Appunti di Elettronica Digitale

Capitolo 3 - Parte I

Circuiti MSI

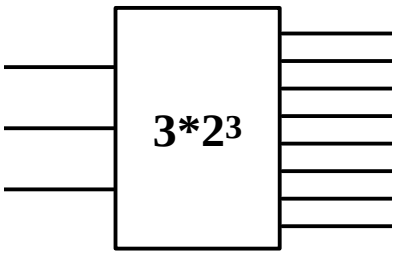
INTRODUZIONE

L'acronimo **MSI** sta per *medium scale of integration*, ossia per media scala di integrazione. I **circuiti MSI digitali** sono circuito che comprendono circa 10 porte logiche sullo stesso chip di silicio, corrispondenti quindi a circa 30÷40 transistor. Circuiti di questo tipo sono normalmente in commercio e costano molto poco, tanto che non conviene assolutamente realizzarli da se.

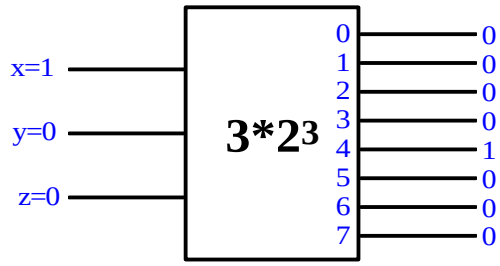
I circuiti MSI permettono di realizzare funzioni logiche combinatorie fino a 7÷8 variabili di ingresso, senza che sia necessario effettuare una minimizzazione delle funzioni stesse (questo aspetto sarà chiaro negli esempi che seguiranno).

DECODER

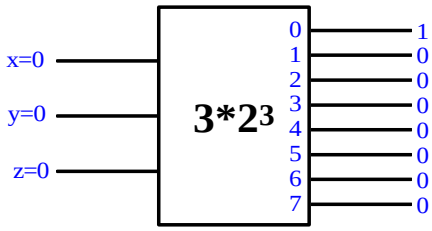
Come primo esempio di circuito MSI digitale consideriamo il cosiddetto **decoder** (decodificatore). Lo scopo di questo circuito è quello si fornire in uscita la traduzione decimale della combinazione binaria che riceve in ingresso. In altre parole, esso riceve in ingresso un certo numero N di bit e fornisce in uscita, nel modo che vedremo tra un attimo, il valore decimale della parola formata da questi N bit. E' chiaro che ad N bit in ingresso corrispondono 2^N possibili valori dell'uscita ed è per questo che il simbolo circuitale di questo circuito è una "scatola" con dentro la scritta **$N*2^N$** che indica il numero di ingressi ed il numero di uscite (che, come vedremo, corrisponde al numero di porte AND necessario per l'implementazione). Nel caso, per esempio, di N=3, il simbolo è il seguente:

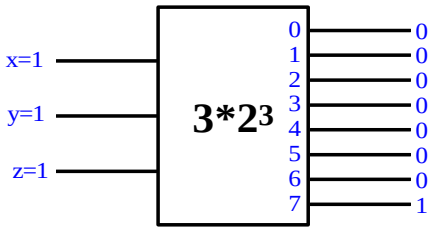


Il modo con cui il circuito effettua la conversione binario→decimale descritta prima è semplice: indichiamo con x,y,z i tre ingressi al circuito e quindi con xyz la parola binaria corrispondente; per esempio, supponiamo che sia x=1,y=0 e z=0, per cui la parola binaria è 100; le uscite sono numerate da 0 fino a 2^3-1=7, come nella figura seguente: allora, dato che il valore decimale di 100 è 4, viene attivata la linea 1, cioè viene posta al livello 1,, mentre le altre rimangono disattivate, cioè poste al valore 0:



E' ovvio che, quando la combinazione di ingresso è x=0,y=0,z=0, la linea che viene attivata è la linea 0, mentre, quando gli ingressi sono tutti ad 1, la linea attivata è la 7:





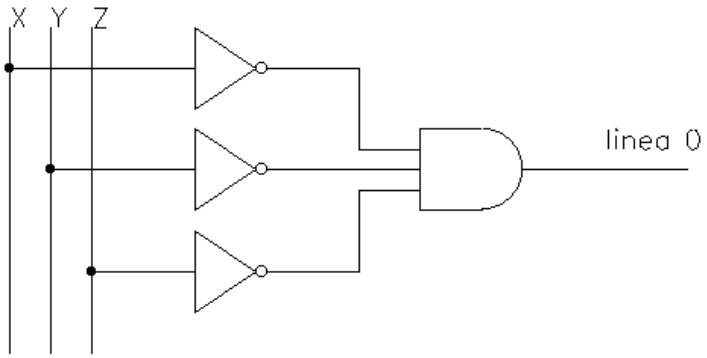
In definitiva, quindi, qualunque sia la combinazione di ingresso, viene attivata solo una linea di uscita.

Dobbiamo adesso capire come realizzare questa funzione, ossia con quale schema logico. Andiamo allora a rappresentare la tavola della verità della funzione:

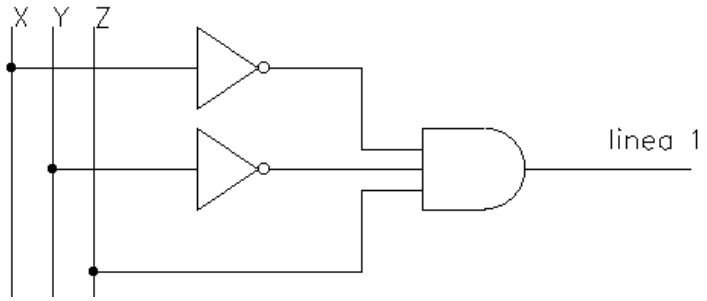
x	y	z	0	1	2	3	4	5	6	7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Intanto, per ottenere 8 uscite avremo bisogno di 8 porte logiche in uscita:

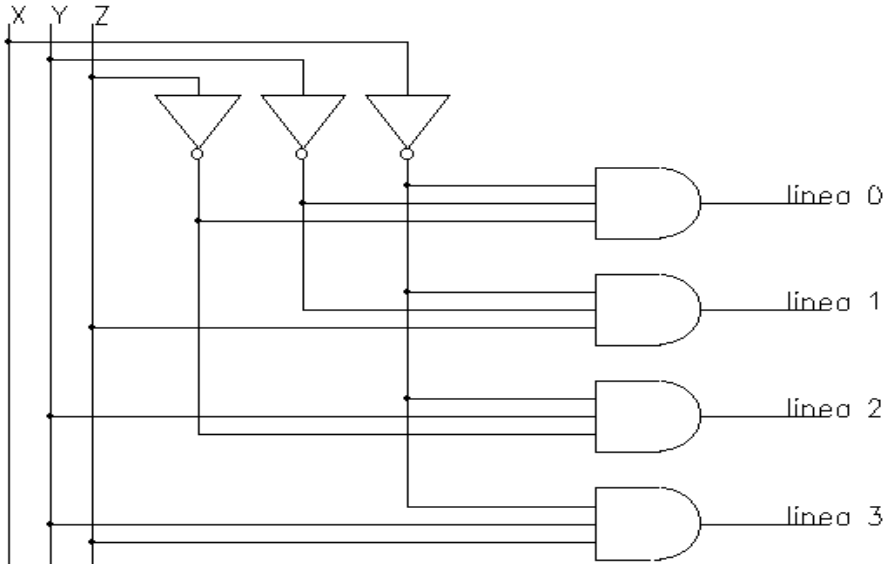
- dalla porta logica la cui uscita corrisponde alla linea 0 deve venir fuori 1 solo se tutti e tre gli ingressi sono nulli, o, ciò che è lo stesso, quando tutti e tre gli ingresi complementati valgono 1; allora, si può ottenere questo risultato con una porta AND a 3 ingressi dove i 3 ingressi giungano complementati: infatti, solo se tutti e tre gli ingressi valgono 1, l'AND fornisce il valore 1. Quindi, il circuito logico per la linea 0 è fatto nel modo seguente:



- in modo del tutto analogo, dalla porta logica la cui uscita corrisponde alla linea 1 deve venir fuori 1 solo se l'ingresso z vale 1 e gli altri 2 valgono 0: basta allora usare ancora una volta una porta AND a 3 ingressi dove gli ingressi x ed y giungano complementati:



E' ovvio che non conviene usare, per ciascuna linea, il corrispondente numero di porte NOT in ingresso. Al contrario, conviene predisporre, oltre ai conduttori che recano direttamente i 3 ingressi, altri 3 conduttori che rechino i complementi di tali ingressi. Così facendo, il circuito assume lo schema seguente (sono disegnate, per pura comodità, solo le prime 4 porte):



Si ottiene dunque un circuito formato da tante porte AND quante sono le uscite e tante porte NOT quanti sono gli ingressi: ciò significa che il costo della funzione è $N+2^N$. Nel caso di N=3 ingressi, abbiamo 11 porte, a conferma del fatto che *un decoder a 3 ingressi può essere realizzato con un circuito MSI.*

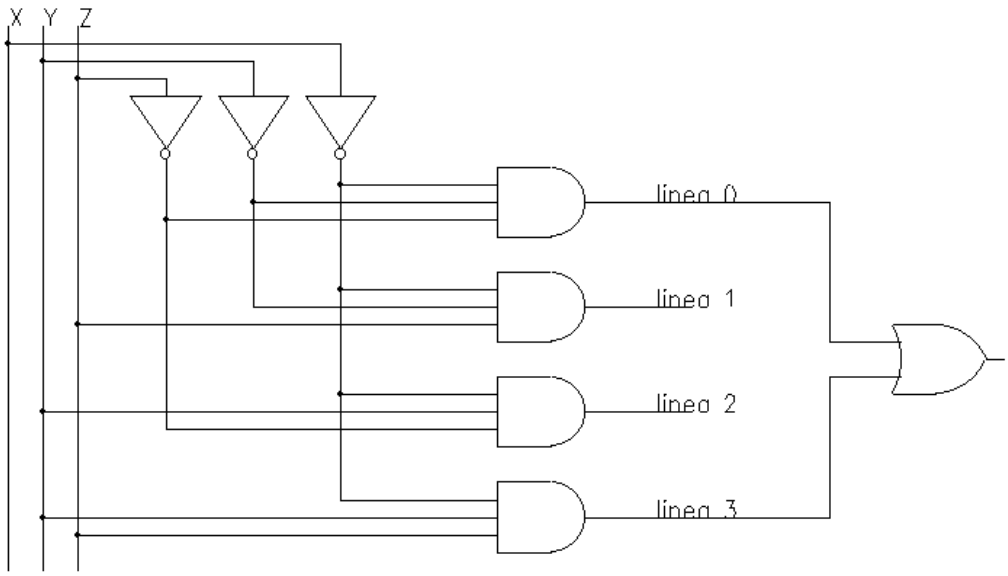
Osservazione

Consideriamo un decoder $3*2^3$: una cosa interessante da osservare è che le 8 uscite del decoder rappresentano i valori degli 8 possibili mintermini di una generica funzione booleana di 3 variabili. Ad esempio, la linea 0 corrisponde al mintermine x'y'z', proprio perchè vale 1 solo quando i 3 ingressi sono entrambi nulli; la linea 1 corrispondente invece al mintermine x'y'z e così via le altre linee. Questo è un risultato importante, in quanto, se dobbiamo implementare una qualsiasi funzione booleana di 3 variabili, ci basterà trovarne la prima forma canonica, in modo da individuare i mintermini presenti, e sommare (con una porta OR) le uscite del DECODER corrispondenti a tali mintermini, senza quindi la necessità di minimizzare la funzione.

Facciamo un esempio. Consideriamo il caso semplice di una funzione T(x,y,z) avente la seguente forma canonica:

$$T = x'y'z'+x'yz$$

Come visto prima, questi due mintermini corrispondono alle linee 0 e 3 del decoder, per cui l'implementazione della funzione sarà possibile semplicemente facendo un OR di tali linee:



Quindi, come preannunciato in precedenza, il decoder è uno di quei tipici circuiti che consentono di implementare, con relativa facilità, le funzioni booleane senza doverle prima minimizzare. E' ovvio che questo è vero fin quando il numero di ingressi è piccolo (si può arrivare fino a 5 o 6 ingressi), dopo di che la minimizzazione diventa invece necessaria (e comunque non si potrebbe più ricorrere a circuiti MSI dato il numero di porte necessario):

Realizzazione con porte NAND

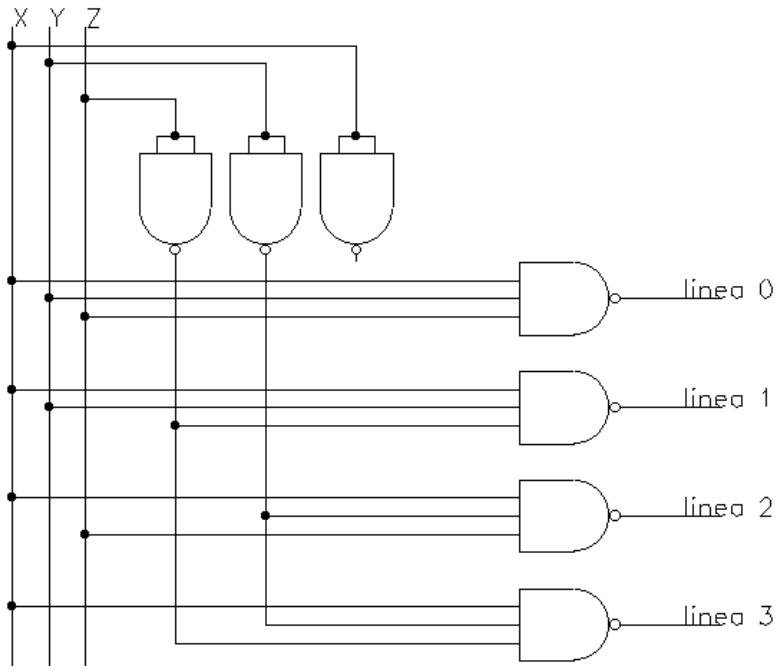
Sappiamo che un qualsiasi circuito logico può essere implementato mediante un qualsiasi insieme funzionalmente completo di porte logiche: l'insieme $\{\text{AND}, \text{OR}, \text{NOT}\}$ è un insieme funzionalmente

completo, ma lo è anche quello formato da sole porte NAND, per cui possiamo pensare di realizzare un decoder anche solo mediante porte NAND.

Il modo con cui realizzare il circuito è assolutamente analogo a quello visto nei paragrafi precedenti, salvo due differenze fondamentali, derivanti entrambe dal fatto che la NAND fornisce, come risultato, il complemento dell'AND degli ingressi:

- in primo luogo, gli ingressi non vanno più complementati quando valgono 0, mentre vanno complementati quando valgono 1;
- in secondo luogo, le linee di uscita saranno disattivate quando sono al valore 1, mentre saranno attivate quando sono al valore 0.

Tenendo conto di queste differenze, il circuito è fatto nel modo seguente (con riferimento sempre alle prime 4 porte di un decoder $3*2^3$):



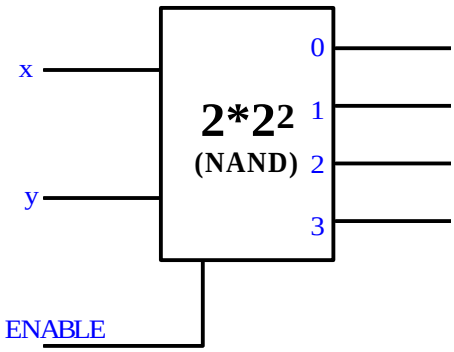
(come si nota nella figura, nessuna delle 4 porte di uscita riceve in ingresso x complementata, che invece viene utilizzata da tutte e 4 le porte successive).

E' ovvio che abbiamo dovuto utilizzare le porte NAND non solo per le uscite, ma anche per realizzare le complementazioni degli ingressi[1]. Il numero di porte è rimasto comunque invariato rispetto a quello del circuito realizzato con porte NOT e porte AND e valgono perciò anche le considerazioni a proposito della possibilità di realizzare tutti i mintermini delle funzioni a 3 variabili booleane.

LINEA ENABLE

Spesso, nei decoder, oltre ai normali ingressi viene aggiunta una particolare linea, detta **linea ENABLE**, che ha lo scopo di far funzionare o meno il circuito a seconda del proprio valore. Consideriamo, ad esempio, un decoder $N*2^N$ realizzato con porte NAND e con linea ENABLE:

[1] Ricordiamo, a tal proposito, che una porta NAND (così come una porta NOR) effettua il complemento di una variabile quando essa viene mandata su entrambi gli ingressi della porta stessa.



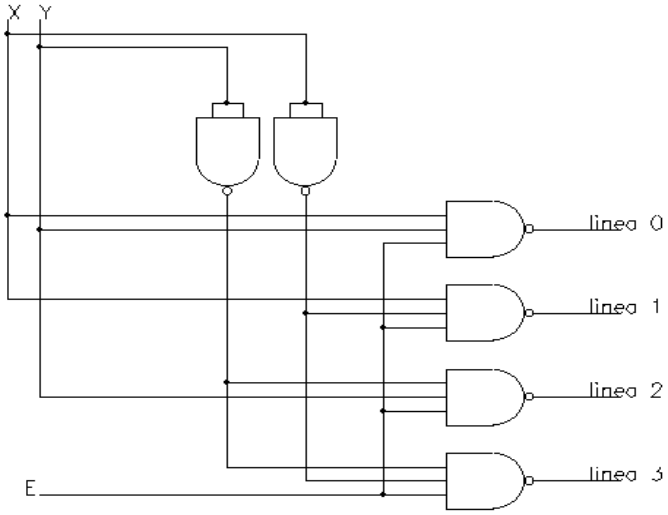
La linea ENABLE serve a questo: quando essa vale 1 (cioè in pratica quando il circuito è attivato), allora il circuito si comporta effettivamente come decoder; quando, invece, essa vale 0, allora il circuito presenta tutte le linee disattivate a prescindere dalla combinazione di bit in ingresso.

Per esempio, abbiamo detto, nel paragrafo precedente, che, nel caso di porte NAND, le linee di uscita si ritengono disattivate quando valgono tutte 1. Allora, la tavola della verità del decoder, considerando la linea ENABLE come un normale ingresso, è la seguente:

x	y	E	0	1	2	3
∀	∀	0	1	1	1	1
0	0	1	0	1	1	1
1	0	1	1	0	1	1
0	1	1	1	1	0	1
1	1	1	1	1	1	0

Per qualunque combinazione di ingresso, se la ENABLE vale 0 il circuito presenta tutte le linee di uscita disattivate. Al contrario, quando la ENABLE vale 1, il circuito attiva (cioè pone a 0) la linea che corrisponde al valore decimale della parola binaria xy in ingresso.

E' molto semplice realizzare circuitalmente questo tipo di funzionamento: per ogni porta NAND corrispondente ad una linea di uscita, basta mandare in ingresso, oltre agli ingressi veri e propri, anche la linea ENABLE, secondo lo schema della figura seguente:



DEMULTIPLEXER

Consideriamo nuovamente la tabella della verità del decoder $2*2^2$ descritto nel paragrafo precedente, concentrandoci, in particolare, su quello che avviene quando la linea ENABLE è attivata:

x	y	E	0	1	2	3
0	0	1	0	1	1	1
1	0	1	1	0	1	1
0	1	1	1	1	0	1
1	1	1	1	1	1	0

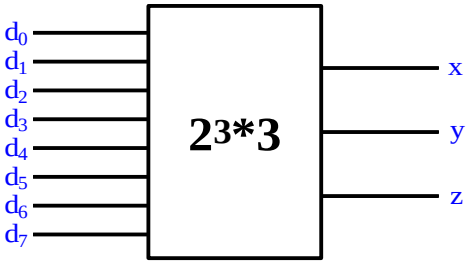
In questa tabella si osserva, tra le altre cose, una corrispondenza tra la linea di uscita che di volta in volta viene attivata (quando E=1) e il valore della linea ENABLE: la linea di uscita indirizzata dall'ingresso xy (e quindi attivata) fornisce sempre, come valore, quello della linea enable complementata.

Quindi, il circuito assume un funzionamento particolare: dato l'ingresso ENABLE posto ad 1, l'uscita posta a 0 è quella indirizzata dalla coppia di ingressi xy. Un funzionamento di questo tipo è caratteristico dei cosiddetti **circuiti multiplexer** (brevemente indicati con **demux**)e questo è il motivo per cui la maggior parte dei decoder vengono individuati con il nome *decoder/multiplexer*.

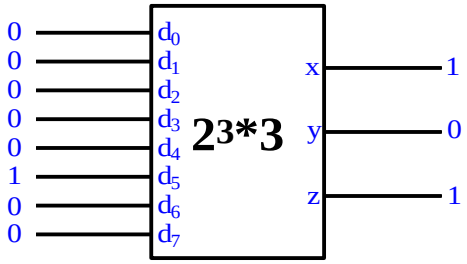
ENCODER

Mentre nei paragrafi precedenti abbiamo visto il funzionamento del decoder, vediamo adesso come funziona e come è realizzato il circuito opposto, cioè il cosiddetto **encoder**.

In primo luogo, l'encoder è un circuito avente 2^N linee di ingresso e N linee di uscita, per cui lo si indica con 2^N*N. Nel caso semplice in cui N=3, abbiamo dunque quanto segue:



Il funzionamento, come detto, è l'opposto del decoder: quando viene attivata una linea di ingresso, le uscite forniscono il numero decimale corrispondente a tale linea. Ad esempio, se viene attivata la linea 5, le uscite sono x=1,y=0,z=1, in modo che la parola binaria di uscita sia 101, ossia appunto 5 in binario:



Per capire come realizzare praticamente (cioè in termini di porte logiche) un simile funzionamento, andiamo a tracciare la tabella della verità della funzione (che avrà evidentemente 8 ingressi e 3 uscite):

d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

Per prima cosa, osserviamo che, delle 2^8=256 possibili combinazioni di ingresso, ne vengono usate soltanto 8, lasciando le rimanenti 247 inutilizzate.

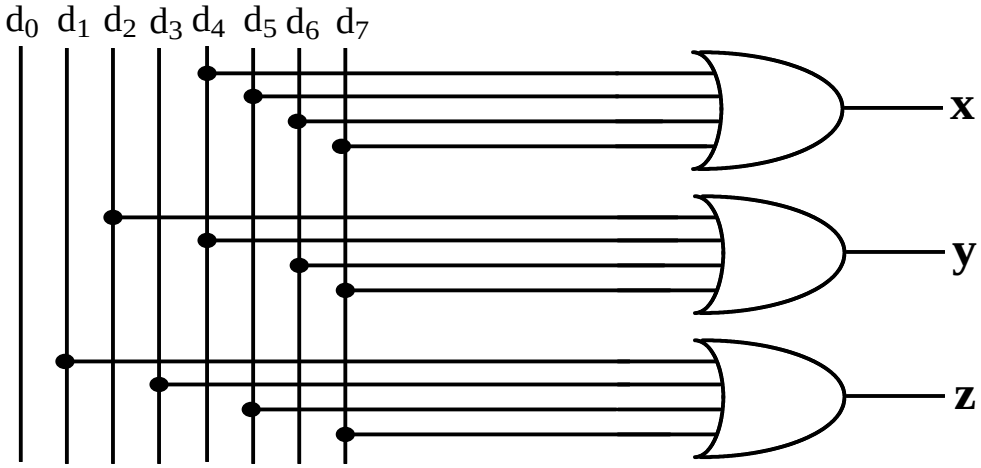
Dalla tabella della verità si osserva anche che ciascuna delle 4 uscite vale 1 in corrispondenza di 4 combinazioni di ingresso:

- l’uscita x vale 1 quando è attivata la linea d_4 oppure la linea d_5 oppure la linea d_6 oppure la linea d_7;
- l’uscita y vale 1 quando è attivata la linea d_2 oppure la linea d_3 oppure la linea d_6 oppure la linea d_7;
- l’uscita x vale 1 quando è attivata la linea d_1 oppure la linea d_3 oppure la linea d_5 oppure la linea d_7.

Allora è evidente che possiamo scrivere quanto segue:

$$x = d_4 + d_5 + d_6 + d_7$$
$$y = d_2 + d_3 + d_6 + d_7$$
$$x = d_1 + d_3 + d_5 + d_7$$

Abbiamo cioè ricavato che, per realizzare un encoder 2_3*3, basta usare 3 porte OR a 4 ingressi:



ENCODER CON PRIORITÀ

Sempre guardando la tabella della verità di un encoder, ci accorgiamo che esso presenta due inconvenienti:

- in primo luogo, osserviamo che, quando è attivata la linea d_0 in ingresso, le uscite sono tutte a 0; lo stesso però accade quando nessuna delle linee di ingresso è attivata; siamo cioè in presenza di una ambiguità: in corrispondenza di una stessa combinazione di uscita, abbiamo due possibili combinazioni di ingresso. In particolare, questa ambiguità è relativa alla linea d_0: non possiamo giudicare se la linea d_0 è attivata o meno, come si nota anche nel circuito logico disegnato poco fa, nel quale la linea d_0 è l'unica che non è collegata ad alcuna porta OR di uscita;
- in secondo luogo, l'encoder presenta dei problemi quando, a causa di un errore nei segnali che gli giungono in ingresso, risulta attivata più di una linea di ingresso: per esempio, supponiamo che vengano contemporaneamente attivate le linee di uscita d_3,d_5 e d_6:

    * se d_3=1, dalla tabella della verità deduciamo che vengono attivate le uscite y e z;
    * se d_5=1, vengono attivate le uscite x e z;
    * se d_6=1, vengono attivate le uscite x e y.

  In complesso, tutte e tre le uscite vengono poste ad 1, dal che si dovrebbe dedurre che è stata attivata la linea 7, che invece è disattivata. Questa è dunque una seconda ambiguità.

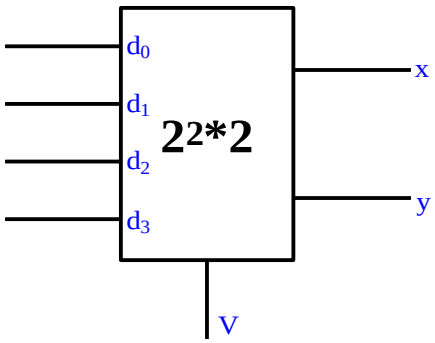
Questi due inconvenienti rendono l'encoder inutilizzabile così com'è. Per aggirare questi inconvenienti, si usano due accorgimenti diversi, che caratterizzano il cosiddetto **encoder con priorità**:

- in primo luogo, si utilizza una ulteriore linea oltre a quelle di ingresso (**linee di indirizzo**) e a quelle di uscita: si tratta della cosiddetta **linea VALIDATE** (simbolo V), che assume valore 1 quando i dati in uscita sono validi e valore 0 in caso contrario;
- in secondo luogo, si attribuisce una priorità diversa alle linee di indirizzo: in particolare, la priorità più bassa è quella della linea d_0 e poi va via via aumentando per le linee d_1,d_2,... fino all'ultima linea d_{N-1}.

Vediamo allora perchè questi due accorgimenti sono utili. Mettiamoci nuovamente nel caso in cui, a seguito di errori, sono state attivate le linee d_3,d_5 e d_6: l'encoder senza priorità, per quanto visto prima, forniva in uscita la configurazione 111 corrispondente alla linea d_7, che invece non era attivata; al contrario, l'encoder con priorità fornisce in uscita la configurazione che corrisponde alla linea, tra quelle effettivamente attivate, con priorità maggiore: nel nostro caso, la linea attivata avente priorità maggiore è la d_6, per cui l'uscita è xyz=110. Così facendo, anche se non si può prescindere dall'errore in ingresso, viene eliminata l'ambiguità.

In base a queste considerazioni, cerchiamo adesso di capire a cosa serve la linea VALIDATE. E' ovvio che questa linea serve, tra le altre cose, a risolvere l'ambiguità circa l'attivazione o meno della linea d_0.

Per semplicità, consideriamo un encoder con N=2, cioè quindi con 2^2=4 linee di indirizzo, 2 linee di uscita e 1 linea validate:



Vogliamo determinare la tabella della verità di questo dispositivo, ossia come variano x,y,V al variare dei valori delle 4 linee di indirizzo:

- il primo caso è quello in cui tutte e 4 le linee sono disattivate: al fine di eliminare l'ambiguità prima sottolineata circa l'attivazione o meno di d_0, si fa in modo che, solo in questa situazione, risulti V=0. Quindi, dire che V=0 equivale a dire che le linee di ingresso sono tutte disattivate e quindi l'uscita xy non ha alcun significato;
- il secondo caso è quello in cui d_0=1 e d_1=d_2=d_3=0: in questo caso non c'è possibilità di errore, per cui V=1 e viene posto xy=00, in modo da indicare che l'unica linea di indirizzo attivata è d_0;
- il terzo caso è quello in cui d_1=1 e d_0=d_2= d_3=0: anche qui non c'è possibilità di errore, per cui V=1 e viene posto xy=01, in modo da indicare che l'unica linea di indirizzo attivata è d_1;
- il quarto caso è quello in cui, a seguito di errori, risulta ad esempio d_0=d_1=1 e d_2=d_3=0: in questo caso, si pone sempre V=1 e, in base al meccanismo della priorità, viene fornita in uscita la combinazione corrispondente a d_1 (cioè la linea attivata avente priorità maggiore), per cui viene posto non c'è possibilità di errore, per cui V=1 e viene posto xy=01. Questa è la stessa uscita generata nel caso precedente, nel senso che abbiamo la stessa uscita in corrispondenza di due diverse combinazioni di ingresso: in particolare, la situazione è quella per cui l'uscita non dipende dal valore di d_0.

Proseguendo in questo modo, si perviene alla seguente tabella della verità:

d_0	d_1	d_2	d_3	V	x	y
0	0	0	0	0	∀	∀
1	0	0	0	1	0	0
0/1	1	0	0	1	0	1
0/1	0/1	1	0	1	1	0
0/1	0/1	0/1	1	1	1	1

Il passo successivo è adesso quello di determinare il circuito logico che implementa questa funzione. In particolare, le funzioni da implementare sono 3 e cioè x,y e V. Per ciascuna di esse, ricorriamo allora alla minimizzazione tramite le mappe di Karnaugh.

Cominciamo dalla funzione x. Dobbiamo individuare tutte le combinazioni di ingresso in corrispondenza delle quali risulta x =1 ; ognuna di queste combinazioni corrisponde ad un mintermine della funzione x ed è facile verificare che si tratta dei mintermini riportati nella seguente tabella della verità:

$d_0d_1 \setminus d_2d_3$	00	01	11	10
00	0000 **X**	0001 **1**	0011 **1**	0010 **1**
01	0100	0101 **1**	0111 **1**	0110 **1**
11	1100	1101 **1**	1111 **1**	1110 **1**
10	1000	1001 **1**	1011 **1**	1010 **1**

funzione x

E' presente chiaramente una condizione don't care in corrispondenza della combinazione di ingresso 0000, in quanto abbiamo detto prima che, in questo caso, risulta V=0, per cui non è importante conoscere il valore delle uscite.

E' evidente che, ai fini della minimizzazione, conviene porre a 0 tale condizione don't care, in modo da avere due subcubi di ordine 3 (formati ciascuno da 8 caselle) che rappresentano due implicanti primi. E' chiaro anche che si tratta di implicanti primi essenziali, in quanto ciascuno copre dei mintermini che non sono coperti dall'altro, per cui la minimizzazione è immediata:

$$x = A + B = (1,3,5,7,9,11,13,15,) + (2,3,6,7,10,11,14,15) = d_3 + d_2$$

La funzione x si realizza dunque semplicemente facendo l'OR tra le linee 2 e 3.

Procediamo in modo analogo per la funzione y, la cui mappa di Karnaugh risulta essere la seguente:

$d_0d_1 \setminus d_2d_3$	00	01	11	10
00	0000 **X**	0001 **1**	0011 **1**	0010
01	0100 **1**	0101 **1**	0111 **1**	0110
11	1100 **1**	1101 **1**	1111 **1**	1110
10	1000	1001 **1**	1011 **1**	1010

funzione y

Anche qui è chiaramente presente la condizione don't care in corrispondenza della combinazione di ingresso 0000. Ponendola uguale a 0 anche in questo caso, si individuano un subcubo di ordine 3 ed un subcubo di ordine 2, che quindi rappresentano due implicanti primi. Si tratta anche di implicanti primi essenziali, per cui

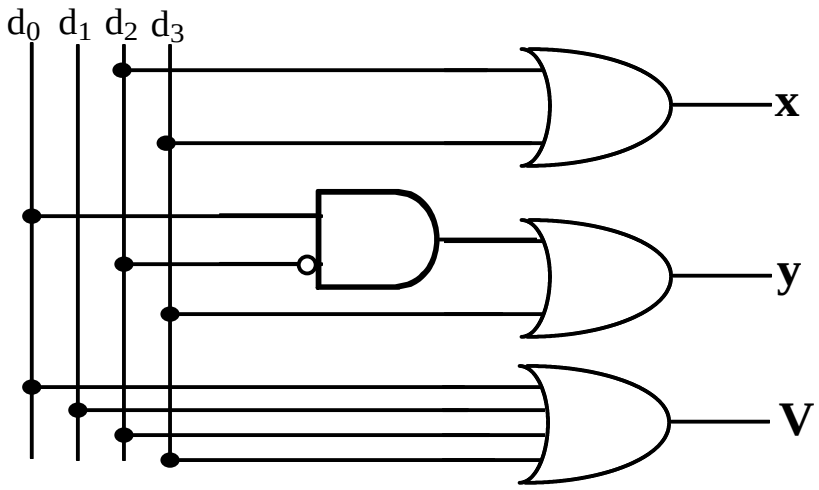
$$x = C + D = (1,3,5,7,9,11,13,15,) + (4,5,12,13) = d_3 + d_1 d'_2$$

La funzione x si realizza dunque semplicemente facendo l'AND tra la linea 1 e la linea 2 complementata e mandando il risultato ad una OR con la linea 3.

Infine, consideriamo la funzione VALIDATE: questa funzione vale 0 solo quando tutte le linee di indirizzo sono a 0, mentre vale 1 quando almeno una di esse è a 0, per cui deduciamo immediatamente che

$$V = d_0 + d_1 + d_2 + d_3$$

Possiamo dunque realizzare il circuito logico che realizza l'**encoder con priorità**:

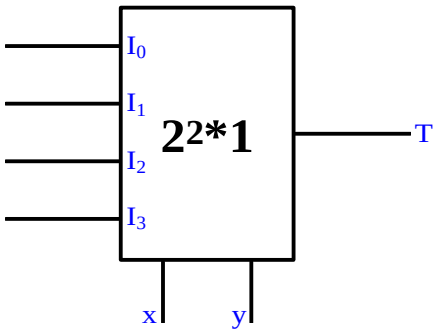


Una semplice osservazione, al fine di semplificare leggermente questo circuito, riguarda la porta OR usata per realizzare la funzione V: infatti, dato che la funzione x rappresenta già l'OR di d_2 e d_3, si potrebbe usare, per la funzione V, una porta OR a 3 ingressi, dove tali ingressi sono x, d_0 e d_1.

MULTIPLEXER

Così come l'encoder rappresenta l'inverso del decoder, il cosiddetto **circuito multiplexer** rappresenta l'inverso del demultiplexer.

In primo luogo, si tratta di un circuito che ha 2^N linee **linee di ingresso**, N **linee di indirizzo** e 1 sola uscita. Nel caso di N=2, il simbolo circuitale è il seguente:



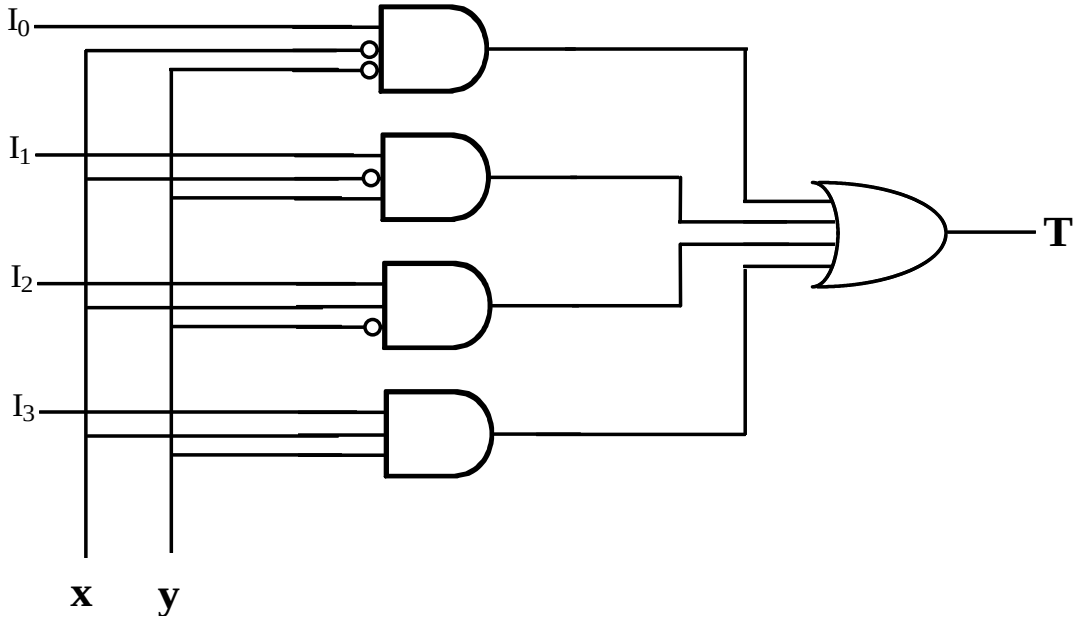
Le linee di indirizzo hanno lo scopo di individuare una precisa linea di ingresso: individuata questa linea, l'uscita T del circuito ne fornisce il valore (1 se è attivata e 0 in caso contrario). Ad esempio, se xy=10, la linea indirizzata è I_2, per cui l'uscita corrisponde al valore di I_2.

La tabella della verità è dunque fatta nel modo seguente:

x	y	T
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

E' molto facile realizzare un circuito logico che implementi questa funzione: il principio di fondo è quello per cui l'uscita si ottiene come OR di 4 linee, ciascuna delle quali corrisponde ad una linea di ingresso e viene portata al valore di tale linea solo se questa è indirizzata.

Il circuito è fatto nel modo seguente:



In base a questo schema, le linee di indirizzo vanno in ingresso, opportunamente complementate, a 4 porte AND, al cui ingresso giungono anche le 4 linee di ingresso. L'unica porta che fornisce in uscita il valore della linea I_k che riceve in ingresso è quella indirizzata dalla coppia xy. E' immediato perciò verificare che questo circuito implementa la tabella della verità della funzione.

E'interessante ricavare una espressione analitica della funzione T direttamente per ispezione del circuito: in primo luogo, la funzione T è data dall'OR delle uscite delle 4 porte AND, per cui scriviamo

$$T = U_0 + U_1 + U_2 + U_3$$

La prima porta AND da risultato 1 solo se I_0=1,x=0 e y=0, per cui possiamo scrivere che $U_0 = I_0 x'y'$. Procedendo in modo analogo per le altre tre porte, ricaviamo che

$$\boxed{T = I_0 x'y' + I_1 x'y + I_2 xy' + I_3 xy}$$

Questa espressione, che in pratica traduce la tabella della verità scritta prima per la funzione T, è di fondamentale importanza per le applicazioni che si fatto del multiplexer: infatti, è evidente che, fissata una qualsiasi combinazione di x ed y, quindi fissata una linea I_k tra quelle di ingresso, il valore che la funzione assume in corrispondenza di tale combinazione dipende proprio da I_k. Questo si può usare per realizzare, con il multiplexer, una qualsiasi funzione booleana di 2 variabili booleane. Per esempio, supponiamo di voler realizzare la funzione EXOR, la cui tabella della verità è la seguente:

x	y	$T = x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

Come realizziamo questa funzione? Vediamo caso per caso:

- quando x=0 e y=0, la linea indirizzata è I_0, che quindi deve valere 0 così come indicato dalla tavola della verità di $T = x \oplus y$;
- quando x=0 e y=1, la linea indirizzata è I_1, che quindi deve valere 1 così come indicato dalla tavola della verità di $T = x \oplus y$;
- quando x=1 e y=0, la linea indirizzata è I_2, che quindi deve valere 1;
- infine, quando x=1 e y=1, la linea indirizzata è I_3, che quindi deve valere 0.

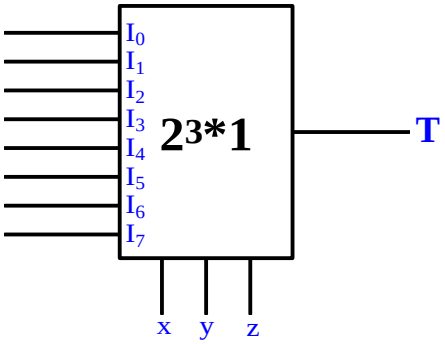
Fissando dunque il valore delle linee di ingresso come indicato, possiamo realizzare la funzione EXOR:

x	y	$T = x \oplus y$
0	0	$I_0 = 0$
0	1	$I_1 = 1$
1	0	$I_2 = 1$
1	1	$I_3 = 0$

Con questo procedimento, *possiamo dunque implementare, con il multiplexer 2^2*1, una qualsiasi funzione booleana di 2 variabili booleane:* basta fissare il valore delle linee di ingresso in accordo a quanto indicato dalla tabella della verità della funzione considerata.

Il meccanismo è assolutamente generale: *con un multiplexer 2_N*1, possiamo implementare una qualsiasi funzione booleana di N variabili booleane.*

Consideriamo, per esempio, N=3:



Con questo circuito possiamo implementare una qualsiasi funzione booleana di 3 variabili booleane, in base alla seguente tavola della verità:

x	y	z	T
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

In realtà, si può fare qualcosa di più. Infatti, *con un multiplexer 2^N*1 si possono realizzare sia funzioni booleane di N variabili booleane, nel modo visto poco fa, sia anche funzioni di N+1 variabili booleane.*

Esempio

Facciamo subito un esempio concreto per chiarire quanto appena detto. Vogliamo realizzare una generica funzione booleana di 3 variabili booleane. Per quanto detto poco fa, possiamo sicuramente usare un multiplexer 2^3*1, ma è facile verificare che si possono anche usare due multiplexer 2^2*1 opportunamente connessi tra loro.

Il principio di fondo da adottare è il seguente: una generica funzione T(x,y,z) di 3 variabili booleane comprende i seguenti mintermini:

x	y	z	T	
0	0	0	x'y'z'	(0)
0	0	1	x'y'z	(1)
0	1	0	x'yz'	(2)
0	1	1	x'yz	(3)
1	0	0	xy'z'	(4)
1	0	1	xy'z	(5)
1	1	0	xyz'	(6)
1	1	1	xyz	(7)

I mintermini 1,3,5,7 hanno in comune la variabile z, mentre i rimanenti mintermini hanno in comune la variabile z': se, allora, mettiamo in evidenza la z, otteniamo quanto segue:

x	y	T
0	0	x'y'
0	1	x'y
1	0	xy'
1	1	xy

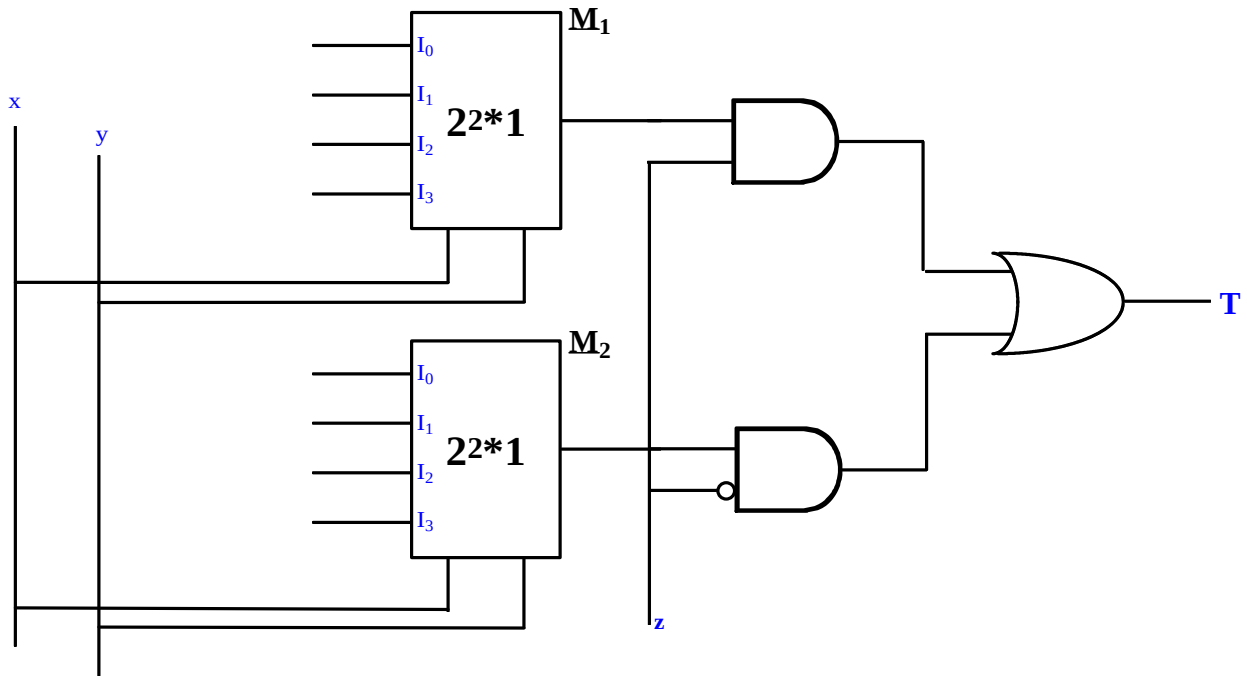
$z = 0$

x	y	T
0	0	x'y'
0	1	x'y
1	0	xy'
1	1	xy

$z = 1$

Allora, per considerare tutti i mintermini della funzione, possiamo prima realizzare due funzioni di 2 variabili (x ed y), aventi la stessa tabella della verità, poi fare un AND tra ciascuna funzione è la z (in un caso complementata e nell'altro no) ed infine fare un OR dei due risultati finali.

Naturalmente, le due funzioni di 2 variabili booleane possono essere implementate ciascuna mediante un multiplexer 2^2*1, come nel circuito seguente:



Impostando il valore delle linee di ingresso dei due multiplexer, possiamo realizzare una qualsiasi tra le 256 possibili funzioni booleane di 4 variabili booleane. Per esempio, supponiamo di voler realizzare la funzione T avente la seguente tabella della verità:

x	y	z	T
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Come impostiamo le linee? Consideriamo nuovamente le 2 tabelle in cui è possibile dividere la tabella completa della funzione:

x	y	T
0	0	x'y'
0	1	x'y
1	0	\ \
1	1	\ \

z = 0

x	y	T
0	0	x'y'
0	1	\ \
1	0	\ \
1	1	xy

z = 1

Abbiamo ovviamente escluso i mintermini non posseduti dalla funzione. Per implementare queste due funzioni mediante i due multiplexer, dovremmo impostare le linee nel modo seguente:

x	y	T
0	0	$I_0 = 1$
0	1	$I_1 = 1$
1	0	$I_2 = 0$
1	1	$I_3 = 0$

multiplexer 2

x	y	T
0	0	$I_0 = 1$
0	1	$I_1 = 0$
1	0	$I_2 = 0$
1	1	$I_3 = 1$

multiplexer 1

Esempio

Nell'esempio precedente abbiamo visto come implementare una funzione booleana di 3 variabili booleana mediante due multiplexer 2^2*1. Vediamo adesso come realizzare una funzione di 4 variabili booleane mediante un solo multiplexer 2^3*1.

Supponiamo che la funzione in esame abbia la seguente prima forma canonica:

$$T(a,b,c,d) = \sum(0,1,4,7,9,10,11,14)$$

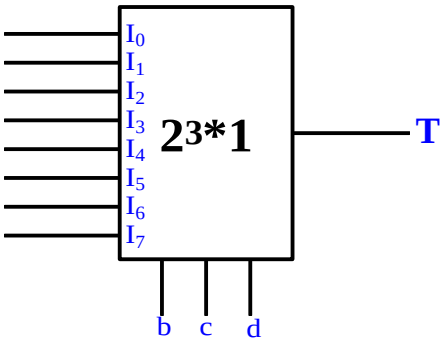
Consideriamo, di questa funzione, una rappresentazione tabellare analoga alla mappa di Karnaugh:

a \ bcd	000	001	010	011	100	101	110	111
0	1	1			1			1
1		1	1	1			1	

In questa tabella, così come nella mappa di Karnaugh, sono indicati i mintermini della funzione, con la differenza, però, di non ordinare tali mintermini in base alle adiacenze: in altre parole, due mintermini adiacenti non necessariamente differiscono per una sola variabile. Il criterio con cui vengono sistemate le varie caselle (cioè appunto i vari mintermini) è quello di porre in orizzontale le caselle corrispondenti alle varie combinazioni delle variabili meno significative b,c e d e di porre in verticale la variabile più significativa.

Con un procedimento di questo tipo, otteniamo 16 caselle (corrispondenti ai 16 mintermini), ognuna delle quali è individuata da un indirizzo, rappresentato dai valori della sequenza bcd, e da un valore della variabile a: per esempio, il mintermine numero 0 è individuato dall'indirizzo 000 e dal valore a=0; in modo analogo, il mintermine 10 è individuato dall'indirizzo 001 e dal valore a=1 e così via per gli altri mintermini.

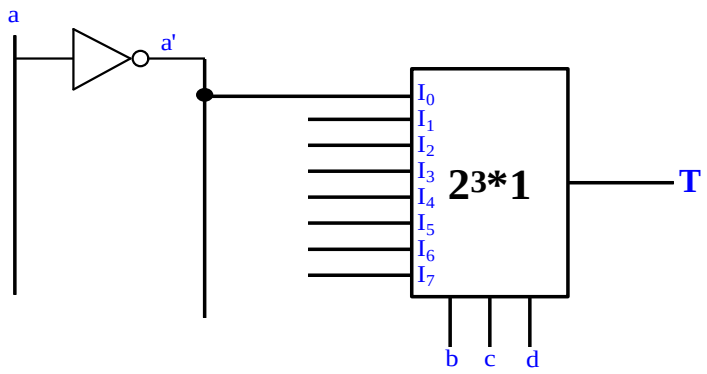
A questo punto, se abbiamo 8 indirizzi, possiamo associarli alle 8 linee di un multiplexer 2^3*1, avente proprio b,c,d come ingressi:



Di conseguenza, nella tabella di prima, al posto degli indirizzi binari possiamo direttamente indicare le corrispondenti linee di ingresso del multiplexer:

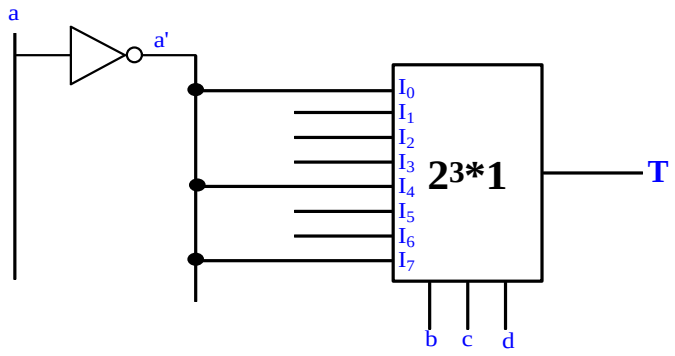
a \ bcd	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7
0	1	1			1			1
1		1	1	1			1	

Supponiamo, allora, che l'ingresso al multiplexer sia la combinazione b=0,c=0,d=0, per cui la linea indirizzata è la I_0, nel senso che l'uscita del multiplexer fornisce proprio il valore di I_0. Possiamo a questo punto imporre che la linea I_0, e quindi l'uscita del dispositivo, assuma valore 1 se a=0 oppure valore 0 se a=1. E' chiaro che, per ottenere questo, ci basta porre I_0=a':

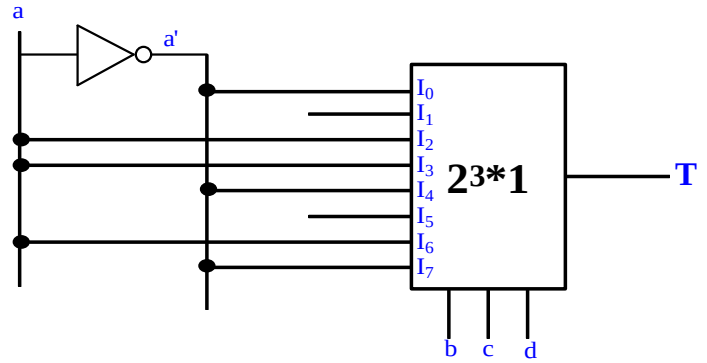


Con questo sistema abbiamo garantito che l'uscita del multiplexer valga 1 nel caso in cui la combinazione degli ingressi sia $(a,b,c,d)=(0,0,0,0)$, ossia abbiamo garantito il primo mintermine della funzione.

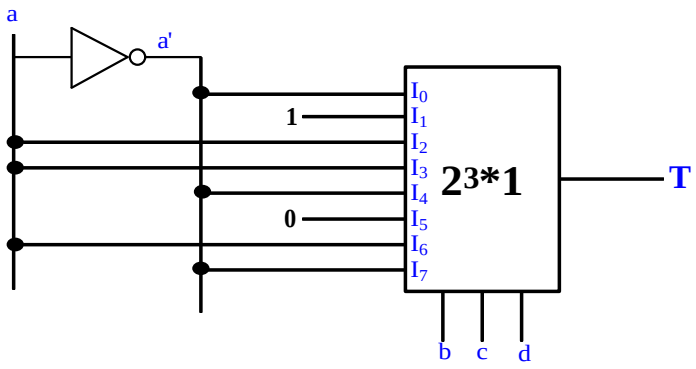
In modo del tutto analogo possiamo procedere per la linea I_4 e per la linea I_7:



Se invece consideriamo le linee I_2 , I_3 ed I_6, ci accorgiamo che dobbiamo fare il contrario, nel senso che dobbiamo imporre che, una volta indirizzate, esse diano 0 quando a=0 e 1 quando a=1. Basta allora porle uguali proprio ad a:



Due casi particolari sono invece le rimanenti linee I_1 ed I_5: la prima deve dare 0 a prescindere dal valore di a, mentre la seconda deve dare sempre 1. Possiamo perciò concludere che la nostra funzione è implementata dal seguente circuito:



Il principio è dunque il seguente: le variabili che danno le cifre meno significative dei mintermini vanno poste in ingresso, in modo da indirizzare le linee del multiplexer; la rimanente variabile viene usata per imporre che la linea di volta in volta indirizzata assuma il valore desiderato (che sarà 1 se la funzione possiede il corrispondente mintermine oppure 0 in caso contrario).

Facciamo qualche semplice verifica:

- consideriamo la configurazione di ingresso $(a,b,c,d)=(0,0,0,1)$, corrispondente al mintermine 1: in base alla forma canonica di partenza, la funzione contiene questo mintermine, per cui deve assumere valore 1 in corrispondenza di questa configurazione. Le cifre meno significative indicano l'indirizzo 001, per cui la linea indirizzata è I_1: questa linea vale 1 a prescindere dal valore di a, per cui l'uscita è 1, come volevamo;
- consideriamo adesso la configurazione di ingresso $(a,b,c,d)=(1,1,1,1)$, corrispondente al mintermine 15: in base alla forma canonica di partenza, la funzione non contiene questo mintermine, per cui l'uscita del circuito deve essere 0. Le cifre meno significative indicano l'indirizzo 111, per cui la linea indirizzata è I_7: questa linea vale a', per cui, essendo a=1, l'uscita del circuito è 0, come volevamo.

Proseguendo in questo modo, si verifica facilmente che il circuito implementa la funzione considerata.

Questo è dunque un altro modo, alternativo a quello indicato nell'esempio precedente, di implementare, senza minimizzarla, una funzione di N+1 variabili mediante un multiplexer 2^N*1.

Esempio

Facciamo un altro esempio analogo al precedente. Supponiamo che la funzione da implementare abbia la seguente prima forma canonica:

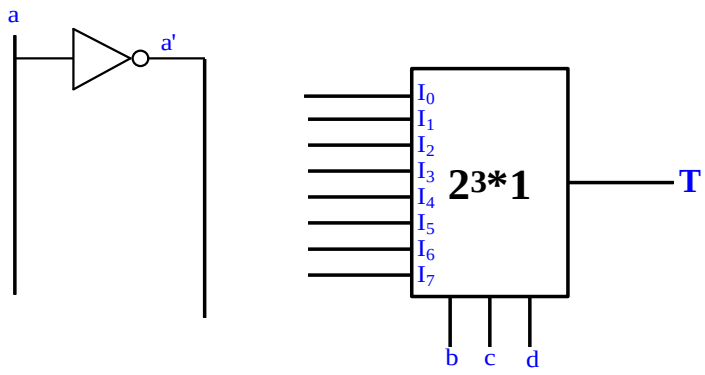
$$T(a,b,c,d)=\sum(0,3,5,6,9,10,12,15)$$

In generale, dovendo implementare una funzione, si cerca di farlo nel modo più economico possibile, cioè con meno porte logiche possibile. Di conseguenza, è sempre opportuno verificare se essa possa essere semplificata. Questo è un tipico caso in cui non è possibile fare alcuna semplificazione, in quanto la mappa di Karnaugh della funzione è fatta nel modo seguente:

ab \ cd	00	01	11	10
00	0000 **1**	0001	0011 **1**	0010
01	0100	0101 **1**	0111	0110 **1**
11	1100 **1**	1101	1111 **1**	1110
10	1000	1001 **1**	1011	1010 **1**

E’ una mappa, per così dire, a scacchiera, priva cioè di adiacenze e quindi di implicanti primi.

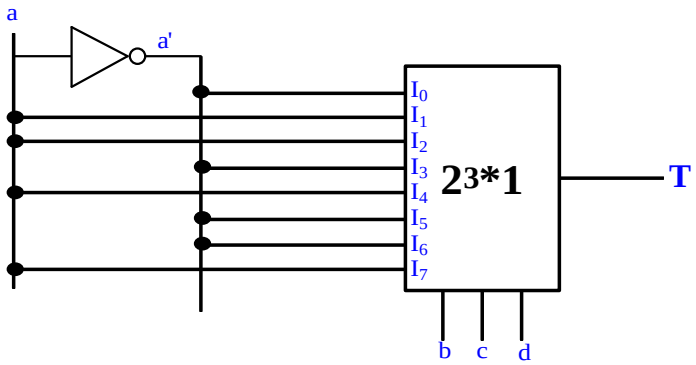
Sappiamo, allora, che possiamo realizzare questa funzione di 4 variabili mediante un multiplexer 8*1. Si tratta solo di capire come impostare le linee di ingresso del multiplexer:



Andiamo allora a rappresentare i mintermini della funzione mediante una tabella del tutto analoga a quella vista nell’esempio precedente:

a \ bcd	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7
0	1			1		1	1	
1		1	1		1			1

In base a questa tabella, lo schema da adottare è il seguente:



Esempio

Adesso ci poniamo un problema leggermente diverso rispetto a quello dei due esempi precedenti. Supponiamo di avere una funzione di 7 variabili booleane, avente la seguente forma canonica:

$$T(a,b,c,d,e,f,g) = \sum(0,1,4,6,8,15,21,25,34,37,51,57,65,81,89,90,101,102,104,106,109,118,121)$$

Avendo 7 variabili, potremmo utilizzare, per la implementazione di questa funzione, un multiplexer 2^6*1 variabili: esso avrebbe 2^6=64 linee, indirizzate dalla combinazione (b,c,d,e,f,g), ognuna delle quali corrisponderebbe a 2 mintermini della funzione, a seconda che la variabile più significativa a valga 0 o 1. Potremmo perciò procedere come negli esempi precedenti.

Supponiamo, invece, di non avere un multiplexer 2^6*1, ma di poter disporre, al più, di multiplexer 16*1 (in numero qualsiasi). Ci chiediamo se possiamo ugualmente realizzare la funzione.

Il modo più semplice di procedere, in questo caso, è il seguente.

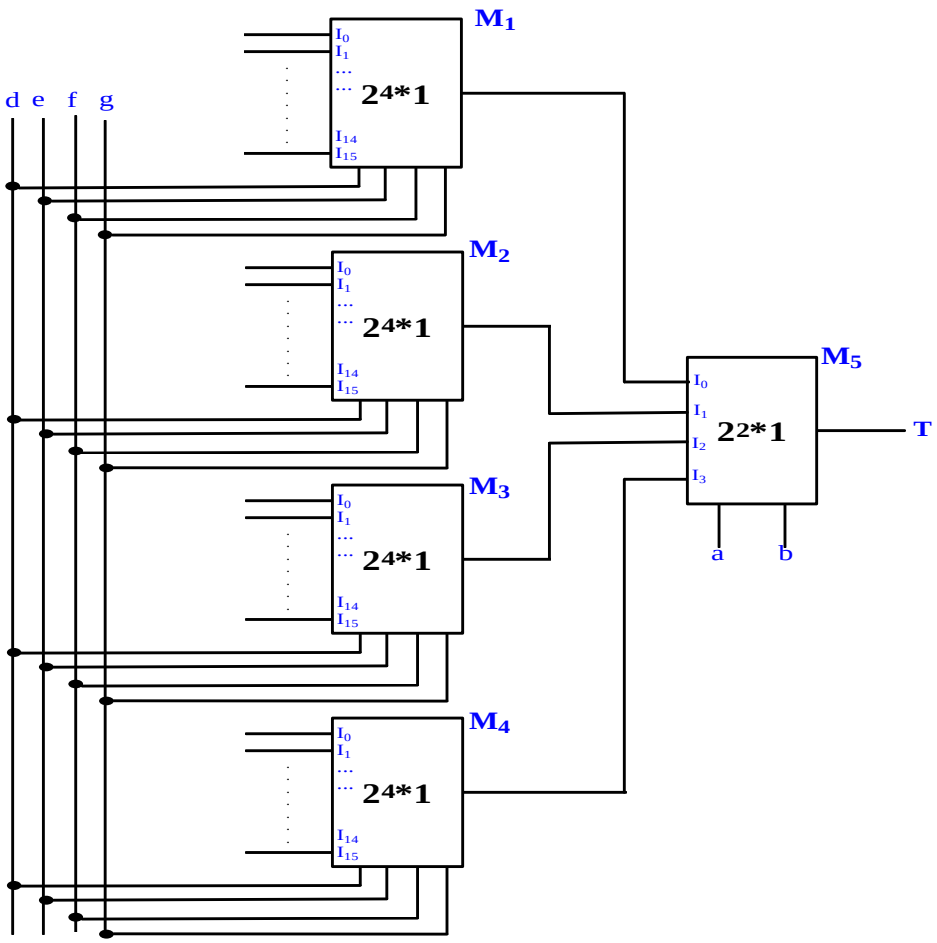
Una funzione, come quella considerata, di 7 variabili presenta 2^7=128 mintermini, ognuno caratterizzato da una combinazione di ingresso, ossia quindi da un indirizzo (da 0 a 127). Se consideriamo tutti questi 128 indirizzi, ci rendiamo conto che possiamo raggrupparli in 4 gruppi, caratterizzati dalle seguenti coppie a,b: 32 mintermini con indirizzo che comincia con 00, 32 mintermini con indirizzo che comincia con 01, 32 mintermini con indirizzo che comincia con 10 e 32 mintermini con indirizzo che comincia con 11.

Possiamo allora elencare i mintermini, indicando quelli effettivamente posseduti dalla funzione, in una tabella del tipo seguente:

abc\defg	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
000	0 **1**	1 **1**			4 **1**		6 **1**		8 **1**							15 **1**
001						21 **1**				25 **1**						
010			34 **1**			37 **1**										
011				51 **1**						57 **1**						
100		65 **1**														
101		81 **1**								89 **1**	90 **1**					
110						101 **1**	102 **1**		104 **1**		106 **1**			109 **1**		
111							118 **1**			121 **1**						

Abbiamo 16 colonne, ognuna contraddistinta da una combinazione delle variabili meno significative d,e,f,g, e poi 8 righe, ognuna contraddistinta da una combinazione delle variabili più significative a,b,c. Queste righe hanno in comune, a coppie, i valori di a e b: quindi, ciascuna delle configurazioni ab individua 2 righe, ossia 32 mintermini, come detto prima. Ciascuno di questi 4 gruppi di 32 mintermini contiene, a sua volta, una riga (di 16 mintermini) corrispondente a c=0 ed un'altra riga corrispondente a c=1.

Allora, questo ci suggerisce di adottare uno *schema a 2 livelli* del tipo seguente:



Il primo livello è costituito da 4 multiplexer 16*1, ciascuno dotato di 4 linee di indirizzo e 16 linee di ingresso; le uscite di questi 4 multiplexer costituiscono le linee di ingresso di un multiplexer 4*1, le cui linee di indirizzi sono solo 2.

Il multiplexer di uscita, le cui linee di indirizzo corrispondono alle variabili a e b, serve a indirizzare uno dei 4 multiplexer di ingresso. I multiplexer di ingresso hanno, tutti, le linee di indirizzo corrispondenti alle variabili d,e,f,g. Così facendo, ci siamo quindi ricondotti allo stesso problema degli esempi precedenti, in quanto basta impostare le 16 linee di ingresso di ciascuno dei multiplexer di ingresso per ottenere i mintermini della funzione considerata.

Consideriamo, per esempio, il multiplexer M1, corrispondente ai 16 mintermini aventi, come primi due bit di indirizzo, la configurazione a=0,b=0: si tratterà, perciò, dei mintermini compresi tra 0 e 15. Usiamo una tabella analoga a quella degli esempi precedenti:

c\defg	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	**1**	**1**			**1**		**1**		**1**							**1**
1						**1**				**1**						

Se agli indirizzi binari riportati in ascisse sostituiamo le corrispondenti linee, abbiamo quanto segue:

c \ defg	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}	I_{11}	I_{12}	I_{13}	I_{14}	I_{15}
0	**1**	**1**			**1**		**1**		**1**							**1**
1						**1**				**1**						

Così facendo, siamo in grado di dire come devono essere impostate le 16 linee di M_1. Per esempio, per M_2 abbiamo quanto segue:

c \ defg	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}	I_{11}	I_{12}	I_{13}	I_{14}	I_{15}
0			**1**			**1**										
1				**1**						**1**						

In modo del tutto analogo possiamo procedere per gli altri 2 multiplexer, per cui riassumiamo i risultati nella tabella seguente:

	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}	I_{11}	I_{12}	I_{13}	I_{14}	I_{15}
M_1	c'	c'	0	0	c'	c	c'	0	c'	c	0	0	0	0	0	c'
M_2	0	0	c'	c	0	c'	0	0	0	c	0	0	0	0	0	0
M_3	0	1	0	0	0	0	0	0	0	c	c	0	0	0	0	0
M_4	0	0	0	0	0	c'	1	0	c'	c	c'	0	0	c'	0	0

Sulla base di questa tabella possiamo dunque effettuare le connessioni delle 64 linee di ingresso dei multiplexer 16*1 in modo da ottenere la funzione desiderata.

ROM: READ ONLY MEMORY

I multiplexer, da quanto visto nei paragrafi precedenti, sono molto utili per realizzare funzioni di più variabili ma ad una sola uscita. Nel caso in cui, invece, si debbano realizzare **circuiti ad uscite multiple**, ossia circuiti che implementano più funzioni degli stessi argomenti, è necessario utilizzare componenti più complicati (dal punto di vista del numero di porte logiche contenute all'interno).

Tipico caso è quello dei **circuiti ROM**. L'acronimo ROM sta per Read Only Memory, ossia memoria a sola lettura; questa espressione evidenzia due importanti caratteristiche di questi circuiti: si tratta di *memorie* e, in particolare, di *memorie non riscrivibili*, nel senso che, una volta stabilito il loro contenuto, esso non può più essere modificato. In aggiunta a questo, si tratta di *memorie non volatili*: al contrario delle memorie RAM, infatti, il contenuto informativo di una ROM non viene perso nel momento in cui la ROM stessa viene disenergizzata (cioè viene staccata l'alimentazione).

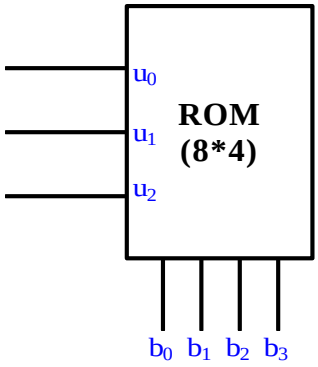
Abbiamo parlato di *memorie ROM*: in realtà, questa dicitura sembrerebbe presupporre che una ROM sia costituita da un certo numero di caselle di memoria, ciascuna destinata a contenere un 1 logico oppure uno 0 logico. In realtà non è così, in quanto *una ROM è visualizzabile come una memoria, ma non è un insieme di caselle di memoria; si tratta, invece, di un normale circuito logico, con alcune caratteristiche particolari.* Cerchiamo allora di capire quali siano queste caratteristiche.

Cominciamo col visualizzare una ROM come una memoria: essa sarà allora dotata di N locazioni di memoria (dove N è una potenza di 2), ciascuna formata da M bit (dove M non necessariamente è una potenza di 2). Per esempio, consideriamo una ROM di 8 locazioni di memoria, ciascuna costituita da 4 celle (cioè da un mezzo byte):

	b_0	b_1	b_2	b_3
#0				
#1				
#2				
#3				
#4				
#5				
#6				
#7				

Ciascuna locazione ha un indirizzo (da #0 a #7) e contiene i 4 bit b_0,b_1,b_2,b_3. Il fatto che il numero di locazioni sia una potenza di 2 comporta che sia sufficiente un decoder (il cui numero di uscite è a sua volta una potenza di 2) per realizzare l’indirizzamento.

Una volta effettuato l’indirizzamento della ROM, ossia la scelta di una particolare locazione di memoria, la ROM deve fornire in uscita il contenuto dei bit corrispondenti a tale locazione di memoria. Possiamo dunque schematizzare una ROM, con indirizzamento a 3 bit (quindi 2^3=8 locazioni) e con M=4 bit per ogni locazione, nel modo seguente:



Per ottenere questo risultato, si fa la seguente considerazione: dato che il contenuto della ROM va fissato una volta sola, è ovvio che ad ogni combinazione di ingresso (u_0,u_1,u_2) corrisponderà sempre la stessa combinazione di uscita b_0,b_1,b_2,b_3. Questo equivale a fissare 4 funzioni booleane, i cui argomenti sono dati dalle linee di ingresso della ROM stessa:

$$b_0 = f_0(u_0,u_1,u_2)$$
$$b_1 = f_1(u_0,u_1,u_2)$$
$$b_2 = f_2(u_0,u_1,u_2)$$
$$b_3 = f_3(u_0,u_1,u_2)$$

Per esempio, supponiamo che il contenuto della ROM sia quello indicato nella tabella seguente:

	b_0	b_1	b_2	b_3
#0	1	1	0	0
#1	1	0	0	0
#2	0	0	0	0
#3	0	1	1	0
#4	0	0	1	0
#5	0	0	0	0
#6	1	0	0	0
#7	0	0	1	1

Affinchè l'uscita b_0,b_1,b_2,b_3 sia quella richiesta, dobbiamo semplicemente realizzare un circuito che implementi le seguenti 4 funzioni:

u_0	u_1	u_2	b_0	b_1	b_2	b_3
0	0	0	1	1	0	0
0	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	1	0	1	1	0
1	0	0	0	0	1	0
1	0	1	0	0	0	0
1	1	0	1	0	0	0
1	1	1	0	0	1	1

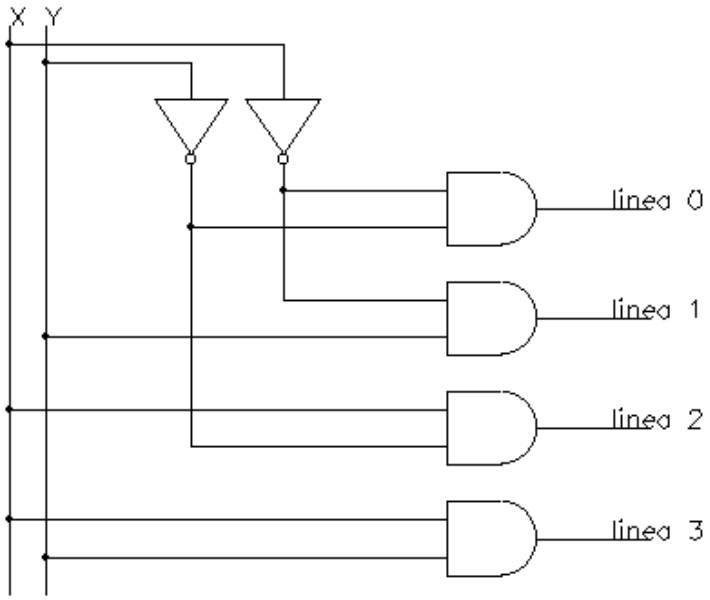
Da qui, dunque, si capisce il motivo per cui una ROM può essere realizzata mediante un circuito: in corrispondenza di ciascuna combinazione di ingresso, cioè di ciascun indirizzo, le singole funzioni devono fornire il valore corrispondente alle celle appartenenti alla locazione di memoria indirizzata.

Il passo successivo consiste nel capire due cose: come realizzare il circuito che implementa una ROM e come effettuare la programmazione della ROM stessa.

Il problema di come realizzare il circuito può essere facilmente impostato: abbiamo un numero M di funzioni, ciascuna con N variabili di ingresso; con N variabili di ingresso possiamo formare 2^{2^N} funzioni diverse[1]; tali funzioni si differenziano per i mintermini compresi nelle rispettive forme canoniche: i mintermini, in totale, sono 2^N, per cui, se vogliamo garantirci di poter realizzare tutte le 2^{2^N} funzioni, dobbiamo realizzare un circuito che implementi tutti i 2^N mintermini.

Ma noi sappiamo già come realizzare una cosa del genere, in quanto l'abbiamo già analizzata a proposito del decoder. Nel caso di N=2, lo schema logico di un decoder era il seguente:

[1] Nonostante questo, il numero M di celle per ogni locazione di memoria è indipendente dal numero N di ingressi di indirizzo.

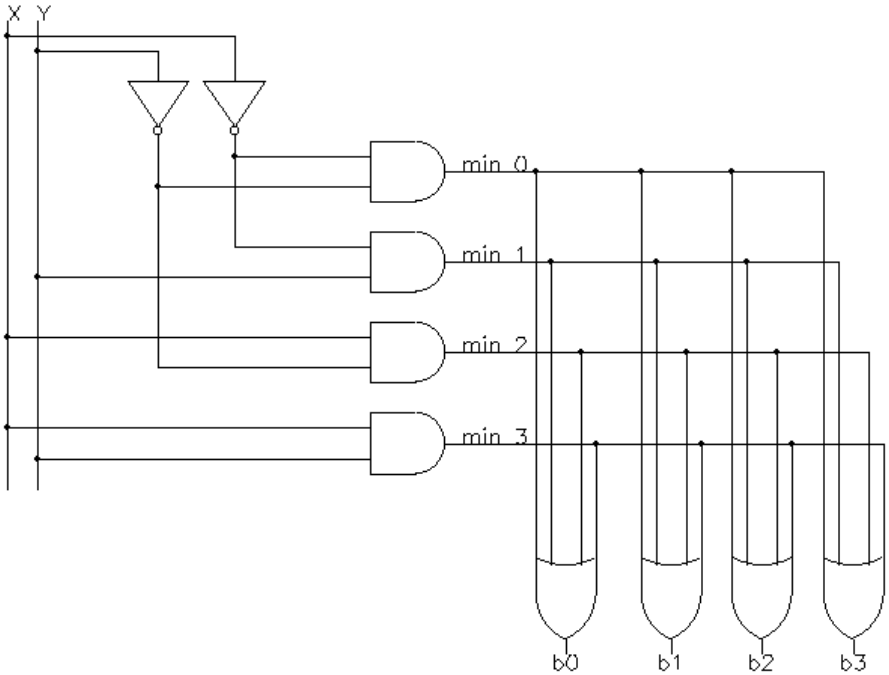


Abbiamo detto che ogni porta AND costituisce un mintermine della funzione, nel senso che la sua uscita vale 1 solo per la combinazione di ingresso corrispondente al suddetto mintermine:

$$\text{linea } 0 \longrightarrow \text{min} = x'y'$$
$$\text{linea } 1 \longrightarrow \text{min} = x'y$$
$$\text{linea } 2 \longrightarrow \text{min} = xy'$$
$$\text{linea } 3 \longrightarrow \text{min} = xy$$

Di conseguenza, se sommiamo le uscite delle porte AND mediante una porta OR, otteniamo la somma di tutti i mintermini.

Allora, se vogliamo realizzare M funzioni, dovremo disporre di M porte OR, a ciascuna delle quali porteremo in ingresso le uscite delle 2^N porte AND. Nel caso, per esempio, di 2^2=4 locazioni, ossia N=2 ingressi alla ROM, con ciascuna locazione formata da M=4 bit, avremo uno schema del tipo seguente:



Ovviamente, così facendo otteniamo 4 funzioni dotate di tutti i mintermini, il che corrisponde a funzioni aventi valore 1 per qualsiasi combinazione di ingresso. Al contrario, ogni funzione avrà solo un certo numero di mintermini, in base alla sua tabella della verità. Ciò significa che non tutti i conduttori che "trasportano" i mintermini vanno portati in ingresso alle OR finali: ciascuna OR riceverà in ingresso tanti conduttori quanti sono i mintermini posseduti dalla corrispondente funzione.

La scelta di quali uscite delle porte AND mandare in ingresso alle porte OR consiste appunto nella programmazione della memoria ROM. La si effettua collegando i conduttori che vanno in ingresso alle OR con dei fusibili: bruciando un fusibile, si fa in modo che il conduttore non porti alcun ingresso alla porta OR dove è diretto.

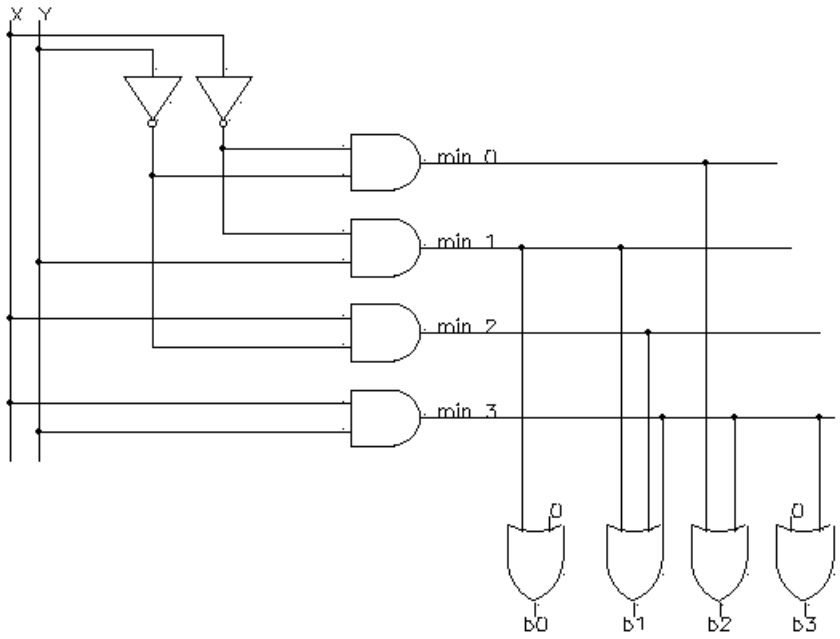
Facciamo un esempio, con riferimento sempre ad una ROM a 4 locazioni, ciascuna di 4 bit. Supponiamo che il contenuto della ROM debba essere il seguente:

	b_0	b_1	b_2	b_3
#0	0	0	1	0
#1	1	1	0	0
#2	0	1	0	0
#3	0	1	1	1

Dobbiamo dunque realizzare 4 funzioni:

u_0	u_1	b_0	b_1	b_2	b_3
0	0	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	0
1	1	0	1	1	1

La funzione b_0 presenta solo il mintermine 1, quindi manderemo alla corrispondente porta OR solo il corrispondente reoforo, ponendo gli altri ingressi a 0; la funzione b1 presenta invece i mintermini 1,2 e 3, per cui manderemo in ingresso i 3 corrispondenti reofori, ponendo a 0 il rimanente ingresso. Procedendo in modo analogo per le altre due porte OR, otteniamo il seguente schema logico:



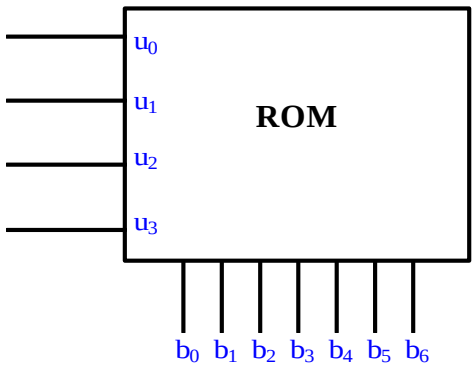
Esempio

Abbiamo dunque capito, nel paragrafo precedente, che la memoria ROM non è altro che un circuito logico in cui, bruciando gli opportuni fusibili, si simula, in pratica, la programmazione di una memoria. Dato che questo circuito consente di sommare tutti i possibili mintermini di una funzione di N variabili, dove N sono le linee di indirizzo della ROM, è ovvio che quest'ultima può essere utilizzare semplicemente per realizzare una o più funzioni booleane, così come abbiamo visto per tutti i circuiti precedenti.

Facciamo allora un esempio di questo tipo di applicazione. Vogliamo realizzare un circuito che riceva in ingresso un numero binario in formato BCD e fornisca in uscita il valore binario del suo quadrato.

Dobbiamo per prima cosa scegliere i parametri della ROM, vale a dire numero di ingressi e numero di uscite: se in ingresso dobbiamo inviare sequenze binarie in formato BCD, avremo bisogno di N=4 ingressi; in uscita, dato che il massimo numero ottenibile è 81 (cioè il quadrato di 9), dobbiamo utilizzare 7 linee, dato che 6 linee basterebbero a rappresentare solo il numero 64, mentre con 7 possiamo arrivare fino a 128.

La nostra ROM sarà dunque del tipo seguente:



Dobbiamo dunque realizzare 7 funzioni, le cui tabelle della verità sono le seguenti:

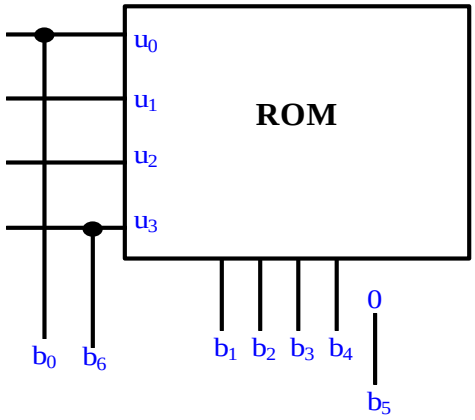
u_0	u_1	u_2	u_3	b_0	b_1	b_2	b_3	b_4	b_5	b_6	
0	0	0	0	0	0	0	0	0	0	0	(0)
0	0	0	1	0	0	0	0	0	0	1	(1)
0	0	1	0	0	0	0	0	1	0	0	(4)
0	0	1	1	0	0	0	1	0	0	1	(9)
0	1	0	0	0	0	1	0	0	0	0	(16)
0	1	0	1	0	0	1	1	0	0	1	(25)
0	1	1	0	0	1	0	0	1	0	0	(36)
0	1	1	1	0	1	1	0	0	0	1	(49)
1	0	0	0	1	0	0	0	0	0	0	(64)
1	0	0	1	1	0	1	0	0	0	1	(81)

Alcune di queste funzioni sono molto facilmente implementabili. Ad esempio, si osserva che la funzione b_5 da sempre valore 0: potremmo semplicemente porre a 0 gli ingressi della porta OR della ROM corrispondente a b_5, ma possiamo più semplicemente prendere una linea posta perennemente a 0, in modo da considerare una ROM con 6 uscite anzichè 7.

In modo analogo, si osserva che la funzione b_6 assume gli stessi valori dell'ingresso u_3, per cui anche in questo caso non è necessario utilizzare la ROM, mentre è sufficiente porre un collegamento tra il pin di u_3 e quello di b_6.

Infine, si osserva che la funzione b_0 coincide con l'ingresso u_0, per cui ci basterà anche in questo caso un semplice reoforo.

Possiamo dunque disegnare lo schema nel modo seguente:

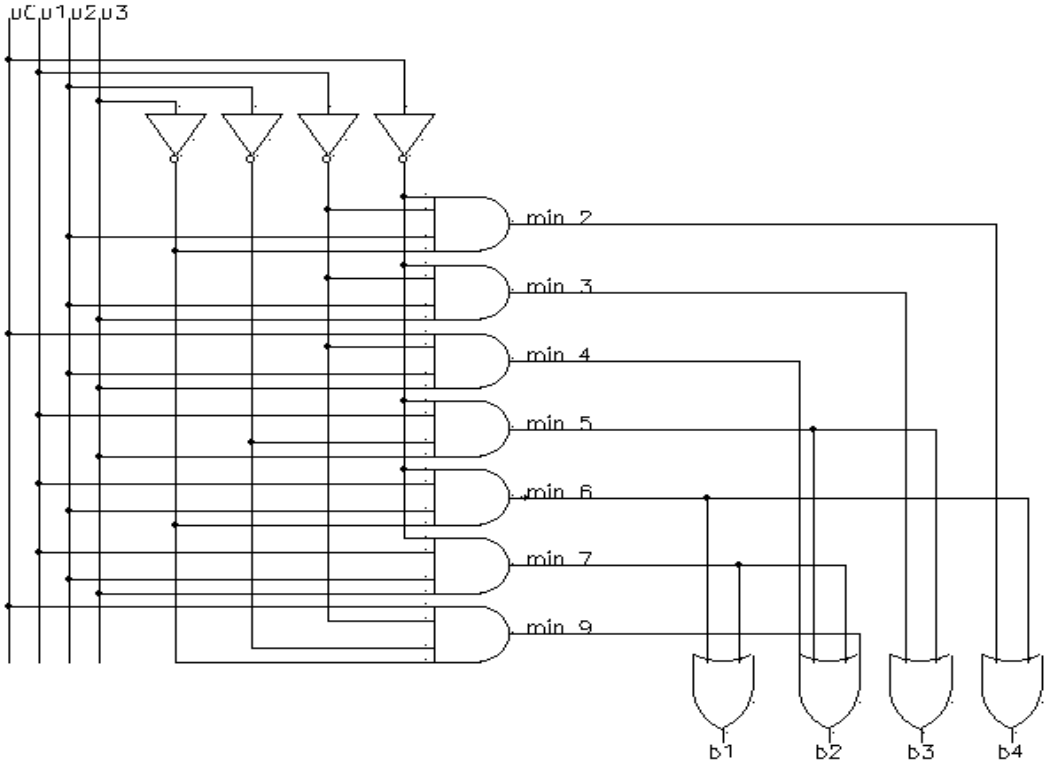


A questo punto, non essendo possibili altre semplificazioni, siamo in grado di stabilire che tipo di ROM ci serve inpratica: dovrà avere 4 ingressi (quindi 2^4=16 locazioni, ossia 16 porte AND) e 4 uscite (ossia 4 celle per ogni locazione, ossia 4 porte OR finali).

Il passo successivo è quello di effettuare la programmazione. A tal fine, dobbiamo individuare i mintermini delle funzioni da implementare: dalla tabella di verità disegnata prima, abbiamo quanto segue:

$$b_1 = \sum(6,7) \qquad b_2 = \sum(4,5,7,9) \qquad b_3 = \sum(3,5) \qquad b_4 = \sum(2,6)$$

In totale, quindi, dei 16 possibili mintermini, ne vengono usati solo 7. Lo schema logico sarà il seguente (dove riportiamo solo i 7 mintermini utilizzati, cioè le corrispondenti porte AND, ricordando però che la ROM contiene comunque tutte e 16 le porte AND corrispondenti ai 16 mintermini possibili):



Osserviamo che, nel circuito appena realizzato, l'uscita rappresenta un numero binario, a 7 bit, corrispondente al quadrato del numero binario, in codice BCD, in ingresso. A volte, può capitare di dover avere anche l'uscita del circuito in forma BCD: questo, per esempio, è il caso in cui tale uscita serve a pilotare un display a 7 segmenti, formato dalle 2 cifre corrispondenti al numero considerato.

In questo caso, quindi, è necessario esprimere il quadrato dell'ingresso ancora in formato BCD, il che significa codificare le singole cifre del risultato in BCD, cioè assegnando a ciascuna di esse 4 bit, secondo la seguente tabella della verità:

Ingresso decimale	i_0	i_1	i_2	i_3	Uscita decimale	u_0	u_1	u_2	u_3	u_4	u_5	u_6	u_7
(0)	0	0	0	0	(00)	0	0	0	0	0	0	0	0
(1)	0	0	0	1	(01)	0	0	0	0	0	0	0	1
(2)	0	0	1	0	(04)	0	0	0	0	0	1	0	0
(3)	0	0	1	1	(09)	0	0	0	0	1	0	0	1
(4)	0	1	0	0	(16)	0	0	0	1	0	1	1	0
(5)	0	1	0	1	(25)	0	0	1	0	0	1	0	1
(6)	0	1	1	0	(36)	0	0	1	1	0	1	1	0
(7)	0	1	1	1	(49)	0	1	0	0	1	0	0	1
(8)	1	0	0	0	(64)	0	1	1	0	0	1	0	0
(9)	1	0	0	1	(81)	1	0	0	0	0	0	0	1

Anche in questo caso possiamo utilizzare una ROM: osservando che l'uscita u7 coincide con i_3, mentre non sono possibili altre semplificazioni, tale ROM dovrà avere 4 ingressi e 7 uscite.

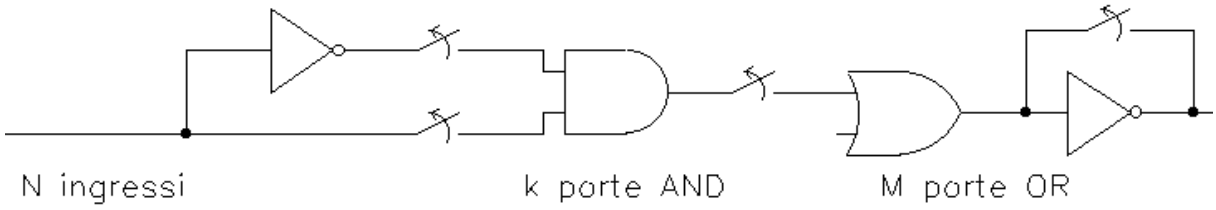
Osservazioni: EPROM e EEPROM

Abbiamo detto, introducendo la ROM, che essa viene programmata (cioè ne vengono riempite le teoriche celle di memoria) all'atto della fabbricazione o, comunque, una ed una sola volta: una volta bruciati gli opportuni fusibili (al fine di implementare le funzioni richieste), non è più possibile ripristinarli. In effetti, spesso è opportuno invece poter riprogrammare la ROM ed è per questo motivo che sono sorte le **ROM programmabili**: tra queste segnaliamo in particolare le **EPROM** (*Erasable Programmable ROM*), realizzate principalmente usando transistori MOS, e le **EEPROM** (*Electrically Erasable Programmable ROM*). In queste ultime, la cancellazione, prima della successiva riprogrammazione, avviene elettricamente.

PLA: PROGRAMMABLE LOGIC ARRAY

Così come una ROM si può usare per implementare una qualsiasi funzione degli ingressi, qualcosa di simile accade per la **PLA** (in italiano, l'acronimo sta per *schiera logica programmabile*): *la differenza principale con una ROM è che una PLA non può realizzare tutte le possibili funzioni degli N ingressi, ma solo un certo numero M di esse.* Anche in questo caso, una volta individuate le funzioni di implementare, si scelgono le caratteristiche della ROM (cioè, in definitiva, il numero di ingressi, il numero di uscite ed il numero di porte logiche) e, bruciando fusibili posti in posizione opportuna, si effettua la programmazione.

Prima di esaminare un caso pratico, che meglio chiarisca la struttura di una PLA, possiamo schematizzare tale struttura con riferimento alla figura seguente:



La PLA presenta N ingressi, ciascuno dei quali può andare in ingresso ad una porta AND, eventualmente complementato (basta bruciare il corrispondente fusibile). Le porte AND, ciascuna ad N ingressi, sono in tutto k: con questo procedimento, quindi, ogni porta AND può fornire in uscita un qualsiasi mintermine di una funzione ad N variabili.

L'uscita delle generica porta AND può andare in ingresso (dipende dall'annesso fusibile) ad una porta OR: le porte OR (ciascuna a k ingressi, tanti quante sono le AND) sono in totale M.

Il numero totale di fusibili impiegati si individua facilmente:

- per ogni uscita abbiamo 1 fusibile, per cui ci sono M fusibili di uscita;
- le uscite delle k porte AND vanno in ingresso, ciascuna mediante un fusibile, ad una porta OR, per cui abbiamo altri kM fusibili;
- ciascuna variabile va infine direttamente in ingresso a ciascuna porta OR, per cui kN fusibili, oppure ci va complementata, per cui altri kN fusibili.

In tutto, abbiamo dunque 2k(N+M) fusibili, che ci servono per la programmazione.

L'uscita delle porte OR, eventualmente complementata, costituisce l'uscita della PLA: ciò comporta che il circuito possa implementare, al più, M funzioni delle N variabili di ingresso. E' chiaro che, essendo 2^{2^N} il numero totale di funzioni di N variabili, se $M = 2^{2^N}$, il circuito può realizzare tutte le funzioni di N variabili. Questo, però, non accade mai, in quanto, come vedremo adesso, il meccanismo su cui si basa la programmazione della PLA è diverso da quello visto per la ROM e per i circuiti precedenti.

Per comprendere come scegliere e, successivamente, programmare una PLA, consideriamo un esempio pratico. Consideriamo, in primo luogo, 3 variabili, per cui la PLA dovrà avere N=3 ingressi. Con 3 variabili possiamo realizzare al più 2^3=8 funzioni booleane distinte: supponiamo, ad esempio, di volerne realizzare solo 4. Dato che la PLA deve presentare tante uscite quante sono le funzioni da realizzare, deduciamo che deve essere M=4.

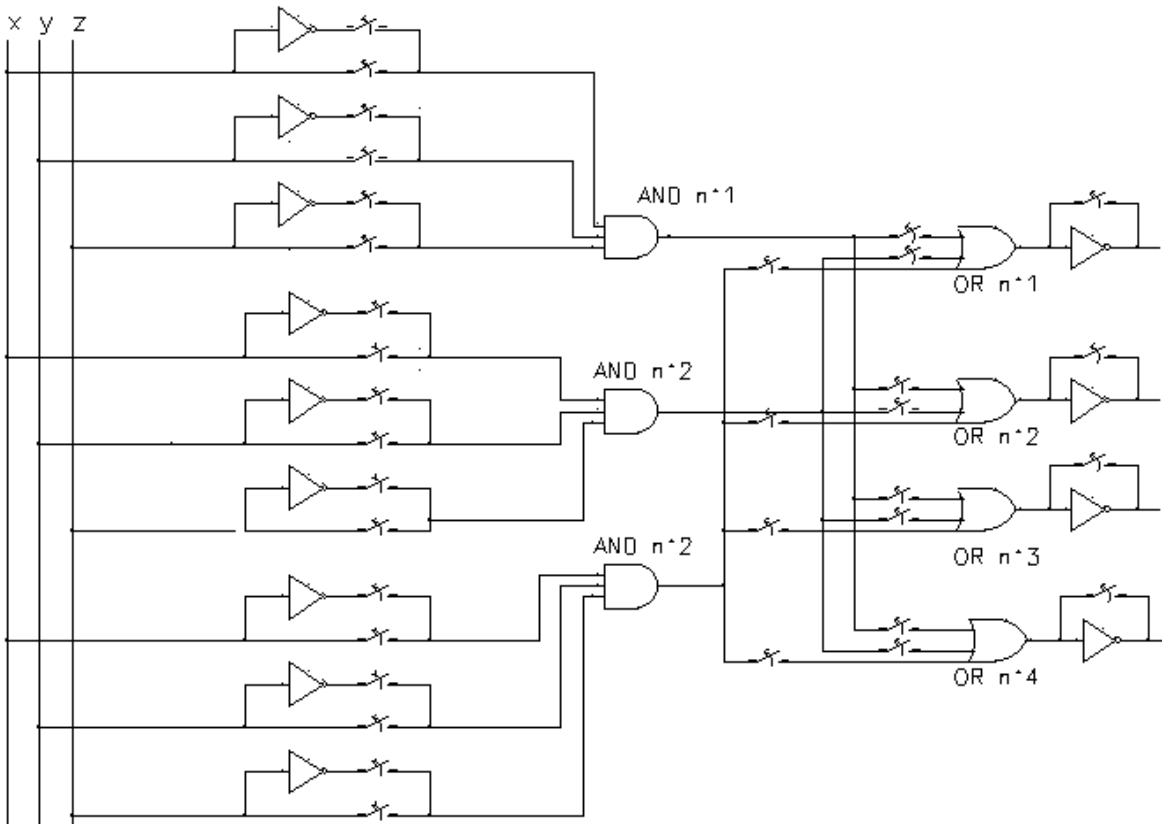
Resta da stabilire quante porte AND servono. A tal fine, serve conoscere l'espressione analitica delle funzioni da realizzare. In particolare, a differenza dei circuiti esaminati in precedenza, nel caso della PLA è fondamentale operare una minimizzazione delle funzioni: questa minimizzazione, però, non ha lo scopo di giungere alle forme minime delle funzioni, bensì ha lo scopo di esprimere tali funzioni in termini di implicanti primi. In particolare, ***i vantaggi della PLA vengono fuori qualora le funzioni considerate abbiano uno o più implicanti primi in comune:*** infatti, come si può intuire dallo schema generale tracciato prima, l'uscita di ciascuna porta AND può rappresentare non solo un mintermine, ma anche semplicemente un implicante, dato che non necessariamente tutti gli ingressi devono giungere in ingresso alla porta stessa. Se due funzioni hanno un implicante in comune, basterà 1 sola porta AND per realizzarlo, dopo di che esso sarà inviato alle 2 distinte porte OR che forniscono in uscite le 2 funzioni. In questo consiste il grande pregio di una PLA. Vediamo cosa accade nel nostro caso.

Supponiamo che le espressioni analitiche delle 4 funzioni considerate, in termini di implicanti primi, siano le seguenti:

$$F_1 = xy$$
$$F_2 = xy + z$$
$$F_3 = x'y'z + z$$
$$F_4 = xy + x'y'z$$

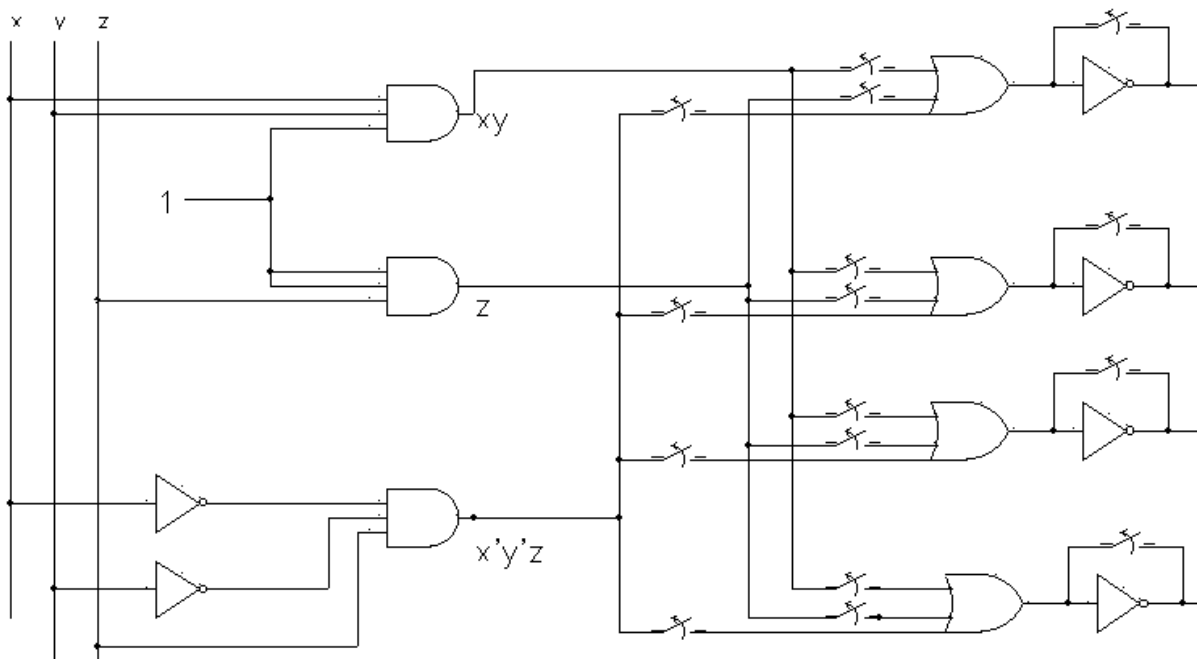
Il criterio con cui scegliere il numero k di porte AND è immediato: basta contare il numero di implicanti primi da realizzare, cui eventualmente aggiungere i mintermini. Nel nostro caso, abbiamo 2 implicanti (xy e z) ed 1 mintermine (x’y’z), per cui dobbiamo predisporre una PLA con k=3 porte AND.

Possiamo dunque cominciare a tracciare la struttura generale della PLA:



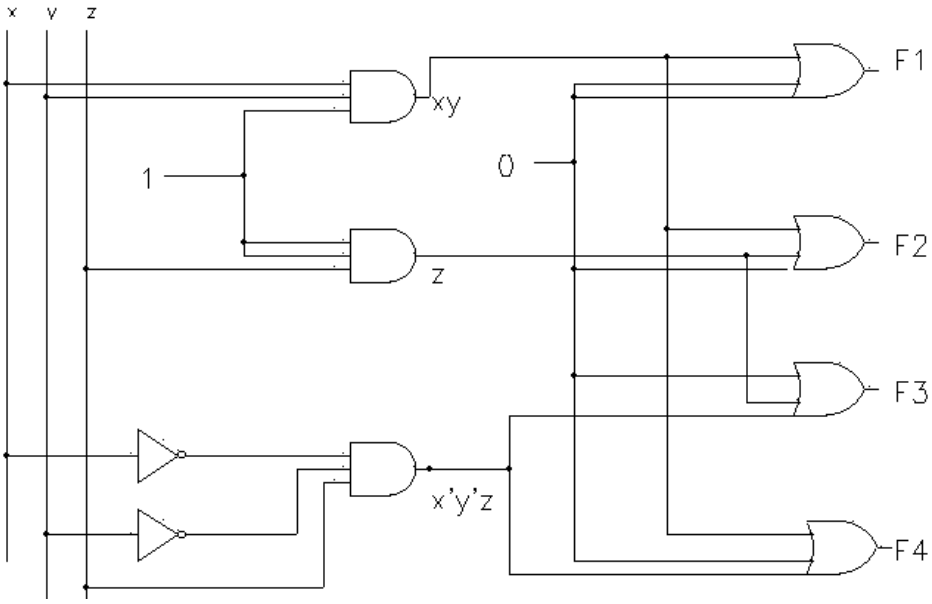
Per ottenere l’implicante xy, comune alle prime due funzioni e all’ultima, ci basta mandare in ingresso alla prima porta AND le variabili x ed y, ponendo ad 1 il terzo ingresso. In modo analogo, per ottenere l’implicante z, dobbiamo inviare z in ingresso alla seconda porta AND e porre ad 1 gli altri due ingressi. Infine, per ottenere il mintermine x’y’z, basta mandare, in ingresso alla terza porta AND, le variabili x ed y complementate e z.

Possiamo perciò programmare la prima parte della PLA, relativa alle porte AND ed ai loro ingressi. Bruciando gli opportuni fusibili, otteniamo lo schema seguente:

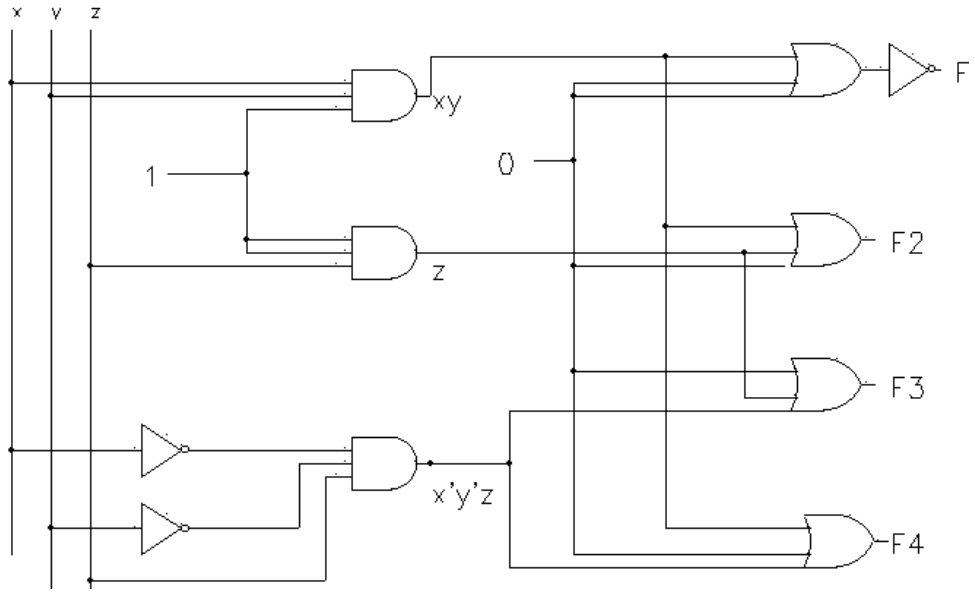


Il passo successivo consiste nel comporre le uscite delle 3 porte AND in modo da ottenere le funzioni desiderate: l'uscita della prima porta AND va portata in ingresso alle porte OR 1,2 e 4; l'uscita della seconda porta AND va invce in ingresso alle porte OR 2 e 3; infine, l'uscita della terza porta AND va in ingresso alle porte OR 3 e 4.

Otteniamo così il seguente schema, nel quale eliminiamo gli inverter a valle delle porte OR in quanto ci interessano le uscite non complementate:



Ci si può, allora, chiedere a cosa servano gli INVERTER posti sulle uscite delle porte OR. Il motivo è semplice: supponiamo, per esempio, dover realizzare la funzione $F = x'+y'$; se ne facciamo il complemento, otteniamo $G = F' = xy$ e questa è proprio la funzione F1 considerata prima. Quindi, se al posto di F_1 avessimo dovuto realizzare F, avremmo potuto realizzare lo stesso identico schema circuitale, includendo però (tramite l'apposito fusibile) l'inverter in cascata alla prima porta OR:



Osservazione: le PAL (Programmable Array Logic)

Distinte dalle PLA viste nel paragrafo precedente sono le **PAL** (*logiche a schiere programmabili*). Le differenze principali sono due: in primo luogo, la PAL fanno uso di un numero molto minore di fusibili rispetto alle PLA; in secondo luogo, ciascuna delle porte OR di uscita è collegata ad un uguale numero di porte AND di ingresso.

Autore: **SANDRO PETRIZZELLI**
e-mail: sandry@iol.it
sito personale: http://users.iol.it/sandry
succursale: http://dilander.il.it/sandry1