Circuiti Aritmetici

▼ Creatore originale: @Samuele Gentile

Ripasso di numerazione binaria

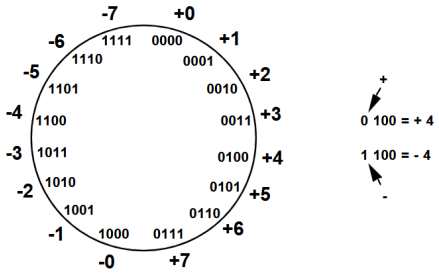
Modulo e segno

Nella rappresentazione Modulo e Segno (M&S), il bit più significativo è il segno, con 0 che indica il positivo e 1 che indica il negativo.

I restanti bit indicano l'effettivo valore numerico, chiamato modulo.

In questa rappresentazione:

- lo 0 è rappresentato con due valori binari diversi;
- somma e sottrazione sono complesse perché bisogna confrontare i moduli per decidere il segno del risultato.



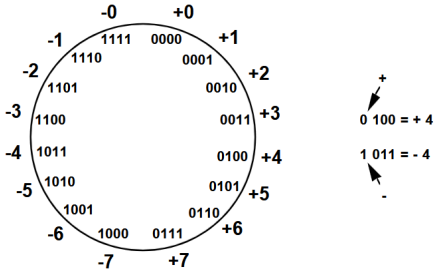
Rappresentazione dei valori M&S a 4 bit

Complemento a 1 (CA1)

Nella rappresentazione a complemento a 1 (CA1), si cambiano tutti i valori al loro complemento (0 → 1, 1 → 0).

In questa rappresentazione:

- lo 0 è rappresentato con due valori binari diversi;
- se abbiamo un numero positivo, il suo opposto sarà ottenuto tramite il complemento ad 1, e viceversa;
- la sottrazione è semplice da implementare, poiché prima faccio il complemento a 1 del sottraendo, e poi sommo;
- la somma è più complessa per i casi in cui bisogna gestire l'overflow.



Rappresentazione dei valori CA1 a 4 bit

Complemento a 2 (CA2)

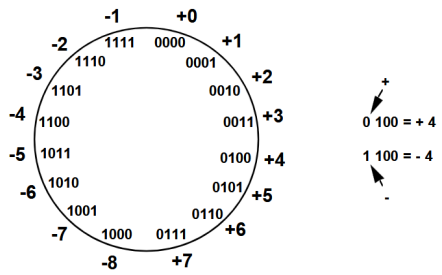
Nella rappresentazione a complemento a 2 (CA2), si esegue il CA1, per poi sommare 1 in binario.

Per esempio, trasformiamo in CA2 0111_2:

$$0111_{\text{CA1}} + 1_2 = 1000_{\text{CA1}} + 1_2 = 1001_{\text{CA2}}$$

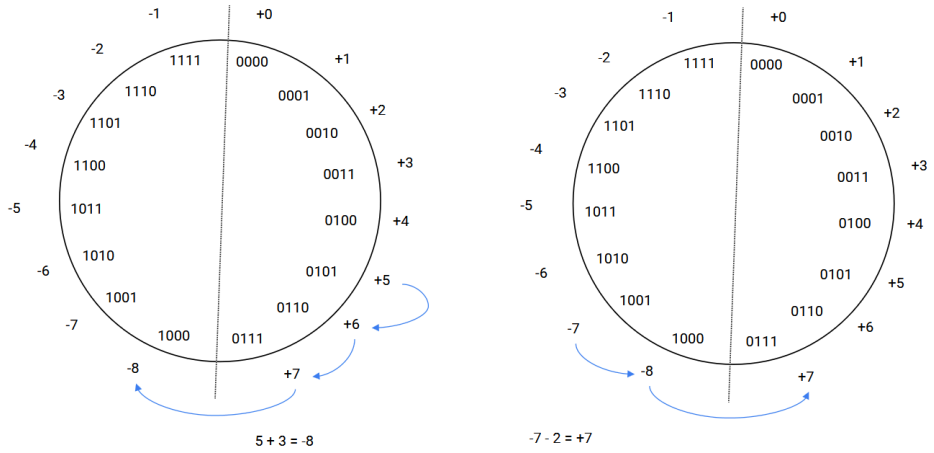
In questa rappresentazione:

- lo 0 è rappresentato con un unico valore binario;
- rispetto alle altre rappresentazioni, il range di numeri che copriamo con gli stessi bit è aumentato di uno, poiché abbiamo una sola rappresentazione per 0;
- la condizione di overflow si verifica quando, sommando due numeri entrambi positivi o entrambi negativi, otteniamo un numero di segno opposto rispetto agli addendi.



Rappresentazione dei valori CA2 a 4 bit

- esiste un metodo abbastanza semplice per verificare se è avvenuto overflow, ovvero controllare che il riporto sul segno e il riporto in uscita siano uguali.



Esempi di overflow

	0111				1000		
5	0101	-7	1001	5	0101	-3	1101
3	0011	-2	1110	2	0010	-5	1011
-8	1000	7	10111	7	0111	-8	11000
Overflow		Overflow		Non è overflow		Non è overflow	

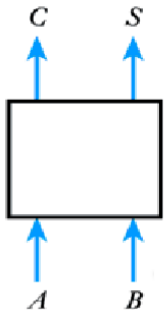
Esempi di calcolo di overflow e di non overflow

Half Adder (semisommatore)

Nel circuito Half Adder (HA), definiamo A e B come i due bit che vogliamo sommare, S come il bit di somma e C come il bit di riporto.

Si noti che:

- S è realizzabile con una porta XOR;
- C è realizzabile con una porta AND.



A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Visualizzazione di un circuito Half Adder

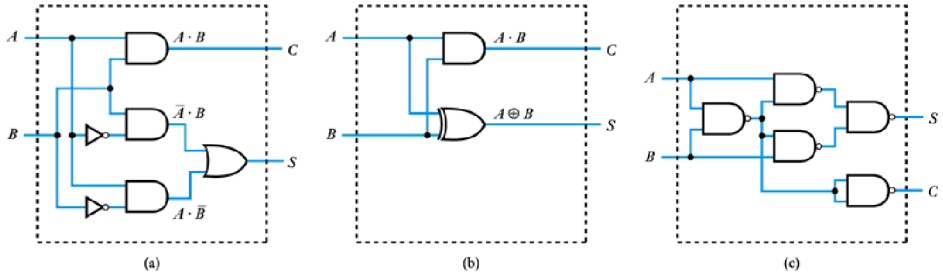
Possibili implementazioni

Di seguito sono riportate alcune implementazioni possibili:

a. si realizza con porte AND, OR e NOT;

b. si realizza con una porta AND e XOR;

c. si realizza solo con porte NAND.



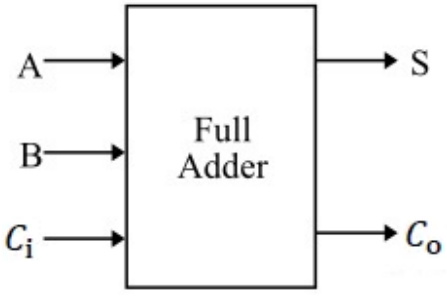
Possibili implementazioni di un circuito Half Adder

Full Adder (sommatore)

Nel circuito Full Adder (FA), definiamo A e B come i due bit che vogliamo sommare, S come il bit di somma, C_i come il riporto in ingresso (carry-in) e C_o come il riporto in uscita (carry-out).

$$S = A \oplus B \oplus C_i$$

$$\begin{aligned} C_o &= A \cdot B + A \cdot C_i + B \cdot C_i \\ &= A \cdot B + C_i \cdot (A + B) \end{aligned}$$



Visualizzazione di un circuito Full Adder

Si noti, quindi, che un generico sommatore deve gestire in ingresso anche un riporto proveniente dal sommatore precedente.

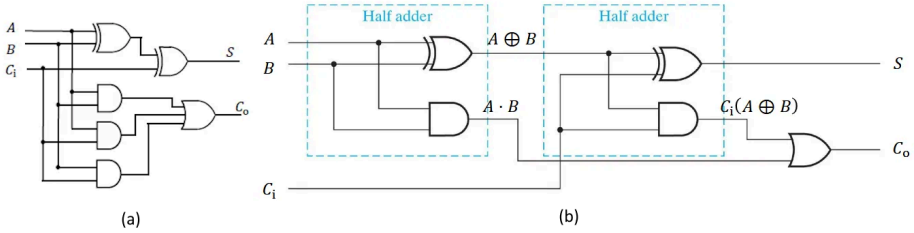
Possibili implementazioni

Di seguito sono riportate alcune implementazioni possibili:

a. si realizza con porte AND, OR e XOR;

b. cascata di due Half Adder, in cui notiamo che l'unica differenza è in C_o, che adesso ha uno XOR $A \oplus B$ e non un semplice OR. Non è una modifica che possiamo fare sempre, ma in questo caso specifico sì.

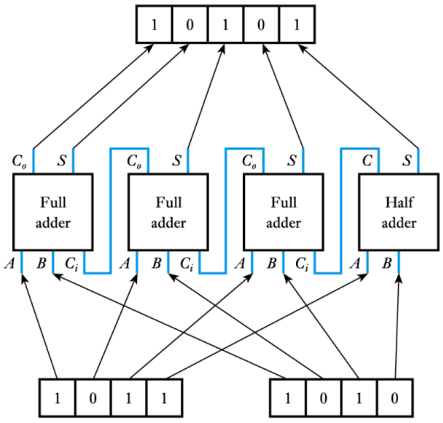
   Il caso in cui lo XOR si comporta diversamente dall'OR è quando A e B sono entrambi al valore 1, ma in quel caso il loro AND sommato a qualsiasi valore binario restituisce sempre 1.



Possibili implementazioni di una rete Half Adder

Adder Ripple Carry

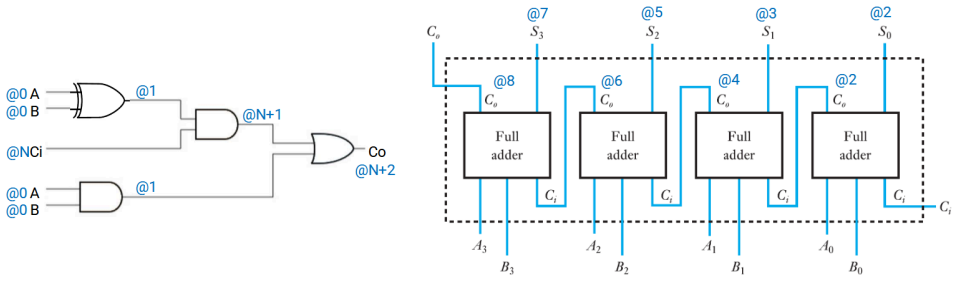
Nel sommatore multibit, chiamato più comunemente Adder Ripple Carry vediamo come esiste una propagazione dei riporti per determinare l'ultimo riporto di uscita. Questo comporta ritardi ragionevoli se la catena di Full Adder dovesse diventare troppo lunga.



Esempio di sommatore di parole a 4 bit

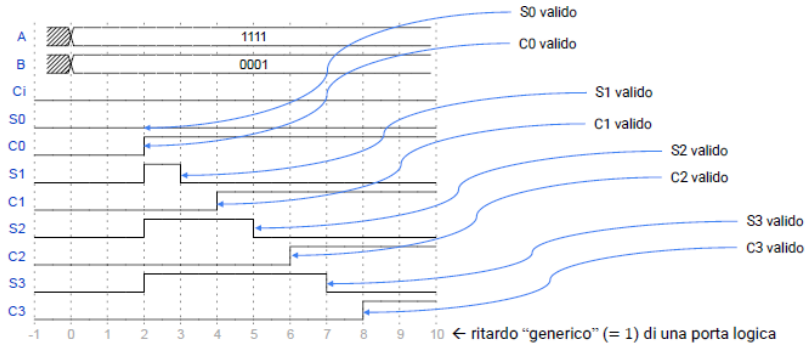
Temporizzazione

Se indichiamo con la notazione $@X$ il tempo di arrivo di un certo dato in un certo punto del circuito in esame e assumiamo che ogni elemento introduca un ritardo unitario, otteniamo le uscite del sommatore con i ritardi indicati in figura.



Esempio di temporizzazione per Adder Ripple Carry

Studiamo come variano le uscite di un Adder Ripple Carry nel caso in cui volessimo sommare 1111_2 e 0001_2. Questa è la situazione peggiore, poiché tutti gli output intermedi devono assestarsi prima che sia valido il MSB.

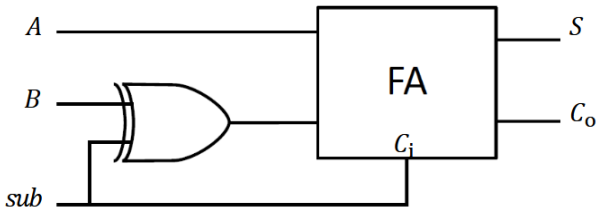


Esempio di variazione delle uscite sommando 1111_2 e 0001_2

⚠ S_1, S_2 e S_3 hanno l'uscita a 1 per un breve periodo di tempo, in modo non corretto, questo perché non hanno ancora ricevuto il carry-in a causa dei ritardi di propagazione.

(*) Addizionatore/sottrattore binario

Possiamo realizzare un sommatore/sottrattore tramite un FA e una porta XOR, e quest'ultima serve per selezionare il comportamento atteso e gestire il valore del secondo addendo.



Esempio di addizionatore/sottrattore binario

In base al valore di sub:

- se $\text{sub} = 0$, lo XOR trasmette B al FA e calcola $A_2 + B_2$ con $C_i = 0$;
- se $\text{sub} = 1$, esegue la differenza, poiché lo XOR trasmette il CA1 di B e il FA esegue la somma $S_2 = A_2 + B_{\text{CA1}} + 1_2$, ma sappiamo che $B_{\text{CA1}} + 1_2 = B_{\text{CA2}}$, e quindi, in CA2, si ha una sottrazione.

Circuito di Carry Lookahead (CLA)

Il circuito di carry lookahead (CLA) serve per minimizzare il ritardo di propagazione dei riporti in un sommatore. Per realizzare questo comportamento, si introducono due espressioni:

- Carry Generate G_i;

$$G_i = A_i \cdot B_i$$

- Carry Propagate P_i.

$$P_i = A_i \oplus B_i$$

La cosa importante è che sia G che P dipendano solo dagli ingressi A e B, non dai valori dei riporti. Possiamo riscrivere le uscite somma e riporto in funzione di G_i e P_i.

$$
\begin{aligned}
S_i &= A_i \oplus B_i \oplus C_i \\
&= P_i \oplus C_i
\end{aligned}
$$

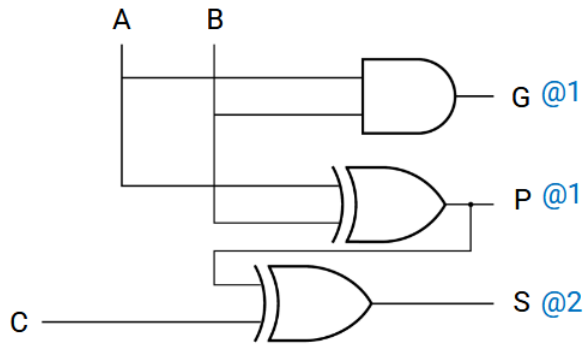
$$
\begin{aligned}
C_{i+1} &= A_i B_i + A_i C_i + B_i C_i \\
&= A_i B_i + C_i (A_i \oplus B_i) \\
&= G_i + C_i P_i
\end{aligned}
$$

Definiamo ora C_i.

$$C_1 = G_0 + P_0 C_0 \tag{1}$$
$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0 \tag{2}$$
$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0 \tag{3}$$
$$\dots \tag{4}$$
$$C_i = \sum_{j=-1}^{i-1} G_j \prod_{k=j+1}^{i-1} P_k, \ \text{con} \ (G_{-1} = C_0) \tag{5}$$

Carry Lookahead Cell (CLC)

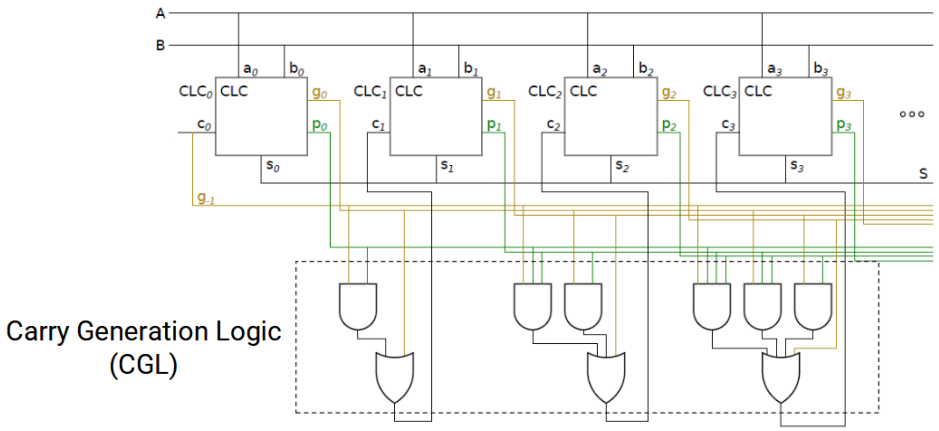
Il Carry Lookahead Cell (CLC), simile ad un Full Adder, ma genera in uscita i segnali G e P e non il carry out.



Esempio di implementazione di CLC

Implementazione del CLA

Le variazioni in ingresso sono solamente gli addendi e il riporto in ingresso allo stadio zero. Questo discorso si traduce nel circuito in figura.

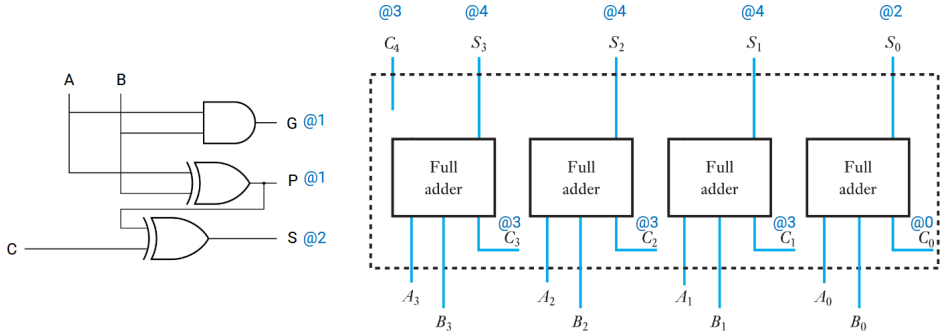


Circuito di Carry Lookahead a 4 bit

A partire da sinistra verso destra vediamo la realizzazione dei vari Carry (Carry Generation Logic), dove il primo blocco implementa C_1, il secondo blocco implementa C_2, e così via. Man mano che le porte logiche nei carry lookahead aumentano (ne bastano anche già 4) il blocco diventa lento nel calcolo del carry_out, di conseguenza si cerca si realizzare Carry Lookahead non troppo grossi.

Temporizzazione

Studiando i ritardi per un carry-lookahead, notiamo che il cammino più lungo è praticamente costante, poiché tutti i carry in (tranne il primo) arrivano contemporaneamente ai Full Adder.

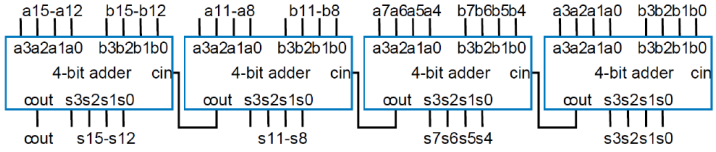


Esempio di calcolo della temporizzazione di un CLA a 4 bit

Ripple CLA

Come abbiamo visto, al crescere del numero di stadi cresce il numero di gate e il rispettivo fan-in. Questo porta ad avere un fan-in troppo elevato e richiede di essere sostituito da una struttura a più livelli, sfruttando porte più piccole.

Una soluzione possibile è quella di collegare in cascata più CLA di dimensione più piccola, dove ogni CLA aspetta il carry in del CLA precedente.

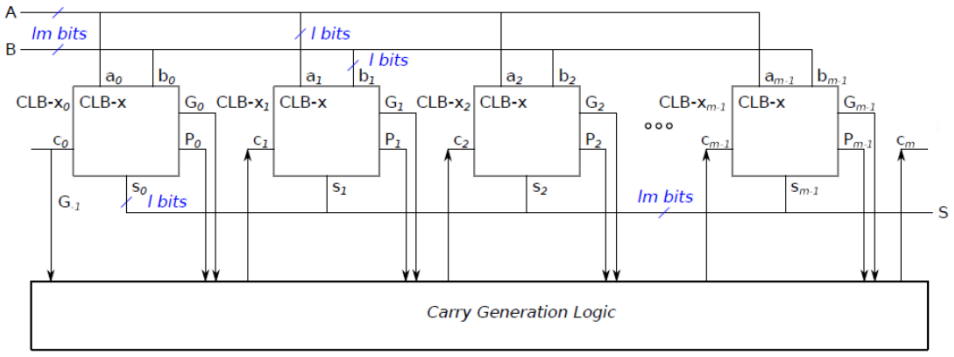


Esempio con 4 CLA da 4 bit

CLA gerarchico

Suddividiamo il parallelismo in ingresso in m parallelismi di l bit. Gli l bit vengono gestiti ognuno da un Carry Lookahead Block (CLB), che può includere qualsiasi tipo di sommatore:

- CLB-r (ripple);
- CLB-c (carry lookahead).

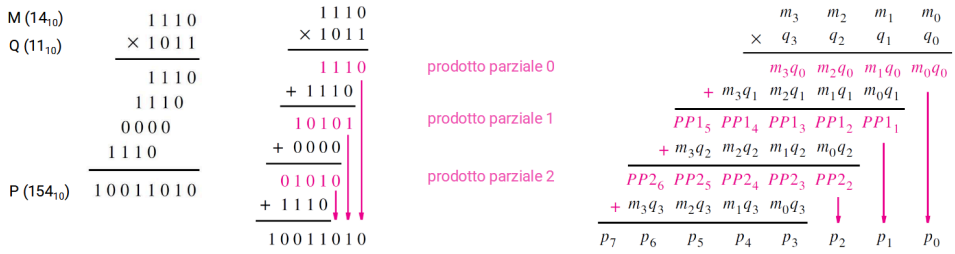


Esempio di CLA gerarchico a 4 blocchi

Moltiplicatore parallelo

Una moltiplicazione binaria per un numero senza segno si può fare come quella in base decimale, esaminando i bit da destra a sinistra:

- se un bit è a 1, si aggiunge una versione opportunamente traslata del moltiplicando a un prodotto parziale;
- se invece il bit è a 0 non fornisce contributo.

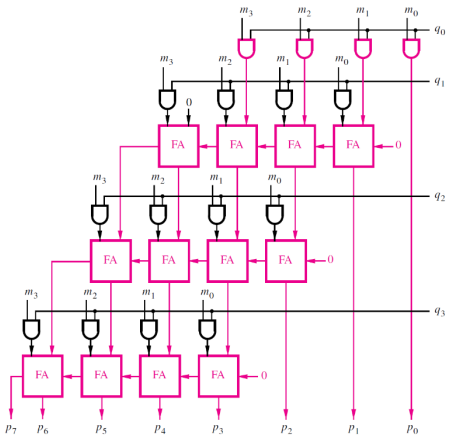


Esempio di moltiplicazione binaria

Per evitare di introdurre sommatori con un numero non costante di ingressi, dobbiamo implementare prodotti e somme parziali, come in figura, e con i seguenti componenti:

- 16 porte AND per calcolare i prodotti bit-a-bit;
- 12 sommatori per la somma dei prodotti parziali (Carry-In nullo).

Per le somme di ordine più elevato si sfruttano più Carry-Out in parallelo.



Implementazione di un moltiplicatore parallelo

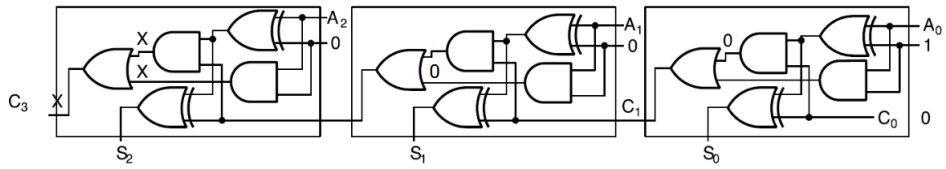
Altre funzioni aritmetiche

Molte volte conviene semplificare dei blocchi già esistenti rimuovendo una parte di circuiteria ridondante quando alcuni input sono costanti:

- incremento/decremento, fissando ad una costante uno dei due operandi;
- moltiplicazione/divisione per una costante;
- estensione del segno e zero-fill, con cui vado a riempire di 1 o 0 tutti i nuovi bit che sono stati aggiunti prima se, ad esempio, cambio la lunghezza di una parola.

Esempio - Semplificazione di un sommatore Ripple Carry

Cerchiamo di semplificare un sommatore Ripple Carry, il quale fa un incremento unitario.



Circuito iniziale del sommatore Ripple Carry di esempio

Prendiamo, per esempio, il blocco di porte logiche più a sinistra, definito dagli ingressi A_0 e 1 e dalle uscite S_0 e C_1. Si può cercare di descrivere il comportamento della porta logica in funzione degli ingressi.

Cerchiamo di definire la tabella in funzione dell'ingresso A_0 e 1, utilizzando la funzione logica definita dal circuito iniziale:

$$\begin{cases} S_0 = (A_0 \oplus 1) \oplus C_0 \\ C_1 = [C_0 \cdot (A_0 \oplus 1)] + [A_0 \cdot 1] \end{cases}$$

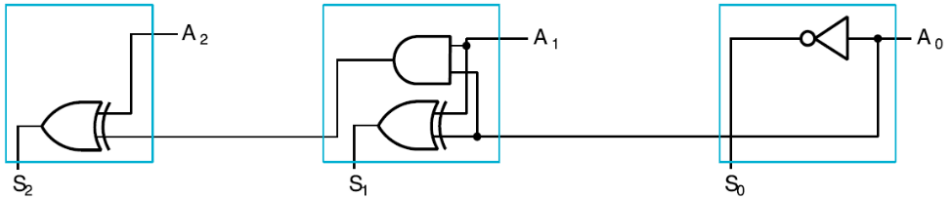
A_0	C_0	S_0
0	0	1
1	0	0

Si può notare, attraverso la tabella creata, che C_0 vale sempre 0, poiché il carry della prima cifra, il quale non ha alcuna operazione che lo definisce, vale sempre 0. In questo caso, si ha che C_0 non influisce in alcun modo sul risultato finale, quindi si può omettere.

A_0	S_0
0	1
1	0

Successivamente, si può notare come $S_0 = \overline{A}_0$ e $C_1 = A_0$ attraverso l'ultima tabella definita, perciò si ha, nel circuito, che l'uscita S_0 è definita attraverso la negazione dell'ingresso S_0, mentre l'uscita C_1 è la forma diretta di A_0.

Si ripete lo stesso ragionamento per ognuno dei tre blocchi di porte logiche, stando attenti al valore dei rispettivi carry che, al contrario di C_0, possono avere sia valore 0, sia valore 1, e si ottiene il seguente schema:



Semplificazione del sommatore Ripple Carry