

| 4. Moduli standard e circuiti combinatori

| Circuiti combinatori

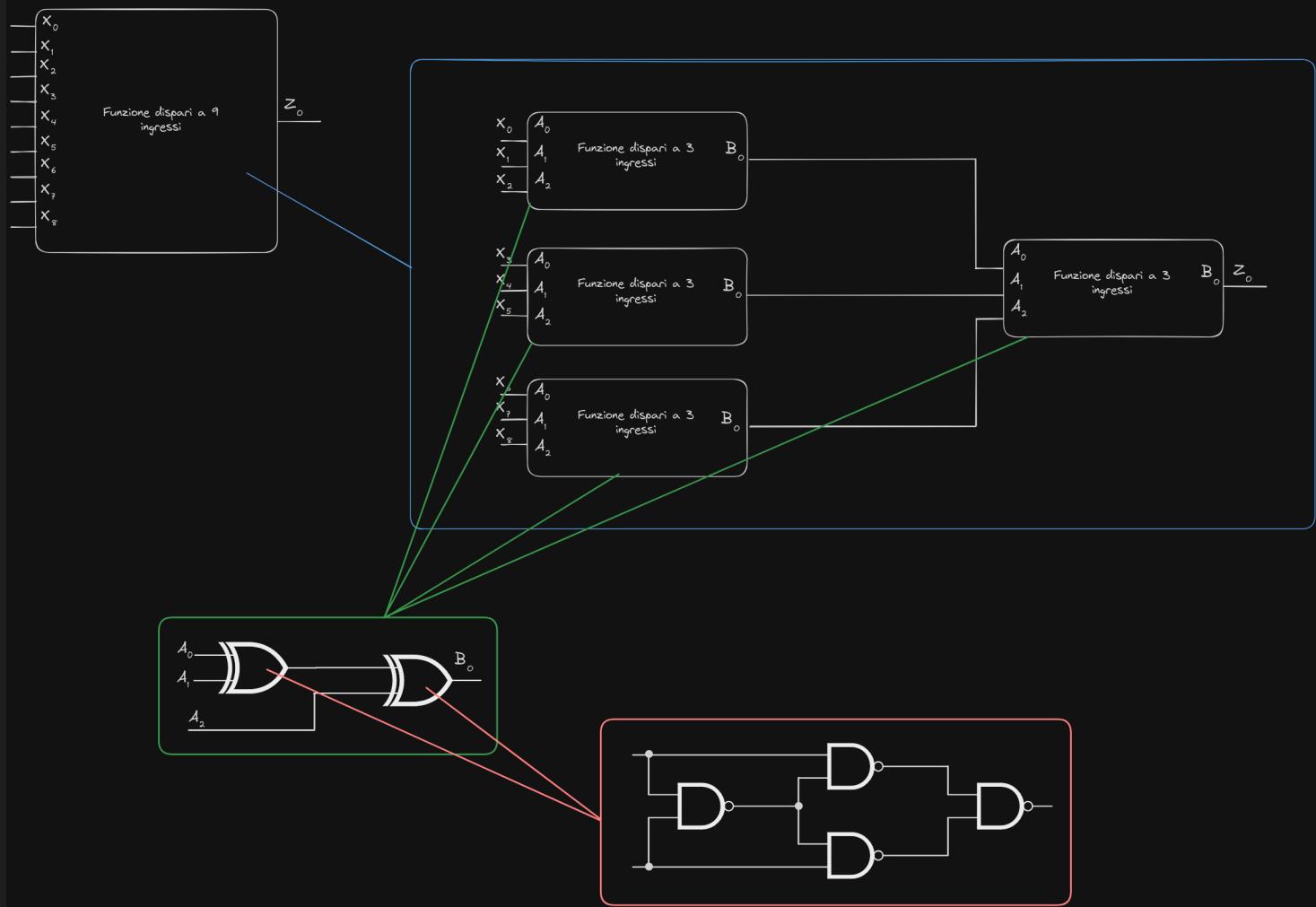
I circuiti logici dei sistemi digitali possono essere combinatori o [sequenziali](#). La differenza è che un circuito combinatorio è composto da [porte logiche](#) le cui uscite dipendono puramente dai valori in ingresso, in qualsiasi istante. I circuiti sequenziali invece utilizzano elementi di memorizzazione, e quindi le loro uscite dipendono sia dai valori di ingresso, che dai valori memorizzati in un certo istante di tempo, poiché il dato memorizzato verrà usato come input per un'altra funzione.

In questo capitolo, ci si concentrerà sui circuiti combinatori ed i loro moduli.

| Gerarchia di Progettazione

Un circuito è rappresentato da un insieme di porte logiche interconnesse tra di loro, e nei sistemi complessi possono esserci anche milioni di porte logiche, e sarebbe impossibile ed inefficiente rappresentare graficamente un sistema del genere. Per la loro progettazione allora si usa un approccio *dividi et impera*, suddividendo il circuito in tante parti, chiamate blocchi, che vengono collegate tra di loro. I blocchi vengono poi riutilizzati per evitare ridondanza.

UTILIZZO DELLA GERARCHIA DI PROGETTAZIONE
PER RAPPRESENTARE UNA FUNZIONE DISPARI A 9
INGRESSI PER IL CONTROLLO DELLA PARITÀ PARI
SU UN BYTE AL QUALE È STATO AGGIUNTO UN BIT
DI PARITÀ



La struttura della gerarchia può essere rappresentata cominciando dal blocco più ad alto livello e riportando sotto i blocchi di livello più basso che lo compongono, fino ad arrivare ad i blocchi primitivi che fanno da "foglie" dell'albero.

Nei prossimi paragrafi verranno descritti i moduli standard o blocchi funzionali, cioè blocchi che svolgono funzioni ampiamente utilizzate e che sono ben conosciuti.

Procedura di Analisi e Sintesi di un circuito combinatorio

L'analisi di un circuito combinatorio ha lo scopo di, dato il diagramma logico del circuito e come è fatto, determinare il suo comportamento, cioè quello che deve fare, e come si comporta quando gli vengono forniti valori di ingresso specifici. In questa fase:

- 1) Ci si assicura che il circuito sia effettivamente combinatorio (cioè, non devono essere presenti elementi di memorizzazione e le uscite delle porte non devono ricollegarsi al loro ingresso)
- 2) Si individuano le funzioni booleane che corrispondono al circuito e le relative tabelle

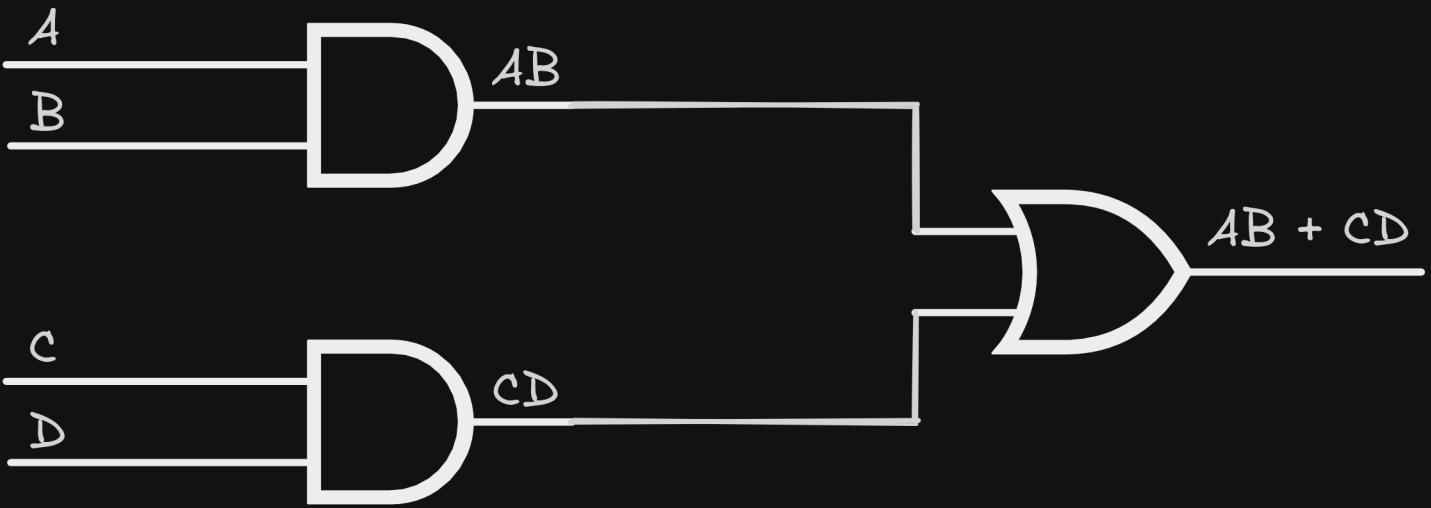
di verità

3) Si definisce una eventuale descrizione del funzionamento del circuito analizzato.

Per trovare le funzioni booleane, si procede simulando la logica delle porte, etichettando con un simbolo ogni uscita ed eseguendo solo operazioni binarie.

Per trovare le tabelle di verità, in genere si:

- 1) Determina il numero di variabili di ingresso nel circuito;
- 2) Divide il circuito in blocchi con una sola uscita;
- 3) Scrive la tavola di verità dei blocchi che dipendono solo dalle variabili in ingresso nel circuito;
- 4) Scrivono le tavole di verità dei blocchi le cui entrate dipendono dalle tavole precedentemente scritte.



La funzione booleana che corrisponde all'output del circuito è:

$$AB + CD$$

A	B	C	D	AB	CD	$AB + CD$
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	0	1	1	0	1	1
0	1	0	0	0	0	0
0	1	0	1	0	0	0
0	1	1	0	0	0	0

0	1	1	1	0	1	1
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	0	1	1	0	1	1
1	1	0	0	1	0	1
1	1	0	1	1	0	1
1	1	1	0	1	0	1
1	1	1	1	0	1	1

Conclusioni: il circuito rileva se i segnali A e B oppure C e D sono attivi contemporaneamente

La sintesi di un circuito combinatorio è il contrario dell'analisi: ha lo scopo di, dato il comportamento del circuito e la descrizione di cosa fa, determinare il suo diagramma logico, cioè quali sono i componenti e le interconnessioni necessarie per realizzarlo. In questa fase:

- 1) Si determina il numero e quali ingressi ed uscite necessita il circuito, assegnandoli specifici simboli (per esempio in un circuito che fa la somma tra due bit, posso chiamare A e B i due addendi di input ed S e C gli output che rappresentano il valore della somma e del riporto);

- 2) Si stende la tabella di verità che definisce le relazioni tra ingressi e uscite (cioè, identifico qual'è il valore delle uscite per tutte le combinazioni di input);
- 3) Si identificano per ogni uscita le funzioni booleane minimizzate in funzione delle variabili di ingresso (per esempio con le mappe di Karnaugh) ed infine si disegna e verifica la correttezza del diagramma.

Il procedimento di sintesi verrà usato in alcuni dei paragrafi seguenti per ottenere il diagramma logico dei moduli standard a partire dalla descrizione del loro funzionamento.

| Adder

| Half-Adder

Un *half adder* è un circuito aritmetico che costruisce la somma binaria tra due bit, prendendo in input due addendi ad 1 bit, e restituendo due uscite: 1 bit rappresentante la somma, ed un 1 bit rappresentante il riporto.

Si chiama così perché si usano due half adder per fare un full-adder.

| Tavola di verità

È la stessa dell'operazione di somma binaria:

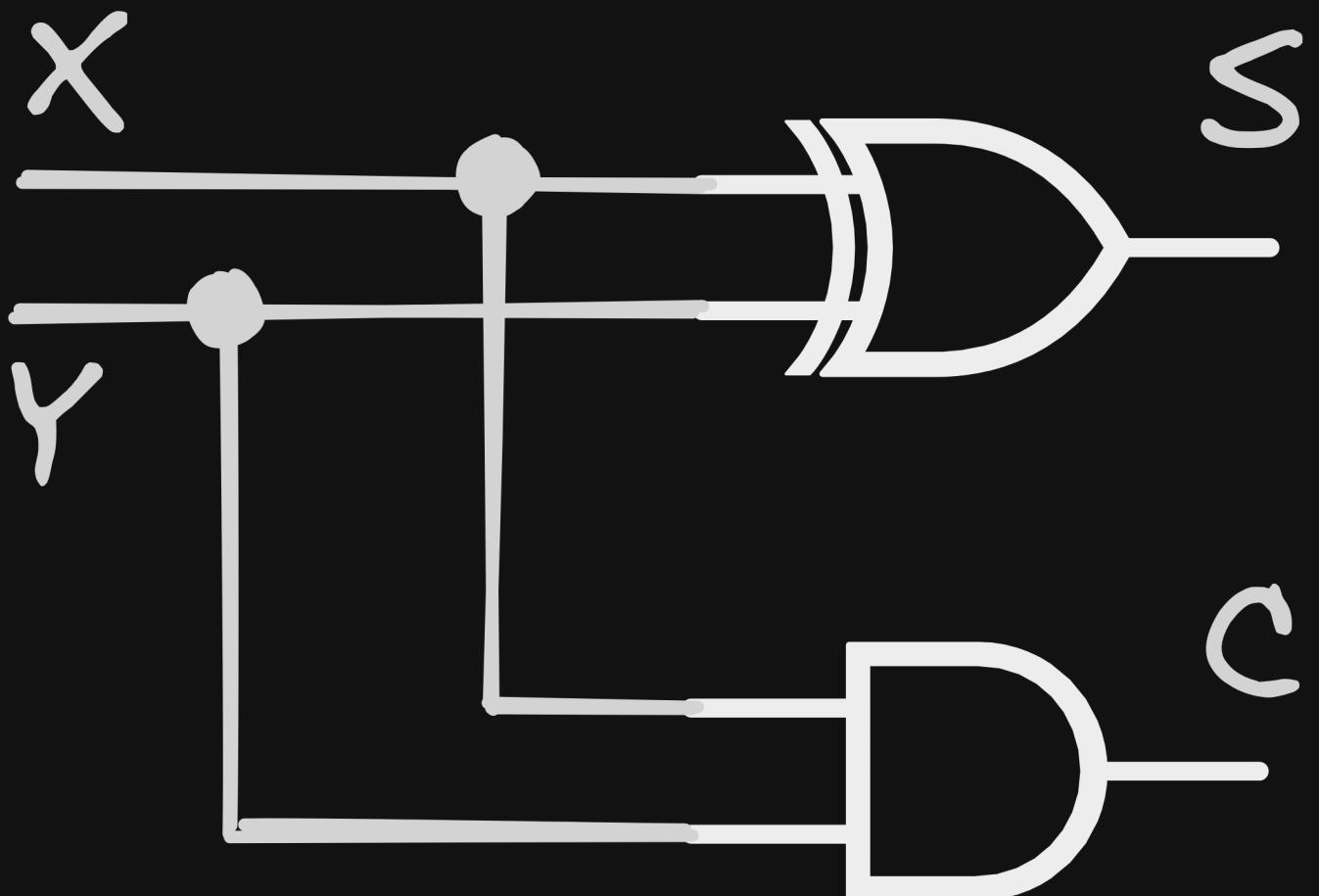
- X : primo addendo
- Y : secondo addendo
- C : riporto
- S : somma

X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Anche senza fare la mappa di Karnaugh, è facile osservare dalla tabella di verità che:

- $S = \overline{XY} + X\overline{Y} = X \oplus Y$
- $C = XY$

| Diagramma logico



| Full-Adder

Un *full adder* è un circuito aritmetico composto da due half adder che costruisce la somma binaria tra tre bit, prendendo in input tre addendi ad 1 bit (i due addendi + il terzo addendo, che volendo potrebbe essere il riporto di un somma precedente), e restituendo due uscite: 1 bit rappresentante la somma, ed un 1 bit rappresentante il riporto.

| Tavola di verità

- X : primo addendo
- Y : secondo addendo
- Z : riporto di un'operazione precedente o terzo addendo
- C : riporto
- S : somma

X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1

X	Y	Z	C	S
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

Possiamo osservare che:

- Se tutti i bit di input sono 0, allora anche le uscite sono 0
- Se solo uno oppure tutti e tre gli ingressi sono uguali ad 1, allora $S = 1$
- Se due o tutti e tre gli ingressi sono uguali ad 1, allora $C = 1$

Indipendentemente da queste osservazioni, possiamo trovare con le mappe di Karnaugh l'espressione minimale.

SOMMA

\cancel{Z}	0	1
$\cancel{X}Y$	0	1
00	0	1
01	1	0
11	0	1
10	1	0

$\bar{X}\bar{Y}Z$

$\bar{X}\bar{Y}\bar{Z}$

$X\bar{Y}Z$

$\bar{X}\bar{Y}\bar{Z}$

RIPORTO

\cancel{Z}	0	1
$\cancel{X}Y$	0	0
00	0	0
01	0	1
11	1	1
10	0	1

YZ

XY

XZ

$$SOP = \bar{X}\bar{Y}\bar{Z} + \bar{X}\bar{Y}\bar{Z} + X\bar{Y}Z + \bar{X}\bar{Y}Z$$

$$SOP = XY + XZ + YZ$$

Semplificabile
in:

$$SOP = (X \oplus Y) \oplus Z$$

Semplificabile
in:

$$SOP = X(Y \oplus Z)$$

La SOP della somma è stata semplificata attraverso [il seguente assioma dell'XOR](#):

$$X \oplus Y \oplus Z = (X\bar{Y} + \bar{X}Y)\bar{Z} + (XY + \bar{X}\bar{Y})Z = X\bar{Y}\bar{Z} + \bar{X}Y\bar{Z} + XYZ + \bar{X}\bar{Y}Z$$

mentre la SOP del riporto richiede un po' più di intuizione: applicando [la proprietà distributiva](#), possiamo trasformare $XY + XZ + YZ$ in

$$XY + Z(X + Y)$$

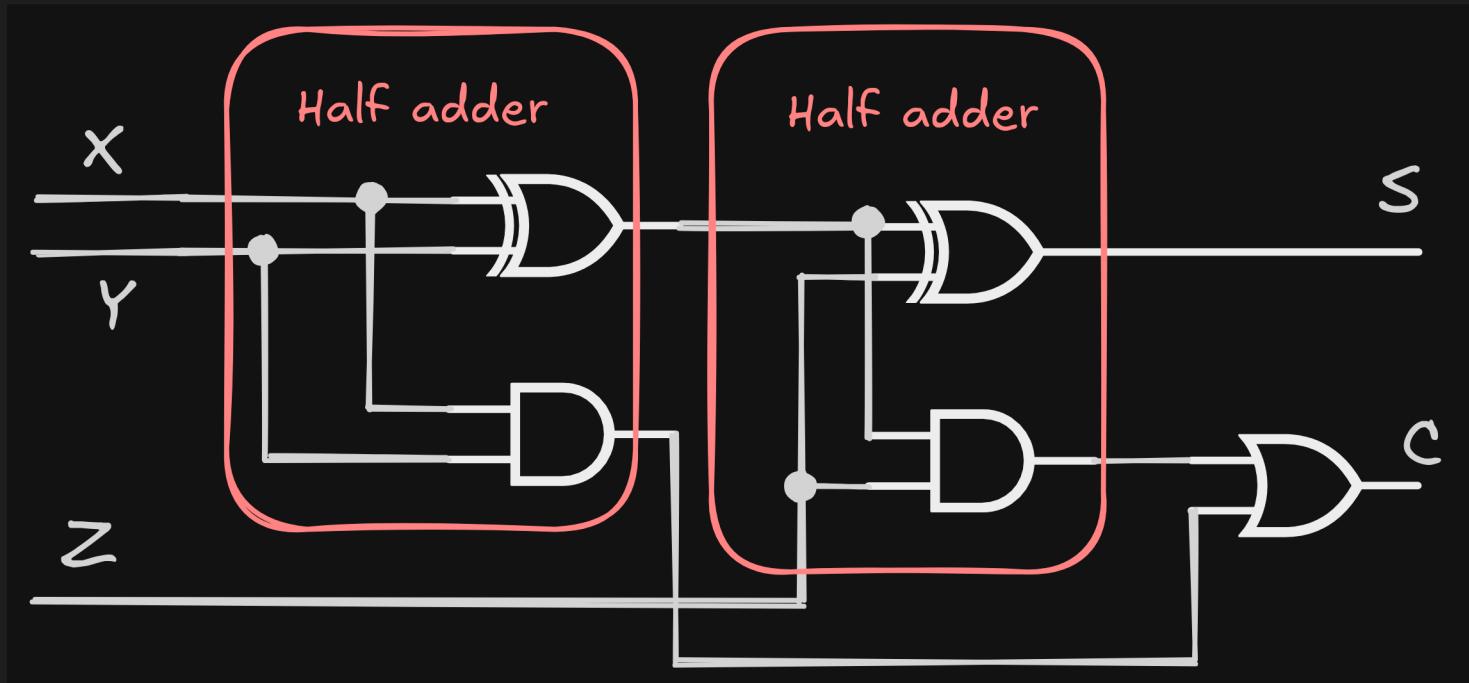
Tuttavia, guardando la mappa di Karnaugh, possiamo notare che i casi in cui si hanno X ed Y uguali sono già coperti dalla prima parte dell'espressione XY poiché il valore di Z non influenza il risultato dell'espressione^[1]. Dunque, devo preoccuparmi solo dei casi in cui X ed Y hanno valori diversi, dove il risultato dell'espressione è dettato dal valore di Z ^[2]. Dunque, posso riscrivere l'espressione come

$$XY + Z(X \oplus Y)$$

Il vantaggio di avere l'espressione in questa forma è che:

- Posso riutilizzare nel calcolo del riporto il valore $X \oplus Y$ calcolato per ottenere la somma
- Posso scrivere il full adder come due half adder dove i riporti confluiscano in una porta OR

| Diagramma logico



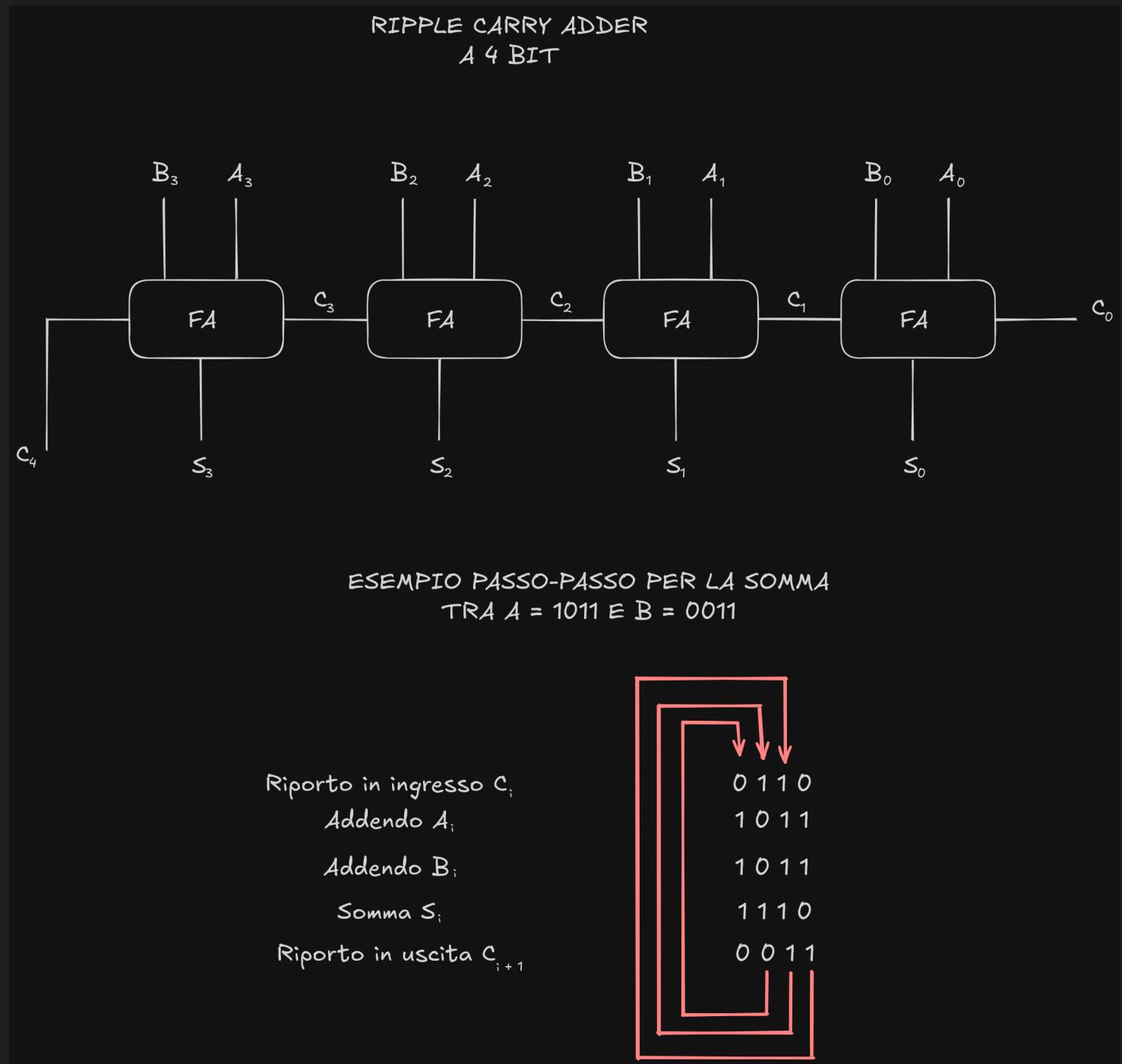
| Ripple-Carry Adder

Il *Ripple Carry Adder* è un circuito che restituisce la somma aritmetica di due numeri binari di n bit. Vengono usati n full adder ai cui ingressi sono applicati gli n bit di ciascun ingresso,

ma sono connessi in cascata, con l'uscita del riporto di ognuno connessa all'ingresso del riporto del successivo, relativo ad una cifra in posizione più significativa.

Si chiama così perché i full adder hanno i riporti (*carry*) disposti in cascata (l'uscita di uno è l'ingresso di un altro).

| Diagramma logico



| Decodificatore

Molte informazioni contenute nei calcolatori digitali sono rappresentate tramite codici binari. Un codice ad n bit è grado di rappresentare fino a 2^n informazioni distinte.

Un decodificatore è un circuito combinatorio che converte una certa combinazione di n bit in ingresso nella relativa informazione in uscita tra i 2^n mintermini rappresentabili.

Se alcune combinazioni di bit in entrata non rappresentano nessuna informazione (cioè, con n bit di ingresso, non mi servono tutte le 2^n combinazioni possibili), allora si può usare un numero minore di uscite (in questo caso, il decodificatore è detto non standard).

In questo paragrafo vediamo i decodificatori $n-m$ dove $m \leq 2^n$ con lo scopo di creare 2^n (o meno) mintermini di n variabili.

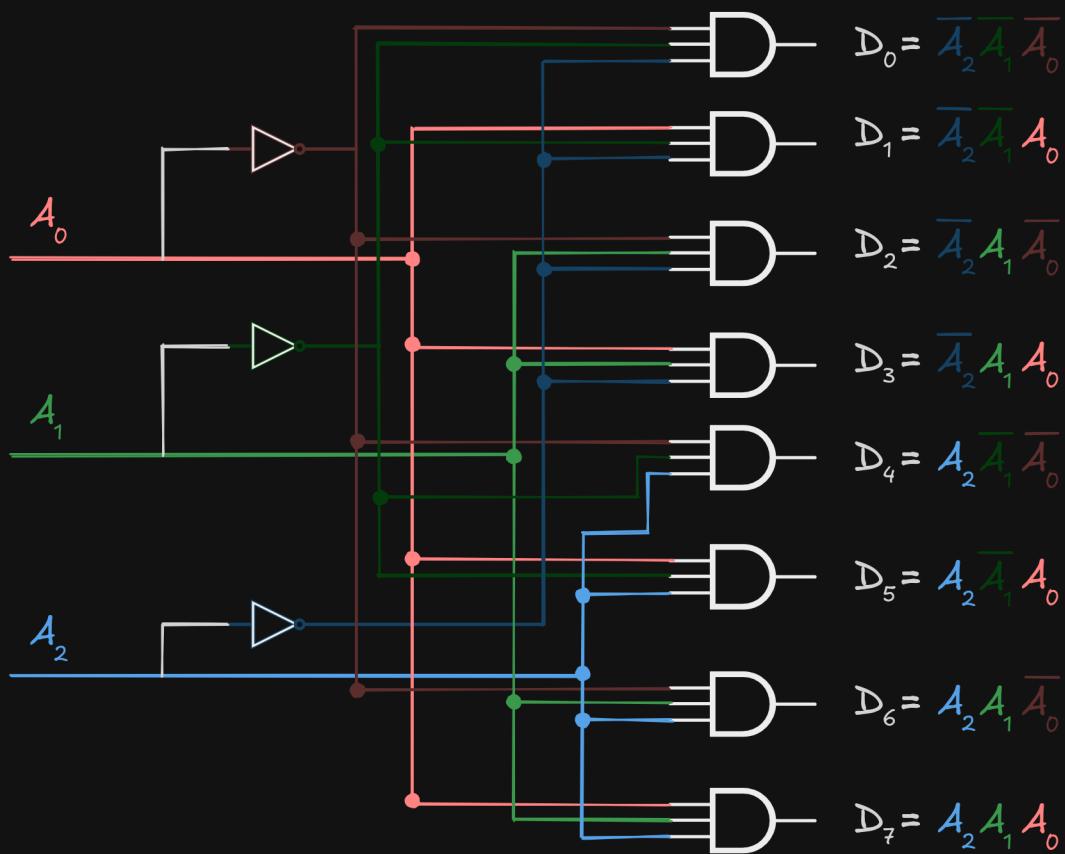
Per generare tutti i mintermini per un decodificatore 3-8, per esempio, posso usare le porte NOT per ottenere la complementazione degli ingressi, e genero poi ogni specifico mintermine mettendo in AND le varie combinazioni di input.

Osservando il diagramma logico e la tavola di verità, notiamo che per ciascuna delle possibili combinazioni, solo una delle 8 uscite D_i sarà uguale ad 1, mentre le altre saranno 0, permettendo di riconoscere quale combinazione di bit è stata decodificata.

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

DECODIFICATORE

3-8



Alcuni decodificatori sono costruiti usando direttamente porte NAND per efficienza di costo e di ritardo. I prodotti che si ottengono in uscita sono però i mintermini complementati che si otterrebbero utilizzando le porte AND (quindi nel caso della tavola di verità sopra, gli 0 e gli 1 delle uscite D_i sarebbero invertiti).

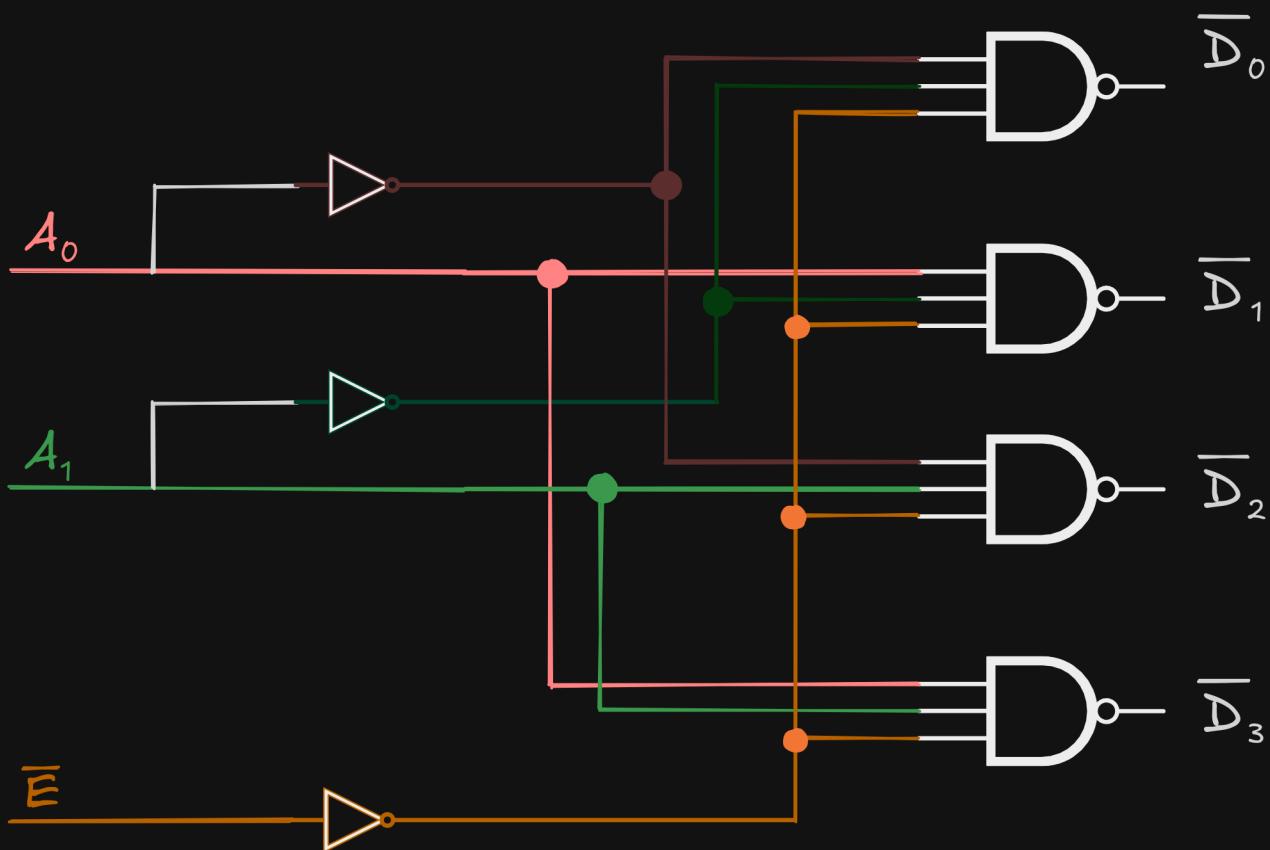
Inoltre, alcuni decodificatori usano in ingresso una o più linee di abilitazione (*enable*) per abilitare o disabilitare un'uscita.

DECODIFICATORE

2-4

PORTE NAND

LINEA ENABLE



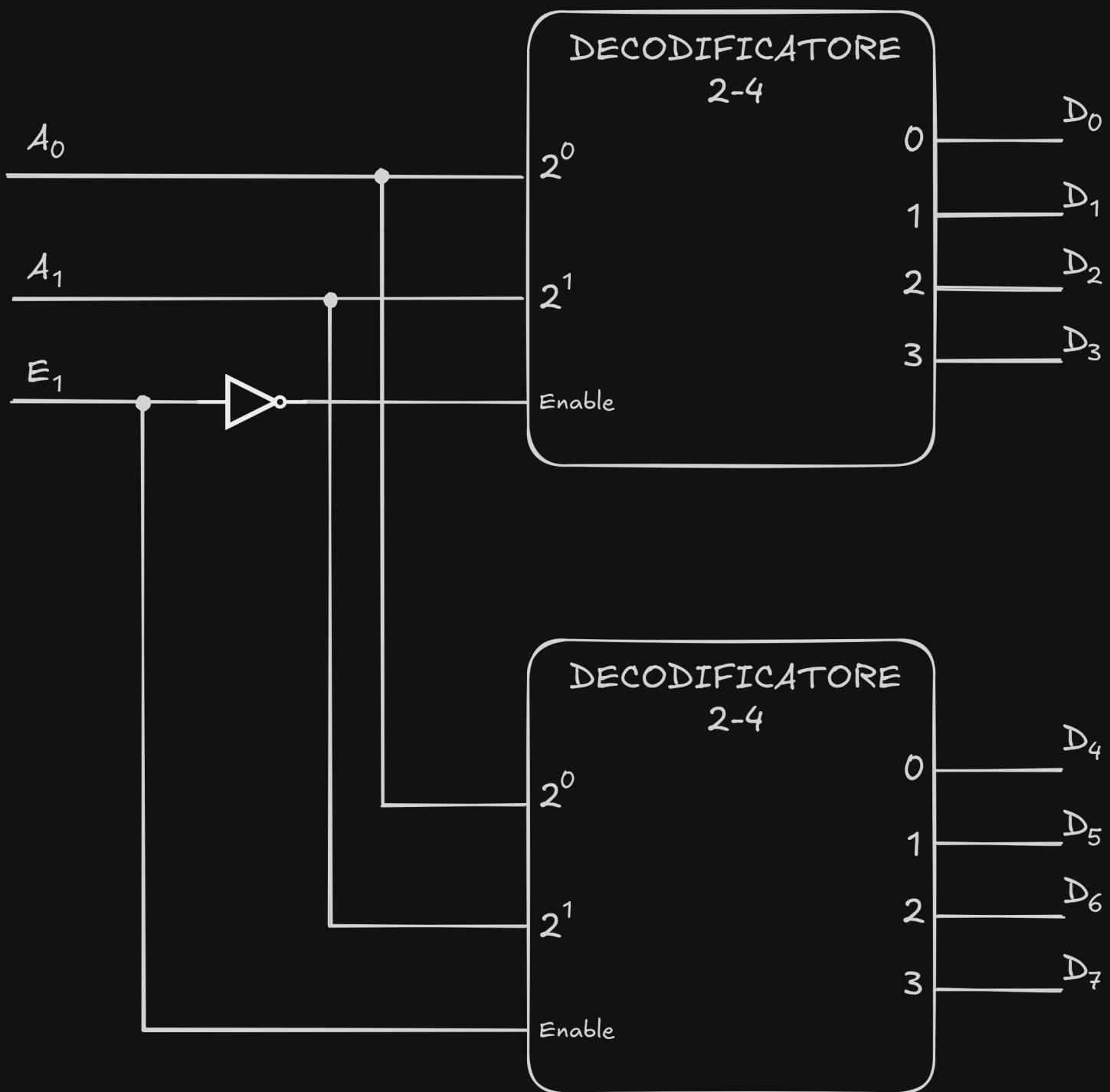
Avendo usato porte NAND, soltanto una uscita, in un dato momento, assume il valore 0, mentre le altre assumono il valore 1. L'uscita con valore 0 rappresenta il mintermine selezionato dagli ingressi A_1 ed A_0 . In questo caso quando $\bar{E} = 1$, l'uscita è disabilitata e restituirà 1 indipendentemente dagli ingressi. Quando nessuna uscita è abilitata, allora nessun termine verrà mai selezionato.

Di seguito, la tavola di verità.

\bar{E}	A_1	A_0	\bar{D}_0	\bar{D}_1	\bar{D}_2	\bar{D}_3
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0
1	X	X	1	1	1	1

Come abbiamo detto, in alcuni decodificatori sono presenti più linee di attivazione: è normale, e servono ad implementare criteri di attivazione più complessi. Per esempio, per motivi di costi si potrebbe voler implementare un decodificatore 3-8 tramite due decodificatori 2-4:

DECODIFICATORE 3-8 DA 2 DECODIFICATORI 2-4



Quando $E_1 = 0$, il decodificatore in alto è abilitato e genera i mintermini da D_0 a D_3 mentre il decodificatore in basso è disabilitato.

Al contrario, quando $E_1 = 1$, il decodificatore in alto è disabilitato, mentre quello in basso è abilitato e genera i mintermini da D_4 a D_7 .

Parlando di decodificatori, si può incappare nella rappresentazione in matrice di AND: è praticamente un altro modo di scrive la tavola di verità, sotto forma di diagramma logico che mostra tutte le combinazioni di ingressi, e le relative uscite prodotte. Viene usata la matrice di AND perché tipicamente nel decodificatore per ottenere un'uscita è richiesta una combinazione specifica di input.

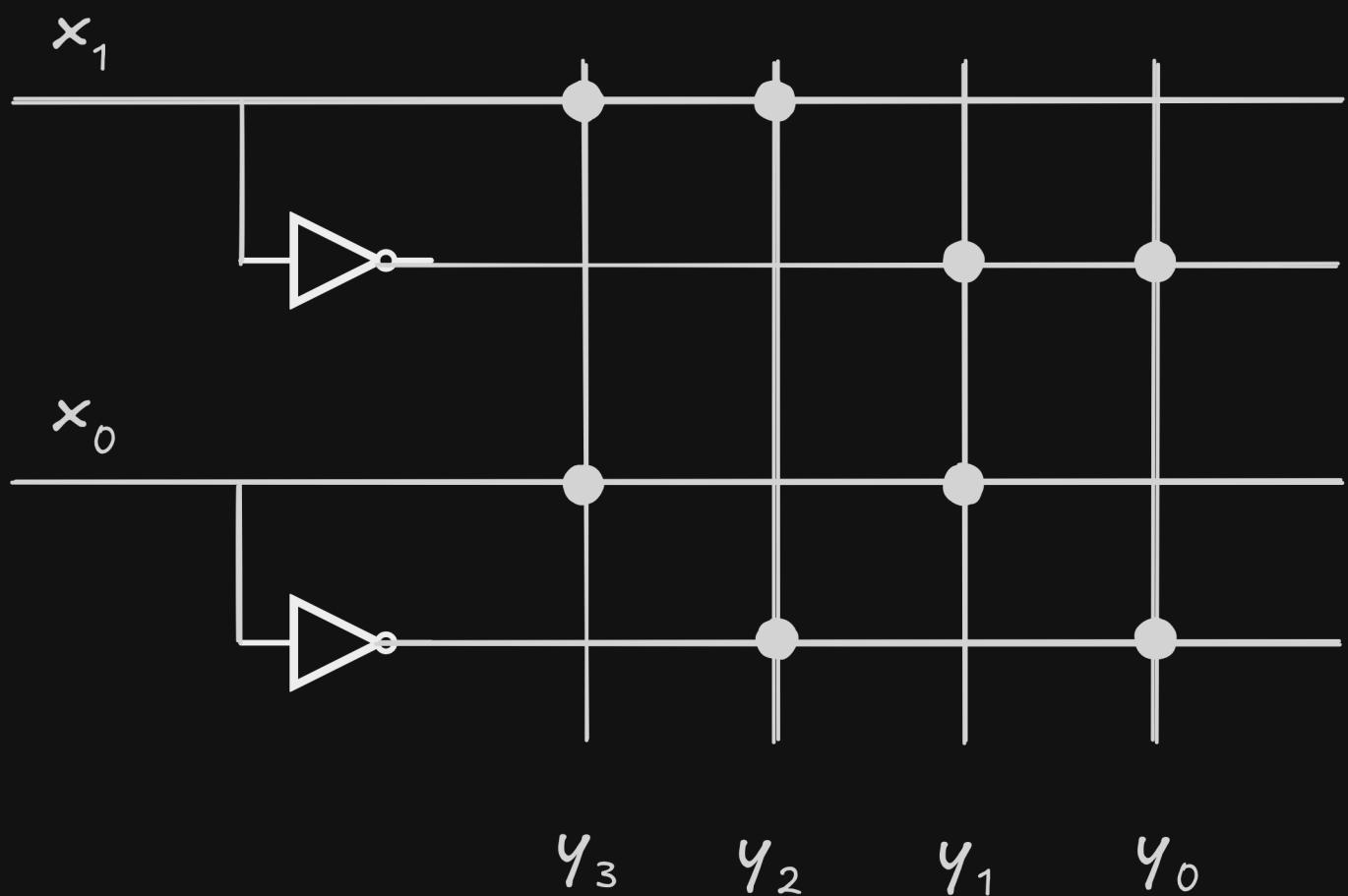
Utilizza dei puntini pieni per rappresentare i valori di input da mettere in AND tra di loro per decodificare una specifica uscita. Dove non è presente il puntino, vuol dire che quella variabile non è usata nella decodifica del valore.

Nel seguente esempio, possiamo notare che l'uscita y_0 è decodificata quando si ha come input $\overline{x_1} \overline{x_0}$, cioè entrambi gli ingressi impostati a 0 .

TAVOLA DI VERITÀ DECODIFICATORE 2-4

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

MATRICE DI
AND



Implementare circuiti combinatori con decodificatori

È bene sapere che poiché i decodificatori producono tutti i 2^n mintermini corrispondenti alle n variabili di ingresso, e qualunque funzione booleana è esprimibile come somma di mintermini, allora è possibile realizzare qualsiasi circuito combinatorio con un decodificatore e delle porte OR con n ingressi ed m uscite: il decodificatore $n-m$ genera i mintermini, e le m porte OR li sommano.

Per maggiori approfondimenti, si riporta a pagina 125 del libro "*Reti logiche - M. Morris Mano, Charles R.Kime*".

| Codificatore

Il codificatore è il circuito combinatorio inverso al decodificatore: dati 2^n o meno ingressi, vengono restituite in output n linee i cui valori formano il codice binario corrispondente al valore in ingresso. Si assume che solamente uno degli ingressi in qualunque condizione valga [1](#).

Per esempio, se l'unico ingresso con valore [1](#) è D_7 , si avrà in uscita $A_2 = 1$, $A_1 = 1$ ed $A_0 = 1$ perché $7 = 111$.

Di seguito, la tavola di verità di un codificatore ottale (8-3):

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 codificatore si può implementare con OR utilizzando gli ingressi individuati dalla tavola di verità:

- L'uscita A_0 assume il valore 1 se la cifra ottale in ingresso è dispari

- L'uscita A_1 assume il valore 1 se la cifra ottale è D_2, D_3, D_6 o D_7
- L'uscita A_2 assume il valore 1 se la cifra ottale è D_4, D_5, D_6 o D_7

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

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

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

Quindi possiamo usare tre porte OR a 4 ingressi. Prima di proseguire con l'implementazione però, dobbiamo risolvere un paio di ambiguità:

- 1) Se due ingressi sono uguali ad 1, il risultato è errato. Per esempio, con $D_3 = 1$ e $D_6 = 1$ contemporaneamente, si avrebbe come risultato **111**, che non rappresenta nessuno dei due ingressi.
- 2) Si risolve rendendo il codificatore con priorità, cioè si fa passare l'ingresso con pedice più alto (quindi nell'esempio precedente, verrebbe considerato solamente D_6)
- 3) Se tutti gli ingressi hanno valore **0**, allora in uscita si riceve il valore **0**, che corrisponde a D_0 e diventa quindi impossibile distinguere quando tutti gli ingressi sono **0** o se solamente $D_0 = 1$.
- 4) Si risolve aggiungendo una linea di uscita V la quale verifichi che almeno uno degli ingressi sia uguale ad 1.

Dunque, analizziamo la tavola di verità *densa* (cioè completa di ogni singola combinazione) di un codificatore 4-2 con priorità:

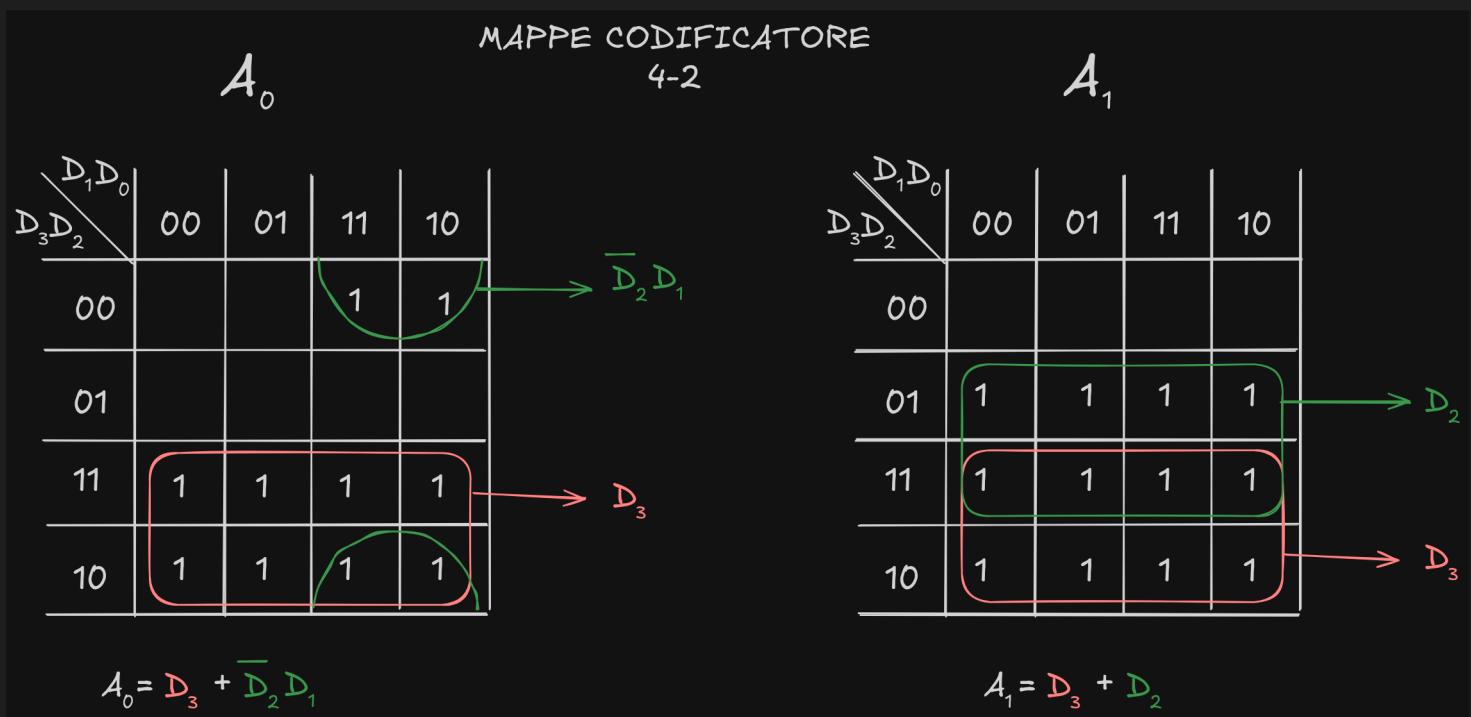
💡 L'uso delle X nella seguente tavola

- Le X nelle uscite rappresentano condizioni di non specificazione (Viene effettivamente generato un valore, ma non ci interessa quale sia e quindi mettiamo X)
- Le X negli ingressi rappresentano l'assenza della variabile nel mintermine corrispondente (per esempio, **001X** rappresenta il mintermine $\overline{D}_3 \overline{D}_2 D_1$ e quindi è assente D_0)

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

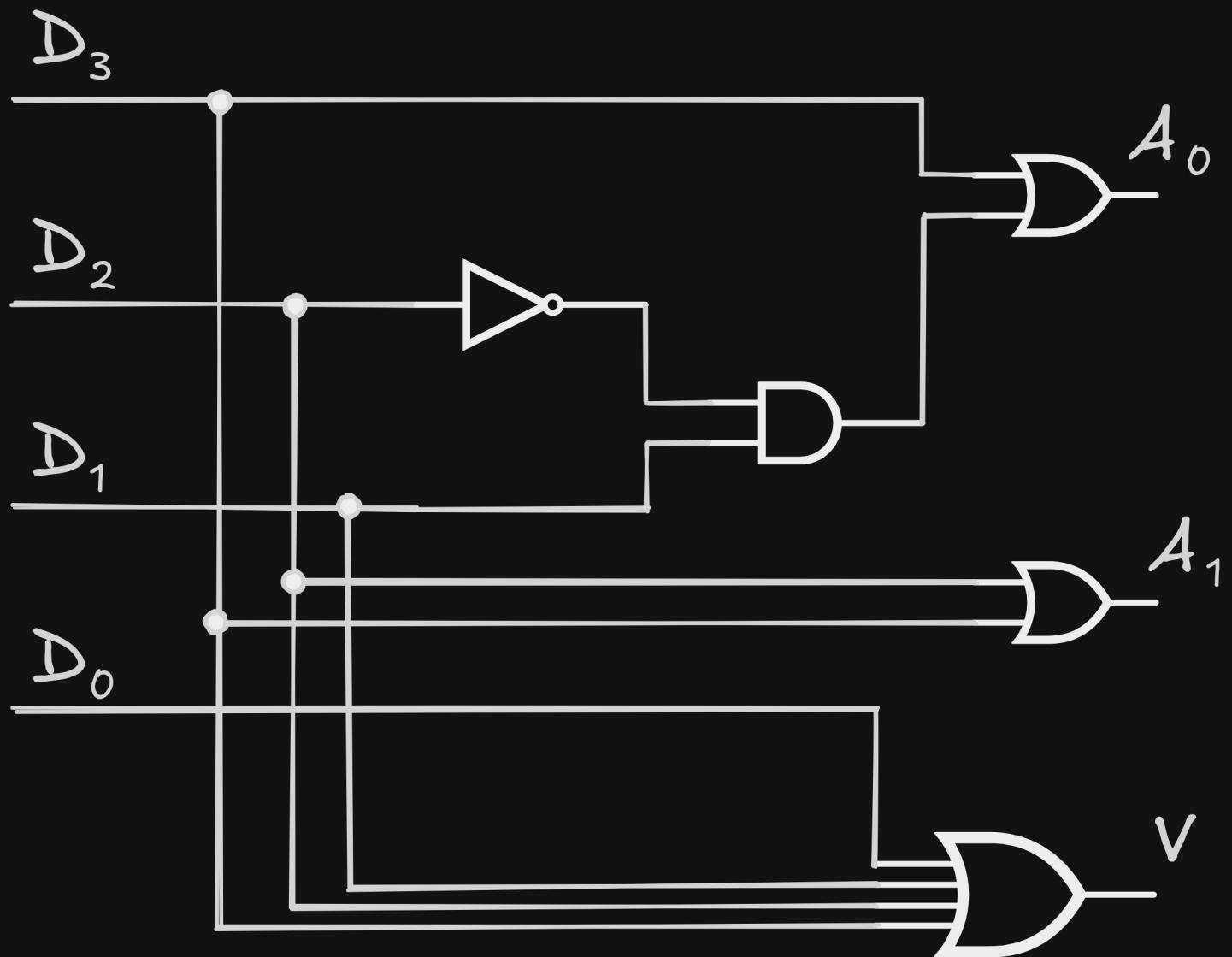
L'uscita V indica se almeno uno dei valori in ingresso era uguale ad 1.

Produciamo le mappe di Karnaugh per le uscite in modo da poter implementare il circuito in formato minimale. Per V non serve, perché possiamo notare che corrisponde all'OR di tutte le variabili in ingresso (perché ha valore 1 se almeno uno degli ingressi è 1).



Infine, posso implementare il circuito:

CODIFICATORE CON PRIORITÀ 4-2



Matrici di OR

Parlando di codificatori, si può incappare nella rappresentazione in matrice di OR: è praticamente un altro modo di scrive la tavola di verità, sotto forma di diagramma logico che mostra tutte le combinazioni di ingressi, e le relative uscite prodotte. Viene usata la matrice di OR perché tipicamente nel codificatore per ottenere un'uscita è richiesto che almeno un bit sia uguale ad 1.

Utilizza dei puntini vuoti per rappresentare i valori che vanno messi in OR tra di loro per ottenere una specifica uscita. Dove non è presente il puntino, vuol dire che quella variabile non è usata nella codifica del valore.

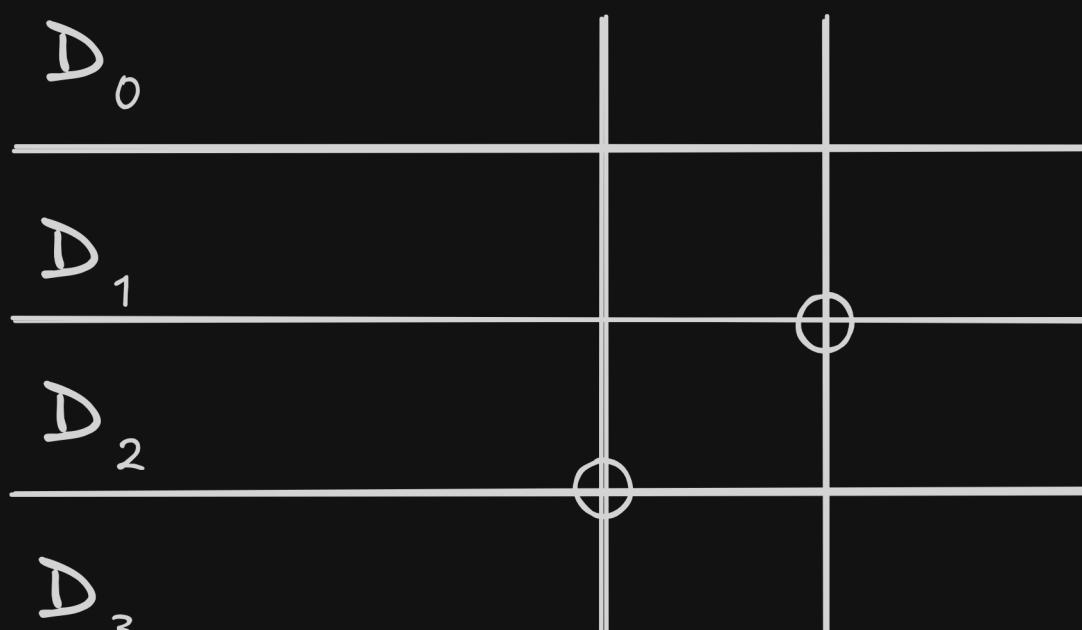
Nel seguente esempio, possiamo notare che la codifica dell'uscita A_0 è data dal valore di $D_1 + D_3$, e la codifica dell'uscita A_1 è data dal valore di $D_2 + D_3$. Infine, D_0 non è usato nella codifica di alcun valore.

TAVOLA DI VERITÀ CODIFICATORE 4-2

VISTO PRECEDENTEMENTE

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

MATRICE DI
OR



Rivedremo questa rappresentazione quando parleremo dei [PLA](#).

| ROM

Una *Read Only Memory (ROM)* è un dispositivo in cui sono immagazzinate informazioni in modo permanente e ne fanno parte integrante, e dunque il contenuto non può essere modificato elettronicamente.

Una ROM riceve k ingressi rappresentanti l'indirizzo della word^[3] da recuperare e restituisce in uscita gli n bit della word immagazzinata in quell'indirizzo. La ROM è implementata con un decodificatore che ricava l'indirizzo dai k bit di ingresso, seguito da un codificatore generalizzato, che restituisce gli n bit della word.

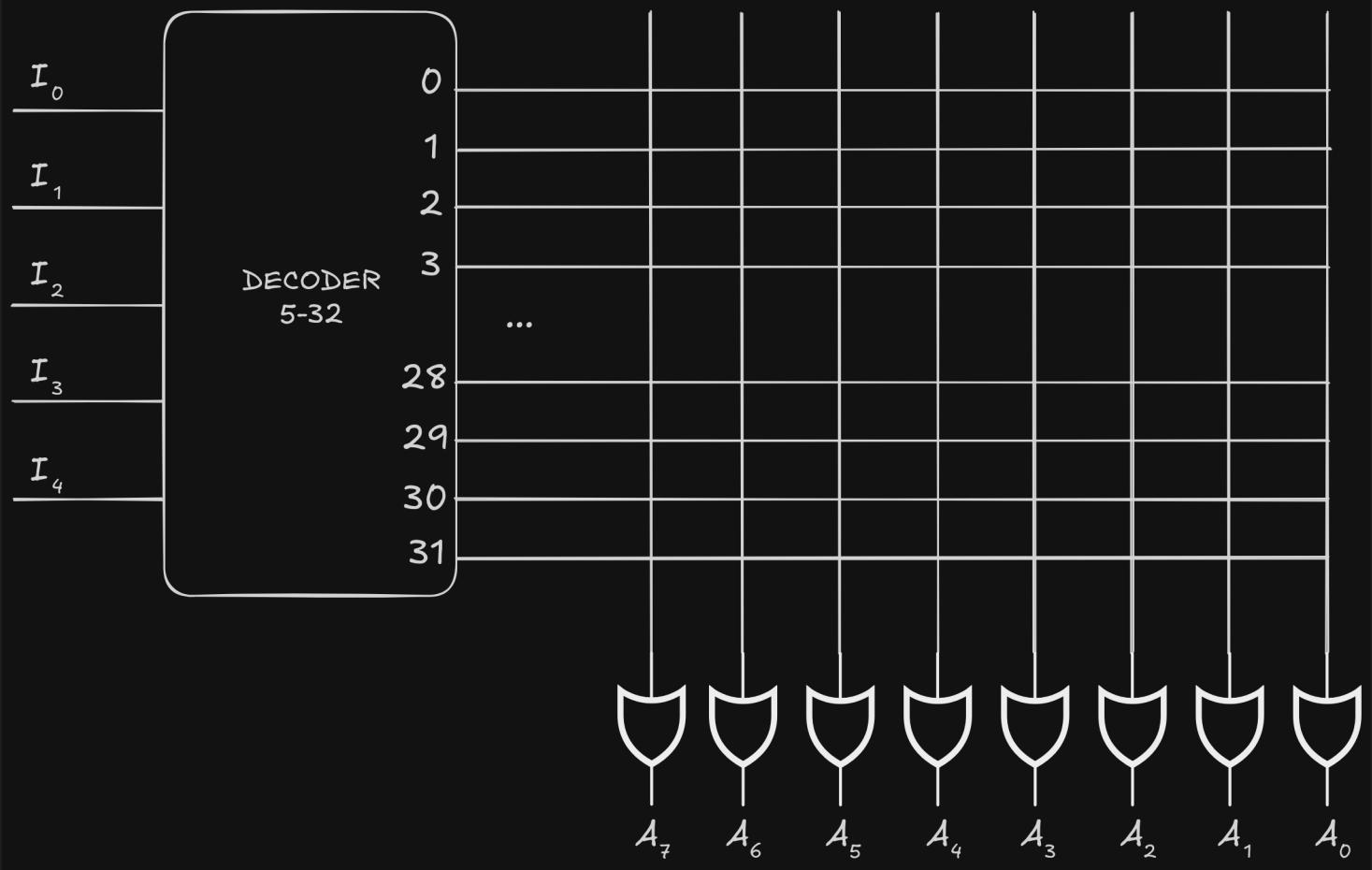
Ne segue dunque che il numero di word immagazzinabili da una ROM è limitato dai k ingressi, con cui il decodificatore può individuare al massimo 2^k word.

Ovviamente, essendo una memoria in solo lettura, non ci sono linee per introdurre dati, tuttavia ci possono essere degli ingressi abilitanti ed uscite a più stati per facilitare la costruzione di strutture multiple di maggiori dimensioni.



Consideriamo una ROM di 32 word da 8 bit ciascuna: usa 5 linee di ingresso per generare $2^5 = 32$ indirizzi (combinazioni binarie), da 0 a 31, tramite un decodificatore 5-32, le cui uscite sono collegate alle 8 porte OR da 32 ingressi l'una, che fanno da codificatore e restituiscono i relativi 8 bit della word.

LOGICA INTERNA DI UNA ROM 32x8



Il contenuto della ROM è descritto da una tabella di verità che riporta, per tutti gli indirizzi possibili, il contenuto della parola memorizzata.

💡 **Riguardo la seguente tavola di verità**

- Le uscite A_i sono il contenuto delle word, e per questo esempio, sono valori casuali
- Per semplicità sono mostrate solo le prime ed ultime 4 righe della tavola di verità, normalmente dovrebbero essere elencate tutte e 32.

TAVOLA VERITÀ (PARZIALE)
ROM

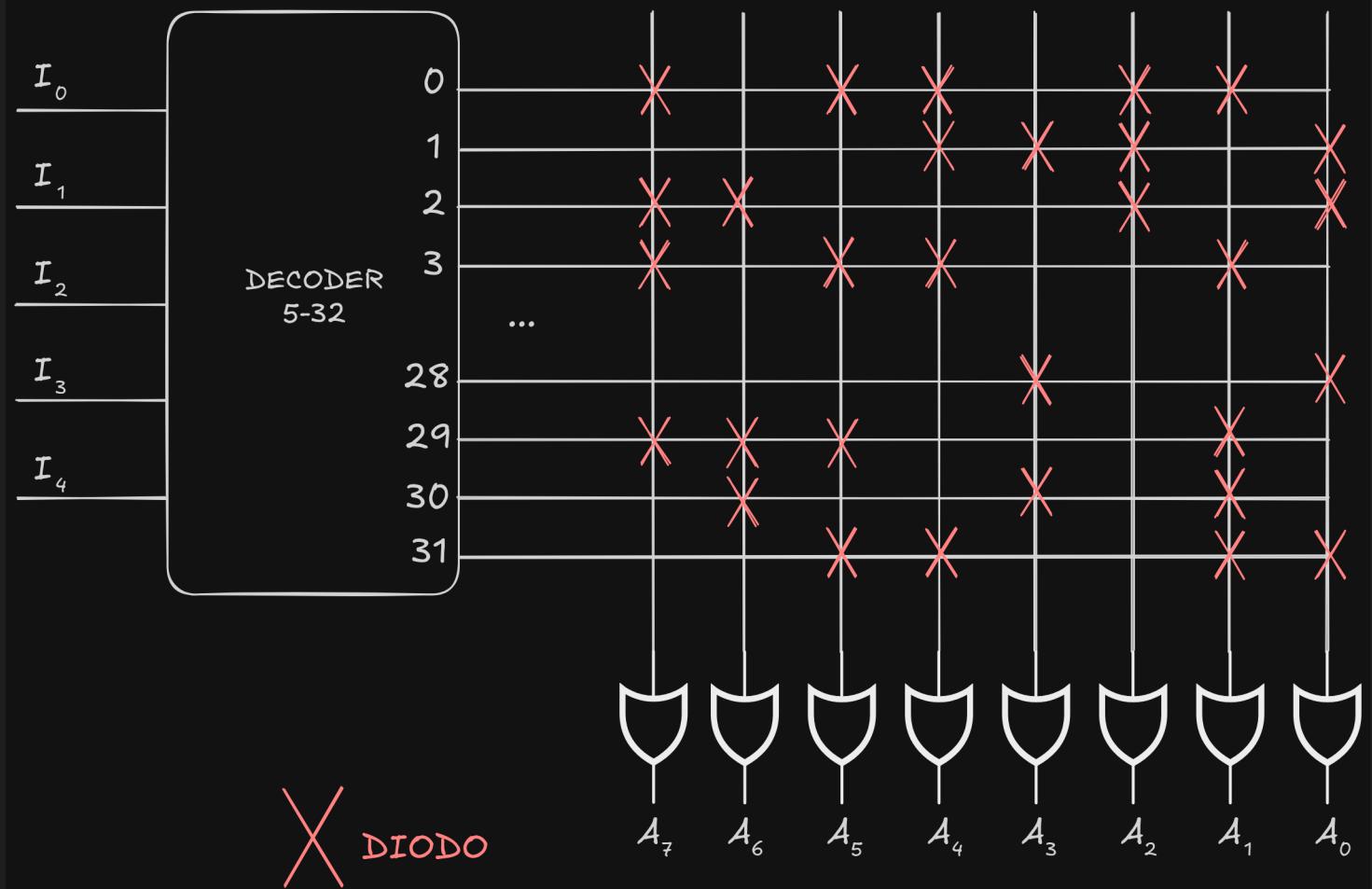
I_4	I_3	I_2	I_1	I_0	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
0	0	0	0	0	1	0	1	1	0	1	1	0
0	0	0	0	1	0	0	0	1	1	1	0	1
0	0	0	1	0	1	1	0	0	0	1	0	1
0	0	0	1	1	1	0	1	1	0	0	1	0

.....

1	1	1	0	0	0	0	0	0	1	0	0	1
1	1	1	0	1	1	1	1	0	0	0	1	0
1	1	1	1	0	0	1	0	0	1	0	1	0
1	1	1	1	1	0	0	1	1	0	0	1	1

Questa tavola di verità genera dunque la seguente logica della ROM:

LOGICA INTERNA DI UNA ROM 32x8



Cerchiamo di capire con un esempio cosa stiamo guardando: la tavola di verità richiede che la word **10110010** sia memorizzata nell'indirizzo **00011**. In corrispondenza agli **0** (cioè dove non è presente la X rossa), il circuito è lasciato aperto, mentre in corrispondenza con gli **1** (dove invece la X è presente) viene saldato un diodo per chiudere il circuito: dunque, quando in ingresso nella ROM c'è la combinazione **00011**, tutte le uscite del decodificatore saranno **0**, tranne l'uscita 3 che invece sarà **1**, dove il segnale si propagherà attraverso i circuiti chiusi dai diodi, le porte OR ed infine alle uscite A_7 , A_5 , A_4 ed A_1 , mentre le altre uscite rimarranno a **0**, non riuscendo il segnale a propagarsi.

✍ Tipi di ROM

Per programmare le ROM si usano quattro tecnologie:

- ROM: viene usata la programmazione tramite maschere durante la fabbricazione
- PROM (*ROM programmable*): si usano fusibili e la memoria può essere programmata dall'utente.

- EPROM (*Erasable programmable ROM*): viene usata la tecnologia cancellabile con base fluttuante.
- EEPROM o E²PROM (*Electrically Erasable programmable ROM*): viene usata la tecnologia cancellabile elettricamente.

Il tipo di tecnologia utilizzata dipende da vari fattori che influenzano la produzione, la programmabilità e le prestazioni.

| PLA

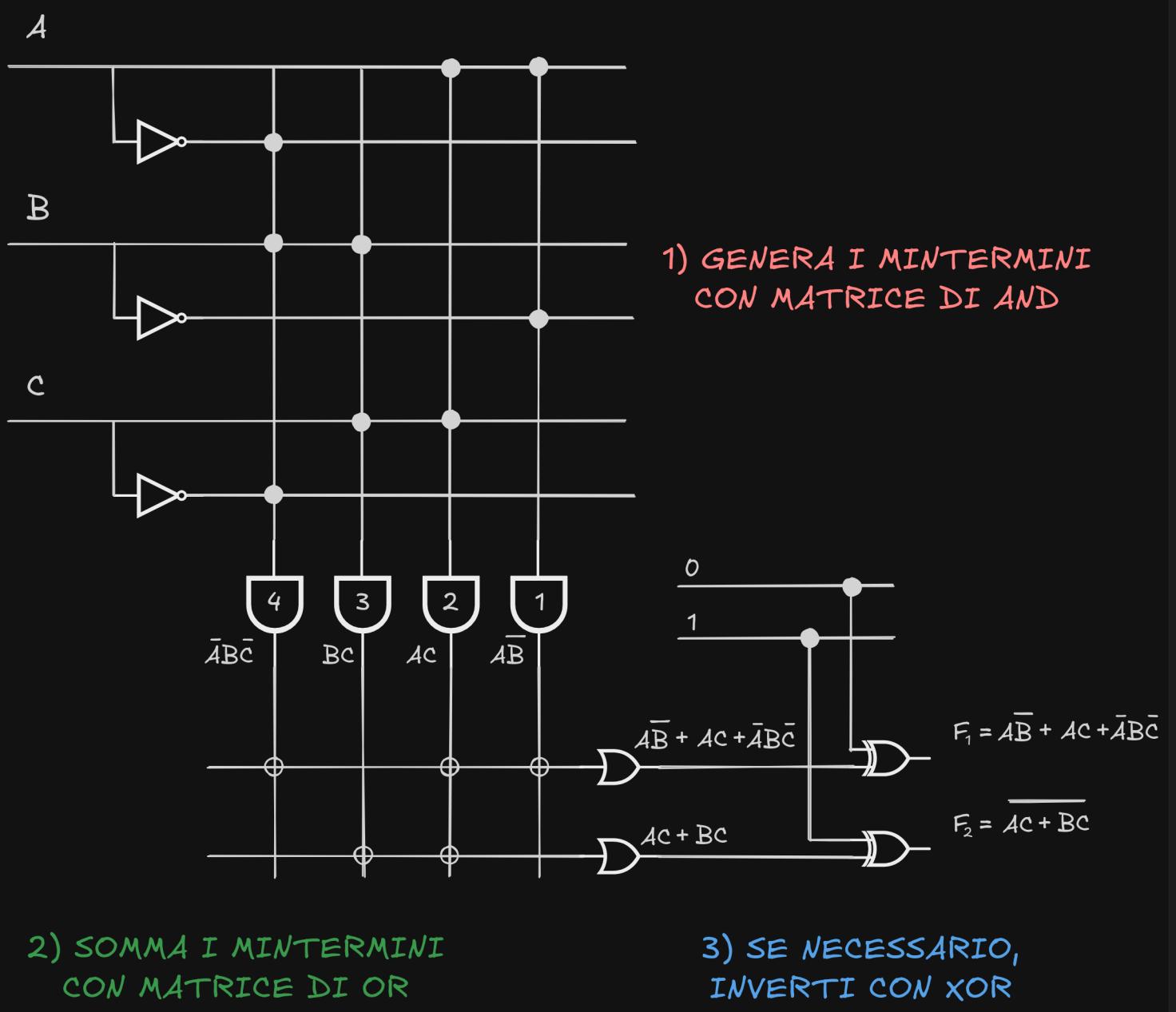
Un PLA (*Programmable Logic Array*) è un circuito combinatorio integrato con n ingressi ed m uscite simile alla ROM, ma a differenza di quest'ultima, non genera tutti i mintermini, ma solo quelli scelti da noi. È composto da 2/3 stadi interni:

- 1) Una matrice di AND programmabile^[4] per generare i mintermini di cui ho bisogno (e che sostituisce il decodificatore nella ROM)
- 2) Una matrice di OR per sommare i mintermini ed ottenere la SOP
- 3) Opzionalmente, delle porte XOR prima delle uscite che permettono di invertire il segnale (perché ricordiamo che $X \oplus 0 = X$ ed $X \oplus 1 = \bar{X}$)

Quindi per n ingressi, k mintermini ed m uscite, la logica interna del PLA consiste di n segnali di ingresso e relative porte NOT, k porte AND ed m porte OR ed XOR.

⚠ Dati nel seguente esempio

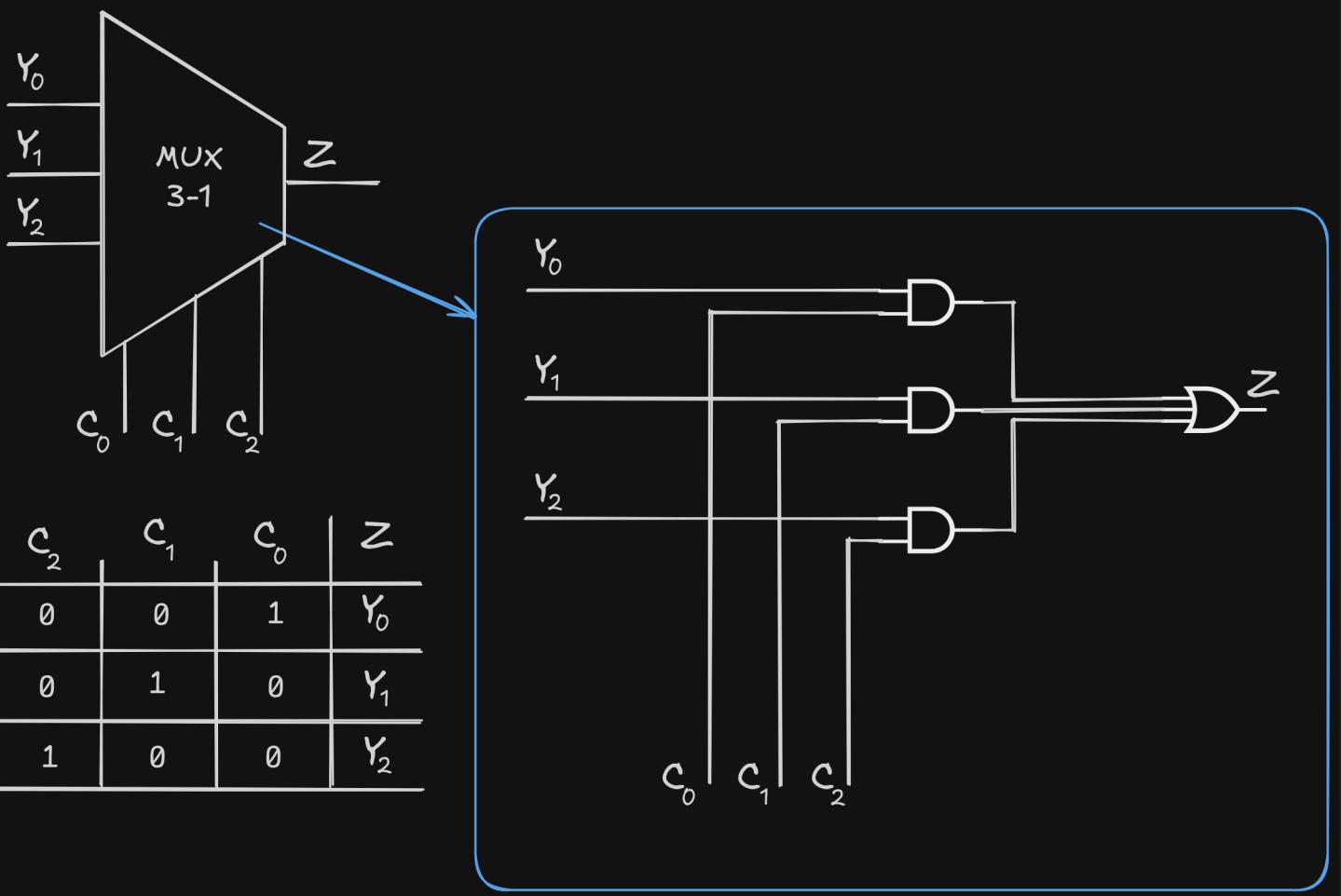
I mintermini generati nel seguente esempio, per semplicità, sono casuali. Normalmente bisognerebbe ricavarsi prima la forma SOP minimale tramite tavole di verità e mappe di Karnaugh dell'espressione da esprimere con il PLA per sapere quali mintermini è necessario generare.



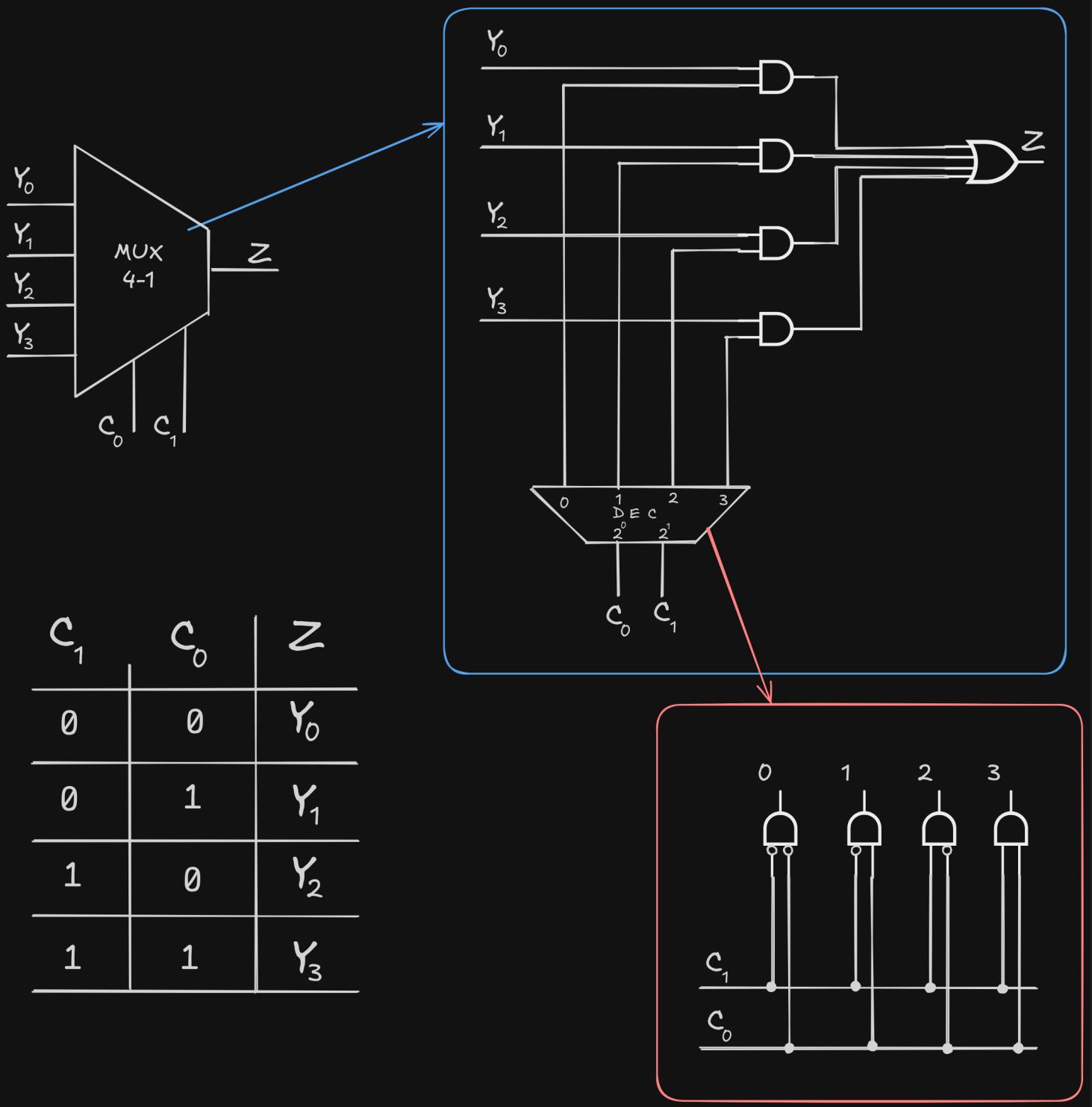
Multiplexer

Il multiplexer (*Mux*) è un circuito combinatorio che, date n linee di ingresso, ne fa passare solamente una in uscita. Per fare ciò, si usano n linee di controllo, ognuna associata ad un ingresso.

Il multiplexer è implementato tramite una serie di porte AND che confluiscono in una porta OR. Solo uno dei segnali di controllo può assumere il valore 1 in un determinato istante.



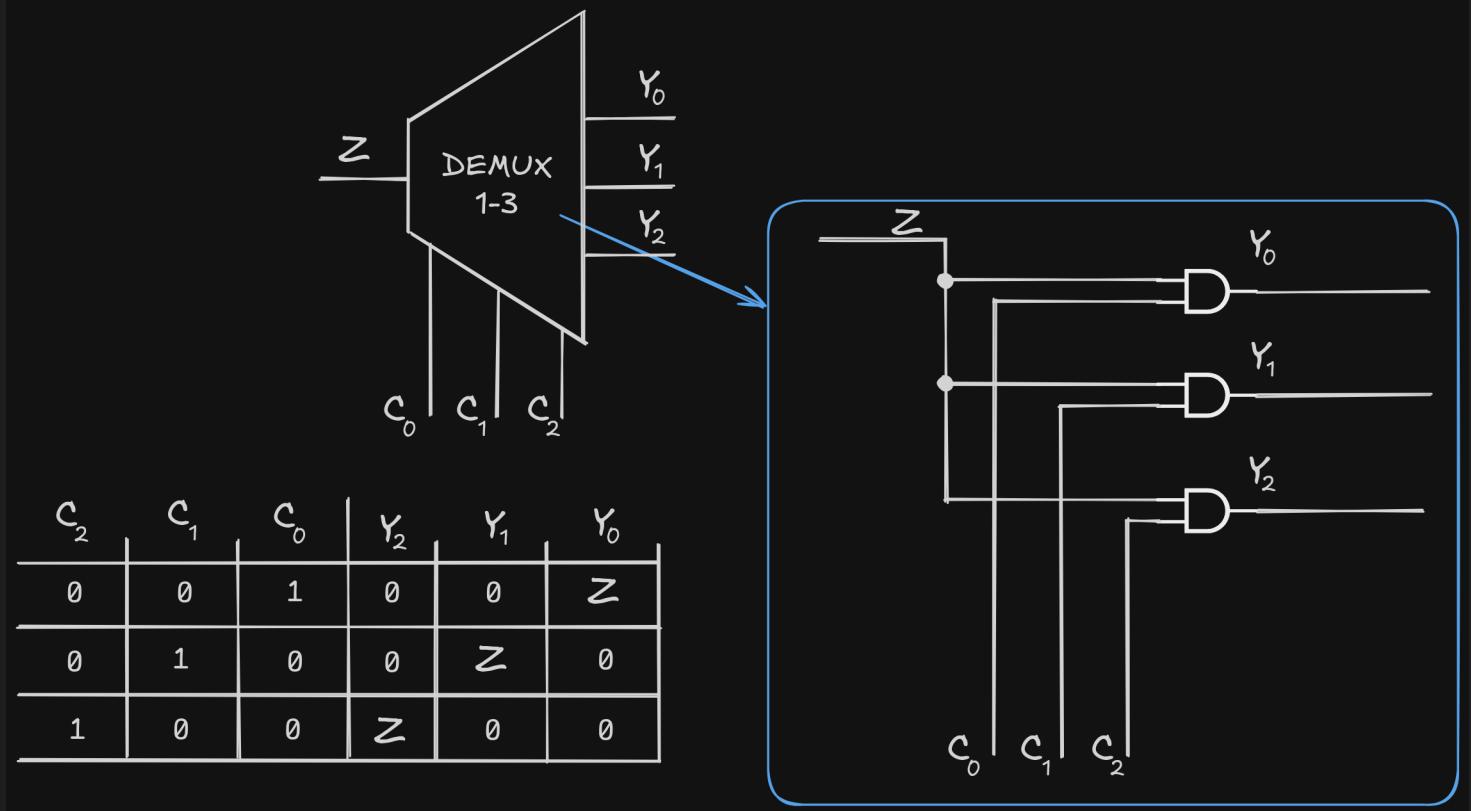
Tuttavia, è possibile ottimizzare il multiplexer diminuendo le linee di controllo: se ho n ingressi, allora per identificarli mi bastano $\log_2 n$ linee di controllo per rappresentare in binario il numero dell'ingresso da lasciar passare, ed ottengo i segnali equivalenti da un decodificatore.



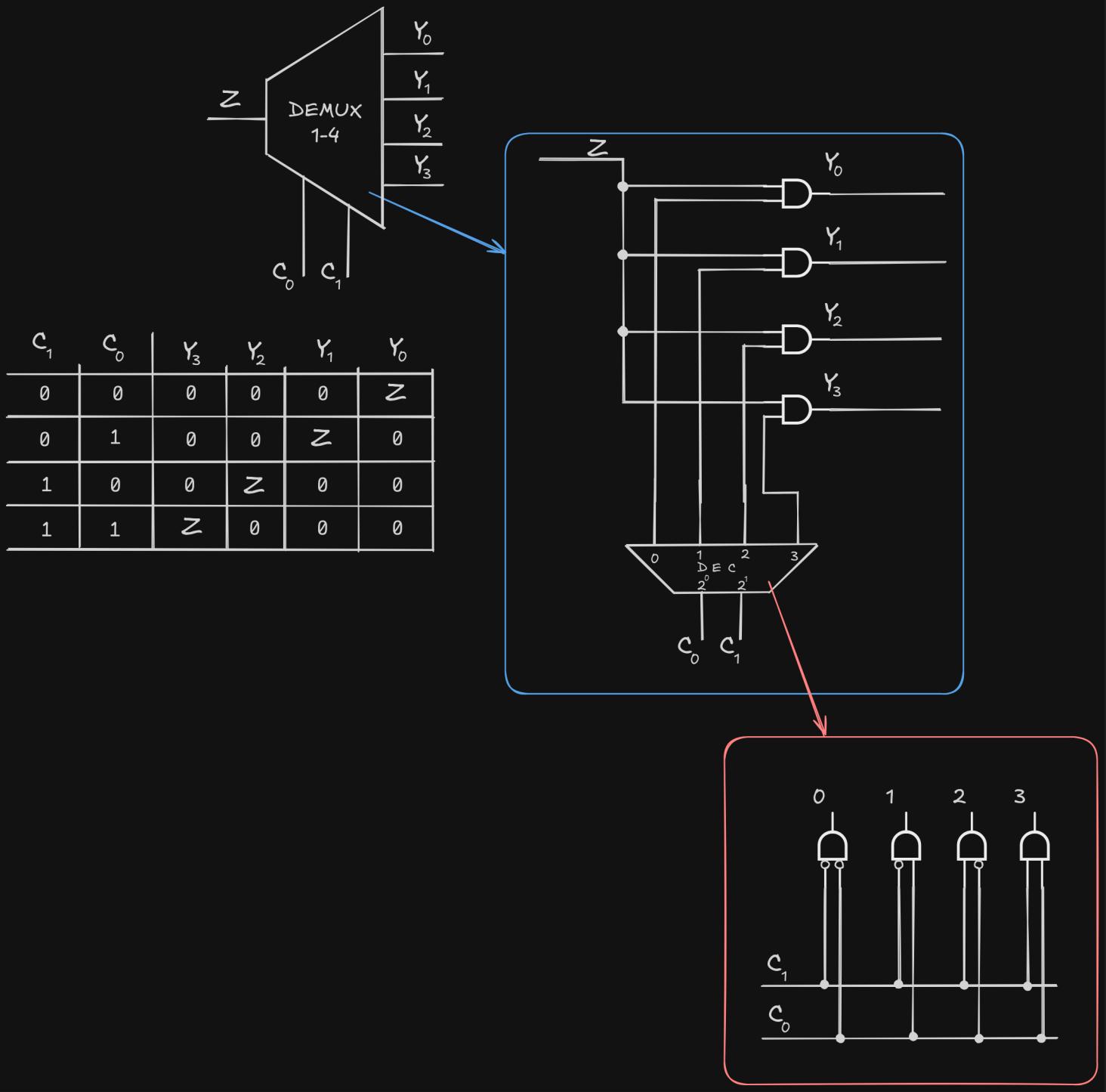
| Demultiplexer

Il Demultiplexer (Demux) è un circuito combinatorio che esegue l'operazione inversa del multiplexer: riceve informazioni da una singola linea in ingresso, ed n segnali di controllo indicano su quale delle n uscite trasmettere il segnale di ingresso. Solo uno dei segnali di controllo può assumere il valore 1 in un determinato istante.

Il demultiplexer è implementato tramite una serie di porte AND.



Tuttavia, è possibile ottimizzare il demultiplexer diminuendo le linee di controllo: se ho n uscite, allora per identificarli mi bastano $\log_2 n$ linee di controllo per rappresentare in binario il numero dell'uscita in cui trasmettere il segnale di ingresso, ed ottengo i segnali equivalenti da un decodificatore.



Comparatore

I comparatori servono, ovviamente, a comparare gli ingressi, ottenendo in uscita delle informazioni sugli ingressi.

Si possono dividere in:

- Comparatore logico
- Comparatore aritmetico

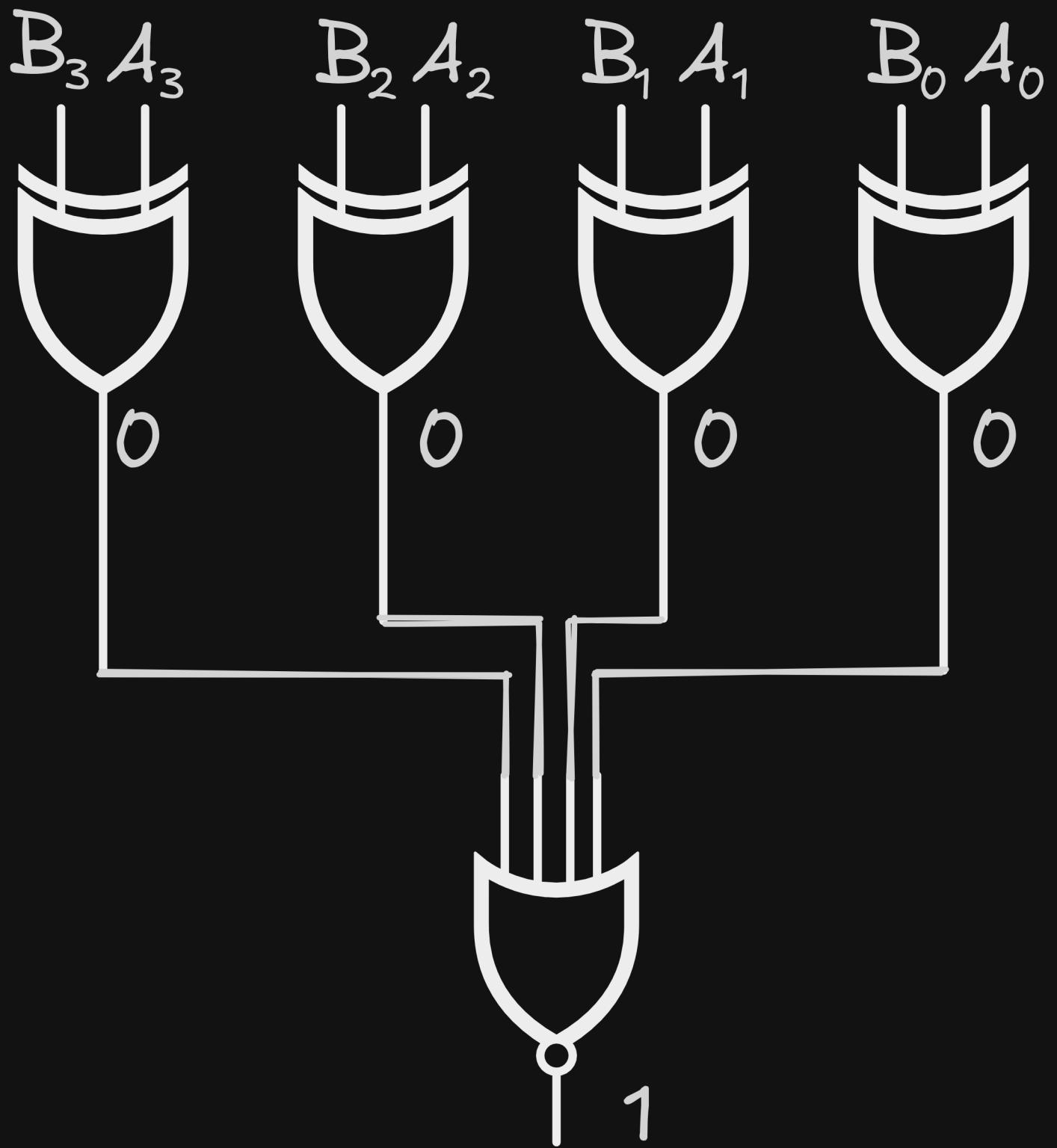
Comparatore logico

Un comparatore logico è un circuito combinatorio che verifica l'uguaglianza tra due sequenze di bit, restituendo **1** se lo sono oppure **0** altrimenti.

Per fare ciò, i 2 bit da confrontare vengono collegati ad una porta XOR, che dunque restituisce **0** se sono uguali o **1** se non lo sono. Le uscite di tutte le porte XOR (Se devo confrontare l'uguaglianza tra più segnali) vengono poi collegate ad una porta NOR, la quale restituirà **1** se e solo se tutte le uscite delle porte XOR sono **0** (cioè tutti i segnali esaminati sono uguali), altrimenti restituisce **0**.

$A = 1001$

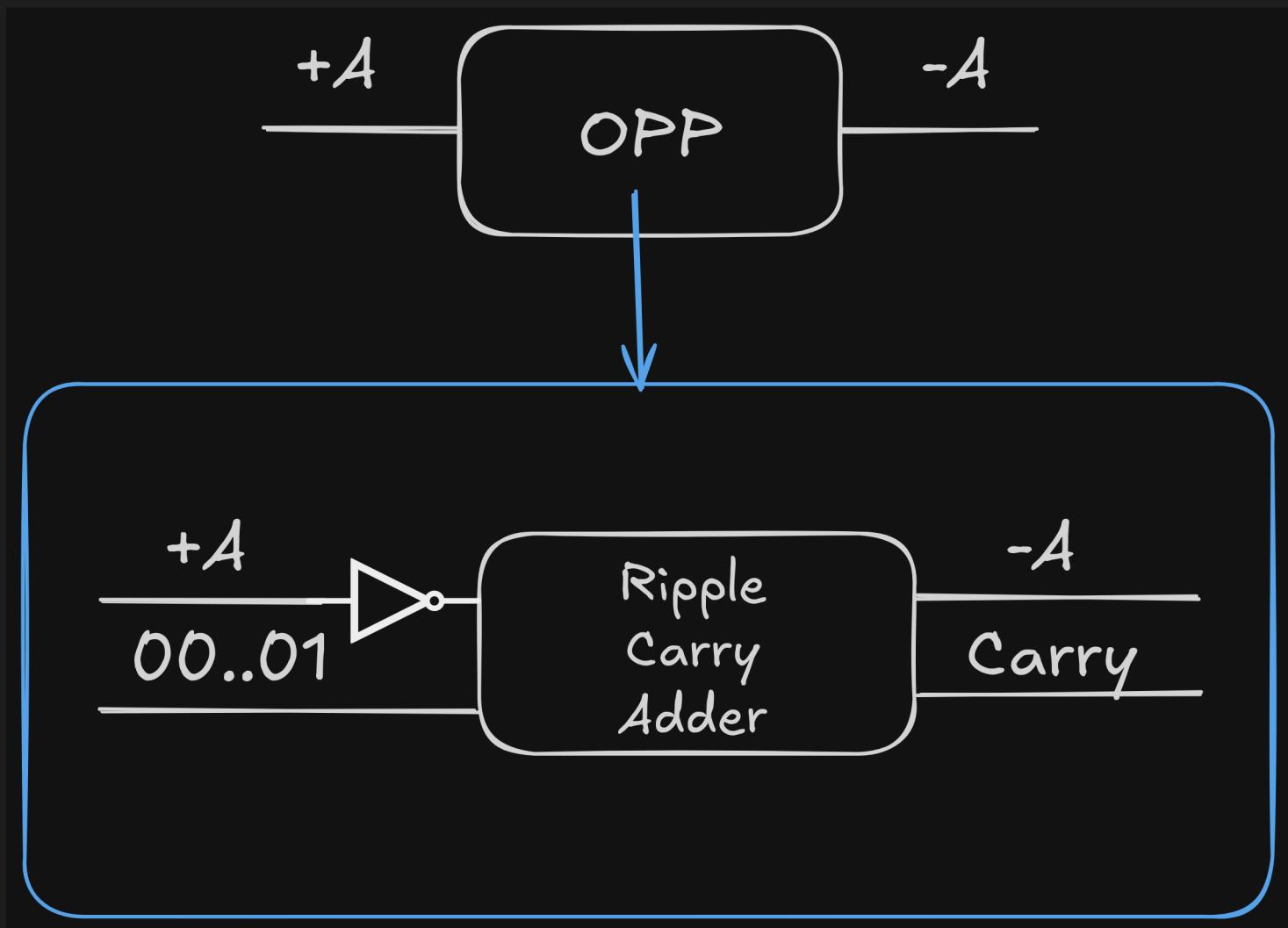
$B = 1001$



Comparatore aritmetico

Il comparatore aritmetico è un circuito combinatorio che date le sequenze di bit A e B , controlla se $A - B > 0$, cioè, se $A > B$. Prende in ingresso le due sequenze da confrontare, e restituisce in uscita un segnale che indica se $A > B$, un altro segnale che indica se $A = B$, e se entrambi i segnali sono 0, allora sappiamo che $A < B$.

Dunque, innanzi tutto ci serve, dato B , ricavarci $-B$. Per farlo basta ricordarsi, [come già detto in precedenza](#), che $-B = \overline{B} + 1$ (Per esempio, dato $B = -4 = 1100$ in CA2, $-B = 0011 + 1 = 0100 = 4$). Possiamo implementare ciò con un Ripple-Carry Adder che somma la negazione del primo ingresso con 1.



Adesso bisogna capire se il risultato è uguale o maggiore di 0. Non ci basta un comparatore logico, perché se da come uscita 0, non sappiamo se è perché $A > B$ o $A < B$. Ci servono dei moduli che, dati i bit delle sequenze da confrontare (a e b) e gli esiti delle comparazioni precedenti ($c_>$ e $c_=$), ci dicano se siano uguali o uno maggiore dell'altro ($c'_>$ e $c'_=$), per poi propagare questa informazione al comparatore dei bit successivi, fino ad arrivare a quelli più significativi.

Per ogni caso,

a e b sono i bit di A e B che sto comparando al momento.

$c_>$ e $c_=$ sono i bit che rappresentano l'esito della comparazione precedente.

$c'_>$ e $c'_=$ sono l'esito della comparazione tra a e b .

- $c_>$ e $c_=$ non possono mai essere 1 contemporaneamente, perché A non può essere sia uguale che maggiore di B e quindi uso il simbolo del *don't care* δ .

Possono invece essere contemporaneamente 0, perché sta a significare che A non è né maggiore né uguale a B , e cioè $A < B$.

- Se a e b sono uguali, $c'_>$ e $c'_=$ sono uguali a $c_>$ e $c_=$, perché:

- se A e B erano uguali quando ho esaminato i bit precedenti, allora adesso continuano ad esserlo.

- Se A era più grande di B quando ho esaminato i bit precedenti, anche se ora hanno lo stesso bit uguale, A resta comunque più grande di B .

- Viceversa, se B era più grande di A quando ho esaminato i bit precedenti, anche se ora hanno lo stesso bit uguale, B resta comunque più grande di A

- Se $a = 0$ e $b = 1$, $c'_>$ e $c'_=$ sono entrambi 0, perché:

- se A e B erano uguali quando ho esaminato i bit precedenti, allora adesso B è più grande di A .

- Se A era più grande di B quando ho esaminato i bit precedenti, adesso B ha il bit più significativo esaminato fin'ora uguale ad 1, mentre A no, quindi B è sicuramente più grande di A (per esempio, $B = 1000$ è più grande di $A = 0111$).

- Se B era più grande di A quando ho esaminato i bit precedenti, ora lo è ancora di più.

- Se $a = 1$ e $b = 0$, allora $c'_>$ è 1 e $c'_=$ è 0, perché:

- se A e B erano uguali quando ho esaminato i bit precedenti, allora adesso A è più grande di B .

- Se B era più grande di A quando ho esaminato i bit precedenti, adesso A ha il bit più significativo esaminato fin'ora uguale ad 1, mentre B no, quindi A è sicuramente più grande di B (per esempio, $A = 1000$ è più grande di $B = 0111$).

- Se A era più grande di B quando ho esaminato i bit precedenti, ora lo è ancora di più.

Come abbiamo già detto, nelle mappe di Karnaugh i *don't care* si possono usare per minimizzare le espressioni booleane.

a	b	$c_>$	$c_=$	$c'_>$	$c'_=$
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	δ	δ
0	1	0	0	0	0
0	1	0	1	0	0
0	1	1	0	0	0
0	1	1	1	δ	δ
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	δ	δ
1	1	0	0	0	0
1	1	0	1	0	1
1	1	1	0	1	0
1	1	1	1	δ	δ

$c'_>$

$a \backslash b$	00	01	11	10
00	0	0	δ	1
01	0	0	δ	0
11	0	0	δ	1
10	1	1	δ	1

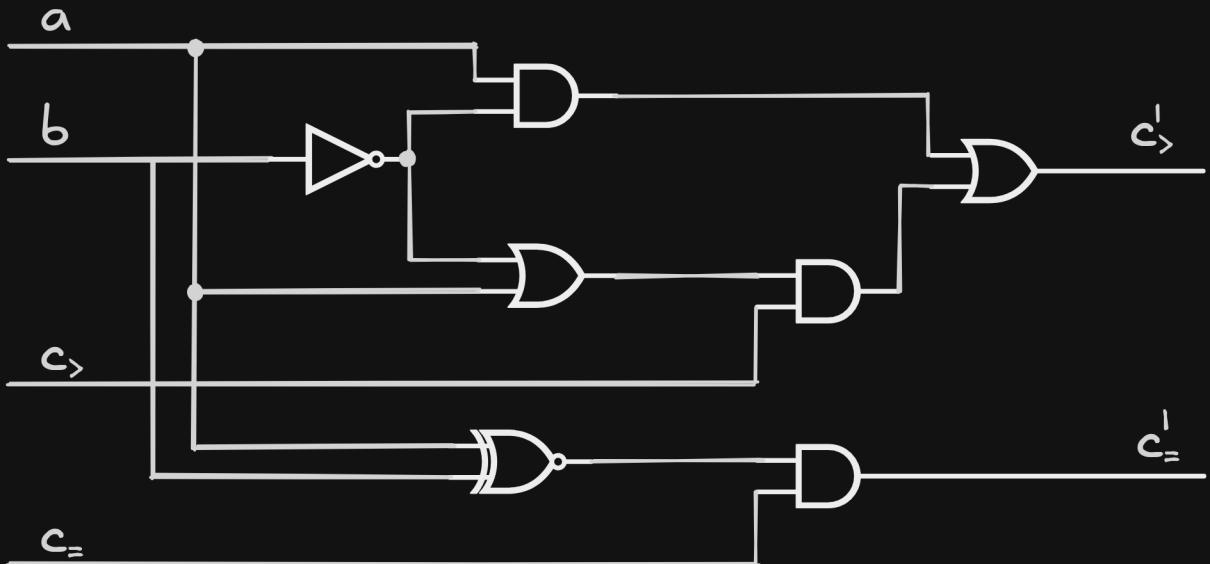
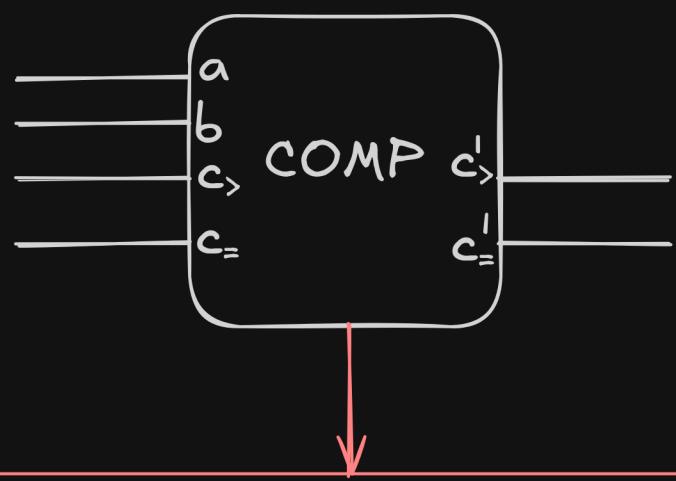
$$\bar{ab} + ac_> + \bar{bc}_> = \\ ab + (a + \bar{b})c_>$$

$c'_=$

$a \backslash b$	00	01	11	10
00	0	1	δ	0
01	0	0	δ	0
11	0	1	δ	0
10	0	0	δ	0

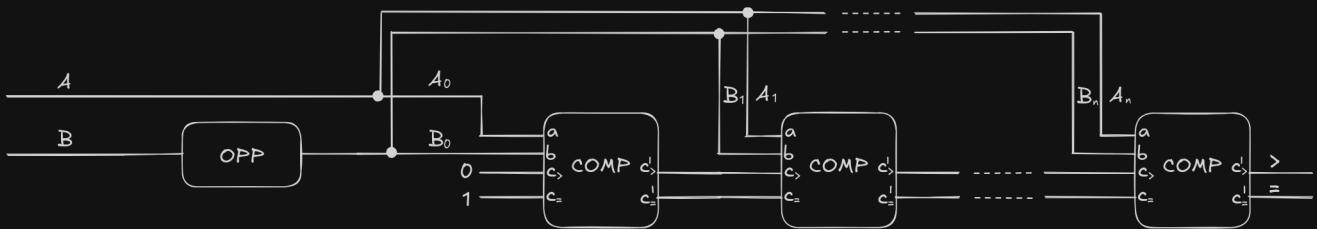
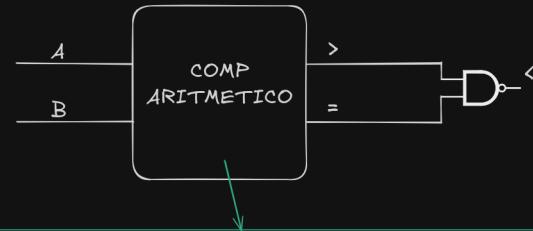
$$\bar{abc}_> + abc_> = (\bar{ab} + ab)c_> = \\ (a \oplus b)c_>$$

Possiamo ricavare dall'espressione il circuito del comparatore:



Quando si comparano i bit meno significativi della sequenza, $c_>$ e $c_=$ sono rispettivamente 0 ed 1, non essendoci bit precedenti su cui basarsi.

Adesso abbiamo tutto il necessario: trasformiamo B in $-B$, e poi usiamo n (dove n è uguale al numero di bit della sequenza) comparatori in cascata per ottenere $c'_>$ e $c'_=$ che ci dicono se $A = B$ o $A > B$, e se sono entrambi 0, allora $A < B$ e possiamo farli passare per una NAND per ottenere 1 quando $A < B$.



| ALU

L' ALU (*Arithmetic Logical Unit*) è un circuito che si occupa di eseguire operazioni *bitwise*^[5]. È inoltre il sistema che si occupa di eseguire i calcoli sui numeri naturali ed interi nelle CPU (spesso i float sono invece calcolati dalla *floating point unit*).

In genere la ALU ha due ingressi per gli operandi, due segnali di controllo che selezionano la operazione da effettuare sugli operandi (nelle CPU, questi segnali vengono emessi dalla CU (*Control Unit*)) e cinque uscite:

- *R*: indica il risultato dell'operazione
- *N*: indica se il risultato è negativo
- *Z*: indica se il risultato è zero.
- *C*: indica il riporto in uscita dell'operazione eseguita dalla ALU
- *W*: indica se è avvenuto un overflow

| R - Risultato e C - Carry

Il risultato *R* ed il carry *C* sono l'output di un [Ripple-Carry Adder](#).

| N - Negativo

Il negativo *N* è [il bit più significativo del risultato](#).

| Z - Zero

Lo zero *Z* è l'output di un [comparatore logico](#) che confronta i bit del risultato con altrettanti bit impostati a 0.

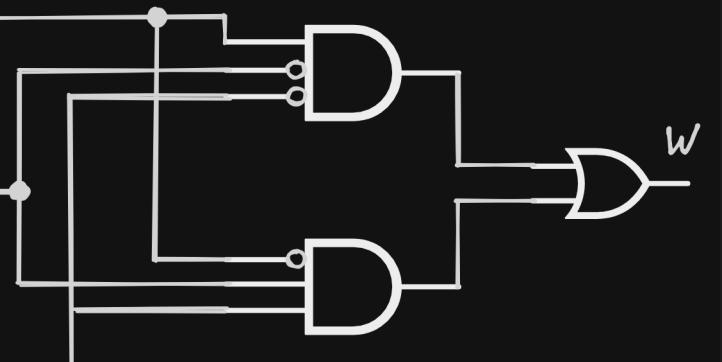
| W - Overflow

Per individuare un overflow, devo controllare se sommando due numeri positivi ho ottenuto un negativo (cioè, i bit più significativi dei due operandi A e B e del risultato R sono rispettivamente 0 , 0 , ed 1) oppure se sommando due numeri negativi ho ottenuto un positivo (cioè, i bit più significativi dei due operandi A e B e del risultato R sono rispettivamente 1 , 1 , e 0). Se mi trovo in uno di questi due casi, allora il segnale di overflow W deve essere impostato a 1 , altrimenti a 0 .

Bit più significativo di R

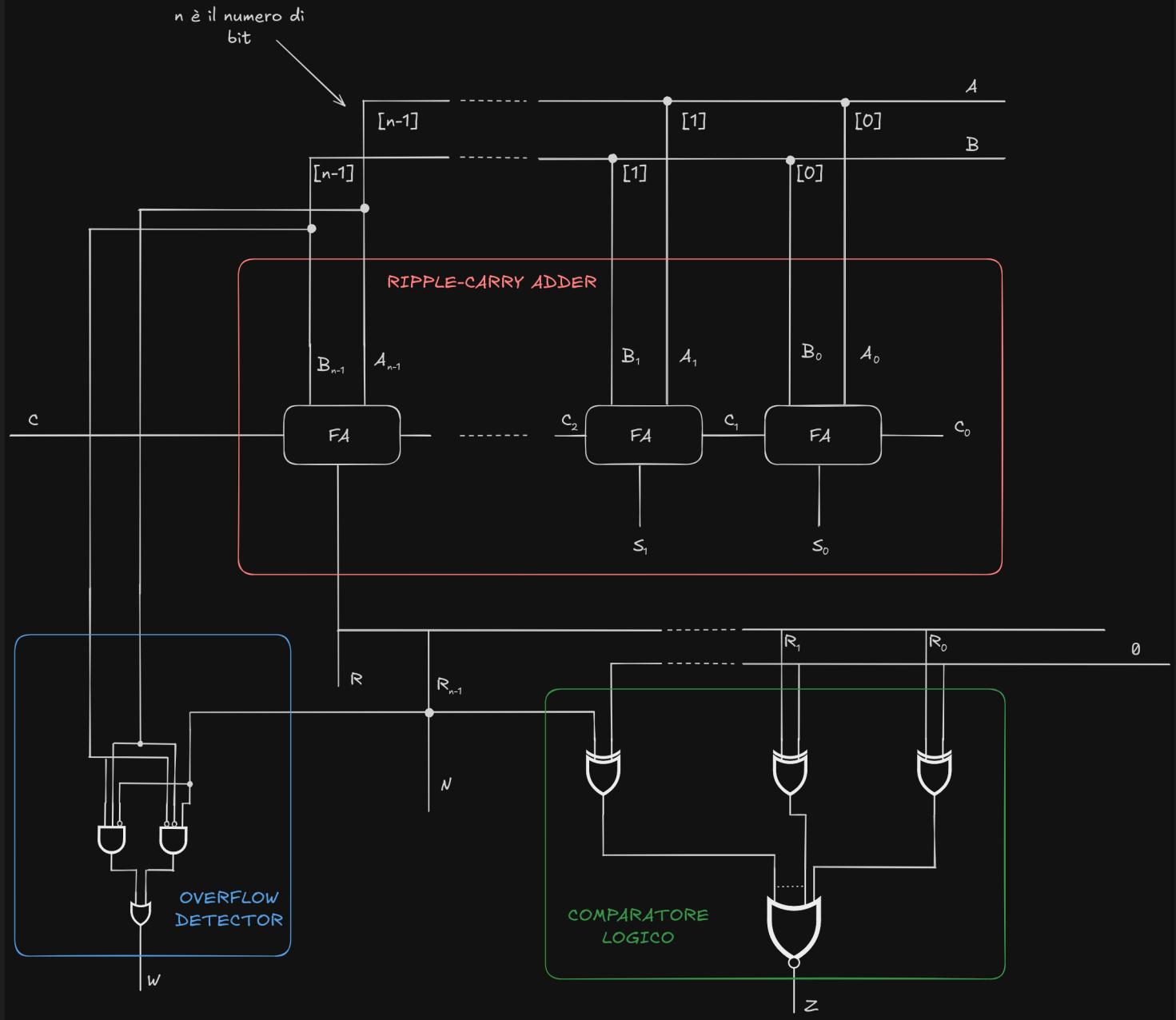
Bit più significativo di A

Bit più significativo di B



| Implementazione di una semplice ALU

Adesso che sappiamo quali sono i componenti usati da una ALU per generare i segnali di uscita, proviamo a vedere una semplice implementazione.



- 1) Perché se X ed Y sono entrambi 1 , allora XY è 1 e quindi il risultato dell'espressione è 1 indipendentemente dal valore di Z . Invece quando X ed Y sono entrambi 0 , XY è 0 ed anche $Z(X + Y)$ è 0 , e dunque anche questa volta il risultato dell'espressione è 0 indipendentemente dal valore di Z ↵
- 2) Perché se X ed Y sono diversi, la prima parte dell'espressione XY sarà 0 , mentre nella seconda parte $(X + Y)$ sarà 1 e dunque il risultato dell'intera espressione è dettato dal valore di Z . ↵
- 3) Una word è un'unità di dati indivisibile e letta per intero che rappresenta il dato: per esempio nelle CPU MIPS una word è grande 4 byte (32 bit) e quindi per leggere il dato rappresentato dalla word vanno letti tutti i 32 bit corrispondenti a quella word. La dimensione di una word varia tra architetture e tipi di ROM. ↵
- 4) Cioè, "posso scegliere la posizione dei pallini". ↵
- 5) Su singoli bit o sequenze di bit. ↵