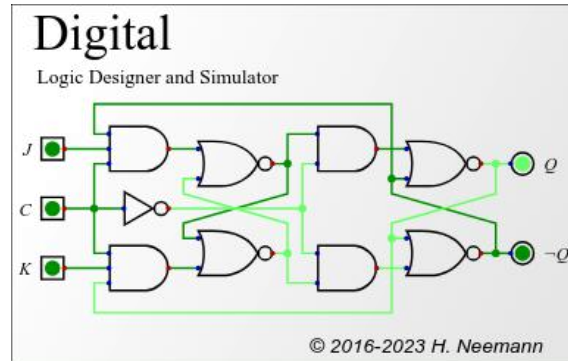


Circuiti combinatori

Alessandro Pellegrini
a.pellegrini@ing.uniroma2.it

Lo “stack software”

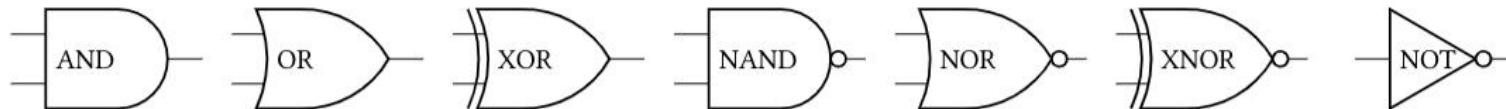
- È possibile utilizzare il simulatore Digital per realizzare e simulare semplici circuiti digitali



- <https://github.com/hneemann/Digital>
- Implementato in Java, funziona su tutti i sistemi
- Le istruzioni per l'uso sono sulla pagina del progetto

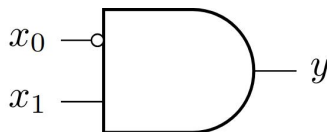
Circuiti logici (o di commutazione)

- I circuiti logici sono reti di componenti che accettano variabili booleane in input e restituiscono variabili booleane in output.
- Per la loro sintesi, è utile affidarsi all'algebra booleana
- Gli operatori booleani sono implementati in hardware da circuiti chiamati *porte logiche*
- Tali porte vengono *astratte* con dei simboli standard:

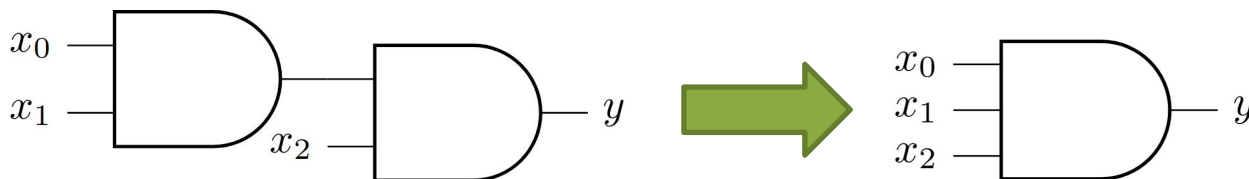


Negazioni e porte a più ingressi

- Circuitualmente è più efficiente inserire il calcolo della negazione degli input direttamente nelle porte logiche:



- È possibile costruire porte a più ingressi (fino ad un certo limite, dovuto alla tecnologia costruttiva)



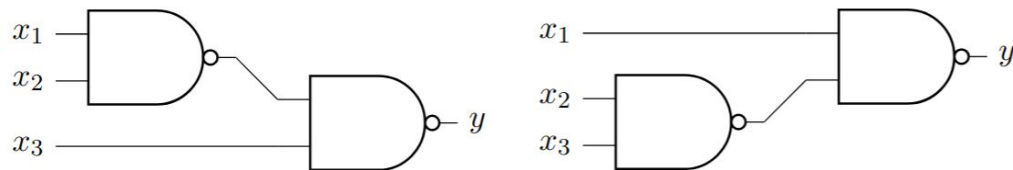
- La maggior parte degli operatori è associativo e commutativo

NAND a più ingressi

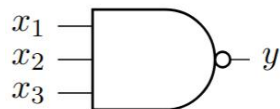
- Abbiamo già visto che

$$x_1|(x_2|x_3) = \overline{x_1} + x_2x_3 \neq x_1x_2 + \overline{x_3} = (x_1|x_2)|x_3$$

- Il che porta alla non equivalenza dei circuiti

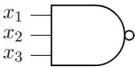


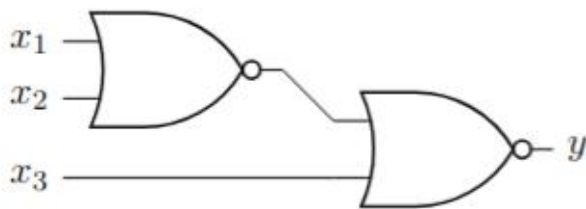
- Tuttavia è possibile costruire la seguente porta logica



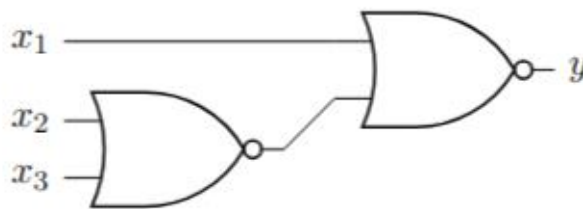
- Qual è il suo significato?

NAND a più ingressi

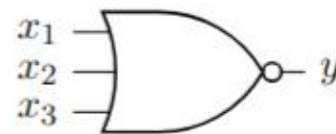
- Possiamo estendere $x_1|x_2|x_3 = \overline{x_1 \cdot x_2 \cdot x_3}$ costruendo quindi un operatore NAND associativo
- Il circuito  calcola una funzione differente dal NAND dell'algebra booleana!
- Lo stesso ragionamento può essere applicato all'operatore duale NOR



Non associativo: $y = \overline{x_3}(x_1 + x_2)$



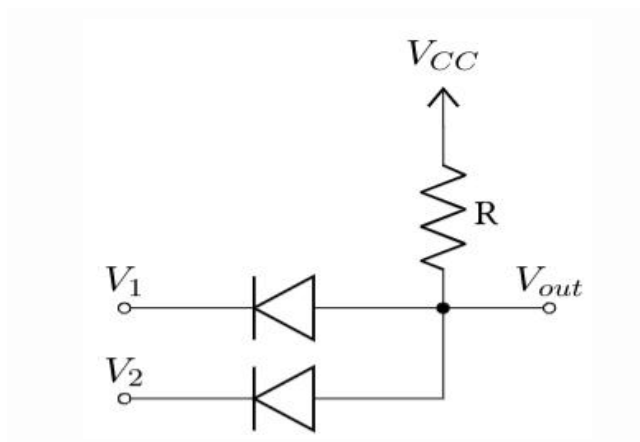
Non associativo: $y = \overline{x_1}(x_2 + x_3)$



Associativo: $y = \overline{x_1 + x_2 + x_3}$

Porte logiche a diodi

- Immaginando un diodo ideale, è possibile costruire un *selettore di minimo* secondo il circuito in figura
- Poiché operiamo con circuiti di commutazione, assumiamo che ci siano solo due livelli ammissibili di tensione: V_L e V_H

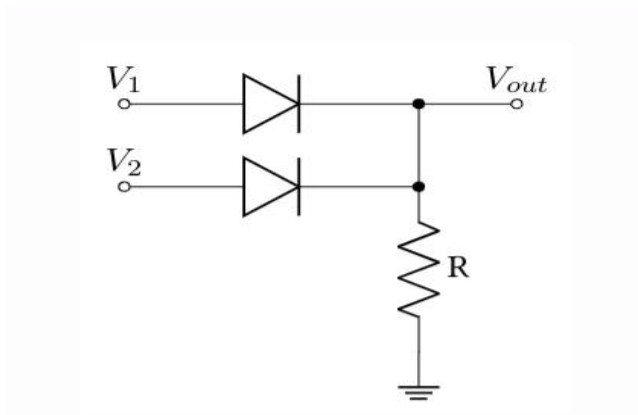


V_{out}	V_1	V_2
V_L	V_L	V_L
V_L	V_L	V_H
V_L	V_H	V_L
V_H	V_H	V_H

- Tale circuito implementa quindi una porta logica di tipo AND

Porte logiche a diodi

- Immaginando un diodo ideale, è possibile costruire un *selettore di massimo* secondo il circuito in figura
- Poiché operiamo con circuiti di commutazione, assumiamo che ci siano solo due livelli ammissibili di tensione: V_L e V_H



V_{out}	V_1	V_2
V_L	V_L	V_L
V_H	V_L	V_H
V_H	V_H	V_L
V_H	V_H	V_H

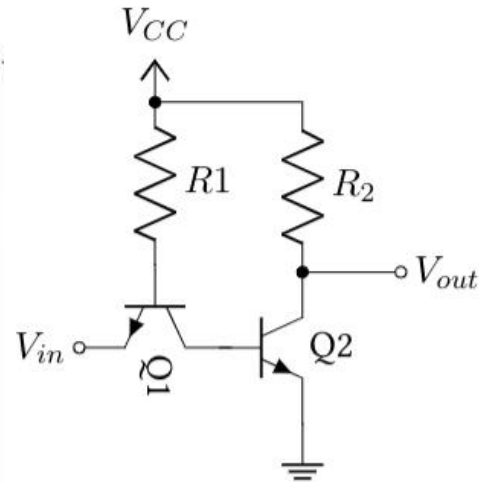
- Tale circuito implementa quindi una porta logica di tipo OR

Problemi delle porte logiche a diodi

- Le porte basate su diodi provocano un'attenuazione del segnale
- Collegando in cascata più porte, quindi, il segnale subirà un'attenuazione progressiva che può distruggere la differenza tra V_L e V_H
- Tale fenomeno ha portato al progressivo abbandono di queste porte negli anni '60 a favore delle *porte a transistor*

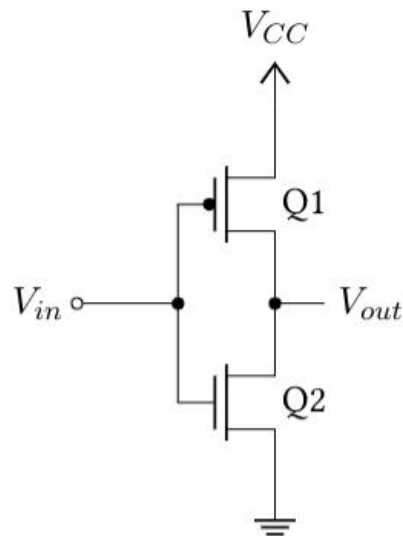
Logica TTL

- La Transistor/Transistor Logic si basa sull'uso di alcuni transistor per la realizzazione della funzione di commutazione e di alcuni transistor per l'amplificazione del segnale
- In questo modo, si elimina il fenomeno dell'attenuazione
- Si basano sul concetto di *resistenze di pull up*
- Esempio: inverter TTL
(implementa una porta NOT)

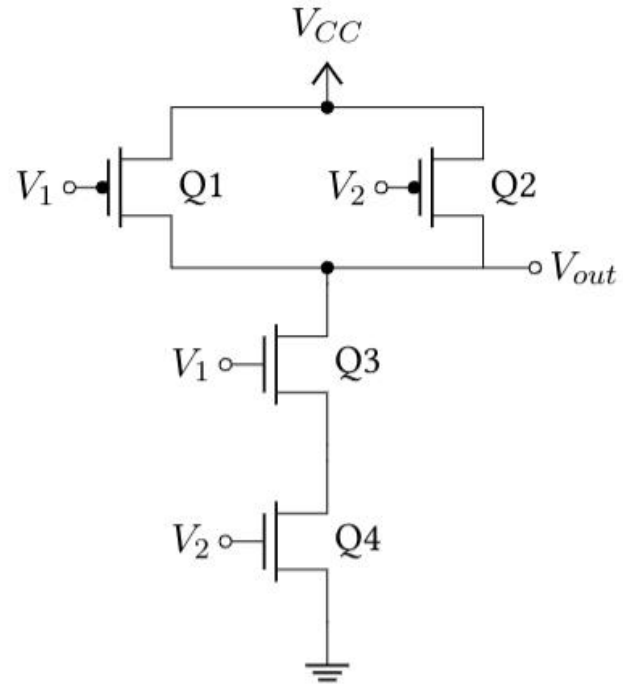
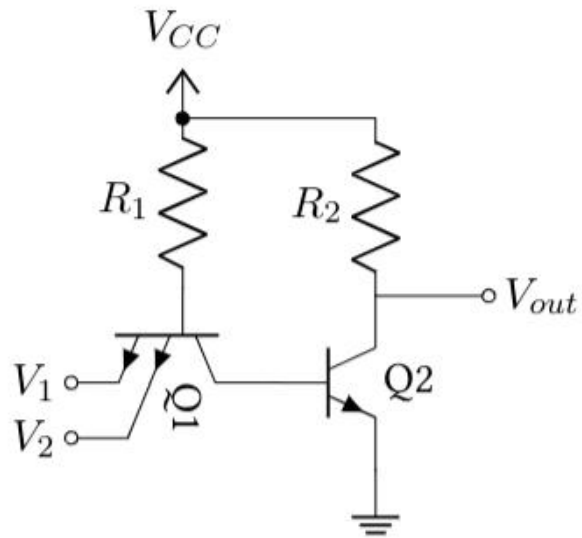


Logica CMOS (circa 1980)

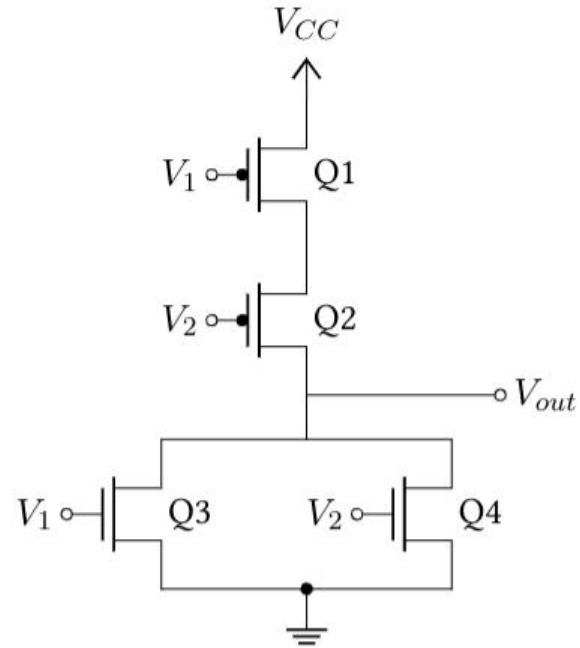
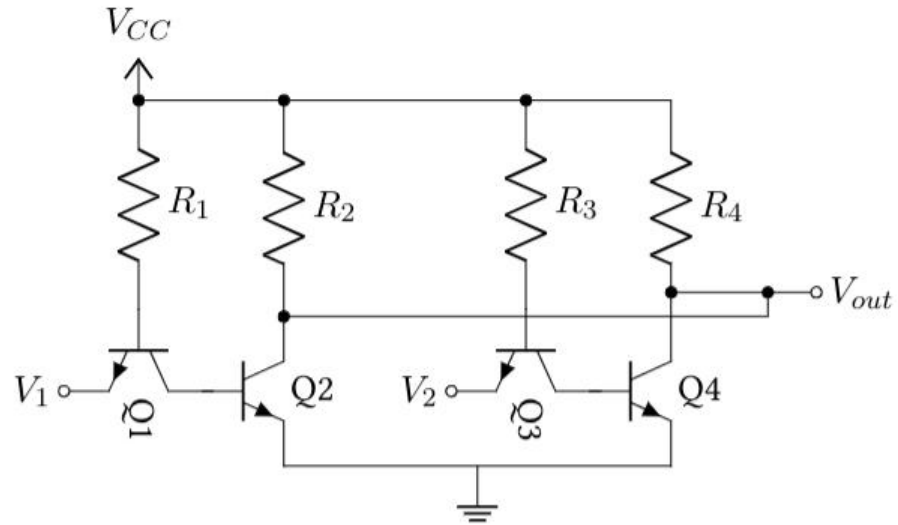
- Si basa sull'uso di transistor pMOS e nMOS nella stessa rete
- Il vantaggio è che l'area utilizzata è estremamente piccola
- La rete pMOS si comporta da *pull up*, mentre la rete nMOS implementa la parte di *pull down*
- Esempio: inverter CMOS



Porta NAND



Porta NOR

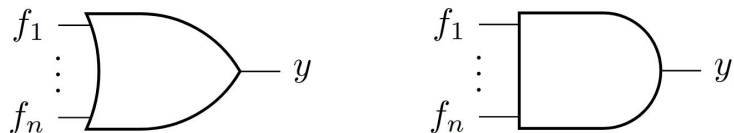


Riduzione del costo nelle operazioni AND/OR

- Immaginiamo di avere una funzione composta, del tipo:

$$y = \sum_{i=1}^n f_i \text{ oppure } y = \prod_{i=1}^n f_i$$

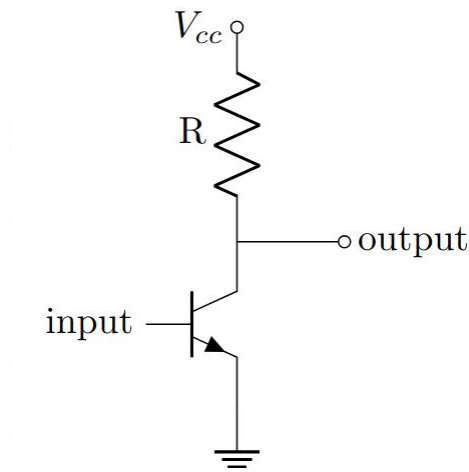
- Il circuito logico risultante è del tipo:



- Il problema è nei “puntini”
 - Il numero di transistor per aumentare il numero di ingressi cresce
 - Non è detto che si conosca a priori il numero di input da considerare
 - Se gli input sono tanti, l’accumulo di corrente potrebbe danneggiare i dispositivi elettronici

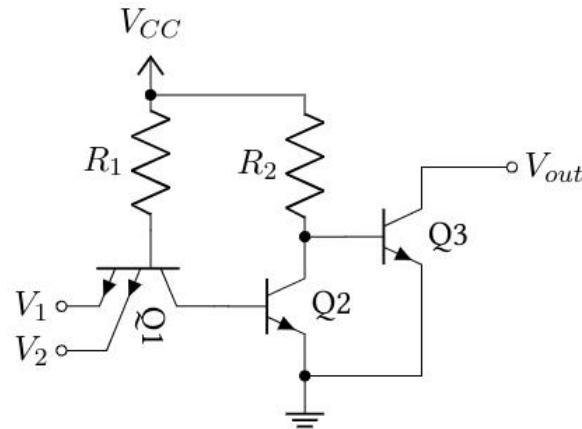
Tecnologia in Open Collector

- Le porte open collector usano un transistor aggiuntivo in cui il collettore è connesso all'uscita della porta
- L'uscita funziona quindi come un circuito aperto o come un circuito connesso a massa
- Viene utilizzata una resistenza di pull up che fa salire la tensione quando il circuito è chiuso (si lavora in *logica negata*)



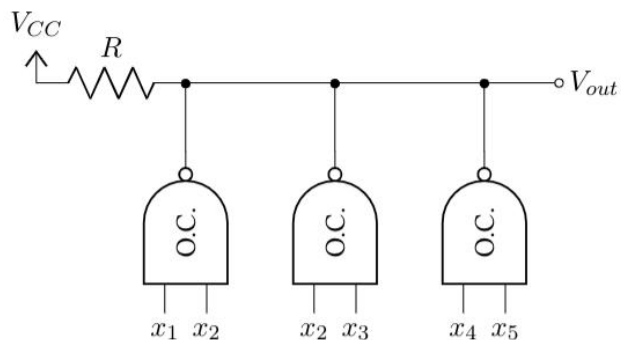
Tecnologia in Open Collector

- Per indicare un operatore logico in open collector, viene utilizzato il simbolo della porta logica negata, marcato con “OC”
- Esempio: NAND in Open Collector
- La resistenza di pull up diventa esterna alla porta



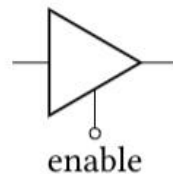
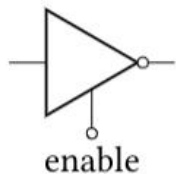
Uso dell'Open Collector

- Le porte in open collector vengono usate quando c'è la necessità di collegare insieme più porte che possono fornire corrente in uscita contemporaneamente
- L'accumulo di corrente in uscita, in caso di un numero elevato di componenti, può produrre danni fisici al circuito
- Con l'uso dell'open collector la quantità di corrente che fluisce nel circuito non è più funzione del numero di porte con l'uscita attiva



Buffer Three-state

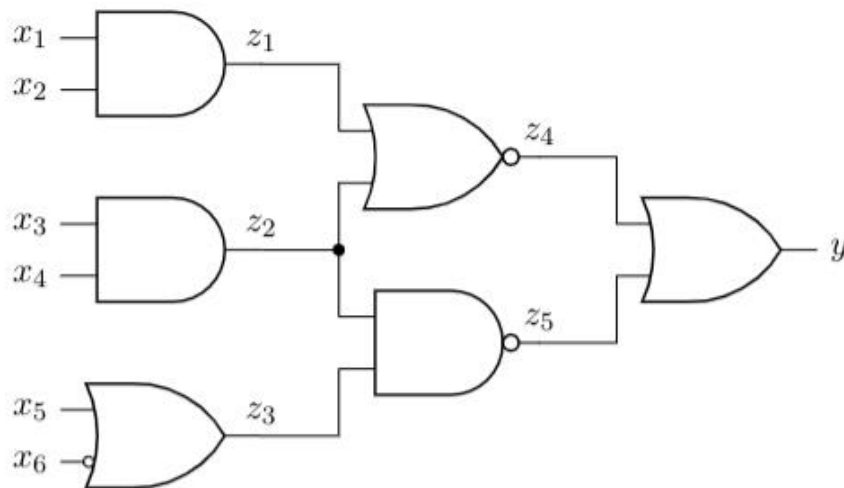
- Il buffer three-state è un circuito che viene utilizzato per collegare tra loro parti di un circuito complesso
- È un circuito a tre canali:
 - un segnale di ingresso
 - un segnale di uscita
 - un segnale di controllo
- Il segnale di uscita è uguale al segnale di ingresso se il segnale di controllo è attivo, altrimenti l'uscita è ad alta impedenza
- Si comporta quindi da interruttore



Dalle funzioni booleane ai circuiti e viceversa

- Le porte logiche che abbiamo introdotto permettono di implementare in hardware qualsiasi funzione booleana
- È interessante capire come passare da una funzione booleana a un circuito e viceversa
- Per il passaggio dalla funzione al circuito:
 - Si considerano i termini più interni (per precedenza) e si realizza quella parte della funzione come circuito
 - Le uscite delle porte così definite vengono usate come input delle funzioni (porte) più esterne
- Per il passaggio dal circuito alla funzione:
 - Si assegna un nome all'uscita di ogni porta
 - Queste variabili ausiliarie vengono via via eliminate fino a quando non si ha un'espressione che usa solo le variabili di input

Dal circuito alla funzione



- $y = z_4 + z_5$
- $y = (z_1 \downarrow z_2) + (z_2 | z_3)$
- $y = ((x_1 \cdot x_2) \downarrow (x_3 \cdot x_4)) + ((x_3 \cdot x_4) | (x_5 + \overline{x_6}))$
- $y = \overline{x_1 x_2 + x_3 x_4} + \overline{x_3 x_4 (x_5 + \overline{x_6})}$

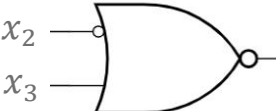
Dalla funzione al circuito

- Consideriamo la funzione:

$$y = x_1 x_2 ((\overline{x_2} \downarrow x_3) + (\overline{x_1}(x_1 + \overline{x_3})))$$

- Per costruire il circuito rispettiamo le precedenze
 - partiamo dai termini più interni e costruiamo le parti corrispondenti del circuito

- $(x_1 + \overline{x_3})$ 

- $(\overline{x_2} \downarrow x_3)$ 

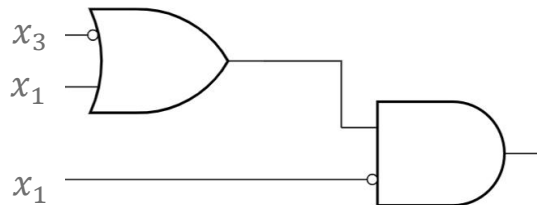
Dalla funzione al circuito

- Consideriamo la funzione:

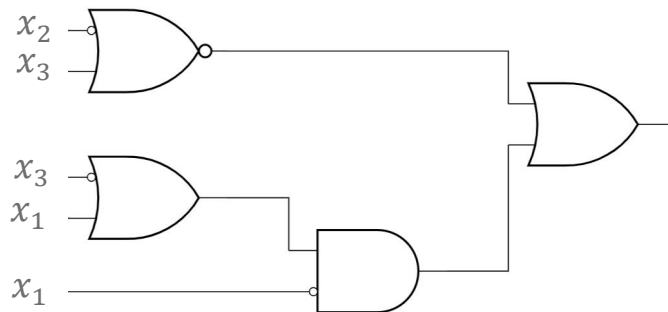
$$y = x_1 x_2 ((\overline{x_2} \downarrow x_3) + (\overline{x_1}(x_1 + \overline{x_3})))$$

- Ci spostiamo poi verso i termini più esterni

- $(\overline{x_1}(x_1 + \overline{x_3}))$



- $((\overline{x_2} \downarrow x_3) + (\overline{x_1}(x_1 + \overline{x_3})))$

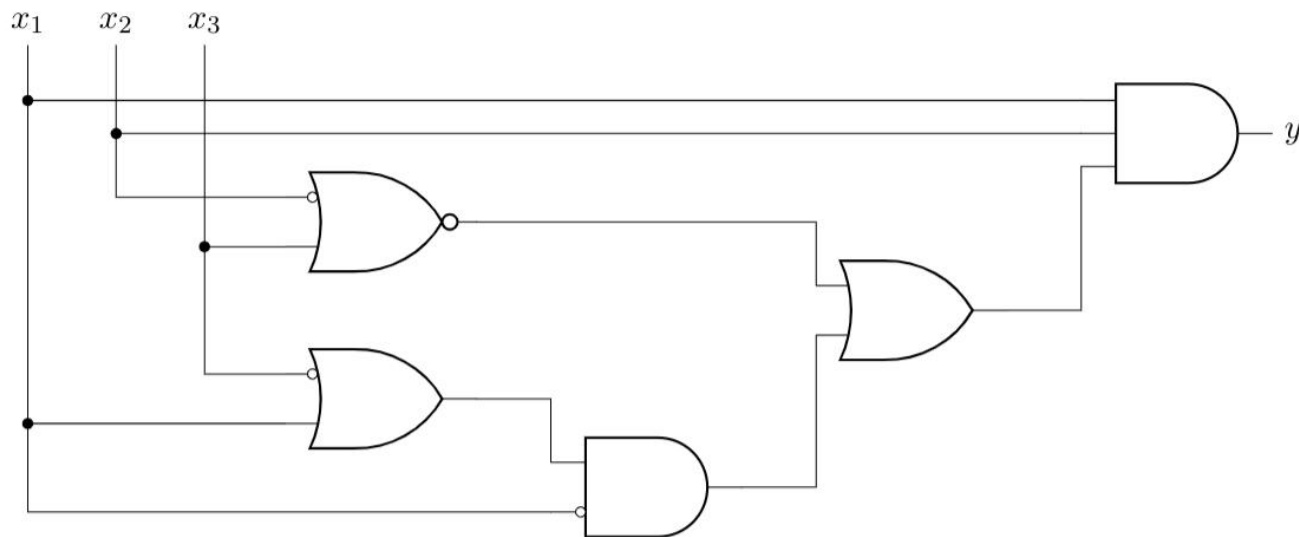


Dalla funzione al circuito

- Consideriamo la funzione:

$$y = x_1 x_2 ((\overline{x_2} \downarrow x_3) + (\overline{x_1}(x_1 + \overline{x_3})))$$

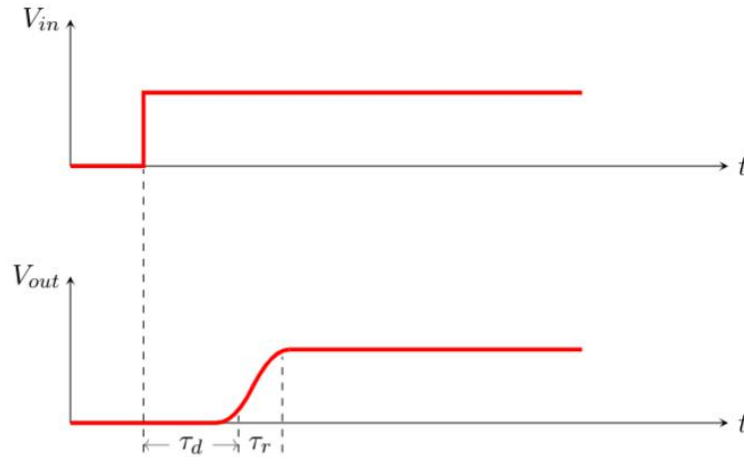
- Alla fine, otteniamo il circuito per l'intera funzione



Bontà del circuito realizzato

- È necessario chiederci quanto è “buono” il circuito che abbiamo appena realizzato
- “Buono” ha due accezioni in questo contesto:
 - Costo: quante componenti elettroniche utilizziamo per realizzare il circuito?
 - Tempo: quanto è veloce il circuito a calcolare l’uscita dati gli ingressi?
- Per quanto riguarda il tempo, fino a questo momento abbiamo barato
 - Una funzione booleana è impulsiva
 - Un circuito di commutazione *ha un ritardo di calcolo*

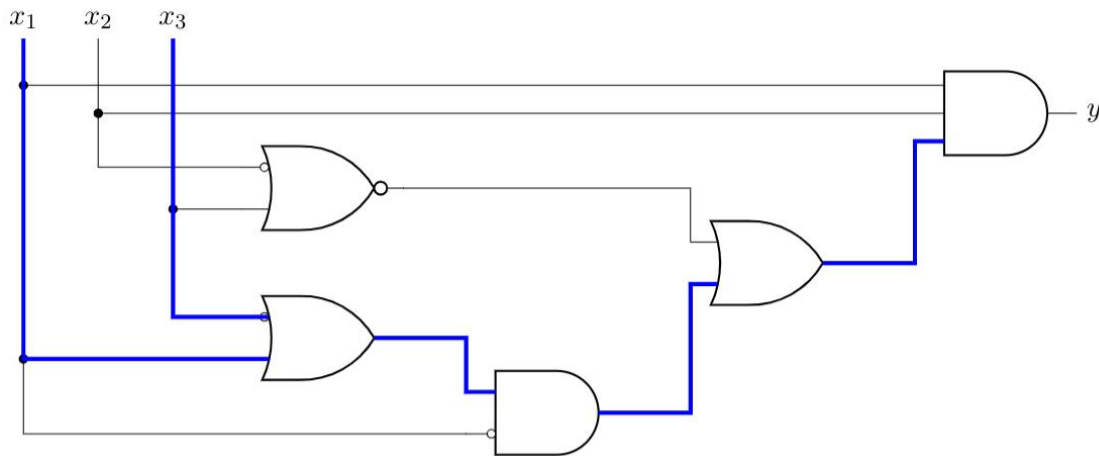
Ritardo di commutazione



- I circuiti sono componenti fisiche, pertanto se applichiamo un gradino in ingresso, l'uscita si stabilizzerà dopo un po' di tempo
 - τ_d : ritardo di commutazione (tempo per arrivare al 10% del valore finale)
 - τ_r : tempo di salita (tempo per arrivare al 90% del valore finale)
- Per semplicità, di solito si considera un unico *tempo di propagazione* τ_p

Ritardo su un circuito complesso

- Il tempo di propagazione si *accumula* quando ci sono più porte in cascata che implementano una funzione
- Per stimare le prestazioni, si può ricorrere al *critical path*:
 - il percorso più lungo che un segnale di input attraversa in un circuito



- Possiamo stimare un ritardo pari a $4\tau_p$: è un circuito a *quattro livelli*

Costo di produzione

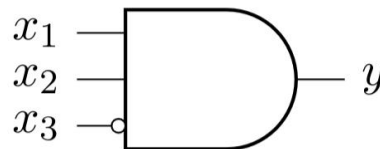
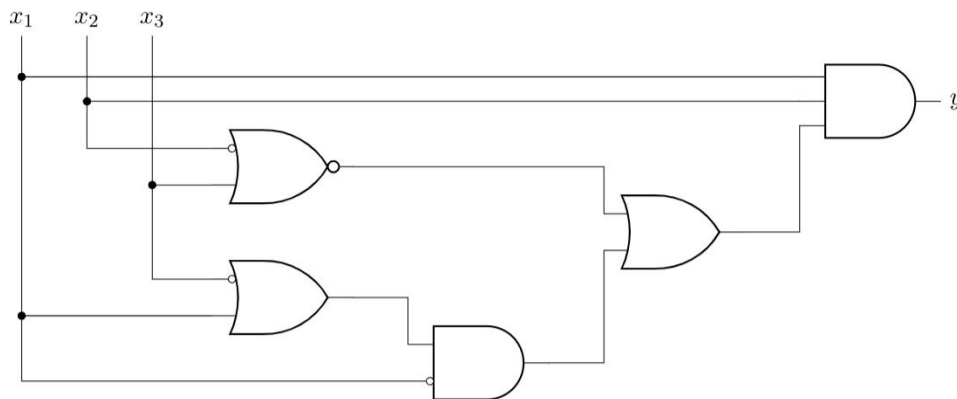
- Ogni porta logica richiede un certo numero di componenti per la sua realizzazione
- Abbiamo osservato l'implementazione di alcune porte a *due ingressi*
- Il numero di componenti necessarie è anche proporzionale al numero di ingressi di ciascuna porta
- Se riduciamo sia il numero di porte che il numero di ingressi di ciascuna porta in un circuito, “consumeremo” meno transistor nell'implementazione finale

Minimizzazione del circuito

- Possiamo sfruttare le tecniche di minimizzazione delle funzioni booleane per cercare di minimizzare il costo e massimizzare le prestazioni del nostro circuito
- $y = x_1 x_2 ((\overline{x_2} \downarrow x_3) + (\overline{x_1}(x_1 + \overline{x_3})))$
- $y = x_1 x_2 ((x_2 \cdot \overline{x_3}) + (\overline{x_1}(x_1 + \overline{x_3})))$ (definizione dell'operatore NOR)
- $y = x_1 x_2 ((x_2 \cdot \overline{x_3}) + (\overline{x_1} x_1 + \overline{x_1} \overline{x_3}))$ (proprietà distributiva)
- $y = x_1 x_2 (x_2 \cdot \overline{x_3} + \overline{x_1} \cdot \overline{x_3})$ (elemento inverso e elemento identità)
- $y = x_1 x_2 \overline{x_3} + x_1 \overline{x_1} x_2 \overline{x_3}$ (proprietà distributiva e idempotenza)
- $y = x_1 x_2 \overline{x_3}$ (elemento inverso e elemento identità)

Minimizzazione del circuito

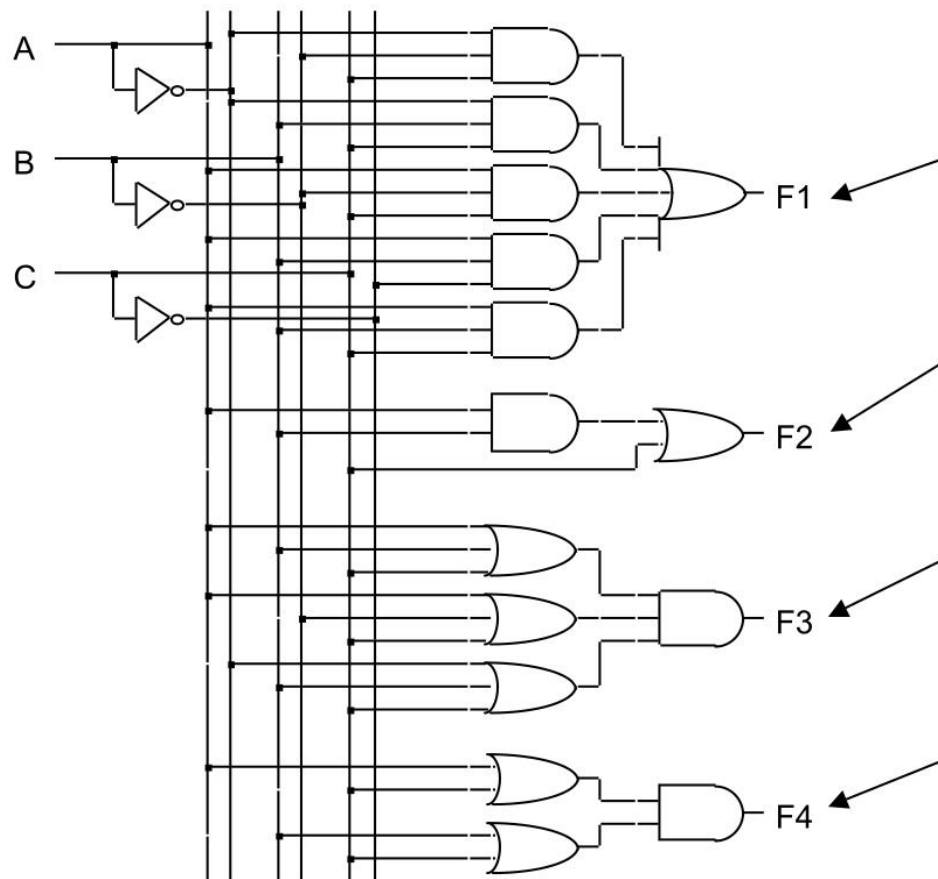
- Le tecniche di minimizzazione delle funzioni booleane ci permettono quindi identificare l'equivalenza tra i due circuiti seguenti
- È evidente la maggiore efficienza e il minor costo
- A livello elettronico, si effettuano ottimizzazioni a livello di rete di transistor che appartengono più al mondo dell'elettronica che all'informatica



Forme canoniche come reti

- Ogni funzione booleana y può essere espressa in forma canonica
- Le forme canoniche possono essere realizzate usando *2 soli livelli* di porte logiche (AND/OR)
 - velocità elevata
- Non è detto che una rappresentazione in forma canonica sia in forma minima
 - minimizzare a partire da una forma canonica può permettere di mantenere la velocità legata ai due livelli
- La scelta della forma di una funzione booleana per la sua realizzazione in hardware dipende anche da aspetti elettronici e tecnologici

Esempio di realizzazione di $f = ab + c$



somma di prodotti
canonica

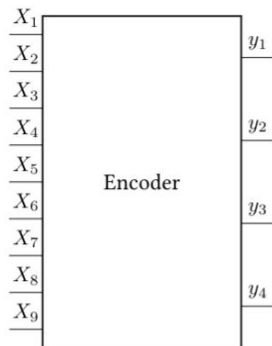
somma di prodotti
minimizzata

prodotto di somme
canonico

prodotto di somme
minimizzato

Codificatore

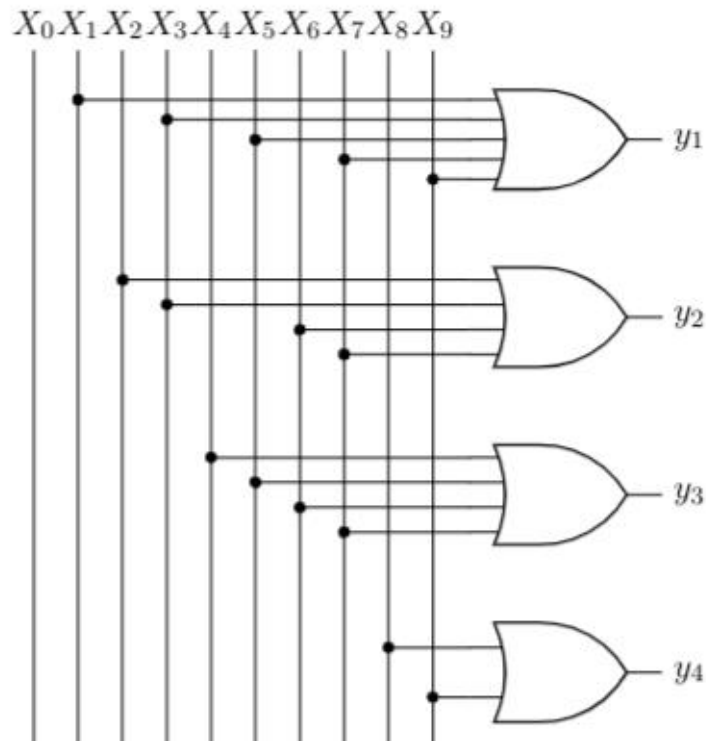
- Il codificatore (encoder) è un circuito che realizza la funzione di *codifica binaria*
 - Associa ad ogni elemento di un certo insieme di codifica composto da m simboli una sequenza distinta di n bit
- Per ogni simbolo, genera il codice corrispondente (con $2^n \geq m$)
- Il circuito ha quindi m linee di ingresso x_0, \dots, x_{m-1} ed n linee di uscita y_0, \dots, y_{n-1}



Codificatore: un esempio

- Codificatore per le cifre decimali in BCD

Base 10	BCD	Base 10	BCD
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001



Decodificatore

- Realizza la funzione inversa del codificatore
- A partire da una parola di un codice binario, genera una uscita che identifica uno dei simboli dell'insieme di interesse
- Per ciascuna configurazione di ingresso, una sola uscita vale 1. Le altre uscite valgono 0.

Decodificatore: un esempio

- Proviamo a invertire il codificatore per BCD
- Vogliamo generare uno dei segnali X_0, \dots, X_9 a partire dalla codifica $y = \langle y_1, y_2, y_3, y_4 \rangle$
- Il generico segnale X_i è generato dall'i-esimo mintermine delle variabili y_1, y_2, y_3, y_4
- Le equazioni sono quindi del tipo:

$$X_0 = \overline{y_1} \overline{y_2} \overline{y_3} \overline{y_4}$$

$$X_1 = y_1 \overline{y_2} \overline{y_3} \overline{y_4}$$

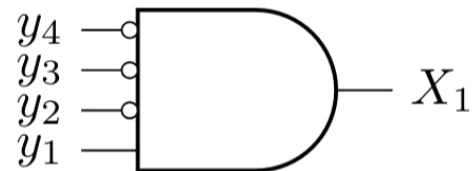
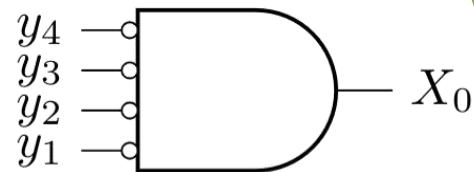
$$\vdots$$

$$X_9 = y_1 \overline{y_2} \overline{y_3} y_4$$

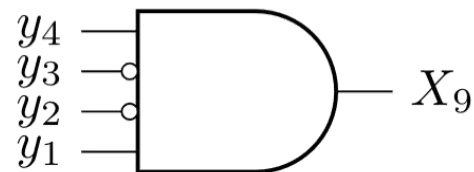
- Il circuito può quindi essere realizzato con una serie di porte AND

Decodificatore: un esempio

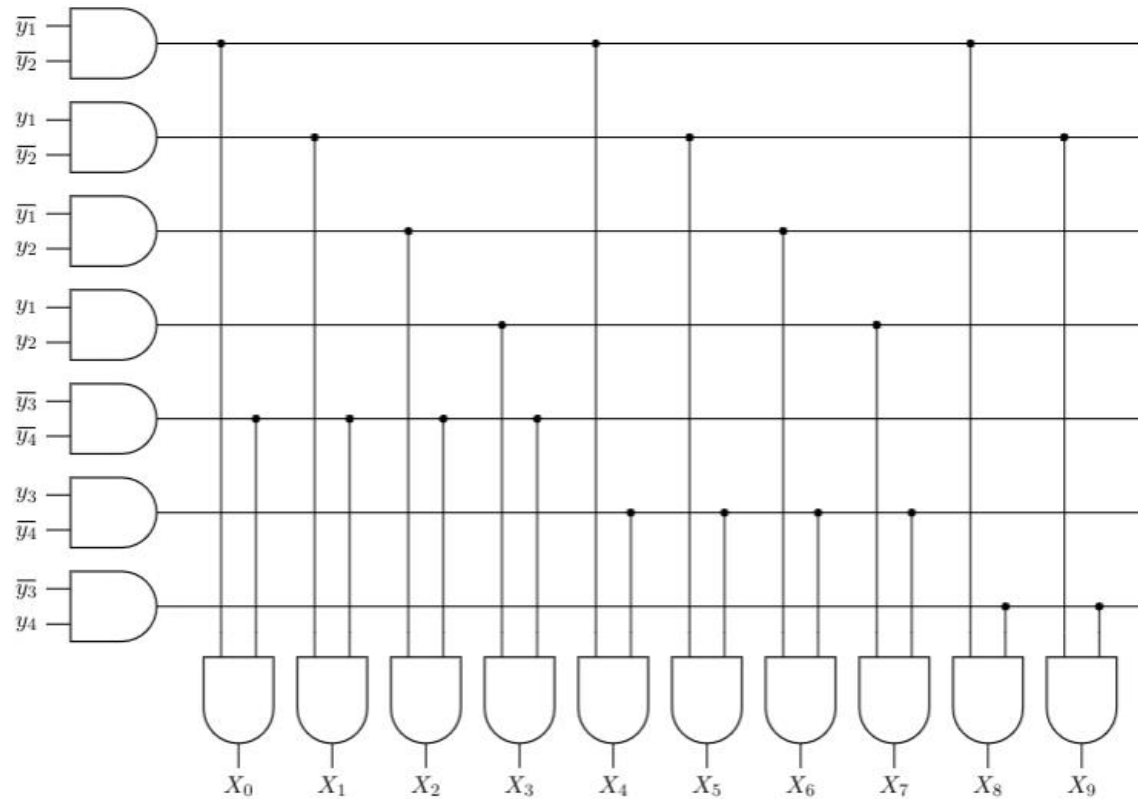
- La prima realizzazione è immediata
- $X_0 = \overline{y_1} \overline{y_2} \overline{y_3} \overline{y_4}$
- $X_1 = y_1 \overline{y_2} \overline{y_3} \overline{y_4}$
- \vdots
- $X_9 = y_1 \overline{y_2} \overline{y_3} y_4$
- Tuttavia, possiamo ridurre la complessità (aumentando il tempo di calcolo) usando un circuito a due livelli
- Decodifichiamo prima $y_1 y_2$ e poi $y_3 y_4$
- Calcoliamo poi i prodotti incrociati



\vdots

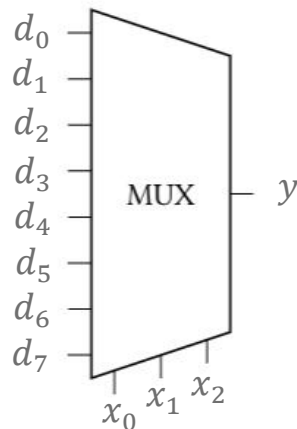


Decodificatore: un esempio



Multiplexer

- In alcuni casi è necessario *scegliere* tra più segnali in input o in output
- Questo è vero soprattutto quando si integrano circuiti più complessi
 - Immaginiamo, ad esempio, di voler fornire input diversi ad una stessa funzione booleana implementata in hardware
- Il *multiplexer* è un circuito che permette di selezionare, tra un insieme di input, un solo output
- Il multiplexer si basa su n segnali di controllo (\mathbf{x}), 2^n segnali dati (\mathbf{d}) e una sola uscita (\mathbf{y}).



Multiplexer

- L'uscita assume il valore d_i quando $x = i$
- Possiamo quindi scrivere la funzione di uscita come somma logica tra il prodotto di tutti i mintermini di x e i dati d :

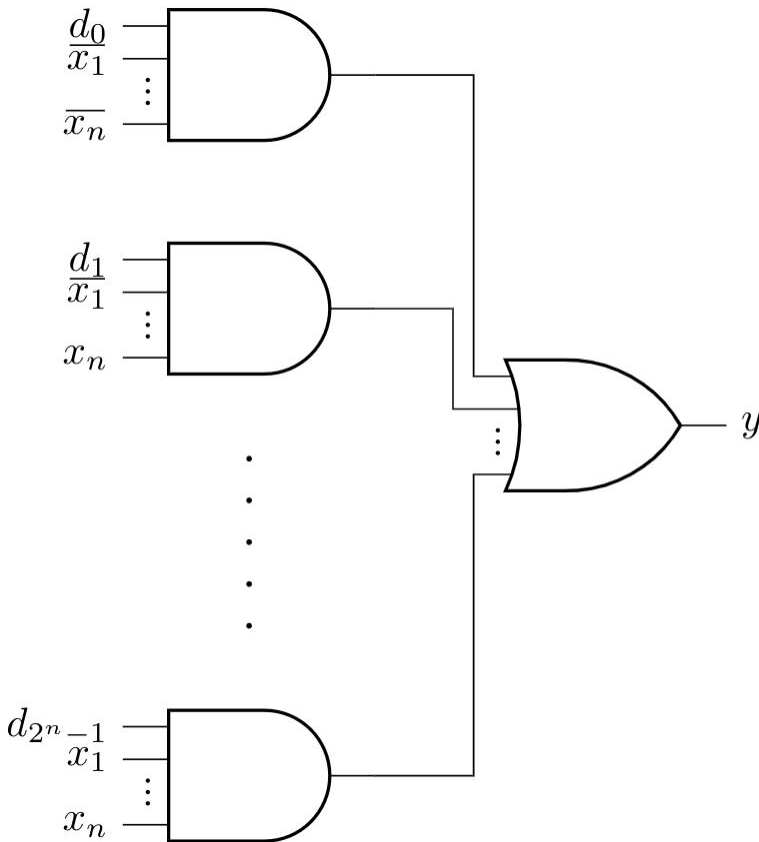
$$y = \sum_{i=0}^{2^n-1} d_i m_i$$

x	y
0	d_0
1	d_1
2	d_2
3	d_3
4	d_4
\vdots	\vdots
$2^n - 1$	d_{2^n-1}

Multiplexer

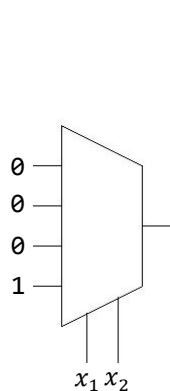
- L'uscita assume il valore d_i quando $\mathbf{x} = i$
- Possiamo quindi scrivere la funzione di uscita come somma logica tra il prodotto di tutti i mintermini di \mathbf{x} e i dati d :

$$y = \sum_{i=0}^{2^n-1} d_i m_i$$



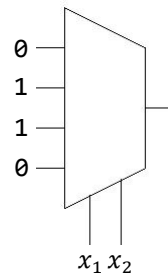
Multiplexer come generatore di funzioni

- Un multiplexer può essere utilizzato per “implementare” qualsiasi funzione booleana di n variabili
- La rappresentazione in forma tabellare di una funzione determina per ogni configurazione dell’input il valore di output
- Un multiplexer può implementare tale tabella



AND

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

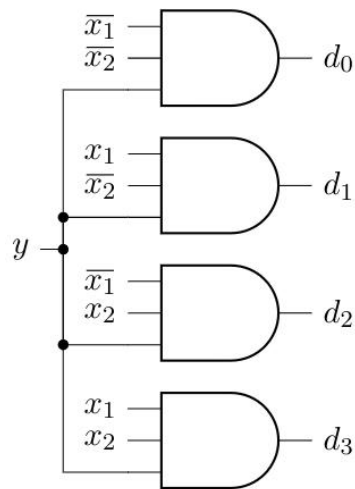
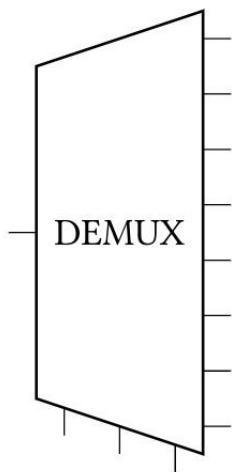


XOR

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

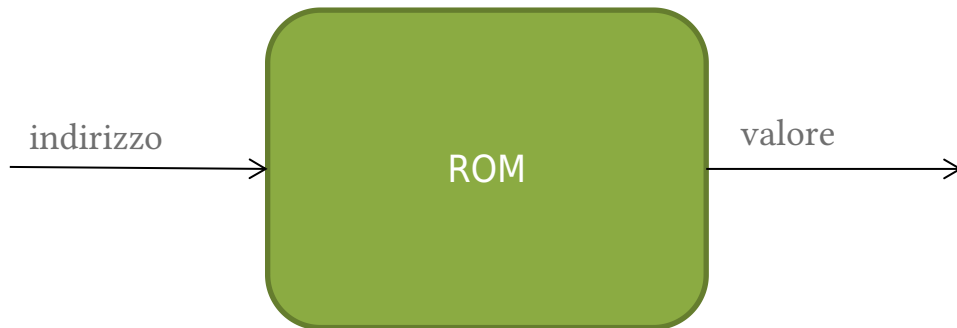
Demultiplexer

- Il circuito duale del multiplexer è il *demultiplexer*
- L'uscita i -esima assume il valore y quando la variabile di controllo $\mathbf{x} = i$
- Forte somiglianza con il decoder



Read Only Memory

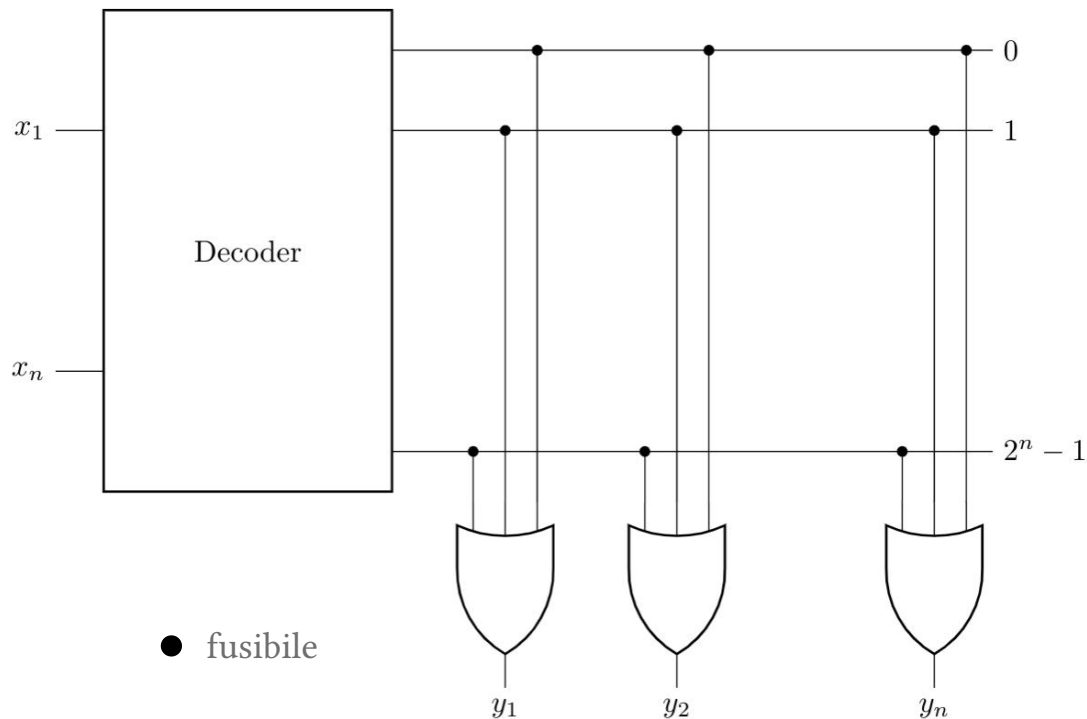
- È una memoria in sola lettura
- Le locazioni di memoria possono essere lette specificandone l'indirizzo



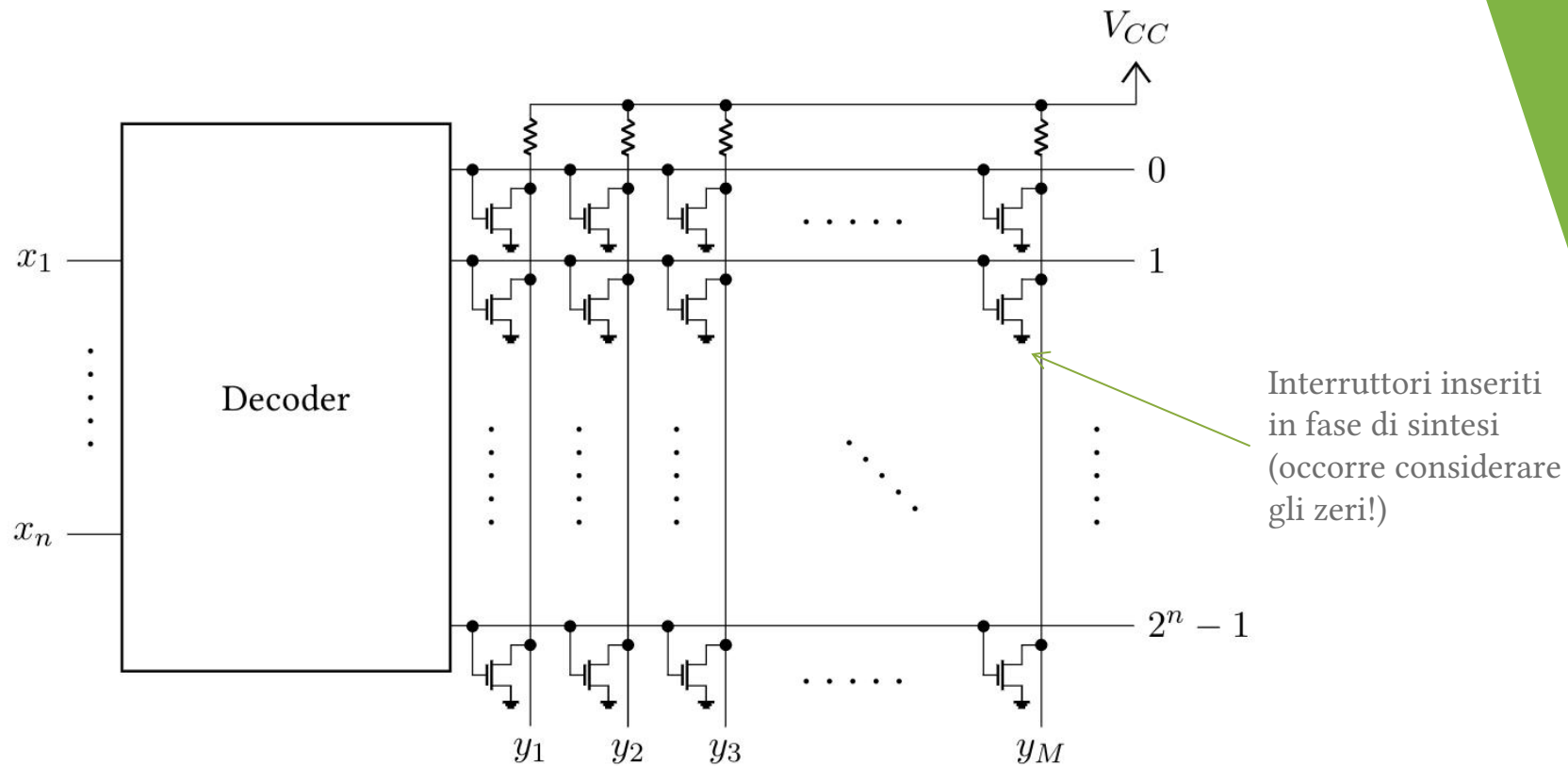
- È un'implementazione alternativa di un circuito combinatorio
 - dato un ingresso, c'è una sola uscita

Schema logico di una ROM

- Le funzioni di commutazione sono realizzate come OR di mintermini

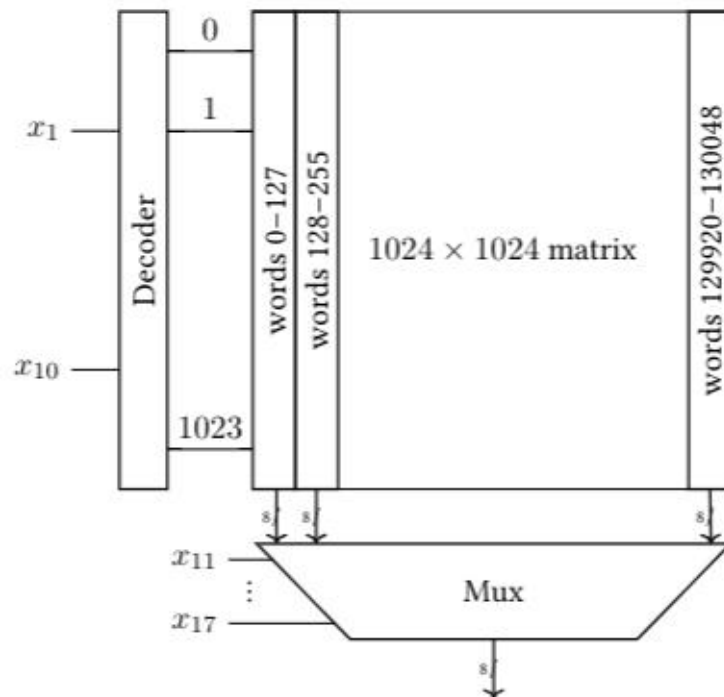


Implementazione di una ROM



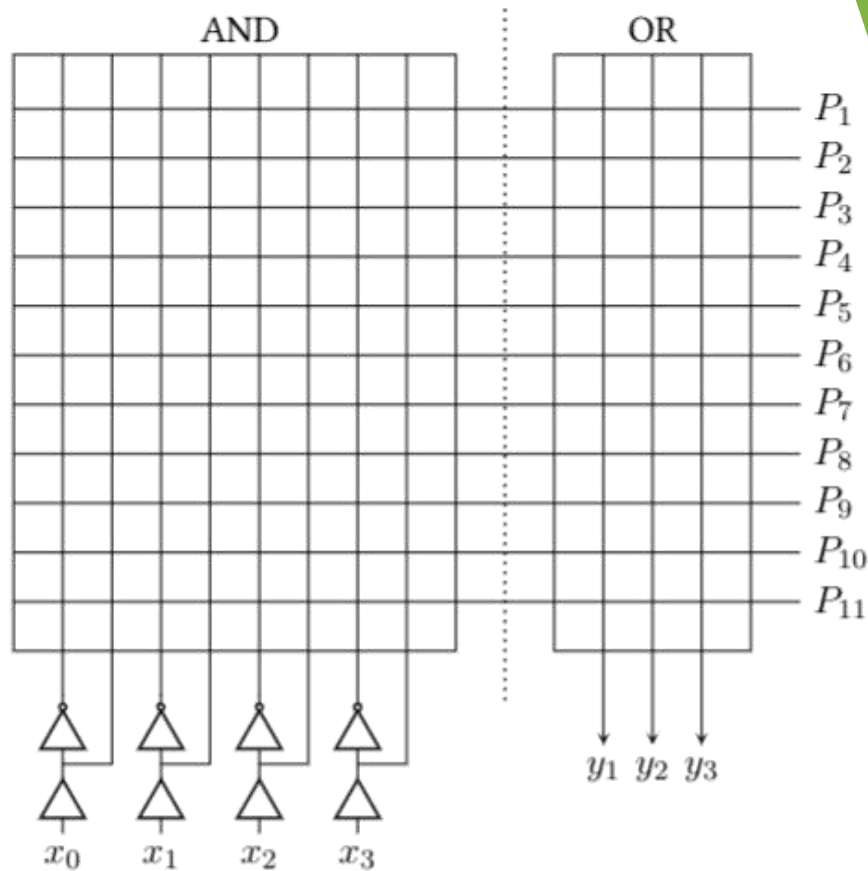
ROM Paginata

- Per motivi di ottimizzazione di superficie, si cerca di realizzare le ROM in forma quadrata
- Una ROM viene quindi organizzata in “pagine”
- Una parte dei bit dell’indirizzo viene usata per selezionare la pagina
- La restante parte seleziona la parola all’interno della pagina



Programmable Logic Array

- Logicamente una ROM è una matrice di AND (il decoder) che implementa tutti i mintermini, accoppiata ad una matrice di OR che implementa le varie funzioni
- Si potrebbe rendere programmabile sia la rete AND che la rete OR
- La Programmable Logic Array (PLA) realizza questa struttura



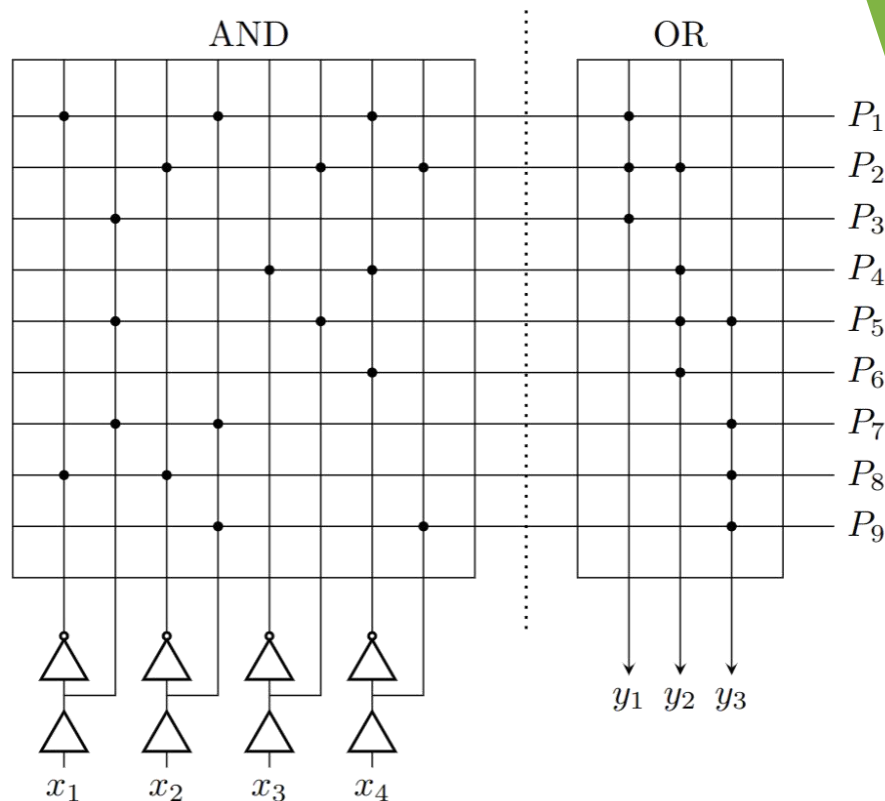
Programmable Logic Array

- Come nel caso della ROM, la programmazione avviene a tempo di sintesi
- Dei “fusibili” possono essere utilizzati per collegare i fili sia nella rete AND che nella rete OR
- Esempio:

$$y_1 = \overline{x_1}x_2\overline{x_4} + \overline{x_2}x_3x_4 + x_1$$

$$y_2 = \overline{x_2}x_3x_4 + \overline{x_3}\overline{x_4} + x_1x_3 + x_2\overline{x_4}$$

$$y_3 = x_1x_2 + \overline{x_1}\overline{x_2} + x_1x_3 + x_2x_4$$



Programmable Logic Array: implementazione

