materiale sotto licenza FDL – Free Documentation License - si veda la prima pagina e l'ultima sezione

appunti di

CALCOLATORI ELETTRONICI I

parte I

basati sui miei appunti di calcolatori elettronici.
Contributi:
-il libro del bucci;
-gli appunti di alessio bazzica;
-droscy (teoria dell'informazione);
-chiunque altro vuole contribuire!!

convenzioni:
//?testo?// indica del testo non chiaro, non preciso, probabilmente sbagliato, provvisorio o da correggere.
/*testo*/ indica un commento

nota: il “sorgente” è scritto con openoffice.org 1.1.3 su sistema linux, usando prevalentemente i caratteri “Nimbus Roman No9 L”, “Bitstream Vera Sans Mono”, “Nimbus Mono L”. Prima di modificare il documento conviene installarsi questi font, altrimenti il testo potrebbe risultare un po' scombinato. Questi caratteri sono scaricabili da http://www.perpassione.org/downloads/fonts.zip.

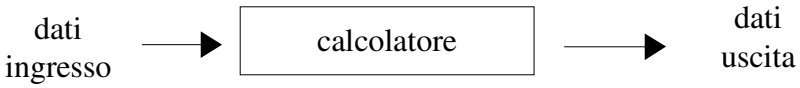
Indice

Corso di calcolatori elettronici a.a. 2004-2005
Le prime pagine introduttive sono state riprese dagli appunti di Alessio Bazzica.

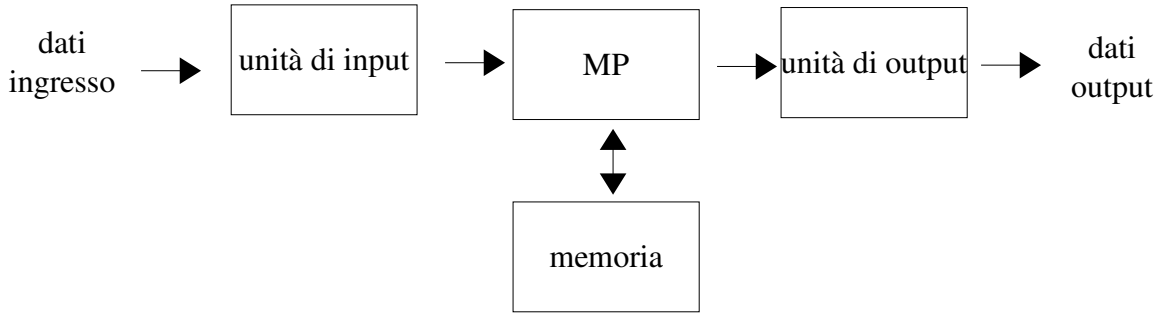
Introduzione al corso

parte	***argomenti***
I	1. introduzione ai sistemi digitali 2. rappresentazione dell'informazione 3. reti logiche (combinatorie e sequenziali)
II	4. il microprocessore (CPU) 5. la memoria 6. l'I/O 7. 8086: architettura hardware 8. 8086: programmazione

Un primo sistema a blocchi che rappresenti un calcolatore generico nel più semplice dei modi è il seguente:



questo schema può essere commentato dicendo che il calcolatore elabora i dati di ingresso restituendo un'uscita.
per avere un'idea di architettura possiamo osservare il seguente modello, modello in cui viene focalizzata l'importanza della memoria per contenere dati e istruzioni da eseguire. il modello, per questa sua ultima caratteristica, prende esempio dalla struttura di calcolatore di Von Neumann.



bisogna notare il verso delle frecce tra i vari blocchi. l'unità di input invia i dati (comprensibili dal MP) al MP (microprocessore). il MP esegue il programma presente in memoria (leggendo istruzioni) che prevede di elaborare i dati d'ingresso (che vengono quindi scritti in memoria al loro arrivo). il programma prevede di leggere dati giunti dall'unità di input per elaborarli e generarne nuovi utili all'output. Dopo l'elaborazione, sempre nel linguaggio del MP, i dati vengono portati ad un'unità di output che, per esempio, si occupa della loro visualizzazione su uno schermo.

il linguaggio compreso dalle reti logiche è il linguaggio binario, ovvero un linguaggio dove ogni cosa

può assumere due determinati valori che rappresentano due soli stati associabili ai valori 0 e 1 (alto o basso, acceso o spento).
quindi un singolo bit si può esprimere nel seguente modo: $b \in \{0,1\}$ (può assumere due valori, è possibile aggregare più bit formando una parola di bit. per esempio possiamo ottenere una parola $b_1 b_0$ utilizzando due bit. In questo caso otteniamo 4 combinazioni $b_1 b_0 \in \{00,01,10,11\}$.
Quindi, data una certa parola di n bit otteniamo 2^n combinazioni.
È importante quindi conoscere le potenze del due. Ricordarsi delle proprietà delle potenze $a^b a^c = a^{b+c}$ per calcolare potenze di valore elevato senza conoscerle a memoria.
Le potenze fondamentali sono:
$2^{20} = 1K \cdot 1K = 1M = 1.048.576$; $2^{30} = 1K \cdot 1M = 1G$; $2^{40} = 1G \cdot 1K = 1T$

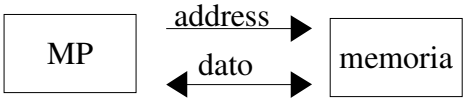
N	2^n
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

Fare attenzione alle abbreviazioni (es 1M, 1G, etc). Sono forme informatiche di abbreviazione delle quantità di dati. 1K NON è 1000, ma è 1024!

Per calcolare le potenze del 2 possiamo sfruttare le proprietà delle potenze citate in precedenza. Esempio:
$2^{19}=2^{10}\ 2^9=1024*512=524288$

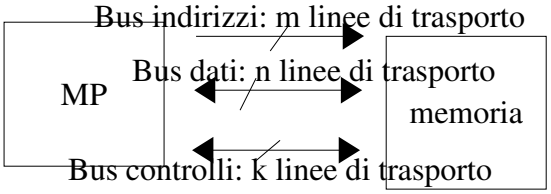
→ 1K (forma informatica per esprimere 1024)

Se vediamo più in dettaglio si può notare come il MP interagisce con la memoria:



il MP e la memoria comunicano i dati tra loro attraverso un bus dati. I dati sono allocati in memoria e ognuno di essi si trova in una cella che ha un proprio indirizzo. Il MP imposta un indirizzo per raggiungere un dato in memoria (nel caso della lettura) o imposta un indirizzo di destinazione (nel caso della scrittura).

Arrivando ad una rappresentazione completa dell'interfaccia tra MP e memoria, notiamo anche il bus controlli; queste linee di comunicazione sono bidirezionali e, per esempio, impostano lo stato di lettura o scrittura dal processore verso la memoria.



I bus, canali di comunicazione, traportano i dati sulle linee di comunicazione. Ogni linea corrisponde ad un bit.
Con si indica un bus di x bit

Conoscendo il numero di bit indirizzati dai bus possiamo conoscere le possibili locazioni di memoria e la dimensione della memoria:
locazioni di memoria=2^m (m è il numero di bit del bus indirizzi). Sapendo poi che ogni locazione di memoria individua una stringa di n bit moltiplicando il numero di locazioni di memoria per la lunghezza della stringa dati di una cella di memoria (ovvero il numero di bit del bus dati):

dimensione memoria=2^m n bit = $2^m \cdot \frac{n}{8}$ byte.

È tipico esprimere la quantità di memoria in byte e nei suoi multipli (KB, MB, ecc).
Nota: 1byte=8bit.

Notare anche il ruolo del bus controlli: per esempio il bit per la lettura/scrittura può essere indicato con $R\backslash\bar{W}$. Questa scrittura significa che se il bit relativo a questo segnale è attivo (1) allora stiamo richiedendo R, la fase di lettura, se vogliamo scrivere (W) dobbiamo porre W=1, ma W=1 => $\bar{W}=0$ quindi il bit deve essere portato ad un valore logico basso (zero).

Rappresentazione dei dati

La rappresentazione dei dati si occupa di assegnare ad una parola di n bit un significato. Ad una combinazione di bit può essere assegnato più di un significato (che varia in base alla codifica scelta)

Un esempio di codifica è il set di caratteri ASCII.

Qualsiasi informazione viene quantificata e poi codificata in linguaggio binario per essere comprensibile alla macchina. È necessario illustrare le rappresentazioni utilizzate per rappresentare le informazioni quantificate. Per esempio un utente di un programma inserisce più facilmente un numero decimale che poi verrà convertito in base binaria per essere interpretabile dal calcolatore. Un programmatore per indicare indirizzi di memoria utilizza la base esadecimale in modo da usare meno cifre per rappresentare un indirizzo di una locazione di memoria.

Basi per la rappresentazione

Dato un numerale (valore numerico di cui non si conosce la base) di n cifre e la base b, la scrittura per indicarlo è la seguente:

$(c_{n-1}c_{n-2}...c_1c_0)_b$

in cui c_{n-1} è la cifra più significativa (MSB, Most Significant Bit)
c_0 è la cifra meno significativa (LSB).

Questo numero si esprime come somma pesata, ovvero come $\sum_{i=n-1}^{0} c_i b^i$.

Esempio $(1475)_{10}=1\cdot10^3+4\cdot10^2+7\cdot10^1+5\cdot10^0$

Ricordare che un numerale si legge leggendo le cifre separatamente e non come un numero decimale.
Se abbiamo un indirizzo di memoria a 24 bit può essere più comodo rappresentarlo in base 16 ovvero in base esadecimale. Le cifre ammesse in questa base sono 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.
Ecco un esempio di conversione:

indirizzo binario 1011 0010 0110 0111 1011 1000
indirizzo hex B 2 6 7 B 8 => $(B267B8)_{16}$

è possibile passare da una base binaria ad un'altra base in modo veloce raggruppando un numero di bit n tale che 2^n=base di destinazione. Per convertire in base 8 per esempio si utilizzano gruppi di 3 bit.
Per passare, invece, da una base decimale ad un'altra base si effettuano delle divisioni successive sul numero da convertire e come divisore di utilizza la base di destinazione. Le divisioni si effettuano finché non si ottiene quoziente nullo. Il risultato si legge accodando i resti delle divisioni partendo dall'ultimo (che diventa quindi la MSB del numero convertito) verso il primo (diventa LSB).
Esempio: trovare il corrispondente in binario di $(21)_{10}$
faccio così:

$\frac{21}{2}=10\,con\,resto\,1\,(LSB),\frac{10}{2}=5\,con\,resto\,0,\frac{5}{2}=2\,con\,resto\,1,\frac{2}{2}=1\,con\,resto\,0,\frac{1}{2}=0\,con\,resto\,1\,(MSB)$

=> $(21)_{10}=(10101)_2$

Infine per passare da una qualsiasi base b alla base 10 si usa il metodo delle somme pesate, cioè

$(b_{k-1}b_{k-2}...b_0)_b=(b_{k-1}\cdot b^{k-1}+b_{k-2}\cdot b^{k-2}+...+b_0\cdot b^0)_{10}$, in cui $b_i\in\{0,...,b-1\}\,per\,i=1,...,k-1$,

k è il numero di cifre del numero di partenza e b è la base del numero di partenza, espressa in base 10.

Rappresentazione binaria di numeri interi relativi

sia $\{N\}=b_{k-1}2^{k-1}+\sum_{i=0}^{k-2} b_i 2^i$, in cui $MSB=b_{k-1}$ e $M=\sum_{i=0}^{k-2} b_i 2^i$

Rappresentazione del numero N in modulo e segno: in questo metodo viene usato il MSB per indicare il segno (1=negativo, 0=positivo) mentre le altre cifre binarie servono per rappresentare il modulo.
Quindi $\{N\}_{ms}=(-1)^{MSB}\cdot M$. In questo modo si perde un bit per rappresentare il segno ed il valore zero viene rappresentato due volte, una volta come +0 (MSB=0 e M=0) e una volta come -0 (MSB=1 e M=0). Questo è uno spreco. Per una parola di n bit il range di valori rappresentabili risulta

dall'intervallo $[-2^{n-1}+1, 2^{n-1}-1]$.
Calcolare il complemento a b di un numero : dato un numero N in base b il suo complemento a b considerando k bit è $C_b(N)=b^k-N$.

Rappresentazione del numero N in complemento a 1: $\{N\}_{c_1}=MSB(-2^{k-1}+1)+M$. Anche in questa rappresentazione lo zero è rappresentato due volte (una volta quando MSB=0 e M=0, l'altra quando MSB=1 e M=tutti 1=$2^{k.1}$-1).
Il complemento a 1 di N, in base 2, su k bit, si calcola così: $C_1(N)=2^k-1-N$. Per calcolarlo in modo rapido basta notare che in binario $C_1(N)$=NOT(N), cioè che il complemento ad 1 di N si calcola scambiando in N, bit per bit, zero con uno e viceversa. Esempio: N=010110 => $C_1(N)$=1000000-1-010110=101001.

Rappresentazione del numero N in complemento a 2: $\{N\}_{c_2}=MSB(-2^{k-1})+M$. In questa rappresentazione lo zero è rappresentato una sola volta (solo quando MSB=0 e M=0; infatti quando MSB=1 e M=tutti 1=2^{k-1}-1 ottengo il numero -1). In questo modo il range di rappresentazione si amplia e diventa $[-2^{k-1}, 2^{k-1}-1]$. Si nota inoltre che $\{N\}_{c_2}=\{N\}_{c_1}-MSB$.
Il complemento a 2 di N, in base 2, su k bit, si calcola così: $C_2(N)=2^k-N$. È immediato vedere che $C_2(N)=C_1(N)+1$. Una regola veloce per trovare direttamente il complemento a 2 di un numero N è riscrivere il numero N da destra verso sinistra finché non si incontra un bit 1; una volta scritto anche tale bit si procede a scrivere i bit seguenti (sempre verso sinistra) ricordandosi di scrivere 1 al posto di 0 e viceversa. Ad esempio: N=010110 => $C_2(N)$=1000000-010110=101010.

Tabella riepilogativa di esempio, avendo a disposizione 4 bit

Decimale	***Modulo e segno***	***Complemento a 1***	***Complemento a 2***
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000; -8
-0	1000	1111; -7	
-1	1001	1110; -6	1111; -7
-2	1010	1101; -5	1110; -6
-3	1011	1100; -4	1101; -5
-4	1100	1011; -3	1100; -4

Decimale	*Modulo e segno*	*Complemento a 1*	*Complemento a 2*
-5	1101	1010; -2	1011; -3
-6	1110	1001; -1	1010; -2
-7	1111	1000; -0	1001; -1
-8	Non rappresentabile	Non rappresentabile	1000; -0

Note:
-dato un numero binario espresso in complemento a 2 si può trovare il suo equivalente decimale, con il seguente modo:
$(b_{k-1}b_{b-2}...b_0)_{C_2}=(b_{k-1}(-2^{k-1})+b_{k-2}\cdot 2^{k-2}+b_{k-3}\cdot 2^{k-3}+...+b_0\cdot 2^0)_{10}$.
Ad esempio $(101)_{C_2}=1\cdot(-2^2)+0\cdot 2^1+1\cdot 2^0=-4+1=-3$.
-se provo a rappresentare in complemento a due con k bit un numero non rappresentabile con soli k bit in tale rappresentazione ottengo come risultato il numero dal quale sono partito. Ad esempio
$C_2(100)\,con\,3\,bit=100$ per il fatto che per rappresentare 100 (cioè 4) ho bisogno di almeno 4 bit. Con 4 bit ottengo $C_2(100)=1100$.

Le memorie degli elaboratori hanno celle per la rappresentazione dei valori di lunghezza fissa. Spesso però devono essere svolti calcoli tra i valori di lunghezza differente. In rappresentazione complemento a due, per esempio, non è possibile effettuare direttamente la somma di due numeri di lunghezza differente.
Proprietà dell'estensione di segno: afferma che un numero in complemento a due è uguale ad un altro numero in complemento a due costituito da un gruppo di bit significativi di lunghezza arbitraria e di valore 1 seguito dal numero iniziale, ad esempio $(-3)_{10}\rightarrow(101)_{C_2}=(1101)_{C_2}$, lo si vede facilmente ricordando che $(101)_{C_2}=-4+0+1=-3$ e che $(1101)_{C_2}=-8+4+0+1=-3$.

Operazione somma nel calcolatore

In binario: 0+0=0, 1+0=0+1=1, 1+1=0 con riporto di 1.
Ecco alcuni esempi di procedure per effettuare la somma di due numeri binari di due bit (il numero di bit è indipendente da ciò che si deve dimostrare, per semplicità ne sono stati scelti due). Supponiamo di avere quindi a disposizione 2 bit e che i numeri siano espressi in complemento a 2.

```
 11+
 01=
100
```

dove **1** in grassetto è il bit di riporto

```
 01+
 01=
010
```

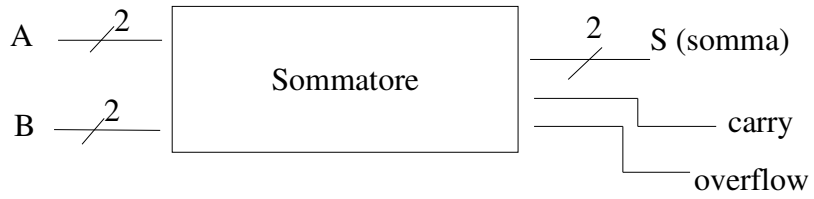
in questo caso il bit di riporto è nullo. Il risultato è sbagliato, il numero +2 non è rappresentabile in C_2 con soli 2 bit

```
 11+
 10=
101
```

in questo caso il bit di riporto è 1. Il risultato è sbagliato (devo considerare solo i 2 bit di destra), è la somma di due negativi e non può essere un positivo

Chiamiamo il bit di riporto con il termine CARRY.

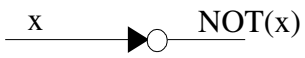
Diciamo che abbiamo un OVERFLOW quando il risultato di una somma non è rappresentabile con il numero di bit a disposizione o non è logicamente accettabile.
È possibile notare che l'overflow in C_2 si verifica quando, avendo a disposizione k bit, sommando due quantità dello stesso numero di bit e dello stesso segno si ottiene una quantità, considerando k cifre a partire da destra, di segno opposto (quindi senza considerare il riporto).
In generale se devo fare $a_1 a_0 + b_1 b_0 = s_1 s_0$ ho un overflow <=> OR(AND($a_1 = 1, b_1 = 1, s_1 = 0$), AND($a_1 = 0, b_1 = 0, s_1 = 1$)
Schematizzazione di un sommatore:



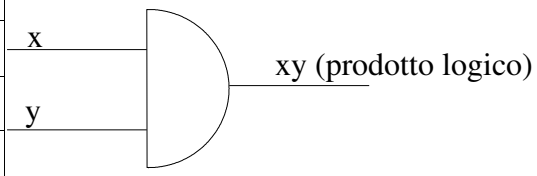
Operatori logici (porte logiche)
Una porta logica è un circuito digitale composto principalmente da transistor (componenti elettronici) che effettua calcoli in logica booleana sugli ingressi. Le porte logiche fondamentali sono tre: la negazione (NOT), la somma logica (OR) e il prodotto logico (AND).

operatore NOT	
x	***NOT(x) =*** $\bar{x}$
0	1
1	0

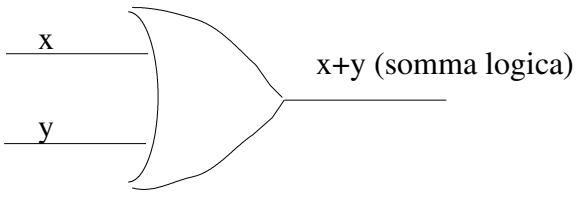
Rappresentazioni logiche



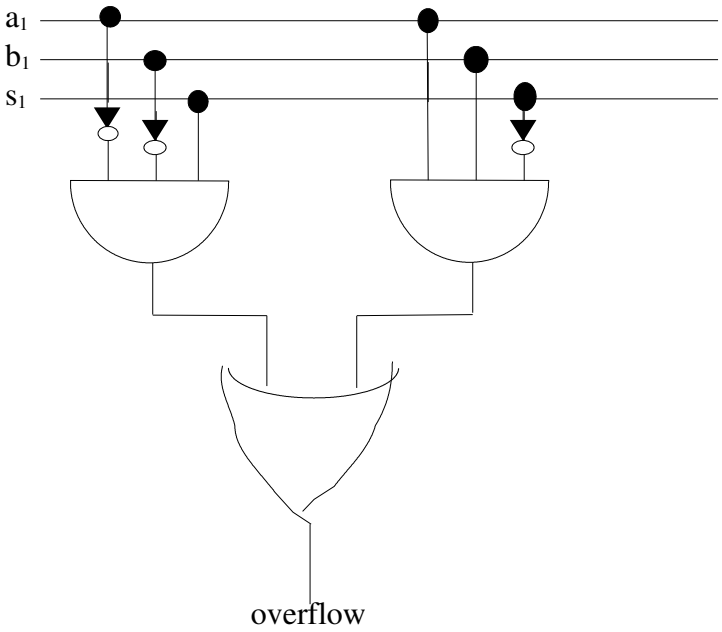
operatore AND		
x	***y***	***AND(x, y)=xy (prodotto logico)***
0	0	0
0	1	0
1	0	0
1	1	1



operatore OR		
x	***y***	***OR(x, y)=x+y (somma logica)***
0	0	0
0	1	1
1	0	1
1	1	1



NOTA: per rappresentare una negazione posso usare anche un pallino (vuoto) e basta, inserito ad esempio al termine di una linea o sulla punta di un OR di un AND.
Posso quindi riscrivere la condizione per l'overflow in questo modo: $a_1 \cdot b_1 \cdot \bar{s}_1 + \bar{a}_1 \cdot \bar{b}_1 \cdot s_1$
o anche



shift left e shift right
-dato un numero in binario l'operazione di scorrimento a sinistra di p bit (ottenuta scrivendo n volte 0 a destra di LSB), detta shift left, corrisponde ad una moltiplicazione per 2^p (attenzione se il range di rappresentazione è limitato!);
-dato un numero in binario l'operazione di scorrimento a destra di p bit (ottenuta scrivendo n volte 0 a sinistra di MSB e cancellando ogni volta l'LSB (il numero di bit è fissato)), detta shift right, corrisponde ad una divisione per 2^p. Se i vari LSB sono 0, la divisione è esatta. Esempio: lo shift a destra di $(1101)_2$ è $(0110)_2$

Rappresentazione di caratteri alfanumerici

I caratteri alfanumerici comprendono caratteri numerici ('0', '1', '2', '3', ..., '9'), caratteri alfabetici ('a', ..., 'z', 'A', ..., 'Z') e simboli ('.', ':', ...). Per rappresentare un carattere alfanumerico in un elaboratore, che lavora solo in binario, è necessario stabilire una corrispondenza fra un insieme di bit ed i caratteri da rappresentare.
Per la conversione esiste la codifica ASCII (acronimo di American Standard Code for Interchange Information) che ad ogni carattere assegna un corrispondete numero binario di 8 bit (lo standard originario era su 7 bit, ma è stato ampliato per rappresentare più caratteri, mettendo l'ottavo bit a zero). La tabella di corrispondenza fra caratteri e cifre binarie si può trovare su internet //? copia incollare la tabella! ?//.
Attenzione però, in generale $0 \neq '0'$, cioè la rappresentazione binaria del numero zero è diversa dal

numero binario che corrisponde al carattere alfanumerico '0', infatti 0 è rappresentato con tutti i bit a zero, mentre '0' in ASCII è 0011 0000, cioè 30h (h sta per 'base esadecimale'). Per passare da un carattere numerico al corrispondente numero ci sono due metodi (validi solo per il fatto che la tabella di codifica dei caratteri ASCII è stata costruita in un certo modo).
Il primo metodo di tipo aritmetico consente di scrivere che 7='7'-'0', cioè
$0000\,0111 = 0011\,0111 - 0011\,000$, il secondo metodo di tipo logico, che impiega meno tempo per essere eseguito, consente di scrivere che 7=AND('7', 0Fh), cioè
0000 0111=AND(0011 0111, 0000 1111).

Esiste inoltre un altro tipo di rappresentazione, chiamata rappresentazione BCD (Binary Coded Decimal), che ad ogni cifra decimale associa la sua rappresentazione binaria su 4 bit. Ad esempio il numero 147 in BCD è 0001 0100 0111, che in binario corrisponde a 10010011

Logica booleana

Usando i tre operatori AND, OR, NOT sopra descritti è possibile costruire qualsiasi funzione logica $f:\{0,1\}^n \to \{0,1\}^m$, espressa in forma vettoriale, dove n è il numero di bit che compongono le parole (in genere le funzioni si esprimono componente per componente).

Algebra booleana (tutte le dimostrazioni possono essere fatte per induzione perfetta, cioè provando tutti i possibili casi costruendosi le necessarie tabelle di verità)

proprietà		
proprietà di idempotenza	$x+x=x$	$x\cdot x=x$
	$x+\bar{x}=1$	$x\cdot\bar{x}=0$
elemento neutro	$x+0=x$	$x\cdot 1=x$
elemento forzante	$x+1=1$	$x\cdot 0=0$
commutativa	$x+y=y+x$	$x\cdot y=y\cdot x$
associativa	$(x+y)+z=x+(y+z)$	$(x\cdot y)\cdot z=x\cdot(y\cdot z)$
distributiva	$z(x+y)=(z\cdot x)+(z\cdot y)$	$(x+y)\cdot(x+z)=x+(y\cdot z)$
proprietà di assorbimento	$x+x\cdot y=x$ dim: $x+x\cdot y=x\cdot 1+x\cdot y=x\cdot(1+y)=$ $=x\cdot 1=x$	$x\cdot(x+y)=x$ dim: $x\cdot(x+y)=x\cdot x+x\cdot y=$ $=x+x\cdot y=x$
doppia negazione (è autoduale)	$\bar{\bar{x}}=x$	
leggi di De Morgan	$\overline{x+y}=\bar{x}\cdot\bar{y}$	$\overline{x\cdot y}=\bar{x}+\bar{y}$

Tutte le proprietà elencate sono state date in coppia, tale fatto deriva dal principio di dualità. Per passare da una proprietà alla corrispondente è necessario scambiare gli OR con gli AND e viceversa e gli 1 con gli 0 e viceversa

Funzioni logiche

Se ho n ingressi, in generale possono avere $2^{(2^n)}$ funzioni univoche associate.

Esempio: se ho 1 ingresso (n=1) posso trovare 4 funzioni, cioè:

x	f_0 (id)	f_1 (not)	f_2	f_3
0	0	1	0	1
1	1	0	0	1

Esempio: se ho 2 ingressi (n=2) posso trovare 16 funzioni, cioè:

x	***y***	***0***	***AND***	***x>y***	***x***	$x<y$	***y***	***XOR*** $x\neq y$	***OR***	***NOR***	***NXOR*** ***x=y***	$\bar{y}$	$x\geq y$	$\bar{x}$	$x\leq y$ $x\Rightarrow y$	***NAND***	***1***
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

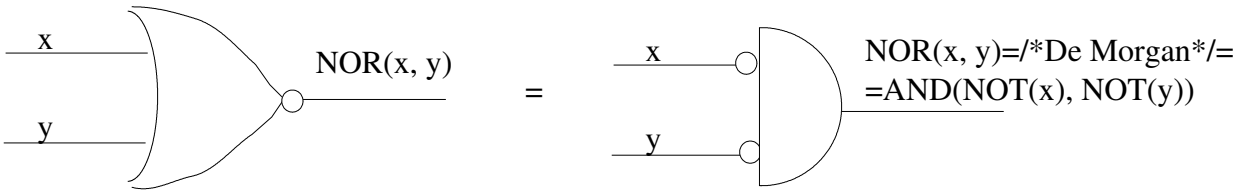
Note: NOR sta per NOT OR, NAND sta per NOT AND, XOR sta per eXclusive OR (OR esclusivo), NXOR sta per NOT XOR.

In generale se ho due ingressi posso scrivere

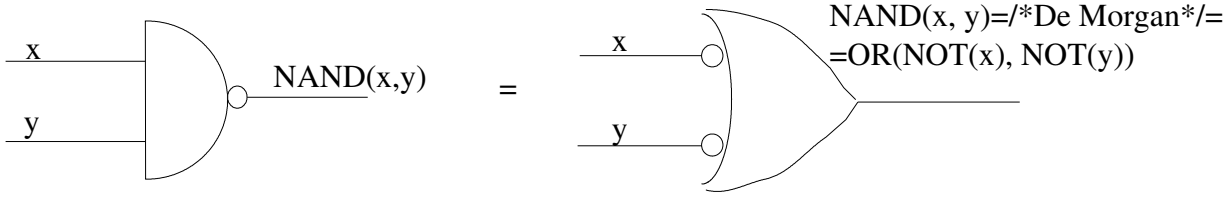
x	***y***	***f***
0	0	f_{00}
0	1	f_{01}
1	0	f_{10}
1	1	f_{11}

rete logica: traduzione di una funzione logica in termini di circuito.

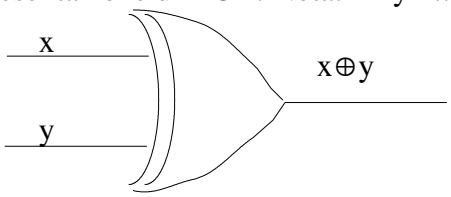
NOR si rappresenta negando l'OR, cioè con il pallino sulla punta della “freccia”



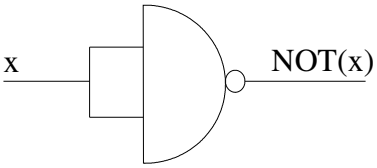
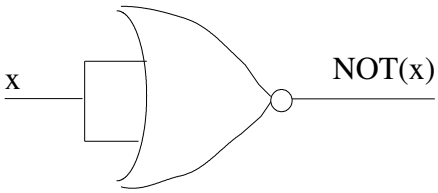
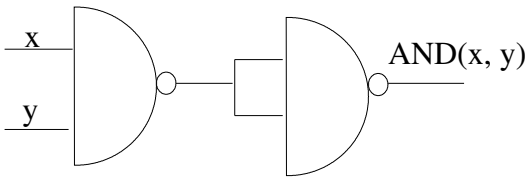
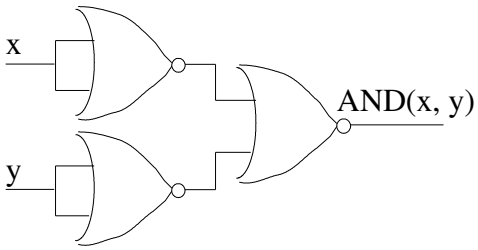
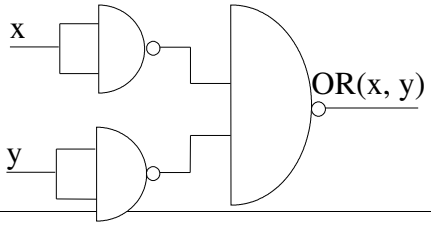
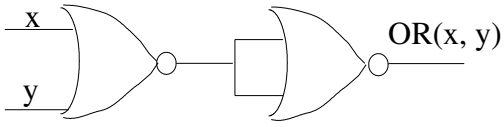
la stessa cosa vale per il NAND



Rappresentazione di XOR. Nota: $x \oplus y = \bar{x}\,y + x\,\bar{y}$ (prima forma canonica)



È possibile scrivere qualsiasi funzione logica utilizzando esclusivamente l'operatore NAND oppure l'operatore NOR.
Dimostrazione: (ricordando che NOT(AND(x, y))=NAND(x, y) e che NOT(OR(x, y))=NOR(x, y))

Operatore	***NAND***	***NOR***
NOT	NOT(x)=NOT(AND(x, x))= =NAND(x, x)  	NOT(x)=NOT(OR(x, x))=NOR(x, x) 
AND	AND(x, y)=NOT(NOT(AND(x, y)))=NOT (NAND(x, y))= /*uso che NOT(x)=NAND(x, x)*/ =NAND(NAND(x, y), NAND(x, y))  	AND(x, y)=NOT(NOT(AND(x, y)))= /*uso De Morgan*/ =NOT(OR(NOT x, NOT y))= =NOR(NOT x, NOT y)= /*uso che NOT(x)=NOR(x, x)*/ =NOR(NOR(x, x), NOR(y, y) 
OR	OR(x, y)=NOT(NOT(OR(x, y)))= /*uso De Morgan*/ =NOT(AND(NOT x, NOT y))= =NAND(NOT x, NOT y)= /*uso che NOT(x)=NAND(x, x)*/ =NAND(NAND(x, x), NAND(y, y))  	OR(x, y)=NOT(NOT(OR(x, y)))= =NOT(NOR(x, y)= /*uso che NOT(x)=NOR(x, x)*/ =NOR(NOR(x, y), NOR(x, y)) 

Forme canoniche

Ci sono due forme canoniche, la prima forma canonica (SP, somma di prodotti) e la seconda forma canonica (PS, prodotti di somme).

Prendiamo ad esempio questa funzione generale con due ingressi:

x	***y***	***f***
0	0	f_{00}
0	1	f_{01}
1	0	f_{10}
1	1	f_{11}

posso scrivere una qualsiasi di queste funzioni in questo modo:

$$f(x,y)=\bar{x}\,\bar{y}\,f_{00}+\bar{x}\,y\,f_{01}+x\,\bar{y}\,f_{10}+x\,y\,f_{11}$$

ciascun prodotto degli ingressi si chiama prodotto fondamentale. Ogni prodotto fondamentale vale 1 soltanto per una unica configurazione di ingresso.

Data una funzione, per trovarne la prima forma canonica (o somma di prodotti, SP) si deve scrivere la somma di tanti addendi quante sono le righe in cui la funzione vale 1 e ciascun addendo è costituito dal prodotto di tutti i termini (prodotti fondamentali o mintermini), ciascuno dei quali appare in forma negata o meno, a seconda del suo valore in corrispondenza di una certa riga (va negato se il suo valore è 0).

Ad esempio per questa funzione di tre variabili

x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

la prima forma canonica si scrive così:

$$f(x_1,x_2,x_3)=\bar{x}_1 x_2 \bar{x}_3+\bar{x}_1 x_2 x_3+x_1 x_2 \bar{x}_3+x_1 x_2 x_3$$

in cui ogni addendo è un prodotto fondamentale. In questo caso la funzione è sempre uguale ad x_2, quindi si può semplificare. Infatti

$$\bar{x}_1 x_2 \bar{x}_3+\bar{x}_1 x_2 x_3+x_1 x_2 \bar{x}_3+x_1 x_2 x_3=\bar{x}_1 x_2(\bar{x}_3+x_3)+x_1 x_2(\bar{x}_3+x_3)=\bar{x}_1 x_2+x_1 x_2=x_2(\bar{x}_1+x_1)=x_2 \quad .$$

Data una funzione, per trovarne la seconda forma canonica (prodotti di somme, PS) si deve scrivere il

prodotto di tanti fattori tanti quante sono le righe in cui la funzione vale zero e ciascun fattore è costituito dalla somma di tutti i termini (somme fondamentali o maxtermini), ciascuno dei quali appare in forma negata o meno a seconda del suo valore in corrispondenza di una certa riga (va negato se il suo valore è 1). La seconda forma canonica dell'esempio sopra è la seguente:

$f(x_1, x_2, x_3)=(x_1+x_2+x_3)\cdot(x_1+x_2+\bar{x}_3)\cdot(\bar{x}_1+x_2+x_3)\cdot(\bar{x}_1+x_2+\bar{x}_3)$

in cui ogni fattore è una somma fondamentale.
È importante notare che da un punto di vista logico le due forme sono equivalenti, rappresentano la stessa funzione. Notare inoltre che la seconda forma canonica non è altro che la forma duale della prima forma canonica.

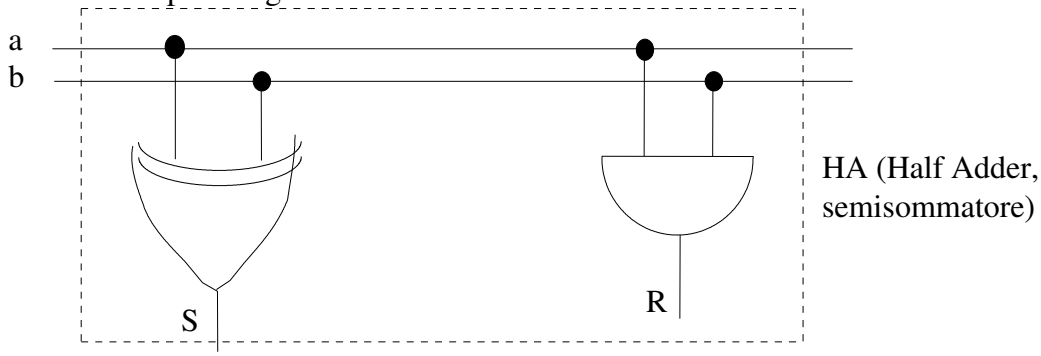
Esercizio:
scrivere la funzione *f(a, b, c)=a⊕ (b+c)* in prima forma canonica e rappresentarla usando solo operatori NAND.
*a⊕(b+c) = /*prima forma canonica di XOR*/=* $\bar{a}(b+c)+a(\overline{b+c})$ =
= $\bar{a}\,b+\bar{a}\,c+a\,\bar{b}\,\bar{c}$ = /*moltiplico il primo termine per $(c+\bar{c})$ ed il secondo per $(b+\bar{b})$ (che sono uno)*/= $\bar{a}\,b\,c+\bar{a}\,b\,\bar{c}+\bar{a}\,b\,c+\bar{a}\,\bar{b}\,c+a\,\bar{b}\,\bar{c}$ = /*elimino il terzo termine perché x+x=x **/ =**
$\bar{a}\,b\,c+\bar{a}\,b\,\bar{c}+\bar{a}\,\bar{b}\,c+a\,\bar{b}\,\bar{c}$ **ed ottengo la funzione in prima forma canonica (SP).**
Negando due volte ogni addendo ottengo
$\overline{\overline{\bar{a}\,b\,c}}+\overline{\overline{\bar{a}\,b\,\bar{c}}}+\overline{\overline{\bar{a}\,\bar{b}\,c}}+\overline{\overline{a\,\bar{b}\,\bar{c}}}$ **=** */*uso che NOT(AND(x, y, z))=NAND(x, y, z) e che OR(* $\bar{a}, \bar{b}, \bar{c}, \bar{d}$ *)=NOT(AND(a, b, c, d))=NAND(a, b, c, d) */ =*
=NAND(NAND($\bar{a}, b, c$), NAND($\bar{a}, b, \bar{c}$), NAND($\bar{a}, \bar{b}, c$), NAND($a, \bar{b}, \bar{c}$))

Sommatore

Si vuole costruire un circuito che faccia la somma aritmetica a n bit facendo delle operazioni logiche.
Vediamo come sommare due parole di 1 bit:

a	*b*	*S(a, b) (somma)*	*R(a, b) (riporto)*
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Si vede immediatamente che S(a, b)=XOR(a, b)=*a*⊕b e che R(a, b)=AND(a, b)=ab
Rappresentazione a porte logiche:



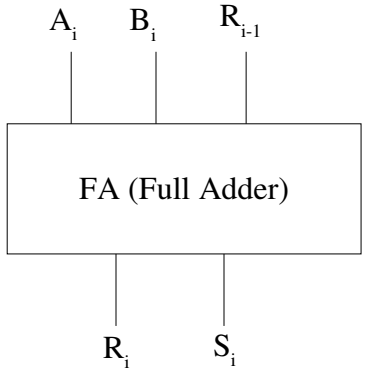
Consideriamo ora di avere due parole di k bit:

$A = A_{k-1} A_{k-2} \ldots A_2 A_1 A_0$

$B = B_{k-1} B_{k-2} \ldots B_2 B_1 B_0$

Per costruire un sommatore completo è necessario tenere conto del riporto R_{i-1} ottenuto dalla somma delle cifre precedentemente sommate, per cui:

A_i	B_i	R_{i-1}	S_i	R_i
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



Scriviamo il tutto in prima forma canonica:

$S_i = \bar{A}_i \bar{B}_i R_{i-1} + \bar{A}_i B_i \bar{R}_{i-1} + A_i \bar{B}_i \bar{R}_{i-1} + A_i B_i R_{i-1} = \bar{A}_i (\bar{B}_i R_{i-1} + B_i \bar{R}_{i-1}) + A_i (\bar{B}_i \bar{R}_{i-1} + B_i R_{i-1}) =$

=/*il contenuto della prima parentesi è la forma canonica di XOR, il contenuto della seconda parentesi è la forma canonica di XOR negato*/=

$= \bar{A}_i (B_i \oplus R_{i-1}) + A_i \overline{(B_i \oplus R_{i-1})}$ =/*è un altro XOR negato*/=

$= A_i \oplus (B_i \oplus R_{i-1})$

nota: questa espressione, in pratica, controlla che il numero complessivo di 1 in ingresso sia dispari e se lo è restituisce 1, altrimenti zero.

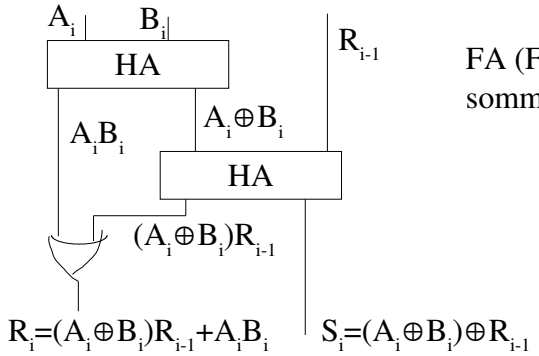
mentre per R_i la forma canonica è la seguente:

$R_i = \bar{A}_i B_i R_{i-1} + A_i \bar{B}_i R_{i-1} + A_i B_i \bar{R}_{i-1} + A_i B_i R_{i-1} = R_{i-1} (\bar{A}_i B_i + A_i \bar{B}_i) + A_i B_i =$

$= (A_i \oplus B_i) R_{i-1} + A_i B_i$ per costruire il sommatore si utilizza questa espressione, che può però diventare

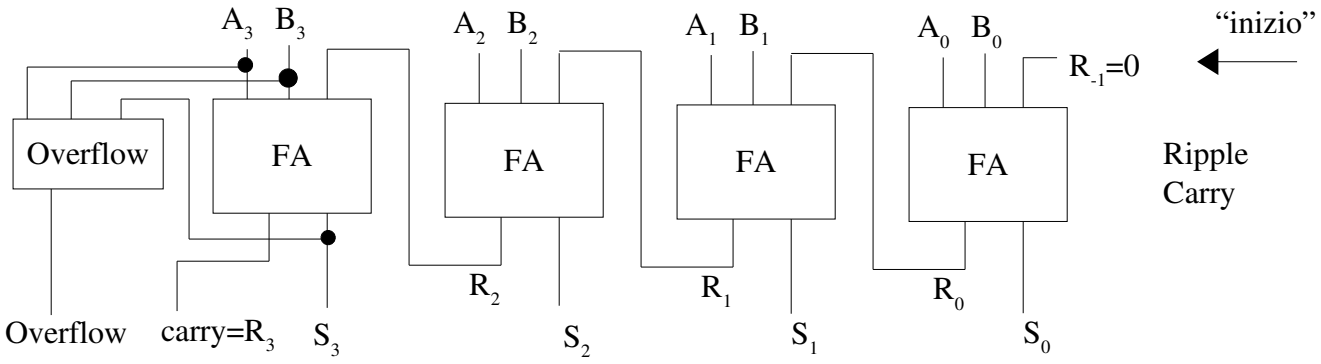
$= (A_i + B_i) R_{i-1} + A_i B_i = A_i R_{i-1} + B_i R_{i-1} + A_i B_i$

Rappresentazione a porte logiche del Full Adder, per mezzo dell'HA:



FA (Full Adder): esegue la somma di due bit con riporto

Per fare la somma di due parole di k bit è necessario quindi usare k FA, collegandone uno con il successivo per mezzo del riporto. Ad esempio per k=4 bit:

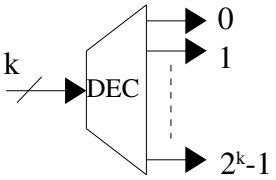


C'è da dire che il risultato fornito NON è immediatamente il risultato esatto, in quanto per avere un bit corretto nel risultato è necessario che siano stati prima calcolati tutti i bit precedenti, per via dei riporti. Quindi se il tempo di calcolo per ogni FA è T, ottengo LSB dopo un tempo T ed il risultato completo ed esatto dopo un tempo 4T dall'inizio del processo.
NOTA: il circuito di overflow, che preleva A_3, B_3, S_3, è implementabile con il metodo visto in precedenza.

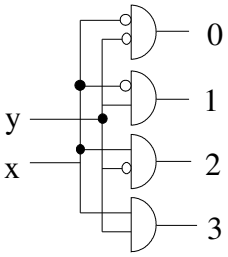
Moduli di logica combinatoria

decodificatori (decoder)

Un decodificatore (decoder) è un componente combinatorio binario che accetta in ingresso un codice di k bit e che presenta in uscita 2^k linee.
In uscita viene asserita la sola linea che corrisponde alla codifica in ingresso.
Le linee di uscita sono numerate da 0 a 2^k-1, in modo tale che se in ingresso ho una certa parola in uscita ho asserito solo la linea il cui numero rappresenta tale parola.



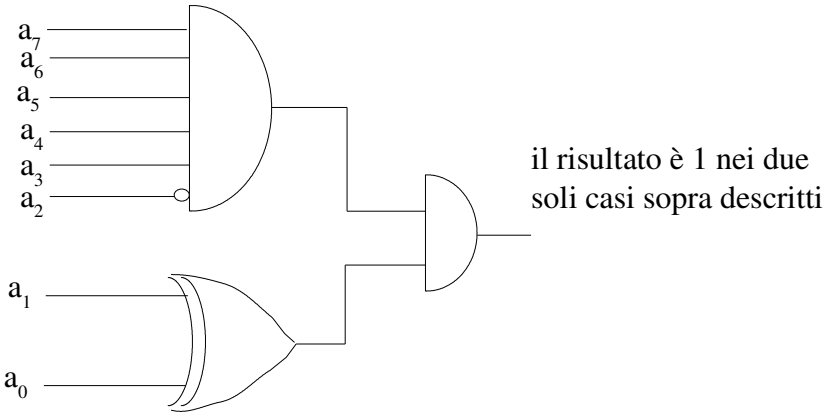
Esempio con n=2: se (x, y)=(1, 0) allora in uscita la sola linea che vale 1 è la numero 2



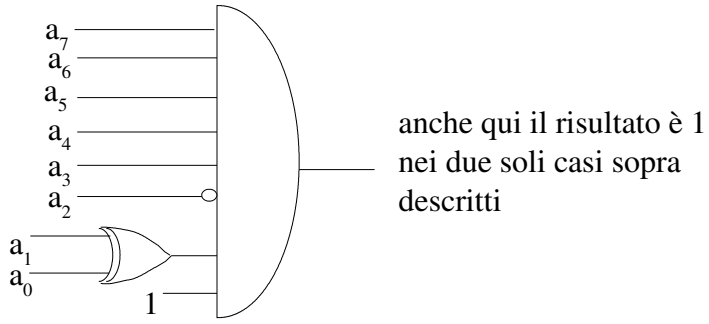
Ci sono però dei casi in cui posso non essere interessato a decodificare tutti i possibili ingressi, ma solo alcuni. Ad esempio nelle interfacce dei dispositivi (ad esempio per l'interfaccia di I/O) ci sono dei decodificatori di indirizzo che rispondono solo a particolari indirizzi.
Supponiamo ad esempio che l'interfaccia di I/O risponda solo agli indirizzi F9h e FAh (dove h sta ad indicare che gli indirizzi sono espressi in formato esadecimale) con un bus a 8 bit.

```
F9h=11111001
Fah=11111010
```

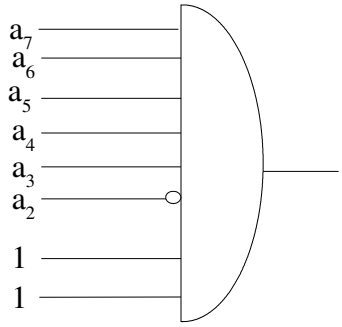
Si nota che solo i primi due bit sono diversi. Il tutto può essere realizzato così:



oppure anche così, tramite una porta AND a 8 ingressi (tipicamente il numero di ingressi delle porte è pari a 2^n) ed una XOR.



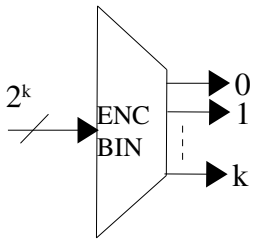
ultimo esempio: la seguente porta



risponde a

```
F8h=11111000
F9h=11111001
FAh=11111010
FBh=11111011
```

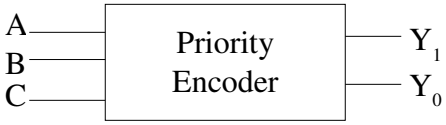
codificatori (encoder)



Un codificatore binario senza priorità fa l'inverso di quel che fa un decodificatore. Un codificatore binario ammette 2^k ingressi e k uscite. Ogni ingresso deve contenere obbligatoriamente un (unico) 1, cioè se gli ingressi sono $x_{0,} x_{1,} \ldots, x_{2^k-1}$ e x_i=1 allora deve essere che $x_j = 0$ per ogni $j \neq i$ e la corrispondente configurazione di uscita vale, per il codificatore binario, i, tradotto in binario (in genere gli indici sono espressi in base 10).
esempio: se gli ingressi (x) sono 2^2 (k=2) parte della tabella di verità di un codificatore binario è questa:

x_3	x_2	x_1	x_0	z_1	z_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Un codificatore con priorità (priority encoder) ha invece più ingressi che possono essere asseriti e quindi deve avere una "politica delle priorità" per stabilire quale valore passare in uscita. Un esempio di priority encoder è questo:



che potrebbe avere questa tabella di verità:

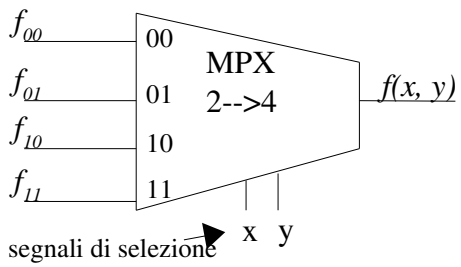
A	B	C	Y_1	Y_0
0	0	0	0	0
1	X	X	0	1
0	1	X	1	0
0	0	1	1	1

nota: X=don't care, cioè non importa quale valore abbia.
In questo esempio la priorità è massima per A e minima per C

Un codificatore non binario ha n ingressi ed m uscite, n e m arbitrari, senza vincoli su quali delle m uscite possono essere asserite.

multiplexer

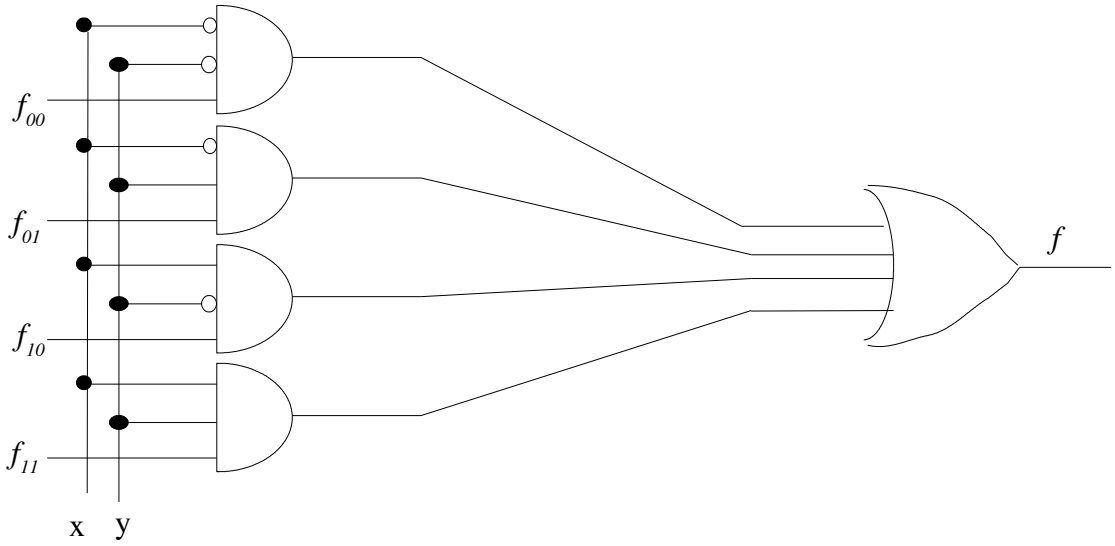
Un multiplexer (MPX) è un dispositivo che permette di selezionare, in base ad un segnale di selezione, uno dei k ingressi e di inviarlo sull'unica uscita presente. Con un multiplexer è possibile realizzare una qualsiasi funzione logica.



Esempio, nel caso di una funzione di due variabili *f(x, y)*.
Se (x, y) è (1, 0), allora *f* assume il valore f_{10}

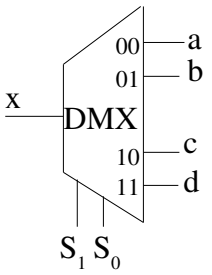
x	***f***	***f(x, y)***
0	0	f_{00}
0	1	f_{01}
1	0	f_{10}
1	1	f_{11}

Si realizza così:



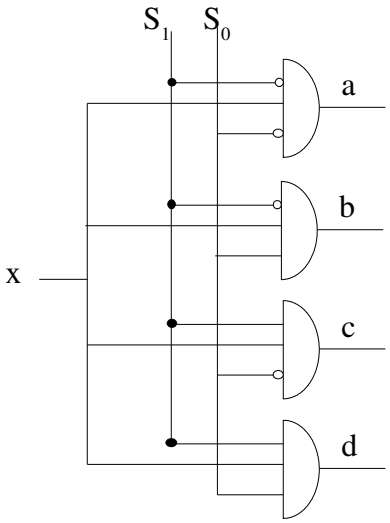
demultiplexer

Un demultiplexer (DMX) è un dispositivo che permette di dirottare l'ingresso x su una delle k uscite, in base ad un segnale di selezione S.



S_1	S_0	a	b	c	d
0	0	x	0	0	0
0	1	0	x	0	0
1	0	0	0	x	0
1	1	0	0	0	x

Si realizza così:



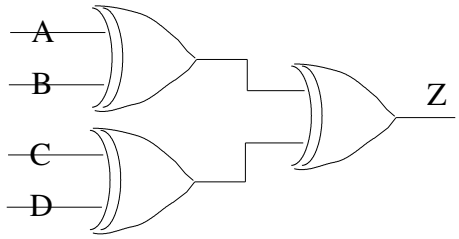
parity checker

Attraverso il parity checker è possibile rilevare errori dovuti a disturbi o a malfunzionamenti in fase di trasmissione. Il parity checker serve per controllare la parità di una parola (cioè il numero di 1 presenti nella parola; la parità è 1 se la parola è dispari, cioè se il numero di 1 è dispari, 0 se la parola è pari). Il parity checker può essere usato per controllare la presenza di errori nelle parole trasmesse, in questo modo.

Ho una parola ABCD (quindi di 4 bit). Trovo la parità P della parola $ABCDP_i$, dove P_i è 1 se voglio che tutte le parole trasmesse abbiano sempre parità dispari, 0 se voglio che tutte la parole trasmesse abbiano sempre parità pari, e trasmetto la parola ABCDP.

Il parity checker in arrivo controlla la parità della parola ABCDP ricevuta e se tale parità non corrisponde a quella prevista inizialmente allora posso dire certamente di aver avuto almeno un errore, perché un numero dispari di bit è sbagliato. Se la parità invece corrisponde a quella prevista è SOLO probabile che non ci siano stati errori, in quanto un numero pari di errori non viene rilevato (se cambio un numero pari di bit la parità della parola non varia).

Posso controllare la parità Z di una parola ABCD con l'operatore logico XOR, in questo modo (nota: per lo XOR vale la proprietà associativa):

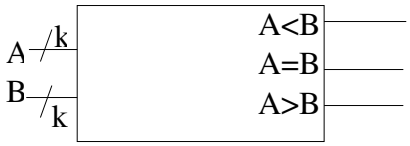


Come detto sopra tramite il partity checker può darsi che qualche errore sfugga in ogni caso (se ho un numero pari di errori).
Un altro metodo per rilevare e anche correggere gli errori consiste nel trasmettere ogni informazione più di una volta. Se inviassi ogni bit due volte avrei (nel caso del bit 0) in invio 00 ed in ricezione potrei avere 00, 01, 10, 11 rispettivamente commettendo 0, 1, 1 e 2 errori. Nel caso invece del bit 1 avrei in invio 11, e in ricezione potrei avere 11, 10, 01, 00 rispettivamente commettendo 0, 1, 1, 2 errori. Evidentemente non sono in grado, se commetto errori, di distinguere se ho inviato 0 oppure 1. Se invece invio ogni bit 3 volte, nel caso del bit 0, in invio ho 000 ed in ricezione potrei avere (sempre supponendo di commettere al più un errore) 000, 001, 010, 100. In questo modo il ricevitore in caso di errore può ad esempio stabilire che è stato inviato il bit che compare più volte, quindi rilevando e correggendo automaticamente l'errore. Se p è la probabilità di avere un errore sulla trasmissione di un bit (e quindi 1-p è la probabilità di non averlo), la probabilità di commettere al massimo un errore trasmettendo tre volte lo stesso bit è $Prob\{max\,1\,errore\}=(1-p)^3+3(1-p)^2 p$, cioè la probabilità di non commettere alcun errore (il primo addendo) o di commettere solo 2 errori (secondo addendo). In pratica questa è la probabilità di non fare alcun errore, perché se c'è un solo errore, come visto sopra, sono in grado di correggerlo. La probabilità di fare più di un errore è
$Prob\{più\,di\,un\,errore\}=1-Prob\{max\,1\,errore\}$.
Se $p<\frac{1}{2}$ allora *Prob{più di un errore}<p*.
La soluzione di inviare 3 volte lo stesso bit, pur permettendo di correggere eventuali errori, costa molto, esattamente 3 volte di più di un normale invio.

nota: i sistemi digitali sono molto resistenti al rumore (alle alterazioni dovute a interferenze, ecc) perché le informazioni sono trasmesse utilizzando due soli segnali (basso-alto, acceso-spento, ...) ed è quindi necessario un forte rumore per poter far saltare un segnale da uno stato all'altro. I sistemi analogici invece sono più soggetti al rumore e spesso tale rumore è più difficile da rimuovere.

comparatore

Un comparatore può essere schematizzato così:



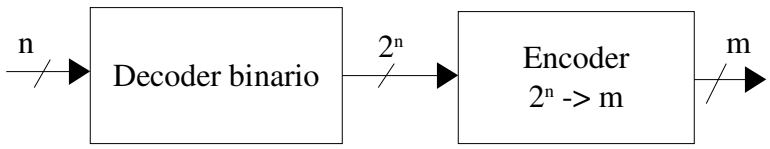
In ingresso ha due parole di k bit; ci sono 3 possibili uscite, ovviamente solo una può essere asserita.
Scriviamo $A=a_{k-1}a_{k-2}...a_1a_0$ e $B=b_{k-1}b_{k-2}...b_1b_0$ e vediamo la tabella di verità bit per bit:

a_i	b_i	$a_i=b_i$ cioè NXOR(a_i,b_i)	$a_i<b_i$ cioè $\bar{a}_i b_i$	$a_i>b_i$ cioè $a_i \bar{b}_i$
0	0	1	0	0
0	1	0	1	0
1	0	0	0	1
1	1	1	0	0

Notare che $NXOR(a_i,b_i)=NOT(\bar{a}_i b_i + a_i \bar{b}_i)$, infatti due numeri sono uguali se non vale che uno dei due è minore (o maggiore) dell'altro.
Scrivo che $E_i=NXOR(a_i,b_i)$, quindi $E_i=1 \Leftrightarrow a_i=b_i$. Quindi posso scrivere che:
- $A=B \Leftrightarrow E_{k-1}\cdot E_{k-2}\cdot ...\cdot E_1\cdot E_0=1$, cioè se e solo se tutti i bit che si corrispondono sono uguali;
- $A>B \Leftrightarrow a_{k-1}\cdot \bar{b}_{k-1} + E_{k-1}\cdot a_{k-2}\cdot \bar{b}_{k-2} + E_{k-1}\cdot E_{k-2}\cdot a_{k-3}\cdot \bar{b}_{k-3} + ...$, cioè se e solo se MSB(a) > MSB(b) oppure MSB(a)=MSB(b) e la cifra a sinistra di MSB(a) è maggiore della cifra a sinistra di MSB(b), oppure ...
- $A<B \Leftrightarrow \bar{a}_{k-1}\cdot b_{k-1} + E_{k-1}\cdot \bar{a}_{k-2}\cdot b_{k-2} + E_{k-1}\cdot E_{k-2}\cdot \bar{a}_{k-3}\cdot b_{k-3} + ...$, è uguale a sopra, solo scambio a con b.

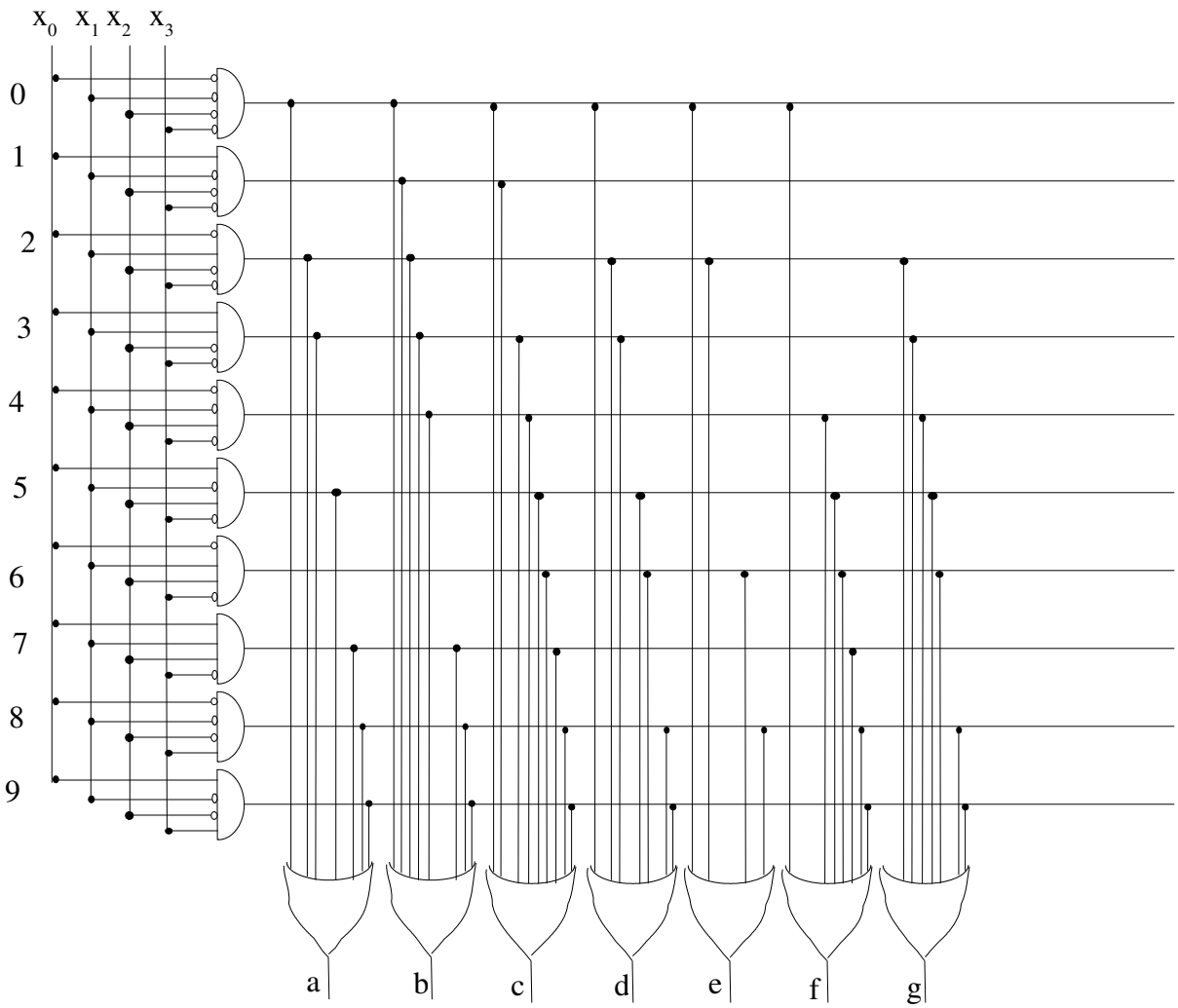
Le ROM

Una ROM (Read Only Memory) è un dispositivo logico combinatorio che realizza una qualsiasi funzione combinatoria di n ingressi ed m uscite. La ROM può essere scritta una sola volta. Una ROM può essere realizzata attraverso un decoder binario ed un encoder, con il seguente schema:



La dimensione della ROM, come la dimensione della RAM, è data da $2^n \cdot m$ bit. Notare che per ogni bit di ingresso in più la dimensione della ROM raddoppia.
Un esempio di impiego di una ROM potrebbe essere in una calcolatrice: sulla ROM dovrebbero essere codificate tutte le possibili combinazioni di operazioni disponibili con il numero di bit fissato. Una calcolatrice così costruita di fatto non calcolerebbe però un bel niente (tutti i risultati infatti sono già pronti all'interno della ROM), guadagnando in velocità ma spendendo molto (è necessario costruire una ROM che contenga tutti i possibili risultati di tutte le operazioni). Notare il conflitto memoria –

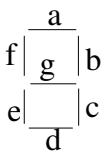
calcolo.
Esempio di ROM:



che ha questa tabella di verità:

n	***x₃***	***x₂***	***x₁***	***x₀***	***a***	***b***	***c***	***d***	***e***	***f***	***g***
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	0	0	1	1	1	1	1

n	x_3	x_2	x_1	x_0	*a*	*b*	*c*	*d*	*e*	*f*	*g*
7	0	1	1	1	1	1	1	0	0	1	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	1	0	1	1

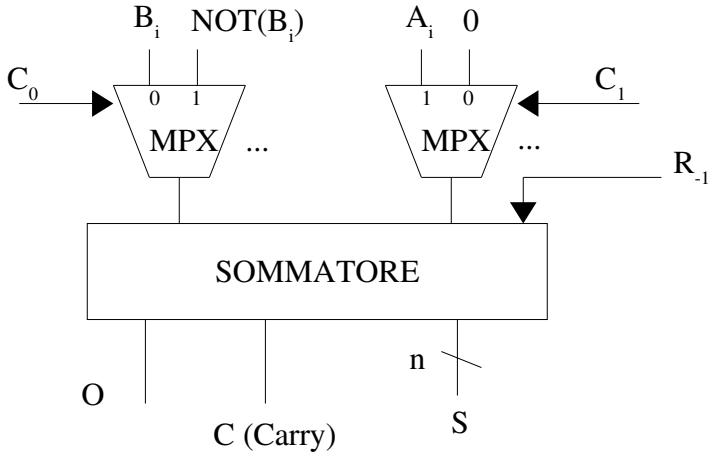


Questa ROM rappresenta un modo per effettuare la codifica di un numero da BCD a 7 segmenti, in cui BCD è la codifica presentata qualche pagina indietro e la "7 segmenti" serve per far accendere gli opportuni segmenti di un display a 7 segmenti in base al numero che si vuole mostrare.
Notare che c'è una stretta corrispondenza fra i punti neri sulla ROM che indicano i contatti e gli uno presenti nella tabella di verità. Una ROM può essere rappresentata anche in modo più schematico e generale, mettendo una sola linea generica di n bit (fili) su ogni porta AND (le porte AND sono, al massimo, 2^n) ed una sola linea generica di, al massimo, n bit (fili) per ogni porta OR, con un punto nero in corrispondenza del filo uscente dalla porta AND della quale si vuole passare il valore (quindi al massimo in corrispondenza "del" filo di ogni porta OR ci possono essere tanti punti neri quante sono le porte AND).

Esistono inoltre altri tipi di ROM, che sono:
-PROM (dove P sta per Programmable), è un tipo di ROM sulla quale è possibile anche scrivere (ma non cancellare o fare modifiche);
-EPROM, sulla quale è possibile sia scrivere che modificare.

ALU

La ALU (unità aritmetico logica) è un dispositivo combinatorio che permette di eseguire varie operazioni in base ai segnali trasmessi su delle linee di controllo. Si realizza così:



dove A_i e B_i sono i bit delle parole A e B, formate da n bit, sulle quali si deve operare; C_0 e C_1 sono linee di controllo che permettono di selezionare uno dei due ingressi previsti in ciascun multiplexer, al fine di eseguire un'operazione piuttosto che un'altra; S è il risultato dell'operazione eseguita; C è

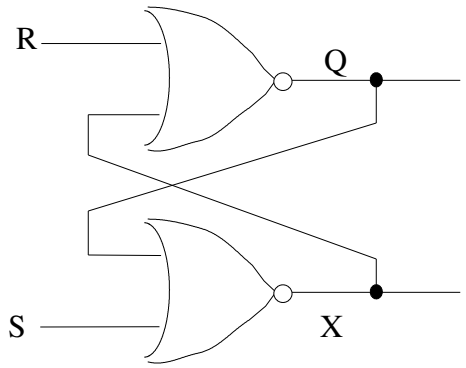
l'eventuale carry; O è l'eventuale segnale di overflow; R_{-1} è il riporto da fornire in ingresso per eseguire la somma (che quando faccio la somma pongo a zero), ma viene utilizzato anch'esso come linea di controllo nel caso in cui non voglia eseguire una semplice somma. Notare i puntini presenti a destra dei due multiplexer: questo perché il sommatore ha n ingressi per la parola A (A è di n bit) ed n ingressi per la parola B (B è di n bit), quindi in tutto ci sono 2n multiplexer, uno per ogni bit ed ognuno è collegato ovviamente alla giusto linea di controllo (i multiplexer per i bit della parola B sono collegati con C_0, quelli per i bit della parola A sono collegati con C_1).
Le operazioni eseguite dalla ALU sopra riportata sono rappresentate qui di seguito, al variare delle linee di controllo C_0, C_1, R_{-1}.

C_1	C_0	R_{-1}	*S (risultato)*	*Operazione eseguita*
0	0	0	$0+B=B$	Selezione di B
0	0	1	$0+B+1=B+1$	Incremento di B
0	1	0	$0+\bar{B}+0=\bar{B}$	NOT(B)
0	1	1	$0+\bar{B}+1=\bar{B}+1=-B$	Cambia segno a B (complemento a due)
1	0	0	$A+B$	Somma A e B
1	0	1	$A+B+1$	non significativo
1	1	0	$A+\bar{B}=A-B-1$	non significativo
1	1	1	$A+\bar{B}+1=A-B$	Differenza A-B (complemento a due)

Aggiungendo, all'interno del sommatore (realizzato come riportato sul bucci a pg 101, cioè esprimendo S_i in funzione di R_i) due linee di controllo C_2 e C_3 è possibile far eseguire alla ALU anche le operazioni AND(A, B) e OR(A, B). La tabella di verità è uguale a questa qui sopra, solo con due righe e due colonne in più (il valore di C_2 è 1 per ogni riga della attuale tabella, mentre C_3 è zero per ogni riga della attuale tabella).

Reti sequenziali

In una rete sequenziale si ha che l'uscita è funzione non solo degli ingressi, ma anche dello stato. Prendiamo una rete così fatta:



memo: NOR		
a	***b***	$\overline{a+b}$
0	0	1
0	1	0
1	0	0
1	1	0

nella quale gli ingressi sono R e S. Vediamo come opera questa rete

t	***R***	***Q***	***S***	***X***	
t_0		*0*		*0*	inizialmente pongo Q=0 e X=0
t_1	*0*	1	*1*	0	pongo R=0 e S=1, Q e X sono i precedenti
t_2	0	1	*0*	0	pongo S=0, gli altri sono i precedenti
t_3	*1*	0	0	1	pongo R=1, gli altri sono i precedenti
t_4	*0*	0	0	1	pongo R=0, gli altri sono i precedenti

Alla fine di ogni periodo di tempo la rete è in uno stato stabile, cioè le uscite rimangono costanti fino a quando non vengono variati gli ingressi.
Si vede subito che questa non è una rete combinatoria: infatti al tempo t_2 e t_4 ho gli stessi ingressi (S=R=0), ma ho due uscite diverse, in particolare in t_2 ho Q=1 e X=0 e in t_4 ho Q=0 e X=1. Quindi l'uscita non dipende solo dagli ingressi, ma anche dallo stato in cui il sistema si trova, cioè da Q (nei casi visti sopra era sempre $X=\bar{Q}$). Quindi in questa rete ho memorizzato un'informazione. Notare che ho utilizzato le stesse tecnologie utilizzate per le reti combinatorie, ho solo cambiato l'organizzazione.
Rappresento questa rete così:

e chiamo questo modulo LATCH (chiavistello).
In base al primo schema riportato vedo che:

$Q=\overline{R+X}$ e $X=\overline{S+Q}$

$Q=\bar{R}\cdot\bar{X}=\bar{R}(S+Q)=S\bar{R}+\bar{R}Q$

vediamo quando risulta verificata $Q=S\bar{R}+\bar{R}Q$:
-per ogni Q: ho che S=0 e R=0. In questa situazione Q non varia, rimane sempre uguale. Questa è la condizione di HOLD.
-se Q=0 devo avere che $S\bar{R}=0$, che posso scindere in quattro casi:
-S=0 e R=0 (caso già visto); S=0 e R=1;
-R=1 e S=0; R=1 e S=1;
cioè S=0 e R=1 oppure R=1 e S=1. In questi due casi Q viene forzato a zero (basta sostituire). Queste due sono le condizioni di RESET.
-se Q=1 devo avere che $S\bar{R}+\bar{R}=1\Rightarrow R=0$. Qui ho due casi:
-R=0 e S=0 (caso già visto);
-R=0 e S=1;
In quest'ultimo caso (R=0 e S=1) Q viene forzato a 1 (basta sostituire). Questa è la condizione di SET.

In tutti i casi, tranne quando R=1 e S=1, si ha $X=\bar{Q}$.
Nota: la combinazione R=1 e S=1 non viene utilizzata perché non consente di cambiare

contemporaneamente gli ingressi. Cioè, se gli ingressi vengono cambiati contemporaneamente quando la situazione di ingresso è R=1 e S=1, il valore di Q non è determinabile a priori perché il cambiamento degli ingressi non avviene esattamente nello stesso istante (anche se il cambiamento dell'ingresso fosse simultaneo, non lo sarebbe la reazione delle due porte NOR). Infatti nel passare da R=1 e S=1 a R=0 e S=0 se per primo cambia R si ha Q=1, mentre se per primo cambia S sia ha Q=0. Nelle altre situazioni di ingresso invece, anche in caso di cambiamento contemporaneo degli ingressi, Q ha sempre lo stesso valore sia che per primo cambi R sia che per primo cambi S.
Se indichiamo con Q lo stato presente (cioè quello che c'è in una macchina sequenziale all'istante in cui vengono applicati gli ingressi) e con Q' lo stato futuro (cioè lo stato al quale arriva Q dopo che sono stati applicati gli ingressi), possiamo costruire la seguente tabella che esprime Q' in funzione di S, R, Q:

S	***R***	***Q***	***Q'***
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0 (X)
1	1	1	0 (X)

Scrivendo la prima forma canonica (Q' è l'uscita) si ottiene: $Q' = S\bar{R} + \bar{R}Q$, che è uguale a quanto ottenuto sopra.
Per il motivo scritto sopra la combinazione di ingressi S=1 R=1 faccio in modo da non averla mai, pertanto il valore relativo a Q' in corrispondenza di tali ingressi non mi importa quale sia (X, don't care), anche se in realtà è zero. Siccome il valore Q'=X non si potrà mai avere, decido di attribuirgli, per comodità, il valore Q'=1. Posso allora modificare la forma canonica aggiungendogli i due mintermini $SR\bar{Q} + SRQ = SR$, ottenendo $Q' = S\bar{R} + \bar{R}Q + SR = S + \bar{R}Q$ (valido solo per SR=0, cioè $S \neq 1 \; e \; R \neq 1$). Posso riscrivere la tabella di verità in forma più compatta:

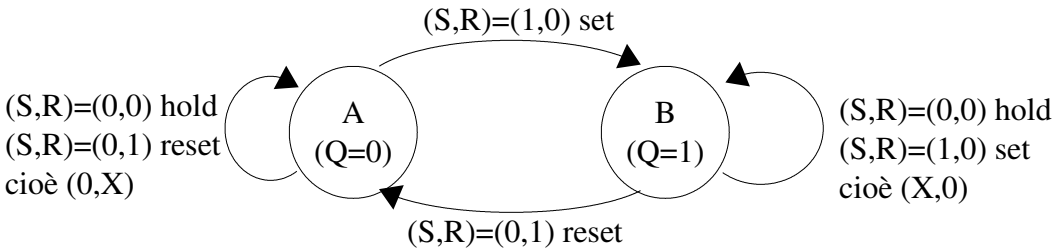
S	***R***	***Q'***	
0	0	Q	HOLD
0	1	0	RESET
1	0	1	SET
1	1	-	-

La memoria costruita si ricorda quindi un bit (Q, perché X è in funzione di Q), cioè possiede una variabile di stato e quindi due stati (zero e uno), in generale se una memoria ha n variabili di stato possiede 2^n stati.

In generale il LATCH SR (SR sta per Set Reset) lo rappresento così (sempre con SR=0):

Diagramma di stato

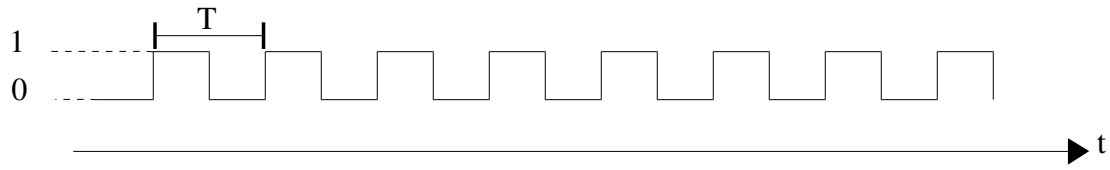
Il diagramma di stato è una rappresentazione grafica degli stati, degli ingressi e delle uscite e di come sono collegati tra loro. È il modo più naturale per descrivere il comportamento di una rete sequenziale. Questo è il diagramma di stato del LATCH SR:



in cui ogni palla rappresenta uno stato, e ogni freccia è una transizione, funzione degli ingressi (indicati al fianco della freccia) e dello stato corrente (punto di partenza della freccia).

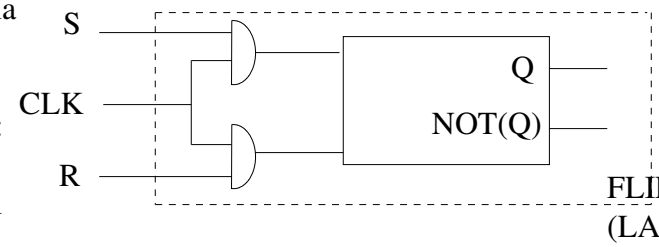
Una rete sequenziale può essere di tipo asincrono o di tipo sincrono. Una rete di tipo asincrono reagisce immediatamente alla variazione degli ingressi, portandosi dallo stato presente al nuovo stato nel solo tempo richiesto dalla comunicazione dei suoi componenti. Una rete si dice sincrona invece quando gli ingressi vengono analizzati solo dopo l'impulso di un segnale detto “clock” (CLK). Fino ad ora gli esempi sono stati esempi di reti sequenziali asincrone.

Il segnale di clock (CLK) ha un certo periodo, T (reciproco della frequenza). Questa è la schematizzazione del segnale di clock in funzione del tempo:



nota: in realtà queste linee non sono verticali!

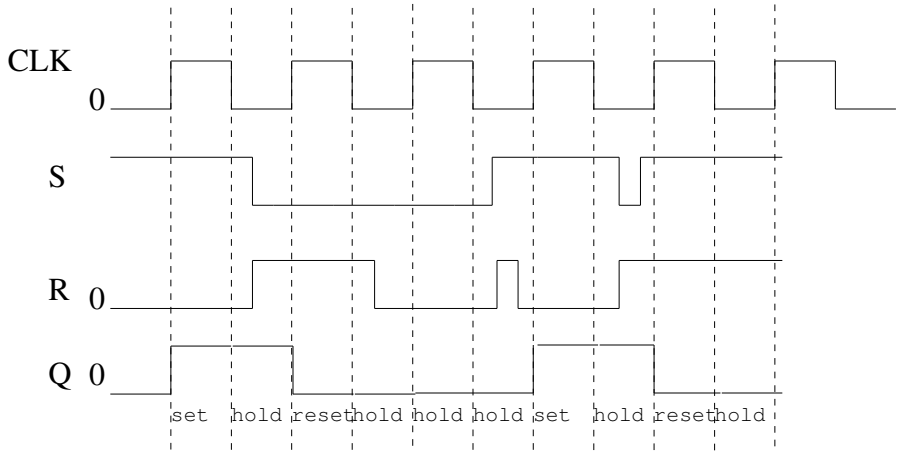
Gli ingressi vengono “sentiti” dal LATCH solo quando il clock è alto, mentre quando è basso lo stato del LATCH rimane lo stesso (la variazione degli ingressi non viene sentita). Questo effetto viene ottenuto in questo modo:



FLIP FLOP SR
(LATCH con CLOCK)

Questo LATCH con segnale di clock prende il nome di FLIP FLOP SR. Notare che se CLK è a zero l'ingresso del latch è 00 qualunque siano S e R: ciò corrisponde

alla condizione di hold. Durante questa fase posso cambiare gli ingressi ed avrò il cambiamento di stato solo quando il clock sarà tornato a 1. Esempio:

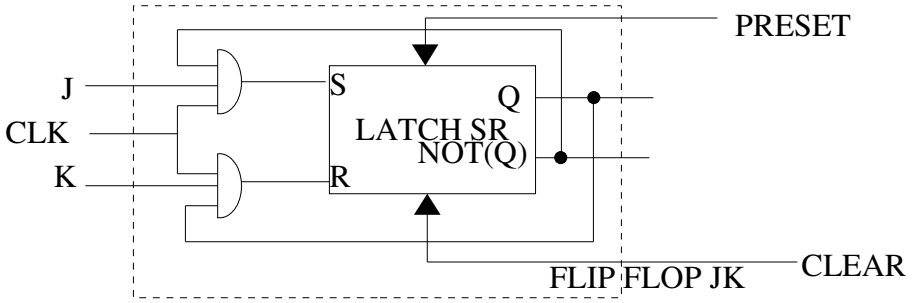


NOTA: quando il clock è alto gli ingressi devono essere mantenuti al loro valore per evitare incertezze.

Un FF che "sente" gli ingressi solo quando il clock è alto (come quello rappresentato) si dice che "funziona sui livelli"; sono usati (prevalentemente) FF che funzionano sui fronti (di salita o di discesa), cioè che "sentono" gli ingressi esclusivamente durante il passaggio a 1 (sul fronte di salita) o a zero (sul fronte di discesa) del clock. Tali FF si chiamano "edge triggered flip flop", o semplicemente "flip flop che commutano sul fronte".

Flip Flop JK

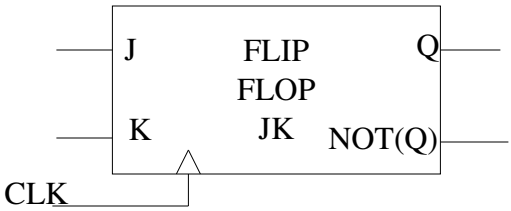
Un modello più generale di Flip Flop è il Flip Flop JK, che introduce una nuova operazione sul bit di stato e permette anche l'ingresso 11. Si realizza così:



L'equazione della transizione di stato per il LATCH SR era questa: $Q' = S + \bar{R} Q$. Nel FLIP FLOP JK si ha che $S = J \bar{Q}$ e che $R = K Q$; sostituendo si ha l'equazione della transizione di stato del FLIP FLOP JK: $Q' = J \bar{Q} + Q(\bar{K} + \bar{Q}) = J \bar{Q} + \bar{K} Q$, dalla quale si ricava la tabella di stato:

J	***K***	***Q'***	
0	0	Q	hold
0	1	0	reset
1	0	1	set
1	1	$\bar{Q}$	toggle (commutazione)

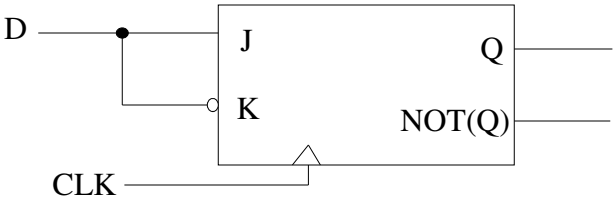
L'ingresso (J, K)=(1, 1) è permesso senza problemi perché se (J, K) sono (1, 1) S e R non possono esserlo contemporaneamente, per via dell'AND con Q e Q negato.
In generale il FLIP FLOP JK lo posso rappresentare così:



Gli ingressi CLEAR (CLR) e PRESET sono indipendenti dal CLOCK e servono rispettivamente per azzerare o mettere a 1 il contenuto del FF nel momento stesso in cui vengono asseriti, SENZA bisogno di aspettare che il CLOCK sia alto prima che vengano "sentiti". Tali ingressi ci possono essere anche negli altri FF, anche se non indicato.

Flip flop D

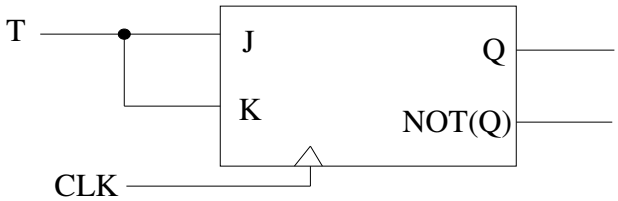
Il FLIP FLOP D (D sta per Delay, ritardo) è caratterizzato dall'avere $D=J=\bar{K}$, ed è quindi fatto così:



Siccome $D=J=\bar{K}$ la sua equazione di transizione di stato sarà $Q'=D\bar{Q}+DQ=D$, cioè il FLIP FLOP D memorizza il valore D che gli viene fornito in ingresso (ciò che viene mandato in ingresso ritorna in uscita dopo un po' di tempo). Questo è il tipo di FLOP FLOP più utilizzato. Nota bene: in questo caso la transizione di stato non dipende dallo stato attuale, ma solo dall'ingresso (le uniche due operazioni possibili sono il SET, per D=1, ed il RESET, per D=0.

Flip flop T

Il FLIP FLOP T è caratterizzato dall'avere T=J=K, quindi è realizzato così:



L'equazione di transizione di stato sarà quindi $Q'=T\bar{Q}+\bar{T}Q=T\oplus Q$. Come si vede dall'equazione di transizione di stato il comportamento del FLIP FLOP T è sempre legato allo stato precedente: se T=0 Q non varia (HOLD), se T=1 allora $Q'=\bar{Q}$ (TOGGLE, fa come il clock ma con frequenza dimezzata).

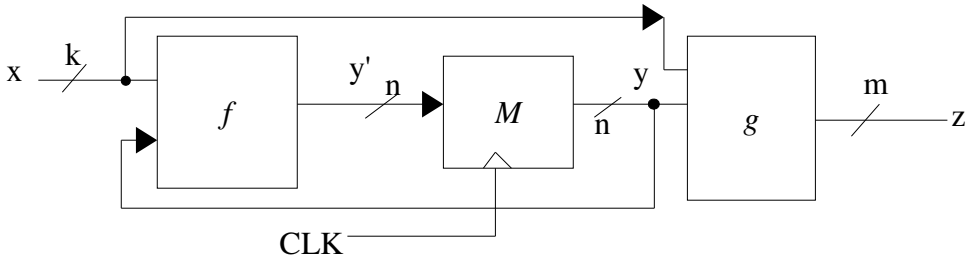
NOTA: per descrivere una rete sequenziale sono necessarie due equazioni: l'equazione di transizione di stato, che esprime come varia lo stato in funzione dello stato precedente e degli ingressi e l'equazione di uscita che esprime l'output in funzione dello stato e degli ingressi.

tabella di riepilogo dei FLIP FLOP:

Componente	***Equazione di transizione di stato***
FLIP FLOP SR	$Q' = S + \bar{R}Q$ (con SR=0) (SR: 00 - hold, 01 - reset, 10 - set)
FLIP FLOP JK	$Q' = J\bar{Q} + \bar{K}Q$ (JK: 00 - hold, 01 - reset, 10 - set, 11 - toggle)
FLIP FLOP D	$Q' = D$ (hold)
FLIP FLOP T	$Q' = T \oplus Q$ (T: 0 - hold, 1 - toggle)

In ogni caso quando CLK=0 il FLIP FLOP è in HOLD.

Vediamo ora, in generale, come è rappresentabile una macchina sequenziale sincrona:



Dove:

-x è l'ingresso, di k bit;

-y è lo stato presente, di n bit;

-y' è lo stato futuro, funzione dello stato presente e dell'ingresso (y'=*f*(x, y));

-z è l'uscita, funzione dello stato presente e dell'ingresso (z=*g*(x, y));

-*f* è un blocco combinatorio che realizza la funzione di transizione di stato

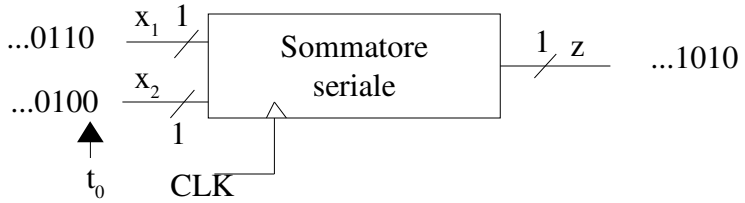
-*g* è un blocco combinatorio che realizza la funzione di uscita (nei casi visti sino ad ora *g* era la funzione z=y, cioè l'uscita era sempre uguale allo stato presente);

-M è il "cuore" della macchina sequenziale, ad esso viene applicato il segnale di temporizzazione; è un blocco di ritardo (lo stato futuro diventa presente con il successivo colpo di clock).

Nel caso in cui z dipenda sia dallo stato che dagli ingressi, diciamo che abbiamo una macchina di Mealy, mentre quando invece z dipende esclusivamente dallo stato abbiamo una macchina di Moore. Si può sempre passare da una macchina sequenziale nella forma di Moore ad una macchina nella forma di Mealy e viceversa, facendo opportune modifiche. La rappresentazione sopra è nella forma di Mealy. Per la rappresentazione della forma di Moore bisogna togliere la linea che dall'ingresso x porta al blocco *g* (ovviamente si ottiene un comportamento diverso!).

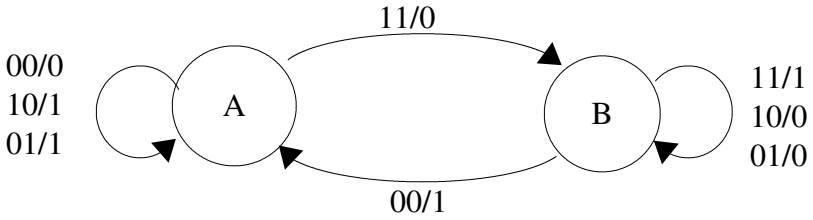
Sommatore seriale

Si vuole costruire una macchina che accetti in ingresso due parole di n bit (n non definito a priori) e che restituisca in uscita la somma delle parole ricevute in ingresso: tale macchina deve perciò avere un comportamento seriale. Questo è lo schema generale di tale macchina:



Si vede subito che la macchina ha un comportamento sequenziale per il fatto che, in questo esempio, gli ingressi al tempo t_0 e t_4 sono uguali, ma hanno uscite diverse.

Questo è il diagramma degli stati, nella forma di Mealy (l'uscita è funzione sia degli ingressi che dello stato), della macchia sopra schematizzata, indicando con A lo stato "non c'è stato riporto", con B lo stato "c'è stato riporto" e con ab/c l'ingresso (a,b) e la corrispondente uscita c:

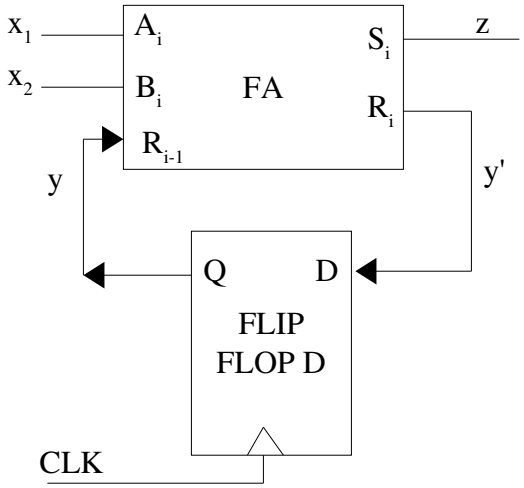


La macchina ha due stati: A, B. Per codificare tali stati serve un bit. Posso assegnare ad A il valore 0 ed a B il valore 1 o viceversa, entrambe le scelte sono lecite, anche se un tipo di codifica piuttosto che un altro tipo di codifica influenza la rete logica risultante.
Vediamo l'equivalente forma tabellare (tabella di flusso) nel caso più logico in cui A=0 e B=1:

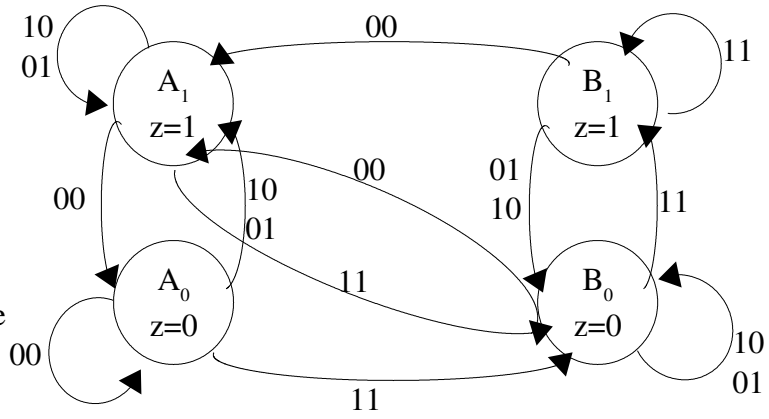
x_1	x_2	y	y'	z
0	0	A - 0	A - 0	0
0	0	B - 1	A - 0	1
0	1	A - 0	A - 0	1
0	1	B - 1	B - 1	0
1	0	A - 0	A - 0	1
1	0	B - 1	B - 1	0
1	1	A - 0	B - 1	0
1	1	B - 1	B - 1	1

Cercando la prima forma canonica per y' e per z si ritrovano le due equazioni del Full Adder, $z=x_1 \oplus x_2 \oplus y$ e $y'=x_1x_2 + x_1y+x_2y$.

Quindi il tutto si può schematizzare così:



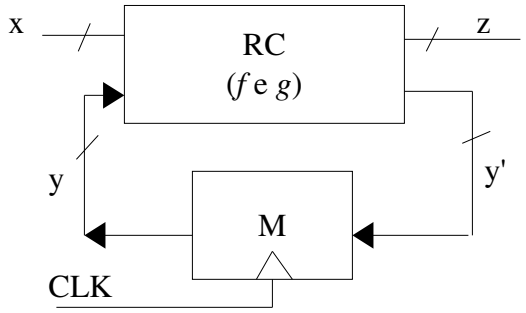
In questa schematizzazione FA include il blocco *f* (che genera il riporto in funzione degli ingressi e del riporto precedente) ed il blocco *g* (che genera l'uscita in funzione degli ingressi e del riporto, cioè dello stato). Il FLIP FLOP D è il blocco M.
Il blocco M è detto registro; lo posso sempre pensare come un aggregato di FFD.
Proviamo ora a costruire il diagramma degli stati di questo esempio nella forma di Moore, a partire dal diagramma nella forma di Mealy sopra riportato. Nella forma di Moore ad ogni stato corrisponde un'uscita, quindi molto probabilmente si avranno più "palle" rappresentanti lo stesso stato, ma con uscite associate diverse. Per passare dalla forma di Mealy a quella di Moore si devono raggruppare le transizioni (cioè le varie frecce), suddividendole in base allo stato al quale conducono e in base all'uscita che ciascuna produce (l'elemento dopo lo slash). Le frecce che conducono allo stesso stato si separano poi in base all'uscita che producono e si identificano in fine tante palle quanti sono i raggruppamenti finali, magari chiamando in modo simile le palle che corrispondono ad uno stesso stato ma differiscono per uscita (ad esempio differenziando i nomi con un pedice). Vediamo la realizzazione dell'esempio di sopra. Ho due possibili stati e ad ogni stato corrispondono due possibili uscite, quindi ho alla fine 4 palle. Una volta definite le palle basta collegarle in modo appropriato indicando come al solito sulle frecce gli ingressi che si debbono avere per effettuare la transizione, ma senza indicare l'uscita (l'uscita è indicata nella palla di arrivo).



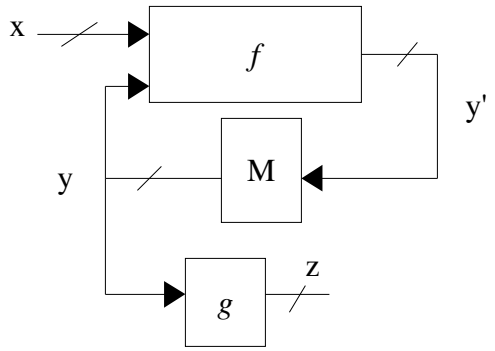
L'esempio sopra viene così come rappresentato a fianco.
Per realizzare la macchina secondo la rappresentazione di Moore ho bisogno di $\log_2(num\,stati)$ FLIP FLOP. In questo caso ne servono $\log_2 4 = 2$.

In generale, per rappresentare una qualsiasi rete sequenziale, nella versione di Mealey, posso usare una

struttura di questo tipo (allo stesso modo in cui tutte le funzioni logiche le posso realizzare con una ROM):



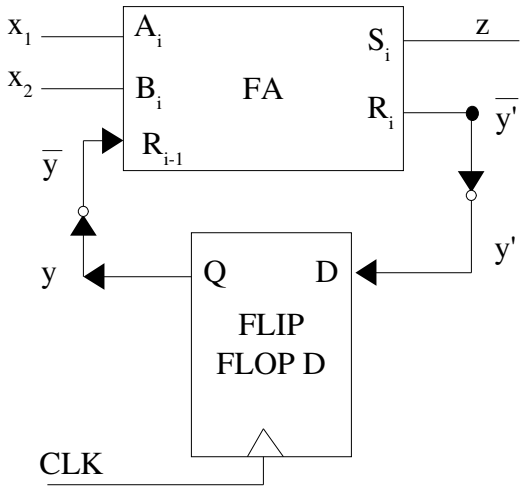
Dove RC indica Rete Combinatoria e contiene le due funzioni logiche fondamentali che caratterizzano il comportamento della rete (funzione di transizione di stato e funzione di uscita), M è un blocco standard che va scelto delle dimensioni opportune. Questo schema permette di realizzare, come detto, una funzione sequenziale qualsiasi, anche se non è efficiente (permette, anche se a volte non è necessario, di saltare da uno stato ad un qualsiasi altro stato).
In pratica RC sarà una ROM di grandi dimensioni.
La versione di Moore di una generica rete sequenziale si può così rappresentare:



Se invece scelgo l'altra codifica, quella in cui A=1 e B=0, la tabella (riordinata) è la seguente:

x_1	x_2	y	y'	z
0	0	B - 0	A - 1	0
0	0	A - 1	A - 1	1
0	1	B - 0	B - 0	1
0	1	A - 1	A - 1	0
1	0	B - 0	B - 0	1
1	0	A - 1	A - 1	0
1	1	B - 0	B - 0	0
1	1	A - 1	B - 0	1

e ottengo che $z = x_1 \oplus x_2 \oplus \bar{y}$, (che è il negato dello z ottenuto con la prima codifica, si vede dalla tabella) e anche $y' = \bar{x}_1 \bar{x}_2 \bar{y} + \bar{x}_1 \bar{x}_2 y + \bar{x}_1 x_2 y + x_1 \bar{x}_2 y = \bar{x}_1 \bar{x}_2 + y(\bar{x}_2 x_1 + x_2 \bar{x}_1) = \overline{\overline{\bar{x}_1 \bar{x}_2 + y(\bar{x}_2 x_1 + x_2 \bar{x}_1)}}$
= $\overline{(\bar{x}_1 \bar{x}_2)\overline{(y(x_1 \bar{x}_2 + \bar{x}_1 x_2))}} = \overline{(x_1 + x_2)(\bar{y} + \overline{(x_1 \bar{x}_2 + \bar{x}_1 x_2)})} = \overline{(x_1 + x_2)(\bar{y} + (\overline{x_1 \bar{x}_2})(\overline{\bar{x}_1 x_2}))}$
= $\overline{(x_1 + x_2)(\bar{y} + (\bar{x}_1 + x_2)(x_1 + \bar{x}_2))} = \overline{(x_1 + x_2)(\bar{y} + \bar{x}_1 \bar{x}_2 + x_1 x_2)} = \overline{x_1 \bar{y} + x_1 x_2 + x_2 \bar{y} + x_1 x_2}$
= $\overline{x_1 \bar{y} + x_1 x_2 + x_2 \bar{y}} \Leftrightarrow \bar{y}' = x_1 \bar{y} + x_1 x_2 + x_2 \bar{y}$.
La rappresentazione è la seguente:



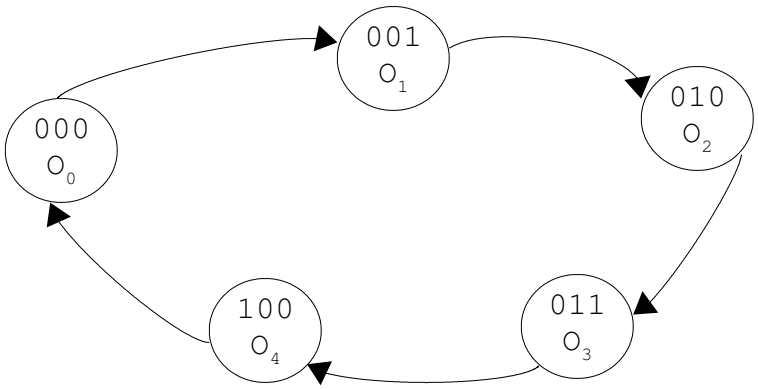
il funzionamento è lo stesso della prima soluzione, ma qui dobbiamo usare due porte NOT in più.

Descrizione in parte operativa e parte di controllo

Possiamo sempre suddividere una macchina sequenziale in due macchine più semplici, una che esegue (unità operativa) e l'altra che impartisce ordini (unità di controllo). Quella che impartisce ordini invia comandi, mentre l'altra ritorna condizioni.

Macchine autonome (o reti autonome)

Una macchina che non ha ingressi si dice macchina autonoma. In una macchina autonoma la sequenza degli stati è predeterminata. Siccome non ci sono ingressi, il passaggio da uno stato all'altro è scandito dal clock (che non è considerato un ingresso). Un esempio di macchina autonoma (versione di Moore) è questo:



Questa macchina conta, in modulo 5, il numero di colpi di clock. Ha la seguente tabella:

SP (Stato Presente, y)			***SF (Stato Futuro, y')***			***O (Output)***
0	0	0	0	0	1	O_0
0	0	1	0	1	0	O_1
0	1	0	0	1	1	O_2
0	1	1	1	0	0	O_3
1	0	0	0	0	0	O_4

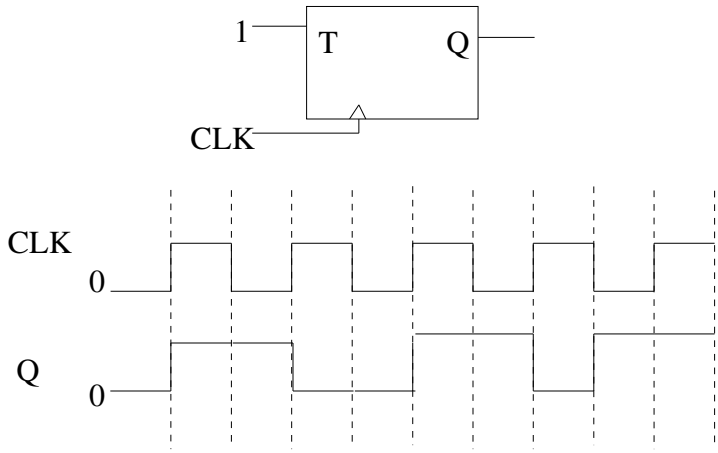
In questa macchina ho 5 stati, mi servono 3 FLIP FLOP (2^2=4, 2^3=8), alcuni stati non sono mai raggiunti. L'uscita O è funzione dello stato, O=*g*(SP). L'equazione di transizione di stato è la seguente: $y'=(y+1)_{\text{modulo 5}}$

Una macchina con le seguenti caratteristiche:
-è una rete autonoma;
-gli stati sono codificati in maniera consecutiva;
-l'output è lo stato;
si chiama REGISTRO CONTATORE.

Un caso importante di contatori sono quelli di modulo 2^n, cioè hanno n bit di stato e contano da zero a 2^n-1, sfruttando tutti gli stati possibili.

Quello sopra rappresentato si chiama CONTATORE UP.

Un esempio di contatore molto semplice, che conta solo fino a 1 (cioè 0 e 1), è un FLIP FLOP T con ingresso costantemente 1 (con tale ingresso il FLIP FLOP T funziona in modalità TOGGLE).

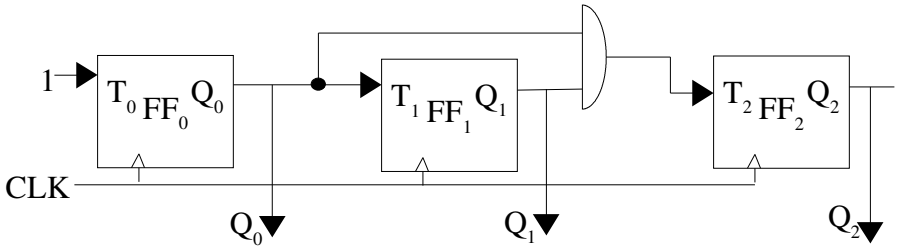


Un esempio di contatore UP fino a 7 (da zero a sette, modulo 8), è invece il seguente, che utilizza 3 bit per la descrizione dello stato:

	Q_2	***Q_1***	***Q_0***
0	0	0	0
1	0	0	1

	Q_2	Q_1	Q_0
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

In cui $Q_0' = \bar{Q}_0$, $Q_1' = Q_1 \oplus Q_0$ e $Q_2' = Q_2 \oplus (Q_1 Q_0)$, che si realizza così:



In pratica per poter commutare un FF tutti i FF precedenti devono essere a 1. In generale l'input del FF_k è l'AND degli output di tutti i FF precedenti. Questo contatore si poteva realizzare anche usando dei FFD e qualche porta XOR in più.

Si può costruire anche un contatore DOWN.
Un esempio di contatore DOWN, sempre modulo 8, ha la seguente tabella:

t	Q_2	Q_1	Q_0
0	0	0	0
1	1	1	1
2	1	1	0
3	1	0	1
4	1	0	0
5	0	1	1
6	0	1	0
7	0	0	1

Dalla tabella si vede che: $Q_0' = \bar{Q}_0$, $Q_1' = \bar{Q}_0 \oplus Q_1$, $Q_2' = \bar{Q}_1 \bar{Q}_0 \oplus Q_2$, quindi per gli ingressi dei FFT, se inizialmente $Q_2 = Q_1 = Q_0 = 0$ allora si ha $T_0 = 1$, $T_1 = \bar{Q}_0$, $T_2 = \bar{Q}_0 \bar{Q}_1$.

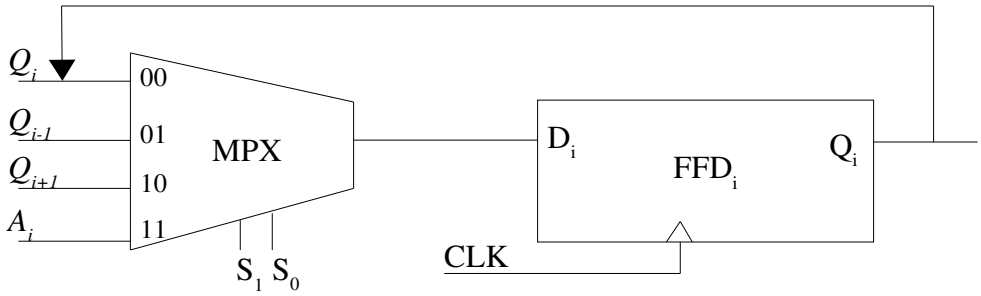
Se invece volessi costruire un contatore UP-DOWN non avrei più una rete autonoma, perché la sequenza degli stati non sarebbe più prefissata e quindi avrei bisogno di un ingresso per poter scegliere se contare UP o DOWN. In particolare, avendo due sole alternative, avrei bisogno di 1 bit di ingresso. Chiamando tale bit $U/\bar{D}$ (U=Up, D=Down), oppure semplicemente U (Up), avrei che $T_0=1$, $T_1=UQ_0+\bar{U}\,\bar{Q}_0$, $T_2=UQ_1Q_0+\bar{U}\,\bar{Q}_1\bar{Q}_0$. Nota: nella realizzazione pratica della rete utilizzerei dei multiplexer.

Registro

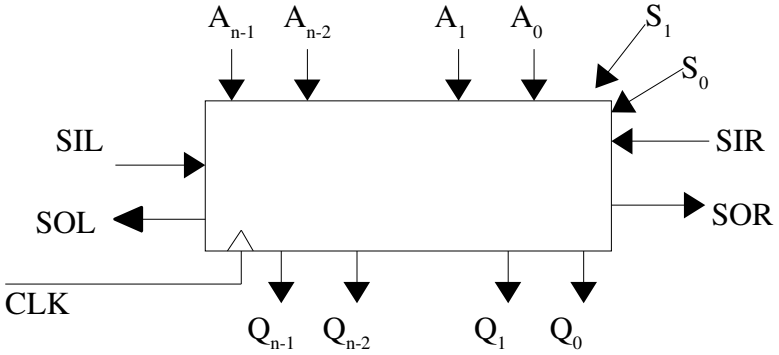
Un registro semplice è un insieme di n identici elementi di memoria, ad esempio FFD, sincronizzati tra loro tramite un unico clock. I registri semplici hanno due sole modalità: HOLD e LOAD, quindi è sufficiente una linea di controllo (1 bit) per manovrarli, che possiamo chiamare $LOAD/\overline{HOLD}$. In genere quando si fa riferimento ad un generico blocco di memoria M ci si riferisce ad un registro di questo tipo. Inoltre i registri hanno altre due linee di controllo: R_{in} ed R_{out} che abilitano/disabilitano rispettivamente gli ingressi e le uscite. Tale abilitazione/disabilitazione è realizzata con un BUFFER TRISTATE (vedi parte II).

Shift register (registro a scorrimento)

Consideriamo di avere una batteria di FFD, il cui i-esimo oggetto è così costituito:



in cui Q_{i-1} è l'uscita del FFD precedente (FFD_{i-1}), Q_{i+1} è l'uscita del FFD successivo e A_i è un ingresso che arriva dall'esterno della batteria. In pratica sono dei registri semplici collegati tra loro. Nel suo insieme lo shift register, composto da n FFD, si può rappresentare così:



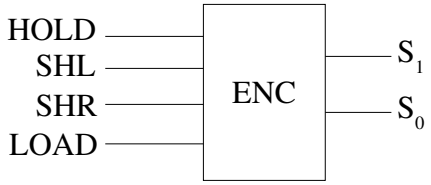
Dove A, Q, S sono quelli descritti in precedenza, mentre SIL sta per Serial Input Left, SIR sta per Serial Input Right, SOL sta per Serial Output Left, SOR sta per Serial Output Right.
Questo è il funzionamento del registro, in base alle linee di controllo S_0 e S_1:

S_1	S_0	$D_i=Q_i'$	*mode*
0	0	Q_i	HOLD
0	1	Q_{i-1}	SHIFT LEFT (SHL)
1	0	Q_{i+1}	SHIFT RIGHT (SHR)
1	1	A_i	LOAD

In caso di SHIFT LEFT è necessario introdurre un valore per il FFD che si trova più a destra: questo valore è SIR; lo stesso vale per SHIFT RIGHT e SIL. Sempre a seguito di uno SHIFT è possibile "raccogliere" il valore che è di troppo in seguito allo SHIFT. Per uno SHIFT LEFT è possibile raccogliere il valore del FFD più a sinistra: questo valore viene raccolto con SOL; lo stesso vale per SHIFT RIGHT e SOR. Nota: facendo uno SHIFT LEFT, per fare l'operazione di SHIFT LEFT (cioè moltiplicare per due) descritta diverse pagine indietro, è necessario che SIR sia a zero; mentre per fare lo SHIFT RIGHT descritto diverse pagine indietro (cioè dividere per due) è necessario che SIL sia zero se si sta trattando con numeri positivi, 1 se si sta trattando con numeri negativi espressi in complemento a due (in pratica si ricopia il bit di segno).
Il LOAD invece serve per caricare una parola intera, in modo parallelo, nel registro.
Nota: supponiamo n=4. Se a tempo zero ho $Q_3=Q_2=Q_1=Q_0=0$, SIL=1 ed eseguo degli SHIFT RIGHT ottengo che l'ingresso SIL è stato ritardato di 4 colpi di clock prima di ritornare come SOR, infatti:

SIL	*t*	Q_3	Q_2	Q_1	Q_0	*SOR*
1	0	0	0	0	0	
1	1	1	0	0	0	0
1	2	1	1	0	0	0
1	3	1	1	1	0	0
1	4	1	1	1	1	0
1	5	1	1	1	1	1

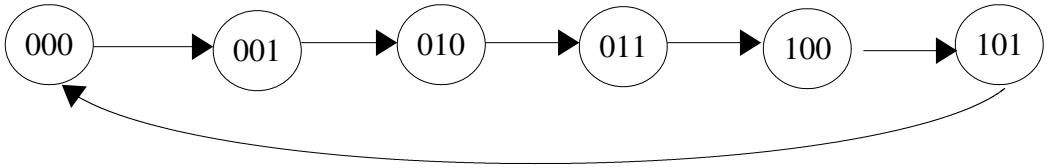
Nota: posso anche supporre di voler comunicare allo SHIFT REGISTER l'azione da eseguire su 4 linee diverse, una per ogni comando (delle quali solo una risulta asserita): in tal caso dovrei anteporre un encoder 4=>2 a S_1S_0, in questo modo:

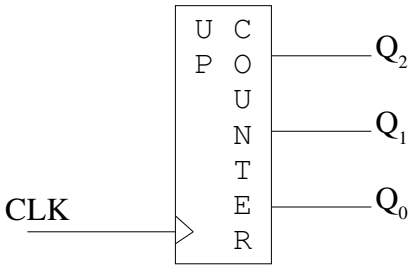


Alcuni problemi

dato un contatore UP, farlo contare DOWN

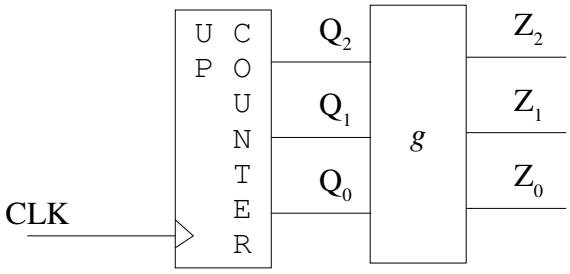
Dato un contatore UP come questo





Per farlo contare DOWN è possibile implementare una funzione di uscita Z=*g*(Q) diversa dall'identità, in modo che abbia ad esempio il seguente comportamento:

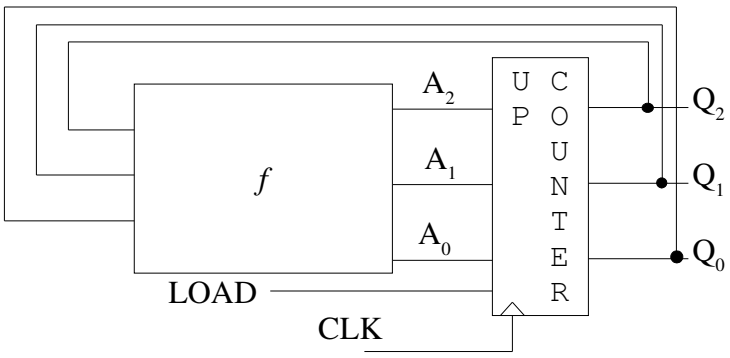
Q_2	Q_1	Q_0	Z_2	Z_1	Z_0
0	0	0	1	0	1
0	0	1	1	0	0
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	0	0	1
1	0	1	0	0	0



Posso implementare *g* ad esempio con una ROM. In ogni caso la rete non ha ingressi, quindi cicla sempre sugli stessi valori.

dato un contatore solo UP, farlo contare UP/DOWN

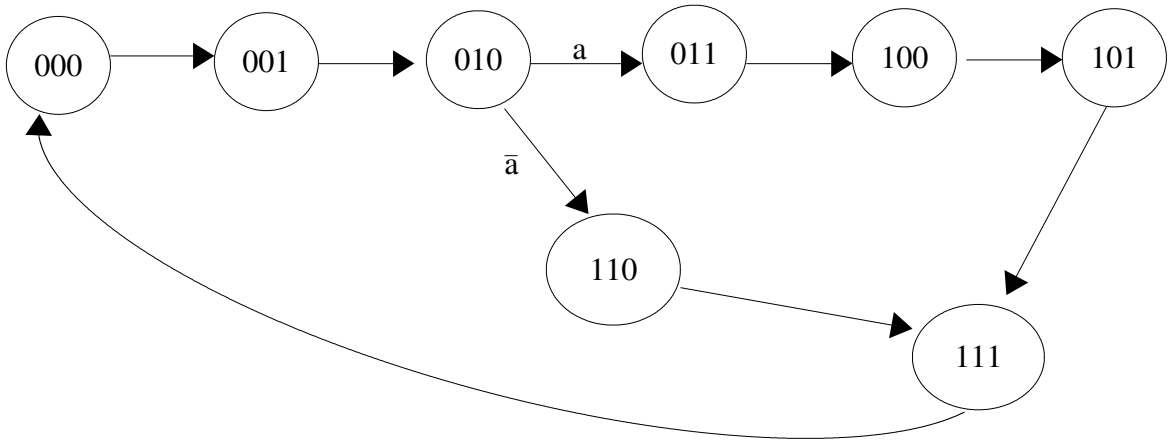
Supponendo che il contatore abbia un caricamento di tipo parallelo si può realizzare ad esempio in questo modo



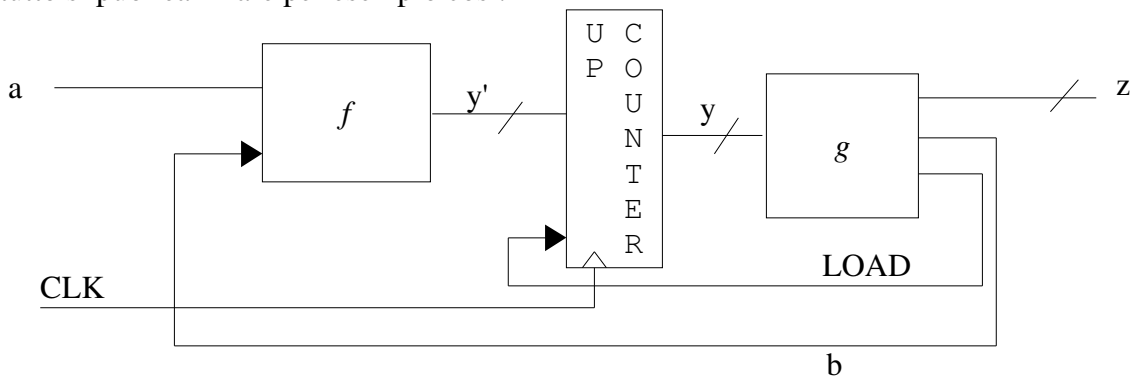
Per il comportamento UP basta non asserire LOAD, mentre per il comportamento DOWN è necessario asserire LOAD e caricare tramite *f* il valore di conteggio corrispondente allo stato futuro, cioè A=*f*(Q) =Q-1. Quando faccio il caricamento UP COUNTER si comporta come un registro.

esempio

realizzare una rete che faccia questa roba, utilizzando un UP COUNTER che abbia un caricamento parallelo:



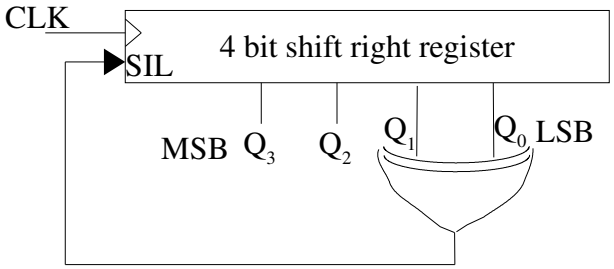
Devo aggiungere dei componenti al contatore affinché:
-quando lo stato è 010 (e solo in questo caso) devo campionare l'ingresso "a" e, in base a quello, scegliere se passare allo stato 110 o allo stato 011. Se $\bar{a}$ devo passare dallo 010 stato 110 e per fare ciò devo alterare, con un LOAD, lo stato del COUNTER, altrimenti devo passare dallo stato 010 allo stato 011, senza quindi bisogno di intervenire sul COUNTER.
-quando sono nello stato 101 devo, in ogni caso, alterare con un LOAD il conteggio per passare allo stato 111.
Il tutto si può realizzare per esempio così:



con:
-*g* che asserisce la linea LOAD solo quando y=010 oppure y=101 e pone ad esempio b=0 quando y=010 e b=1 quando y=101;
-*f* che quando b=0 e $\bar{a}$ pone y'=110, altrimenti pone y'=111.
nota: b è espresso con un solo bit perché ci sono due sole situazioni in cui c'è da fare un salto nel conteggio.

esercizio 09/09/2004

Trovare le equazioni di transizione di stato di



le equazioni sono $Q_3'=Q_1 \oplus Q_0$, $Q_2'=Q_3$, $Q_1'=Q_2$, $Q_0'=Q_1$. Poi, supponendo Q(0)=(1,1,0,1) lo stato iniziale, trovare quanto vale Q(t) con t=1, ..., 20.
Calcolando tutti gli stati si vede che a t=13 lo stato inizia a ripetersi. Essendo una macchina sequenziale senza ingressi era ovvio che avesse un comportamento periodico (in questo caso potevo avere al massimo 2^4 uscite diverse). Nota: il modo e la frequenza con la quale gli stati si ripetono dipendono sia da come è fatta la rete si dallo stato iniziale. Questi circuiti possono trovare applicazione nella crittografia perché generano sequenze pseudocasuali (se ci sono molti più bit è difficile individuare la periodicità).

Moltiplicatore

di interi positivi

Se devo moltiplicare due numeri interi positivi di k bit so per certo che il risultato sarà contenibile in 2k bit, per il fatto che $(2^k-1)(2^k-1)<2^{2k}-1$ (cioè il prodotto dei due più grandi numeri rappresentabili su k bit è minore del più grande numero rappresentabile su 2k bit).
esempio di prodotto fra due interi positivi di k=4 bit:

```
  1101x
  1011
  1101+ passo 0
 1101-+ passo 1
0000--+ passo 2
1101---  passo 3
10001111 risultato
```

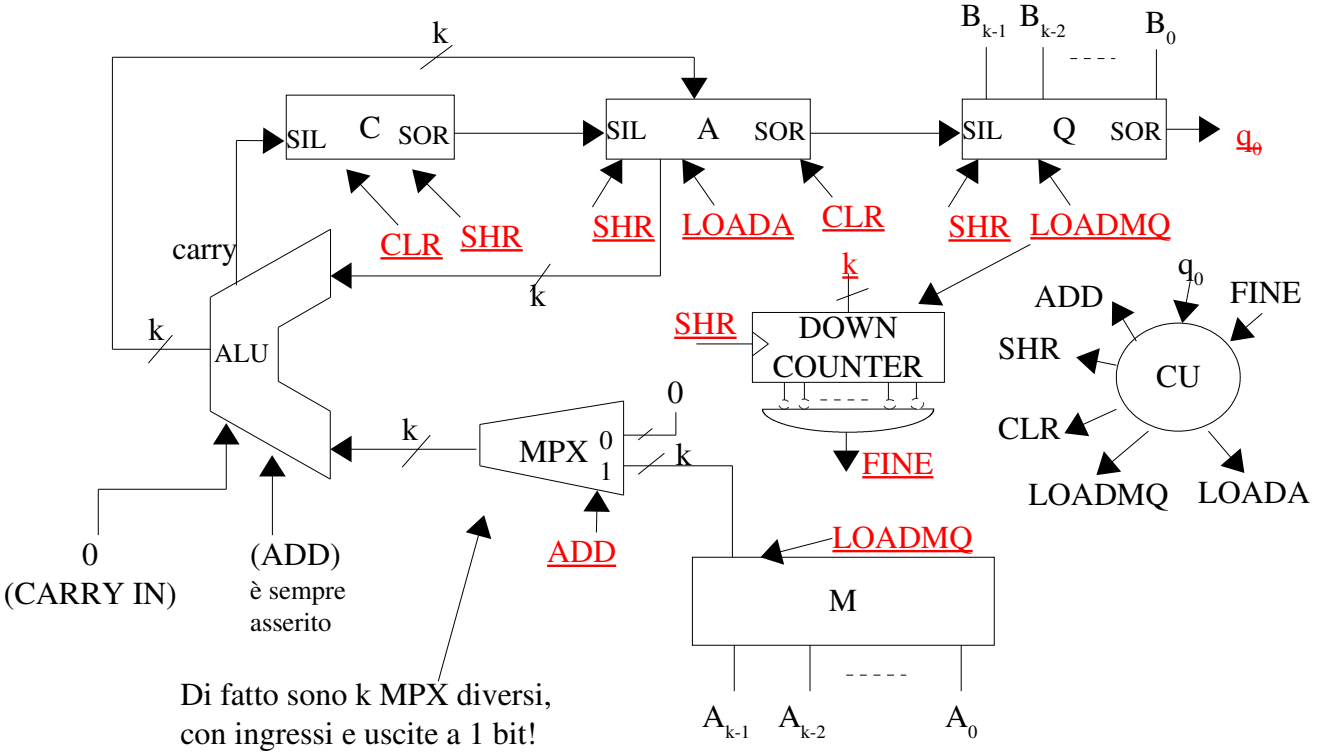
da questo esempio si nota che:
- -il prodotto può essere visto come una sequenza di k somme di singoli prodotti;
- -il singolo prodotto del passo i è zero se lo è la cifra di posto i del secondo fattore, altrimenti è uguale al primo fattore shiftato a sinistra i volte, con i=0, ..., k-1

Se invece di eseguire le somme dei singoli prodotti alla fine le eseguo via via ottengo:

```
   0000+    R-1
  1101 passo 0
 01101+    R0
  1101- passo 1
100111+    R1
  0000-- passo 2
0100111+   R2
 1101--- passo 3
10001111    R3
```

qui si nota che:
- -al passo i se la cifra di posto i del secondo fattore è 1 allora R_i è uguale alla somma fra R_{i-1} e il primo fattore shiftato a sinistra i volte, altrimenti $R_i=R_{i-1}$, con i=0, ..., k-1
- -al passo i le cifre di posto i, i-1, ..., 0 di R_i sono già corrette, con i=0, ..., k-2

Il moltiplicatore può essere ad esempio realizzato così:



con M, A, Q SHIFT REGISTER a k bit, C (C=Carry) SHIFT REGISTER a 1 bit (in pratica un FFD), CLR=Clear, SHR=Shift Right, q_0 è la cifra per la quale si sta correntemente facendo il prodotto, FINE è un segnale che indica, se asserito, che sono stati fatti k shift right (notare infatti che l'ingresso del clock del counter è collegato al segnale SHR). I segnali sottolineati sono segnali che provengono o sono destinati alla CU (comandi e condizioni).
In questo dispositivo:

-A e C vengono inizializzati a zero con CLR;

-M viene inizializzato con il primo fattore (k bit), Q viene inizializzato con il secondo fattore (k bit), DOWN COUNTER viene caricato al valore k, tutto tramite il segnale LOADMQ;

-ADD viene posto a 1 quando q_0 (che al passo zero corrisponde alla cifra di posto zero del secondo fattore) è uguale a 1, altrimenti viene posto a zero, facendo conseguentemente passare come primo ingresso della ALU il valore di M oppure zero;

-l'altro ingresso della ALU è A che inizialmente (vedi R_{-1}) vale zero;

-il risultato della somma su k bit viene memorizzato in A; il carry della somma viene memorizzato in C; siccome LSB(A) è già il bit di posizione 0 corretto del prodotto, C, A e Q vengono shiftati a destra tramite il segnale SHR (in questo momento DOWN COUNTER decrementa), facendo in modo che il valore di C si porti in MSB(A), tramite SOR di C e SIL di A, LSB(A) si porta in MSB (Q) sempre tramite SOR e SIL e q_0 assume il valore del bit di posizione 1 del secondo fattore di moltiplicazione. In Q quindi in questo momento c'è memorizzato in MSB il bit di posizione zero del prodotto (definitivo), seguito poi da tutte le cifre del secondo fattore eccetto quella di posizione zero (quindi in LSB di Q c'è la cifra di posizione 1 del secondo fattore). Notare quindi che il Q originario

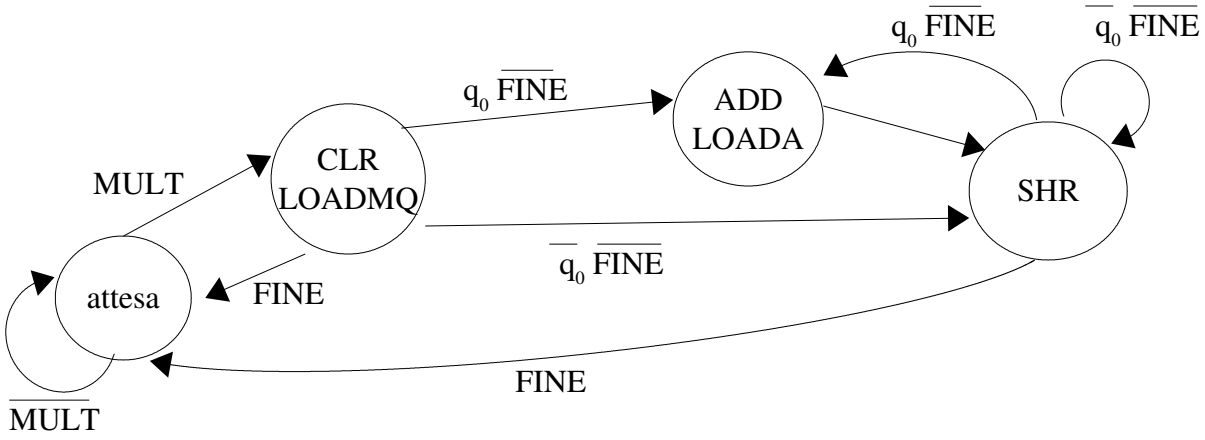
progressivamente scompare per via degli shiftamenti successivi e viene progressivamente sostituito dalla parte “definitiva” del prodotto. La sostituzione è completata al k-esimo shift right.

-il procedimento si ripete, solo che questa volta A non è zero ma contiene R_0;

-il procedimento si arresta quando sono stati eseguiti k shift right (SHR), ossia quando il segnale FINE viene asserito: in quel momento il risultato corretto del prodotto si trova in A e Q, letto da sinistra verso destra (in Q c'è la parte di parola meno significativa ed in A c'è quella più significativa).

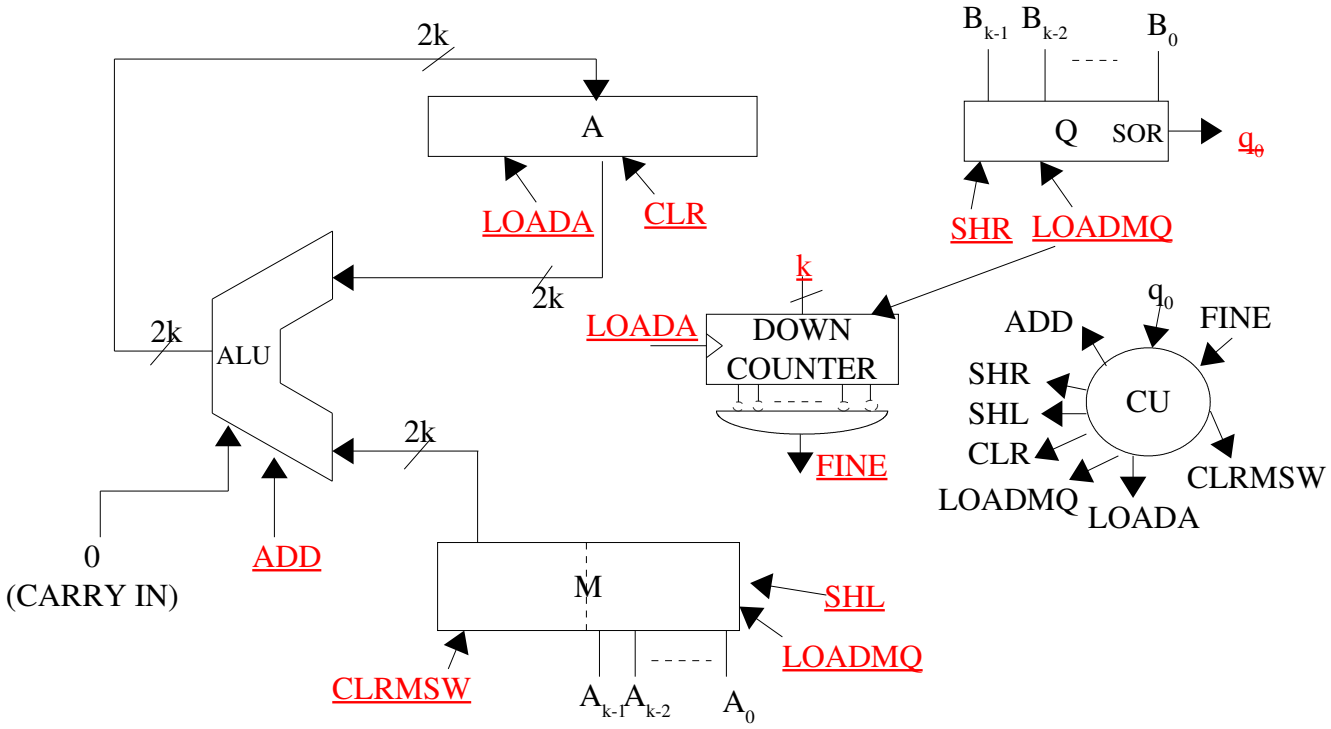
Note: nel problema della moltiplicazione fra due numeri interi non è indispensabile usare una rete sequenziale, si può fare anche in modo totalmente combinatorio; il modo sequenziale però fa risparmiare molto in termini di numero e tipologia dei componenti, impiegando però più tempo.
Il problema della moltiplicazione NON si può risolvere in modo seriale, come fatto per la somma (vedi SOMMATORE SERIALE): questo perché sarebbe necessaria una macchina con un numero infinito di stati (mentre per il sommatore seriale bastavano due soli stati, zero oppure uno, per ricordarsi se nella somma precedente c'era stato o meno il riporto).

Vediamo ora il diagramma di stato della CU:



nota: il tempo impiegato per fare il calcolo non è costante, ma varia in base alla scelta del secondo fattore; infatti se il secondo fattore è tutto uno il tempo di esecuzione è massimo, mentre è minimo se il secondo fattore è tutto zero, questo per il fatto che se q_0=0, come si vede dal diagramma, da SHR torno in SHR evitando di passare dallo stato ADD-LOADA.
In questo esempio l'unità di conteggio fa parte dell'unità operativa, mentre avrei potuto anche decidere di non effettuare alcun conteggio (e quindi di non avere il DOWN COUNTER e il segnale FINE) ma semplicemente avrei potuto disegnare il diagramma della CU scrivendo le parti del diagramma che si ripetono per un numero predeterminato di volte e tornando sullo stato di attesa esclusivamente dall'ultima operazione di SHR. In questa seconda opzione però la CU sarebbe stata più rigida.

il moltiplicatore poteva essere realizzato ad esempio anche così:



con M e A registri a 2k bit, Q registro a k bit, ALU a k bit, CLRMSW sta per CLeaR Most Significant Word (cioè azzera la parte alta del registro, formata da k bit).
Inizialmente A è posto a zero, la parte bassa (primi k bit) di M viene caricata con il primo fattore, mentre la parte alta viene posta a zero con CLRMSW, Q viene caricato con il secondo fattore. Alla fine del calcolo il risultato si trova interamente in A. Il risultato finale e corretto è raggiunto dopo che sono stati compiuti k LOADA (infatti il segnale di "clock" per il down counter è LOADA).
Questa seconda schematizzazione sarà utile perché più vicina a quella del divisore.

moltiplicatore di interi

Per fare il prodotto fra due numeri interi qualsiasi

$p = A \times B$

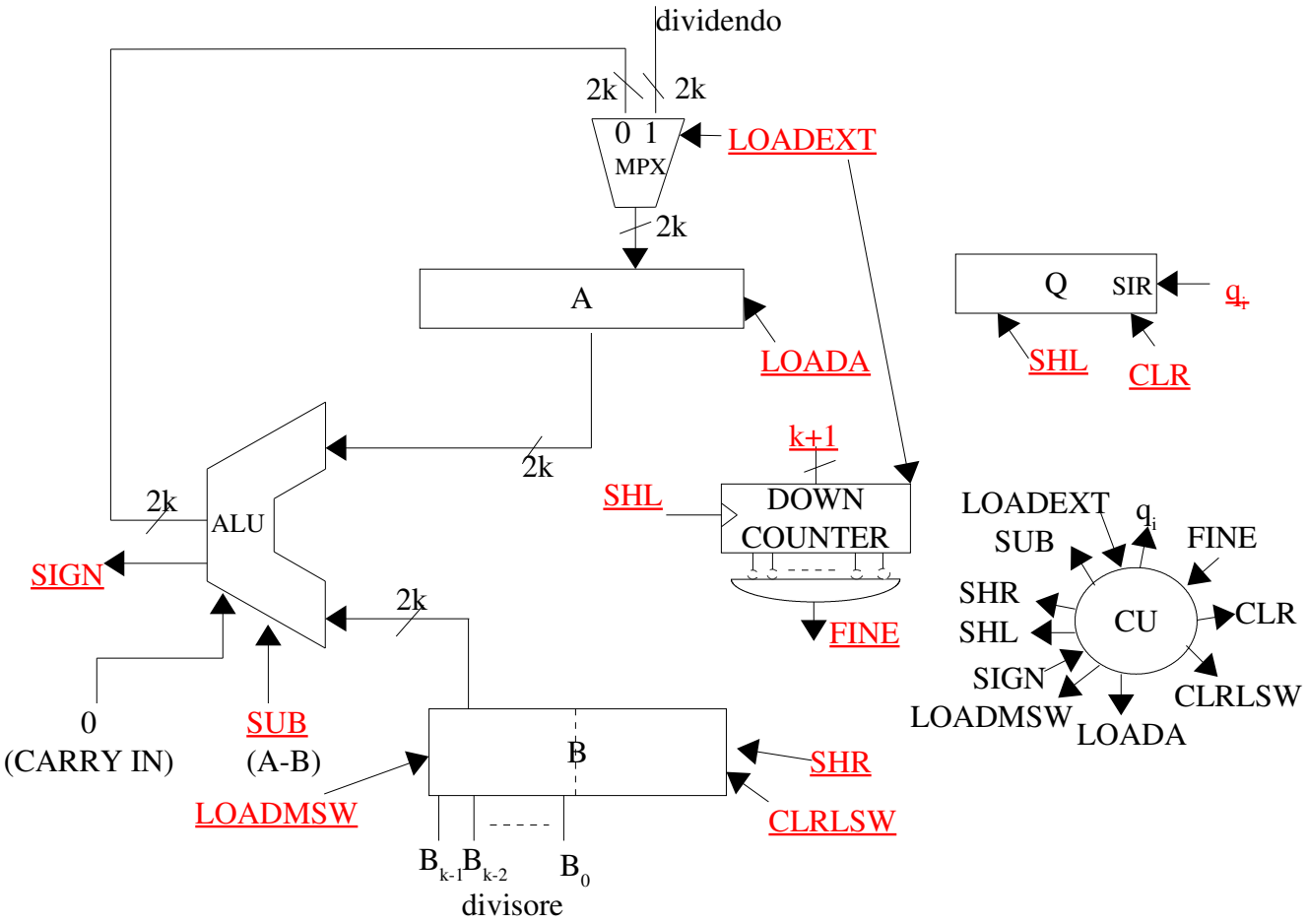
devo seguire questa regola: se B<0 allora devo complementarli a due entrambi e poi fare il prodotto normalmente, facendo attenzione a fare correttamente l'estensione di segno quando faccio le somme dei singoli prodotti.
Esempio: voglio calcolare
/*....NON TORNA!!! qualcuno ha provato a finire l'esempio che ha fatto lui? il risultato non torna, come funziona? pg 135 mio quaderno*/

Divisore di interi positivi

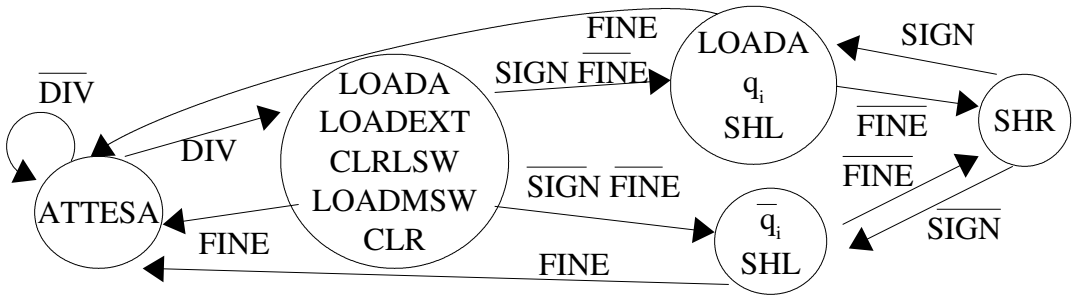
La divisione fra due numeri interi positivi
A/B=Q+R
posso vederla come una sequenza di differenze, come segue:
se k è il numero di bit di B e R, k+1 è il numero di bit di Q, 2k è il numero di bit di A, E_i è il divisore (B) shiftato a sinistra k-i volte, R_{-1} è il dividendo A e q_i è la cifra di posto i del risultato Q, allora
se $R_{i-1} - E_i >= 0$ allora $R_i = R_{i-1} - E_i$ e $q_{k-i}=1$,
altrimenti $R_i = R_{i-1}$ e $q_{k-i}=0$,
per i=0, ..., k.
Il bit q_k è un bit di controllo: se è 1 significa che non è possibile fare la divisione perché c'è overflow (cioè il risultato non entra su k bit).
Al passo i=k ottengo il resto finale R della divisione, $R=R_k$.
Esempio: 10100010/1011=1110 + 1000

k=4
R_{-1} 10100010
i=0
E_0 10110000 $R_{-1}-E_0<0$ => $q_4=0$ (no overflow) e $R_0=R_{-1}$
R_0 10100010
i=1
E_1 01011000 $R_0-E_1>=0$ => $q_3=1$ e $R_1=R_0-E_1=01001010$
R_1 01001010
i=2
E_2 00101100 $R_1-E_2>=0$ => $q_2=1$ e $R_2=R_1-E_2=00011110$
R_2 00011110
i=3
E_3 00010110 $R_2-E_3>=0$ => $q_1=1$ e $R_3=R_2-E_3=00001000$
R_3 00001000
i=4
E_4 00001011 $R_3-E_4<0$ => $q_0=0$ e $R_4=R_3$
R_4 00001000
il risultato è $q_3q_2q_1q_0$ con resto R_4, cioè 1110 con resto 1000

il tutto si può realizzare ad esempio così



dove A è un registro a 2k bit che viene inizializzato con il dividendo, B è un registro a 2k bit che viene inizializzato al divisore shiftato a sinistra k volte (quindi nella parte di sinistra di B c'è il divisore, nella parte di destra ci sono k zero), Q è un registro a k+1 bit inizializzato a zero, SIGN è il segno della differenza fra A e B.
B viene progressivamente shiftato a destra e A viene via via caricato con i resti parziali della divisione. Dopo ogni operazione di somma la CU, in base a SIGN, asserisce o meno q_i che viene poi caricato in Q. Alla fine del processo, cioè quando sono stati compiuti k+1 SHL ed A è stato caricato con l'ultimo resto (quello definitivo) si ha che in Q c'è il risultato della divisione ed in A c'è il resto.
Il diagramma di stato della CU è questo:



Nota: si vede subito che la realizzazione del divisore è simile alla seconda realizzazione del moltiplicatore, per cui è facile integrare il moltiplicatore ed il divisore in un'unica rete, fondendo le due parti operative modificando opportunamente le linee di controllo ed il diagramma di stato della CU risultante. Nel caso che fra le due CU non sia possibile condividere niente, semplicemente le due CU saranno congiunte nella fase iniziale, per poi proseguire verso la moltiplicazione o verso la divisione a seconda di un ingresso di scelta.

Teoria dell'informazione

Dieci anni dopo l'applicazione della Logica Booleana ai circuiti elettrici, Shannon, ha realizzato la Teoria dell'Informazione, che è legata all'incertezza di ricevere correttamente un messaggio digitale.

Supponiamo di avere n simboli di un alfabeto $A=\{a_1, a_2, a_3, \dots, a_n\}$, se ogni simbolo avesse la stessa probabilità di essere trasmesso avremo bisogno di **$b=int(\log_2(n))+1$** bit per rappresentare ogni simbolo dell'alfabeto A (es. per rappresentare tutte le cifre decimali abbiamo bisogno di $4=int(\log_2(10))+1$ bit per cifra).

Consideriamo un alfabeto di 10 simboli:
se trasmetto un solo simbolo ho 10 possibili combinazioni di trasmissione (una per simbolo), se invece trasmetto coppie di simboli ho 100 combinazioni (10^2), se trasmetto triplette ho 1000 (10^3) combinazioni ⇒

- per trasmettere un solo simbolo necessito mediamente di $H_1=\log_2(10)=3{,}32$ bit
- per trasmettere coppie di simboli necessito mediamente di $H_2=\log_2(100)=6{,}64$ bit
- per trasmettere triplette di simboli necessito mediamente di $H_3=\log_2(1000)=9{,}97$ bit

⇒

Realmente per trasmettere un solo simbolo ho necessità di:

$$H_T=\left\{\begin{array}{l} 4\ bit\ per\ la\ codifica\ singola \\ \frac{7}{2}=3{,}5\ per\ la\ codifica\ doppia \\ \frac{10}{3}=3{,}33\ per\ la\ codifica\ tripla \end{array}\right.$$

Questo perché, dai precedenti conti, $int(H_2)+1=7$ ma sono 7 bit su **due** simboli ⇒ servono **3,5 bit per simbolo**, ugualmente per $int(H_3)+1=10$ ma i simboli sono 3 ⇒ 3,33 bit per simbolo.
Tutto ciò significa che conviene trasmettere i simboli non singolarmente ma a triplette, e questi valori rappresentano il limite minimo di bit per trasmettere simboli equiprobabili senza perdere informazioni.

Generalmente però i simboli di un qualunque alfabeto non hanno la stessa probabilità di essere trasmessi ⇒ è necessario ottimizzare questo metodo per ridurre al minimo il numero di bit da scambiare, e quindi anche le possibilità di errori.
Per ottenere questo scopo ci affidiamo al **Teorema Fondamentale della Codificazione,** il quale si basa sul codificare simboli più probabili con un numero minore di bit, e simboli meno probabili con un numero maggiore di bit:

L'Entropia della Sorgente $H=\sum_{i=1}^{n} p_i \log_2\left(\frac{1}{p_i}\right)$ (p_i = probabilità del simbolo i-esimo di essere trasmesso) rappresenta il numero medio dei bit necessari per trasmettere un messaggio con simboli

non equiprobabili, mentre $H_i = p_i \log_2\left(\frac{1}{p_i}\right)$ è il numero di bit necessari per trasmettere un simbolo che ha probabilità p_i di essere trasmesso.

Caso limite: se io trasmetto la stringa "BA", questa ha probabilità 1 di essere trasmessa $\Rightarrow H_{BA} = p_{BA} \log_2\left(\frac{1}{p_{BA}}\right) = 1 \cdot \log_2(1) = 0 \Rightarrow$ non servono simboli, se invece voglio calcolare i bit necessari per codificare tutti gli altri simboli non trasmessi ho necessità di $H_{\overline{BA}} = p_{\overline{BA}} \log_2\left(\frac{1}{p_{\overline{BA}}}\right) = 0 \cdot \log_2\left(\frac{1}{0}\right) = o \cdot \infty = \infty$ bit $\Rightarrow$ non li posso trasmettere.

/*
Poi il prof ha parlato della Ridondanza, ma ho solo la formula ma non so cosa sia, cmq la aggiungo

Ridondanza:

$R = 1 - \frac{H_1}{H_0}$ dove H_0 è il numero massimo di bit necessari.

*/.

GNU Free Documentation License

GNU Free Documentation License
Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been

arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated

version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the

original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of thisLicense "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

    Copyright (c) YEAR YOUR NAME.
    Permission is granted to copy, distribute and/or modify this document
    under the terms of the GNU Free Documentation License, Version 1.2
    or any later version published by the Free Software Foundation;
    with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
    A copy of the license is included in the section entitled "GNU
    Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

    with the Invariant Sections being LIST THEIR TITLES, with the
    Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.