Circuiti aritmetici

Somme, sottrazioni e così via

Circuiti logici e aritmetici

- **I circuiti visti fino ad ora realizzano funzioni logiche**
  - Scelte, priorità, codifiche
  - Gli operatori base sono operatori della *logica*, non dell'aritmetica
- **Numerazione binaria**
  - I valori logici di *vero* e *falso* vengono però spesso interpretati come cifre 1 e 0
  - Lo abbiamo fatto fino ad ora più o meno consapevolmente
  - Possiamo usare le cifre 1 e 0 per rappresentare numeri in base 2
  - I teoremi dell'algebra Booleana ci assicurano che gli operatori logici siano sufficienti per realizzare qualunque funzione su questa base
- **Circuiti aritmetici**
  - Tramite la codifica è quindi possibile realizzare circuiti che calcolano funzioni di tipo aritmetico

Numerazione posizionale

- **Numero rappresentato come somma di contributi a diversi ordini di grandezza secondo una convenzione posizionale**
  - Non tutte le numerazioni sono posizionali (e.g., i numeri Romani)
  - Ogni cifra corrisponde ad un ordine di grandezza
  - In base 10, ogni cifra corrisponde a 10 unità della cifra immediatamente meno significativa
  - Le cifre hanno valori da 0 a 9
  - Per esempio, 1950.43 è dato da
    - 1*1,000 + 9*100 + 5*10 + 0*1 + 4*0.1 + 3*0.01

a_5	a_4	a_3	a_2	a_1	a_0	.	a_{-1}	a_{-2}	a_{-3}
10^5	10^4	10^3	10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}
100,000	10,000	1,000	100	10	1		0.1	0.01	0.001

Numerazione binaria

- **In base 2 funziona uguale**
  - Ogni cifra corrisponde ad un ordine di grandezza (in questo caso una potenza di 2)
  - Ogni cifra corrisponde a 2 unità della cifra immediatamente meno significativa
  - Le cifre hanno valori da 0 a 1
  - Per esempio, 45.625 è dato da
    - 1*32 + 0*16 + 1*8 + 1*4 + 0*2 + 1*1 + 1*0.5 + 0*0.25 + 1*.125
    - = 101101.101

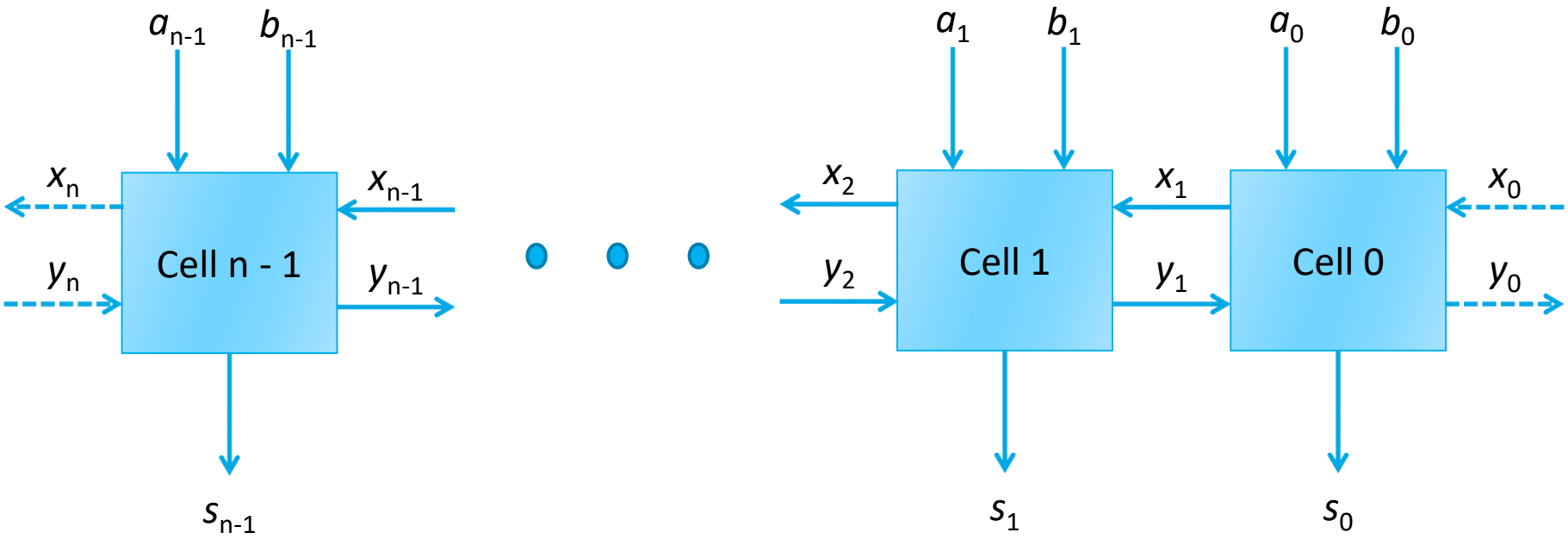
a_5	a_4	a_3	a_2	a_1	a_0	.	a_{-1}	a_{-2}	a_{-3}
2^5	2^4	2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}
32	16	8	4	2	1		0.5	0.25	0.125

Realizzazione

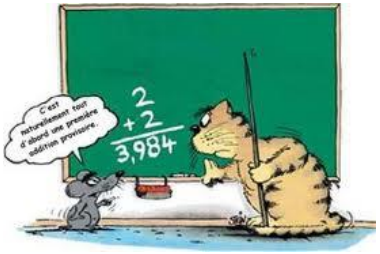
- **Nulla di diverso da quanto fatto fin'ora**
  - Si definisce una codifica dei numeri, per esempio binaria pura
  - Si scrive una funzione logica che trasforma la codifica di due numeri nella codifica, per esempio, della loro somma
  - Si deriva un circuito semplificando la funzione logica
- **Difficoltà**
  - Il numero di cifre cresce con la dimensione dei numeri (anche se non troppo velocemente)
  - Ben presto il numero di variabili di ingresso diventa ingestibile
  - Occorre usare il metodo gerarchico
- **Soluzione**
  - Si lavora una cifra alla volta (metodo iterativo)
  - Ogni cifra usa i risultati delle cifre vicine

Struttura iterativa generica

- **Composta da un numero di celle pari alle cifre**
  - Ogni cella esegue il conto per la cifra corrispondente
  - Il risultato finale è dato dai contributi di ogni cella



Somma



- **Partiamo dalla somma di due numeri a 1 bit ciascuno**
  - 0 + 0 = 0
  - 0 + 1 = 1
  - 1 + 0 = 1
  - 1 + 1 = 10
- **La somma di due numeri a 1 bit richiede 2 bit per la rappresentazione del risultato**
  - Si possono interpretare come un bit per la somma ed un bit per il riporto (carry)
  - Il carry può essere usato dalla eventuale cella successiva

Mezzo sommatore (half adder)

a	*b*	*c*	*s*
0	**0**	0	0
0	**1**	0	1
1	**0**	0	1
1	**1**	1	0

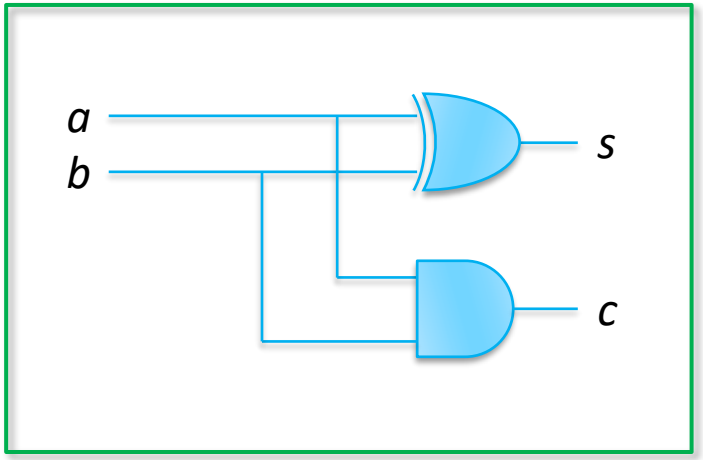
$$s = a'b + ab'$$
$$c = ab$$

a \ *b*	0	1
0	0	1
1	1	0

s

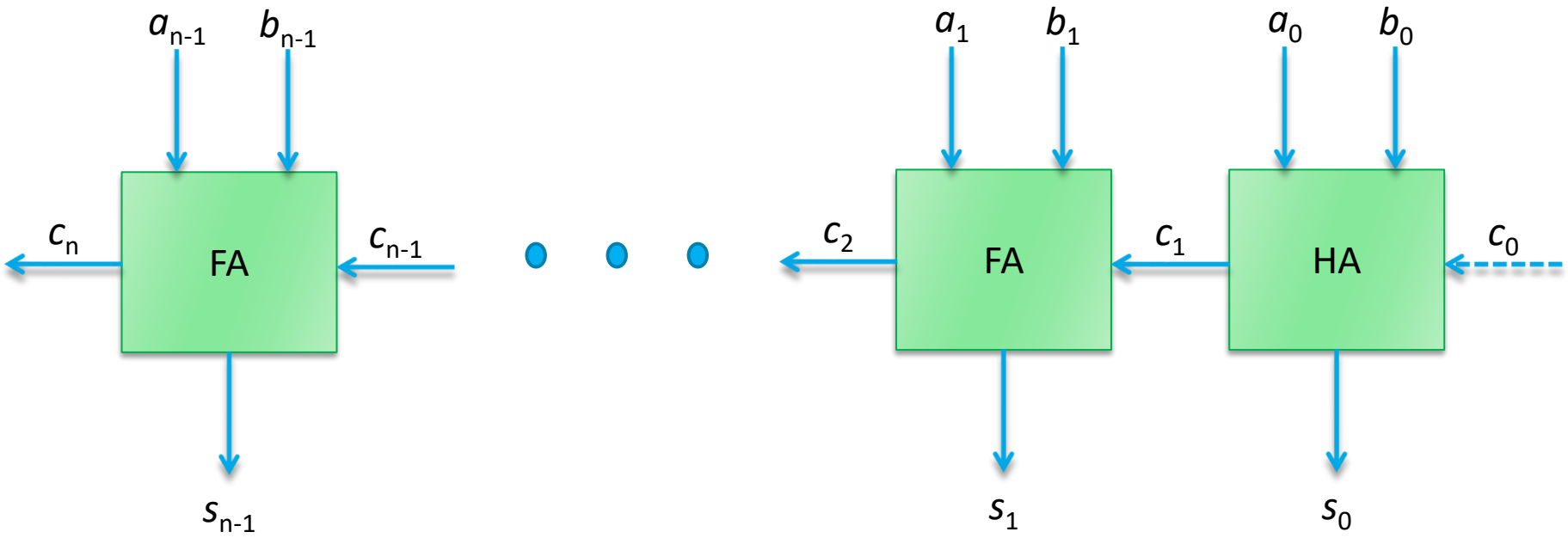
a \ *b*	0	1
0	0	0
1	0	1

c



Sommatore ripple carry

- **Dobbiamo sommare ogni coppia di bit assieme al riporto dalla cella precedente**
  - Ci occorre quindi un circuito più complesso, chiamato Full Adder
  - La prima cella può usare l'half adder, se non c'è bisogno di un riporto in ingresso
  - L'ultima cella fornisce un riporto finale



Full adder

- **Dobbiamo sommare 3 bit**
  - I due bit dei due numeri
  - Il riporto proveniente dalla cella precedente
  - 0 + 0 + 0 = 0
  - 0 + 0 + 1 = 1
  - 0 + 1 + 0 = 1
  - 0 + 1 + 1 = 0 con il riporto di 1
  - 1 + 0 + 0 = 1
  - 1 + 0 + 1 = 0 con il riporto di 1
  - 1 + 1 + 0 = 0 con il riporto di 1
  - 1 + 1 + 1 = 1 con il riporto di 1
- **Due uscite sono sufficienti**
  - Somma e riporto

c	a	b	c_{+1}	s
0	**0**	**0**	0	0
0	**0**	**1**	0	1
0	**1**	**0**	0	1
0	**1**	**1**	1	0
1	**0**	**0**	0	1
1	**0**	**1**	1	0
1	**1**	**0**	1	0
1	**1**	**1**	1	1

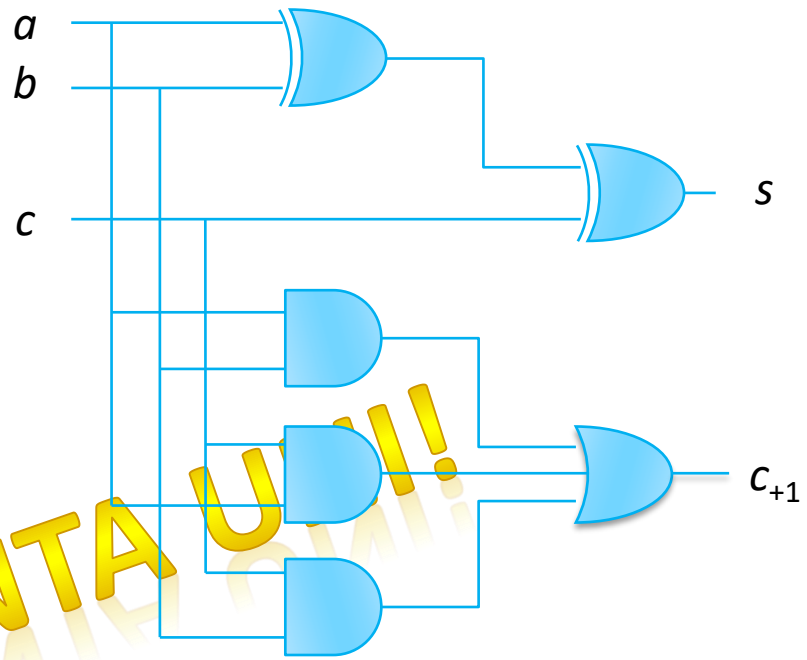
Realizzazione del full adder

$c \setminus ab$	00	01	11	10
0	0	1	0	1
1	1	0	1	0

s

$c \setminus ab$	00	01	11	10
0	0	0	1	0
1	0	1	1	1

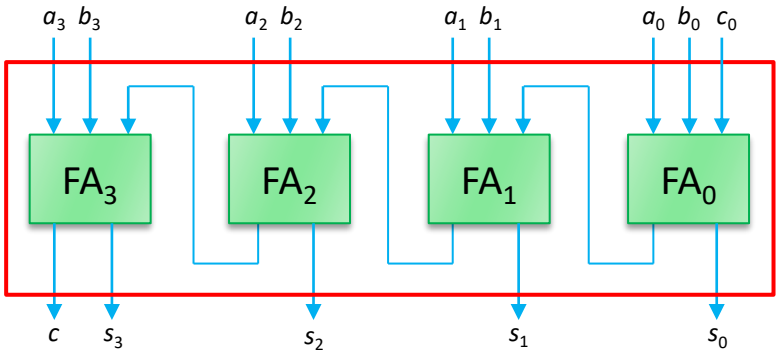
c_{+1}

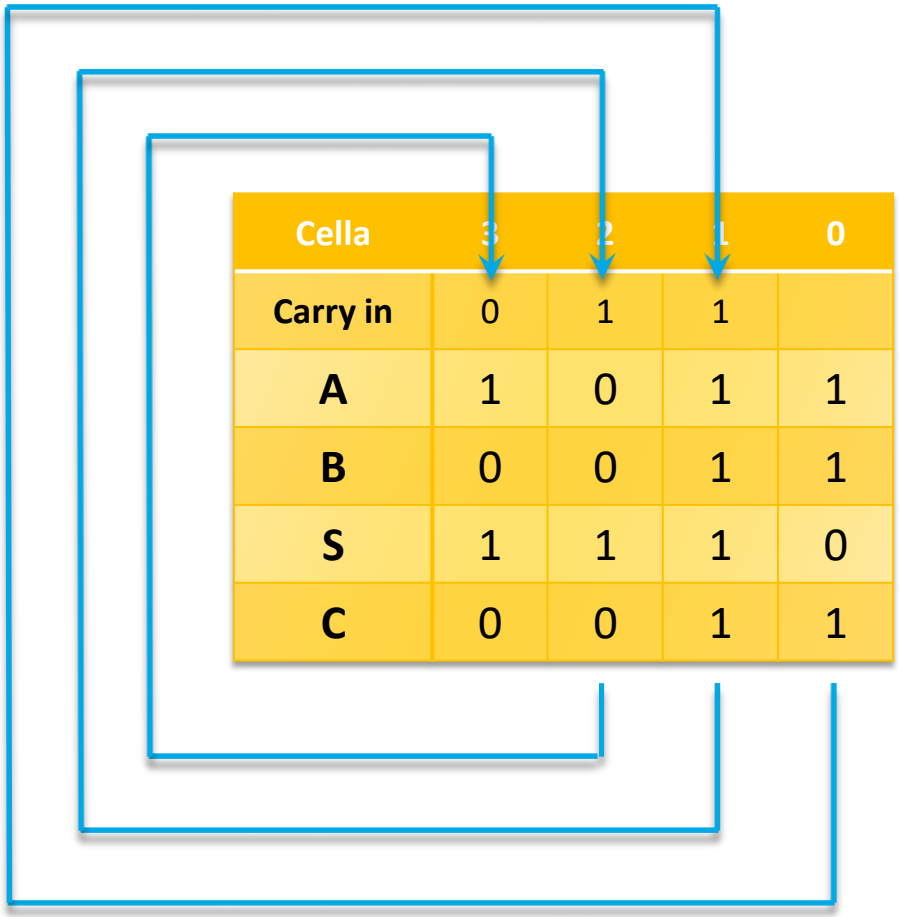


$$s = a'b'c + a'bc' + abc + ab'c'$$
$$= a \oplus b \oplus c$$
$$c_{+1} = ab + bc + ac$$

Sommatore ripple carry

- **E' come fare la somma in colonna**
  - Esempio
  - $11_{10} + 3_{10} = 14_{10}$
  - $1011_2 + 0011_2 = 1110_2$

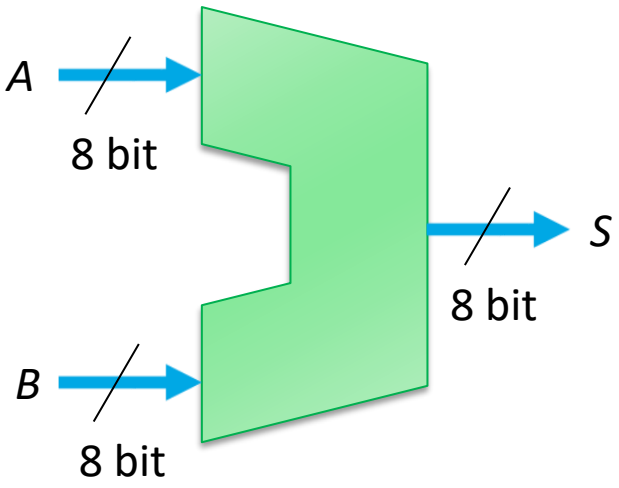




Cella	3	2	1	0
Carry in	0	1	1	
A	1	0	1	1
B	0	0	1	1
S	1	1	1	0
C	0	0	1	1

Osservazioni

- **Progetto ad alto numero di variabili di ingresso**
  - 8 solo per gli ingressi di dato
  - 9 se si include l'eventuale *carry in* per la prima cifra (512 caselle!!)
  - Il progetto iterativo gerarchico ci consente comunque di arrivare ad una soluzione in modo semplice
- **Facile realizzare sommatori grandi**
  - Per fare somme a 8 bit, basta mettere assieme due sommatori a 4 bit
  - Occorre collegare il *carry out* del primo al *carry in* del secondo
- **Overflow**
  - L'eventuale *carry out* finale a 1 indica una condizione di overflow
  - I bit di risultato non sono sufficienti per rappresentare la somma



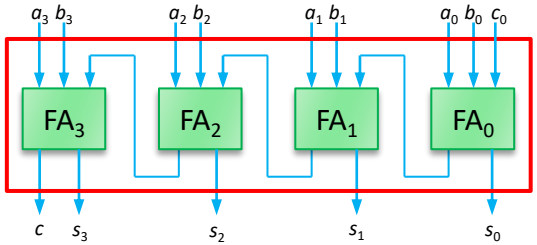
Esempio di overflow a 4 bit

- **A 4 bit rappresentiamo numeri da 0 a 15**
  - $12_{10} + 7_{10} = 19_{10}$
  - $1100_2 + 0111_2 = 10011_2$
  - $1100_2 + 0111_2 = 0011_2$ con riporto 1
  - Il risultato vale 3, il riporto vale 16 (2^4)

Cella	3	2	1	0
Carry in	1	0	0	
A	1	1	0	0
B	0	1	1	1
S	0	0	1	1
C	1	1	0	0

8	4	2	1	*M*
1	**1**	**1**	**1**	15
1	**1**	**1**	**0**	14
1	**1**	**0**	**1**	13
1	**1**	**0**	**0**	12
1	**0**	**1**	**1**	11
1	**0**	**1**	**0**	10
1	**0**	**0**	**1**	9
1	**0**	**0**	**0**	8
0	**1**	**1**	**1**	7
0	**1**	**1**	**0**	6
0	**1**	**0**	**1**	5
0	**1**	**0**	**0**	4
0	**0**	**1**	**1**	3
0	**0**	**1**	**0**	2
0	**0**	**0**	**1**	1
0	**0**	**0**	**0**	0

Prestazioni



- **Dimensioni**
  - Il sommatore ha dimensioni alquanto ridotte
  - Se si aggiunge un bit ai numeri di ingresso, le dimensioni aumentano di un solo stadio
  - La dimensione del sommatore ripple carry cresce linearmente con il numero di bit
  - Si può dimostrare che una soluzione a due livelli crescerebbe in modo esponenziale con il numero di bit
    - Perché la mappa della somma esce fuori sempre a scacchiera, quindi gli implicanti sono minterm
- **Tempistiche**
  - La catena dei carry forma un cammino molto lungo, quindi il sommatore è lento
  - E' tanto più lento quanto più alto è il numero di bit
  - Ovviamente, i bit di uscita cambiano in istanti differenti a causa della propagazione del carry
  - Alee impossibili da eliminare

Soluzioni alternative

- **Il sommatore è uno dei circuiti più diffusi**
  - Esistono letteralmente decine di diversi modi di realizzarlo
  - Ogni soluzione è un diverso compromesso tra velocità, dimensioni e scalabilità
  - Il sommatore a ripple carry è il più semplice, il più lento ed il più piccolo, con facile scalabilità
- **Rete di calcolo dei riporti**
  - Le soluzioni alternative normalmente agiscono sulla rete di calcolo del carry
  - Si cerca di anticipare il calcolo, in modo da rendere il sommatore più veloce
  - Questo porta ad un incremento delle dimensioni
- **Codifiche diverse**
  - Usando codifiche alternative (ridondanti) si può addirittura eliminare il riporto
  - Utili in situazioni che possono fare uso di tali codifiche

Sommatore carry look-ahead

- **Il ritardo sulla rete del ripple carry è di circa 2*n* + 2 porte logiche, per sommatori a *n* bit**
  - Facilmente può diventare un ritardo significativo in un sistema
  - Per 16 bit sono circa 34 porte
- **Conveniente quindi cercare di ottimizzare il calcolo del carry**
  - La realizzazione sarà più complessa
  - Riduciamo a 2 livelli di logica il calcolo del carry su gruppi di bit
- **Per vedere meglio, cerchiamo di separare la somma dal carry**
  - La somma la lasceremo invariata

Realizzazione con due half adder

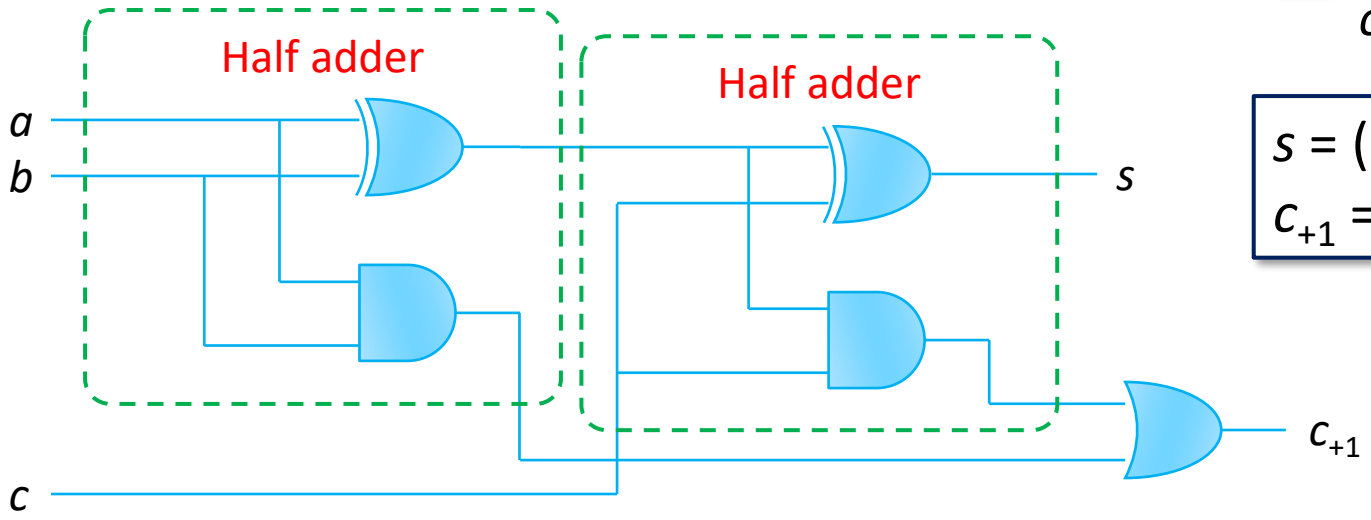
- **Manipolando l'espressione**
  - Abbiamo già il termine $(a \oplus b)$ nel circuito
  - $c_{+1} = ab + ac + bc$
  - $= ab + ab'c + a'bc$
  - $= ab + c(ab' + a'b)$
  - $= ab + c(a \oplus b)$

$c \setminus ab$	00	01	11	10
0	0	1	0	1
1	0	1	0	1

$(a \oplus b)$

$c \setminus ab$	00	01	11	10
0	0	0	1	0
1	0	1	1	1

c_{+1}

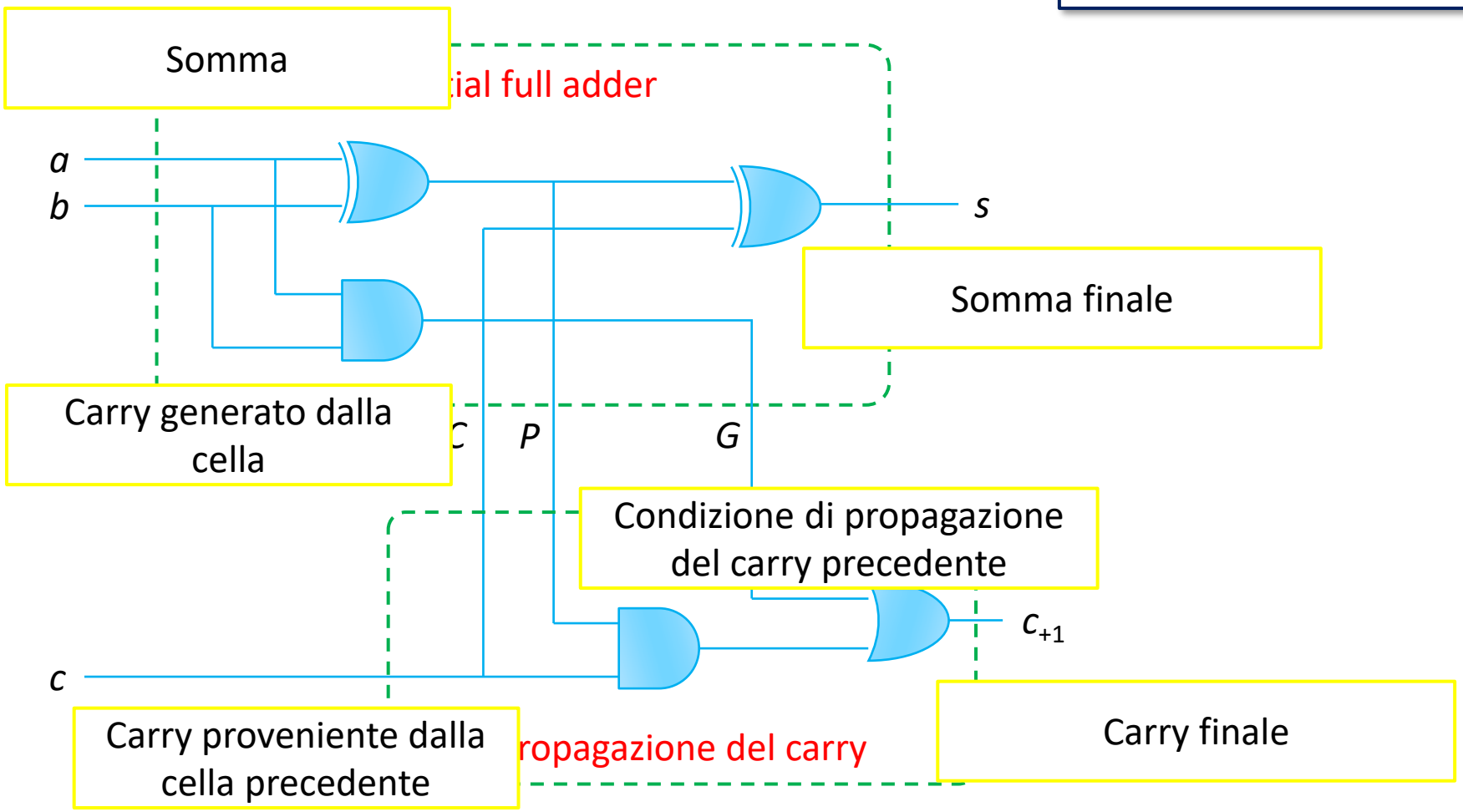


$s = (a \oplus b) \oplus c$

$c_{+1} = ab + c(a \oplus b)$

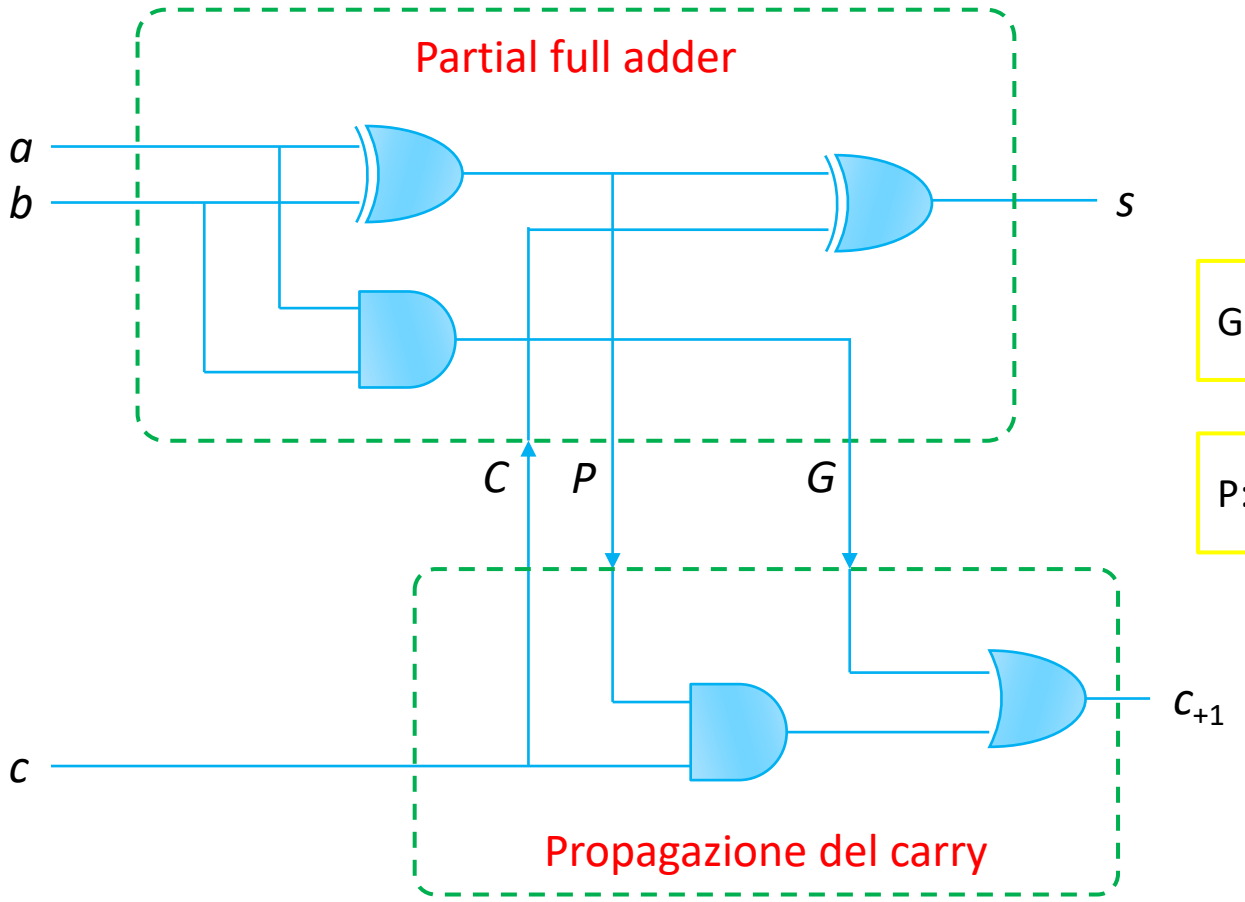
Generazione e propagazione del carry

$c_{+1} = ab + c(a \oplus b)$



Generazione e propagazione del carry

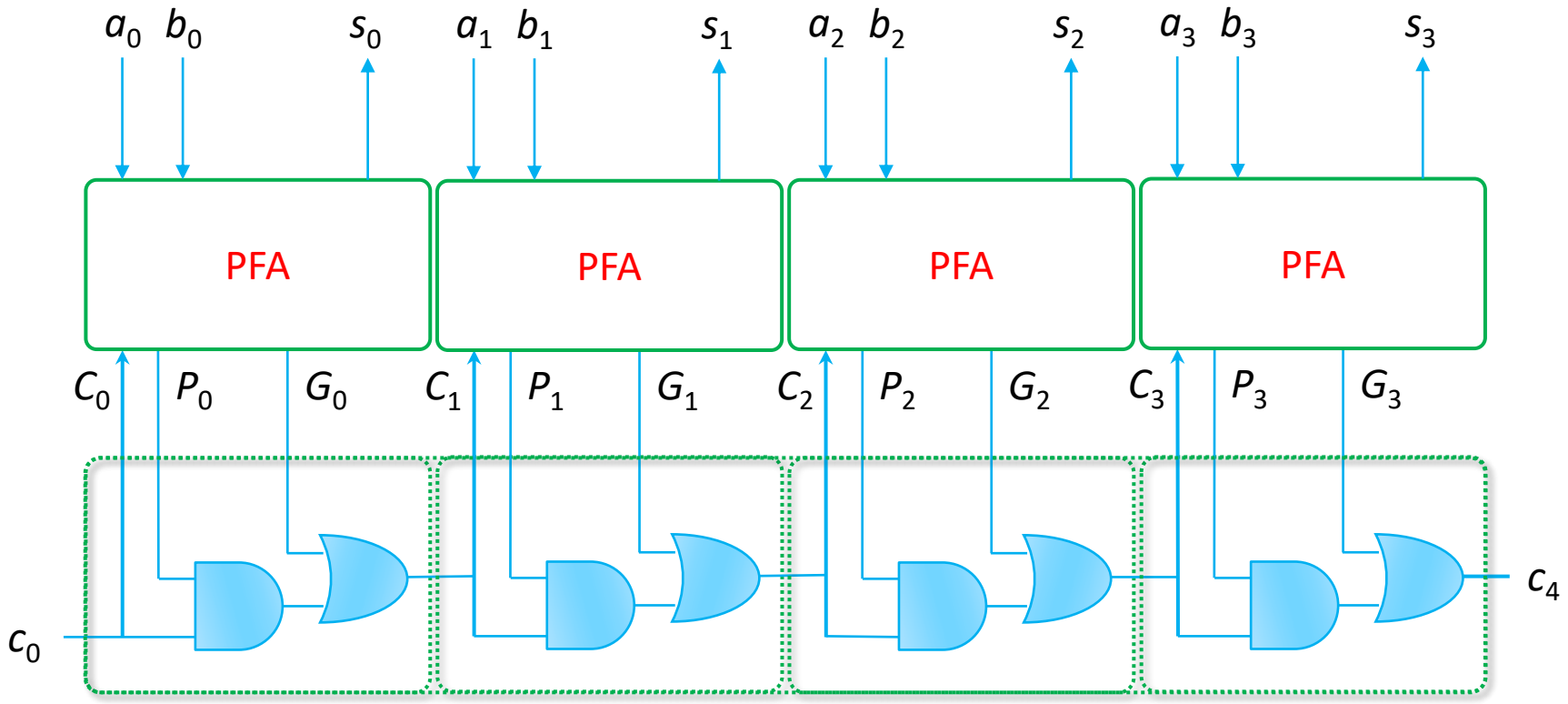
$$c_{+1} = ab + c(a \oplus b)$$



G: Carry generato dalla cella

P: Condizione di propagazione

Ripple carry (di nuovo!)



$C_1 = G_0 + P_0C_0$

$C_2 = G_1 + P_1C_1$
$= G_1 + P_1 (G_0 + P_0C_0)$
$= G_1 + P_1G_0 + P_1P_0C_0$

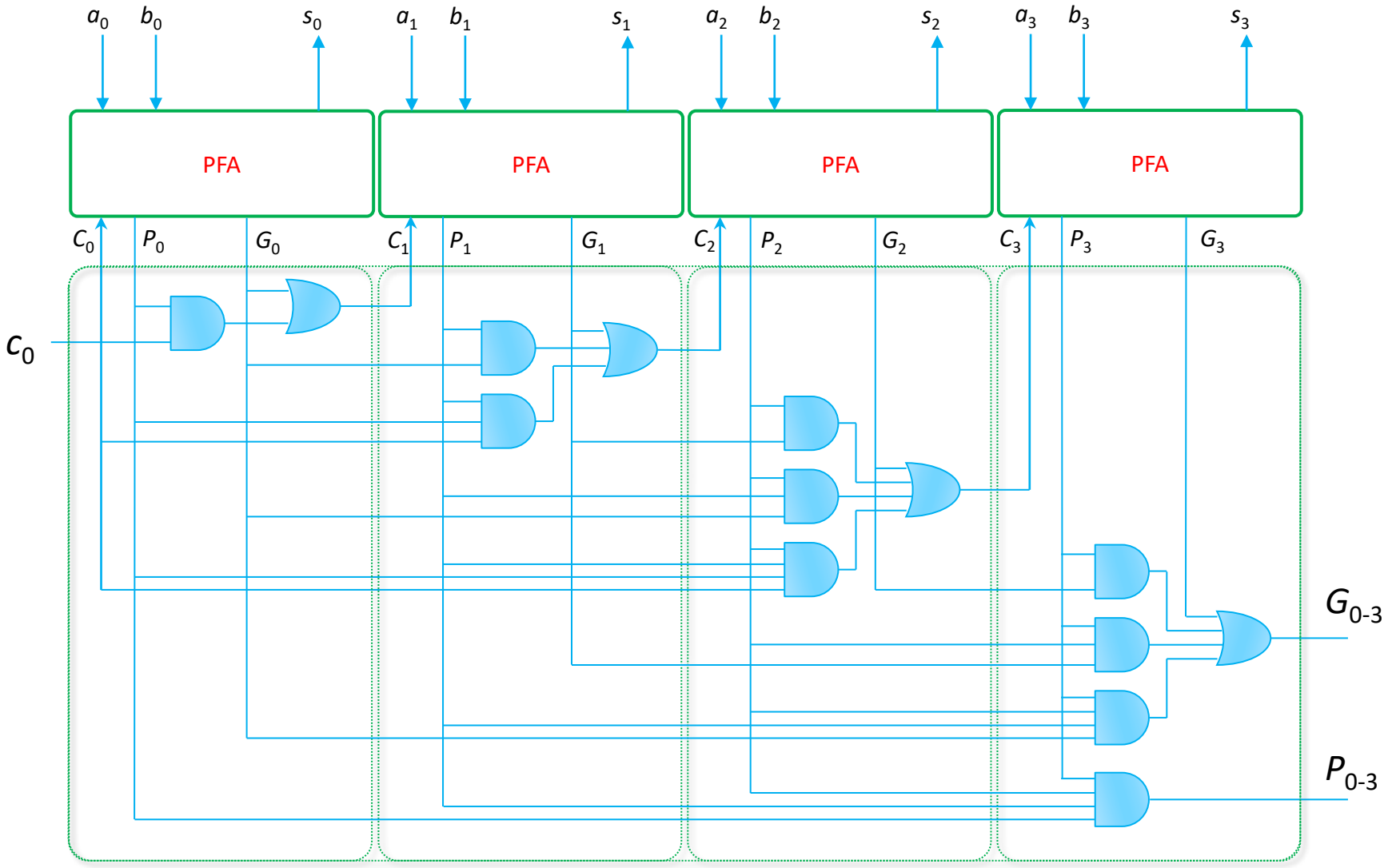
$C_3 = G_2 + P_2C_2$
$= G_2 + P_2(G_1 + P_1G_0 + P_1P_0C_0)$
$= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$

- **Cammino critico lungo**
  - Possiamo ottimizzare la propagazione ricalcolando le uscite di ogni stadio e realizzandole a due livelli

Ultimo carry

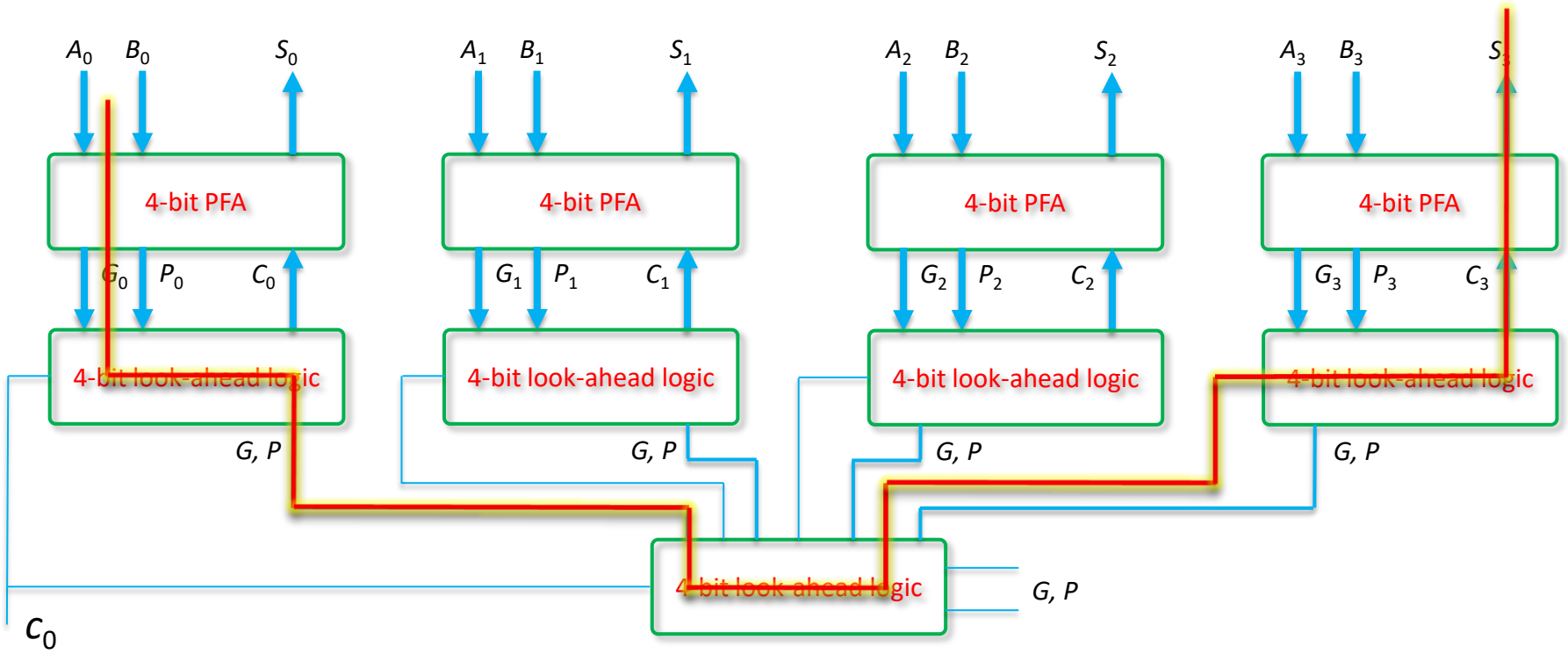
- **Per l'ultimo carry, si può usare lo stesso metodo**
  - Le porte hanno però un fan-in sempre maggiore, aumentando i ritardi
- **E' possibile definire per l'ultima cifra due uscite di generazione $G_{0\text{-}3}$ e propagazione $P_{0\text{-}3}$ del carry**
  - Preserviamo la struttura del PFA
  - In questo modo si può usare il blocco sommatore a 4 bit come elemento base di sommatori gerarchici carry look-ahead
- **Propagazione**
  - Per propagare da C_0 a C_4 occorre avere tutte le propagazioni attive
  - $P_{0\text{-}3} = P_3P_2P_1P_0$
- **Generazione**
  - Un carry generato deve potersi propagare fino al fondo
  - $G_{0\text{-}3} = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$

Sommatore carry look-ahead



Osservazioni

- **Ritardo ridotto da 10 livelli a 6 livelli di logica**
  - Includendo 2 livelli per generare l'ultimo carry o generare l'uscita
- **Possiamo costruire sommatori più grossi**
  - Per fare un sommatore a 16 bit usiamo 4 sommatori a 4 bit, combinati di nuovo con la rete di calcolo del carry
  - Il ritardo in questo caso scende da 34 a 10 livelli di logica



Sottrazione

- **Apparentemente come la somma**
  - Si può realizzare il sottrattore a 1 bit con prestito
  - Si combinano poi con una rete di calcolo del prestito
- **Ma ora il risultato può essere negativo**
  - Occorre una rappresentazione
  - Per esempio si può aggiungere un bit per indicare il segno (modulo e segno)

Pres	0	0	0	0
22	1	0	1	1	0
18	1	0	0	1	0
4	0	0	1	0	0

Pres	0	0	1	1
22	1	0	1	1	0
19	1	0	0	1	1
3	0	0	0	1	1

Pres					
19	1	0	0	1	1
30	1	1	1	1	0
			?		

Pres	0	0	1	1
30	1	1	1	1	0
19	1	0	0	1	1
-11	0	1	0	1	1

Codifica in complemento a 2

- **Numeri negativi codificati in complemento a 2**
  - Compl. a 2 di $M = 2^n - M$
  - Notare che il complemento a 2 del complemento a 2 di M è di nuovo M
  - Un bit indica se il numero è positivo o negativo

- **Risultato della somma**
  - Se sommo due numeri positivi è come prima
  - Se sommo un positivo con un negativo (quindi con il suo complemento a 2) ottengo la differenza!
    - $M + (-N) = M + (2^n - N) = M - N + 2^n = M - N$
  - E se dovesse essere negativa la lascio in complemento a 2
  - Se sommo due negativi ottengo di nuovo il risultato corretto (nella codifica in complemento a 2)

-8	4	2	1	*M*
0	**1**	**1**	**1**	7
0	**1**	**1**	**0**	6
0	**1**	**0**	**1**	5
0	**1**	**0**	**0**	4
0	**0**	**1**	**1**	3
0	**0**	**1**	**0**	2
0	**0**	**0**	**1**	1
0	**0**	**0**	**0**	0
1	**1**	**1**	**1**	-1
1	**1**	**1**	**0**	-2
1	**1**	**0**	**1**	-3
1	**1**	**0**	**0**	-4
1	**0**	**1**	**1**	-5
1	**0**	**1**	**0**	-6
1	**0**	**0**	**1**	-7
1	**0**	**0**	**0**	-8

Codifica in complemento a 2

- **Vantaggi**
  - C'è una sola rappresentazione per lo 0
  - C'è comunque il bit di segno
  - Per fare la sottrazione si può usare un sommatore con il complemento a 2 del sottraendo
  - Non c'è quindi bisogno di un sottrattore dedicato
- **Overflow**
  - Quando la somma di due positivi è negativa
  - Quando la somma di due negativi è positiva
  - In pratica questo si verifica quando gli ultimi due riporti sono diversi
    - Non quando l'ultimo riporto è a 1
- **Range**
  - Per n bit si può rappresentare l'intervallo $[-2^{n-1}, 2^{n-1}-1]$
  - $n = 4$ [-8, 7]
  - $n = 5$ [-16, 15]
  - $n = 8$ [-128, 127]
  - $n = 32$ [-2.147.483.648, 2.147.483.647]

-8	4	2	1	*M*
0	**1**	**1**	**1**	7
0	**1**	**1**	**0**	6
0	**1**	**0**	**1**	5
0	**1**	**0**	**0**	4
0	**0**	**1**	**1**	3
0	**0**	**1**	**0**	2
0	**0**	**0**	**1**	1
0	**0**	**0**	**0**	0
1	**1**	**1**	**1**	-1
1	**1**	**1**	**0**	-2
1	**1**	**0**	**1**	-3
1	**1**	**0**	**0**	-4
1	**0**	**1**	**1**	-5
1	**0**	**1**	**0**	-6
1	**0**	**0**	**1**	-7
1	**0**	**0**	**0**	-8

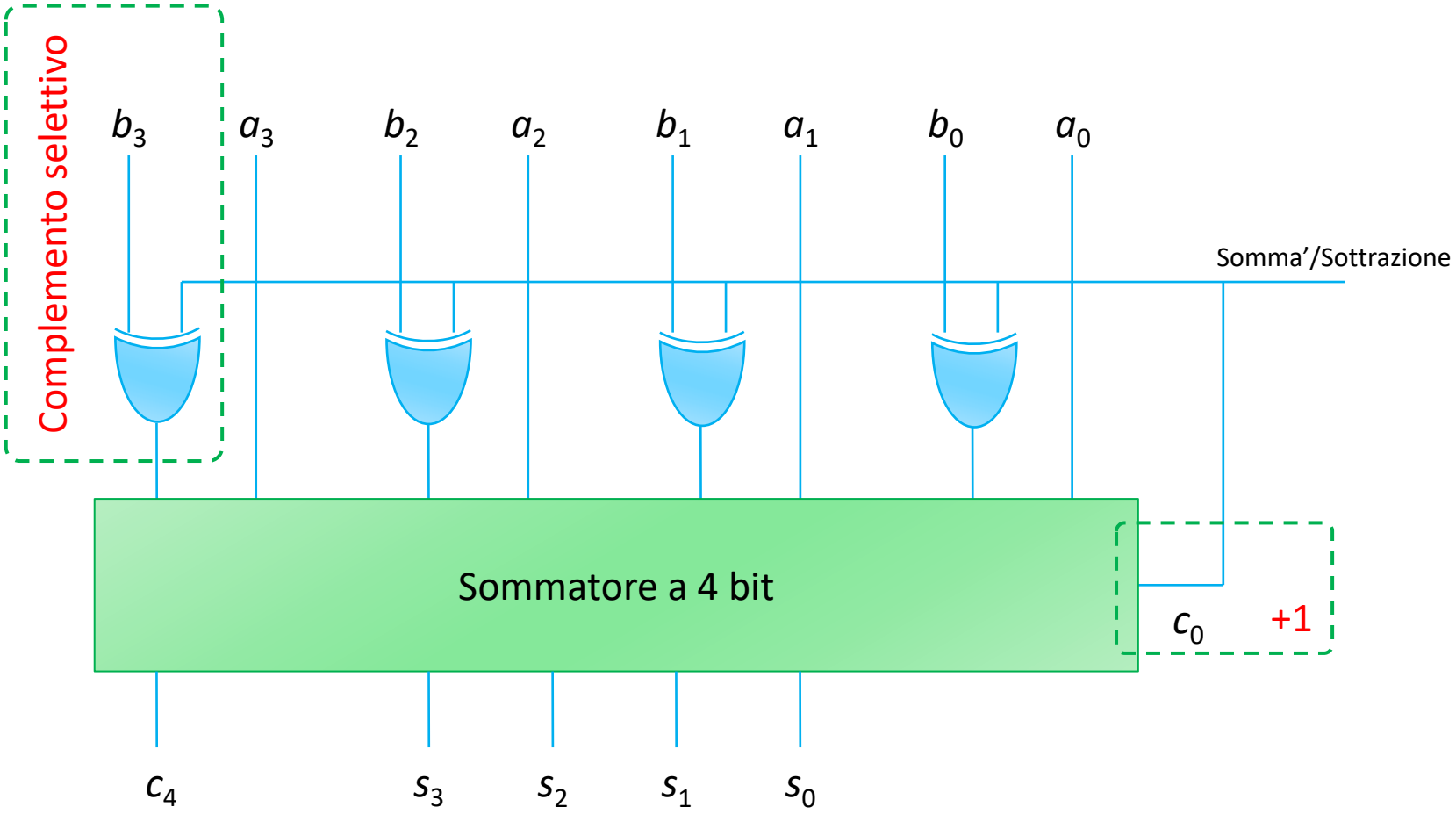
Calcolo del complemento a 2

- **Scomponiamo ($2^n - M$) come**
  - $(2^n - 1 + 1) - M$
  - $(2^n - 1 - M) + 1$
- **Il termine ($2^n - 1 - M$) si chiama *complemento a 1* di *M***
  - Lo si ottiene facilmente
  - Osservate infatti che $2^n - 1$ sono tutti 1
  - Quindi il complemento a 1 di *M* si ottiene semplicemente invertendo tutti i bit di *M* (1 – 0 = 1, 1 – 1 = 0)
  - Infine, per avere il complemento a 2, basta sommare 1

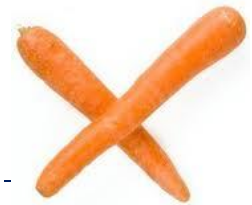
$2^5 = 100000$
$2^5 - 1 = 11111$

M	binario	$2^n - 1 - M$	$2^n - 1 - M + 1$	compl. a 2
5	0101	1010	1011	-5
7	0111	1000	1001	-7
12	01100	10011	10100	-12

Sommatore / sottrattore



Moltiplicatori

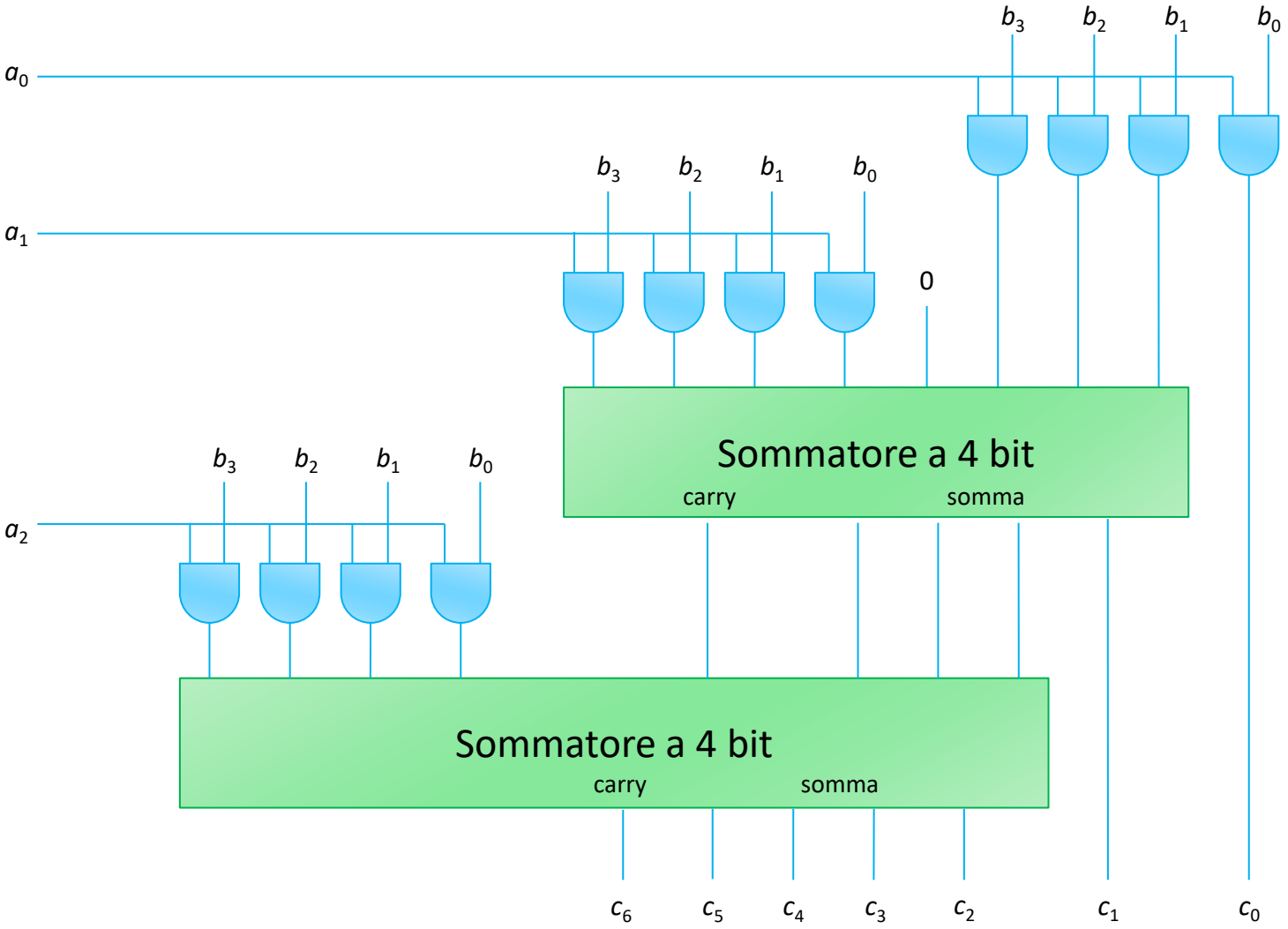


- **Seguiamo la stessa logica e li facciamo in colonna**
  - 0 x 0 = 0
  - 0 x 1 = 0
  - 1 x 0 = 0
  - 1 x 1 = 1
  - Il moltiplicatore a 1 bit è una AND
- **Multi-bit**
  - Occorre traslare i risultati intermedi
  - E quindi sommare tutti i contributi
  - Con *n* bit, il risultato può avere 2*n* bit

11			1	0	1	1
5			0	1	0	1
			1	0	1	1
		0	0	0	0	
	1	0	1	1		
55	1	1	0	1	1	1

Moltiplicatore 4 x 3

11			1	0	1	1
5			0	1	0	1
			1	0	1	1
		0	0	0	0	
	1	0	1	1		
55	1	1	0	1	1	1



Altre operazioni

- **Si possono realizzare molti altri operatori**
  - Divisione
  - Radice quadrata
  - Comparatori
  - Rotazioni e traslazioni
  - Funzioni trigonometriche
- **Numeri con la virgola**
  - Si possono trattare numeri con la virgola
  - In virgola fissa, usando la codifica binaria
  - In virgola mobile, distinguendo mantissa ed esponente
  - Che vantaggi hanno le due rappresentazioni?

Comparatori

- **Una uscita che indica se A è maggiore o uguale a B**
  - Si può fare a due livelli
  - Oppure usare il metodo iterativo e fare confronti una cifra alla volta
  - Le cifre più significative trasferiscono una sorta di riporto a quelle meno significative
  - Da vedere a esercitazione

Take away

- **Tramite una opportuna codifica dei numeri, i circuiti logici possono eseguire operazioni aritmetiche**
  - Abbiamo visto appena una minima parte dei circuiti possibili
  - Si potrebbe fare un intero corso apposta
  - Molteplici soluzioni architetturali
- **La codifica ha forte influenza sulla realizzazione**
  - La codifica in complemento a 2 è oggi la più usata
  - Da valutare attentamente i compromessi tra virgola fissa e virgola mobile
  - Codifiche ridondanti possono portare vantaggi in prestazioni, a scapito alle volte delle dimensioni
  - Ha senso realizzare circuiti dedicati, visto l'enorme campo applicativo dei circuiti aritmetici