

Architettura degli Elaboratori

circuiti combinatori: ALU

slide a cura di Salvatore Orlando, Andrea Torsello, Marta Simeoni

ALU

ALU (Arithmetic Logic Unit)

- circuito combinatorio all'interno del processore per l'esecuzione di **istruzioni macchina** di tipo **aritmetico/logiche**

Esempi di istruzioni aritmetico/logiche (e di confronto)

– and \$2, \$3, \$4	# \$2 = \$3 and \$4
– or \$2, \$3, \$4	# \$2 = \$3 or \$4
– add \$2, \$3, \$4	# \$2 = \$3 + \$4
– sub \$2, \$3, \$4	# \$2 = \$3 - \$4
– slt \$2, \$3, \$4	# if (\$3<\$4) \$2=1 else \$2=0
– bne \$4,\$5,Label	# Se \$4 ≠ \$5, prossima istr. caricata dall'indirizzo Label
– beq \$4,\$5,Label	# Se \$4 = \$5, prossima istr. caricata dall'indirizzo Label

ALU

Quindi l'ALU deve essere in grado di eseguire:

- somme (add)
- sottrazioni (sub)
- istruzioni di confronto (slt, beq, bne)
- funzioni logiche (and, or)

Vediamo come viene costruita...

Addizionatore

L'ALU deve includere un addizionatore per realizzare le somme di numeri interi in complemento a 2

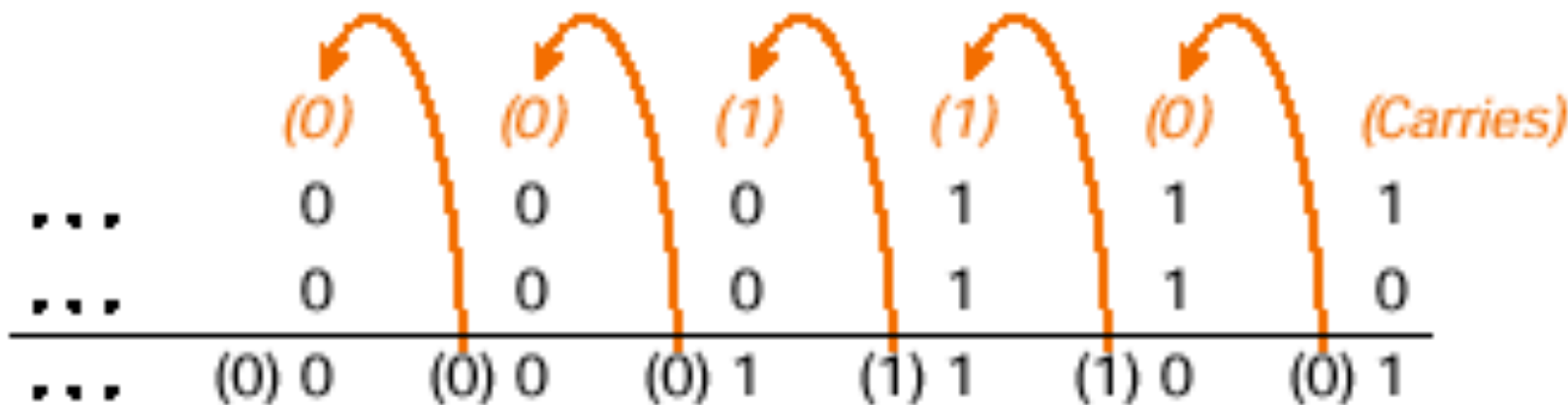
Potremmo costruire un unico circuito combinatorio che implementa un addizionatore a n bit

- dati $2*n$ input: $A_{n-1} \dots A_0 \quad B_{n-1} \dots B_0$
- $n+1$ diverse funzioni di output: $Rip \quad C_{n-1} \dots C_0$
- solo due livelli di logica, ma con porte AND e OR con molti input
 - fan-in delle porte molto elevato (non ammissibile)

Addizionatore

Soluzione di compromesso, basata su una serie di **1-bit adder** collegati in sequenza

- il segnale deve attraversare più livelli di logica
- porte con fan-in limitato (ammissibile)
- circuito che usa lo stesso metodo usato *dall'algoritmo carta e penna* a cui siamo abituati



Addizionatore a singolo bit

La tabella di verità dell'*addizionatore a singolo bit*

A	B	Rip	Sum	Rip_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

Sum

Rip	AB			
	00	01	11	10
0		1		1
1	1		1	

$$\text{Sum} = \sim A \sim B \text{ Rip} + \sim A B \sim \text{Rip} + A B \text{ Rip} + A \sim B \sim \text{Rip}$$

Rip_out

Rip	AB			
	00	01	11	10
0			1	
1		1	1	1

$$\text{Rip_out} = \text{Rip} B + A B + \text{Rip} A$$

Addizionatore a singolo bit

La funzione Sum non può essere semplificata

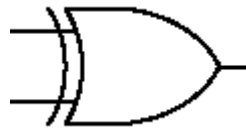
- ben 4 porte AND

La costruzione di un 1-bit adder diventa più semplice impiegando porte XOR

- funzione logica che vale 0 (F) se entrambi gli ingressi sono uguali, ovvero entrambi 0 (F) o entrambi 1 (T)
- esempio di **or esclusivo** (XOR) nel linguaggio comune:

“o rimango a casa oppure vado al cinema”

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0



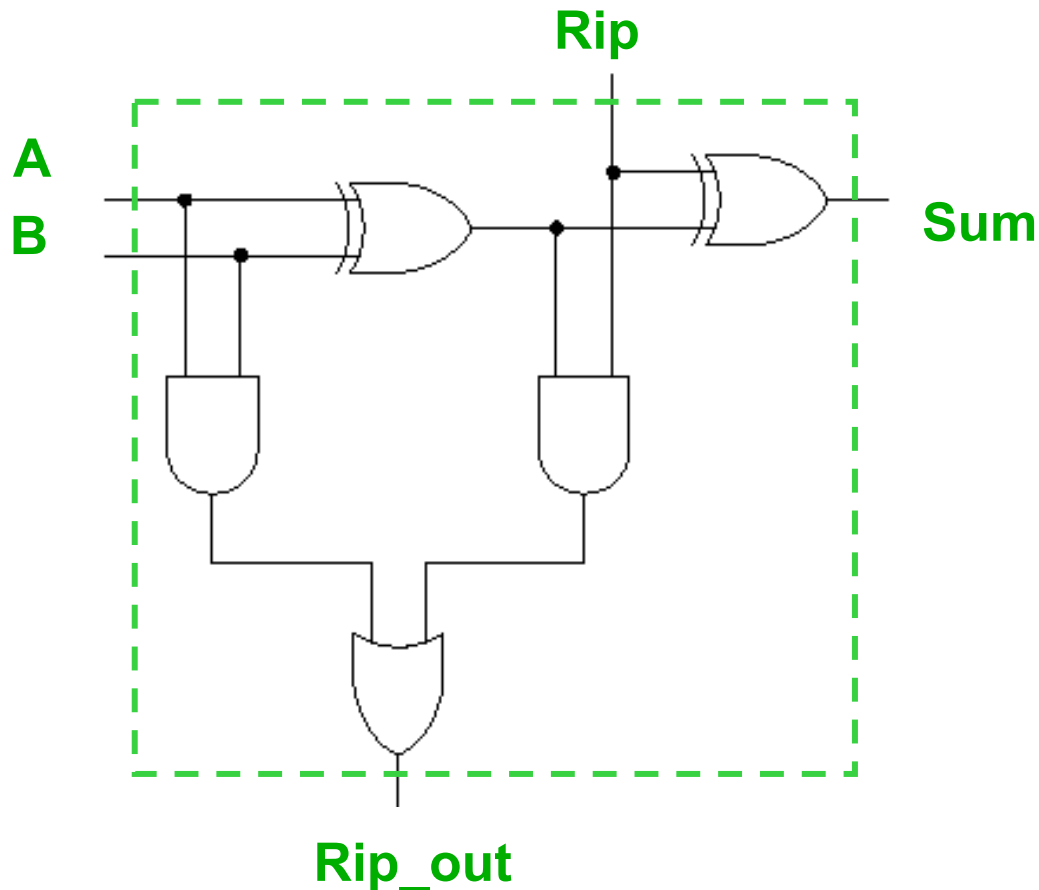
Porta XOR

$A \text{ xor } B$ esprime esattamente la somma di A e B

1-bit adder usando porte XOR

Visto che la somma tra una coppia di bit A e B si esprime con $A \oplus B$ abbiamo:

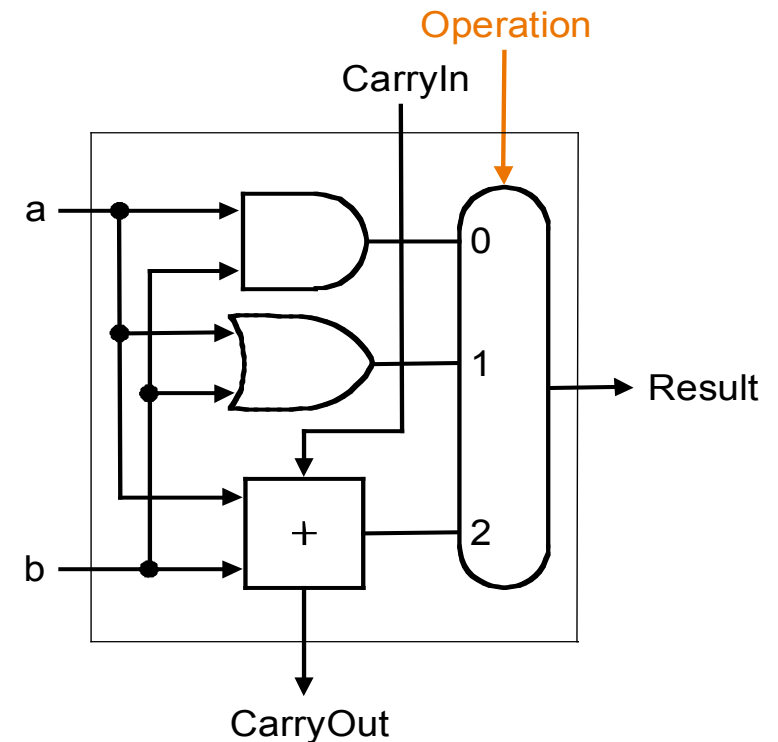
- $\text{Sum} = A \oplus B \oplus \text{Rip}$
- $\text{Rip_out} = A B + (A \oplus B) \text{Rip}$



1-bit ALU

1-bit ALU usata per eseguire le istruzioni macchina seguenti:

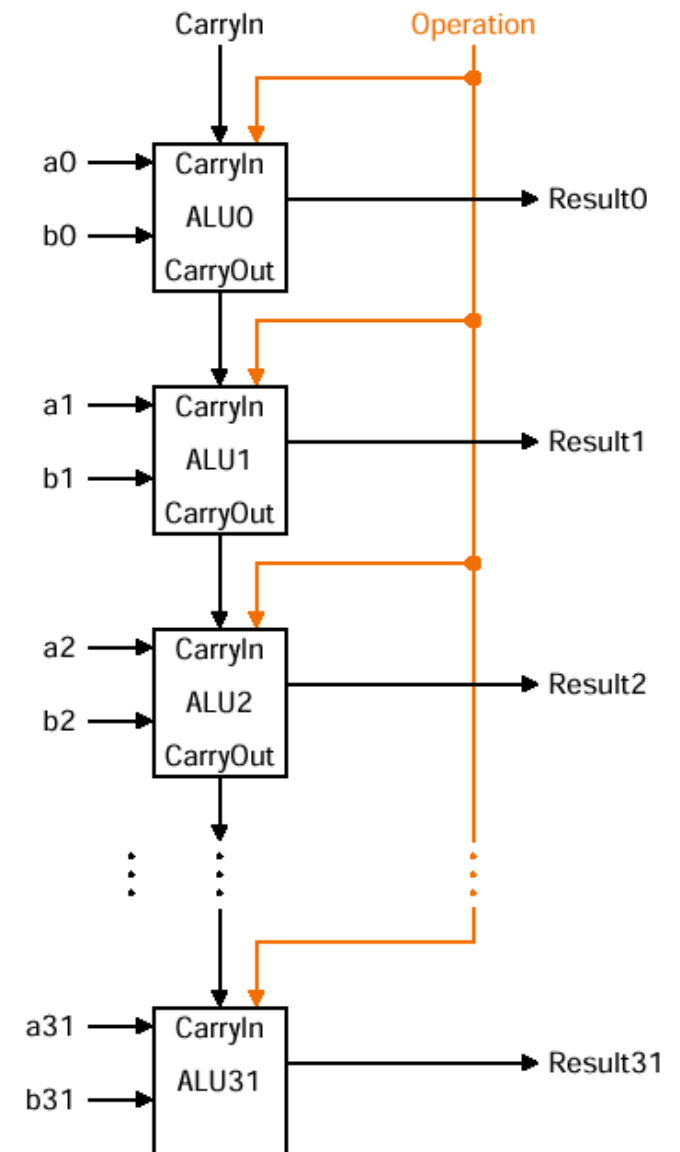
- **and** \$2, \$3, \$4
- **or** \$2, \$3, \$4
- **add** \$2, \$3, \$4
- **Operation** è un **segnale di controllo** a 2 bit usato per selezionare quale ingresso del multiplexer deve uscire
 - determina il tipo di operazione che l'ALU deve eseguire
- l'ALU è la tipica componente che fa parte del *Datapath* (**Parte operativa**) del processore
- La **Parte Controllo** comanda l'esecuzione delle varie istruzioni
 - settando opportunamente i segnali di controllo dell'ALU (e delle altre componenti della **Parte operativa**)



32-bit ALU

32-bit ALU

- catena di 1-bit ALU con **propagazione del Carry**
- segnali di controllo per determinare l'operazione che l'ALU deve eseguire
 - **Operation**: propagato a tutte le 1-bit ALU



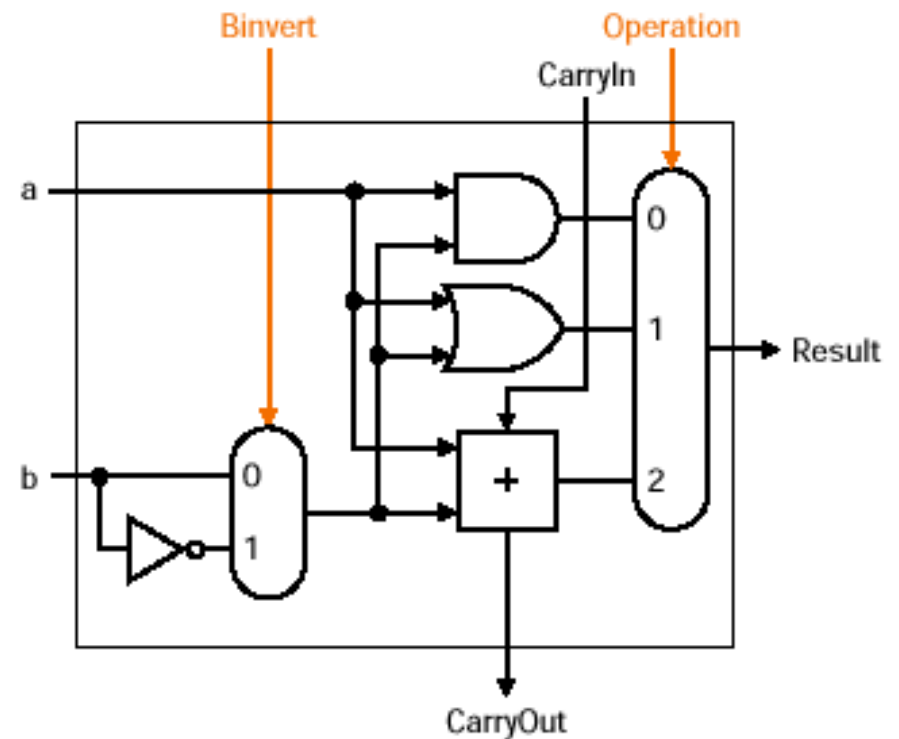
Inversione e sottrazione

L'1-bit ALU precedente può essere *resa più complessa* per poter eseguire:

- sub \$2, \$3, \$4

Operazione di sottrazione:

- $\$2 = \$3 - \$4$ trasformata in:
 $\$2 = \$3 + (-\$4)$
- $(-\$4)$ significa che bisogna prima determinare l'opposto (il complemento a 2) del *numero signed* contenuto in \$4
 - 1. effettuando l'inversione (complemento a 1 bit-a-bit)
 - 2. sommando 1
- l'ALU deve quindi possedere i circuiti predisposti per
 - **invertire** il secondo operando e **sommare 1**
 - per sommare 1, basta porre ad 1 il *carry-in* dell'ALU



Binvert	Operation	Istruzione
0	00	and
0	01	or
0	10	sum
1	10	sub

Istruzioni di confronto

`slt $2, $3, $4` (*set less than*)

– $\$2=1$ se è vero che $\$3 < \4

– $\$2=0$ altrimenti

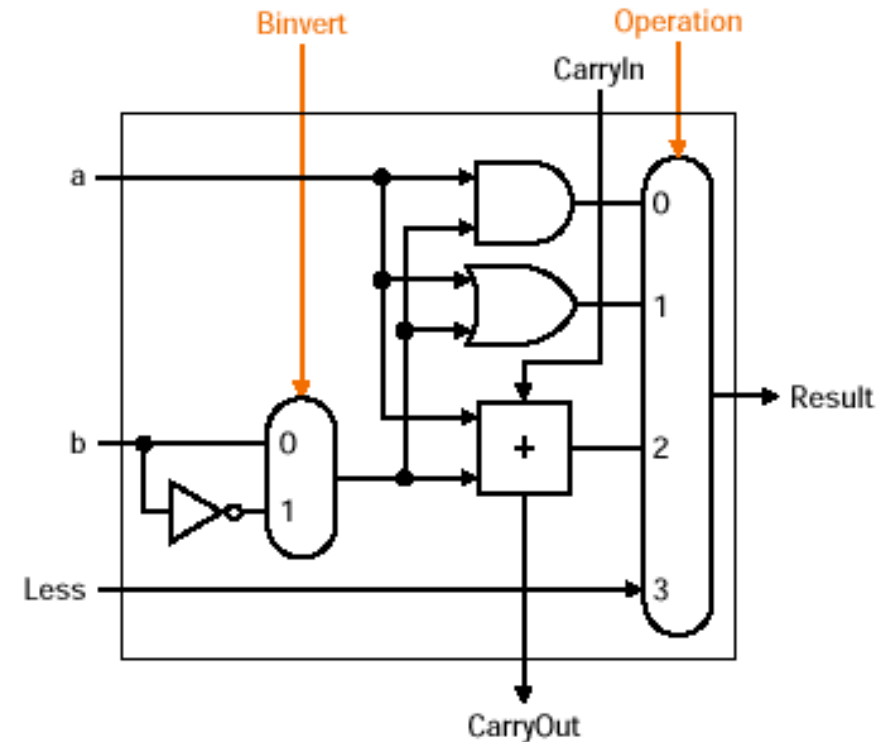
Se $\$3 < \4 allora $\$3 - \$4 < 0$

Quindi, per effettuare il confronto, possiamo semplicemente:

- **sottrarre** e controllare il **bit di segno**
- se **non c'è overflow** durante la sottrazione
 - il valore del **bit di segno** del risultato della sottrazione può essere semplicemente assegnato al **bit meno significativo** dei 32 bit in output
 - tutti gli altri bit in output devono essere posti a 0

Tutte le **1-bit ALU** devono quindi avere un **ingresso in più**

- l'input **Less**, che verrà posto a 0 o a 1 sulla base del risultato dell'istruzione `slt`



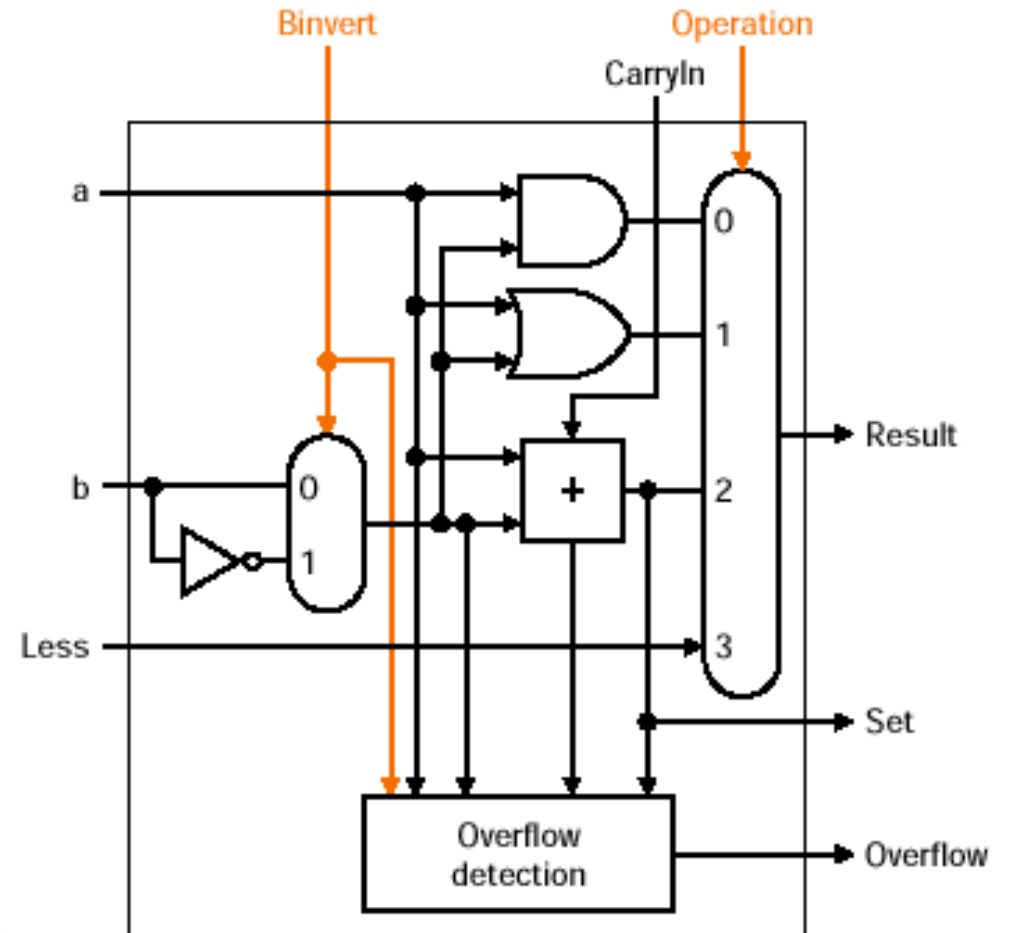
Istruzioni di confronto

La 1-bit ALU che determina la cifra più significativa è più complessa poiché

- deve controllare l'**overflow**
- deve fornire, come ulteriore output, il bit di segno del risultato della sottrazione (**Set**)
 - questo per permettere l'implementazione di **slt**
 - **Set** deve essere ridiretto verso la 1-bit ALU che fornirà in output il bit meno significativo del risultato

Il blocco che controlla l'overflow lo fa sulla base

- del tipo di operazione (**sum** o **sub**), identificata tramite **Binvert**
- i segni degli operandi
- il segno del risultato



Alu complessiva

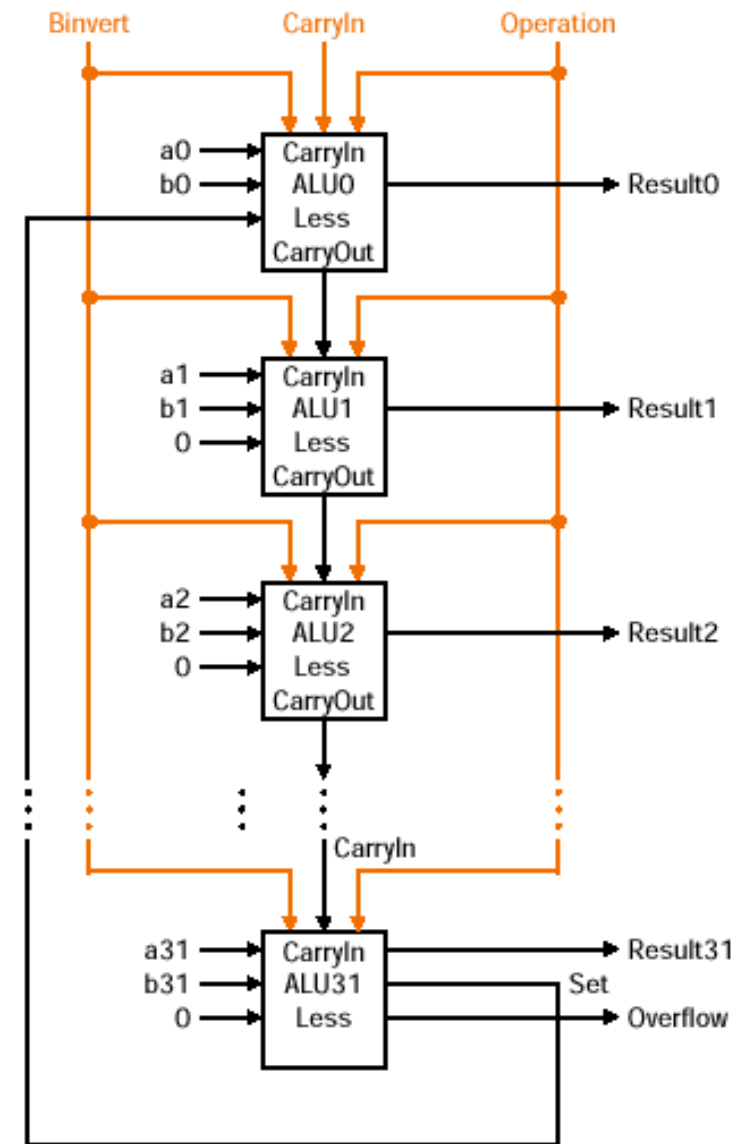
Output **Set** dell'**ultima** 1-bit ALU viene ridiretto sull'input **Less** della **prima** 1-bit ALU

Tutti i bit **Less** delle varie 1-bit ALU (eccetto la prima) vengono posti a **0**

Segnali di controllo:

- **Binvert** e **Carryin** vengono entrambi *asserted* (*affermati*) per sottrarre (**sub** e **slt**)
- I bit di **Operation** sono posti a **11** per far passare in output l'ultimo bit in ingresso ai Multiplexer 4:1

Binvert	Carryin	Operation	Istruzione
0	0	00	and
0	0	01	or
0	0	10	sum
1	1	10	sub
1	1	11	slt



slt e overflow

Il circuito proposto per implementare l'ultima 1-bit ALU della catena

- potrebbe **NON FUNZIONARE** per il **slt** nel caso di overflow
- non è ottimale per quanto riguarda l'overflow

Caso di malfunzionamento relativo a **slt**:

- **slt \$2, \$3, \$4**
- se **\$3 > 0** e **\$4 < 0**
 - potremmo concludere direttamente che è vero che **\$3 > \$4** \Rightarrow **\$2 = 0**
 - se invece **sottraiamo** per implementare **slt**, finiamo per sommare due numeri positivi (**\$3 + (-\$4)**)
 - » potremmo avere **overflow**, ottenendo così un bit di segno (**Set**) non valido (uguale a 1, invece che uguale a 0)
- un ragionamento analogo può essere fatto nel caso in cui **\$3 < 0** e **\$4 > 0**

Circuito per slt

Set deve essere determinato in modo da evitare il malfunzionamento precedente, relativo a un overflow non voluto

Siano

- $a = a_{31} \dots a_0$ e $b = b_{31} \dots b_0$ i due numeri da confrontare
- $res = res_{31} \dots res_0$ il risultato degli 1-bit adder
- $c = c_{31} \dots c_0$ il risultato della ALU, che potrà solo essere:
 $0 \dots 01$ oppure $0 \dots 00$

Se $a \geq 0$ e $b < 0$, allora $a > b$, e possiamo porre direttamente **Set = 0**

Se $a < 0$ e $b \geq 0$, allora $a < b$, e possiamo porre direttamente **Set = 1**

- nei 2 casi di sopra, all'ALU viene comandato di eseguire una sottrazione, ma l'eventuale OVERFLOW verrà *ignorato*

Se $a > 0$ e $b > 0$, oppure se $a < 0$ e $b < 0$, allora

- possiamo considerare il risultato della sottrazione, e possiamo porre **Set = res_{31}**
- in questi casi non si può verificare OVERFLOW, per cui res_{31} conterrà il bit di segno corretto

Circuito corretto per slt

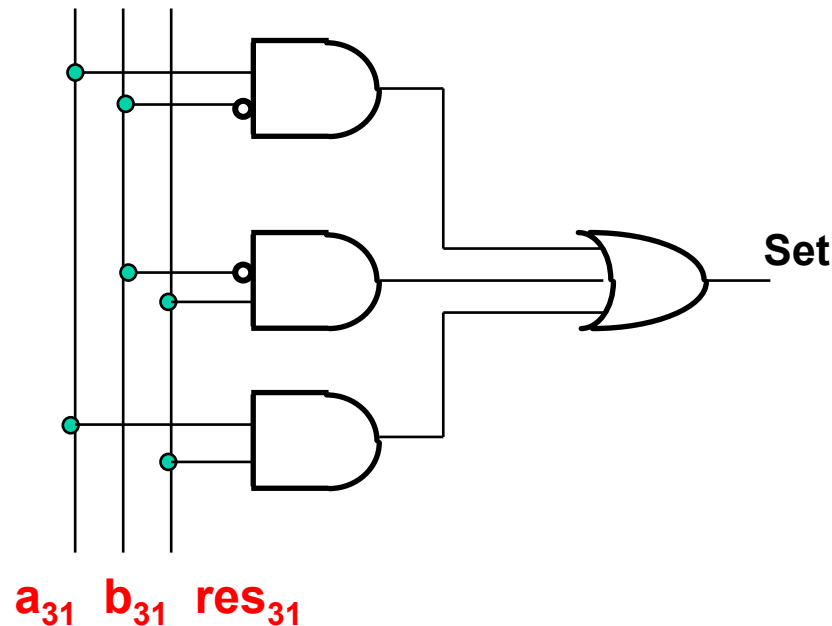
a_{31}	b_{31}	res_{31}	Set
0	0	0	0
0	0	1	1
1	1	0	0
1	1	1	1
0	1	X	0
1	0	X	1

← $a \geq 0$ e $b < 0$

← $a < 0$ e $b \geq 0$

	a_{31}	b_{31}		
res_{31}	00	01	11	10
0				1
1	1		1	1

$$\text{Set} = a_{31} \sim b_{31} + \sim b_{31} res_{31} + a_{31} res_{31}$$



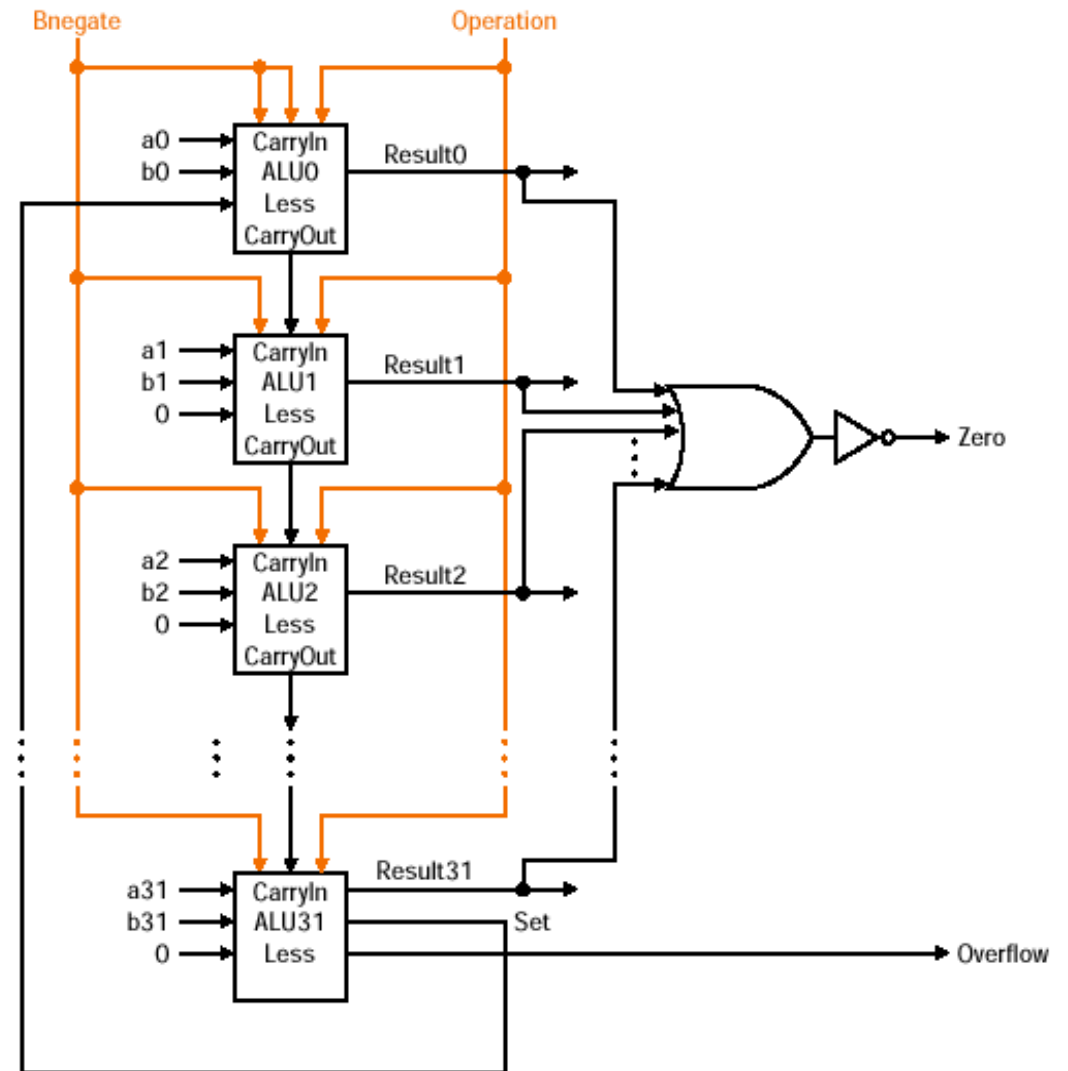
Alu finale

Abbiamo risparmiato un bit di controllo

- **Bnegate** al posto di:
(**Binvert**, **Carryin**)
- Nell'ALU precedente, infatti,
(**Binvert**, **Carryin**) venivano sempre *asserted* o *deasserted* assieme

Abbiamo ulteriormente specializzato l'ALU per l'esecuzione delle istruzioni di *branch condizionato*

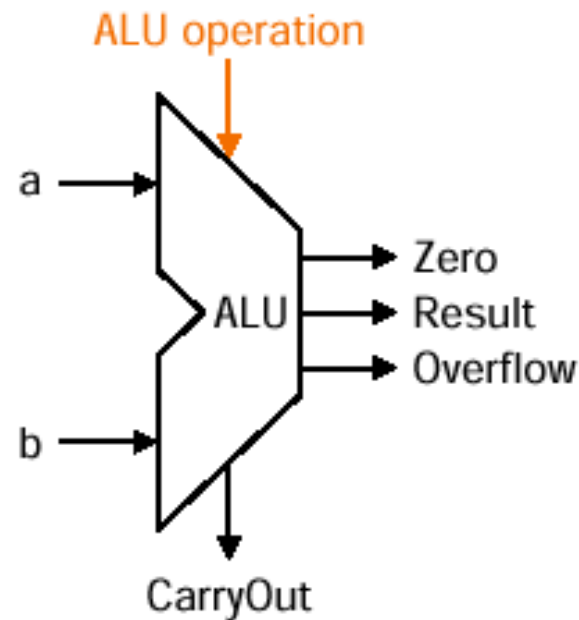
- **beq** e **bne**
- devo controllare se
 - **a==b** oppure se **a != b**
- posso *comandare* alla ALU di sottrarre, e controllare se
 - **a-b=0** oppure se **a-b != 0**
- **Zero=1** \Leftrightarrow **a-b==0** (**a==b**)



Componente combinatoria: ALU

Simbolo usato per rappresentare la componente ALU nel progetto della CPU.

Alu operation comprende i segnali di controllo Bnegate e Operation.



ALU e Somma veloce

Considerazioni sulla velocità dell'ALU nell'eseguire la somma:

- l'ingresso CarryIn di ogni 1-bit adder dipende dal valore calcolato dall'1-bit adder precedente
- il bit più significativo della somma deve quindi attendere 32 volte il tempo di attraversamento del segnale attraverso i vari sommatorei ⇒ LENTO

... ci sono metodi per velocizzare il calcolo del riporto?

- sì, il **metodo del Carry Lookahead**
- si cerca di far passare il segnale per un numero minore di porte, per anticipare il riporto
- non lo vediamo a lezione, lo trovate comunque sul libro di testo (Appendice C per la terza edizione; appendice B per la seconda edizione)