

Codici numerici

Rappresentazione DECIMALE, araba/indiana

$27 = 2 \cdot 10^1 + 7 \cdot 10^0$ rappresentazione A LUNGHEZZA VARIABILE

Numerazione ROMANA

I	V	X	L	C	D	M
1	5	10	50	100	500	1000

Scriviamo 2021:

MMXXI anche questa rappresentazione è a lunghezza variabile (lo spazio vuoto è significativo)

Altro es:

MCMLXX 1970

(per usare meno CARATTERI, nella numerazione romana si può usare anche l'operazione di somma oltre a quella di differenza: voglio scrivere 9 → non scrivo VIIII, scrivo IX, cioè 10-1)

Criteri per capire qual è il miglior tipo di codice:

- Sinteticità
- Algoritmi di calcolo (+,-,·,/) applicabili subito sulla rappresentazione

In base 10 (0,1,2,3,4,5,6,7,8,9) 17 lo scrivo 17

In base 16 (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F) 17 lo scrivo 11

In base 8 (0,1,2,3,4,5,6,7) 17 lo scrivo 21

In base 2 (0,1) 17 lo scrivo 10001

$17/2=8$ resto 1

$8/2=4$ resto 0

$4/2=2$ resto 0

$2/2=1$ resto 0

$1/0=0$ resto 1

10001

Dal punto di vista dell'implementazione hardware la scelta BINARIA è quella più conveniente

Come si passa da una base all'altra?

Es. da base 2 a base 8

10101110010

Prima di tutto decodifichiamo il numero in base 2 (non so che cosa ho scritto)

Quindi

1 0 1 0 1 1 1 0 0 1 0

$2^{10} 2^9 2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$

1024+

0+

256+

0+

64+

32+

16+

0+

0+

2+

0

=1394 IN BASE 10

Ovviamente gli 0 si possono omettere (si considerano direttamente solo le potenze di due in corrispondenza dei caratteri 1; quelle in corrispondenza degli 0 sono come “disattivate”)

Per passare da 10101110010 (in base 2) alla scrittura in base 8 non ho bisogno di decodificare prima (cioè di fare tutto il calcolo che mi porta a 1394).

L'unica cosa da fare è suddividere la rappresentazione in blocchi di 3 caratteri/cifre binarie/bit (perché 8 è 2^3) e leggerli, separatamente.

| 1 0 | 1 0 1 | 110 | 010

$$\cancel{2^2} + 2^1 + \cancel{2^0} \quad 2^2 + \cancel{2^1} + 2^0 \quad 2^2 + 2^1 + \cancel{2^0} \quad \cancel{2^2} + 2^1 + \cancel{2^0}$$

2 5 6 2

(sarebbe 2^1) (sarebbe $2^0 + 2^2$) (sarebbe $2^1 + 2^2$) (sarebbe 2^1)

Se voglio invece trovare la rappresentazione in base 16 di 10101110010

Basta dividere in blocchi di 4 bit

Quindi

| 101 | 0111 | 0010

$$1+4=5 \quad 1+2+4=7 \quad 2$$

DECODIFICA DA BASE 12(0,1,2,3,4,5,6,7,8,9,A,B)

Numero A1B.

A

1

B

Cifra delle dozzine al quadrato (12^2)

Cifra delle dozzine (12^1)

Cifra delle unità (12^0)

$$= \begin{array}{r} 10 \cdot 12^2 \\ + \end{array}$$

$$1 \cdot 12^1$$

$$+ \begin{array}{r} 11 \cdot 12^0 \\ + \end{array}$$

$$= \begin{array}{r} 10 \cdot 144 \\ + \end{array}$$

$$12$$

$$+ \begin{array}{r} 11 \cdot 1 \\ + \end{array}$$

$$= \begin{array}{r} 1440 \\ + \end{array}$$

$$12$$

$$+ \begin{array}{r} 11 \\ + \end{array}$$

$$= 1463$$

CODIFICHiamo IN BASE 12 UN NUMERO CHE è IN BASE 10

AD ESEMPIO 1234

Lo dividiamo per 12

$1234/12 = 102$ con resto 10 (la cifra del resto è quella delle unità)

Quindi scriviamo già

A

Ora si divide il 102 per 12 \rightarrow 8 con resto 6 (la cifra del resto è quella delle dozzine e quella del risultato: 8, siccome è minore di 12, è quella delle dozzine al quadrato). Quindi scriviamo 86°

Passiamo ora alla rappresentazione a lunghezza FISSA

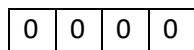
Fissiamo a priori il numero di cifre che vogliamo usare

Ad esempio, in base 11 con 4 CIFRE

Disegniamo una "griglia" del tipo 

E in ognuno di questi quadratini scriviamo sempre una cifra a base 11

Ad esempio, per scrivere "0" dobbiamo scrivere 4 volte 0



Mentre nelle rappresentazioni/codici a lunghezza variabile ci bastava scrivere 0

(vediamo come cambia la situazione in termini di sinteticità)

Quale è il numero più grande che possiamo scrivere, ricordandoci di essere in base 11? È $11^4 - 1$

Quindi abbiamo determinato un intervallo di valori possibili:

$$0 \leq \boxed{\quad \quad \quad \quad} \leq 11^4 - 1$$

Vantaggi di questa rappresentazione: non ho bisogno dello SPAZIO VUOTO (di prima)

Poiché ho anteriormente FISSATO che i numeri sono di 4 cifre, quindi se, per esempio, trovassi



Sarebbe chiaro che, dopo le prime 4 cifre inizia un altro numero, posso scriverlo attaccato.

Introduciamo la scrittura BINARIA (base 2)

Quante cifre (quadratini/bit) usiamo però? Dipende

Se io prendessi 8 cifre binarie, potrei rappresentare tutti i numeri compresi tra 0 e 255 (perché ho 8 caselle e 2^8 fa 256)

$$0 \leq \boxed{\quad \quad \quad \quad \quad \quad \quad \quad \quad} \leq 255$$

A cosa potrebbe servirmi dal punto di vista pratico?

Per esempio, per la rappresentazione dei SUONI DIGITALIZZATI: se io ho una scheda audio, vengono presi dei campioni e questi campioni sono rappresentazioni di numeri su 8 bit

Un qualcosa di simile, ma a 7 bit, è il codice ASCII

ESSO ELENCA I CARATTERI CHE IO VOGLIO POTER STAMPARE SUL TERMINALE

Ci sono le 10 cifre decimali (0,1,2,3,4,5,6,7,8,9), le lettere MAIUSCOLE dell'alfabeto (A,B,C,...,Z), le lettere minuscole dell'alfabeto (a, b, c,...,z), i segni di punteggiatura, lo spazio, il backslash, l'asterisco, ecc. ecc.

E si arriva a 128 CARATTERI DIVERSI (128 perché siamo a 7 bit, in base 2: 2^7 fa 128)

Il primo simbolo viene codificato con 0, il secondo con 1, il terzo con 2, così via fino all'ultimo che sarà codificato con 127

Quindi ogni volta che ho un blocco da 7 bit, in ASCII, ho un NUMERO d'ORDINE, che rappresenta un carattere nella tabella ASCII

Cosa parallela a IP: una variabile int tipicamente è rappresentata su 32 bit

Nei sistemi di calcolo, se volgiamo "risparmiare" possiamo usare la rappresentazione a 32 bit, che va bene per fare molte cose, se non ci basta c'è quella "estesa" a 64bit, che va ben oltre i soliti limiti pratici, raramente avremmo bisogno di rappresentare numeri più grandi di 2^{64}

Se fosse il caso si farebbe ricorso ad una rappresentazione a lunghezza variabile.

Qual è il vantaggio di usare, nei sistemi di calcolo, una rappresentazione a lunghezza fissa?

Quello di poter IMPLEMENTARE gli algoritmi di somma, sottrazione, moltiplicazione e divisione in un modo PREDEFINITO per quella lunghezza di rappresentazione (ad esempio 64bit) e quindi INDIPENDENTE dal valore.

CODICI A LUNGHEZZA FISSA

ASCII, mi permette di rappresentare una serie di caratteri stampabili sul terminale più un'altra serie di caratteri: i caratteri di controllo, che non sono immediatamente visibili sul terminale

Tra i caratteri stampabili, quelli usati con maggior frequenza sono le dieci cifre decimali

"0", "1", "2", "3", "4", "5", "6", "7", "8", "9" codificati rispettivamente con le rappresentazioni numeriche che vanno da 48 a 57 (quindi 48 è il carattere "0", 49 è il carattere "1", così fino a 57 che è il carattere "9")

Ci sono poi i vari segni di punteggiatura, i segni di maggiore e minore ecc

Ci sono le lettere dell'alfabeto, partendo da quelle maiuscole

"A" è codificato da 65, "B" da 66 e così via fino a "Z" che è codificato dalla rappresentazione numerica 90, ci sono altri simboli come il backslash, accenti ecc ecc

Poi ci sono le lettere minuscole, codificate con numeri a partire da 97 ("a") fino a 122, che codifica "z"

E poi qualche altro carattere fino ad arrivare al numero 128, c'è poi la versione estesa di ASCII che copre fino a 256 rappresentazioni numeriche

Come possiamo usare questo codice per rappresentare altre informazioni in un sistema di calcolo, che non siano necessariamente un singolo carattere?

Esempio

Supponiamo di voler memorizzare la targa italiana di un'automobile (2 lettere 3 cifre 2 lettere)

Possiamo usare 7 caratteri ASCII per rappresentare un numero di targa qualsiasi.

La rappresentazione di un singolo CARATTERE richiede 7 bit (siamo in ASCII), la rappresentazione di 7 caratteri richiederà 49 bit ($7 \times 7 = 49$)

RIDONDANZA

$$R = \frac{\text{numero totale di combinazioni}}{\text{numero di combinazioni usate}}$$

Sappiamo che non tutti i caratteri ASCII sono visualizzabili sullo schermo ("stampabili"), ci sono i cosiddetti caratteri di controllo per esempio.

0 NULL (comando che serve, per esempio, in C++ per terminare una stringa)

8 BACKSPACE (per far ritornare indietro il cursore di una posizione)

27 ESCAPE (per esempio per terminare un'applicazione e iniziare un'altra)

12 FORM FEED

I caratteri di controllo in totale sono 32. I restanti caratteri quindi sono tutti stampabili.

Quale sarebbe il coefficiente di Ridondanza del codice ASCII se noi volessimo utilizzarlo per rappresentare solo il testo visibile sullo schermo? (quindi consideriamo solo i cosiddetti caratteri "stampabili")

$$R = \frac{2^7}{128 - 32} = \frac{128}{96} = 1.\overline{3}$$

Il coefficiente di ridondanza è sempre maggiore o uguale a 1

Se volessimo codificare le targhe automobilistiche italiane, avremmo il numero totale di combinazioni che vale 2^{49} (poiché una targa è formata da 7 caratteri e quindi da 49 bit, bisogna considerare tutte le combinazioni di caratteri possibili e così si arriva al coefficiente di 2^{49})

Per quanto riguarda il numero di combinazioni utilizzabili effettivamente, siccome in una targa abbiamo SEMPRE 2 lettere alfabetiche maiuscole, 3 cifre decimali, altre 2 lettere alfabetiche maiuscole, quindi tra tutti i caratteri ASCII userò sempre e solo quelli dal numero 48 ('0') al numero 57 ('9') più quelli dal numero 65 ('A') al numero 90 ('Z').

Sono quindi 10+26 caratteri.

Consideriamo tutte le combinazioni possibili

Partiamo con le combinazioni possibili delle cifre decimali (in una targa ci sono 3 cifre, essendo 10 tutte le cifre decimali bisogna fare 10^3 per trovare tutte le combinazioni possibili)

10^3 lo devo moltiplicare per le combinazioni delle lettere alfabetiche maiuscole (26^4 , perché le lettere sono 26 ed in una targa ci stanno 4 lettere, 2 all' inizio e due alla fine)

Troviamo quindi $10^3 * 26^4 = 456976000$

Ritornando al coefficiente di ridondanza:

$$\frac{2^{49}}{10^3 * 26^4} = \frac{2^9 * 2^{40}}{10^3 * 26^4} = \frac{512 * 2^{40} \sim 512 * 10^{12}}{10^3 * 26^4 = 456976 * 10^3} = \sim \frac{512 * 10^{12}}{456976000} = \sim 10^6$$

Stiamo sprecando tanti bit.

Con un coefficiente di ridondanza di 10^6 possiamo "togliere" circa 19 quasi 20 bit

(poiché facciamo il logaritmo in base 2 di $10^6 = 19.9$ e ci accorgiamo che tutta l'informazione può stare tranquillamente in 30 bit, non ne servono 49)

Con un coefficiente di ridondanza maggiore di 2 possiamo togliere 1 bit.

Se abbiamo tolto 20 bit abbiamo un coefficiente di Ridondanza molto minore.

Nello specifico se facciamo il logaritmo in base 2 del numeratore ($\log_2 2^{49} = 49$) e il logaritmo in base 2 del denominatore ($\log_2 (10^3 * 26^4) = 28.7$) e facciamo il rapporto tra di essi il coefficiente risulta 1.7. È un coefficiente di Ridondanza minore di 2, si dice MINIMALE.

Un codice si dice minimale quando ha coefficiente di ridondanza compreso tra 1 e 2

$$1 \leq R < 2$$

Come abbiamo già visto l'informazione che dobbiamo rappresentare (2 lettere 3 cifre decimali 2 lettere) è 10^3 (che rappresenta il numero totale di combinazioni POSSIBILI per quanto riguarda le cifre decimali) **moltiplicato per 26^4** (che rappresenta il numero totale delle combinazioni possibili per quanto riguarda le lettere alfabetiche, perché sono 4 lettere da mettere e le lettere nell'alfabeto sono 26)

Più semplicemente arrotondando per eccesso il logaritmo in base 2 della moltiplicazione ($10^3 * 26^4$) troviamo il numero MINIMO di bit che ci serve per codificare la tale informazione (29 bit (28,7 arrotondati per eccesso) come visto in precedenza)

Consideriamo la rappresentazione di una DATA

GIORNOMESEANNO

Es: 05102021

(se sappiamo già in precedenza che stiamo rappresentando una DATA, facciamo a meno di usare i caratteri '/', risparmiamo bit)

Valutiamo il coefficiente di Ridondanza

Sicuramente utilizziamo solo i caratteri ASCII dal numero 48 al numero 57 (da '0' a '9')

Per ogni data abbiamo 8 caratteri ASCII, ogni carattere richiede 7 bit. $7 \times 8 = 56$. Il numero di combinazioni totali è 2^{56}

Per quanto riguarda il numero di combinazioni possibili per la rappresentazione che ci interessa attuare (data)

(31 è il numero massimo di giorni che ci possono essere in un mese, 12 è il numero di mesi che ci sono in un anno, 10^4 perché stiamo usando 4 cifre DECIMALI, che ci permettono di rappresentare tutti gli anni dall'anno 0 all'anno 9999)

$$R = \frac{2^{56}}{31 \times 12 \times 10^4} = \frac{2^{56}}{372 \times 10^4} = \frac{2^6 \times 2^{50}}{372 \times 10^4} = \frac{64 \times 2^{50}}{3,72 \times 10^6} \cong \frac{64 \times 10^{15}}{3,72 \times 10^6} \cong \frac{6,4 \times 10^{16}}{3,72 \times 10^6} \cong 10^{10}$$

(non siamo sicuri che sia precisamente 10^{10} ma è sopra il 10^9 e ci va molto vicino, quindi nel dubbio arrotondiamo per eccesso)

Facendo il logaritmo in base 2 di 10^{10} ($\log_2 10^{10} = 33,2$) e arrotondando per eccesso ci accorgiamo che possiamo risparmiare 34 bit, usandone quindi 22 invece che 56

In che modo potrei fare quindi per usare meno bit, riducendo lo spreco al minimo?

- Per prima cosa per rappresentare i giorni di un mese (valore massimo da 0 a 31) posso usare SOLO 5 bit, non me ne servono di più perché

In codice binario con 5 bit, il numero più piccolo che posso rappresentare è 0 (00000)

0	0	0	0	0
---	---	---	---	---

Mentre il numero più grande che posso rappresentare è

1	1	1	1	1
2^4	2^3	2^2	2^1	2^0

Il valore che corrisponde a questa rappresentazione è

$$2^0 + 2^1 + 2^2 + 2^3 + 2^4 = 1 + 2 + 4 + 8 + 16 = 31$$

- Per rappresentare i mesi di un anno (12) mi servono SOLO 4 bit, non me ne servono di più, perché con 4 bit posso già scrivere

1	1	1	1
2^3	2^2	2^1	2^0

Per cui arrivo a un valore decimale di $8+4+2+1=15$

Quindi ne ho già abbastanza

- Per gli stessi motivi per rappresentare i valori che gli anni possono assumere (da 0 a 9999) mi bastano 14 bit per coprire tale richiesta.

5bit+4bit+14bit= 23 bit

Giorni mesi anni

C'è 1 bit "in più" rispetto ai 22 calcolati in precedenza poiché codifichiamo i giorni separatamente giorni mesi e anni (quindi valori tra 0 e 31, valori tra 0 e 12 e valori tra 0 e 9999)

AdC continuazione appunti

Esercizio

Se volessimo codificare il codice fiscale di una persona

(L: lettera

N: numeri)

LLLLLLNNLNLLNNNNL

Codice a lunghezza fissa usando 16 caratteri ASCII

$$\frac{2^{112}}{10^7 * 26^9} \cong \frac{4 * 10^{33}}{5 * 10^{12} * 10^7} \cong 10^{14}$$

Stiamo sprecando tanti bit

Il numero di bit ragionevoli da usare, in modo che il codice sia minimale, sarebbe

$$\log_2(50 * 10^{18}) \cong 50 * 2^{60} \cong 2^{66} = \log_2(2^{66}) \cong 66$$

Possiamo risparmiare 66 bit, circa metà.

Perché diciamo sempre che, se abbiamo un coeff. Di Ridondanza maggiore o uguale a 2 possiamo togliere un bit?

Perché $2^{10} \cong 10^3$

(2^{10} è 1024; 10^3 è 1000)

10^3 equivale a 10 bit. (Poiché equivale quasi a 2^{10})

Per adesso abbiamo capito che è uno spreco ma ci sono dei casi in cui avere un coefficiente di ridondanza grande, maggiore di 2, può portarci dei vantaggi.

In che casi ci può essere utile un CODICE RIDONDANTE (NON minimale)?

Supponiamo di voler fare una codifica binaria ma di sbagliarci.

Esempio, dobbiamo scrivere

1010101010

Ma scriviamo, per sbaglio

1011001010

C'è la possibilità di riconoscere la presenza di un errore?

Se il coefficiente di Ridondanza è minore di 2 non ho la possibilità di ricercare la presenza di eventuali errori.

Per confrontare sequenze binarie:

DISTANZA DI HAMMING

Si fa il confronto "bit a bit" delle due sequenze, se sono uguali hanno distanza 0.

Se abbiamo un bit diverso (es.: Da una parte abbiamo 1 e dall'altra 0) si dice che ciò induce una distanza di 1.

Contando tutti i bit e sommando tutte le "distanze di 1" si arriva alla distanza totale di HAMMING.

Prendendo l'esempio di prima

$\begin{matrix} +1 & +1 \end{matrix}$
1010101010

1011001010

Ho una distanza di HAMMING totale di 2.

Ci dice "quanto" sono diverse. Son diverse di 2 bit. Quindi ci sono stati 2 errori.

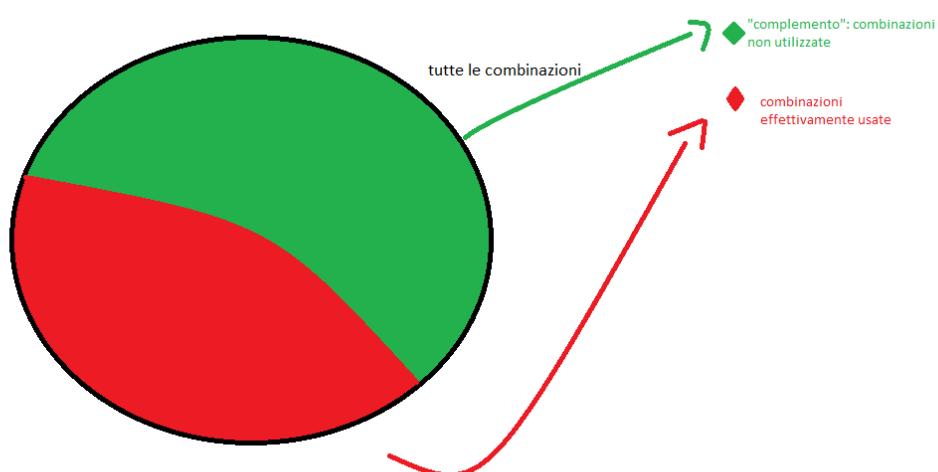
Se stiamo confrontando 2 sequenze di 10 bit la distanza di Hamming può variare da 0 a 10, se c'è distanza 0 significa che le due sequenze sono uguali, se c'è distanza 1 sono diverse solo in 1 dei 10 bit, se c'è distanza di Hamming= 10 significa che, in una sequenza, tutti i bit sono invertiti rispetto all'altra.

Ad esempio

1010101010

0101010101

Proviamo ad immaginarcici l'insieme di tutti i valori possibilmente rappresentabili su 10 bit (tutte le combinazioni): abbiamo 2^{10} combinazioni possibili (1024).



Ipotizziamo di star usando un codice RIDONDANTE.

Le combinazioni che effettivamente usiamo sono di meno di tutte le combinazioni possibili.

Se noi riceviamo una combinazione che l'utente non stava usando (quindi fa parte del sottoinsieme del complemento, non delle combinazioni effettivamente usate) siamo sicuri che c'è stato almeno un errore.

Se ci arriva una tra le combinazioni effettivamente usate non possiamo sapere se c'è stato o meno un errore, potrebbe essere che ci abbiano mandato una combinazione e si sia risolta in un'altra, che però è un'altra di quelle "consentite", (potrebbe esserci stato un errore che ci ha fatto passare da una combinazione all'altra, ma comunque nel sottoinsieme delle combinazioni "**consentite**", oppure ci potrebbe non essere stato nessun errore e noi potremmo aver ricevuto il valore corretto, quello che il mittente intendeva dall'inizio) quindi non possiamo sapere se c'è stato un errore in fase di "transizione", può essere sì ma può essere no".

Da questo punto di vista il fatto che ci sia una ridondanza maggiore di 1 ci può essere utile. Se utilizzassimo un codice con ridondanza 1 non potremmo MAI accorgerci di alcun tipo di errore.

All'aumentare del coefficiente di ridondanza aumenta la mia possibilità di trovare dei valori che non possono essere stati utilizzati dal mittente, e quindi sono **SICURAMENTE SBAGLIATI**.

Se il coefficiente di Ridondanza cresce e supera il valore 2 può aiutarci ancora meglio a riconoscere la presenza di un errore

ESISTE UN ALGORITMO CHE CI PERMETTE DI AUMENTARE IL COEFF DI RIDONDANZA IN MODO DA POTER RICONOSCERE SEMPRE LA PRESENZA DI UN ERRORE (se si tratta di un errore che cambia la combinazione da una "**consentita**" a una "**non consentita**").

Supponiamo di avere il nostro codice a 10 bit con coefficiente di ridondanza di 1,33

Per aumentare il coefficiente di ridondanza dovremmo aggiungere almeno un bit (aumentando il numero di bit aumenta anche il coefficiente di ridondanza)

Prendiamo la codifica di prima

1010101010

E aggiungiamo 1 bit

1010101010

Il sottoinsieme dei valori **usati** non cambia, quello che cambia è il numero totale di combinazioni, che raddoppia (aggiungendo 1 bit)

Supponiamo che, in fase di trascrizione, sia stato fatto un solo errore

+1

1010101010	<table border="1"><tr><td>1</td></tr><tr><td></td></tr></table>	1	
1			
1011101010	<table border="1"><tr><td>1</td></tr><tr><td></td></tr></table>	1	
1			

Usando l'algoritmo basato sul cosiddetto “controllo di parità” si può riconoscere sempre la presenza di tale errore.

Aggiungere un “controllo di parità” vuol dire contare il numero di “uni” (cifre binarie 1) all'interno della nostra rappresentazione.

Posso dire, per esempio, che “sono consentite” rappresentazioni solo con un numero pari di “uni”.

Sono partito da una rappresentazione a 10 bit che contiene un numero dispari di “uni” (5 per la precisione): per fare in modo che, con l'aggiunta dell' 11° bit il numero di “uni” diventi pari, devo riempire la casella con un 1.

Come abbiamo già detto assumiamo ci sia un solo errore: aggiungendo il bit di parità gli “uni” diventano 7, 7 è un numero dispari e quindi non è più verificata la condizione di parità.

Ma, analogamente, se l'errore fosse stato commesso da un'altra parte, il controllo di parità varrebbe lo stesso, poiché, per esempio, se l'errore fosse invece sul 3° bit, avrei tolto un 1, quindi rimarrei con 4 “uni”, aggiungendo il bit di parità arrivo a 5, numero dispari di nuovo: non è verificata la condizione di parità.

+1

1010101010	<table border="1"><tr><td>1</td></tr><tr><td></td></tr></table>	1	
1			
1000101010	<table border="1"><tr><td>1</td></tr><tr><td></td></tr></table>	1	
1			

Quindi, se aggiungo un bit di parità e faccio il controllo di quante siano le cifre binarie 1, se trovo una rappresentazione con un numero dispari di “uni”, so che in quella rappresentazione c’è un errore.

ATTENZIONE: L'errore potrebbe essere benissimo anche sul bit di parità:

+1

1010101010	<table border="1"><tr><td>1</td></tr><tr><td></td></tr></table>	1	
1			
1010101010	<table border="1"><tr><td>0</td></tr><tr><td></td></tr></table>	0	
0			

Prima dell' 11° bit le rappresentazioni sono completamente uguali ma nel primo caso, come bit di parità, abbiamo un 1, così da arrivare ad un numero pari di "uni" ($5+1=6$), nel secondo caso abbiamo uno 0, facendo $5+0$ rimane 5, quindi la condizione di parità non è verificata e quindi c'è 1 errore.

Cosa succede usando l'algoritmo di parità? Succede che, partendo da una codifica corretta, introducendo un singolo errore, "passo" per forza "dall'altra parte", cioè nel sottoinsieme delle codifiche "**non consentite**" (perché abbiamo esplicitato all'inizio dell'algoritmo che consentiamo solo le rappresentazioni con un numero pari di "uni").

Tale algoritmo però funziona solo per un numero dispari di errori. Questo possiamo capirlo facendo banalmente la prova: se gli errori fossero 2 entrambe le rappresentazioni risulterebbero con un numero pari di "uni".

Ad es:

+1		+1
1010101010	1	
1000101010	0	

Se gli errori invece fossero 3:

+1	+1	+1
1010101010	1	
1000100010	0	

Avremmo 6 "uni" (numero pari) nella rappresentazione posta sopra e 3 "uni" (numero dispari) in quella posta sotto.

Come faccio, però, una volta che mi sono accorto che ci sono errori, a capire se ce ne sia 1 oppure 3 oppure 5 o 7? Ci sono 2 atteggiamenti diversi per affrontare il problema dopo aver riscontrato errori:

- Mi accorgo che ci sono errori (c'è almeno un errore), quindi che sia 1 o 3 o 5 non mi interessa, butto tutto e rifaccio, (se per esempio stiamo parlando di trasmissione di informazione da un mittente ad un destinatario, chiedo al mittente di ritrasmettere, perché ci sono stati errori). Quindi da questo punto di vista l'unica cosa importante è riconoscere la presenza di errori oppure no.
- Mi accorgo che ci sono errori, cerco di correggerli. Ma questo è molto più complicato, perché, tra l'altro, l'algoritmo di parità non mi permette di sapere DOVE c'è stato l'errore oltre a non permettermi di sapere quanti errori ci siano stati

Ciò su cui l'algoritmo basato sul controllo di parità mi aiuta è farmi vedere che c'è **ALMENO 1 ERRORE**.

(ovviamente il controllo di parità funziona anche se decidiamo di "consentire" un numero dispari di "uni", funziona alla stessa maniera, solo che bisogna verificare la condizione di disparità, e non quella di parità)

Appunti 7 ottobre

0

1	0	0	0	0	1	0
---	---	---	---	---	---	---

parità

ASCII (LETTERA B, VALORE 66)

1

1	0	0	0	0	1	1
---	---	---	---	---	---	---

parità

ASCII LETTERA C (VALORE 67)

Distanza di Hamming tra le due prima del controllo di parità: 1

$$H('B', 'C')=1$$

Dopo il controllo di parità, la distanza di Hamming diventa 2

$$H('B', 'C')=2$$

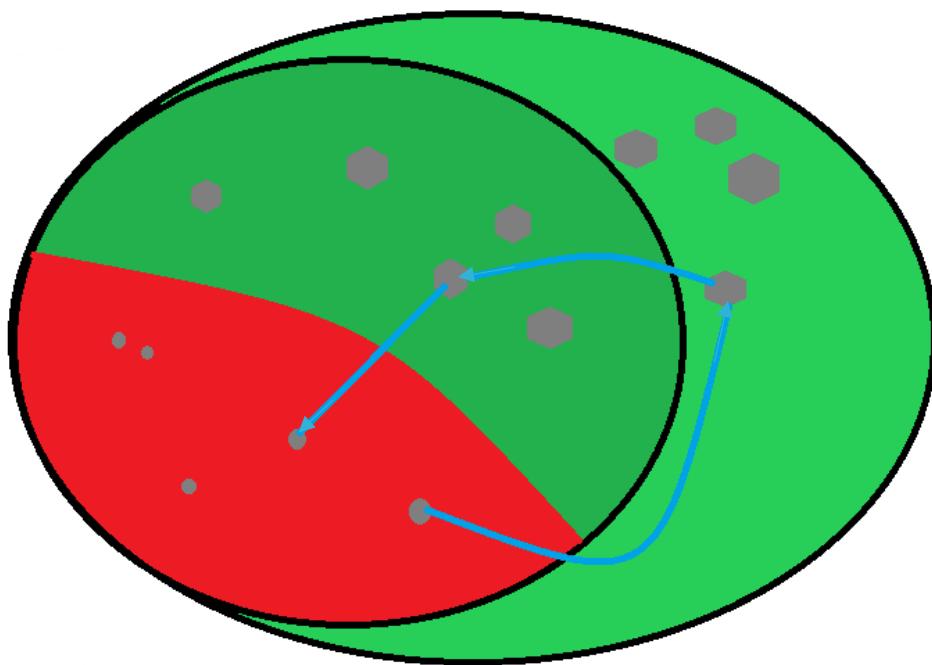
Abbiamo una controprova del fatto che, per passare da una rappresentazione “legale” (“consentita”) ad un’altra **consentita** dobbiamo cambiare 2 bit. (se cambiassimo un solo bit, in una rappresentazione, il controllo di parità andrebbe a buon fine, mentre nell’altra no)

Grazie al controllo “di parità” possiamo rilevare la presenza di un numero dispari di errori (1,3,ecc) *

(aggiungendo il bit di parità mi garantisco che la distanza di Hamming tra 2 codici corretti sia maggiore o uguale a 2, ciò ha come effetto il fatto che, se viene introdotto anche un singolo errore, “esco” dal sottoinsieme delle combinazioni possibili e mi accorgo subito dell’errore)

Cosa dovremmo fare per riconoscere la presenza di 2 errori? Devo aggiungere un secondo bit di parità
Perché? aggiungendo un altro bit di parità, "aggiungo" delle altre configurazioni **non consentite**, quindi faccio in modo che, per poter passare da una qualunque configurazione **consentita** ad un'altra qualsiasi configurazione **consentita** sia necessario "passare attraverso" 2 configurazioni **non consentite**.

In questo modo:



Devo fare in modo che i 2 codici abbiano distanza di Hamming maggiore o uguale a 3, così che, con anche un solo errore (ma anche con 2 errori), io esca dal sottoinsieme delle combinazioni possibili e mi renda conto che si è verificato un errore.

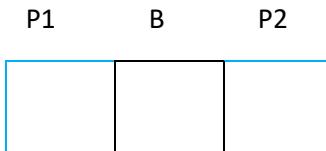
Se io voglio rilevare la presenza di **10 errori**, dovrò fare in modo che il mio codice abbia distanza di Hamming ≥ 11 , per ogni coppia di **rappresentazioni consentite**

Un codice con $H \geq 3$ ci permette di riconoscere con certezza la presenza di 1 oppure 2 errori

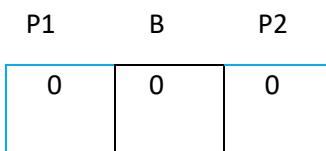
Supponiamo di avere una informazione codificata su un singolo bit. (valore del tipo 1 o 0, V o F)



Aggiungo 2 bit di parità.

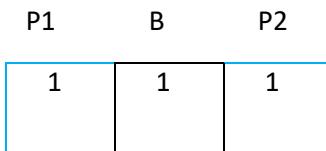


B: Bit originario; P1 e P2: Bit di parità 1 e 2

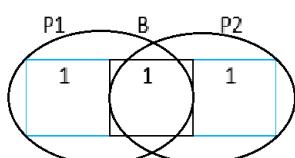


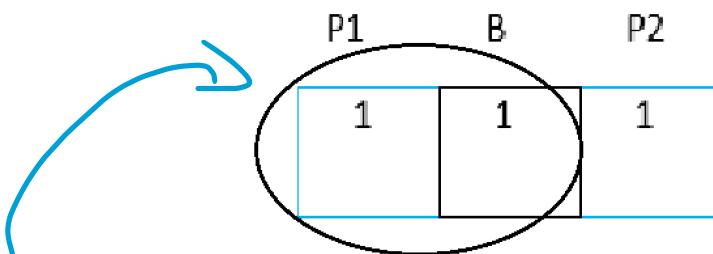
Supponiamo che il Bit originario valga 0, i 2 bit di parità devono valere 0 anch'essi

Supponiamo ora che il Bit originario sia 1, P1 deve valere 1, P2 deve valere anch'esso 1



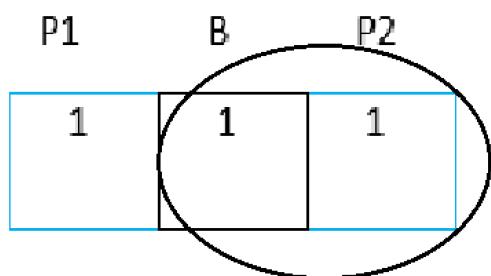
Perché? Perché io verifico la condizione di parità rispetto al Bit originario, quindi non bisogna tenere conto del complesso, ma del sottoinsieme all'interno del quale verifichiamo la condizione di parità.





Ossia:

quando aggiungo il bit di parità 1 (P1) io tengo conto di QUESTO sottoinsieme per fare la verifica di parità.



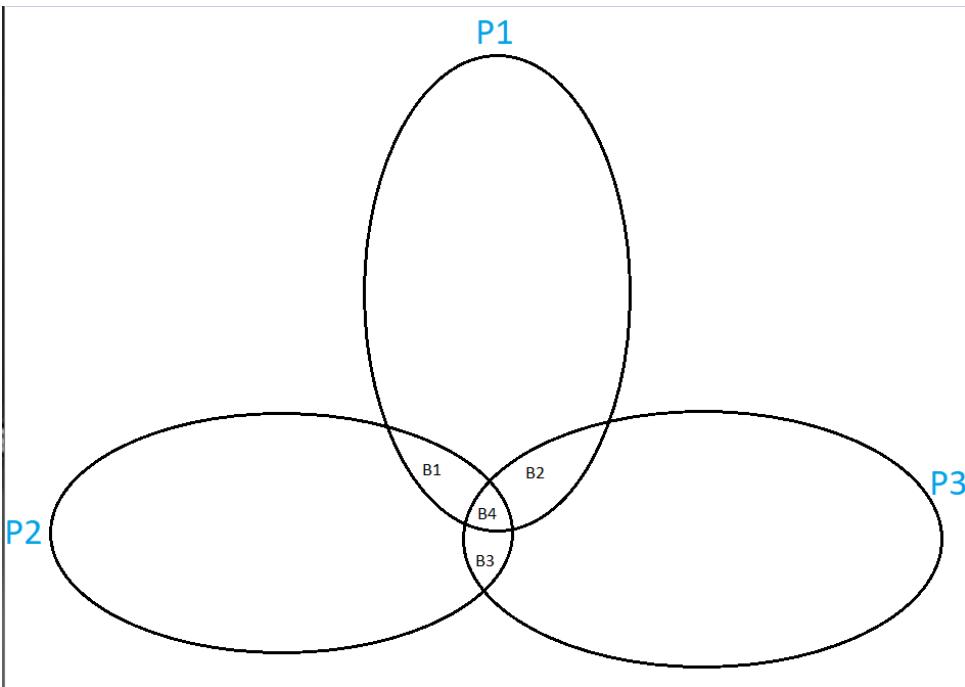
Quando invece aggiungo il bit di parità 2 (P2) devo fare la verifica su QUEST'ALTRO sottoinsieme.

Avendo un solo bit di informazione, introducendo 2 bit di parità, sono passato da distanza di Hamming=1 ($0 \neq 1$, senza bit di parità) a distanza di Hamming=3.

Se voglio coprire più bit di informazione mi servono più bit di parità, in modo da garantire $H \geq 3$ per ogni coppia di rappresentazioni diverse.

Posso ragionare sul modo in cui farlo attraverso le intersezioni tra i sottoinsiemi di copertura dei bit di parità.

Come abbiamo visto in precedenza, nell'intersezione tra 2 bit di parità ci sta 1 bit di informazione.



N.B: questo modo di calcolare i bit di parità prevede che, per esempio, per calcolare il bit di parità P1 devo contare il numero di "uni" compresi in B1, B2 e B4, per calcolare il bit di parità P2 devo contare il numero di "uni" presenti in B1, B3 e B4, per calcolare il bit di parità P3 devo contare il numero di "uni" presenti in B2, B3 e B4.

Ci accorgiamo che, per costruire un codice con distanza di Hamming ≥ 3 per ogni coppia di rappresentazioni diverse, (siccome abbiam visto che per 1 bit di informazione usiamo 2 bit di parità, e ora, per 4 bit di informazione usiamo 3 bit di parità) abbiamo un rapporto tra numero di bit di informazione e numero di bit di parità che si può riassumere con una tabella di questo tipo:

Bit di parità	Bit di informazione	N° totale di bit
2	1	3
3	4	7
4	11	15
5	26	31
6	57	63

$2^2 - 1$
 $2^3 - 1$
 $2^4 - 1$
 $2^5 - 1$
 $2^6 - 1$

Il numero di bit totali cresce come $(2^{\text{numero di bit di parità}}) - 1$

Notiamo che, se vogliamo costruire un codice con $H \geq 0$ con una certa quantità di bit, il numero di bit di parità è $= [\log_2(n^{\circ} \text{ tot di bit} + 1)]$ oppure, più semplicemente, è il $\log_2(n^{\circ} \text{ tot di bit})$ arrotondato per eccesso.

Come si può semplificare questo processo, rendendolo lineare?

Costruisco un codice arbitrariamente lungo, seguendo questa regola:

P1	P2	B1	P3	B2	B3	B4	P4	B5	B6	B7	B8	B9	B10	B11	P5	B12	B13	B14	B15	B16	B17	B18	B19	B20	B21
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
2^0	2^1	2^2					2^3								2^4										

Costruisco tale tabella lineare basandomi sull'assunto (ricavato dalla tabella precedente) che, con 2 bit di parità posso avere 1 bit di informazione, con 3 bit di parità posso avere 4 bit di informazione, e così via.

N.B: i numeri sottolineati rappresentano la POSIZIONE del bit.

Osservando la tabella mi accorgo subito che i bit di parità (P) stanno in posizioni rappresentate da POTENZE DI 2 (1,2,4,8,ecc)

Con questo "ordinamento" dei bit posso facilmente calcolarmi i bit di parità, ad es:

quali sono i bit di parità che "coprono" B3? Sono P2 e P3 (come vediamo anche nello schema delle intersezioni*), ma con questo ordinamento è molto più semplificato: in effetti, se noi andiamo a vedere le posizioni di P2 e P3 (posiz. 2 e posiz. 4) e le sommiamo arriviamo a 6 (la posizione di B3), ciò si può fare quindi, anche con un più grande numero di bit (quando invece con la rappresentazione grafica non potevo calcolarmi, per esempio, B5, poiché avrei dovuto disegnare a 3 dimensioni).

Es:

- quali sono i bit di parità per B4 (sta in posizione 7)? P1, P2 e P3 (posizioni 1 2 e 4 $\rightarrow 4+2+1=7$)
- quali sono i bit di parità per B5 (sta in posizione 9)? P1 e P4 (posizioni 1 e 8 $\rightarrow 1+8=9$)
- quali sono i bit di parità per B15 (sta in posizione 20)? P3 e P5 (posizioni 4 e 16 $\rightarrow 4+16=20$)

Praticamente scomponiamo l'indice del bit di informazione in questione in SOMMA DI POTENZE DI 2, e quelle potenze di 2 sono le posizioni dei bit di parità che vengono calcolati tenendo conto di tale bit di informazione.

N.B: Per dire "posizione" posso anche usare il termine "indice".

N.B: La striscia/tabella lineare si chiama array

Torniamo alle rappresentazioni in codice ASCII (7 BIT)

B1	B2	B3	B4	B5	B6	B7
1	0	0	0	0	1	0

Lettera B

Abbiamo 7 bit, spostiamoli in un array come quelli costruiti in precedenza

P1 P2 B1 P3 B2 B3 B4 P4 B5 B6 B7

		1		0	0	0		0	1	0
1	2	3	4	5	6	7	8	9	10	11

Riempiamo i bit di parità avvalendoci di quanto detto prima:

- Per B1 abbiamo P1 e P2 come bit di parità.
- Per B2 abbiamo P1 e P3 come bit di parità.
- Per B3 abbiamo P2 e P3 come bit di parità.
- Per B4 abbiamo P1, P2 e P3 come bit di parità.
- Per B5 abbiamo P1 e P4 come bit di parità.
- Per B6 abbiamo P2 e P4 come bit di parità.
- Per B7 abbiamo P1, P2 e P4 come bit di parità.

P1 P2 B1 P3 B2 B3 B4 P4 B5 B6 B7

1	0	1	0	0	0	0	1	0	1	0
1	2	3	4	5	6	7	8	9	10	11

- I bit di informazione di posizione dispari contribuiscono al calcolo del bit di parità P1, in questi 5 bit c'è un solo 1 → per far tornare il controllo di parità P1 dobbiamo inserire il valore 1
- I bit di informazione che contribuiscono al calcolo del bit di parità P2 sono: B1, B3, B4, B6 e B7, in questi 5 bit ci sono 2 "uni" → per far tornare il controllo di parità dobbiamo inserire il valore 0
- I bit di informazione che contribuiscono al calcolo del bit di parità P3 sono: B2, B3 e B4, in questi 3 bit non c'è nessun 1 → per far tornare il controllo di parità dobbiamo inserire il valore 0.

- I bit di informazione che contribuiscono al bit di parità P4 sono: B5, B6 e B7, in questi 3 bit c'è un 1 → per far tornare il controllo di parità devo inserire il valore 1.

Quindi, partendo da una rappresentazione ASCII a 7 bit 1000010, per poter identificare 1 o 2 errori introduciamo una rappresentazione su 11 bit 10100001010 abbiamo aggiunto 4 bit di **parità**, calcolati opportunamente.

Esercizio: fare la stessa cosa con il carattere C.

E quindi verificare che, dopo l'introduzione del bit di parità, la distanza di Hamming tra B e C valga effettivamente 3.

Come già detto, tutto ciò mi permette di riconoscere la presenza di 1 o 2 errori,

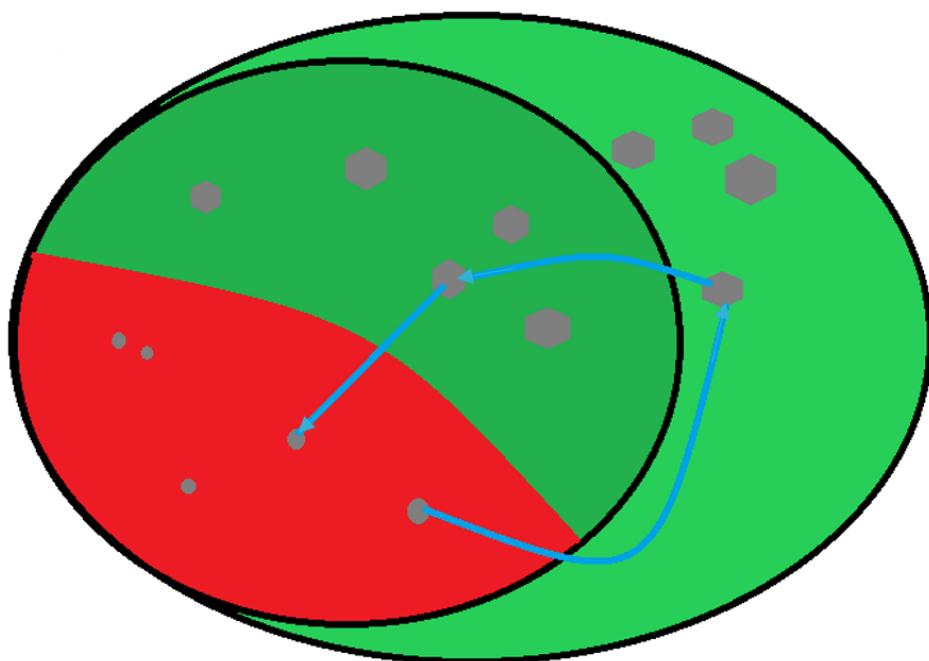
per sapere se si tratta di 1 o di 2 errori devo aggiungere un ulteriore bit di parità che mi permette di fare questa distinzione.

Ma come faccio a correggerli?

Con questi tipi di codici con $H \geq 3$ posso correggere 1 errore.

Per prima cosa, se sono certo che c'è stato solo 1 errore, posso misurare la distanza di Hamming tra la configurazione sbagliata e quelle giuste: prendo la configurazione che SO che è SBAGLIATA (il controllo di parità non torna) e calcolo la distanza con tutte le configurazioni giuste.

Supponiamo di avere un numero contenuto di configurazioni corrette, come nel disegno:



Tra queste ce ne sarà solo UNA con $H=1$ (le altre configurazioni corrette avranno distanza di Hamming maggiore o uguale a 2 dalla configurazione sbagliata)

Se volessi correggere 2 errori dovrei avere un codice con $H \geq 5$, in modo da avere UNA SOLA tra le codifiche **corrette** che abbia distanza di Hamming=numero di errori dalla codifica sbagliata; e tutte le altre codifiche **corrette** a distanza maggiore dalla codifica sbagliata.

Dunque, volendo aggiungere una formula: se io voglio correggere una quantità di errori E ho bisogno di

$$H \geq 2E+1$$

Quindi le applicazioni dei codici a $H \geq 3$ sono:

- Identificare la presenza di 1 o 2 errori
- Correggere l'errore, in caso se ne fosse verificato solo 1.

Per fare un esempio di correzione usiamo lo stesso array di prima:

P1	P2	B1	P3	B2	B3	B4	P4	B5	B6	B7
1	0	1	0	0	0	0	1	0	1	0
1	2	3	4	5	6	7	8	9	10	11

Introduciamo, per esempio, l'errore nel bit B2

P1	P2	B1	P3	B2	B3	B4	P4	B5	B6	B7
1	0	1	0	1	0	0	1	0	1	0
1	2	3	4	5	6	7	8	9	10	11

Innanzitutto, vediamo se i controlli di parità falliscono (succede per forza, almeno per un P, perché ho cambiato 1 bit):

- Controllo di parità P1: guardo i bit nelle posizioni 3, 5, 7, 9 e 11 → senza controllo di parità gli "uni" sono 2; quindi il controllo di parità fallisce, poiché con un altro 1 (di P1) diventano 3 "uni".

- Controllo di parità P2: guardo i bit nelle posizioni 3, 6, 7, 10 e 11: ci sono 2 “uni”, P1 è 0, quindi gli “uni” rimangono 2 → questo controllo di parità non fallisce
- Controllo di parità P3: guardo i bit nelle posizioni 5, 6 e 7: c’è un 1. P3 è 0 quindi rimane un solo 1 → controllo di parità fallisce
- Controllo di parità P4: guardo i bit nelle posizioni 9, 10 e 11: c’è un 1. P4 è 1, quindi gli “uni” diventano 2 → questo controllo di parità non fallisce.

È sufficiente che un solo controllo di parità fallisca per dire che c’è stato un errore.

Per fare la correzione di tale errore devo tenere conto dei controlli di parità che hanno fallito (P1, che si trova in posizione 1 e P3, che si trova in posizione 4, sommando le posizioni trovo 5: la posizione in cui si è verificato l’errore) → per correggere l’errore devo semplicemente invertire il valore: se c’è 1 metto 0, se c’è 0 metto 1.

Breve osservazione: supponiamo che, con un’altra configurazione binaria, fallisca anche il controllo di parità P4: facendo la somma dei P falliti (P1, P3 e P4, che si trovano nelle posizioni 1, 4 e 8 trovo 13, ma io ho SOLO 11 bit → cosa vuol dire? Vuol dire che ci sono 2 errori)

Quindi

N.B.: se facendo il calcolo della posizione del bit da correggere ci viene fuori un indice superiore al numero di bit che stiamo usando, abbiamo la certezza che non si è verificato solo 1 errore

Ma che tipo di errori esistono?

Prendiamo in esempio, invece di errori “casuali” nella “trascrizione”, un errore causato da un guasto (anche momentaneo) di un dispositivo.

Come si corregge?

CORREZIONE DI CANCELLAZIONE

Abbiamo un codice a 8 bit

1	0	1	1	0	0	0	1
---	---	---	---	---	---	---	---

Un modo per prevenire l’errore in caso di guasto del dispositivo in cui è contenuta l’informazione è fare una “copia” → prendere un altro dispositivo uguale e memorizzarci dentro lo stesso valore.

1	0	1	1	0	0	0	1
---	---	---	---	---	---	---	---

Ciò significa raddoppiare il numero di bit, però.

Oppure posso usare 2 dispositivi, ma in ognuno di essi far entrare la metà delle informazioni.

1	0	1	1
---	---	---	---

0	0	0	1
---	---	---	---

E aggiungere un altro dispositivo per il controllo di parità

1	0	1	0
---	---	---	---

Supponiamo che uno dei 3 dispositivi smetta di funzionare.

Se smette di funzionare quello di parità basta sostituire il dispositivo e ricalcolare i bit di parità (siccome gli altri funzionano bene)

Supponiamo che si guasti il primo dispositivo: basta comunque calcolare il bit di parità e riavere le informazioni di prima

Ad esempio:

G	U	AS	TO
---	---	----	----

0	0	0	1
---	---	---	---

1	0	1	0
---	---	---	---

CALCOLO parità BIT A BIT.

1 0 1 1

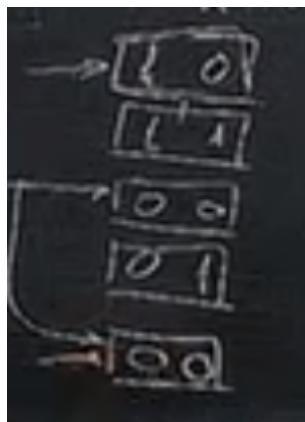
0	0	0	1
---	---	---	---

1	0	1	0
---	---	---	---

Sostituisco il dispositivo guasto e ci scrivo dentro i bit di parità calcolati e ritorno quindi alla configurazione iniziale

Se quindi si guasta UN DISPOSITIVO, è risolvibile, grazie al controllo di parità. Ciò non basta se se ne guastano 2 o più.

Per diminuire la ridondanza si può spezzettare in blocchi da 2, per esempio



Il blocco dei bit di parità stesso è fatto da 2 bit (viene 0 0 perché $1+1=2$ e $1+1=2$)

Il problema è che, aumentando il numero di dispositivi si aumenta anche la probabilità che 1 o più di 1 si guastino.

Rappresentazione numeri con segno

MODULO E SEGNO

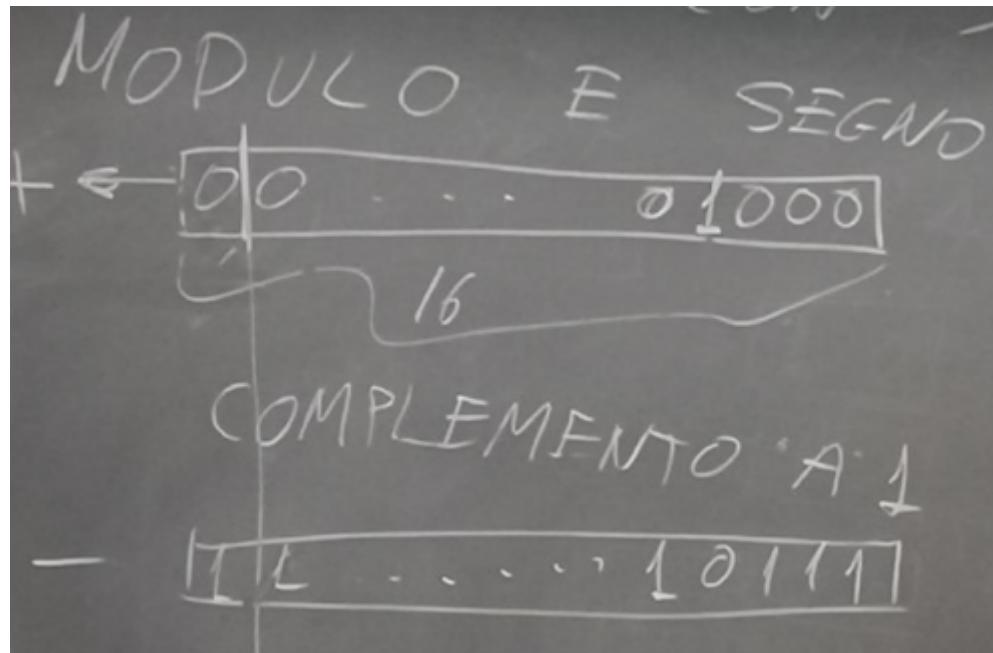
+	←	0							
-	←	1							

1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
----------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

-8 in una rappresentazione in modulo e segno a 16 bit (15 bit di modulo e uno di segno)

Questo tipo di rappresentazione però ha diversi svantaggi, per esempio per quanto riguarda l'implementazione delle operazioni algebriche. Se dovessi fare, per esempio, A+B (in cui A vale 7 e B vale -8) l'algoritmo sarebbe troppo complesso perché bisognerebbe confrontare i segni eccccc.

Posso usare la rappresentazione in **COMPLEMENTO A 1**



In complemento a 1, come potrei rappresentare -8? Partiamo sempre dalla rappresentazione del modulo (quindi i 15 bit di modulo sono identici alla rappresentazione in modulo e segno, per adesso)

E scriviamo nel bit di segno il segno positivo

0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
----------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Così facendo abbiamo scritto il valore +8. Come passiamo dal valore +8 al -8? Invertiamo tutti i bit.

(così facendo, "negando" tutti i bit troviamo un valore che è l'opposto di quello di partenza)

1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1
----------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

-8 in complemento a 1

(la rappresentazione di un numero positivo in modulo e segno oppure in complemento a 1 è, quindi, la stessa)

Neanche la rappresentazione in complemento a 1 ci dà vantaggi per quanto riguarda le operazioni ma è un passo intermedio tra la modulo e segno e la complemento a 2 che invece ci aiuta nelle operazioni

Da complemento 1 a complemento 2 si passa sommmando alla rappresentazione a complemento 1 la costante 1

1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1
----------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

+1

1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
----------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(in questo caso sommare la costante 1 significa sommare alla rappresentazione che abbiamo una rappresentazione di questo tipo

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(la rappresentazione di un numero positivo è la stessa sia per complemento a 1, sia per modulo e segno e sia per complemento a 2)

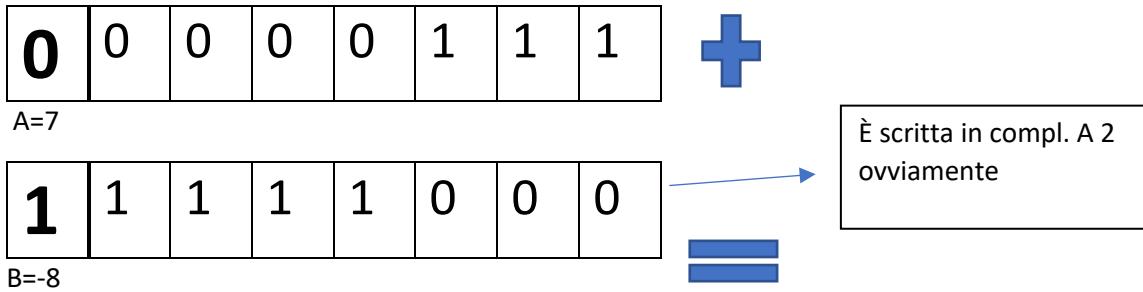
Se mi capitasse di avere, per esempio, una rappresentazione a 16 bit in complemento a 1, a cui devo sommare 1 per arrivare al complemento a 2 e nell'ultimo bit a sinistra mi trovassi a sommare 1 con 1, così da avere 0 con riporto di 1, il riporto starebbe nel 17° bit, quindi non ne terrei conto, lo "caccerei via", perché sto usando una rappresentazione a 16 bit.

Proviamo a fare l'operazione A+B, dove a vale 7 e b vale -8

Quindi 7 +(-8)

Usiamo una rappresentazione a 8 bit

+7 si scrive così, su 8 bit, sia in modulo e segno, sia in compl. A 1 e sia in compl. A 2



1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

Abbiamo trovato la rappresentazione che, in complemento a 2, vale -1. Quindi è giusto.

Prova a fare altri esempi

.....

In complemento a 2 c'è UNA SOLA rappresentazione del valore 0, caratteristica che, dal punto di vista dei confronti è considerata un vantaggio

Avendo UNA sola rappresentazione di 0 (a differenza del modulo e segno e del complemento a 1, che ne hanno 2) posso rappresentare un numero negativo in più.

OGNI VALORE HA UNA SOLA RAPPRESENTAZIONE.

POSSIAMO EFFETTUARE LE OPERAZIONI DI SOMMA FACILMENTE. (senza far conto del segno, considerando solo i bit, ottengo un risultato che, se lo concepisco come rappresentazione in complemento a 2 è corretto)

Quindi la complemento a 2 è preferibile alle altre due rappresentazioni.

All'interno dei sistemi di calcolo, quindi, viene quasi esclusivamente usata la rappresentazione in complemento a 2 per rappresentare numeri con segno.

Altra rappresentazione:

ECCESO 2^{N-1}

N=8 (il numero di bit è 8)

Cosa vuol dire ECCESO 2^{N-1} ?

Se io prendo un certo valore V e gli sommo la costante 2^{N-1} ottengo un altro numero (ovviamente)

Se io metto una restrizione sul valore di V, e dico che può valere al massimo 127, il valore massimo che posso ottenere da $V + 2^{N-1}$ è 127. $127+128=255$. Questo valore, posso scriverlo in numero binario come

1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

Quindi questa è la rappresentazione di $V = +127$

$$C = \text{Valore} + 2^{N-1}$$

Supponiamo che V valga 0

1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

Questa è la rappresentazione in eccesso 2^{N-1} di 0

C'è una unica rappresentazione di 0, come in complemento a 2

Se voglio tenere $C \geq 0$, il valore minimo di V può essere -128

Quindi l'intervallo di valori rappresentabili con questa notazione va da -128 a +127 (come in complemento a 2)

- Posso rappresentare tutti i numeri interi da -128 a +127
- Ho una sola rappresentazione per lo 0
- Ho un numero negativo in più rispetto a compl. A 1 e modulo e segno

Questa è quindi, da questi punti di vista, una rappresentazione equivalente a quella in complemento a 2

L'unica cosa che cambia è l'interpretazione del bit di segno: ci accorgiamo che usando l'eccesso 2^{N-1} 0 significa - e 1 significa +

Quindi ciò che non abbiamo è la facilità di sommare numeri senza tener conto del segno (come invece si può in complemento a 2)

Altro es:

rappresentazione di +3 in eccesso 2^{N-1}

1	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---

Per trasformare questa rappresentazione in una rappresentazione in complemento a 2 mi basta cambiare il bit di segno

0	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---

+3 in complemento a 2

Rappresentazione numeri frazionari

Fixed point (a virgola fissa)

PRENDIAMO UNA RAPPRESENTAZIONE BINARIA, per esempio il valore 3, e lo moltiplichiamo per una costante K

0	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---

* K

$K=2^{-5}$

Ciò significa $3 * 2^{-5}$ = un numero molto più piccolo di 1

Quindi posso immaginare di inserire una virgola all'interno della rappresentazione

Se moltiplico per 2^{-5} , per esempio, posso immaginare di mettere la virgola qui

$2^2 \quad 2^1 \quad 2^0 \quad 2^{-1} \quad 2^{-2} \quad 2^{-3} \quad 2^{-4} \quad 2^{-5}$

0	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---

,

virgola che separa la parte intera da quella frazionaria, ciò significa che dò una interpretazione diversa alle potenze di 2 che corrispondono a ciascuna posizione.

2^0 per esempio, non sarà più nel primo bit da destra, ma nel primo bit da destra “prima della virgola”, in questo caso il 6° bit da destra. Se quindi ho 2^0 nel 6° bit da destra, nel 5° avrò 2^{-1} , nel 4° 2^{-2} , così via fino a 2^{-5} nel primo bit.

Se io volessi fare la somma di 2 numeri in rappresentazione a virgola fissa dovrei codificare i due numeri mettendo la virgola nello stesso posto

--	--	--	--	--	--	--	--

,

se per esempio volessi sommare il valore di prima a 1.

Devo codificare il valore 1 in questo modo

0	0	1	0	0	0	0	0
---	---	---	---	---	---	---	---

,

supponiamo ora di voler rappresentare dei valori di probabilità (che possono variare tra 0 e 1)

min: 0 e max: 1

posso mettere la virgola dopo il primo bit a sinistra

0	0		0	0	0	0	0
---	---	--	---	---	---	---	---

,

il bit più significativo (a sinistra) rappresenta l’unità e i bit successivi sono delle frazioni

come rappresento quindi i valori?

0	1	0	0	0	0	0	0
---	---	---	---	---	---	---	---

,

questo sarà il valore $\frac{1}{2}$ (cioè 2^{-1})

0	0	1	0	0	0	0	0
---	---	---	---	---	---	---	---

,

e questo sarà il valore $\frac{1}{4}$ (cioè 2^{-2})

facendo la somma dei due valori ottengo $\frac{3}{4}$ (cioè $2^{-1} + 2^{-2}$):

0	1	1	0	0	0	0	0
---	---	---	---	---	---	---	---

,

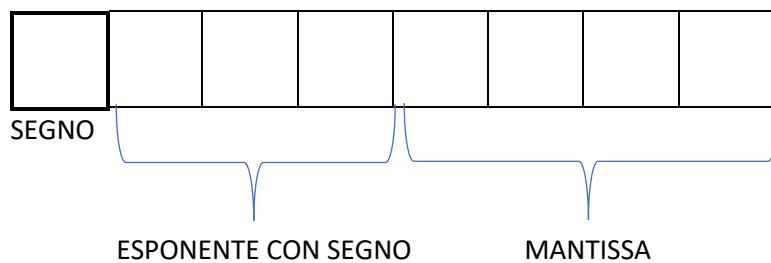
i dispositivi, quindi, sono costruiti allo stesso modo, anche per le rappresentazioni a fixed point,

siamo noi che decodifichiamo in un certo modo. È il nostro modo di interpretare la rappresentazione binaria che cambia. ("interpretazione" della virgola e convenzione della posizione della stessa)

L'esempio qui di sopra è una rappresentazione binaria in 8 bit che avrebbe valore 96, ma noi, immaginando di avere una virgola dopo il primo bit a sinistra, la interpretiamo come $\frac{3}{4}$ (cioè $2^{-1} + 2^{-2}$).

Appunti del 14 ottobre

Rappresentazione di numeri frazionari in virgola mobile (FLOATING POINT)



formula per codificare e decodificare i numeri in questo formato:

$$V = (-1)^S * 2^E * M$$

Il valore 1, allora, lo rappresenteremmo come M=1, E=0

Ma possiamo rappresentarlo in svariati modi usando la formula, per esempio

M=2, E=-1;

M=4, E=-2;

M= 0,5, E=+1

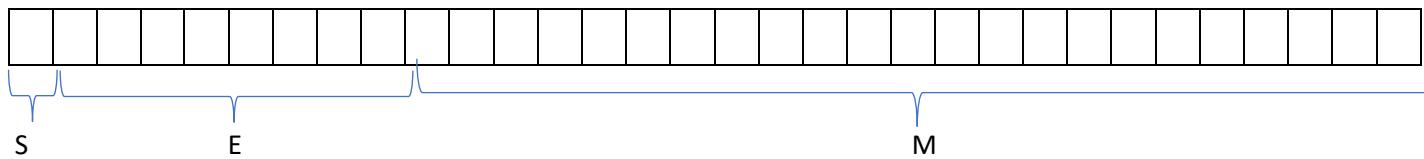
Questo è quindi molto sconveniente: non c'è una unica rappresentazione per ogni valore, ma ce ne sono tantissime. Non vogliamo sprecare bit per avere tante rappresentazioni di uno stesso numero, cosa che ci complica anche i confronti di uguaglianza.

Ci serve una RAPPRESENTAZIONE UNICA per ogni valore, mettiamo dei vincoli sui possibili valori che M ed E possono assumere

RAPPRESENTAZIONE CANONICA: STANDARD PER NUMERI IN VIRGOLA MOBILE SI CHIAMA IEEE754

Tale standard permette rappresentazioni a 16, 32, 64, 128 e 256 bit.

IEEE per 32 bit permette che ci sia 1 bit di Segno, 8 di esponente e 23 di Mantissa.



Forma normalizzata (mette un vincolo sulla mantissa):

$$1 \leq M < 2$$

Non per tutti i numeri, però, si può usare la rappresentazione normalizzata, ad esempio, per 0 non si può usare (dovrei avere Mantissa= 0 per rappresentare 0, ma la mantissa nella forma normalizzata deve essere ≤ 1 .

Per i numeri non normalizzabili si usa, quindi, una rappresentazione non normalizzata.

Siccome abbiamo la forma $1 \leq M < 2$

Il primo bit vale sempre 1, quindi non devo scrivere 1,b₁ b₂ b₃ ecc

È implicito. È inutile scrivere l'1 se sappiamo che il primo bit della mantissa vale sempre 1. L'idea è quindi quella di usare tutti e 23 i bit della mantissa per codificare i numeri dopo la virgola.

Posso usare dunque 23 bit per la mantissa (24 implicitamente, perché c'è anche l'1 prima della virgola, che però non scrivo), quindi 23 spazi solo per i bit dopo la virgola, senza contare l'1, che non scrivo.

Devo quindi immaginare di vedere:

1,



per quanto riguarda la parte di bit dell'esponente che deve essere un numero con segno codificato con una rappresentazione in eccesso 127

quindi l'esponente 0 viene codificato con la stringa di numeri binari corrispondenti al valore 127, l'esponente +1 verrà codificato come 128 ($V=1 + 127$), l'esponente 2 come 129 ($V=2 + 127$), -1 come 1226 ($V=-1 + 127$).

Ma perché usare eccesso 127 invece che eccesso 2^7 ? (128)

Per avere delle codifiche diverse dalla codifica normalizzata

Il più piccolo valore che l'esponente potrebbe assumere (0) e che quindi in stringa binaria in eccesso 127 corrisponderebbe a -127, non viene interpretato come -127, bensì come "rappresentazione non normalizzata".

Il più piccolo valore che l'esponente può assumere sarà quindi -126 (in rappresentazione in eccesso a 127: $-127 + 127$ farebbe 0, ma sappiamo che questo rappresenta una rappresentazione non normalizzata)

(quello che sarebbe -127 diventa "rappresentazione non normalizzata"). Cosa analoga succede per l'altro estremo: il valore più grande rappresentabile è +127, poiché quello che sarebbe l'ultimo valore è "tenuto libero" per un'altra rappresentazione non normalizzata → E=255 non è un valore di un esponente in forma normalizzata. Quando E assume valore 255 non è un numero, ma è il risultato di una operazione di divisione per 0 → tipicamente situazione di errore nei sistemi di calcolo

Si differenziano 3 casi:

- NaN Not a Number (0/0)
- $+\infty$ (+V/0)
- $-\infty$ (-V/0)

Quando la mantissa contiene tutti 0 ed E ha valore 255 allora ho un ∞ , per sapere il segno dell' ∞ basta guardare il bit di segno.

Se la mantissa contiene un valore diverso da 0 ed E ha valore 255, allora ho un NaN (0/0)

$-126 \leq E \leq 127$

I valori compresi tra 1 e 254, quindi, sono interpretati come rappresentazioni in forma normalizzata

Se voglio decodificare l'esponente devo sottrarre 127 al valore iniziale (compreso tra 1 e 254).

Se voglio codificare un numero floating point procedo nel seguente modo:

mettiamo di dover codificare il valore -5

$$V=-5$$

Formula $V = (-1)^S * 2^E * M$

Il bit di segno dovrà valere 1 (-) ;

quanto dovranno valere esponente e mantissa? Sappiamo che, date le condizioni di normalizzazione M non deve essere strettamente minore di 2 e maggiore o uguale a 1

$$1 \leq M < 2$$

Allora, per "scalare" la mantissa (che intuitivamente sarebbe 5), dividiamo 5 per 4, così da ottenere un numero **COMPRESO** tra 1 e 2 da moltiplicare in seguito per 2^E .

Il numero che otteniamo da $5/4$ è $1,25$

Tale numero va moltiplicato per 2^E che, ovviamente, per "riscalare" la mantissa, deve essere 4 (perché $5/4 * 4 = 5$). L'esponente vale, quindi, 2 ($2^2=4$)

Abbiamo quindi

S=1

M=1.25

E=2

Applicando la formula il risultato viene correttamente

$$V = (-1)^1 * 2^2 * 1.25$$

$$-1 * 4 * 1.25$$

adesso bisogna solo inserire tutto nei bit:

E vale 2, rappresentato in eccesso 127 è 129 ($2+127$)

129 si scrive 10000001 come numero binario ($2^0 + 2^7$)

M vale 1,25. Abbiamo già detto che l'1 è implicito quindi non si scrive. Per scrivere la parte dopo la virgola: 0,25 devo scrivere 0100000000000000000000000000 (2⁻², cioè appunto $\frac{1}{4}$ o 0,25 come dir si voglia)

PARLIAMO ORA DELLA RAPPRESENTAZIONE in forma NON NORMALIZZATA

In forma normalizzata, nella mantissa c'è un 1 implicito "prima della virgola", non posso dunque scrivere uno 0 (poiché $1 \leq M < 2$), deduciamo che il più GRANDE numero che possiamo scrivere in forma non normalizzata è 0 (implicito) e poi 23 "uni" nella mantissa

Sempre per quanto riguarda lo standard IEEE 754, per le rappresentazioni a 64 bit avremo:

1 bit di segno, 11 di esponente, 52 di mantissa

E l'esponente viene codificato in eccesso 1023 (quindi valori che E può assumere: $-1023 \leq E \leq 1023$)

Ciò accresce la precisione di rappresentazione.

CODICI A ESPANSIONE

sono una "via di mezzo" tra codici a lunghezza fissa ed a lunghezza variabile

hanno diverse applicazioni:

per esempio la compressione di dati.

Supponiamo di avere un testo in italiano scritto sotto forma di sequenza di caratteri ASCII

LA MELA E BELLA.

Frequenza di occorrenza dei caratteri

L= 4 volte

A= 3 volte

E= 3 volte

M= 1 volta

B= 1 volta

Le altre lettere (D, F, G H,...Z)= 0 volte

Creiamo un codice ottimizzato per la lingua italiana

Lettere dell'alfabeto italiano maiuscole: 21 (togliamo quindi J,K,W,X,Y)

+ SPAZIO + “.” + “,” + “?” + “!”

Per scrivere una qualsiasi frase usando le combinazioni possibili di tali caratteri ci basterà una configurazione a 5 bit.

2^4	2^3	2^2	2^1	2^0

Se vogliamo scrivere la frase riportata sopra dovremo rappresentare 16 caratteri (le lettere, il punto e 3 spazi). Aggiungiamo un altro carattere che sarà il terminatore di stringa (null).

Quindi siamo a 17 caratteri, moltiplicati per 5 (numero di bit della configurazione)

$17 * 5 = 85$ numero di bit che ci serve per rappresentare LA MELA E BELLA.

Ho notato, in precedenza, che ho ripetizioni dello stesso carattere (E 3 volte, A 3 volte, L 4 volte), posso decidere di usare un codice a lunghezza variabile, in cui A, E, L occupano meno di 5 bit, riuscendo quindi a ridurre il numero di bit che mi serve per la stringa.

Quale è il numero minimo di bit che posso usare per rappresentare i 3 caratteri? 2 (perché con 1 bit riesco a rappresentare massimo 2 caratteri, corrispondenti, rispettivamente a 0 e 1)

2 bit	

Posso porre che

A	<table border="1"><tr><td>0</td><td>1</td></tr></table>	0	1
0	1		

E	<table border="1"><tr><td>1</td><td>0</td></tr></table>	1	0
1	0		

L	<table border="1"><tr><td>0</td><td>0</td></tr></table>	0	0
0	0		

I caratteri diversi da questi inizino con la sequenza di valori

1	1
---	---

Es: carattere a caso

1	1	?	?	?	?	?
---	---	---	---	---	---	---

(ai caratteri più frequenti, dunque, sono assegnati codici abbreviati)

Una delle tecniche per avere una espansione del codice è prendere, tra le 4 combinazioni possibili 00,11,10,01, una per indicare che stiamo usando l'espansione (in questo caso 11) → se in una configurazione vediamo 11 sappiamo già che è uno degli altri caratteri, non quelli frequenti (A,E,L, a cui abbiamo assegnato codici abbreviati)

Una altra tecnica è NUMERARE i formati diversi:

0	100	4
L	101	3
2	110	2
3	111	1

(utilizziamo i primi 2 bit di ogni configurazione per codificare la "costante" → così da sapere se è una rappresentazione, per esempio, a 4, a 8, a 12 ecc)

ENTROPIA

Dal punto di vista della teoria dell'informazione

$$E(X) = -\sum P(x_i) \cdot \log_2(P(x_i))$$

Calcolando l'entropia otteniamo il contenuto informativo

Lo stesso messaggio ricevuto 2 volte in tempi diversi la prima volta ha contenuto informativo 1, la seconda ha contenuto informativo 0 (una volta che so già la prima risposta, quando leggo la seconda, che è equivalente alla prima, non imparo niente)

Se mando 2 bit, il contenuto di entropia sarà minore o uguale a 2 bit
(se mando 2 bit, il destinatario non può “imparare” più di 2 bit di informazione).

Vogliamo avere la minima differenza possibile tra il numero di bit che mandiamo e l'effettivo contenuto informativo in bit.

Circuiti logici combinatori

Concepiamo una scatola con un solo filo d'ingresso e un solo filo di uscita



Posso avere 4 comportamenti:

I	U
0	0
1	0

I	U
0	1
1	1

IN QUESTI 2 CASI L'USCITA è COSTANTE, indipendentemente dal filo di entrata(ingresso)

I	U
0	0
1	1

Questa è la FUNZIONE identità, non ci interessa neanche questa (0 è 0 e 1 è 1)

L'unica interessante è la FUNZIONE negazione:

I	U
0	1
1	0

Proviamo ora a vedere che succede se abbiamo 2 fili in ingresso ed 1 in uscita

I_1	I_2	U
0	0	
0	1	
1	0	
1	1	

Nella parte della U potrò avere diverse combinazioni

U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U
0	1	1	0	1	0	0	0	0	0	0	0	0	1	1	1	0	1
0	1	0	1	1	1	0	0	1	1	0	1	0	0	1	0	1	0
0	1	0	0	1	1	1	0	0	1	1	0	1	0	0	0	1	0
0	1	0	0	0	1	1	1	0	0	0	1	1	1	1	1	1	0

Ci sono $2^4 = 16$ funzioni possibili, di cui prendo in considerazione 12 (le funzioni *identità* e quelle *costanti* vanno eliminate), ne elimino anche altre:

- la **negazione** di I_1 (perché si potrebbe usare una scatola più piccola che abbia solamente l'ingresso I_1 e contenga la sua negazione)
- la **negazione** di I_2 (perché si potrebbe usare una scatola più piccola che abbia solamente l'ingresso I_2 e contenga la sua negazione)

sono quindi rimasto a 10 funzioni.

U	U	U	U	U	U	U	U	U	U
1	1	0	0	0	0	0	1	1	1
0	1	1	0	1	1	0	0	0	1
0	1	1	0	0	1	1	1	0	0
0	0	1	1	0	0	0	1	1	1

FUNZIONI più rilevanti:

XOR: OR esclusivo

XNOR: negazione dell'XOR

N: negazione

AND OR XOR XNOR NAND NOR

U	U	U	U	U	U
0	0	0	1	1	1
0	1	1	0	1	0
0	1	1	0	1	0
1	1	0	1	0	0

Dopo aver considerato le 3 funzioni rilevanti e le rispettive negazioni, rimangono fuori 4 funzioni (10-6), a cui non diamo un nome

Se considerassimo delle scatole di verità con 3 ingressi e una uscita

Avremmo delle tavole di verità con 8 righe.

Quindi avremo 2^8 funzioni possibili

Formalizzando: con un numero K di ingressi avremo 2^{2^K} funzioni possibili

Chiamiamo gli ingressi A,B,C, ecc.

Il comportamento della “scatola” in uscita è dato dalla funzione del tipo $f(A,B,C)$

Quindi possiamo anche chiamare la colonna di uscita “ $f()$ ”

A	B	C	$f(A,B,C)/U$

Abbiamo visto le funzioni not,

A	U
0	1
1	0

and,

or

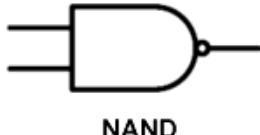
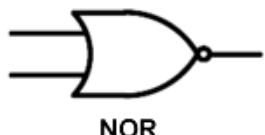
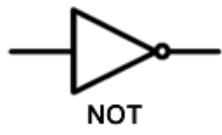
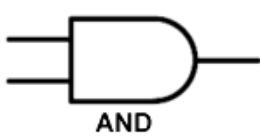
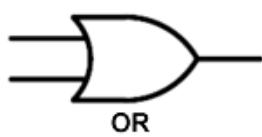
e xor

A	B	U	A	B	U	A	B	U
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

Tali funzioni possono essere combinate

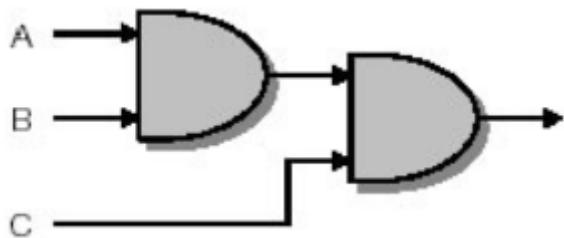
UNENDO una funzione and, per esempio con una funzione del tipo not produciamo un risultato NAND

Simboli delle funzioni:



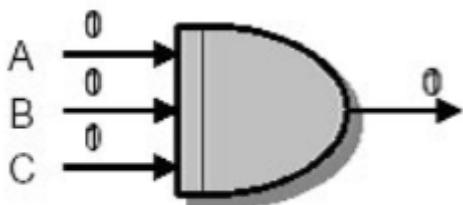
Le funzioni and, or e xor possono essere abbastanza facilmente “estese” a casi di più di 2 ingressi.

Le funzioni AND a 3 ingressi, ad esempio, si realizzano spesso come albero di funzioni AND a 2 ingressi, in questo modo:



Per quanto riguarda i simboli:

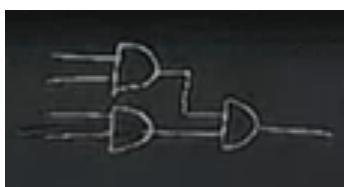
Se ci sono 2 ingressi ho due fili, se volessi rappresentare un AND con 3 ingressi, farei così (aggiungo semplicemente un filo in entrata)



Se volessi 4 ingressi farei così



Dal punto di vista pratico, una funzione AND a 4 ingressi potrei rappresentarla come la connessione di 3 funzioni AND a 2 ingressi:



Abbiamo già diverse rappresentazioni:

tavole di verità, simboli (disegnini)

altra rappresentazione: funzioni matematiche

ALGEBRA BINARIA

$$A+0=A$$

Con il simbolo + rappresentiamo la funzione or

$$A \cdot 1 = A$$

Con il simbolo \cdot rappresentiamo la funzione and

per dimostrare $A+0=A$ consultiamo la tavola di verità dell'or, fissiamo B al valore 0, non consideriamo quindi le righe in cui B vale 1, ci rimangono solo la riga 1 e 3. In queste 2 righe abbiamo la funzione identità , quello che volevamo dimostrare: $A+0=A$.

Per dimostrare $A \cdot 1 = A$ consultiamo la tavola di verità dell'and e fissiamo $B=1$, cancellando quindi le righe in cui B vale 0, anche qui abbiamo l'identità, dimostrando $A \cdot 1 = A$

And • or + xor

Altre proprietà algebriche:

$$A \cdot 0 = 0$$

A+1=1 (stiamo parlando di algebra BINARIA, nell'algebra normale $A+1$ è $A+1$, mentre qua, se guardiamo la tavola di verità dell'or e fissiamo B con il valore 1, vediamo che in uscita avremo sempre 1)

$$A+B = B+A$$

$$A \cdot B = B \cdot A$$

$$A \cdot (B+C) = A \cdot B + A \cdot C$$

(tutte le distributive e associative:

$$(a+b)+c = a+(b+c) \text{ (associatività della somma)}$$

$$(a \cdot b) \cdot c = a \cdot (b \cdot c) \text{ (associatività del prodotto)}$$

$$a \cdot (b+c) = (a \cdot b) + (a \cdot c) \text{ (distributività rispetto alla somma)}$$

$$a+(b \cdot c) = (a+b) \cdot (a+c) \text{ (distributività rispetto al prodotto)}$$

Proprietà legate alla negazione:

$$(A \text{ negato si può notare come } \neg A \text{ oppure } \bar{A})$$

Proprietà della negazione:

$$A + \bar{A} = 1 \text{ TAUTOLOGIA}$$

$$A \cdot \bar{A} = 0 \text{ NEGAZIONE}$$

$$A+A = A \text{ (idempotenza della somma)}$$

$$A \cdot A = A \text{ (idempotenza del prodotto)}$$

$$\neg(\neg A) = A \text{ DOPPIA NEGAZIONE}$$

a cosa serve questa algebra binaria? A scrivere le funzioni in forma normale (disgiuntiva o congiuntiva)

DISGIUNTIVA: $f(A,B,C) : \Sigma (\Pi \text{ term})$ (sommatoria di prodotti di termini)

CONGIUNTIVA: $f(A,B,C) = \prod (\sum \text{term})$ (produttoria di somme di termini)

Qualsiasi funzione può essere scritta in forma normale disgiuntiva O congiuntiva.

(data una qualsiasi tavola di verità io posso sempre ricondurmi ad una realizzazione che contiene solo NOT, AND e OR).

Quindi, sapendo implementare le funzioni NOT, AND e OR, sappiamo implementare qualsiasi altra funzione.

Esempio:

Prendiamo la funzione XOR(o una qualsiasi altra funzione non facente parte di quelle elementari NOT, AND, OR)

A	B	U
0	0	0
0	1	1
1	0	1
1	1	0

Si procede una riga alla volta, consideriamo una per una le righe che contengono, per esempio il valore 1.

Devo fare in modo che tale valore corrisponda ad un prodotto di termini, deve essere quindi ricavabile dalla funzione AND (prodotto).

AND •

A	B	U
0	0	0
0	1	0
1	0	0
1	1	1

In AND c'è un unico 1, ed è nell'ultima riga, mentre nella nostra funzione ce l'abbiamo in riga 2 e 3: dovrei scambiare, in AND, la 2° riga con la 4°, come posso farlo? Negando l'ingresso A, così da avere in uscita 1 alla 2° riga e 0 alla 4° riga.

Scriviamo quindi $\neg A \cdot B$

Passiamo alla riga successiva: alla riga 3, nella mia funzione ho un 1, nella funzione AND ho uno 0, devo scambiare, quindi, tra di loro la riga 3 e la riga 4, quindi negare B, così da avere 1 alla riga 3 e 0 alla riga 4.

Scriviamo quindi $A \cdot \neg B$.

Ora devo fare la somma

$$\text{XOR}(A,B) = \neg A \cdot B + A \cdot \neg B$$

Qual è l'utilità? Evitare di realizzare una funzione elementare XOR, possiamo ottenere lo stesso risultato usando 2 funzioni not, 2 funzioni and (prodotto) ed una funzione or (somma)

Altro esempio:

Prendiamo una funzione (generata randomicamente) combinatoria a 3 input e 1 output

A	B	C	U
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

A	B	C	U
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

La realizziamo passando attraverso la formula SOMMA DI PRODOTTI

Quindi ci soffermiamo di nuovo sulle righe che hanno in uscita il valore 1 e facciamo riferimento alla funzione AND a 3 ingressi

A	B	C	U
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

LA FUNZIONE AND a 3 ingressi avrebbe valore 1 solo nell'ottava posizione, nelle altre avrebbe 0

Come faccio a "spostare" l'1 dall'8° riga alla prima? (nella mia funzione ce l'ho nella 1°, nella 4° e nella 6° riga)

Nego tutti i termini, così ottengo 1 in uscita della prima riga e 0 in tutte le altre righe

Scrivo $\neg A \cdot \neg B \cdot \neg C$

Passo ora alla quarta riga della mia funzione, in cui ho il valore 1 in uscita. Considerando la funzione AND a 3 ingressi (in cui in uscita alla quarta riga c'è uno 0), cosa devo fare per avere un 1 in uscita? Nego l'ingresso A, così da avere 1 in A, 1 in B e 1 in C, che mi risulta in un 1 in uscita.

Continuiamo a comporre la funzione quindi

$\neg A \cdot \neg B \cdot \neg C + \neg A \cdot B \cdot C \dots$

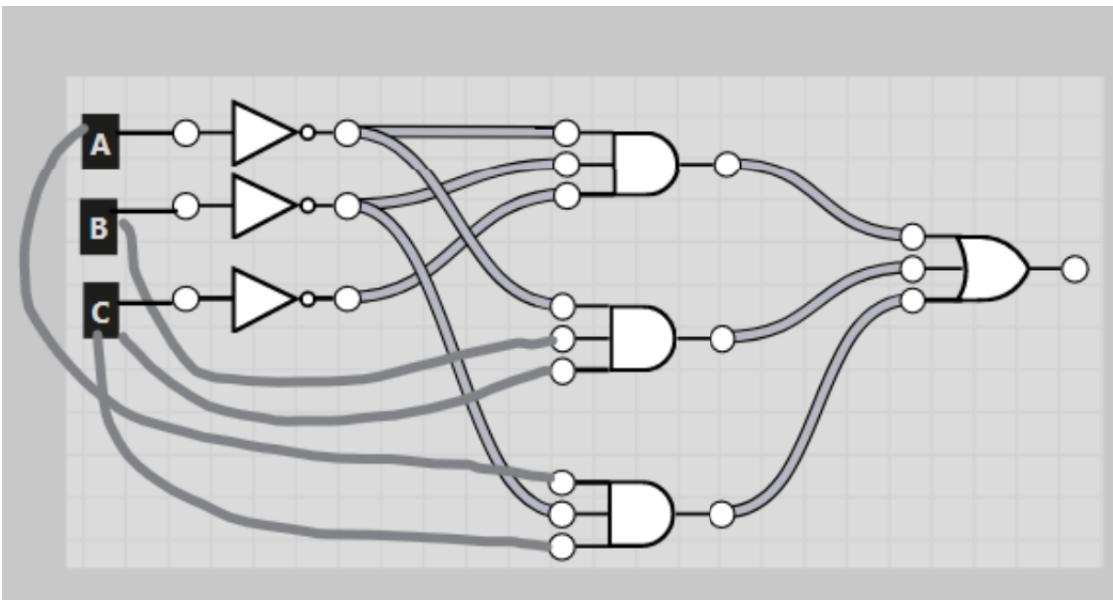
Passo adesso a considerare la sesta riga della mia funzione, in cui ho il valore 1 in uscita. Considerando la funzione AND a 3 ingressi (in cui in uscita alla sesta riga c'è uno 0), cosa devo fare per avere un 1 in uscita? Nego l'ingresso B, così da avere 1 in A, 1 in B e 1 in C, che mi risulta in un 1 in uscita.

Finisco di comporre la funzione quindi

$\neg A \cdot \neg B \cdot \neg C + \neg A \cdot B \cdot C + A \cdot \neg B \cdot C$

In generale, ciascun prodotto sarà formato dalla combinazione di 3 variabili (A,B,C) eventualmente negate. Data una funzione qualsiasi possiamo quindi risalire ad una combinazione di funzioni and, or e not.

Possiamo anche fare il disegnino della funzione che abbiamo appena disegnato come combinazione di funzioni elementari:



(INDICATIVAMENTE)

Tecnicamente avrei dovuto far partire i valori non negati da un pallino pieno prima del pallino del not, invece che farlo partire direttamente dall'etichetta)

In questa scatola ho quindi 7 funzioni elementari (3 and, 3 not e 1 or).

Un dispositivo di questo tipo consumerà 7 volte tanto rispetto a un dispositivo con una sola funzione elementare. Tanto consumo significa tanto surriscaldamento, se ho troppo consumo di energia elettrica rischio che il dispositivo si fonda e quindi diventi inutilizzabile, dovrei procedere al raffreddamento. Se il mio dispositivo funzionasse a batteria, dovrei considerare che più consuma e più diminuisce la durata della vita della batteria: se ho tanto consumo, il mio dispositivo funzionerà per poco tempo prima che si scarichi la batteria. Per tutti questi motivi è opportuno cercare di semplificare il più possibile la realizzazione fisica dei dispositivi.

Quindi quella che abbiamo realizzato è una rappresentazione che si può usare, ma sarebbe interessante arrivare a una rappresentazione minima.

Minimizzazione

Posso ridurre la complessità di queste formule?

Sì, attraverso le proprietà algebriche.

È difficile però scrivere prima tutte le formule da implementare e poi minimizzarle, non è pratico. È molto più ragionevole trovare degli algoritmi, che possono minimizzare le formule. Ci sono quindi altre rappresentazioni delle funzioni (oltre ai disegnini, alle tavole di verità, ecc) che sono utili per la realizzazione di algoritmi di minimizzazione.

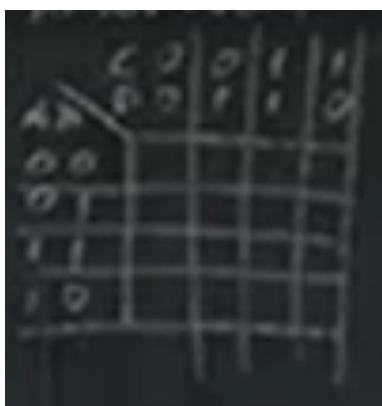
Ulteriore modo di rappresentare queste funzioni:

mappe di Karnaugh

a differenza delle tavole di verità, queste mappe sono bidimensionali (non uni)

suddividiamo l'insieme degli ingressi in 2 sottoinsiemi

ad es. se abbiamo una funzione a 4 ingressi, le combinazioni dei valori possibili in ingressi A e B li scriviamo sulle righe, le combinazioni delle altre due variabili (C e D) invece sulle colonne



esempio

(nei quadratini vuoti ci saranno le uscite) → la riga ci dirà la combinazione sui valori A e B, la colonna ci dice la combinazione dei valori su C e D.

Particolarità di questo tipo di rappresentazione è il fatto che, quando scriviamo tutte le combinazioni possibili, mettiamo prima 1 1 e poi 1 0, così che, tra ogni configurazione abbia distanza di Hamming 1 rispetto alla configurazione della riga precedente.

Questa sequenza particolare viene chiamata codice Gray

Riempiamo “a caso” le celle vuote

cd ab	00	01	11	10
00	1	0	0	1
01	1	1	0	0
11	1	0	0	0
10	1	0	0	1

Potrei usare lo stesso metodo per definire una rappresentazione in forma normale Σ (Π)

Otterremmo 7 prodotti, quindi un or a 7 ingressi e 7 and a 4 ingressi + qualche not (da calcolare)

Per "semplificare" la rappresentazione dobbiamo cercare una aggregazione di caselle adiacenti che contengano tutte il valore 1 e il cui numero sia una potenza di 2 (sono caselle "adiacenti" anche la quarta e la prima casella di una data colonna, poiché è un "ciclo unico" basato sul fatto che ci sia distanza di hamming 1, infatti se prendiamo la parte dei valori possibili di ab e guardiamo l'ultima cella: 10, notiamo che ha distanza di hamming 1 rispetto alla prima, la prima ha distanza di hamming 1 rispetto alla seconda e così via... ogni cella ha distanza di Hamming 1 rispetto alla cella che viene subito dopo e subito prima, e il ciclo si ripete)

LA PRIMA COLONNA SODDISFA il nostro requisito

cd ab	00	01	11	10
00	1	0	0	1
01	1	1	0	0
11	1	0	0	0
10	1	0	0	1

Possiamo dunque scrivere direttamente una formula semplificata rispetto a cosa avremmo scritto usando la tavola di verità

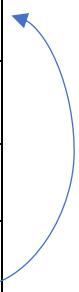
Posso scrivere $\neg C \cdot \neg D$ (perché mi accorgo che i valori 1 in uscita non dipendono dalle combinazioni in entrata A e B poiché, qualsiasi valore assumano A e B, l'uscita vale sempre 1) per rappresentare tutte e 4 le celle della prima colonna

Se invece avessi dovuto scrivere i valori 1, uno per volta, come quando usavo la tavola di verità, avrei dovuto scrivere

$$\neg A \cdot \neg B \cdot \neg C \cdot \neg D + \neg A \cdot B \cdot \neg C \cdot \neg D + A \cdot B \cdot \neg C \cdot \neg D + A \cdot \neg B \cdot \neg C \cdot \neg D$$

Per quello che abbiamo spiegato in precedenza, quindi, anche queste due caselle sono adiacenti

cd ab	00	01	11	10
00	1	0	0	1
01	1	1	0	0
11	1	0	0	0
10	1	0	0	1



Ma consideriamo che anche le colonne possono essere adiacenti (nei valori possibili di c e d ho 00 nella prima cella ed 10 nella seconda: distanza di Hamming 1)

cd ab	00	01	11	10
00	1	0	0	1
01	1	1	0	0
11	1	0	0	0
10	1	0	0	1

Quindi posso individuare un gruppo di 4 caselle **adiacenti**: (rispetto alle colonne e rispetto alle righe)

	cd ab	00	01	11	10
00		1	0	0	1
01		1	1	0	0
11		1	0	0	0
10		1	0	0	1

Quindi posso scriverlo usando un prodotto di 2 termini (non considero, quindi, neanche c, poiché qualsiasi valore c assuma, in uscita ho sempre un 1)

Posso scrivere quindi direttamente

$$\neg B \cdot \neg D$$

Aggiungo questo prodotto di due termini alla realizzazione iniziata in precedenza

$$\neg C \cdot \neg D + \neg B \cdot \neg D$$

Abbiamo quindi "coperto" per adesso, tutti gli "uni" tranne **uno**.

	cd ab	00	01	11	10
00		1	0	0	1
01		1	1	0	0
11		1	0	0	0
10		1	0	0	1

Questo 1 lo posso "accoppiare" con l'1 vicino (da cui ha distanza di Hamming=1)

La differenza tra queste due celle sta nel valore di D, che quindi, in base a ciò che ho detto prima, posso "cancellare"/non prendere in considerazione), scrivo quindi questi due elementi (le due caselle contenenti **1 1**) come $\neg A \cdot B \cdot \neg C$.

COMPLETIAMO QUINDI LA rappresentazione/realizzazione

$$\neg C \cdot \neg D + \neg B \cdot \neg D + \neg A \cdot B \cdot \neg C.$$

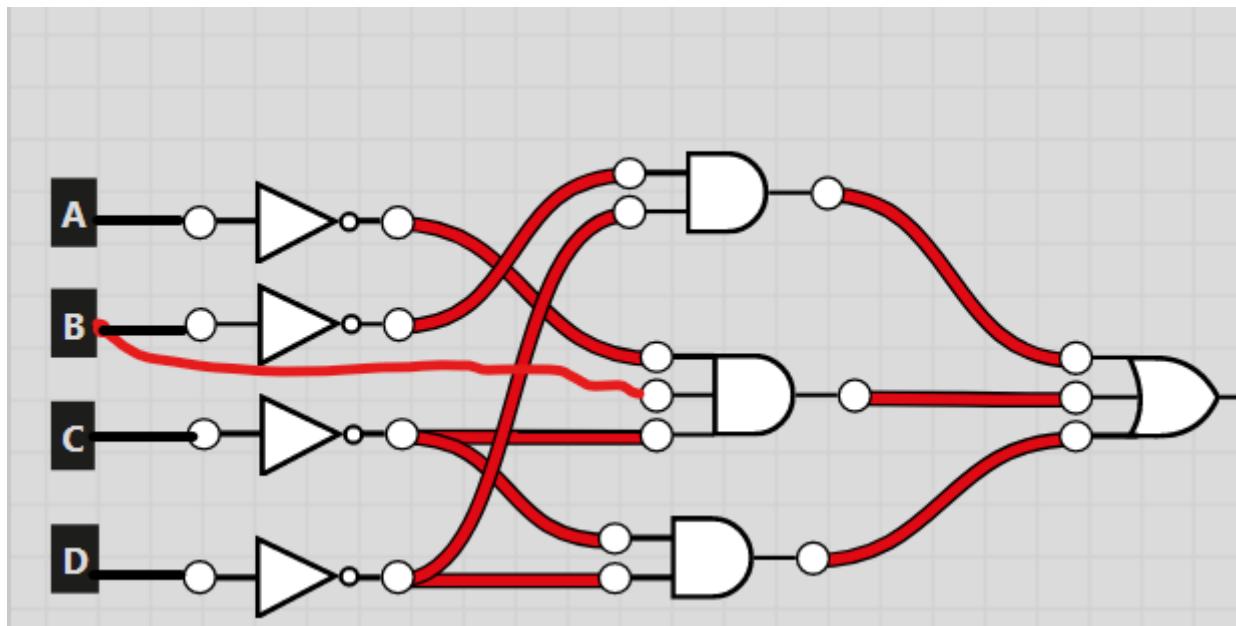
Quindi stiamo usando:

- 4 funzioni NOT (tutte le variabili sono negate almeno una volta)

- 1 funzione OR a 3 ingressi (per sommare i 3 blocchi, i 3 prodotti)
- 1 funzione AND a 3 ingressi + 2 funzioni AND a 2 ingressi

ABBIAMO TROVATO LA RAPPRESENTAZIONE MINIMA, che non può essere ulteriormente semplificata.

LA MAPPA che rappresenterà la realizzazione fisica di tale funzione è più o meno questa



Realizzazione in 3 livelli (not, and, or)

Minimizzare il numero di livelli corrisponde a minimizzare i tempi di ritardo di tali realizzazioni

(essendo dei dispositivi fisici non potranno reagire immediatamente alla variazione dei segnali d'ingresso, avranno un certo ritardo, quindi minimizzando il numero di componenti, minimizzo il ritardo dell'intero dispositivo (l'intera funzione)).

È proprio necessario usare prima le funzioni not, poi le and, e poi le or, per realizzare tali dispositivi?

Per rispondere a simili domande possiamo far ricorso alle leggi di De Morgan

$$A+B = \neg(\neg A \cdot \neg B)$$

POSSO REALIZZARE DEI CIRCUITI IN LOGICA A 3 LIVELLI non per forza usando le funzioni not, and, or ma semplicemente usando solo funzioni NAND.

Quindi, dal punto di vista pratico, ho bisogno di un solo tipo di dispositivo, non ho bisogno di tre tipi diversi di dispositivi

Se analizziamo anche solo la funzione sopra riportata vediamo che ci sono diverse "strade" per quanto riguarda la risposta del dispositivo a valori di ingressi verso valori di uscita, ad es:

dall'input B fino all'output si "toccano" solo 2 dispositivi: una funzione and ed una funzione or, mentre se invece prendessimo $\neg B$ come esempio, avremmo un percorso più lungo, che coinvolge 3 dispositivi (1 funzione not, 1 funzione and ed 1 funzione or, in questo caso).

Per stabilire il ritardo di una certa realizzazione conta, però, il "percorso di lunghezza massima"

Potrebbero esserci dei casi, infatti, come nella raffigurazione qua sopra, in cui il cambiamento del valore di B in entrata potrebbe ripercuotersi più rapidamente sull'uscita rispetto al cambiamento di A, perché la sua "strada" precede solo due unità di tempo di ritardo (coinvolge solo due dispositivi) mentre nella strada di A ce ne sono 3 (3 funzioni, e quindi più ritardo).

Questa realizzazione viene chiamata, appunto, a "3 livelli" perché 3 è il numero massimo di dispositivi da attraversare per passare da un qualunque ingresso verso l'uscita.

Usando delle applicazioni possiamo produrre, in maniera algoritmica, la realizzazione minimale di un certo dispositivo a partire dalla sua definizione.

Sono stati sviluppati, quindi, degli altri tipi di rappresentazioni che sono più comode da manipolare all'interno di un algoritmo.

BDD (Binary Decision Diagrams)

Devo partire da un ordinamento delle variabili di ingresso (es: quello alfabetico, altri tipi di ordinamento)

Prendiamo questo ordinamento come esempio:

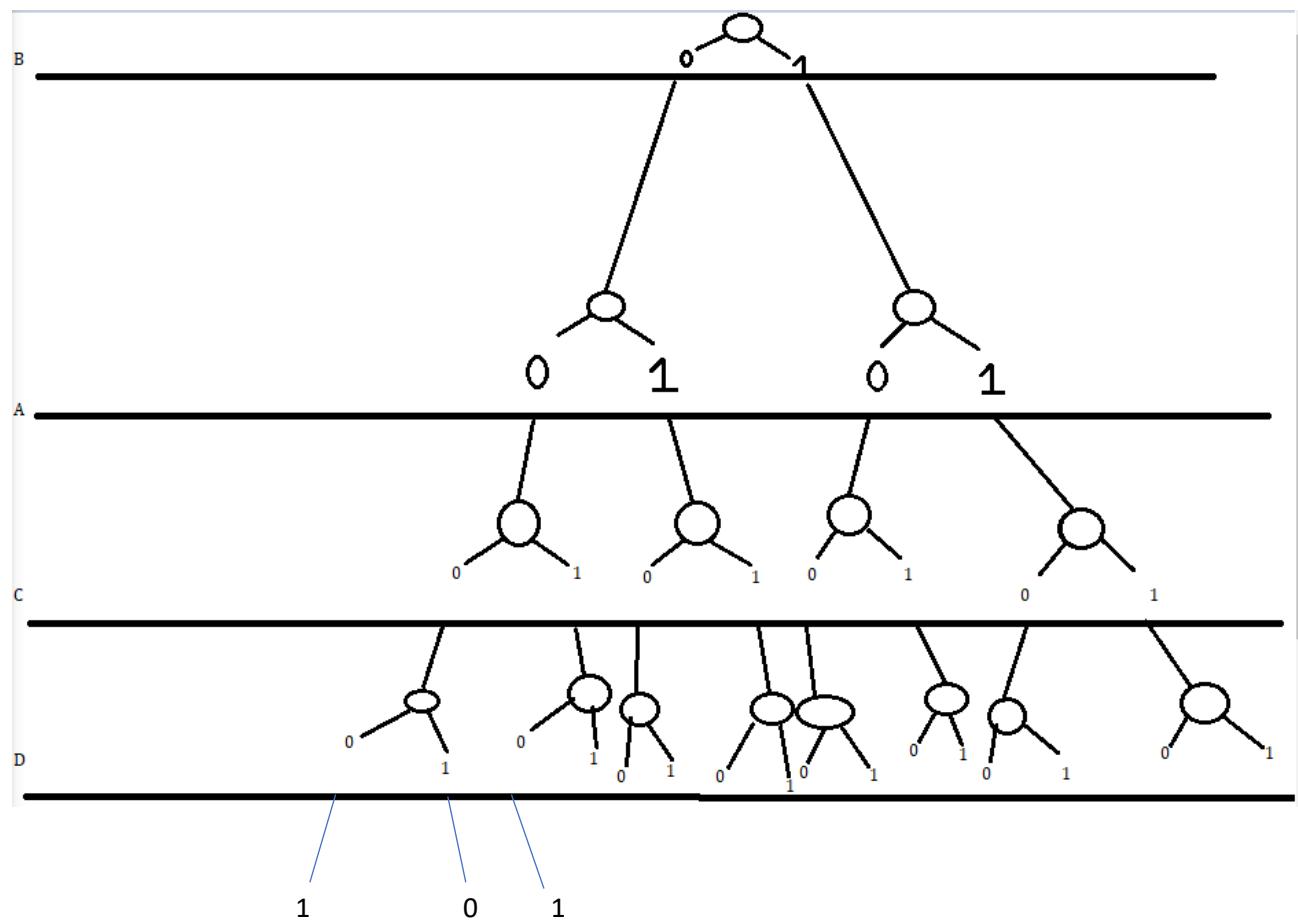
B

A

C

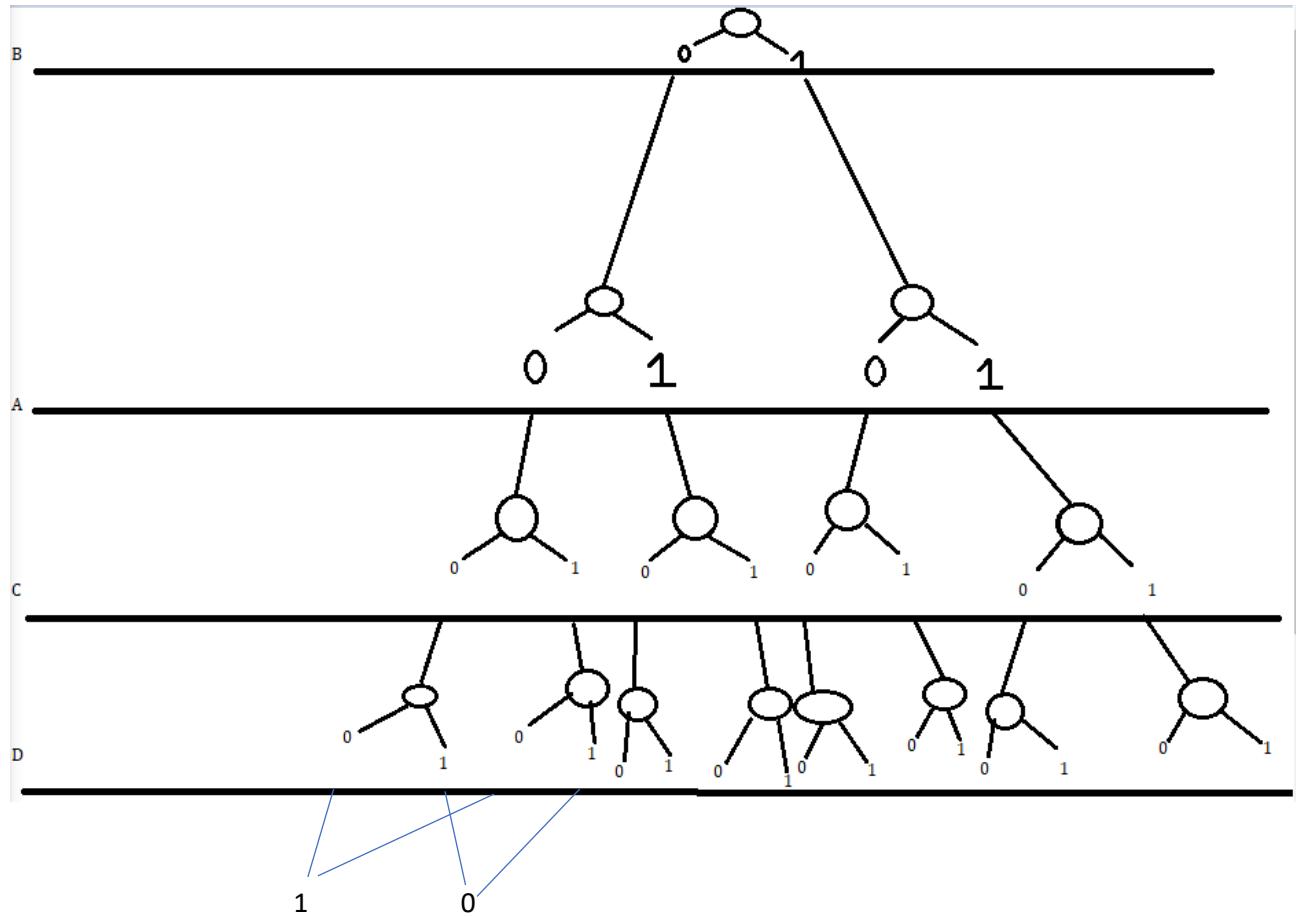
D

PER OGNI VALORE D'INGRESSO DOBBIAMO DIRE QUANTO VALE L'USCITA

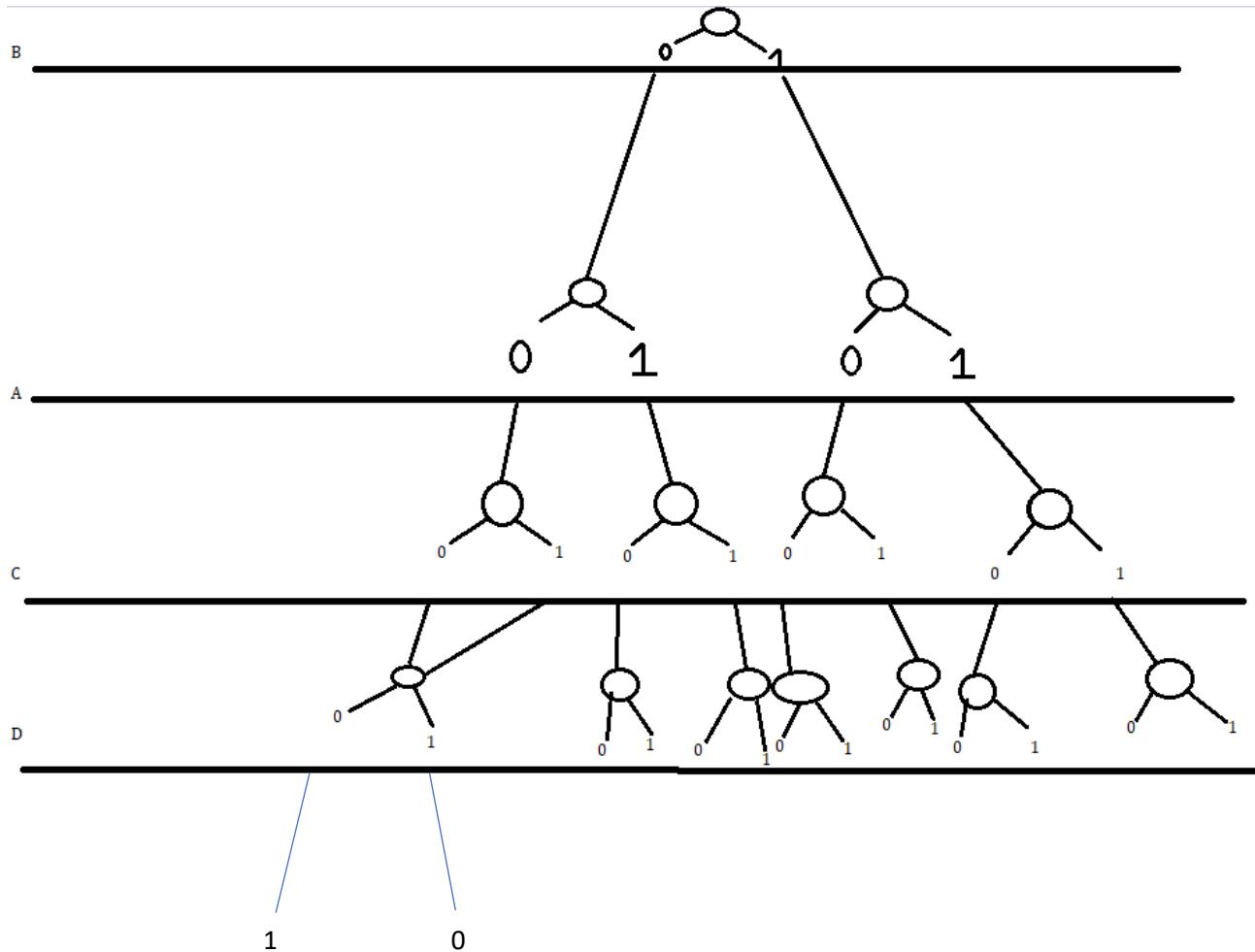


Eccetera

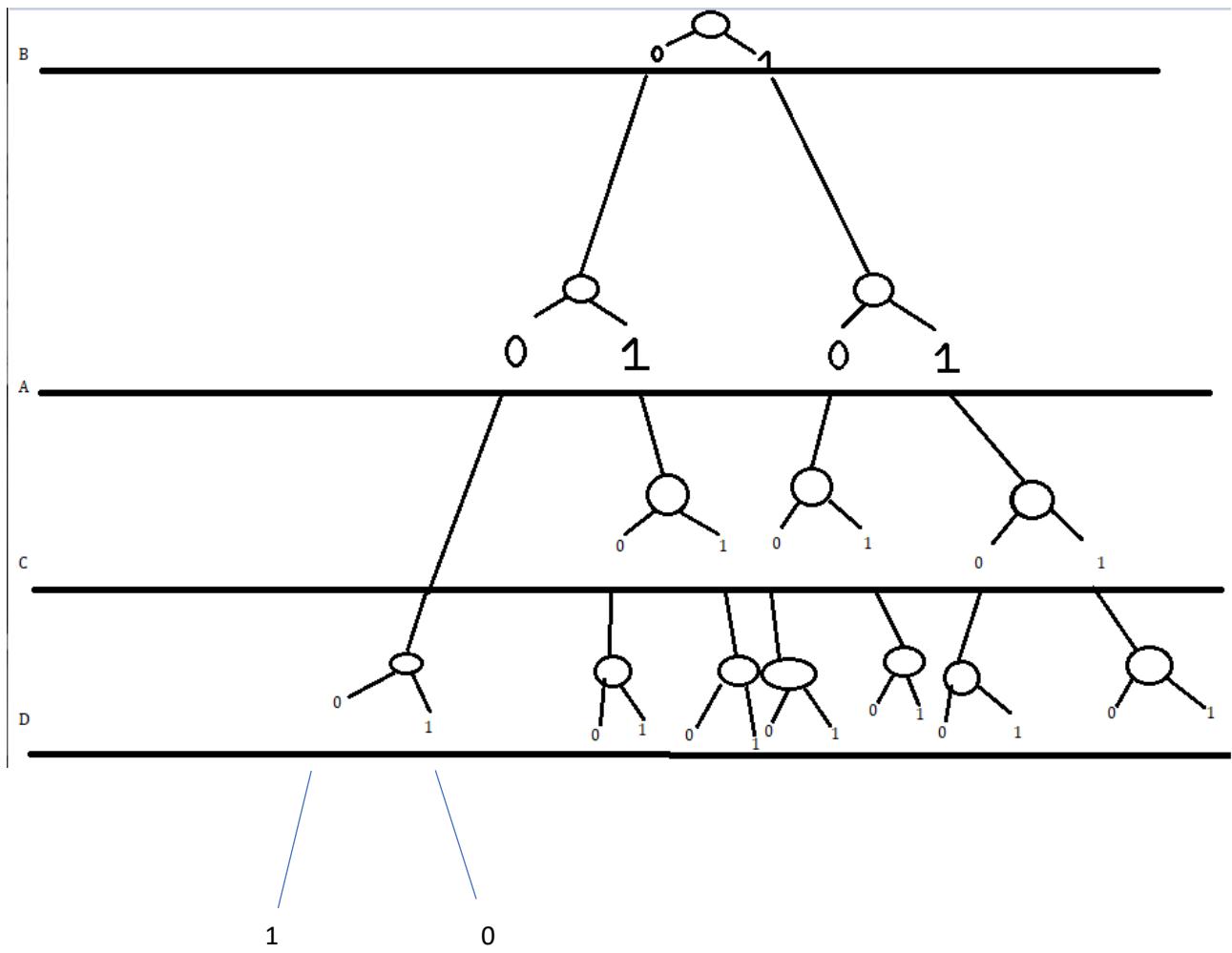
Più semplicemente posso collegare i valori di uscita delle diverse combinazioni ad un 1 e uno 0 che scrivo una sola volta, es:



Ma, semplificando ancora, posso “collegare” già da prima i nodi, eliminando, in questo caso, il secondo nodo da cui si diramano i valori di D, perché portano allo stesso risultato del primo nodo, sono quindi due copie identiche della stessa cosa (nel primo caso, quando D è 0, l’uscita è 1, quando D è 1 l’uscita è 0, analogamente, anche nel secondo caso, quando D è 0 l’uscita vale 1, quando D è 1 l’uscita vale 0)



A questo punto, vedo che con $C=0$ vado nel primo nodo di D, con $C=1$ analogamente nel primo nodo di D, mi accorgo che conoscere i valori di C, in questo caso, non mi serve:



C'è quindi la possibilità di semplificare notevolmente tale rappresentazione

(partendo da una rappresentazione di un grafo completo, possiamo eliminare via via tutte le parti ridondanti, eliminando le duplicazioni, arrivando quindi ad una rappresentazione compatta in cui ad ogni livello ci può essere un nodo oppure no; se c'è un nodo ad un certo livello significa che il valore della variabile corrispondente al livello è SIGNIFICATIVO per determinare il valore in uscita, mentre se non c'è il nodo significa che il valore di tale variabile è irrilevante rispetto al valore che verrà prodotto in uscita).

Terminologia: le "linee" che collegano i nodi sono dette "archi".

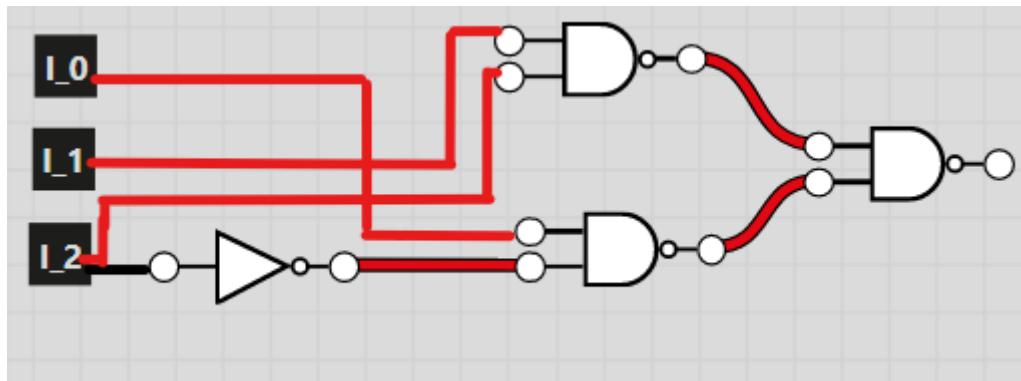
Ciò che abbiamo appena spiegato, anche se "sommariamente" è come funziona un sistema di CAD ELETTRONICO (applicazioni che creano la configurazione in digitale delle realizzazioni fisiche in maniera automatica-algoritmica)

Prendiamo come esempio una funzione con tale rappresentazione in mappa di Karnaugh

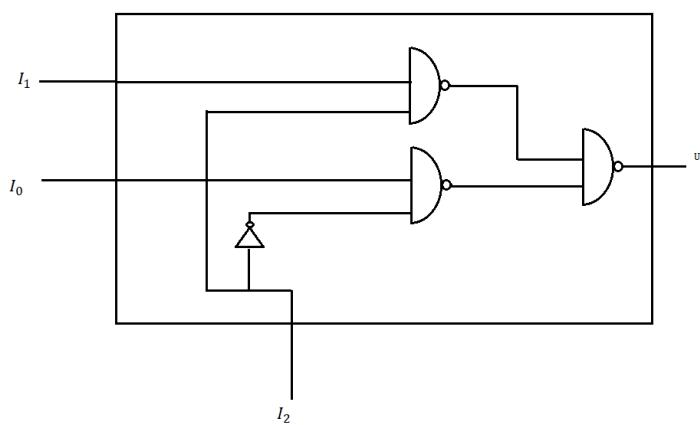
I_2		
$I_0 \ I_1$	0	1
0 0	0	0
0 1	0	1
1 1	1	1
1 0	1	0

$$\text{USCITA} = I_1 \cdot I_2 + I_0 \cdot \neg I_2$$

Realizzazione fisica in logica a tre livelli (NAND):



Ridisegniamo la realizzazione fisica di tale funzione in un altro modo:

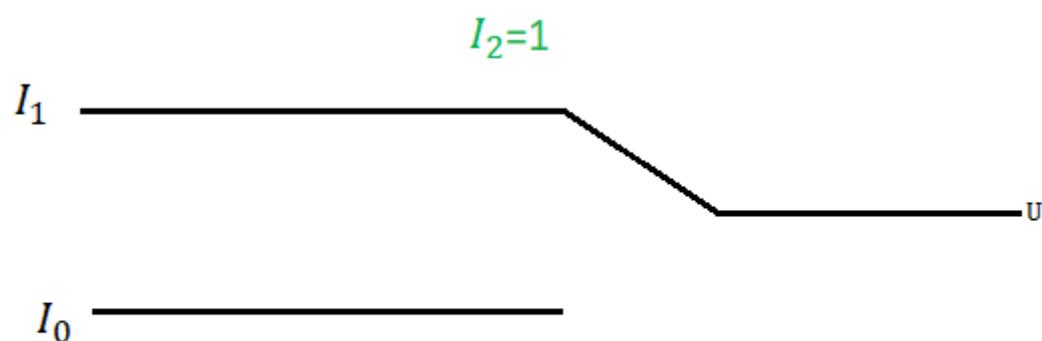
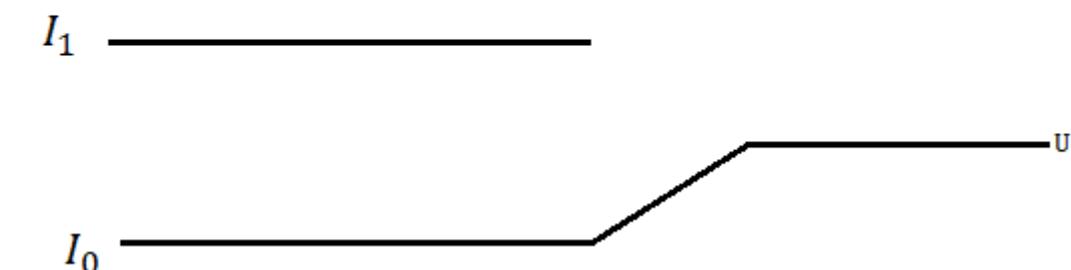


Il valore di uscita U è identico al valore di ingresso I_0 quando I_2 assume il valore 0, mentre è identico al valore di ingresso I_1 quando I_2 assume il valore 1.

Ciò posso vederlo anche in questo modo:

usando un commutatore riporto in uscita o l'ingresso I_0 o l'ingresso I_1 , in base al **valore** di ingresso di I_2 :

$I_2=0$



Un circuito del genere viene chiamato MUX.

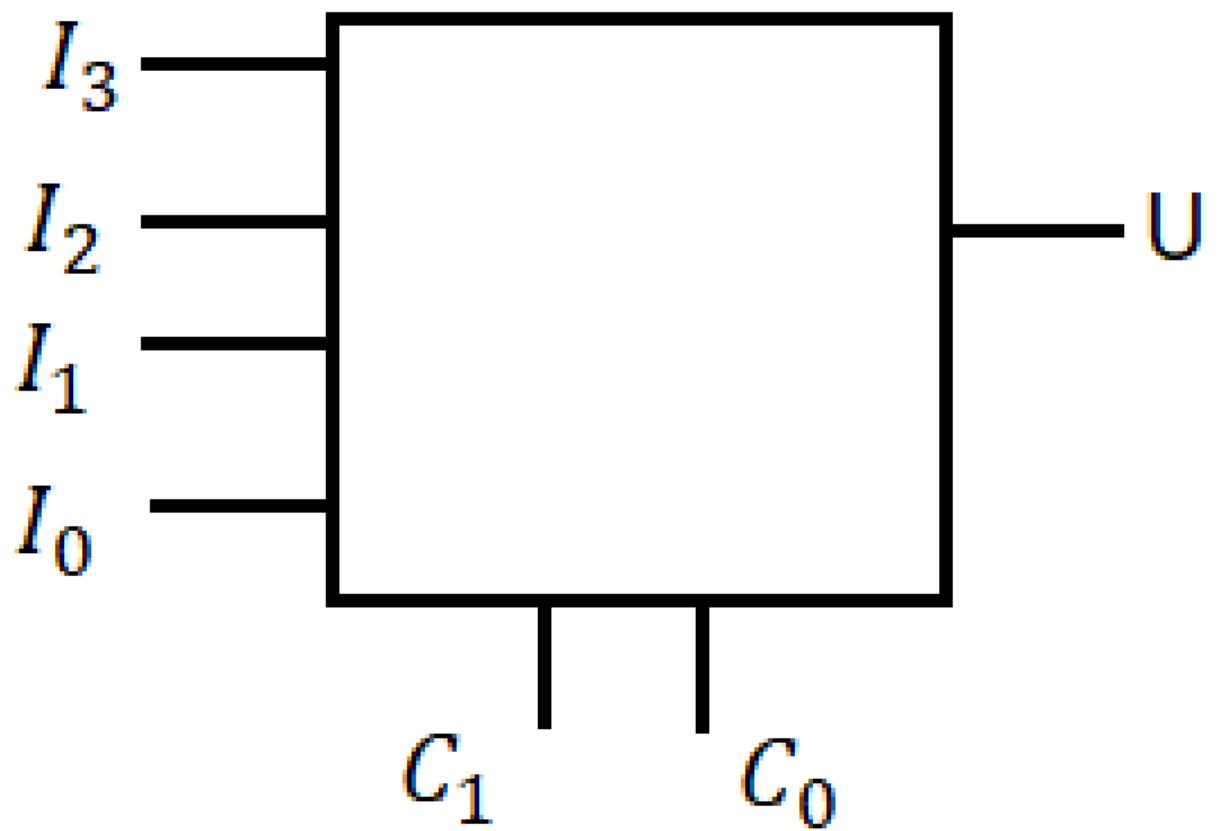
Possiamo chiamare I_2 "C", che sta per controllo (perché in base al suo valore viene determinata la posizione del commutatore, che collegherà all'uscita o l'ingresso I_1 oppure l'ingresso I_0 .)

Questo caso specifico si chiama multiplexer "a 2 vie".

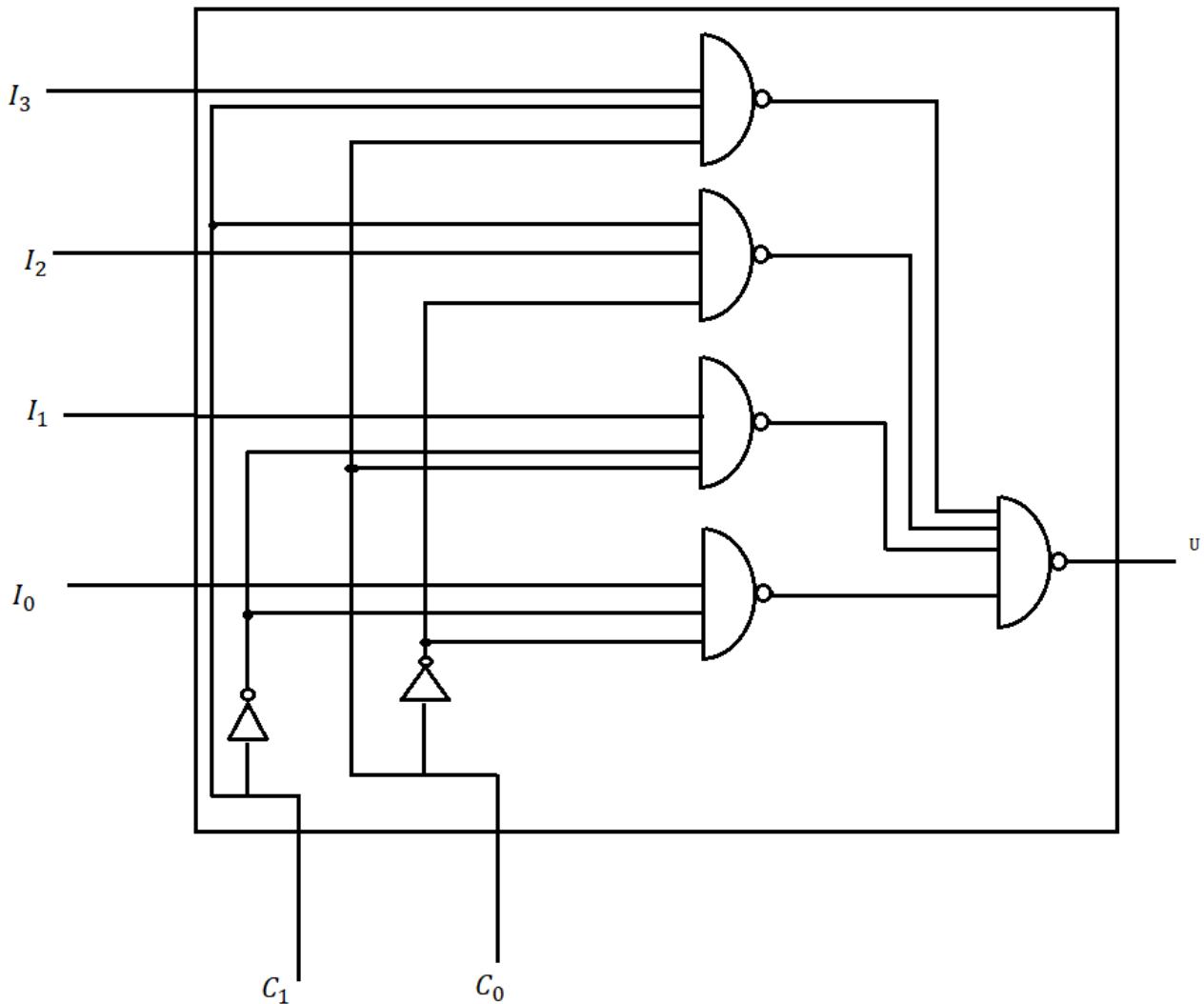
Tale idea si può estendere: si possono aumentare i numeri di ingressi tra cui "scegliere".

Ad esempio posso passare ad un multiplexer a 4 ingressi.

Avrei quindi 4 ingressi + 2 ingressi di controllo: ad esempio, avendo nei controlli il valore 00 collego l'ingresso I_0 , avendo 01 collego I_1 , avendo 10 collego I_2 ed avendo 11 collego I_3 :



Come realizzo l'implementazione, in questo caso? (con 4 variabili)



(sto disegno dovrei rifarlo con colori diversi, per far capire bene dove vanno i fili)

VOLENDO AUMENTARE ANCORA GLI INGRESSI (passare ad una realizzazione ad 8 ingressi) devo duplicare il blocchetto che ho ed aggiungere un altro ingresso di controllo e quindi aumentare ovviamente gli ingressi della funzione NAND finale.

Nota bene: la funzione in questione, quindi, si chiama “funzione multiplexer”.

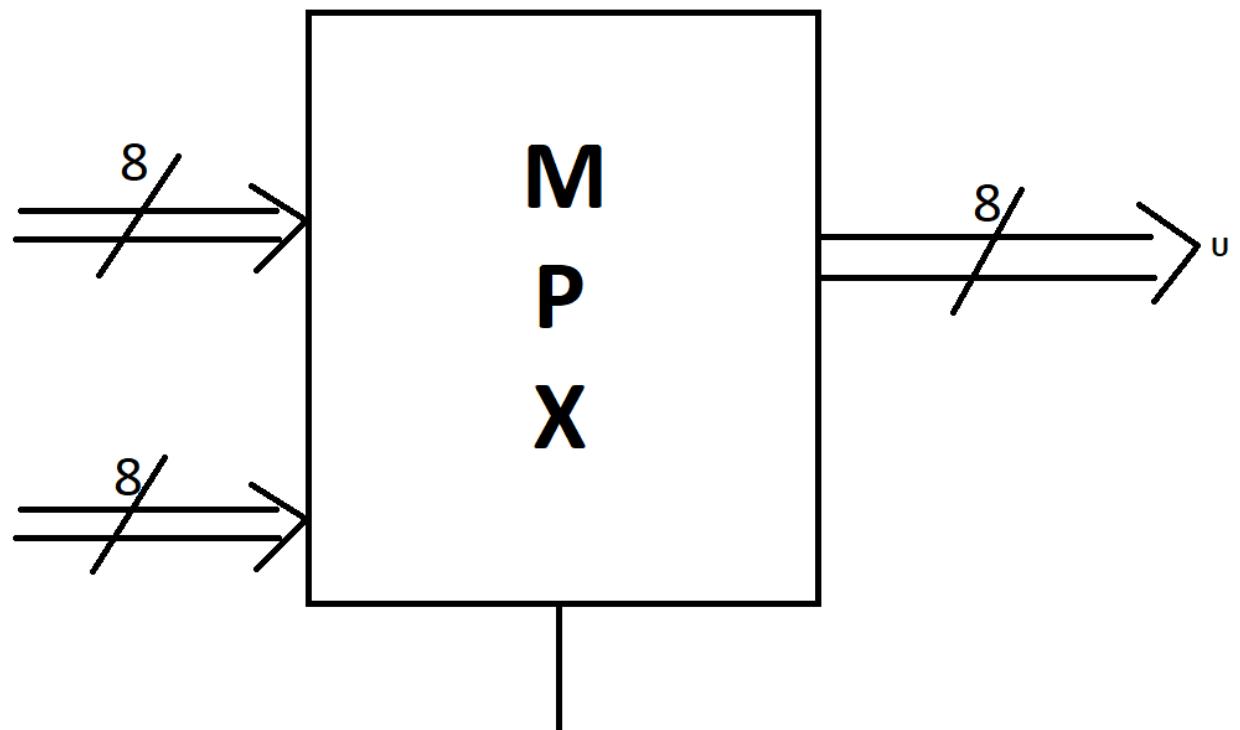
Alla funzione multiplexer viene associato un simbolo grafico



questa è, per esempio, la rappresentazione sintetica di un dispositivo multiplexer a 2 ingressi ed una uscita.

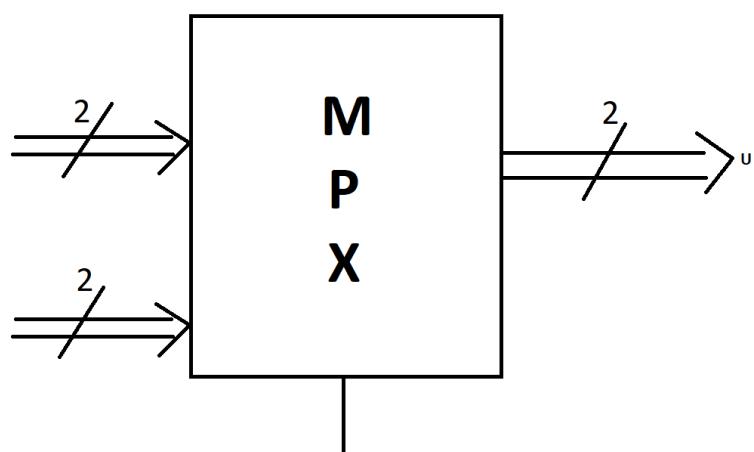
Altro modo per estendere il multiplexer (oltre a, come già visto, aggiungere ingressi): replicare tutta la struttura più volte, unendo tra di loro tutti gli ingressi di controllo.

Dal punto di vista grafico questa estensione si può rappresentare così:

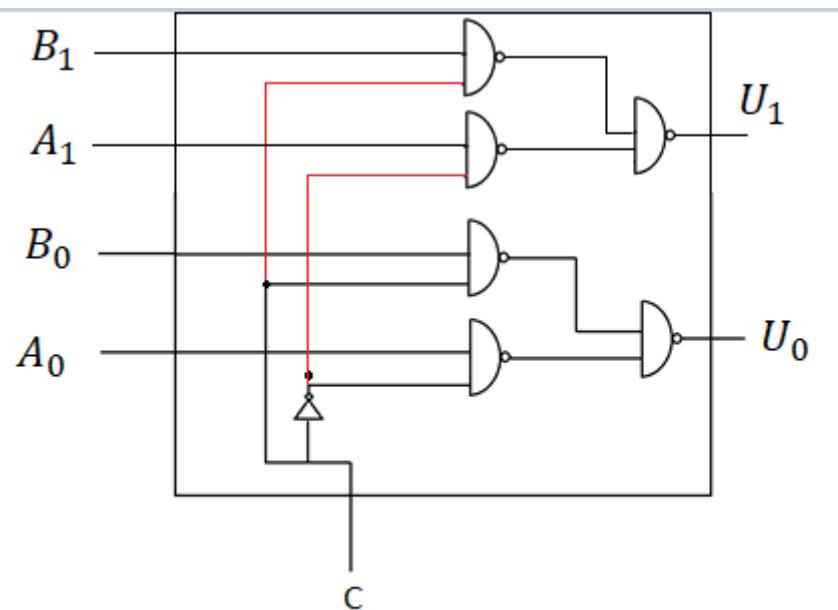


Dove il numero rappresenta la molteplicità dei fili, e quindi il numero di bit che possono essere commutati contemporaneamente (8, in questo caso)

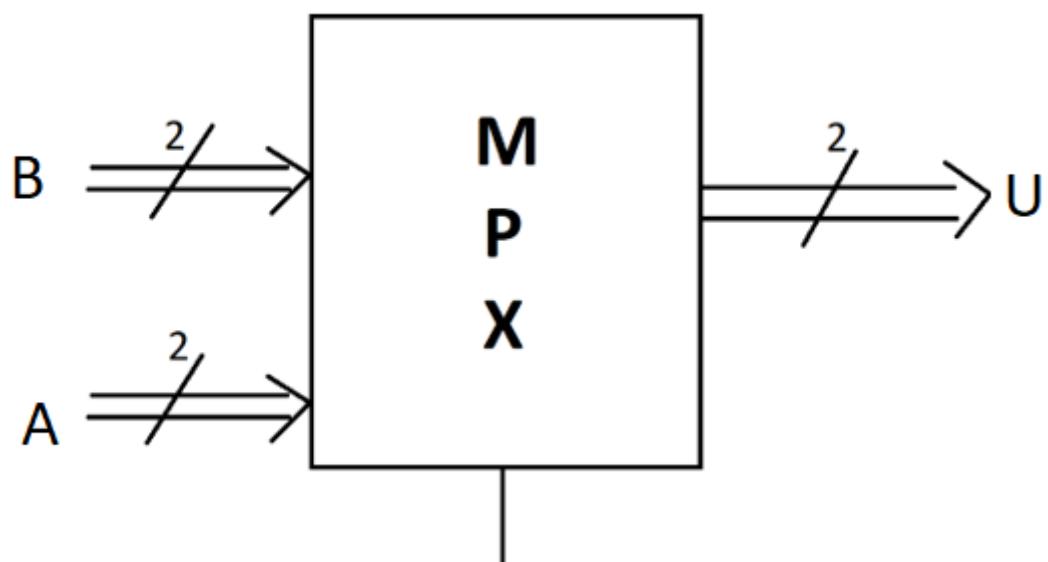
Es: facciamo un esempio con 2 bit



Come si fa la realizzazione fisica di un circuito multiplexer a 2 vie in cui ciascuna informazione è codificata su 2 bit?

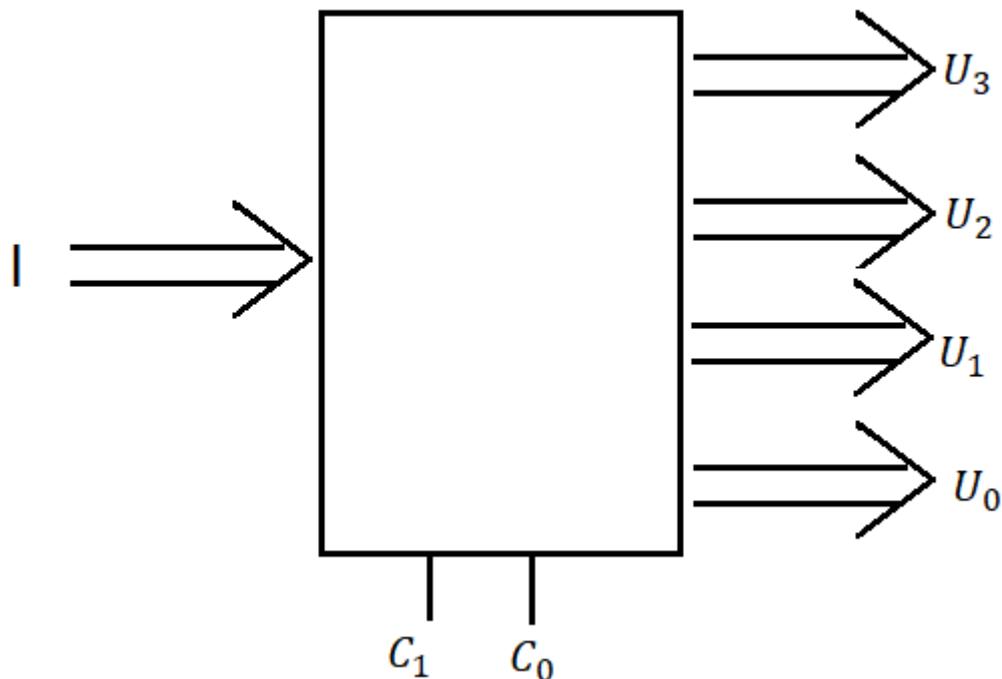


Quindi con A_0 ed A_1 rappresento l'ingresso A, mentre con B_0 e B_1 rappresento l'ingresso B



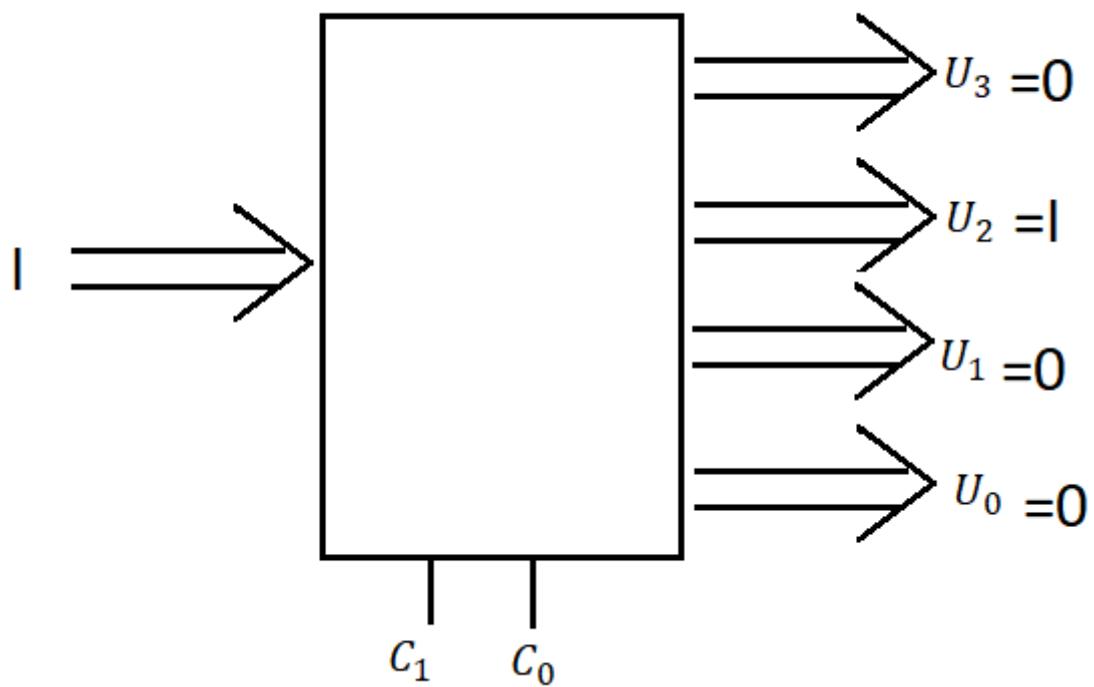
Altri dispositivi della stessa “famiglia”:

demultiplexer (fa l'operazione inversa rispetto al multiplexer: partendo da 1 solo ingresso produce n uscite)



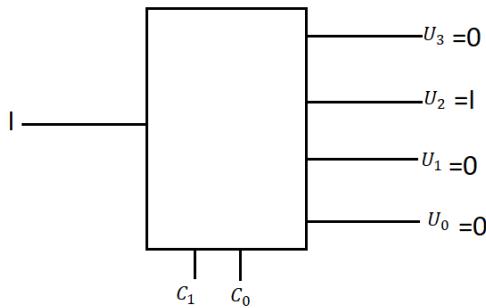
Nelle uscite “non selezionate” da C_0 e C_1 avremo valore 0.

Ad esempio, avendo $C_0=0$ e $C_1 = 1$ identifichiamo l'uscita U_2 (che quindi avrà valori identici ad I), mentre le altre uscite riporteranno il valore 0

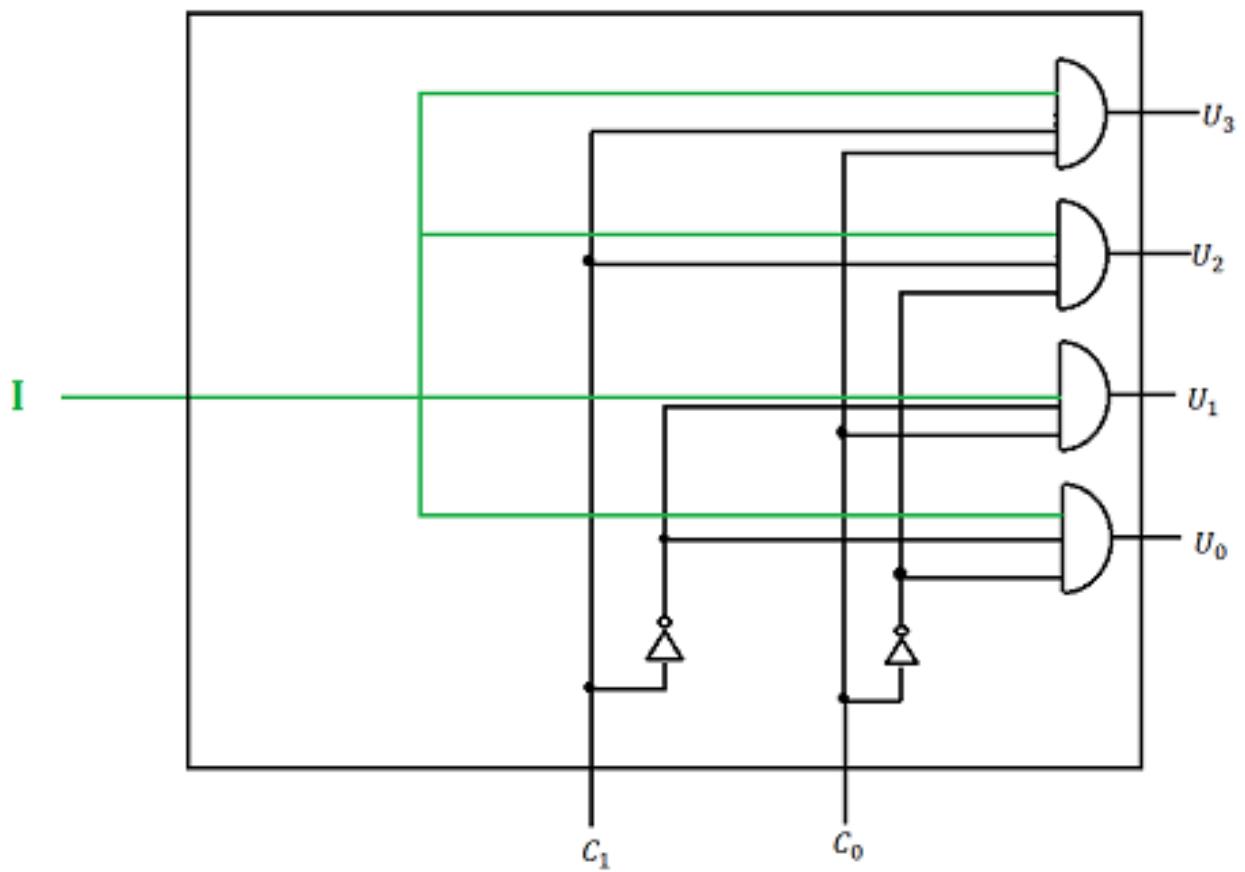


COME REALIZZIAMO UN DEMULTIPLEXER IN LOGICA A 3 LIVELLI (NAND)?

Facciamo l'esempio con un demultiplexer sempre a 4 uscite ma in cui ogni informazione è codificata su 1 bit:



Come potremmo fare? Avendo chiara la struttura del multiplexer dovrebbe bastare farla funzionare “alla rovescia”



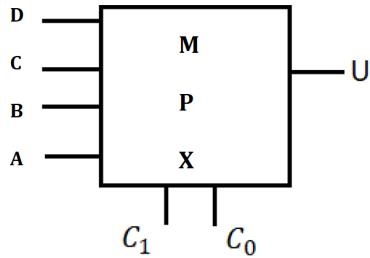
Dando un'occhiata alle tavole di verità che possiamo produrre sulla base di tale realizzazione vediamo che, per esempio, PER L'USCITA U_0 il tutto funziona esattamente come dovrebbe:

quando $C_1 = 0$ e $C_0 = 0$ l'uscita assume il valore di I (0 nella **prima riga**, 1 nella **quinta**), in tutti gli altri casi, in cui C_1 e C_0 assumono un valore diverso da 00, allora all'uscita è assegnato il valore 0 (come abbiamo spiegato in precedenza)

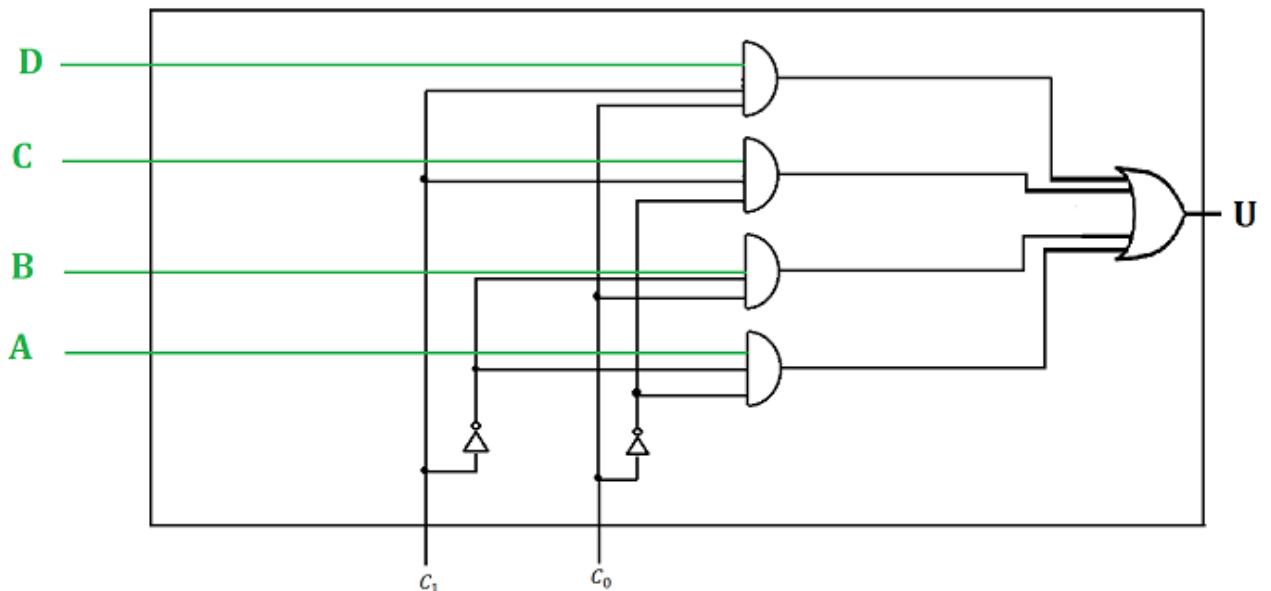
I	C_1	C_0	U_0
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Disegniamo adesso, per comodità, un multiplexer che è l'esatto inverso del demultiplexer appena disegnato

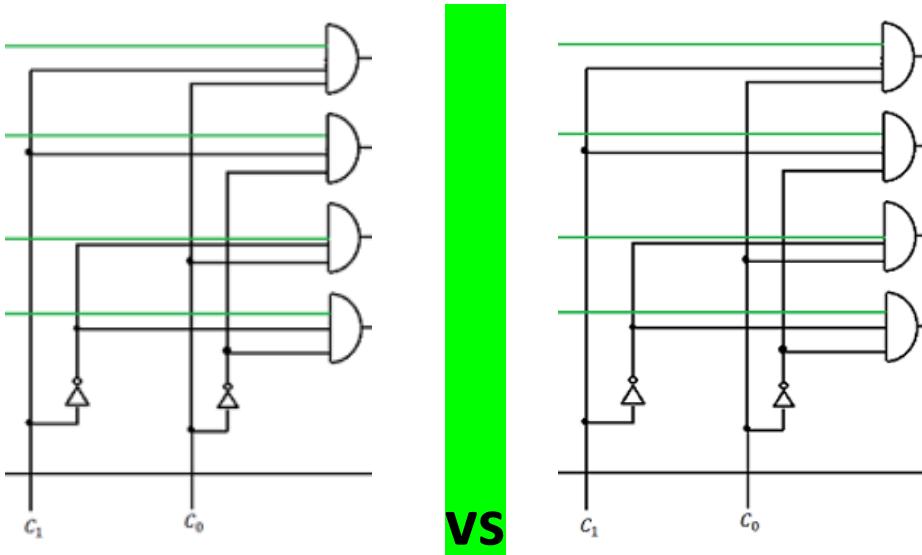
(così da avere una estensione a 4 vie di un multiplexer con codifica su un singolo bit)



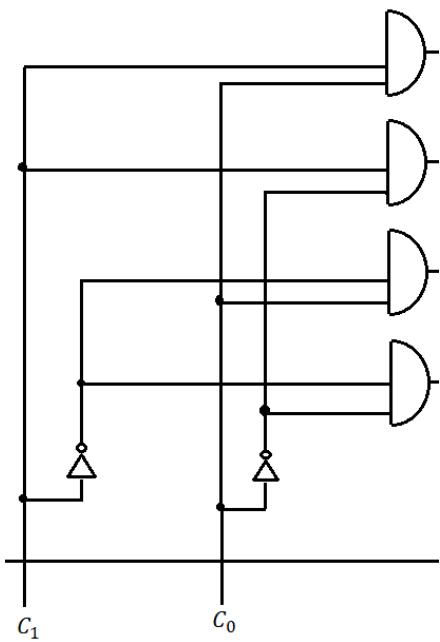
Sarebbe più o meno fatta così:



Guardando demultiplexer e multiplexer si nota che la parte delle variabili di controllo C (tutta la struttura in nero) è identica



Quindi una parte del demultiplexer è identica a una parte del multiplexer (hanno una parte della realizzazione in comune). Questa parte in comune è chiamata decoder



Un dispositivo di tipo decoder ha solamente ingressi di controllo (non ha ingressi principali) e le uscite prendono tutte il valore 0 tranne quella selezionata da una combinazione di controllo.

CIRCUITI NUMERICI

SOMMA

$A_1 A_0$	B_1	0	0	1	1
	B_0	0	1	1	0
0	0				
0	1				
1	1				
1	0				

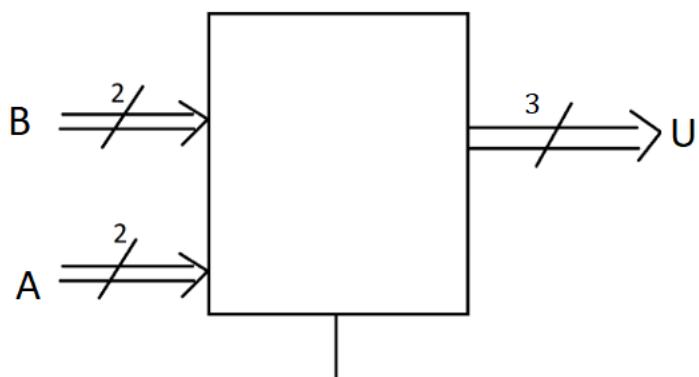
(i valori possibili sono 00,01,11,10 quindi si possono rappresentare i valori compresi tra 0 e 3) → quindi il massimo risultato che posso avere da queste operazioni di somma è 6, questo valore lo dovrò rappresentare su 3 bit (perché 2 bit sono abbastanza solo per rappresentare valori da 0 a 3)

Compattando la notazione possiamo scrivere tutti e 3 i bit del risultato all'interno di ogni singola casella

$A_1 A_0$	B_1	0	0	1	1
	B_0	0	1	1	0
0	0	000	001	011	010
0	1	001	010	100	011
1	1	011	100	110	101
1	0	010	011	101	100

Ricordandoci che i valori 0,1,2,3 vengono codificati rispettivamente da 00,01,10,11 → quindi a 3 bit saranno codificati come 000,001,010,011
 4, 5, 6 di conseguenza saranno 100,101,110

Per quanto riguarda la realizzazione fisica, quindi, realizzeremo un dispositivo che ha come ingressi due "fasci" da 2 bit ciascuno, ed in uscita invece un fascio da 3 bit (i valori del risultato saranno codificati su 3 bit, mentre quelli di ingresso su 2)



$B_1 \backslash B_0$	0	0	1	1
$A_1 A_0$	0	1	1	0
0 0	000	001	011	010
0 1	001	010	100	011
1 1	011	100	110	101
1 0	010	011	101	100

Avendo presente la mappa di Karnaugh utilizziamo il nostro metodo di sintesi: per prima cosa dobbiamo trovare gruppi di caselle **adiacenti** (formati da un numero di caselle che sia una potenza di 2) che riportino il valore 1 nel bit meno significativo

Le quattro caselle sottolineate le posso quindi rappresentare come

$$A_0 \cdot \neg B_0$$

Le altre 2 sopra e 2 sotto le possiamo scrivere sotto forma di prodotto di termini in questa maniera:

$$\neg A_0 \cdot B_0$$

Sommando, quindi:

$$U_0 = A_0 \cdot \neg B_0 + \neg A_0 \cdot B_0$$

Ora guardiamo quelle che hanno il valore 1 nel secondo bit

B_1	0	0	1	1	
B_0	0	1	1	0	
$A_1 A_0$	0 0	000	001	011	010
0 1	001	010	100	011	
1 1	011	100	110	101	
1 0	010	011	101	100	

$$A_1 \cdot \neg B_0 \cdot \neg B_1$$

B_1	0	0	1	1	
B_0	0	1	1	0	
$A_1 A_0$	0 0	000	001	011	010
0 1	001	010	100	011	
1 1	011	100	110	101	
1 0	010	011	101	100	

Posso anche considerare questa adiacenza

$$A_1 \cdot \neg A_0 \cdot \neg B_1$$

ORA CONSIDERIAMO LE 3 CASELLE RIMASTE

B_1	0	0	1	1	
B_0	0	1	1	0	
$A_1 A_0$	0 0	000	001	011	010
0 1	001	010	100	011	
1 1	011	100	110	101	
1 0	010	011	101	100	

$$\neg A_1 \cdot \neg A_0 \cdot B_1 + \neg A_1 \cdot \neg B_0 \cdot B_1$$

Sommendo il tutto fino ad ora siamo a

$$A_1 \cdot \neg B_0 \cdot \neg B_1 + A_1 \cdot \neg A_0 \cdot \neg B_1 + \neg A_1 \cdot \neg A_0 \cdot B_1 + \neg A_1 \cdot \neg B_0 \cdot B_1$$

Adesso ci rimangono solo queste 2 caselle, non adiacenti a niente:

	B_1	0	0	1	1
	B_0				
$A_1 A_0$		0	1	1	0
0 0	000	001	011	010	
0 1	001	010	100	011	
1 1	011	100	110	101	
1 0	010	011	101	100	

Dobbiamo quindi scriverle in forma completa

$$\neg A_1 \cdot A_0 \cdot \neg B_1 \cdot B_0 + A_1 \cdot A_0 \cdot B_1 \cdot B_0$$

Sommendo tutto otteniamo l'uscita che si può scrivere come

$$U_1 = A_1 \cdot \neg B_0 \cdot \neg B_1 + A_1 \cdot \neg A_0 \cdot \neg B_1 + \neg A_1 \cdot \neg A_0 \cdot B_1 + \neg A_1 \cdot \neg B_0 \cdot B_1 + \neg A_1 \cdot A_0 \cdot \neg B_1 \cdot B_0 + A_1 \cdot A_0 \cdot B_1 \cdot B_0$$

prendiamo ora in considerazione i gruppi di caselle adiacenti in cui il primo bit da sinistra abbia valore **1**.

	B_1	0	0	1	1
	B_0				
$A_1 A_0$		0	1	1	0
0 0	000	001	011	010	
0 1	001	010	100	011	
1 1	011	100	110	101	
1 0	010	011	101	100	

$$U_2 = A_1 \cdot B_1 + A_1 \cdot A_0 \cdot B_0 + A_0 \cdot B_1 \cdot B_0$$

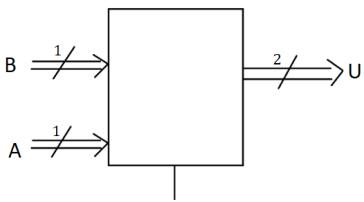
Riprendiamo ora tutte e tre le rappresentazioni appena scritte

$$U_0 = A_0 \cdot \neg B_0 + \neg A_0 \cdot B_0 \quad \text{questa è la funzione } A_0 \text{ XOR } B_0$$

$$U_1 = A_1 \cdot \neg B_0 \cdot \neg B_1 + A_1 \cdot \neg A_0 \cdot \neg B_1 + \neg A_1 \cdot \neg A_0 \cdot B_1 + \neg A_1 \cdot \neg B_0 \cdot B_1 + \neg A_1 \cdot A_0 \cdot \neg B_1 \cdot B_0 + A_1 \cdot A_0 \cdot B_1 \cdot B_0$$

$$U_2 = A_1 \cdot B_1 + A_1 \cdot A_0 \cdot B_0 + A_0 \cdot B_1 \cdot B_0$$

Realizziamo ora la versione più semplice (a due vie A e B, in cui ciascuna informazione di ingresso sarà codificata da 1 bit, il risultato da 2 bit)



		B		
		A	0	1
		0	00	01
		1	01	10

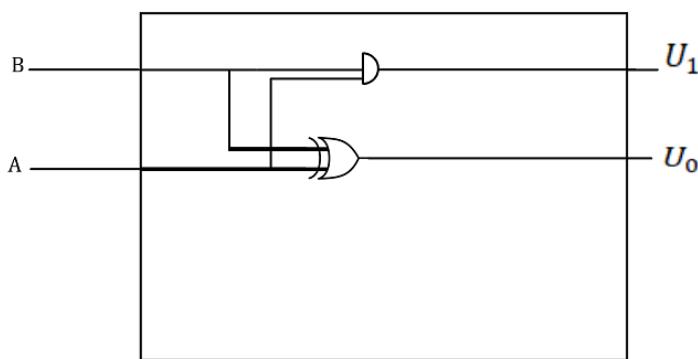
- Consideriamo l'uscita U_0 , in cui cerchiamo gruppi di caselle adiacenti in cui il bit meno significativo assuma valore 1

Non ci sono caselle adiacenti

Scriviamo $A \cdot \neg B + \neg A \cdot B$ questa è, quindi, sempre una funzione XOR

- Considerando adesso l'uscita in cui cerchiamo gruppi di caselle adiacenti in cui il bit più significativo assuma il valore 1 (c'è solo una casella) ci accorgiamo che in questo caso c'è una funzione AND ($A \cdot B$) (nella tavola di verità, in uscita, tutte le righe avrebbero il valore 0 tranne una)

L'implementazione fisica può dunque essere questa

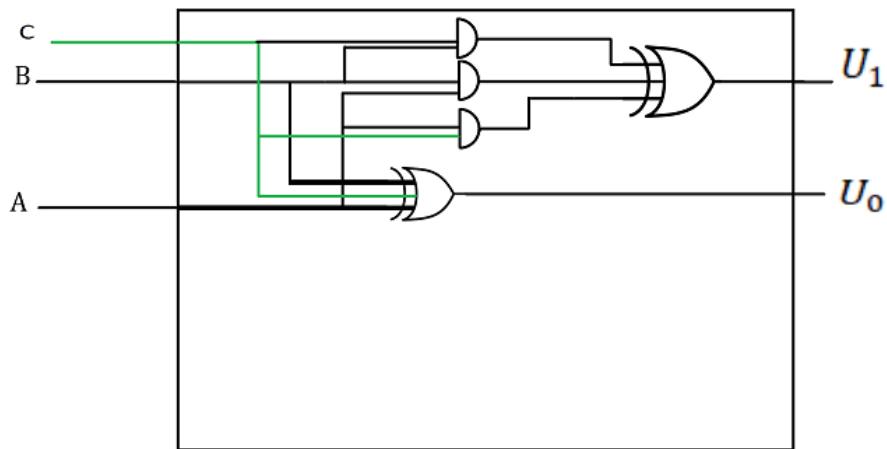


Tale circuito è chiamato half adder (semi sommatore, perché non si riesce mai a produrre il valore 3, ossia 11)

Se volessi creare un circuito sommatore che mi permetta di produrre in uscita anche il numero 3, cosa dovrei fare?

- Aggiungere un ingresso.

- PER QUANTO RIGUARDA l'estensione dell'USCITA U_0 (riguardante il bit meno significativo) basta collegare anche il nuovo ingresso alla funzione XOR
- L'estensione di U_1 (che, ricordiamo, dipende da TUTTI gli ingressi) è più complicata: per avere l'1 in uscita devo avere almeno 2 ingressi che assumono il valore 1, dobbiamo dunque aggiungere delle funzioni AND

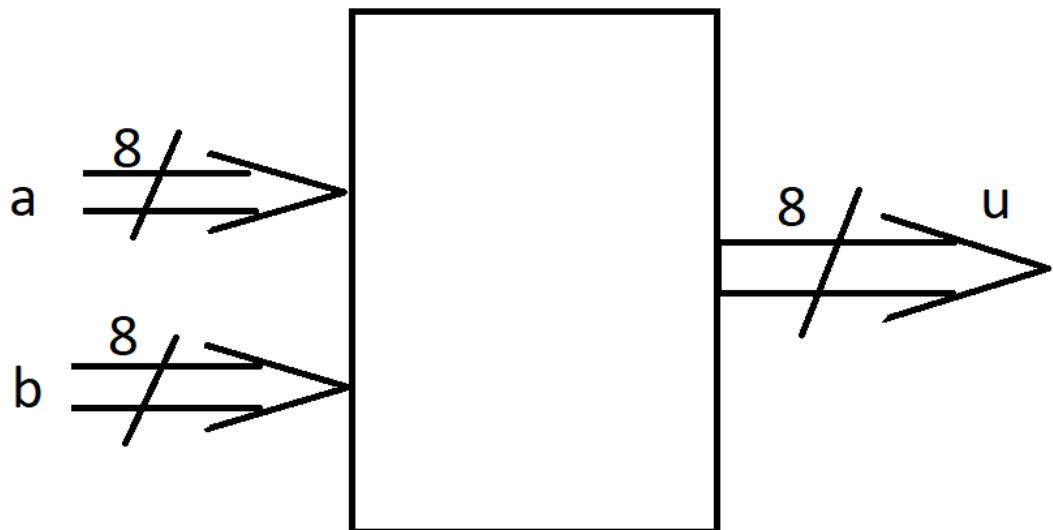


QUESTO DISPOSITIVO verrà chiamato FULL ADDER
 (con questo dispositivo faccio somme tra valori espressi su 3 bit, producendo risultati espressi su 2 bit, quindi valori da 0 a 3)

Appunti del 2 novembre

I sommatori visti finora ci aiutano su rappresentazioni di numeri su 1 solo bit, come posso fare per aumentare il numero di bit della rappresentazione? Come faccio la somma di 2 numeri rappresentati su 8 bit, ad esempio?

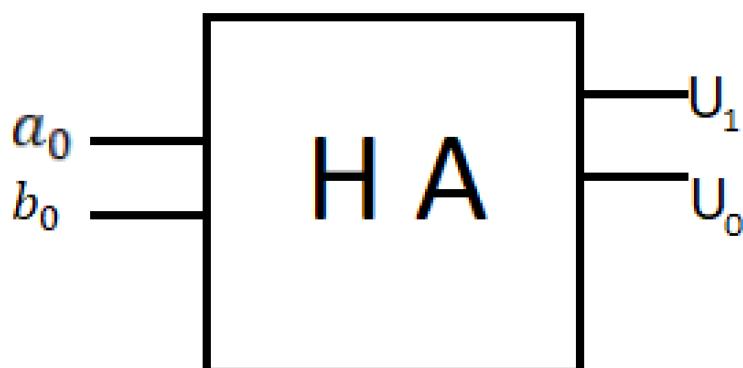
Utilizzo più volte questi dispositivi per sommare i bit in ingresso



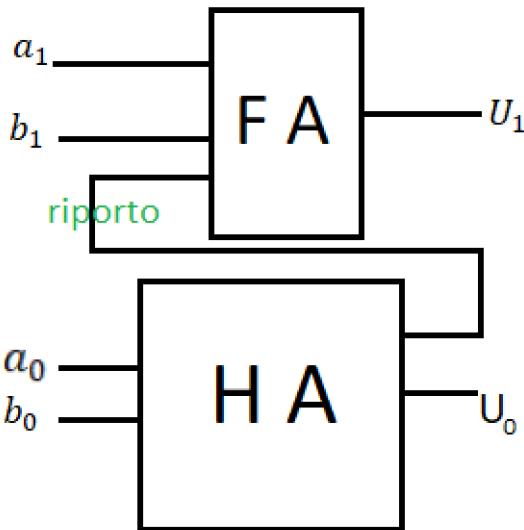
Possiamo usare i half adder o i full adder come componenti da inserire all'interno di questa scatola

L'idea sarebbe di partire dal bit meno significativo, quindi usare il half adder per effettuare la somma di $a_0 + b_0$, mandiamo quindi in uscita il bit meno significativo

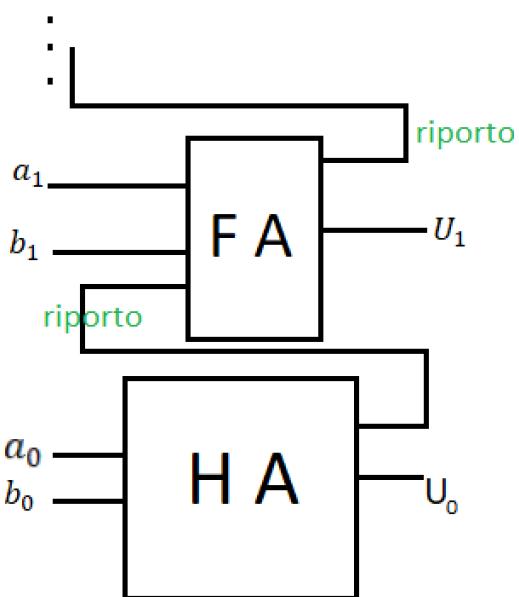
Sappiamo che un half adder è fatto più o meno così



Sappiamo che l'half adder su 2 bit ci produce 2 bit anche in uscita, cosa ce ne facciamo, dunque di due uscite? La seconda uscita la possiamo usare come riporto per la cifra successiva: al secondo passo useremo dunque un addizionatore completo (full adder) che terrà conto di 3 valori in ingresso: a_1, b_1 e *il riporto*. Mandiamo quindi in uscita U_1



e generiamo il riporto verso la cifra successiva



e così via...

da questo punto in avanti possiamo aggiungere bit utilizzando un numero opportuno di circuiti di tipo fulladder, che somma 1 bit di a, 1 bit di b e il bit di riporto dal dispositivo precedente.

Potendo realizzare tali dispositivi in logica a 3 livelli, possiamo in qualche modo prevedere il ritardo (fisso) da cui questi dispositivi saranno caratterizzati.

Come calcoliamo il ritardo con cui verrà prodotta l'uscita, in caso di variazioni dei valori di ingresso?

Es. partiamo dall'inizio:

per produrre U_0 usiamo un solo dispositivo; quindi abbiamo il ritardo fisso di un solo dispositivo, per produrre U_1 invece, U_1 può dipendere anche dalla cifra di riporto, quindi, essendo coinvolti due dispositivi (il primo half adder e il primo full adder) abbiamo ritardo di 2 dispositivi, se quindi avessimo un sommatore a 8 bit il ritardo sarebbe 8 volte il tempo di ritardo di un singolo dispositivo.

Se invece di 8 bit ne mettessimo 32, la situazione peggiora. Ogni volta che aumentiamo il numero di bit, produciamo un dispositivo sempre più lento.

Quale potrebbe essere la velocità di un dispositivo costruito in questo modo? Supponiamo che il tempo di ritardo di una singola porta logica NAND sia di **100ps** (picosecondi)

Ricorda: Pico= 10^{-12}

Per ciascun addizionatore (semi o meno) abbiamo 3 livelli di logica, quindi $100 \times 3 = 300$, ciascun dispositivo (sia full che half) ha un ritardo stimato di 300ps.

Moltiplico 300 per 64 e ottengo 19.200: il tempo di ritardo in picosecondi di un sommatore a 64 bit. →19,2 ns (nanosecondi).

Quante operazioni di somma riesco a fare in un secondo, tenendo conto di questo ritardo di 19,2 ns? 50 milioni di operazioni.

Quindi siamo a 50 MIPS (Million Instructions Per Second). 50 MIPS Non sono tantissimi, se accresco troppo i bit il dispositivo prodotto sarà troppo lento. Quale è l'alternativa? Usare altri circuiti combinatori, oltre i sommatori.

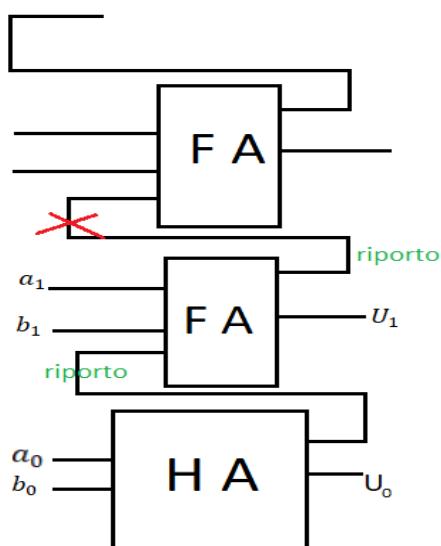
Questi altri circuiti che vengono usati sono chiamati “previsori del riporto” (Carry-look ahead).

Tale circuito cerca di ridurre l’“effetto a catena del riporto” tra un bit e il bit successivo.

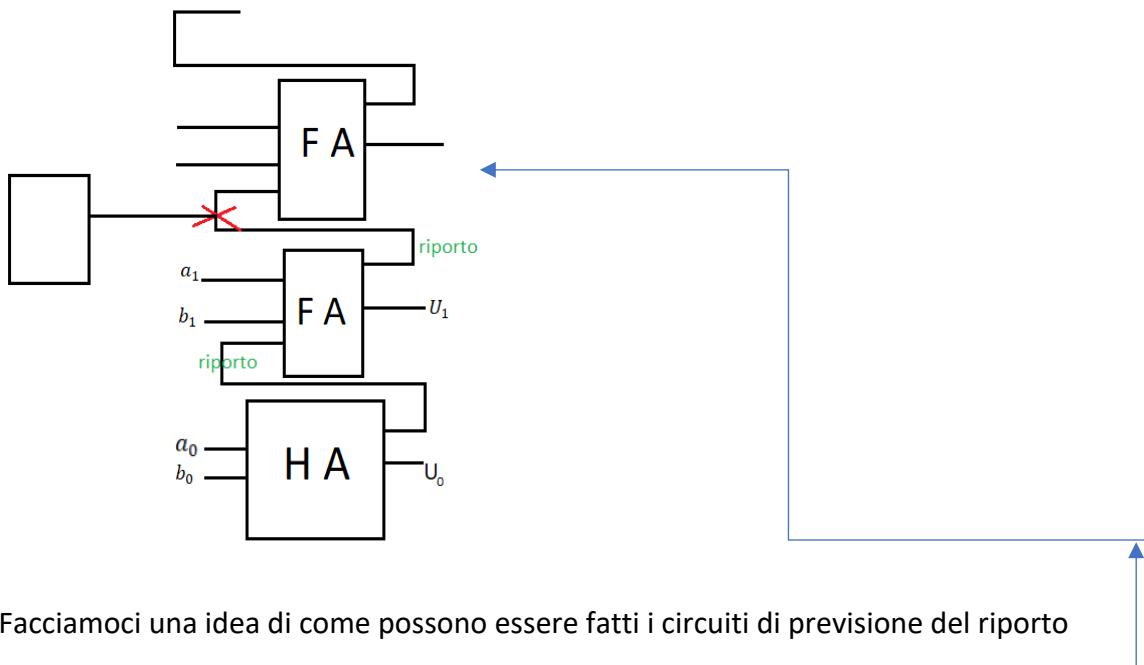
Per un certo numero di addizionatori consecutivi si può tranquillamente usare la configurazione elencata in precedenza

N.B: la prima configurazione che abbiamo visto (con la catena di riporti) si chiama “ripple carry”

; ad un certo punto, per evitare che il tempo di ritardo cresca in modo troppo elevato si “taglia” la catena dei riporti:



E la sostituisco con un circuito combinatorio del tipo previsore del riporto, il quale tiene conto di tutti i bit in ingresso precedenti ed ha una uscita, la quale assume il valore 1 quando viene generato 1 bit di riporto, il valore 0 quando non viene generato nessun riporto



Facciamoci una idea di come possono essere fatti i circuiti di previsione del riporto

Indichiamo con C il bit di riporto, pensando di fare la previsione del riporto per la 3° cifra,

quando è che viene generato un riporto sul 3° bit? Quando il risultato dell'operazione "eccede" il numero di valori rappresentabili su un solo bit, ad esempio se a_1 e b_1 hanno valore 1, in uscita viene prodotto un valore (10) che non può essere scritto su un solo bit, quindi viene generato un riporto; allo stesso modo se ho un riporto già dall'operazione precedente (e quindi a_0 e b_0 valgono entrambi 1) basta che solo uno tra a_1 e b_1 abbiano valore 1 perché si generi un riporto sulla posizione di mio interesse (poiché ho già un bit di riporto da prima che sommo con un 1 adesso)

Scriviamo dunque

$$C = a_1 \cdot b_1 + a_1 \cdot a_0 \cdot b_0 + b_1 \cdot b_0 \cdot a_0$$

Quale è la particolarità di questa rappresentazione? Non ho nessuna negazione, quindi posso classificare tale dispositivo come un dispositivo a 2 livelli (non mi servono funzioni NOT). Se, quindi, come ho detto in precedenza, una funzione ha 100ps di ritardo, il ritardo del carry lookahead sarà di 200ps.

Se voglio che il mio dispositivo raggiunga un tasso di MIPS più alto devo aggiungere circuiti carry-look ahead tanti quanti ne bastano per "tagliare" il tempo di calcolo dell'ultima cifra.

Considerazione: se voglio costruire un addizionatore a più bit che costi poco ed usi poca corrente e quindi non si surriscaldi, devo minimizzare la complessità circuitale (quindi userò solo half e full adder, senza carry-look ahead), lo svantaggio di tale configurazione è che è lenta, se voglio

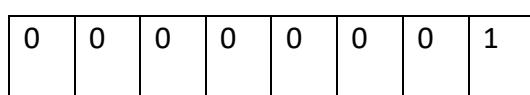
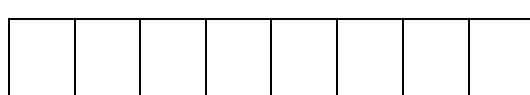
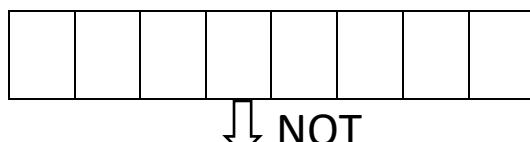
aumentare la velocità aggiungo dei carry-look ahead in diversi posti, ciò però comporterà un aumento del volume e della complessità del dispositivo e dunque anche la necessità di provvedere al raffreddamento di quest'ultimo).

Se questi circuiti sommatori che realizziamo sono applicati su rappresentazioni in complemento a 2, allora posso sia sommare numeri con segno che numeri senza segno. Quindi posso anche fare le sottrazioni, oltre che le somme.

Cosa posso fare per cambiare il segno? Se dovessi fare $B-A$ anziché $B+A$, come faccio per realizzare un dispositivo capace di cambiare il segno di un numero in complemento a 2?

Ricordandoci la lezione sulle diverse rappresentazioni (modulo e segno, complemento a 1 e complemento a 2) sappiamo che per passare da modulo e segno a complemento a 2 bisogna:

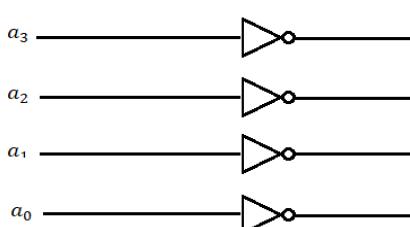
- Prima di tutto “negare” la rappresentazione modulo e segno, cambiando quindi tutti i valori, dove c’è scritto 1 ci metteremo 0 e dove c’è scritto 0 ci metteremo 1, così facendo si giunge ad una rappresentazione in complemento a 1
- Aggiungere la costante 1 alla rappresentazione in complemento a 1, così da arrivare ad una rappresentazione in complemento a 2.



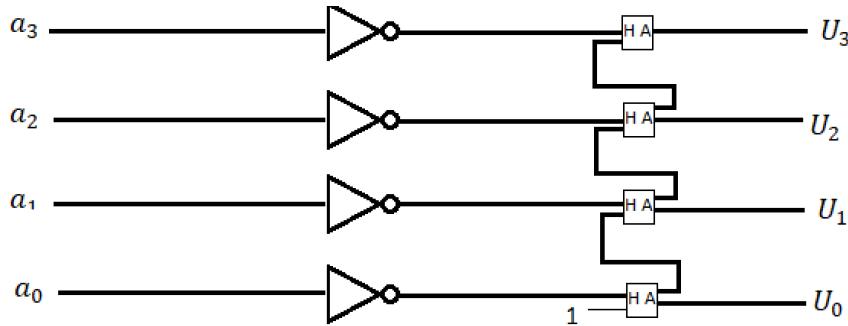
Devo quindi realizzare un dispositivo capace di fare tale operazione

Come lo faccio?

Poniamo di voler usare rappresentazioni a 4 bit



In questo modo avrei prodotto la rappresentazione in complemento a 1 di a . Adesso devo sommare la costante 1 (per ottenere la complemento a 2). Per farlo posso usare dei semiaddizionatori (half adder)

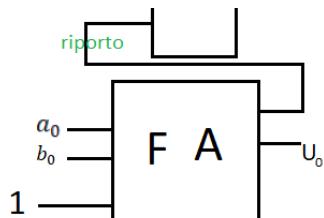


$$U = -A$$

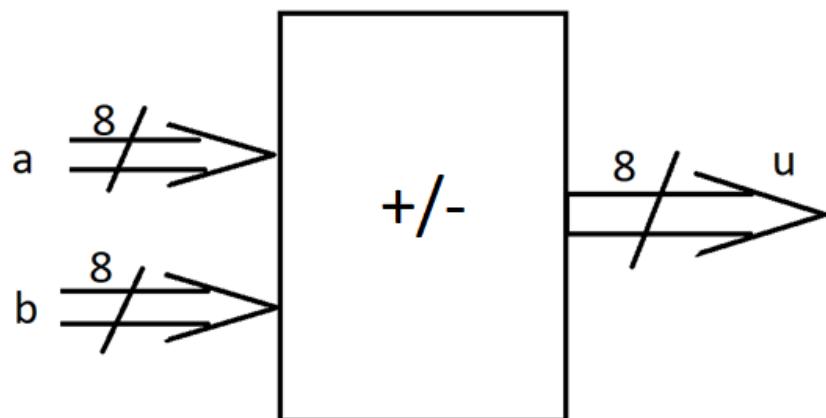
Arrivo quindi alla rappresentazione in complemento a 2 di $-A$

Questa misura è, però, un po' eccessiva, siccome sto introducendo una "sfilza" di semiaddizionatori per sommare semplicemente la costante 1.

Il mio scopo è fare una operazione di differenza (del tipo $B-A$), quindi dopo la rappresentazione appena fatta metterò un addizionatore a 4 bit, posso compattare la rappresentazione usando un addizionatore completo all'inizio (invece che un half adder) facendogli arrivare il valore costante 1 da sommare al primo bit:

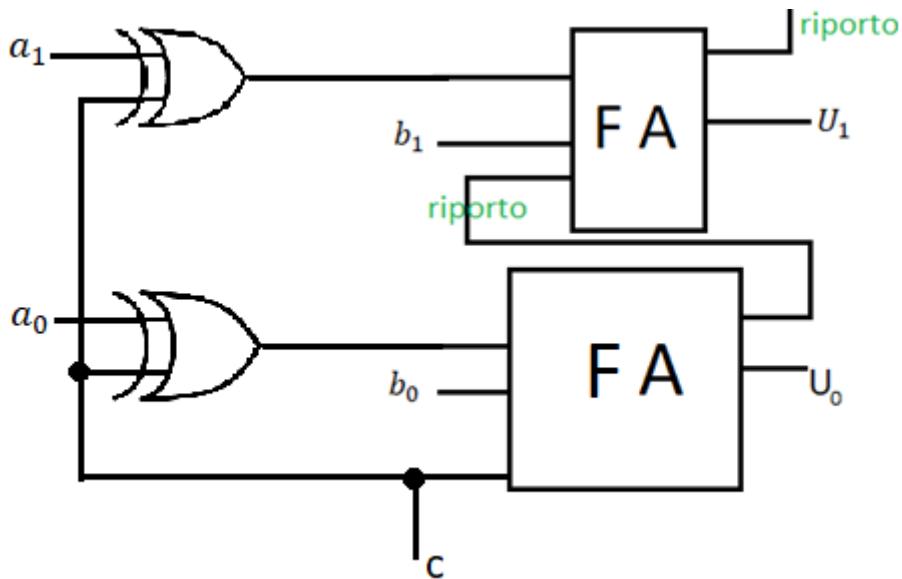


Posso quindi costruire un dispositivo un po' più complesso ma che riesca a fare sia operazioni di addizione che di sottrazione:



(rappresentazione grafica di un dispositivo che può fare operazioni di somma e sottrazione su 8 bit)

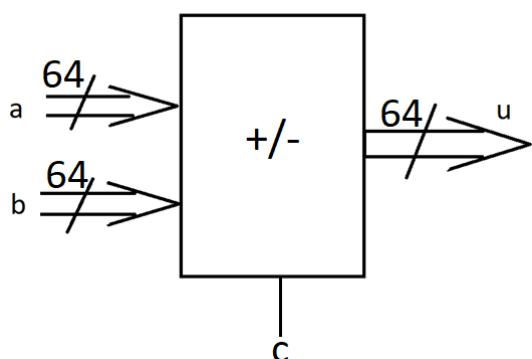
Come costruisco un simile dispositivo?



- Se voglio fare le somme (del tipo $A+B$) NON devo negare nulla, quindi il valore che inserirò in basso sarà 0;
- se invece volessi fare, per esempio, $B-A$, allora dovrei negare A, e per fare la negazione di A in complemento a 2, abbiamo visto che, oltre ad usare una funzione NOT, dobbiamo inserire il valore 1 nel fulladder insieme ai due ingressi principali. Per negare A posso anche usare la funzione or esclusivo (XOR) invece che NOT, così da avere un ingresso in più, da usare come ingresso di controllo per scegliere se fare o meno la negazione in questione.

Quindi nello XOR abbiamo 2 fili: il primo è il valore che sarebbe da negare o meno (a_1, a_0 , per esempio), il secondo è un filo che parte dalla variabile di controllo C (che è il bit di riporto della cifra precedente in ingresso al full adder).

La schematizzazione è quindi una di questo tipo



Abbiamo quindi 3 ingressi:

- l'ingresso a su un certo numero di bit (ho scritto 64 ma ci potrebbe essere qualsiasi potenza di 2, perché sto parlando in generale, se fossero 8 bit scriverei 8, se fossero 32 scriverei 32)
- l'ingresso b su un certo numero di bit
- l'ingresso c di controllo, che determina se la negazione avverrà o meno e quindi determinerà quale funzione calcolerà in uscita: ad esempio $B+A$ oppure $B-A$

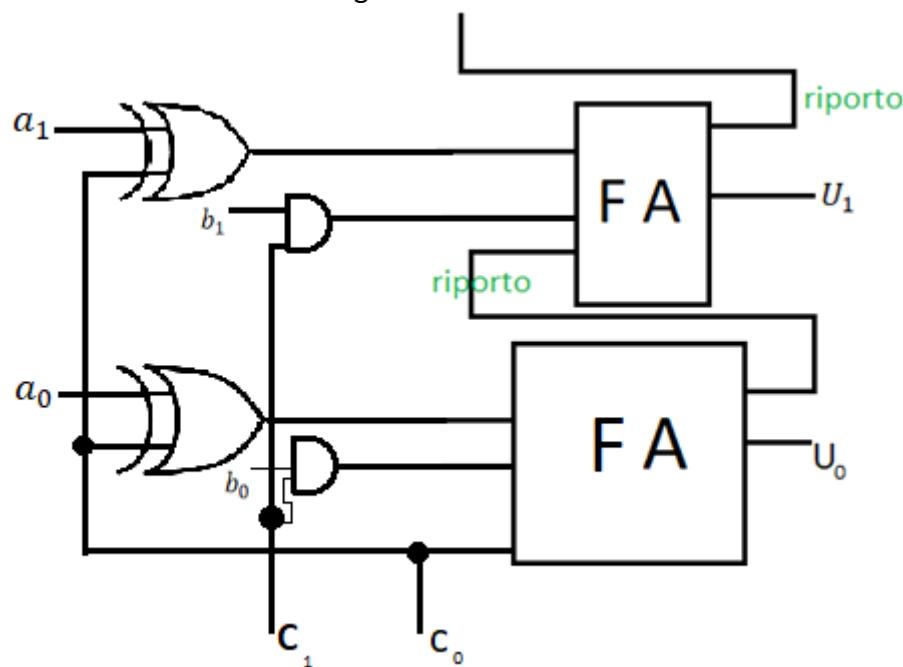
C	U
0	B+A
1	B-A

Questa idea si può “estendere” per altre operazioni, ci potrebbe interessare, ad esempio, solamente cambiare il segno di un numero, senza fare somme.

Dato A calcolare -A per esempio.

Possiamo implementare l’operazione “cambio di segno” nel dispositivo costruito precedentemente: per fare tale operazione dovremmo azzerare B, così da avere, in uscita, 2 casi in più (oltre ai già presenti B+A e B-A), cioè A e -A.

Come azzero B? Aggiungendo un bit di controllo e usando il bit di controllo come secondo ingresso di una funzione AND a due ingressi



C_1	C_0	U
0	0	A
0	1	-A
1	0	B+A
1	1	B-A

Quindi riassumendo, in questo caso, se l'ingresso di controllo C_1 vale 0 allora B verrà azzerato, i risultati ne conseguono (0+A nella prima riga, 0-A nella seconda).

Estendendo ancora di più tale idea si arriva a definire un unico dispositivo capace di eseguire tutte le operazioni aritmetiche che ci interessano: unità aritmetica.

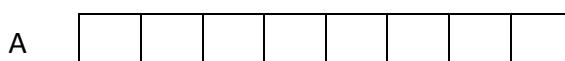
L'unità aritmetica è un componente del processore.

Quali sono le altre operazioni che ci possono servire all'interno di un processore?

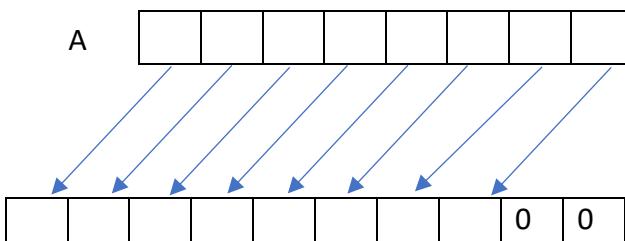
Moltiplicazione e divisione.

Per quanto riguarda la moltiplicazione p una operazione abbastanza complicata, tranne per la moltiplicazione per una potenza di 2

Se io ho



Fare $4 \cdot A$, per esempio, non è difficile, basta inserire 2 zeri nelle posizioni meno significative della nuova rappresentazione e copiare in seguito tutti i valori di A



$4 \cdot A$

Se vorrò moltiplicare per 8 scriverò 3 zeri e copierò di seguito tutti i valori di A, se vorrò moltiplicare per 2 lascerò solo uno zero, moltiplicare per 16 lascerò 4 zeri e così via.

Questa è una operazione che viene chiamata di scorrimento (shift, arithmetic shift per la precisione)

Se si fa uno scorrimento verso sinistra abbiamo una operazione di moltiplicazione, quando lo scorrimento è a destra abbiamo una operazione di divisione.

Come potrei fare per implementare delle operazioni di scorrimento a sx e dx?

Ci servirà un multiplexer: mettiamoci nella situazione di un singolo bit su tutta la rappresentazione completa:



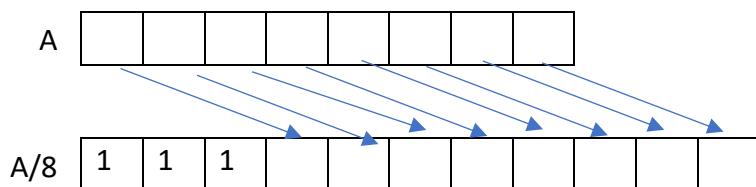
Qui dentro ci sarà uno dei bit di A, che verrà deciso in base al valore per cui si moltiplica (o si divide): se per esempio si moltiplica per 4 in quel preciso posto ci andrà un bit di una precisa posizione di A, se si moltiplicasse per 2 ci andrebbe un bit che sta in un'altra posizione precisa di A (perché si "scala" aggiungendo e togliendo zeri, a destra o a sinistra, in base al fatto che si moltipichi o si divida).

Il multiplexer avrà come ingressi tutti i bit di A, una uscita in cui verrà riportato il bit "opportuno" /"scelto" di A ed un ingresso di controllo che "sceglierà" il bit di A, in dipendenza alla costante per cui voglio moltiplicare o dividere.

Bisogna quindi usare un multiplexer ad n ingressi per ogni bit (dove n è il numero di bit della rappresentazione A)

Nota bene: per le operazioni di divisione di un numero negativo non devo più scrivere zeri a sinistra, bensì uni.

Ad esempio, volendo fare la divisione A/8, se A è negativo dovrò mettere 3 uni:



Quindi devo distinguere due casi:

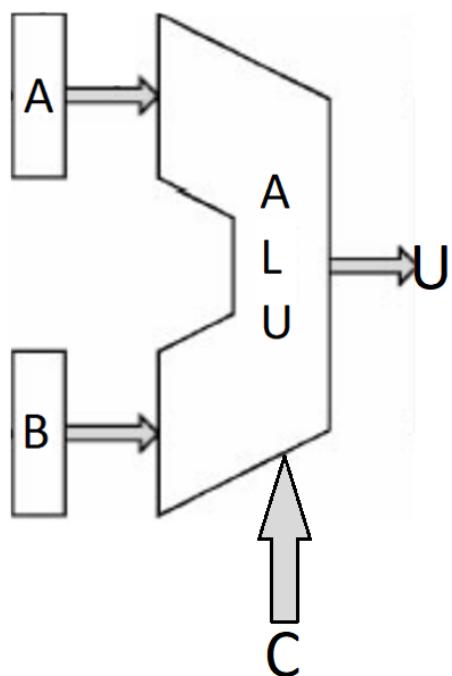
- Lo scorrimento verso destra è applicato ad un numero con segno in complemento a 2 e allora devo replicare il bit di segno nelle cifre aggiunte a sinistra: se il segno è positivo, quindi il bit di segno è 0 dovrò scrivere 0, se il bit di segno è 1 (quindi negativo) dovrò scrivere 1.
- Lo scorrimento verso destra è applicato ad una rappresentazione di numero senza segno, in questo caso metterò sempre 0, indipendentemente da quanto valga il primo bit (perché il primo bit adesso non è il bit di segno)

Abbiamo quindi esplicato addizione, sottrazione, moltiplicazione e divisione (implementazioni semplici).

Che altre operazioni ha senso inserire in un sistema di calcolo?

Le funzioni NOT, AND, OR e XOR.

Mettendo insieme tutte queste funzioni ed operazioni elementari produciamo un unico dispositivo, chiamato Unità Aritmetico Logica (ALU)



Esistono anche altre operazioni di scorrimento oltre a quelle aritmetiche già viste.

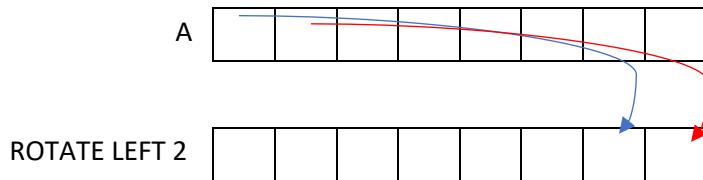
Esse sono implementate quando le rappresentazioni binarie A e B non sono necessariamente rappresentazioni di numeri.

Le altre operazioni di shift che possono essere introdotte sono, ad esempio, quelle di rotazione

Come funziona l'operazione di rotazione?

La differenza tra l'operazione di shift aritmetico e quella di rotazione è che non vengono aggiunte costanti (1 o 0) per un certo numero ma tali spazi vengono occupati dai bit della rappresentazione di partenza.

Ad esempio, se voglio fare una operazione di rotazione a sinistra di 2 posizioni



TENIAMO CONTO CHE, per esempio, su rappresentazioni a 32 bit, una rotazione a sinistra di 31 bit equivale a una rotazione a destra di 1 bit.

Nelle ALU vengono portate in uscita, oltre al risultato vero e proprio, anche delle “condizioni” (bit di condizione) che ci danno delle informazioni aggiuntive rispetto al risultato, informazioni interessanti da sapere in più sono, per esempio:

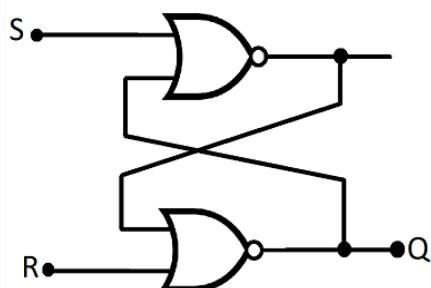
- se si è generato un riporto o meno (la condizione di riporto viene chiamata “carry”, riporto in inglese appunto) (se all’ALU non viene chiesta un’operazione aritmetica, il carry assume tipicamente valore 0)
- se, avendo operato su numeri con segno (compl. A 2), il risultato ha il segno corretto oppure no (se si è verificata o meno una situazione di overflow)
- se il risultato è 0 (anche se è una condizione deducibile dal risultato U, basta guardare tutti i bit), tale bit di condizione assumerà il valore 1 se il risultato/l’uscita è 0, mentre assumerà valore 0 se il risultato è diverso da 0. Tale bit si calcola facendo un NOR su tutti i bit di U (NOR a 32/64/... ecc ingressi)

CIRCUITI SEQUENZIALI

Differenza tra combinatori e sequenziali? In quelli combinatori abbiamo delle funzioni che ci permettono di determinare univocamente l’uscita in funzione della combinazione dei valori di ingresso.

Nei circuiti sequenziali le uscite non dipendono solo dai valori di ingresso attuali ma anche da quelli precedenti.

Più semplice Esempio di circuito sequenziale:



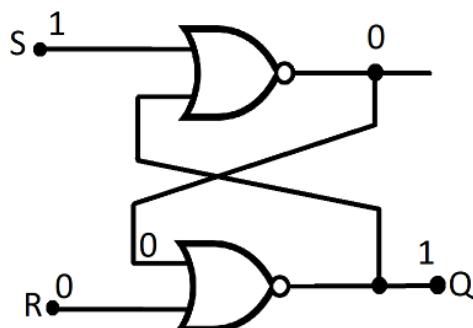
(C’è un “ritorno” da una uscita verso un ingresso)

alcuni dei valori prodotti in uscita vengono riportati sugli ingressi: i circuiti sequenziali sono caratterizzati quindi da almeno un ciclo chiuso.

Studiamo il comportamento di un simile dispositivo

S	R	Q
0	0	
0	1	0
1	0	1
1	1	0

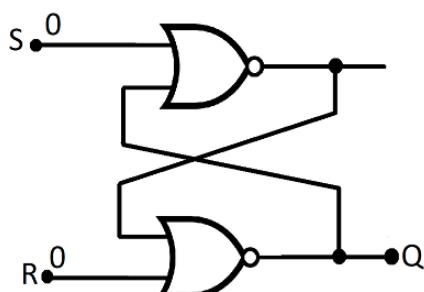
ATTENZIONE, nella terza riga siccome sull'ingresso S abbiamo 1 sicuramente l'uscita della funzione NOR che sta sopra sarà 0 (per essere 1 devo avere due zeri), ma vediamo che, in questo tipo di circuito, l'uscita della prima funzione NOR, quella che sta sopra, viene riportata in ingresso alla seconda funzione NOR).



Nell'uscita Q in questo caso avrò dunque un 1, poiché in ingresso ci sono i valori 0 0.

Ragionamenti simili potrebbero valere, quindi, per le altre righe.

Cosa succede nella prima riga della tavola di verità?



Come facciamo a calcolare quanto varrà Q se abbiamo 0 in entrambi gli ingressi?

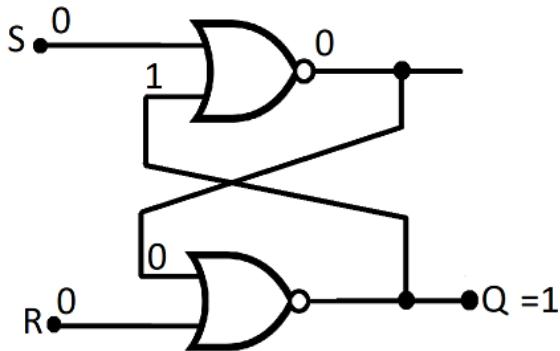
Per quanto riguarda la seconda funzione NOR ho uno 0 in ingresso (R) quindi l'uscita potrebbe valere 1, ma dipende dall'altro ingresso, e quell'altro ingresso è l'uscita della prima funzione NOR, ma, a sua volta, l'uscita della prima funzione NOR dipende, oltre che dall'ingresso S, anche dal secondo ingresso, il quale è l'uscita della seconda funzione NOR.

Effetto “cane che si morde la coda”. Cosa facciamo?

Possiamo fare delle ipotesi. Attualmente conosciamo per certo solo gli ingressi S ed R.

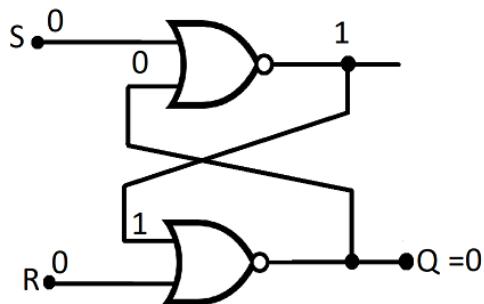
Supponiamo di conoscere il valore Q, per esempio che Q sia 1

Se partiamo da tale ipotesi possiamo “risolvere” tutto il resto:



(se partiamo dall’ipotesi che Q vale 1 troviamo la conferma che Q vale effettivamente 1).

Ipotizziamo ora che Q valga 0



Partendo dall’ipotesi che Q=0 troviamo che Q vale effettivamente 0.

Ambedue le ipotesi sono quindi ragionevoli.

In questo caso quindi non siamo in grado di determinare univocamente il valore in uscita in funzione degli ingressi.

Possiamo quindi scrivere così:

S	R	Q
0	0	Q

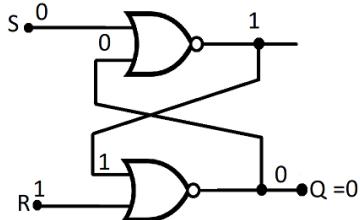
Per poter determinare il valore in uscita devo conoscere quale fosse la combinazione in ingresso precedente.

Devo prendere in considerazione, quindi, una sequenza di valori.

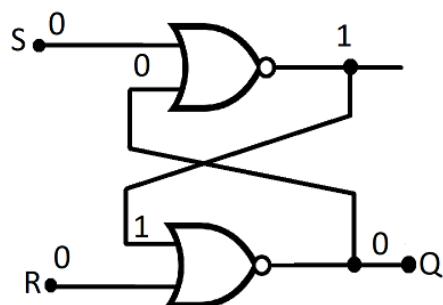
Consideriamo di avere

Prima gli ingressi S=0 R=1 e poi cambiano e diventano S=0 R=0

Se ho 0 su S e 1 su R avrò 0 in uscita Q, per cambiare l'uscita una volta che l'ingresso R cambia da 1 a 0 ci vorrà per forza un po' di tempo nella realizzazione fisica (ogni dispositivo ha un seppur minimo ritardo), partendo quindi da S=0 e R=1 avrò una rappresentazione simile (ho un 1 in ingresso alla seconda funzione NOR quindi in uscita verrà sicuramente prodotto 0, questo 0 va anche in ingresso alla prima funzione NOR, la quale, avendo due zeri produce 1, il quale va in ingresso alla seconda NOR)



Se io cambio il valore di ingresso di R, e quindi adesso ho S=0 e R=0, sapendo che ci sono dei ritardi, sono certo che, per una certa frazione di tempo, la seconda funzione NOR continua ad avere in ingresso 1 (l'1 che risultava dall'uscita della prima funzione NOR), si ritrova quindi in ingresso i valori 1 e 0 (cioè R, appena cambiato), ma se si ritrova 1 e 0 continua a produrre 0 in uscita, valore il quale va in ingresso della prima funzione NOR e produce 1 in uscita, il quale continua di nuovo ad andare in ingresso alla seconda funzione NOR, che quindi continua a produrre 0



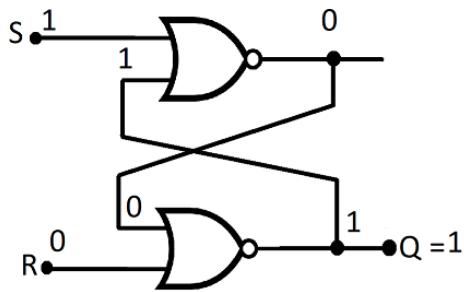
Quindi so per certo che se passo da S=0,R=1 a S=0,R=0 in uscita avrò il valore 0

S	R	Q
0	0	Q
0	1	0
1	0	1
1	1	0

Se il ritardo fosse NULLO (se il cambio dei valori in uscita fosse istantaneo) ciò non accadrebbe.

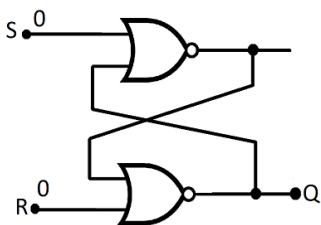
Consideriamo ora, invece, di aver avuto gli ingressi prima S=1 e R=0, che ora cambiano e diventano S=0 e R=0.

Nello stato iniziale, con S=1, la prima funzione NOR produce il valore 0:

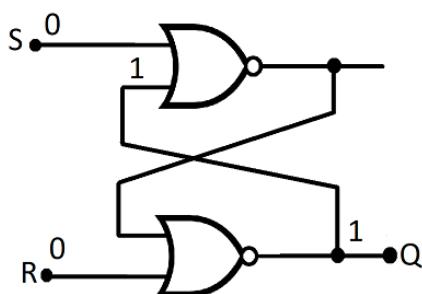


0 che andrà in ingresso alla seconda funzione NOR, la quale ha in ingresso un altro 0 (R), e quindi produce il valore 1 in uscita, valore che andrà nel secondo ingresso della prima funzione NOR, che avendo 1 1 negli ingressi continua a produrre 0 in uscita.

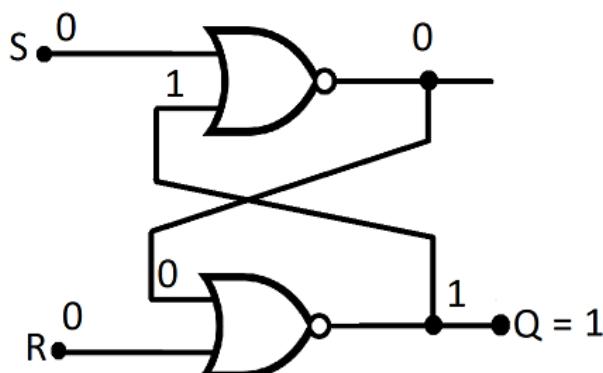
Ora realizzo il cambiamento: faccio passare S da S=1 a S=0



Ma sapendo che, dopo aver cambiato gli ingressi, c'è un minimo ritardo nella realizzazione fisica a processare l'uscita in base a questi cambiamenti, sono certo che per almeno un istante la prima funzione NOR continuerà ad aver in ingresso un 1 (1 che esce dalla seconda funzione NOR, di prima, e che non cambia all'istante, perché la funzione ha un ritardo)



E se la prima funzione NOR ha in ingresso 0 e 1 sicuramente produrrà il valore 0 in uscita, tale valore andrà in ingresso alla seconda funzione NOR, che, avendo due zeri, continuerà a produrre 1 in uscita



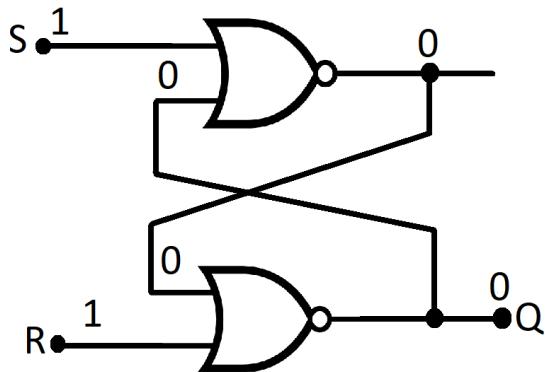
quindi so per certo che se passo da S=1, R=0 a S=0, R=0, in uscita avrò il valore 1.

S	R	Q
0	0	Q
0	1	0
1	0	1
1	1	0



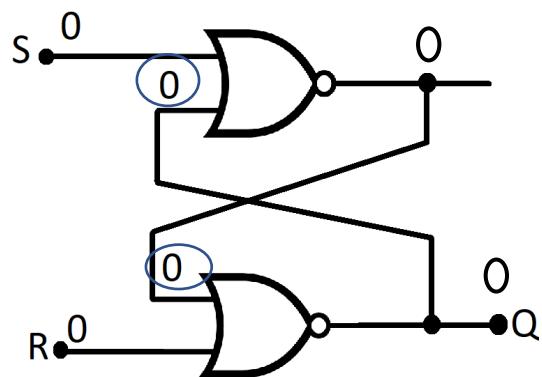
Cosa succede invece se parto da $S=1, R=1$ e cambio facendo diventare gli ingressi $S=0, R=0$?

Se all'inizio ho $S=1, R=1$ entrambe le funzioni NOR producono 0 in uscita

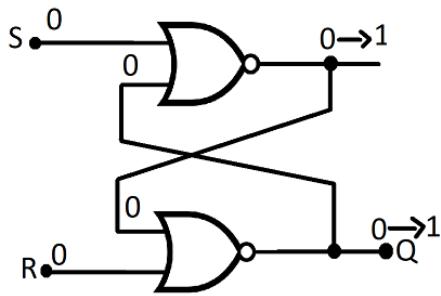


E lo 0 dell'uscita della prima funzione NOR andrà in ingresso alla seconda funzione NOR, lo 0 in uscita della seconda funzione NOR andrà in ingresso alla prima funzione NOR. Ambedue le funzioni NOR quindi hanno in ingresso un 1 e uno 0.

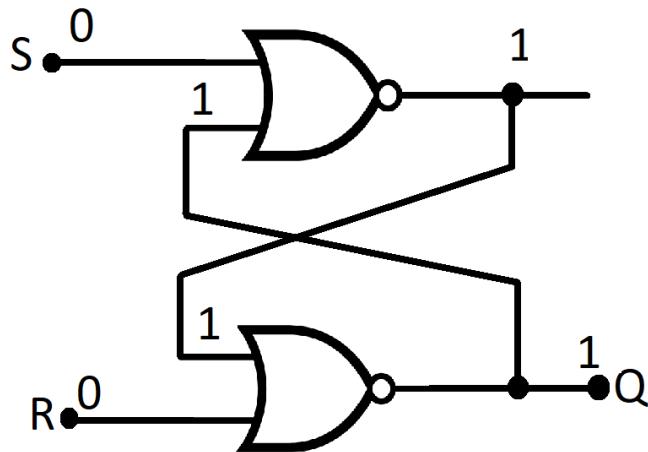
Applicando il cambiamento, passando quindi ai valori 0 su S e 0 su R, continuerò però ad avere, per qualche istante, sia nella prima funzione NOR che nella seconda, lo 0 di prima (poiché le uscite, non cambiando all'istante, hanno ancora il valore 0 per un momento)



Se aspetto il tempo (equivalente al ritardo del dispositivo), alla fine le uscite cambieranno e diventeranno 1 (perché in ambedue le funzioni ho 0 0, e ciò mi deve dare 1 in uscita)



Ma siccome adesso ho 1 in uscita sia nella prima funzione che nella seconda, tali valori vengono riportati sia in ingresso alla prima funzione NOR sia in ingresso alla seconda funzione NOR



Ma adesso, quindi, le due funzioni hanno un 1 e uno 0 ciascuno, e quindi devono produrre il valore 0, aspettando il tempo dovuto, quindi, il valore in uscita cambierà di nuovo, diventando 0, ma poi, ripetendo il ragionamento di prima cambierà di nuovo a 1 e poi di nuovo a 0, e così via...

$0 \rightarrow 1 \rightarrow 0 \rightarrow 1 \rightarrow 0 \rightarrow 1$

Quindi che valore c'è in uscita? non c'è un valore fisso, l'uscita oscilla continuamente tra valore 0 e valore 1.

L'idea è quindi che, se voglio usare correttamente questo dispositivo, NON DEVO prendere in considerazione la possibilità di passare direttamente dalla configurazione $S=1, R=1$ alla configurazione $S=0, R=0$



S	R	Q
0	0	Q
0	1	0
1	0	1
1	1	0

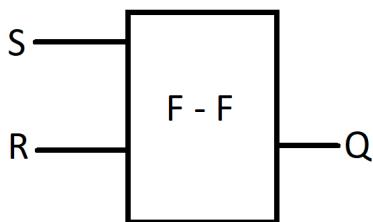
Vietiamo, quindi, del tutto la combinazione di ingressi 1, 1

S	R	Q
0	0	Q
0	1	0
1	0	1
1	1	0

X

Tale circuito sequenziale è chiamato "flip-flop di tipo set-reset"

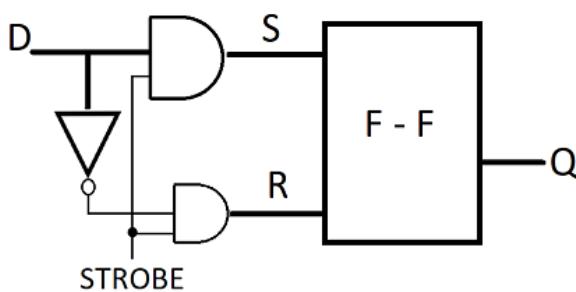
Può essere schematizzato così:



Riassumiamo il comportamento del flip-flop di tipo set-reset:

- prevede che sia esclusa la combinazione 1, 1 in ingresso (quindi al massimo uno, tra S e R, può assumere valore 1)
- quando sia S che R assumono il valore 0, presenta un valore in uscita che non dipende dalla attuale combinazione di ingresso, ma da quella che c'era prima
- quando S assume valore 1 (e R assume valore 0) l'uscita varrà 1 (set)
- quando S assume valore 0 e R valore 1 l'uscita varrà 0 (reset)

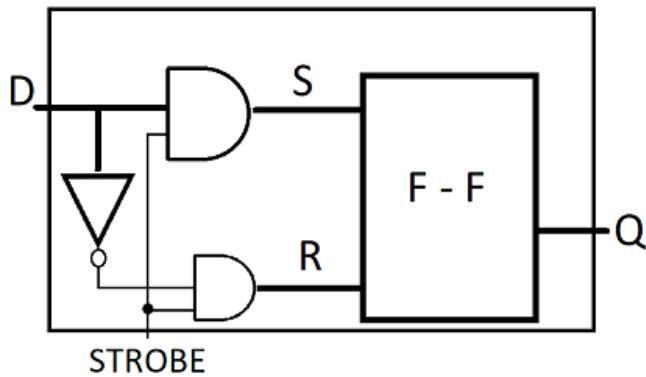
come facciamo, quindi, per ESCLUDERE la combinazione 1 1? Prima del flip-flop di tipo set-reset introduciamo 2 funzioni AND, che hanno lo scopo di poter azzerare contemporaneamente S ed R:



(quando l'ingresso STROBE vale 0, gli ingressi S ed R del flip-flop valgono entrambi 0)

Se metto valore 1 in STROBE avrò il valore D su S ed il valore $\neg D$ su R

Una volta verificato che abbiamo ottenuto ciò che volevamo (escluso la combinazione 1 1 in ingresso), possiamo creare tale dispositivo, contenente, oltre a un flip-flop con quindi le sue funzioni NOR, anche le due funzioni AND e la funzione NOT, che servono al mio scopo di escludere la combinazione 1 1)



Tale dispositivo è chiamato Flip-Flop di tipo D.

Esaminiamo il comportamento del Flip-Flop di tipo D:

- STROBE=0 \rightarrow S e R valgono 0 \rightarrow l'uscita Q non può cambiare, rimane quella che c'era prima
- STROBE=1 \rightarrow S e R dipendono da D:

S avrà valore D e R avrà valore $\neg D \rightarrow$ l'uscita stessa, quindi, dipende da D: $D=1 \rightarrow Q=1$; $D=0 \rightarrow Q=0$

Si può dire quindi che l'uscita Q valga D.

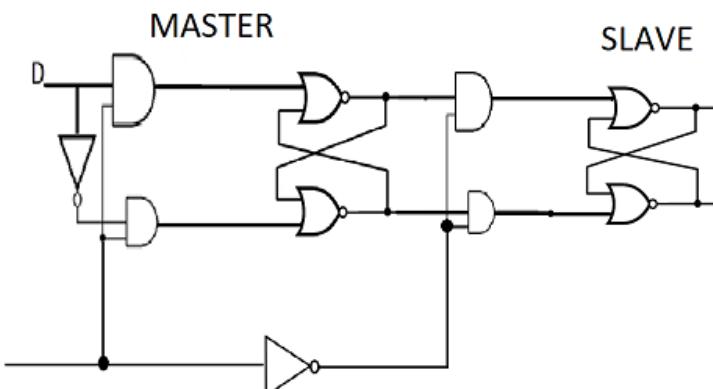
Tale dispositivo, quindi, sa fare due cose:

- Quando STROBE vale 1 riproduce in uscita lo stesso valore che c'è in ingresso D
- Quando STROBE vale 0 (lo faccio passare da 1 a 0) si ricorda l'ultimo valore assunto da D.

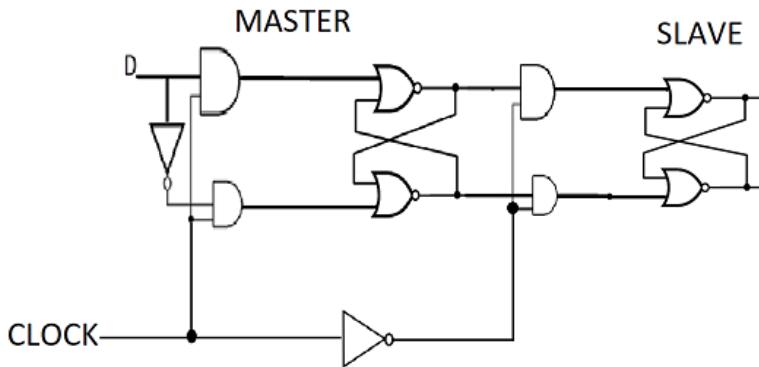
Questa configurazione mi può permettere quindi di memorizzare 1 bit di informazione

(è come se "scattassi una foto dell'ingresso D" in un certo momento, e mantenesse memoria in uscita Q di quanto valesse D in quel determinato momento) (per "scattare" la foto agisco sull'ingresso STROBE: quando esso è "aperto" il dispositivo riproduce in uscita lo stesso valore che c'è in ingresso, quando è "chiuso" mi ricorda il valore che c'era prima, anche se l'ingresso continuasse a cambiare).

Rimanendo sull'analogia della macchina fotografica, come faccio a "scattare una foto istantanea"? cioè, come faccio a memorizzare il valore D ad un certo istante di tempo senza sbagliarmi? Uso una configurazione più complicata ma con caratteristiche simili: raddoppio il numero di flip-flop:

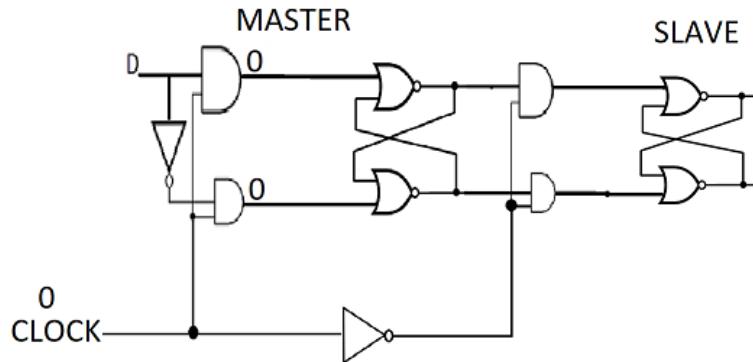


Ho quindi messo 2 flip-flop “uno dopo l’altro”, quindi adesso ho due STROBE, che “unisco” con una funzione NOT, tale funzione fa sì che, qualsiasi valore ci sia in ingresso uno strobe abbia valore 1 ed un altro abbia valore 0, quindi in ogni caso uno STROBE sarà aperto e l’altro sarà chiuso, sempre (quale sarà aperto e quale chiuso dipenderà dal valore in ingresso: se ad un momento x invertto il valore di ingresso \rightarrow lo strobe che era aperto si chiude e quello che era chiuso si apre). L’ingresso che “controlla” l’apertura e chiusura degli strobe viene chiamato “CLOCK”.



Analizziamo il comportamento di questo tipo di flip-flop.

Supponiamo di partire con $CLOCK=0$

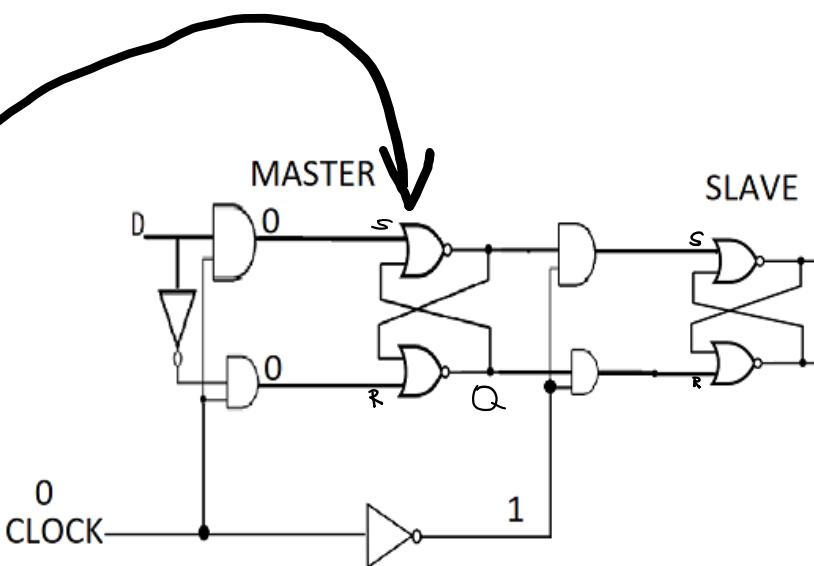


- Abbiamo, quindi, nelle 2 funzioni di tipo AND, il valore 0 e quindi in ingresso alle NOR (S e R) avremo 0 e 0, quindi il circuito sequenziale MASTER è nella condizione di memorizzare una configurazione precedente:

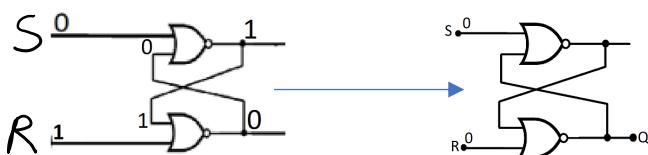
S	R	Q
0	0	Q

Per precisione: l’ingresso D può variare ma la sua variazione non ha alcun effetto sui valori che verranno prodotti in uscita dalle funzioni NOR (poiché sia una che l’altra riceveranno due zeri in entrata, siccome un ingresso della funzione NOR è l’uscita della funzione AND, la quale, qualunque sia D, avrà sempre in uscita 0 se clock è 0)

- Se clock vale 0 il secondo strobe varrà 1 (perché il valore di clock viene negato dalla funzione NOT):

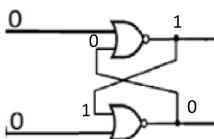


Come vediamo, questo valore 1 va in ingresso alle funzioni AND, che quindi produrranno una uscita in base alla "sequenza di valori precedenti ricordati" del circuito MASTER, se, infatti, nel MASTER la combinazione di valori "ricordati" è $S=0 R=1$, e quindi "passo" da $S=0 R=1$ a $S=0 R=0$:



(ricordandomi il discorso dei ritardi la configurazione sarà quindi la seguente)

avrò in uscita sopra 1 e sotto 0:



E quindi in ingresso alla prima funzione AND avrò 1 (dello strobe) e 1 (dal MASTER), mentre in ingresso alla seconda funzione AND avrò 1 (dello strobe) e 0 (dal MASTER), la prima e la seconda funzione AND produrranno, quindi, rispettivamente i valori 1 e 0. Se LA COMBINAZIONE di VALORI "ricordati" fosse l'altra (cioè $S=1 R=0$) succederebbe l'esatto contrario (avrei 1 0 nella prima funzione AND e quindi 0 in uscita, mentre avrei 1 1 nella seconda funzione AND e quindi 1 in uscita). Quindi, brevemente ho da una parte il valore Q risultante dal passaggio che si fa dalla configurazione precedente a quella attuale, dall'altra parte avrò la sua negazione.

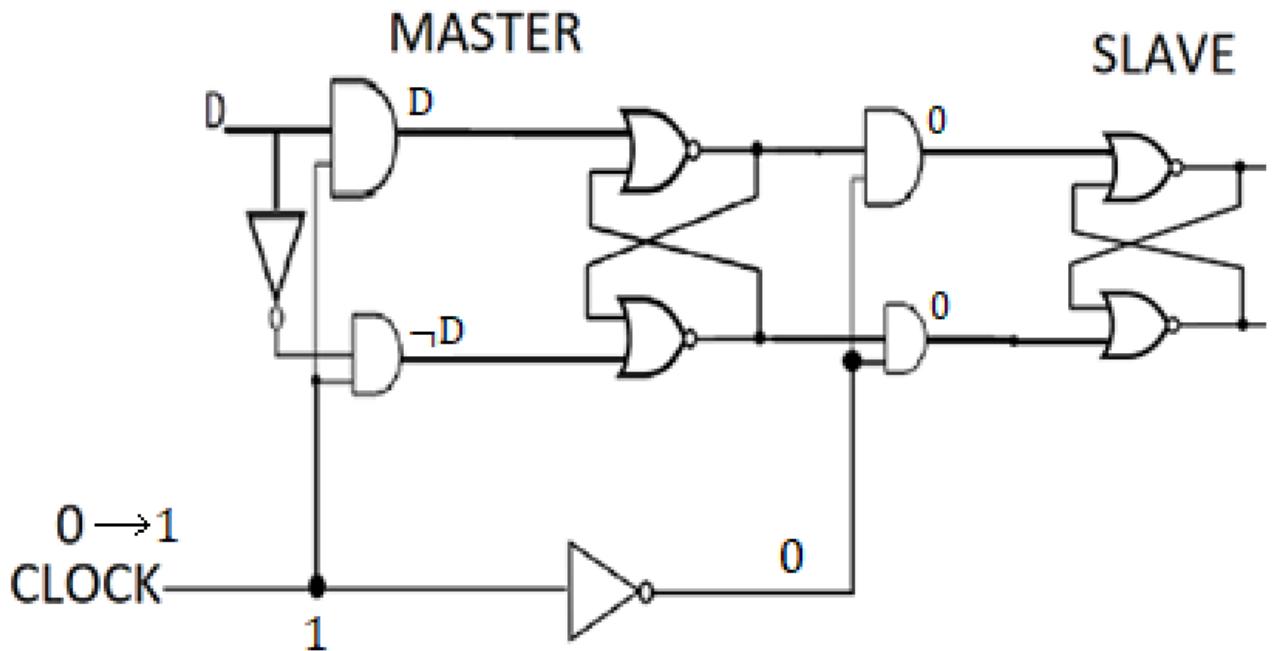
- I valori di ingresso S e R del circuito SLAVE, siccome sono le uscite delle funzioni AND, dipenderanno anch'essi dal valore "ricordato" dal MASTER, avrò da una parte il valore "ricordato" dal MASTER e dall'altra la sua negazione (quindi potrò avere o $S=1 R=0$ oppure $S=0 R=1$)

S	R	Q
0	0	Q
0	1	0
1	0	1

Queste due combinazioni della tavola di verità, quindi.

In sostanza quando $\text{Clock}=0$ il MASTER si "ricorda" ciò che è stato inserito in precedenza, lo slave riproduce lo stesso valore in uscita

Vediamo cosa succede se passo da clock=0 a clock=1

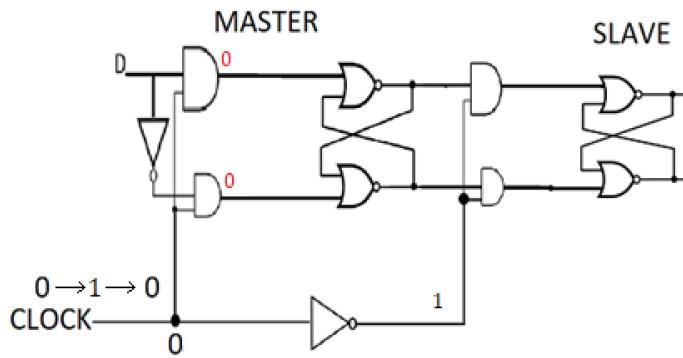


Con clock=1 il circuito MASTER passa nello stato in cui riproduce in uscita lo stesso valore che c'è in ingresso mentre lo slave passa nello stato di "ricordarsi" il valore inserito in precedenza

(si invertono i "ruoli" rispetto a quando clock vale 0)

(ma quindi se ad esempio avessi 0 in ingresso D, con clock=0 avrei il master che ricorda il valore precedente e lo slave che riproduce tale valore, quindi lo stesso valore che c'era in precedenza su D appare in uscita allo slave, mentre passando a clock=1 avrei il master che ha S=0 e R=1, avendo R=1 la seconda funzione NOR dà sicuramente 0, che va in entrata alla prima NOR, che dà 1 in uscita, avendo 0 0, 1 che va in entrata alla seconda NOR, quindi ho 1 in entrata alla prima AND dello slave e 0 in entrata alla seconda AND, tali valori sono però irrilevanti dato che ho in entrambi casi in entrata alle AND anche un 0, che è il valore di clock negato, lo slave "ricorda" quindi il valore precedente, siccome la configurazione del master è S=0, R=1, lo slave produrrà in uscita 0, cioè quello che avevamo posto come valore di ingresso di D, quindi anche in questo caso il valore che avrò in uscita rimane lo stesso che c'è in ingresso)

Quindi in ambedue i casi (sia clock=0 che clock=1) il valore che ho in uscita sarà lo stesso valore D (che c'è in ingresso), cosa devo fare per far sì che cambi il valore in uscita?
Far cambiare di nuovo clock (da clock =1 a clock=0):



Ora il MASTER è di nuovo nella posizione di "ricordare" l' ultimo ingresso D visto in precedenza
Mentre lo slave riproduce in uscita il valore presente attualmente sul MASTER (quindi l'uscita cambia, e diventa il valore presente sull'ingresso D nel momento in cui Clock cambia da 1 a 0)

Possiamo fare un "grafico temporale"
(chiamiamo Q l'uscita del flip-flop complessivo Master-Slave, come abbiamo fatto prima)

Immaginiamo che l'ingresso D vari nel tempo



E vediamo come si comporta l'uscita Q in base al cambiamento del clock

CLOCK _____

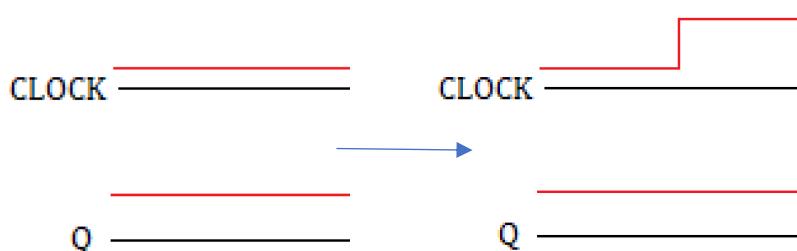
Q _____

Supponiamo che nel momento da noi esaminato Q sia uguale a 1 e CLOCK=0

CLOCK _____

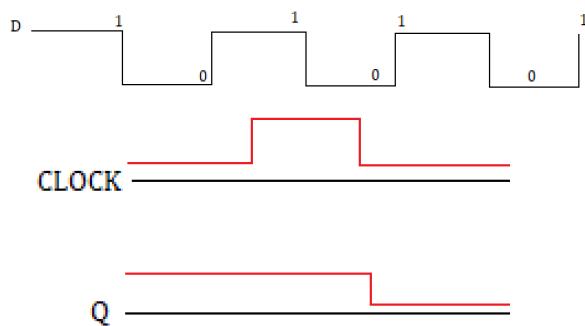
Q _____

Se adesso cambio clock e lo porto a 1, questo come si ripercuote sull'uscita? non ha alcun effetto, come abbiamo visto, Q rimane 1, il valore supposto all'inizio



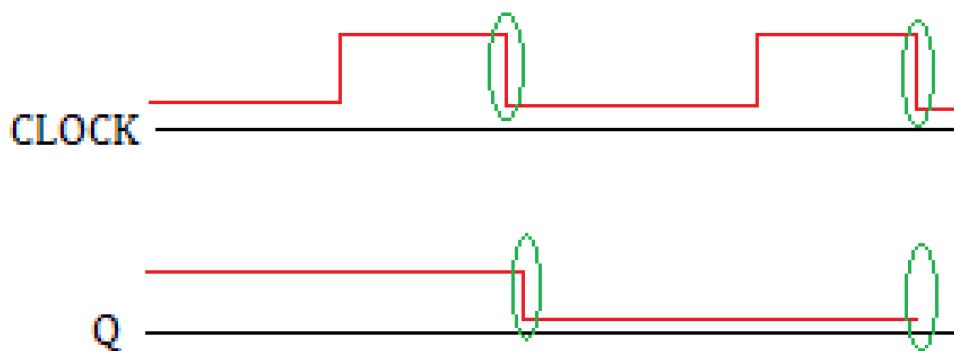
(ma dal punto di vista del nostro master/slave, come abbiamo visto, cambia il ruolo che hanno master e slave, se prima MASTER memorizzava i valori precedenti, adesso è slave che memorizza i valori, mentre il MASTER riproduce lo stesso valore che c'è in ingresso, "segue" quindi le variazioni dell'ingresso D)

Immaginiamo che la variazione di D sia come segue



Se ora riporto il clock a 0, dopo un certo ritardo l'uscita Q assumerà il valore di D al momento del cambio di clock da 1 a 0 (quindi assumerà valore 0, in questo caso)

Q non cambierà, inoltre, fin quando non ci sarà un'altra transizione da 1 a 0 nel valore di clock:



Raddoppiando quindi il numero di flip-flop (usando master/slave) abbiamo ottenuto quello che volevamo: poter identificare un singolo istante di tempo (che corrisponde alla variazione dell'ingresso clock da 1 a 0) in cui viene memorizzato un valore d'ingresso D, che verrà mantenuto costante in "memoria" (in uscita Q), fino a quando clock non passerà di nuovo da 1 a 0.

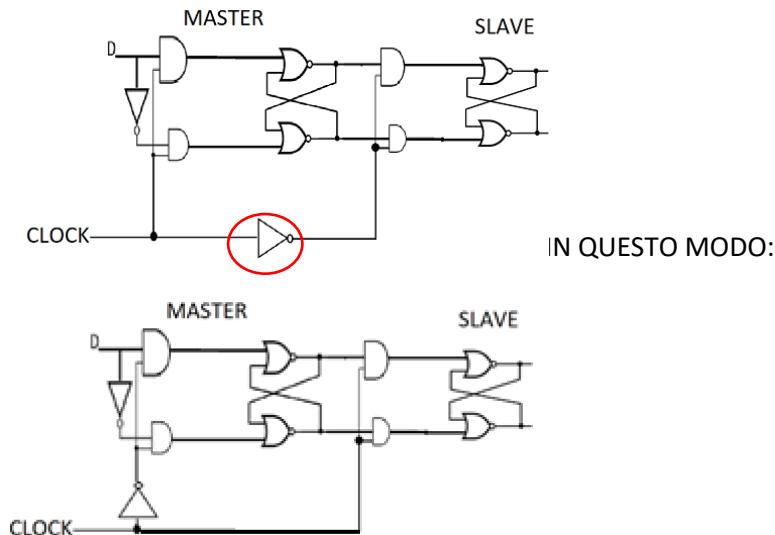
(come abbiamo visto nel grafico, D potrà avere anche altre 100 mila variazioni, ma una nuova variazione verrà "registrata" solo la prossima volta che clock passa da 1 a 0)

(È abbastanza intuitivo anche il perché del nome "clock" per questo controllo)

I flip-flop di tipo master/slave vengono detti dispositivi di tipo "edge-triggered" (i valori in uscita sono determinati non dal valore del clock, ma dalla sua variazione, nel nostro caso da 1 a 0)

(quindi ritornando all'esempio della macchina fotografica, noi "scattiamo la foto" nel momento in cui cambiamo il valore del clock)

Ovviamente potrei ottenere un dispositivo con una funzione simile a questa, solo che la variazione significativa non sia più da 1 a 0, ma sia quella da 0 a 1, basterebbe collegare in modo diverso la funzione NOT



(avendo invertito gli strobe rispetto alla configurazione precedente, ho invertito anche la transizione del clock che si rivelerà significativa al momento di "memorizzare" il valore)

Nego il clock sul master ed affermo il clock sullo slave: transizione significativa è da 0 a 1

Affermo il clock sul master e nego il clock sullo slave: transizione significativa è da 1 a 0

Ulteriore classificazione dei dispositivi per tenere conto delle caratteristiche:

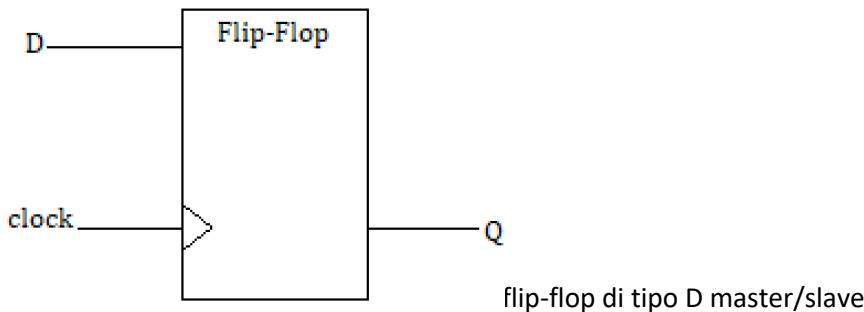
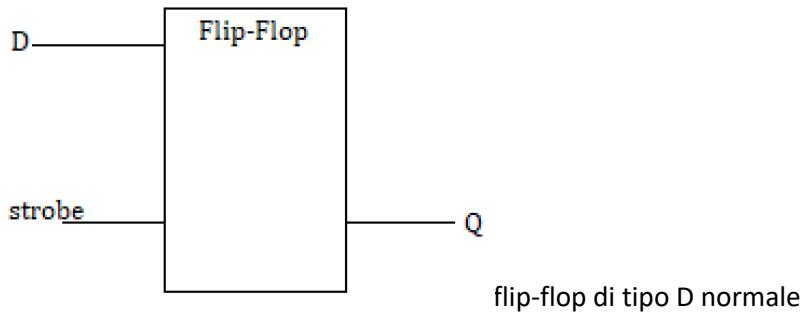
- Circuiti sincroni (che hanno un singolo controllo/variabile in ingresso da cui dipende il valore di uscita, anche se ci sono diversi ingressi che possono variare, essi non determinano nessuna variazione dei valori in uscita, c'è un unico ingresso, la cui variazione "permette" alle uscite di cambiare valore: quello che indichiamo come clock)
- Circuiti asincroni (che non hanno questa caratteristica dei circuiti sincroni, quindi non c'è una unica variabile in ingresso che possa "bloccare" il cambiamento dei valori in uscita)

Quindi tra quelli che abbiamo visto

Sincroni } flip-flop master/slave

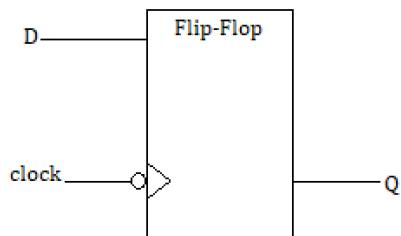
Asincroni } flip-flop set/reset; flip-flop di tipo D

Come differenziare, coi simboli, un flip-flop normale da uno master/slave:



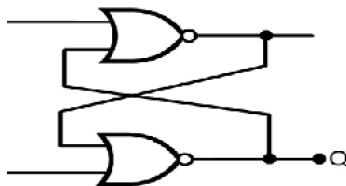
Con il triangolino sull'ingresso clock evidenzio che si tratta di un dispositivo di tipo edge-triggered

Quando voglio indicare che il dispositivo funziona in base alla transizione da 0 a 1 metto il triangolo come nell'immagine, quando voglio indicare che la transizione significativa è quella da 1 a 0 metto il "pallino" di negazione:



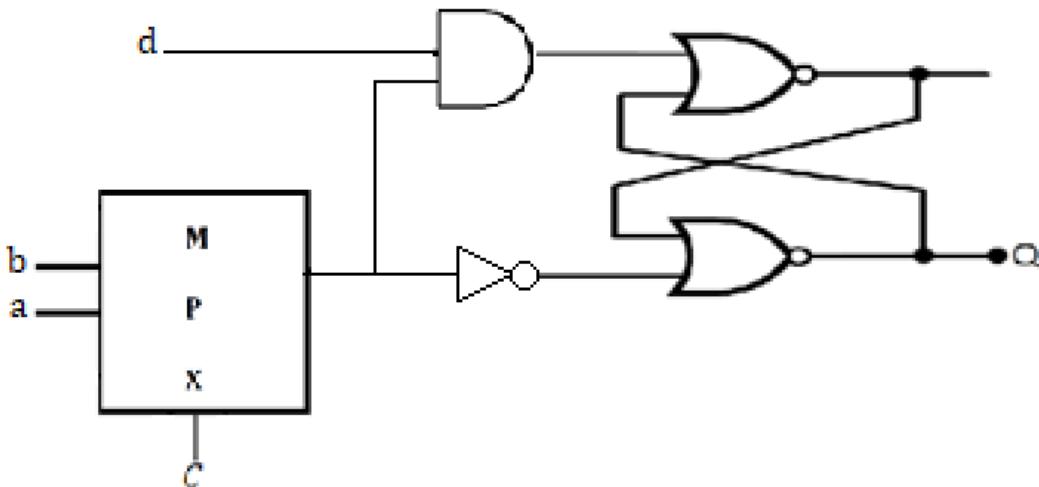
Capiamo meglio la differenza, dal punto di vista pratico, tra l'usare circuiti sequenziali asincroni e sincroni

Partiamo da un set-reset

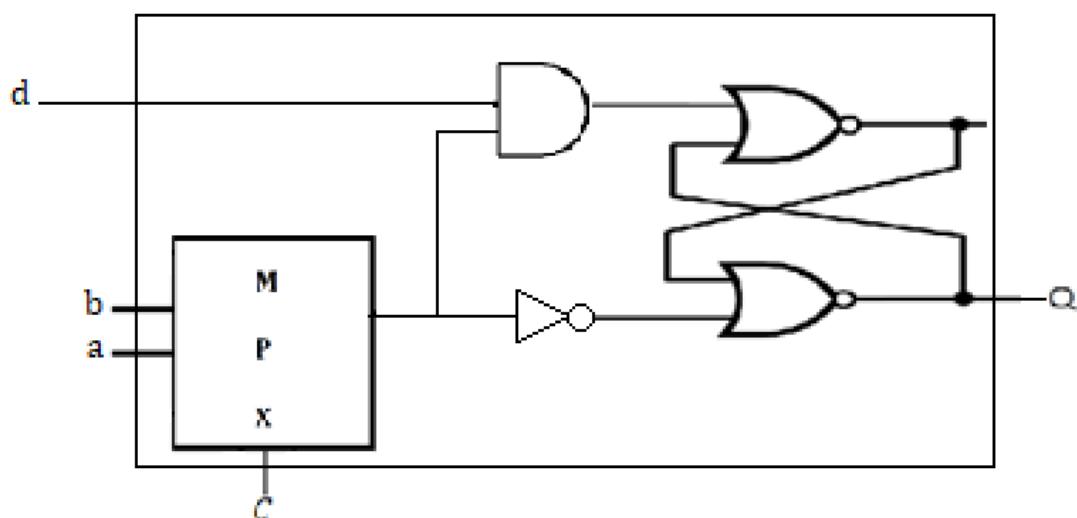


E aggiungiamo delle funzioni combinatorie

(una NOT, una AND, e anche un multiplexer)



Chiudiamo il tutto in una scatola: abbiamo ottenuto un circuito con 4 ingressi ed 1 uscita



Siccome il circuito contiene un flip-flop set/reset possiamo dire a priori che si tratta di un circuito sequenziale asincrono.

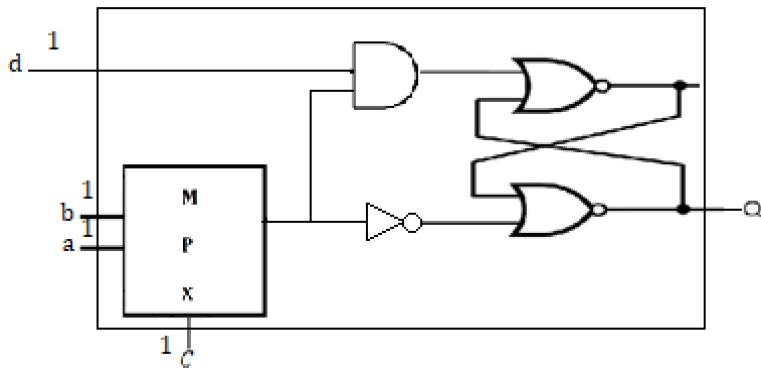
Studiamo il suo comportamento:

immaginiamo una sequenza di tre combinazioni di valori in ingresso

	t=1	t=2	t=3
d	1		
c	1		
b	1		
a	1		

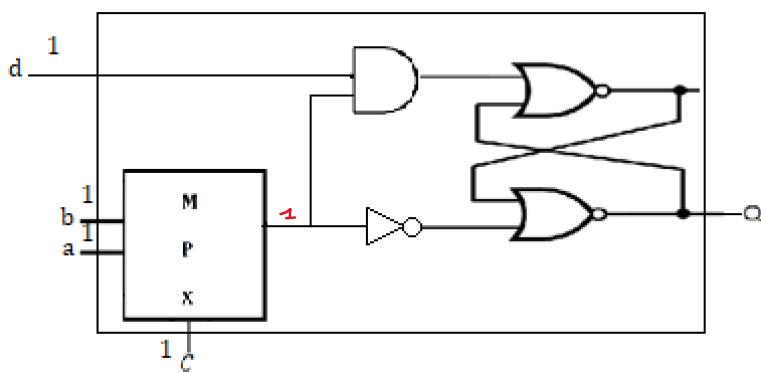
(t: istante di tempo)

Quando tutti gli ingressi assumono valore 1 che succede?

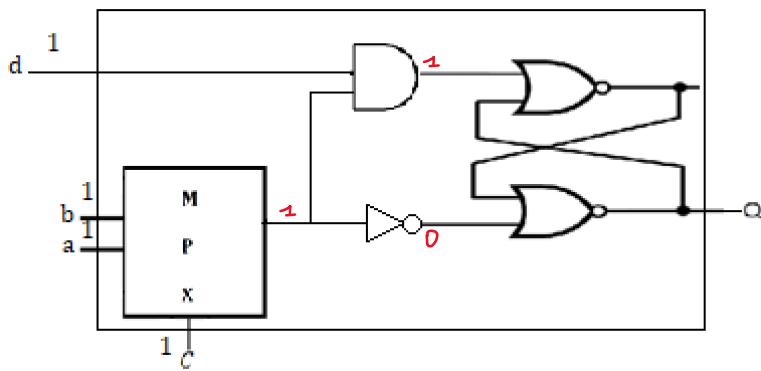


Ci affidiamo alle varie tavole di verità delle funzioni e del multiplexer

Se un MPX a tre ingressi con valore 1 l'uscita sarà **1**

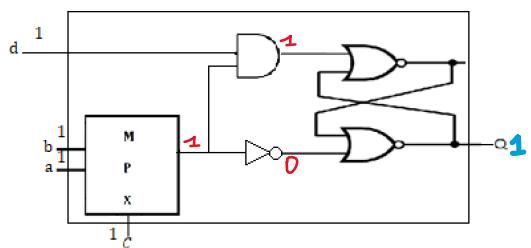


Come vediamo nel circuito tale 1 va in entrata ad una funzione AND, che riceve due 1 in ingresso e quindi produce 1, ed entra anche in un NOT, quindi viene negato ed esce 0



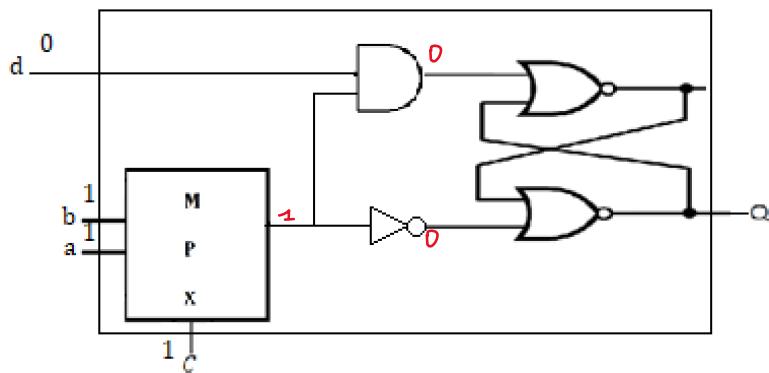
Quindi, per quanto riguarda il nostro flip-flop set/reset, abbiamo S=1 R=0, guardando la tavola del set-reset sappiamo che l'uscita varrà **1**

S	R	Q
1	0	1



Seconda combinazione (d passa da 1 a 0, il resto rimane uguale)

	t=1	t=2	t=3
d	1	0	
c	1	1	
b	1	1	
a	1	1	



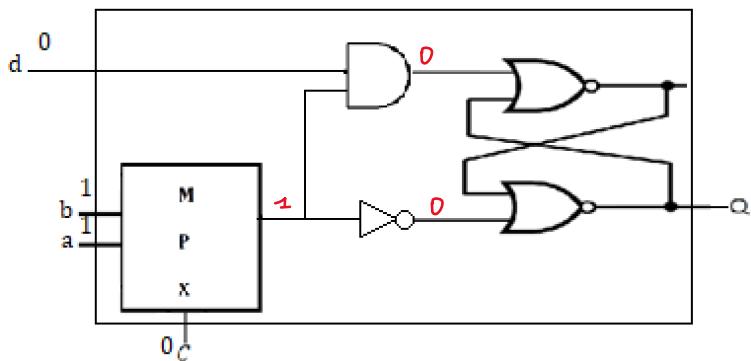
La funzione AND, quindi, ha 1 0 in ingresso, produrrà valore 0, per il resto tutto uguale; quindi, nel set/reset abbiamo S=0 R=0 (siamo nella condizione di memorizzare una configurazione precedente)

S	R	Q
0	0	Q

Quindi l'uscita continua a valere 1

Seconda combinazione (anche c passa da 1 a 0)

	t=1	t=2	t=3
d	1	0	0
c	1	1	0
b	1	1	1
a	1	1	1

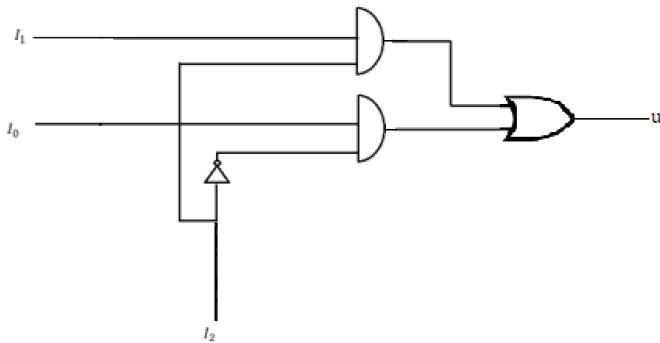


Per quello che abbiamo studiato sui multiplexer non dovrebbe cambiare niente, perché settando l'ingresso di controllo c a 0 vado a "collegare" l'altro ingresso che non ho collegato prima (tra a e b), ma ho comunque

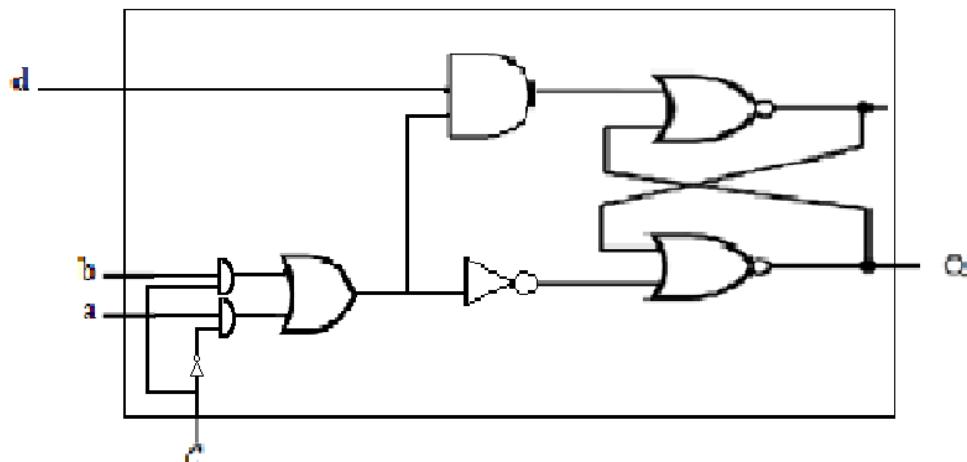
lo stesso valore perché sia a che b valgono 1, quindi in uscita avrò comunque 1, e l'uscita di tutto il circuito complessivo sarà sempre 1.

Osservazione sui ritardi:

Sappiamo che un multiplexer a 2 ingressi + un ingresso di controllo è fatto così (rappresentato in logica a 3 livelli)



Quindi il nostro circuito complessivo si presenta così



Supponiamo che ci sia un certo ritardo δ nelle funzioni AND, OR, NOT e NOR e quindi in tutta la realizzazione

(quindi quando cambio un ingresso, questo cambiamento non si "ripercute" subito sull'uscita, bensì dopo δ unità di tempo)

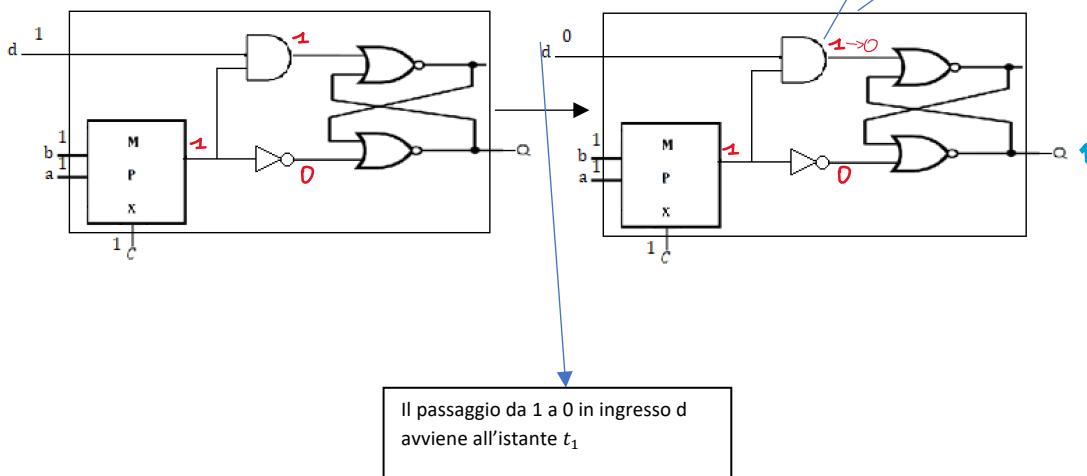
δ è molto piccolo ma non 0

Riesaminiamo il primo cambiamento:

	t=1	t=2	t=3
d	1	0	0
c	1	1	0
b	1	1	1
a	1	1	1

L'uscita di questa funzione AND non passerà immediatamente da 1 a 0 (abbiamo detto che c'è un ritardo δ)

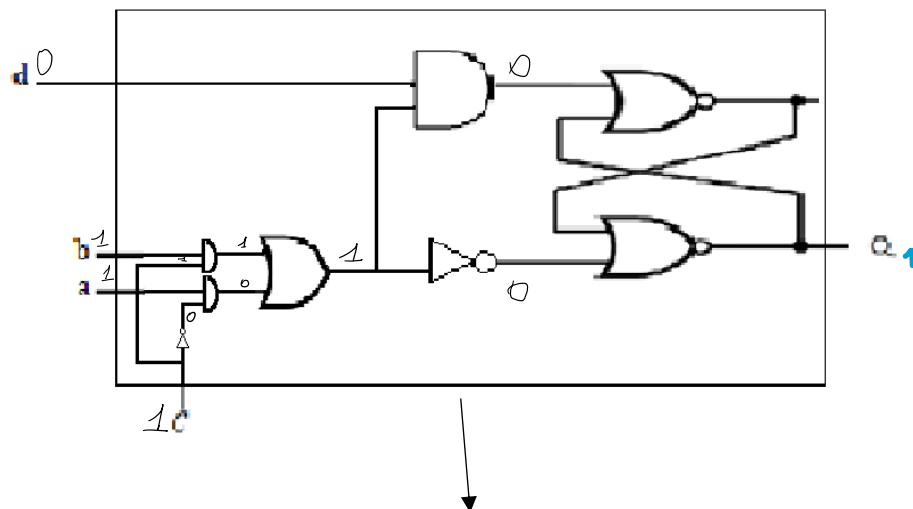
Il passaggio da 1 a 0 avverrà all'istante $t_1 + \delta$

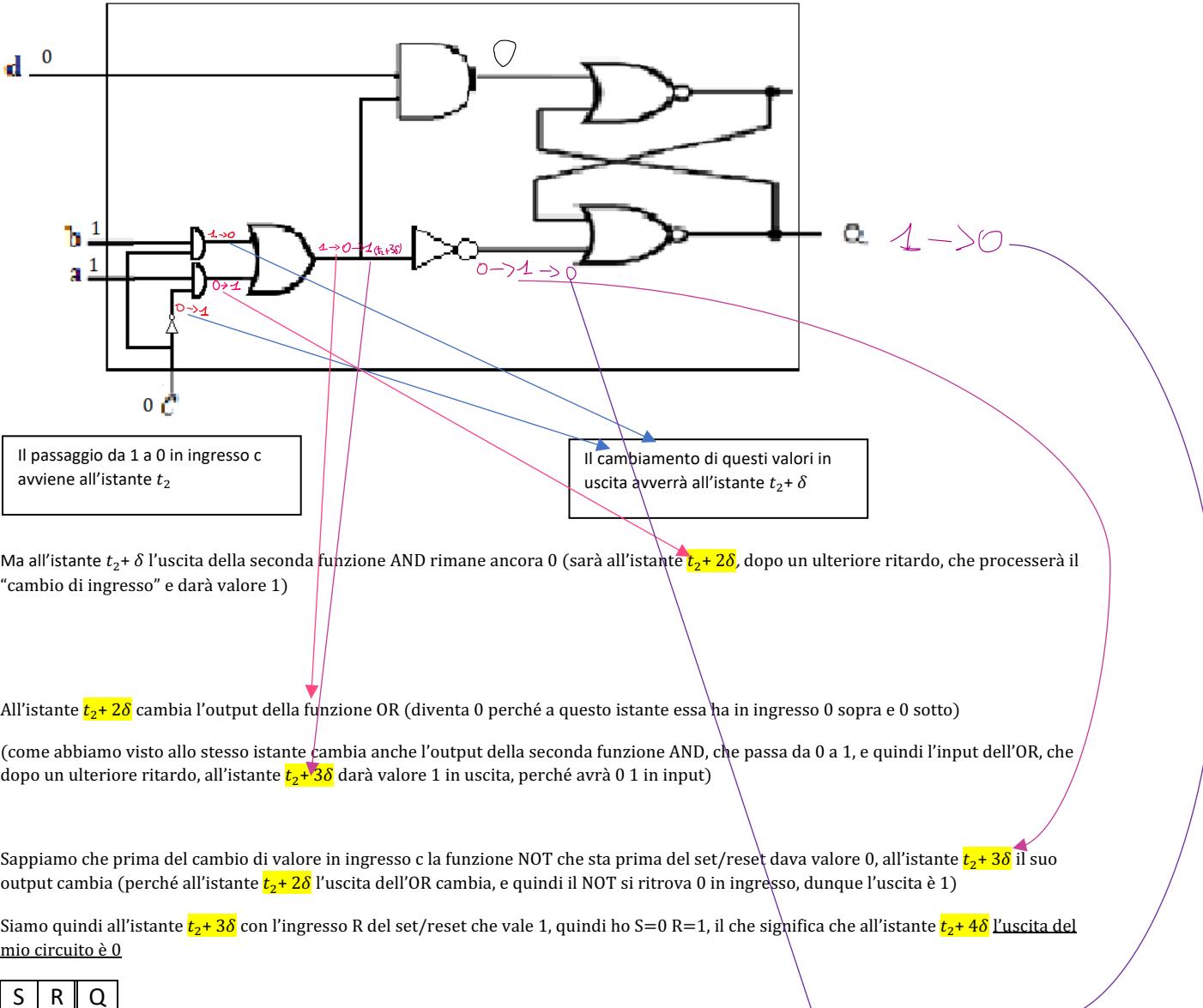


L'uscita del nostro circuito complessivo rimane sempre 1 (a noi non cambia nulla, perché se abbiamo S=1 R=0, come nel primo caso, o S=0 R=0 l'uscita rimane sempre 1, nel secondo caso il circuito si ricorda la precedente)

Analizziamo il cambiamento che avviene da t=2 a t=3

	t=1	t=2	t=3
d	1	0	0
c	1	1	0
b	1	1	1
a	1	1	1





Sappiamo che prima del cambio di valore in ingresso c la funzione NOT che sta prima del set/reset dava valore 0, all'istante $t_2 + 3\delta$ il suo output cambia (perché all'istante $t_2 + 2\delta$ l'uscita dell'OR cambia, e quindi il NOT si ritrova 0 in ingresso, dunque l'uscita è 1)

Siamo quindi all'istante $t_2 + 3\delta$ con l'ingresso R del set/reset che vale 1, quindi ho $S=0$ $R=1$, il che significa che all'istante $t_2 + 4\delta$ l'uscita del mio circuito è 0

S	R	Q
0	1	0

sappiamo che all'istante $t_2 + 3\delta$ l'OR ha 1 in uscita, quindi il NOT si ritrova 1 in ingresso, e quindi all'istante $t_2 + 4\delta$ l'output del NOT diventa di nuovo 0, e quindi ho $S=0$ $R=0$.

So che con $S=0$ $R=0$ sono nella condizione di ricordare una configurazione precedente

S	R	Q
0	0	Q

L'uscita del mio circuito quindi rimane sempre 0.

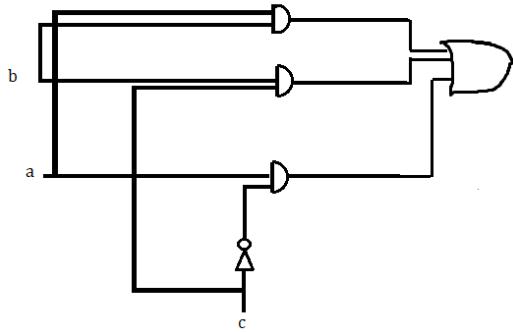
Quindi se teniamo conto dei ritardi (e dobbiamo farlo, per questioni fisiche) questo circuito si comporta in modo diverso da quanto ci aspetteremmo, ci sono sempre degli attimi in cui l'uscita del circuito non rispecchia i valori delle tavole di verità (immediatamente dopo aver cambiato i valori in ingresso), nei circuiti combinatori questo non ci dà tanto fastidio, perché basta che si aspetti il ritardo che serve al dispositivo per ricalcolare il valore giusto e poi si tenga conto solo del valore finale (ignorando quindi quelli intermedi), non possiamo procedere allo stesso modo nei circuiti sequenziali, poiché nei sequenziali i valori intermedi vengono memorizzati.

Non posso pensare di progettare un circuito sequenziale asincrono senza tenere conto dei ritardi delle funzioni (e degli effetti che essi hanno sulla realizzazione)

Questi tipi di problemi sono chiamati "alee di commutazione"

Come posso fare per ovviare a questo inconveniente?

Aggiungere una funzione AND dentro al multiplexer che abbia in input a e b, e la cui uscita venga mandata nella funzione OR, così che, quando a e b valgono 1 ho sempre sicuramente 1 in uscita (indipendentemente da c, che può cambiare) e quindi ho tolto l'alea di commutazione



Quindi devo complicare la realizzazione del mio multiplexer (non è più una realizzazione minimale, aggiungere una AND è ridondante dal punto di vista algebrico, ma mi è utile a evitare)

Ricordiamoci la realizzazione minimale del multiplexer è $m=a \cdot \neg c + b \cdot c$

La nostra realizzazione utile a evitare aleee è $m= a \cdot \neg c + b \cdot c + a \cdot b$

Sapendo che tali errori dipendono anche dalla struttura del multiplexer sappiamo anche che sono risolvibili studiando il funzionamento del multiplexer in questione (caso per caso), ma la via più comoda è evitare sempre questo tipo di realizzazioni

Come evito gli errori delle aleee di commutazione? Usando circuiti sequenziali sincroni, tali dispositivi infatti non sono soggetti a questi errori.

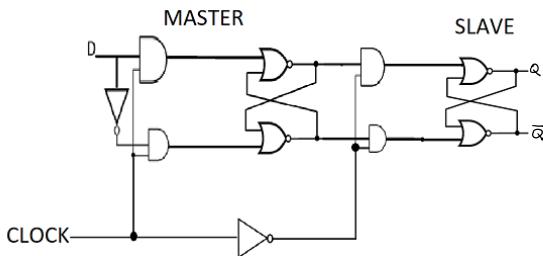
Arriviamo a logica al motivo per cui i sincroni non sono soggetti ad aleee di commutazione:

nei sincroni abbiamo una singola variabile (clock) in base alla quale cambiano le uscite: finché non facciamo variare clock le uscite non possono cambiare

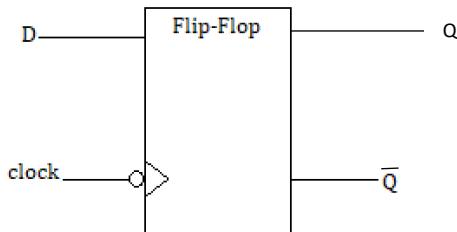
in implementazioni fisiche (le cui parti hanno un ritardo, seppur minimo) possiamo sfruttare questo a nostro vantaggio quando abbiamo variazioni dei valori in ingresso:

blocchiamo i cambiamenti dei valori in uscita, cambiamo gli ingressi, aspettare tutti gli eventuali ritardi, infine far cambiare il clock (così che le uscite possano cambiare)

(così prendiamo in considerazione solo i valori finali prodotti dai dispositivi combinatori, e non anche quelli intermedi). Per i motivi appena esposti, ci occuperemo principalmente di sequenziali sincroni.



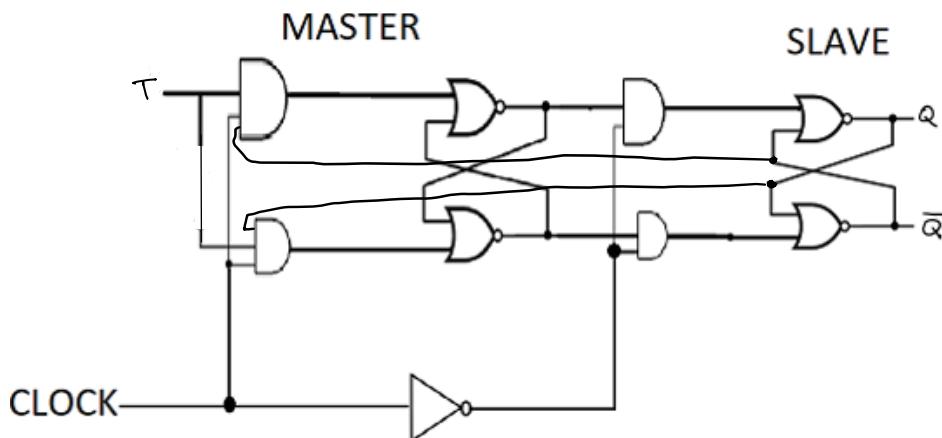
Flip-flop di tipo D master/slave



(per precisione le due uscite del flip-flop sono Q e Q negato)

(La variazione significativa è da 1 a 0)

Partendo da questa configurazione cerchiamo di ottenere dei circuiti sequenziali sincroni diversi

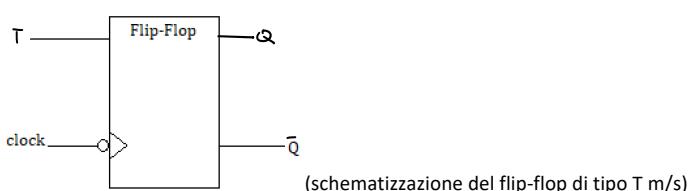


Tolto il not iniziale (non nego più l'ingresso) e mandato le uscite $\neg Q$ e Q in input rispettivamente alla prima e seconda funzione AND

Chiamiamo l'ingresso T, non più D

Tale dispositivo è chiamato flip-flop master/slave di tipo T

Questo circuito avrà un'altra utilità rispetto al D master/slave (il quale ci ricordiamo memorizzava, e riportava in uscita, il valore dell'ingresso D ad un certo istante)



(come in ogni sincrono le uscite potranno cambiare solo a seguito di variazioni della variabile clock)

T si comporta più o meno come D (ma non viene negato) e mi determina gli eventuali cambiamenti: dice che valore può assumere l'uscita quando cambia.

Esaminiamo il funzionamento

Legenda:

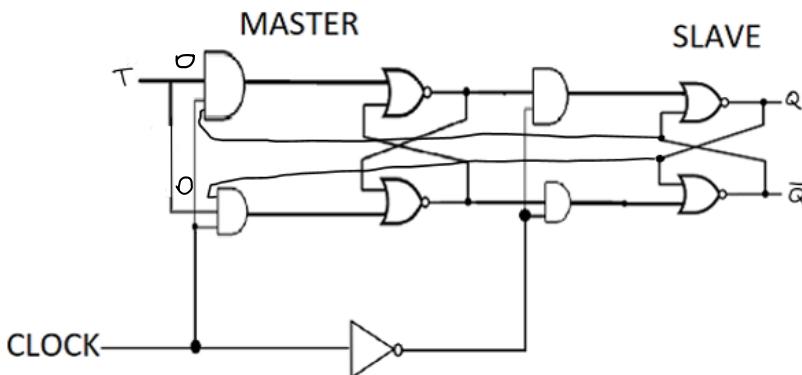


transizione del clock da 1 a 0



transizione del clock da 0 a 1

T	clock	Q
0		Q

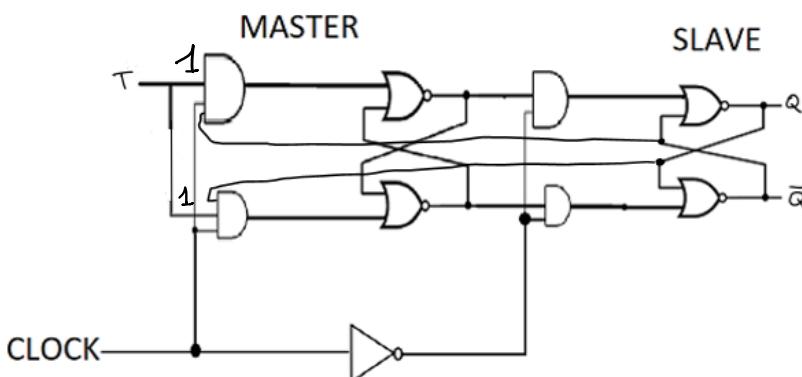


(Le due AND hanno 0 in ingresso, e quindi produrranno 0), quindi ho S=0 R=0 nel master, sono nella condizione

S	R	Q
0	0	Q

T	clock	Q
0		Q
1		$\neg Q$

cosa succede qui in uscita? Perché ho Q negato quando T passa a 1?



Quindi il flip-flop di tipo T master/slave può

a)non fare niente b) negare il bit di informazione che era stato memorizzato in precedenza

(con T=0 tale bit rimane invariato, con T=1 viene invertito) (T=toggle)

Facciamo lo stesso tipo di tabella per il funzionamento del D flip-flop

D	clock	Q
0	↑	0
1	↑	1

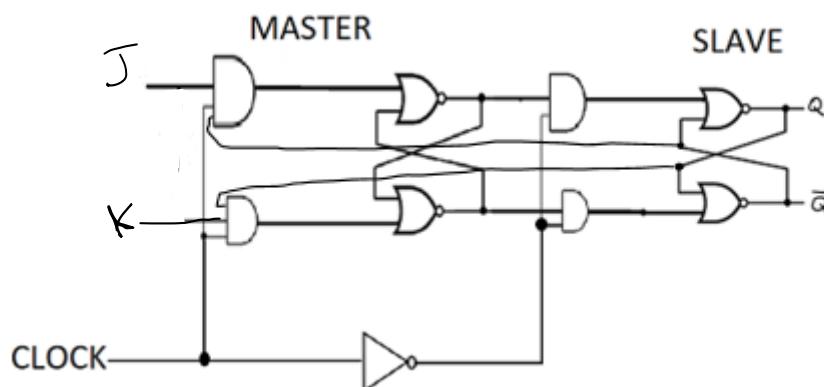
VEDIAMO QUINDI CHE I 2 FLIP-FLOP FANNO DUE COSE MOLTO DIVERSE:

nel D non importa nemmeno che valore c'era in precedenza: in uscita Q avrà il valore che c'è su D al momento di ↑ su clock

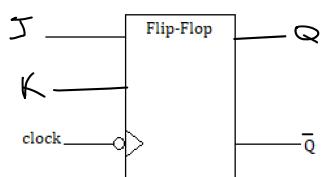
nel T importa eccome il valore che c'era prima: se sono nella condizione di ↑ ed ho T=0 tale valore rimane invariato, se ho T=1 viene invertito.

Variazione sul tema: J K Flip-Flop

Funziona più o meno come il flip-flop di tipo T, solo che ha due ingressi (J e K) invece che un solo ingresso T:



Rappresentazione:



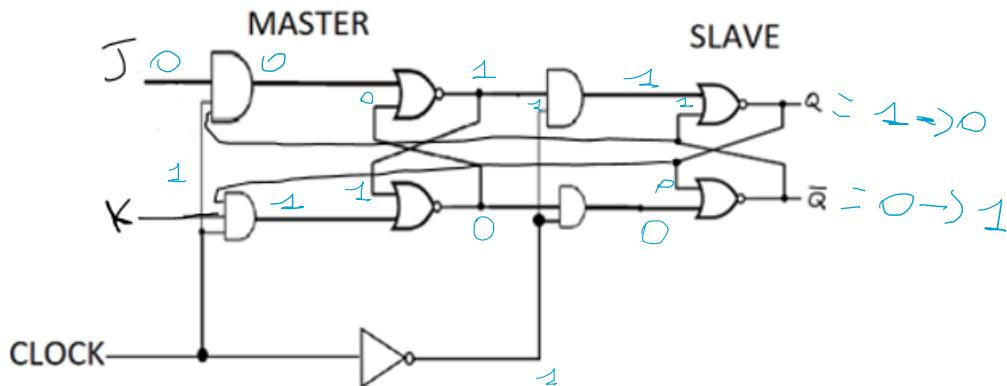
Funzionamento:

J	K	clock	Q	
0	0	↑	Q	Intuibile (è equivalente a T=0)
0	1	↑	0	
1	0	↑	1	Intuibile (è equivalente a T=1)
1	1	↑	¬Q	

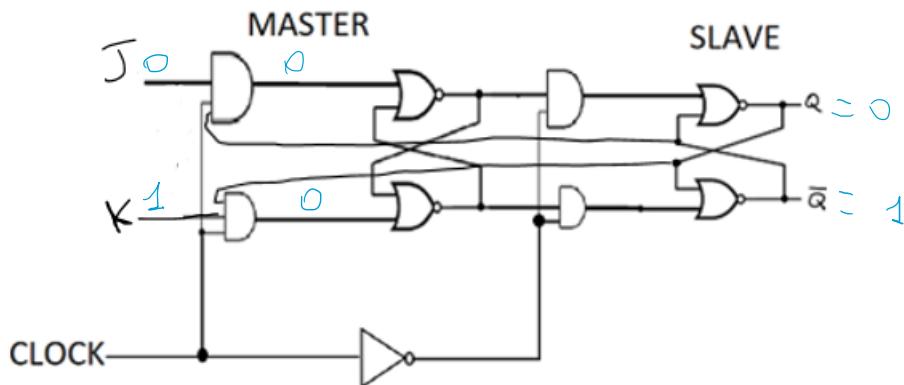
Riga 2 e 3:

- quando J=0 K=1

Ipotizziamo $Q=1 \ -Q=0$



Ipotizziamo $Q=0 \ -Q=1$



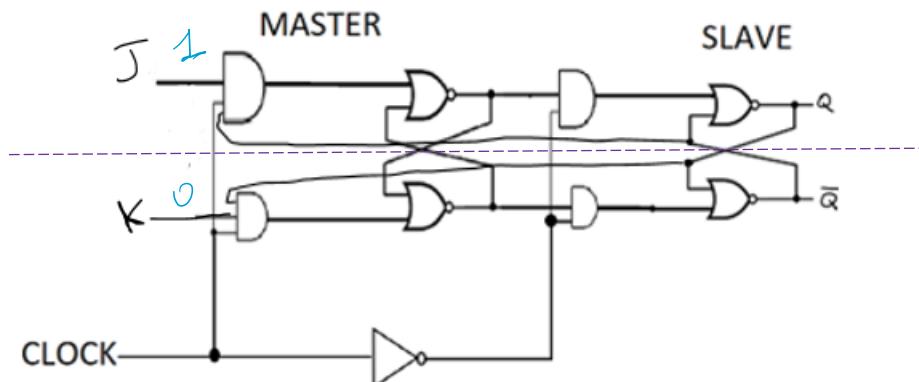
(sono nella condizione di memorizzare

S	R	Q
0	0	Q

 quindi in uscita rimane il valore precedente: $Q=0$)

Quindi con $J=0 \ K=1$, quando $Q=0$ l'uscita resta $Q=0$, quando invece $Q=1$ l'uscita viene negata (diventa $Q=0$), quindi in tutti i casi $Q=0$.

- Con $J=1 \ K=0$



Vediamo che possiamo disegnare una linea di simmetria e “scambiando” J e K quindi otteniamo che il flip flop ha il comportamento duale rispetto a $J=0 \ K=1$

Le prime tre righe sono identiche al set-reset (ma ci ricordiamo che nel set-reset avevamo vietato la configurazione 1 1, l'avevamo vietata perché nel set-reset, essendo esso un circuito asincrono, passare direttamente da 1 1 a 0 0 avrebbe provocato una oscillazione del tipo $0 \rightarrow 1 \rightarrow 0 \rightarrow 1 \rightarrow 0$ ecc), il JK però è

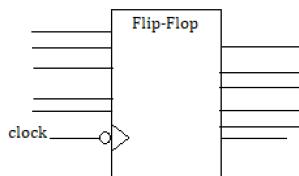
sincrono, quindi non può cambiare i valori a meno che non ci sia una variazione sul clock (adesso l'oscillazione non è più incontrollata, ma diventa una negazione nel momento in cui cambio il clock)

Quindi (sempre se mi trovo nella situazione di \overline{t} nel clock) quando ho 0 0 il valore precedente rimane invariato, quando ho 1 1 inverte il valore precedente, quando ho 0 1 inserisce il valore 0, quando ho 1 0 inserisce il valore 1)

È come se fosse una “variante sincrona” del flip-flop di tipo set-reset

(Come possiamo ben notare l'unica differenza tra T flip-flop e JK flip-flop è il numero di fili in input, la realizzazione interna è la stessa, quindi il costo per realizzare un T flip-flop o un JK flip-flop sarebbe lo stesso, se non fosse per il numero di ingressi)

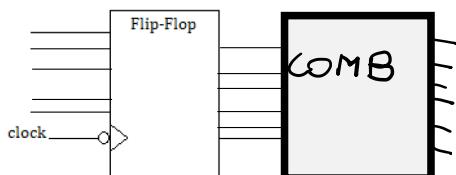
Partiamo dal master-slave come “nucleo centrale” per realizzare un circuito sequenziale sincrono arbitrario esempio



Come posso fare per realizzare operazioni più “complicate” a mio piacimento (senza stravolgere il funzionamento sincrono del circuito)?

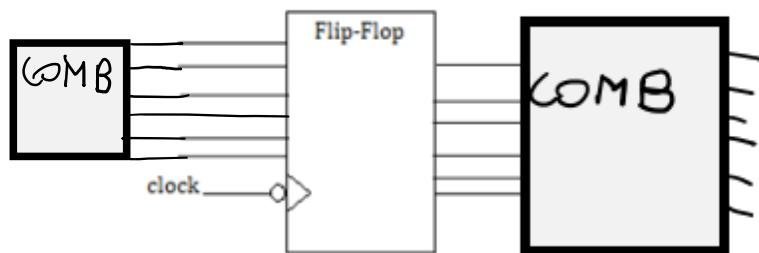
(vogliamo mantenere la proprietà che ci garantisce che solo a seguito di una variazione del clock le uscite possano cambiare)

Prendo le uscite e le mando in ingresso ad un circuito combinatorio



Posso inoltre aggiungere un altro circuito combinatorio prima del mio flip-flop sequenziale, in modo da “calcolare” cosa mandare in ingresso al mio flip flop

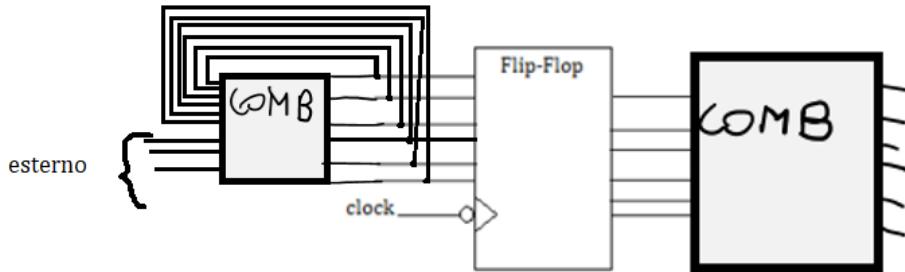
Ad esempio se ho un JK flip flop (invece di scegliere arbitrariamente i valori di J e K) posso mettere un circuito combinatorio, le cui uscite vengano mandate in ingresso al mio JK flip-flop



(così quando cambiano le uscite del primo comb cambiano gli ingressi del Flip-Flop)

Gli ingressi del primo circuito combinatorio potrebbero essere così suddivisi

- Un "sottoinsieme" di ingressi che provengono dall'esterno
- Un altro "sottoinsieme" che viene dalle uscite del flip-flop



E quindi una parte degli ingressi può variare in qualsiasi momento (quelli che arrivano dall'esterno), un'altra parte potrà variare solo a seguito di variazioni del clock

Se cambio la frequenza con cui varia il clock avrò dispositivi più lenti/più veloci

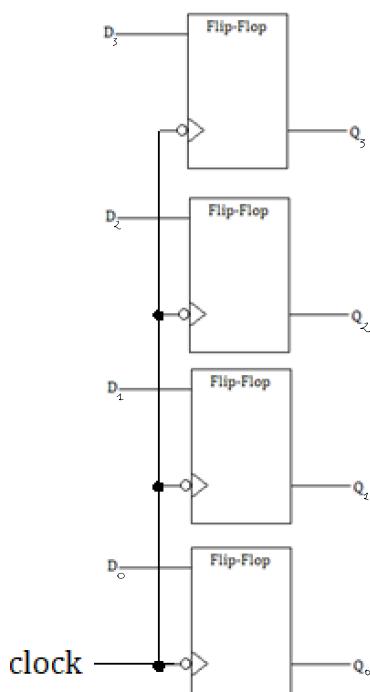
(ma non posso far variare il clock troppo velocemente senza tener conto del tempo di ritardo dei singoli dispositivi che compongono il circuito, in modo da assicurarmi che i circuiti combinatori abbiano calcolato il loro valore finale, quello delle tavole di verità)

Come abbiamo già visto il blocco fondamentale da cui partire per costruire un circuito sequenziale sincrono è il Master/Slave flip-flop

Esempi più semplici di circuiti sequenziali sincroni:

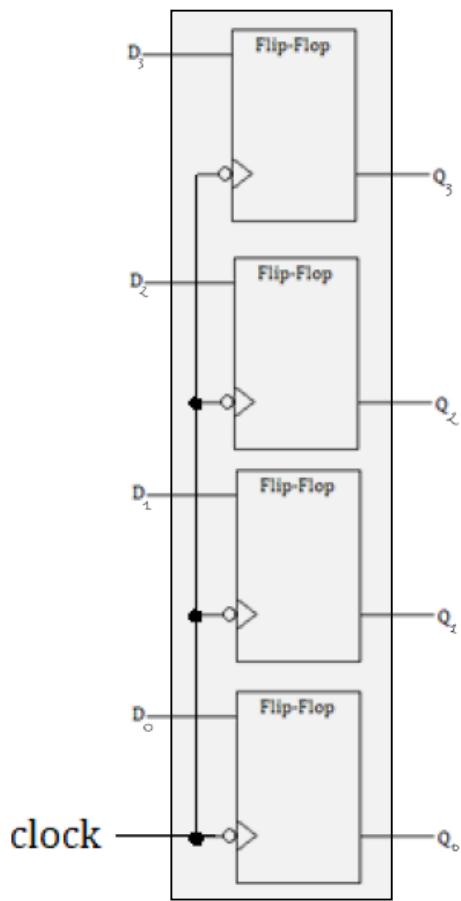
Registri.

Esempio: registro D (più flip-flop di tipo D M/S collegati fra loro)



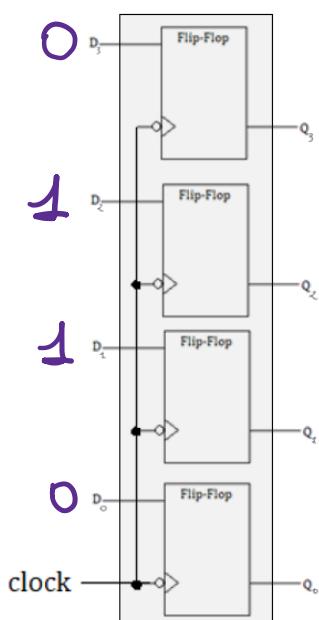
Possiamo numerare i flip-flop (e quindi anche le rispettive uscite)

Racchiudendo il tutto in una scatola abbiamo un registro di tipo D a 4 bit (sappiamo che il D flip-flop memorizza 1 bit di informazione, quindi il nostro registro sa memorizzare configurazioni binarie da 4 bit)



Cosa ci posso fare ad esempio?

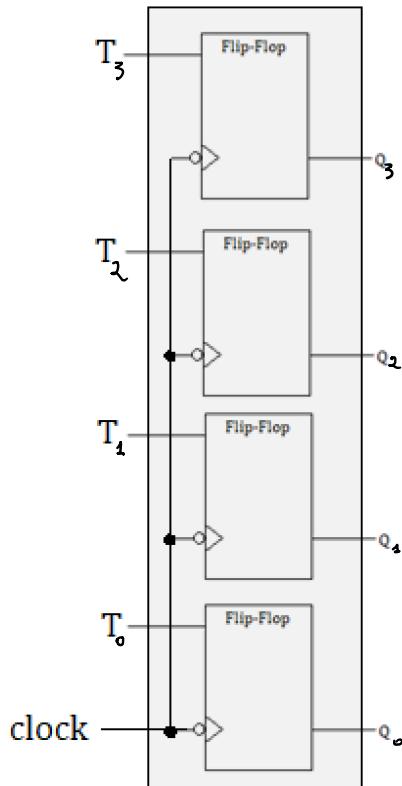
Ho una configurazione binaria iniziale 0110, trasmetto un impulso al clock (lo faccio cambiare da 0 a 1, e poi da 1 a 0) ed il registro memorizza in uscita la configurazione che c'è in ingresso al momento 



Poi posso cambiare i valori in ingresso in maniera arbitraria, e questo cambiamento non avrà nessun effetto sull'uscita (fino a quando non ci sarà un altro shift del clock)

Quindi è un dispositivo di memoria.

Proviamo a fare un registro che abbia come moduli di base dei T flip-flop



È praticamente un contatore (si chiama così infatti):

usando infatti più flip-flop di tipo T andrò ad incrementare il valore memorizzato ad ogni variazione del clock

quindi $\nabla N \rightarrow N+1$

- Cominciamo dal bit meno significativo (T_0): cosa dobbiamo farci per passare da N a $N+1$?

Bisogna invertirlo (sappiamo che nella rappresentazione binaria un numero dispari finisce con 1, un numero pari finisce con 0, sappiamo che vogliamo passare da un numero al suo successivo, quindi se partiamo da un numero pari dobbiamo far sì che il suo bit meno significativo sia 1, se partiamo da un dispari dobbiamo far sì che il bit meno significativo sia 0, sapendo che il successivo di un pari è dispari ed il successivo di un dispari è pari), quindi mettiamo $T_0 = 1$ (ci ricordiamo che nel T flip-flop se ho $T=1$ inverto il bit memorizzato precedentemente)

- Per quanto riguarda T_1 se il numero che voglio incrementare è pari, allora T_1 DEVE RIMANERE INVARIATO, mentre se il numero che voglio incrementare è dispari T_1 DEVE ESSERE INVERTITO,

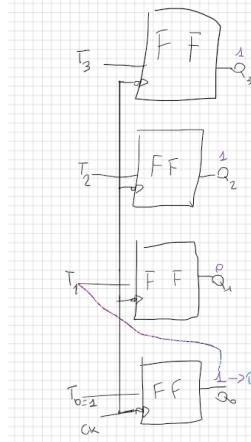
quindi se $T_0 = 0$ allora T_1 deve rimanere invariato,

se $T_0 = 1$ allora T_1 deve essere invertito,

quindi in entrata al secondo flip-flop dovrò mettere l'uscita Q_0 , siccome con $T=1$ INVERTO e con $T=0$ lascio INVARIATO. (qui per uscita Q_0 si intende il valore che l'uscita Q_0 ha prima del "colpo di clock", quindi il bit meno significativo della rappresentazione da cui partiamo)

(vogliamo ad esempio passare dal numero 13 al numero 14, quindi stiamo partendo dalla configurazione binaria 1101: abbiamo detto che invertiamo il bit meno significativo: avremo quindi __ 0 (lo abbiamo invertito facendo passare il valore 1 su T_0), inoltre siccome abbiamo un numero dispari dobbiamo anche invertire il secondo bit meno significativo, quindi prendiamo l'uscita Q_0 che, prima del cambio di clock vale 1, e la portiamo in ingresso a T_1 così da invertire il bit che sta in quella posizione, siccome sappiamo che avere 1 in ingresso vuol dire invertire)

Config precedente 1101

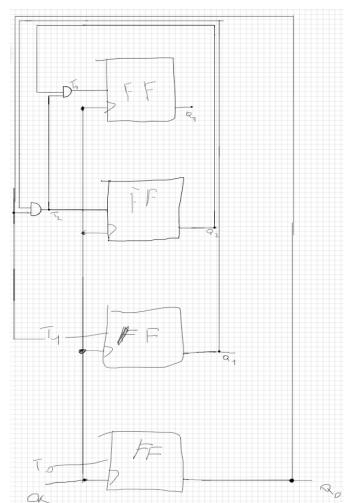


- Cosa succede con T_2 ? Ricordiamoci il discorso sui riporti fatto quando parlavamo di rappresentazioni binarie e circuiti combinatori (sommatori), riportiamo una parte sul funzionamento dei carry look-ahead:

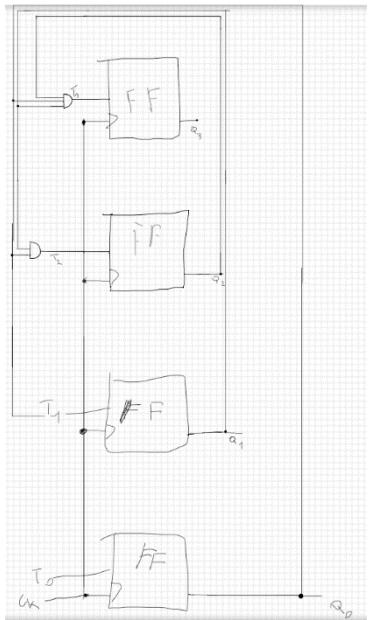
Facciamoci una idea di come possono essere fatti i circuiti di previsione del riporto, quando è che viene generato un riporto sul 3° bit? Quando il risultato dell'operazione "eccede" il numero di valori rappresentabili su un solo bit, ad esempio se a_1 e b_1 hanno valore 1, in uscita viene prodotto un valore (10) che non può essere scritto su un solo bit, quindi viene generato un riporto; allo stesso modo se ho un riporto già dall'operazione precedente (e quindi a_0 e b_0 valgono entrambi 1) basta che solo uno tra a_1 e b_1 abbiano valore 1 perché si generi un riporto sulla posizione di mio interesse (poiché ho già un bit di riporto da prima che sommo con un 1 adesso)

quindi se i due valori d'uscita dei due flip flop precedenti hanno valore 1 devo invertire il valore di T_2 , per fare ciò basta realizzare una funzione AND che entri in ingresso a T_2 e che abbia come ingressi i due valori di uscita dei primi due flip-flop (Q_0 e Q_1) (così se entrambi valgono 1 avrà 1 in output dell'AND, e quindi in input T_2 e quindi invertirò il bit di informazione in questione, mentre se anche solo uno dei due bit vale 0 l'uscita dell'AND varrà 0 e quindi il bit in questione rimarrà invariato)

- Per quanto riguarda T_3 perché venga generato un riporto dobbiamo avere tutti 1 nei precedenti flip-flop, dobbiamo realizzare una AND a 2 ingressi, che, intuitivamente in ingresso avrà l'uscita Q_2 e l'uscita dell'AND tra Q_0 e Q_1 , così, se sia Q_0 che Q_1 valgono 1 allora avremo 1 in uscita da quest'AND.



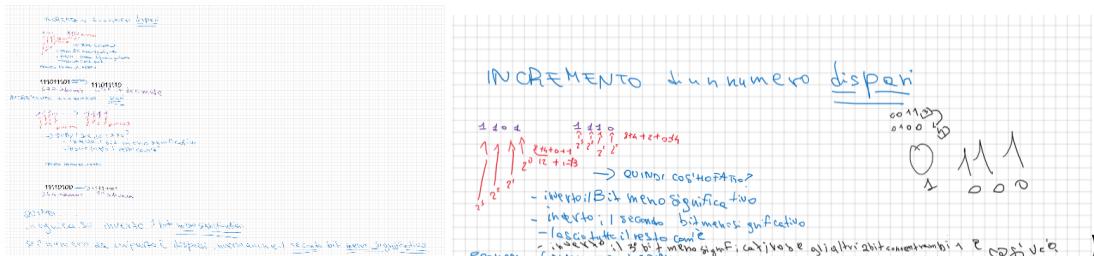
potevo anche fare in modo diverso (non risparmiando sul numero di ingressi della funzione ma guadagnando in termini di velocità del dispositivo): mettere direttamente Q_0 , Q_1 e Q_2 nella funzione AND, avendo quindi 3 ingressi:



ovviamente il secondo metodo comporta un livello di complessità maggiore rispetto al primo (più ingressi) ma anche una velocità maggiore di produzione del risultato finale (non devo aspettare l'uscita del primo AND, perché metto direttamente i valori Q_0 , Q_1 e Q_2 in ingresso al secondo AND, ho un solo livello di logica, quindi ho meno ritardo), in termini di velocità non c'è una così grande differenza in questo caso, che siamo a registri a 4 bit, ma immaginiamo di star lavorando su 32 bit; il primo metodo sarebbe circa 30 volte più lento del secondo: perché per quanto riguarda il bit più significativo avrà una crescita del tempo proporzionale al numero di bit contenuti all'interno del registro.

Col secondo metodo infatti avrei come ingresso del flip-flop che calcola il bit più significativo l'uscita di una AND a 31 ingressi, ma avrei un dispositivo molto più veloce rispetto a uno realizzato col primo metodo, che mi dovrebbe "far aspettare" per l'uscita delle 30 funzioni AND precedenti prima di produrre il risultato del bit più significativo

(Breve spiegazione dell'incremento:



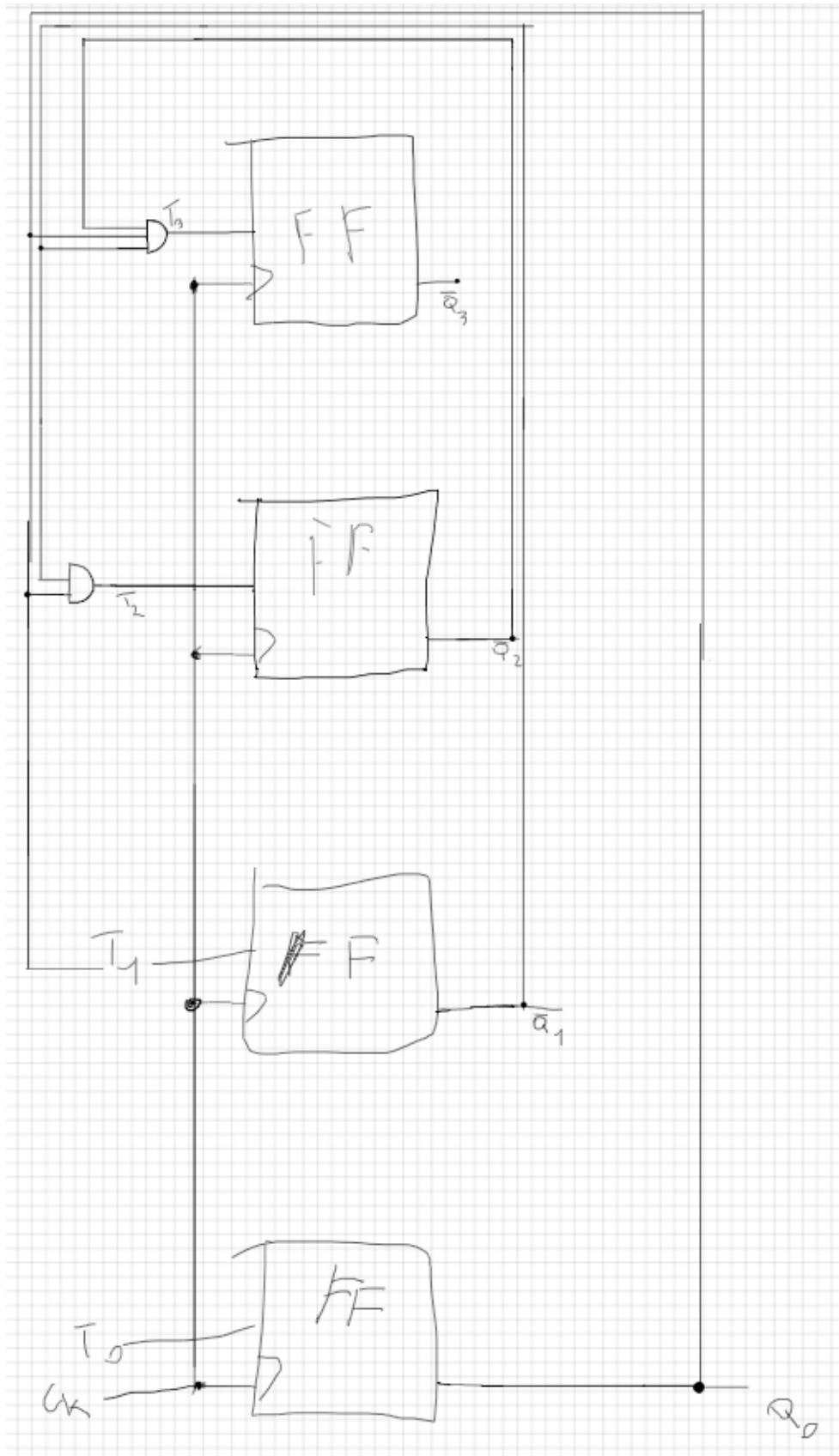
Come faccio per fare decremento?

Come creo un counter che conti alla rovescia?

Uso lo stesso modello appena usato, ma le uscite che mi interesseranno saranno i vari $\neg Q_i$ (non più Q_i): so che i flip-flop di tipo T mi producono Q e $\neg Q$ (prenderò in considerazione queste ultime).

Per quanto riguarda il bit meno significativo non cambia niente: anche decrementare significa dover passare da un numero pari a uno dispari/viceversa: quindi dover invertire in ogni caso il bit meno significativo.

Per le altre uscite dovrò considerare Q negato e non Q, come appena detto.

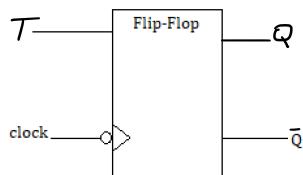


Quindi sappiamo come realizzare dispositivi per contare in avanti/all'indietro, ma quale è il valore iniziale da cui partiamo?

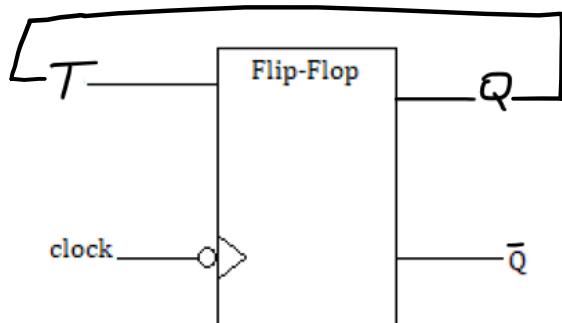
(sappiamo che i flip-flop di tipo T, che compongono i nostri counters, fanno una cosa: lasciano invariato oppure negano un valore esistente in precedenza, ma quale è questo valore?)

Come facciamo per esempio a "far partire il conteggio" da 0? Del tipo contare 0, 1, 2, 3, 4, 5, ecc

Prendiamo il nostro flip-flop di tipo T:



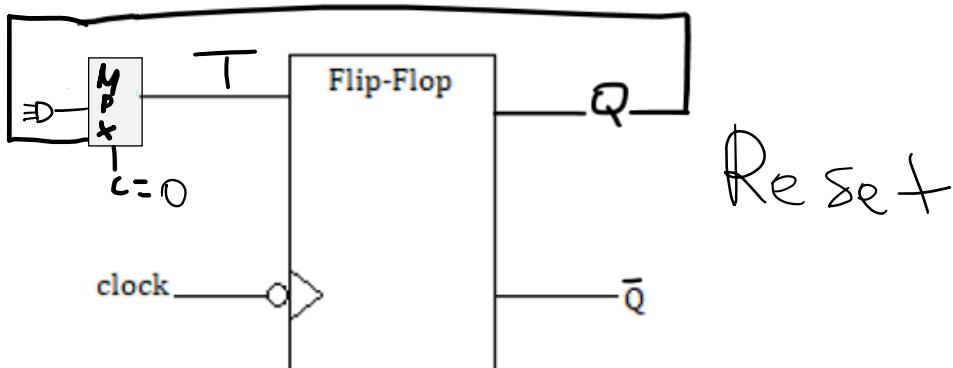
Riusciamo, data una configurazione iniziale che non conosciamo, a produrre (in seguito ad un impulso sul clock) il valore 0? Sì, basta portare Q in ingresso a T; in questo modo, se Q vale 1, anche T varrà 1 e quindi si dovrà invertire l'1 precedente (e quindi produrremo 0), mentre se Q vale 0 anche T varrà 0 e quindi bisognerà lasciare invariato lo 0 precedente (e quindi l'uscita rimane 0)



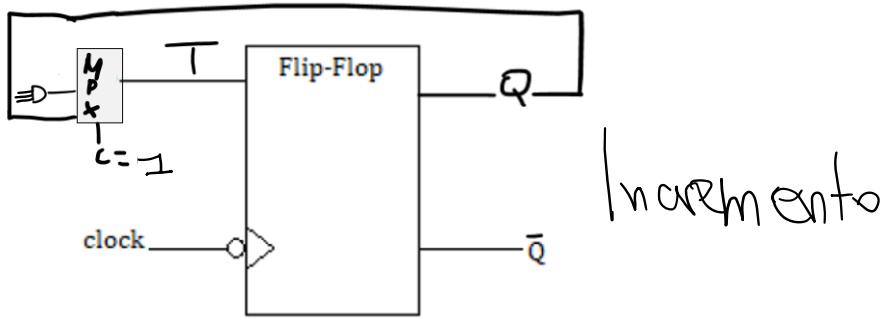
(quindi abbiamo di fatto ottenuto la funzionalità reset: azzeramento del flip-flop)

Come faccio ad "unire" queste due fasi? (azzerare il registro e poi contare)

Posso usare un multiplexer a due ingressi con il quale, attraverso un ingresso di controllo scelgo una o l'altra funzionalità (collego o l'ingresso legato all'azzeramento o l'altro ingresso)

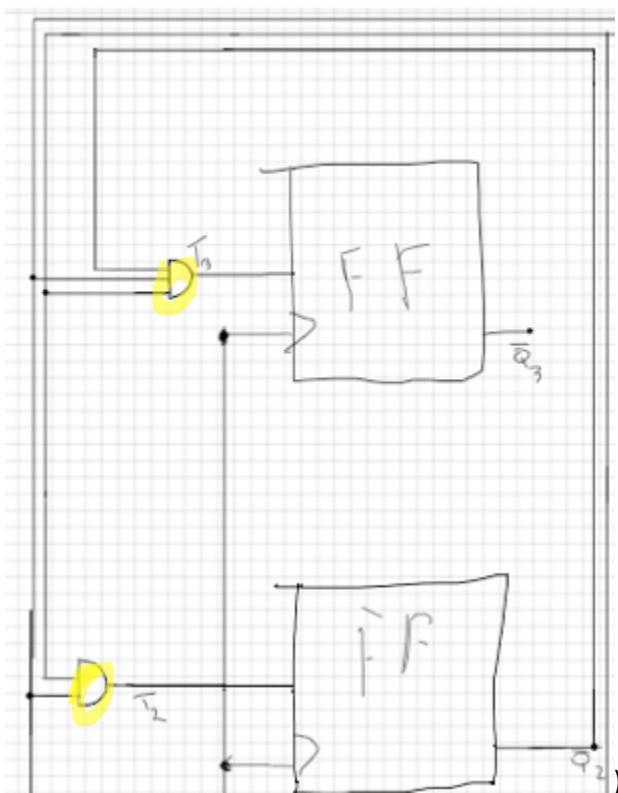


(la funzione AND è l'AND di tutte le cifre precedenti)



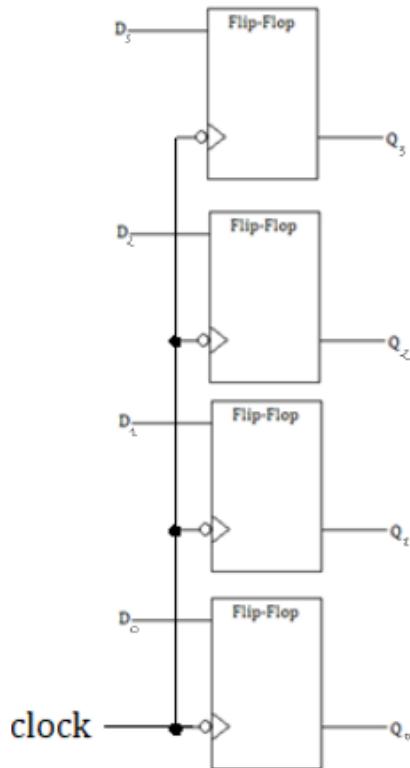
(quando l'ingresso di controllo c del multiplexer è 0 il Flip-Flop riceve in entrata Q , quindi viene azzerato, mentre quando il controllo è 1 il mio Flip-Flop riceve in entrata l'AND di tutte le cifre precedenti, così incrementa di 1)

(ovviamente nei primi due bit non ci sarà questa funzione AND: nel caso del bit meno significativo ci sarà la costante 1; devo sempre invertire come già visto, nel caso del secondo bit meno significativo ci sarà l'uscita Q_0 , dal terzo bit meno significativo in su avremo bisogno dell'AND, come abbiamo già visto nella struttura del dispositivo di incremento: sono queste AND qui, per intenderci



Possiamo continuare su questa linea di pensiero: se usassimo un multiplexer a 4 vie (invece che a 2) potremmo usarlo per ampliare il numero di funzionalità del nostro dispositivo: oltre a incremento e reset, potremmo avere un ingresso che, se scelto dal controllo c , farà fare il decremento, oppure un altro che farà mantenere il valore precedente.

Abbiamo visto i registri semplici (dispositivi di memoria), i contatori, ora vediamo i “registri di scorrimento” (shift)



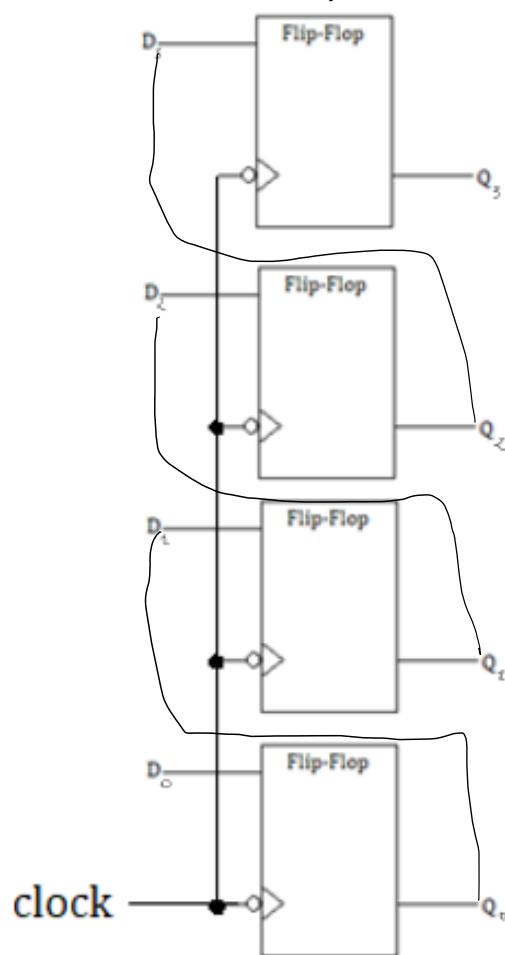
Questa è la nostra base di partenza. Prendiamo ogni uscita e la colleghiamo all'ingresso del flip-flop successivo: Q_0 andrà in ingresso T_1 e così via...

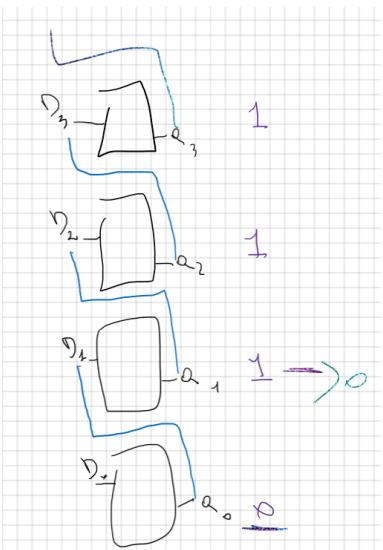
Se metto la costante 0 in ingresso D_0 cosa succede all'impulso del clock?

La configurazione binaria memorizzata dentro questo registro viene moltiplicata per 2:

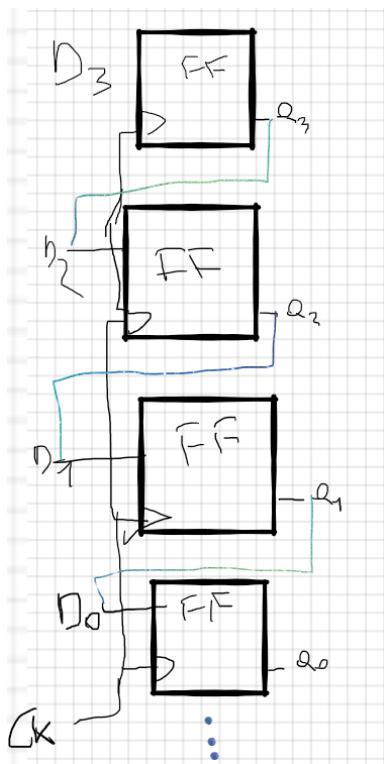
$N \rightarrow 2N$

(ovviamente se aggiungiamo uno zero ad una qualsiasi configurazione binaria la stiamo raddoppiando; tutte le cifre si "spostano più a sinistra di una posizione": esempio 1101101 (rappresentazione binaria di 109), aggiungo uno 0 quindi ho 11011010 (rappresentazione di 218))





Se facessi al contrario:

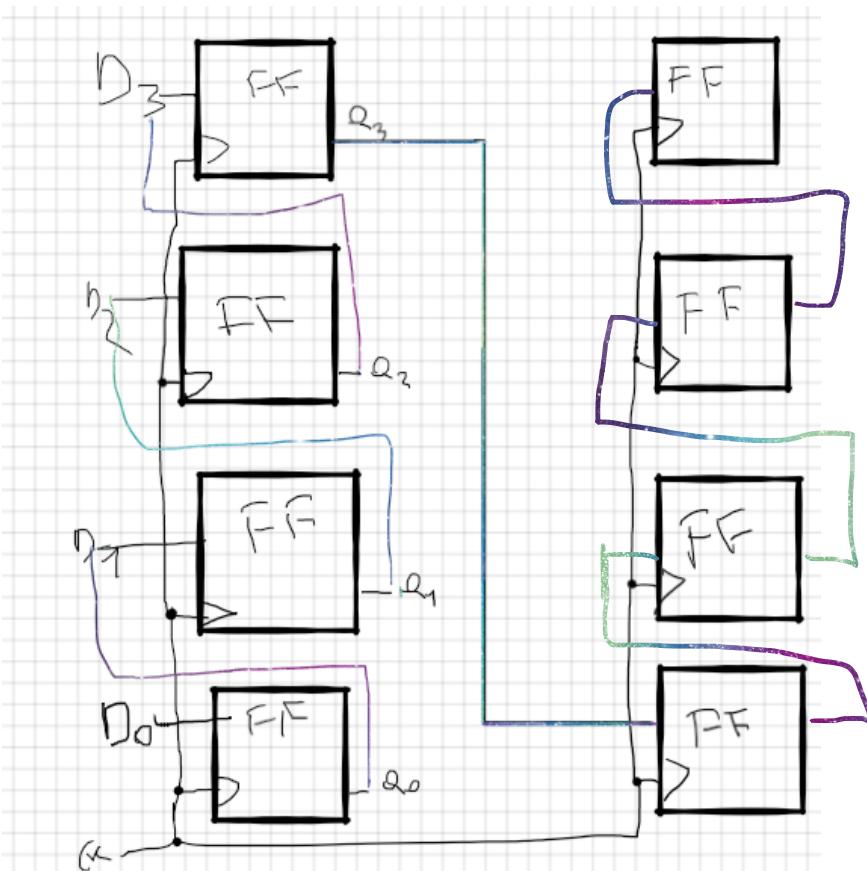


otterrei $N \rightarrow 2N/2$

Per questo tipo di calcolo si può usare anche un circuito combinatorio (come abbiamo visto nella spiegazione dell'ALU)

Come faccio a passare da un shift register a 4 bit a uno a 8 bit?

Innanzitutto so che devo usare altri 4 flip-flop



E dovrò collegare l'uscita dell'ultimo dei primi quattro flip-flop all'ingresso D del primo flip-flop tra i 4 aggiunti (e ovviamente questi altri 4 saranno collegati tra di loro come abbiamo già visto)

Adesso supponiamo di star lavorando solo su rappresentazioni a 4 bit (anche se su registro a scorrimento a 8 bit):

quindi abbiamo la metà a sinistra riempita dalla nostra rappresentazione; al primo impulso di clock avremo il bit più significativo che passa nel primo Flip-Flop della seconda metà (come abbiamo visto)

intuitivamente dopo quattro impulsi di clock avremo tutti 0 nei flip-flop a sinistra e la nostra rappresentazione iniziale nei flip-flop a destra

ciò significa che costruendo uno shift register a 8 bit e lavorando su rappresentazioni a 4 bit ho effettivamente costruito un dispositivo che mi permette di trasferire 4 bit di informazione da un "posto all'altro" (da un computer all'altro ad esempio) mediante l'utilizzo di 2 fili di collegamento (collegamento dell'uscita dell'ultimo dei primi quattro flip-flop all'ingresso D del primo flip-flop tra gli altri 4 e collegamento dei clock: sappiamo infatti che tutti gli ingressi clock devono "rispondere allo stesso")

quindi per fare questo trasferimento utilizziamo una quantità costante di fili (che non dipende dal numero di bit che devono essere trasferiti quindi, ovviamente il tempo che ci metterò a completare il trasferimento dipende eccone dal numero di bit: se ho 4 bit da una parte e li devo trasferire dall'altra dovrò aspettare 4 impulsi di clock fino a quando tutti e 4 i bit si trasferiranno, quindi in generale: attendo una quantità di cicli di clock pari al numero di bit da trasferire)

(se voglio trasferire da una parte all'altra un numero N di bit di informazione mi servirà un shift register a $2N$ bit)

quindi se voglio trasferire da una parte all'altra un numero N di bit di informazione mi servirà:

- Un registro a scorrimento a 64 bit (metà registro da una parte, metà dall'altra)
- 2 Fili di collegamento

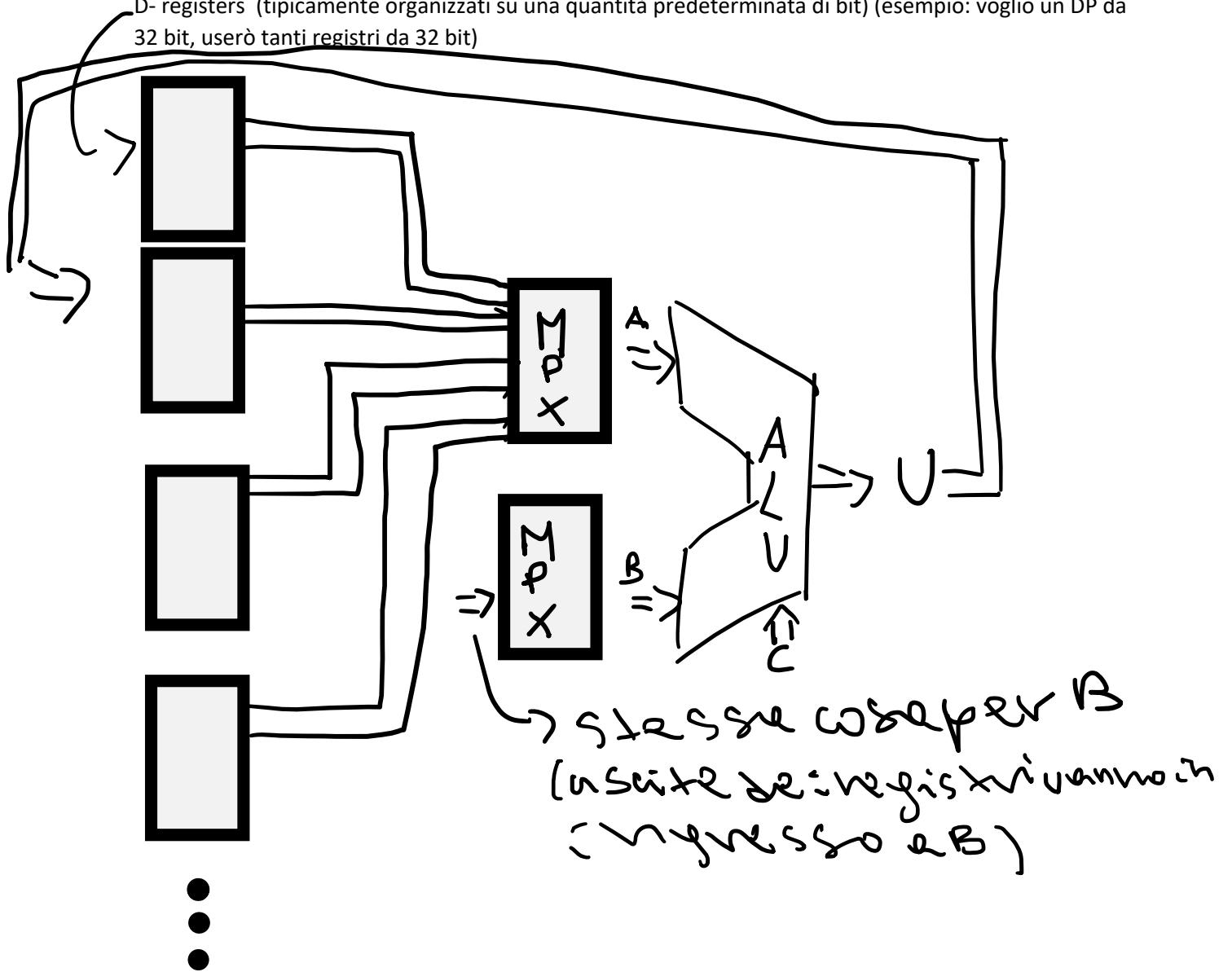
Carico il valore da trasferire nei primi 32 flip-flop durante 32 cicli di clock, altri 32 cicli serviranno per trasferire i bit dall'altra parte

(l'alternativa a questo sarebbe quella di dover passare ogni uscita singolarmente dall'altra parte, quindi nel caso 32 dovrò usare altri 32 fili di collegamento, ma far passare tanti fili da una parte all'altra è costoso, altro vantaggio di non usare così tanti fili è il fatto che più sono e più è possibile che qualcheduno si rompa: se attribuisco a ciascun filo una certa probabilità che esso si rompa, la probabilità che uno tra 2 fili si rompa è certamente minore piuttosto che uno tra 32 fili si rompa)

DATA PATH

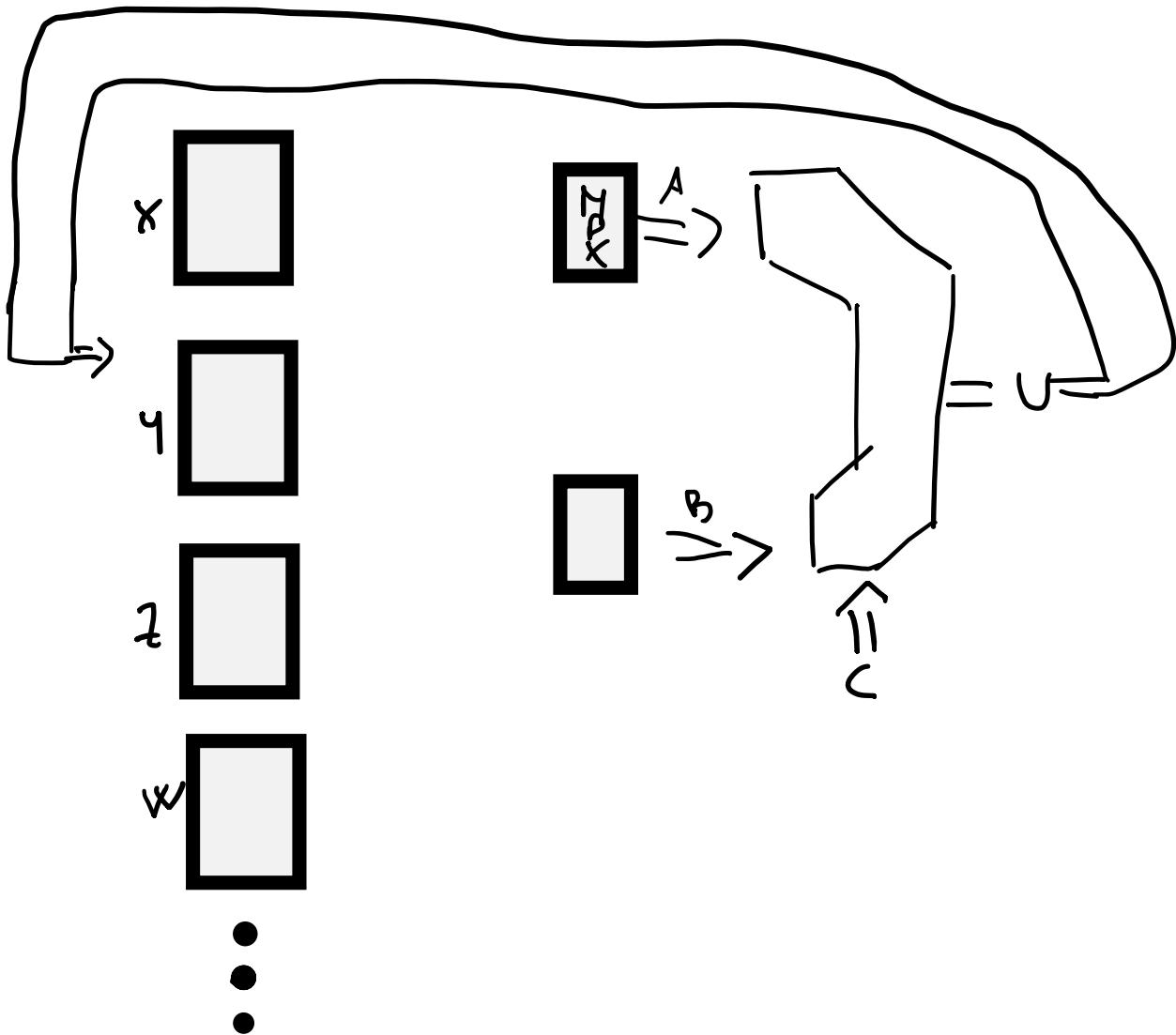
Un circuito sequenziale sincrono che è una delle componenti principali di un processore

D-registers (tipicamente organizzati su una quantità predeterminata di bit) (esempio: voglio un DP da 32 bit, userò tanti registri da 32 bit)



Immaginiamo di dover realizzare questo calcolo $w=x+y-z$

"assegniamo" ad ogni registro un "ruolo"



(ma dove va questa uscita? ogni volta devo poter decidere in quale registro memorizzare l'uscita U, e posso scegliere in quale attraverso un decoder per esempio)

Il data path sarà capace di eseguire una operazione alla volta, quindi bisogna aspettare un dato numero di cicli di clock per trovare il risultato)

Devo dividere queste operazioni in più fasi

Al primo ciclo di clock posso per esempio calcolare

$$w' = x + y$$

e poi al secondo ciclo di clock

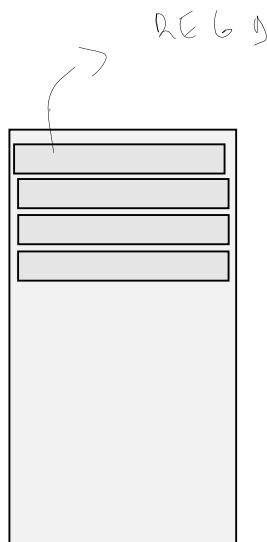
$$w = w' - z$$

oltre al data path per realizzare operazioni servirà anche un altro circuito sequenziale: il **control path**, che servirà a produrre i vari ingressi di controllo (del multiplexer e dell'ALU e il controllo per scegliere i vari registri in cui memorizzare valori) in base a cosa dobbiamo fare.

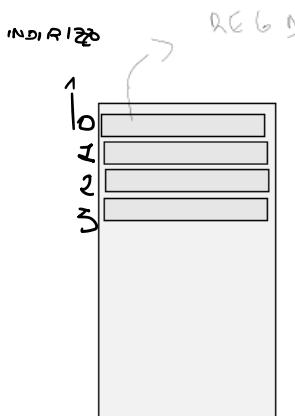
Altra componente fondamentale di un sistema di calcolo: l'unità di memoria.

Esistono diversi tipi di unità di memoria, cominciamo a vedere il più semplice: la RAM statica.

Dal punto di vista concettuale può essere vista come un insieme di tanti registri D su cui poter effettuare operazioni di lettura e scrittura (read/write).



Come identifichiamo la cella sulla quale vogliamo fare lettura scrittura? Basta “numerare” le celle di memoria, ogni cella ha il suo indirizzo:



possiamo pensare alla stessa sintassi che si usa in C/C++ per gli array, quindi di fatto pensare alla RAM come a un array:

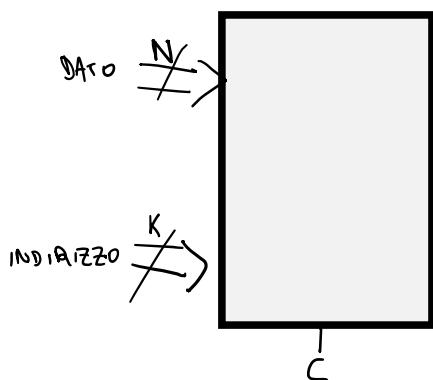
RAM[IND]

Useremo sempre un dispositivo che abbia un numero di celle pari a una potenza di 2, se quindi partiamo dalla cella numero 0, l'ultima cella avrà indirizzo $2^k - 1$, dove k è il numero di bit che sto usando per rappresentare gli indirizzi.

Devo definire anche il numero di bit di informazione contenuto in ciascun registro.

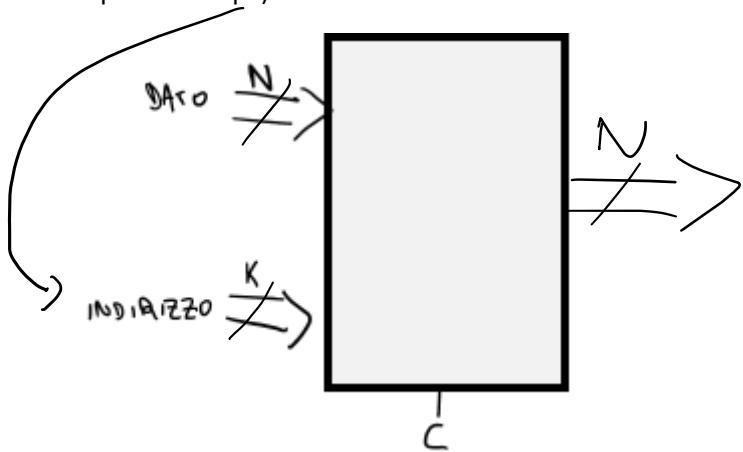
(quindi per poter effettuare una operazione di scrittura abbiamo bisogno di specificare, oltre all'indirizzo della cella in cui vogliamo fare write, anche il contenuto che vogliamo scriverci dentro: la configurazione binaria che vogliamo inserire in tale cella).

Servirà un segale di controllo per distinguere tra le due operazioni (scrittura e lettura)



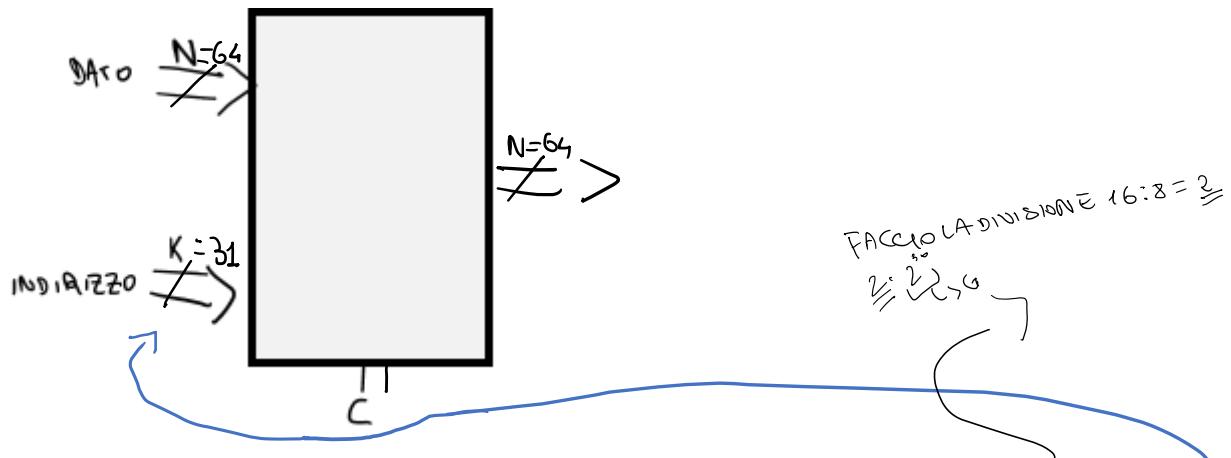
(i bit che mi serviranno per il controllo sono 2, perché non devo codificare due operazioni, ma 3: la scrittura, la lettura e il "non fare niente", perché ci possono essere momenti in cui non voglio né scrivere né leggere da una cella).

Per l'operazione di lettura ci serve, oltre l'indirizzo della cella interessata (che abbiamo già nel disegno), un output: leggere significa mandare in uscita il contenuto di una certa cella di memoria (il cui indirizzo è specificato qui)



Immaginiamo quale potrebbe essere la quantità di fili da usare per connettere una RAM statica al resto del sistema. N tipicamente corrisponde alla dimensione dei registri del processore: quando usiamo un processore a 32 bit useremo una memoria RAM con celle di memoria a 32 bit, quando usiamo un processore a 64 bit useremo una RAM organizzata in celle di memoria da 64 bit.

Proviamo a vedere che succede con N=64



sappiamo che 64 bit = 8 byte, quindi ogni cella di memoria (ogni registro) è da 8 byte

RICORDIAMO CHE 1 GB = 2³⁰ byte (PARLAMO DI Gibibyte)

Immaginando di avere una RAM da 16GiB, questa dovrà avere circa 2 miliardi di celle di memoria ($2 * 2^{30} = 2.147.483.648$), $2^{31}-1$ è quindi l'indirizzo dell'ultima cella di memoria (sappiamo quindi che K è 31).

Contiamo i fili di connessione: $64 + 64 + 31 + 2 = 161$ fili; sono troppi, come riduciamo i fili?

E se utilizzassimo una organizzazione a 32 bit?

$$\begin{aligned} &\text{FACCIO LA DIVISIONE } 16:4 = 4 \\ &4 \cdot 2^{30} = 2^{11} - 2 \end{aligned}$$

N=32. Quindi ogni cella di memoria ha 4 bit. Se immaginiamo di avere sempre una RAM da 16GiB, questa avrà circa 4 miliardi di celle di memoria ($2^{32}=4.294.967.296$), $2^{32} - 1$ è quindi l'indirizzo dell'ultima cella di memoria,

Con N=32 K sarà 32. Quindi abbiamo $32 + 32 + 32 + 2 = 98$ fili

(quindi passando semplicemente da 64 bit a 32 siamo passati da 161 fili di collegamento a 98)

Questo discorso si può continuare continuando a diminuire il numero di bit su cui è organizzata la memoria:

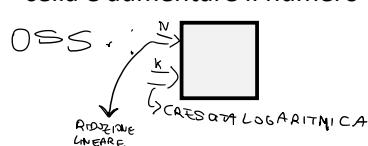
rimaniamo sempre su una RAM di 16GiB, con celle di memoria a 8 bit (quindi 1 byte)

$$\begin{aligned} &\text{FACCIO LA DIVISIONE } 16:1 = 16 \\ &16 \cdot 2^{30} = 2^4 \cdot 2^{30} = 2^{34} \end{aligned}$$

$2^{34} - 1$ è l'indirizzo dell'ultima cella di memoria (k è 34), quindi ho $8 + 8 + 34 + 2 = 52$ fili

Ovviamente questa diminuzione ha anche degli svantaggi: nello stesso numero di cicli di clock con la organizzazione a 64 bit posso leggere/scrivere una parola da 64 bit, ad esempio, mentre con una organizzazione a 8 bit posso leggere/scrivere solo 8 bit; quindi riducendo il numero di bit su cui è organizzata la memoria riduco la quantità di fili richiesti per connettere il dispositivo al sistema (e quindi il costo), ma anche la velocità di trasferimento di dati.

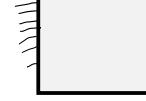
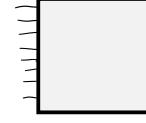
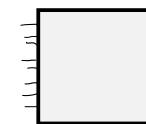
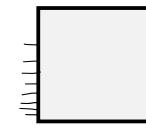
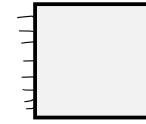
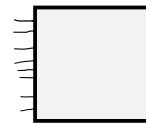
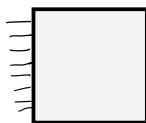
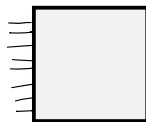
Quindi la soluzione più conveniente per chi produce tali dispositivi è ridurre la dimensione della singola cella e aumentare il numero delle celle



Se usassimo l'approccio a 8 bit, con quindi 52 fili di interconnessione

PA ROME DA 1 BYTE (8bit)
 $\approx 2^k$ parole

e volessimo realizzare una RAM da 64 bit per ogni parola? Potrei unire 8 di questi dispositivi da 8 bit quindi di fatto “suddivido” la realizzazione a 64 bit in 8 dispositivi diversi:



Come potremmo realizzare internamente il dispositivo?

Ci servono, come abbiamo detto, dei registri di tipo D

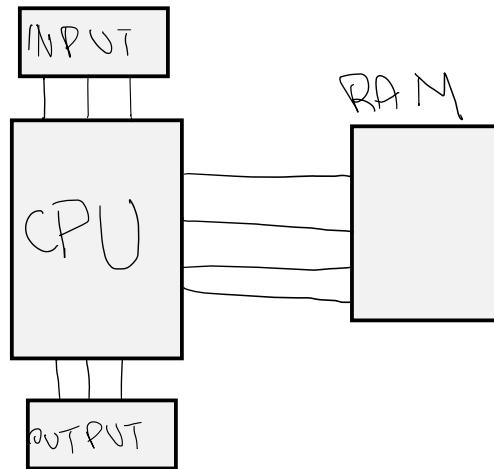
Cos'altro ci serve? Un multiplexer, così da poter scegliere una qualunque delle celle di memoria da connettere sull'uscita (per fare l'operazione di lettura). Questo multiplexer avrà quindi 2^k ingressi (così che si possa scegliere tra tutte le celle di memoria); i segnali di controllo del multiplexer saranno i k fili di indirizzamento.

Per l'operazione di scrittura ci serve un decoder per scegliere quale delle celle di memoria sottoporre all'operazione di scrittura, anche in questo caso i k fili di indirizzamento saranno i segnali di controllo del decoder (si potrebbe anche usare un demultiplexer, che porta il segnale di clock a una sola di queste celle di memoria).

Ovviamente non dovrò "replicare" il decoder per l'operazione di scrittura e quella di lettura perché, come abbiamo già visto a inizio corso, il multiplexer e il demultiplexer hanno una parte in comune (chiamata appunto decoder): basterà usare un solo decoder (che servirà e per l'implementazione del demultiplexer, nel caso dell'operazione di scrittura e per l'implementazione del multiplexer nel caso dell'operazione di lettura).

Forma più semplice possibile per organizzare un sistema di calcolo:

architettura von Neumann

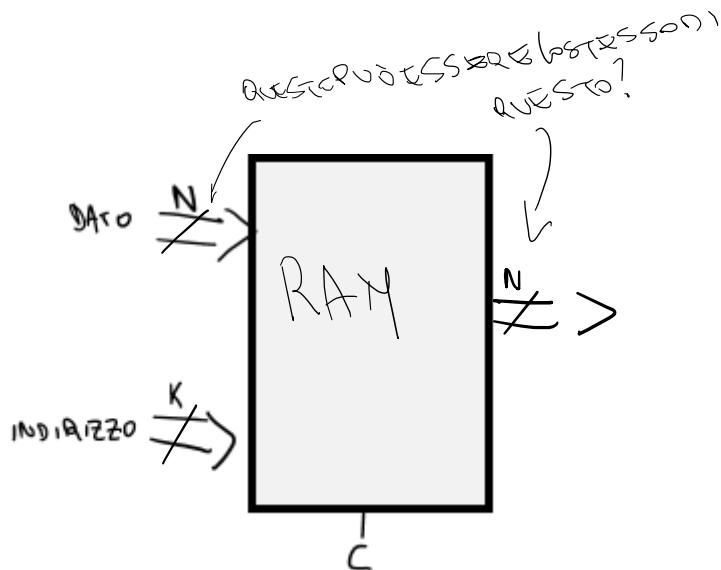


(cpu: central processing unit)

Come comunicano CPU e RAM tra di loro?

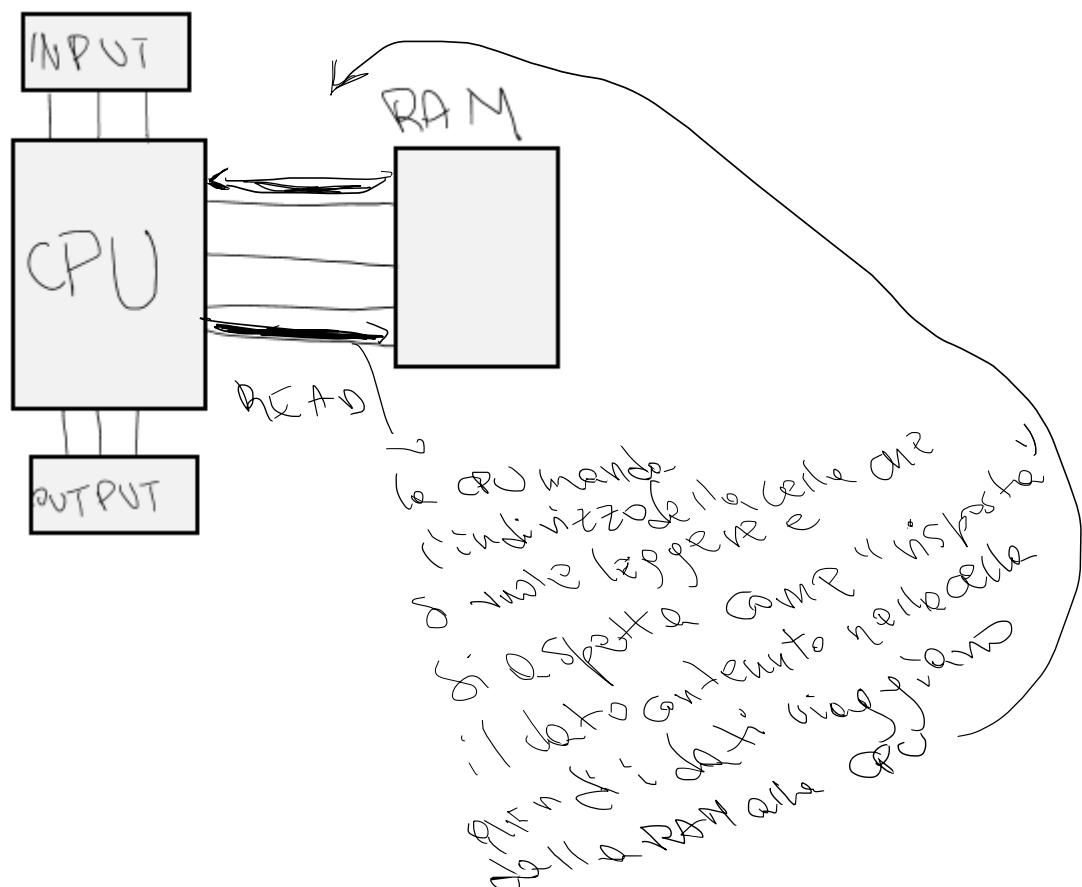
Tramite un protocollo definito master-slave (master: CPU, comanda; slave: RAM, esegue gli ordini)

Per quanto riguarda i discorsi dei fili della RAM: si possono "unificare" fili di input e output? Cioè si può usare lo stesso filo sia per input che per output?



Normalmente no, non si può usare lo stesso filo durante le operazioni di lettura e scrittura per far passare i dati (anche se ci permetterebbe di risparmiare notevolmente sulla quantità di fili di interconnessione), perché dovremmo gestire un flusso bidirezionale:

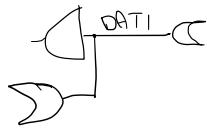
immaginiamo di dover fare una operazione di lettura



Ciò significa che all'interno della CPU ci deve essere "qualcosa" che prende in ingresso quel filo di dati (un circuito combinatorio, eccetera), per esempio una funzione AND (e nella RAM ovviamente ci deve essere una uscita di una funzione per portare fuori questi dati)

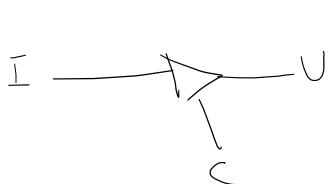


Se vogliamo realizzare l'operazione di scrittura il flusso di dati sarà invertito, ma se io uso lo stesso filo allora nella CPU ci dovrà essere l'uscita di una qualche funzione e ciò non va bene: non possiamo avere dal punto di vista combinatorio due uscite di funzioni diverse collegate allo stesso filo...quale uscita dovrei prendere in considerazione se esse producono valori diversi? (è quindi una situazione da evitare)



Si possono però utilizzare dispositivi a 3 stati

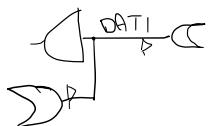
(se l'ingresso di controllo assume valore 1, in uscita avrò ciò che c'è sull'ingresso I, se invece l'ingresso di controllo assume valore 0 l'uscita non viene collegata)



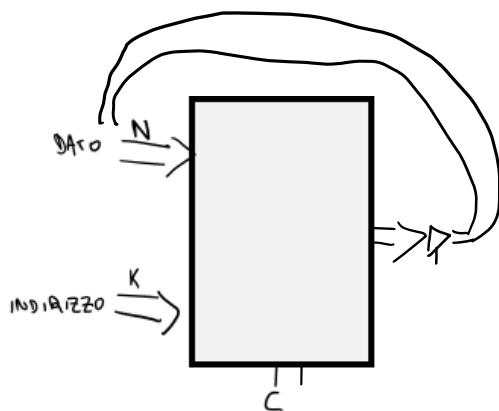
C	U
0	NON COLLEG.
1	I

Un dispositivo a 3 stati ci può permettere di risolvere il problema precedente:

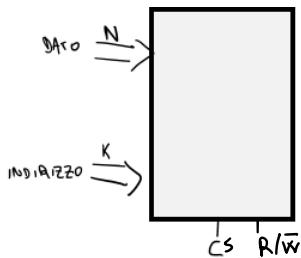
aggiungiamo un dispositivo a 3 stati per ogni uscita che vogliamo collegare al filo



(quindi possiamo condividere i fili di dati sia per la scrittura che la lettura se usiamo i dispositivi a 3 stati avviando gli ingressi di controllo in maniera opportuna) (così facendo dimezziamo i fili di dati)



Come sono organizzati i fili di controllo nella RAM statica?

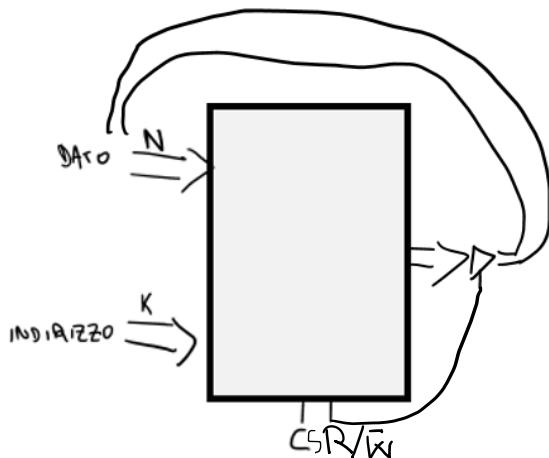


CS: chip select: quando vale 0 non si fa niente sulla RAM, quando vale 1 attiva la RAM per fare o una operazione di scrittura o una di lettura

Quale delle due operazioni? Guardiamo l'altro filo:

R/ \bar{W} : quando vale 1 faccio lettura, quando vale 0 faccio scrittura

Il filo R/ \bar{W} lo posso usare direttamente come filo di controllo dei dispositivi a tre stati



(dentro alla CPU ci saranno altri dispositivi a 3 stati che sono controllati dalla negazione di questo filo R/ \bar{W} , di modo che l'uscita della CPU sui fili di dati sia connessa solo durante l'operazione di scrittura)

Abbiamo ridotto il numero di fili di interconnessione usando i dispositivi a tre stati che ci permettono di utilizzare gli stessi fili per la comunicazione in entrambe le direzioni: in momenti diversi i dati scorrono dalla CPU alla RAM oppure dalla RAM alla CPU. Una volta ridotto al minimo il numero di fili, ci chiediamo quanto grandi possono essere le RAM che realizziamo:

sappiamo che ogni cella è un registro di tipo D (insieme di più D flip-flop in configurazione Master-Slave), quindi ogni bit che appartiene a una cella di memoria è realizzato attraverso un flip-flop Master-Slave

i flip-flop di questo tipo, come abbiamo visto contengono 9 funzioni logiche elementari, ogni funzione logica ha delle componenti fisiche elementari (transistor, eccetera), mettiamo caso che ogni funzione tra quelle usate nel Master-Slave richieda 5 transistor, moltiplichiamo per 9 e otteniamo 45 (quindi ogni bit di informazione della RAM richiede decine di componenti fisiche per la sua realizzazione?)

si arriva allora ad un limite oltre il quale non è più conveniente realizzare il dispositivo: quando la probabilità che esso contenga dei difetti diventa troppo alta, e questa probabilità ovviamente cresce con l'aumentare del numero di componenti elementari del dispositivo.

Le memorie ram statiche non possono avere delle dimensioni ragionevoli (nel 2022): non possiamo ad esempio realizzare delle RAM da 16 giga con la tecnologia statica.

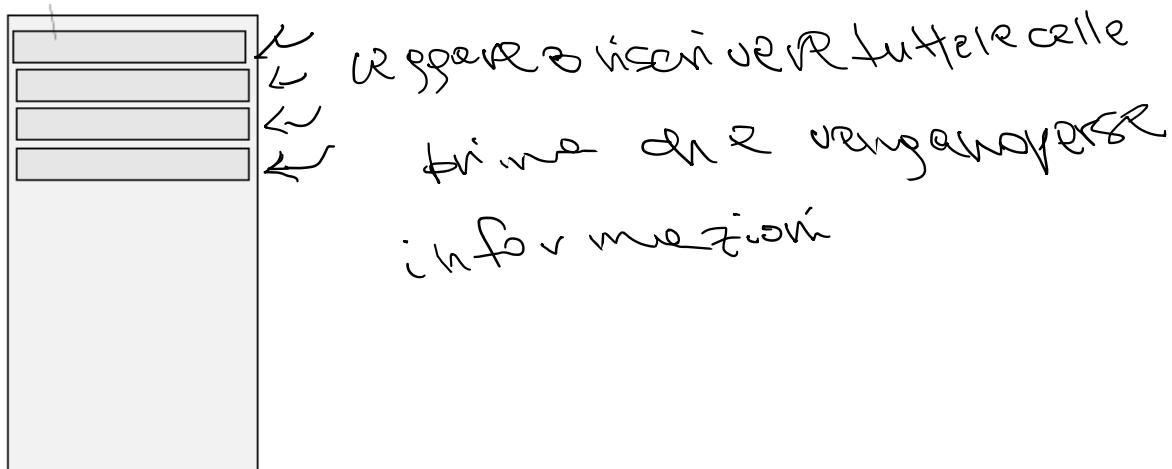
Si passa alla tecnologia dinamica.

Qui la situazione è diversa: un singolo bit viene memorizzato utilizzando un solo componente elementare (a differenza delle decine di componenti che servivano per memorizzare un solo bit nella statica). C'è uno svantaggio però: a differenza delle RAM statiche, quelle dinamiche non sono in grado di ricordare il valore memorizzato se non per una molto breve quantità di tempo (si tratta di qualche millisecondo), bisogna tener conto di questa caratteristica per far funzionare una memoria dinamica:

la tecnica è quella di rileggere il valore che è stato appena memorizzato prima che venga "dimenticato" e successivamente riscriverlo tale e quale:

questa operazione è il refresh: periodicamente tutte le celle di memoria vengono lette e riscritte con lo stesso valore che c'era prima (con questa "riscrittura" quindi facciamo ripartire il conteggio del tempo, prima che i valori vengano dimenticati nuovamente):

ciò che è fondamentale quindi è riuscire a completare tutto il ciclo di lettura e riscrittura prima che avvenga la perdita di informazioni:



Ci sarà quindi una complessità circuitale non indifferente (serviranno dispositivi che permettano di fare tale operazione nell'adeguato lasso di tempo), ma d'altro canto si ha risparmiato sulla complessità di realizzazione dei singoli bit (rispetto alla memoria statica)

Altro svantaggio delle dinamiche: rapporto tra refresh e write/read; viene abbastanza naturale capire che mentre sto facendo un refresh non posso fare una normale operazione di lettura o scrittura,

quindi se vorrò fare una operazione di scrittura/lettura dovrò aspettare che il mio dispositivo non stia facendo refresh (perché come detto le due operazioni non possono essere simultanee).

Quindi le memorie dinamiche sono più lente di quelle statiche (le operazioni di scrittura e lettura sono ritardate a causa della presenza del circuito di refresh: il dispositivo in sé è veloce ma impiega una parte del tempo a fare l'operazione di refresh, quindi se ad esempio viene richiesta una operazione di lettura in un certo istante ma in quell'istante si sta refreshando, non si potrà iniziare immediatamente l'operazione di lettura).

Nelle ram dinamiche quindi, oltre ai registri (le celle di memoria) ci sarà un'altra componente che si occupa di realizzare l'operazione di refresh.

Altro dispositivo di memoria:

memoria associativa(in inglese Content Addressable Memory: CAM)

Una singola cella di memoria associativa è composta da 2 registri D:

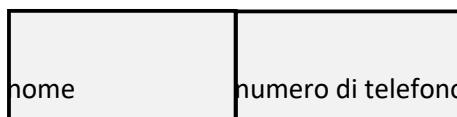


TAG DATA

L'idea delle memorie associative è quella di individuare una data cella di memoria non sulla base della sua posizione (come avviene nella ram, con gli indirizzi di memoria), ma sulla base del contenuto della TAG

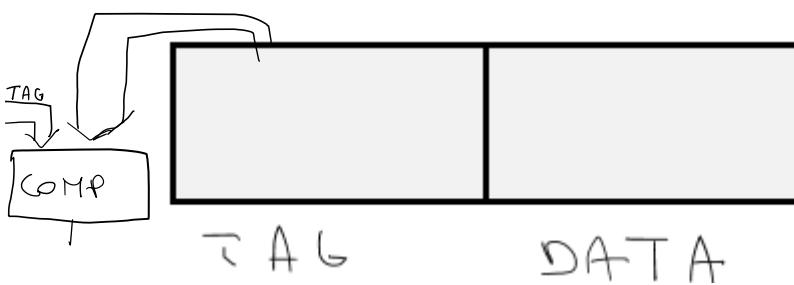
La memoria associativa serve a realizzare meccanismi di ricerca ad alta velocità: associando due valori, se ne conosco uno posso risalire all'altro, esempio:

rubrica telefonica:



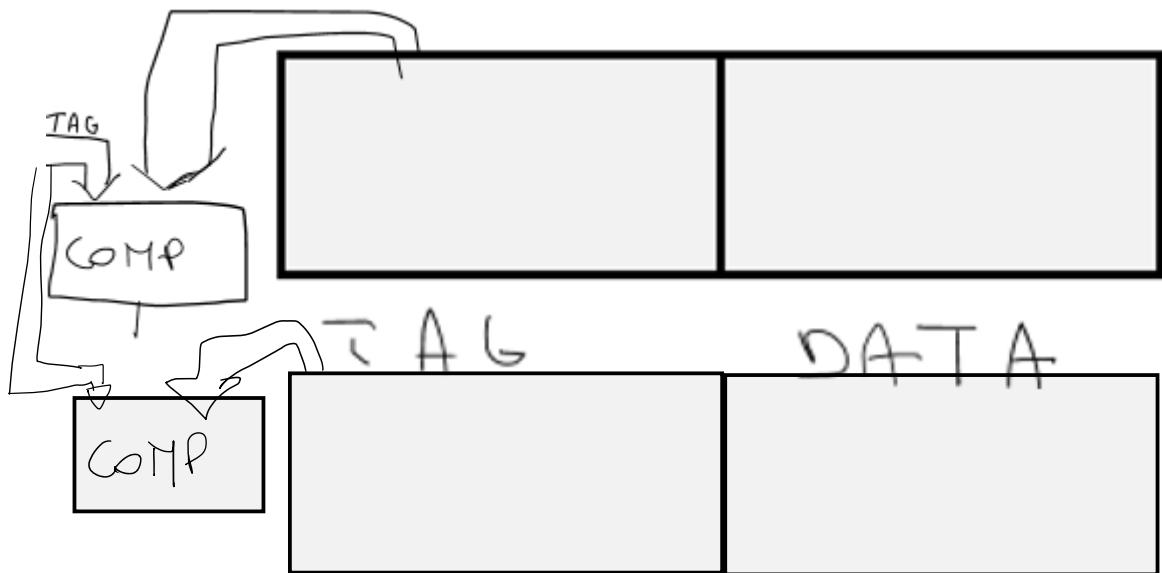
l'obiettivo è costruire un dispositivo che abbia la stessa velocità sia con poche che con tante celle (la velocità di accesso deve essere indipendente dal numero di celle contenute)

per fare ciò aumentiamo la complessità hardware: ad ogni registro TAG associo un dispositivo in grado di fare il confronto con un valore proveniente dall'esterno: tipicamente un circuito comparatore (dispositivo che fa il confronto tra una rappresentazione binaria ed un'altra rappresentazione binaria sullo stesso numero di bit e mi dà in uscita il valore 1 quando i due ingressi sono uguali, 0 quando sono diversi)



Quindi per fare una ricerca devo mandare in ingresso un certo TAG che verrà confrontato con il TAG della cella di memoria, e in uscita avrò il risultato che mi dice se i due ingressi sono uguali o diversi.

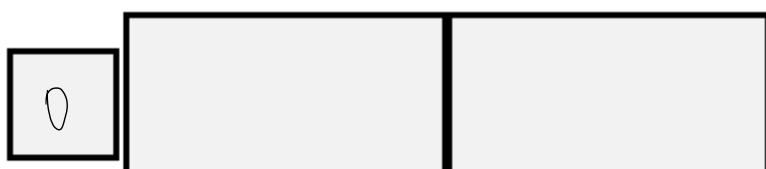
Se voglio fare una ricerca su 2 celle di memoria devo duplicare il comparatore (se voglio farla su 1000 celle di memoria serviranno 1000 comparatori, eccetera)



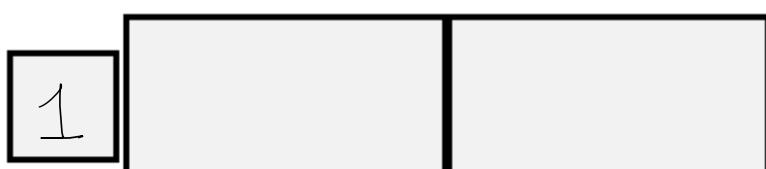
Dobbiamo tenere conto dell'eventualità che 1 o più celle siano vuote: aggiungo quindi un ulteriore bit che mi darà questa informazione



Come si fa ad introdurre un nuovo valore in una cella di memoria associativa? Prendiamo una cella vuota



La "facciamo diventare" piena: scriviamo 1 nel bit di presenza delle informazioni



E dentro un registro metteremo il TAG di ricerca e dentro l'altro il dato associato.

Operazione di lettura: mando un tag che mi interessa e vedo se almeno uno dei comparatori mi dà risultato 1, guardo la cella/e di memoria che ha il TAG uguale a quello "richiesto" e ne leggo il contenuto in DATA.

Ci può essere 1 solo dato associato ad ogni tag; se voglio inserire dei dati in una situazione in cui la memoria non è completamente vuota devo fare prima l'operazione di ricerca del tag: se richiedo un tag che poi vedo essere identificativo di una cella che ha già un dato, andrò a sovrascrivere quel dato (se invece nella ricerca ho esito negativo significa che non c'è nessuna cella con quel tag, a questo punto prendo una cella libera, la "rendo occupata" e scrivo TAG e dato corrispondente), quindi tecnicamente per fare una operazione di scrittura è richiesta una operazione di lettura (la ricerca del TAG).

Sono sempre realizzate con tecnologia statica (così che venga garantita la massima velocità possibile) e quindi per questo sono limitate a livello di dimensioni.

Confronto CAM vs RAM (statica)

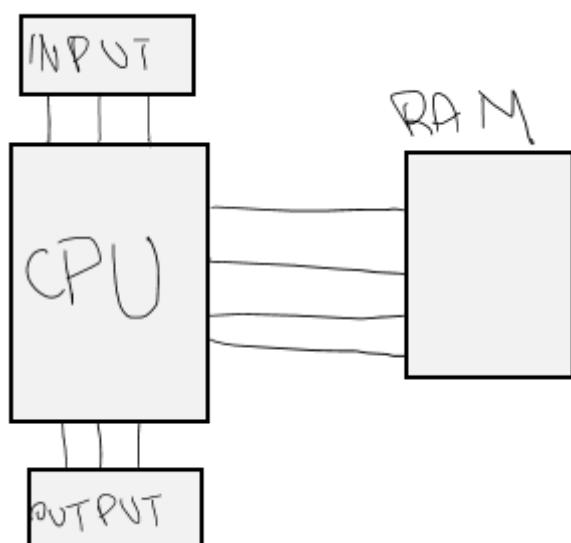
Se pensiamo a una configurazione dei dati su 32 bit per esempio, sappiamo che sarà più grande la CAM (perché nella RAM ogni cella di memoria è realizzata da 1 registro di tipo D, e quindi per coprire quei 32 bit ci vogliono un tot di registri D, nella CAM una cella è realizzata da 2 registri D, quindi ce ne vogliono il doppio, se ipotizziamo che anche i TAG siano da 32 bit e io possa realizzare al massimo memoria da 100 MebiByte vuol dire che a parità di dimensione potrò realizzare per la memoria associativa al massimo la metà delle celle di memoria che posso realizzare per la RAM: la capacità della RAM è maggiore)

(vediamo subito come la CAM sia più complicata della RAM)

Altre complicazioni: i circuiti comparatori, realizzare la logica per le operazioni di scrittura (fare la lettura prima per vedere se c'è il tag, prendere la cella libera, occuparla) tutte cose che richiedono l'aggiunta di ulteriori dispositivi

Differenza fondamentale tra CAM e RAM è quella di poter fare una ricerca in tempo costante (indipendente dalla dimensione del dispositivo); se volessi simulare una CAM avendo a disposizione solo RAM per esempio su 32 bit 100 MiB avrei bisogno di una RAM di dimensioni DOPPIE rispetto alla classica RAM da 32 bit 100 MiB (perché dovrei mettere sia TAG che data nella RAM e quindi occuperei il doppio dello spazio per immagazzinare un certo numero di dati).

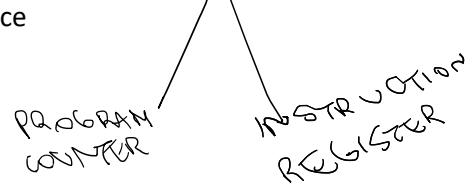
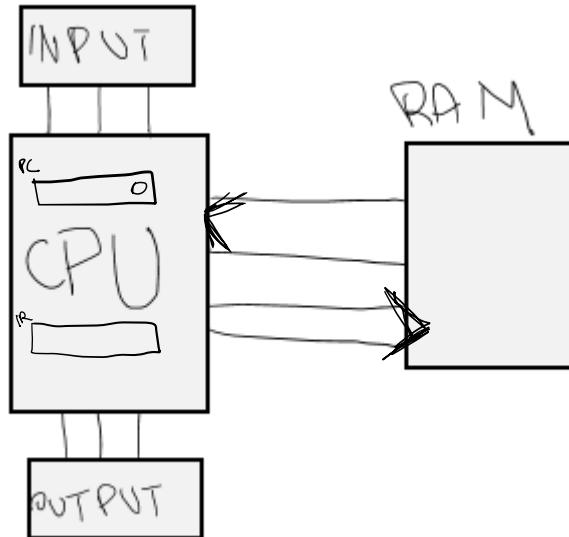
Riprendiamo l'idea dell'architettura Von Neumann



Principale differenza tra architettura di Von Neumann e quelle più vecchie: sia la rappresentazione binaria dei dati che quella del codice da eseguire vanno a finire in un unico dispositivo di memoria (la RAM).

Altra particolarità della Von Neumann: inserire il codice in modo ordinato all'interno della RAM, se ad esempio ho 10 istruzioni da eseguire userò 10 celle della memoria RAM (la prima istruzione andrà alla cella con indirizzo 0, la seconda a quella con indirizzo 1 e così via) (si chiama esecuzione sequenziale)

Come abbiamo già visto, dentro alla CPU c'è il Data Path, dentro ci saranno anche 2 registri, che ci serviranno per realizzare l'idea dell'esecuzione sequenziale del codice



Useremo il PC per individuare un indirizzo sul quale effettuare una operazione di lettura dalla RAM, useremo IR per riportarvi dentro il contenuto della cella della RAM a seguito dell'operazione di lettura.

Dobbiamo fare iniziare l'esecuzione sequenziale dei nostri programmi dall'indirizzo 0, quindi quando accendo la macchina voglio che dentro al registro PC ci sia 0.

$D \quad PC \leftarrow 0$

Se voglio realizzare una operazione di lettura come già detto dovrò mandare i segnali di controllo $CS=1$, $R/W=1$, inoltre dovrò usare come indirizzo per la RAM il contenuto del Program Counter, aspetto un po' di tempo e avrò in uscita dalla RAM il valore contenuto all'interno della cella di indirizzo 0, contenuto che inserirò all'interno del IR (registro delle istruzioni)

$1 \quad IR \leftarrow RAM[PC]; \quad PC \leftarrow PC + 1 \quad \text{FETCH}$

Dopo aver completato questa operazione voglio incrementare di 1 il contenuto del registro PC (se ho una frequenza di clock che me lo permette, se la RAM è abbastanza veloce, posso pensare di fare le 2 operazioni appena descritte nello stesso ciclo di clock)

Questa prima operazione si chiama fase di fetch.

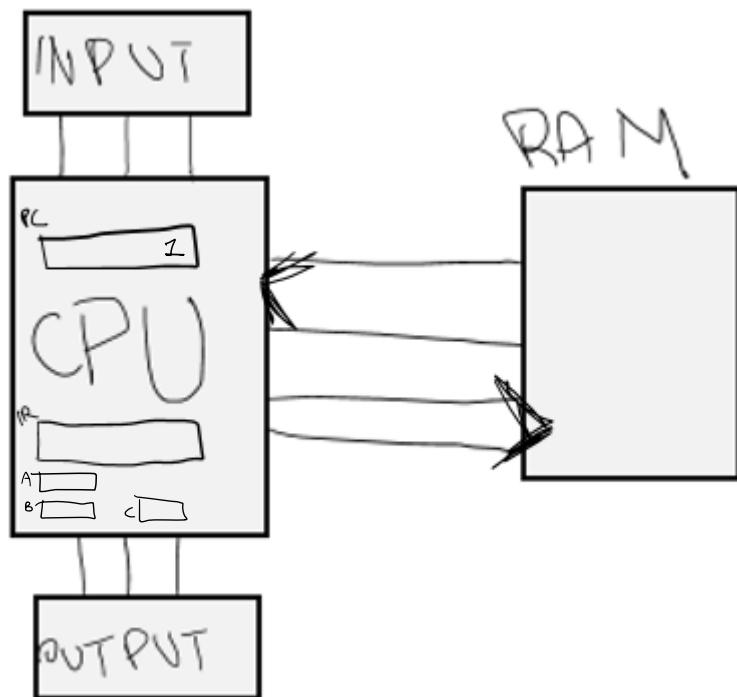
Dopo un ciclo di Clock quindi avrò su 1 nel PC e dentro a IR una copia del valore che era contenuto all'interno della prima cella di memoria RAM (cella a indirizzo 0).

Abbiamo detto che nella RAM ci possono essere sia dati che codice; quando una codifica binaria va a finire nel registro delle istruzioni sappiamo che è la codifica di una istruzione che deve essere eseguita, non può essere un dato.

Ora dovrei prendere il contenuto dell' IR (la mia prima istruzione) e decodificarlo

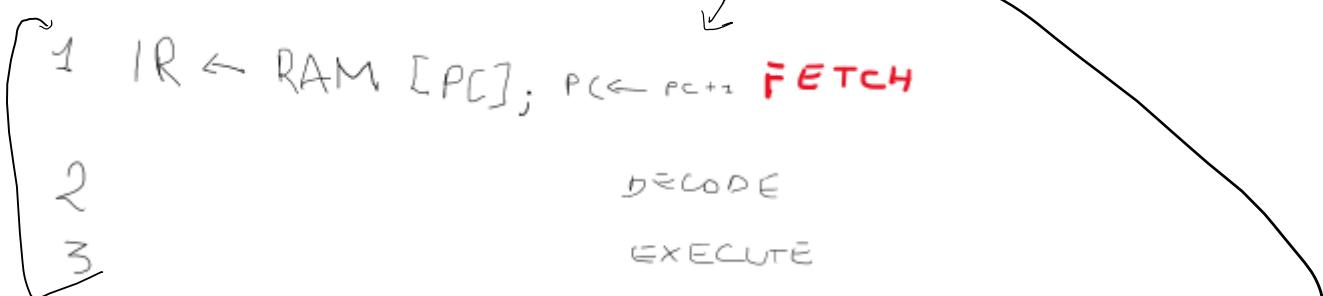


Supponiamo di avere dentro al processore, oltre a IR e PC, altri 3 registri (che chiamiamo A, B e C)



E supponiamo di aver capito, decodificando, che si richiede di fare l'operazione $C = (A + B)$, alla fase 3 quindi eseguiremo tale istruzione: prendo il contenuto di A e lo mando su un ingresso dell'ALU, prendo il contenuto di B e lo mando su un altro ingresso dell'ALU, "dicendo" all'ALU di fare la somma (attraverso controlli) e metto l'output dell'ALU in input al mio registro C, do il colpo di clock e memorizzo il risultato dentro al registro C.

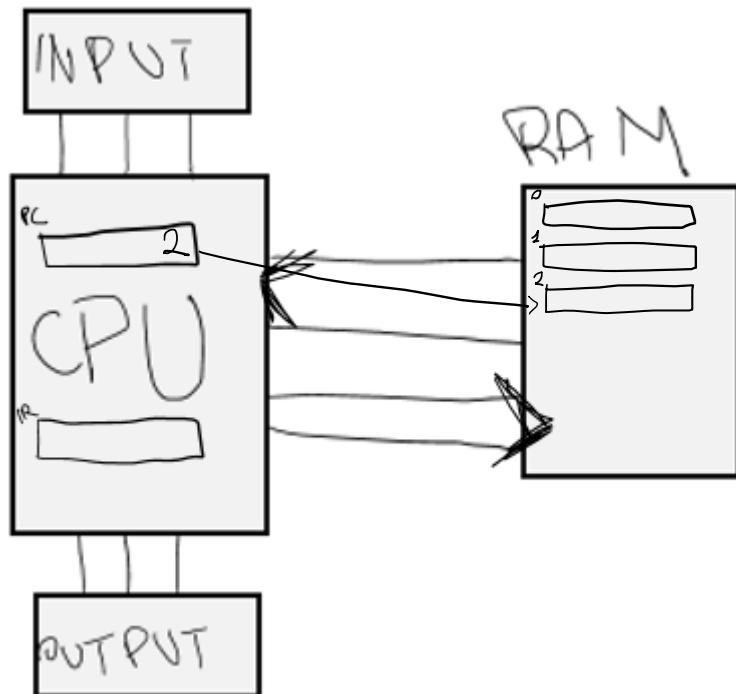
Terminata la fase di esecuzione si ritorna al fetch



Questa volta dentro PC ho 1, e quindi andrò a guardare la cella della RAM all'indirizzo 1, prenderò il suo contenuto e lo inserirò nell'IR, in seguito "incrementerò" PC

Si passerà alla decodifica e all'esecuzione, poi di nuovo al fetch, avrà dentro al PC il valore 2, così guarderà alla cella a indirizzo 2, e così via.

(stiamo eseguendo in modo sequenziale il codice)



Quindi come abbiamo visto avremo bisogno di un ciclo di clock all'inizio per far partire la macchina, più 3 cicli di clock per ogni istruzione (fetch, decode ed execute ognuno un ciclo).

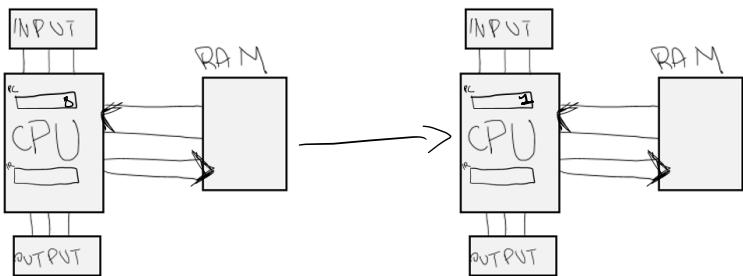
Abbiamo specificato all'inizio che ogni istruzione occuperà una cella di memoria, quindi se dovessi eseguire un programma con mille istruzione dovrei avere 1000 celle di memoria RAM; non è molto comodo come sistema, è limitante (facciamo un'analogia con i linguaggi di programmazione: sarebbe assurdo avere solo parti di esecuzione sequenziale, ad esempio 1° istruzione; 2° istruzione; n° istruzione. Ci sono anche i cicli, le istruzioni condizionali, eccetera, a livello hardware si possono realizzare queste modifiche modificando il contenuto del Program Counter, se ad esempio mi "stufassi" di continuare ad eseguire istruzioni ad indirizzi successivi potrei specificare una istruzione che assegna un valore diverso al registro PC; le istruzioni che modificano il contenuto del registro PC vengono chiamate istruzioni di salto (jump/branch))



Come posso realizzarne una? Codifico come una delle istruzioni possibili una istruzione che inserisca un valore completamente diverso all'interno di PC, ad esempio

$PC \leftarrow 1$

Supponiamo di trovare questa istruzione alla cella a indirizzo 7 della RAM, se ho completato il fetch avrà 8 in PC (devo fare $PC \leftarrow PC + 1$), valore che viene sostituito dal valore determinato dall'istruzione di salto, in questo caso 1



E quindi se ora il PC vale 1 dovrò prendere in considerazione la cella di memoria a indirizzo 1 (quindi di fatto fetchare, decodificare ed eseguire l'istruzione che sta nella cella a indirizzo 1 della RAM), quindi una volta che ho 8 nel PC e metto una istruzione di salto del tipo PC=1, la sequenza di istruzioni da 1 a 7 potrà essere ripetuta un numero arbitrario di volte.

Altro esempio di istruzione di salto:

$PC \leftarrow A$

Prende il contenuto di un altro registro (A) e lo inserisce all'interno del PC

E il contenuto di questo registro A potrebbe essere il risultato di un calcolo precedente, ad esempio

$A \leftarrow B \times 1$

Queste istruzioni possono essere chiamate quindi "automodificanti", potremmo avere un programma che calcola qualcosa, e sulla base delle cose calcolate in precedenza modifica il suo comportamento futuro.

Quindi è estremamente potente perché ci permette di eseguire operazioni complicate usando poche istruzioni macchina, ma è anche rischioso perché le istruzioni possono dipendere anche dai valori che sta calcolando il programma, anche dagli input del programma (e quindi sbagliando a calcolare si possono far fare al programma operazioni che non dovrebbe mai fare, eseguendo pezzi di codice che non dovrebbe eseguire, ad esempio); come faccio a vedere quali pezzi di codice vengono eseguiti e quali no? Simulando il comportamento della macchina passo passo, eseguendo le fasi di fetch, decodifica ed esecuzione.

Diventa molto complicato farsi un'idea della eventuale completezza/correttezza/presenza di errori di un certo programma con decine di istruzioni (simulando il comportamento del programma a livello "logico" possiamo renderci conto della presenza di errori in programmi piccoli, o comunque non abbiamo la stessa velocità fisica di calcolo di un computer e quindi potremmo simulare solo un certo numero di operazioni al minuto, mentre un elaboratore ne processa anche centinaia al secondo)