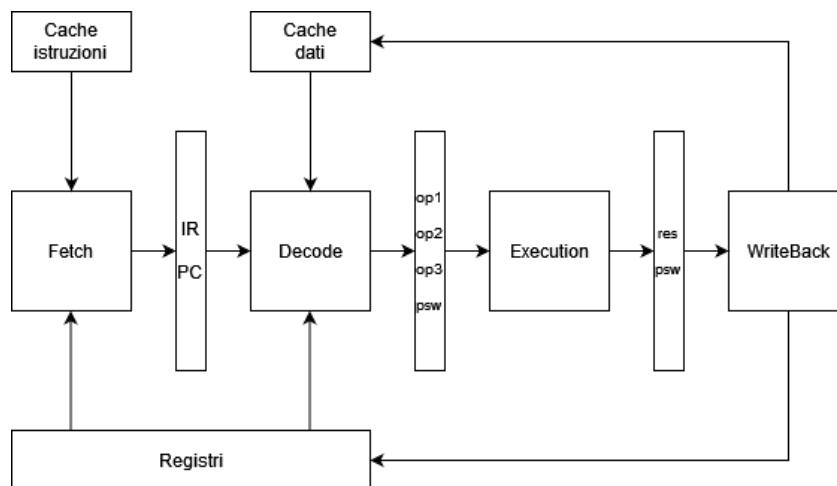


Figure 4.18: Circuito di pipeline

¹⁰Nella maggior parte dei casi è così. L'eccezione è quando si hanno cicli e condizioni.

Nella teoria si svolge in questo modo, tuttavia passando al lato pratico si presentano un paio di problemi. Il primo è dato da un'eventuale complessità degli stati ed un conseguente sfalsarsi delle operazioni col clock. Un secondo problema è dato dalle **bolle**, ovvero istruzioni che non possono essere eseguite nello stesso ciclo di clock a causa di una dipendenza condizionale o ciclica.

Osserva bene la tabella seguente 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 preparato da WriteBack, quindi sarà necessario aspettare un ciclo di clock per effettuare poi il decoding. Da questo puoi notare come un registro ritorni pronto solo dopo l'esecuzione della WriteBack.

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**¹¹.

In tal 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 $\frac{12-4}{6} = 1,35CPI$, che si avvicina molto al nostro obiettivo.

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

Figure 4.19: Istruzioni pipeline

4.12 Modello CISC e RISC

Parliamo adesso di com'è nato il modello di architettura sul quale stai leggendo questo PDF: il **RISC**, Reduced Instruction Set Computer; figlio del connubio di tutto ciò che abbiamo studiato finora. Si tratta dell'architettura con l'attuale migliore ottimizzazione del tempo impiegato dalla CPU¹² per elaborare le informazioni.

¹¹La 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.

¹²Più precisamente, vale l'equazione: $T_{CPU} = Tot_{Istruzioni} \times CPI_{Medio} \times \frac{1}{Freq_{Clock}}$.

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, dunque, 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 molto più impegnativo, per non parlare della pipeline che deve costantemente aspettare il lavoro finito dello stato precedente.

Qui entrarono in gioco due professori dell'Università di Stanford; 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 faticosa 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, lo erano anche le istruzioni in codice Assembly.

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**, Complex Instruction Set Computer. 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.



Figure 4.20: John Hennessy e David Patterson

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.

4.13 Architetture parallele

È nostro volere ottimizzare quanto possibile, 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 utilizza una componente di pre-fetch che invia 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

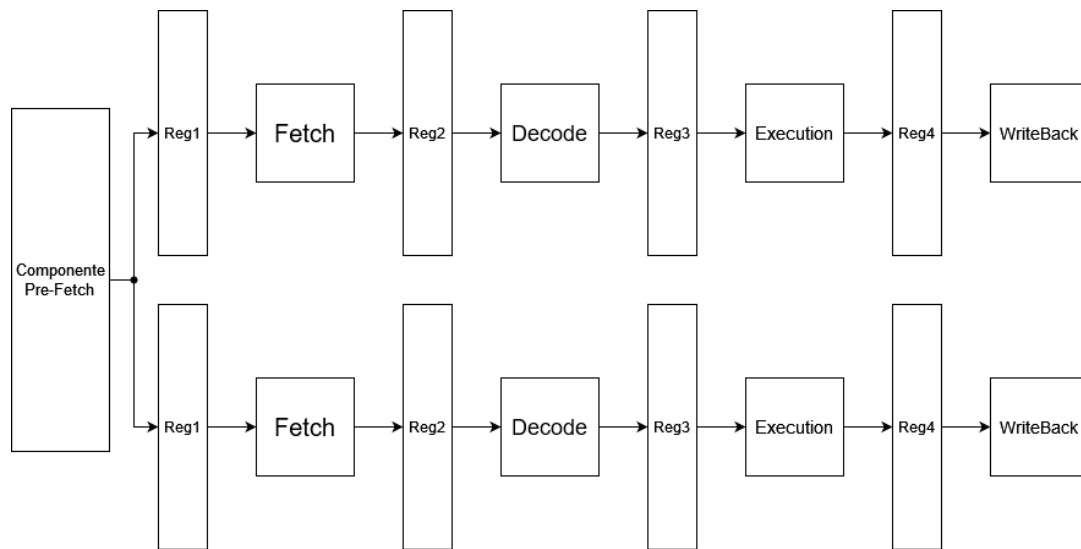


Figure 4.21: Funzionamento dual-pipeline di PowerPC

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, allora?

Ebbene, questo è a causa della dipendenza delle istruzioni. Ulteriori pipelines equivalgono a molte 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. Come risolvere?

Un primo tentativo di risoluzione si ebbe con le architetture di microprocessori **VLIW**, Very Long Instruction Word, 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, si utilizzerebbero 1000 cicli di clock, ma inserendo 500 circuiti di somma, ne verranno usati solamente due; un miglioramento spaventoso. Tuttavia, per collegare le mini-CPU è necessario utilizzare una **matrice di interconnessione**, che causa una sovrabbondanza di collegamenti.

Un altro problema è presentato dalla **Legge di Amdahl**, la quale afferma che *"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 introduciamo 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*¹³ 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 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 informazioni¹⁴.

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 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, Graphic Processing Unit, ed il capo supremo del loro mercato è NVidia.

Col passare del tempo questi modelli si svilupparono e videro la luce le **GPGPU**, General Purpose GPU, 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.

¹³ $L = \frac{T}{W}$, dove T è il tempo e W il totale del lavoro eseguito per la task.

¹⁴Se non ti sembra aver senso invito la revisione della sezione dedicata all'arbitraggio del BUS.

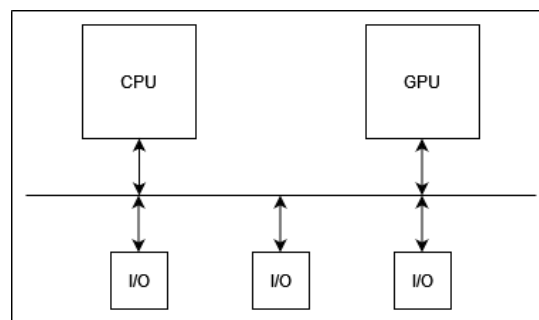


Figure 4.22: Architettura di Von Neumann contemporanea

Chapter 5

SIS e Verilog

5.1 Introduzione a SIS

5.2 Sintesi combinatoria esatta

5.3 Sintesi combinatoria approssimata multilivello

5.4 Modellazione di FSM

5.5 Modellazione di FSMD

5.6 Introduzione a Verilog

5.7 Modellazione in Verilog

5.8 Modellazione di FSM

5.9 Modellazione di FSMD

5 - comprende: Sis - Subsequential interactive synthesis Linguaggi HDL - Verilog

Chapter 6

Il linguaggio Assembly

6.1 Introduzione ad Assembly

Assembly è un linguaggio a basso livello di astrazione che permette di lavorare a stretto contatto con le componenti hardware. Consente l'accesso ai registri della CPU e, di conseguenza, scrivere codice estremamente ottimizzato. È comune a tutti gli elaboratori, seppur presentando alcune differenze dialettali.

Nel nostro caso, lavoreremo con **Assembly Intel x86** con sintassi **AT&T**. Tale linguaggio lavora con i seguenti tipi di registro:

1. Registri generici

- **AX**: Accumulation register, accumulatore per operazioni aritmetiche, contenente il risultato dell'operazione.
- **BX**: Base register, usato per operazioni di indirizzamento della memoria.
- **CX**: Counter register, usato per contare, di norma nei costrutti ciclici.
- **DX**: Data register, usato nelle operazioni di input/output, nelle divisioni e nelle moltiplicazioni.

2. Registri di segmento

- **CS**: Code segment, punta alla zona di memoria che contiene il codice.
- **DS**: Data segment, punta alla zona di memoria che contiene i dati.
- **ES**: Extra segment, utilizzabile come registro di segmento ausiliario.
- **SS**: Stack segment, punta alla zona di memoria in cui risiede la stack.

3. Registri puntatore

- **SP**: Stack pointer, punta in cima alla stack ed è modificato dalle operazioni push/pop.
- **BP**: Base pointer, punta alla base della porzione di stack attualmente gestita.
- **IP**: Instruction pointer, punta alla prossima istruzione.

4. Registri indice

- **SI**: Source index, punta alla stringa/vettore sorgente.
- **DI**: Destination index, punta alla stringa/vettore destinazione.
- **FLAGS**: Utilizzato per memorizzare lo stato corrente del processore, ogni bit fornisce una particolare informazione.

L'ISA a 32b consente di estendere tutti i registri sovramenzionati aggiungendo una E al loro nome. Quindi $AX = 16b$, mentre $EAX = 32b$. In tal merito, segue la divisione dello spazio dei vari registri:

Parliamo ora delle **modalità di indirizzamento**, ovvero il modo in cui sono specificate le istruzioni da eseguire. Ci sono sette modi diversi:

- **A registro** [%eax]: Operando contenuto in un registro e il suo nome è specificato nell'istruzione.
- **Diretto** [(IND)]: Operando contenuto in una locazione di memoria e l'indirizzo della locazione viene specificato nell'istruzione.
- **Immediato** [\$VAL]: Operando è valore costante ed è definito esplicitamente nell'istruzione.
- **Indiretto** [(%eax) oppure (\$VAL)]: L'indirizzo di un operando è contenuto in un registro o in una locazione di memoria. L'indirizzo della locazione o il registro viene specificato nell'istruzione.
- **Indicizzato** [SPI(%eax)]: L'indirizzo effettivo dell'operando è calcolato sommando un valore costante al contenuto di un registro.
- **Con autoincremento**: L'indirizzo effettivo dell'operando è il contenuto di un registro specificato nell'istruzione. Dopo l'accesso, il contenuto del registro viene incrementato per puntare all'elemento successivo.
- **Con autodecremento**: Il contenuto di un registro specificato nell'istruzione viene decrementato. Il nuovo contenuto viene usato come indirizzo effettivo dell'operando.

6.2 Istruzioni e Sintassi

In Assembly, le istruzioni seguono la sintassi generale [istruzione OP1, OP2, ...] e costituiscono il nostro spazio di manovra con il quale potremo scrivere i programmi. Per cominciare, un file Assembly ha estensione `.s` o `.asm` ed è suddiviso in varie sezioni. Segue esempio di `hello world`:

```
.section .data # Sezione di variabili globali e costanti.
# Definizione della stringa "hello" e relativa lunghezza "helloLen"
hello: .ascii "Hello World!"
helloLen: .long .- hello

.section .bss # Sezione per variabili non inizializzate.

.section .text # Sezione esecutiva
.global _start # Fornisce la zona di memoria dove inizia il programma

_start:
    movl $4, %eax # Direttiva SYS_WRITE
    movl $1, %ebx # File descriptor 1; STDOUT
    leal hello, %ecx # Indirizzo di hello
    movl helloLen, %edx # Lunghezza di hello
    int $0x80 # Interruzione del kernel

    movl $1, %eax # Direttiva SYS_EXIT
    xorl %ebx, %ebx # Azzera il registro %ebx (richiesto da direttiva)
    int $0x80
```

Noterai che è un linguaggio particolarmente verboso e tedioso, tuttavia questi sono i processi che il pc esegue a livello dei registri per lo svolgimento delle funzioni più astratte.

Scritto il programma, bisogna assemblarlo, linkarlo e poi eseguirlo; ciò si fa attraverso i seguenti comandi a terminale:

```
as --32 nomeFile.s -o nomeFile.o
ld -m elf_i386 nomeFile.o -o nomeEseg
./nomeEseg
```

Una caratteristica utile da ricordare del linguaggio è che si possono stampare a video solamente caratteri `ascii`, quindi stringhe. Il passaggio da singolo intero ad `ascii` **ITOA** si ha con $[n + 48]$, dove 48 è la codifica `ascii` del numero 0. Di conseguenza per passare da `ascii` a intero **ATOI** si fa l'operazione inversa: $[n - 48]$.

Se invece dobbiamo lavorare con stringhe, sarà necessario concatenare i caratteri mediante moltiplicazioni per 10.

Assembly non ha propriamente istruzioni come `while` e `for`, quindi bisognerà costruire costrutti ciclici da zero. Ciò si fa con le istruzioni di comparazione e salto.

```
.section .data
.section .text
.global _start

_start:
    cmpl %eax, %ebx # Compara i valori di eax ed ebx

    jg _jumped      # Salta a _jumped se eax > ebx
    jl _jumped      # Salta se eax < ebx
    jge _jumped     # Salta se eax >= ebx
    jle _jumped     # Salta se eax <= ebx
    je _jumped      # Salta se eax == ebx
    jne _jumped     # Salta se eax != ebx
    jcxz _jumped    # Salta se cx == 0
    jmp _jumped     # Salto senza condizioni
    loop _jumped    # Salta e ecx--

    test %ebx, %ebx # Esegue AND bit a bit fra gli operandi

_jumped:
    movl $1, %eax
    xorl %ebx, %ebx
    int $0x80
```

Si lascia come esercizio la scrittura di costrutti condizionali e ciclici base come `if-else`, `while` e `for`.

6.3 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 aggiungendo il flag di debug: **-gstabs**, dopodiché si potrà eseguire il programma sotto debugger.

```
# 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
> break numeroRiga

# Passa alla prossima istruzione
> next
> nexti

# Va avanti fino al prossimo breakpoint
> continue

# Refresha la visuale. L'interfaccia si bugga spesso.
> ref
```

Seguono 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 stampa 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 programma con i flag di debug peggiora le prestazioni totali; inoltre comporta problemi di sicurezza perché consentirebbe di leggere il codice sorgente tramite debugger.

6.4 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 della dinamica governante la stack. Solo l'ultimo elemento inserito potrà essere rimosso.
- **FIFO**; First In, First Out, solo il primo elemento inserito potrà essere rimosso.

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

```
pushl %eax # Mette il valore contenuto in eax nella stack.
popl %eax  # Preleva l'ultimo valore inserito in stack e lo inserisce in
            eax.
```

Ogni volta in cui si pusha qualche valore sulla stack, 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, infatti, conta come una stringa ed è 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 tante volte quante necessarie 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 ora 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
```

```
.type nomeFunzione, @function # Dichiarazione della funzione

# ...
# Blocco di istruzioni
# ...

# Ritorno al file chiamante la funzione
ret
```

La direttiva **ret** farà ritorno alla funzione chiamante. È possibile manipolare la zona in cui rientra attraverso il registro esi. Sconsiglio questa azione, tuttavia.

6.5 Assembly e C

```
#include<stdio.h>

int main() {
    return 0;
}
```
