Lezione 5

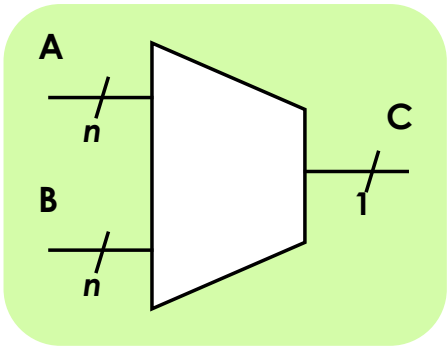
Circuiti combinatori notevoli

F. Pedersini

Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano

Comparatore

- Confronta parole di **n** bit
  - IN: **2** gruppi di **n** bit
  - OUT: **1** bit
  - OUT = 1 se i due IN sono uguali
  - OUT = 0 se diversi.



A_0	B_0	C_0
0	0	1
0	1	0
1	0	0
1	1	1

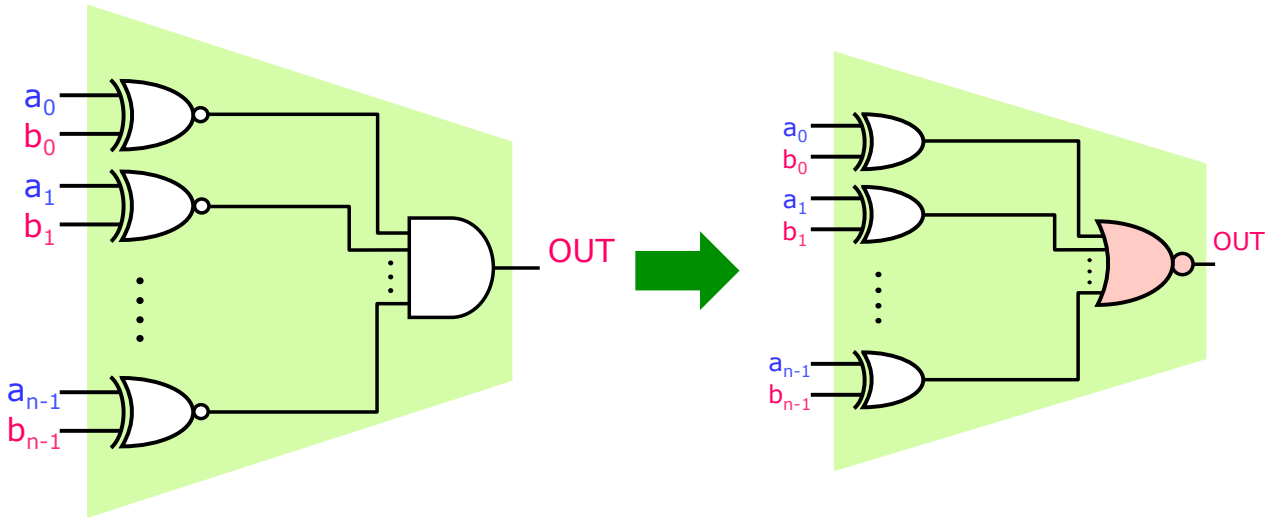
A_1	B_1	C_1
0	0	1
0	1	0
1	0	0
1	1	1

$$C_i = \overline{A_i \oplus B_i}$$

$$C = C_0 \cdot C_1 \cdot \ldots \cdot C_{n-1}$$

Comparatore

- Comparatore a n ingressi: schema circuitale

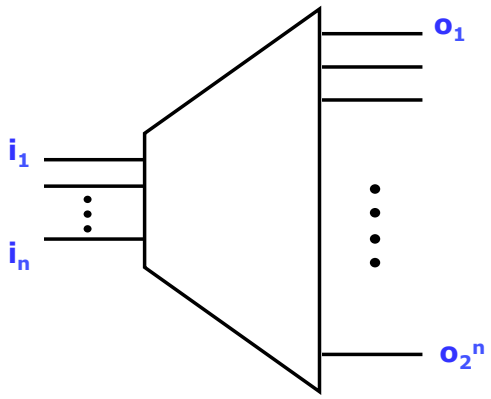


n porte **XNOR** + 1 **AND** ad **n** ingressi

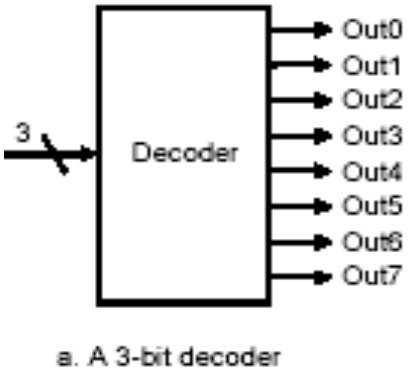
n porte **XOR** + 1 **NOR** ad **n** ingressi

Decodificatore (decoder)

- **n** ingressi, 2^n uscite
- Il numero espresso sugli ingressi è usato per asserire (porre a 1) la linea di uscita di tale indice
  - con 4 linee di input e 16 di output (da 0 a 15), se in ingresso arriva il valore 0110, in uscita si alza la linea di indice 5 (la sesta!)
- usato per inidirizzare la memoria



Decoder



a. A 3-bit decoder

A	B	C	U_0	U_1	U_2	U_3	U_4	U_5	U_6	U_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

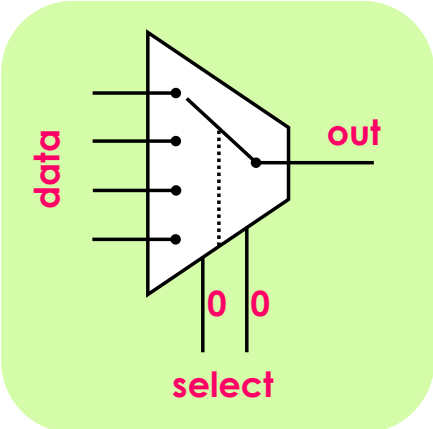
Le funzioni di uscita sono 2^n per n input:

$U_0 = \sim A \sim B \sim C$

...

$U_7 = A\,BC$

- Operatore di **selezione**
  - **IN:** **n** linee di input (**data**)
    k linee di controllo (**select**)
  - **OUT:** **1** linea
    - Il valore fornito sulla linea di controllo viene connessa all'uscita la linea di ingresso selezionata.



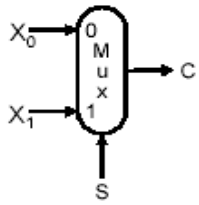
- Quante linee di controllo?

$\mathbf{k} = \text{ceil}\,(\log_2 n)$

  - Linee di input: $n = 4$
  - Linee di controllo: $k = ceil(\log_2 4) = 2$

Multiplexer binario

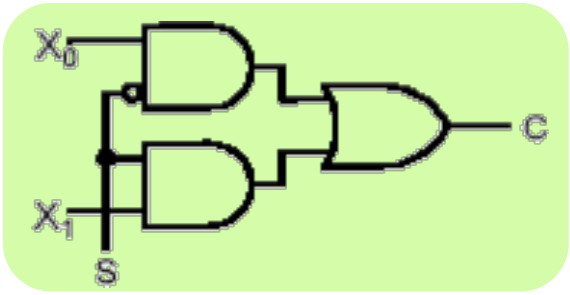
- **n = 2, k = 1**
  - Selezione S "apre" la porta opportuna
  - Circuito logico a **3 ingressi, 1 uscita**



S	X_0	X_1	C
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

SoP:

$$C = \overline{S}X_0\overline{X_1} + \overline{S}X_0X_1 + S\overline{X_0}X_1 + SX_0X_1 =$$
$$= \overline{S}X_0 + SX_1$$



Sintesi Multiplexer in forma canonica POS

$$C = (S + X_0 + X_1)(S + X_0 + \overline{X_1})(\overline{S} + X_0 + X_1)(\overline{S} + \overline{X_0} + X_1) =$$

$$\begin{cases} a = \overline{S} + X_1 \\ b = S + X_0 \end{cases}$$

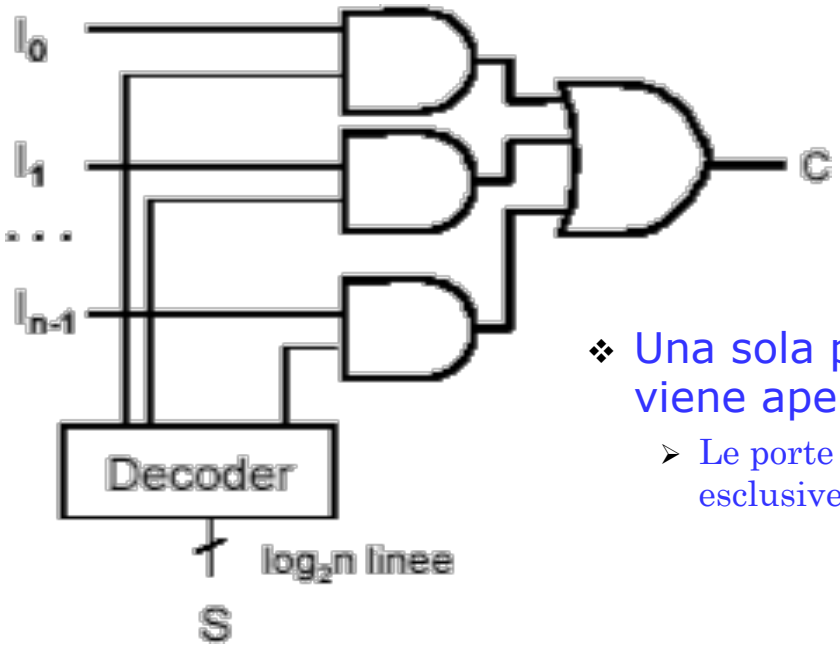
$$= \left[(b + X_1)(b + \overline{X_1})\right] \cdot \left[(a + X_0)(a + \overline{X_0})\right] =$$

$$= \left[b + b(X_1 + \overline{X_1}) + X_1\overline{X_1}\right] \cdot [a] = ba =$$

$$= (S + X_1)(\overline{S} + X_0)$$

S	X_0	X_1	C
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Multiplexer (Mux) a più vie.



- Una sola porta alla volta viene aperta dal segnale S.
  - Le porte sono mutuamente esclusive.

HALF Adder (1 bit)

- Somma aritmetica tra 2 bit
  - 2 ingressi: addendi: **a** , **b**
  - 2 uscite: somma: **s**
    riporto: **r**

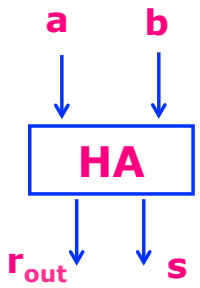
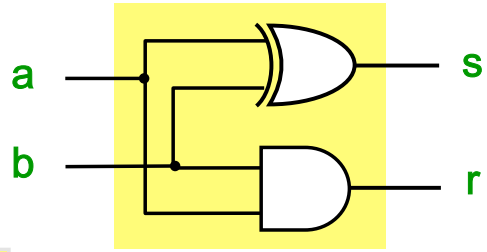


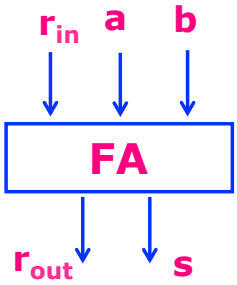
Tabella della verità			
a	b	somma	riporto
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



$s = a \oplus b$
$r = a \cdot b$

FULL Adder (1 bit)

- Gestisce anche il riporto in ingresso
  - 3 ingressi: **a, b, r_{IN}**
  - 2 uscite: **s, r_{OUT}**



a	b	r_{in}	r_{out}	s
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

SOP:

$$s = m_1 + m_2 + m_4 + m_7$$
$$r_{out} = m_3 + m_5 + m_6 + m_7$$

$$s = \bar{a}b\bar{r}_{in} + a\bar{b}\bar{r}_{in} + \bar{a}\bar{b}r_{in} + abr_{in}$$
$$r_{out} = ab\bar{r}_{in} + \bar{a}br_{in} + a\bar{b}r_{in} + abr_{in}$$

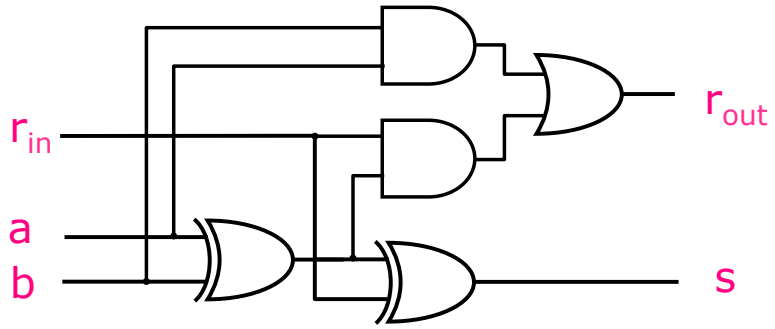
Semplificando le espressioni

$$s = \bar{a}b\bar{r}_{in} + a\bar{b}\bar{r}_{in} + \bar{a}\bar{b}r_{in} + abr_{in} =$$
$$= (a \oplus b)\bar{r}_{in} + (ab + \bar{a}\bar{b})r_{in} =$$
$$= (a \oplus b)\bar{r}_{in} + \overline{(a \oplus b)}r_{in} =$$
$$= (a \oplus b) \oplus r_{in}$$

$$r_{out} = ab\bar{r}_{in} + \bar{a}br_{in} + a\bar{b}r_{in} + abr_{in} =$$
$$= ab(r_{in} + \bar{r}_{in}) + (a\bar{b} + \bar{a}b)r_{in} =$$
$$= ab + (a \oplus b)r_{in}$$

$$s = (a \oplus b)\overline{r_{in}} + \overline{(a \oplus b)}r_{in} = (a \oplus b) \oplus r_{in}$$

$$r_{out} = ab + (a \oplus b)r_{in}$$



Moltiplicando

1 1 0 1 1 x (27_{10})
1 1 1 = (7_{10})

1 1 1 1 1 1
1 1 0 1 1
1 1 0 1 1 –
1 1 0 1 1 – –

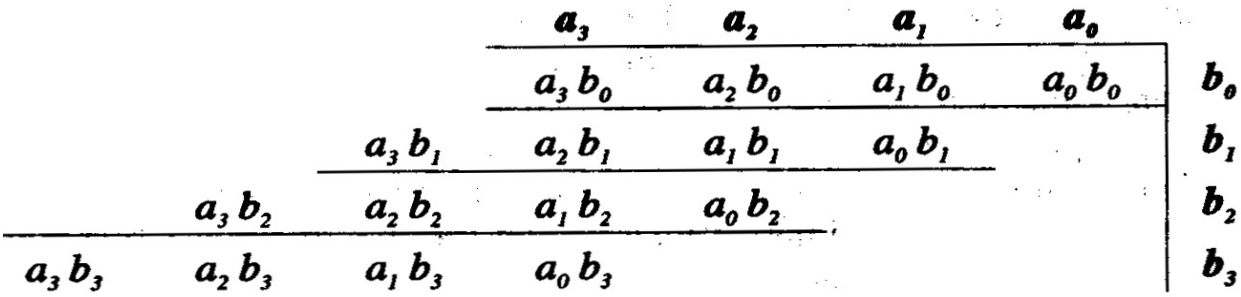
1 0 1 1 1 1 0 1 (189_{10})

Moltiplicatore

Prodotto

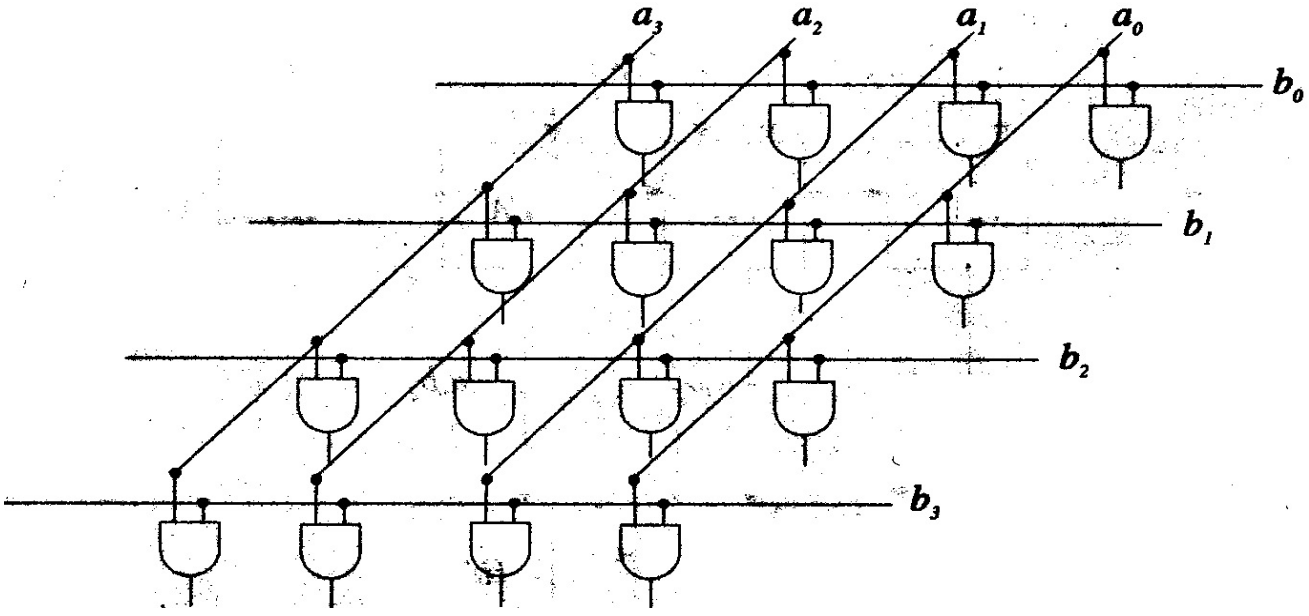
- Come fare il calcolo con circuiti logici ?
  - Possiamo scomporre l'operazione in due stadi:
  - Primo stadio: prodotti parziali
    - si mette in AND ciascun bit del moltiplicatore con i bit corrispondenti del moltiplicando
  - Secondo stadio: somme
    - si effettuano le somme (full adder) dei bit sulle righe contenenti i prodotti parziali

La matrice dei prodotti parziali

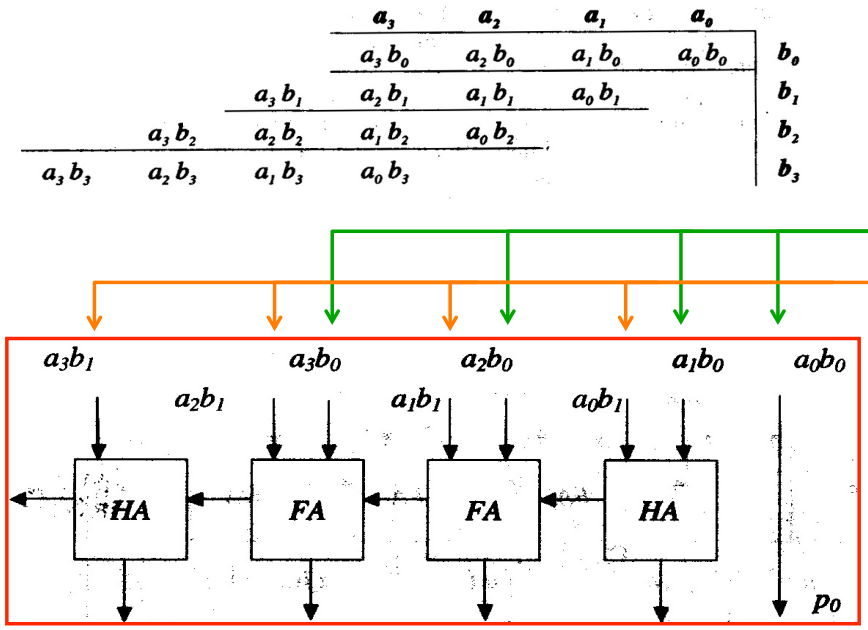


In binario i prodotti parziali sono degli AND

Il circuito che effettua i prodotti

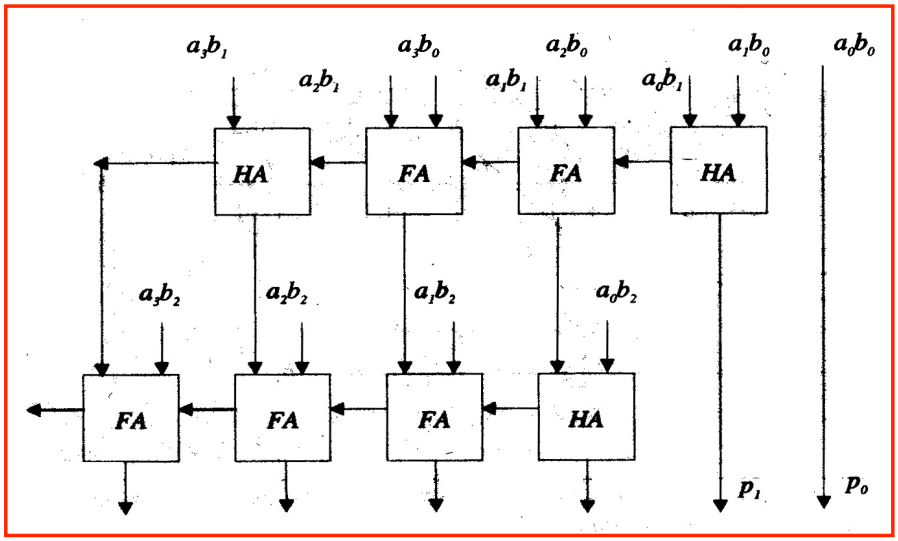


Somma – prime 2 righe dei prodotti parziali



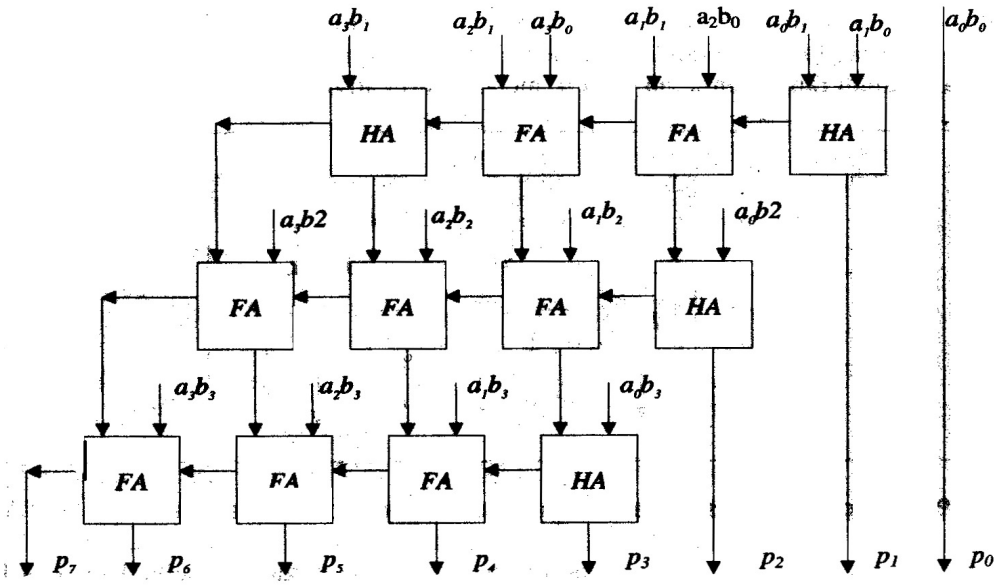

```
      1 0 1 1  x
      0 1 1 1  =
  ---------------
        1 1 1
      1 0 1 1  +
    1 0 1 1 -
  ---------------
    1
  1 0 0 0 0 1  +
  1 0 1 1 - -
  ---------------
1 1 0 1 1 0 1
```


Somma della terza riga




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

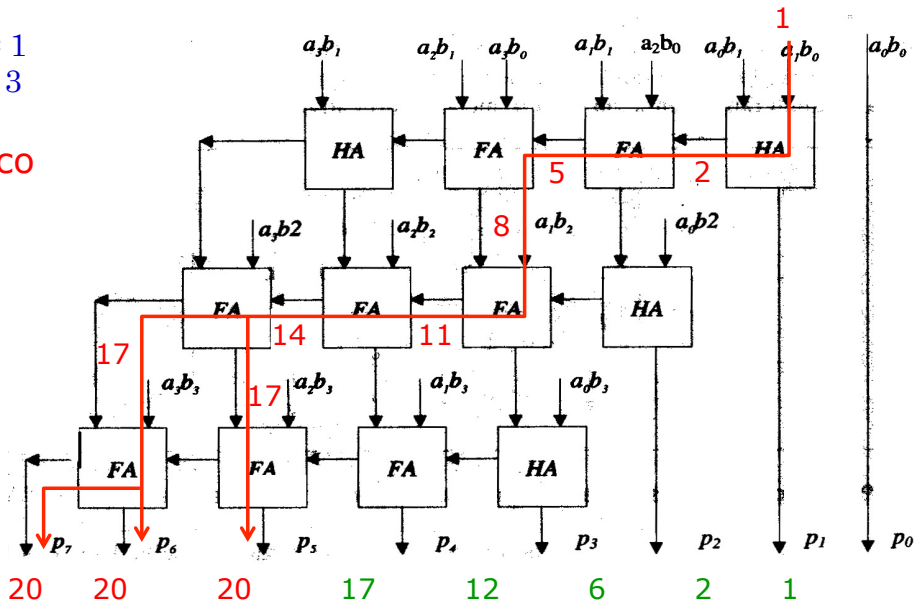
Somma prodotti parziali – circuito completo



Overflow: A e B su: **N bit** → P su: **2N bit**

Valutazione del cammino critico

- **N = 4**
  - Ritardo AND = 1
  - Ritardo HA = 1
  - Ritardo FA = 3
- Cammino critico = 20



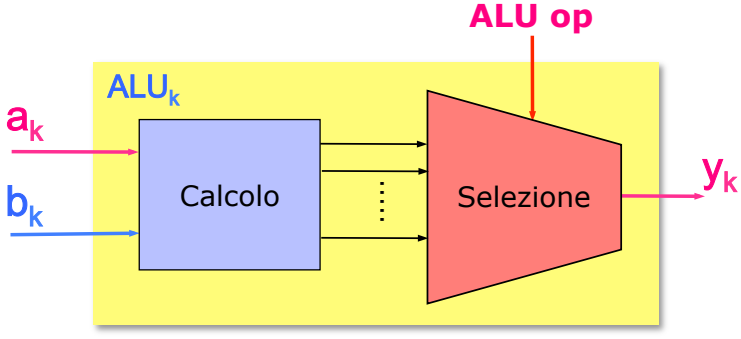
ALU: Arithmetic-Logic Unit

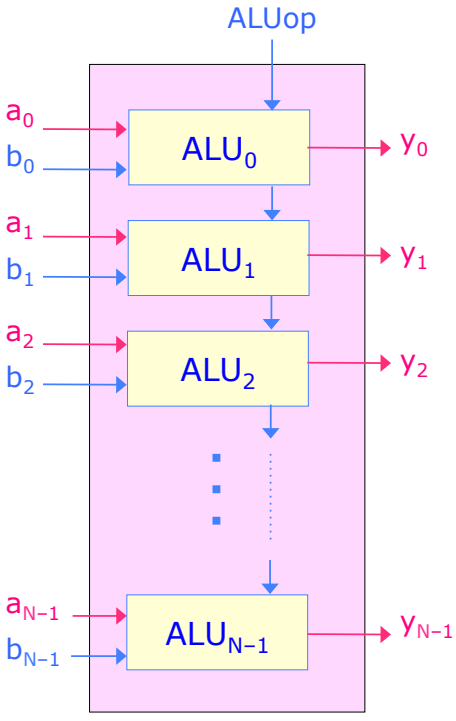
- Esegue le operazioni aritmetico-logiche
  + , – , x , ÷ , ...
  and , or , not , xor , = , ≠ , ...
- Normalmente integrata nel processore
  - Inizio anni '90 → introduzione con il nome di co-processore matematico
  - Le ALU non compaiono solamente nei micro-processori
- E' un circuito combinatorio
  - Rappresentabile come insieme di funzioni logiche
- Opera su **parole (N bit)**
  - 6502, 8080, Z-80 8 bit
  - MIPS, 80386: 32 bit
  - PowerPC G5, Athlon64: 64 bit
- Struttura modulare
  - Blocchi funzionali da 1 bit, replicati N volte

ALU a N bit → N ALU a 1 bit (ALU elementari)

Struttura ALU elementare:

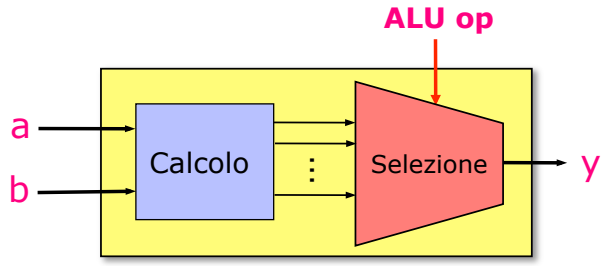
- Ingressi: Operandi: a_k, b_k
  Riporto in ingresso: r_{in}
  Selettore operazione: ALUop
- Uscite: Risultato: y_k
  Riporto in uscita: r_{out}

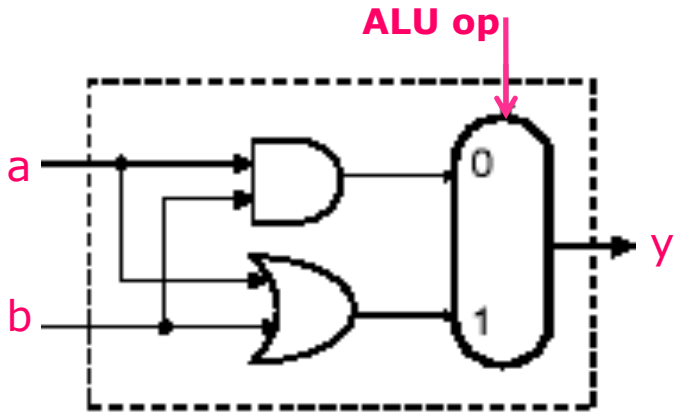




Progettazione della ALU

- Porta AND / OR
  - Selezionabile
- Componenti:
  - 1 porta AND
  - 1 porta OR
  - 1 Multiplexer (MUX)

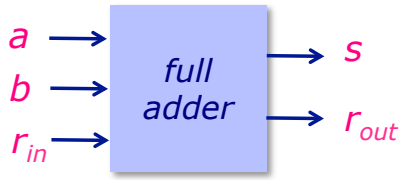




ALUop = 0 → y = AND(a,b)
ALUop = 1 → y = OR(a,b)

FULL Adder (1 bit)

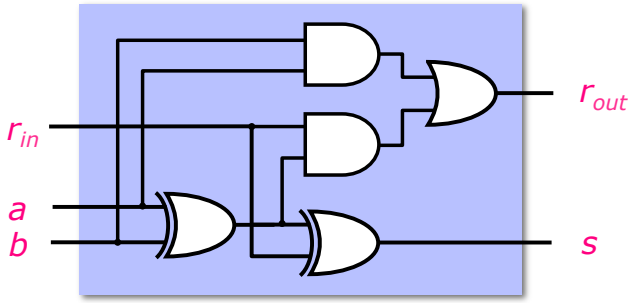
- Gestisce anche il riporto in ingresso
  - 3 ingressi: **a, b, r_{IN}**
  - 2 uscite: **s, r_{OUT}**



a	b	r_{in}	r_{out}	s
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

$$s = \bar{a}\bar{b}r_{in} + a\bar{b}\bar{r}_{in} + \bar{a}b\bar{r}_{in} + abr_{in} = a \oplus b \oplus r_{in}$$

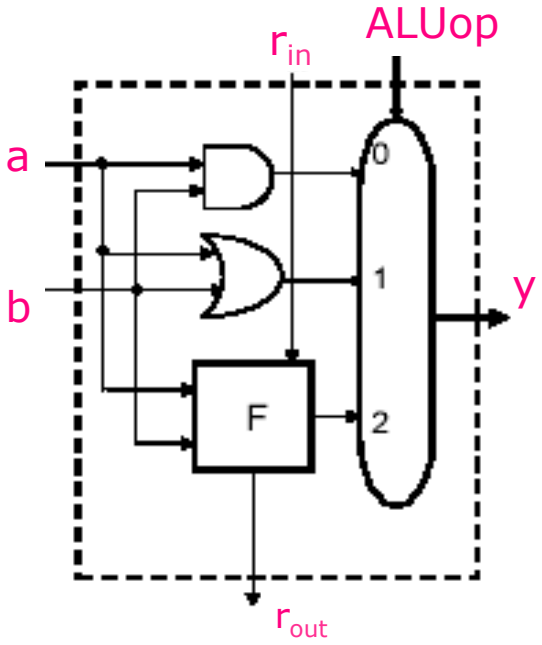
$$r_{out} = ab\bar{r}_{in} + \bar{a}br_{in} + a\bar{b}r_{in} + abr_{in} = ab + (a \oplus b)r_{in}$$



ALU 1 bit

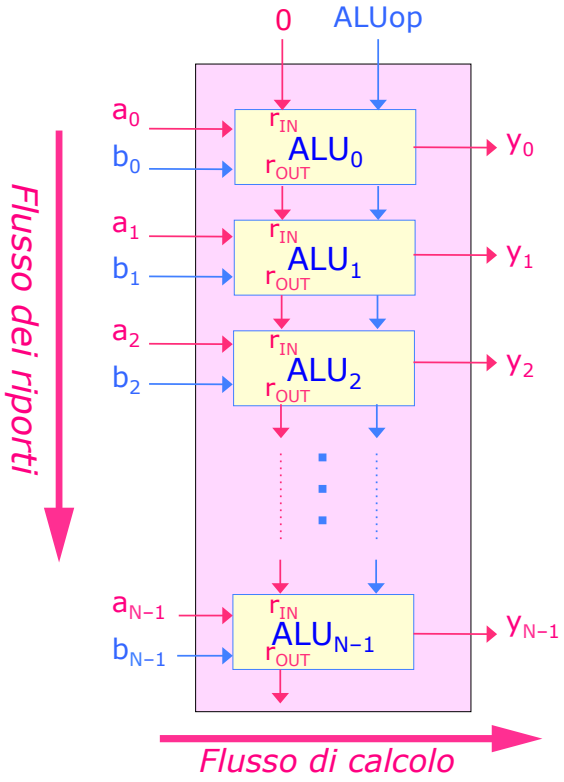
Operazioni:

- OR, AND, somma
- ALUop: 2 bit
  - 00: s = a and b
  - 01: s = a or b
  - 10: s = a + b + r_{in}



ALU a 32 bit

- Come collegare N ALU a 1 bit per ottenere ALU a N bit?
- ALU a N bit:
- ALU in parallelo, ma...
  - Propagazione dei riporti
  - Limite alla velocità di calcolo



Sottrazione

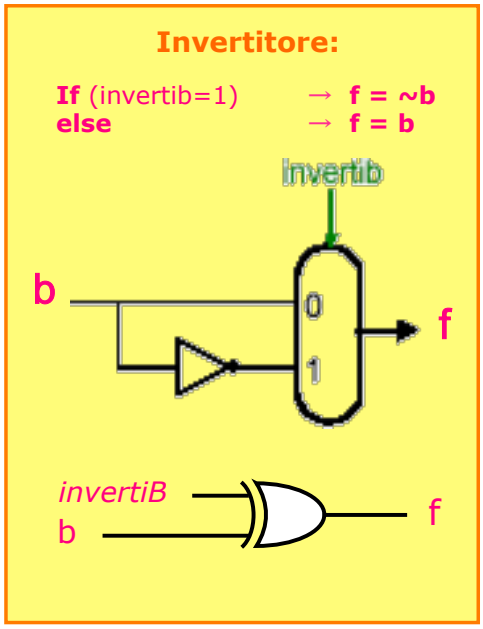
- **Sottrazione** → addizione dell'opposto: **a – b = a + (– b)**
  - Posso farlo con gli stessi circuiti dell'addizione, ma devo costruire –b a partire da b

- **Complemento a 2:**
  – b = not(b) + 1
  - Inversione logica bit-a-bit
  - Aggiunta della costante "1": pongo $r_{in}(0) = 1$

Inversione logica bit-a-bit:

invertiB	b	f
0	0	0
0	1	1
1	0	1
1	1	0

f = *b* **XOR** *invertiB*

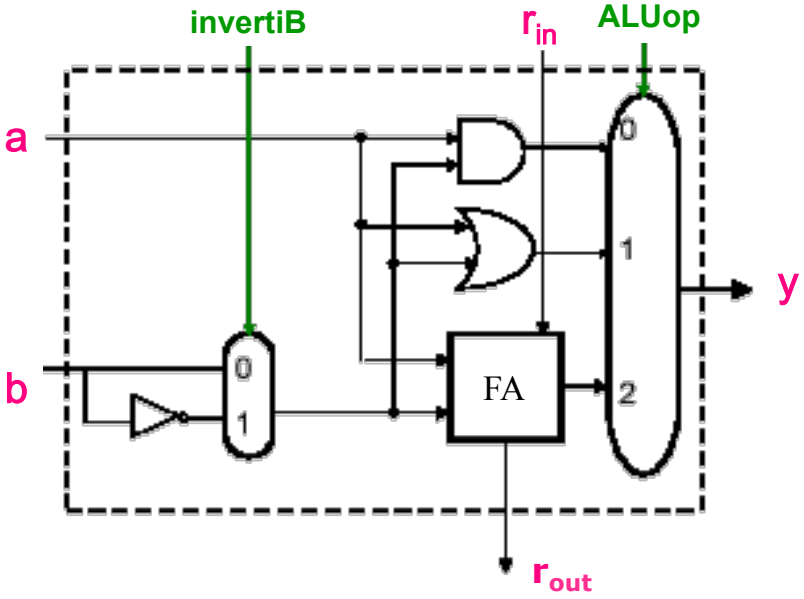


ALU - bit i-esimo

Operazioni: **AND , OR , + , –**
Propagazione riporti: $r_{in}(i) = r_{out}(i-1)$ i = 1, 2, 3, ...31

Addizione:
$r_{in}(0) = 0$, invertiB = 0

Sottrazione:
$r_{in}(0) = 1$, invertiB = 1



Comparazione (confronto)

- **Comparazione:**

  if **a** < **b** then **y** = 1 (y = 0...01)

  - Fondamentale per dirigere il flusso di esecuzione (test, cicli...)
    if (a<b) ⟶ y = [0 0 0 ... 0 **1**]
    else ⟶ y = [0 0 0 ... 0 **0**]

- Implementazione:

  if ALUop = "comparazione"
  then y(i) = 0, i = 1, 2, 3, ... , N–1
  if (a < b) y(0) = 1
  else y(0) = 0

  Devo:
  - Imporre tutti i bit di **y** (tranne y_0) a 0;
  - Calcolare y_0 in base alla condizione a<b

Come sviluppare la comparazione?

- IDEA: in complemento a 2, il MSB della somma (bit di segno) = 1 per numeri negativi → $s_{MSB} = 1$

a < b → a – b < 0 → $s_{MSB} = 1$

Implementazione:

- Nuovo ingresso: **LESS**

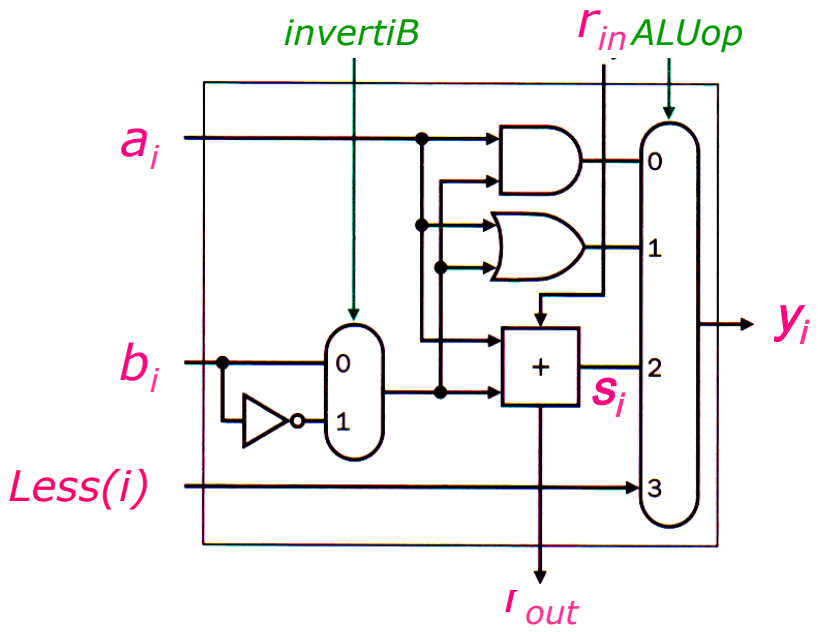
  IF:ALUop = "comparazione" → $s_i = LESS_i$

- Operazioni:
  - Calcolare la differenza (a – b) (senza mandarla in uscita)
  - Inviare l'uscita del sommatore del **MSB** a **LESS di ALU_0**

    $s_{N-1} \rightarrow LESS_0$
  - Questa uscita viene chiamata **segnale di set**

Less(i) ← 0 **i = 1, 2, 3, …. N–1**

Less(0) ← s_{31} **iff (a < b) & (S = comparazione)**



Overflow

- Esempio decimale:
  - **a + b = c** dove **a,b,c tutti** codificati con 2 cifre decimali
  - **a = 19, b = 83**
  - **Overflow: 19 + 83 = (1)02**
- Supponendo il MSB dedicato al bit di segno…
  - 019 + 083 = 102
  - L'overflow modifica il **MSB** (in compl. a 2, dedicato al **segno**)
- **Overflow nella somma** quando:

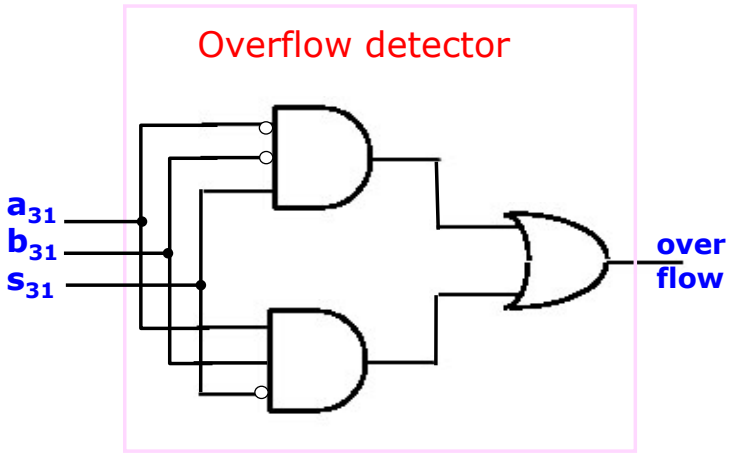
  a + b = s, a > 0, b > 0 → **MSB di a e b = 0, MSB di s = 1**

  a + b = s, a < 0, b < 0 → **MSB di a e b = 1, MSB di s = 0**
- Si può avere overflow con **a** e **b** di segno opposto ?

Circuito di riconoscimento dell'overflow

- 3 ingressi, tutti dalla ALU_{N-1}:
  - MSB di a, b e somma: a_{N-1} b_{N-1} s_{N-1}

a_{31}	b_{31}	s_{31}	overflow
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0



Overflow

Overflow nella differenza:

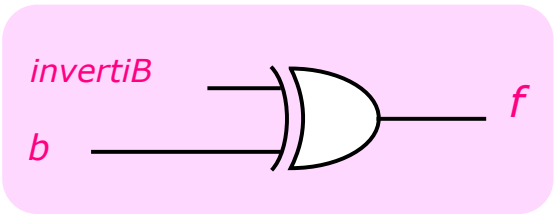
019 – (–**0**83) = **1**02

- **Overflow nella differenza** quando:

a + (–b) = s, a > 0, (–b) > 0 → **MSB di a e (–b) = 0, MSB di s = 1**
a + (–b) = s, a < 0, (–b) < 0 → **MSB di a e (–b) = 1, MSB di s = 0**

\+ : MSB di **b**
– : MSB di (**–b**) = MSB di **NOT(b)**

➔ +/– : MSB di **f**

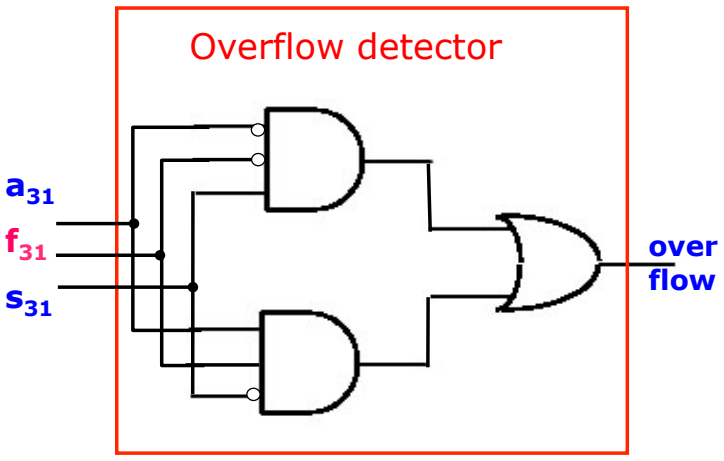


- Utilizzando **f**, a valle dello XOR, anziché **b**, il rivelatore di overflow funziona sia per la somma che per la differenza!

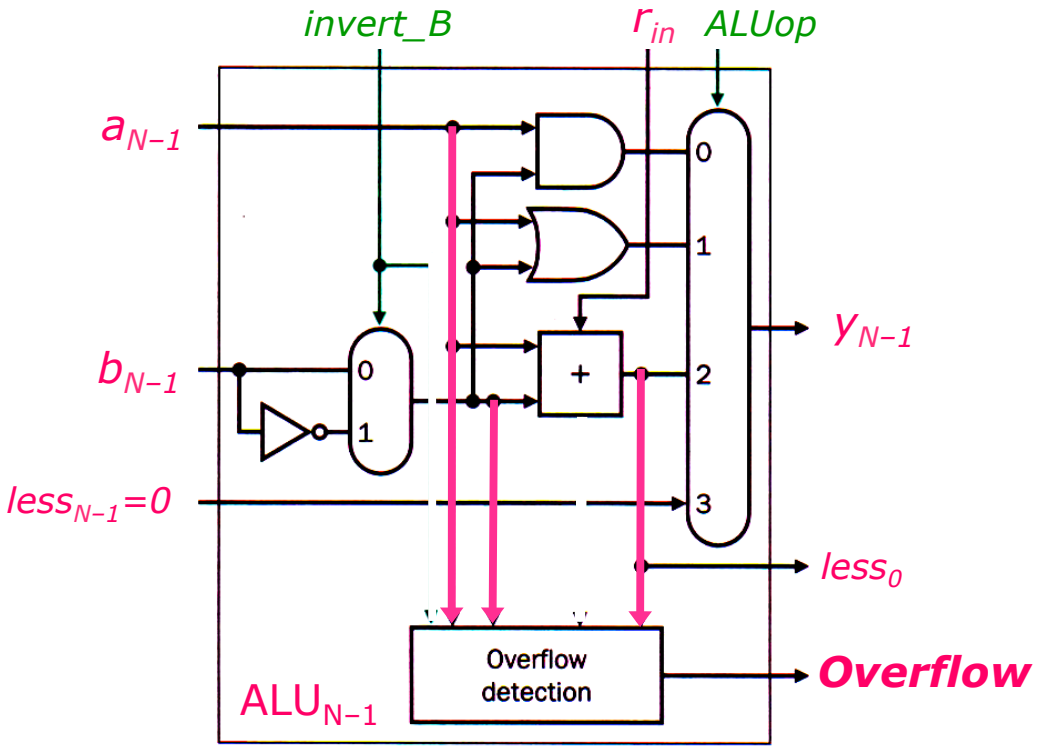
Circuito di riconoscimento dell'overflow

- 3 ingressi, tutti dalla ALU31:
  - MSB di a, b e somma: a_{31} b_{31} s_{31}

a_{31}	f_{31}	s_{31}	overflow
0	0	0	0
0	**0**	**1**	**1**
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	**1**	**0**	**1**
1	1	1	0



Struttura ALU_{N-1} con Overflow detector



ALU completa a 32 bit

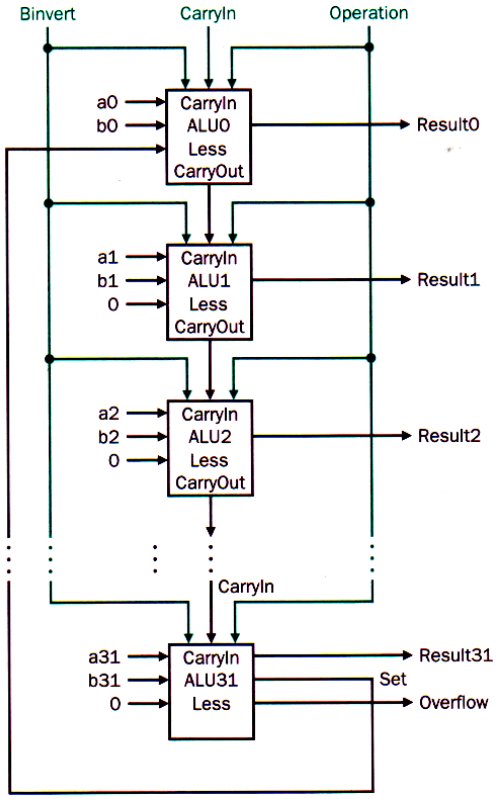
- *InvertiB e $r_{IN}(0)$ sono lo stesso segnale*

Test di uguaglianza:

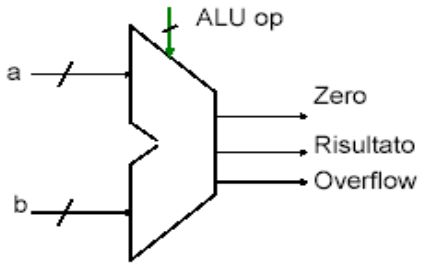
- *Operazioni necessarie*
  - Impostare una differenza.
  - Effettuare l' OR di tutti i bit somma.
  - Uscita dell' OR = 0 → i due numeri sono uguali

Operazioni possibili:

- AND
- OR
- Somma / Sottrazione
- Comparazione
- Test di uguaglianza



ALU a 32 bit: struttura finale



ALUop	funzione
000	and
001	or
010	+ (add)
110	– (sub)
111	set less than