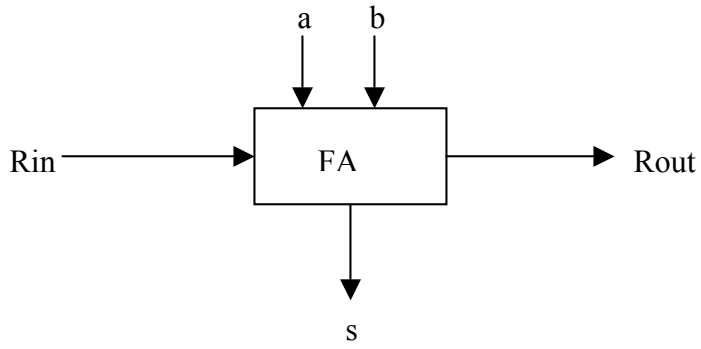
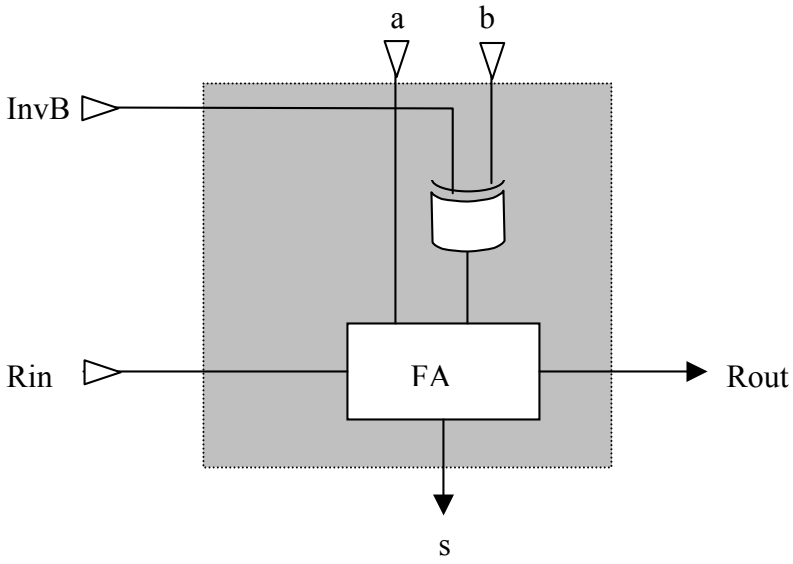
Arithmetic Logic Unit

ALU: Arithmetic Logic Unit. Un esempio a 1 bit

Consideriamo un modulo FA:

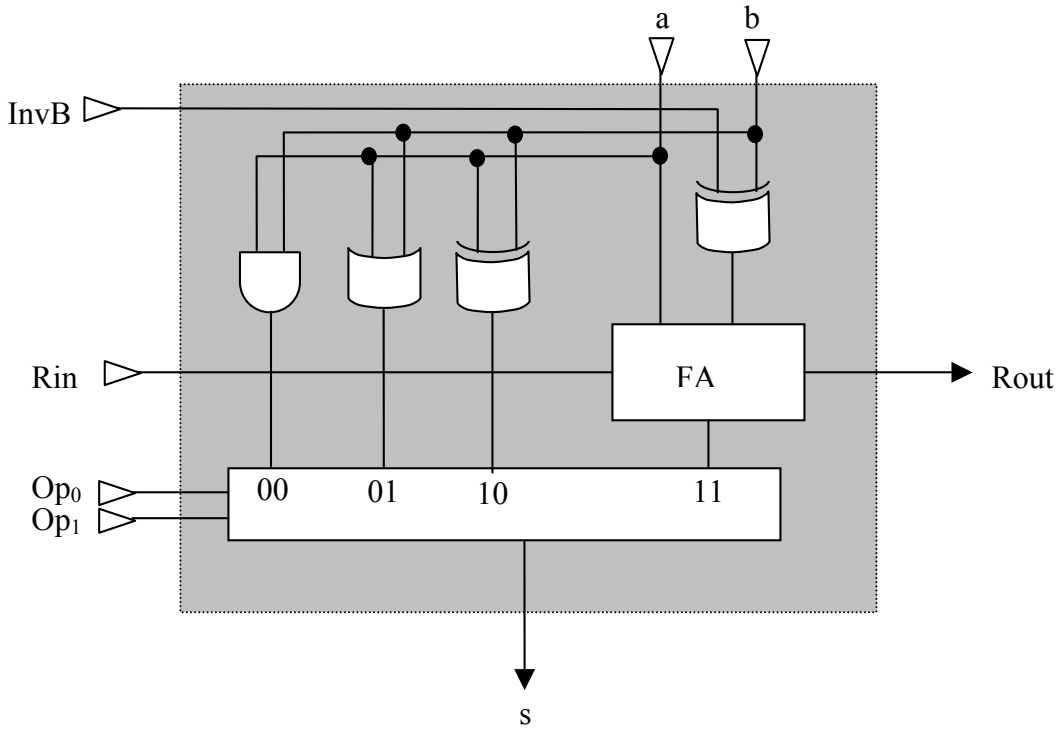


Aggiungendo un segnale di controllo ed un invertitore possiamo ottenere un l'inversione del bit b, utile per ralizzare un somma/sottrattore:



Massimo Marchi – marchi@dsi.unimi.it

Se aggiungiamo in uscita un multiplexer per selezionare cosa mandare verso l'uscita possiamo introdurre nuove operazioni sui due bit in ingresso[1]:



Il modulo così ottenuto è in grado di computare una serie di operazioni sui due bit in ingresso a e b in parallelo. Il risultato in uscita viene selezionato in base ai selettori del multiplexer Op_1Op_0.
La funzione computata è mostrata dalla seguente tabella:

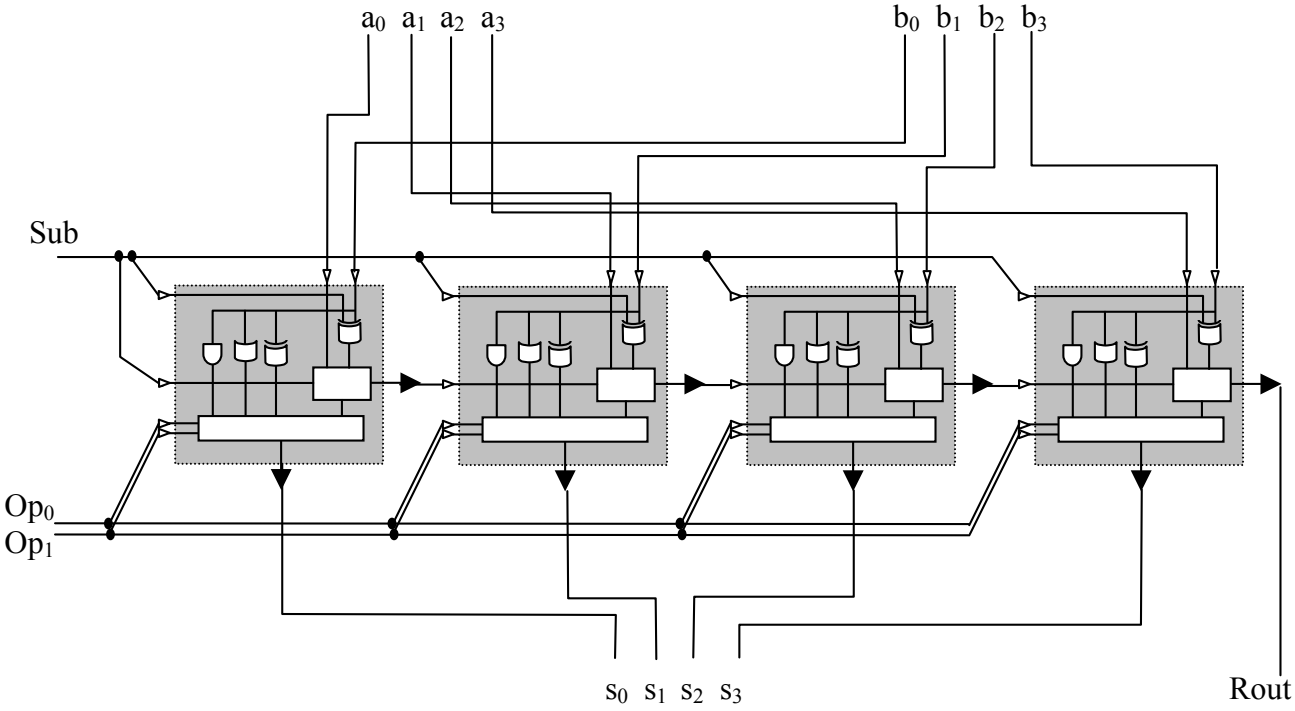
InvB	Op_1	Op_0	S	Rout
X	0	0	S=A and B	*Rout=AB+Rin (a ⊕ b)*
X	0	1	S=A or B	*Rout=AB+Rin (a ⊕ b)*
X	1	0	S=A ⊕ B	*Rout=AB+Rin (a ⊕ b)*
0	1	1	S=Axor B xor Rin	Rout=AB+Rin (a ⊕ b)
1	1	1	S=A ⊕ ~B ⊕ Rin	Rout=A~B+Rin (a ⊕ ~b)

E' possibile estendere l'insieme delle funzioni computate usando come secondo ingresso delle porte aggiuntive il segnale della XOR di inversione di B invece che B direttamente. Le configurazioni 100,101 e 110 in questo caso computerebbero le funzioni A and ~B, A or ~B e A XOR ~B, in tutto e per tutto valide anche se di poco significato pratico.
Da notare che per ogni configurazione viene sempre computato Rout anche se la funzione selezionata per l'uscita non è la somma.

[1] E' possibile risparmiare due porte estraendo dal sommatore FA i segnali relativi alle funzioni a AND b e a XOR b

ALU: Arithmetic Logic Unit. Un esempio a 4 bit realizzato a moduli

Collegando quattro moduli in cascata si può realizzare una ALU a 4 bit:



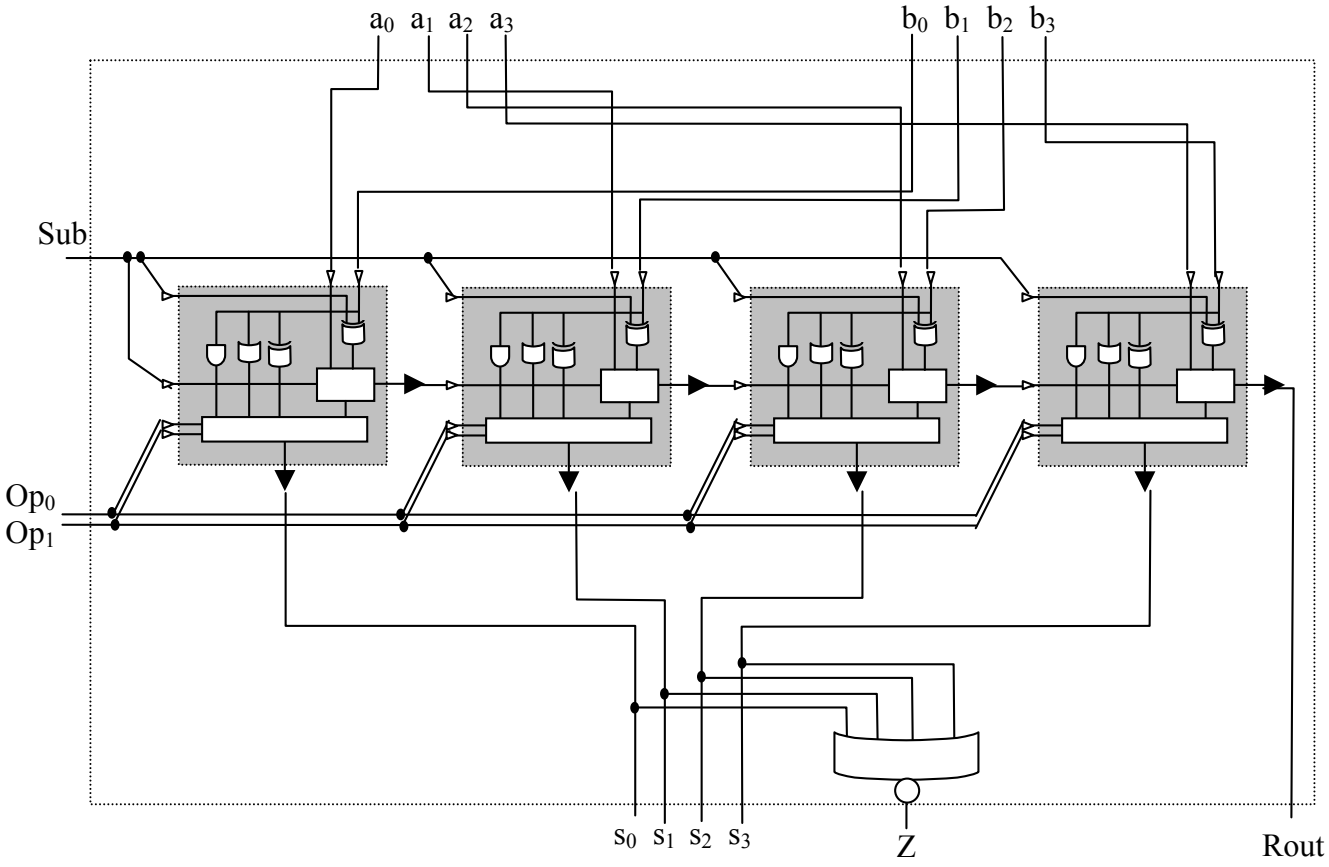
Il modulo computa le seguenti funzioni:

Sub	Op_1	Op_0	S
X	0	0	S=A and B
X	0	1	S=A or B
X	1	0	S=A $\oplus$ B
0	1	1	<Rout,S>=A+B
1	1	1	<Rout,S>=A-B

Massimo Marchi – marchi@dsi.unimi.it

Indicatore di zero

A volte è utile sapere se il risultato di una somma o di una sottrazione somma è uguale a zero. Ad esempio se si vuole sapere se due parole binarie sono uguali è sufficiente realizzare la sottrazione tra le due parole e guardare se il risultato è uguale a 0[2]. Se consideriamo l'uscita di ogni modulo e realizziamo un or tra tutte le uscite possiamo realizzare un comparatore utilizzando la funzione sottrazione già implementata nel modulo precedente:

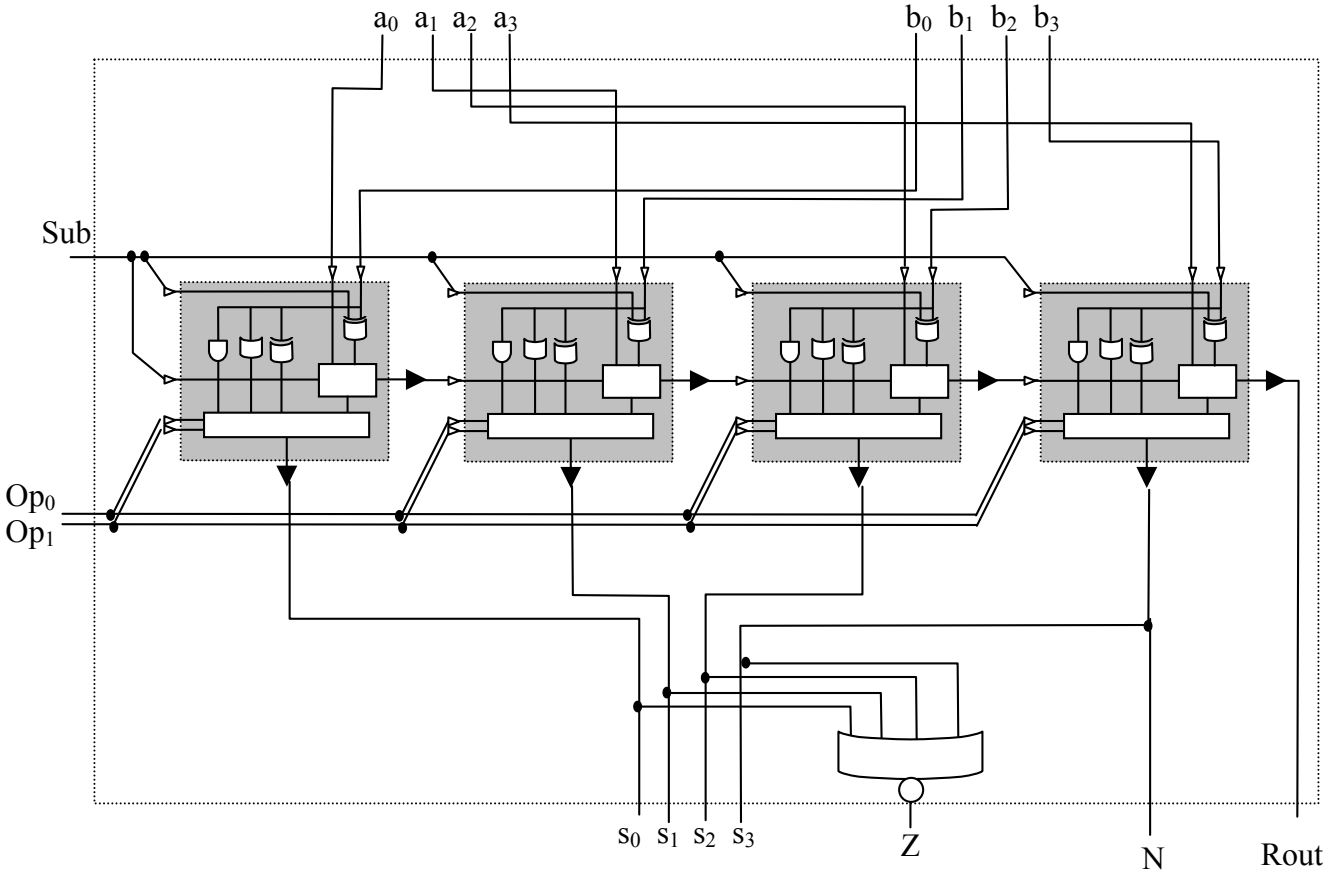


L'uscita Z del modulo vale 0 quando il risultato S è diverso da 0 mentre vale 1 quando il risultato è uguale a 0.

[2] Un metodo più efficiente in termini di cammino consiste nel portare alla porta NOR i segnali a XOR B presenti all'interno dei moduli

Comparatore

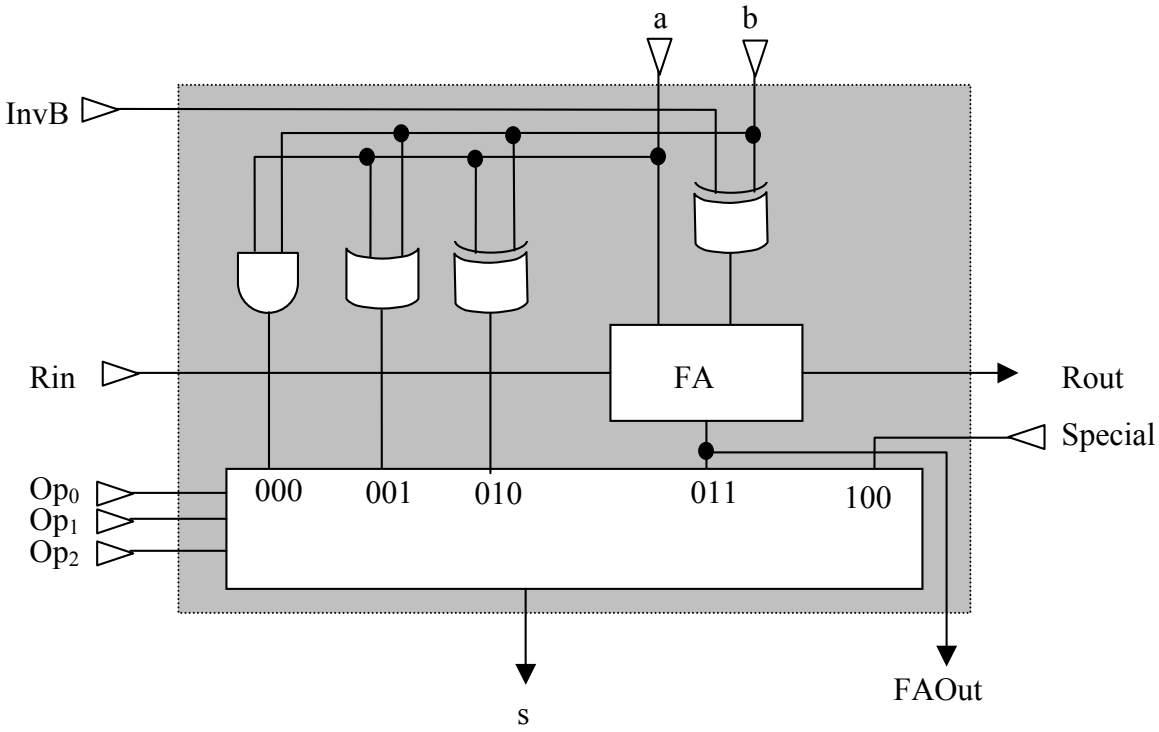
Se si desidera sapere se l'ingresso A è maggiore dell'ingresso B allora si può realizzare la sottrazione tra i due ingressi e valutare successivamente il bit di segno s_3. Alcune CPU riportano esplicitamente il bit di segno del risultato come segnale di stato N (negative).



L'uscita N vale 0 quando il risultato e' positivo (>=0), 1 se negativo. Se impostiamo la sottrazione otteniamo su N il valore 1 se A<B, 0 se A>=B.

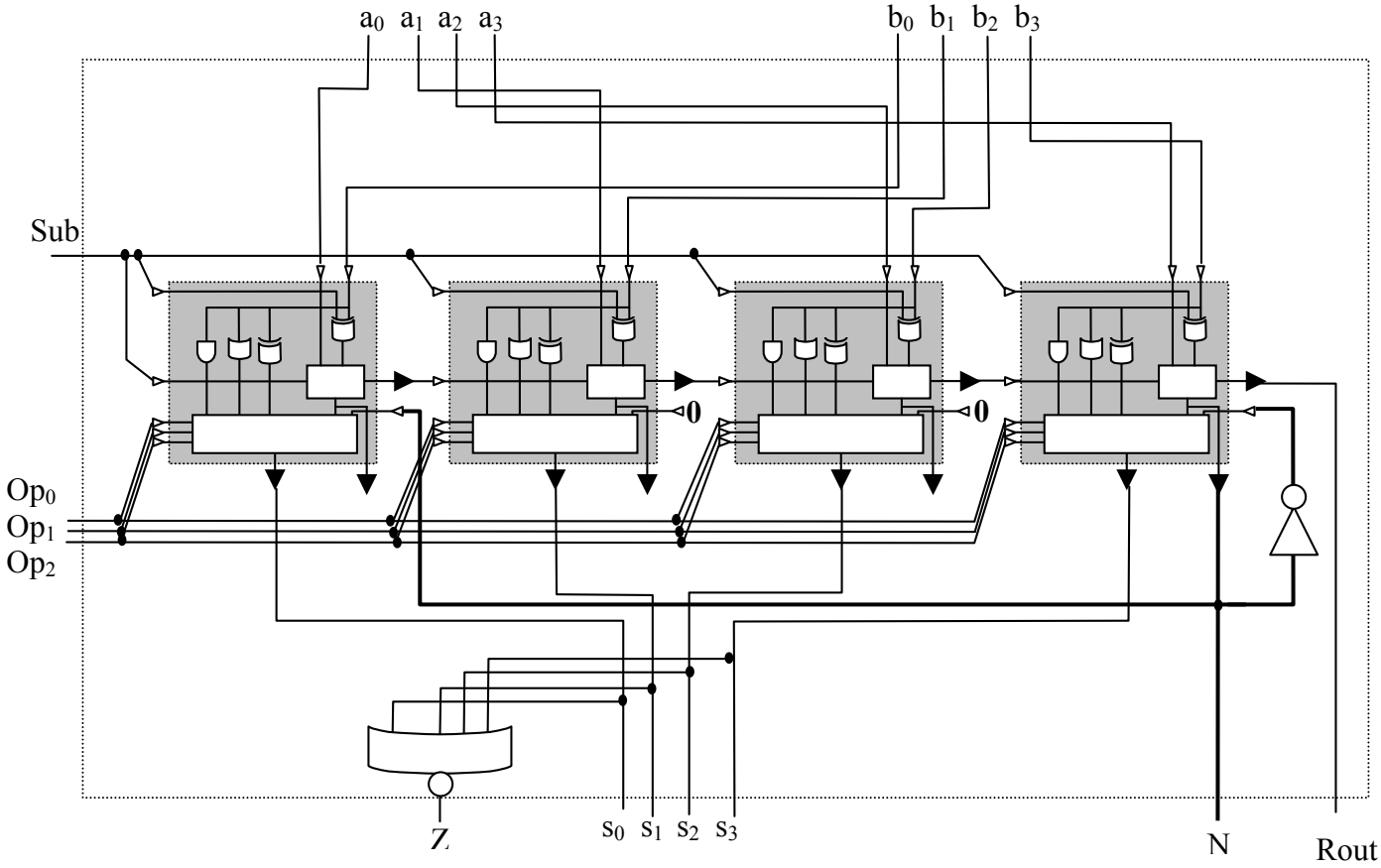
Massimo Marchi – marchi@dsi.unimi.it

Questo segnale N può essere usato per generare delle configurazioni particolari in uscita.
Modifichiamo il modulo base a un bit:



Il multiplexer è in grado di selezionare fino ad 8 ingressi; in questo caso le configurazioni 101,110,111 sono inutilizzate. La configurazione 100 permette ora di mandare verso l'uscita l'ingresso *Special* generato esternamente al modulo. Poiché ci occorre il segno del risultato della sottrazione per valutere la comparazione, aggiungiamo un'uscita FAOut per il risultato del FA, indipendente dalla selezione multiplexer.

Supponiamo ora di voler generare in uscita la parola 1000 se A>=B e la parola 0001 se A<B. Utilizziamo il modulo base modificato. Se A>=B allora N=0 ed in uscita si deve generare 1000, se A<B N vale 1 ed in uscita occorre generare 0001. le due configurazioni coincidono per i bit in posizione 1 e 2, i cui valori possono essere generati staticamente senza l'uso di porte. Il bit 0 si nota che segue il valore di N: quando N vale 0, il bit 0 vale 0; quando N vale 1 il bit vale 1. Il bit 3 ha il comportamento opposto: quando N vale 0, il bit 3 vale 1; quando N vale 1, il bit 3 vale 0. Ne segue che se N può essere usato così com'è per generare il bit 0 e negato per generare il bit 3.

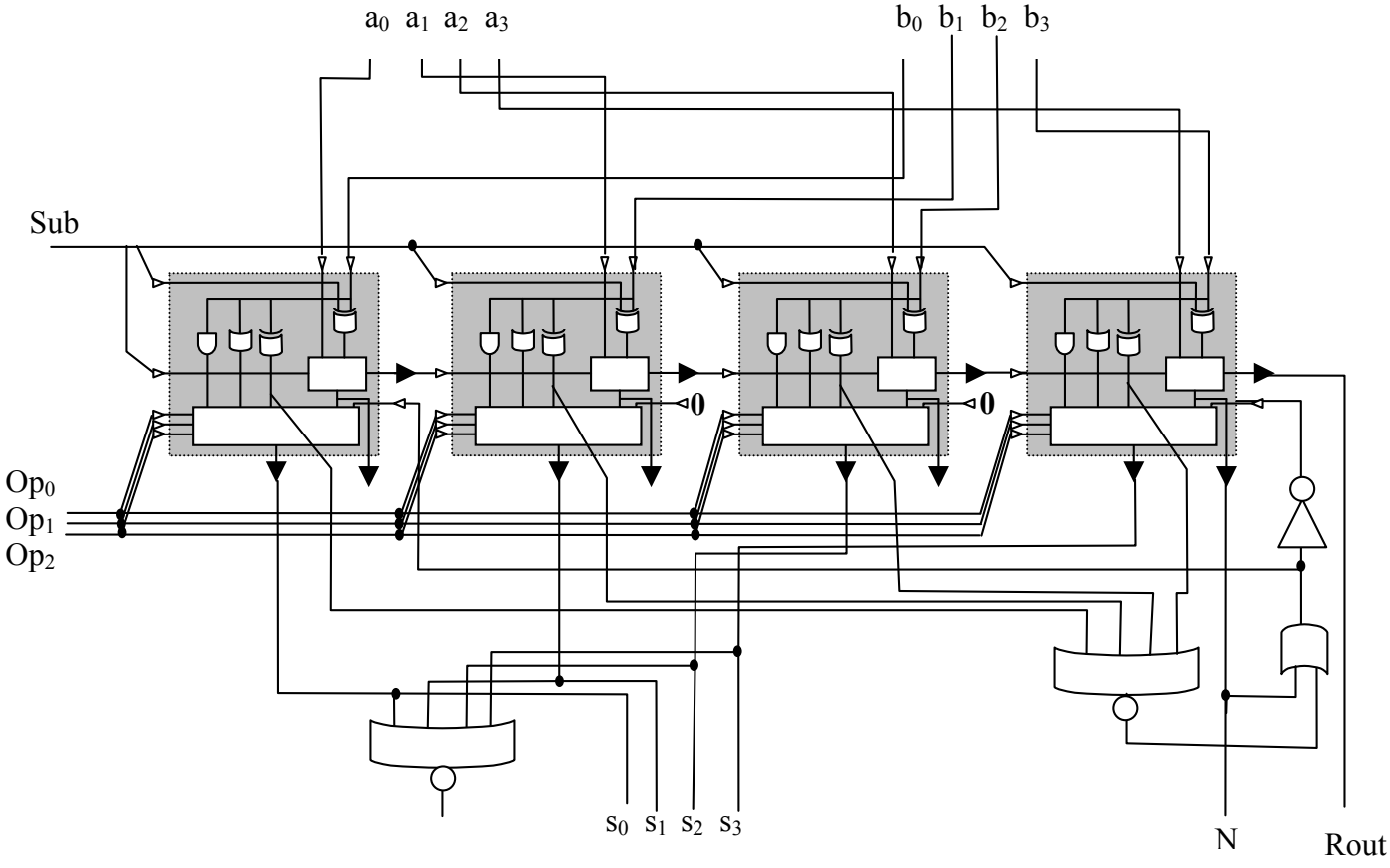


Il modulo computa le seguenti funzioni:

Sub	Op_2	Op_1	Op_0	S
x	0	0	0	S=A and B
x	0	0	1	S=A or B
x	0	1	0	S=A ⊕ B
x	0	1	1	<Rout,S>=A+B
1	0	1	1	<Rout,S>=A-B
1	1	0	0	S=1000 seA>=B S=0001 se A<B

Massimo Marchi – marchi@dsi.unimi.it

Se invece la richiesta è di generare in uscita la parola 1000 se A>B e la parola 0001 se A<=B, allora occorre analizzare meglio il comportamento del circuito quando A=B. L'esame del bit di segno, come nel caso precedente, non è sufficiente ad implementare la richiesta poiché per A=B la risposta è opposta a quella attesa. Occorre correggere il risultato quando A=B. Consideriamo il modulo base modificato ed estraiamo il segnale di uscita della porta XOR in modo da realizzare il confronto bit a bit delle parole A e B. Se A=B allora tutti questi segnali saranno a 0. Se A<>B allora almeno uno di questi segnali sarà a 1. Se portiamo tutti questi segnali in una porta OR a 4 ingressi otteniamo un segnale che varrà 0 se A=B e 1 se A<>B.



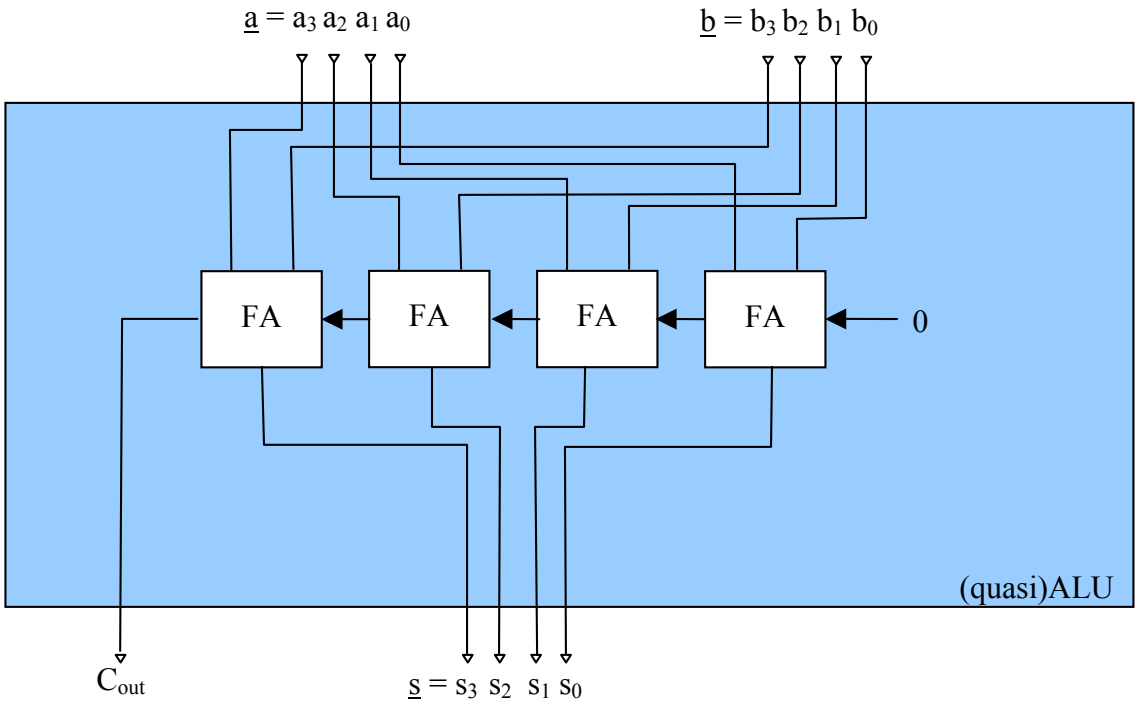
Ora possisamo modificare il segnale N in uscita dall'ultimo modulo affinché abbia il comportamento atteso.Quando A<>B il segnale N si comporta correttamente: se A>B allora vale 0, se A<B allora vale 1. Quando invece A=B assume valore 0 mentre dovrebbe essere 1. Per risolvere il problema possiamo mettere il segnale N in OR con il segnale di uguaglianza che abbiamo creato
Il modulo computa le seguenti funzioni:

Sub	Op_2	Op_1	Op_0	S
x	0	0	0	S=A and B
x	0	0	1	S=A or B
x	0	1	0	S=A ⊕ B
x	0	1	1	<Rout,S>=A+B
1	0	1	1	<Rout,S>=A-B
1	1	0	0	S=1000 seA>B S=0001 se A<=B

Massimo Marchi – marchi@dsi.unimi.it

***Another Yet* ALU: Un esempio a 4 bit visto in termini di parola**

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** [3] e restituisce in uscita 4 bit di risultato **s** ed il riporto dell'ultimo stadio $\mathbf{C_{out}}$ (*Carry Out*):



Gli ingressi come l'uscita sono visualizzati con in bit più significativo, il bit di segno, a sinistra. Per semplicità, il sommatore è riportato con gli stadi invertiti. Lo stadio più a sinistra elabora ora i bit più significativi.

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.

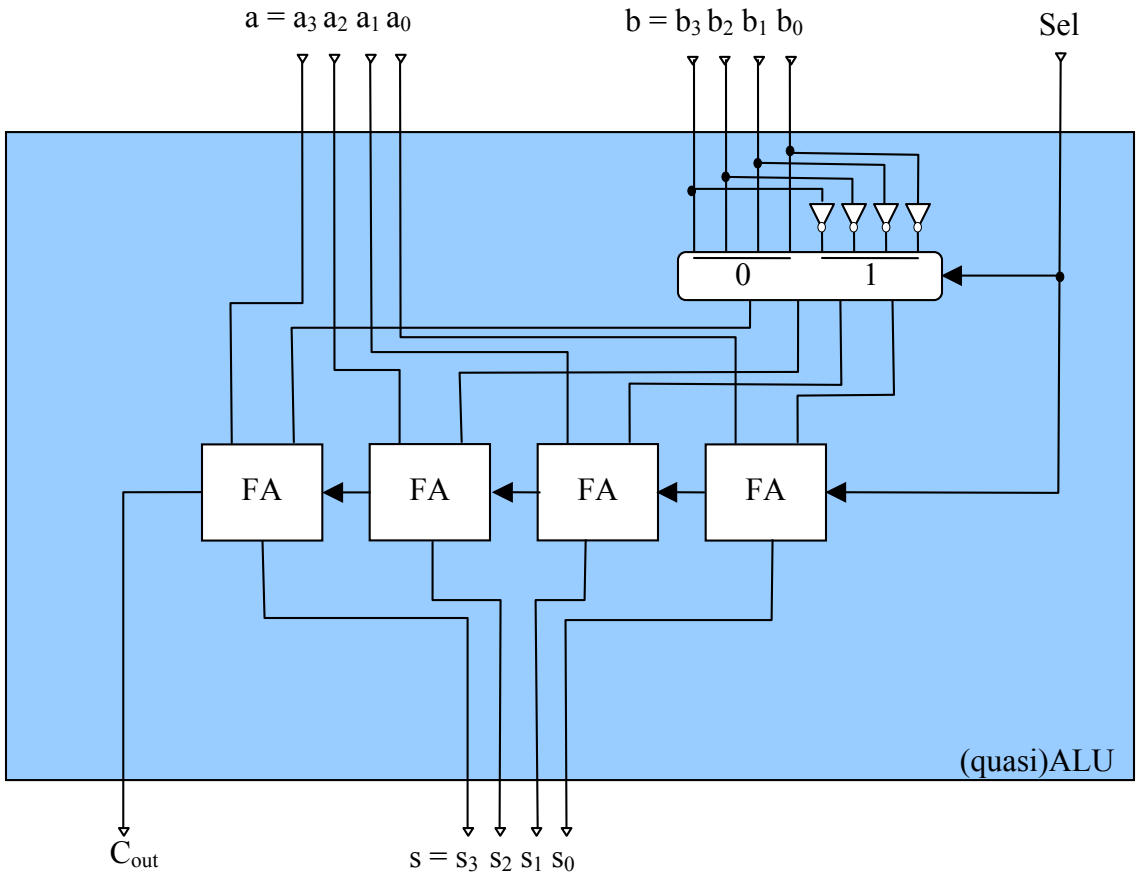
[3] Indichiamo con il sottolineato i vettori di segnali.

Supponiamo ora di voler aggiungere alla (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:

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

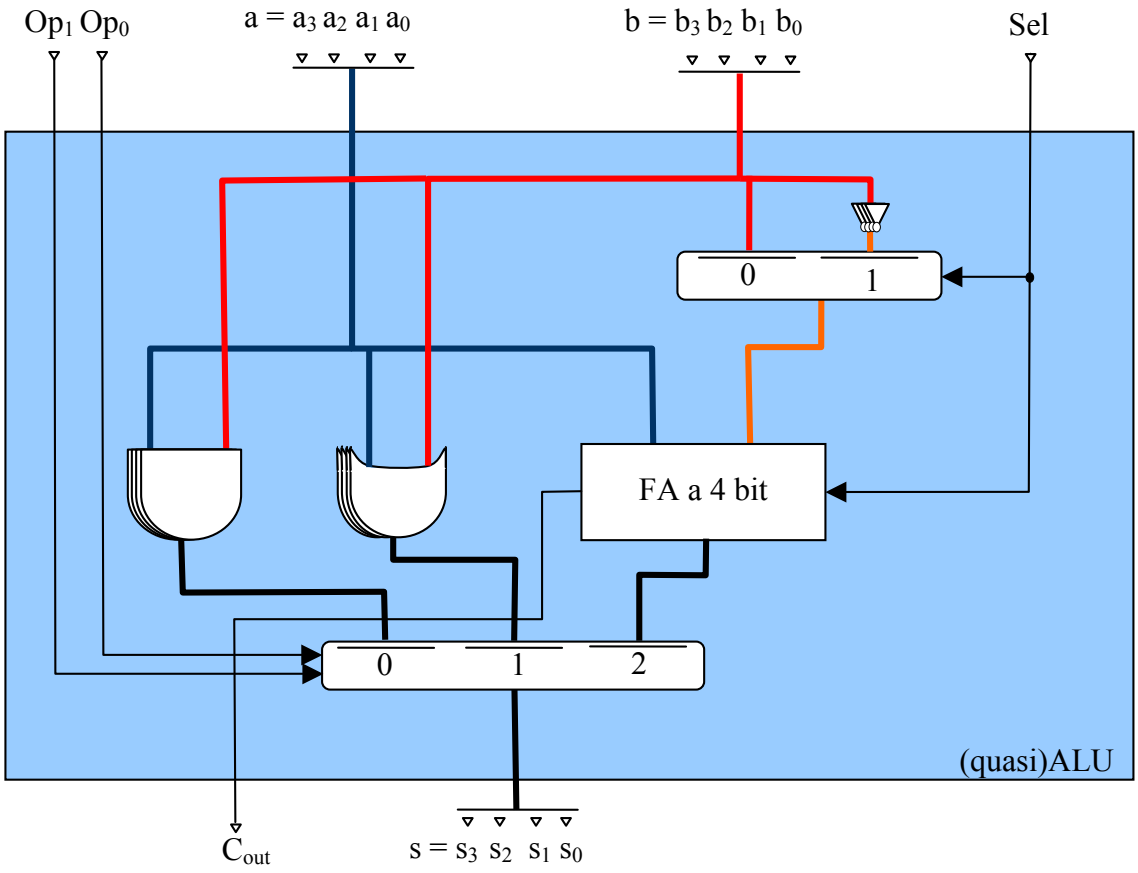
Porre il riporto in ingresso al primo stadio di un sommatore uguale a 1 equivale a sommare 1 al risultato finale. Ne segue che possiamo riorganizzare il circuito sommatore per eseguire all'occorrenza anche la sottrazione.

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.

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:

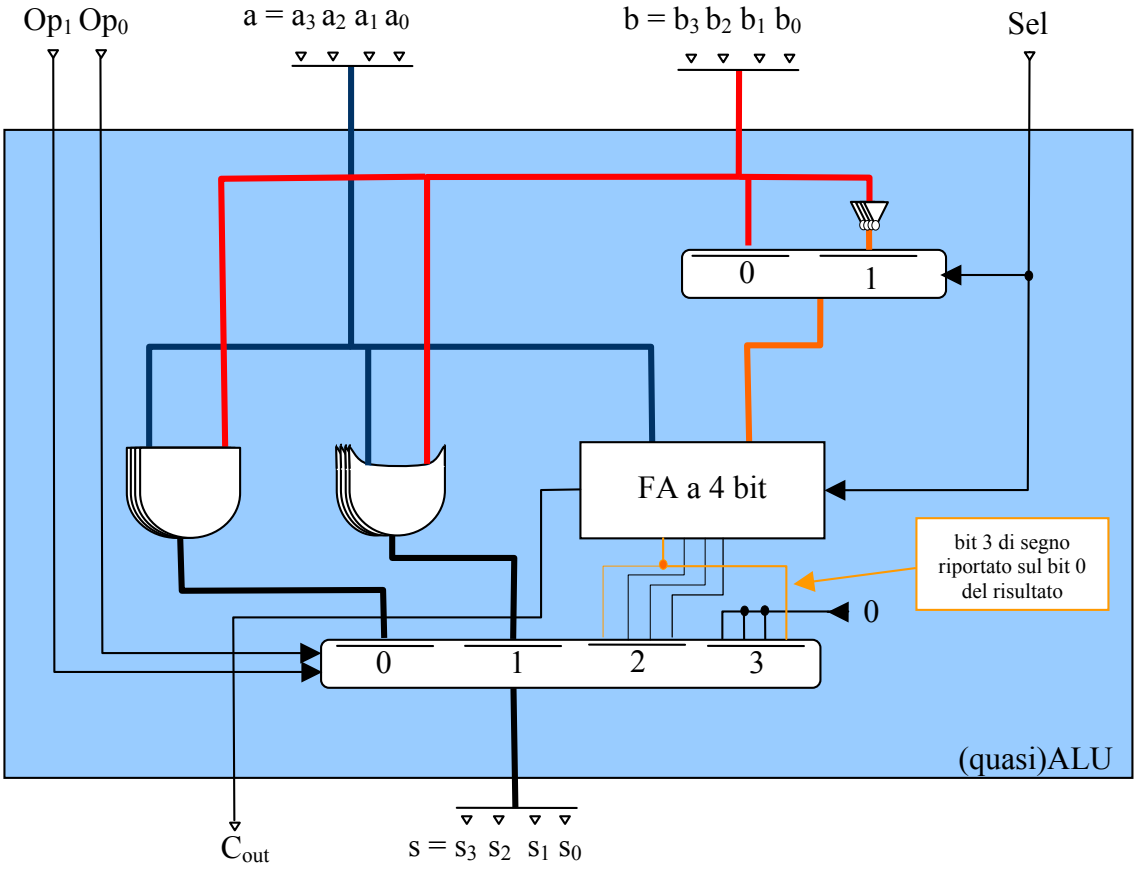
1 se **a** < **b** , **0** se **a** >= **b**

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 **Op_1=1** e **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**).

Op_1	Op_0	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