



UniCT – DMI
Architettura dei Calcolatori
ANNO ACCADEMICO
2019/20

Autori:
Alessio Tudisco

Sommario

Introduzione	9
Com'è fatto un calcolatore elettronico?	9
Come sono rappresentate le istruzioni e i dati in un calcolatore elettronico?	10
Come vengono eseguite le istruzioni?	10
Come vengono ricevuti/inviati i dati da/a periferiche durante l'esecuzione di un programma? ..	10
I sistemi di numerazione e la rappresentazione binaria.....	11
Cos'è un sistema di numerazione e tipologie	11
Definizione di un sistema di numerazione posizionale	11
La conversione di un numero in base 10 in un'altra base	11
I numeri interi relativi nel sistema di numerazione binaria	11
L'estensione e riduzione del segno.....	12
La somma binaria naturale	12
La somma binaria con il complemento a due.....	12
Il Trabocco (overflow).....	13
Il Trabocco nella somma binaria naturale	13
Il Trabocco nella somma a complemento a due.....	13
I numeri frazionari e la rappresentazione binaria (fixed point e floating point, convenzioni IEEE e conversione dec->bin della parte decimale)	13
Rappresentazione a virgola fissa (fixed point).....	13
Rappresentazione a virgola mobile (floating point)	14
I caratteri e la rappresentazione binaria.....	15
Il controllo d'integrità e la rappresentazione binaria	16
Cause di errori nelle informazioni codificate.....	16
Bit di parità	16
Byte di controllo.....	16
Checksum (O somma di controllo)	17
Distanza di Hamming.....	17
I formati multimediali e la rappresentazione binaria	19
Tipologie di formati multimediali	19
Campionamento e Quantizzazione.....	19
La compressione dei dati (fasi, categorie, rapporto di compressione).....	19
Compressioni famose	20
Algebra Booleana della commutazione	20
Cos'è l'algebra booleana.....	20
Le funzioni logiche fondamentali (NOT, AND, OR)	20
La somma logica (OR)	20
Il prodotto logico (AND).....	20

Il complemento (NOT)	21
Alcune proprietà condivise dal OR e dal AND	21
I teoremi di De Morgan	21
Le espressioni logiche	21
La precedenza degli operatori nelle espressioni logiche	22
Come ottenere la funzione logica dall'espressione logica.....	22
Come ottenere un'espressione logica a partire dalla tabella di verità di una funzione (mintermini-prima_canonica, maxtermini-seconda_canonica).....	22
Il mintermine e la prima forma canonica	22
Il maxtermine e la seconda forma canonica.....	23
La forma minima delle espressioni logiche	23
Minimizzare la prima forma canonica	23
Ottenere un'espressione minima dalla tabella di verità con le mappe di Karnaugh.....	24
Condizione di indifferenza nelle mappe di Karnaugh	24
<i>Circuiti e porte logiche</i>	<i>26</i>
Le porte logiche.....	26
Le reti combinatorie	26
Le porte logiche AND, OR, NOT	26
Le porte AND e OR a più ingressi.....	26
L'equivalenza fra le espressioni logiche e le reti combinatorie	27
La differenza simmetrica (XOR o OR esclusivo)	27
Le porte NAND e NOR.....	27
NAND e NOR, le porte universali.....	28
Trasformare una SOP in una rete NAND	28
Realizzare la porta NOT con le porte NAND e NOR.....	28
Realizzare la porta AND con le porte NAND e NOR	28
Realizzare la porta OR con le porte NAND e NOR.....	28
Realizzare porte NAND a più ingressi	28
<i>I Circuiti elettrici</i>	<i>29</i>
Rappresentazione di variabili binarie in un calcolatore	29
I transistor MOS	29
I transistor NMOS.....	30
I transistor PMOS	30
Il circuito NOT con NMOS	30
Il circuito NOR con NMOS	30
Il circuito NAND con NMOS.....	31
I transistor CMOS	31

Il circuito NOT fatto con CMOS	31
Il circuito NOR con CMOS.....	32
Il circuito NAND con CMOS	32
Il circuito AND/OR con CMOS	32
Ritardi in un circuito	32
La porta tri-state	33
I circuiti integrati	33
Il decodificatore (decoder).....	33
Il multiplatore (multiplexer)	34
L'addizionatore completo (full adder)	34
L'addizionatore a propagazione del riporto (ripple carry adder)	35
L'addizionatore e il trabocco.....	35
L'addizionatore algebrico a n bit	35
ALU a 1 bit	36
ALU a n bit	36
Le reti sequenziali	37
Bistabile asincrono	37
Bistabile sincrono	37
Bistabile di tipo D.....	38
L'effetto trasparenza	38
Flip-flop master-slave (o tipo D).....	38
Flip-flop tipo T.....	38
Flip-flop tipo JK	39
Flip-flop con preset e clear.....	39
I Registri.....	40
I Registri paralleli	40
I Registri a scorrimento.....	40
I Registri seriale-parallelo	41
I Registri universali	41
Il Contatore	41
Istruzioni macchina	42
Come lavora un calcolatore?.....	42
Organizzazione della memoria (indirizzamento e ordinamento byte)	42
Instruction set architecture ISA.....	42
Tipologie di ISA (RISC e CISC)	43
Istruzioni base per l'accesso alla memoria.....	43
Istruzioni avanzate per l'accesso alla memoria	43

Istruzioni aritmetiche base (somma e sottrazione)	44
Istruzioni aritmetiche avanzate (moltiplicazione e divisione)	44
Istruzioni logiche base (AND e OR)	45
Istruzioni di scorrimento logico (logical shift)	45
Istruzioni di scorrimento aritmetico (arithmetical shift)	45
Istruzioni di rotazione dei bit	45
Istruzioni di rotazione dei bit con riporto	46
Istruzione base per il salto condizionale	46
Istruzione base per il salto non condizionale	46
Istruzione MOV (RISC e CISC)	46
Varianti delle istruzioni aritmetiche e logiche in CISC.....	46
Bit di esito o condizione.....	47
Cos'è un modo di indirizzamento?	47
Modo di indirizzamento immediato	47
Modo di indirizzamento di registro	47
Modo di indirizzamento assoluto o diretto.....	48
Modo di indirizzamento indiretto da registro	48
Modo di indirizzamento con indice e spiazzamento	48
Modo di indirizzamento con base e indice.....	48
Modo di indirizzamento con auto-incremento (CISC)	48
Modo di indirizzamento con auto-decremento (CISC)	49
Modo di indirizzamento relativo su PC (CISC)	49
Direttive di assemblatore	49
Direttiva di eguaglianza (EQU)	49
Direttiva di origine (ORIGIN)	49
Direttiva di riserva di memoria (RESERVE)	50
Direttiva di riserva di memoria con inizializzazione (DATAWORD).....	50
Struttura di una linea di codice assembletivo	50
Notazione dei numeri	50
La pila (Stack) e le operazioni.....	51
Cos'è il sottoprogramma e le istruzioni di salto speciali	51
Passare dei parametri ai sottoprogrammi e restituzione di un risultato.....	52
Area di attivazione in pila (stack frame)	53
Annidamento dei sottoprogrammi.....	53
Codifica numerica delle istruzioni	53
Codifica: Formato con operandi in registri.....	54

Codifica: Formato con operando immediato	54
Codifica: Formato per chiamata	54
Operazioni I/O	54
Interfacce di dispositivo.....	54
Spazio di indirizzamento di I/O	55
L'interfaccia della tastiera.....	55
L'interfaccia dello schermo	55
I/O controllati da programma	56
Controllare lo stato di un singolo bit e istruzione TestBit (CISC).....	56
Confronto del contenuto di due registri	56
Tecnica di interruzione	57
Meccanismo dell'interruzione e annidamenti.....	57
Gestione dispositivi multipli	58
Tecnica: Polling	58
Tecnica: Interruzione vettorizzata	58
Registri di controllo del processore	58
Accesso ai registri di controllo del processore	59
Eccezioni	59
Il livello software	59
Come si programma?.....	59
Come viene generato un programma oggetto finale	60
Il processo assemblativo.....	60
Assemblaggio a due passi	60
Il loader	61
Il linker	61
Le librerie.....	61
Il compilatore.....	61
Il debugger.....	61
Il sistema operativo.....	62
Struttura base del processore.....	63
Struttura in dettaglio del processore.....	63
Il banco di memoria:.....	63
L'unità aritmetico-logica (ALU).....	63
Generatore di indirizzi delle istruzioni	64
Fasi dell'esecuzione di un'istruzione	64
Organizzazione a 5 stadi dell'istruzioni RISC	65

Datapath (Percorso dei dati)	65
Alcune datapath	66
Attesa di memoria (MFC)	67
Segnali di controllo	67
Controllo Cablato	67
Organizzazione di un processore CISC	68
Interconnessione via BUS	68
Il datapath CISC	69
Controllo microprogrammato	69
Pipelining	69
La tecnica di pipeline	69
I problemi del pipelining	70
Conflitto: Dipendenze di dato e soluzioni	70
Conflitto: Ritardi della memoria e soluzione	71
Conflitto: Ritardo nei salti e soluzioni	71
Conflitto: Limite di risorse e soluzione	72
Valutazione della prestazione ed effetti dei conflitti	72
Processori superscalari	73
Dipendenze di dato e le stazioni di prenotazione	73
I ritardi e il buffer di riordino	73
Funzionamento dei dispositivi I/O	74
Il bus di sistema	74
Bus sincrono	74
Bus asincrono	74
Bus asincrono	Errore. Il segnalibro non è definito.
La struttura delle interfacce I/O	75
Le porte parallele	75
Le porte seriali	75
Standard di trasmissione	76
Lo standard USB	76
Firewire	77
Bus PCI e PCIe	77
Bus SCSI e SATA	78
Memoria	78
Gerarchia della memoria	78
Il ruolo delle operazioni di memoria nella gerarchia della memoria	78

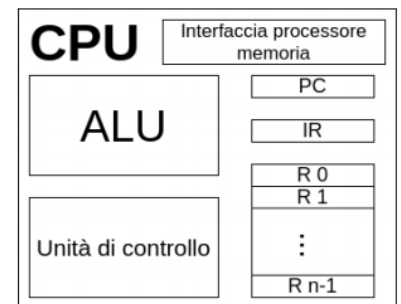
Memoria cache (Cache hit e cache miss)	79
Schemi di indirizzamento.....	80
Schema di indirizzamento diretto	80
Schema di indirizzamento associativo	80
Schema di indirizzamento associativo a gruppi	80
Gli algoritmi di sostituzione	81
Memoria virtuale	81
MMU e Traduzione degli indirizzi.....	81
La cache dell'MMU.....	82
Direct Memory Access	82
Miglioramenti sul controllo della cache.....	83
Memoria ad accesso casuale (RAM)	83
RAM statica (SRAM)	83
RAM dinamica (DRAM).....	84
Memoria a sola lettura (ROM) e varianti (PROM, EPROM, EEPROM).....	85
Memoria flash.....	85
Dischi magnetici	86
Calcoli aritmetici.....	86
Differenza fra Addizionatore con riporto e anticipo del riporto	86
Moltiplicazione di numeri interi binari senza segno	86
Moltiplicazione di numeri interi binari con segno (Soluzione diretta, algoritmo di booth e bit pair).....	87
Moltiplicazione veloce.....	87
Divisione di numeri interi binari (Divisione con e senza ripristino)	87
Somma e sottrazione di numeri frazionari binari	88
Moltiplicazione di numeri frazionari binari.....	88
Divisione di numeri frazionari binari	88

Introduzione

Com'è fatto un calcolatore elettronico?

Un calcolatore elettronico è costituito da un insieme di componenti che cooperano per lo stesso fine:

- **Processore (CPU):** è un circuito elettronico integrato (chip) che agisce da *cervello* del calcolatore, carica ed esegue le *istruzioni elementari* necessarie ad eseguire i *programmi*. Un programma è una sequenza di istruzioni elementari fornite dall'utente per uno specifico scopo. Il processore è a sua volta composta da:
 - **L'unità aritmetica-logica (ALU):** esegue le operazioni aritmetiche e logiche (quali addizione, sottrazione, divisione, moltiplicazione e confronto fra numeri) necessarie ad eseguire le istruzioni. Gli operandi e i risultati vengono inseriti in un elemento di memoria apposito detto *registro*.
 - **Registri:** sono blocchi di memoria interni alla CPU caratterizzati da una velocità di scrittura e lettura elevatissima e dalla dimensione pari ad una *word* (spesso 16bit). Contengono le istruzioni in esecuzione, gli indirizzi e i dati. Si suddividono in *special-purpose* e *general-purpose*.
 - **L'interfaccia processore-memoria:** gestisce il trasferimento dei dati fra la memoria e il processore.
 - **L'unità di controllo:** coordina il funzionamento delle varie componenti: caricamento dei dati dalla memoria e gestione dei salti d'esecuzione;
- **Memoria:** le unità di memoria sono utilizzate per immagazzinare le informazioni necessarie ad eseguire i programmi. Le informazioni che contengono possono essere costituite da istruzioni, eseguite dal processore, o dati, numero o caratteri usati come operandi dalle istruzioni. La memoria si suddivide in due categorie:
 - La memoria centrale o primaria: è una memoria ad accesso veloce denominata RAM (Random Access Memory) che garantisce una velocità di accesso costante per ogni indirizzo. Ha una capacità limitata ed è volatile, ovvero una volta interrotta l'alimentazione il suo contenuto viene perso. Fisicamente sono costituite da celle di transistor a semiconduttori, dalla capacità di un bit. Le varie operazioni non interagiscono con le singole celle ma con gruppi di celle di dimensione fissa, dette *memory word*. La memoria centrale è organizzata su dei livelli, più è basso il livello più sono veloci e meno capienti mentre più è alto il livello meno sono veloci e più capienti;
 - La *memoria cache* costituisce il livello più basso, sono memorie velocissime e spesso integrate all'interno del processore. Man mano che vengono caricate informazioni dalla memoria primaria, una copia viene salvata nella cache al fine di velocizzare un successivo richiamo.
 - La memoria di massa o secondaria: è una memoria lenta caratterizzata da una capacità elevata e dalla persistenza dei dati, non sono volatili. Un esempio di memoria di massa sono i dischi magnetici, i dischi ottici e le memorie flash.
- **Le interfacce di Input/Output:** un calcolatore ha necessità di comunicare con il mondo esterno, per esempio può ricevere un *input* direttamente dall'utente attraverso una



tastiera o mouse o può fornire un *output* all'utente attraverso un *monitor* o una *stampante*.

- Il bus di sistema: è un insieme di collegamenti elettrici che *interconnette* le varie componenti del calcolatore. Ogni collegamento ha un ruolo ben preciso e trasporta uno specifico tipo di informazione.

Come sono rappresentate le istruzioni e i dati in un calcolatore elettronico?

I calcolatori sono costituiti da circuiti elettrici. Progettare e realizzare circuiti elettrici a due stati di tipo *ON/OFF* è molto più semplice ed immediato rispetto a dover gestire diversi livelli di tensione. I concetti *ON/OFF* possono essere rappresentati attraverso numeri:

- $OFF = 0$
- $ON = 1$

Il sistema di numerazione binaria è la soluzione perfetta per rappresentare i valori *ON/OFF*, ed è per questo motivo che le varie informazioni all'interno del calcolatore sono rappresentate sotto forma di *sequenze binarie*.

Come vengono eseguite le istruzioni?

L'esecuzione delle istruzioni è caratterizzata da 3 passi elementari:

- *Prelievo*: Mediante l'interfaccia processore-memoria della CPU viene prelevata dalla memoria primaria l'istruzione puntata dal registro specializzato PC (*Program Counter*, tale registro può puntare alla prima istruzione del programma o a quella successiva rispetto a quella corrente) e inserita all'interno del registro specializzato IR (*instruction register*).
- *Decodifica*: Il contenuto del registro IR viene analizzato dall'unità di controllo della CPU per identificare quale operazione bisogna eseguire e dove si trovino i dati necessari.
- *Esecuzione*: l'istruzione viene eseguita all'interno dell'ALU, l'output viene salvato in memoria e il registro PC viene aggiornato per puntare all'istruzione successiva.

Come vengono ricevuti/inviati i dati da/a periferiche durante l'esecuzione di un programma?

Durante l'esecuzione di un programma potrebbe esserci un dispositivo di input o output che necessita dell'esecuzione di un servizio urgente. È quindi possibile interrompere l'esecuzione del programma corrente per poter eseguire rapidamente il servizio desiderato.

Il meccanismo di *interruzioni* permette di operare come sopra descritto:

- Il dispositivo I/O invia al processore il segnale di richiesta di interruzione;
- Il processore interrompe l'esecuzione del programma corrente e salva in memoria lo stato attuale, generalmente viene salvato il contenuto del registro PC e quello dei registri general purpose.
- Il processore esegue la routine di servizio dell'interruzione, la routine è un programma a parte specifico per il servizio richiesto dal dispositivo I/O;
- Al termine dell'esecuzione della routine, viene caricato lo stato precedentemente salvato in memoria per poter riprendere l'esecuzione del programma interrotto;

I sistemi di numerazione e la rappresentazione binaria

Cos'è un sistema di numerazione e tipologie

In un sistema di numerazione si hanno definiti simboli ognuno rappresentante una quantità precisa.

Un sistema di posizionale può essere:

- Posizionale: quando la quantità rappresentata dal simbolo dipende anche dalla sua posizione nella cifra. Per esempio nel sistema decimale si hanno le unità, le decine, le centinaia e così via.
 - *Ciò rende il sistema molto flessibile ed economico in quanto pur avendo un numero limitato di simboli può rappresentare qualunque quantità.*
- Additivo: quando il numero rappresentato è dato dalla semplice somma delle quantità dei simboli che lo costituiscono.
 - *Ciò rende il sistema meno flessibile e dispendioso in quanto al crescere delle quantità da rappresentare si hanno sempre più simboli.*

Definizione di un sistema di numerazione posizionale

Un sistema di numerazione posizionale è definito da:

- Un numero intero **B** detto *base*;
- Un insieme di *B* simboli $S_B = \{S_0, \dots, S_{B-1}\}$, ognuno dei quali rappresenta le quantità $\{0, 1, \dots, B - 1\}$

Un numero a *n* cifre $p_{n-1}p_{n-2} \dots p_0$ con $p_i \in S_B$ e $i = 0 \dots n - 1$ può essere rappresentato come la somma di potenze della base. È quindi possibile rappresentare il range di valori $[0, B^n - 1)$

La conversione di un numero in base 10 in un'altra base

La *conversione* di un numero da una base 10 a una base *X* è attuabile attraverso la tecnica delle *divisioni successive*:

1. Sia *N* il numero in base 10 da convertire;
2. Si calcola la divisione intera $N = \frac{N}{B}$ e si mette da parte il *resto* *R*;
3. Se $N > 0$, si riesegue il punto 2;
4. Se $N = 0$, si riportano i vari resti nel *senso opposto* con cui si sono ottenuti: dal più recente al meno recente, da destra verso sinistra, ect.
5. *La sequenza dei resti così ottenuta rappresenta il numero in base X;*

I numeri interi relativi nel sistema di numerazione binaria

Per rappresentare un numero intero relativo in binario vi sono principalmente 3 tecniche:

- *Segno e valore assoluto*: si utilizza il bit più a sinistra, denominato *bit di segno*, per indicare il segno del numero:
 - *POSITIVO* = 0;
 - *NEGATIVO* = 1;
- *Complemento a 1*: in cui ogni bit viene invertito. Si esegue lo stesso procedimento per ottenere un numero positivo partendo da un numero negativo.
- *Complemento a 2*: in cui si esegue il complemento a 1 del numero e successivamente si somma 1;

Tali tecniche pur riuscendo a rappresentare un numero negativo con lo stesso numero di bit si differenziano per alcuni aspetti:

- Sia “segno e valore assoluto” e “complemento a 1” presentano una doppia rappresentazione dello 0, ciò comporta un range di valori limitato: $(-2^{n-1}, \dots 2^{n-1})$ (valori estremi esclusi);
- Il “complemento a 2” permette un range di valori $[-2^{n-1}, \dots 2^{n-1})$ (valore estremo inferiore incluso) ed è quello utilizzato principalmente nei calcolatori per via della sua efficacia nelle operazioni di addizione e sottrazione;
-

L'estensione e riduzione del segno

Spesso può esserci la necessità di aumentare o diminuire il numero di bit usati per codificare un numero. Si può affermare che un numero non muta replicandogli o rimuovendogli il bit di segno purché esso non si inverta.

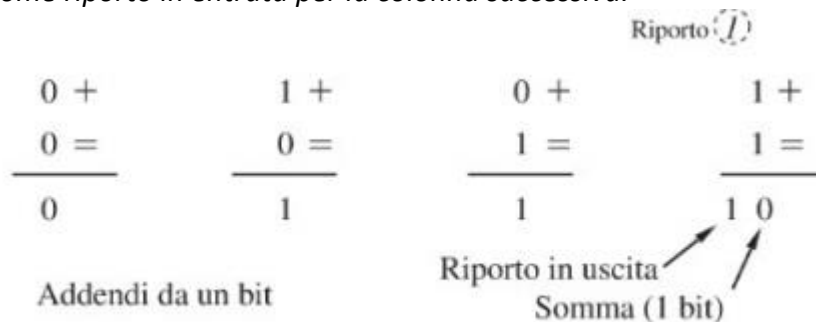
L'operazione di incremento del numero di bit prende il nome di *estensione del segno* mentre la riduzione del numero dei bit prende il nome di *riduzione del segno*:

- *L'estensione del segno*: si aggiungono gli n bit aggiuntivi a sinistra equivalenti al bit di segno del numero. $0101 \Rightarrow 0000\ 0101$ oppure $1011 \Rightarrow 1111\ 1011$
- *La riduzione del segno*: si rimuovono gli n bit in eccesso a sinistra purché il nuovo bit più a sinistra risulti equivalente al bit di segno. $1111\ 1011 \Rightarrow 1011$

La somma binaria naturale

La somma binaria naturale consiste nella normale somma in colonna del sistema decimale, con l'accortezza di utilizzare la base 2.

La somma di due bit pari a 1 equivale a 0 con un riporti in uscita pari a 1, che può essere usato come riporto in entrata per la colonna successiva.



La somma binaria con il complemento a due

La somma di numeri binari a complemento a due è simile alla somma binaria naturale con sola differenza che viene trascurato il *riporto in uscita finale*.

Si ha quindi che:

- L'addizione fra due numeri consiste nella somma binaria naturale, il riporto in uscita finale viene ignorato.
- La sottrazione fra due numeri consiste nell'addizione fra il minuendo e il complemento a due del sottraendo.

(a)	$\begin{array}{r} 0010 + (+2) \\ 0011 = (+3) \\ \hline 0101 \quad (+5) \end{array}$	(b)	$\begin{array}{r} 0100 + (+4) \\ 1010 = (-6) \\ \hline 1110 \quad (-2) \end{array}$
(c)	$\begin{array}{r} 1011 + (-5) \\ 1110 = (-2) \\ \hline 1001 \quad (-7) \end{array}$	(d)	$\begin{array}{r} 0111 + (+7) \\ 1101 = (-3) \\ \hline 0100 \quad (+4) \end{array}$
(e)	$\begin{array}{r} 1101 - (-3) \\ 1001 = (-7) \end{array} \Rightarrow \begin{array}{r} 1101 + \\ 0111 = \\ \hline 0100 \quad (+4) \end{array}$		
(f)	$\begin{array}{r} 0010 - (+2) \\ 0100 = (+4) \end{array} \Rightarrow \begin{array}{r} 0010 + \\ 1100 = \\ \hline 1110 \quad (-2) \end{array}$		

Il Trabocco (overflow)

Si ha un trabocco quando

l'operazione aritmetica produce un risultato fuori dall'intervallo disponibile.

Il Trabocco nella somma binaria naturale

Il trabocco nella somma binaria naturale lo si ha se e solo se il riporto di uscita finale vale 1.

Esempio: in una codifica a 4 bit:

$$(9)_{10} 1001 + (11)_{10} 1010 \Rightarrow (3?)_{10} [1] 0011 \Rightarrow \text{Si è verificato il trabocco}$$

Il Trabocco nella somma a complemento a due

Il trabocco nella somma a complemento a due lo si ha se e solo se:

- Gli addendi sono concordi in segno;
- Il bit di segno del risultato è diverso da quello degli addendi;

Esempio: in una codifica a 4 bit:

$$(-7)_{10} 1001 + (-4)_{10} 1100 \Rightarrow (5?)_{10} 0101 \Rightarrow \text{Si è verificato il trabocco}$$

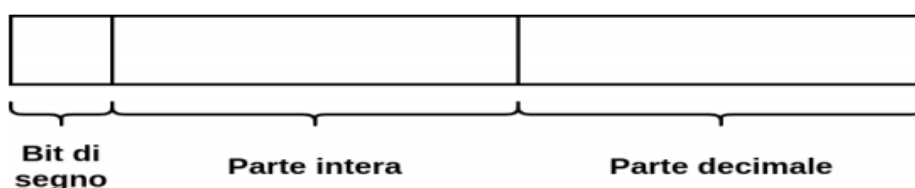
I numeri frazionari e la rappresentazione binaria (fixed point e floating point, convenzioni IEEE e conversione dec->bin della parte decimale)

Per rappresentare i numeri frazionari in binario vi sono principalmente due metodologie: *fixed point* e *floating point*.

Rappresentazione a virgola fissa (fixed point)

L'idea alla base di questa rappresentazione consiste nel posizionare in modo arbitrario la virgola in una posizione da noi scelta al fine di ottenere la seguente struttura:

- Un bit di segno;
- Una porzione di bit fissa che rappresenta la parte intera;
- Una porzione di bit fissa che rappresenta la parte decimale;



Questa rappresentazione non è efficiente in quanto la posizione della virgola influenza:

- L'intervallo di numeri rappresentabili;
- La sensibilità di variazione, ovvero la minima parte rappresentabile;

A riguardo vi sono due casi estremi:

- Un numero a n bit composto da solo il bit di segno e la parte intera:

$[bit\ di\ segno]0 [parte\ intera]1001(.) [parte\ decimale\ assente]$

Tale rappresentazione consente un range $[-2^{n-1} \dots 2^{n-1} - 1]$ con una sensibilità 1.

- Un numero a n bit composto da solo il bit di segno e la parte decimale:

$[bit\ di\ segno]0.1001 [parte\ decimale]$

Tale rappresentazione consente un range $[-1 \dots 1 - 2^{-(n-1)}]$ con una sensibilità $2^{-(n-1)}$

Questa rappresentazione fornisce un intervallo insufficiente per l'uso in ambito scientifico.

Rappresentazione a virgola mobile (floating point)

L'idea alla base di questa rappresentazione consiste nel poter spostare la posizione della virgola dinamicamente sfruttando le proprietà dei sistemi posizionali che ci permettono di spostare la virgola attraverso moltiplicazioni per la base elevata ad un esponente:

$$(10,54)_{10} * 10^2 = (1054)_{10} \text{ o } (100,1)_2 * 2^{-2} = (1,001)_2$$

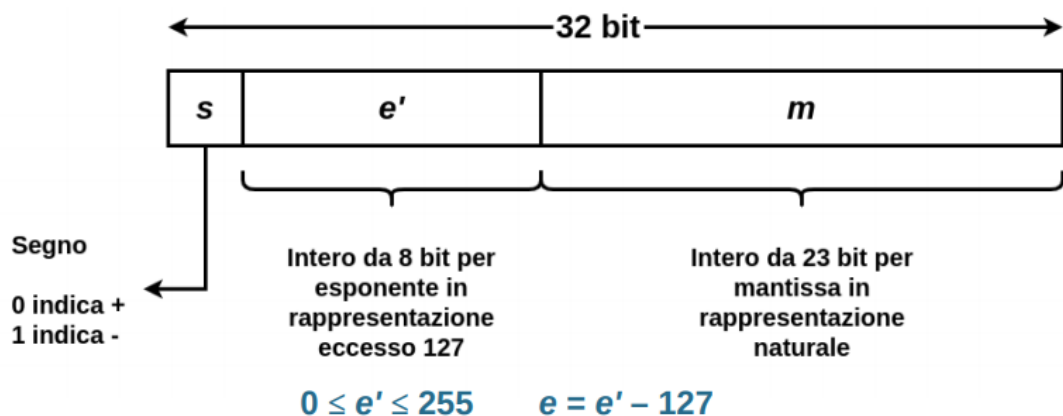
Si stabilisce la seguente struttura:

- Un bit di segno s per il numero;
- Una mantissa m contenente tutti i bit significativi escluso quello più significativo;
- Un esponente e con un segno in base 2;

Il numero sarà quindi rappresentato in *forma normale* come segue:

$$numero = \pm(1, m) * 2^e$$

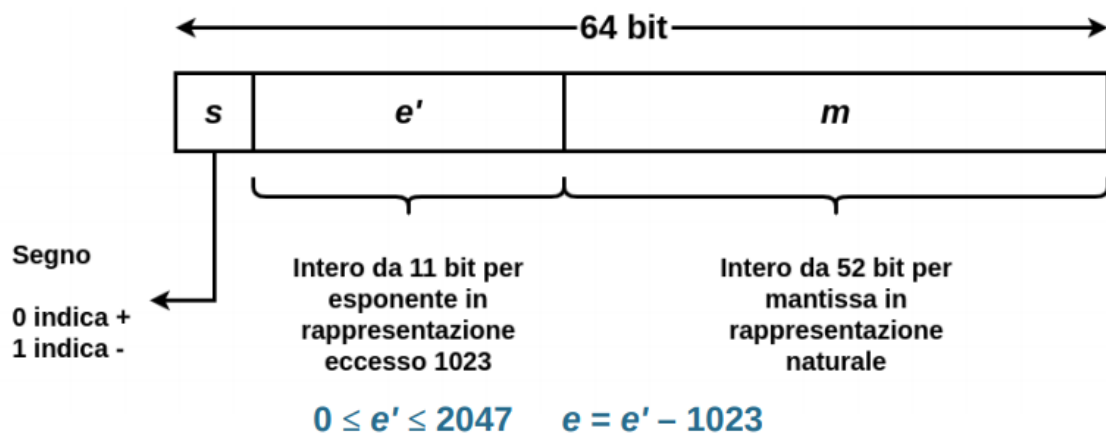
Standard IEEE 754 a 32bit a precisione singola



Valori speciali: $e' = 0, e' = 255$ Intervallo esponente: $-126 \leq e \leq 127$

Fattore di scala nell'intervallo: $[2^{-126}, 2^{127}]$

Standard IEEE 754 a 64bit a precisione doppia



Valori speciali: $e' = 0$, $e' = 2047$ Intervallo esponente: $-1022 \leq e \leq 1023$

Fattore di scala nell'intervallo: $[2^{-1022}, 2^{1023}]$

Accorgimenti

La conversione della parte decimale in base 2

Per effettuare il cambio di base della parte decimale di un numero in base 2 non si esegue la tecnica delle divisioni successive.

La conversione viene come segue:

1. Ipotizzando di voler convertire 5,25 si esegue la tecnica delle divisioni successive per la parte intera, ottenendo: 0101
2. Per convertire la parte decimale si moltiplica per 2 la parte decimale e si aggiunge dopo la virgola la parte intera ottenuta;
3. Il processo termina all'ottenimento del valore 0 o potrebbe non terminare mai;

Esempio: $(5,25)_{10} = (?)_2$

- Parte intera: 0101
- Parte decimale:

$0,25 * 2 = 0,5$ (si aggiunge uno 0 dopo la virgola)

$0,5 * 2 = 1,0$ (si aggiunge un 1 dopo la virgola)

$0 * 2 = 0$ (il processo termina)

$(5,25)_{10} = (0101.01)_2$

Note sugli standard IEEE

- L'esponente e' viene shiftato di 127/1023 per permettere esponenti negativi;
- La combinazione $e' = 0$ e $m = 0$ rappresenta lo 0 esatto;
- La combinazione $e' = 255(2047)$ e $m = 0$ rappresenta l'infinito;
- La combinazione $e' = 255(2047)$ e $m \neq 0$ rappresenta il NaN;
- La combinazione $e' = 0$ e $m \neq 0$ rappresenta un numero in forma non normale;

I caratteri e la rappresentazione binaria

L'idea alla base della rappresentazione binaria dei caratteri è quello di associare un carattere ad ogni possibile valore rappresentabile.

Una sequenza di n bit può rappresentare 2^n permutazioni di 0 e 1, di conseguenza è possibile rappresentare un alfabeto di 2^n simboli.
Esistono degli standard che definiscono alcuni alfabeti.

Uno degli standard più famosi è il *codice ASCII*, una codifica basata su 7 bit (=128 simboli, l'ottavo bit del byte è inutilizzato) che rappresenta lettere, cifre decimali, punteggiatura e caratteri speciali. Per facilitare l'uso le lettere e i numeri sono codificati con codici in ordine crescente. Il *codice ASCII* non è molto flessibile in quanto i caratteri che rappresenta sono adeguati alla lingua inglese, infatti è privo dei caratteri necessari per le altre lingue (come gli accenti per l'italiano).

Esistono standard internazionali che comprendono più caratteri come:

- *ISO 8859*: un'estensione del codice ASCII usando 8 bit, avente il doppio dei simboli;
- *ISO 10646*: una rappresentazione universale di caratteri che estende su più byte l'ISO 8859, da cui derivano per esempio *UNICODE* e *UTF-8*;

Il controllo d'integrità e la rappresentazione binaria

Cause di errori nelle informazioni codificate

Le informazioni codificate in binario possono essere affette da errori causati da disturbi nel canale di trasmissione oppure da un'alterazione, accidentale o voluta, nei dispositivi di memorizzazione. L'idea alla base per individuare, e possibilmente correggere, gli errori è quella di *aggiungere uno o più bit di controllo alla sequenza originale*.

Bit di parità

La tecnica consiste nell'aggiungere un bit di controllo, detto *bit di parità*, alla sequenza binaria originale con la seguente logica:

- 0: se il numero di 1 all'interno della sequenza è *pari*;
- 1: se il numero di 1 all'interno della sequenza è *dispari*;

Alla ricezione dell'informazione viene effettuato il conteggio dei bit a 1 e viene confrontato con il bit di parità, se il bit di parità non è valido allora la sequenza presenta degli errori. Non è un approccio molto sicuro in quanto possono accadere più errori che non alterano la parità.

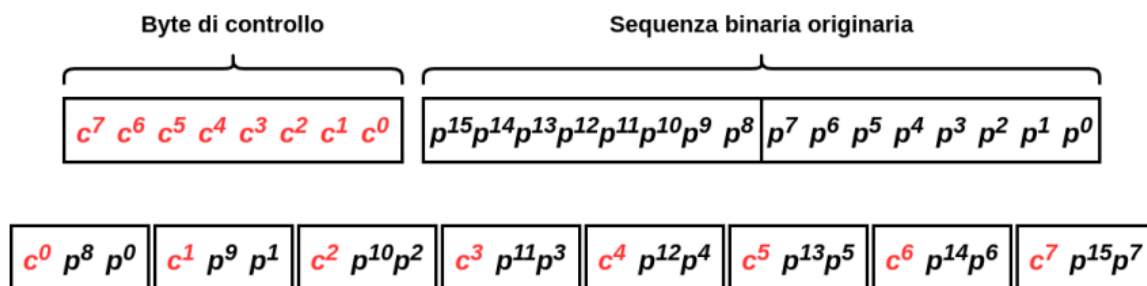
Questa tecnica permette di *rilevare solo errori su di 1 bit e non può correggerli*, in quanto non abbiamo informazioni su quali bit siano stati alterati.

Byte di controllo

L'idea è quella di poter effettuare controlli di integrità su *sequenze di più byte*.

La tecnica consiste nel:

- Aggiungere 1 byte, contenente *informazioni di controllo*, alla sequenza originale;
- Ogni bit del *byte di controllo* rappresenta la parità di *sequenze non contigue* di bit a distanza 8;
 - Tali *sotto-sequenze* saranno di dimensione $n + 1$, dove n è il numero di byte che costituisce la sequenza originale.



Ancora una volta, questo approccio non è molto sicuro in quanto più errori nei bit a distanza 8 potrebbero non alterare la parità.

Questa tecnica ci permette di *rilevare più errori in uno stesso byte* ma non ci permette di correggerli, in quanto non abbiamo informazioni su quali bit siano stati alterati.

Checksum (O somma di controllo)

L'idea alla base dei checksum è estendere la sequenza di controllo su più byte. Esistono diverse varianti, semplici e complesse:

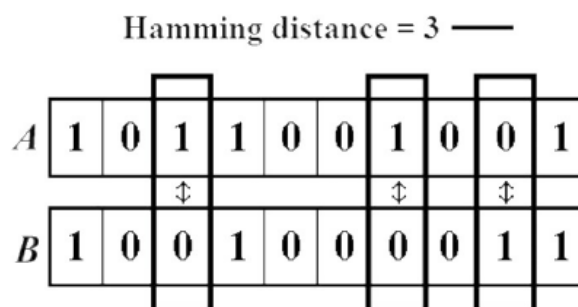
- Algoritmi *semplici e meno robusti*:
 - Somma di bit: dove la sequenza di controllo è costituita dal valore della somma dei bit che formano la sequenza iniziale;
 - Parità di sotto-sequenze;
- *CRC (codice a ridondanza ciclica)*:
 - *Algoritmi basati sull'algebra dei polinomi*;
 - Permettono di rilevare errori su lunghe sequenze di bit contigue;

Distanza di Hamming

Questa tecnica permette di avere *codici di controllo con le informazioni necessarie per correggere fino a k errori sui bit*.

La *distanza di Hamming* è definita:

- Su due sequenze, come il numero di bit diversi in posizioni corrispondenti;
- Su un insieme di sequenze, come la *distanza di Hamming* minima di coppie di sequenza distinte nell'insieme;



Un *codice c* a n bit che può rappresentare un alfabeto A di al più 2^n simboli è una funzione iniettiva: $c : a \rightarrow \{0,1\}^n$, ovvero associa ad ogni carattere dell'alfabeto una sequenza di 0 e 1 di dimensione n univoca.

La *distanza di Hamming di un codice* è quindi definita come la *distanza di Hamming della sua immagine*, l'immagine è l'insieme di sequenze binarie rappresentative del codice.

Si afferma quindi che un codice con:

- Distanza di Hamming h può *rilevare fino a $h - 1$ errori*;

- Distanza di Hamming $h =$ può *correggere fino a k errori*, con $k < \frac{h-1}{2}$
 - Ricevuto un'informazione con errori è possibile correggerla in quanto sappiamo che l'informazione (immagine) corretta è quella avente la *distanza di Hamming* minima rispetto all'informazione errata;
 - Tale approccio è possibile poiché fino a che vi sono k errori abbiamo solo un'immagine avente *distanza di Hamming minima*, con più errori si avrebbero più possibili immagini valide e non sarebbe possibile capire quale sia quella corretta;

I formati multimediali e la rappresentazione binaria

Tipologie di formati multimediali

I principali dati multimediali che usiamo sono di tipo:

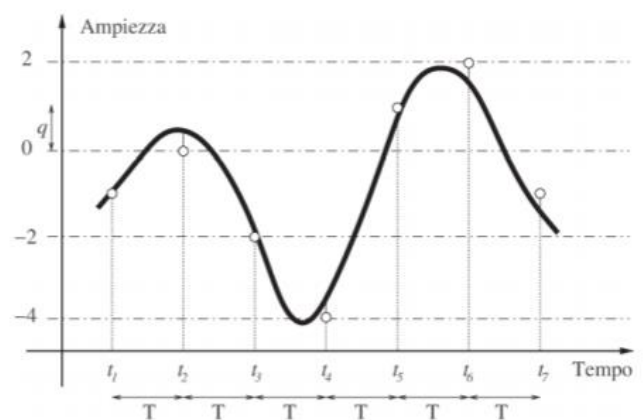
- Audio: variazione della pressione acustica nel tempo;
- Immagini: distribuzione di valori continui di luminanza in uno spazio bidimensionale;
- Video: una sequenza di immagini con eventuale accompagnamento sonoro;

Tali informazioni per poter essere utilizzati da un calcolatore devono essere digitalizzate al fine di ottenere numeri discreti, utilizzabili dal calcolatore.

Campionamento e Quantizzazione

La *discretizzazione del dominio* di un segnale (il tempo) è denominata *campionamento*:

- Il segnale viene suddiviso in intervalli di tempo regolari, ovvero si tiene in considerazione solo il valore assunto dal segnale nei momenti individuati;
- L'intervallo di tempo è denominato *periodo T* ;
- La frequenza di campionamento è definita come il reciproco del periodo, è espressa in *hertz (Hz)*;
- Per poter ottenere un segnale digitale fedele è necessario campionare con una frequenza doppia rispetto a quella del segnale analogico (Teorema di Nyquist-Shannon);



La *discretizzazione del codominio* di un segnale (l'intensità) è denominata *quantizzazione*:

- L'*unità di campionamento* è detta *q* ;
- I *valori analogici* del segnale vengono approssimati al *multiplo di q più vicino*;
- I valori quantizzati vengono rappresentati dai coefficienti di *q* ;

La compressione dei dati (fasi, categorie, rapporto di compressione)

La compressione di un'informazione consiste nel rappresentare la stessa informazione, o quasi, con sequenze binarie più brevi.

Il processo di compressione è costituito da due fasi:

- *Codifica*: $c(s) = t$, con cui si genera una *stringa compressa t* partendo dalla stringa *sorgente s* ;
- *Decodifica*: $d(t) \cong s$, con cui si ricostruisce esattamente, o quasi, la stringa *sorgente s* partendo dalla *stringa compressa t* ;

Una compressione può essere di tipo:

- *Lossless*: quando non vi è perdita di dati ed è possibile ricostruire esattamente l'informazione originale;
- *Lossy*: quando per comprimere vengono scartati dei dati e perciò l'informazione ricostruita è una approssimazione di quella originale;

Infine si definisce *rapporto di compressione* il rapporto fra il peso dell'informazione originale e quello dell'informazione compressa: $R = \text{len}(s)/\text{len}(c(s))$

Compressioni famose

- *Run-length*: consiste nel codificare il numero di occorrenze di ciascun simbolo;
- *Differenziale*: suddividere la sequenza in blocchi, salvare solo il primo elemento e i successivi come differenze dal precedente;
- *Codifica di Huffman*: codifica basata sull'entropia dei simboli;
- *Dizionario*: la sequenza viene descritta come parole di un dizionario;
 - Se il dizionario è creato dinamicamente, la tecnica si dice *adattiva*;

Algebra Booleana della commutazione

Cos'è l'algebra booleana

L'algebra booleana è un sistema algebrico in cui ogni variabile può assumere solo 2 valori: $\{0,1\}$.

Le funzioni logiche fondamentali (NOT, AND, OR)

L'algebra booleana possiede 3 operazioni base, dette *funzioni logiche fondamentali*:

- Somma logica, detta *OR*;
- Prodotto logico, detto *AND*;
- Complemento, detto anche *negazione* o *NOT*;

Ogni funzione logica prende in ingresso una o più variabili binarie ed ha una variabile binaria in uscita. Una funzione logica può essere espressa attraverso una *tabella di verità*, la quale è unica per ogni funzione.

La somma logica (OR)

La somma logica è una funzione che vale 1 solo se almeno uno dei suoi ingressi vale 1.

Graficamente è indicata con l'operatore $+$ e ha come forma algebrica: $f(x_1, x_2) = x_1 + x_2$

x_1	x_2	...	x_{n-1}	x_n	$f(x_1, x_2, \dots, x_{n-1}, x_n)$
0	0	...	0	0	$y_{00\dots00}$
0	0	...	0	1	$y_{00\dots01}$
...
1	1	...	1	0	$y_{11\dots10}$
1	1	...	1	1	$y_{11\dots11}$

La funzione *OR* gode delle seguenti proprietà:

- *Proprietà commutativa*, l'ordine degli operandi non importa:
$$x_1 + x_2 = x_2 + x_1$$
- *Proprietà associativa*, è possibile raggruppare:
$$x_1 + (x_2 + x_3) = (x_1 + x_2) + x_3$$
- *Estensione a più variabili*, è applicabile su più variabili in serie:
$$f = x_1 + \dots + x_n$$
- *Proprietà dell'elemento neutro*, lo 0 rappresenta l'elemento neutro: $0 + x = x$

x_1	x_2	$f(x_1, x_2) = x_1 + x_2$
0	0	0
0	1	1
1	0	1
1	1	1

Il prodotto logico (AND)

Il prodotto logico è una funzione che vale 1 solo se tutti i suoi ingressi valgono 1.

Graficamente è indicata con l'operatore $*$ e ha come forma algebrica: $f(x_1, x_2) = x_1 * x_2$

La funzione AND gode delle seguenti proprietà:

- *Proprietà commutativa*, l'ordine degli operandi non importa:

$$x_1 * x_2 = x_2 * x_1$$
- *Proprietà associativa*, è possibile raggruppare:

$$x_1 * (x_2 * x_3) = (x_1 * x_2) * x_3$$
- *Estensione a più variabili*, è applicabile su più variabili in serie:

$$f = x_1 * \dots * x_n$$
- *Proprietà dell'elemento neutro*, l'1 rappresenta l'elemento neutro:

$$1 * x = x$$

x_1	x_2	$f(x_1, x_2) = x_1 \cdot x_2$
0	0	0
0	1	0
1	0	0
1	1	1

Il complemento (NOT)

Il complemento è una funzione che inverte il valore della variabile in ingresso.

Graficamente è indicata con la sopra lineatura \bar{x} e ha come forma algebrica:

$$f(x_1) = \bar{x}_1$$

La funzione NOT gode della *proprietà di involuzione*:

$$\bar{\bar{x}} = x \text{ (doppia negazione)}$$

x	$f(x) = \neg x$
0	1
1	0

Alcune proprietà condivise dal OR e dal AND

SOMMA	Proprietà distributiva:	PRODOTTO
$x + y \cdot z = (x + y) \cdot (x + z)$		$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
Proprietà di idempotenza:		
$x + x = x$		$x \cdot x = x$
Proprietà di complemento:		
$x + \bar{x} = 1$		$x \cdot \bar{x} = 0$
Proprietà dello 1 e dello 0:		
$1 + x = 1$		$0 \cdot x = 0$

I teoremi di De Morgan

Vi sono 2 teoremi enunciati da De Morgan che affermano:

- *Addizione*: La negazione di una somma logica di variabili equivale al prodotto dei complementi delle variabili.

$$\overline{x_1 + x_2 + x_3 \dots} = \bar{x}_1 * \bar{x}_2 * \bar{x}_3 \dots$$

- *Prodotto*: La negazione di un prodotto logico di variabili equivale alla somma dei complementi delle variabili.

$$\overline{x_1 * x_2 * x_3 \dots} = \bar{x}_1 + \bar{x}_2 + \bar{x}_3 \dots$$

Le espressioni logiche

Combinando assieme più funzioni logiche fondamentali si ottengono le *espressioni logiche*, che rappresentano una possibile realizzazione di una funzione logica.

Esistono infinite espressioni logiche che realizzano *la stessa funzione logica*, ma esiste *solo una tabella di verità che descrive tale funzione*.

Due *espressioni logiche* si definiscono *equivalenti* se rappresentano la stessa funzione, ovvero se hanno la stessa tabella di verità.

$$(x_1 + x_2)(x_1 + \overline{x_2})(\overline{x_1} + x_2) = x_1 x_2$$

x_1	x_2	$f(x_1, x_2) = x_1 \cdot x_2$
0	0	0
0	1	0
1	0	0
1	1	1

La precedenza degli operatori nelle espressioni logiche

Operatore	Precedenza
Negazione - NOT	1
Prodotto - AND	2
Somma - OR	3

Per forzare la precedenza degli operatori si usano le parentesi.

Come ottenere la funzione logica dall'espressione logica

Per sapere quale funzione sia rappresentata da un'espressione logica basta calcolare la tabella di verità, ovvero calcolare i valori assunti dall'espressione per ogni possibile valore che possono assumere le variabili d'ingresso.

$$(x_1 + x_2)(x_1 + \overline{x_2})(\overline{x_1} + x_2)$$

x_1	x_2	$x_1 + x_2$	$x_1 + \neg x_2$	$\neg x_1 + x_2$	$f(x_1, x_2) = (x_1 + x_2)(x_1 + \neg x_2)(\neg x_1 + x_2)$
0	0	0	1	1	0
0	1	1	0	1	0
1	0	1	1	0	0
1	1	1	1	1	1

Come ottenere un'espressione logica a partire dalla tabella di verità di una funzione (mintermini-prima_canonica, maxtermini-seconda_canonica)

Principalmente esistono due metodi per ottenere una possibile espressione logica di una funzione partendo dalla tabella di verità, basati rispettivamente su:

- Mintermini, che danno origine a una forma unica di rappresentazione detta *prima forma canonica*;
- Maxtermini, che danno origine a una forma unica di rappresentazione detta *seconda forma canonica*;

Il mintermine e la prima forma canonica

Il *mintermine* è una funzione a n variabili che vale 1 solo per una specifica configurazione di variabili e vale 0 in tutti gli altri casi.

Assumendo di saper ottenere le espressioni rappresentanti i mintermini che valgono 1 per tutte le configurazioni in cui la funzione f_1 vale 1 : $\{m_0, m_1, m_3, m_7\}$: la *somma logica dei mintermini equivale alla funzione cercata* $f_1 = m_0 + m_1 + m_3 + m_7$

Un mintermine di configurazione c di n variabili può essere rappresentato come prodotto delle sue variabili:

- In *forma diretta* se in c la variabile vale 1;
- In *forma negata* se in c la variabile vale 0;

La funzione viene quindi rappresentata da un'espressione nella forma di *somma di prodotto (SOP)* denominata *prima forma canonica*:

$$f_1 = \overline{x_1}\overline{x_2}\overline{x_3} + \overline{x_1}\overline{x_2}x_3 + \overline{x_1}x_2x_3 + x_1x_2x_3$$

Il maxtermine e la seconda forma canonica

Il maxtermine è una funzione a n variabili che vale 0 solo per una specifica configurazione di variabili e vale 1 in tutti gli altri casi.

Assumendo di saper ottenere le espressioni rappresentanti i maxtermini che valgono 0 per tutte le configurazioni in cui la funzione f_1 vale 0 : $\{M_2, M_4, M_5, M_6\}$: il prodotto logico dei maxtermini equivale alla funzione cercata $f_1 = M_2 * M_4 * M_5 * M_6$

Un maxtermine di configurazione c di n variabili può essere rappresentato come somma delle sue variabili:

- In *forma diretta* se in c la variabile vale 0;
- In *forma negata* se in c la variabile vale 1;

La funzione viene quindi rappresentata da un'espressione nella forma di *prodotto di somme (POS)* denominata *seconda forma canonica*:

$$f_1 = (x_1 + \overline{x_2} + x_3) * (\overline{x_1} + x_2 + x_3) * (\overline{x_1} + x_2 + \overline{x_3}) * (\overline{x_1} + \overline{x_2} + x_3)$$

La forma minima delle espressioni logiche

Una funzione logica può essere realizzata da infinite espressioni logiche, che possono differire di costo.

Il criterio di costo di una espressione logica può avere diverse forme:

- *Costo dei letterali*: cioè il numero di comparse di variabili nell'espressione stessa;
- *Numero di operatori e variabili*;
- *Tempi di esecuzione su circuiti*;

Le espressioni SOP e POS adottano il criterio di *costo dei letterali*.

Un'espressione in forma minima risulta più semplice ed economica da realizzare come circuito rispetto alle altre forme.

$$(x_1 + x_2)(x_1 + \overline{x_2})(\overline{x_1} + x_2) = x_1x_2$$

Costo 6

Costo 2

Minimizzare la prima forma canonica

Per minimizzare un'espressione in prima forma canonica si possono eseguire specifici passi basati sull'applicazione delle varie proprietà delle funzioni logiche fondamentali:

x_1	x_2	x_3	f_1
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

x_1	x_2	x_3	f_1
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

- Usando la **proprietà distributiva**, associare le coppie di mintermini che posseggono una sola variabile in forma discordante (diretta e negata)

$$\bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 = \bar{x}_1\bar{x}_2(\bar{x}_3 + x_3)$$

- Usare la **legge di complemento** per trasformare in 1 le somme di variabili complementari

$$\bar{x}_1\bar{x}_2(\bar{x}_3 + x_3) = \bar{x}_1\bar{x}_2$$

- Usare la **legge di idempotenza** per duplicare dei mintermini nel caso fosse necessario

$$x + x = x$$

Ottenere un'espressione minima dalla tabella di verità con le mappe di Karnaugh
Semplificare un'espressione logica in forma minima è spesso un processo complicato, oggi eseguito da applicativi specializzati.

Il metodo di *Karnaugh* è un metodo geometrico che facilita l'ottenimento della forma minima di una funzione, l'idea alla base è la rappresentazione della tabella di verità in una forma differente, detta *mappa di Karnaugh*, in cui si effettua la minimizzazione raggruppando geometricamente i mintermini.

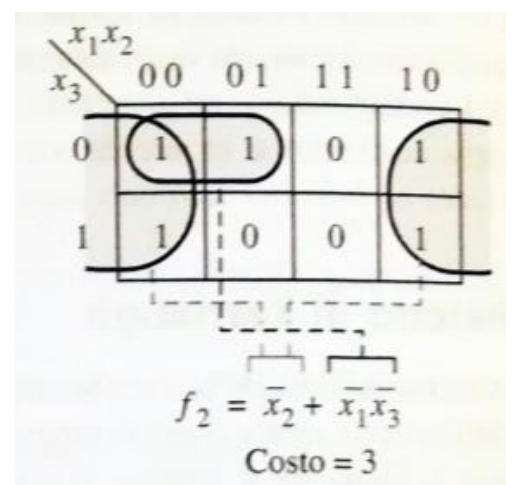
Tale metodo risulta vantaggioso per funzioni a poche variabili (3 o 4), con un numero maggiore di variabili la facilità viene meno a causa della necessità di creare più mappe una sopra l'altra.

Una *mappa di Karnaugh* è una *mappa bidimensionale* che rappresenta la tabella di verità, in cui le colonne e le righe rappresentano coppie di variabili ordinate in modo che le caselle adiacenti abbiano solo una variabile dal valore differente.

Inoltre la mappa deve essere immaginata piegata su sé stessa, ovvero gli estremi si considerano adiacenti all'estremo opposto.

Il metodo di *Karnaugh* è abbastanza semplice:

1. Ottenuta la mappa, si raggruppano le celle di valore 1 adiacenti orizzontalmente e verticalmente;
2. Successivamente si cerca di creare raggruppamenti più grandi, con un numero di celle multiplo di 2;
3. Ogni gruppo rappresenta il prodotto delle variabili che rimangono inalterate (forma diretta se 1 e negata se 0);
4. Si ottiene un'espressione SOP in forma minima dove ogni gruppo rappresenta uno dei prodotti dell'espressione;



Condizione di indifferenza nelle mappe di Karnaugh

Spesso capita che una funzione logica non sia definita su tutte le combinazioni di valori delle sue variabili, le variabili inutilizzate si dicono in *condizione di indifferenza* e nella tabella di verità vengono con X.

È possibile assegnare a piacimento il loro valore al fine di minimizzare il più possibile un'espressione logica, ma tale processo non è facile.

Circuiti e porte logiche

Le porte logiche

Le operazioni logiche base (*AND*, *OR*, *NOT*) possono essere realizzate da semplici *circuiti elettrici* che prendono il nome di *porte logiche*.

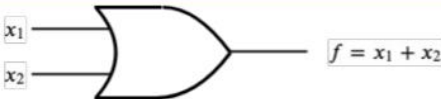
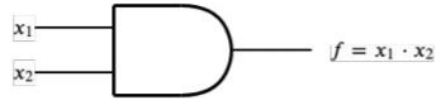
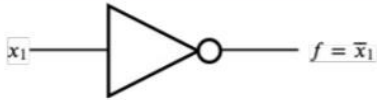
Le reti combinatorie

Una rete di porte logiche collegate fra loro prende il nome di *rete combinatoria* ed ha *n ingressi binari* e *m uscite binarie* con n e $m \geq 1$.

Le porte sono collegate a livelli e un livello comunica solo con quello immediatamente successivo, il risultato di una rete combinatoria dipenderà esclusivamente dagli stati assunti in un determinato tempo T .

Le porte logiche AND, OR, NOT

Le porte delle operazioni fondamentali possono essere **rappresentate graficamente**:

OR	
AND	
NOT	

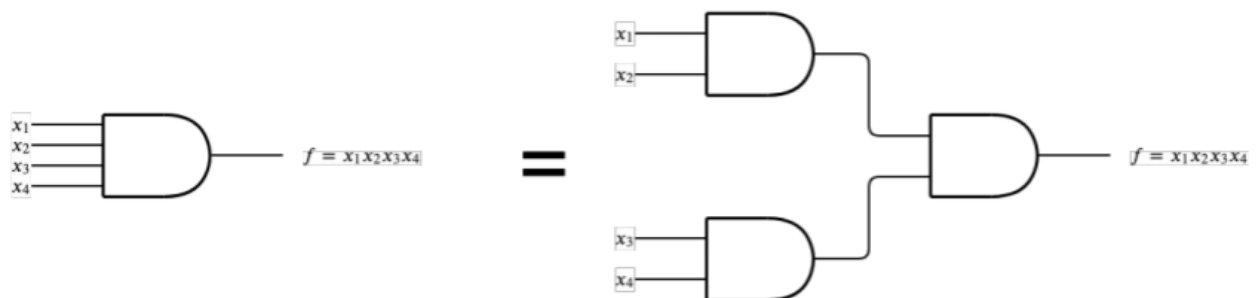
Collegando entrate e uscite delle porte si possono rappresentare le reti combinatorie

Le porte AND e OR a più ingressi

Grazie alla proprietà associativa, le porte AND e OR possono essere estese a più di due ingressi.

È possibile rappresentare graficamente tali porte con più ingressi, in quanto equivale a mettere in due livelli a cascata porte AND o OR a due ingressi.

Esempio porta AND a 4 ingressi:



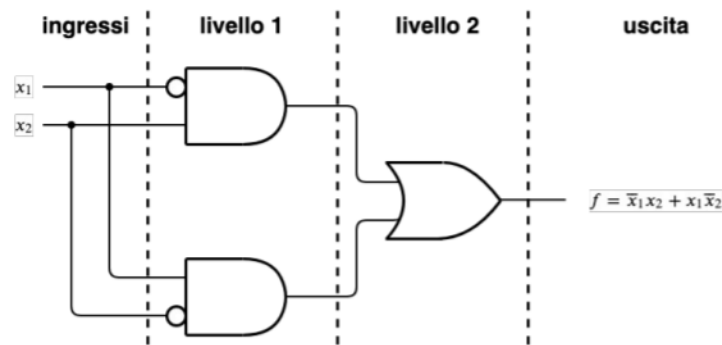
L'equivalenza fra le espressioni logiche e le reti combinatorie

Un'espressione logica può essere rappresentata come rete combinatoria con:

- Una porta per ogni operatore logico presente nell'espressione;
- Le porte sono collegate fra loro ad albero seguendo i livelli di priorità nell'espressione;

Si ha quindi che per ogni espressione logica si ha sempre una rete combinatorie e che esistono infinite reti combinatorie per una funzione logica.

Le espressioni SOP e POS corrispondono a reti a due livelli:



La differenza simmetrica (XOR o OR esclusivo)

La differenza simmetrica è una funzione che vale 1 solo se gli 1 nei suoi ingressi sono dispari.

Graficamente è indicata con l'operatore \oplus e ha come forma algebrica:

$$f(x_1, x_2) = x_1 \oplus x_2 = \overline{x_1}x_2 + x_1\overline{x_2}$$

La funzione XOR è rappresentata dalla seguente porta logica:



x_1	x_2	$f(x_1, x_2) = x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

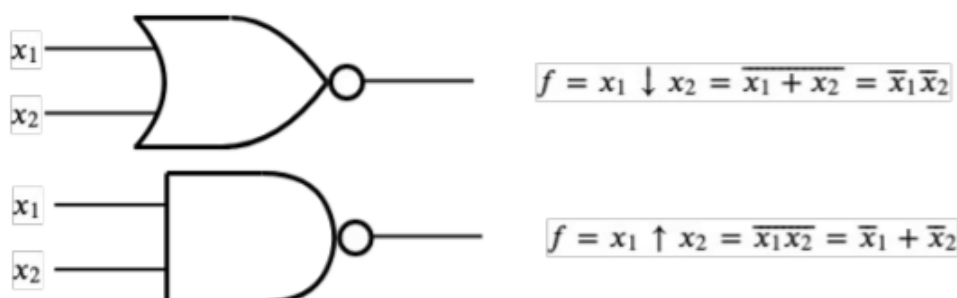
Le porte NAND e NOR

Due porte molto usate nell'ambito dei calcolatori sono le porte *NAND* e *NOR*, che si comportano rispettivamente come un *AND negato* e un *OR negato*.

Graficamente sono indicate con:

- *NAND* ha come operatore \downarrow e ha come forma algebrica: (vedi img)
- *NOR* ha come operatore \uparrow e ha come forma algebrica (vedi img)

Le funzioni NAND e NOR sono rappresentate dalle seguenti porte logiche:



x_1	x_2	$\neg(x_1 + x_2)$	$\neg(x_1 x_2)$
0	0	1	1
0	1	0	1
1	0	0	1
1	1	0	0

NAND e NOR, le porte universali

Le porte *NAND* e *NOR* sono considerabili porte *universali* in quanto è possibile realizzare una qualsiasi funzione logica attraverso l'uso di *solamente NAND* o *solamente NOR*.

Trasformare una SOP in una rete NAND

È possibile trasformare un'espressione SOP di porte binarie in una rete di sole *NAND*:

1. Cambiare tutte le porte *AND* e *OR* con porte *NAND*
2. *Mantenere gli ordini di priorità tra le operazioni dell'espressione di partenza;*

Bisogna inoltre saper realizzare:

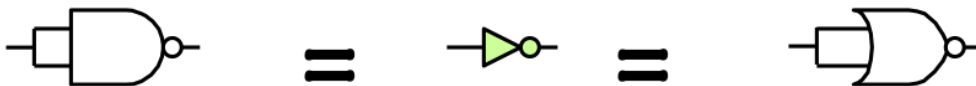
- Le porte *NOT* con porte *NAND*;
- Porte *NAND* a più ingressi;

$$\begin{aligned}(x_1 \uparrow x_2) \uparrow (x_3 \uparrow x_4) &= \overline{(\overline{x_1 \cdot x_2}) \cdot (\overline{x_3 \cdot x_4})} = \text{(De Morgan)} \\ &= \overline{\overline{x_1 x_2} + \overline{x_3 x_4}} = \text{(Involuzione)} \\ &= x_1 x_2 + x_3 x_4 \text{ (Sum of Products)}\end{aligned}$$

Realizzare la porta NOT con le porte NAND e NOR

Una porta *NAND* con ingressi unificati si comporta come una porta *NOT*, negando la variabile d'ingresso.

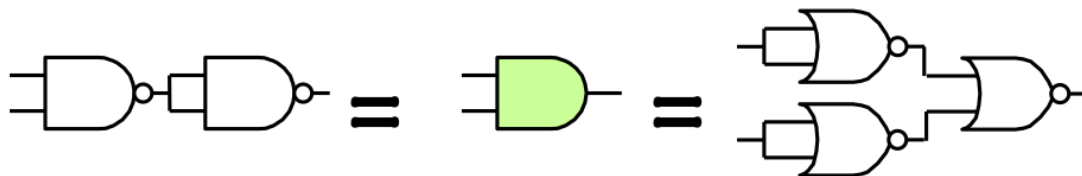
Si nota che vale lo stesso per la porta *NOR*.



Realizzare la porta AND con le porte NAND e NOR

Per realizzare una porta *AND* utilizzando solo porte *NAND* è semplicemente necessario negare una porta *NAND* (utilizzando una *NOT* realizzata con *NAND*).

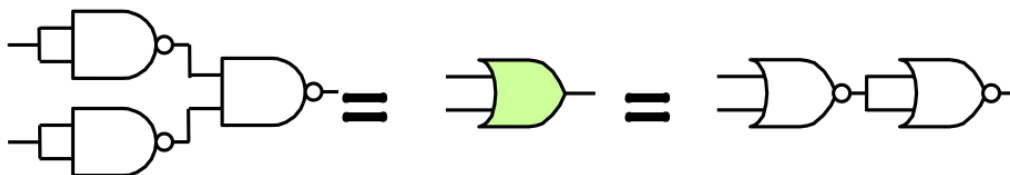
Per realizzare una porta *AND* utilizzando solo porte *NOR* è necessario negare gli ingressi di una porta *NOR* (utilizzando delle *NOT* realizzate con *NOR*).



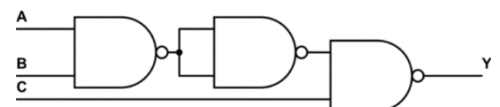
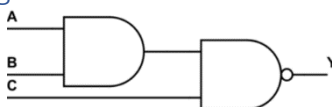
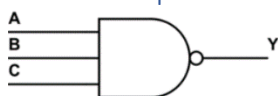
Realizzare la porta OR con le porte NAND e NOR

Per realizzare una porta *OR* utilizzando solo porte *NOR* è semplicemente necessario negare una porta *NOR* (utilizzando una *NOT* realizzata con *NOR*).

Per realizzare una porta *OR* utilizzando solo porte *NAND* è necessario negare gli ingressi di una porta *NAND* (utilizzando delle *NOT* realizzate con *NAND*).



Realizzare porte NAND a più ingressi



I Circuiti elettrici

Rappresentazione di variabili binarie in un calcolatore

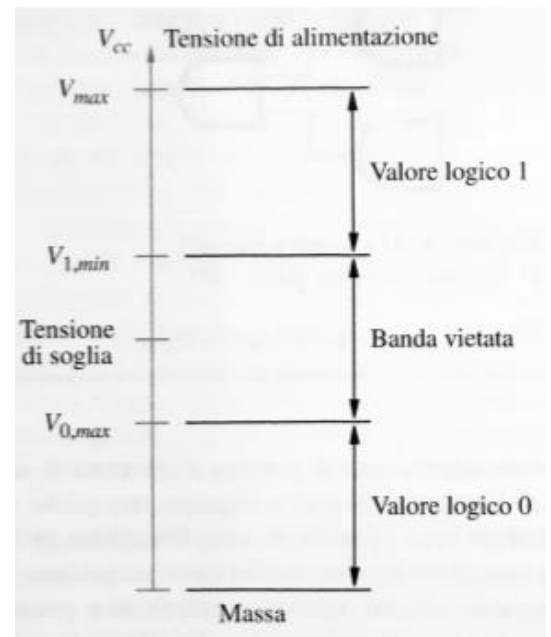
Come già detto il calcolatore è composto da circuiti elettrici.

Nei circuiti elettrici per rappresentare le variabili binarie si usano valori di tensione elettrica (*voltaggio*).

La tensione elettrica è però un valore continuo, per discretizzare tale valore si fa uso una *soglia di separazione*:

- I valori al di sopra di tale soglia vengono considerati come 1;
- I valori al di sotto di tale soglia vengono considerati come 0;

Il rumore presente all'interno di un circuito può però causare un effetto di aliasing, ovvero basse tensioni in prossimità della soglia di separazione che vengono considerate alte. Per risolvere a tale problematica i valori vicini alla soglia di separazione vengono ignorati, creandosi così una distinzione fra *banda di valore logico 0*, *banda vietata* e *banda di valore logico 1*.



I transistor MOS

I transistor sono delle componenti elettroniche che svolgono la funzione di *interruttore*: a seconda della tensione ricevuta in ingresso possono trovarsi in uno stato di:

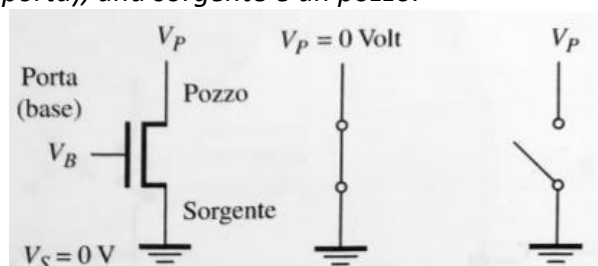
- *Conduzione*, quando si comportano da *interruttore chiuso*, chiudono il circuito e lasciano passare la corrente;
- *Interdizione*: quando si comportano da *interruttore aperto*, aprono il circuito bloccando il passaggio della corrente;

I transistor più comuni sono:

- $V_{alimentazione} = 5 \text{ volt}$ e $V_{soglia} = 2,5 \text{ volt}$;
- $V_{alimentazione} = 3,3 \text{ volt}$ e $V_{soglia} = 1,5 \text{ volt}$;

Più è alto il voltaggio di un transistor più esso è robusto contro il rumore, grazie alle bande più larghe, ciò però costituisce anche un maggiore consumo di energia.

I transistor più diffusi si basano sulla tecnologia *metallo-ossido semiconduttore (MOS)*, costituiti da una *base (porta)*, una *sorgente* e un *pozzo*.

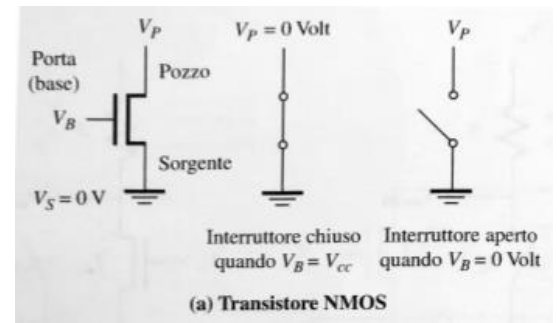


A seconda della tensione in ingresso nella *base*, il transistor collegherà o meno la *sorgente* al *pozzo*. Quando il transistor è in *conduzione* la tensione del pozzo sarà equivalente a quella della *sorgente*.

I transistor NMOS

I transistor di tipo NMOS sono caratterizzati dalle seguenti caratteristiche:

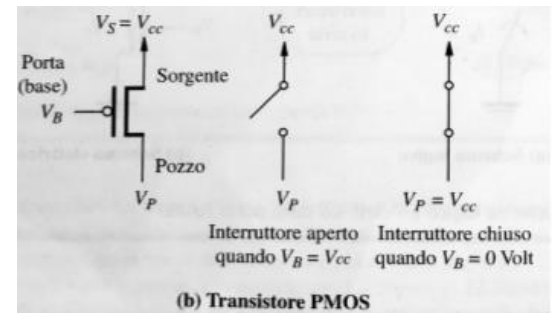
- Una tensione di base alta costituisce lo stato di conduzione;
- Una tensione di base bassa costituisce lo stato di interdizione;
- La sorgente è collegata alla massa;



I transistor PMOS

I transistor di tipo PMOS sono caratterizzati dalle seguenti caratteristiche:

- Una tensione di base alta costituisce lo stato di interdizione;
- Una tensione di base bassa costituisce lo stato di conduzione;
- La sorgente è collegata all'alimentazione;



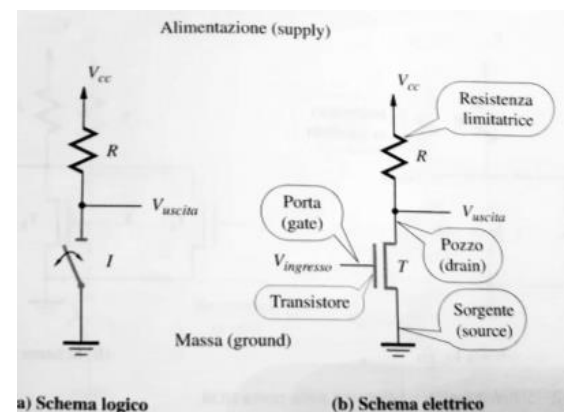
Il circuito NOT con NMOS

Un circuito che esegue la porta NOT lo si ottiene con un transistor NMOS:

- La sorgente è collegata alla massa;
- Il pozzo è collegato all'alimentazione mediante una resistenza;

Durante lo stato di interdizione (0) il pozzo sarà collegato con l'alimentazione, ottenendo in uscita una tensione alta (1).

Durante lo stato di conduzione (1) il pozzo sarà collegato alla massa, ottenendo in uscita una tensione bassa (0).

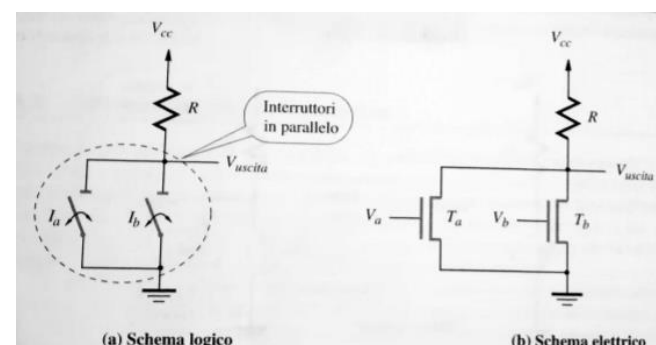


Il circuito NOR con NMOS

Un circuito che esegue la porta NOR lo si ottiene con due transistor NMOS collegati in parallelo:

- La sorgente è collegata alla massa;
- Il pozzo è collegato all'alimentazione mediante una resistenza;

Il pozzo avrà in uscita una tensione alta (1) solo quando entrambi i transistor sono entrambi in stato di interdizione (0).

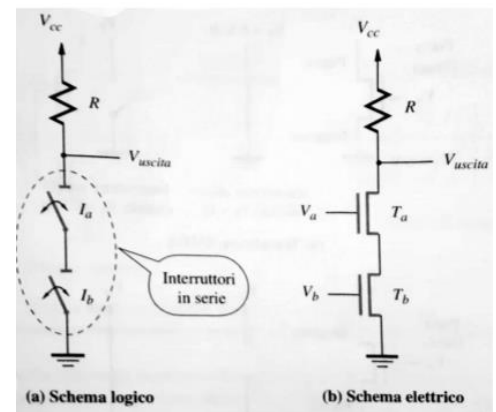


Il circuito NAND con NMOS

Un circuito che esegue la porta *NAND* lo si ottiene con due *transistor NMOS* collegati in serie:

- La *sorgente* è collegata alla massa;
- Il *pozzo* è collegato all'alimentazione mediante una resistenza;

Il pozzo avrà in uscita una tensione bassa (0) solo quando entrambi i transistor sono entrambi in stato di *conduzione*.



I transistor CMOS

I transistor *NMOS* presentano un problema: in stato di conduzione vi è una connessione diretta fra massa e alimentazione con una resistenza in mezzo per evitare cortocircuito, la quale aumenta il consumo energetico.

Questa problematica è stata risolta con la tecnologia *MOS Complementare (CMOS)*, che consiste in un circuito composto da un ramo di transistor *NMOS* collegato in serie ad uno di *PMOS*:

- Il punto di collegamento fra i due rami costituisce il *pozzo*.
- Ogni *NMOS* ha un *PMOS* complementare con cui condivide la tensione in ingresso.
- Gli ingressi dei rami costituiscono le alimentazioni nei MOS.

Il comportamento dei due rami è complementare:

- Quando il ramo *PMOS* è in interdizione, il ramo *NMOS* è in conduzione collegando la massa e il pozzo;
- Quando il ramo *NMOS* è in interdizione, il ramo *PMOS* è in conduzione collegando il pozzo all'alimentazione;

Si preferisce l'utilizzo dei *CMOS* poiché rispetto ai *NMOS* sono caratterizzati da:

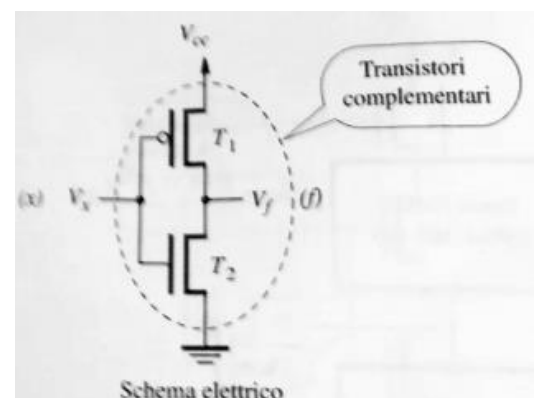
- Un consumo di potenza ridotto, in quanto vi è consumo solo in fase di commutazione;
- Sono di dimensioni più piccole, ciò permette un maggior numero di transistor e una più alta frequenza di commutazione;

Il circuito NOT fatto con CMOS

Un circuito che esegue la porta *NOT* basato su *CMOS* lo si ottiene con un transistor *NMOS* collegato in serie con un transistor *PMOS*.

Quando un transistor è in stato di interdizione l'altro è in stato di conduzione.

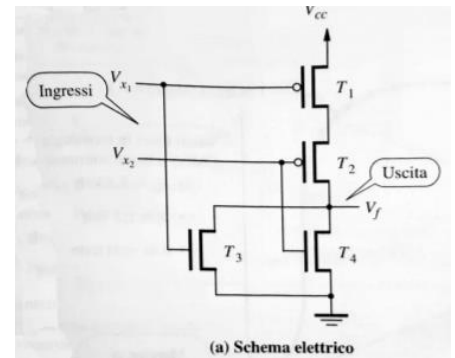
Il pozzo avrà in uscita una tensione alta (1) quando il ramo dei *PMOS* è in conduzione.



Il circuito NOR con CMOS

Un circuito che esegue la porta *NOR* basato su CMOS lo si ottiene con un ramo di due *transistor NMOS* in parallelo collegato con un ramo di due *transistor PMOS* in serie.

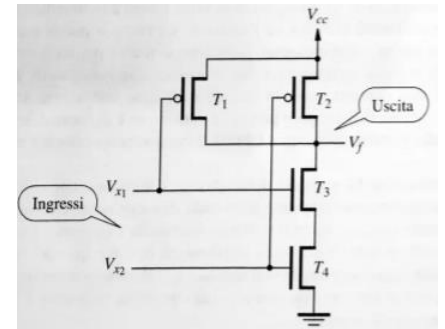
Il pozzo avrà in uscita una tensione alta (1) quando il ramo dei PMOS è in conduzione.



Il circuito NAND con CMOS

Un circuito che esegue la porta *NAND* basato su CMOS lo si ottiene con un ramo di due *transistor NMOS* in serie collegato con un ramo di due *transistor PMOS* in parallelo.

Il pozzo avrà in uscita una tensione bassa (1) quando il ramo dei NMOS è in conduzione.



Il circuito AND/OR con CMOS

Il circuito che esegue la porta AND basato su CMOS è realizzabile collegando una porta NOT all'uscita di una porta NAND, ciò però costituisce una porta AND di 6 *transistor*, rispetto alla normale porta *NAND* a 4 *transistor*.

Questa osservazione la si può applicare in modo analogo per una porta *OR*.

L'utilizzo delle porte NAND e NOR nella realizzazione dei circuiti risulta più conveniente dell'uso di OR e AND poiché le prime hanno un costo inferiore.

Ritardi in un circuito

In un circuito i ritardi possono essere causati dai tempi di commutazione e dalle componenti stesse del circuito.

Definiamo:

- *Tempo di transizione*, il tempo impiegato da un segnale per transitare di livello (da 0 a 1 per esempio);
- *Ritardo di propagazione*, il tempo che impiega l'uscita di un circuito ad adattarsi ai nuovi valori in ingresso (*il tempo di "aggiornamento dello stato"*);
- Il ritardo di propagazione del percorso più lento che collega un ingresso a un'uscita è detto *critico*;
- *Frequenza di lavoro di un circuito*, definita come il numero di volte che esso commuta in un determinato tempo;

La porta tri-state

Uno delle problematiche più comuni nei circuiti è data dal fatto che *non è possibile collegare più segnali in un unico ingresso, poiché non sarebbe possibile distinguere i singoli valori d'ingresso.*

Questa limitazione può essere aggirata acquisendo la capacità di attivare un solo segnale in ingresso alla volta: per fare ciò si utilizza la porta *tri-state*.

La porta *tri-state* è caratterizzata da:

- Due ingressi: un ingresso rappresentante il *segnale desiderato* e un ingresso rappresentato un *segnale di controllo*, denominato *abilitazione*;
- Un'uscita a 3 stati (0, 1, alta impedenza (Z));

Il comportamento della porta *tri-state* dipende dall'ingresso di controllo:

- Quando l'*abilitazione* vale 1, l'uscita della porta equivale al *segnale di ingresso desiderato potenziato*;
- Quando l'*abilitazione* vale 0, l'uscita della porta è in *alta impedenza*;

La porta *tri-state* rappresenta una soluzione economica all'impiego dei *multiplexer*, i quali sono più costosi.

e	x	f
0	0	Z
0	1	Z
1	0	0
1	1	1



I circuiti integrati

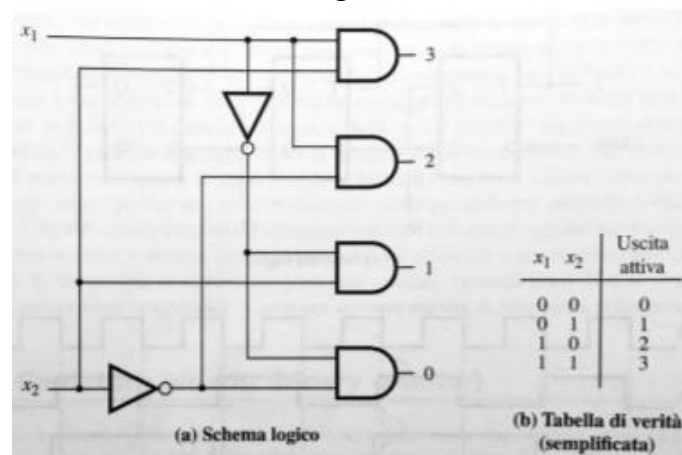
Le realizzazioni circuitali di porte logiche sono raggruppate in circuiti integrati, piastrine in silicio incapsulate in un involucro protettivo dotato di morsetti (pin) esterni.

Esistono 4 tipi di circuiti integrati a seconda della scala di integrazione:

- SSI (piccola): poche porte logiche;
- MSI (media): addizionatore, sottrattore, singoli registri, moltiplicatore, etc.
- LSI (grande): ALU, banco di registri, piccoli processori;
- VL0203SI (molto grande): memorie molto capaci, processori potenti;

Il decodificatore (decoder)

Il decodificatore, detto anche decoder, è un blocco funzionale combinatorio in grado di decodificare un codice binario in ingresso, possiede n ingressi e 2^n uscite e attiva la linea di uscita corrispondente al numero binario in ingresso.

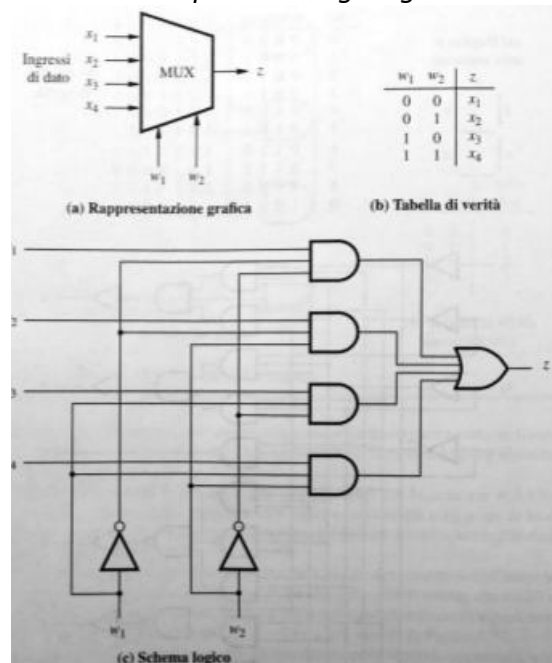


Il moltiplicatore (multiplexer)

Il moltiplicatore è un circuito logico in grado di selezionare uno dei suoi "ingressi dato" da convogliare nella sua uscita.

Il circuito del moltiplicatore consiste in n ingressi di selezione, 2^n ingressi dato e un'uscita. La selezione dell'ingresso dato da convogliare nell'uscita viene effettuata attraverso la configurazione degli n bit di selezione.

Può essere realizzata come somma dei prodotti degli ingressi.



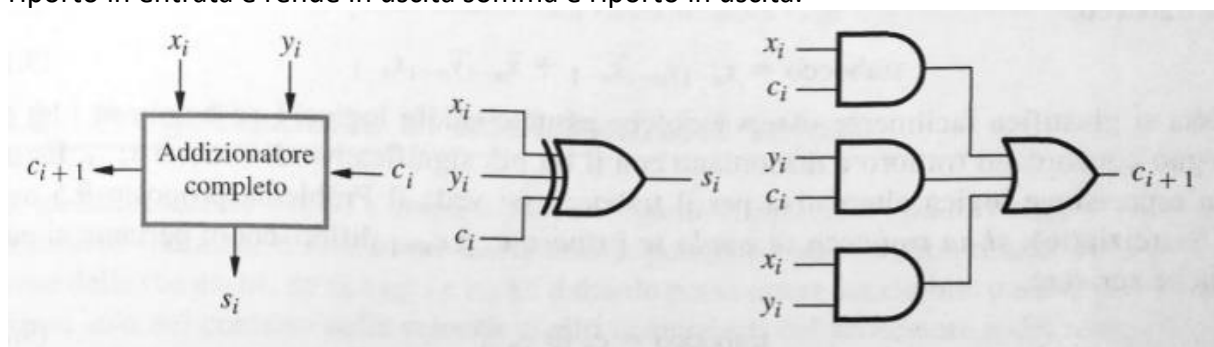
Un ulteriore impiego del multiplexer è la *sintesi di funzioni combinatorie*: una funzione logica a 3 variabili può essere rappresentata con un moltiplicatore con due ingressi di selezione.

L'addizionatore completo (full adder)

Un addizionatore tra due singoli bit può essere espresso da 2 funzioni logiche a tre ingressi (i due bit da sommare più il riporto in ingresso):

- $S_i = x_i \oplus y_i \oplus c_i$ che calcola la somma tra i bit ed il riporto in ingresso;
- $C_{i+1} = x_i c_i + y_i c_i + x_i y_i$ che calcola il riporto in uscita;

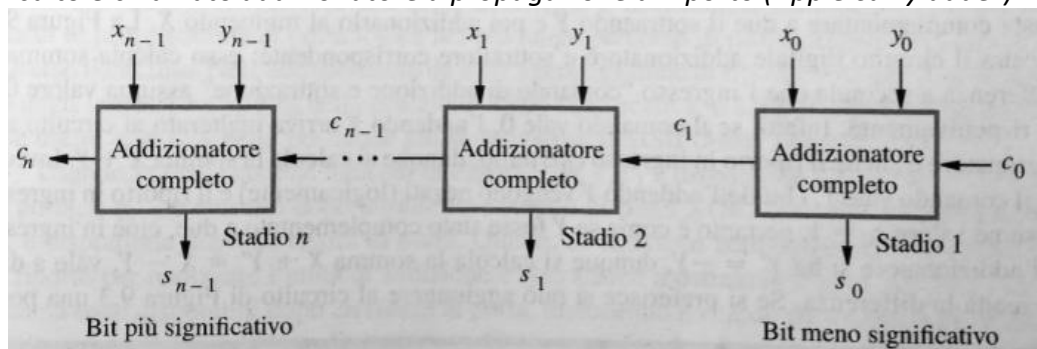
Unendo assieme in un singolo circuito le reti logiche per le funzioni di somma e riporto in uscita si ottiene l'*addizionatore completo*, che prende in ingresso i due bit da sommare e il riporto in entrata e rende in uscita somma e riporto in uscita.



L'addizionatore a propagazione del riporto (ripple carry adder)

Collegando una catena di n addizionatori completi in modo da propagare il riporto si ottiene un circuito in grado di sommare numeri binari di n bit.

Tale circuito è chiamato *addizionatore a propagazione di riporto (ripple carry adder)*.



L'addizionatore e il trabocco

A seconda di come vengono eseguiti i calcoli con l'addizionatore il trabocco viene gestito in modo diverso:

- Eseguendo la somma binaria naturale, è possibile sapere se vi è stato trabocco controllando il *riporto di uscita dell'ultimo addizionatore*, se è 1 allora vi è stato trabocco;
- Eseguendo la somma di *numeri in complemento a 2*, il trabocco viene indicato dall'espressione $\text{trabocco} = c_n \oplus c_{n-1}$

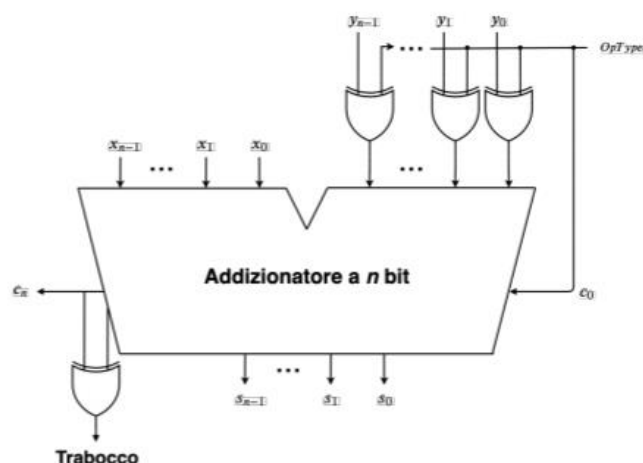
L'addizionatore algebrico a n bit

Utilizzando lo stesso circuito dell'addizionatore a propagazione del riporto è possibile effettuare sottrazioni di numeri in complemento a 2:

- Si nega i bit y ;
- Si fornisce al primo addizionatore il valore 1 come riporto in ingresso;

Per rendere un circuito di un addizionatore generico possiamo collegare ciascun bit y come ingresso a una porta XOR, gli ingressi rimanenti delle porte XOR e il riporto in ingresso del primo addizionatore saranno collegati a un *segnale di controllo, detto OpType*, che se posto a 1 eseguirà il complemento a 2 del numero y .

Si ottiene così un *addizionatore algebrico a n bit* capace di eseguire addizioni e sottrazioni, presenta pure la logica per il controllo del trabocco.

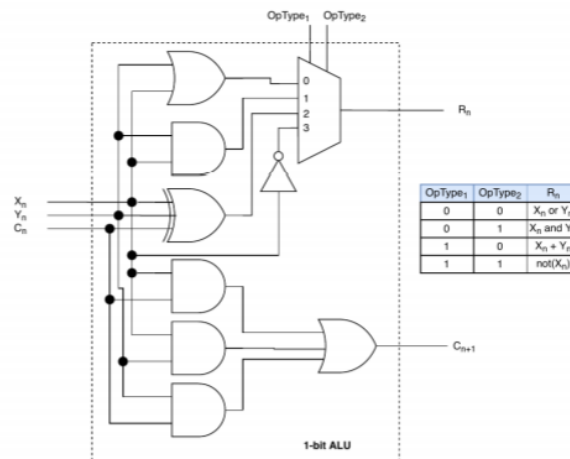


ALU a 1 bit

È possibile estendere l'addizionatore completo includendo anche la possibilità di effettuare le seguenti operazioni logiche bitwise AND, OR e NOT.

Si aggiunge dunque un multiplexer che consente di mandare in output, alternativamente, l'uscita del:

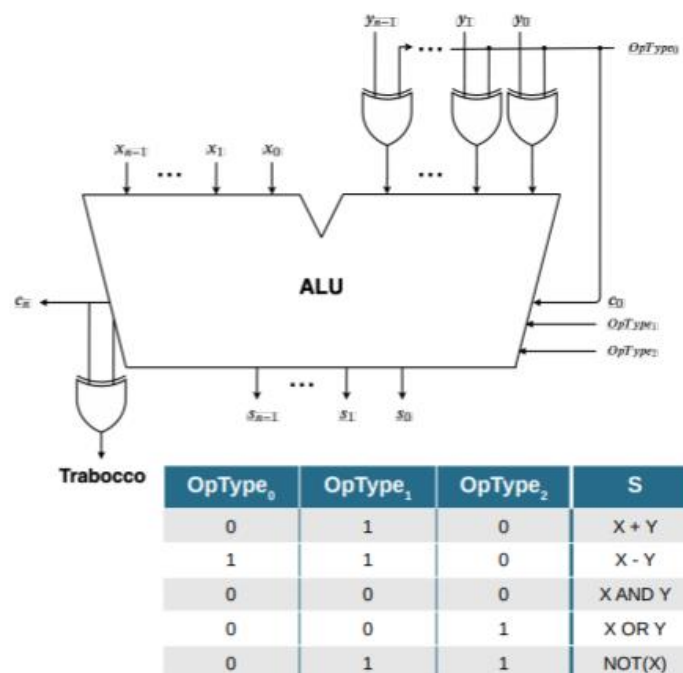
- Sommatore: $x_i + y_i$
- Porta AND: $x_i \text{ AND } y_i$
- Porta OR: $x_i \text{ OR } y_i$
- Porta NOT: \bar{x}_i



ALU a n bit

Collegando in serie n ALU a 1 bit in un'unità logica simile all'addizionatore a propagazione del riporto otterremo una *ALU a n bit*.

I bit di controllo *OpType1* e *OpType2* servono a selezionare l'operazione da eseguire (addizione, AND, OR o NOT), mentre *OpType0* serve a selezionare la sottrazione e complementare a 2 il sottraendo.



Le reti sequenziali

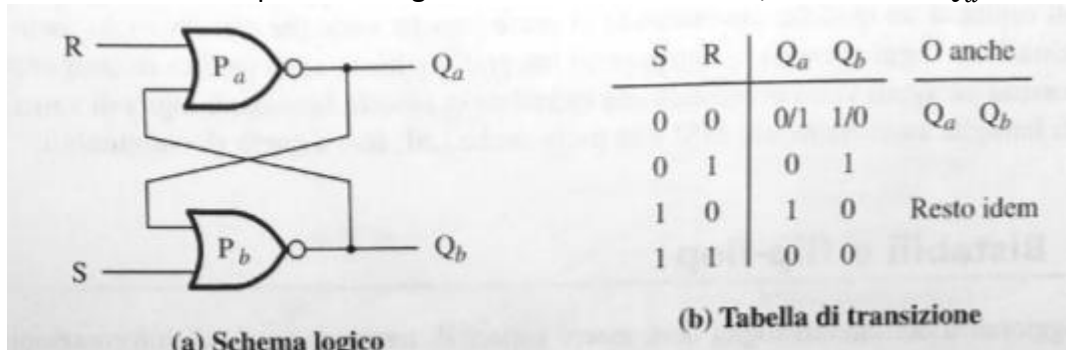
Le reti combinatorie non possono essere utilizzate per memorizzare un'informazione.

Vi è necessità di una rete logica le cui uscite non dipendano solo dall'input attuale ma anche dai suoi stati precedenti, ovvero presentano dei cicli, tali reti prendono il nome di *reti sequenziali*.

Le reti sequenziali lavorando nel tempo possono essere descritte da dei diagrammi temporali, che rappresentano nel tempo l'andamento dei segnali di ingresso e di uscita.

Bistabile asincrono

Il bistabile è una rete sequenziale in grado di memorizzare 1 bit, denominato Q_a .



Tale circuito ha il seguente comportamento:

- Tenendo i segnali *Set e Input* a 0, il bistabile mantiene la sua uscita precedente;
- Mettendo *Set* a 1 e *R* a 0 si ha Q_a a 0 e Q_b a 1;
- Mettendo *Set* a 0 e *R* a 1 si ha Q_a a 1 e Q_b a 0;
- Mettendo *Set* a 1 e *R* a 1 si ha Q_a e 0 e Q_b entrambi a 0;
 - Tale configurazione non viene usata poiché passando da questa a R0 e S0 non è possibile capire quale stato otterrà poiché le due porte non commuteranno in modo contemporaneo;

Non usando la configurazione 1 1 è possibile valutare l'uscita solo in funzione di Q_a in quanto Q_b è sempre il complementare di Q_a .

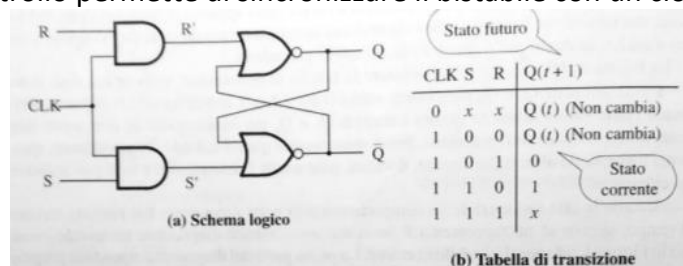
Un problema del bistabile asincrono è che il cambiamento dei suoi stati può avvenire in qualsiasi momento al cambiamento di R e S, ciò non permette una facile sincronizzazione con i segnali di altri circuiti.

Bistabile sincrono

Il bistabile sincrono risolve il problema della sincronizzazione del bistabile asincrono.

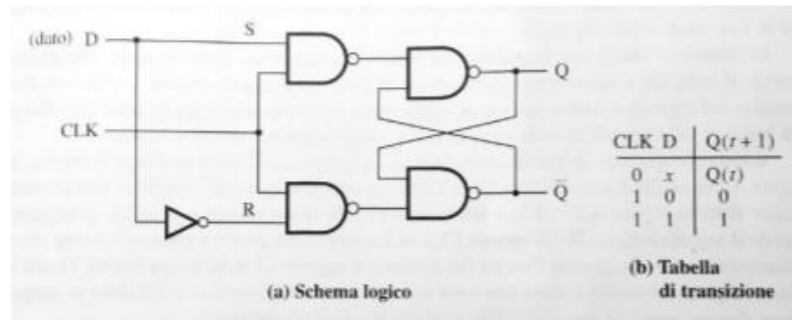
Presenta un *bit di controllo CLK* in ingresso oltre a *Set e Reset*:

- Quando CLK è a 0 lo stato non cambia in quanto è come avere R ed S a 0;
- Quando CLK è a 1 i segnali di R e S passeranno e si comporterà come un asincrono;
- L'input di controllo permette di sincronizzare il bistabile con un clock.



Bistabile di tipo D

Dato che Set e Reset si usano sempre con valore opposto, si può usare un singolo input D, che viene memorizzato dal bistabile.



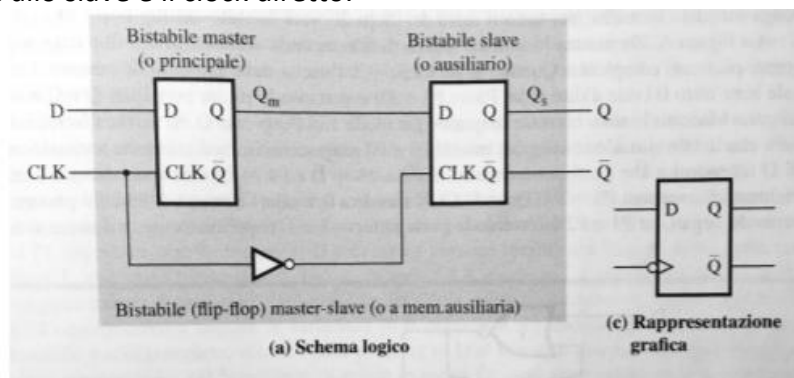
L'effetto trasparenza

L'effetto trasparenza rappresenta una problematica dei bistabili sincroni che consiste nelle commutazioni multiple ad ogni clock.

Flip-flop master-slave (o tipo D)

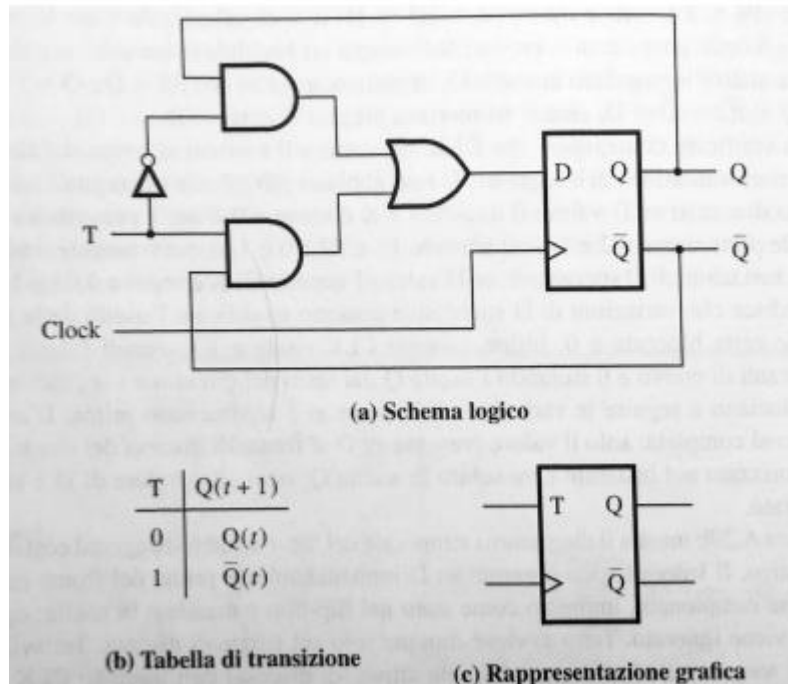
Il Flip-Flop master-slave è una rete sequenziale che risolve il problema della trasparenza. È formato da due bistabili D in serie (*master e slave*) che condividono il segnale CLK: ad ogni segnale di clock, il bistabile master cambia il proprio valore, il quale viene passato come input al bistabile slave, ma questo non cambia valore in quanto riceve un clock negato. Nel momento in cui il clock si abbassa, lo slave cambierà valore in funzione dell'ultimo input ricevuto dal master.

Questo tipo di flip-flop commuta solo durante il fronte di discesa del clock e perciò viene chiamato *edge-triggered negativo*, esiste la variante positiva ottenibile fornendo al master il clock negato e allo slave e il clock diretto.



Flip-flop tipo T

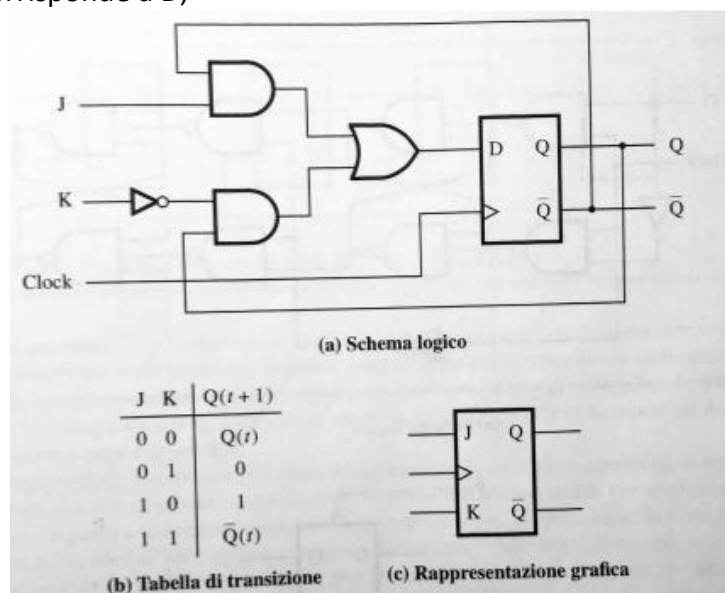
Il flip-flop di tipo T commuta stato ogni ciclo di clock se il suo input T è a 1, altrimenti viene riconfermato. A seconda del valore di T viene rimandata in ingresso ad un flip-flop di tipo D la sua uscita diretta o negata. Viene utilizzato per la realizzazione di contatori.



Flip-flop tipo JK

Il flip-flop di tipo JK unisce le funzionalità dei flip-flop di tipo D e T. Presenta due ingressi denominati J e K .

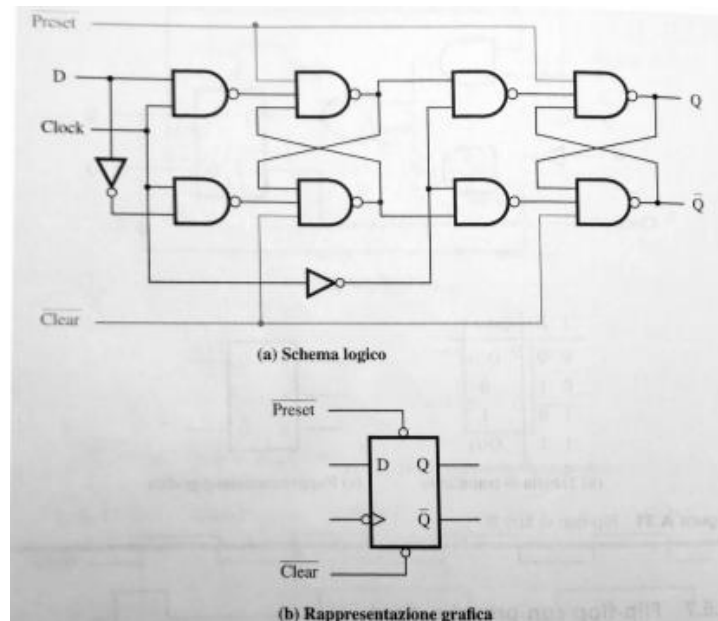
- Se lo stesso bit viene duplicato negli ingressi J e K , il circuito si comporta come un flip-flop T (1 commuta lo stato e 0 lo riconferma);
- Se J e K sono complementari, il circuito si comporta come un flip-flop di tipo D, dove l'input J corrisponde a D ;



Flip-flop con preset e clear

Spesso è necessario inizializzare lo stato del flip-flop indipendentemente dai valori di ingresso e dello stato corrente.

Aggiungendo due ingressi (preset e clear) è possibile ottenere questa funzionalità: quando preset è attivo si forza lo stato a 1, quando clear è attivo si forza a 0. Preset e clear sono attivi bassi e mai attivi nello stesso momento.



I Registri

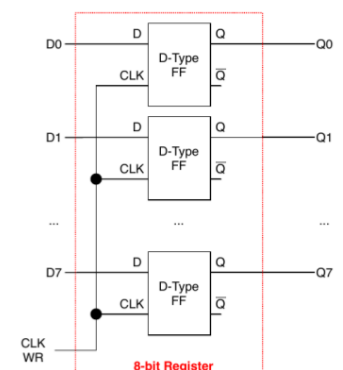
I registri sono dei circuiti elettronici in grado di immagazzinare una sequenza di n bit, Finora abbiamo visto blocchi logici in grado di memorizzare 1 bit (flip-flop).

I Registri paralleli

Un registro parallelo è formato da un insieme di flip-flop di tipo D collegati in parallelo ad uno stesso segnale di clock.

Permettono la lettura e la scrittura di tutti i bit in contemporanea. Il clock si comporta come una variabile di controllo di scrittura:

- Quando è pari a 1 i flip flop assumeranno il valore dei loro ingressi;
- Quando è 0 i flip flop manterranno il loro valore;



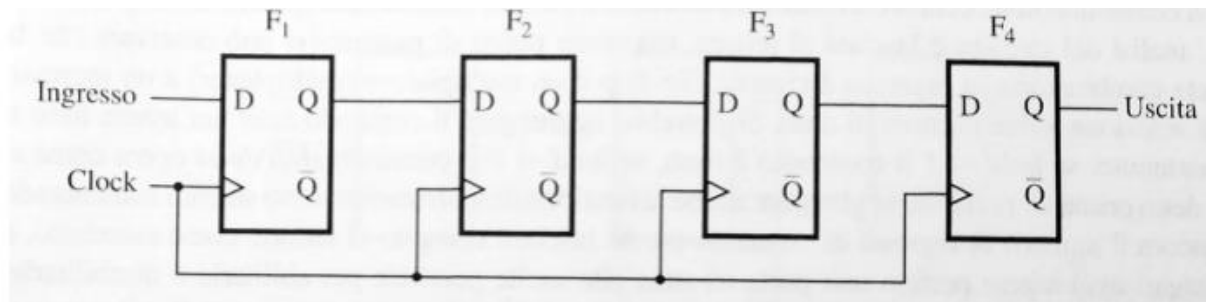
I Registri a scorrimento

Spesso potrebbe essere necessario scrivere o leggere i dati sequenzialmente bit per bit.

Inoltre si potrebbe voler scorrere a destra o sinistra il contenuto del registro. I registri a scorrimento permettono tali operazioni.

I registri a scorrimento sono formati da n flip-flop di tipo D collegati in serie attraverso le loro uscite/entrate: ad ogni ciclo di clock gli stati dei flip flop scorrono di una posizione a destra, il primo flip flop assume il valore in ingresso e il valore in uscita dall'ultimo flip flop verrà perso.

Leggere un registro a scorrimento equivale a leggere l'uscita dell'ultimo flip-flop. È possibile mantenere il contenuto del registro a scorrimento e farli semplicemente ruotare è possibile collegare l'uscita dell'ultimo flip flop con l'ingresso del primo flip-flop.



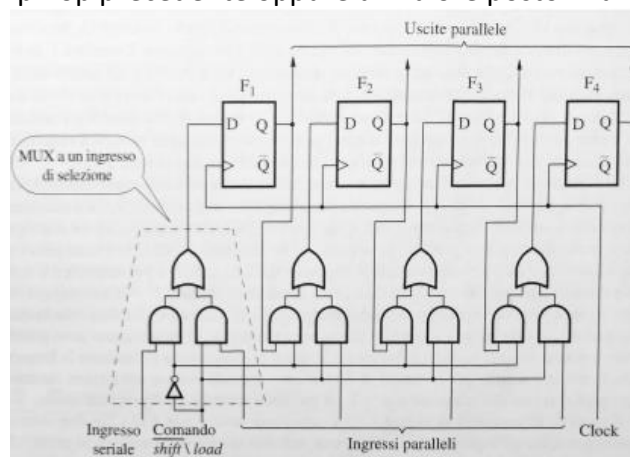
I Registri seriale-parallelo

Il registro seriale parallelo unisce le funzionalità dei registri paralleli e di quelli seriali.

Presenta un comando shift/load che permette di cambiare modalità di ingresso:

- Quando il comando è pari a 0, il registro si comporta come un registro a scorrimento;
- Quando il comando è pari a 1 si comporta come un registro parallelo;

Si aggiunge un multiplexatore connesso all'ingresso di ogni flipflop che decide se dare ai flip flop il contenuto del flip flop precedente oppure un valore posto in un ingresso parallelo.



I Registri universali

Si definisce *registro universale* un registro seriale-parallelo capace di fare shift in tutte e due le direzioni.

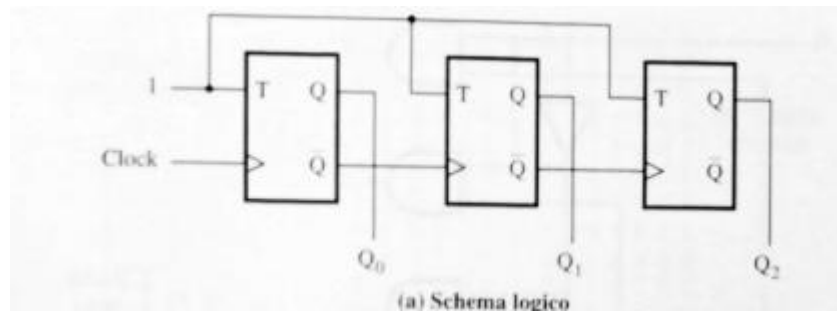
Il Contatore

Un contatore a n bit è in grado di percorrere la sequenza di numeri da 0 a $2^n - 1$ ciclicamente, incrementando di 1 unità ogni ciclo di clock.

È realizzato attraverso n flip-flop di tipo T, collegando in serie l'uscita \bar{Q} di ogni flipflop all'ingresso di clock del flip-flop successivo. Tutti i flip-flop condividono lo stesso segnale in ingresso.

Il numero attuale del conteggio è dato dalla stringa delle uscite Q (dal bit meno significativo al più significativo).

Ogni flip flop eccetto il primo commuta ad una frequenza pari a $1/(2^n)$ di quella d'ingresso, dove n è la posizione relativa del flip flop nella serie.



Istruzioni macchina

Come lavora un calcolatore?

Un calcolatore esegue le istruzioni *sequenzialmente*, ovvero una dopo l'altra. Inoltre non lavora sui singoli bit ma su gruppi di bit detti *parole* che a seconda dell'architettura del processore può variare da 8 a 64bit (sempre a potenze di 2).

Le informazioni, come istruzioni e dati, possono occupare una o più parole.

Organizzazione della memoria (indirizzamento e ordinamento byte)

Le informazioni vengono immagazzinate in memorie sotto forma di vettore di parole, ovvero una successione di parole, e l'unità minima di informazione indirizzabile in memoria è il byte.

Ad ogni parola del vettore viene associato un indirizzo univoco, inoltre le parole consecutive sono associate ad indirizzi consecutivi. Inoltre ai byte contenuti in ciascuna parola vengono assegnati indirizzi consecutivi. L'insieme degli indirizzi associati a ciascuna parola di memoria viene denominato spazio di indirizzamento. Gli indirizzi delle parole saranno quindi multipli della lunghezza in byte della parola utilizzata dal processore.

Vi sono due schemi di indirizzamento dei byte:

- Crescente, denominato *big-endian*, in cui i bit più significativi stanno a destra;
- Decrescente, denominato *little-endian*, in cui i bit più significativa stanno a sinistra;

Instruction set architecture ISA

Un processore è in grado di eseguire un insieme di operazioni base chiamate *istruzioni macchina*. L'insieme delle istruzioni eseguibili da un processore e le modalità d'uso è chiamata *instruction set architecture* ed è proprietario, ovvero ogni produttore ne implementa uno proprio.

Le istruzioni presenti dell'ISA sono definite utilizzando il linguaggio macchina, ovvero sequenze di 0 e 1.

Essendo arduo per un programmatore scrivere un programma direttamente in binario, si è inventato il linguaggio assembly che è una rappresentazione simbolica e leggibile del linguaggio macchina.

Le istruzioni di un ISA devono comprendere le seguenti funzionalità:

- Trasferimento dei dati fra memoria e registri del processore, come ad esempio il caricamento e il salvataggio;
- Trasferimento dei dati fra unità I/O e registri del processore, come ad esempio la lettura e la scrittura;

- Operazioni aritmetiche e logiche sui dati, come ad esempio addizione, sottrazione, confronto;
- Operazioni di controllo dell'ordine di esecuzione delle istruzioni, come ad esempio eseguire salti di istruzioni;

Tipologie di ISA (RISC e CISC)

Nella progettazione di un ISA esistono due diversi approcci:

- *Reduced Instruction Set Computer (RISC)* in cui:
 1. Si ha insieme di istruzioni base ridotto e inoltre ogni istruzione occupa solo una parola;
 2. Si ha una struttura generalmente semplice;
 3. Prestazioni elevate grazie all'elaborazione a stadi, usando le pipeline;
 4. Gli operandi delle operazioni aritmetiche e logiche devono trovarsi necessariamente nei registri del processore;
 5. Non è possibile fare trasferimenti diretti fra locazioni di memoria, bisogna passare per i registri;
- *Complex Instruction Set Computer (CISC)* in cui:
 1. Si ha insieme di istruzioni più complesse e inoltre ogni istruzione occupa più di una parola;
 2. Si ha una struttura generalmente più complicata;
 3. Non è possibile usare le pipeline;
 4. Gli operandi delle operazioni aritmetiche e logiche non devono trovarsi nei registri del processore;
 5. È possibile fare trasferimenti diretti fra locazioni di memoria;

Le locazioni di memoria sono identificate attraverso il loro indirizzo che può essere:

- Una costante numerica;
- Una costante simbolica dichiarata in precedenza, un'etichetta;

Istruzioni base per l'accesso alla memoria (Load e Store)

Una delle operazioni più basilari che un programma può eseguire è il caricamento/salvataggio di un dato dalla/in memoria.

Si definiscono le seguenti istruzioni base per l'accesso alla memoria:

- **Load "destinazione", "sorgente"**: carica un dato dalla memoria ad un registro del processore.
 - Il campo *destinazione* è il nome di un registro del processore;
 - Il campo *sorgente* è una locazione di memoria;
- **Store "sorgente", "destinazione"**: salva in memoria un dato presente in un registro del processore;
 - Il campo *sorgente* è il nome di un registro del processore;
 - Il campo *destinazione* è una locazione di memoria;

Istruzioni avanzate per l'accesso alla memoria (LoadByte e StoreByte)

Oltre alle operazioni basilari, potrebbe essere necessario leggere/scrivere singoli byte dalla/nella memoria.

Si definiscono le seguenti istruzioni avanzate per l'accesso alla memoria:

- **LoadByte "destinazione", "LOCBYTE"**: carica un singolo byte dalla memoria e lo salva negli 8 bit meno significativi di un registro del processore.
 - Il campo *destinazione* è il nome di un registro del processore;

- Il campo *LOCBYTE* è una locazione di memoria;
- ***StoreByte "sorgente", "LOCBYTE"***: salva in memoria un byte costituito dagli 8 bit meno significativi di un registro del processore;
 - Il campo *sorgente* è il nome di un registro del processore;
 - Il campo *LOCBYTE* è una locazione di memoria;

Istruzioni aritmetiche base (somma e sottrazione) (Add e Sub)

Si definiscono le seguenti istruzioni base per calcoli aritmetici come la somma e sottrazione, gli operandi vengono espressi come *nomi di registri o come valore immediato (#valore)*:

- ***Add "destinazione", "sorgente1", "sorgente2"***: somma il contenuto di due registri e ne salva il risultato.
 - Il campo *destinazione* è il nome di un registro del processore in cui verrà salvato il risultato;
 - I campi *sorgente1* e *sorgente2* rappresentano i numeri da sommare;
- ***Subtract "destinazione", "sorgente1", "sorgente2"***: sottrae il contenuto di due registri e ne salva il risultato.
 - Il campo *destinazione* è il nome di un registro del processore in cui verrà salvato il risultato;
 - I campi *sorgente1* e *sorgente2* rappresentano i numeri da sottrarre;

Istruzioni aritmetiche avanzate (moltiplicazione e divisione) (Multiply e Divide)

Oltre alle istruzioni base per la somma e sottrazione, solo alcune architetture hanno istruzioni avanzate per la moltiplicazione e la divisione di numeri in complemento a 2, rispettivamente implementabili come addizioni e sottrazioni successive.

Si definiscono le seguenti istruzioni avanzate per calcoli aritmetici come la moltiplicazione e divisione, gli operandi vengono espressi come *nomi di registri o come valore immediato (#valore)*:

- ***Multiply "destinazione", "sorgente1", "sorgente2"***: moltiplica il contenuto di due registri e ne salva il risultato.
 - Il campo *destinazione* è il nome di un registro del processore in cui verrà salvato il risultato.
Per rappresentare il prodotto servirebbe $2n$ bit:
 - Alcune architetture salvano solo i bit meno significativi del prodotto e i più significativi non vengono calcolati;
 - Alcune architetture salvano i bit meno significativi nel registro *destinazione* e i bit più significativi nel registro *destinazione + 1*;
 - I campi *sorgente1* e *sorgente2* rappresentano i numeri da moltiplicare;
- ***Divide "destinazione", "sorgente1", "sorgente2"***: divide il contenuto di due registri e ne salva il risultato.
 - Il campo *destinazione* è il nome di un registro del processore in cui verrà salvato il risultato, ovvero il quoziente. Il resto non viene calcolato. Alcune architetture potrebbero salvare il resto nel registro *destinazione + 1*;
 - I campi *sorgente1* e *sorgente2* rappresentano i numeri da dividere;

Istruzioni logiche base (AND e OR)

Si definiscono le seguenti istruzioni logiche base per le operazioni logiche, tali istruzioni agiscono *bit a bit* (*bitwise*) e gli operandi vengono espressi come *nomi di registri* o come *valore* (*#valore*):

- **AND** "*destinazione*", "*sorgente1*", "*sorgente2*": esegue un AND bit a bit sulle sorgenti e salva il risultato;
 - Il campo *destinazione* è il nome di un registro del processore in cui verrà salvato il risultato;
 - I campi *sorgente1* e *sorgente2* rappresentano delle sequenze binarie;
- **OR** "*destinazione*", "*sorgente1*", "*sorgente2*": esegue un OR bit a bit sulle sorgenti e salva il risultato;
 - Il campo *destinazione* è il nome di un registro del processore in cui verrà salvato il risultato;
 - I campi *sorgente1* e *sorgente2* rappresentano delle sequenze binarie;

Istruzioni di scorrimento logico (logical shift)

Le operazioni di scorrimento logico fanno scorrere i bit all'interno di un registro a destra o a sinistra di *n posizioni*. I *bit in uscita* vengono persi ad eccezione dell'ultimo bit che viene memorizzato nel bit di riporto C. Le posizioni lasciate libere vengono riempite con lo 0.

Si definiscono due istruzioni analoghe per le due direzioni:

- **LShiftR** "*destinazione*", "*sorgente*", "*N*": esegue uno shift verso destra di *n bit* del registro *sorgente* e ne salva il risultato nel registro *destinazione*;
 - *Uno shift verso destra applicato a un numero equivale alla divisione per 2^n*
- **LShiftL** "*destinazione*", "*sorgente*", "*N*": esegue uno shift verso sinistra di *n bit* del registro *sorgente* e ne salva il risultato nel registro *destinazione*;
 - *Uno shift verso sinistra applicato a un numero equivale alla moltiplicazione per 2^n*

Istruzioni di scorrimento aritmetico (arithmetical shift)

Una problematica dello scorrimento logico lo si ha nella rappresentazione dei numeri in complemento a 2: un numero negativo shiftato verso destra perderebbe il bit di segno a causa dell'aggiunto degli 0.

Lo scorrimento aritmetico nel caso di shift verso sinistra si comporta come uno scorrimento logico, mentre per lo shift verso destra il riempimento delle posizioni libere segue il bit più significativo. L'ultimo bit in uscita viene scritto *nel bit di riporto C*.

Si definiscono due istruzioni analoghe per le due direzioni:

- **AShiftR** "*destinazione*", "*sorgente*", "*N*": esegue uno shift verso destra di *n bit* del registro *sorgente* e ne salva il risultato nel registro *destinazione*;
 - *Uno shift verso destra applicato a un numero equivale alla divisione per 2^n*
- **AShiftL** "*destinazione*", "*sorgente*", "*N*": esegue uno shift verso sinistra di *n bit* del registro *sorgente* e ne salva il risultato nel registro *destinazione*;
 - *Uno shift verso sinistra applicato a un numero equivale alla moltiplicazione per 2^n*

Istruzioni di rotazione dei bit

Le operazioni di rotazione fanno scorrere i bit all'interno di un registro a destra o a sinistra di *n posizioni*, in cui i *bit in uscita* di un lato rientrano dall'altro. L'ultimo bit in uscita viene scritto *nel bit di riporto C*.

Si definiscono due istruzioni analoghe per le due direzioni:

- **RotateR** "*destinazione*", "*sorgente*", "*N*": esegue una rotazione verso destra di *n bit* del registro *sorgente* e ne salva il risultato nel registro *destinazione*;
- **RotateL** "*destinazione*", "*sorgente*", "*N*": esegue una rotazione verso sinistra di *n bit* del registro *sorgente* e ne salva il risultato nel registro *destinazione*;

Istruzioni di rotazione dei bit con riporto

Le operazioni di rotazione con riporto si comportano come la normale rotazione con l'eccezione che il contenuto del *bit di riporto C* viene incluso nella sequenza di bit da ruotare.

Si definiscono due istruzioni analoghe per le due direzioni:

- **RotateRC** "*destinazione*", "*sorgente*", "*N*": esegue una rotazione con riporto verso destra di *n bit* del registro *sorgente* e ne salva il risultato nel registro *destinazione*;
- **RotateLC** "*destinazione*", "*sorgente*", "*N*": esegue una rotazione con riporto verso sinistra di *n bit* del registro *sorgente* e ne salva il risultato nel registro *destinazione*;

Istruzione base per il salto condizionale

Si definisce la seguente istruzione base per il salto condizionale:

- **Branch_if_[condizione]** "*destinazione*": salta all'esecuzione dell'istruzione specificata nel caso in cui la condizione risulti vera.
 - Il campo *destinazione* è espresso come locazione di memoria contenente l'istruzione da eseguire nel caso in cui la condizione risulti vera;
 - La *condizione* può comprendere valori contenuti nei registri, rappresentati fra parentesi quadre, e/o valori espressi esplicitamente.

Esempio: (Se il contenuto di R2 è maggiore di 0, salta all'esecuzione di CICLO).

Branch_if_[R2] > 0 CICLO

Istruzione base per il salto non condizionale

Si definisce la seguente istruzione base per il salto non condizionale:

- **Branch** "*destinazione*": salta all'esecuzione dell'istruzione specificata.
 - Il campo *dest-inazione* è espresso come locazione di memoria contenente l'istruzione da eseguire nel caso in cui la condizione risulti vera;

Esempio: (Si salta all'esecuzione di CICLO).

Branch CICLO

Istruzione MOV (RISC e CISC)

L'istruzione *MOV* ha comportamenti differenti a seconda della tipologia di architettura:

- In una architettura *RISC*, l'istruzione permette di inserire in un registro *destinazione* il contenuto di un altro registro, denominato *sorgente*, o un valore immediato.
- In una architettura *CISC*, l'istruzione può eseguire le funzionalità delle istruzioni *RISC Load, Store e MOV*. Permette di effettuare trasferimenti di dati fra locazioni di memoria senza l'utilizzo dei registri, caratteristica delle architetture *CISC*.

Varianti delle istruzioni aritmetiche e logiche in CISC

L'architettura *CISC* presenta delle istruzioni aritmetiche e logiche differenti rispetto alle varianti *RISC*.

Le istruzioni CISC fanno uso di solo 2 *operandi*: il primo operando fornito farà sia da destinazione sia da sorgente.

Esempio: (La somma di R4 e R5 sarà inserita in R4)

Add R4, R5

Bit di esito o condizione

I *bit di esito o condizione* sono bit particolari memorizzati in un registro speciale del processore denominato *registro di stato*. Questi *bit* tengono traccia dell'esito di svariate operazioni e vengono spesso impiegati per le condizioni di salto.

Il registro di stato viene aggiornato ad ogni esecuzione di operazioni aritmetiche, logiche o trasferimento di dati.

Si definiscono i seguenti *bit di esito*:

- *N (Negative)*: Pari a 1 se il risultato è negativo, pari a 0 se è positivo o nullo;
- *Z (Zero)*: Pari a 1 se il risultato è nullo, pari a 0 altrimenti;
- *V (Overflow)*: Pari a 1 se è avvenuto trabocco in complemento a 2, 0 altrimenti;
- *C (Carry)*: Pari a 1 se è avvenuto trabocco in binario naturale, 0 altrimenti;

Cos'è un modo di indirizzamento?

Si definisce modo di indirizzamento il metodo con cui vengono specificati gli operandi e i risultati delle istruzioni, i quali possono essere espressi in modi diversi.

In particolare menzioniamo i seguenti modi:

- *Indirizzamento Immediato*;
- *Indirizzamento di Registro*;
- *Indirizzamento assoluto o diretto*;
- Indirizzamento indiretto da registro;
- Indirizzamento con indice e spiazzamento;
- Indirizzamento con base e indice;
- Indirizzamento con auto-incremento (CISC);
- Indirizzamento con auto-decremento (CISC);
- Indirizzamento relativo su PC (CISC);

Modo di indirizzamento immediato

Tale metodo di indirizzamento permette di usare una costante numerica come operando.

L'operando viene fornito in modo esplicito nell'istruzione: la costante numerica viene preceduta da un cancelletto.

Nei processori RISC si ha un limite sul numero di bit utilizzabili per il valore immediato causato dalla codifica dell'istruzione (vedi dopo), un processore a 32bit utilizza 16bit per la codifica del valore immediato.

Esempio: (Si salva in R4 la somma di R6 e 200)

Add R4, R6, #200

Modo di indirizzamento di registro

Tale metodo di indirizzamento consiste nel fornire all'istruzione il nome, ovvero l'indirizzo, di un registro del processore che contiene l'operando o conterrà il risultato.

Esempio: (Si salva in R4 la somma di R6 che contiene 5 ed R5 che contiene 9)

Add R4, R6, R5

Modo di indirizzamento assoluto o diretto

Tale metodo di indirizzamento è simile all'indirizzamento immediato, consiste nel fornire all'istruzione l'indirizzo di una parola di memoria che contiene l'operando o conterrà il risultato.

Come per l'indirizzamento immediato, vi è un limite sul numero di bit utilizzabili per la locazione di memoria fornita a causa della codifica dell'istruzione, un processore a 32bit utilizza 16bit per la codifica della locazione di memoria.

Esempio: (Si carica in R4 il contenuto della locazione di memoria avente etichetta NUM)

Load R4, NUM

Modo di indirizzamento indiretto da registro

Tale metodo di indirizzamento consiste nel fornire all'istruzione il nome di un registro del processore che contiene l'indirizzo di una locazione di memoria che contiene l'operando o conterrà il risultato. Il funzionamento è paragonabile al concetto dei puntatori di C e C++.

Questo metodo di indirizzamento è rappresentato ponendo il nome del registro tra parentesi tonde. Questa tecnica ci permette di riutilizzare una istruzione in memoria più volte semplicemente cambiando gli operandi, ovvero cambiando gli indirizzi contenuti nei registri.

Esempio: (Si carica in R4 il contenuto della locazione di memoria puntata da R2)

Load R4, (R2)

Modo di indirizzamento con indice e spiazzamento

Tale metodo di indirizzamento consiste nel fornire all'istruzione come indirizzo della locazione di memoria su cui operare la somma fra una costante numerica e un indirizzo di una locazione di memoria contenuto all'interno di un registro del processore.

Questo metodo di indirizzamento è rappresentato ponendo una costante numerica seguita dal nome del registro tra parentesi tonde. Questa tecnica viene utilizzata per gestire vettori o liste, i cui dati sono consecutivi con uno spiazzamento fisso.

Esempio: (Si carica in R4 il contenuto della locazione di memoria indicata da R2 con offset di 20)

Load R4, 20(R2)

Modo di indirizzamento con base e indice

Tale metodo di indirizzamento è simile all'indirizzamento per indice e spiazzamento con la differenza che viene fornita all'istruzione come indirizzo della locazione di memoria su cui operare la somma fra due registri, che fungono da base e indice.

Questo metodo di indirizzamento è rappresentato ponendo due registri tra parentesi tonde.

Esempio: (Si carica in R4 il contenuto della locazione di memoria indicata da R2 + R3)

Load R4, (R2, R3)

Modo di indirizzamento con auto-incremento (CISC)

Questo metodo di indirizzamento è caratteristico delle architetture CISC e raramente presente nelle architetture RISC.

Tale metodo di indirizzamento è simile all'indirizzamento indiretto, consiste nel fornire all'istruzione il nome di un registro del processore che contiene l'indirizzo di una locazione di memoria che contiene l'operando o conterrà il risultato, una volta eseguita l'istruzione l'indirizzo verrà incrementato di un'unità. Il funzionamento è paragonabile all'incremento *i++* di C e C++.

Questo metodo di indirizzamento è rappresentato ponendo un registro tra parentesi tonde seguito dal simbolo '+'. Può essere usato per eseguire più facilmente il *pop* di una pila. Esempio: (Si carica in R4 il contenuto della cima della pila e si incrementa il registro SP)

MOV R4, (SP)+

Modo di indirizzamento con auto-decremento (CISC)

Questo metodo di indirizzamento è caratteristico delle architetture *CISC* e raramente presente nelle architetture *RISC*.

Tale metodo di indirizzamento è opposto all'indirizzamento con auto-incremento, consiste nel fornire all'istruzione il nome di un registro del processore che contiene l'indirizzo di una locazione di memoria che contiene l'operando o conterrà il risultato, prima che venga eseguita l'istruzione l'indirizzo viene decrementato di un'unità. Il funzionamento è paragonabile al decremento *--i* di C e C++.

Questo metodo di indirizzamento è rappresentato ponendo un registro tra parentesi tonde preceduto dal simbolo '-'. Può essere usato per eseguire più facilmente il *push* di una pila. Esempio: (Si carica in pila, decrementando il registro SP, il contenuto della locazione di memoria *NUM*)

MOV -(SP), NUM

Modo di indirizzamento relativo su PC (CISC)

Questo metodo di indirizzamento è caratteristico delle architetture *CISC* e raramente presente nelle architetture *RISC*.

Tale metodo di indirizzamento è simile all'indirizzamento con indice e spiazamento con la differenza che come indice viene usato il registro speciale PC. La principale utilità di questo indirizzamento è la rappresentazione di indirizzi in dimensione ridotta, per esempio gli indirizzi delle istruzioni di salto invece che essere assoluti potrebbe far riferimento al registro PC e spiazamento.

Direttive di assembler

Le direttive di assembler sono comandi specifici per l'assembler. Queste direttive non vengono tradotte e inserite nel programma oggetto, ma forniscono informazioni utili al processo assemblativo.

Direttiva di eguaglianza (EQU)

Tale direttiva serve ad associare un valore numerico ad un nome, utilizzabile nel programma sorgente. Durante il processo assemblativo, l'assembler sostituirà ogni occorrenza del nome con il valore numerico indicato.

La sintassi generica è la seguente:

NOME EQU valore_numerico

Direttiva di origine (ORIGIN)

Tale direttiva indica all'assembler l'indirizzo di partenza, od *origine*, da cui inserire le istruzioni e i dati definiti nelle righe seguenti. A tutte le istruzioni e i dati seguenti la direttiva *ORIGIN* verranno assegnati gli indirizzi a partire dall'indirizzo *di origine*.

La sintassi generica è la seguente:

ORIGIN indirizzo_di_memoria

Direttiva di riserva di memoria (RESERVE)

Tale direttiva indica all'assemblatore di riservare uno spazio di memoria la cui dimensione è espressa in byte. Tale spazio di memoria viene solamente riservato, non viene effettuata alcuna inizializzazione.

La sintassi generica è la seguente:

RESERVE Spazio_in_Byte

Direttiva di riserva di memoria con inizializzazione (DATAWORD)

Tale direttiva indica all'assemblatore di riservare una parola di memoria e inizializzarla con il contenuto specificato.

La sintassi generica è la seguente:

DATAWORD Contenuto

Struttura di una linea di codice assembly

Una linea di codice assembly presenta i seguenti campi:

<i>Etichetta</i>	<i>Operazione</i>	<i>Operandi</i>	<i>Commento</i>
------------------	-------------------	-----------------	-----------------

- *Etichetta*: il nome che viene associato all'indirizzo di parola di memoria assegnata all'istruzione o all'indirizzo di memoria riservato. È opzionale.
- *Operazione*: il nome, codice operativo, dell'istruzione oppure una direttiva dell'assemblatore.
- *Operandi*: informazione di indirizzamento per accedere agli operandi.
- *Commento*: Testo che fa da commento, ignorato dall'assemblatore.

Notazione dei numeri

L'assemblatore ci permette di denotare i numeri in formati diversi:

- *Binario*: si utilizza il prefisso %, un esempio di somma con indirizzamento immediato:

Add R2, R3, #%0010

- *Decimale*: non si utilizza alcun prefisso, un esempio di somma con indirizzamento immediato:

Add R2, R3, #55

- *Esadecimale*: si utilizza il prefisso 0x, un esempio di somma con indirizzamento immediato:

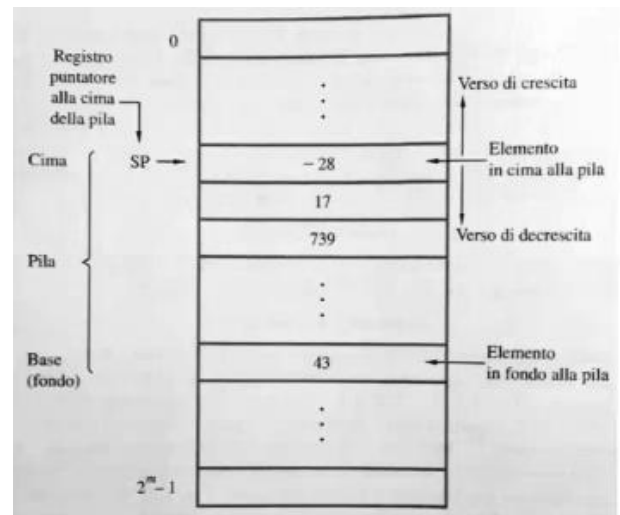
Add R2, R3, #0xA0

La pila (Stack) e le operazioni

La pila è una struttura dati costituita da una lista di elementi (quindi parole) impilati uno sopra l'altro. Gli elementi possono essere aggiunti o prelevati solo dalla cima della pila: l'ultimo elemento inserito è il primo ad essere prelevato, ciò costituisce la filosofia *LIFO* (*Last In First Out*).

L'ultimo elemento della lista viene chiamato *cima* mentre il primo elemento viene chiamato *base*: gli elementi della pila hanno indirizzi in ordine decrescente dalla base alla cima.

Per la gestione dello stack vi è un registro *speciale* dedicato denominato *Stack Pointer (SP)* che punta alla cima della pila, tale registro viene aggiornato ad ogni aggiunto o rimozione di elementi.



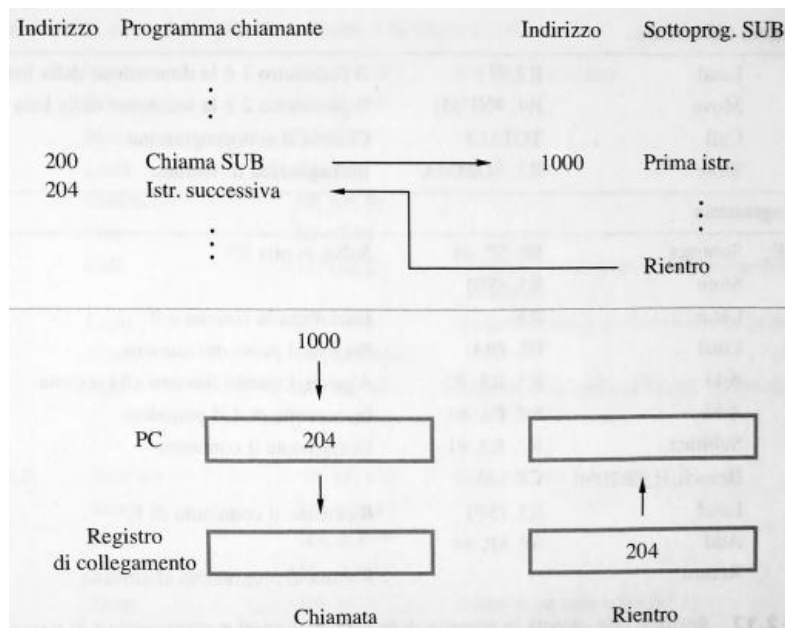
La pila prevede quindi le seguenti operazioni:

- **Push:** tale operazione inserisce un elemento in cima alla pila. Nel calcolatore questa operazione è rappresentata dalla diminuzione dell'indirizzo contenuto in SP e dal successivo Store in tale indirizzo. Esempio:
$$\text{Subtract SP, SP, \#4}$$
$$\text{Store } R_j, (\text{SP})$$
- **Pop:** tale operazione preleva un elemento dalla cima della pila. Nel calcolatore questa operazione viene rappresentata dal Load del contenuto puntato dall'indirizzo in SP e dal successivo incremento di tale indirizzo. Esempio:
$$\text{Load } R_j, (\text{SP})$$
$$\text{Add SP, SP, \#4}$$

Cos'è il sottoprogramma e le istruzioni di salto speciali

Un *sottoprogramma* è una lista di istruzioni che eseguono un compito specifico e che possono essere richiamate in un qualsiasi momento durante l'esecuzione di un programma. Il meccanismo di collegamento a una routine si basa sull'uso di 2 *istruzioni di salto specifiche* e 1 *registro speciale*:

- Il **Link Register** è un registro speciale in cui viene memorizzato l'indirizzo dell'istruzione da eseguire dopo il rientro del sottoprogramma.
- **Call instruction:** è una istruzione di salto che ci permette di richiamare un sottoprogramma durante l'esecuzione di un programma:
 1. Salva il contenuto del registro PC nel Link Register;
 2. Salta all'indirizzo di destinazione indicato nella chiamata Call;
- **Return instruction:** è una istruzione di salto che ci permette di rientrare dal sottoprogramma all'istruzione successiva alla chiamata Call:
 1. Salva il contenuto del registro Link Register nel PC;



Passare dei parametri ai sottoprogrammi e restituzione di un risultato (Passaggio per Registri e Passaggio per Pila)

In generale un sottoprogramma ha bisogno di:

- Parametri in ingresso su cui operare;
- Restituire un risultato al programma chiamante;

Questi bisogni rappresentano il *passaggio di parametri* fra un programma chiamante e il sottoprogramma.

Esistono principalmente 2 *tecniche* per il passaggio dei parametri:

- *Passaggio tramite registri*: i registri del processore sono accessibili sia dal programma sia dal sottoprogramma, è perciò possibile utilizzarli per poter passare informazioni fra i due. Risulta essere un approccio facile ed immediato ma è caratterizzato da una limitazione del numero di parametri dato dal numero di registri presenti;
- *Passaggio tramite la pila*: lo scambio di informazioni fra il programma e il sottoprogramma avviene mediante la pila, il primo inserisce i parametri nello stack mentre il secondo li legge e termina inserendo il risultato nello stack.

Un comportamento comune a entrambe le tecniche è l'impiego della pila per salvare il contenuto dei registri che verranno utilizzati dal sottoprogramma per poterli ripristinare al momento del rientro.

Area di attivazione in pila (stack frame)

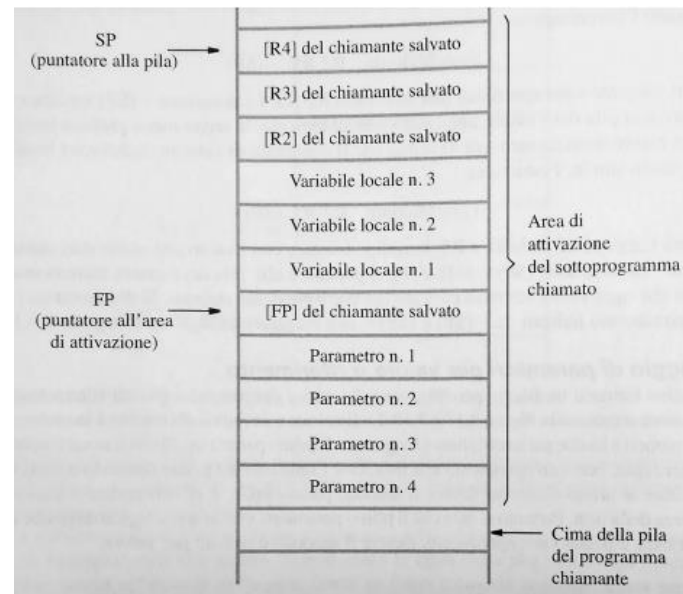
Si definisce *area di attivazione (stack frame)* di un sottoprogramma il blocco di memoria in pila riservato a quest'ultimo.

Il *Frame Pointer (FP)* è un registro speciale che punta allo Stack Frame del programma in esecuzione.

Lo stack frame contiene i parametri, il FP del programma chiamante, le variabili locali e i valori di registro salvati. È quindi possibile usare FP come indirizzo base per l'accesso agli elementi:

- Con uno spiazzamento positivo si ottengono i parametri;
- Con uno spiazzamento negativo si ottengono le variabili locali e i dati di registro salvati;

Il frame pointer punta alla parola dov'è memorizzato il FP del programma chiamante.



Annidamento dei sottoprogrammi (Problematica LR e approccio pila)

Un possibile problema dell'annidamento di sottoprogrammi è costituito dalla perdita del contenuto del registro Link Register:

1. Un programma chiama un sottoprogramma, viene inserito nel registro LR l'indirizzo di rientro per il programma;
2. Il sottoprogramma chiama a sua volta un ulteriore sottoprogramma, viene inserito nel registro LR l'indirizzo di rientro per il sottoprogramma chiamante;
3. L'indirizzo di rientro per il programma viene perso.

Una soluzione per tale problematica consiste nel salvare gli indirizzi di rientro, ovvero il contenuto del registro LR, delle chiamate annidate all'interno dell'area di attivazione dei programmi chiamanti. Le chiamate *Call* e *Return* si basano comunque sul registro LR, il contenuto del registro dovrà quindi essere gestito in modo adeguato con operazioni di caricamento e salvataggio.

Le architetture che non fanno uso del registro LR e usano direttamente un sistema basato sulla pila vengono detti *pila a modo implicito*.

Codifica numerica delle istruzioni

Nello stile RISC le istruzioni devono essere codificate in una parola a n bit. A seconda delle istruzioni, il tipo di indirizzamento usato e il tipo degli operandi esistono differenti codifiche.

In particolare menzioniamo le seguenti codifiche:

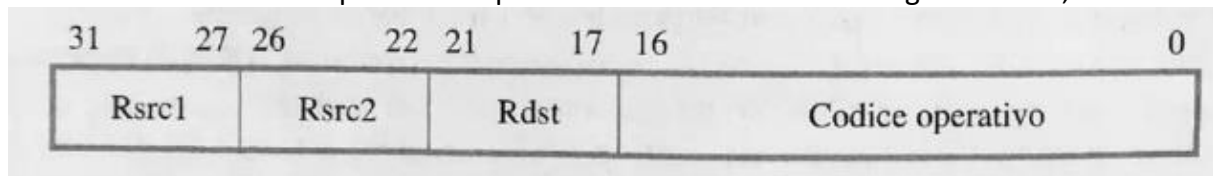
- *Formato con operandi in registri;*
- *Formato con operando immediato;*
- *Formato per chiamata;*

Codifica: Formato con operandi in registri

Questa codifica viene utilizzata per le istruzioni che utilizzano 3 registri come operandi, come le istruzioni aritmetiche e logiche.

La codifica dell'istruzione segue la seguente struttura:

- I 17 bit meno significativi rappresentano il codice operativo dell'istruzione;
- I 15 bit più significativi rappresentano i 3 operandi, 5 bit per ogni registro.
 - Si denota quindi che è possibile indirizzare fino a 32 registri diversi;

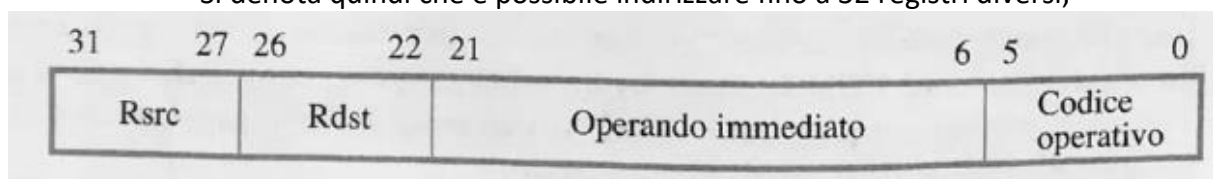


Codifica: Formato con operando immediato

Questa codifica viene utilizzata per le istruzioni che utilizzano 3 registri come operandi, come le istruzioni aritmetiche e logiche, e uno degli operandi è fornito tramite indirizzamento immediato oppure rappresenta una locazione di memoria.

La codifica dell'istruzione segue la seguente struttura:

- I 6 bit meno significativi rappresentano il codice operativo dell'istruzione;
- I 16 bit meno significativi successivi rappresentano il valore immediato, l'indirizzo di memoria o lo spiazzamento;
- I 10 bit più significativi rappresentano i 2 operandi rappresentati dai registri, 5 bit per ogni registro.
 - Si denota quindi che è possibile indirizzare fino a 32 registri diversi;

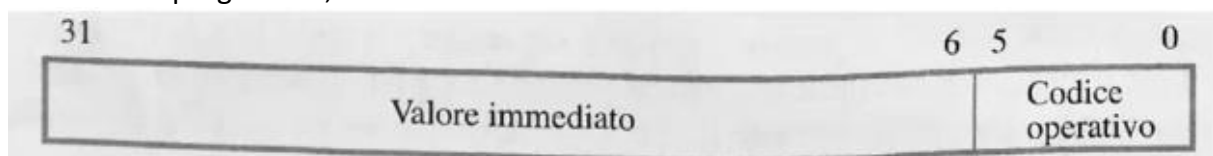


Codifica: Formato per chiamata

Questa codifica viene utilizzata per le istruzioni di chiamata a un sottoprogramma.

La codifica dell'istruzione segue la seguente struttura:

- I 6 bit meno significativi rappresentano il codice operativo dell'istruzione;
- I 26 bit più significativi rappresentano l'indirizzo della prima istruzione del sottoprogramma;



Operazioni I/O

Interfacce di dispositivo

Il collegamento tra le periferiche di I/O e il bus di sistema avviene tramite dei circuiti elettronici detti Interfacce di dispositivo.

L'interfaccia di ogni periferica contiene dei registri grazie ai quali riesce a interagire con il processore attraverso il BUS di sistema:

- Un *registro di dati*: usato come buffer per il trasferimento dati dal processore al dispositivo o viceversa;
- Un *registro di stato*: contenente informazioni sullo stato corrente del dispositivo, come la presenza di nuovi dati nel buffer;
- Un *registro di controllo*: contenente informazioni per controllare il comportamento del dispositivo, come per esempio la sua attivazione o disattivazione;

Spazio di indirizzamento di I/O

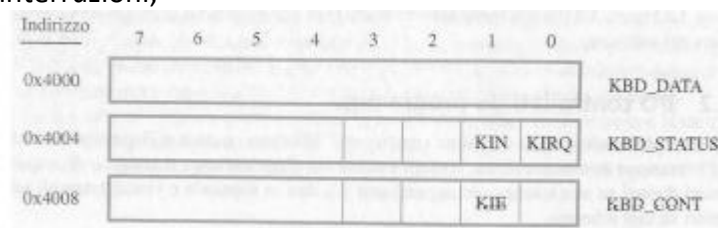
Molte architetture adottano un'organizzazione dello spazio di indirizzamento I/O denominato *Memory Mapped I/O*.

I registri delle interfacce appaiono al processore come un insieme di locazioni indirizzabili, infatti solitamente i dispositivi I/O e la memoria condividono lo stesso spazio di indirizzi del processore. Ciò permette alle istruzioni di accesso alla memoria di operare sugli indirizzi dei registri I/O o di memoria indistintamente.

L'interfaccia della tastiera

La tastiera è un dispositivo di input capace di inviare al processore la codifica ASCII del carattere corrispondente al tasto premuto. Assumiamo perciò che i registri della sua interfaccia siano a 8 bit, i quali sono:

- **KBD_DATA**: un registro di indirizzo *0x4000* usato come buffer in cui viene registrata la codifica ASCII di un carattere una volta premuto un tasto;
- **KBD_STATUS**: un registro di indirizzo *0x4004* contenente due *bit di stato relativi al dispositivo*:
 - **KIN**: quando il bit è posto a 1 indica che un nuovo carattere è disponibile;
 - **KIRQ (Keyboard Interrupt Request)**: quando il bit è posto a 1 indica che il dispositivo vuole fare una richiesta di interruzione;
- **KBD_CONT**: un registro di indirizzo *0x4008* contenente un *bit di controllo* denominato **KIE (Keyboard Interrupt Enable)** che se è posto a 1 permette al dispositivo di richiedere interruzioni;

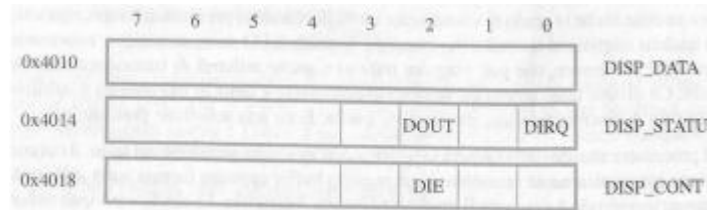


L'interfaccia dello schermo

Lo schermo è un dispositivo di output capace di rappresentare a video dei dati presenti in memoria. Assumiamo perciò che i registri della sua interfaccia siano a 8 bit, i quali sono:

- **DISP_DATA**: un registro di indirizzo *0x4010* usato come buffer in cui viene registrata il dato da mostrare a video;
- **DISP_STATUS**: un registro di indirizzo *0x4014* contenente due *bit di stato relativi al dispositivo*:
 - **DOUT**: quando il bit è posto a 1 indica che lo schermo è pronto a visualizzare un nuovo dato;
 - **DIRQ (Display Interrupt Request)**: quando il bit è posto a 1 indica che il dispositivo vuole fare una richiesta di interruzione;

- *DISP_CONT*: un registro di indirizzo *0x4018* contenente un bit di controllo denominato *DIE* (*Display Interrupt Enable*) che se è posto a 1 permette al dispositivo di richiedere interruzioni;



I/O controllati da programma

Il metodo più semplice per realizzare un programma in grado di gestire lo scambio di dati tra dispositivi I/O e processore è costituito dalla tecnica di *I/O controllati da programma*.

La tecnica prevede che il processore resti in attesa fino a quando la periferica I/O sia pronta per effettuare lo scambio di dati, ovvero si attende per esempio che il bit *KIN* o *DOUT* sia posto a 1.

Ciò risulta essere poco efficiente in quanto il processore rimane bloccato in attesa della risposta da parte della periferica I/O e non può eseguire altre istruzioni. Inoltre si ha un fenomeno di *collo di bottiglia*: le periferiche più veloci dovranno rimanere in attesa di quelle più lente.

Controllare lo stato di un singolo bit e istruzione TestBit (CISC)

Per poter operare con i dispositivi I/O è necessario valutare il contenuto dei singoli bit. In una architettura *RISC* è possibile usare l'istruzione di *AND* abbinato a una "maschera" che ci permetterà di valutare un bit specifico è posto a 1. La maschera è costituita da una sequenza binaria di 0 eccetto per la posizione del bit che si vuole valutare che è posto a 1. Esempio: (Vogliamo sapere se nel registro *KBD_STATUS* il bit *KIN* è pari a 1)

```
LoadByte R4, KBD_STATUS
AND R4, R4, #2
Branch_if[R4] = 0 SOMETHING
```

Le architetture *CISC* prevedono un'istruzione avanzata denominata *TestBit* capace di eseguire tale controllo:

- *TestBit "destinazione" #K*: controlla se il k-esimo bit in *destinazione* è pari a 0, in caso pone il *bit di condizione Z* a 0 o lo pone a 1 altrimenti.

Esempio: (Vogliamo sapere se nel registro *KBD_STATUS* il bit *KIN* è pari a 1)

```
TestBit KBD_STATUS, #1
Branch = 0 SOMETHING
```

Confronto del contenuto di due registri

Per confrontare il contenuto di due registri la strategia generale è eseguire la sottrazione e controllare se si ottiene 0.

Questa strategia è spesso utilizzata nelle architetture *RISC*.

Le architetture *CISC* prevedono invece un'istruzione avanzata denominata *Compare* capace di effettuare tale operazione:

- *Compare "destinazione" "sorgente"*: effettua la sottrazione dei due operandi e controlla se il risultato è pari a 0, aggiorna il *bit di stato* e scarta il risultato.

Esempio: (Vogliamo sapere se il R4 e R5 hanno lo stesso contenuto)

Compare R4, R5
Branch = 0 NOT_EQUAL

Tecnica di interruzione

Un altro metodo per gestire lo scambio di dati fra il processore e i dispositivi I/O si basa sulla *tecnica delle interruzioni*, ovvero i *dispositivi I/O* avvisano il processore quando sono pronti, lasciando quest'ultimo libero di eseguire altre istruzioni nel frattempo.

Questi *avvisi* vengono chiamati *segnali di interruzione* e vengono inviati attraverso una *linea dedicata* del *bus di controllo* denominata *Interrupt Request (INT_REQ)*.

Quando viene lanciato un *segnale di interruzione* il processore interrompe l'esecuzione del programma, salva il suo stato e salta all'esecuzione della routine di servizio dell'interruzione con i seguenti passi:

1. Viene salvato in memoria o in pila il contenuto del registro di controllo PS (Program Status) contenente informazioni come *i bit di esito e il bit IE*. Inoltre esiste un ulteriore registro denominato *IPS* che contiene una copia del registro PS precedente alla chiamata dell'interruzione.
2. Viene salvato in pila il contenuto dei registri temporanei usati dalla routine, le locazioni di memoria condivise fra l'interruzione e il programma interrotto, e il registro PC.
3. Viene aggiornato il registro PC per puntare alla prima istruzione della routine di servizio dell'istruzione.
4. Il ritorno da un'interruzione esegue il ripristino del registro PS, usando la copia presente in IPS;

Meccanismo dell'interruzione e annidamenti

La presenza del *bit di controllo IE* sia nel processore sia nei dispositivi I/O costituisce un meccanismo che ci permette di abilitare o disabilitare le interruzioni da entrambi le parti:

- Ponendo il *bit IE del registro PS del processore a 1*, si abilita il processore a *rispondere alle interruzioni*, mentre ponendolo a 0 il processore ignorerà le interruzioni ed eseguirà il programma ininterrottamente;
- Ponendo il *bit IE del registro di controllo del dispositivo I/O a 1*, si abilita il dispositivo a *inviare richieste di interruzione* mentre ponendolo a 0 il dispositivo non sarà in grado di richiederle;

Per evitare l'annidamento di interruzioni, il processore prima di iniziare un'interruzione pone a 0 il bit IE in PS, ciò permette di eseguire un'interruzione alla volta.

Alcuni dispositivi non possono però attendere molto prima di essere serviti, ciò necessita di gestire interruzioni annidate. Una soluzione consiste nell'usare un sistema di priorità:

- Attraverso l'utilizzo di alcuni bit del registro PS viene assegnato un livello di priorità all'attuale stato del processore;

- Qualora un dispositivo con priorità strettamente maggiore a quella attuale del processore invii una richiesta di interruzione, il processore passerà all'esecuzione di quest'ultima aggiornando di conseguenza il suo livello di priorità;
- Al termine dell'esecuzione dell'interruzione, il processore ritorna ad eseguire il livello di priorità precedente, contenuto nella copia di PS in IPS;

Gestione dispositivi multipli

Una delle problematiche riguardanti la gestione di dispositivi multipli consiste nel fatto che il processore deve essere in grado di riconoscere quale dispositivo I/O ha richiesto l'interruzione.

Approfondiamo due particolari tecniche che ci permettono di risolvere tale problematica:

- Polling
- Interruzione vettorizzata

Tecnica: Polling

La tecnica denominata *polling* si basa sull'utilizzo del *bit di controllo IRQ* presente nel registro di controllo delle interfacce dei dispositivi:

1. Quando un dispositivo vuole effettuare una richiesta di interrupt pone il suo *bit IRQ* a 1 ed invia un segnale nella linea *Interrupt Request*;
2. Quando il processore riceve il segnale per individuare il dispositivo richiedente scansiona tutti i bit *IRQ* dei dispositivi e salta all'esecuzione dell'interruzione del dispositivo avente tale bit posto a 1;
 - a. La scansione di tutti i bit *IRQ* può essere effettuata da un comparto hardware specializzato o lato software attraverso l'utilizzo di una routine generica;

Questa tecnica risulta semplice ma al tempo stesso inefficiente in quanto la scansione, a seconda dei dispositivi I/O connessi, potrebbe risultare molto costosa in fatto di tempo.

Tecnica: Interruzione vettorizzata

Una tecnica più efficace del *polling* è denominata *interruzione vettorizzata*.

Quando un dispositivo vuole richiedere un'interruzione manda nel bus un *codice univoco di identificazione* di pochi bit, circa 4 o 8, che viene utilizzato dal processore come *indice della tabella dei vettori d'interruzione*. Ogni campo della tabella contiene l'indirizzo iniziale della routine di servizio di uno specifico dispositivo che il processore caricherà nel registro PC per iniziarne l'esecuzione.

Nel bus viene mandato un codice invece che l'indirizzo iniziale della routine desiderata poiché mandare l'intero indirizzo risulta più costoso.

Registri di controllo del processore

Il processore ha al suo interno una serie di *registri di controllo* necessari per una corretta comunicazione con i dispositivi I/O:

- Il registro *PS (Program Status)* contiene informazioni sullo stato del programma, come *bit di esito*, *bit di priorità* o il *bit IE*;
- Il registro *IPS* è un registro in cui viene automaticamente inserita una copia del registro PS prima dell'esecuzione di un'interruzione;

- Il registro *IEnable* contiene un bit per ogni dispositivo I/O, porre a 1 il bit di uno dispositivo significa accettare le sue richieste di interruzione o ignorarle altrimenti;
- Il registro *IPending* contiene un bit per ogni dispositivo I/O, un bit è posto a 1 se il dispositivo ha una interruzione in attesa di risposta;

Accesso ai registri di controllo del processore

A causa della loro particolare struttura non è possibile eseguire nei registri di controllo le istruzioni aritmetiche e logiche.

Per poter operare su di essi è necessario copiare il contenuto di tali registri in un registro generico, esegue le operazioni desiderare ed infine salvare il contenuto del registro generico nel registro di controllo. Per effettuare questi trasferimenti si utilizza un'istruzione speciale:

- *MovControl "destinazione" "sorgente"*: carica il contenuto del registro sorgente nel registro destinazione;
 - Il campo *destinazione o sorgente*, a seconda di cosa si vuole fare, fa riferimento a un registro di controllo.

Eccezioni

Il meccanismo di interruzione capace di bloccare l'esecuzione di un programma in risposta a un particolare evento non è utilizzato unicamente per le operazioni I/O.

Possiamo definire *eccezione* un evento generico che causa un'interruzione, e quindi affermare che le interruzioni I/O non sono altro che *particolari tipi di eccezioni*.

I tipi di eccezioni più comuni che conosciamo sono:

- Eccezioni per il ripristino da errore: lanciate da meccanismi del calcolatore che individuano mal funzionamenti dei componenti fisici, la routine tenta, per quanto possibile, di rimediare all'errore causando l'interruzione dell'istruzione in esecuzione;
- Eccezioni di debugging: particolari eccezioni lanciate dal debugger per interrompere momentaneamente l'esecuzione di un programma;
- Eccezioni del Sistema Operativo: eccezioni dal SO per gestire tutte le attività del calcolatore;

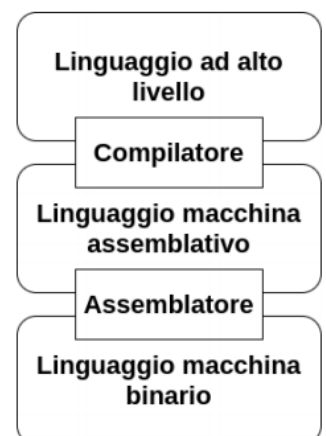
Il livello software

Come si programma?

Il programmatore potrebbe scrivere programmi direttamente in linguaggio assembler (assembly) ma oggi si fa affidamento a linguaggi più ad alto livello che consentono una maggiore espressività.

Una volta scritto il programma del linguaggio ad alto livello viene utilizzato il compilatore relativo al linguaggio che si occupa di tradurre il codice ad alto livello in codice assembler.

Infine il codice assembler viene tradotto in sequenze binarie dall'assemblatore.



Come viene generato un programma oggetto finale

La generazione di un programma oggetto finale avviene attraverso svariati step effettuati da strumenti differenti.

Quando un programmatore scrive un programma, questo può essere scritto:

- Utilizzando un linguaggio ad alto livello, creando dei file sorgenti di linguaggio ad alto livello;
- Utilizzando un linguaggio assembleativo, creando dei file sorgenti assembly;

Nel caso in cui sia stato utilizzato in linguaggio ad alto livello, il primo passo viene svolto dal compilatore che si occupa di trasformare tutti i file sorgenti ad alto livello in file sorgenti assembly.

I sorgenti assembly, risultanti dalla compilazione o scritti dal programmatore, vengono affidati ad un assemblatore che si occuperà di trasformarli in file oggetto temporanei. Alcuni file oggetto potrebbero risultare *incompleti* in quanto i file sorgente da cui derivano potrebbero avere riferimenti ad un sottoprogramma o a un dato scritto in un altro file sorgente (come ad esempio una libreria), si hanno quindi riferimenti esterni. L'assemblatore non è capace di gestire tali riferimenti e lascia e li lascia indefiniti.

Infine, vi è un linker che prende in input i file oggetti generati dall'assemblatore, insieme ad un insieme noto di file oggetti (come ad esempio delle librerie standard), e genera, grazie a una conoscenza globale dei file oggetto, in output un file oggetto unico e completo.

Il processo assembleativo

L'assemblatore si occupa di trasformare i file sorgenti scritti in assembly in un file oggetto, scritto in codice binario.

L'assemblatore genera un file oggetto attraverso un processo di diversi passi:

1. Legge tutte le istruzioni contenute nel file sorgente generando per ognuna la codifica binaria indicandone il codice operativo e gli operandi a seconda del metodo di indirizzamento usato;
2. Riconosce le direttive di assemblatore per l'allocazione di memoria, come *origin* o *dataword*, e inserisce queste informazioni nel header del file oggetto. Tali informazioni saranno successivamente utilizzate da uno strumento denominato loader;
3. Riconosce le direttive che assegnano i nomi a costanti, come EQU, e sostituisce nel codice ogni occorrenza del nome con il valore binario;
4. Sostituisce nel codice ogni occorrenza di etichette con un valore binario relativo;

Assemblaggio a due passi

L'assemblatore man mano che incontra etichette e direttive di assegnamento tiene traccia dei nomi e dei valori binari corrispondenti in una tabella dei simboli.

Può però capitare che di elaborare un'istruzione che fa riferimento ad un'etichetta o a una variabile ancora non presente all'interno della tabella dei simboli (l'assemblatore non lo ha ancora "incontrato").

Per risolvere tale problematica l'assemblaggio viene effettuato a 2 passate:

- Passata 1: Si scorre tutto il codice sorgente al fine di popolare la tabella dei simboli;
- Passata 2: Si scorre tutto il codice sorgente applicando la sostituzione con i valori in tabella;

Al termine si otterrà il codice oggetto finale.

Il loader

I file sorgenti, i file oggetto e i dati risiedono inizialmente all'interno della memoria secondaria. Un programma oggetto e i suoi dati devono essere trasferiti nella memoria primaria per essere eseguiti dal processore.

Il caricamento in memoria primaria avviene su richiesta dell'utente e viene effettuato da un programma specifico denominato *loader* che esegue il seguente passi:

1. Legge le informazioni di allocazione, come dimensione del programma e locazione di caricamento, che l'assemblatore ha inserito nell'header del file oggetto;
2. Sulle base delle informazioni ottenute carica in memoria primarie il programma;
3. Salta all'esecuzione della prima istruzione del programma;

Il linker

La maggior parte delle volte i programmi sono distribuiti su più file sorgenti. Ciò porta l'assemblatore a produrre file oggetto incompleti, con riferimenti indefiniti.

Il *linker* prende in input i file oggetti generati dall'assemblatore e genera in output un file oggetto unico e completo, ciò è possibile poiché ogni file oggetto contiene *una lista dei nomi esterni usati e una lista delle proprie etichette da esportare*:

- Revisionando i file oggetto individua l'uso di nomi esterni e li sostituisce con il corretto valore binario, ottenuto dalla lista delle etichette esportare dei file;

Le librerie

Le librerie sono un insieme di file oggetti contenenti sottoprogrammi utilizzabili da altri programmi, l'obiettivo è quello di scrivere il codice una volta sola e riutilizzarlo il più possibile.

Le librerie vengono create da un programma di utilità detto *archivier*, in particolare aggiunge ai file oggetto informazioni aggiuntive utili al linker per risolvere i nomi esterni in altri programmi.

Quando un programma fa uso di librerie, i file di tali librerie devono essere specificati nell'invocazione del linker per permettergli di includerli nel file oggetto finale.

Il compilatore

Oggigiorno la maggior parte dei programmi è scritta in linguaggi ad alto livello. Il compilatore è il programma che si occupa di trasformare i file sorgenti di un linguaggio ad alto livello in file sorgenti in assembly.

Il compilatore automatizza molti compiti del programmatore assembly (come la gestione delle aree di attivazione dei programmi), inoltre particolari compilatori, detti *ottimizzanti*, attuano una riorganizzazione, secondo una logica interna, delle istruzioni al fine di ottimizzare il codice.

Il debugger (Trace Mode e Breakpoints)

Il compilatore è in grado di rilevare errori sintattici nel codice, ovvero errori che non rendono possibile completare il processo di compilazione. Tali errori rientrano nella categoria *compile-time error*.

Il compilatore non può rilevare errori logici, come cicli infiniti, o bug, comportamenti indesiderati, i quali rientrano nella categoria *runtime error*.

Il debugger è il programma che ci permette analizzare l'esecuzione di un programma passo per passo: ci permette di mettere in pausa il programma in un qualsiasi momento e ci fornisce informazioni come l'istruzione in esecuzione in quel momento e il contenuto dei registri.

Il debugger può essere eseguito in due diverse modalità, entrambe basate sull'utilizzo delle interruzioni:

- *Trace mode*: il programma viene eseguito passo per passo, interrompendosi ad ogni istruzione:
 1. *Al termine di ogni istruzione viene generata un'eccezione;*
 2. *Il debugger viene lanciato come routine di servizio dell'istruzione;*
 3. *Il programmatore supervisiona le informazioni fornite dal debugger, prendendo nota di eventuali problemi/correzioni;*
 4. *Una volta ultimata l'analisi il programmatore seleziona un comando di "ripresa dell'esecuzione" che effettuerà il rientro dell'interruzione con conseguente esecuzione dell'istruzione successiva;*
- *Breakpoint*: vengono selezionati dal programmatore dei punti specifici in cui interrompere il programma:
 1. Con il debugger in esecuzione, il programmatore sceglie i punti (breakpoint) in cui vuole interrompere il programma;
 2. Il debugger sostituisce e mette da parte le istruzioni corrispondenti ai breakpoint con delle interruzioni software speciali, denominate trap;
 3. L'esecuzione del programma prosegue normalmente fino al raggiungimento di una di *trap*, la quale passerà l'esecuzione al debugger;
 4. *Il programmatore supervisiona le informazioni fornite dal debugger, prendendo nota di eventuali problemi/correzioni;*
 5. *Una volta ultimata l'analisi il programmatore seleziona un comando di "ripresa dell'esecuzione" che effettuerà il rientro dell'interruzione: il debugger reinserisce nel codice l'istruzione precedentemente rimossa e non appena quest'ultima sarà eseguita dal programma il debugger effettuerà nuovamente la sostituzione, la trap potrebbe trovarsi all'interno di un ciclo e quindi dovrebbe essere possibile rieseguirlo ad ogni iterazione;*

Il sistema operativo

Il *sistema operativo* gestisce il coordinamento generale di tutte le attività del calcolatore, attività come l'esecuzione concorrente dei programmi, la gestione degli I/O, gestione dell'accesso alla memoria e la gestione dell'interfaccia utente.

Il sistema operativo è formato da un insieme di routine essenziali, sempre presenti in memoria principale e pronte all'esecuzione, e da un insieme di programmi di utilità collocati nella memoria secondaria che vengono caricati in memoria primaria quando c'è bisogno di eseguirli, come ad esempio i nostri programmi.

Durante l'inizializzazione del sistema, un processo di avvio denominato *bootstrapping* si occupa di caricare in memoria principale una piccola porzione del sistema operativo che si occuperà di caricare ed eseguire le varie parti del sistema.

Un sistema operativo capace di eseguire più programmi alla volta è definito *multitasking o concorrente*, ma in verità è solo un'illusione in quanto la macchina esegue sempre e comunque un'istruzione alla volta. Il trucco alla base del multitasking è il *time slicing*: l'esecuzione dei programmi viene suddivisa in *quantità di tempo*, un contatore del sistema operativo a ogni tempo *T* lancia attraverso un'interruzione la routine *scheduler* che sceglie un quanto di tempo da eseguire.

Questo sistema di multitasking prevede che i programmi possano trovarsi in 3 possibili stati:

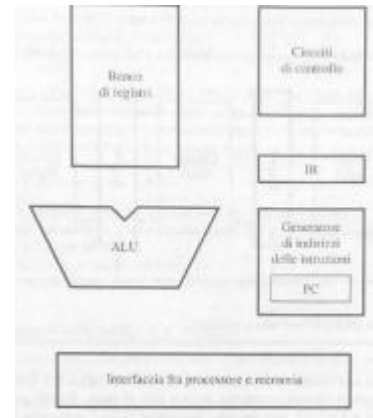
- Running, ovvero è il programma attualmente in esecuzione;
- Runnable, ovvero il programma è pronto e può essere selezionato dallo scheduler;
- Blocked, ovvero il programma è in attesa di un input e quindi non può essere selezionato dallo scheduler;

Struttura base del processore

Struttura in dettaglio del processore

Il processore è composto da svariati parti ognuna aventi una funzione specifica:

- L'*unità aritmetico-logica (ALU)*: esegue le operazioni aritmetiche e logiche necessarie ad eseguire le istruzioni;
- I *circuiti di controllo*: generano i bit di controllo per gestire il funzionamento della CPU;
- Il *banco dei registri*: un blocco di memoria contenente i registri generici della CPU;
- Un *insieme di registro speciali*: come i registri PC e IR che contengono rispettivamente l'indirizzo della prossima istruzione e l'istruzione in esecuzione;
- Il *generatore di indirizzi delle istruzioni*: aggiorna il contenuto di PC;
- L'*interfaccia processore-memoria*: gestisce il trasferimento dei dati tra la memoria e la CPU o viceversa;



Il banco di memoria:

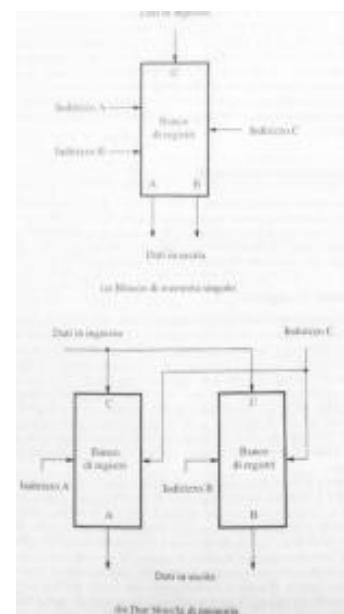
Il banco di registri è un blocco di memoria interno al processore caratterizzato da piccole dimensioni e da una elevata velocità di lettura e scrittura.

Consiste in vari registri con un circuito per l'accesso in scrittura e in lettura, può essere immaginato come un blocco avente 4 ingressi e 2 uscite capace di effettuare la lettura contemporanea di due registri e la scrittura di un singolo registro:

- I due ingressi *indirizzo A* e *indirizzo B* contengono gli indirizzi dei registri che si vuole leggere, il loro contenuto verrà dato in uscita rispettivamente nelle porte di uscita A e B;
- L'ingresso *indirizzo C* contiene l'indirizzo del registro che si vuole scrivere, il contenuto che si vuole memorizzare viene fornito nell'*ingresso dati*;

La *lettura contemporanea* di 2 registri può essere realizzata in due modi diversi:

- Attraverso un singolo blocco di registri con percorso dati e circuiti di accesso duplicati;
- Attraverso due copie del banco di registro, una per il registro A e l'altro per B;



L'unità aritmetico-logica (ALU)

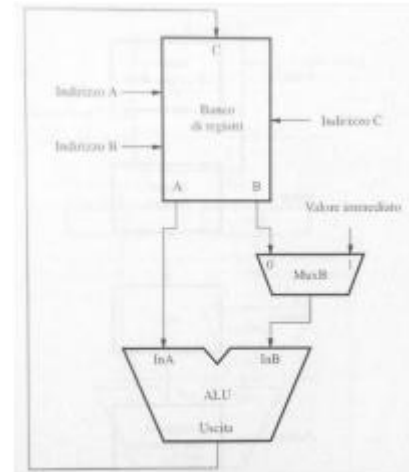
L'ALU è la componente del processore che esegue le operazioni aritmetiche e logiche necessarie a eseguire le istruzioni.

Può essere immaginato come *un blocco avente due ingressi e un'uscita*:

- Gli ingressi *InA* e *InB* rappresentano gli operandi in ingresso;
- L'uscita *out* rappresenta il risultato dell'operazione;

Si può eseguire un collegamento semplificato tra *ALU* e il *banco dei registri* nel seguente modo:

1. Gli ingressi dell'ALU vengono collegati alle uscite del banco dei registri;
 - Viene inserito un multiplexer per poter scegliere se fornire come input all'ALU il valore in uscita dal banco dei registri o un valore immediato;
2. Il risultato in uscita dall'ALU viene usato come *input dati di scrittura* nel banco dei registri;



Generatore di indirizzi delle istruzioni

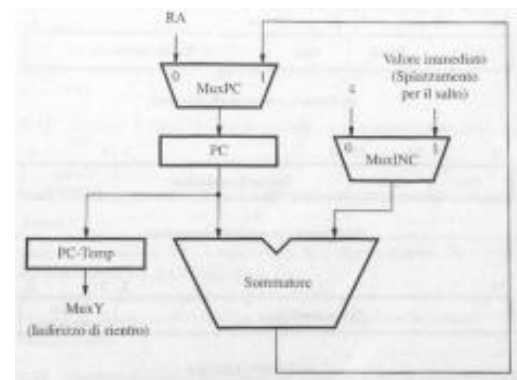
Il *generatore di indirizzi delle istruzioni* è il componente che si occupa di aggiornare il contenuto del *registro speciale PC* per puntare alla prossima *istruzione*.

L'aggiornamento del registro PC può avvenire in diversi modi:

- Passaggio alla prossima istruzione del programma: rappresentato da una somma dell'attuale contenuto di PC e 4;
- Salto a una specifica istruzione: rappresentato da una somma dell'attuale contenuto del PC e uno *spiazzamento*;

Il circuito della componente è quindi costituito da:

- Un sommatore: usato per incrementare il PC di 4 o per lo spiazzamento. Prende in ingresso il contenuto del registro PC e la quantità da sommare;
- Un *multiplatore MuINC*: che decide quale quantità da sommare dare al sommatore;
- Un multiplatore *MuPC*: che decide se salvare in PC il risultato del sommatore o un indirizzo arbitrario (in caso di chiamata al sottoprogramma);
- Un registro temporanea denominato *Temp-PC* usato per salvare il contenuto di PC nel registro LR in caso di chiamata a un sottoprogramma;



Fasi dell'esecuzione di un'istruzione

L'esecuzione delle istruzioni da parte del processore può essere suddivisa in due fasi, ognuna costituita da dei specifici passi:

- La prima fase è detta *fase di prelievo (fetch phase)*, in cui:
 1. Viene prelevata dalla memoria l'istruzione puntata dal registro PC e viene memorizzata all'interno del registro IR;
 2. Viene incrementato di 4 il registro PC per puntare alla prossima istruzione;
- La seconda fase è detta *fase di esecuzione (execution phase)*, in cui:
 1. Viene decodificata ed eseguita l'istruzione prelevata;

Organizzazione a 5 stadi dell'istruzioni RISC

Durante la fase di esecuzione generalmente le istruzioni prevedono operazione come: lettura/scrittura da/in una locazione di memoria, lettura da registri, esecuzione di operazioni aritmetiche e logiche, etc.

È possibile suddividere l'esecuzione delle istruzioni in 5 stadi distinti, ognuno rappresentante una specifica azione (uno stadio può essere saltato se l'istruzione non prevede quella specifica azione):

- **Stadio 1 (Prelievo):** Prelievo dell'istruzione e incremento del PC;
- **Stadio 2 (Decodifica):** Decodifica dell'istruzione e lettura dei registri dal banco dei registri;
- **Stadio 3 (Elaborazione):** Esecuzione dell'operazione nell'ALU;
- **Stadio 4 (Memoria):** Lettura o scrittura da/in una locazione della memoria;
- **Stadio 5 (Scrittura):** Scrittura del risultato nel registro di destinazione;



L'organizzazione in stadi ha favorito la suddivisione del circuito combinatorio, che costituisce il calcolatore, in circuiti più semplici connessi in cascata (uno per stadio), con la presenza di registri temporanei fra uno stadio e l'altro.

Datapath (Percorso dei dati)

Con il termine *datapath* si intende il *percorso* che i dati seguono durante l'esecuzione di un'istruzione. L'illustrazione del percorso dei dati viene fatta attraverso il seguente circuito e lo svolgimento dei vari stadi, dal 2 in poi:

- **Stadio 2:**
 - Le porte di uscita A e B del banco dei registri vengono copiate all'interno dei registri temporanei RA ed RB;

- **Stadio 3:**

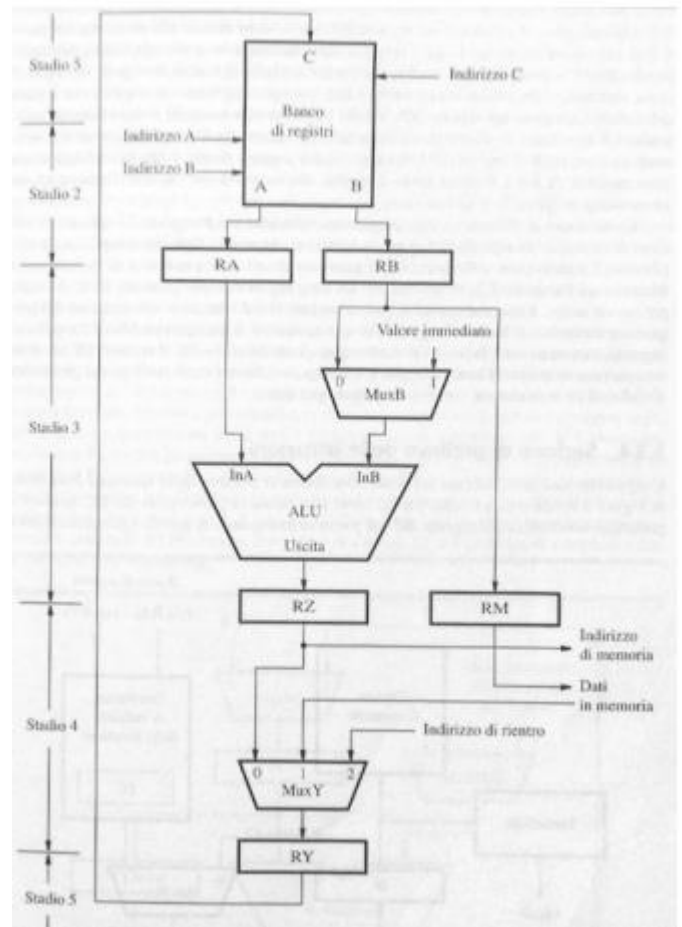
- Il registro RA viene usato come primo ingresso dell'ALU;
- Il multiplexer MuxB decide se dare il registro RB o un valore immediato come secondo ingresso dell'ALU;
- Il risultato dell'operazione eseguita dall'ALU viene salvato nel registro temporaneo RZ;
- Il contenuto di RB viene salvato nel registro temporaneo RM;

- **Stadio 4:**

- In caso di un'istruzione di memorizzazione, il contenuto di RM viene salvato in memoria;
- In caso di un'istruzione di caricamento dalla memoria o salvataggio in una locazione di memoria, il contenuto di RZ viene mandato all'interfaccia processore-memoria;
- Il multiplexer MuxY decide se salvare in RY il risultato dell'operazione eseguita dall'ALU, i dati caricati dalla memoria o l'indirizzo di rientro da sottoprogramma;

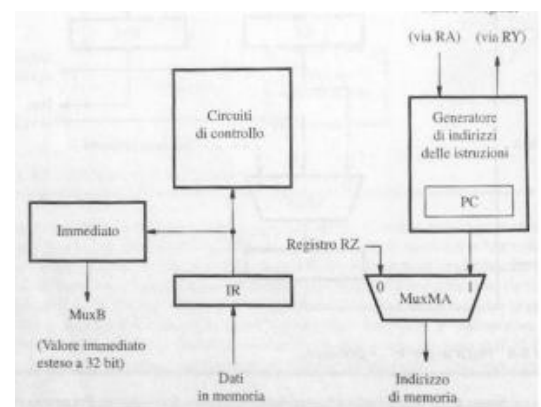
- **Stadio 5:**

- Il contenuto del registro temporaneo RY viene salvato nel banco dei registri;



Lo **stadio 1** viene eseguito da un circuito differente:

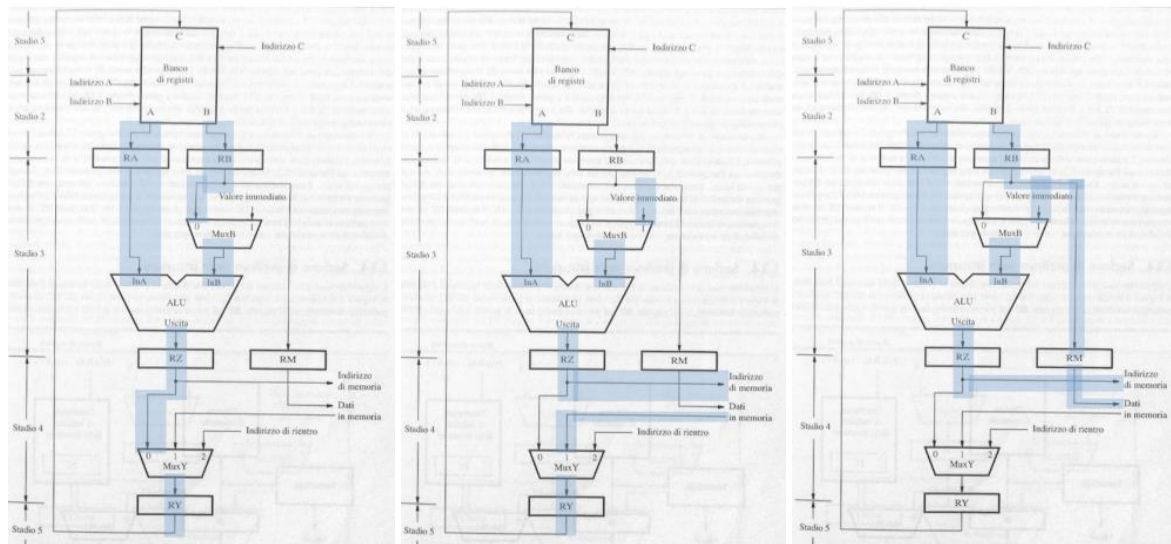
- In caso di prelievo di un'istruzione dalla memoria, il multiplexer MuxMA decide se dare il registro RZ o il registro PC all'interfaccia processore-memoria;
- L'istruzione ottenuta viene caricata nel registro IR;
- I circuiti di controllo si occupano di decodificare l'istruzione e fornisce l'eventuale valore *immediato* al multiplexer MuxB;



Alcune datapath

Seguono i datapath delle seguenti operazioni:

- *Add R3, R4, R5*
- *Load R3, 10(R5)*
- *Store R3, 10(R7)*



I datapath di istruzioni come il branch, call e return si basano sull'utilizzo del circuito sopracitato insieme al generatore di indirizzi.

Attesa di memoria (MFC)

Con l'organizzazione a stadi del RISC si è ipotizzato che ogni stadio possa essere eseguito in un ciclo di clock.

Ciò non è possibile poiché gli accessi alla memoria non possono essere sempre eseguiti in un ciclo di clock, se il dato da prelevare non si trova in cache è necessario che l'esecuzione del passo corrente si blocchi fino al completamento dell'operazione di memoria richiesta.

Gli stadi interessati da questa problematica sono lo *stadio 1*, poiché deve prelevare l'istruzione puntata dal PC dalla memoria, e lo *stadio 4*, poiché si potrebbe eseguire un'istruzione di Load o Store.

Una soluzione a tale problema è ottenuta utilizzando il segnale *MFC (Memory Function complete)*, che viene lanciato al completamento dell'operazione di memoria. Il circuito di controllo interrompe l'esecuzione dell'istruzione finché il MFC non viene posto a 1.

Segnali di controllo

Per poter eseguire le istruzioni il processore deve generare le sequenze di segnali di controllo per ogni stadio.

I segnali di controllo costituiscono in:

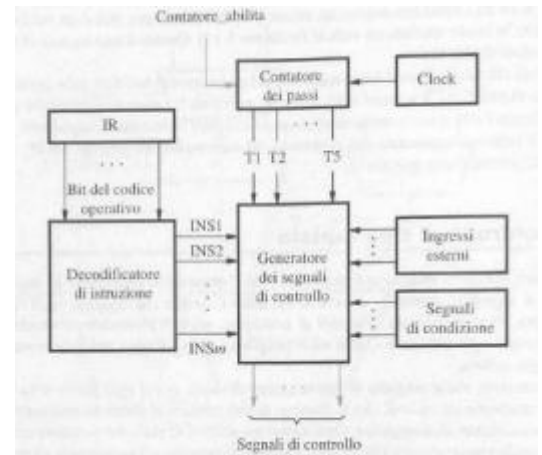
- Segnali di selezione per i multiplatori;
- Segnali di attivazione di alcuni registri;
- Segnali di condizione;
- Segnali per la gestione della memoria;
- Operazioni da eseguire nell'ALU;
- Informazioni lette dal registro IR;

Controllo Cablato

Il controllo cablato è una tecnica di generazione di segnali di controllo, tipica dei sistemi RISC.

Il controllo cablato consiste in:

- Un *contatore modulo 5* che scandisce gli stadi di esecuzione;
- Un *decodificatore di istruzione* che genera un vettore lungo quanto il numero di istruzioni presenti e pone a 1 solo il bit corrispondente all'istruzione attualmente in esecuzione;
- Un *generatore dei segnali di controllo*, che grazie al decodificatore sa quale istruzione è attualmente in esecuzione e grazie al contatore sa in quale stadio si trova l'esecuzione, produce i segnali di controllo sulla base dei segnali di condizione e ingressi esterni che riceve;



Come già detto alcuni stadi potrebbe durare più di un ciclo di clock a causa dei ritardi dovuti all'attesa della memoria. Per tale motivo il contatore possiede un ingresso di controllo che abilita o meno momentaneamente il contatore per bloccare passaggio allo stadio successivo. Tale segnale di controllo è definito come:

- $Contatore_{abilita} = \overline{WMFC} + MFC$:
 - \overline{WMFC} viene posto a 1 quando inizia un'operazione di memoria;
 - MFC viene posto a 1 quando si conclude un'operazione di memoria;

Organizzazione di un processore CISC

Con le istruzioni CISC non è possibile eseguire un approccio a stadi come per quelle RISC, poiché diverse istruzioni richiedono un numero di passi diverso per essere eseguite.

Il processore CISC è quindi caratterizzato da:

- Un blocco di *interconnessione* che permette ad ogni componente del processore di comunicare con gli altri componenti;
 - Tale blocco è realizzato tramite BUS;
- Un insieme di *registri temporanei* in cui vengono memorizzati i risultati intermedi, tali registri sostituiscono i *registri interstadio* dell'approccio a stadi;

Interconnessione via BUS

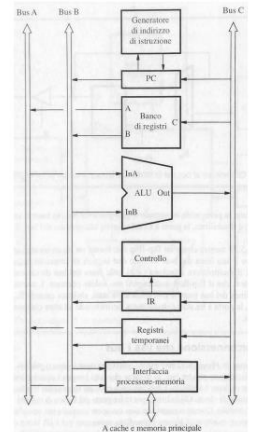
Un *bus* è formato da un insieme di linee a cui sono connessi i vari dispositivi, in cui:

- Per evitare ambiguità, un solo dispositivo alla volta può inviare dati;
- La porta logica che invia un segnale su una linea del bus viene denominato *bus driver*;
- I dispositivi sono collegati al bus attraverso porte *tri-state*, tutte disattivate ad eccezione del *bus driver*;

Il datapath CISC

Il datapath CISC può essere così immaginato:

- Vengono utilizzati 3 bus: due come sorgenti *A* e *B* delle operazioni e uno come destinazione *C* per i risultati;
- Tutte le componenti del processore sono collegate ai 3 bus: le *sorgenti A e B* vengono usate come ingressi per l'ALU o per l'interfaccia processore-memoria;
- Il generatore di indirizzi delle istruzioni è collegato direttamente al registro *PC*;
- Il blocco di controllo legge direttamente il registro *IR*;

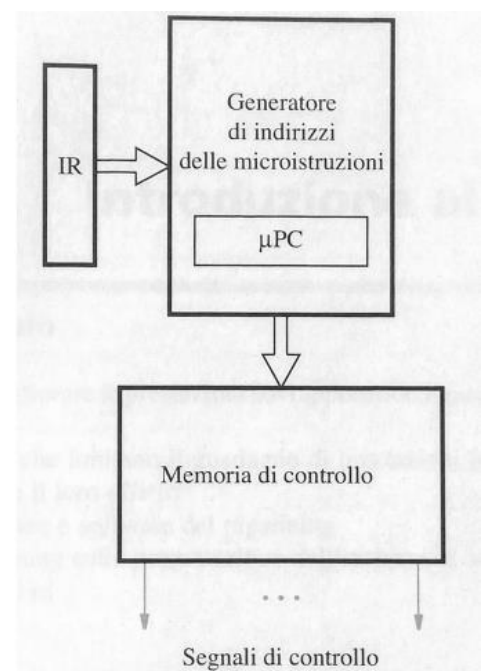


Controllo microprogrammato

Il *controllo microprogrammato* è una tecnica di gestione dei segnali di controllo tipica delle architetture CISC.

Tale tecnica consiste nel:

- Raccogliere *i segnali di controllo* di ogni passo in una parola di memoria chiamato *microistruzione*;
- L'insieme delle *microistruzioni* rappresentanti di un passo dell'istruzione viene chiamato *microroutine*;
 - *Completata una microroutine, si passa a quella successiva corrispondente al passo successivo*;
- Le *microistruzioni di ogni microroutine* vengono memorizzate in locazione consecutive della memoria di controllo;
- Un registro simile al PC denominato *microPC* contiene l'indirizzo della prossima microistruzione da caricare;
- All'inizio di un'istruzione *macchina*, il *generatore di indirizzi delle microistruzioni* carica nel registro *microPC* l'indirizzo della prima microistruzione della microroutine corrispondente;



Questa tecnica risulta essere più lenta rispetto all'approccio

cablato del RISC poiché vengono usati svariati clock per l'inizializzazione di tale sistema.

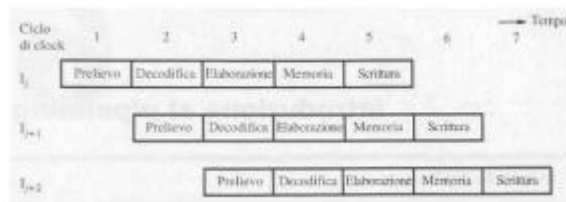
Pipelining

La tecnica di pipeline

Il pipelining permette di parallelizzare l'esecuzione delle istruzioni nelle architetture RISC basandosi sull'organizzazione a 5 stadi.

L'idea generale dell'esecuzione parallela è eseguire più istruzioni eseguendone i diversi stadi una dopo l'altro, come in una catena di montaggio.

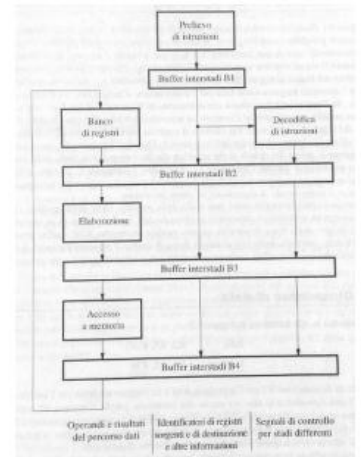
Nel caso ottimale, in cui ogni stadio impiega un ciclo di clock, si hanno 5 istruzioni eseguite in parallelo.



Per poter gestire l'esecuzione in pipeline di più istruzioni è necessario mantenere delle informazioni tra uno stadio e l'altro, tali informazioni vengono mantenute nei *buffer interstadi che contengono*:

- I registri *interstadio* (RA, RB, PC_Temp , ect..)
- Il registro IR per mantenere gli identificatori dei registri sorgente e destinazione;
- I segnali di controllo per i vari stadi

La gestione dei segnali di controllo avviene in modo diverso rispetto al controllo cablato, durante la fase di decodifica vengono generati tutti i segnali per i vari stadi e vengono eseguiti e trasportati man mano che l'istruzione avanza nella pipeline.



I problemi del pipelining

Non è sempre possibile avere il caso ottimale in cui vengono eseguite 5 istruzioni in parallelo.

Durante l'esecuzione parallela delle istruzioni potrebbero accadere dei conflitti che ritardano l'ingresso di nuove istruzioni nella pipeline, i quali sono:

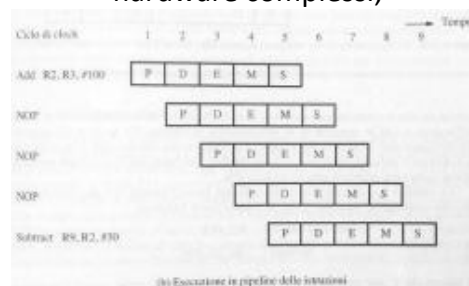
- *Dipendenze di dato*;
- *Ritardi nell'accesso alla memoria*;
- *Ritardi nei salti*;
- *Limite di risorse*;

Conflitto: Dipendenze di dato e soluzioni

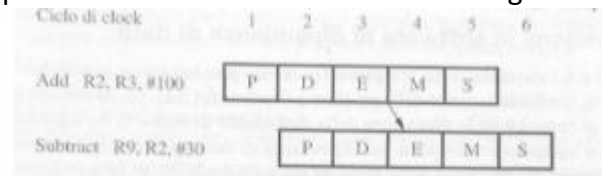
Si ha la *dipendenza di dato* quando un'istruzione usa un registro sorgente che non è ancora stato aggiornato dalle istruzioni precedenti, ovvero il registro sorgente è il registro di destinazione di una istruzione precedente che non ha ancora raggiunto lo stadio di scrittura.

La soluzione consiste nel mettere in stallo l'esecuzione dell'istruzione fino a quando il registro non viene aggiornato. Per tale scopo vi è un approccio software o hardware:

- L'approccio software consiste nell'inserire delle *istruzioni nulle (NOP)* tra le istruzioni in conflitto, tali istruzioni creano un ciclo di inattività denominato *bolla*. Queste istruzioni nulle possono essere inserite da un compilatore ottimizzante o da circuiti hardware complessi;



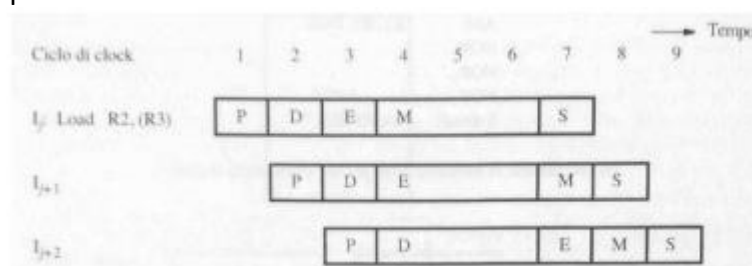
- L'approccio hardware consiste nell'*inoltrare degli operandi*, ovvero i registri interstadio successivi vengono inoltrati a stadi precedenti. A livello circuitale consiste nel mandare gli operandi come RZ o RY come ulteriori ingressi dei multiplexer.



Conflitto: Ritardi della memoria e soluzione

Si ha la *ritardo della memoria* quando un'istruzione fa un accesso alla memoria che, qualora il dato cercato non sia in cache, può richiedere svariati clock.

Una soluzione per tale problematica consiste nel ritardare tutte le istruzioni successive dello stesso numero di passi.



Conflitto: Ritardo nei salti e soluzioni

Durante un'istruzione di salto incondizionato, o condizionato qualora la condizione risulti vera, l'indirizzo di destinazione viene caricato nel registro PC nel corso dello *stadio 3 di esecuzione*. Ciò comporta lo scarto delle due istruzioni successive *rispettivamente in stadio 2 di decodifica e stadio 1 di prelievo*, producendo un ritardo di 2 cicli clock.

Per poter risolvere tale problematica vi sono diverse soluzioni aventi risultati diversi:

- Anticipare la valutazione del salto: questa soluzione permette di ridurre la penalità di salto ad un solo ciclo di clock e consiste nel modificare l'hardware per riuscire a valutare ed eseguire un salto nello stadio 2 di decodifica:
 - Si aggiunge il circuito di un sommatore, per calcolare la destinazione, e di un comparatore, per valutare la condizione, allo *stadio 2 di decodifica*;
- *Salto differito*: questa soluzione ci permette di eliminare o ridurre al minimo la penalità del salto e consiste nell'eseguire comunque le istruzioni successive al salto anziché scartarle: il trucco è organizzare l'ordine di esecuzione delle istruzioni, spesso fatto dal compilatore, in modo che le istruzioni successive a un salto debbano essere eseguite comunque in caso di salto o meno;
- *Predizione dei salti*: questa soluzione tenta attraverso una analisi statistica di ridurre a 1 la penalità di salto, si suddivide in due varianti:
 - *Predizione Statica*: è il metodo più semplice e consiste nel supporre che un salto condizionato sia effettuato sempre, ovvero prelevare sempre l'istruzione destinazione, o meno, ovvero prelevare l'istruzione successiva del programma. La base statistica consiste nel fatto che i salti all'indietro a fine ciclo sono più probabili rispetto ai salti in avanti. Il comportamento da

applicare può essere determinato dal segno dello spiazzamento oppure includendo un bit di predizione *nell'istruzione macchina*;

- *Predizione dinamica: questa soluzione si basa sull'attuale comportamento di salto per influenzare la predizione, si utilizzano macchine a 2 o 4 stati per indicare la probabilità di un salto.*

Per le soluzioni basate sulla predizione è necessario saper prelevare l'indirizzo di destinazione nella fase di prelievo del salto, per fare ciò utilizza una memoria denominata *buffer di destinazione del salto*.

Questo buffer contiene una tabella che indicizza tutte le istruzioni di salto del programma con le seguenti informazioni:

- *Indirizzo dell'istruzione di salto;*
- *Uno o due bit per l'algoritmo di predizione;*
- *Indirizzo di destinazione del salto;*

Una volta prelevata un'istruzione il suo indirizzo verrà cercato all'interno della tabella e se risulta essere un'istruzione di salto il registro PC verrà aggiornato secondo le informazioni contenute in tabella.

Conflitto: Limite di risorse e soluzione

La pipeline può andare in stallo quando una risorsa hardware è richiesta da più istruzioni contemporaneamente. Un esempio può essere la memoria che può essere richiesta in contemporanea dallo *stadio 1 di prelievo, per prelevare le istruzioni*, e dallo *stadio 4 di memoria, per caricare o salvare dati*.

Una soluzione semplice e ampiamente adottata consiste nell'avere *cache separate per istruzioni e dati*.

Valutazione della prestazione ed effetti dei conflitti

Per un processore *pipeline* definiamo: (N = N° istruzioni, S = Cicli di clock per istruzione e R = frequenza di clock del processore)

- *Tempo di esecuzione T come:* $T = \frac{N \cdot S}{R}$
- *Throughput, numero di istruzioni al secondo come:* $P_{NP} = \frac{R}{S}$

Per un processore *con pipeline*, ovvero con caso ottimo in cui ogni istruzione impiega 1 clock:

- *Throughput, numero di istruzione al secondo come:* $P_P = \frac{R}{1} = R$

Il verificarsi di conflitti durante la pipeline fa aumentare il *numero medio di cicli di clock per istruzione S* . Ogni tipologia di conflitto indipendente aumenta S di un delta δ dato dalla percentuale di occorrenza del conflitto p e il numero medio di cicli di stallo introdotti c :

- *Conflitto di dipendenza di dato:* $\delta_{dato} = p_{dato} * c_{dato}$
- *Conflitto di salto:* $\delta_{salto} = p_{salto} * c_{salto}$
- *Conflitto di ritardo memoria (cache miss):* $\delta_{miss} = p_{miss} * c_{miss}$

Il throughput deve tener conto dei ritardi introdotti dai conflitti:

$$P_P = R / (S + \delta_{dato} + \delta_{salto} + \delta_{miss})$$

Processori superscalari

I processori superscalari costituiscono un ulteriore metodo di parallelismo. L'idea alla base è quella di avere più unità di esecuzione al fine di eseguire in parallelo più istruzioni, sfruttando così l'emissione multipla.

Le unità di esecuzione possono essere unità dedicate all'aritmetica o unità dedicate all'accesso alla memoria.

I processori superscalari sono caratterizzati dalle seguenti componenti:

- *L'unità di prelievo*: preleva più istruzione alla volta e le mette in coda in un buffer denominato *Coda di istruzioni*;
- *L'unità di smistamento*: smista le istruzioni presenti in coda e le distribuisce alle unità di esecuzione corrispondenti mettendole nel buffer dell'unità di esecuzione denominato *Stazione di Prenotazione*;
 - *L'unità di smistamento previene i deadlock, istruzioni bloccate a causa di dipendenze reciproche*;
 - *Prima che l'unità di smistamento possa inviare un'istruzione deve verificare che:*
 - *Siano disponibili e vengano prenotati i registri temporanei per i risultati*;
 - *Vi sia spazio a disposizione nella stazione di prenotazione dell'unità di esecuzione desiderata*;
 - *Vi sia una locazione disponibile nel buffer di riordino*;
- *Varie unità di esecuzione*: eseguono l'operazione per cui sono specializzate e salvano i risultati nei *registri temporanei*;
- *L'unità di commitment*: organizza i risultati dell'unità di esecuzione nell'ordine di prelievo delle istruzioni attraverso il *buffer di riordino*;

Dipendenze di dato e le stazioni di prenotazione

Può avvenire che due istruzioni dipendenti vengano inviate a due unità di esecuzione differenti e non se ne possa garantire l'ordine di esecuzione. In questo caso bisogna bloccare l'esecuzione dell'istruzione con dipendenze di dato finché l'altra istruzione non sia stata eseguita.

Le Stazioni di prenotazione contengono:

- L'istruzione smistata in attesa di esecuzione;
- Informazioni e operandi rilevanti a ciascuna istruzione smistata;

Appena i risultati che creano dipendenze vengono calcolati sono inoltrati alle stazioni di prenotazione.

Un'istruzione viene mandata in esecuzione solo quando tutti i suoi operandi sono disponibili.

I ritardi e il buffer di riordino

A causa dei ritardi causati *dalla memoria o dai salti* potrebbero avvenire esecuzioni di istruzioni fuori ordine, ovvero un'istruzione che non doveva essere eseguita modifica i contenuti di registri o locazioni di memoria.

Per risolvere tale problematica i risultati delle unità di esecuzione vengono salvati in registri temporanei che assumono il ruolo di *registri permanenti*. L'*unità di commitment* si occupa di organizzare i risultati dell'unità di esecuzione nell'ordine di prelievo delle istruzioni attraverso il *buffer di riordino*.

Funzionamento dei dispositivi I/O

Il bus di sistema (bus dati, bus indirizzi e bus controllo, ruoli)

Il *bus di sistema* è la rete di interconnessione tra i vari componenti del calcolatore. Il bus è formato da fasci di linee capaci di trasportare *un solo bit*, tali linee si suddividono in 3 gruppi:

- *Dati*: contenente i dati trasmessi;
- *Indirizzi*: contenente l'indirizzo dei registri dell'unità slave;
- *Controllo*: contenente i *bit di controllo* relativi alle operazioni sul bus, *come scrittura o lettura, e bit di sincronizzazione*;

Tutte le unità sono connesse al bus tramite delle porte denominate *bus driver*, implementate con *porte tri-state* che vengono attivate solo quando avviene un trasferimento, poiché per evitare ambiguità solo un'unità alla volta può trasferire dati sul bus.

Le unità coinvolte in un trasferimento di dati assumono due ruoli distinti:

- L'unità master: che richiede il trasferimento, che può essere lettura o scrittura;
- L'unità slave: che risponde alla richiesta;

Il trasferimento dei dati può avvenire in modo sincrono, attraverso un bus sincrono, o in modo asincrono, attraverso bus asincrono;

Bus sincrono

In un *bus sincrono* un *segnale di clock* viene distribuito contemporaneamente tramite a una linea del bus a tutte le unità collegate: ciò permette di sincronizzare il funzionamento di tutte le unità.

Il clock è un'onda quadrata, mentre un ciclo di clock è il periodo fra i *fronti di salita o discesa*. L'operazione di trasferimento è divisa in *cicli di bus*, i quali *possono durare uno o più cicli di clock*.

L'operazione di trasferimento è soggetta a ritardi causati dal tempo impiegato dai segnali a percorrere i circuiti:

- Ritardi dovuti al tempo necessario a produrre e immettere i segnali sul bus;
- Ritardi dovuti al tempo necessario ai segnali e ai dati per percorrere il bus;

Bus asincrono

In un *bus asincrono* il sincronismo fra le unità viene ottenuto attraverso un processo di *handshaking*: vi è un segnale di controllo per indicare lo *stato pronto* sia per l'unità master sia per l'unità slave.

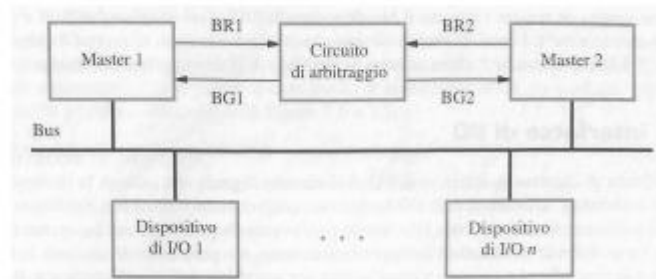
La trasmissione asincrona è affetta da un ritardo nell'attivazione del segnale *master-pronto* dovuto allo sfasamento temporale tra unità slave;

Arbitraggio

In un sistema complesso potrebbe succedere che più dispositivi *master* tentino di accedere al *bus contemporaneamente*, ma come già detto solo un dispositivo alla volta può usarlo. Per gestire tale problematica viene utilizzato il *circuito di arbitraggio* che regola l'ordine di accesso al bus attraverso un sistema di priorità:

- Le *unità master* inviano una *richiesta di accesso* al circuito di arbitraggio tramite i segnali *BR (Bus Request)*;

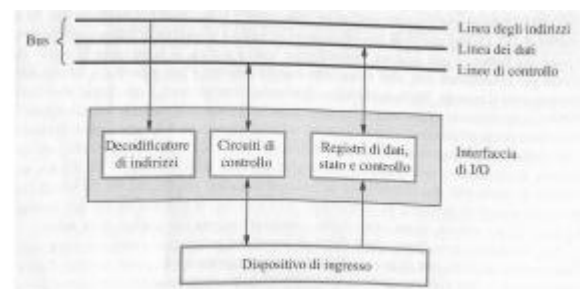
- Il *circuito di arbitraggio* concede l'accesso al bus secondo un ordine di priorità attraverso i segnali *BG (Bus Granted)*;



La struttura delle interfacce I/O

Come già detto, i dispositivi I/O sono collegati al bus attraverso *le interfacce I/O, chiamate anche porte*, le quali sono caratterizzate da:

- *Decodificatore di indirizzi*: si occupa di vedere se l'indirizzo richiesto è il suo;
- *Circuiti di controllo*: per sapere che tipo di operazione eseguire o, in caso di trasferimento, cambiare lo stato dei *bit di controllo*;
- *Registri di dati, di stato e di controllo*: contenenti i bit necessari per gestire le richieste;



L'*interfaccia* presenta due lati:

- *Lato bus*: che comunica con il bus di sistema;
- *Lato periferica*: che comunica con la periferica;
-

Le *interfacce I/O* si possono suddividere in *porte parallele* e *porte seriali*.

Le porte parallele

Le *porte parallele* presentano collegamenti paralleli sia *nel lato bus* sia *nel lato periferica*. Ciò implica che tutti i trasferimenti avvengono in parallelo: vengono inviati o ricevuti tutti i bit contemporaneamente.

Le porte seriali (trasmissione sincrona e asincrona)

Le *porte parallele* presentano un collegamento parallelo nel *lato bus*, il quale si comporta allo stesso modo delle porte parallele, e un collegamento seriale nel *lato periferica*, cioè il dispositivo invia o legge un bit alla volta.

Il *lato periferica* deve quindi gestire una conversione *seriale-parallelo*, in caso di lettura, e una conversione *parallelo-seriale*, in caso di scrittura. La conversione viene effettuata attraverso un registro a scorrimento: il registro a scorrimento permette una lettura/scrittura in parallelo da/verso i registri di buffer. La combinazione di registri a scorrimento e registri paralleli costituisce un parallelismo, il quale permette un flusso costante di dati.

Una delle principali problematiche delle porte seriali è la necessità di sincronizzare la frequenza di scorrimento dei registri con il clock delle periferiche collegate. Vi sono due soluzioni: *la trasmissione asincrona* e *la trasmissione sincrona*.

Nella *trasmissione asincrona* vengono aggiunti dei bit ad inizio e fine della sequenza dato per definirne i limiti:

- Un segnale di 1 indica lo *stato di inattività*;
- Un bit 0 viene aggiunto in testa alla sequenza di bit da inviare;
- Uno o due bit a 1 vengono aggiunti in coda alla sequenza da inviare;
- Il clock del ricevente risulta più veloce del clock del trasmittente ma con l'utilizzo di un *contatore modulato sulla differenza di velocità* si riesce a campionare il dato in corrispondenza del centro di ciascun bit;
 - Il campionamento al centro permette di essere meno sensibili a possibili sfasamenti nella trasmissione;

Nella *trasmissione sincrona* il clock del ricevente viene sincronizzato con il clock del trasmittente osservando i tempi di transizione da 0 e 1 e viceversa nei dati ricevuti. Per garantire un numero sufficiente di transizioni necessari al mantenimento della sincronizzazione vengono usati schemi di codifica particolari.

I dati codificati vengono trasmessi in grandi blocchi, marcati da codici di inizio e fine blocco.

Standard di trasmissione

Le interfacce I/O sono realizzate seguendo degli *standard specifici* al fine di garantire l'usabilità dei dispositivi su tutti i calcolatori, rimuovendo la dipendenza da una specifica realizzazione hardware.

Tali standard sono realizzati da aziende, come Intel e lo standard Thunderbolt, o organizzazioni come IEEE.

Lo standard USB

L'*Universal-Serial-Bus* è uno standard di interconnessione largamente usato da molte periferiche oggi in commercio. Nel corso degli anni sono state realizzate diverse versioni con sempre più funzionalità e velocità.

Le caratteristiche principali dell'USB sono:

- Trasmissione seriale dei dati: viene inviato un bit alla volta;
- Connessioni punto-punto: ogni nodo è collegato direttamente a un altro nodo;
- Struttura ad albero:
 - Le *foglie rappresentano i dispositivi*;
 - I *nodi* rappresentano gli *hub*, ovvero *dei punti di trasferimento intermedi*;
 - La radice dell'albero è denominato *hub radice*;
- *Trasmissione dati tramite Polling* (interrogazione periodica da parte del processore);
- L'*hub radice* appare come un singolo dispositivo al processore, quando quest'ultimo vuole comunicare con una specifica periferica comunica con l'hub radice che si occuperà di fare da tramite;

Fisicamente, una connessione USB è costituita da 4 linee:

- 2 linee di alimentazione (massa e +5V): piccoli dispositivi possono essere alimentati direttamente dall'USB;
- 2 linee dati: usate per la trasmissione dei bit. Possono essere usate in due configurazioni:
 - *Single-ended*: questa configurazione permette una velocità di trasferimento bassa, ogni linea rappresenta uno stato, 0 o 1, e la trasmissione di un bit avviene mandando nella linea desiderata un voltaggio alto rispetto alla massa.

Questa configurazione è sensibile al rumore poiché un aliasing può produrre delle informazioni corrotte;

- *Segnalazione differenziale*: questa configurazione permette una velocità di trasferimento alta, le due linee sono intrecciate e la trasmissione di un bit avviene valutando la differenza di voltaggio fra le linee. Questa configurazione è più robusta al rumore poiché le linee variano insieme lasciando inalterata la differenza;

Una delle funzionalità più famose dell'USB è il *plug-and-play*: è possibile collegare una periferica in un qualsiasi momento ed essere capaci di utilizzarla.

Il sistema operativo per utilizzare una periferica deve conoscere *il tipo di dispositivo* e i *suoi indirizzi*, tali informazioni sono contenute all'interno di una memoria speciale del dispositivo che verrà letta al momento del collegamento. Per tale motivo il processore interroga periodicamente l'hub radice per sapere se vi sono nuove periferiche connesse e in caso iniziarla.

Firewire

Il *firewire* è uno standard di connessione pensato per i dispositivi audio e video, oggi giorno soppiantato dall'USB. Le principali caratteristiche sono:

- Collegamenti seriali punto a punto differenziali;
- Struttura del collegamento a festone, i collegamenti fra dispositivi creano una catena;
- Ottimizzato per il traffico isocrono, traffico multimediale, ad alta velocità;
- Trasferimenti *peer-to-peer* tra dispositivi: i dati possono essere trasmessi fra i dispositivi senza passare per il processore;
- Fisicamente è costituito da 6 piedini:
 - Due coppie per i dati;
 - Due linee di alimentazione;
 - Ciò permette di alimentare dispositivi di medie dimensioni;

Bus PCI e PCIe

Lo standard PCI è spesso incorporato all'interno dello schema madre del calcolatore.

Consiste in un bus denominato *bus PCI* a cui si collegano i dispositivi, attraverso altri standard, connesso direttamente al processore e alla memoria attraverso un controller denominato *Bridge PCI*.

Un'operazione PCI di trasferimento dati completata viene detta *transazione*, e in essa si hanno i seguenti ruoli:

- Il *master* del bus viene detto *iniziatore*;
- Lo *slave* del bus viene detto *obbiiettivo*;

Lo standard PCIe (Express) nasce dalla necessità di maggiore velocità di trasferimento (come le GPU). Lo standard usa collegamenti seriali punto-punto in cui il *Complesso Radice* ha un collegamento diretto con memoria e processore.

Al complesso vengono collegati nodi di commutazione per ottenere una struttura ad albero, è possibile collegare interfacce *ponte* per gli altri standard.

Un collegamento PCIe base è costituita da 2 coppie di linee intrecciate dalla frequenza di 2,5Gbps, inoltre è possibile incrementare la velocità incrementando il numero di coppie.

Bus SCSI e SATA

Gli standard SCSI e SATA sono standard usati principalmente per collegare unità di memorizzazione al calcolatore. Il primo sfrutta un approccio parallelo mentre il secondo un approccio seriale.

Memoria

Gerarchia della memoria

Le unità di memoria sono utilizzate per immagazzinare le informazioni necessarie ad eseguire i programmi. Le informazioni che contengono possono essere costituite da istruzioni, eseguite dal processore, o dati, numeri o caratteri usati come operandi dalle istruzioni.

La memoria si suddivide in due categorie:

- La memoria primaria: è una memoria ad accesso veloce denominata RAM (Random Access Memory) che garantisce una velocità di accesso costante per ogni indirizzo. Ha una capacità limitata ed è volatile, ovvero una volta interrotta l'alimentazione il suo contenuto viene perso. Fisicamente sono costituite da celle di transistor a semiconduttori, dalla capacità di un bit. La memoria centrale è organizzata su dei livelli, più è basso il livello più sono veloci e meno capienti mentre più è alto il livello meno sono veloci e più capienti;
 - La *memoria cache* costituisce il livello più basso, sono memorie velocissime e spesso integrate all'interno del processore. Man mano che vengono caricate informazioni dalla memoria primaria, una copia viene salvata nella cache al fine di velocizzare un successivo richiamo.
- La memoria secondaria: è una memoria lenta caratterizzata da una capacità elevata e dalla persistenza dei dati, non sono volatili. Un esempio di memoria di massa sono i dischi magnetici, i dischi ottici e le memorie flash.

All'interno di un calcolatore è quindi possibile definire la seguente gerarchia a piramide, più sono in alto più sono piccole, veloci e costose:

1. *Registri*: velocissimi e di capacità molto ridotta, integrati all'interno del processore. Basati sulla tecnologia flip-flop;
2. *Livelli di cache (L1, L2)*: molto veloci con dimensione che vanno da decine di KB a qualche MB, sono integrati all'interno del processore e sono costituiti da SRAM (*RAM statica*);
3. *Memoria centrale*: veloce con capacità di GB costituiti da DRAM (*RAM dinamica*);
4. *Memoria secondaria*: lenta con capacità di TB, costituita da dischi magnetici o memorie flash;

Durante l'esecuzione di un programma, un dato spesso risale la piramide dal basso ciò è necessario poiché la memoria secondaria è troppo lenta per la frequenza di lavoro del processore, causando bottleneck.

Il ruolo delle operazioni di memoria nella gerarchia della memoria

È possibile definire le operazioni di memoria come segue:

- L'operazione di Load costituisce il passaggio di un dato al livello superiore;
- L'operazione di Store costituisce il passaggio di un dato al livello inferiore;
- L'operazione di Copia costituire un passaggio orizzontale di un dato nello stesso livello;

Memoria cache (Cache hit e cache miss)

Le cache sono memorie velocissime collocate in una posizione intermedia fra il processore e la memoria primaria. Man mano che vengono caricate informazioni dalla memoria primaria, una copia viene salvata nella cache al fine di velocizzare un successivo richiamo.

Gli algoritmi che gestiscono la cache si basano su dei principi di località:

- *Località temporale* (istruzioni): se un'istruzione è prelevata nel *ciclo i* , con probabilità elevata verrà prelevata nuovamente nel *ciclo $i + p$* (con p piccolo intero positivo);
- *Località spaziale* (dati): se un dato collocato all'indirizzo i viene usato dal processore, con probabilità elevata verrà usato anche il dato collocato all'indirizzo $i \pm q$ (con q piccolo intero positivo);
- *Località spaziale-temporale* (dati e istruzioni): se un blocco di parole va in uso da parte del processore, con probabilità elevata entro breve tempo e per più volte esso verrà usato nuovamente;



Quando la CPU accede alla memoria, il blocco contenente la parola interessata viene caricato nella cache, la quale è formata da spazi grandi quanto i blocchi di memoria e sono denominati *linee di cache*.

Durante l'esecuzione di un programma possono accadere due eventi:

- *Cache hit*: Si ha *cache hit* quando il processore accede a una parola di memoria contenuta in un blocco già presente in cache:
 - *Cache hit in lettura*: Quando il cache hit avviene per un'operazione di lettura, il processore leggerà direttamente il dato dalla cache;
 - *Cache hit in scrittura*: Quando il cache hit avviene per un'operazione di scrittura, vi sono due possibili comportamenti:
 - *Scrittura immediata*: si aggiorna sia la parola in cache sia la parola in memoria centrale. L'obiettivo è mantenere le memorie sincronizzate, pagando però una penalità di spreco di cicli clock;
 - *Scrittura differita*: si aggiorna solo la parola in cache marcandola come modificata attraverso un bit di stato, la modifica si propagherà alla memoria primaria solo quando si libera la posizione corrispondente in cache;
- *Cache miss*: Si ha *cache miss* quando il processore accede a una parola di memoria non presente in cache:
 - *Cache miss in lettura*: Quando il cache miss avviene per un'operazione di lettura, vi sono due possibili comportamenti:
 - *Lettura immediata*: il processore legge la parola appena essa viene caricata in cache senza aspettare che tutto il blocco venga caricato;
 - *Lettura differita*: il processore attende che tutto il blocco venga caricato in cache prima di leggere la parola;
 - *Cache miss in scrittura*: Quando il cache miss avviene per un'operazione di scrittura, vi sono due possibili comportamenti:
 - *Scrittura immediata*: la parola viene modificata direttamente in memoria centrale evitando le attese al processore;
 - *Scrittura differita*: il processore attende che il blocco sia caricato in cache e poi modifica la parola marcandola come modificata attraverso un bit di stato;

Schemi di indirizzamento

Il numero di posizioni di una cache è assai inferiore rispetto al numero di blocchi in memoria, vengono usati perciò degli *schemi di indirizzamento* che costituiscono una *funzione di associazione fra i blocchi di memoria e le posizioni in cache*.

Esistono 3 tipi di indirizzamento:

- *Indirizzamento diretto;*
- *Indirizzamento associativo;*
- *Indirizzamento associativo a gruppi;*

Schema di indirizzamento diretto

Ogni blocco di memoria centrale è caricabile in una sola specifica posizione di cache, conoscendo il blocco da caricare si conosce anche la posizione in cui caricarlo.

La metodologia è alquanto semplice:

1. Si numerano i blocchi in memoria in ordine a partire da 0;
2. La *posizione* in cache di *un blocco i* è calcolata mediante modulo: $i \bmod n$, dove n è il numero di posizioni della cache;
3. Ciò crea degli insiemi di n blocchi, dove n è il numero di posizioni della cache.

L'indirizzo di un dato in cache è costituito da 3 capi:

- Spiazzamento: indica la posizione della parola all'interno del blocco;
- Blocco: indica la posizione del blocco che contiene la parola nell'insieme di n blocchi;
- Etichetta: indica la posizione dell'insieme di blocchi all'interno della memoria primaria;
 - A ciascuna posizione viene associata un'etichetta per riconoscere il blocco caricato;

Una problematica dell'indirizzamento diretto è che a causa del modulo, usato per identificare la posizione del blocco, più blocchi possono contendersi la stessa posizione e nonostante le altre posizioni in cache siano libere siamo costretti a caricare il blocco in quella posizione, sovrascrivendo il vecchio blocco.

Schema di indirizzamento associativo

Ogni blocco di memoria centrale è caricabile in una qualunque posizione di cache.

L'indirizzo di un dato in cache è costituito da 2 capi:

- Spiazzamento: indica la posizione della parola all'interno del blocco;
- Etichetta: indica la posizione del blocco all'interno della memoria primaria;
 - A ciascuna posizione viene associata un'etichetta per riconoscere il blocco caricato;

Quando vi sono posizioni libere e vi è necessità di caricare un blocco, quest'ultimo viene caricato in una qualsiasi posizione libera. Quando però la cache è piena è necessario un algoritmo di sostituzione per scegliere quale posizione svuotare.

Schema di indirizzamento associativo a gruppi

Ogni blocco di memoria centrale è caricabile in una sola specifica posizione di cache. Inoltre le posizioni in cache sono divise in *gruppi* di v posizioni. Ciò implica che un blocco può essere caricato in una sola specifica posizione di ogni gruppo.

Questo indirizzamento rappresenta la generalizzazione degli altri due indirizzamenti. Con gruppi da un solo elemento si ottiene l'indirizzamento diretto, con un gruppo capace di contenere tutti gli elementi si ha l'indirizzamento associativo.

La metodologia è la seguente:

1. Si numerano i blocchi in memoria in ordine a partire da 0;
2. La *posizione* in cache di *un blocco* i è calcolata mediante modulo: $i \bmod \left(\frac{n}{v}\right)$, dove n è il numero di posizioni della cache;
3. Ciò crea degli insiemi di v blocchi, dove v è il numero di posizioni nei gruppi;

L'indirizzo di un dato in cache è costituito da 3 capi:

- Spiazzamento: indica la posizione della parola all'interno del blocco;
- Gruppo: indica la posizione del blocco che contiene la parola di v blocchi;
- Etichetta: indica la posizione dell'insieme di v blocchi all'interno della memoria primaria;
 - A ciascuna posizione viene associata un'etichetta per riconoscere il blocco caricato;

Gli algoritmi di sostituzione

Negli indirizzamenti associativi bisogna scegliere quale posizione di cache liberare nel caso tutte le posizioni siano occupate. Vi sono vari possibili algoritmi:

- LRU (least recently used): sostituire il blocco usato meno di recente. Si assegna un contatore ad ogni blocco che incrementa ad ogni ciclo di clock:
 - Se accade cache hit: si azzerava il contatore del blocco, si incrementano di 1 i contatori inferiori e si lasciano invariati quelli superiori;
 - Se accade cache miss con gruppo non pieno: si carica il blocco in una posizione vuota azzerandone il contatore e si incrementano di 1 gli altri contatori;
 - Se accade cache miss con gruppo pieno: si libera la posizione con contatore massimo, vi si carica il nuovo blocco azzerandone il contatore e si incrementano di 1 gli altri contatori;
- FIFO (first in first out): sostituire il blocco caricato meno di recente;
- Casuale su distribuzione uniforme;

Memoria virtuale

Il processore vede la memoria primaria e secondaria come un'unica unità denominata *memoria virtuale* veloce come la cache e capiente come la memoria secondaria.

Quando il processore richiede un'informazione utilizza l'indirizzo logico, ovvero la posizione che l'informazione ha nella memoria virtuale. Tale indirizzo dovrà essere tradotto in un indirizzo fisico al fine di individuare la posizione che ha l'informazione nella memoria primaria o in caso effettuare il suo caricamento dalla memoria di massa.

MMU e Traduzione degli indirizzi

L'unità di gestione della memoria (MMU) è integrata all'interno del processore e si occupa degli indirizzamenti fra processore e memoria. In particolare si occupa di tradurre gli indirizzi logici di memoria virtuale in indirizzi fisici della memoria primaria e, nel caso in cui il blocco non si trovi in memoria primaria, richiede al sistema operativo di caricarla dalla memoria secondaria attraverso un segnale di *fault di pagina*.

L'unità elementare di informazione trasferibile fra memoria centrale e memoria di massa è detta *pagina*, ha una dimensione che va da 2K e 16K parole. La regione della memoria principale capace di contenere una pagina è denominata *area di pagina*.

L'indirizzo logico viene diviso in:

- Spiazzamento: posizione della parola all'interno della pagina;
- Numero di pagina logico: posizione della pagina nella memoria virtuale;

La traduzione di un indirizzo logico in un indirizzo fisico consiste in un indirizzo avente lo stesso spiazzamento e il numero di pagina fisico, che indica la posizione della pagina nella memoria primaria.

L'MMU esegue la traduzione utilizzando la *tabella delle pagine*, una tabella contenente un campo per ogni numero di pagina logico. Ogni campo della tabella contiene dei *bit di controllo* e il *numero di pagina fisico*. Quando l'MMU effettua la traduzione accede al campo della tabella per effettuare controlli e ottenere il numero di pagina fisico.

Per accedere a un campo della tabella si ha che l'indirizzo del campo relativo a una *pagina logica A* è equivalente a $\text{indirizzo} = \text{registro}_{\text{base}} + \text{numero}_{\text{logico}}$, dove $\text{registro}_{\text{base}}$ contiene l'indirizzo del primo elemento della tabella.

La cache dell'MMU

Nonostante l'MMU sia all'interno del processore, la tabella delle pagine è dimensione molto grande che fa sì essa risieda nella memoria principale. Ciò comporta una perdita di efficienza data dalla lentezza della memoria primaria rispetto al processore.

Per risolvere tale problematica l'MMU possiede una cache denominata *Translation Lookaside Buffer (TLB)* che contiene una copia delle righe della tabella delle pagine usate più di recente.

I campi della TLB posseggono un ulteriore campo costituito dal numero logico della pagina, utilizzato per eseguire la ricerca in cache che può risultare in:

- *LTB hit*: quando il numero di pagina logico è presente nella LTB e quindi si crea l'indirizzo fisico;
- *LTB miss*: il numero logico cercato non si trova nella LTB ed è necessario caricare la pagina dalla tabella delle pagine;
- *Fault di pagina*: quando la pagina cercata non si trova in memoria centrale, ovvero ha il bit di validità a 0, ed è necessario caricarla dalla memoria secondaria;
 - Ciò costituisce un'interruzione che blocca l'esecuzione dell'istruzione corrente;

Analogamente alla cache, quando la TLB è piena ed è necessario caricare ulteriori pagine si effettua una sostituzione mediante uno degli algoritmi noti.

Direct Memory Access

Le operazioni sulla memoria secondaria sono molto lente, la necessità dell'intervento del processore per i trasferimenti di dati comporta una grossa perdita di performance.

Per risolvere tale problematica si è definita la *Direct Memory Access (DMA)*, una tecnica che permette ad un dispositivo I/O di interagire con la memoria indipendentemente dal processore. La tecnica consiste in un controllore speciale denominato *Controllore DMA* collegato da un lato al bus e dall'altro alla periferica che gestisce il trasferimento di blocchi di dato fra la periferica e la memoria centrale.

Il trasferimento dei dati è inizializzato attraverso dei registri appositi:

- *Registro di controllo*: contenente i bit di controllo di trasferimento come *bit di operazione bit di fine*, *bit di abilitazione interruzione* e *bit richiesta interruzione*;
- *Registro di indirizzo*: contenente l'indirizzo iniziale del buffer di memoria per il blocco di dati;
- *Registro di dimensione*: contenente la dimensione del buffer di memoria per il blocco di dati;

Il trasferimento è del tutto indipendente dal processore, il quale viene avvisato del completamento attraverso la generazione da parte del controller DMA di un'interruzione.

Miglioramenti sul controllo della cache

Per migliorare la gestione della cache vi sono principalmente 3 tecniche:

- *L'utilizzo di un buffer di scrittura* in cui salvare temporaneamente il dato da scrivere in memoria, il contenuto del buffer verrà scritto in memoria in un secondo momento, ciò permette al processore di non attendere durante la fase di scrittura immediata;
- *Il caricamento anticipato in cache dei blocchi di memoria*, effettuato via software con istruzioni che simulano cache miss o via hardware con circuiti di controllo appositi;
- *L'uso di cache non bloccante*, ovvero una cache che non si blocca durante il caricamento di un blocco dovuto a un cache miss, realizzabile con sistemi di controllo più complessi;

Memoria ad accesso casuale (RAM)

La memoria ad accesso denominata RAM (Random Access Memory) garantisce una velocità di accesso costante per ogni indirizzo. Ha una capacità limitata ed è volatile, ovvero una volta interrotta l'alimentazione il suo contenuto viene perso. La memoria RAM è utilizzata per realizzare le memorie cache e la memoria principale.

Fisicamente sono costituite da celle di transistor a semiconduttori, dalla capacità di un bit. Gruppi di celle da un bit formano un *memory chip*. Gruppi di *memory chip* formano il banco di memoria.

Nei memory chip le celle da un bit sono organizzate a matrice:

- Le righe rappresentano le parole;
- Le celle di una riga sono collegate ad una *linea di parola* comune;
- Le celle di una colonna sono collegate a due linee dati comuni, che rappresentano i morsetti d'ingresso e uscita;

Un *decodificatore di indirizzi* seleziona quale *linea di parola* attivare e un segnale di controllo *RW* decide che operazione effettuare.

Nei banchi di memoria i memory chip sono disposti a matrice e vengono gestiti in modo analogo ai memory chip;

La memoria RAM si suddivide in due categorie: RAM statica e RAM dinamica.

RAM statica (SRAM)

La *memoria statica (SRAM)* è una memoria *volatile* che mantiene il suo stato fintanto che viene alimentata.

Una cella di memoria statica viene realizzata con una coppia di negatori retroazionati connessi alle linee dato mediante due transistor.

La linea di parola mette in conduzione o interdizione i transistor causando il seguente comportamento:

- Con la linea di parola disattivata ($=0$) la cella isolata manterrà il suo valore;
- Con la linea di parola attivata ($=1$) a seconda del segnale RW leggerà o forzerà il dato sulle linee dato;

I negatori della cella di memoria possono essere realizzati con *negatori CMOS* al fine di mantenere bassi i consumi. La realizzazione di una cella richiede 6 CMOS, ciò le rende molto costose e per questo motivo la memoria statica viene principalmente usata per produrre cache.

RAM dinamica (DRAM)

La *memoria dinamica (DRAM)* è memoria volatile più economica della memoria statica che però tende a perdere il proprio stato dopo poco tempo seppure alimentata.

Una cella di memoria dinamica è viene realizzata con un *condensatore collegato alla linea di bit attraverso un transistor, il quale è collegato alla linea di parola.*

Lo stato di una cella è rappresentato come *carica elettrica del condensatore*. Il condensatore però dissipa la carica in poco tempo, è perciò necessario *“rinfrescare” la carica periodicamente.*

La linea di parola mette in conduzione o interdizione i transistor causando il seguente comportamento:

- Con la linea di parola disattivata ($=0$) il condensatore isolato mantiene la carica fino alla dissipazione;
- Con la linea di parola attivata ($=1$) a seconda del segnale RW si ha che:
 - In caso di lettura, il circuito legge il dato e rinfresca il contenuto della linea di bit;
 - In caso di scrittura: il circuito aggiorna il contenuto della linea di bit;

Il costo della realizzazione delle celle di memoria dinamica, 2 componenti, è nettamente inferiore a quello delle celle di memoria statica, per questo motivo vengono impiegate per la realizzazione della memoria principale.

La memoria dinamica si suddivide in due tipologie:

- DRAM asincrona: quando il controllo è gestito da un circuito di controllo esterno al chip. Vi è inoltre la presenza di due registri buffer d'ingresso denominate *Row/Column Address Strobe che permettono la fast page mode, l'invio veloce di blocchi di dati appartenenti alla stessa riga.*
- DRAM sincrona (SDRAM): quando il controllo è scandito da un ciclo di clock ed è gestito da un circuito di controllo integrato nel chip. Il rinfresco è gestito da un contatore di rinfresco. Inoltre sono presenti registri buffer sia in ingresso sia in uscita.

Le memorie dinamiche sincrone (SDRAM) standard sono generalmente sincronizzate sul fronte di salita del ciclo di clock.

Oggi è largamente diffusa una variante *denominata SDRAM a frequenza doppia* che selezionano le righe nel fronte di salita ma *trasferiscono dati* in entrambi i fronti, raddoppiando quindi *la banda passante.*

Memoria a sola lettura (ROM) e varianti (PROM, EPROM, EEPROM)

La memoria a sola lettura è una memoria in grado di tenere il proprio stato in modo permanente.

Lo stato dei bit di una memoria ROM viene impostato durante la fase di produzione e non può più essere cambiato, sono realizzate con un *interruttore* fisso collegato alla *linea dei bit* attraverso un *transistor*, il quale è collegato alla *linea di parola*.

A seconda di come viene impostato l'interruttore si ha:

- Con l'interruttore aperto la linea di bit avrà una tensione di alimentazione, è quindi varrà 1;
- Con l'interruttore chiuso la linea di bit avrà una tensione di massa, è quindi varrà 0;

Col tempo si sono realizzate diverse varianti di memorie ROM per diversi impieghi:

- Programmable ROM: memorie che possono essere scritte una sola volta. Sono realizzate sostituendo l'interruttore della ROM con un fusibile. Dopo la fabbricazione un chip PROM è vergine, ovvero tutte le celle contengono zero. La programmazione avviene mediante un dispositivo programmatore che brucia i fusibili delle celle mettendole così a 1;
- Erasable PROM: memorie che possono essere riprogrammate più volte, spesso utilizzate per realizzare prototipi. Sono realizzate sostituendo l'interruttore delle ROM con un transistor speciale capace di trattenere una carica elettrica:
 - La cella vale 1 quando viene inviata una carica elettrica al transistor che verrà trattenuta;
 - La cella vale 0 quando viene liberata la carica elettrica trattenuta dal transistor;

La cancellazione di *tutta la memoria* avviene esponendo il chip a dei raggi UV, una volta cancellata la memoria il chip può essere riprogrammato da una macchina programmatrice. Il principale problema della EPROM è costituito dal fatto che per cancellare la memoria è necessario scollegare il chip dal circuito in cui è inserito.

- Electrically Erasable PROM (EEPROM): memorie che possono essere riprogrammate più volte. Rappresentano l'evoluzione delle EPROM in quanto:
 - Le celle possono essere programmate e cancellare in modo puramente elettrico, ciò risolve la problematica di dover scollegare il chip dal circuito;
 - È possibile modificare le singole celle indipendentemente;

Le operazioni di cancellazione, programmazione e lettura avvengono con 3 livelli di tensione diversi, ciò implica un circuito di pilotaggio abbastanza complesso.

Memoria flash

Le memorie flash sono memorie simili alle EEPROM con le seguenti caratteristiche:

- Memoria cancellabile e programmabile più volte;
- Un unico livello di tensione per le operazioni di cancellazione, programmazione e lettura;
- Si può operare solo su blocchi di celle anziché le singole celle;
- Densità di bit e capacità maggiori delle EEPROM;
- Costo e consumi inferiori delle EEPROM;

Queste caratteristiche hanno favorite l'enorme diffusione di tale tecnologia per le unità di memorizzazione come schede di memorie, SSD o chiavette di memoria;

Dischi magnetici

I dischi magnetici rappresentano la *tecnologia di massa più diffusa*. Le unità di memorizzazione che adottano tale tecnologia sono costituite da:

- Uno o più dischi rivestiti da materiale ferromagnetico;
 - In un disco i dati sono organizzati in sezioni concentriche del disco dette *tracce*, a sua volta una traccia è costituita da settori, che rappresentano la minima unità di lettura e scrittura dati;
- Al di sopra di ogni disco vi è una testina, formata da una bobina elettrica, responsabile della scrittura o lettura dei dati;
 - Per scrivere un dato, viene mandata una corrente alla testina la quale magnetizzerà la porzione di disco su cui si trova;
 - Per leggere un dato, la porzione del disco magnetizzata induce una carica sulla testina che verrà letta e amplificata;
- Un pilota del disco (disk driver) composto da un motore per la rotazione e movimento delle testine;
- Un circuito elettrico di controllo spesso esterno al dispositivo;

Calcoli aritmetici

Differenza fra Addizionatore con riporto e anticipo del riporto

L'addizionatore con riporto a n bit calcola l'ultimo riporto in uscita in $2n$ ritardi, mentre calcola l'ultimo bit della somma in $2n-1$ ritardi.

È possibile ottenere due funzioni particolari dall'espressione che descrive l'addizionatore:

- $G = x_i y_i$: funzione di generazione;
- $P_i = x_i + y_i$: funziona di propagazione;

Funzioni che dipendono solo dagli ingressi x e y e calcolabili in un solo ritardo di porta.

Modificando la cella del sommatore a 1 bit affinché utilizzi tali funzioni si ottengono le *celle di stadio a un bit*.

Realizzando un addizionatore con n celle si ottiene un addizionatore con anticipo del riporto che riesce a calcolare l'ultimo riporto e il risultato con meno ritardi.

Moltiplicazione di numeri interi binari senza segno

La moltiplicazione binaria di *due numeri interi senza segno* segue la stessa logica della moltiplicazione fra due numeri decimali in colonna:

- Si moltiplica ciascuna cifra del moltiplicatore per il moltiplicando e si sommano i risultati facendoli scorrere di una posizione ogni cifra;
 - Quando si moltiplica per 0 la riga sarà fatta di soli zeri, quando si moltiplica per 1 si ricopia il moltiplicando;
- Il risultato di una moltiplicazione di due numeri binari a n bit sarà un numero di $2n$ bit;

È possibile realizzare la moltiplicazione con un circuito di celle full-adder a matrice: ogni riga rappresenta un prodotto parziale. Viene eseguita la somma per colonne e il riporto viene inoltrato alla colonna successiva;

La rete sequenziale capace di realizzare una moltiplicazione è costituita da un addizionatore a n bit con due registri a scorrimento, ad ogni ciclo viene effettuata la somma del moltiplicando o un set di 0 e il prodotto parziale fatto scorrere di una posizione;

Moltiplicazione di numeri interi binari con segno (Soluzione diretta, algoritmo di booth e bit pair)

La moltiplicazione binaria di *due numeri interi con segno* segue la seguente logica:

- Se il moltiplicatore è positivo:
 - E il moltiplicando è positivo si esegue la normale la stessa logica della moltiplicazione senza segno;
 - E il moltiplicando è negativo: si estende il bit di segno e si esegue la moltiplicazione;
- Se il moltiplicatore è negativo:
 - Si esegue il complemento a 2 sia del moltiplicando sia del moltiplicatore per ricadere nel caso precedente;

Esiste una tecnica più generica per eseguire la moltiplicazione basata sull'algoritmo di Booth. L'algoritmo di Booth consiste nel ricodificare il moltiplicatore come una *somma e sottrazione di potenze di 2*.

La codifica consiste in una sequenza di 0, -1 e +1 basata su porzioni contigue di 1:

- Data una porzione contigua di 1 si ha una posizione j di inizio e una posizione i di fine:
 - Nella posizione j si mette un -1;
 - Nella posizione $i+1$ si mette un +1;
 - Le sequenze di 0 non precedute da 1 o precedute da +1 vengono ricopiate;

Quando si utilizza il moltiplicatore in tale codifica si ha che:

- Quando si moltiplica per 0, si somma una riga di 0;
- Quando si moltiplica per -1, si somma il complemento del moltiplicando;
- Quando si moltiplica per 1, si somma il moltiplicando;

Dalla codifica di Booth è possibile ottenere una codifica più efficiente denominata *bit-pair*: consiste nel raggruppare a coppie i bit del moltiplicatore codificato con Booth al fine di dimezzare il numero di addizioni.

Moltiplicazione veloce

Per ottenere un circuito capace di eseguire una moltiplicazione veloce si hanno i seguenti passi:

1. Utilizzare la codifica bit-pair;
2. Usare un albero di riduzione CSA per ridurre gli addendi a 2;
3. Calcolare il prodotto sommando i due addendi con un addizionatore ad anticipo del riporto;

Divisione di numeri interi binari (Divisione con e senza ripristino)

La divisione binaria di numeri interi segue una logica simile a quella della divisione binaria:

1. Si allineano il divisore con il dividendo a partire da sinistra;
2. Si scendono cifre fintanto non si ha che il divisore è contenuto;
 - Se è contenuto si aggiunge 1 al quoziente;
 - Se non è contenuto si aggiunge 0 al quoziente;
3. Si esegue la sottrazione per ottenere la divisione parziale;
4. Si allinea il divisore e si ritorna al passo 2;

Somma e sottrazione di numeri frazionari binari

Per poter effettuare la somma o sottrazione di numeri frazionari si eseguono i seguenti passi:

- Si sceglie il numero con esponente più piccolo e si fa scorrere la sua mantissa verso destra, moltiplicazione per potenze negative di 2, per un numero di salti pari alla differenza fra gli esponenti;
- Si pone l'esponente del risultato pari all'esponente più grande;
- Si addizionano i fattori interi tenendo conto del segno;
- Se il numero risultante non è in forma normale, lo si normalizza.

Moltiplicazione di numeri frazionari binari

Per poter effettuare la moltiplicazione di numeri frazionari si eseguono i seguenti passi:

- Si sommano gli esponenti e si sottrae 127 alla somma, ottenendo l'esponente provvisorio;
- Si moltiplicano i fattori interi ottenendo segno e fattore del prodotto;
- Se il numero risultante non è in forma normale, lo si normalizza.

Divisione di numeri frazionari binari

Per poter effettuare la divisione di numeri frazionari si eseguono i seguenti passi:

- Si sottrae l'esponente del divisore all'esponente del dividendo e si aggiunge 127, ottenendo l'esponente provvisorio;
- Si dividono i fattori interi ottenendo segno e fattore del prodotto;
- Se il numero risultante non è in forma normale, lo si normalizza.