Matteo Di Gioacchino
Soluzioni Esercitazione 02/12/2024

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

1)

1.1)

11	2 1
5	2 1
2	2 0
1	2 1

1011_2

1.2)

87	2 1
43	2 1
21	2 1
10	2 0
5	2 1
2	2 0
1	2 1

1010111_2

1.3)

118	2 0
59	2 1
29	2 1
14	2 0
7	2 1
3	2 1
1	1 1

1110110_2

1.4)

365	2 1
182	2 0
91	2 1
45	2 1
22	2 0
11	2 1
5	2 1
2	2 0
1	2 1

101101101_2

1.5)

512	2 0
256	2 0
128	2 0
64	2 0
32	2 0
16	2 0
8	2 0
4	2 0
2	2 0
1	2 1

1000000000_2

2) Vi scrivo la mia data di nascita direttamente in binario, provate a convertirla in decimale

10001 / 11 / 11111001110

Ci chiediamo, come potrebbe una macchina capire quando termina una parte della data e quando inizia l'altra?
Osserviamo che la data piu' grande possibile, escluse date che vanno dall'anno 10.000+, sarebbe 31/12/9999

Ossia:

11111 / 1100 / 10011100001111

Restringendoci ad un anno più realistico (2024), ci servirebbero 11 bit, tenendo sempre in considerazione che, quando si raggiungerà un anno abbastanza grande (quale?) avremmo bisogno di 1 bit in più.

EXTRA:

Tipicamente una macchina lavora male con parole che non siano della lunghezza di un multiplo di 8 bit (1 byte), approssimiamo quindi a:
8 bit / 8 bit / 16 bit
e aggiungiamo un numero di 0 davanti alla data calcolata all'inizio per renderla facilmente leggibile ad una macchina

00010001 / 00000011 / 0000011111001110

il 1o byte sarà il giorno, il 2o byte il mese e i restanti 2 byte saranno l'anno

Per quanto riguarda invece l'esadecimale, ci serviranno almeno 2 cifre per coprire tutti i numeri da 1 a 31 (per i giorni), 1 cifra che copra da 1 a 12 (per i mesi) e 4 cifre per coprire tutti i numeri da 1 a 9999 (per gli anni)

3)

3.1) $1010_2 = 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 8 + 2 = 10_{10}$

3.2) $110110_2 = 1*2^5 + 1*2^4 + 1*2^2 + 1*2^1 + 0*2^0 = 32+16+4+2 = 54_{10}$

3.3) $11001100_2 = 1*2^7 + 1*2^6 + 1*2^3 + 1*2^2 = 128 + 64 + 8 + 4 = 204_{10}$

3.4) $11110000_2 = 240_{10}$

3.5) $01011010_2 = 90_{10}$

4) abbiamo due possibili modi, il primo è dividere i numeri per 16 e mantenere progressivamente il resto, il secondo è trasformare i numeri in binario, aggiungere tanti 0 quanto bastano per trasformare il numero di bit in un multiplo di 4, successivamente, raggrupparli in "gruppi" da 4 bit ciascuno e convertire quei 4 gruppi in esadecimale.
Per i numeri dell'esercizio 1 utilizziamo il 1o metodo, per quelli dell'esercizio 3 utilizziamo il 2o.

4.1.1) 11 | 16 11 = B 11_10 = B_hex

4.1.2)
```
87 | 16  5
 7 | 16  7
```
87_10 = 57_hex

4.1.3)
```
118 | 16  7
  6 | 16  6
```
118_10 = 76_hex

4.1.4)
```
365 | 16  13 = D
 22 | 16   6
  1 | 16   1
```
365_10 = 16D_hex

4.1.5)
```
512 | 16  0
 32 | 16  0
  2 | 16  2
```
512_10 = 200_hex

4.2.1) 1010_2 = A_hex

4.2.2) 0011 0110_2 = 36_hex

4.2.3) 1100 1100_2 = CC_hex

4.2.4) 1111 0000_2 = F0_hex

4.2.5) 0101 1010_2 = 5A_hex

5) 5.1) A5_hex = 1010 0101_2 = 165_10
5.2) 35B1_hex = 0011 0101 1011 0001_2 = 13745_10
5.3) CEE22_hex = 1100 1110 1110 0010 0010_2 = 847394_10
5.4) 6E42_hex = 0110 1110 0100 0010_2 = 28226_10
5.5) D0000000_hex = 1101 0000 0000 0000 0000 0000 0000 0000_2 =
= 3489660928_10

6) 0350191 = 350191_10 =
= 0101 0101 0111 1110 1111_2 =
= 557EF_hex

7) per comodità: 2* := complemento a 2

Per convertire in complemento a 2:
- Numeri Positivi: scrivo il numero in binario come sempre
- Numeri Negativi: scrivo il numero in binario, inverto tutti gli 0 e 1, aggiungo 1 al bit meno significativo (LSB)

7.1) 11_10 = 0000001011_2

7.2) 87_10 = 0001010111_2
-87_10 = 1110101000 + 1 = 1110101001_2*

7.3) 118_10 = 0001110110_2

7.4) 365_10 = 0101101101_2
-365_10 = 1010010010 + 1 = 1010010011_2*

7.5) 512_10 = 1000000000_2
-512_10 = 011111111 + 1 = 1000000000_2*

8) 8.1) 010101_2* = -0*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0 =
= 16 + 4+ 1 = 21_10

8.2) 101010_2* = -32 + 8 + 2 = -22_10

8.3) 110101_2* = -32 + 16 + 4 + 1 = -11_10

8.4) 011111_2* = 16 + 8 + 4 + 2 + 1 = 31_10

8.5) 111111_2* = -32 + 16 + 8 + 4 + 2 +1 = -1_10

9) 9.1) 16_10 = 010000_2* 9_10 = 001001_2*

010000 +
001001 =
011001 = 25_10

NO OVERFLOW

9.2) 27_10 = 011011_2* 31_10 = 011111_2*

011011 +
011111 =
111010 = -6_10

OVERFLOW

9.3) -4_10 = 111100_2* 19_10 = 010011_2*

111100 +
010011 =
1 001111 = 15 10

ANCHE SE 1 BIT DI RESTO "VA SPRECATO",
PER LA DEFINIZIONE, NON ABBIAMO OVERFLOW

9.4) 3_10 = 000011_2* -32_10 = 100000_2*

000011 +
100000 =
100011 = -29_10

NO OVERFLOW

9.5) -27_10 = 100101_2* -31_10 = 100001_2*

100101 +
100001 =
1 000110 = 6_10

OVERFLOW

10) M = 1001101 a = 1100001 t = 1110100
e = 1100101 o = 1101111

Matteo = 1001101 1100001 1110100 1110100 1100101 1101111

M = 0100 1101_2 = 4D_hex a = 0110 0001_2 = 61_hex
t = 0111 0100_2 = 74_hex e = 0110 0101_2 = 65_hex
o = 0110 1111_2 = 6F_hex

11)

Assorbimento	$x(x+y)=x$	$x+xy=x$
Combinazione	$xy+x\bar{y}=x$	$(x+y)(x+\bar{y})=x$
Consenso	$xy+\bar{x}z+yz=xy+\bar{x}z$	$(x+y)(\bar{x}+z)(y+z)=(x+y)(\bar{x}+z)$

Assorbimento:

x	y	(x+y)	x(x+y)
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

x	y	xy	x+xy
0	0	0	0
0	1	0	0
1	0	1	1
1	1	1	1

Combinazione:

x	y	!y	xy	x!y	xy+x!y
0	0	1	0	0	0
0	1	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1

x	y	!y	x+y	x+!y	(x+y)(x+!y)
0	0	1	0	1	0
0	1	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	1

Consenso:

x	y	z	!x	xy	!xz	yz	xy + !xz	xy + !xz + yz
0	0	0	1	0	0	0	0	0
0	0	1	1	0	1	0	1	1
0	1	0	1	0	0	0	0	0
0	1	1	1	0	1	1	1	1
1	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0
1	1	0	0	1	0	0	1	1
1	1	1	0	1	0	1	1	1

x	y	z	!x	(x+y)	(!x+z)	(y+z)	(x+y)(!x+z)	(x+y)(!x+z)(y+z)
0	0	0	1	0	1	0	0	0
0	0	1	1	0	1	1	0	0
0	1	0	1	1	1	1	1	1
0	1	1	1	1	1	1	1	1
1	0	0	0	1	0	0	0	0
1	0	1	0	1	1	1	1	1
1	1	0	0	1	0	1	0	0
1	1	1	0	1	1	1	1	1

12)

Assorbimento:

- x(x+y) = xx + xy = x + xy = x(1+y) = x
- x + xy = x(1+y) = x

Combinazione:

- xy + x!y = x(y+!y) = x
- (x+y)(x+!y) = xx + x!y + xy + y!y = x + x(!y+y) + 0 = x + x = x

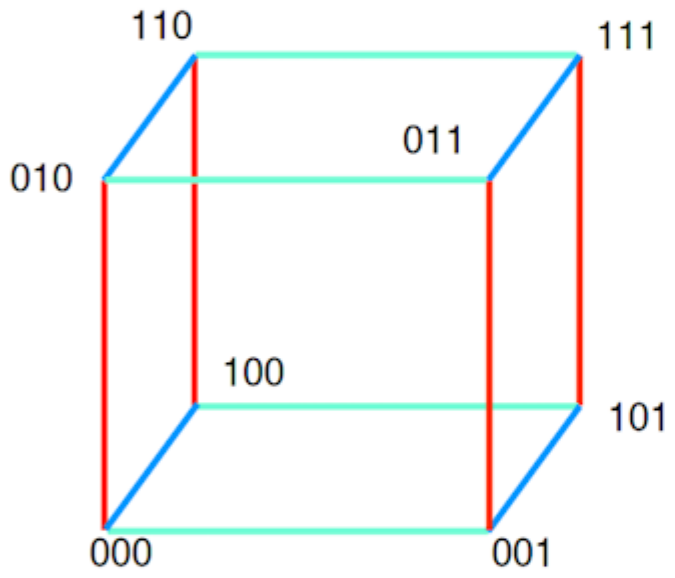
Consenso:

- Se "x=1", allora: "xy + !xz + yz = 1*y + 0*z + yz = y + yz = y(1+z) = y"
  yz risulta ridondante, possiamo rimuoverlo
- Se "x=0", allora "xy + !xz + yz = 0*y + 1*z + yz = z + yz = z(1+y) = z"
  yz risulta ridondante, possiamo rimuoverlo

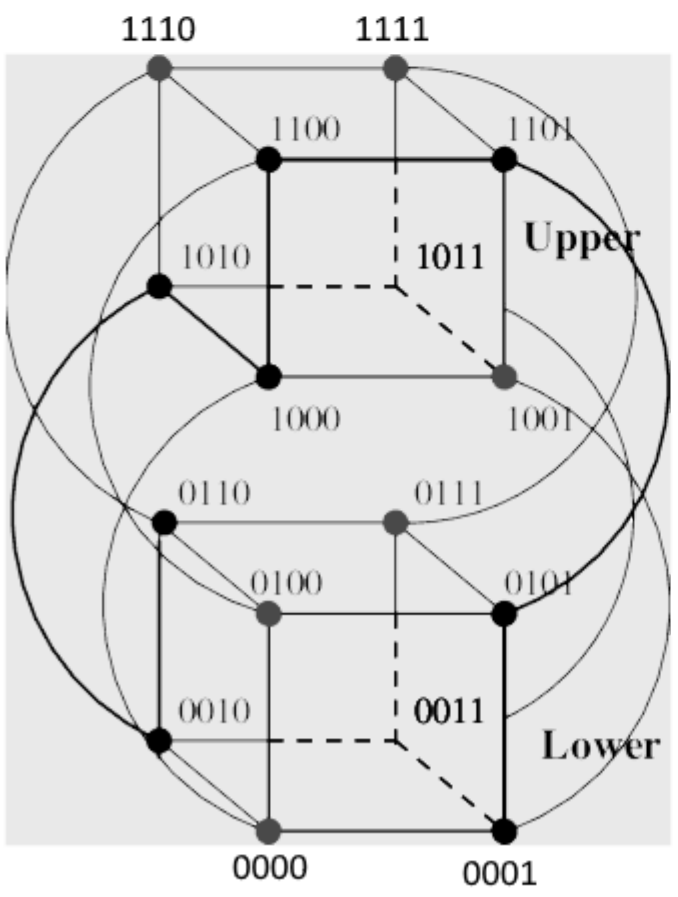
- Se "x=1", allora "(x+y)(!x+z)(y+z) = (1+y)(0+z)(y+z) = (y)(z)(y+z) = (yz)(y+z)"
  (y+z) risulta ridondante, possiamo rimuoverlo
- Se "x=0", allora "(x+y)(!x+z)(y+z) = (0+y)(1+z)(y+z) = (y)(z)(y+z) = (yz)(y+z)"
  (y+z) risulta ridondante, possiamo rimuoverlo

13)

3-CUBO: V = {0,1}^3 = { 000, 001, 010, 011, 100, 101, 110, 111 }



4-CUBO: V = {0,1}^3 = { 0000, 0001, 0010, 0011, 0100,
0101, 0110, 0111, 1000, 1001,
1010, 1011, 1100, 1101, 1110, 1111 }



Strategia per costruire un k-cubo
(o k-ipercubo):

- Partite da due (k-1)-(iper)cubi
- Aggiungete 0 davanti alla codifica di ogni nodo in uno dei due (iper)cubi, e 1 davanti alla codifica di ogni nodo dell'altro.
- unite con un arco i nodi che, nelle due codifiche iniziali, avevano le stesse codifiche (Ex: 1.000 & 0.000, 1.010 & 0.010, etc...)

14)

Pensiamoci, un 2-cubo contiene |V| = |{0,1}^2| = |{00,01,10,11}| = 4 Nodi, e |E| = |{(00,01),(00,10),(10,11)(01,11)}| = 4 Archi
Un 3-cubo contiene |V| = 2^3 = 8 Nodi e, pensando alla strategia di costruzione precedente, |E| = (|E'|+|E''|)+4 = (4+4)+4 = 12 Archi
Un 4-cubo contiene |V| = 2^4 = 16 Nodi e |E| = (12+12)+8 = 32 Archi
Vedete un pattern?
Se abbiamo un k-cubo, abbiamo |V| = 2^k nodi. Ogni nodo è connesso a ESATTAMENTE k nodi che differenziano dalla sua codifica per un singolo bit.

Quindi:

$$|E_k| = \frac{2^k \cdot k}{2} = 2^{k-1} k$$

2^k → |V|
k → |nodi-vicini(u)| $\forall\, u \in V$
2 → NON contiano un arco 2 volte, [ex; (00,01) & (01,00)]

Proviamo a dimostrare il risultato su E_k per induzione:

PASSO BASE (k=0, un singolo nodo):

|E_0| = (2^0 * 0) /2 = 0 OK

FACCIAMO ANCHE (k=1, due nodi con un singolo arco):

|E_1| = (2^1 * 1)/2 = 1 OK

IPOTESI INDUTTIVA:

Suppongo vero fino a "k = n"

PASSO INDUTTIVO:

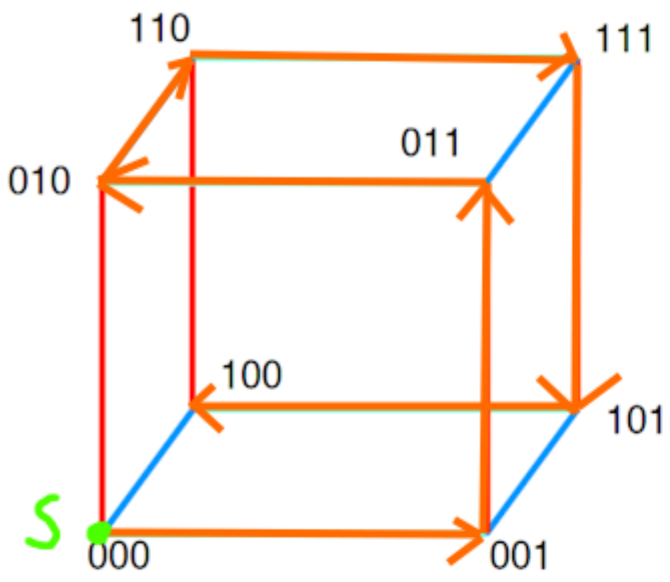
Dimostro per "k = n+1".

Il numero di archi sarà pari a 2 volte il numero di archi presenti nei 2 n-cubi al passo precedente, più 2^n archi che "uniscono" i due n-cubi

|E_(n+1)| = 2*|E_n| + 2^n = 2*(2^(n-1) * n) + 2^n = 2^n * n + 2^n =
= 2^n * (n+1) = 2^(k-1) * k ∎

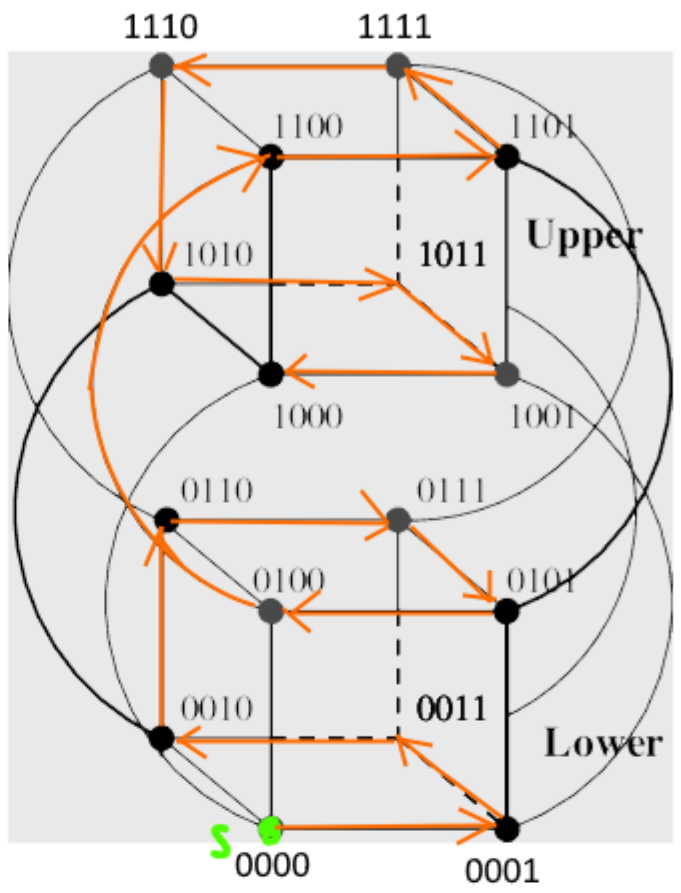
15)

3-cubo:



0 0 0
0 0 1
0 1 1
0 1 0
1 1 0
1 1 1
1 0 1
1 0 0

4-cubo:



4 bit Gray Code

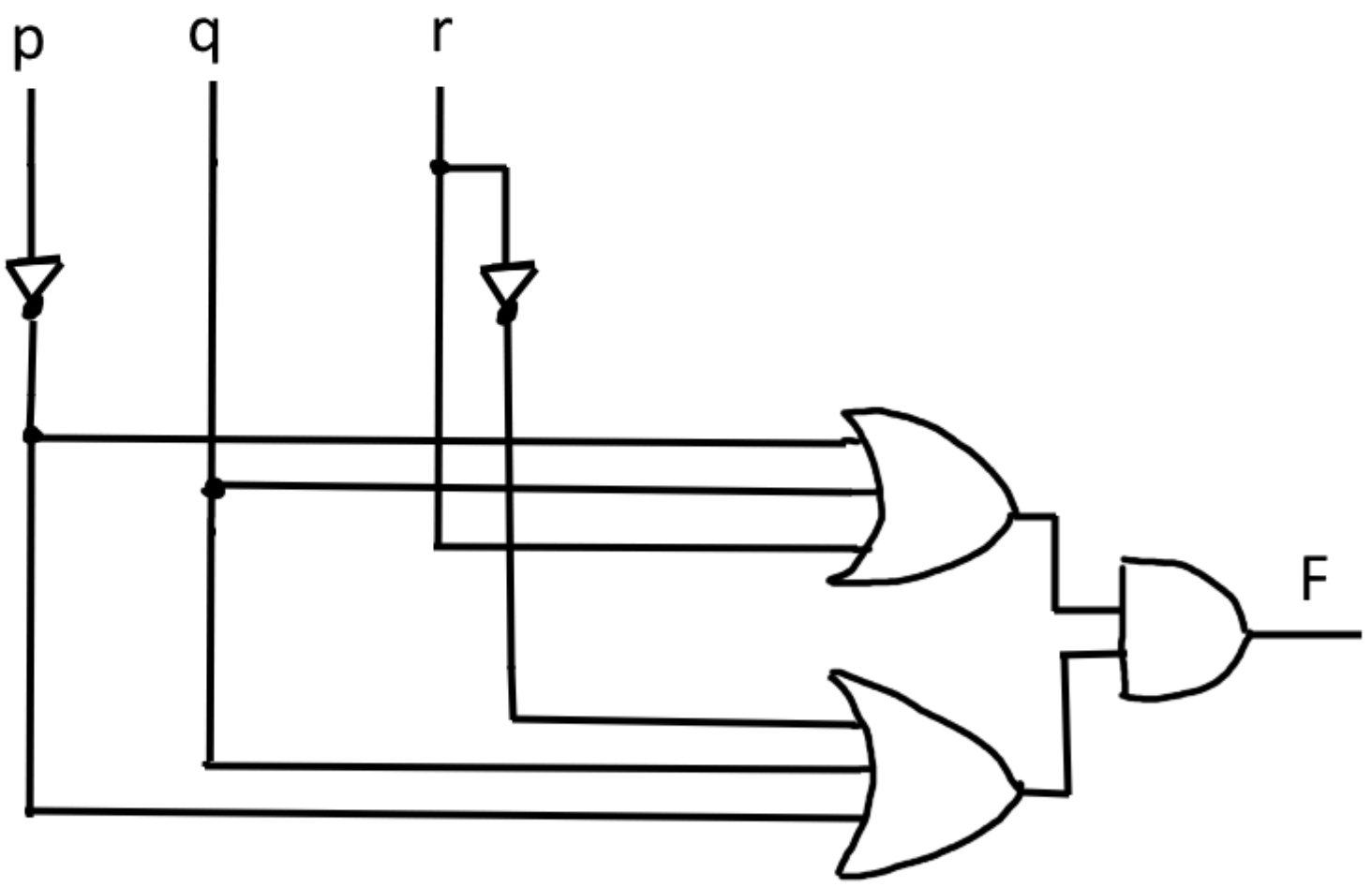
$G_1G_2G_3G_4$

0 0 0 0
0 0 0 1
0 0 1 1
0 0 1 0
0 1 1 0
0 1 1 1
0 1 0 1
0 1 0 0
1 1 0 0
1 1 0 1
1 1 1 1
1 1 1 0
1 0 1 0
1 0 1 1
1 0 0 1
1 0 0 0

16)

p	q	r	((p → (q ∧ r)) ∨ (¬q → ¬p))
F	F	F	T
F	F	T	T
F	T	F	T
F	T	T	T
T	F	F	F
T	F	T	F
T	T	F	T
T	T	T	T

F = (!p + q + r) (!p + q + !r)



17)

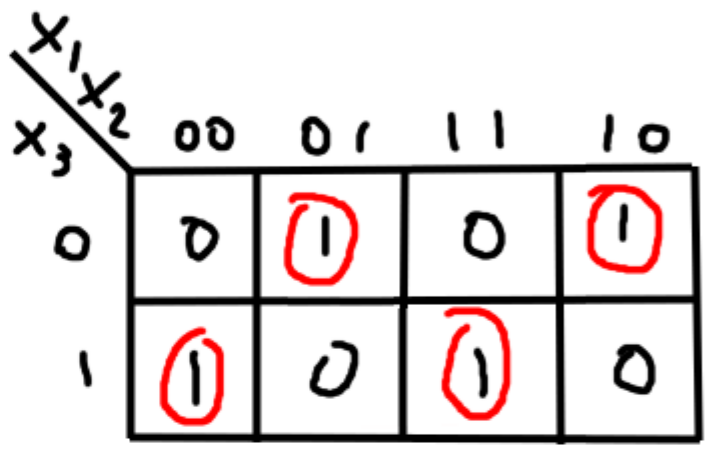
p	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
q	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
r	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
s	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
?	0	0	0	1	1	0	1	1	0	0	0	0	1	0	1	0

Il procedimento è del tutto identico all'Esercizio precedente:

- Scrivete F in CNF o DNF
- Costruite il circuito usando il numero giusto di NOT, OR & AND

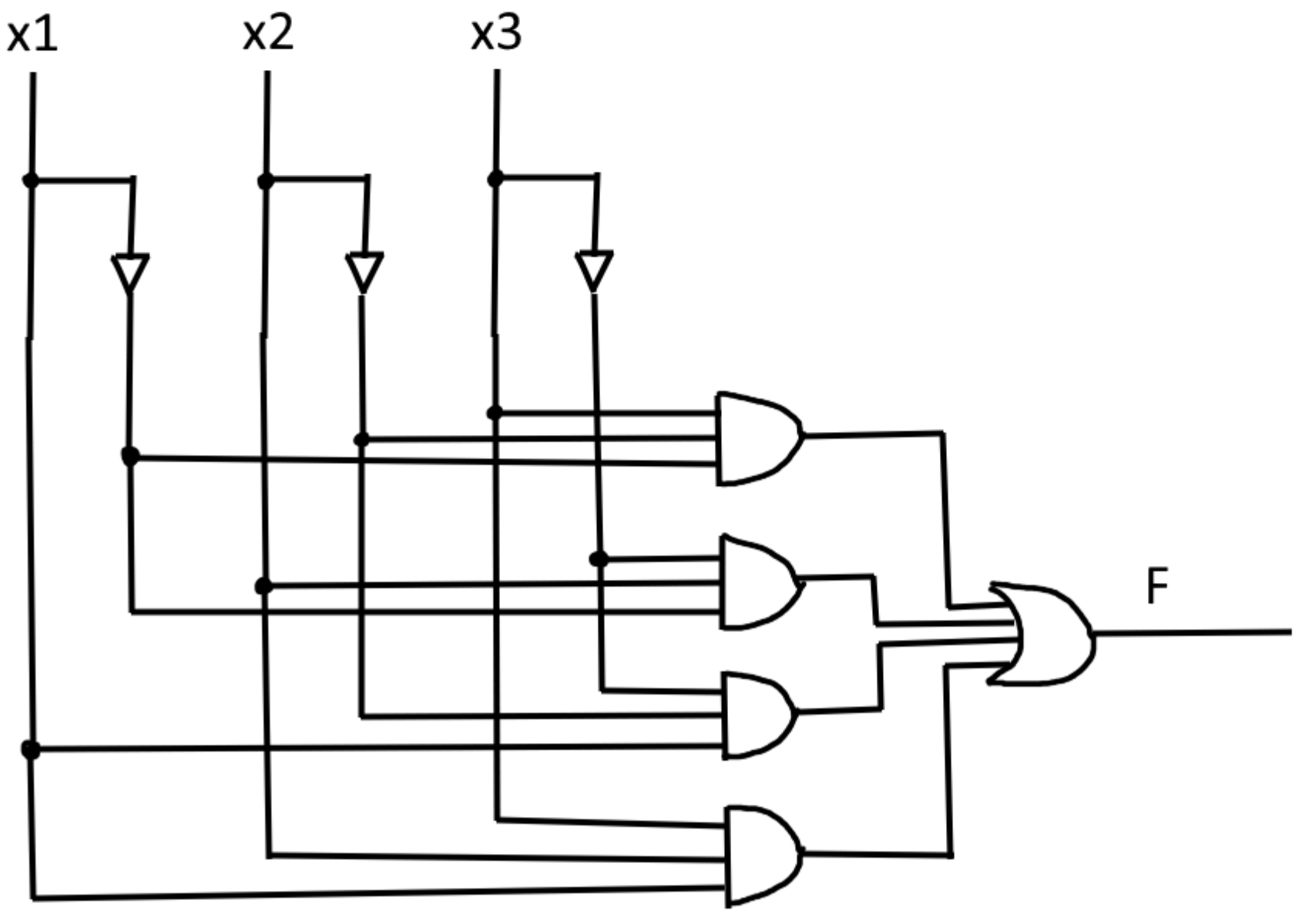
18)

x1	x2	x3	x1 XOR x2	(x1 XOR x2) XOR x3
0	0	0	0	0
0	0	1	0	1
0	1	0	1	1
0	1	1	1	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	0	1



Karnaugh non ci aiuta...

F = (!x1*!x2*x3) + (!x1*x2*!x3) + (x1*!x2*!x3) + (x1*x2*x3)



19)

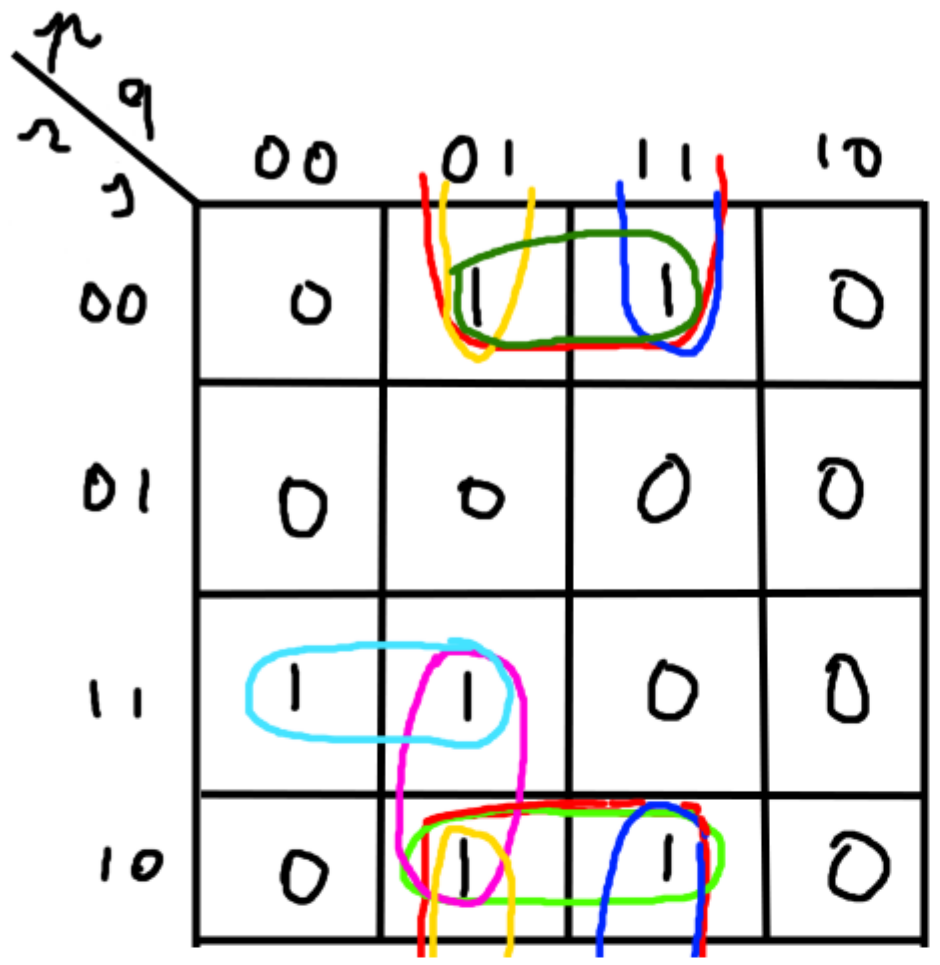
$$f(x_1, x_2, x_3, x_4) = \begin{cases} 1 & \text{se } x_1 + x_2 + x_3 + x_4 \equiv 0 \mod 3 \\ 0 & \text{altrimenti} \end{cases}$$

x1	x2	x3	x4	f(x1,x2,x3,x4)
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Lascio a voi la costruzione del circuito, il procedimento è sempre lo stesso

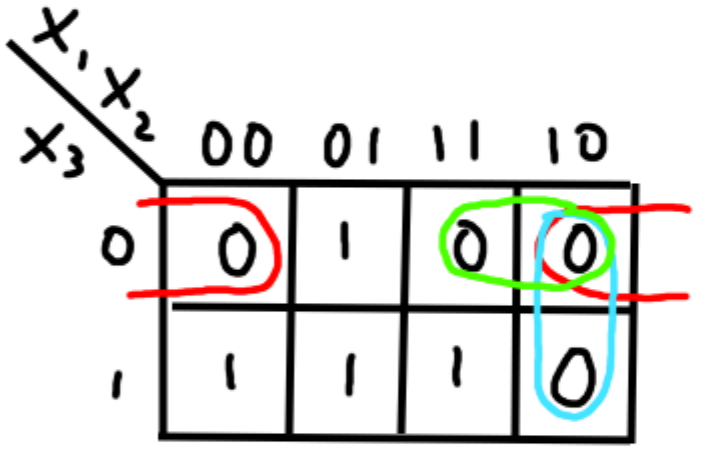
20)

p	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
q	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
r	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
s	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
?	0	0	0	1	1	0	1	1	0	0	0	0	1	0	1	0



In totale, escludendo quelli banali contenenti una sola variabile, ne abbiamo 7, contando anche quelli banali ne abbiamo 7+6 = 13

21)



x1	x2	x3	y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

C1: (!x1+x3) C2: (x2+x3)

C3: (!x1+x2)

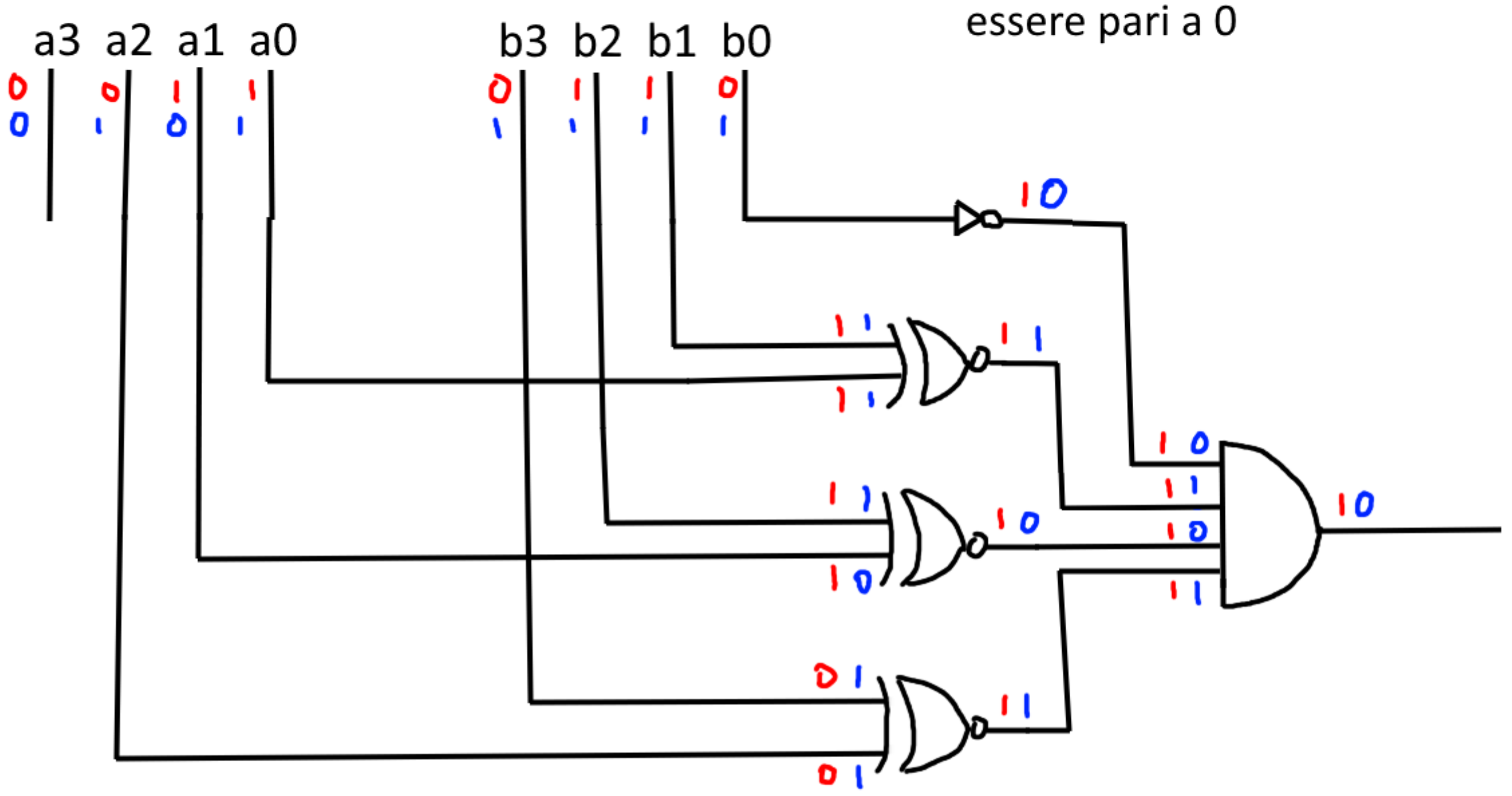
F = (!x1 + x3) (x2 + x3) (!x1 + x2)

x1	x2	x3	((¬x1 ∨ x3) ∧ ((x2 ∨ x3) ∧ (¬x1 ∨ x2)))
F	F	F	**F**
F	F	T	**T**
F	T	F	**T**
F	T	T	**T**
T	F	F	**F**
T	F	T	**F**
T	T	F	**F**
T	T	T	**T**

- Cerchiamo gli implicanti con gli 0
- Come sempre, prendiamo le variabili che non cambiano assegnazione negli implicanti.
- Neghiamo le variabili corrette
- Leghiamo le variabili delle clausole con il connettivo OR
- Leghiamo le clausole con l'AND

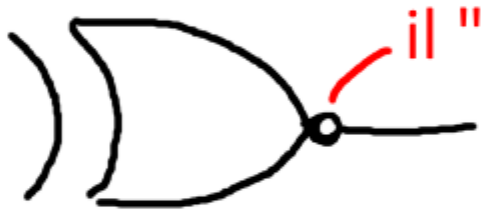
22) Moltiplicare per 2 in binario equivale a spostare verso sinistra ogni bit e aggiungere uno 0 al LSB, per esempio:
3_10 = 0011_2 6_10 = 0110_2

Il circuito potrebbe quindi fare la seguente cosa:
- Prendiamo i primi 3 bit del numero più piccolo e li confrontiamo con gli ultimi 3 bit del numero più grande, se coincidono, ritorniamo 1, altrimenti, ritorniamo 0
- Inoltre, il 1o bit di "b" dovrà essere pari a 0

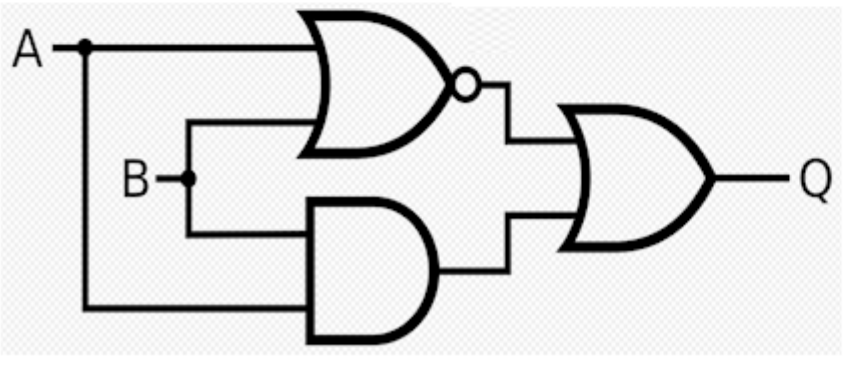


Per l'uguaglianza, ci serve la porta che implementi l'operazione di XNOR:

a	b	a XNOR b
0	0	1
0	1	0
1	0	0
1	1	1



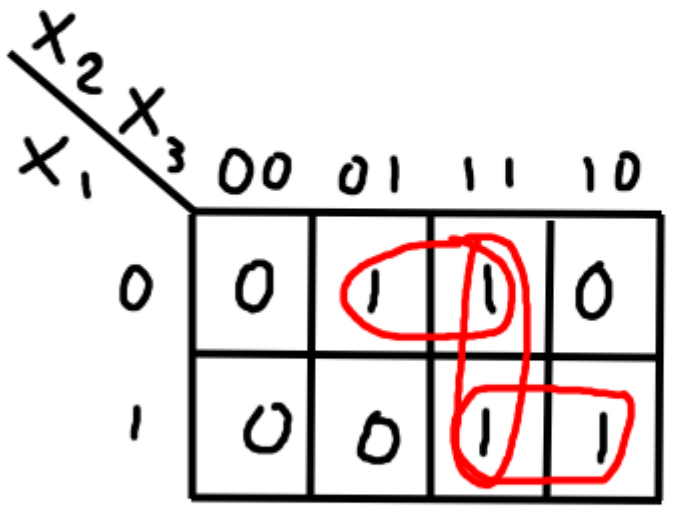
Piccola anticipazione, si può creare anche usando solo AND, OR & NOT



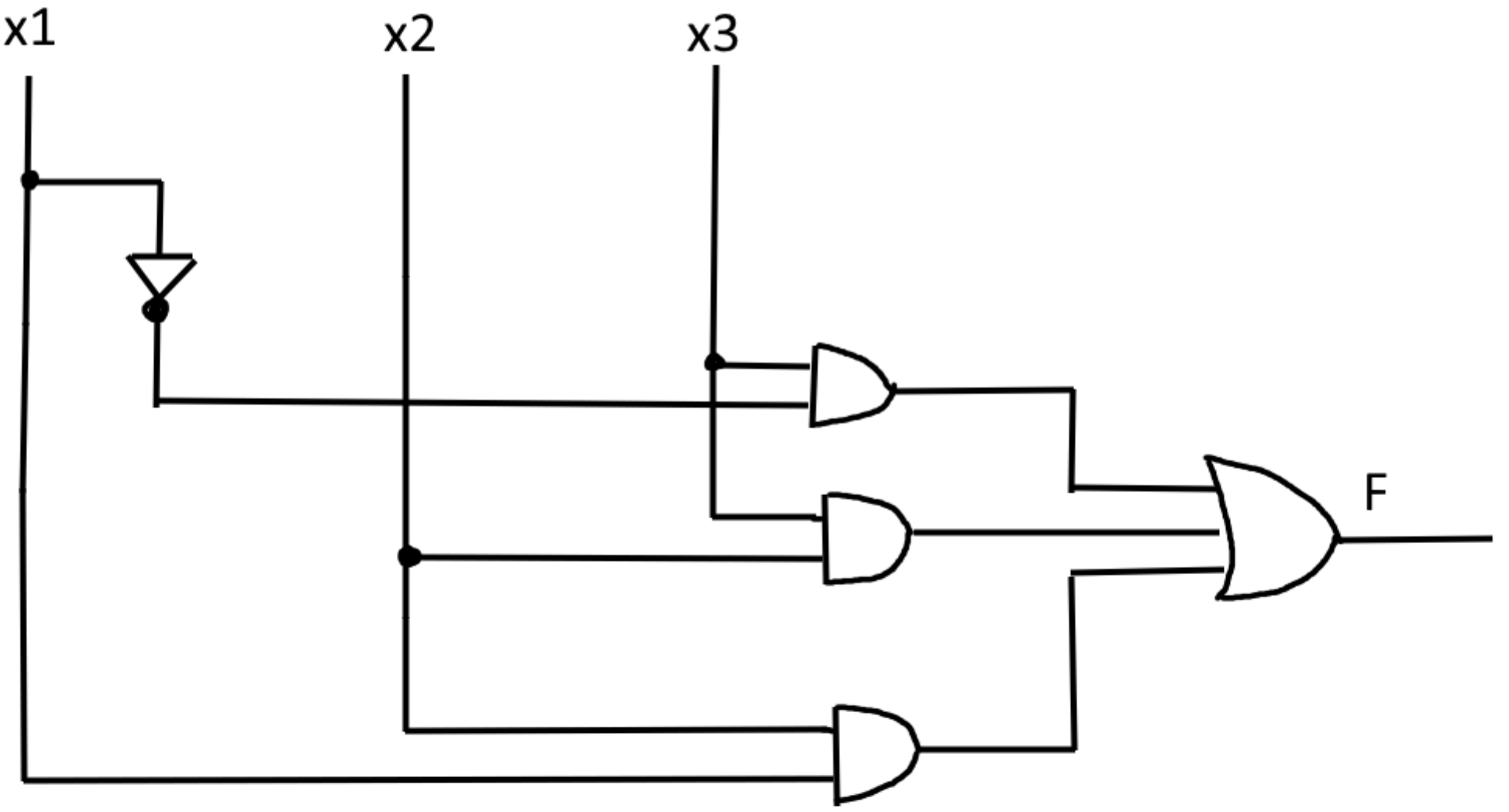
23)

INPUT: Tre bit, x_1, x_2, x_3.
OUTPUT: Un bit, y.
if $x_1 = 1$ **then**
    $y = x_2$
else
    $y = x_3$
return y

x1	x2	x3	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

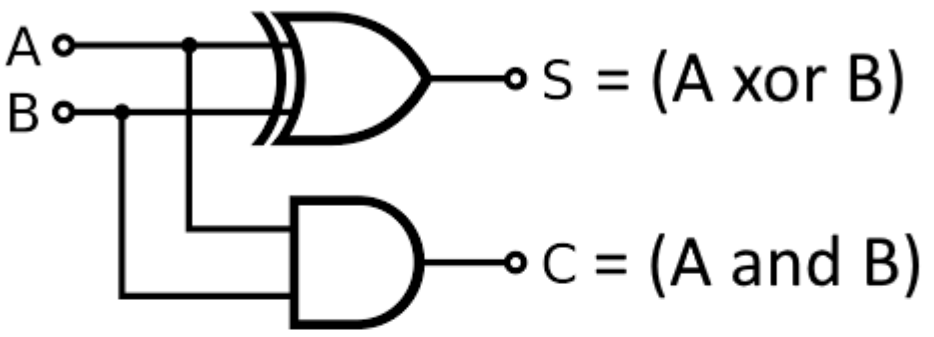


F = (!x1 * x3) + (x2 * x3) + (x1 * x2)



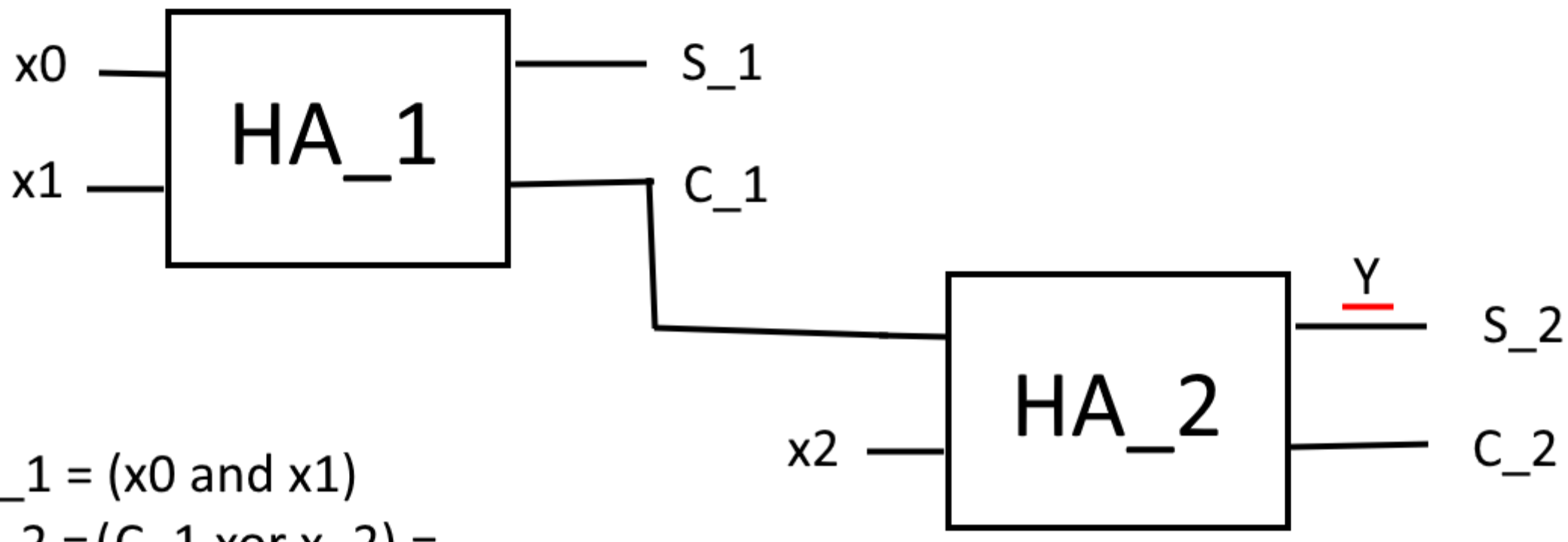
24)

$$y = \overline{x_0}x_2 + \overline{x_1}x_2 + x_0x_1\overline{x_2}$$



HALF ADDER

x0	x1	x2	((¬x0 ∧ x2) ∨ ((¬x1 ∧ x2) ∨ (x0 ∧ (x1 ∧ ¬x2))))
F	F	F	**F**
F	F	T	**T**
F	T	F	**F**
F	T	T	**T**
T	F	F	**F**
T	F	T	**T**
T	T	F	**T**
T	T	T	**F**



C_1 = (x0 and x1)

S_2 = (C_1 xor x_2) =

= ((x0 and x1) xor x2) = Y

x_0	x_1	x_2	$x_0 \wedge x_1$	$(x_0 \wedge x_1) \oplus x_2$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	0	1
1	0	0	0	0
1	0	1	0	1
1	1	0	1	1
1	1	1	1	0