Le operazioni aritmetiche e la costruzione di una ALU

Aritmetica binaria in codifica naturale: somma

- L'operazione di addizione tra numeri rappresentati in base 2 segue le seguenti regole di base:

0	+	0	+	1	+	1	+
0	=	1	=	0	=	1	=
0		1		1		(1)	0

- Consideriamo la somma di due valori in codifica binaria su 8 bit:
  - $X = 7_{10} = 00000111_2$
  - $Y = 6_{10} = 00000110_2$

0	0	0	0	0	1	1	0
0	0	0	0	0	1	1	1
0	0	0	0	0	1	1	0
(0) 0	(0) 0	(0) 0	(0) 0	(0) 1	(1) 1	(1) 0	(0) 1

- Per rappresentare correttamente il risultato di una somma su n bit sono necessari n+1 bit

Aritmetica binaria in codifica naturale: somma

- In generale si vuole operare su parole della stessa lunghezza, quindi la somma tra operandi di n bit deve essere rappresentata su n bit
- Questo implica che non tutte le operazioni producono un risultato corretto su n bit
- La correttezza è data dal bit di riporto più a sinistra, c_{n+1}

  $C_{n+1} = 0$ -> operazione corretta su n bit

  $C_{n+1} = 1$ -> overflow

Somma in codifica binaria naturale - Esempio

- Es. 1: Si vuole calcolare la somma 8 + 11 lavorando con parole di 4 bit:

1	0	0	0	
1	0	0	0	+
1	0	1	1	=
0	0	1	1	

- Il bit di riporto più a sinistra è 1. Quindi l'operazione genera un errore di overflow.

- Es. 2: Si vuole calcolare la somma 8 + 11 lavorando con parole di 5 bit:
- Il bit di riporto più a sinistra è 0. Quindi il risultato è corretto su 5 bit

0	1	0	0	0	
0	1	0	0	0	+
0	1	0	1	1	=
1	0	0	1	1	

Sottrazione in codifica binaria naturale

- L'operazione di sottrazione tra numeri
- rappresentati in base 2 segue le
- seguenti regole di base:

0	-	0	-	1	-	1	-
0	=	1	=	0	=	1	=
0		(1) 1		1		0	

- Consideriamo la sottrazione di due valori in codifica binaria su 8 bit:
-

$$X = 13_{10} = 00001101_2$$
$$Y = \ \ 6_{10} = 00000110_2$$

0	0	0	0	0	1	1	0
0	0	0	0	0	1	1	0
(0) 0	(0) 0	(0) 0	(0) 0	(0) 0	(1) 1	(1) 1	(0) 1

- In codifica naturale, il risultato è corretto se il bit di prestito più a sinistra è 0
  - Il problema sta nel fatto che la codifica naturale non può rappresentare i valori negativi
  - Si osservi che quando il minuendo è maggiore del sottraendo il risultato è sempre corretto su n bit

Sottrazione in codifica binaria naturale - Esempi

- Es. 1: Si vuole calcolare la sottrazione 10 - 3 lavorando con parole di 4 bit:
- Il risultato è corretto perché il bit di prestito più a sinistra è 0

0	1	1	1	
	1	0	1	0 -
	0	0	1	1 =
	0	1	1	1

- Es. 2: Si vuole calcolare la sottrazione 3 - 5 lavorando con parole di 4 bit:
- Il risultato non è corretto perché il bit di prestito più a sinistra è 1

1	1	0	0	
	0	0	1	1 -
	0	1	0	1 =
	1	1	1	0

Somma algebrica in codifica binaria Modulo e Segno

- È necessario calcolare separatamente il segno ed il modulo del risultato: 4 casi
  - X>0, Y>0

    $X_{2,MS} + Y_{2,MS} = |X| + |Y| = +(|X|+|Y|)_{2,MS}$

  - X>0, Y<0 (simmetricamente per Y>0, X<0)
    - Se |X|>|Y|

      $X_{2,MS} + Y_{2,MS} = X - |Y|$

    - Se |X|<|Y|

      $X_{2,MS} + Y_{2,MS} = -(|Y| - X)$

  - X<0, Y<0

    $X_{2,MS} + Y_{2,MS} = -|X|-|Y| = -(|X|+|Y|)_{2,MS}$

Somma in MS-Esempi

- Es. 1: si calcoli la somma 3 + 4 operando in MS su 4 bit

```
  0  0  0
     0  1  1  +
     1  0  0  =
  ----------
     1  1  1
```

imponendo poi il segno positivo si ottiene $0111_{2,MS} = 7_{10}$

- Es.2: si calcoli la somma 3+(-4) operando in MS su 4 bit

Segno discorde, il risultato sarà negativo e calcoliamo –(4 -3)

```
  0  1  1
     1  0  0  -
        1  1  =
  ----------
     0  0  1
```

al quale aggiungiamo il segno negativo, quindi otteniamo $1001_{2,MS} = -1_{10}$

Somma algebrica in complemento a 1

- Si analizzano i 4 casi:
  - X>0, Y>0:
    $X_{2,C1} + Y_{2,C1} = X + Y = (X + Y)_{2,C1}$
  - X>0, Y<0: si esprime Y secondo la definizione di complemento a 1 e si esegue la somma (X<0, Y>0 è simmetrico)
    $X_{2,C1} + Y_{2,C1} = X + (2^n - 1 - |Y|) = \mathbf{2^n - 1 - (|Y| - X)}$

Se (|Y| – X) >= 0, cioè |Y|>=X, otteniamo la rappresentazione in complemento a 1 della quantità positiva (|Y|-X), ovvero della quantità negativa (X-|Y|)

ES: X=3, Y=-6, calcoliamo 3+(-6) su 4 bit
- si calcola $(-6)_{10} = 1001_{2,C1}$
- il riporto più a sinistra, 0, ci dice che essendo X<|Y| il risultato è corretto
- infatti $1100_{2,C1} = -3_{10}$

0	0	1	1	
0	0	1	1	+
1	0	0	1	=
1	1	0	0	

Somma algebrica in complemento a 1

$X_{2,C1} + Y_{2,C1} = 2^n - 1 - (|Y| - X)$

Se $(|Y| - X) < 0$, cioè $|Y| < X$

$2^n - 1 - (|Y| - X) = 2^n - 1 + (X - |Y|)$

Dove $(X - |Y|) >= 0$, ma operando in modulo 2^n, e considerando $Y<0$

$2^n - 1 + (X - |Y|) = -1 + (X - |Y|) = \mathbf{X+Y - 1}$

Cioè il risultato cercato (X+Y) a meno di una unità, questo vuol dire che quando |Y|<X dobbiamo aggiungere 1 al risultato

ES: X=6, Y=-3 calcoliamo X+Y=6+(-3)

- si calcola $(-3)_{10} = 1100_{2,C1}$ e
- si esegue la somma $(2)_{10} = 0010_{2,C1}$

- Il riporto più a sinistra 1 indica che dobbiamo aggiungere 1 al risultato per ottenere il valore cercato

1	1	0	0	
0	1	1	0	+
1	1	0	0	=
0	0	1	0	+
			1	=
0	0	1	1	

Somma algebrica in complemento a 1

- X<0, Y<0:
  sia X sia Y devono essere espressi secondo la definizione di complemento a 1

$X_{2,C1} + Y_{2,C1} = (2^n - 1 - |X|) + (2^n - 1 - |Y|) = \mathbf{2^n + [2^n - 1 - (|X| + |Y|)] - 1}$

Dato che le operazioni avvengono in modulo 2^n

$2^n + [2^n - 1 - (|X| + |Y|)] - 1 =$

$= [2^n - 1 - (|X| + |Y|)] - 1 = [-(|X| + |Y|)]_{2,C1} - 1$

Essendo necessariamente |X|+|Y| >= 0 l'espressione ottenuta rappresenta la codifica in complemento a 1 del valore |X|+|Y|-1,

Per ottenere il risultato corretto è necessario anche in questo caso aggiungere 1

ES: X=-2, Y=-3, calcoliamo X+Y= -2 + (-3)

1	1	0	0		
	1	1	0	1	+
	1	1	0	0	=
	1	0	0	1	+
				1	=
	1	0	1	0	

Somma in complemento a 1 - Overflow

- La somma in C1 si svolge secondo la seguente regola:

1. Si calcola il risultato della somma su n bit
2. **Al risultato si somma il riporto in posizione più a sinistra**
3. Si può verificare un overflow solo se gli operandi hanno lo stesso segno:
   - Operandi con segno negativo e risultato positivo
   - Operandi con segno positivo e risultato negativo

Somma algebrica in complemento a 1 – esempi di Overflow

- Es.3: si calcoli la somma 3+6 operando in su 4 bit

0	1	1	0	
	0	0	1	1 +
	0	1	1	0 =
	1	0	0	1

Il segno del risultato negativo con operandi positivi è indice di overflow

- Es.4: si calcoli la somma (-4)+(-6) operando in su 4 bit

1	0	1	1	
	1	0	1	1 +
	1	0	0	1 =
	0	1	0	0 +
				1 =
	0	1	0	1

Il segno del risultato positivo con operandi negativi è indice di overflow

Somma algebrica in complemento a 2

- Si analizzano 3 casi:
  - X>0, Y>0:
    $X_{2,C2} + Y_{2,C2} = X + Y = (X + Y)_{2,C2}$
  - X>0, Y<0: si esprime Y secondo la definizione di complemento a 2 e si esegue la somma (X<0, Y>0 è simmetrico)
    $X_{2,C2} + Y_{2,C2} = X + (2^n - |Y|)$
    Dato che si opera in modulo 2^n
    $X - |Y| = (X - |Y|)_{2,C2}$

ES: X=3, Y=-6, calcoliamo X+Y= 3+ (-6)

- Si calcola $(-6)_{10} = 1010_{2,C2}$
- E si esegue la somma, ottenendo $(1101)_{2,C2} = 3_{10}$

0	0	1	0		
	0	0	1	1	+
	1	0	1	0	=
	1	1	0	1	

Somma algebrica in complemento a 2

- X<0, Y<0: è necessario esprimere sia X che Y in complemento a 2

$X_{2,C2} + Y_{2,C2} = 2^n - |X| + 2^n - |Y| =$

$= 2^n + [2^n - (|X|+|Y|)]$

Operando in modulo 2^n ...

$= [2^n - (|X|+|Y|)]_{2,C2}$

ES: X=-2, Y=-3 calcoliamo X+Y= -2 + (-3)

- Si calcola $(-2)_{10} = 1110_{2,C2}$ e $(-3)_{10} = 1101_{2,C2}$
- Si esegue la somma, ottenendo $1101_{2,C2} = -5_{10}$

1	1	0	0		
	1	1	1	0	+
	1	1	0	1	=
	1	0	1	1	

Somma algebrica in complemento a 2- Overflow

- Come in C1 anche in C2 si verifica overflow solo se gli operandi sono concordi ed il risultato è discorde rispetto agli operandi:

  - Es.2: si calcoli la somma –(3) + (-6) operando su 4 bit

1	0	0	0	
	1	1	0	1 +
	1	0	1	0 =
	0	1	1	1

Gli operandi sono concordi negativi, il risultato è positivo, quindi si è un overflow

Somma algebrica forma polarizzata

- Per tutti i 4 casi:

  $X_{2,p} + Y_{2,p} = X + M + Y + M = M + (X + Y + M) = M + (X + Y)_{2,p}$

- La somma algebrica di due numeri in forma polarizzata contiene 2 volte il fattore di polarizzazione, quindi il risultato va corretto togliendo M.
- Esempio

  ES: X=3, Y=-6 su 4 bit (fattore di polarizzazione 7),
  calcoliamo X+Y= 3+7 (-6 +7)
  - Si calcolano 3= 3+7 = 10 → $1010_{2,p}$ e (-6 + 7) = 1 = $0001_{2,p}$
  - E si esegue la somma, ottenendo
    $(1011)_{2,p} = 11_{10} = -3_{2,p} + 7$

forma polarizzata perche'?

- La somma algebrica di due numeri in forma polarizzata contiene 2 volte il fattore di polarizzazione, quindi il risultato va corretto togliendo M. → non una buona rappresentazione per eseguire somme.

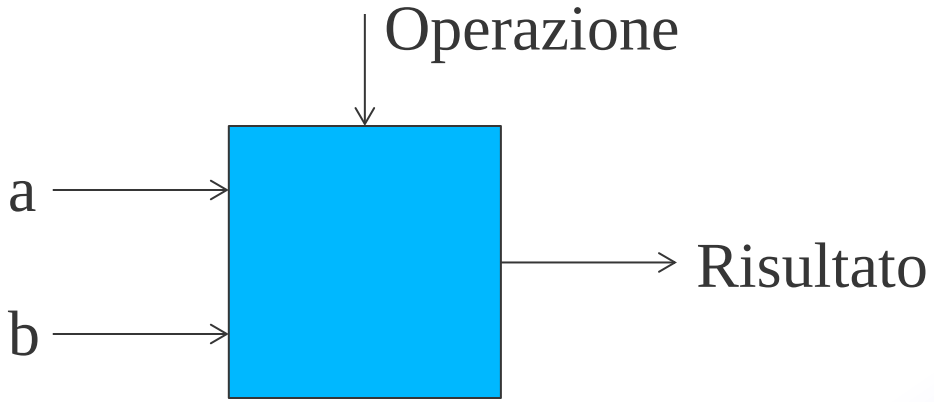
- Vediamo il confronto fra 2 numeri su 4 bit (fattore di polarizzazione 7),
- Chi e' piu' grande?
- es 4 e 6 = $1011_{2,p}$ e $1101_{2,p}$
- es -2 e -5 = $0101_{2,p}$ e $0010_{2,p}$

- Confronto bit a bit partendo da sinistra...
- Se trovo un 1 confrontato con uno 0 posso dire che il numero con l'1 e' piu' grande

Costruzione di una ALU

Costruire una ALU - Arithmetic logic unit

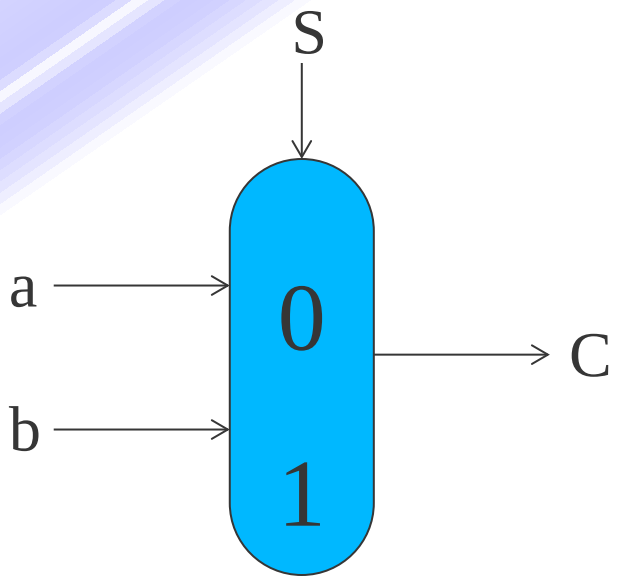
- Costruiamo una ALU per eseguire le operazioni di ANDi e ORi
  - Costruiamo una ALU ad 1 bit, e ne usiamo 32 in parallelo



OP	a	b	res
ANDi	0	0	0
	0	1	0
	1	0	0
	1	1	1
ORi	0	0	0
	0	1	1
	1	0	1
	1	1	1

Il Multiplexor (MUX)

- Seleziona uno degli input come output sulla base di un segnale di controllo

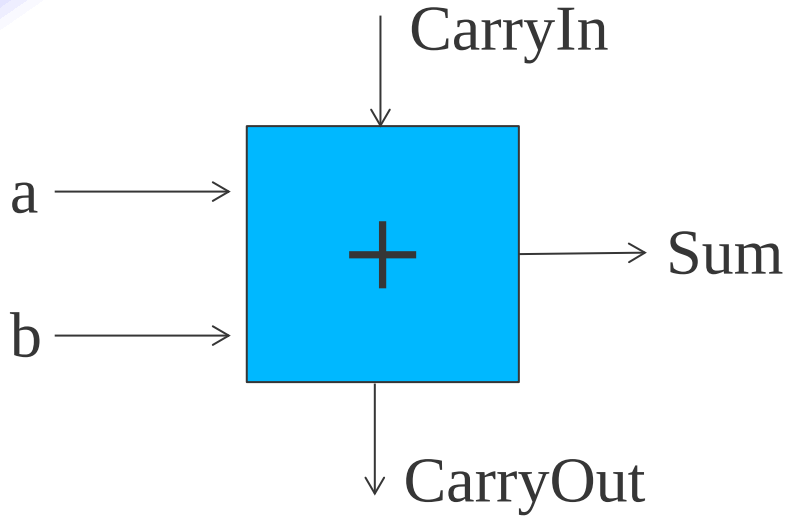


Nota: questo è un multiplexor a 2 vie anche se ha 3 input!

- **Costruiamo una ALU usando MUX**: tutte le operazioni saranno eseguite in parallelo ma selezioneremo il risultato di quella desiderata

Diverse Implementazioni

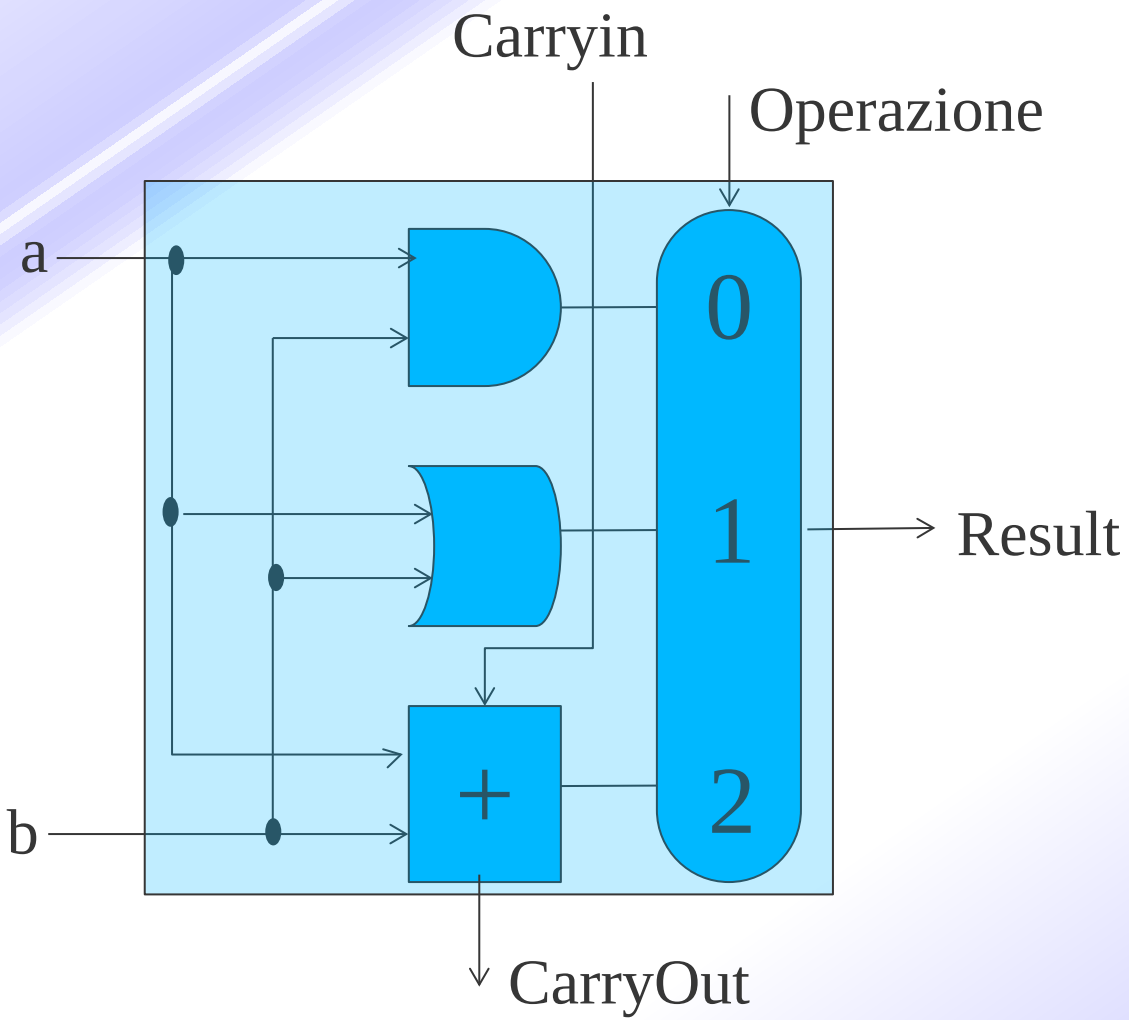
- Non è facile decidere la strada migliore per implementare qualcosa
  - Non vogliamo troppi input in un singolo circuito
  - Non vogliamo dover attraversare troppe porte
  - Per i nostri scopi è importante la semplicità!!
- Guardiamo una ALU a 1-bit per la somma

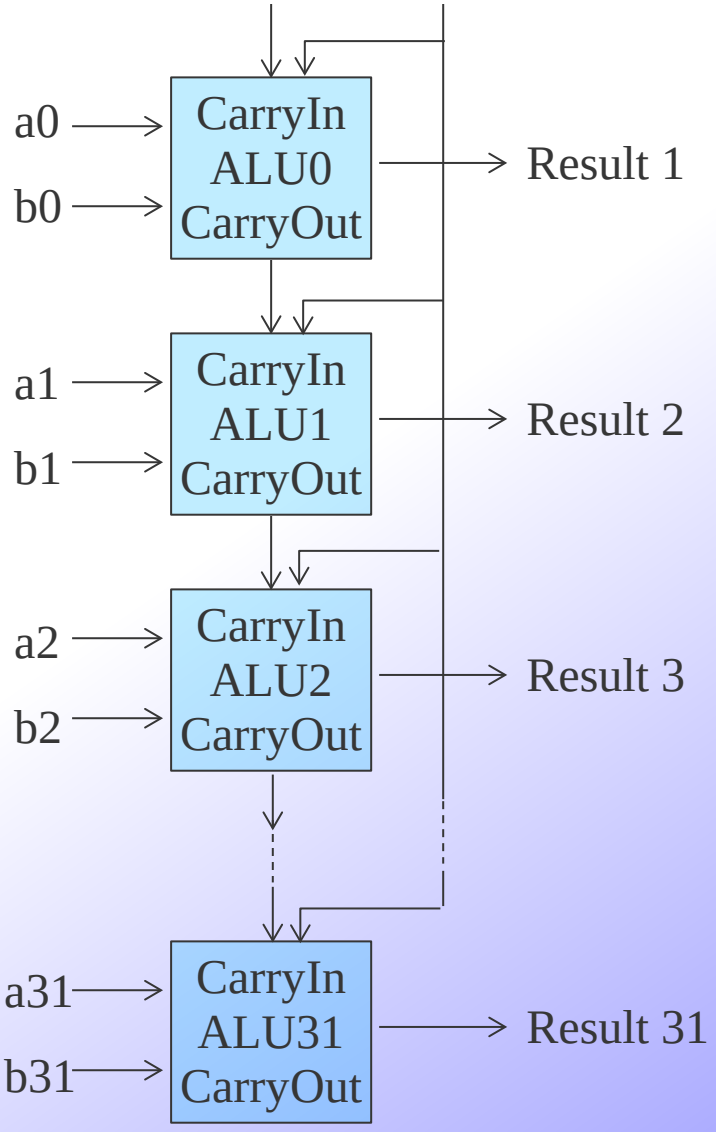


$$c_{out} = ab + ac_{in} + bc_{in}$$
$$sum = a \; xor \; b \; xor \; c_{in}$$

- Come costruire un ALU a 1-bit per ADD, AND e OR ??
- Come costruire un ALU a 32-bit?

Costruire una ALU a 32-bit

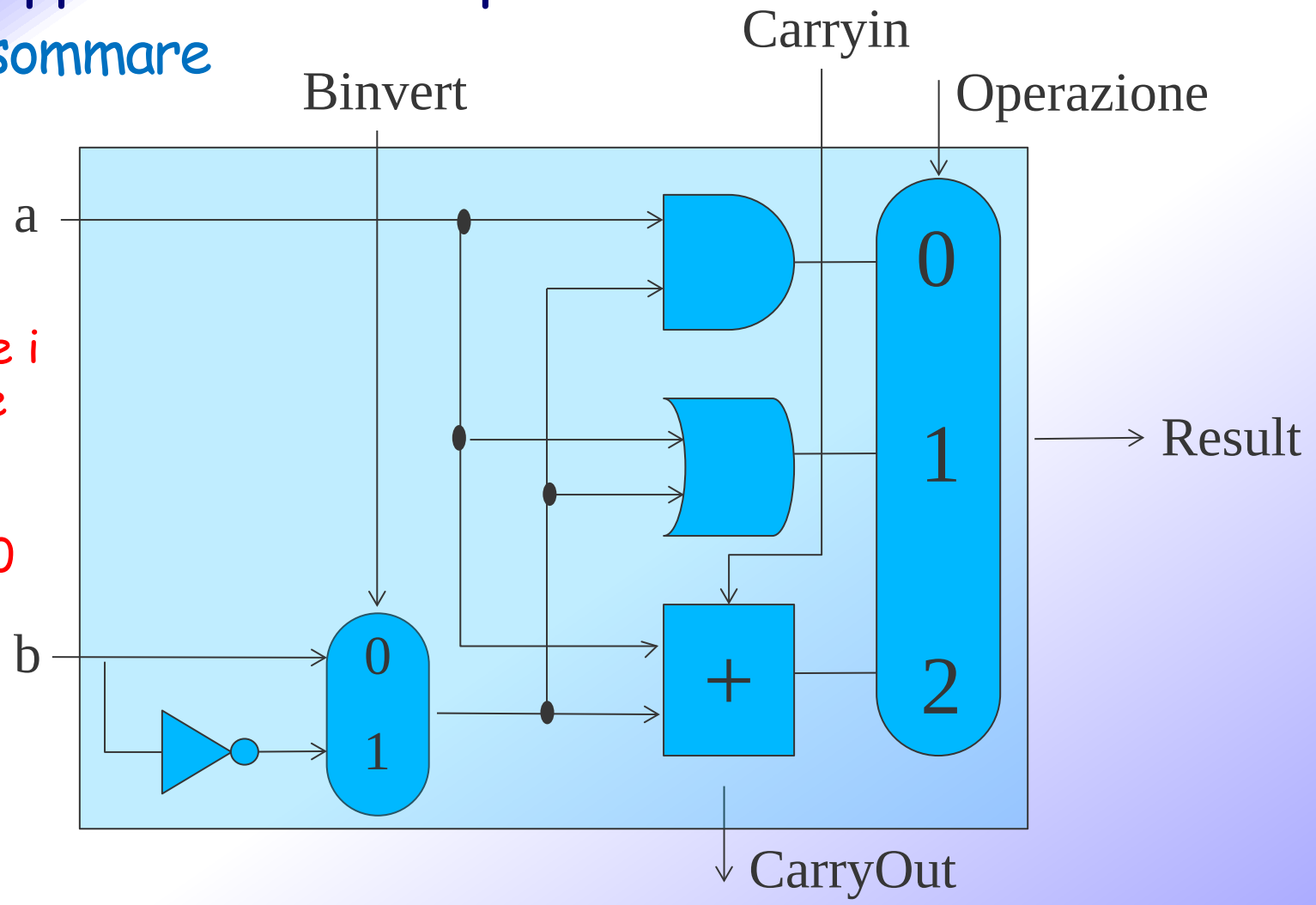




E per la Sottrazione (a-b) ?

- Utilizziamo l'approccio del complemento a 2:
  - Negare b e sommare

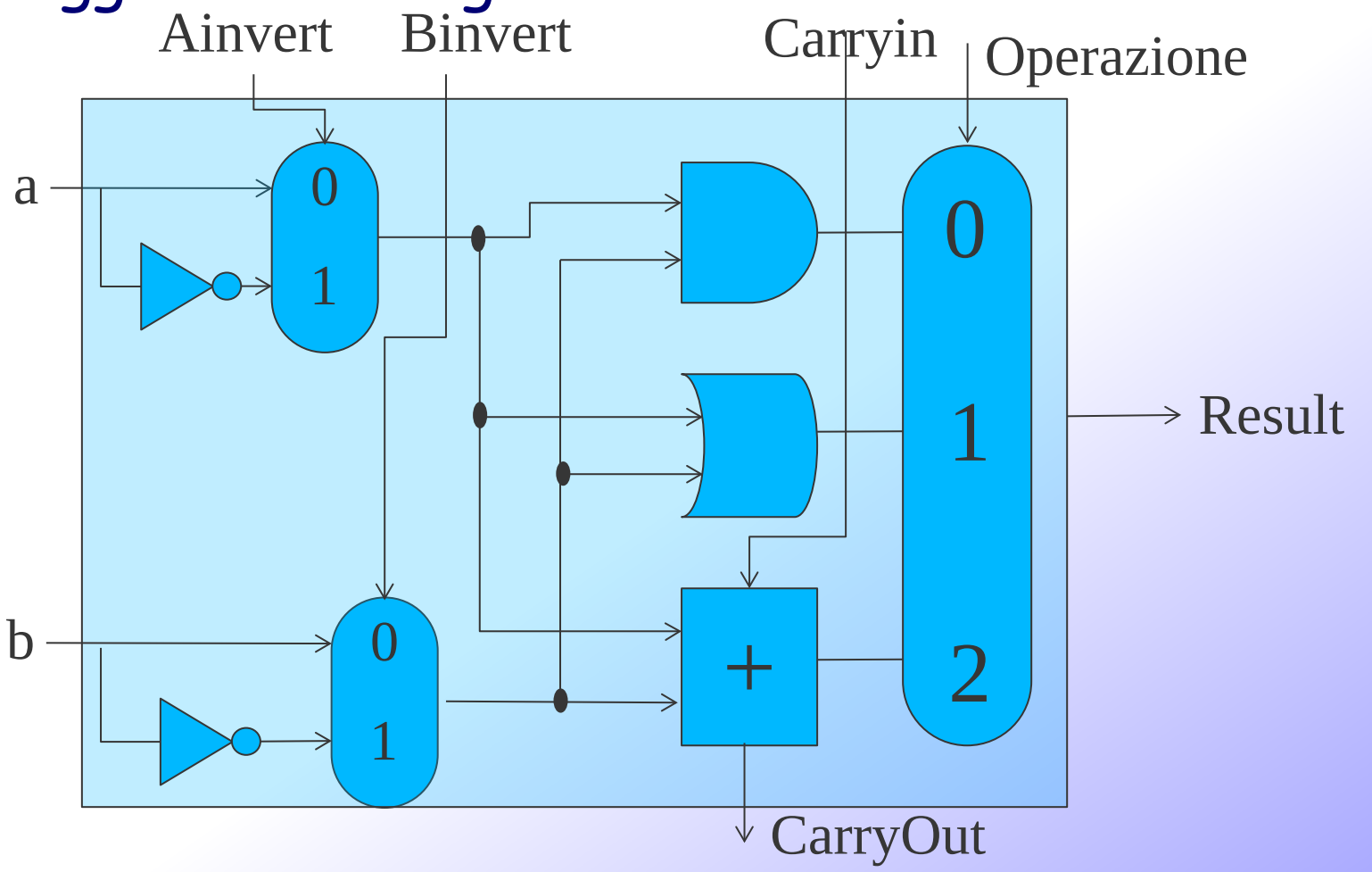
Per negare b dobbiamo invertire i bit di b e sommare 1. lofacciamo mettendo a 1 il Carryin della ALU0



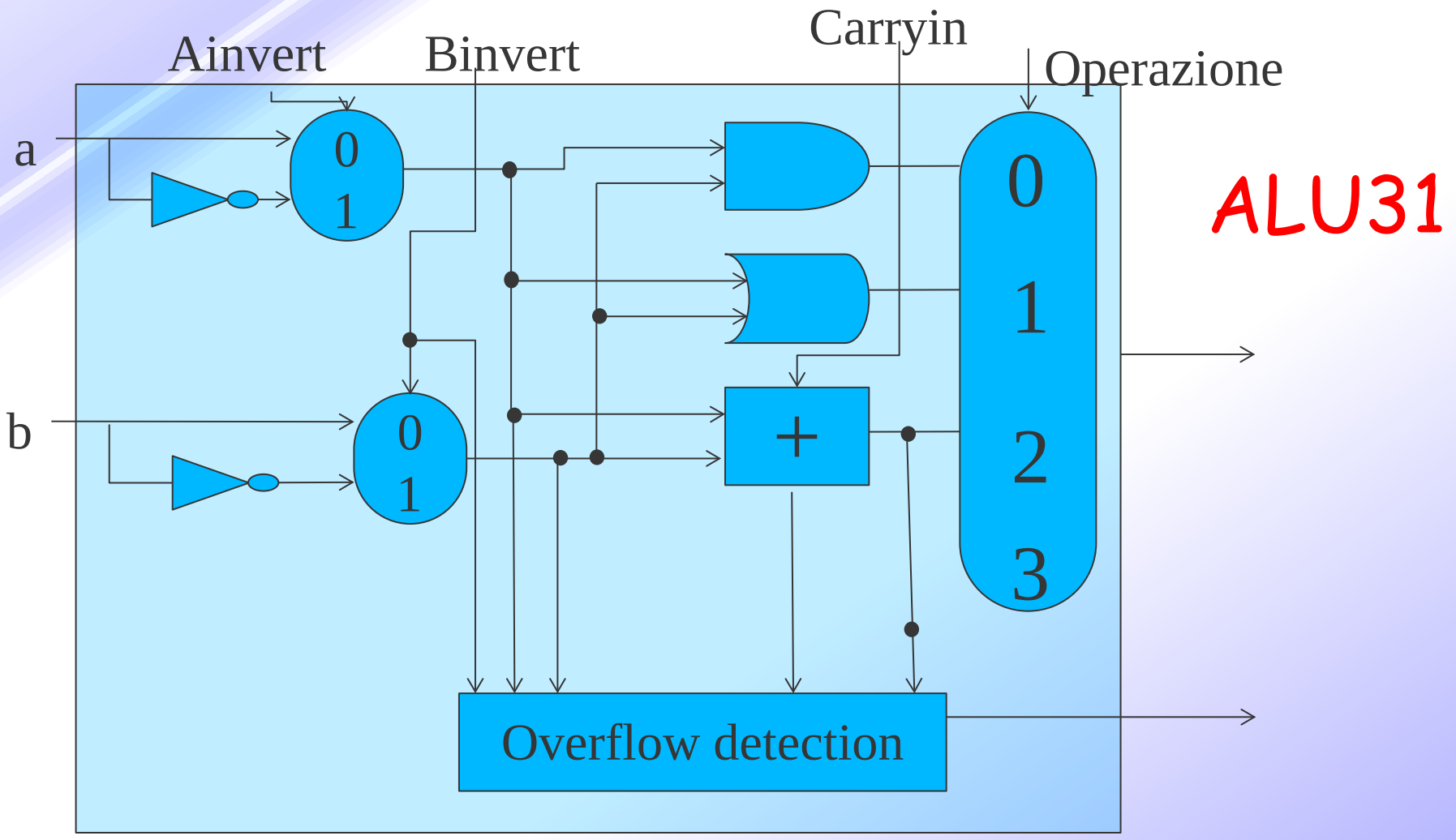
Operatore NOR

- Invece di aggiungere una porta in più per l'operazione di NOR, utilizziamo la legge di De Morgan:

$$\overline{a + b} = \overline{a} \cdot \overline{b}$$



Overflow


Ainvert
Binvert
Carryin
Operazione
a
b
0
1
0
1
+
0
1
2
3
Overflow detection
ALU31

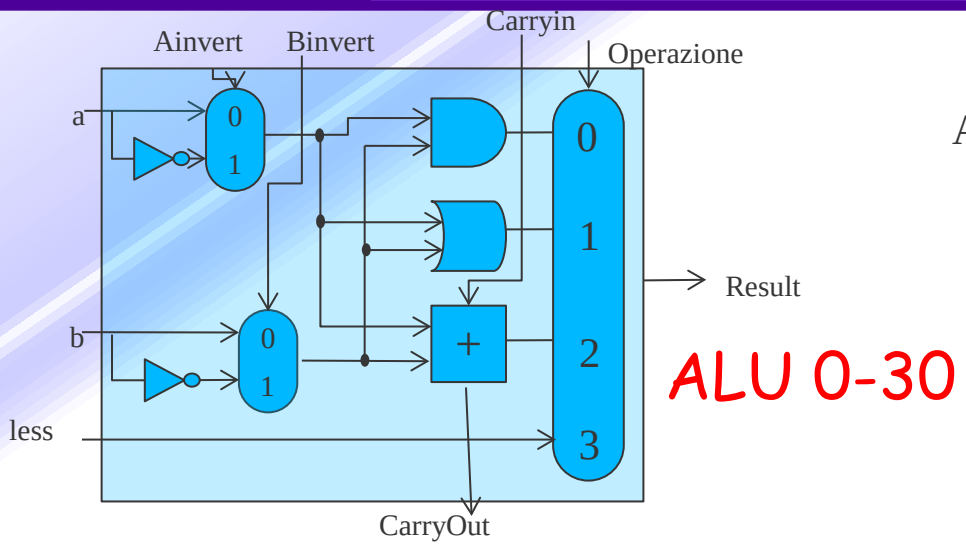
Operatore di confronto slt

- L'operatore Slt (Set On Less) restituisce
  1 se a < b, 0 altrimenti
  - Slt imposta tutti i bit a 0 tranne quello più a destra (bit 0)
  - Ci serve un altro ingresso nel muliplexor a cui collegare 0 per gli ultimi 31 bit della ALU (1-31) ed il risultato della slt nel bit 0.
  - Come calcolare il primo bit?

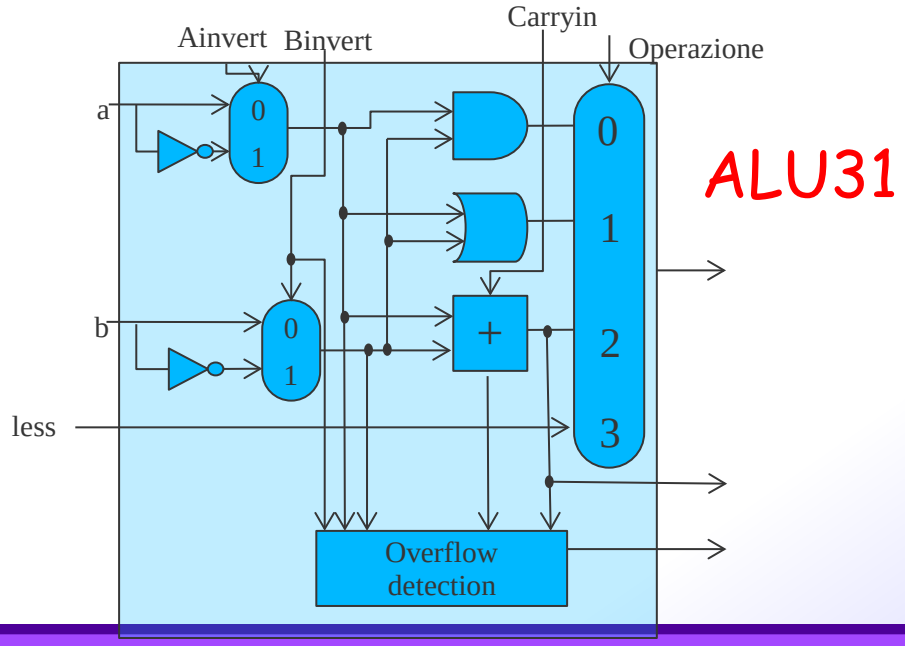
$$(a - b) < 0 \Rightarrow \big((a - b) + b\big) < (0 + b) \Rightarrow a < b$$

  - Calcoliamo a-b e settiamo il risultato del bit 0 a 1 se è a-b<0, 0 altrimenti.

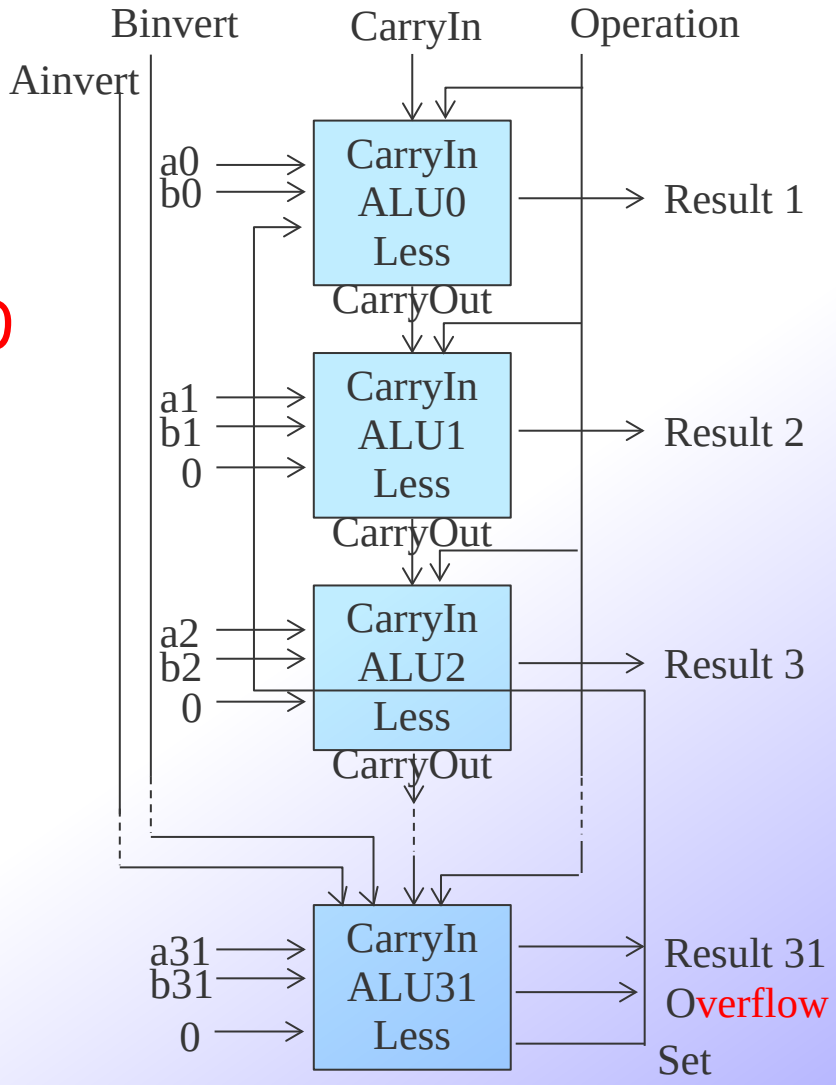
Architettura a supporto di SLT



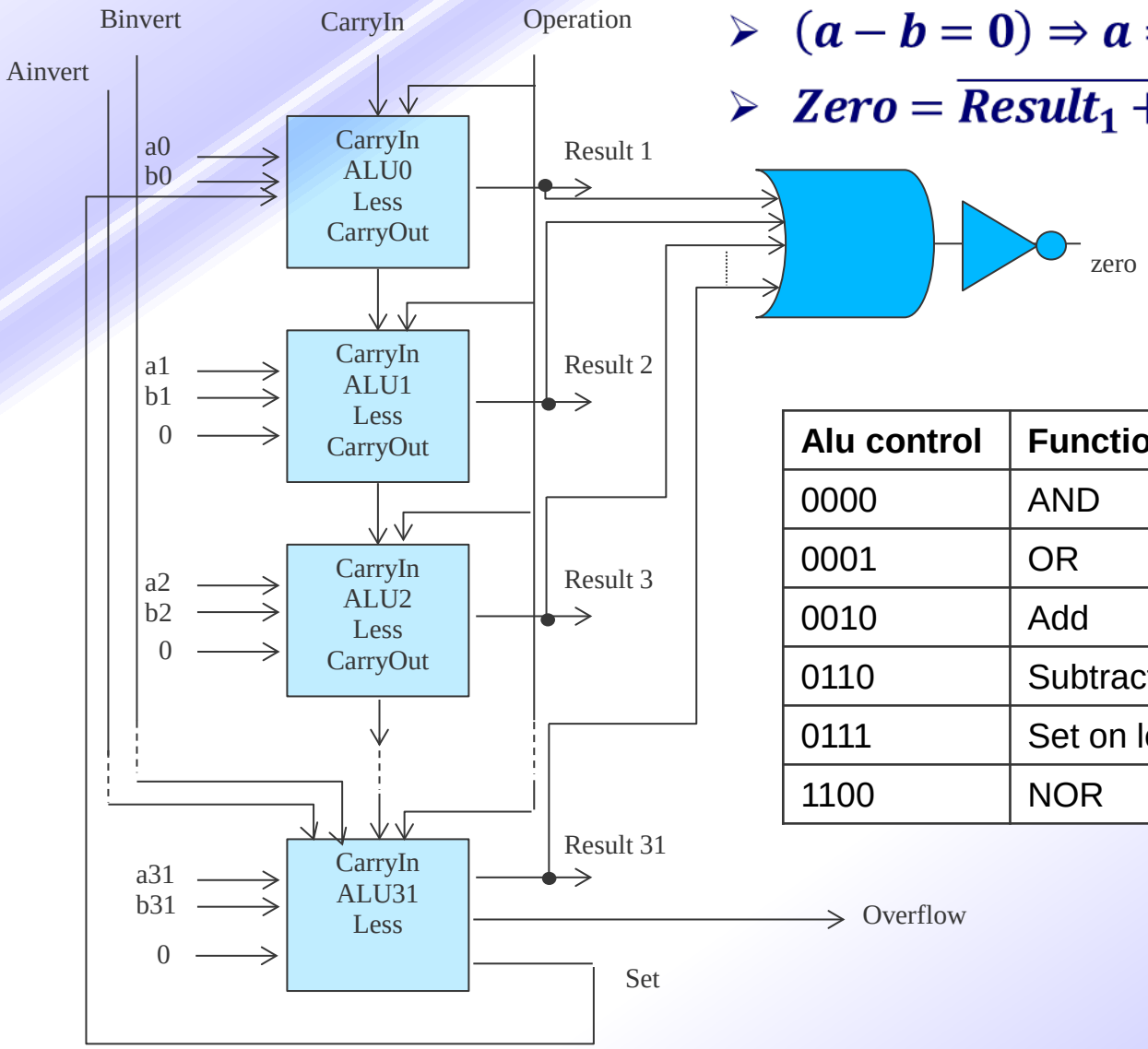
ALU 0-30



ALU31

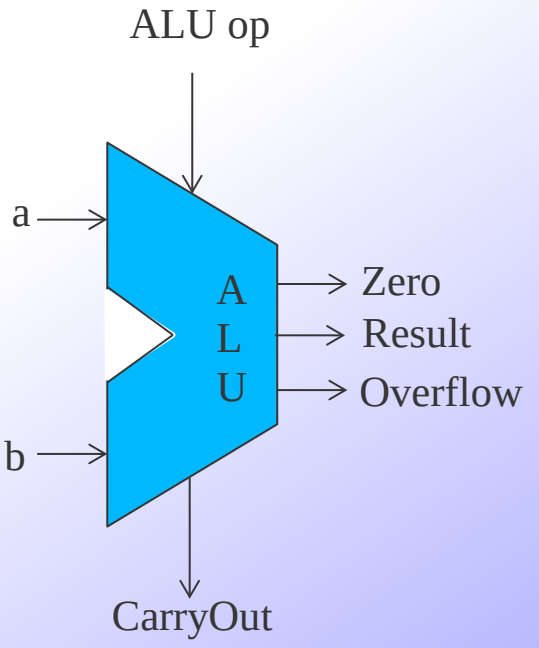


Architettura di una ALU a 32 bit



- $(a - b = 0) \Rightarrow a = b$
- $Zero = \overline{Result_1 + Result_2 + \cdots + Result_{32}}$

Alu control	Function
0000	AND
0001	OR
0010	Add
0110	Subtract
0111	Set on less than
1100	NOR



Osservazioni

- Sappiamo costruire una ALU a supporto delle istruzioni MIPS
  - Idea chiave: usare multiplexor per selezionare l'output desiderato
  - Possiamo usare il complemento a 2 per effettuare efficacemente la sottrazione
  - Possiamo replicare una ALU ad 1 bit per ottenere una ALU a 32 bit
- Osservazioni sull'Hardware
  - Tutte le porte logiche sono sempre in funzione
  - La velocità di una porta logica è funzione del numero di input
  - La velocità di un circuito è influenzata dal numero di porte logiche in serie
- Il nostro scopo primario: comprensione, ma…
  - Cambiamenti opportuni all'architettura possono migliorare le performance
    - (è come usare algoritmi migliori nel software)
  - Vediamo un esempio per l'addizione ed in seguito un altro per la moltiplicazione

Il problema del riporto

- La chiave per velocizzare l'addizione sta nel determinare prima il bit di riporto:

$$c_1 = a_0c_0 + b_0c_0 + a_0b_0$$
$$c_2 = a_1\mathbf{c_1} + b_1\mathbf{c_1} + a_1b_1$$
$$\dots$$

  - Sostituendo e ricavando c2

$$c_2 = a_1a_0b_0 + a_1a_0c_0 + a_1b_0c_0 + b_1a_0b_0 + b_1a_0c_0 + b_1b_0c_0 + a_1b_1$$

  - È facile immaginare come questa espressione esploda con l'aumentare del numero di bit
  - Realizzare in hardware questa soluzione non è praticabile.

Carry lookahead – primo livello di astrazione

- Si usa un livello di astrazione per limitare la complessità

$$c_{i+1} = b_i c_i + a_i c_i + a_i b_i = a_i b_i + (a_i + b_i) c_i$$

  - Se scriviamo l'equazione per c2

$$c_2 = a_1 b_1 + (a_1 + b_1)(a_0 b_0 + (a_0 + b_0) c_0)$$

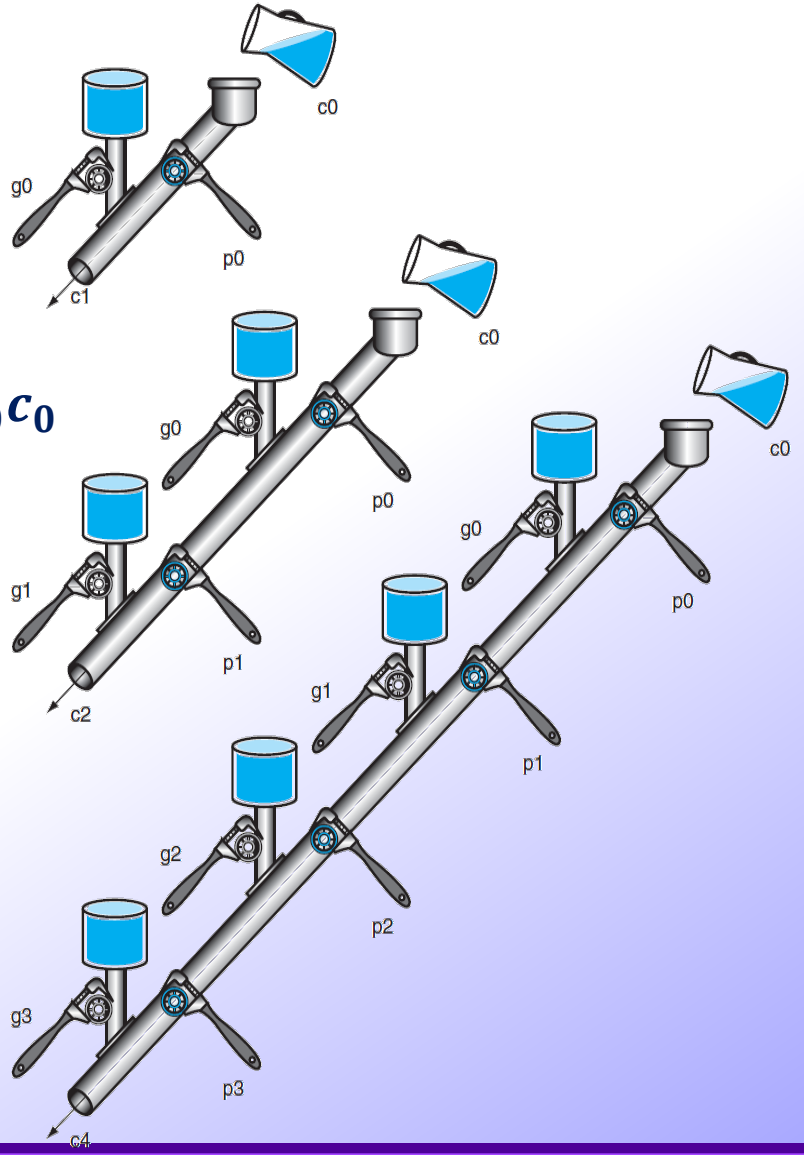
  - Notiamo le occorrenze dei termini
    - Generate : $g_i = a_i b_i$
    - Propagate: $p_i = a_i + b_i$
  - Se li usiamo per definire c_{i+1}

- $C_{i+1} = g_i + p_i c_i$
  - Con generate=1 l'addizionatore genera un riporto indipendentemente dal riporto precedente
  - Con propagate=1 l'addizionatore propaga il riporto precedente

Addizionatore Carry-lookahead - 2

- Basandoci sulle definizioni di generate and propagate

$$c_1 = g_o + p_0 c_0$$
$$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$$
$$c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$$
$$c_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$$



- $c_i = 1$ se
  - $g_j = 1, \quad j < i$
    qualche addizionatore precedente genera un riporto e
  - $p_k = 1, \quad j < k < i$
    Tutti gli addizionatori intermedi lo propagano

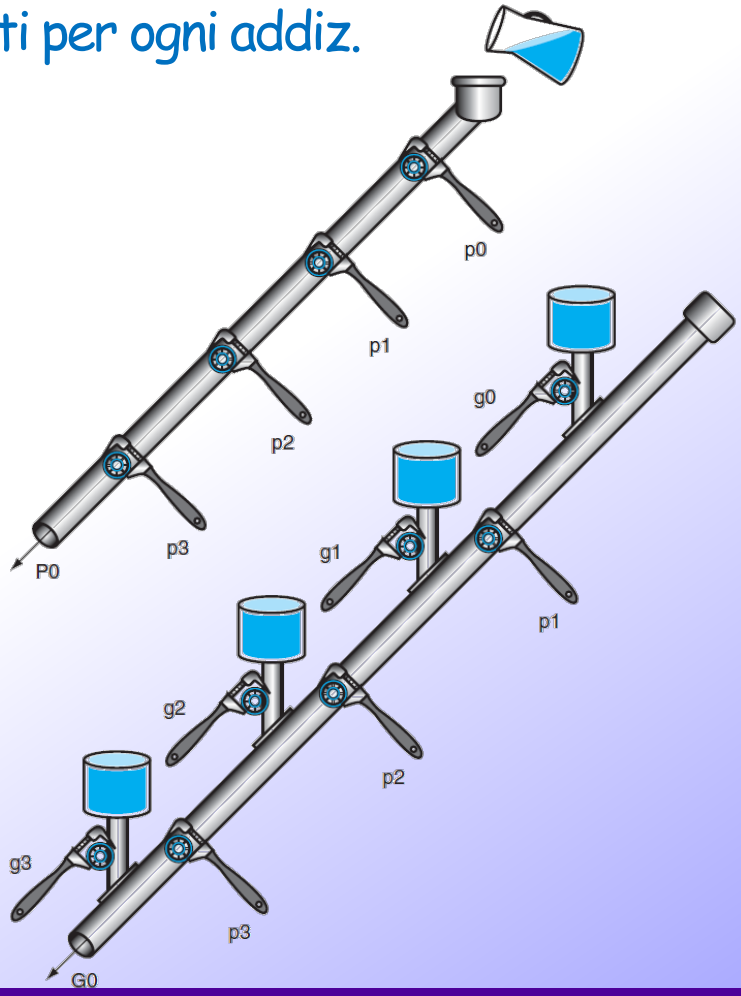
Carry lookahead – secondo livello di astrazione

- Consideriamo un addizionatore a 4 bit con unità carry-lookahead usandone 4 per costruire un addizionatore a 16 bit
  - Abbiamo bisogno di «propagare» e «generare» riporti per ogni addiz.

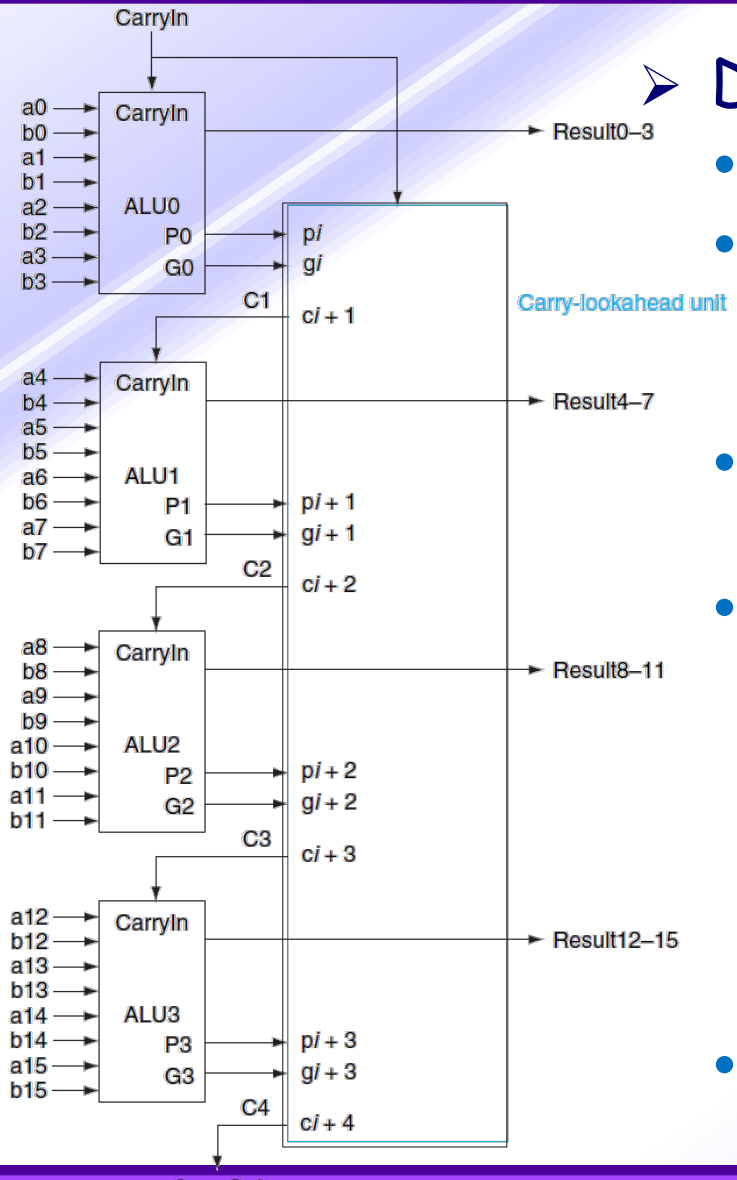
$$P_0 = p_3p_2p_1p_0$$
$$P_1 = p_7p_6p_5p_4$$
$$P_2 = p_{11}p_{10}p_9p_8$$
$$P_3 = p_{15}p_{14}p_{13}p_{12}$$

$$G_0 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0$$
$$G_1 = g_7 + p_7g_6 + p_7p_6g_5 + \mathbf{p_7p_6p_5g_4}$$
$$G_2 = g_{11} + p_{11}g_{10} + p_{11}p_{10}g_9 + p_{11}p_{10}p_9g_8$$
$$G_3 = g_{15} + p_{15}g_{14} + p_{15}p_{14}g_{13} + p_{15}p_{14}p_{13}g_{12}$$

$$C_1 = G_o + P_0c_0$$
$$C_2 = G_1 + P_1G_0 + P_1P_0c_0$$
$$C_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0c_0$$
$$C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0$$



Prestazioni del carry look ahead



- Di quanto il carry-lookahead è più veloce?
  - Assumiamo lo stesso tempo per AND e OR
  - Contiamo il numero di porte logiche attraversate nel caso di riporto semplice e nel caso di carry-lookahead
  - Riporto semplice:
    - 2 porte per ogni addizionatore: 16x2=32
  - Carry-lookahead:
    - C4 è calcolato in termini di Pi e Gi (l'OR di diversi termini in AND).
    - Pi è calcolato in un AND usando pi
    - Gi è calcolato usando pi e gi, caso peggiore è di due porte.
    - il caso peggiore è 2+1+2 = 5.
  - Il carry-lookahead è 6 volte più veloce