

CALCOLATORI ELETTRONICI

Prof. Alessandro Cilardo – A.A. 2022/23

Luca Maria Incarnato

INDICE DEGLI ARGOMENTI

RAPPRESENTAZIONE DELL'INFORMAZIONE

1. CHE COSA È UN CALCOLATORE (p. 3)
2. RAPPRESENTAZIONE NUMERICA INTERA (p. 5)
3. OVERFLOW E UNDERFLOW (p. 8)
4. RAPPRESENTAZIONE DEI NUMERI FRAZIONARI (p. 10)
5. RAPPRESENTAZIONE DI UN'INFORMAZIONE NON NUMERICA (p. 12)

ARCHITETTURA DEI CALCOLATORI

6. IL PROCESSORE (p. 14)
7. LA MEMORIA CENTRALE E IL CICLO DEL PROCESSORE (p. 16)
8. CLASSIFICAZIONE DELLE ISTRUZIONI (p. 19)
9. LINGUAGGI ASSEMBLER E IL MOTOROLA 68K (p. 22)
10. OPERAZIONI ELEMENTARI IN ASSEMBLY (p. 27)
11. MODI DI INDIRIZZAMENTO (p. 32)
12. COSTRUTTI PER IL CONTROLLO DI FLUSSO (p. 35)
13. SOTTOPROGRAMMI (p. 40)
14. PERIFERICHE DI I/O (p. 49)
15. INTERRUZIONI (p. 51)

RETI LOGICHE

16. PORTE LOGICHE E L'ALGEBRA DI BOOLE (p. 58)
17. I VARI MODELLI DI ALGEBRA DI BOOLE (p. 62)
18. FUNZIONI BOOLEANE E STUDIO DELLE RETI LOGICHE (p. 63)
19. LE FUNZIONI DI COSTO E LE FORME MINIME (p. 66)
20. PUNTI DI INDETERMINAZIONE – DON'T CARE (p. 75)
21. ALTRE PORTE LOGICHE FONDAMENTALI (p. 76)
22. RETI COMBINATORIE NOTEVOLI (P. 81)
23. RETI LOGICHE SEQUENZIALI (p. 90)
24. ANALISI E SINTESI DI UNA RETE SEQUENZIALE (p. 97)
25. STATI E MACCHINE EQUIVALENTI (p. 105)

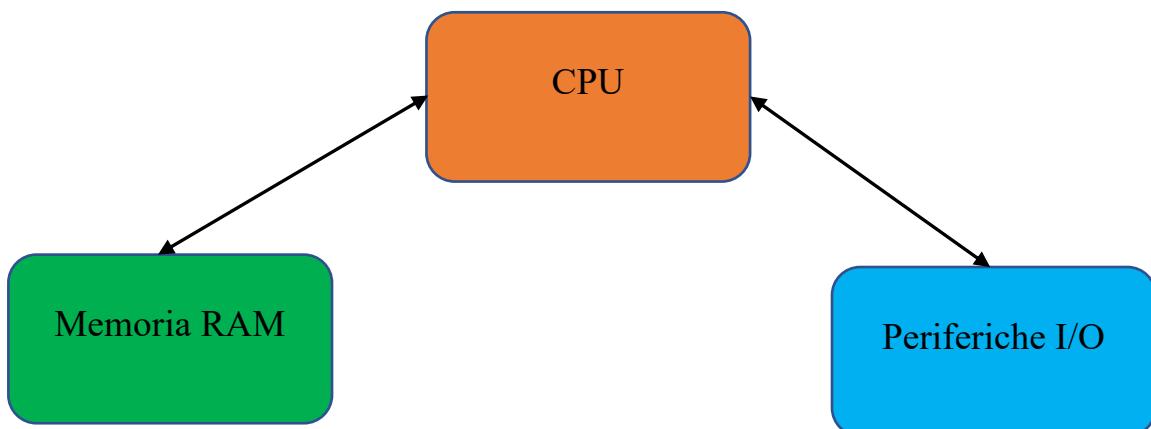
RAPPRESENTAZIONE DELL'INFORMAZIONE

CHE COSA È UN CALCOLATORE

Un **calcolatore** è un **dispositivo hardware** che viene **programmato** per eseguire un'operazione attraverso un **software**, indipendentemente da come esso viene programmato, dal tipo di componentistica o dallo scopo. L'**architettura di un calcolatore** è **specificata** per ogni utilizzo e ogni casa produttrice; tuttavia, è possibile andare a **definire alcuni componenti e il modo in cui essi comunicano indipendentemente** da tali fattori. Tale modello, che prende il nome di modello **black box**, serve a comprendere **come si comportano i componenti** nel loro utilizzo **senza tenere conto del modo pratico in cui essi sono fatti**. Il modello black box di **Von Neumann** è attualmente considerato il **più affidabile**, esso prevede che ogni macchina sia composta di tre componenti fondamentali:

1. **CPU** (Central Process Unit, o processore), il componente fondamentale che gestisce ogni istruzione;
2. Memoria **RAM**, nella quale vengono contenute le istruzioni che il processore dovrà eseguire;
3. **Dispositivi di I/O** (o periferiche), componenti adibite all'input o all'output di dati ai fini del completamento di un'istruzione.

Questi tre componenti **comunicano tra di loro** al fine di scambiarsi i dati, ad esempio la memoria comunica con il processore per fornirgli le istruzioni da eseguire o, viceversa, per conservare il risultato di un'operazione. In genere i collegamenti tra questi componenti possono essere riassunti con il seguente schema:



Tali componenti comunicano attraverso i **bus**, **collegamenti di interconnessione** che permettono lo **scambio di informazioni**. Nel modello di Von Neumann il termine **bus** è **limitante**, in quanto fa riferimento alla pratica, ormai in disuso, con la quale i componenti erano collegati da fili in comune. Ad oggi un **collegamento tra due componenti** può essere anche **isolato** o **indipendente** dagli altri: ad esempio, CPU e RAM devono avere un canale di comunicazione dedicato per una maggiore velocità di calcolo, poiché non occupato da altre trasmissioni. Infatti, l'**interconnessione genera inevitabilmente un ritardo** che, per essere ridotto al minimo, necessita di uno spazio di percorrenza il più breve e libero possibile. A tal fine, nei calcolatori moderni, il sistema di bus è organizzato **gerarchicamente**: componenti **più importanti** hanno **bus dedicati, più corti e più veloci**, al costo

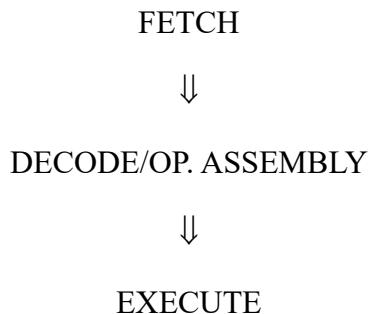
di una complessità di progettazione maggiore; tuttavia, in ogni bus di n fili sono codificabili 2^n informazioni.

In genere i bus possono essere di tre tipologie:

1. Bus di **controllo**, che trasporta informazioni sull'esecuzione dell'istruzione;
2. Bus di **indirizzo**, che trasporta gli indirizzi di memoria da cui prelevare l'informazione;
3. Bus **dato**, che trasporta le informazioni su cui eseguire le istruzioni.

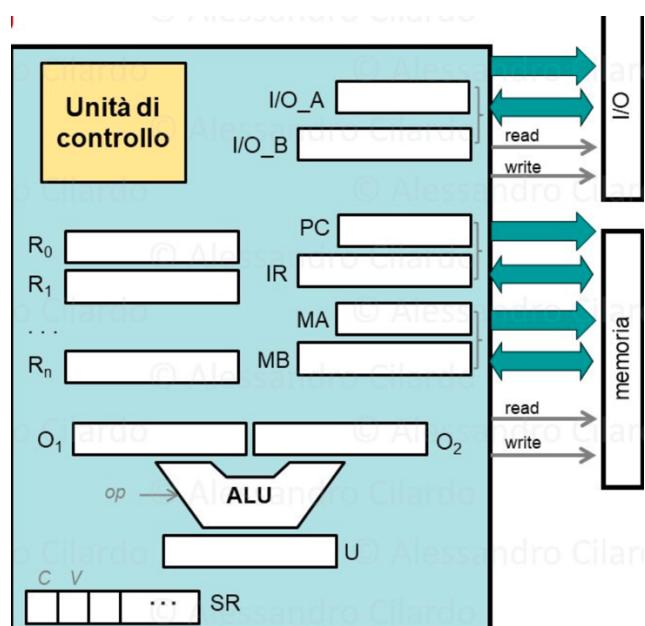
Ogni tipologia di bus ha una dimensione determinata in funzione della mole di informazioni da trasmettere. Ad esempio, il bus indirizzo ha la stessa dimensione di un indirizzo, cioè 2^n dove n è la lunghezza di ogni indirizzo, direttamente proporzionale alla quantità di memoria.

Il processo di funzionamento di un calcolatore viene indicato come **ciclo del processore** e si compone di tre fasi principali, seguenti la fase di boot:



Queste tre fasi vengono **ripetute periodicamente** per ogni singola istruzione. È possibile anche introdurre un'altra fase, quella di **interruzione**, nel caso la CPU deleghi alcuni compiti ad altre periferiche.

Il modello di Von Neumann è definito modello a **stored procedure**, il che significa che le istruzioni sono **conservate in memoria RAM** e prese all'occorrenza dal processore. In genere il modo di operare di un processore è riassunto dal seguente schema:



Esso tiene anche in considerazione del prelievo di istruzioni in memoria. I rettangoli contrassegnati da sigle sono dei **registri interni** che servono all'Unità di Controllo per gestire le operazioni e i dati che servono alla loro esecuzione. Ad oggi esistono anche dei **blocchi di memoria** che sono inseriti **dentro al circuito del processore** per garantire un **trasferimento più veloce** e, quindi, l'esecuzione più rapida di istruzioni; tali memorie sono dette **cache** e in esse sono contenute le **informazioni che alla CPU servono con più accessibilità**, infatti generalmente **più una memoria è vicina al processore più essa è rapida** nel consegnargli dati (le memorie di massa sono molto lente perché sono lontane dalla CPU).

RAPPRESENTAZIONE NUMERICA INTERA

La maggior parte dei calcolatori moderni è realizzata con tecnologie che permettono la **rappresentazione di soli due stati stabili**, codificati come 0 e 1. Da ciò consegue che l'informazione gestita da un calcolatore è organizzata in **bit**, degli **elementi di informazione binaria** che possono assumere il valore 0 o il valore 1. Dal momento in cui le informazioni che vengono comunicate al computer non sono di natura binaria (basti pensare agli addendi per una somma o a un messaggio da inviare ad un altro calcolatore), è necessario sviluppare un sistema che effettui una **corrispondenza** tra il sistema di numerazione binario e le informazioni che bisogna comunicare.

Il sistema di numerazione binario è composto da un **alfabeto di cardinalità due** [0; 1]. Esso risulta utile perché condivide con il sistema decimale il concetto di **peso**: entrambi i **sistemi di numerazione** sono **pesati**, cioè la cifra alla posizione n viene pesata in funzione della base alla n:

$$567 = 5 \cdot 10^2 + 6 \cdot 10^1 + 7 \cdot 10^0$$

Inoltre, l'incremento di un numero avviene modificando sempre la cifra più a destra. Queste affinità permettono la **conversione di un numero in binario in numero decimale** con estrema facilità, dal momento in cui condividono alcune regole di numerazione.

Per convertire un numero in base 2 bisogna applicare il procedimento sopra mostrato ma con base 2:

$$[10110]_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = [22]_{10}$$

Per l'operazione inversa basta **dividere il numero per due** e registrare il resto finché non si raggiunge un dividendo più piccolo del divisore. Il **numero binario sarà l'insieme di resti** letti in **ordine inverso** di uscita:

$$22: 2 = 11 \ r0$$

$$11: 2 = 5 \ r1$$

$$5: 2 = 2 \ r1$$

$$2: 2 = 1 \ r0$$

$$1: 2 = 0 \ r1$$

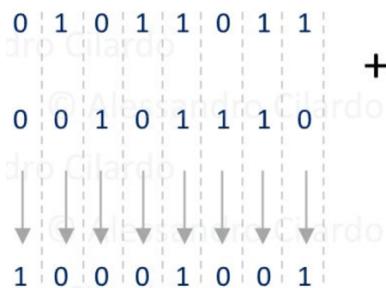
$$[22]_{10} = [10110]_2$$

Esistono diversi modi di rappresentare un numero in binario e viceversa, tutti con un proprio **intervallo di rappresentazione** (l'intervallo di numeri che la codifica permette di tradurre). Il modo

più semplice è quello posizionale pesato mostrato di sopra, che tiene conto solo di **numeri interi** positivi appartenenti all'intervallo di rappresentazione $[0; 2^n - 1]$, dove n è il numero di bit massimi che può avere un numero.

Sui numeri binari è possibile eseguire **qualsiasi operazione matematica** (a patto che si rispettino il numero di bit a disposizione); tuttavia, un calcolatore ha bisogno **solo di due circuiti** per eseguire le operazioni più elementari, uno per la **somma in colonna** e uno per lo **shift** di un numero.

Per la somma non bisogna fare altro che **incolonnare i due numeri e sommare bit a bit**, tenendo conto che la somma di 0 e 1 è 1 e la somma di 1 e 1 è 10, quindi 0 con un riporto di 1 alla posizione successiva.



Per lo shift (a sinistra) non bisogna fare altro che **spostare ogni cifra di una posizione a sinistra**, aggiungendo lo zero al bit più a destra ed eliminando la cifra a sinistra nel caso sia un 1. Poiché il sistema binario è un sistema pesato, ogni cifra aumenta la sua posizione di un posto e, quindi, il suo peso aumenta di una potenza di due. Ciò significa che ogni cifra viene moltiplicata per due e, il risultato finale dello shift, sarà la **duplicazione del numero**.

Lo shift corrisponde alla moltiplicazione di due numeri, infatti **shiftando un numero di k posizioni lo si moltiplica per 2^k** . Questa operazione, però, tiene conto solo di fattori potenze di due e per moltiplicare per numeri generici bisogna andare a ricorrere ai bit alzati del corrispettivo numero binario. Infatti, moltiplicare un numero per una certa quantità significa sommare i numeri shiftati di volta in volta delle posizioni in cui tale quantità ha bit alzati. Ad esempio:

$$\begin{aligned} 5 = 101 &= 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 2^2 + 2^0 \Rightarrow n \times 5 = (n \times 2^2) + (n \times 2^0) \\ &= (n \times 4) + (n \times 1) \end{aligned}$$

Con queste due operazioni è possibile effettuare qualsiasi operazione algebrica elementare: l'**elevamento a potenza** equivale alla **reiterazione della moltiplicazione**, cioè dello shift a sinistra, mentre per la **divisione**, in quanto operazione inversa alla moltiplicazione, bisogna effettuare uno **shift a destra**.

Per la **sottrazione** basterebbe **sommare una quantità per l'opposto di un'altra**; tuttavia, l'attuale metodo di codifica non permette l'**espressione di quantità negative**, che possono essere implementate cambiando sistema. Tra i sistemi che permettono la codifica di quantità negative si trovano:

- **Rappresentazione segno e modulo**

In questo tipo di codifica **il bit più a sinistra viene adibito alla codifica del segno** e i restanti alla codifica del modulo. Inevitabilmente, tale sistema riduce l'intervallo di rappresentazione di un bit, cioè $[-2^{n-1} + 1; 2^{n-1} - 1]$; inoltre, è possibile notare come questo sistema codifichi **due volte il numero zero**, infatti con un modulo tutto 0 il segno può essere ancora positivo o negativo, quindi:

$$[0000]_2 = [0]_{10} = [1000]_2$$

L'implementazione di questo sistema di rappresentazione **richiede**, quindi, la **costruzione di altri circuiti** che gestiscano il segno e il bit più a sinistra.

- **Complemento alla base**

Questo sistema è identico a quello numerale pesato, con l'unica differenza che **la valutazione della posizione più a sinistra è negativa**:

$$101 = -2^2 + 2^0 = -3$$

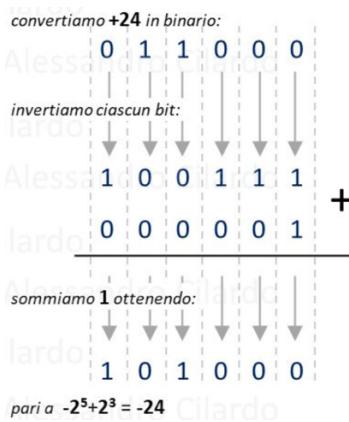
- **Complemento a due**

Questo tipo di rappresentazione viene determinata dividendo tutti i numeri binari in quelli che hanno bit a sinistra 1 e 0. Quelli con 1 saranno numeri negativi e quelli con 0 positivi; per determinare il numero in sé e per sé si **trasla l'intervallo di rappresentazione** di 2^n :

000	001	010	011	100	101	110	111
0	1	2	3	4	5	6	7
0	1	2	3	-4	-3	-2	-1
-4	-3	-2	-1	NN	NN	NN	NN

Poiché la base del nostro sistema di numerazione è 2, il sistema in **complemento a due** e in **complemento alla base coincidono** e l'intervallo di rappresentazione è $[-2^{n-1}; 2^{n-1} - 1]$

Avendo a disposizione un numero positivo è possibile ricavare il suo **opposto invertendo tutti i bit e sommando 1**:



È anche possibile andare ad estendere il numero oltre i bit prefissati; con un numero positivo basta aggiungere tanti 0 quanti necessitati, con un numero negativo 1.

- **Complementi diminuiti**

Questo sistema è **uguale** al sistema in **complemento a due** per i **numeri positivi**; per i **numeri negativi**, invece, bisogna prima **sottrarre di uno** e poi usare la codifica in complemento a due, il che significa che per trovare l'opposto di un numero bisogna semplicemente invertire tutte le cifre. Per la somma e la sottrazione vanno fatte alcune considerazioni: interpretando i numeri in complemento diminuiti come complemento a due, si commette un errore di -1 per i numeri negativi, che verrà

portato nella somma. È quindi necessario un aggiustamento dopo l'operazione e sommare 1 nel caso in cui si sommano due addendi negativi o due addendi discordi con risultato positivo.



- **Rappresentazione in eccesso k**

Il numero rappresentato viene **incrementato di k** e poi tradotto in binario puro e viceversa.

Riepilogando le varie rappresentazioni:

C₃	C₂	C₁	C₀	senza segno	complem. alla base	complem. diminuiti	segno e modulo	eccesso-k (ad es k=8)
0	0	0	0	0	0	0	0	-8
0	0	0	1	1	1	1	1	-7
0	0	1	0	2	2	2	2	-6
0	0	1	1	3	3	3	3	-5
0	1	0	0	4	4	4	4	-4
0	1	0	1	5	5	5	5	-3
0	1	1	0	6	6	6	6	-2
0	1	1	1	7	7	7	7	-1
1	0	0	0	8	-8	-7	-0	0
1	0	0	1	9	-7	-6	-1	1
1	0	1	0	10	-6	-5	-2	2
1	0	1	1	11	-5	-4	-3	3
1	1	0	0	12	-4	-3	-4	4
1	1	0	1	13	-3	-2	-5	5
1	1	1	0	14	-2	-1	-6	6
1	1	1	1	15	-1	0	-7	7

OVERFLOW E UNDERFLOW

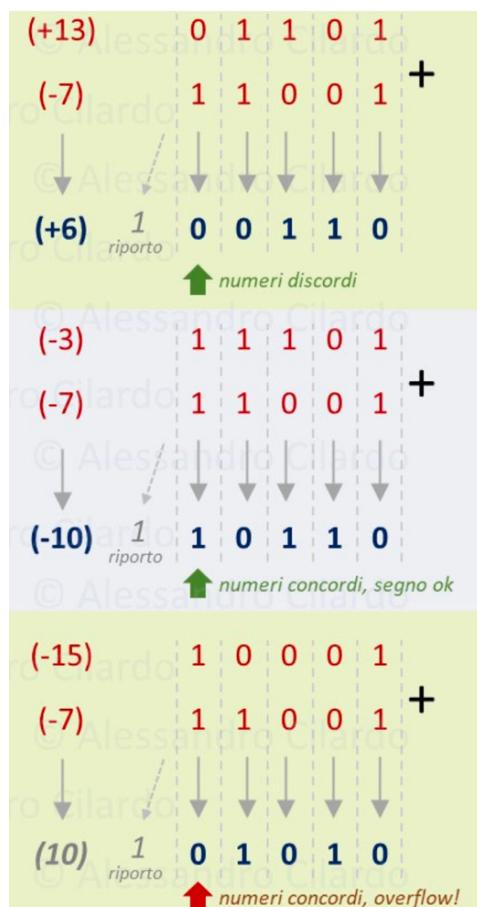
Quando un'operazione binaria si fa su carta viene **spontaneo e naturale aggiungere bit** a sinistra o a destra **nel momento del bisogno**; tuttavia, questa pratica **per i calcolatori è impossibile** perché **hanno una precisa quantità di bit** a disposizione che possono sfruttare e andare **oltre** equivale alla **perdita di quell'informazione**. Nei calcolatori viene implementato un circuito che interviene quando si verifica l'overflow o l'underflow, condizioni per le quali si sforano i bit a disposizione.

Per **overflow** si intende lo **sforamento del parallelismo** (il massimo bit) a sinistra di almeno una cifra e per **underflow** lo **sforamento del parallelismo a destra**. I due equivalgono alla codifica di **un numero maggiore del più grande numero rappresentabile** e di **un numero più piccolo del più piccolo rappresentabile**.

Con i **numeri positivi** è difficile raggiungere l'**underflow** ma è facile raggiungere l'**overflow** quando si lavora con numeri prossimi al limite massimo. Con le **moltiplicazioni** si arriva facilmente all'**overflow** come all'**underflow** per le **divisioni**; in corrispondenza di queste due operazioni l'**approssimazione si commette nell'ordine delle cifre shiftate** (infatti una moltiplicazione può anche duplicare la quantità di bit necessarie, nel caso si stia shiftando di 2^{MSB}).

Quando si effettuano **divisioni**, a meno che gli operandi non siano multipli, il risultato è un **numero frazionario**, codificabile inserendo come peso le potenze **negative** di 2. Nel caso in cui queste cifre non siano previste nella codifica si raggiunge l'**underflow**, che può essere evitato approssimando il **numero per difetto**.

Lavorando con le **sottrazioni** l'overflow va messo sotto un'ottica diversa. Infatti, non tutte le sottrazioni che sforano il parallelismo sfocano in un overflow; ad esempio, l'operazione $-1 - 3 = -4$ sfora il parallelismo ma, indipendentemente se si aggiunge o meno il bit sforato, il risultato binario è sempre -4. Dunque, generalmente **non si considera l'overflow quando la sottrazione prende in esame operandi discordi o operandi concordi con un risultato concorde**.



Con ciò è possibile concludere che un calcolatore, oltre ai **circuiti per la somma in colonna e lo shift**, deve avere un **circuito che compari gli ultimi bit di un'operazione** e di uno che sollevi **overflow/underflow** solo quando le condizioni si presentano.

RAPPRESENTAZIONE DEI NUMERI FRAZIONARI

Fino a questo momento sono stati presi in considerazione solo i numeri interi, senza tenere conto delle cifre decimali. Una numero con delle **cifre decimali** può essere rappresentato con diversi metodi di codifica, tutti però si distinguono per la quantità di cifre codificabili dopo la virgola. Esistono due metodi principali:

- **Codifica dei numeri in virgola fissa**, dove un numero ha una quantità finita di bit per la parte intera e una quantità finita di bit per la parte frazionaria, con il vantaggio di essere più semplice e lo svantaggio di una quantità ridotta di intervalli rappresentabili;
- **Codifica dei numeri in virgola mobile**, la quantità di bit per le cifre dopo la virgola è determinata in base al numero da rappresentare (non è predefinita), con il vantaggio di avere intervalli di rappresentazione più ampi ma con lo svantaggio di una maggior complessità.

Nella rappresentazione in **virgola fissa** si va a considerare un numero seguendo la **definizione di sistema di numerazione pesato** aggiungendo una quantità finita di posizioni **dopo lo zero con un indice negativo**:

$$[101110,101]_2 = 2^5 + 2^3 + 2^2 + 2^1 + 2^{-1} + 2^{-3} = [46,625]_{10}$$

I numeri frazionari sono facili da ottenere come risultato di una divisione per 2^k , quindi con uno shift di k posizioni. Avendo a disposizione n bit per la codifica di un numero di cui k per la codifica dei numeri dopo la virgola, si possono determinare:

- **Il numero più piccolo** (dopo lo zero) **rappresentabile**, che sarebbe il numero più piccolo dopo lo zero su n bit shiftato di k posizioni;

$$00001 \rightarrow 000,01$$

$$\frac{1}{2^k}$$

- **Il numero più grande rappresentabile**, che sarebbe il numero più grande su n bit shiftato di k posizioni.

$$11111 \rightarrow 111,11$$

$$\frac{2^n - 1}{2^k}$$

Con i numeri frazionari si può incontrare facilmente l'**underflow** (soprattutto con la virgola fissa), che accade quando il numero da rappresentare avrebbe bisogno di più cifre per la virgola. In quel caso il numero è “**non nullo**” ma non è rappresentabile e, quindi, viene **approssimato a zero**; l’**errore massimo che si commette** nell’approssimazione è **nell’ordine delle cifre dedicate alle cifre decimali**, cioè:

$$\frac{1}{2^k}$$

Per convertire un numero frazionario in binario si applica un **algoritmo analogo a quello per la conversione dei numeri interi** solo che, al posto di dividere per due e registrare il resto in ordine

inverso, si moltiplica per due (il numero senza parte intera) e si registra la parte intera ottenuta nello stesso ordine con cui compare. Ad esempio:

$$[0,625]_{10} = [0,101]_2$$

$$0,625 \cdot 2 = 1,25 \ i1$$

$$0,25 \cdot 2 = 0,5 \ i0$$

$$0,5 \cdot 2 = 1,0 \ i1$$

$$0,0 \cdot 2 = 0,0 \ i0$$

Nella rappresentazione in **virgola fissa** dei numeri frazionari **la quantità di cifre dedicate alla parte decimale è limitata e determinata a priori**, senza la possibilità di aumentarle o diminuirle. Ciò risulta un **problema**, dal momento in cui **nella realtà è difficile rispettare** un limite di cifre decimali e, per ovviare a tale limitazione, è stato sviluppato un sistema che gestisca in **maniera dinamica le cifre decimali** di un numero: il **sistema di numerazione in virgola mobile**.

La virgola mobile permette di rappresentare **per ogni numero una diversa posizione della virgola**, garantendo un **risultato più accurato** di quello che si otterrebbe da un'operazione in virgola fissa; tuttavia, il costo per la risoluzione del problema è una **maggior complessità** nelle operazioni aritmetiche come la somma e il prodotto. Alla base di questo sistema c'è il concetto di **mantissa ed esponente**, infatti ogni numero viene rappresentato come il **prodotto di una mantissa per la base elevata ad un esponente**:

$$m \cdot b^e$$

Per eliminare l'ambiguità che sorge dal fatto che spostando di un posto la virgola della mantissa e diminuendo l'esponente si ottengono diverse forme per lo stesso numero, si sceglie sempre la **forma normalizzata** del numero, cioè quella forma che prevede la parte intera della mantissa compresa tra zero e la base esclusi:

$$m \in (0; b)$$

Come conseguenza della normalizzazione, quando **la base è 2 la parte intera** della mantissa non può essere 0 o 2 ma **necessariamente 1**; ciò significa che **ogni numero avrà come prima cifra 1** ed essa potrà essere **omessa**. Per normalizzare una cifra binaria si decide di posizionare la virgola dopo il primo 1 da sinistra:

$$101101,1101 \cdot 2^1 \rightarrow 1,011011101 \cdot 2^6$$

Esistono diverse altre variabili da tenere conto per codificare un numero (come il segno) e qui intervengono gli standard di rappresentazione. Il più usato è lo **standard IEEE 754** che divide il numero in: **segno, esponente e mantissa**. Ci sono **due possibili dimensioni** per ogni numero (escludendo la precisione doppia che non è quasi mai usata):

- Numeri in **singola precisione**, o **32 bit**;

Segno	Esponente (eccesso 127)	Mantissa normalizzata
1 bit	8 bit	23 bit

- Numeri in **precisione doppia**, o **64 bit**.

Segno	Esponente (eccesso 127)	Mantissa normalizzata
1 bit	11 bit	52 bit

Un esempio di codifica di un numero nel formato IEEE 754:

1 10000100 00011011000000000000000000000000

Segno: 1 → numero negativo

Esponente: $(10000100)_2 = 132 \rightarrow 132 - 127 = 5$

Mantissa: $(1,000110110000000000000000)_2 = 1, (2^{-4} + 2^{-5} + 2^{-7} + 2^{-8}) = 1,10546875$

Il numero codificato è: $-1,10546875 \cdot 2^5 = -35,375$

La codifica IEEE 754 permette anche la rappresentazione di alcuni stati particolari:

- Esponente $(256)_{10} = (1111111)_2$:
 - Mantissa = 0, il valore rappresentato è $\pm\infty$;
 - Mantissa $\neq 0$, il valore rappresentato codifica per la condizione di **NaN** (Not a Number) che si ottiene con operazioni tipo $\frac{n}{0}$;
- Esponente $(0)_{10} = (00000000)_2$, viene rappresentato un numero **denormalizzato**, in tale situazione come esponente viene assunto il valore -126 e davanti la mantissa non si pone 1 ma 0. Tale condizione rappresenta **numeri molto piccoli** ma con meno cifre significative.

RAPPRESENTAZIONE DI UN'INFORMAZIONE NON NUMERICA

Non sempre c'è la necessità di usare il sistema di numerazione binario per rappresentare un numero, ma bisogna impiegarlo nella **risoluzione di generici problemi di codifica** (ad esempio, la codifica dei dodici mesi dell'anno). È possibile **associare ad ogni numero binario un determinato significato**, purché si implementino abbastanza bit da codificare almeno tutte informazioni da codificare; in tale ottica, il codice si dice:

- **Ridondante**, se si usano più bit di quelli strettamente richiesti;
- **Incompleto**, se non si usano tutte le combinazioni di bit a disposizione;
- **Insufficiente**, se si usano meno bit di quelli necessari.

Genericamente è **meglio un codice ridondante o incompleto che insufficiente** perché con essi si può comunque rappresentare l'informazione richiesta. In ottica binaria, se si hanno **n informazioni da codificare** è necessario implementare parole codice di lunghezza:

$$\lceil \log_2 n \rceil$$

Dove $\lceil \quad \rceil$ indica l'**approssimazione per eccesso**.

Un esempio di **codifica non numerica** che sfrutta il codice binario è il codice **ASCII** (American Standard Code for Information Interchange) che associa ad **ogni sequenza di bit un carattere** (stampabile o no) per comunicare una **parola** o una **frase**. Originariamente era a 7 bit, fu poi esteso a **8 bit** per includere caratteri di altre lingue. I caratteri ASCII vengono **raggruppati** in:

- Caratteri di controllo non stampabili, 0-31;
- Caratteri di interruzione e speciali, 32-47;
- Cifre decimali, 48-57;
- Caratteri di interruzione e speciali, 58-64;
- Lettere maiuscole dell'alfabeto inglese, 65-90;
- Caratteri di interruzione e speciali, 91-96;
- Lettere minuscole dell'alfabeto inglese, 97-122;
- Caratteri speciali, 122-127

ARCHITETTURA DEI CALCOLATORI

IL PROCESSORE

Il **processore**, o CPU (Central Process Unit), è il **componente fondamentale** di un calcolatore, quello che permette l'esecuzione di una qualsiasi operazione. Esso è in grado di eseguire un insieme di **azioni elementari** più o meno complesse, suddivise in **istruzioni**; un'istruzione è un comando che specifica il tipo di operazione e su che dati eseguire e comanda il trasferimento in input e output dalla CPU agli altri componenti del calcolatore.

Il processore è in **continuo scambio di informazioni** con la memoria centrale (la **RAM**) perché vi sono **conservati sia i dati di ingresso che le istruzioni** da eseguire, infatti il modello viene detto a "**stored procedure**"; una volta prelevata un'istruzione, essa viene **eseguita singolarmente**, cioè il processore non esegue più di una istruzione alla volta in un **modello gerarchico** dettato dalle varie istruzioni che si susseguono.

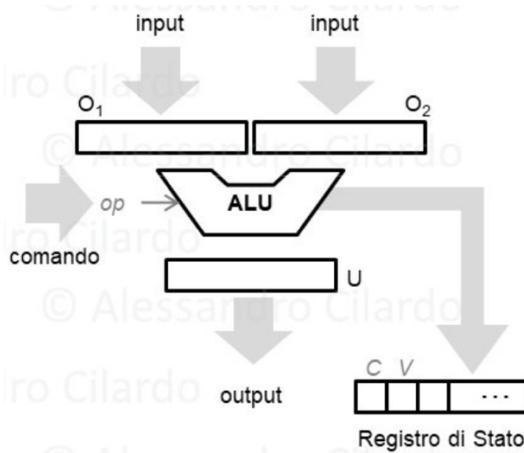
Il processore è composto dai seguenti componenti fondamentali:

- **CU**, o Unità di Controllo, che determina cosa e come il processore deve eseguire un'istruzione (la sequenza di passi da compiere);
- **ALU**, o Unità Logico-Aritmetica, che contiene i circuiti necessari all'esecuzione di problemi di natura logica (porte logiche) o aritmetica (circuiti per la somma e lo shift), essa viene sempre comandata dalla CU;
- **Sezione di collegamento con la RAM**, cioè una serie di circuiti e bus che permette alla CPU di comunicare in ingresso e uscita con la memoria centrale;
- **Sezione di collegamento con le periferiche di I/O**, cioè una serie di circuiti e bus che permettono alla CPU di comunicare con tali dispositivi;
- Una serie di **registri** che semplificano e velocizzano le operazioni alla CU e alla ALU:
 - **PC**, detto program counter o registro prossima istruzione, contiene l'indirizzo di memoria dove è contenuta l'istruzione successiva;
 - **IR**, o instruction register, contiene l'istruzione che il processore sta eseguendo in quel momento;
 - **SR**, o status register, contiene una serie di bit che controllano lo stato di un'operazione (se è stato sollevato overflow, se c'è riporto...);
 - **MA**, memory address, registro che contiene l'indirizzo di un'allocazione di memoria da cui prelevare l'informazione;
 - **MB**, memory buffer, registro che consente il transito di dati da e per la memoria;
 - **Registri ad uso generico**.

Per programmare un processore si usa un **linguaggio macchina** che non è altro che la **codifica binaria delle istruzioni** che può eseguire un processore.

La CU, per far eseguire all'ALU determinate operazioni, deve fornirgli sia gli operandi su cui lavorare e sia un comando che specifici l'operazione da eseguire. In tale ottica, gli **operandi** devono essere **copiati nei registri di ingresso** e il **risultato** dell'operazione nel **registro di output**; alla fine dell'operazione l'ALU andrà anche ad alzare i flag del registro SR nel caso si siano presentate le condizioni.

Per velocizzare le operazioni di copia in input e output dei dati, l'**ALU viene connessa direttamente in memoria** con dei bus dedicati.



Come già anticipato, il processore può servirsi di alcuni registri per eseguire con più facilità e rapidità le operazioni. La CPU può contare su:

- **Registri interni**, sono necessari al funzionamento del processore ma non sono direttamente utilizzabili dal programmatore;
- **Registri di macchina**, che sono visibili al programmatore ed appartengono al modello di programmazione.

Tra i registri di macchina assumono un peso rilevante i **registri generali**, perché permettono di **conservare temporaneamente** dei dati. Invece che tenere i dati in memoria, **il programmatore può utilizzare** i registri generali per conservare **dati di uso frequente**, velocizzando notevolmente le operazioni eseguite su di essi. Al giorno d'oggi usare questi registri significa effettuare operazioni **registro su registro** (dette anche operazioni interne):

$$[R_1] + [R_3] \rightarrow R_4$$

Mentre in passato era anche possibile effettuare operazioni **Memoria su registro o registro su memoria** (memory-to-register e register-to-memory):

$$[R_0] + M[1001] \rightarrow R_0$$

$$M[1001] + [R_1] \rightarrow M[1001]$$

Queste operazioni, definite parzialmente interne, sono possibili solo nel momento in cui **la velocità del processore e della memoria sono dello stesso ordine di grandezza**, cosa che **adesso non accade** (il processore lavora in picosecondi e la memoria cento volte più lentamente). Effettuare delle operazioni interne significa eseguire più istruzioni ma con maggior rapidità.

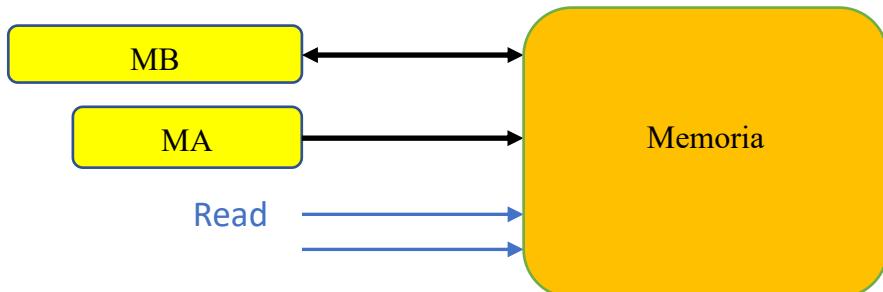
I diagrammi sopra raffigurati indicano delle operazioni aritmetiche da effettuare su registri. Quando un registro è indicato con le parentesi quadre significa che si sta accedendo al suo contenuto, mentre se è indicato senza si sta accedendo allo spazio di memoria. I registri a destra del \rightarrow indicano la lettura e l'esecuzione delle operazioni, a destra si indica la scrittura.

Il collegamento tra memoria **RAM** e **CPU** avviene grazie a dei bus dedicati che permettono un rapido trasferimento dei dati. **Per comunicare**, i due componenti si servono dei due **registri MA e MB**. Accoppiati ad essi si trova un segnale, che specifica se l'operazione è di scrittura o di lettura:

- In un'operazione di **scrittura** il processore consegna alla memoria un indirizzo tramite il MA in cui conservare il dato inserito nel registro MB;

- In un'operazione di **lettura** il processore comunica alla memoria l'indirizzo da cui prendere l'informazione da leggere, che sarà messa nel registro MB, tramite il registro MA.

I bus che codificano per questi due segnali possono essere di ingresso e uscita, nel caso del MB, e solo di uscita nel caso del MA in quanto la memoria non ha l'abilitazione per scrivere un indirizzo sul bus.



LA MEMORIA CENTRALE E IL CICLO DEL PROCESSORE

La **RAM** è un **grande contenitore** dove vengono conservati i dati, funziona analogamente a come funziona il temporaneo mantenimento di un dato nella CPU, cioè con **i registri**; infatti, la memoria non è altro che **un insieme di registri** dove si può scrivere un dato.

La memoria è **organizzata in sequenze di m bit**, dette **locazioni**, dove gli m bit costituiscono una **word**, definita anche come **la quantità di bit che la memoria può gestire contemporaneamente**. Il processore può **accedere** con una sola operazione **ad una locazione** grazie all'uso degli **indirizzi**, delle etichette che permettono a RAM e CPU di identificare una specifica locazione.

In passato le memorie erano costruite a nastro, con un lettore che si doveva spostare dove c'era l'informazione da prelevare; ciò implicava che l'accesso ad un dato dipendeva da dove esso si trovava nel nastro. Ad oggi non accade più così e le **RAM** sono molto più **veloci**; il nome stesso RAM indica che tale pratica non è più in uso, infatti l'acronimo significa **memoria ad accesso casuale** (Random Access Memory) nel senso che il tempo di accesso **non dipende dalla posizione del dato**.

La memoria può eseguire **due operazioni: lettura e scrittura**. Durante le operazioni di lettura la CPU fornisce l'indirizzo dove si trova il dato da leggere, la memoria lo preleva e lo trasferisce, mentre durante le operazioni di scrittura la CPU fornisce alla memoria sia il dato sia l'indirizzo dove scrivere l'informazione.

Tipicamente le **dimensioni della parola** variano tra gli **8** e i **64 bit** ed ospitano in memoria **dati elementari** come numeri binari, numeri in virgola mobile o caratteri. Quando le words sono costituite da **multipli del byte** la memoria viene progettata per essere **byte-addressable**, in cui si associa un **indirizzo ad ogni byte** piuttosto che ad una word.



È importante conoscere **l'ordine con cui questi dati sono memorizzati** nelle words, ovvero sapere **dove è collocato il MSB e il LSB**, in tale ottica esistono due modi con cui memorizzare delle words in memoria:

- **Little-endian**, si memorizzano i byte meno significativi in indirizzi più bassi;
- **Big-endian**, si memorizzano i byte meno significativi in indirizzi più alti.

Ad esempio il dato 11010000 00001110 11110000 00000001 verrebbe memorizzato come segue:

Little-endian				Big-endian (Motorola 68K)			
31	30	29	28	31	30	29	28
11010000	00001110	11110000	00000001	00000001	11110000	00001110	11010000

Ai fini dell'organizzazione del calcolatore, **la scelta tra le due modalità non è rilevante**, tuttavia essa **condiziona il modo in cui sono organizzate le strutture dati dei programmi** e gli accessi in memoria da parte del processore: strutture dati pensate per architetture big-endian non sono immediatamente utilizzabili su architetture little-endian, andrebbe prima fatto un **rovesciamento** che inverta la posizione dei dati nei vari indirizzi e per fare ciò si deve conoscere in dettaglio la specifica organizzazione del calcolatore e della struttura dati.

Un'altra conseguenza dell'organizzazione dei dati in word è il **vincolo sul posizionamento** quando sono composti da più byte. Il vincolo nasce dal fatto che **il processore accede ai byte di una parola con una sola operazione**. Ad esempio, in una memoria che ha word di 32 bit (4 byte), se il processore impone l'allineamento, i dati di 4 byte devono essere posizionati ad indirizzi che siano multipli di 4, altrimenti saranno disallineati

Indirizzo del byte				
3	2	1	0	0
7	6	5	4	4
11	10	9	8	8

Alcuni processori consentono **accessi non allineati**, ma necessiteranno di **più operazioni** per accedere ad un dato.

Può capitare che le **dimensioni di registri interni, bus e registri** di memoria siano **diversi**, solitamente i registri hanno dimensione maggiore o uguale a quella del bus indirizzo ma **non è detto che tutti i bit di un registro indirizzo vengano collegati fisicamente con la memoria**. Ciò significa che lo **spazio di indirizzamento fisico è diverso dallo spazio di indirizzamento logico**, risultando nel fenomeno dell'**aliasing**.

Ad esempio, una memoria con massimo 2^{24} locazioni avrà degli indirizzi di 24 bit e può accadere che il registro indirizzi della CPU abbia 32 bit; ciò significa che gli 8 bit più significativi sono ignorati

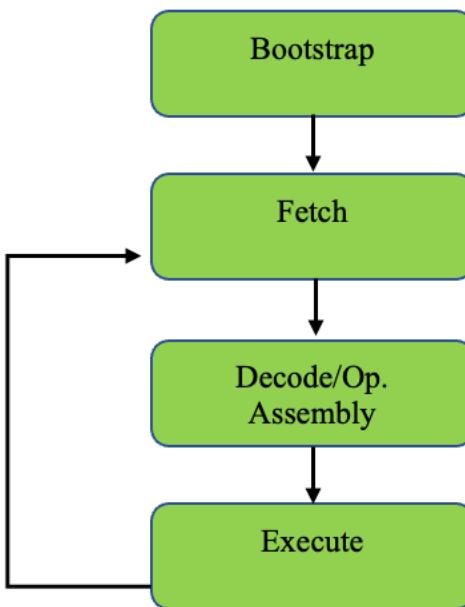
con la conseguenza che **valori diversi contenuti in un registro indirizzo possono attivare la stessa locazione fisica** di memoria, come per gli indirizzi \$0A00A3B2 e \$0000A3B2.

Il modello di Von Neumann è, come già anticipato, un modello costruito sul protocollo “**stored procedure**”, grazie al quale nella memoria **non vengono solo memorizzati i dati** con cui il processore lavora su un programma, ma **anche le istruzioni del programma stesso**, in **ordine sequenziale**. A tal proposito, la memoria viene divisa in due blocchi:

- **Area codice**, la parte della memoria che si occupa di memorizzare in sequenza le istruzioni da eseguire, assegnando un indirizzo per istruzione;
- **Area dati**, la parte della memoria che si occupa di memorizzare i dati con cui il programma deve lavorare.

Poiché queste due parti della memoria sono separate, alcuni registri non indirizzeranno mai a una delle due; ad esempio, i registri **PC e IR indirizzeranno sempre l'area codice**, dal momento in cui contengono un'istruzione o un indirizzo ad istruzione, mentre il **MA indirizzerà sempre l'area dati**, in quanto si occupa di fornire alla memoria l'indirizzo del dato da prelevare e portare alla CPU.

Nella sua attività, **il processore opera in un ciclo continuo** che **inizia con il boot** del calcolatore e **termina con il suo shut-down**. Tale ciclo, detto **ciclo del processore** o **ciclo di Von Neumann** è composto di una serie di fasi e può essere graficamente riassunto come segue:



Esso si compone di diverse fasi:

1. **Fase di bootstrap**, viene avviato il calcolatore con un'operazione inserita in una memoria non volatile caricata inserendo un opportuno valore nel registro PC;
2. **Fase di Fetch**, viene prelevata dalla memoria l'istruzione grazie all'indirizzo inserito nel registro PC e caricata nel registro IR;
3. **Fase di Decode/Operand Assembly**, la CU analizza l'istruzione contenuta nel registro IR e ne ricava il tipo di operazione e l'indirizzo dei dati a cui fare accesso per eseguire l'istruzione, i quali verranno prelevati con un'ulteriore accesso in lettura alla memoria;
4. **Fase di esecuzione**, la CU comunica alla ALU l'operazione e i dati su cui effettuarla e comanda l'invio del risultato ad un registro interno o in memoria.

5. Il PC viene automaticamente aumentato a $PC + k$, dove k è il numero di byte dell'istruzione e si riavvia il processo.

Quest'ultima operazione serve al processore per poter eseguire istruzioni in sequenza; infatti, aumentando l'indirizzo dell'istruzione dei byte che componevano l'istruzione stessa, si va direttamente ad indirizzare l'istruzione successiva. Per i costrutti iterativi o di selezione si implementano particolari istruzioni, dette **istruzioni di salto**, che portano il PC all'indirizzo dell'istruzione necessaria.

CLASSIFICAZIONE DELLE ISTRUZIONI

Un'istruzione è generalmente composta da una **tripla**:

$$i = (f, P_1, P_2)$$

Dove:

- f è un elemento dell'**insieme di codici operativi** eseguibili dal calcolatore, cioè un'operazione definita a **livello del linguaggio macchina**;
- P_1 è l'**insieme di operandi-sorgente**, cioè di valori/indirizzi di registro/indirizzi di locazione su cui l'istruzione f opera;
- P_2 è l'**insieme di operandi-destinazione**, cioè di indirizzi di registro o locazione, dove l'operazione f registra il proprio risultato.

Per comodità, si implementa **un solo operando-destinazione**, che spesso coincide con uno degli **operandi-sorgente**. Dal momento in cui si sta adottando un modello a *stored procedure*, è necessario che sia i **codici operativi** che gli **operandi** siano **registrati in memoria**, di conseguenza è obbligatorio effettuare una **codifica** che permetta ad entrambi gli oggetti di essere **tradotti in binario**.

Con gli operandi è possibile usare la codifica numerica affrontata in precedenza ma per i **codici operativi** si deve ricorrere alla **codifica dello specifico set di istruzioni** definito sul particolare processore. La lunghezza in bit del codice operativo può essere sia a **lunghezza fissa** che a **lunghezza variabile**: nel primo caso si limita la quantità di operazioni codificabili a 2^n (dove n è il numero di bit per istruzione) ma il calcolatore sa che i primi n bit di un'istruzione sono dedicati all'operazione, nel secondo caso si può certamente incrementare il numero di operazioni eseguibili ma al costo di un'informazione in più da comunicare al processore, ossia il **numero di bit che occupa l'operazione** nell'istruzione.

In funzione agli operandi su cui operano, le istruzioni macchina si diversificano per:

- **Tipo di operando**, un intero a 8 bit piuttosto che un carattere;
- **Numero** di operandi impliciti;
- **Natura**, se ad esempio sono costanti o se è il contenuto di un registro piuttosto che di una locazione di memoria;
- **Tecnica di indirizzamento**, se l'operando è隐式的 o esplicito.

Per **operando esplicito** si indica un operando che deve essere indicato nell'istruzione mentre per **implicito** un operando che non deve essere indicato nell'istruzione. Un esempio di operando implicito è il registro accumulatore, a cui si può fare accesso senza esplicitare alcun operando nell'istruzione.

Un'istruzione può avere **diverse forme, in base a quanti operandi espliciti** vengono implementati:

0. In tal caso si fa riferimento ad **operandi impliciti**

$$OP \rightarrow ClearAcc$$

1. L'operando può essere **solo di destinazione**;

$$OP\ O_1 \rightarrow Clear\ R_0$$

2. Gli operandi possono essere **sia di destinazione che di sorgente**;

$$OP\ O_1, O_2 \rightarrow Move\ R_3, R_0$$

3. Pratica **non usata spesso** dal momento in cui si può dichiarare un operando sorgente anche come destinazione.

$$OP\ O_1, O_2, O_3 \rightarrow Add\ R_2, R_3, R_0$$

La classificazione delle istruzioni avviene anche per la loro natura, infatti è possibile riconoscere istruzioni:

- **Memoria-Immediato;**
- **Memoria-Registro;**
- **Memoria-Memoria;**
- **Registro-Immediato;**
- **Registro-Registro.**

Dove il **primo termine indica l'operando destinazione e il secondo l'operando sorgente** (o viceversa in funzione della convenzione adottata dal processore). Questa classificazione è il **caso generale**, ma può accadere che **alcune combinazioni** di operandi **non siano supportate** dal singolo processore.

Un altro **tipo di classificazione** coinvolge i **codici operativi**; infatti, ogni CPU è caratterizzata da un proprio **repertorio di istruzioni macchina**, detto **Instruction Set Architecture (ISA)**. Tale repertorio può essere **più o meno ricco** ma, in generale, tutti possono essere ricondotti a uno di questi due protocolli:

- **CISC** (Complex Instruction Set Computer), nel quale sono presenti **molte istruzioni** che assolvono a **compiti specifici** ma che sono più **lente** nella loro esecuzione;
- **RISC** (Reduced Instruction Set Computer), nel quale sono presenti **poche istruzioni** che assolvono a **compiti generali** e che sono più **veloci** nella loro esecuzione.

In entrambi i casi il repertorio di istruzioni può essere suddiviso in poche **classi di istruzioni fondamentali** ma, sebbene in un protocollo RISC la velocità di esecuzione di un'istruzione è maggiore, per eseguire un determinato compito ci vogliono mediamente più istruzioni di quelle che si userebbero usando un protocollo CISC. Nonostante ciò, tuttavia, il vantaggio di avere una minor quantità di istruzioni con una maggior velocità è stato tale da far prevalere storicamente il protocollo RISC.

Le istruzioni possono essere raggruppate in un set di classi fondamentali:

- **Istruzioni di trasferimento dati**

Sono istruzioni con le quali vengono copiati dati da un operando ad un altro. Tipicamente sono istruzioni a due operandi espliciti ma in altre occasioni possono averne uno隐式 e uno esplicito (nel caso ci sia il registro accumulatore) o uno solo (nel caso delle istruzioni clear). Entrambi gli operandi, sorgente e destinatario, possono essere un qualsiasi registro/locazione ma solo quello sorgente può essere immediato, il destinatario no.

- **Istruzioni aritmetiche**

Sono istruzioni con le quali si effettuano operazioni aritmetiche su operandi numerici con la conseguente registrazione del risultato nell'operando destinazione. In genere è possibile usare un operando sorgente come operando destinazione e costruire un'istruzione con soli due operandi espliciti.

- **Istruzioni logiche**

Istruzioni con le quali è possibile effettuare operazioni logiche booleane bitwise sia unarie che binarie usando delle stringhe di bit. Con questo tipo di operazione è anche possibile effettuare un'operazione di bitmask, cioè combinare una stringa con un'altra prefissata per alterare in modo controllato i suoi bit (AND mette a zero, OR mette a uno e XOR inverte)

- **Istruzioni di scorrimento**

Anch'essi sono effettuati su stringhe di bit ma possono essere sia di tipo logico che di tipo aritmetico (nel caso in cui si preservi il segno). Il verso dello shift può essere o a destra o a sinistra mentre il numero di bit da shiftare può essere fisso ad uno o mobile (specificato da un operando immediato). Uno shift oltre che aritmetico e logico può essere circolare, in tal caso il bit che viene lasciato fuori viene replicato dal lato opposto.

- **Istruzioni di confronto**

Alterano i flag del registro di stato del processore in base all'esito del confronto bit a bit tra due operandi sorgente espliciti (se uno dei due è implicito esso è solitamente 0). In genere questo tipo di operazione è utile nel caso si voglia effettuare un salto condizionato.

- **Istruzioni di salto**

Alterano il flusso sequenziale di istruzioni al fine di poterne eseguire altre che sono lontane da quella in esecuzione (if-else o loop); esse agiscono modificando il registro PC (Program Counter) e, se vengono eseguite solo se è vera una condizione, sono detti salti condizionati. I salti possono essere di due tipi, assoluti (Jump, se contengono anche l'indirizzo destinazione) o relativi (Branch, se contengono uno spiazzamento x con il quale si salta a x locazioni dopo o prima l'attuale valore del PC).

- **Istruzioni di collegamento a sottoprogramma**

Sono un tipo particolare di salto che permette il Jump ad una precisa locazione, collocando il valore del PC in un apposito registro, al quale si farà ritorno con un'istruzione di return dopo l'esecuzione di un preciso blocco di codice.

- **Istruzioni di input/output**

Dotazione esclusiva di alcuni processori, permettono di inviare/ricevere alle/dalle periferiche di I/O. Spesso lo spazio di indirizzamento di queste periferiche è diverso da quello della memoria ma nel caso coincidessero (sistemi con I/O “memory mapped”) si possono sostituire queste istruzioni con un semplice scambio dati.

Una volta scritto un programma con le relative istruzioni **non c'è necessità di scrivere manualmente la codifica di ogni istruzione** e caricarla nel calcolatore, come invece si faceva in passato. In genere, dopo la scrittura di un programma avvengono una serie di **processi di analisi e sintesi** che scompongono e analizzano le singole istruzioni al fine di poter **produrre l'eseguibile codificato**; questo processo **varia in funzione del linguaggio di programmazione** che si sta adottando, ad esempio i linguaggi compilati come il C o il C++ fanno uso di un **compilatore**, mentre linguaggi come quelli **assembly** fanno uso di un **assemblatore**.

Una volta che il compilatore/assemblatore ha **prodotto un file oggetto** da ogni file sorgente, si passa il prodotto al **linker**, che ha lo scopo di **accorpare tutti i file oggetto** (includendo anche eventuali file implementati da librerie esterne) e di **produrre un'eseguibile**. Infine, il **loader** estrae l'immagine dall'eseguibile, accorda eventuali librerie a caricamento dinamico e **carica l'immagine in memoria**, posizionando istruzioni e variabili. **Tale immagine verrà poi eseguita** virtualmente da un simulatore o lanciata su macchina reale ed eseguita passo passo da un debugger.

Tuttavia, il **ciclo di un programma** compilato differisce significativamente dal ciclo di un programma assemblato e ciò è dovuto al fatto che si trova un grosso **gap semantico** tra un linguaggio compilato e un linguaggio assemblato.

LINGUAGGI ASSEMBLER E IL MOTOROLA 68K

Tra i linguaggi in cui può essere scritto il codice sorgente si distinguono quelli **assembler**, linguaggi di **“basso-livello”** che si **avvicinano molto al linguaggio macchina**. Infatti, con questo linguaggio si possono descrivere i **programmi come sequenza di istruzioni** effettivamente **supportate dal processore**, esprimendo concetti molto vicini all'effettiva struttura delle istruzioni in memoria: le “etichette” sono usate come segnaposti per indirizzi in memoria, gli “opcode” mnemonici sono usati per riferirsi alle istruzioni del processore, gli operandi sono quelli su cui realmente il processore va a lavorare ecc...

Non esiste un univoco linguaggio assembly ma esso viene sempre **definito dal costruttore** del processore, il quale implementerà la codifica binaria per ogni istruzione del processore. Ciò significa che se si usa un linguaggio assembly costruito su un processore su un altro processore che ha delle istruzioni in più, si avrà un linguaggio che non può accedere a tutte le istruzioni del processore; tuttavia, ciò non preclude l'esistenza di linguaggi assembly forniti da terze parti. Tra due linguaggi possono variare gli mnemonics (i codici testuali con cui si fa riferimento alle istruzioni), il formato delle linee del sorgente e i formati per specificare i modi di indirizzamento ma non può variare la funzione e l'effetto finale di ogni singola istruzione macchina.

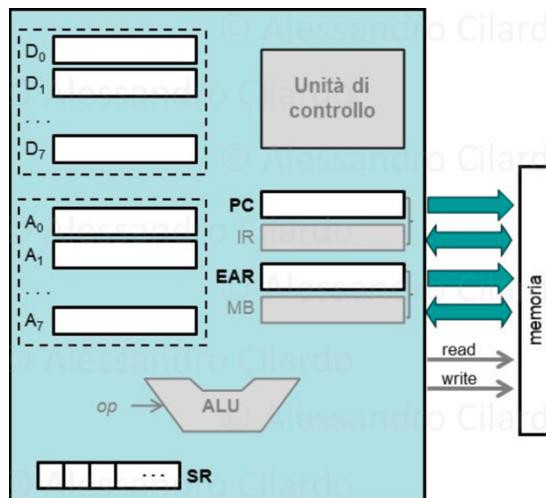
In un linguaggio assembly **una linea di codice corrisponde sempre ad un'istruzione** o, in generale, ad **un elemento disposto in memoria in successione** (che sia esso un dato o un'istruzione), mentre **l'etichetta associata ad ogni linea di codice denota simbolicamente l'indirizzo occupato** in memoria. Ci sono, in realtà, delle **eccezioni** a queste regole:

- **Macro**, una linea assembler che corrisponde a più istruzioni macchina;
- **Pseudo-istruzioni**, una linea assembler che corrisponde a nessuna istruzione macchina (è l'equivalente delle direttive di precompilazione di un linguaggio compilato);
- **Esempi di pseudo-istruzione**:

A differenza dei linguaggi compilati, in cui dichiarare una variabile era sufficiente a poterla utilizzare, nei linguaggi assembly per usare una variabile bisogna prima **allocarla**, cioè decidere quali e quanti locazioni di memoria o registri di CPU utilizzare.

Quando si scrive un programma in assembly è importante gestire e saper usare il **Program Location Counter** (PLC), una **variabile interna dell'assemblatore** che **punta alla locazione di memoria in cui andrà caricata** (a runtime) l'istruzione assemblata. Durante il processo di assemblaggio il PLC viene **incrementato della dimensione di un'istruzione**, in modo da puntare sempre ad una locazione che corrisponde ad un'istruzione; inoltre, è possibile **inizializzarlo ad un indirizzo diverso** usando la pseudo istruzione ORG (origin), mentre per accedere al **valore corrente** si usa il simbolo *. In genere il PLC fornisce un **controllo maggiore sul posizionamento** dei dati in memoria, infatti con esso è possibile **comunicare l'indirizzo da cui iniziare a posizionare** un blocco di istruzioni o di dati.

Il **processore Motorola 68K** è uno dei processori storicamente migliori per capire, a livello didattico, il **funzionamento dei linguaggi assembly**. Al fine di adempiere a tale scopo, di seguito sono riportate le **specifiche tecniche** riguardo tale processore



Il **M68K** è un processore che codifica **dati all'esterno in parole di 16 bit** e **all'interno in registri 32 bit** con **indirizzi di 24 bit per l'esterno** e **32 bit per l'interno**. La differenza di dimensione tra registri interni ed esterni nella rappresentazione degli indirizzi permette, per questo processore, il fenomeno dell'**aliasing**, con uno spazio di indirizzamento logico di 4GB e fisico di 16MB. Poiché la memoria ha parole di 16 bit (cioè 2 byte) è **byte addressable**, vi è stata implementata la **convenzione big-endian** e **ogni parola deve essere allineata ad un indirizzo pari**.

Il **buffer dato** è chiamato **MB** (Memory Buffer) mentre il **buffer indirizzo** è detto **EAR** (Effective Address Register, anche se non è parte del modello di programmazione del processore) e, come già detto, è di 32 bit. All'interno del processore si trovano **16 registri generici**, 8 chiamati **D** e 8 chiamati **A**:

- I **registri D** sono usati per contenere i **dati** in senso generico, con o senza segno;

- I registri A sono usati per contenere gli **indirizzi**, quindi non codificano per numeri con il segno;

Entrambi i tipi di registro possono essere “tagliati” e usati solo per una ridotta quantità di bit (gli A con 16 bit e i D con 8 o 16 bit). Per quanto riguarda i registri A, è possibile gestire **indirizzi a 16 bit** da **0000 a 7FFF** mappandoli su indirizzi a 32 bit nei **primi 32KB** e da **8000 a FFFF** mappandoli su indirizzi di 32 bit negli **ultimi 32KB**.



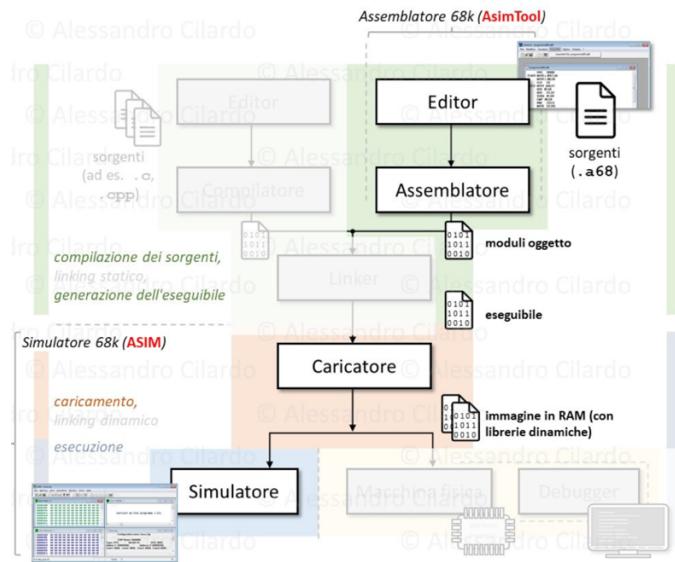
Per i **registri D** il motivo di questo taglio è dovuto al fatto che quando si va a lavorare con numeri di 16 (o 8) bit non è sufficiente lavorare con numeri di 32 bit in cui i 16 (o 24) bit più significativi sono 0 perché non si riuscirebbe a notificare l'overflow al 17esimo (o 9ono) bit sul registro SR; quindi, tagliando il registro a 16 bit **si può far lavorare il registro SR su 16 bit**, con un **risultato più preciso**.

Il registro di stato è composto da 16 bit, ognuno dei quali svolge una funzione precisa, tutti associabili a una di queste classi:

- **Flag di stato;**
 - C Carry;
 - V Overflow;
 - Z Zero;
 - N Negative;
 - X Extra, memorizza il riporto uscente e può essere usato come riporto entrante;
- **Livello di interruzione;**
- **Bit supervisore;**
- **Bit di trace**, tracciamento dell’interruzione ad ogni istruzione.

I **flag di stato** operano **principalmente sui dati**, infatti si possono trovare bit adibiti all’overflow o alla gestione del segno; essi sono una **fotografia dell’ultima operazione aritmetica** e, pertanto, non interverranno mai su stringhe che rappresentano un **indirizzo**.

I programmi scritti sul Motorola 68K saranno eseguiti attraverso un **ciclo di assemblaggio** che fa uso di **AsimTool** come assemblatore e di **ASIM** come simulatore. Il ciclo di questi programmi assembly **non prevede il passaggio per il linker**, infatti dopo l’assemblaggio dei codici sorgenti viene automaticamente prodotto un **eseguibile** da caricare in memoria ed eseguire con il simulatore.



In riferimento al processore in esame, una linea di istruzione assembly sarà composta dai seguenti componenti:

LABEL + TAB + OP.CODE + TAB + OPERANDI + TAB + COMMENTI

- **Label**, o **etichetta**, una stringa alfanumerica che denota un nome per il corrispondente indirizzo;
- **Tab** (caratteri spazio) + **Op.Code**, codice mnemonico o pseudo-operazione, determina un'istruzione in linguaggio macchina o la modifica del Program Location Counter;
- **Tab + Operandi**, oggetti dell'azione specificata dall'Op.Code e variano in sua funzione e in funzione del modo di indirizzamento;
- **Tab + Commenti**, testo arbitrario inserito dal programmatore.

Gli spazi bianchi tra i vari componenti (individuati da Tab) hanno solo uno **scopo organizzativo** e vengono ignorati dall'assemblatore, mentre i **commenti** devono sempre essere preceduti dal simbolo “*”. In queste istruzioni gli **argomenti numerici** si intendono in **notazione decimale**, se non diversamente espresso, o in **notazione esadecimale** se sono preceduti dal simbolo “\$”, mentre il simbolo “#” nell'indicazione degli operandi denota un **indirizzamento immediato** (un valore diretto, senza che si facciano accessi in memoria).

Il linguaggio assembly per il M68K prevede anche l'implementazione di alcune **pseudo-istruzioni**, o **direttive**, cioè delle **istruzioni che regolano il processo di assemblaggio** delle istruzioni e di produzione dell'eseguibile **senza scrivere nulla in memoria**. Alcune di esse sono:

- **ORG** (preceduto da TAB altrimenti è un'etichetta), che si occupa di inizializzare il PLC e di determinare a quale indirizzo sarà posta la successiva sezione di istruzioni o dati;

Esempio: ORG \$8100

- **END**, che viene usata per terminare la fase di assemblaggio e impostare l'entry point del programma (cioè la prima istruzione da eseguire, facendo caricare al loader il relativo indirizzo nel PC), le istruzioni che seguono verranno ignorate dall'assemblatore;

Esempio: END TARGETLAB

- **DS** (Define Storage), viene usato per incrementare il PLC in modo da riservare spazio in memoria per una variabile;

Esempio: LABEL DS.W NUMSKIP

- **DC** (Define Constant), per inizializzare il valore di una variabile;

Esempio: LABEL DC.W VALUE

- **EQU**, usato per definire una costante usata nel sorgente assembler, ogni volta che l'assemblatore incontra l'etichetta associata sa che essa ha un preciso e costante valore.

Esempio: LABEL EQU VALUE

Ognuna di queste pseudo-istruzioni, ma anche le effettive istruzioni, fanno uso di alcuni **suffissi** che determinano **quanto spazio dedicare ad una locazione**; tali suffissi corrispondono al tipo di variabile di un linguaggio compilato, come int o char e sono:

- **B**, 1 byte, 8 bit;
- **W**, 1 word, 2 byte, 16 bit;
- **L**, 1 long word, 2 word, 4 byte, 32 bit.

Se non si specifica alcun suffisso, l'assemblatore interpreta il **caso di default come W**.

Le **Label** sono stringhe di testo **arbitrarie** anteposte ad un'istruzione o ad un dato all'interno del programma assembler. Servono a **riferirsi al particolare indirizzo che contiene** quel dato o quell'istruzione e sono **utilizzati per i salti o per gestire le variabili**. Nel codice che segue, ALOOP serve per riferirsi all'istruzione MOVE, SUM per gestire una variabile e IVAL per una costante:

```

ALOOP    MOVE.W      D0,CNT
        ADD.W       SUM,D0
...
SUM     DS.W        1
IVAL   EQU         17
...

```

Come preannunciato, lo strumento utilizzato per scrivere il codice sorgente in Assembly del M68K si ricorre all'assemblatore **AsimTool**, che **legge i file .a68 e produce un file .lis**; i file prodotti dall'assemblatore saranno poi **inviati al simulatore**, in questo caso **ASIM**, che **produrrà l'output voluto**. Un esempio di codici scritti in assembly è il seguente, dove a sinistra si può notare l'immagine e a destra il codice sorgente:

PLC	contenuto	label	opcode	operands	comments
00000000	00000000	1 * Somma i primi 17 interi			© Alessandro Cilardo
00000000	00000000	2 *			© Alessandro Cilardo
00008000	4279 00008032	3	ORG	\$8000	© Alessandro Cilardo
00008006	3039 00008034	4 START	CLR.W	SUM	© Alessandro Cilardo
0000800C	33C0 00008030	5 ALOOP	MOVE.W	ICNT,D0	© Alessandro Cilardo
00008012	D079 00008032	6	MOVE.W	D0,CNT	© Alessandro Cilardo
00008018	33C0 00008032	7	ADD.W	SUM,D0	© Alessandro Cilardo
0000801E	3039 00008030	8	MOVE.W	D0,SUM	© Alessandro Cilardo
00008024	0640 FFFF	9	MOVE.W	CNT,D0	© Alessandro Cilardo
00008028	66E2	10	ADD.W	#-1,D0	© Alessandro Cilardo
0000802A	4EE9 00008008	11	BNE	ALOOP	© Alessandro Cilardo
00008030	=00008008	12	JMP	SYS08008	© Alessandro Cilardo
00008030	00000000	13 SYS08008	EQU	\$8008	© Alessandro Cilardo
00008030	00000000	14 CNT	DS.W	1	© Alessandro Cilardo
00008032	00000000	15 SUM	DS.W	1	© Alessandro Cilardo
00008034	=00000011	16 IVAL	EQU	ALESSANDRO CILARDO	© Alessandro Cilardo
00008034	0011	17 ICNT	DC.W	IVAL	© Alessandro Cilardo

Symbol Table					
ALOOP	800C	CNT	8030	© IVAL	0011
START	8000	SUM	8032	ICNT	8034

Per il M68K esistono alcune **istruzioni di base**:

- **CLR**, pone il valore 0 nella destinazione;
- **MOVE**, sposta un dato da sorgente a destinazione;
- **ADD**, somma il valore della sorgente alla destinazione;
- **ADDQ**, somma un valore immediato (quick) alla destinazione, dove l'immediato è compreso tra un ristretto range di numeri (8 positivi e 8 negativi zero escluso)
- **SUB**, sottrae il valore della sorgente dalla destinazione;
- **SUBQ**, sottrae un immediato dalla destinazione;
- **CMP**, confronta i due operandi sottraendo sorgente da destinazione (due operandi in memoria occuperebbero troppi bit, uno in memoria e uno a registro è conveniente, sarebbero 64bit contro 48bit);
- **JMP**, salta ad un'istruzione diversa dalla prossima, forzando un valore nel PC;
- **Bcc**, salta ad un'istruzione diversa dalla prossima se è vera l'istruzione specificata dai flag cc.

Per una più dettagliata descrizione della sintassi e delle istruzioni fare riferimento al manuale del processore Motorola 6800.

OPERAZIONI ELEMENTARI IN ASSEMBLY

Un'istruzione di salto non è altro che una **forzatura di un indirizzo nel PC che non corrisponde all'indirizzo successivo in memoria**; il salto **permette di passare da una sequenza di istruzioni ad un'altra** anche quando esse non sono consecutive in memoria e possono essere **"incondizionate"** o **"condizionate"**: nel primo caso **si effettua il salto a prescindere da qualsiasi altra situazione**, mentre nel secondo caso **è eseguito unicamente al verificarsi di una condizione** precisa nel registro di stato.

Per il processore M68K le istruzioni di salto incondizionato sono identificate con **JMP**, mentre quelle di salto condizionato con **Bcc** (Branch on cc) dove **cc** denota **una particolare combinazione di condizioni nel registro di stato** che corrispondono a **diversi esiti possibili di operazioni logico/aritmetiche**. Alcune di queste istruzioni di salto condizionato sono:

- **Single Bit**
 - **BCS**, branch on carry set ($C=1$);
 - **BCC**, branch on carry clear ($C=0$);
 - **BVS**, branch on overflow set ($V=1$);
 - **BVC**, branch on overflow clear ($V=0$);
 - **BEQ**, branch on equal (zero) ($Z=1$);
 - **BNE**, branch on not qual ($Z=0$);
 - **BMI**, branch on minus ($N=1$)
 - **BPL**, branch on plus ($N=0$)
- **Signed**, riflette l'esito di una **CMP**
 - **BLT**, branch on less than (zero) ($N \oplus V = 1$);
 - **BGE**, branch on greater than or equal ($N \oplus V = 0$);
 - **BLE**, branch on less than or equal ($(N \oplus V) + Z = 1$);
 - **BGT**, branch on greater than ($(N \oplus V) + Z = 0$);
- **Unsigned**
 - **BLS**, branch on lower than or same ($C + Z = 1$);

- **BHI**, branch on higher than ($C+Z=0$);
- **BRA**, branch always

La **differenza** sostanziale tra una **JMP** e una **BRA** è l'**offset di indirizzi a cui si salta**, nel primo caso il salto avviene **posizionando nel PC l'indirizzo** dell'etichetta con **l'istruzione a cui saltare**, **senza calcolare quanto è lungo il salto**, mentre nel secondo caso si **incrementa il PC con la differenza di istruzioni** che ci sono tra la BRA e l'istruzione a cui saltare. Ovviamente la **BRA ha un range di movimento limitato** rispetto alla JMP, anche se la dimensione dell'offset può essere variata tra .S, .B e .L.

Un **vettore** (analogamente per le **matrici**) è una **sequenza ordinata di dati dello stesso tipo**. In memoria, essi **vengono gestiti** considerando i **dati contenuti in allocazioni adiacenti**; ad esempio, un vettore (come quello allocato qui sopra) di 5 elementi che viene allocato all'indirizzo \$8100 che occupa 10 byte di memoria occuperà i cinque spazi:

00008100 2A11 2FB1 CC92 3E40 A119
0000810A ----- 10 (5 x 2) byte -----

Per fare questo tipo di operazione in assembly si usa la **direttiva DC .W** dove il suffisso viene scelto in funzione della situazione.

Il **Motorola 68K** mette a disposizione alcune **istruzioni** per le **operazioni di shift**, che possono essere di diverse tipologie:

- **Shift logico** a sinistra;

LSL.W #2, D2

- **Shift logico** a destra;

LSR.W #2, D2

- **Rotazione** a sinistra;

ROL.W #2, D2

- **Rotazione** a destra;

ROR.W #2, D2

- **Shift aritmetico** a sinistra,

ASL.W #2, D2

- **Shift aritmetico** a destra.

ASR.W #2, D2

Con le **operazioni di test** è possibile **verificare particolari condizioni di un operando e marcarne l'esito nel registro di stato**, per poi successivamente eseguire delle operazioni in loro funzione; esse sono:

- **Test**, verifica se l'operando è negativo o zero;

TST D2

- **Bit Test**, verifica se un determinato bit è 1;

BTST #1, D2

- **Bit Change**, testa un bit e lo inverte;

BCHG #2, D2

- **Bit Clear**, testa un bit e lo pone a 0;

BCLR #2, D2

- **Bit Set**, testa un bit e lo pone a 1;

BSET #2, D2

- **Test and Set**, influenza i bit N e Z dello SR sulla base del valore dell'operando e scrive subito dopo il valore 128 (cioè alza il bit in posizione 7, $2^7 = 128$); l'uso dello SR permette di sapere quale fosse il valore precedentemente contenuto nell'operando (la lettura e la scrittura sono fatte in maniera atomica). Si può notare che la dimensione di un operando è solo di un byte.

TAS.B (A1)

Con le **operazioni di scambio** è possibile **scambiare il contenuto di due registri oppure la parte bassa e la parte alta di un solo registro**, rispettivamente con le istruzioni:

- **Exchange**;

EXG D3, D2

- **Swap**.

SWAP D2

Spesso occorre effettuare **incrementi o decrementi di piccole quantità** ma un'istruzione può codificare **valori immediati solo su 16 o 32 bit**, il che risulta in uno **spreco di memoria**. Per risolvere questo problema sono state introdotte le istruzioni:

- **Add Quick**;

ADDQ #2, D2

- **Sub Quick**.

SUBQ #2, D2

In questo caso **l'immediato è codificabile su tre bit**, contemplando valori da 1 a 8 (lo 0 non serve). In genere, quando si lavora con gli immediati **è possibile far scegliere all'assemblatore la codifica migliore** disponibile per quel tipo di operandi; per fare ciò sono introdotte le seguenti istruzioni:

- **ADDI**, aggiunge un immediato ad un operando destinazione;

ADDI #2, D2

- **SUBI**, sottrae un immediato ad un operando destinazione;

SUBI #2, D2

- **ANDI, ORI, EORI**, applicano all'immediato le corrispondenti funzioni logiche;

ANDI #\$FE, (A2)

- **CMPI**, confronta un immediato con un operando destinazione.

CMPI #100, D2

Sebbene per la **moltiplicazione** ci sia l'istruzione di shift, è possibile effettuare questa operazione (come la divisione) anche **con un'istruzione proprietaria**. L'operazione viene sempre effettuata su 16 bit in modo da richiedere al massimo 32 bit:

- **MULU**, moltiplica due operandi unsigned;

MULU (A2), D2

- **DIVU**, divide due operandi unsigned;

DIVU (A2), D2

- **MULS**, moltiplica due operandi signed;

MULS D3, D2

- **DVIS**, divide due operandi signed.

DIVS D3, D2

È anche possibile **manipolare degli indirizzi**, in particolare **trasferendoli o comandoli**:

- **MOVEA**, trasferisce registri Ax non alterando il registro di stato;

MOVEA.L #\$8100, A2

- **ADDA**, applica similmente l'addizione su un registro indirizzo.

ADDA #\$10, A2

- **MOVEM**, trasferisce in blocco interi gruppi di registri (infatti è detto trasferimento multiplo) provenienti da qualsiasi sottoinsieme di registri generici Dx/Ax e si utilizza in combinazione con una modalità di indirizzamento con autoaggiornamento.

MOVEM.L D2-D4/A0/A2-A4, - (A7)

MOVEM.L (A7) +, D0-D4/A0/A2-A4

Esiste una variante della CMP che consente confronti con entrambi gli operandi in memoria:

- **CMPM**, con essa è ammessa solo la modalità di indirizzamento indiretto con post-incremento ed è tipicamente usata per confrontare vettori (sequenze di locazioni tutte della stessa dimensione).

CMPM (A4) +, (A3) +

Per gestire in modo sintetico i cicli è possibile sostituire con un'unica istruzione il decremento di un contatore e la verifica di una condizione, operazioni che normalmente richiedono più istruzioni:

- **DBcc**, fintantoché la condizione cc rimane falsa
 - Decrementa il registro Dx fornito;
 - Se non vale -1 (cioè se non era 0 prima del decremento) salta all'etichetta fornita all'istruzione
 - In altri casi passa all'istruzione seguente

DBT D0, LOOP

Supporta tutti i cc del comune Bcc e ammette le forme DBF e DBT, dove T è true e F è false, per ignorare la condizione ed usare solo il registro di conteggio.

È possibile usare un'istruzione per scrivere tutti 0 o tutti 1 a seconda di una determinata condizione:

- **Scc**, controlla la condizione e, se verificata, pone tutti i bit dell'operando (solo di un byte, non si possono usare W o L) ad 1, in caso contrario a 0

SEQ D2

In questo esempio se l'operazione precedente (tipo una CMP) ha dato esito EQ, gli 8 bit meno significativi di D2 sono posti a 1.

È possibile modificare il registro di stato e porre il processore in una condizione di attesa sfruttando il meccanismo delle interruzioni con l'istruzione STOP:

STOP #001000000010010

Mentre con l'istruzione NOP, si comanda di non fare nulla; è un'istruzione vuota, talvolta usata per il debugging o per occupare parti dell'area del codice di sviluppo con spazio vuoto. NOP non esegue alcuna operazione e permette all'unità di esecuzione di oziare.

MODI DI INDIRIZZAMENTO

Per **indirizzamento** si indica il modo con cui la CPU accede agli operandi specificati in un'istruzione, che sono sempre degli **indirizzi** (EA, Effective Address): nel caso di un dato da manipolare, l'indirizzo sarà quello dove il dato è registrato, nel caso di un'istruzione di salto sarà l'indirizzo dove è registrata l'istruzione a cui saltare.

Esistono **numerosi metodi** di indirizzamento ma nessun processore li supporta tutti, il Motorola 68K ne supporta una buona parte. I **principali** sono:

- **Register direct**
 - Data-register direct
 - Address-register direct

È il modo **più semplice**, infatti il ridotto numero di registri permette di codificarne l'indice nei 16 bit dell'istruzione. Con tale metodo **l'operando sorgente e/o l'operando destinazione** (su cui vengono effettuate operazioni di lettura e sovrascrittura) sono entrambi **registri dato (D)** o **indirizzo (A)**; in molte istruzioni a due operandi uno dei due registri può anche fare da operando destinazione, effettuando prima un'operazione di lettura e dopo di sovrascrittura.

È un modo di indirizzamento **veloce**, in quanto **non effettua accessi in memoria**, e **leggero**; infatti, fa uso di **istruzioni molto corte**: usa **tre bit** per specificare **uno degli otto registri** da usare (reg) e **un bit** per la **scelta del tipo di registro** (mode). Ad esempio:

$$mod = 0; reg = 1 - 7 \Rightarrow D0 - D7$$

$$mod = 1; reg = 1 - 7 \Rightarrow A0 - A7$$

Per codificare un'istruzione elementare bastano solo **16 bit**. L'utilizzo più frequente di questo indirizzamento viene chiamato “*scratchpad storage*” perché fa uso di **variabili usate frequentemente** e, pertanto, **contenute all'interno del circuito del processore**.

- **Immediate (or Literal)**

Il **valore dell'operando** su cui lavorare è **direttamente specificato nell'istruzione**, senza il passaggio per un registro di memoria; tale operando deve **necessariamente essere di sorgente**, dal momento in cui **costituisce una costante** sulla quale non si possono effettuare operazioni di scrittura. In un'istruzione viene praticamente indicato con il simbolo #.

Anche questo modo di indirizzamento può essere **scritto in soli 16 bit di istruzione**, ma solo in alcuni specifici casi: **quando la costante è di dimensioni ridotte** (come nel caso della ADDQ e parenti), **non sono necessarie altre word aggiuntive** per codificare il literal oltre la parola codice, **l'operando destinazione è un indirizzo o quando non sono richiesti altri (lenti) accessi in memoria**; tuttavia, in generale, se la costante è abbastanza lunga o se l'istruzione non supporta la versione “quick” è necessario usare una o più word aggiuntive che seguono la parola codice.

- **Absolute (or direct)**
 - Short
 - Long

L'istruzione fornisce **direttamente l'indirizzo di memoria** da cui prelevare l'operando ed è il modo più semplice di specificare un indirizzo di memoria completo. Un'istruzione che fa uso di questo

modo di indirizzamento **richiede due accessi in memoria**: il primo relativo al **prelievo dell'istruzione stessa** e il secondo relativo al **prelievo dell'operando**.

- **Address-Register indirect**

Nell'istruzione viene **specificato il registro Ax** che contiene l'effettivo indirizzo dell'operando, il processore non deve fare altro che **accedere all'area puntata**.

- **Auto-Decrement**

È sempre un **modo di indirizzamento indiretto** (come il precedente) ma se il modo di indirizzamento è specificato come – (An), il contenuto del registro indirizzo è **automaticamente decrementato di una quantità pari alla dimensione dell'operando prima dell'uso** (pre-decremento). Ad esempio, nel seguente codice si sottrae ad A0 la dimensione dell'indirizzo, .W = 2 byte, effettuando un push di D0 sullo stack puntato da A0:

```
MOVE.W D0, - (A0)
```

- **Auto-Increment**

Analogamente al caso precedente, **il contenuto del registro indirizzo è automaticamente incrementato di una quantità pari alla dimensione dell'operando dopo dell'uso** (post-decremento). Nell'esempio seguente si aggiunge ad A0 la dimensione dell'indirizzo, .W = 2 byte, effettuando un pop in D0 sullo stack puntato da A0:

```
MOVE.W (A0) +, D0
```

Alcuni modi di indirizzamento non sono implementati nel Motorola 68K per mancanza di componenti interni; ad esempio, in processori come **Intel 8086**, il quale supporta **registri dedicati al calcolo dell'EA** (registri base BP/BX e registri indice SI/DI), sono possibili speciali modalità basate su combinazioni di tali registri ed un eventuale spiazzamento:

- **Indexed**

La posizione dell'operando è ottenuta sommando un registro indice (SI o DI) con uno spiazzamento a 8 o 16 bit. Questo metodo è adatto per **accedere a dati strutturati** presenti in memoria, come ad esempio array.

- **Based**

L'EA dell'operando è ottenuta sommando un registro base (BP o BX) con uno spiazzamento a 8 o 16 bit.

- **Based Indexed;**
 - Short;
 - Long;

Il primo elemento della somma fa da **base address**, mentre **il secondo** funge da **indice** e consente di **calcolare a tempo di esecuzione la posizione iniziale e quella relative in tabelle o array**. Questa modalità è disponibile anche con uno spiazzamento a 8 o 16 bit.

Tornando al **Motorola 68K**, il processore supporta varie **modalità di indirizzamento con indice, base e spiazzamento**:

- **Indiretto con spiazzamento**

È usato un **registro indirizzo Ax con l'aggiunta di uno spiazzamento**, rappresentato su **16 bit con segno** nonostante venga esteso a 32 bit prima della somma con il registro.

```
MOVE.B 6(A0),D0
```

In questo caso la MOVE accede ad un indirizzo ottenuto sommando la costante 6 al contenuto del registro A0.

- **Indiretto con indice**

Somma due registri (il primo di tipo Ax) **per ottenere l'EA dell'operando**. Tale metodo è adatto per **accedere a strutture dati regolari** presenti in memoria.

```
MOVE.B (A0,D2),D0
```

In questo caso la MOVE accede ad un indirizzo ottenuto sommando al contenuto del registro A0 il contenuto del registro D2.

- **Indiretto con spiazzamento e indice**

Somma due registri (il primo di tipo Ax) **per ottenere l'EA dell'operando ma viene anche aggiunto uno spiazzamento ad 8 bit**. È adatto per accedere a **dati strutturati** presenti in memoria **secondo schemi più complessi** (ad esempio **array di struct**, in tal caso i due registri sono usati per localizzare l'indirizzo di partenza del record e lo spiazzamento per localizzare il campo specifico del record).

```
MOVE.W $6C(A0,D2),D0
```

La MOVE accede ad un indirizzo ottenuto sommando la costante 0x6C al contenuto del registro A0 ed al contenuto del registro D2.

- **Relativo (al Program Counter)**

Il calcolo dell'indirizzo è relativo al PC, cioè calcolato per differenza rispetto all'indirizzo dell'istruzione attualmente eseguita; in questo modo si calcola **l'EA come somma di uno spiazzamento fisso** (spesso piccolo, 8 o 16 bit, per specificare indirizzi "vicini" anziché ricorrere a indirizzi assoluti di 32 bit) specificato nell'istruzione e **del valore corrente del PC**. Può essere anche usato per **saltare ad aree di memoria Read Only**, infatti non può essere usato per dati che potrebbero essere modificati e, pertanto, è spesso trovato usato per le istruzioni piuttosto che per i dati.

```
MOVE.W $6C(PC,D2),D0
```

La MOVE accede ad un indirizzo contenuto sommando la costante 0x6C al contenuto del PC ed al contenuto del registro D2.

Quando c'è bisogno di **caricare in un registro Ax un Effective Address (EA)** viene in aiuto l'istruzione **LEA** (Load Effective Address) che **permette di evitare di far calcolare direttamente**

all'istruzione l'EA; infatti, l'EA è **dapprima generato e in seguito salvato nel registro Ax scelto.** Nel caso di **indirizzi costanti** (che non vanno calcolati) **non è particolarmente utile** tale istruzione, che può essere tranquillamente sostituita da una MOVE:

LEA \$0010FFFF, A5

MOVE.L #0010FFFF, A5

Dove le due istruzioni differiscono per la sola considerazione dell'indirizzo sorgente: se lo si fosse specificato nel secondo caso non come un immediato (cioè come è stato fatto per il primo esempio) si sarebbe solo copiato il contenuto di tale indirizzo nel registro destinazione.

Si suppone di avere un ciclo in cui viene ripetutamente usata una modalità di indirizzamento complessa e in cui l'EA non cambia durante il ciclo. **L'istruzione LEA è utile perché permette di effettuare una sola volta l'indirizzamento**, caricando in un registro indirizzo il risultato di tale operazione. Ad esempio:

LEA \$1C(A3,D2), A5

...

ADD.W (A5), D0

COSTRUTTI PER IL CONTROLLO DI FLUSSO

Nei linguaggi di alto livello le **strutture di controllo** vengono tradotte automaticamente dal compilatore in righe assembly coerenti con lo scopo del codice; si possono distinguere **cinque strutture di controllo essenziali** e si possono rappresentare le rispettive **codifiche (individuate da specifici pattern)** in assembly:

- **Struttura di selezione, if**

Questa struttura è caratterizzata da una **biforcazione rappresentata dalla condizione**, la quale se **vera** comanda l'**esecuzione di un blocco di istruzioni**, altrimenti lascia scorrere il programma nel suo flow.



In C/C++:

```
if (CONDIZIONE == TRUE) {
```

BLOCCO DI CODICE

}

BLOCCO SUCCESSIVO

In assembly:

VALUTA CONDIZIONE

B (NOT CONDIZIONE) SKIP

BLOCCO DI CODICE

SKIP BLOCCO SUCCESSIVO

Ad esempio:

C/C++	Assembly
if (D0 == 5) {	CMPI.L #5, D0
D1++;	BNE SKIP
}	ADDQ.L #1, D1
D2 = D0	SKIP MOVE.L D0, D2

- **Struttura di selezione, if – else**

Questa struttura è **analogia alla precedente** ma la **biforcazione** permette l'esecuzione di uno solo tra due blocchi di codice, individuati dalla **falsità** o dalla **verità della condizione**, e la successiva ripresa del flusso del codice.



In C/C++:

```
if (CONDIZIONE == TRUE) {  
    BLOCCO DI CODICE (CONDIZIONE == TRUE)  
} else {  
    BLOCCO DI CODICE (CONDIZIONE == FALSE)  
}  
  
BLOCCO SUCCESSIVO
```

In assembly:

```
VALUTA CONDIZIONE  
  
B (NOT CONDIZIONE)      SKP1  
  
BLOCCO DI CODICE (CONDIZIONE == TRUE)  
  
BRA SKIP2  
  
SKP1 BLOCCO DI CODICE (CONDIZIONE == FALSE)  
  
SKIP2 BLOCCO SUCCESSIVO
```

Ad esempio:

C/C++	Assembly
if (D0 == 5) {	CMPI.L #5,D0
D1++;	BNE SKP1
} else {	ADDQ.L #1,D1
D1--;	BRA SKP2
}	SKP1 ADDQ.L #-1,D1
D2 = D0	SKP2 MOVE.L D0,D2

- **Struttura di iterazione, do – while**

Questo costrutto permette l'esecuzione ripetuta di un blocco di codice in funzione di una particolare condizione la quale, se vera, permette la reiterazione dello stesso blocco. In particolare la verifica della condizione avviene dopo l'esecuzione del blocco stesso e le n iterazioni vanno da 0 a n-1 (ovvero il blocco verrà eseguito almeno una volta).



In C/C++:

```

do {
    BLOCCO DI CODICE
} while (CONDIZIONE == TRUE)
BLOCCO SUCCESSIVO
  
```

In assembly:

```

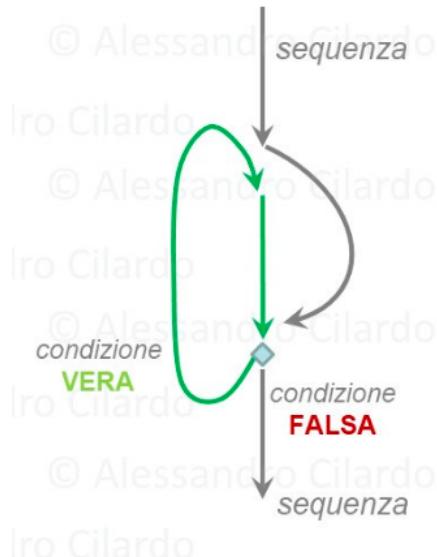
LOOP BLOCCO DI CODICE
VALUTA CONDIZIONE
B (CONDIZIONE) LOOP
BLOCCO SUCCESSIVO
  
```

Ad esempio:

C/C++	Assembly
D0 = 1;	MOVE.B #N,D2
D1 = 1;	MOVE.B #1,D1
N = 4;	MOVE.W #1,D0
do {	LOOP MULU.W #3,D0
D0 = D0 * 3;	ADDQ.B #1,D1
D1++;	CMP.B D2,D1
} while(D1 <= N);	BLE LOOP

- **Struttura di iterazione, while**

È molto simile alla precedente solo che la verifica della condizione avviene prima di eseguire il blocco, quindi le n iterazioni vanno da 1 a n (il blocco di codice può anche non essere eseguito).



In C/C++:

```
while (CONDIZIONE == TRUE) {
    BLOCCO DI CODICE
}
BLOCCO SUCCESSIVO
```

In assembly:

```
BRA COND
LOOP BLOCCO DI CODICE
COND VALUTA CONDIZIONE
B (CONDIZIONE) LOOP
BLOCCO SUCCESSIVO
```

Ad esempio:

C/C++	Assembly
D0 = 1;	MOVE .B #N, D2
D1 = 1;	MOVE .B #1, D1
N = 4;	MOVE .W #1, D0

```

while (D1 <= N) {
    D0 = D0 * 3;
    D1++;
}
BRA CND
LOOP MULU.W #3,D0
ADDQ.B #1,D1
CND CMP.B D2,D1
BLE LOOP

```

- Struttura di iterazione, for

Questo costrutto non è altro **che una variante del costrutto while**, eseguito su condizioni numeriche piuttosto che booleane. A livello di codice, un for può essere scritto come while:

```

int i = 0
while(i <= N) {
    BLOCCO DI CODICE
    I++
}
BLOCCO SUCCESSIVO
for(int i = 0; i++; i <= N) {
    BLOCCO DI CODICE
}
BLOCCO SUCCESSIVO

```

Ma se si osserva il **diagramma di flusso** si può notare che **le due strutture coincidono**:

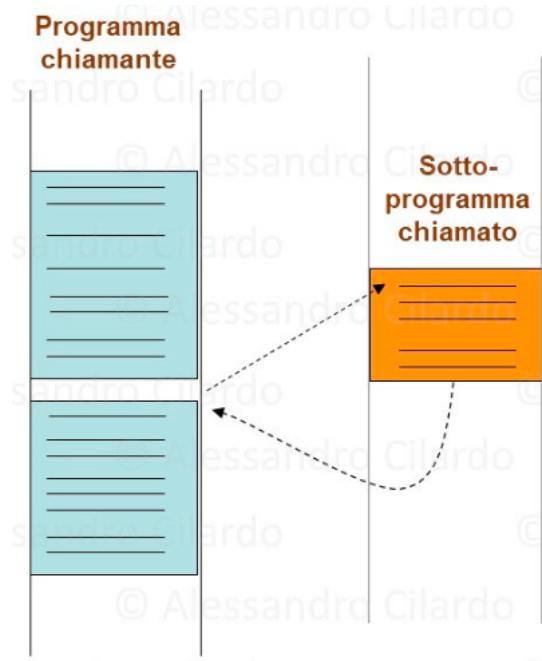


Pertanto, **in assembly non c'è differenza tra uno e l'altro**.

SOTTOPROGRAMMI

Nel momento in cui un **blocco di codice** va ripetuto diverse volte, è utile effettuare un'**astrazione procedurale**, consentendo l'**organizzazione modulare dei programmi**. La pratica della **modularità**

prevede la **creazione di sottoprogrammi**, o subroutine, per le quali un **segmento di codice** (che svolge un'elaborazione su dati forniti in input e restituisce, eventualmente, dati in output indipendentemente da altri segmenti) viene **astratto e fornito tramite un'interfaccia** con la quale è possibile **richiamare ed eseguire diverse volte il segmento stesso** all'interno di un programma.



Un sottoprogramma **scambia con il programma chiamante dei dati sia in input che in output**, che vengono chiamati **parametri**. Nel testo che specifica il sottoprogramma, tali parametri sono identificati da **nomi simbolici**, che **non sono relativi agli effettivi dati del programma chiamante** e, pertanto, sono detti **parametri formali**; nel momento in cui **il sottoprogramma è attivato** (o invocato), esso richiede che **ai parametri formali siano assegnati degli opportuni valori** che il chiamante può usare e che sono detti **parametri effettivi**.

Il modo in cui un calcolatore rende **possibili le operazioni di chiamata e ritorno delle procedure**, gestendo opportunamente gli indirizzi delle istruzioni, è detto **collegamento** (o **linkage**), mentre per **passaggio dei parametri** si intende **l'insieme di convenzioni sulle modalità di scambio dei parametri**, che definiscono:

- Quali informazioni si scambiano il chiamante e il sottoprogramma, cioè il **cosa**;
- Attraverso quali meccanismi avviene lo scambio (memoria, registri...), cioè il **come**.

Nei linguaggi di alto livello, l'elenco di parametri va definito nella dichiarazione della procedura, in particolare nella sua interfaccia:

```
void proced(int a, int b, bool *p);
```

Mentre i **meccanismi di scambio** sono **convenuti nel linguaggio**, la convenzione è **implementata dal compilatore**. In assembly, invece, è **il programmatore che stabilisce e implementa le convenzioni** (a meno che non si debbano scrivere moduli che interagiscono con moduli di altri linguaggi di alto livello, in tal caso è necessario che il programmatore rispetti le convenzioni dei relativi compilatori).

In alcuni linguaggi di alto livello si fa la distinzione tra:

- **Funzioni**, sottoprogrammi al cui nome è associato un valore di ritorno assegnabile ad una variabile

```
a := max(b, c);
```

- **Procedure**, sottoprogrammi che invece compiono solo un'elaborazione sui parametri di scambio ma non hanno valori di ritorno

```
stampaVett(vettore, n);
```

In C/C++ questa convenzione non è adottata e tutti i sottoprogrammi sono detti funzioni, anche se le procedure possono essere espresse come funzioni con un dato di ritorno di tipo `void`.

L'esecuzione di un sottoprogramma avviene mediante i seguenti passi:

1. **Scrittura dei parametri effettivi** secondo la convenzione stabilita per lo scambio;
2. **Trasferimento del controllo** al sottoprogramma (la cosiddetta “chiamata”);
3. **Esecuzione** del sottoprogramma;
4. **Scrittura dei parametri di output**;
5. **Trasferimento del controllo** al programma chiamante (“ritorno dal sottoprogramma”) ed esecuzione dell’istruzione successiva a quella di chiamata.

In C/C++ un sottoprogramma può essere scritto e chiamato come segue:

```
#include <iostream>
using namespace std;

// Interfaccia del sottoprogramma
float calcolaDet(float *f, int n);

// Programma chiamante
Int main() {
    ...
    // Chiamata con passaggio di parametri effettivi
    det = calcolaDet(myMat, dim);
    return 0;
}

// Sottoprogramma, dichiarato con parametri formali
float calcolaDet(float *f, int n) {
    ...
    // Valore di ritorno
    return f
}
```

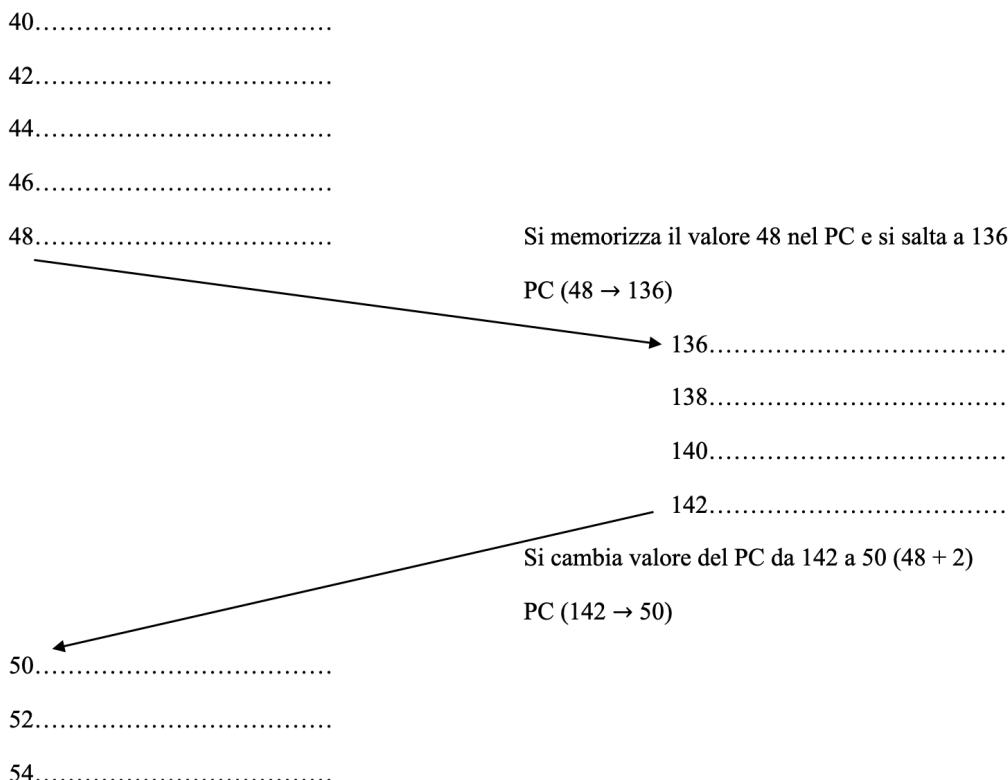
La chiamata di un sottoprogramma interrompe l'esecuzione sequenziale del programma, infatti rappresenta un **particolare caso di istruzione di salto**: l'**indirizzo della prima istruzione della subroutine** (ovvero il suo entry point) viene **caricato nel PC** ma, a differenza di una normale istruzione di salto, è **necessario un meccanismo per il ritorno al chiamante**. Pertanto si individuano le seguenti istruzioni:

- **Istruzioni di salto a sottoprogramma** (Jump To Subroutine, o call), che salvano il valore corrente del PC per consentire il ritorno al chiamante ma allo stesso tempo effettuano un salto all'entry point del sottoprogramma;
- **Istruzioni di ritorno da sottoprogramma** (Return From Subroutine), che ripristinano il valore del PC salvato per realizzare un salto al programma chiamante.

Il valore del PC che viene salvato dall'istruzione di salto a sottoprogramma può essere **memorizzato in un apposito registro, il Link Register (LR) e/o in una opportuna area di memoria chiamata stack**. Nel Motorola 68K le istruzioni per la chiamata di subroutine sono:

`jsr label, bsr label`

Il **meccanismo** con cui queste due istruzioni lavorano è:



Il problema del ritorno all'istruzione successiva a quella di chiamata a sottoprogramma è detto **problema di collegamento del sottoprogramma (subroutine linkage)** e sono possibili quattro soluzioni:

- L'indirizzo di ritorno è **salvato in un registro macchina**;
- L'indirizzo di ritorno è **salvato in una locazione di memoria** (ad esempio nelle locazioni immediatamente precedenti l'entry point della subroutine);
- L'indirizzo di ritorno è **salvato in una locazione di memoria associata al programma chiamante**;
- L'indirizzo di ritorno è **salvato in un area di memoria separata, lo stack**.

In alcune architetture un'istruzione di **jump&link** carica l'indirizzo di ritorno nel **Link Register (\$ra)** prima di cambiare il valore del PC e un'istruzione reciproca effettua il ritorno da sottoprogramma:

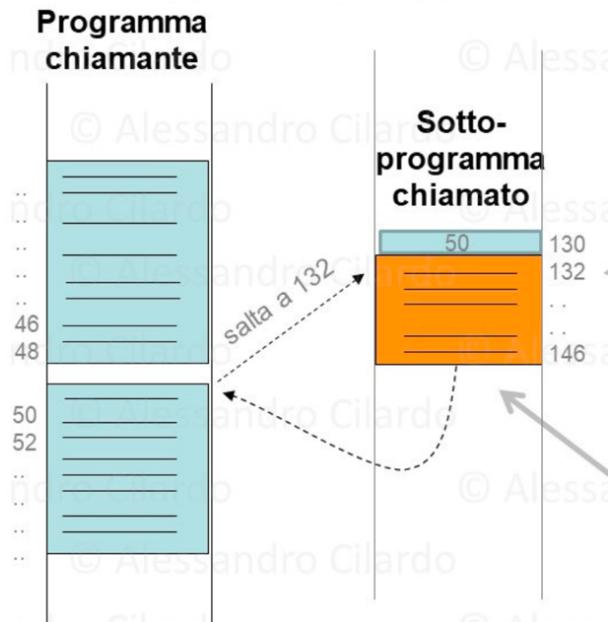
```
jar label = [PC] → $ra ∧ label → PC
jr $ra = $ra → PC
```

Tuttavia, l'annidamento e la ricorsione richiedono il salvataggio del Link Register in una memoria esterna.

Per quanto riguarda il **salvataggio in una locazione di memoria**, esso può avvenire in **due modi**:

- **Il sottoprogramma** alloca un'area di memoria dedicata all'indirizzo di ritorno;
- **Il programma chiamante** dichiara un'area di memoria adibita al salvataggio dell'indirizzo di ritorno e ne comunica la posizione al sottoprogramma.

L'assembler 68K, quando incontra una chiamata a sottoprogramma, **salva l'indirizzo successivo nella locazione riservata in testa al sottoprogramma** (in questo caso salva l'indirizzo 50 in 130, dove 132 è l'entry point della subroutine); al termine dell'esecuzione, il sottoprogramma legge l'indirizzo 50 dall'apposita locazione e lo ripristina nel PC, determinando il ritorno al programma chiamante.



In assembly risulterebbe:

```
MAIN MOVE.L    #RET,SUBR * Salva il return prima dell'entry point
        JMP   SUBR+4    * Salta al sottoprogramma
RET    istruzioni successive

SUBR DS.L 1      * Locazione riservata per contenere il ritorno
                  Istruzioni del sottoprogramma
MOVEA.L   SUBR,A0
JMP   (A0) * Salta all'istruzione di ritorno
```

La convenzione per la quale l'indirizzo di ritorno è salvato nelle locazioni di memoria precedenti l'entry point del sottoprogramma **presenta degli inconvenienti**, ad esempio **non è applicabile se il codice della subroutine è in ROM e non consente chiamate multiple, l'annidamento o la**

ricorsione. In quest'ultimo caso si otterrebbe la sovrascrizione dell'entry point, che non permetterebbe il ritorno da sottoprogramma della prima chiamata.

Il modo più efficace per gestire aree di memoria per lo scambio dei parametri è lo **stack**, dove i dati sono allocati nella prima area di memoria disponibile. In particolare **in una memoria stack ogni elemento è memorizzato nella locazione successiva a quella in cui è stato posto il dato oggetto della precedente memorizzazione**; il nome prende dal modo in cui i dati si organizzano dopo una serie di memorizzazioni, infatti essi **si vanno a disporre come una pila** (in inglese stack) dove vige il **protocollo LIFO** (Last In First Out), il quale prevede che **i primi dati ad uscire dalla pila sono gli ultimi inseriti**.

Lo stack è gestito tramite un puntatore, tipicamente **un registro** che contiene l'indirizzo di memoria della prima locazione libera e che prende il nome di **stack pointer (SP)**. Le **operazioni sullo stack** possono essere di due tipologie (**non è possibile l'accesso a posizioni arbitrarie**):

- **Processo di memorizzazione (Push)**, che consiste nel porre il dato in memoria all'indirizzo specificato in SP e incrementare tale registro in modo che continui a puntare all'indirizzo della prima locazione libera;
- **Processo di prelievo (Pop)**, che consiste nell'effettuare in maniera speculare le operazioni del Push, cioè il decremento del registro SP e la lettura del dato (in modo da liberare la locazione che era precedentemente puntata da SP).

Questa proposta è una sola delle possibili modalità di funzionamento dello stack, **è infatti possibile implementare il registro SP in modo che:**

- **Contenga l'indirizzo dell'ultima locazione scritta** (invece che della prima libera);
- **Sia decrementato con un Push (stack a decrescere)** invece che incrementato.

Queste impostazioni, relative alla cima e alla direzione dello stack, sono **convenzionalmente impostate**, secondo uno dei due schemi, **in maniera indipendente**.

Di base **lo stack non è un meccanismo fisico** ma un modo di gestire la memoria e **le varie convenzioni possono essere applicate senza dover richiedere cambiamenti nell'architettura del processore**; **i supporti architettonici sono opzionali** e spesso consistono in **registri stack pointer dedicati, istruzioni per il push e il pop o il salvataggio automatico dell'indirizzo di ritorno** in occasione della chiamata a funzione.

Il **Motorola 68K** adotta un'organizzazione dello **stack a decrescere con SP che punta all'ultima locazione occupata**: il **registro A7** è usato esplicitamente o implicitamente da alcune istruzioni come **Stack Pointer** (infatti in assembly è anche indicato come SP), mentre **A7' è uno Stack Pointer fisicamente diverso dal precedente**, usato in modalità supervisore (ad esempio dal codice del sistema operativo); **non sono presenti istruzioni dedicate al push e al pop** ma esse sono ottenute, rispettivamente, come **pre-decremento e post-decremento**:

PUSH → MOVE.L D1, -(SP)

POP → MOVE.L (SP)+, D3

In alcuni processori, tra cui il Motorola 68K, **l'istruzione di salto a subroutine salva automaticamente l'indirizzo di ritorno sulla cima dello Stack**:

JSR SUBR = push(SP, PC) \wedge subr → PC

Mentre l'istruzione di ritorno da subroutine consente il ripristino dell'indirizzo di ritorno dalla cima dello stack:

RTS = pop (SP, PC)

Lavorare con i sottoprogrammi non implica solo il **problema del linkage** ma anche quello del **passaggio dei parametri**; in questo caso **il problema non è salvare un singolo valore specifico** (quello del PC nel caso precedente) ma, invece, **rendere accessibili al sottoprogramma i valori dei parametri** su cui esso lavorerà una volta chiamato (in genere possono essere di qualsiasi numero e dimensione) e, inoltre, c'è bisogno che **il sottoprogramma renda accessibili** al chiamante **i risultati**.

Il **passaggio dei parametri** può avvenire **con i registri**, risulta essere la tecnica **più veloce** perché **non richiede accessi in memoria**: il programma chiamante e il sottoprogramma si **accordano sull'uso dei registri** in modo che **i parametri di ingresso sono sempre in specifici registri** dove quest'ultimo si aspetterà di trovarli e che **i risultati vengano scritti in altri** dove il chiamante si aspetterà a sua volta di trovarli. Il difetto di questa tecnica sta nel **numero limitato di registri**, che impone al sottoprogramma un **numero limitato di parametri**.

Un'alternativa che elide il limite appena riscontrato coinvolge un'**area fissa in memoria** (quindi si perde la **rapidità** che i registri forniscono) **che il chiamante e il sottoprogramma condividono** e usano per parametri, variabili locali e risultati. Usando questa tecnica, **lo scambio di parametri consiste unicamente nella scrittura dei dati e nel passaggio del solo indirizzo di partenza dell'area prestabilita**; ovviamente **è necessaria una convenzione (calling convention)** da adottare sull'inserimento dei dati in modo che si possa accedere ai parametri e ai risultati semplicemente usando uno spiazzamento rispetto all'indirizzo base e, infine, **è necessario che l'area di memoria sia sempre disponibile ad entrambi i programmi**, perché la chiamata può avvenire in qualsiasi momento del flusso.

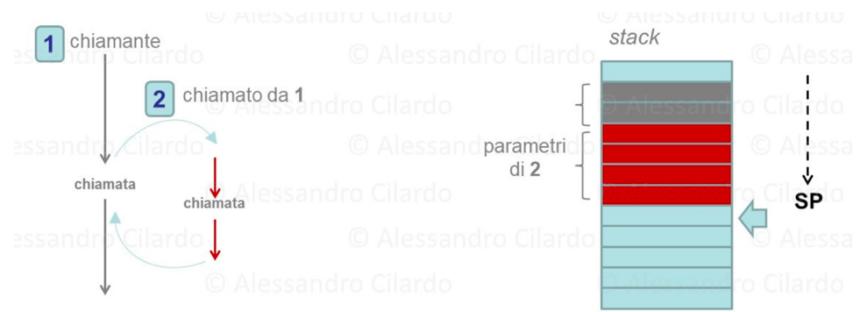
Come per il linkage, si può pensare all'area fissa con due possibili soluzioni:

- **Il sottoprogramma definisce** un'area di memoria adibita allo scambio di parametri;
- **Il programma chiamante alloca un'area di memoria** adibita allo scambio di parametri e **ne passa al sottoprogramma l'indirizzo iniziale**.

L'indirizzo di partenza può essere **comunicato tramite un unico registro indirizzo**, in modo da poter **usare di volta in volta aree fisiche diverse**.

Una possibile variante del passaggio di parametri con un'area di memoria fissa può essere **l'utilizzo dello stack**, che fornisce un **meccanismo ideale** per la gestione di aree di memoria comuni grazie al protocollo **LIFO**: **sono consentite chiamate annidate e chiamate ricorsive** (poco efficienti perché occupano molta memoria). Infatti **in una chiamata ricorsiva** (analogo per una chiamata annidata) **il primo sottoprogramma a terminare**, quindi la prima porzione dello stack ad essere deallocata con un **pop**, è **l'ultimo ad esser stato chiamato**:





In questo modo è possibile anche aggiungere una nuova area, quindi chiamare una nuova subroutine, che a sua volta sarà la prima a terminare e la relativa area dello stack la prima ad essere deallocated:

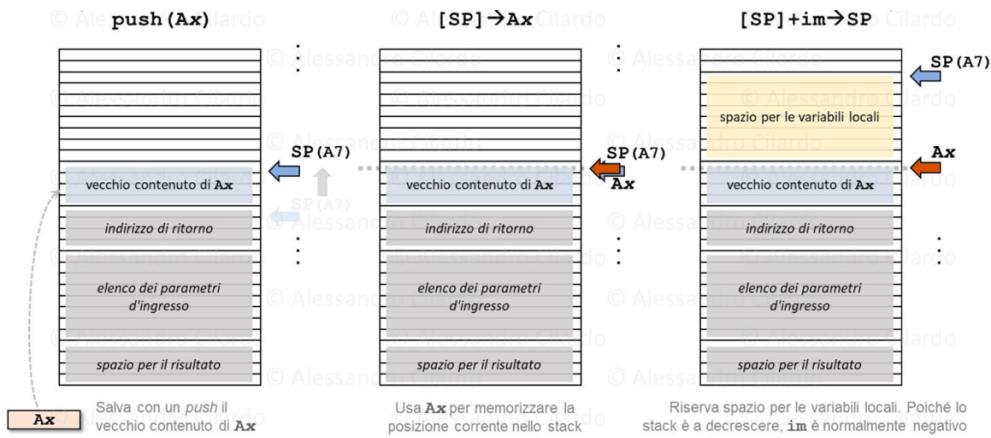


Come anticipato, nello stack oltre ai parametri e all'indirizzo di ritorno **è possibile inserire anche le variabili locali** del sottoprogramma, cioè variabili che la subroutine usa per eseguire delle operazioni ma che non sono visibili all'esterno del programma. In genere si fa riferimento alla **struttura composta da indirizzo di ritorno, parametri, locazioni per eventuali variabili di ritorno e locazioni per le variabili locali** con il nome di **Stack Frame o Record di Attivazione**.

Per riservare dello spazio nella stack per le variabili locali si può semplicemente **decrementare lo Stack Pointer**, con un effetto simile ad una **DS** ma con un'**allocazione dinamica**, cioè effettuata al momento della chiamata e con relativa **deallocazione al ritorno**. Al momento della chiamata la subroutine accede allo **stack frame** sfruttando il valore dello stack pointer, quindi le variabili locali, i parametri e gli spazi per i valori di ritorno saranno acceduti con un **offset rispetto al valore che punta lo stack pointer**. Tuttavia, durante lo svolgimento del programma, lo **stack frame cambia spesso** (possono essere fatti ulteriori push) e può risultare confusionario e tedioso dover memorizzare volta per volta l'offset delle singole locazioni a cui accedere; pertanto, è **opportuno che**, al momento della chiamata, **si salvi l'indirizzo di partenza dello stack frame** (chiamato Frame Pointer) in un registro, in modo esso sia fisso.

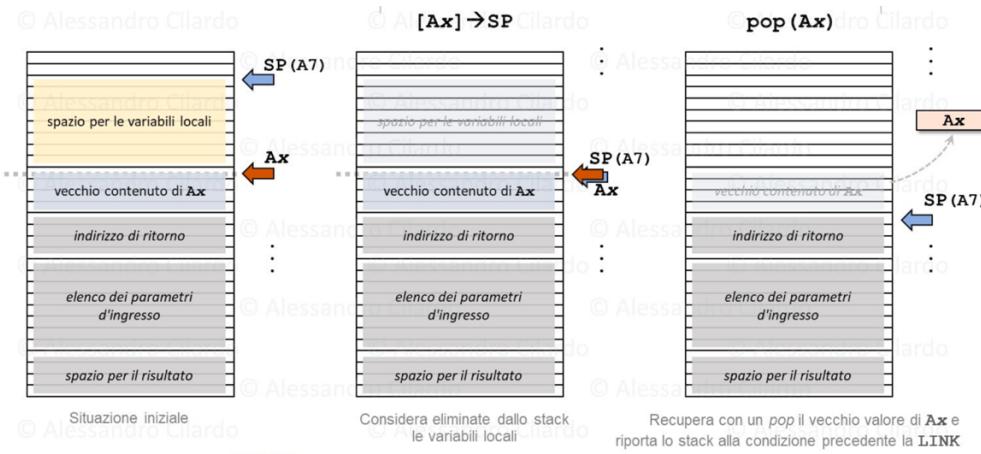
Per facilitare queste operazioni, il **Motorola 68K** mette a disposizione un'istruzione specifica, **LINK**, che si occupa di **effettuare il push** del vecchio contenuto del registro nello stack (in modo da non perdere eventuali dati già contenuti), **muovere il contenuto dello Stack Pointer** nel registro e **incrementarlo**; sommando un valore negativo al contenuto dello Stack Pointer, l'istruzione riserva un'area di memoria sulla cima dello stack che verrà usata per l'allocazione delle variabili locali:

LINK Ax, #im = push (Ax); [SP] → Ax; [SP] + im → SP



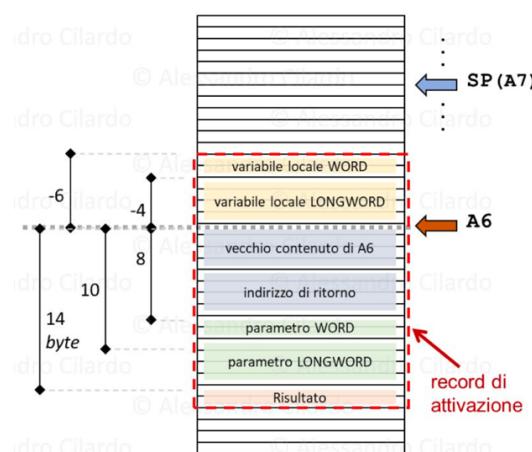
Dualmente, l'istruzione opposta alla LINK è UNLINK, che **cambia prima il valore dello Stack Pointer** (facendolo tornare a dove era prima della link) e poi **effettua un pop** per ripristinare l'eventuale valore del registro:

ULNK Ax = [Ax] → SP; pop (Ax)



Con questa tecnica, lo Stack Frame include:

- **Parametri** della chiamata;
- **Indirizzo di ritorno;**
- **Vecchio contenuto** del Frame Pointer;
- Area per le **variabili locali**.



Questo ordine **non è casuale**, infatti grazie all'architettura del Motorola 68K saranno **posizionati ad indirizzi più alti i primi elementi allocati**, cioè i **parametri** con la JSR, successivamente l'**indirizzo di ritorno** e solo dopo le **variabili locali**, che sono **allocate dopo che si è entrati nella subroutine**.

Anche in questo caso si accede alle varie locazioni dello **stack con gli spiazzamenti** ma risulta più organizzato e semplice, infatti con **spiazzamenti positivi** rispetto al Frame Pointer si raggiungono i **parametri** (almeno 8 byte, visto che il vecchio contenuto del registro e l'indirizzo di ritorno precedono i dati) e con **offset negativi** le **variabili locali**. Poiché si usa come **riferimento** un **indirizzo fisso**, l'offset di una locazione sarà sempre la stessa, mentre **lo Stack Pointer è libero di variare**.

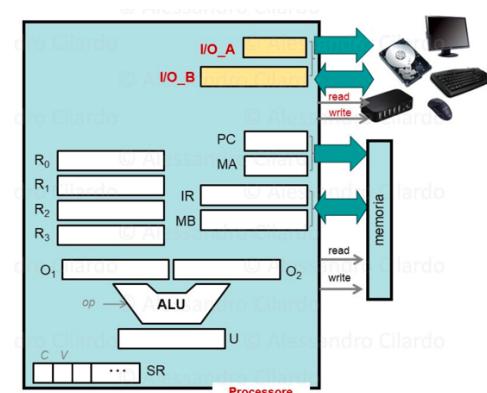
PERIFERICHE DI I/O

Con **periferiche di I/O** si intendono tutti i **dispositivi esterni all'interazione diretta processore-memoria**. Non c'è un **comportamento che accomuna tutte le periferiche** perché ognuna fa **qualcosa di diverso in modi diversi**; tuttavia, l'**interfacciarsi con il processore** di una periferica è classificabile in due tecniche:

- “**I/O mapped” I/O**

Il processore **dispone di un'interfaccia indirizzo/dato proprietaria** e i **dispositivi occupano uno spazio di indirizzamento proprio** i cui indirizzi non hanno nulla a che fare con quelli in memoria, mentre le operazioni sono codificate con istruzioni proprie.

Questo protocollo **prevede collegamenti e circuiti specifici** per la comunicazione processore-periferica, in totale ci saranno **due raggruppamenti di bus**: quelli per la memoria e quelli per le periferiche. Ovviamente ci sarà una **maggior specificità** per le operazioni ma al costo di una **complessità di progettazione più elevata**.



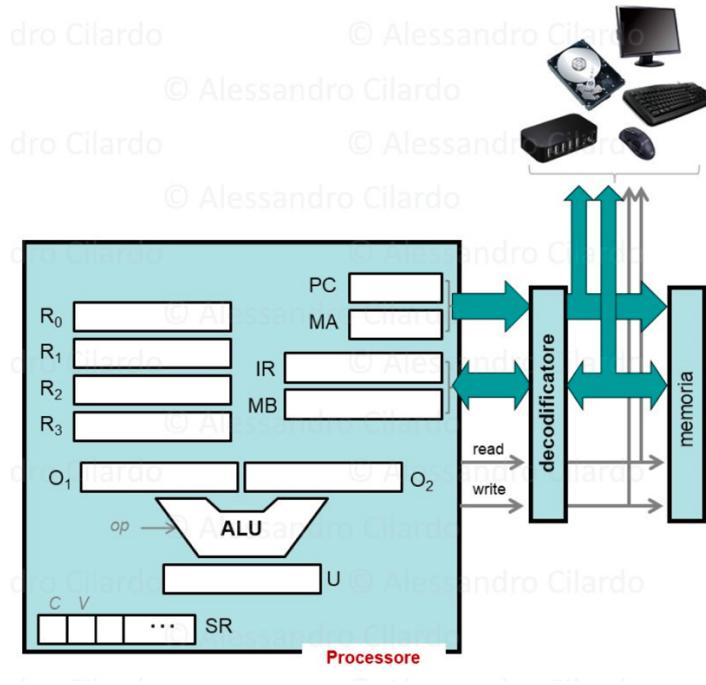
- “**Memory mapped” I/O (M68K)**

I **dispositivi sono identificati da specifici indirizzi in memoria** e le **operazioni eseguite su di essi sono codificate come accessi in memoria**.

Poiché alcuni specifici indirizzi non mappano locazioni di memoria, è **necessaria l'implementazione di un circuito, il decodificatore**, che “**instrada**” le richieste in base all'indirizzo, inviando il segnale alla memoria o alle periferiche. Ovviamente **alcune locazioni di memoria non potranno mai essere raggiunte** perché il decodificatore, una volta incontrato il relativo indirizzo, manda il

segna fuori dalla memoria (anche se la maggior parte degli indirizzi saranno indirizzi di memoria).

Questo protocollo consente una **progettazione meno complessa** al costo di una **maggior generalità** delle operazioni effettuabili sulle periferiche e su una minor quantità di locazioni di memoria disponibili. Infatti, il processore non viene modificato, le periferiche vengono considerate come **un'estensione della memoria** e si può cambiare il mapping della memoria a piacimento.



Come già anticipato, l'**interfaccia di una periferica è un'informazione poco astratta**, infatti è tipica per ogni periferica. Tuttavia, **ogni periferica condivide la struttura in registri**, sui quali si possono leggere o scrivere le informazioni; i registri possono essere di diversi tipi:

- **Controllo** (dal processore) o di **stato** (verso il processore)

I primi sono scritti dal processore e, mediante varie codifiche, specificano cosa la periferica deve fare, mentre i secondi sono scritti dalla periferica per il processore e informano il processore, attraverso opportune codifiche, l'esito di determinate operazioni.

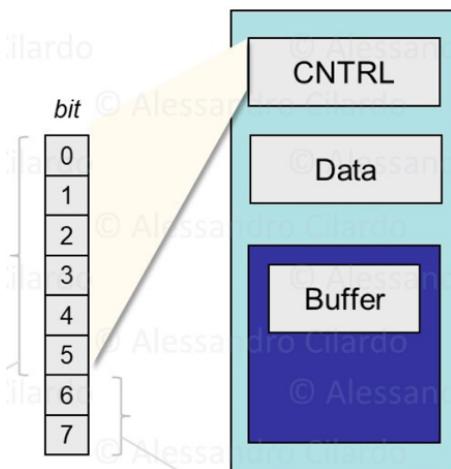
- **Indirizzi e dato**

Svolgono lo stesso ruolo dei relativi registri di memoria ma fanno riferimento ad un insieme molto più ristretto di locazioni fisiche.



Un esempio di periferica supportata dal Motorola 68K è il **terminale**, un dispositivo che fornisce un **interfaccia di input** (la tastiera) e **di output** (lo schermo) **combinati**. I registri che caratterizzano un terminale sono due (più un buffer) di un byte ciascuno:

- **CNTRL**, un indirizzo combinato di Controllo/Stato;
- **DATA**, un registro dato sul quale contenere i dati da mostrare su schermo o i dati da ricevere da tastiera;
- **Buffer**, un registro interno per contenere i dati inseriti da tastiera prima che essi vengano trasferiti.



Il registro **CNTRL** è composto da:

0. Abilita interruzione su BUFFER NULL;
1. Abilita interruzione su ENTER;
2. Cancella video;
3. Pulisce buffer tastiera;
4. Abilita tastiera;
5. Abilita eco (scrive su schermo quello che si sta scrivendo su tastiera);
6. Stato di BUFFER NULL;
7. Stato di ENTER inviato.

Dove i bit da **0 a 5** sono **di controllo** e **6 e 7 di stato**. Per quanto riguarda il registro **DATA**, poiché esso è **di un byte**, permette la scrittura di un solo carattere alla volta, quindi la periferica deve **automaticamente incrementare la posizione di scrittura quando si inseriscono più caratteri** consecutivamente.

Quando il registro **CNTRL** segnala il fatto che è stato premuto il tasto **ENTER** si alza il bit più significativo (bit di stato), ma nel frattempo che il tasto non è premuto si può implementare un **ciclo che legge il registro CNTRL dal quale si esce solo se tale bit è 1** (cioè se il tasto è stato premuto). Questo tipo di operazioni vengono chiamate **interazioni CPU-periferica**.

INTERRUZIONI

Normalmente le **istruzioni sono seguite una dopo l'altra** in un ordine ben preciso che può essere rotto nel momento in cui si verifichino dei **salti che spostano l'esecuzione su un'altra porzione di memoria**, tuttavia in molti casi è **necessario che il processore esegua delle routine solo quando si**

verificano determinate condizioni esterne, dettate spesso dalle periferiche: in questo caso l'evento è indipendente dal flusso del programma in esecuzione e si presenta in momenti che non possono essere predetti dal processore; questi eventi sono quasi sempre asincroni e si presentano con una frequenza molto più bassa rispetto a quella con la quale il processore carica ed esegue le istruzioni. Esistono due modi per far interagire il processore con questi eventi:

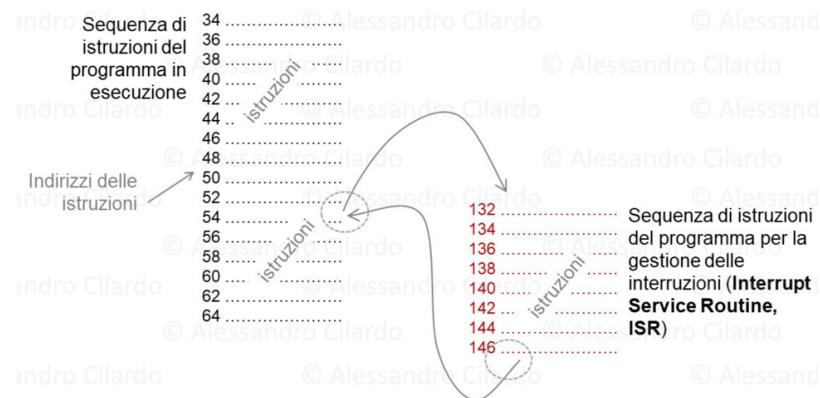
- **Polling**

Con questa tecnica la CPU accede ripetutamente ai registri di stato della periferica finché essa non è pronta (come l'ENTER del caso precedente) in un continuo stato di attesa detto busy-waiting. Il limite del Polling sta nella contemporaneità delle varie operazioni, infatti mentre la CPU è in attesa essa non può svolgere altre funzioni, visto che si basa su un modello gerarchico dove le istruzioni sono eseguite una alla volta; di contro, il vantaggio di questo meccanismo sta nel tempo di latenza assente tra un'istruzione e un'altra.

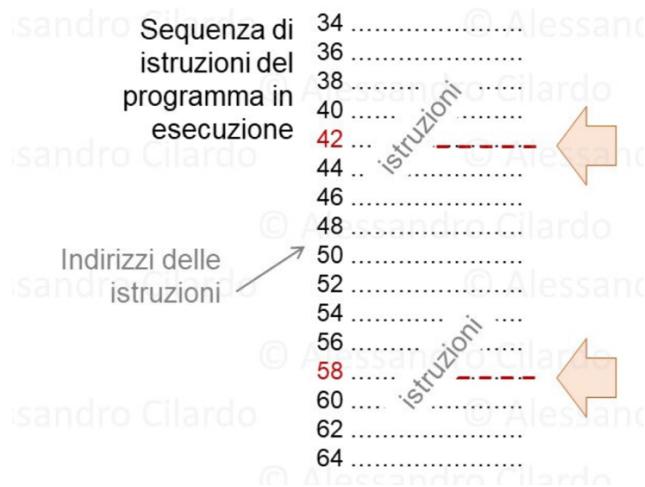
- **Interruzioni**

Il meccanismo delle interruzioni (exceptions) fornisce un'alternativa al polling con cui il processore può eseguire altre operazioni mentre la periferica lavora. Quando essa avrà finito, verrà richiamata l'attenzione del processore (tramite un'opportuna infrastruttura circuitale), che passa ad eseguire la routine che gestisce l'interruzione (Interrupt Service Routine, ISR); non si tratta però di una chiamata a sottoprogramma "consapevole" del processore, il quale ne deve rimanere ignaro.

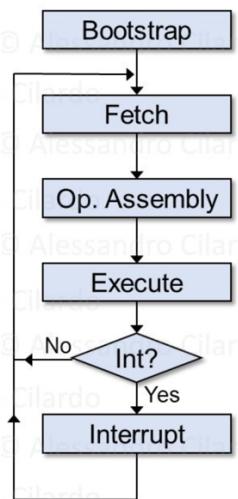
Le interruzioni sono una deviazione dal ciclo di Von Neumann, con cui si prevede che un evento esterno possa modificare il flusso del programma (ad esempio la pressione del tasto ENTER); ciò significa che l'interruzione è asincrona rispetto al programma e la sua accettazione implica la sospensione del programma in esecuzione.



Il meccanismo con cui si invoca la ISR è lo stesso di una chiamata a sottoprogramma con la sola differenza che il salto può avvenire in qualsiasi punto del programma: non c'è un'istruzione JMP che comunica al processore il punto da cui saltare e quello a cui arrivare e non c'è linkage o JSR; risulta che il controllo dell'interruzione deve essere eseguito indipendentemente dall'istruzione che il processore sta eseguendo, cioè ad ogni ciclo. Poiché il processore deve essere ignaro dell'interruzione (dal momento in cui non può prevedere il punto in cui avviene il salto), è necessario garantire la neutralità della ISR, cioè non devono essere persi i dati nello stack o nei registri del processore che vengono usati dal programma interrotto; a tal fine si può salvare, e in seguito ripristinare, lo stato del processore (quindi stack e memoria) prima dell'interruzione.

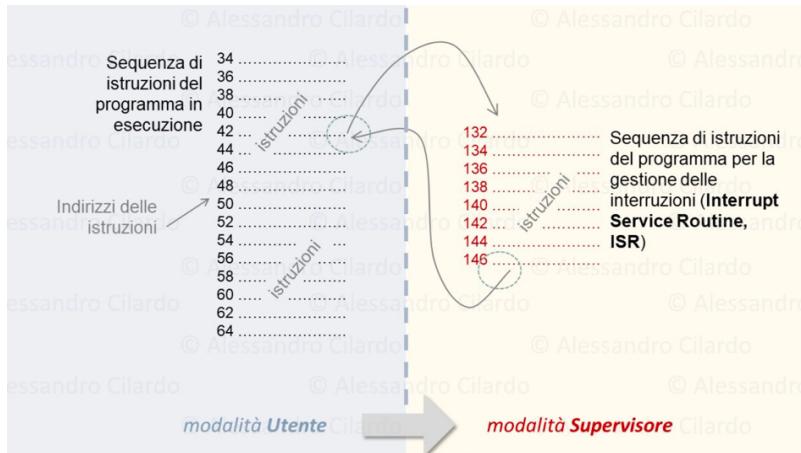


Implementare le interruzioni **sull'attuale ciclo di Von Neumann** comporta alcuni **problemi**: in primis **un'applicazione potrebbe impadronirsi della CPU senza mai terminare e senza la possibilità di essere rimossa**, ma poi **il Sistema Operativo in generale avrebbe un controllo limitato sul sistema**. Dunque, è necessario modificare il ciclo per permettere al SO di prendere il controllo **del processore al verificarsi di eventi “eccezionali” (exceptions)**, di solito **asincroni** con l'esecuzione del programma in corso. Per implementare questa funzionalità **si costruisce un circuito di interruzione che collega le periferiche alla CPU: ogni volta che si solleva un'interruzione** (cioè quando almeno una periferica alza il bit nel bus dedicato, OR) **si fotografa (circuitalmente) lo stato del processore e si salta all'interruzione**, alla fine della quale una Return from interruption ripristina lo stato del programma interrotto; **quando si entra nell'interruzione il fetch successivo sarà su una subroutine**.



Poiché la maggior parte degli eventi che sollevano interruzioni hanno a che fare con **funzionalità hardware di basso livello**, è necessario che i **relativi programmi accedano a tutte le caratteristiche del calcolatore**, di cui i programmi utente non devono essere a conoscenza. Pertanto, molti processori prevedono due modalità:

- **Supervisore**, con cui si ha pieno accesso alle funzionalità del sistema;
- **Utente**, accesso limitato alle istruzioni di uso generale.



Quando la CPU accetta un'interruzione può cambiare la modalità da Utente a Supervisore. Spesso modalità di esecuzione differenti accedono a stack diversi, come il caso del Motorola 68K: **sono presenti due Stack Pointer, A7 e A7', che vengono usati rispettivamente in modalità Utente e Supervisore;** in qualsiasi caso, **la modalità attiva** in un certo momento è parte dello **stato del processore**, nel Motorola 68K è specificata nel **registro di stato** dal **bit S**.

Le **interruzioni in senso generale**, o exceptions, si possono distinguere in:

- **Reset**

Riportano la macchina in uno stato iniziale noto e sono generate da **condizioni d'errore non recuperabili**.

- **Traps**

Forniscono un **meccanismo controllato di passaggio allo stato supervisore** ma sono eventi sincroni rispetto all'elaborazione; infatti, **generano un'interruzione al termine di ogni istruzione eseguita** o in corrispondenza di particolari situazioni determinate dal programma. Sono spesso usate per **eseguire passo-passo un programma o per il debug**.

Nel Motorola 68K sono **abilitate dal bit T** (trace, tracciamento) **del registro di Stato** ed innestate dall'istruzione TRAP.

- **Interruzioni in senso stretto**

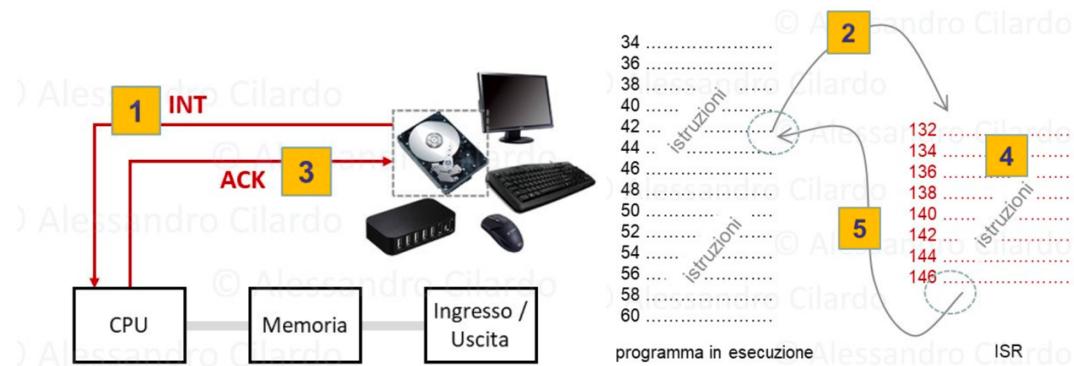
Permettono di **gestire richieste di “attenzione”** da parte di periferiche e sono **usati tipicamente dall'I/O**, ovviamente in modo asincrono. L'insieme di routine, comprese le ISR, che conoscono il funzionamento di una particolare periferica e ne gestiscono le interruzioni è detto **driver**.

Poiché **ogni periferica ha il proprio funzionamento**, ognuna necessita di un **proprio driver** e perciò nei computer commerciali l'installazione di una periferica richiede l'installazione dei corrispondenti driver. Ovviamente **ogni processore ha le proprie terminologie e i propri meccanismi** di riferimento per le interruzioni; praticamente, **tra processore e periferiche** si instaurano **due bus** che codificano per due segnali diversi:

- **Segnale INT**, inviato dalle periferiche e rappresenta il segnale di interruzione con il quale le periferiche richiedono l'attenzione del processore;
- **Segnale ACK**, inviato dal processore alle periferiche per comunicare l'accettazione dell'interruzione.

Solo quando viene accolta l'interruzione si può lanciare l'ISR. La sequenza di attivazione dell'interruzione risulta essere:

1. La periferica **alza il segnale INT**;
2. Il processore **interrompe il programma** in esecuzione;
3. La periferica viene **informata** dell'accettazione dell'interruzione con il segnale ACK;
4. Viene eseguita la **procedura ISR**;
5. Si **ripristina** il programma originale.



Entrando nei dettagli, durante questo processo avviene anche il **salvataggio del contesto hardware**, l'**identificazione del device che ha sollevato l'interruzione**, il **salvataggio del contesto software** e il loro ripristino, dopodiché riprende l'esecuzione normale del programma.

Come anticipato in precedenza, il **programma interrotto non deve essere consapevole dell'interruzione** e, pertanto, è necessario che vengano salvati e successivamente ripristinati tutti i **registri o le locazioni di memoria** che l'ISR può aver sporcato durante la sua elaborazione. Le informazioni essenziali da preservare comprendono il **valore del PC** prima dell'ISR, lo **SR** e il **contenuto di qualsiasi registro che sia usato da entrambi i programmi**. I primi due vanno salvati a prescindere al momento dell'accettazione, mentre per i registri e lo stack pointer è la stessa ISR che si occupa di fare dei **push** sullo stack per conservare tali dati e un **pop** per ripristinarli, ovviamente è un'operazione che richiede **accessi in memoria**, pertanto è buona pratica **limitarla allo stretto necessario** perché le interruzioni possono essere molto frequenti.

Tutte queste operazioni, dal controllo dell'interruzione al salvataggio dello stato, **non sono immediate ed istantanee**, incrementano il ritardo tra la ricezione della richiesta di interruzione e la sua esecuzione; questo tempo viene detto **latenza di interrupt** e, come anticipato in precedenza, è tipico del meccanismo delle interruzioni.



I calcolatori **non hanno collegati a sé una sola periferica** e può accadere che **allo stesso momento due o più periferiche sollevino un segnale di interruzione**, pertanto è **necessario introdurre un metodo di priorità delle interruzioni** che aiuta la gestione di eventuali segnali contemporanei; oppure, può accadere che **un'interruzione a priorità maggiore interrompa una a priorità**

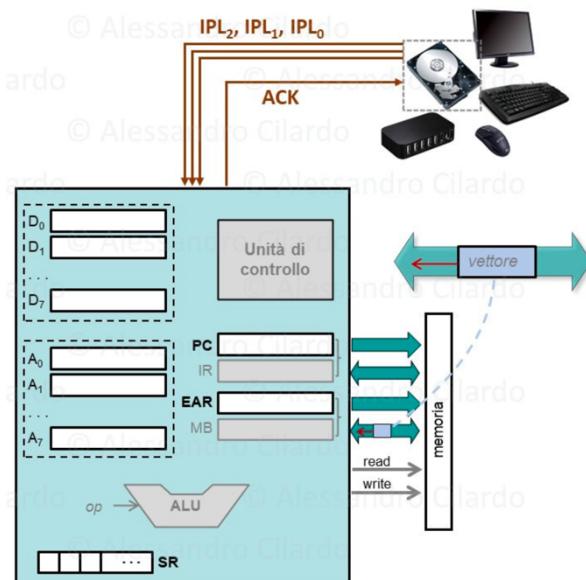
minore, quindi deve essere possibile gestire interruzioni innestate salvando e ripristinando lo stato di ciascuna ISR a sua volta interrotta (come in subroutine innestate).

Spesso la presenza di più periferiche risulta difficile da gestire, può accadere che le linee fisiche per la segnalazione dell'interruzione (INT) siano inferiori rispetto alle periferiche realmente connesse al calcolatore; ciò comporta che più periferiche possono condividere gli stessi collegamenti al processore, il quale deve saperle distinguere rispetto al segnale che ha sollevato l'interruzione. Da ciò nasce la necessità delle interruzioni vettorizzate, con le quali **ogni periferica manda un codice identificativo sul bus** (chiamato **vettore**) per farsi riconoscere dalla CPU e che può essere usato anche per individuare la specifica ISR che deve gestire quell'interruzione.

Nel **Motorola 68K** sono presenti tre linee fisiche che codificano sia la **presenza di interruzione** che il suo **livello di priorità**, da 1 a 7, dove il livello 000 è codificato come assenza di interruzione e 111 rappresenta il RESET, la priorità su tutto. Dunque, durante un'interruzione, nel **bit S si troverà 1** (visto che le ISR sono sempre in modalità supervisore) e nei **tre bit I il livello di priorità**: nel caso in cui venga sollevata un'interruzione con priorità minore di quella attualmente in esecuzione, essa non verrà accettata.

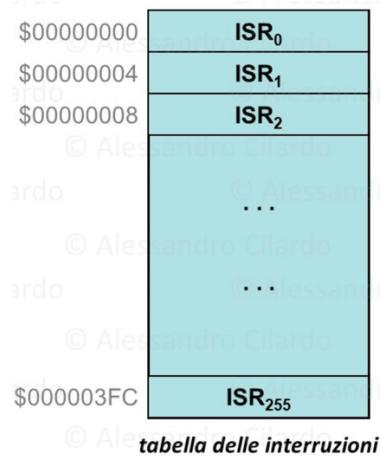


Nel caso in cui più periferiche condividano lo stesso livello di priorità, all'accettazione dell'interruzione la **CPU aspetta sul bus di sistema un'informazione di 8 bit**, cioè il vettore inviato dalla periferica per identificare sé stessa.



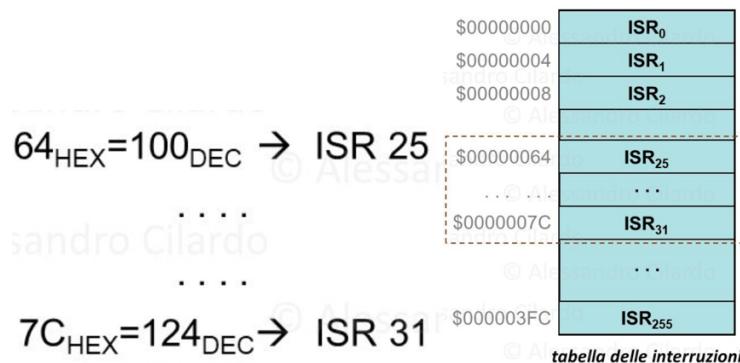
Identificare la periferica significa poter individuare la ISR corretta che può gestirla, infatti il processore gestisce una **tabella delle interruzioni** contenente gli indirizzi di tutte le ISR presenti in cui il vettore può essere usato come indice per individuare la routine corretta. Questa tabella è posta in memoria a partire dall'indirizzo **\$00000000** e contiene **256 locazioni di 4 byte**, ognuna contenente l'indirizzo dell'entry point di una ISR; pertanto, esistono **256 ISR differenti**:

- Le prime 24 hanno funzioni speciali;
- Da 25 a 31 sono interruzioni autovettorizzate;
- Da 32 a 47 sono trap;
- Le restanti interruzioni utilizzabili con periferiche generiche.



Le **interruzioni autovettorate** sono **ISR** che vengono invocate senza la necessità di specificare il vettore (solo con un segnale che indica al processore l'autovettorizzazione) e nascono dall'esigenza di **interruzioni particolarmente veloci**. Per determinare quale delle varie interruzioni autovettorate invocare il processore fa riferimento solo al livello di priorità, infatti esse sono **solo 7**:

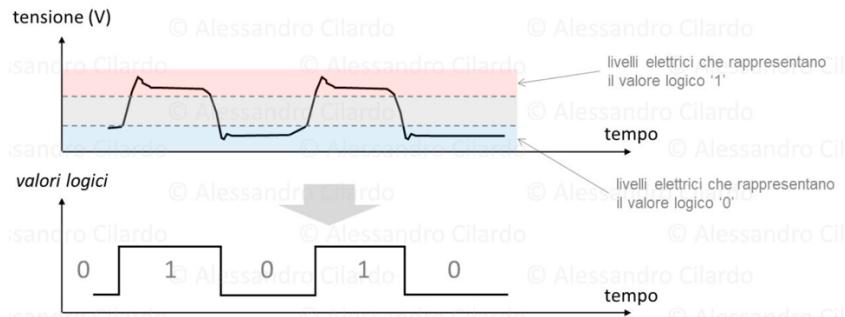
$$(24 + \text{priorità}) \cdot 4 = [60]_{16} + (\text{priorità}) \cdot 4 = \text{indirizzo entry point}$$



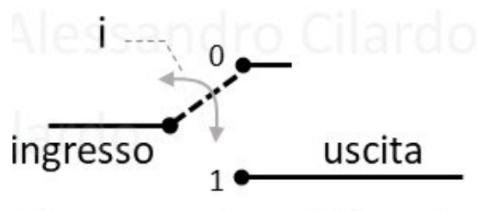
RETI LOGICHE

PORTE LOGICHE E L'ALGEBRA DI BOOLE

Nella pratica, in un calcolatore i **segnali binari** che codificano un'informazione sono rappresentati tramite un **segnale elettrico analogico**; lo “**standard logico**” definisce concretamente quali sono le **soglie e i margini di rumore** con cui questa operazione può essere effettuata:



I componenti atomici che permettono il passaggio di corrente sono i **transistor**, **interruttori di controllo** che possono essere **pilotati tramite un segnale i** con il quale si mette in **contatto** un **ingresso** e un'uscita, che chiude o apre il circuito:



Quando il **transistor è chiuso** c'è **passaggio di corrente** e si può codificare il **valore logico 1**, mentre se l'interruttore è **aperto la corrente non passa** e si codifica il **valore 0**. I transistor possono essere posizionati **in serie o in parallelo**, su appositi wafer semiconduttori a formare una **rete di comunicazione** con la quale si può **controllare l'output in funzione dell'input**; queste reti sono dette **porte logiche** e vengono **costruite sulla base di determinate funzioni matematiche definite dall'algebra booleana**.

L'algebra booleana, o **algebra di Boole** (sviluppata dall'omonimo matematico nel XIX secolo ma contestualizzata ai segnali elettrici da Claude Shannon), è un **sistema formale di astrazione matematica** con la quale si possono **risolvere problemi di natura logica**; in questa sede l'algebra booleana viene **usata per descrivere le funzionalità pratiche e il costo realizzativo dei circuiti** attraverso funzioni matematiche manipolabili.

Le **porte logiche fondamentali** sono costruite sulla base degli **operatori di base** dell'algebra booleana:

 NOT : Negazione $y = \bar{a}$	 AND : Congiunzione (o prodotto) $y = a \cdot b$ vale '1' se e solo se entrambi a e b sono '1'	 OR : Disgiunzione (o somma) $y = a + b$ vale '0' se e solo se entrambi a e b sono '0'	 NAND : negata della funzione AND $y = a \uparrow b$	 NOR : negata della funzione OR $y = a \downarrow b$																																																																		
<table border="1" style="display: inline-table; vertical-align: middle;"> <thead> <tr> <th>a</th> <th>y</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> </tr> </tbody> </table>	a	y	0	1	1	0	<table border="1" style="display: inline-table; vertical-align: middle;"> <thead> <tr> <th>a</th> <th>b</th> <th>y</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </tbody> </table>	a	b	y	0	0	0	0	1	0	1	0	0	1	1	1	<table border="1" style="display: inline-table; vertical-align: middle;"> <thead> <tr> <th>a</th> <th>b</th> <th>y</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </tbody> </table>	a	b	y	0	0	0	0	1	1	1	0	1	1	1	1	<table border="1" style="display: inline-table; vertical-align: middle;"> <thead> <tr> <th>a</th> <th>b</th> <th>y</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </tbody> </table>	a	b	y	0	0	1	0	1	0	1	0	0	1	1	0	<table border="1" style="display: inline-table; vertical-align: middle;"> <thead> <tr> <th>a</th> <th>b</th> <th>y</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </tbody> </table>	a	b	y	0	0	1	0	1	0	1	0	0	1	1	0
a	y																																																																					
0	1																																																																					
1	0																																																																					
a	b	y																																																																				
0	0	0																																																																				
0	1	0																																																																				
1	0	0																																																																				
1	1	1																																																																				
a	b	y																																																																				
0	0	0																																																																				
0	1	1																																																																				
1	0	1																																																																				
1	1	1																																																																				
a	b	y																																																																				
0	0	1																																																																				
0	1	0																																																																				
1	0	0																																																																				
1	1	0																																																																				
a	b	y																																																																				
0	0	1																																																																				
0	1	0																																																																				
1	0	0																																																																				
1	1	0																																																																				

La prima porta è l'unica che, nella sua formulazione essenziale, ha solo un ingresso, le altre sono tutte a più ingressi; **ogni ingresso corrisponde ad un segnale elettrico di input** che viene **processato** attraverso una serie di **transistor** nel corpo della porta e **restituito come un segnale in output**. Con **n ingressi**, la porta logica può produrre **2^n segnali di output**, i quali possono essere individuati con le **tabelle di verità**.

Gli operatori fondamentali dell'algebra booleana sono:

- **AND**

$$\cdot : K \times K \rightarrow K$$

- **OR**

$$+ : K \times K \rightarrow K$$

- **NOT**

$$\bar{} : K \rightarrow K$$

Poiché l'**algebra booleana non è stata sviluppata appositamente per la rappresentazione di segnali binari**, l'insieme **K** può avere infiniti valori. Tali operatori sono **definiti a partire dai seguenti assiomi**:

1. Proprietà commutativa

$$x \cdot y = y \cdot x$$

$$x + y = y + x$$

2. Proprietà associativa

$$(x \cdot y) \cdot z = x \cdot (y \cdot z)$$

$$(x + y) + z = x + (y + z)$$

3. Proprietà di idempotenza

$$x \cdot x = x$$

$$x + x = x$$

4. Proprietà di assorbimento

$$x \cdot (x + y) = x$$

$$x + (x \cdot y) = x$$

5. Proprietà distributiva

$$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$$

$$x + (y \cdot z) = (x + y) \cdot (x + z)$$

6. Proprietà di convoluzione

$$\bar{\bar{x}} = x$$

7. Proprietà del minimo e massimo

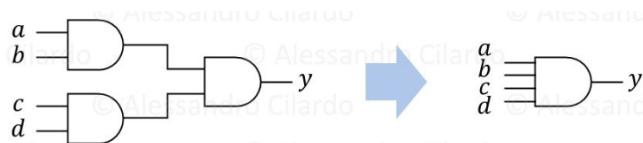
$$x \cdot 0 = 0$$

$$x + 1 = 1$$

La **proprietà di idempotenza** assume un ruolo di rilievo in ottica della **semplificazione di un circuito**, infatti con essa si può **omettere un ingresso** che è superfluo oppure **aggiungerne uno** al fine di produrre un particolare risultato.

Le **funzioni svolte dalle porte logiche** devono necessariamente **soddisfare questi sette postulati**, il che può essere verificato ispezionando direttamente le tabelle delle porte logiche.

Come anticipato, le regole dell'algebra di Boole consentono di **manipolare la rappresentazione dei circuiti digitali**, permettendone la **semplificazione** al fine di ridurne il costo; un esempio di semplificazione è:



Con l'algebra di Boole, individuata un'identità è possibile ricavarne sicuramente un'altra tramite il **principio di dualità**, il quale afferma che **uno stato o un operatore possono essere sostituiti con il proprio duale**:

$$\cdot \rightarrow +$$

$$+ \rightarrow \cdot$$

$$0 \rightarrow 1$$

$$1 \rightarrow 0$$

Il principio di dualità è stato già mostrato con l'elencazione dei sette postulati, infatti è stato affermato che, ad esempio, $x \cdot x = x$ e $x + x = x$ valgono entrambe; il motivo per cui sono entrambe vere risiede nel fatto che, **se una sola delle due è valida, per dualità è valida anche l'altra**.

Altre **proprietà** utili ai fini mostrati in questa sede possono essere **dimostrate** a partire dai postulati, quindi sono **teoremi**:

- **Teorema dell'opposto**

$$\bar{0} = 1 \wedge \bar{1} = 0$$

- **Teorema dell'elemento neutro della somma**

$$a + 0 = a$$

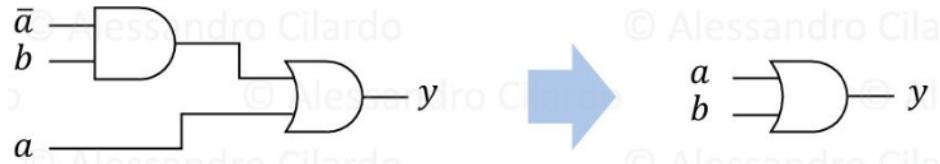
- **Teorema dell'elemento neutro del prodotto**

$$a \cdot 1 = a$$

- **Teorema di assorbimento del complemento**

$$a + \bar{a}b = a + b$$

Praticamente, un'applicazione di quest'ultimo teorema può essere la semplificazione del seguente circuito:



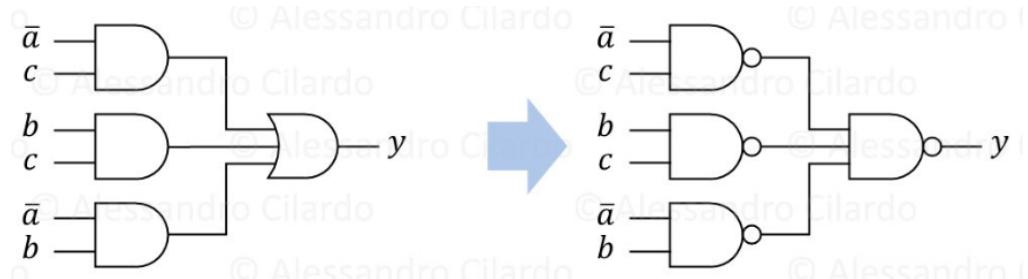
Il **teorema di De Morgan** è uno dei teoremi più importanti dell'algebra Booleana, ai fini pratici è utile perché permette di considerare una **AND come una NOR** e una **OR come una NAND**; questa corrispondenza è molto utile dal momento in cui le porte **AND e OR sono molto più difficili da realizzare** rispetto alle NOR e NAND, il che rende la **realizzazione del circuito più semplice**. Il teorema afferma che **il negato della somma/prodotto è uguale al prodotto/somma dei negati**:

$$\overline{a + b} = \bar{a} \cdot \bar{b}$$

$$\overline{a \cdot b} = \bar{a} + \bar{b}$$

Il teorema è stato mostrato **partendo da due ingressi**, tuttavia è applicabile con una **forma estesa a n ingressi**. Un esempio di circuito in cui è stato applicato il teorema di De Morgan è:

$$f = \bar{a}c + bc + \bar{a}b = \overline{\overline{\bar{a}c} + bc + \bar{a}b} = \overline{\bar{a}c} \cdot \overline{bc} \cdot \overline{\bar{a}b}$$



Si noti che il circuito ha **mantenuto il numero di ingressi e di collegamenti** ma ha mutato le porte logiche in uso, implementando **solo NAND** (un circuito semplice da realizzare).

Nonostante si stia lavorando con $+$ e \cdot , non bisogna confondere l'algebra booleana con l'algebra numerica; possono esserci alcuni punti di tangenza (come la proprietà associativa che esiste ed è valida in entrambe) ma ci sono alcuni principi che non valgono nell'algebra booleana, come il **principio di eliminazione**:

$$x + y = x + z \not\Rightarrow y = z$$

Infatti ponendo $x = 1 \wedge y = 0 \wedge z = 1$ l'uguaglianza permane ma non è vero che $y = z$; analogamente, per il principio di dualità:

$$x \cdot y = x \cdot z \not\Rightarrow y = z$$

I VARI MODELLI DI ALGEBRA DI BOOLE

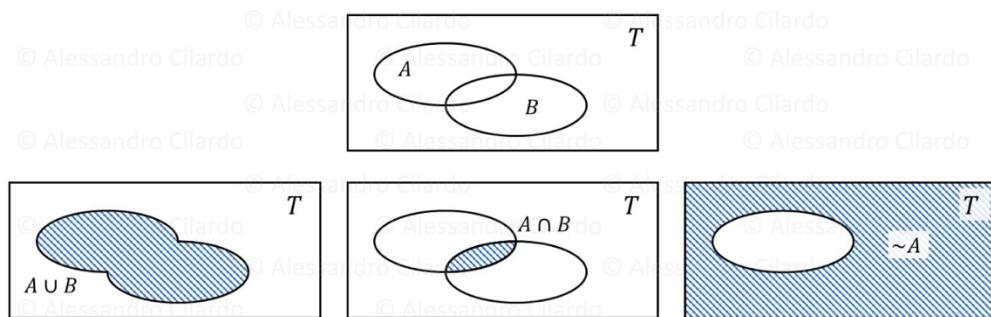
È stato precedentemente anticipato che la **definizione di Algebra di Boole non specifichi chi sia K , quanti e quali elementi contenga, e come siano definite le operazioni $+$, \cdot e \neg** ma specifica solo un insieme di proprietà che devono essere soddisfatte da tali operazioni; pertanto, sono possibili diversi modelli di algebra booleana, uno dei quali è trattato in questa sede e prende in considerazione valori binari manipolati da circuiti digitali.

Uno di questi modelli è detto **algebra delle proposizioni** e si avvicina molto all'algebra booleana binaria perché prevede che K contenga solo due elementi, **Vero** e **Falso**. Anche alcune operazioni coincidono:

- **Congiunzione (\wedge)**, corrispondente all'AND;
- **Disgiunzione (\vee)**, corrispondente all'OR;
- **Negazione (\neg)**, corrispondente al NOT;
- **Implicazione (\rightarrow)**;
- **Doppia implicazione (\leftrightarrow)**.

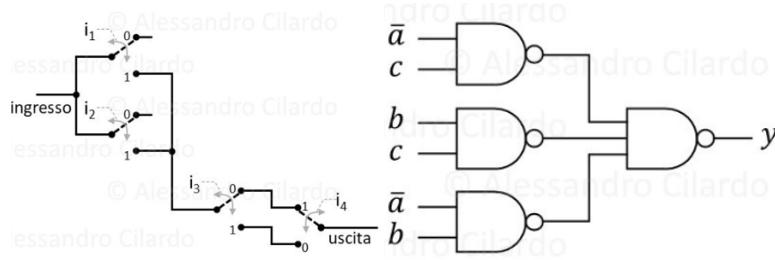
L'**algebra degli insiemi** prevede che le operazioni sono definite sul supporto K , costituito da tutti i possibili sottoinsiemi di un insieme universo T , incluso l'insieme vuoto \emptyset (**K è insieme delle parti di T**). Le operazioni definite sono:

- **Unione (\cup)**, corrispondente all'OR;
- **Intersezione (\cap)**, corrispondente all'AND;
- **Complemento (\sim)**, corrispondente al NOT;



Quella su cui ci si concentra è detta **algebra dei circuiti**. Essa **lavora sulle reti** (o circuiti) **bilaterali**, dove l'uscita è determinata dalla presenza o assenza di "contatto" tra due punti; un'alternativa è la **rete unilaterale**, dove l'uscita corrisponde a un valore di una grandezza elettrica misurata in opportuni punti del circuito.

In entrambi i casi, che sono essenzialmente la stessa **rappresentazione sotto due punti di vista differenti**, il **flusso dell'elaborazione procede fisicamente in una sola direzione**: dai segnali di ingresso a quelli di uscita.



Come già mostrato, esistono dei punti di tangenza tra alcune algebre: **il modello degli insiemi è quello più generale** ed è assunto spesso come strumento per **verificare o dimostrare una qualsiasi proprietà**; infatti **ogni algebra di Boole può essere rappresentata su un'algebra ad insiemi**.

Le proprietà dell'algebra dei circuiti di rappresentazione canonica delle reti attraverso funzioni logiche permettono di effettuare un processo di **sintesi e trasformazione sistematica sulle stesse funzioni finalizzato alla riduzione al minimo del costo realizzativo della rete**; questo processo viene detto **risoluzione del problema di ottimizzazione**.

FUNZIONI BOOLEANE STUDIO DELLE RETI LOGICHE

Viene definito **valore booleano** un elemento dell'insieme di sostegno K , mentre una **variabile booleana** è una variabile che può assumere solo valori booleani, cioè solo i valori di K . Con queste informazioni si può definire una **funzione booleana** come un'applicazione che prende in input variabili booleane e restituisce valori booleani:

$$\forall x_1, x_2 \dots x_n \in K, f(x_1, x_2 \dots x_n) \in K$$

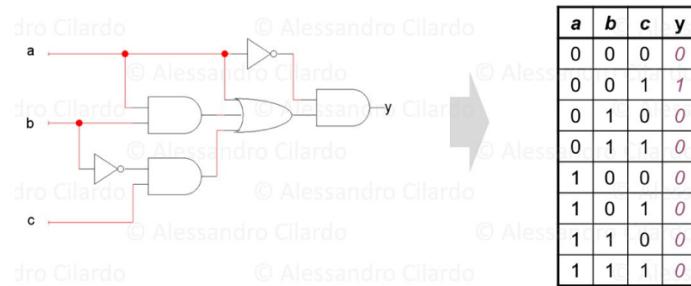
Le **variabili booleane** prese in argomento da una funzione **possono anche essere, a loro volta, funzioni booleane**; in tal caso la funzione più esterna è detta **funzione booleana composta**. Un insieme F di funzioni f su K è detto **funzionalmente completo** se qualsiasi funzione dell'algebra può essere ottenuta come **composizione di funzioni appartenenti ad F** .

Si può notare che se l'algebra è finita, qualsiasi funzione può essere rappresentata mediante una **tabella**, detta tabella di verità:

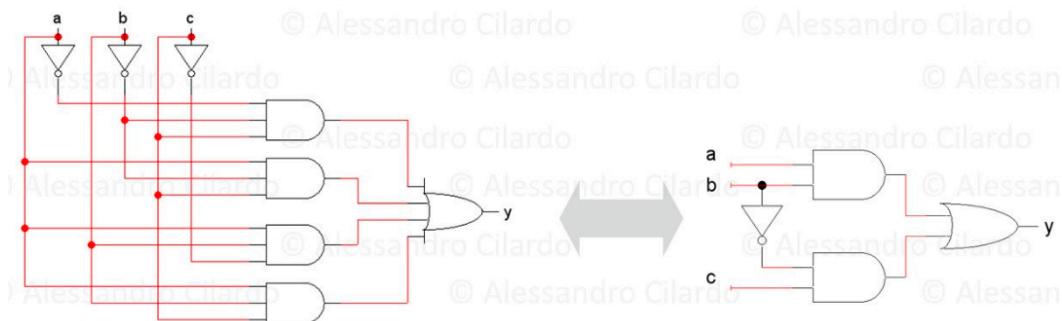
a	b	c	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Prendendo in considerazione una funzione con **n ingressi** e un'uscita, le colonne saranno **$n + 1$** e le righe 2^n , dove **ogni colonna rappresenta tutti i possibili valori di quell'ingresso e le righe una particolare combinazione**, ovvero una **particolare istanza della funzione**. Con **n ingressi** il numero di **funzioni distinte** può essere 2^{2^n} .

Come già anticipato, le funzioni booleane possono essere ottenute anche dalla composizione di diverse funzioni booleane più semplici; in particolare le **funzioni componenti** possono essere quelle che **descrivono il comportamento delle porte logiche realizzabili circuitalmente**:



Ovviamente questa associazione può essere fatta anche al contrario, **osservando quali sono i valori della funzione** e gli ingressi in modo da **astrarne una legge**. Quando si ottiene una rete da questo processo **non è sempre detto che sia la migliore possibile**; questa proprietà può essere verificata solo **osservando la funzione algebrica ed assicurandosi che essa sia semplificata al massimo**. Ad esempio, queste due reti assolvono alla stessa funzione, infatti osservando le rispettive tabelle di verità si può notare come a pari ingressi corrispondono pari uscite:



Ovviamente questa sintesi è **necessaria in ottica del problema di ottimizzazione**, infatti la prima presenta una quantità maggiore di porte e di collegamenti rispetto alla seconda, il che la rende più difficile e costosa da realizzare su un wafer.

La **sintesi delle reti logiche (combinatorie)** risulta essere molto **più semplice** quando le si vanno a considerare sotto forma di **funzione algebrica**, in questo modo si possono osservare e riconoscere meglio tutti gli schemi di semplificazione dell'algebra di Boole. In questa ottica si distinguono i seguenti elementi atomici di una funzione algebrica booleana:

- **Letterale** (o literal), che rappresentano un singolo ingresso

$$a$$

- **Clausola**, che rappresenta il termine elementare

$$a\bar{b}c$$

- **Fattore elementare**, che rappresenta una porta o una singola uscita

$$(a + \bar{c})$$

- **Somma di prodotti**

$$abd + b\bar{c}d + \bar{c}d\bar{a}$$

- **Prodotto di somme**

$$(a + b) \cdot (\bar{b} + d) \cdot (\bar{c} + \bar{d})$$

- **Mintermine P_i** di una funzione di n ingressi, rappresenta uno qualsiasi dei 2^n prodotti in cui figurano tutte le n variabili. Sono denotati con un pedice che, in base alla sua rappresentazione binaria, indica la posizione dei negati (negato $\rightarrow 0$)

$$P_5 = P_{101} = a\bar{b}c$$

- **Maxtermine S_i** di una funzione di n ingressi, rappresenta una qualsiasi delle 2^n somme in cui figurano tutte le n variabili. Sono denotati con un pedice che, in base alla sua rappresentazione binaria, indica la posizione dei negati rovesciata (negato $\rightarrow 1$)

$$S_5 = S_{101} = \bar{a} + b + \bar{c}$$

Bisogna fare attenzione a dove negare: **nei mintermini si nega in corrispondenza dello 0** mentre **nei maxtermini in corrispondenza degli 1**. Ciò è dovuto sempre al **principio di dualità**, infatti:

$$\bar{P}_i = S_i$$

Il **prodotto di due mintermini** diversi tra loro è **sempre 0**, la **somma di due maxtermini sempre 1**, mentre la **somma di tutti i mintermini è sempre 1** e il **prodotto di tutti i maxtermini sempre 0**.

Qualsiasi funzione logica può essere scritta come somma di mintermini; questo modo di descrivere una funzione è detta **forma canonica di tipo P**. Presa in esame la seguente tabella di verità e osservandone i termini alzati (1):

a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

La si può scrivere come somma dei relativi mintermini:

$$f = \bar{a}\bar{b}c + a\bar{b}c + ab\bar{c} + abc$$

È possibile **anche partire da un'espressione scritta come somma di prodotti** (ma in generale qualsiasi espressione) e da essa ottenere **una forma canonica P**; infatti, laddove manchi un letterale, basta moltiplicare per l'elemento neutro $1 = (\bar{x} + x)$. Ad esempio:

$$\begin{aligned} ad + b\bar{c}d &= a(b + \bar{b})(c + \bar{c})d + (a + \bar{a})b\bar{c}d = \\ abcd + ab\bar{c}d + a\bar{b}cd + a\bar{b}\bar{c}d + ab\bar{c}d + \bar{a}b\bar{c}d = \\ abcd + ab\bar{c}d + a\bar{b}cd + a\bar{b}\bar{c}d + \bar{a}b\bar{c}d \end{aligned}$$

Analogamente, si può descrivere **ogni funzione logica come prodotto di maxtermini**; questa forma è denominata **forma canonica di tipo S** e si ottiene osservando i termini abbassati (0) di una tabella di verità. Riprendendo l'esempio di prima:

$$f = (a + b + c) \cdot (a + \bar{b} + c) \cdot (a + \bar{b} + \bar{c}) \cdot (\bar{a} + b + c)$$

I maxtermini catturano le situazioni in cui la funzione è 0, mentre i mintermini quando la funzione è 1.

Infine si definisce **implicante di una funzione f** la funzione h che, se alta, rende alta f , ovvero quella funzione tale che:

$$\bar{h} + f = 1$$

Si può notare che in una funzione scritta come **somma di prodotti**, ciascun **prodotto** rappresenta un esempio di **implicante**; infatti, in una somma se è alto anche solo un termine è alta l'intera somma.

Un implicante viene detto **implicante primo** se non implica, a sua volta, nessun altro implicante della funzione; ritornando all'esempio precedente, **i termini di una somma sono sempre implicanti ma non necessariamente implicanti primi**:

$$y = \bar{a}b\bar{c} + \bar{a}bc + abc = \bar{a}b\bar{c} + bc(\bar{a} + a) = \bar{a}b\bar{c} + bc$$

In questa funzione abc è un implicante di y ma non è primo perché è anche implicante di bc , mentre quest'ultimo non implica nient'altro che y e quindi è un implicante primo:

$$abc \rightarrow bc \rightarrow y$$

In genere **una clausola Y implica un'altra clausola X se Y contiene tutti i letterali di X** . Un **espressione scritta** usando solo **implicanti primi** è detta **forma PI**.

La somma di due clausole di ordine n che contengono $n - 1$ letterali uguali ed uno negato è una clausola di ordine $n - 1$, formata dai letterali comuni, detto **consenso**:

$$abc + \bar{a}bc = (a + \bar{a})bc = bc$$

I **consensi** sono **fondamentali** nella **ricerca degli implicanti primi**.

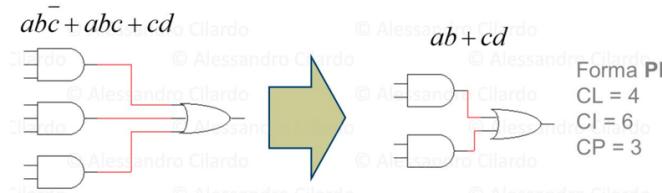
LE FUNZIONI DI COSTO E LE FORME MINIME

Lo studio delle **funzioni di costo** è una componente essenziale della costruzione di una rete, tali funzioni legano l'**espressione algebrica all'ipotetico circuito attraverso la valutazione del suo costo realizzativo**. Le funzioni di costo possono prendere in considerazione **diversi parametri**:

- **Costo di Letterali (CL)**, che valuta il numero di letterali che appaiono distintamente (anche se ripetuti) in un'espressione;
- **Costo di Porte (CP)**, che valuta il numero di funzioni (indipendentemente dal numero di ingressi) utilizzate nell'espressione;
- **Costo di Ingressi (CI)**, che valuta il numero di ingressi per ogni porta;

Queste funzioni **non sono legate tra loro**, infatti se una restituisce un valore basso non è detto che anche le altre seguiranno lo stesso comportamento; lo scopo di chi progetta la rete è quello di **abbassare il valore di ognuna di queste funzioni pur mantenendo la funzionalità logica**.

Sulle funzioni di costo è possibile definire due **teoremi** di particolare rilievo: il primo afferma che **una forma elementare che minimizzi il CI e il CL è una forma PI**, mentre il secondo afferma che **tra le forme minime a due livelli che minimizzano il CP c'è almeno una forma PI**. La crasi di questi due teoremi è un terzo teorema, il quale afferma che **sotto il vincolo di rete a due livelli la forma minima va ricercata tra le forme PI**. Ad esempio:



Un **implicante primo E_i di una funzione f** è detto **implicante primo essenziale** se è l'unico ad essere implicato da un mintermine di f , ovvero se è l'unico a “coprire” un determinato mintermine della funzione. Si considera, a tal proposito, **nucleo della funzione** la somma dei suoi **implicant primi essenziali**:

$$N = \sum_{i=1}^k E_i$$

Ogni **forma minima di f** è del tipo:

$$f = N + R$$

Dove **R** è l'**insieme degli implicant primi non essenziali**; tale formulazione, che ammette la nullità di N e R, risulta essere una **forma PI** (è comunque la somma di implicant primi, essenziali e non) e quindi non va in contraddizione al teorema appena enunciato.

Sulla base di quanto appena detto, per scrivere una **funzione minima a partire da un'espressione algebrica** la si svilupperà come **forma PI**, la quale includerà **implicant primi essenziali** (cioè il nucleo N) e **anche quelli non essenziali in modo che**, a costo minimo, **sommati tra loro e con il nucleo restituiscano la stessa funzione logica** dell'espressione iniziale. Lo schema da seguire è il seguente:

1. Si ottiene la funzione f da minimizzare;
2. Si riscrive la funzione in forma PI;
3. Si sceglie il sottoinsieme ottimale di PI la cui somma dà ancora f ;
4. Si minimizza la funzione;
5. Si costruisce il circuito.

Tra il passaggio 1 e il passaggio 2 è molto importante **individuare tutti gli implicanti primi**; esistono molti metodi per fare ciò ma i più efficaci sono:

- **Metodo algebrico, metodo di Quine**

Per usare il metodo di Quine bisogna, innanzitutto, **scrivere la funzione come forma P**, cioè come somma di mintermini; successivamente **si cercano tutti i possibili consensi tra coppie qualsiasi di clausole** (aventi lo stesso numero di letterali) per poi **eliminare quelle che sono state “assorbite” in clausole più piccole** (poiché implicanti non primi), infine si iterano i **raccoglimenti per clausole** di dimensione via via minore **finché non sarà più possibile individuare consensi**. A questo punto si è ottenuta la **funzione in forma PI**.

Ad esempio:

$$\begin{aligned}
 f(a,b,c) &= \bar{a}(b+c) + ac \quad \text{© Alessandro Cilardo} \\
 \bar{a}b + \bar{a}c + ac &= \quad \text{© Alessandro Cilardo} \\
 \bar{a}b(c+\bar{c}) + \bar{a}(b+\bar{b})c + a(b+\bar{b})c &= \quad \text{© Alessandro Cilardo} \\
 \bar{a}bc + \bar{a}\bar{b}c + \bar{a}\bar{b}\bar{c} + abc + \bar{a}bc &= \quad \text{© Alessandro Cilardo} \xrightarrow{\text{Forma P}} \\
 ab + \bar{a}c + bc + \bar{b}c + ac &= \quad \text{© Alessandro Cilardo} \\
 \bar{a}b + c + c = \bar{a}b + c &= \quad \text{© Alessandro Cilardo}
 \end{aligned}$$

- **Metodo grafico, metodo di Karnaugh**

Con il metodo di Karnaugh si **dispone la tabella di verità in una configurazione** leggermente diversa: presi ad esempio tre ingressi (a, b, c), il prodotto di due ingressi è posto come colonna (dove ogni colonna rappresenta il valore del prodotto nell'ordine 00, 01, 11, 10) e **il restante ingresso come riga** (0 e 1). In questa configurazione **da cella a cella cambia un solo letterale**, quindi è facile individuare i **consensi** perché sono sempre **celle adiacenti alzate**; questo principio vale anche per le **celle di bordo** (effetto Pac Man) ed è per questo motivo che le colonne si dispongono in quell'ordine, perché così anche tra fine ed inizio cambia un solo letterale:

a	b	c	y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

	bc	00	01	11	10
a	0				
1					

In entrambe le configurazioni si possono notare i consensi $\bar{a}b = \bar{a}bc + \bar{a}b\bar{c}$ e $bc = \bar{a}$. Comparando i due metodi finora visti:

$$\begin{aligned}
 f(a,b,c) &= \bar{a}(b+c) + ac \\
 \bar{a}b + \bar{a}c + ac &= \\
 \bar{a}b(c + \bar{c}) + \bar{a}(b + \bar{b})c + a(b + \bar{b})c &= \\
 (\bar{a}bc) + (\bar{a}\bar{b}c) + \bar{a}bc + abc + a\bar{b}c &= \\
 \bar{a}b + \bar{a}c + bc + \bar{b}c + ac &= \\
 \bar{a}b + c &=
 \end{aligned}$$

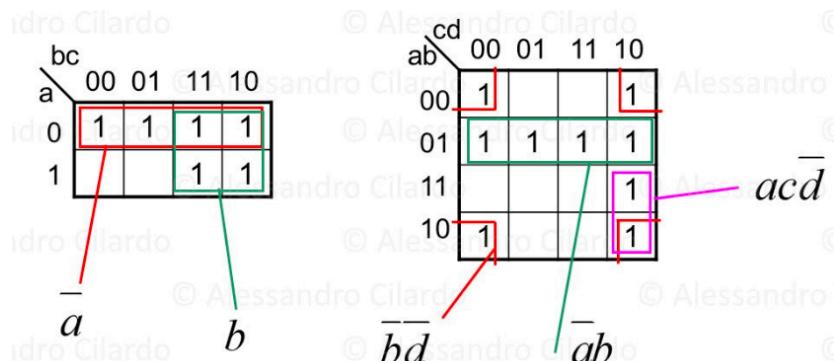
	bc	00	01	11	10
a					
0		1	1	1	1
1		1	1		

E così via...

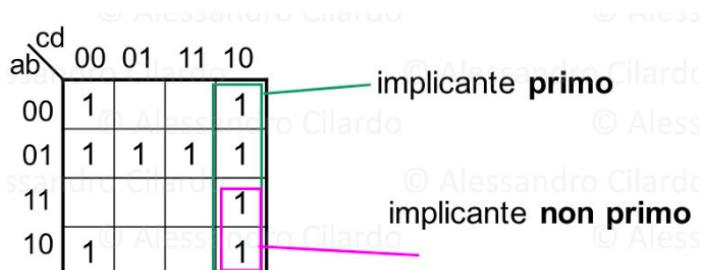
I **gruppi di celle sulle mappe di Karnaugh che corrispondono ad una clausola** (ad esempio $\bar{a}b$ oppure c) sono detti “**cubi**”:

	bc	00	01	11	10
a					
0		1	1	1	1
1		1	1		

La caratteristica essenziale di un cubo è quella di **contenere alcune variabili sempre con lo stesso segno**, mentre le rimanenti assumono tutte le possibili combinazioni di valori negati o no; pertanto, **un cubo conterrà sempre un numero di celle pari ad una potenza di 2**:



Con le mappe di Karnaugh si possono anche trovare gli **implicanti primi**, semplicemente individuando i **cubi più larghi** (che corrispondono a clausole più piccole) che coprano **celle con 1**:



- **Metodo tabellare, metodo di McCluskey**

Il metodo di McCluskey serve ad organizzare in **forma tabellare la ricerca dei consensi** e ad evitare confronti non strettamente necessari effettuati dal metodo algebrico; ciascuna clausola della funzione viene indicata da una **stringa di 0 e 1**, dove il primo indica la **forma negata** e il secondo quella **affermata**, e da un – per le variabili che non compaiono nel prodotto. Ad esempio, su quattro ingressi:

$$ab\bar{d} \rightarrow 11-0$$

L'idea per riconoscere i consensi è la stessa di Karnaugh: è facile identificare **coppie di clausole che differiscono per un solo simbolo** (incluso il trattino) e che individuano un **consenso**, il quale si può indicare come un'unica clausola con il trattino al posto del termine discordante.

$\bar{a}bc \rightarrow 011$	$\bar{a}bcd \rightarrow 0110$	$\bar{a}bc \rightarrow 011$	$\bar{a}bd \rightarrow 01-1$
$abc \rightarrow 111$	$\bar{a}b\bar{c}d \rightarrow 0100$	$ab\bar{c} \rightarrow 110$	$abd \rightarrow 11-1$
consenso	consenso	nessun consenso	consenso

Il primo consenso verrà quindi scritto – 11 cioè bc , infatti:

$$\bar{a}bc + abc = bc$$

Per applicare questo metodo si scrive innanzitutto la funzione in forma P, si denotano i mintermini con la convenzione appena mostrata e si suddividono le clausole in classi contenenti elementi con egual numero di 1 (riduce il numero di confronti e facilita l'identificazione di consensi). Ad esempio:

a	b	c	d	y	
0	0	0	0	0	
0	0	0	1	1	1)
0	0	1	0	0	
0	0	1	1	1	2)
0	1	0	0	0	
0	1	0	1	0	
0	1	1	0	0	
0	1	1	1	1	4)
1	0	0	0	0	
1	0	0	1	0	
1	0	1	0	0	
1	1	0	0	1	3)
1	1	0	1	1	5)
1	1	1	0	1	6)
1	1	1	1	1	7)

Implicanti
del 4° ordine

1) 0001

2) 0011

3) 1100

4) 0111

5) 1101

6) 1110

7) 1111

Per ogni i da 0 a $k - 1$ si generano consensi dell'ordine $k - i$ accoppiando le clausole della classe i con quelle della classe $i + 1$, marcando quelle che generano consenso (le clausole non marcate sono implicanti primi di ordine k) ed eliminando i doppiioni. I consensi ottenuti sono ordinati e riorganizzati da classe 1 a classe $k - 1$:

Impliciti del 4° ordine	Impliciti del 3° ordine	Impliciti del 2° ordine
1) 0001 ✓	1-2) 00-1	
2) 0011 ✓	2-4) 0-11	3-5-6-7) 11--
3) 1100 ✓	3-5) 110- ✓	3-6-5-7) 11--
4) 0111 ✓	3-6) 11-0 ✓	
5) 1101 ✓	4-7) -111	
6) 1110 ✓	5-7) 11-1 ✓	
7) 1111 ✓	6-7) 111- ✓	

In questo modo sono evidenti tutti gli implicanti primi della funzione:

$$\bar{a}\bar{b}d, \bar{a}cd, bcd, ab$$

Una **forma minimizzata** dovrà essere una **somma di tali PI**, ma non necessariamente li includerà tutti. Ad esempio:

$$f = \bar{a}\bar{b}d + \bar{a}cd + bcd + ab =$$

$$\bar{a}\bar{b}d + \bar{a}\bar{b}cd + \bar{a}bcd + bcd + ab =$$

$$\bar{a}\bar{b}d + bcd + ab$$

Quindi, **individuati gli implicanti primi**, occorre **scegliere** tra essi un **insieme di "copertura"** che consenta di **assolvere ancora al compito logico** della funzione **senza porte o circuiti superflui**. Il problema della copertura minima si può ridurre alla seguente formulazione: **data una matrice di N righe e M colonne**, dove gli elementi a_{ij} sono 0 o 1, **si dice che una riga i copre una colonna j se $a_{ij} = 1$** ; di conseguenza **per la copertura minima è sufficiente selezionare il numero minimo di righe che coprano tutte le colonne**.

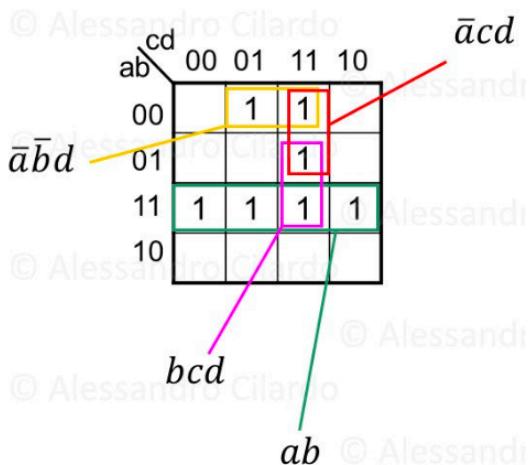
Sulle **righe** della matrice si posizionano gli **implicanti primi** e sulle **colonne i mintermini che implicano gli implicanti primi**; in questo modo dire che **"la riga X copre la colonna Y"** equivale a dire che **l'implicante primo X è implicato dal mintermine Y**. Ad esempio:

a	b	c	d	y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

A questa tabella di verità si può ricondurre la seguente matrice:

	P_1	P_3	P_7	P_{12}	P_{13}	P_{14}	P_{15}
$\bar{a}cd$ (0-11)		X	X				
$\bar{a}\bar{b}d$ (00-1)	X	X					
bcd (-111)			X				X
ab (11--)				X	X	X	X

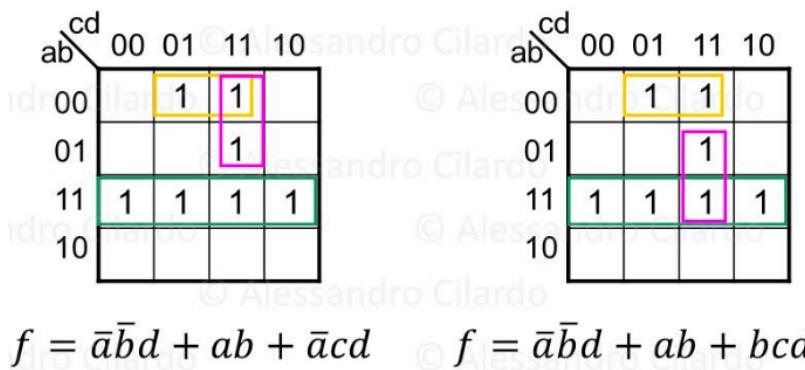
Che tradotta con Karnaugh:



Il primo passo è **individuare il nucleo** (certamente sarà **parte della copertura minima**) osservando le **colonne che hanno un solo 1** ($P_1, P_{12}, P_{13}, P_{14}$); in questo esempio il nucleo sarà costituito dagli implicanti ab e $\bar{a}\bar{b}d$, per cui gli implicanti $P_1, P_3, P_{12}, P_{13}, P_{14}, P_{15}$ risultano coperti. Rimane fuori P_7 , la cui copertura è opinabile tra bcd e \bar{acd} ; la scelta tra i due è **indifferente**, visto che non sono implicanti primi essenziali. La funzione sarà possibile scriverla come:

$$f = \bar{a}\bar{b}d + ab + \bar{a}cd \vee f = \bar{a}\bar{b}d + ab + bcd$$

A questo risultato si può **giungere intuitivamente** dalla **mappa di Karnaugh**:



La selezione degli implicanti primi non essenziali non è obbligata a priori, a differenza di quelli essenziali; la scelta deve essere ottimata secondo le prefissate funzioni di costo e ottenibile in maniera computazionale efficiente. Due fondamentali alternative per la riduzione della matrice di copertura sono il **metodo tabellare** (righe/colonne dominanti) o **algebrico** (metodo di Petrick):

- **Metodo di Petrick**

Esprime la condizione alebrica per la quale **tutte le colonne devono essere coperte almeno da un implicante** attraverso una AND di OR, oppure trasformata in una OR di AND. In questa matrice:

	P_1	P_8	P_9	P_{24}	P_{27}
A= -100-		X	X	X	
B= --001	X		X	X	
C= 0-00-	X	X	X		
D= 11-11					X
F= 110-1				X	X

Il problema della copertura si risolve solo se è presente:

- Per P_1 almeno uno tra B e C, e;
- Per P_8 almeno uno tra A e C, e;
- Per P_9 almeno uno tra A, B e C, e;
- Per P_{24} almeno uno tra A, B e F, e;
- Per P_{27} almeno uno tra D e F.

In forma algebrica:

$$(B + C) \cdot (A + C) \cdot (A + B + C) \cdot (A + B + F) \cdot (D + F) = BAD + BAF \dots$$

Ogni **prodotto della somma** rappresenta un **sottoinsieme di implicanti** per il quale è risolto il problema della copertura. (ad esempio $\{B, A, D\} \dots$)

- **Metodo tabellare**

Il metodo tabellare si basa sul concetto di **righe/colonne dominanti**: **una riga o una colonna L_1 è dominante rispetto ad una L_2 se contiene un soprainsieme delle X in L_2** . Eliminando le righe dominate e le colonne dominanti si ottiene una **matrice equivalente**, cioè che restituisce lo stesso problema di copertura; infatti, una riga dominata corrisponde ad un implicante che copre meno mintermini della sua riga dominante mentre una colonna dominante richiede un soprainsieme degli implicanti richiesti dalla riga dominata, pertanto si possono eliminare.

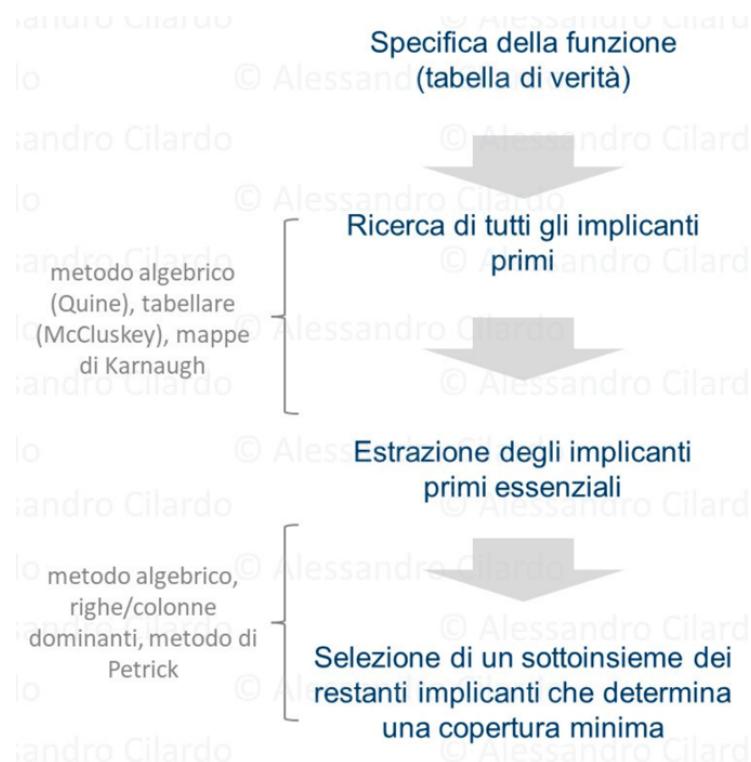
	P_1	P_8	P_9	P_{24}	P_{27}
A= -100-		X	X	X	
B= --001	X		X	X	
C= 0-00-	X	X	X		
D= 11-11					X
F= 110-1				X	X

In questa matrice, D è dominata da F e P_9 domina P_8 , quindi si possono escludere dal problema di copertura.

Ricapitolando, il processo con cui si risolve il problema di copertura con il metodo tabellare prevede che:

1. Si ricerchino gli implicanti primi e quelli essenziali;
2. Si includano i PI essenziali nella forma minima e si rimuovano dalla matrice contestualmente ai mintermini coperti;
3. Si eliminino le righe dominate e le colonne dominanti;
4. Si individuino i PI essenziali “secondari”, cioè quelli che ora sono gli unici a coprire un mintermine;
5. Si reiteri finché possibile.

Con queste informazioni si può **generalizzare il processo di minimizzazione di una funzione logica** nel seguente schema:



Il **processo di minimizzazione** è stato mostrato prendendo in considerazione la forma P, tuttavia si presenta in **forma duale** quando si vanno a considerare **forme S**: al posto di **mintermini** si considerano **maxtermini** e al posto delle **clausole** le **somme elementari**, mentre al posto degli **implicanti** si considerano le **somme che rendono zero la funzione**; i **consensi** sono generati come segue:

$$\begin{aligned}
 f(a, b, c, d) &= (a + \bar{b})(\bar{a} + b + \bar{c} + \bar{d}) \dots (\bar{a} + b + c + \bar{d})(a + \bar{d}) = \\
 &= (a + \bar{b})(c\bar{c} + \bar{a} + b + \bar{d}) \dots (a + \bar{d}) = \\
 &= (a + \bar{b})(\bar{a} + b + \bar{d}) \dots (a + \bar{d})
 \end{aligned}$$

Si noti che **maxtermini** e **somme** sono **definiti in forma duale** rispetto a mintermini e clausole, quindi **corrispondono agli zero della funzione**; di conseguenza nella **mappa di Karnaugh** non si andranno a cercare i **cubi di area massima che coprano solo 1 ma quelli che coprono solo 0**:

	cd	00	01	11	10
ab	00	0			0
	01	0	0		0
	11				
	10	0	0	0	0

In maniera analoga si possono sviluppare gli altri metodi visti per la forma P.

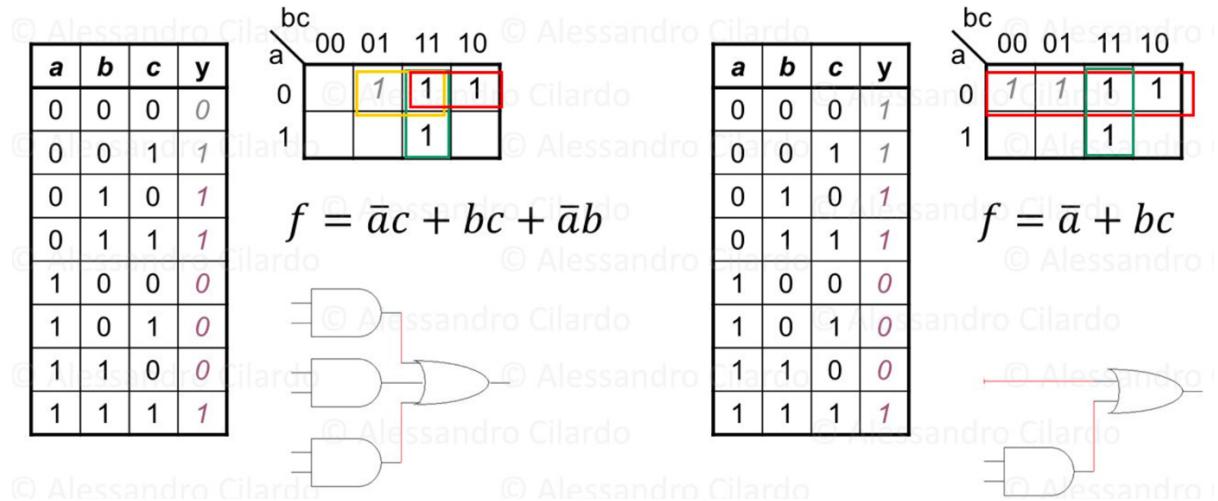
PUNTI DI INDETERMINAZIONE – DON'T CARE

Può capitare che **alcune combinazioni in ingresso** producano un **output** che alla funzione non interessa o che non codifica **alcuna informazione**; questi ingressi sono detti **punti di indeterminazione**, o **don't care**. Ad esempio, su un ingresso di 4 segnali (x_0, x_1, x_2, x_3) si deve codificare un numero BCD (Binary Codec Decimal) ma il numero di combinazioni possibili su 4 bit è $2^4 = 16$; pertanto, 6 numeri binari non verranno corrisposti ad alcuna informazione e la rete deve essere consapevole che i valori da 1010 a 1111 devono essere ignorati. Un don't care viene **identificato in fase di sintesi** da un – che specifica l'**indifferenza del risultato**:

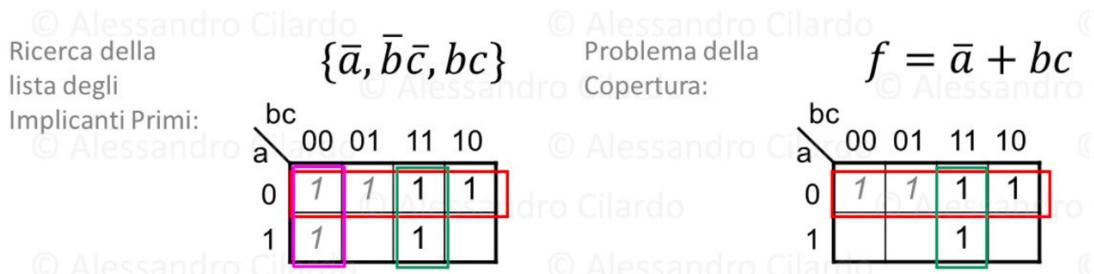
a	b	c	d	y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	-
1	0	1	1	-
1	1	0	0	-
1	1	0	1	-
1	1	1	0	-
1	1	1	1	-

Poiché il **risultato** dei don't care **non è importante** al fine realizzativo della rete e poiché ogni combinazione di risultati specifica una funzione, in una tabella di verità in cui compaiono n don't care ci saranno 2^n funzioni con quel comportamento; infatti, se due funzioni assumono i valori della tabella specificati ma hanno combinazioni diverse di 0 e 1 per i don't care, all'occhio della rete il compito è svolto lo stesso. Queste funzioni sono dette **compatibili** e, sebbene diverse, sono **tutte esaustive per la rete**, sta a chi la progetta scegliere quella con il costo realizzativo minore; in

questa ottica i don't care risultano essere dei jolly che permettono l'implementazione di reti più convenienti:



Si può notare che un numero maggiore di 1 in una tabella di verità consente implicant più larghi, di contro un maggior numero di 0 riduce il numero di mintermini da coprire; quindi, quando si ricercano gli implicant conviene considerare i don't care come 1 e come 0 quando si ricerca la copertura:



ALTRE PORTE LOGICHE FONDAMENTALI

In alcuni processi produttivi risulta più facile la produzione di porte logiche negate come la NAND o la NOR. La funzione NAND, denotata con il simbolo \uparrow , risulta essere la negazione della AND e si rappresenta con De Morgan:

$$x \uparrow y = \overline{x \cdot y} = \bar{x} + \bar{y}$$



La funzione NOR, denotata con il simbolo \downarrow , risulta essere la negazione della OR e si rappresenta con De Morgan:

$$x \downarrow y = \overline{x + y} = \bar{x} \cdot \bar{y}$$



Come si può notare, **da una porta NAND può essere prodotta una porta OR e da una porta NOR una porta AND**; inoltre, **da entrambe si può produrre una NOT**:

$$x \uparrow 1 = \overline{x \cdot 1} = \bar{x}$$

$$x \downarrow 0 = \overline{x + 0} = \bar{x}$$

Oppure:

$$x \uparrow x = \overline{x \cdot x} = \bar{x} = \overline{x + x} = x \downarrow x$$

Poiché da una NAND si possono ottenere una OR e una NOT, si possono ottenere come conseguenza anche una NOR e una AND. Lo stesso vale per la NOR, pertanto si dice che **i due insiemi {NAND} e {NOR} sono funzionalmente completi**, perché da essi si possono produrre qualsiasi funzioni elementari e no.

Grazie alla legge di De Morgan e grazie alla proprietà appena mostrata, **ogni funzione somme di prodotti (forma P) può essere trasformata in una funzione a sole NAND e, dualmente, ogni funzione prodotti di somme (forma S) può essere trasformata in una funzione a sole NOR**:

$$\begin{aligned} f &= \bar{a}c + bc + \bar{a}b + d = \overline{\bar{a}c + bc + \bar{a}b + d} = \overline{\bar{a}c} \cdot \overline{bc} \cdot \overline{\bar{a}b} \cdot \overline{d} = (\bar{a} \uparrow c) \uparrow (b \uparrow c) \uparrow (\bar{a} \uparrow b) \uparrow \bar{d} \\ f &= (\bar{a} + c) \cdot c \cdot (\bar{a} + \bar{c} + d) = \overline{(\bar{a} + c) \cdot c \cdot (\bar{a} + \bar{c} + d)} = \overline{(\bar{a} + c)} + \bar{c} + \overline{(\bar{a} + \bar{c} + d)} = (\bar{a} \downarrow c) \downarrow \bar{c} \downarrow (\bar{a} \downarrow \bar{c} \downarrow d) \end{aligned}$$

Gli esempi portati finora mostrano le due porte con soli due ingressi ma possono essere estese ad un insieme di **n ingressi**, nel quale la legge di De Morgan vale ancora; tuttavia, con **le porte NAND e NOR cessa di valere la proprietà associativa**:

$$\begin{aligned} \overline{x_1 \cdot x_2 \cdot x_3} &= x_1 \uparrow x_2 \uparrow x_3 \neq (x_1 \uparrow x_2) \uparrow x_3 \neq x_1 \uparrow (x_2 \uparrow x_3) \\ \overline{x_1 + x_2 + x_3} &= x_1 \downarrow x_2 \downarrow x_3 \neq (x_1 \downarrow x_2) \downarrow x_3 \neq x_1 \downarrow (x_2 \downarrow x_3) \end{aligned}$$

Un'altra funzione fondamentale è la **funzione implicazione**, essa prende **in input due segnali e restituisce 1 se il primo segnale implica il secondo**, cioè non può accadere che il primo segnale sia 1 senza che il secondo non lo sia:

x	y	$x \Rightarrow y$
0	0	1
0	1	1
1	0	0
1	1	1

La funzione **non è commutativa**, infatti l'ordine con cui vengono specificati gli ingressi conta. L'implicazione **può anche essere espressa come una OR**:

$$f = \bar{x}\bar{y} + \bar{x}y + xy = \bar{x} + y$$

Le funzioni **EQ** (o funzione **equivalenza**) e **XOR** (o funzione **OR esclusivo**) si definiscono come **una la negazione dell'altra**:

- **EQ, denotata con \equiv**

$$x \equiv y = x \cdot y + \bar{x} \cdot \bar{y} = (x + \bar{y}) \cdot (\bar{x} + y)$$

- **XOR, denotata con \oplus**

$$x \oplus y = x \cdot \bar{y} + \bar{x} \cdot y = (x + y) \cdot (\bar{x} + \bar{y})$$

Infatti:

$$x \oplus y = \overline{x \equiv y}$$

Analogamente alla NAND e alla NOR, combinate con 0 o 1 restituiscono una NOT, tuttavia non rappresentano un insieme funzionalmente completo:

$$x \oplus 1 = x \equiv 0 = x \cdot 0 + \bar{x} \cdot 1 = \bar{x}$$

Si possono generalizzare queste due **funzioni su n ingressi**, in tal caso è più facile notare un'ulteriore proprietà che caratterizza la **XOR** e la **EQ**: la prima risulta essere **1 quando il numero di 1 in ingresso è dispari**, mentre **la seconda quando tale numero è pari**; pertanto, le due funzioni possono essere dette anche **funzioni disparità e parità**.

Fino ad ora sono stati mostrati **processi di minimizzazione e reti combinatorie su due livelli** (un livello è un'applicazione di funzioni elementari) per un semplice motivo: quando una rete è a due livelli **riduce significativamente il ritardo del segnale** conseguente al passaggio attraverso una porta; tuttavia, **non sempre questo principio è valido**, ci sono **alcune funzioni** che sviluppate su **più di due livelli** **garantiscono un ritardo minore** rispetto al loro sviluppo su due livelli. Una di queste funzioni è la **XOR**, la cui implementazione circuitale a tre livelli può essere:

$$\begin{aligned} x \oplus y &= \bar{x} \cdot x + x \cdot \bar{y} + \bar{x} \cdot y + \bar{y} \cdot y = \\ &= x \cdot (\bar{x} + \bar{y}) + y \cdot (\bar{x} + \bar{y}) = \\ &= (x \uparrow (x \uparrow y)) \uparrow (y \uparrow (x \uparrow y)) \end{aligned}$$

La **forma minima** di queste due funzioni coincide con la **forma canonica di tipo P**, che equivale a dire che **non esiste un'ottimizzazione migliore** della semplice somma di mintermini; ciò è sostenuto dalla **struttura a scacchiera della mappa di Karnaugh**, che non permette di individuare alcun consenso:

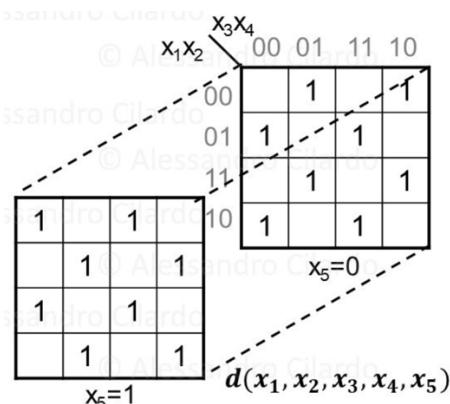
		$x_3 x_4$	00	01	11	10
		$x_1 x_2$	00	1		1
		01	1	1		
		11		1		1
		10	1		1	
$d(x_1, x_2, x_3, x_4)$						

Inoltre, per un numero di ingressi elevato non conviene usare la forma minima in quanto **il costo di porte andrebbe ad aumentare esponenzialmente**; di conseguenza **sono necessari più livelli** per

ottenere una rete conveniente dal punto di vista del costo, la quale va **progettata ad hoc per la specifica configurazione**.

Con la mappa di Karnaugh risulta anche più chiaro il problema: siano presi ad esempio **cinque ingressi**, la relativa mappa sarà una **scacchiera su più dimensioni** dove **non si genera nessun consenso** e dove **il numero di 1 occupa solo la metà delle 2^5 caselle** a disposizione; di conseguenza **il numero di mintermini da considerare nella funzione è:**

$$\frac{2^n}{2} = 2^{n-1}$$



Per realizzare questo circuito saranno necessarie **2^{n-1} porte AND a n ingressi** e **una porta OR con un numero di ingressi pari al numero di mintermini**; ciò porta il **costo di porte a $C_P = 2^{n-1} + 1$** .

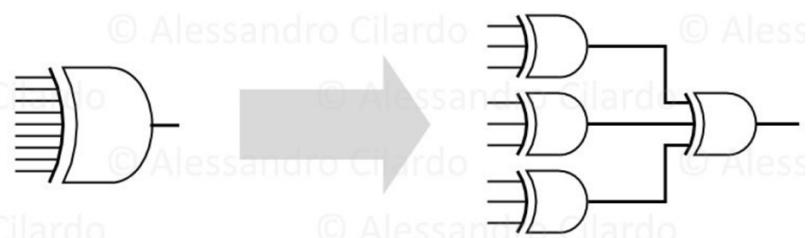
La **funzione disparità** gode della **proprietà associativa** per la quale, **partizionando un insieme di n ingressi X in k sottoinsiemi X^i** :

$$d(x_1, x_2, \dots, x_n) = d[d(X^1), d(X^2), \dots, d(X^k)]$$

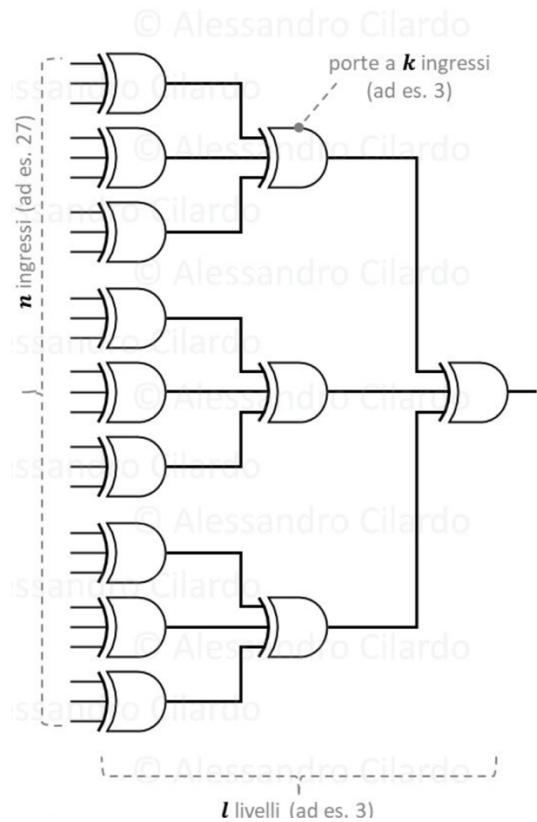
Grazie a questa proprietà, si può affermare che **la funzione sugli ingressi X è alta se è dispari il numero di partizioni X^i in cui il numero di 1 è dispari**. Questa proprietà non vale per la funzione **parità**:

$$1 = p(0,1,1,0,1,1,0,1,1) \neq p[p(0,1,1), p(0,1,1), p(0,1,1)] = 0$$

Ovviamente realizzando prima una funzione disparità con tale proprietà e poi negando l'uscita si può ottenere la funzione parità. Infine, grazie all'associatività, **una XOR può essere impostata come una XOR di XOR**:



Questa configurazione prende il nome di **struttura ad albero**.



Si supponga di disporre di **porte XOR con k ingressi** e di doverle organizzare su una **rete a l livelli**, il **numero di ingressi totali n** che si possono realizzare è:

$$n = k^l$$

Ovviamente se **il numero di ingressi è dato** a priori, il **numero di livelli** sarà:

$$l = \lceil \log_k n \rceil$$

Per quanto riguarda il **costo di porta**, si può affermare che la singola porta XOR “riduce” i k segnali **ad un segnale** (vengono sottratti $k - 1$ ingressi), mentre l’albero riduce n ingressi **ad un segnale**; ciò significa che per passare da n ingressi **ad un’uscita** (ovvero sottrarre $n - 1$ ingressi) serve un **numero di porte pari a**:

$$CP = \left\lceil \frac{n - 1}{k - 1} \right\rceil$$

Questa configurazione ad albero, sebbene abbia **più livelli**, presenta un **costo di porta che cresce linearmente con il numero di ingressi**, che è decisamente **più sostenibile**. Il **costo di ingressi** è, invece:

$$CI = k \left\lceil \frac{n - 1}{k - 1} \right\rceil$$

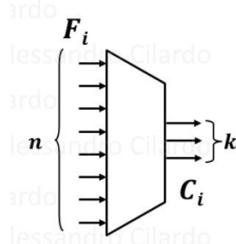
Ovviamente, **avendo più livelli, il ritardo della rete tende ad aumentare** perché il numero di porte e livelli attraversati è maggiore. Questa valutazione diventa **più accurata** se si tiene conto della **struttura interna della XOR**, ovvero una **NOT/AND/OR o NAND**.

RETI COMBINATORIE NOTEVOLI

La **funzione disparità su n ingressi** vista precedentemente è un esempio di **rete combinatoria notevole**, cioè una rete il cui svolgimento ottimale è garantito solo da una **progettazione ad hoc**. Nei calcolatori sono coinvolte molte reti notevoli, come addizionatori, multiplexer, codificatori ecc...

In questa sede si distinguono:

- **Reti di codifica**



È una rete in cui n ingressi decodificati F_i risultano in un valore C_i codificato attraverso k uscite; tra gli n ingressi, però, solo uno può essere alzato e gli altri devono essere necessariamente abbassati, mentre le C_i possono essere rappresentate come la somma dei vari F_i per i quali la cifra i vale 1 nella codifica applicata:

$$C_i = \sum_{j=0}^{n-1} F_j \cdot \tau_j$$

Dove τ_j è il segnale relativo alla cifra j . Un esempio di rete di codifica può associare al numero dell'ingresso alzato il corrispettivo numero BCD:

F_7	F_6	F_5	F_4	F_3	F_2	F_1	F_0	C_2	C_1	C_0
-	-	-	-	-	-	-	1	0	0	0
-	-	-	-	-	-	1	-	0	0	1
-	-	-	-	-	1	-	-	0	1	0
-	-	-	-	1	-	-	-	0	1	1
-	-	-	1	-	-	-	-	1	0	0
-	-	1	-	-	-	-	-	1	0	1
-	1	-	-	-	-	-	-	1	1	0
1	-	-	-	-	-	-	-	1	1	1

Dove:

$$C_0 = F_1 + F_3 + F_5 + F_7$$

$$C_1 = F_2 + F_3 + F_6 + F_7$$

$$C_2 = F_4 + F_5 + F_6 + F_7$$

Può capitare che ci si confonda tra il caso in cui $F_0 = 1$ e $F_i = 0 \forall i > 0$ e il caso in cui $F_i = 0 \forall i \geq 0$; in entrambi i casi si codificherebbe intuitivamente il numero BCD 0 ma in realtà quest'ultimo non corrisponde a nessun valore codificato, pur essendo un segnale di ingresso da tenere in considerazione. Per ovviare a questo problema si implementa un'uscita p che specifica per la validità

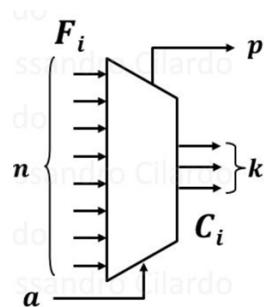
dell'output C_i : essa sarà la OR di tutti gli ingressi, in modo che se almeno un ingresso è alto il valore codificato è accettabile, mentre se tutti gli ingressi sono 0 viene abbassato tale segnale e il risultato ignorato.

$$p = \sum_{i=0}^n F_i$$

Infine, gli ingressi possono essere mascherati da un'ulteriore segnale a che abbassa eventualmente tutti i bit di F_i , rendendolo inefficace; di conseguenza gli effettivi ingressi che vanno al codificatore sono:

$$F'_i = F_i \cdot a$$

Il codificatore in questa ultima configurazione si presenta:



- **Reti di codifica a priorità**

È un codificatore preceduto da una rete a priorità, nella quale più ingressi possono essere alti nel segnale X_i , che verrà poi codificato nel segnale F_i in base alla priorità assegnata ai vari ingressi. Ovviamente la priorità deve essere definita a priori, in genere X_i ha priorità su X_j se $i > j$; di conseguenza il segnale F_i è alto se e solo se è alto il relativo segnale X_i e sono bassi i segnali a priorità maggiore di esso:

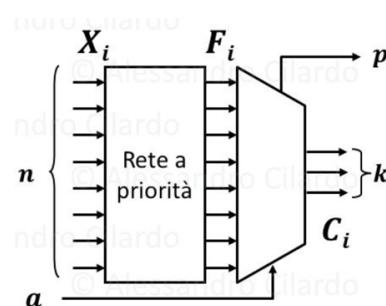
$$F_0 = X_0 \bar{X}_1 \dots \bar{X}_{n-1}$$

$$F_1 = X_1 \bar{X}_2 \dots \bar{X}_{n-1}$$

...

$$F_{n-2} = X_{n-2} \bar{X}_{n-1}$$

$$F_{n-1} = X_{n-1}$$



Questo modello è ottimizzabile, infatti non è necessario specificare le condizioni per ogni F_i in maniera completa: ciascuna uscita C_i è una OR di alcuni ingressi F_i e si può sfruttare il fatto che le codifiche corrispondenti ad alcune combinazioni in ingresso rispettano implicitamente la condizione di priorità:

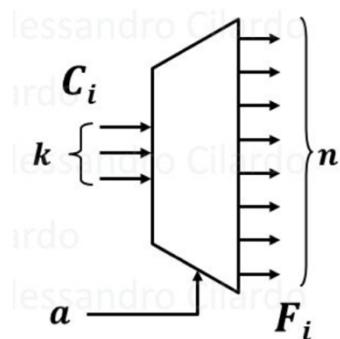
- 7(111) contiene tutti gli ‘1’ delle altre codifiche;
- 5(101) contiene tutti gli ‘1’ delle codifiche di 0(000), di 1(001) e di 4(100);
- 3(011) contiene tutti gli ‘1’ delle codifiche di 0(000), di 1(001) e di 2(010);
- E così via.

Di conseguenza le equazioni generali si semplificano come segue:

$$\begin{aligned} F_7 &= X_7 & F_3 &= X_3 \overline{X_4} \overline{X_5} \overline{X_6} \\ F_6 &= X_6 & F_2 &= X_2 \overline{X_4} \overline{X_5} \\ F_5 &= X_5 \overline{X_6} & F_1 &= X_1 \overline{X_2} \overline{X_4} \overline{X_6} \\ F_4 &= X_4 & F_0 &= X_0 \end{aligned}$$

• Reti di decodifica

È considerabile l'inverso di una rete di codifica, infatti il processo è reversibile e, in questo caso, prende in ingresso un valore codificato su k segnali C_i e restituisce un valore decodificato F_i su n segnali di cui uno solo è alzato. Anche in queste reti è presente la maschera a che azzerà tutte le uscite indipendentemente dagli ingressi.



Ognuna delle uscite F_i può essere rappresentata come il mintermine corrispondente al valore codificato in ingresso:

$$F_1 = C_0 \bar{C}_1$$

Un'alternativa a questo modo di sviluppare una rete decodificata fa uso dei decodificatori incompleti: si supponga di avere quattro segnali in ingresso per codificare le cifre BCD, andranno codificate dieci funzioni distinte individuate dalla seguente tabella:

$C_3 C_2$	00	01	11	10	
$C_1 C_0$	00	F_0	F_4	-	F_8
	01	F_1	F_5	-	F_9
	11	F_3	F_7	-	-
	10	F_2	F_6	-	-

In questo modo si potranno implementare un numero di porte AND pari al numero di porte della rete completa ma con un numero di ingressi variabile, abbattendo il costo CI:

$$F_0 = \overline{C_3} \cdot \overline{C_2} \cdot \overline{C_1} \cdot \overline{C_0}$$

$$F_3 = \overline{C_2} \cdot C_1 \cdot C_0$$

$$F_6 = C_2 \cdot C_1 \cdot \overline{C_0}$$

$$F_9 = C_3 \cdot C_0$$

$$F_1 = \overline{C_3} \cdot \overline{C_2} \cdot \overline{C_1} \cdot C_0$$

$$F_4 = C_2 \cdot \overline{C_1} \cdot \overline{C_0}$$

$$F_7 = C_2 \cdot C_1 \cdot C_0$$

$$F_2 = \overline{C_2} \cdot C_1 \cdot \overline{C_0}$$

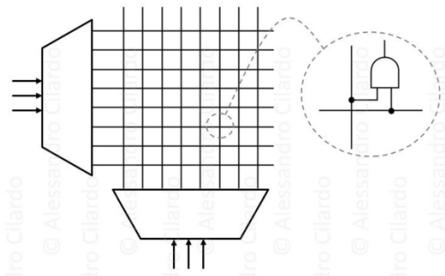
$$F_5 = C_2 \cdot \overline{C_1} \cdot C_0$$

$$F_8 = C_3 \cdot \overline{C_0}$$

Più decodificatori possono essere composti per formare degli schemi più ampi:

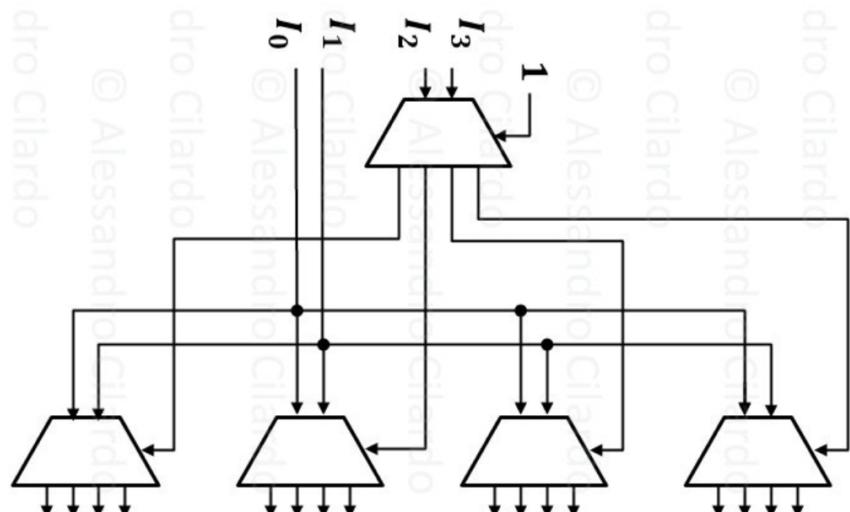
- **Schema a matrice**

Gli ingressi sono divisi in due gruppi (righe e colonne), dove ogni gruppo entra in un decodificatore; ad ogni incrocio (ci saranno in totale righe×colonne incroci) si colloca una AND che vedrà in ingresso due 1 solo in corrispondenza di quella particolare combinazione di ingressi che vede alta la riga e la colonna su cui è collocata. In questo modo si realizza un decodificatore con un ridotto numero di ingressi ma una quantità elevata di uscite (nel caso di sei ingressi, divisi in gruppi di tre, si ottiene un 6:64).



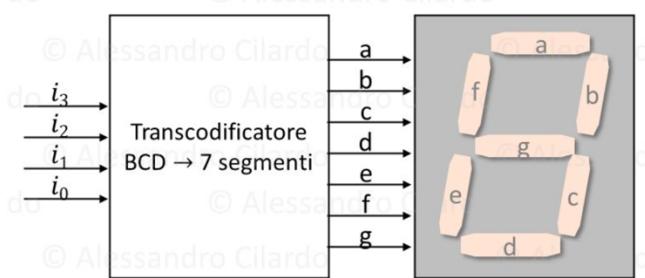
- **Schema ad albero**

In un classico schema ad albero può capitare che alcuni segnali di ingresso siano collegati direttamente con i decodificatori più esterni, mentre i restanti pilotano un ulteriore decodificatore che determina quale dei decodificatori esterni sia effettivamente attivo e alzandone il relativo segnale di abilitazione. Nell'esempio seguente viene proposto un decodificatore ad albero 4:16 :



- **Reti di transcodifica**

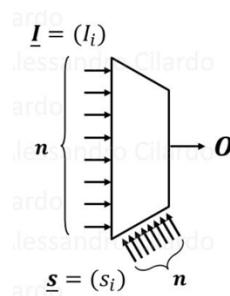
Con queste reti si può rappresentare un dato mediante due codifiche differenti, associando la parola codice C_1 alla parola codice C_2 . Ad esempio un transcodificatore BCD prende in ingresso un numero BCD e fornisce in uscita una codifica dello stesso numero su 7 bit che rappresenta quali segmenti di un display illuminare:



- **Multiplexer**
 - **Lineare**

Un multiplexer binario lineare prende in ingresso un insieme di n ingressi “dato” $\underline{I} = (I_i)$ primari e n ingressi “indirizzo/selezione” $\underline{S} = (S_i)$ secondari decodificati (quindi mutuamente esclusivi) e considera una sola uscita O pari all’ingresso dato I_i indicato dall’ingresso di selezione S_i attivo; lo scopo di questo circuito è selezionare l’ingresso da collegare all’uscita tra i vari I_i .

$$O = \sum_{i=0}^{n-1} I_i \cdot s_i$$



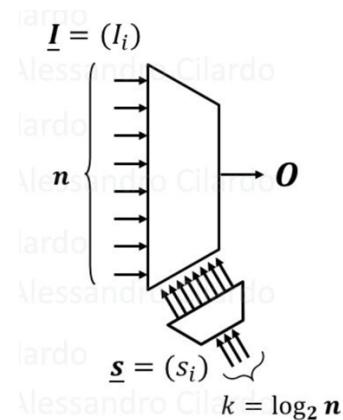
Un utilizzo di questo circuito può essere quello di selezionare il contenuto del registro specificato nell’istruzione di un programma.

- **Codificato**

Un multiplexer binario codificato prende in ingresso un insieme di n ingressi “dato” $\underline{I} = (I_i)$ primari e $k = \log_2 n$ ingressi “indirizzo/selezione” $\underline{S} = (S_i)$ secondari codificati e considera una sola uscita O pari all’ingresso dato I_i dove i è codificato dall’ingresso indirizzo \underline{S} :

$$O = \sum_{i=0}^{n-1} I_i \cdot P_i(\underline{S})$$

Dove $P_i(\underline{S})$ è il mintermine i delle k variabili in \underline{S} .



Ad esempio:

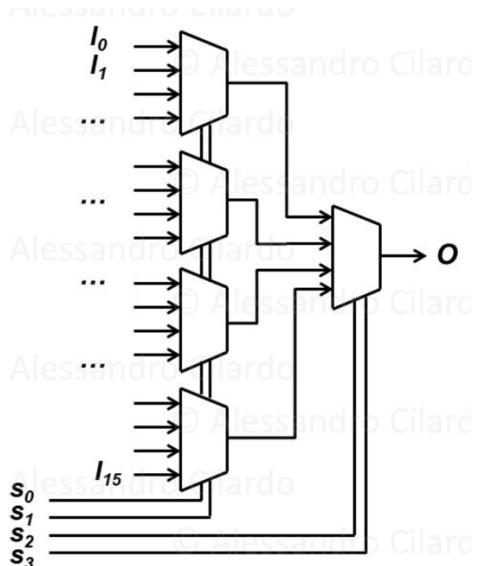
s_2	s_1	s_0	O
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7

$$O = \sum_{i=0}^7 I_i \cdot P_i(s_2, s_1, s_0)$$

$$= I_0 \cdot \bar{s}_2 \bar{s}_1 \bar{s}_0 + I_1 \cdot \bar{s}_2 \bar{s}_1 s_0 + \dots + I_6 \cdot s_2 s_1 \bar{s}_0 + I_7 \cdot s_2 s_1 s_0$$

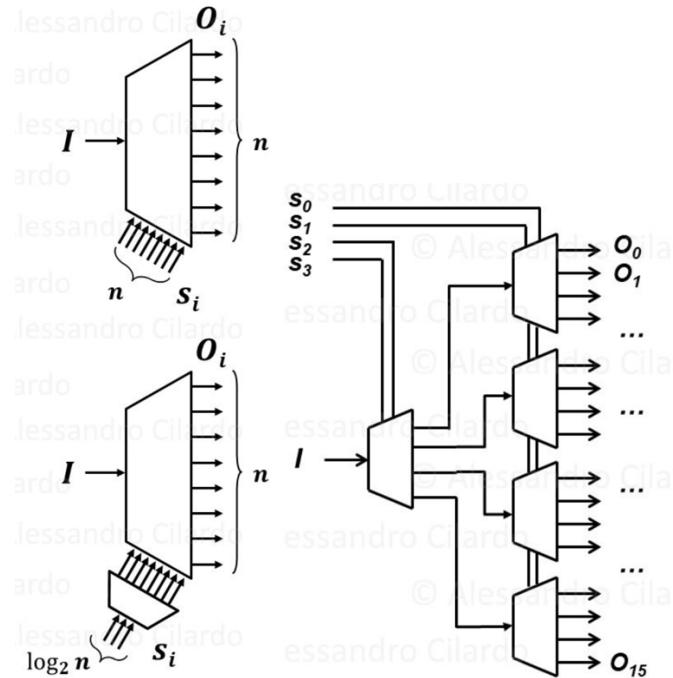
○ Composto

Non è altro che un multiplexer lineare composto da tanti piccoli multiplexer:



- **Demultiplexer**

È una rete in grado di collegare l'ingresso I ad una delle n uscite e, proprio come il multiplexer, può essere lineare, codificato o composto.



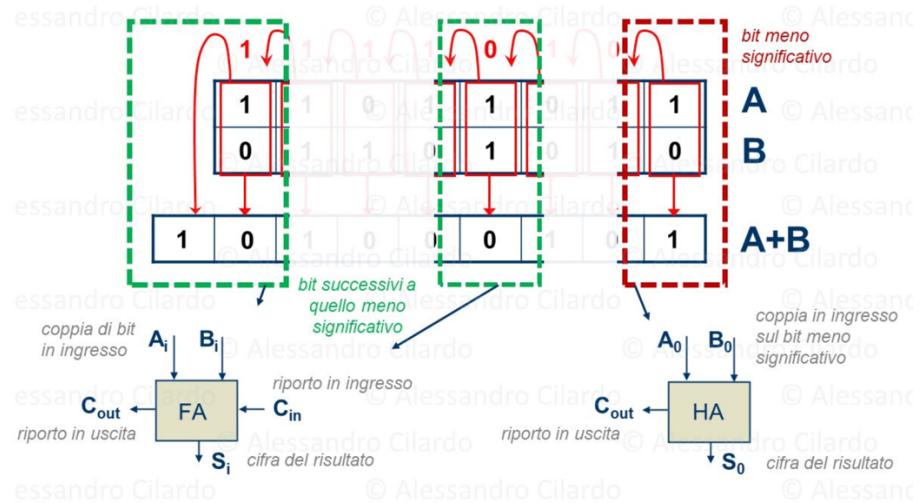
Nell'esempio seguente si può notare come ciascuna uscita è sempre zero, tranne nel caso in cui il valore codificato dai segnali di selezione è pari al pedice dell'uscita:

S_2	S_1	S_0	O_2
0	0	0	0
0	0	1	0
0	1	0	I
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

- **Addizionatore**

Per scrivere il circuito di un addizionatore si sfrutta la sua proprietà iterativa: ogni coppia di bit in ingresso e il riporto entrante da destra possono essere processati da una diversa cella che realizza la funzione.

Viene così definito l'addizionatore completo (Full Adder) e il semiaddizionatore (Half Adder): il primo effettua l'addizione di due bit, ai quali viene combinato anche il riporto entrante, e viene restituito il risultato con un riporto uscente, mentre per il secondo non viene implementato l'ingresso per il riporto entrante; l'utilizzo di queste due reti viene schematizzato come segue:



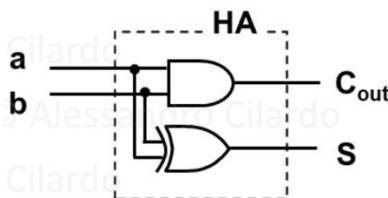
Per il HA si identifica la seguente tabella di verità:

A_i	B_i	S	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Le cui equazioni sono:

$$S = a \oplus b = d(a, b) = \bar{a}b + a\bar{b}$$

$$C_{out} = a \cdot b$$



Per il FA si identifica la seguente tabella di verità:

C_{in}	A_i	B_i	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Mentre valgono le seguenti uguaglianze:

$$S = a \oplus b \oplus C_{in} = D(a, b, C_{in}) = \bar{a}\bar{b}C_{in} + \bar{a}b\bar{C}_{in} + a\bar{b}\bar{C}_{in} + abC_{in}$$

$$C_{out} = \bar{a}bC_{in} + a\bar{b}C_{in} + ab\bar{C}_{in} + abC_{in} = ab + bC_{in} + aC_{in} = ab + C_{in}(a + b)$$

È possibile esprimere queste quantità in funzione di $P = a \oplus b$ e $G = ab$:

$$S = (a \oplus b) \oplus C_{in} = P \oplus C_{in}$$

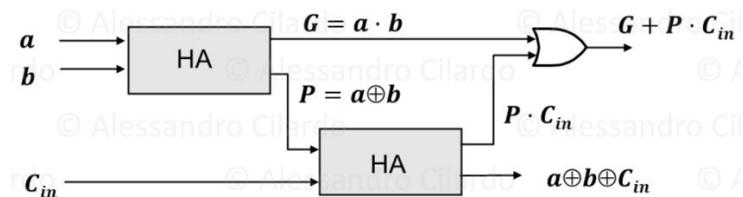
$$\begin{aligned} C_{out} &= ab + bC_{in} + aC_{in} = ab + abC_{in} + \bar{a}bC_{in} + ab\bar{C}_{in} = ab + \bar{a}bC_{in} + ab\bar{C}_{in} \\ &= ab + C_{in}(a \oplus b) = G + C_{in} \cdot P \end{aligned}$$

Da cui derivano le equazioni del Full-Adder:

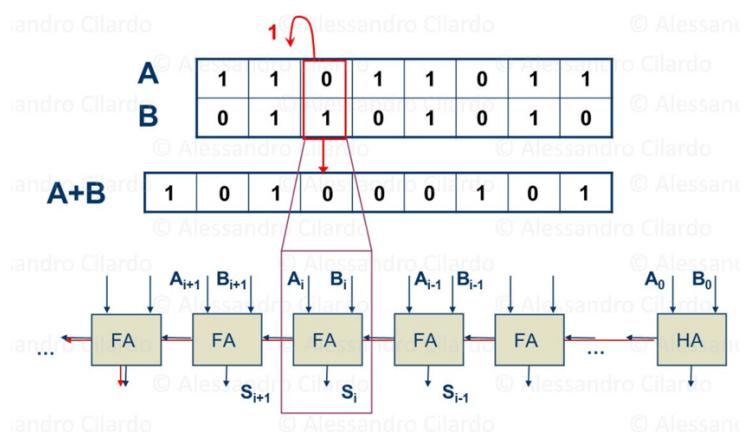
$$S = P \oplus C_{in}$$

$$C_{out} = G + C_{in} \cdot P$$

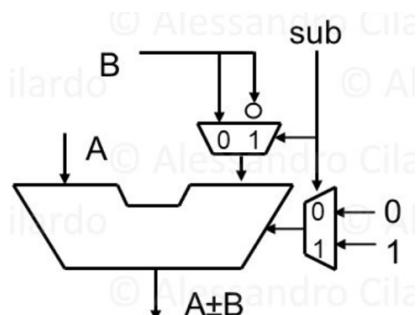
Ciò significa che un FA è possibile ottenerlo a partire da due HA:



Quindi, un addizionatore è composto come segue:



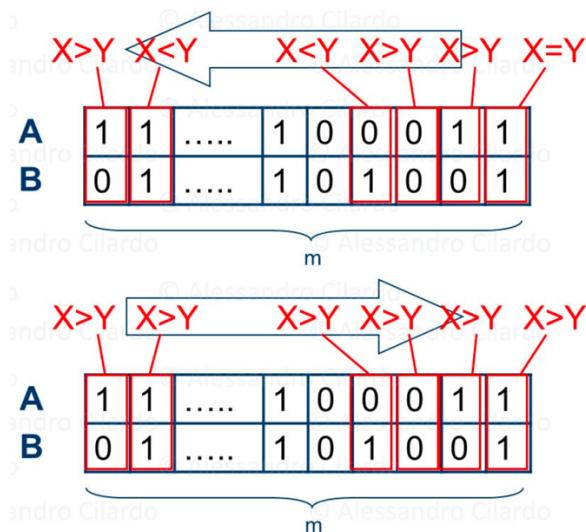
Per quanto riguarda la sottrazione, è stato già visto in precedenza come essa possa essere considerata come una variante dell'addizione; di conseguenza o si pone una NOT prima del segnale B, ma in tal caso bisognerebbe implementare due circuiti per le due operazioni, o si può implementare un addizionatore-sottrattore, che presenta anche un ingresso che codifica l'operazione (tramite un multiplexer) e che regola di conseguenza gli operandi:



Quando il segnale *sub* è 0 si effettua un'addizione, quando è 1 si inverte il valore di B effettuando una sottrazione.

- **Comparatore**

È un circuito che effettua il confronto di due numeri binari sia da destra che da sinistra: rispettivamente, l'ultima o la prima differenza trovata indica quale dei due numeri è più grande (in quanto si va a comparare il bit con più significato). In entrambi i casi, il confronto è effettuato "localmente" tra una singola coppia di bit, tenendo conto dell'esito precedentemente effettuato:



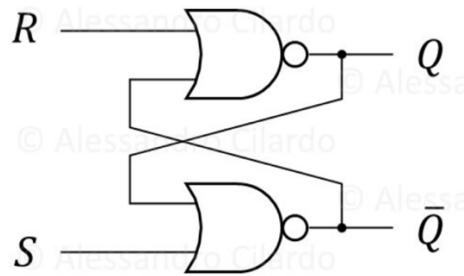
RETI LOGICHE SEQUENZIALI

In certe situazioni è necessario implementare delle reti capaci di **registrare un dato e di mantenerlo nel tempo**, le reti logiche sequenziali; in esse l'uscita in un istante non è funzione dei dati in ingresso relativi a quell'istante ma anche di tutti valori negli istanti precedenti. Per fare ciò, sono necessari circuiti combinatori elementari che permettono l'**autosostentimento di un dato al fine di non farlo mutare nel tempo**: attraverso una **retroazione** un dato in uscita torna in ingresso, viene **negato due volte e si autosostiene**. Lo schema basilare è il seguente e viene definito **bistabile**:



Tuttavia, questa rete non permette all'esterno di modificare il valore dello stato memorizzato all'interno.

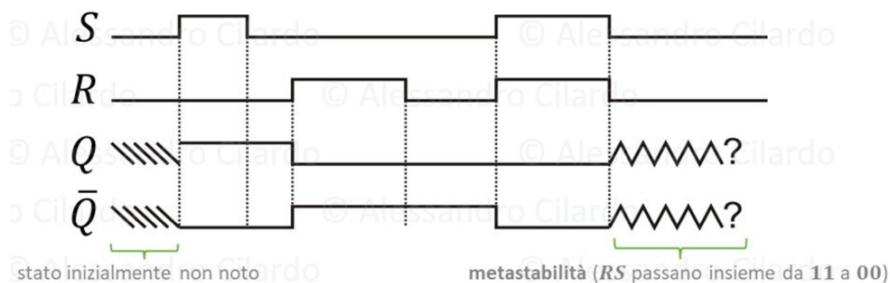
Sostituendo la NOT con una NOR, è possibile modificare dall'esterno lo stato memorizzato, agendo sugli ingressi esterni *R* (reset) e *S* (set) con una rete che viene chiamata **Latch RS** (pur mantenendo le caratteristiche di un bistabile):



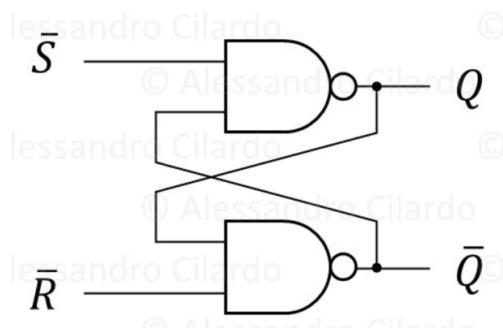
Quando questi due segnali sono **entrambi abbassati** le NOR si comportano come le NOT e lo stato viene conservato, quando **uno dei due segnali si alza** si forza un'uscita ad un preciso valore (ad esempio se S è 1, Q diventa 1 e \bar{Q} diventa 0), mentre quando i due ingressi tornano a 0 la rete permane nello stato impostato dall'ultimo ingresso e memorizza il valore.

Nonostante siano indicati così, **i due ingressi \bar{Q} e Q non sono legati** e il loro valore non è necessariamente opposto, infatti questa nomenclatura è puramente una convenzione che indica il loro “usuale” essere opposti ma può capitare, come nel caso in cui gli ingressi RS assumono entrambi il valore 1, che assumano lo stesso valore, nell'esempio 0.

Un **comportamento anomalo da evitare** si può riscontrare quando i due ingressi R e S passano simultaneamente a 0, in tal caso si entra in uno **状態の metastabilität**; il circuito si comporta come una NOT: in un istante sia in ingresso che in uscita si vede il valore zero, che viene portato ad 1, ma poi tornando in ingresso viene riportato a 0, e così via... Il circuito **oscilla tra i valori 0 e 1 per un tempo non predicibile, fino a raggiungere una configurazione stabile ma aleatoria per i valori \bar{Q} e Q** ; infatti le condizioni che codifica questo stato non sono logiche ma fisiche (le due porte potrebbero avere delle differenze di costruzione ed oscillare in modi diversi) e non possono essere utilizzate ai fini della rete:



Questo stesso circuito può essere realizzato con le NAND, tuttavia in tal caso **andranno usati i segnali R e S in forma duale rispetto al precedente circuito**: il valore neutro sarà 1 e la modifica dello stato avverrà con l'abbassamento a 0 di uno dei due segnali. Per convenzione, nella schematizzazione grafica i segnali saranno implementati in forma negata:



Il cambiamento di stato si suppone avvenga istantaneamente, con i fronti di discesa o di salita completamente verticali; tuttavia, **nella realtà fisica il cambiamento avviene con un tempo non nullo** e i fronti presentano delle discese o delle salite non verticali, il che **contribuisce all'aleatorietà delle uscite \bar{Q} e Q durante la condizione di metastabilità**.

Per i bistabili si possono distinguere **due tabellazioni**:

- **Tabella funzionale**, indica quale sarà il prossimo stato $Q(t + 1)$ per ogni coppia di segnali R, S e stato corrente $Q(t)$;

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

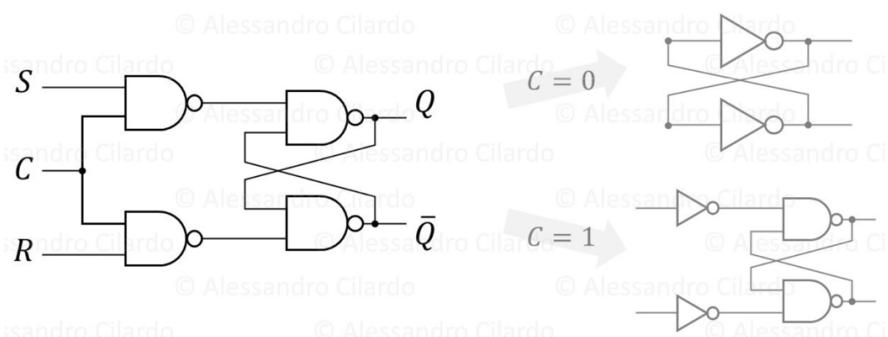
© Alessandro Cilardo
tabella **funzionale** per il Latch RS
con porte NOR

- **Tabella di eccitazione**, indica la combinazione di ingressi R e S necessaria per la transizione da $Q(t)$ a $Q(t + 1)$;

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

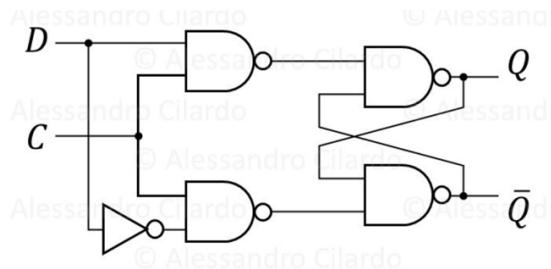
© Alessandro Cilardo
tabella di **eccitazione** per il Latch
RS con porte NOR

Una variante del dispositivo appena mostrato è il **Latch RS con abilitazione**: viene implementato nel circuito un **segnale C** (controllo, o Clock) che, collegato ai segnali R e S con una **NAND**, permette di memorizzare i dati anche ignorando i valori dei segnali in ingresso; infatti, quando **C è alzato (1)** il Latch si comporta come un normale RS, mentre quando **C è basso (0)** il dispositivo memorizza lo stato indipendentemente dai segnali R e S , che possono assumere anche configurazioni non ammesse (come quelle che conducono alla metastabilità).



Un altro modo di eliminare l'indeterminazione del caso $RS = 11$ è l'**implementazione di un solo ingresso** (Dato, D) che viene **memorizzato nello stato Q**: quando **C = 1** il latch è trasparente

all'ingresso, cioè lo stato segue l'ingresso, mentre **lo stato Q congela l'ingresso quando $C = 0$** . Questo dispositivo prende il nome di **Latch D**, e si costruisce come segue:



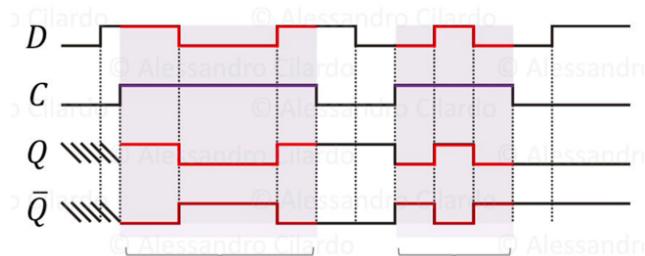
$Q(t) \rightarrow Q(t + 1)$	D
$0 \rightarrow 0$	0
$0 \rightarrow 1$	1
$1 \rightarrow 0$	0
$1 \rightarrow 1$	1

tabella di eccitazione per il Latch D (quando abilitato)

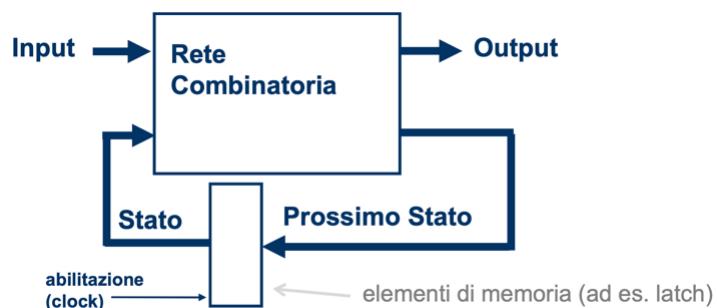
D	Q	$Q(t + 1)$
0	0	0
0	1	0
1	0	1
1	1	1

tabella funzionale per il Latch D (quando abilitato)

Queste tabelle sono **riferite al caso di latch abilitato**, $C = 1$, nella configurazione opposta ($C = 0$) lo stato rimane in ogni caso inalterato. Nella **tempificazione del Latch D** si può notare con maggior evidenza il concetto di **trasparenza**:



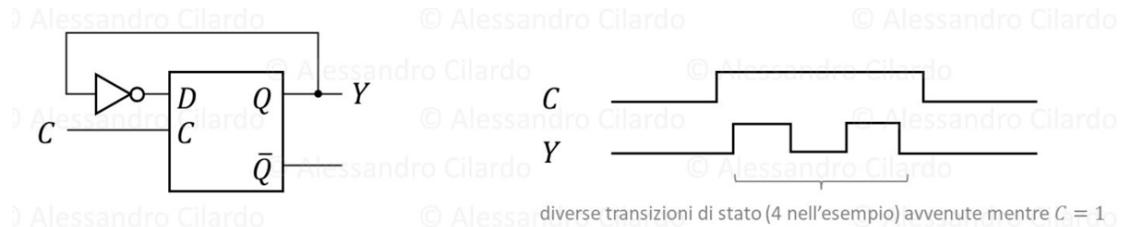
Questi **elementi di memoria** saranno spesso usati come **“registri”** per **memorizzare lo stato corrente di una macchina sequenziale**: da tale stato dipende, attraverso un’opportuna rete combinatoria, lo stato prossimo, a sua volta ingresso per gli elementi di memoria. A causa dei diversi **ritardi della rete**, le uscite possono presentare delle **transizioni “spurie”** (non previste) prima di assestarsi sui loro **valori definitivi**; nell’utilizzo di Latch RS abilitati o D, per tutto il tempo in cui l’abilitazione è alta ($C = 1$) gli elementi di memoria sono trasparenti e sensibili a queste transizioni.



Ad esempio, si consideri una macchina sequenziale che realizzi la funzione:

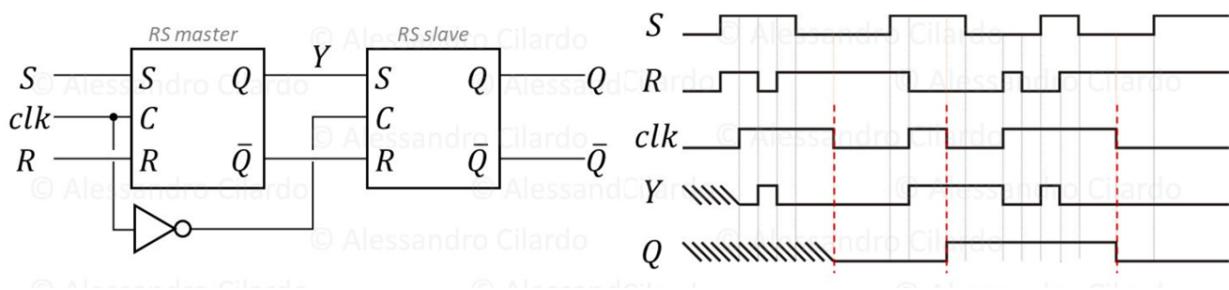
$$Stato = Uscita = Y \wedge Stato Prossimo = \bar{Y}$$

Ad ogni impulso di C lo stato Y viene aggiornato con \bar{Y} , tuttavia il latch è trasparente per tutto l'intervallo di tempo in cui $C = 1$, quindi l'aggiornamento di stato si ripete più volte durante questo intervallo, determinando **transizioni aleatorie (in un numero che dipende sia dalla durata della condizione $C = 1$ che dalla velocità di risposta del latch):**



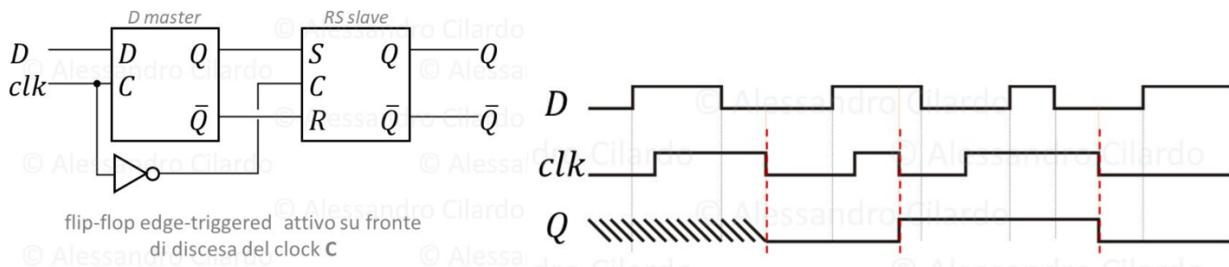
Una possibile soluzione a questo problema è **bloccare la propagazione del nuovo valore dall'ingresso all'uscita dell'elemento di memoria**, ovvero **impedire la trasparenza** del latch: non deve essere mai possibile che l'uscita dipenda dall'ingresso applicato nello stesso istante. A tale scopo si implementano i **flip-flop**, reti sequenziali sincrone che **cambiano l'uscita solo in particolari istanti in cui varia il segnale di abilitazione (C, clock)**; ciò significa che l'uscita è **sensibile all'ingresso solo in un intervallo di tempo estremamente limitato**, di durata pressoché nulla. I flip-flop possono essere di due tipi: Master-Slave (o Pulse-Triggered) o Edge-Triggered.

Un **flip-flop master-slave** è composto a partire da **due latch RS**, un master ed uno slave. Quando l'abilitazione è alta, $C = 1$, l'ingresso è “osservato” dal master, che risulta essere **trasparente**: la sua uscita Y può variare, ma lo slave è **disabilitato e non la legge**; quando $C = 0$, il valore è “memorizzato” dallo slave, mentre il master è **disabilitato e mantiene la sua uscita Y ad un valore costante**. Lo slave cambia stato solo in corrispondenza dell'istante in cui il segnale di abilitazione passa da alto a basso (fronte di discesa), mentre con una NOT al segnale C si possono ottenere cambiamenti sul fronte di salita. Considerando un flip-flop master slave su fronti di discesa:



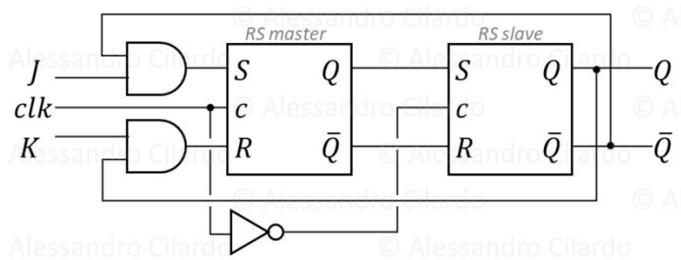
Il problema relativo all'uso dei latch viene **risolto solo in parte** da questo tipo di flip-flop: è non-trasparente agli ingressi e la transizione in uscita avviene sul fronte di C $1 \rightarrow 0$; tuttavia, è ancora sensibile alle transizioni in ingresso: il latch RS master è direttamente esposto agli ingressi, quindi eventuali transizioni contemporanee degli ingressi RS da 11 a 00 causeranno ancora metastabilità.

Un **flip-flop edge-triggered** è composto da **un latch D e un latch RS**: esso legge (“campiona”) il valore dell'ingresso soltanto sui **fronti di clock**, non durante tutta la fase attiva (ad esempio, nell'istante del fronte di discesa di C $1 \rightarrow 0$, non per tutto il tempo in cui $C = 1$); infatti, questo tipo di flip-flop **funziona in modo da “congelare” l'ultimo valore seguito da D**.



Il primo latch legge l'ingresso e il secondo lo riporta in uscita solo in corrispondenza dell'istante in cui il segnale di abilitazione **C** passa da 1 a 0.

Un ulteriore tipo di flip-flop master slave è di tipo JK (flip-flop JK): il **clock C** tempifica questo flip-flop come nel caso standard di un **master-slave**, sono però presenti gli ingressi **JK** che fungono da **SR**; tuttavia, il caso **JK = 11** è ammissibile e codifica per la **modalità “toggle”** (se lo stato **Q** è 1, diventa 0 sul fronte successivo e viceversa):



Quando **JK** è **11** (cioè in modalità toggle) uno dei due ingressi verrà mascherato dalla corrispondente porta AND in base a quali sono i valori di **Q** e **Q-bar**. Le tabelle funzionale e di eccitazione di questo dispositivo sono:

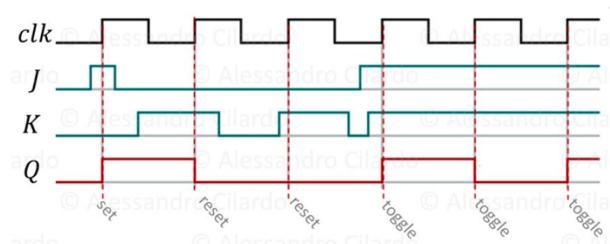
$Q(t) \rightarrow Q(t+1)$	K	J
0 → 0	-	0
0 → 1	-	1
1 → 0	1	-
1 → 1	0	-

tabella di eccitazione per il flip-flop JK

K	J	$Q(t+1)$
0	0	$Q(t)$
0	1	1
1	0	0
1	1	$\overline{Q(t)}$

tabella funzionale per il flip-flop JK

Gli ingressi **JK** possono essere implementati anche con una tempificazione edge-triggered:



Il flip-flop può essere implementato anche solo con un ingresso T (flip-flop T), che permette l'operatività solo in modalità **toggle**: **T = 0** lo stato **Q** rimane inalterato, **T = 1** sul successivo fronte d'onda lo stato **Q** verrà invertito. Questo circuito può essere realizzato da un flip-flop JK

collegando i due segnali, oppure con realizzazioni **ad hoc per avere a disposizione la funzionalità toggle; le relative tabelle sono:**

$Q(t) \rightarrow Q(t + 1)$	T
0 → 0	0
0 → 1	1
1 → 0	1
1 → 1	0

tabella di *eccitazione* per il flip-flop T

T	Q	$Q(t + 1)$
0	0	0
0	1	1
1	0	1
1	1	0

tabella *funzionale* per il flip-flop T

È possibile implementare (opzionalmente) in ognuno di questi dispositivi un'altra **coppia di segnali, usati per impostare lo stato** (altrimenti indeterminato) **in cui gli elementi di memoria si trovano all'accensione**:

- Direct set (o **preset**), impone lo stato di Q alto;
- Direct clear (o **clear**), impone lo stato di Q basso.

Questi due segnali possono essere “**asincroni**”, cioè possono agire **indipendentemente dal segnale clock** in qualsiasi momento, o “**sincroni**”, cioè possono agire **solo quando l'elemento di memoria è abilitato dal segnale clock**.

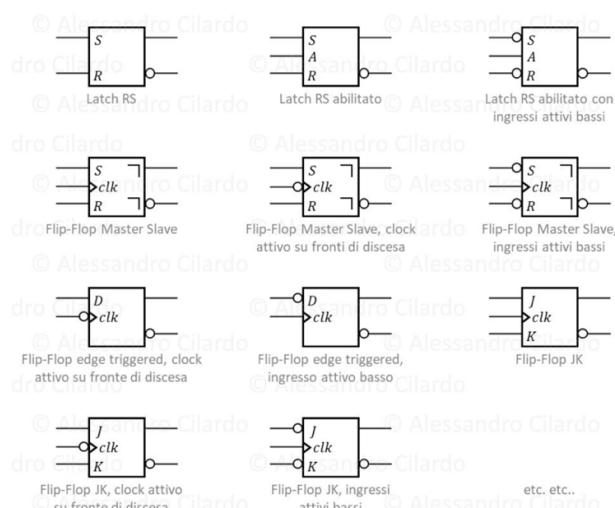
Riepilogando, gli elementi di memoria possono essere **classificati per tipo di temporizzazione**:

- **Latch**;
- **Flip-flop master-slave** (o pulse-triggered);
- **Flip-flop attivo su fronti** (o **edge-triggered**).

O per **funzionamento degli ingressi**:

- Tipo **RS**;
- Tipo **D**;
- Tipo **JK**;
- Tipo **T**.

Ovviamente ognuno di questi dispositivi ha le **proprie tabelle funzionale e di eccitazione**; le relative **notazioni grafiche** sono:



Generalizzando quanto detto finora, le reti sequenziali sono reti dotate di memoria, in grado di realizzare **funzioni complesse dipendenti dalla “storia”**, o sequenza temporale, degli ingressi ad essa applicati. Per tali reti è necessaria l'introduzione della nozione di stato: attraverso un'opportuna funzione, **dallo stato corrente della rete e dagli ingressi dipendono lo stato prossimo e l'uscita della rete**; lo stato corrente viene memorizzato con **elementi di memoria**, che fungono da registri.

È necessario trovare una metodologia per comporre reti combinatorie ed elementi di memoria al fine di ottenere un **qualsiasi comportamento sequenziale complesso**, valutando anche **come calcolare le funzioni da porre in ingresso ai registri stessi**.

La scelta del tipo di elementi di memoria influenza notevolmente il progetto (in particolare il fatto che possono essere **latch o flip-flop**) ma anche **la differenza di tempificazione** (master-slave o edge-triggered) per la quale **cambia il modo in cui la macchina risponde al segnale di abilitazione** (o clock) cambiando il proprio stato. Inoltre, in base al tipo di elemento usato, **cambiano anche i segnali** (detti segnali di posizionamento) **che la rete combinatoria deve generare in ingresso agli elementi di memoria**.



Le reti sequenziali possono quindi essere differenziate in:

- **Reti sincrone**

Il comportamento di queste reti è definito solo in **istanti discreti** e gli elementi di memoria cambiano stato solo in tali circostanze, esse sono **sincronizzate da un particolare segnale**, detto clock (comune a tutti i componenti con memoria del sistema), **oppure dagli ingressi stessi** (reti autosincronizzate). Se opportunamente progettati, i sistemi sincroni hanno un funzionamento indipendente da differenze di ritardi nei singoli componenti.

- **Reti asincrone**

Il comportamento di queste reti **dipende dai valori degli ingressi in ogni istante**, quindi hanno tempo continuo; inoltre, **la successione delle transizioni degli ingressi determina l'evoluzione dello stato**. Le reti sincrone sono un **particolare caso di rete asincrona**, perché il clock è un segnale come gli altri, ma l'astrazione di questi sistemi semplifica enormemente il progetto; infatti, **le reti asincrone richiedono un progetto più complesso** e, pertanto, vengono spesso impiegate in specifici componenti usati all'interno di sistemi sincroni.

ANALISI E SINTESI DI UNA RETE SEQUENZIALE

L'**analisi di una rete parte da un dato circuito** (o da equazioni booleane) e **determina il comportamento della rete**, mentre la **sintesi parte da un comportamento richiesto** (detto specifica) e **deriva un circuito ad esso conforme**. Per un esercizio di **analisi** andranno considerate le **tabelle funzionali** relative agli elementi di memoria usati nella rete (le **tabelle di eccitazione** per

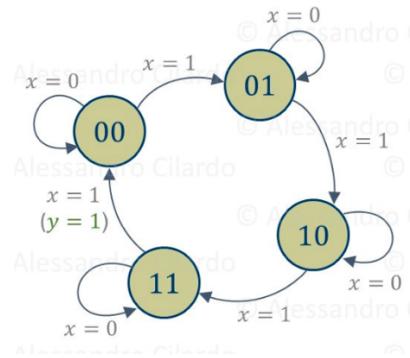
la **sintesi**) e per ogni combinazione di ingresso x_i e stato corrente Q_i andranno derivati i valori dei segnali di posizionamento e dello stato prossimo.

x	Q_1	Q_0	K_1	J_1	K_0	J_0	Q_1'	Q_0'	y
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1	0
0	1	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	1	0
1	0	0	0	0	1	1	0	1	0
1	0	1	1	1	1	1	1	0	0
1	1	0	0	0	1	1	1	1	0
1	1	1	1	1	1	1	0	0	1

Il **comportamento di una rete** può essere descritto tramite un **grafo di transizione di stato**, una rappresentazione caratterizzata da:

- **Nodi**, tutti i possibili stati in cui può trovarsi una macchina sequenziale (nominati con una codifica a piacere, a meno che non si tratti di contatori);
- **Archi**, tutte le possibili transizioni da uno stato all'altro, causate da ciascun valore dell'ingresso applicato quando la macchina si trova in ciascun differente stato.

Se le uscite dipendono dallo stato e dall'ingresso, gli archi riporteranno anche i valori delle corrispondenti uscite (che possono dipendere anche solo dallo stato, in tal caso saranno indicate solo nel nodo).



Il **grafo** e la **tabella** mostrati costituiscono **due modi per rappresentare una FSM** (Finite State Machine o Macchina/Automa a Stati Finiti), un **modello formale per la rappresentazione di sistemi sequenziali** comunemente impiegato in molti ambiti dell'ingegneria dell'informazione. Esistono essenzialmente **due varianti** di FSM:

- **Macchine di Melay**

La definizione di macchina sequenziale a stati finiti di Melay prevede che essa sia una **quintupla ordinata** $M(Q, I, U, t, \omega)$, dove:

- Q è un insieme finito di “Stati Interni”;

- I è un insieme finito di “Stati di Ingresso”;
- U è un insieme finito di “Stati di Uscita”;
- t è una funzione $Q \times I \rightarrow Q$, detta funzione stato prossimo;
- ω è una funzione $Q \times I \rightarrow U$, detta funzione uscita.

La funzione uscita ω dipende dallo stato e dall’ingresso applicato alla macchina.

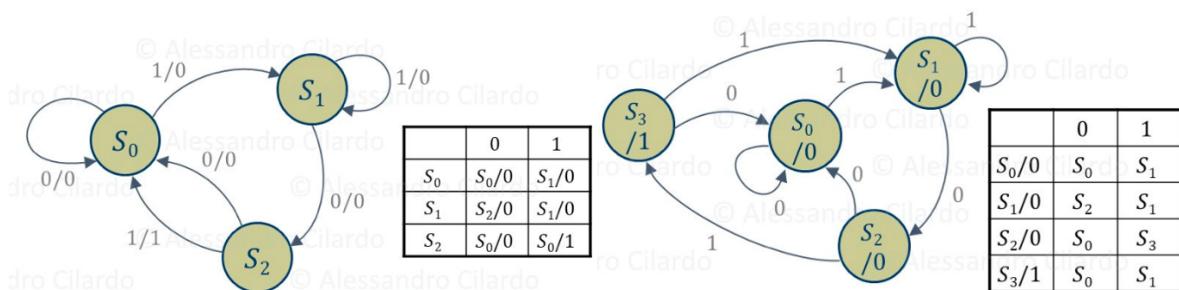
- **Macchine di Moore**

La definizione di macchina sequenziale a stati finiti di Moore prevede che essa sia una **quintupla ordinata $M(Q, I, U, t, \omega)$** , dove:

- Q è un insieme finito di “Stati Interni”;
- I è un insieme finito di “Stati di Ingresso”;
- U è un insieme finito di “Stati di Uscita”;
- t è una funzione $Q \times I \rightarrow Q$, detta funzione stato prossimo;
- ω è una funzione $Q \rightarrow U$, detta funzione uscita.

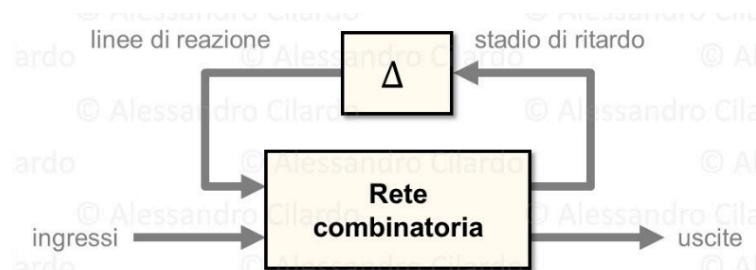
La funzione uscita ω dipende solo dallo stato e non anche dall’ingresso applicato alla macchina.

Il comportamento di una macchina sequenziale può essere descritto come una **tabella Stato/Ingresso → prossimo stato** o come un **grafo di transizione di stato**: in una macchina di Melay l’uscita è associata all’arco, mentre in una macchina di Moore al nodo.



Nel primo caso il grafo rappresenta una macchina di Melay in grado di riconoscere la sequenza 101, nel secondo una macchina di Moore con lo stesso scopo; per ottenere un comportamento equivalente alla macchina di Melay, nella macchina di Moore è stato necessario aggiungere uno stato.

Esiste un **modello realizzativo generale di una rete sequenziale**, il **modello di Huffman**. Tale modello prevede che **una macchina sequenziale possa essere realizzata con una macchina combinatoria** (che riceve l’ingresso esterno e lo stato corrente e calcola l’uscita esterna e lo stato prossimo) e **uno stadio di ritardo**, che ritarda fino al prossimo passo la transizione dallo stato corrente a quello prossimo. Nel modello astratto **il funzionamento dell’elemento di ritardo non è specificato**, infatti non ha ancora una diretta corrispondenza fisica.

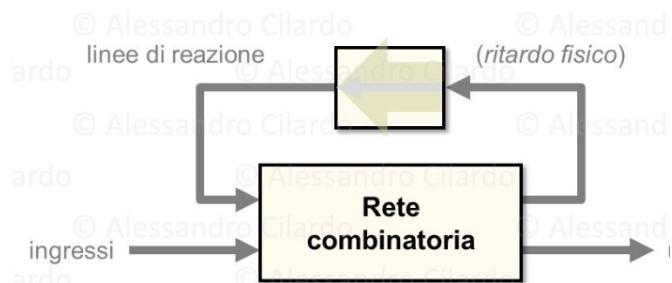


Sulla base della tipologia di elemento di ritardo utilizzato è possibile **tracciare la seguente classificazione**:

- **Reti asincrone**

In questa tipologia di macchine **l'elemento di ritardo è trasparente al suo ingresso**, che ricompare in uscita semplicemente traslato in avanti nel tempo. L'elemento può essere **realizzato anche come semplice collegamento diretto**, il che aggiunge un **ritardo fisico di propagazione** tra stato prossimo e stato corrente; di conseguenza, **non occorrono né latch né flip-flop**.

Lo **stato prossimo** si ripresenta **direttamente in ingresso** alla rete combinatoria **attraverso la linea di reazione**; la sua **propagazione** lungo la linea non è interrotta ma semplicemente **ritardata**.



Questo tipo di rete porta con sé un **problema di tempificazione**: **il nuovo stato**, propagato alla macchina combinatoria, **può causare una nuova transizione di stato mentre l'ingresso precedente è ancora applicato**. Il corretto funzionamento della macchina asincrona **impone quindi il controllo preciso della durata degli ingressi** ma, normalmente, non è possibile controllare con precisione la durata dei segnali nella realizzazione fisica dei circuiti. Questo vincolo si può rilassare (rendendo fisicamente realizzabile la macchina asincrona) se la macchina a stati finiti rispetta alcune proprietà. La problematica del progetto di reti asincrone non verrà approfondita in questa sede.

Una delle **condizioni necessarie per realizzare una macchina asincrona** impone alla tabella la tendenza verso uno **stato stabile per qualsiasi ingresso applicato costantemente**. In genere, uno stato q è stabile sotto un ingresso i se, applicando l'ingresso i nello stato q , la macchina permane in maniera indefinita in q :

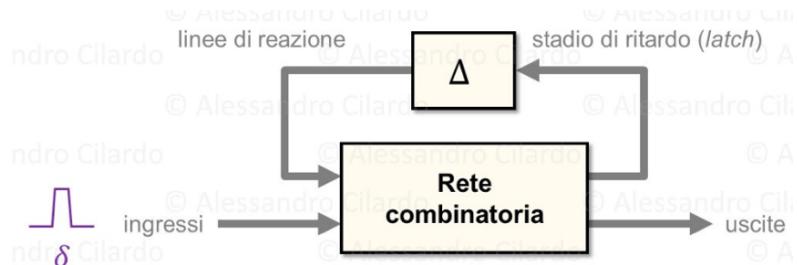
	i_1	i_2	i_3
q_1	q_1	q_2	q_3
q_2	q_4	q_2	q_3
q_3	q_1	q_4	q_3
q_4	q_4	q_4	q_3

- **Reti sincrone**

Un progetto di rete asincrona deve tenere in conto numerosi fattori ed è spesso estremamente complesso; pertanto, **nella maggior parte dei casi si ricorre ad un progetto di reti sincrone**. In questo tipo di rete **il ritardo è realizzato attraverso un elemento di memoria** che interrompe la propagazione dello stato prossimo lungo la linea di reazione: **l'istante in cui avviene la transizione di stato può essere controllato attraverso i segnali che pilotano gli elementi di memoria**, permettendo di risolvere buona parte dei problemi di tempificazione.

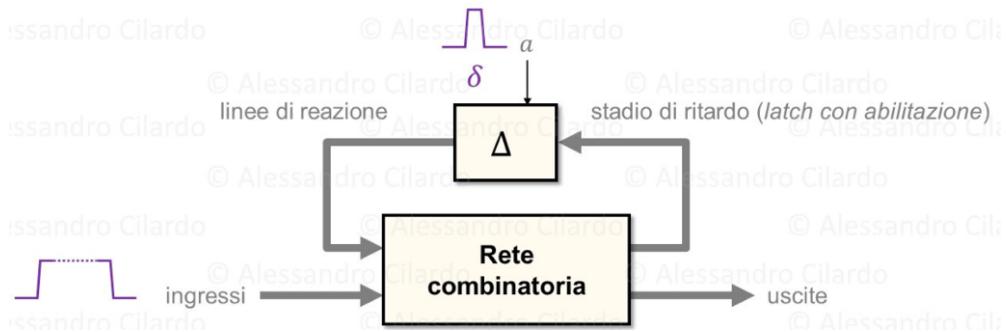
- Ad ingressi impulsivi (autosincronizzate)

In questo tipo di reti sincrone l'elemento di memoria è tipicamente un **latch**, gli ingressi sono **impulsivi e determinano l'istante di transizione di stato**, assumendo la configurazione che innesca il cambiamento di stato solo per il breve tempo δ richiesto dalla transizione. In corrispondenza dell'ingresso impulsivo, la rete combinatoria genera gli opportuni segnali (ad esempio R e S) necessari per pilotare gli elementi di ritardo.

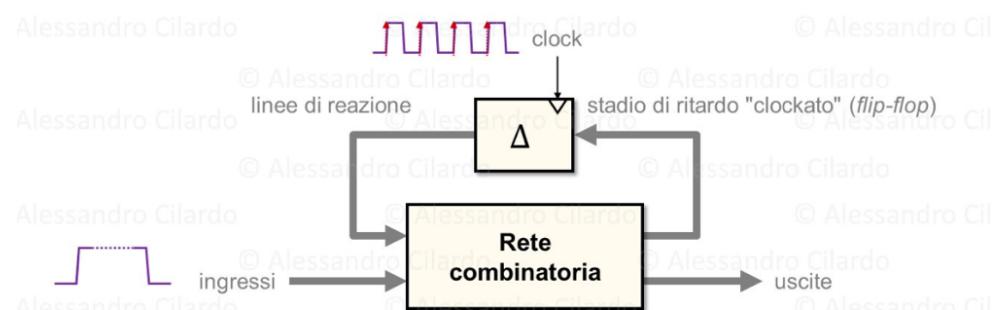


- A sincronizzazione esterna

Le reti a sincronizzazione esterna possono usare **latch con abilitazione a** , in cui i momenti di transizione di stato sono dettati da impulsi su tale segnale, mentre gli ingressi sono a livelli, ovvero tenuti stabili in corrispondenza degli impulsi. Tra un impulso ed un altro, la rete combinatoria può anche produrre uscite temporaneamente non corrette, queste però non avranno effetto sui latch.



Più spesso, per questo tipo di rete, sono usati **flip-flop edge-triggered**: i momenti di transizione di stato sono dettati dai fronti del segnale di clock e gli ingressi sono sempre a livelli. Tra un fronte e l'altro, la rete può ancora produrre uscite temporaneamente non corrette, ma non avranno effetto sui flip-flop.

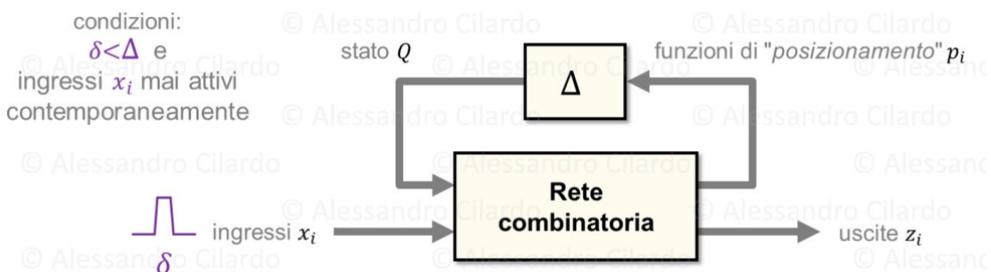


L'unico vincolo in un progetto di reti sincrone è che la durata δ dell'impulso che abilita la transizione sia inferiore al ritardo Δ con cui gli elementi di memoria registrano il nuovo stato; nel caso di reti autosincronizzate δ è la durata dell'impulso applicato sugli ingressi (quindi occorre ancora controllare la durata dei segnali, sebbene in questo caso sia più facile garantire il

rispetto del vincolo), mentre nel caso di reti a sincronizzazione esterna l'impulso corrisponde ad un fronte di clock, quindi δ è pressoché nullo.

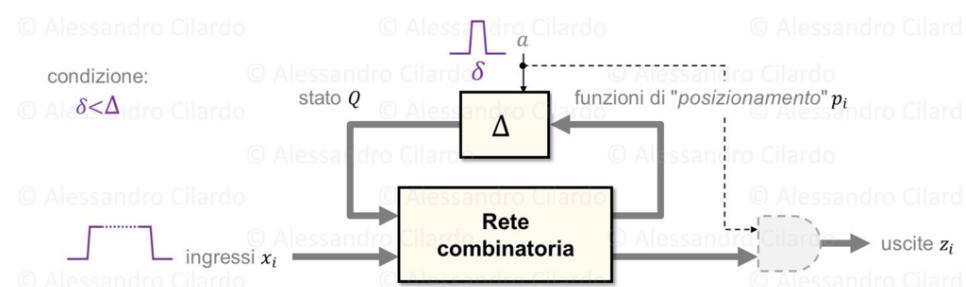
In un progetto di reti autosincronizzate, assumendo gli ingressi impulsivi di durata δ attivi alti, il vincolo è rappresentato dal fatto che gli ingressi x_i non sono mai attivi contemporaneamente; gli stati Q sono a “livelli”, cioè stabili tra un impulso e un altro, e le funzioni di posizionamento p_i sono impulsive, ottenute come:

$$p_i = \sum x_i f_{ij}(Q)$$

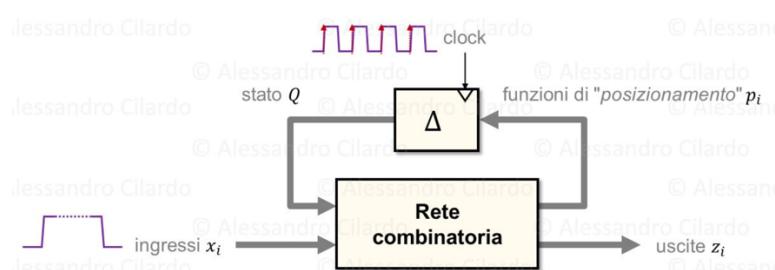


Le uscite possono essere a **livelli** (se funzione solo dello stato), in tal caso $z_i = z_i(Q)$, oppure **impulsive** (funzione di stato e ingressi), dove $z_i = \sum x_i z_{ij}(Q)$.

In un progetto di reti a sincronizzazione esterna con latch abilitati, gli ingressi, gli stati e le funzioni di posizionamento sono tutti a livelli, quindi stabili tra un impulso e l'altro del segnale di abilitazione. Le uscite possono essere: a **livelli** (funzione di stato e ingressi), in tal caso $z_i = z_i(X, Q)$, oppure **impulsive** (mascherate dall'abilitazione), dove $z_i = a \cdot z_i(X, Q)$.



In un progetto di reti a sincronizzazione esterna con flip-flop, gli ingressi, gli stati e le funzioni di posizionamento sono tutti a livelli, quindi stabili tra un fronte e l'altro del clock, mentre le uscite sono solo a livelli (funzione di stato e ingressi), $z_i = z_i(X, Q)$. Quando gli ingressi x_i sono costanti, le uscite cambiano dopo un certo tempo ϵ dal fronte e restano stabili fino al successivo fronte (che, un ϵ dopo, ne causerà l'eventuale nuova variazione); se gli ingressi x_i sono a loro volta generati da una rete sincrona con lo stesso clock, si può assumere che ingressi x_i e uscite z_i siano stabili tra un momento “fronte del clock + ϵ ” ed il successivo.



In generale, per un **progetto di reti sincrone**, si applica la seguente procedura:

1. **Ricavare il diagramma e la tabella di transizione di stato** a partire dalla descrizione del problema;
2. **Minimizzare gli stati** (potrebbero essere presenti stati dal comportamento equivalente);
3. **Codificare gli stati**;
4. **Ricavare le equazioni di ingresso** (posizionamento) dei latch o dei flip-flop a partire da stato corrente e ingressi (dipende dagli elementi usati, si possono usare tabelle di eccitazione);
5. **Ricavare le equazioni di uscita**;
6. Applicare i procedimenti di **minimizzazione di una rete combinatoria** per semplificare le espressioni di funzioni di posizionamento ed uscite

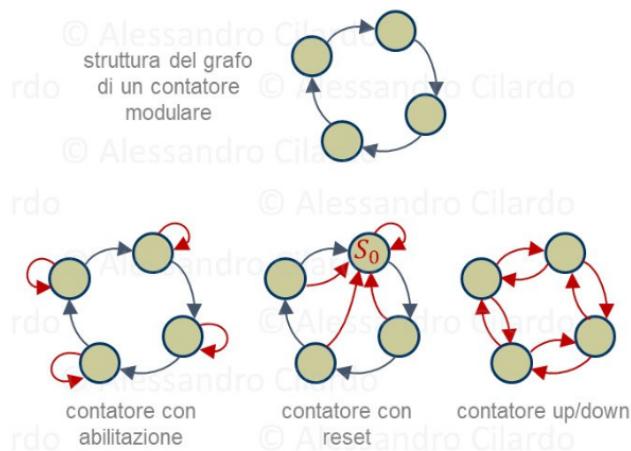
Successivamente **si distinguono i due casi**:

1. **Reti a sincronizzazione esterna**, in cui esiste un unico “ingresso impulsivo”, la sequenza dei fronti del clock, che è implicito nella definizione della macchina e agisce solo sugli elementi con memoria e sulle uscite (al più);
2. **Reti autosincronizzate**, in cui gli ingressi sono tutti impulsivi e si assume che non siano mai attivi contemporaneamente, inoltre si può sfruttare la relazione $p_i = \sum x_i f_{ij}(Q)$ potendo quindi definire e minimizzare le f_{ij} separatamente.

Le **reti sincrone**, nella pratica, **possono essere molto complesse** (ad esempio la CU). In questa sede si considerano solo **alcune tipologie di macchina rappresentabile su pochi stati**:

- **Contatori**

Hanno un **grafo di transizione di stato** lineare, spesso **ciclico**: ad ogni passo la rete **transita allo stato successivo lungo la sequenza**. Possono essere implementati anche degli **ingressi aggiuntivi**: **reset** (forza il passaggio allo stato iniziale), **abilitazione** (ferma o consente i cambiamenti di stato) e **up/down** (che determina il passaggio al successivo o al precedente stato); le **uscite** sono un singolo **segnale di fine conteggio** o il **valore contato codificato** (in questo caso l'uscita coincide con lo stato).



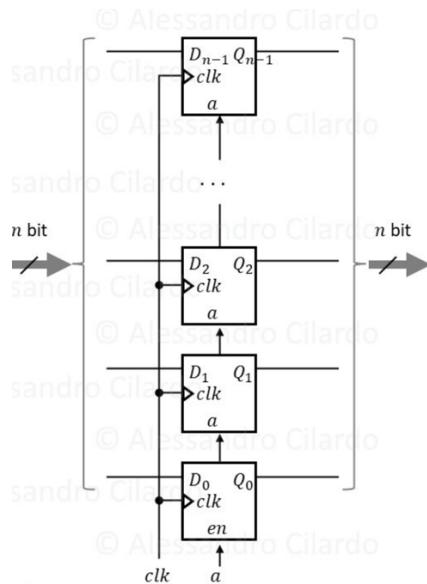
- **Riconoscitori di sequenza**

Attivano un’uscita quando gli ultimi n valori che si sono presentati in ingresso sono pari ad una prefissata sequenza di valori (ad esempio, l’uscita è alta quando gli ultimi tre valori sull’ingresso

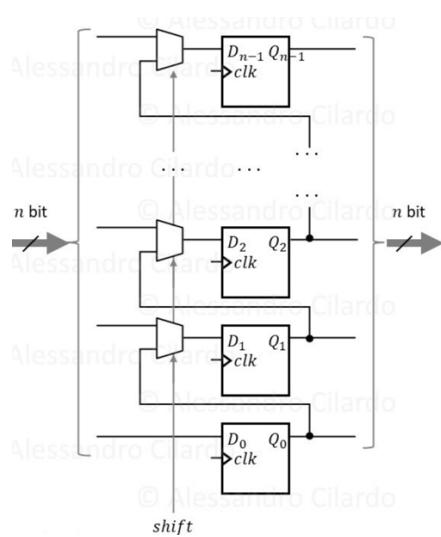
sono stati 101), con la possibilità opzionale che le **sequenza riconosciute** siano **parzialmente sovrapposte** (ad esempio l'ingresso 10101 produrrà come uscita 00101 e non 00100).

Così come nel **caso combinatorio**, anche per le **macchine sequenziali** esistono delle reti “**notevoli**” **progettate ad hoc**, come registri con vari tipi di caricamento dall'esterno, registri a scorrimento, contatori composti, contatori asincroni ecc...

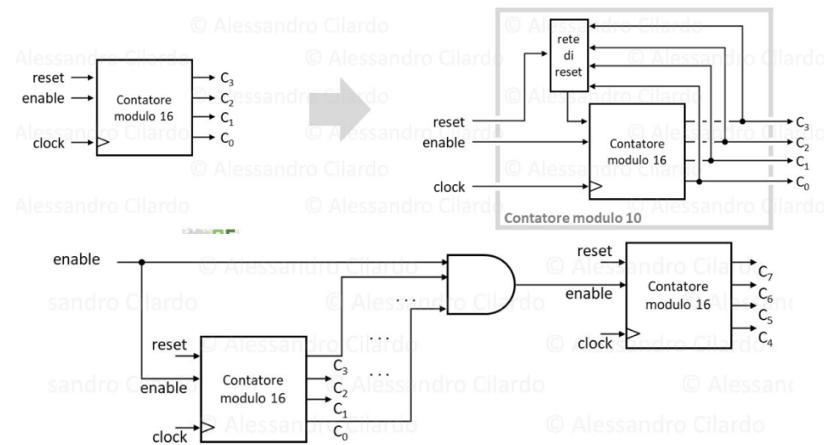
I **registri paralleli** sono uno dei vari tipi di rete sequenziale notevole, in cui un **registro a n bit** viene ottenuto come un **gruppo di n flip-flop affiancati**: spesso un segnale di **abilitazione a** permette di decidere se **caricare o meno i bit** all'interno del registro



Per quanto riguarda i **registri a scorrimento**, ingressi ed uscite dei singoli flip-flop possono essere **collegati in vario modo tra loro**: ad esempio, collegando l'uscita in posizione meno significativa all'ingresso del successivo si può ottenere l'effetto dello shifting; il caricamento può essere controllato tramite un multiplexer, in modo da ottenere diverse modalità di funzionamento (in figura è mostrato un registro a scorrimento con caricamento parallelo):



Un **contatore** può anche essere **usato come un “modulo” da comporre** con altri per ottenere un **contatore di un dato modulo** partendo da un **modulo maggiore**, oppure **contatori combinati** in un **unico contatore il cui modulo è il prodotto** dei moduli dei due componenti.



Infine, come rete sequenziale notevole, si trova il **contatore asincrono**, uno schema in cui **alcuni elementi di memoria hanno l'ingresso clock a sua volta controllato da uscite di altri elementi**, come un contatore asincrono a cascata (o ripple counter): **ogni due transizioni del flip-flop in posizione $i - 1$ si ha un fronte di discesa, e quindi una transizione, per il flip-flop in posizione i** . Sono dispositivi di **semplice realizzazione**, in cui vengono attivate solo le parti del circuito interessate alle transizioni e in cui è garantito un **minor consumo di potenza**, tuttavia sono particolarmente **lenti per molti bit** e sono **più critici da progettare** a causa dei ritardi e del clock skew.

STATI E MACCHINE EQUIVALENTI

Intuitivamente, **due macchine hanno lo “stesso comportamento”** se reagiscono allo stesso modo, ovvero con le stesse uscite $y(t)$ alle stesse sequenze di ingressi $x(t)$ per qualsiasi sequenza di ingressi applicata; inoltre, si parla di **“stati equivalenti”** facendo riferimento a **due stati q e q' di macchine diverse (o anche della stessa macchina) che producono la stessa sequenza di uscite per qualsiasi sequenza di ingressi $x(t)$ applicata**.

L'**equivalenza** può essere **definita anche in maniera ricorsiva**, infatti si dice che due stati sono equivalenti se:

1. **Sono uguali le uscite** associate ai due stati per ciascun possibile valore di ingresso applicabile;
2. **Sono equivalenti** tutte le possibili coppie dei loro **stati successivi**.

Questa definizione permette più accuratamente il riconoscimento di due stati equivalenti ed è fondamentale negli algoritmi di minimizzazione degli stati. Generalizzando il concetto, si dice che **due macchine M e M' sono equivalenti se per ciascuno stato q di M esiste almeno uno stato q' di M' ad esso equivalente e viceversa**.

Ovviamente, tra due macchine equivalenti si sceglierà quella con il **minor numero di stati**; questa affermazione introduce il **problema di minimizzazione**, cioè il problema relativo al **cercare la macchina con il minor numero di stati** tra tutte le possibili macchine equivalenti. Formalizzando il problema, **partendo da una macchina $M(Q, I, U, \tau, \omega)$, si vuole trovare una macchina $M'(Q', I', U', \tau', \omega')$ ad essa equivalente ma con un minor numero di stati**.

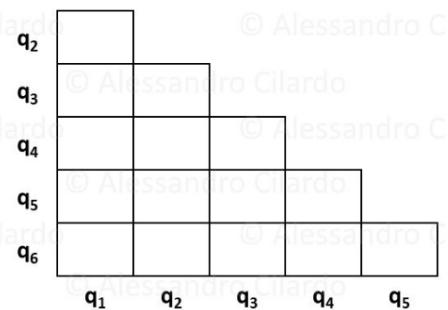
Considerando la **partizione indotta dalla relazione di equivalenza tra stati $F = (S_1, S_2, \dots, S_n)$** , dove $S_i = \{x', x'', \dots\}$ raccoglie tutti gli stati tra loro equivalenti, si può mostrare che

$M'(Q', I', U', \tau', \omega')$, definita sull'insieme di stati F , è equivalente alla macchina iniziale $M(Q, I, U, \tau, \omega)$ e ha il numero di stati minimo tra tutte le macchine equivalenti ad essa. Infatti, tutti gli elementi S_i di F sono disgiunti, uniti coprono l'insieme degli stati Q e tutti gli stati di un insieme S_i portano alla stessa uscita (in quanto equivalenti); infine, F è chiusa, da due stati di uno stesso S_i si arriva a due stati inclusi nello stesso S'_i .

Per la **minimizzazione**, si può ricorrere ad un **algoritmo di partizionamento**: vengono individuati gli stati incompatibili rispetto alle uscite per ciascun ingresso, le partizioni rilevate si esaminano rispetto allo stato prossimo e si itera finché esse non verificano la definizione di equivalenza; il meccanismo procede per “eliminazione”, infatti si parte da una presunta F (che inizialmente coincide con l'insieme di tutti gli stati Q) per cercare di individuare incompatibilità fin quando è possibile.

Uno di questi algoritmi è il **metodo tabellare di Paull-Unger**, che ricalca quanto appena descritto ma lo traspone in forma tabellare. La **griglia** su cui si basa questo metodo è usata per **confrontare ogni possibile coppia di stati**:

	i ₁	i ₂
q ₁	q ₂ / y ₁	q ₄ / y ₂
q ₂	q ₃ / y ₂	q ₅ / y ₁
q ₃	q ₅ / y ₁	q ₄ / y ₂
q ₄	q ₂ / y ₂	q ₂ / y ₂
q ₅	q ₂ / y ₂	q ₅ / y ₁
q ₆	q ₃ / y ₁	q ₅ / y ₂



Per procedere, si marciano come incompatibili le coppie di stati che portano ad uscite differenti per almeno un ingresso (violano la proprietà ricorsiva degli stati equivalenti): se la coppia di stati produce la stessa uscita per ogni valore dell'ingresso, essi sono potenzialmente equivalenti e si riportano nella corrispondente cella le coppie di differenti stati prossimi prodotti per ciascun ingresso. Terminato lo scorrimento della griglia, si riparte dall'inizio, verificando per ciascuna cella rimasta non marcata se le corrispondenti coppie di stati prossimi sono incompatibili (determinando l'incompatibilità della cella stessa); si termina quando non è più possibile marcare altre celle, da quelle non marcate è possibile derivare le classi di equivalenza.

La **disposizione a griglia triangolare** è dovuta al fatto che **ogni stato è confrontato con tutti gli altri solo una volta**:

$$q_1 \rightarrow q_2 \dots q_6$$

$$q_2 \rightarrow q_3 \dots q_6$$

Nel caso in cui per due stati siano uguali tutte le possibili uscite e tutti i possibili stati prossimi, l'**equivalenza è sempre garantita** e i due stati sono di fatto un unico stato (possono essere accoppiati direttamente (**row-merging**) eliminando uno dei due e sostituendo le ricorrenze):

	i ₁	i ₂
q ₁	q ₂ / y ₂	q ₅ / y ₁
q ₂	q ₂ / y ₂	q ₅ / y ₁
...
q ₄	q ₆ / y ₂	q ₂ / y ₂



	i ₁	i ₂
q ₁	q ₁ / y ₂	q ₅ / y ₁
...
q ₄	q ₆ / y ₂	q ₁ / y ₂

