



Materiale di PED



Alcune regole per usufruire correttamente delle seguenti dispense:

1. questo materiale non deve essere letto con occhi passivi, che accettano tutto ciò che viene detto, ma, essendo creato da studenti, ci potrebbero essere degli errori. Detto questo, utilizzate lo **spirito critico** per capire gli argomenti e non fermatevi a ciò che è scritto;
2. per navigare agevolmente il materiale all'interno di una pagina, ognuna di esse è fornita di un indice a comparsa sul suo lato destro;
3. nelle pagine di teoria gli approfondimenti saranno indicati attraverso il simbolo (*). Essi sono argomenti che non risultano strettamente necessari al superamento dell'esame;
4. se notate un errore, qualcosa che non risulta chiaro oppure un concetto che potrebbe essere spiegato meglio, potete notificare chi ha creato la pagina (indicato all'inizio della relativa pagina di teoria oppure all'inizio del relativo esercizio nelle esercitazioni) cercando il suo contatto Telegram sulla [pagina dei crediti](#);
5. molti concetti che sono stati dati per scontati, o addirittura sono stati spiegati in maniera superficiale durante le lezioni, sono stati trattati nelle relative pagine di teoria attraverso l'ausilio di materiale vario trovato nel Web e nelle slide relative del corso, oltre che da conoscenze pregresse dei collaboratori al progetto.



Dato che il corso è ancora in fase di erogazione, le pagine fornite non comprendono ancora tutti gli argomenti che verranno trattati all'esame, e non possiamo neanche prevedere cosa sarà più o meno importante ai fini dell'esame, essendo un corso nuovo.

Man mano che completiamo argomenti, essi saranno pubblicati in questa pagina.

Teoria

In questa sezione saranno presenti tutti i concetti di teoria illustrati nel corso, oltre a qualche argomento aggiuntivo che viene tralasciato, ma che non risulta necessariamente ovvio per tutti.

Concetti di base

Funzioni e porte logiche



Circuiti logici

Bistabili e Flip-Flop



Tempi di clock, di setup, di hold e frequenza massima

Registri

Contatori e divisori

Macchina a stati finiti (FSM)

Forme Canoniche

Mappe di Karnaugh (K-Map)

Ritardi e alee delle reti combinatorie

Ottimizzazione per FSM

ASM → SOON

 Circuiti Aritmetici

 Unità di controllo e datapath

Register Transfer Level (RTL) → SOON

Tecnologia CMOS

 Transistori

 Implementazione delle porte logiche CMOS

 Tempo di propagazione (CMOS)

 Tempi di salita e di discesa

Analisi delle prestazioni di circuiti logici → SOON

Verilog

 Basi del linguaggio Verilog

 Modellazione strutturale

 Modellazione a livello di porta logica (GATE-LEVEL)

 Modellazione RTL

 Modellazione comportamentale

 Testbench in Verilog

 Descrizione di registri, ROM, RAM in Verilog

Esercitazioni

In questa sezione saranno presenti le esercitazioni trattate durante il corso, con spiegazioni approfondite riguardo alle strategie adottate nelle soluzioni e richiami utili alla teoria.

1 [Esercitazione 1 - Progetto guidato di un circuito](#)

2 [Esercitazione 2 - Minimizzazione e mappe di Karnaugh](#)

3 [Esercitazione 3 - Tempi di salita, di discesa, di propagazione e frequenza massima](#)

5 [Esercitazione 5 + iverilog e gtkwave](#)

6 [Esercitazione 6 - Logica programmabile e memorie a semiconduttore](#)

Funzioni e porte logiche

▼ Creatore originale: @LucaCaffa

- @Giacomo Dandolo (14/04/2025): aggiunta nomenclatura per le funzioni logiche e per l'algebra di Boole.

[Grandezze elettriche](#)

[Threshold](#)

[Funzioni logiche](#)

[Funzione logica NOT](#)

[Funzione logica AND](#)

[Funzione logica NAND](#)

[Funzione logica OR](#)

[Funzione logica NOR](#)

[Funzione logica XOR](#)

[Funzione logica XNOR](#)

[Porte logiche](#)

[Reti Logiche](#)

[Diagramma temporale](#)

[Algebra di Boole](#)

[Assiomi](#)

[Teoremi](#)

[Proprietà](#)

[Leggi di De Morgan](#)

Grandezze elettriche

Utilizziamo gli stati logici 1 e 0 per rappresentare il comportamento delle grandezze elettriche. La convenzione maggiormente usata è la seguente:

- Tensione **alta** → 1 logico;
- Tensione **bassa** → 0 logico.

Threshold

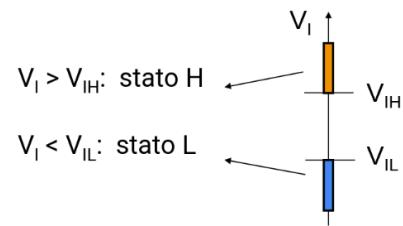
In un modello reale, una tensione non è mai costante, ma spesso presenta dei rumori (ha diverse oscillazioni), quindi non basta considerare tensione bassa e alta.

Si introduce il concetto di **threshold** (soglia), definito attraverso una tensione di soglia chiamata V_T . Ipotizzando di avere una generica tensione V , si ha che:

- se $V > V_T \rightarrow 1$ logico;
- se $V < V_T \rightarrow 0$ logico.

Lo stato logico viene trovato confrontando la tensione di ingresso V_I con quella di soglia V_T . La soglia non è un valore fisso, ma un range $V_T = [V_{IL}, V_{IH}]$, dove:

- se $V_I > V_{IH} \rightarrow 1$ logico;
- se $V_I < V_{IL} \rightarrow 0$ logico;
- la tensione tra V_{IL} e tra V_{IH} non è definita.



Rappresentazione del concetto di threshold

Funzioni logiche

Funzione logica NOT

La funzione logica NOT indica la negazione di un ingresso, quindi se in ingresso abbiamo 0, in uscita avremo 1 e viceversa.

$$y = \text{NOT } x = \bar{x}$$

x	y
0	1
1	0

La dicitura descritta indica che y è l'inverso di x .

Le varie notazioni sono:

$$\text{NOT } x = \bar{x} = x' = !x = \sim x = x^*$$

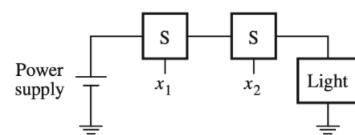


E' importante notare che la funzione logica NOT può essere applicata a ogni funzione logica, e che ogni funzione logica negata ha la tabella di verità con valori di uscita opposti alla funzione non negata.

Funzione logica AND

Nella figura:

- "Power Supply" è l'alimentatore;
- x_1 e x_2 sono due interruttori (Switch);
- Light è la lampadina, che rappresenta la nostra uscita y .



(a) The logical AND function (series connection)

Rappresentazione di un circuito che implementa la funzione logica AND

Per far accendere la lampadina abbiamo bisogno che gli switch siano entrambi chiusi, in modo da far arrivare la tensione alla lampadina.

Questo concetto si traduce con la seguente funzione logica AND:

$$f(x_1, x_2) = y = x_1 \text{ AND } x_2 = x_1 \cdot x_2 = x_1 x_2$$

Questa funzione dice che $y = 1$ solo se x_1 e x_2 sono entrambi 1.

x_1	x_2
0	0
0	1
1	0
1	1

Funzione logica NAND

La funzione logica AND negata si chiama **NAND**, cioè "NOT AND".

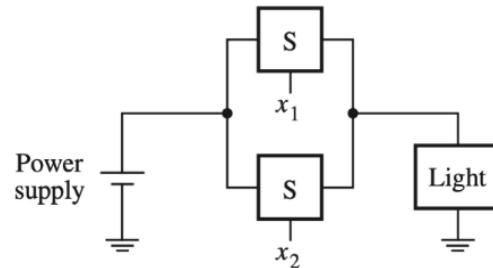
$$f(x_1, x_2) = y = x_1 \text{ NAND } x_2 = \overline{x_1 \cdot x_2}$$

x_1	x_2
0	0
0	1
1	0
1	1

Funzione logica OR

Nella [figura](#):

- "Power Supply" è l'alimentatore;
- x_1 e x_2 sono due interruttori (**S**witch);
- Light è la lampadina, che rappresenta la nostra uscita y .



(b) The logical OR function (parallel connection)

Rappresentazione di un circuito che implementa la funzione logica OR

Per far accendere la lampadina basta che uno dei due interruttori sia chiuso, in modo da far passare la tensione.

Questo concetto si traduce con la seguente funzione logica OR:

$$f(x_1, x_2) = y = x_1 \text{ OR } x_2 = x_1 + x_2$$

Questa funzione dice che $y = 1$ se x_1 o x_2 valgono 1.

x_1	x_2
0	0
0	1
1	0
1	1

Funzione logica NOR

La funzione logica OR negata si chiama **NOR**, cioè "NOT OR".

$$f(x_1, x_2) = y = x_1 \text{ NOR } x_2 = \overline{x_1 + x_2}$$

x_1	x_2
0	0
0	1
1	0
1	1

Funzione logica XOR

La funzione logica **XOR** corrisponde ad un OR esclusivo, in cui l'uscita è 1 se uno e uno solo degli ingressi vale 1.

$$f(x_1, x_2) = y = x_1 \text{ XOR } x_2 = x_1 \oplus x_2$$

x_1	x_2
0	0
0	1
1	0

x_1	x_2
1	1

Funzione logica XNOR

La funzione logica **XNOR** corrisponde ad uno XOR negato, in cui l'uscita è 1 se gli ingressi sono uguali.

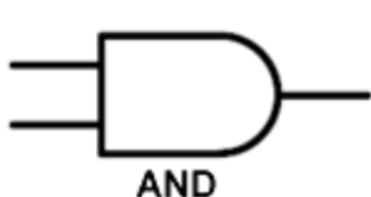
$$f(x_1, x_2) = y = x_1 \text{ XNOR } x_2 = \overline{x_1 \oplus x_2}$$

Lo XNOR può anche essere trovato, nella letteratura, come NXOR, EXNOR, ENOR o XAND.

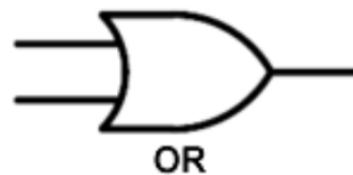
x_1	x_2
0	0
0	1
1	0
1	1

Porte logiche

Le porte logiche sono dei circuiti che contengono transistori connessi in modo da comportarsi come una funzione logica. Le porte logiche più utilizzate sono quelle a due ingressi, ma ne esistono anche a più di due ingressi.



Porta logica AND



Porta logica OR



Porta logica NOT



NAND

Porta logica NAND



NOR

Porta logica NOR



XOR

Porta logica XOR



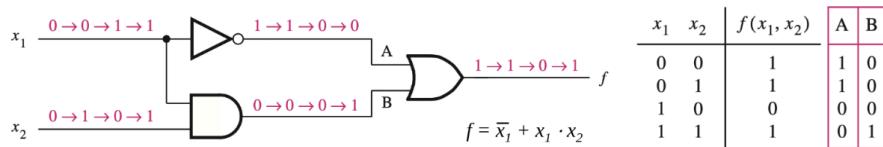
NXOR

Porta logica XNOR

Reti Logiche

Una combinazione di porte logiche crea una "rete logica", anche chiamato "circuito logico".

Possiamo analizzare il comportamento di una rete logica per determinarne la funzione e determinare l'uscita in base agli ingressi.



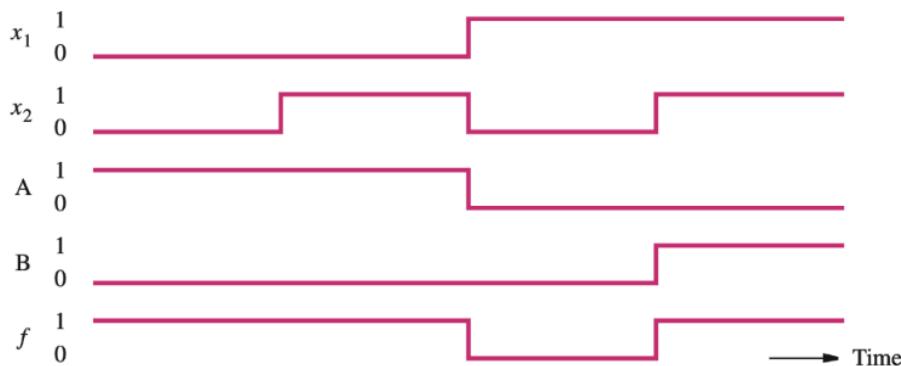
Esempio di rete logica con tabella di verità che descrive il comportamento del circuito logico

Diagramma temporale

Una volta analizzato il circuito logico, possiamo trovarne il diagramma temporale, disegnando le variazioni delle forme d'onda per ogni variabile, in cui gli ingressi sono definiti in maniera arbitraria o attraverso delle letture sul circuito.



I diagrammi temporali sono molti importanti, poiché sono il modo in cui rappresentiamo i segnali elettrici. Oltre ad indicare i livelli logici (1 o 0), si possono indicare i [tempi di transizione ed un eventuale presenza di rumore](#).



Esempio di diagramma temporale (comportamento ideale)

Algebra di Boole

L'algebra di Boole definisce assiomi, teoremi e proprietà legati alle funzioni logiche, permettendo la semplificazione di funzioni logiche definite.

Axiomi

Gli assiomi sono delle equivalenze che permettono di semplificare le funzioni logiche definite e di definire le [dimostrazioni dei teoremi](#), che non verranno mostrate in queste dispense.

$$0 \cdot 0 = 0$$

$$1 \cdot 1 = 1$$

$$0 \cdot 1 = 1 \cdot 0 = 0$$

$$0 + 0 = 0$$

$$1 + 1 = 1$$

$$1 + 0 = 1$$

Teoremi

Definiamo i teoremi della funzione logica AND.

$$x \cdot 0 = 0$$

$$x \cdot 1 = x$$

$$x \cdot x = x$$

Definiamo i teoremi della funzione logica OR.

$$x + 0 = x$$

$$x + 1 = 1$$

$$x + x = x$$

Definiamo i teoremi di esistenza del complemento.

$$x \cdot \bar{x} = 0$$

$$x + \bar{x} = 1$$

Proprietà

Definiamo le proprietà commutative.

$$x + y = y + x$$

$$x \cdot y = y \cdot x$$

Definiamo le proprietà associative.

$$x + (y + z) = (x + y) + z$$

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z$$

Definiamo le proprietà di assorbimento.

$$x + (x \cdot y) = x$$

$$x \cdot (x + y) = x$$

$$x + \bar{x} \cdot y = x + y$$

$$x \cdot (\bar{x} \cdot y) = x \cdot y$$

$$x \cdot y + x \cdot \bar{y} = x$$

$$(x + y) \cdot (x + \bar{y}) = x$$

Definiamo le proprietà distributive.

$$x \cdot (y + z) = x \cdot y + x \cdot z$$

$$x + (y \cdot z) = (x + y) \cdot (x + z)$$

Definiamo le proprietà di idempotenza.

$$x + x = x$$

$$x \cdot x = x$$

Leggi di De Morgan

Le leggi di De Morgan forniscono delle equivalenze tra le funzioni logiche OR e AND.

$$\overline{x \cdot y} = \bar{x} + \bar{y}$$

$$\overline{x + y} = \bar{x} \cdot \bar{y}$$



Circuiti logici

▼ Creatore originale: @LucaCaffa

[Circuiti logici combinatori](#)

[Circuiti logici sequenziali](#)

[Circuiti logici sincroni](#)

[\(*\) Circuiti logici asincroni](#)

Circuiti logici combinatori

Un circuito logico è **combinatorio** se le uscite dipendono solo dal valore logico degli ingressi in quell'istante.

La funzione, trascurando i ritardi, è come segue:

$$y(t) = f(x_1(t), x_2(t), \dots, x_n(t))$$

LOGICA COMBINATORIA



Struttura generica di un circuito combinatorio



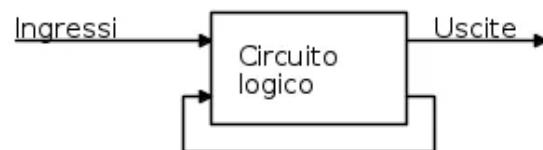
Come si vede dall'[immagine](#), gli ingressi arrivano nello "scatolotto" centrale, dove vengono elaborati, e poi arrivano all'uscita.

Circuiti logici sequenziali

Un circuito logico è **sequenziale** se, ad ogni istante t , le uscite dipendono dal valore logico degli ingressi in quell'istante e in quelli precedenti.

La funzione, trascurando i ritardi, è come segue:

LOGICA SEQUENZIALE



Struttura generica di un circuito sequenziale

$$y(t) = f(x_1(t), \dots, x_n(t), x_1(t-1), \dots, x_n(t-1), \dots)$$

La caratteristica dei circuiti sequenziali è quella di avere **memoria**.



Come si vede dall'[immagine](#), lo "scatolotto" centrale ha due ingressi a sinistra: uno rappresenta gli ingressi in quell'istante, mentre l'altro rappresenta gli ingressi precedenti, quindi l'uscita dipende sia dagli stati passati che da quelli presenti.

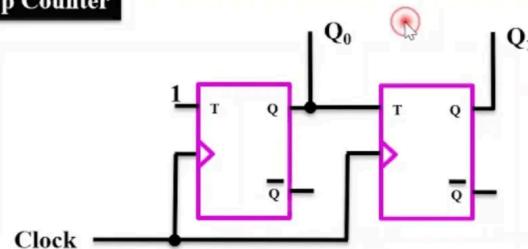
Circuiti logici sincroni

Un circuito logico è **sincrono** se funziona seguendo il "ritmo" di un segnale, chiamato **clock**.

Il circuito può attivarsi seguendo il periodo del clock in due modi:

- quando il clock diventa 1;
- quando il clock diventa 0.

Up Counter



Esempio di contatore sincrono



Il clock è un segnale periodico a onda quadra, che assume valore 1 ad intervalli regolari.

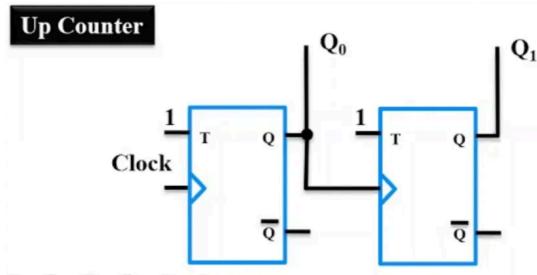
Può risultare comune far attivare il circuito sulle transizioni, ovvero quando si presenta la transizione $0 \rightarrow 1$, e viceversa. E' importante notare che, quando il circuito non viene attivato dal clock, mantiene il suo stato, ed è quindi in uno **stato di memoria**.



A partire dall'[immagine](#), è importante notare che le uscite Q_0 e Q_1 vengono generate entrambe in corrispondenza del segnale di clock, secondo una delle modalità definite.

(*) Circuiti logici asincroni

Un circuito logico è **asincrono** se non dipendono dal clock. Sono molto complessi da progettare.



Esempio di contatore asincrono



A partire dall'[immagine](#), è importante notare che l'uscita Q_1 è generata solo dopo che Q_0 arriva al Flip-Flop di destra. Si noti come l'uscita è stabile dopo i ritardi dei Flip-Flop.



Bistabili e Flip-Flop

▼ Creatore originale: @LucaCaffa

[Bistabile](#)

[Stati stabili](#)

[Metastabilità](#)

[Flip-Flop](#)

[Sincronizzazione dei FF](#)

[Latch](#)

[Clock](#)

[Inverter all'ingresso](#)

[Flip-Flop Set-Reset](#)

[Condizioni di comando](#)

[Stato di memoria e stato proibito](#)

[Implementazione con porte NOR](#)

[Implementazione con porte NAND](#)

[Flip-Flop Latch \(FF Latch\)](#)

[Flip-Flop Latch D \(D Latch\)](#)

[Comandi asincroni per il D Latch](#)

[Flip-Flop Master-Slave \(D-FF\)](#)

[Funzionamento rispetto al Clock](#)

[Comandi asincroni per il D-FF](#)

[Flip-Flop Jack-Kilby \(FF-JK\)](#)

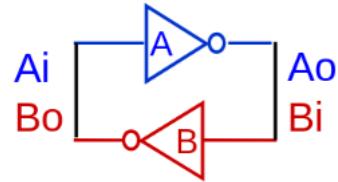
[Tavola di Verità](#)

Bistabile

Consideriamo un anello di inverter, in cui l'ingresso di una porta NOT (inverter A) è collegato all'uscita di un'altra porta NOT (inverter B).

Per ogni inverter, utilizzeremo una notazione del tipo:

- X_i : ingresso dell'inverter X (es. A_i , ingresso di inverter A);
- X_o : uscita dell'inverter X (es. A_o , uscita dell'inverter A).



Visualizzazione dell'anello di inverter

Nell'[elemento descritto](#), l'entrata di un inverter è uguale all'uscita dell'altro: $A_o = B_i$, $A_i = B_o$.

Stati stabili

In questo circuito, abbiamo 2 stati stabili:

- stato 0 (S_0): se l'uscita di A è 1, allora l'uscita di B è 0;
 - $A_o = H$;
 - $B_o = L$.
- stato 1 (S_1): se l'uscita di A è 0, allora l'uscita di B è 1.
 - $A_o = L$;

- $Bo=H$.

Dati questi due stati, si parla di circuito **bistabile**.

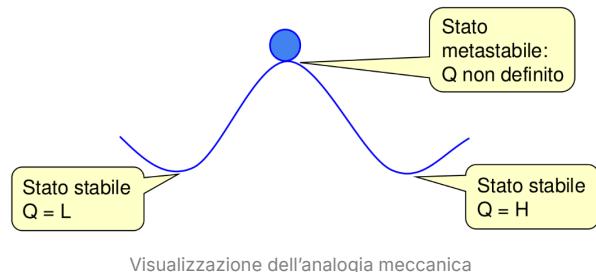
Metastabilità

Nella realtà esiste un terzo stato non stabile, chiamato **stato metastabile**.

Risulta poco intuitivo, ma possiamo immaginare di avere un circuito dove gli ingressi e le uscite non sono definite, e quindi non hanno un valore che possiamo determinare.

Per descrivere il concetto, possiamo utilizzare un'[analogia meccanica](#) (pallina su una collina):

- se la pallina è sul lato sinistro della collina, siamo nello stato stabile S_0 ;
- se la pallina è sul lato destro della collina, siamo nello stato stabile S_1 ;



- se la pallina è sulla cima della collina, siamo nello **stato metastabile S_x** , dove una minima influenza esterna modifica la posizione della pallina, portandola ad una delle due posizioni stabili descritte nei punti precedenti.

Lo stato metastabile è uno stato che non fornisce alcuna informazione importante, poiché non possiamo osservare alcun valore. Non vogliamo **MAI** trovarci in questo stato, poiché il minimo rumore può portare il circuito in uno dei due stati, portando un fattore definito dalla probabilità al circuito.

Flip-Flop

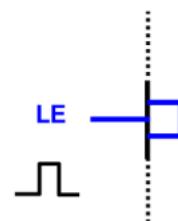
I Flip-Flop (FF) sono dei componenti bistabili, ovvero che presentano due stati stabili.

Sincronizzazione dei FF

Latch

Il comando **LE** è un segnale di tipo asincrono. Esso è identificato attraverso un quadrato nei circuiti.

- $LE = 1$: stato di trasparenza;
- $LE = 0$: stato di memoria;
- $LE = (1 \rightarrow 0)$: memorizza l'ingresso.



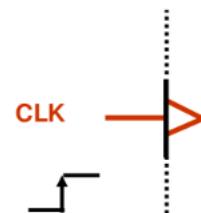
Segnale di Enable (Latch, LE)

Clock

Il comando **CLK** è un segnale di tipo sincrono. Esso è identificato attraverso un triangolo nei circuiti, inoltre vengono chiamati Edge-triggered.

Si possono definire due tipi di clock:

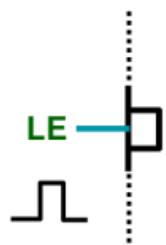
- l'uscita del componente commuta sulla transizione del Clock $1 \rightarrow 0$.
- l'uscita del componente commuta solo sulla transizione del Clock $0 \rightarrow 1$.



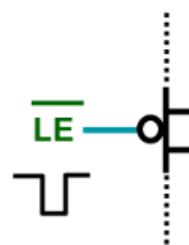
Segnale di clock (CLK)

Inverter all'ingresso

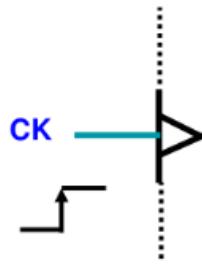
Ogni tipo di segnale, sia esso LE o CLK, può avere un'inverter all'ingresso, portando all'attivazione sul valore opposto.



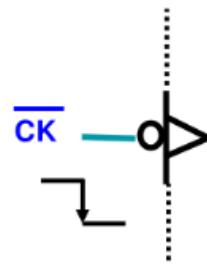
LE con valore diretto



LE con valore inverso



CLK con valore diretto



CLK con valore inverso

Flip-Flop Set-Reset

Il Flip-Flop Set-Reset (FF-SR) prende il nome dai suoi due ingressi: SET (S) e RESET (R).

Esso presenta due uscite, Q_a e Q_b , le quali sono complementari (ovvero una è il negato dell'altro). Data la complementarità delle uscite, possiamo anche chiamarle rispettivamente Q e Q^* . Questo componente ha dei [ritardi di propagazione](#), classici delle porte NOR.

Condizioni di comando

Descriviamo i due stati stabili S_0 e S_1 ([condizioni di comando](#)) del FF-SR:

- $S_0(S = 1, R = 0)$: si chiama [stato di SET](#), poiché $S = 1$;
- $S_1(S = 0, R = 1)$: si chiama [stato di RESET](#), poiché $R = 1$.

Stato di memoria e stato proibito

Oltre ai due stati stabili, esistono le restanti combinazioni dei valori Q_a e Q_b , che permettono di definire altri due stati S_2 e S_3 :

- S_2 : si chiama **stato di memoria**, poiché permette di conservare un valore.

$$Q_a(t) = Q_a(t - 1)$$

$$Q_b(t) = Q_b(t - 1)$$

Come si nota dalle equazioni che definiscono $Q_a(t)$ e $Q_b(t)$, il valore delle uscite è definito da quello dell'istante precedente, definendo la **condizione di memoria**. Grazie a questo stato, il FF-SR è uno dei componenti fondamentali per le memorie;

- S_3 : si chiama **stato proibito**, poiché il valore delle uscite non è definito.

$$Q_a(t) = ?$$

$$Q_b(t) = ?$$

Come si nota dalle equazioni che definiscono $Q_a(t)$ e $Q_b(t)$, il valore delle uscite non è definito, definendo la **condizione non permessa** (o **proibita**). Questo stato è definito dalla [metastabilità](#).

Implementazione con porte NOR

L'implementazione con porte NOR di un FF-SR si costruisce in maniera analoga all'[anello di inverter](#) visto prima, sostituendo gli inverter con delle porte NOR.

Gli stati in questa implementazione sono definiti come segue:

- condizione di comando SET S_0 ;

$$S = 1, R = 0$$

- condizione di comando RESET S_1 ;

$$S = 0, R = 1$$

- condizione di memoria S_2 ;

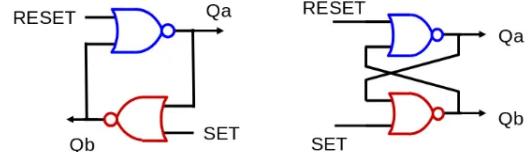
$$S = 0, R = 0$$

- condizione proibita S_3 .

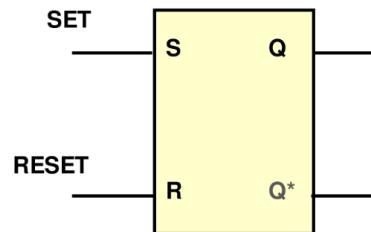
$$S = 1, R = 1$$

La tavola di verità del FF-SR implementato con le porte NOR è definita a lato, utilizzando questa legenda:

- condizioni di comando;
- condizione di memoria;
- condizione proibita.



Implementazione di FF-SR attraverso due anelli di porte NOR



Simbolo del Flip-Flop Set-Reset con porte NOR

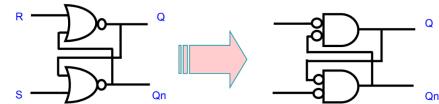
S	R	Q
0	0	Q_{-1}
0	1	0
1	0	1
1	1	?

Implementazione con porte NAND

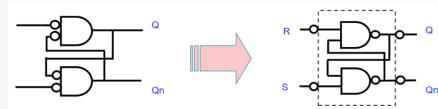
L'implementazione con porte NAND di un FF-SR si costruisce in maniera analoga all'[implementazione con porte NOR](#), ma la differenza è nella tavola di verità, in cui tutte le condizioni sono invertite.

Per capire il motivo del comportamento identico e invertito, ricordiamo la legge di De Morgan:

$$\overline{x+y} = \overline{x} \cdot \overline{y}$$



Tramite De Morgan sostituiamo le NOR con delle porte AND a cui sono stati negati gli ingressi.



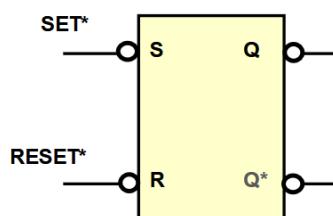
Traslando opportunamente gli ingressi delle porte AND, si ottiene la configurazione di un FF SR

Si noti come, in questo caso, quella che prima era l'uscita Q diventa \overline{Q} , e viceversa:

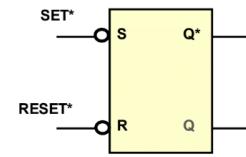
$$Q \rightarrow \overline{Q}$$

$$\overline{Q} \rightarrow \overline{\overline{Q}} = Q$$

Si ha, quindi, una corrispondenza tra la porta NOR e la porta NAND, e ciò permette di realizzare l'anello di inverter con le porte NAND al posto degli inverter, come definito nella [visualizzazione](#).



Simbolo del Flip-Flop Set-Reset con porte NAND



Simbolo del FF S-R con porte NAND, equivalente alla figura di sinistra.

Gli stati in questa implementazione sono definiti come segue:

- condizione di comando SET S_0 ;

$$S = 1, R = 0 \Rightarrow S^* = 0, R^* = 1$$

- condizione di comando RESET S_1 ;

$$S = 0, R = 1 \Rightarrow S^* = 1, R^* = 0$$

- condizione di memoria S_2 ;

$$S = 0, R = 0 \Rightarrow S^* = 1, R^* = 1$$

- condizione proibita S_3 .

$$S = 1, R = 1 \Rightarrow S^* = 0, R^* = 0$$

La tavola di verità del FF-SR implementato con le porte NAND è definita a lato, utilizzando questa leggenda:

- condizioni di comando;

S	R	S^*	R^*
1	1	0	0

- condizione di memoria;
- condizione proibita.

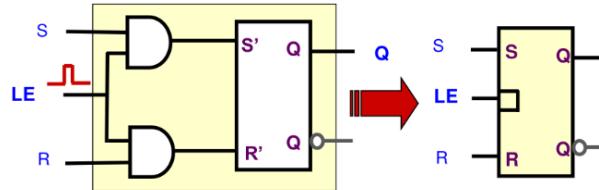
S	R	S^*	R^*
1	0	0	1
0	1	1	0
0	0	1	1

Si noti come le uscite delle condizioni di comando sono invertite rispetto all'[implementazione con le porte NOR](#). Per esempio, se prima con $S = 1, R = 0$ si avevano $Q = 1, Q^* = 0$, in questo caso l'output sarà opposto: $Q = 0, Q^* = 1$.

Flip-Flop Latch (FF Latch)

Partendo da un FF-SR e aggiungendo agli ingressi due porte AND, ricaviamo un componente chiamato [FF Latch](#).

La caratteristica fondamentale di questo componente è quella di avere un segnale **LE** che attiva o disattiva gli ingressi, come segue:



Implementazione del Flip-Flop Latch a partire da un FF-SR

- $LE = 1$ (stato trasparente): l'uscita può cambiare stato in base agli ingressi S e R ;
- $LE = 0$ (stato di memoria/latched mode): l'uscita non può cambiare stato.

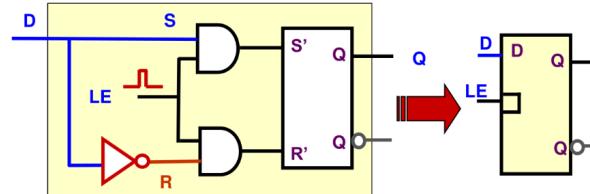
Ciò significa che, anche se S e R assumono dei valori, il segnale entrerà nel FF solo se $LE = 1$, come conseguenza dell'aggiunta delle porte AND.



Il comando **LE** è identificato attraverso un quadrato, che identifica un segnale di tipo asincrono.

Flip-Flop Latch D (D Latch)

Per migliorare il [FF Latch](#), possiamo fare in modo che i **due ingressi diventino un unico segnale**, e che quindi S e R siano sostituiti da un unico segnale. Questo possiamo farlo perché S e R devono sempre essere uno il negato dell'altro.



Implementazione completa di un FF Latch D a partire da un FF-SR

L'unico caso in cui devono essere uguali è il **comando di memoria**, ma poiché qui abbiamo il segnale **LE** che ci permette di creare lo stato di memoria, possiamo rendere S e R sempre opposti.

Per permettere il funzionamento con un singolo ingresso, bisogna fare in modo che il ramo entrante in S abbia segnale diretto, mentre il ramo entrante in R abbia segnale negato.

Dopo questa aggiunta, si ottiene un [Latch D](#), dove D è l'ingresso unico. Si noti come questa versione del Latch D è **asincrona**.



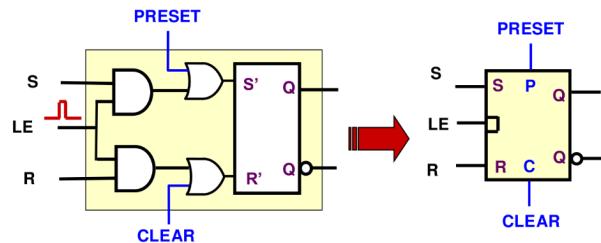
Il Latch D è di fondamentale importanza, poiché risolve il problema dello stato metastabile del FF-SR. Avendo un unico ingresso, in un caso negato e nell'altro diretto, è impossibile ricadere nella condizione di metastabilità.

Comandi asincroni per il D Latch

Possiamo aggiungere al Latch D appena visto due comandi, che non dipendono dal segnale LE, e per questo sono definiti **comandi asincroni del Latch D**.

I comandi sono PRESET (P) e CLEAR (C):

- se CLEAR = 1, si imposta $Q = 0$, ovvero l'uscita Q va immediatamente a 0;
- se PRESET = 1, si imposta $Q = 1$, ovvero l'uscita Q va immediatamente a 1.



Costruzione Latch D con i comandi asincroni Clear (C) e Preset (P)

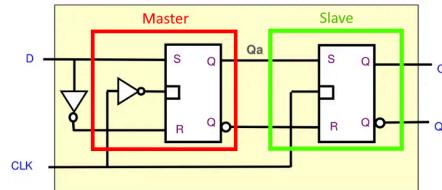


Se PRESET e CLEAR si attivano contemporaneamente, il FF si porta nella condizione proibita.

Flip-Flop Master-Slave (D-FF)

Il FF Master-Slave (D-FF) è un componente formato da una cascata di FF Latch con abilitazione complementare dei dati attraverso il segnale di clock.

Come si può notare dall'[implementazione mostrata](#), si ha un solo segnale di ingresso D. Le uscite del primo FF Latch sono le entrate del secondo FF Latch.

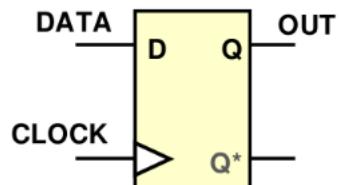


Implementazione completa di un FF Master-Slave a partire da due FF Latch

Funzionamento rispetto al Clock

In questo componente, il clock abilita uno dei due FF, attraverso il collegamento a Latch, durante la fase di transizione 0 → 1 come descritto in seguito:

- CLK = 0: abilita il primo FF Latch (Master) a registrare l'ingresso D, mentre il secondo (Slave) rimane nello stato di memoria. In questo caso, si parla di **Master trasparente** e **Slave in memoria**;
- CLK = 1: abilita il secondo FF Latch (Slave) ad ottenere il valore in memoria del Master, portandolo come valore in uscita al componente, mentre il secondo (Master) rimane nello stato di memoria. In questo caso, si parla di **Master in memoria** e **Slave trasparente**.



Simbolo del Flip-Flop Master-Slave



Il comando CLK è identificato attraverso un triangolo, che identifica un segnale di tipo sincrono.

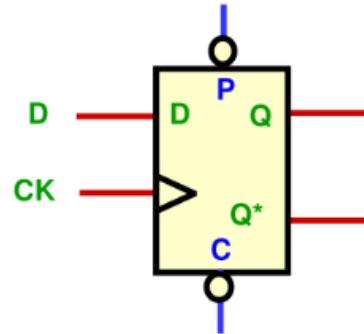


E' il tipo di porta sequenziale più usato.

Comandi asincroni per il D-FF

Come per il D Latch, anche il D-FF può avere i due comandi asincroni PRESET e CLEAR.

- se CLEAR = 1, si imposta $Q = 0$, ovvero l'uscita Q va immediatamente a 0;
- se PRESET = 1, si imposta $Q = 1$, ovvero l'uscita Q va immediatamente a 1.



Simbolo del Flip-Flop Master-Slave con comandi asincroni Clear (C) e Preset (P)



Se PRESET e CLEAR si attivano contemporaneamente, il FF si porta nella condizione proibita.

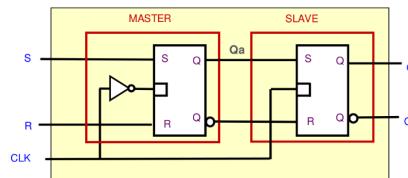


In questo esempio, i comandi PRESET e CLEAR sono attivabili se il loro valore è 1. Si può anche realizzare con i comandi attivabili se il valore è 0.

Flip-Flop Jack-Kilby (FF-JK)

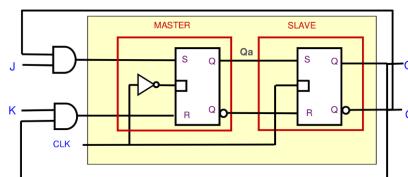
Costruiamo il FF-JK a partire dal D-FF seguendo i passi descritti:

1. a partire dal D-FF, si rimuove lo stato comune D, in modo da avere nuovamente due ingressi distinti sul componente;



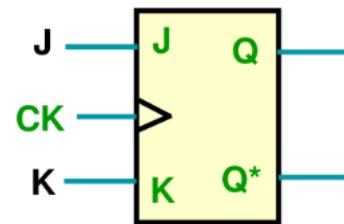
Primo passo della costruzione del FF-JK

2. creiamo una retroazione con 2 AND, definendo due nuovi segnali chiamati J e K.



Secondo passo della costruzione del FF-JK

Attraverso questa costruzione otteniamo il Flip-Flop Jack-Kilby (FF-JK), ovvero un [componente sequenziale](#), il cui funzionamento è molto simile a quello del [FF-SR](#), ma senza il problema legato allo [stato metastabile](#).



Simbolo del Flip-Flop Jack-Kilby (FF-JK)

Tavola di Verità

La tavola di verità del FF-JK è definita a lato, utilizzando questa legenda:

- condizione di memoria;
- condizioni di comando;
- condizione per complementare l'uscita.

<i>J</i>	<i>K</i>	<i>S</i>	<i>R</i>
0	0	0	0
0	1	0	Q_{-1}
1	0	Q_{-1}^*	0
1	1	Q_{-1}^*	Q_{-1}

Dalla tavola di verità riusciamo a capire bene il comportamento di questo componente:

- le [condizioni di comando](#) evidenziano che le uscite dipendono dal valore dei vecchi ingressi;
- la [condizione di memoria](#) permette di memorizzare gli stati precedenti di Q e Q^* ;
- la vecchia condizione di metastabilità diventa la [condizione per complementare l'uscita](#), ovvero un comando per scambiare i valori nelle uscite Q e Q^* .

Si noti come gli ingressi del FF-JK siano strettamente legati alle uscite, quindi è possibile definire le equazioni dello stato futuro Q .

$$Q = JQ_{-1}^* + Q_{-1}\bar{K}$$



Tempi di clock, di setup, di hold e frequenza massima

▼ Creatore originale: @Gianbattista Busonera 😊

- @Gianbattista Busonera(12/04/2025): Trattazione tramite video e testo di un argomento **filosofico** del corso, utile in numerosi esercizi.
- @Giacomo Dandolo (13/04/2025): aggiunta di piccole descrizioni, miglioramento dell'evidenziatura del testo, in modo da catturare meglio le parti importanti.

Ritardo di propagazione del clock

Ritardi di propagazione su transizione L→H e H→L

Finestra di campionamento

Tempo di setup

Calcolo del periodo di clock minimo (ritardi massimi)

Calcolo della frequenza massima

Tempo di hold

Calcolo del tempo di hold massimo (ritardi minimi)

Esempio - Soddisfacimento delle condizioni di setup e hold



Il materiale di questa pagina
è stato preso:

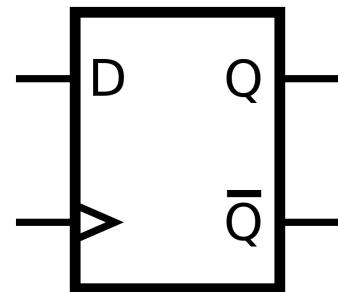
- dalle lezioni di ELAP
2018: [LINK](#)
- da video di indiani su
Youtube, di cui allego il
principale: [LINK](#)

<https://youtu.be/rTB2H5gs9vA>

Spiegazione di questo argomento in formato video (NB: il materiale mostrato in video potrebbe non essere 1:1 con ciò che è mostrato in questa pagina, ma i concetti sono equivalenti)

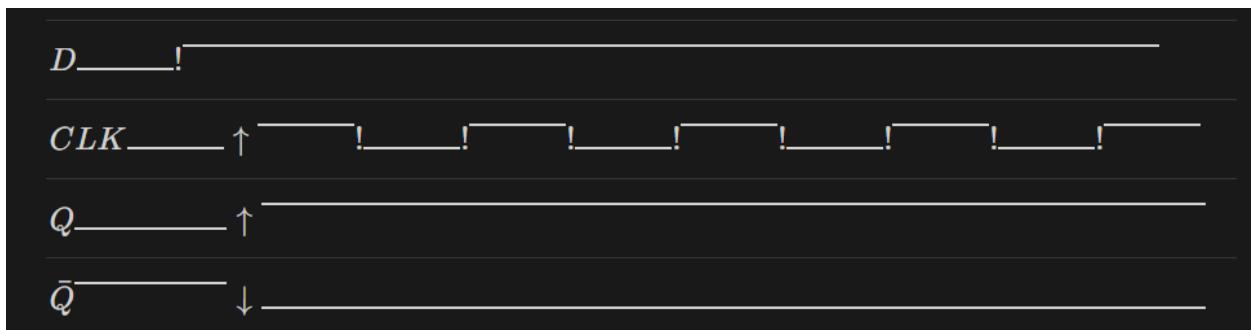
Tutte le porte logiche, quindi anche quelle che formano i flip-flop, introducono dei ritardi.

Prendiamo in considerazione un Flip Flop di tipo D che commuta sul fronte di salita del clock.



Visualizzazione del
componente Flip-Flop di
tipo D

In un mondo ideale, senza ritardi di alcun tipo, otterremmo questo risultato:

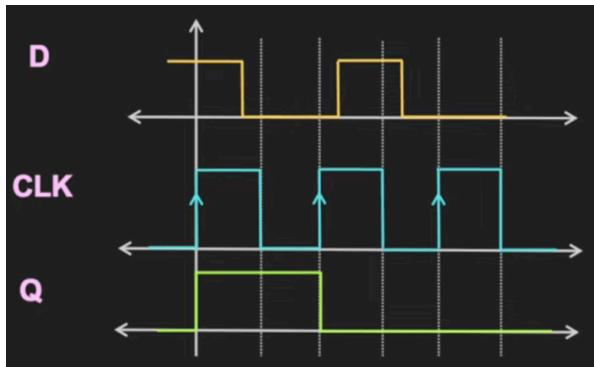


Questo perché il Flip Flop commuta sul fronte di salita del clock e, di conseguenza, mantiene il valore alto (1 logico) per tutto il tempo in cui D resta alto, senza alcun ritardo.

Fino ad ora stiamo assumendo che, al fronte di salita del clock (CLK), qualsiasi sia il valore di D (ingresso), quest'ultimo verrà campionato istantaneamente (in Q).

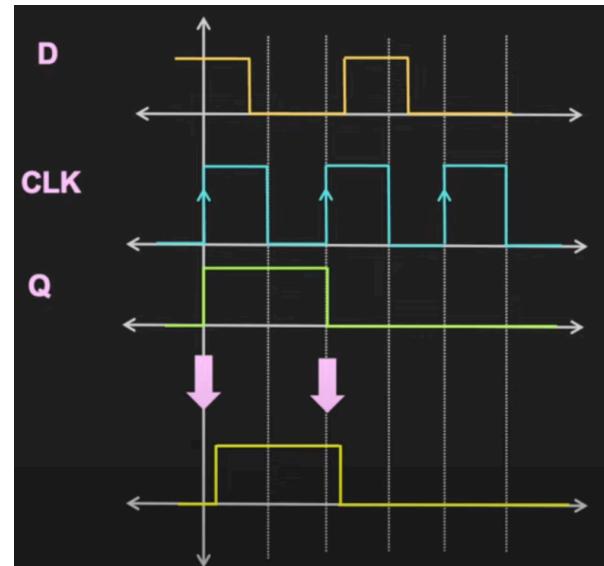
Ritardo di propagazione del clock

In un modello ideale, l'uscita Q del flip flop assume istantaneamente il valore di D, come visto precedentemente.



Uscita Q commuta istantaneamente (modello ideale)

In un modello reale, l'uscita Q commuta con un certo ritardo, detto ritardo di propagazione t_{CK-Q} .



Uscita Q che commuta con un certo ritardo (reale)

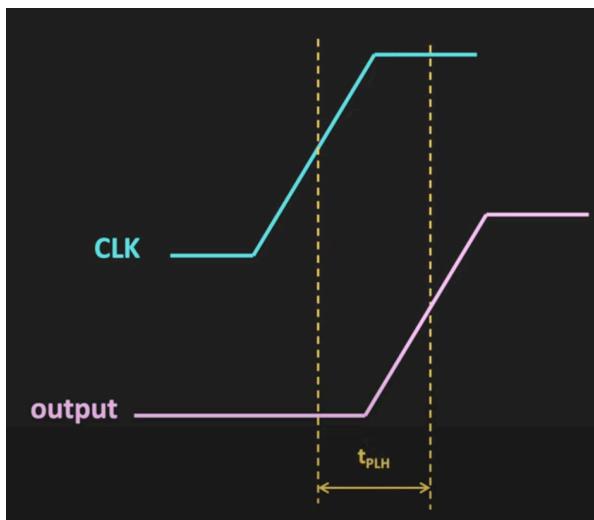
Il ritardo di propagazione t_{CK-Q} è dovuto al fatto che il clock non commuta istantaneamente da uno stato basso a uno alto, ma quest'ultimo deve passare una certa soglia V_T affinché si possa effettivamente considerare "commutato".

Ritardi di propagazione su transizione L→H e H→L

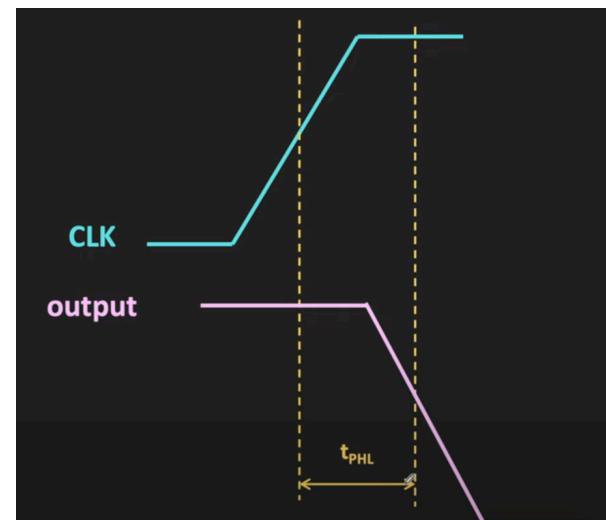
I tempi descritti in questa sezione sono quelli di propagazione sulla transizione L→H ($t_{CK,L \rightarrow H}$) e sulla transizione H→L ($t_{CK,H \rightarrow L}$), dove L indica lo stato basso

(0), mentre H indica lo stato alto (1).

Come visibile in foto, l'output inizia a cambiare stato solo quando **il clock raggiunge effettivamente lo stato alto**. Inoltre, l'uscita non commuta istantaneamente al valore d'ingresso, ma è necessario un certo **tempo transitorio**, cioè un breve tempo necessario affinché anche l'output possa effettivamente commutare (in quanto anche lui ha un certo ritardo di propagazione da uno stato alto a basso o viceversa).



Visualizzazione della commutazione di output quando il valore di clock arriva allo stato alto (ideale)



Visualizzazione della commutazione di output quando il valore di clock arriva allo stato alto (reale)

Finestra di campionamento

In un modello reale, l'input dovrebbe rimanere stabile per una certa quantità di tempo, detto **tempo di acquisizione** o **finestra di campionamento**.

In particolare, tale finestra è suddivisa in:

- **tempo di setup t_{su}** ;
- **tempo di hold t_{hold}** .



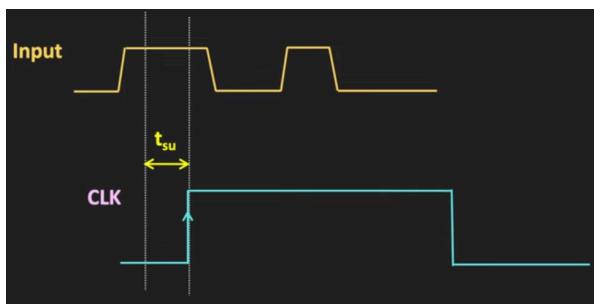
La finestra di campionamento e, quindi, i tempi di setup t_{su} e di hold t_{hold} di cui è composta, riguarda solamente la quantità di tempo per cui l'**input** deve rimanere stabile.

Tempo di setup

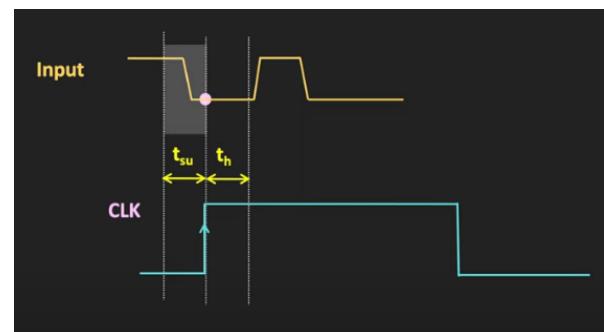
Il tempo di setup t_{su} è l'**intervallo di tempo per cui l'ingresso D (Input) deve rimanere stabile** (non può commutare) prima del fronte di salita del clock.

Sono presenti due casi:

1. nel primo diagramma temporale, si può vedere che l'input è **stabile al valore alto** per il tempo di setup. Di conseguenza, non sono presenti violazioni e il dato può essere campionato;
2. nel secondo diagramma temporale, si può vedere che l'input **non è stabile** per il tempo di setup. Di conseguenza, sono presenti violazioni e il dato risulta inconsistente con ciò che ci si aspetta.



Soddisfacimento del tempo di setup, in quanto l'input NON commuta durante il tempo di setup



Violazione del tempo di setup, in quanto l'input commuta durante il tempo di setup



Si noti che il tempo di setup dipende unicamente dalle **caratteristiche intrinseche del flip flop** ed è un dato che ci verrà sempre fornito.

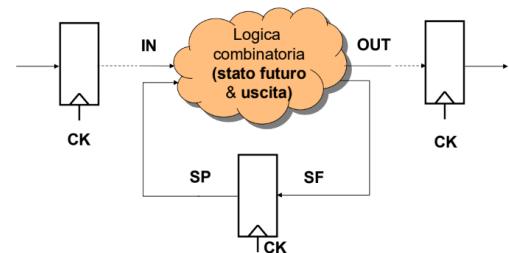
Calcolo del periodo di clock minimo (ritardi massimi)

Consideriamo il seguente modello di circuito, in cui il flip flop a sinistra (d'ora in poi chiamato **FF1**) deve trasferire il suo valore (**IN**, equivalente a Q_1) nella logica combinatoria, la quale farà le dovute trasformazioni su tale dato.

Una volta processato, il dato (**OUT**) è pronto per essere immagazzinato nel flip flop a destra (**FF2**) come input.

Valutiamo un diagramma temporale di esempio che ci permette di vedere le variazioni su **IN** e **OUT** tra FF1 e FF2:

1. concentriamoci sull'input: esso non viene propagato istantaneamente sulla logica combinatoria, ma deve aspettare un tempo t_{CK-Q} .
2. per un certo periodo di tempo $t_{LC,max}$ (definito come il tempo massimo di elaborazione della logica combinatoria), il dato resta variabile prima di stabilizzarsi a un certo valore.

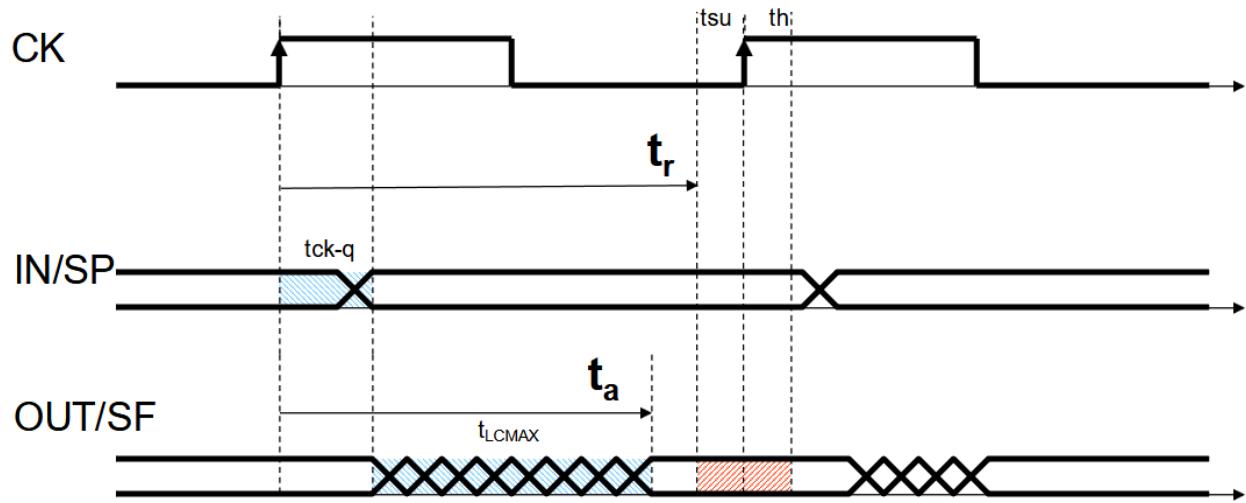


Circuito con 3 flip flop di tipo D e una logica combinatoria



Si prende il ritardo massimo della logica combinatoria in quanto vogliamo considerare il **caso peggiore** per il quale dobbiamo garantire il corretto funzionamento del circuito e, quindi, determinare un **periodo di clock minimo**.

E' importante che il dato si stabilizzi prima del **tempo di setup del periodo di clock successivo** (**OUT** è l'input D del FF2, e deve soddisfare i vincoli della finestra di campionamento).



Consideriamo il primo Flip Flop a sinistra come già "stabile"; il problema è l'input del flip flop a destra, il quale risente dei ritardi del primo flip flop (t_{CK-Q}) e dei ritardi della logica combinatoria ($t_{LC,MAX}$).

Il dato che parte dall'uscita del FF1 si propaga in ingresso nella logica combinatoria (come in via indiretta in ingresso del FF2) dopo un certo $t_{arrivo} = t_a$ pari a:

$$t_a = t_{CK-Q} + t_{LC,max}$$

Però, è necessario che tale dato sia memorizzato nel FF2 in un certo $t_{richiesta} = t_r$ pari a:

$$t_r = T_{CK} - t_{su}$$



Il dato che parte dall'uscita del FF1, per essere memorizzato nel FF2, deve restare stabile per un tempo di setup prima del successivo colpo di clock. In tal senso, si parla di **tempo minimo richiesto per evitare violazioni**.

Per quanto detto in precedenza, consideriamo la seguente relazione:

$$t_a \leq t_r$$

Riportando i valori di t_a e t_r definiti in precedenza ed esplicitando il valore di T_{CK} , si ottiene:

$$t_{CK-Q} + t_{LC,max} \leq T_{CK} - t_{su} \Rightarrow T_{CK} \geq t_{CK-Q} + t_{LC,max} + t_{su}$$

Si capisce, dunque, che il **periodo di clock minimo** affinché il circuito funzioni senza violazioni di setup è pari a:

$$T_{CK,min} = t_{CK-Q} + t_{LC,max} + t_{su}$$

Calcolo della frequenza massima

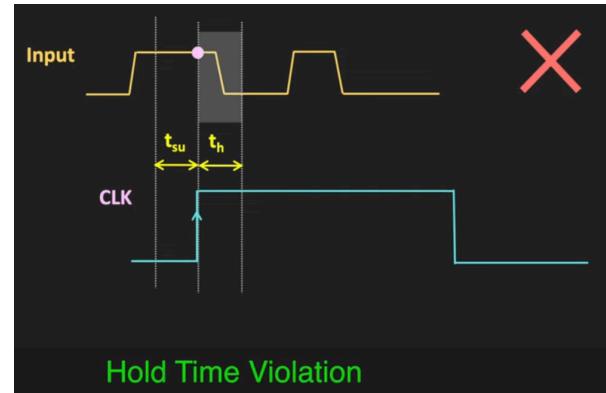
La frequenza massima per la quale un circuito può funzionare senza problemi è facilmente calcolabile come segue (ricordando che la frequenza è l'inverso del periodo):

$$F_{max} = \frac{1}{T_{CK,min}} = \frac{1}{t_{CK-Q} + t_{LC,max} + t_{su}}$$

Tempo di hold

Il tempo di hold t_{hold} è l'**intervallo di tempo per cui l'ingresso D (input) deve rimanere stabile dopo il fronte di salita del clock**.

Nel [diagramma temporale di esempio](#), si può notare che l'input non è stabile per il tempo di hold, in quanto passa da alto a basso. Di conseguenza, è presente una violazione di hold.



Violazione del tempo di hold, in quanto l'input commuta durante il tempo di hold



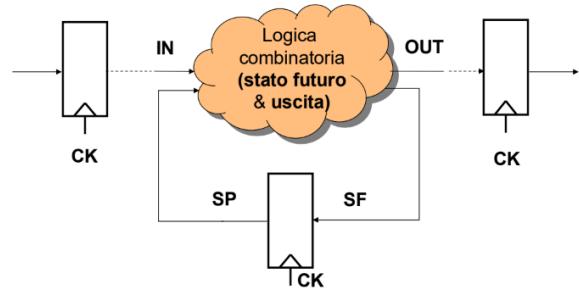
Si noti che il tempo di hold è **più subdolo** del tempo di setup, in quanto non dipende unicamente dal flip flop, ma **anche dal contesto esterno** (ritardi di propagazione, logica combinatoria, e via dicendo)

Un'analogia interessante per questo concetto risulta essere nella macchina fotografica:

- per catturare una foto con tanta luce, il tempo è breve (**contesto favorevole**);
- per catturare una foto con poca luce, il tempo risulta lungo (**contesto sfavorevole**).

Calcolo del tempo di hold massimo (ritardi minimi)

Consideriamo il seguente modello di circuito, descritto già quando si è parlato del calcolo del periodo di clock minimo nella sezione del tempo di setup.



Circuito con 3 flip flop di tipo D e una logica combinatoria

Valutiamo un diagramma temporale di esempio che ci permette di vedere le variazioni su IN e OUT tra FF1 e FF2:

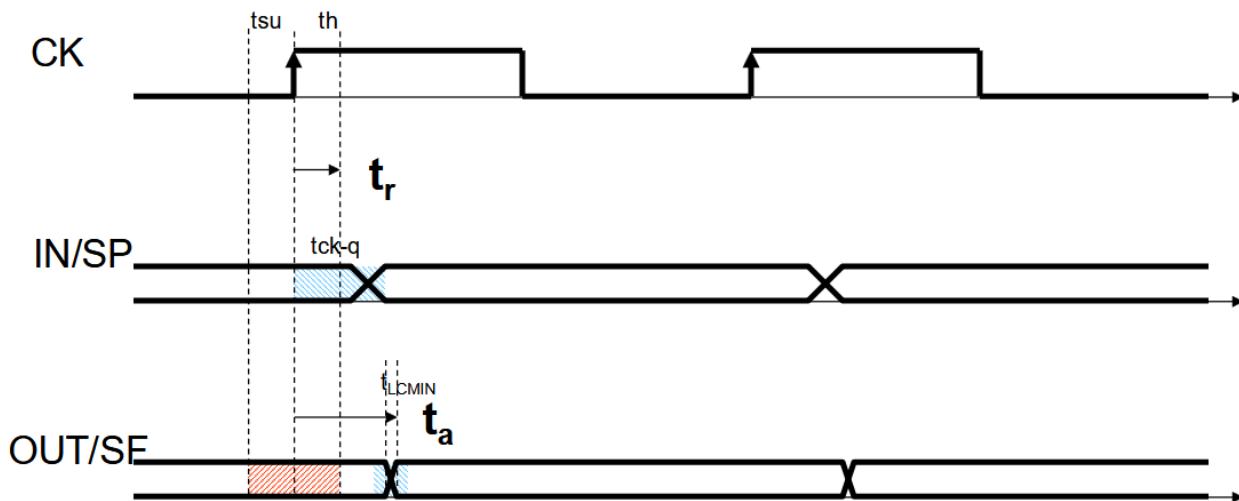
1. concentriamoci sull'input: esso **non viene propagato istantaneamente sulla logica combinatoria** all'arrivo del fronte di salita del clock, ma deve aspettare un tempo t_{CK-Q} .
2. per un certo periodo di tempo $t_{LC,MIN}$ (definito come il tempo minimo di elaborazione della logica combinatoria), **il dato resta variabile** prima di stabilizzarsi a un certo valore.



Si prende il **ritardo minimo della logica combinatoria** in quanto vogliamo considerare il **caso peggiore** per il quale dobbiamo garantire il corretto funzionamento del circuito e, quindi, determinare un **tempo di hold massimo**.

E' importante che il **FF2 mantenga il dato precedente** (area evidenziata in arancione) **per un certo tempo di hold** prima che assuma un nuovo valore. Il tempo di hold, infatti, ha come vincolo il **periodo di clock attuale** (e non il successivo, come nel caso del tempo di setup).

Se il nuovo dato (**IN**) arriva troppo in fretta (dati i tempi t_{CK-Q} e $t_{LC,min}$ troppo brevi), rischiamo che **OUT** commuti durante il suo tempo di hold, commettendo una violazione di hold.



Consideriamo il primo Flip Flop a sinistra (FF1) come già "stabile"; il problema sta nel valore "precedente" del flip flop a destra, il quale deve rimanere stabile per almeno il tempo di hold del periodo di clock attuale

Il dato che parte dall'uscita del FF1 si propaga in ingresso della logica combinatoria (e quindi in via indiretta in ingresso del FF2) dopo un certo $t_{arrivo} = t_a$ pari a:

$$t_a = t_{CK-Q} + t_{LC,min}$$

Però è necessario che tale dato arrivi nel FF2 solo dopo che sia passato il tempo di hold per il suo precedente valore e, quindi, dopo un certo $t_{\text{richiesta}} = t_r$ pari a:

$$t_r = t_{\text{hold}} = t_h$$



Il dato che parte dall'uscita del FF1, deve evitare di arrivare troppo in fretta all'ingresso del FF2, altrimenti violeremmo il tempo di hold del FF2 per il precedente valore (riferito allo stesso periodo di clock). In tal senso, si parla di **tempo massimo di hold richiesto per evitare violazioni**.

Per quanto detto in precedenza, consideriamo la seguente relazione:

$$t_a \geq t_r$$

Riportando i valori di t_a e t_r definiti in precedenza ed esplicitando il valore di t_{hold} , si ottiene:

$$t_{\text{CK-Q}} + t_{\text{LC,min}} \geq t_{\text{hold}} \Rightarrow t_{\text{hold}} \leq t_{\text{CK-Q}} + t_{\text{LC,min}}$$

Si capisce, dunque, che il **tempo di hold massimo** affinché il circuito **funzioni senza violazioni di hold** è pari a:

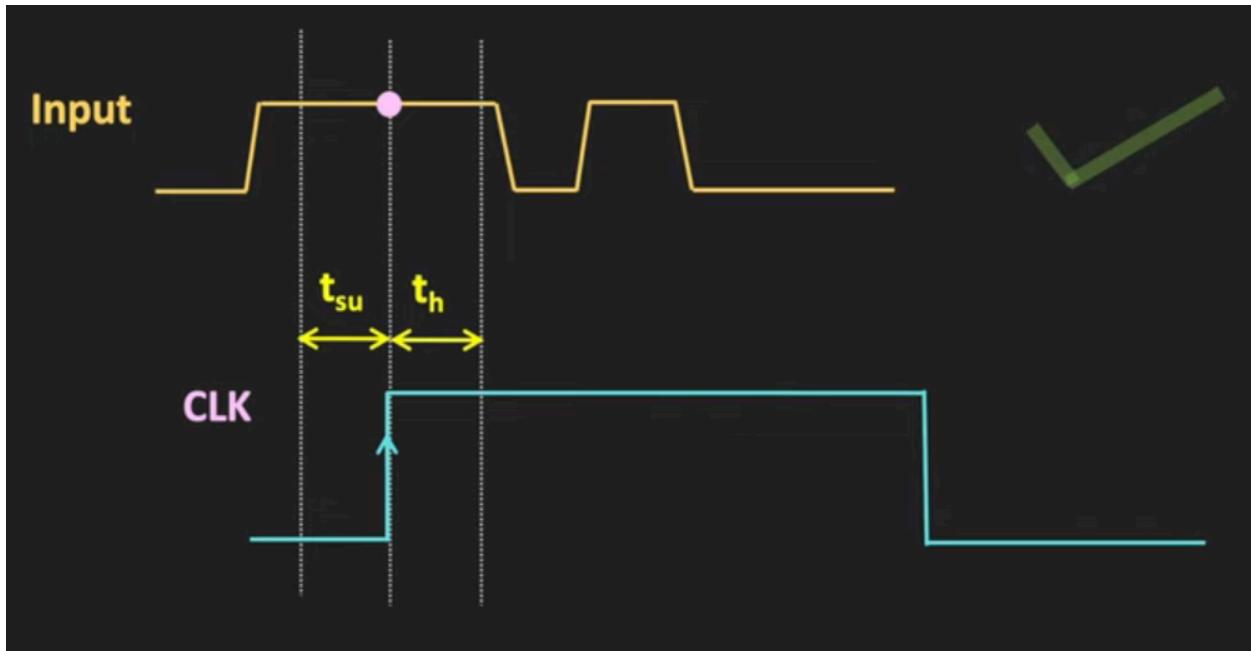
$$t_{\text{hold,max}} = t_{\text{CK-Q}} + t_{\text{LC,min}}$$



In pratica, il problema è che un segnale può arrivare nel FF2 così rapidamente da modificare un suo ingresso che, invece, doveva rimanere stabile per un tempo di hold.

Esempio - Soddisfacimento delle condizioni di setup e hold

L'input descritto nel diagramma temporale soddisfa entrambe le condizioni di setup e di hold.



Esempio di soddisfacimento di tempi di setup e di hold



Un'analogia interessante per descrivere la finestra temporale di acquisizione risulta essere nella macchina fotografica: immaginiamo di utilizzare una macchina fotografica e di voler scattare una foto al Prof. "C" (ovvero il nostro **input D** del FF).

Il **fronte di salita del clock** è dato dalla pressione del bottone per scattare la foto, e vogliamo che il soggetto della nostra foto (input D):

- resti fermo per un **dato periodo di tempo** prima dello scatto, così che non faccia smorfie poco prima che sia scattata e si possa scegliere l'istante giusto, definendo il **tempo di setup**;
- resti fermo un **istante** dopo lo scatto, cosicché la macchina fotografica abbia il tempo di catturare correttamente l'immagine, definendo il **tempo di hold**.



Registri

▼ Creatore originale: @LucaCaffa

[Segnali Seriali](#)

[Vantaggi e svantaggi](#)

[Segnali Paralleli](#)

[Vantaggi e svantaggi](#)

[Registri](#)

[Registro parallelo \(PIPO\)](#)

[Shift-register \(registro a scalamento/scorrimento\)](#)

[SISO](#)

[SIPO](#)

[PISO](#)

[Shift-register completo](#)

Segnali Seriali

I segnali seriali utilizzano un solo canale per trasferire un valore di più bit, presentando un bit per volta in uscita. Il trasferimento dei bit viene temporizzato dal clock, quindi per trasferire N bit è necessario far passare N cicli di clock.

Il tempo totale per vedere il valore intero sarà quindi $N \cdot T_{\text{ck}}$, dove T_{ck} è un periodo di clock.

Vantaggi e svantaggi

I vantaggi sono il basso costo e il minor consumo, ma lo svantaggio è quello di avere una bassa velocità. Vengono spesso utilizzati sulle lunghe distanze.

Segnali Paralleli

I segnali paralleli utilizzano N canali per trasferire un valore di N bit. Il trasferimento avviene in un solo periodo di clock, poiché i bit sono presenti contemporaneamente su più canali.

Il tempo totale per vedere il valore intero sarà quindi T_{ck} , dove T_{ck} è un periodo di clock.

Vantaggi e svantaggi

Il vantaggio è la velocità, ma lo svantaggio è quello di avere un costo elevato e un consumo elevato. Vengono spesso utilizzati sulle brevi distanze.

Registri

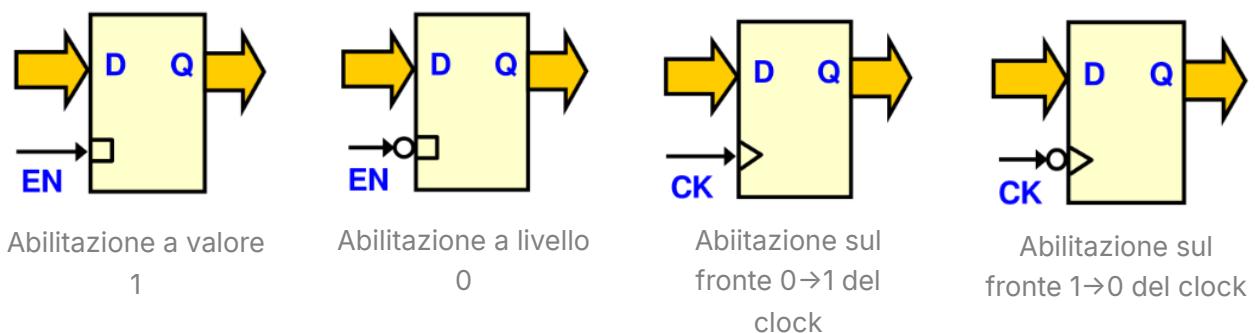
Un registro è un componente formato da N Flip-Flop o Latch. Esattamente come per i FF, i registri possono essere edge-triggered se attivi sul fronte del clock, oppure Latch quando l'abilitazione è attiva.

Possono presentare comandi comuni, come Reset e Clear.

Registro parallelo (PIPO)

I registri PIPO sono definiti come registri con Parallel Input e Parallel Output.

Si distinguono in base al tipo di segnale di comando esattamente come i FF:



Shift-register (registro a scalamento/scorrimento)

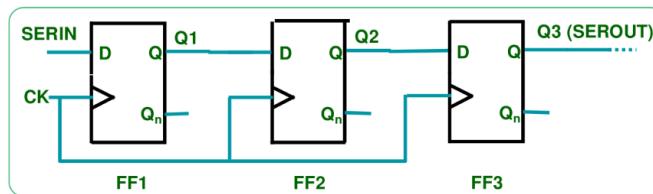
SISO

Lo shift-register **SISO** è formato da una cascata di FF-D, e sono definiti come registri con Serial Input e Serial Output.

Dal nome si intuisce il funzionamento, infatti il valore viene memorizzato uno per volta nei FF-D e, mentre si memorizzano i bit, quelli già memorizzati si spostano di FF, e quindi 'scalano'.

Ipotizziamo di voler memorizzare un valore su 3 bit (ad esempio **100**):

1. trasferiamo il bit più a sinistra (**1**), che viene posto sul primo FF;
2. dopo il successivo ciclo di clock trasferiamo **0** sul primo FF e **1** scala al secondo FF;
3. dopo il successivo ciclo di clock trasferiamo **0** sul primo FF, **0** scala sul secondo FF e **1** scala sul terzo FF.

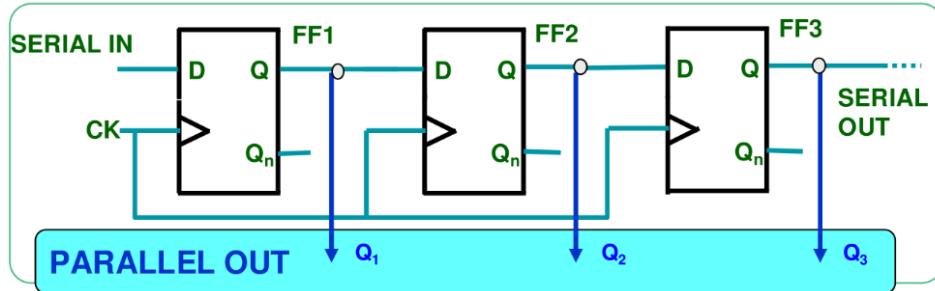


Shift-Register SISO formato da 3 FF-D

SIPO

Lo shift-register **SIPO** è formato da una cascata di FF-D, e sono definiti come registri con Serial Input e Parallel Output. Viene chiamato SIPO perché può convertire un dato seriale in un dato parallelo.

L'ingresso è seriale, quindi l'informazione sarà totalmente dentro i FF-D solo dopo N periodi di clock. La differenza con lo shift-register SISO è che qui il dato è disponibile in forma parallela.

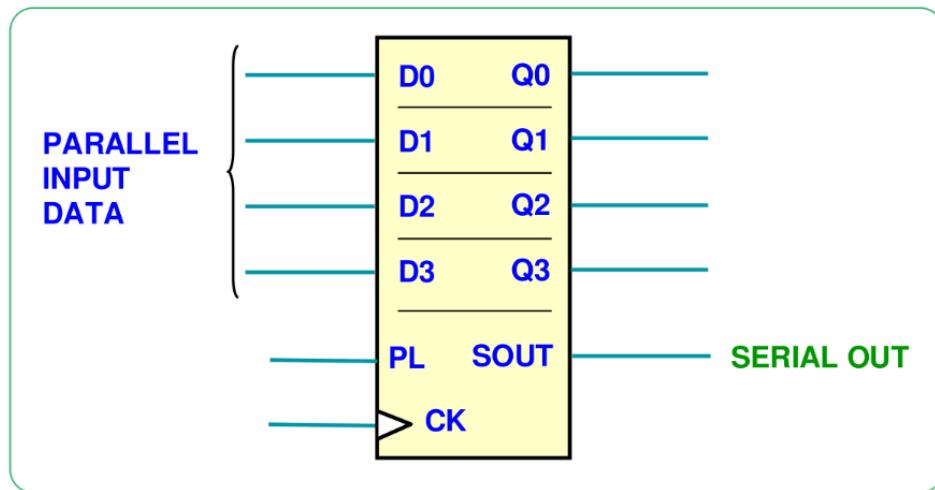


Shift-Register SIPO formato da 3 FF-D

PISO

Lo shift-register **PISO** è formato da FF-D in parallelo, e sono definiti come registri con **Parallel Input** e **Serial Output**. Viene chiamato PISO perché può convertire un dato parallelo in un dato seriale.

La caratteristica di questo registro è la presenza di un segnale di comando chiamato **Parallel Load** (PL). Quando PL è attivo, si memorizza il valore di ingresso su tutti i FF-D in contemporanea, ma l'uscita sarà disponibile in maniera seriale, un bit per volta.

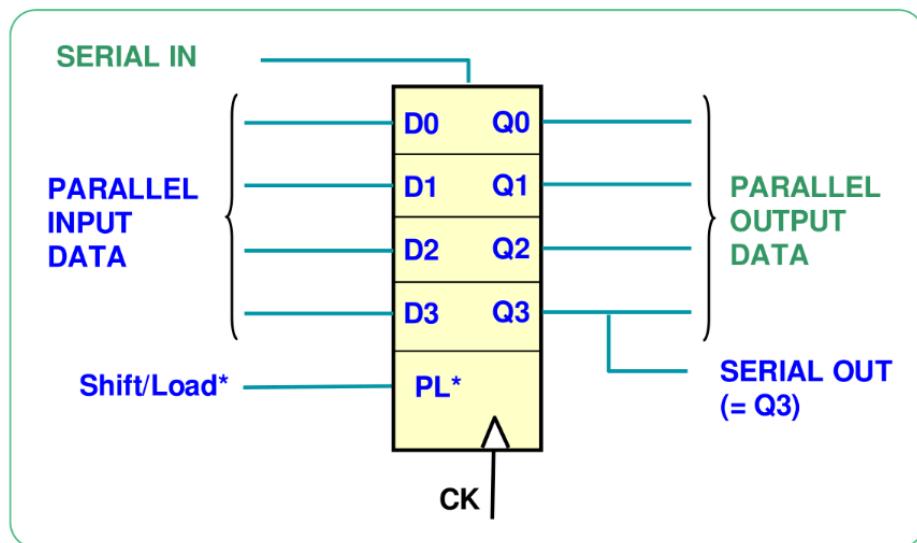


Shift-Register PISO

Shift-register completo

Lo shift-register completo è un registro formato da un insieme di FF-D, ognuno dei quali ha in ingresso un multiplexer che seleziona uno tra due dati in ingresso: parallelo o seriale.

Questa selezione avviene tramite un segnale di comando chiamato Shift/Load*. Si chiama registro completo, perché da solo può essere utilizzato come registro PIP0, SISO, PISO o SIPO.



Shift-Register completo



Contatori e divisori

▼ Creatore originale: @LucaCaffa

Contatore

Contatore asincrono

Contatore sincrono

Differenze dei ritardi nei contatori sincroni/asincroni

FF-JK come contatore / divisore

Divisore

Divisore modulo 2/4

Contatore

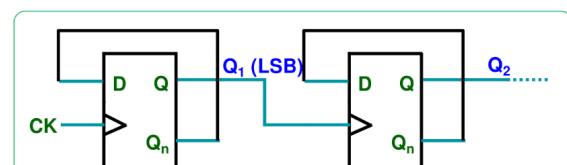
Un contatore è un circuito logico (formato da diversi componenti, tra cui FF) che riesce a generare sulle uscite una sequenza di conteggio binario, che incrementa ad ogni ciclo di clock. Esistono contatori UP (ascendenti) e DOWN (descendenti).

Contatore asincrono

I clock dei FF sono collegati a catena (ripple), quindi l'uscita di un FF è collegata al clock del FF successivo.

I ritardi di commutazione si sommano, perché bisogna aspettare di avere la prima uscita per avere il segnale di clock del successivo FF.

Il ritardo totale è $T_{pdM} = M \cdot T_{pd}$, dove M è il numero di FF e T_{pd} è il ritardo di un singolo FF.



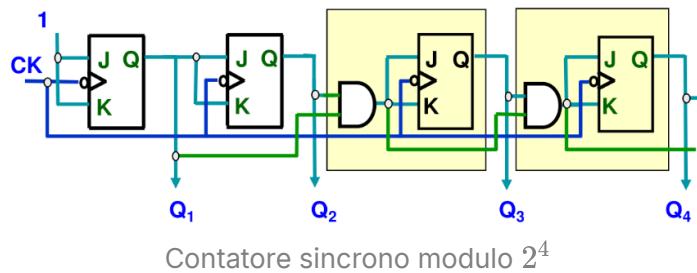
Contatore asincrono

Contatore sincrono

Il contatore sincrono è spesso realizzato con FF-JK che ricevono lo stesso clock, quindi tutte le uscite commutano con lo stesso ritardo, ma la commutazione è condizionata dallo stato di J e K.

Si nota, nella [figura a destra](#), come i clock sono tutti collegati in maniera diretta, ma il valore delle uscite dipende dalle uscite degli stati precedenti.

Dal terzo FF in poi, gli stadi sono tutti uguali.



Differenze dei ritardi nei contatori sincroni/asincroni

Dal [grafico a destra](#), si vede come i ritardi del contatore asincrono si accumulano, invece i ritardi del contatore sincrono sono tutti uguali.

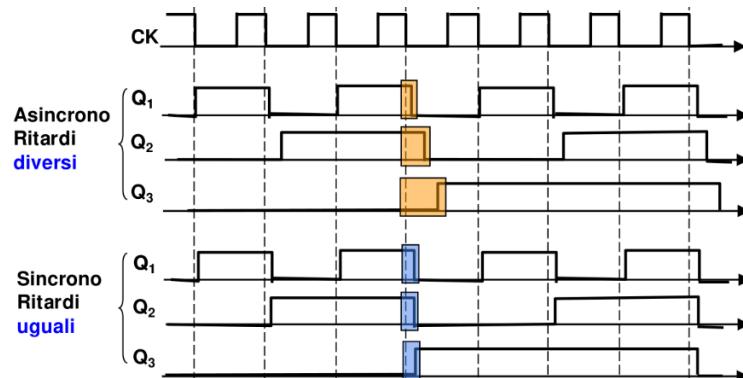
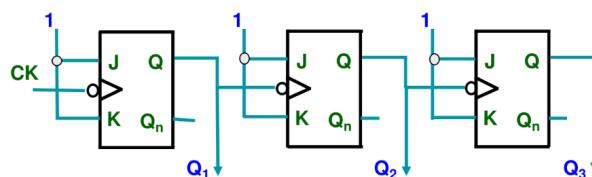


Diagramma temporale dei contatori sincroni e asincroni

FF-JK come contatore / divisore

Se mettiamo $J = 1$ e $K = 1$ un FF-JK cambia stato a ogni clock, quindi permette di realizzare dei contatori asincroni.



FF-JK a tre stadi (modulo 8), attivo sul fronte 1→0 del clock

Il primo FF commuta ad ogni fronte di discesa, il secondo ogni due, il terzo ogni 4. In questo modo, abbiamo in uscita un contatore a tre stadi.

L'uscita ha il MSB indicato dall'ultimo FF e viceversa, mentre ha il LSB indicato dal primo FF.

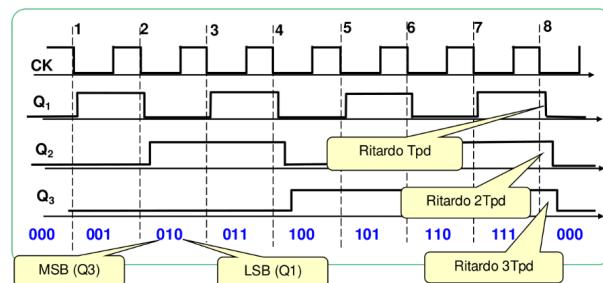


Diagramma temporale FF-JK a tre stadi

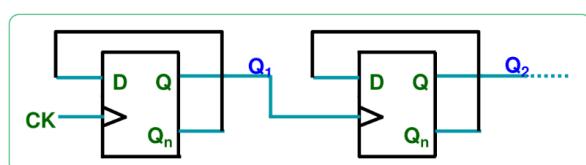
Divisore

Un divisore è un tipo particolare di contatore di cui viene utilizzata una sola uscita Q_n , con modulo M (variabile).

$$M = \frac{F_{clock}}{F_{qn}}$$

Divisore modulo 2/4

La caratteristica del divisore di modulo 2/4 è quella di dividere la frequenza di clock in modulo 2/4. Con M stadi, si ha una divisione modulo 2^M .



Divisore modulo 2/4.

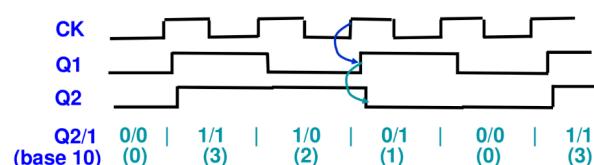


Diagramma temporale divisore modulo 2/4.



Macchina a stati finiti (FSM)

▼ Creatore originale: @LucaCaffa

FSM di Moore

Rete di stato futuro

Registro di stato

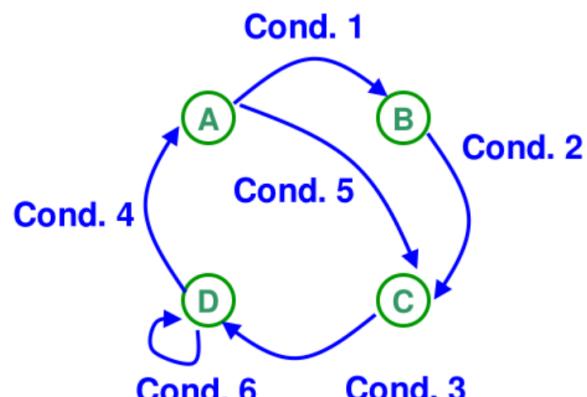
Rete di uscita

FSM di Mealy

Esempio - FSM per una lavatrice

Una **macchina a stati finiti** (in inglese Finite State Machine o **FSM**) è un modello usato per rappresentare un sistema che può trovarsi in un numero finito di stati e che può cambiare stato in risposta a determinati eventi o input.

Gli elementi di memoria (flip-flop) contengono la condizione 0/1 che identifica gli stati, e da ogni stato possono esserci più uscite. I passaggi da uno stato ad un altro sono rappresentati da archi.



Esempio di diagramma degli stati



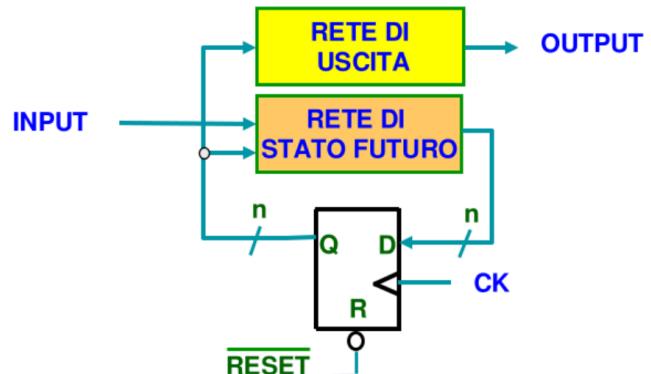
Le condizioni di uscita da un singolo stato devono essere **mutuamente esclusive**. Prendendo in esempio lo stato A, la condizione 1 e la 5 devono essere diverse.

FSM di Moore

Nella FSM di Moore (circuito sequenziale), l'uscita dipende **solo dallo stato corrente**.

Gli elementi che compongono questa FSM sono:

- una rete di uscita;
- una rete di stato futuro;
- un registro di stato.



FSM di Moore come circuito sequenziale

Rete di stato futuro

La rete di stato futuro è un circuito combinatorio e la sua uscita dipende sia dall'input che dallo stato presente.

Registro di stato

Il registro di stato memorizza lo stato attuale codificato su n bit. Partendo da uno stato, si passa al successivo in base alla condizione di uscita.

Rete di uscita

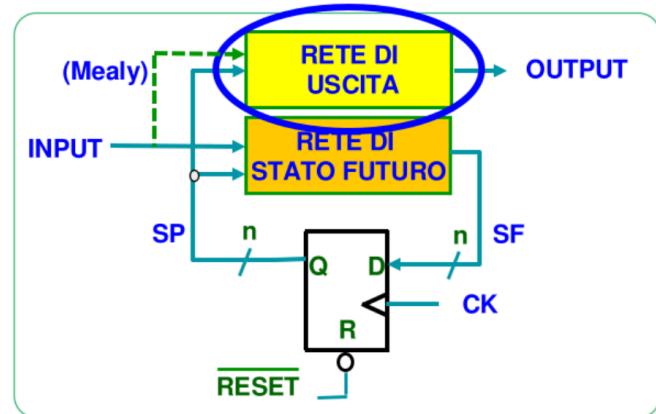
La rete di uscita è un circuito sequenziale e decide le uscite in base allo stato presente indipendentemente dagli ingressi.

FSM di Mealy

Nelle FSM di Mealy l'uscita dipende **sia dallo stato presente che dall'input**.

Gli elementi che la compongono sono gli stessi della macchina di Moore.

La differenza sta nella rete di uscita, perché sceglierà l'output analizzando l'input e lo stato attuale.



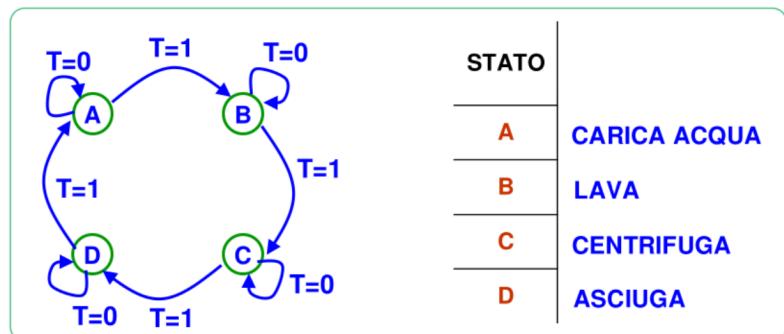
FSM di Mealy

Esempio - FSM per una lavatrice

Prendiamo in considerazione il funzionamento di una lavatrice.

La FSM che descrive il suo funzionamento avrà 4 stati:

- Carica Acqua;
- Lava;
- Centrifuga;
- Asciuga.



FSM per una lavatrice.

Ogni stato ha 2 condizioni: una per cambiare stato e una per rimanere nello stato attuale. Questo perché vogliamo che uno stato perduri nel tempo e non si esca subito.

Questa possibile
rappresentazione della FSM
potrebbe avere come input
un contatore, che fornisce il
segnalet $T=1$ (cambiare
stato) ogni 15 minuti.
A lato è presente la tavola
che descrive gli stati futuri
in base allo stato presente.

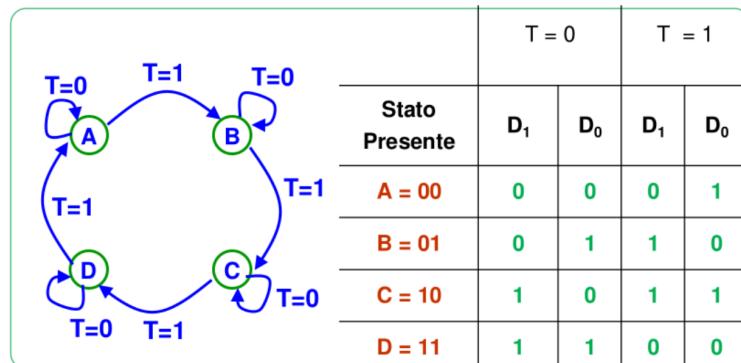


Tavola degli stati futuri

Riusciamo a capire il funzionamento della FSM guardando questa tabella:

Osservando la prima riga, vediamo che siamo in $A = 00$;

- se $T=0$ rimaniamo in A, infatti $D_1 = 0, D_0 = 0$ (stato A),
- se $T=1$ dobbiamo cambiare stato e andare in B, quindi dobbiamo andare in $D_1 = 0, D_0 = 1$ (stato B).

Lo stesso ragionamento si può applicare a tutti i rimanenti stati.



Nell'immagine sopra non viene descritto, ma spesso i bit per lo stato
presente vengono indicati come Q_1 e Q_0 .

100

Forme Canoniche

▼ Creatore originale: @LucaCaffa

Forme Canoniche

[Mintermini](#)

[Maxtermini](#)

[Rappresentazione standard](#)

[Somma canonica di mintermini \(SOM\)](#)

[Prodotto canonico di maxtermini \(POM\)](#)

Forme standard

[Somma di prodotti \(SOP\) standard](#)

[Esempio - Semplificazione di SOP](#)

[Insiemi di operazioni completi](#)

[Insieme NAND](#)

[Insieme NOR](#)

Forme Canoniche

Le **forme canoniche** sono modi standardizzati per descrivere le funzioni booleane. Queste forme hanno una diretta corrispondenza con le tabelle di verità, permettendo di determinare se due funzione sono identiche.

Le principali forme sono due:

- somma di mintermini;
- prodotto di maxtermini.

Mintermini

Un **minterm** è una combinazione delle variabili di ingresso che rappresenta una singola riga della tabella della verità in cui la funzione vale 1. Un minterm è il **prodotto (AND)** di tutte le variabili di ingresso che appaiono in forma negata o affermata.

$$f(x_1, x_2, x_3) = x_1 x_2 x_3$$



Se la variabile ha valore 1 appare in forma diretta (x), se ha valore 0 appare in forma negata (\bar{x}).

Maxtermini

Il **maxterm** è l'opposto dei mintermini, poiché rappresentano le combinazioni delle variabili di ingresso per cui una funzione booleana vale 0. Un maxterm è la **somma (OR)** di tutte le variabili di ingresso che appaiono in forma negata o affermata.

$$f(x_1, x_2, x_3) = x_1 + x_2 + x_3$$



Se la variabile ha valore 0 appare in forma diretta (x), se ha valore 1 appare in forma negata (\bar{x}).

Rappresentazione standard

Abbiamo visto come ogni combinazione delle variabili di una funzione booleana può essere scritto come:

- un mintermine, dove la funzione vale 1;
- un maxtermine, dove la funzione vale 0.

Per identificarli in maniera comoda, utilizziamo dei pedici per "contare" a quale mintermine (o maxtermine) ci stiamo riferendo.

A	B	C	Mintermine
0	0	0	m_0
0	0	1	m_1
0	1	0	m_2
0	1	1	m_3
1	0	0	m_4
1	0	1	m_5
1	1	0	m_6
1	1	1	m_7

Proviamo a scrivere il mintermine m_2 : avendo $A = 0, B = 1, C = 0$, si ha la sequenza 010, che porta ad avere il mintermine corrispondente: $\overline{A}B\overline{C}$.

Proviamo a scrivere il maxtermine M_2 : avendo $A = 0, B = 1, C = 0$, si ha la sequenza 010, che porta ad avere il maxtermine corrispondente: $A\overline{B}C$.



Dal teorema di De Morgan, si ha che M_2 è il complemento di m_2 , e viceversa.

$$M_2 = \overline{m_2} \Leftrightarrow m_2 = \overline{M_2}$$

Somma canonica di mintermini (SOM)

In questa forma, una funzione booleana è scritta come la somma (OR) di prodotti (AND). Ogni prodotto è un mintermine.

Supponiamo di avere la tabella della verità definita a lato per una funzione $f(A, B)$.

La funzione vale 1 per:

- $A=0, B=1$, per cui il mintermine è $\overline{A}B$;
- $A=1, B=0$, per cui il mintermine è $A\overline{B}$.

A	B	$f(A, B)$
0	0	0
0	1	1
1	0	1
1	1	0

La somma di mintermini è definita come:

$$f(A, B) = \overline{A}B + A\overline{B} = m_1 + m_2$$

Prodotto canonico di maxtermini (POM)

In questa forma, una funzione booleana è scritta come il prodotto (AND) di somme (OR). Ogni somma è un maxtermine.

Supponiamo di avere la tabella della verità definita a lato per una funzione $f(A, B)$.

La funzione vale 0 per:

- $A=0, B=0$, per cui il maxtermine è $A + B$;
- $A=1, B=1$, per cui il maxtermine: $\overline{A} + \overline{B}$.

A	B	$f(A, B)$
0	0	0
0	1	1
1	0	1
1	1	0

Il prodotto di maxtermini è definito come:

$$f(A, B) = (A + B)(\overline{A} + \overline{B}) = M_1 + M_2$$

Forme standard

Oltre alle forme SOM e POM, esistono altre forme standard, come:

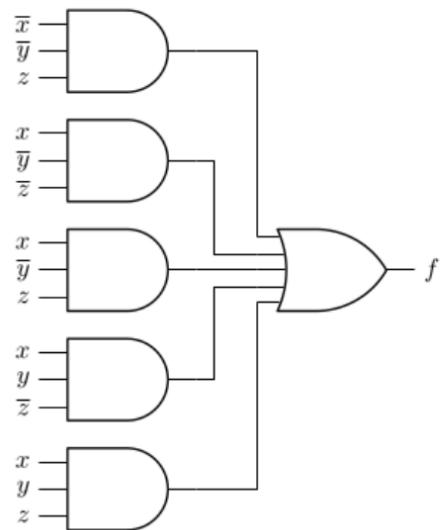
- somma di prodotti (SOP);
- prodotti di somme (POS).

La differenza con SOM e POM è il numero di variabili, infatti non abbiamo bisogno che tutte le variabili di ingresso siano presenti. In realtà, SOP è una forma più generica e semplice di SOM, come POS è una forma più generica di POM.

Esistono anche delle forme miste non standard che mischiano prodotti e somme.

Somma di prodotti (SOP) standard

Questa forma contiene somme di prodotti, ovvero termini OR di termini AND. L'implementazione è, quindi, una rete a due livelli di porte logiche, dove il primo livello sono delle porte AND e il secondo livello è una singola porta OR.



Rete a due livelli di porte logiche

▼ Esempio - Semplificazione di SOP

Prendiamo, in esempio, la seguente funzione:

$$f(x, y, z) = \bar{x}\bar{y}z + x\bar{y}\bar{z} + x\bar{y}z + xy\bar{z} + xyz$$

1. possiamo semplificare mettendo in comune x ;

$$f(x, y, z) = \bar{x}\bar{y}z + x(\bar{y}\bar{z} + \bar{y}z + y\bar{z} + yz)$$

2. ! Ricordiamo che $A + \bar{A} = 1$ e che $Y + 1 = Y$ come proprietà dei valori logici.

Se ipotizziamo che $\bar{y}z = A$, allora $\bar{A} = \bar{\bar{y}}\bar{z} = y\bar{z}$, possiamo sommare i termini $\bar{y}z$ e $y\bar{z}$, in modo da semplificare ulteriormente l'espressione;

$$f(x, y, z) = \bar{x}\bar{y}z + x(\bar{y}\bar{z} + 1 + yz) = \bar{x}\bar{y}z + x(\bar{y}\bar{z} + yz)$$

3. nuovamente, sapendo che $\bar{y}\bar{z} + yz = 1$, sommiamo i termini dentro le parentesi, come nel punto (2);

$$f(x, y, z) = \bar{x}\bar{y}z + x$$

4.  Ricordiamo la legge di De Morgan:

$$\overline{AB} = \bar{A} + \bar{B} \Rightarrow \overline{xy} = \bar{x} + \bar{y}$$

Espandiamo il prodotto;

$$f(x, y, z) = (\bar{x} + \bar{y})z + x$$

5. eseguiamo i prodotti con z ;

$$f(x, y, z) = \bar{x}z + \bar{y}z + x$$

6.  Si ricorda la seguente proprietà:

$$A + \overline{AB} = A + B$$

Dimostriamola brevemente come segue, attraverso l'identità di $A + \overline{A}$.

$$\begin{aligned} (A + \overline{A})(A + B) &= A + AB + 0 + \overline{A}B \\ &= A(1 + B) + \overline{A}B \\ &= A + \overline{A}B \end{aligned}$$

Semplifichiamo per $\bar{x}z + x$;

$$f(x, y, z) = z + \bar{y}z + x$$

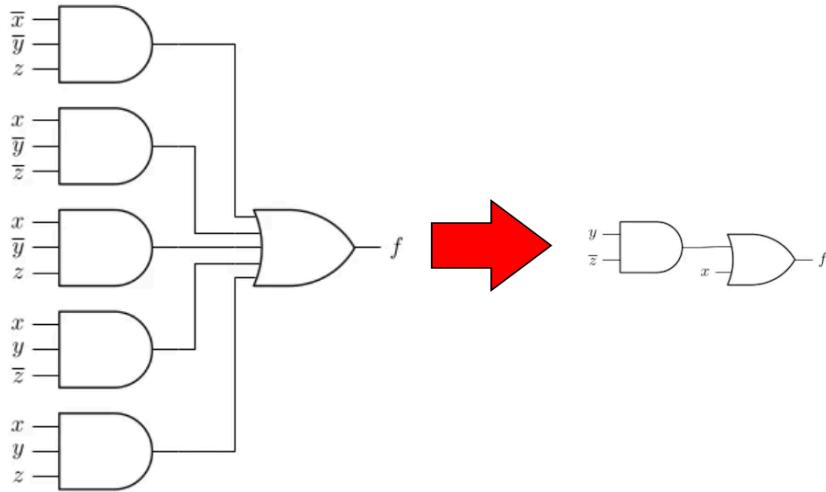
7. mettiamo in comune z ;

$$f(x, y, z) = z(1 + \bar{y}) + x$$

8. visto che $(1 + \bar{y}) = \bar{y}$, si può terminare la semplificazione come segue:

$$f(x, y, z) = z\bar{y} + x$$

Dopo tutto questo procedimento, abbiamo la seguente semplificazione di circuito logico:



Semplificazione del circuito logico in SOP

Insiemi di operazioni completi

L'algebra booleana è interamente derivabile da un'insieme di porte $\{\text{NOT}, \text{AND}, \text{OR}\}$, poiché, grazie al teorema di De Morgan, possiamo passare da un'insieme all'altro.

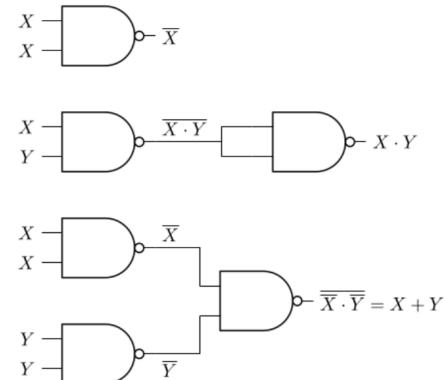
Ci sono due ulteriori insiemi molto importanti, gli insiemi $\{\text{NAND}\}$ e $\{\text{NOR}\}$, da cui possiamo derivare l'intera algebra booleana.

Insieme NAND

Dall'insieme $\{\text{NAND}\}$ si possono ricavare gli operatori NOT, AND e, dal teorema di De Morgan, l'operatore OR, combinando le porte NAND nel modo rappresentato in [figura](#).

A partire dal circuito logico più in alto:

1. utilizzo di una porta NAND per la realizzazione di una funzione NOT;
2. utilizzo di due porte NAND per la realizzazione di una funzione AND;
3. utilizzo di due porte NAND per la realizzazione di una funzione OR.



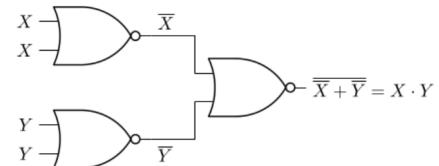
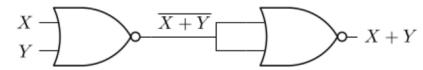
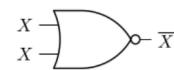
Da porta NAND a NOT, AND e OR.

Insieme NOR

Dall'insieme {NOR}, si possono ricavare gli operatori NOT, OR e, dal teorema di De Morgan, l'operatore AND, combinando le porte NOR nel modo rappresentato in [figura](#).

A partire dal circuito logico più in alto:

1. utilizzo di una porta NOR per la realizzazione di una funzione NOT;
2. utilizzo di due porte NOR per la realizzazione di una funzione OR;
3. utilizzo di due porte NOR per la realizzazione di una funzione AND.



Da porta NOR a NOT, OR e AND.



Mappe di Karnaugh (K-Map)

▼ Creatore originale: @LucaCaffa

Esempio - K-map di 3 variabili

Impliciti

Caratteristiche

Impliciti primi

Impliciti essenziali

Passi di minimizzazione

Copertura con valori "don't care"

Metodo di Quine-McCluskey (QMC)

Spiegazione di ogni passo del metodo QMC

Fase di Copertura

Fase di espansione

Fase di espansione - Ulteriore esempio

Metodo di Espresso

Esempio - Minimizzazione con Espresso

Esempio - Minimizzazione con Espresso 2

Una **mappa di Karnaugh** è uno strumento grafico utilizzato in logica booleana per semplificare espressioni logiche o funzioni booleane.

È rappresentata attraverso una tabella che serve per rappresentare i valori di verità di una funzione logica in modo ordinato, così da rendere più facile trovare gruppi di valori uguali (1 o 0) e, quindi, poter semplificare l'espressione logica.

Esempio - K-map di 3 variabili

Prendiamo in esempio la tabella di verità a lato.

Scriviamo una tabella che ha come indice delle colonne le combinazioni degli ingressi xy, mentre come indice delle righe ha le combinazioni di z.

Ogni cella con il valore 1 è descritta da un mintermine.

f	xy	00	01	11	10
z	0	0	1	0	0
¬z	0	1	1	1	1

x	y
0	0
0	0
0	1
0	1
1	0
1	0
1	1
1	1

Tabella di Karnaugh associata alla tabella di verità a lato

Il nostro obiettivo è semplificare la funzione logica, in modo da avere la sua forma meno costosa possibile dal punto di vista dei mintermini.

Per esempio, se prendessimo tutte le caselle con 1, avremmo 4 mintermini da 3 variabili ciascuno.

Possiamo, invece, prendere gli 1 a rettangoli grandi di dimensione pari a una potenza di due, in modo da semplificare la funzione, ricordando che, rispetto a un determinato rettangolo, si prendono i valori all'interno degli indici che **non variano** per il rettangolo:

- se prendiamo 4 rettangoli da 1 blocco ognuno avremo:

$$f(x, y, z) = \bar{x}y\bar{z} + \bar{x}yz + xyz + x\bar{y}z$$

	xy	00	01	11	10
z	0	0	1	0	0
	1	0	1	1	1

- se prendiamo 2 rettangoli da 2 blocchi ognuno avremo:

$$f(x, y, z) = \bar{x}y + xz$$

	xy	00	01	11	10
z	0	0	1	0	0
	1	0	1	1	1

Si può notare come nel primo tipo di raggruppamento abbiamo 4 mintermini, mentre nel secondo ne abbiamo due, e quindi il secondo possibile raggruppamento risulta meno costoso.

Implicanti

Un implicante è un gruppo di **celle adiacenti** (dove il valore è 1) in una mappa di Karnaugh che può essere semplificato a un singolo termine booleano.

Caratteristiche

- Ogni implicante rappresenta una **combinazione di variabili** che è comune a tutte le celle del gruppo;
- Gli implicanti possono essere di **diverse dimensioni**, da un singolo 1 (minimo implicante) a gruppi più grandi;
- Più grande è l'implicante, più semplice è l'espressione booleana che rappresenta.

Implicanti primi

Un **implicante primo** è un implicante che non può essere esteso ulteriormente senza includere celle che non contengono il valore 1.

Un implicante è considerato primo se non può essere diviso in gruppi più piccoli che coprono gli stessi 1.

Implicanti essenziali

Un **implicante essenziale** è un implicante che copre almeno un 1 che non è coperto da alcun altro implicante.

Gli implicanti essenziali sono quelli che **non possono essere evitati** se si vogliono coprire tutti i valori 1 nella mappa di Karnaugh.

Passi di minimizzazione

In maniera informale, si può individuare una sequenza di passi per trovare la soluzione minima:

1. si trovano tutti gli implicanti primi;
2. si includono nella soluzione tutti gli implicanti essenziali;
3. si prende un insieme di implicanti primi (non essenziali) che coprono tutti gli 1 non ancora presi;
4. si cerca la minima sovrapposizione tra implicanti primi.

Copertura con valori "don't care"

Siamo in condizione di "don't care" quando un determinato valore della funzione non ci interessa. In celle di questo tipo, graficamente si indicano valori di "don't care" sulla mappa con il simbolo "-" o "x". Possiamo considerare queste celle come contenenti sia 0 che 1, a seconda dell'utilità nel creare insiemi di implicanti di dimensione massima.

Ad esempio, se abbiamo $x=0$, $y=1$, $z=1$ e $w=0$ e non ci importa sapere quanto vale la funzione in quella cella, indichiamo il valore non importante con "x".

Nella K-map di fianco, i simboli "x" sono i valori don't care. In questo caso, ci conviene prendere il valore "don't care" in posizione 1101 (considerandolo come 1), ma non ci conviene prendere quello in posizione 1100, né quello in posizione 0110, poiché porterebbero ad una soluzione peggiore.

		xy	00	01	11	10
		zw	00	01	11	10
00	00	0	0	x	0	
		1	1	x	1	
01	01	1	1	0	0	
		0	x	0	0	

Esempio di copertura con don't care

Metodo di Quine-McCluskey (QMC)

Il metodo di Quine-McCluskey è un [algoritmo sistematico per la minimizzazione](#), simile alla mappa di Karnaugh, ma più adatto a funzioni con molte variabili.

Si compone di due fasi che usano delle tabelle per memorizzare i vari risultati intermedi:

- espansione: generazione di tutti gli implicanti primi;
- copertura: scelta del minor numero di implicanti primi.

Spiegazione di ogni passo del metodo QMC

Per spiegare ogni passo, è più comodo utilizzare un esempio. Ipotizziamo di avere la tabella che contiene i valori degli ingressi e della funzione di uscita.

La prima colonna, identificata con il simbolo i , è utilizzata solo per contare e identificare le righe, ma non è un ingresso.

i	x	y
0	0	0
1	0	0
2	0	1
3	0	1
4	1	0
5	1	0
6	1	1

i	x	y
7	1	1

Fase di Copertura

Definiamo i vari passi:

1. guardare la tabella e prendere i mintermini dove la funzione vale 1. In questo esempio, abbiamo i mintermini 0, 2, 3, 4, 6, 7;
2. dobbiamo prendere i mintermini e metterli in dei gruppi, secondo i seguenti ragionamenti:

- a. analizziamo m_0 , cercando di capire quanti "1" ha tra i suoi ingressi. Non avendone nessuno, si cercano altri mintermini che non hanno "1" negli ingressi;
- b. dato che non ci sono mintermini che non hanno 1 negli ingressi, m_0 è in un gruppo da solo;
- c. procediamo con il mintermine m_2 , cercando di capire quanti "1" ha tra i suoi ingressi. Avendone uno, ($m_2 = 010_2$), si cercano altri mintermini che contengono un singolo "1" negli ingressi;
- d. si sono altri mintermini che hanno un "1" tra gli ingressi, come $m_4 = 100_2$, e quindi mettiamo m_2 e m_4 nello stesso gruppo;
- e. continuando così, si arriva alla situazione descritta nella tabella a lato, dove i gruppi sono descritti dai diversi colori.

3. una volta costruita la tabella con i gruppi, dobbiamo cercare di ridurla ulteriormente, combinando i gruppi tra loro, secondo i seguenti ragionamenti:

- a. è necessario chiedersi se due mintermini di gruppi diversi possono unirsi, a patto di considerare una variabile in meno, e quindi i due mintermini si possono unire se differiscono solo di un bit;
- b. prendiamo come esempio m_0 e m_2 , i bit di ingresso sono, rispettivamente, 000_2 e 010_2 . Questi due mintermini differiscono solo per il valore di y , quindi se si scrive $0-0_2$, si possono unire e indicare entrambi in una singola riga;
- c. ripetere questo ragionamento per tutti i gruppi, ricordando che $(0,2)$ significa che ho unito m_0 e m_2 .

i	x
0	0
2	0
4	1
3	0
6	1
7	1

4. ripartendo dal [passo 2](#), si ripete il ragionamento fatto fino ad ora sui gruppi, cercando di capire se si può semplificare ancora, finché non si arriva ad una situazione di stallo;
5. alla fine delle semplificazioni, si avrà una tabella come quella a lato. La colonna p serve solo per dare un nome alle soluzioni parziali trovate fino ad ora.

i	x	y	z
(0,2)	0	-	0
(0,4)	-	0	0
(2,3)	0	1	-
(2,6)	-	1	0
(4,6)	1	-	0
(3,7)	-	1	1
(6,7)	1	1	-

i	x	y
(0,2,4,6)	-	-
(2,3,6,7)	-	1

Fase di espansione

Lo scopo di questa fase è scegliere il minor numero di implicanti primi possibili utilizzabili per coprire tutta la funzione. Esistono diversi metodi matematici per risolvere questo problema, ma non sono trattati nel corso, quindi possiamo sceglierli "ad occhio":

1. iniziamo dall'ultima tabella trovata. In questo momento non ci interessa la colonna i , che serviva per distinguere i gruppi, ma dobbiamo ragionare per righe;

i	x	y
(0,2,4,6)	-	-

i	x	y
(2,3,6,7)	-	1

2. costruiamo una **tabella di copertura**, cioè una tabella che ci dice, per ogni implicant primo, quale mintermine è coperto.
Ricordiamo che i mintermini sono tutti i valori nella colonna i , mentre gli implicanti sono le righe della colonna p ;

a. guardiamo l'ultima tabella ottenuta e mettiamo una colonna per ogni mintermine.
Quindi, inseriamo i valori $m_0, m_2, m_3, m_4, m_6, m_7$ in ordine sulle colonne;

	m_0	m_2	m_3	m_4	m_6

b. possiamo inserire, in ragione di uno per riga, gli implicant primi.
Quindi, inseriamo p_0, p_1 ;

	m_0	m_2	m_3	m_4	m_6	m_7
p_0						
p_1						

c. segniamo, per ogni riga, e quindi per ogni impilante, quale mintermine copre, mettendo il simbolo oppure un 1 o una X;

	m_0	m_2	m_3	m_4	m_6	m_7
p_0						
p_1						

d. per sapere quali mintermini copre un impilante, torno all'[ultima tabella della fase di espansione](#) e guardo la corrispondente cella della colonna i .

	m_0	m_2	m_3	m_4	m_6
p_0	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
p_1		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>

Per p_0 abbiamo
(0,2,4,6), e
quindi p_0 copre
 m_0, m_2, m_4, m_6

Ripeto il
procedimento
per p_1 .

- e. abbiamo, quindi, la [tabella di copertura completa](#). Dobbiamo prendere degli implicanti (righe) in modo da coprire tutti i mintermini (colonne), così che nessun mintermine rimanga tralasciato.

In questo esempio abbiamo già finito, poiché per coprire tutti i mintermini devo prendere entrambi gli implicanti p_0 e p_1 .

Fase di espansione - Ulteriore esempio

Poiché dallo scorso esempio non avevamo bisogno di considerazioni finali, prendo un'altra tabella di copertura già costruita, così da analizzarla di passo in passo.

In generale, l'unica regola che applichiamo è prendere gli implicanti che coprono più mintermini possibili.

	m_1	m_2	m_3	m_6	m_9	m_{10}
p_0						
p_1						
p_2	✓		✓		✓	
p_3	✓				✓	
p_4		✓	✓			✓
p_5		✓		✓		✓

Definiamo i vari passi:

1. controlliamo se esistono degli implicanti primi essenziali, ovvero delle righe che coprono dei mintermini che non copre

	m_1	m_2	m_3	m_6	m_9	m_{10}
p_0						
p_1						
p_2	✓		✓		✓	
p_3	✓				✓	
p_4		✓	✓			✓
p_5		✓		✓		✓

nessun
altro.

Nel nostro
esempio, p_5
copre m_6 ,
ma m_6 non
è coperto
da nessun
altro
implicante
primo
essenziale.

Nella
soluzione,
 p_5 sarà
presente.

Segniamo
in verde le
colonne che
abbiamo
preso
mettendo
 p_5 nella
soluzione;

2. si può
notare
come m_{12}
è ricoperto
sia da p_0
che da p_1 .
In questo
caso è
indifferentе
quale dei
due viene
preso, e
noi
prendiamo
quindi p_0 ;

	m_1	m_2	m_3	m_6	m_9	m_{10}
p_0						
p_1						
p_2	✓		✓		✓	
p_3	✓				✓	
p_4		✓	✓			✓
p_5		✓		✓		✓

3. I mintermini
mancanti sono
 m_1, m_2, m_9, m_{11}
, e sono tutti
coperti da p_2 ,
quindi lo
prendiamo nella
soluzione.

	m_1	m_2	m_3	m_6	m_9	m_{11}
p_0						
p_1						
p_2	✓		✓		✓	
p_3	✓				✓	
p_4		✓	✓			✓
p_5		✓		✓		✓

Avremmo potuto prendere anche p_3, p_4 insieme, ma non sarebbe stata la soluzione minima, e quindi neanche la migliore.

Dopo questo procedimento, abbiamo trovato la seguente soluzione.

$$f(x, y, z, w) = p_0 + p_2 + p_5 = xy\bar{z} + \bar{y}w + z\bar{w}$$

Metodo di Espresso

Il metodo Espresso è un **algoritmo avanzato per la minimizzazione di funzioni booleane**. È più efficiente del metodo di Quine-McCluskey, soprattutto per funzioni complesse con molte variabili.

Il problema del metodo QMC è la crescita esponenziale degli implicanti primi, oltre al fatto che il riconoscimento della copertura minima può rientrare nella classe dei problemi NP-completi.

Il metodo espresso opera in più passaggi:

- **espansione (Expand)**: trasformare ogni termine in un implicante primo più grande possibile;
- **copertura ottima (Irredundant Cover)**: trovare il minor numero di implicanti primi che coprono tutti i mintermini della funzione;
- **riduzione (Reduce)**: ridurre gli implicanti per migliorare la copertura globale;
- **iterazione (Repeat Until Convergence)**: le fasi Expand, Reduce e Irredundant vengono ripetute finché non si raggiunge una soluzione stabile.

▼ Esempio - Minimizzazione con Espresso

Consideriamo la seguente funzione, con valori "don't care" in $\sum_d(0, 3)$.

$$F(A, B, C) = \sum(1, 2, 5, 6, 7)$$

1. Eseguiamo l'espansione.

I mintermini sono: $001, 010, 101, 110, 111$.

Gli implicanti primi trovati potrebbero essere:

- $0\bar{1}$ (copre 001 e 101): $\bar{A}C$;
- $\bar{1}0$ (copre 010 e 110): $B\bar{C}$;
- $1\bar{1}\bar{1}$ (copre 110 e 111): AB .

AB/C	00	01
0	X	1
1	1	X

2. Eseguiamo la riduzione e la copertura.

La tabella di copertura mostra che:

- $B\bar{C}$ copre 010 e 110 ;
- $\bar{A}C$ copre 001 e 101 ;
- AB copre 110 e 111 .

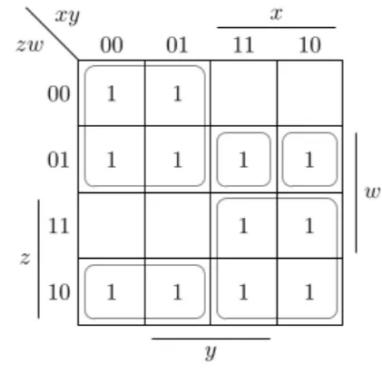
La soluzione minima potrebbe, quindi, essere:

$$F(A, B, C) = B\bar{C} + \bar{A}C + AB$$

▼ Esempio - Minimizzazione con Espresso 2

- partiamo dalla [k-map a lato](#).

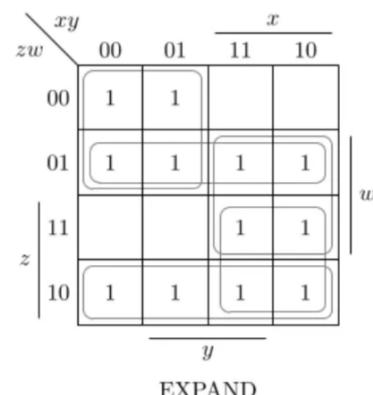
Il nostro obiettivo è considerare gli implicanti più grandi possibili, per poi andare a ridurli, verificando ogni volta la totale copertura;



Assunzione 1

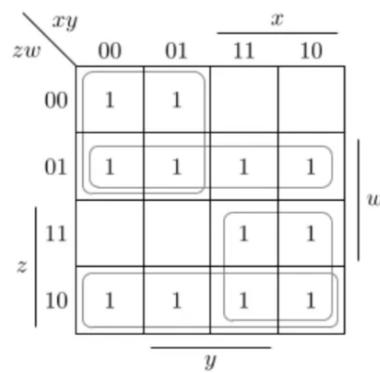
- fase di espansione, in cui prendiamo gli implicanti in modo da coprire tutta la tabella.

Stiamo cercando l'insieme con gli implicanti più grandi possibili. La [figura a lato](#) mostra come questo sia l'insieme massimale.



Espansione 1

- fase di copertura ottima, in cui, dall'insieme appena trovato, estraiamo una soluzione non ridondante, in modo che copra tutti i mintermini.



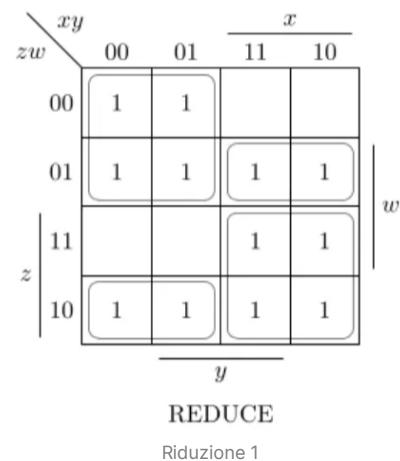
Copertura ottima 1

4. potremmo aver finito, oppure potrebbe esistere una soluzione migliore.

Quello che si fa è ridurre, ad uno ad uno, la dimensione di ogni implicante, garantendo sempre la copertura totale dei mintermini.

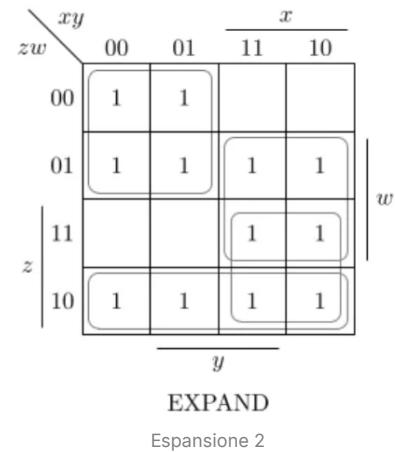
Se si riesce a fare una modifica, iteriamo i passi in modo continuare la ricerca di una soluzione ottima.

Qui abbiamo ridotto l'implicante in basso a sinistra, per cui abbiamo capito che il numero di implicanti può essere 4. La soluzione trovata non è ottima, e dobbiamo tornare al punto 2, cercando solo 4 implicanti;

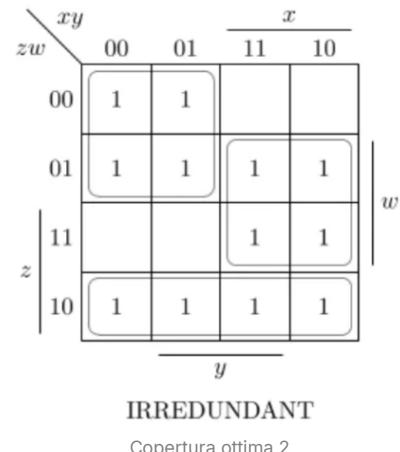


5. si espande nuovamente, ma con 4 implicanti.

Prendiamo l'insieme più grande di implicanti, cercandone 4;



6. fase di copertura ottima, in cui prendiamo un sotto insieme dall'insieme trovato al [punto 5](#). Adesso abbiamo trovato una soluzione ottima, e l'algoritmo finisce.





Ritardi e alee delle reti combinatorie

▼ Creatore originale: @LucaCaffa

Ritardi nelle Porte Logiche

Ritardo di Propagazione

Ritardi di Trasporto (Transport Delays)

Ritardi Inerziali (Inertial Delays)

Alee (Hazards) nelle reti combinatorie

Alee Statiche

Esempio - Circuito con Alea statica

Eliminazione delle Alee Statiche

Teorema Fondamentale

Metodo Sistematico

Esempio - Correzione

Trade-off

Ritardi nelle Porte Logiche

Ritardo di Propagazione

Ogni porta logica introduce un certo ritardo tra la variazione dell'ingresso e la corrispondente variazione dell'uscita.

Ritardi di Trasporto (Transport Delays)

I ritardi di trasporto sono i seguenti:

- t_{PLH} : tempo di transizione da livello basso a alto (Low-to-High);
- t_{PHL} : tempo di transizione da livello alto a basso (High-to-Low).

Ritardi Inerziali (Inertial Delays)

I ritardi inerziali si hanno quando si parla di transizioni non istantanee, con un andamento lineare nel tempo. In pratica, i segnali impiegano un tempo finito per stabilizzarsi.

Si considereranno **solo i ritardi di propagazione** per semplicità, ignorando i ritardi inerziali.

Alee (Hazards) nelle reti combinatorie

Le alee sono transizioni impulsive nell'uscita, causate dai ritardi di propagazione asimmetrici tra i percorsi logici. La causa è il disallineamento temporale tra segnali che dovrebbero cambiare simultaneamente. Si hanno, quindi, delle brevi instabilità nell'uscita, che però possono essere mascherati.

Alee Statiche

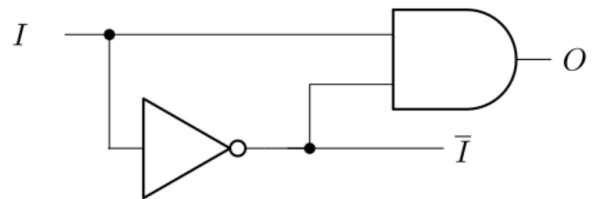
Le alee statiche si verificano quando l'uscita dovrebbe rimanere stabile, ma mostra un impulso indesiderato a causa di ritardi di trasporto.

Esempio - Circuito con Alea statica

1. Prendiamo in considerazione il circuito a lato, che comprende una porta AND e una porta NOT.

In teoria, l'uscita O non dovrebbe mai essere 1, perché in ingresso alla porta AND arrivano i segnali I e \bar{I} .

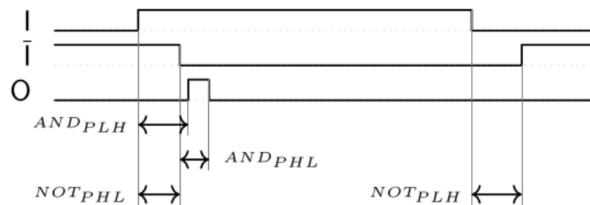
In realtà, è presente un breve istante in cui l'uscita O è 1.



Circuito con una porta AND e una porta NOT

2. Se guardiamo il segnale I , ad un certo istante va a 1.

La porta NOT, però, non manda immediatamente il segnale \bar{I} (quindi 0) alla porta AND, e, quindi, per un breve istante si vede che l'uscita O diventa 1, portando ad avere un'alea statica.



Analizziamo i ritardi uno per volta:

- AND_{PLH} è il tempo che la porta AND ci mette a passare da 0 a 1, ed inizia quando I va a 1, terminando quando la porta AND imposta 1 in uscita;
- NOT_{PLH} è il tempo che la porta NOT ci mette a far passare il segnale \bar{I} da 0 a 1, iniziando nell'istante in cui I cambia;
- AND_{PHL} è il tempo che la porta AND ci mette a passare da 1 a 0, iniziando quando \bar{I} diventa 0;
- NOT_{PLH} è il tempo che la porta NOT ci mette a passare da 0 a 1, iniziando quando I passa a 0.

Eliminazione delle Aleee Statiche

Teorema Fondamentale

Il seguente **teorema fondamentale** definisce la **condizione di libertà da Aleee**: una rete combinatoria a due livelli (SOP - Sum of Products) è libera da aleee statiche se, per ogni coppia di 1 adiacenti nella mappa di Karnaugh, esiste un implice primo che li copre entrambi.

Metodo Sistematico

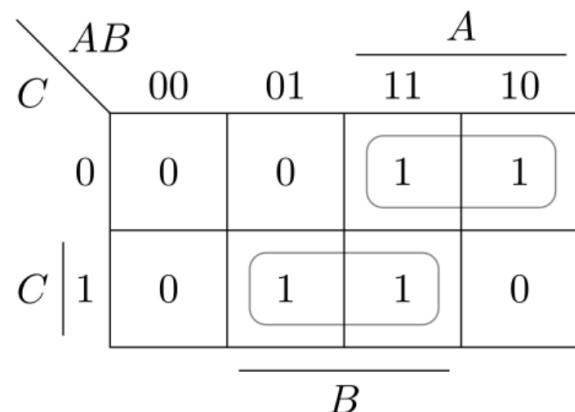
Come metodo sistematico, si definiscono due punti:

1. identificare le **transizioni critiche**, cercando coppie di 1 adiacenti nella mappa non coperti dallo stesso implice;

- aggiungere gli implicanti ridondanti, introducendo termini di prodotto aggiuntivi per mascherare le alee, anche se ciò aumenta la complessità del circuito.

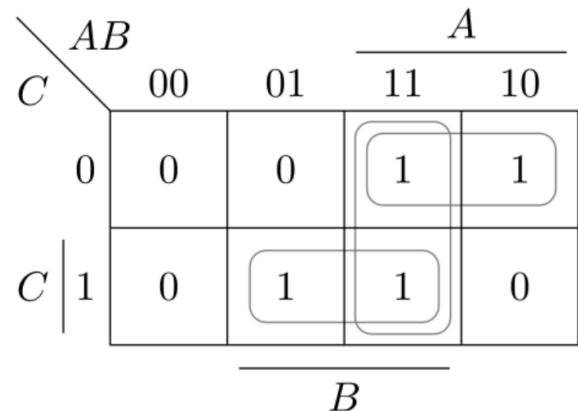
Esempio - Correzione

La K-map a lato mostra una possibile alea, poiché sappiamo, dal [teorema fondamentale](#), che due 1 adiacenti non coperti dallo stesso "rettangolo" possono creare una alea.



K-map con un possibile problema di alea

Possiamo risolvere il problema eliminando la alea, per cui dobbiamo prendere un implicante aggiuntivo ridondante, in modo da coprire ogni 1 adiacente con uno stesso implicante. Dalla [figura a lato](#) si vede l'implicante aggiuntivo.



K-map con problema di alea risolto

Trade-off

Per garantire la stabilità temporale, e quindi nessuna presenza di alea statiche, bisogna aumentare il numero di porte logiche. Il costo aumenta e il circuito non è minimo, ma si ha la sicurezza di non avere alee.



Ottimizzazione per FSM

▼ Creatore originale: @Samuele Gentile

Algoritmo "informale"

Esempio - Controllore di parità

Esempio - Riconoscitore di sequenze

Metodo della tabella delle implicazioni

Esempio - Riempimento e semplificazione di una tabella

Esempio - Controllore di parità con tabella delle implicazioni

Codifica degli stati

Codifiche di stato e misura delle metriche con distanza di Hamming

Approccio basato sulla mappa di Karnaugh:

Metodi euristici

Esempio - Metodi euristici 1

Esempio - Metodi euristici 2

Durante la realizzazione di una FSM è possibile avere degli stati equivalenti, ovvero che:

- per qualsiasi ingresso producono le stesse uscite;
- hanno transizioni verso stati uguali o equivalenti.

Quando esistono stati equivalenti, possiamo combinarli in uno solo. In questo modo possiamo avere un risparmio a livello di FF e di porte logiche.

Se, ad esempio, avessimo 5 stati, avremmo bisogno di 3 bit, ma se riuscissimo a condensare due stati in uno ne basterebbero 2, e quindi servirebbe un FF in meno.

L'obiettivo è, quindi, quello di utilizzare il minor numero di bit per coprire tutti gli stati.

Algoritmo "informale"

Per l'algoritmo "informale", bisogna:

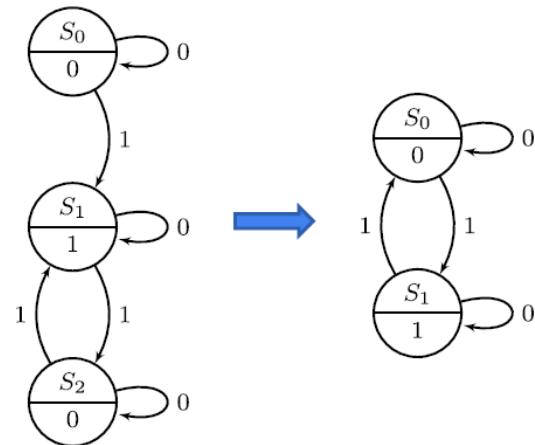
1. guardare la tabella delle transizioni tra gli stati;
2. cercare gli stati con stesso comportamento per le uscite;
3. se hanno gli stessi stati futuri sono equivalenti;
4. si combinano gli stati equivalenti trovati;
5. ripetere fino a che non si riesce più a compattare.

▼ Esempio - Controllore di parità

La FSM illustrata è un controllore di parità, ovvero conta i numeri di 1:

- se sono in numero dispari, restituisce 1;
- se sono in numero pari, restituisce 0.

Ad "occhio", si può vedere che gli stati S1 e S2 sono equivalenti.



Semplificazione della FSM del controllore di parità

Costruendo la tabella degli stati, possiamo vedere come non possiamo applicare l'algoritmo sopra elencato, poiché sono presenti delle "etichette" diverse nella colonna degli stati futuri di S1 e S2.

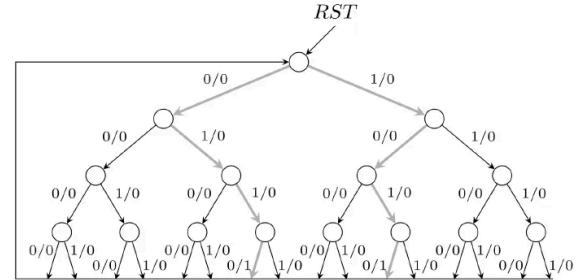
L'algoritmo non copre il comportamento del cappio che si è verificato in questo caso.

Stato Presente	Stato Futuro		Uscita
	$x = 0$	$x = 1$	
S_0	S_0	S_1	0
S_1	S_1	S_2	1
S_2	S_2	S_1	0

Tabella degli stati della FSM

▼ Esempio - Riconoscitore di sequenze

In questo caso, vogliamo costruire una FSM che riconosca solo le righe 1010 o 0110.



Visualizzazione dell'albero RST delle scelte per la sequenza di caratteri

Come visto dall'algoritmo informale, partiamo dalla tabella degli stati.

Suddividiamo le righe in base al numero di bit per ogni sequenza di ingresso.

Sequenza Ingresso	Stato Presente	Stato Futuro		Uscita	
		$x = 0$	$x = 1$	$x = 0$	$x = 1$
RST	S_0	S_1	S_2	0	0
0	S_1	S_3	S_4	0	0
1	S_2	S_5	S_6	0	0
00	S_3	S_7	S_8	0	0
01	S_4	S_9	S_{10}	0	0
10	S_5	S_{11}	S_{12}	0	0
11	S_6	S_{13}	S_{14}	0	0
000	S_7	S_0	S_0	0	0
001	S_8	S_0	S_0	0	0
010	S_9	S_0	S_0	0	0
011	S_{10}	S_0	S_0	1	0
100	S_{11}	S_0	S_0	0	0
101	S_{12}	S_0	S_0	1	0
110	S_{13}	S_0	S_0	0	0
111	S_{14}	S_0	S_0	0	0

Tabella degli stati iniziale

Notiamo che S_{10} e S_{12} hanno gli stessi stati futuri e le stesse uscite, e risultano, quindi, equivalenti.

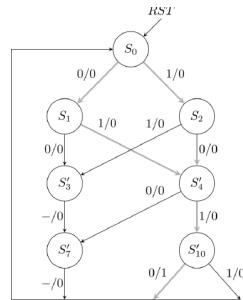
Notando ciò, S_{10} e S_{12} vengono accorpati in un unico stato S_{10}' .

Continuando con questo ragionamento, tutti gli stati evidenziati in grigio hanno gli stessi stati futuri e le stesse uscite. Di fatto, essi posso essere riscritti come un unico stato S_7' .

Sequenza Ingresso	Stato Presente	Stato Futuro		Uscita	
		$x = 0$	$x = 1$	$x = 0$	$x = 1$
RST	S_0	S_1	S_2	0	0
0	S_1	S_3	S_4	0	0
1	S_2	S_5	S_6	0	0
00	S_3	S_7	S_8	0	0
01	S_4	S_9	S_{10}'	0	0
10	S_5	S_{11}	S_{10}'	0	0
11	S_6	S_{13}	S_{14}	0	0
000	S_7	S_0	S_0	0	0
001	S_8	S_0	S_0	0	0
010	S_9	S_0	S_0	0	0
011 101	S_{10}'	S_0	S_0	1	0
100	S_{11}	S_0	S_0	0	0
110	S_{13}	S_0	S_0	0	0
111	S_{14}	S_0	S_0	0	0

Passaggio intermedio per l'ottimizzazione della tabella degli stati

Si procede in questo modo fino a raggiungere un punto in cui non ci sono più stati equivalenti. Notiamo come, dai 15 stati iniziali, siamo arrivati a solo 8.

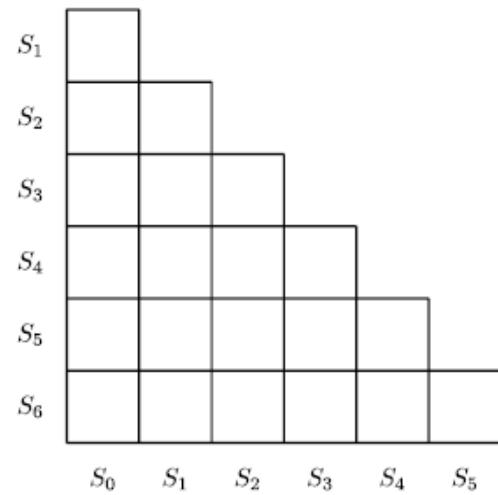
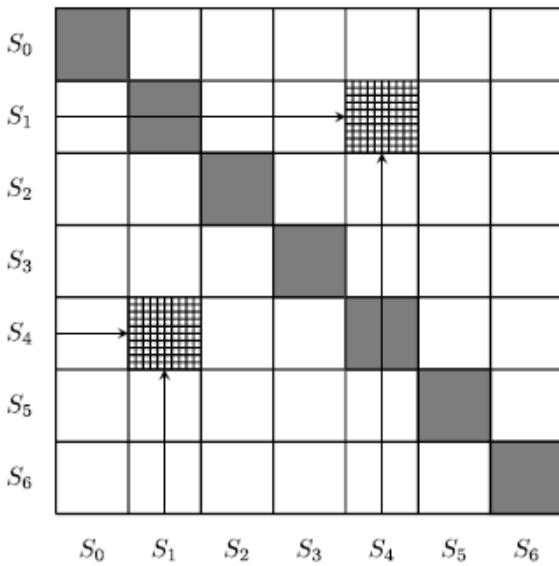


Sequenza Ingresso	Stato Presente	Stato Futuro		Uscita	
		$x = 0$	$x = 1$	$x = 0$	$x = 1$
RST	S_0	S_1	S_2	0	0
0	S_1	S'_3	S'_4	0	0
1	S_2	S'_4	S'_3	0	0
00 11	S'_3	S'_5	S'_7	0	0
01 10	S'_4	S'_6	S'_{10}	0	0
$\neg(011 101)$	S'_7	S'_8	S_0	0	0
011 101	S'_{10}	S_0	S_0	1	0

Tabella degli stati senza stati equivalenti

Metodo della tabella delle implicazioni

La tabella delle implicazioni è una tabella che definisce le relazioni tra tutti gli stati di una FSM. È una tabella con una struttura simmetrica ($S1 \equiv S4 \Leftrightarrow S4 \equiv S1$) e riflessiva ($S4 \equiv S4$) rispetto alla sua diagonale. Per queste proprietà, si semplifica la tabella rimuovendo la diagonale e tutte le celle al di sopra di essa.

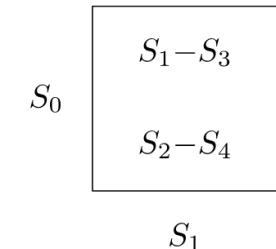


Esempio di struttura della tabella delle implicazioni

Le caselle sono riempite con due coppie di valori:

- la prima coppia di valori definisce gli stati a cui transiziona lo stato definito dall'indice della riga;
 - il primo valore della coppia è lo stato a cui transiziona con ingresso 0;
 - il secondo valore della coppia è lo stato a cui transiziona con ingresso 1.
- la seconda coppia di valori definisce gli stati a cui transiziona lo stato definito dall'indice della colonna.
 - il primo valore della coppia è lo stato a cui transiziona con ingresso 0;
 - il secondo valore della coppia è lo stato a cui transiziona con ingresso 1.

Per esempio, se S_0 va a S_1 con ingresso 0 e a S_2 con ingresso 1, e S_1 va a S_3 con ingresso 0 e a S_4 con ingresso 1, nella cella di indice $\langle S_0, S_1 \rangle$ ci saranno le coppie S_1-S_3 e S_2-S_4 .



Esempio di cella $\langle S_0, S_1 \rangle$

▼ Esempio - Riempimento e semplificazione di una tabella

Ipotizziamo di avere la seguente tabella degli stati.

Sequenza Ingresso	Stato Presente	Stato Futuro		Uscita	
		$x = 0$	$x = 1$	$x = 0$	$x = 1$
RST	S_0	S_1	S_2	0	0
0	S_1	S_3	S_4	0	0
1	S_2	S_5	S_6	0	0
00	S_3	S_0	S_0	0	0
01	S_4	S_0	S_0	1	0
10	S_5	S_0	S_0	0	0
11	S_6	S_0	S_0	1	0

Tabella degli stati iniziale

Possiamo utilizzare il seguente procedimento:

1. definire la tabella delle implicazioni e popolare le sue celle, come descritto precedentemente, eliminando le celle che hanno stati con uscite diverse.

Ad esempio, S_0 ha uscite 00, mentre S_4 ha uscite 10, e quindi la cella definita da $\langle S_4, S_0 \rangle$ viene eliminata, poiché è definita dall'incrocio di due stati non equivalenti;

S_1-S_3					
S_2-S_4					
S_1-S_5	S_3-S_5				
S_2-S_6	S_4-S_6				
S_1-S_0	S_3-S_0	S_5-S_0			
S_2-S_0	S_4-S_0	S_6-S_0			
S_4					
S_5	S_1-S_0	S_3-S_0	S_5-S_0	S_0-S_0	
	S_2-S_0	S_4-S_0	S_6-S_0	S_0-S_0	
S_6					
	S_0	S_1	S_2	S_3	S_4
					S_5

Tabella delle implicazioni iniziale

2. per semplificare ulteriormente la tabella, si passa su ogni casella non cancellata (dall'alto verso il basso e, una volta arrivato alla fine, si passa alla colonna successiva) e si controlla che entrambe le sue coppie di stati futuri siano equivalenti. Se anche solo una delle coppie non è equivalente, allora deve essere cancellata anche quella casella.

Per esempio, partiamo dalla prima casella $\langle S_1, S_0 \rangle$, che ha come stati futuri rispettivamente S_1-S_3 e S_2-S_4 . Si nota come la casella $\langle S_1, S_3 \rangle$ non definisce che gli stati siano non equivalenti, mentre è ovvio per la casella $\langle S_2, S_4 \rangle$, che risulta avere due stati equivalenti.

Dato che basta una coppia di stati futuri non equivalenti, è necessario eliminare anche la casella $\langle S_1, S_0 \rangle$;

3. dopo aver passato tutte le celle la prima volta, l'algoritmo deve essere reiterato fino a quando la tabella resta uguale dopo un'iterazione, raggiungendo il fix-point.

Si raggiunge la conclusione che gli stati S_3 e S_5 sono equivalenti (S_1'), e lo stesso vale per S_4 e S_6 (S_4') e per S_2 e S_1 (S_3').

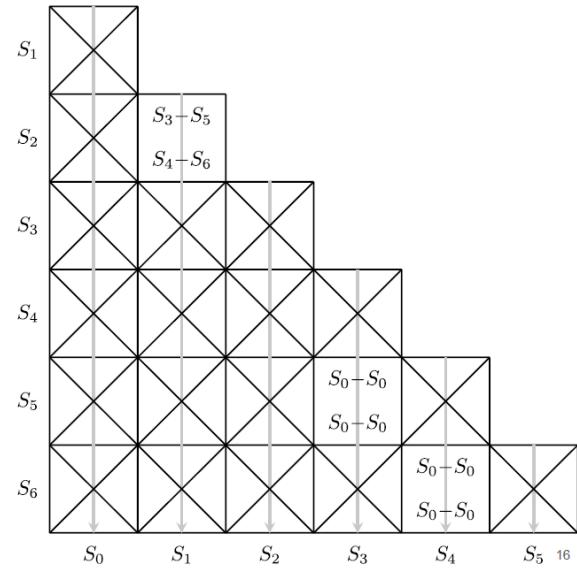


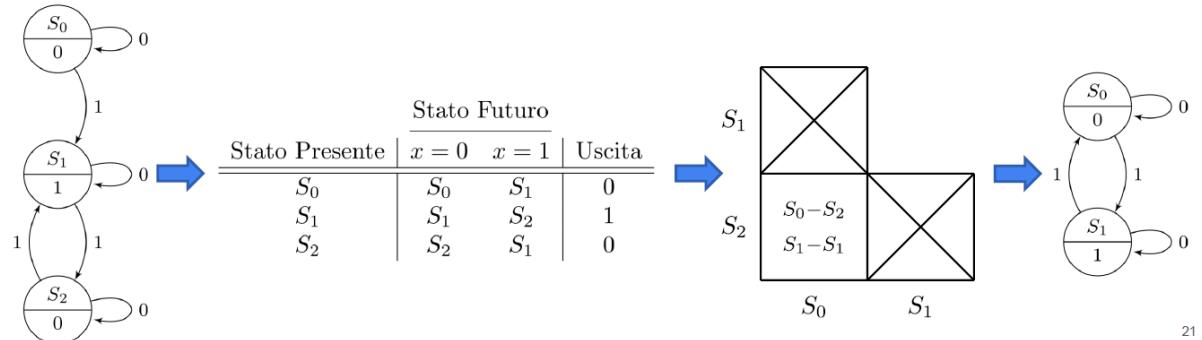
Tabella delle implicazioni al termine della semplificazione

Sequenza Ingresso	Stato Presente	Stato Futuro		Uscita	
		$x = 0$	$x = 1$	$x = 0$	$x = 1$
RST	S_0	S'_1	S'_1	0	0
0 1	S'_1	S'_3	S'_4	0	0
00 10	S'_3	S_0	S_0	0	0
01 11	S'_4	S_0	S_0	1	0

Tabella degli stati finale

▼ Esempio - Controllore di parità con tabella delle implicazioni

Applicando il metodo della tabella delle implicazioni, si nota come si arriva subito alla conclusione che S0 e S2 sono stati equivalenti.



21

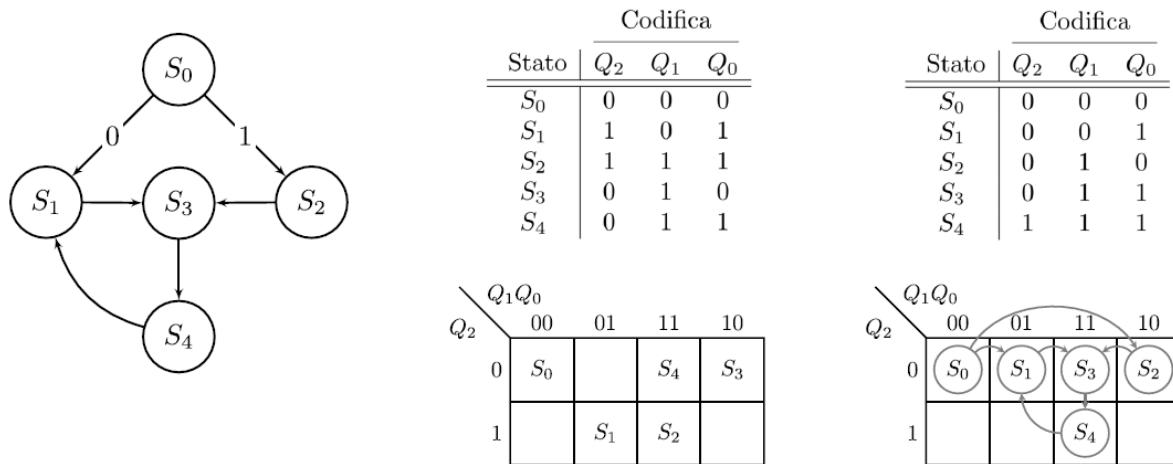
Procedimento di ottimizzazione della FSM con tabella della implicazioni

Codifica degli stati

Codifiche di stato e misura delle metriche con distanza di Hamming

Un ulteriore lavoro per l'ottimizzazione delle FSM può essere fatto scegliendo la codifica degli stati.

In questo caso, ad esempio, perché lo stato S0 deve essere codificato con 000 e non con 011 oppure qualsiasi altro codice?



Esempio di visualizzazione di una FSM

Esiste un criterio che si basa sul calcolo delle **distanze di Hamming** per definire la preferenza di una codifica degli stati rispetto ad un'altra.

Partendo dall'esempio, si vuole decidere quale delle due codifiche proposte sia migliore. Il calcolo delle distanze di Hamming si basa sulle variazioni di codice di ogni coppia di stati collegati da un arco nella FSM, attraverso i passaggi seguenti:

1. partendo dalla prima codifica, la coppia S_0 (**000**) e S_1 (**101**) ha una variazione di codice di 2 bit, ovvero cambiano il primo e il terzo bit, mentre il secondo rimane invariato. Inseriamo nella colonna Δ_1 il valore 2 che appena trovato;
2. per la seconda codifica, la coppia S_0 (**000**) e S_1 (**001**) ha una sola variazione, ovvero il terzo bit. Aggiungiamo 1 nella colonna Δ_2 .
3. svolgiamo lo stesso ragionamento per tutti gli altri stati collegati da un arco e riempiamo le due colonne;
4. la distanza di Hamming della codifica è la sommatoria di tutti i valori della colonna corrispondente. Per la prima codifica, la sommatoria dei valori di Δ_1 è 13, mentre per Δ_2 è 7.
5. la codifica migliore è quella con distanza di Hamming minore, che, nel nostro caso, risulta essere la seconda.

	Δ_1	Δ_2
$S_0 \rightarrow S_1$	2	1
$S_0 \rightarrow S_2$	3	1
$S_1 \rightarrow S_3$	3	1
$S_2 \rightarrow S_3$	2	1
$S_3 \rightarrow S_4$	1	1
$S_4 \rightarrow S_1$	2	2
Σ	13	7

Calcolo delle distanze di Hamming

Approccio basato sulla mappa di Karnaugh:

Si può utilizzare una mappa di Karnaugh per disporre gli stati in modo da evidenziare le adiacenze logiche e per trovare una codifica ottimizzata, dove le

transizioni tra stati comportano variazioni minime nei bit. Si possono eseguire i seguenti passi:

1. rappresentare gli stati della FSM come celle di una mappa di Karnaugh;
2. posizionare gli stati in modo che quelli con transizioni frequenti, collegati da archi nella FSM, siano adiacenti nella mappa, come nella [tabella di Karnaugh in basso a destra dell'esempio](#);
3. assegnare codici binari alle celle, sfruttando la proprietà della mappa di Karnaugh, per cui celle adiacenti differiscono di un solo bit;
4. dopo tutti questi passaggi, si ottiene una codifica che minimizza le variazioni di bit durante le transizioni tra stati frequentemente collegati.

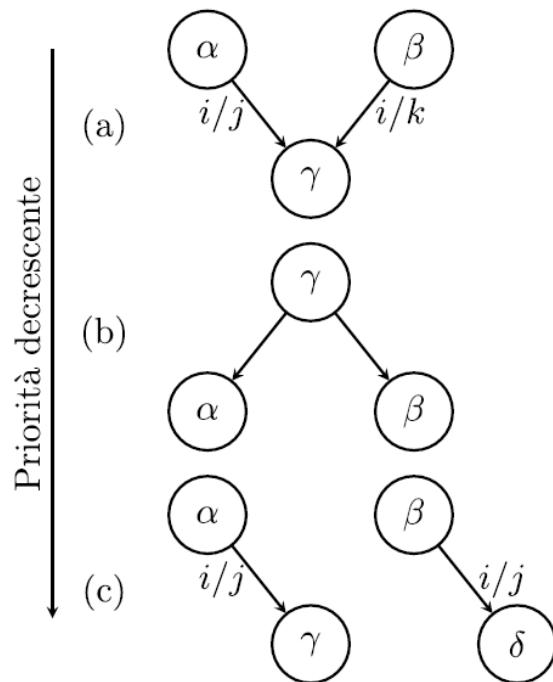
Metodi euristici

Quando le FSM diventano più complesse rispetto all'esempio precedente, si assegnano codici adiacenti privilegiando certe condizioni, con priorità differenti:

- stati con lo stesso stato futuro (a);
- stati con lo stesso stato precedente (b);
- stati con la stessa uscita (c).

La logica dietro questa scelta è che con (a) e (b) riusciamo a raggruppare gli 1 nella mappa dello stato futuro, mentre con (c) raggruppiamo gli 1 nella mappa delle uscite.

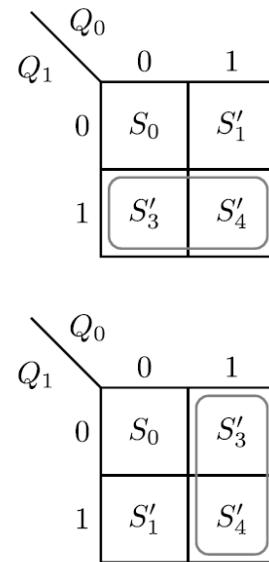
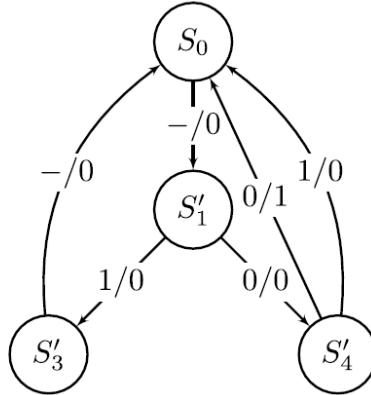
Si ricordi come avere tanti 1 vicini nelle mappe di Karnaugh ci permette di realizzare circuiti più semplici.



Esempio - Metodi euristici 1

Ipotizziamo di avere la seguente situazione:

- FSM di riferimento:
 - Identificatore di sequenza a 3 bit
- Priorità massima per lo stesso stato futuro ($S'_3 - S'_4$)
- Priorità media per lo stesso stato precedente ($S'_3 - S'_4$)
- Priorità minima per la stessa uscita
 - 0/0: (S_0, S'_1, S'_3)
 - 1/0: (S_0, S'_1, S'_3, S'_4)



Descrizione della situazione

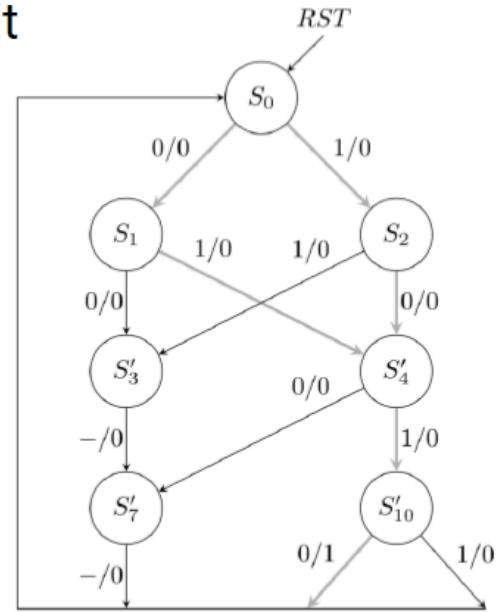
- S_3' e S_4' hanno entrambi lo stato futuro S_0 ;
- S_3' e S_4' hanno entrambi lo stesso predecessore;
- 0/0 identifica la ricezione in ingresso di uno 0 restituendo in uscita 0, mentre 1/0 identifica la ricezione in ingresso di un 1 e restituendo in uscita 0.

Mettiamo S_0 come 00, in modo da utilizzarlo come una sorta di reset, e ci rimangono due scelte per posizionare S_3' e S_4' vicini. Nell'esempio, le due possibili mappe di Karnaugh sono [rappresentate nel lato destro](#).

Esempio - Metodi euristici 2

Ipotizziamo di avere la seguente situazione:

- FSM di riferimento:
riconoscitore di stringa a 4 bit
- Priorità *massima* per lo stesso stato *futuro*
 - $(S'_3 - S'_4), (S'_7 - S'_{10})$
- Priorità *media* per lo stesso stato *precedente*
 - $(S_1 - S_2), (S'_3 - S'_4), (S'_7 - S'_{10})$
- Priorità *minima* per la stessa uscita
 - 0/0: $(S_0, S_1, S_2, S'_3, S'_4, S'_7)$
 - 1/0: $(S_0, S_1, S_2, S'_3, S'_4, S'_7)$



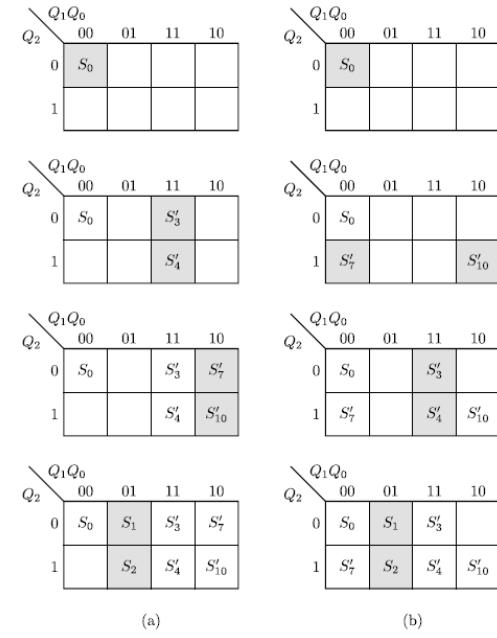
Descrizione della situazione

Si noti come abbiamo raggruppato le coppie di stati che dovremmo mettere vicine nella FSM con priorità decrescente.

Descriviamo la soluzione dell'esempio:

- utilizziamo S_0 codificato con 000 come RST;
- nell'immagine a destra, sono presenti due esempi lato a lato di riempimento differenti, inserendo prima le coppie di stati con priorità massima e successivamente quelli con priorità minimi;
- una codifica rimane senza stato, perciò questa cella potrebbe essere usata in modo strategico, in modo da avere valori "don't care" nelle mappe di Karnaugh degli stati futuri e delle uscite, in modo da semplificare ulteriormente il circuito.

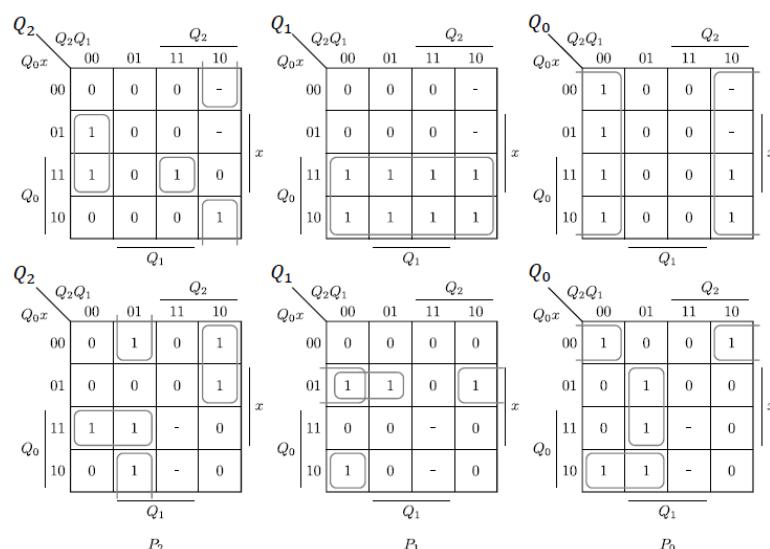
Passando dalla nostra codifica degli stati alle relative mappe di Karnaugh dell'uscita e degli stati futuri, si nota che la prima codifica raggruppa meglio i valori 1 nella mappa dello stato futuro.



Esempi di riempimento delle mappe di Karnaugh lato a lato

Stato Presente	Stato Futuro	
	$x = 0$	$x = 1$
(S_0)	000	001 101
(S_1)	001	011 111
(S_2)	101	111 011
(S'_3)	011	010 010
(S'_4)	111	010 110
(S'_7)	010	000 000
(S'_{10})	110	000 000

Stato Presente	Stato Futuro	
	$x = 0$	$x = 1$
(S_0)	000	001 010
(S_1)	001	011 100
(S_2)	010	100 011
(S'_3)	011	101 101
(S'_4)	100	101 110
(S'_7)	101	000 000
(S'_{10})	110	000 000



Confronto delle soluzioni ottenute dai due diversi riempimenti



Circuiti Aritmetici

▼ Creatore originale: @Samuele Gentile

Ripasso di numerazione binaria

[Modulo e segno](#)

[Complemento a 1 \(CA1\)](#)

[Complemento a 2 \(CA2\)](#)

[Half Adder \(semisommatore\)](#)

[Possibili implementazioni](#)

[Full Adder \(sommatore\)](#)

[Possibili implementazioni](#)

[Adder Ripple Carry](#)

[Temporizzazione](#)

[\(*\) Addizionatore/sottrattore binario](#)

[Circuito di Carry Lookahead \(CLA\)](#)

[Carry Lookahead Cell \(CLC\)](#)

[Implementazione del CLA](#)

[Temporizzazione](#)

[Ripple CLA](#)

[CLA gerarchico](#)

[Moltiplicatore parallelo](#)

[Altre funzioni aritmetiche](#)

[Esempio - Semplificazione di un sommatore Ripple Carry](#)

Ripasso di numerazione binaria

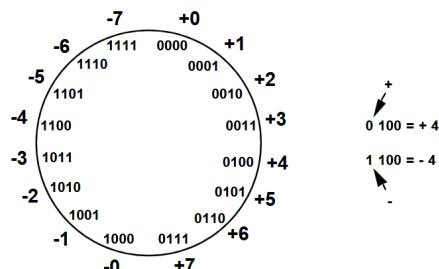
Modulo e segno

Nella rappresentazione Modulo e Segno (M&S), il bit più significativo è il segno, con 0 che indica il positivo e 1 che indica il negativo.

I restanti bit indicano l'effettivo valore numerico, chiamato modulo.

In questa rappresentazione:

- lo 0 è rappresentato con due valori binari diversi;
- somma e sottrazione sono complesse perché bisogna confrontare i moduli per decidere il segno del risultato.



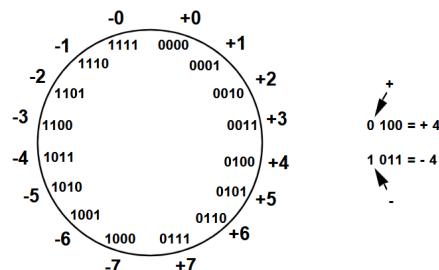
Rappresentazione dei valori M&S a 4 bit

Complemento a 1 (CA1)

Nella rappresentazione a complemento a 1 (CA1), si cambiano tutti i valori al loro complemento ($0 \rightarrow 1, 1 \rightarrow 0$).

In questa rappresentazione:

- lo 0 è rappresentato con due valori binari diversi;
- se abbiamo un numero positivo, il suo opposto sarà ottenuto tramite il complemento ad 1, e viceversa;
- la sottrazione è semplice da implementare, poiché prima faccio il complemento a 1 del sottraendo, e poi sommo;
- la somma è più complessa per i casi in cui bisogna gestire l'overflow.



Rappresentazione dei valori CA1 a 4 bit

Complemento a 2 (CA2)

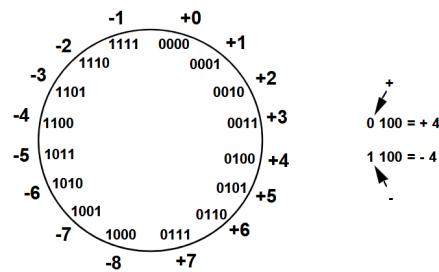
Nella rappresentazione a complemento a 2 (CA2), si esegue il CA1, per poi sommare 1 in binario.

Per esempio, trasformiamo in CA2 0111₂:

$$0111_{\text{CA1}} + 1_2 = 1000_{\text{CA1}} + 1_2 = 1001_{\text{CA2}}$$

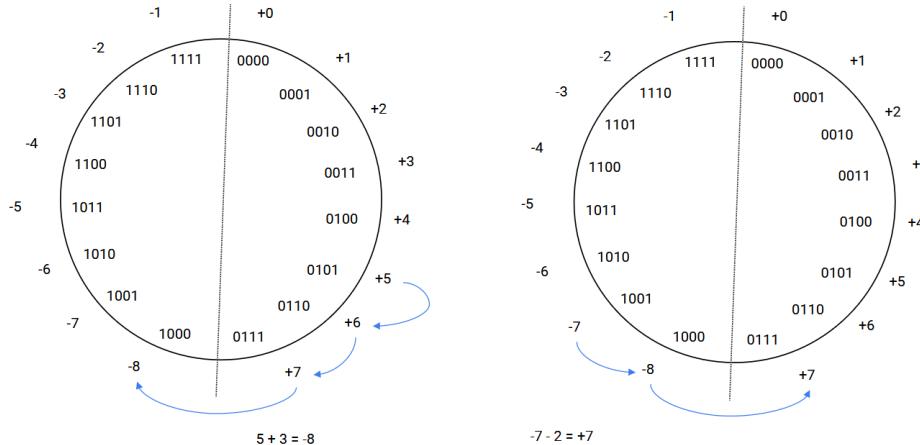
In questa rappresentazione:

- lo 0 è rappresentato con un unico valore binario;
- rispetto alle altre rappresentazioni, il range di numeri che copriamo con gli stessi bit è aumentato di uno, poiché abbiamo una sola rappresentazione per 0;
- la condizione di overflow si verifica quando, sommando due numeri entrambi positivi o entrambi negativi, otteniamo un numero di segno opposto rispetto agli addendi.



Rappresentazione dei valori CA2 a 4 bit

- esiste un metodo abbastanza semplice per verificare se è avvenuto overflow, ovvero controllare che il riporto sul segno e il riporto in uscita siano uguali.



Esempi di overflow

$$\begin{array}{r}
 5 \quad 0111 \\
 3 \quad 0011 \\
 \hline -8 \quad 1000
 \end{array}
 \quad
 \begin{array}{r}
 -7 \quad 1000 \\
 -2 \quad 1110 \\
 \hline 7 \quad 10111
 \end{array}
 \quad
 \begin{array}{r}
 5 \quad 0000 \\
 2 \quad 0010 \\
 \hline 7 \quad 0111
 \end{array}
 \quad
 \begin{array}{r}
 -3 \quad 1111 \\
 -5 \quad 1011 \\
 \hline -8 \quad 11000
 \end{array}$$

Overflow

Overflow

Non è overflow

Non è overflow

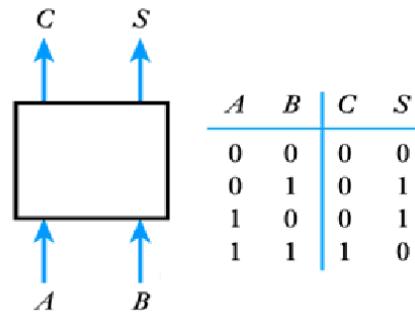
Esempi di calcolo di overflow e di non overflow

Half Adder (semisommatore)

Nel circuito Half Adder (HA), definiamo A e B come i due bit che vogliamo sommare, S come il bit di somma e C come il bit di riporto.

Si noti che:

- S è realizzabile con una porta XOR;
- C è realizzabile con una porta AND.

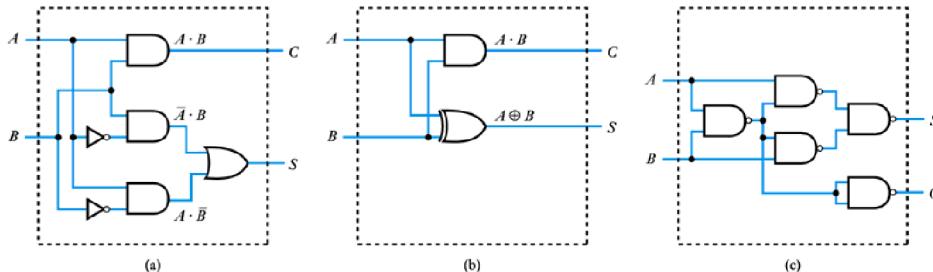


Visualizzazione di un circuito Half Adder

Possibili implementazioni

Di seguito sono riportate alcune implementazioni possibili:

- si realizza con porte AND, OR e NOT;
- si realizza con una porta AND e XOR;
- si realizza solo con porte NAND.



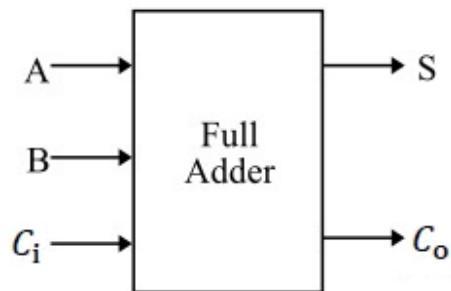
Possibili implementazioni di un circuito Half Adder

Full Adder (sommatore)

Nel circuito Full Adder (FA), definiamo A e B come i due bit che vogliamo sommare, S come il bit di somma, C_i come il riporto in ingresso (carry-in) e C_o come il riporto in uscita (carry-out).

$$S = A \oplus B \oplus C_i$$

$$\begin{aligned} C_o &= A \cdot B + A \cdot C_i + B \cdot C_i \\ &= A \cdot B + C_i \cdot (A + B) \end{aligned}$$



Visualizzazione di un circuito Full Adder

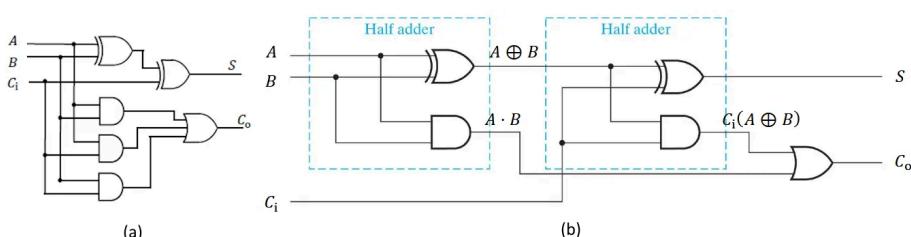
Si noti, quindi, che un generico sommatore deve gestire in ingresso anche un riporto proveniente dal sommatore precedente.

Possibili implementazioni

Di seguito sono riportate alcune implementazioni possibili:

- a. si realizza con porte AND, OR e XOR;
- b. cascata di due [Half Adder](#), in cui notiamo che l'unica differenza è in C_o , che adesso ha uno XOR $A \oplus B$ e non un semplice OR. Non è una modifica che possiamo fare sempre, ma in questo caso specifico sì.

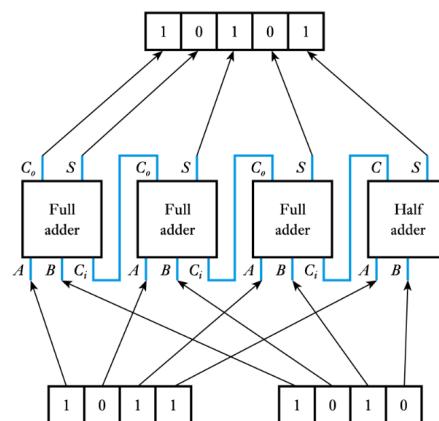
Il caso in cui lo XOR si comporta diversamente dall'OR è quando A e B sono entrambi al valore 1, ma in quel caso il loro AND sommato a qualsiasi valore binario restituisce sempre 1.



Possibili implementazioni di una rete Half Adder

Adder Ripple Carry

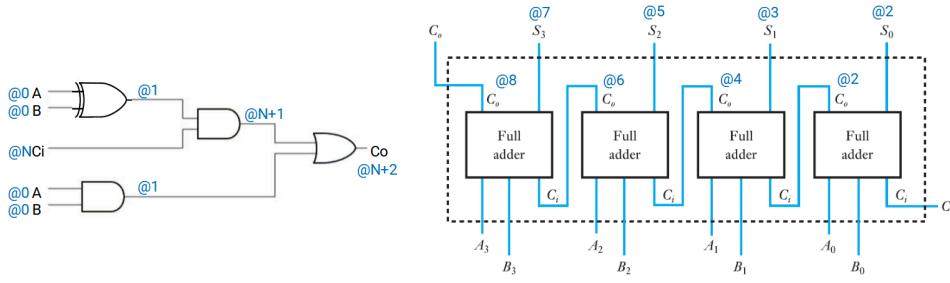
Nel sommatore multibit, chiamato più comunemente Adder Ripple Carry vediamo come esiste una propagazione dei riporti per determinare l'ultimo riporto di uscita. Questo comporta ritardi ragionevoli se la catena di Full Adder dovesse diventare troppo lunga.



Esempio di sommatore di parole a 4 bit

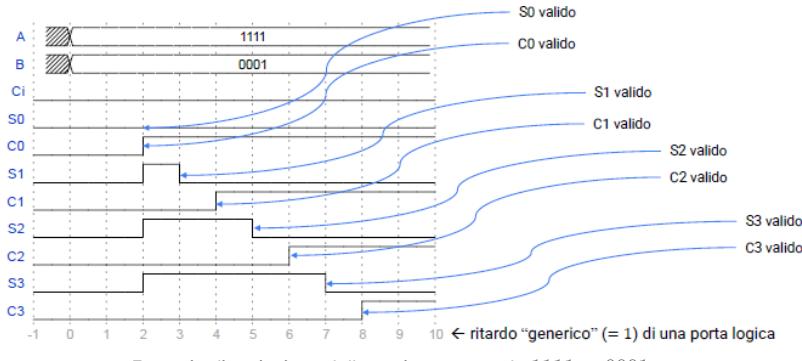
Temporizzazione

Se indichiamo con la notazione $@X$ il tempo di arrivo di un certo dato in un certo punto del circuito in esame e assumiamo che ogni elemento introduca un ritardo unitario, otteniamo le uscite del sommatore con i ritardi indicati in figura.



Esempio di temporizzazione per Adder Ripple Carry

Studiamo come variano le uscite di un Adder Ripple Carry nel caso in cui volessimo sommare 1111_2 e 0001_2 . Questa è la situazione peggiore, poiché tutti gli output intermedi devono assestarsi prima che sia valido il MSB.

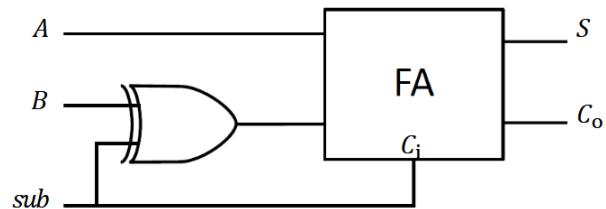


Esempio di variazione delle uscite sommando 1111_2 e 0001_2

! S_1, S_2 e S_3 hanno l'uscita a 1 per un breve periodo di tempo, in modo non corretto, questo perché non hanno ancora ricevuto il carry-in a causa dei ritardi di propagazione.

(*) Addizionatore/sottrattore binario

Possiamo realizzare un sommatore/sottrattore tramite un FA e una porta XOR, e quest'ultima serve per selezionare il comportamento atteso e gestire il valore del secondo addendo.



Esempio di addizionatore/sottrattore binario

In base al valore di sub:

- se $\text{sub} = 0$, lo XOR trasmette B al FA e calcola $A_2 + B_2$ con $C_i = 0$;
- se $\text{sub} = 1$, esegue la differenza, poiché lo XOR trasmette il CA1 di B e il FA esegue la somma $S_2 = A_2 + B_{\text{CA1}} + 1_2$, ma sappiamo che $B_{\text{CA1}} + 1_2 = B_{\text{CA2}}$, e quindi, in CA2, si ha una sottrazione.

Circuito di Carry Lookahead (CLA)

Il circuito di carry lookahead (CLA) serve per minimizzare il ritardo di propagazione dei riporti in un sommatore. Per realizzare questo comportamento, si introducono due espressioni:

- Carry Generate G_i :

$$G_i = A_i \cdot B_i$$

- Carry Propagate P_i :

$$P_i = A_i \oplus B_i$$

La cosa importante è che sia G che P dipendano solo dagli ingressi A e B , non dai valori dei riporti. Possiamo riscrivere le uscite somma e riporto in funzione di G_i e P_i .

$$\begin{aligned} S_i &= A_i \oplus B_i \oplus C_i \\ &= P_i \oplus C_i \end{aligned}$$

$$\begin{aligned} C_{i+1} &= A_i B_i + A_i C_i + B_i C_i \\ &= A_i B_i + C_i (A_i \oplus B_i) \\ &= G_i + C_i P_i \end{aligned}$$

Definiamo ora C_i .

$$C_1 = G_0 + P_0 C_0 \quad (1)$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0 \quad (2)$$

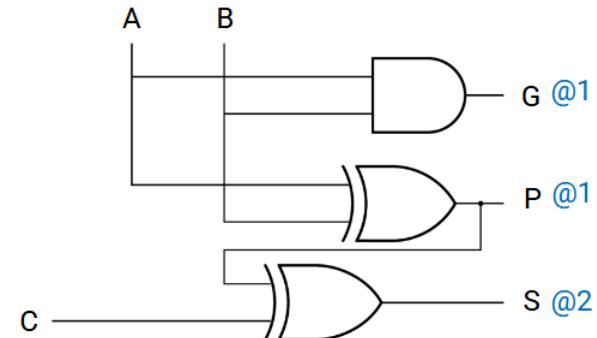
$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0 \quad (3)$$

$$\dots \quad (4)$$

$$C_i = \sum_{j=-1}^{i-1} G_j \prod_{k=j+1}^{i-1} P_k, \text{ con } (G_{-1} = C_0) \quad (5)$$

Carry Lookahead Cell (CLC)

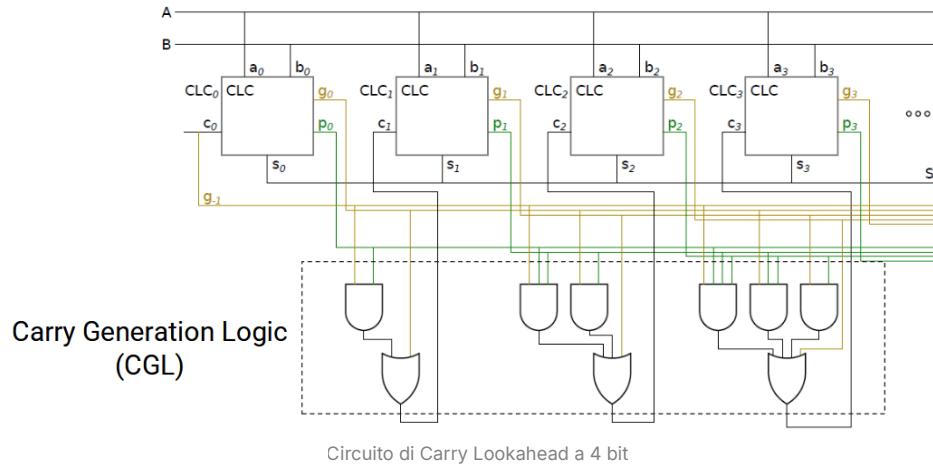
Il Carry Lookahead Cell (CLC), simile ad un Full Adder, ma genera in uscita i segnali G e P e non il carry out.



Esempio di implementazione di CLC

Implementazione del CLA

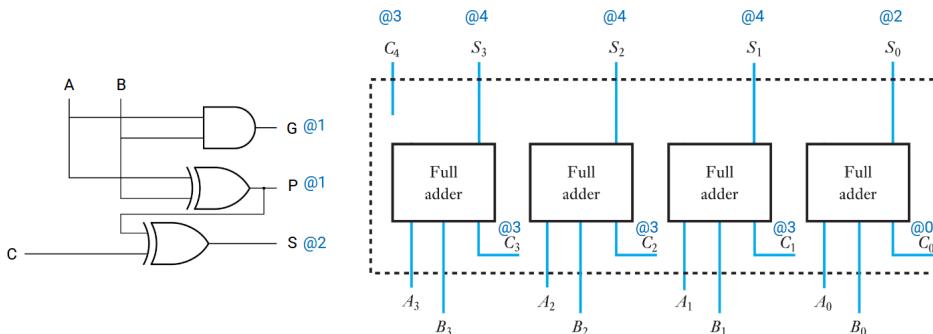
Le variazioni in ingresso sono solamente gli addendi e il riporto in ingresso allo stadio zero. Questo discorso si traduce nel circuito in [figura](#).



A partire da sinistra verso destra vediamo la realizzazione dei vari Carry (Carry Generation Logic), dove il primo blocco implementa C_1 , il secondo blocco implementa C_2 , e così via. Man mano che le porte logiche nei carry lookahead aumentano (ne bastano anche già 4) il blocco diventa lento nel calcolo del carry_out, di conseguenza si cerca di realizzare Carry Lookahead non troppo grossi.

Temporizzazione

Studiando i ritardi per un carry-lookahead, notiamo che il cammino più lungo è praticamente costante, poiché tutti i carry in (tranne il primo) arrivano contemporaneamente ai Full Adder.

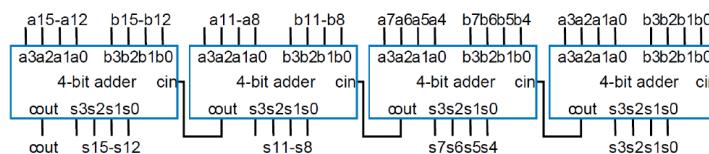


Esempio di calcolo della temporizzazione di un CLA a 4 bit

Ripple CLA

Come abbiamo visto, al crescere del numero di stadi cresce il numero di gate e il rispettivo fan-in. Questo porta ad avere un fan-in troppo elevato e richiede di essere sostituito da una struttura a più livelli, sfruttando porte più piccole.

Una soluzione possibile è quella di collegare in cascata più CLA di dimensione più piccola, dove ogni CLA aspetta il carry in del CLA precedente.

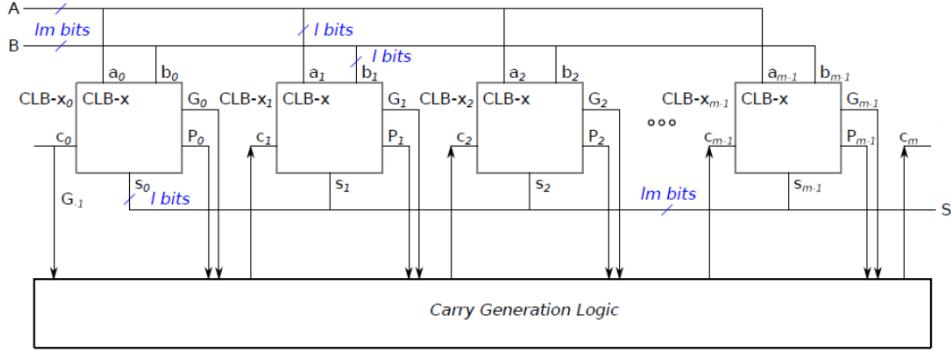


Esempio con 4 CLA da 4 bit

CLA gerarchico

Suddividiamo il parallelismo in ingresso in m parallelismi di l bit. Gli l bit vengono gestiti ognuno da un Carry Lookahead Block (CLB), che può includere qualsiasi tipo di sommatore:

- CLB-r (ripple);
- CLB-c (carry lookahead).



Esempio di CLA gerarchico a 4 blocchi

Moltiplicatore parallelo

Una moltiplicazione binaria per un numero senza segno si può fare come quella in base decimale, esaminando i bit da destra a sinistra:

- se un bit è a 1, si aggiunge una versione opportunamente traslata del moltiplicando a un prodotto parziale;
- se invece il bit è a 0 non fornisce contributo.

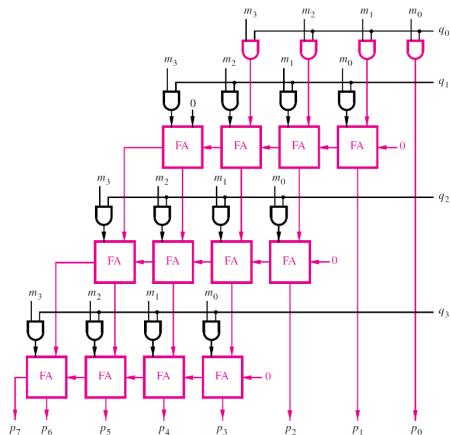
$ \begin{array}{r} M(14_{10}) \\ \times 1011 \\ \hline 1110 \\ 1110 \\ 0000 \\ \hline 1110 \\ + 0000 \\ \hline 01010 \\ + 1110 \\ \hline 10011010 \end{array} $	$ \begin{array}{r} 1110 \\ \times 1011 \\ \hline 1110 \\ + 1110 \\ \hline 01011 \end{array} $	<p>prodotto parziale 0</p>	$ \begin{array}{r} m_3 \quad m_2 \quad m_1 \quad m_0 \\ \times q_3 \quad q_2 \quad q_1 \quad q_0 \\ \hline m_3q_0 \quad m_2q_0 \quad m_1q_0 \quad m_0q_0 \\ + m_3q_1 \quad m_2q_1 \quad m_1q_1 \quad m_0q_1 \\ \hline PP1_5 \quad PP1_4 \quad PP1_3 \quad PP1_2 \quad PP1_1 \end{array} $
		prodotto parziale 1	
		prodotto parziale 2	
			$ \begin{array}{r} m_3q_2 \quad m_2q_2 \quad m_1q_2 \quad m_0q_2 \\ + m_3q_3 \quad m_2q_3 \quad m_1q_3 \quad m_0q_3 \\ \hline PP2_6 \quad PP2_5 \quad PP2_4 \quad PP2_3 \quad PP2_2 \end{array} $
			$ \begin{array}{r} p_7 \quad p_6 \quad p_5 \quad p_4 \quad p_3 \quad p_2 \quad p_1 \quad p_0 \end{array} $

Esempio di moltiplicazione binaria

Per evitare di introdurre sommatori con un numero non costante di ingressi, dobbiamo implementare prodotti e somme parziali, come in [figura](#), e con i seguenti componenti:

- 16 porte AND per calcolare i prodotti bit-a-bit;
- 12 sommatori per la somma dei prodotti parziali (Carry-In nullo).

Per le somme di ordine più elevato si sfruttano più Carry-Out in parallelo.



Implementazione di un moltiplicatore parallelo

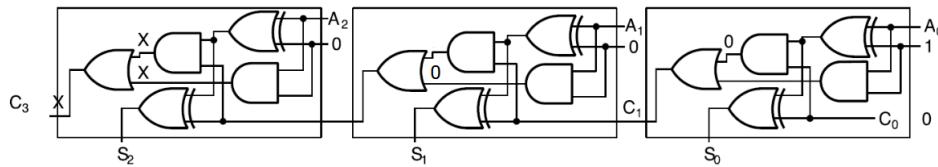
Altre funzioni aritmetiche

Molte volte conviene semplificare dei blocchi già esistenti rimuovendo una parte di circuiteria ridondante quando alcuni input sono costanti:

- incremento/decremento, fissando ad una costante uno dei due operandi;
- moltiplicazione/divisione per una costante;
- estensione del segno e zero-fill, con cui vado a riempire di 1 o 0 tutti i nuovi bit che sono stati aggiunti prima se, ad esempio, cambio la lunghezza di una parola.

Esempio - Semplificazione di un sommatore Ripple Carry

Cerchiamo di semplificare un sommatore Ripple Carry, il quale fa un incremento unitario.



Circuito iniziale del sommatore Ripple Carry di esempio

Prendiamo, per esempio, il blocco di porte logiche più a sinistra, definito dagli ingressi A_0 e 1 e dalle uscite S_0 e C_1 . Si può cercare di descrivere il comportamento della porta logica in funzione degli ingressi.

Cerchiamo di definire la tabella in funzione dell'ingresso A_0 e 1, utilizzando la funzione logica definita dal circuito iniziale:

$$\begin{cases} S_0 = (A_0 \oplus 1) \oplus C_0 \\ C_1 = [C_0 \cdot (A_0 \oplus 1)] + [A_0 \cdot 1] \end{cases}$$

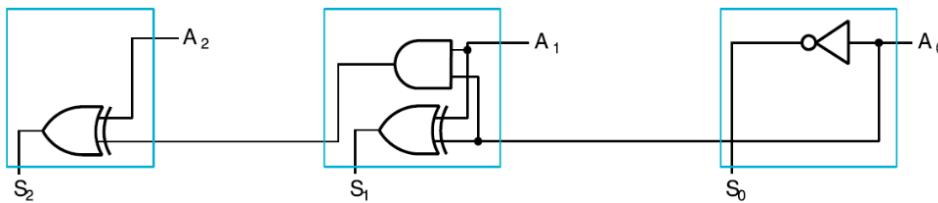
A_0	C_0	S_0
0	0	1
1	0	0

Si può notare, attraverso la tabella creata, che C_0 vale sempre 0, poiché il carry della prima cifra, il quale non ha alcuna operazione che lo definisce, vale sempre 0. In questo caso, si ha che C_0 non influisce in alcun modo sul risultato finale, quindi si può omettere.

A_0	S_0
0	1
1	0

Successivamente, si può notare come $S_0 = \overline{A}_0$ e $C_1 = A_0$ attraverso l'ultima tabella definita, perciò si ha, nel circuito, che l'uscita S_0 è definita attraverso la negazione dell'ingresso A_0 , mentre l'uscita C_1 è la forma diretta di A_0 .

Si ripete lo stesso ragionamento per ognuno dei tre blocchi di porte logiche, stando attenti al valore dei rispettivi carry che, al contrario di C_0 , possono avere sia valore 0, sia valore 1, e si ottiene il seguente schema:



Semplificazione del sommatore Ripple Carry



Unità di controllo e datapath

▼ Creatore originale: @Samuele Gentile

Controllore e datapath

Definizioni

Sistemi di controllo aperto

Sistemi a controllo retroazionato

Design di sistemi con controllore e datapath

Esempio - Bit counter

Definizione del circuito

Analisi del datapath

Analisi del controllore

Esempio - Moltiplicatore binario SHIFT-ADD

Analisi del circuito

Analisi del controllore

Controllore e datapath

Fino ad ora, ci siamo concentrati sulla realizzazione di unità dedicate ad implementare funzioni specifiche. In generale, però, nei sistemi digitali queste unità sono interconnesse e comandate tramite opportuni segnali dedicati alla selezione dell'operazione da svolgere e su quali valori in ingresso.

Le operazioni possono essere selezionate e svolte sulla base di ulteriori criteri:

- stato del sistema;
- temporizzazione;
- altre condizioni.

Inoltre, nonostante a livello teorico una FSM possa rappresentare un gran numero di sistemi, ci sono casi in cui possono risultare troppo complessi per essere modellati efficacemente. Per esempio:

- un registro a scorrimento a 8 bit è descrivibile con una FSM con 256 stati differenti;
- un singolo registro a 32 bit definisce uno spazio degli stati con 2^{32} stati differenti.

Definizioni

Si utilizzano, quindi, due parti importanti nel circuito:

- controllore: è un'unità logica che, basandosi su ingressi e sul proprio stato interno, genera segnali di controllo per guidare il comportamento di un sistema digitale.

È spesso realizzato come una macchina a stati finiti (FSM) e determina quali operazioni devono essere eseguite e **in** quale ordine, coordinando i vari componenti di un sistema (ad esempio ALU, registri, bus, memorie, ecc.).

Un esempio pratico è la Control-Unit di una CPU, che coordina il ciclo di fetch-decode-execute.

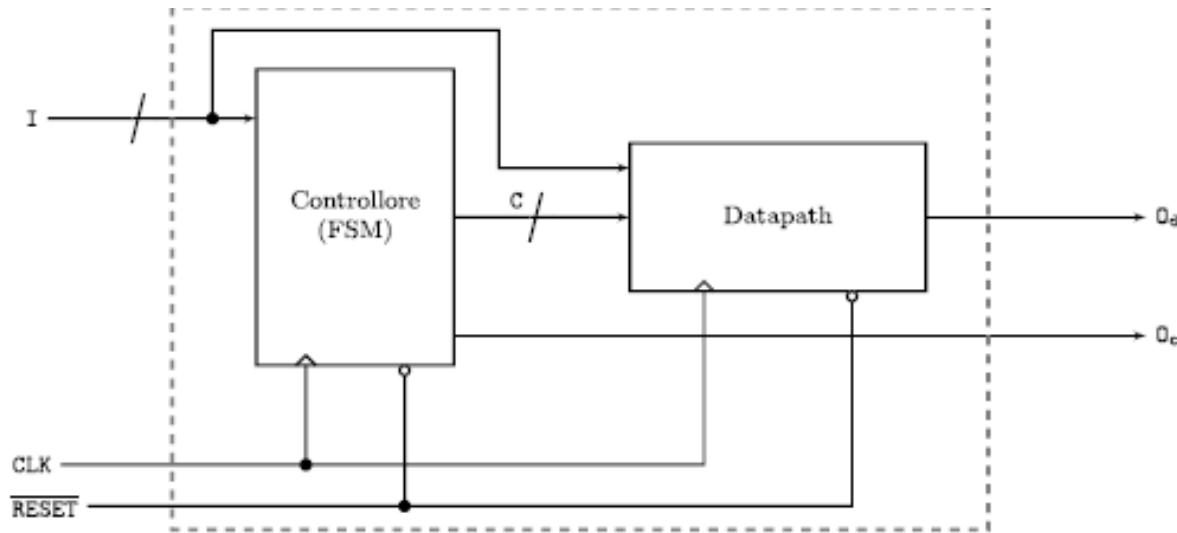
- **datapath**: parte del circuito che si occupa dell'elaborazione e del trasporto dei dati.

Sistemi di controllo aperto

Con controllo aperto, il controllore invia comandi e segnali al datapath senza ricevere nessuna retroazione.

Le uscite del sistema possono essere generate sia dal controllore, sia dal datapath.

Gli ingressi del sistema possono interessare entrambi i componenti.

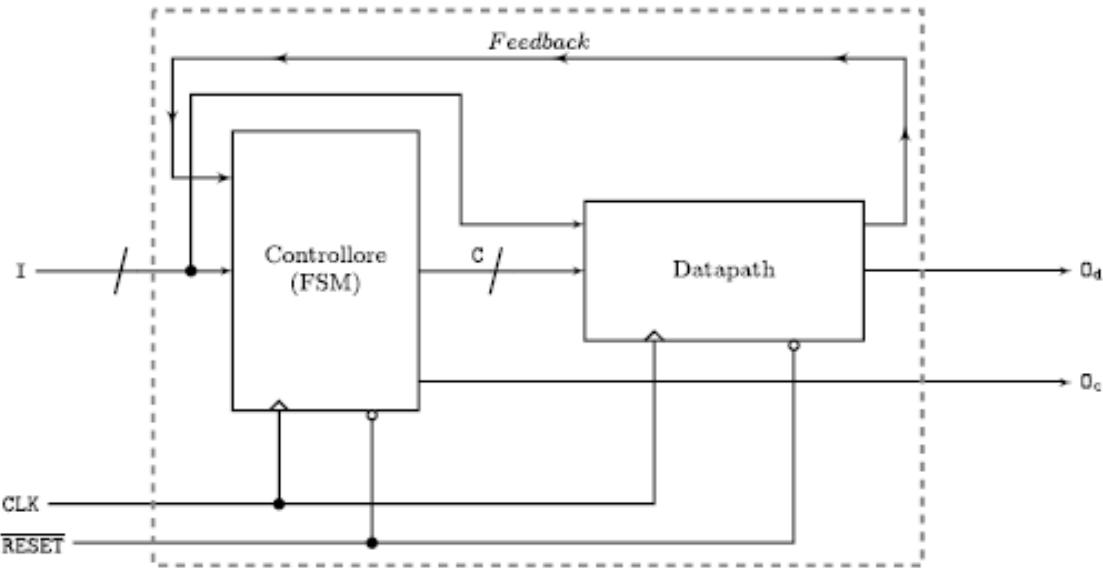


Esempio di sistema di controllo aperto

Sistemi a controllo retroazionato

Con controllo retroazionato, il controllore riceve informazioni di ritorno dal datapath ai fini del controllo dell'operato del sistema, in modo da poterlo usare come dato durante le successive fasi dell'algoritmo.

Per esempio, se dobbiamo fare una serie di shift fino ad esaurire il contenuto di un registro, il controllore imporrà di eseguire uno shift fino a quando il datapath non manda un segnale per dire che il registro è stato svuotato completamente.



Esempio di sistema a controllo retroazionato

Design di sistemi con controllore e datapath

Per creare un design di sistemi con controllore e datapath il procedimento è come segue:

1. descrivere le funzionalità del sistema da modellare;
2. determinare quali componenti del datapath siano necessari (registri, ALU, MUX, DEMUX ecc...);
3. definire le interconnessioni tra elementi del datapath;
4. identificare i segnali di input e output del controllore;
5. definire la sequenza di segnali che il controllore deve generare perché il sistema presenti il comportamento atteso;
6. definire la FSM adatta a rappresentare il controllore necessario;
7. verifica, manuale o automatica, del sistema complessivo. In caso di errori e/o incongruenze, rivedere i passi da [\(2\)](#) a seguire;
8. implementazione e test del sistema definitivo. In caso di errori e/o incongruenze rivedere i passi da [\(2\)](#) a seguire.

Esempio - Bit counter

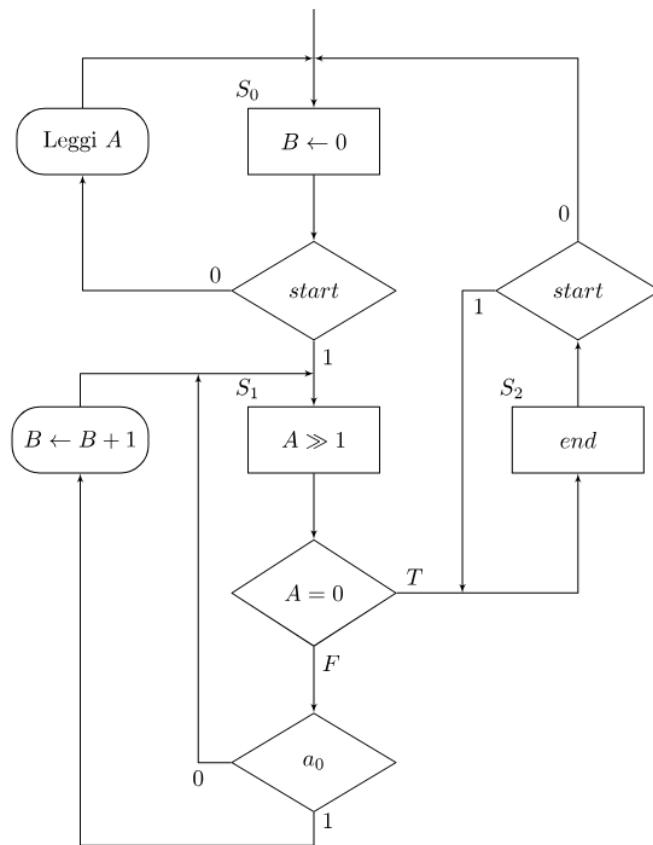
L'obiettivo è partire da un registro A e contare quanti bit sono ad 1.

1. Si inizializza un registro B a zero;
2. Con un ciclo while si va a prendere il bit meno significativo di A, fino a quando A non è uguale a 0, mentre se è 1 incrementiamo B;
3. Per ogni iterazione del ciclo, si shifta il valore del registro A a destra.

$$B \leftarrow 0$$

```
while  $A \neq 0$  do
    if  $a_0 = 1$  then
         $B \leftarrow B + 1$ 
     $A \gg 1$ 
```

Pseudocodice dell'algoritmo



Visualizzazione della ASM

Definizione del circuito

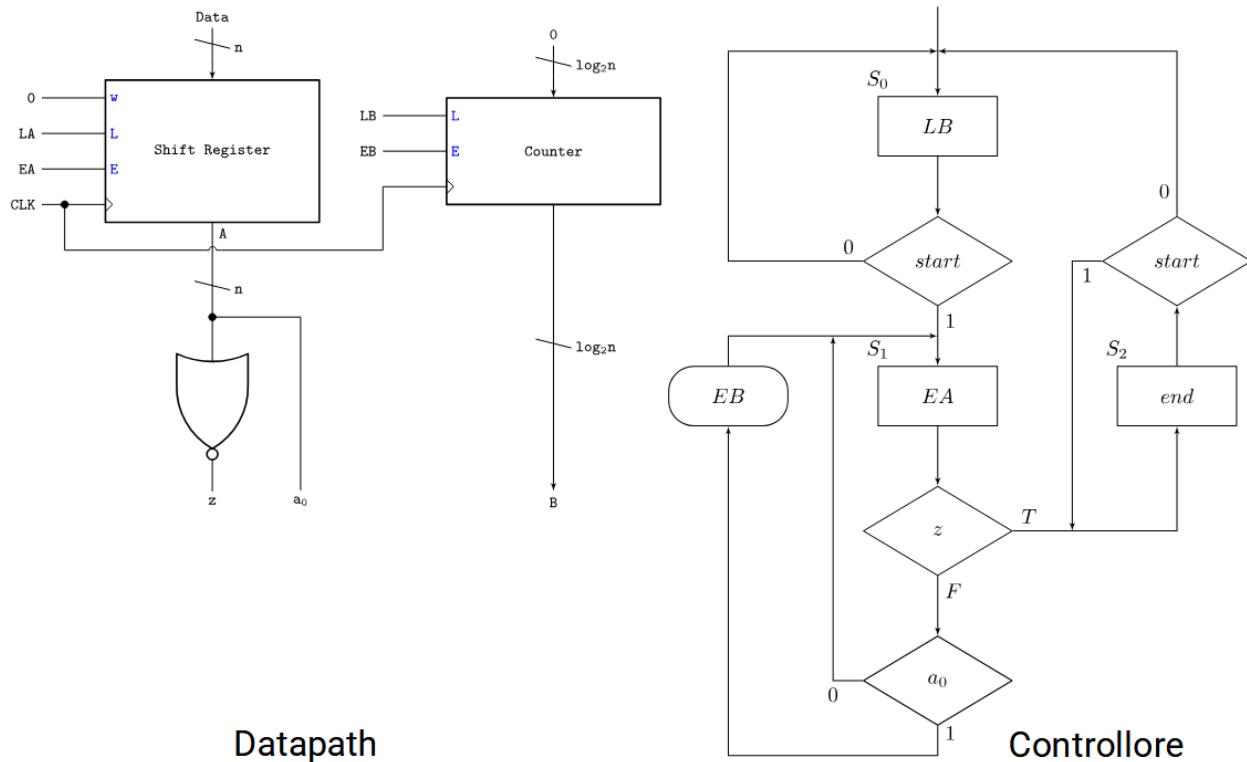
Nello pseudocodice e nel grafico a blocchi:

- la freccia puntata verso sinistra è un'**assegnazione**, mentre il simbolo di uguale è una **verifica di uguaglianza** (come `==` in C);
- esiste una condizione di **start** che inizia ad operare quando gli è stato detto di farlo, e finché non usciamo dallo start si rimane in attesa (nel loop superiore);

- una volta che la condizione di start è a 1, la prima operazione che incontriamo è lo shift di A, ma ricordiamo che il risultato dell'operazione di shift sarà "visibile" solo al colpo di clock successivo, ovvero quando il valore del registro verrà aggiornato.

Dallo studio dell'ASM, capiamo che le operazioni che dobbiamo implementare nel datapath sono definite da uno Shift-Register ($A \gg 1$), da un contatore per B ($B \leftarrow B + 1$) e da due blocchi di condizione:

- uno per verificare se A vale 0 ($A \neq 0$);
- uno per verificare se a_0 vale 0 ($a_0 = 1$), sapendo che può essere solo un valore binario.



Analisi del datapath

Lo **Shift-Register**:

- riceve in ingresso **Data** gli n bit iniziali di A;

- l'ingresso **W** è impostato a 0_2 serve per inserire un nuovo bit, il quale verrà inserito quando shiftiamo il registro;
- il segnale **LA** è il segnale che serve per abilitare l'ingresso dei bit di Data;
- il segnale **EA** è il segnale di enable, necessario per abilitare lo shift register, in modo che ogni volta che arriva un colpo di clock avvenga effettivamente l'operazione di shift.

Il Counter:

- riceve in ingresso il valore formato da $\log_2 n$ cifre da cui partire, definito solo da 0;
- il parallelismo è solo $\lfloor \log_2 n + 1 \rfloor$, poiché se, ad esempio, ho un registro A in ingresso di 16 bit, il suo numero di bit a 1 può essere al massimo 16, portando a una codifica in binario di 5 bit;
- Il segnale **LB** (Load B) serve per prendere il valore iniziale di **B**;
- Il segnale **EB** è il segnale che, se attivato ogni colpo di clock, incrementa **B** di 1.

La porta **NOR** con, in ingresso, l'uscita A dello Shift-Register, permette di implementare la funzione di confronto di A con il valore 0.

Analisi del controllore

Dato che il controllore è la parte di circuito che deve preoccuparsi della gestione dei vari segnali in ingresso, ci sarà un primo momento in cui **LA** e **LB** dovranno essere alti, in modo da impostare i dati al valore iniziale e, successivamente, dovranno essere impostati al valore basso per evitare che i dati vengano reimpostati al valore iniziale.

Il controllore deriva dall'ASM, dai suoi stati e dalle sue condizioni, come descritto in seguito:

- nel blocco della ASM, dove avevamo $B \leftarrow 0$, ora abbiamo il relativo segnale che svolge il compito nel counter, ovvero **LB**;
- $A \gg 1$ diventa **EA**;
- $B \leftarrow B + 1$ diventa **EB**;

- la prima condizione $A = 0$ viene definita dal segnale **z**, ottenuto dall'uscita della porta NOR;
 - la seconda condizione viene definita dal segnale **a0**;
 - si assume che il caricamento e la gestione del valore di A siano fatti da circuiteria esterna.

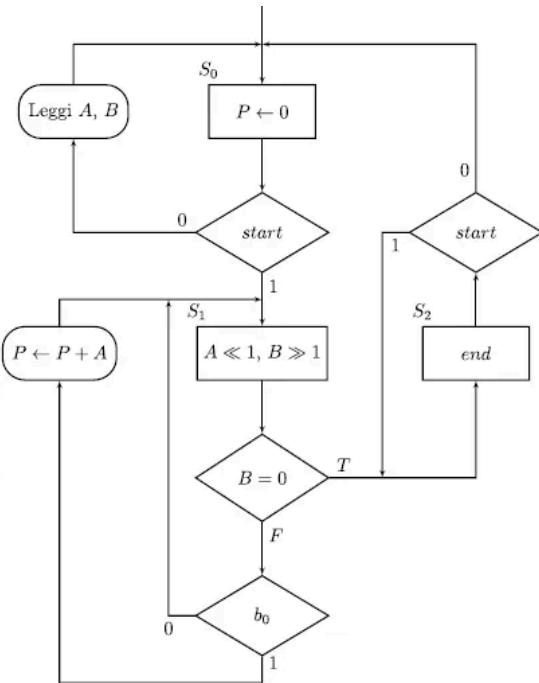
Esempio - Moltiplicatore binario SHIFT-ADD

In questo esempio, vedremo come implementare un moltiplicatore binario sfruttando il paradigma controller/datapath.

L'algoritmo è lo stesso visto quando si è trattato il moltiplicatore parallelo.

$P \leftarrow 0$	\times	1	1	0	1	$\leftarrow A$
for $i \leftarrow 0$ to $n - 1$ do		1	0	1	1	$\leftarrow B$
if $b_0 = 1$ then		1	1	0	1	
$P \leftarrow P + A$		1	1	0	1	
$A \ll 1$	0	0	0	0		
$B \gg 1$	1	1	0	1		
	1	0	0	0	1	1
	1	0	0	0	1	1

Pseudocodice dell'algoritmo



Visualizzazione della ASM

Analisi del circuito

Il circuito è molto simile a quello implementato nel [primo esempio](#):

- due **Shift-Register**, uno con scorrimento a sinistra (Shift-Left Register, per A) e uno con scorrimento a destra (Shift-Right Register, per B), con possibilità di caricare dati in parallelo e con segnale di abilitazione;
 - un **registro** con enable per il valore di P, permettendo il mantenimento delle somme parziali;
 - un **sommatore** per calcolare il valore P+A;

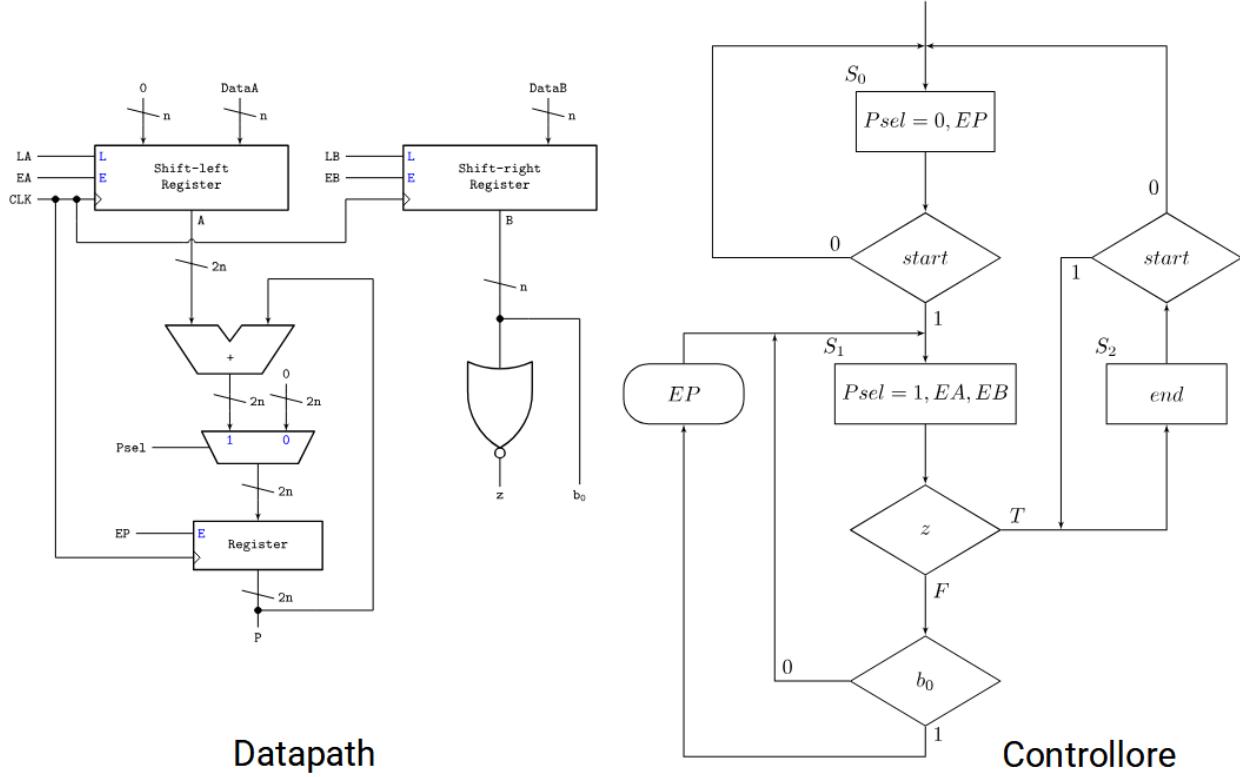
- un multiplexer 2 a 1 per controllare i dati caricati nel registro dedicato a P , permettendo la definizione di Psel, ovvero un segnale che permette di reimpostare la somma parziale se Psel=0, oppure di continuare ad eseguire le somme se Psel=1).

La porta logica NOR con, in ingresso, l'uscita B dello Shift-Right Register, implementa la funzione di confronto di B con il valore 0.

Analisi del controllore

Il controllore deriva dall'ASM, dai suoi stati e dalle sue condizioni, come descritto in seguito:

- il blocco $P \leftarrow 0$ viene implementato attraverso Psel=0 ed EP attivo alto, in modo da impostare tutti i valori a 0 per iniziare la somma nel Register;
- si assume che il caricamento e la gestione del valore di A e di B siano fatti da circuiteria esterna;
- il blocco $A \ll 1$ e $B \gg 1$ vengono realizzati tramite EA, EB e con Psel=1, in modo da eseguire le somme. Le somme verranno sempre eseguite, ma verranno salvate nel registro solo se b0=1, dato che b0 pilota il segnale EP;
- il confronto su z si fa sempre, come prima, con il blocco NOR.



Datapath e controllore ottenuti



Transistori

▼ Creatore originale: @Stefano Alverino

NMOS

- [Fase di accumulo](#)
- [Fase di svuotamento](#)
- [Fase di inversione](#)
- [Commento sul funzionamento](#)

PMOS

- [Commento sul funzionamento](#)

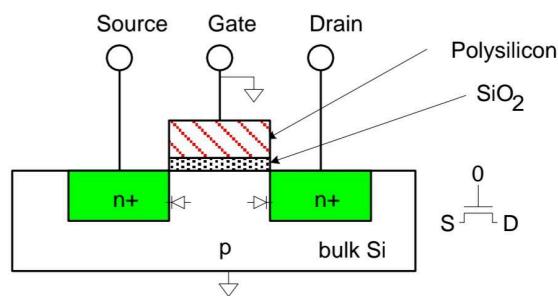
NMOS

Il transistore MOS a canale n (**NMOS**) presenta quattro terminali: **Gate**, **Source**, **Drain** e **Body** (quest'ultimo è a potenziale fisso, tipicamente a **ground**).

Gate e Body sono due conduttori separati da un isolante, e questa disposizione permette la creazione di un condensatore MOS (**metallo-ossido-semiconduttore**).

Il transistore presenterà, quindi, diversi modi di funzionamento, a seconda della tensione V_g tra gate e body:

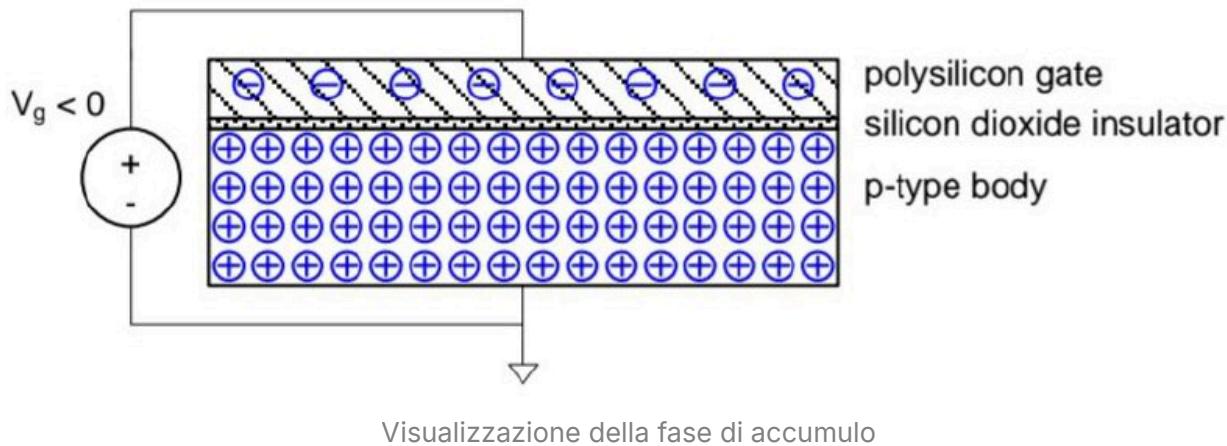
- accumulo: $V_g < 0$;
- svuotamento: $0 < V_g < V_t$;
- inversione: $V_g > V_t$.



Visualizzazione di un transistore NMOS

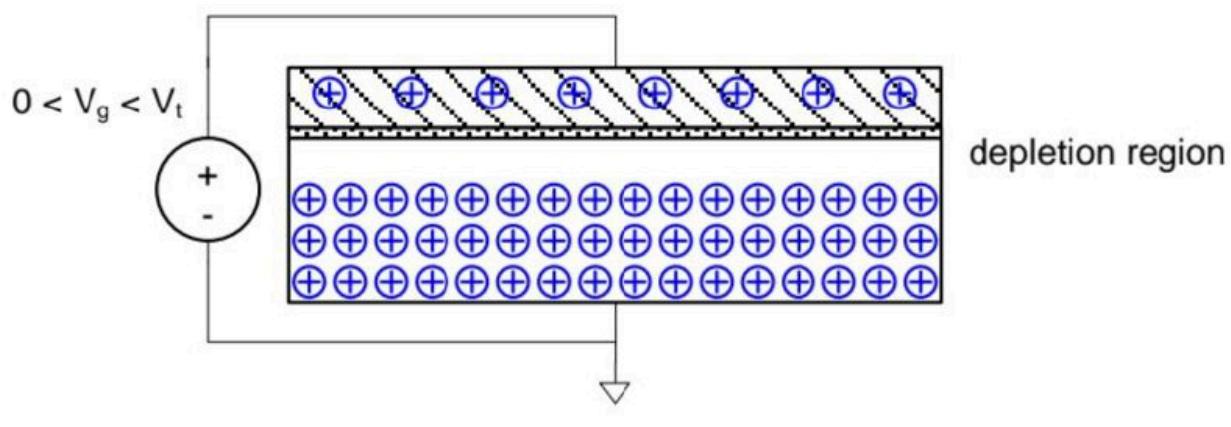
Fase di accumulo

Nella fase di **accumulo**, le cariche positive presenti nel Body vengono attratte dalle negative poste sul gate, creando così un “muro” che blocca il passaggio di corrente.



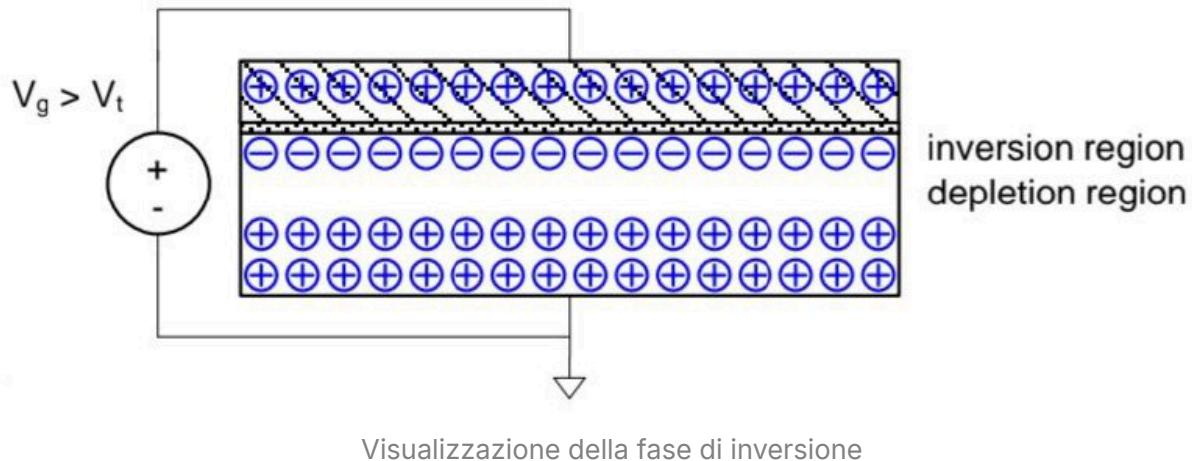
Fase di svuotamento

Quando V_g supera lo 0, si passa alla fase di **svuotamento**, in cui il potenziale positivo tende a portare cariche negative dal body al gate, ma dove la tensione non è abbastanza elevata.



Fase di inversione

Quando V_g supera V_t (tensione di soglia o threshold), si passa alla fase di **inversione**, dove le cariche negative del body vengono effettivamente portate al gate, producendo un "canale" di passaggio per la corrente tra il terminale di drain e il terminale di source.



Commento sul funzionamento

In un transistore NMOS, quando la tensione del gate è inferiore alla tensione di soglia, il passaggio di corrente tra source e drain è bloccato, producendo così un **valore logico 0 (OFF)**.

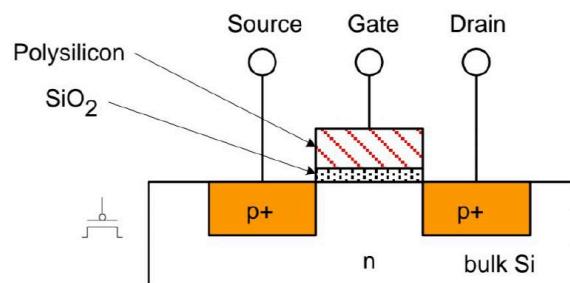
Al contrario, quando la tensione del gate supera la tensione di soglia, il passaggio di corrente è abilitato, generando un **valore logico 1 (ON)**.

$$\begin{cases} \text{OFF} & \text{se } V_g < V_t \\ \text{ON} & \text{se } V_g > V_t \end{cases}$$

PMOS

Il transistore MOS a canale p (**PMOS**) presenta quattro terminali: **Gate**, **Source**, **Drain** e **Body** (quest'ultimo è a potenziale fisso, tipicamente a tensione alta V_{DD}).

Gate e Body sono due conduttori separati da un isolante, questa disposizione permette la creazione di un condensatore MOS (metallo-ossido-semiconduttore).



Visualizzazione di un transistore PMOS

Commento sul funzionamento

Il **comportamento** è esattamente **inverso all'NMOS**.

Per una tensione di gate V_g inferiore alla tensione di soglia V_t avverrà un passaggio di corrente tra drain e source, producendo il **valore logico 1 (ON)**.

Per una tensione di gate superiore alla tensione di soglia non verrà consentito il passaggio di corrente, producendo un **valore logico 0 (OFF)**.

$$\begin{cases} \text{ON} & \text{se } V_g < V_t \\ \text{OFF} & \text{se } V_g > V_t \end{cases}$$



Implementazione delle porte logiche CMOS

▼ Creatore originale: @Stefano Alverino

CMOS

Comportamento del CMOS

Definizione delle due soglie di ingresso e dell'uscita

Calcolo della resistenza equivalente

Approccio combinatorio:

Approccio euristico:

Calcolo della capacità equivalente

Implementazione di porte logiche con CMOS

Correlazione tra rete di pull-down e rete di pull-up

NAND a 2 ingressi

NOR a 2 ingressi

NAND a 3 ingressi

Esempio - costruzione di una funzione logica

Obiettivo

Implementazione della funzione logica

Esempio - costruzione di una funzione logica

Obiettivo

Implementazione della funzione logica

La tecnologia CMOS è fortemente utilizzata nella creazione di svariate porte logiche. Alla base di questa tecnologia vi è l'uso di due tipi di transistori: **NMOS** e **PMOS**.

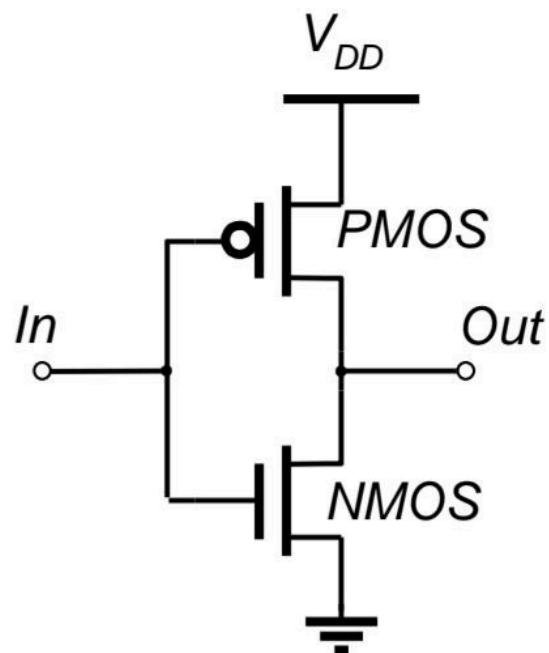
CMOS

Il componente CMOS (Complementary MOS), detto anche **inverter**, è composto da due transistori complementari:

- un **PMOS**, avente il terminale di gate connesso all'ingresso, il terminale di source alla tensione V_{DD} e il terminale di drain all'uscita. Prende il nome di **rete di pull-up**;
- un **NMOS** avente il terminale di gate connesso all'ingresso, il terminale di source a ground e il terminale di drain all'uscita. Prende il nome di **rete di pull-down**;

E' bene notare che il nome inverter è dovuto alla relazione ingresso-uscita di questo MOS:

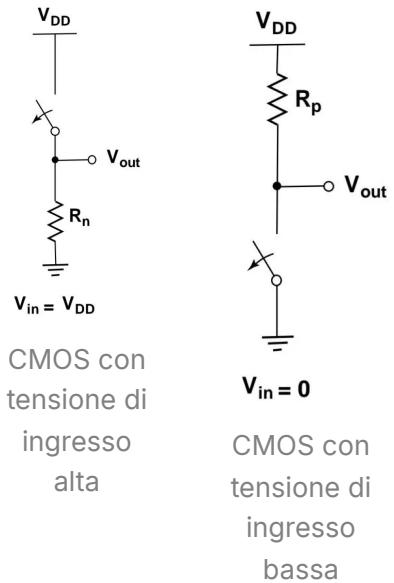
$$\begin{cases} V_{out} = V_{DD} & \text{se } V_{in} < V_t \\ V_{out} = 0 & \text{se } V_{in} > V_t \end{cases}$$



Comportamento del CMOS

I risultati ottenuti possono essere facilmente compresi analizzando il comportamento dei due MOS complementari:

- quando la tensione di ingresso V_{in} è alta (1), cioè supera il valore di soglia, e l'uscita assume un valore logico basso (0);
 - il PMOS si comporta come un circuito aperto;
 - l'NMOS si chiude e si comporta come una resistenza, collegando l'uscita V_{out} a massa.
- quando la tensione di ingresso V_{in} è bassa (0), quindi inferiore al valore di soglia, e l'uscita assume un valore logico alto (1);
 - l'NMOS si comporta come un circuito aperto;
 - il PMOS si chiude, collegando V_{out} a V_{DD} .



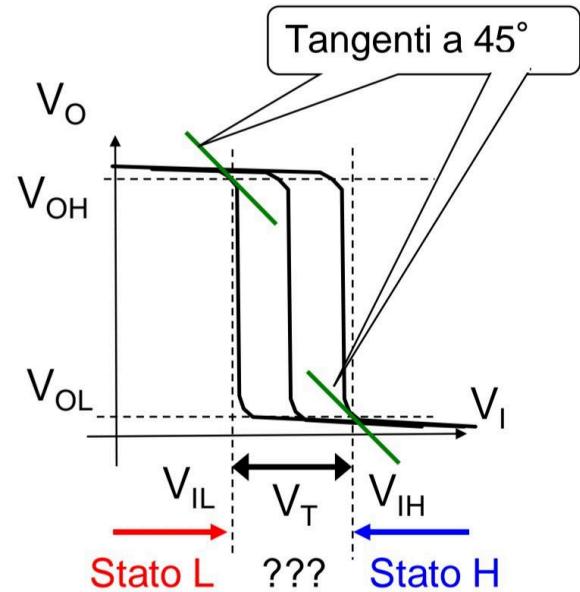
Definizione delle due soglie di ingresso e dell'uscita

La tensione di soglia V_t di un inverter CMOS può variare con l'alimentazione, la temperatura e altri tipi di disturbo, quindi è difficile determinarne un valore preciso.

Per gestire questa incertezza, si definiscono due soglie:

- **tensione di soglia inferiore V_{IL}** , e sotto questo valore l'ingresso è interpretato come 0 logico;
- **tensione di soglia superiore V_{IH}** , e sopra questo valore è interpretato come 1 logico.

Nel range $[V_{IL}, V_{IH}]$, lo stato logico non è definito, ed è quindi una **zona instabile da evitare**.



Visualizzazione delle tensioni di soglia per ingresso e uscita

L'uscita V_O può essere:

- uscita alta V_{OH} quando l'ingresso è basso;
- uscita alta V_{OL} quando l'ingresso è alto.

V_{IL} e V_{IH} sono, dal punto di vista grafico, i punti dove la curva ha una pendenza di -1, e quindi tangenti a 45° .

Calcolo della resistenza equivalente

Il calcolo della resistenza equivalente di un circuito basato su CMOS può essere effettuato tramite due approcci differenti:

▼ Approccio combinatorio:

In questo approccio si prevede di considerare tutti i possibili cammini che collegano V_{DD} (in caso di fronte di salita) o GND (in caso di fronte di discesa) all'uscita. Ogni MOS attraversato introduce una certa resistenza, sommando le resistenze lungo ogni cammino e applicando le regole per serie e paralleli, si ottiene una resistenza equivalente per ciascun percorso.

Una volta calcolate tutte, si può selezionare quella minima o massima a seconda dello scenario da modellare.

Questo approccio è efficace e preciso però diventa impegnativo in caso di grande numero di cammini

▼ Approccio euristico:

Questo approccio sfrutta le proprietà base delle reti resistive:

Le resistenze in parallelo riducono la resistenza totale, mentre quelle in serie la aumentano.

Quindi, per stimare la resistenza minima si considera il cammino che presenta il maggior numero di resistenze in parallelo e il minor numero di serie, mentre per la massima si fa l'opposto.

Pur essendo meno accurato, questo metodo è estremamente utile per un'analisi rapida e per avere un ordine di grandezza della resistenza complessiva.



Vi rimando ad un [esercizio](#) in cui viene utilizzato questo metodo per rendervi più chiare le idee.

Calcolo della capacità equivalente

Per capacità equivalente si intende la capacità totale vista dal nodo di uscita del circuito CMOS analizzato.

Per calcolarla si devono sommare tutti i contributi delle capacità di gate dei transistor MOS collegati direttamente all'uscita del circuito in esame.

Se l'uscita è collegata a

N porte logiche, e ognuna di queste possiede un numero M_i di gate MOS collegati a tale uscita, la capacità equivalente di carico può essere espressa come:

$$C_{eq} = \sum_{i=1}^N M_i * C_g$$



Vi rimando ad un [esercizio](#) in cui viene utilizzato questo metodo per rendervi più chiara e semplice la comprensione.

Implementazione di porte logiche con CMOS

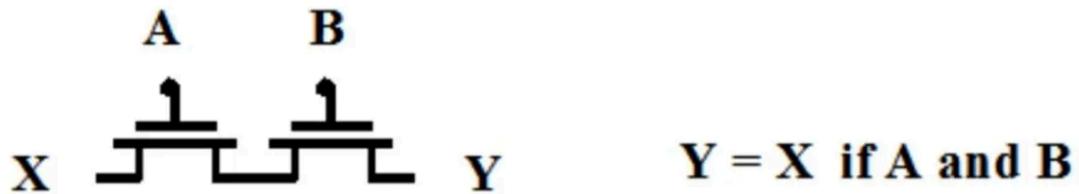
Utilizzando due reti logiche di pull-up e pull-down, al posto dei singoli transistor (come nell'inverter), si possono creare diverse porte logiche.



Quando la rete di **pull-up** risulta accesa, la rete di **pull-down** deve risultare **spenta**, e viceversa.

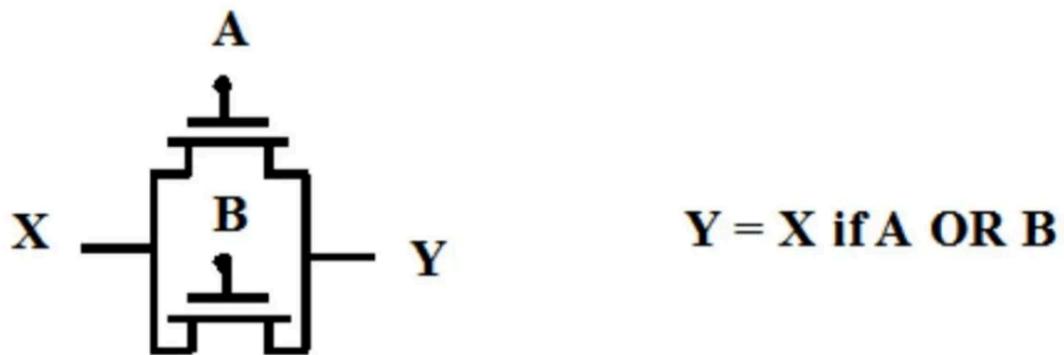
Per comprendere a pieno le implementazioni delle porte logiche, è bene analizzare il comportamento degli NMOS e dei PMOS in serie e in parallelo:

- due NMOS in serie lasciano arrivare il segnale X all'uscita Y se e solo se sia A che B presentano valore logico alto, creando una **porta logica AND**;



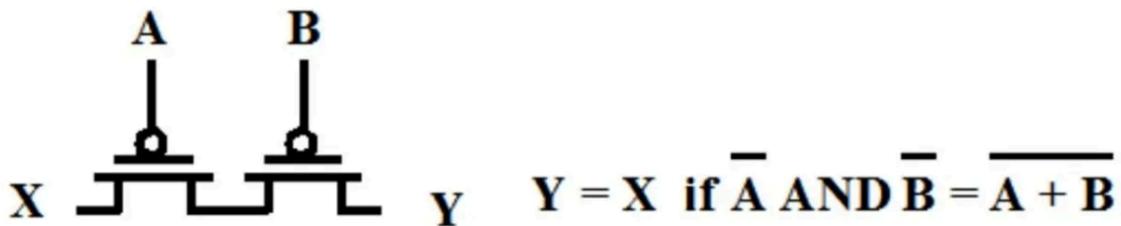
Implementazione di una porta logica AND con CMOS

- due NMOS in parallelo lasciano arrivare il segnale X all'uscita Y se almeno uno tra A e B presenta un valore logico alto, creando una **porta logica OR**;



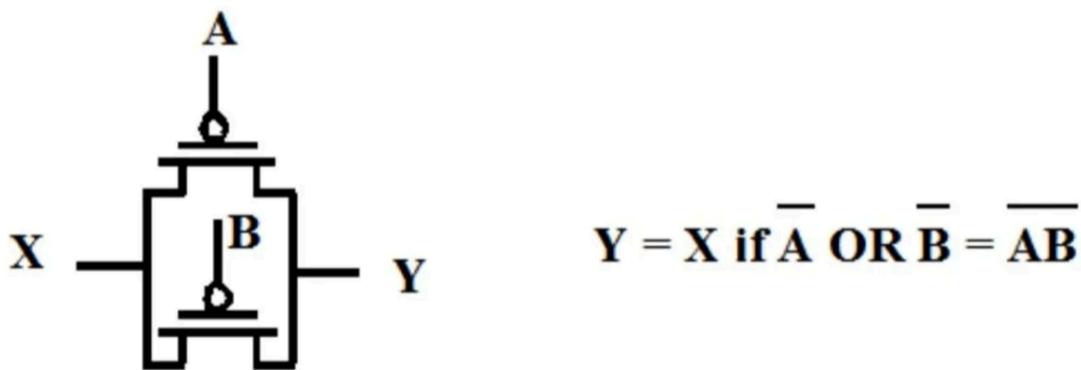
Implementazione di una porta logica OR con CMOS

- due PMOS in serie lasciano arrivare il segnale X all'uscita Y se e solo se sia A che B presentano valore logico basso, creando, tramite le [leggi di De Morgan](#), una **porta logica NOR**;



Implementazione di una porta logica NOR con CMOS

- due PMOS in parallelo lasciano arrivare il segnale X all'uscita Y se almeno uno tra A e B presenta un valore logico basso, creando, tramite le [leggi di De Morgan](#), una porta logica NAND.



Implementazione di una porta logica NAND con CMOS

Correlazione tra rete di pull-down e rete di pull-up

Per costruire la rete di pull-down a partire da quella di pull-up, bisogna:

1. scrivere la funzione logica implementata dalla rete di pull-up;
2. negare questa funzione logica usando le [leggi di De Morgan](#);
3. utilizzare la funzione risultante per costruire la rete di pull-down con transistori NMOS.

In pratica, si realizza la [funzione complementare](#) di quella della rete di pull-up, ma con struttura opposta:

- il PMOS in serie diventa NMOS in parallelo;
- il PMOS in parallelo diventa NMOS in serie.

Per costruire la rete di pull-up da quella di pull-down si può seguire il medesimo procedimento.

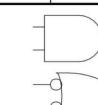
NAND a 2 ingressi

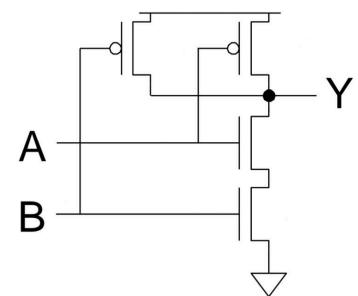
La rete di pull-up è composta da 2 PMOS in parallelo, mentre la rete di pull-down è composta da 2 NMOS in serie.

Vogliamo che la nostra porta logica restituisca il valore logico 0 solo quando entrambi gli ingressi presentano valore logico 1.

$$Y = \overline{A \cdot B}$$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0





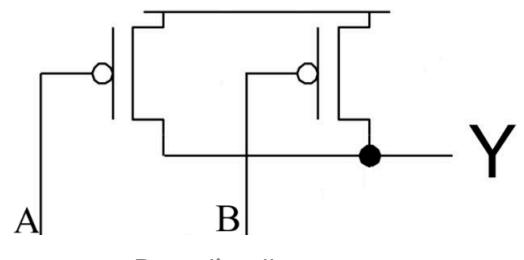
Implementazione della porta logica NAND a 2 ingressi con CMOS

Procediamo alla costruzione della porta logica NAND, ricordando che la rete di pull-up è quella che produce i **valori logici alti** sull'uscita:

1. partendo dal NAND, applichiamo De Morgan.

$$\overline{A \cdot B} = \overline{A} + \overline{B}$$

L'OR dei due valori negati è rappresentato dal parallelo di due PMOS, formando la **rete di pull-up**;

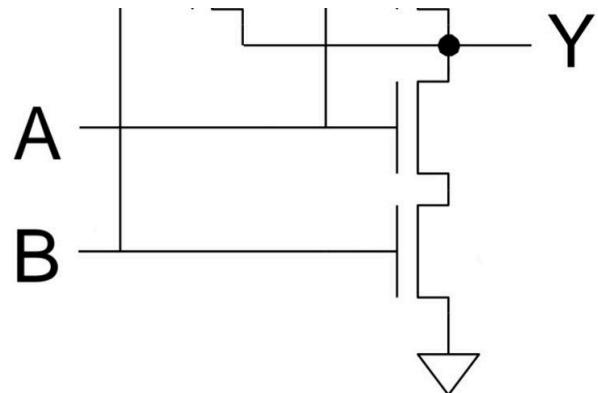


Rete di pull-up

2. partendo dalla rete di pull-up, possiamo iniziare negando la sua funzione logica.

$$\overline{\overline{A \cdot B}} = A \cdot B$$

L'AND di due valori è rappresentato dalla serie di due NMOS, formando la **rete di pull-down**.



Rete di pull-down

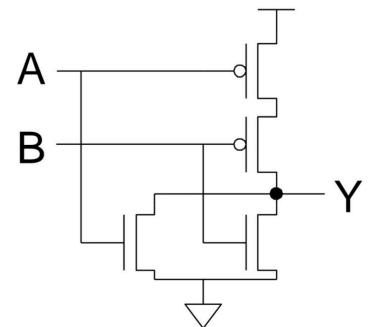
NOR a 2 ingressi

La rete di pull-up è composta da 2 PMOS in serie, mentre la **rete di pull-down** è composta da 2 NMOS in parallelo.

Vogliamo che la nostra porta logica restituisca il valore logico 1 solo quando entrambi gli ingressi presentano valore logico 0.

$$Y = \overline{A + B}$$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

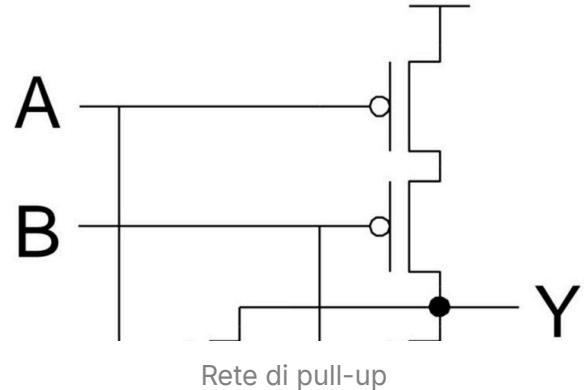
Implementazione della porta logica NOR a 2 ingressi con CMOS

Procediamo alla costruzione della porta logica NOR, ricordando che la rete di pull-up è quella che produce i **valori logici alti** sull'uscita:

1. partendo dal NOR, applichiamo De Morgan.

$$\overline{A + B} = \overline{A} \cdot \overline{B}$$

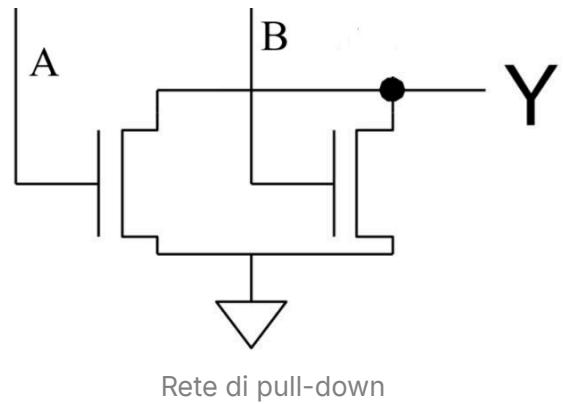
L'AND dei due valori negati è rappresentato dalla serie di due PMOS, formando la **rete di pull-up**;



2. partendo dalla rete di pull-up, possiamo iniziare negando la sua funzione logica.

$$\overline{\overline{A + B}} = A + B$$

L'OR di due valori è rappresentato dal parallelo di due NMOS, formando la **rete di pull-down**.

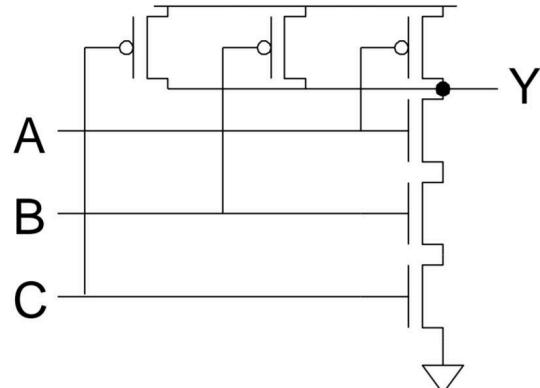


NAND a 3 ingressi

La rete di pull-up è composta da 3 PMOS in parallelo, mentre la **rete di pull-down** è composta da 3 NMOS in serie.

Vogliamo che la nostra porta logica restituisca il valore logico 0 solo quando tutti gli ingressi presentano valore logico 1.

$$Y = \overline{A \cdot B \cdot C}$$



Implementazione della porta logica NAND a 3 ingressi con CMOS

Il processo di costruzione della rete di pull-up e di quella di pull-down è il medesimo del NAND a 2 ingressi, ma inserendo un MOS aggiuntivo per ogni rete, in modo tale da gestire il terzo ingresso.

Esempio - costruzione di una funzione logica

Obiettivo

Si consideri la funzione logica U .

$$U = A + B$$

Si implementi tale funzione logica tramite porte CMOS.

▼ Implementazione della funzione logica

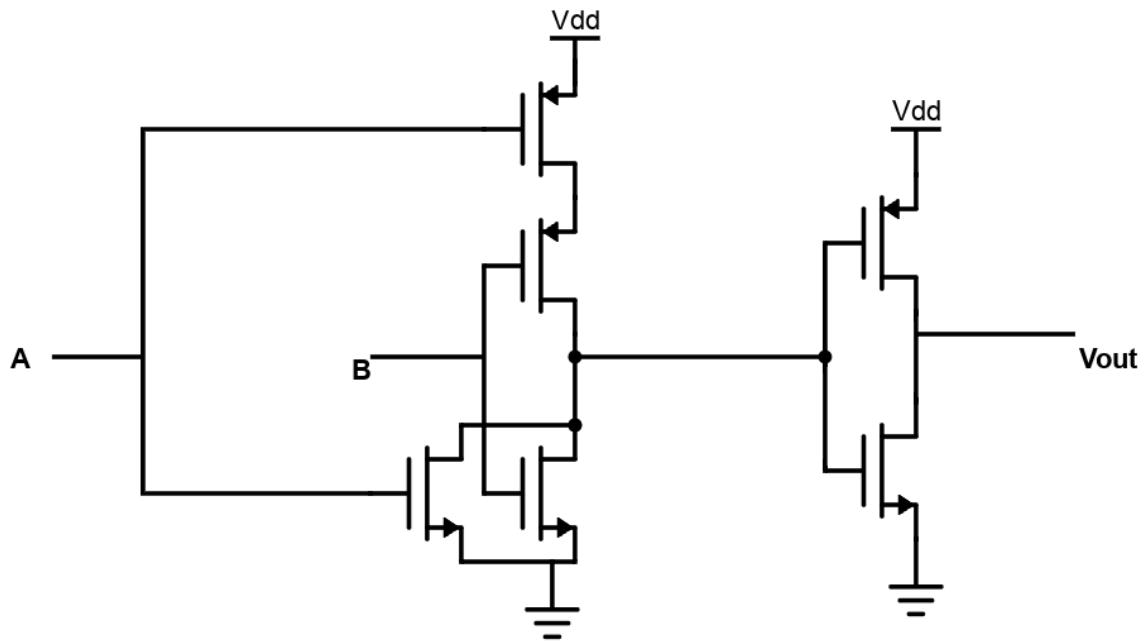
Dobbiamo ricondurci a una funzione logica del seguente tipo:

$$f = \overline{A \text{ <operazione>} B}$$

La prima cosa da fare è considerare:

$$U = \overline{\overline{U}} = \overline{\overline{A + B}}$$

In questo caso particolarmente fortunato, possiamo utilizzare due porte logiche note per implementare tale circuito, poiché $\overline{A + B}$ rappresenta una porta NOR a cui, però, dobbiamo aggiungere un inverter, definito dall'altra negazione, in modo da raggiungere la funzione $\overline{\overline{A + B}}$.



Il primo blocco a sinistra è la realizzazione della porta NOR, mentre il secondo blocco rappresenta l'inverter

Esempio - costruzione di una funzione logica

Obiettivo

Si consideri la funzione logica U .

$$U = \overline{A} \cdot \overline{B} + \overline{C}$$

Si implementi tale funzione logica tramite porte CMOS.

▼ Implementazione della funzione logica

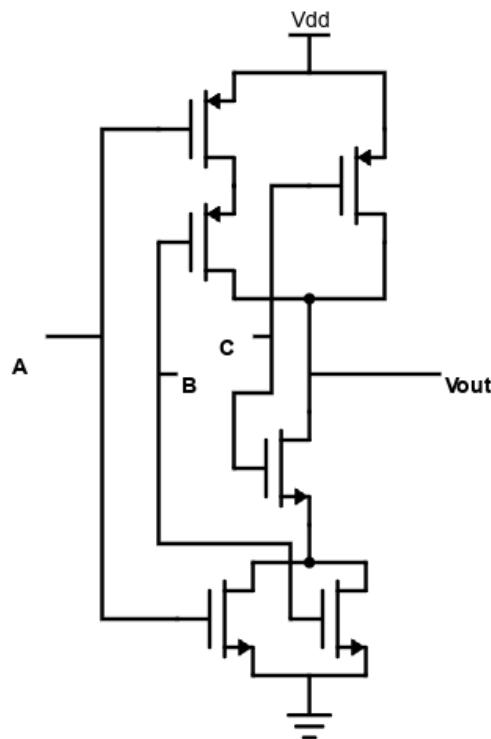
Applicando le leggi di De Morgan:

$$\overline{A} \cdot \overline{B} + \overline{C} = \overline{A + B} + \overline{C}$$

Riapplicando nuovamente le leggi di De Morgan, otteniamo:

$$\overline{A + B} + \overline{C} = \overline{(A + B) \cdot C}$$

Tale funzione logica è ora pienamente realizzabile attraverso porte logiche con implementazione CMOS.



Implementazione della porta logica richiesta con CMOS



Tempo di propagazione (CMOS)

▼ Creatore originale: @Gianbattista Busonera

- @<utente> (<data>): <modifiche effettuate>

Tempo di propagazione di stato da basso ad alto

Calcolo del tempo di propagazione di stato da basso ad alto

Tempo di propagazione di stato da alto a basso

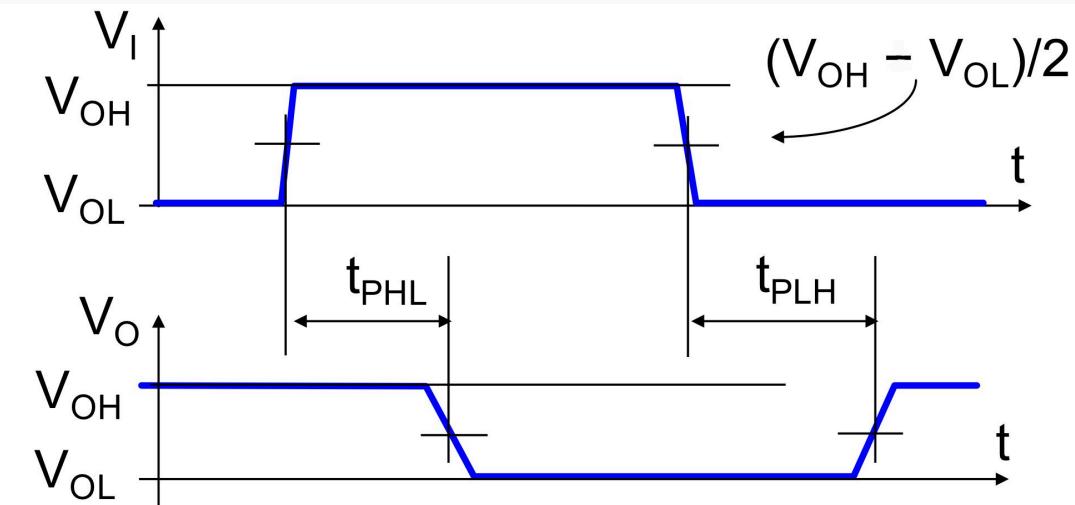
Il calcolo del tempo di propagazione è un concetto estremamente simile al calcolo del tempo di salita e di discesa di un condensatore.

E' un ritardo valutato fra coppie di segnali, in particolare fra la tensione in ingresso V_{in} e la tensione in uscita V_{out} .

I ritardi tra ingressi e uscite di una porta sono definiti con riferimento al 50% della variazione $V_{OH} - V_{OL}$, cioè fra l'istante in cui la tensione in ingresso raggiunge per la prima volta il valore del 50% della tensione finale in ingresso e l'istante in cui la tensione di uscita raggiunge il 50% della tensione in finale in uscita.



Ci stiamo chiedendo, insomma, quanto tempo - t_p - impiega la tensione in ingresso V_{in} , una volta raggiunta la tensione di soglia $V_T = \frac{V_{OH} - V_{OL}}{2}$ a far commutare la tensione di uscita V_{out} dove, per "commutare", si intende il raggiungimento della tensione di soglia.



L'immagine (sistemata rispetto alle slide che presentavano un errore) rappresenta bene cosa si intende per tempo di propagazione.

In questo caso, a differenza del tempo di salita/discesa, non ci chiediamo più quanto tempo ci si mette a passare dal 10% al 90% del valore finale V_∞ , ma quanto tempo è necessario affinché si superi la tensione di soglia V_T .



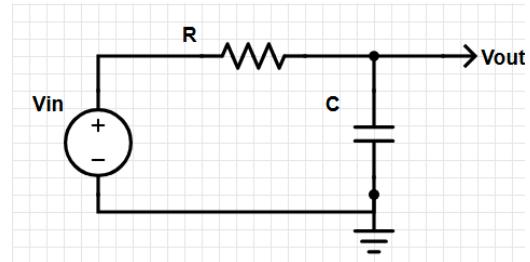
Se non espressamente indicato dal testo, possiamo ipotizzare che la tensione di soglia sia definita come:

$$V_T = \frac{V_{DD} - V_{GND}}{2} = \frac{V_{DD}}{2}$$

Tempo di propagazione di stato da basso ad alto

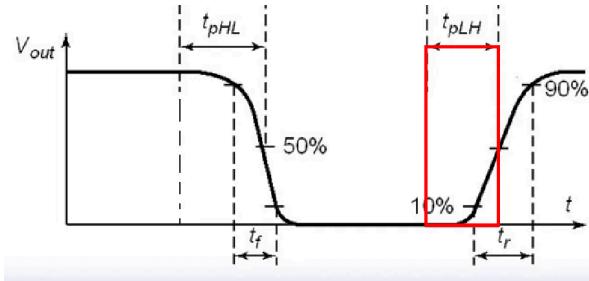
Rappresentiamo un circuito RC, che possiede quindi un resistore R e un condensatore C (carico). La tensione sul carico (condensatore) è definita come:

$$V_{\text{out}} = V_c(t) = V_\infty + (V_0 - V_\infty)e^{-\frac{t}{\tau}}$$



Rappresentazione di un circuito RC

Il tempo di propagazione dallo stato basso allo stato alto è definito come il tempo che il **segnale di uscita** impiega a salire dallo stato basso (0V solitamente) fino alla tensione di soglia (V_T).



In rosso, il tempo di propagazione da alto a basso

Di conseguenza, noi vogliamo determinare quanto vale $t_p^{\text{L} \rightarrow \text{H}}$, ovvero il tempo necessario affinché la tensione sul condensatore sia pari alla tensione di soglia V_T

$$V_c(t_p^{\text{L} \rightarrow \text{H}}) = V_T$$

Calcolo del tempo di propagazione di stato da basso ad alto

Iniziamo definendo l'equazione da cui ottenere il valore.

$$V_c(t_p^{\text{L} \rightarrow \text{H}}) = V_T = V_\infty + (V_0 - V_\infty)e^{-\frac{t_p^{\text{L} \rightarrow \text{H}}}{\tau}}$$

Ipotizzando che $V_0 = V_{\text{GND}} = 0 \text{ V}$, $V_T = \frac{V_{\text{DD}}}{2}$, $V_\infty = V_{\text{DD}}$, si ottiene:

$$\frac{V_{DD}}{2} = V_{DD} + (0 - V_{DD})e^{-\frac{t_p^{L \rightarrow H}}{\tau}} \quad (1)$$

$$\frac{V_{DD}}{2} = V_{DD} \left(1 - e^{-\frac{t_p^{L \rightarrow H}}{\tau}} \right) \quad (2)$$

$$\frac{1}{2} = 1 - e^{-\frac{t_p^{L \rightarrow H}}{\tau}} \quad (3)$$

$$\frac{1}{2} = e^{-\frac{t_p^{L \rightarrow H}}{\tau}} \quad (4)$$

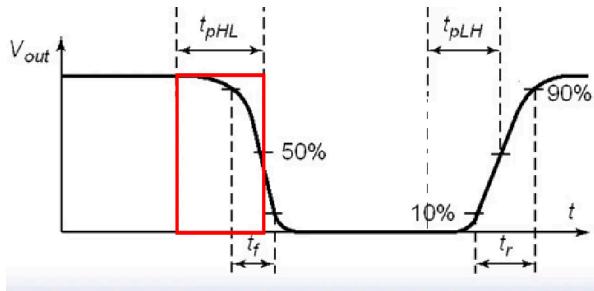
$$\ln \left(\frac{1}{2} \right) = -\frac{t_p^{L \rightarrow H}}{\tau} \quad (5)$$

Al termine, si ottiene il valore di $t_p^{L \rightarrow H}$.

$$t_p^{L \rightarrow H} = -\ln \left(\frac{1}{2} \right) \tau \simeq 0.69\tau \simeq 0.69R_{eq}C_{eq}$$

Tempo di propagazione di stato da alto a basso

Il tempo di propagazione dallo stato alto allo stato basso è il tempo necessario per far passare l'uscita dalla tensione alta $V_c(t) = V_\infty$ alla tensione $V_c(t) = V_T$.



In rosso il tempo di propagazione da alto a basso

A tale scopo, vista l'estrema somiglianza dei calcoli, si allega direttamente la formula per il calcolo del tempo di propagazione dallo stato alto allo stato basso $t_p^{H \rightarrow L}$:

$$t_p^{H \rightarrow L} = -\ln \left(\frac{1}{2} \right) \tau \simeq 0.69\tau \simeq 0.69R_{eq}C_{eq}$$



Tempi di salita e di discesa

▼ Creatore originale: @Gianbattista Busonera

- @Giacomo Dandolo (17/04/2025): evidenziati meglio tempo di salita/discesa nelle immagini

Tempo di salita

Calcolo del tempo di salita

Primo metodo

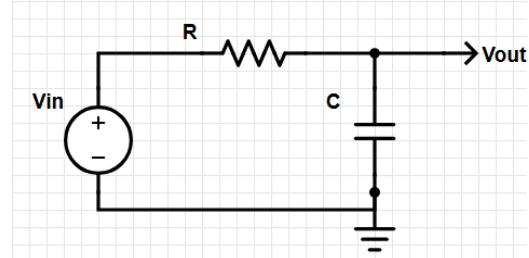
Secondo metodo

Tempo di discesa

Tempo di salita

Rappresentiamo un circuito RC, che possiede quindi un resistore R e un condensatore C (carico). La tensione sul carico (condensatore) è definita come:

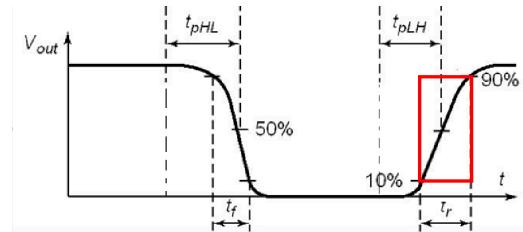
$$V_{\text{out}} = V_c(t) = V_\infty + (V_0 - V_\infty)e^{-\frac{t}{\tau}}$$



Rappresentazione di un circuito RC

Il tempo di salita t_{rise} (rise time) è definito come il tempo che il segnale impiega a salire dal 10% al 90% del valore finale (V_∞). Di conseguenza, noi vogliamo sapere quanto vale:

$$t_{\text{rise}} = t_{90\%} - t_{10\%}$$



Il tempo di salita è evidenziato in rosso

Calcolo del tempo di salita

Ci sono due metodi:

- il primo, più lungo, che utilizza l'[equazione di \$V_{\text{out}}\$](#) ;
- il secondo, più rapido, utilizza il [risultato del primo metodo](#).

▼ Primo metodo

1. Calcoliamo $t_{10\%}$, cioè il tempo in cui la tensione sul condensatore è pari al 10% del valore della tensione finale V_∞ .

$$V_c(t_{10\%}) = \frac{10}{100} V_\infty = V_\infty + (V_0 - V_\infty) e^{-\frac{t_{10\%}}{\tau}} = 0.1 V_\infty$$

Consideriamo il condensatore inizialmente scarico, con $V_0 = 0$ V.

$$V_\infty + (V_0 - V_\infty) e^{-\frac{t_{10\%}}{\tau}} = 0.1 V_\infty \quad (1)$$

$$-0.9 = \left(\frac{0 - V_\infty}{V_\infty} \right) e^{-\frac{t_{10\%}}{\tau}} \quad (2)$$

$$e^{-\frac{t_{10\%}}{\tau}} = \frac{-0.9 \cdot V_\infty}{-V_\infty} \quad (3)$$

$$-\frac{t_{10\%}}{\tau} = \ln \left(\frac{0.9 \cdot V_\infty}{V_\infty} \right) \quad (4)$$

$$t_{10\%} = -\tau \cdot \ln(0.9) \simeq 0.105\tau \quad (5)$$

2. Calcoliamo $t_{90\%}$, cioè il tempo in cui la tensione sul condensatore è pari al 90% del valore della tensione finale V_∞ .

$$V_c(t_{90\%}) = \frac{90}{100} V_\infty = V_\infty + (V_0 - V_\infty) e^{-\frac{t_{90\%}}{\tau}} = 0.9 V_\infty$$

Consideriamo il condensatore inizialmente scarico, con $V_0 = 0$ V.

$$V_\infty + (V_0 - V_\infty)e^{-\frac{t_{90\%}}{\tau}} = 0.9V_\infty \quad (6)$$

$$-0.1 = \left(\frac{0 - V_\infty}{V_\infty} \right) e^{-\frac{t_{90\%}}{\tau}} \quad (7)$$

$$e^{-\frac{t_{90\%}}{\tau}} = \frac{-0.1 \cdot V_\infty}{-V_\infty} \quad (8)$$

$$-\frac{t_{90\%}}{\tau} = \ln \left(\frac{0.1 \cdot V_\infty}{V_\infty} \right) \quad (9)$$

$$t_{90\%} = -\tau \cdot \ln(0.1) \simeq 2.306\tau \quad (10)$$

3. Calcoliamo t_{rise} .

$$t_{\text{rise}} = t_{90\%} - t_{10\%} = (2.306 - 0.105)\tau \simeq 2.2\tau$$

▼ Secondo metodo

Si ipotizza, come nel primo metodo, il caso in cui $V_0 = 0$ V. Si ottiene, quindi, in generale:

$$V_c(t) = V_\infty(1 - e^{-\frac{t}{\tau}})$$

1. Calcoliamo $t_{10\%}$:

$$V_c(t_{10\%}) = V_\infty(1 - e^{-\frac{t_{10\%}}{\tau}}) = 0.1V_\infty$$

$$0.1 = 1 - e^{-\frac{t_{10\%}}{\tau}} \quad (11)$$

$$0.9 = e^{-\frac{t_{10\%}}{\tau}} \quad (12)$$

$$t_{10\%} = -\tau \cdot \ln(0.9) \quad (13)$$

2. Calcoliamo $t_{90\%}$:

$$V_c(t_{90\%}) = V_\infty(1 - e^{-\frac{t_{90\%}}{\tau}}) = 0.9V_\infty$$

$$0.9 = 1 - e^{-\frac{t_{90\%}}{\tau}} \quad (14)$$

$$0.1 = e^{-\frac{t_{90\%}}{\tau}} \quad (15)$$

$$t_{90\%} = -\tau \cdot \ln(0.1) \quad (16)$$

3. Calcoliamo t_{rise} :

$$t_{\text{rise}} = t_{90\%} - t_{10\%} = -[\ln(0.1) - \ln(0.9)]\tau = -\ln\left(\frac{0.1}{0.9}\right)\tau = \ln(9)\tau \simeq 2.2\tau$$

Il tempo di salita si può calcolare nel seguente modo:

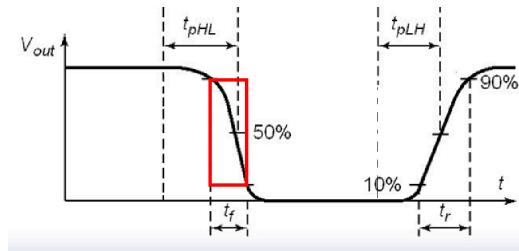
$$t_{\text{rise}} = 2.2\tau = 2.2 \cdot R_{\text{eq}}C_{\text{eq}}$$

Tempo di discesa

Il tempo di discesa è il tempo necessario per passare dal 90% al 10% del valore finale (V_∞). Utilizzeremo le stesse formule viste in precedenza:

$$t_{\text{fall}} = t_{90\%} - t_{10\%}$$

$$t_{\text{fall}} = 2.2\tau = 2.2 \cdot R_{\text{eq}}C_{\text{eq}}$$



Il tempo di discesa è evidenziato in rosso



Basi del linguaggio Verilog

▼ Creatori originali: @Gianbattista Busonera

[Introduzione](#)

[Utilizzo](#)

[Input e Output](#)

[Tipi di dato](#)

[net](#)

[variabile](#)

[Tipi di dato aggregato](#)

[Vettori di bit](#)

[Esempi - Vettori di bit](#)

[Matrici in Verilog](#)

[Esempi - Matrici](#)

[Valori logici in Verilog](#)

[Esempio delle combinazioni logiche](#)

[Costanti in Verilog](#)

[Sintassi delle costanti in Verilog](#)

[Esempi - Costanti](#)

[Macro in Verilog](#)

[Definizione di una macro](#)

[Esempio - Definizione di una macro](#)

[For, if, case](#)

[Tipi di modellazione](#)

[Modellazione strutturale](#)

[Definizione di un modulo](#)

[Esempio - Interfaccia del modulo ALU](#)

[Esempio - Interfaccia di un processore](#)

[Modellazione a livello di porta logica \(GATE-LEVEL\)](#)

[Porte logiche elementari in Verilog](#)

[Valori di uscita delle porte logiche elementari](#)

[Esempio - Modellazione a livello di porta logica](#)

[Esempio - Semisommatore con porte logiche](#)

[Ritardi di porte logiche](#)

[Esempio - Semisommatore con porte logiche, con ritardi](#)

[Esempio - Utilizzo di sottomoduli](#)

[Esempio - Sommatore a quattro bit con sottomoduli](#)

[Modellazione RTL](#)

[Operatore Assign](#)

[Esempi - Operatore assign](#)

[Operatori Verilog utili in assegnazioni continue](#)

[Esempi - Operatori Verilog](#)
[Esempio - XOR con ritardo \(RTL\)](#)
[Esempio - Sommatore completo](#)
[Esempio - Sommatore a 4 bit](#)
[Realizzazione di moduli parametrici](#)
[Esempio - Sommatore a N bit parametrico](#)
[Passaggio di argomenti e parametri](#)
[Ordine delle istruzioni](#)
[Blocco generate con for e if](#)
[Esempio - Sommatore completo su N bit con generate](#)
Modellazione comportamentale
[Always con circuiti combinatori](#)
[Esempio - Porta AND](#)
[Always con circuiti sequenziali](#)
[Esempio - Porta AND + FF](#)
[Assegnazioni bloccanti](#)
[Assegnazioni non bloccanti](#)
[Esempio - Realizzazione di un circuito sequenziale Mealy \(FSM\)](#)
[Esempio - Realizzazione di un circuito sequenziale Moore \(FSM\)](#)

Introduzione

Verilog è uno dei tre principali linguaggi utilizzati per modellare l'hardware. Essendo basato su C, risulta essere più semplice da comprendere e apprendere di altri linguaggi, ma risulta meno tipizzato.

Inoltre, è un linguaggio intrinsecamente **parallelo** per modellare il parallelismo intrinseco dell'hardware.

Con Verilog si possono modellare sistemi a **diversi livelli di astrazione**:

1. modellazione a **livello di porte logiche**:
 - a. livello più basso, che utilizza porte elementari Verilog;
 - b. tutte le istruzioni sono **simultanee**;
 - c. l'**ordine** delle istruzioni **NON** è importante.
2. modellazione **strutturale**:
 - a. descrive la struttura di un circuito con i moduli, a diversi livelli;
 - b. tutte le istruzioni sono **simultanee**;
 - c. l'**ordine** delle istruzioni **NON** è importante.
3. modellazione **flusso di dati** (dataflow) o **register transfer level (RTL)**:
 - a. descrive il flusso di dati tra ingressi e uscite, utilizzando istruzioni di **assegnazione simultanea**;

- b. l'assegnazione è **continua**;
 - c. tutte le istruzioni sono **simultanee**;
 - d. l'ordine delle istruzioni **NON è importante**.
4. modellazione **comportamentale**:
- a. descrive ciò che fa il circuito utilizzando blocchi procedurali d'istruzioni;
 - b. le assegnazioni sono **procedurali**, e quindi le istruzioni, vengono **eseguite in sequenza**;
 - c. l'ordine delle istruzioni **è importante**.



Nello stesso progetto possono essere utilizzati modelli di diverso tipo.

Utilizzo

Verilog può essere utilizzato per:

- Simulazione;
- Sintesi;
- Verifica formale.

Input e Output

In Verilog esistono diversi tipi di dichiarazioni delle "porte" di ingresso/uscita in un modulo:

- Input: porta di ingresso;
- Output: porta di uscita;
 - Signed: va esplicitamente dichiarato! È, però, necessario solo nelle moltiplicazioni;
 - Unsigned: scelta di default.
- Inout: porta di ingresso e uscita.



Di base, Verilog fa le operazioni in CA2 e, quindi, le operazioni di somma e sottrazione non subiscono variazioni in caso di "signed" o "unsigned", ma solo quelle di moltiplicazione.

Tipi di dato

Verilog ha due tipi principali di dati: **net** e **variabile**.

net

I dati di tipo `net` permettono connessioni tra le parti di un progetto, come fossero fili.

Un esempio di dato `net` è il tipo `wire`, il quale:

- permette di trasferire un valore tra i componenti;
- può essere pilotato da un'assegnazione continua;
- non permette un'assegnazione procedurale;
- non può memorizzare un valore;
- non può memorizzare uno stato, e deve quindi essere costantemente pilotato, altrimenti il valore viene perso;

```
// definizione di un tipo di dato wire  
wire <nome_wire>
```

Esistono altri tipi di dato `net`, quali `input`.

variabile

I dati di tipo `variabile` permettono di memorizzare i valori dei dati.

Un esempio di dato `variabile` è il tipo `reg`, il quale:

- può memorizzare il valore di un'assegnazione procedurale;
- ricorda l'ultimo valore assegnato;
- non può essere pilotato da un'assegnazione continua;
- non corrisponde necessariamente a un registro nell'hardware.

```
// definizione di un tipo di dato reg  
reg <nome_reg>
```

Esistono altri tipi di dato `variabile`, quali `intero` e `genvar`.



Qualsiasi tipo di dato può essere utilizzato come ingresso (RHS) in un'espressione, indipendentemente dal suo genere.

Tipi di dato aggregato

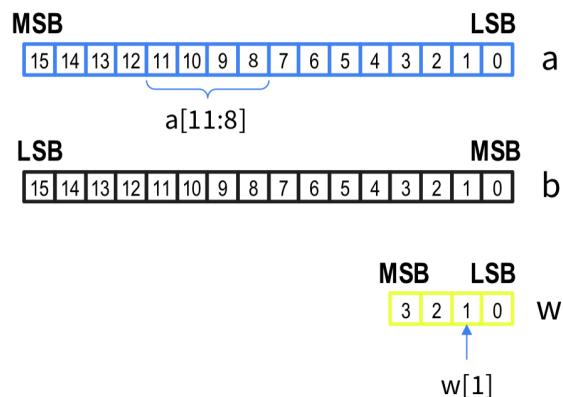
Vettori di bit

Sia che si utilizzi un tipo di dato "net" o un tipo di dato "variabile", possiamo specificare il parallelismo di una data connessione (net) o di una data variabile (reg) tramite:

Un vettore di bit, o bus, è una dichiarazione multi-bit che utilizza un singolo nome. Esso viene specificato come un intervallo `[msb:lsb]`.

Prendono anche il nome di "**vectors**" (o packed arrays).

Esistono anche gli "**unpacked arrays**", utilizzati come classici array, non approfonditi.



Esempio di un vettore di bit

▼ Esempi - Vettori di bit

Definiamo alcuni esempi di definizione e di accesso ai vettori di bit:

```
input [15:0] a; // a è un vettore d'ingresso a 16 bit → tipo di dato net
                // serve infatti per poter collegare l'ingresso di un
                // componente con l'uscita di un altro
```

```
reg [0:15] b; // il bit 0 è il MSB; il vettore b è di tipo reg
wire [3:0] w; // il bit 3 è il MSB; il vettore w è di tipo wire
```

```
w[1];      // selezione del secondo bit del vettore w
//^ si prende il bit numero 1
```

```
a[11:8]; // selezione dell'intervallo [11:8] di 4 bit
//^ si prendono i bit 11, 10, 9, 8
```

Si noti come l'intervallo di selezione deve essere coerente con la dichiarazione del vettore.

Matrici in Verilog

Le matrici possono esser definite specificando un intervallo di indirizzi, è necessario fornire gli indici superiore ed inferiore , dopo il nome dell'identificatore.

Può essere formata da scalari (0 o 1) o vettori di bit (matrice di vettori). Possono avere qualsiasi numero di dimensione.



La sintassi è la seguente:

```
<tipo> [intervallo_colonne] nome_matrice [intervallo_righe];
```

▼ Esempi - Matrici

```
// per comprendere bene da dove si parte per la creazione di matrici,  
// conviene creare inizialmente una matrice monodimensionale (cioè, un vettore)  
reg a[15:0] // la matrice monodimensionale (è un vettore)  
// abbiamo quindi 16 elementi da (implicitamente) 1 bit, ed è simile a scrivere:  
// reg [0:0] a [15:0]  
// otteniamo così 16 "righe" fatte da reg di 1 bit ciascuna, cioè da scalari.  
  
wire [7:0] b[15:0] // b matrice di 16 elementi di tipo wire ognuno con  
// dimensione (parallelismo) 8 bit  
  
wire [7:0] c[3:0][3:0] // c è matrice 4×4, 16 elementi di dimensione 8 bit
```

Spesso una matrice di registri rappresenta una memoria.

```
reg [7:0] mem [1023:0] // mem è una matrice di 1024 elementi, ciascuno da 8 bit  
  
mem[i]          // indica un byte specifico della memoria (indirizzo i)  
mem[i][j]        // indica un bit specifico di un byte specifico  
                  // (indirizzo i,j)  
                  // posso dichiarare anche matrici
```

Valori logici in Verilog

Verilog utilizza quattro valori di base, evidenziati nella tabella.

Simbolo	Significato	Descrizione
0	Basso logico (Low)	Rappresenta il livello logico basso, come in digitale "spento".
1	Alto logico (High)	Rappresenta il livello logico alto, come in digitale "acceso"
x o X	Indefinito (Unknown)	Valore sconosciuto o conflitto (es. due inverter che pilotano valori diversi). E' utile in simulazione per rilevare

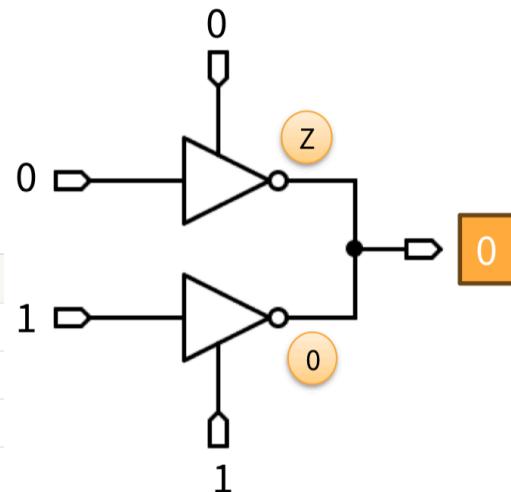
Simbolo	Significato	Descrizione
		errori o stati non inizializzati.
	Alta impedenza (High Impedance)	Stato di disconnessione del segnale (es. quando una linea è "libera"). Viene usato per linee condivise, ad esempio nei bus tri-state, dove solo un dispositivo alla volta guida la linea.

Esempio delle combinazioni logiche

Nel caso di un valore  come in [figura](#) in uscita da un inverter di tipo tri-state non abilitato in combinazione a un altro inverter di tipo tri-state (abilitato) produce sul nodo d'uscita un valore corretto (cioè 0).

Cosa succede nel caso di due uscite collegate insieme? Otteniamo la seguente tabella:

	0	1	X
0	0	X	X
1	X	1	X
X	X	X	X
Z	0	1	X



Esempio di una delle combinazioni, utilizzando delle porte tri-state

Costanti in Verilog

Le costanti sono valori letterali usati per rappresentare numeri (binari, decimali, esadecimali, ecc.) nel codice. Sono utili per assegnare valori a segnali o registri e per descrivere maschere di bit, pattern, indirizzi, ecc.

Sintassi delle costanti in Verilog

Definiamo la sintassi generale delle costanti:

```
<numero_bit>'<base><valore>
```

- `numero_bit` indica la dimensione in bit della costante dichiarata;
 - se omesso, la dimensione è **32 bit**.
- `'` separatore, sempre necessario;
- `base` indica la [base numerica](#) in cui è espresso il valore della costante;

Simbolo	Base
b	binaria
o	ottale
d	decimale
h	esadecimale (hex)

- se la base non è specificata, si assume sia in base **decimale!**
- valore** è il numero nella base specificata;
- è possibile utilizzare il carattere **_** per migliorare la leggibilità di gruppi di bit.

```
8'b10111011 // senza underscore
8'b1011_1011 // con underscore, più leggibile

// entrambi rappresentano esattamente lo stesso valore, nel secondo caso
// sono stati divisi i bit in gruppi di 4 per migliorarne la leggibilità
```

▼ Esempi - Costanti

```
// binario su 8 bit
8'b1011_1011

// esadecimale su 32 bit (se la dimensione della costante
// non è specificata, la costante è composta da 32 bit)
'hA3F0

// ottale su 16 bit
16'o56377

// decimale su 32 bit
32'd999 // uguale a 999
```

Macro in Verilog

Le macro in Verilog sono direttive definite dal preprocessore, simili a quelle di C e C++, e servono a definire costanti, frammenti di codice riutilizzabili o condizionali di compilazione.

Vengono interpretate prima della compilazione vera e propria.

Definizione di una macro

La direttiva `'define` serve a dichiarare una macro, ovvero un'etichetta che verrà sostituita dal testo associato in fase di pre-processamento.



La virgoletta (`'`, backtick) non è un apostrofo, ma su Windows è ottenuto tramite `ALT+96`.

```
'define NOME_MACRO valore_macro // in C è #define NOME_MACRO valore
```

- `NOME_MACRO` definisce il nome della macro, solitamente a caratteri maiuscoli;
- `valore_macro` è una qualsiasi sequenza di caratteri, e molto spesso una costante numerica.

Ovunque venga utilizzato il valore `'NOME_MACRO`, esso verrà sostituito dal valore associato `valore_macro`.

▼ Esempio - Definizione di una macro

```
// associa al nome XSIZE il valore decimale 96
#define XSIZE 8'd96

wire [XSIZEx1:0] xpos; // 'XSIZEx prende il valore di XSIZEx, cioè 96
```

For, if, case

Tali sintassi ricordano estremamente la sintassi del C, sostituendo "{" con `begin` e "}" con `end`:

- per il for:

```
genvar i, j; // si utilizza per ciclare
for(i = 0; i < N; i++)
    istruzione1;

for(j = 0; j < N; j++) begin
    istruzione1;
    ...
    istruzioneK;
end
```

- per gli if:

```
output [1:0] risultato;
input a, b;
```

```

if(sel == 2'b00) // se il selettore è uguale a 00 in base 2 (su due bit)
    risultato = a+b;
else if (sel == 2'b01) begin
    risultato = a-b;
end
else if (sel == 2'b10)
    risultato = a&b;
else
    risultato = a|b;

```

- per i case:

```

case (sel)
  2'b00: begin
    risultato = a + b;
  end
  2'b01: risultato = a - b;
  2'b10: risultato = a & b;
  default: risultato = 16'bX; // mette tutto in unknown ... come don't care
endcase

```

Si noti come non è necessario utilizzare `begin` ed `end` nel caso di una sola istruzione.

Tipi di modellazione

Modellazione strutturale

Describe la struttura gerarchica del circuito tramite istanze di moduli (che possono a loro volta essere RTL, gate-level, etc.), è simile a disegnare uno schematico, ma con una descrizione testuale.

L'elemento base è `module endmodule` per ogni blocco, le cui istanze si collegano con net (tipicamente `wire`).

```

module <nome> ([lista delle porte]);
  // contenuti del modulo
endmodule

// si noti come un modulo può omettere la
// lista delle porte
module <nome>;

```

```
endmodule
```

Tutte le connessioni sono simultanee, e quindi l'ordine delle istanze non conta.

Il loro uso tipico è:

- integrare blocchi IP (Intellectual Property block), ovvero blocchi di logica già progettati, testati ed ottimizzati, che è possibile comprare, licenziare o riutilizzare all'interno di un progetto invece che implementarlo da zero;
- costruire sistemi gerarchici (per esempio, CPU + periferiche).

Nella pratica moderna, il 90% del codice è scritto in **RTL o dataflow**, per poi lasciare al sintetizzatore la generazione gate-level finale. Si usa, invece, la modellazione **strutturale** per mettere insieme i vari moduli.

Definizione di un modulo

Un modulo è un componente riutilizzabile. In Verilog, ogni blocco di circuito è racchiuso tra le parole chiave `module` ed `endmodule`.

All'interno del modulo si dichiarano:

- elenco degli ingressi e delle uscite (le uscite vanno inserite prima per convenzione);
- le caratteristiche delle porte (ingressi e uscite, eventualmente signed) e il loro parallelismo;
- eventuale dichiarazione di parametri (costanti di cui possiamo modificare il valore), che ci consentono di rendere parametrico il nostro circuito;
- tramite la parola chiave `'include` possiamo inserire nel nostro modulo altri componenti definiti in precedenza;
- segue la parte più corposa del nostro modulo, che analizzeremo man mano nella trattazione.



Visualizzazione della struttura interna di un modulo

```
module nome_modulo
  #(parametri opzionali) // facoltativo: parametri generici
  (elenco_porte);      // porte fra parentesi tonde
  /* dichiarazioni */
  // 1. dichiarazione delle porte:
  //   input, output (signed) , inout(+ eventuale larghezza bus)
```

```

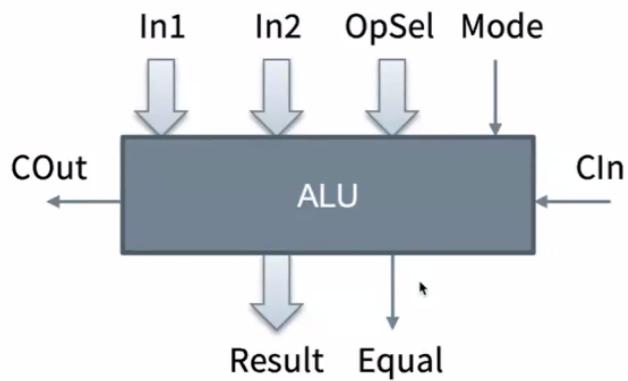
// 2. dichiarazione di segnali locali (wire, reg, logic, ecc.)
// 3. descrizione della logica con:
//   - primitive di porta      (gate-level)
//   - assign continui        (data-flow / RTL)
//   - blocchi always/initial (behavioral / RTL)
endmodule

```

▼ Esempio - Interfaccia del modulo ALU

Si progetti, tramite Verilog,
l'interfaccia del modulo
ALU in [figura](#).

Ipotizziamo che questa
ALU possa fare solo
operazioni di somma e
sottrazione e che gli
operatori `In1` e `In2` siano su
3 bit.



Visualizzazione di un modulo ALU

```

module ALU (Result, COut, Equal,      // prima le uscite, per convenzione
            In1, In2, OpSel, CIn, Mode); // seguete dagli ingressi.
                                    // si dichiarano esplicitamente il
                                    // tipo delle "porte" (ingressi/uscite)
                                    // e il parallelismo

output [4:0] Result; // porta d'uscita Result su 4 bit
                     // in caso di somme di numeri a 3 bit potremmo
                     // ottenere un numero su 4 bit

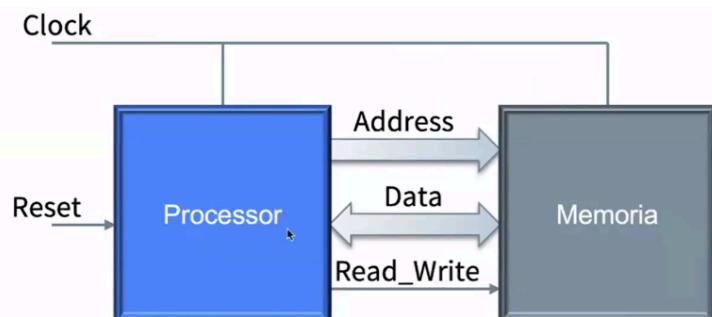
output COut;      // carry out
output Equal;     // se Equal = 1 ⇒ In1 = In2
input [2:0] In1;  // primo operando
input [2:0] In2;  // secondo operando
input [2:0] OpSel; // ipotizziamo ci siano 8 operazioni possibili
input CIn;
input Mode;       // modalità aritmetica (1) o logica (se 0)
// FINE INTERFACCIA
...
endmodule

```

▼ Esempio - Interfaccia di un processore

Si progetti, tramite Verilog, l'interfaccia del modulo Processore in [figura](#).

Tale processore ha la capacità di leggere o scrivere in memoria, ed è comandato da un clock.



Visualizzazione di un modulo Processore

```
module Processor (Read_Write, Data, Clock, Reset, Address);  
  
output Read_Write;  
output [19:0] Address;  
inout [15:0] Data; // è sia un input che un output!  
                    // Il processore può leggere o scrivere dati in/da memoria  
input Clock;  
input Reset;  
// FINE INTERFACCIA  
...  
endmodule
```



Normalmente, le porte di tipo `inout` dovrebbero essere pilotate con porte tri-state, in modo da evitare la creazione di conflitti.

Modellazione a livello di porta logica (GATE-LEVEL)

La modellazione a livello di porta logica rappresenta il circuito come un'[interconnessione esplicita di porte elementari](#) (AND, OR, XOR, NAND, flip-flop, ecc.). Ogni istanza di porta è contemporanea, e si ha quindi del parallelismo.

Usi tipici includono:

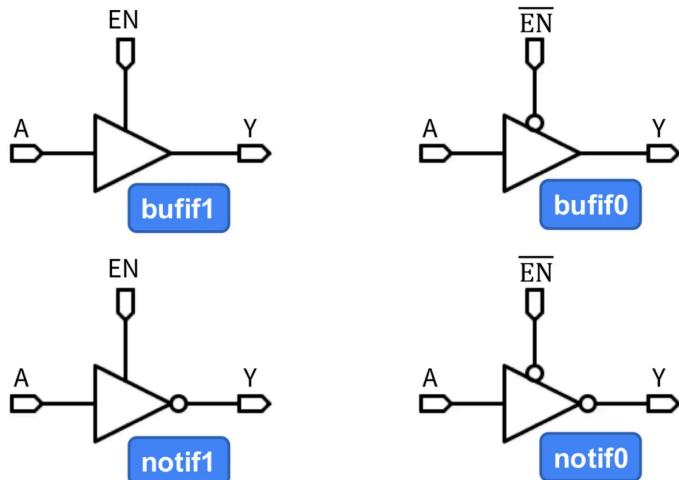
- checks post-sintesi;
- netlist generata dai tool;
- esercizi per piccole logiche.

Simula esattamente la rete fisica, e può includere ritardi di propagazione.

Porte logiche elementari in Verilog

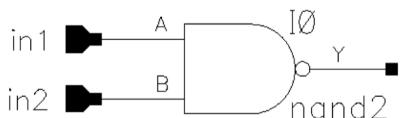
Le porte logiche di base sono:

- `and` ;
- `or` ;
- `not` ;
- `buf` ;
- `nand` ;
- `nor` ;
- `xor` ;
- `xnor` ;
- porte logiche tri-state.
 - `bufif1` , `bufif0` ;
 - `notif1` , `notif0` .

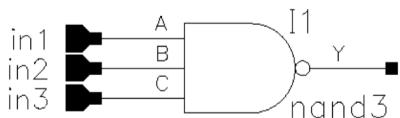


Rappresentazione di porte logiche tri-state

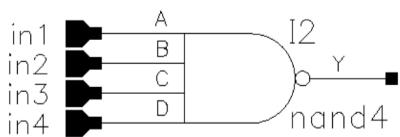
I pin delle porte logiche elementari sopra citate sono espandibili, come mostrato nella [figura](#).



`nand (y, in1, in2);`



`nand (y, in1, in2, in3);`



`nand (y, in1, in2, in3, in4);`

1 uscita

N ingressi

Espansione dei pin delle porte logiche

Valori di uscita delle porte logiche elementari

AND	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

OR	0	1	X	Z
0	0	1	X	X
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X

XOR	0	1	X	Z
0	0	1	X	X
1	1	0	X	X
X	X	X	X	X
Z	X	X	X	X

NAND	0	1	X	Z
0	1	1	1	1
1	1	0	X	X
X	1	X	X	X
Z	1	X	X	X

NOR	0	1	X	Z
0	1	0	X	X
1	0	0	0	0
X	X	0	X	X
Z	X	0	X	X

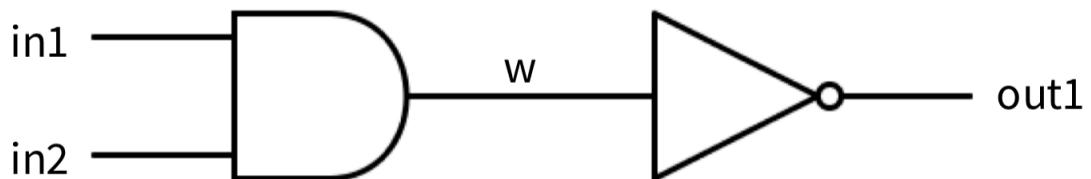
XNOR	0	1	X	Z
0	1	0	X	X
1	0	1	X	X
X	X	X	X	X
Z	X	X	X	X

BUF	0	1	X	Z
0	0			
1	1			
X	X			
Z	X			

NOT	0	1	X	Z
0	1			
1	0			
X	X			
Z	X			

Tabelle che rappresentano i valori di uscita delle porte logiche elementari

▼ Esempio - Modellazione a livello di porta logica



Circuito di riferimento da simulare: una porta NAND

```

module my_nand (out1, in1, in2);
    // inizio interfaccia
    output out1;
    input in1, in2; // input inseriti nella stessa definizione,
                    // in quanto sono entrambi da 1 bit
    // fine interfaccia

    // visto che devo collegare la porta AND e la porta NOT, mi serve un wire "w"
    wire w;
    and(w, in1, in2); // l'uscita della porta AND è il wire "w"
    not(out1, w);   // l'uscita della porta not è out1, l'ingresso è w
endmodule
    
```

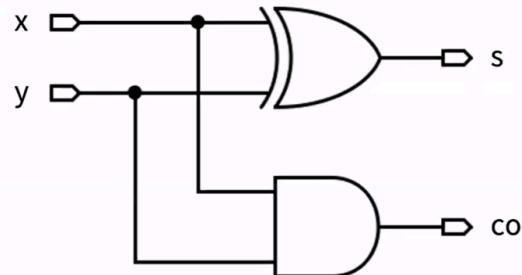
▼ Esempio - Semisommatore con porte logiche

Si realizzi l'Half Adder in [figura](#), tale che

$s = x + y$ e co sia il Carry Out.

In tal senso, occorre ricordare che vogliamo fare in modo che s sia:

$$0+0 = 0 \quad 1+0 = 1 \quad 0+1 = 1 \quad 1+1$$



Visualizzazione del modulo Half Adder

Questo si può ottenere facendo lo XOR tra x e y , mentre il Carry Out è ottenibile dall'AND di x e y .

```
module hadd(
    s, co, // uscite
    x, y // ingressi
);
    input x, y;
    output s, co;
    // FINE INTERFACCIA

    xor(s, x, y); // somma
    and(carry, x, y); // riporto
endmodule
```

Ritardi di porte logiche

I ritardi in Verilog sono preceduti da un cancelletto (`#`). Tipicamente, un ritardo associato a una porta logica è dichiarato come segue:

$$\#(\text{ritardo di propagazione low} \rightarrow \text{high}, \text{ritardo di propagazione h} \rightarrow \text{l}) = \\ \#(t_{\text{p}}^{\text{L} \rightarrow \text{H}}, t_{\text{p}}^{\text{H} \rightarrow \text{L}})$$

Si può essere ancora più specifici, fornendo il range dei ritardi minimi, tipici e massimi. Sui valori di $t_{\text{p}}^{\text{L} \rightarrow \text{H}}$ o di $t_{\text{p}}^{\text{H} \rightarrow \text{L}}$ possiamo definire i ritardi `minimi:tipici:massimi` .

È inoltre indispensabile specificare l'unità di misura di tali ritardi tramite la direttiva:

```
'timescale <unità_di_tempo>/<precisione_temporale>
// solitamente si specifica ad inizio file
```



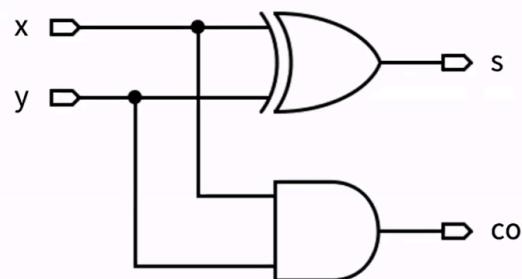
Si noti che tali ritardi sono utili solo nella simulazione comportamentale del circuito. Non hanno, infatti, alcun significato nella sintesi e nella realizzazione effettiva del circuito.

▼ Esempio - Semisommatore con porte logiche, con ritardi

Si realizzi un Half Adder del tipo in [figura](#), tale che $s = x + y$ e co sia il Carry Out.

Si sa che:

- $t_{\text{PXOR},\min}^{\text{L}\rightarrow\text{H}} = 2 \text{ ns}$;
- $t_{\text{PXOR},\max}^{\text{L}\rightarrow\text{H}} = 4 \text{ ns}$;
- $t_{\text{PAND}}^{\text{H}\rightarrow\text{L}} = 5 \text{ ns}$;
- $t_{\text{PAND}}^{\text{L}\rightarrow\text{H}} = t_{\text{PAND}}^{\text{H}\rightarrow\text{L}} = 3.6 \text{ ns}$.



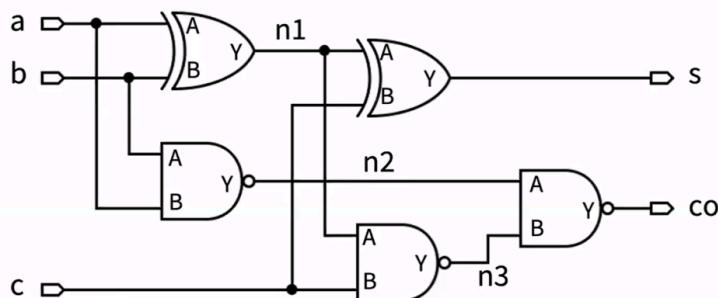
```
// dico che l'unità di tempo è 1 ns e che la precisione temporale
// è pari a 100 ps = 0.1 ns
'timescale 1ns/100ps
...
module hadd(
    s, co, // uscite
    x, y // ingressi
);
    input x,y;
    output s, co;
    // FINE INTERFACCIA

    xor # // definizione del ritardo
        (2:3:4,    // definizione del tempo minimo:tipico:medio del tp I→h XOR
         5)       // definizione del tempo tipico del tp h→I XOR
        (s, x, y); // somma
    // per intero, andrebbe scritto:
    // xor #(2:3:4, 5) (s,x,y);

    and #(3.567) // definizione del ritardo della porta AND
    // NOTA BENE: visto che la precisione è di 100 ps = 0.1 ns,
    // 3.567 ns viene arrotondato a 3.6 ns
    (co, x, y);
    // and #(3.6) (co,x,y);
endmodule
```

▼ Esempio - Utilizzo di sottomoduli

L'obiettivo è realizzare un Full Adder a 3 ingressi tramite porte logiche elementari.



Schema circuitale

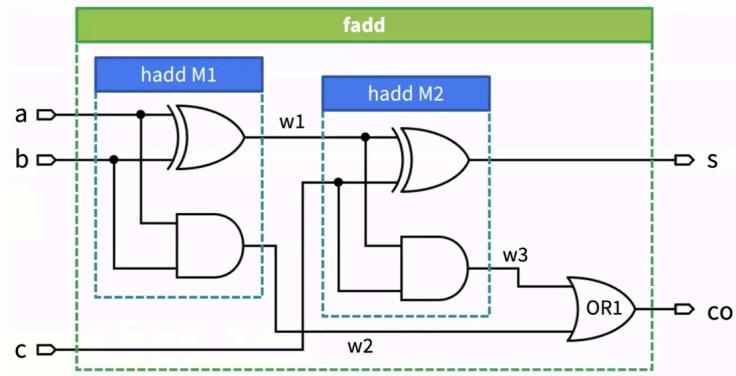
```
module fadd(s, co, a,b,c); // come di consuetudine: prima gli output, poi gli input
    output co, s;
    input a,b,c;
    // FINE INTERFACCIA

    wire n1, n2, n3;
    // FINE NET

    // l'ordine dei comandi sotto è stato volutamente "sparso" per rimarcare che,
    // con questa metodologia di programmazione, l'ordine delle istruzioni
    // NON è importante
    nand(n3,n1,c);
    xor(n1,a,b);
    nand(n2,a,b);
    xor(s,n1,c);
    nand(co,n2,n3);
endmodule
```

E' anche vero, però, che possiamo realizzare un Full Adder come interconnessione di Half Adder!

Quindi, avendo prima creato il modulo hadd, possiamo istanziarlo due volte per creare lo schema in figura.



Schema Full Adder definito come interconnessione di Half Adder

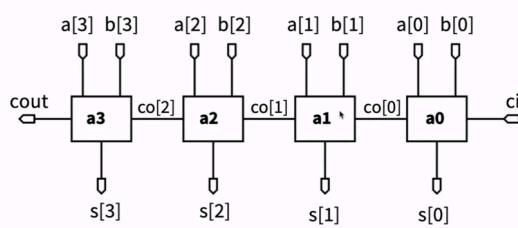
```
module fadd(co,s,a,b,c);
    output co,s;
    input a,b,c;

    wire w1, w2, w3;
    // si istanzia il modulo hadd dandogli un nome!
    // hadd (nome modulo) M1 (nome istanza) (componenti)
    hadd M1 (w1, w2, a,b); // la somma va su w1, il carry su w2
    hadd M2 (s, w3, w1, c); // la somma va su s, il carry su w3
    // N.B. serve avere un modulo hadd in un altro file!
    or OR1 (co, w3, w2);
endmodule
```

▼ Esempio - Sommatore a quattro bit con sottomoduli

- Implementazione gerarchica del modulo add4

- Quattro moduli fadd
- Ogni fadd consiste di
 - Due moduli hadd
 - Una porta logica elementare OR



```
1 module add4 (s, cout, ci, a, b);
2
3     input [3:0] a, b;
4     input ci;
5     output [3:0] s;
6     output cout;
7
8     wire [2:0] co;
9
10    fadd a0 (co[0], s[0], a[0], b[0], ci);
11    fadd a1 (co[1], s[1], a[1], b[1], co[0]);
12    fadd a2 (co[2], s[2], a[2], b[2], co[1]);
13    fadd a3 (cout, s[3], a[3], b[3], co[2]);
14
15 endmodule
```

Definizione del sommatore a quattro bit con sottomoduli

Modellazione RTL

La modellazione RTL consente di compattare la sintassi e "allontanarci" dalle porte logiche grazie a un meccanismo chiamato "assegnazione continua".

Tale meccanismo prevede di assegnare operazioni complesse di alto livello in una sola riga di codice, per poi lasciare al sintetizzatore logico l'utilizzo delle porte logiche necessarie.

A tale scopo, si possono implementare:

- Sommatori;
- Comparatori;
- Multiplexer;
- Moltiplicatori paralleli.

Operatore Assign

Ogni qualvolta in cui un operando presente nell'assegnazione di tipo `assign` cambia valore, al contempo viene "ricalcolata" la variabile che è stata dichiarata come `assign`.

```
assign #ritardo <nome_rete> = <espressione>;  
// il ritardo (opzionale) prende l'unità di misura dal `timescale a inizio file
```

La destinazione di un'assegnazione dovrebbe sempre essere un wire o una porta di uscita.

▼ Esempi - Operatore assign

```
assign out = a&b|c; // out = a AND b OR c  
// significa che out cambierà "istantaneamente" ogni volta che a, b o c variano  
// tutto questo è più comodo di utilizzare effettivamente porte AND e OR
```

```
assign eq = (a+b == c) // ci si chiede se la somma tra a e b sia uguale a c  
// se a+b == c, eq = 1; diversamente 0.
```

```
wire #10 inv = ~in; // la variabile inv cambierà dopo 10 timescale  
// da quando "in" varia  
// inv = NOT(in) con ritardo.  
// !!! questa sintassi è equivalente a:  
//   wire inv; assign #10 inv = ~in  
// posso comunque definire e assegnare un wire!
```

```
wire [7:0] c = a+b; // anche questa è un'assegnazione (assign) continua (implicita)
```



Occhio a evitare loop del tipo `assign a = b+a;`, poiché tale tipo di assegnazione non è sintetizzabile.

Operatori Verilog utili in assegnazioni continue

Operatori aritmetici		Operatori bit a bit		Operatori di spostamento	
<code>a + b</code>	Somma	<code>~a</code>	NOT bit a bit	<code>a << n</code>	Shift logico a sinistra
<code>a - b</code>	Differenza	<code>a & b</code>	AND bit a bit	<code>a >> n</code>	Shift logico a destra
<code>-a</code>	Cambio segno	<code>a b</code>	OR bit a bit	<code>a <<< n</code>	Shift aritmetico a sinistra
<code>a * b</code>	Moltiplicazione	<code>a ^ b</code>	XOR bit a bit	<code>a >>> n</code>	Shift aritmetico a destra
<code>a / b</code>	Divisione	<code>a ~^ b</code>	XNOR bit a bit	<code>{a, b}</code>	Concatenazione
<code>a % b</code>	Resto	<code>a ^~ b</code>	XNOR bit a bit		
Operatori relazionali		Operatori di riduzione		Operatori logici	
<code>a == b</code>	Uguale	<code>&a</code>	AND tutti i bit	<code>!a</code>	Negazione logica
<code>a != b</code>	Diverso	<code> a</code>	OR tutti i bit	<code>a && b</code>	AND logico
<code>a < b</code>	Minore	<code>^a</code>	XOR tutti i bit	<code>a b</code>	OR logico
<code>a > b</code>	Maggiore	<code>~&a</code>	NAND tutti i bit	<code>sel?a:b</code>	Condizionale ternario
<code>a <= b</code>	Minore o uguale	<code>~ a</code>	NOR tutti i bit		
<code>a >= b</code>	Maggiore o uguale	<code>~^a</code>	XNOR tutti i bit		

Operatori Verilog supportati nelle assegnazioni continue



La tilde (`~`) è ottenuta su Windows con `ALT+126`.

▼ Esempi - Operatori Verilog

```
assign c = a+b; // c sarà (in automatico???) di N+1 bit se a e b sono su N bit
```

```
assign c = a>b; // c starà su 1 bit; mi restituisce un valore logico
```

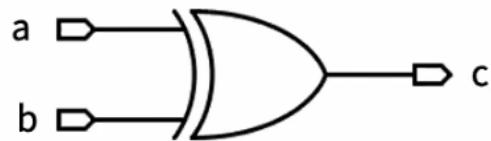
```
assign c = a||b; // a e b sono entrambi su 1 bit, così come c e si fa l'OR logico
// se a o b fossero stati su N bit, sarebbero stati gestiti come 0 solo se
// fossero 0, altrimenti a e b sarebbero stati ridotti a 1 logico
```

```
assign c = a&b; // a,b,c su N bit, in quanto si fa l'AND bit a bit
```

```
assign c = &a; // AND di tutti i bit di a.
```

▼ Esempio - XOR con ritardo (RTL)

Si realizzi la porta logica XOR, come in [figura](#).



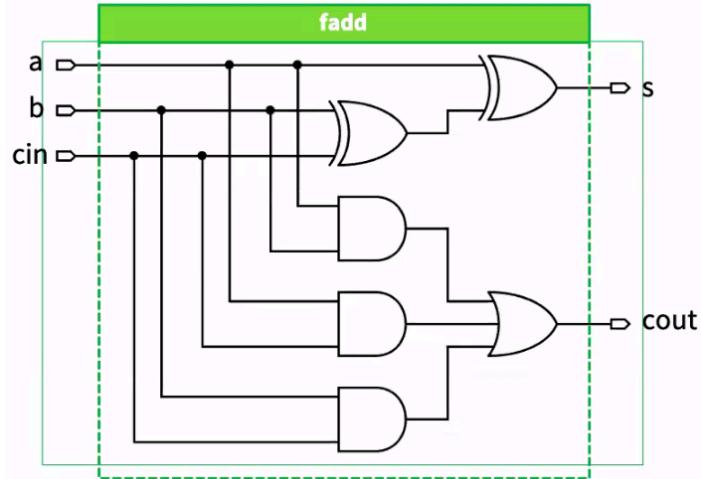
Porta XOR

```
module my_xor(c, a, b); // al solito, prima uscita, poi ingressi
    output c;
    input a,b;

    assign #2 c = a ^ b; // a XOR (bit a bit) b con ritardo di 2 timescale
endmodule
// SI RICORDI CHE TALE RITARDO SI USA SOLO IN SIMULAZIONE (NON IN SINTESI)!
```

▼ Esempio - Sommatore completo

Si implementi un Full Adder generico, come in [figura](#).



Schema di un Full Adder

```
module fadd (cout, s, a, b, cin)
    output cout, s;
    input a,b,cin;

    assign s = (b ^ cin) ^ a;

    assign cout = (a&b) | (a&cin) | (b&cin);
endmodule
```

▼ Esempio - Sommatore a 4 bit

Si implementi un sommatore tale che gli operandi siano su 4 bit.

```
module fh4(s, cout, a,b,cin);
    output [3:0] s; // somma su 4 bit
    output cout; // carry out su un bit
    input [3:0] a,b; // operandi di ingresso su 4 bit
    input cin; // carry in di ingresso su un bit

    // Si potrebbe fare, come prima utilizzo di XOR e (AND + OR) oppure,
    // come in questo caso, sfruttare l'operatore di concatenazione
    assign {cout,s} = a+b+cin; // a+b+cin genera un output a 5 bit di cui l'MSB
                                // sarà il carryout
```

Realizzazione di moduli parametrici

Il modulo "generico", definito come parametrico, ha come obiettivo quello di farci fornire, dall'utente, un parametro che possa servirci per scopi quali:

- la dichiarazione di un vettore con le corrette dimensioni;
- l'istanziazione di un modulo che prevede "N" (con N passato come parametro) iterazioni.

La sintassi per l'utilizzo di un modulo parametrico è la seguente:

```
<nome_modulo> #(elenco_parametri) <nome_istanza> (elenco_porte);
```

La sintassi per la dichiarazione di un parametro è, invece, la seguente:

```
parameter <nome_parametro>; // se non si vuole dare un valore di default al parametro
parameter <nome_parametro> = <valore_default>; // se si vuole dare un valore di default
```

▼ Esempio - Sommatore a N bit parametrico

Vogliamo realizzare un sommatore parametrico tale che il numero N di bit degli operandi sia passato dall'utente.

```
module #(parameter N = 2) adder (cout, s, a, b, cin);
    // specifichiamo un parametro "N", di default uguale a 2
    input [N-1:0] a,b; // specifico che gli operandi sono su N bit
    input cin;
    output [N-1:0] s; // specifico che la somma (senza cout) sta su N bit
    output cout;
```

```
assign {cout, s} = a+b+cin;  
endmodule
```

Vediamo ora un altro modo (meno leggibile) di implementare questo codice.

```
module adder (cout, s, a, b, cin);  
    parameter N = 2; // specifichiamo un parametro "N", di default uguale a 2  
    input [N-1:0] a,b; // specifico che gli operandi sono su N bit  
    input cin;  
    output [N-1:0] s; // specifico che la somma (senza cout) sta su N bit  
    output cout;  
  
    assign {cout, s} = a+b+cin;  
endmodule
```

Vediamo ora un po' di utilizzi del modulo parametrico `adder`.

```
// in un altro file!  
adder #(8) adder_8 (co, s, a,b,ci)      // sommatore a 8 bit  
adder adder_2 (co, s, a, b, ci)        // sommatore a 2 bit (uso il default)  
adder #(12) adder_12 (co, s, a, b, ci)  // sommatore a 12 bit  
adder #(.N(8)) adder_8 (co, s, a, b, ci) // specifico che mi riferisco a N
```



Come faccio a specificare il ritardo di un modulo da me creato?

Nel caso di un sommatore a 8 bit con 4 ns di ritardo NON si può fare una chiamata di questo tipo:

```
'timescale 1ns/100ps  
...  
adder #(8) #4 adder_8 (co, s, a,b,ci) // sommatore a 8 bit
```

I ritardi in Verilog non possono essere specificati su moduli dichiarati da noi in maniera diretta (come facevamo con le porte elementari), possiamo rendere il ritardo parametrico, come segue:

```
module adder (cout, s, a, b, cin);  
    parameter N = 2; // specifichiamo un parametro "N", di default uguale a 2  
    parameter delay = 0;  
    input [N-1:0] a,b; // specifico che gli operandi sono su N bit  
    input cin;  
    output [N-1:0] s; // specifico che la somma (senza cout) sta su N bit  
    output cout;  
  
    assign #(delay) {cout, s} = a+b+cin;  
    // si noti che, nel caso di ritardo costante, avremmo potuto inserire  
    // assign #4 {cout, s} ....; Poichè stiamo dando un parametro, però  
    // occorre metterlo tra parentesi per aiutare il parser.  
endmodule
```

Successivamente, si può effettuare una chiamata come segue:

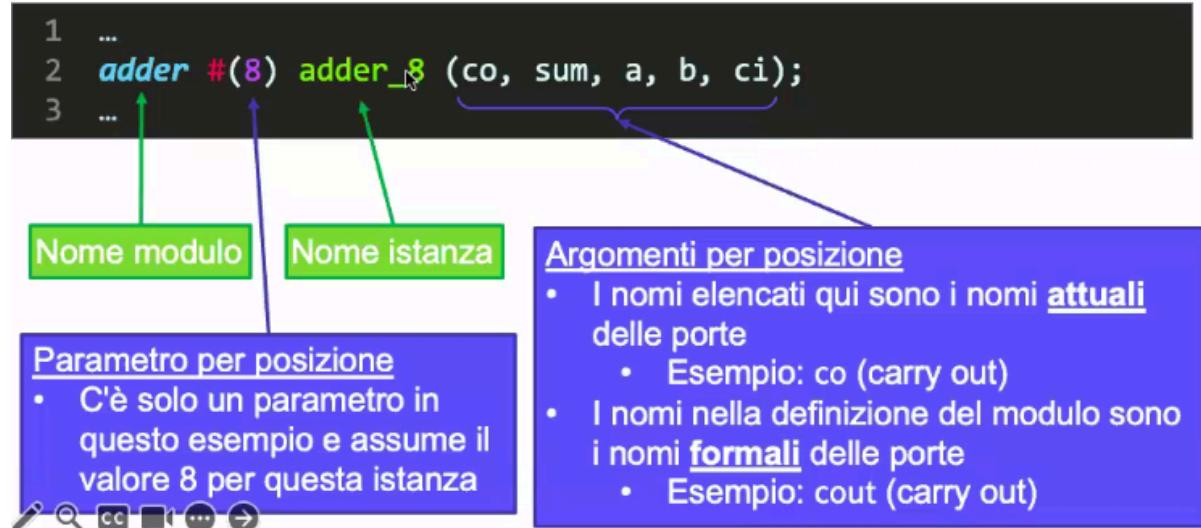
```
adder #(8, 4) adder_8_4 (co, s, a, b, cin); // che funziona correttamente  
adder #(.N(8), .delay(4)) adder_8_4 (co, s, a, b, cin); // più leggibile  
adder #(.delay(4), .N(8)) adder_8_4 (co, s, a, b, cin); // più leggibile
```

Si presta attenzione che tali informazioni non vengono dal docente, ma sono ricavate dalla rete e, pertanto, l'assegnazione di un ritardo ad assign #(delay) potrebbe non funzionare correttamente con tutti i compilatori!

Passaggio di argomenti e parametri

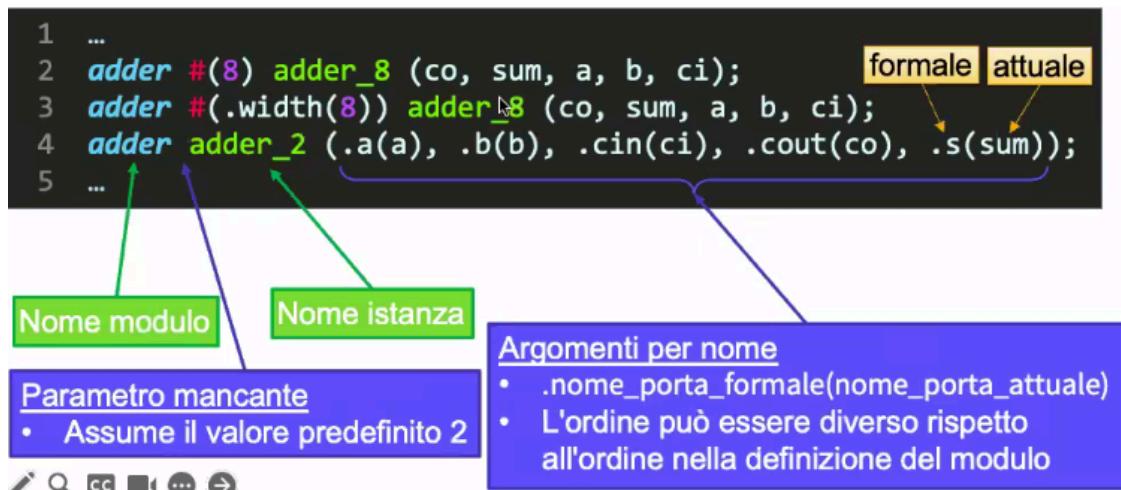
Quando chiamiamo un modulo possiamo scegliere di passare i parametri "attuali" al modulo, il quale contiene i parametri "formali" per posizione (come in C) o per nome (in maniera tale da rendere l'ordine ininfluente e rendere il codice più leggibile).

Data la seguente funzione, un esempio di passaggio di parametri per posizione è il seguente:



Modulo parametrico implementato in precedenza, in cui tutti gli argomenti e i parametri sono passati per posizione

Un esempio di passaggio di parametri per nome è il seguente:



Modulo parametrico implementato in precedenza, dove si fa utilizzo in riga 3 di argomenti e parametri per nome.

Si noti che width, da noi, è stato chiamato N

Si possono, inoltre, lasciare delle porte non collegate come segue:

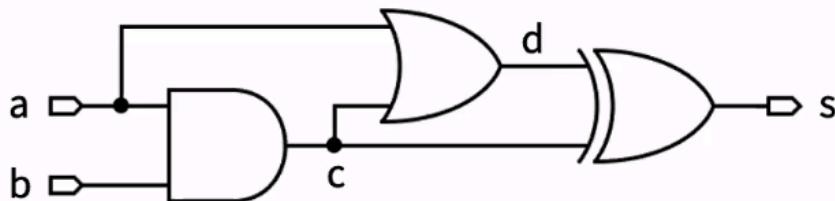
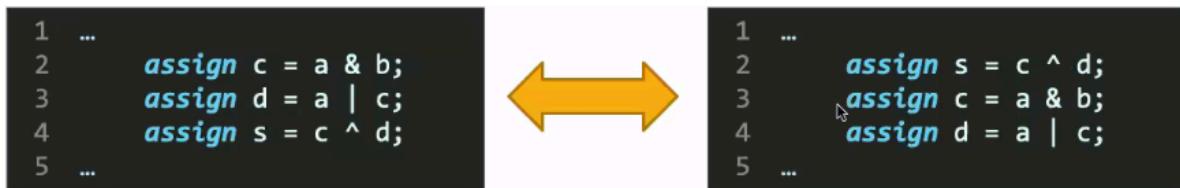
```
adder adder_2 (.s(sum), .a(a), .b(b), .cin(ci));
adder #(2) adder_2 (, sum, a, b, ci);
```

Se un'uscita non ci serve, possiamo decidere di non collegarla a niente. Ciò non si può fare con gli ingressi!

Ordine delle istruzioni

L'ordine delle istruzioni, come detto nell'introduzione, non è importante in RTL, poiché:

- tutte le istruzioni sono eseguite in parallelo;
 1. le espressioni a destra dell'uguale vengono prima calcolate tutte insieme;
 2. vengono assegnate tutte insieme ai wire a sinistra dell'uguale;
- sono sempre equivalenti a uno schema a blocchi.



Equivalenza tra codici anche con ordine delle istruzioni differente!

Blocco generate con for e if

Come detto in [precedenza](#), si può parametrizzare un modulo in maniera tale che si iteri N volte.

In questo caso, il blocco `generate` ci consente di effettuare cicli e blocchi condizionali solo per strutture "ripetitive" (si tratta di una esecuzione durante la fase di compilazione, non in runtime), per le quali dovremmo usare tante righe di codice per istanziare uno stesso comando, come nel seguente esempio:

```

parameter N; // generica variabile parametrica che descrive il numero delle iterazioni
genvar i; // variabile utile nel ciclo for

generate
    for (i = 0; i < N; i++) begin
        if(i == 0)
            istruzione1;
        else if (i == N-1)
            istruzione2;
        else begin;
            istruzione3;
            istruzione4;
        end
    end
endgenerate

```

```
// Si noti come:  
// - con una istruzione si comporta come in C, ovvero non  
// sono necessarie le graffe;  
// - con più istruzioni sotto un blocco condizionale o un for  
// sono necessarie le "graffe", definite da begin ed end.
```

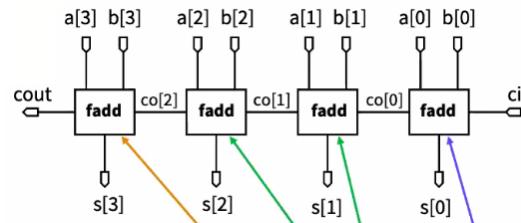


Nei blocchi `generate` non è possibile dare nomi ai moduli istanziati in maniera "semplice", ma avrebbe comunque poco senso farlo ([vedi esempio](#)).

Si noti che non è un loop iterativo che assicura di realizzare istruzioni in sequenza, ma è come se servisse solo a replicare il codice.

▼ Esempio - Sommatore completo su N bit con generate

Si parta dal sommatore in figura a 4 bit, per poi estenderlo a N bit.



In blu la prima istanza, in verde le intermedie, in arancione l'ultima istanza

```
module adder (s, cout, ci, a, b);  
  parameter N = 4; // di default realizziamo un sommatore a 4 bit, come in figura  
  input [N-1:0] a,b;  
  input ci;  
  output [N-1:0] s;  
  output cout;  
  
  wire [N-2:0] co; //si utilizza un vettore che contenga i vari wire  
  // sono in totale N-1 bit in quanto l'N-esimo è cout  
  genvar i; // variabile intera utile per l'iterazione  
  
  generate  
    for (i = 0; i < N; i++) begin  
      if(i == 0)  
        fadd(co[i], s[i], a[i], b[i], ci); // ISTANZA BLU  
      else if (i == N-1)
```

```

fadd(co[i], s[i], a[i], b[i], co[i-1]); // ISTANZA ARANCIONE
else
    fadd(cout, s[i], a[i], b[i], co[i-1]); // ISTANZA VERDE
end
endgenerate
endmodule

// Si noti che nelle chiamate al modulo "fadd" non si
// specifica il nome dell'istanza.
//
// Nel caso del generate, ciò non si può fare perché
// si andrebbe, negli intermedi, a istanziare moduli
// con lo stesso nome.

```

Modellazione comportamentale

La modellazione comportamentale è ottenuta tramite blocchi `always`, i quali definiscono una sorta di processi, ovvero funzioni le cui istruzioni sono **eseguite in sequenza, ma nello stesso istante**.

Per fare un paragone, un blocco `always` è una specie di chiamata ricorsiva che viene effettuata ogni qual volta i parametri "sensibili" (che devono essere `reg` o `wire`) del blocco `always` vengono modificati **al di fuori del blocco `always` stesso**.

Con tale paradigma si possono modellare sia circuiti combinatori, sia sequenziali.

Ecco la sintassi di un blocco `always`:

```

always @(<elenco_sensibilità>) begin
    ...
end

// Quando uno o più segnali presenti nell'elenco di sensibilità
// cambia valore, il blocco always viene eseguito.

```



Tutte le uscite di un blocco `@always` devono essere di tipo `reg`:

- le assegnazioni usano solo `=` o `<=`;
- mantengono memoria dell'ultima assegnazione, al contrario dei `wire` (che, in uscita, sono frutto di assegnazioni "continue").

Il tipo `reg` assomiglia a un registro, ma non ne rappresenta necessariamente uno.

Dentro i blocchi `always` possono essere utilizzati i costrutti `if` e `case`, grazie al fatto che le operazioni vengono eseguite **in ordine**.

Always con circuiti combinatori

La caratteristica dei circuiti combinatori è tale per cui l'uscita dipende, in ogni istante, dal valore degli ingressi e, pertanto, il blocco `always` sarà del tipo:

```
always @(ingresso1, ingresso2, ..., ingressoN) begin  
    ...  
end
```

// L'elenco di sensibilità è dato dagli ingressi presenti nel circuito combinatorio.

Per essere sicuri di valutare tutti gli ingressi in una logica combinatoria si può anche utilizzare tale sintassi:

```
always @(*) begin // Viene eseguito sulla variazione di:  
    out1 = a+b+c+d; // sulla variazione di uno tra a,b,c,d  
    out2 = f*e+3; // sulla variazione di uno tra f ed e.  
end
```

// Il blocco always viene eseguito ogni volta che un
// secondo membro di un'assegnazione cambia valore.

▼ Esempio - Porta AND

Si realizzi la porta AND in figura, tramite il blocco

`always`.



Porta AND

```
reg y; // uscita sempre di tipo reg  
wire a,b; // ingressi possono essere wire o reg.  
always @(a or b) begin  
    y = a&b;  
end
```

Tale codice è identico a scrivere il seguente codice:

```

reg y; // uscita sempre di tipo reg, mantiene memoria dell'ultima assegnazione
wire a,b; // ingressi possono essere wire o reg.
always @(*)
    y = a&b; // con una sola riga di codice posso evitare di usare begin e end

```

Esso è ancora uguale a:

```

wire y, a,b; // il wire presuppone un'assegnazione continua che esiste sempre!
assign y = a & b; // qui possiamo usare un wire come uscita!

// Reg si potrebbe usare come secondo membro di un'operazione
// di assign ma non come primo membro

```

Always con circuiti sequenziali

Nel caso di circuiti sequenziali è necessario far sì che il blocco `always` sia attivato solo durante il fronte di salita/discesa del clock tramite le parole chiave:

- `posedge`: fronte di salita;
- `negedge`: fronte di discesa.

```

always @(posedge clock) // esempio 1
always @(negedge clock or reset) // esempio 2, FF attivo su fronte di discesa
// con reset asincrono

// Il clock è un input, non una parola chiave (wire o input che sia)

```

▼ Esempio - Porta AND + FF

Quando arriva il fronte di salita del clock, si vuole che un flip flop memorizzi l'AND tra due ingressi a e b.

```

reg a,b,ff;
always @(posedge clock) // ogni fronte di salita del clock voglio che il ff memorizzi
    ff = a & b;

```

Assegnazioni bloccanti



È importante ricordare che, all'interno del corso, utilizzeremo **solo l'assegnazione bloccante**.

Le assegnazioni bloccanti hanno un effetto immediato (come in C):

```
always @(a or b) begin // 1
    a = b;           // 2
    b = a;           // 3
end               // 4
```

// Qualora a o b (o entrambi) abbiano una variazione di valore,
// il blocco always NON verrà eseguito.

1. supponiamo che, durante la prima esecuzione del blocco `always`, si abbia `a = 0` e `b = 1`, come in tabella;

	1° riga di codice	2° riga di codice	3° riga di codice	4° riga di codice
a	0	1	1	1
b	1	1	1	1

2. alla seconda riga di codice otteniamo `a = b` e, quindi, `a` passa da 0 a 1, mentre `b` mantiene il suo valore precedente (pari a 1);
3. alla terza riga di codice, `a` mantiene il suo valore precedente (1) e si esegue `b = a`, ma `a = 1`, quindi `b = a = 1`;
4. si raggiunge la fine del blocco `always`: `a` e `b` mantengono i loro valori precedenti (1 e 1).



Ci si potrebbe chiedere come mai non si rientri nel blocco `always`, dato che `a` ha cambiato valore.

Questo non avviene perché il blocco `always` viene rieseguito se e solo se `a` o `b`, o addirittura entrambi subiscono variazioni, ma esse devono avvenire **al di fuori** del blocco `always` stesso.

Visto che le variazioni ai parametri di sensibilità (`a` e `b`) sono solo interne al blocco `always`, quest'ultimo **non viene nuovamente eseguito!**

Assegnazioni non bloccanti

Le assegnazioni non bloccanti avvengono solo al termine dell'esecuzione del blocco `always` e, in tal senso, comporterebbero una nuova esecuzione del blocco `always` nel caso di variazione.

```
always @(a or b) begin // 1
    a <= b;           // 2 assegna a = b alla fine del blocco always
    b <= a;           // 3 assegna b = a alla fine del blocco always
end               // 4
```

```
// qualora a o b (o entrambi) abbiano una variazione di
// valore, il blocco always verrà eseguito.
```

- supponiamo che, durante la prima esecuzione del blocco `always`, si abbia `a = 0` e `b = 1`, come in tabella;

1° esecuzione	1° riga di codice	2° riga di codice	3° riga di codice	4° riga di codice	Valore futuro
a	0	0	0	1	1
b	1	1	1	0	0

- viene eseguita l'istruzione `a <= b` e, quindi, il **valore futuro** di `a` sarà pari al valore corrente di `b` (1) solo alla fine del blocco `always`;
- viene eseguita l'istruzione `b <= a` e, quindi, il valore futuro di `b` sarà pari al valore corrente di `a` (0) solo alla fine del blocco `always`;
- raggiunta la fine del blocco `always` vengono assegnati i valori futuri di `a` e `b`.



Poiché `a` e `b` sono variate al di fuori del blocco `always` e sono nella lista di sensibilità, il blocco si riattiva e viene nuovamente eseguito.

2° esecuzione	1° riga di codice	2° riga di codice	3° riga di codice	4° riga di codice	Valore futuro
a	1	1	1	0	0
b	0	0	0	1	1

Poiché, nuovamente, `a` e `b` sono variate e si trovano nella lista di sensibilità, il blocco `always` viene nuovamente eseguito (indefinitivamente, come in [figura](#)).

1° esecuzione	2° esecuzione	3° esecuzione	4° esecuzione	5° esecuzione	6° esecuzione	7° esecuzione	8° esecuzione	9° esecuzione	Futuro
1 2 3 4	1 2 3 4	1 2 3 4	1 2 3 4	1 2 3 4	1 2 3 4	1 2 3 4	1 2 3 4	1 2 3 4	1 2 3 4
a 0 0 0 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 1 0	a 0 0 0 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 1 0	a 0 0 0 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 1 0	a 0 0 0 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 1 0	a 0 0 0 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 1 0	a 0 0 0 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 1 0	a 0 0 0 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 1 0	a 0 0 0 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 1 0	a 0 0 0 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 1 0	a 0 0 0 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 1 0
b 1 1 1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 0	b 1 1 1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 0	b 1 1 1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 0	b 1 1 1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 0	b 1 1 1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 0	b 1 1 1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 0	b 1 1 1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 0	b 1 1 1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 0	b 1 1 1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 0	b 1 1 1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 0

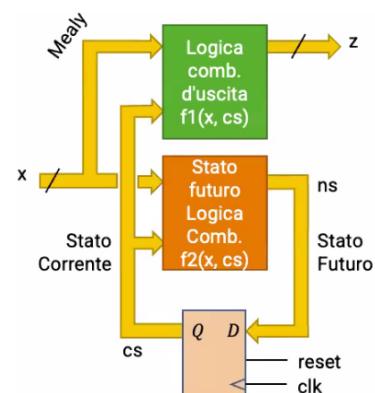
Tipicamente questo comportamento non è desiderato, anche se somiglia a un clock

Esempio - Realizzazione di un circuito sequenziale Mealy (FSM)

Come noto, una macchina di Mealy è tale per cui il valore dell'uscita dipende sia dal valore corrente degli ingressi, sia dallo stato corrente.

Si vuole, quindi, realizzare una macchina di Mealy come in [figura](#).

Si noti che il FF-D è attivo sul fronte di salita.



Macchina di Mealy

```

module MealyFSM (z, clk, reset, x);
    output reg z;
    input clk, reset, x;
    // si poteva anche fare output z; reg z;
    reg ns; // next state (sono come wire ma necessariamente devono essere dichiarati reg)
    reg cs; // current state (come wire ma serve dichiararlo come reg poichè output).

    /// LE REG SONO NECESSARIE PER USARLE COME "uscita" dei blocchi always

    // GESTIAMO LA VARIAZIONE DEGLI STATI (dipendente da clock e reset)
    always @(posedge clock) // attivo sul fronte di salita del clock
        or
        posedge reset) // reset asincrono: se reset passa da 0→1 allora si entra
    begin: stateReg // etichetta utile per documentazione, nome del blocco always
        if (reset)
            cs = 0;
        else
            cs = ns; // current state = next state... aggiorno lo stato corrente
    end
    always @(x or cs) begin: outputFunction
        // la lista di sensibilità è data dall'ingresso e dallo stato corrente.
        z = f1(x,cs); // l'uscita dipende da ingresso e stato corrente.
    end

    always @(x or cs) begin: nextStateTransition
        ns = f2(x, cs); // il next state dipende da ingresso e stato corrente
    end
endmodule

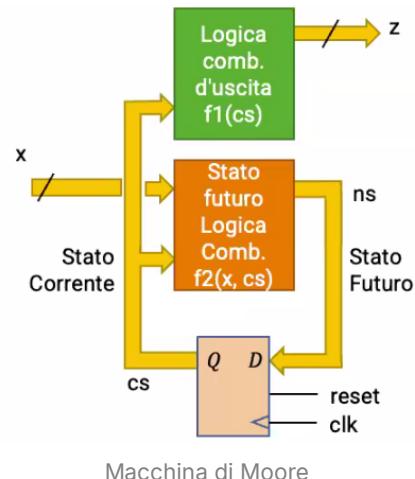
```

Esempio - Realizzazione di un circuito sequenziale Moore (FSM)

Come noto, una macchina di Moore è tale per cui il valore dell'uscita dipende solo dallo stato corrente.

Si vuole, quindi, realizzare una macchina di Moore come in [figura](#).

Si noti che il FF-D è attivo sul fronte di salita e che il reset è sincrono (cioè deve arrivare insieme al clock).



Macchina di Moore

```
module MealyFSM (z, clk, reset, x);
    output z;
    input clk, reset, x;
    reg ns; // next state (sono come wire ma necessariamente devono essere dichiarati reg)
    reg cs; // current state (come wire ma serve dichiararlo come reg poichè output).
    reg z;
    /// LE REG SONO NECESSARIE PER USARLE COME "uscita" dei blocchi always

    // GESTIAMO LA VARIAZIONE DEGLI STATI (dipendente da clock)
    always @(posedge clk) // vogliamo un reset sincrono!
    begin: stateReg
        if (reset) // Il reset deve arrivare durante la transizione del clock!
            cs = 0;
        else begin
            cs = ns;
        end
    end
    always @(cs) begin: outputFunction
        // la lista di sensibilità è data dal solo stato corrente.
        z = f1(cs); // l'uscita dipende da ingresso e stato corrente.
    end

    always @(x or cs) begin: nextStateTransition
        ns = f2(x, cs); // il next state dipende da ingresso e stato corrente
    end
endmodule
```



Modellazione strutturale

▼ Creatore originale: @Gianbattista Busonera

Definizione di un modulo

[Esempio - Interfaccia del modulo ALU](#)

[Esempio - Interfaccia di un processore](#)

La modellazione strutturale descrive la struttura gerarchica del circuito tramite istanze di moduli (che possono a loro volta essere RTL, gate-level, etc.), è simile a disegnare uno schematico, ma con una descrizione testuale.

L'elemento base è `module endmodule` per ogni blocco, le cui istanze si collegano con net (tipicamente `wire`).

```
module <nome> ([lista delle porte]);
    // contenuti del modulo
endmodule

// si noti come un modulo può omettere la
// lista delle porte
module <nome>;
    // ...
endmodule
```

Tutte le connessioni sono simultanee, e quindi l'ordine delle istanze non conta.

Il loro uso tipico è:

- integrare blocchi IP (Intellectual Property block), ovvero blocchi di logica già progettati, testati ed ottimizzati, che è possibile comprare, licenziare o riutilizzare all'interno di un progetto invece che implementarlo da zero;
- costruire sistemi gerarchici (per esempio, CPU + periferiche).

Nella pratica moderna, il 90% del codice è scritto in **RTL o dataflow**, per poi lasciare al sintetizzatore la generazione gate-level finale. Si usa, invece, la modellazione **strutturale** per mettere insieme i vari moduli.

Definizione di un modulo

Un modulo è un componente riutilizzabile.

In Verilog, ogni blocco di circuito è racchiuso tra le parole chiave `module` ed `endmodule`.

All'interno del modulo si dichiarano:

- elenco degli ingressi e delle uscite (le uscite vanno inserite prima per convenzione);
- le caratteristiche delle porte (ingressi e uscite, eventualmente signed) e il loro parallelismo;
- eventuale dichiarazione di parametri (costanti di cui possiamo modificare il valore), che ci consentono di rendere parametrico il nostro circuito;
- tramite la parola chiave `'include` possiamo inserire nel nostro modulo altri componenti definiti in precedenza;
- segue la parte più corposa del nostro modulo, che analizzeremo man mano nella trattazione.

```
module nome_modulo
  #(parametri opzionali) // facoltativo: parametri generici
  (elenco_porte);      // porte fra parentesi tonde
  /* dichiarazioni */
  // 1. dichiarazione delle porte:
```



Visualizzazione della struttura interna di un modulo

```

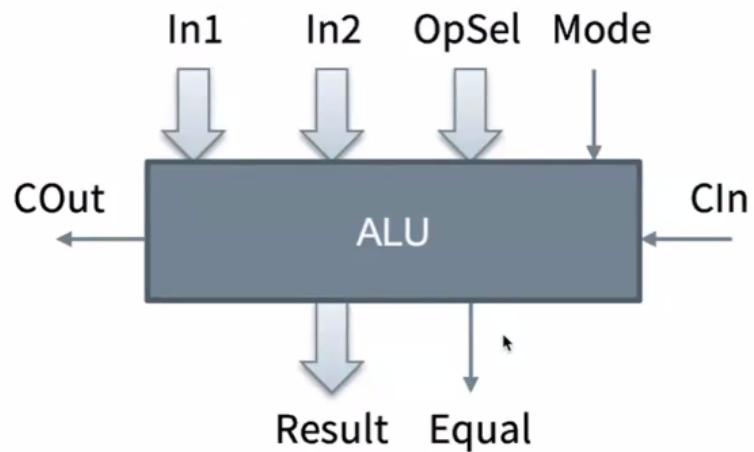
// 1. input, output (signed) , inout(+ eventuale larghezza bus)
// 2. dichiarazione di segnali locali (wire, reg, logic, ecc.)
// 3. descrizione della logica con:
//   - primitive di porta      (gate-level)
//   - assign continui        (data-flow / RTL)
//   - blocchi always/initial (behavioral / RTL)
endmodule

```

▼ Esempio - Interfaccia del modulo ALU

Si progetti, tramite Verilog, l'interfaccia del modulo ALU in [figura](#).

Ipotizziamo che questa ALU possa fare solo operazioni di somma e sottrazione e che gli operandi **In1** e **In2** siano su 3 bit.



Visualizzazione di un modulo ALU

```

module ALU (Result, COut, Equal,      // prima le uscite, per convenzione
            In1, In2, OpSel, CIn, Mode); // seguite dagli ingressi.
                                    // si dichiarano esplicitamente
                                    // tipo delle "porte" (ingressi/
                                    // e il parallelismo

output [4:0] Result; // porta d'uscita Result su 4 bit
                     // in caso di somme di numeri a 3 bit potremmo
                     // ottenere un numero su 4 bit

output COut;      // carry out
output Equal;     // se Equal = 1 ⇒ In1 = In2

```

```

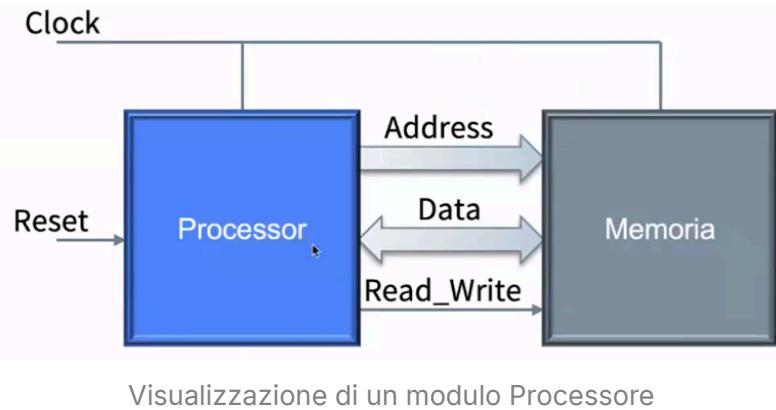
input [2:0] In1; // primo operando
input [2:0] In2; // secondo operando
input [2:0] OpSel; // ipotizziamo ci siano 8 operazioni possibili
input CIn;
input Mode; // modalità aritmetica (1) o logica (se 0)
// FINE INTERFACCIA
...
endmodule

```

▼ Esempio - Interfaccia di un processore

Si progetti, tramite Verilog, l'interfaccia del modulo
Processore in [figura](#).

Tale processore ha la capacità di leggere o scrivere in memoria, ed è comandato da un clock.



```

module Processor (Read_Write, Data, Clock, Reset, Address);

output Read_Write;
output [19:0] Address;
inout [15:0] Data; // è sia un input che un output!
                    // Il processore può leggere o scrivere dati in/da memoria
input Clock;
input Reset;
// FINE INTERFACCIA
...
endmodule

```



! Normalmente, le porte di tipo `inout` dovrebbero essere pilotate con porte tri-state, in modo da evitare la creazione di conflitti.



Modellazione a livello di porta logica (GATE-LEVEL)

▼ Creatore originale: @Gianbattista Busonera

[Porte logiche elementari in Verilog](#)

[Valori di uscita delle porte logiche elementari](#)

[Esempio - Modellazione a livello di porta logica](#)

[Esempio - Semisommatore con porte logiche](#)

[Ritardi di porte logiche](#)

[Esempio - Semisommatore con porte logiche, con ritardi](#)

[Esempio - Utilizzo di sottomoduli](#)

[Esempio - Sommatore a quattro bit con sottomoduli](#)

La modellazione a livello di porta logica rappresenta il circuito come un'**interconnessione esplicita di porte elementari** (AND, OR, XOR, NAND, flip-flop, ecc.). Ogni istanza di porta è contemporanea, e si ha quindi del parallelismo.

Usi tipici includono:

- checks post-sintesi;
- netlist generata dai tool;
- esercizi per piccole logiche.

Simula esattamente la rete fisica, e può includere ritardi di propagazione.

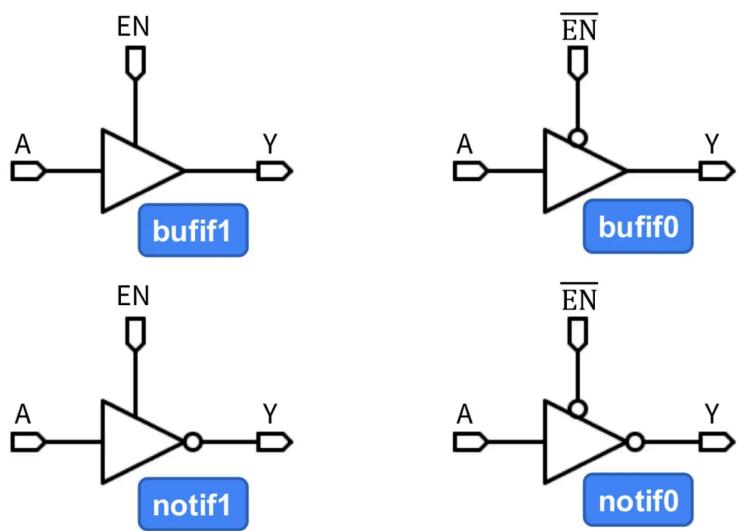
Porte logiche elementari in Verilog

Le porte logiche di base

sono:

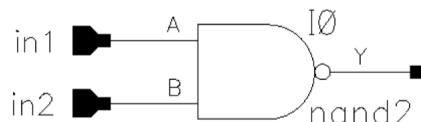
- `and` ;
- `or` ;
- `not` ;
- `buf` ;
- `nand` ;
- `nor` ;
- `xor` ;
- `xnor` ;
- porte logiche tri-state.

- `bufif1` , `bufif0` ;
- `notif1` , `notif0` .

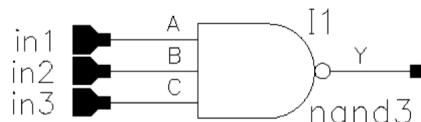


Rappresentazione di porte logiche tri-state

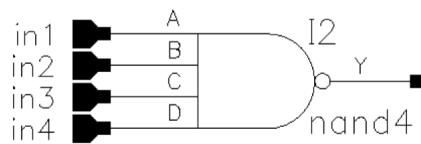
I pin delle porte logiche elementari sopra citate sono espandibili, come mostrato nella [figura](#).



nand (y, in1, in2);



nand (y, in1, in2, in3);



nand (y, in1, in2, in3, in4);

1 uscita

N ingressi

Espansione dei pin delle porte logiche

Valori di uscita delle porte logiche elementari

AND	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

OR	0	1	X	Z
0	0	1	X	X
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X

XOR	0	1	X	Z
0	0	1	X	X
1	1	0	X	X
X	X	X	X	X
Z	X	X	X	X

NAND	0	1	X	Z
0	1	1	1	1
1	1	0	X	X
X	1	X	X	X
Z	1	X	X	X

NOR	0	1	X	Z
0	1	0	X	X
1	0	0	0	0
X	X	0	X	X
Z	X	0	X	X

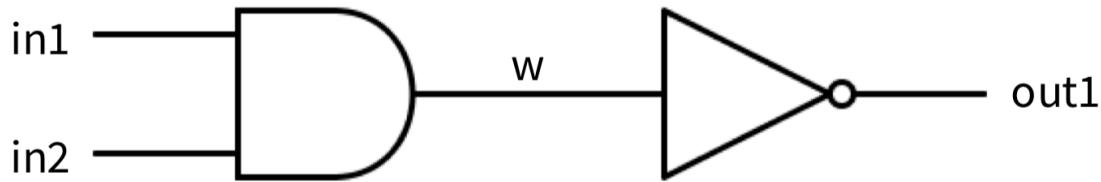
XNOR	0	1	X	Z
0	1	0	X	X
1	0	1	X	X
X	X	X	X	X
Z	X	X	X	X

BUF
0
1
X
Z

NOT
0
1
X
Z

Tabelle che rappresentano i valori di uscita delle porte logiche elementari

▼ Esempio - Modellazione a livello di porta logica



Circuito di riferimento da simulare: una porta NAND

```
module my_nand (out1, in1, in2);
    // inizio interfaccia
    output out1;
    input in1, in2; // input inseriti nella stessa definizione,
                    // in quanto sono entrambi da 1 bit
    // fine interfaccia

    // visto che devo collegare la porta AND e la porta NOT, mi serve un wire "w"
    wire w;
    and(w, in1, in2); // l'uscita della porta AND è il wire "w"
    not(out1, w);    // l'uscita della porta not è out1, l'ingresso è w
endmodule
```

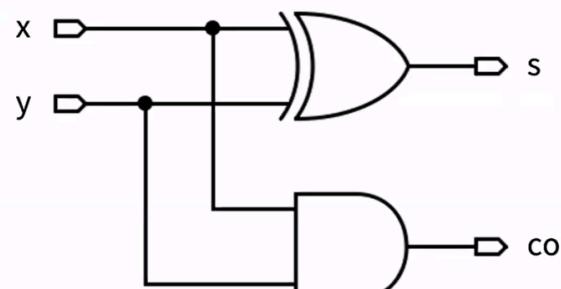
▼ Esempio - Semisommatore con porte logiche

Si realizzi l'Half Adder in [figura](#), tale
che

$s = x+y$ e co sia il Carry Out.

In tal senso, occorre ricordare che
vogliamo fare in modo che s sia:

$$0+0 = 0 \quad | \quad 1+0 = 1 \quad | \quad 0+1 = 1$$



Visualizzazione del modulo Half Adder

Questo si può ottenere facendo lo XOR tra x e y, mentre il Carry Out è ottenibile dall'AND di x e y.

```

module hadd(
    s, co, // uscite
    x, y // ingressi
);
    input x, y;
    output s, co;
    // FINE INTERFACCIA

    xor(s, x, y); // somma
    and(carry, x, y); // riporto
endmodule

```

Ritardi di porte logiche

I ritardi in Verilog sono preceduti da un cancelletto (#). Tipicamente, un ritardo associato a una porta logica è dichiarato come segue:

$$\#(\text{ritardo di propagazione low} \rightarrow \text{high}, \text{ritardo di propagazione h} \rightarrow \text{l}) = \\ \#(t_p^{\text{L} \rightarrow \text{H}}, t_p^{\text{H} \rightarrow \text{L}})$$

Si può essere ancora più specifici, fornendo il range dei ritardi minimi, tipici e massimi. Sui valori di $t_p^{\text{L} \rightarrow \text{H}}$ o di $t_p^{\text{H} \rightarrow \text{L}}$ possiamo definire i ritardi `minimi:tipici:massimi`.

È inoltre indispensabile specificare l'unità di misura di tali ritardi tramite la direttiva:

```

'timescale <unità_di_tempo>/<precisione_temporale>
// solitamente si specifica ad inizio file

```



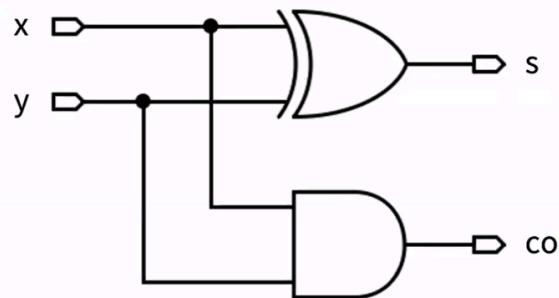
Si noti che tali ritardi sono utili solo nella simulazione comportamentale del circuito. Non hanno, infatti, alcun significato nella sintesi e nella realizzazione effettiva del circuito.

▼ Esempio - Semisommatore con porte logiche, con ritardi

Si realizzi un Half Adder del tipo in [figura](#), tale che $s = x+y$ e co sia il Carry Out.

Si sa che:

- $t_{pxor,min}^{L \rightarrow H} = 2 \text{ ns}$;
- $t_{pxor,max}^{L \rightarrow H} = 4 \text{ ns}$;
- $t_{pxor}^{H \rightarrow L} = 5 \text{ ns}$;
- $t_{pand}^{L \rightarrow H} = t_{pand}^{H \rightarrow L} = 3.6 \text{ ns}$.



```

// dico che l'unita di tempo è 1 ns e che la precisione temporale
// è pari a 100 ps = 0.1 ns
'timescale 1ns/100ps

...
module hadd(
    s, co, // uscite
    x, y // ingressi
);
    input x,y;
    output s, co;
    // FINE INTERFACCIA

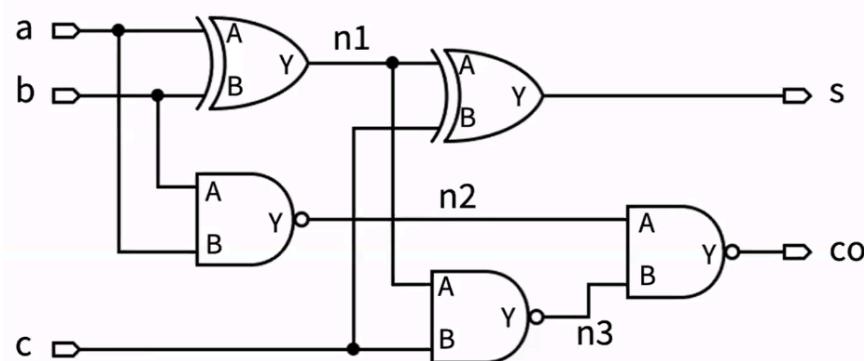
    xor # // definizione del ritardo
        (2:3:4, // definizione del tempo minimo:tipico:medio del tp I→h XOR
         5) // definizione del tempo tipico del tp h→I XOR
        (s, x, y); // somma
    // per intero, andrebbe scritto:
    // xor #(2:3:4, 5) (s,x,y);

    and #(3.567) // definizione del ritardo della porta AND
    // NOTA BENE: visto che la precisione è di 100 ps = 0.1 ns,
    // 3.567 ns viene arrotondato a 3.6 ns
    (co, x, y);
  
```

```
// and #(3.6) (co,x,y);
endmodule
```

▼ Esempio - Utilizzo di sottomoduli

L'obiettivo è realizzare un Full Adder a 3 ingressi tramite porte logiche elementari.



Schema circuitale

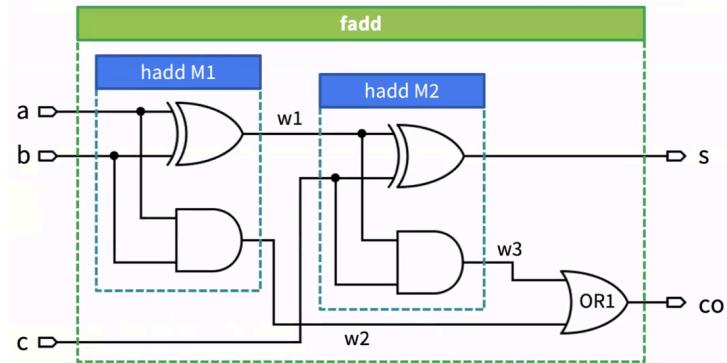
```
module fadd(s, co, a,b,c); // come di consuetudine: prima gli output, poi gli input
  output co, s;
  input a,b,c;
  // FINE INTERFACCIA

  wire n1, n2, n3;
  // FINE NET

  // l'ordine dei comandi sotto è stato volutamente "sparso" per rimarcare che
  // con questa metodologia di programmazione, l'ordine delle istruzioni
  // NON è importante
  nand(n3,n1,c);
  xor(n1,a,b);
  nand(n2,a,b);
  xor(s,n1,c);
  nand(co,n2,n3);
endmodule
```

E' anche vero, però, che possiamo realizzare un Full Adder come interconnessione di Half Adder!

Quindi, avendo prima creato il modulo hadd, possiamo istanziarlo due volte per creare lo schema in figura.



Schema Full Adder definito come interconnessione di Half Adder

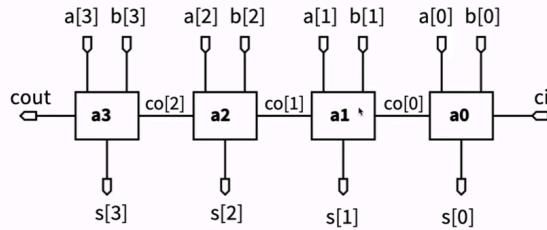
```
module fadd(co,s,a,b,c);
    output co,s;
    input a,b,c;

    wire w1, w2, w3;
    // si istanzia il modulo hadd dandogli un nome!
    // hadd (nome modulo) M1 (nome istanza) (componenti)
    hadd M1 (w1, w2, a,b); // la somma va su w1, il carry su w2
    hadd M2 (s, w3, w1, c); // la somma va su s, il carry su w3
    // N.B. serve avere un modulo hadd in un altro file!
    or OR1 (co, w3, w2);
endmodule
```

▼ Esempio - Sommatore a quattro bit con sottomoduli

- Implementazione gerarchica del modulo add4

- Quattro moduli fadd
- Ogni fadd consiste di
 - Due moduli hadd
 - Una porta logica elementare OR



```

1 module add4 (s, cout, ci, a, b);
2
3   input  [3:0] a, b;
4   input      ci;
5   output [3:0] s;
6   output      cout;
7
8   wire  [2:0] co;
9
10  fadd a0 (co[0], s[0], a[0], b[0], ci);
11  fadd a1 (co[1], s[1], a[1], b[1], co[0]);
12  fadd a2 (co[2], s[2], a[2], b[2], co[1]);
13  fadd a3 (cout, s[3], a[3], b[3], co[2]);
14
15 endmodule
  
```

Definizione del sommatore a quattro bit con sottomoduli



Modellazione RTL

▼ Creatore originale: @Gianbattista Busonera

[Operatore Assign](#)

[Esempi - Operatore assign](#)

[Operatori Verilog utili in assegnazioni continue](#)

[Esempi - Operatori Verilog](#)

[Esempio - XOR con ritardo \(RTL\)](#)

[Esempio - Sommatore completo](#)

[Esempio - Sommatore a 4 bit](#)

[Realizzazione di moduli parametrici](#)

[Esempio - Sommatore a N bit parametrico](#)

[Passaggio di argomenti e parametri](#)

[Ordine delle istruzioni](#)

[Blocco generate con for e if](#)

[Esempio - Sommatore completo su N bit con generate](#)

La modellazione RTL consente di compattare la sintassi e "allontanarci" dalle porte logiche grazie a un meccanismo chiamato "assegnazione continua".

Tale meccanismo prevede di assegnare operazioni complesse di alto livello in una sola riga di codice, per poi lasciare al sintetizzatore logico l'utilizzo delle porte logiche necessarie.

A tale scopo, si possono implementare:

- Sommatori;
- Comparatori;
- Multiplexer;
- Moltiplicatori paralleli.

Operatore Assign

Ogni qualvolta in cui un operando presente nell'assegnazione di tipo `assign` cambia valore, al contempo viene "ricalcolata" la variabile che è stata dichiarata come `assign`.

```
assign #ritardo <nome_rete> = <espressione>;  
// il ritardo (opzionale) prende l'unità di misura dal `timescale a inizio file
```

La destinazione di un'assegnazione dovrebbe sempre essere un wire o una porta di uscita.

▼ Esempi - Operatore assign

```
assign out = a&b|c; // out = a AND b OR c  
// significa che out cambierà "istantaneamente" ogni volta che a, b o c variano  
// tutto questo è più comodo di utilizzare effettivamente porte AND e OR
```

```
assign eq = (a+b == c) // ci si chiede se la somma tra a e b sia uguale a c  
// se a+b == c, eq = 1; diversamente 0.
```

```
wire #10 inv = ~in; // la variabile inv cambierà dopo 10 timescale  
// da quando "in" varia  
// inv = NOT(in) con ritardo.  
// !!! questa sintassi è equivalente a:  
//     wire inv; assign #10 inv = ~in  
// posso comunque definire e assegnare un wire!
```

```
wire [7:0] c = a+b; // anche questa è un'assegnazione (assign) continua (implicita)
```



Occhio a evitare loop del tipo `assign a = b+a;`, poiché tale tipo di assegnazione non è sintetizzabile.

Operatori Verilog utili in assegnazioni continue

Operatori aritmetici		Operatori bit a bit		Operatori di spostamento	
a + b	Somma	~a	NOT bit a bit	a << n	Shift logico a sinistra
a - b	Differenza	a & b	AND bit a bit	a >> n	Shift logico a destra
-a	Cambio segno	a b	OR bit a bit	a <<< n	Shift aritmetico a sinistra
a * b	Moltiplicazione	a ^ b	XOR bit a bit	a >>> n	Shift aritmetico a destra
a / b	Divisione	a ~^ b	XNOR bit a bit	{a, b}	Concatenazione
a % b	Resto	a ^~ b	XNOR bit a bit		
Operatori relazionali		Operatori di riduzione		Operatori logici	
a == b	Uguale	&a	AND tutti i bit	!a	Negazione logica
a != b	Diverso	a	OR tutti i bit	a && b	AND logico
a < b	Minore	^a	XOR tutti i bit	a b	OR logico
a > b	Maggiore	~&a	NAND tutti i bit	sel?a:b	Condizionale ternario
a <= b	Minore o uguale	~ a	NOR tutti i bit		
a >= b	Maggiore o uguale	~^a	XNOR tutti i bit		

Operatori Verilog supportati nelle assegnazioni continue



La tilde (~) è ottenuta su Windows con ALT+126 .

▼ Esempi - Operatori Verilog

```
assign c = a+b; // c sarà (in automatico???) di N+1 bit se a e b sono su N bit
```

```
assign c = a>b; // c starà su 1 bit; mi restituisce un valore logico
```

```
assign c = a||b; // a e b sono entrambi su 1 bit, così come c e si fa l'OR logico  
// se a o b fossero stati su N bit, sarebbero stati gestiti come 0 solo se  
// fossero 0, altrimenti a e b sarebbero stati ridotti a 1 logico
```

```
assign c = a&b; // a,b,c su N bit, in quanto si fa l'AND bit a bit
```

```
assign c = &a; // AND di tutti i bit di a.
```

▼ Esempio - XOR con ritardo (RTL)

Si realizzzi la porta logica XOR, come in [figura](#).



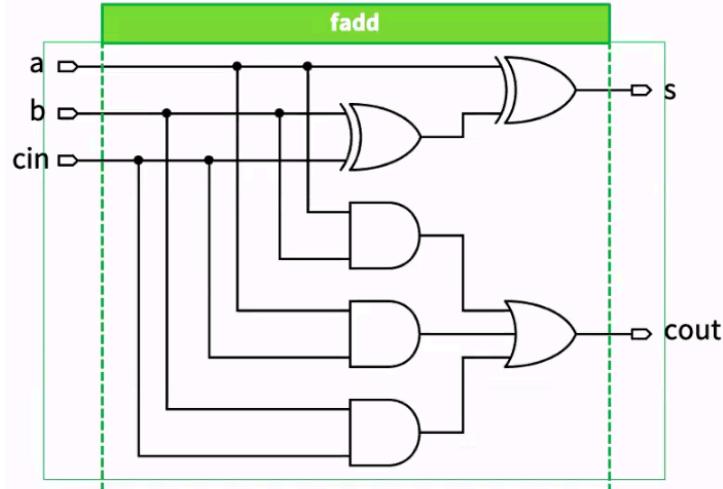
Porta XOR

```
module my_xor(c, a, b); // al solito, prima uscita, poi ingressi
    output c;
    input a,b;

    assign #2 c = a ^ b; // a XOR (bit a bit) b con ritardo di 2 timescale
endmodule
// SI RICORDI CHE TALE RITARDO SI USA SOLO IN SIMULAZIONE (NON IN SINTE
```

▼ Esempio - Sommatore completo

Si implementi un Full Adder generico, come in [figura](#).



Schema di un Full Adder

```
module fadd (cout, s, a, b, cin)
    output cout, s;
    input a,b,cin;
```

```

assign s = (b ^ cin) ^ a;

assign cout = (a&b) | (a&cin) | (b&cin);
endmodule

```

▼ Esempio - Sommatore a 4 bit

Si implementi un sommatore tale che gli operandi siano su 4 bit.

```

module fh4(s, cout, a,b,cin);
    output [3:0] s; // somma su 4 bit
    output cout; // carry out su un bit
    input [3:0] a,b; // operandi di ingresso su 4 bit
    input cin; // carry in di ingresso su un bit

    // Si potrebbe fare, come prima utilizzo di XOR e (AND + OR) oppure,
    // come in questo caso, sfruttare l'operatore di concatenazione
    assign {cout,s} = a+b+cin; // a+b+cin genera un output a 5 bit di cui l'MSB
                                // sarà il carryout

```

Realizzazione di moduli parametrici

Il modulo “generico”, definito come parametrico, ha come obiettivo quello di farci fornire, dall’utente, un parametro che possa servirci per scopi quali:

- la dichiarazione di un vettore con le corrette dimensioni;
- l’istanziazione di un modulo che prevede “N” (con N passato come parametro) iterazioni.

La sintassi per l’utilizzo di un modulo parametrico è la seguente:

```
<nome_modulo> #(elenco_parametri) <nome_istanza> (elenco_porte);
```

La sintassi per la dichiarazione di un parametro è, invece, la seguente:

```

parameter <nome_parametro>; // se non si vuole dare un valore di default al parametro
parameter <nome_parametro> = <valore_default>; // se si vuole dare un valore di default

```

▼ Esempio - Sommatore a N bit parametrico

Vogliamo realizzare un sommatore parametrico tale che il numero N di bit degli operandi sia passato dall'utente.

```
module #(parameter N = 2) adder (cout, s, a, b, cin);
    // specifichiamo un parametro "N", di default uguale a 2
    input [N-1:0] a,b; // specifico che gli operandi sono su N bit
    input cin;
    output [N-1:0] s; // specifico che la somma (senza cout) sta su N bit
    output cout;

    assign {cout, s} = a+b+cin;
endmodule
```

Vediamo ora un altro modo (meno leggibile) di implementare questo codice.

```
module adder (cout, s, a, b, cin);
    parameter N = 2; // specifichiamo un parametro "N", di default uguale a 2
    input [N-1:0] a,b; // specifico che gli operandi sono su N bit
    input cin;
    output [N-1:0] s; // specifico che la somma (senza cout) sta su N bit
    output cout;

    assign {cout, s} = a+b+cin;
endmodule
```

Vediamo ora un po' di utilizzi del modulo parametrico `adder`.

```
// in un altro file!
adder #(8) adder_8 (co, s, a,b,ci)      // sommatore a 8 bit
adder adder_2 (co, s, a, b, ci)        // sommatore a 2 bit (uso il default)
adder #(12) adder_12 (co, s, a, b, ci)  // sommatore a 12 bit
adder #(.N(8)) adder_8 (co, s, a, b, ci) // specifico che mi riferisco a N
```



Come faccio a specificare il ritardo di un modulo da me creato?

Nel caso di un sommatore a 8 bit con 4 ns di ritardo NON si può fare una chiamata di questo tipo:

```
'timescale 1ns/100ps  
...  
adder #(8) #4 adder_8 (co, s, a,b,cin) // sommatore a 8 bit
```

I ritardi in Verilog non possono essere specificati su moduli dichiarati da noi in maniera diretta (come facevamo con le porte elementari), possiamo rendere il ritardo parametrico, come segue:

```
module adder (cout, s, a, b, cin);  
    parameter N = 2; // specifichiamo un parametro "N", di default uguale a 2  
    parameter delay = 0;  
    input [N-1:0] a,b; // specifico che gli operandi sono su N bit  
    input cin;  
    output [N-1:0] s; // specifico che la somma (senza cout) sta su N bit  
    output cout;  
  
    assign #(delay) {cout, s} = a+b+cin;  
    // si noti che, nel caso di ritardo costante, avremmo potuto inserire  
    // assign #4 {cout, s} ....; Poichè stiamo dando un parametro, però  
    // occorre metterlo tra parentesi per aiutare il parser.  
endmodule
```

Successivamente, si può effettuare una chiamata come segue:

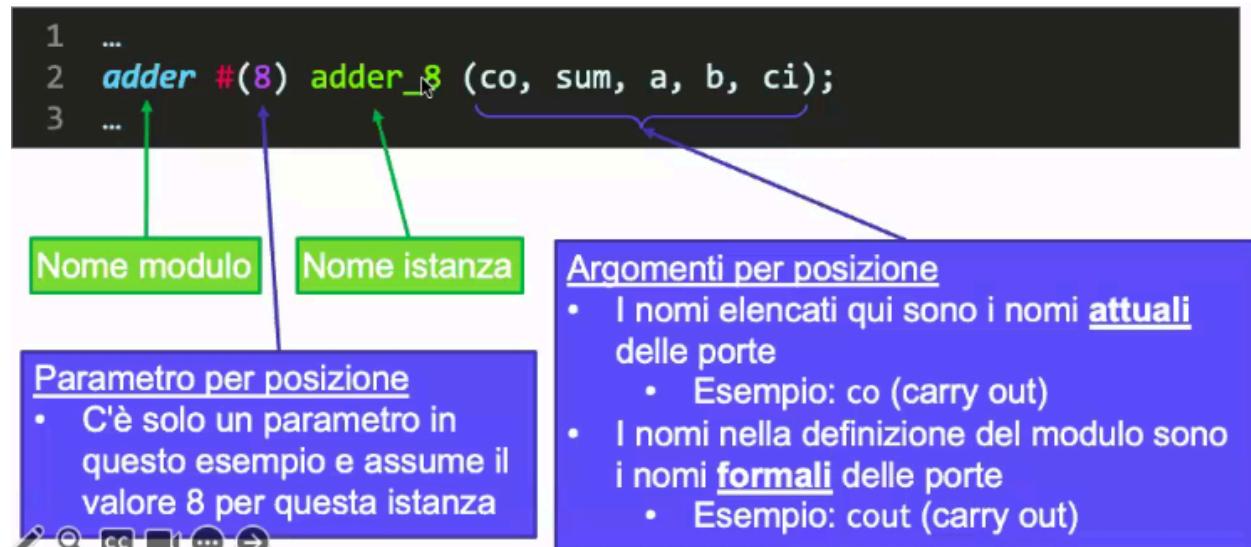
```
adder #(8, 4) adder_8_4 (co, s, a, b, cin); // che funziona correttamente  
adder #(.N(8), .delay(4)) adder_8_4 (co, s, a, b, cin); // più leggibile  
adder #(.delay(4), .N(8)) adder_8_4 (co, s, a, b, cin); // più leggibile
```

Si presti attenzione che tali informazioni non vengono dal docente, ma sono ricavate dalla rete e, pertanto, l'assegnazione di un ritardo ad assign #(delay) potrebbe non funzionare correttamente con tutti i compilatori!

Passaggio di argomenti e parametri

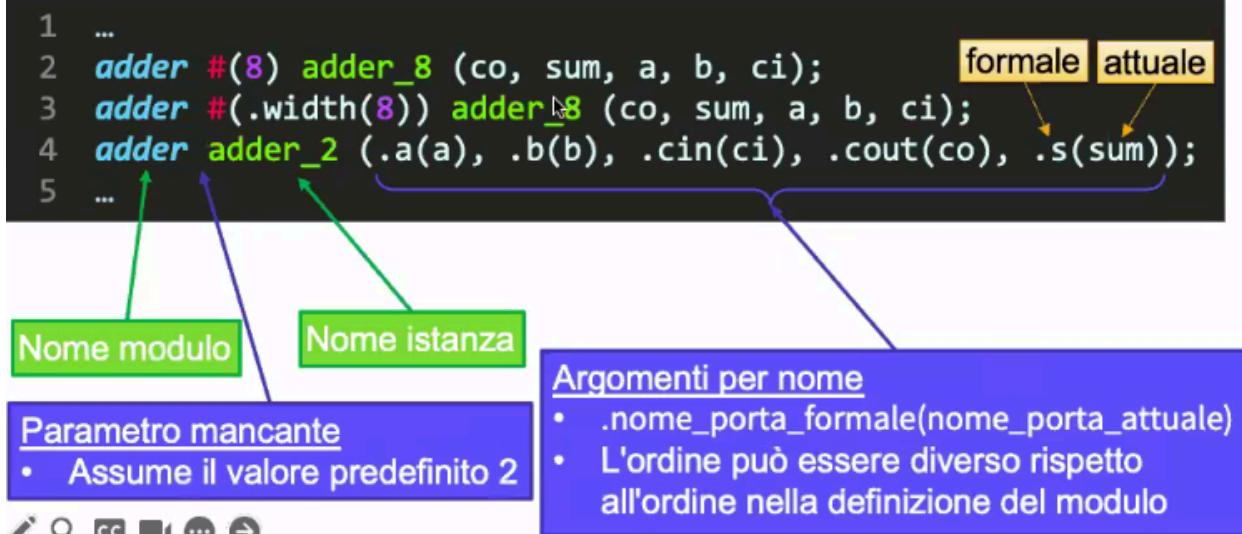
Quando chiamiamo un modulo possiamo scegliere di passare i parametri "attuali" al modulo, il quale contiene i parametri "formali" per posizione (come in C) o per nome (in maniera tale da rendere l'ordine ininfluente e rendere il codice più leggibile).

Data la seguente funzione, un esempio di passaggio di parametri per posizione è il seguente:



Modulo parametrico implementato in precedenza, in cui tutti gli argomenti e i parametri sono passati per posizione

Un esempio di passaggio di parametri per nome è il seguente:



Modulo parametrico implementato in precedenza, dove si fa utilizzo in riga 3 di argomenti e parametri per nome.

Si noti che width, da noi, è stato chiamato N

Si possono, inoltre, lasciare delle porte non collegate come segue:

```

adder adder_2 (.s(sum), .a(a), .b(b), .cin(ci));
adder #(2) adder_2 (, sum, a, b, ci);

```

Se un'uscita non ci serve, possiamo decidere di non collegarla a niente. Ciò non si può fare con gli ingressi!

Ordine delle istruzioni

L'ordine delle istruzioni, come detto nell'introduzione, non è importante in RTL, poiché:

- tutte le istruzioni sono eseguite in parallelo;
 - le espressioni a destra dell'uguale vengono prima calcolate tutte insieme;
 - vengono assegnate tutte insieme ai wire a sinistra dell'uguale;
- sono sempre equivalenti a uno schema a blocchi.

```

1 ...      assign c = a & b;
2      assign d = a | c;
3      assign s = c ^ d;
4 ...
5 ...

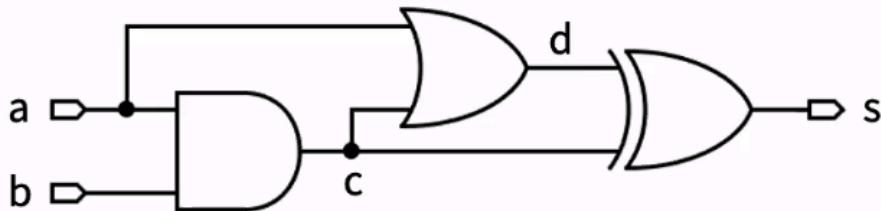
```



```

1 ...      assign s = c ^ d;
2      assign c = a & b;
3      assign d = a | c;
4 ...
5 ...

```



Equivalenza tra codici anche con ordine delle istruzioni differente!

Blocco generate con for e if

Come detto in [precedenza](#), si può parametrizzare un modulo in maniera tale che si iteri N volte.

In questo caso, il blocco `generate` ci consente di effettuare cicli e blocchi condizionali solo per strutture “ripetitive” (si tratta di una esecuzione durante la fase di compilazione, non in runtime), per le quali dovremmo usare tante righe di codice per istanziare uno stesso comando, come nel seguente esempio:

```

parameter N; // generica variabile parametrica che descrive il numero delle iterazioni
genvar i; // variabile utile nel ciclo for

generate
  for (i = 0; i < N; i++) begin
    if(i == 0)
      istruzione1;
    else if (i == N-1)
      istruzione2;
    else begin;
      istruzione3;
      istruzione4;
    end
  end
endgenerate

```

```
// Si noti come:  
// - con una istruzione si comporta come in C, ovvero non  
// sono necessarie le graffe;  
// - con più istruzioni sotto un blocco condizionale o un for  
// sono necessarie le "graffe", definite da begin ed end.
```

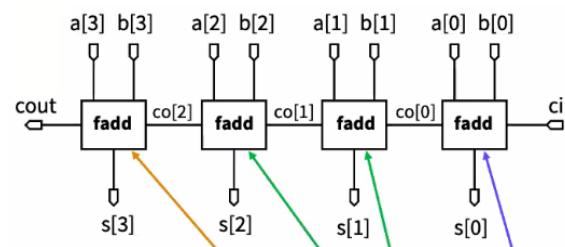


Nei blocchi `generate` non è possibile dare nomi ai moduli istanziati in maniera "semplice", ma avrebbe comunque poco senso farlo ([vedi esempio](#)).

Si noti che non è un loop iterativo che assicura di realizzare istruzioni in sequenza, ma è come se servisse solo a replicare il codice.

▼ Esempio - Sommatore completo su N bit con generate

Si parta dal sommatore in figura a 4 bit, per poi estenderlo a N bit.



In blu la prima istanza, in verde le intermedie,
in arancione l'ultima istanza

```
module adder (s, cout, ci, a, b);  
  parameter N = 4; // di default realizziamo un sommatore a 4 bit, come in figura  
  input [N-1:0] a,b;  
  input ci;  
  output [N-1:0] s;  
  output cout;  
  
  wire [N-2:0] co; //si utilizza un vettore che contenga i vari wire  
  // sono in totale N-1 bit in quanto l'N-esimo è cout  
  genvar i; // variabile intera utile per l'iterazione
```

```

generate
    for (i = 0; i < N; i++) begin
        if(i == 0)
            fadd(co[i], s[i], a[i], b[i], ci); // ISTANZA BLU
        else if (i == N-1)
            fadd(co[i], s[i], a[i], b[i], co[i-1]); // ISTANZA ARANCIONE
        else
            fadd(cout, s[i], a[i], b[i], co[i-1]); // ISTANZA VERDE
    end
endgenerate
endmodule

// Si noti che nelle chiamate al modulo "fadd" non si
// specifica il nome dell'istanza.
//
// Nel caso del generate, ciò non si può fare perché
// si andrebbe, negli intermedi, a istanziare moduli
// con lo stesso nome.

```



Modellazione comportamentale

▼ Creatore originale: @Gianbattista Busonera

[Always con circuiti combinatori](#)

[Esempio - Porta AND](#)

[Always con circuiti sequenziali](#)

[Esempio - Porta AND + FF](#)

[Assegnazioni bloccanti](#)

[Assegnazioni non bloccanti](#)

[Esempio - Realizzazione di un circuito sequenziale Mealy \(FSM\)](#)

[Esempio - Realizzazione di un circuito sequenziale Moore \(FSM\)](#)

La modellazione comportamentale è ottenuta tramite blocchi `always`, i quali definiscono una sorta di processi, ovvero funzioni le cui istruzioni sono eseguite in sequenza, ma nello stesso istante.

Per fare un paragone, un blocco `always` è una specie di chiamata ricorsiva che viene effettuata ogni qual volta i parametri "sensibili" (che devono essere `reg` o `wire`) del blocco `always` vengono modificati al di fuori del blocco `always` stesso.

Con tale paradigma si possono modellare sia circuiti combinatori, sia sequenziali.

Ecco la sintassi di un blocco `always`:

```
always @(<elenco_sensibilità>) begin
    ...
end

// Quando uno o più segnali presenti nell'elenco di sensibilità
// cambia valore, il blocco always viene eseguito.
```



Tutte le uscite di un blocco `@always` devono essere di tipo `reg`:

- le assegnazioni usano solo `=` o `<=`;
- mantengono memoria dell'ultima assegnazione, al contrario dei `wire` (che, in uscita, sono frutto di assegnazioni "continue").

Il tipo `reg` assomiglia a un registro, ma non ne rappresenta necessariamente uno.

Dentro i blocchi `always` possono essere utilizzati i costrutti `if` e `case`, grazie al fatto che le operazioni vengono eseguite **in ordine**.

Always con circuiti combinatori

La caratteristica dei circuiti combinatori è tale per cui l'uscita dipende, in ogni istante, dal valore degli ingressi e, pertanto, il blocco `always` sarà del tipo:

```
always @(ingresso1, ingresso2, ..., ingressoN) begin
```

```
    ...
```

```
end
```

```
// L'elenco di sensibilità è dato dagli ingressi presenti nel circuito combinatorio.
```

Per essere sicuri di valutare tutti gli ingressi in una logica combinatoria si può anche utilizzare tale sintassi:

```
always @(*) begin // Viene eseguito sulla variazione di:
```

```
    out1 = a+b+c+d; // sulla variazione di uno tra a,b,c,d
```

```
    out2 = f*e+3; // sulla variazione di uno tra f ed e.
```

```
end
```

```
// Il blocco always viene eseguito ogni volta che un
```

```
// secondo membro di un'assegnazione cambia valore.
```

▼ Esempio - Porta AND

Si realizzi la porta AND in figura, tramite il blocco

```
always .
```



Porta AND

```
reg y; // uscita sempre di tipo reg
wire a,b; // ingressi possono essere wire o reg.
always @(a or b) begin
    y = a&b;
end
```

Tale codice è identico a scrivere il seguente codice:

```

reg y; // uscita sempre di tipo reg, mantiene memoria dell'ultima assegnazione
wire a,b; // ingressi possono essere wire o reg.
always @(*)
    y = a&b; // con una sola riga di codice posso evitare di usare begin e end

```

Esso è ancora uguale a:

```

wire y, a,b; // il wire presuppone un'assegnazione continua che esiste sempre!
assign y = a & b; // qui possiamo usare un wire come uscita!

// Reg si potrebbe usare come secondo membro di un'operazione
// di assign ma non come primo membro

```

Always con circuiti sequenziali

Nel caso di circuiti sequenziali è necessario far sì che il blocco `always` sia attivato solo durante il fronte di salita/discesa del clock tramite le parole chiave:

- `posedge`: fronte di salita;
- `negedge`: fronte di discesa.

```

always @(posedge clock) // esempio 1
always @(negedge clock or reset) // esempio 2, FF attivo su fronte di discesa
// con reset asincrono

// Il clock è un input, non una parola chiave (wire o input che sia)

```

▼ Esempio - Porta AND + FF

Quando arriva il fronte di salita del clock, si vuole che un flip flop memorizzi l'AND tra due ingressi a e b.

```

reg a,b,ff;
always @(posedge clock) // ogni fronte di salita del clock voglio che il ff memorizzi
    ff = a & b;

```

Assegnazioni bloccanti



È importante ricordare che, all'interno del corso, utilizzeremo solo l'assegnazione **bloccante**.

Le assegnazioni bloccanti hanno un effetto immediato (come in C):

```
always @(a or b) begin // 1
    a = b;           // 2
    b = a;           // 3
end                 // 4
```

// Qualora a o b (o entrambi) abbiano una variazione di valore,
// il blocco always NON verrà eseguito.

1. supponiamo che, durante la prima esecuzione del blocco `always`, si abbia `a = 0` e `b = 1`, come in tabella;

	1° riga di codice	2° riga di codice	3° riga di codice	4° riga di codice
a	0	1	1	1
b	1	1	1	1

2. alla seconda riga di codice otteniamo `a = b` e, quindi, `a` passa da 0 a 1, mentre `b` mantiene il suo valore precedente (pari a 1);
3. alla terza riga di codice, `a` mantiene il suo valore precedente (1) e si esegue `b = a`, ma `a = 1`, quindi `b = a = 1`;
4. si raggiunge la fine del blocco `always`: `a` e `b` mantengono i loro valori precedenti (1 e 1).



Ci si potrebbe chiedere come mai non si rientri nel blocco `always`, dato che `a` ha cambiato valore.

Questo non avviene perché il blocco `always` viene rieseguito se e solo se `a` o `b`, o addirittura entrambi subiscono variazioni, ma esse devono avvenire **al di fuori** del blocco `always` stesso.

Visto che le variazioni ai parametri di sensibilità (`a` e `b`) sono solo interne al blocco `always`, quest'ultimo **non viene nuovamente eseguito!**

Assegnazioni non bloccanti

Le assegnazioni non bloccanti avvengono solo al termine dell'esecuzione del blocco `always` e, in tal senso, comporterebbero una nuova esecuzione del blocco `always` nel caso di variazione.

```

always @(a or b) begin // 1
    a <= b;           // 2 assegna a = b alla fine del blocco always
    b <= a;           // 3 assegna b = a alla fine del blocco always
end                  // 4

```

// qualora a o b (o entrambi) abbiano una variazione di
// valore, il blocco always verrà eseguito.

- supponiamo che, durante la prima esecuzione del blocco `always`, si abbia `a = 0` e `b = 1`, come in tabella;

1° esecuzione	1° riga di codice	2° riga di codice	3° riga di codice	4° riga di codice	Valore futuro
a	0	0	0	1	1
b	1	1	1	0	0

- viene eseguita l'istruzione `a <= b` e, quindi, il **valore futuro** di `a` sarà pari al valore corrente di `b` (1) solo alla fine del blocco `always`;
- viene eseguita l'istruzione `b <= a` e, quindi, il valore futuro di `b` sarà pari al valore corrente di `a` (0) solo alla fine del blocco `always`;
- raggiunta la fine del blocco `always` vengono assegnati i valori futuri di `a` e `b`.



Poiché `a` e `b` sono variate al di fuori del blocco `always` e sono nella lista di sensibilità, il blocco si riattiva e viene nuovamente eseguito.

2° esecuzione	1° riga di codice	2° riga di codice	3° riga di codice	4° riga di codice	Valore futuro
a	1	1	1	0	0
b	0	0	0	1	1

Poiché, nuovamente, `a` e `b` sono variate e si trovano nella lista di sensibilità, il blocco `always` viene nuovamente eseguito (indefinitivamente, come in [figura](#)).

1 ^a esecuzione	2 ^a esecuzione	3 ^a esecuzione	4 ^a esecuzione	5 ^a esecuzione	6 ^a esecuzione	7 ^a esecuzione	8 ^a esecuzione	9 ^a esecuzione	Futuro
1 2 3 4	1 2 3 4	1 2 3 4	1 2 3 4	1 2 3 4	1 2 3 4	1 2 3 4	1 2 3 4	1 2 3 4	
a 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1									
b 1 1 1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1									

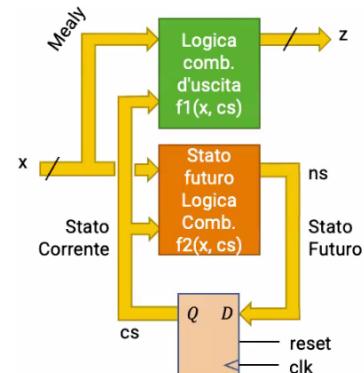
Tipicamente questo comportamento non è desiderato, anche se somiglia a un clock

▼ Esempio - Realizzazione di un circuito sequenziale Mealy (FSM)

Come noto, una macchina di Mealy è tale per cui il valore dell'uscita dipende sia dal valore corrente degli ingressi, sia dallo stato corrente.

Si vuole, quindi, realizzare una macchina di Mealy come in [figura](#).

Si noti che il FF-D è attivo sul fronte di salita.



Macchina di Mealy

```
module MealyFSM (z, clk, reset, x);
    output reg z;
    input clk, reset, x;
    // si poteva anche fare output z; reg z;
    reg ns; // next state (sono come wire ma necessariamente devono essere dichiarati reg)
    reg cs; // current state (come wire ma serve dichiararlo come reg poichè output).

    /// LE REG SONO NECESSARIE PER USARLE COME "uscita" dei blocchi always

    // GESTIAMO LA VARIAZIONE DEGLI STATI (dipendente da clock e reset)
    always @(posedge clock) // attivo sul fronte di salita del clock
        or
        posedge reset) // reset asincrono: se reset passa da 0→1 allora si entra
    begin: stateReg // etichetta utile per documentazione, nome del blocco always
        if (reset)
            cs = 0;
        else
            cs = ns; // current state = next state... aggiorno lo stato corrente
    end
    always @(x or cs) begin: outputFunction
        // la lista di sensibilità è data dall'ingresso e dallo stato corrente.
        z = f1(x,cs); // l'uscita dipende da ingresso e stato corrente.
    end

```

```

always @(x or cs) begin: nextStateTransition
    ns = f2(x, cs); // il next state dipende da ingresso e stato corrente
endmodule

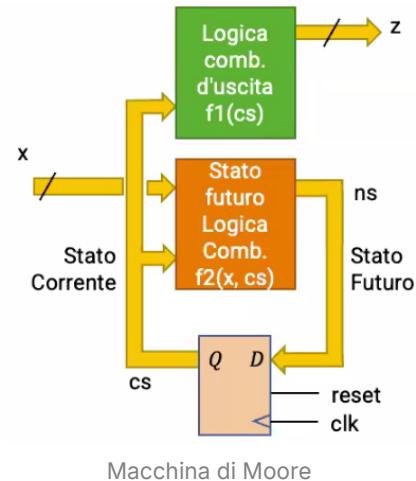
```

▼ Esempio - Realizzazione di un circuito sequenziale Moore (FSM)

Come noto, una macchina di Moore è tale per cui il valore dell'uscita dipende solo dallo stato corrente.

Si vuole, quindi, realizzare una macchina di Moore come in figura.

Si noti che il FF-D è attivo sul fronte di salita e che il reset è sincrono (cioè deve arrivare insieme al clock).



```

module MealyFSM (z, clk, reset, x);
    output z;
    input clk, reset, x;
    reg ns; // next state (sono come wire ma necessariamente devono essere dichiarati reg)
    reg cs; // current state (come wire ma serve dichiararlo come reg poichè output).
    reg z;
    /// LE REG SONO NECESSARIE PER USARLE COME "uscita" dei blocchi always

    // GESTIAMO LA VARIAZIONE DEGLI STATI (dipendente da clock)
    always @(posedge clock) // vogliamo un reset sincrono!
    begin: stateReg
        if (reset) // Il reset deve arrivare durante la transizione del clock!
            cs = 0;
        else begin
            cs = ns;
        end
    end
    always @(cs) begin: outputFunction
        // la lista di sensibilità è data dal solo stato corrente.
        z = f1(cs); // l'uscita dipende da ingresso e stato corrente.
    end

```

```
always @(x or cs) begin: nextStateTransition
    ns = f2(x, cs); // il next state dipende da ingresso e stato corrente
endmodule
```



Testbench in Verilog

▼ Creatore originale: @Gianbattista Busonera

Come scrivere un testbench

Primo passo - Istanziare modulo testbench e DUT

Secondo passo - Inizializzazione del sistema in esame

Modellazione del tempo

Blocco initial

Ciclo forever e generazione di un segnale di clock

Terzo passo - Definire il comportamento dell'unità di benchmark

Valutazione del comportamento

Esempio - Test di un modulo già realizzato

Istanziazione dei componenti

Inizializzazione del sistema in esame

Comportamento del sistema

Esempio - Test di un up-counter a 4 bit

Una volta terminata l'implementazione di un modulo, può essere utile realizzare un ulteriore modulo, denominato "testbench" (banco di prova). Questo modulo non realizza un'ulteriore circuito, ma serve esclusivamente per verificare la corretta funzionalità di un modulo creato in precedenza.

Il componente che viene viene sottoposto al test può essere chiamato:

- MUT, ovvero module under test;
- DUT, ovvero design under test.

Come scrivere un testbench

Primo passo - Istanziare modulo testbench e DUT

Il primo passo della scrittura di un testbench consiste nella creazione di un modulo, il quale solitamente non ha né input, né output.

```
module <module_name>_tb ();
    // istanziazione DUT
    // testbench body
endmodule
```

Come convenzione, il nome del modulo di testbench è definito con il nome del modulo da testare, seguito dalla dicitura `_tb`, salvando i file nello stesso modo.

Adesso possiamo istanziare il modulo da testare, che utilizzerà la seguente sintassi:

```
<module_name> #(.<param_name1>(<param_value1>), .<param_name2>(<param_value2>))
    <instance_name> (.<parametro_formale1>(<parametro_attuale1>), ...);

// in maniera più compatta
<module_name> #(elenco_parametri) <instance_name> (<elenco_porte>);
```

Il tutto risulterà, quindi, come segue:

```

module <module_name>_tb ();
    // istanziazione parametri e porte
    // INSTANZIAZIONE DUT
    <module_name> #(<elenco_parametri>) <nome_istanza> (<elenco_porte>);
    // testbench body
endmodule

```

Secondo passo - Inizializzazione del sistema in esame

L'inizializzazione del sistema in esame riguarda anche, ma non solo, la gestione del segnale di clock nel caso in cui ci troviamo a testare un circuito sequenziale.

Modellazione del tempo

Per modellare il tempo:

- si può fare in modo di aspettare un certo quantitativo di tempo tra un'istruzione e l'altra;

```

<istruzione 1>;
#5 // l'assenza del ";" è voluta!
<istruzione 2>;

```

In questo caso, viene eseguita l'istruzione 1, si attendono 5 unità di tempo definite dalla variabile `#timescale` e, successivamente, viene eseguita l'istruzione 2.

- si può fare in modo di assegnare un risultato ad una variabile con un certo ritardo, come già noto.

```

... // definizione variabili
#10 a = 1'b1; // Assegna ad a il valore 1 dopo 10 unità di tempo

```

Blocco initial

Soltanente, l'inizializzazione del sistema in esame avviene attraverso i cosiddetti blocchi `initial`, tali per cui il codice al loro interno viene eseguito una sola volta ad inizio simulazione. Come per il blocco `always`, si tratta di un blocco procedurale, ma, a differenza di quest'ultimo, non è un blocco sintetizzabile.

```

initial begin
    ... // istruzioni varie
end

```

Un esempio di utilizzo può essere il seguente:

```

initial begin
    gate_in = 2b'00; // assegna 00 a gate in
    #10          // attende 10 unità di tempo
    gate_in = 2b'01; // assegna 01 a gate in
    #10          // attende 10 unità di tempo
    gate_in = 2b'10; // assegna 10 a gate in
    #10          // attende 10 unità di tempo
    gate_in = 2b'11; // assegna 11 a gate in
end // si noti che gate_in mantiene il valore 11 fino alla fine della simulazione
// Tale blocco initial termina dopo 30 unità di tempo in totale.

```

Ciclo forever e generazione di un segnale di clock

La parola chiave `forever` può modellare un ciclo infinito, tale per cui le istruzioni al suo interno sono eseguite per una durata indefinita durante la simulazione.

```
forever begin
    ... // istruzioni varie
end
```

Si può utilizzare per generare un segnale di clock periodico.

```
initial begin
    clk = 1b'0; // inizializzo il clock a 0... perchè posso
    forever begin
        #1 clk = ~clk; // con un ritardo di una unità di tempo il clock nega il suo valore
    end
end
```

Nel frammento di codice appena visto, si può notare come si istanzi un clock inizializzato a 0 che, per ogni unità di tempo (la quale può essere scelta in base alle necessità), commuta il proprio valore all'infinito, secondo la sequenza 0→1→0→1→0→...→0→1→...

Unità di tempo	0	1	2	3	4	5	6
clk	0	1	0	1	0	1	0

Terzo passo - Definire il comportamento dell'unità di benchmark

In questa sezione, si definiscono i blocchi `always` necessari a descrivere il comportamento dell'unità di benchmark, il quale è, generalmente, un modulo che genera dei test e, opzionalmente, verifica le risposte del DUT.

Si deve notare come una parte dell'inizializzazione o della generazione degli stimoli possa apparire in blocchi di questo tipo, come i segnali di clock.

Valutazione del comportamento

Per valutare il comportamento di un circuito si possono utilizzare le cosiddette "system tasks" (o "system functions"), il cui nome è generalmente preceduto dal simbolo "\$", e risultano simili a delle syscall.

Tra le "funzioni" più utilizzate ricordiamo:

- `$display("...")`, che è simile a una `printf` in C, permettendo di stampare a schermo una volta;

Si riporta un esempio di utilizzo di tale funzione, accostato dai modificatori utilizzabili:

```
$display("x (binario) = %b");
$display("x (decimale) = %d");
$display("x (esadecimale) = %h");
```

Modificatore	Descrizione
%h, %H	Rappresentazione in esadecimale
%d, %D	Rappresentazione in decimale
%b, %B	Rappresentazione in binario
%o, %O	Rappresentazione in ottale
%m, %M	Visualizza nome gerarchico del modulo corrente
%s, %S	Rappresentazione come stringa di testo
%t, %T	Rappresentazione come tempo
%f, %F	Rappresentazione in virgola mobile
%e, %E	Rappresentazione in formato esponenziale

Tabella dei modificatori utilizzabili

- `$monitor("...")` permette di stampare a schermo ogni volta che un parametro tra i suoi argomenti cambia valore;

Si riporta un esempio di utilizzo di tale funzione:

```
$monitor("ingresso_1 = %b, ingresso_2 = %d, ingresso_3 = %h", bin, dec, hex);
// STAMPERA' OGNI QUAL VOLTA UNO TRA I 3 INGRESSI CAMBIA VALORE!
```

- `$time()` permette di stampare il tempo raggiunto dall'inizio di una simulazione in un certo formato;

Il system task `$time` ritorna il tempo come vettore su 64 bit, anche se esistono alcune varianti:

- `$stime` ritorna il tempo come intero su 32 bit;
- `$realtime` ritorna il tempo come numero reale.

Si può modificare il formato della rappresentazione attraverso la seguente sintassi:

```
$timeformat(esponente, precision, "suffix", min_width);
// unità di misura del tempo, numero di cifre dopo la virgola, suffisso,
```

Un esempio di applicazione è il seguente:

```
// Multipli di 1ns, 2 posizioni decimali e almeno 10 cifre complete del dato
$timeformat(-9, 2, "ns", 10); // 10E-9s, 2 decimali, suffisso "ns", campo da 10
$display("ingresso = %b al tempo time = %t", binario, $time);
```



A cosa serve `min_width`?

Si tratta della **larghezza minima**, in termini di caratteri, del campo di stampa:

- se il tempo stampato occupa meno di `min_width` caratteri, verranno aggiunti **spazi a sinistra** (padding) per raggiungere quella larghezza minima.
- se il tempo stampato occupa più spazio, viene stampato comunque interamente e non viene quindi troncato.

Vediamo un esempio pratico:

```
$timeformat(-9, 2, " ns", 12);
$display("Time: %t", $realtime);
```

Se il tempo corrente è `123.45 ns`, l'output sarà:

```
Time: 123.45 ns
```

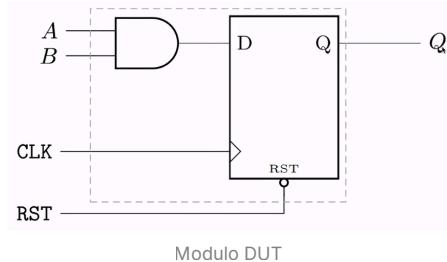
(larghezza totale del campo = 12 caratteri, spazi inclusi)

- `$finish` conclude la simulazione, altrimenti la simulazione andrebbe avanti per sempre.

Esempio - Test di un modulo già realizzato

Si supponga di voler testare il circuito (DUT) in figura, già realizzato, in un file chiamato "example.v" in cui vi sia un modulo chiamato `example`.

Il nostro intento è quello di creare un ulteriore file "example_tb.v" e un modulo `example_tb` per eseguire dei test su di esso.



▼ Istanziazione dei componenti

Creiamo il modulo `example_tb` e istanziamo il modulo `example`, di cui si riporta una possibile realizzazione:

```

`timescale 1ns / 1ps
module example(output reg Q, input clk, rst, a, b);
    // simuliamo che l'uscita Q possa cambiare o sul fronte di salita del clock
    // o quando il reset è attivo basso
    always @ (posedge clk or !rst) begin
        if (!rst)
            Q = 0;
        else
            Q = a&b;
    end
endmodule
    
```

Procediamo con la creazione del testbench:

```

`timescale 1ns / 1ps
module example_tb();
    reg clock, reset;
    wire a, b;
    wire Q; // il testbench di solito non ha ingressi e uscite!
    // INSTANZIAZIONE module "example"
    example dut(
        .Q(Q),
        .clk(clock),
        .rst(reset),
        .a(a),
        .b(b)
    );
    // CONTINUAZIONE
    ...
endmodule
    
```

▼ Inizializzazione del sistema in esame

Si generano i segnali di clock e reset, i quali saranno utili nella definizione del comportamento del sistema.

```

`timescale 1ns / 1ps
module example_tb();
    reg clock;
    wire reset, a, b;
    
```

```

wire Q; // il testbench di solito non ha ingressi e uscite!
// INSTANZIAZIONE module "example"
example dut(
    .Q(Q),
    .clk(clock),
    .rst(reset),
    .a(a),
    .b(b)
);

/// NUOVO CODICE QUI
// istanziazione clock tramite ciclo infinito
initial begin
    clock = 1'b0;
    forever begin
        #1 clock = ~clock;
    end
end

// facciamo variare il reset...
initial begin
    reset = 1'b0; // inizializziamo il sistema in esame
    #10 // attendiamo 10 unità di tempo
    reset = 1'b1; // ora il sistema può funzionare
end
/// FINE NUOVO CODICE
// CONTINUAZIONE
...
endmodule

```

▼ Comportamento del sistema

Definiamo gli stimoli da applicare: in questo caso, essi saranno dentro un blocco `initial`, poiché non ha senso testarlo all'infinito, ma basta testarlo con le 4 combinazioni possibili di `a` e `b`.

```

`timescale 1ns / 1ps
module example_tb();
reg clock;
wire reset, a, b;
wire Q; // il testbench di solito non ha ingressi e uscite!
// INSTANZIAZIONE module "example"
example dut(
    .Q(Q),
    .clk(clock),
    .rst(reset),
    .a(a),
    .b(b)
);

// istanziazione clock tramite ciclo infinito
initial begin
    clock = 1'b0;
    forever begin
        #1 clock = ~clock;
    end
end

```

```

    end
end

// facciamo variare il reset...
initial begin
    reset = 1'b0; // inizializziamo il sistema in esame
    #10 // attendiamo 10 unità di tempo
    reset = 1'b1; // ora il sistema può funzionare
end

/// NUOVO CODICE QUI
initial begin
    $monitor("time=%3d, in_a=%b, in_b=%b, q=%b\n", $time, a, b, q);
    // STAMPIAMO A SCHERMO AD OGNI VARIAZIONE DI UNO TRA a, b, time e q
    a = 1'b0;
    b = 1'b0;
    // configurazione a = 0, b = 0
    #20 // propagazione dei valori pari a 20 unità di tempo = 20ns
    a = 1'b1;
    // configurazione a = 1, b = 0
    #20 // propagazione dei valori pari a 20 unità di tempo = 20ns
    a = 1'b0;
    b = 1'b1;
    // configurazione a = 0, b = 1
    #20 // propagazione dei valori pari a 20 unità di tempo = 20ns
    a = 1'b1;
    // configurazione a = 1, b = 1
end
/// FINE NUOVO CODICE

endmodule

```

Esempio - Test di un up-counter a 4 bit

Consideriamo un up-counter a 4 bit come elemento da modellare e testare, con:

- un segnale di clock che regola la temporizzazione del conteggio;
- un segnale di reset, attivo con il valore basso, per gestire il reset del dispositivo;
- un'uscita su 4 bit per rappresentare il conteggio;

Questo componente, al raggiungimento del valore massimo del conteggio, riparte dal valore 0.

```

module uc4b (out, clk, reset);
    output reg[3:0] out;
    input clk, reset;

    always @(posedge clk)
        if (!reset)
            out <= 0; // 4'b0000
        else
            out <= out + 1;

```

```
endmodule
```

Nel seguente frammento di codice, si può vedere una modalità alternativa per fare il ciclo `forever` con `always #5 clk = ~clk`, per cui è necessario definire un primo valore del clock. In caso non si facesse, si avrebbe valore `x`, che ricordiamo essere il valore sconosciuto. Per questo motivo si vede l'istruzione `clk <= 0` nel blocco `initial`.

```
module uc4b_tb ();
    reg clk, reset;
    wire [3:0] out;

    uc4b c(.clk(clk), .reset(reset), .out(out));

    always #5 clk = ~clk;

    initial begin
        $monitor("[%0tns] clk=%b rstn=%b out=0x%0h", $time, clk, reset, out);
        clk <= 0;
        reset <= 0;
        #20 reset <= 1;
        #80 reset <= 0;
        #50 reset <= 1;
        #20 $finish;
    end
endmodule
```



Descrizione di registri, ROM, RAM in Verilog

▼ Creatore originale: @Gianbattista Busonera

Registri

Esempio - Registro a 4 bit

Componenti

Versione del codice sulle slide

Versione del codice corretta

Testbench e verifica (entrambe le versioni)

Esempio - Registro parametrico

Esempio - Registro a 4 bit con FF-D

Register File

Spiegazione generica Register File 4×32

Implementazione strutturale del Register File 4×32

Implementazione Register File 8×8 con 2 porte di lettura e 1 di scrittura

Random Access Memory (RAM)

Implementazione RAM parametrica

Read Only Memory (ROM)

Inizializzazione delle memorie

Esempi - Inizializzazione

Implementazione ROM parametrica

In questa sezione verrà approfondito l'utilizzo del blocco always (modellazione comportamentale) per descrivere registri e memorie.

Registri

Nella maggioranza dei casi, considereremo Flip-Flop attivi sul fronte di salita/discesa del clock, chiamati edge-triggered Flip Flop. È anche possibile descrivere registri di tipo Latch, che sono attivi su un segnale di enable.

Esempio - Registro a 4 bit

Si vuole definire un registro da 4 bit per modellare circuiti di memoria tali che consentano una sola operazione alla volta, sia essa di lettura, di scrittura o di reset sul fronte di salita del clock.

Si vuole fare in modo che il segnale di `reset` (attivo con il valore basso) imposti il contenuto del registro a zero e che il segnale `read_write` si comporti come segue:

- se `read_write == 0`, allora voglio effettuare una scrittura;
- se `read_write == 1`, allora voglio effettuare una lettura.

▼ Componenti

Per tale scopo, sono necessari:

- una variabile di tipo `reg` da 4 bit, che sarà l'uscita del blocco `always`;
- un selettore di lettura/scrittura, chiamato `read_write`;
- un segnale di clock che sincronizza la scrittura e il reset;
- un segnale di reset (attivo al valore basso) che consente di impostare il valore nel registro a 0;
- un input `write_data` per la scrittura dei dati nel registro;
- un output `read_data` su cui sarà disponibile il dato memorizzato nel registro quando viene effettuata un'operazione di lettura, altrimenti si avrà alta impedenza, portando a un comportamento simile ad un buffer tri-state.

▼ Versione del codice sulle slide

Questo codice è presente nelle slide del corso, ma risulta errato.

Il problema è che l'operatore `assign` descrive bene una assegnazione combinatoria, ma non una sequenziale: infatti `read_data` assume il valore in storage ogni qual volta `rw = 1`, indipendentemente dal clock, mentre viene richiesta una lettura sul fronte di salita del clock.

```
module reg4b (output [3:0] read_data,
              input [3:0] write_data,
              input clk, rst, rw);

    reg [3:0] storage; // rappresenta il contenuto (memoria) del registro
```

```

initial storage = 4'b0001; // SOLO PER TESTBENCH

assign read_data = (rw ? storage: 4'bZ); // se il segnale read_write = 1 assegna
// read_data il valore contenuto in memoria, diversamente alta impedenza.

always @(posedge clk) begin // sul fronte di salita del clock...
    if(rst == 0) // se reset == 0, devo resettare il contenuto.
        storage = 0;
    else if(rw == 0)
        storage = write_data;
    end
endmodule

```

▼ Versione del codice corretta

```

module reg4b (read_data, write_data, clk, rst, rw);
    output reg [3:0] read_data; // uscita su 4 bit
    input [3:0] write_data; // ingresso su 4 bit
    input clk, rst, rw; // ingressi vari su 1 bit

    reg [3:0] storage;

    initial storage = 4'b0001; // SOLO PER TESTBENCH
    // SCRITTURA E RESET
    always @(posedge clk) begin
        if (rst == 0) begin // DIAMO PRIORITA' al reset
            storage = 0;
            read_data = 0; // Reset anche dell'uscita
        end
        else if (rw == 0) begin // Scrittura → rw = 0
            storage = write_data; // Scrittura nei "FF"
            read_data = 4'bZ; // Alta impedenza durante la scrittura
        end
        else
            read_data = storage; // Lettura sincronizzata al fronte di salita
    end
endmodule

```

```
    end  
endmodule
```

▼ Testbench e verifica (entrambe le versioni)

```
module reg4b_tb;  
  
    reg clk;          // Clock  
    reg rst;          // Reset  
    reg rw;           // Read/Write control  
    reg [3:0] write_data; // Input data to write to the register  
    wire [3:0] read_data; // Output data from the register  
  
    // Instanza del registro a 4 bit  
    reg4b uut (  
        .read_data(read_data),  
        .write_data(write_data),  
        .clk(clk),  
        .rst(rst),  
        .rw(rw)  
    );  
  
    // Generazione del clock (periodo 10 ns)  
    always begin  
        #5 clk = ~clk; // Periodo di clock di 10 ns  
    end  
  
    // Stimoli di input  
    initial begin  
        // Inizializza i segnali  
        clk = 0;  
        rst = 1; // Inizia con reset disabilitato  
        rw = 0; // Iniziamo con modalità scrittura  
        write_data = 4'b0000; // Scriviamo un valore iniziale  
  
        // Aspetta un po' prima di testare  
        #10;
```

```

// Reset attivo per il primo test (rst=0)
rst = 0; // Applicare il reset
#10; // Attendere un ciclo di clock per completare il reset

// Fine del reset
rst = 1; // Disabilitare il reset
write_data = 4'b1010; // Scrivere un nuovo dato (1010)

// Scrittura nel registro con rw = 0
rw = 0; // Modalità scrittura
#10; // Aspettare un ciclo di clock per la scrittura

// Lettura dal registro con rw = 1
rw = 1; // Modalità lettura
#10; // Aspettare un altro ciclo di clock

// Scrittura di un nuovo valore
write_data = 4'b1111;
rw = 0; // Modalità scrittura
#10; // Aspettare un ciclo di clock

// Lettura del nuovo valore
rw = 1; // Modalità lettura
#10; // Aspettare un altro ciclo di clock

// Reset del registro
rst = 0; // Applicare il reset
#10; // Attendere un ciclo di clock per il reset

// Verifica di lettura dopo il reset (dovrebbe essere 0000)
rst = 1; // Disabilitare il reset
rw = 1; // Modalità lettura
#10; // Aspettare un ciclo di clock

// Finire la simulazione
$finish;

```

```

end

// Monitor personalizzato: stampa dopo ogni fronte di salita del clock con un p
always @(posedge clk) begin
    #0.01; // Ritardo piccolo dopo il fronte di salita
    $display("Time: %0t | clk: %b | rst: %b | rw: %b | write_data: %b | read_data: %b",
             $time, clk, rst, rw, write_data, read_data);
end
endmodule

```

1. Nella versione del codice presente sulle slide si ottiene il seguente output:

Si nota facilmente come, nel caso della riga in cui `time=30`, ci aspettavamo che `read_data` fosse pari a `zzzz`, in quanto il clock è allo stato basso e precedentemente `read_data` era impostato a `zzzz`.

Ciò significa che **ho letto anche se il clock era allo stato basso**, che è un comportamento non voluto.

2. Nella versione corretta del codice si ottiene il seguente output:

Si noti come, al tempo `time=40`, troviamo `read_data=1010`, che risulta diverso da `write_data=1111`, come ci si aspetta, essendo sul fronte di discesa del clock, e per cui il valore di `read_data` non dovrebbe essere modificato.

Infatti, il valore verrà letto al tempo `time=55`, dove abbiamo che `clk=1` e che `rw=1`, portando all'effettiva lettura del valore in scrittura.

```

Time: 0 | clk: 0 | rst: 1 | rw: 0 | write_data: 0000 | read_data: zzzz
Time: 5 | clk: 1 | rst: 1 | rw: 0 | write_data: 0000 | read_data: zzzz
Time: 10 | clk: 0 | rst: 0 | rw: 0 | write_data: 0000 | read_data: zzzz
Time: 15 | clk: 1 | rst: 0 | rw: 0 | write_data: 0000 | read_data: zzzz
Time: 20 | clk: 0 | rst: 1 | rw: 0 | write_data: 1010 | read_data: zzzz
Time: 25 | clk: 1 | rst: 1 | rw: 0 | write_data: 1010 | read_data: zzzz
Time: 30 | clk: 0 | rst: 1 | rw: 1 | write_data: 1010 | read_data: 1010
Time: 35 | clk: 1 | rst: 1 | rw: 1 | write_data: 1010 | read_data: 1010
Time: 40 | clk: 0 | rst: 1 | rw: 0 | write_data: 1111 | read_data: zzzz
Time: 45 | clk: 1 | rst: 1 | rw: 0 | write_data: 1111 | read_data: zzzz
Time: 50 | clk: 0 | rst: 1 | rw: 1 | write_data: 1111 | read_data: 1111
Time: 55 | clk: 1 | rst: 1 | rw: 1 | write_data: 1111 | read_data: 1111
Time: 60 | clk: 0 | rst: 0 | rw: 1 | write_data: 1111 | read_data: 1111
Time: 65 | clk: 1 | rst: 0 | rw: 1 | write_data: 1111 | read_data: 0000
Time: 70 | clk: 0 | rst: 1 | rw: 1 | write_data: 1111 | read_data: 0000
Time: 75 | clk: 1 | rst: 1 | rw: 1 | write_data: 1111 | read_data: 0000
reg4b_tb.v:69: $finish called at 80 (1s)
Time: 80 | clk: 0 | rst: 1 | rw: 1 | write_data: 1111 | read_data: 0000

```

Output della prima versione del codice

N.B. il reset è attivo basso!

```

Time: 0 | clk: 0 | rst: 1 | rw: 0 | write_data: 0000 | read_data: xxxx
Time: 5 | clk: 1 | rst: 1 | rw: 0 | write_data: 0000 | read_data: zzzz
Time: 10 | clk: 0 | rst: 0 | rw: 0 | write_data: 0000 | read_data: zzzz
Time: 15 | clk: 1 | rst: 0 | rw: 0 | write_data: 0000 | read_data: 0000
Time: 20 | clk: 0 | rst: 1 | rw: 0 | write_data: 1010 | read_data: 0000
Time: 25 | clk: 1 | rst: 1 | rw: 0 | write_data: 1010 | read_data: zzzz
Time: 30 | clk: 0 | rst: 1 | rw: 1 | write_data: 1010 | read_data: zzzz
Time: 35 | clk: 1 | rst: 1 | rw: 1 | write_data: 1010 | read_data: 1010
Time: 40 | clk: 0 | rst: 1 | rw: 0 | write_data: 1111 | read_data: 1111
Time: 45 | clk: 1 | rst: 1 | rw: 0 | write_data: 1111 | read_data: 1111
Time: 50 | clk: 0 | rst: 1 | rw: 1 | write_data: 1111 | read_data: 1111
Time: 55 | clk: 1 | rst: 1 | rw: 1 | write_data: 1111 | read_data: zzzz
Time: 60 | clk: 0 | rst: 0 | rw: 1 | write_data: 1111 | read_data: 1111
Time: 65 | clk: 1 | rst: 0 | rw: 1 | write_data: 1111 | read_data: 0000
Time: 70 | clk: 0 | rst: 1 | rw: 1 | write_data: 1111 | read_data: 0000
Time: 75 | clk: 1 | rst: 1 | rw: 1 | write_data: 1111 | read_data: 0000
reg4b_tb.v:69: $finish called at 80 (1s)
Time: 80 | clk: 0 | rst: 1 | rw: 1 | write_data: 1111 | read_data: 0000

```

Output della seconda versione del codice

N.B. il reset è attivo basso!

▼ Esempio - Registro parametrico

Si vuole realizzare un [registro parametrico](#) a numero di bit variabile (N).

Si vuole fare in modo che si possa leggere, scrivere e resettare solo sul fronte di salita del clock.

A differenza del caso precedente, si propone solo l'implementazione del creatore della pagina, in quanto sulle slide è presente lo [stesso errore già visto](#).

```
module #(parameter N = 4) regNb (
    output reg [N-1:0] read_data, // Dichiarato come 'reg' perché assegnato in un
    input [N-1:0] write_data,
    input clk, rst, rw
);
    reg [N-1:0] storage; // Utile per la memoria!

    always @(posedge clk) begin
        if (!rst) begin
            storage = 0;
            read_data = 0; // Reset anche dell'uscita
        end
        else if (!rw) begin
            storage = write_data; // Scrittura
            read_data = {N{1'bZ}}; // Alta impedenza durante la scrittura
            // CREA UN VETTORE D N bit con tutti Z
        end
        else
            read_data = storage; // Lettura sincronizzata al fronte di salita
    end
endmodule
```

▼ Esempio - Registro a 4 bit con FF-D

Potremmo descrivere un registro a 4 bit mediante modellazione strutturale, assumendo di avere un componente Flip-Flop D ([ffD](#)) del tipo:

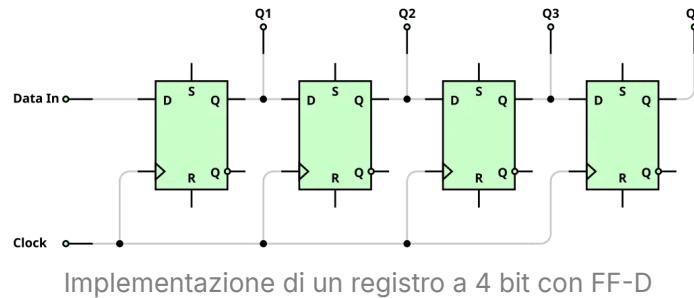
```
module ffD(output reg Q, input D, clk);
    always @ (posedge clk)
```

```

Q=D;
endmodule

```

Si possono, quindi, utilizzare 4 componenti **ffd** per implementare un registro a 4 bit.



Implementazione di un registro a 4 bit con FF-D

```

module reg4b(inout[3:0] out, input [3:0] in, clk, rw);
    ffd f3(out[3], rw ? out[3] : in[3], clk);
    ffd f2(out[2], rw ? out[2] : in[2], clk);
    ffd f1(out[1], rw ? out[1] : in[1], clk);
    ffd f0(out[0], rw ? out[0] : in[0], clk);
endmodule

```

In scrittura (`rw == 0`) si propaga l'input (`in`) agli ingressi **D** dei FF.

In lettura (`rw == 1`) scriviamo nuovamente in uscita il valore precedentemente memorizzato.

Visto che nella modellazione strutturale non è possibile utilizzare costrutti condizionali, è necessario utilizzare un costrutto condizionale ternario per selezionare l'ingresso **D** dei FF.



La versione illustrata differisce leggermente da quella implementata sulle slide, ma la reputiamo più immediata per comprendere che l'uscita resta invariata, mentre l'ingresso è fornito dalla vecchia uscita nel caso di lettura (mantenendo in memoria) o dal nuovo input nel caso in cui si debba scrivere. Si riporta anche il codice delle slide per completezza, pur sapendo che contiene un errore.

```
module reg4b(output [3:0] out, input [3:0] in, clk, rw);
    wire [3:0] Q;

    ffD f3(Q[3], rw ? Q[3] : in[3], clk);
    ffD f2(Q[2], rw ? Q[2] : in[2], clk);
    ffD f1(Q[1], rw ? Q[1] : in[1], clk);
    ffD f0(Q[0], rw ? Q[0] : in[0], clk);

    assign out <= Q; // non si può fare questa assegnazione non bloccante
    // assign out = Q; è corretto
endmodule
```

In questo caso, il docente ha preferito utilizzare una variabile di appoggio Q probabilmente per aggiungere della logica combinatoria tra l'uscita Q e l'uscita out.

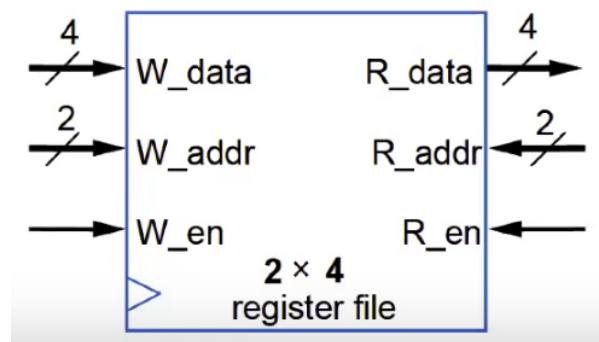
Risulta essere leggermente più verboso, ma segue lo stesso principio del codice implementato in precedenza.

Register File

Il Register File è un blocco di tanti registri (matrice di registri) che possono avere almeno una porta di lettura (solitamente almeno 2) e almeno una di scrittura.

Un esempio di Register File è descritto in [figura](#), ed è formato da:

- 4 registri da 4 bit ciascuno;
 - Di conseguenza, saranno necessari $\log_2(4) = 2$ bit di indirizzo ($R_{address}$ e $W_{address}$);
 - W_{data} e R_{data} sono su 4 bit rispettivamente
- un segnale di abilitazione per la scrittura;
- un segnale di abilitazione per la lettura.



Esempio di Register File

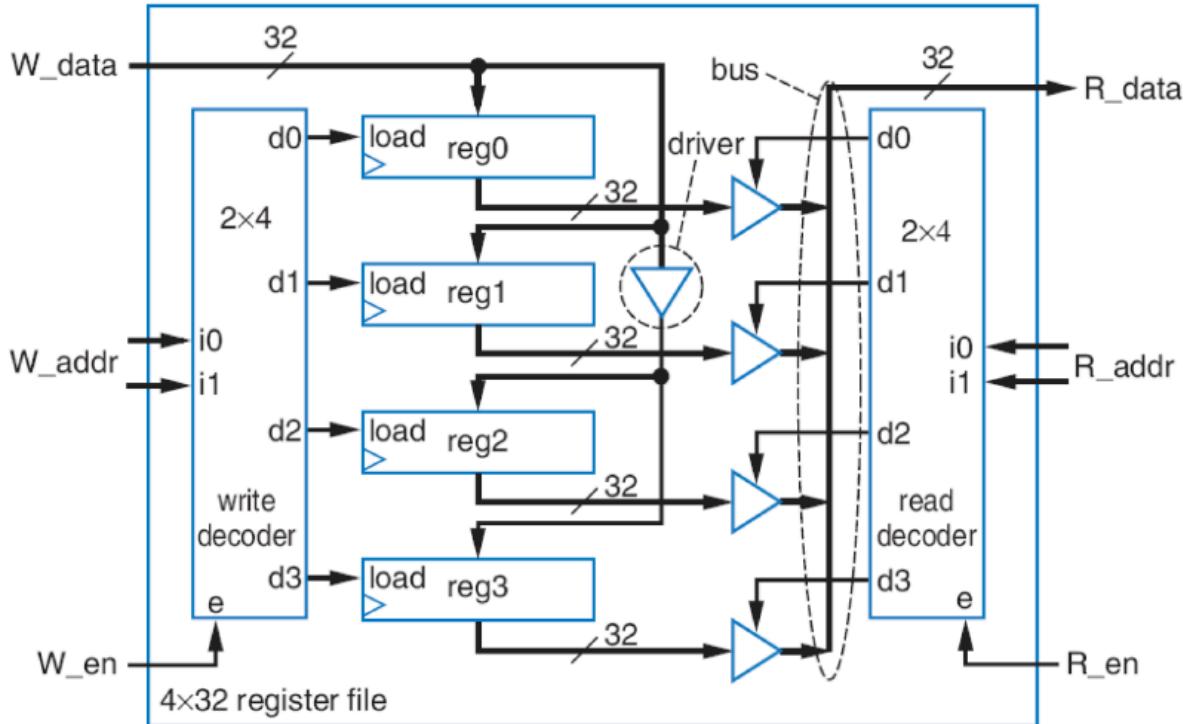
▼ Spiegazione generica Register File 4×32

Facciamo una breve spiegazione del funzionamento del Register File 4×32:

- caratteristiche generali:
 - parallelismo dei dati su 32 bit;
 - il numero di registri è 4.
- scrittura:
 - è necessario un decodificatore di scrittura che, dato l'indirizzo su 2 bit (in quanto ci sono 4 registri in totale), seleziona quale dei 4 registri attivare per la scrittura;
- lettura:
 - è necessario un altro decodificatore che, dato l'indirizzo su 2 bit, seleziona quale dei 4 buffer tri-state collegati ai registri abilitare.
 - nel caso in cui il buffer tri-state sia abilitato, quest'ultimo lascia passare la parola memorizzata nel rispettivo registro, altrimenti impone l'alta impedenza sul bus.



Ricordiamo cosa succede nel caso di combinazioni logiche con buffer tri-state.



Visualizzazione del circuito interno generico per un Register File 4x32

▼ Implementazione strutturale del Register File 4x32

Si immagini un file che descrive il modulo `reg32en`, il quale dovrà comportarsi come un registro con:

- uscita di lettura;
- ingresso di scrittura;
- condizione di abilitazione a lettura;
- condizione di abilitazione alla scrittura;
- clock e reset.

I due decoder sono presenti poiché, in base al valore dell'indirizzo `r_addr`, definiscono quale degli elementi va effettivamente letto/scritto.

In tal senso, si riporta anche l'implementazione strutturale, con:

- 4 registri da 32 bit con output `enable`;
- 2 decoder 2:4 con segnale di `enable` per abilitare letture e scritture nei singoli registri interni, per cui:
 - `enable` attivo abilita l'uscita decodificata dai decoder;
 - `enable` inattivo imposta tutte le uscite a 0.

```
module reg4×32 (r_data , w_data, r_addr, w_addr, r_en, w_en, clk, rst);
    output [31:0] r_data;
    input [31:0] w_data;
    input [1:0] r_addr, w_addr;
    input r_en, w_en, clk, rst;

    wire w_r3, w_r2, w_r1, w_r0, r_r3, r_r2, r_r1, r_r0;
    dec2×4en r_dec (r_addr[1], r_addr[0], r_en, r_r3, r_r2, r_r1, r_r0);
    dec2×4en w_dec (w_addr[1], w_addr[0], w_en, w_r3, w_r2, w_r1, w_r0);

    reg32en r0 (r_data, w_data, r_r0, w_r0, clk, rst);
    reg32en r1 (r_data, w_data, r_r1, w_r1, clk, rst);
    reg32en r2 (r_data, w_data, r_r2, w_r2, clk, rst);
    reg32en r3 (r_data, w_data, r_r3, w_r3, clk, rst);
endmodule
```

▼ Implementazione Register File 8×8 con 2 porte di lettura e 1 di scrittura

Si voglia implementare un Register file con 8 registri da 8 bit ciascuno. Si faccia in modo che ci siano 2 porte di lettura e 1 porta di scrittura, e che, se reset è attivo basso in modalità sincrona, si pulisca completamente il Register File.

Si vuole fare in modo che si possa leggere indipendentemente dal segnale di clock e che si possa selezionare il segnale `r1r2w` per selezionare l'operazione da effettuare, e che si possa scrivere e resettare solo sul fronte di salita del clock, come segue:

- se `r1r2w=1`, si legge dalla porta di lettura 1;
- se `r1r2w=2`, si legge dalla porta di lettura 2;
- se `r1r2w=0`, si scrive sulla linea selezionata.

```

module regFile (data_out1, data_out2, data_in, clk, rst, r1r2w, address);
    // DICHIARAZIONI
    output [7:0] data_out1, data_out2;
    input [7:0] data_in;
    // Poichè si può leggere o scrivere solo da una porta per volta
    // non ha senso utilizzare 3 indirizzi differenti come ha fatto lui.
    // io userò solo un indirizzo "address", anzichè 3 distinti.
    input [2:0] address; // servono 3 bit per selezionare le 8 parole.
    input clk, rst;
    input [1:0] r1r2w; // Segnale di selezione

    reg [7:0] storage [7:0]; // matrice 8×8 → 8 registri da 8 bit ciascuno

    assign data_out1 = (r1r2w == 1) ? storage[address] : 'bZ;
    assign data_out2 = (r1r2w == 2) ? storage[address] : 'bZ;
    // Le due assegnazioni continue sopra, nel caso in cui non si sia stato
    // selezionato correttamente la porta di lettura, mettono l'uscita in alta
    // impedenza su tutti i bit disponibili

    always @(posedge clk) begin
        if(rst == 0) // Se ho deciso di resettare
            storage = 0; // azzera tutti i registri in automatico su fronte di salita
        else if(r1r2w == 0)
            storage[address] = data_in; // WRITE solo su fronte di salita
    end
endmodule

```

Random Access Memory (RAM)

Una memoria RAM è una memoria sia leggibile che scrivibile in un tempo unitario attraverso degli indirizzi. Da qui il nome Random Access Memory, a discapito delle memorie ad accesso sequenziale.

Praticamente è identica a un Register File, ma con alcune piccole differenze:

- una RAM ha solitamente una sola porta condivisa per lettura e scrittura;
- una RAM è in genere più grande delle 512 o 1024 parole dei Register File;

- una RAM memorizza i bit in maniera più efficiente dei comuni Flip Flop e, oltretutto, è solitamente implementata su chip di forma quadrata per ridurre le lunghezze delle connessioni più lunghe.

▼ Implementazione RAM parametrica

Si vuole implementare una RAM parametrica con una porta condivisa per lettura e scrittura.

Da specifica vogliamo che:

- le operazioni di lettura e scrittura devono essere sincrone al fronte di salita del clock;
- la RAM è abilitata solo quando il "chip select" `cs` è attivo alto;
- se "write enable" `we = 1` e `cs = 1`, si può scrivere;
- se "output enable" `oe = 1` e `cs = 1`, possiamo leggere, assicurandoci che `we = 0`, poiché l'operazione di lettura può essere eseguita solo quando non è in esecuzione l'operazione di scrittura, in modo da evitare inconsistenze di qualsiasi tipo.

```
module RAM(data, address, clk, cs, we, oe);
    // DICHIARAZIONE PARAMETRI
    parameter DATA_WIDTH = 8; // parallelismo dei dati pari a 8 di default
    parameter ADDR_WIDTH = 8; // indirizzi su 8 bit di default
    // Poichè abbiamo indirizzi su 8 bit ⇒ possiamo scrivere 2^8 parole

    parameter WORDS_COUNT = 1 << ADDR_WIDTH; // shift a sinistra di "1"
    // tante volte quanto è il valore di "ADDR_WIDTH" così che WORDS_COUNT = :
    // DICHIARAZIONE INGRESSI

    inout [DATA_WIDTH-1:0] data; // bus condiviso per leggere e scrivere con par-
    // selezionato in precedenza
    input [ADDR_WIDTH-1:0] address;
    input clk, cs, we, oe;

    // DEFINISCO LA VERA E PROPRIA MEMORIA DI DATA_WIDTH * WORDS_COUI
    reg [DATA_WIDTH-1:0] mem [WORDS_COUNT-1:0];
    // Definisco una variabile ausiliaria per leggere altrimenti dovrei
```

```

// istanziare data come reg (a causa dell'always)!
reg [DATA_WIDTH-1:0] data_out;
// Definisco un'altra variabile ausiliaria
reg oe_r; // output enable registrato... Ci sarà un flip flop
// che contiene tale valore sulla base della condizione di lettura.

/// SCRITTURA
always @(posedge clk) begin
    if(cs && we) // se chip select = 1 (ram attiva) e write enable = 1 (scrittura)
        mem[address] = data; // memorizzo nell'apposito indirizzo la parola
end
/// LETTURA
always @(posedge clk) begin
    if(cs && oe && !we) begin // se sono nella condizione di poter leggere
        data_out = mem[address]; // assegno al registro temporaneo data_out il v
        // letto
        oe_r = 1; // segno nella variabile ausiliaria che ho registrato il valore letto
    end
    else
        oe_r = 0; // non ho letto effettivamente il valore
end

assign data = (cs && oe_r && !we) ? data_out : 'bZ;
// Assegnazione continua che mi consente di memorizzare data_out nel caso in
// abbia effettivamente letto qualcosa, altrimenti alta impedenza.

// Nota come non potevamo usare "oe" direttamente al posto di "oe_r",
// sarebbe stato infatti possibile leggere da data_out anche nel caso in cui
// non avessimo ancora letto dal blocco always!
// e, oltretutto, si fa in modo che SOLO sul fronte di salita del clock
// si legga effettivamente grazie al fatto che oe_r può valere 1 solo
// durante un fronte di salita del clock.
endmodule

```

Read Only Memory (ROM)

Una memoria ROM è una memoria che può essere letta, ma non modificata, per cui è necessario un solo bus di lettura non condiviso `e`, e per cui di conseguenza non servono segnali di selezione read/write come la coppia `oe` e `we`.

Rispetto alle RAM, ci sono alcuni vantaggi, analizzati meglio nella sezione di teoria relativa:

- maggiore compattezza, dovuta al fatto che devono solo essere lette;
- essendo una memoria non volatile, può essere più veloce di alcuni tipi di RAM;
- bassa potenza, e non serve alimentazione per conservare i bit.

Chiaramente, si utilizza una ROM rispetto a una RAM se i dati da memorizzare cambiano raramente o, meglio, non cambiano proprio.

Inizializzazione delle memorie

Verilog consente di leggere da file per inizializzare vettori o matrici.

In particolare, si possono leggere file in formato:

- esadecimale tramite `syscall $readmemh` ;
- binario tramite `syscall $readmemb` .

```
$readmemh("hex_file.txt", mem_array, [start_address], [end_address]);
// gli ultimi due parametri sono opzionali e ci consentono di inizializzare
// una sottoparte del vettore o della matrice
$readmemb("bin_file.txt", mem_array, [start_address], [end_address]);
```

Nel caso in cui il contenuto del file non sia coerente con la dimensione della memoria, la lettura viene troncata fino all'ultima locazione di memoria disponibile.



I valori nei file di input possono essere separati:

- Singoli spazi ' ';
- TAB: ' ';
- Carattere a capo: '\n' ;

Tutti questi separatori possono anche essere mischiati tra loro.

Nel file di input si possono aggiungere dei commenti con ' // '.

▼ Esempi - Inizializzazione

1. Primo esempio:

```
reg [15:0] es1 [0:3];
$readmemh("es1_file.mem", es1);
// es1 è un registro che può
// contenere 4 parole da 16 bit
```

```
dead // Commento
be ef
0 a 0 a
1234
```

2. Secondo esempio:

```
reg [2:0] es1 [0:5];
$readmemb("es2_file.mem", es2);
// es2 è un registro che può
// contenere 6 parole da 3 bit
```

```
011 101 111
111
001 101
```

3. Terzo esempio:

Si noti come viene specificato l'indirizzo di partenza!

La stringa

dead verrà inserita in es[4] e le successive all'indirizzo 5,6,7, e così via.

```
reg [15:0] es3 [0:255];
$readmemh("es3_file.mem", es3, 4
// es3 è un registro che può
// contenere 256 parole da 16 bit
```

```
dead beef aaaa
casu 0110 aabb
9876 5432 1001
```

! Si noti come si vuole fare in modo che la riga 0 sia la più significativa!
Facendo così, possiamo trovare in memoria il file caricato in ordine.

▼ Implementazione ROM parametrica

In questo frammento di codice, verrà utilizzata per la prima volta la parola chiave `initial`, la quale permette di eseguire delle istruzioni (che possano essere letture da file, blocchi always...) solo all'inizio della simulazione del modulo.

```
module ROM (data, // data output
            clk, // clock input
            address, // address input
            re, // Read enable, analogo a output enable
            cs); // chip select per abilitazione ROM

    // DICHIARAZIONE PARAMETRI
    parameter DATA_WIDTH = 8; // parallelismo dei dati pari a 8 di default
    parameter ADDR_WIDTH = 8; // indirizzi su 8 bit di default
    // Poichè abbiamo indirizzi su 8 bit ⇒ possiamo leggere 2^8 parole
    parameter WORDS_COUNT = 1 << ADDR_WIDTH; // shift a sinistra di "1"
    // tante volte quanto è il valore di "ADDR_WIDTH" così che WORDS_COUNT = 2^ADDR_WIDTH

    // DICHIARAZIONE INGRESSI
    output [DATA_WIDTH-1:0] data; // bus per leggere con parallelismo pari a data_width
    input [ADDR_WIDTH-1:0] address;
    input clk, cs, re;

    // DEFINISCO LA VERA E PROPRIA MEMORIA DI DATA_WIDTH * WORDS_COUNT
    reg [DATA_WIDTH-1:0] mem [0:WORDS_COUNT-1];
    reg re_r; // di nuovo variabile ausiliaria per riprendere il codice RAM

    // INIZIALIZZAZIONE DELLA ROM
    initial
        $readmemb ("input.data", mem); // lettura da file
        // Solo ad inizio simulazione viene memorizzato nel registro mem il contenuto
```

```
// del file 'input.data'

// LETTURA
always @(posedge clk) : MEM_READ // etichetta 'MEM_READ' utile per debug
begin
    if(cs && re)
        re_r = 1;
    else
        re_r = 0;
end

assign data = (cs && re_r) ? mem[address] : 'bZ;
// Analogico a quanto visto e spiegato in precedenza.
endmodule
```

1

Esercitazione 1 - Progetto guidato di un circuito

Esercizio 1 - Rilevamento della velocità dei veicoli

Obiettivo

[Logica della realizzazione](#)

[Descrizione del circuito](#)

Esercizio 2 - Calcolo della frequenza massima

Obiettivo

[Tempistiche del circuito](#)

[Calcolo della frequenza massima](#)

[Verifica delle violazioni di hold](#)

Esercizio 1 - Rilevamento della velocità dei veicoli

▼ Creatore originale: @Giacomo Dandolo

- @<Utente>(<Data>): <Descrizione della modifica>

Obiettivo

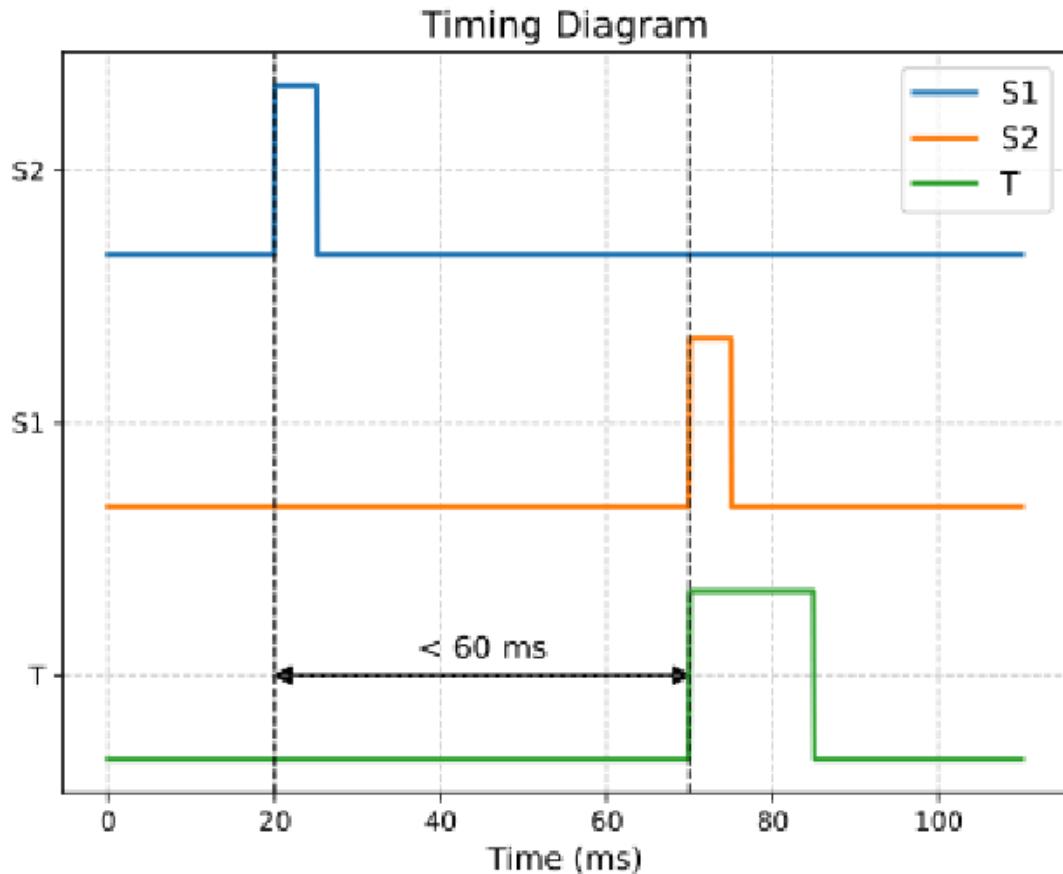


Diagramma temporale del circuito da creare

Si vuole realizzare il comportamento del [diagramma temporale](#) attraverso un circuito logico.

▼ Logica della realizzazione

La logica della realizzazione è:

1. generare un segnale Q che va a 1 nell'intervallo di tempo tra i due segnali dei sensori S1 e S2, utilizzando un componente che viene impostato (SET) dal segnale del sensore S1 e resettato (RESET) dal segnale del sensore S2.

Il componente utilizzato è un Flip-Flop SR, poiché permette di impostare ($S = \text{SET}$) e resettare ($R = \text{RESET}$) il valore Q in uscita;

2. misurare la durata della fase in cui Q è a 1.

Il componente utilizzato è un clock accoppiato ad un contatore, che permette di contare (e quindi misurare) la durata della fase;

3. confrontare la durata con una soglia.

- Durata superiore alla soglia: velocità inferiore al limite;
- Durata inferiore alla soglia: velocità superiore al limite, bisogna attivare la telecamera.

Il componente utilizzato è un comparatore, che permette di confrontare l'ingresso A con l'ingresso B aritmeticamente, portando una delle tre uscite ($>$, $<$, $=$) a 1.

▼ Descrizione del circuito

Circuito in analisi

Analizziamo le varie parti:

- (1) Il FF-SR è impostato per eseguire SET quando $S_1 = 1$, mentre eseguire RESET quando $S_2 = 1$, in modo da, rispettivamente, iniziare o terminare la conta.
- (2) Si utilizza il FF-D sincrono con un clock CK per sincronizzare l'uscita Q del FF-SR, in modo da evitare che ci siano inconsistenze nei periodi di conta.
- (3) Impostando che il tempo di clock sia $T_{CK} = 10ms$, si imposta come soglia CNTR = 6:

$$\begin{cases} X = 0 & \text{CNTR} \geq 6 \\ X = 1 & \text{CNTR} < 6 \end{cases}$$

(4) Se il segnale $Q_0 = \overline{Q} = 1$, il segnale di CLR viene attivato, poiché il segnale di RESET è impostato a 1, e si è quindi resettato il valore del counter di tempo. Se il segnale $Q_1 = 1$ oppure $Q_2 = 1$, si ha che FZ = 1.

(5) Se i segnali $\overline{Q} = 0$ (segnale di SET è impostato a 1, si sta ancora contando), $X = 1$ e $FZ = 1$ hanno i valori definiti, allora si deve attivare la telecamera.

Per attivarla, si usano due FF-D:

- il primo FF-D sincrono permette di memorizzare il valore di $Y_0 = FZ \cdot \overline{Q} \cdot X$, portando in uscita $Y_1 = \overline{Y}_0$;

- il secondo FF-D sincrono permette di memorizzare il valore di $Y_1 = Y_2$.

Si definisce $T = Y_1 \cdot Y_2$.

$$\begin{cases} T = 0 & \text{telecamera disattivata} \\ T = 1 & \text{telecamera attivata} \end{cases}$$

Questa implementazione utilizza due FF-D perché si utilizza un comparatore che imposta $X = 1$ quando $\text{CNTR} < 6$, ma la telecamera si deve attivare quando $\text{CNTR} = 6$. Grazie al clock, il quale sincronizza tutto il circuito, è necessario che sia Y_1 che Y_2 siano uguali a 1 nello stesso periodo di clock, ossia quando $Y_0 = 1$ per due periodi di clock successivi.

Esercizio 2 - Calcolo della frequenza massima

▼ Creatore Originale: @Giacomo Dandolo

- @Giacomo Dandolo (13/04/2025): aggiunti i collegamenti ad argomenti di teoria.
- @<Utente>(<Data>): <Descrizione della modifica>

Obiettivo

Negli esercizi precedenti abbiamo considerato $T_{\text{CK}} = 10 \text{ ms}$ come periodo di clock, con frequenza di clock $F_{\text{CK}} = \frac{1}{T_{\text{CK}}} = 100 \text{ Hz}$. L'errore di misura, però, risulta molto grande, dell'ordine di $\pm 10 \text{ ms}$, ed è quindi molto elevato. Conviene, quindi, utilizzare una frequenza di clock più alta.

Qual è la massima frequenza di clock F_{max} ?

▼ Tempistiche del circuito

Riportiamo le tempistiche del circuito:

- tempo di propagazione della porta OR: $t_{\text{OR}} = 1 \text{ ns}$;
- tempo di propagazione della porta AND: $t_{\text{AND}} = 2 \text{ ns}$;

- tempo di propagazione del comparatore: $t_{\text{comp}} = 10 \text{ ns}$;
- FF-D e contatore:
 - tempo di propagazione del clock;

$$t_{\text{CK-Q}} = 1 \text{ ns}$$

- tempo di setup;

$$t_{\text{SU}} = 0.8 \text{ ns}$$

- tempo di hold.

$$t_{\text{H}} = 0.5 \text{ ns}$$

▼ Calcolo della frequenza massima

Si devono considerare i vari percorsi da Q a D per i vari flip-flop per il calcolo della frequenza massima F_{max} , facendo in modo di trovare il percorso con il tempo di percorrenza maggiore.

Visualizzazione dei percorsi su cui calcolare il tempo di percorrenza

(1)

$$t_1 = t_{\text{CK-Q}} + t_{\text{OR}} = 1 \text{ ns} + 1 \text{ ns} = 2 \text{ ns}$$

(2)

$$t_2 = t_{\text{CK-Q}} + t_{\text{OR}} + t_{\text{AND}} = 1 \text{ ns} + 1 \text{ ns} + 2 \text{ ns} = 4 \text{ ns}$$

(3)

$$t_3 = t_{\text{CK-Q}} + t_{\text{comp}} + t_{\text{AND}} = 1 \text{ ns} + 10 \text{ ns} + 2 \text{ ns} = 13 \text{ ns}$$

Si ottiene che il tempo di clock minimo è t_3 sommato a t_{SU} , essendo t_3 il percorso di costo massimo in termini di tempo tra quelli definiti.

$$T_{\text{CK}} \geq T_{\text{CK, min}} = t_3 + t_{\text{SU}} = 1 \text{ ns} + 10 \text{ ns} + 2 \text{ ns} + 0.8 \text{ ns} = 13.8 \text{ ns}$$

Dopo il calcolo di $T_{\text{CK, min}}$, si può definire la frequenza massima F_{max} .

$$F_{\text{CK}} = \frac{1}{T_{\text{CK}}} \leq F_{\text{max}} = \frac{1}{T_{\text{CK, min}}} = 72.5 \text{ MHz}$$

▼ Verifica delle violazioni di hold

Visualizzazione dei percorsi su cui verificare la condizione di hold

(1)

$$t_{\text{CK-Q}} = 1 \text{ ns} > t_H = 0.5 \text{ ns}$$

Dato che la condizione di hold è verificata, non ci possono essere violazioni.

2

Esercitazione 2 - Minimizzazione e mappe di Karnaugh

Esercizio 1 - Minimizzazione dell'espressione (3 variabili)

Obiettivo

[Mappa di Karnaugh](#)

[Minimizzazione dell'espressione](#)

Esercizio 2 - Minimizzazione dell'espressione (4 variabili)

Obiettivo

[Mappa di Karnaugh](#)

[Minimizzazione dell'espressione](#)

Esercizio 3 - Minimizzazione dell'espressione (con don't care)

Obiettivo

[Mappa di Karnaugh](#)

[Determinazione dell'espressione](#)

Esercizio 4 - Minimizzazione dell'espressione (4 variabili)

Obiettivo

[Mappa di Karnaugh](#)

[Minimizzazione dell'espressione](#)

Esercizio 5 - Mappa di Karnaugh senza alee

Obiettivo

[Mappa di Karnaugh](#)

[Eliminazione delle alee statiche](#)

Esercizio 6 - Analisi del comportamento di un circuito

Obiettivo

[Sistema di equazioni per lo stato futuro](#)

[Mappa di Karnaugh](#)

[Grafo di transizione degli stati](#)

Esercizio 7 - Design di circuito sequenziale da specifica

Obiettivo

[FSM](#)

[Mappe di Karnaugh](#)

[Circuito sequenziale](#)

Esercizio 8 - Semplificazione di circuiti multi-output

Obiettivo

[Mappe di Karnaugh delle funzioni](#)

[Prima forma del circuito multi-output](#)

[Semplificazione del circuito multi-output](#)

Esercizio 1 - Minimizzazione dell'espressione (3 variabili)

▼ Creatore originale: @Giacomo Dandolo

- @<utente> (<data>): <modifiche>

Obiettivo

Semplificare la seguente espressione logica utilizzando la mappa di Karnaugh.

$$D = \bar{A}B\bar{C} + AB\bar{C} + A\bar{B}\bar{C} + \bar{A}\bar{B}C + \bar{A}BC + ABC$$

▼ **Mappa di Karnaugh**

La mappa di Karnaugh è ottenuta attraverso questa procedura:

1. prendiamo un mintermine dell'espressione di D (per esempio, $\bar{A}B\bar{C}$);
2. cerchiamo la sua posizione nella mappa di Karnaugh, definendo i valori negati come 0 e i valori diretti come 1 (per esempio, 010);

		AB	00	01	11	10	
		C	0	0	1	1	1
			1	1	1	1	0

Mappa di Karnaugh ottenuta

3. inseriamo, all'interno della cella definita dalla posizione trovata, il valore 1;
4. ripetiamo, a partire dal [punto 1](#), per tutti i mintermini, fino a che non sono terminati all'interno dell'espressione;
5. riempiamo le celle vuote con degli zeri, in modo da ottenere la [mappa di Karnaugh finale](#).

▼ Minimizzazione dell'espressione

A partire dalla [mappa di Karnaugh](#), si può ottenere l'espressione iniziale di D minimizzata definendo le combinazioni di caselle che riducono il numero di letterali in un termine prodotto, rispettivamente chiamati D_1 , D_2 e D_3 , come nell'[immagine di riferimento](#).

		AB	00	01	11	10	
		C	0	0	1	1	1
			1	1	1	1	0

Mappa di Karnaugh con le combinazioni che minimizzano il numero di letterali

Cerchiamo ora di definire le espressioni delle combinazioni di caselle D_1 , D_2 e D_3 .

(1) Cerchiamo quali valori non cambiano, come coordinate, all'interno della combinazione di caselle:

- A cambia valore ($0 \leftrightarrow 1$), quindi non si prende;
- B non cambia valore ($1 \leftrightarrow 1$), ed essendo con valore 1, si prende la sua versione diretta;
- C cambia valore ($0 \leftrightarrow 1$), quindi non si prende.

$$D_1 = B$$

(2) Cerchiamo quali valori non cambiano, come coordinate, all'interno della combinazione di caselle:

- A non cambia valore ($0 \leftrightarrow 0$), ed essendo con valore 0, si prende la sua versione negata;
- B cambia valore ($0 \leftrightarrow 1$), quindi non si prende;
- C non cambia valore ($1 \leftrightarrow 1$), ed essendo con valore 1, si prende la sua versione diretta.

$$D_2 = \bar{A}C$$

(3) Cerchiamo quali valori non cambiano, come coordinate, all'interno della combinazione di caselle:

- A non cambia valore ($1 \leftrightarrow 1$), ed essendo con valore 1, si prende la sua versione diretta;
- B cambia valore ($0 \leftrightarrow 1$), quindi non si prende;
- C non cambia valore ($0 \leftrightarrow 0$), ed essendo con valore 0, si prende la sua versione negata.

$$D_3 = A\bar{C}$$

Eseguendo l'OR tra i mintermini, si ottiene la seguente espressione di D , che risulta minimizzata:

$$D = B + AC + A\bar{C}$$

Esercizio 2 - Minimizzazione dell'espressione (4 variabili)

▼ Creatore originale: @Giacomo Dandolo

- @<utente> (<data>): <modifiche>

Obiettivo

Semplificare la seguente espressione logica utilizzando la mappa di Karnaugh.

$$\begin{aligned} E = & \bar{A}\bar{B}\bar{C}\bar{D} + A\bar{B}\bar{C}\bar{D} + A\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}\bar{C}D + \bar{A}\bar{B}\bar{C}D + A\bar{B}\bar{C}D \\ & + A\bar{B}\bar{C}D + \bar{A}BCD + ABCD + \bar{A}BC\bar{D} + ABC\bar{D} \end{aligned}$$

▼ Mappa di Karnaugh

La mappa di Karnaugh è ottenuta attraverso questa procedura:

1. prendiamo un mintermine dell'espressione di E (per esempio, $\bar{A}B\bar{C}\bar{D}$);
2. cerchiamo la sua posizione nella mappa di Karnaugh, definendo i valori negati come 0 e i valori diretti come 1 (per esempio, 0100);
3. inseriamo, all'interno della cella definita dalla posizione trovata, il valore 1;
4. ripetiamo, a partire dal [punto 1](#), per tutti i mintermini, fino a che non sono terminati all'interno dell'espressione;

		AB	00	01	11	10
		CD	00	01	11	10
E		00	0	1	1	1
		01	1	1	1	1
E		11	0	1	1	0
		10	0	1	1	0

Mappa di Karnaugh ottenuta

5. riempiamo le celle vuote con degli zeri, in modo da ottenere la [mappa di Karnaugh finale](#).

▼ Minimizzazione dell'espressione

A partire dalla [mappa di Karnaugh](#), si può ottenere l'espressione iniziale di E minimizzata definendo le combinazioni di caselle che riducono il numero di letterali in un termine prodotto, rispettivamente chiamati E_1 , E_2 ed E_3 , come nell'[immagine di riferimento](#).

		AB	00	01	11	10
		CD	00	1	1	1
E		00	0			
		01	1	1	1	1
11		0	0	1	1	0
10		0	0	1	1	0

Mappa di Karnaugh con le combinazioni che minimizzano il numero di letterali

Cerchiamo ora di definire le espressioni delle combinazioni di caselle E_1 , E_2 ed E_3 .

(1) Cerchiamo quali valori non cambiano, come coordinate, all'interno della combinazione di caselle:

- A cambia valore ($0 \leftrightarrow 1$), quindi non si prende;
- B non cambia valore ($1 \leftrightarrow 1$), ed essendo con valore 1, si prende la sua versione diretta;
- C cambia valore ($0 \leftrightarrow 1$), quindi non si prende;
- D cambia valore ($0 \leftrightarrow 1$), quindi non si prende.

$$E_1 = B$$

(2) Cerchiamo quali valori non cambiano, come coordinate, all'interno della combinazione di caselle:

- A non cambia valore ($1 \leftrightarrow 1$), ed essendo con valore 1, si prende la sua versione diretta;
- B cambia valore ($0 \leftrightarrow 1$), quindi non si prende;
- C non cambia valore ($0 \leftrightarrow 0$), ed essendo con valore 0, si prende la sua versione negata;
- D cambia valore ($0 \leftrightarrow 1$), quindi non si prende.

$$E_2 = A\bar{C}$$

(3) Cerchiamo quali valori non cambiano, come coordinate, all'interno della combinazione di caselle:

- A cambia valore ($0 \leftrightarrow 1$), quindi non si prende;
- B cambia valore ($0 \leftrightarrow 1$), quindi non si prende;
- C non cambia valore ($0 \leftrightarrow 0$), ed essendo con valore 0, si prende la sua versione negata;
- D non cambia valore ($1 \leftrightarrow 1$), ed essendo con valore 1, si prende la sua versione diretta.

$$E_3 = \bar{C}D$$

Eseguendo l'OR tra i mintermini, si ottiene la seguente espressione di E , che risulta minimizzata:

$$E = B + A\bar{C} + \bar{C}D$$

Esercizio 3 - Minimizzazione dell'espressione (con don't care)

▼ Creatore originale: @Giacomo Dandolo

- @<utente> (<data>): <modifiche>

Obiettivo

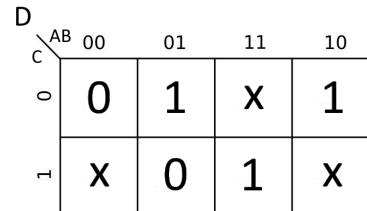
Determinare la funzione corrispondente a partire dalla tabella di verità data.

A	B	C
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

▼ Mappa di Karnaugh

La mappa di Karnaugh è ottenuta attraverso questa semplice procedura:

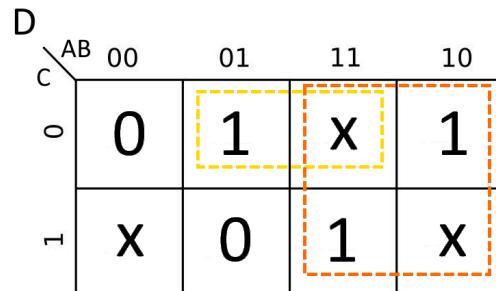
1. prendiamo una riga della tabella (per esempio, A=0, B=0, C=0);
2. cerchiamo la sua posizione nella mappa di Karnaugh (per esempio, 000);
3. inseriamo, all'interno della cella definita dalla posizione trovata, il valore corrispondente all'uscita della riga (per esempio, D=0);
4. ripetiamo, a partire dal [punto 1](#), per tutte le righe della tabella, fino al riempimento della mappa di Karnaugh.



Mappa di Karnaugh ottenuta

▼ Determinazione dell'espressione

A partire dalla [mappa di Karnaugh](#), si può ottenere l'espressione di D definendo le combinazioni di caselle che riducono il numero di letterali in un termine prodotto, rispettivamente chiamati D_1 e D_2 , come nell'[immagine di riferimento](#).



Mappa di Karnaugh con le combinazioni che minimizzano il numero di letterali

Cerchiamo ora di definire le espressioni delle combinazioni di caselle D_1 e D_2 .

(1) Cerchiamo quali valori non cambiano, come coordinate, all'interno della combinazione di caselle:

- A cambia valore ($0 \leftrightarrow 1$), quindi non si prende;
- B non cambia valore ($1 \leftrightarrow 1$), ed essendo con valore 1, si prende la sua versione diretta;
- C non cambia valore ($0 \leftrightarrow 0$), ed essendo con valore 0, si prende la sua versione negata.

$$D_1 = B\bar{C}$$

(2) Cerchiamo quali valori non cambiano, come coordinate, all'interno della combinazione di caselle:

- A non cambia valore ($1 \leftrightarrow 1$), ed essendo con valore 1, si prende la sua versione diretta;
- B cambia valore ($0 \leftrightarrow 1$), quindi non si prende;
- C cambia valore ($0 \leftrightarrow 1$), quindi non si prende.

$$D_2 = A$$

Eseguendo l'OR tra i mintermini, si ottiene la seguente espressione di D :

$$D = B\bar{C} + A$$

Esercizio 4 - Minimizzazione dell'espressione (4 variabili)

▼ Creatore originale: @Giacomo Dandolo

- @<utente> (<data>): <modifiche>

Obiettivo

Semplificare la seguente espressione logica utilizzando la mappa di Karnaugh.

$$\begin{aligned} E = & \bar{A}B\bar{C}\bar{D} + \bar{A}B\bar{C}D + AB\bar{C}D + A\bar{B}\bar{C}D \\ & + \bar{A}\bar{B}CD + \bar{A}BCD + ABCD + ABC\bar{D} \end{aligned}$$

▼ Mappa di Karnaugh

La mappa di Karnaugh è ottenuta attraverso questa procedura:

1. prendiamo un mintermine dell'espressione di E (per esempio, $\bar{A}B\bar{C}\bar{D}$);
2. cerchiamo la sua posizione nella mappa di Karnaugh, definendo i valori negativi come 0 e i valori diretti come 1 (per esempio, 0100);
3. inseriamo, all'interno della cella definita dalla posizione trovata, il valore 1;
4. ripetiamo, a partire dal punto 1, per tutti i mintermini, fino a che non sono terminati all'interno dell'espressione;

		AB	00	01	11	10
		CD	00	01	11	10
E		00	0	1	0	0
		01	0	1	1	1
E		11	1	1	1	0
		10	0	0	1	0

Mappa di Karnaugh ottenuta

5. riempiamo le celle vuote con degli zeri, in modo da ottenere la [mappa di Karnaugh finale](#).

▼ Minimizzazione dell'espressione

A partire dalla [mappa di Karnaugh](#), si può ottenere l'espressione iniziale di D minimizzata definendo le combinazioni di caselle che riducono il numero di letterali in un termine prodotto, rispettivamente chiamati E_1 , E_2 , E_3 e E_4 , come nell'[immagine di riferimento](#).

		AB	00	01	11	10
		CD	00	01	11	10
E	00	00	0	1	0	0
		01	0	1	1	1
11	11	1	1	1	0	
		10	0	0	1	0

Mappa di Karnaugh con le combinazioni che minimizzano il numero di letterali

Cerchiamo ora di definire le espressioni delle combinazioni di caselle E_1 , E_2 , E_3 e E_4 .

(1) Cerchiamo quali valori non cambiano, come coordinate, all'interno della combinazione di caselle:

- A non cambia valore ($0 \leftrightarrow 0$), ed essendo con valore 0, si prende la sua versione negata;
- B non cambia valore ($1 \leftrightarrow 1$), ed essendo con valore 1, si prende la sua versione diretta;
- C non cambia valore ($0 \leftrightarrow 0$), ed essendo con valore 0, si prende la sua versione negata;
- D cambia valore ($0 \leftrightarrow 1$), quindi non si prende.

$$E_1 = \bar{A}B\bar{C}$$

(2) Cerchiamo quali valori non cambiano, come coordinate, all'interno della combinazione di caselle:

- A non cambia valore ($1 \leftrightarrow 1$), ed essendo con valore 1, si prende la sua versione diretta;
- B cambia valore ($0 \leftrightarrow 1$), quindi non si prende;
- C non cambia valore ($0 \leftrightarrow 0$), ed essendo con valore 0, si prende la sua versione negata;
- D non cambia valore ($1 \leftrightarrow 1$), ed essendo con valore 1, si prende la sua versione diretta.

$$E_2 = A\bar{C}D$$

(3) Cerchiamo quali valori non cambiano, come coordinate, all'interno della combinazione di caselle:

- A non cambia valore ($1 \leftrightarrow 1$), ed essendo con valore 1, si prende la sua versione diretta;
- B non cambia valore ($1 \leftrightarrow 1$), ed essendo con valore 1, si prende la sua versione diretta;
- C non cambia valore ($1 \leftrightarrow 1$), ed essendo con valore 1, si prende la sua versione diretta;
- D cambia valore ($0 \leftrightarrow 1$), quindi non si prende.

$$E_3 = ABC$$

(4) Cerchiamo quali valori non cambiano, come coordinate, all'interno della combinazione di caselle:

- A non cambia valore ($0 \leftrightarrow 0$), ed essendo con valore 0, si prende la sua versione negata;
- B cambia valore ($1 \leftrightarrow 1$), quindi non si prende;
- C non cambia valore ($1 \leftrightarrow 1$), ed essendo con valore 1, si prende la sua versione diretta;

- D non cambia valore ($1 \leftrightarrow 1$), ed essendo con valore 1, si prende la sua versione diretta.

$$E_4 = \bar{A}CD$$

Eseguendo l'OR tra i mintermini, si ottiene la seguente espressione di E , che risulta minimizzata:

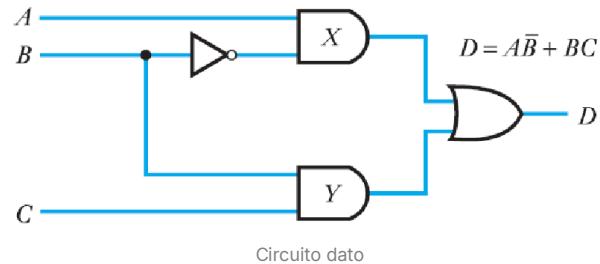
$$E = \bar{A}B\bar{C} + A\bar{C}D + ABC + \bar{A}CD$$

Esercizio 5 - Mappa di Karnaugh senza aleee

▼ Creatore originale: @Giacomo Dandolo

Obiettivo

Ricavare la k-map per il [circuito dato](#) e rifinire la copertura per escludere la possibilità di aleee statiche.



▼ Mappa di Karnaugh

Definiamo innanzitutto la mappa di Karnaugh che descrive la funzione logica, come negli esercizi precedenti.

$$D = A\bar{B} + BC$$

		AB	00	01	11	10
		C	0	0	0	1
D	AB	00	0	0	0	1
		01	0	1	1	1

Mappa di Karnaugh ottenuta

▼ Eliminazione delle aleee statiche

Per eliminare le [aleee statiche](#), è necessario eliminare possibili instabilità durante la transizione da uno stato ad un altro.

In questo caso, per esempio, si ha che la transizione $111 \rightarrow 101$ contiene la possibilità di un'alea, poiché la transizione contiene due 1 che non fanno parte dello stesso implicante.

Per rimuovere questa possibilità, si crea un nuovo implicante non necessario, come nella [mappa di Karnaugh mostrata](#).

D	AB	00	01	11	10
C	0	0	0	0	1
C	1	0	1	1	1

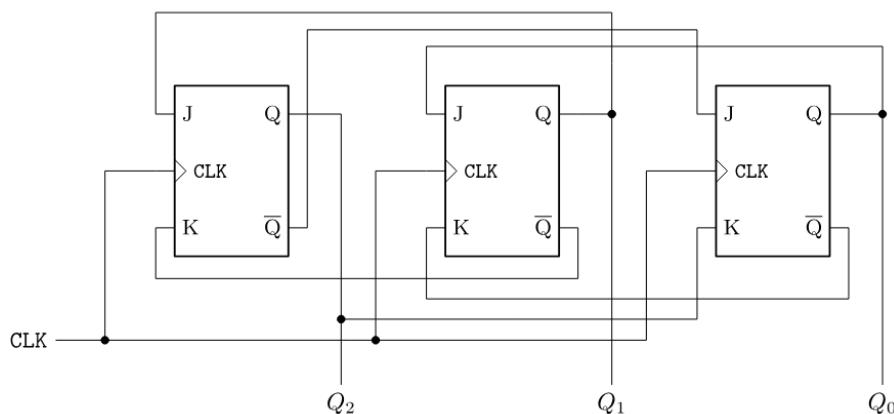
Mappa di Karnaugh con eliminazione delle alee statiche

Esercizio 6 - Analisi del comportamento di un circuito

▼ Creatore originale: @Giacomo Dandolo

Obiettivo

Dato un circuito sequenziale, determinare il grafo di transizione degli stati a partire dalle corrispondenti mappe di Karnaugh.



Circuito sequenziale in analisi

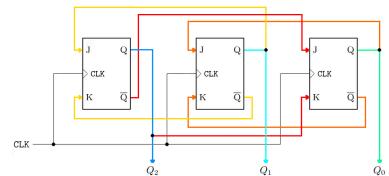
▼ Sistema di equazioni per lo stato futuro

Definiamo la formula che descrive il comportamento di ogni FF-JK.

$$Q^+ = J\bar{Q} + QK$$

Attraverso questa equazione, possiamo definire il sistema di equazioni per lo stato futuro, dove è presente un'equazione per ogni FF-JK.

$$\begin{cases} Q_2^+ = Q_1 \bar{Q}_2 + Q_2 Q_1 \\ Q_1^+ = Q_0 \bar{Q}_1 + Q_1 Q_0 \\ Q_0^+ = \bar{Q}_2 \bar{Q}_0 + Q_0 \bar{Q}_2 \end{cases}$$



Visualizzazione dei valori di entrata per ogni FF-JK

▼ Mappa di Karnaugh

Definiamo la mappa di Karnaugh a partire dal sistema di equazioni per lo stato futuro ottenute.

$$\begin{cases} Q_2^+ = Q_1 \bar{Q}_2 + Q_2 Q_1 \\ Q_1^+ = Q_0 \bar{Q}_1 + Q_1 Q_0 \\ Q_0^+ = \bar{Q}_2 \bar{Q}_0 + Q_0 \bar{Q}_2 \end{cases}$$

Per farlo, si devono seguire questi passi:

1. prendiamo una posizione all'interno della mappa di Karnaugh (per esempio, 000);
2. valutiamo ogni equazione all'interno del sistema di equazioni per lo stato futuro, in modo da trovare il valore $Q_2^+ Q_1^+ Q_0^+$ ed inserirlo all'interno della rispettiva cella (per esempio, 001);

		$Q_2 Q_1$	00	01	11	10	
		Q_0	0	001	101	100	000
		1	1	011	111	110	010

Mappa di Karnaugh ottenuta dal sistema delle equazioni di stato futuro

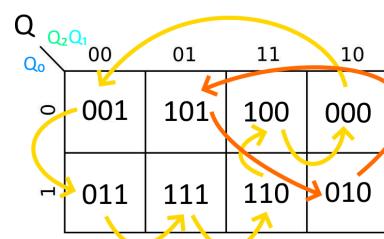
$$\begin{cases} Q_2^+ = 0 \cdot 1 + 0 \cdot 0 = 0 \\ Q_1^+ = 0 \cdot 1 + 0 \cdot 0 = 0 \\ Q_0^+ = 1 \cdot 1 + 0 \cdot 1 = 1 \end{cases}$$

3. ripetere questo procedimento fino al riempimento di ogni cella, in modo da ottenere [la mappa di Karnaugh finale](#).

▼ Grafo di transizione degli stati

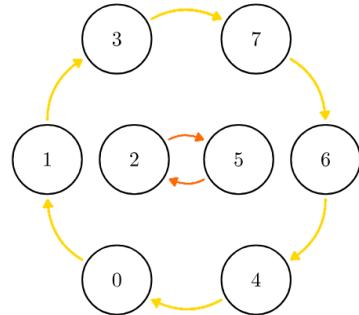
Il grafo di transizione si ottiene attraverso questi passi:

1. prendiamo una posizione all'interno della mappa di Karnaugh (per esempio, 000);
2. disegniamo uno stato all'interno del grafo di transizione, con un valore pari a quello della posizione;
3. andiamo alla cella definita dal valore all'interno della cella (per esempio, 001), e disegniamola come il valore precedente;
4. colleghiamo lo stato precedente a quello attuale con un arco direzionato;



Visualizzazione del procedimento di passaggio dei vari stati

5. se si ottiene un nuovo ciclo al collegamento e sono presenti stati nella mappa di Karnaugh che non sono stati inseriti, si ricomincia dal punto 1;
6. ripetiamo lo stesso procedimento fino al termine delle celle, ottenendo il [grafo di transizione degli stati](#).



Grafo di transizione degli stati ottenuto

Esercizio 7 - Design di circuito sequenziale da specifica

▼ Creatore originale: @Giacomo Dandolo

Obiettivo

Modellare un riconoscitore di sequenza adatto a riconoscere una qualsiasi sequenza che termini in 101_2 .

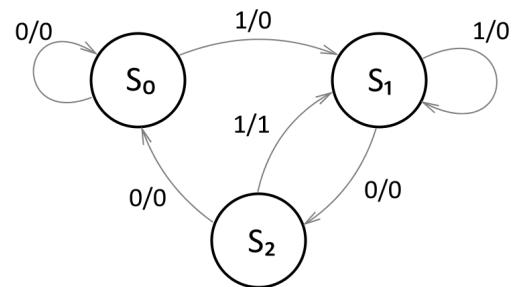
▼ FSM

Considerando che il rilevamento della sequenza richiede una determinata serie di valori, è necessario che il valore di uscita della FSM dipenda sia dagli input, sia dallo stato presente.

Per questo motivo, la FSM richiesta è definita attraverso l'[architettura di Mealy](#).

Definiamo i tre stati necessari per modelarla:

- S_0 : identificatore binario 00;
- S_1 : identificatore binario 01;
- S_2 : identificatore binario 10.



FSM di Mealy ottenuta

Definendo le coppie X/Z come etichette sugli archi che definiscono rispettivamente Input e Output, si ottiene la FSM in [figura](#).

▼ Mappe di Karnaugh

A partire dalla [FSM ottenuta](#), si definiscono le mappe di Karnaugh per lo stato futuro dei FF (Q_1 e Q_2) e dell'uscita (Z).

Iniziamo definendo la mappa di Karnaugh della FSM, in cui ogni cella contiene il valore di tre cifre $Q_1^+ Q_2^+ Z$:

- Q_1^+ è la prima cifra dello stato futuro rispetto al valore di $Q_1 Q_2$;
- Q_2^+ è la seconda cifra dello stato futuro rispetto al valore di $Q_1 Q_2$;
- Z è l'uscita dello stato in funzione dell'input X .

$Q_1^+ Q_2^+ Z$	(S_0)	(S_1)	(S_2)
x	000	100	X
0	000	100	X

$Q_1^+ Q_2^+ Z$	(S_0)	(S_1)	(S_2)
x	000	100	X
1	010	010	X

Mappa di Karnaugh della FSM

Definiamo ora le mappe di Karnaugh per ognuno dei valori di uscita Q_1^+ , Q_2^+ e Z , in modo da riuscire ad identificare la loro correlazione con i valori di input Q_1 , Q_2 e X .

Si ottiene questa tabella prendendo, per ogni cella, il valore correlato a Q_1^+ nella [mappa di Karnaugh della FSM](#).

Definendo i mintermini, otteniamo l'equazione di Q_1^+ .

$$Q_1^+ = D_{Q_1} = \bar{X}Q_2$$

Q_1^+	$Q_1 Q_2$	00	01	11	10
0	x	0	1	x	0
1	x	0	0	x	0

Mappa di Karnaugh che definisce Q_1^+

Si ottiene questa tabella prendendo, per ogni cella, il valore correlato a Q_2^+ nella [mappa di Karnaugh della FSM](#).

Definendo i mintermini, otteniamo l'equazione di Q_2^+ .

$$Q_2^+ = D_{Q_2} = X$$

Q_2^+	$Q_1 Q_2$	00	01	11	10
0	x	0	0	x	0
1	x	1	1	x	1

Mappa di Karnaugh che definisce Q_2^+

Si ottiene questa tabella prendendo, per ogni cella, il valore correlato a Z nella [mappa di Karnaugh della FSM](#).

Definendo i mintermini, otteniamo l'equazione di Z .

$$Z = XQ_1$$

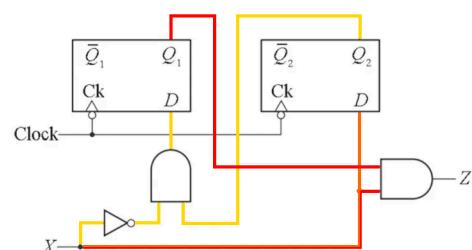
Z	$Q_1 Q_2$	00	01	11	10
0	x	0	0	x	0
1	x	0	0	x	1

Mappa di Karnaugh che definisce Z

▼ Circuito sequenziale

Utilizzando dei Flip-Flop D, e avendo le espressioni di D_{Q_1} , D_{Q_2} e Z , si possono definire i componenti da utilizzare per costruire il circuito sequenziale, ovvero:

- due Flip-Flop D;
- due porte AND;
- una porta NOT.



Circuito sequenziale ottenuto

$$D_{Q_1} = \bar{X}Q_2$$

$$D_{Q_2} = X$$

$$Z = XQ_1$$

Esercizio 8 - Semplificazione di circuiti multi-output

▼ Creatore originale: @Giacomo Dandolo

Obiettivo

Data la seguente descrizione per un circuito multi-output a 4 ingressi e 3 uscite, realizzare un circuito in grado di soddisfare la specifica.

$$\begin{cases} f_1 = \sum m(2, 3, 5, 7, 8, 9, 10, 11, 13, 15) \\ f_2 = \sum m(2, 3, 5, 6, 7, 10, 11, 14, 15) \\ f_3 = \sum m(6, 7, 8, 9, 13, 14, 15) \end{cases}$$

▼ Mappe di Karnaugh delle funzioni

Partiamo dalla funzione f_1 . Per leggere questi valori e trasformarli in una mappa di Karnaugh, bisogna ricordare che i valori presenti nelle parentesi indicano in quali celle è presente il valore 1 in una funzione, in questo caso a 4 bit, e che ogni cella è rappresentata da un valore binario $x_1x_2x_3x_4$.

Per esempio, $2_{10} = 0010_2$, quindi nella cella corrispondente all'incrocio tra $x_3x_4 = 10_2$ e $x_1x_2 = 00_2$ si inserisce 1. Prendendo un secondo esempio, $9_{10} = 1001_2$, nella cella corrispondente all'incrocio tra $x_3x_4 = 01_2$ e $x_1x_2 = 10_2$ si inserisce 1.

Si ripete questo procedimento per ogni valore presente tra le parentesi della funzione, e successivamente per ogni funzione, ottenendo le espressioni di f_1 , f_2 e f_3 e la relativa mappa di Karnaugh.

$$\begin{aligned} f_1 &= x_1\bar{x}_2 + x_2x_4 + \bar{x}_2x_3 \\ &= \rho_{11} + \rho_{12} + \rho_{13} \end{aligned}$$

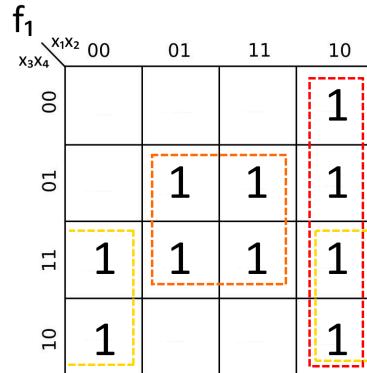


Tabella di Karnaugh (Prima funzione)

$$f_2 = x_3 + \bar{x}_1 x_2 x_4 \\ = \rho_{21} + \rho_{22}$$

		00	01	11	10
		00			
		01			
		11	1	1	1
		10	1	1	1

Tabella di Karnaugh (Seconda funzione)

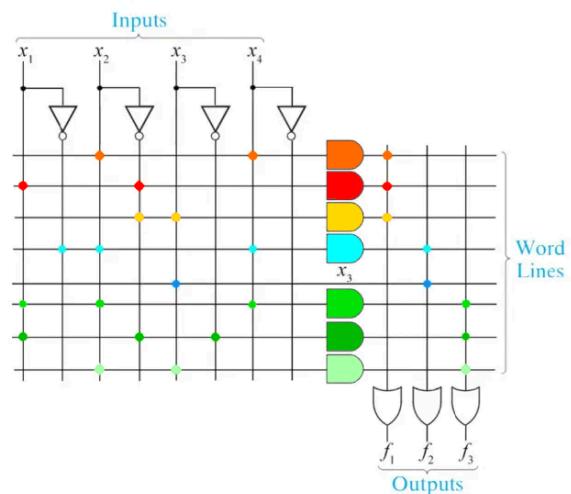
$$f_3 = x_1 \bar{x}_2 \bar{x}_3 + x_1 x_2 x_4 + x_2 x_3 \\ = \rho_{31} + \rho_{32} + \rho_{33}$$

		00	01	11	10
		00			
		01			
		11	1	1	
		10	1	1	

Tabella di Karnaugh (Terza funzione)

▼ Prima forma del circuito multi-output

Definendo gli input nella loro forma negata e nella loro forma diretta, e definendo anche una porta per ogni parte delle funzioni ($\rho_{11}, \rho_{12}, \rho_{13}, \dots$), si ottiene la prima forma del circuito multi-output, la quale non è semplificata in alcun modo, e risulta, di fatto, non un circuito ottimizzato dal punto di vista del costo e del numero di porte.



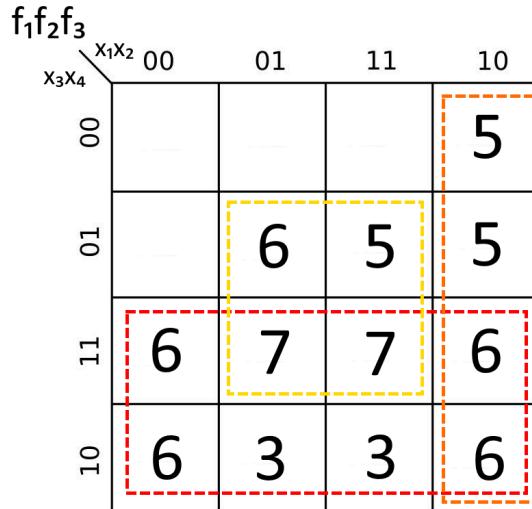
Prima forma del circuito multi-output

▼ Semplificazione del circuito multi-output

Per semplificare il circuito, è necessario definire una mappa di Karnaugh che definisce la relazione tra le varie funzioni. Per realizzarla, è necessario accostare i valori binari, per ogni cella, di ognuna delle mappe di Karnaugh, per poi trasformarlo in valore decimale, in modo da semplificare la tabella di Karnaugh finale e identificare subito quali celle sono necessarie per la semplificazione.

Per esempio, la cella definita da $x_1x_2 = 00_2$ e la cella definita da $x_3x_4 = 11_2$ avrà, al suo interno, il valore 110_2 , che scriveremo con il suo valore decimale, 6_{10} .

Tutto ciò si ripete per ogni cella, fino a formare la mappa di Karnaugh con i relativi raggruppamenti ρ_1 , ρ_2 e ρ_3 .



Mappa di Karnaugh (tutte le funzioni)

Per semplificare l'espressione iniziale, si devono scrivere le espressioni di ρ_1 , ρ_2 e ρ_3 , per poi cercare di trasformare le espressioni in funzione delle porte definite. Eseguiamo i calcoli per ogni espressione.

- ρ_1 :

$$\rho_1 = x_3 = \rho_{13} + \rho_{33} = \sum m(2, 3, 10, 11) + \sum m(6, 7, 14, 15)$$

- ρ_2 :

$$\rho_2 = x_1\bar{x}_2 = \rho_{31} + \rho_{13}/\rho_2 = \sum m(8, 9) + \sum m(2, 3)$$

- ρ_3 .

$$\rho_3 = x_2x_4 = \rho_{22} + \rho_{32} = \sum m(5, 7) + \sum m(13, 15)$$

Dopo averle definite, riprendendo le definizioni iniziali della [prima funzione](#), della [seconda funzione](#) e della [terza funzione](#), si può ridefinire ognuna delle espressioni in funzione delle porte logiche utilizzate per definire ρ_1 , ρ_2 e ρ_3 .

$$\begin{aligned} f_1 &= \rho_{13} + \rho_{31} + \rho_{22} + \rho_{32} \\ &= \bar{x}_2x_3 + x_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2x_4 + x_1x_2x_4 \end{aligned}$$

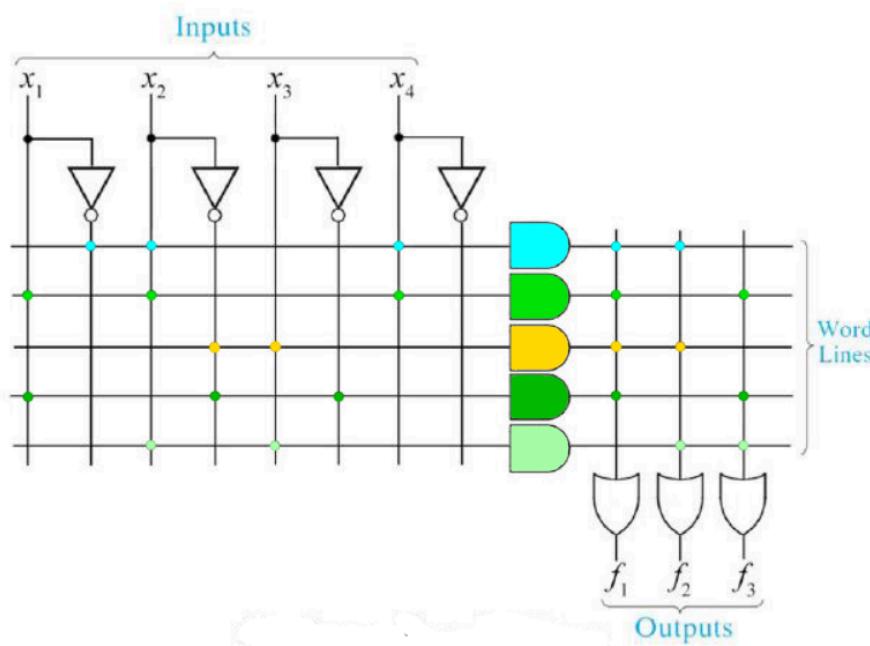
$$\begin{aligned} f_2 &= \rho_{22} + \rho_{33} + \rho_{13} \\ &= \bar{x}_1x_2x_4 + x_2x_3 + \bar{x}_2x_3 \end{aligned}$$

$$\begin{aligned} f_3 &= \rho_{33} + \rho_{32} + \rho_{31} \\ &= x_2x_3 + x_1x_2x_4 + x_1\bar{x}_2\bar{x}_3 \end{aligned}$$

$f_1 f_2 f_3$	$x_1 x_2$	00	01	11	10
$x_3 x_4$	00				5
00	01			5	5
01	11	6	7	7	6
11	10	6	3	3	6

Mappa di Karnaugh finale

Allo stesso modo della prima forma del circuito multi-output, si forma la seconda forma del circuito multi-output, il quale risulta semplificato rispetto al primo.



Seconda forma del circuito multi-output

3

Esercitazione 3 - Tempi di salita, di discesa, di propagazione e frequenza massima

Esercizio 1 - Porte Logiche CMOS

Obiettivo 1 - Completamento della schematica

Costruzione della PDN

Obiettivo 2.1 - Tempo di salita e di discesa

Impostazione

Calcolo della resistenza equivalente minima e massima

Calcolo del tempo di salita

Calcolo del tempo di discesa

Nota a posteriori

Obiettivo 2.2 - Tempi di propagazione

Impostazione

Spiegazione del caso L→H

Caso L→H (E)

Caso L→H (I)

Spiegazione del caso H→L

Caso H→L (E)

Caso H→L (I)

Ritardo complessivo (OUT)

Nota a posteriori

Esercizio 2 - Porte Logiche CMOS

Obiettivo 1 - Implementazione di funzione logica con CMOS

Utilizzo delle leggi di De Morgan

Utilizzo della doppia negazione

Obiettivo 2 - Tempi di salita e di discesa

Calcolo del tempo di salita

Tempo di discesa

Esercizio 3 - Circuito sequenziale CMOS

Descrizione dei parametri del circuito in esame

Obiettivo 1 - Ritardi di propagazione CMOS

Porta NAND

Porta NOR

Obiettivo 2 - Frequenza massima del circuito

Impostazione

Propagazione di un segnale inizialmente alto

Propagazione di un segnale inizialmente basso

Frequenza massima

Obiettivo 3 - Valore massimo del tempo di hold

Determinazione del percorso più breve

Calcolo del tempo di hold

Obiettivo 4 - Potenza dinamica e statica

Potenza statica (NAND)

Potenza statica (NOR)

Nota sulla potenza statica

Potenza dinamica

Esercizio 1 - Porte Logiche CMOS

▼ Creatore originale: @Francesco Ambrosino

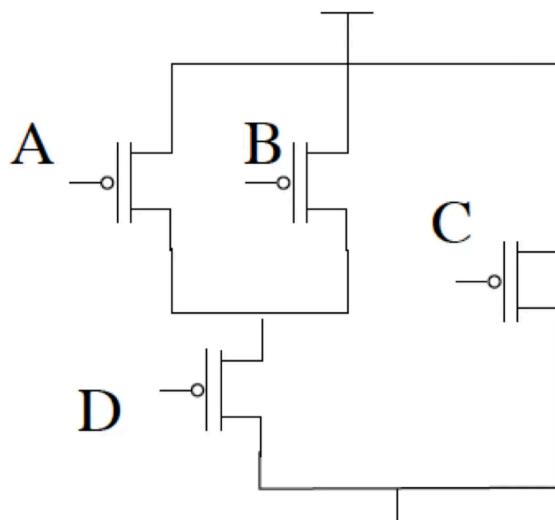
- @Giacomo Dandolo (11/04/2025): Aggiunte descrizioni per passaggi dell'obiettivo 1, aggiunti valori forniti dal testo ed eseguiti i calcoli per l'obiettivo 2.1.

Obiettivo 1 - Completamento della schematica

Completare lo schematica della porta logica.

Dato un pull-up network (pull-down network), il corrispettivo pull-down network (pull-up network) si disegna per antitesi: data una serie di CMOS si ha un parallelo, e viceversa.

Supponiamo di avere la seguente PUN, che è tale perché sono presenti i pMOS. Vogliamo costruirne la PDN.



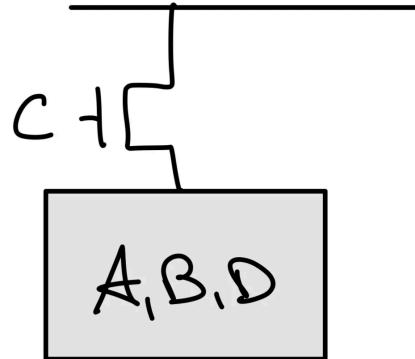
Visualizzazione della PUN in esame

▼ Costruzione della PDN

Costruiamo la PDN, dall'esterno verso l'interno.

1. Evidenziamo C , mettendola in serie al resto del sistema, poiché il parallelo tra C e gli altri componenti è descrivibile da una serie tra gli stessi.

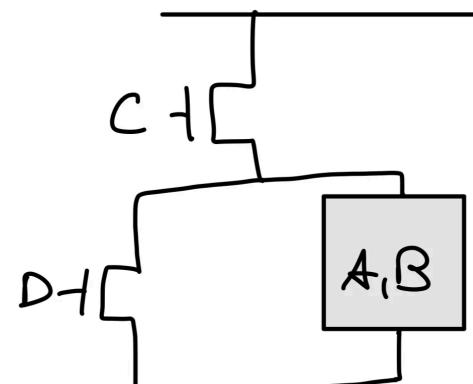
$$C \parallel \{A, B, D\} \longrightarrow C + \{A, B, D\}$$



Prima operazione

2. Evidenziamo D , mettendola in parallelo agli altri componenti, poiché la serie tra D e gli altri componenti è descrivibile da un parallelo degli stessi.

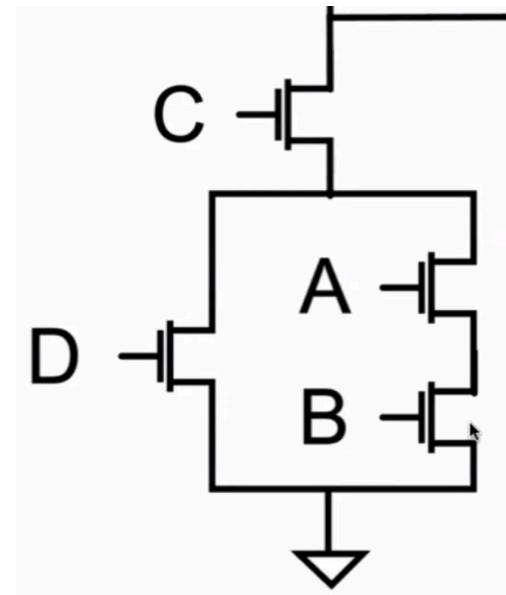
$$D + \{A, B\} \longrightarrow D \parallel \{A, B\}$$



Seconda operazione

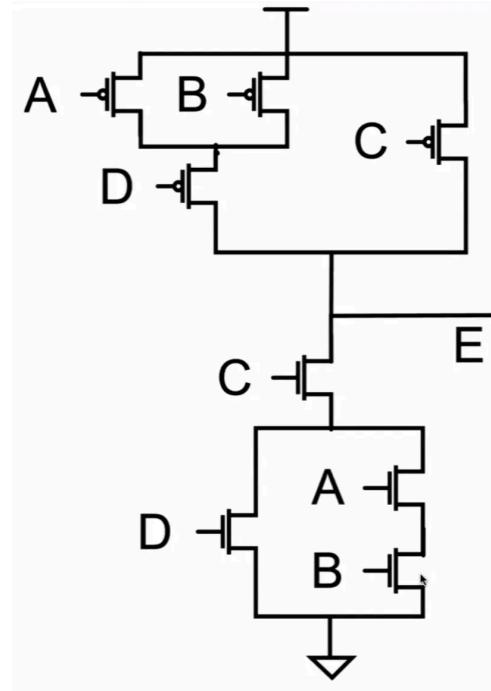
3. Evidenziamo A e B , mettendoli in serie tra loro, poiché il parallelo tra A e B è descrivibile da una serie degli stessi.

$$A \parallel B \longrightarrow A + B$$



Terza operazione

4. Alla fine del procedimento, si ottiene che il circuito finale è come nella figura a lato.



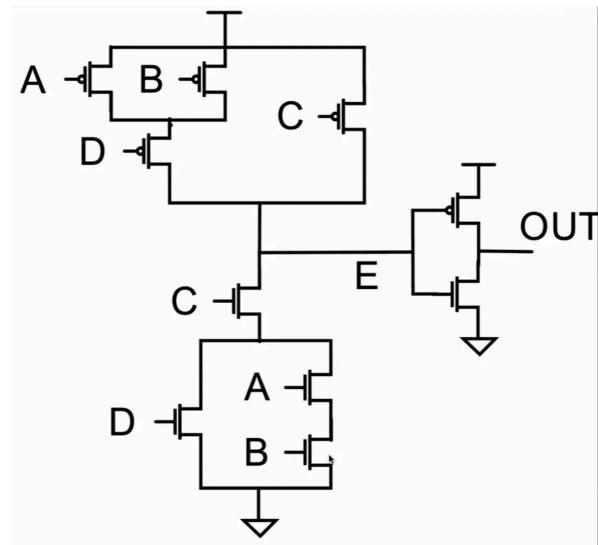
Risultato finale

Obiettivo 2.1 - Tempo di salita e di discesa

Dato lo schema precedentemente ottenuto, determinare il tempo di salita (t_{rise}) e il tempo di discesa (t_{fall}) minimi e massimi al nodo E, nota R_{ON} e C_{gate} dei MOS.

$$R_{ON} = 10 \text{ k}\Omega$$

$$C_{gate} = 5 \text{ fF}$$



Schema ottenuto alla fine dell'obiettivo 1



Richiami teorici

Calcolo del tempo di salita (t_{rise}) e del tempo di discesa (t_{fall})

Per il calcolo di t_{rise} si guarda la PUN, mentre per il calcolo di t_{fall} si guarda la PDN, visto che la carica arriva (rise) dalla Pull Up Network e si disperde (caduta) dalla Pull Down Network.

Tempi di salita e di discesa

Calcolo del tempo di transizione ($t_{transizione}$)

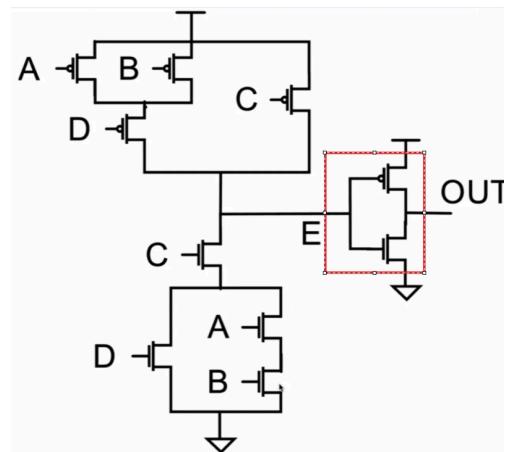
$$t_{transizione} = \tau \cdot \ln(9) = \tau \cdot 2.2 \quad \tau = R_{eq} \cdot C_{carico}$$

Per il calcolo di $t_{transizione, min}$ si usano le $R_{eq, min}$, mentre per il calcolo di $t_{transizione, max}$ si usano le $R_{eq, max}$.

La C_{carico} è la stessa per entrambi i casi.

▼ Impostazione

Considereremo $C_{carico} = 2 \cdot C_{gate}$, poiché il carico della mia uscita è rappresentato dall'inverter il quale è formato da due MOS considerabili in parallelo e, di conseguenza, bisogna considerare il parallelo delle C_{gate} dei MOS che costituiscono l'inverter.



Sia per il tempo di salita, sia per il tempo di discesa, si possono descrivere i rispettivi tempi massimi e minimi come:

$$t_{transizione, min} = 2.2 \cdot R_{eq, min} \cdot C_{carico} = 2.2 \cdot R_{eq, min} \cdot 2C_{gate}$$

$$t_{\text{transizione, max}} = 2.2 \cdot R_{\text{eq, max}} \cdot C_{\text{carico}} = 2.2 \cdot R_{\text{eq, max}} \cdot 2C_{\text{gate}}$$

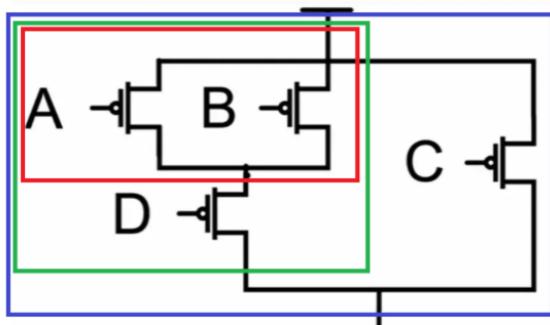
▼ Calcolo della resistenza equivalente minima e massima

Per il calcolo della $R_{\text{eq,min}}$ e $R_{\text{eq,max}}$, in entrambi i casi di tempo di salita e di discesa, si può procedere in due modi:

1. si calcolano tutte le combinazioni di R_{eq} possibili, prendendo quelle che ne minimizzano (o massimizzano) il valore;
2. si osserva che i paralleli diminuiscono il valore della R_{eq} , le serie aumentano il valore della R_{eq} . Sapendo questo, nel calcolo della $R_{\text{eq, min}}$ basta tenere in considerazione più paralleli possibili e solo le serie "obbligate". Nel calcolo della $R_{\text{eq, max}}$ dobbiamo tenere in considerazione più serie possibili, evitando i paralleli.

▼ Calcolo del tempo di salita

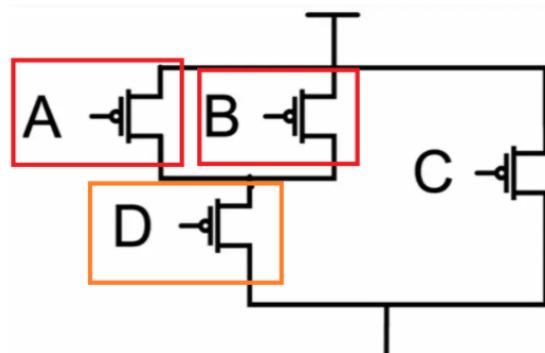
Sviluppiamo i calcoli della $R_{\text{eq, min}}^{\text{PUN}}$ e $R_{\text{eq, max}}^{\text{PUN}}$ in relazione al tempo di salita, osservando la PUN.



Calcolo della resistenza equivalente minima:
in

rosso $A \parallel B$, in verde $(A \parallel B) + D$, in blu
 $((A \parallel B) + D) \parallel C$.

N.B. stiamo considerando la PUN



Calcolo della resistenza equivalente
massima:
in

rosso A oppure B (essendo uguali in
valore), in arancione D e tutti insieme $(A +$
 $D)$ oppure $(B + D)$.

N.B. stiamo considerando la PUN

$$R_{\text{eq, min}}^{\text{PUN}} = ((R_A \parallel R_B) + R_D) \parallel R_C = \frac{3}{5} R_{\text{ON}}$$

$$R_{\text{eq, max}}^{\text{PUN}} = R_A + R_D = 2R_{\text{ON}}$$

Si noti come la serie utilizzata tra $\{A, B\}$ e D è obbligata, perché altrimenti non sarebbe presente collegamento tra E e V_∞ , e quindi deve fare parte del calcolo di $R_{\text{eq, min}}^{\text{PUN}}$.

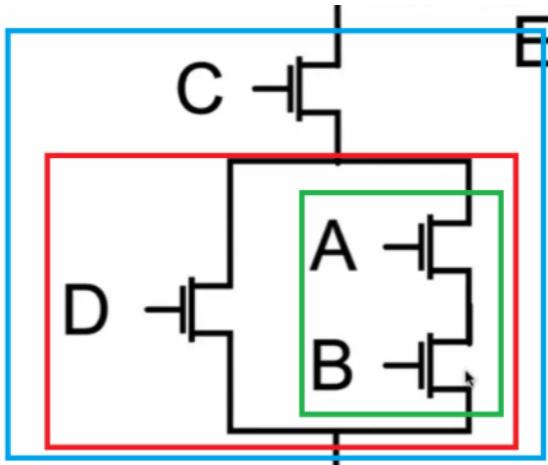
Possiamo, quindi, calcolare $t_{\text{rise,min}}$ e $t_{\text{rise,max}}$.

$$\begin{aligned}\tau_{\text{rise, min}} &= R_{\text{eq, min}}^{\text{PUN}} \cdot 2C_{\text{gate}} & t_{\text{rise,min}} &= 2.2\tau_{\text{rise, min}} = 0.13 \text{ ns} \\ &= \frac{3}{5}R_{\text{ON}} \cdot 2C_{\text{gate}}\end{aligned}$$

$$\begin{aligned}\tau_{\text{rise, max}} &= R_{\text{eq, max}}^{\text{PUN}} \cdot 2C_{\text{gate}} & t_{\text{rise,max}} &= 2.2\tau_{\text{rise, max}} = 0.44 \text{ ns} \\ &= 2R_{\text{ON}} \cdot 2C_{\text{gate}}\end{aligned}$$

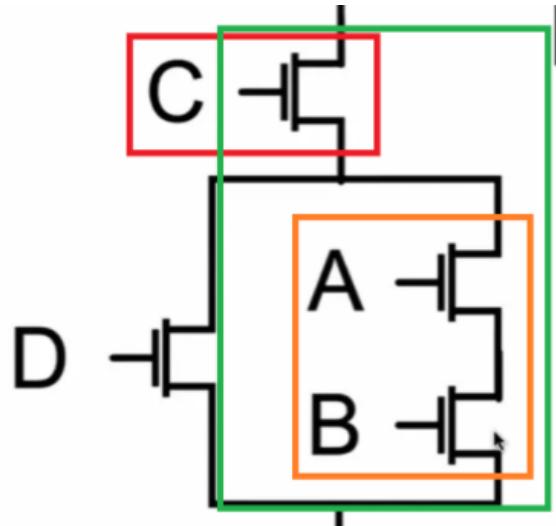
▼ Calcolo del tempo di discesa

Sviluppiamo i calcoli della $R_{\text{eq, min}}^{\text{PDN}}$ e $R_{\text{eq, max}}^{\text{PDN}}$ in relazione al tempo di discesa, osservando la PDN.



Calcolo della resistenza equivalente minima:
in verde $A + B$, in rosso $(A + B)\parallel D$ e in blu
 $((A + B)\parallel D) + C$.

N.B. stiamo considerando la PDN



Calcolo della resistenza equivalente massima:
In rosso C , in arancione $A + B$ e in verde
 $A + B + C$;

N.B. stiamo considerando la PDN

$$R_{\text{eq, min}}^{\text{PDN}} = ((R_A + R_B)\parallel R_D) + R_C = \frac{5}{3}R_{\text{ON}}$$

$$R_{\text{eq, max}}^{\text{PDN}} = R_A + R_B + R_C = 3R_{\text{ON}}$$

Possiamo, quindi, calcolare $t_{\text{fall,min}}$ e $t_{\text{fall,max}}$.

$$\begin{aligned} \tau_{\text{fall, min}} &= R_{\text{eq, min}}^{\text{PDN}} \cdot 2C_{\text{gate}} & t_{\text{fall,min}} &= 2.2\tau_{\text{fall, min}} = 0.37 \text{ ns} \\ &= \frac{5}{3}R_{\text{ON}} \cdot 2C_{\text{gate}} \end{aligned}$$

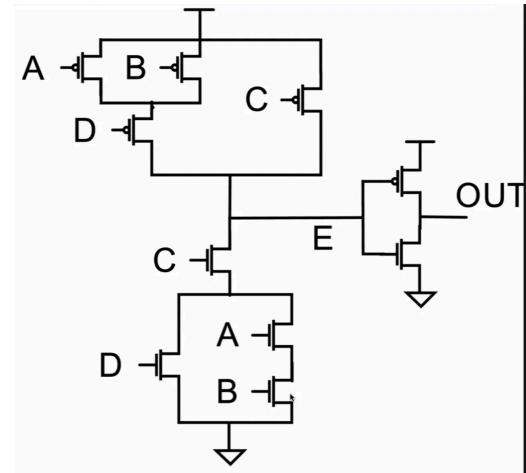
$$\begin{aligned} \tau_{\text{fall, max}} &= R_{\text{eq, max}}^{\text{PDN}} \cdot 2C_{\text{gate}} & t_{\text{fall,max}} &= 2.2\tau_{\text{fall, max}} = 0.66 \text{ ns} \\ &= 3R_{\text{ON}} \cdot 2C_{\text{gate}} \end{aligned}$$

▼ Nota a posteriori

In questo caso, $R_{\text{eq},\min}^{\text{fall}}$ risulta uguale a $R_{\text{eq},\min}^{\text{rise}}$, ma con il coefficiente invertito. Questa non è una regola: la spiegazione è che in questo caso entrambe le R_{eq} considerano tutto il circuito, quindi al più può essere vista come una conferma di simmetria tra PUN e PDN.

Obiettivo 2.2 - Tempi di propagazione

Dato lo schema precedentemente ottenuto, determinare $t_{\text{propagazione, min}}$ e $t_{\text{propagazione, max}}$, noti V_{IL} , V_{IH} , V_{DD} , R_{ON} , $C_{\text{carico,E}}$, $C_{\text{carico,I}}$, dove $V_{IL} < V_{IH}$.



Schema ottenuto nell'obiettivo 1

▼ Impostazione

Per calcolare il tempo di propagazione $t_{\text{propagazione}}$, sia esso minimo o massimo, dobbiamo considerare quattro situazioni:

1. commutazioni in E;

 - a. $L \rightarrow H$;
 - b. $H \rightarrow L$.

2. commutazioni in OUT.

 - a. $L \rightarrow H$;
 - b. $H \rightarrow L$.

Per gestire questi casi, è necessario ricordarsi la seguente formula:

$$V_c(t) = V_\infty + (V_0 - V_\infty)e^{-\frac{t}{\tau}}$$

▼ Spiegazione del caso L→H

Per il caso in cui è presente la transizione LOW (L) → HIGH (H), si deve:

- guardare la PUN, ricordando che si vuole raggiungere HIGH;
- ricordando che si parte da LOW, equivalente a GND, si ha:

$$V_0 = V_{\text{GND}}$$

$$V_\infty = V_{\text{DD}}$$

- sviluppando l'equazione generale, si ottiene:

$$t = \tau \cdot \ln \left(\frac{V_{\text{DD}}}{V_{\text{DD}} - V_c(t)} \right)$$

▼ Caso L→H (E)

Utilizzeremo i valori $R_{\text{eq, min}}$ ed $R_{\text{eq, max}}$ del tempo di salita calcolati all'obiettivo precedente, che d'ora in poi alleggeriremo omettendo l'apice PUN.

$$R_{\text{eq, min}} = \frac{3}{5} R_{\text{ON}}$$

$$R_{\text{eq, max}} = 2R_{\text{ON}}$$

Per il calcolo di $t_{\text{L} \rightarrow \text{H}, \text{min}}$ usiamo $V_c(t) = V_{\text{IL}}$ per due motivi:

- motivo fisico: essendo in salita (L→H), per il calcolo del tempo minimo usiamo la soglia più bassa, poiché è quella che viene attraversata prima tra le due;
- motivo matematico: volendo trovare il tempo minimo, usiamo il valore più piccolo poiché, essendo col segno negativo al denominatore, è quello che minimizza il risultato della frazione. Detto in altri termini, se usassimo V_{IH} otterremmo un valore maggiore rispetto ad usare V_{IL} .

$$t_{\text{L} \rightarrow \text{H}, \text{min}} = R_{\text{eq, min}} \cdot C_{\text{carico, E}} \cdot \ln \left(\frac{V_{\text{DD}}}{V_{\text{DD}} - V_{\text{IL}}} \right) = 0.04 \text{ ns}$$

Per il calcolo di $t_{\text{L} \rightarrow \text{H}, \text{max}}$ usiamo $V_c(t) = V_{\text{IH}}$ per motivi analoghi a quelli già definiti nel caso precedente.

$$t_{L \rightarrow H, \text{max}} = R_{\text{eq, max}} \cdot C_{\text{carico, E}} \cdot \ln \left(\frac{V_{DD}}{V_{DD} - V_{IH}} \right) = 0.36 \text{ ns}$$

▼ Caso L→H (I)

Per il calcolo delle $R_{\text{eq, min}}$ ed $R_{\text{eq, max}}$ dell'inverter, guardiamo la PUN, in cui notiamo che c'è un solo percorso possibile. Si ha, quindi:

$$R_{\text{eq}} = R_{\text{eq, min}} = R_{\text{eq, max}} = R_{\text{ON}}$$

Come per il punto E, per il calcolo di $t_{L \rightarrow H, \text{min}}$ usiamo $V_c(t) = V_{IL}$ e per il calcolo di $t_{L \rightarrow H, \text{max}}$ usiamo $V_c(t) = V_{IH}$.

$$t_{L \rightarrow H, \text{min}} = R_{\text{eq}} \cdot C_{\text{carico, I}} \cdot \ln \left(\frac{V_{DD}}{V_{DD} - V_{IL}} \right) = 0.69 \text{ ns}$$

$$t_{L \rightarrow H, \text{max}} = R_{\text{eq}} \cdot C_{\text{carico, I}} \cdot \ln \left(\frac{V_{DD}}{V_{DD} - V_{IH}} \right) = 1.79 \text{ ns}$$

▼ Spiegazione del caso H→L

Per il caso in cui è presente la transizione HIGH (H) → LOW (L), si deve:

- guardare la PDN, ricordando che si vuole raggiungere LOW;
- ricordando che si parte da HIGH, equivalente a DD, si ha:

$$V_0 = V_{DD} \quad V_\infty = V_{GND}$$

- sviluppando l'equazione generale, si ottiene:

$$t = \tau \cdot \ln \left(\frac{V_{DD}}{V_c(t)} \right)$$

▼ Caso H→L (E)

Utilizzeremo i valori $R_{\text{eq, min}}$ ed $R_{\text{eq, max}}$ del tempo di discesa calcolati all'obiettivo precedente, che d'ora in poi alleggeriremo omettendo l'apice PDN.

$$R_{\text{eq, min}} = \frac{5}{3} R_{\text{ON}}$$

$$R_{\text{eq, max}} = 3 R_{\text{ON}}$$

Per il calcolo di $t_{H \rightarrow L, \min}$ usiamo $V_c(t) = V_{IH}$ per due motivi:

- motivo fisico: essendo in discesa ($H \rightarrow L$), per il calcolo del tempo minimo usiamo la soglia più alta, poiché è quella che viene attraversata prima tra le due;
- motivo matematico: volendo trovare il tempo minimo, usiamo il valore più alto poiché, essendo col segno positivo al denominatore, è quello che minimizza il risultato della frazione. Detto in altri termini, se usassimo V_{IL} otterremmo un valore maggiore rispetto ad usare V_{IH} .

$$t_{H \rightarrow L, \min} = R_{eq, \min} \cdot C_{carico, E} \cdot \ln \left(\frac{V_{DD}}{V_{IH}} \right) = 0.03 \text{ ns}$$

Per il calcolo di $t_{L \rightarrow H, \max}$ usiamo $V_c(t) = V_{IH}$ per motivi analoghi a quelli già definiti nel caso precedente.

$$t_{H \rightarrow L, \max} = R_{eq, \max} \cdot C_{carico, E} \cdot \ln \left(\frac{V_{DD}}{V_{IL}} \right) = 0.21 \text{ ns}$$

▼ Caso $H \rightarrow L$ (I)

Per il calcolo delle $R_{eq, \min}$ ed $R_{eq, \max}$ dell'inverter, guardiamo la PDN, in cui notiamo che c'è un solo percorso possibile. Si ha, quindi:

$$R_{eq} = R_{eq, \min} = R_{eq, \max} = R_{ON}$$

Come per il punto E, per il calcolo di $t_{H \rightarrow L, \min}$ usiamo $V_c(t) = V_{IH}$ e per il calcolo di $t_{H \rightarrow L, \max}$ usiamo $V_c(t) = V_{IL}$.

$$t_{H \rightarrow L, \min} = R_{eq} \cdot C_{carico, I} \cdot \ln \left(\frac{V_{DD}}{V_{IH}} \right) = 0.18 \text{ ns}$$

$$t_{H \rightarrow L, \max} = R_{eq} \cdot C_{carico, I} \cdot \ln \left(\frac{V_{DD}}{V_{IL}} \right) = 0.69 \text{ ns}$$

Si noti come si ottiene lo stesso risultato ottenuto per la PDN, poiché i due network dell'inverter sono identici a livello di MOS.

▼ Ritardo complessivo (OUT)



Facciamo alcuni commenti preliminari:

- per il calcolo dei tempi minimi (massimi) totali si usano i tempi minimi (massimi) locali. Nella pratica, in un'equazione dobbiamo controllare che i pedici siano tutti minimi o tutti massimi;
- fisicamente parlando, i circuiti implementati tramite CMOS comportano **sempre** una variazione di livello logico. Questo significa che, in una serie di circuiti con porte CMOS, la somma tra i tempi di propagazione deve avere i pedici H→L e L→H alternati poiché **un MOS che riceve un valore logico basso/alto lo trasmette in uscita alto/basso.**

Come esempio, quando vorremo calcolare un $t_{H \rightarrow L}$ dovremo impostare i calcoli sapendo che l'**ultimo tempo sommato dovrà essere anch'esso $t_{H \rightarrow L}$** , in modo da garantire la coerenza sul valore logico finale e, a cascata, i tempi precedenti dovranno avere i **pedici alternati**, secondo l'ordine dato dal circuito stesso.

Definiamo alcuni esempi:

- circuito composto da 3 moduli:

$$t_{H \rightarrow L, \text{tot}} = t_{H \rightarrow L,1} + t_{L \rightarrow H,2} + t_{H \rightarrow L,3}$$

$$t_{L \rightarrow H, \text{tot}} = t_{L \rightarrow H,1} + t_{H \rightarrow L,2} + t_{L \rightarrow H,3}$$

- circuito composto da 4 moduli:

$$t_{H \rightarrow L, \text{tot}} = t_{L \rightarrow H,1} + t_{H \rightarrow L,2} + t_{L \rightarrow H,3} + t_{H \rightarrow L,4}$$

$$t_{L \rightarrow H, \text{tot}} = t_{H \rightarrow L,1} + t_{L \rightarrow H,2} + t_{H \rightarrow L,3} + t_{L \rightarrow H,4}$$

Per calcolare il tempo di propagazione complessivo $t_{\text{prop, tot}}$ dobbiamo considerare i quattro casi possibili:

1. $t_{H \rightarrow L, \text{min}}$;
2. $t_{H \rightarrow L, \text{max}}$;
3. $t_{L \rightarrow H, \text{min}}$;
4. $t_{L \rightarrow H, \text{max}}$.

Una volta visionati e capiti i [commenti preliminari](#), lo sviluppo dei calcoli per il nostro esercizio, in cui il circuito è composto da 2 parti, risulta come segue:

$$t_{H \rightarrow L, \text{tot,min}} = t_{L \rightarrow H, E, \text{min}} + t_{H \rightarrow L, I, \text{min}} = 0.22 \text{ nS}$$

$$t_{H \rightarrow L, \text{tot,max}} = t_{L \rightarrow H, E, \text{max}} + t_{H \rightarrow L, I, \text{max}} = 1.05 \text{ nS}$$

$$t_{L \rightarrow H, \text{tot,min}} = t_{H \rightarrow L, E, \text{min}} + t_{L \rightarrow H, I, \text{min}} = 0.72 \text{ nS}$$

$$t_{L \rightarrow H, \text{tot,max}} = t_{H \rightarrow L, E, \text{max}} + t_{L \rightarrow H, I, \text{max}} = 2.00 \text{ nS}$$

▼ Nota a posteriori

Per semplificare la trattazione, è stata omessa l'importanza di verificare sempre che $t_{\text{max}} > t_{\text{min}}$, ma, essendo i calcoli soggetti a piccoli errori di attenzione, questa verifica può risultare assai importante.

Esercizio 2 - Porte Logiche CMOS

▼ Creatore originale: @Gianbattista Busonera

- @<utente> (<data>): <descrizione della modifica>

Obiettivo 1 - Implementazione di funzione logica con CMOS

Implementare, tramite porte CMOS, la seguente funzione logica.

$$U = \overline{A} \cdot \overline{B} + \overline{C} \cdot \overline{D}$$

Come visto nella sezione di teoria relativa all'[implementazione delle porte logiche CMOS](#), è possibile implementare una funzione logica grazie a un circuito con rete

di pull down e rete di pull up. La via più conveniente è solitamente quella di partire dalla rete di pull-down (nMOS).

Per far ciò, è necessario avere la funzione logica U scritta in maniera tale che sia completamente negata. Per renderlo possibile, è necessario utilizzare le leggi di De Morgan per ricondurci a tale forma ed, eventualmente, ricorrere a una doppia negazione.

▼ Utilizzo delle leggi di De Morgan

Ricordiamo le leggi di De Morgan:

$$\overline{A \cdot B} = \overline{A} + \overline{B}$$

$$\overline{A \cdot \overline{B}} = \overline{\overline{A} + B}$$

Da queste, possiamo ricavare:

$$U = \overline{A + B} + \overline{C \cdot D}$$

▼ Utilizzo della doppia negazione

Utilizziamo la doppia negazione:

$$U = \overline{\overline{U}} = \overline{\overline{\overline{A + B} + \overline{C \cdot D}}}$$

Concentriamoci, per il momento, su $\overline{\overline{A + B} + \overline{C \cdot D}}$, notando che è stata rimossa una negazione dalla definizione di U , in modo da visualizzare meglio cosa succede.

$$\begin{cases} X = \overline{A + B} \\ Y = \overline{C \cdot D} \end{cases} \quad \overline{X + Y} = \overline{X} \cdot \overline{Y}$$

Si ha, quindi, che:

$$\overline{U} = \overline{\overline{\overline{A + B} + \overline{C \cdot D}}} = (A + B) \cdot (C \cdot D)$$

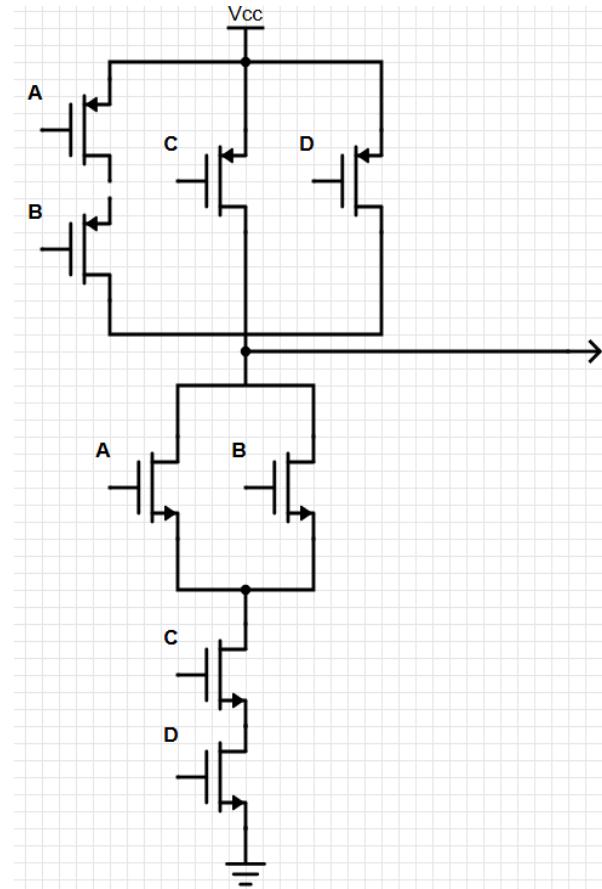
Ricordiamo, però, che precedentemente è stata rimossa una negazione, perciò:

$$U = \overline{\overline{U}} = \overline{(A + B) \cdot (C \cdot D)}$$

Otteniamo pertanto il [circuito CMOS equivalente](#) che implementa tale funzione logica:

Obiettivo 2 - Tempi di salita e di discesa

Determinare i tempi di salita e discesa minimi, sapendo che $R_{ON} = 20 \text{ k}\Omega$ e che l'uscita U è collegata ad un carico di 50 fF (50 femtofarad).



Circuito CMOS equivalente che implementa la funzione logica $U = \overline{\overline{U}} = \overline{(A + B) \cdot (C \cdot D)}$

▼ Calcolo del tempo di salita

Come visto nell'esercizio 1, per calcolare il tempo di salita è necessario guardare la rete di Pull Up (quella dove sono presenti i P-MOS).

Per il calcolo del tempo di salita minimo $t_{rise,min}$ è necessario considerare il maggior numero di percorsi possibili e, visto che quando i PMOS sono "accesi",

cioè fanno passare corrente, li possiamo considerare dei resistori con resistenza R_{ON} , conviene considerare come percorso "minimo":

$$\begin{aligned} R_{\text{eq, min}} &= (R_A + R_B) \parallel R_C \parallel R_D \\ &= 2R \parallel (R \parallel R) = 2R \parallel \frac{R}{2} \\ &= \frac{\frac{R}{2} \cdot 2R}{\frac{R}{2} + 2R} = \frac{2R^2}{5R} = \frac{2}{5}R \end{aligned}$$

Ci viene detto nel testo che il carico dell'uscita (rappresentato come una freccia verso destra) ha capacità pari a $C = 50 \text{ fF}$. Sapendo che $t_{\text{rise}} = 2.2\tau_{\text{rise}}$, possiamo calcolare:

$$t_{\text{rise, min}} = 2.2\tau_{\text{rise, min}} = 2.2R_{\text{eq, min}} \cdot C = 2.2 \left(\frac{2}{5}RC \right) = 0.88 \text{ ns}$$

Il calcolo del **tempo di salita massimo** differisce solo per il fatto che consideriamo il **percorso peggiore** che la corrente può attraversare (quello che farebbe impiegare più tempo). Chiaramente, questo percorso viene dato dall'attraversamento delle due resistenze A e B, senza considerare alcun parallelo.

$$R_{\text{eq, max}} = R_A + R_B = 2R$$

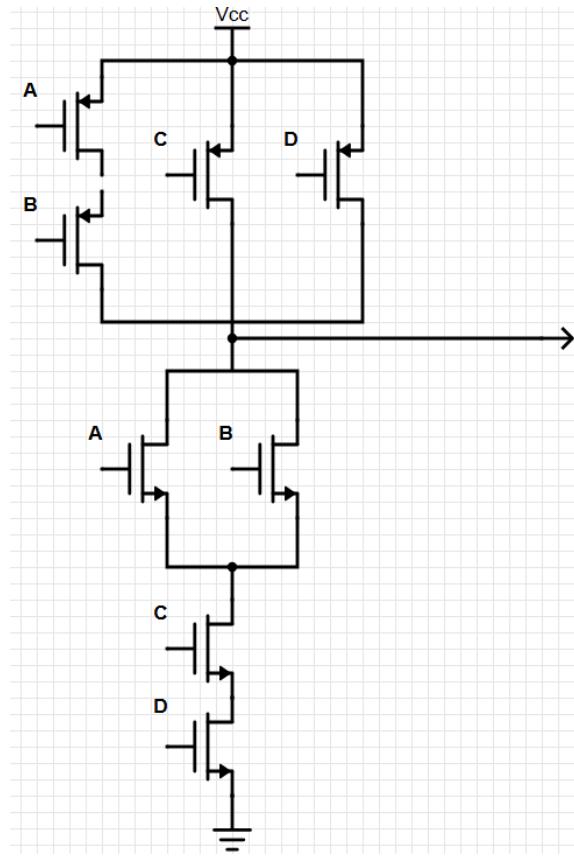
$$t_{\text{rise, max}} = 2.2\tau_{\text{rise, max}} = 2.2R_{\text{eq, max}} \cdot C = 2.2(2RC) = 4.4 \text{ ns}$$

▼ Tempo di discesa

Come visto nell'esercizio 1, per calcolare il tempo di discesa è necessario guardare la rete di Pull Down (quella dove sono presenti gli N-MOS).

Per il calcolo del tempo di discesa minimo è necessario considerare il maggior numero di percorsi possibili e, visto che quando gli NMOS sono "accesi", cioè fanno passare corrente, li possiamo considerare dei resistori con resistenza R_{ON} , conviene considerare come percorso "minimo":

$$\begin{aligned} R_{eq,min} &= (R_A \parallel R_B) + R_C + R_D \\ &= \frac{R}{2} + 2R = \frac{5}{2}R \end{aligned}$$



Circuito CMOS equivalente che implementa la funzione logica $U = \overline{\overline{U}} = \overline{(A + B) \cdot (C \cdot D)}$.

Ci viene detto che il carico dell'uscita (rappresentato come una freccia verso destra) ha capacità pari a $C = 50 \text{ fF}$. Sapendo che $t_{fall} = 2.2\tau_{fall}$, possiamo calcolare:

$$t_{fall,min} = 2.2\tau_{fall,min} = 2.2R_{eq,min} \cdot C = 2.2 \left(\frac{2}{5}RC \right) = 5.5 \text{ ns}$$

Il calcolo del tempo di discesa massimo differisce solo per il fatto che consideriamo il percorso peggiore che la corrente può attraversare (quello che farebbe impiegare più tempo). Chiaramente, questo percorso viene dato dall'attraversamento delle due resistenze A o B e dalla serie di C e D.

$$R_{eq,max} = R_A + R_C + R_B = 3R$$

$$t_{\text{fall,max}} = 2.2\tau_{\text{fall, max}} = 2.2R_{\text{eq,max}} \cdot C = 2.2(3RC) = 6.6 \text{ ns}$$

Esercizio 3 - Circuito sequenziale CMOS

▼ Creatori originali: @Gianbattista Busonera, @Francesco Ambrosino

- @<utente> (<data>): <descrizione della modifica>

Descrizione dei parametri del circuito in esame

Nel circuito in esame, i FF hanno $T_{\text{CK-Q}} = 0.1 \text{ ns}$, $T_{\text{SU}} = 0.15 \text{ ns}$ e capacità di ingresso C_{in} .

$$C_{\text{in}} = 20 \text{ fF}$$

Le due porte logiche combinatorie sono di tipo CMOS, in cui tutti i MOS hanno resistenze

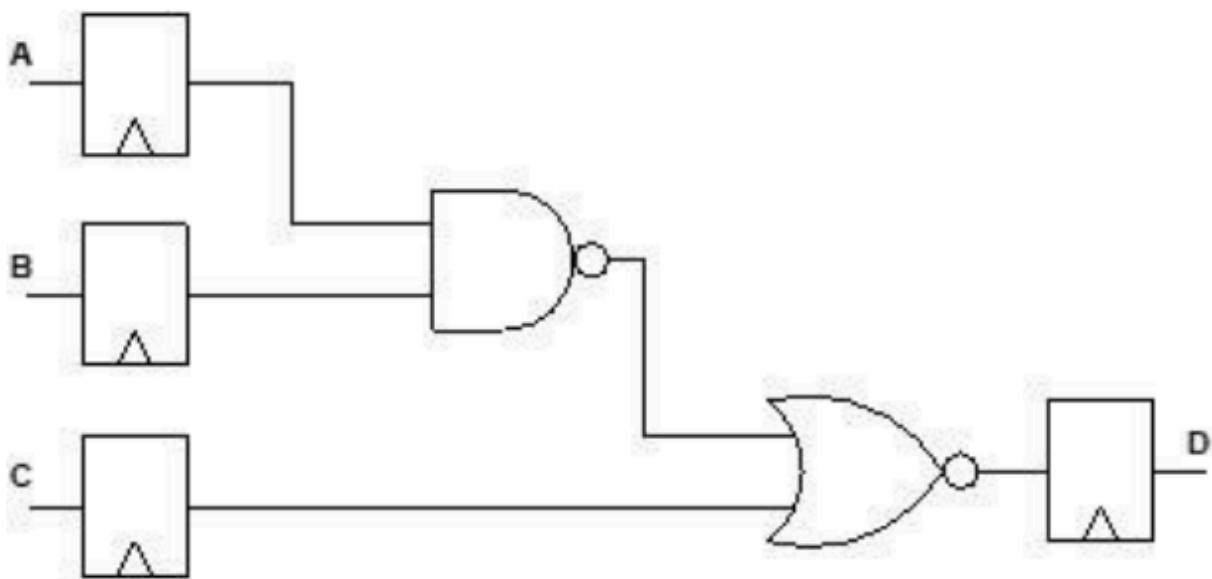
R_{ON} , capacità di gate C_g , I_{off} e V_{DD} .

$$R_{\text{ON}} = 20 \text{ k}\Omega$$

$$C_g = 5 \text{ fF}$$

$$I_{\text{off}} = 10 \text{ nA}$$

$$V_{\text{DD}} = 1.8 \text{ V}$$



Circuito sequenziale CMOS di riferimento



La tensione di soglia V_T che riguarda la tensione in cui si commuta da uno stato alto/basso ad uno basso/alto non è specificata e, pertanto, possiamo assumerla per entrambi i casi implicitamente pari a:

$$V_T = \frac{V_\infty - V_0}{2} = \frac{V_\infty}{2}$$

Questa situazione semplifica la trattazione e i calcoli:

$$V_c(t) = V_\infty + (V_0 - V_\infty)e^{-\frac{t}{\tau}} = V_T = \frac{V_\infty}{2}$$

$$e^{-\frac{t}{\tau}} = \frac{V_T - V_\infty}{V_0 - V_\infty} = \frac{V_\infty/2 - V_\infty}{-V_\infty} = \frac{1}{2}$$

$$t = \ln 2\tau = 0.69\tau$$

Questo è un risultato notevole, da ricordare perché utilizzato sempre quando il valore della tensione V_T non è reso noto dai dati del testo.

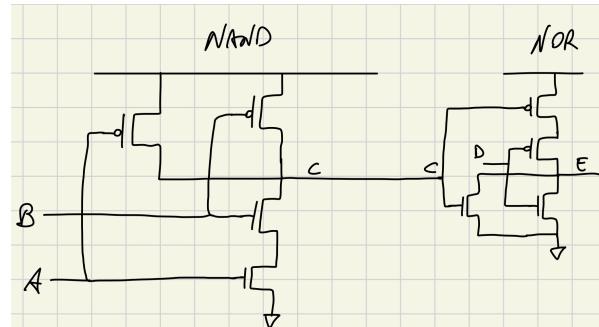
Obiettivo 1 - Ritardi di propagazione CMOS

Lo svolgimento è simile a quello fatto nell'[esercizio 1 obiettivo 2.2](#).

Calcolare i ritardi $t_{p, L \rightarrow H, \max}$ e $t_{p, H \rightarrow L, \max}$ della porta NAND e NOR.

Per quanto appena discusso, si userà in entrambi i casi:

$$t = 0.69\tau = 0.69 \cdot R_{eq} \cdot C_{carico}$$



Rappresentazione porta NAND in serie con porta NOR con condensatori

▼ Porta NAND

Si ha $C_{carico} = 2C_g$, poiché all'uscita della porta NAND c'è solo il parallelo dei condensatori nel punto identificato dalla targa C (si tratta di un NMOS e di un PMOS), che equivale alla somma delle capacità C_g , per questo C_{eq} non vale $4C_g$.

Studiamo i due casi:

- caso $L \rightarrow H$:

Guardando la PUN della NAND, per il calcolo delle resistenze equivalenti possiamo prendere una sola resistenza o il parallelo delle due resistenze, come sappiamo dall'[esercizio 1](#).

Per il calcolo della $R_{eq, \max}$ dobbiamo evitare i parallelî, poiché diminuiscono il valore totale della resistenza equivalente.

$$R_{eq, \max} = R_{ON}$$

Si ottiene, quindi, il tempo di propagazione richiesto in transizione $L \rightarrow H$ per la porta NAND come segue:

$$t_{p, L \rightarrow H, \max}^{NAND} = 0.69 \cdot R_{ON} \cdot 2C_g = 0.138 \text{ nS}$$

- caso $H \rightarrow L$:

Guardando la PDN della NAND, possiamo prendere solo la serie delle resistenze.

$$R_{\text{eq, max}} = 2R_{\text{ON}}$$

Si ottiene, quindi, il tempo di propagazione richiesto in transizione $H \rightarrow L$ per la porta NAND come segue:

$$t_{p, H \rightarrow L, \text{max}}^{\text{NAND}} = 0.69 \cdot 2R_{\text{ON}} \cdot 2C_g = 0.276 \text{ nS}$$

▼ Porta NOR

Si ha $C_{\text{carico}} = C_{\text{in}}$, poiché all'uscita della porta NOR c'è il flip-flop D, il quale ha capacità di ingresso C_{in} .

Studiamo i due casi:

- caso $L \rightarrow H$:

Guardando la PUN della NOR, possiamo prendere solo la serie delle resistenze.

$$R_{\text{eq, max}} = 2R_{\text{ON}}$$

Si ottiene, quindi, il tempo di propagazione richiesto in transizione $L \rightarrow H$ per la porta NOR come segue:

$$t_{p, L \rightarrow H, \text{max}}^{\text{NOR}} = 0.69 \cdot 2R_{\text{ON}} \cdot C_{\text{in}} = 0.552 \text{ nS}$$

- caso $H \rightarrow L$:

Guardando la PDN della NOR, per il calcolo delle resistenze equivalenti possiamo prendere una sola resistenza o il parallelo delle due resistenze, come sappiamo dall'[esercizio 1](#).

Per il calcolo della $R_{\text{eq, max}}$ dobbiamo evitare i paralleli, poiché diminuiscono il valore totale della resistenza equivalente.

$$R_{\text{eq, max}} = R_{\text{ON}}$$

Si ottiene, quindi, il tempo di propagazione richiesto in transizione $H \rightarrow L$ per la porta NOR come segue:

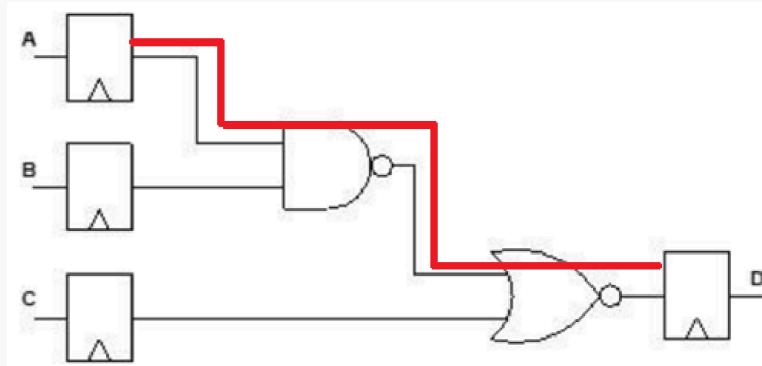
$$t_{p, H \rightarrow L, \max}^{\text{NOR}} = 0.69 \cdot R_{\text{ON}} \cdot C_{\text{in}} = 0.276 \text{ nS}$$

Obiettivo 2 - Frequenza massima del circuito

Calcolare la massima frequenza di clock del circuito F_{max} .



Detto in altri termini: qual è il tempo minimo di caso peggiore (il massimo tempo per far propagare il segnale dalle porte logiche) che garantisce al circuito il corretto funzionamento?



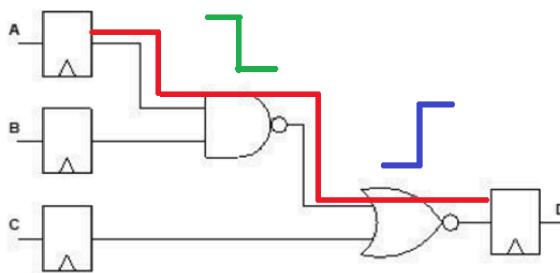
E' necessario, pertanto, cercare il **tempo minimo necessario all'attraversamento del percorso più lungo** che connette due FF.

In questo caso, prendiamo come riferimento l'uscita del FF-A e l'ingresso del FF-D, ma anche l'uscita del FF-B e l'ingresso del FF-D sarebbero adeguate a tale scopo.

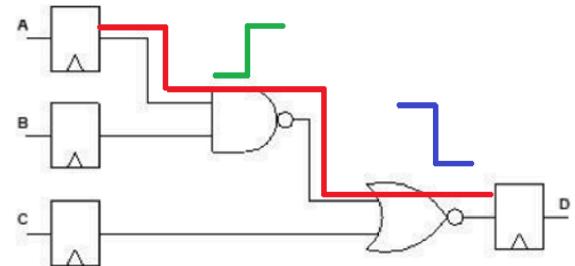
▼ Impostazione

Dobbiamo distinguere il tempo di arrivo massimo in due casi:

- propagazione di un segnale inizialmente basso da A o B (la porta NAND è sempre in stato alto se A = 0 o B = 0);
- propagazione di un segnale inizialmente alto (A = 1 e B = 1).



Transizione alto basso seguita da una basso alto



Transizione basso alto seguita da una alto basso.

La formula sacra da considerare per calcolare il tempo di clock minimo che garantisce il funzionamento del circuito è come segue:

$$t_{\text{arrivo}} < t_{\text{richiesto}}$$

Definendo $t_{\text{arrivo}} = t_{\text{CK},q} + t_{\text{percorso}}$ e $t_{\text{richiesto}} = T_{\text{CK}} - t_{\text{setup}}$.

$$T_{\text{CK}} > t_{\text{CK},Q} + t_{\text{setup}} + t_{\text{percorso}}$$

▼ Propagazione di un segnale inizialmente alto

1. Abbiamo assunto di partire dal caso in cui A o B siano 0 e, di conseguenza, la porta NAND dovrà passare da uno stato basso a uno alto (in quanto deve dare 1 come risultato);
2. Il tempo di propagazione massimo necessario ad attraversare il percorso tra A (o B) e la porta NAND è dato da $t_{p,\text{NAND},\text{max}}^{\text{L}\rightarrow\text{H}}$, calcolato nell'obiettivo 1 di questo esercizio;
3. Adesso che siamo ad uno stato alto entriamo nella porta NOR, che darà un risultato sicuramente basso per come è implementata la porta NOR (se diamo un 1 in ingresso, quello della porta NAND, sicuramente darà un risultato basso);
4. Il tempo di propagazione massimo necessario ad attraversare il percorso tra la porta NAND e la porta NOR è dato da $t_{p,\text{NOR},\text{max}}^{\text{H}\rightarrow\text{L}}$, calcolato nell'obiettivo 1 di questo esercizio.

Otteniamo quindi:

$$t_{p,1} = t_{p,\text{NAND},\text{max}}^{\text{L}\rightarrow\text{H}} + t_{p,\text{NOR},\text{max}}^{\text{H}\rightarrow\text{L}} = 0.414 \text{ ns}$$

▼ Propagazione di un segnale inizialmente basso

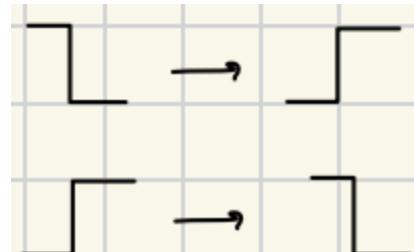
In questo caso, invece, non ci sono evidenti forzature dell'uscita della porta NOR, ma si cerca un tempo di caso peggiore, e quindi il maggior numero di transizioni non considerate finora:

$$t_{p,2} = t_{p,\text{NAND},\text{max}}^{\text{H}\rightarrow\text{L}} + t_{p,\text{NOR},\text{max}}^{\text{L}\rightarrow\text{H}} = 0.828 \text{ ns}$$

▼ Frequenza massima

In ogni caso, i tempi di propagazione saranno tali da mantenere una certa continuità tra stati logici:

- se la NAND passa da alto a basso, la NOR dovrà passare da basso ad alto;
- se la NAND passa da basso ad alto, la NOR dovrà passare da alto a basso.



Visualizzazione della relazione
NAND-NOR definita

Tra i due tempi di propagazione $t_{p,1}$ e $t_{p,2}$, sceglieremo il tempo **maggior**. Dalla [formula vista nella sezione di implementazione](#), si ottiene che $T_{\text{CK,min}}$ è definita come:

$$\begin{aligned} T_{\text{CK,min}} &= t_{\text{CK,q}} + t_{\text{setup}} + t_{\text{LC,max}} \\ &= 0.1 \text{ ns} + 0.15 \text{ ns} + t_{p,2} \\ &= 0.1 + 0.15 + 0.828 \text{ ns} = 1.078 \text{ ns} \end{aligned}$$

Infine, si può calcolare la frequenza massima F_{max} come:

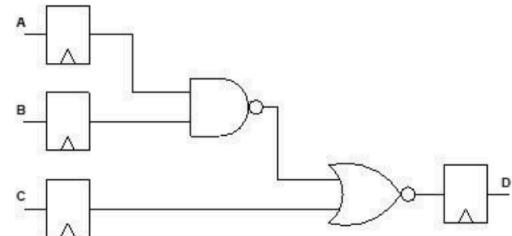
$$F_{\text{max}} = \frac{1}{T_{\text{CK,min}}} = 928 \text{ MHz}$$

Obiettivo 3 - Valore massimo del tempo di hold



Per svolgere questo esercizio, si rimanda alla teoria: [LINK](#).

Calcolare il valore massimo del tempo di hold (T_H) dei flip-flop che consente di evitare violazioni.



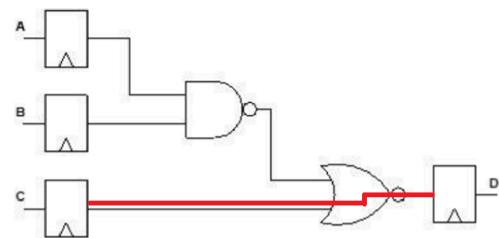
Circuito sequenziale di riferimento

▼ Determinazione del percorso più breve

Ci potrebbero venire in mente due alternative:

1. scegliere il percorso con la porta NAND e la porta NOR;
2. scegliere il percorso con la sola porta NOR.

Chiaramente, il secondo è un **sottoinsieme** del primo e, pertanto, sarà sicuramente lui il percorso più breve.



Visualizzazione del percorso più breve

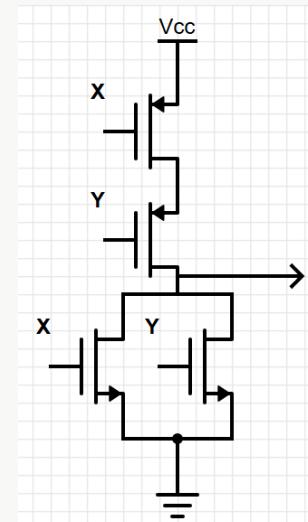


Prima di considerare i due casi di transizione, è opportuno ricordare come è implementata una porta NOR.

Una porta NOR implementa, in genere, una funzione logica:

$$g(X, Y) = \overline{X + Y}$$

Si implementa, quindi, con un parallelo tra X e Y nella PDN e una serie tra X e Y della PUN.



Visualizzazione della porta NOR implementata in CMOS

Il tempo di propagazione minimo è dato dal percorso che implementa la porta NOR, ed è definibile attraverso:

- transizione da basso ad alto della NOR ($L \rightarrow H$);

Per considerare il tempo necessario per passare da uno stato basso ad uno alto, di base è necessario guardare la PUN (rete P-MOS). Dal punto precedente otteniamo:

$$t_{p,NOR, \min}^{L \rightarrow H} = 0.276 \text{ ns}$$

- transizione da alto a basso della NOR ($H \rightarrow L$): $t_{p,NOR,min}^{H \rightarrow L}$

Per considerare il tempo necessario per passare da uno stato alto ad uno basso, di base è necessario guardare la PDN (rete N-MOS).

E' evidente che sia il **tempo minimo di propagazione**, in quanto è presente un parallelo e, di conseguenza:

$$R_{\text{eq,min}} = R_{\text{ON}} \| R_{\text{ON}} = \frac{R_{\text{ON}}}{2}$$

Ricordando che il carico è un flip flop, di cui conosciamo la capacità equivalente in ingresso C_{in} dai dati dell'esercizio, otteniamo:

$$t_{p,\text{NOR,min}}^{\text{H} \rightarrow \text{L}} = 0.69 \cdot \tau = 0.69 \cdot R_{\text{eq,min}} C_{\text{carico}} \quad (1)$$

$$= 0.69 \frac{R_{\text{ON}}}{2} C_{\text{in}} = 0.69 \cdot 10 \text{ k}\Omega \cdot 20 \text{ fF} = 0.138 \text{ ns} \quad (2)$$

Il caso limite è presentato, dunque, dal tempo di propagazione in cui la porta NOR passa da uno stato alto ad uno basso, in quanto impiega meno tempo e, di conseguenza, potrebbe sporcare il dato memorizzato del flip flop prima che lo memorizzi completamente.

$$t_{p,\text{LC,min}} = t_{p,\text{NOR,min}}^{\text{H} \rightarrow \text{L}}$$

▼ Calcolo del tempo di hold

La condizione da rispettare, di cui ricordiamo la [relativa lezione di teoria](#), è la seguente:

$$t_{\text{arrivo}} > t_{\text{richiesto}}$$

L'equazione si espande come segue:

$$t_{\text{CK-Q}} + t_{p,\text{LC,min}} > t_{\text{hold}} \quad (3)$$

$$t_{\text{hold}} \leq t_{\text{CK-Q}} + t_{p,\text{NOR,min}} \quad (4)$$

$$t_{\text{hold}} \leq 0.1 \text{ ns} + 0.138 \text{ ns} \quad (5)$$

$$t_{\text{hold}} \leq 0.238 \text{ ns} \quad (6)$$

Si ottiene che il tempo massimo di hold vale:

$$t_{\text{hold,max}} = 0.238 \text{ ns}$$

! Nella sua soluzione sbaglia anche a considerare anche il tempo di propagazione della NAND ma a lezione ha considerato solo la NOR...
P.S. (15/05/2025) Ho chiesto a Casu e ha confermato la mia soluzione. 😊

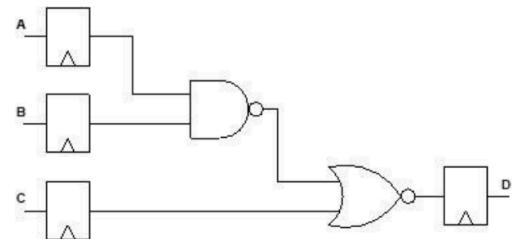
Obiettivo 4 - Potenza dinamica e statica

▼ Creatore originale: @Francesco Ambrosino



Per svolgere questo esercizio, si rimanda alla teoria: [LINK](#).

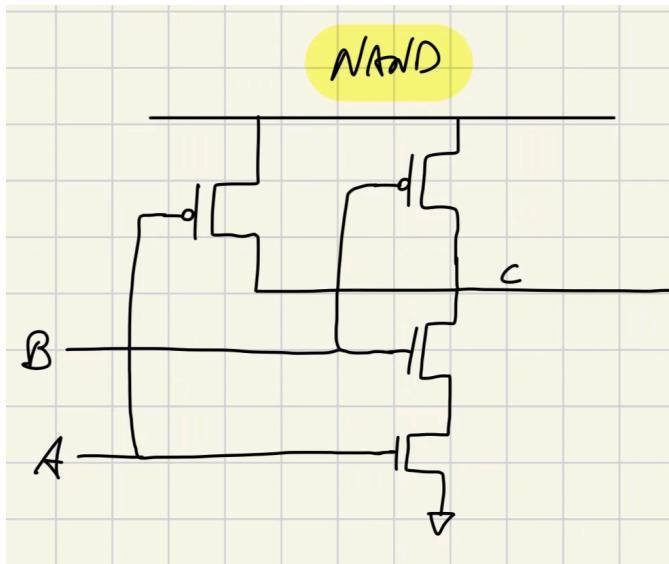
Calcolare la potenza dinamica e statica consumata dalle due porte NAND e NOR nell'ipotesi che ogni ingresso abbia activity $\alpha = 0.2$.



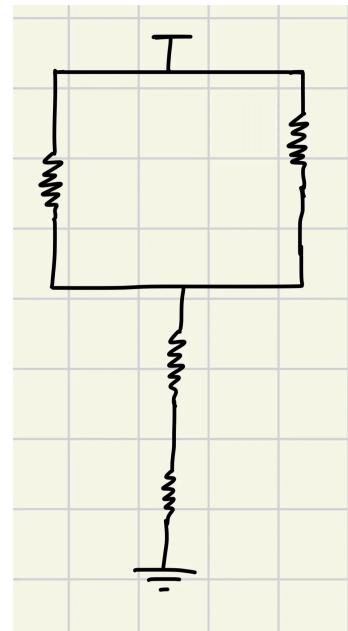
Circuito sequenziale di riferimento

▼ Potenza statica (NAND)

Definiamo due visualizzazioni che torneranno utili nel capire il calcolo della potenza statica nella porta NAND.



Visualizzazione della porta NAND implementata in CMOS



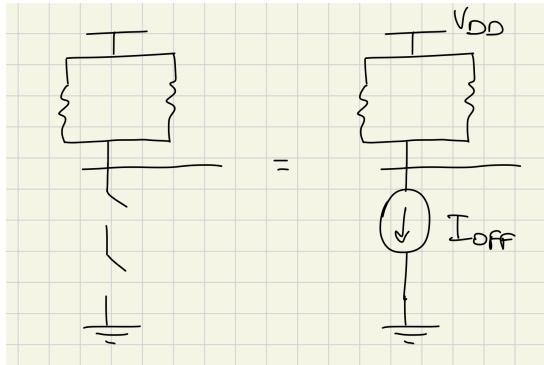
Visualizzazione della porta NAND con le sole resistenze

Consideriamo le varie casistiche possibili per i valori di ingresso di A e B :

- $A = 0, B = 0$:

In questa configurazione le resistenze nella PDN sono circuiti aperti, mentre quelle nella PUN sono circuiti chiusi, essendo i valori di A e B negati nella PDN, e quindi diretti nella PUN.

La situazione è rappresentata nell'immagine a lato, da cui si evince che:



Rappresentazione del caso $A = 0, B = 0$

$$P_{\bar{A}, \bar{B}} = V_{DD} \cdot I_{OFF}$$

- $A = 0, B = 1$ oppure $A = 1, B = 0$:

In queste configurazioni equivalenti la PDN globalmente risulta un circuito aperto (si noti come solo una delle 2 resistenze è un circuito aperto, ma sono in serie tra loro), mentre la PUN è costituita da una resistenza e un circuito aperto (che, essendo in parallelo tra loro, risultano equivalenti ad una resistenza).

La situazione è rappresentata nell'[immagine a lato](#), da cui si evince che:

$$P_{\bar{A},B} = P_{A,\bar{B}} = V_{DD} \cdot I_{OFF}$$

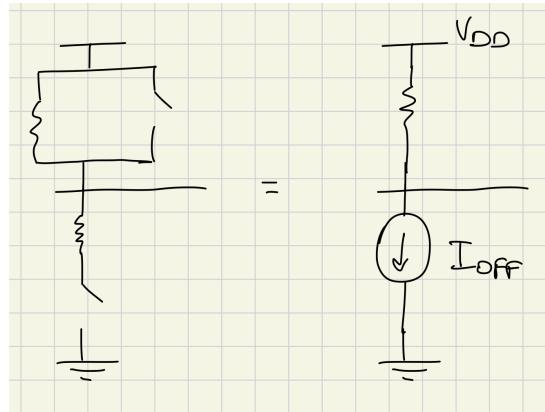
- $A = 1, B = 1$:

In questa configurazione le resistenze nella PDN sono circuiti chiusi, mentre quelle nella PUN sono circuiti aperti, portando ad una situazione speculare al [primo caso](#), essendo i valori di A e B diretti nella PDN, e quindi negati nella PUN.

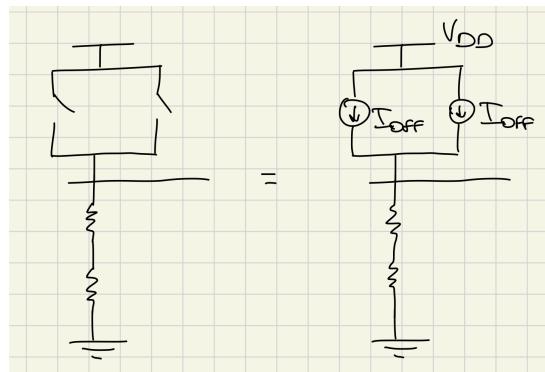
La situazione è rappresentata nell'[immagine a lato](#), da cui si evince che:

$$P_{A,B} = V_{DD} \cdot 2I_{OFF}$$

Dati i valori della potenza per tutte le possibili configurazioni, calcoliamo la potenza statica media per la porta NAND:



Rappresentazione del caso $A = 1, B = 0$
e del caso $A = 0, B = 1$

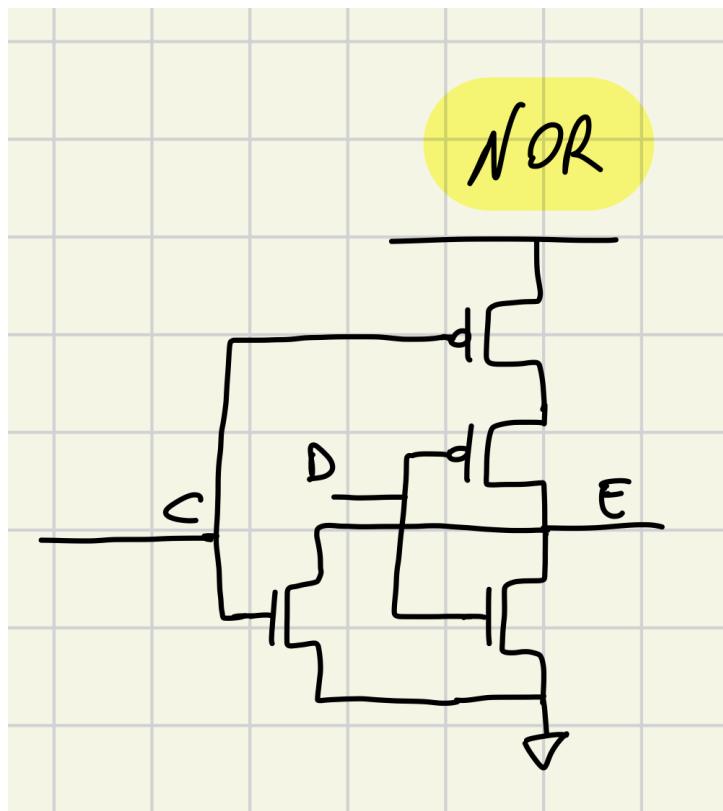


Rappresentazione del caso $A = 1, B = 1$

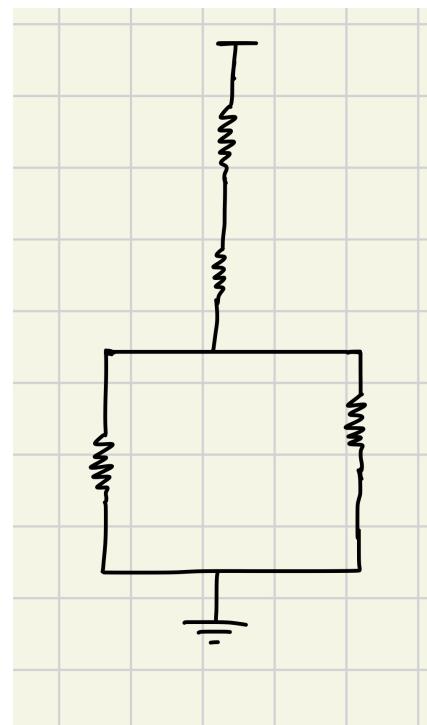
$$P_{s, \text{media}}^{\text{NAND}} = \frac{3}{4} \cdot V_{DD} \cdot I_{OFF} + \frac{1}{4} \cdot V_{DD} \cdot 2I_{OFF} = 22.5 \text{ mW}$$

▼ Potenza statica (NOR)

Definiamo due visualizzazioni che torneranno utili nel capire il calcolo della potenza statica nella porta NOR.



Visualizzazione della porta NOR implementata in CMOS



Visualizzazione della porta NOR con le sole resistenze

Il ragionamento è analogo a quanto [discusso per la porta NAND](#).

Essendo le visualizzazioni con le sole resistenze della NAND e della NOR specchiate rispetto al punto di incontro di PDN e PUN, si avranno situazioni opposte per i casi con variabili entrambe concordi e stessa situazione per i casi con variabili discordi.

Consideriamo le varie casistiche possibili per i valori di ingresso di A e B :

- $A = 0, B = 0$:

in questa configurazione le resistenze nella PDN sono circuiti aperti, mentre quelle nella PUN sono circuiti chiusi, essendo i valori di A e B negati nella PDN, e quindi diretti nella PUN.

Avendo i circuiti aperti della PDN in parallelo, si ha che:

$$P_{\bar{A}, \bar{B}} = V_{DD} \cdot 2I_{OFF}$$

- $A = 0, B = 1$ oppure $A = 1, B = 0$:

E' una situazione analoga al [caso della porta NAND](#), quindi si ha che:

$$P_{\bar{A}, B} = P_{A, \bar{B}} = V_{DD} \cdot I_{OFF}$$

- $A = 1, B = 1$:

In questa configurazione le resistenze nella PDN sono circuiti chiusi, mentre quelle nella PUN sono circuiti aperti, essendo i valori di A e B diretti nella PDN, e quindi negati nella PUN.

Avendo i circuiti aperti della PUN in serie, si ha che:

$$P_{A, B} = V_{DD} \cdot I_{OFF}$$

Il calcolo della potenza statica media per la porta NOR è banale e superfluo, essendo i valori identici a quelli ottenuti per la NAND.

$$P_{s, \text{media}}^{\text{NOR}} = \frac{3}{4} \cdot V_{DD} \cdot I_{OFF} + \frac{1}{4} \cdot V_{DD} \cdot 2I_{OFF} = 22.5 \text{ mW} = P_{s, \text{media}}^{\text{NAND}}$$

▼ Nota sulla potenza statica

Talvolta il testo potrebbe fornire informazioni esplicite circa la probabilità del verificarsi delle singole configurazioni. Qualora tali informazioni siano assenti, come nel caso di questo esercizio, si assumono le (in questo caso) **4 configurazioni equi-probabili** e si procede al calcolo della potenza statica media utilizzando la media aritmetica, come fatto.

▼ Potenza dinamica

Come formula generale, per la potenza dinamica P_d abbiamo la seguente:

$$P_d = f \cdot C \cdot V_{DD}^2$$

La traccia dell'esercizio ci fornisce un valore di activity (α), e quindi sappiamo che la nostra uscita non cambia stato in ogni periodo di clock, ma ogni N periodi.

Attraverso la relazione $T = 2NT_{\text{CK}}$, possiamo ottenere:

$$f = \frac{1}{2N} f_{\text{CK}}$$

con $f_{\text{CK}} = f_{\text{MAX}} = 928 \text{ MHz}$, calcolata all'[obiettivo 2](#).



Si noti come, se l'uscita commutasse ogni periodo di clock, avremmo $f = \frac{1}{2} f_{\text{CK}}$ e il testo dell'esercizio non ci avrebbe fornito un valore di α .

.

Ricordando che $\alpha = \frac{1}{N}$, otteniamo la formula finale:

$$P_d = \frac{0.2}{2} F_{\text{MAX}} \cdot C \cdot V_{\text{DD}}^2 = 0.1 F_{\text{MAX}} \cdot C \cdot V_{\text{DD}}^2$$

Essendo la tensione fornita dal testo e la frequenza massima un valore già calcolato, l'unica incognita da calcolare è la capacità di carico C , diversa per le due porte, ma calcolata già all'[obiettivo 1](#).

Calcoliamo, quindi, l'effettiva potenza dinamica per le due porte:

- porta NAND;

$$C_{\text{NAND}} = 2C_g$$

$$P_d^{\text{NAND}} = 0.1 F_{\text{MAX}} \cdot 2C_g \cdot V_{\text{DD}}^2 = 3 \text{ mW}$$

- porta NOR.

$$C_{\text{NOR}} = C_{\text{in}}$$

$$P_d^{\text{NOR}} = 0.1 F_{\text{MAX}} \cdot C_{\text{in}} \cdot V_{\text{DD}}^2 = 6 \text{ mW}$$

Si ha che la potenza dinamica totale del circuito risulta essere:

$$P_{d, \text{TOT}} = P_d^{\text{NAND}} + P_d^{\text{NOR}} = 9 \text{ mW}$$

5

Esercitazione 5 + iverilog e gtkwave

[Installazione iverilog e gtkwave](#)

[Esercizio 1 - sommatore completo](#)

[Soluzione 1 - porte logiche](#)

[Soluzione 2 - assegnazione continua](#)

[Soluzione 3 - modellazione comportamentale](#)

[Implementazione testbench e verifica](#)

[Esercizio 2 - multiplexer](#)

[Soluzione 1 - assegnazione continua](#)

[Soluzione 2 - modellazione comportamentale](#)

[Testbench e verifica](#)

[Esercizio 3 - latch trasparente](#)

[Soluzione](#)

[Testbench](#)

[Esercizio 4 - FSM](#)

[Soluzione](#)

[Testbench](#)

[GTKWave e visualizzazione forme d'onda](#)

[Esercizio 5 - Shift-Register completo](#)

[Soluzione](#)

[Esercizi extra](#)

▼ Installazione iverilog e gtkwave

▼ Creatore originale: @Gianbattista Busonera

[LINK DOWNLOAD, TUTORIAL](#)

Per utilizzarlo su windows occorre aggiungere alla variabile di sistema \$PATH il path relativo a iverilog e gtkwave.

OneDrive	C:\Users\gianb\AppData\Roaming\Python\Python311\Scripts
OneDriveConsumer	C:\Users\gianb\AppData\Local\GitHubDesktop\bin
Path	C:\Users\gianb\AppData\Local\Programs\Microsoft VS Code\bin
TEMP	C:\MinGW\bin
TMP	C:\iverilog\bin\iverilog.exe
	C:\iverilog\bin



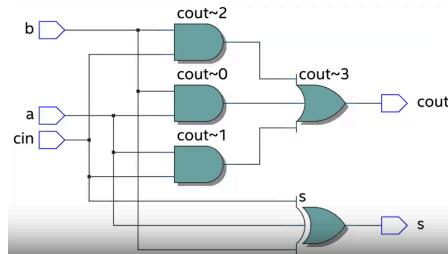
Durante l'installazione di iverilog vi chiederà se volete installare anche gtkwave e se volete aggiungerle alle variabili di sistema... Cliccate sì, facilita di tanto l'installazione.
Fatto ciò riavviate il computer!



Esercizio 1 - sommatore completo

▼ Creatore originale: @Gianbattista Busonera

Si supponga di voler implementare un componente che realizza un sommatore dato il seguente schema:



▼ Soluzione 1 - porte logiche

```
module fadd (cout, s, a, b, cin);
    output cout, s;
    input a, b, cin;
    wire and_b_cin;
    wire and_b_a;
    wire and_a_cin;

    and(and_b_cin, b, cin);
    and(and_b_a, b, a);
    and(and_a_cin, a, cin);
    or(cout, and_b_cin, and_b_a, and_a_cin);
    xor(s, a, b, cin);
endmodule
```

▼ Soluzione 2 - assegnazione continua

```
module fadd (cout, s, a, b, cin);
    output cout, s;
    input a, b, cin;
```

```

assign s = a^b^cin;
assign cout = (a&b)|(a&cin)|(b&cin);
endmodule

```

▼ Soluzione 3 - modellazione comportamentale

Si ricordi che stiamo modellando un circuito combinatorio e, pertanto, l'uscita cambia ogni volta cambiano gli ingressi.

E' pertanto necessario inserire nella lista di sensibilità tutti gli ingressi del nostro circuito combinatorio (a, b, cin).

```

module fadd (s, cout, a, b, cin);
    output reg s, cout;
    input a,b,cin;

    always @(*) begin
        s = a^b^cin;
        cout = (a & b) | (a & cin) | (b & cin);
    end
endmodule

```

▼ Implementazione testbench e verifica

```

`timescale 1ns/1ps

module fadd_tb;
    // segnali per collegarsi al modulo fadd
    reg a, b, cin;
    wire s, cout;
    // istanza del modulo da testare
    fadd uut (
        .s(s),
        .cout(cout),
        .a(a),
        .b(b),
        .cin(cin)
    );
    // procedura di test
    initial begin
        $display(" a | b | cin | s | cout ");
        $display("-----");
        // test per tutte le combinazioni
        for (integer i = 0; i < 8; i = i + 1) begin
            {a, b, cin} = i[2:0]; // assegna valori da 000 a 111
            #1; // attesa di 1ns per propagazione
            $display(" %b | %b | %b | %b | %b", a, b, cin, s, cout);
        end
        $finish; // termine modellazione
    end
endmodule

```

Come sfruttare un testbench

Bisogna salvare (per semplicità) nella stessa cartella i file fadd.v e fadd_tb.v (dove l'estensione .v indica che il codice è scritto in verilog). Una volta fatto ciò e, verificata la corretta installazione di iverilog occore aprire il terminale (cmd) nella cartella di riferimento e digitare:

```
iverilog -o fadd_tb fadd.v fadd_tb.v
vvp fadd_tb
```

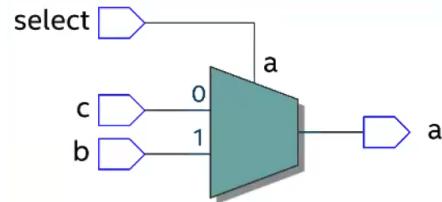
Il primo comando compilerà il testbench dandogli nome "fadd_tb", il secondo comando lo eseguirà e, a schermo, vedremo:

```
PS C:\Users\gianb\Desktop\UNI\ELAP\Lab\E05\ES01> iverilog -o fadd_tb fadd.v fadd_tb.v
PS C:\Users\gianb\Desktop\UNI\ELAP\Lab\E05\ES01> vvp fadd_tb
a | b | cin | s | cout
-----
0 | 0 | 0 | 0 | 0
0 | 0 | 1 | 1 | 0
0 | 1 | 0 | 1 | 0
0 | 1 | 1 | 0 | 1
1 | 0 | 0 | 1 | 0
1 | 0 | 1 | 0 | 1
1 | 1 | 0 | 0 | 1
1 | 1 | 1 | 1 | 1
fadd_tb.v:30: $finish called at 8000 (1ps)
```

Esercizio 2 - multiplexer

▼ Creatore originale: @Gianbattista Busonera

Si implementi un multiplexer 2→1 come in figura



▼ Soluzione 1 - assegnazione continua

Soluzione funzionante ma un po' bovina in quanto utilizza il costrutto condizionale ternario presente anche in C. Non si può fare diversamente in quanto gli if, case, etc. sono utilizzabili solo nei costrutti always e initial.

```
module mux2x1 (a, b, c, select);
    output a;
    input b, c, select;

    assign a = (select ? b : c); // se select = 1 a = b, se select = 0, a = c
        // operatore ternario
endmodule
```

▼ Soluzione 2 - modellazione comportamentale

Soluzione decisamente migliore dal punto di vista della leggibilità in quanto più vicino al nostro modo di scrivere codice:

```
module mux2x1 (output reg a, input b, input c, input select);
    always @(*) begin
        if(select == 1)
            a = b;
        else
```

```

    a = c;
end
endmodule

```

▼ Testbench e verifica

```

`timescale 1ns/1ps

module mux2x1_tb;

// segnali per collegarsi al modulo
reg in0, in1, sel;
wire out;

// istanza del modulo da testare
mux2x1 uut (
    .out(out),
    .in1(in1),
    .in0(in0),
    .sel(sel)
);

initial begin
    $display("in1 in0 sel | out");
    $display("-----");

    // test per tutte le 8 combinazioni
    for (integer i = 0; i < 8; i = i + 1) begin
        {in1, in0, sel} = i[2:0]; // assegna ogni combinazione
        #1; // attesa per stabilizzazione
        $display("%b %b %b | %b", in1, in0, sel, out);
    end

    $finish;
end

endmodule

```

E, in uscita, otteniamo:

Come preventivamente, se sel = 0, out = c

se sel = 1, out = b.

b	c	sel		out
0	0	0		0
0	0	1		0
0	1	0		1
0	1	1		0
1	0	0		0
1	0	1		1
1	1	0		1
1	1	1		1

Esercizio 3 - latch trasparente

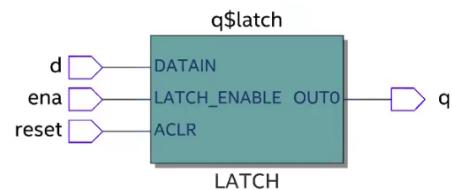
▼ Creatore originale: @Gianbattista Busonera

Si chiede sostanzialmente:

Se enableA è uguale a 1, l'uscita q deve diventare pari a d;

Se il reset è attivo (pari a 1), l'uscita q deve diventare pari a 0.

Il reset deve essere di tipo asincrono!



▼ Soluzione

```
module dlatch (q, d, ena, reset);
    output reg q;
    input d;
    input ena;
    input reset;
    // DEVO NECESSARIAMENTE USARE UN BLOCCO ALWAYS
    always @(d or ena or reset) begin
        if(reset) // il reset ha priorità rispetto al valore dell'ingresso!
            q = 0;
        else if(ena) // solo se enable a = 1 => q = d;
            q = d;
    end
endmodule
```



Dapprima inserire d nei parametri di sensibilità potrebbe sembrare insensato ma immaginiamo il caso:

- reset = 0; (costante)
- enableA = 1; (costante)
- d cambia da 0 a 1;

In questo caso sarebbe necessario che l'uscita q vari come è variato d.

▼ Testbench

```
'timescale 1ns/1ps

module tb_dlatch();
    // Segnali di test
    reg d;
    reg ena;
    reg reset;
    wire q;

    // Istanza del modulo da testare
    dlatch uut (
        .q(q),
        .d(d),
        .ena(ena),
```

```

.reset(reset)
};

// Inizializzazione
initial begin
    $dumpfile("dlatch_waveform.vcd");
    $dumpvars(0, tb_dlatch);

    // Inizializzazione ingressi
    d = 0;
    ena = 0;
    reset = 1;

    // Test del reset
    #10 reset = 0;

    // Test enable con d=0
    #10 ena = 1;
    #10 d = 1;
    #10 d = 0;

    // Test disable
    #10 ena = 0;
    #10 d = 1; // q non dovrebbe cambiare
    #10 d = 0;

    // Test enable di nuovo
    #10 ena = 1;
    #10 d = 1;

    // Test reset asincrono
    #10 reset = 1;
    #10 reset = 0;

    // Fine simulazione
    #20 $finish;
end

// Monitor per visualizzare i cambiamenti
initial begin
    $monitor("Time = %0t: reset = %b, ena = %b, d = %b, q = %b",
             $time, reset, ena, d, q);
end
endmodule

```

```

Time = 0: reset = 1, ena = 0, d = 0, q = 0
Time = 10000: reset = 0, ena = 0, d = 0, q = 0
Time = 20000: reset = 0, ena = 1, d = 0, q = 0
Time = 30000: reset = 0, ena = 1, d = 1, q = 1
Time = 40000: reset = 0, ena = 1, d = 0, q = 0
Time = 50000: reset = 0, ena = 0, d = 0, q = 0
Time = 60000: reset = 0, ena = 0, d = 1, q = 0
Time = 70000: reset = 0, ena = 0, d = 0, q = 0
Time = 80000: reset = 0, ena = 1, d = 0, q = 0
Time = 90000: reset = 0, ena = 1, d = 1, q = 1
Time = 100000: reset = 1, ena = 1, d = 1, q = 0
Time = 110000: reset = 0, ena = 1, d = 1, q = 1
dLatch_tb.v:50: $finish called at 130000 (1ps)

```

Esercizio 4 - FSM

▼ Creatore originale: @Gianbattista Busonera



Si rimanda link alla generica realizzazione di una FSM: [LINK](#)

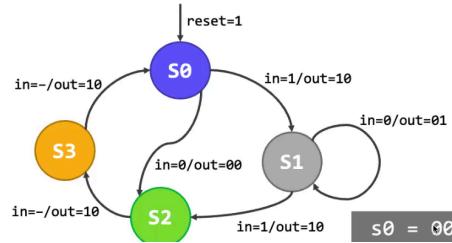
Si voglia descrivere in verilog una macchina a stati con quattro stati codificati come segue:

s0	00
s1	01
s2	10
s3	11

Si riporta inoltre l'FSM da realizzare:

La macchina a stati presenta un reset sincrono attivo alto, due bit di uscita (out) e un bit di ingresso (in).

Si noti che si tratta di una FSM di Mealy in quanto l'uscita non dipende solamente dallo stato corrente ma anche dal valore corrente dell'ingresso.



▼ Soluzione

```

module es4(out, clk, reset, in);
    output reg [1:0] out; // serve farla di tipo reg in quanto la utilizzeremo
    // nei blocchi always!
    input clk, reset, in;
    // SOLO PER COMODITA' decidiamo di utilizzare dei parametri costanti per descrivere
    // gli stati:
    parameter s0 = 2'b00,
              s1 = 2'b01,
              s2 = 2'b10,
              s3 = 2'b11;
    // Descriviamo ora la memoria (FF) che contiene lo stato corrente.
    reg [1:0] cs; // current state
    // è però necessario anche utilizzare una variabile d'appoggio che descriva lo stato
    // futuro, di tipo reg poichè finirà nel blocco always
    reg [1:0] ns; // next state

```

```

// BLOCCO ALWAYS CHE DESCRIVO LO STATO CORRENTE che cambia solo su fronte del clock!
always @(posedge clk) begin
    if(reset) cs = s0; // reset sincrono con priorità allo stato 00
    else cs = ns; // se non devo fare reset passo al next state sul fronte del clock.
end

// BLOCCO ALWAYS CHE CALCOLA IL NEXT STATE
// si ricordi che lo stato futuro dipende solo dal valore degli ingressi e dallo
// stato corrente
always @(in or cs) begin
    case(cs)
        s0: begin
            if(in) ns = s1;
            else ns = s2;
        end
        s1: begin
            if(in) ns = s2;
            else ns = s1;
        end
        s2: ns = s3; // indipendentemente dall'ingresso
        s3: ns = s0; // indipendentemente dall'ingresso
    endcase
end

// BLOCCO ALWAYS CHE DESCRIVE L'USCITA
// Si ricordi che l'uscita dipende solo dal valore corrente dell'ingresso
// e dallo stato corrente.
always @(in or cs) begin
    case(cs)
        s0: begin
            if(in) out = 2'b10;
            else out = 2'b00;
        end
        s1: begin
            if(in) out = 2'b10;
            else out = 2'b01;
        end
        s2: out = 2'b10; // indipendentemente dall'ingresso
        s3: out = 2'b10; // indipendentemente dall'ingresso
    endcase
end
endmodule

```

Potenzialmente potevamo unire le uscite e gli stati futuri nello stesso blocco always ma, per maggiore chiarezza, ho preferito separarli.



Se anche uno degli stati non fosse previsto, converrebbe inserire una condizione di default nel codice così da coprire in ogni caso tutti i casi!

▼ Testbench

Si riporta il codice dichiaratamente generato per il testbench:

```

`timescale 1ns / 1ps

module tb_es4;

// Segnali
reg clk;
reg reset;
reg in;
wire [1:0] out;

// Istanziazione del modulo
es4 uut (
    .out(out),
    .clk(clk),
    .reset(reset),
    .in(in)
);

// Stato corrente interno (accesso diretto per debug)
wire [1:0] state = uut.cs;

// Clock con periodo 10ns
always #5 clk = ~clk;

// Monitoraggio ogni 5ns
always #5 begin
    #0.1 // leggero ritardo per garantire aggiornamento variabili
    $display("t=%0dns | clk=%b | reset=%b | in=%b | state=%02b | out=%02b",
        $time, clk, reset, in, state, out);
end

// Sequenza di test sincrona
initial begin
    // Inizializzazione
    clk = 0;
    reset = 1;
    in = 0;

    #10;      // t = 10ns
    reset = 0;

    // Test transizioni con input cambiati tra i fronti
    in = 1;   // S0 → S1
    @(posedge clk);

    in = 1;   // S1 → S2
    @(posedge clk);

    in = 0;   // S2 → S3
    @(posedge clk);

    in = 1;   // S3 → S0
    @(posedge clk);

```

```

in = 0; // S0 → S2
@(posedge clk);

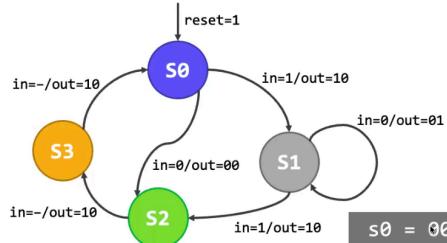
@(posedge clk); // S2 → S3

$display("Fine test.");
$finish;
end

endmodule

```

Che da il seguente output:



t=5ns	clk=1	reset=1	in=0	state=xx	out=xx
t=10ns	clk=0	reset=0	in=1	state=00	out=00
t=15ns	clk=1	reset=0	in=1	state=00	out=10
t=20ns	clk=0	reset=0	in=1	state=01	out=10
t=25ns	clk=1	reset=0	in=1	state=01	out=10
t=30ns	clk=0	reset=0	in=0	state=10	out=10
t=35ns	clk=1	reset=0	in=0	state=10	out=10
t=40ns	clk=0	reset=0	in=1	state=11	out=10
t=45ns	clk=1	reset=0	in=1	state=11	out=10
t=50ns	clk=0	reset=0	in=0	state=00	out=00
t=55ns	clk=1	reset=0	in=0	state=00	out=00
t=60ns	clk=0	reset=0	in=0	state=10	out=10
t=65ns	clk=1	reset=0	in=0	state=10	out=10

Tempo	clk	reset	in	Stato	Out	<input checked="" type="checkbox"/> Stato atteso	<input checked="" type="checkbox"/> Uscita attesa
5ns	1	1	0	00	00	S0 (da reset)	00
10ns	0	0	1	00	10	S0	10
15ns	1	0	1	01	10	S1	10
20ns	0	0	1	01	10	S1	10
26ns	1	0	0	10	10	S2	10
31ns	0	0	0	10	10	S2	10
36ns	1	0	1	11	10	S3	10
41ns	0	0	1	11	10	S3	10
46ns	1	0	0	00	00	S3 → S0	00
51ns	0	0	0	00	00	S0	00
56ns	1	0	0	10	10	S0 + in=0 ⇒ S2	10
61ns	0	0	0	10	10	S2	10

▼ GTKWave e visualizzazione forme d'onda

Per visualizzare le forme d'onda è necessario inserire il seguente frammento di codice al testbench precedente e ricompilare:

```
initial begin
    $dumpfile("waveform.vcd"); // Nome del file da creare
    $dumpvars(0, es04); // Dump di tutte le variabili del modulo "es04"
end
```

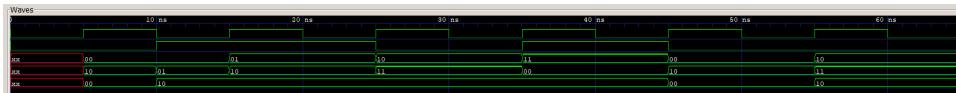
E dovremmo ottenere, dopo la ricompilazione e riesecuzione il seguente output:

```
PS C:\Users\gianb\Desktop\UNI\ELAP\Lab\E05\ES04> iverilog -o es04_tb es04.v es04_tb.v
PS C:\Users\gianb\Desktop\UNI\ELAP\Lab\E05\ES04> vvp es04_tb
VCD info: dumpfile waveform.vcd opened for output.
t=5ns | clk=1 | reset=1 | in=0 | state=00 | out=00
t=10ns | clk=0 | reset=0 | in=1 | state=00 | out=10
t=15ns | clk=1 | reset=0 | in=1 | state=01 | out=10
t=20ns | clk=0 | reset=0 | in=1 | state=01 | out=10
t=26ns | clk=1 | reset=0 | in=0 | state=10 | out=10
t=31ns | clk=0 | reset=0 | in=0 | state=10 | out=10
t=36ns | clk=1 | reset=0 | in=1 | state=11 | out=10
t=41ns | clk=0 | reset=0 | in=1 | state=11 | out=10
t=46ns | clk=1 | reset=0 | in=0 | state=00 | out=00
t=51ns | clk=0 | reset=0 | in=0 | state=00 | out=00
t=56ns | clk=1 | reset=0 | in=0 | state=10 | out=10
t=61ns | clk=0 | reset=0 | in=0 | state=10 | out=10
```

Scrivendo dunque su console:

```
gtkwave waveform.vcd
```

Si aprirà il programma gtkwave e, inserendo i segnali d'interesse nel grafico vediamo:



Esercizio 5 - Shift-Register completo

Si richiede di implementare in Verilog un componente adatto a realizzare la funzionalità di uno shift-register parametrico completo (con uscita q) con reset asincrono tale per cui:

- ctrl = 0 → nessuna uscita modificata
- ctrl = 1 → shift a destra, data[N-1] entra come MSB
- ctrl = 2 → shift a sinistra, data[0] entra come LSB
- ctrl = 3 → caricamento parallelo di data[N-1:0]

▼ Soluzione

```
module shift_register(q, clk, reset, ctrl, data);
    parameter N = 8;
    output [N-1:0] q;
    input clk, reset;
    input [1:0] ctrl;
    input [N-1:0] data;

    reg [N-1:0] mem; // DEFINISCO LA MEMORIA DEL MIO CIRCUITO
```

```

always @(posedge clk or posedge reset) begin
    if(reset)
        mem = 0;
    else begin
        case (ctrl)
            2'b00 : begin
                // no operation
            end
            2'b01 : begin // SHIFT A DESTRA
                mem = {data[N-1], mem[N-1:1]}; // perdo l'ultimo bit e inserisco data[N-1] come
                // MSB come da specifica
            end
            2'b10 : begin // SHIFT A SINISTRA
                mem = {mem[N-2:0], data[0]}; // perdo il primo bit e inserisco data[0] come LSB
            end
            2'b11 : begin
                mem = data;
            end
        endcase
    end
end

assign q = mem; // la mia uscita prende il valore della memoria.
endmodule

```

Esercizi extra

Non farò gli esercizi aggiuntivi perchè sono bene o male già stati trattati ([FF e ALU](#)) nella sezione di teoria relativa al linguaggio verilog e sono anche più semplici degli esercizi in precedenza.

Esercitazione 6 - Logica programmabile e memorie a semiconduttore

[Esercizio 1 - Logica programmabile 1](#)

[Considerazioni iniziali](#)

[Soluzione](#)

[Esercizio 2 - Logica programmabile 2](#)

[Obiettivo 1 - Registro SIPO](#)

[Considerazioni iniziali](#)

[Soluzione](#)

[Obiettivo 2 - Registro PIPO](#)

[Considerazioni iniziali](#)

[Soluzione](#)

[Esercizio 3 - Decoder indirizzi e ritardo wordline](#)

[Obiettivo 1 - 64 celle](#)

[Commento preliminare](#)

[Ritardo Inverter 1](#)

[Ritardo NANDs](#)

[Ritardo Inverter 2](#)

[Ritardo Totale](#)

[Obiettivo 2 - 16 celle](#)

[Decoder banco](#)

[Wordline decoder](#)

[Ritardo totale](#)

[Esercizio 4 - Tensioni nella cella DRAM \(teorico\)](#)

[Soluzione](#)

[Esercizio 5 - Lettura in DRAM](#)

[Obiettivo 1 - Variazione di tensione a seguito di una lettura di un 1](#)

[Calcolo variazione di tensione di bitline](#)

[Calcolo variazione di tensione di storage](#)

[Obiettivo 2 - Variazione di tensione a seguito di una lettura di uno 0](#)

[Calcolo variazione di tensione di bitline](#)

[Calcolo variazione di tensione di storage](#)

Obiettivo 3 - Massimo numero M di celle connesse

Soluzione

Esercizio 6 - Rinfresco celle DRAM (todo)

Introduzione

Svolgimento

Esercizio 7 - Ritenzione in cella Flash

Introduzione

Soluzione

Esercizio 8 - Max numero di cicli P/E in celle Flash

Introduzione generale

Ripassino sulle FLASH e contesto

Obiettivo 1 - Caso senza wear levelling

Introduzione

Soluzione

Obiettivo 2 - Caso con wear levelling

Introduzione

Soluzione

Esercizio 1 - Logica programmabile 1

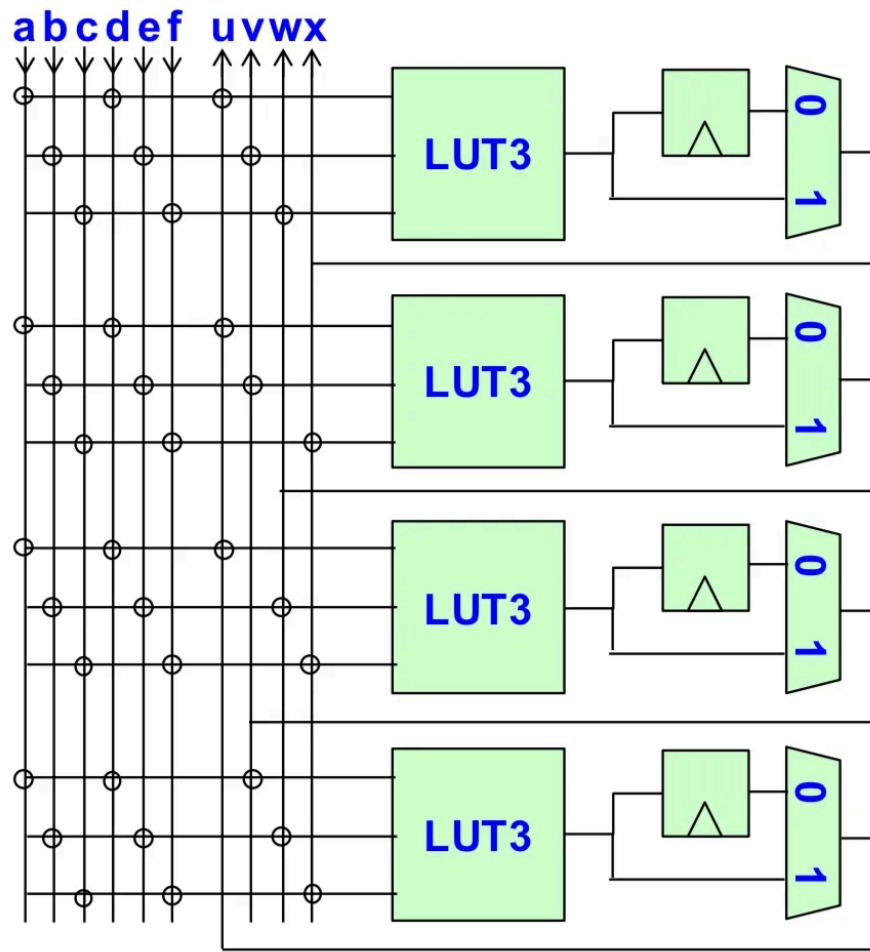
▼ Creatore originale: @Francesco Ambrosino

Determinare

- la programmazione delle connessioni
- la programmazione delle LUT
- la programmazione dei multiplexer

per realizzare le funzioni logiche:

- $u = abc + def$
- $v = (ab)^* = \overline{ab}$
- $w = def$
- $x = \text{NON USATO}$



▼ Considerazioni iniziali

1. a, b, c, d, e, f hanno freccia entrante quindi rappresentano gli **ingressi**.
2. u, v, w, x hanno freccia uscente quindi rappresentano le **uscite**.
3. $LUT3$ sta per lookup table a 3 ingressi (possiede quindi $2^3 = 8$ celle di memoria da 3 bit in cui possiamo memorizzare una qualsiasi funzione logica di questi 3 ingressi).
4. Su un cavo collegato alla LUT può o non essere attivo nessun ingresso/uscita o esserne attivo solo uno, non più di un pallino può essere riempito per ogni riga.
5. Ogni mux (multiplexer) all'uscita delle LUT è collegato ad una linea di una uscita (**l'uscita del mux comanda le uscite del circuito**). Seguendo infatti l'uscita dei mux si può vedere che:
 - a. La prima LUT comanda l'uscita **x**;

- b. La seconda LUT comanda l'uscita w;
 - c. La terza LUT comanda l'uscita v;
 - d. La quarta LUT comanda l'uscita u.
6. Leggendo le funzioni logiche da realizzare possiamo affermare che vogliamo realizzare un **circuito puramente combinatorio** (visto che le uscite dipendono solamente dagli ingressi), quindi non avremo bisogno di utilizzare i flip-flop (ne vedremo un esempio di utilizzo nel prossimo esercizio), per questo motivo i valori di tutti i mux saranno a 1 in questo esercizio (l'uscita delle LUT comanda DIRETTAMENTE le uscite).
- Non è infatti necessario sincronizzare l'uscita con il valore di un ipotetico clock (che comanderebbe anche il FF).

▼ Soluzione

1 Osserviamo in primis che $u = abc + def$ è **funzione di 6 ingressi**, non è quindi realizzabile con una sola LUT (poiché **ogni LUT ha massimo 3 ingressi**).

2 In secondo luogo notiamo che def , operando di u , definisce l'uscita w , possiamo dunque riutilizzarlo riducendo gli ingressi di "u": $u = abc + w$ che però è funzione di 4 ingressi e dunque non ancora realizzabile.

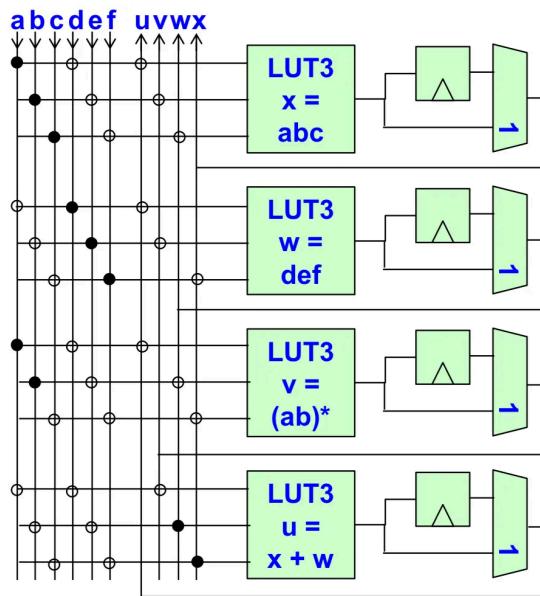
3 Possiamo utilizzare x , lasciato libero dalle specifiche del testo, e porlo uguale a abc .

Sistemiamo così la situazione per $u = x + w$ con $x = abc$ e $w = def$.

4 Infine guardando a $v = (ab)^*$ notiamo subito che non abbiamo problemi di implementazione in questo caso perché v è già funzione con meno di tre ingressi.

Nella tabella che segue ci sono:

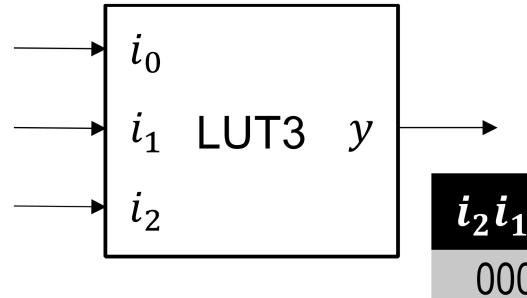
- connessioni definite (un pallino nero definisce una connessione riga/colonna),
- funzioni logiche esplicitate dentro le LUT (NOTA: le LUT contengono della logica combinatoria modificabile, non farti problemi su come le LUT capiscano che gli ingressi sia in OR, in AND, negati o affermati... c'è della logica dentro.),
- uscite dei mux tutte a 1 per quanto spiegato prima.



Volendo la LUT3(u) si poteva realizzare collegando gli ingressi c, v e w.

L'esercizio non è però finito qui poiché dobbiamo **implementare la programmazione** delle LUT.

Lo schema generale è



da questo scriveremo le **tabelle di verità** per le quattro LUT.

! La terza e la quarta LUT (che comandano rispettivamente le uscite v e u) hanno attivi solo 2 ingressi, **l'ingresso non attivo deve essere considerato un *don't care***

$i_2 i_1 i_0$	LUT3x	LUT3w	LUT3v	LUT3u
000	0	0	1	0
001	0	0	1	0
010	0	0	1	1
011	0	0	0	1
100	0	0	1	1
101	0	0	1	1
110	0	0	1	1
111	1	1	0	1

La tabella è di facile comprensione, l'unica attenzione da porre è quella nel **far corrispondere i_0 , i_1 e i_2 agli ingressi rispettivi delle LUT**. Ad esempio nella $LUT_3(v)$ i_0 corrisponde ad a , i_1 corrisponde a b e i_2 è libero, mentre invece nella $LUT_3(w)$ è i_0 l'ingresso libero (da interpretare come *don't care*).

Esercizio 2 - Logica programmabile 2

▼ Creatore originale: @Francesco Ambrosino

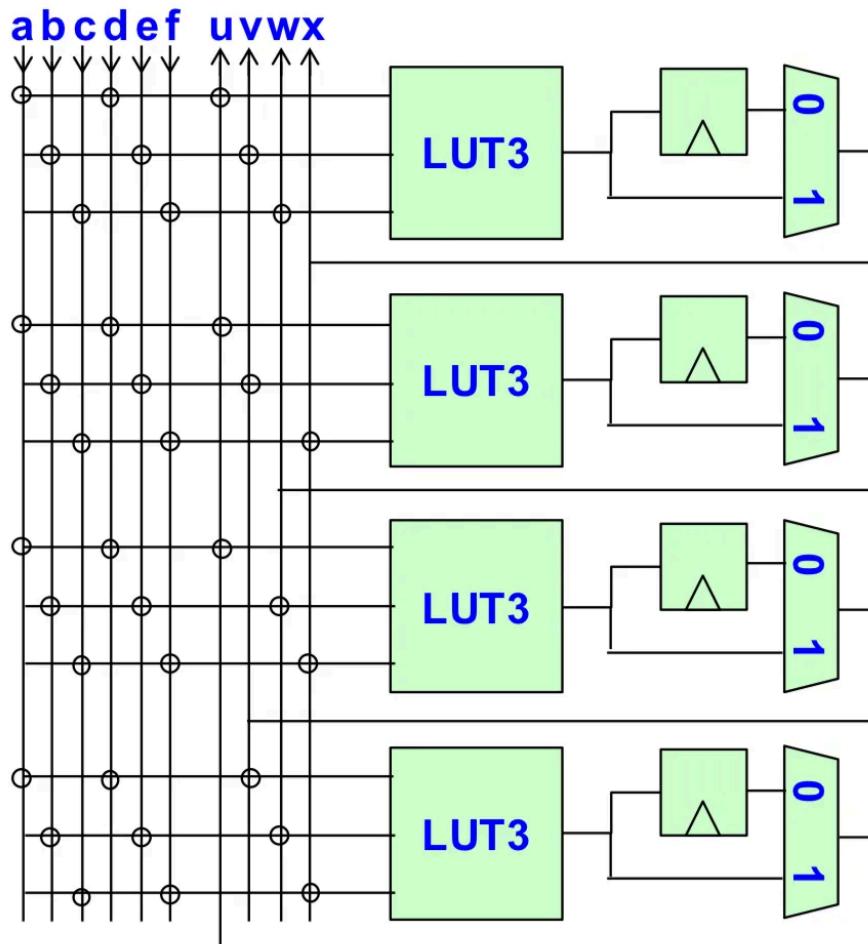
Obiettivo 1 - Registro SIPO

Determinare

- la programmazione delle connessioni
- la programmazione delle LUT
- la programmazione dei multiplexer

per realizzare le funzioni logiche:

Registro SIPO (Serial Input Parallel Output) a 4 bit con ingresso seriale $S_{in} = a$ e uscite $Q_{0,1,2,3} = (u, v, w, x)$.



Si noti che $S_{in} = a$ e che le uscite $Q_{0\dots3} = u, v, w, x$

▼ Considerazioni iniziali

UGUALI ALLO SCORSO ESERCIZIO:

1. a, b, c, d, e, f hanno freccia entrante quindi rappresentano gli ingressi.
2. u, v, w, x hanno freccia uscente quindi rappresentano le uscite.
3. $LUT3$ sta per lookup table a 3 ingressi infatti ad ogni LUT sono collegati 3 cavi.
4. Su un cavo collegato alla LUT può o non essere attivo nessun ingresso/uscita o esserne attivo solo uno, non più di un pallino può essere riempito per ogni riga.
5. Ogni mux (multiplexer) all'uscita delle LUT è collegato ad una linea di una uscita (l'uscita del mux comanda le uscite del circuito).

DIVERSA:

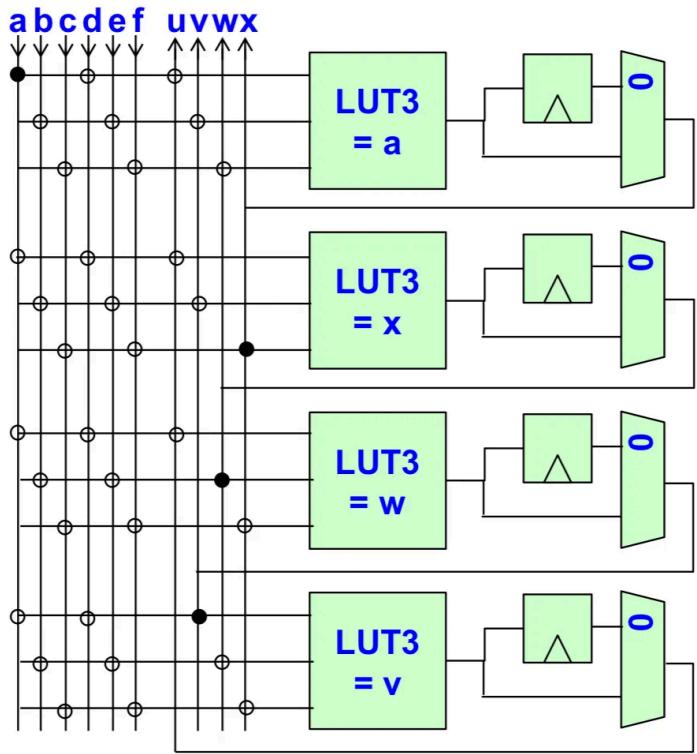
Dovendo implementare un registro sappiamo di star lavorando con un circuito **sequenziale**, dobbiamo dunque **utilizzare i flip-flop**.

Quanti flip-flop dobbiamo utilizzare? Registro a 4 bit → ci servono 4 flip-flop, per questo **i valori di tutti i mux saranno a 0 in questo esercizio** (l'uscita delle LUT NON comanda direttamente le uscite ma lo fa attraverso il flip-flop).

▼ Soluzione

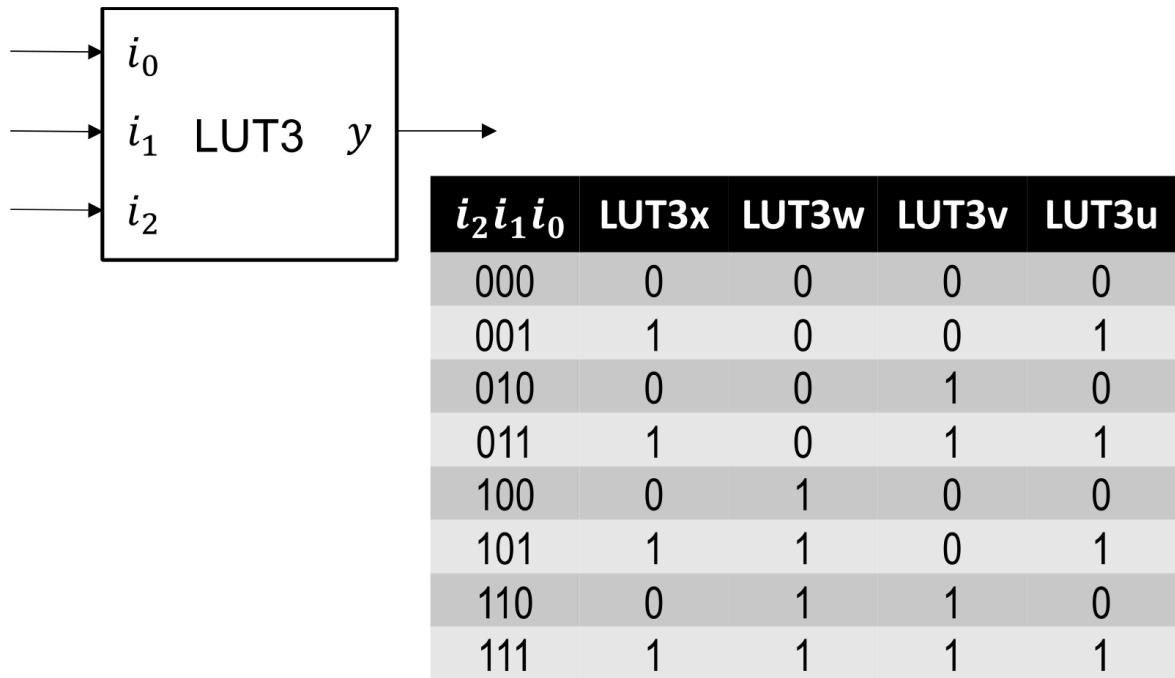
Dobbiamo realizzare un registro SIPO (banalmente uno shift register), cioè un circuito in cui:

- il 1° flip-flop riceve (attraverso la prima LUT) l'ingresso seriale a
- il 2° flip-flop riceve (attraverso la seconda LUT) l'uscita del 1° flip-flop
- il 3° flip-flop riceve (attraverso la terza LUT) l'uscita del 2° flip-flop
- il 4° flip-flop riceve (attraverso la quarta LUT) l'uscita del 3° flip-flop



In questo esercizio le LUT non implementano una vera e propria logica combinatoria, banalmente riportano all'ingresso del flip-flop l'ingresso che ricevono a loro volta.

Riporto di seguito le tabelle di verità.



Come per l'esercizio 1 l'unica difficoltà può essere nel capire la corrispondenza tra gli ingressi:

- $LUT_3(x) = a = i_0$
- $LUT_3(w) = x = i_2$
- $LUT_3(v) = w = i_1$
- $LUT_3(u) = v = i_0$

Obiettivo 2 - Registro PIPO

Determinare

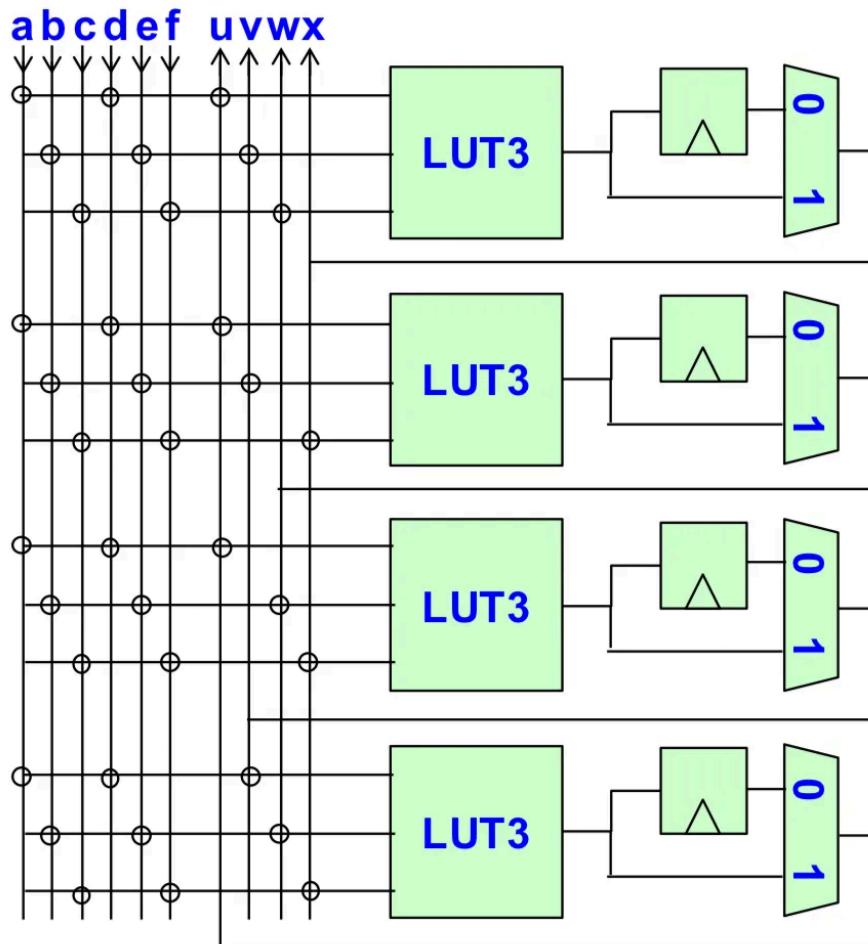
- la programmazione delle connessioni
- la programmazione delle LUT
- la programmazione dei multiplexer

per realizzare le funzioni logiche:

Registro PIPO (Parallel Input Parallel Output) a 3 bit con:

- ingresso parallelo (a, b, c)

- reset (e) attivo alto
- uscite $Q_{0,1,2} = (u, v, w)$.
 - L'uscita w corrisponde all'ingresso a ;
 - L'uscita v corrisponde all'ingresso b ;
 - L'uscita u corrisponde all'ingresso c .



▼ Considerazioni iniziali

UGUALI ALLO SCORSO OBIETTIVO:

1. a, b, c, d, e, f hanno freccia entrante quindi rappresentano gli ingressi.
2. u, v, w, x hanno freccia uscente quindi rappresentano le uscite.
3. $LUT3$ sta per lookup table a 3 ingressi infatti ad ogni LUT sono collegati 3 cavi.

4. Su un cavo collegato alla LUT può o non essere attivo nessun ingresso/uscita o esserne attivo solo uno, non più di un pallino può essere riempito per ogni riga.
5. Ogni mux (multiplexer) all'uscita delle LUT è collegato ad una linea di una uscita (l'uscita del mux comanda le uscite del circuito).

MODIFICATA:

Come prima, dovendo implementare un registro sappiamo di star lavorando con un circuito sequenziale, dobbiamo dunque utilizzare i flip-flop.

Quanti flip-flop dobbiamo utilizzare? Registro a 3 bit → ci servono 3 flip-flop.

La quarta LUT verrà utilizzata in modo asincrono (senza flip-flop) per gestire il reset.

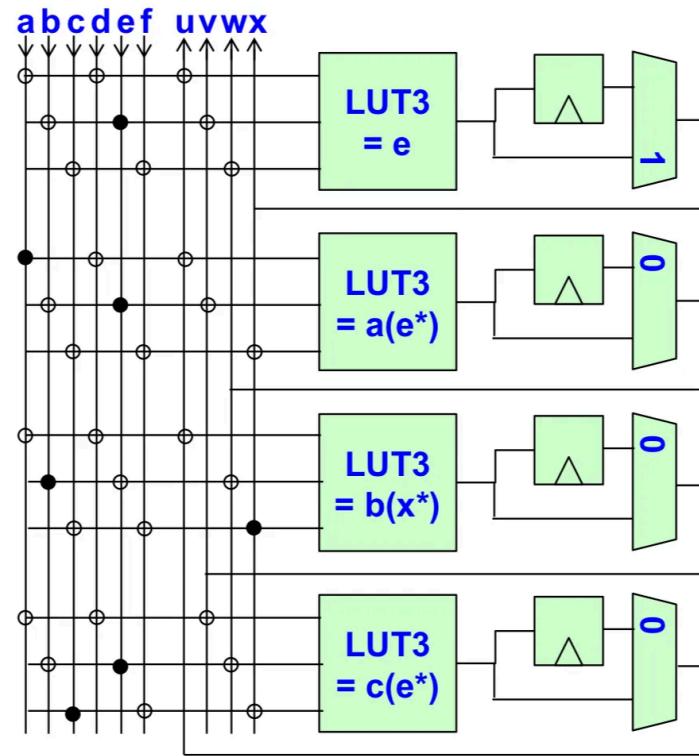
▼ Soluzione

Le specifiche di progetto specificano che le 3 uscite del registro debbano essere u, v, w , da questo deduciamo che le LUT che comandano quelle uscite (le ultime tre) debbano avere il valore del mux alle rispettive uscite collegato a 0 mentre la prima LUT, che comanda x , avrà in ingresso e e il mux che comanda la sua uscita a 1.

I 3 ingressi del registro parallelo sono a, b, c per cui avremo che:

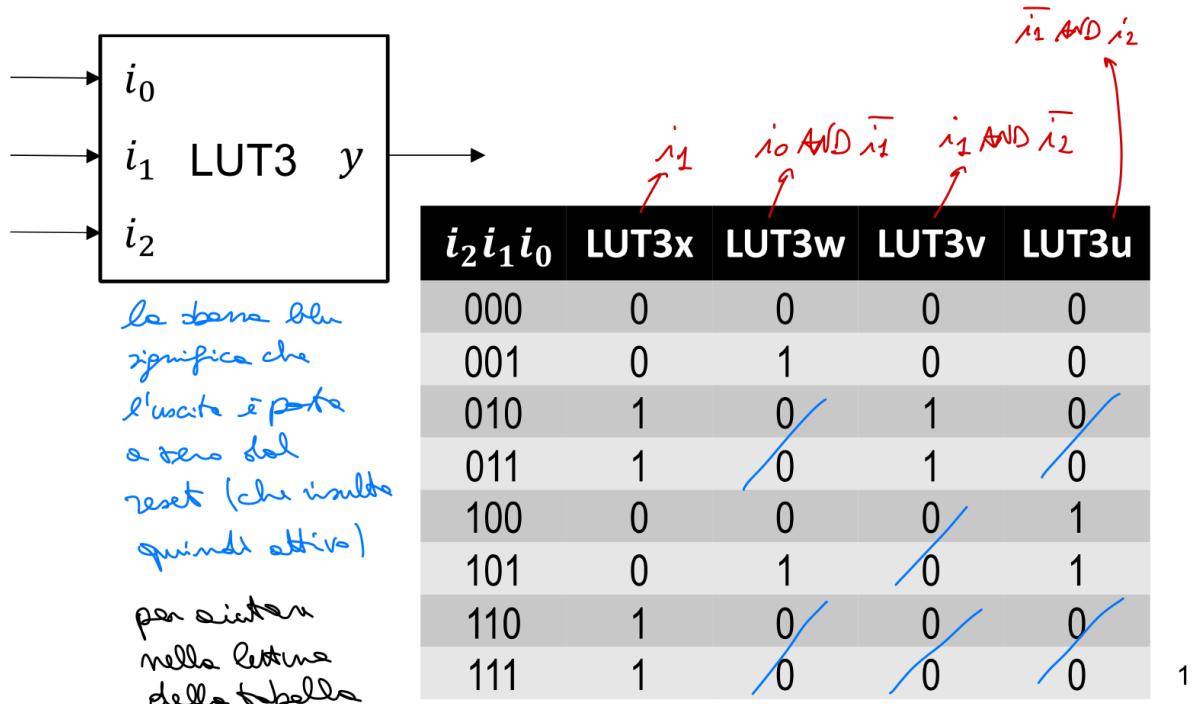
- il 2° flip-flop riceve (attraverso la seconda LUT) in ingresso a
- il 3° flip-flop riceve (attraverso la terza LUT) in ingresso b
- il 4° flip-flop riceve (attraverso la quarta LUT) in ingresso c

Questo però non basta perché dobbiamo collegare in ingresso a queste 3 LUT anche il **reset**. Per la seconda e la quarta LUT possiamo collegare e in maniera diretta, nell'architettura della terza non possiamo collegare e poiché si trova sulla stessa riga (sullo stesso cavo) di b che deve essere ingresso di questa LUT, possiamo risolvere questo problema semplicemente collegando x (uscita asincrona della prima LUT) a cui è collegato in ingresso e (si ha infatti $x = e$).



NOTA: l'ingresso di reset compare negato e in AND con l'ingresso parallelo in modo che quando è posto a 1, la sua negazione è 0 e quindi manda a 0 l'uscita delle LUT indipendentemente dai valori di a, b, c perché questi sarebbero in AND con valore logico 0.

Riporto di seguito le tabelle di verità.



1

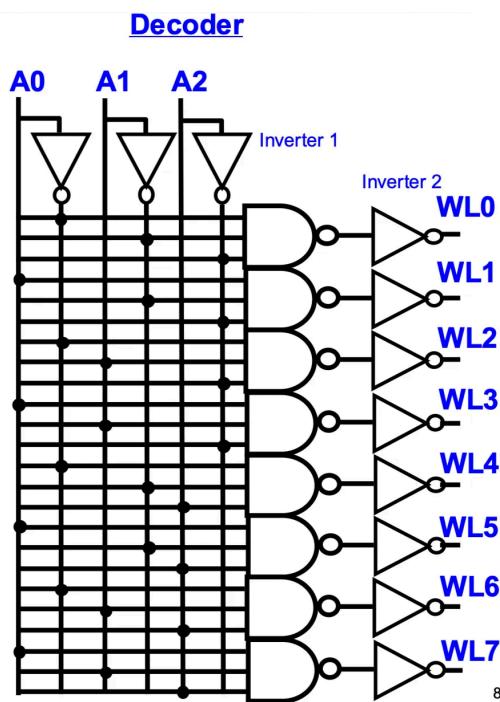
Esercizio 3 - Decoder indirizzi e ritardo wordline

▼ Creatore originale @Francesco Ambrosino

Obiettivo 1 - 64 celle

Si ha un banco di memoria DRAM con 8 wordlines connesse a $N = 64$ celle ciascuna. Il pass transistor della cella ha una capacità di gate di $0,1 \text{ fF}$ e la linea metallica di una wordline ha una resistenza complessiva di $10 \Omega \cdot N = 640 \Omega$.

- Progettare con porte CMOS il **decoder** che pilota le 8 wordlines a partire dai segnali di indirizzo.
- Sapendo che ogni transistore MOS usato nel decoder ha resistenza $R_{ON} = 100 \Omega$ e capacità di gate $C_g = 1 \text{ fF}$, determinare il massimo ritardo di attivazione delle wordlines.



Commenti al progetto:

- 8 wordlines \rightarrow 3 bit di indirizzo (A_0, A_1, A_2)
- Ogni bit di indirizzo viene preso sia affermato che negato (con gli Inverter 1)
- Ogni porta NAND ha 3 ingressi corrispondenti ai 3 bit di ingresso (affermati o negati)
- Alle uscite della NAND c'è un inverter (chiamato Inverter 2) così che i valori delle wordline corrispondono all'**AND logico dei 3 bit in ingresso** (NAND in serie con NOT da AND).
- Come visto negli altri esercizi il pallino pieno corrisponde alla connessione riga/colonna.

▼ Commento preliminare

La formula che si utilizzerà per il calcolo dei ritardi è:

$$t = 0.69\tau = 0.69 \cdot R_{\text{eq}} \cdot C_{\text{carico}}$$

Perché?

[LINK](#) teoria

[LINK](#) esercitazione 3

Il massimo ritardo di propagazione in una wordline è dato dall'attraversamento di un Inverter 1 + NAND + inverter 2.

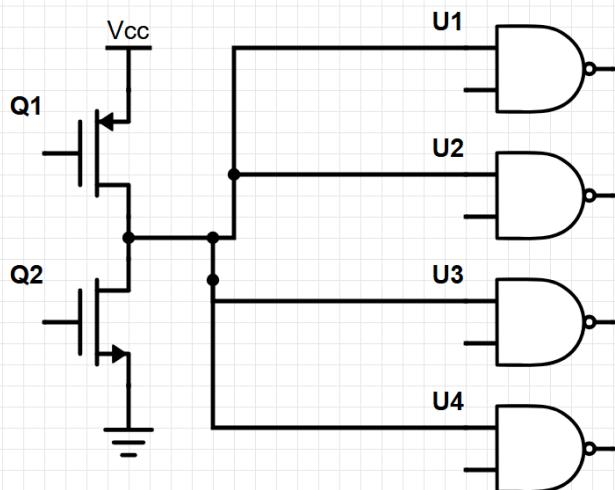


Vogliamo valutare il massimo ritardo di attivazione delle wordlines.

Attivare una wordline significa avere una "inverter 2" che commuta da uno stato basso a uno stato alto e, pertanto, ricerchiamo il tempo di propagazione di:

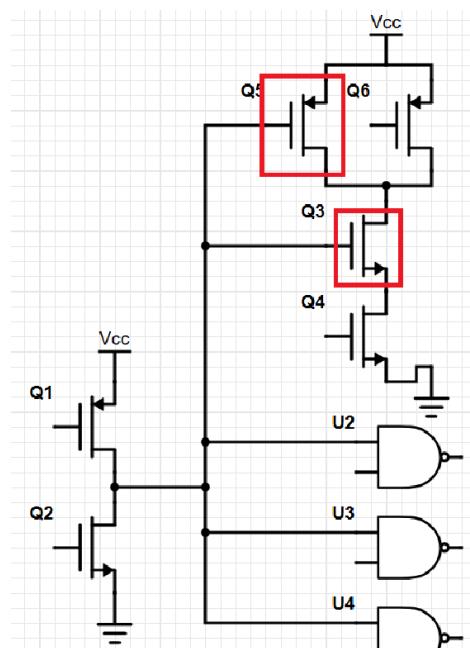
- Inverter 1: L→H (si noti che ogni inverter 1 ha un carico pari a 4 porte NAND)
- NAND: H→L
- Inverter 2: L→H (così la linea è attiva)

▼ Ritardo Inverter 1



Ogni Inverter 1 (a sinistra) è collegato a 4 porte NAND (sarebbero da tre ingressi ma è poco influente per la comprensione) che danno ognuna $2C_g$ di carico (la singola uscita dell'inverter è collegata a 2 MOS per ogni porta NAND come in foto di fianco). Quindi la capacità di carico dell'Inverter 1 è pari a $C_{carico} = 8C_g$.

Nel dettaglio, per ogni NAND, si ha una situazione del tipo:



Per ogni NAND (da 2 ingressi anziché tre per facilitare il disegno) l'inverter "pilota" due MOS e, quindi, un carico di $2C_g$ ognuna.

Come si può leggere dalla caption della prima immagine risulta dunque $C_{carico} = 8C_g$.

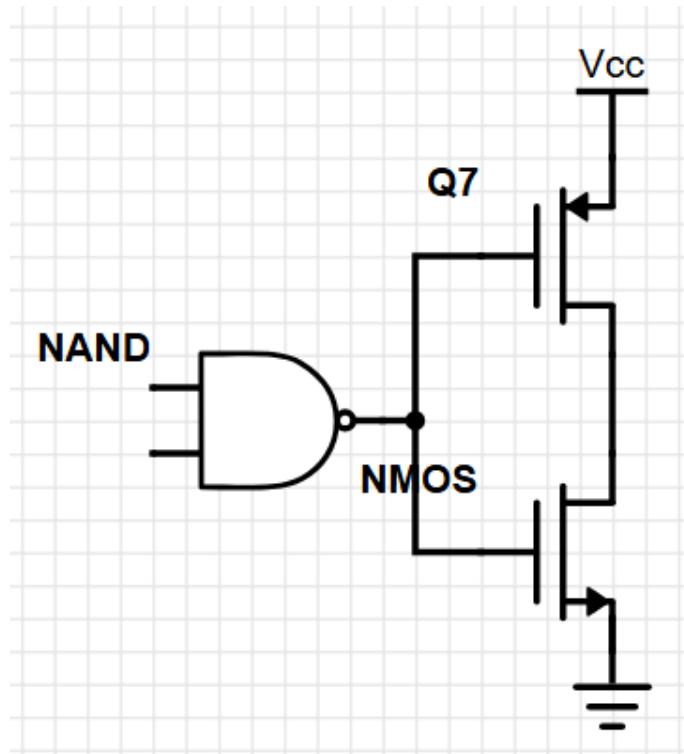
La sua R_{eq} sappiamo essere uguale proprio a R_{ON} , visto che sia nel caso di tempo di propagazione da L→H (si guarda la PUN), sia nel caso di tempo di propagazione H→L (si guarda la PDN) si ha a che fare con un solo MOS e quindi con una sola R_{ON} .

$$t = 0,69(R_{ON} \cdot 4 \cdot 2C_g) = 0,69 \cdot 100 \Omega \cdot 4 \cdot (2 \cdot 1 \text{ fF}) = 0,55 \text{ ps}$$

- Fattore “4”: ciascun output di Inverter 1 è collegato a 4 inputs di porta NAND
- Fattore “2”: ciascun input NAND è collegato a un gate PMOS ($1 C_g$) e a un gate NMOS ($1 C_g$), in parallelo

▼ Ritardo NANDs

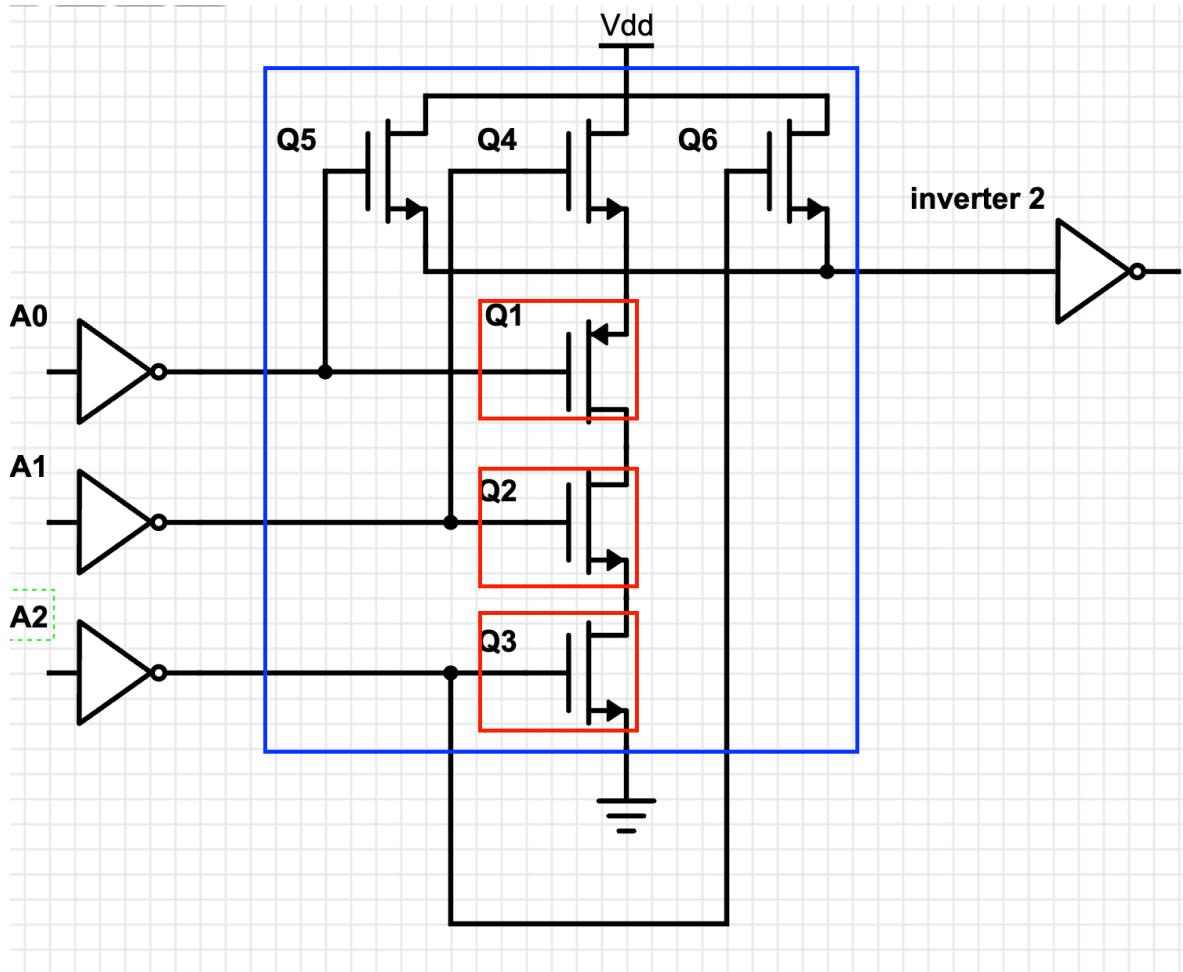
Ogni NAND è collegata a 1 inverter che dà $C_{carico} = 2C_g$ (come prima data dai transistori in parallelo collegati all’ingresso).



In rosso i 2 MOS dell’inveter 2 ciascuno di capacità C_g . Si noti che la porta NAND sarebbe da tre ingressi anzichè 2.

La sua R_{eq} può assumere diversi valori a seconda del numero di MOS attivi al suo ingresso (a seconda che si consideri una propagazione da alto a basso o da basso a alto). Volendo noi calcolare il ritardo massimo prenderemo la situazione di caso

peggiore ovvero quando ci sono 3 MOS (uno per ogni linea) attivi in serie (quindi propagazione H \rightarrow L) della porta NAND. Segue disegno esplicativo della nostra situazione.



Il riquadro blu evidenzia la porta NAND a 3 ingressi rappresentata in porte MOS.
In rosso sono evidenziati i 3 MOS in serie ciascuno con $R=R_{on}$ che danno complessivamente contributo di $3R_{on}=R_{eq}$ di caso peggiore.

Nota: i tre ingressi sono collegati a 3 inverter, per attivare gli nMOS bisogna avere 1 in uscita e quindi 0 in ingresso,
il fulcro del disegno rimane capire perché $R_{eq}=3R_{on}$.

$$0,69(3R_{on} \cdot 2C_g) = 0,41 \text{ ps}$$

- Fattore “3”: nel caso peggiore, l’uscita è collegata a GND con 3 MOS in serie

▼ Ritardo Inverter 2

Ogni Inverter 2 è collegato ad una wordline che è formata da 64 celle ognuna con $C_{g, \text{cella}} = 0,1 \text{ fF}$ (dato del testo). Questo significa che la $C_{\text{carico}} = 64C_{g, \text{cella}}$.

La R_{eq} in questo caso deve tenere conto della resistenza dell'Inverter R_{ON} e della resistenza della wordline (non ideale) $R_{\text{WL}} = 10 \Omega \cdot 64 \text{ celle} = 640 \Omega$ (dato del testo). Queste sono collegate in serie e ci danno quindi una $R_{\text{eq}} = R_{\text{ON}} + R_{\text{WL}}$.

$$\begin{aligned} & 0,69(R_{\text{ON}} + R_{\text{WL}})C_{\text{WL}} \\ & = 0,69(100 \Omega + 64 \cdot 10 \Omega) \cdot (64 \cdot 0,1 \text{ fF}) = 3,27 \text{ ps} \end{aligned}$$

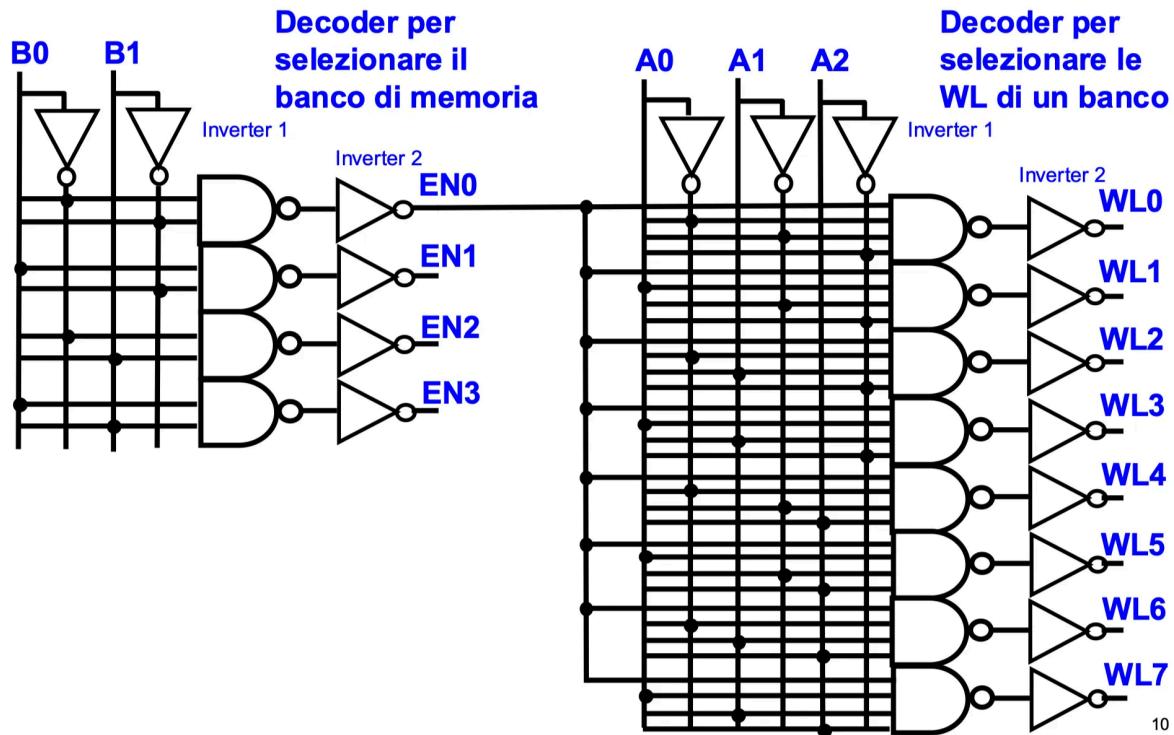
▼ Ritardo Totale

Sommo i tre valori parziali calcolati e ottengo:

$$0,55 \text{ ps} + 0,41 \text{ ps} + 3,27 \text{ ps} = 4,23 \text{ ps}$$

Obiettivo 2 - 16 celle

Ripetere il progetto e il calcolo del ritardo se la memoria è partizionata in $B = 4$ banchi ciascuno con 8 wordlines connesse a $N = \frac{64}{B} = 16$ celle.



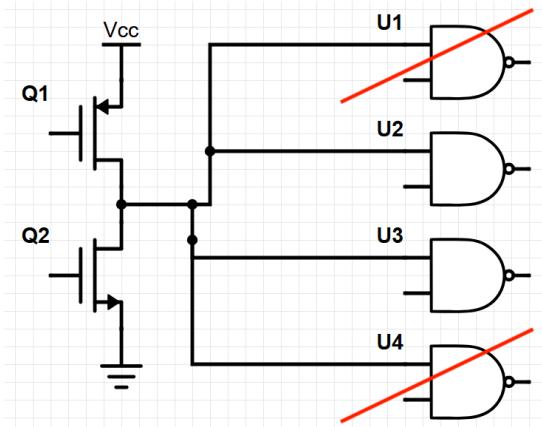
Commenti al progetto:

- 4 banchi \rightarrow 2 bit di indirizzo (B_0, B_1)
- Ogni bit di indirizzo viene preso sia affermato che negato (con gli Inverter 1 del banco)
- Ogni porta NAND del banco ha 2 ingressi corrispondenti ai 2 bit di ingresso (affermati o negati)
- Alle uscite della NAND del banco c'è un inverter (chiamato Inverter 2 del banco) così che i valori delle wordline corrispondano all'AND logico dei 2 bit in ingresso, questo funge da vero e proprio abilitatore/attivatore del rispettivo decoder di linea.
- Ogni porta NAND della wordline ha ora infatti 4 ingressi corrispondenti ai suoi 3 bit di ingresso (affermati o negati) + l'ingresso relativo all'inverter 2 di banco.
- Come visto negli altri esercizi il pallino pieno corrisponde alla connessione riga/colonna.
- Ovviamente il disegno è incompleto poiché anche a EN_1, EN_2, EN_3 è collegato un decoder di linea identico a quello visibile nel disegno collegato a EN_0 .

Dividiamo la trattazione in:

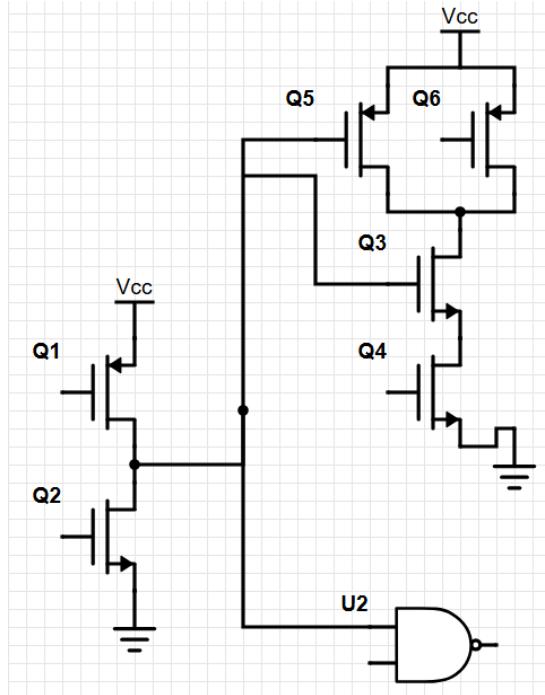
▼ Decoder banco

- Ritardo Inverter 1



Ogni Inverter 1 (a sinistra) è collegato a 2 porte NAND che danno ognuna $2C_g$ di carico (la singola uscita dell'inverter è collegata a 2 MOS per ogni porta NAND). Quindi la capacità di carico dell'Inverter 1 è pari a $C_{carico} = 4C_g$.

Nota: l'immagine è ripresa dall'obiettivo precedente con lo scopo di far capire che la situazione è la medesima con un numero di porte logiche (NAND in questo caso) inferiore.



Rappresentazione più pulita della situazione in esame.

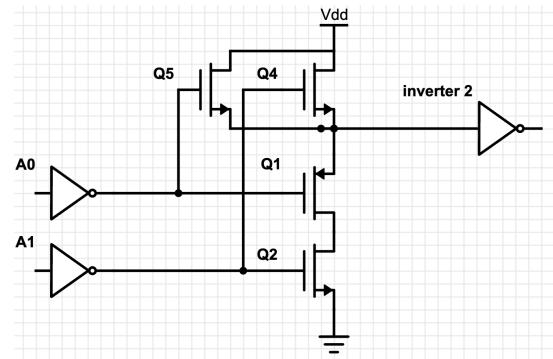
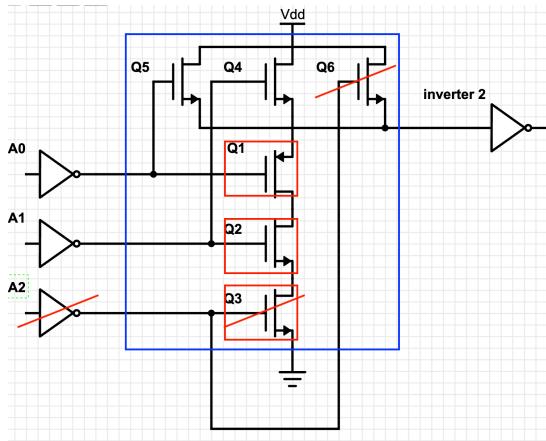
Ogni Inverter 1 è collegato a 2 porte NAND che danno ognuna $2C_g$ di carico (data dai transistori in parallelo collegati all'ingresso). Quindi la capacità di carico dell'Inverter 1 è pari a $C_{carico} = 4C_g$.

La sua R_{eq} sappiamo essere uguale proprio a R_{ON} essendo un inverter.

- Ritardo NANDs

Ogni NAND è collegata a 1 inverter che dà $C_{carico} = 2C_g$ (come prima data dai transistori in parallelo collegati all'ingresso).

La sua R_{eq} può assumere diversi valori a seconda del numero di MOS attivi al suo ingresso. Volendo noi calcolare il **ritardo massimo** prenderemo la situazione di caso peggiore ovvero quando ci sono 2 MOS (uno per ogni linea) attivi in serie e quindi $R_{eq} = 2R_{ON}$.



Rappresentazione più pulita della situazione in esame.

Come prima il riquadro blu evidenzia la porta NAND a 3 2 ingressi rappresentata in porte MOS e in rosso sono evidenziati i 3 2 MOS in serie ciascuno con $R=R_{ON}$ che danno complessivamente contributo di $2R_{ON}=Req$ di caso peggiore.

- **Ritardo Inverter 2**

Ogni Inverter 2 è collegato alle 8 NAND del wordline decoder (le abilita/disabilita tutte). Questo significa che la $C_{carico} = 8 \cdot 2C_g$, ricordando che le porte NAND che danno ognuna $2C_g$ di carico.

La R_{eq} in questo caso è proprio solo quella dell'inverter, quindi pari a R_{ON} .

Di seguito sono riportate formule e valori di quanto detto fino ad ora a parole

- **Inverter 1:** $0,69 \cdot [R_{ON} \cdot 2 \cdot (2C_g)] = 0,69 \cdot [100 \Omega \cdot 2 \cdot (2 \cdot 1 fF)] = 0,28 \text{ ps}$
- **NAND:** $0,69 \cdot (2R_{ON} \cdot 2C_g) = 0,69 \cdot (2 \cdot 100 \Omega \cdot 2 \cdot 1 fF) = 0,28 \text{ ps}$
- **Inverter 2:** $0,69 \cdot [R_{ON} \cdot 8 \cdot (2C_g)] = 0,69 \cdot [100 \Omega \cdot 8 \cdot (2 \cdot 1 fF)] = 1,1 \text{ ps}$

▼ Wordline decoder

- **Ritardo Inverter 1 (non è cambiato nulla)**

Ogni Inverter 1 è collegato a 4 porte NAND che danno ognuna $2C_g$ di carico (data dai transistori in parallelo collegati all'ingresso). Quindi la capacità di carico dell'Inverter 1 è pari a $C_{carico} = 8C_g$.

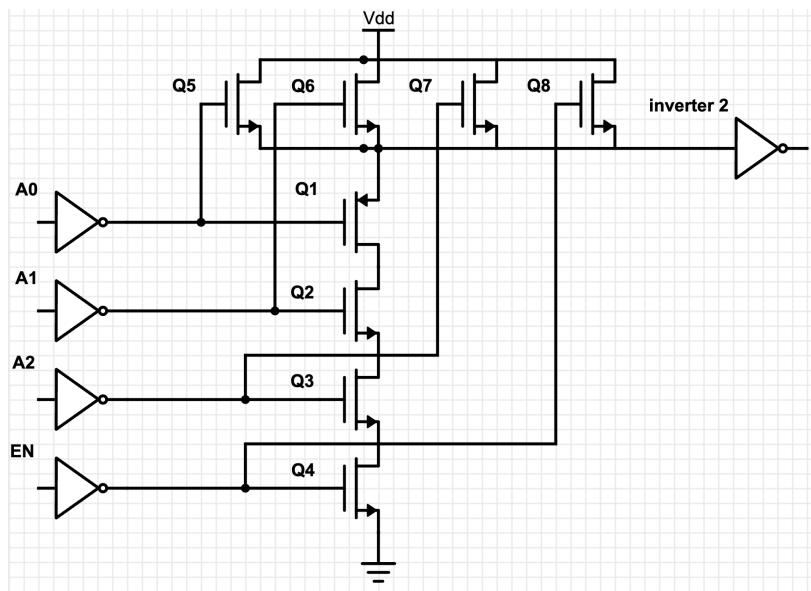
La sua R_{eq} sappiamo essere uguale proprio a R_{ON} .

- Ritardo NANDs

Ogni NAND è collegata a 1 inverter che dà $C_{\text{carico}} = 2C_g$ (come prima data dai transistori in parallelo collegati all'ingresso).

Visto che stiamo parlando di una NAND a 4 ingressi (3 dati da A0, A1, A2 e uno dato dal segnale di EN), il caso peggiore si ha quando il segnale attraverso la NAND si propaga da uno stato alto a uno basso ($H \rightarrow L$), in tal senso vanno considerati i 4 NMOS in serie e, pertanto, assumono $R_{eq} = 4R_{ON}$. E' inoltre necessario considerare tale stato affinchè dopo l'inverter 2 commuti da $L \rightarrow H$.

La rappresentazione è analoga a quelle già viste con l'unica differenza che essendo una porta NAND a 4 ingressi avrà 4 nMOS collegati in serie a GND che sono pilotati come detto dalle 3 linee di dato e dal segnale di enable (EN) che arriva dall'inverter 2 del decoder di banco.



Come dovrebbe essere chiaro le 4 resistenze che in serie danno $Req=4Ron$ sono Q1, Q2, Q3 e Q4.

- Ritardo Inverter 2

Ogni Inverter 2 è collegato ad una wordline che è a sua volta collegata a 16 celle ognuna con $C_{g, \text{cella}} = 0.1 \text{ fF}$ (dato del testo). Questo significa che la $C_{\text{carico}} = 16C_{g, \text{cella}}$.

La R_{eq} in questo caso deve tenere conto della resistenza dell'Inverter R_{ON} e della resistenza della wordline $R_{WL} = 10 \Omega \cdot 16 \text{ celle} = 160 \Omega$ (dato del testo). Queste sono collegate in serie e ci danno quindi una $R_{eq} = R_{ON} + R_{WL}$.

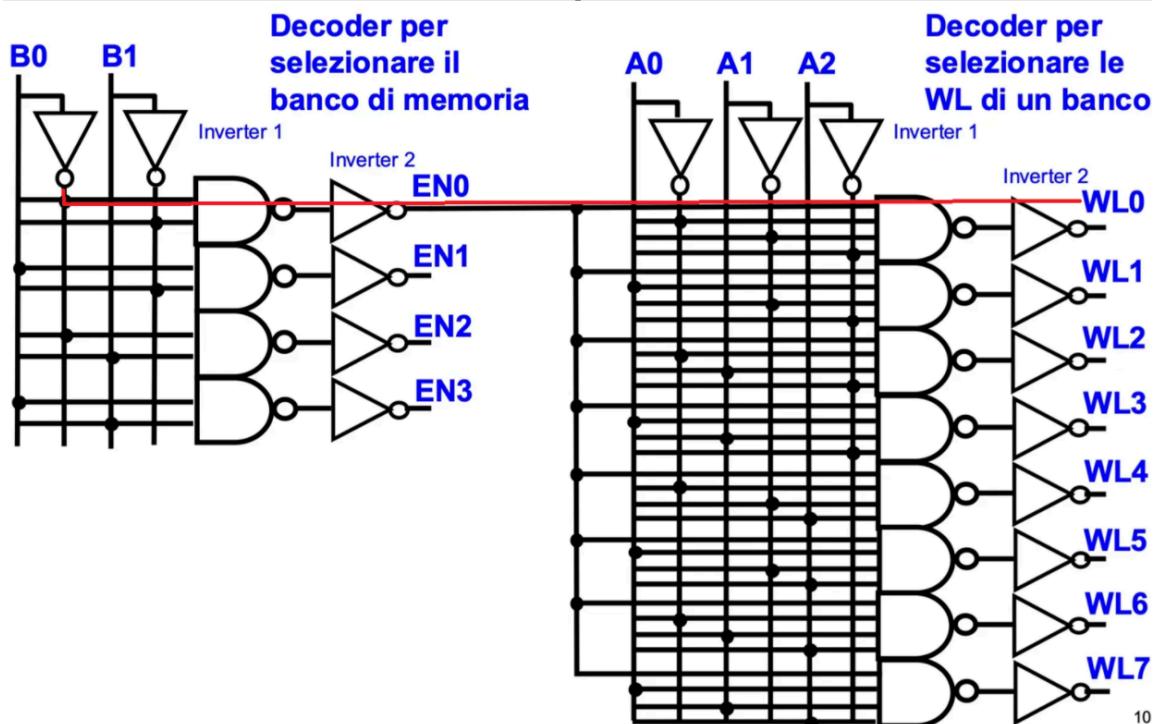
Di seguito sono riportate formule e valori di quanto detto fin ora a parole

- **Inverter 1:** $0,69 \cdot (R_{ON} \cdot 4 \cdot 2C_g) = 0,69 \cdot 100 \Omega \cdot 4 \cdot 2 \cdot 1 \text{ fF} = 0,55 \text{ ps}$
 - Fattore "4": l'output di ciascun Inverter 1 è collegato a 4 inputs porta NAND
- **NAND:** $0,69 \cdot (4R_{ON} \cdot 2C_g) = 0,69 \cdot (4 \cdot 100 \Omega \cdot 2 \cdot 1 \text{ fF}) = 0,55 \text{ ps}$
- **Inverter 2:** $0,69 \cdot (R_{ON} + R_{WL}) \cdot C_{WL}$
 $= 0,69 \cdot (100 \Omega + 16 \cdot 10 \Omega) \cdot 16 \cdot 0,1 \text{ fF} = 0,29 \text{ ps}$

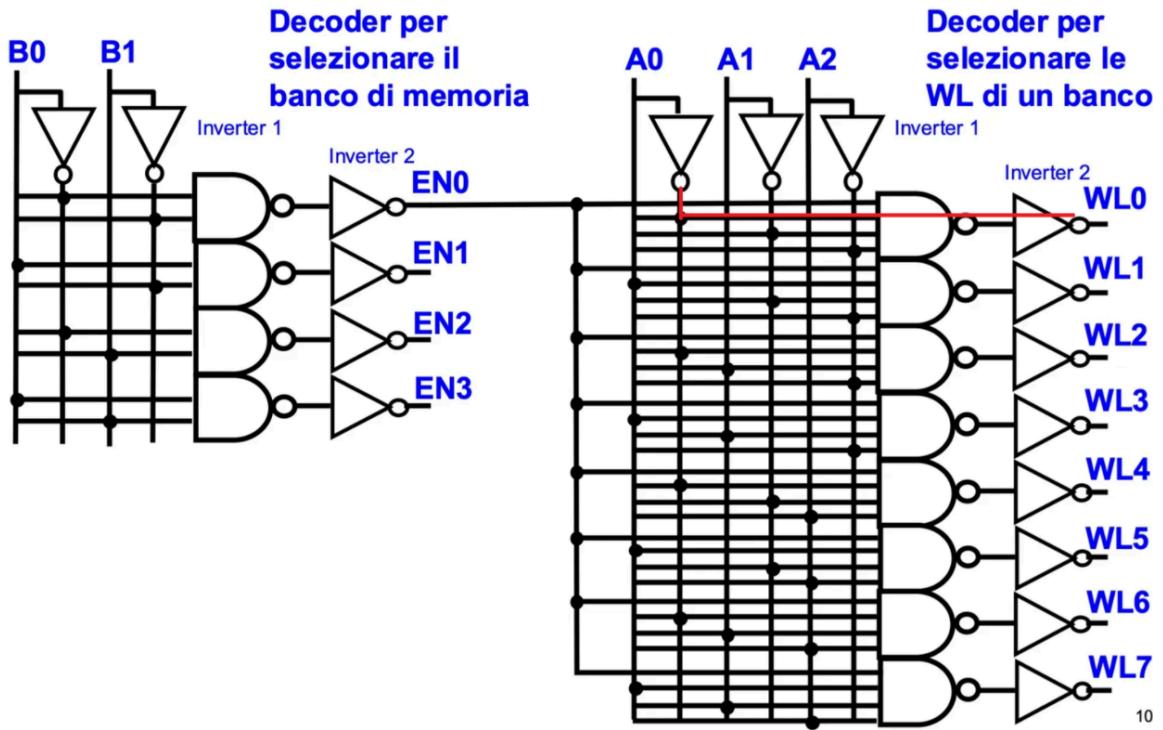
▼ Ritardo totale

Per il calcolo del ritardo totale possiamo considerare 2 percorsi possibili:

- $B \rightarrow EN \rightarrow WL$



- $A \rightarrow WL$



Questi due percorsi hanno in comune il ritardo della NAND (0.55 ps) e dell'Inverter 2 della wordline (0.29 ps).

Prima di arrivare alla NAND, il primo percorso accumula un ritardo massimo dato dalla somma di Inverter 1, NAND e Inverter 2 del banco ($0.28 + 0.28 + 1.1$) mentre il percorso 2 ha il solo contributo dell'Inverter 1 della wordline (0.55).

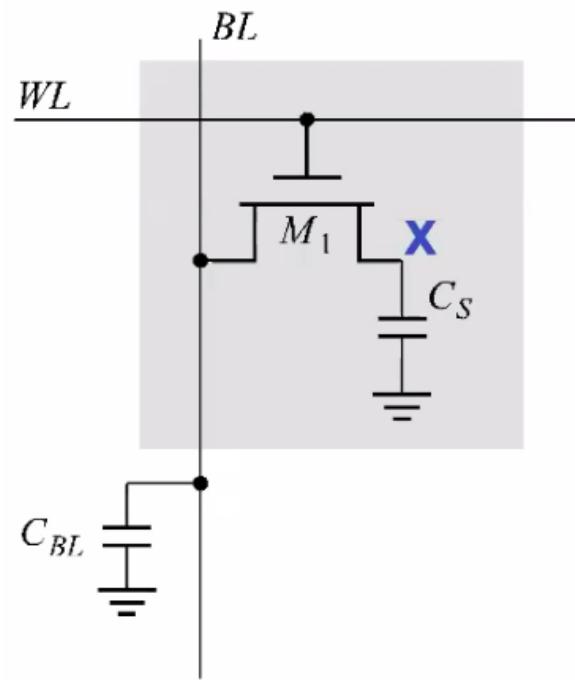
Dobbiamo prendere ovviamente il massimo tra questi due valori, ovvero come potevamo aspettarci quello del percorso che parte dal banco, e sommarlo ai ritardi di NAND e Inverter 2 della wordline.

$$\begin{aligned} & \max(0,28 \text{ ps} + 0,28 \text{ ps} + 1,1 \text{ ps}, 0,55 \text{ ps}) + 0,55 \text{ ps} + 0,29 \text{ ps} \\ & = (0,28 \text{ ps} + 0,28 \text{ ps} + 1,1 \text{ ps}) + 0,55 \text{ ps} + 0,29 \text{ ps} = 2,5 \text{ ps} \end{aligned}$$

Esercizio 4 - Tensioni nella cella DRAM (teorico)

▼ Creatore originale @Gianbattista Busonera

Calcolare la variazione di tensione della bitline di una memoria *DRAM* alimentata con tensione V_{DD} e che ha capacità di storage C_S e capacità della bitline C_{BL} .



EA06 - Cella DRAM standard

▼ Soluzione

Si procede valutando inizialmente:

- Tensione V_{BL} iniziale: V_{pre}
 - Normalmente la linea di bitline è precaricata ad una certa tensione, solitamente pari alla tensione di soglia $V_T = \frac{V_{DD}}{2}$
- Tensione V_S iniziale: V_{bit}
 - Tale valore consente di valutare se il condensatore C_S "contiene" un uno/zero logico:
 - $V_{bit} = V_{DD} \Rightarrow C_S$ contiene un uno logico
 - $V_{bit} = 0 \Rightarrow C_S$ contiene uno zero logico

Quando accendiamo il transistore M_1 le due tensioni V_{BL} e V_S si egualgano come visibile in seguito:



Ripasso formule condensatori:

$$Q = cV \rightarrow \Delta Q = c\Delta V$$

Quando due condensatori C_1 e C_2 idealmente isolati tra loro vengono collegati tra loro:

- $\Delta Q_{tot} = 0$ (principio di conservazione della carica)
- $V_f = V_1 = V_2$

Quando si collegano C_S e C_{BL} , la carica totale si conserva, si passa dunque da:

1. $Q_{tot,i} = Q_S + Q_{BL} = C_S V_{bit} + C_{BL} V_{pre}$
2. $Q_{tot,f} = C_S V_S + C_{BL} V_{BL}$
 - a. Le due capacità C_S e C_{BL} si trovano però alla stessa tensione V_f dopo la redistribuzione delle cariche, dunque: $Q_{tot,f} = (C_S + C_{BL})V_f$
3. Imponendo $\Delta Q_{tot} = 0$ si ottiene $Q_{tot,f} = Q_{tot,i}$ e dunque:
 - a. $C_S V_{bit} + C_{BL} V_{pre} = (C_S + C_{BL})V_f$
 - b. Si ricava dunque $V_f = \frac{C_S V_{bit} + C_{BL} V_{pre}}{C_S + C_{BL}}$
4. Si ricavano dunque dunque:
 - a. $\Delta V_{BL} = V_f - V_{pre} = \frac{C_S V_{bit} + C_{BL} V_{pre}}{C_S + C_{BL}} - V_{pre} = \frac{C_S (V_{bit} - V_{pre})}{C_S + C_{BL}}$
 - b. $\Delta V_S = V_f - V_{bit} = \frac{C_S V_{bit} + C_{BL} V_{pre}}{C_S + C_{BL}} - V_{bit} = \frac{C_{BL} (V_{pre} - V_{bit})}{C_S + C_{BL}}$



Si ricavano dunque due formule importanti per gli esercizi successivi:

$$\Delta V_{BL} = (V_{bit} - V_{pre}) \frac{C_S}{C_S + C_{BL}}$$

$$\Delta V_S = (V_{pre} - V_{bit}) \frac{C_{BL}}{C_S + C_{BL}}$$

P.S. La capacità C_{BL} , cioè la capacità vista dalla bitline è approssimabile con la capacità equivalente di tutti gli M transistor con capacità C_{Drain} e, pertanto:

$$C_{BL} \simeq M \cdot C_{Drain}$$

Distinguiamo i due casi:

Bit memorizzato in C_S	V_{bit}	ΔV_{BL}	Dopo il "sense amplifier"
0	0	$-V_{pre} \frac{C_S}{C_S + C_{BL}} < 0$	$\Delta V_{BL} = 0 = 0$ logico
1	$V_{DD} - V_{th}$	$(V_{DD} - V_{th} - V_{pre}) \frac{C_S}{C_S + C_{BL}} > 0$	$\Delta V_{BL} = V_{DD} = 1$ logico

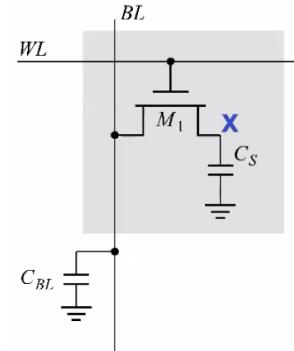
La capacità di storage C_S è solitamente molto piccola mentre C_{BL} molto grande \Rightarrow La variazione ΔV_{BL} è una variazione piccola di tensione e, quindi, non è sufficiente per interpretare un uno/zero logico. E' necessario introdurre un cosiddetto "sense amplifier" il quale amplifica la tensione portandola a zero se negativa o a V_{DD} se positiva.

Esercizio 5 - Lettura in DRAM

▼ Creatore originale: @Gianbattista Busonera

Una cella DRAM ha:

- $C_S = 20 \text{ fF}$
- $M = 512$ celle collegate alla bitline. (Si noti come una cella è formata da un transistor M_i e una cella di storage C_S come visibile nell'evidenziazione grigia in figura).
- Pass transistor M_i con:
 - $C_{Dra\in} = C_D = 0.1 \text{ fF}$
 - Tensione di soglia $V_{th} = 0.1V$
- Tensione di alimentazione $V_{DD} = 1V$



Visto che ci sono 512 celle collegate alla bitline, considereremo la capacità equivalente vista dalla bitline:

$$C_{BL} = M \cdot C_{Dra\in} = 512 \cdot 0.1 \text{ fF} = 51.2 \text{ fF}$$

Visto che ci sono M celle, ci sono anche M condensatori di Drain di ogni transistor (i quali sono collegati in parallelo alla bitline).

Obiettivo 1 - Variazione di tensione a seguito di una lettura di un 1

Determinare la variazione di tensione sulla bitline ΔV_{BL} e sul condensatore di storage ΔV_S a seguito di una lettura di un 1 logico memorizzato nella cella di storage C_S .

▼ Calcolo variazione di tensione di bitline

Facciamo riferimento alla formula trovata nell'esercizio 4:



$$\Delta V_{BL} = (V_{bit} - V_{pre}) \frac{C_S}{C_S + C_{BL}}$$

In questo caso, ipotizzando di leggere uno 0 logico si ha che:

- $V_{bit} = V_{DD} - V_{th} = 1 - 0.1 = 0.9V$ (avendo ipotizzato un 1 logico)

- $V_{pre} = \frac{V_{DD}}{2} = \frac{1}{2} = 0.5V$

Sostituendo nella formula:

$$\Delta V_{BL} = (0.9 - 0.5) \frac{C_S}{C_S + 512C_{DraIn}} = 0.4 \frac{20}{71.2} = 112 \text{ mV}$$

▼ Calcolo variazione di tensione di storage

Facciamo riferimento alla formula trovata nell'esercizio 4:



$$\Delta V_S = (V_{pre} - V_{bit}) \frac{C_{BL}}{C_S + C_{BL}}$$

In questo caso, ipotizzando di leggere uno 0 logico si ha che:

- $V_{bit} = V_{DD} - V_{th} = 1 - 0.1 = 0.9V$ (avendo ipotizzato un 1 logico)
- $V_{pre} = \frac{V_{DD}}{2} = \frac{1}{2} = 0.5V$

Sostituendo nella formula:

$$\Delta V_S = (0.5 - 0.9) \frac{51.2}{20 + 51.2} = -0.4 \frac{51.2}{71.2} = -288 \text{ mV}$$



In seguito a una lettura mi piacerebbe non modificare il contenuto della cella! Notiamo però che ΔV_S si degrada (-288 mV)

Dopo la lettura di un uno logico, il sense amplifier amplifica ΔV_{BL} , la porta su alla tensione di alimentazione V_{DD} e riscrive nella cella C_S il valore appena letto.

Obiettivo 2 - Variazione di tensione a seguito di una lettura di uno 0

Determinare la variazione di tensione sulla bitline ΔV_{BL} e sul condensatore di storage ΔV_S a seguito di una lettura di uno 0 logico memorizzato nella cella di storage C_S .

▼ Calcolo variazione di tensione di bitline

Facciamo riferimento alla formula trovata nell'esercizio 4:



$$\Delta V_{BL} = (V_{bit} - V_{pre}) \frac{C_S}{C_S + C_{BL}}$$

In questo caso, ipotizzando di leggere uno 0 logico si ha che:

- $V_{bit} = 0V$ (avendo ipotizzato uno 0 logico)
- $V_{pre} = \frac{V_{DD}}{2} = \frac{1}{2} = 0.5V$

Sostituendo nella formula:

$$\Delta V_{BL} = (0 - 0.5) \frac{C_S}{C_S + 512C_{Dra\text{in}}} = -0.5 \frac{20}{20 + 51.2} = -0.5 \frac{20}{71.2} = -140 mV$$

▼ Calcolo variazione di tensione di storage

Facciamo riferimento alla formula trovata nell'esercizio 4:



$$\Delta V_S = (V_{pre} - V_{bit}) \frac{C_{BL}}{C_S + C_{BL}}$$

In questo caso, ipotizzando di leggere uno 0 logico si ha che:

- $V_{bit} = 0V$ (avendo ipotizzato uno 0 logico)
- $V_{pre} = \frac{V_{DD}}{2} = \frac{1}{2} = 0.5V$

Sostituendo nella formula:

$$\Delta V_S = (0.5 - 0) \frac{51.2}{20 + 51.2} = 0.5 \frac{51.2}{71.2} = 360 mV$$



In seguito a una lettura mi piacerebbe non modificare il contenuto della cella! Notiamo però che ΔV_S aumenta (360 mV)

Dopo la lettura di uno zero logico, il sense amplifier porta a zero ΔV_{BL} e riscrive nella cella C_S il valore appena letto (0 logico).

Obiettivo 3 - Massimo numero M di celle connesse

Nel caso in cui la **sensibilità** del Sense Amplifier fosse di 50mV (nel caso in cui la variazione fosse inferiore il sense amplifier non riuscirebbe a identificare se ΔV_{BL} corrisponda a un uno o uno zero logico), **determinare il massimo numero M di celle connesse alla bitline.**

▼ Soluzione

L'incognita è in questo caso il numero massimo di celle connesse alla bitline.

Quello che vogliamo è che $|\Delta V_{BL}| \geq 50 \text{ mV}$ così che il sense amplifier si accorga di variazioni.

Nota la formula:

$$\Delta V_{BL} = (V_{bit} - V_{pre}) \frac{C_S}{C_S + C_{BL}}$$

Esplicitando $C_{BL} = M \cdot C_{D_{rain}}$:

$$\Delta V_{BL} = (V_{bit} - V_{pre}) \frac{C_S}{C_S + M \cdot C_{D_{rain}}}$$

Risolviamo, per semplicità, la disequazione senza valore assoluto:

$$(V_{bit} - V_{pre}) \frac{C_S}{C_S + M \cdot C_{D_{rain}}} \geq 50\text{mV}$$

$$\frac{C_S(V_{bit} - V_{pre}) - 0.05(C_S + M \cdot C_{D_{rain}})}{C_S + M \cdot C_{D_{rain}}} \geq 0$$

$$C_S(V_{bit} - V_{pre}) - 0.05C_S \geq 0.05M \cdot C_{D_{rain}}$$

$$\frac{C_S(V_{bit} - V_{pre}) - 0.05C_S}{0.05C_{Dra\text{in}}} \geq M$$

$$M \leq \frac{C_S}{C_{Dra\text{in}}} \left(\frac{V_{bit} - V_{pre}}{0.05} - 1 \right)$$

Bisognerebbe distinguere chiaramente il caso “peggiore” di lettura tra 0 e 1 logico. Bisogna dunque osservare le variazioni di ΔV_{BL} in entrambi i casi. Dai punti precedenti:

- Nel caso di 1 logico: $\Delta V_{BL} = 112 \text{ mV}$
- Nel caso di 0 logico: $\Delta V_{BL} = -140 \text{ mV}$

Visibilmente, la lettura di un uno logico lascia un margine minore per il sense amplifier, dunque considereremo come caso peggiore la lettura di un uno logico.

Otteniamo dunque:

$$M \leq \frac{20}{0.1} \left(\frac{0.9 - 0.5}{0.05} - 1 \right) = 1400 \text{ celle}$$

Da cui $M \leq 1400$ celle.

Esercizio 6 - Rinfresco celle DRAM (todo)

▼ Creatore originale: @Gianbattista Busonera

La corrente di leakage nei pass transistor delle celle di una DRAM è $I_{leak} = 25 \text{ fA}$ e la capacità di storage è $C_S = 20 \text{ fF}$.

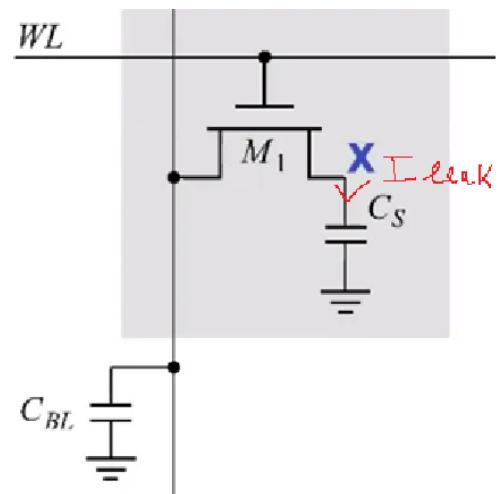
Determinare il periodo di refresh per garantire una variazione massima della tensione del condensatore di storage di $0.1V$, cioè $|\Delta V_S| < 0.1V$

▼ Introduzione

Il transistore di una cella, quando non conduce, non è esattamente un circuito aperto!

In realtà fa passare la corrente di leakage che, a lungo andare, modifica la tensione sul condensatore di storage

C_S . Pertanto è necessario, di tanto in tanto, fare una operazione di rinfresco in cui il valore di tutte le celle viene letto e successivamente riscritto.



Vogliamo trovare la condizione tale per cui il refresh viene fatto quando la tensione di storage sul condensatore varia di $0.1V = 100mV$.

▼ Svolgimento

Dall'equazione costitutiva di un condensatore C_S :

$$q = C_S V \rightarrow i(t) = C_S \frac{dV}{dt}$$

Se però la corrente di leakage è costante otteniamo $I_{leak} = C_S \frac{\Delta V}{\Delta t}$.

Esplicitiamo la variazione di tensione e imponiamola inferiore a 0.1V.

$$\Delta V = \frac{I_{leak} \Delta t}{C_S} \Rightarrow \frac{I_{leak} \Delta t}{C_S} \leq 0.1V$$

Da cui si ricava:

$$\Delta t = T_{refresh} \leq \frac{0.1}{I_{leak}} C_S = \frac{0.1V}{25fA} 20fF = 80 ms$$



Se aspettassi più di 80 ms, il condensatore di storage varierebbe la propria tensione di più di 0.1V.

Esercizio 7 - Ritenzione in cella Flash

▼ Creatore originale: @Gianbattista Busonera

Una cella FLASH ha una capacità tra il gate di controllo e il gate flottante $C_{PP} = 50 \text{ aF} = 50 \cdot 10^{-18} \text{ F}$.

Se la massima variazione ammissibile della tensione di soglia per garantire la ritenzione del dato memorizzato per 10 anni nella cella è $\Delta V_{th} = 1V$, determinare la massima corrente di perdita I_{leak} tra il gate flottante e substrato (in ampere e in elettroni persi/settimana).

▼ Introduzione

Nelle celle FLASH, così come in generale in tutte le memorie non volatili, ci sono delle cariche immagazzinate nel gate flottante (floating gate).

Tali cariche, per quanto pensiamo siano intrappolate, hanno una piccola possibilità di scappare. Esiste infatti una piccola corrente di perdita tra il gate flottante e il substrato dovuta a imperfezioni varie...

Vogliamo stabilire qual è la massima corrente di perdita accettabile affinchè la variazione della quantità di carica contenuta dentro il gate flottante sia contenuta entro una certa soglia.

La presenza di queste cariche intrappolate nel gate flottante modifica la tensione di soglia (corrispettivo dell'informazione memorizzata nella cella).



Se ci sono cariche \Rightarrow tensione di soglia aumenta \Rightarrow 1 logico

Se non ci sono cariche \Rightarrow tensione di soglia diminuisce \Rightarrow 0 logico

Si vuole che, in un tempo molto lungo (10 anni), tale variazione di tensione di soglia sia di al massimo 1 V.

▼ Soluzione

Dall'equazione costitutiva di un condensatore C_{PP} :

$$Q = C_{PP}V \rightarrow \Delta Q = C_{PP}\Delta V$$

Di conseguenza, posto $\Delta V_{th} = 1V$, otteniamo che la variazione di carica sul condensatore C_{PP} è pari a $\Delta Q = 50 \text{ aF} \cdot 1V = 50 \text{ aC}$

Ricaviamo il numero di elettroni con la formula: $q = n \cdot e$

$$n_{elettroni} = \frac{\Delta Q}{e} = \frac{50 \text{ aC}}{1.6 \cdot 10^{-19} C} = \frac{50 \cdot 10}{1.6} = 313 \text{ elettroni}$$



Significa che al massimo ci si può permettere di perdere 313 elettroni per non far variare la tensione di soglia più di un volt.

Volendo ricavare la corrente $I_{leak} = \frac{\Delta Q}{\Delta t}$, cioè la quantità di carica che fluisce nell'intervallo temporale di 10 anni, basta risolvere l'equazione considerando come intervallo temporale:

$$\Delta t = 10 \text{ anni} = 10 \cdot 365 \cdot 24 \cdot 60 \cdot 60 \text{ secondi}$$

$$\text{Si ottiene } I_{leak} = \frac{\Delta Q}{\Delta t} = \frac{50 \cdot 10^{-18}}{10 \cdot 365 \cdot 24 \cdot 60 \cdot 60} = 1.6 \cdot 10^{-25} A$$

Volendo poi calcolare invece la quantità di elettroni persi per ogni settimana, consideriamo il numero di settimane totale in 10 anni: $N_s = 10 \cdot 52 = 520$ settimane.

$$I_s = \frac{\text{elettroni totali}}{N_s} = \frac{313}{520} = 0.6 \frac{\text{elettroni}}{\text{settimana}}$$

Esercizio 8 - Max numero di cicli P/E in celle Flash

▼ Creatore originale: @Gianbattista Busonera

Le celle di una memoria flash da $4096 = 4 \cdot 1024 = 2^{12}$ blocchi possono essere programmate e cancellate al massimo per 10^4 volte.

In un ipotetico scenario di utilizzo, vengono continuamente eseguiti cicli di P/E (programmazione / erasing, cancellazione) su file di 50 blocchi ad un tasso medio di 1 file ogni 10 minuti. I file occupano al massimo 200 blocchi complessivi simultaneamente.

Determinare la **durata massima della flash** (in anni) nei due seguenti casi:

- No "wear levelling", ossia i file sono scritti sempre negli stessi 200 blocchi.
- "wear levelling", ossia i file vengono distribuiti uniformemente su tutti i 4096 blocchi.

▼ Introduzione generale

Quando si fanno delle operazioni di programmazione e cancellazione su una memoria non volatile, si alterano le caratteristiche di tale memoria. Dopo un certo numeri di cicli non è più affidabile fare cicli di programmazione e cancellazione sulla stessa cella.

Il "wear levelling" consiste nel distribuire gli accessi in maniera tale che si massimizzi la durata della memoria. Se facessimo accesso negli stessi blocchi finiremmo per renderli inutilizzabili dopo un certo numero di P/E.

▼ Ripassino sulle FLASH e contesto

- **Pagina** = unità minima di **lettura e scrittura** (es. 2–16 KB) e **unità minima di lettura e scrittura**
- **Blocco** = gruppo di pagine (es. 64–256 pagine) e **unità minima di cancellazione**
 - Prima di poter riscrivere una pagina è necessario cancellare tutto il blocco che la contiene
- **File** = entità logica che può estendersi su più blocchi

Nel nostro esercizio:

- La memoria non volatile FLASH ha 4096 blocchi (4096 set di pagine)
- Ogni file occupa 50 blocchi
- Al massimo possono esserci 200 blocchi "attivi" simultaneamente nel caso di assenza di wear levelling ⇒ al massimo 4 file attivi per volta.
 - Nel caso di wear levelling si possono invece utilizzare tutti i 4096 blocchi.

Obiettivo 1 - Caso senza wear levelling

▼ Introduzione

Non appena le celle su cui sto facendo accesso arrivano a $10^4 = 10000$ cicli, la memoria non è più affidabile.

Il testo ci dice che ogni 10 minuti viene scritto un nuovo file e che quindi ci troviamo in una situazione del tipo:

0-9 minuti	10-19 minuti	20-29 minuti	30-39 minuti
------------	--------------	--------------	--------------

Scrivo un file nei primi 50 blocchi	Scrivo un file nei secondi 50 blocchi	Scrivo un file nel terzo set da 50 blocchi	Scrivo un file nel quarto set da 50 blocchi
-------------------------------------	---------------------------------------	--	---

Al 40° minuto tenteremo nuovamente di scrivere ma ciò non sarà possibile perché dobbiamo prima cancellare! Notiamo dunque che riscriviamo sullo stesso blocco ogni 40 minuti!

▼ Soluzione

Mediamente accedo allo stesso file una volta ogni 4 accessi visto che i su 200 blocchi entrano 4 file (ognuno da 50 blocchi) $\Rightarrow T_a = \text{periodo di accesso} = \frac{200}{50} \cdot T_{scrittura} = 4 \cdot 10 \text{ min.}$

Visto che mediamente, ogni 40 minuti, tento di scrivere sullo stesso set di blocchi (e quindi dovrei cancellare), impiegherò un tempo per fare $N_{max} = 10000$ cicli di P/E su una cella pari a:

$$T = N_{max} \cdot T_a = 10000 \cdot 40 \text{ min} = 400000 \text{ minuti} = 278 \text{ giorni} < 1 \text{ anno}$$

Obiettivo 2 - Caso con wear levelling

▼ Introduzione

Non appena le celle su cui sto facendo accesso arrivano a $10^4 = 10000$ cicli, la memoria non è più affidabile.

Il testo ci dice che mediamente accediamo ad un file composto da 50 blocchi (ma con wear levelling di blocchi utilizzabili ne avrei 4096 $\Rightarrow 4096/50$ file) mediamente ogni 10 minuti.

▼ Soluzione

Si calcola il tempo necessario per un ciclo di P/E per lo stesso set di blocchi:

$$T_{accesso} = \frac{N_{blocchi\ tot}}{N_{blocchi\ per\ file}} \cdot T_{scrittura} = \frac{4096}{50} \cdot 10 \text{ min} = \frac{4096}{5} \text{ min}$$

Maiamente, dunque, scriviamo e cancelliamo (P/E) uno stesso set di blocchi (file) ogni 812,2 minuti.

Calcoliamo dunque il tempo necessario affinchè si effettuino 10000 cicli di P/E su uno stesso set di blocchi:

$$T = N_{max} \cdot T_{accesso} = 10000 \cdot \frac{4096}{5} \text{ min} = 8192000 \text{ min}$$

$$8192000 \text{ min} = \frac{8192999}{60 \cdot 24 \cdot 365} \simeq 15.59 \text{ anni}$$