Progettazione di Sistemi Digitali

riassunto a cura di
Matteo Petrillo

Università degli Studi di Roma "La Sapienza"

Appunti presi dalle lezioni della prof. Massini e del prof. Gorla

Indice

1 Sistemi Numerici

1.1 Codifica e Decodifica

1.1.1 Definizione e proprietà dei sistemi di codifica

I calcolatori elettronici sono macchine in grado di elaborare informazioni trasformandole in altre informazioni.

Le informazioni devono essere codificate attraverso un *codice C* e composto da simboli appartenenti ad un opportuno *alfabeto S* (alfabeto di supporto di C).

1.1.2 Codifica

La codifica di un insieme di informazioni I in un dato codice C è una funzione:

$$f : I \to C$$

1.1.3 Decodifica

La decodifica, tipicamente funzione inversa della codifica, è una funzione che mappa da un dato codice C, le informazioni I corrispondenti:

$$f : C \to I$$

1.2 Sistemi di numerazione posizionali

Un sistema di codifica per essere funzionale deve rispettare dei criteri:

- **Economico**: avere pochi simboli è sicuramente un vantaggio.
- **Efficienza**: un linguaggio semplice da codificare e decodificare porterà vantaggi in termini di efficienza.
- **Semplicità di Elaborazione**: poter svolgere operazioni in maniera semplice e agevole.

Si userà quindi un sistema di numerazione posizionale (tipo quello arabo a cui siamo abituati) determinato da una base ***b***.

Il sistema decimale che utilizziamo quotidianamente non è nient altro che un sistema posizionale in **base 10**.

Possiamo quindi spiegare un sistema posizionale mediante la seguente formula:

$$N = \sum_{i=0}^{m-1} c_i \cdot b^i; \, c = \textit{coeff. moltiplicativo e } b = \textit{base} \tag{1}$$

Quindi una diversa sequenza di simboli interpretata in basi diverse rappresenta un diverso numero naturale. Ad esempio 10 in **base 2**:

$$10_2 = 1 \cdot 2^1 + 0 \cdot 2^0 = 2$$

Questa diversa interpretazione avviene in altri codici più comuni, ad esempio la parola *case* , in italiano è il plurale di casa mentre nel codice di lingua inglese vuol dire *valigia*, bisogna quindi capire bene in quale codice si sta lavorando per poter decodificare le informazioni nella maniera corretta.

In ambito informatico si userà il **codice binario**.

1.3 Il Sistema Binario

Il codice binario è un codice costituito da soli due simboli: 0 e 1, quindi si lavorerà in *base 2*.

I simboli 0 ed 1 prendono il nome di *bit (binary digit)*. Ci sono due principali motivi per cui si lavora in binario:

- Da un punto di vista teorico, Boole dimostrò che tutta la logica può essere espressa in binario
- Il codice binario fu trovato molto utile nella teoria della commutazione per descrivere il comportamento dei circuiti digiatli (1 = acceso e 0 = spento), oppure per indicare se un relay meccanico sia aperto o chiuso.

Ogni tipo di informazione può essere codificata in binario.

Adattando quindi la formula 1 alla base 2 del sistema binario otterremo:

$$N_2 = \sum_{i=0}^{n-1} c_i \cdot 2^i \tag{2}$$

In una stringa binaria, il bit c_0 viene detto **LSB** (Less Signifying Bit), ossia il bit meno significativo (quello più a destra) poiché elevato alla potenza più basse, mentre il bit c_{n-1} viene detto **MSB** (Most Signifying Bit) ossia il bit più significativo (quello più a sinistra) poiché elevato alla potenza più grande.

2 I Numeri Naturali

2.1 Metodi di Conversione

2.1.1 Metodo Polinomiale

Il metodo polinomiale è efficiente nel trasformare un numero **da base x** a base 10 (decimale)

Esempio:

$$111001_2 = 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

Tuttavia è estremamente più complesso convertire un numero N da una base x ad una base y.

2.1.2 Metodo delle Divisioni Iterate

Il metodo delle divisioni iterate è ottimo per la conversione **da base 10** a base x. Possiamo distinguere 2 casistiche principali:

1. La base di arrivo è potenza della base di partenza ($b = a^k$)
2. La base di arrivo non è potenza della base di partenza ($b \neq a^k$)

Caso 1

L'algoritmo di risoluzione è il seguente:

1. Convertire la base b in k-ple di cifre, dalla meno alla più significativa di N_a
2. Convertire i gruppi di cifre ottenute nella base di arrivo

Esempio:

Convertire 101001101101 da base **2** a base **8**

Osservo che $8 = 2^{\mathbf{3}}$ quindi dovrò suddividere la sequenza binaria in gruppi da **3** cifre. Successivamente interpreto i numeri nella base di partenza e li converto nella base di arrivo.

Quindi:

$$(101\ 001\ 101\ 101)_2 \rightarrow (5\ 1\ 5\ 5)_8$$

NB: Se l'ultimo gruppo non rispetta la grandezza necessaria (quello più a sinistra), bisognerà solamente riempirlo di 0 affinché rispetti tale grandezza.

Caso 2

In questo caso l'algoritmo di risoluzione è il seguente:

1. Calcolare iterativamente il resto ottenuto dalla divisione del numero da convertire e la base di arrivo
2. La sequenza di cifre del resto ottenute (letta al contrario) è il numero nella nuova base

Esempio:

Convertire 26_{10} in base 2

$$\begin{aligned} 26/2 &= 13 \textit{(resto } \boldsymbol{0}\textit{)};\\ 13/2 &= 6 \textit{(resto } \boldsymbol{1}\textit{)};\\ 6/2 &= 3 \textit{(resto } \boldsymbol{0}\textit{)};\\ 3/2 &= 1 \textit{(resto } \boldsymbol{1}\textit{)};\\ 1/2 &= 0 \textit{(resto } \boldsymbol{1}\textit{)}; \end{aligned}$$

Rileggendo quindi i resti delle divisioni in ordine inverso abbiamo che:

$$26_{10} = 11010_2$$

Infatti

$$11010_2 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 16 + 8 + 2 = 26_{10}$$

2.2 Intervallo di Rappresentabilità

Prima di tutto bisogna soffermarsi su un punto:

Quante diverse sequenze posso rappresentare con n bit?

Si possono rappresentare 2^n sequenze di bit, dal valore 0 a $2^n - 1$, quindi ad esempio con 4 bit posso rappresentare 16 sequenze, da 0 a 15.

Quindi:

$$\sum_{i=0}^{n-1} 2^i \tag{3}$$

Invece, al contrario, per capire quanti bit mi servono per rappresentare un numero naturale N bisognerà usare l'inverso dell'esponenziale, ossia il logaritmo $\log_b k$ dove $b = 2$.

Quindi per rappresentare il numero k dovrò:

1. Calcolare $\log_2 k$
2. Prendere il numero intero e aggiungere 1 (essendo il risultato un numero irrazionale mi servirà del margine per assicurarmi la rappresentazione)

Esempio:

Quanti bit mi servono per rappresentare 57_{10}?

1. $\log_2 57 = 5,832\ldots$
2. Mi serviranno quindi 5+1 = 6 bit

Infatti con 6 bit potrò rappresentare l'intervallo [0;63].

Per semplificare il conto del logaritmo bisogna trovare la potenza di 2 più vicina al numero da voler rappresentare. (Nel caso precedente è $2^5 = 32$, di conseguenza 5+1 = 6)

Questo ragionamento è importante poiché nel momento in cui vogliamo fare un'operazione bisogna capire su quanti bit bisogna lavorare per rappresentare correttamente gli operandi e il risultato.

2.2.1 Overflow

Per semplicità gli elaboratori lavorano su parole di codice di lunghezza predeterminata (in potenze da 2 bit):

- 8 bit = byte
- 16 bit = half-word
- 32 bit = word
- 64 bit = long-word

Se la parola di codice è più corta della lunghezza decisa allora verrà riempito lo spazio in eccesso con una sequenza di 0, invece al contrario la parola (o il numero) non è rappresentabile per mancanza di informazioni e si cadrebbe nell'**overflow**.

3 Operazioni in base 2

3.1 Somma e Differenza

La somma e differenza in binario vengono eseguite esattamente come nel sistema decimale, per semplicità le si effettueranno in colonna.

Bisognerà prestare attenzione nel caso di prestiti o riporti di leggere i numeri in base 2 per non commettere errori.

Esempio 1:

Effettuare $4_{10} + 7_{10}$

1. Calcolo i bit necessari per rappresentare operandi e risultati:
   per rappresentare 4 mi occorrono 3 bit
   per rappresentare 7 mi occorrono 3 bit
   per rappresentare 11 mi occorrono 4 bit
   quindi dovrò lavorare con 4 bit per rappresentare correttamente la mia operazione

2. Converto i numeri da base 10 a base 2:
   $4_{10} = 0100_2$
   $7_{10} = 0111_2$

3. Effettuo l'operazione:

$$\begin{array}{r} 0100 + \\ 0111 = \\ \hline 1011 \end{array}$$

   Quando effettuo 1+0, il risultato è ovviamente 1, quando effettuo però 1+1, il risultato sarà 2, che in binario è 10, quindi scriverò 0 con riporto di 1.

Esempio 2:

Effettuare $12_{10} - 3_{10}$

1. Calcolo i bit necessari per rappresentare operandi e risultati:
   per rappresentare 12 mi occorrono 4 bit
   per rappresentare 3 mi occorrono 2 bit
   per rappresentare 9 mi occorrono 4 bit
   quindi dovrò lavorare con 4 bit per rappresentare correttamente la mia operazione

2. Converto i numeri da base 10 a base 2:
   $12_{10} = 1100_2$
   $3_{10} = 0011_2$

3. Effettuo l'operazione:

$$\begin{array}{r} 1100\ - \\ 0011\ = \\ \hline 1001 \end{array}$$

In questo caso non posso effettuare 0-1 quindi dovrò chiedere in prestito un'unità dalla cifra più a sinistra, da ricordarsi che l'unità in questione è in base 2.

3.2 Moltiplicazione e Divisione

La moltiplicazione è esattamente come per le operazioni in decimale, quindi:

1. Moltiplicare ogni cifra del secondo fattore per il primo fattore facendo shiftare a sinistra di una posizione ogni volta che si cambia cifra
2. Effettuare l'addizione tra tutte le sequenze calcolate

Esempio:

Effettuare $3_{10} \times 4_{10}$

I bit necessari per rappresentare una moltiplicazione a n-bit sono 2n bit. In questo caso i fattori posso rappresentarli con 3 bit, ma per rappresentare il risultato dovrò usare 6 bit.

$$\begin{array}{r} \times 0011 \\ = 0100 \\ \hline 0000 \\ 0000\text{-} \\ 0011\text{–} \\ 0000\text{—} \\ \hline 001100 \end{array}$$

La divisione non verrà approfondita per motivi pratici.

4 I Numeri Interi

Affrontando ora i numeri interi dovremo preoccuparci di rappresentare il segno oltre al valore, per fare ciò ci sono 3 possibili strade:

1. Modulo e Segno
2. Complemento a 1
3. Complemento a 2 *(o complemento alla base in generale)*

4.1 Modulo e Segno

In questa rappresentazione si userà l'MSB per rappresentare il segno ($0 \to +$ e $1 \to -$) e i rimanenti bit rappresenteranno il valore.

Esempio. $110_2 = -2_{10}$

4.2 Complemento a 1

In questa rappresentazione se l'MSB = 1 allora saprò che il numero rappresentato è negativo e per rappresentare il valore dovrò fare il complemento dei restanti bit (quindi switchare lo 0 con l'1 e viceversa).

Esempio. $110_2 = (-)01_2 = -1$

4.3 Complemento a 2

Infine abbiamo la rappresentazione più utilizzata, quella del complemento a 2.

In questo caso l'MSB è sempre associato al segno ($0 \to +$ e $1 \to -$) ma verrà calcolata la potenza associata alla posizione (negativa in caso di 1).

Quindi:

$$-c_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-1} c_i \cdot 2^i \tag{4}$$

Esempio 1. $010_{ca2} = -0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = -0 + 2 + 0 = +2_{10}$

Esempio 2. $101_{ca2} = -1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = -4 + 0 + 1 = -3_{10}$

Ecco quindi le rappresentazioni possibili con 2 bit secondo le tre strade descritte:

Num	MS	CA1	CA2
00	+0	+1	0
01	+1	+0	+1
10	-0	-1	-2
11	-1	-0	-1

Grazie alla tabella si può vedere come MS e CA1 abbiano due rappresentazioni per lo 0, infatti il loro intervallo di rappresentazione *(con n-bit)* sarà $[-2^{n-1}+1; 2^{n-1}-1]$, invece nel CA2 avrò un intervallo asimmetrico poiché lo zero ha una sola rappresentazione e di conseguenza per un valore non ci sarà il suo opposto (*es.* $\in -2; \notin +2$).

Quindi in CA2 l'intervallo di rappresentazione sarà $[-2^{n-1}; 2^{n-1}-1]$.

Esempio: con 4 bit, in CA2 potrò rappresentare [-8;7]

4.3.1 Operazioni Aritmetiche in CA2

La somma e la moltiplicazione sono come in binario, bisogna però dare il giusto significato al risultato.

Nel caso della sottrazione invece eseguirò l'addizione tra un operando e l'opposto dell'altro, quindi:

$$A - B = A + (-B)$$

Esempio

$$\text{Eseguire } -3_{10} + 2_{10}$$

Farò i seguenti passaggi:

1. Calcolo il numero di bit necessario a rappresentare gli operandi ed il risultato: 3 bit
2. Converto i numeri rappresentandoli con i bit necessari:
   $-3_{10} = 101_{ca2}$
   $+2_{10} = 010_{ca2}$
3. Eseguo l'operazione e interpreto il risultato in CA2:

$$\begin{array}{r} 101 + \\ 010 = \\ \hline 111 \end{array}$$

$$111_{ca2} = -4 + 3 = -1_{10}$$

In CA2 avrò un caso di overflow se sommando due positivi ottengo un numero negativo.

5 I Numeri Reali

Considerato il numero limitato di bit, ovviamente potremo rappresentare un numero limitato di numeri reali e non con assoluta precisione.

5.1 Conversione in base 2

Approccerò la conversione di un numero reale in base 2 separatamente per la parte intera e la parte decimale.

- Per la parte intera userò sempre il metodo delle divisioni iterate
- Per la parte decimale invece userò le **moltiplicazioni iterate**

5.1.1 Moltiplicazioni Iterate

L'algoritmo è il seguente:

1. Considero solo la parte decimale del numero da convertire *(sarà quindi 0,...)*
2. Moltiplico quel valore $\times 2$ finché non ottengo un numero ¿1
3. Decido se fermarmi o continuare ad applicare l'algoritmo prendendo in considerazione sempre e solo la parte decimale
4. Una volta finito (poiché mi è venuto un numero intero o poiché limitato dai bit) prenderò in considerazione la serie di 0 e 1 della parte intera in ordine di calcolo

Esempio

Convertire $0,125_{10}$ in base 2

$$0,125 \times 2 = \mathbf{0},25$$
$$0,25 \times 2 = \mathbf{0},5$$
$$0,5 \times 2 = \mathbf{1},0$$

Quindi $0,125_{10} = 0,001_2$

Ovviamente se il numero è irrazionale e quindi non verrà mai un numero intero, la sua precisione è relativa al numero di bit utilizzati per rappresentarlo.

5.2 Rappresentazione dei Numeri con la virgola

Ci sono due modi per poter rappresentare in binario i numeri con la virgola: virgola fissa e virgola mobile.

5.2.1 Virgola Fissa

In virgola fissa si decide a priori quanti bit destinare alla parte intera e alla parte decimale, tuttavia questa rappresentazione è limitata e insufficiente in alcuni contesti (basti pensare alle misure astronomiche o microscopiche).

5.2.2 Virgola Mobile

Il metodo più comune è quello della rappresentazione in virgola mobile, che funziona analogalmente alla annotazione scientifica.

Ad esempio l'unità astronomica è uguale a $1,495978707 \cdot 10^{11}$ *metri.*

Si dedicherà quindi 1 bit al segno, alcuni bit all'esponente e alcuni bit alla mantissa (in cui bisognerà decidere se rappresentarla come 1,... o 0,...).

5.3 Standard IEEE-754

Lo standard di rappresentazione dei numeri in virgola mobile IEEE-754 definisce come vengono suddivisi i bit tra segno/esponente/mantissa, avremo quindi i 3 seguenti casi:

1. 16 bit (half-precision):
   - 1 bit x segno
   - 5 bit x esponente
   - 10 bit x mantissa
2. 32 bit (single precision):
   - 1 bit x segno
   - 8 bit x esponente
   - 23 bit x mantissa
3. 64 bit (double precision):
   - 1 bit x segno
   - 11 bit x esponente
   - 52 bit x mantissa

Secondo lo standard la mantissa sarà rappresentata secondo la notazione $1, ... \times 2^n$ poiché comodo in base 2 avere un qualsiasi numero diverso da 0 prima della virgola (e quindi per forza 1).

Un numero in virgola mobile secondo lo standard IEEE-754 verrà scritto nella seguente maniera:

$$< \text{segno} ; \text{esponente} ; \text{mantissa} >$$

La condizione di **overflow** avviene quando non si potrà rappresentare l'esponente (esponente troppo grande per i bit assegnati secondo la rappresentazione scelta).

5.3.1 Segno

Il segno verrà rappresentato da 1 bit: 0 se positivo e 1 se negativo.

5.3.2 Esponente

Per rappresentare l'esponente secondo lo standard bisognerà effettuare i seguenti passaggi (*n corrisponde al numero di bit dedicati all'esponente*):

1. Considerare l'esponente in CA2 e quindi il suo intervallo $[-2^{n-1}; 2^{n-1}-1]$
2. Togliere le 2 rappresentazioni più piccole, quindi l'intervallo diventerà $[-2^{n-1}+2; 2^{n-1}-1]$
3. Sommare il **bias** (dato da $2^{n-1}-1$) , quindi l'intervallo dell'esponente sarà $[1; 2^n-2]$

Ad esempio in half-precision (16 bit) dedichiamo 5 bit all'esponente, quindi il suo intervallo di rappresentazione sarà $[1; 30]$

In questo modo verranno eslusi dall'intervallo solamente lo 0 e il 31, due numeri facili da rappresentare (00000 e 11111).

5.3.3 Mantissa

La mantissa come detto è rappresentata secondo la notazione $1,... \times 2^n$, in questa maniera considerando implicito 1,... prima della parte decimale si potrà memorizzare un bit in più in coda.

Bisognerà però ricordarsi dell'1 implicito nel momento in cui bisognerà shiftare la mantissa ad esempio a seguito di un cambio di esponente.

Esempio

Convertire $-14,25_{10}$ in base 2 secondo lo standard IEEE-754

Eseguirò i seguenti passaggi:

1. Converto la parte intera in base 2 con le divisioni iterate:
   $14_{10} = 1110_2$
2. Converto la parte decimale in base 2 con le moltiplicazioni iterate:
   $0,25_{10} = 0,01_2$

   Quindi il mio numero sarà -1110,01
3. Faccio scorrere l'esponente di 3 posizioni verso sinistra così da portare la mantissa secondo lo standard:
   $-1110,01 = -1,11001 \times 2^3$
4. Sommo il bias (15 essendo in half-precision) all'esponente e converto in binario: $3 + 15 = 18_{10} = 10010_2$
5. Controllo infine il segno per attribuire 0 o 1 al primo bit

Quindi $-14,25_{10}$ sarà rappresentato secondo lo standard IEEE-754 (half-precision):

$$< 1; 10010; 1100100000 >^{1}$$

Infine, spesso, questi numeri verranno rappresentati nel formato esadecimale o verrà richiesta la conversione, per fare ciò si prende la stringa di 16 bit senza suddivisioni e si divide in 4 gruppi da 4 bit e si convertono i singoli gruppi in esadecimale, prendendo $< 1; 10010; 1100100000 >$:

1. La stringa intera sarà 1100101100100000
2. Suddivido in gruppi da 4 bit:
   1100/1011/0010/0000
3. Converto i numeri da base 2 a base 16:
   C/B/2/0

Quindi $-14,25_{10} =< 1; 10010; 1100100000 >$=CB20

5.3.4 Tabella di Rappresentazione

Basandosi sui valori di esponente e mantissa possiamo distinguere varie tipologie di numero:

Tipo	**Esponente**	**Mantissa**
Zeri	0	0
Num. Denormalizzati < 1	0	$\neq 0$
Normali	$[1; 2^n - 2]$	qualunque
$\pm\infty$	$2^n - 1$	0
NotANumber (NAN)	$2^n - 1$	$\neq 0$

5.3.5 Operazioni Aritmetiche

Somma e Differenza

Per risolvere le operazioni di somma e differenza tra numeri in virgola mobile, bisogna seguire questo algoritmo:

1. Confrontare gli esponenti e portare il più piccolo pari a quello più grande
2. Shiftare la mantissa a destra (ricordandosi dell'1 implicito)
3. Eseguire l'operazione tra le due mantisse dopo aver stabilito l'ordine degli operandi (valutando magnitudo e segno)
4. Decidere il segno del risultato
5. **Normalizzare** il risultato:

[1] Son stati messi cinque 0 in coda per rispettare i 10 bit della mantissa, essi non alterano il valore

- Posizionare la virgola secondo lo standard 1,...
- Calcolare l'esponente finale e shiftare la mantissa
- Far combaciare i bit della mantissa con quelli dello standard (troncando i bit meno significativi o aggiungendo zeri in coda)

Moltiplicazione e Divisione

Per eseguire moltiplicazioni o divisioni tra numeri in virgola mobile si seguirà il seguente algoritmo:

1. Stabilire il segno:
   - + se i valori sono concordi
   - - se i valori sono discordi
2. Eseguire la somma o differenza tra gli esponenti *(ricordandosi che se entrambi sono già dotati di bias, bisognerà aggiungere o togliere un bias per avere l'esponente finale corretto)*
3. Eseguire l'operazione tra le mantisse
4. Normalizzare il risultato secondo lo standard

6 Codice Ascii

Per la rappresentazione di caratteri si usa la codifica ASCII, anch'esso rappresenta uno standard.

Il codice ASCII è un codice a 7 bit:

- 3 bit riservati al **prefisso**
- 4 bit riservati alla **codifica**

6.1 Prefissi

Ci sono 3 tipologie di prefissi:

1. Numeri = 001
2. Maiuscole = 100 oppure 101
3. Minuscole = 110 oppure 111

6.2 Codifica di Lettere

I restanti 4 bit vengono impiegati nella codifica di lettere e il prefisso indicherà se quel carattere è maiuscolo, minuscolo...

Ad esempio:

- A = 100**0001**
- a = 110**0001**

Normalmente si usa un numero di bit pari ad una potenza di 2, infatti l'ottavo bit si userà come **bit di parità**.

6.3 Bit di Parità

Il bit di parità consiste in un doppio controllo da il mittente e il ricevente di un carattere, infati bisognerà decidere se optare per parità pari o parità dispari:

- Parità Pari: ai 7 bit verrà aggiunto uno 0 o un 1 (in testa) in modo tale che la sequenza di 8 bit avrà un numero **pari** di 1 nel complesso.

  Esempio: A = 1000001 → 0**1**000001 *(parità pari)*

- Parità Dispari: ai 7 bit verrà aggiunto uno 0 o un 1 (in testa) in modo tale che la sequenza di 8 bit avrà un numero **dispari** di 1 nel complesso.

  Esempio: a = 1100001 → 0**11**00001 *(parità dispari)*

Quindi il circuito di trasmissione controlla tramite il bit di parità la correttezza delle informazioni ricevute.

7 Algebra Booleana

L'Algebra di Boole è definita usando:

1. **Alfabeto di Supporto:** l'alfabeto di supporto in questo caso corrisponde a $\Sigma = \{0; 1\}$
2. **Operazioni:** le operazioni dell'algebra booleana sono

   - **NOT** (Complementazione)

x	$\overline{x}$
0	1
1	0

   - **AND** (Prodotto Logico)

x y	$x \cdot y$
0 0	0
0 1	0
1 0	0
1 1	1

   - **OR** (Somma Logica)

x y	$x + y$
0 0	0
0 1	1
1 0	1
1 1	1

   - **XOR** (Exclusive-Or)

a b	$a \oplus b$
0 0	0
0 1	1
1 0	1
1 1	0

   Esempio. $a \oplus b = \overline{a}b + a\overline{b}$ oppure $(a + b)(\overline{a} + \overline{b})$

3. **Assiomi:**

   - **Associatività**:
     $(a + b) + c = a + (b + c)$
     $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
     $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
   - **Commutatività**:
     $a + b = b + a$
     $a \cdot b = b \cdot a$
   - **Distribuitività**:
     $a \cdot (b + c) = a \cdot b + a \cdot c$
     $a + b \cdot c = (a + b) \cdot (a + c)$

- **Elemento Neutro**:
  $a + 0 = a$
  $a \cdot 1 = a$
  $a \oplus 1 = \overline{a}$
- **Complemento**:
  $a + \overline{a} = 1$
  $a \cdot \overline{a} = 0$
  $\overline{a \oplus b} = ab + \overline{a}\overline{b}$

4. **Proprietà**:

   - **Elemento Nullificatore**:
     $a + 1 = 1$
     $a \cdot 0 = 0$
   - **Involuzione:**
     $\overline{\overline{a}} = a$
   - **Idempotenza:**
     $a + a = a$
     $a \cdot a = a$
   - **Assorbimento**:
     $a + ab = a$
     $a \cdot (a + b) = a$
   - **Leggi di De Morgan**:
     $\overline{a + b} = \overline{a} \cdot \overline{b}$
     $\overline{a \cdot b} = \overline{a} + \overline{b}$

Osservando queste proprietà possiamo notare una relazione di **dualità** tra somma e prodotto.

7.1 Relazione di Dualità

Data un'espressione possiamo ottenere la sua duale scambiando gli operatori e/o scambiando i simboli dell'alfabeto di supporto.

Può capitare che per facilitare l'analisi di un'espressione serva ottenere proprio la sua duale.

Quindi: se E = espressione allora $\tilde{E}$ = duale di E.

7.2 Espressione Booleana

Nelle espressioni booleane possiamo trovare:

- Simboli dell'alfabeto di supporto
- Variabili
- Operatori Logici
- Parentesi

Tutto ciò porta alla definizione di espressione.

Definizione. Un'espressione boolena si definisce:

- Elementi di Σ sono espressioni booleane
- Le variabili sono espressioni booleane
- Data E anche $\tilde{E}$ è un espressione booleana
- Date E_1 ed E_2 anche $E_1 + E_2$ oppure $E_1 \cdot E_2$ sono espressioni booleane
- Le espressioni booleane sono solo quelle che si ottengono applicando un numero finito di volte le regole precedenti

7.3 Espressione Complementare

L'espressione compelementare si trova o usando le leggi di De Morgan oppure si ottiene la duale e si compelementano le singole variabili:

Esempio. $a\overline{b} + \overline{a}c = (\overline{a} + b) \cdot (a + \overline{c})$

L'espressione complementare è utile per verificare l'identità di un'espressione.

Ovviamente essendo complementare i suoi valori di identità saranno opposti a quelli dell'espressione normale.

7.4 Forme SOP e POS

Una funzione la possiamo esprimere tramite un'espressione in forma SOP *(Sums of Products)* o POS *(Products Of Sums)*.

Ad esempio l'espressione $ac + \overline{a}b\overline{c}$ è in forma SOP, ma la stessa espressione posso trasformarla in forma POS e sarà quindi $(a + b)(a + \overline{c})(\overline{a} + c)(b + c)$.

Bisogna stare attenti che sia per la forma SOP sia per la forma POS, i termini dell'espressione devono essere singole variabili/simboli e quindi **non** contenere a loro volta altri prodotti/somme.

Vediamo ora la tavola di verità dell'espressione precedente sia in forma SOP che in forma POS:

abc	f(sop)	f(pos)
000	0	0
001	0	0
010	1	1
011	0	0
100	0	0
101	1	1
110	0	0
111	1	1

Table 1: Funzioni SOP e POS

Il metodo per trovare facilmente i valori di verità posso avvalermi delle proprietà delle espressioni booleane, in particolare andrò a controllare:

- Per la forma SOP, quando uno degli addendi vale 1, di conseguenza tutta l'espressione varrà 1 e le restanti combinazioni 0
- Per la forma POS, quando uno dei due fattori vale 0, di conseguenza tutta l'espressione varrà 0 e le restanti combinazioni daranno 1

Come da tabella, qualunque forma si scelga i valori di verità dovranno essere i medesimi.

7.4.1 Ottenere un'espressione in forma SOP o POS

Per ricavare una delle due forme si può procedere secondo il seguente algoritmo:

1. Applicare, se necessario, le leggi di De Morgan così da ottenere le eventuali complementazioni sulle singole variabili
2. Applicare la proprietà distributiva della somma sul prodotto o la sua duale
3. Eliminare i termini ripetuti grazie all'idempotenza
4. Applicare se possibile la proprietà di assorbimento

7.4.2 Forma Canonica

Sia nella SOP sia nella POS, si fa riferimento alla **forma canonica**, essa prevede che in tutti gli addendi o fattori compaiano tutte le variabili (in forma normale o negata).

Nella forma SOP, un termine-prodotto in cui compaiono tutte le variabili si chiama **mintermine**.

Nella forma POS, un termine-somma in cui compaiono tutte le variabili si chiama **maxtermine**.

Quindi possiamo affermare che:

- Una forma canonica SOP è un'espressione in cui tutti i termini prodotto sono mintermini

- Una forma canonica POS è un'espressione in cui tutti i termini somma sono maxtermini

Come ottenere la forma canonica da una forma normale Per eseguire questa trasformazione bisogna seguire i seguenti passaggi:

1. Moltiplicare/Sommare per la somma/prodotto della variabile mancante e il suo complemento
2. Applicare la proprietà distributiva corretta
3. Se presenti, eliminare i termini ripetuti

Esempio: data l'espressione SOP in forma normale: $ac + \overline{a}b\overline{c}$:

1. Moltiplico per la somma della variabile mancante con il suo complemento al primo termine: $ac \cdot (b + \overline{b}) + \overline{a}b\overline{c}$
2. Applicando la proprietà distributiva ottengo: $abc + a\overline{b}c + \overline{a}b\overline{c}$, quindi tutti i termini ora sono **mintermini**.

Analogalmente prendendo l'espressione precedente nella sua forma POS: $(a + c)(\overline{a} + b + \overline{c})$:

1. Aggiungo il prodotto della variabile mancante con il suo complemento al primo termine: $(a + c + b\overline{b})(\overline{a} + b + \overline{c})$
2. Applicando la proprietà distributiva ed eliminando eventuali termini ripetuti ottengo: $(a + c + b)(a + c + \overline{b})(\overline{a} + b + \overline{c})$.
   Ora tutti i termini sono **maxtermini**.

Possiamo ottenere le FC (Forme Canoniche) anche attraverso le tavole di verità, indicando nell'espressione il riferimento al M (maxtermine) o m (mintermine).

Esempio da tabella 1. $\mathrm{FC}_{SOP} = M_2 + M_5 + M_7 = \overline{a}b\overline{c} + a\overline{b}c + abc$

7.5 Operatori Universali

Introduciamo ora due nuovi operatori, il **NAND** e il **NOR**, essi si definiscono **operatori universali** poiché si possono ottenere tutti gli operatori di base tramite questi due e si possono realizzare espressioni booleane ALL-NAND e ALL-NOR.

7.5.1 NAND

L'operatore NAND rappresenta la negazione del prodotto logico, gli altri operatori si possono ottenere applicando le leggi di De Morgan inverse, per cui:

- $\overline{a} = \overline{a \cdot a}$
- $a \cdot b = \overline{\overline{ab}} = \overline{\overline{ab} \cdot \overline{ab}}$
- $a + b = \overline{\overline{a} \cdot \overline{b}} = \overline{\overline{a \cdot a} \cdot \overline{b \cdot b}}$

L'applicazione di questo operatore è consigliata per espressioni in forma SOP.

7.5.2 NOR

L'operatore NOR rappresenta la negazione della somma logica e possiamo ottenere tutti gli operatori di base mediante l'applicazione delle leggi di De Morgan inverse, per cui:

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

L'applicazione di questo operatore è consigliata per espressioni in forma POS.

8 Reti Combinatorie

Per ogni EB *(Espressione Booleana)* possiamo associare un Circuito Combinatorio (o Rete Combinatoria).

Essi ottengono un segnale che porta il valore di 0 o 1 basato sulla differenza di potenziale (*alta = 1 e bassa = 0)*.

Una rete combinatoria è un circuito elettronico digitale in grado di calcolare in modo automatico una **funzione booleana**.

Un circuito elettronico è un sistema costituito da blocchi elementari *(porte)* interconnesse tra loro in maniera *aciclica*.

Gli operatori booleani sono rappresentati dalle **porte logiche**.

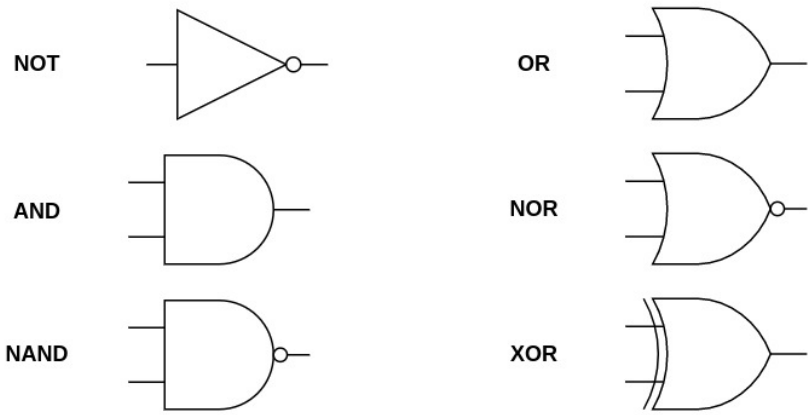


Figure 1: Porte Logiche

Esempio

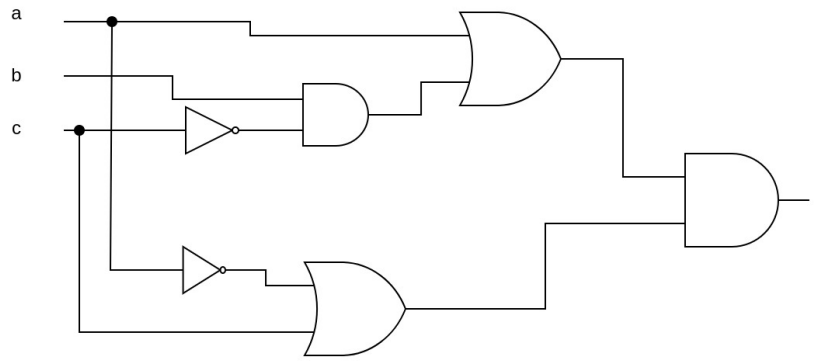


Figure 2: $(a + b\overline{c})(\overline{a} + c)$

Per semplicità di realizzazione è sempre bene semplificare il più possibile l'espressione booleana prima di trasformarla in circuito combinatorio.

Possiamo trovare due principali workflow riguardanti le reti combinatorie: analisi e sintesi.

8.1 Analisi di una Rete Combinatoria

In questo tipo di workflow ci viene fornita una rete combinatoria e lo scopo è trovare la funzione booleana calcolata.

Passaggi:

1. Dalla rete combinatoria passare all'espressione booleana ricordandosi che *per ogni RC a m uscite, esiste una m-pla di EB che la descrive.*
2. Passare dall'espressione booleana alla funzione booleana (o alla tavola di verità) mediante l'induzione perfetta o usando le Forme Normali o Canoniche.
3. Interpretare la TV per descrivere la FB

Quindi per semplificare i passaggi:

Rete Comb. $\rightarrow$ *Espr. Booleana* $\rightarrow$ *Tavola di Verità* $\rightarrow$ *Descr. Verbale*

8.2 Sintesi di una Rete Combinatoria

Questo è il workfolw inverso dell'analisi, tuttavia invece di partire da una funzione, molto spesso bisognerà partire da una descrizione verbale di una funzione.

Quindi:

1. Trovare dalla descrizione verbale le variabili di input e calcolare la tavola di verità in base alla funzione booleana descritta
2. Dalla tavola di verità bisogna passare ad un'espressione booleana, tendenzialmente si può fare attraverso le **Mappe di Karnaugh** trovando così la Forma Normale (SOP o POS) minima e semplificarla ulteriormente, se possibile, usando assiomi o porte NAND/NOR/XOR/XNOR
3. Ottenuta l'espressione booleana si costruirà la rete combinatoria associando le porte logiche agli operatori dell'espressione

Quindi per semplificare i passaggi:

Descr. Verbale $\rightarrow$ *Tavola di Verità* $\rightarrow$ *Esp. Booleana* $\rightarrow$ *Rete Comb.*

8.3 Minimizzazione di EB

Al fine della realizzazione di una Rete Combinatoria è essenziale trovare la *forma minima* di un'espressione booleana, in questo modo si andrà a ridurre il numero di porte logiche necessarie alla realizzazione.

Questo ha conseguenze in termini di **costo** e **tempo di attraversamento** poiché il tempo di risposta di una rete combinatoria dipende dal numero di poret logiche attraversate e quindi riducendole si avrà una maggiore efficienza del circuito.

Una rete è **minimale** se ha:

- il minimo di porte and (o or)
- per ogni porta, il minor numeri di ingressi

8.3.1 Mappe di Karnaugh

Per effettuare al meglio la minimizzazione si utilizzano le MdK *(Mappe di Karnaugh)*, esse funzionano solo fino a 4 variabili, poi servirebbero metodi più complessi e in genere automatizzati.

Data la seguente tavola di verità di un'ipotetica funzione f:

a b c	f
0 0 0	1
0 0 1	0
0 1 0	1
0 1 1	0
1 0 0	1
1 0 1	1
1 1 0	0
1 1 1	0

Table 2: Tavola di Verità della funzione f

Per realizzare una MdK bisognerà:

1. Riscrivere la Tavola di Verità come tabella a doppia entrata posizionando nel caso di 3 variabili, 2 su un'entrata e la restante nell'altra entrata facendo attenzione al **salto**: poiché il valore delle variabili per ogni riga dovrà differenziarsi per 1 unità, bisognerà invertire 10 con 11 (rispetto all'ordine della tavola di verità)

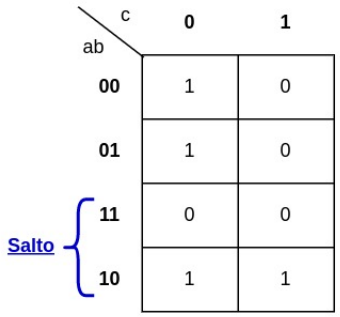
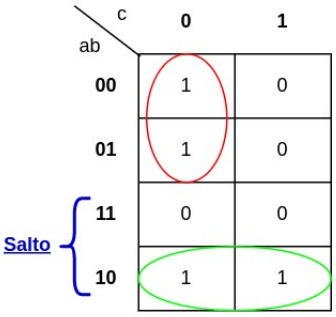


Figure 3: MdK a 3 Variabili

2. Posso realizzare la forma minima sia in forma SOP che POS, in base a quella che voglio realizzare andrò a raccogliere gli 1 o gli 0 **adiacenti** formando dei *cubi* di 2^n elementi



Alla fine di questo processo tutti gli 1 o gli 0 dovranno essere stati raccolti in un insieme, se un elemento non può essere incluso in un insieme allora verrà riscritto il mintermine o maxtermine corrispondente nell'espressione.

La prima e ultima riga/colonna sono considerate adiacenti poiché differiscono di una sola unità, quindi ad esempio posso creare un insieme

che includana un elemento alla prima riga e un'elemento all'ultima riga (della stessa colonna) o anche raccogliendo gli elementi ai 4 angoli.

Esempio:

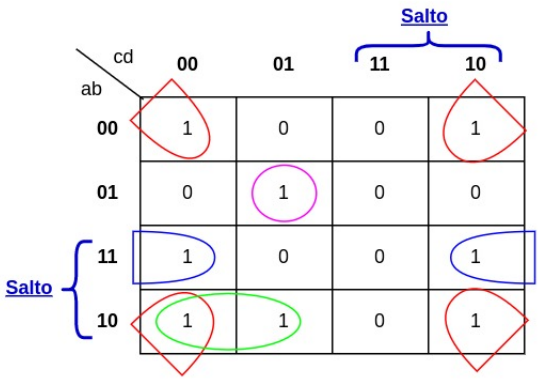


Figure 4: MdK a 4 Variabili

3. Scrivo nell'espressione la somma o il prodotto delle variabili **in comune** dati gli insiemi raccolti.

   Se sto realizzando una forma SOP, voglio che tutte le variabili mi diano 1 (elemento neutro), quindi se la variabile in comune è $x = 0$ nell'espressione andrò a scrivere $\overline{x}$, al contrario in forma POS voglio che la somma mi dia 0 e quindi se avranno in comune $x = 1$ andrò a scrivere nell'espressione $\overline{x}$.

   Quindi facendo riferimento alla figura 4 avrò che:

$$minSOP(f) = \overline{b}\overline{d} + \overline{a}b\overline{c}d + ab\overline{d} + a\overline{b}\overline{c}$$

E' possibile che una funzione booleana non sia definita su tutte le n-ple ma solo su alcune, in tal caso nel risultato della funzione verrà posto un "-" o "δ" (chiamato **don't care**), nella trasposizione in MdK il δ può essere usato sia come 0 che come 1 se tramite esso è possibile creare un cubo più grande.

Esempio di TV con f non definita per tutte le n-ple:

$x_2\ x_1\ x_0$	$y_3\ y_2\ y_1\ y_0$
0 0 0	0 0 1 0
0 0 1	0 0 1 0
0 1 0	0 1 0 0
0 1 1	$\delta\ \delta\ \delta\ \delta$
1 0 0	1 1 1 0
1 0 1	1 1 1 1
1 1 0	0 0 0 0
1 1 1	0 0 0 1

8.4 Moduli Aritmetici

Di seguito si analizzeranno due principali moduli aritmetici che ritroveremo più avanti nella costruzione di moduli più avanzati.

8.4.1 Half-Adder

Questi sono due moduli fondamentali successivamente per la costruzione dell'*addizionatore*. L'*Half-Adder* è il modulo base con cui verrà costruito anche il Full-Adder, esso riceve in input 2 bit e produrrà in output 2 bit, uno che corrisponde al risultato della somma dei due bit in input e il secondo che corrisponde al riporto.

Ecco la TV dell'Half-Adder:

x y	r s
0 0	0 0
0 1	0 1
1 0	0 1
1 1	1 0

L'Half-Adder produce il risultato s tramite $x \oplus y$ e il riporto r tramite $x \cdot y$. Ecco lo schema circuitale:

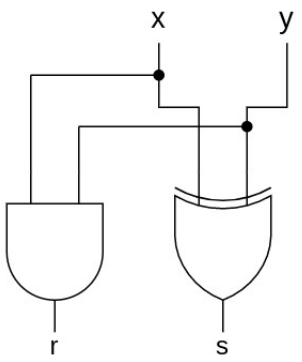


Figure 5: Half-Adder

8.4.2 Full-Adder

Il Full-Adder sono due Half-Adder in OR, esso esegue la somma tra 3 bit producento in out la soluzione s e il riporto r.

Ecco la TV del Full-Adder:

a b c	r s
0 0 0	0 0
0 0 1	0 1
0 1 0	0 1
0 1 1	1 0
1 0 0	0 1
1 0 1	1 0
1 1 0	1 0
1 1 1	1 1

Il Full-Adder esegue quindi $(a + b) + c$ e il riporto della prima somma è in or con il riporto della seconda, quindi $r_{out} = r_1 + r_2$.

Ecco lo schema del Full-Adder come composizione di Half-Adder:

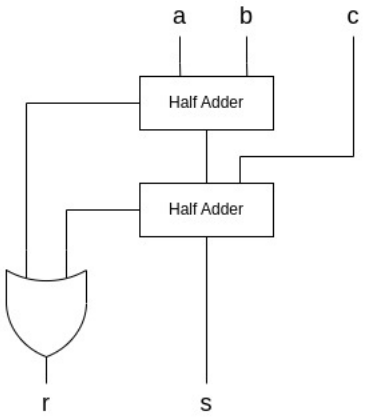


Figure 6: Full-Adder

8.5 Moduli Standard

Oltre ai moduli aritmetici ci sono moduli particolari che si trovano nelle architetture degli elaboratori.

8.5.1 Codificatore

Il codificatore è caratterizzato da:

- n ingressi di cui solo uno vale 1
- $log_2 n$ uscite che rappresentano la codifica associata all'unico ingresso che vale 1.

Ecco la TV del COD base:

i_3 i_2 i_1 i_0	u_1 u_0
0 0 0 1	0 0
0 0 1 0	0 1
0 1 0 0	1 0
1 0 0 0	1 1

Come si può notare $u_1 = i_2 + i_3$ mentre $u_2 = i_1 + i_3$.
Ecco come viene rappresentato circuitalmente:

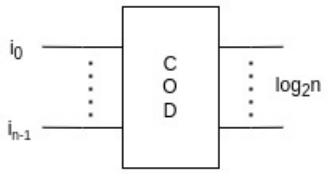


Figure 7: Codificatore

Nella figura 7 vediamo il COD come una black-box, al suo interno possiamo invece trovare il seguente circuito (basato sul COD base):

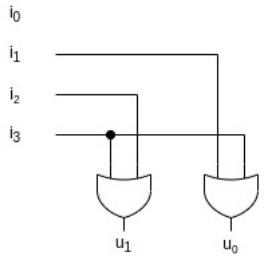


Figure 8: COD Base

Quando, come in questo caso, si ha un insieme di variabili in or (o in and) si usano per semplificare i circuiti delle **matrici** di or (pallini vuoti) o and (pallini pieni). Ecco il precedente esempio usando una matrice di or:

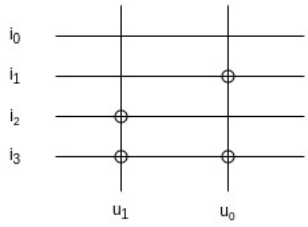


Figure 9: COD Base in Matrice di OR

8.5.2 Decodificatore

Il DEC è caratterizzato da:

- n ingressi
- 2^n uscite di cui una sola vale 1, cioè quella a cui si associa la combinazione in ingresso

Il DEC **produce tutti i mintermini**.

La sua TV è:

i_1 i_0	u_0 u_1 u_2 u_3
0 0	1 0 0 0
0 1	0 1 0 0
1 0	0 0 1 0
1 1	0 0 0 1

Avremo quindi che:

$$u_0 = \overline{i_1} \cdot \overline{i_0}$$
$$u_1 = \overline{i_1} \cdot i_0$$
$$u_2 = i_1 \cdot \overline{i_0}$$
$$u_3 = i_1 \cdot i_0$$

Questo circuito è un insieme di porte *and* e possiamo rappresentarlo con questa **matrice**:

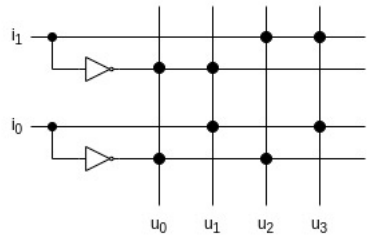


Figure 10: DEC Base in Matrice di AND

8.5.3 ROM (Read Only Memory

La ROM può essere vista sia come unità di memoria o, come in questo caso, una unità logica che realizza espressioni booleane

Essa presenta le seguenti caratteristiche:

- n linee di ingresso *(che, vista come unità di memoria, corrispondono agli indirizzi di memoria)*
- m uscite

Essa viene usata per **rappresentare funzioni booleane**.

Circuitalmente la ROM è un decodificatore con in cascata un codificatore.

Di conseguenza ci sarà una matrice di *and* le cui uscite saranno gli ingressi di una matrice di *or*.

Esempio. Data la seguente TV:

x_1 x_0	y_4 y_3 y_2 y_1 y_0
0 0	0 1 1 0 1
0 1	1 0 1 0 0
1 0	1 0 1 1 1
1 1	0 0 0 0 1

Allora possiamo rappresentarla circuitalmente attraverso la ROM nel seguente modo:

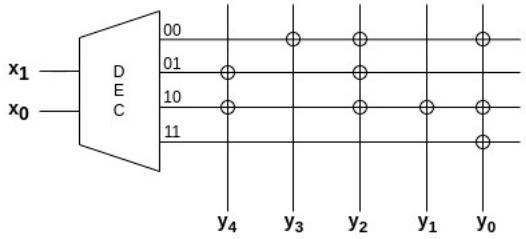


Figure 11: Raffigurazione tramite ROM

Grazie alla ROM possiamo realizzare espressioni in forma canonica.

8.5.4 PLA (Programmable Logic Array)

Il PLA serve a realizzare insiemi di funzioni booleane in forma SOP minimale.

Esso sarà composto da:

- Una matrice di *and* che realizzerà solamente i termini prodotto che appaiono nella espressione minimale
- Una matrice di *or* che per ogni funzione somma i termini prodotto

La massima efficienza del PLA è in casi con 3-4 variabili poiché si possono sfruttare termini prodotto ripetuti.

Esempio. Data la seguente TV:

x_1 x_0	y_4 y_3 y_2 y_1 y_0
0 0	0 1 1 0 1
0 1	1 0 1 0 0
1 0	1 0 1 1 1
1 1	0 0 0 0 1

Allora possiamo rappresentarla circuitalmente attraverso il PLA nel seguente modo:

1. Bisogna ricavare le EB minimali per ogni funzione tramite le MdK:

- $y_4 = \overline{x_1}x_0 + x_1\overline{x_0}$
- $y_3 = \overline{x_1}\ \overline{x_0}$
- $y_2 = \overline{x_1} + \overline{x_0}$
- $y_1 = x_1\overline{x_0}$
- $y_0 = x_1 + \overline{x_0}$

2. Realizzare una matrice di *and* (ruotata) che comporrà quindi i mintermini ed essi verranno poi mandati nella matrice di *or* che andrà a comporre le funzioni in forma minimale

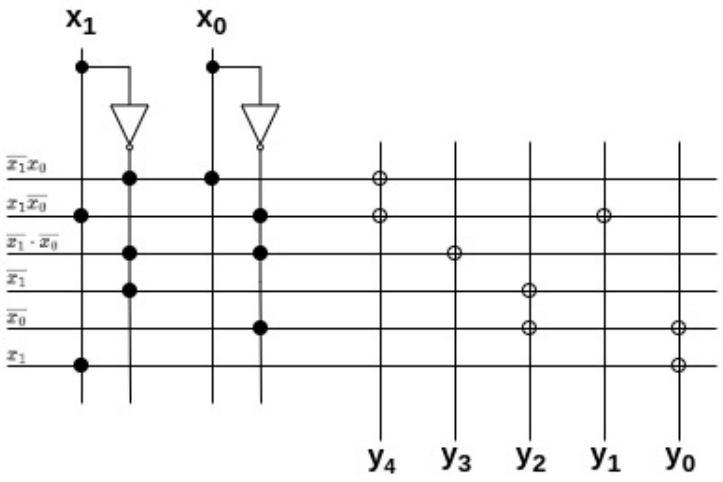


Figure 12: Raffigurazione tramite PLA

8.5.5 Multiplexer

Il MUX è caratterizzato da:

- n linee dati
- n_c linee di controllo (di cui una sola vale 1)
- 1 linea di uscita

Il MUX serve per selezionare tramite le linee di controllo quale linea di ingresso edve essere mandata in uscita. Circuitalmente quello che accade dentro al MUX può essere rappresentato nella seguente maniera:

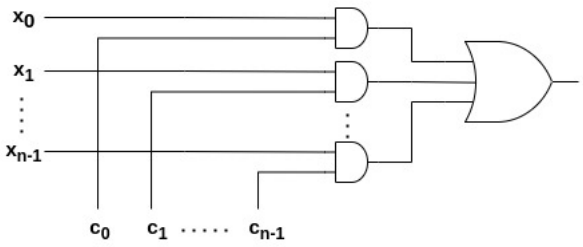


Figure 13: Interno del MUX

Quindi le porte *and* eseguono una **funzione di gating** (o controllo) poiché se l'in = 0 allora sicuramente quella porta sarà = 0, invece se l'in è = 1 allora lo stato della porta dipenderà dall'altro ingresso.

Dato che solamente una linea di controllo può essere = 1, posso sfruttare un decodificatore in cui far passare le variabili in input così da avere esattamente le linee di controllo che mi servono. Per cui possiamo rappresentare il MUX nella seguente maniera:

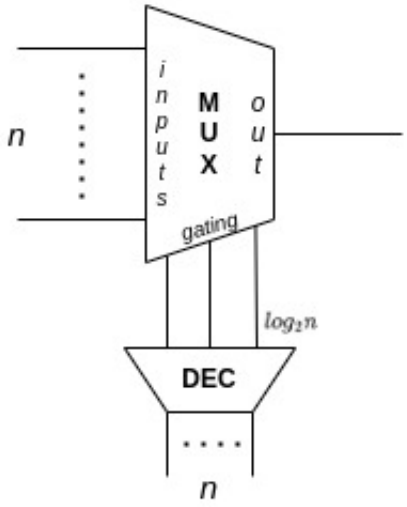


Figure 14: MUX

8.5.6 Multiplexer per Funzioni Booleane

Per ottimizzare il MUX nella rappresentazione di FB, posso usare meno variabili di controllo e rappresentare gli input osservando il comportamento della funzione rispetto alla variabili mancanti (nel controllo).

Esempio. Data la TV di f:

x_2 x_1 x_0	f
0 0 0	0
0 0 1	1
0 1 0	0
0 1 1	0
1 0 0	1
1 0 1	1
1 1 0	1
1 1 1	0

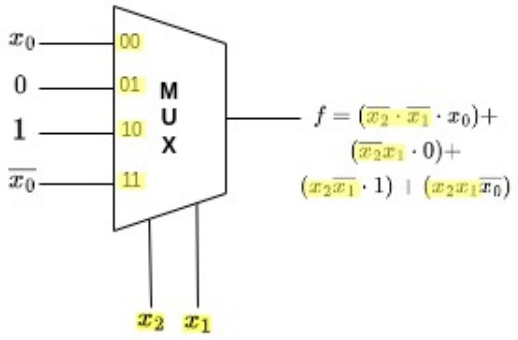
Possiamo rappresentare f tramite un MUX nel seguente modo:

In questa maniera in base alla combinazione delle linee di controllo, il MUX deciderà che input mandare in out.

In questa maniera però abbiamo dovuto utilizzare un MUX 8:1, invece possiamo usare un MUX 4:1 e, per ogni combinazione delle due linee di controllo, osserviamo come si comporta f rispetto a x_0.

x_2 x_1 x_0	f	MUX_{in}
0 0 **0**	**0**	
0 0 **1**	**1**	x_0
0 1 **0**	**0**	
0 1 **1**	**0**	0
1 0 **0**	**1**	
1 0 **1**	**1**	1
1 1 **0**	**1**	
1 1 **1**	**0**	$\overline{x_0}$

Quindi possiamo realizzare il MUX nella seguente maniera:



8.5.7 Demultiplexer

Il DEMUX svolge l'operazione inversa del MUX, avremo quindi:

- 1 linea di input
- n linee di output

- $log_2 n$ linee di controllo

Il DEMUX quindi sceglierà su quale out mandare il segnale in ingresso, infatti è chiamato anche distributore.

Ecco la sua rappresentazione circuitale:

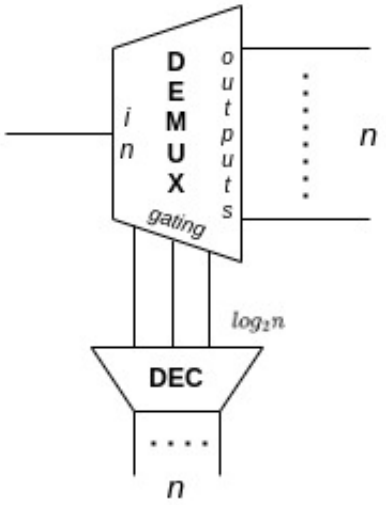


Figure 15: DEMUX

8.5.8 Addizionatore (Ripple-Carry Adder)

L'addizionatore a propagazione di riporto si realizza come una catena di *Full-Adder*, infatti data la seguente operazione di esempio:

$$\begin{array}{rr} & \textit{1 1 1} \\ \text{A} & 0\ 1\ 1\ 1 \\ \text{B} + & 0\ 1\ 0\ 1 \\ \hline & 1\ 1\ 0\ 0 \end{array}$$ possiamo generalizzarla come: $$\begin{array}{r} c_3\ c_2\ c_1\ (c_0) \\ a_3\ a_2\ a_1\ a_0 \\ +\ b_3\ b_2\ b_1\ b_0 \\ \hline s_3\ s_2\ s_1\ s_0 \end{array}$$

e realizzare quindi l'addizionatore concatenando Full-Adder che prendono in input un bit del primo addendo a_i , un bit del secondo addendo b_i e il riporto c_i e producono di conseguenza un bit s_i che farà parte della soluzione e il riporto in out c_{i+1} che sarà il riporto in input del Full-Adder successivo:

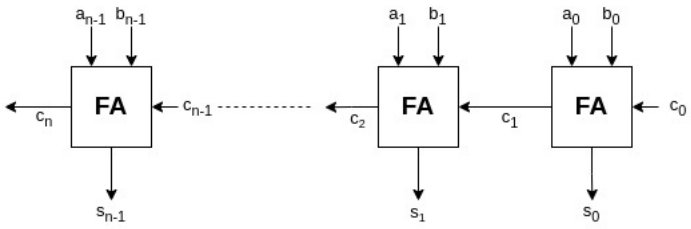


Figure 16: Catena di Full-Adder

C'è da evidenziare tuttavia un problema di *temporizzazione* poiché al momento dell'attivazione del circuito, tutti i Full-Adder daranno immediatamente un risultato **ma** non tutti i calcoli saranno effettuati nello stesso momento quindi bisognerà fissare un momento t_0 in cui il risultato s_0 e il riporto c_1 saranno corretti, successivamente un momento t_1 ... finché non arriverà il momento t_{n-1} in cui tutti i bit-soluzione saranno corretti.

Per la sottrazione si userà la rappresentazione in CA_2 e quindi $A - B = A + (-B)$ che tuttavia è rappresentabile anche come $A + (\overline{B} + 1)$. Quindi nell'addizionatore verrà passato $\overline{b}$ per ogni b e verrà settato $c_0 = 1$.

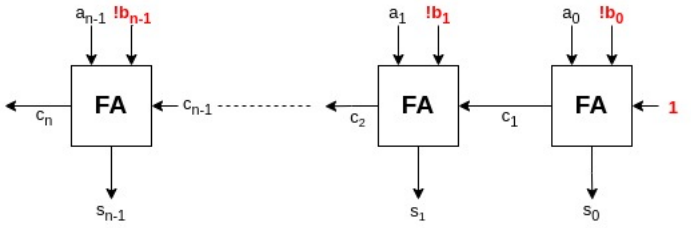


Figure 17: Addizionatore per la sottrazione

Infine avrò un segnale di controllo che indicherà all'*adder* se effettuare un'addizione (0) o una sottrazione (1), quindi infine il circuito dell'adder sarà:

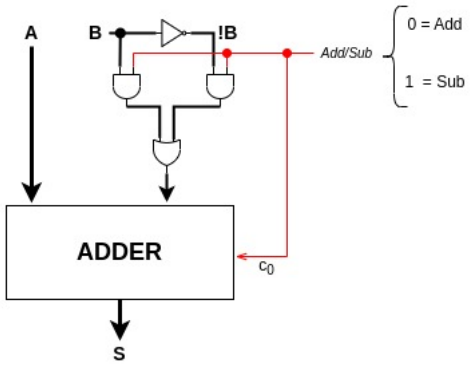


Figure 18: Adder

8.5.9 Comparatore Logico

Il comparatore logico stabilisce se due sequenze di bit sono uguali e restituisce 1 se $in_1 = in_2$ e invece 0 se $in_1 \neq in_2$.

Per eseguire questo controllo il modulo prende in esame coppie di bit, uno appartenente alla prima sequenza ed uno appartenente alla seconda, se una coppia differisce allora restituirà 0.

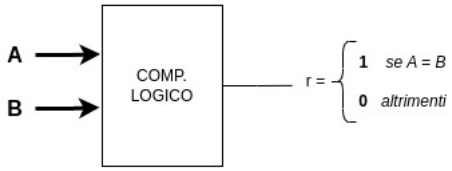
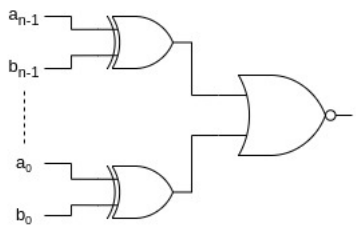


Figure 19: Comparatore

All'interno il funzionamento è realizzato da un circuito in cui tutte le coppie in input sono in XOR *(così da restituire 1 solo quando differiscono)* e gli output sono canalizzati in una porta NOR *(così da restituire 1 se tutte le coppie sono uguali)*.



8.5.10 Comparatore Aritmetico

Il secondo tipo di comparatore è quello aritmetico, esso determina una relazione di $>$, $<$ e lo fa mediante una sottrazione, infatti $A < B \longleftrightarrow A - B < 0$.

Inoltre dato che nell'addizionatore i numeri vengono rappresentati in CA_2 , basterà guardare il MSB: se $= 1$ allora $A < B$.

Possiamo quindi rappresentare il comparatore aritmetico nella seguente maniera:

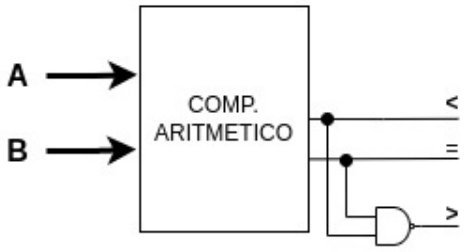


Figure 20: Comparatore Aritmetico

9 Reti Sequenziali

La base su cui si f ondano le reti sequenziali è il bisogno di **memorizzare**.

Le uscite di una rete sequenziale dipendono tanto dai valori presenti in quel momento quanto dai valori precedenti, infatti le funzioni generate dalle reti sequenziali sono **dipendenti dal tempo**.

Nelle reti sequenzaili sono ammessi quindi *cicli*, cioè sono ammesse linee di feedback. Quindi, in sostanza, per generare un risultato Q, avremo bisogno di Q stesso, ma in una frazione di tempo differente.

9.1 Latch e Flip-Flop

9.1.1 Latch SR

Come prima rete sequenziale analizziamo il Latch SR (Set-Reset). Esso si presenta in questo modo:

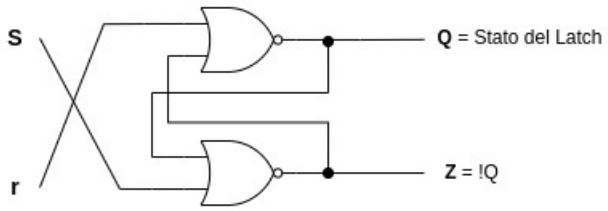


Figure 21: Schema interno del Latch SR

Quindi possiamo ricavare la seguente espressione: $\mathbf{Q(t+1)} = \overline{r + Z} = \overline{r + \overline{S + Q}} = \overline{r}(S + \mathbf{Q(t)})$ e come possiamo notare Q dipende da Q in un tempo antecedente, per cui andrò ad inserire nelle uscite una linea di ritardo e dovrò aspettare un tempo opportuno per avere un nuovo valore di Q.

Ora analizziamo le funzioni del Latch SR:

S r	Q(t+1)	Funzione
0 0	Q(t)	Memorizzazione
0 1	0	Reset
1 0	1	Set
1 1	*Non Ammesso*	*Non Ammesso*

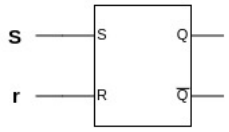


Figure 22: Latch SR

Avendo quindi a che fare con il tempo, si presenta il bisogno di avere un segnale orologio (*clock*) che si può utilizzare per controllare ciò che succede:

- E' rappresentato da un'onda quadra
- E' prodotto da un *Clock Pulse Generator*

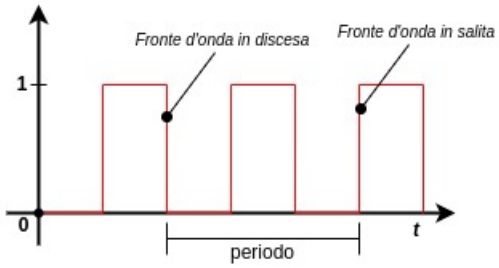


Figure 23: Grafico del Clock

Il clock viene mandato in *and* con altri valori, quindi si avrà un filtro *(come succede ad esempio nel MUX)*. Quindi in un circuito sequenziale le variabili variano in base al clock grazie alla funzione di gating.

9.1.2 Flip-Flop SR

Il Flip-Flop SR è un latch a cui viene aggiunto il clock come filtro, di seguito possiamo vedere com'è realizzato e infine come viene schematizzato:

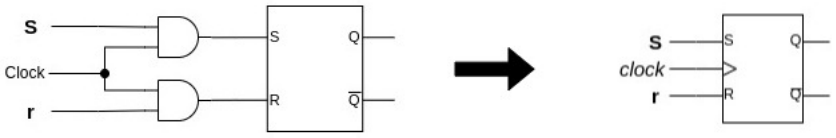


Figure 24: Flip-Flop SR

In questo modo possiamo mandare in maniera temporizzatt ai segnali di ingresso.

9.1.3 Flip-Flop D (Delay)

Il Flip-Flop D è un FF in cui viene mandato un segnale D in forma vera e falsa, quindi si prenderanno in analisi solamente le funzioni di set e reset (poiché non potranno esserci le combinazioni 00 e 11).

Esso trasporterà il segnale mandato al tempo t al tempo $t+1$:

D	Q(t+1)
0	0
1	1

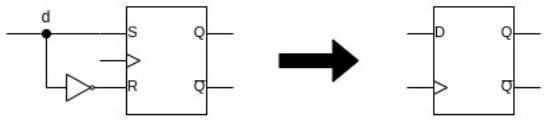


Figure 25: Flip-Flop D

9.1.4 Flip-Flop JK

Nel Flip-Flop SR abbiamo la combinazione 11 che non è assolutamente produttiva, il Flip-Flop JK invece aggiunge a questa combinazione una funzione. Il Flip-Flop JK è così composto e schematizzato:

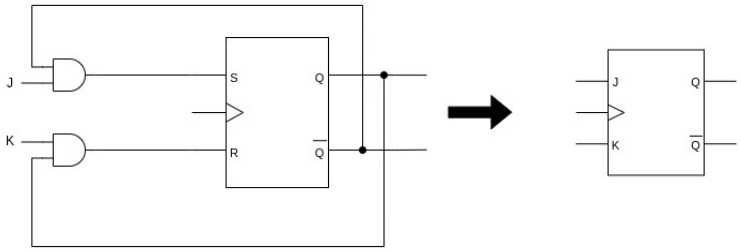


Figure 26: Flip-Flop JK

Introducento le nuove linee J e K avremo che $S = J\overline{Q}$ e $R = KQ$, quindi lo stato dipenderà da J e K:

J K	Q(t+1)	Funzione
0 0	Q(t)	Memorizzazione
0 1	0	Reset
1 0	1	Set
1 1	$\overline{Q}$(t)	Compl. dello Stato

9.1.5 Flip-Flop T (Toggle)

Sulla base del Flip-Flop JK, verrà aggiunto un segnale T che in base al suo valore andrà ad attivare la funzione di Memorizzazione o di Complementazione dello Stato:

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

Ed ecco la sua costruzione e rappresentazione schematica:

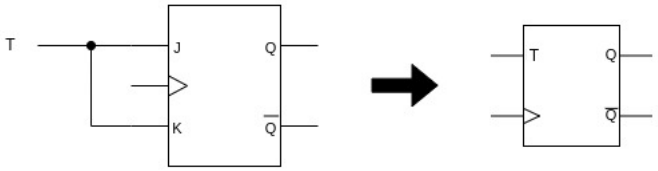


Figure 27: Flip-Flop T

I Flip-Flop sono i dispositivi essenziali per le reti sequenziali (e a loro volta sono essi stessi delle reti sequenziali), un insieme di FF rappresenta la parte di memora in un circuito e ogni FF memorizza un bit.

9.2 Analisi di Reti Sequenziali

In un circuito sequenziale troviamo una parte combinatoria a cui aggiungiamo la parte di memoria data dai Flip-Flop che vengono impostati e mantenuti usando gli ingressi e assegnando valori tramite funzioni di eccitazione. L'analisi di una rete sequenziale si suddivide nei seguenti passaggi:

1. Calcolare le Espressioni Booleane delle funzioni di eccitazione
2. Calcolare le Espressioni Booleane degli output
3. Stendere la Tavola degli Stati Futuri (una estensione della tavola di verità)
4. Rappresentare tramite diagramma l'evoluzione nel tempo, quindi disegnare gli automi a stati finiti
5. Fornire una descrizione verbale di ciò che fa il circuito

Solitamente le reti sequenziali sono divise in due macro-classi: Generatori di Sequenze e Riconoscitori di Sequenze.

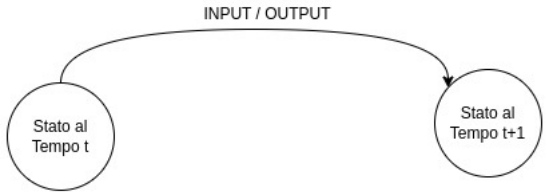
9.2.1 Tavola degli Stati Futuri

Questa tavola rappresenta un'estensione della classica tavola di verità, in essa andremo a racchiudere i seguenti valori:

Comb. degli Input(X) e degli stati dei FF(Q)	Funzioni di Eccitazione	Funzioni di Output	Stato Futuro (Q(t+1))

9.2.2 Automa a Stati Finiti

Il diagramma degli stati mi permette di rappresentare il variare della memoria al variare del tempo e degli ingressi. Ogni stato rappresenta una valore diverso della memoria:



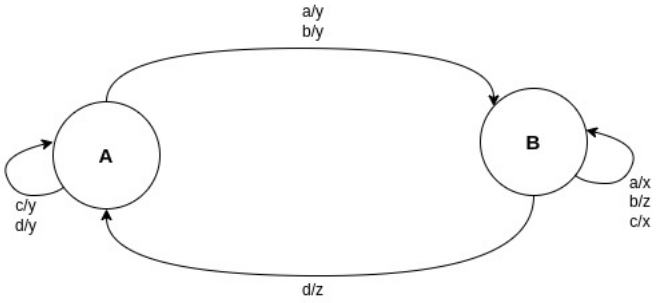
9.2.3 Automa della Macchina Sequenziale

Nei diagrammi di stato spesso ci si sgancia dai nomi e valori di stato e variabili e si passa quindi dal diagramma di stato della rete sequenziale alla rappresentazione della *macchina* sequenziale, quindi unqualsiasi circuito che esegue la funzione in oggetto.

Ad esempio si può creare questa associazione di nomi:

Stato: 0 = A ; 1 = B
In: 00 = a ; 01 = b ; 10 = c ; 11 = d
Out: 00 = w ; 01 = x ; 10 = y ; 11 = z

Ed ecco il diagramma:



9.2.4 Automa di Moore e di Mealy

L'automa a stati finiti è rappresentato da una sestupla $< Q, \Sigma, \delta, q_0, U, \lambda >$, in cui:

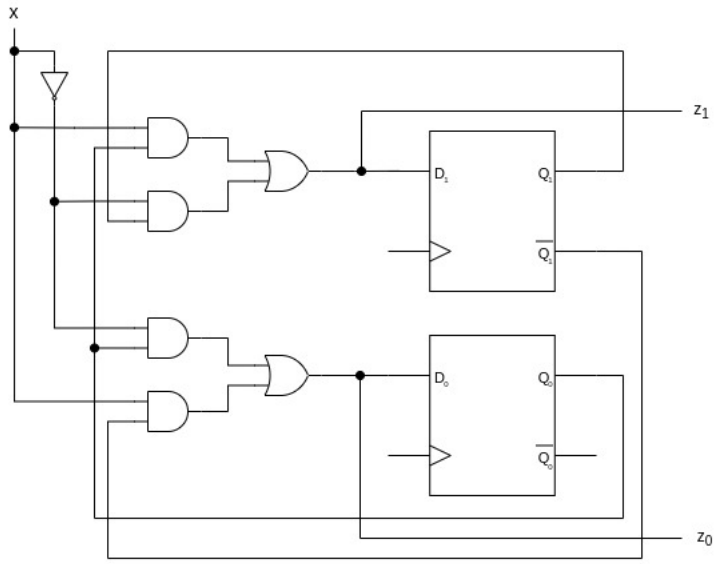
- Q = Insieme finito degli stati
- Σ = Alfabeto di ingresso

- δ = Funzione di transizione: $\delta : Q \times \Sigma \to Q$ (tutte le possibili coppie di stato-ingresso)
- q_0 = Stato iniziale
- U = Alfabeto di Uscita
- λ = Funzione di Uscita. Ed è qui che i due modelli differiscono, poichè:
  - **Modello di Mealy** $\lambda : Q \times \Sigma \to U$. Esso associa l'output all'alfabeto di uscita.
  - **Modello di Moore** $\lambda : Q \to U$. L'uscita è strettamente associata allo stato.

  Quindi nel modello di Mealy sono raffigurati gli stati e per ogni stato viene esplicitato nell'arco di congiunzione l'input e l'output mentre nel modello di Moore si disegnerà uno stato diverso per ogni coppia di stato/output e negli archi verranno quindi solamente identificati gli input che faranno avvenire il passaggio.

9.2.5 Esempio di Analisi di una Rete Sequenziale

Analizzare la seguente Rete Sequenziale

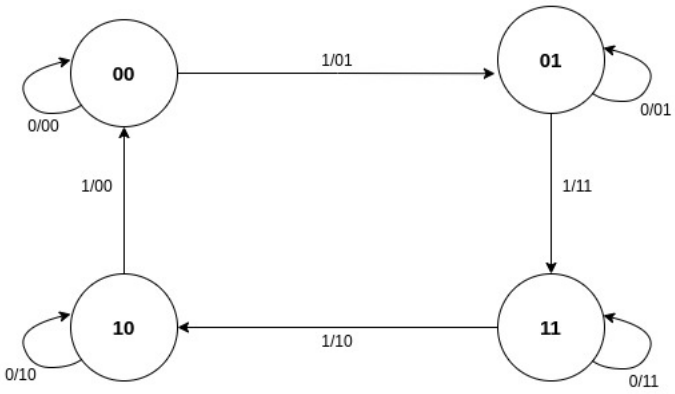


1. **EB delle Funzioni di Eccitazione e degli Output**
   $Z_1 = D_1 = xQ_0 + \overline{x}Q_1$
   $Z_0 = D_0 = \overline{x}Q_0 + x\overline{Q_1}$

2. **Tavola degli Stati Futuri**

x	Q_1	Q_0	D_1 D_0	Z_1 Z_0	Q_1(t+1)	Q_0(t+1)
0	0	0	0 0	0 0	0	0
0	0	1	0 1	0 1	0	1
0	1	0	1 0	1 0	1	0
0	1	1	1 1	1 1	1	1
1	0	0	0 1	0 1	0	1
1	0	1	1 1	1 1	1	1
1	1	0	0 0	0 0	0	0
1	1	1	1 0	1 0	1	0

3. **Automa a stati finiti (diagramma degli stati)**
   Utilizzo Q_1Q_0 per rappresentare lo stato di partenza, in base all'input di x trovo lo stato a t+1 e faccio gli opportuni collegamenti.



4. **Considerazioni**
   Noto che se passo in input 0, rimango nello stesso stato mentre se ricevo 1 avviene un cambio di stato. Questo automa è un **contatore di 1 modulo 4** (poiché avrò 4 cambi di stato prima di tornare al primo). La sequenza di conteggio è caratterizzata dal differimento di un solo bit (codice di Gray)

Posso inoltre effettuare i seguenti passaggi aggiuntivi:

- **Automa sotto forma di tabella**
  Codifico innanzitutto gli stati nel seguente modo:
  - $S_0 = 00$
  - $S_1 = 01$
  - $S_2 = 10$
  - $S_3 = 11$

  Successivamente stendo la seguente tabella a doppia entrata:

	0	1
S_0	$S_0/00$	$S_1/01$
S_1	$S_1/01$	$S_3/11$
S_2	$S_2/10$	$S_0/00$
S_3	$S_3/11$	$S_2/10$

- **Diagramma Temporale della Rete Sequenziale** Posso vedere l'evoluzione degli stati della rete in base ad uno stato di partenza e una sequenza di input, in esso rappresentiamo quindi:
  1. Il Clock
  2. Sequenza di ingresso (con tutti i bit)

3. Sequenza di stati (con tutti i bit)
4. Sequenza degli output (con tutti i bit)

Ad esempio possiamo prendere come riferimento di stato $S_3 = 11$ e la sequenza di input: 1011.
Ecco il diagramma temporale:

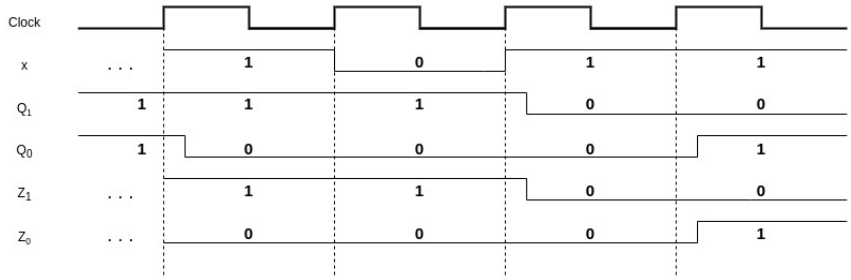


Figure 28: Diagramma Temporale

9.3 Minimizzazione di Automi

Può essere che data la tavola di un automa, ci siano più stati **equivalenti**, quindi che a fronte degli stessi ingressi, essi produgono gli stessi output e transitano negli stessi stati successivi. Posso quindi racchiudere tutti gli stati equivalenti in un insieme di stati che rappresentano un determinato comportamento.

Per minimizzare gli automi si utilizzerà una tabella triangolare (ricavata dalla tavola dell'automa) per confrontare tutti gli stati tra di loro ma senza ripetizioni.

Una volta stesa la tabella inizierò a confrontare gli stati e scriverò:

1. X se gli stati non sono equivalenti (mi basti osservare che ad esempio producono diversi output a fronte degli stessi input)
2. Lascerò la casella vuota se gli stati sono equivalenti
3. Scriverò le coppie di stati rispetto all'input se si hanno gli stessi output poiché potrebbero essere equivalenti per transitività
4. Finita la tabella tornerò a rivedere le situazioni dubbie usando le proprietà di equivalenza.

9.4 Sintesi di Reti Sequenziali

Il processo di sintesi di una rete sequenziale è il procedimento che ci porta da una descrizione verbale a disegnare il circuito per realizzarla e questo procedimento è caratterizzato dalle seguenti fasi:

1. Individuare dalla descrizione verbale gli input e output
2. Stendere la tavola dell'automa della macchina sequenziale
3. Disegnare il diagramma della macchina sequenziale
4. Codifico, se necessario, gli stati, gli ingressi e le uscite (quindi passo all'automa della rete sequenziale) e scelgo quindi cosa memorizzare
5. Scelgo quali Flip Flop utilizzare e stendo la tavola degli stati futuri
6. Ricavo le espressioni booleane delle funzioni di eccitazione e degli output (tramite K-Mappe)
7. Disegno la rete sequenziale

Ecco alcuni accorgimenti per la sintesi di reti sequenziali:

- Il numero di FF da utilizzare sarà $log_2 S$ dove S è il numero degli stati, approssimato. Quindi se ad esempio ho 6 stati, dovrò utilizzare 3 FF, se ne ho 3-4 ne utilizzerò 2, se ne ho 1-2 ne posso utilizzare 1.
- La codifica degli stati si intende attribuire ad uno stato X la combinazione di 0 e 1. Ad esempio se avessi 4 stati potrei codificarli come segue:

Stato	Q_1 Q_0
A	0 0
B	0 1
C	1 0
D	1 1

- Assicurarsi prima di codificare gli stati che l'automa sia minimo.
- Oltre agli stati, in base alla descrizione verbale, posso codificare anche gli input. Ad esempio al posto di A, B, C posso identificarli come x_1, x_0 grazie alla combinazioni 00, 01, 1-.

10 Registri

Per memorizzare 1 bit bisogna utilizzare un FF ma gli elaboratori lavorano con sequenze di bit, per questo si usano i **registri**, ossia *un insieme di n FF ordinati in cui memorizzare sequenze da n bit.*

10.1 Caricamento e Scaricamento Dati

In bae alla modalità di scrittura e lettura dei dati abbiamo le seguenti tipologie di registri:

1. **PIPO**: Parallel Input-Parallel Output
2. **SISO**: Serial Input-Serial Output
3. **SIPO**: Serial Input-Paralell Output
4. **PISO**: Parallel Input-Serial Output

10.1.1 Registro PIPO

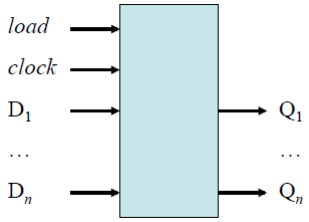
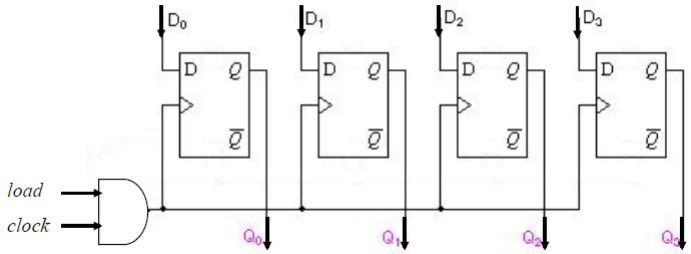


Figure 29: Registro PIPO

Questo registro riceve in input n linee di dati, tali linee vengono memorizzate quando il segnale di *load* è a 1 e al momento di un fronte d'onda discendente del clock. Il valore memorizzato viene riproposto in out dopo un certo ritaro τ.

Ecco com'è formato internamente un registro PIPO:

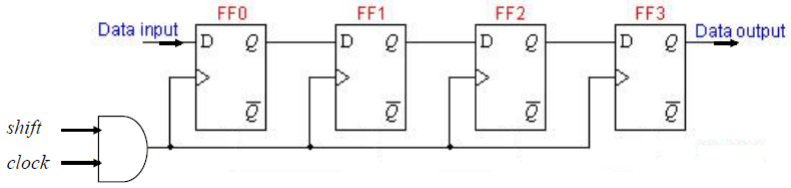


10.1.2 Registro SISO

Figure 30: Registro SISO

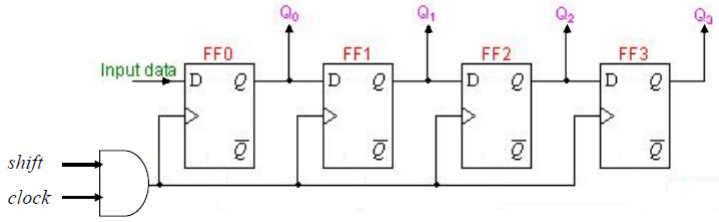
Il registro SISO avrà un **caricamento seriale**, ossia un segnale unico di ingresso e in base al segnale di controllo *shift* i dati scorrono verso i FF alla loro dx. Nel medesimo modo avviene l'output, quindi uno ad uno i FF scaricano in out i dati memorizzati nell'ordine in cui erano entrati nel registro, questo avviene quando lo shift è settato a 1 e durante un fronte d'onda discendente del clock.

Ecco com'è composto un registro SISO:



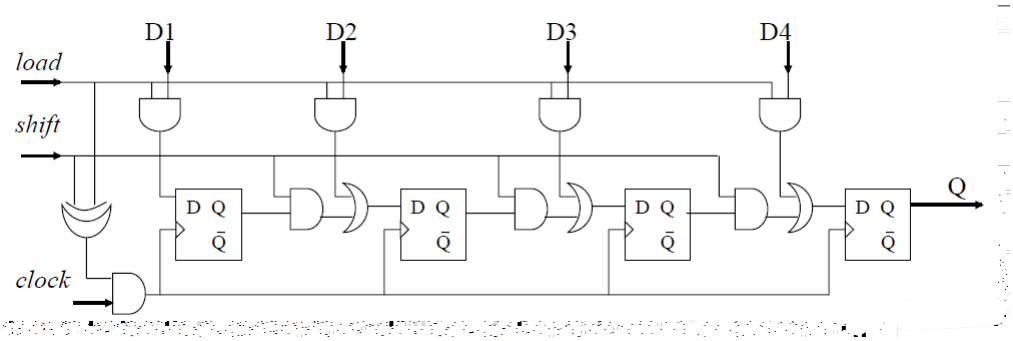
10.1.3 Registro SIPO

Il registro SIPO sarà un ibrido, infatti avrà un caricamento **seriale** e uno scaricamento **parallelo**, infatti serviranno n colpi di clock per caricare i dati ma uno solo per scaricarli.



10.1.4 Registro PISO

Anche questa tipologia è un'ibrido ed è la più complessa circuitalmente, in questo caso avrò un caricamento **parallelo** quindi con un solo colpo di clock, mentre lo scaricamento sarà seriale e quindi mediante n cicli di clock. Inoltre il segnale di *load* e di *shift* **non** potranno essaere a 1 nello stesso momento.



10.2 Funzionalità dei Registri

Indipendentemente dalle modalità di input/output, un registro possiede due fondamentali funzioni:

- **Funzione di Shift**: Registri a Scorrimento
- **Funzione di Count**: Registri Contatori

10.3 Registri a Scorrimento

I registri a scorrimento, detti anche shifter, sono utili nel momento in cui bisogna shiftare la sequenza di bit per realizzare ad esempio moltiplicazioni e/o divisioni in base 2, difatti esattamente come nel sistema decimale se moltiplichiamo o dividiamo per una potenza di 2 dovremo far scorrere la sequenza a destra o sinistra. Essi sono composto fondamentalmente grazie ad uno o più segnali di controllo per eseguire lo shift del contenuto.

I registri a scorrimento vengono tipicamente realizzatic on FF di tipo D.

Infatti avremo diversi tipi di registri shifter:

- **Right Shifter**: Ad ogni fronte d'onda discendente in cui shift vale 1, il contenuto dei FF si sposta da i a i+1
- **Left Shifter**: Funziona come il right shifter ma il contenuto si sposta dal FF i a i-1
- **Shifter Bidirezionale**: Esso può eseguire lo shift sia a destra che a sinistra in base ad un segnale di controllo *right* che: se vale 1 si esegue lo shift a destra, se vale 0 si eseguirà lo shift a sinistra
- **Shifter con Mantenimento dei Bit**: nel momento in cui si shifta il contenuto a destra o sinsitra si perderà il MSB o LSB, se ad esempio si lavora interpretando i bit in CA_2 si andrà a sballare il valore della sequenza, quindi possiamo mandare in input al primo o ultimo FF il suo stesso output così da mantenere il bit ed esso può essere attivato o meno da un segnale di controllo *hold*.
- **Registri Circolari** Essi, come indica anche il nome, funzionano a loop quindi quando il registro scarica l'output, sarà esso stesso il suo input, essi vengono utilizzati se si vogliono considerare sequenzialmente e ripretutamente tutti i bit, uno alla volta. Essi possono eseguire il loop in senso orario, antiorario e bidirezionale.

10.4 Registri Contatori

Un contatore è un registro usato per contare il numero di occorrenze di un determinato evento, sempre modulo un certo numero naturale. Quindi se formato da n FF, un contatore potrà contare fino a modulo 2^n.

Tipicamente gli evento che può contare sono i colpi di colck o le occorrenze di alcuni valori (o sequenze) di input.

Essi si distinguono in due categorie: sincroni e asincroni.

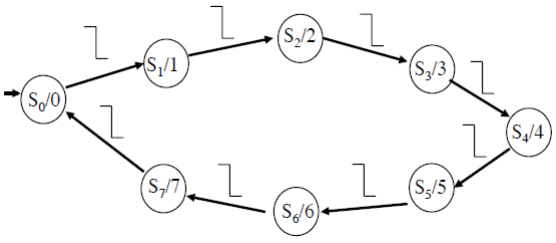
Entambe le categoria tuttavia possono contare a salire o a scendere e possono essere settabili, quindi forzare un valore che non rispetta la normale sequenza.

Tipicamente i contatori vengono realizzati con FF di tipo JK.

10.4.1 Contatori Sincroni

Un contatore modulo 2^n parte da 0 e ad ogni fronte d'onda discendente del clock incrementa di 1 il suo valore fino ad arrivare a $2^n - 1$, poi torna a 0 e ricomincia.

Possiamo vedere le sue transizioni prendendo in esempio un contatore modulo 8:



10.4.2 Contatori Asincroni

Essi son formati da FF con ingressi asincroni, sono quindi due ingressi aggiuntivi ai comuni FF con le funzioni di **preset** e di *clear*, che funzionano in maniera *asincrona* rispetto al clock, ossia sono usati per settare o resettare il FF in modo istantaneao, indipendentemente da input o colpi di clock.

Nei registri contatori vengono usati gli ingressi asincroni dei FF in genere per avere un contatore modulo $k \neq 2^n$, quindi per passare da k-1 a k, si andrà a forzare un reet del contatore mediante il segnale *clear*.

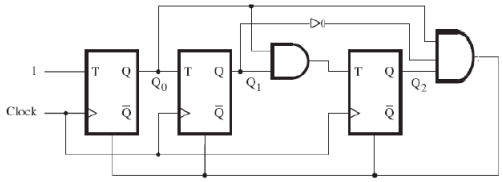


Figure 31: Contatore MOD 5

Un altro uso è invece la realizzazione di contatori preselezionabili, ossia sfruttare la funzione di *preset* per forzare un valore e mantenerlo indipendentemente da input e clock.

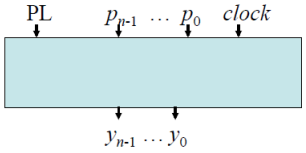


Figure 32: Contatore Preselezionabile

Quindi quando il segnale PL vale 0, esso si comporta come un contatore MOD 2^n, quando viene invece impostato a 1, esso forza i valori presenti sulle linee di ingresso dei rispettivi FF e finché PL varrà uno verrà mantenuto in memoria il valore forzato sulla linea di preset.

10.5 Interconnessione fra Registri

Per venire elaborate o memorizzate, le informazioni devono essere trasferite/copiate da un registro all'altro all'interno di un elaboratore.

Questo processo di trasferimento si effettua tramite **reti di interconnessioni**.

Per creare una rete di registri non mi interessa la tipologia di essi o le modalità di input/output, a questo livello di astrazione verranno viste come black-box.

Ci sono 4 tipi di interconnessione:

1. Interconnessione Punto a Punto (sorgente e destinazione prefissate)
2. Sorgente Variabile e Destinazione Prefissata (molti a uno)
3. Sorgente Prefissata e Destinazione Variabile (uno a molti)
4. Sorgente e Destinazione Variabile (molti a molti)

10.5.1 Interconnessione Punto a Punto

Questo è il caso più semplice, è già prefissata sia la sorgente che la destinazione quindi bisognerà collegare l'output di R con l'input di R' e settare il segnale di controllo *inR* a 1 quando si vuole attivare il trasferimento.

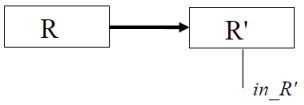
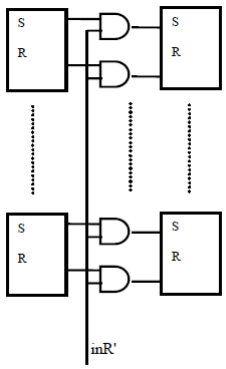


Figure 33: Interconnessione Punto a Punto

Quindi avrò sia nela sorgente sia nella destinazione dei FF SR connessi nel seguente modo:



Sfrutterò quindi porte logiche and oppure un buffer tristate.

10.5.2 Interconnessione Molti a Uno

In questo caso avrò un numero variabile di registri sorgente e un registro di destinazione prefissato. Per realizzare questa rete si userà un multiplexer *(schematicamente se ne indicherà uno ma in realtà ce ne sarà uno per ogni FF).*

Infine i segnali di selezione del MUX forniscono la codifica binaria dell'indice i del registro R_i il cui contenuto deve essere copiato in R_d.

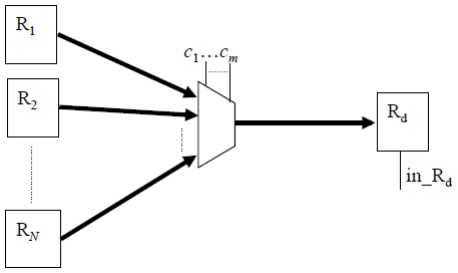
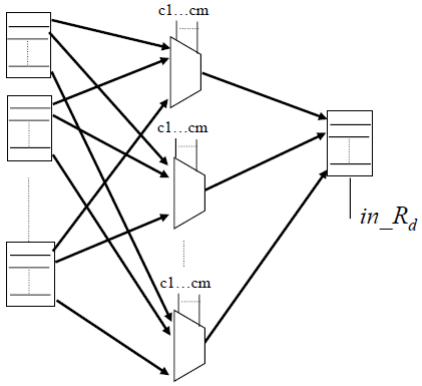


Figure 34: Interconnessione con Sorgente Variabile e Destinazione Prefissata

Ecco il funzionamento di selezione nel dettaglio:



10.5.3 Interconnesione Uno a Molti

Per questa rete avrò un registro sorgente prefissato e un numero variabile di registri destinazione, si sfrutterà di conseguenza un decodificatore. Avrò un *inR* globale in and con le uscite del decodificatore che attiverà il trasfrimento di dati.

Gli ingressi del decodificatore saranno le linee di controllo $c_0 \dots c_{n-1}$ che forniscono la codifica binaria dell'indice i del registro R_i dove copiare l'informazione contenuta in R_s.

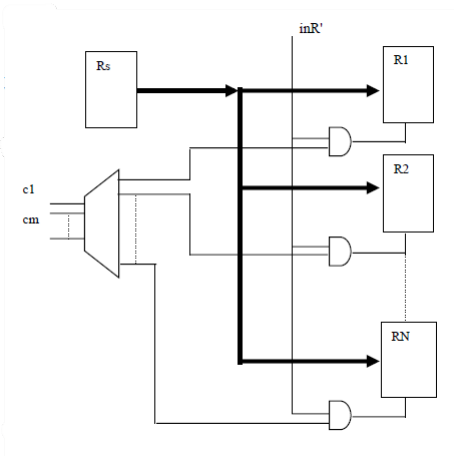


Figure 35: Interconnessione con Sorgente Prefissata e Destinazione Variabile

10.5.4 Interconnessione Molti a Molti

Infine troviamo il caso in cui sia i registri sorgenti sia i registri destinazione possono essere di numero variabile, questa tipologia si può effettuare in due modi: mediante **mesh** o mediante **bus**.

Interconnesione tramite Mesh In questa modalità avremo per ogni registro sorgente un MUX che prende in input *tutte* le linee degli altri registri sorgente e tramite le linee di controllo si decide quale R_s mandare a quale R_d.

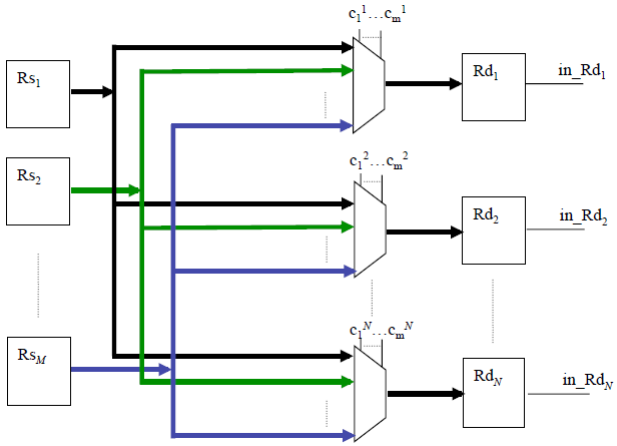


Figure 36: Interconnessione tramite Mesh

Il vantaggio di questa modalità è poter avere tanti caricamenti in parallelo ma è una soluzione **estremamente costosa**.

Posso ridurre il costo sfruttando un solo MUX ma avendo così perso l'opzione del parallelismo posso decidere di eseguire questi trasferimenti mediante bus.

Interconnessione tramite Bus Un bus è una fascia di n linee che connettono i registri sia in input che in output, per evitare conflitti infatti si useranno dei buffer tristate per abilitare il trasferimento sul bus dei dati.

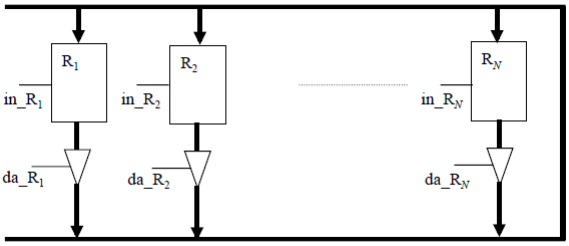


Figure 37: Interconnessione tramite Bus

Negli elaboratori abbiamo 2 tipi di memorie:

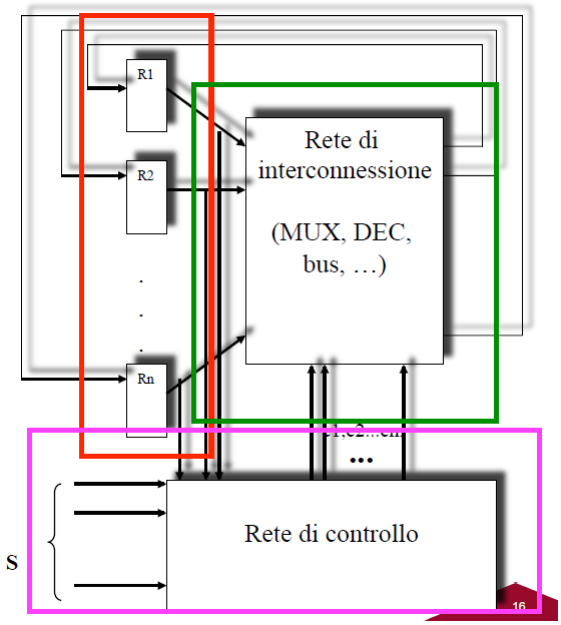
- **Centrale**(nel microprocessore)
- di **Massa**

La memoria centrale ha pochi registri e ha bisogno di tanta velocità quindi si opterà per una interconnessione tramite mesh, mentre nella memoria di massa che è formata da milioni di registri si opterà per una soluzione più lenta ma meno costosa come il bus.

10.5.5 Progettare un'Interconnessione

Il progetto di una rete di registri è divisa in 3 parti:

1. I Registri coinvolti
2. La rete di Interconnessione (la tipologia varia in base alle specifiche del progetto)
3. Rete di Controllo (la parte combinatoria che gestisce l'abilitazione dei trasferimenti di dati)



10.5.6 Tips per la risoluzione degli esercizi

Per risolvere gli esercizi dovremo dividere il processo in due step:

- Individuare il tipo di interconnessione da realizzare in base alla specifica verbale
- Impostare la parte combinatoria mediante l'uso di moduli *(cmp, adder...)* e porte per abilitare il trasferimento dei dati