



SAPIENZA
UNIVERSITÀ DI ROMA

UNIVERSITÀ "SAPIENZA" DI ROMA
FACOLTÀ DI INFORMATICA

Progettazione di Sistemi Digitali

Appunti integrati con il libro "Digital Design and Computer Architecture - ARM Edition", Sarah Harris, David Harris

Author
Simone Bianco

24 gennaio 2023

Indice

0	Introduzione	1
1	Sistema numerico binario	2
1.1	Numeri binari naturali	2
1.1.1	Conversioni, Unità di misura e Sistema esadecimale	3
1.1.2	Operazioni aritmetiche in sistema binario	6
1.2	Numeri binari negativi	8
1.2.1	Numeri in Sign/Magnitude	8
1.2.2	Numeri in Complemento a 2	8
1.3	Numeri binari razionali	10
1.3.1	Numeri a virgola fissa	11
1.3.2	Numeri a virgola mobile	12
2	Circuiti combinatori	16
2.1	Definizioni	16
2.2	Porte logiche	17
2.3	Equazioni booleane	19
2.3.1	Somma di prodotti (SOP) e Prodotto di somme (POS)	20
2.3.2	Algebra di Boole	21
2.4	Dalla logica ai circuiti	24
2.4.1	Regole di lettura	24
2.4.2	Output multipli, X e Z	24
2.4.3	Le Mappe di Karnaugh (K-Maps)	27
2.4.4	Logica combinatoriale multi-livello	31
2.4.5	Timing dei circuiti	34
3	Circuiti sequenziali	35
3.1	Elementi di stato	35
3.1.1	Circuito bistabile	36
3.1.2	SR Latch	36
3.1.3	D Latch	37
3.1.4	D Flip-Flop	39
3.1.5	Varianti di Flip-Flop	40
3.1.6	Ring Oscillator	43

3.2	Circuiti Sequenziali Sincroni	44
3.2.1	Macchine a Stato Finito	44
3.2.2	Esempio di FSM di Moore e FSM di Mealy	46
3.2.3	Da FSM di Moore a FSM di Mealy e viceversa	49
3.2.4	Temporizzazioni sequenziali	51
3.2.5	Pipelines e parallelismo	55
4	Blocchi costruttivi digitali	59
4.1	Blocchi aritmetici	59
4.1.1	Sommatori	59
4.1.2	Sottrattori e Comparatori	63
4.1.3	Shifter	64
4.1.4	Moltiplicatori	65
4.2	Blocchi utilitari	66
4.2.1	Contatore	66
4.2.2	Shift Register	67
4.3	Memorie	68
4.3.1	Celle di bit delle memorie	69
4.3.2	DRAM, SRAM e ROM	70
4.4	Array Logici	72
4.4.1	Programmable Logic Array	72
4.4.2	Field Programmable Gate Array	73
4.5	Arithmetic Logic Unit (ALU)	74
4.5.1	ALU con Flag di stato	75
5	Linguaggi descrittivi dell'Hardware	76
5.1	Moduli Comportamentali e Strutturali	76
5.1.1	Moduli Comportamentali	76
5.1.2	Moduli strutturali	78
5.2	Convenzioni del SystemVerilog	79
5.2.1	Operazioni su vettori di bit	79
5.2.2	Operatori di riduzione	80
5.2.3	Assegnamenti condizionali	81
5.2.4	Variabili interne	81
5.3	Numeri e Manipolazione di bit	82
5.4	Datatype logici del SystemVerilog	83
5.5	Delay	83
5.6	Logica Sequenziale	84
5.6.1	Elementi di Stato in SysVerilog	84
5.6.2	Case e Casez	86
5.6.3	Assegnamento Bloccante e Non-bloccante	87
5.7	Macchine a Stati Finiti	89
5.8	Moduli di Testbench	90
5.9	Moduli parametrizzati	94

Capitolo 0

Introduzione

Negli ultimi tre decenni, i **microprocessori** e il loro perfezionamento hanno rivoluzionato il mondo attorno a noi al punto tale da rendere attualmente quasi impossibile immaginare come fosse vivere prima dell'avvento del digitale. Per questo motivo, è facile essere travolti dall'*appeal* di quella che probabilmente è la più importante invenzione umana subito dopo la scoperta del fuoco sia dal punto di vista sociale, sia dal punto di vista tecnico e sia da quello economico.

In questo corso apprenderemo cosa c'è al di sotto di un *computer*, come modellare un *sistema digitale* e i principi del *digital design*, concentrandoci in particolare su:

- Le basi della **logica binaria** (porte logiche, sistemi di numerazione, rappresentazione dell'informazione)
- La **progettazione di sistemi combinatori** (equazioni booleane, algebra booleana, mappe K, blocchi combinatori, timing)
- La **progettazione di sistemi sequenziali** (Latch e Flip-Flop, progettazione di logica sincrona, macchine a stati finiti, tempistiche della logica sequenziale)
- I **blocchi digitali costitutivi** (circuiti aritmetici, blocchi sequenziali, matrici di memoria, matrici logiche)
- I **linguaggi descrittivi dell'hardware o HDL** (logica combinatoria, logica sequenziale, macchine a stati finiti, system verilog, VHDL)

Per cominciare, vederemo nel dettaglio il funzionamento del **sistema numerico binario**, ossia la base senza cui nessun microprocessore moderno potrebbe funzionare.

Menzioni onorevoli

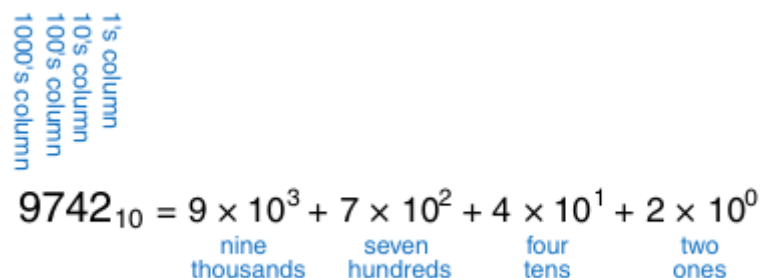
Si ringrazia il Prof. Salvatore Pontarelli per le lezioni, spiegazioni aggiuntive e il materiale fornito. Ringrazio inoltre coloro che hanno contribuito con correzioni e segnalazioni su eventuali errori/typo commessi, in particolare Matteo Collica. Ovviamente ringrazio anche coloro che hanno usufruito ed usufruiranno di questi appunti :)

Capitolo 1

Sistema numerico binario

1.1 Numeri binari naturali

Nella maggior parte dei casi, nella vita reale siamo abituati a contare elementi e ad eseguire calcoli matematici in **decimale**, utilizzando l'insieme di numeri 0, 1, 2, ..., 9. Per poter rappresentare i numeri più grandi di questi, abbiamo instaurato un **sistema posizionale** dove ad ogni *colonna* ha un valore 10 volte più grande della precedente. Dunque, da destra a sinistra, le colonne del sistema decimale assumono il valore della cifra rappresentata moltiplicata per una *potenza di 10*, come nel seguente esempio:


$$9742_{10} = 9 \times 10^3 + 7 \times 10^2 + 4 \times 10^1 + 2 \times 10^0$$

nine thousands seven hundreds four tens two ones

Abbiamo quindi definito le caratteristiche di un sistema numerico in **base 10** (che indicheremo con un 10 al di sotto del numero rappresentato). Utilizzando le stesse caratteristiche, possiamo definire dei sistemi numerici in *base n*, dove ogni colonna assume un valore *n* volte più grande della precedente.

In particolare, il sistema numerico con cui lavorano i microprocessori è il sistema in **base 2**. L'impiego di questo sistema numerico, seppur di difficile utilizzo per noi umani, è estremamente conveniente per i sistemi digitali, poiché abbiamo solo due cifre disponibili: lo **0** e l'**1**. Traducendo ciò in termini di circuiti elettronici, è possibile utilizzare queste uniche due cifre per rappresentare gli unici due stati che un componente può assumere: *acceso* (dunque con un passaggio di corrente al suo interno, rappresentato con il numero 1) o *spento* (senza passaggio di corrente, rappresentato con il numero 0).

Seguendo le caratteristiche definite precedentemente per il sistema numerico decimale, vediamo la rappresentazione di un numero in sistema binario:

$$\begin{array}{c}
 \text{1's column} \\
 \text{2's column} \\
 \text{4's column} \\
 \text{8's column} \\
 \text{16's column}
 \end{array}
 \quad
 10110_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 22_{10}$$

one sixteen
no eight
one four
one two
no one

Per facilitare le conversioni tra i due sistemi che effettueremo in seguito, possiamo tabellare le potenze di 2 più comuni:

Potenze	Valore decimale	Potenze	Valore decimale
2^{-5}	0.03125	2^3	8
2^{-4}	0.0625	2^4	16
2^{-3}	0.125	2^5	32
2^{-2}	0.25	2^6	64
2^{-1}	0.5	2^7	128
2^0	1	2^8	256
2^1	2	2^9	512
2^2	4	2^{10}	1024

1.1.1 Conversioni, Unità di misura e Sistema esadecimale

Conversione da Binario a Decimale

Per poter convertire un numero binario in uno decimale, il procedimento è molto semplice: partendo dalla cifra più a destra, **moltiplichiamo** ogni colonna con una **potenza di 2**, incrementando l'elevamento di 1 ad ogni passaggio, partendo da 0. Infine, **sommeremo** tutti i valori precedentemente calcolati per ottenere il numero decimale equivalente.

Per comprendere meglio, riprendiamo l'esempio di numero binario riportato precedentemente, 10110_2 . Partendo dalla cifra più a destra, numeriamo i valori delle potenze corrispondenti ad ogni colonna:

- 1° cifra più a destra (0) \Rightarrow moltiplico per $2^0 \Rightarrow 0 \cdot 2^0$
- 2° cifra più a destra (1) \Rightarrow moltiplico per $2^1 \Rightarrow 1 \cdot 2^1$
- 3° cifra più a destra (1) \Rightarrow moltiplico per $2^2 \Rightarrow 1 \cdot 2^2$
- 4° cifra più a destra (0) \Rightarrow moltiplico per $2^3 \Rightarrow 0 \cdot 2^3$
- 5° cifra più a destra (1) \Rightarrow moltiplico per $2^4 \Rightarrow 1 \cdot 2^4$

Una volta calcolati i valori assunti da ogni colonna, non ci resta che sommarli per ottenere il numero in base decimale:

$$10110_2 = 0 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4 = 0 + 2 + 4 + 0 + 16 = 22_{10}$$

Conversione da Decimale a Binario

Per convertire un numero decimale in un numero binario, possiamo procedere in due modi:

1. Trovare la **più alta potenza di due** minore del numero da convertire e **sottrarla** a quest'ultimo. Se il valore ottenuto non è 0, allora *ripetere* il procedimento. Riscrivere in binario le potenze trovate.

$$53_{10} = ?_2$$

$$(a) \ 53 - 2^5 = 21 \Rightarrow \text{Scriverò 1 nella 6}^\circ \text{ colonna da destra}$$

$$(b) \ 21 - 2^4 = 5 \Rightarrow \text{Scriverò 1 nella 5}^\circ \text{ colonna da destra}$$

$$(c) \ 5 - 2^2 = 1 \Rightarrow \text{Scriverò 1 nella 3}^\circ \text{ colonna da destra}$$

$$(d) \ 1 - 2^0 = 0 \Rightarrow \text{Scriverò 1 nella 1}^\circ \text{ colonna da destra}$$

$$\text{Risultato: } 53_{10} = 110101_2$$

2. **Dividere per 2** (divisione con resto) il numero da convertire. *Ripetere* il procedimento fino a che non si raggiunge 0. Infine, riscrivere da sinistra verso destra **tutti i resti** trovati a partire *dal basso verso l'alto*

$$53_{10} = ?_2$$

Divisione	Quoziente	Resto
53/2	26	1
26/2	13	0
13/2	6	1
6/2	3	0
3/2	1	1
1/2	0	1

$$53_{10} = 110101_2$$

Conversioni approssimate

Come abbiamo visto nella sezione 1.1, il valore di 2^{10} (ossia 1024) è molto vicino a 1000. Per via di questa coincidenza, possiamo facilmente calcolare (approssimativamente) le potenze di 2 di grande dimensione:

$$2^{32} = 2^{10} \cdot 2^{10} \cdot 2^{10} \cdot 2^2 \approx 1000 \cdot 1000 \cdot 1000 \cdot 4 \approx 4'000'000'000$$

Ovviamente, ciò implica che più sia grande la potenza di 2 che andiamo a calcolare in questo modo, più grande sarà l'errore di approssimazione (ad esempio, in realtà $2^{32} = 4'294'967'296$).

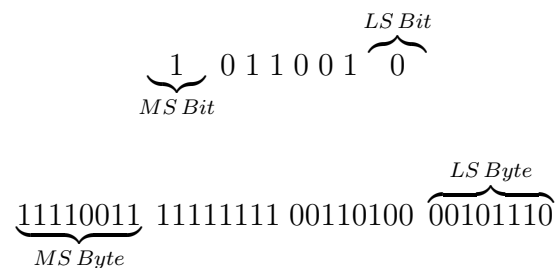
Unità di misura

Nel corso degli anni, si sono andate a definire quelle che sono delle vere e proprie **unità di misura** del sistema di numerazione binario:

1. **1 Bit** = 1 cifra binaria
2. **1 Nibble** = 4 Bit
3. **1 Byte** = 8 Bit = 2 Nibble
4. **1 Kilobyte** = 1024 Byte
5. **1 Megabyte** = 1024 Kilobyte
6. **1 Gigabyte** = 1024 Megabyte
7. ...

Esiste anche un'unità di misura generica chiamata **word**, indicante una quantità di Bit a scelta dell'utilizzatore o del contesto

È necessario affermare anche l'esistenza di due **terminologie** importanti che ci torneranno utili in futuro: **most significant** e **least significant**. Queste due terminologie vengono applicate solitamente ai *Bit* o ai *Byte*, indicando rispettivamente il Bit/Byte *più a sinistra* (most significant) e il Bit/Byte *più a destra* (least significant).



Sistema di numerazione esadecimale

Oltre al classico sistema numerico decimale e al sistema binario, un ulteriore sistema che utilizzeremo per rappresentare i numeri è il **sistema esadecimale (hexadecimal)**, ossia in *base 16*.

Ma come possiamo rappresentare i valori di un sistema numerico in base 16 utilizzando solo i 10 simboli forniti dal sistema decimale? La risposta è molto semplice: utilizzando una scrittura **alfanumerica**. L'insieme dei valori rappresentabili in un sistema decimale corrisponde a:

Decimale	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Esadecimale	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Poiché il numero 16 equivale ad una potenza di 2, questo sistema numerico risulta particolarmente comodo per rappresentare *lunghe sequenze di Byte*, poiché ci permette di rappresentare con una singola *cifra* (o *valore*) un **insieme di 4 Bit** (o 1 Nibble):

Decimale	Binario	Esadecimale
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Per via di questa proprietà, quindi, il sistema esadecimale viene utilizzato come **abbreviazione** del sistema binario *facilmente convertibile* da binario a esadecimale (o viceversa), come nel seguente esempio:

$$\underbrace{0100}_4 \underbrace{1010}_A \underbrace{1111}_F \Rightarrow 4AF_{16} \text{ oppure } 0x4AF$$

Invece, per poter effettuare delle conversioni tra il sistema esadecimale e quello decimale, possiamo utilizzare i vari metodi già impiegati effettuare le conversioni tra binario e decimale, *sostituendo* l'uso del 2 con l'uso del 16.

1.1.2 Operazioni aritmetiche in sistema binario

Per poter **sommare** due numeri binari, il procedimento da seguire è lo stesso valido per la somma tra due numeri decimali: sommo colonna per colonna le cifre dei due numeri, scrivendo il valore ottenuto e il suo **riporto**, il quale conterà come valore aggiuntivo da considerare successiva somma in colonna. Di seguito un esempio grafico

$$\begin{array}{rcccccc} 3 & ^17 & ^13 & 4 & + & & ^10 & ^11 & ^10 & 1 & + \\ 5 & 1 & 6 & 8 & = & & 0 & 0 & 1 & 1 & = \\ \hline 8 & 9 & 0 & 2 & & & 1 & 0 & 0 & 0 & \end{array}$$

A sinistra: addizione tra due numeri decimali.

A Destra: addizione tra due numeri binari.

Allo stesso modo, le **moltiplicazioni** in sistema binario seguono anche esse la stessa logica delle moltiplicazioni decimali:

2 3 0 x	1 0 1 x
4 2 =	1 1 1 =
-----	-----
4 6 0 +	1 0 1 +
9 2 0 =	11 0 1 +
-----	-----
9 6 6 0	11 0 1 =

	1 0 0 0 1 1

A sinistra: prodotto tra due numeri decimali.

A Destra: prodotto tra due numeri binari.

Per poter effettuare **sottrazioni** e **divisioni**, vengono utilizzate le *proprietà matematiche*, trasformando ad esempio una sottrazione tra due numeri in una somma tra essi dove uno dei due assume valore negativo (es: $100_2 - 11_2 = 100_2 + (-11_2)$). Nei paragrafi successivi vedremo dei **metodi di rappresentazione** che permettono di poter lavorare con i numeri negativi.

Overflow

poiché i sistemi digitali operano con un **numero fisso** di Bit, è necessario considerare quei casi in cui, durante un'addizione o una moltiplicazione, il numero di Bit sia *insufficiente* a poter rappresentare il risultato. Questa problematica viene definita col termine **overflow** (ossia *stra-bordare*).

Nell'ultima moltiplicazione del paragrafo precedente, possiamo vedere un tipico esempio di overflow: i due moltiplicandi utilizzano entrambi 3 Bit, ma il loro prodotto ne richiede *almeno* 6. In questi casi, i *Bit in eccesso* vengono **scartati**, generando un **risultato "sballato"** (se nell'esempio tagliassimo i Bit in eccesso, la moltiplicazione risulterebbe come $101 \times 111 = 011$, dunque $5 \times 7 = 3$). Tuttavia, come vedremo in seguito, vi sono casi in cui l'overflow viene utilizzato come **vantaggio** e non come svantaggio.

Shift di Bit a destra e a sinistra

Un operatore binario aggiuntivo rispetto alla normale aritmetica decimale è l'operazione di **shift** (o *spostamento*). Lo shift è un operatore che prevede lo spostamento **a destra o a sinistra** di una certa quantità di *caselle* di tutti i Bit del numero binario sul quale viene applicato.

$$000101_2 \ll 2 = 010100$$

$$011000_2 \gg 2 = 000110$$

1.2 Numeri binari negativi

Fino ad ora, abbiamo parlato di numeri binari strettamente positivi. Tuttavia, per compiere ulteriori operazioni aritmetiche è necessario, dunque, introdurre il **segno** nell'ambito dei numeri binari. Vedremo due diversi **metodi di rappresentazione** dei numeri negativi, i numeri in **segno/magnitudine** e i numeri in **complemento a 2**.

1.2.1 Numeri in Sign/Magnitude

Nel primo metodo di rappresentazione, le carte in regola non cambiano molto: il **most significant Bit** del numero rappresenta il suo **segno**, mentre tutti gli altri rappresentano il suo valore (o **magnitudine**). Nel caso in cui il bit del segno assuma *valore 0*, allora saremo davanti ad un numero **positivo**, mentre se assumerà *valore 1* allora sarà un numero **negativo**. Di seguito due esempi:

$$\begin{array}{cc} \text{Segno} & \text{Segno} \\ \underbrace{0} & \underbrace{1} \\ 01011_2 & \Rightarrow +23 \quad \quad 01011_2 \Rightarrow -11 \end{array}$$

L'uso del MSB per il segno, ovviamente, **dimezza** e la gamma di valori positivi rappresentabili con una determinata quantità di bit, tuttavia sblocca l'accesso ad altrettanti numeri negativi. Ricapitolando, dati n *bit*, nei due metodi di rappresentazione abbiamo i seguenti **intervalli** di valori disponibili:

$$\begin{array}{cc} \text{Unsigned} & \text{Sign/Magnitude} \\ \underbrace{[0, 2^n - 1]} & \underbrace{[-(2^{n-1} - 1), 2^{n-1} - 1]} \end{array}$$

Nonostante ciò, anche questo formato ha delle problematiche: le **addizioni non funzionano** (come mostrato nell'esempio inferiore) e abbiamo **due modi** per rappresentare lo **zero** (1000 e 0000, corrispondenti a -0 e +0).

$$\begin{array}{rcccccc} & {}^11 & {}^11 & 1 & 0 & + \\ & 0 & 1 & 1 & 0 & = \\ \hline & \textcolor{red}{1} & 0 & 1 & 0 & 0 \end{array}$$

Errore!! Anche se scartassimo l'overflow, avremmo comunque $-6 + 6 = +4$

1.2.2 Numeri in Complemento a 2

Per risolvere queste problematiche, è stato introdotto il sistema binario basato sul **complemento a 2**, il quale aggiunge un **ulteriore significato** al **MSB**. In questo sistema, il MSB assume il suo **effettivo valore**, tuttavia, a differenza del sistema senza segno, il valore assunto sarà **negativo**. Dati n *bit*, dunque, il MSB assumerà il valore $-(2^{n-1})$, mentre tutti gli altri bit manterranno il valore **positivo**:

$$10111_{Ca2} = 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + (-1 \cdot 2^4) = -9_{10}$$

Il Complemento a 2 di un numero decimale **positivo** corrisponde esattamente alla sua normale rappresentazione nel sistema binario.

Per poter calcolare il Complemento a 2 di un numero decimale **negativo**, invece, è prima necessario calcolare il **Complemento ad 1** del suo corrispettivo numero binario **positivo**. Con la terminologia "calcolare il Complemento ad 1" si intende semplicemente **invertire** tutti i bit, trasformando quindi tutti i suoi 0 in 1 e viceversa:

$$-25_{10} \Rightarrow +25_{10} = 011001_2 \Rightarrow 100110_{Ca1}$$

Una volta calcolato il Complemento ad 1, il prossimo passo sarà semplicemente **sommare 1 al LSB del Ca1**

$$-25_{10} \Rightarrow 100110_{Ca1} \Rightarrow 100110_{Ca1} + 1 \Rightarrow 100111_{Ca2}$$

Per confermare che la conversione sia avvenuta correttamente, possiamo calcolare il valore del Ca2 ottenuto contando il valore dei singoli bit come mostrato ad inizio paragrafo:

$$100111_{Ca2} = 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 0 \cdot 2^4 + (-1 \cdot 2^5) = -25_{10}$$

Considerando tutte le proprietà del Ca2 che abbiamo descritto, possiamo notare come **calcolare il Ca2** di un numero *già scritto* in forma Ca2 equivale a **invertire** il suo segno:

$$\begin{aligned} +25_{10} &= 011001_{Ca2} & -25_{10} &= 100111_{Ca2} \\ Ca2(011001_{Ca2}) &= 100111_{Ca2} \end{aligned}$$

L'utilità del Ca2 è evidente: per via del modo in cui viene rappresentato, è possibile svolgere **operazioni** tra numeri di segno opposto senza problematiche (come mostrato nell'esempio inferiore). Inoltre, viene risolto anche il problema dei **due zeri**, poiché 0000 corrisponderà al normale 0, mentre 1000 corrisponderà a -8.

$$+6_{10} = 0110_{Ca2} \quad -6_{10} = 1010_{Ca2}$$

$$\begin{array}{rcccccc} & {}^11 & {}^10 & 1 & 0 & + \\ & 0 & 1 & 1 & 0 & = \\ \hline \textcolor{red}{1} & 0 & 0 & 0 & 0 & \end{array}$$

Corretto! Scartando l'overflow otteniamo che $-6 + 6 = 0$

poiché 1000 rappresenta un numero aggiuntivo rispetto al sistema sign/magnitude, il **range di valori** negativi rappresentabili in Ca2 è $-(2^{n-1})$, invece di $-(2^{n-1} - 1)$:

$$\begin{array}{ccc} \underbrace{[0, 2^n - 1]}_{Unsigned} & \underbrace{[-(2^{n-1} - 1), 2^{n-1} - 1]}_{Sign/Magnitude} & \underbrace{[-(2^{n-1}), 2^{n-1} - 1]}_{Ca2} \end{array}$$

Incrementare il numero di bit

Per lavorare con i numeri in Ca2, in alcuni casi può essere necessario incrementare il numero di bit utilizzati. Tuttavia, quando viene compiuta questa operazione è necessario tenere conto del **segno** del numero rappresentato:

- Il numero 010011_{Ca2} può essere "allungato" aggiungendo la quantità desiderata di **0** davanti al numero: $01011_{Ca2} \Rightarrow 00001011_{Ca2}$. Questo è possibile poiché si tratta di un numero *positivo*
- Il numero 110011_{Ca2} può essere "allungato" aggiungendo la quantità desiderata di **1** davanti al numero: $11011_{Ca2} \Rightarrow 11111011_{Ca2}$. Questo è possibile poiché si tratta di un numero *negativo*, quindi è necessario mantenere costanti il segno e il valore finale del numero. Nel caso in cui fosse utilizzato lo 0 al posto dell'1, il valore finale del numero in Ca2 **cambiarebbe**

Lo Shift come moltiplicazione o divisione

Poiché lo shift prevede lo **spostare i bit** di un numero binario a sinistra e a destra, nel caso dei numeri in Ca2 questo operatore può essere utilizzato come **moltiplicazione** e **divisione**, ma solo nel caso in cui il moltiplicatore o divisore sia **una potenza di 2**:

$$B \cdot 2^n = B \ll n \qquad B/2^n = B \ggg n$$

Il motivo per cui nella divisione utilizziamo il **triplo shift a destra** (\ggg), è semplicemente un modo per segnalare la necessità di dover tener conto del **segno** nel caso in cui venga applicato lo shift su un numero in Ca2 **negativo**. Inoltre, nell'eseguire questi calcoli tramite shift, è necessario sfruttare l'overflow a proprio vantaggio. Di seguito alcuni esempi:

- $00001 \ll 2 = 00100 \Rightarrow 1 \cdot 2^2 = 4$
- $11101 \ll 2 = 10100 \Rightarrow -3 \cdot 2^2 = -12$
- $01000 \ggg 2 = 00010 \Rightarrow 8/2^2 = 2$
- $10000 \ggg 2 = 11100 \Rightarrow -16/2^2 = -4$

1.3 Numeri binari razionali

Come per il sistema decimale, anche nel sistema binario è possibile rappresentare i numeri razionali. Tuttavia, nel caso del sistema binario esistono due tipologie di rappresentazione utilizzabili: i numeri a **virgola fissa** (**fixed-point numbers**) e i numeri a **virgola mobile** (**floating-point number**).

1.3.1 Numeri a virgola fissa

Questo sistema di rappresentazione è l'equivalente *1 ad 1* della **classica rappresentazione** dei numeri razionali nel sistema decimale. In quest'ultimo, il modo in cui i numeri razionali vengono rappresentati corrisponde alla seguente somma:

$$134.56_{10} = 1 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0 + 5 \cdot 10^{-1} + 6 \cdot 10^{-2}$$

Allo stesso modo, il seguente numero binario espresso in forma razionale corrisponderà ad una somma tra il prodotto delle singole cifre per le loro potenze di 2 rispettive, risultando quindi in una semplice **estensione** della normale rappresentazione binaria di un numero:

$$110.11_2 = 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} = 6.75_{10}$$

Per poter convertire un numero decimale razionale in un numero binario razionale, è necessario scomporre il processo in **due fasi**:

1. Conversione in binario della **parte intera** del numero decimale
2. Conversione in binario della **parte razionale** del numero decimale

La prima fase siamo già in grado di svilupparla tramite le metodologie imparate nella sezione 1.1.1. Per la seconda, invece, è necessario introdurre un nuovo metodo:

1. Moltiplicare la parte razionale per 2.
 - (a) Se il risultato ottenuto è **uguale ad 1**, allora passa al punto 2.
 - (b) Se il risultato ottenuto è **maggiore di 1**, allora sarà necessario **sottrarre 1** al risultato ottenuto. Successivamente, ripeti il punto 1 utilizzando l'attuale risultato parziale come parte frazionaria
 - (c) Se il risultato ottenuto è **minore di 1**, allora ripeti il punto 1 utilizzando l'attuale risultato parziale come parte frazionaria
2. Una volta che la parte frazionaria non sia più presente nel risultato parziale (dunque quando si arriva al punto 1.a), sarà necessario riscrivere partendo dall'alto tutte le **parti intere dei risultati parziali ottenuti** durante lo svolgimento dell'algoritmo

Per comprendere meglio questo processo, vediamo un esempio pratico dove convertiremo il numero 17.625_{10} in binario:

- La parte intera siamo già in grado di convertirla, quindi $17_{10} \Rightarrow 010001_2$
- Per la parte razionale, applichiamo l'algoritmo descritto precedentemente:
 1. $0.625 \cdot 2 = \textcolor{red}{1}.25 \Rightarrow (1.25 - 1 = 0.25)$
 2. $0.25 \cdot 2 = \textcolor{red}{0}.5$
 3. $0.5 \cdot 2 = \textcolor{red}{1}$

Una volta raggiunto 1, consideriamo partendo dall'alto tutte le parti intere dei risultati parziali (segnate in rosso), ottenendo la parte razionale del nostro numero binario finale, ossia **.101**₂.

- L'ultimo passo è quello di unire la parte intera binaria a quella razionale, ottenendo che:

$$17.625_{10} = 010001.101_2$$

1.3.2 Numeri a virgola mobile

Così come i numeri a virgola fissa sono l'equivalente binario dei numeri razionali decimali, i numeri a virgola mobile sono l'equivalente binario della **notazione scientifica** dei numeri decimali.

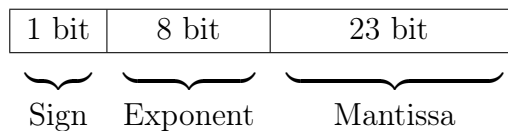
In linea generica, i numeri in notazione scientifica (sia decimale che binaria) vengono rappresentati nella seguente forma

$$\pm M \cdot B^E$$

dove:

- **M** = Mantissa
- **B** = Base
- **E** = Esponente

Poiché lavoriamo con definite quantità di bit, solitamente i numeri floating-point vengono rappresentati utilizzando **32 bit**, dove il **primo bit** rappresenta il *segno*, i successivi **8 bit** rappresentano l'*esponente* e i restanti **23 bit** rappresentano la *mantissa*.



Per convertire un numero decimale nella sua forma floating-point, è necessario seguire i seguenti passaggi:




1. Convertire il numero in un binario **fixed-point unsigned**, poichè il segno sarà rappresentato dal primo bit del float
2. **Riscrivere il numero** in "notazione scientifica binaria" portando la virgola fino ad una sola unità
3. Riscrivere il *segno*, la *mantissa* e l'*esponente* nei loro **campi equivalenti** della rappresentazione floating-point,

Riprendendo l'esempio fatto per la conversione in fixed-point nella sezione [1.3.1](#), convertiamo il numero 17.625_{10} in floating-point:

1. $17.625_{10} = 10001.101_2$
2. $10001.101_2 = 1.0001101_2 \cdot 2^4$

3. Converto i tre campi necessari in binario per poi riscriverli nei campi rispettivi:

- $Sign = 0_2$
- $Exponent = 100_2$
- $Mantissa = 10001101_2$

0	00000100	100011010000000000000000
		
Sign	Exponent	Mantissa

Facendo attenzione al **secondo passaggio** del procedimento, possiamo notare come **qualsiasi numero** venga rappresentato in notazione scientifica binaria abbia un 1 come unità a cui viene attribuita la virgola:

- $17.625_{10} = 10001.101_2 = 1.0001101_2 \cdot 2^4$
- $-58.25_{10} = 111010.01_2 = 1.1101001_2 \cdot 2^5$
- $228_{10} = 11100100_2 = 1.1100100_2 \cdot 2^7$

Per convenzione, quindi, nel momento in cui andiamo a riempire i rispettivi campi del floating-point possiamo direttamente **ignorare** l'1 prima della virgola, andando a guadagnare un bit aggiuntivo da poter utilizzare per rappresentare le cifre *dopo* la virgola:

$$17.625_{10} = 10001.101_2 = 1.0001101_2 \cdot 2^4$$

0	00000100	100011010000000000000000
---	----------	--------------------------

 \Rightarrow Senza ignorare l'1 davanti la virgola

0	00000100	000110100000000000000000
---	----------	--------------------------

 \Rightarrow Ignorando l'1 davanti la virgola

Nonostante questo passaggio possa sembrare superfluo, esso è **estremamente utile**, poiché nel caso in cui dovessimo rappresentare in float un numero in notazione scientifica binaria con 23 bit **dopo** la virgola, saremmo costretti a dover **tagliare** una parte di precisione scartando il LSB, poiché non possiamo inserire 24 bit nello spazio di 23 predisposto dalla mantissa.

Lo Standard IEEE 754

Ora che abbiamo guadagnato un bit aggiuntivo da poter usare per la precisione della mantissa, resta solo una problematica da risolvere: come possiamo rappresentare i numeri in notazione scientifica binaria con **esponente negativo**?

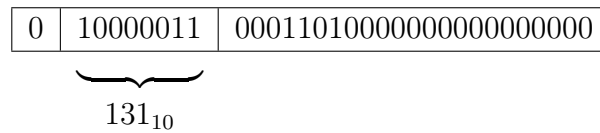
Lo **Standard IEEE 754**, il più utilizzato, propone una soluzione semplice: aggiungiamo un **bias di 127 all'esponente**.

Riprendiamo ancora una volta l'esempio della sezione [1.3.1](#):

$$17.625_{10} = 10001.101_2 = 1.0001101_2 \cdot 2^4$$

Una volta portato il numero in questa forma, aggiungiamo **127** al suo esponente, per poi convertirlo nel formato floating-point (ignorando il bit prima della virgola della mantissa):

$$17.625_{10} = 10001.101_2 = 1.0001101_2 \cdot 2^{(4+127)} = 1.0001101_2 \cdot 2^{131}$$



Grazie al bias di 127, possiamo scrivere anche i numeri con un esponente negativo **fino a -127** (poiché aggiungendo il bias l'esponente finale sarebbe 0, quindi rappresentabile positivamente). Tuttavia, perdiamo anche l'uso di una parte di esponenti positivi, poiché il valore massimo rappresentabile in 8 bit è 255.

Casi speciali, Tipi di precisione e rappresentazione in Esadecimale

Nel caso dei numeri floating-point, sono stati stabiliti dei **casi speciali** per poter rappresentare dei valori particolari o non esprimibili numericamente:

Numero Equivalente	Segno	Esponente	Mantissa
0	1 o 0	00000000	000000000000000000000000
∞	0	11111111	000000000000000000000000
$-\infty$	1	11111111	000000000000000000000000
NaN	1 o 0	00000000	diverso da 0
Num. Denormali	1 o 0	00000000	$(-1)^S \cdot 2^{-126} \cdot 0.m$ con $m \neq 0$

Inoltre, esistono vari **tipi di precisione** possibili per i numeri float in base al numero di bit utilizzati per rappresentarli:

Tipologia	Bit Tot.	Segno	Esponente	Mantissa	Bias
Half-precision	16 bit	1 bit	5 bit	10 bit	15
Single-precision	32 bit	1 bit	8 bit	23 bit	127
Double-precision	64 bit	1 bit	11 bit	52 bit	1023

L'utilità di avere più tipi di precisione è per prevenire la possibilità di un **overflow** o un **underflow** (ossia quando il numero è troppo piccolo per essere rappresentato), poiché altrimenti sarebbe necessario **arrotondare** il valore, perdendo una parte di precisione (guarda [questo video](#) se sei interessato ad una spiegazione dettagliata)

Un ulteriore modalità di **rappresentazione** dei numeri float è tramite l'uso dei numeri decimali. Poiché 32 bit corrispondono a 8 gruppi da 4 bit, possiamo riscrivere ogni numero float come **8 numeri esadecimali**, utilizzando il metodo visto nella sezione [1.1.1](#):

$$17.625_{10} = \underbrace{0}_{4} \underbrace{100}_{1} \underbrace{0001}_{8} \underbrace{1}_{D} \underbrace{000}_{0} \underbrace{0000}_{0} \underbrace{0000}_{0} \underbrace{0000}_{0} \text{ float} \Rightarrow 0x418D0000$$

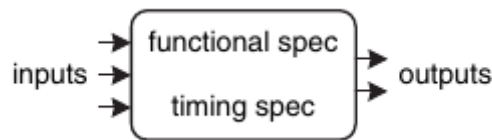
Capitolo 2

Circuiti combinatori

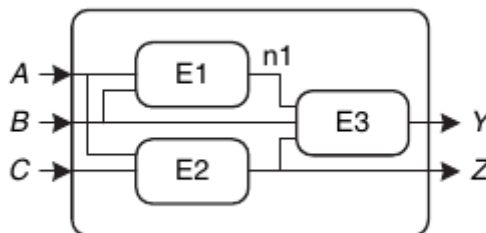
2.1 Definizioni

Nell'elettronica digitale, un **circuito logico** è una rete di dispositivi elettronici che processa un **valore discreto**. Ogni circuito è provvisto di un insieme di **input** che vengono processati all'interno di esso, restituendo uno o più **output**.

Oltre ai suoi input e ai suoi output, ogni circuito viene descritto da una **specificazione funzionale**, che esprime il modo in cui il circuito agisce sugli input per generare gli output, e una **specificazione temporale**, che esprime il tempo impiegato dal circuito per processare gli input e generare gli output.



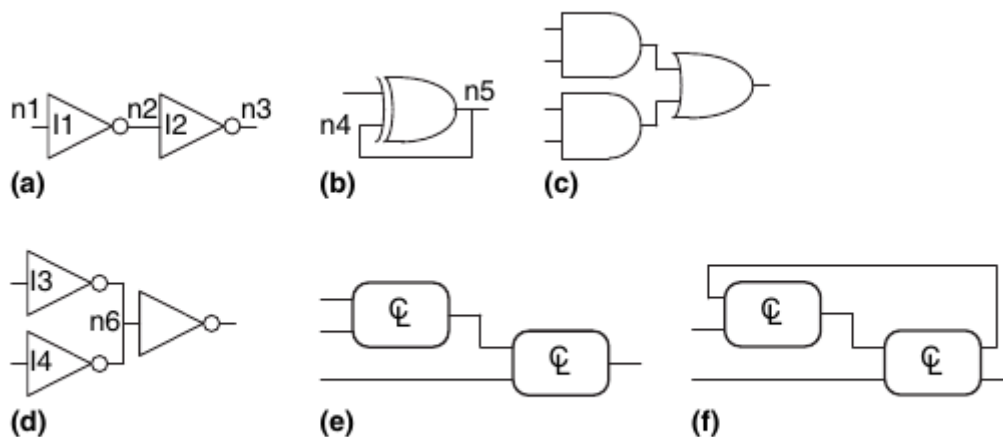
I circuiti logici, quindi, possono essere visti come delle **scatole nere** di cui è necessario tener conto solo degli input che vengono forniti e degli output che vengono restituiti. Al loro interno, i circuiti logici sono composti da **nodi** e **elementi** (E1, E2, E3, ...), che spesso sono a loro volta dei circuiti logici. I nodi, invece, sono dei fili elettrici il cui voltaggio trasporta un valore discreto. Possono essere dei nodi di *input* (A, B, C, ...), di *output* (Y, Z, ...) o *interni* al circuito stesso (n1, ...), ricevendo e consegnando dati da e al **mondo esterno**.



I circuiti logici possono essere divisi in due macro-categorie: circuiti **combinatori** e circuiti **sequenziali**. I primi sono **senza memoria**, dunque il valore degli output dipende solo dagli input e dal modo in cui vengono elaborati al suo interno. I secondi, invece, sono dotati di **memoria**, dunque il valore degli output dipende sia dal valore degli input nel momento precedente all'elaborazione, sia dal valore che hanno assunto nelle precedenti elaborazioni. In questo capitolo, vedremo prima il funzionamento e le regole che determinano i circuiti combinatori, per poi passare a quelli sequenziali.

Per poter essere definito come combinatorio, un circuito logico deve rispettare determinate **caratteristiche**:

- Ogni elemento del circuito deve essere a sua volta *combinatorio*
- Ogni nodo è un *input* oppure si connette ad esattamente *un output*
- Nel circuito non sono presenti dei collegamenti *ciclici*



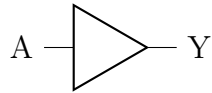
Nella foto mostrata, solo i circuiti *a*, *c* ed *e* sono combinatori, poiché rispettano tutte le caratteristiche precedentemente elencate.

2.2 Porte logiche

Le **porte logiche** sono la forma più semplice di circuiti logici, poiché la loro funzione prevede il ricevere in input uno o più valori **valori binari** per produrre un output a sua volta binario. Poiché si tratta di circuiti digitali, e dunque di elettronica, il valore assunto dagli input e dagli output è descritto da due semplici relazioni: se vi è **passaggio di corrente** (indipendentemente dall'intensità) allora il nodo assume valore 1, mentre in sua **assenza** assume valore 0.

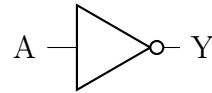
La relazione tra gli input e gli output delle porte logiche vengono descritte tramite delle **tabelle della verità** e tramite le **equazioni Booleane** (derivanti da Boole, inventore di questo tipo di algebra), ossia un'espressione matematica che utilizza solo valori binari.

Di seguito, vedremo una carrellata contenente tutte le porte logiche rappresentate graficamente, con annesse tabelle della verità ed equazioni booleane:

BUFFER

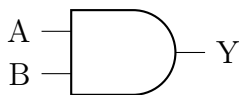
$$Y = A$$

A	Y
0	0
1	1

NOT

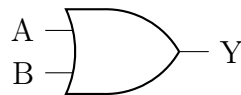
$$Y = \overline{A}$$

A	Y
0	1
1	0

AND

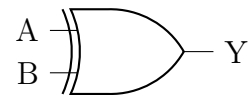
$$Y = AB$$

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

OR

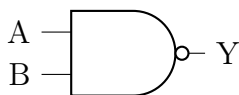
$$Y = A + B$$

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

XOR

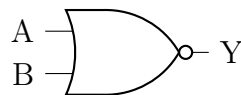
$$Y = A \oplus B$$

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

NAND

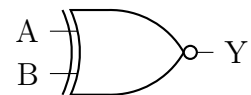
$$Y = \overline{AB}$$

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

NOR

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

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

XNOR

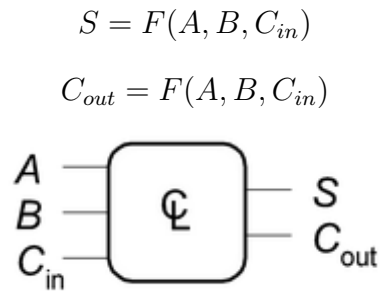
$$Y = \overline{A \oplus B}$$

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

2.3 Equazioni booleane

Una volta appresi i concetti base dei circuiti logici e il funzionamento delle porte logiche, possiamo di ciò che descrive dettagliatamente le **specifiche funzionali** di circuito logico, ossia le già citate **equazioni booleane**.

Il seguente circuito logico (CL) è dotato di tre input (A, B e C_{in}) e due output (S, C_{out}), e può essere descritto sia in forma scritta sia in forma grafica:



Per descrivere le specifiche funzionali del circuito, è necessario definire le equazioni booleane che mettono in **relazione** gli input con gli output. In questo esempio, andremo a descrivere le due funzioni che generano gli output S e C_{out} come:

$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$

In aggiunta, possiamo anche definire la **tabella della verità** corrispondente all'equazione descritta:

Inputs			Middle terms			Outputs	
A	B	C_{in}	AB	AC_{in}	BC_{in}	S	C_{out}
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	0	0	1	0
0	1	1	0	0	1	0	1
1	0	0	0	0	0	1	0
1	0	1	0	1	0	0	1
1	1	0	1	0	0	0	1
1	1	1	1	1	1	1	1

Senza saperlo, abbiamo appena descritto completamente un **sommatore ad 1 bit con riporto**, dove S corrisponde al valore ottenuto dalla somma dei bit A, B e C_{in} , ossia il possibile riporto di una precedente somma, mentre C_{out} corrisponde al riporto generato dalla somma stessa.

Nel caso non sia chiaro il suo funzionamento, nessun problema: questo componente fondamentale **verrà analizzato in uno stato molto avanzato del corso**.

Prima di poter parlare dettagliatamente di equazioni booleane, è necessario introdurre una serie di **terminologie**:

- **Complement**: l'opposto stretto di una variabile $\Rightarrow \bar{A}$ è il complemento di A
- **Literal**: una variabile o un suo complemento $\Rightarrow A, \bar{A}, B, \bar{B}$
- **Implicant**: un prodotto tra letterali $\Rightarrow \bar{A}\bar{B}, \bar{A}\bar{C}, ABC$
- **Minterm**: un prodotto tra tutte le variabili in input $\Rightarrow ABC, \bar{A}\bar{B}C, \bar{A}B\bar{C}$
- **Maxterm**: una somma tra tutte le variabili in input $\Rightarrow A+B+C, A+\bar{B}+C, \overline{A+B+C}$

2.3.1 Somma di prodotti (SOP) e Prodotto di somme (POS)

Due ulteriori modi per rappresentare le equazioni booleane è attraverso le *forma canoniche* **Sum-of-Products (SOP)** e **Product-of-Sums (POS)**. È opportuno sottolineare che **ogni** equazione booleana può essere definita in queste due forme.

Nella forma **SOP**, ad ogni riga della tabella della verità, descrivente il rapporto tra input e output, corrisponde un **minterm** tra le variabili della funzione, dove gli zeri assumono la forma di **complementare**, mentre gli uno non vengono negati. Infine, la funzione descritta dall'equazione booleana viene rappresentata come la **somma dei minterm** il cui output è **1**.

Nella forma **POS**, invece, ad ogni riga della tabella della verità corrisponde un **maxterm** tra le variabili della funzione, dove gli uno assumono la forma di **complementare**, mentre gli zeri non vengono negati. Infine, la funzione descritta dall'equazione booleana viene rappresentata come il **prodotto dei maxterm** il cui output è **0**.

Proviamo a descrivere l'equazione $Y = A \oplus B$ prima in forma SOP e poi in forma POS:

A	B	Minterm	Maxterm	Y
0	0	$\bar{A} \cdot \bar{B}$	$A + B$	0
0	1	$\bar{A} \cdot B$	$A + \bar{B}$	1
1	0	$A \cdot \bar{B}$	$\bar{A} + B$	1
1	1	$A \cdot B$	$\bar{A} + \bar{B}$	0

$$\text{Forma SOP} \Rightarrow Y = F(A, B) = \bar{A}B + A\bar{B} = \Sigma(1, 2)$$

$$\text{Forma POS} \Rightarrow Y = F(A, B) = (A + B)(\bar{A} + \bar{B}) = \Pi(0, 3)$$

In seguito, vedremo l'utilità di poter riscrivere ogni equazione booleana in termini di somme e prodotti al fine di poter semplificare le equazioni ai **minimi implicant** necessari

2.3.2 Algebra di Boole

L'algebra introdotta dal matematico George Boole, seppur molto poco utilizzata nel periodo in cui venne introdotta, trovò un'applicazione estremamente efficiente nel campo dell'elettronica digitale. Proprio come nella normale algebra, Boole introdusse una serie di **assiomi** e **teoremi** che dettano il rapporto tra i valori discreti 0 ed 1.

Tramite questi assiomi e teoremi, è possibile **semplificare** notevolmente le equazioni booleane descriventi i circuiti logici. La caratteristica fondamentale di essi è la loro **dualità**, ossia la completa *intercambiabilità* tra AND (\cdot) ed OR ($+$) e tra 1 e 0, senza alterare la veridicità dell'assioma o del teorema.

Assiomi e Teoremi booleani

Nome	Assioma/Teorema	Assioma/Teorema duale
Assioma 1	$A = 0 \Leftrightarrow A \neq 1$	$A = 1 \Leftrightarrow B \neq 0$
Assioma 2	$\overline{0} = 1$	$\overline{1} = 0$
Assioma 3	$0 \cdot 0 = 0$	$1 + 1 = 1$
Assioma 4	$1 \cdot 1 = 1$	$0 + 0 = 0$
Assioma 5	$0 \cdot 1 = 1 \cdot 0 = 0$	$1 + 0 = 0 + 1 = 1$
T. d'identità	$A \cdot 1 = A$	$A + 0 = A$
T. di elemento nullo	$A \cdot 0 = 0$	$A + 1 = 1$
T. d'idempotenza	$AA = A$	$B + B = B$
T. d'involuzione	$\overline{\overline{B}} = B$	$\overline{\overline{B}} = B$
T. del complementare	$\overline{B}B = 0$	$\overline{B} + B = 1$
P. commutativa	$AB = BA$	$A + B = B + A$
P. associativa	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
P. distributiva	$A(B + C) = AB + AC$	$A + BC = (A + B)(A + C)$
1° T. dell'assorbimento	$A + AB = A$	$A(A + B) = A$
2° T. dell'assorbimento	$A + \overline{A}B = A + B$	$A(\overline{A} + B) = AB$
3° T. dell'assorbimento	$AB + \overline{A}B = B$	$(A + B)(\overline{A} + B) = B$
T. di DeMorgan	$\overline{AB} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A} \cdot \overline{B}$

Possiamo provare la **veridicità** di questi teoremi in due modalità confrontando per confronto le tavole della verità di entrambi i membri dell'equazione, oppure usando reciprocamente i **teoremi** e gli **assiomi**

Una volta appresi gli assiomi e i teoremi dell'algebra booleana, possiamo metterli in pratica cercando di semplificare l'equazione $Y = ABC + \overline{A}\overline{B} + \overline{A}$

1. Applico il teorema dell'**assorbimento** $\Rightarrow ABC + \overline{A}\overline{B} + \overline{A} = ABC + \overline{A}$
2. Applico la proprietà **distributiva** $\Rightarrow ABC + \overline{A} = (A + \overline{A})(B + \overline{A})(C + \overline{A})$
3. Applico il teorema del **complementare** $\Rightarrow (A + \overline{A})(B + \overline{A})(C + \overline{A}) = (B + \overline{A})(C + \overline{A})$
4. Applico la proprietà **distributiva** all'inverso $\Rightarrow (B + \overline{A})(C + \overline{A}) = \overline{A} + BC$

Concludiamo quindi che: $Y = ABC + \overline{A}\overline{B} + \overline{A} = \overline{A} + BC$

Per verificare che la semplificazione sia effettivamente valida e non errata, possiamo definire la tabella della verità rappresentante entrambe le due forme dell'equazione:

A	B	C	$ABC + \overline{A}\overline{B} + \overline{A}$	$\overline{A} + BC$
0	0	0	1	1
0	0	1	1	1
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	1	1	1

Proviamo ora invece a semplificare l'equazione $Y = (A + BC)(A + DE)$. In questo caso, possiamo applicare la proprietà distributiva in due modi diversi:

1. Duale della p. distributiva in modo inverso:

- $(A + BC)(A + DE) = A + (BC \cdot DE) = A + BCDE$

2. Distribuisco $(A + BC)$ in $(A + DE)$ per poi applicare il duale del primo teorema dell'assorbimento:

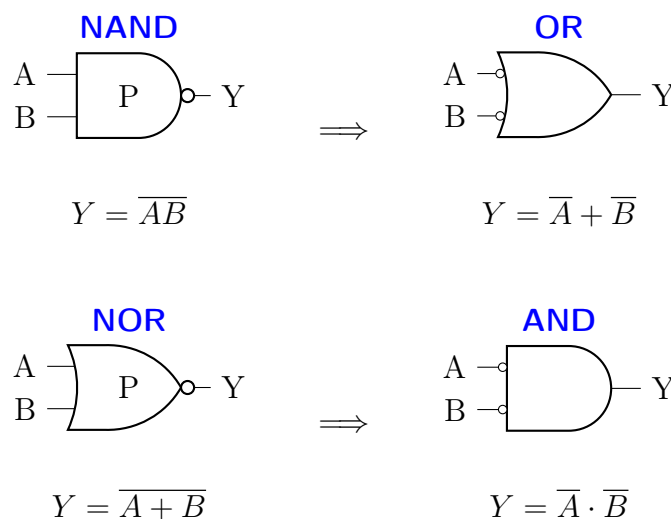
- $(A + BC)(A + DE) = AA + ADE + ABC + BCDE = A + ADE + ABC + BCDE$

- $AA + ADE + ABC + BCDE = A + ABC + BCDE = A + BCDE$

Il Teorema di DeMorgan e il Bubble Pushing

Il **Teorema di DeMorgan** stabilisce che il **complemento dei prodotti** equivale alla **somma dei complementi** e che di conseguenza il suo duale afferma che il **complemento della somma** equivale al **prodotto dei complementi**.

Nonostante possa sembrare poco intuitivo, questo teorema può essere anche rivisitato in **forma grafica**, rendendo la sua interpretazione più semplice:

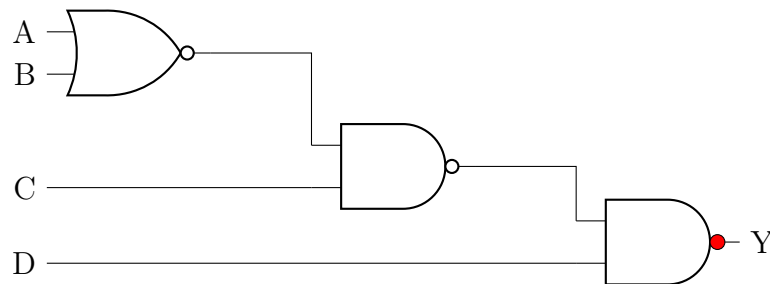


Come possiamo notare nell'immagine precedente, l'applicazione del teorema di DeMorgan corrisponde all'invertire un AND con un OR (o viceversa) e allo spostare la *bolla* del NOT dall'**output agli input** (o viceversa).

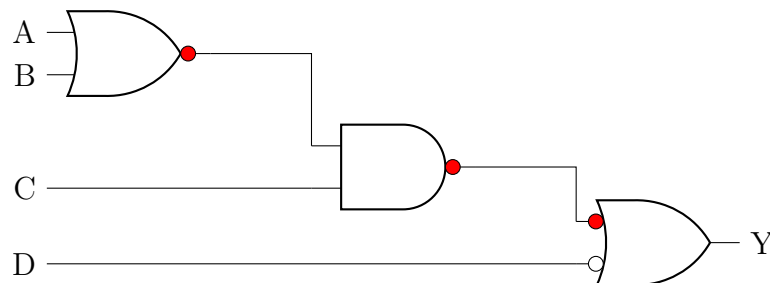
Per via di quest'ultima impostazione a livello grafico, nell'ambito della progettazione dei sistemi digitali è nata una *convenzione*, chiamata **bubble pushing**, dove si cerca di spostare tutte le bolle NOT presenti nel circuito da sinistra verso destra (**forward bubble pushing**) oppure da destra verso sinistra (**backwards bubble pushing**), rendendo più fluido il processo di semplificazione.

Proviamo a semplificare il seguente circuito utilizzando il **backwards bubble pushing**:

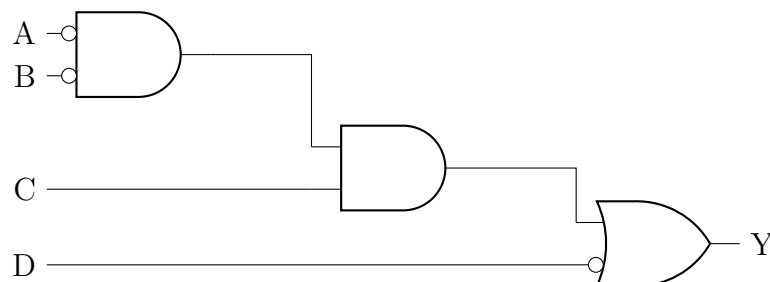
1. Partendo da destra, utilizziamo il **bubble pushing** sull'output della porta NAND finale



2. Successivamente, possiamo **semplificare** il NOT in uscita dalla porta NAND centrale con il NOT in entrata alla porta OR appena modificata. Inoltre, possiamo applicare nuovamente il **bubble pushing** sull'output della porta NOR iniziale.



3. Una volta effettuati i passaggi, abbiamo ottenuto una versione semplificata del circuito, dove abbiamo applicato il **teorema di DeMorgan** in forma grafica



Completezza delle porte

Una volta discussi tutti gli assiomi e teoremi fondanti dell'algebra di Boole, è possibile notare una sua particolare caratteristica, ossia la **completezza funzionale** posseduta dalle porte **NAND** e **NOR**, poiché **tutte le porte** sono realizzabili tramite l'uso di solo queste due semplici porte e l'applicazione dei teoremi su di essi:

Porta	Soli NAND	Soli NOR
\overline{A}	$\overline{A \cdot A}$	$\overline{A + A}$
$A \cdot B$	$\overline{\overline{A \cdot B} \cdot \overline{A \cdot B}}$	$\overline{\overline{A + A} + \overline{B + B}}$
$A + B$	$\overline{\overline{A \cdot A} \cdot \overline{B \cdot B}}$	$\overline{A + B + A + B}$
...

2.4 Dalla logica ai circuiti

2.4.1 Regole di lettura

Nell'ambito dei circuiti, esistono alcune **regole di lettura generali** adottate come standard:

- Gli input partono da **sinistra** (o dall'alto)
- Gli output vanno verso **destra** (o verso il basso)
- Le porte vanno da **destra a sinistra**
- Si preferisce l'uso di fili **non curvi** per facilitare la lettura dei collegamenti
- I fili che formano un **incrocio a T** sono connessi tra loro
- I fili che formano un **incrocio normale** non sono connessi tra loro
- I fili che formano un **incrocio normale** ed hanno un **pallino** al centro sono connessi tra loro

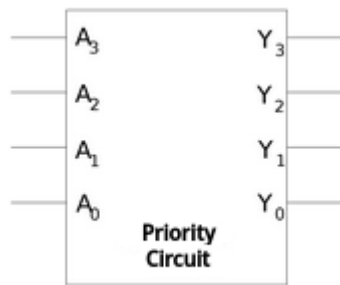
2.4.2 Output multipli, X e Z

Fin'ora abbiamo visto solo circuiti ad output singolo, tuttavia in elettronica digitale esiste una grande gamma di circuiti ad **output multiplo**.

In particolare, di seguito vedremo il funzionamento di un **circuito prioritario**, il quale, dati **N input**, restituisce **N output**, dove però **solo un output** assume valore 1, mentre tutti gli altri assumono valore 0. Quale tra gli N output sarà vero viene stabilito in base al **valore vero più significativo** assunto dagli input.

Immaginando un circuito composto dagli input $A_3 A_2 A_1 A_0$ e gli output $Y_3 Y_2 Y_1 Y_0$, nel caso in cui A_3 sia **vero** allora l'output Y_3 sarà **vero**, mentre tutti gli altri saranno **falsi**, poiché A_3 è il **valore vero più significativo** in input (dunque ha la precedenza sugli altri). Nel caso in cui A_3 sia **falso**, il controllo sul valore vero più significativo verrà effettuato sull'*input successivo* nell'ordine di **precedenza**, ossia A_2 , rendendo l'output Y_2 vero (e tutti gli altri falsi) nel caso in cui A_2 sia vero. Questo processo di selezione dell'output in base alla precedenza viene effettuato su ogni input, fino a raggiungere quello con precedenza più bassa (nel nostro esempio A_0).

Per capirne meglio l'esempio, vediamo una schematica del circuito assieme alla sua tabella della verità sviluppata in base al **funzionamento** stesso del circuito che abbiamo precedentemente descritto:



A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

Per via del modo in cui vengono stabiliti i valori degli output, possiamo direttamente **ignorare** il valore assunto da tutti gli input **meno significativi** rispetto all'input con il valore vero più significativo. In questo caso, i valori ignorati vengono definiti con una **X** e vengono chiamati valori **don't care** (dall'inglese: di non interesse).

Possiamo quindi riscrivere la tabella indicando anche quali siano i valori don't care presenti nella logica del circuito, per poi semplificarla in una sua versione ristretta:

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
0	1	X	X	0	1	0	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0
1	X	X	X	1	0	0	0
1	X	X	X	1	0	0	0
1	X	X	X	1	0	0	0
1	X	X	X	1	0	0	0
1	X	X	X	1	0	0	0
1	X	X	X	1	0	0	0
1	X	X	X	1	0	0	0

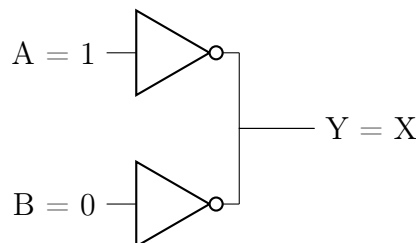
\Rightarrow

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

Occhio alle X e alle Z!

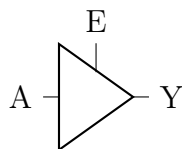
Nell'ambito dei circuiti, una X **non** viene utilizzata **solo** nel caso in cui ci si trovi davanti ad un valore **don't care**, bensì viene utilizzata per indicare la presenza di un **valore conteso (o ignoto)** tra uno 0 ed un 1, ossia la presenza di un **errore** nella progettazione di un circuito.

- **Contesa:** il circuito tenta di generare in output sia uno 0 che un 1
- Il vero valore assunto è **tra 0 ed 1**, ma potrebbe essere interpretato come uno 0, un 1 oppure una zona proibita
- Il risultato potrebbe **cambiare** in base a voltaggio, temperatura, tempo e rumore magnetico
- Spesso causa un'eccessiva dissipazione di energia



Tuttavia, esiste anche **un'altra casistica** in cui un output può assumere un valore pari a 0, ad 1 o tra 0 ed 1, ossia nel caso in cui si presenti un **alta impedenza (o floating value)**, che viene indicata con il valore **Z**.

A differenza del valore conteso, il valore assunto da una Z può essere **cambiato** nel caso in cui ci sia un **altro segnale** elettrico connesso ad esso (indicato con E). In questo caso, si parla di un **buffer tristate (o triplo stato)**, dove l'output può assumere valore 0, 1 o Z a seconda dello **stato del secondo segnale**:



E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

In un buffer tristate, quindi, nel caso in cui il segnale di controllo E **non sia attivo**, allora si avrà un **valore Z** come output, mentre nel caso in cui E **sia attivo**, allora l'output sarà uguale al **valore di input** (ossia A).

2.4.3 Le Mappe di Karnaugh (K-Maps)

Per facilitare l'applicazione dei due **teoremi dell'assorbimento** previsti dall'algebra booleana, il matematico Maurice Karnaugh inventò un metodo grafico costituito da una **mappatura** dei mintermini e dei maxtermini di un'equazione booleana, dove, seguendo delle semplici regole, è possibile semplificare ai minimi termini quest'ultima. Questa rappresentazione grafica viene chiamata **Mappa di Karnaugh (o K-Maps)**.

La struttura delle K-map prevede la **traslazione** degli output di una funzione, descritti all'interno della sua tabella della verità, in una mappatura che mette a confronto i **termini simili tra loro**.

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

		AB			
		00	01	11	10
C	0	1	0	0	0
	1	1	0	0	0

		AB			
		00	01	11	10
C	0	$\overline{A}BC$	$\overline{A}B\overline{C}$	$AB\overline{C}$	$A\overline{B}\overline{C}$
	1	$\overline{A}B\overline{C}$	$\overline{A}BC$	ABC	$A\overline{B}C$

Normalmente, potremmo riscrivere questa funzione in **forma SOP** (o POS) utilizzando i mintermini (o maxtermini) descritti nella sua tabella della verità, ottenendo l'equazione $Y = \overline{A} \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot \overline{B} \cdot C$. Successivamente, potremmo applicare il **teorema dell'assorbimento** per semplificare l'equazione ottenuta, ottenendo l'equazione finale $Y = \overline{A} \cdot \overline{B}$.

Guardando le K-map descritte sopra, possiamo notare la vicinanza tra i due mintermini che assumono valore 1, ossia i mintermini nelle posizioni (AB=00; C=0) e (AB=00, C=1). Confrontando questi due mintermini, possiamo notare come l'unica **differenza tra essi** sia il literal definito da C. Possiamo quindi **raggruppare** graficamente sulla mappa questi due mintermini e considerare **solo** la loro **parte in comune**, ossia l'**implicante** $\overline{A} \cdot \overline{B}$. Poiché esso è l'unico implicante presente nella mappa, possiamo direttamente scrivere l'equazione minima come $Y = \overline{A} \cdot \overline{B}$.

		AB			
		00	01	11	10
C	0	1	0	0	0
	1	1	0	0	0

		AB			
		00	01	11	10
C	0	$\overline{A}BC$	$\overline{A}B\overline{C}$	$AB\overline{C}$	$A\overline{B}\overline{C}$
	1	$\overline{A}B\overline{C}$	$\overline{A}BC$	ABC	$A\overline{B}C$

In questo modo, abbiamo applicato il teorema dell'assorbimento in un modo estremamente semplificato, permettendoci di ottenere direttamente la **forma minima possibile** con cui è possibile rappresentare l'equazione descritta nella tabella della verità.

Regole per i raggruppamenti

- Ogni 1 deve essere raggruppato **almeno una volta**, anche se solo con se stesso (implicante ad 1 variabile)
- Ogni raggruppamento può estendersi **solo** per una quantità di quadrati pari ad una **potenza di 2** (1, 2, 4, ...) in ogni direzione
- Ogni raggruppamento deve essere **il più grande possibile**, in modo da poter ridurre ai termini minimi possibili. Inoltre, il raggruppamento più grande presente in una K-map viene chiamato **implicante primo**.
- I raggruppamenti possono estendersi **oltre i bordi**, ad esempio in una mappa a 3 valori è possibile raggruppare le celle (AB=0; C=0) e (AB=10; C=0) se entrambe presentano degli 1
- Un valore **don't care (X)** può essere raggruppato, ma solo se è utile per minimizzare l'equazione, altrimenti può essere ignorato

Le K-map sono **duali**, dunque possono essere utilizzate non solo per le forme SOP, ma anche per le **forme POS**: i raggruppamenti vengono effettuati tra gli **0** ed ogni quadrato rappresenta un **maxterm**.

Esempi di semplificazioni tramite le K-map

Ovviamente, applicare manualmente il teorema dell'assorbimento nell'esempio precedentemente visto risulta molto banale, dunque utilizzare le K-map potrebbe essere visto come una perdita di tempo. Tuttavia, esse risultano **estremamente comode** nel caso in cui ci si trovi dinanzi ad una funzione con molti output possibili, come nel seguente esempio:

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

		AB			
		00	01	11	10
CD	00	1	0	0	1
	01	0	1	0	1
	11	1	1	0	0
	10	1	1	0	1

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

Per comprendere meglio i passaggi effettuati per ottenere gli implicant che costituiscono l'equazione finale in base ai raggruppamenti effettuati, analizziamo quest'ultimi e i loro termini comuni:

- **Raggruppamento 1:** ognuno dei 4 implicant raggruppati contiene \overline{B} e \overline{D} , dunque l'implicante minimo corrispondente sarà $\overline{B}\overline{D}$
- **Raggruppamento 2:** ognuno dei 4 implicant raggruppati contiene \overline{A} e C , dunque l'implicante minimo corrispondente sarà $\overline{A}C$
- **Raggruppamento 3:** ognuno dei 2 implicant raggruppati contiene \overline{A} , B e D , dunque l'implicante minimo corrispondente sarà $\overline{A}BD$
- **Raggruppamento 4:** ognuno dei 2 implicant raggruppati contiene A , \overline{B} e \overline{C} , dunque l'implicante minimo corrispondente sarà $A\overline{B}\overline{C}$

K-Map con Don't care

Vediamo ora una situazione simile alla precedente, dove però alcuni output sono stati sostituiti ad un valore **don't care (X)**:

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	X
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

		AB			
		00	01	11	10
CD	00	1	0	X	1
	01	0	X	X	1
	11	1	1	X	X
	10	1	1	X	X

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

Notiamo come in questo caso una X non sia stata raggrupata. Il motivo è semplice: nel caso in cui venisse raggrupata si andrebbe ad un **mintermine eccessivo**, poiché tutti gli 1 sono già stati coperti (vedi le regole nella sezione 2.4.3).

K-map con Maxtermini

Le K-map possono essere utilizzate non solo per semplificare un'equazione in forma minima SOP, ma anche in **forma minima POS**.

Le regole in gioco sono le stesse ma a parti invertite: **vengono raggruppati gli 0** e gli elementi in comune vengono riscritti **sottoforma di maxtermini** (ad esempio: $AB = 10$, $CD = 00$ verrà riscritto come $(\overline{A} + B + C + D)$).

Nei casi in cui vi è una grande quantità di 1 all'interno della tavola della verità dell'equazione di cui vogliamo trovare la forma minima (come nell'esempio precedente), è conveniente utilizzare questa versione in forma POS delle K-map, poiché **più rapida**.

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	X
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

		AB			
		00	01	11	10
CD	00	1	0	X	1
	01	0	X	X	1
	11	1	1	X	X
	10	1	1	X	X

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

ATTENZIONE:, ricordiamo che la forma POS e la forma SOP sono **equivalenti**, dunque scegliere quale delle due "tecniche" applicare è indifferente sul risultato.

Tuttavia, notiamo come provando a convertire la forma POS ricavata in una forma SOP, **non otteniamo la stessa SOP** ricavata nell'esempio precedente.

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

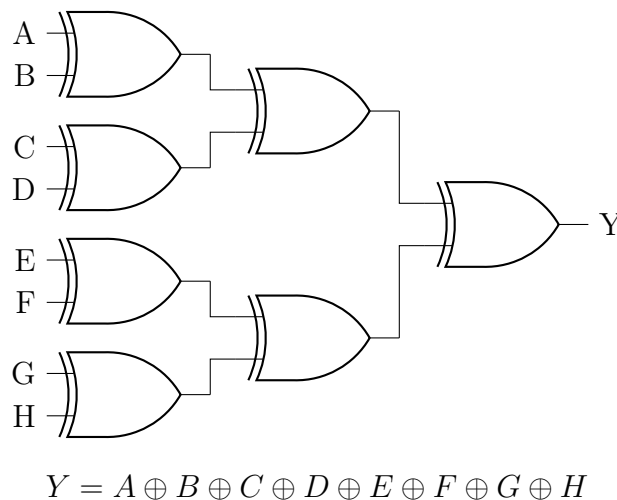
Ma come? Non dovrebbero essere equivalenti le due forme SOP e POS?

Niente panico: l'**incongruenza** è dovuta all'uso di alcuni **don't care diversi** tra i due raggruppamenti. Il funzionamento dell'equazione, tuttavia, è lo stesso, poiché ricordiamo che, per definizione stessa, **possiamo ignorare i casi in cui si verifica un termine don't care**.

2.4.4 Logica combinatoriale multi-livello

Nonostante siano molto semplici da realizzare, i circuiti con una **logica a due livelli** (ossia riducibili a forme SOP e POS) di grandi dimensioni richiedono molto più hardware per essere realizzati, dunque più costosi, rispetto a dei circuiti con **logica multi-livello**. Inoltre, tecniche come il *bubble pushing* risultano molto più utili su quest'ultimi, piuttosto che sui primi.

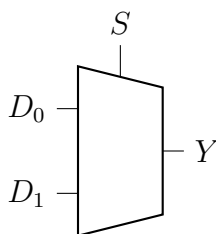
Basti immaginare come per costruire uno **XOR ad 8 input** servano 7 XOR a 2 input interconnessi tra loro, dove (ricordando che $Y = A \oplus B = \overline{A}B + A\overline{B}$) ognuno di quest'ultimi è composto a sua volta da **due AND** ed **un OR**. Per realizzare un singolo componente, dunque, la quantità di elementi necessari sarebbe **elevata** e poco gestibile.



Nei circuiti con **logica multi-livello**, i due componenti principali utilizzati che vedremo sono il **multiplexer** e il **decoder**.

Multiplexer

Il **multiplexer** (o **MUX**) è un componente che si occupa di **selezionare solo uno di N input**, collegandolo direttamente all'output. La selezione dell'input avviene tramite un ulteriore **input di controllo**, dove in base al suo stato viene deciso il valore assunto dall'output. Per capire meglio, vediamo il funzionamento di un **MUX 2:1**:



S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

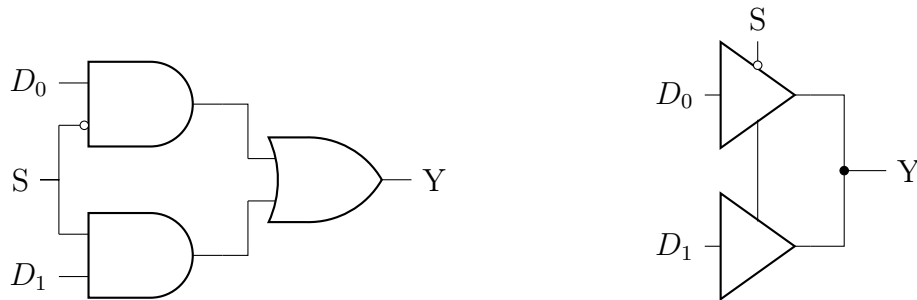
S	Y
0	D_0
1	D_1

Se $S = 0$, allora D_0 viene scelto

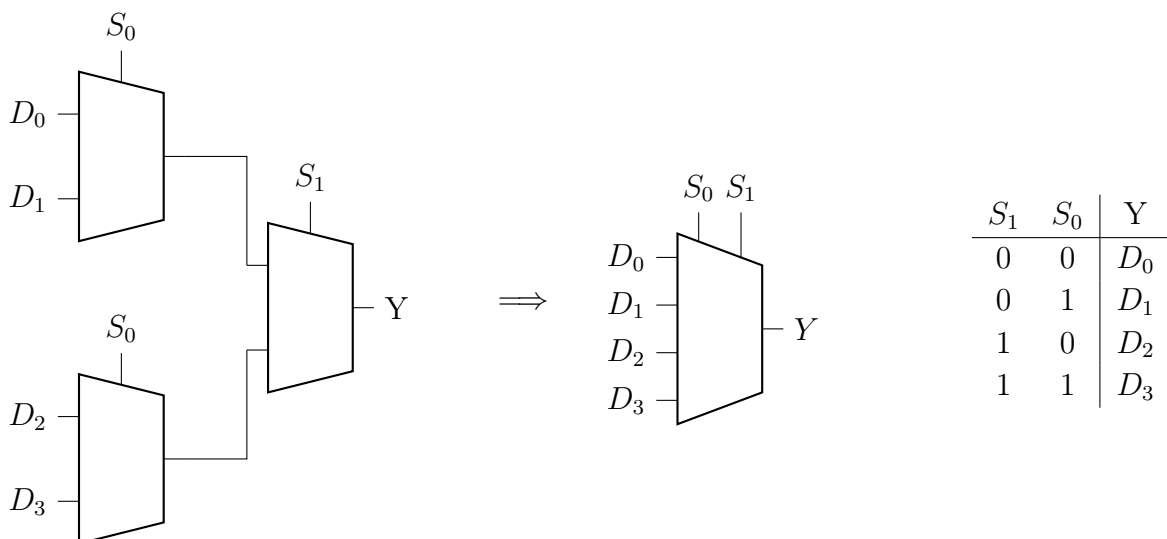
Se $S = 1$, allora D_1 viene scelto

Dunque, $Y = \overline{S}D_0 + SD_1$

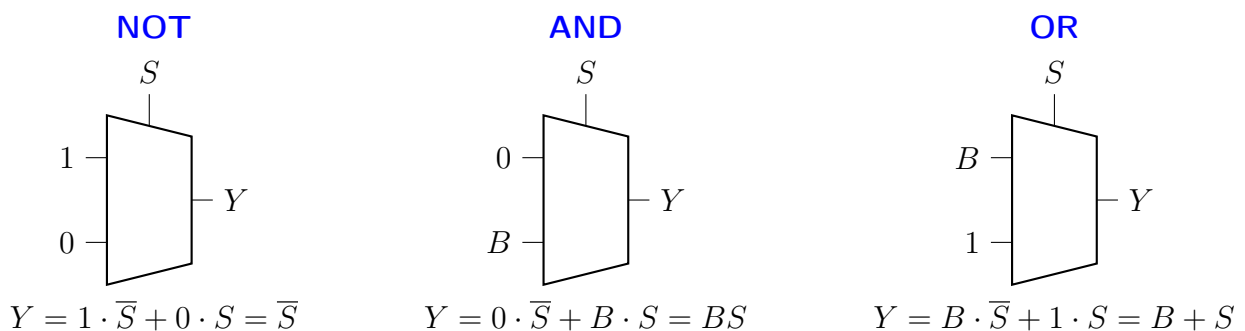
Come abbiamo visto, è possibile implementare un MUX 2:1 utilizzando **3 porte logiche**. Tuttavia, è possibile ridurre ancora il numero di componenti necessari, realizzando lo stesso MUX tramite **2 buffer tristate**:



Per via della sua semplicità come componente base, utilizzando tre MUX 2:1, possiamo realizzare ad esempio un **MUX 4:1**, dunque con 4 input ed 1 solo output:

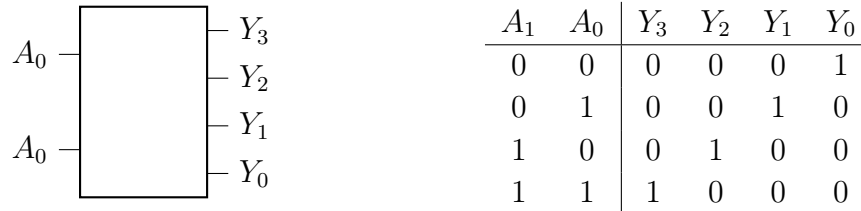


Un'altra importante caratteristica intrinseca al MUX, sempre poiché componente base, è la sua **completezza** analoga a quella delle porte NAND e NOR. Dunque, è possibile realizzare ogni singola porta logica utilizzando un MUX:



Decoder

Il **decoder** è un componente che si occupa di **selezionare solo uno tra 2^n output (codifica one-hot)** generati da n valori in input. Al contrario del MUX, la selezione non avviene tramite un aggiuntivo segnale di controllo, bensì essa è intrinseca al decoder stesso. Per capire meglio, vediamo il funzionamento di un **decoder 2:4**:



$$\left. \begin{aligned} Y_0 &= \overline{A_1} \overline{A_0} \\ Y_1 &= \overline{A_1} A_0 \\ Y_2 &= A_1 \overline{A_0} \\ Y_3 &= A_1 A_0 \end{aligned} \right\}$$
 Possiamo notare come gli output del decoder corrispondano a **tutti i mintermini** ottenibili con le variabili A_0 e A_1 . Per questo motivo, potremmo potenzialmente collegare una porta OR a due di questi output ed ottenere tutte le porte logiche (**completezza parziale**)

$$\text{es: } A_1 \oplus A_0 = Y_1 + Y_2 = \overline{A_1} A_0 + A_1 \overline{A_0}$$

Teorema di Shannon

Nell'ambito dei circuiti combinatori multilivello, il **Teorema di Shannon** afferma che data una funzione booleana f di n variabili, la seguente uguaglianza risulta valida:

$$f(x_1, x_2, \dots, x_n) = x_1 \cdot f(1, x_2, \dots, x_n) + \overline{x_1} \cdot f(0, x_2, \dots, x_n)$$

Questo teorema può essere utilizzato per riscrivere una funzione di n variabili come **2 funzioni da $n-1$ variabili**, in modo da poter implementare la funzione originale tramite un **MUX 2:1**.

Analisi e Sintesi

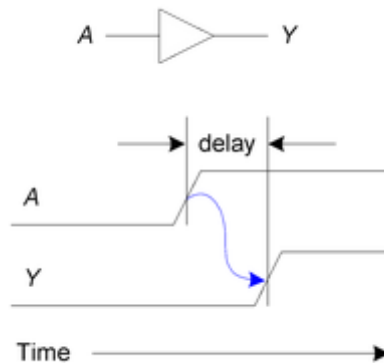
Giunti verso la fine del capitolo sui circuiti combinatori, è opportuno precisare due **terminologie**, in modo che non vengano confuse tra di loro:

- Per **analisi** di un circuito combinatorio si intende l'individuazione della funzione logica realizzata dal circuito, formulata come espressione booleana.
- Per **sintesi** di un circuito combinatorio si intende il disegno di un circuito combinatorio a partire da un'espressione booleana (o dalla tabella di verità).

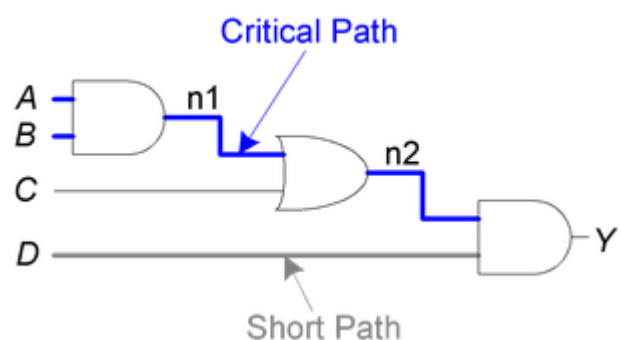
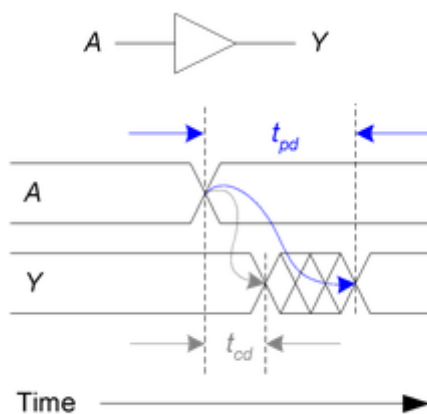
2.4.5 Timing dei circuiti

Fino ad ora abbiamo analizzato i vari circuiti combinatori solo da un punto di vista non del tutto realistico, ignorando un fattore molto importante, ossia il **tempo**.

In ogni componente di un circuito, è possibile identificare un **delay**, ossia il tempo trascorso dal variare dell'input al variare dell'output. Questo delay può dipendere da effetti dell'ambiente esterno, come i limiti della velocità della luce, o dal modo in cui è stato costruito il componente stesso, come la sua capacità e la sua resistenza.



Possiamo distinguere due categorie di delay: **delay di contaminazione** (t_{cd}) e **delay di propagazione** (t_{pd}). Il primo corrisponde al **delay minimo** che deve trascorrere affinché l'output possa essere considerato come variato (dunque non sappiamo ancora se sia effettivamente cambiato). Il secondo, invece, è il **delay massimo** che deve trascorrere affinché l'output sia considerabile come variato (dunque siamo sicuri al 100% del suo avvenuto cambiamento)



Durante la progettazione di circuiti, quindi, vengono favoriti gli **short path**, ossia quei percorsi dove vanno ad accumularsi il minor numero di delay, in modo che il circuito possa essere il più veloce possibile. Nell'esempio sulla destra (non legato a quello sulla sinistra) possiamo notare come A e B passino per un due porte aggiuntive rispetto a D, risultando in un **critical path** (o long path):

- Critical Path = $2 \cdot t_{pd_{AND}} + t_{pd_{OR}}$
- Short Path = $t_{pd_{AND}}$

Capitolo 3

Circuiti sequenziali

Come abbiamo accennato nella sezione [2.1](#), la caratteristica principale che distingue i **circuiti sequenziali** da quelli combinatori è la presenza di **memoria a breve termine**, ossia gli output del circuito dipendono sia dagli input correnti sia dagli output precedenti.

Vediamo ora alcune definizioni che verranno utilizzate nelle sezioni successive:

- **Stato**: ogni informazione riguardo un circuito necessaria per poter descrivere il suo futuro comportamento
- **Latch e Flip-Flop**: elementi di stato che memorizzano un bit rappresentante lo stato attuale del circuito
- **Circuiti sequenziali sincroni**: circuiti combinatori seguiti da un banco (ossia un insieme) di flip-flop

3.1 Elementi di stato

Gli **elementi di stato** sono quei componenti circuitali che si occupano di immagazzinare lo **stato attuale** di un circuito, in modo che quest'ultimo possa influenzare il futuro output del circuito stesso.

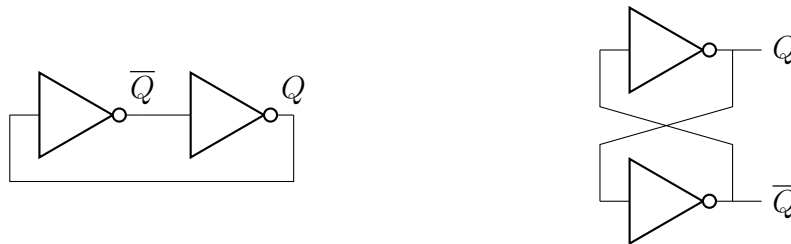
In questa sezione analizzeremo i seguenti elementi:

- **Circuito bistabile**
- **SR Latch**
- **D Latch**
- **D Flip-Flop e le sue varianti**
- **Ring Oscillator**

3.1.1 Circuito bistabile

Il **circuito bistabile** corrisponde al **blocco portante** tramite cui, imitando la sua logica, vengono realizzati gli altri elementi di stato. Questo circuito possiede **due output** (Q e \bar{Q}) ma, tuttavia, esso **non possiede input**, dunque non è un reale componente utilizzabile (motivo per cui gli altri elementi di stato **imitano** la sua logica).

Un circuito bistabile può essere implementato in due modi, in *forma elementare* (sinistra) o in *forma normale* (destra):

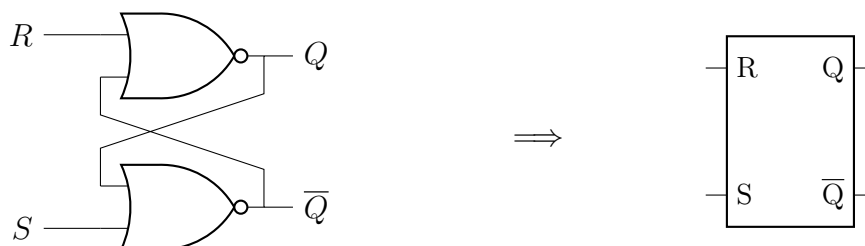


Immaginando che il segnale elettrico presente nel circuito sia un valore molto vicino a 5V (dunque 1), esso passerà attraverso i **due inverter** (le porte NOT), che lo renderanno prima pari a un valore molto vicino a 0V (dunque 0) e poi nuovamente un valore ancora più vicino a 5V (dunque sempre 1). Nel caso in cui il segnale elettrico sia un valore molto vicino a 0V, esso verrà prima trasformato in un valore molto vicino a 5V e poi nuovamente in un valore ancora più vicino a 0V.

Poiché non vi sono interferenze esterne, questo ciclo di avvicinamento costante a 5V e 0V è **perpetuo**, rendendo il circuito bistabile in grado di conservare il suo **stato**.

3.1.2 SR Latch

Una volta vista la logica alla base degli elementi di stato, possiamo introdurre il primo elemento realmente utilizzabile, ossia il **Set/Reset Latch**. A differenza dei circuiti bistabili, questo componente è dotato di due porte NOR (al posto delle due porte NOT) e di **due input**, chiamati **Set** e **Reset**, che permettono di **modificare** lo stato del circuito.



Per capire il funzionamento di un SR Latch, analizziamo i seguenti **quattro casi**:

- Nel caso in cui **S valga 1 e R valga 0**, il valore della porta NOR *inferiore* sarà **0**, indipendentemente dal valore assunto dal secondo input (ossia Q), mentre la porta NOR *superiore* assumerà valore **1**, poiché entrambi gli input (R e \bar{Q}) valgono 0.

- Nel caso in cui **S valga 0 e R valga 1**, il valore della porta NOR *superiore* sarà **0**, indipendentemente dal valore assunto dal secondo input (ossia \overline{Q}), mentre la porta NOR *inferiore* assumerà valore **1**, poiché entrambi gli input (S e Q) valgono 0.
- Nel caso in cui **S valga 0 e R valga 0**, il valore della porta NOR *inferiore* sarà **0** nel caso in cui Q valga 1, mentre sarà **1** nel caso in cui Q valga 0. Allo stesso modo, il valore della porta NOR *superiore* sarà **0** nel caso in cui \overline{Q} valga 1, mentre sarà **1** nel caso in cui \overline{Q} valga 0. In questo caso, dunque, il circuito **conserva il suo stato**, poiché il valore di Q e \overline{Q} dipendono strettamente dal loro valore precedente (che chiameremo Q_p).
- Nel caso in cui **S valga 1 e R valga 1**, il valore della porta NOR *inferiore* sarà **0**, indipendentemente dal valore assunto dal secondo input (ossia Q). Allo stesso modo, il valore della porta NOR *superiore* sarà **0**, indipendentemente dal valore assunto dal secondo input (ossia \overline{Q}). In questo caso, dunque, sia Q sia \overline{Q} assumono valore 0, rendendo lo stato del circuito **illegale**, poiché l'imposizione logica $Q \neq \overline{Q}$ verrebbe violata.

Possiamo quindi riassumere il funzionamento del circuito in **tavole della verità**:

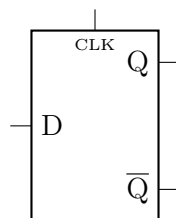
S	R	Q_p	$\overline{Q_p}$	Q	\overline{Q}		S	R	Q
0	0	0	1	0	1		0	0	Q_p
0	0	1	0	1	0		0	1	0
0	1	X	X	0	1	\Rightarrow	1	0	1
1	0	X	X	1	0		1	1	ILL
1	1	X	X	0	0				

3.1.3 D Latch

Come abbiamo visto, l'SR Latch presenta una falla fondamentale al suo interno che genera uno **stato illegale**. Per prevenire ciò, è stata ideata un'evoluzione di questo componente, chiamata **D Latch**.

Un D Latch è provvisto di **due input**:

- **D**: controlla *come* lo stato del circuito deve cambiare
- **CLK**: controlla *quando* lo stato del circuito deve cambiare

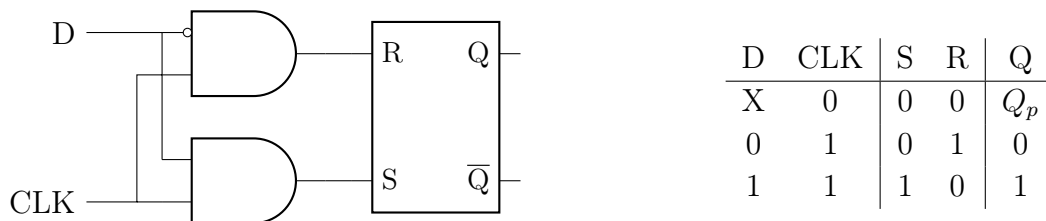


Nel caso in cui **CLK valga 1**, il circuito viene considerato **trasparente**, dunque il valore di D verrà lasciato passare a Q. Nel caso in cui **CLK valga 0**, il circuito viene considerato **opaco**, dunque non lascerà passare D e permetterà a Q di mantenere il suo precedente stato.

Ma come possono questi due input prevenire la presenza di uno stato invalido? La risposta è semplice: **D** e **CLK** in realtà controllano il valore assunto da **S** e **R**:

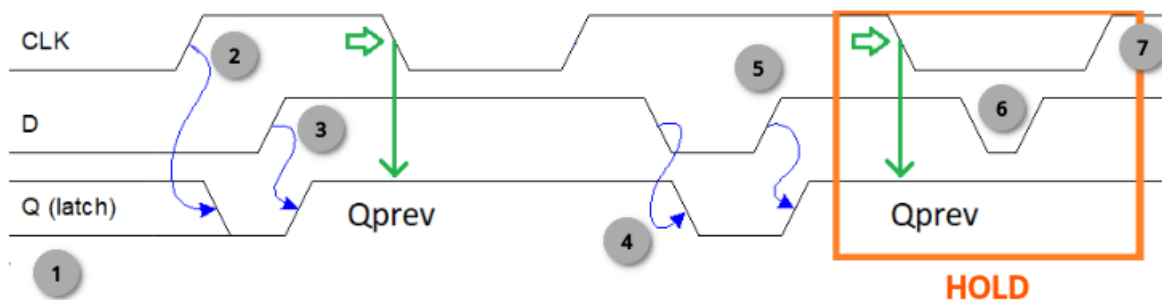
- $S = CLK \cdot D$
- $R = CLK \cdot \bar{D}$

Dunque, un D Latch non è altro che un SR Latch connesso a due porte AND



Esempio di forma d'onda di un D Latch

Proviamo ad analizzare il comportamento di un D Latch tramite la sua rappresentazione del suo stato in **forma d'onda**

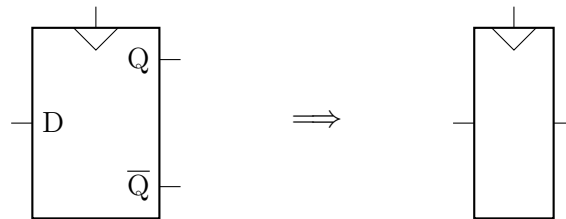


1. Il Latch è in **hold state**. Non sappiamo ancora il valore di Q poiché abbiamo appena acceso l'alimentazione, dunque potrebbe valere **sia 0 che 1**
2. Il Latch diventa trasparente ($CLK = 1$), dunque $Q = D = 0$
3. D **varia** da 0 ad 1 e visto che $CLK = 1$ allora $Q = D = 1$
4. D **varia** da 1 ad 0 e visto che $CLK = 1$ allora $Q = D = 0$
5. D **varia** da 0 ad 1 e visto che $CLK = 1$ allora $Q = D = 1$
6. D **varia** da 0 ad 1 e poco dopo da 1 ad 0, ma visto che **CLK=0** in entrambi i casi allora $Q = Q_p$ (**hold state**)
7. D **varia** da 0 ad 1 e visto $CLK=1$ allora $Q = D = 1$, tuttavia il valore di Q in realtà non cambia poiché $Q_p = 1$

3.1.4 D Flip-Flop

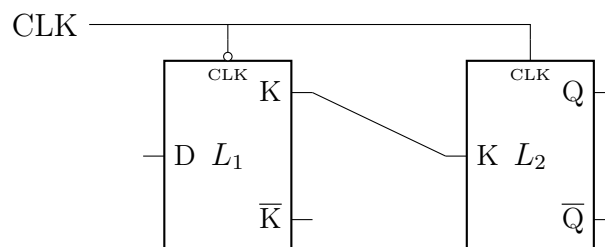
L'elemento di stato più utilizzato è il **D Flip-Flop**, ossia un'ulteriore evoluzione del D Latch. A differenza del suo antenato, lo stato di un D Flip-Flop cambia **solo sul fronte di salita (rising edge)**, ossia nell'esatto momento in cui CLK varia da 0 ad 1.

Per questo motivo, un D flip-flop viene definito **edge-triggered** e, per distinguerlo graficamente da un D Latch, il segnale CLK viene rappresentato da un *triangolino*. Inoltre, poiché uno dei componenti più utilizzati, spesso viene rappresentato in forma *ridotta* (destra).



Un D flip-flop viene implementato tramite **due D Latch interconnessi** tra loro e controllati dallo stesso identico segnale **CLK**, dove tuttavia l'ingresso CLK del primo Latch (L_1) viene negato, mentre il secondo (L_2) no. In questo modo è possibile ottenere la seguente logica:

- Quando **CLK=0**:
 - L_1 è **trasparente**
 - L_2 è **opaco**
 - D **passa** per L_1 , dunque $K = D$
- Quando **CLK=1**:
 - L_1 è **opaco**
 - L_2 è **trasparente**
 - K **passa** per L_2 , dunque $Q = K$

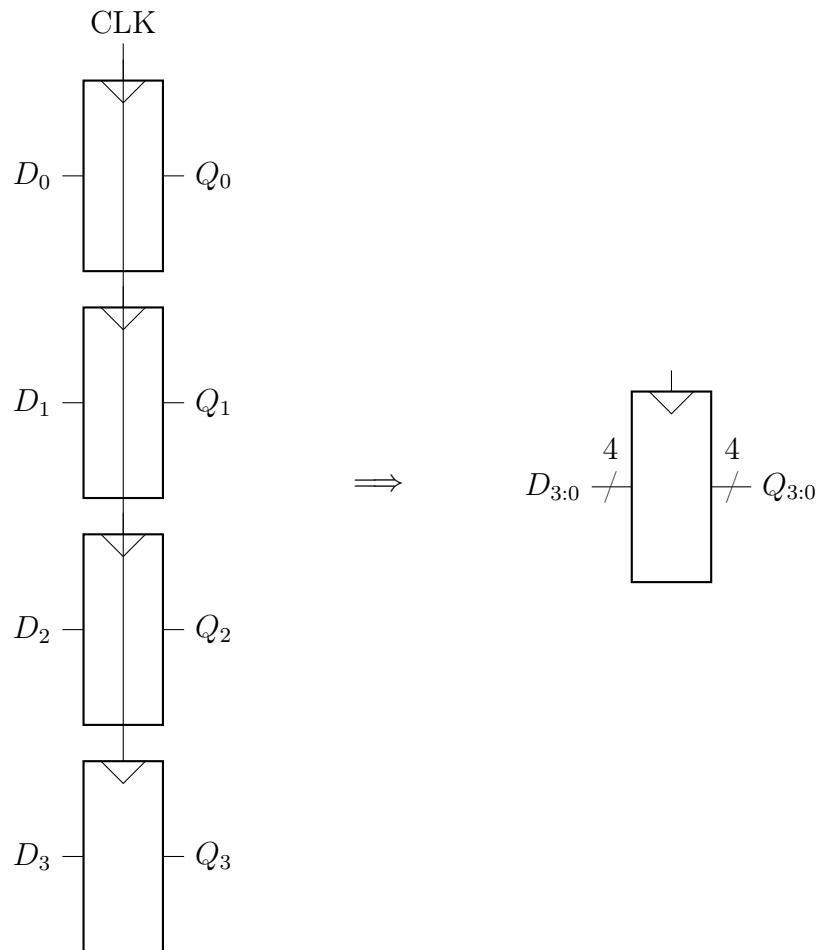


Nel caso in cui ad essere negato fosse L_2 al posto di L_1 ci troveremmo d'innanzi ad un **D flip-flop falling edge**, dove lo stato viene cambiato solo nell'esatto momento in cui CLK varia da 1 ad 0.

Banchi di Flip-Flop come Registri

Come abbiamo accennato nella sezione 3, spesso i flip-flop vengono utilizzati in **quantità multiple**, in modo che possano conservare più di soli due stati all'interno di una macchina, poiché un F-F è in grado di memorizzare **solo 1 bit** al suo interno (dunque 1 flip-flop = 2 possibili stati, 2 flip-flop = 4 possibili stati, ...).

Perciò, a livello grafico viene usata la seguente **convenzione**, in modo da semplificare rappresentazione di circuiti richiedenti molti F-F controllati tutti dallo **stesso CLK**:



3.1.5 Varianti di Flip-Flop

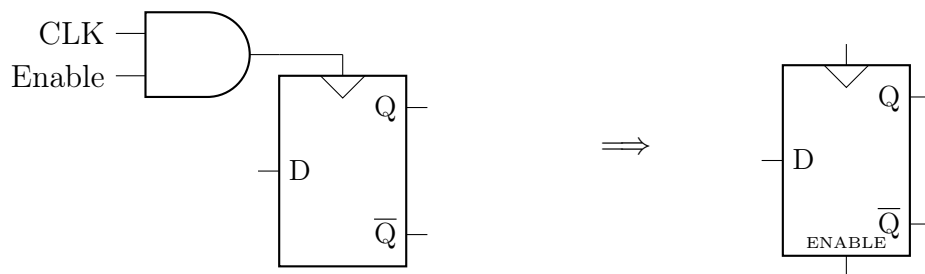
Poiché uno dei componenti più utilizzati in assoluto, è facile immaginare che nel corso degli anni siano state realizzate numerose varianti del classico D Flip-Flop. Di seguito vedremo le più comuni:

- **Enabled Flip-Flop**
- **Settable e Resettable Flip-Flop**
- **SR, JK e T Flip-Flop**

Enabled Flip-Flop

Al contrario di un normale D Flip-Flop, un **segnale di Enable** controlla quando viene concesso a D di poter passare a Q.

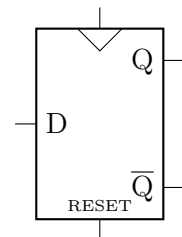
- Se **Enable** = 1 allora D ad Q quando CLK si attiva
- Se **Enable** = 0 allora Q_p ad Q anche se CLK si attiva



Resettable Flip-Flop

Viene aggiunto un segnale di Reset in grado di cambiare autonomamente lo stato di Q.

- Se **Reset** = 1 allora Q viene forzato a 0
- Se **Reset** = 0 allora il flip-flop si comporta normalmente

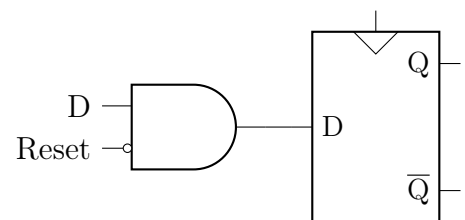


A differenza della precedente variante di F-F, esistono due tipi di Resettable F-F:

- **Sincrono**: il reset avviene solo quando anche CLK si alza
- **Asincrono**: il reset avviene immediatamente ed indipendentemente

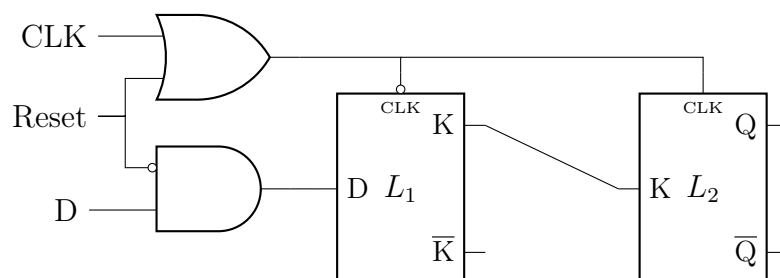
Synchronous Resettable Flip-Flop

Poiché il valore di Q deve essere forzato a 0 (reset) solo nel momento in cui **sia il segnale di reset che il segnale di clock sono pari a 1**, possiamo ottenere questa variante del F-F aggiungendo una semplice porta AND tra D e \overline{Reset}



Asynchronous Resettable Flip-Flop

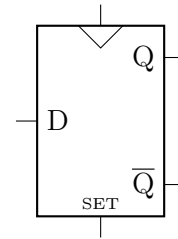
Nella variante asincrona, invece, è necessario modificare il **circuito interno** del F-F:



Settable Flip-Flop

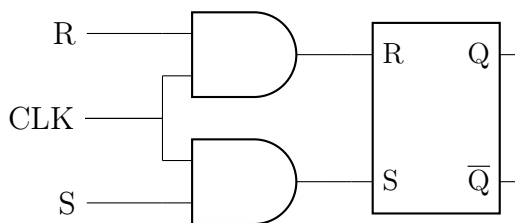
Segue la stessa logica del Resettable, dove all'interno dei circuiti le porte $D \cdot \overline{Reset}$ vengono sostituite con $D + Set$.

- Se **Set** = 1 allora Q viene forzato a 1
- Se **Set** = 0 allora il flip-flop si comporta normalmente



SR Flip-Flop

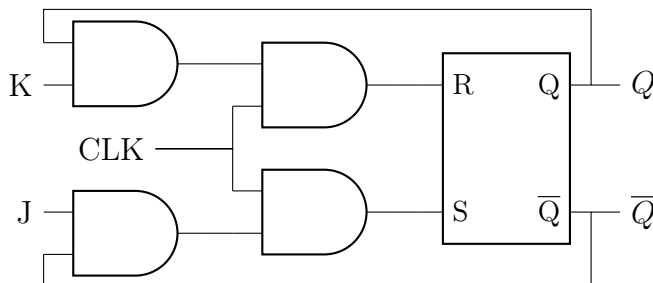
Esattamente uguale ad un SR Latch con l'**aggiunta di un segnale CLK** che controlla il momento della campionatura.



S	R	CLK	Q
X	X	0	Q_p
0	0	1	Q_p
0	1	1	0
1	0	1	1
1	1	1	ILL

JK Flip-Flop

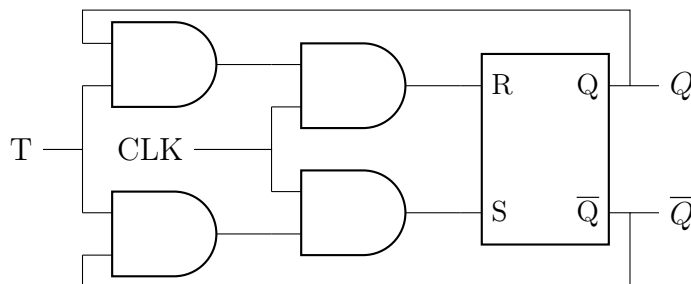
È una versione migliorata dell'SR Flip-Flop (infatti J corrisponde a S mentre K a R), dove **viene risolto** il problema dello **stato invalido**: se J e K valgono entrambi 1, allora il valore di Q diventa il **complementare del suo stato precedente**



J	K	CLK	Q
X	X	0	Q_p
0	0	1	Q_p
0	1	1	0
1	0	1	1
1	1	1	$\overline{Q_p}$

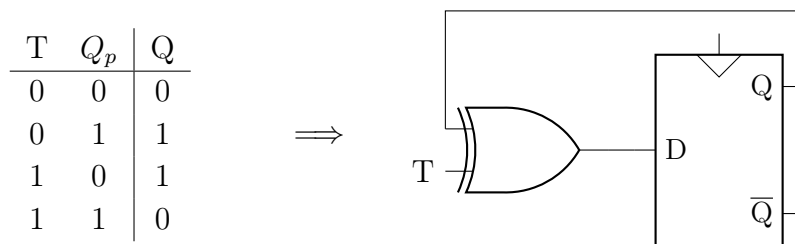
T Flip-Flop

Si tratta di una modifica del JK Flip-Flop, dove gli input J e K sono controllati da un segnale T: se $T = 1$, allora Q diventa il **complementare dello stato precedente**, altrimenti rimane uguale.



T	CLK	Q
X	0	Q_p
0	1	Q_p
1	1	$\overline{Q_p}$

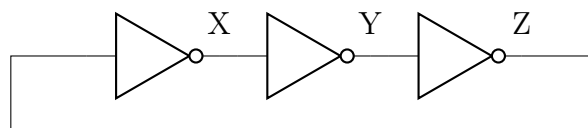
Sviluppando per intero la tabella del T Flip-Flop, notiamo come essa corrisponda ad un semplice **XOR** tra **T** e Q_p dato in input ad un **D Flip-Flop**:



3.1.6 Ring Oscillator

Fino ad ora abbiamo parlato di segnali di clock, dove tuttavia in realtà non vi è un vero "orologio" ad inviare un **segnale periodico**. Tuttavia, utilizzando la logica sequenziale è possibile realizzare un **circuito instabile (o oscillante)** il cui valore **cambia** dopo un determinato periodo di tempo.

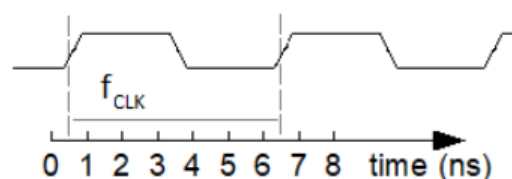
Il modo più semplice per realizzare questa tipologia di circuiti sequenziali è tramite un **Ring Oscillator**, una variante del circuito bistabile:



Ipotizzando che per ognuno dei **tre inverter** valga $t_{cd} = t_{pd} = 1\text{ ns}$, possiamo facilmente notare come il segnale all'interno del circuito abbia il seguente **pattern**:

1. ($t_0 = 0\text{ ns}$) $X = 0, Y = 1, Z = 0$
2. ($t_1 = 3\text{ ns}$) $X = 1, Y = 0, Z = 1$
3. ($t_2 = 6\text{ ns}$) $X = 0, Y = 1, Z = 0$
4. ($t_3 = 9\text{ ns}$) $X = 1, Y = 0, Z = 1$
5. ...

In questo modo abbiamo realizzato un **segnale variabile con periodo 6 ns**, ossia un clock di **frequenza** $f = \frac{1}{6\text{ ns}} = 166.666\text{ MHz}$.



3.2 Circuiti Sequenziali Sincroni

Una volta appresi i modi con cui è possibile memorizzare uno stato, possiamo introdurre questo concetto in sistemi più complessi.

Questi sistemi sono **sincronizzati con un clock**, dunque il loro stato cambia ad ogni suo fronte di salita, sono dotati di **registri**, ossia un numero definito di elementi che ne possano conservare lo stato, e sono collegati a dei **circuiti combinatori** che calcolano gli output del sistema in base allo stato.

In particolare, i **circuiti sequenziali sincroni** seguono il seguente set di regole:

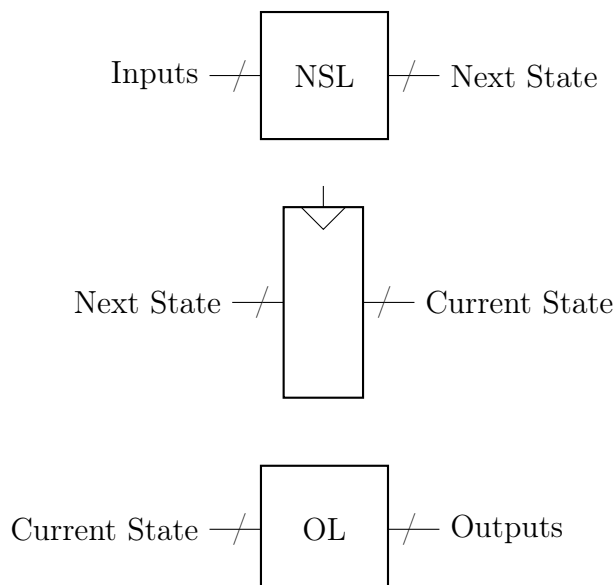
- Ogni elemento del circuito è un **registro** o un **circuito combinatorio**
- **Almeno un elemento** del circuito è un registro
- Tutti i registri sono connessi allo **stesso segnale di clock**
- Ogni **percorso ciclico** contiene almeno un registro

Nelle sezioni successive vedremo i due tipi più comuni di CSS, ossia le **Macchine a Stato Finito (Finite State Machines)** e le **Pipeline**.

3.2.1 Macchine a Stato Finito

Le **Finite State Machine (FSM)** sono costituite da **tre blocchi** fondamentali:

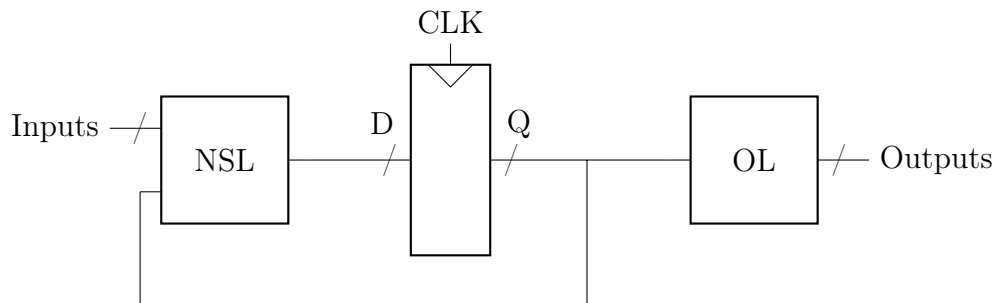
- Un blocco combinatorio chiamato **Next State Logic**, che si occupa di calcolare il successivo stato del sistema in base agli input e allo stato attuale (e all'output precedente in alcuni casi)
- **Uno o più registri** che ne conservino lo stato attuale, corrispondenti ad un banco di flip-flop
- Un ulteriore blocco combinatorio chiamato **Output Logic**, che si occupa di calcolare l'output del sistema in base al suo stato attuale



Macchina di Moore e Macchina di Mealy

Esistono due principali tipologie di FSM:

- **Macchina di Moore:** gli output dipendono solo dallo **stato attuale**



- **Macchina di Mealy:** gli output dipendono dallo **stato attuale** e dagli **input**

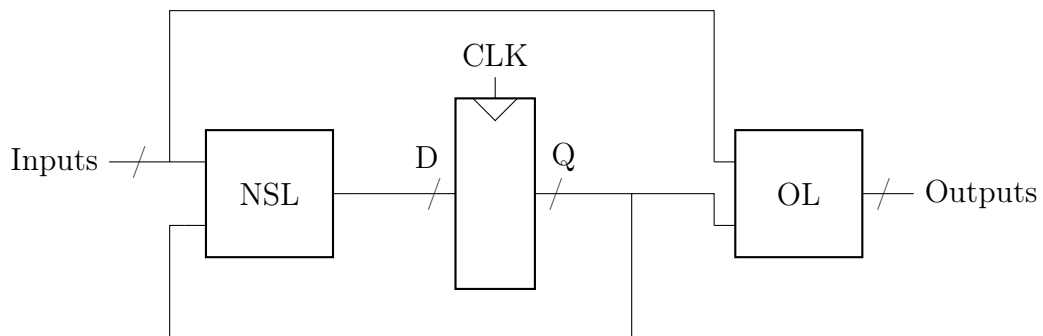
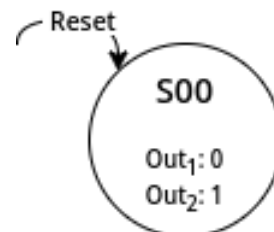


Diagramma delle Transizioni di Stato

Sappiamo dunque che le FSM sono delle macchine che hanno un **numero definito di stati**, in base al numero di registri presenti al loro interno, che **cambiano** a seconda dello stato attuale e/o degli input. Dobbiamo quindi trovare un modo per rappresentare graficamente le **transizioni** da uno stato all'altro:

- **Stati:** vengono rappresentati come cerchi con un'etichetta centrale
- **Transizioni:** vengono rappresentati come archi (o frecce) che collegano uno stato ad un altro, ai quali vengono aggiunti tutti gli input legati alla Next State Logic
- **Output degli Stati:** nella FSM di Moore vengono rappresentati come un'ulteriore etichetta interna ai cerchi, mentre nella FSM di Mealy vengono aggiunti agli archi, poiché necessari per calcolare il Next State assieme agli input



Codifica degli Stati

Poiché gli stati vengono memorizzati da un banco di flip-flop (dunque come un insieme di 0 ed 1), è necessario definire una **codifica** che possa associare dei **bit** ad ogni **stato**. Immaginiamo di voler rappresentare quattro stati: $S0$, $S1$, $S2$ e $S3$.

- **Codifica binaria:** possiamo utilizzare 2 bit (dunque 2 flip-flop) per rappresentare tutti i quattro stati

Stato	Registri	
$S0$	0	0
$S1$	0	1
$S2$	1	0
$S3$	1	1

- **Codifica One-hot:** utilizziamo un registro per ogni stato, dunque solo un registro può essere attivo alla volta. In questo esempio, abbiamo quindi bisogno di 4 F-F.

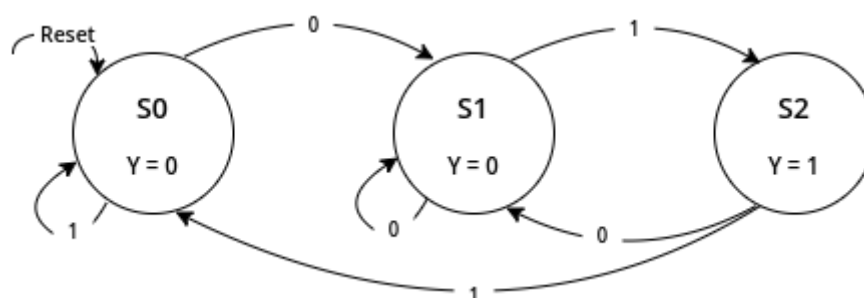
Stato	Registri			
$S0$	0	0	0	1
$S1$	0	0	1	0
$S2$	0	1	0	0
$S3$	1	0	0	0

3.2.2 Esempio di FSM di Moore e FSM di Mealy

Alyssa P. Hacker ha una lumaca che striscia su un nastro di carta su cui sono disegnati degli 0 e degli 1. La lumaca sorride solo quando le ultime due cifre su cui ha strisciato sono uno 0 seguito da un 1. Vogliamo progettare una FSM di Moore ed una FSM di Mealy corrispondenti al cervello della lumaca.

Macchina di Moore - Stati di Transizione

- **$S0$:** la lumaca ha letto un 1 preceduto da un 1. Se l'input sarà 0 allora lo stato attuale diventerà $S1$, altrimenti rimarrà $S0$
- **$S1$:** la lumaca ha letto uno 0 preceduto da un 1. Se l'input sarà 1 allora lo stato attuale diventerà $S2$, altrimenti rimarrà $S1$
- **$S2$:** la lumaca ha letto un 1 preceduto da uno 0. Se l'input sarà 1 allora lo stato attuale diventerà $S0$, altrimenti sarà $S1$



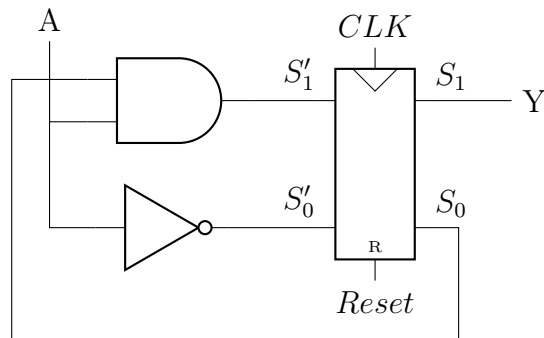
Macchina di Moore - Tabelle di Stato e di Output

State	Encoding		Curr. State	Output
S	S_1	S_0		
$S0$	0	0	$S0$	0
$S1$	0	1	$S1$	0
$S2$	1	0	$S2$	1

Curr. State		Inputs	Next State			S	A	S'	Y
S_1	S_0	A	S'_1	S'_0					
0	0	0	0	1	\Rightarrow	$S0$	0	$S1$	0
0	0	1	0	0		$S0$	1	$S0$	0
0	1	0	0	1		$S1$	0	$S1$	0
0	1	1	1	0		$S1$	1	$S2$	0
1	0	0	0	1		$S2$	0	$S1$	1
1	0	1	0	0		$S2$	1	$S0$	1

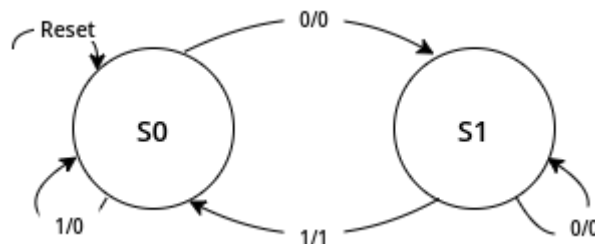
Macchina di Moore - Schematica del circuito

- $S'_0 = \bar{A}$
- $S'_1 = S_0 \cdot A$
- $Y = S_1$



Macchina di Mealy - Stati di Transizione

- **S0**: l'output precedente è uno 0. Se l'input sarà 0 allora lo stato attuale diventerà $S1$, altrimenti rimarrà $S0$
- **S0**: Se l'output precedente è un 0 e l'input sarà 0, allora lo stato attuale rimarrà $S1$, mentre se l'output precedente è un 1 e l'input sarà 1, allora lo stato attuale diventerà $S0$,



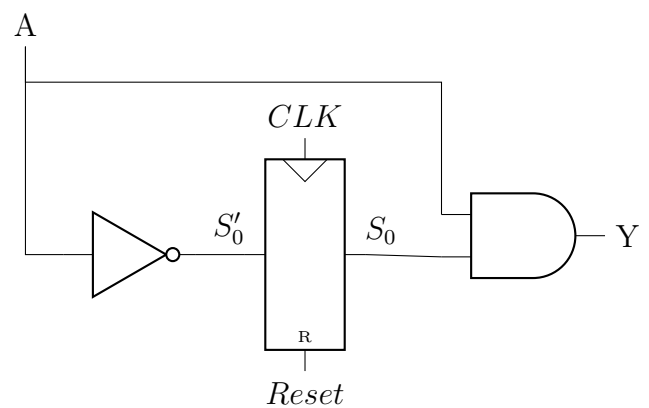
Macchina di Mealy - Tabelle di Stato e di Output

State	Encoding	Curr. State	Inputs	Next State	Outputs
S	S_0	S_0	A	S'_0	Y
S_0	0	0	0	1	0
S_1	1	0	1	0	0
		1	0	1	0
		1	1	0	1

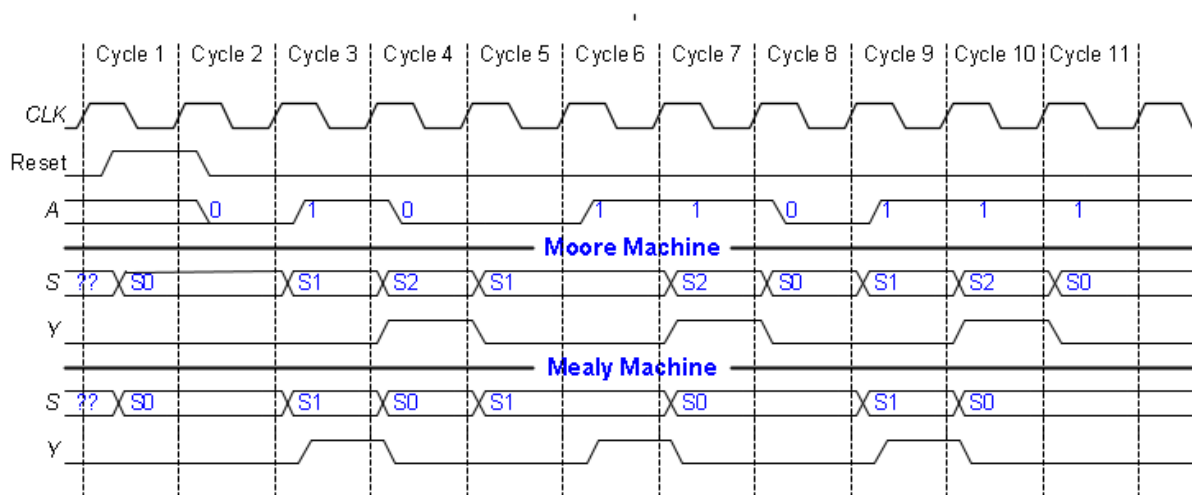
Macchina di Mealy - Schematica del circuito

- $S'_0 = \overline{A}$
- $Y = S_0 \cdot A$

Notiamo che questa FSM di Mealy cicla una volta in meno attraverso i registri, dunque l'output viene definito **un colpo di clock prima** rispetto alla FSM di Moore che abbiamo realizzato prima

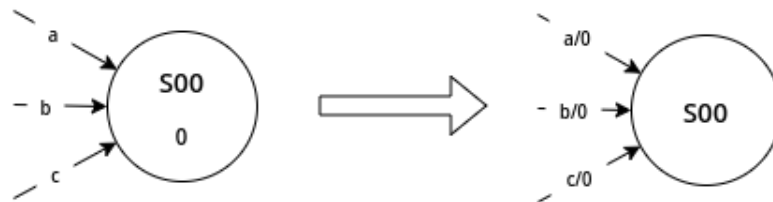


Confronto tra i diagrammi temporali

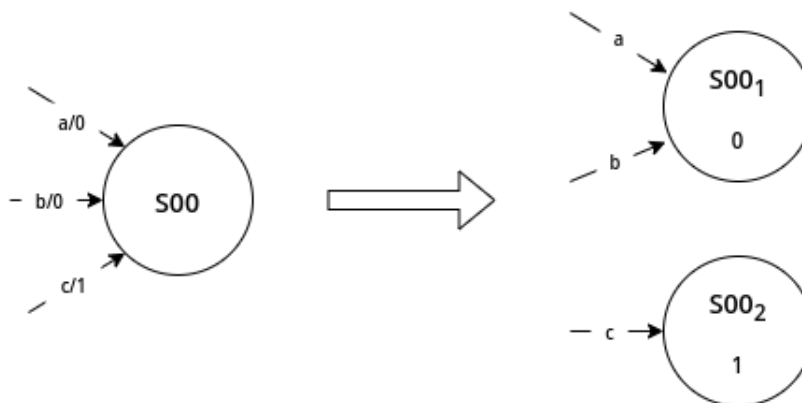


3.2.3 Da FSM di Moore a FSM di Mealy e viceversa

Una proprietà particolare di queste due tipologie di FSM è la totale intercambiabilità tra di loro. Infatti, ogni **FSM di Moore** può essere **trasformata** in una **FSM di Mealy associando l'uscita** appartenente a uno stato **a tutte le transizioni** che partono da tale stato. Nel caso in cui vadano a crearsi degli stati completamente uguali, è possibile semplificarli tra di loro mantenendone solo uno.



Allo stesso modo, una **FSM di Mealy** può essere **trasformata** in una **FSM di Moore** replicando lo stato della macchina di Moore compiendo il **processo inverso**, dunque associando l'output delle transizioni agli stati generati. Se le transizioni che portano a tale stato hanno delle **uscite differenti**, allora lo stato verrà **replicato**, in modo da avere **una sola** combinazione di transizioni/output per ogni stato.



Inoltre, nel caso in cui sia necessario **replicare** uno stato, le sue **uscite** originali verranno attribuite ad **entrambi gli stati replicati**.



Inizializzazione delle FSM e Stati indefiniti

Poiché lo stato iniziale di un flip-flop appena acceso è indeterminato, dunque potrebbe contenere sia uno 0 che un 1, anche lo **stato della FSM è indeterminato**.

Per evitare questo problema si dotano sempre le FSM di un **segnale di Reset** che forza la macchina in uno **stato noto** (tipicamente tutti 0) da cui può partire l'elaborazione.

Inoltre, una FSM può avere degli **stati irraggiungibili**, ossia degli stati che non possono essere raggiunti con nessuna sequenza di valori di ingresso. L'**irraggiungibilità dipende dallo stato iniziale** della FSM (quindi lo stato deciso dal Reset). Spesso questi stati sono presenti nelle FSM che non hanno un numero di stati esattamente pari ad una potenza di 2.

Infatti, gli stati non utilizzati contribuiscono comunque a generare i output della NSL, ma non sono mai raggiunti, ossia l'output della NSL **non corrisponde mai** alla codifica di questi stati.

Procedura per un design corretto delle FSM

1. Identificare gli input e gli output
2. Disegnare il diagramma delle transizioni
3. Estrapolare la tabella delle transizioni
4. Scegliere un encoding per gli stati
5. Scrivere la tabella degli output
6. Estrapolare le equazioni booleane della NSL e della OL
7. Disegnare il circuito

Ricavare il funzionamento di una FSM dal suo circuito

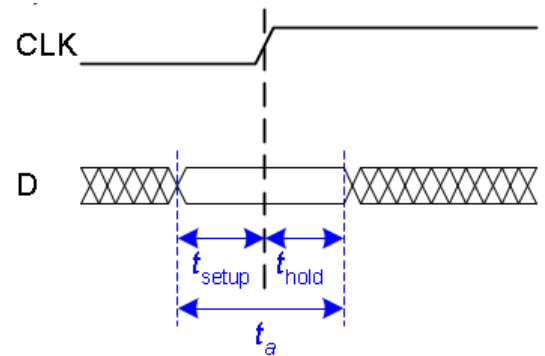
1. Esaminare il circuito, individuandone input, output e bit di stato
2. Scrivere le equazioni booleane della NSL e della OL
3. Estrapolare la tabella delle transizioni e degli output, scartando eventuali stati irraggiungibili
4. Scegliere un encoding per gli stati
5. Disegnare il diagramma delle transizioni
6. Descrivere a parole il funzionamento della FSM

3.2.4 Temporizzazioni sequenziali

Come abbiamo visto, i circuiti sequenziali sincroni utilizzano dei D Flip-Flop, i quali **campionano** il valore in ingresso D quando il clock è in **rising-edge**. Per questo motivo, viene logico pensare che i valori in ingresso **debbano rimanere stabili** nel momento in cui il clock si alza, altrimenti la loro campionatura potrebbe non andare a buon fine (metastabilità).

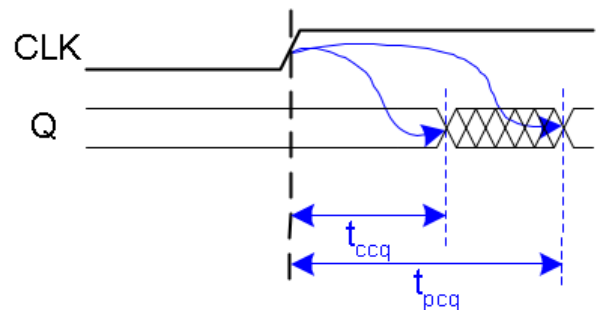
Dobbiamo quindi analizzare alcuni **periodi temporali** necessari affinché gli **input** possano rimanere **stabili**, permettendo una campionatura corretta:

- **Tempo di setup (t_{setup})**: periodo di tempo *precedente* alla salita del clock in cui D deve rimanere stabile
- **Tempo di hold (t_{hold})**: periodo di tempo *successivo* alla salita del clock in cui D deve rimanere stabile
- **Tempo di apertura (t_a)**: periodo di tempo *totale* in cui D deve rimanere stabile ($t_a = t_{\text{setup}} + t_{\text{hold}}$)



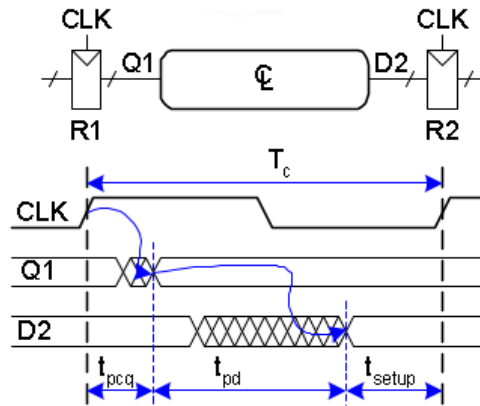
Analogamente, è necessario assicurarsi che **anche gli output siano stabili**, poiché non possiamo essere completamente sicuri che il valore Q venga generato immediatamente al momento della campionatura:

- **Delay di contaminazione (t_{pcq})**: periodo di tempo *successivo* alla salita del clock in cui il valore di Q potrebbe essere instabile
- **Delay di propagazione (t_{ccq})**: periodo di tempo *successivo* alla salita del clock dopo cui il valore di Q è sicuramente stabile



Disciplina dinamica

Immaginando un circuito composto da due registri R_1 e R_2 connessi tra di loro da un circuito combinatorio, è facile trovare una problematica intrinseca ai registri stessi: essi vengono attivati dallo **stesso segnale di clock** allo stesso istante. Tuttavia, affinché il **registro R_2** possa campionare in modo stabile il valore in entrata, è necessario **calcolare** accuratamente il suo **tempo di setup**.

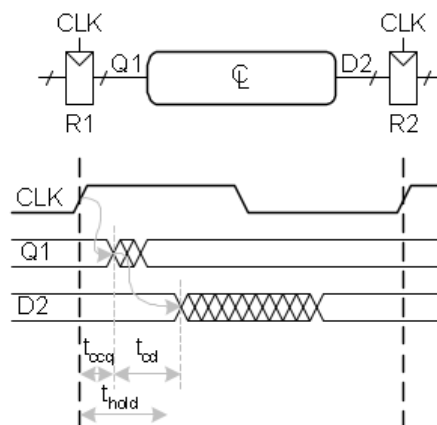


Da questo diagramma temporale possiamo notare come **vincolo temporale (T_c)** dipenda dal **delay massimo** affinché il valore in uscita da R_1 possa essere campionato stabilmente da R_2 , passando attraverso il circuito combinatorio. L'**input** di R_2 dovrà dunque essere stabile **prima del clock edge** per un periodo t_{setup} (**Vincolo temporale di setup**).

Il vincolo temporale T_c sarà quindi **maggiore o uguale** al delay massimo, corrispondente alla somma tra il delay di propagazione di R_1 (t_{pcq}), il delay di propagazione del circuito combinatorio (t_{pd}) e il tempo di setup di R_2 (t_{setup}).

$$T_c \geq t_{pcq} + t_{pd} + t_{setup}$$

Tuttavia, è necessario considerare anche il **delay minimo** per cui il valore in uscita da R_1 debba rimanere stabile. Di conseguenza, ricaviamo che anche l'**input** di R_2 debba rimanere stabile **dopo il clock edge** per almeno t_{hold} affinché esso possa essere campionato correttamente (**Vincolo temporale di hold**).

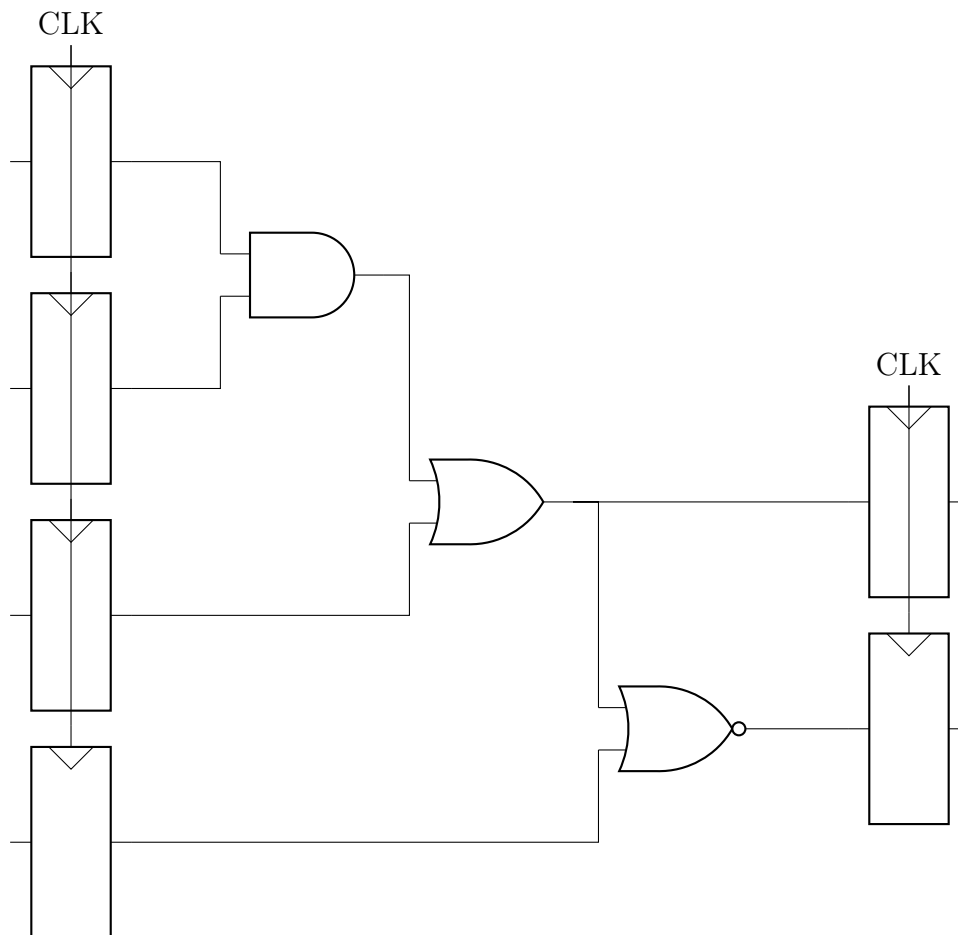


$$t_{hold} < t_{ccq} + t_{cd}$$

Esempio di analisi temporale

Consideriamo i seguenti valori e il seguente circuito, verificando che i vincoli temporali di setup e di hold siano rispettati:

- $t_{ccq} = 30ps$
- $t_{pcq} = 50ps$
- $t_{pd} = 35ps$ (per tutte le porte)
- $t_{cd} = 25ps$ (per tutte le porte)
- $t_{setup} = 60ps$
- $t_{hold} = 70ps$



Vincolo temporale di setup

$$t_{pd} = 3 \cdot 35ps = 105ps$$

$$T_c \geq (50 + 105 + 60)ps = 215ps$$

$$f_c = 1/T_c = 4.65GHz$$

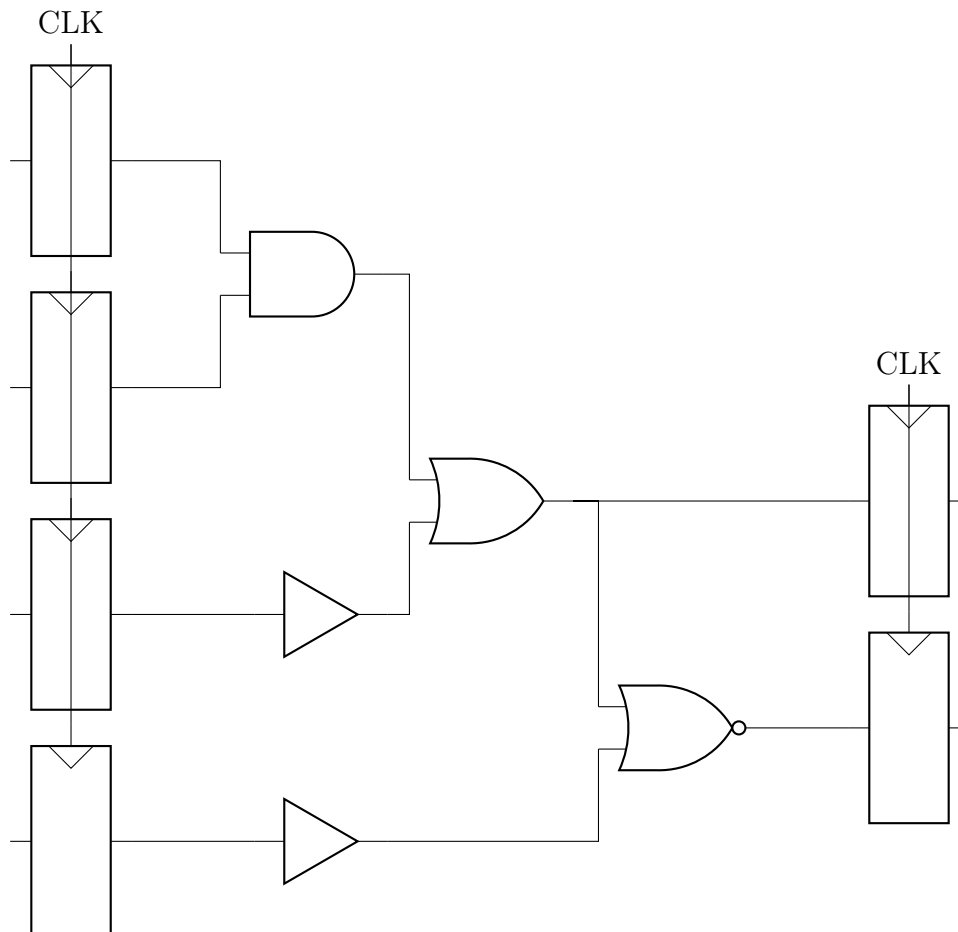
Vincolo temporale di hold

$$t_{cd} = 25ps$$

$$t_{hold} < t_{ccq} + t_{cd} \Rightarrow 70 < (30 + 25)ps$$

Il vincolo non viene rispettato

Notiamo che il vincolo temporale di hold viene violato, rendendo instabile la campionatura del valore in entrata dei registri secondari. Per risolvere il problema, è necessario **aggiungere dei buffer** agli **short path**, in modo che il vincolo possa essere nuovamente rispettato senza alterare il vincolo temporale di setup:



Vincolo temporale di setup

$$t_{pd} = 3 \cdot 35ps = 105ps$$

$$T_c \geq (50 + 105 + 60)ps = 215ps$$

$$f_c = 1/T_c = 4.65GHz$$

Vincolo temporale di hold

$$t_{cd} = 2 \cdot 25ps = 50ps$$

$$t_{hold} < t_{ccq} + t_{cd} \Rightarrow 70 < (30 + 50)ps$$

Il vincolo viene rispettato

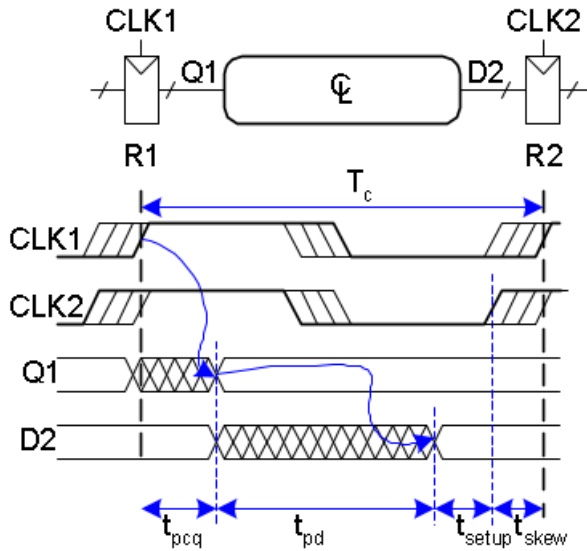
Sfasamento del clock

Fino ad ora, non abbiamo considerato un ultimo fattore in grado di influenzare la campionatura di valori in sequenza, ossia lo **sfasamento del segnale di clock**: poiché si tratta pur sempre di un segnale elettrico, il segnale di clock **non arriva contemporaneamente** a tutti i vari registri. Perciò, nel definire il **tempo di apertura** di un registro in modo che rispetti i vincoli della disciplina dinamica, è necessario considerare anche i **due casi peggiori**:

- Il registro R_2 è in *anticipo* rispetto al registro R_1
- Il registro R_2 è in *ritardo* rispetto al registro R_1

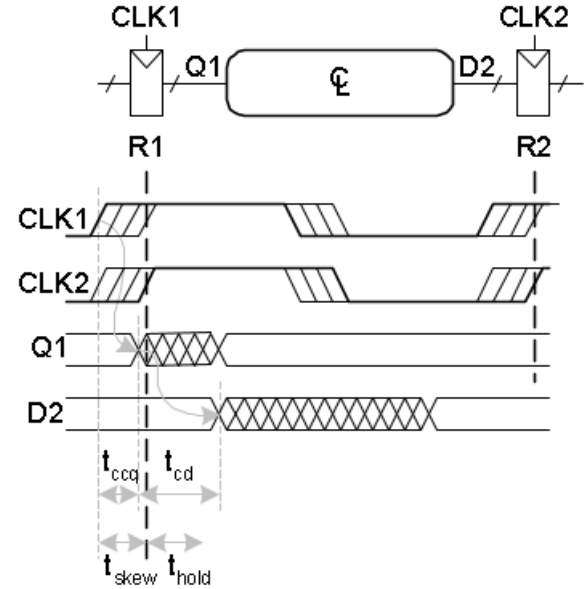
Per ovviare questo problema, quindi, è necessario considerare anche il **tempo di sfasamento del clock (clock skew)** nelle due equazioni dei vincoli temporali di *setup* e di *hold*:

Vincolo temporale di setup



$$T_c \geq t_{pcq} + t_{pd} + t_{setup} + t_{skew}$$

Vincolo temporale di hold



$$t_{hold} + t_{skew} < t_{ccq} + t_{cd}$$

3.2.5 Pipelines e parallelismo

Prima di poter parlare della seconda tipologia di circuiti sequenziali sincroni, ossia le **pipeline**, è necessario enunciare alcune **definizioni**:

- **Token**: un insieme di input processati per generare un insieme di output
- **Latenza**: tempo impiegato da un token per passare dall'inizio alla fine del circuito
- **Portata (Throughput)**: numero di token processati per unità di tempo

Ciò che ci interessa particolarmente, è **aumentare il throughput** di un circuito. Per ottenere ciò, possiamo utilizzare una tecnica chiamata **parallelismo**, che si divide in due tipologie:

- **Parallelismo spaziale**: **duplicare l'hardware** del circuito in modo effettuare lo stesso compito multiple volte contemporaneamente
- **Parallelismo temporale**: il compito viene **diviso in più fasi** svolte simultaneamente (anche chiamato **pipelining**), proprio come avverrebbe in una *catena di montaggio*

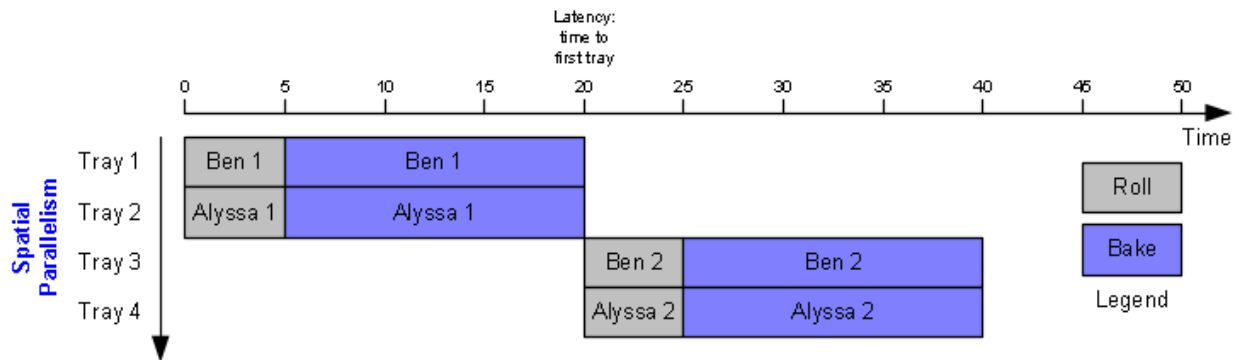
Funzionamento del parallelismo

Ben Bitdiddle sta cucinando dei biscotti per i suoi amici, impiegando 5 minuti per impastare una teglia di biscotti e 15 minuti per cuocerli nel forno. Qual è la *latenza* con cui Ben riesce a sfornare una teglia di biscotti? *Quante teglie* riesce a sfornare in un'ora?

$$\text{Latenza} = 5\text{min} + 15\text{min} = 20\text{min} = \frac{1}{3} \text{ ore}$$

$$\text{Throughput} = \frac{1 \text{ teglia}}{\frac{1}{3} \text{ ore}} = 3 \text{ teglie/ora}$$

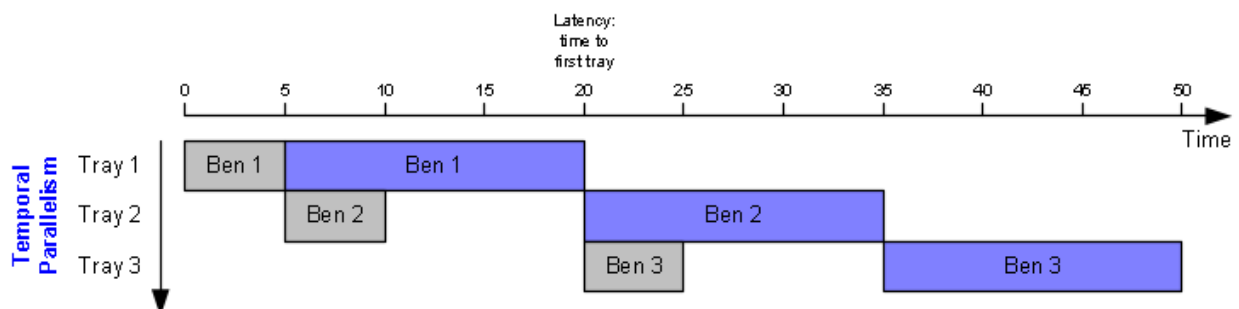
Poiché ha molti amici, Ben vorrebbe aumentare la quantità di biscotti sfornati in un'ora in modo da poter raggiungere prima la quantità necessaria di biscotti. Ben decide di chiedere ad Alyssa di aiutarlo ad impastare, facendosi prestare anche il suo forno (**parallelismo spaziale**).



$$\text{Latenza} = 5\text{min} + 15\text{min} = 20\text{min} = \frac{1}{3} \text{ ore}$$

$$\text{Throughput} = 2 \cdot \frac{1 \text{ teglia}}{\frac{1}{3} \text{ ore}} = 6 \text{ teglie/ora}$$

Con l'aiuto di Alyssa, Ben è riuscito a raddoppiare la quantità di teglie sfornate in un'ora. Tuttavia, Ben pensa anche ad un'altra opzione, ossia impastare i biscotti dell'infornata successiva mentre aspetta che l'infornata attuale sia cotta (**parallelismo temporale**).



ATTENZIONE: è opportuno ricordare che con **latenza** si intende il tempo impiegato da un token per passare dall'inizio alla fine del circuito. Escludendo la prima, possiamo notare dal grafico come ogni infornata sia costituita da 5 minuti di tempo di impasto, 10 minuti di tempo di attesa affinché il forno sia libero e 15 minuti di tempo di cottura, per un totale di 30 minuti.

Analogamente, ricordiamo che per **throughput** si intende il numero di token processati per unità di tempo. Dunque, escludendo ancora la prima infornata, possiamo notare come dal forno esca una teglia ogni 15 minuti, dunque 4 teglie all'ora.

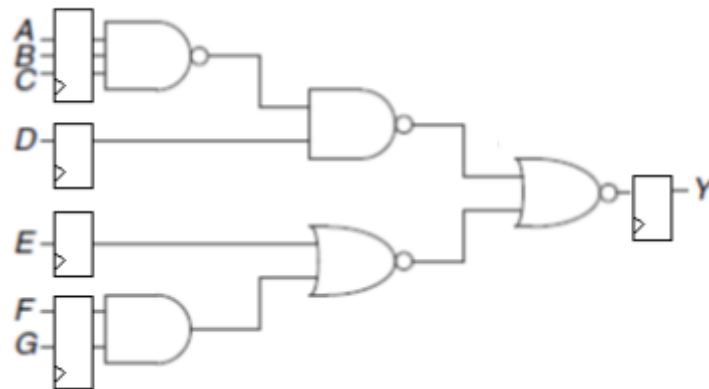
$$\text{Latenza} = 5min + 10min + 15min = 30min = \frac{1}{2} \text{ ore}$$

$$\text{Throughput} = \frac{1 \text{ teglia}}{\frac{1}{4} \text{ ore}} = 4 \text{ teglie/ora}$$

Combinando **entrambe le tecniche** di parallelismo, Ben e Alyssa riuscirebbero a sfornare 8 teglie all'ora con una latenza di 30 minuti.

Parallelismo nei circuiti

Una volta capita la differenza tra i due parallelismi, vediamo **un esempio più concreto** in cui è possibile velocizzare un circuito applicando il parallelismo temporale:



Sapendo che $t_{pcq} = 15ps$, $t_{setup} = 65ps$ e $t_{pd} = 40ps$ per ogni porta, calcoliamo:

$$\text{Latenza} = 1 \text{ colpo di clock}$$

$$\text{Periodo} = 15ps + 3 \cdot 40ps + 65ps = 200ps$$

$$\text{Frequenza} = \frac{1}{200ps} = 5000000000Hz = 5GHz$$

$$\text{Throughput} = \frac{1 \text{ operazione}}{5GHz} = 5Goperazioni/sec$$

Capitolo 4

Blocchi costruttivi digitali

Per **blocchi costruttivi digitali** si intende l'insieme di **elementi digitali più semplici**, situati alla base della gerarchia dei componenti digitali: porte logiche, mux, decoder, registri, circuiti aritmetici, contatori, vettori di memoria e vettori logici.

Questi elementi hanno delle interfacce e funzioni ben definite (**modularità**) e la loro struttura regolare (**regolarità**) permette di estenderli facilmente in formati più grandi (ad esempio, una porta 8-AND è molto semplice da realizzare utilizzando 8 porte 1-AND). Tramite questi blocchi digitali vengono realizzati dispositivi complessi, come un microprocessore.

4.1 Blocchi aritmetici

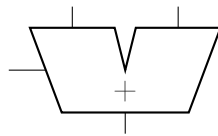
La prima tipologia di blocchi costruttivi digitali che vedremo è l'insieme di elementi che si occupano di compiere le principali **operazioni aritmetiche** della matematica:

- Sommatore
- Sottrattori
- Comparatori
- Shifter
- Moltiplicatori

4.1.1 Sommatore

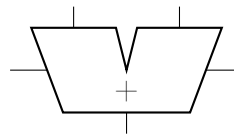
Vediamo le due tipologie più semplici di sommatore ad 1 bit: **half-adder** e **full-adder**. Entrambi si occupano di **sommare due bit** ricevuti in ingresso, restituendo in output il risultato della somma e il riporto generato da essa (carry). Tuttavia il secondo riceve un **ulteriore** ingresso da sommare agli altri due, ossia l'ingresso del riporto.

Ovviamente, è facile immaginare come per effettuare somme tra sequenze di bit siano necessari più sommatore ad 1-bit (successivamente vedremo come).

Half-Adder

$$S = A \oplus B$$

$$C_{out} = AB$$

Full-Adder

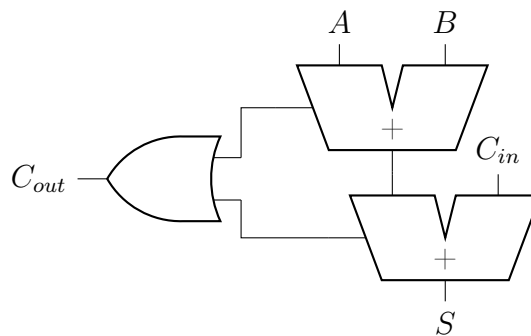
$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$

A	B	S	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

C_{in}	A	B	S	C_{out}
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

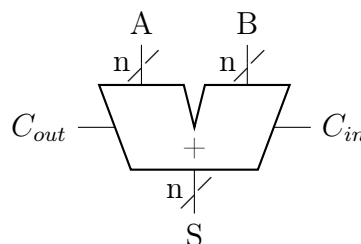
A riprova dell'**elevata regolarità** di tali componenti, possiamo puntualizzare come un full-adder possa essere realizzato concatenando due half-adder:



Sommatori a più bit

I sommatore a più bit, anche chiamati *sommatori a propagazione del riporto* (**Carry Propagate Adders** o **CPA**), si dividono in più tipologie. Tuttavia, noi ne vedremo solo due:

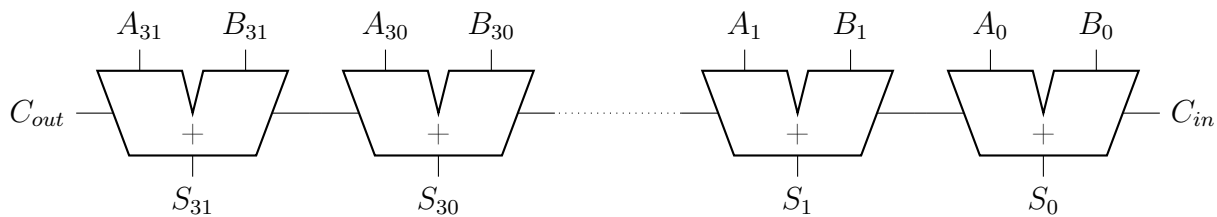
- **Ripple-carry Adders**, una versione più lenta
- **Carry-lookahead Adders**, una versione più veloce



Ripple-carry Adder

Vengono concatenati **N sommatore ad 1-bit**. Di conseguenza, ogni riporto in uscita di un sommatore M sarà il carry in ingresso del sommatore $M+1$ (difatti, il termine **ripple**, in questo caso traducibile come "propagazione", evidenzia proprio questo aspetto).

Tuttavia, ciò necessita di **un ritardo notevole**, poiché è richiesto che il **riporto** venga *trasportato* per tutta la catena di sommatore, dunque ogni sommatore dovrà attendere che il suo predecessore abbia effettuato la somma. Di seguito vediamo un esempio di **RCA a 32-bit**:



$$t_{ripple} = N \cdot t_{FA} - \text{dove } t_{FA} \text{ è il delay di un full-adder}$$

Carry-lookahead Adder

Poiché utilizzare un RCA per somme tra sequenze di un elevato numero bit necessiterebbe di un ritardo enorme per ottenere il riporto finale, viene implementato un metodo che permette di **spezzare in blocchi di K sommatore** l'intera catena di sommatore, **calcolando preventivamente il riporto di ogni blocco (lookahead, ossia "previsione")**: se calcolo preventivamente il riporto del blocco M , il blocco $M+1$ potrà calcolare le sue somme in contemporanea al blocco M .

Definizioni:

- A livello logico, un sommatore i in una catena di N sommatore può **generare** un riporto di uscita nella sua somma oppure **propagare** il riporto in ingresso verso quello di uscita
- Possiamo quindi calcolare preventivamente i **segnali generatori** e i **segnali propagatori** di ogni sommatore, per poi andare a calcolare il suo **riporto in uscita**:

- **Generatore**: un sommatore i può generare un riporto solo se A_i e B_i valgono entrambi 1

$$G_i = A_i B_i$$

- **Propagatore**: un sommatore i può propagare un riporto solo se A_i o B_i valgono 1

$$P_i = A_i + B_i$$

- **Carry out**: il riporto di un sommatore i è:

$$C_i = G_i + P_i C_{i-1} = A_i B_i + (A_i + B_i) C_{i-1}$$

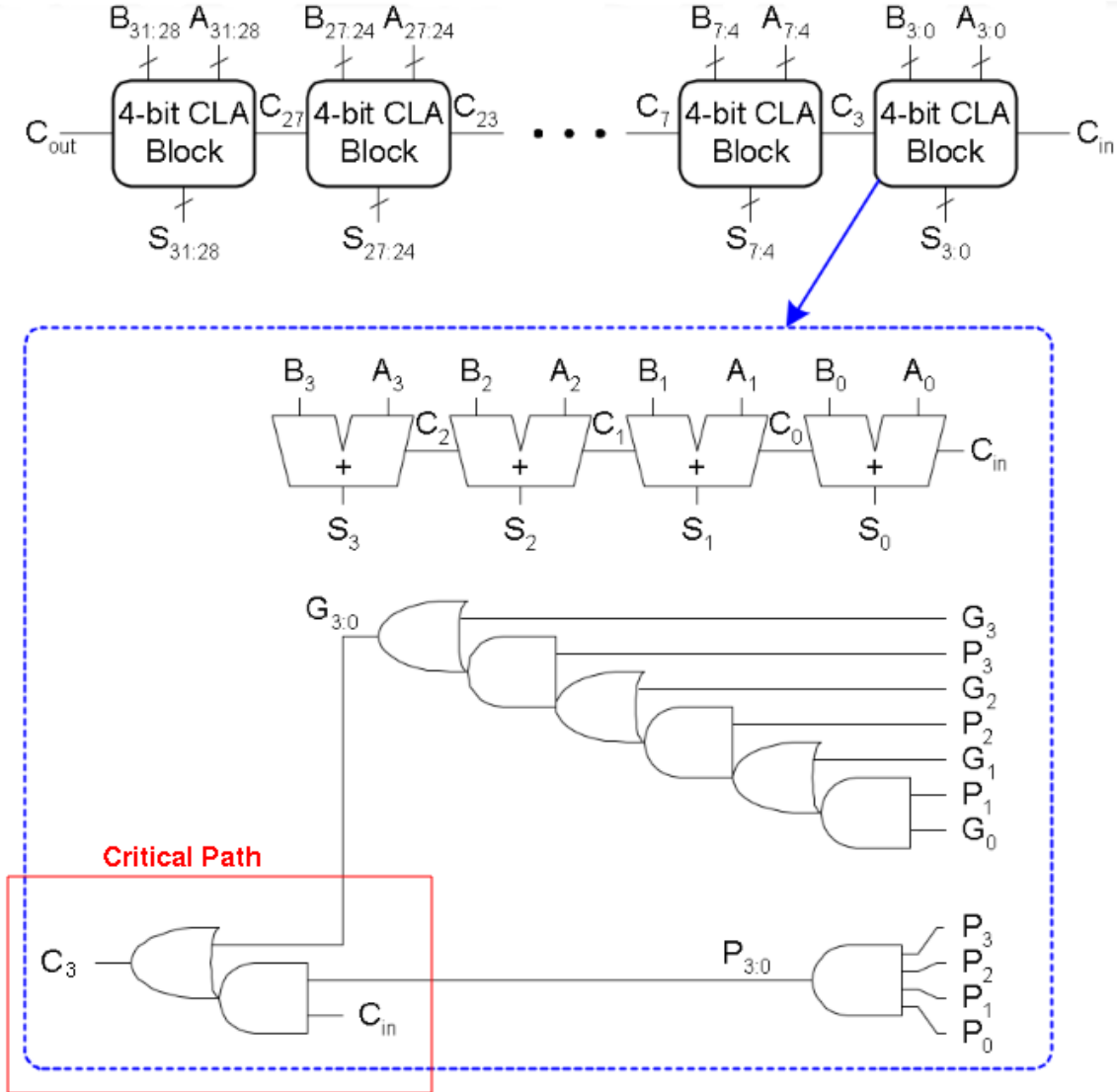
Una volta calcolati i generatori e i propagatori di ogni sommatore i in un blocco k , possiamo ora calcolare se il **blocco stesso** sarà un generatore e/o un propagatore.

Immaginiamo di aver suddiviso la catena RCA di 32-bit mostrata precedentemente in **blocchi da 4 bit**. Scriviamo le equazioni booleane che descrivono il comportamento del blocco 3:0 (ossia dal sommatore 0 al sommatore 3).

$$\begin{aligned} P_{3:0} &= P_3 \cdot P_2 \cdot P_1 \cdot P_0 \\ G_{3:0} &= G_3 + P_3(G_2 + P_2(G_1 + P_1 \cdot G_0)) \\ C_3 &= G_{3:0} + P_{3:0} \cdot C_{in} \end{aligned}$$

Poiché ogni blocco della catena segue lo stesso comportamento, definiamo le seguenti **equazioni generali**:

$$\begin{aligned} P_{i:j} &= P_i \cdot P_{i-1} \cdot P_{i-2} \cdot P_j \\ G_{i:j} &= G_i + P_i(G_{i-1} + P_{i-1}(G_{i-2} + P_{i-2} \cdot G_j)) \\ C_i &= G_{i:j} + P_{i:j} \cdot C_{j-1} \end{aligned}$$



Il **percorso critico** del sistema viene quindi **ridotto** a t_{AND-OR}

Possiamo quindi riassumere l'addizione tramite blocchi CSA in:

1. Calcolare G_i e P_i per **ogni sommatore** del blocco
2. Calcolare $G_{i:j}$ e $P_{i:j}$ per **ogni blocco** della catena
3. Mentre vengono calcolate le somme dei **blocchi intermedi**, C_{in} si **propaga** in base alle logiche di generazione/propagazione dei blocchi stessi
4. Viene calcolata la somma dell'**ultimo blocco**

In questo modo, il **ritardo della catena** viene notevolmente diminuito: per una catena di N bit suddivisa in blocchi di K bit avremo che:

$$t_{CLA} = t_{pg} + t_{pg-block} + \left(\frac{N}{K} - 1 \right) \cdot t_{AND-OR} + k \cdot t_{FA}$$

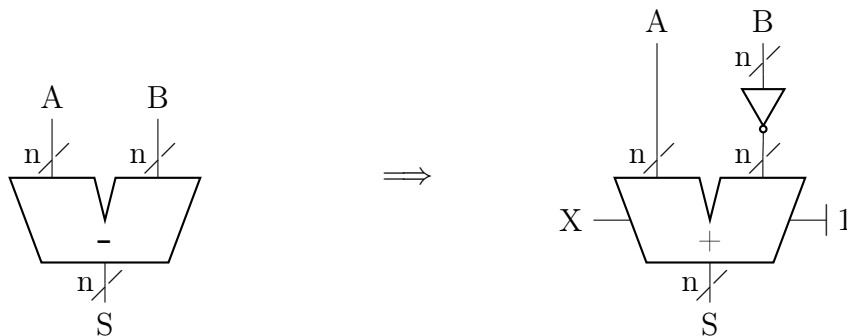
dove:

- t_{pg} è il delay necessario per calcolare tutti i G_i e P_i di ogni sommatore
- $t_{pg-block}$ è il delay necessario per calcolare tutti i $G_{i:j}$ e $P_{i:j}$ di ogni blocco
- t_{AND-OR} è il delay impiegato da ogni C_{j-1} per passare nelle ultime due porte AND ed OR fino ad arrivare a C_i (il percorso critico)

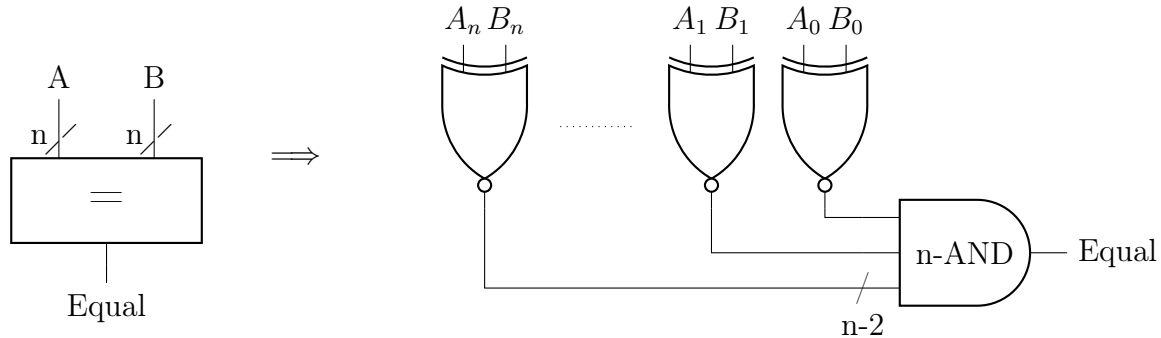
Generalmente, un sommatore CLA risulta **più veloce** di un sommatore RCA quando il numero di bit è **maggiore di 16**.

4.1.2 Sottrattori e Comparatori

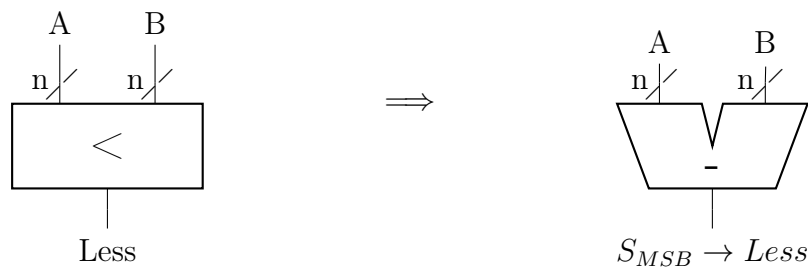
Ricordando il funzionamento dei numeri in **Complemento a 2** (vedi 1.2.2), è facile immaginare come possa essere realizzato un sottrattore partendo da un sommatore: basta **negare** i bit della sequenza B (ed ottenere il $Ca1$) per poi **aggiungere 1** tramite il C_{in} (ed ottenere il $Ca2$). Una volta ottenuto il $Ca2$ di B , possiamo sommare A e B_{Ca2} per ottenere $A - B$, scartando il C_{out} generato.



Nel caso in cui volessimo **comparare due sequenze di N bit** per vedere se siano **esattamente uguali**, possiamo utilizzare una porta logica che possiede già questa funzione, ossia la porta **XNOR**. Basterebbe quindi implementare un numero N di XNOR che possano comparare a coppie gli N bit, per poi far passare tutti gli output attraverso una porta AND.



Se invece volessimo sapere se la **sequenza A è un valore minore della sequenza B**, ci basterebbe **sottrarre B ad A**, per poi andare a prendere **solo il bit più significativo** del risultato, poiché si tratta comunque di un risultato espresso in Ca2: se l'**MSB** è 1, allora $A < B$, altrimenti $A \geq B$.



4.1.3 Shifter

Attenzione: prima di studiare questa sezione è consigliato ripassare le sezioni [1.1.2](#) e [1.2.2](#)

- **Shift logico:** il valore viene shiftato a sinistra o destra, riempiendo gli spazi vuoti con degli 0

$$11011 \gg 2 = 00110$$

$$11011 \ll 2 = 01100$$

- **Shift aritmetico:** uguale allo shift logico, tuttavia durante uno shift a destra gli spazi vengono riempiti con il precedente MSB

$$11011 \ggg 2 = 11110$$

$$11011 \lll 2 = 01100$$

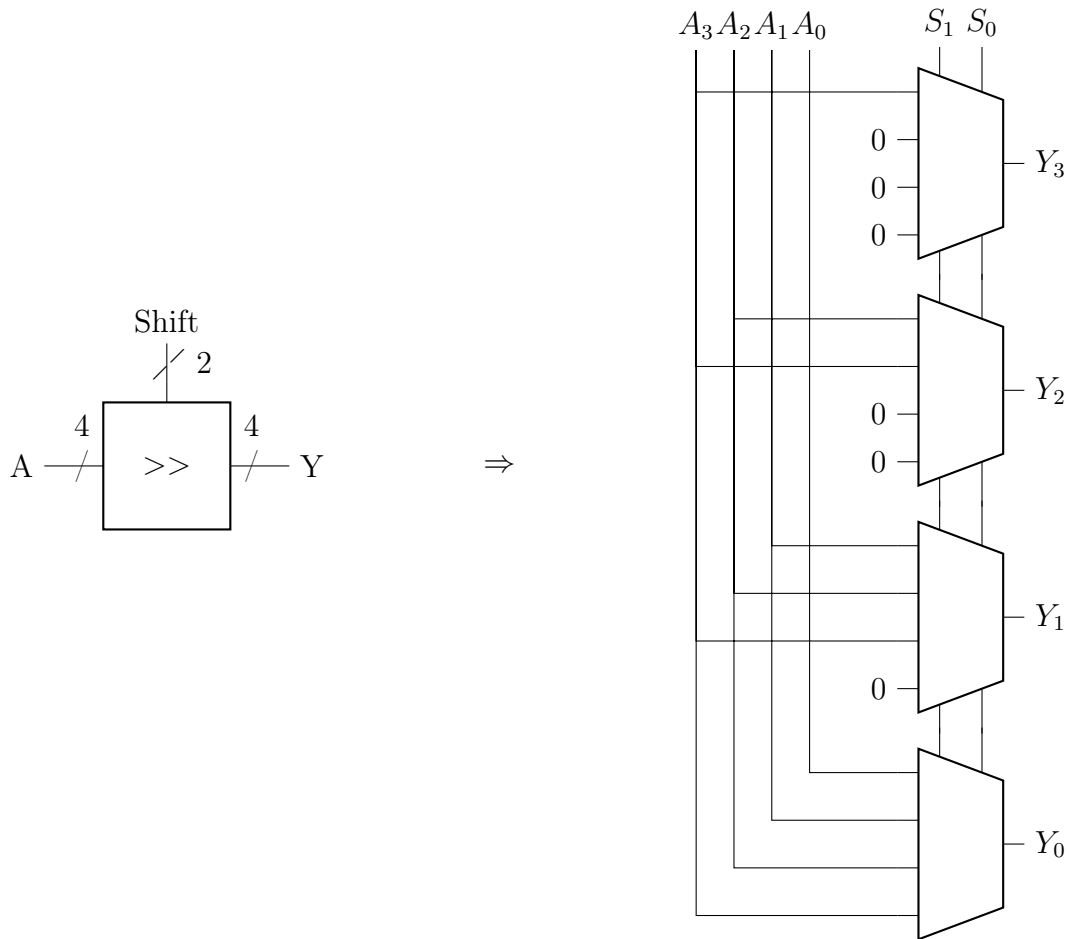
- **Rotazione:** i bit vengono ruotati in circolo verso destra o sinistra, in modo che i bit shiftati "rientrano" dal lato opposto

$$11011 \text{ ROR } 2 = 11110$$

$$11011 \text{ ROL } 2 = 01111$$

Per implementare uno **shifter da N bit**, è necessario implementare **un mux N:1 per ogni bit**, dove i selettori dei mux corrispondono alla quantità di bit da shiftare.

Vediamo l'implementazione di uno **shifter logico a destra da 4 bit**:



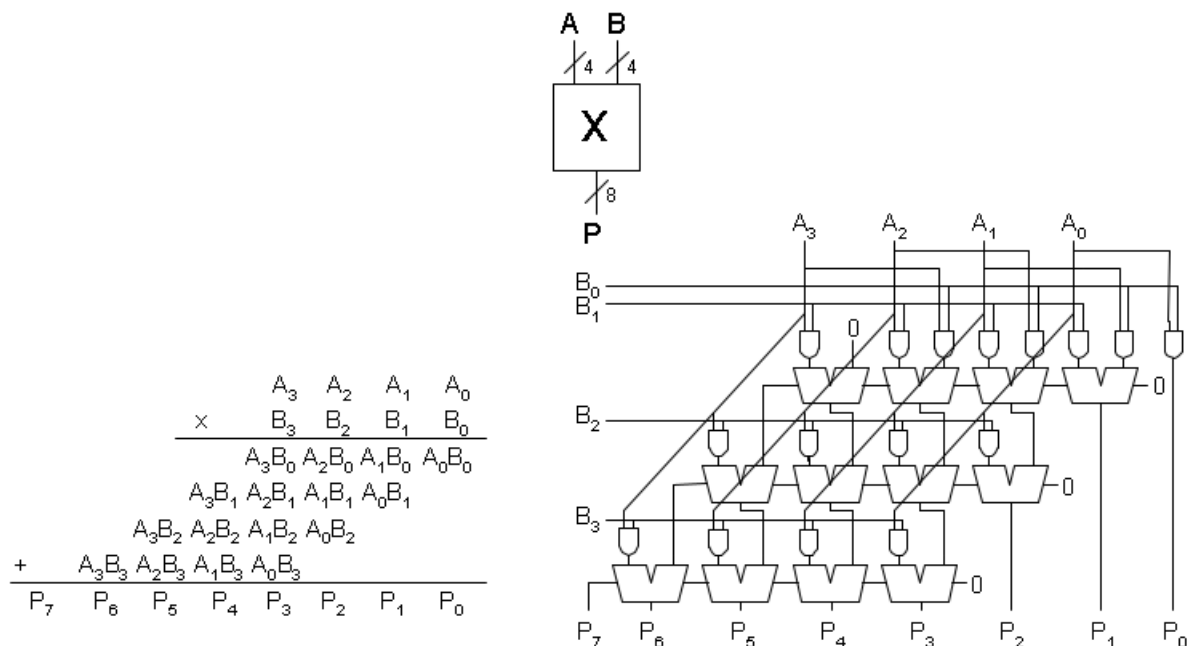
4.1.4 Moltiplicatori

Proprio come accade normalmente nel sistema binario (sezione 1.1.2), nell'ambito dei circuiti le moltiplicazioni possono essere svolte come una **somma di prodotti parziali**, dove ogni prodotto parziale corrisponde ad una **fila di sommatore**:

1 0 1 x	
1 1 1 1 =	
1 0 1 +	
¹ 1 0 1 +	
¹ 1 0 1 +	
¹ 1 0 1 =	
1 0 0 1 0 1 1	

\Rightarrow

Abbiamo bisogno di **3 file di sommatore**



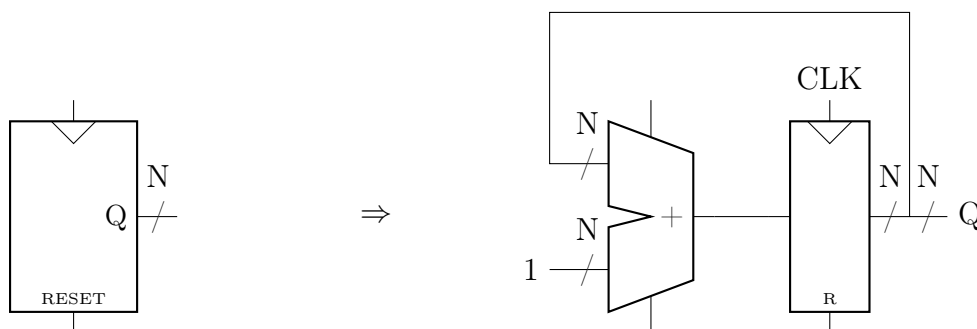
4.2 Blocchi utilitari

Vediamo ora due tipologie di blocchi costruttivi digitali **utilitari**, ossia estremamente comodi in molte situazioni:

- Contatori
- Registri con shift

4.2.1 Contatore

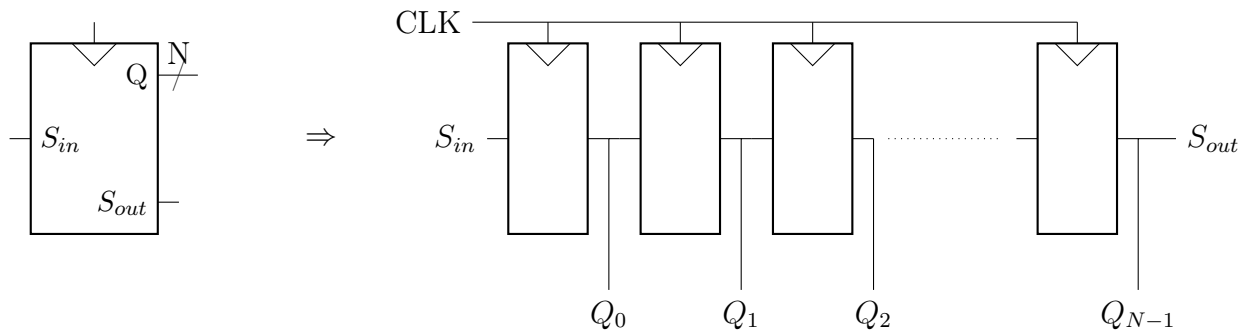
Un **contatore** è un blocco digitale il cui valore memorizzato all'interno viene **incrementato** ad ogni colpo di clock. Viene utilizzato per **ciclare tra un set di numeri** definiti in base alla quantità di flip-flop contenuti al suo interno (es: con 3 flip-flop avremo la sequenza 000, 001, 010, 011, 100, 101, 110, 111, 000, 001, ...).



NB: nonostante l'input secondario in ingresso al sommatore sia **un semplice 1**, ricordiamo che per effettuare una somma siano necessari comunque **N bit** (si veda la figura), dunque il valore in ingresso corrisponderà a 000...001.

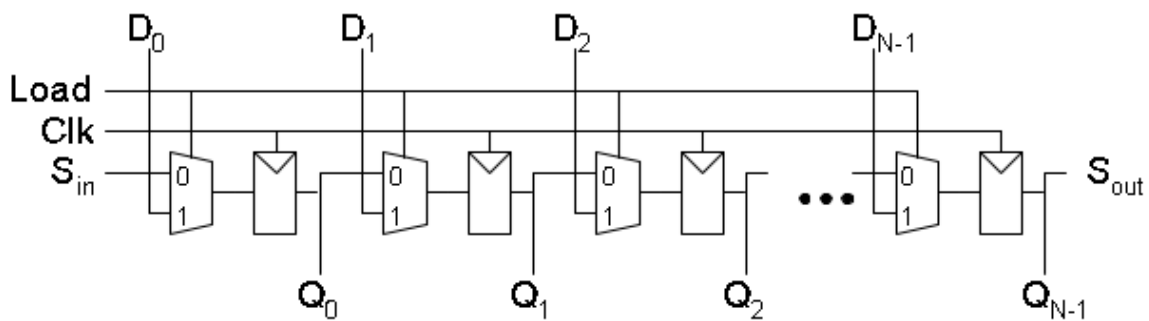
4.2.2 Shift Register

Una particolare tipologia di registri sono i **registri con shift**, dove ad ogni colpo di clock un bit viene **trasferito** da un registro all'altro. Poiché è connesso un output ad ogni registro, questo blocco digitale viene chiamato anche **serial-to-parallel converter**, dove l'input seriale S_{in} viene convertito in N output paralleli.



Esiste anche una variante del normale shift register, ossia uno **shift register con carico parallelo**, dove ad ogni registro viene **aggiunto un mux** che vada a selezionare uno solo tra l'input seriale (dunque l'output del registro precedente ad esso) ed un input esterno.

Tutti i mux vengono controllati dallo stesso segnale, chiamato **Load**. Se $\text{Load} = 1$ allora i registri vengono caricati con il loro rispettivo "input esterno", mentre se $\text{Load} = 0$ essi si comportano come un normale shift register.

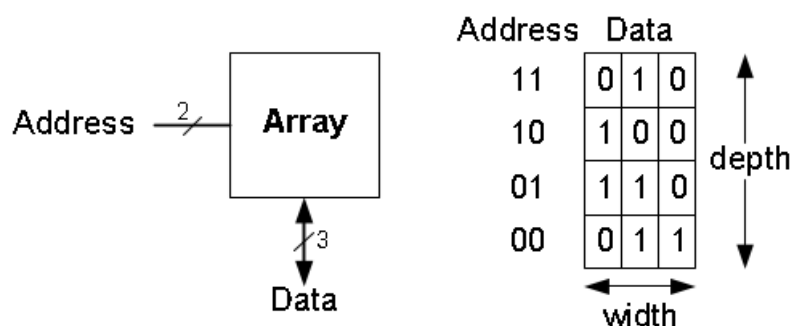


Per via del suo doppio comportamento, questo blocco può agire sia come un **serial-to-parallel converter** ($S_{in} \rightarrow Q_{[N-1],0}$) sia come un **parallel-to-serial converter** ($D_{[N-1:0]} \rightarrow S_{out}$).

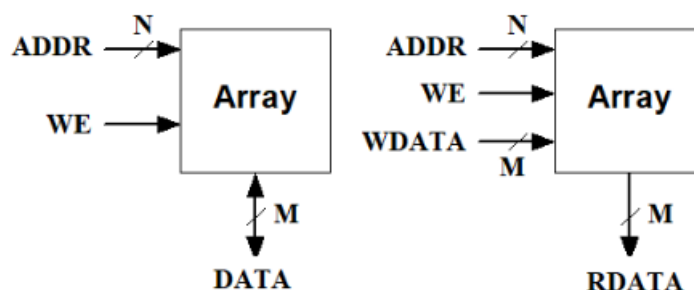
4.3 Memorie

Gli **array di memoria** costituiscono un modo efficiente per immagazzinare grandi quantità di dati. Ogni array di memoria possiede **N input** con cui è possibile codificare 2^N **indirizzi univoci**. Ad ogni indirizzo corrisponde una **parola di M bit**, in cui vengono effettivamente memorizzati i dati.

Una memoria, quindi, corrisponde ad una **matrice** di 2^N **righe** (profondità) e **M colonne** (larghezza).



ATTENZIONE: è necessario puntualizzare come gli M bit **Data** siano sia di input, sia di output, poiché essi vengono usati **sia per la scrittura che per la lettura**. A scegliere quale delle due operazioni venga effettuata, è un ulteriore bit chiamato **WE**, abbreviativo di **write enable**, ossia "abilita scrittura".



Esistono tre tipologie comuni di memorie:

- Static Random Access Memory (SRAM)
- Dynamic Random Access Memory (DRAM)
- Read-only Memory (ROM)

Le prime due vengono definite **memorie volatili**, poiché i dati memorizzati al loro interno vengono persi una volta scollegata l'alimentazione elettrica, mentre la terza viene definita **non volatile**, poiché in grado di memorizzare permanentemente l'informazione al suo interno. Inoltre, quest'ultima è una memoria di **sola lettura**, dunque non è possibile sovrascrivere ciò che è memorizzato.

PS: attualmente si è un po' perso il significato di Read-only in quanto, fatta eccezione di alcune tipologie, tutte le moderne ROM sono riscrivibili (ad esempio le [EEPROM](#))

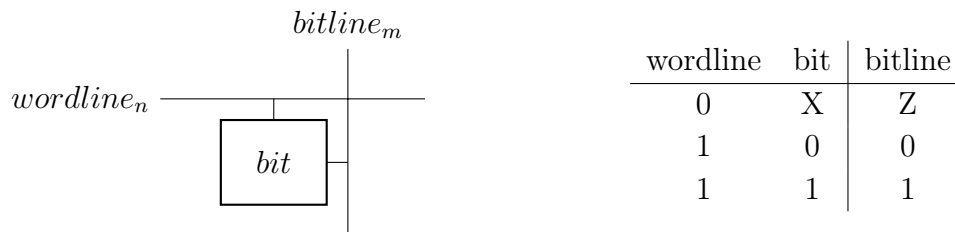
4.3.1 Celle di bit delle memorie

Abbiamo quindi detto che una memoria corrisponde a nient'altro che una **matrice di bit**, dove ogni bit viene memorizzato in una **cella**.

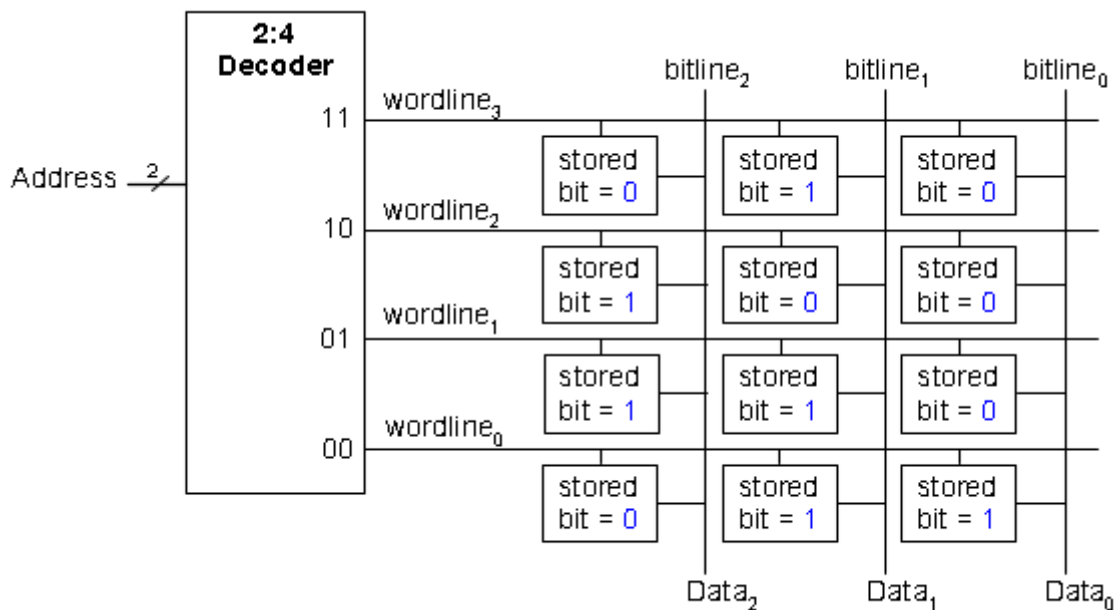
Ogni parola all'interno della memoria è costituita da M celle. Per leggere i valori contenuti nelle parole, ad ogni cella viene collegato un segnale chiamato **bitline**, che ne trasporta il valore fino all'output della memoria.

Abbiamo inoltre detto che ogni parola viene identificata da un **indirizzo di memoria**, codificato da N bit in ingresso per un totale di 2^N indirizzi. Ogni indirizzo corrisponde ad una **wordline**, ossia un segnale connesso a tutte le celle costituenti la parola identificata dall'indirizzo stesso. Possiamo quindi dire che il segnale di wordline funzioni come un **segnale di enable lungo tutta la parola**, dove solo una wordline può essere attiva alla volta (one-hot).

Ricapitolando, se la wordline della parola è attiva, allora il valori contenuti nelle celle della parola vengono **letti/scritti**, altrimenti verrà dato un segnale di **alta impedenza (Z)** sulla bitline, in modo che solo la parola selezionata sia in grado di dettarne il valore.



Poiché **solo una wordline** deve essere attiva alla volta, viene logico immaginare come sia possibile implementare un **decoder $N:2^N$** come elemento in grado di selezionare una sola (ricordiamo che il decoder sfrutta una codifica **one-hot**) determinata wordline in base all'indirizzo di input dato:



4.3.2 DRAM, SRAM e ROM

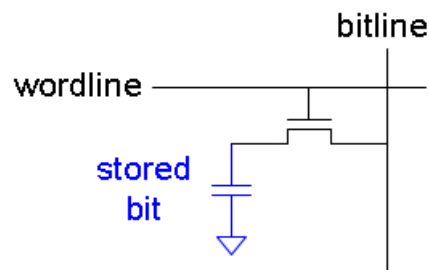
- **Dynamic RAM (DRAM)**

- Il dato viene conservato in un **condensatore**. Se il condensatore è carico allora il bit varrà 1, altrimenti varrà 0
- Viene definita **dinamica** poiché i valori memorizzati devono essere **rinfrescati** dopo una determinata quantità di tempo, poiché i condensatori **perdono le cariche** al loro interno sia col passare del tempo sia durante ogni operazione di lettura

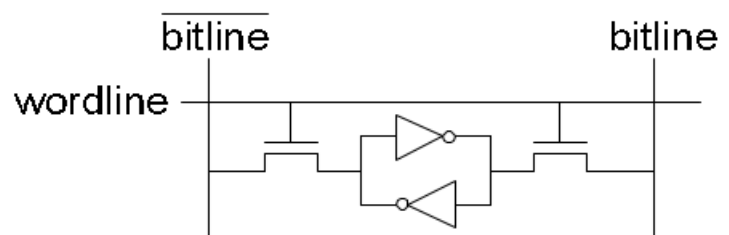
- **Static RAM (SRAM)**

- Il dato viene conservato da un **simil circuito bistabile**
- Richiedono **molti più transistor** rispetto ad una DRAM, per un totale di 6 transistor (2 nel circuito più 2 nel circuito interno di ogni inverter). Tuttavia, al contrario della DRAM, il dato viene conservato in modo **statico**, dunque **non è necessario rinfrescare** il dato periodicamente

Dynamic RAM



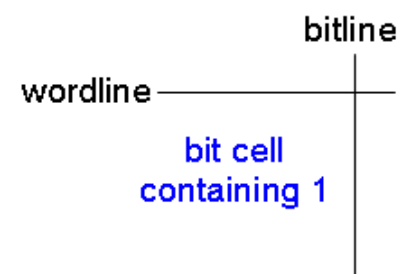
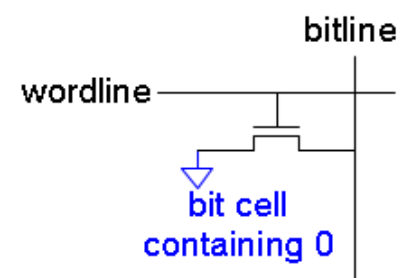
Static RAM



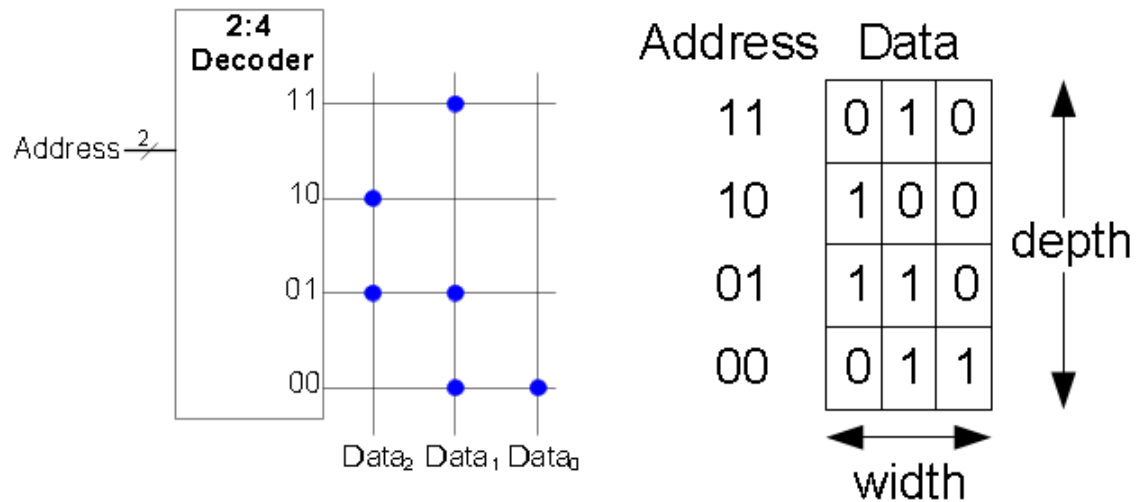
ROM

- **ROM**

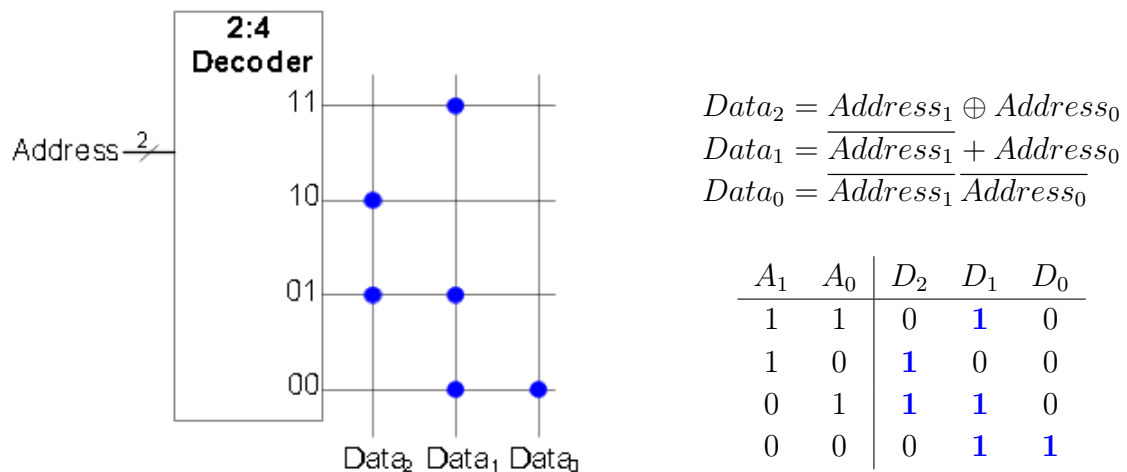
- Il dato non viene realmente "conservato", poiché in realtà **ogni cella** contenente un **bit uguale a 0** corrisponde ad un **un cavo fisicamente collegato alla terra della corrente (ground)**. In questo modo, **ogni altra cella** non connessa alla terra verrà identificata come **un bit uguale ad 1**.
- In questo modo, ogni volta che viene accesa l'alimentazione, la ROM avrà già "al suo interno" gli **stessi identici dati** delle volte precedenti (per questo motivo le ROM non sono riscrivibili, ma solo leggibili)



Per via del modo in cui sono costruite, i dati all'interno delle ROM possono essere rappresentati anche tramite quella che viene chiamata **Dot Notation**, dove viene inserito un **pallino** al posto di ogni cella contenente un **bit uguale ad 1**:



Inoltre, ogni memoria può essere utilizzata per **compiere operazioni logiche**: considerando gli **output Data** come degli **output di un'equazione booleana**, possiamo effettuare tale equazione tramite i bit in memoria sulla **colonna** stessa dell'output, poiché ad ogni riga corrisponde un **mintermine**. Di conseguenza, le memorie possono essere usate anche come un modo per conservare le **tabelle della verità**.



4.4 Array Logici

Gli **array logici** sono dei componenti al cui interno vengono effettuate **operazioni logiche in serie**. Vengono suddivisi in due categorie principali:

- **Programmable Logic Array (PLA)**
- **Field Programmable Gate Array (FPGA)**

4.4.1 Programmable Logic Array

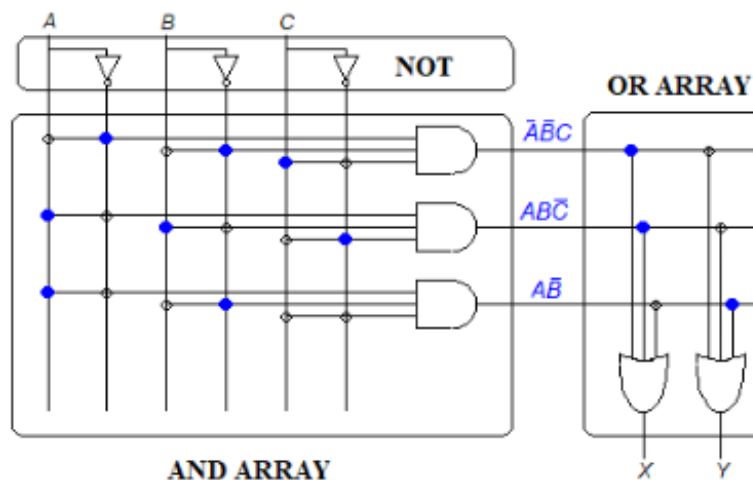
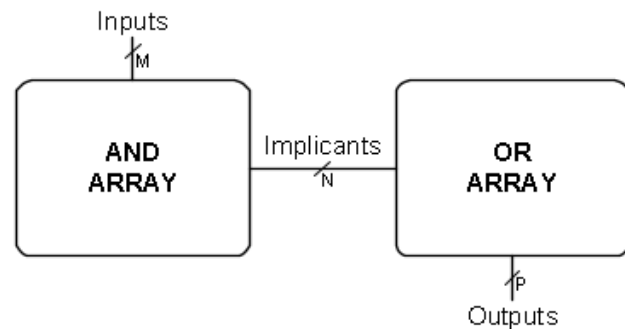
Una PLA è una **rete combinatoria** con **M ingressi**, **N implicanti** e **P uscite**, costituita da **tre piani interni**:

- Un **piano NOT**, dove tutti gli M ingressi vengono invertiti, in modo da avere ogni ingresso e il suo complementare (dunque ottenendo tutti i literals)
- Un **piano AND**, in cui alcuni literals generano N implicanti passando attraverso degli AND connessi in modo fisso
- Un **piano OR**, in cui alcuni implicanti generano P output passando attraverso degli OR connessi in modo fisso

Ad ogni output corrisponde una **funzione logica**, espressa in **forma SOP** (poiché viene effettuato un OR tra degli AND).

Esempio di PLA:

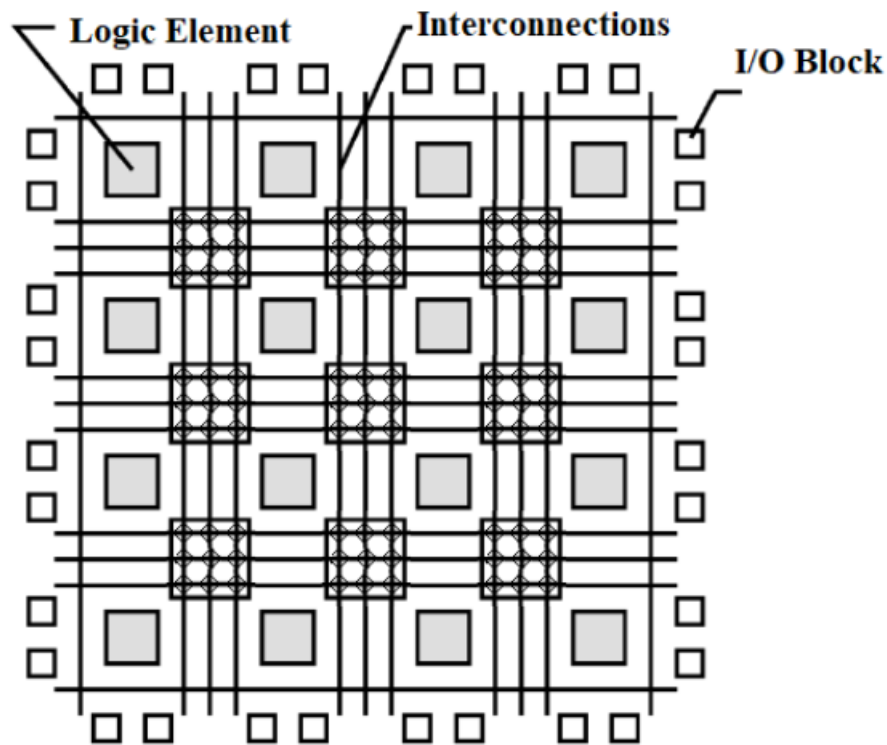
- $X = \overline{A}\overline{B}C + A\overline{B}\overline{C}$
- $Y = A\overline{B}$



4.4.2 Field Programmable Gate Array

Una **Field Programmable Gate Array** è una rete combinatoria e sequenziale composta da un insieme di **tre tipologie di componenti**:

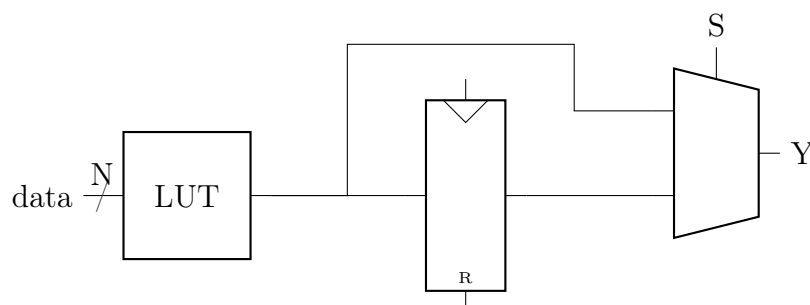
- **Elementi logici o Logic Element (LE)**, che si occupano della parte logica
- **Blocchi I/O**, ossia elementi di input/output che si interfacciano col mondo esterno
- **Interconnessioni programmabili**, che connettono i LE ai blocchi I/O



Logic Element

Ogni **Logic Element** è a sua volta composto da un insieme di **tre elementi**:

- **Lookup Table (LUT)**, ossia una tabella della verità memorizzata per poter svolgere operazioni di **logica combinatoria**
- **Flip Flop**, per poter svolgere operazioni di **logica sequenziale**. Solitamente si tratta di un **registro programmabile**
- **Multiplexer**, per connettere le LUT ai FF



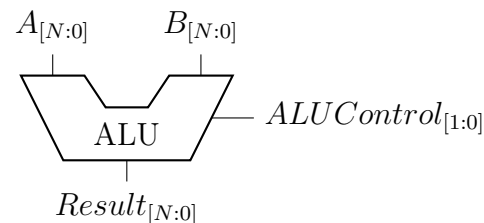
4.5 Arithmetic Logic Unit (ALU)

L'**ALU** è uno dei blocchi digitali **fondamentali**. Dati due input da N bit, si occupa di svolgere **tutte le operazioni logiche ed aritmetiche** di base su di essi, in particolare:

- Addizione
- Sottrazione
- AND
- OR

Poiché si tratta di un **unico blocco** in grado di svolgere **4 operazioni**, è ovviamente necessario implementare un segnale, chiamato **ALUControl**, che possa andare a **selezionare quale operazione** andare ad effettuare sui due segnali in input:

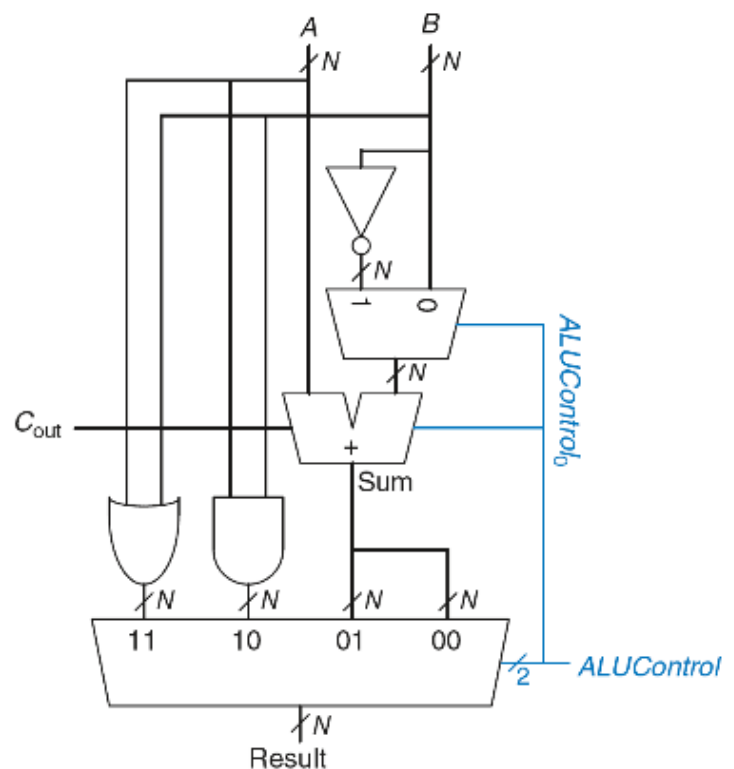
ALUControl _[1:0]	Operazione
00	Addizione
01	Sottrazione
10	AND
11	OR



Sulla destra vediamo il **circuito interno di un'ALU**. Nonostante esso sembri a prima vista complesso, in realtà può essere scomposto in **3 blocchi**: un **blocco OR**, un **blocco AND** ed un **blocco Sommatore**.

Notiamo tuttavia l'**assenza** di un **blocco Sottrattore**, poiché (come visto nella sezione [4.1.2](#)) per effettuare una sottrazione in realtà utilizziamo comunque un sommatore. Dunque, possiamo implementare un **MUX 2:1 interno**, controllato dal segnale $ALUControl_0$, che vada a selezionare se svolgere un'addizione o una sottrazione (si noti anche che tale bit viene dato in input anche come carry, poiché, in caso si scelga di effettuare una sottrazione, esso varrebbe 1, in modo da convertire B in Ca_2).

Infine, viene implementato un **MUX 4:1 esterno** che vada a selezionare **solo una delle 4 operazioni**.



4.5.1 ALU con Flag di stato

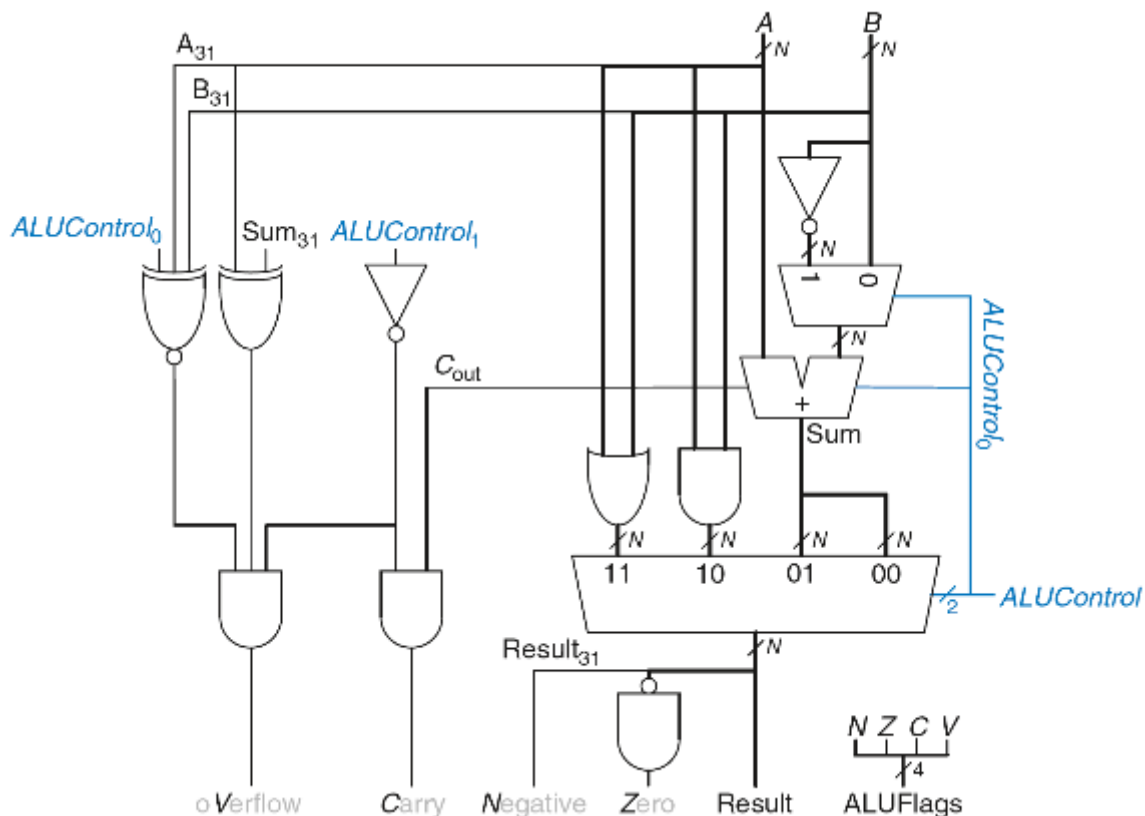
Le moderne ALU sono dotate anche di 4 output aggiuntivi chiamati **Flag di stato** che danno **informazioni aggiuntive relative al risultato** dell'operazione logica/matematica appena effettuata:

- **N** - Il risultato è un numero **negativo**, dunque l'**MSB** del risultato vale **1**.
- **Z** - Il risultato è uno **zero**, dunque **tutti i bit** del risultato sono **0**.
- **C** - L'operazione ha generato un **riporto** (carry), dunque il **C_{out}** del sommatore vale **1** e l'operazione svolta dall'ALU è un'**addizione** o una **sottrazione**
- **V** - L'operazione ha generato un **overflow**, dunque è stata svolta una **somma** tra due **numeri di segno uguale** (oppure una **sottrazione** tra due **numeri di segno opposto**) in cui il **risultato** generato è di **segno opposto**.

ATTENZIONE: ricordiamo che il sommatore effettua calcoli in **Ca2**, dunque:

→ 0111 + 0001 = 1000 equivale a $7 + 1 = -8$

→ $0111 - 1010 = 0111 + 0110 = 1101$ equivale a $7 - (-6) = -3$



NB: Lo schema riporta un'ALU a **32 bit**, dunque ogni bit 31 corrisponde sempre ad un **MSB**. Infatti, Sum_{31} corrisponde all'MSB risultato della somma tra A e B, dunque $Sum_{31} = Result_{31}$. Inoltre, da questo schema possiamo dedurre che $ALUControl_0$ si occupi di **selezionare** se svolgere un operazione logica o matematica, mentre $ALUControl_1$ si occupi di **selezionare** se svolgere un AND/Addizione oppure un OR/Sottrazione.

Capitolo 5

Linguaggi descrittivi dell'Hardware

I linguaggi descrittivi dell'hardware, in inglese **Hardware Descriptive Language (HDL)**, sono linguaggi in grado di interpretare del codice scritto descrivente i comportamenti e la struttura di un circuito, **sintetizzandolo** e producendone un modello **schematizzato**. I due HDL più utilizzati a livello commerciale sono il **SystemVerilog** (che vedremo in questo corso) e il **VHDL 2008**.

I vantaggi principali degli HDL prevedono la possibilità di poter effettuare **simulazioni** del circuito, permettendo ai progettisti di risparmiare enormi quantità di tempo e denaro, e di calcolare automaticamente il **minor numero di porte** necessarie

5.1 Moduli Comportamentali e Strutturali

Nel linguaggio SystemVerilog, i moduli (ossia i componenti) possono essere descritti secondo due modalità:

- **Comportamentale**, ossia descrivendo le operazioni che il modulo deve eseguire
- **Strutturale**, ossia descrivendo il modo in cui i sotto-moduli al suo interno sono interconnessi tra loro, in modo da formare il modulo esterno

5.1.1 Moduli Comportamentali

La sintassi prevista dal SystemVerilog per poter descrivere le operazioni svolte da un modulo è **molto simile** a quella di un normale linguaggio di programmazione, dove un modulo può essere interpretato come una **funzione** alla quale vengono passati degli **argomenti** (ossia gli input del modulo) per poi **restituire dei valori** (ossia gli output del modulo) dopo aver effettuato **operazioni** su di essi:

```
module example(input logic a, b, c,
               output logic y);

    assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;

endmodule
```

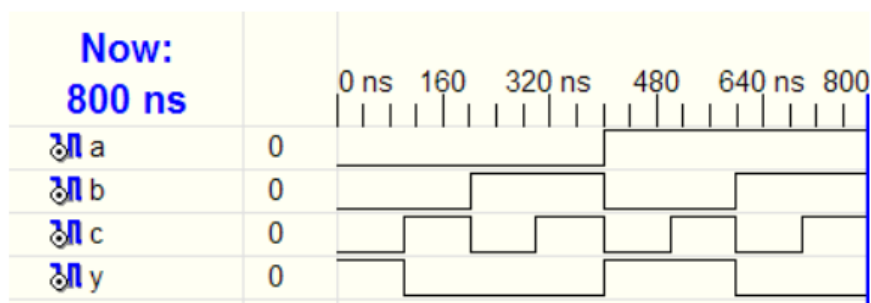
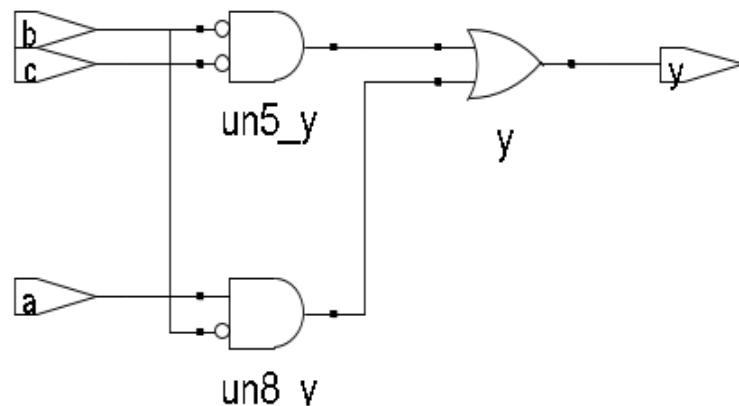
Legenda:

- **module/endmodule**: indicano l'inizio e la fine del modulo. Tutto ciò che vi è all'interno farà parte del modulo.
- **example**: è il nome del modulo e viene deciso dal progettista che scrive il codice
- **input/output**: descrivono le liste di valori in input e in output
- **Operatori Logici**:
 - \sim : NOT
 - $\&$: AND
 - $|$: OR

Il codice descritto sopra, dunque, corrisponde ad un modulo chiamato **example** con i valori **A**, **B** e **C** in input e il valore **Y** in output. All'interno di questo modulo viene svolta solo un'operazione di logica combinatoria, corrispondente all'equazione booleana:

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

Una volta descritto il nostro modulo, il codice verrà letto dall'interprete del SystemVerilog, generando una versione del circuito con il **minor numero di porte possibile**, ottenuta **semplificando** le equazioni booleane descritte, ed una **simulazione** di esso:



Altre regole sintattiche previste dal SystemVerilog sono:

- Ogni parola è **case sensitive**: *reset* e *Reset* non sono la stessa cosa
- I nomi dei moduli non possono cominciare con un **numero**: *2mux* è un nome invalido
- I **commenti del codice** vengono indicati con
 - `//` → Commento a linea singola
 - `/* ... */` → Commento multi-linea

5.1.2 Moduli strutturali

Una volta definiti dei moduli comportamentali semplici, è possibile utilizzarli per poter creare un modulo strutturale:

```
module and3(input logic a, b, c,
           output logic y);

    assign y = a & b & c;
endmodule

module inv(input logic a,
           output logic y);

    assign y = ~a;
endmodule

module nand3(input logic a, b, c
            output logic y);

    logic n1;
    /* è un segnale interno e serve
    a creare l'interconnessione tra
    il modulo and3 e il modulo inv */

    and3 andgate(a, b, c, n1);      //istanza di and3

    inv inverter(n1, y);          //istanza di inv
endmodule
```

In questo modo, abbiamo creato un modulo strutturale chiamato **nand3** composto da due **istanze** (ossia delle "copie") dei due moduli comportamentali **and3** e **inv** interconnessi tra loro.

5.2 Convenzioni del SystemVerilog

Di seguito vedremo le numerose **convenzioni** fornite dal SystemVerilog per poter progettare circuiti di larga dimensione:

- **Operazioni su vettori di bit**
- **Operatori di riduzione**
- **Assegnamenti condizionali**
- **Variabili interne**

5.2.1 Operazioni su vettori di bit

Nel linguaggio SysVerilog, è possibile indicare un **vettore di bit** con la notazione $A[i : j]$, indicando una quantità di bit che vanno da j ad i (es: la notazione $A[3 : 0]$ indica un **vettore di 4 bit**)

Nel caso in cui venga svolta un **operazione** su un vettore di bit, essa verrà svolta su **ogni bit** appartenente al vettore.

Esempio:

- La **forma ridotta**:

```
module and_as_array(input logic [2:0] a, b,
                    output logic [2:0] y);

    assign y = a & b;
endmodule
```

- Corrisponde alla **forma estesa**:

```
module and_as_bits(input logic a0, a1, a2, b0, b1, b2
                  output logic y0, y1, y2);

    assign y0 = a0 & b0;
    assign y1 = a1 & b1;
    assign y2 = a2 & b2;
endmodule
```

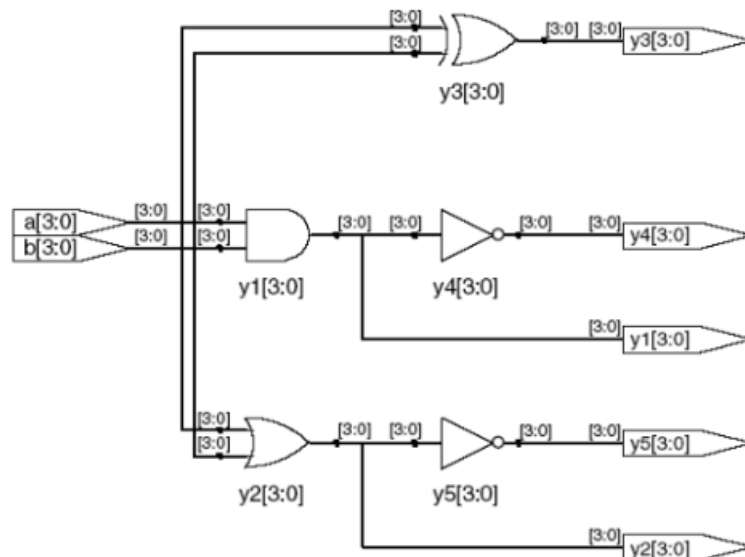
L'uso di questa sintassi ci permette notevolmente di **velocizzare** la scrittura di componenti che eseguono operazioni con un grande numero di bit in ingresso e in uscita

Esempio

Questo **breve codice** corrisponde ad un componente che svolge tutte le operazioni logiche su due vettori di 4 bit:

```
module gates(input logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);

    assign y1 = a & b;    // AND
    assign y2 = a | b;    // OR
    assign y3 = a ^ b;    // XOR
    assign y4 = ~(a & b); // NAND
    assign y5 = ~(a | b); // NOR
endmodule
```



5.2.2 Operatori di riduzione

Nel caso in cui si debba eseguire un'operazione **tra** (e non su) **tutti i bit di un vettore**, è possibile utilizzare la seguente convenzione:

```
module and8(input logic [7:0] a,
            output logic y);

    assign y = &a;

    /* è molto più rapido che scrivere
    assign y = a[7] & a[6] & a[5] & a[4] &
               a[3] & a[2] & a[1] & a[0];
    */
endmodule
```

5.2.3 Assegnamenti condizionali

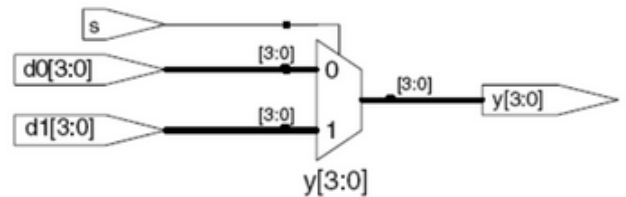
Nel caso in cui si debba andare a svolgere un assegnamento condizionale, ossia selezionare uno solo tra due valori possibili in base ad una condizione (come nel caso di un **mux**) è possibile utilizzare quello che viene chiamato **operatore ternario**:

```
assign output = condizione ? valore_se_vera : valore_se_falsa
```

Vediamo l'implementazione di un modulo **mux2** utilizzando un operatore ternario:

```
module mux2(input logic [3:0] d0, d1,
            input logic s,
            output logic [3:0] y);

    assign y = s ? d1 : d0;
endmodule
```



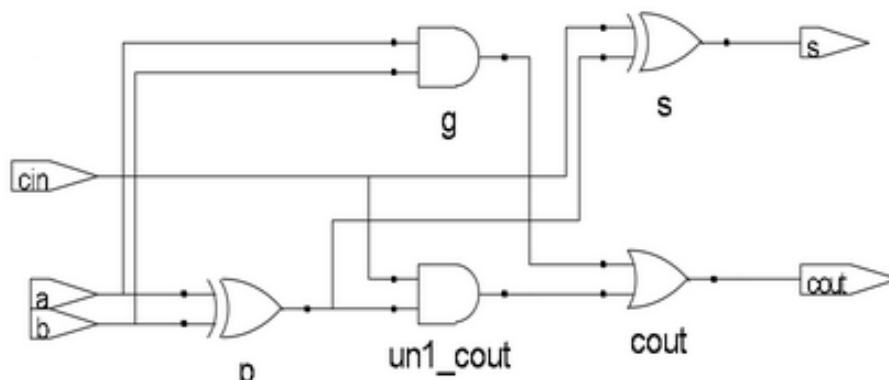
5.2.4 Variabili interne

Il SysVerilog permette anche la creazione di variabili interne ad un modulo, ossia dei **nod** **interni** che non corrispondono agli input ed output del modulo stesso, ma sono tuttavia utilizzati al suo interno:

```
module fulladder(input logic a, b, cin,
                 output logic s, cout);

    logic p, g;    //nodi interni

    assign p = a ^ b;
    assign g = a & b;
    assign s = p ^ cin;
    assign cout = g | (p & cin);
endmodule
```



5.3 Numeri e Manipolazione di bit

Ovviamente, vi sono casi in cui è necessario indicare dei valori non come segnale o variabile ma come numeri veri e propri. In SystemVerilog, i numeri vengono rappresentati con la seguente forma:

$$N'Bvalore$$

Dove **N** corrisponde al **numero di bit**, **B** alla **base** e **valore** al valore effettivo del numero

ATTENZIONE: la clausola $N'B$ è **opzionale**, tuttavia è estremamente **raccomandata**, poiché altrimenti verrebbe sempre interpretato come un **numero decimale di dimensione massima**. Inoltre, i `_` vengono ignorati dall'interprete, dunque possono essere utilizzati come separatore di **comodità**

Numero	N Bit	Base	Val. Decimale	Bit reali
3'b101	3	2	5	101
'b11	unsized	2	3	00...0011
8'b11	8	2	3	00000011
8'b1010_1011	8	2	171	10101011
3'd6	3	10	6	110
6'o42	6	8	34	100010
8'hAB	8	16	171	10101011
42	Unsized	10	42	00...0101010

Manipolazione di Bit

In SysVerilog è possibile manipolare i vettori di bit utilizzando molte convenzioni:

```
// { } indicano sempre un vettore di bit
// se un numero è presente davanti a {},
// allora quel vettore viene ripetuto per N

assign y = {a[2:1], {3{b[0]}}, a[0], 6'b100_010};

// se y è un segnale a 12-bit, l'istruzione qua sopra produce:

y = { a[2] a[1] b[0] b[0] b[0] a[0] 1 0 0 0 1 0 }

// inoltre, possiamo modificare anche
// un singolo bit di un vettore

Y[0] = 1'b1;
Y[1] = 1'b0;
```

5.4 Datatype logici del SystemVerilog

Con il termine **DataTypes logici** intendiamo l'insieme dei **valori logici** che possono essere assunti da un componente. Nella sezione precedente abbiamo visto i **numeri**, che tuttavia corrispondono tutti a due soli datatype, ossia **0 ed 1**, poiché ogni numero viene convertito sempre in binario. Tuttavia, sappiamo dell'esistenza anche di due altri valori assumibili: **Z** e **X**.

- **X** indica un **valore logico invalido o sconosciuto** (e non un **don't care**), ossia una **contesa tra 0 ed 1** (sezione 2.4.2)
- **Z** indica un **valore fluttuante**, ossia un'**alta impedenza**
- All'inizio di ogni simulazione, gli output dei **nodi di stato**, come i flip-flop, vengono inizializzati come uno **stato sconosciuto (X)**, rappresentato graficamente come un segnale nel mezzo (vedere sezione successiva)

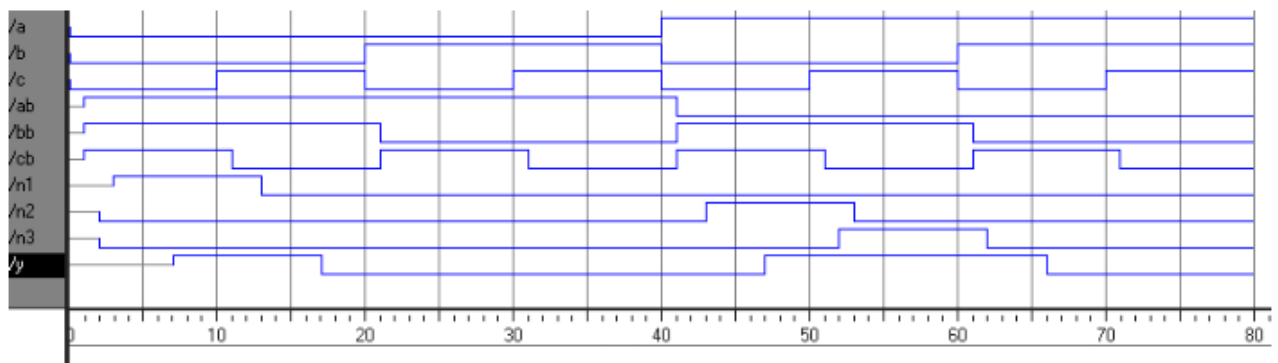
5.5 Delay

In SystemVerilog è anche possibile definire il **delay** di un componente, indicato con un **#** seguito da un numero indicante il numero di colpi di clock di ritardo. L'unità di misura viene specificata esternamente dal progettista (solitamente 1ns).

```
module example(input logic a, b, c,
               output logic y);

    logic ab, bb, cb, n1, n2, n3;

    assign #1 {ab, bb, cb} = ~{a, b, c};
    assign #2 n1 = ab & bb & cb;
    assign #2 n2 = a & bb & cb;
    assign #2 n3 = a & bb & c;
    assign #4 y = n1 | n2 | n3;
endmodule
```



N.B.: guardando l'inizio della simulazione è possibile notare facilmente i delay dei singoli componenti, poiché tutti i componenti vengono inizializzati come X per poi cambiare non appena tutte le condizioni necessarie vengono soddisfatte.

5.6 Logica Sequenziale

Per descrivere il funzionamento dei componenti che sfruttano la logica sequenziale, in SystemVerilog è possibile utilizzare l'istruzione **always** alla quale è possibile accodare una lista di **valori sensibili**, ossia quei segnali che, al loro **cambio di valore**, definiscono il momento in cui il componente deve **aggiornare** il suo valore (es: il segnale CLK in un flip-flop).

```
always @ (valori_sensibili)
    istruzione;
```

Dunque, le istruzioni inserite nel blocco **always** verranno eseguite **ogni volta** in cui uno degli **eventi** nella lista dei valori sensibili si verifica.

Tuttavia, l'istruzione **always** è **generica**, ossia non possiede una sua vera e propria logica di controllo riguardo alle istruzioni che avvengono al suo interno. Dunque, all'interno di esso la gestione delle istruzioni viene affidata al progettista.

Per aggirare questo problema, sono stati creati degli **idiomi** del SystemVerilog, ossia delle sue piccole varianti che aggiungono altre tipologie di istruzione **always** che invece possiedono una logica di controllo, segnalando un errore al progettista nel caso in cui non vengano rispettate le regole definite:

- **always_ff**: utilizzato per i flip-flop
- **always_latch**: utilizzato per i latch
- **always_comb**: utilizzato per la logica combinatoria

In questo modo, se utilizzassimo l'istruzione **always_ff** ma andassimo a definire una logica interna corrispondente a quella di un **latch**, il software segnalerebbe un'**incongruenza** tra la logica richiesta e quella definita.

5.6.1 Elementi di Stato in SysVerilog

Di seguito vedremo una carrellata di moduli con cui è possibile definire tutti i vari **elementi di stato** della logica sequenziale:

D Flip-Flop

```
module flop(input logic clk,
            input logic [3:0] d,
            output logic [3:0] q);

    //posedge è abbreviativo di "positive-edge",
    //indicando quindi il rising edge del clock

    always_ff @(posedge clk)
        q <= d;    // pronunciato "Q prende D"

endmodule
```

Resettable D Flip-Flop (Sincrono)

```
module r_flop_sync(input logic clk,
                  input logic reset,
                  input logic [3:0] d,
                  output logic [3:0] q);

    always_ff @(posedge clk)

        if(reset)
            q <= 4'b0;

            // 4'b0 viene automaticamente
            // convertito in 4'b0000
        else
            q <= d;
endmodule
```

Resettable D Flip-Flop (Asincrono)

```
module r_flop_async(input logic clk,
                   input logic reset,
                   input logic [3:0] d,
                   output logic [3:0] q);

    always_ff @(posedge clk, posedge reset)

        if(reset)    q <= 4'b0;
        else         q <= d;
endmodule
```

Enabled Resettable D Flip-Flop

```
module en_r_flop_async(input logic clk,
                      input logic enable,
                      input logic reset,
                      input logic [3:0] d,
                      output logic [3:0] q);

    always_ff @(posedge clk, posedge reset)

        if(reset)        q <= 4'b0;
        else if(enable)  q <= d;
endmodule
```


D Latch

```

module latch(input logic clk,
             input logic [3:0] d,
             output logic [3:0] q);

    always_latch
        if(clk)    q <= d;

endmodule

```

5.6.2 Case e Casez

Oltre all'if, è possibile utilizzare l'istruzione **case** per definire una lunga serie di possibilità a seconda del **valore assunto da un segnale**. Sotto vediamo un modulo **sevenseg** corrispondente ad un **display a 7 segmenti**, il cui valore cambia a seconda del valore assunto dal segnale **data**:

```

module sevenseg(input logic [3:0] data,
                output logic [6:0] segments);

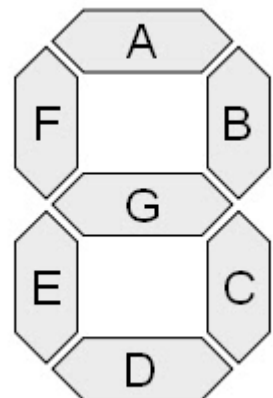
    always_comb
        case (data)
            // abc_defg

            0: segments = 7'b111_1110;
            1: segments = 7'b011_0000;
            2: segments = 7'b110_1101;
            3: segments = 7'b111_1001;
            4: segments = 7'b011_0011;
            5: segments = 7'b101_1011;
            6: segments = 7'b101_1111;
            7: segments = 7'b111_0000;
            8: segments = 7'b111_1111;
            9: segments = 7'b111_0011;
            default: segments = 7'b000_0000;

            // il caso di default è obbligatorio poichè
            // abbiamo utilizzato l'istruzione always_comb,
            // altrimenti il software segnalerebbe
            // una logica corrispondente ad un latch

        endcase
endmodule

```



A differenza dell'istruzione `case`, l'istruzione **casez** permette al valore da comparare con le varie possibilità di assumere anche dei valori **Z** o **?** (ossia il modo in cui vengono rappresentati i don't care in SysVerilog). Ad esempio, utilizzando il `casez` potremmo realizzare un **circuito prioritario**:

```
module priority_casez(input logic [3:0] a,
                    output logic [3:0] y);

    always_comb
        casez(a)

            // ? è un don't care

            4'b1???: y = 4'b1000;
            4'b01??: y = 4'b0100;
            4'b001?: y = 4'b0010;
            4'b0001: y = 4'b0001;
            default: y = 4'b0000;
        endcase
endmodule
```

5.6.3 Assegnamento Bloccante e Non-bloccante

In SystemVerilog esistono due tipologie di **operatore di assegnamento**, `<=` e `=`.

Il primo viene detto **non-bloccante**, poiché tutte le operazioni presenti all'interno di un blocco di codice che utilizzano questo operatore vengono eseguite **in sincrono**. Il secondo, invece, viene detto **bloccante**, presenti all'interno di un blocco di codice che utilizzano questo operatore vengono eseguite **in successione**, dunque nell'esatto ordine in cui sono scritte.

L'**utilizzo scorretto** di questi due operatori spesso porta ad una **logica sequenziale errata**. Vediamo due esempi dello stesso codice ma con i due operatori diversi:

```
module synchronizer(input logic d, clk,
                   output logic q);

    logic n1;

    always_ff @(posedge clk)
        begin
            n1 <= d;    // non-bloccante
            q  <= n1;    // non-bloccante

            //vengono eseguiti in sincrono
        end
endmodule
```

```

module synchronizer(input logic d, clk,
                    output logic q);

    logic n1;

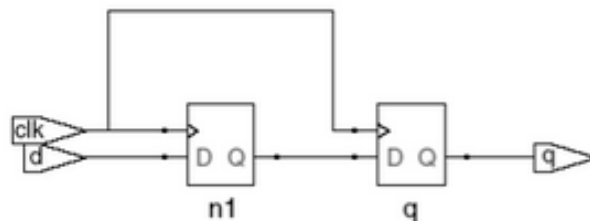
    always_ff @(posedge clk)

        begin
            n1 = d;    // bloccante
            q = n1;    // bloccante

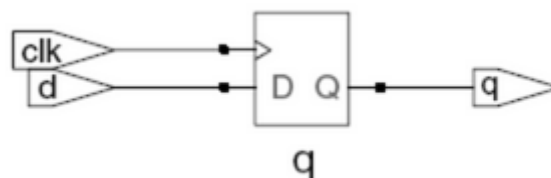
            //vengono eseguiti in successione
        end
endmodule

```

Nel **primo caso**, verranno generati **due flip-flop**, in modo che possano conservare il valore di n1 e q, restituendo q in output. Dunque, il valore di n1 e q viene aggiornato **in sincrono**, dove il valore di q diventerà il **precedente valore di n1** (dunque del colpo di clock precedente).



Nel **secondo caso**, invece, verrà generato **un solo flip-flop**, in modo che possa conservare il valore di q, restituendo q in output. Poiché i due assegnamenti avvengono **in successione**, il valore di q diventerà l'**attuale valore di n1** (dunque del colpo di clock attuale), ossia d.



Errori nell'ordine delle operazioni

Tuttavia, bisogna considerare anche un **caso particolare**, ossia il caso in cui si vada a **scrivere** su un valore **dopo** aver **letto** il valore stesso utilizzando degli **assegnamenti bloccanti**:

```
module synchronizer(input logic d, clk,
                    output logic q);

    logic n1;

    always_ff @(posedge clk)

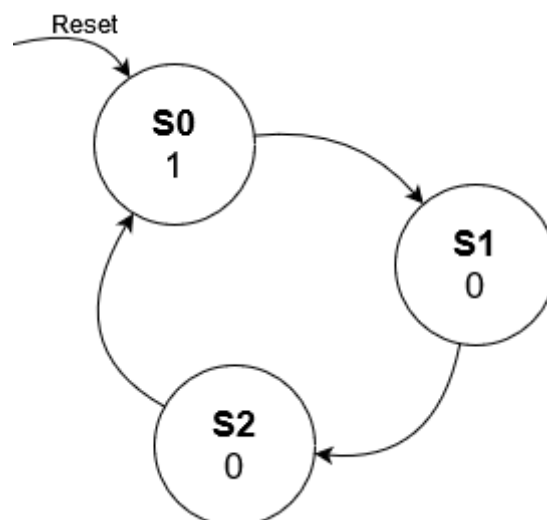
        begin
            q = n1;    // bloccante
            n1 = d;    // bloccante

            //viene generato un latch per conservare n1
        end
endmodule
```

In questo esempio andiamo **prima leggere** il valore di **n1** per **poi andarlo a modificare**. La conseguenza di questo ordine di operazioni è la **generazione di un latch aggiuntivo** che possa conservare il **precedente valore di n1**, il quale verrà assegnato a **q**. Il risultato, quindi, è lo **stesso circuito** del primo caso con operatori **non-bloccanti** che abbiamo visto precedentemente.

5.7 Macchine a Stati Finiti

Come sappiamo, le FSM sono composte da **tre blocchi**: NSL, registri e OL. Per poter descrivere una FSM in SystemVerilog, quindi, ci basta definire attentamente questi stessi tre blocchi. Di seguito vediamo l'esempio di una FSM di cui non definiremo le funzionalità, ma definiremo **solo i suoi stati** e le loro transizioni:



```
module threeStatesFSM (input logic clk, reset,
                      output logic q);

    // definiamo i tre stati della FSM
    typedef enum logic [1:0] {S0, S1, S2} statetype;

    statetype [1:0] state, nextstate;

    // registri della FSM
    always_ff @ (posedge clk, posedge reset)

        if (reset) state <= S0;
        else       state <= nextstate;

    // NSL della FSM
    always_comb
        case (state)
            S0: nextstate = S1;
            S1: nextstate = S2;
            S2: nextstate = S0;
            default: nextstate = S0;
        endcase

    // OL della FSM

    assign q = (state == S0);
    // q è il risultato della condizione (Vero o Falso)
    // se lo stato attuale è S0, allora q = 1, altrimenti q = 0

endmodule
```

5.8 Moduli di Testbench

Nel SystemVerilog, esiste un'altra tipologia di moduli, ossia dei moduli che **non svolgono alcuna funzione interna**, ma si occupano solo di **fornire degli input** ad altri moduli per poi **leggerne l'output** e verificarne la **correttezza**. Questi moduli vengono detti **testbench** e vengono utilizzati per verificare il corretto funzionamento di un modulo, che viene a sua volta chiamato **DUT (Device Under Testing)**, ossia dispositivo in via di verifica.

Immaginiamo di aver progettato il seguente **modulo**:

```
module sillyfunction(input logic a, b, c,
                    output logic y);

    assign y = ~b & ~c | a & ~b;

endmodule
```

Vogliamo ora verificare il corretto funzionamento del modulo. Per fare ciò, realizziamo un **modulo di testbench** che passerà dei valori in input al nostro DUT:

```
module testbench1();
    logic a, b, c;
    logic y;

    // definizione del DUT
    sillyfunction dut(a, b, c, y);

    // applicazione degli input al DUT

    initial begin
        a = 0; b = 0; c = 0; #10;
        c = 1; #10;
        b = 1; c = 0; #10;
        c = 1; #10;

        // il # indica un delay da aspettare
        // prima di testare un nuovo input

        a = 1; b = 0; c = 0; #10;
        c = 1; #10;
        b = 1; c = 0; #10;
        c = 1; #10;

        // guardando bene i valori assegnati,
        // possiamo notare come venga testata l'intera
        // tavola della verità di A, B e C
    end
endmodule
```

Tuttavia, questo modulo non è ancora in grado di verificare la **correttezza dell'output**. Per fare ciò, aggiungiamo un **controllo dopo ogni input**:

```
module testbench2();
    logic a, b, c;
    logic y;

    sillyfunction dut(a, b, c, y);

    initial begin
        // $display() mostra a schermo un messaggio

        a = 0; b = 0; c = 0; #10;
        if (y != 1)    $display("000 fallito!");

        c = 1; #10;
        if (y != 0)    $display("Test 001 fallito!");
    end
endmodule
```

```
    b = 1; c = 0; #10;
    if (y != 0)    $display("Test 010 fallito!");

    c = 1; #10;
    if (y != 0)    $display("Test 011 fallito!");

    a = 1; b = 0; c = 0; #10;
    if (y != 1)    $display("Test 100 fallito!");

    c = 1; #10;
    if (y != 1)    $display("Test 101 fallito!");

    b = 1; c = 0; #10;
    if (y != 0)    $display("Test 110 fallito!");

    c = 1; #10;
    if (y != 0)    $display("Test 111 fallito!");
end
endmodule
```

Testbench con dei Testvector

Come visto sopra, per testare **ogni input** è necessario definire numerose volte lo **stesso codice**. Possiamo facilmente immaginare come nel caso in cui si debba testare un DUT con un elevato numero di input ciò diventi un'enorme perdita di tempo.

Per ovviare tale problema, è possibile definire un **file testvector** (con estensione .tv) contenente gli input e gli output da testare sul DUT.

Vediamo un file corrispondente agli input passati al DUT precedentemente realizzato (le colonne corrispondono a *a*, *b*, *c* e *y_{expected}*)

Contenuto del file "example.tv"

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```

In questa tipologia più complessa di testbench, il valore in output viene confrontato durante la **discesa del clock**, dunque **falling edge** e non rising edge. Successivamente viene riportato l'intero codice del modulo testbench per verificare il DUT progettato precedentemente. Si consiglia una **lettura lenta del codice**, poiché a prima vista potrebbe sembrare *più complesso di quanto sia*:

```
module testbench3();
    logic clk, reset;
    logic a, b, c, yexpected;
    logic y;

    logic [31:0] vectornum, errors;
    // vectornum = contatore della partizione del vettore
    // errors    = contatore del numero di errori generati

    logic [3:0] testvectors[10000:0];
    // array contenente i valori letti dal file .tv

    sillyfunction dut(a, b, c, y);

    // generazione di un clock
    always
        begin
            clk = 1; #5; clk = 0; #5;
        end

    // all'inizio del testing, resetta il modulo
    // e leggi il file .tv
    initial
        begin
            $readmemb("example.tv", testvectors);
            // leggi il file e memorizzalo in testvectors

            vectornum = 0; errors = 0;
            reset = 1; #27; reset = 0;
        end

    // ad ogni rising edge, testa un nuovo input
    always @(posedge clk)
        begin
            #1; {a, b, c, yexpected} = testvectors[vectornum];
        end

    // ad ogni falling edge, controlla l'output del test
    always @(negedge clk)

        if (~reset) begin
            // se il reset è 1, salta la verifica

            if (y != yexpected) begin

                $display("Errore: Val. Input = %b", {a, b, c});
                $display("Val. Output = %b (Val. atteso: %b)", y, yexpected);
            end
        end
    end
```



```
        errors = errors + 1;

        // incrementa il valore di vectornum
        // e leggi il testvector successivo

        vectornum = vectornum + 1;

        // === e !== permettono di confrontare tra 1, 0, Z e X
        if (testvectors[vectornum] === 4'bx)
            begin
                $display("%d test completati con %d errori",
                    vectornum, errors);
                $finish;
            end
        end
    endmodule
```

5.9 Moduli parametrizzati

Fino ad ora, abbiamo visto dei moduli con degli input ed output di **dimensioni definite**. Tuttavia, è possibile realizzare dei moduli con delle **dimensioni generiche** che possono essere poi definite nel momento dell'uso

```
module mux2
    // width è un valore parametrizzato
    #(parameter width = 8)

    // definisci gli input in base al valore di width
    (input logic [width-1:0] d0, d1,
    input logic s,
    output logic [width-1:0] y);

    assign y = s ? d1 : d0;
endmodule
```

In questo modo, abbiamo definito un modulo mux2 che opera con un **numero parametrizzato di bit**. Ciò ci permette di realizzare un solo **modulo generico** per i mux, piuttosto che un modulo specifico per ogni quantità di mux:

```
mux2 #(12) mux2_12bits (d0_12, d1_12, s, out_12);
mux2 #( 7) mux2_7bits  (d0_7,  d1_7,  s, out_7 );
mux2 #(64) mux2_64bits (d0_64, d1_64, s, out_64);
```