SAPIENZA
UNIVERSITÀ DI ROMA

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Dipartimento di Informatica

Progettazione di Sistemi Digitali

Autore:
Simone Lidonnici

7 settembre 2024

Indice

1

Numeri binari

I **numeri binari** sono numeri composti solo dalle cifre 0 e 1. Per convertirli in base decimale bisogna moltiplicare ogni cifra del numero binario per 2^i, in cui i è la posizione partendo da destra e contando da 0.
Esempio:
$1010_2 = 0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 = 10_{10}$

Con N cifre in binario possiamo scrivere 2^N valori con range $[0, 2^N - 1]$

1.1 Numeri esadecimali

I **numeri esadecimali** si utilizzano per comodità perché è molto facile convertire i numeri binari in esadecimali.
Per convertire i numeri binari in esadecimale basta raggrupparli in gruppi da 4 cifre e scriverli con il numero corrispondente da 0 a 15 (i numeri dal 10 al 15 sono rappresentati dalle lettere A,B,C,D,E e F).
Esempio:
$10100110_2 \implies 1010 = A, 0110 = 6 \implies A6_{16}$

Per convertirli in decimale si utilizza lo stesso principio dei numeri binari ma con 16 al posto di 2.
Esempio:
$4AF = 15 \cdot 16^0 + 10 \cdot 16^1 + 4 \cdot 16^2 = 1199$

Le cifre di un numero esadecimale convertite in binario e decimale:

Decimale	Binario	Esadecimale
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	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

Ogni cifra di un numero binario viene chiamata **bit** e vengono raggruppati in gruppi da 8 chiamati **byte**. Il bit più a sinistra in un byte è quello più significativo e quello più a destra quello meno significativo.

1.2 Operazioni con numeri non decimali

1.2.1 Somme e sottrazioni

Le operazioni di somma e sottrazioni si effettuano in modo normale ma al posto di avere il riporto a 10 si ha a 16 o a 2.
Esempi:
$1011 + 0011 = 1110$
$3A09 + 1B17 = 5520$

Di solito i sistemi utilizzano un numeri di bit fisso e se un numero eccede il numero di bit massimo viene chiamato **overflow** e i bit in eccesso vengono scartati.
Esempio:
$1001 + 1100 = 10101 = 0101$ sbagliato

1.2.2 Moltiplicazioni

Per eseguire una moltiplicazione si moltiplica il primo numero per ogni bit del secondo spostandosi a sinistra di una posizione ogni volta.
Esempio:

$$\begin{array}{rl} 0101 & \times \\ 0111 & = \\ \hline 0101 & + \\ 01010 & + \\ 010100 & + \\ 000000 & = \\ \hline 100011 & \end{array}$$

1.3 Numeri binari con il segno

Per scrivere un numero con il segno in binario si utilizza il bit più a sinistra per identificare il segno:

- 0 significa positivo
- 1 significa negativo

Esempio:
$1110 = -6$
$0110 = 6$

Con N cifre in binario possiamo scrivere 2^{N-1} valori con range $[-(2^{N-1}) - 1, 2^{N-1} - 1]$

1.3.1 Complemento a due

Per poter eseguire facilmente addizioni i numeri binari con segno si scrivono utilizzando un altro metodo: il **complemento a due**.
Il primo bit (a sinistra) viene considerato negativo mentre gli altri positivi.
Esempi:
$1001 = -1 \cdot 2^3 + 1 \cdot 2^0 = -7$
$01101 = 2^3 + 2^2 + 2^0 = 8 + 4 + 1 = 13$

Per invertire (passare da positivo a negativo o viceversa) un numero in complemento a due bisogna invertire gli 0 con gli 1 e poi sommare 1.
Esempio:
$27 = 011011$
$-27 = 100100 + 1 = 100101$

Se vengono sommati due numeri con lo stesso segno potrebbe succedere che il risultato esca dal range a causa di un'overflow e bisogna aggiungere un bit.
Esempio:
$0110 + 01010 = 10001 \implies$ sbagliato perché dalla somma di due numeri positivi è uscito un numero negativo, quindi va aggiunto un bit uguale a 0 all'inizio $\implies 010001$

1.4 Estendere il numero di bit

1.4.1 Numeri senza segno

Se durante un'operazione (somma o moltiplicazione) il risultato eccede il numero di bit massimo e si causa un overflow bisogna estendere il numero di bit. Per farlo basta aggiungere tutti 0 davanti agli operandi.
Esempio:
$1010 + 1000 = 10010 \implies$ 1 non viene considerato quindi aggiungiamo uno 0 davanti ad entrambi gli operandi:
$01010 + 01000 = 10010$

1.4.2 Complemento a due

Per estendere il numero di bit di un numero in complemento a due si aggiunge a sinistra la prima cifra significativa ripetuta.
Esempi:
$1010 = 111010$
$0111 = 000111$

1.4.3 Shiftare i numeri binari

Se moltiplichiamo o dividiamo un numero binario per una potenza di 2 possiamo spostare le cifre a destra o sinistra di quanti posti quanto l'esponente di 2 nella potenza.
Esempi:
$1 \cdot 2^2 = 100$
$1000 \cdot 2^2 = 100000$
$1010/2^3 = 1$ (il resto non viene considerato visto che stiamo lavorando con numeri interi)
$10000/2 = 11000$ (nei numeri negativi quando si divide si aggiungono degli 1 all'inizio)

1.5 Numeri binari con la virgola

I numeri binari con la virgola si possono scrivere in diversi modi. Nel più classico semplicemente continuiamo a sommare potenze di 2 anche per la parte decimale, solamente con esponente negativo.
Esempi:
$0.5_{10} = 2^{-1} = 0.1_2$
$3.75_{10} = 3 + 2^{-1} + 2^{-2} = 11.11_2$

Un altro metodo è il **fixed point** in cui si decide preventivamente dove sia la virgola e semplicemente si scrive il numero in modo normale.
Esempio:
01101100(virgola al 4 posto) = 0110.1100 = 6.75

1.6 Floating point

Numeri binari in floating point

Un numero binario secondo la notazione **floating point** si scrive in modo simile alla notazione scientifica, cioè:

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

In cui:

- M = mantissa, numero con una sola cifra prima della virgola
- B = base
- E = esponente

La scrittura standard è a 32 bit di cui 1 per il segno, 8 per l'esponente e 27 per la mantissa. All'esponente bisogna aggiungere 127 (quindi per scrivere 5 dovremo scrivere 132) e nella mantissa non dobbiamo considerare il numero prima della virgola perché è sempre 1. Per convertire un numero binario in questa notazione bisogna:

1. Convertire il numero in binario senza segno
2. Scrivere il numero in notazione scientifica
3. Completare i 32 bit in modo opportuno

Vista la lunghezza di questi numeri, vengono solitamente scritti in esadecimale.

Esempio:
$228 = 11100100$

$$11100100 = 1.11001 \cdot 2^7 = \overbrace{0}^{\text{segno}}\ \overbrace{10000110}^{\text{esponente+127}}\ \overbrace{11001000000000000000000}^{\text{mantissa}} = 0x43640000$$

Nella notazione floating point esistono dei numeri speciali che si scrivono con dei bit specifici:

Numero	Segno	Esponente	Mantissa
0	indifferente	00000000	00...00
∞	0	11111111	00...00
$-\infty$	1	11111111	00...00

Oltre questi esistono una serie di numeri chiamati **numeri denormalizzati** che hanno esponente uguale a 0 e mantissa diversa da 0 e serbono per scrivere numeri con esponente minore del minimo possibile. In questi numeri si considera la cifra prima della virgola come 0 e non 1. Con questa estensione in floating point abbiamo:

- Numero massimo: 2^{126}
- Numero minimo normalizzato: 2^{-126}
- Numero minimo denormalizzato: 2^{-149}

Del floating point esistono altre due versioni con diversi bit:

- Half-precision: 16 bit, 1 per il segno, 5 per l'esponente e 10 per la mantissa. L'esponente va inserito sommato di 15.
- Half-precision: 32 bit, 1 per il segno, 11 per l'esponente e 52 per la mantissa. L'esponente va inserito sommato di 1023.

1.6.1 Operazioni con floating point

Per eseguire la **somma** tra due numeri in floating point:

1. Convertirli in binario e scriverli in notazione scientifica
2. Cambiare l'esponente minore per renderli uguali
3. Sommare le mantisse
4. Riscrivere il risultato in notazione scientifica
5. Arrotondare se non bastano i bit
6. Riscrivere nel formato floating point

Esempio:
$0x3FC00000 + 0x40500000$
$0x3FC00000 = 001111111100\ldots00 = 1.1. \cdot 2^0 = 0.11 \cdot 2^1$
$0x40500000 = 01000000010100\ldots00 = 1.101 \cdot 2^1$
$0.11 \cdot 2^1 + 1.101 \cdot 2^1 = 10.011 \cdot 2^1 = 1.0011 \cdot 2^2$
$1.0011 \cdot 2^2 = 0100000010011000\ldots00 = 0x40980000$

Per eseguire la **moltiplicazione** tra due numeri in floating point:

1. Convertirli in binario e scriverli in notazione scientifica
2. Sommare gli esponenti
3. Moltiplicare le mantisse
4. Riscrivere il risultato in notazione scientifica
5. Arrotondare se non bastano i bit
6. Riscrivere nel formato floating point

Esempio:
$(1.1 \cdot 2^{10}) \cdot (1.0110 \cdot 2^{11})$
$2^{10} + 2^{11} = 2^{21}$
$1.0110 \cdot 1.1 = 10.0001$
$10.001 \cdot 2^{21} = 1.0001 \cdot 2^{22} = 01001010100001000\ldots00 = 0x4A840000$

2

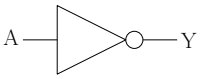
Porte logiche

Le **porte logiche** sono delle funzioni che hanno un output e possono avere:

- 1 input: not, buffer
- 2 o più input: and, or, xor ...

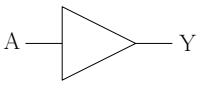
2.1 Tipi di porte logiche

Ci sono molti tipi diversi di porte logiche:
NOT: $Y = \overline{A}$



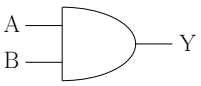
A	Y
0	1
1	0

BUFFER: $Y = A$



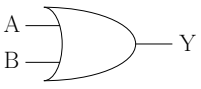
A	Y
0	0
1	1

AND: $Y = A \cdot B$



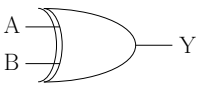
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR: $Y = A + B$



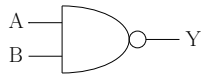
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR: $Y = A \oplus B$



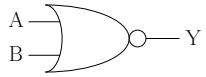
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

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



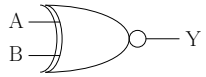
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

NOR: $Y = \overline{A + B}$



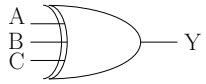
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

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



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

XOR3: $Y = A \oplus B \oplus C$



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

2.2 Circuiti logici

Componenti di un circuito logico

Un **circuito logico** è composto da:

- nodi: input, output e nodi interni
- circuiti elementari

I circuiti sono di due tipi:

1. **Combinatori**: non hanno memoria e gli output sono definiti solo dagli input attuali
2. **Sequanziali:** hanno memoria e gli output sono definiti dagli input attuali e precedenti

I circuiti combinatori hanno 3 regole principali:

1. Tutti i circuiti elementari sono combinatori
2. Ogni nodo è un input o si collega ad esattamente un output
3. Non ci sono cicli

2.3 Equazioni booleane

Nelle equazioni booleane utilizzeremo delle definizioni precise:

- **Complemento**: opposto di una variabile ($\overline{A}$)
- **Literal**: una variabile o un suo complemento (A o $\overline{A}$)
- **Implicante**: prodotto di literal (AB o $B\overline{C}$)
- **Minterm**: prodotto che utilizza tutte le variabili di input (ABC o $\overline{A}B\overline{C}$)
- **Maxterm**: somma che utilizza tutte le variabili di input ($A+B+C$ o $\overline{A}+B+\overline{C}$)

2.3.1 Somma di prodotti (SOP)

Tutte le equazioni possono essere scritte come somma di prodotti.
Se abbiamo la tabella ci basta sommare i minterm che danno come risultato 1. Il risultato può anche essere scritto come sommatoria specificando l'indice dei minterm sommati.
Esempio:

A	B	Y	Minterm	Indice del minterm
0	0	0	$\overline{A}\overline{B}$	0
0	1	1	$\overline{A}B$	1
1	0	0	$A\overline{B}$	2
1	1	1	AB	3

In questo caso: $Y = \overline{A}B + AB = B$
Possiamo anche scrivere: $Y = \sum(1,3)$

2.3.2 Prodotto di somme (POS)

Tutte le equazioni possono essere scritte come prodotto di somme.
Se abbiamo la tabella ci basta sommare i maxterm che danno come risultato 0. Il risultato può anche essere scritto come produttoria specificando l'indice dei minterm sommati.
Esempio:

A	B	Y	Maxterm	Indice del maxterm
0	0	0	$A+B$	0
0	1	1	$A+\overline{B}$	1
1	0	0	$\overline{A}+B$	2
1	1	1	$\overline{A}+\overline{B}$	3

In questo caso: $Y = (A+B)+(\overline{A}+B) = B$
Possiamo anche scrivere: $Y = \prod(0,2)$

2.4 Completezza delle porte logiche

Le porte logiche **NAND** e **NOR** sono dette **funzionalmente complete** perché grazie a queste possono essere create tutte le altre porte logiche.

3

Algebra booleana

Dualità degli assiomi

Ogni assioma e teorema dell'algebra booleana può essere scritto sia con gli AND ($\cdot$) che con gli OR ($+$). Per farlo bisogna invertire i $+$ con i $\cdot$ e gli 0 con gli 1.
La versione alternativa dell'assioma o teorema viene chiamato **duale**.

3.1 Assiomi e Teoremi

L'agebra booleana ha diversi assiomi:

Assioma	Duale
$B = 0 \iff B \neq 1$	$B = 1 \iff B \neq 0$
$\overline{0} = 1$	$\overline{1} = 0$
$0 \cdot 0 = 0$	$1 + 1 + = 1$
$1 \cdot 1 = 1$	$0 + 0 = 0$
$0 \cdot 1 = 1 \cdot 0 = 0$	$1 + 0 = 0 + 1 = 1$

Ci sono anche diversi teoremi:

Assioma	Duale
$B \cdot 1 = B$	$B + 0 = B$
$B \cdot 0 = 0$	$B + 1 = 1$
$B \cdot B = B$	$B + B = B$
$\overline{\overline{B}} = B$	
$B \cdot \overline{B} = 0$	$B + \overline{B} = 1$
$B \cdot C = C \cdot B$	$B + C = C + B$
$B \cdot C \cdot D = B(C \cdot D)$	$B + C + D = B + (C + D)$
$B(C + D) = (B \cdot C) + (B \cdot D)$	$B + (C \cdot D) = (B + C) \cdot (B + D)$
$B(B + C) = B$	$B + (B \cdot C) = B$
$(B \cdot \overline{C}) + (B \cdot C) = B$	$(B + \overline{C}) \cdot (B + C) = B$
$B(1 + C) = B$	$B + (0 \cdot C) = B$

3.2 Semplificare un'equazione

Presa un'equazione booleana, semplificarla significa ridurla al numero minimo di implicanti tramite teoremi o assiomi (detta anche **minimizzazione**). **Esempi:**
$\overline{A}B + A = B + A$
$AB + A = A$
$\overline{A}B + AB = B$

Teorema di De Morgan

Il **teorema di De Morgan** dice che se abbiamo un'equazione booleana con una somma o una moltiplicazione negata ($\overline{AB}$ oppure $\overline{A+B}$), possiamo invertire l'operazione e negare i literal. Cioè:

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

Esempio:
$Y = \left(A + \overline{BD}\right) \cdot \overline{C} = \overline{A + \overline{BD}} + \overline{\overline{C}} = \overline{A}BD + C$

4

Circuiti logici

4.1 Regole dei circuiti

I circuiti logici, quando vengono disegnati, hanno diverse regole:

- Gli input vengono scritti a sinistra o in alto
- Gli output vengono scritti a destra o in basso
- Qualsiasi posta va da sinistra a destra

Inoltre vengono definite delle regole quando di intrecciano i fili:

- Se i fili formano una T sono collegati
- Se i fili formano una X con un punto sono collegati
- Se i fili formano una X senza punto non sono collegati

Nei primi due casi i fili sono collegati e nel terzo no

4.1.1 Circuiti con più output

Possono esserci circuiti che hanno più di un output, un esempio è il circuito a priorità, che ha questa tabella:

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

Molti valori in questa tabella non vengono considerati nel calcolare l'output quindi possiamo riscrivere la tabella in modo semplificato:

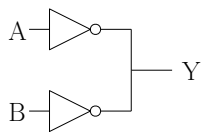
A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	**X**	0	0	1	0
0	1	**X**	**X**	0	1	0	0
1	**X**	**X**	**X**	1	0	0	0

X vuol dire **don't care**, cioè che il valore non cambia il risultato

4.1.2 Contention

La **contention** è una situazione che viene causata quando in un circuito si incontrano due valori diversi. Si indica con X (da non confondere con il don't care) e solitamente implica un problema.

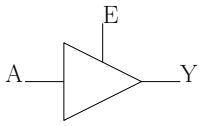
Esempio:



A	B	Y
0	0	1
0	1	**X**
1	0	**X**
1	1	0

4.1.3 Floating

Il **floating** è una situazione che viene causata quando un filo è disattivato o non c'è corrente, si indica con Z.
Un esempio è il **tristate buffer**, un buffer che è attivo solo quando un parametro E è uguale a 1:



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

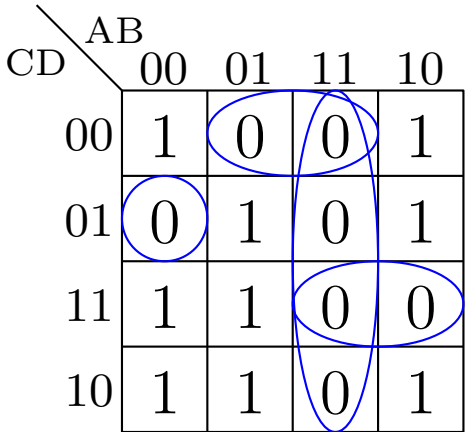
4.2 Mappe di Karnaugh (K-maps)

Le espressioni booleane possono essere semplificate in modo grafico utilizzando le **mappe di Karnaugh**. Questa mappa conterrà in ogni riquadro una riga della tabella della verità. Una volta che abbiamo scritto gli 0 e 1 nella tabella possiamo semplificarli in due modi:

- Utilizzando i minterm e gli 1 secondo le regole:
  1. Bisogna cerchiare ogni 1 almeno una volta
  2. Ogni cerchio deve contenere un numero di 1 pari ad una potenza di 2 (1,2 o 4)
  3. Ogni cerchio deve essere preso il più grande possibile
  4. Un cerchio può contenere don't care se servono a creare un cerchio più grande
  5. Alla fine ogni cerchio rappresenterà un prodotto tra le variabili cerchiate escludendo quelle che all'interno del cerchio appaiono sia negate che normali
- Utilizzando i miaxterm e gli 0 secondo le regole:
  1. Bisogna cerchiare ogni 0 almeno una volta
  2. Ogni cerchio deve contenere un numero di 0 pari ad una potenza di 2 (1,2 o 4)
  3. Ogni cerchio deve essere preso il più grande possibile
  4. Un cerchio può contenere don't care se servono a creare un cerchio più grande
  5. Alla fine ogni cerchio rappresenterà una somma tra le variabili cerchiate escludendo quelle che all'interno del cerchio appaiono sia negate che normali

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



Si scrive 11 vicino a 01 perchè possono essere raggruppati solo riquadri che differiscono per solo una variabile. In questo caso avremo:

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

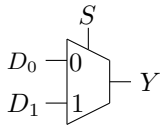
5

Blocchi combinatori

I blocchi combinatori sono un insieme di porte logiche che eseguono delle determinate operazioni specifiche, i due tipi più importanti sono **multiplexer (mux)** e **decoder**.

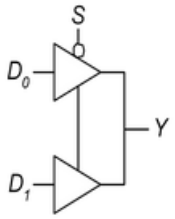
5.1 Multiplexer (Mux)

I multiplexer selezionano uno dei possibili N input (numero potenza di 2) tramite un numeri di selezionatori pari a $\log_2 N$. Anche il mux, come il NOR e il NAND permette di creare tutte le altre porte logiche. Viene disegnato:



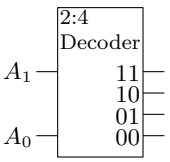
$$Y = \overline{S}D_0 + SD_1$$

Se lo volessimo disegnare con le porte logiche classiche sarebbe:

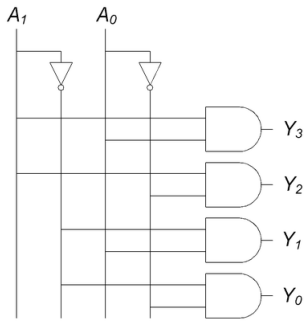


5.2 Decoder

I decoder hanno N input (numero potenza di 2) e un output per ogni combinazione possibile degli input, per un totale di 2^N output. Segue una politica **one-hot**, cioè che solo un output può essere attivo (uguale a 1) alla volta. Viene disegnato:



Se lo volessimo disegnare con le porte logiche classiche sarebbe:



I decoder possono essere usati per creare funzioni logiche partendo dai minterm.

5.3 Teorema di Shannon

Teorema di Shannon

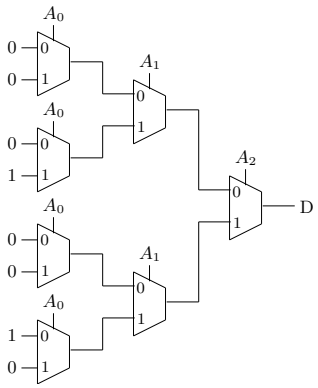
Data una funzione f con n variabili, questa può essere semplificata nella somma di due funzioni ad $n-1$ variabili moltiplicate per i due possibili valori della variabile:

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

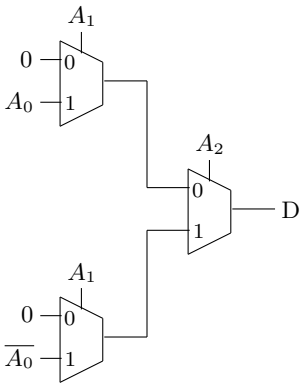
Questo teorema si può applicare ai mux per scrivere qualsiasi funzione.

Esempio:

A_2	A_1	A_0	D
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

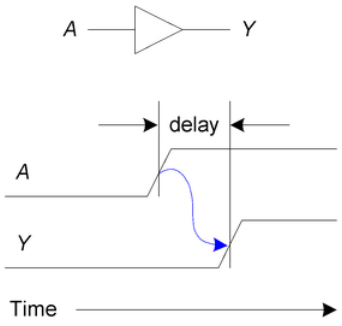


Semplificandolo otteniamo:



5.4 Delay

Il **delay** è il tempo da quando cambia un input fino a quando cambia l'output corrispondente. Viene calcolato da quando l'input è a metà del cambiamento fino a quando l'output è a metà del cambiamento.

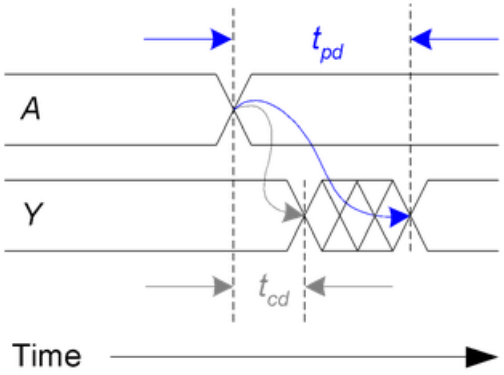


Il delay può essere causato da:

- capactà e resistenze del circuito
- limitazioni della velocità della luce

Ci sono due tipi di tempo che possiamo calcolare:

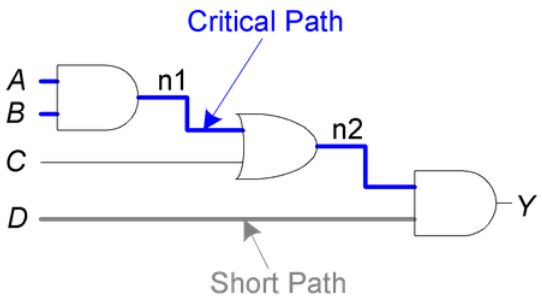
- **Propagation delay** (t_{pd}): tempo massimo per effettuare il cambiamento
- **Contamination delay** (t_{cd}): tempo minimo per effettuare il cambiamento



Il propagation delay si calcola sommando tutti i tempi massimi delle porte logiche del percorso più lungo mentre il delay minimo sommando tutti i tempi minimi delle porte logiche del percorso più corto.
I motivi per cui t_{pd} e t_{cd} sono diversi sono:

- diversi tempi di innalzamento o discesa
- multipli input e output, alcuni più veloci di altri
- rallentamenti dovuti alla tempeatura

Esempio:



In questo caso:

- $t_{pd} = 2 \cdot t_{pd\_AND} + t_{pd\_OR}$
- $t_{cd} = t_{cd\_AND}$

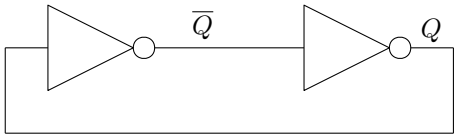
6

Circuiti sequenziali

I **circuiti sequenziali** sono circuiti che posseggono una memoria per immagazzinare i precedenti valori, quindi l'output dipende anche dai valori che ha in memoria. Rappresenta l'insieme dei valori passati racchiudendoli nello stato del sistema, cioè le informazioni su un circuito necessarie per spiegare i comportamenti futuri. Lo stato del sistema viene memorizzato in blocchetti chiamati **Latches** e **Flip-Flops**, che immagazzinano un bit dello stato.

6.1 Circuito bistabile

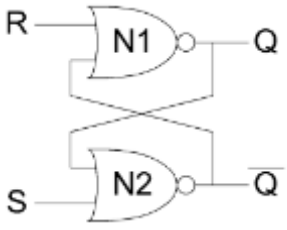
Il **circuito bistabile** è un tipo di circuito sequenziale con 2 output ma nessun input. Può immagazzinare un bit di stato.



Se $Q = 0 \implies \overline{Q} = 1$
Se $Q = 1 \implies \overline{Q} = 0$

6.2 SR (Set/Reset) Latch

Il **SR Latch** è un circuito sequenziale che mantiene i valori di Q e $\overline{Q}$ uguali finchè non vengono attivati degli input S o R che, rispettivamente, settano o resettano il valore di Q (e di conseguenza anche di $\overline{Q}$).



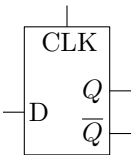
Questo circuito ha 4 casi possibili:

S	R	Q	$\overline{Q}$
0	0	Q precedente	$\overline{Q}$ precedente
0	1	0	1
1	0	1	0
1	1	0	0

L'ultimo caso in verità non è possibile perchè $Q \neq \overline{Q}$ sempre.

6.3 D Latch

Il **D Latch** è un circuito sequenziale che ha 2 input, CLK che decide quando cambia l'output e D che è l'effettivo input. In questo modo si risolve il problema dell'SR latch e si può settare Q a 0 o 1 a piacimento.
Se CLK= 1 $\implies Q = D$
Se CLK= 0 $\implies Q$ rimane uguale

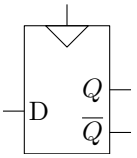


Questo circuito ha 4 casi possibili:

CLK	D	Q	$\overline{Q}$
0	X	Q precedente	$\overline{Q}$ precedente
1	0	0	1
1	1	1	0

6.4 Flip-Flop

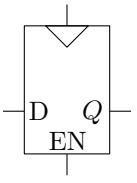
Il **flip-flop** memorizza il valore di D quando CLK passa da 0 a 1, ma non quando passa da 1 a 0.



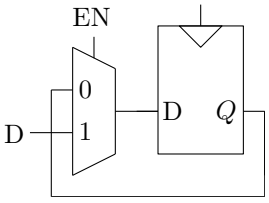
Ci sono diversi tipi di flip-flop.

6.4.1 Flip-Flop attivabili

I flip-flop attivabili sono una tipologia di flip-flop in cui si aggiunge un'altro input EN che permette di spegnere il flip-flop quando il valore di EN è uguale a 0.

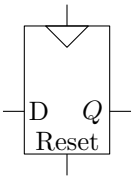


Il circuito interno è:



6.4.2 Flip-Flop resettabili

I flip-flop attivabili sono una tipologia di flip-flop in cui si aggiunge un'altro input R che permette di far diventare $Q = 0$ quando $R = 1$.

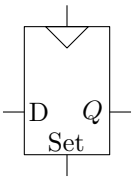


Ce ne sono due tipi:

- Sincrono: Q diventa 0 quando CLK passa da 0 a 1
- Asincrono: Q diventa 0 appena R diventa 1

6.4.3 Flip-Flop settabili

I flip-flop attivabili sono una tipologia di flip-flop in cui si aggiunge un'altro input S che permette di far diventare $Q = 1$ quando $S = 1$.



Ce ne sono due tipi:

- Sincrono: Q diventa 1 quando CLK passa da 0 a 1
- Asincrono: Q diventa 1 appena S diventa 1

6.5 Logica sequenziale sincrona

Nella logica sequenziale sincrona i cicli vengono interrotti da dei registri che contengono lo stato del sistema, lo stato cambia qundo il clock passa da 0 a 1. Ci sono delle regole:

- Ogni circuito è un registro oppure un circuito combinatorio
- Almeno un elemento è un registro
- Tutti i registri hanno lo stesso clock
- Ogni circuito ha almeno un registro

7

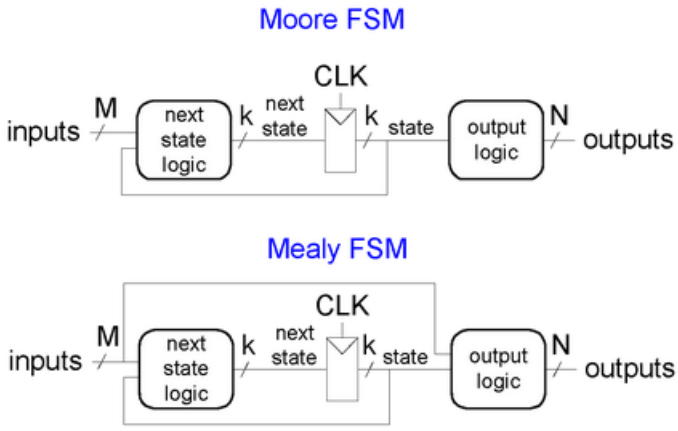
Finite State Machine (FSM)

Una **macchina a stati finiti** è composta da:

- Registri: memorizzano lo stato corrente e cambiano lo stato salvato memorizzato quando il clock si attiva (passa da 0 a 1)
- Logica combinatoria: calcola il prossimo stato e gli output

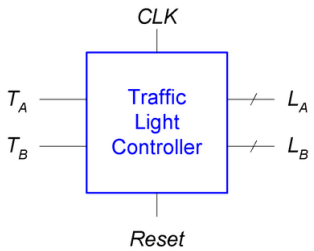
Nelle FSM lo stato successivo è definito da quello precedente.
Le FSM sono di due tipi:

- Moore FSM: l'output dipende solo dallo stato corrente
- Mealy FSM: l'output dipende dallo stato corrente e dagli input



Esempio:
Controllo del traffico con:

- Sensori: $T_a, T_b = True$ quando c'è traffico
- Luci: L_a, L_b



7.1 Diagramma di trascrizione

Il **diagramma di trascrizione** è formato da stati, rappresentati con cerchi, e transizioni, rappresentate con archi. Dal diagramma di trascrizione si può scrivere la tabella di trascrizione, che dagli input e dallo stato corrente definisce il prossimo stato. Dopo si codificano gli stati con dei bit e si riscrive la tabella di trascrizione con gli stati codificati e infine si codificano gli output e si scrive la tabella degli output in base allo stato corrente.
Esempio:

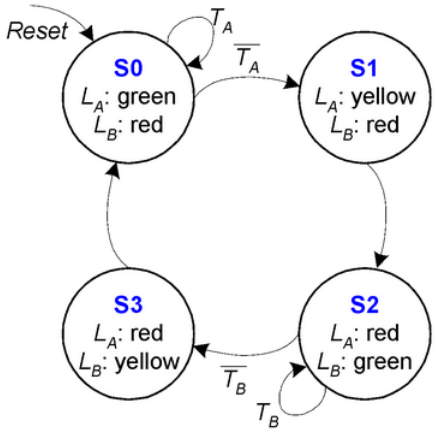


Tabella di trascrizione:

Stato attuale	Input		Prossimo stato
S	T_a	T_b	S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

Tabella codificata:

Stato	Codifica
S0	00
S1	01
S2	10
S3	11

Stato attuale		Input		Prossimo stato	
S_1	S_0	T_a	T_b	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

Tabella degli output:

Output	Codifica
green	00
yellow	01
red	10

Stato attuale		Output			
S_1	S_0	L_{a_1}	L_{a_0}	L_{b_1}	b_0
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

7.1.1 Codifica degli stati di una FSM

La codifica degli stati in una FSM può essere:

- Binaria: in cui ogni combinazione di bit rappresenta uno stato (es. 4 stati: 00, 01, 10, 11)
- One-hot: in cui solo un bit è uguale ad uno alla volta e serve un bit per ogni stato (es. 4 stati: 0001, 0010, 0100, 1000)

Stati irraggiungibili

Gli **stati irraggiungibili** in un FSM sono degli stati che non possono essere raggiunti e di solito si usano come stati di inizio e reset. Sono presenti in FSM con numero di stati non potenza di 2.

7.2 Differenza tra Moore e Mealy

La differenza principale tra le due FSM è che la Moore FSM ha bisogno di più stati di una Mealy FSM. Se l'output di una Mealy FSM fosse rallentata da un flip-flop, avrebbe gli output sincronizzati con una Moore FSM e questo evita output glitchati (sbalzi di valore veloci)
Esempio:
Una macchina che legge una stringa e dà in output 1 se legge una sequenza 01.
Moore:

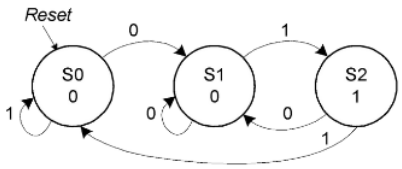


Tabella di trascrizione:

Stato	Codifica
S0	00
S1	01
S2	10

Stato attuale		Input	Prossimo stato	
S_1	S_0	I	S_1'	S_0'
0	0	0	0	1
0	0	1	0	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0

$S_1' = S_0 I$
$S_0' = \overline{I}$
Tabella degli output:

Stato attuale		Output
S_1	S_0	Y
0	0	0
0	1	0
1	0	0

$Y = S_1$

Mealy:

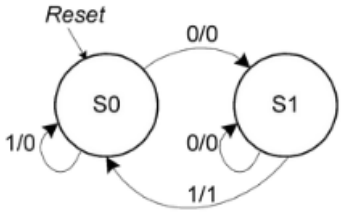


Tabella di trascrizione e output:

Stato	Codifica
S0	0
S1	1

Stato attuale S_0	Input I	Prossimo stato S_0'	Output Y
0	0	1	0
0	1	0	0
1	0	1	0
1	1	0	1

$Y = S_0 A$

7.2.1 Passare da Moore a Mealy

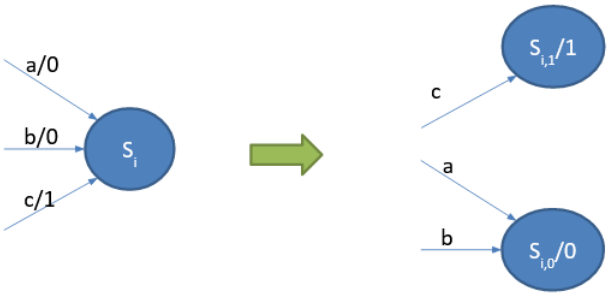
Per passare da una FSM Moore a una FSM Mealy bisogna:

- Spostare gli output dagli stati agli archi
- Ogni arco entrante in uno stato deve avere l'output dello stato
- Se due stati hanno gli stessi archi uscenti possono essere uniti in un solo stato

7.2.2 Passare da Mealy a Moore

Per passare da una FSM Meay a una FSM Moore bisogna:

- Spostare gli output dagli archi agli stati
- Se uno stato ha due archi entranti con output diversi bisogna dividerlo in due stati



7.3 Design di una FSM

Per fare il design di ua FSM bisogna seguire dei passaggi:

1. Identificare input e output
2. Disegnare il diagramma di transizione
3. Scrivere la tabella di trascrizione
4. Codificare gli stati
5. Riscrivere la tabella di trascrizione:
   - 5.1 Moore: Riscrivere la tabella di trascrizione con gli stati codificati e la tabella degli output
   - 5.2 Mealy: Riscrivere la tabella di trascrizione e degli output con gli stati codificati
6. Scrivere l'equazione della **next state logic** e degli output
7. Disegnare il circuito

Invece per derivare una FSM da un circuito bidogna:

1. Esaminare circuiti, input, output e bit di stato
2. Scrivere le equazioni booleane degli stati e degli output
3. Creare la tabella della next state logic
4. Ridurre la tabella eliminando gli stati irraggiungibili
5. Assegnare a ogni combinazione di bit uno stato
6. Riscrivere la tabella con i nomi degli stati
7. Disegnare il diagramma di stato
8. Descrivere a parole la FSM

8

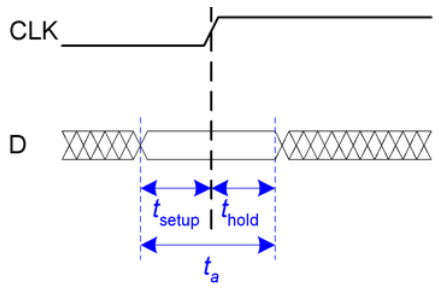
Timing

I flip-flop salvano il valore ad ogni ciclo di clock, ma per poterlo fare bisogna che il valore sia stabile nel momento in quel momento, sennò si entra in un caso di **metastabilità**, cioè uno stato in cui il valore è in uno stato intermedio tra 0 e 1 da cui prima o poi si uscirà tornando a 0 o 1, ma senza sapere quale.

8.1 Tempistiche di input ed output

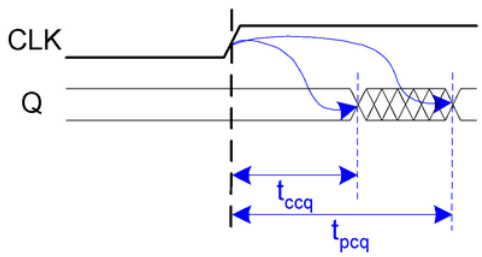
I valori degli input e degli output per poter essere letti o scritti correttamente hanno bisogno di tempo i cui il loro valore deve essere stabile.
Input:

- Periodo di setup (t_{setup}): periodo prima del clock in cui il valore deve essere stabile
- Periodo di hold (t_{hold}): periodo dopo il clock in cui il valore deve essere stabile
- Periodo di apertura (t_a): il tempo totale, prima e dopo il clock, in cui il valore deve essere stabile



Output:

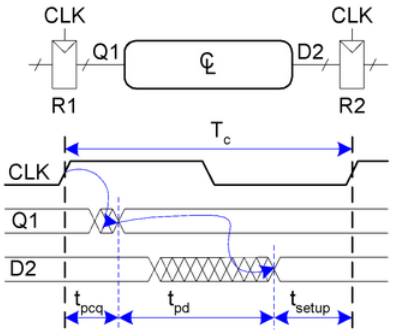
- Delay di propagazione (t_{pcq}): tempo massimo in cui siamo sicuri che l'output diventi stabile
- Delay di contaminazione (t_{ccq}): tempo minimo in cui l'output potrebbe essere stabile



8.2 Disciplina dinamica

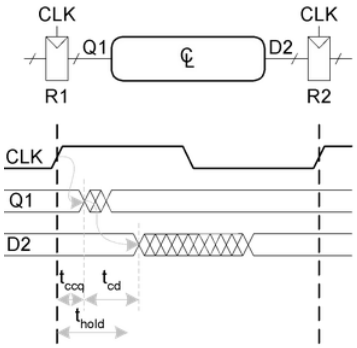
Il delay tra i registri ha anch'esso un minimo e un massimo che dipende dal circuito. Quindi dobbiamo calcolare il tempo di clock per far si che sia maggiore di tutti i tempi massimi sommati per assicurarci che non ci siano problemi con i valori.

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



I valori non possono neanche cambiare prima del tempo di hold, che dovrà quindi essere minore dei valori minimi.

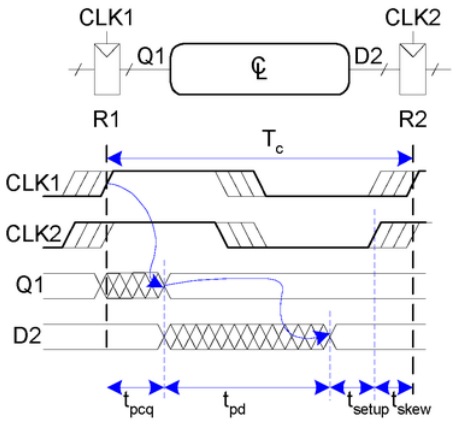
$$T_{hold} < t_{ccq} + t_{cd}$$



8.2.1 Clock stew

Il clock potrebbe non arrivare nello stesso momento in tutte le parti del circuito, ma ci potrebbe essere un tempo t_{skew} di differenza. Quando si decide il tempo di clock bisogna sempre calcolare il caso peggiore, quindi le formule precedenti diventano:

$$T_c \geq t_{pcq} + t_{pd} + t_{setup} + t_{skew}$$
$$t_{hold} + t_{skew} < t_{ccq} + t_{cd}$$



8.3 Parallelismo

Il **parallelismo** è un modo di moltiplicare le operazioni svolte al fine di migliorare la velocità di queste operazioni. Ci sono due tipi di parallelismo:

- Spaziale: un hardware duplicato esegue più operazioni alla volta
- Temporale: le operazioni vengono divise in fasi (pipeline)

Definizioni:

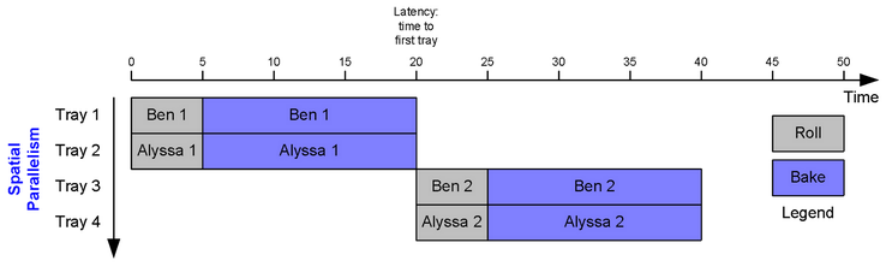
- Token: gruppo di input processati
- Latency: tempo impiegato per processare un Token
- Throughput: numero di token processati in un determinato intervallo di tempo

Esempio:
Se per fare dei biscotti servono 5 minuti per impastarli e 15 minuti per cuocerli allora senza parallelismo avremo:

- Latency: 20 min
- Throughput: 3/h

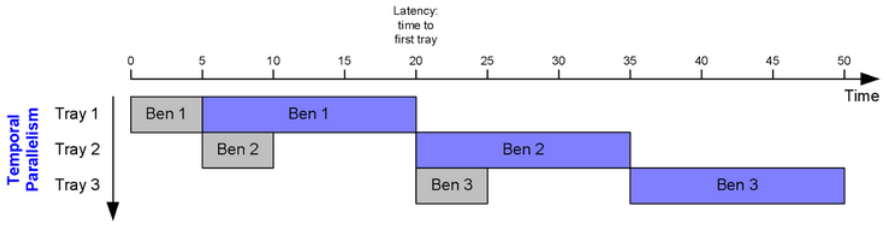
Con parallelismo spaziale (un'altra persona fa dei biscotti contemporaneamente):

- Latency: 20 min
- Throughput: 6/h



Con parallelismo temporale (mentre una teglia di biscotti cuoce ne inizia a impastare un'altra):

- Latency: 30 min
- Throughput: 4/h



9

Blocchi costruttivi

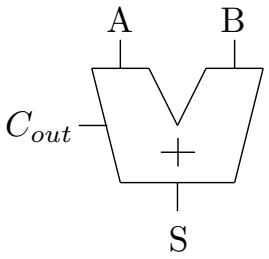
I **blocchi costruttivi** sono dei circuiti che eseguono delle particolari operazioni aritmetico-logiche.

9.1 Adder

Un **adder** è un blocco che dati due numeri in input ne esegue la somma. Ci sono due tipi di adder a 1 bit:

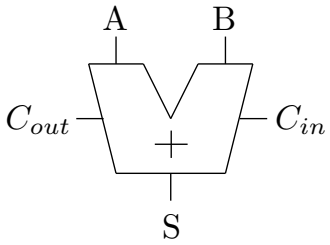
- Half Adder: Esegue la somma dei due numeri ad un bit e restituisce il risultato più un eventuale riporto C_{out}
- Full Adder: Esegue la somma dei due numeri più un riporto in entrata C_{in} e restituisce il risultato più un eventuale riporto C_{out}

Half Adder



$S = A \oplus B$
$C_{out} = AB$

Full Adder



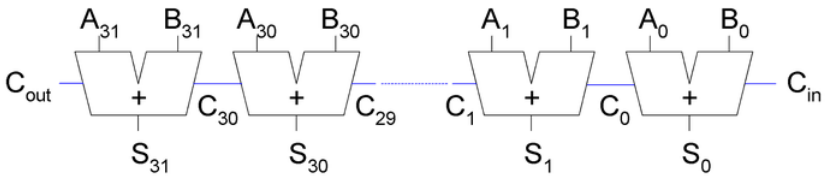
$S = A \oplus B \oplus C_{in}$
$C_{out} = AB + AC_{in} + BC_{in}$

Nel caso di adder multibit ce ne sono di due tipi:

- Ripple-carry: lento
- Carry-lookahead: veloce

9.1.1 Adder Ripple-carry

L'adder ripple-carry semplicemente unisce tra loro N full adder a 1 bit per sommare un due numeri ad N bit. Ogni singolo adder avrà in input un bit del due numeri ed il riporto dell'adder precedente e restituirà la somma e un riporto in output per l'adder successivo.

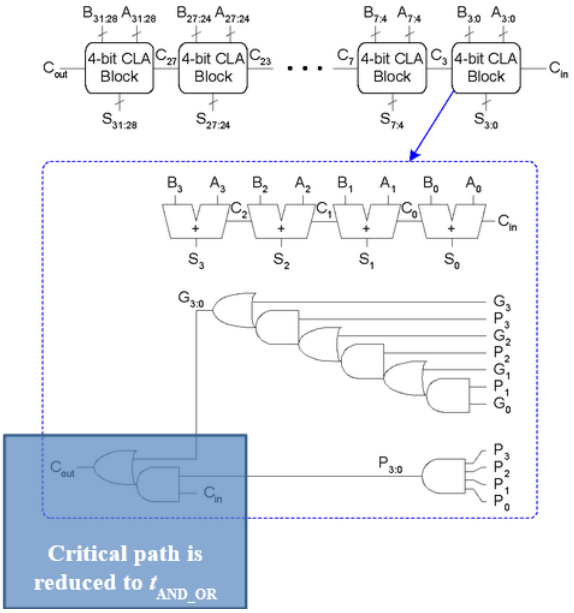


Il tempo di questo adder è:

$$t_{RC} = N \cdot t_{FA}$$

9.1.2 Adder Carry-lokahead

L'adder carry-lookahead per essere più veloce del ripple-carry calcola il riporto finale in anticipo.
$C_i = A_iB_i + (A_i + B_i)(A_{i-1}B_{i-1} + (A_{i-1} + B_{i-1})C_{i-1})$
$G_i = A_i + B_i$ = riporto generato dal bit i
$P_i = A_i + B_i$ = riporto propagato dal bit $i - 1$
$C_i = G_i + P_iC_{i-1}$
$G_{i:j} = G_i + P_iG_{i-1:j}$ = riporto generato tra i bit j e i
$P_{i:j} = P_iP_{i-1}...P_j$ = riporto propagato dal bit j fino al bit i
Per creare degli adder a molti bit si utilizzano dei blocchi più piccoli a k bit:



L'adder quindi esegue in ordine:

- Calcola G_i e P_i per ognuno degli N bit
- Calcola $G_{i:j}$ e $P_{i:j}$ per ognune dei blocchi
- Fa propagare C_in attraverso tutti i blocchi (in contemporanea alle somme)
- Calcola la somma degli ultimi k bit

Il tempo di questo adder è:

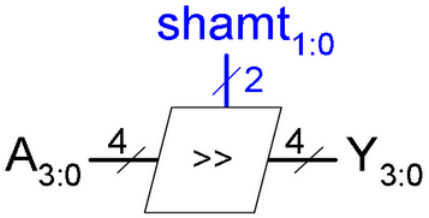
$$t_{CLA} = t_{PG} + t_{PG\_BLOCK} + (\frac{N}{k} - 1)t_{AND\_OR} + kt_{FA}$$

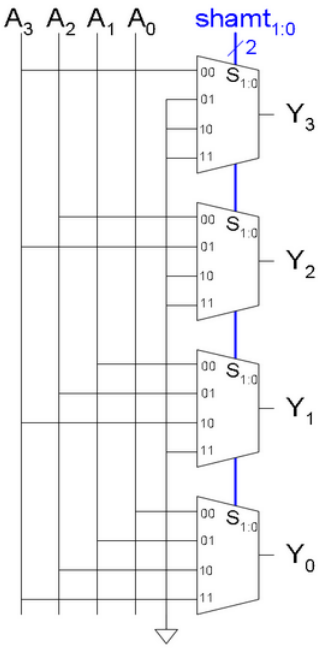
In cui t_{PG} è il tempo per calcolare tutti G_i e P_i e t_{PG_BLOCK} è il tempo per calcolare tutti i $G_{i:j}$ e $P_{i:j}$.

9.2 Shifter

Gli **shifter** sono blocchi che eseguono divisioni o moltiplicazioni per potenze di 2.
Ci sono 3 tipi di shifter:

- Shifter logico: si utilizza per numeri senza segno e riempie gli spazi vuoti con degli 0, potrebbero esserci dei risultati sbagliati se il numero esce dal range
- Shifter aritmetico: si utilizza per numeri binari in complemento a 2 e quando divide riempie gli spazi vuoti col bit più significativo
- Rotatore: fa ruotare i bit, facendoli tornare a destra se escono da sinistra e viceversa

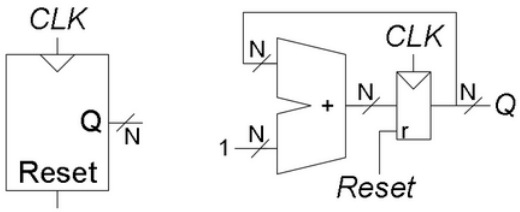




Esempi shifter logico:
11001 >> 2 = 00110
11001 << 2 = 00100 sbagliato perchè esce dal range
Esempi shifter aritmetico:
11001 >> 2 = 11110
11001 << 2 = 00100 sbagliato perchè esce dal range
Esempi rotatore:
11001 ROR2 = 01110
11001 ROL2 = 00111

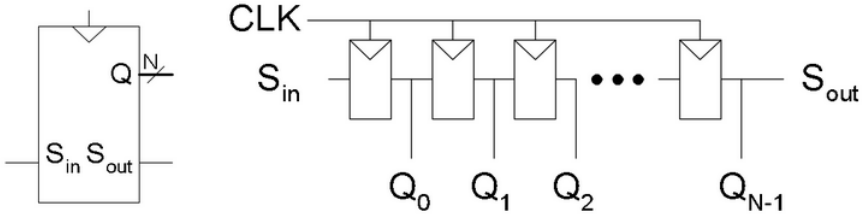
9.3 Counter

Il counter è un blocco che fa aumentare l'output di 1 ogni ciclo di clock a meno che non venga resettato tramite un input R. A livello di circuito viene creato con un adder multi-bit e una serie di flip-flop resettabili.



9.4 Shift registers

Gli shift register sono una serie di registri che ad ogni ciclo di clock shiftano il numero di un bit e aggiungono come ultimo bit un input S_{in}.

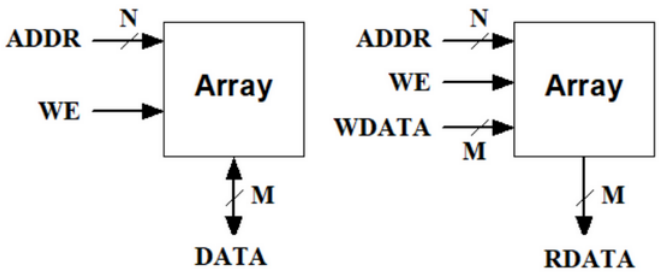


10
Memoria

10.1 Array

Gli **array** immagazzinano grandi quantità di dati. Per ogni **indirizzo** ad N bit vengono scritti M bit di dati. Sono formati da celle con $2N$ righe e M colonne.
Un array di memoria di grandezza $2N \times M$ ha:

- N bit di indirizzo in input
- M bit che possono essere bidirezionali (sia input che output) oppure 2 dati da M bit, uno in input di scrittura e uno in output di lettura
- Un segnale WE che decide se si sta leggendo o scrivendo

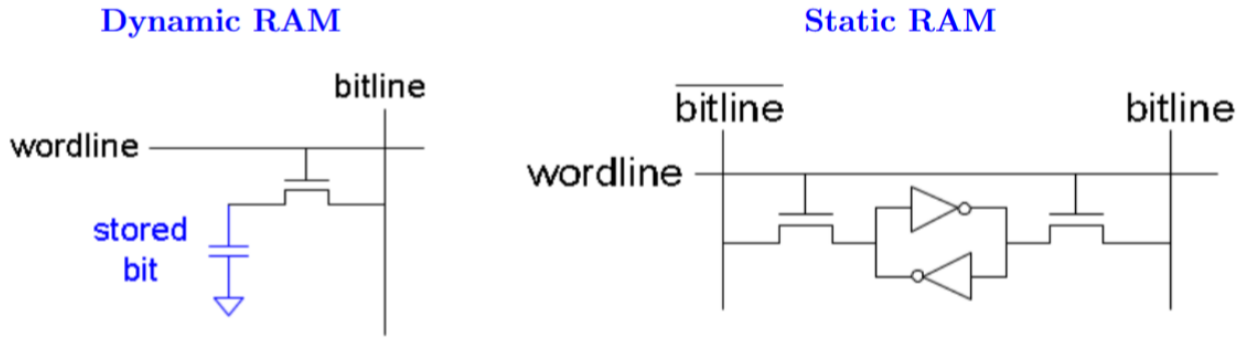


Le memorie si dividono in due tipi:

- **RAM** (Random Access Memory): volatile e divisa a sua volta in:
  - **DRAM** (Dynamic Random Access Memory)
  - **SRAM** (Static Random Access Memory)
- **ROM** (Read Only Memory): non volatile

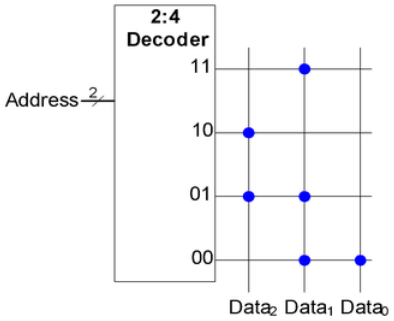
10.1.1 RAM

Nelle RAM i bit vengono storati in dei capacitor, che sono diversi in base al tipo di RAM (DRAM o SRAM). Nella DRAM leggere un valore lo distrugge, quindi bisogna subito inserirne un'altro.
Per sapere quando bisogna leggere un valore ogni riga ha una **wordline** che indica che bisogna leggere la riga e ogni colonna ha un **bitline** che è uguale al bit della cella la cui riga ha wordline uguale a 1.



10.1.2 ROM

Le ROM possono essere usate per eseguire della logica combinatoria, affiancate da un decoder. I pallini si usano per indicare in quali casi il valore deve essere 1. **Esempio:**

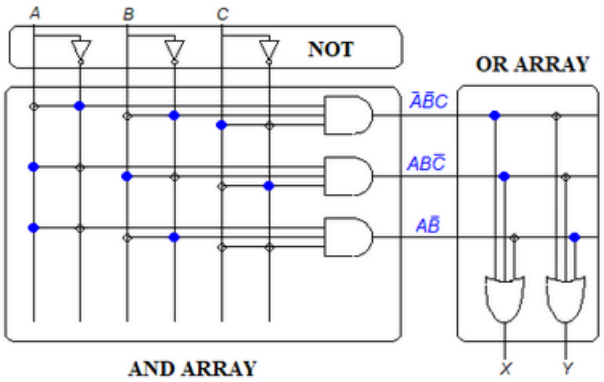


$$Data_2 = A_1 \oplus A_0$$
$$Data_1 = \overline{A_1} + A_0$$
$$Data_0 = \overline{A_1}\,\overline{A_0}$$

10.2 Programmable Logic Array (PLA)

Una PLA è composta da degli array AND seguiti da degli array OR che permettono di svolgere logica combinatoria in formato SOP. Composta da:

- M input
- N output
- 3 stadi interni:
  1. Uno stadio di inversione degli ingressi con dei NOT
  2. Una matrice di AND
  3. Una matrice di OR



10.3 Field Programmable Gate Array (FPGA)

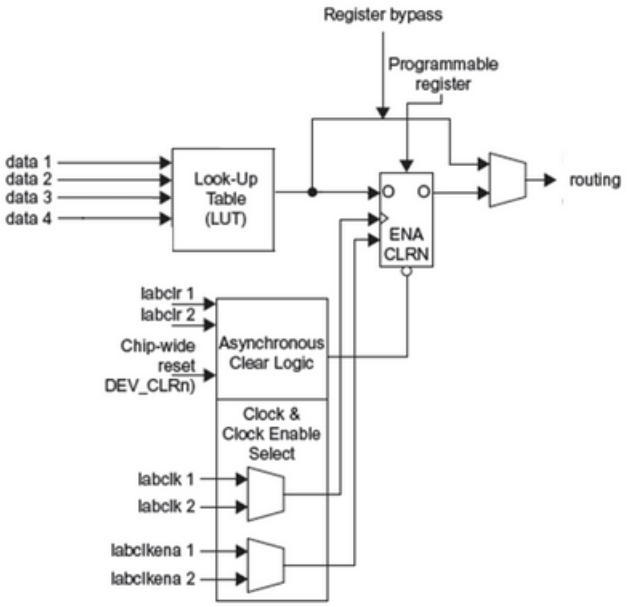
Una FPGA è composta da:

- Elementi logici
- Blocchi I/O: interfaccie con l'esterno
- Connessioni programmabili: collegano gli elementi logici con i blocchi I/O
- Possono contenere altri blocchi come RAM o multipliers

10.3.1 Elementi logici

Gli elementi logici sono composti da:

- **Lookup Table** (LUT) a 4 ingressi: esegue la logica combinatoria
- Flip-Flop: eseguono la logica sequenziale
- Multiplexer: collegano LUT e Flip-Flop



10.4 ALU

La ALU è un blocco che esegue operazioni matematico-logiche su due numeri (A e B) ad N bit (solitamente 32). L'operazione viene scelta da dei bit di controllo chiamati `ALUControl`.
In particolare l'ALU può eseguire:

- Addizioni
- Sottrazioni
- AND
- OR

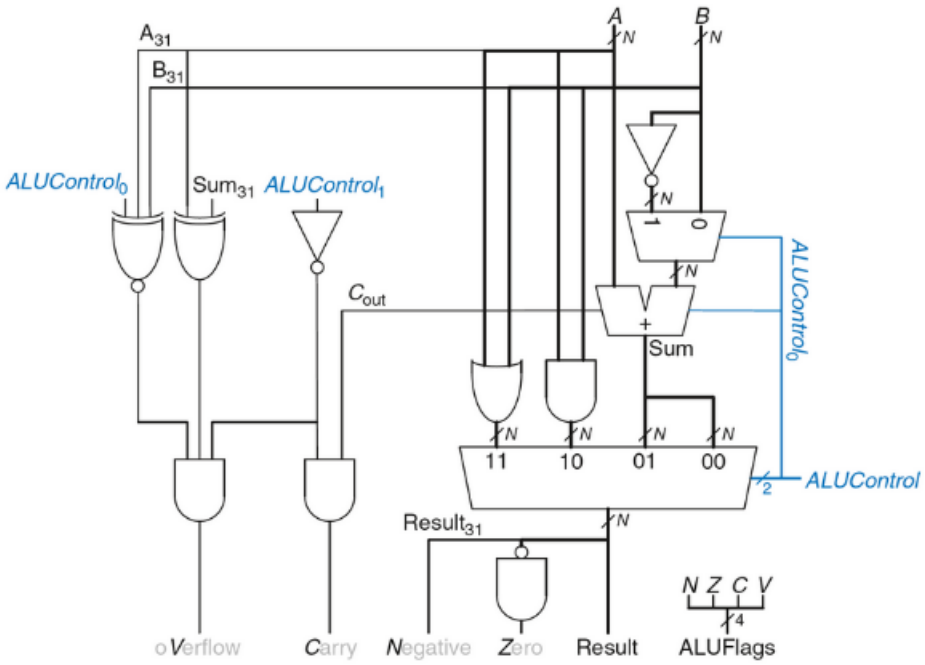
Essendo 4 operazioni serviranno 2 bit per l'`ALUControl`:

$\text{ALUControl}_{1:0}$	Operazione
00	ADD
01	SUB
10	AND
11	OR

L'ALU Restituisce un risultato Y e dei bit chiamati **flag** che indicano determinate cose, cioè:

Flag	Formula	Descrizione
N	Y_{31}	Il risultato è negativo
Z	$\overline{Y + Y}$	Il risultato è 0
C	$C_{out} \cdot \overline{\text{ALUControl}_1}$	La somma ha prodotto un riporto in uscita
V	$\overline{\text{ALUControl}_1} \cdot (\overline{A_{31} \oplus B_{31} \oplus \text{ALUControl}_0}) \cdot (A_{31} \oplus Y_{31})$	La somma ha prodotto un overflow

Il circuito completo dell'ALU è:



11
System Verilog

Il **System Verilog** è un linguaggio di programmazione usato per eseguire logica a basso livello. Come simboli utilizza:

- NOT $=\sim$
- AND = &
- OR = |
- $[x : 0]$ indica che è un array a $x + 1$ bit

11.1 Moduli

Per creare un modulo in system verilog bisogna scrivere:

```
module nome(input logic a, b, c, output logic y)
    assign y=~a & ~b & c | a & b & ~c
endmodule
```

Per richiamarlo basta scrivere:

```
nomemodulo nomeistanza(a, b, c, y)
```