Matteo Di Gioacchino
Soluzioni Esercitazione 16/01/2025

PER QUALSIASI DUBBIO, PROBLEMA O SE PENSATE DI AVER TROVATO UN ERRORE NELLE SOLUZIONI, SCRIVETEMI QUANDO VOLETE

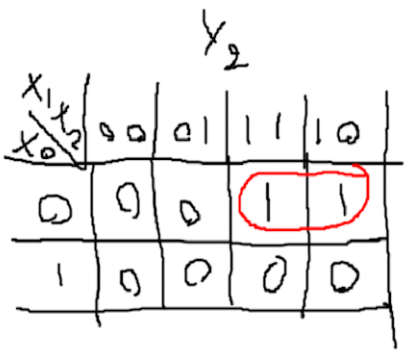
Nota: ne approfitto per aggiungere due piccole correzioni dell'Es. 9 dell'Esercitazione del 16/12/2024, le troverete alla fine di questa.

1)

1o modo:

x_2	x_1	x_0	y_5	y_4	y_3	y_2	y_1	y_0
0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1
0	1	0	0	0	0	1	0	0
0	1	1	0	0	1	0	0	1
1	0	0	0	1	0	0	0	0
1	0	1	0	1	1	0	0	1
1	1	0	1	0	0	1	0	0
1	1	1	1	1	0	0	0	1

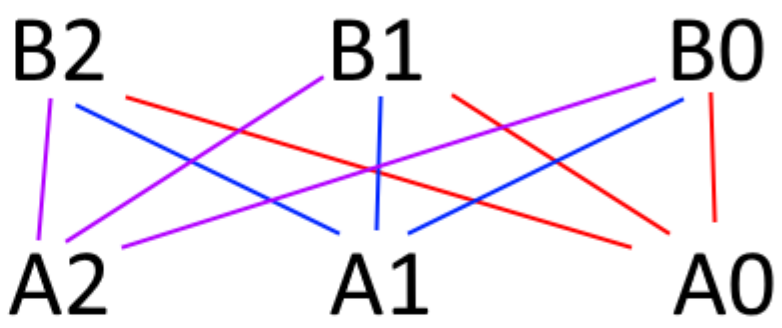
Oss: nel peggiore dei casi dobbiamo elevare alla seconda il numero (111)_2 = (7)_10, il quale, elevato alla 2a, è uguale a 49, per rappresentare 49, ci servono 6 bit.

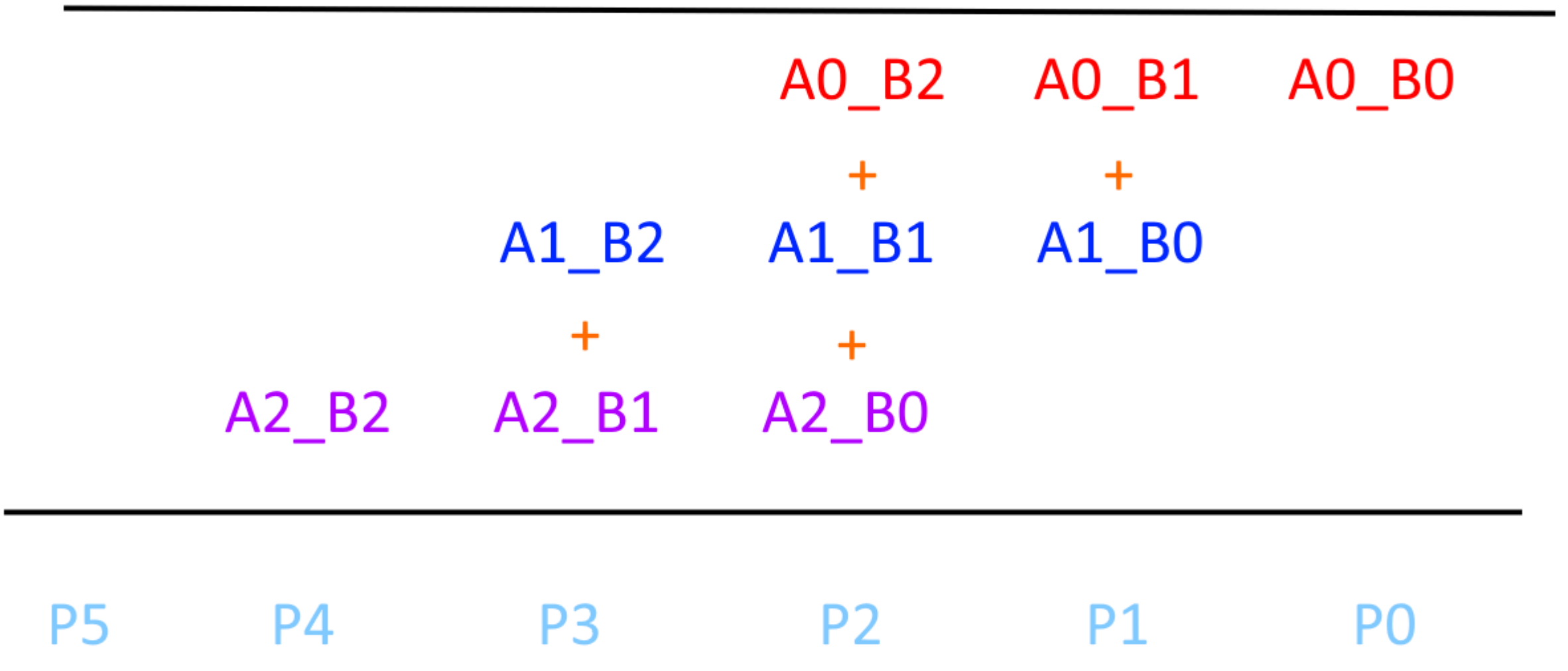


$$
\begin{cases}
y_0 = x_0 \\
y_1 = 0 \\
y_2 = \overline{x_0}\, x_1 \\
y_3 = x_0 x_1 \overline{x_2} + x_0 \overline{x_1} x_2 \\
y_4 = \overline{x_1}\, x_2 + x_0 x_1 x_2 \\
y_5 = x_1 x_2
\end{cases}
$$

Da qui il disegno del circuito dovrebbe essere facile.

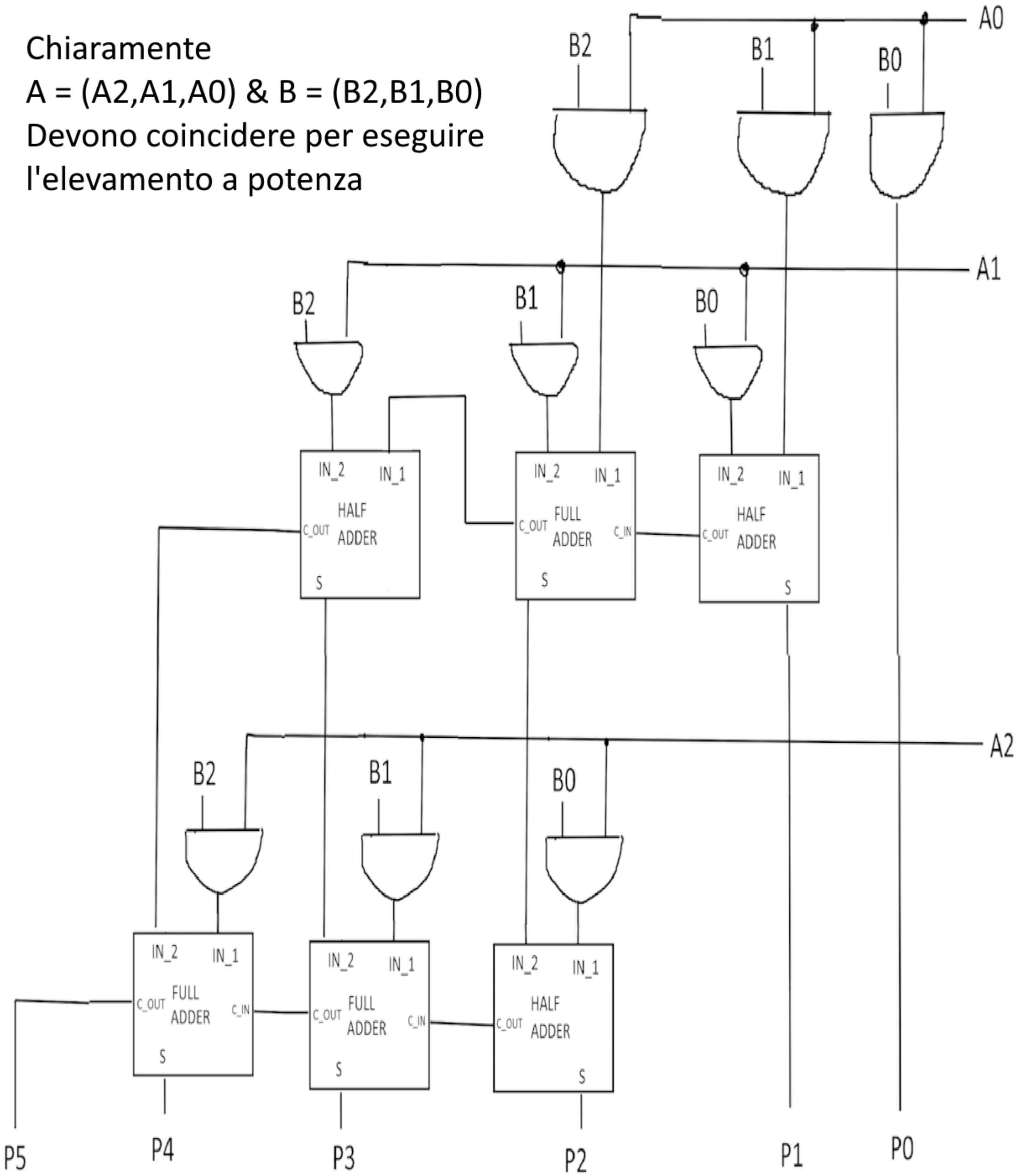
Esiste un modo più "elegante" per risolvere il problema,
ossia eseguire l'algoritmo per la moltiplicazione binaria
e costruirne il corrispondente circuito.
(Come riferimento, vi consiglio di andare a controllare
il video "Binary Multiplication Explained (with Examples)"
del canale Youtube "ALL ABOUT ELECTRONICS").





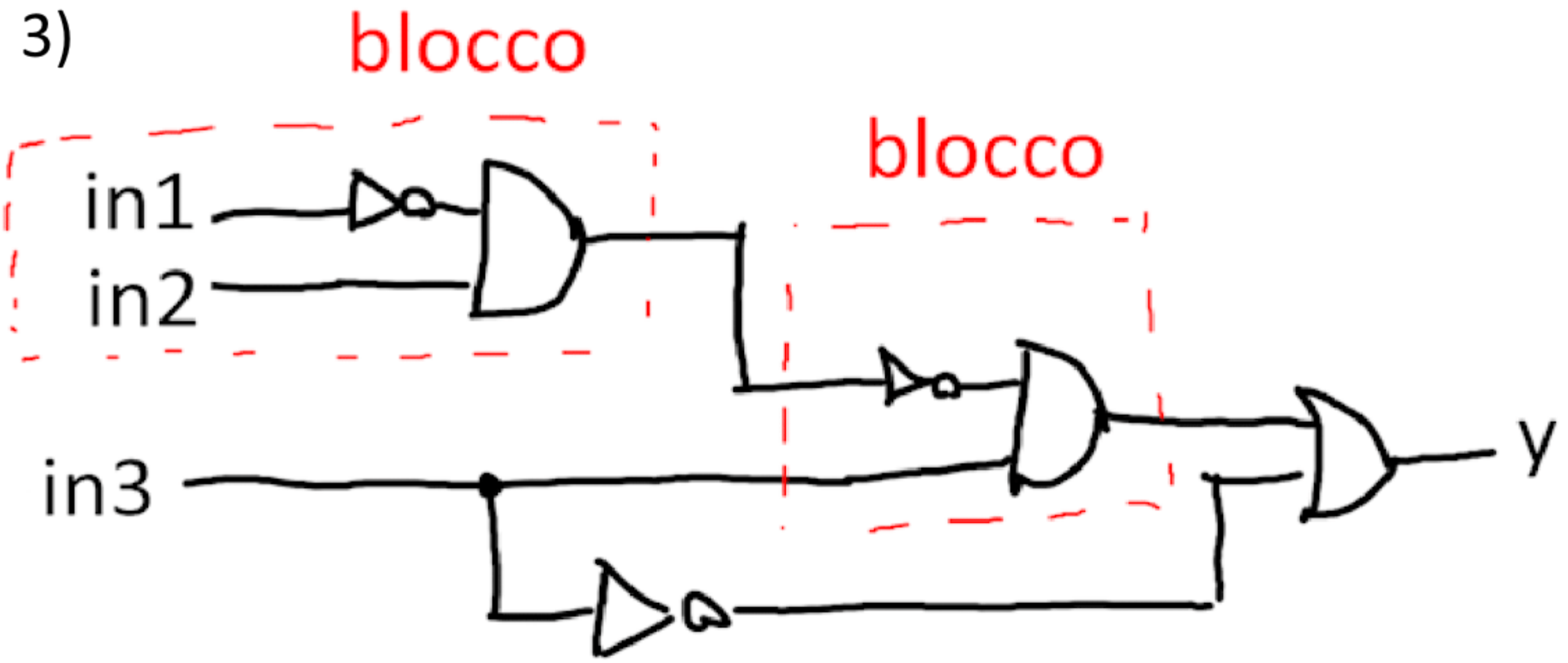
Ora, sappiamo che la porta logica per eseguire la moltipli-
cazione binaria è la porta AND, e sappiamo anche come
costruire un circuito per sommare uno o più numeri binari
in input, a questo punto non ci resta che unire i pezzi, ricor-
dando che il resto della somma dei circuiti sommatori
precedenti va riportato ai sommatori successivi, in questo
modo:

Chiaramente
A = (A2,A1,A0) & B = (B2,B1,B0)
Devono coincidere per eseguire
l'elevamento a potenza



2) Dopo l'esercizio precedente dovrebbe essere facile.

3)



4)

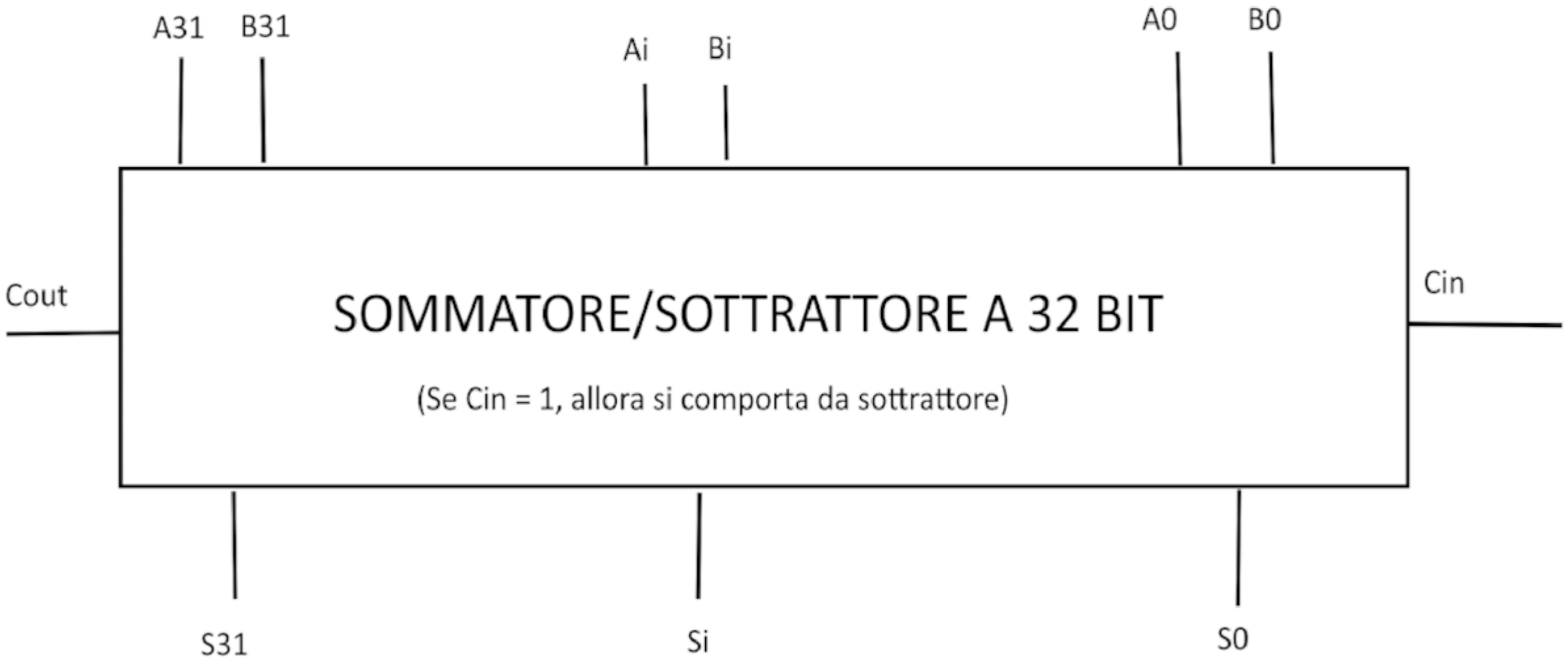
```
module esercizio4(a, b, c, d, y);
        input a, b, c, d;
        output y;

        wire and_ab, and_cd, xor, xor_abcd, or_xorabcd, not_a, and_notad;

        and( and_ab, a, b );
        and( and_cd, c, d );
        xor( xor_abcd, and_ab, and_cd );
        or( or_xorabcd, xor_abcd, and_ab );

        not( not_a, a );
        and( and_notad, not_a, d );

        or( y, and_notad, or_xorabcd );
endmodule
```

5)

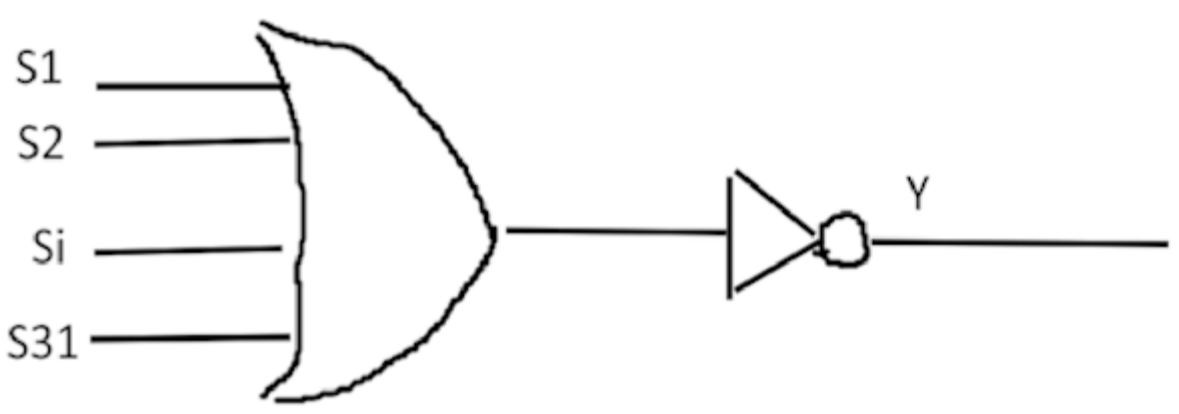
```
module sr_latch_nor( s, r, q, not_q );
        input s, r;
        output q, not_q;

        nor( q, r, not_q );
        nor( not_q, q, s );
endmodule
```

6)

6.1) OSS: se le due sequenze, diciamo A & B, sono diverse, allora A-B != 0, nelle Slides "ep17" è presente un circuito che permette di effettuare facilmente la sottrazione tra due numeri ad "n" bit, applichiamolo a questo caso:



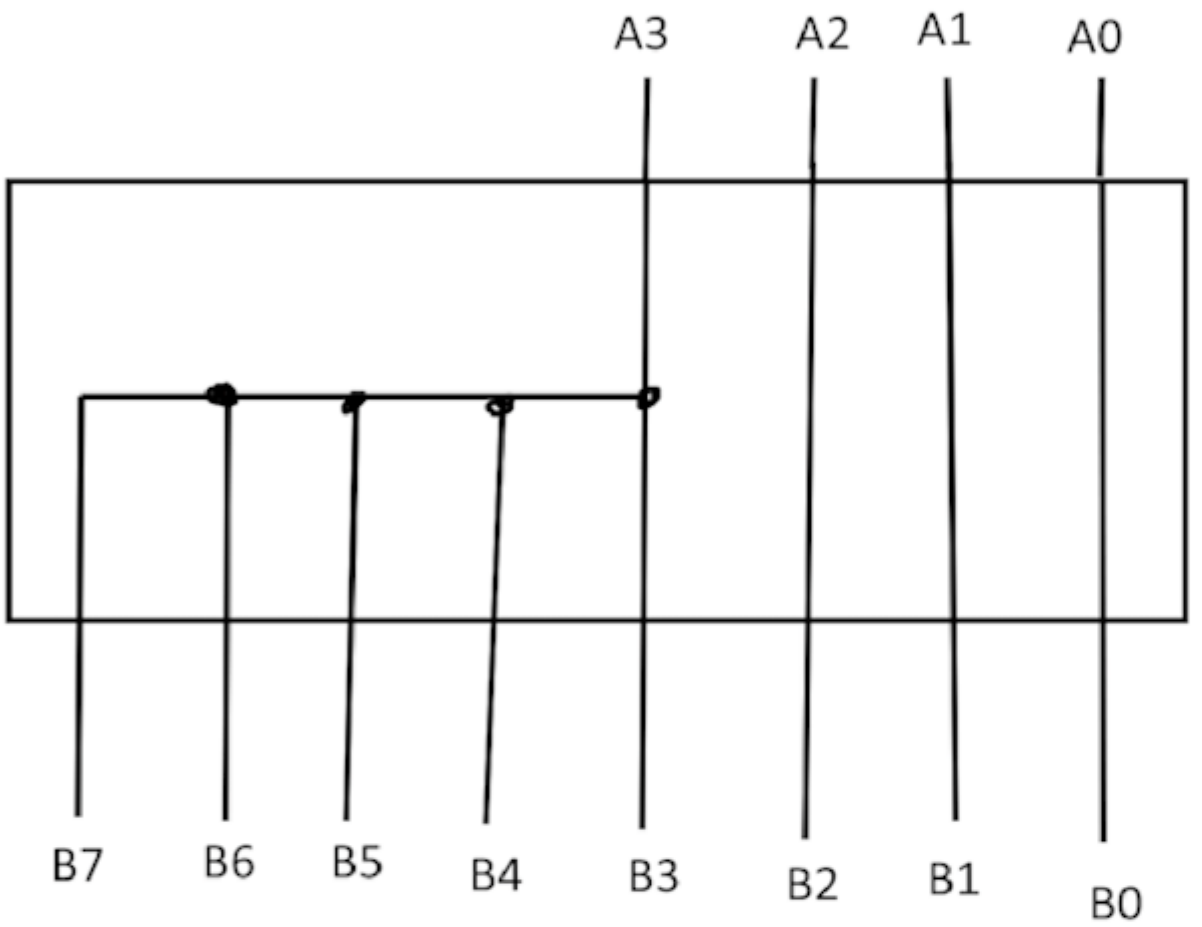
Ora, se le due sequenze sono uguali, allora S0 = S1 = ... = S31 = 0, quindi, ci basta mettere ogni Si (i=0,1,...,31) in una grande OR seguita da una NOT, in questo modo se anche solo uno dei vari Si è uguale ad 1, allora dalla OR uscirà 1, che negato fa 0, viceversa, se ognuno degli Si è uguale a 0, allora dalla OR uscirà 0, che negato fa 1, così:



6.2) Per controllare se A >= B, ci basta controllare Cout, se Cout = 1, allora A >= B, e ritorno 1, viceversa, se Cout = 0, allora A < B, e ritorno 0, provate qualche numero (magari non a 32 bit, ma a 4 o 8) e vedrete che l'approccio funziona.

6.3) Il contrario del punto precedente, se Cout = 0, allora ritorno 1, se Cout = 1, allora ritorno 0.

7)



Se A3 (ossia l'MSB) è pari ad 1, allora il numero è negativo e va esteso con altri 4 bit posti ad 1, se A3 è pari a 0, allora il numero è positivo e va esteso con altri 4 bit posti a 0.

8) Proviamo a seguire il consiglio:

$$\sum_{i=1}^{n} r^i = \frac{r(1-r^n)}{1-r}$$

PASSO BASE: i = 1

$$\sum_{i=1}^{1} r^i = r = \frac{r(1-r^1)}{1-r}$$ OK

IPOTESI INDUTTIVA: suppongo vero fino ad "n"

PASSO INDUTTIVO: dimostro per "n+1"

$$\sum_{i=1}^{n+1} r^i = \sum_{i=1}^{n} r^i + r^{n+1} = \frac{r(1-r^n)}{1-r} + r^{n+1} =$$

$$= \frac{r - \cancel{r^{n+1}} + \cancel{r^{n+1}} - r^{n+2}}{1-r} = \frac{r(1-r^{n+1})}{1-r} \quad \blacksquare$$

8.1)

$$MAX = (2^{16}-1) + \sum_{i=1}^{16} \left(\frac{1}{2}\right)^i = 65535 + \left[\cancel{\frac{1}{2}}\left(\frac{1-\left(\frac{1}{2}\right)^{16}}{\cancel{1-\frac{1}{2}}}\right)\right] =$$

$$= 65535 + \left(1-\left(\frac{1}{2}\right)^{16}\right) = 65535 + \frac{65535}{65536} =$$

$$= 65535.999984741210937 5$$

$$MIN = 0 \quad (\text{BANALE})$$

8.2)

$$MAX = (2^{15}-1) + \left(1-\left(\frac{1}{2}\right)^{16}\right) =$$

$$= 32767.999984741210937 5$$

$$MIN = -2^{15} = -32767$$

9)

a) $(-13.5625)_{10} = -(0000|1101|1001|0000)_2 =$

$= (1111|0010|0111|0000)_{\bar{2}} =$

$= (F270)_H$

valore assoluto

* ho sommato 1 all'LSB

b) $(42.3125)_{10} = (0010|1010|0101|0000)_{\bar{2}} =$

$= (2A50)_H$

c) $(-17.15625)_{10} = -(0001|0001|0010|1000)_2 =$

$= (1110|1110|1101|1000)_{\bar{2}}$

$= (EED8)_H$

10)

a) $(0101|1000)_{\bar{2}} = 2^2 + 2^0 + \frac{1}{2} = (5.5)_{10}$

b) $(1111|1111)_{\bar{2}} = -(0000|0001)_2 = (-0.0625)_{10}$

c) $(1000|0000)_{\bar{2}} = -2^3 = (-8)_{10}$

d) $(0110|0110)_{\bar{2}} = (6.3725)_{10}$

11)

a)

Segno = 1
Numero = 13.5625 = 1101.1001 = 1.1011001 * 2^3
Exp_F = (3+127)_10 = (130)_10 = (1000|0010)_2
Mantissa = 1011001 00000000000000
(-13.5625) = (1100|0001|0101|1001|0000|0000|0000|0000)_IEEE754 =
= (C1590000)_H

b)
(42.3125)_10 = (0100|0010|0010|1001|0100|0000|0000|0000)_IEEE754 =
= (42294000)_H

c)
(-17.15625)_10 = (1100|0001|1000|1001|0100|0000|0000|0000)_IEEE754 =
= (C1894000)_H

12)

(C0123000)_H = (1 10000000 00100100011000000000000)_IEEE754 =
= - (1.14208984375) * 2^(128-127) =
= (-2.2841796875)_10

(81C56000)_H = (1 00000011 10001010110000000000000)_IEEE754 =
= -(1.5419921875) * 2^(3-127) =
= (-1.5419921875 * 2^(-124))_10

(D0B10301)_H = (1 10100001 01100010000001100000001)_IEEE754 =
= -(1.3829041719436646) * 2^(161-127) =
= (- 1.3829041719436646 * 2^34)_10

13) Per i processori, è più facile confrontare due esponenti codificati in eccesso piuttosto che due esponenti codificati con il complemento a 2.
Per esempio, non c'è bisogno di trattare il bit per il segno, poiché tutti gli esponenti saranno rappresentati da numeri pos.

14) Per quanto riguarda i numeri compresi fra (0,1) (quindi 0 ed 1 ESCLUSI), possiamo rappresentare tutti quei numeri con esponente che varia da 0 a 126 , quindi già abbiamo 127 possibili esponenti.

La mantissa è di 23 bits, quindi può rappresentare al massimo 2^23 = 8388608 valori.

Quindi, in totale abbiamo:

#numeri_tra_(0,1) = #esponenti_totali * #mantisse_totali - 2 =
= 127 * 8388608 - 2 = 1.065.353.214

Per quanto riguarda i numeri compresi fra (2^10 , 2^20) (come prima, 2^10 e 2^20 esclusi), il numero di esponenti che possiamo avere è pari a 10 (20-10), come prima, il numero di valori per la mantissa che possiamo avere è pari a 2^23 = 83388608, quindi in totale abbiamo:

10+127 <= #exp_tot <= 20+127 ==> 137 <= #exp_tot <= 147

#exp_tot = 11 (137,138,139,140,141,142,143,144,145,146,147)

#num_tra_(2^10,2^20) = #exp_tot * #mant_tot - 2 =
= 11*8.338.608 - 2 = 92.897.278

15) Quando usavamo 8 bit per l'esponente, sommavamo +127 all'exp del numero binario iniziale, ossia 2^(8-1) - 1 = 128-1 = 127.
Seguendo la stessa logica, se utilizziamo 11 bit per l'esponente, dobbiamo sommare 2^(11-1) - 1 = 2^10 - 1 = 1023

16)

STEP 1: scriviamo i 2 numeri in notazione esponenziale, lasciando la mantissa in binario, per esempio:

X = 1.mantissa_binaria * 2^(esponente)

STEP 2: allineiamo i due esponenti, in particolare, allineiamo l'exp più piccolo a quello più grande, per esempio:

X = 1.mantissa_binaria_1 * 2^(esponente_1) =
= 1.0001111 * 2^5

Y = 1.01001 * 2^4 = 0.101001 * 2^5

STEP 3: sommiamo (sottraiamo) le mantisse in base al loro segno

STEP 4: abbiamo il segno, abbiamo l'esponente, abbiamo la mantissa, uniamo i componenti e "costruiamo" il nuovo numero, stando attenti a normalizzare la nuova mantissa se necessario.

Esistono diversi video su Youtube in merito, per esempio, vi consiglio:
1) "HOW TO: Adding IEEE-754 Floating Point Numbers"
di "Steven Petryk"
2) "IEEE 754 Standard for Floating Point Binary Arithmetic"
di "Computer Science"

17)

(C0123456)_H = (1 10000000 00100100011010001010110)_IEEE754
(81C564B7)_H = (1 00000011 10001010110010010110111)_IEEE754

Il 1o esponente è 1, il 2o esponente è -124

1.00100100011010001010110 * 2^1 +
1.10001010110010010110111 * 2^(-124)

Dobbiamo spostare di 125 posizioni il 2o numero, è come se stessimo sommando al 1o numero qualcosa di molto, MOLTO piccolo.
Perciò, quando andiamo a spostare di 125 posizioni verso destra la mantissa del 2o numero, il suo valore sarà completamente troncato, lasciando solo il valore del 1o numero come risultato finale

(D0B10301)_H = (1 10100001 01100010000001100000001)_IEEE754
(D1B43203)_H = (1 10100011 01101000011001000000011)_IEEE754

Il 1o esponente è 34, il 2o esponente è 36, dobbiamo spostare il 1o numero di 2 posizioni verso destra, scrivendo le mantisse in binario, i numeri diventano:

(D0B10301)_H = - (0.01011000100000011000000001) * 2^36
(D1B43203)_H = - (1.01101000011001000000011) * 2^36

Sommiamo le mantisse, ottenendo:
MANTISSA_FIN = 1.11000000111001011000011 01 (tronchiamo)
NUMERO_FIN = - 1.11000000111001011000011 * 2^36 =
= (1 10100011 11000000111001011000011)_IEE754
= (D1E072C3)_H

(Lo so, non è un calcolo proprio semplice, ma provate a farlo convertendo prima i numeri in decimale e vedrete che torna)

IEEE 754 Converter (JavaScript), V0.22

	Sign	Exponent	Mantissa
Value:	-1	2^{36}	1.7535022497177124
Encoded as:	1	163	6320835
Binary:	☑	☑ ☐ ☑ ☐ ☐ ☐ ☑ ☑	☑ ☑ ☐ ☐ ☐ ☐ ☐ ☐ ☑ ☑ ☑ ☐ ☐ ☑ ☐ ☑ ☑ ☐ ☐ ☐ ☐ ☑ ☑

You entered: -120499759104.0
Value actually stored in float: -120499757056
Error due to conversion: 2048.0
Binary Representation: 11010001111000000111001011000011
Hexadecimal Representation: 0xd1e072c3

+1
-1

(5EF10324)_H = (0 10111101 11100010000001100100100)_IEEE754
(5E039020)_H = (0 10111100 00000111001000000100000)_IEEE754

Entrambi i numeri sono positivi, l'EXP del 1o è 62, mentre del 2o è 61, spostiamo la mantissa del 2o di una posizione a destra e sommiamo le due mantisse:

Mantissa_1 = 1.11100010000001100100100
Mantissa_2 = 1.00000111001000000100000, che diventa:
0.10000011100100000100000

Vi ricordo che qui sto considerando le mantisse espresse in binario per semplice comodità!!!

Vi ricordo inoltre che il bit meno significativo della 2a mantissa, quando viene shiftato verso dx, viene troncato.

Somma delle Mantisse:

$$\begin{array}{r} 1.11100010000001100100100 \\ 0.10000011100100000010000 + \\ \hline 10.01100101100101100110100 \end{array}$$

Shiftiamo ulteriormente questa mantissa verso dx di una posizione per normalizzare, l'esponente aumenterà di +1

Mantissa finale: 1.00110010110010110011010 0
L'esponente aumenta da 62 a 63, che sommato a 127 fa 190, in binario: 10111110
Il segno rimane 0

Numero finale : (0 10111110 00110010110010110011010 0)_IEEE754
In Hex : (5F19659A)_H

IEEE 754 Converter (JavaScript), V0.22

	Sign	Exponent	Mantissa
Value:	+1	2^{63}	1.1984131336212158
Encoded as:	0	190	1664410
Binary:	☐	☑ ☐ ☑ ☑ ☑ ☑ ☑ ☐	☐ ☐ ☑ ☑ ☐ ☐ ☑ ☐ ☑ ☑ ☐ ☐ ☑ ☐ ☑ ☑ ☐ ☐ ☑ ☑ ☐ ☑ ☐

You entered	11053410185240000000
Value actually stored in float:	11053410185241427968
Error due to conversion:	1427968
Binary Representation	01011111000110010110010110011010
Hexadecimal Representation	0x5f19659a

+1
-1

18)

(0.25)_10 = (0 01111101 00000000000000000000000)_IEEE754

(0.30)_10 = (0 01111101 00110011001100110011010)_IEEE754

Ci aspettiamo che il primo "printf" stampi i due numeri di cui sopra approssimati alla 3a cifra decimale (%.3f), quindi per il 1o numero stampi "0.250" e per il 2o numero stampi "0.300".

Il secondo "printf" dovrebbe fare la stessa cosa, ma approssimando i due numeri alla 30esima cifra decimale (%.30f), giusto? Eppure, quando andiamo ad eseguire il codice, ecco cosa viene stampato a schermo:

main.c

```c
#include <stdio.h>

int main(){
    float a = 0.25;
    float b = 0.3;

    printf("PRIMO PRINTF:\na=%.3f; b=%.3f\n\n", a , b );
    printf("SECONDO PRINTF:\na=%.30f; b=%.30f\n", a , b );

}
```

input

```
PRIMO PRINTF:
a=0.250; b=0.300

SECONDO PRINTF:
a=0.250000000000000000000000000000; b=0.300000011920928955078125000000


...Program finished with exit code 0
Press ENTER to exit console.
```

???

Ma perché?

Ciò è dovuto ad errori di approssimazione commessi dal computer quando deve rappresentare alcuni numeri in virgola mobile. Ma quali? Tutti quelli per i quali abbiamo difficoltà a rappresentarli come una somma di (1/2)^i, come per esempio 0.1, 0.2, 0.3, 0.4, 0.6, 0.7, 0.8 e 0.9, ma non 0.5, 0.25, 0.75, etc...

19)

Ci aspetteremmo che stampi correttamente
a = 30000000
b = 30000001

E invece:

main.c

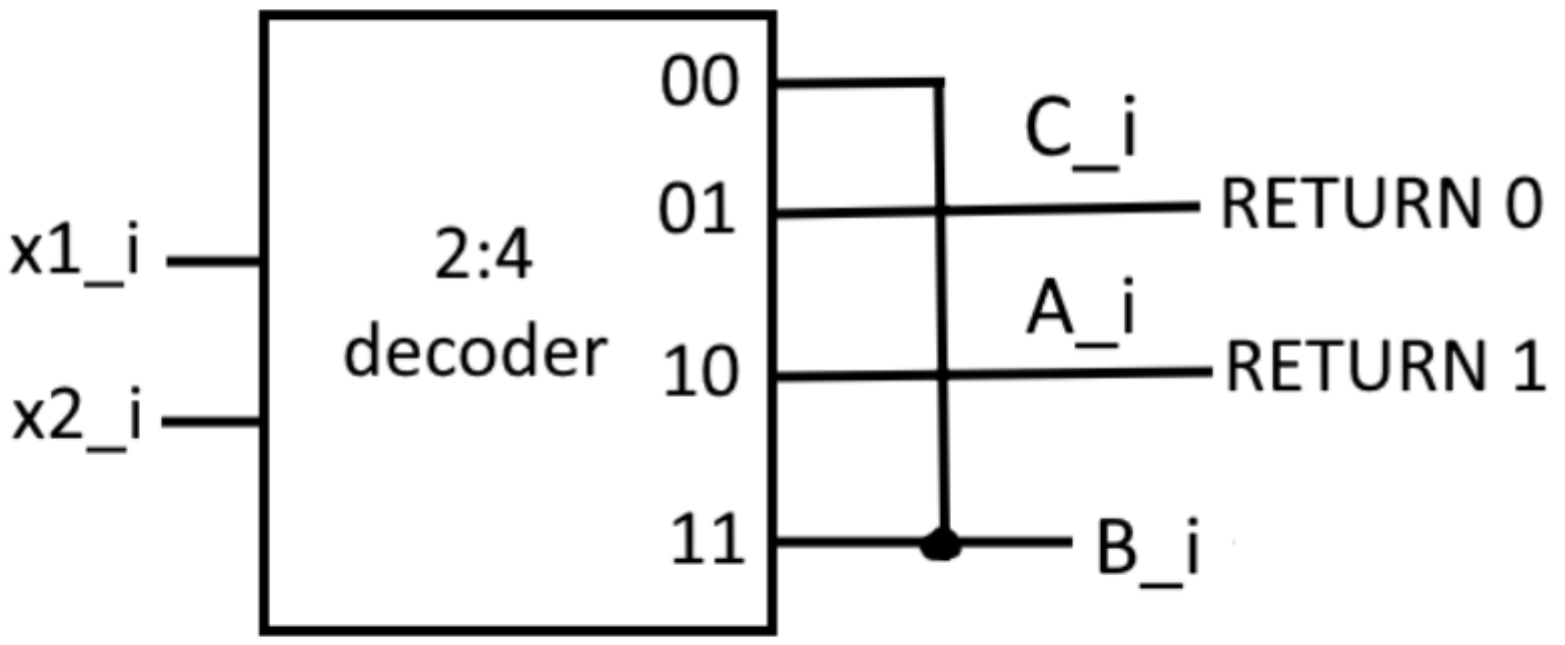
```c
# include <stdio.h>

int main (){
    float a = 30000000; // Trenta milioni
    float b = 30000001; // Trenta milioni più uno
    printf("PRIMO PRINTF:\na=%f\n\n",a);
    printf("SECONDO PRINTF:\nb=%f\n",b);
}
```

```
PRIMO PRINTF:
a=30000000.000000

SECONDO PRINTF:
b=30000000.000000


...Program finished with exit code 0
Press ENTER to exit console.
```

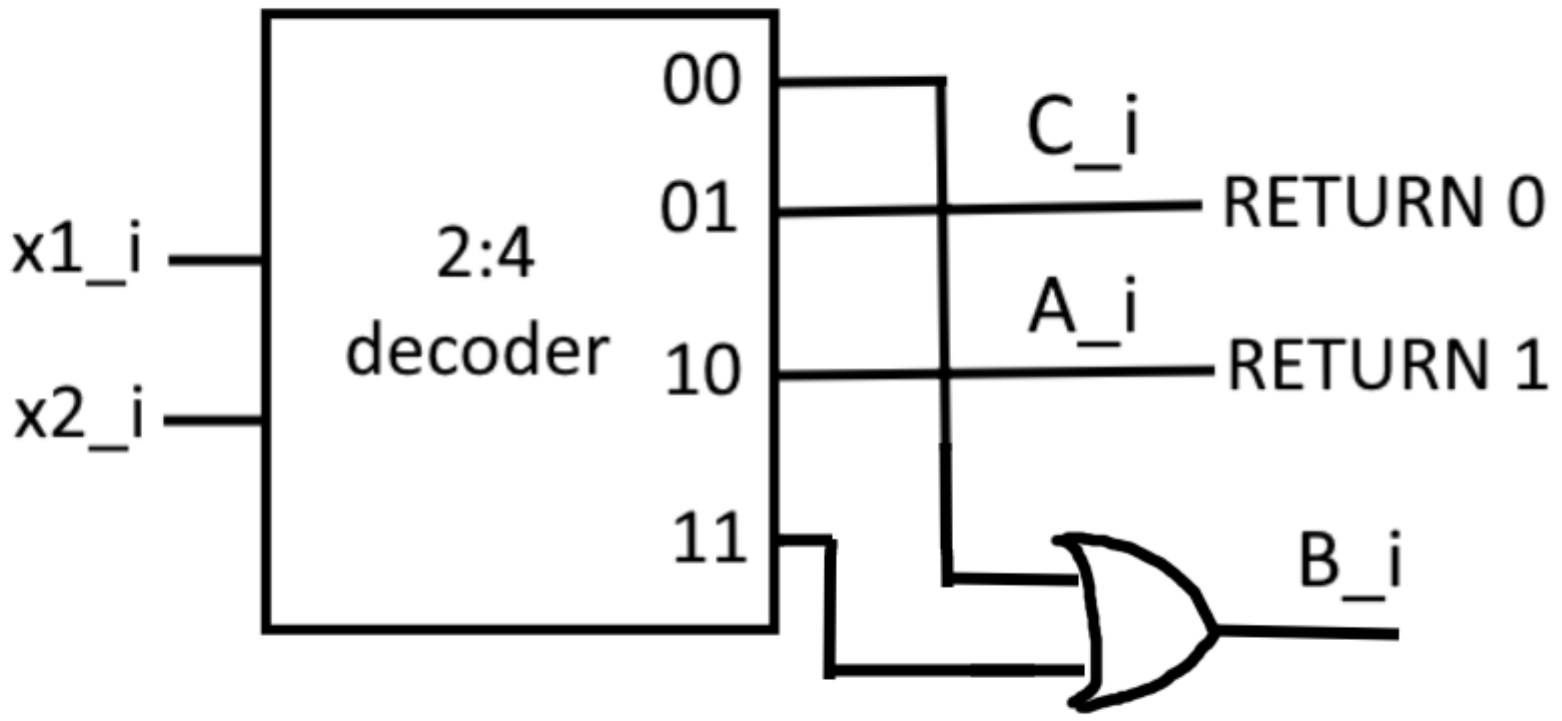
Questo perché, rappresentando b come un float (32 bit con precisione fino a 7 cifre decimali), la macchina è costretta ad effettuare delle approssimazioni che possono portare ad errori di approssimazione come quello appena osservato, e se anziché usare un float usassimo un double (64 bit con precisione fino a 15 cifre decimali)?

Correzioni 16/12/2024

9)



In questo circuito le porte dell'output 00 e 11 dovrebbero essere messe in OR per produrre B_i:



19)

Nel disegno dell'automa c'è un errore, la freccia che parte da S2 e rientra in S2 dovrebbe avere 1/0 come label, e non 1/1, perché naturalmente abbiamo letto 111, e non 1101

Inoltre, l'ultima Clausola dell'equazione per D2 dovrebbe essere Q1 !Q2 !X, e non Q1 !Q2 X, manca un " ! " davanti X