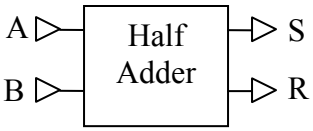
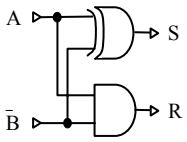
Esercitazione del 23/03/2006 - Soluzioni

1) **Addizionatore Half Adder (senza riporto in ingresso)**:

A	B	S	R
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



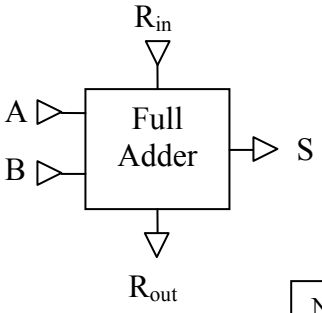


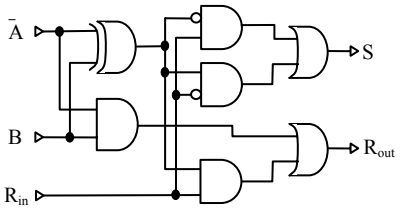
N.Porte = 2 *Cammino Critico* S = 1 , R = 1

$S = A \oplus B \qquad R = A \cdot B$

2) **Addizionatore Full Adder (con riporto in ingresso)**:

R_{in}	A	B	S	R_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1





N.Porte = 7 *Cammino Critico* S = 3 , R = 3

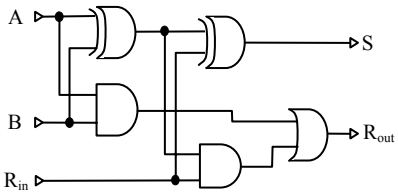
$S = \sim R_{in} \sim AB + \sim R_{in} A \sim B + R_{in} \sim A \sim B + R_{in} A\, B$
$= \sim R_{in}(\sim AB + A \sim B) + R_{in}(\sim A \sim B + A\, B)$
$= \sim R_{in}(A \oplus B) + R_{in} \sim (A \oplus B)$

$R_{out} = \sim R_{in} AB + R_{in} \sim AB + R_{in} A \sim B + R_{in} AB$
$= AB\,(\sim R_{in} + R_{in}) + R_{in}\,(\sim AB + A \sim B)$
$= AB + R_{in}(A \oplus B)$

Nota: **~A~B + A B** = ~A~B + A B = ~(~(~A~B + A B))
= ~(~(~A~B) ~ (A B)) = ~((~~A+~~B) (~A+~B))
=~((A+B) (~A+~B)) = ~(A(~A+~B)+B(~A+~B))
=~(A~A+A~B+B~A+B~B) = ~(A~B+B~A) = **~ (A ⊕ B)**

Semplificazione circuitale:

$S = \sim R_{in}(A \oplus B) + R_{in} \sim (A \oplus B) = R_{in} \oplus (A \oplus B)$



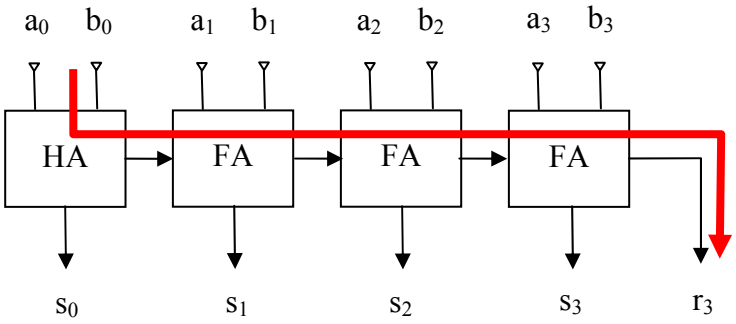
N.Porte = 5 *Cammino Critico* S = 2 , R = 3

***Nota**: le porte **AND e la OR possono essere pensate come "gate"** che lasciano "passare" il segnale presente su un terminale in funzione del segnale presente sull'altro. Diversamente **le porte XOR si comportano come "invertitori"**: il segnale presente su un terminale viene lasciato passare o invertito a seconda del segnale presente sull'altro.*

Massimo Marchi – marchi@dsi.unimi.it

3) Sommatore ad n bit (senza riporto in ingresso)

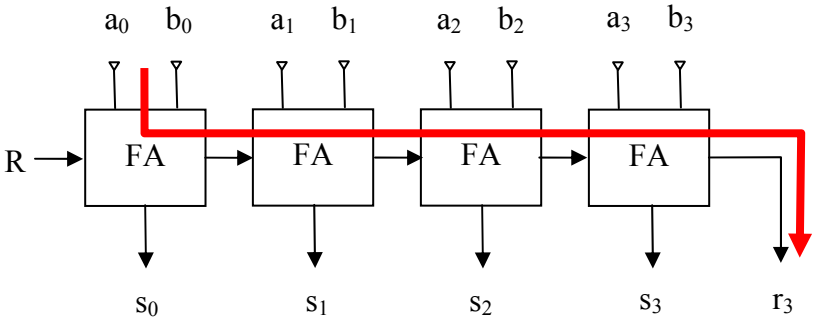
E' possibile realizzare un sommatore ad n bit usando un HA e n-1 FA collegati in cascata.



Il cammino critico è quello definito dalla propagazione dei riporti dal primo modulo all'ultimo. Il **cammino critico** quindi per questo sommatore è **1 + 3 * (n-1)** dove **n** è il numero di bit da sommare.

4) Sommatore ad n bit (con riporto in ingresso)

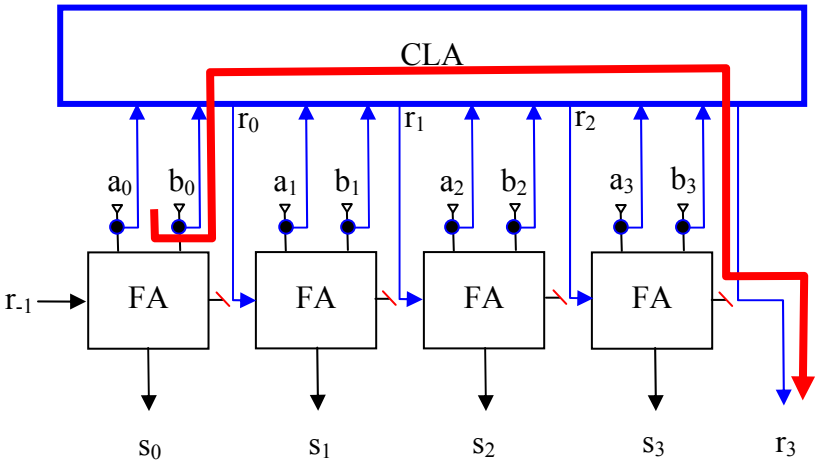
Normalmente si preferisce adottare un FA anche per il primo modulo. Questo permette di mettere in cascata più sommatori e di realizzare semplicemente un sottrattore binario in complemento a due (vedi più avanti).



In questo caso quindi il **cammino critico** è **3 * n**.

5) Sommatori veloci: unita di Carry Look-Ahead

Il tempo di commutazione dei circuito sommatore ad n bit dipende dal tempo di propagazione del riporto dalla coppia di bit meno significativa alla coppia di bit più significativa. Se riusciamo a *prevedere* il riporto r_i che sarà sommato ad ogni coppia di bit prima che avvenga la propagazione dallo stadio precedente possiamo accelerare il processo di somma. Si tratta di anteporre,o meglio affiancare al circuito sommatore un'unità di **Carry Look-Ahead** (**CLA**), di previsione del riporto.



Consideriamo il generico sommatore Full-Adder ad un bit in posizione i-esima. Il suo cammino critico è determinato dal calcolo del resto in uscita:

$$r_i = a_i b_j + r_{i-1}(a_i \oplus b_i) = g_i + r_{i-1} * p_i$$

Il termine $a_i b_j = g_i$ calcola se nello stadio i-esimo verrà generato un riporto mentre il termine $a_i \oplus b_i = p_i$ calcola se il riporto dello stadio precedente verrà propagato a quello successivo. Da notare che tutti questi termini possono essere calcolati in parallelo con cammino critico uguale a 1.
Per il circuito in esempio quindi vale:

$$\mathbf{r_0 = g_0 + r_{-1} * p_0} \quad \text{e} \quad r_1 = g_1 + r_0 * p_1$$

Sviluppando:

$$\mathbf{r_1} = g_1 + r_0 * p_1 = g_1 + (g_0 + r_{-1} * p_0) * p_1 = \mathbf{g_1 + g_0\, p_1 + r_{-1}\, p_0\, p_1}$$

cioè si verifica un riporto r_1 nello stadio 1 se viene generato nello stadio stesso (g_1) oppure se si verifica nel precedente e viene propagato dallo stadio attuale ($g_0\, p_1$) oppure se c'era un riporto in ingresso e i due stadi iniziali lo hanno propagato ($r_{-1}\, p_0\, p_1$). Da notare che il riporto r_1 con questa forma algebrica può essere calcolato senza tenere conto del risultato degli stadi FA, al contrario di come accade con la formula classica.
Sviluppiamo ora i riporti successivi:

$\mathbf{r_2}$ $= g_2 + r_1 * p_2 = g_2 + (g_1 + g_0\, p_1 + r_{-1}\, p_0\, p_1) * p_2$
$= \mathbf{g_2 + g_1\, p_2 + g_0\, p_1\, p_2 + r_{-1}\, p_0\, p_1\, p_2}$

$\mathbf{r_3}$ $= g_3 + r_2 * p_3 = g_3 + (g_2 + g_1\, p_2 + g_0\, p_1\, p_2 + r_{-1}\, p_0\, p_1\, p_2) * p_3$
$= \mathbf{g_3 + g_2\, p_3 + g_1\, p_2\, p_3 + g_0\, p_1\, p_2\, p_3 + r_{-1}\, p_0\, p_1\, p_2\, p_3}$

Riepilogando, un CLA a 4 bit deve realizzare le seguenti funzioni booleane:

$$\mathbf{r_0 = g_0 + r_{-1} * p_0}$$

$$\mathbf{r_1 = g_1 + g_0\, p_1 + r_{-1}\, p_0\, p_1}$$

$$\mathbf{r_2 = g_2 + g_1\, p_2 + g_0\, p_1\, p_2 + r_{-1}\, p_0\, p_1\, p_2}$$

$$\mathbf{r_3 = g_3 + g_2\, p_3 + g_1\, p_2\, p_3 + g_0\, p_1\, p_2\, p_3 + r_{-1}\, p_0\, p_1\, p_2\, p_3}$$

Calcoliamo il cammino critico per queste funzioni: dividiamo gli operatori in modo da avere solo operatori binari inserendo una coppia di parentesi e valutiamo il numero di livelli di parentesi:

$$\mathbf{r_0} = [g_0 + (r_{-1} * p_0)] \rightarrow \textbf{C.Critico = 2 } \textit{(+1)}$$

$$\mathbf{r_1} = \{[g_1 + (g_0\, p_1)] + [r_{-1}\, (p_0\, p_1)]\} \rightarrow \textbf{C.Critico = 3 } \textit{(+1)}$$

$$\mathbf{r_2} = \mathbf{(}\{[g_2 + (g_1\, p_2)] + [g_0\, (p_1\, p_2)]\} + [(r_{-1}\, p_0)(\, p_1\, p_2)]\mathbf{)} \rightarrow \textbf{C.Critico = 4 } \textit{(+1)}$$

$$\mathbf{r_3} = \mathbf{[}\{[g_3 + (g_2\, p_3)] + [g_1(\, p_2\, p_3)]\} + \mathbf{(}[(g_0\, p_1)\, (p_2\, p_3)\,] + \{[r_{-1}\, (p_0\, p_1)](\, p_2\, p_3)\}\mathbf{)]} \rightarrow \textbf{C.Critico = 5 } \textit{(+1)}$$

Tutti questi riporti possono essere calcolati in parallelo quindi il cammino critico più lungo è quello corrispondente a $\mathbf{r_3}$ pari a **6** (il *+1* indica il tempo di calcolo dei termini g_i ed r_i).
Calcoliamo il cammino critico per $\mathbf{s_i}$:

$$\mathbf{s_i} = (a_i \oplus b_i) \oplus r_{i-1} = [(p_i) \oplus (r_{i-1})] \rightarrow \textbf{C.Critico = C.Critico}(\mathbf{r_{i-1}}) \textbf{ + 1}$$

Ne segue che il cammino critico più lungo per il calcolo del risultato corrisponde al calcolo di $\mathbf{s_3}$ ed è pari a **6**, come per il calcolo di $\mathbf{r_3}$.

Da quanto visto, ne segue che il cammino critico complessivo per un sommatore a 4 bit con CLA è pari al cammino critico per $\mathbf{r_3}$ (o anche a quello di $\mathbf{s_3}$). Un corrispondente sommatore classico senza anticipazione di riporto ha cammino critico **3 * 4 = 12** (da cui la dicitura "sommatore veloce"!!).

La complessità di circuito di un'unità CLA aumenta all'aumentare del numero di bit da trattare. Questo rende economico sviluppare sommatori veloci solo per basse cardinalità di bit e se necessario utilizzare questi sommatori come blocchi per sommatori modulari di più grandi dimensioni (vedi slides di teoria).

Massimo Marchi – marchi@dsi.unimi.it

6) **Moltiplicazioni binarie**

(*Rappresentazione dell'Informazione – Conversione da base 10 a base n , slide 24-25)*

a. $1001_2 * 110_2 = ?_2$

$1001_2 * 110_2 = 1001_2 * 0_2 + 1001_2 * 10_2 + 1001_2 * 100_2$

$= 0_2 + 10010_2 + 100100_2$

$= 110110_2$

		1	0	0	1	*	
			1	1	0	=	
						0 +	0
	1	0	0	1	←	+	1
1	0	0	1	←		=	1
1	1	0	1	1	0		

La moltiplicazione binaria si esegue in maniera simile alla moltiplicazione decimale classica. Per le proprietà dell'aritmetica binaria tutto il processo si riduce ad una sequenza di shift a sinistra e somme.

Per ogni cifra del moltiplicatore si esegue un shift a sinistra del primo termine. Il termine shiftato si somma al totale solo se il corrispondente bit del secondo termine è a 1. In altri termini, ogni shift viene messo in AND con il corrispondente bit del moltiplicatore e quindi sommato.

b. $101_2 * 1101_2 = ?_2$

				1	0	1	*	
			1	1	0	1	=	
				1	0	1	+	1
					0	←	=	0
		1	0	1	←			1
		1	*1*	*^{1}0*	*0*	*1*		
	1	0	1	←				1
1	^{1}0	^{1}0	^{1}0	0	0	1		→1000001_2

c. $11101_2 * 1011_2 = ?_2$

				1	1	1	0	1	*	
					1	0	1	1	=	
				1	1	1	0	1	+	1
			1	1	1	0	1	←	=	1
		1	*^{1}0*	*^{1}1*	*^{1}0*	*1*	*1*	*1*	+	
						0	←			0
	1	1	1	0	1	←				1
1	^{1}0	^{1}0	1	1	1	1	1	1		→100111111_2

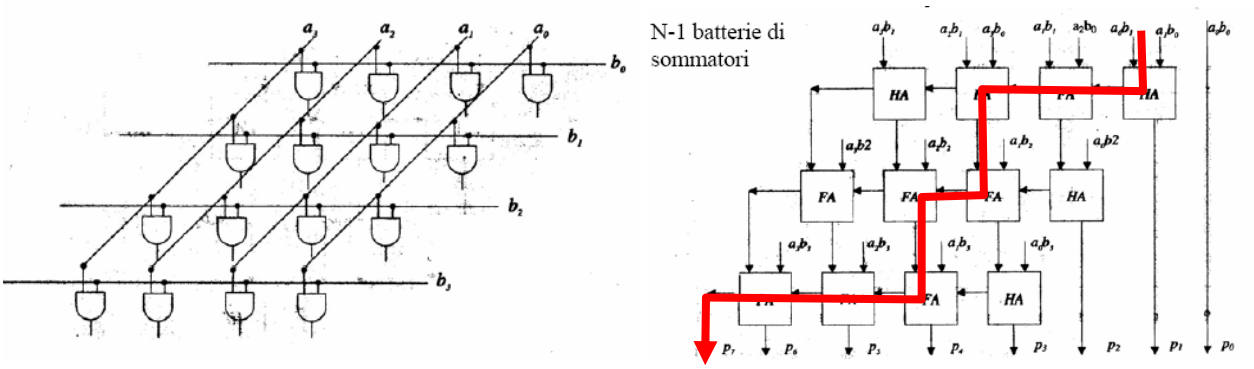
d. $10111_2 * 111_2 = ?_2$

			1	0	1	1	1	*	
					1	1	1	=	
			1	0	1	1	1	+	1
		1	0	1	1	1	←	=	1
	1	*^{1}0*	*^{1}0*	*^{1}0*	*^{1}1*	*^{1}0*	*1*	+	
	1	0	1	1	1	←			1
1	^{1}0	1	^{1}0	^{1}0	^{1}0	0	1		→10100001_2

Massimo Marchi – marchi@dsi.unimi.it

Il circuito della moltiplicazione può essere visto come una matrice di AND seguita da una matrice di sommatori FA-HA. Ogni bit b_i del moltiplicatore si comporta come un gate per la corrispondente parola del moltiplicando opportunamente shiftata a sinistra di i posizioni.

> *Nota: L'operazione di shift a sinistra equivale in algebra binaria ad una moltiplicazione per 10_2, analogamente lo shift a destra equivale ad una divisione per 10_2.*



> *Nota: HA è più rapido di FA:*
>
> HA → *Cammino Critico* S = 1 , R_{out} = 1
>
> FA → *Cammino Critico* S = 2 , R_{out} = 3
>
> *quindi il percorso critico a parità di moduli attraversati è quello che attraversa un maggior numero di FA.*

Il percorso più lungo è quello che propaga gli effetti dei bit a_0 e b_0 su r_n. Per propagare il riporto verso sinistra occorrono 3 passaggi, per eseguire un livello di sommatorie occorrono 2 passaggi.

*Nota: la lunghezza del risultato di una moltiplicazione sarà al più la somma della lunghezza dei due termini. Es: 8 bit * 8 bit →16 bit $1111\ 1111_2 * 1111\ 1111_2 = 1111\ 1110\ 0000\ 0001_2$*

Massimo Marchi – marchi@dsi.unimi.it

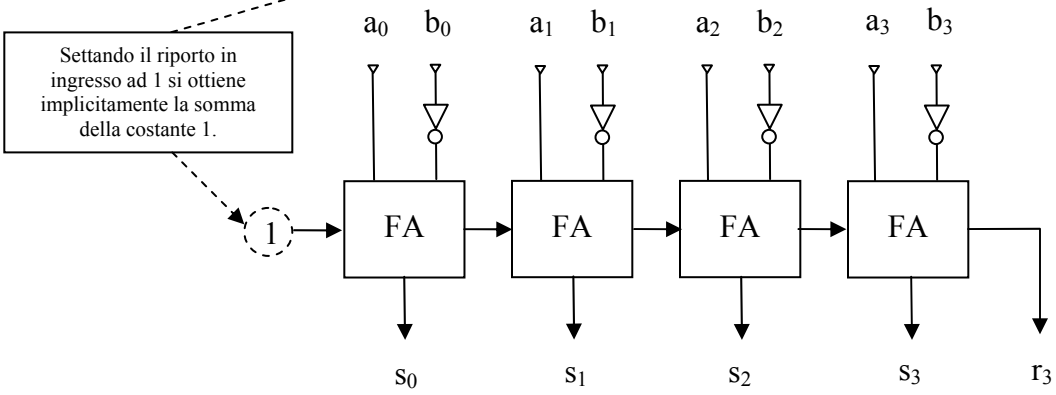
7) Sottrattori ad n bit

In binario la sottrazione ad n bit (con segno) può essere realizzata con una somma secondo la regola:

$$\mathbf{A - B = A + \sim B + 1}$$

dove **~B** è l'inversione bit a bit del secondo termine.

Adottando un sommatore con riporto in ingresso è possibile realizzare un sottrattore binario a n bit in questo modo:



8) Problema dell'overflow

Nel caso che il risultato di un addizione con segno ecceda il limite di rappresentazione della dimensione della parola si verifica un errore di segno (Overflow).

Esempio:

$19_{10} + 15_{10} = 34_{10}$ se eseguita in aritmetica binaria con segno a 6 bit dà come risultato:

1	*1*	*1*	*1*	*1*		*R*
0	**1**	**0**	**0**	**1**	**1**	+
0	**0**	**1**	**1**	**1**	**1**	=
1	^{1}**0**	^{1}**0**	^{1}**0**	^{1}**1**	^{1}**0**	

$0\ 10011_2 + 0\ 01111_2 = 1\ 00010_2$ che in rappresentazione in complemento a due è un numero negativo , precisamente $- 11110_2 = -30_{10}$.

Analogamente si verifica lo stesso effetto di riporto errato sul bit di segno quando si sommano due numeri negativi in valore assoluto troppo grandi:

$-17_{10} + -19_{10} = -35_{10}$ *se eseguita in aritmetica binaria con segno a 6 bit dà come risultato:*

$-17_{10} = -\ 10001_2$ (M&S) $= 101111_2$ (CA2)

$-19_{10} = -\ 10011_2$ (M&S) $= 101101_2$ (CA2)

1	*1*	*1*	*1*	*1*	*1*		*R*
	1	**0**	**1**	**1**	**1**	**1**	+
	1	**0**	**1**	**1**	**0**	**1**	=
~~**1**~~	^{1}**0**	**1**	^{1}**1**	^{1}**1**	^{1}**0**	^{1}**0**	

$1\ 01111_2 + 1\ 01101_2 = 0\ 11100_2$ che in rappresentazione in complemento a due è un numero positivo , precisamente $+ 00100_2 = 4_{10}$.

Massimo Marchi – marchi@dsi.unimi.it

Nel caso di somme miste, negativo con positivo e viceversa, questo problema non si pone poiché il risultato in valore assoluto sarà sempre al più grande come il più grande in valore assoluto dei due termini sommati.

Circuito per rilevare l'overflow:

Da quanto visto sopra ne segue che l'overflow si verifica solo nei seguenti due casi:

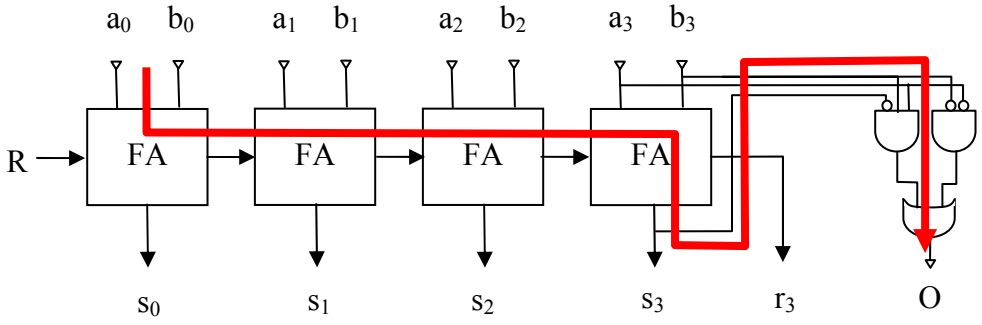
Segno di A = 0 e Segno di B = 0 e Segno del risultato = 1 oppure

Segno di A = 1 e Segno di B = 1 e Segno del risultato = 0

Il circuito che realizza questo controllo sintetizzera la seguente forma tabellare:

r_{n-1}	a_{n-1}	b_{n-1}	Overflow
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

In forma SOP si può scrivere: $O = \sim r_{n-1}\, a_{n-1}\, b_{n-1} + r_{n-1} \sim a_{n-1} \sim b_{n-1}$

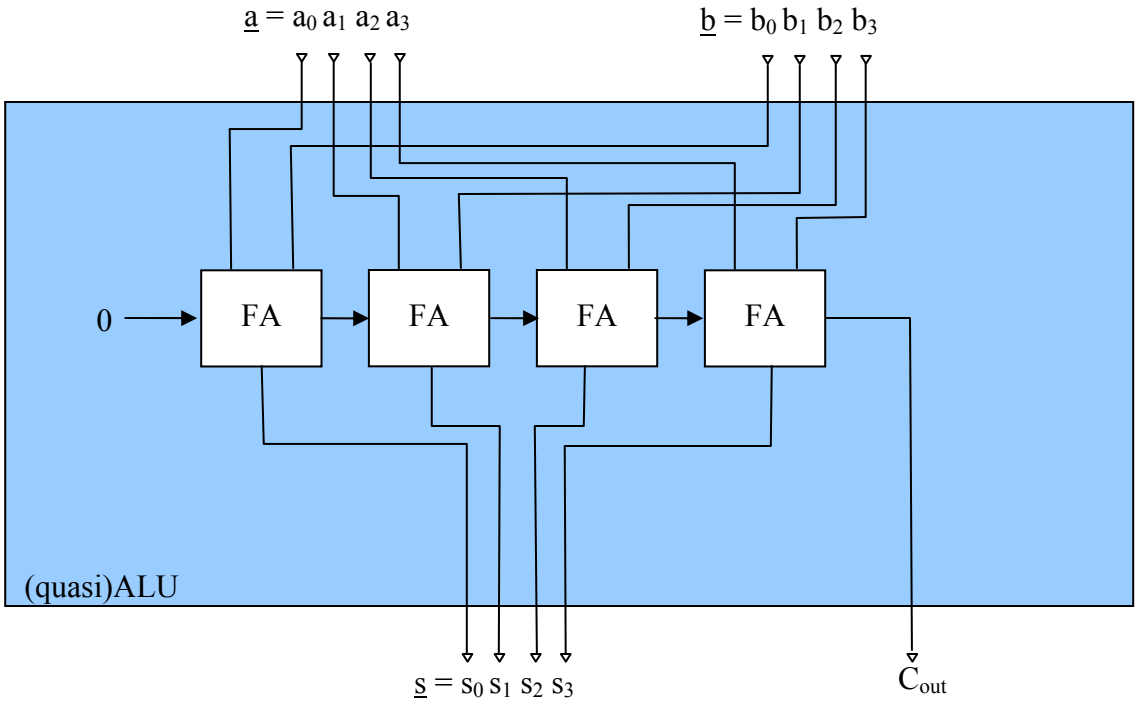


Cammino critico: 3(n-1) +2 +2

Massimo Marchi – marchi@dsi.unimi.it

9) ALU: Arithmetic Logic Unit a 4 bit

Consideriamo un sommatore a 4 bit (per semplicità senza CLA) e consideriamolo come un blocco unico che accetta in ingresso due parole da 4 bit **a** e **b** e restituisce in uscita 4 bit di risultato **s** ed il riporto dell'ultimo stadio $\mathbf{C_{out}}$ (*Carry Out*):



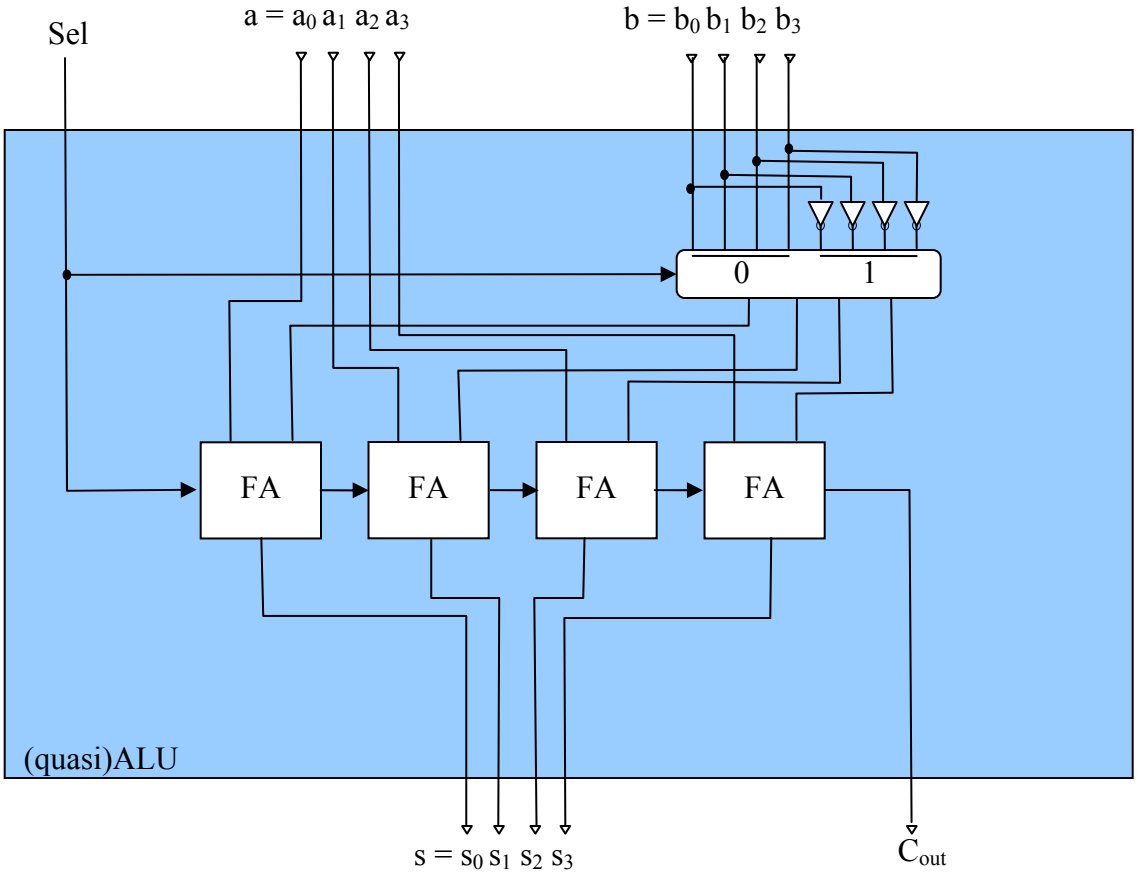
La (quasi)ALU così costruita realizza ovviamente una sola operazione, la somma, e quindi come tale non è necessaria nessuna linea di selezione dell'operazione da eseguire.

Supponiamo ora di voler aggiungere alla mia (quasi)ALU la possibilità di eseguire la sottrazione invece della somma a seconda del segnale presente su una linea di selezione **Sel**. Se **Sel=0** allora si desidera eseguire la somma **a** + **b** , se **Sel=1** allora si desidera eseguire la sottrazione **a** – **b**.
Date due parole binarie **a** e **b** si dimostra che:

$$\mathbf{a} - \mathbf{b} = \mathbf{a} + \mathbf{not}(\mathbf{b}) + \mathbf{1}$$

Porre il riporto in ingresso al primo stadio di un sommatore uguale a 1 equivale a sommare 1 al risultato finale. Ne segue che possiamo riorganizzare il circuito sommatore per eseguire all'occorrenza anche la sottrazione.

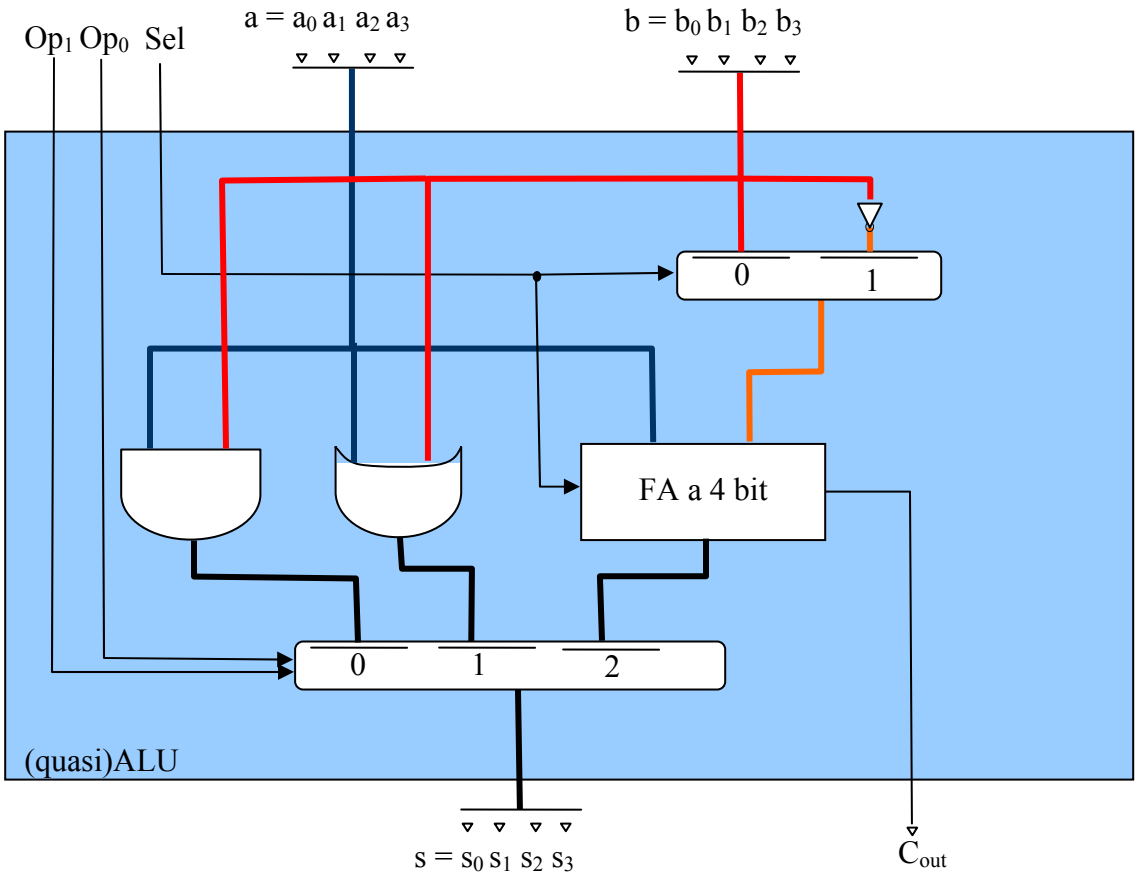
L'operazione di inversione *a comando* della parola **b** può essere ottenuta attraverso una batteria di porte NOT ed un selettore a 4 bit comandato da **Sel:**



La (quasi)ALU così progettata realizza le operazioni di somma e sottrazione.

Massimo Marchi – marchi@dsi.unimi.it

Sfruttando una batteria di porte AND, una batteria di porte OR e una nuova linea di selezione, questa volta a tre vie, possiamo aggiungere due nuove operazioni, AND e OR (*le linee in grassetto indicano bus a 4 bit,il FA come pure le prote AND e OR sono state raggruppate logicamente in blocchi*)



La (quasi)ALU così progettata può eseguire somme, sottrazioni, AND bit a bit ed OR bit a bit. Tramite i segnali di controllo **Sel**, **Op_1** e **Op_0** I blocchi all'interno dell'ALU vengono *combinati* per ottenere in uscita il risultato dell'operazione desiderata. I segnali **Op_1** e **Op_0** selezionano quale risultato mandare verso l'uscita, se il risultato della AND nel caso **Op_1=0** e **Op_0=0,** o il risultato della OR per **01** oppure il risultato del sommatore per **10**, la somma o la differenza a seconda del segnale presente su **Sel** (nei casi AND e OR **Sel** non ha effetto).

Massimo Marchi – marchi@dsi.unimi.it

Supponiamo ora di voler dotare la (quasi)ALU della funzione di comparazione: date due parole di 4 bit **a** e **b**, **comp(a , b)** vale:

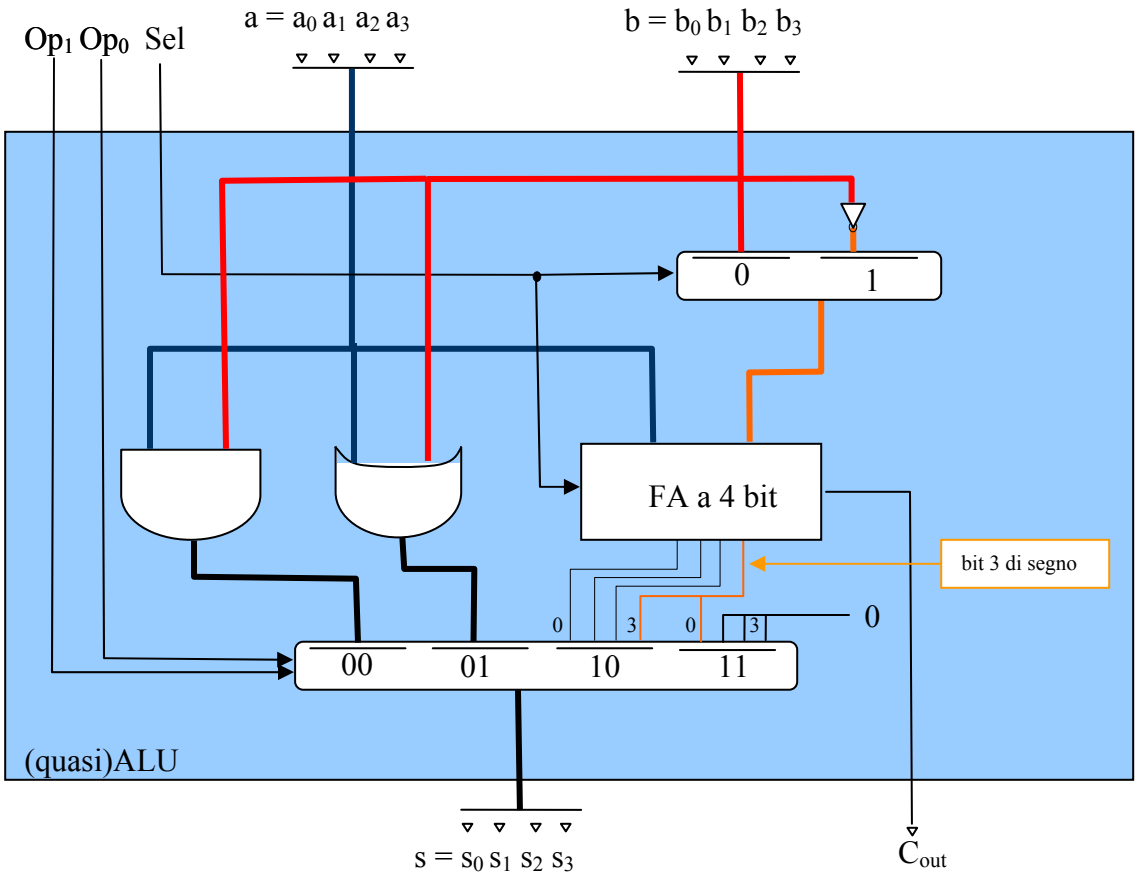
1 se **a** < **b** , **0** se **a** >= **b**

Ora questo equivale a:

1 se **a** - **b** < 0 , **0** se **a** – **b** >= 0

cioè **comp(a , b)** vale **1** se il risultato della differenza è negativo, **0** altrimenti. In altre parole, il risultato della comparazione è pari al valore del bit di segno della differenza tra le due parole **a** e **b**.

Per implementare questa funzione occorre che la ALU venga *programmata* per realizzare la differenza (controllo **Sel=1**) quindi, attraverso un nuovo percorso dei dati all'interno della ALU, il bit di segno deve essere riportato sul **bit 0** mentre tutti gli altri bit del risultato vanno posto a **0** *(dato il bit di segno* ***s*** *il risultato della operazione di comparazione è uguale a* ***000s****)*. Utilizziamo $\mathbf{Op_1=1}$ e $\mathbf{Op_0=1}$, per indicare questo nuovo percorso:



La tabella delle operazioni eseguibili da questa (quasi)ALU è riassunta dalla tabella a fianco. Da notare che per le operazioni AND ed OR il valore del controllo **Sel** è ininfluente mentre la configurazione **Op=11** e **Sel=0** è senza significato (il risultato è il segno dell'addizione tra **a** e **b**).

$\mathbf{Op_1}$	$\mathbf{Op_2}$	Sel	Operazione
0	0	x	**a** AND **b**
0	1	x	**a** OR **b**
1	0	0	**a** + **b**
1	0	1	**a** - **b**
1	1	1	**comp(a,b)**
1	1	0	**????**

Massimo Marchi – marchi@dsi.unimi.it