DEI
DIPARTIMENTO DI
INGEGNERIA DELL'INFORMAZIONE

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Sistemi Digitali

Progettazione di logica combinatoria

Marta Bagatin, marta.bagatin@unipd.it

Corso di Laurea in Ingegneria dell'Informazione

Anno accademico 2022-2023

Riepilogo delle puntate precedenti

1. I sistemi digitali elaborano le informazioni rappresentandole con **codifiche binarie**
2. Le **porte logiche sono gli elementi di base** per eseguire le funzioni desiderate sulle grandezze binarie
3. La manipolazione delle grandezze binarie si basa sull'**algebra booleana**
4. La **semplificazione/minimizzazione delle funzioni booleane** consente di ottenere implementazioni a costo minimo

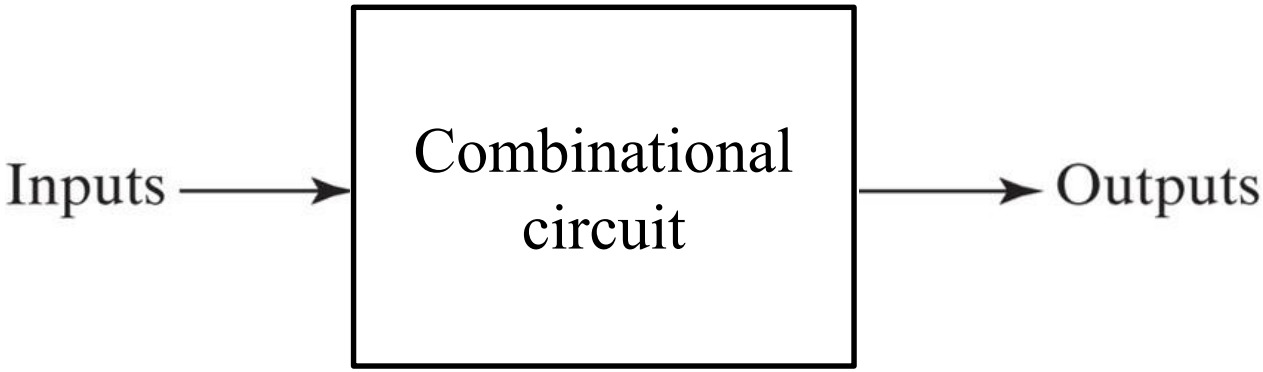
Scopo della lezione

- Definire un sistema logico combinatorio
- Usando i concetti studiati finora, vedere i passi per la **progettazione di un sistema combinatorio**
- Studiare i **blocchi logici combinatori di base** e la loro **descrizione in VHDL**
  - Decoder
  - Encoder
  - Multiplexer

Logica combinatoria vs. Logica sequenziale

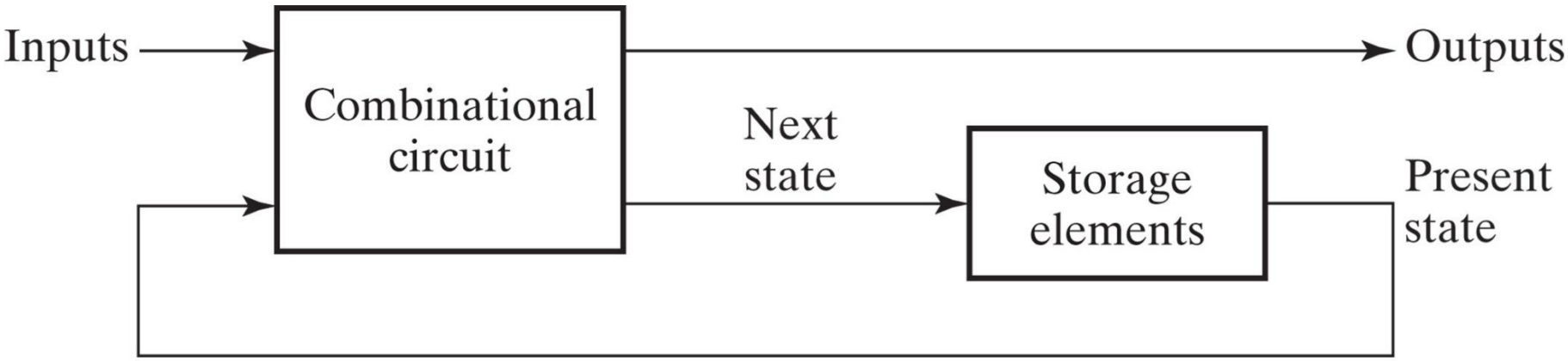
Logica combinatoria

- Un circuito si definisce combinatorio se le sue **uscite**, ad ogni istante di tempo, **dipendono solo dal valore corrente degli ingressi** e NON dalla storia passata degli ingressi
  - Si dice che un sistema combinatorio non ha memoria, la relazione ingresso-uscita non contiene la variabile tempo
- Un circuito combinatorio è composto da una serie di porte logiche interconnesse tra di loro, che non contengono percorsi di retroazione (i.e. dall'uscita all'ingresso)



Logica sequenziale

- Un circuito si definisce sequenziale se le sue **uscite, ad ogni istante di tempo, dipendono sia dal valore corrente degli ingressi che dalla loro storia passata (o stato del sistema)**
- Un circuito sequenziale è costituito da blocchi combinatori ed elementi di memoria, cioè dispositivi in grado di immagazzinare informazioni binarie
  - E' caratterizzato da un percorso di retroazione (feedback) che riporta in ingresso lo stato del sistema



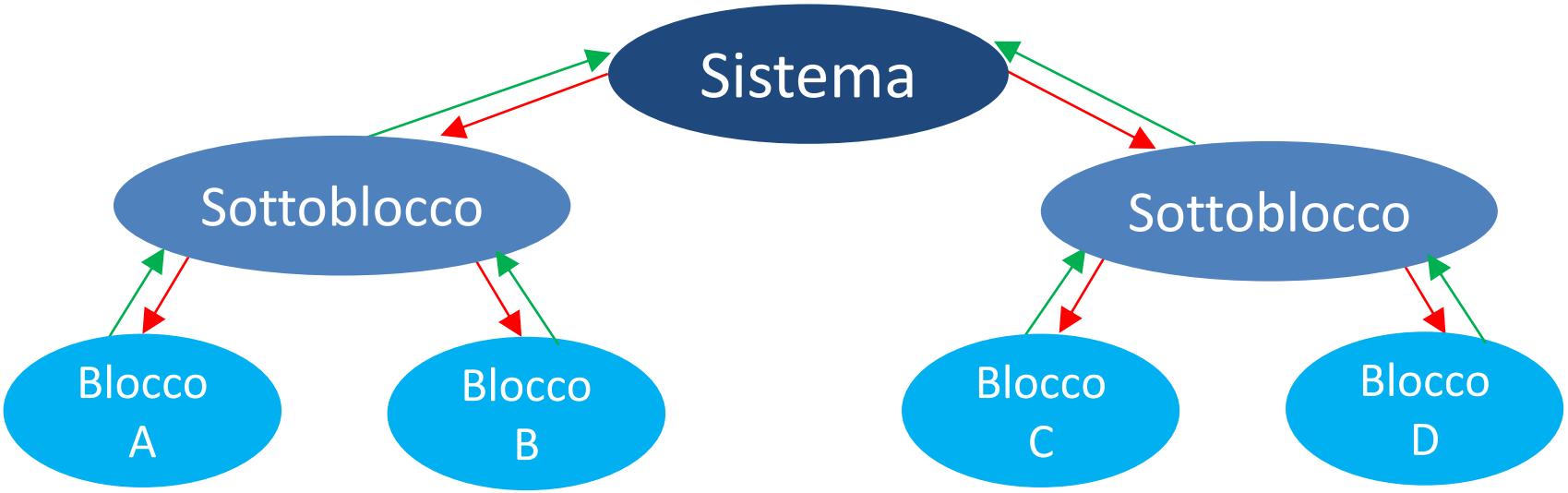
Passi per la progettazione di un circuito combinatorio

1) Identificare le **specifiche di progetto**, determinare il numero di **ingressi e uscite del sistema** e assegnare un nome a ciascuno di essi
2) Trovare la **tabella di verità** che descrive la relazione tra ingressi e uscite
3) Determinare la **funzione booleana a costo minimo per ciascuna uscita in funzione degli ingressi** (mappe di Karnaugh)
4) Disegnare il **diagramma logico** del circuito, usando i blocchi disponibili nella tecnologia scelta
5) Verificare la correttezza del design

Approccio gerarchico

Approccio «divide and conquer»

- E’ un approccio gerarchico
- Il sistema viene **ricorsivamente diviso in sottoblocchi**, fino a che tali blocchi diventano abbastanza semplici da poter essere **progettati singolarmente**
- Si progettano i blocchi: a partire dalle specifiche, si formula la relazione ingressi-uscite, si ottimizza la rappresentazione e si mappa la funzione ottimizzata nelle porte o blocchi logici disponibili nella tecnologia scelta
- I blocchi vengono infine **connessi tra loro** per realizzare il sistema



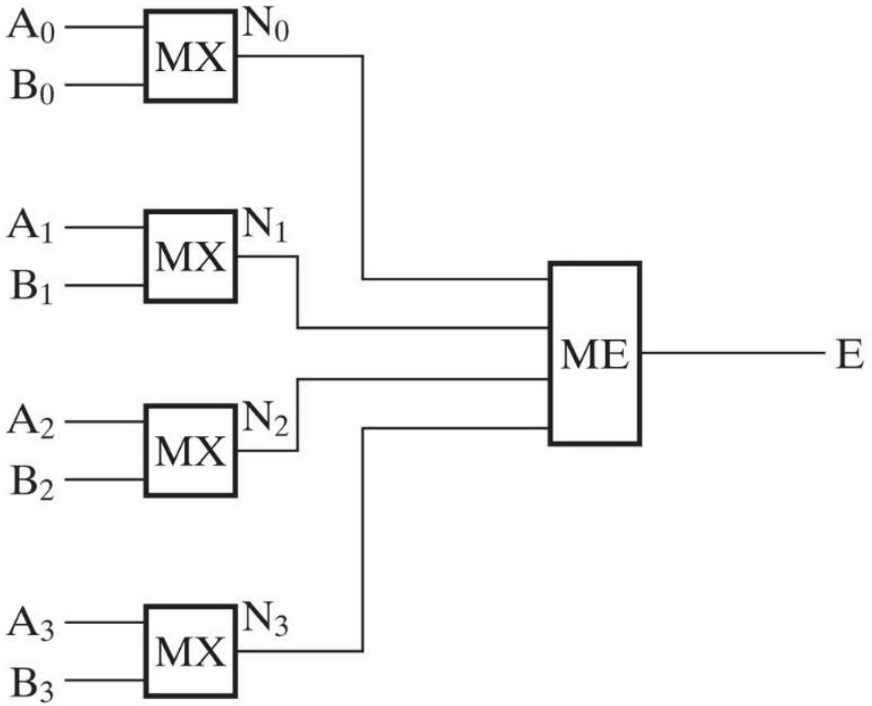
Es. 3.1: Comparatore a 4 bit

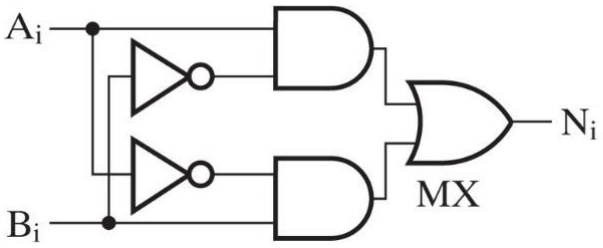
- **Specifica**
  - **Due ingressi A e B** (a 4 bit)
  - **Una uscita E** (a 1 bit)
  - E è uguale a ‘1’ se A e B sono uguali, ‘0’ altrimenti

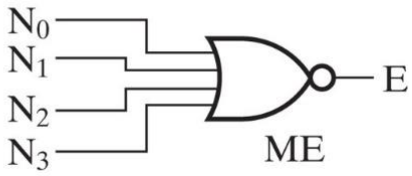
- Con 8 ingressi ($2^8 = 256$ possibili combinazioni) il problema non è gestibile tramite le tabelle di verità ed è impossibile applicare le mappe di Karnaugh, per cui si usa l’approccio gerarchico «divide and conquer»
  - Due vettori di bit sono uguali tra loro se sono uguali tutti i singoli bit che li compongono

Es. 3.1: Comparatore a 4 bit

- <u>1° livello gerarchico</u>
  Sistema diviso in blocchi:
  4 comparatori a 1 bit che confrontano un bit del primo ingresso con il bit corrispondente dell'altro ingresso e un blocco che combina i 4 risultati del confronto per ottenere E
- <u>2° livello gerarchico</u>
  Dettaglio del comparatore a 1 bit e del blocco che combina i risultati dei 4 confronti

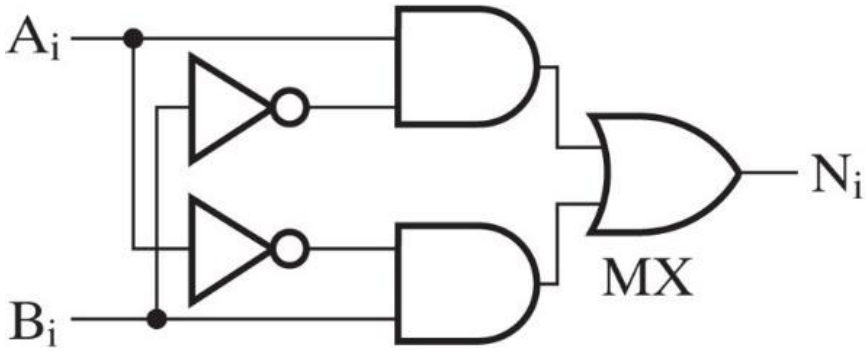






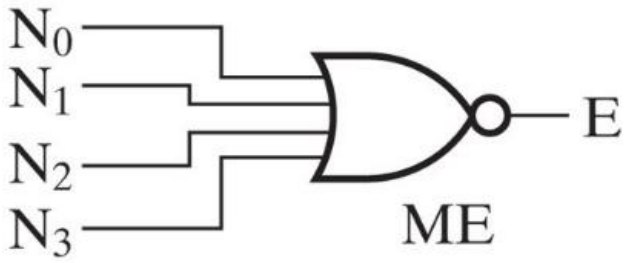
Es. 3.1: Comparatore a 4 bit

- <u>2° livello gerarchico</u>
- Dettaglio del comparatore a 1 bit
  - Uscita è ‘0’ se i due bit in ingresso sono uguali, è uguale a ‘1’ se sono diversi



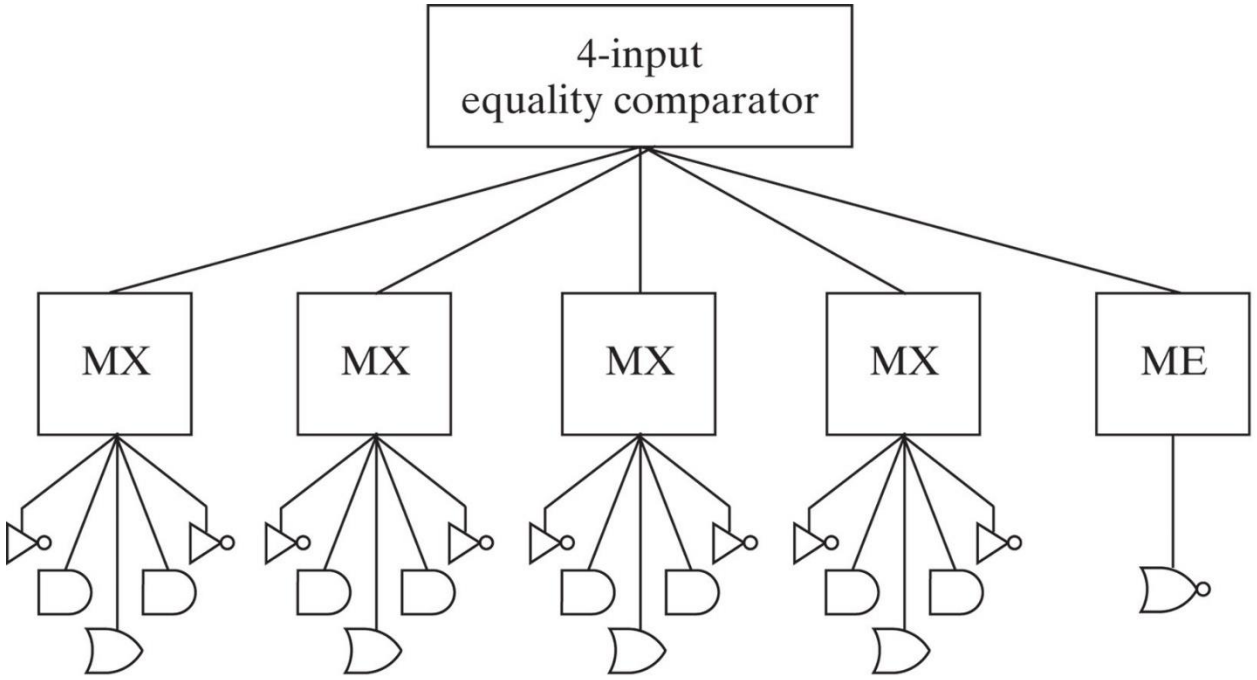
$$N_i = \overline{A_i} B_i + A_i \overline{B_i}$$

- Dettaglio del blocco che combina i risultati dei confronti
  - Uscita è ‘1’ se e solo se tutti i suoi ingressi sono ‘0’ (A = B) ed è uguale a ‘0’ quando almeno uno dei suoi ingressi é ‘1’



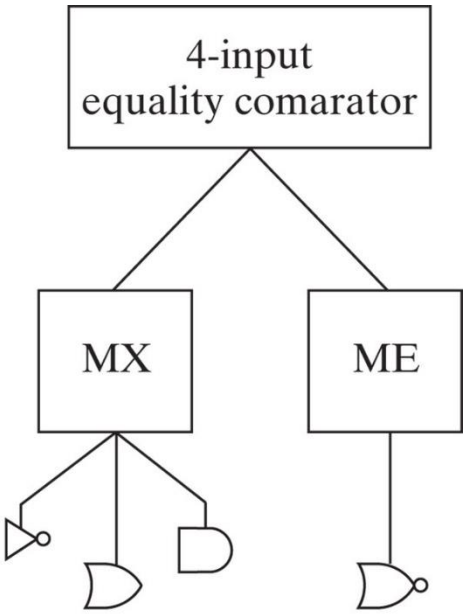
$$E = \overline{N_0 + N_1 + N_2 + N_3}$$

Es. 3.1: Rappresentazione ad albero



- La **rappresentazione gerarchica ad albero** fornisce uno schema semplificato
- Più scendiamo nella gerarchia, più diminuisce il livello di astrazione, per finire con le **foglie**, cioè le porte logiche (**blocchi primitivi**)

Es. 3.1: Rappresentazione ad albero

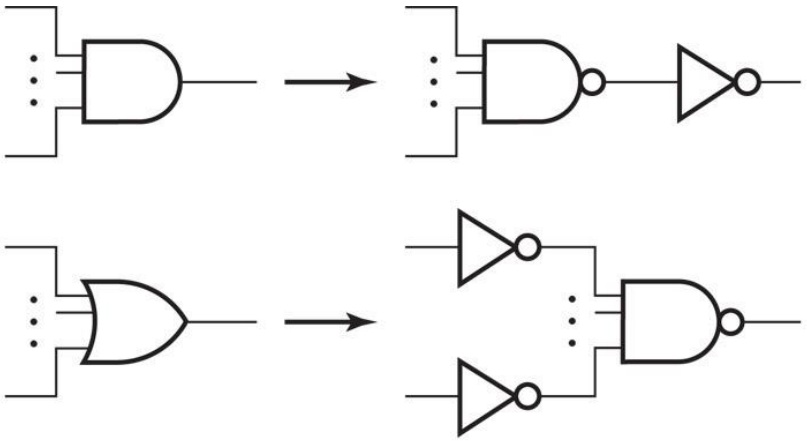


- Una rappresentazione ancora più semplificata mostra una sola volta i blocchi identici tra loro, che, una volta progettati, poi si possono riutilizzare
- **Circuiti regolari** (con blocchi identici ripetuti più volte) richiedono al designer uno sforzo minore quando si tratta di sistemi complessi

- La struttura ad albero non fornisce dettagli sull'implementazione hw, ma solo sul **numero di blocchi da progettare**: il rapporto tra il numero di blocchi primitivi (porte logiche) nel circuito finale e il numero di blocchi in un diagramma gerarchico, inclusi quelli primitivi, fornisce una misura della **regolarità del circuito**

Mappatura tecnologica: cenni

- E’ il passaggio in cui il diagramma logico (o schematico) è sostituito da uno schema che utilizza i componenti disponibili nella particolare tecnologia scelta
- Per motivi legati alle tecnologie usate oggi, le **porte NAND e NOR sono più compatte e veloci** rispetto alle porte AND, OR, NOT, quindi tipicamente si realizzano circuiti che utilizzano solo NAND e NOR



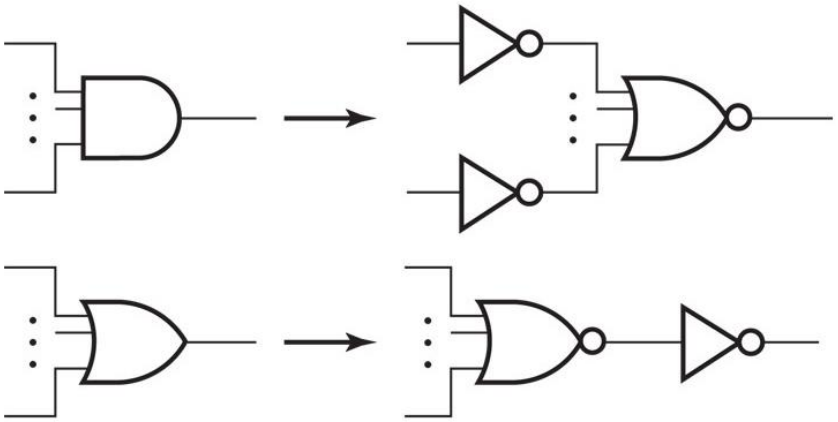
(a) Mapping to NAND gates

Ogni circuito può essere trasformato in un circuito che contiene solo porte NAND e inverter

$$X \cdot Y = \overline{\overline{X \cdot Y}}$$

$$X + Y = \overline{\overline{X + Y}} = \overline{\overline{X} \cdot \overline{Y}}$$

Mappatura tecnologica: cenni



(b) Mapping to NOR gates

In modo duale, possiamo trasformare ogni circuito in un circuito che contiene solo porte NOR e inverter

$$X \cdot Y = \overline{\overline{X \cdot Y}} = \overline{\overline{X} + \overline{Y}}$$

$$X + Y = \overline{\overline{X + Y}}$$

- E' comune progettare circuiti che contengono solo porte NAND e NOT o solo NOR e NOT
  - Caso per caso, si valuta il costo (numero di porte, ingressi, ritardi) e si sceglie l'implementazione meno costosa
- Per rappresentare equazioni booleane in forma SOP, le realizzazioni con porte NAND sono generalmente più adatte, mentre le porte NOR sono più adatte per le rappresentazioni POS

Blocchi logici combinatori

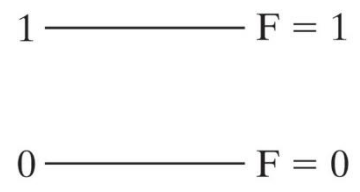
Funzioni ad una variabile

- Sono funzioni ad un **singolo bit**
- **Value-fixing**: alla funzione di uscita viene assegnato un valore costante pari a ‘1’ o ‘0’, indipendentemente dal valore della variabile in ingresso
- **Value-transferring**: il valore presente all’ingresso viene trasferito all’uscita
- **Value-inverting**: il valore presente all’ingresso viene negato all’uscita

X	F = 0	F = X	F = $\overline{X}$	F = 1
0	0	0	1	1
1	0	1	0	1

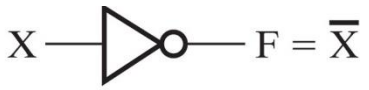
Funzioni ad una variabile: realizzazione

- **Value-fixing**: l'implementazione si indica con la connessione dell'uscita ad un valore logico costante '0' o '1'



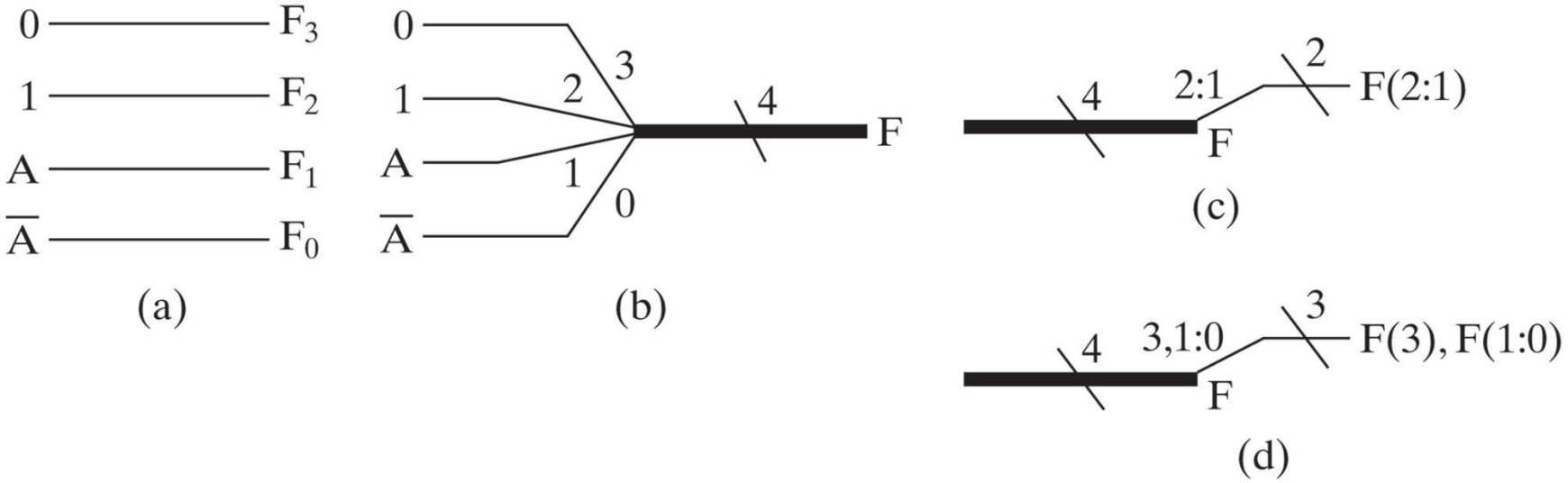
- **Value-transferring**: l'implementazione si indica con una semplice connessione che collega X a F

- **Value-inverting**: l'implementazione corrisponde ad un inverter



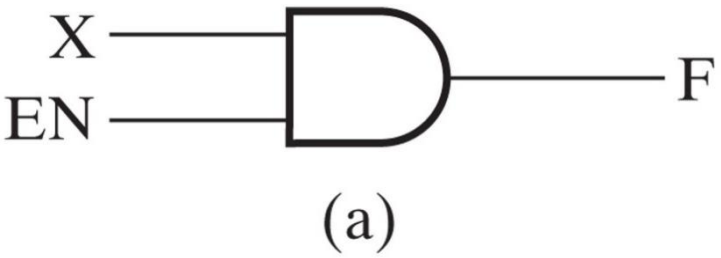
Funzioni a più variabili

- Consideriamo ingressi a bit multipli, e.g. **vettori di singoli bit**
- La notazione per indicare in uno schematico questi vettori è un trattino diagonale con il numero di bit del vettore
- Nel caso vengano trasferiti solo alcuni bit all'interno del vettore, se ne specifica l'indice della posizione nel vettore



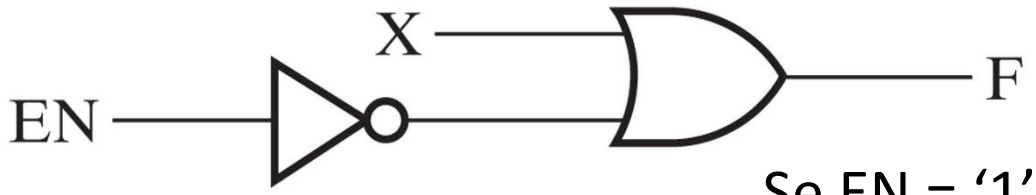
Enabling

- Il circuito di **abilitazione di un segnale**
  - permette al segnale di raggiungere l'uscita se ENABLE (EN) è '1',
  - trasmettere all'uscita un valore fisso ('0' o '1') se ENABLE è '0'
- Tipicamente è **realizzato con una di queste soluzioni**:



(a)

Se EN = '1' F = X, se EN = '0' F = '0'



(b)

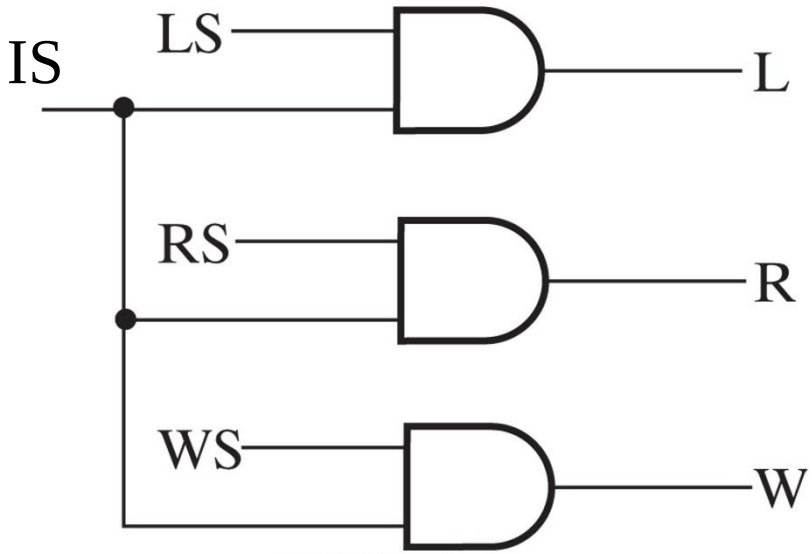
Se EN = '1' F = X, se EN = '0' F = '1'

Es. 3.5: Enabling

- Progettazione del sistema di controllo di un'automobile
  - Luci, radio e alzacristalli sono abilitati solo quando l'auto è accesa
  - Il segnale di accensione (IS) funge da segnale di ENABLE
    - Ignition switch (IS): pulsante accensione auto '0': OFF, '1': ON
    - Light switch (LS): pulsante accensione luci '0': OFF, '1': ON
    - Radio switch (RS): pulsante accensione radio '0': OFF, '1': ON
    - Window switch (WS): pulsante alzacristalli elettrici '0': OFF, '1': ON
    - Lights (L): luci '0' luci spente, '1' luci accese
    - Radio (R): radio '0' radio spenta, '1' radio accesa
    - Power windows (W): alzacristalli '0' alzacristallo inattivo, '1' attivo

Es. 3.5: Enabling

Input Switches				Accessory Control		
IS	LS	RS	WS	L	R	W
0	X	X	X	0	0	0
1	0	0	0	0	0	0
1	0	0	1	0	0	1
1	0	1	0	0	1	0
1	0	1	1	0	1	1
1	1	0	0	1	0	0
1	1	0	1	1	0	1
1	1	1	0	1	1	0
1	1	1	1	1	1	1

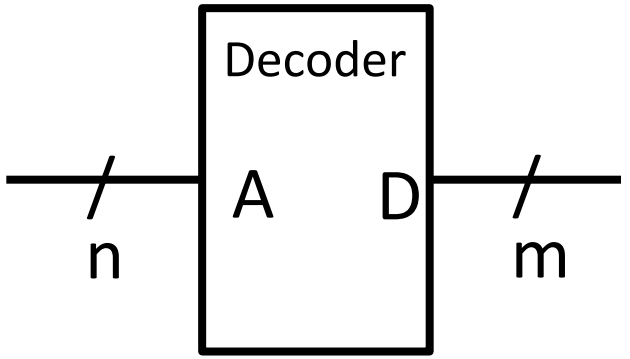


➔ IS = ‘1’: i dispositivi (radio, luci, …) sono controllati dal relativo interruttore

➔ IS = ‘0’: tutti i dispositivi sono spenti, indipendentemente dallo stato del relativo interruttore. Lo stato dei 3 interruttori è indicato con una X nella tabella di verità (don’t care: la prima riga rappresenta $\overline{I}S$, i.e. un termine prodotto che non è un minterm)

Decoder

- Un decoder è un blocco combinatorio che opera una **conversione da una codifica ad un'altra**
- Un **decoder n-to-m trasforma un ingresso a n bit in un'uscita a m bit che rappresenta la sua decodifica 1-hot** (i.e. un numero binario che contiene uno e un solo '1' nella posizione codificata dai bit di ingresso)
  - E.g. Ingresso 3 (11), Uscita 1000
- In altre parole, il decoder pone a '1' l'uscita il cui indice è codificato dai bit in ingresso

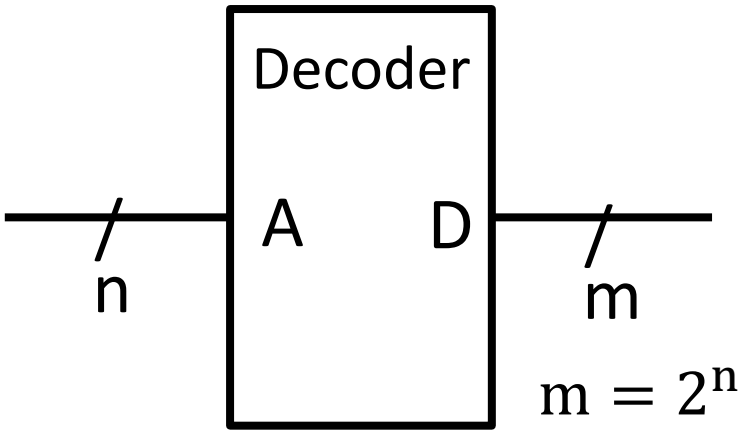


$$n \leq m \leq 2^n$$

- Con n bit si possono rappresentare 2^n valori: $m \leq 2^n$ ($m = 2^n$ se a tutte le combinazioni di bit in ingresso corrisponde un'uscita)

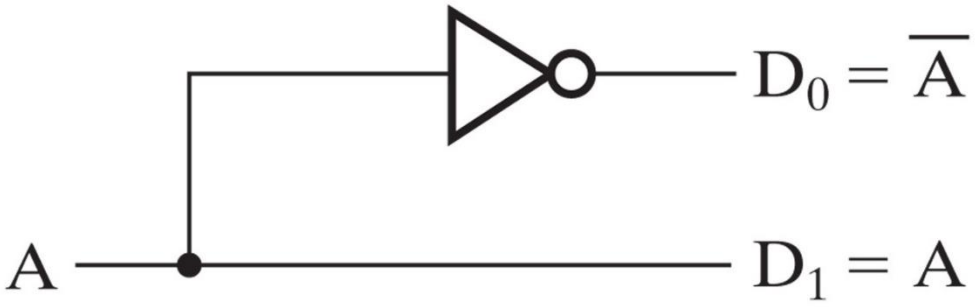
Codifica One-hot

Codifica binaria (n bit)	Codifica one-hot (2^n bit)
000	00000001
001	00000010
010	00000100
011	00001000
100	00010000
101	00100000
110	01000000
111	10000000



Decoder 1-to-2 (n=1, m=2)

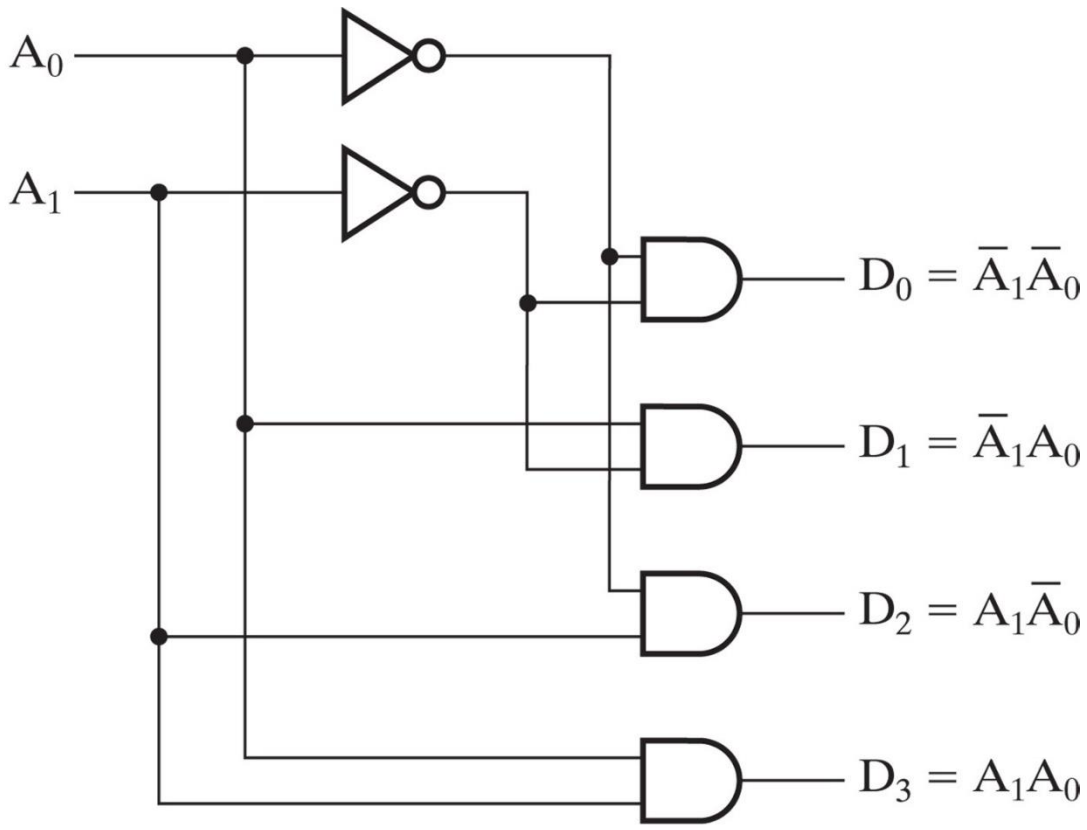
A	D_0	D_1
0	1	0
1	0	1



→ Le uscite di un **decoder n-to-m sono i 2^n minterm corrispondenti alle variabili di ingresso**

Decoder 2-to-4 (n=2, m=4)

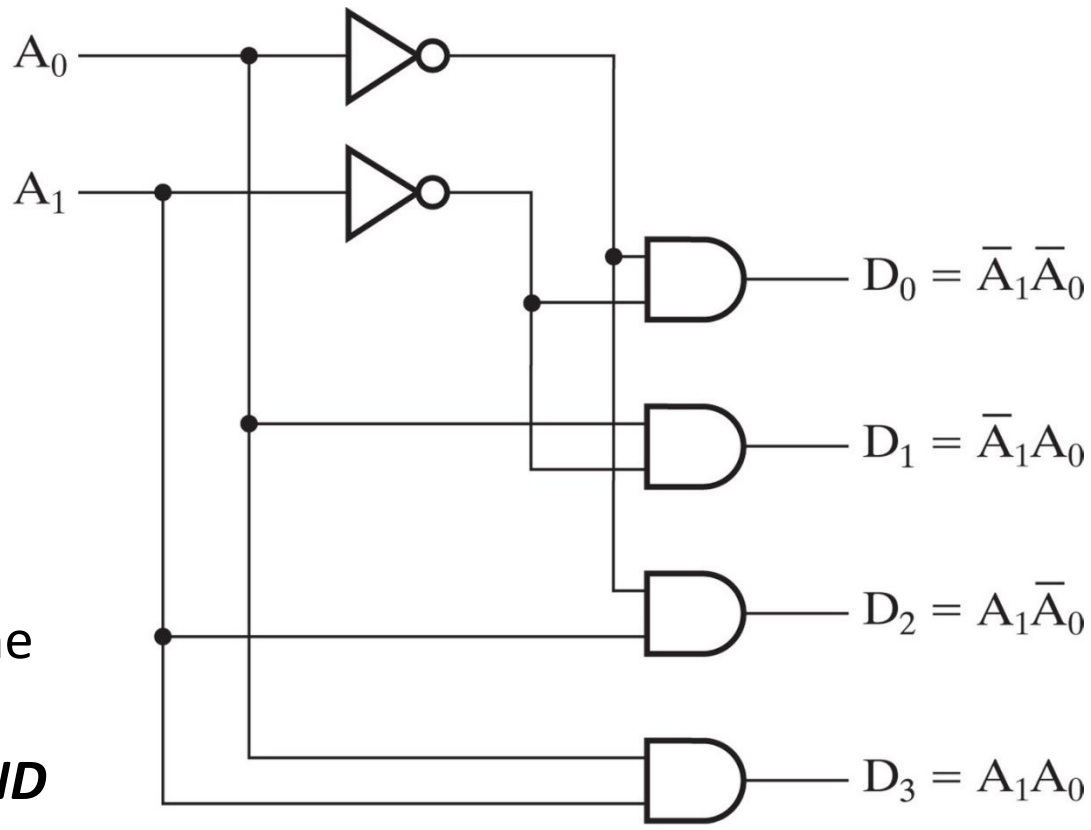
A_1	A_0	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1



Un **decoder n-to-m** si può realizzare con *m* porte AND a *n* ingressi, ciascuna delle quali ha in ingresso una opportuna combinazione degli ingressi (diretti o negati)

Decoder 2-to-4 (n=2, m=4)

A_1	A_0	D_0	D_1	D_2	D_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

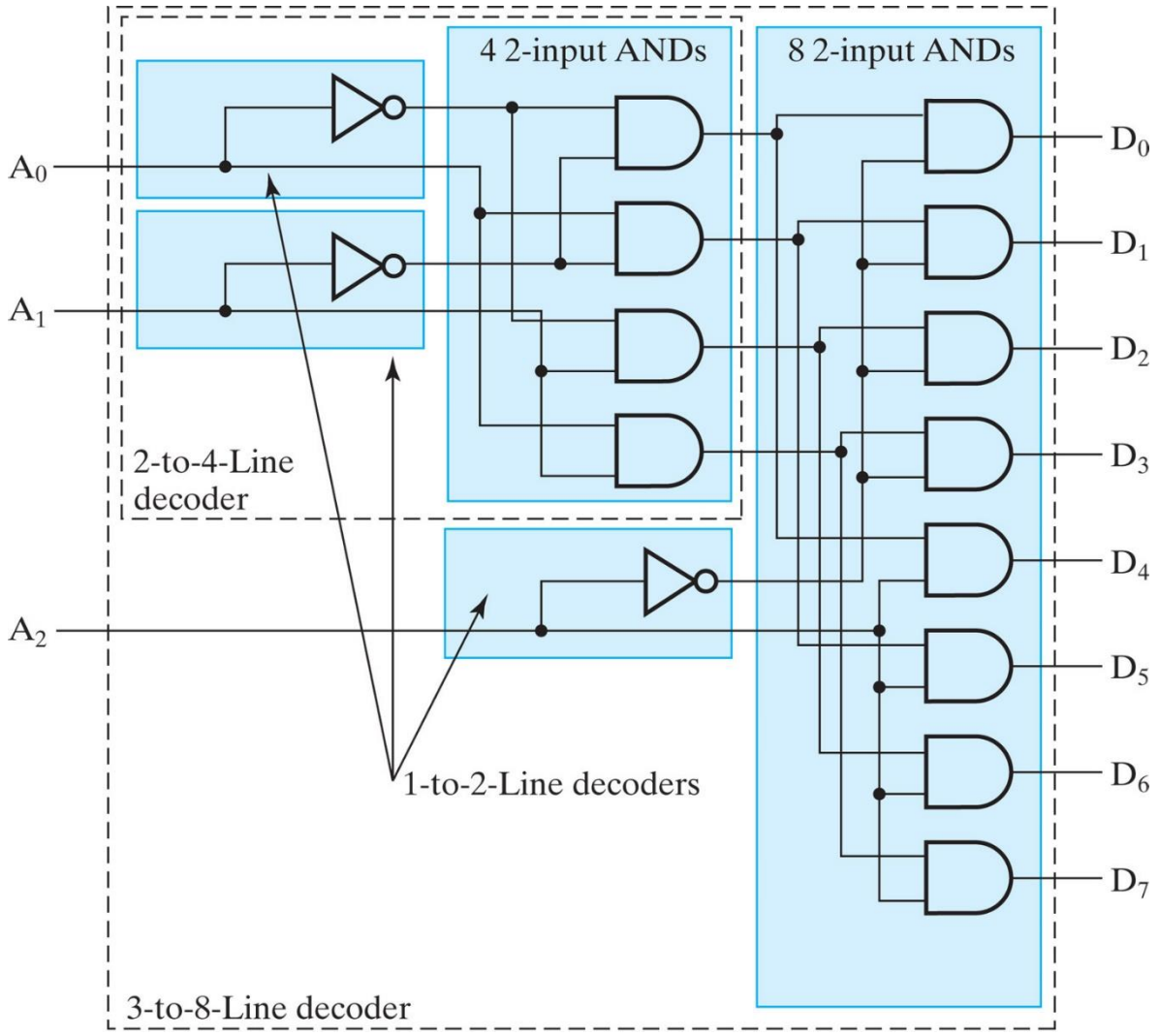


Con una **visione gerarchica**, può essere considerato come un **sistema formato da 2 *decoder 1-to-2* e 4 *porte AND*** (con le 4 combinazioni delle 2 uscite dei decoder connesse agli ingressi delle porte AND)

Decoder: Approccio gerarchico

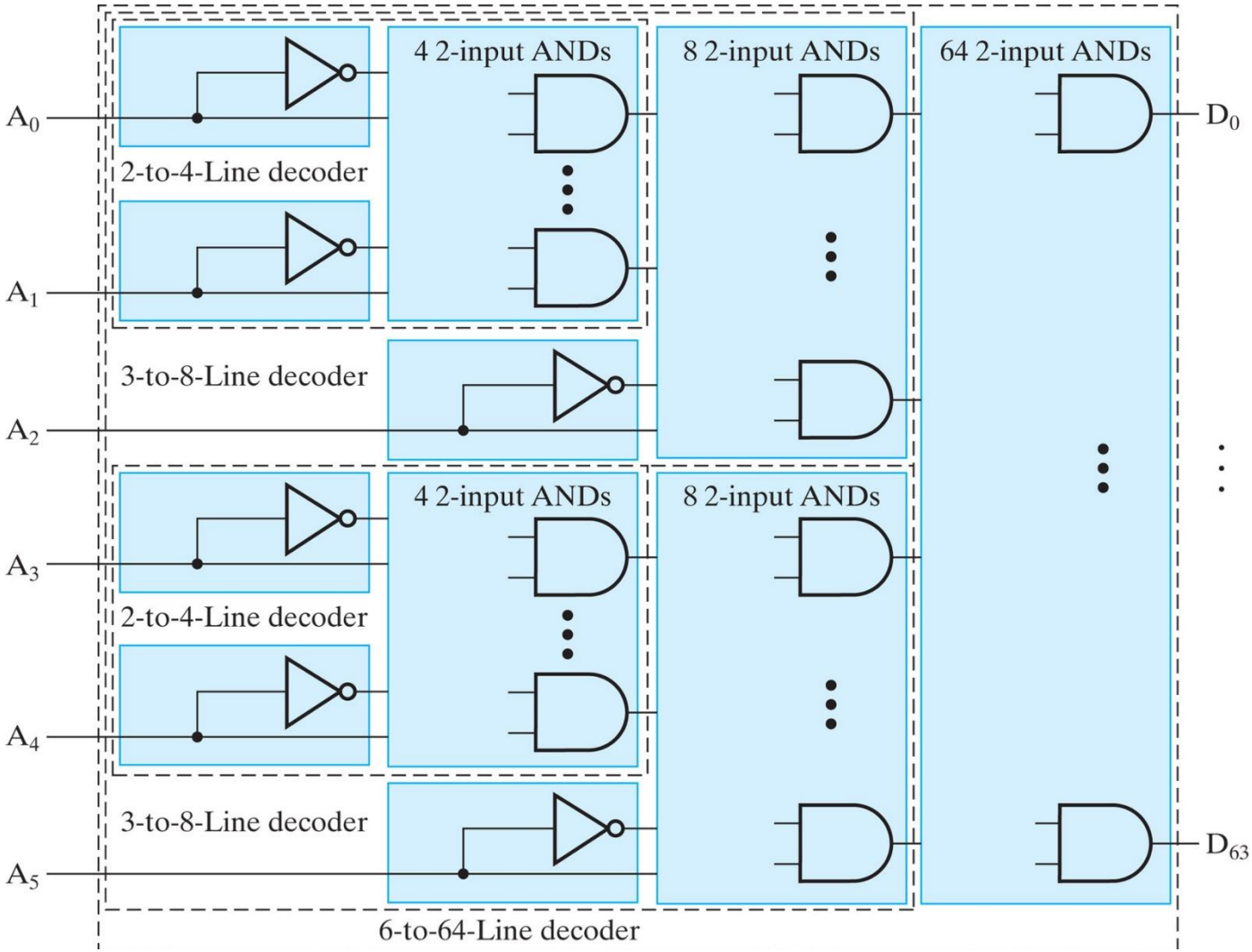
- Via via che la dimensione del decoder cresce, l'approccio visto diventa costoso (in termini di numero di ingressi delle porte AND)
- Per decoder più grandi si usa quindi un approccio gerarchico, connettendo opportunamente tra loro decoder più piccoli
- Esempi
  - *Decoder 3-to-8* si realizza con 1 *decoder 2-to-4* e 1 *decoder 1-to-2*, con in cascata 8 *porte AND* a 2 ingressi
  - *Decoder 6-to-64* si realizza con 2 *decoder 3-to-8*, con in cascata 64 *porte AND* a 2 ingressi
  - Si possono scegliere diverse combinazioni per le implementazioni gerarchiche, con un numero variabile di stadi in cascata, a seconda di quello che si vuole ottimizzare (numero di porte logiche o numero di ingressi per le porte AND)

Decoder 3-to-8



Schema logico di un decoder 3-to-8 con struttura gerarchica

Decoder 6-to-64



Il costo in termini di numero di ingressi è pari a 182

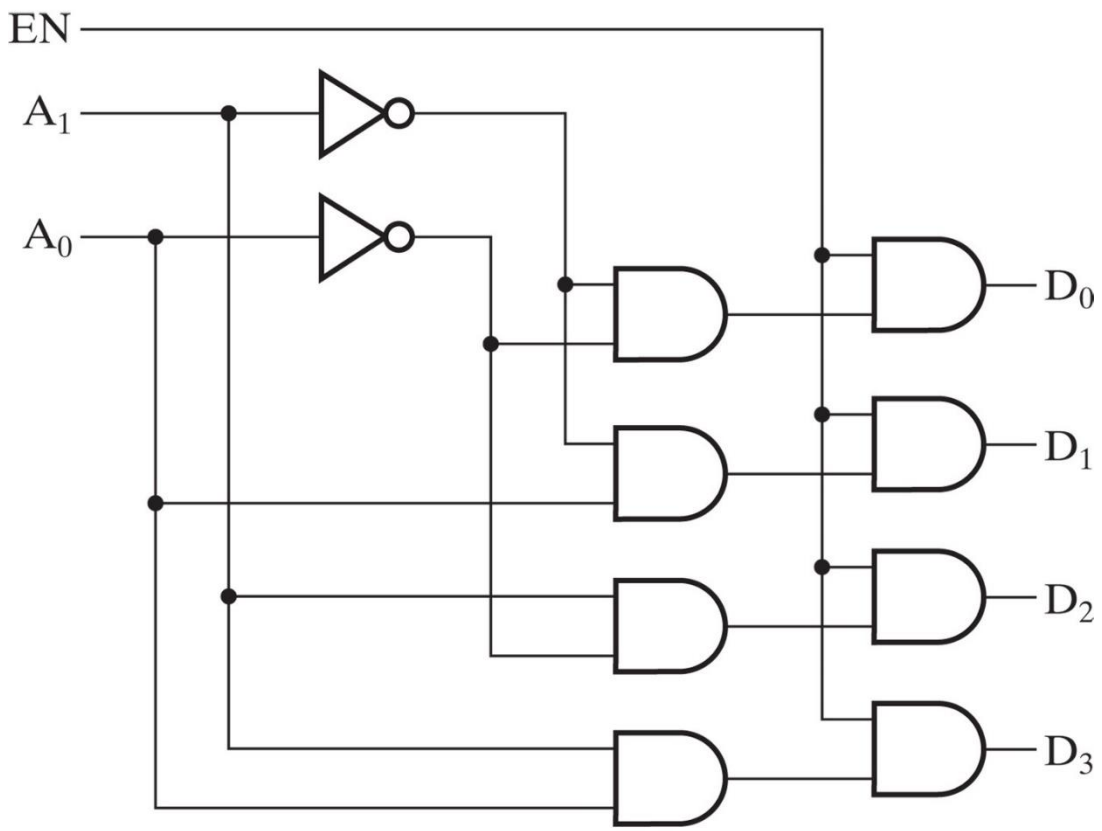
Schema ad albero di un decoder 6-to-64 con struttura gerarchica

Decoder con circuito di enable

- Il circuito di enabling consente di abilitare o meno le uscite del decoder
- Un decoder con circuito di enable si ottiene collegando **m circuiti di enable alle uscite del decoder**
- Per decoder di dimensioni grandi ($n \geq 4$), la soluzione a costo minore (in termini di numero di ingressi delle porte) è quella che collega i circuiti di enable direttamente agli ingressi del decoder anziché alle uscite

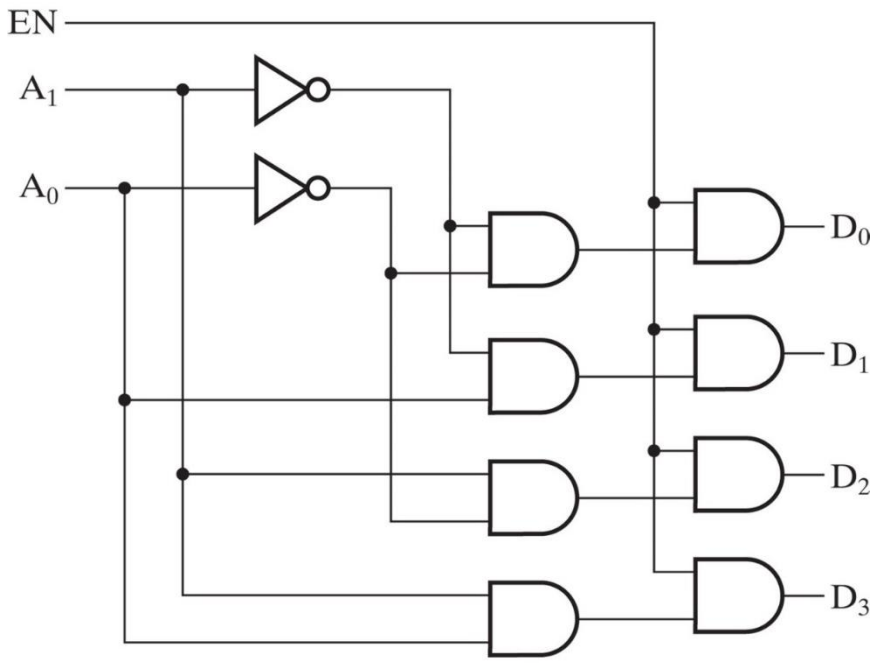
Decoder 2-to-4 con circuito di enable

EN	A_1	A_0	D_0	D_1	D_2	D_3
0	X	X	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1



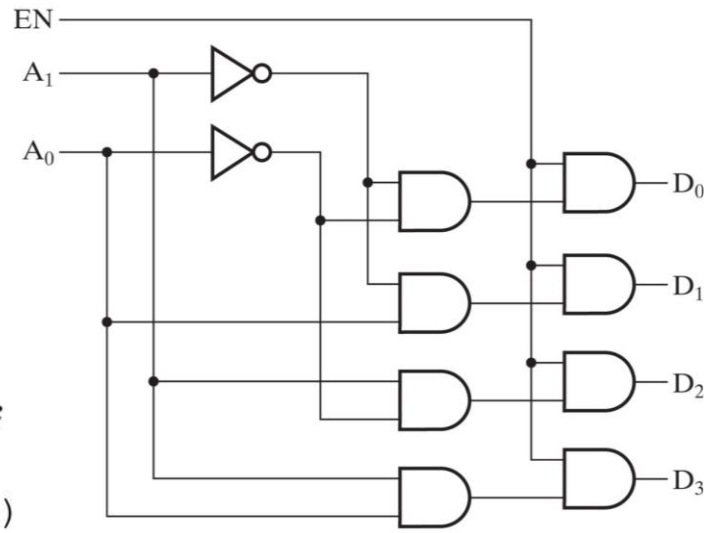
Decoder 2-to-4 con enable: VHDL entity

```
-- 2-to-4-Line Decoder with Enable: Structural VHDL Description
-- (See Figure 3-16 for logic diagram)
library ieee, lcdf_vhdl;
use ieee.std_logic_1164.all, lcdf_vhdl.func_prims.all;
entity decoder_2_to_4_w_enable is
   port (EN, A0, A1: in std_logic;
         D0, D1, D2, D3: out std_logic);
end decoder_2_to_4_w_enable;
```



Decoder 2-to-4 con enable: VHDL structural

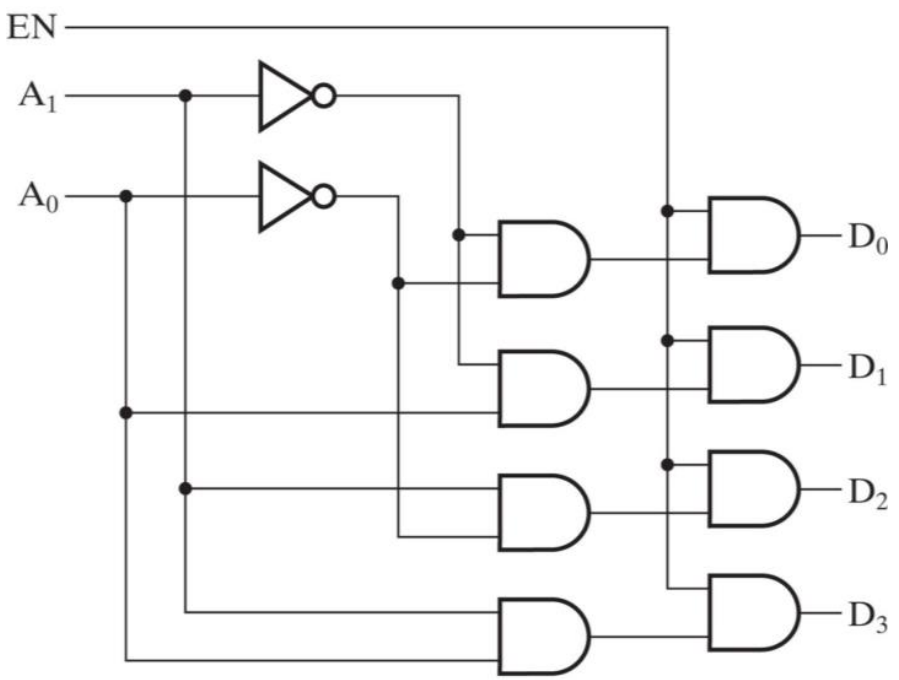
```
architecture structural_1 of decoder_2_to_4_w_enable is
   component NOT1
      port (in1: in std_logic;
            out1: out std_logic);
   end component;
   component AND2
      port (in1, in2: in std_logic;
            out1: out std_logic);
   end component;
   signal A0_n, A1_n, N0, N1, N2, N3: std_logic;
   begin
      g0: NOT1 port map (in1 => A0, out1 => A0_n)
      g1: NOT1 port map (in1 => A1, out1 => A1_n);
      g2: AND2  port map  (in1 => A0_n, in2 => A1_n, out1 => N0);
      g3: AND2  port map  (in1 => A0, in2 => A1_n, out1 => N1);
      g4: AND2  port map  (in1 => A0_n, in2 => A1, out1 => N2);
      g5: AND2 port map   (in1 => A0, in2 => A1, out1 => N3);
      g6: AND2 port map   (in1 => EN, in2 => N0, out1 => D0);
      g7: AND2 port map   (in1 => EN, in2 => N1, out1 => D1);
      g8: AND2 port map   (in1 => EN, in2 => N2, out1 => D2);
      g9: AND2 port map   (in1 => EN, in2 => N3, out1 => D3);
end structural_1;
```



Decoder 2-to-4 con enable: VHDL dataflow

```
architecture dataflow_1 of decoder_2_to_4_w_enable is

signal A0_n, A1_n: std_logic;
begin
    A0_n <= not A0;
    A1_n <= not A1;
    D0 <= A0_n and A1_n and EN;
    D1 <= A0 and A1_n and EN;
    D2 <= A0_n and A1 and EN;
    D3 <= A0 and A1 and EN;
end dataflow_1;
```



Più compatta rispetto alla descrizione strutturale: non fa uso di componenti ma usa gli operatori predefiniti per descrivere il circuito sottoforma di equazioni Booleane

Circuiti combinatori basati su decoder

- Un decoder, dati n bit in ingresso, fornisce in uscita i 2^n minterm relativi
- Tutte le funzioni logiche si possono esprimere in forma SOP (somma dei minterm della funzione)

Qualsiasi funzione logica si può realizzare con un decoder e una porta OR!

Es: Sommatore a 1 bit con decoder

Truth Table for 1-Bit Binary Adder

	X	Y	Z	C	S
m_0	0	0	0	0	0
m_1	0	0	1	0	1
m_2	0	1	0	0	1
m_3	0	1	1	1	0
m_4	1	0	0	0	1
m_5	1	0	1	1	0
m_6	1	1	0	1	0
m_7	1	1	1	1	1

$$C(X, Y, Z) = \sum m(3, 5, 6, 7)$$

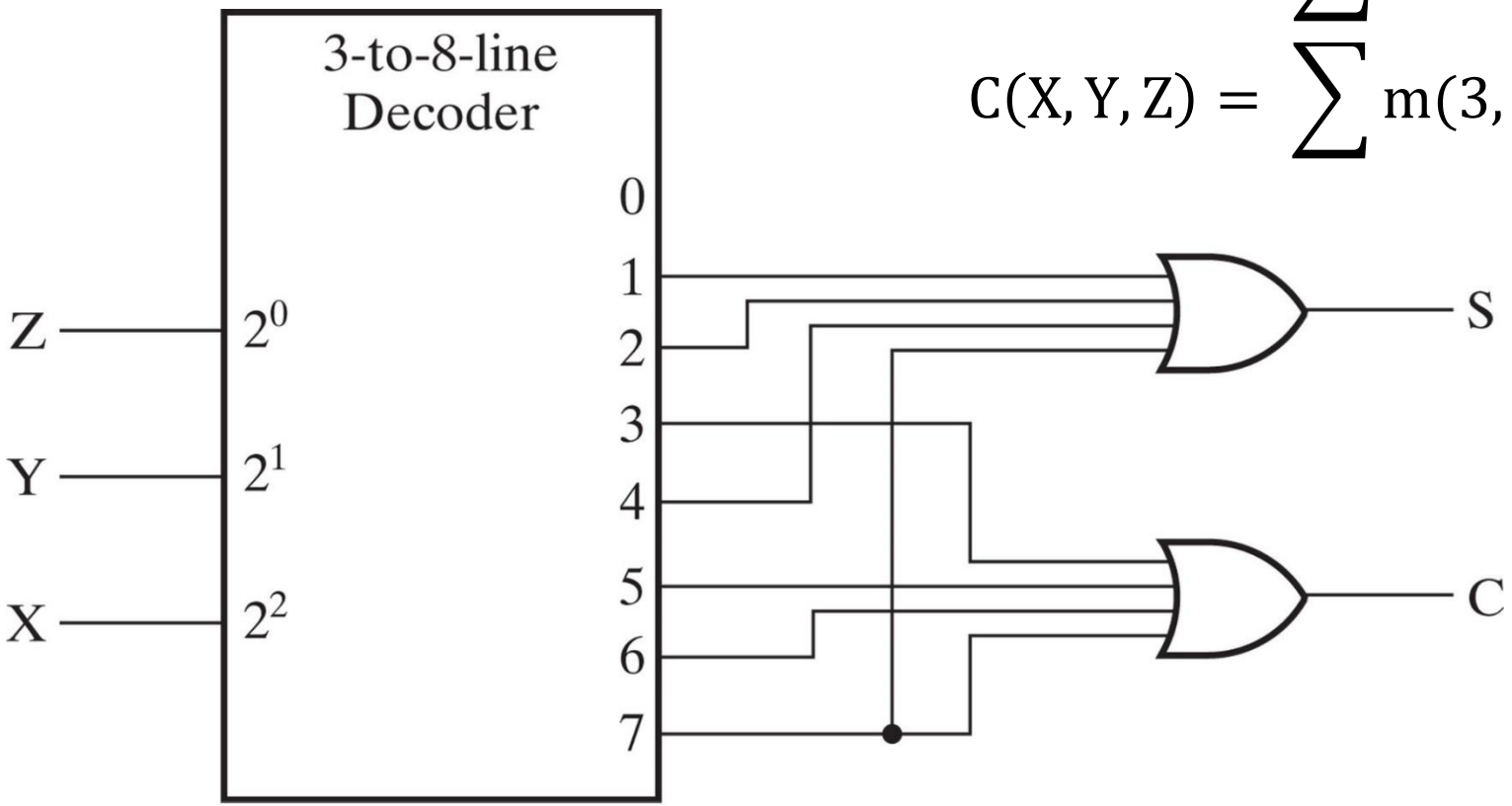
$$S(X, Y, Z) = \sum m(1, 2, 4, 7)$$

Es: Sommatore a 1 bit con decoder

Il decoder genera tutti i minterm, le 2 porte OR sommano quelli che servono per le due funzioni S e C

$$S(X, Y, Z) = \sum m(1, 2, 4, 7)$$

$$C(X, Y, Z) = \sum m(3, 5, 6, 7)$$

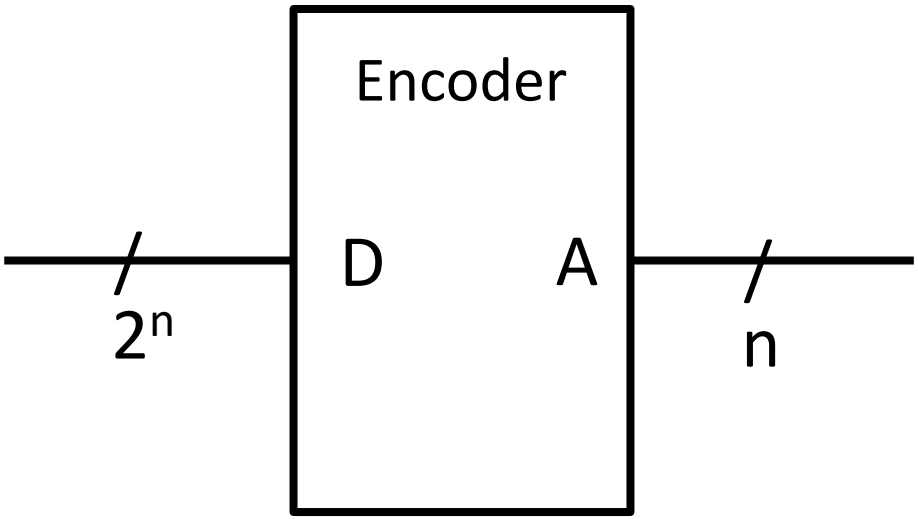


Circuiti combinatori basati su decoder

- L’implementazione di una funzione logica con decoder e porte OR risulta conveniente se la funzione si può esprimere con pochi minterm
- In presenza di tanti minterm della funzione, il numero di ingressi della porta OR aumenta e risulta conveniente sostituire la OR con una NOR avente in ingresso i minterm della funzione negata

Encoder

- E’ il circuito logico che realizza la funzione inversa al decoder: **riceve in ingresso un segnale 1-hot a 2^n bit e produce in uscita la sua codifica binaria a n bit**

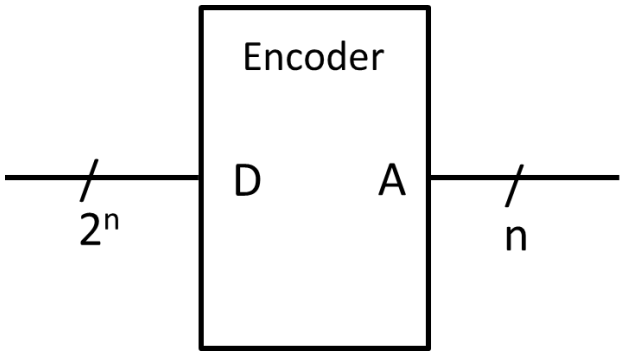


Encoder da ottale a binario

Truth Table for Octal-to-Binary Encoder

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

L'ingresso è il codice one-hot corrispondente alle cifre ottali da 0 a 7, l'uscita è la codifica binaria di ciascuna cifra (3 bit)



Encoder da ottale a binario

Truth Table for Octal-to-Binary Encoder

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Un solo ingresso alla volta può assumere valore ‘1’: la tabella di verità ha 8 righe, per le rimanenti righe le uscite sono don’t care

$$A_2 = D_4 + D_5 + D_6 + D_7$$

$$A_1 = D_2 + D_3 + D_6 + D_7$$

$$A_0 = D_1 + D_3 + D_5 + D_7$$

Encoder: problematiche

Truth Table for Octal-to-Binary Encoder

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Se due ingressi valgono simultaneamente ‘1’ il circuito produce un’uscita errata. Es: se entrambi D_3 e D_6 sono ‘1’, l’uscita è “111” (→ stessa uscita che si ha quando D_7 = ‘1’)

$$A_2 = D_4 + D_5 + D_6 + D_7$$

$$A_1 = D_2 + D_3 + D_6 + D_7$$

$$A_0 = D_1 + D_3 + D_5 + D_7$$

Encoder: problematiche

Truth Table for Octal-to-Binary Encoder

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Altro problema: se tutti gli ingressi valgono ‘0’, l’uscita è “000” (la stessa che si ottiene quando $D_0 = 1$ e $D_1 = D_2 = \ldots D_7 = 0$)

$$A_2 = D_4 + D_5 + D_6 + D_7$$

$$A_1 = D_2 + D_3 + D_6 + D_7$$

$$A_0 = D_1 + D_3 + D_5 + D_7$$

Encoder con priorità

- L'encoder con priorità è in grado di gestire anche le situazioni in cui l'ingresso non è di tipo 1-hot

1) Per **evitare che più di un ingresso assuma il valore '1'**, e quindi l'uscita sia non valida (errata), si introduce una funzione di priorità
   - Si garantisce che venga considerato solo l' '1' con priorità più alta: se più di un ingresso vale '1', l'ingresso con priorità più alta determina il valore dell'uscita, indipendentemente dal valore degli altri ingressi
2) Si aggiunge un'ulteriore uscita V (valid) che viene posta a '1' quando almeno uno degli ingressi vale '1'. In corrispondenza all'ingresso tutti '0', V viene posta a '0' e l'uscita A viene posta a «don't care»

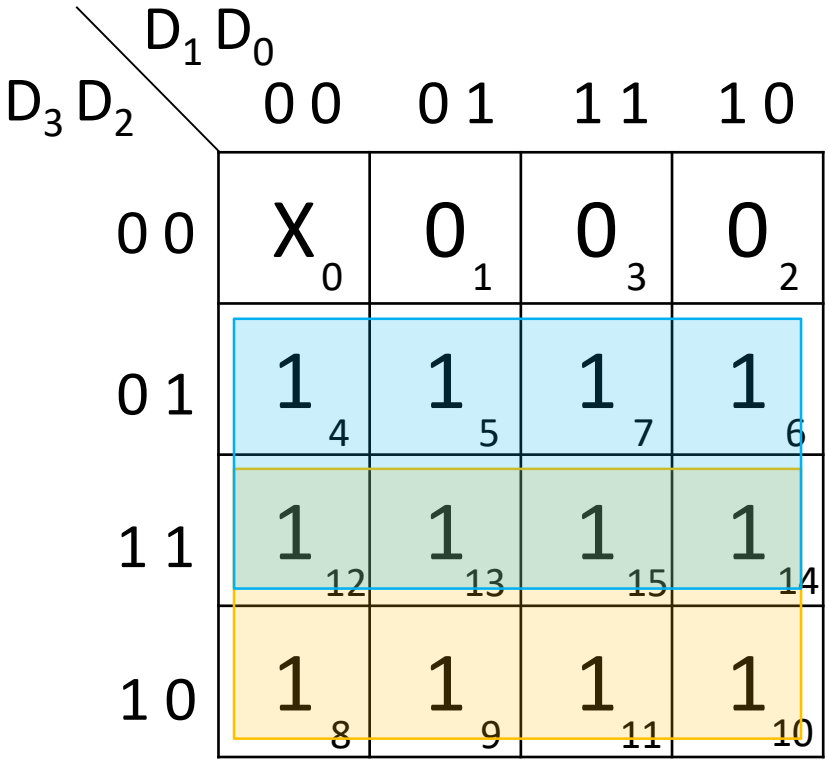
Encoder con priorità a 4 ingressi

Truth Table of Priority Encoder

Inputs				Outputs		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

- D_3 ha priorità più alta: quando D_3 è '1', gli altri input non vengono considerati (indipendentemente dal loro valore, l'uscita è '11', cioè la codifica binaria di '3')
- Il successivo input che ha la priorità dopo D_3, è D_2: se D_3 è '0' e D_2 è '1', indipendentemente da D_1 e D_0, l'uscita sarà '01' (codifica binaria di '2')
- L'output valid (V) indica se l'uscita è valida e vale '0' nel caso in cui tutti gli input valgano '0'

Encoder con priorità a 4 ingressi - MdK

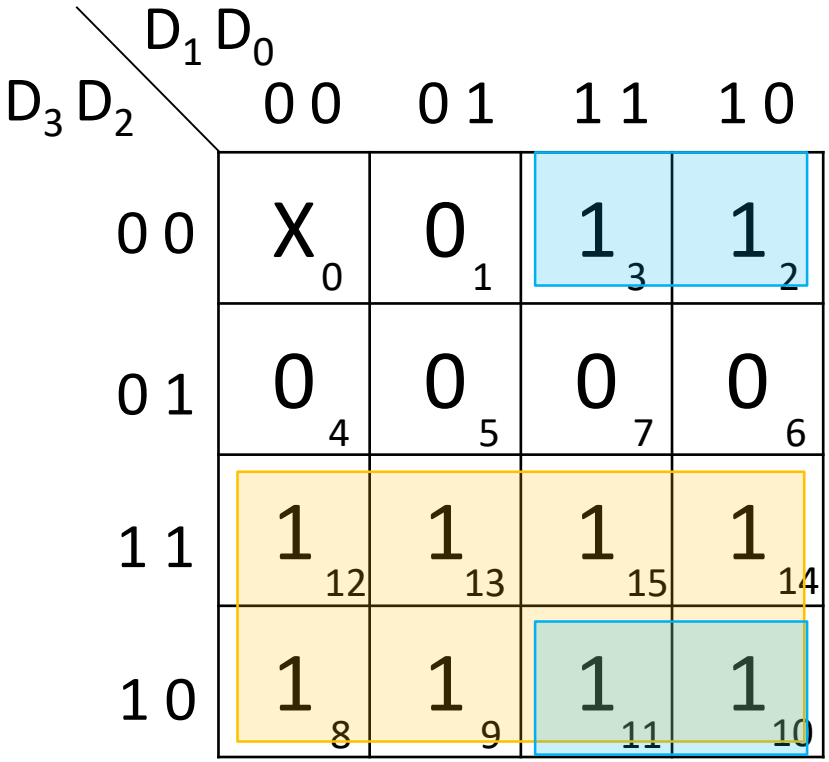


Truth Table of Priority Encoder

Inputs				Outputs		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

$$A_1 = D_2 + D_3$$

Encoder con priorità a 4 ingressi - MdK



Truth Table of Priority Encoder

Inputs				Outputs		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

$$A_0 = D_3 + \overline{D_2}\, D_1$$

Encoder con priorità a 4 ingressi - MdK

$D_3 D_2$ \ $D_1 D_0$	0 0	0 1	1 1	1 0
0 0	0_0	1_1	1_3	1_2
0 1	1_4	1_5	1_7	1_6
1 1	1_{12}	1_{13}	1_{15}	1_{14}
1 0	1_8	1_9	1_{11}	1_{10}

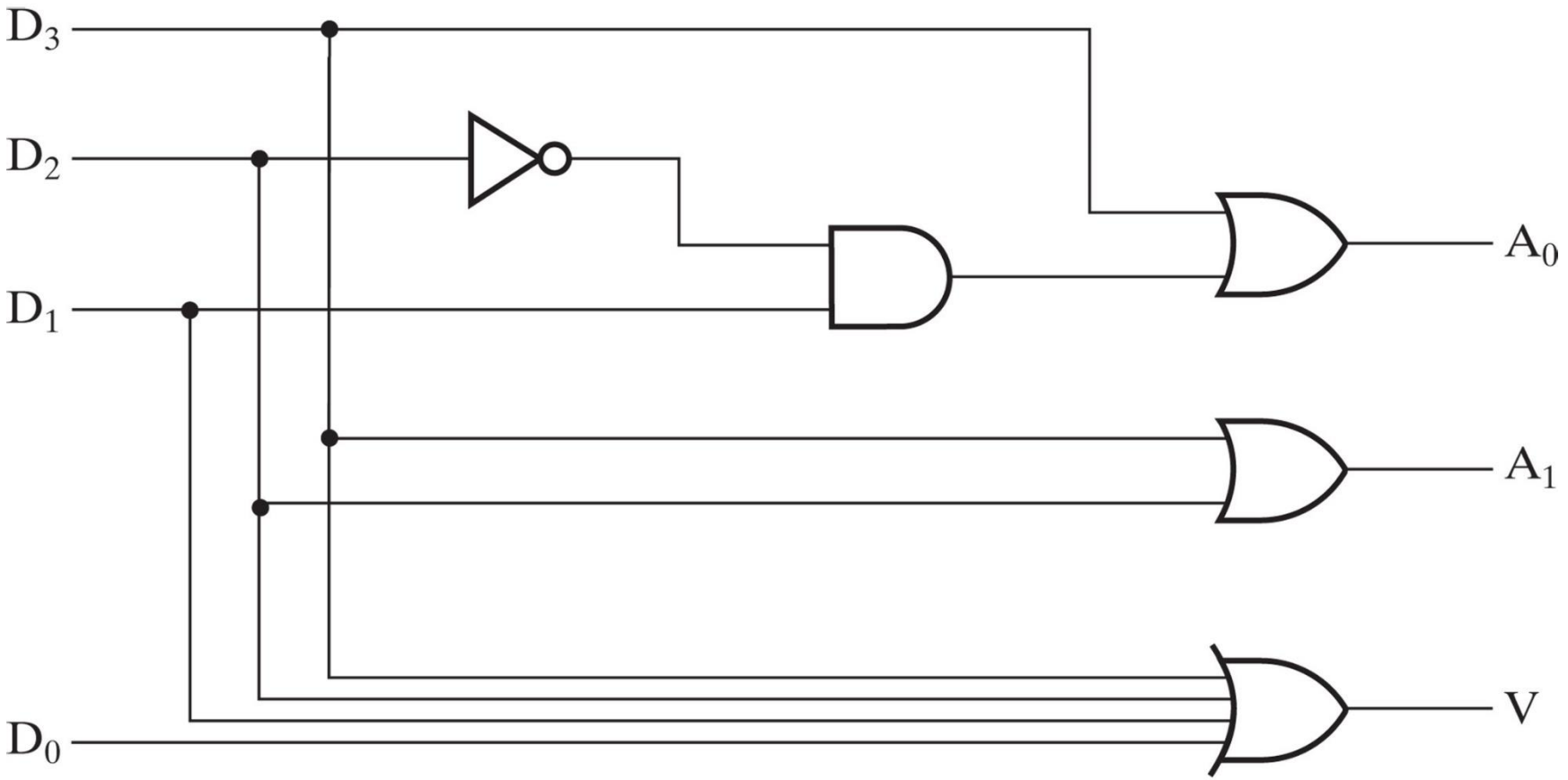
Truth Table of Priority Encoder

Inputs				Outputs		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

$$\overline{V} = \overline{D_3} \cdot \overline{D_2} \cdot \overline{D_1} \cdot \overline{D_0}$$

$$V = D_3 + D_2 + D_1 + D_0$$

Encoder con priorità a 4 ingressi: circuito

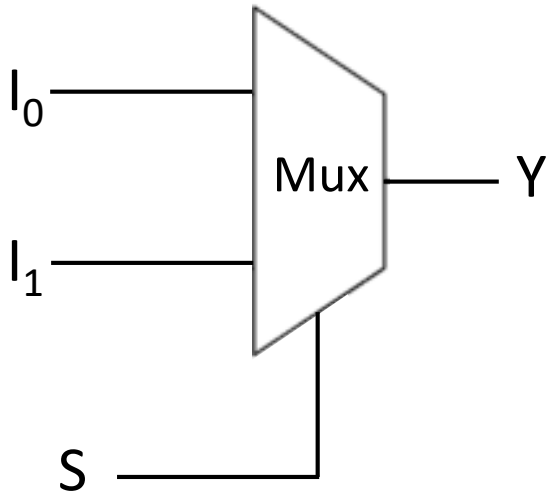


$$A_1 = D_2 + D_3$$

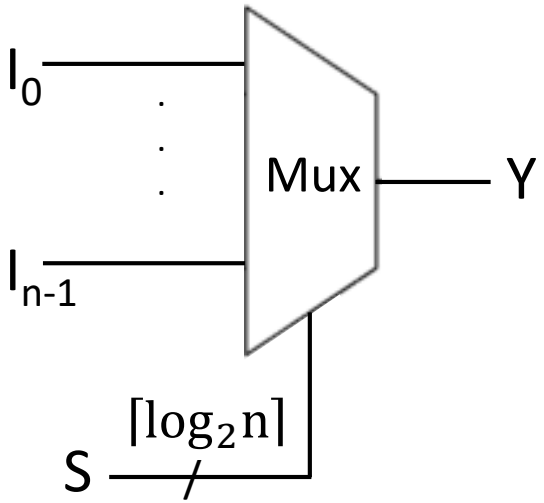
$$A_0 = D_3 + \overline{D_2}\, D_1 \qquad V = D_3 + D_2 + D_1 + D_0$$

Multiplexer

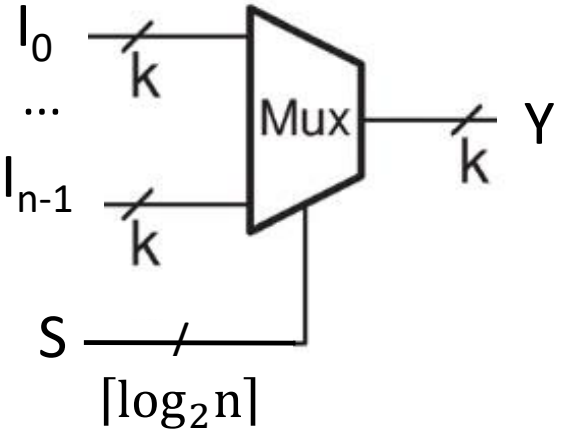
- È un blocco combinatorio per la **selezione dei dati**
- Il multiplexer (MUX) seleziona una delle linee in ingresso, **in base al valore di un segnale di selezione**, e dirige il dato su una singola linea in uscita. L'ingresso di selezione può essere 1-hot o binario



Mux 2-to-1
a 1 bit



Mux n-to-1
a 1 bit

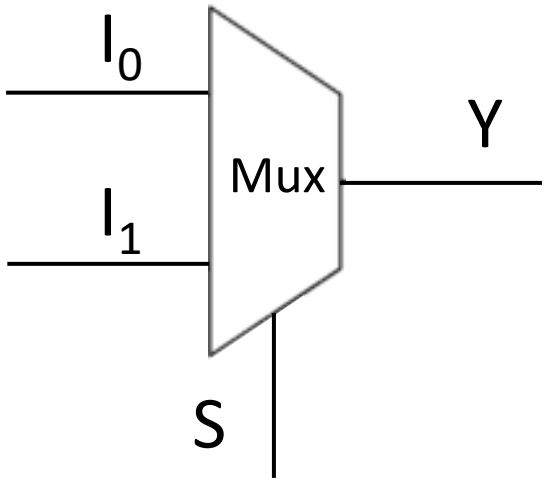


Mux n-to-1
a k bit

Multiplexer 2-to-1

Truth Table for 2–to–1-Line Multiplexer

S	I_0	I_1	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

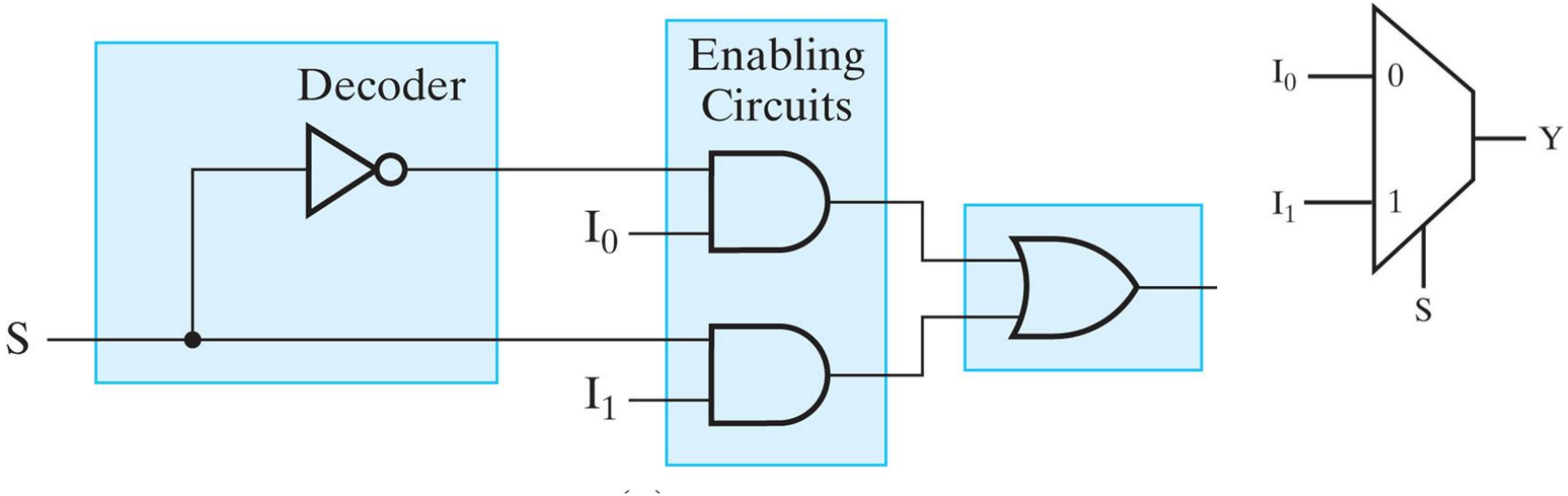


Linee in ingresso: $2 = 2^1$

Ingressi di selezione: 1

Se S='1' seleziona I_1, se S='0' seleziona I_0 : $Y = \bar{S}\ I_0 + S\ I_1$

Multiplexer 2-to-1: implementazione

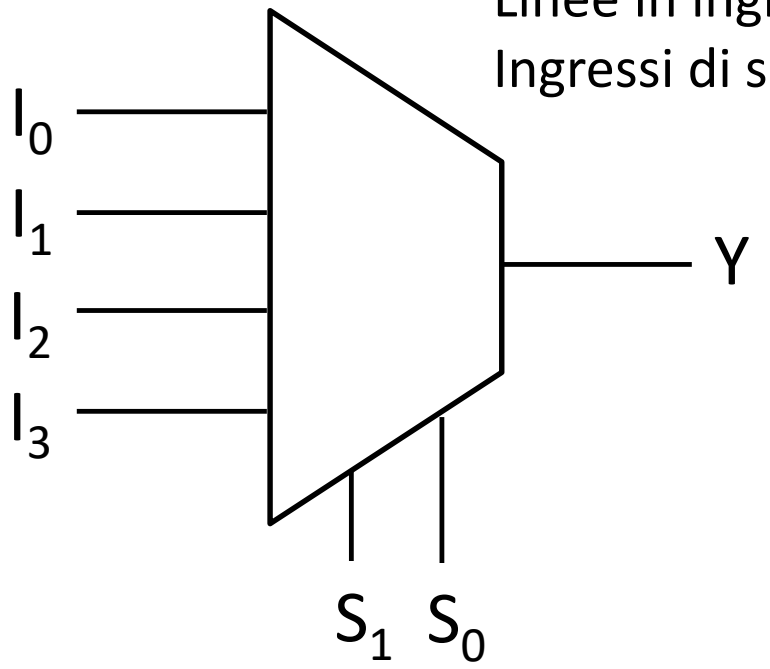


- Un multiplexer 2-to-1 può essere realizzato connettendo le uscite di un decoder 1-to-2 con due circuiti di abilitazione del segnale e connettendo in cascata una porta OR a 2 ingressi
  - Il decoder genera i minterm relativi agli input di selezione. Le porte AND forniscono circuiti di enable che fanno passare o meno gli input I_i in ingresso alla porta OR

Multiplexer 4-to-1

Condensed Truth Table for 4-to-1-Line Multiplexer

S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3



Linee in ingresso: $4 = 2^2$

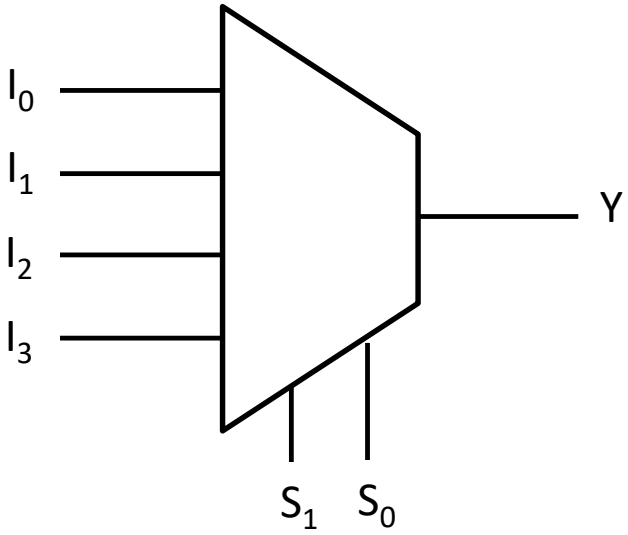
Ingressi di selezione: 2

La funzione da implementare è:

$$Y = \overline{S_1}\,\overline{S_0}\ I_0 + \overline{S_1} S_0\ I_1 + S_1 \overline{S_0}\ I_2 + S_1 S_0 I_3$$

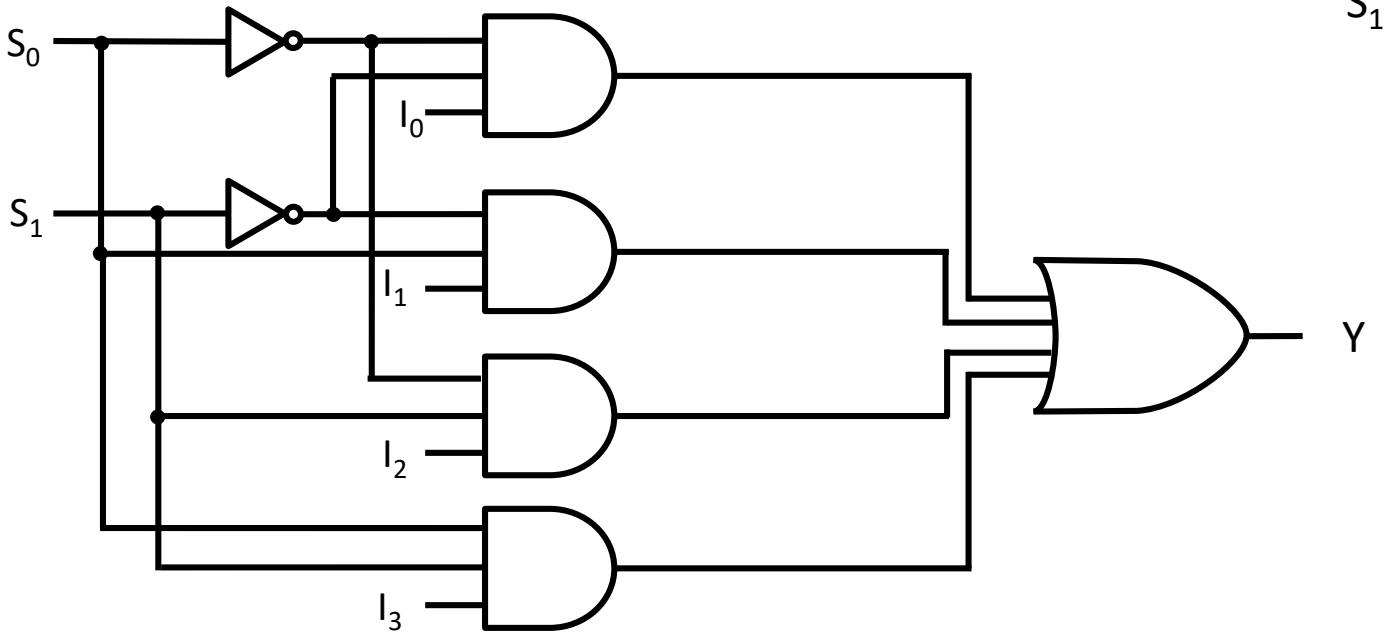
Multiplexer 4-to-1: implementazione 1)

$$Y = \overline{S_1}\,\overline{S_0}\ I_0 + \overline{S_1}\,S_0\ I_1 + S_1\,\overline{S_0}\ I_2 + S_1 S_0 I_3$$



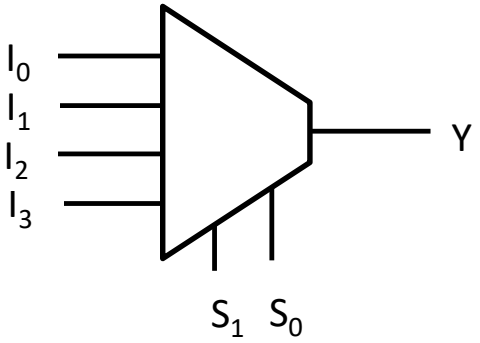
Si può realizzare in modi diversi:

1) Con 2 *inverter*, 4 *AND a 3 input* e *1 OR a 4 input*: **costo numero di ingressi = 18**



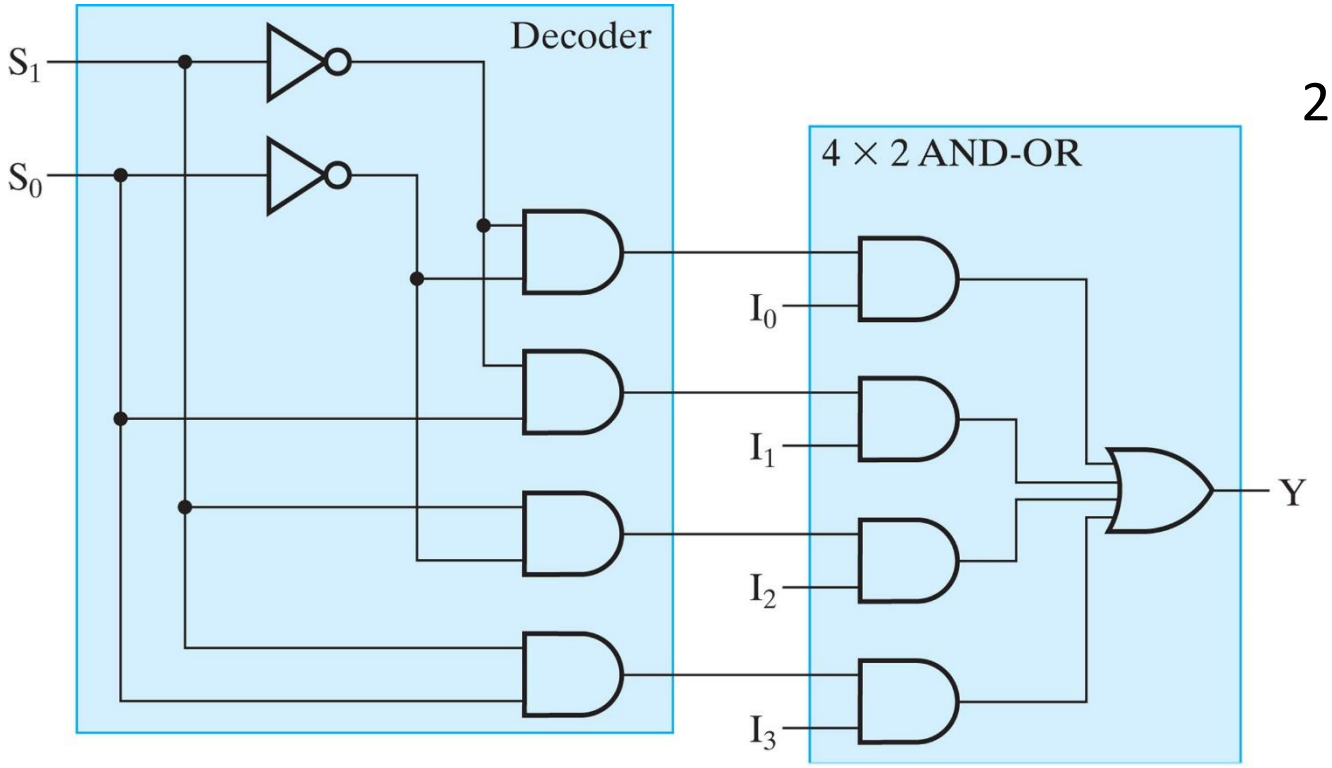
Multiplexer 4-to-1: implementazione 2)

$$Y = \overline{S_1}\,\overline{S_0}\ I_0 + \overline{S_1}S_0\ I_1 + S_1\overline{S_0}\ I_2 + S_1S_0I_3$$



Proprietà associativa:

$$Y = \left(\overline{S_1}\,\overline{S_0}\right) I_0 + \left(\overline{S_1}S_0\right) I_1 + \left(S_1\overline{S_0}\right) I_2 + \left(S_1S_0\right) I_3$$



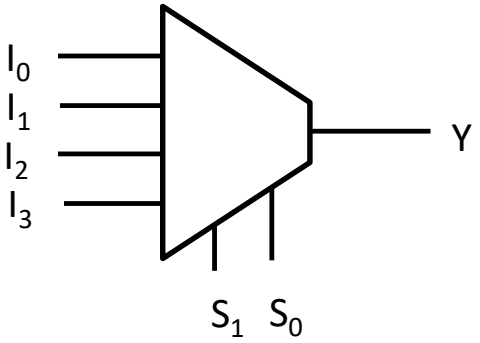
2) Decoder 2-to-4 seguito da 4 AND (circuito di enable) e 1 OR a 4 input: **costo numero di ingressi = 22**

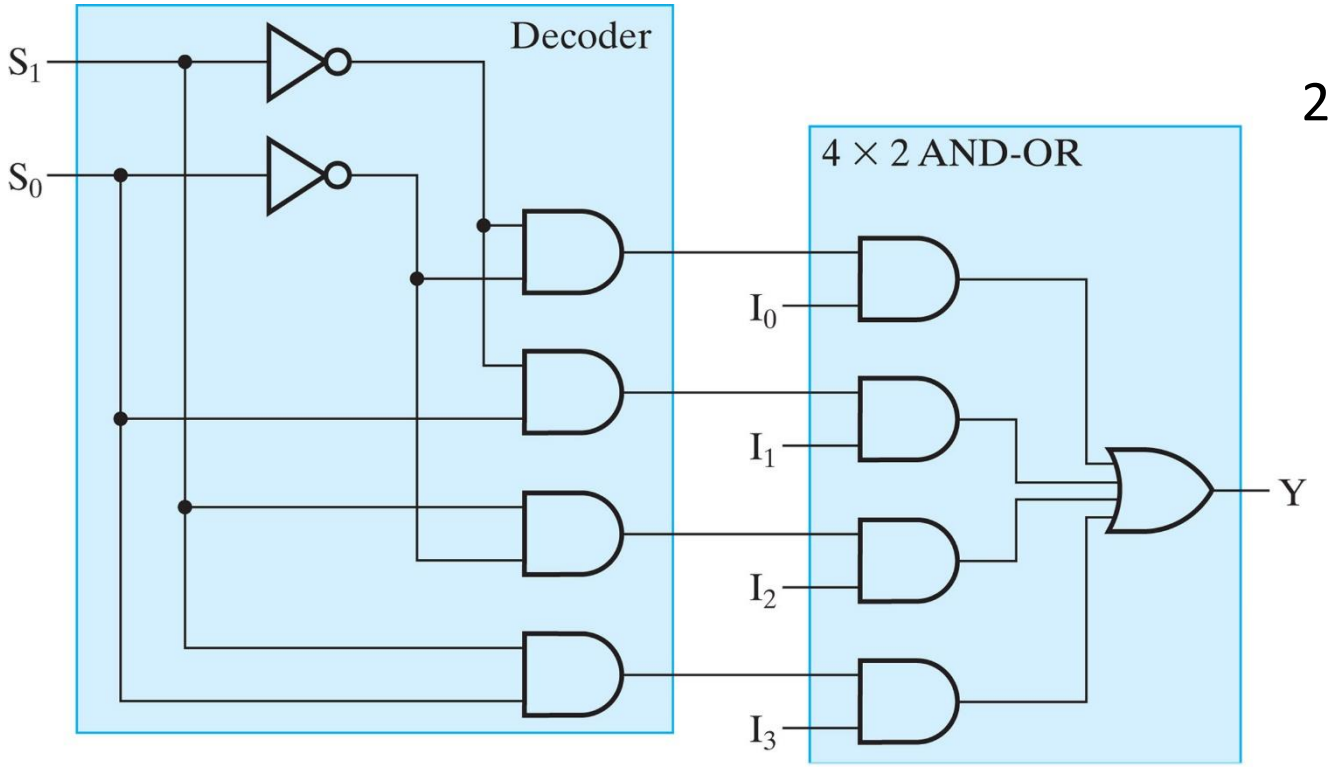
Multiplexer 4-to-1: implementazione 2)

$$Y = \overline{S_1}\,\overline{S_0}\ I_0 + \overline{S_1}S_0\ I_1 + S_1\overline{S_0}\ I_2 + S_1S_0I_3$$

Proprietà associativa:

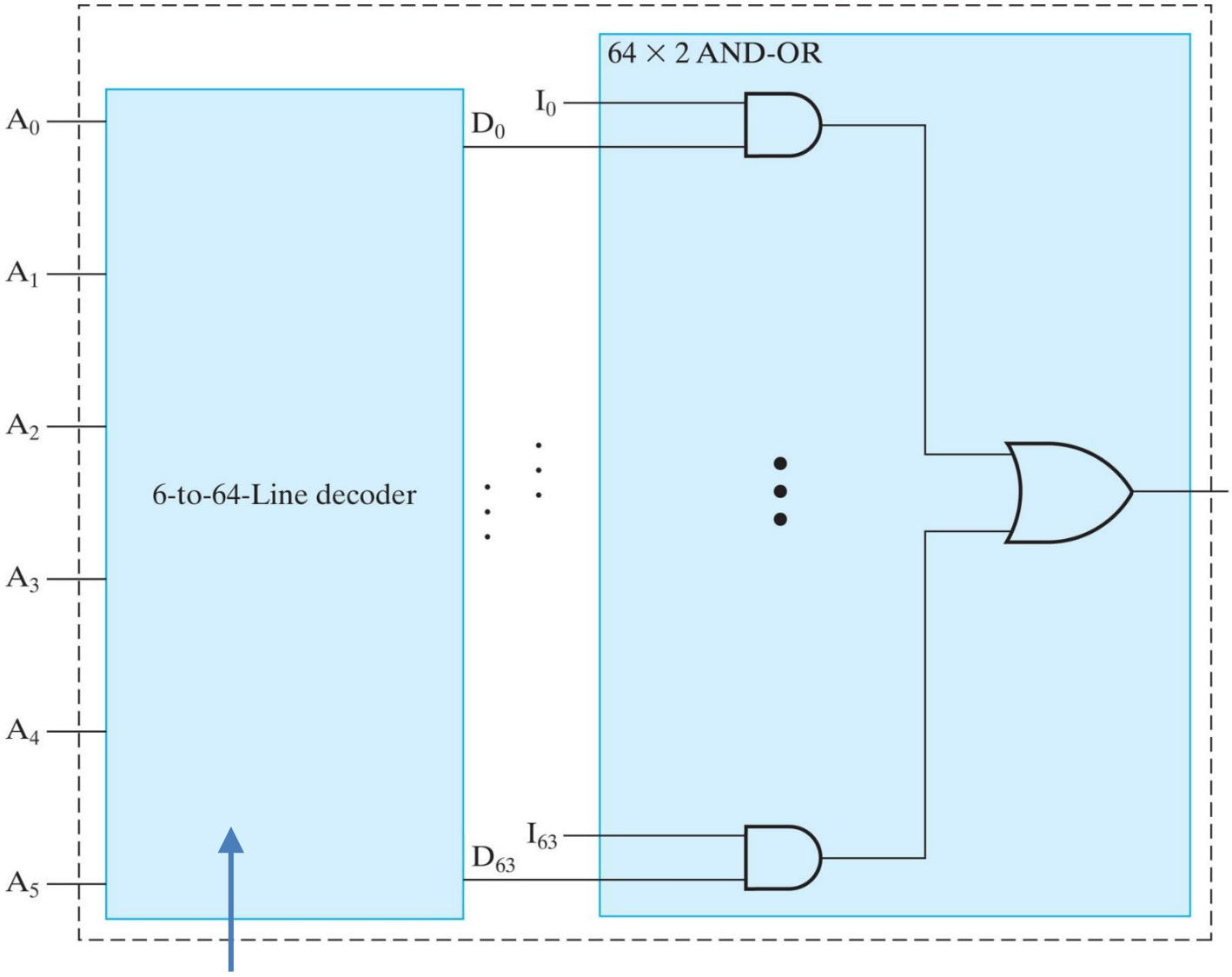
$$Y = \left(\overline{S_1}\,\overline{S_0}\right) I_0 + \left(\overline{S_1}S_0\right) I_1 + \left(S_1\overline{S_0}\right) I_2 + \left(S_1S_0\right) I_3$$





2) Sebbene più costosa in termini di numero di ingressi, può essere **usata in strutture gerarchiche** per realizzare MUX più grandi (2^n dati in ingresso, n ingressi di selezione)

Multiplexer 64-to-1 (impl. tipo 2)



Per selezionare una tra $64=2^6$ linee servono 6 input di selezione

A_0-A_5: ingressi di selezione

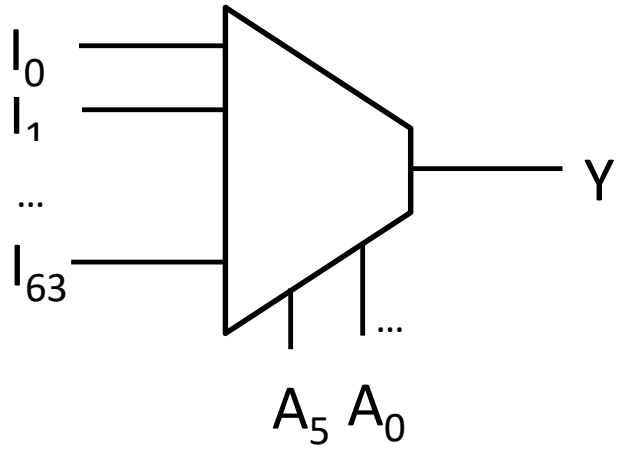
I_0-I_{63}: ingressi di dati

Y: output

6-to-64 decoder, 64 porte AND a 2 ingressi, 1 porta OR a 64 ingressi: costo = **374** (GIC)

Realizzato con struttura gerarchica vista nella slide #31 (costo = 182)

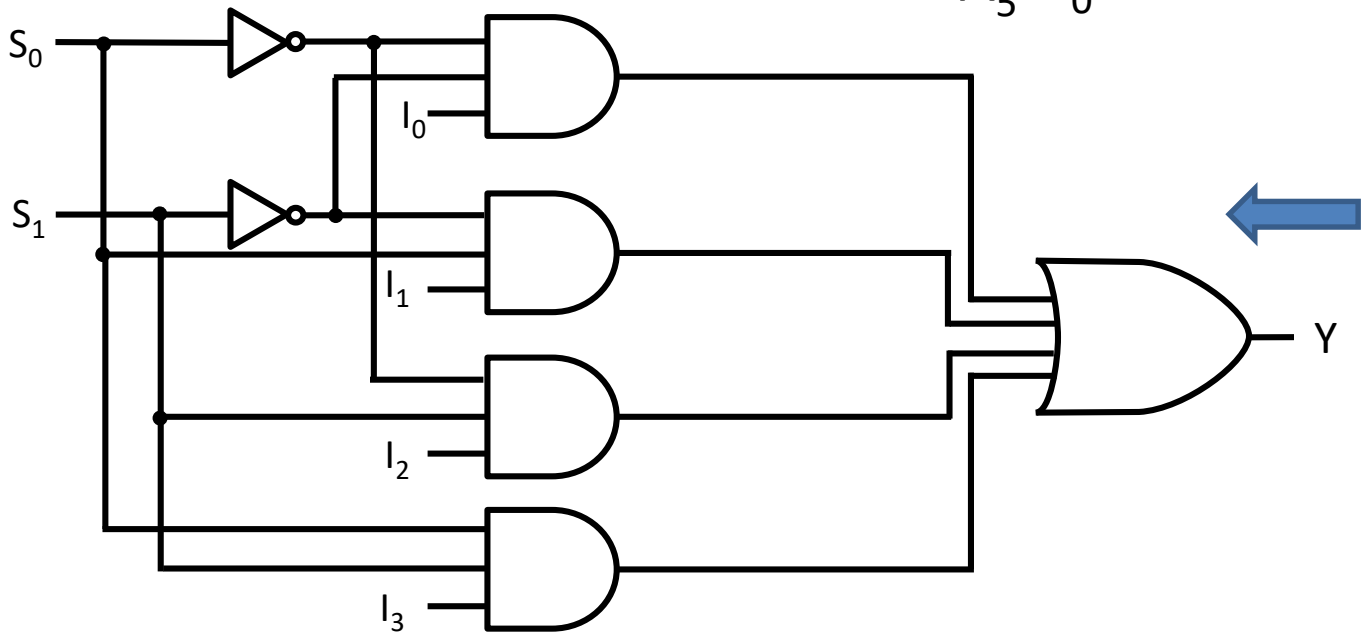
Multiplexer 64-to-1 (impl. tipo 1)



A_0-A_5: selezione

I_0-I_{63}: input

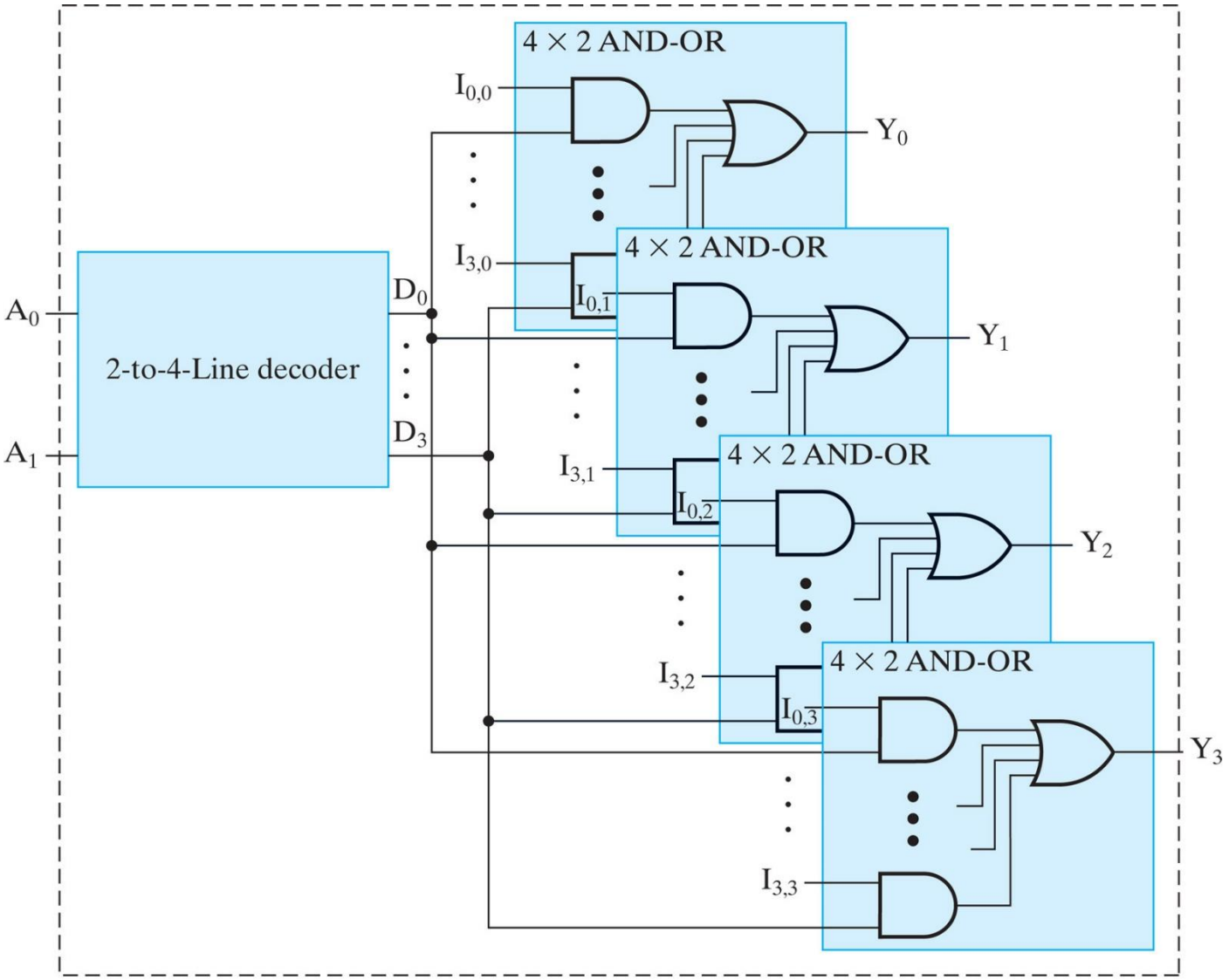
Y: output

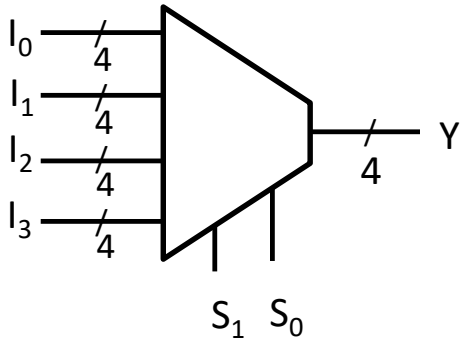


Multiplexer 4 a 1

Se l'avessimo realizzato senza architettura gerarchica (come in slide #56): 6 porte NOT, 64 porte AND a 7 ingressi, 1 porta OR a 64 ingressi, il costo sarebbe stato pari a **518** (GIC)

Quad 4-to-1 Multiplexer





Selettore di vettori

A_0-A_1: selezione

I_0-I_3: input (4 bit)

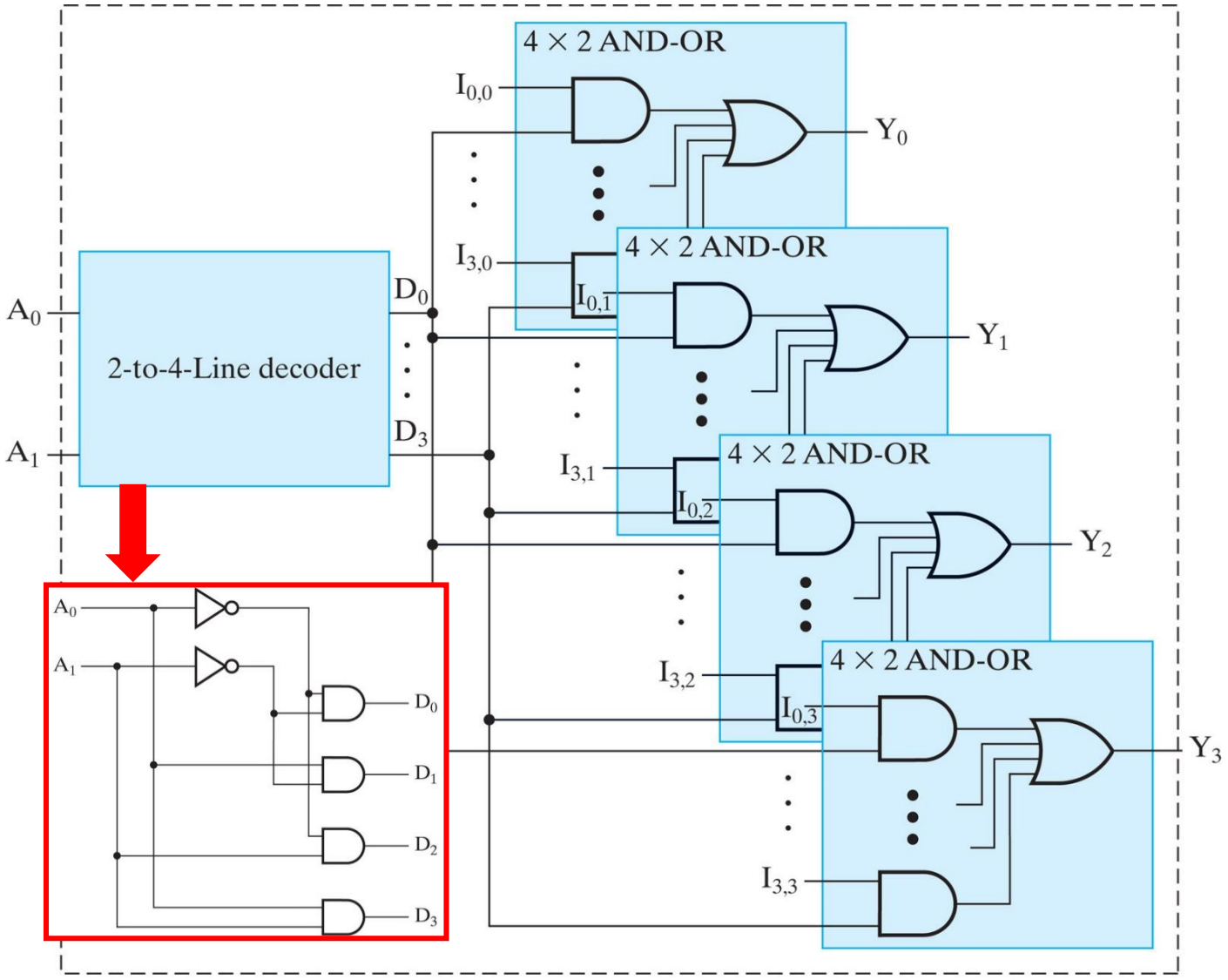
Y: output (4 bit)

4 vettori (4 bit) in

1 vettore (4 bit) out

2 input di selezione

Quad 4-to-1 Multiplexer



Selettore di vettori

A_0-A_1: selezione

I_0-I_3: input (4 bit)

Y: output (4 bit)

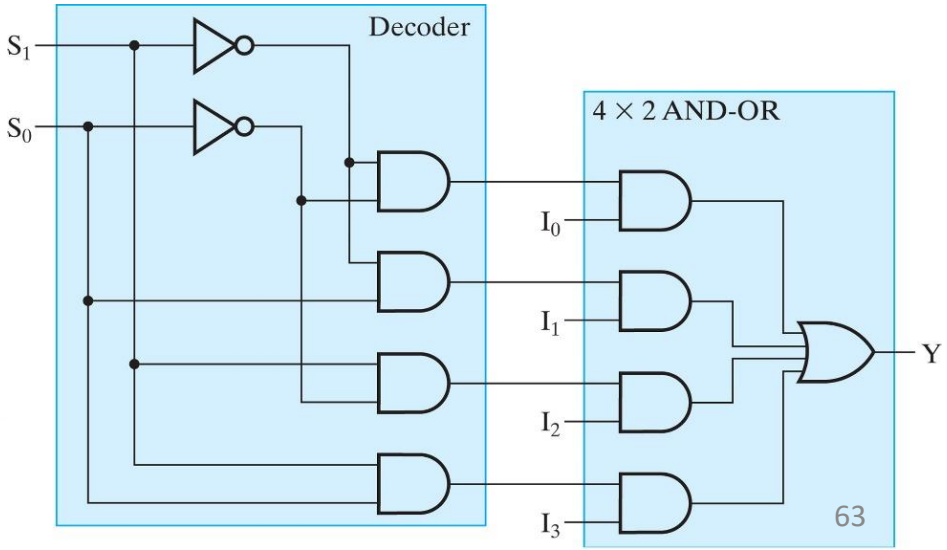
Decoder 2-to-4, seguito da 4 x 4 AND a 2 ingressi + OR a 4 ingressi

Il costo in termini di numero di ingressi è 10 + 48 = 58

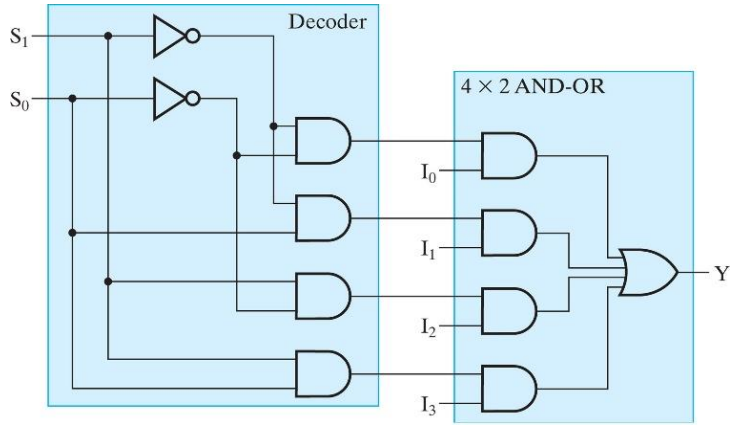
Multiplexer 4-to-1: Structural VHDL (1/2)

```
-- 4-to-1-Line Multiplexer: Structural VHDL Description
-- (See Figure 3-25 for logic diagram)
library ieee, lcdf_vhdl;
use ieee.std_logic_1164.all, lcdf_vhdl.func_prims.all;
entity multiplexer_4_to_1_st is
   port (S: in std_logic_vector(0 to 1);
         I: in std_logic_vector(0 to 3);
         Y: out std_logic);
end multiplexer_4_to_1_st;

architecture structural_2 of multiplexer_4_to_1_st is
   component NOT1
      port(in1: in std_logic;
           out1: out std_logic);
   end component;
   component AND2
      port(in1, in2: in std_logic;
           out1: out std_logic);
   end component;
```



Multiplexer 4-to-1: Structural VHDL (2/2)

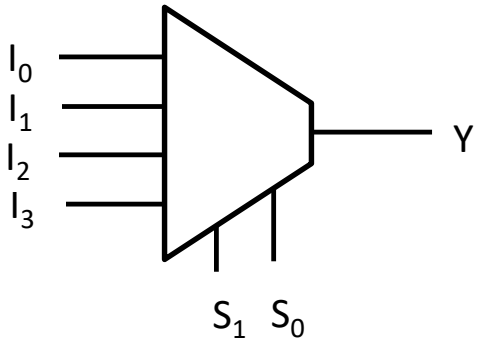



```
component OR4
   port(in1, in2, in3, in4: in std_logic;
        out1: out std_logic);
end component;
signal S_n: std_logic_vector(0 to 1);
signal D, N: std_logic_vector(0 to 3);
begin
   g0: NOT1 port map (S(0), S_n(0));
   g1: NOT1 port map (S(1), S_n(1));
   g2: AND2 port map (S_n(1), S_n(0), D(0));
   g3: AND2 port map (S_n(1), S(0), D(1));
   g4: AND2 port map (S(1), S_n(0), D(2));
   g5: AND2 port map (S(1), S(0), D(3));
   g6: AND2 port map (D(0), I(0), N(0));
   g7: AND2 port map (D(1), I(1), N(1));
   g8: AND2 port map (D(2), I(2), N(2));
   g9: AND2 port map (D(3), I(3), N(3));
   g10: OR4 port map (N(0), N(1), N(2), N(3), Y);
end structural_2;
```

Multiplexer 4-to-1: Dataflow VHDL con logica prioritaria "when-else"

```
-- 4-to-1-Line Mux: Conditional Dataflow VHDL Description
-- Using When-Else (See Table 3-8 for function table)
library ieee;
use ieee.std_logic_1164.all;
entity multiplexer_4_to_1_we is
   port (S : in std_logic_vector(1 downto 0);
         I : in std_logic_vector(3 downto 0);
         Y : out std_logic);
end multiplexer_4_to_1_we;

architecture function_table of multiplexer_4_to_1_we
begin
   Y <= I(0) when S = "00" else
        I(1) when S = "01" else
        I(2) when S = "10" else
        I(3) when S = "11" else
        'X';
end function_table;
```



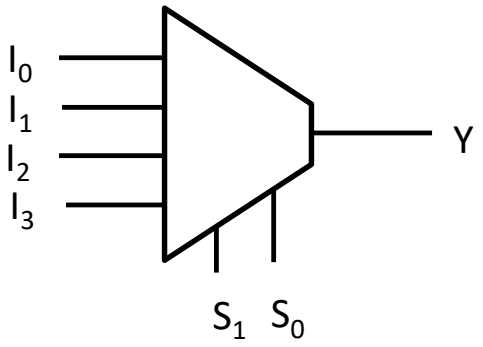
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

else 'X' è necessario: oltre a '0' e '1', std_logic_value può assumere altri valori ('Z', '-', ...) e vogliamo specificare cosa accade anche in quelle condizioni (logica combinatoria)

Multiplexer 4-to-1: Dataflow VHDL con logica parallela "with-select"

```
--4-to-1-Line Mux: Conditional Dataflow VHDL Description
Using with Select (See Table 3-8 for function table)
library ieee;
use ieee.std_logic_1164.all;
entity  multiplexer_4_to_1_ws is
   port (S : in std_logic_vector(1 downto 0);
         I : in std_logic_vector(3 downto 0);
         Y : out std_logic);
end multiplexer_4_to_1_ws;

architecture function_table_ws of multiplexer_4_to_1_ws is
begin
   with  S select
      Y <= I(0) when "00",
           I(1) when "01",
           I(2) when "10",
           I(3) when "11",
           'X'when others;
end function_table_ws;
```



S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

'X' when others: quando si usa il costrutto with-select è necessario enumerare tutti i casi possibili

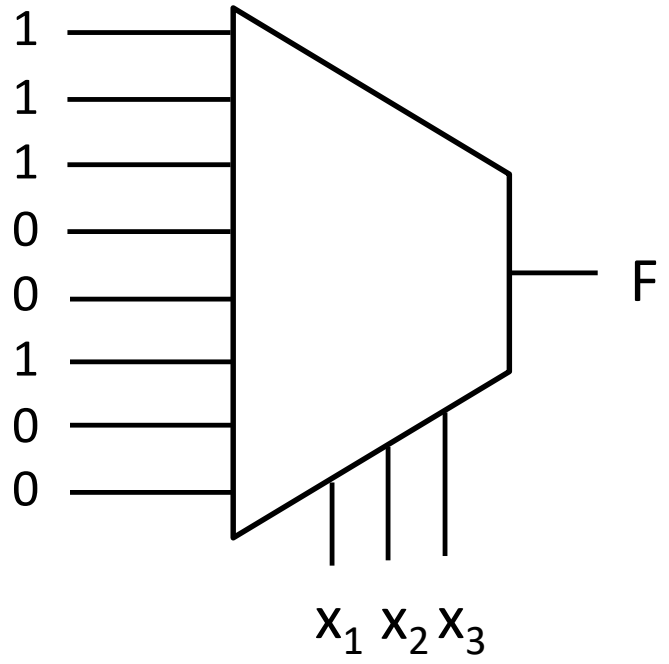
Circuiti combinatori basati su MUX

- Un generico **circuito combinatorio può essere realizzato tramite un multiplexer** (in modo analogo a quanto visto per il decoder)
- Possibile realizzazione: qualunque **funzione booleana di n variabili può essere implementata con un 2^n-to-1 multiplexer** (2^n ingressi di dati, n ingressi di selezione), fissando opportunamente a ‘0’ o ‘1’ i valori degli ingressi I_i

Esempio: Funzione booleana a n variabili con MUX 2^n-to-1

x_1	x_2	x_3	F (x_1 , x_2 , x_3)
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

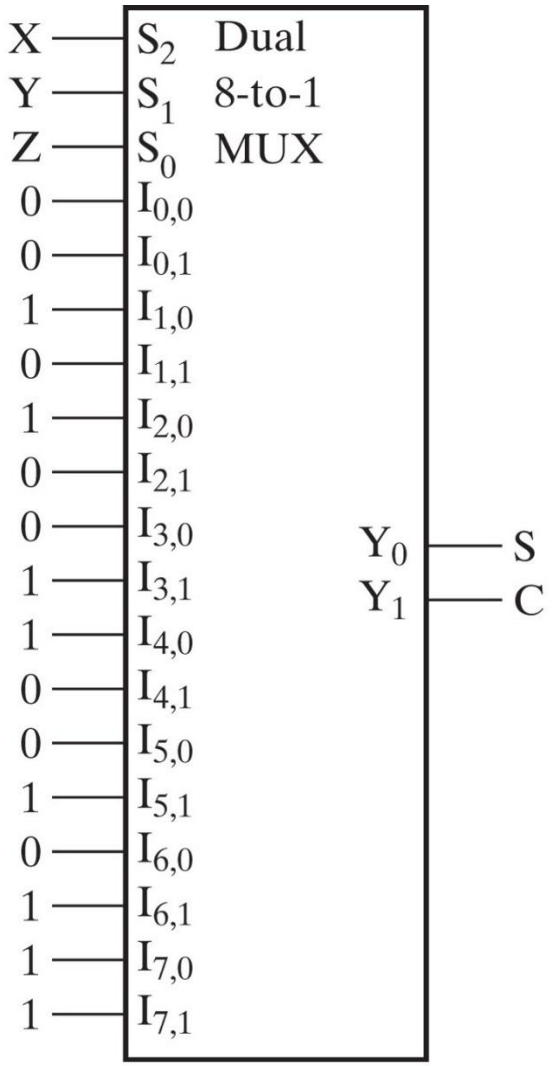
n = 3 → usiamo un MUX 8-to-1, applicando i 3 ingressi della funzione ai segnali di selezione: x_1, x_2, x_3 seleziona l'apposita riga della tabella di verità



Esempio 3-15: Sommatore a 1 bit con multiplexer con dual MUX 8-to-1

Truth Table for 1-Bit Binary Adder

X	Y	Z	C	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



Implementazione con dual MUX 8-to-1: il segnale di selezione X Y Z seleziona l'apposita riga della tabella di verità

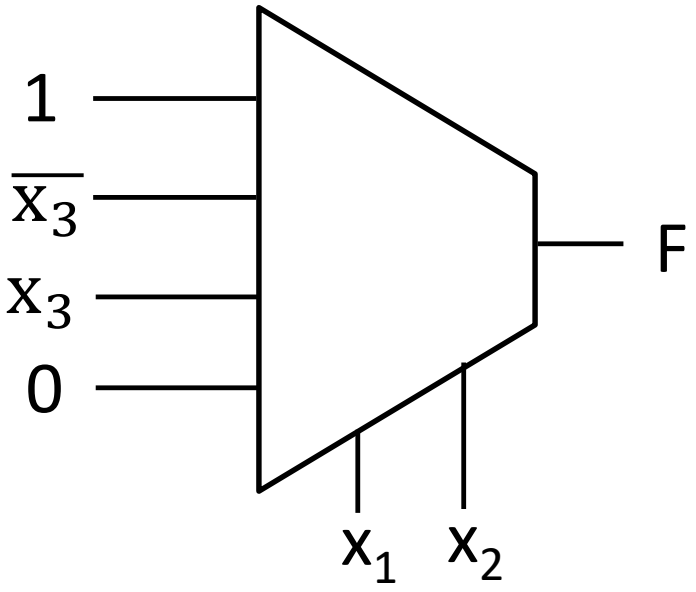
Circuiti combinatori basati su MUX

- Esiste un'altra possibile realizzazione, più efficente
- Possiamo ottimizzare il numero di ingressi del multiplexer, applicando solo una parte degli ingressi della funzione agli input di selezione
  - Nelle slide successive realizzeremo le stesse funzioni viste nelle slide precedenti, utilizzando dei MUX più piccoli

Esempio: Funzione booleana a n variabili con MUX 2^{n-1}-to-1

x_1	x_2	x_3	F (x_1 , x_2 , x_3)
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

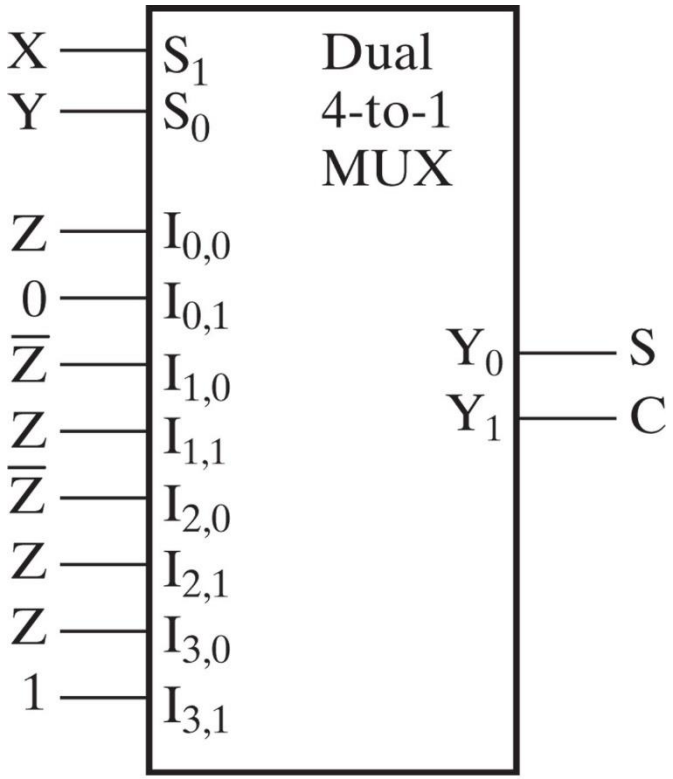
n = 3: Usiamo ora un MUX 4-to-1, applicando solo 2 degli ingressi della funzione (x_1, x_2) ai segnali di selezione: gli input del MUX saranno '0', '1', x_3 o $\overline{x_3}$



Esempio 3-16: Sommatore a 1 bit con multiplexer con dual MUX 4-to-1

Truth Table for 1-Bit Binary Adder

X	Y	Z	C	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



Implementazione con un dual MUX 4 a 1: due degli ingressi X,Y sono collegati ai 2 ingressi di selezione, gli input del MUX saranno ‘0’, ‘1’, Z o $\overline{Z}$, a seconda dei valori nella tabella di verità

Riepilogo

- Definizione di **circuito combinatorio**: le uscite dipendono solo dal valore corrente degli ingressi, non c'è memoria
- Importanza dell'**approccio gerarchico** (regolarità di un circuito)
- Circuito di abilitazione di un segnale (**enabling**) permette ad un segnale di essere o meno trasmesso in uscita
- **Decoder**: genera 2^n minterm a partire da n variabili in ingresso
- **Encoder**: funzione inversa del decoder, genera la codifica binaria a n bit a partire da un ingresso 1-hot da 2^n bit
- **Multiplexer**: seleziona un ingresso, in base al valore di un segnale di selezione e dirige il dato su una singola linea in uscita
- Una generica funzione combinatoria può essere realizzata tramite un decoder o multiplexer

Disclaimer

Figures from *Logic and Computer Design Fundamentals*, Fifth Edition, GE Mano | Kime | Martin