

ARCHITETTURA DEGLI ELABORATORI

A.A. 2020-2021

Università di Napoli Federico II
Corso di Laurea in Informatica

Docenti

Proff.

Luigi Sauro gruppo 1 (A-G)

Silvia Rossi gruppo 2 (H-Z)



Parallelismo

- **2 tipi di parallelismo:**
 - **Spaziale**
 - duplicare l'hardware per eseguire più task contemporaneamente
 - **Temporale**
 - Il task è suddiviso in più fasi
 - Le diverse fasi sono eseguite in pipelining

Latenza e throughput

- **Token:** Gruppo di input da processare per ottenere un output significativo
- **Latency:** Tempo che occorre ad un token per essere processato e produrre un output
- **Throughput:** Numero di output prodotti per unità di tempo

il parallelismo incrementa il throughput

Esempio di parallelismo

- Ben vuole fare delle torte
- 5 minuti per preparare una torta
- 15 minuti per cucinarla

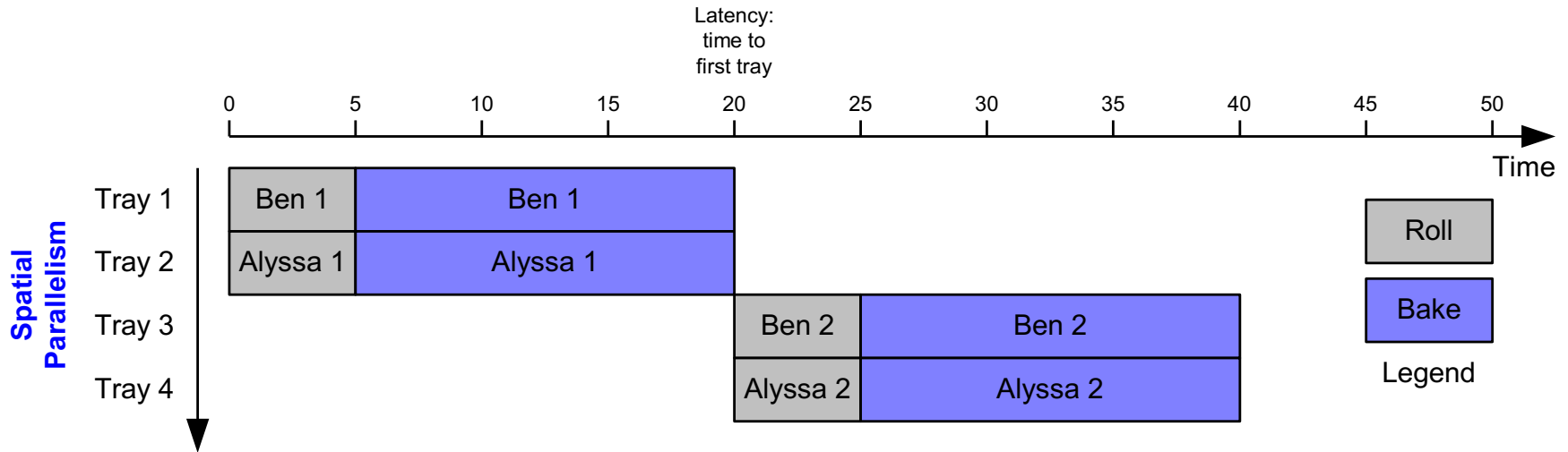
$$\text{Latency} = 5 + 15 = 20 \text{ minuti} = \mathbf{1/3 \text{ h}}$$

$$\text{Throughput} = \frac{1}{\text{latency}} = \mathbf{3 \text{ torte/h}}$$

Esempio di parallelismo

- **parallelismo spaziale:** Ben chiede a Allysa di aiutarlo, usando anche il suo forno
- **parallelismo temporale:**
 - 2 fasi: preparare e cucinare
 - Mentre una torta è in forno, Ben prepara ne prepara un'altra, etc.

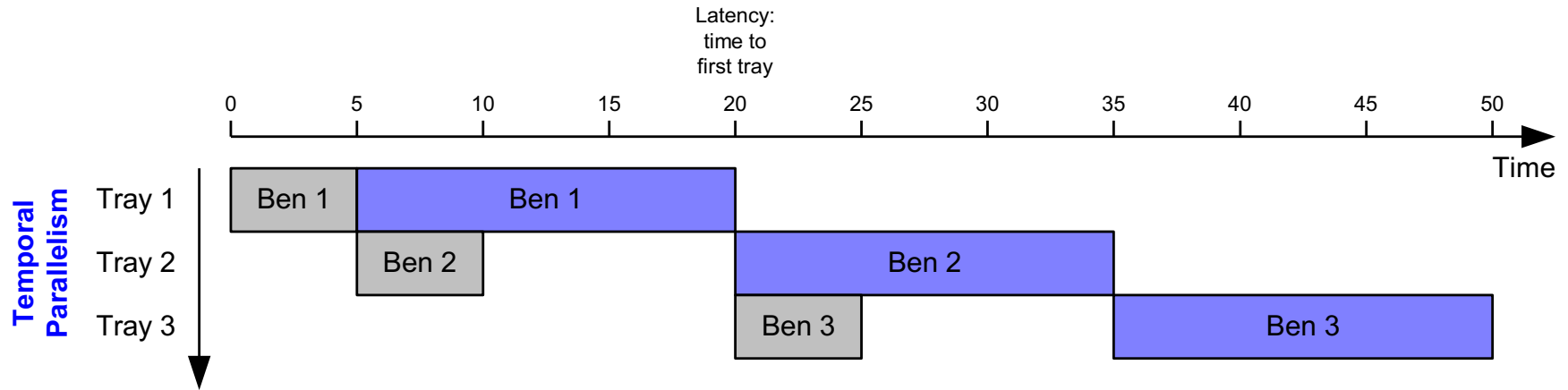
Parallelismo spaziale



$$\text{Latency} = 5 + 15 = 20 \text{ minuti} = \mathbf{1/3 \text{ h}}$$

$$\text{Throughput} = 2 \frac{1}{\text{latency}} = \mathbf{6 \text{ torte/h}}$$

Parallelismo temporale






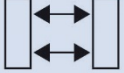
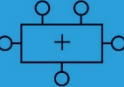
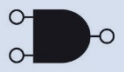
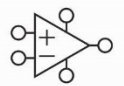


$$\text{Latency} = 5 + 15 = 20 \text{ minuti} = \mathbf{1/3 \text{ h}}$$

$$\text{Throughput} \approx \frac{4}{65} 60 \approx \mathbf{3,7 \text{ torte/h}}$$

CIRCUITI ARITMETICI E MEMORIE

Chapter 5 :: Topics

- Introduction
- Arithmetic Circuits
- Number Systems
- Sequential Building Blocks
- Memory Arrays
- Logic Arrays

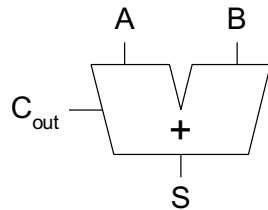
Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

Introduction

- **Digital building blocks:**
 - Gates, multiplexers, decoders, registers, arithmetic circuits, counters, memory arrays, logic arrays
- **Building blocks demonstrate hierarchy, modularity, and regularity:**
 - Hierarchy of simpler components
 - Well-defined interfaces and functions
 - Regular structure easily extends to different sizes
- **Will use these building blocks in Chapter 7 to build microprocessor**

1-Bit Adders

Half Adder

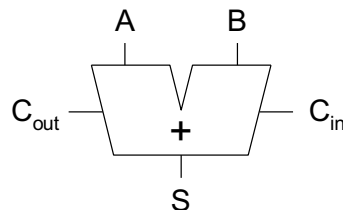


A	B	C_{out}	S
0	0		
0	1		
1	0		
1	1		

S =

C_{out} =

Full Adder



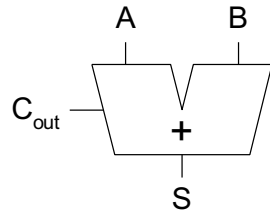
C_{in}	A	B	C_{out}	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

S =

C_{out} =

1-Bit Adders

Half Adder

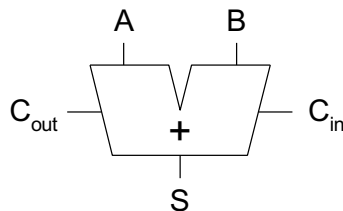


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

S =

C_{out} =

Full Adder



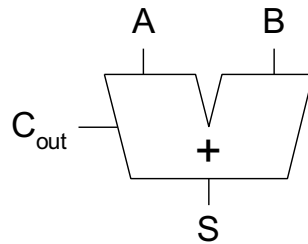
C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

S =

C_{out} =

1-Bit Adders

Half Adder

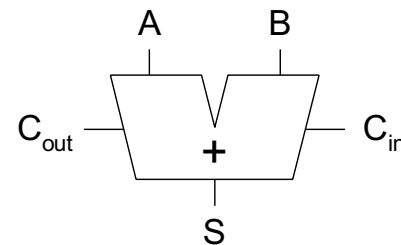


A	B	C _{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = A \oplus B$$

$$C_{out} = AB$$

Full Adder



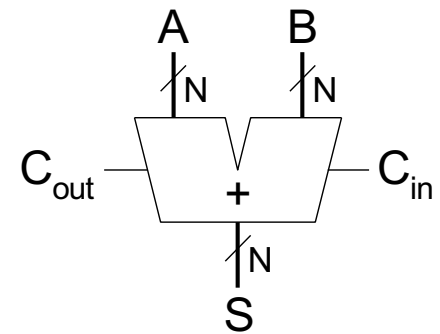
C _{in}	A	B	C _{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$

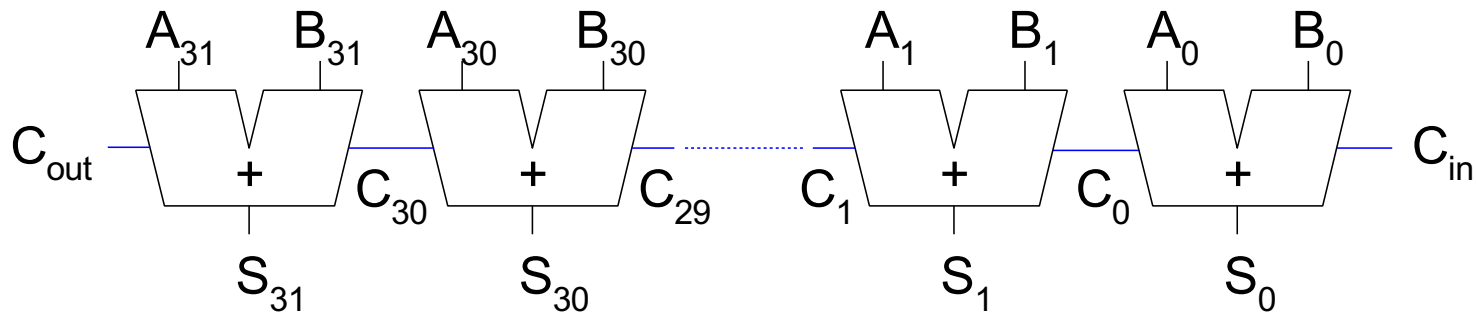
Multibit Adders (CPAs)

- Tipologie carry propagate adders (CPAs):
 - Ripple-carry (lento)
 - Carry-lookahead (veloce)
- Carry-lookahead adders offrono prestazioni migliori ma richiedono più hardware



Ripple-Carry Adder

- 1-bit adder concatenate insieme
- Il riporto si propaga lungo la catena
- Svantaggio: **lento**



Ritardo del Ripple-Carry

$$t_{\text{ripple}} = N t_{FA}$$

t_{FA} è il ritardo di un singolo 1-bit full adder

N è il numero di 1-bit full adders

Carry-Lookahead Adder

Calcolare C_{out} di un blocco di k -bit usando le funzioni *generate* e *propagate*

C_{in}	A_i	B_i	C_{out}
0	0	0	0
1			0
0	0	1	0
1			1
0	1	0	0
1			1
0	1	1	1
1			1

Carry-Lookahead Adder

C_{in}	A_i	B_i	C_{out}
0	0	0	0
1			0
0	0	1	0
1			1
0	1	0	0
1			1
0	1	1	1
1			1

Carry-Lookahead Adder

- Il bit *i-esimo* produce un riporto (carry out) per generazione o per propagazione di un riporto del bit *(i-1)-esimo* (carry in)
- **Generate:** Il bit *i-esimo* genera un carry out se A_i e B_i sono entrambi 1.

$$G_i = A_i B_i$$

- **Propagate:** Il bit *i-esimo* propaga un carry al carry out se A_i o B_i sono 1.

$$P_i = A_i + B_i$$

- **Carry out:** il carry *i* (C_i) è data da:

$$C_i = A_i B_i + (A_i + B_i) C_{i-1} = G_i + P_i C_{i-1}$$

Block Propagate and Generate

Ora dobbiamo estendere le funzioni Propagate and Generate a blocchi di k -bits, ovvero:

- calcolare se un blocco di k -bit $(i, i+1, \dots, i+k-1)$ propaga il carry out del blocco precedente (ovvero da C_{i-1} a C_{i+k-1})
- calcolare se un blocco di k -bit $(i, i+1, \dots, i+k)$ genera un carry out (in C_{i+k-1})

Esempi

0	0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0
	0	0	1	0	1	0	0	1	0	1	0	1	0	0	0	1
	0	0	1	0	0	1	1	0	1	1	0	0	0	0	0	0
	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	1

0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
	0	0	1	0	1	0	0	1	0	1	0	1	0	0	0	1
	0	0	1	0	0	1	1	0	0	1	0	0	0	0	0	0
	0	1	0	0	1	1	1	1	1	0	0	1	0	0	0	1

Esempi

0	0	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0
	0	0	1	0	1	0	0	1	0	1	0	1	0	0	0	1
	0	0	1	0	0	0	1	0	1	1	0	0	0	0	0	0
	0	1	0	0	1	1	0	0	0	0	0	1	0	0	0	1

0	0	1	0	1	1	1	0	0	1	0	0	0	0	0	0	0
	0	0	1	0	1	0	1	1	0	1	0	1	0	0	0	1
	0	0	1	0	0	1	1	0	0	1	0	0	0	0	0	0
	0	1	0	1	0	0	0	1	1	0	0	1	0	0	0	1

Esempi

0	0	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0
	0	0	1	0	1	0	0	1	0	1	0	1	0	0	0	1
	0	0	1	0	0	0	1	0	1	1	0	0	0	0	0	0
	0	1	0	0	1	1	0	0	0	0	0	1	0	0	0	1

0	0	1	0	0	1	1	0	0	1	0	0	0	0	0	0	0
	0	0	1	0	0	0	1	1	0	1	0	1	0	0	0	1
	0	0	1	0	0	1	1	0	0	1	0	0	0	0	0	0
	0	1	0	0	1	0	0	1	1	0	0	1	0	0	0	1

Block Propagate and Generate Signals

- Block propagate and generate per un blocco di 4-bit ($P_{3:0}$ and $G_{3:0}$):

$$P_{3:0} = P_3 P_2 P_1 P_0$$

$$G_{3:0} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 = \\ G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0))$$

- In generale,

$$P_{i:j} = P_i P_{i-1} P_{i-2} \dots P_j$$

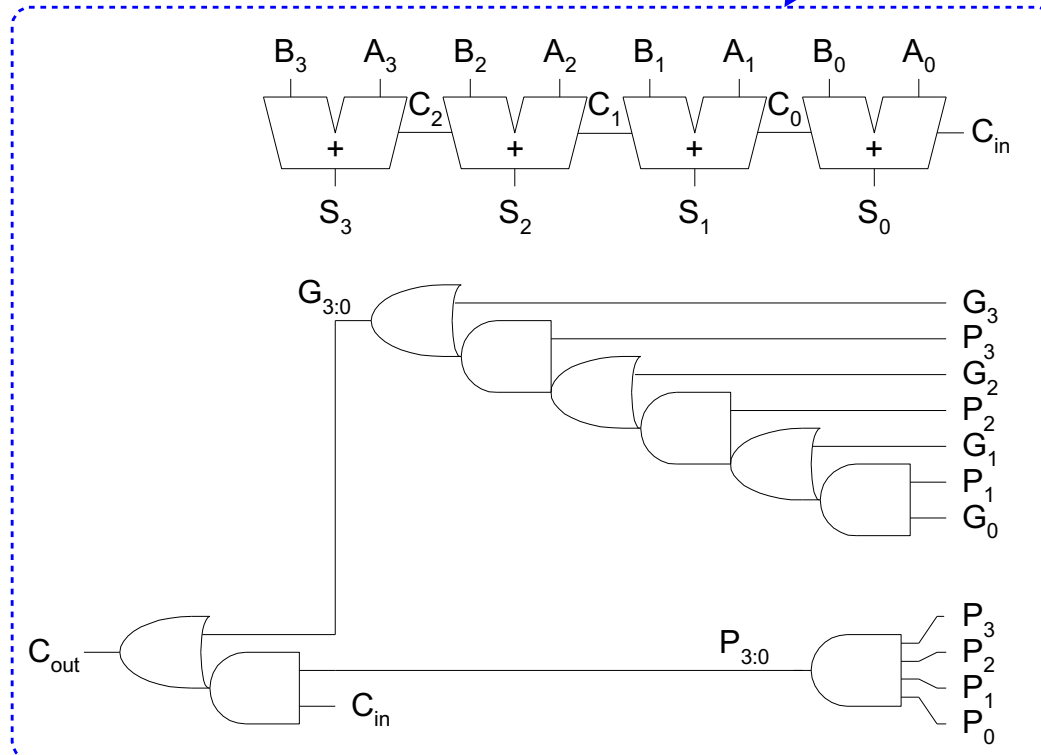
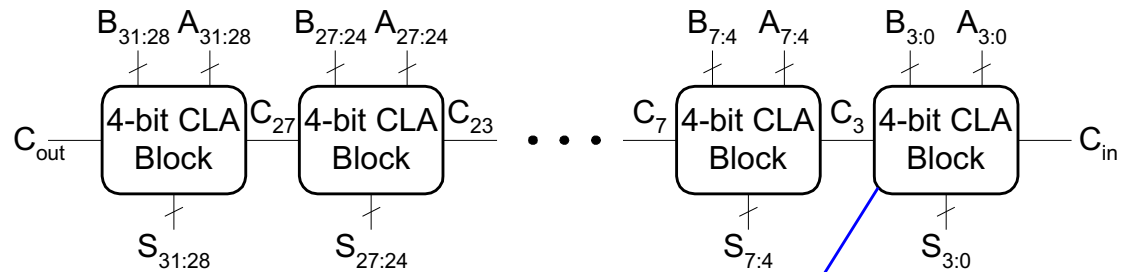
$$G_{i:j} = G_i + P_i (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} (\dots G_j) \dots))$$

$$C_i = G_{i:j} + P_{i:j} C_{j-1}$$

Carry-Lookahead Addition

- **Step 1:** calcola i G_i and P_i per tutte le colonne
- **Step 2:** calcola G and P per blocchi di k -bit
- **Step 3:** C_{in} si propaga mediante la logica propagate/generate dei vari blocchi di k -bit (mentre si calcolano le somme)

32-bit CLA with 4-bit Blocks



$$C_i = G_{i:j} + P_{i:j} C_{j-1}$$

Ritardo del Carry-Lookahead Adder

Per un N -bit CLA con blocchi di k bit:

$$t_{CLA} = t_{pg} + t_{pg_block} + (N/k - 1)t_{AND_OR} + kt_{FA}$$

- t_{pg} : ritardo per generare P_i, G_i (un AND + OR)
- t_{pg_block} : ritardo per generare $P_{i:j}, G_{i:j}$
- t_{AND_OR} : ritardo delle porte AND/OR a monte della logica propagate/generate, questo ritardo si propaga da C_{in} a C_{out} che si propaga lungo i $N/k - 1$ blocchi

Un N -bit carry-lookahead adder è generalmente più veloce di un ripple-carry adder per $N > 16$

Confronto

- 32-bit ripple-carry vs 32-bit CLA con blocchi di 4 bit
- 2-input gate delay = 100 ps; full adder delay = 300 ps

$$\begin{aligned}t_{\text{ripple}} &= Nt_{FA} = 32(300 \text{ ps}) \\ &= \mathbf{9.6 \text{ ns}}\end{aligned}$$

$$\begin{aligned}t_{CLA} &= t_{pg} + t_{pg_block} + (N/k - 1)t_{AND_OR} + kt_{FA} \\ &= [100 + 600 + (7)200 + 4(300)] \text{ ps} \\ &= \mathbf{3.3 \text{ ns}}\end{aligned}$$

Sottrattore

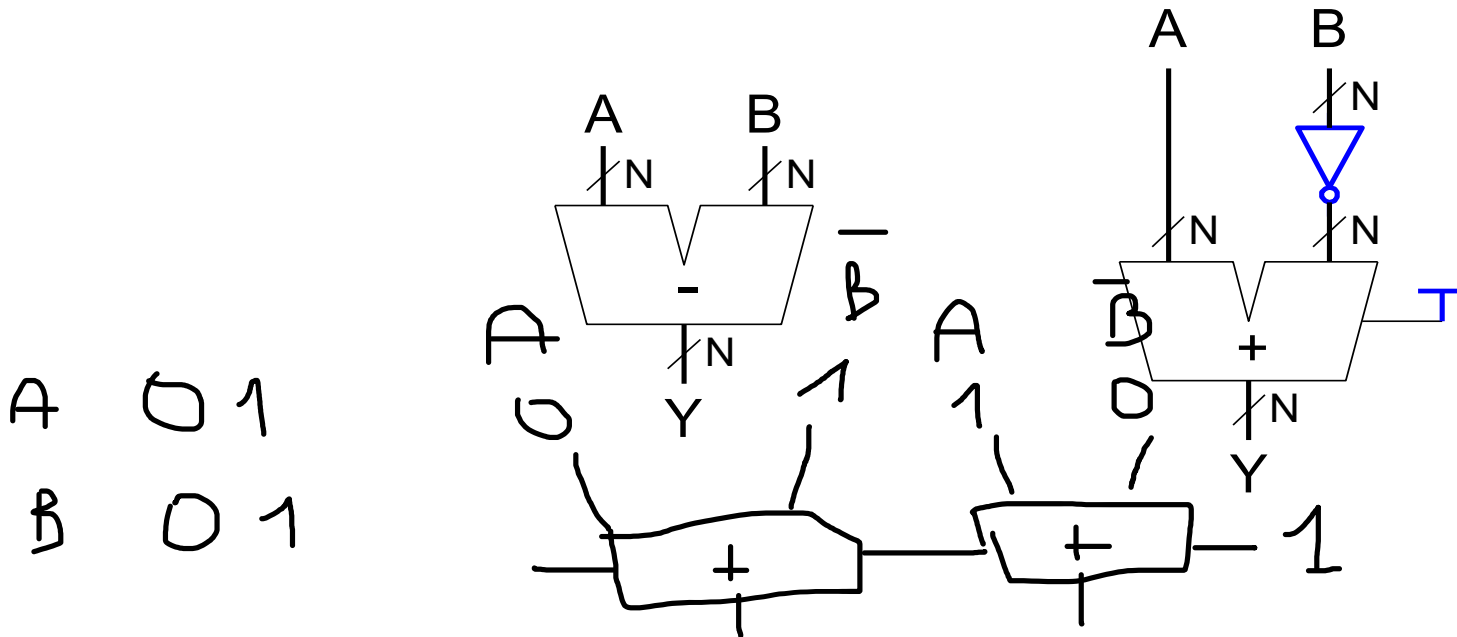
- Nella rappresentazione a complemento a 2 per complementare un numero occorre invertire ogni cifra e sommare 1

$$Y = A - B = A + \overline{B} + 1$$

- Es: $2 = 0010_2 \rightarrow -2 = 1101_2 + 0001_2 = 1110_2$

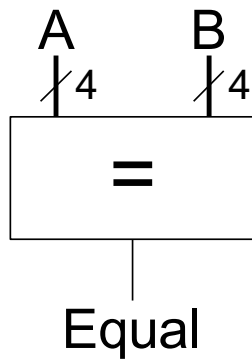
Symbol

Implementation

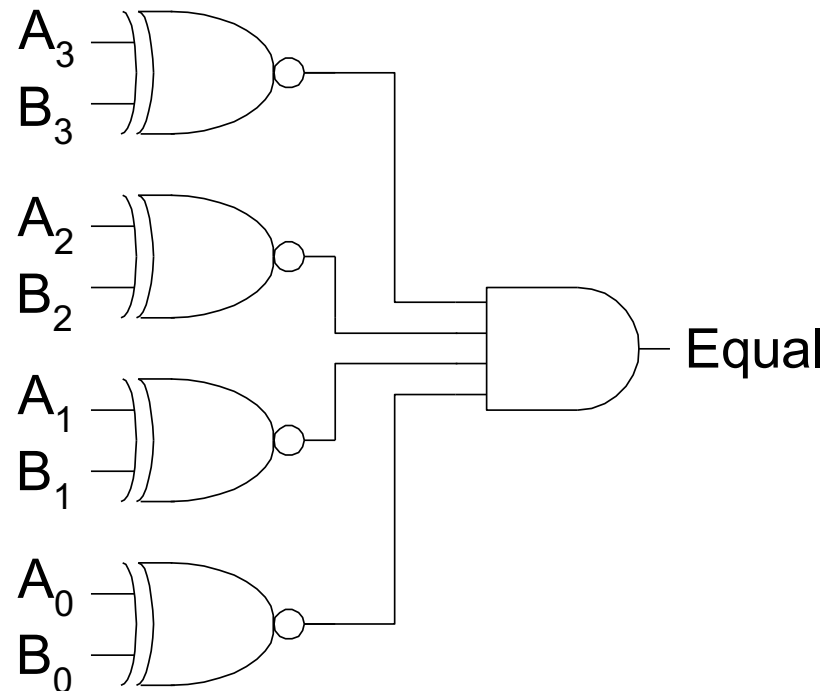


Comparatore: uguaglianza

Symbol

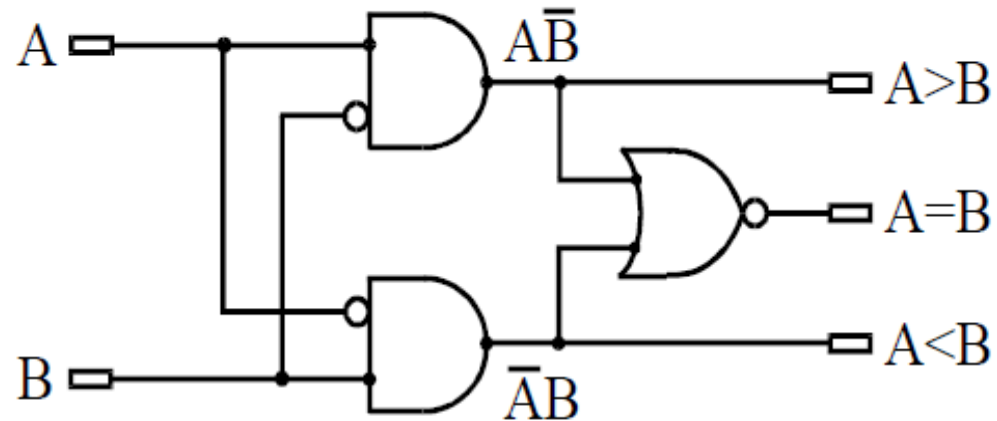


Implementation

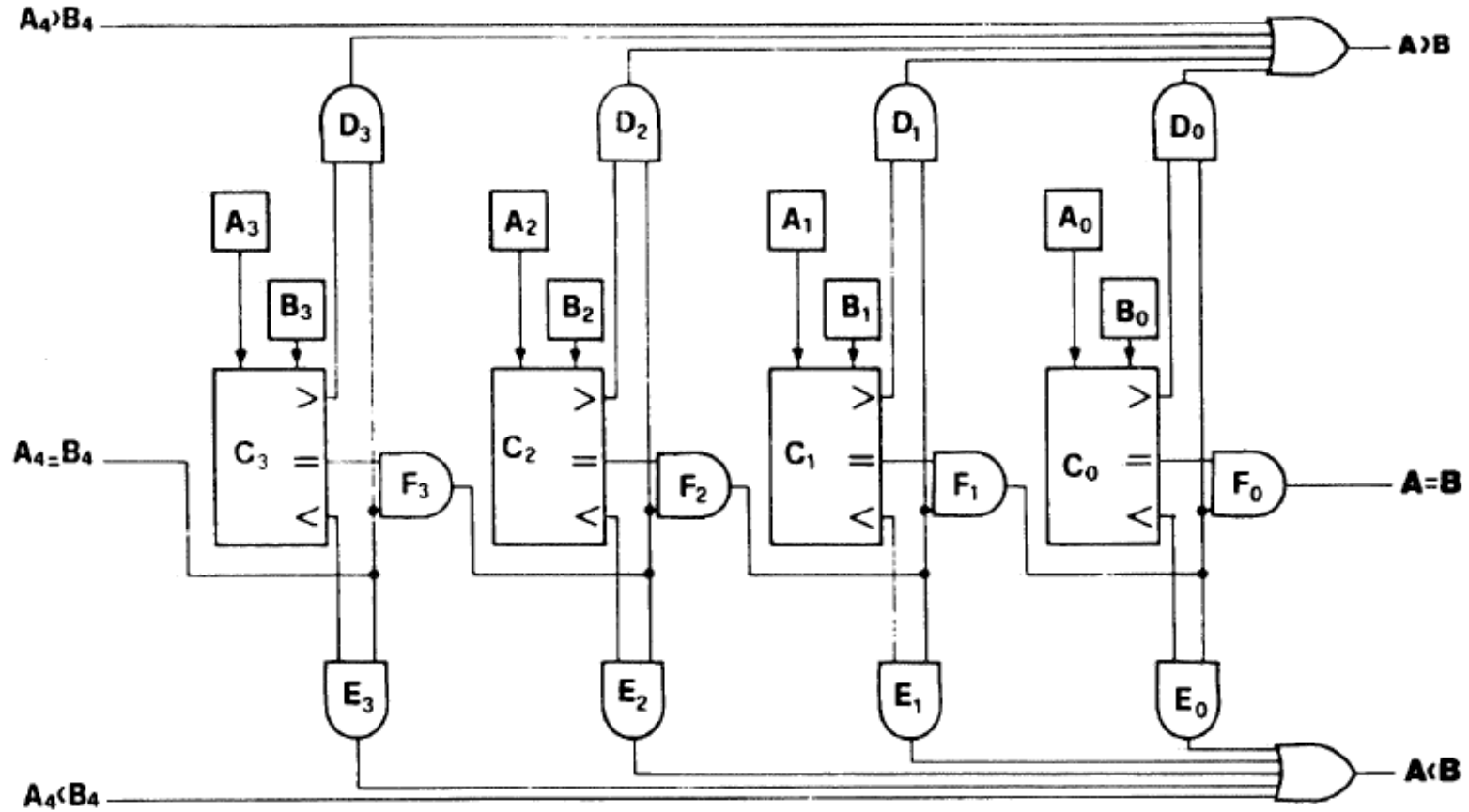


Comparatore completo a un bit

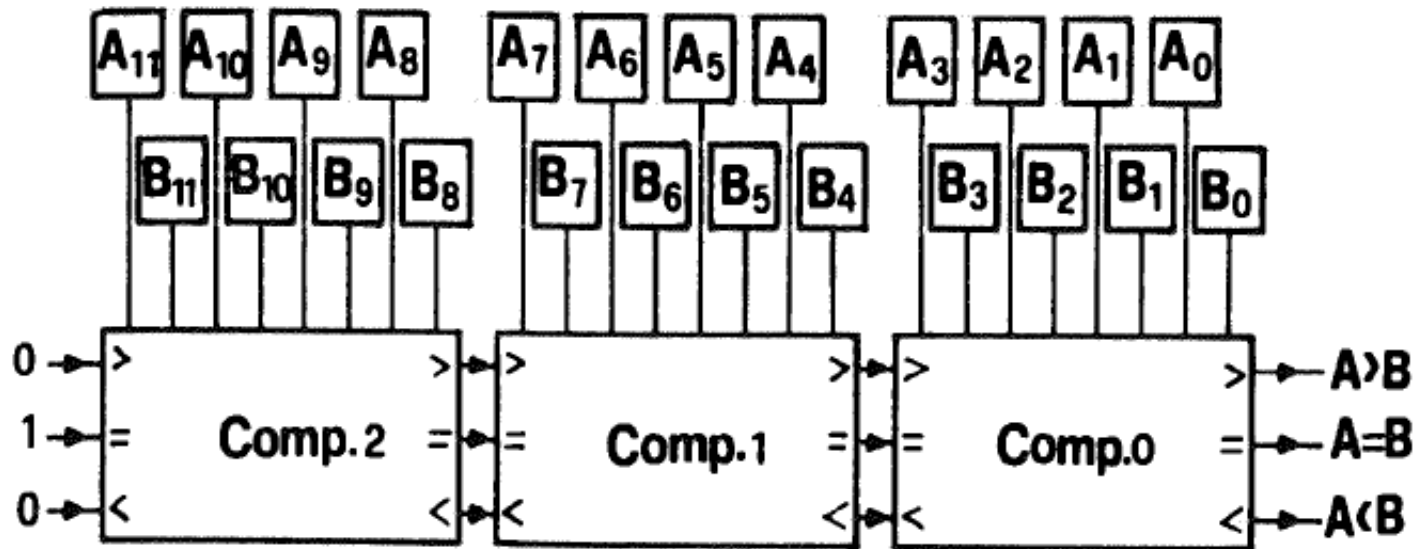
A	B	$A > B$	$A = B$	$A < B$
0	0	0	1	0
0	1	0	0	1
1	0	1	0	0
1	1	0	1	0



Comparatore completo a 4 bit

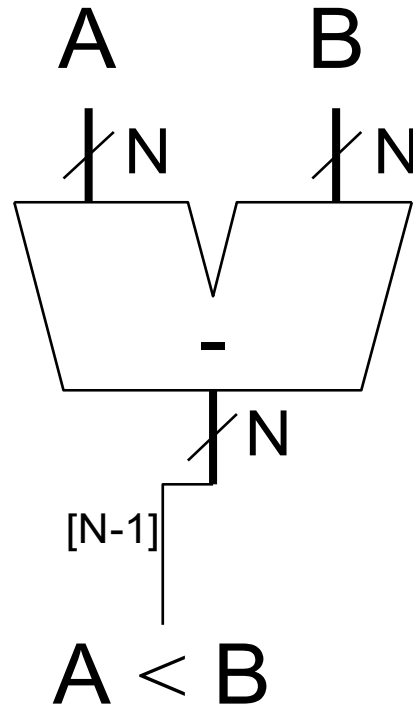


Comparatore completo a 12 bit



Comparator: Less Than

Se il bit del segno =1 allora $A < B$



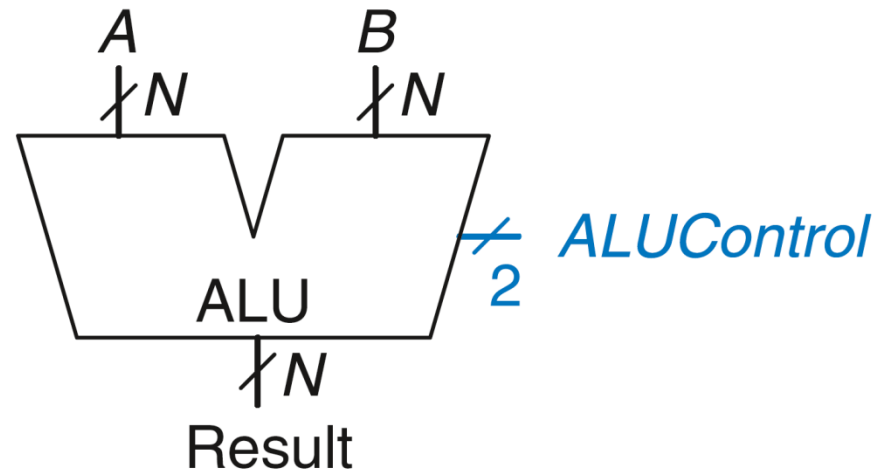
ALU: Arithmetic Logic Unit

Operazioni che ALU tipicamente esegue:

- Addizione
- Sottrazione
- AND
- OR

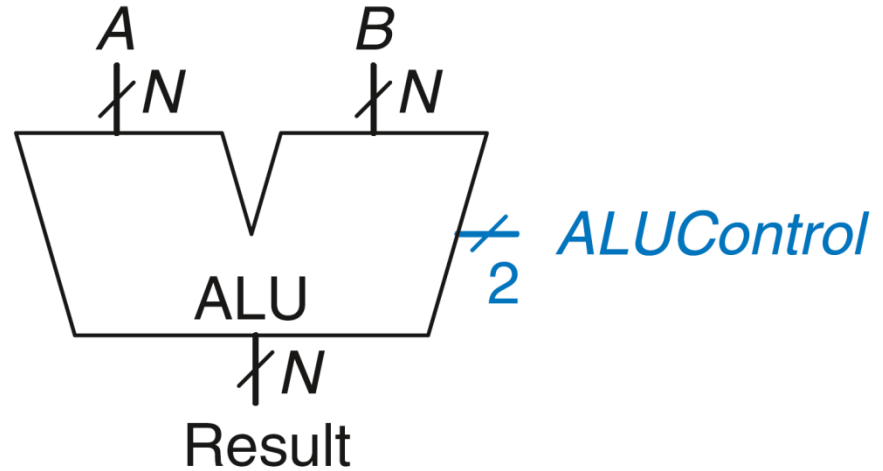
ALU: Arithmetic Logic Unit

ALUControl _{1:0}	Function
00	Add
01	Subtract
10	AND
11	OR



ALU: Arithmetic Logic Unit

ALUControl _{1:0}	Function
00	Add
01	Subtract
10	AND
11	OR



Example: Perform $A + B$

$ALUControl = 00$

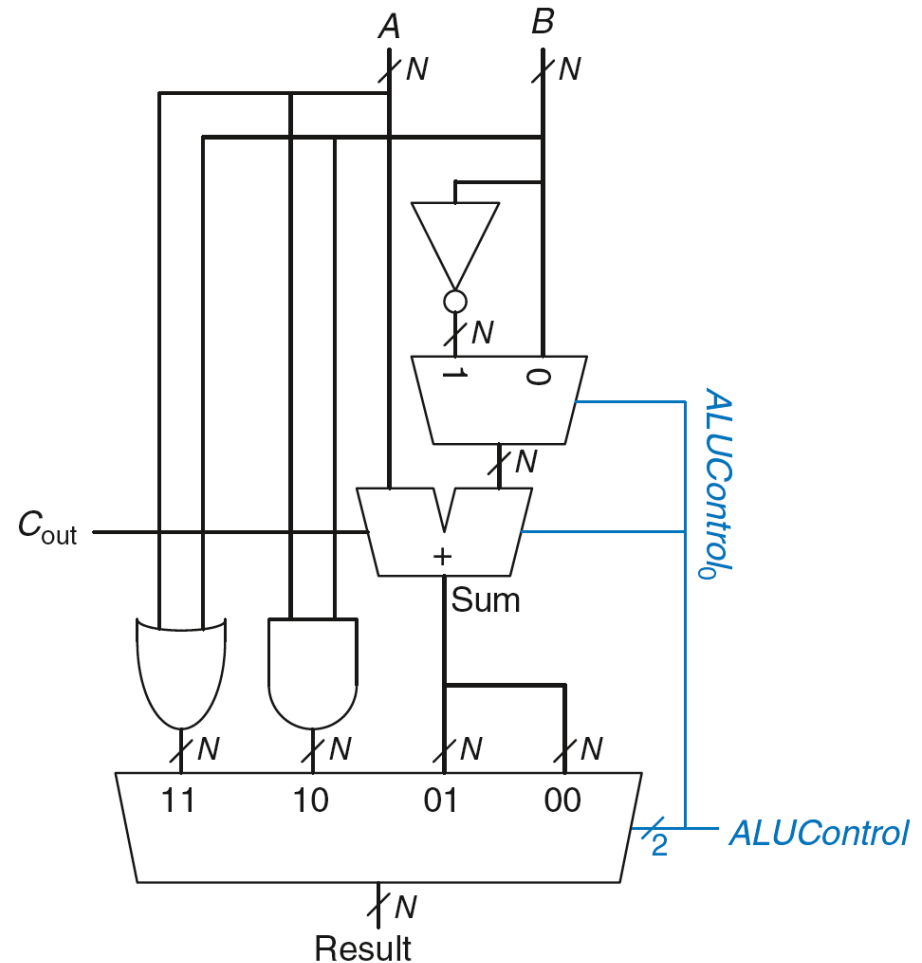
$Result = A + B$

ALU: Arithmetic Logic Unit

ALUControl _{1:0}	Function
00	Add
01	Subtract
10	AND
11	OR

Esempio: A OR B

$ALUControl_{1:0} = 11$



ALU: Arithmetic Logic Unit

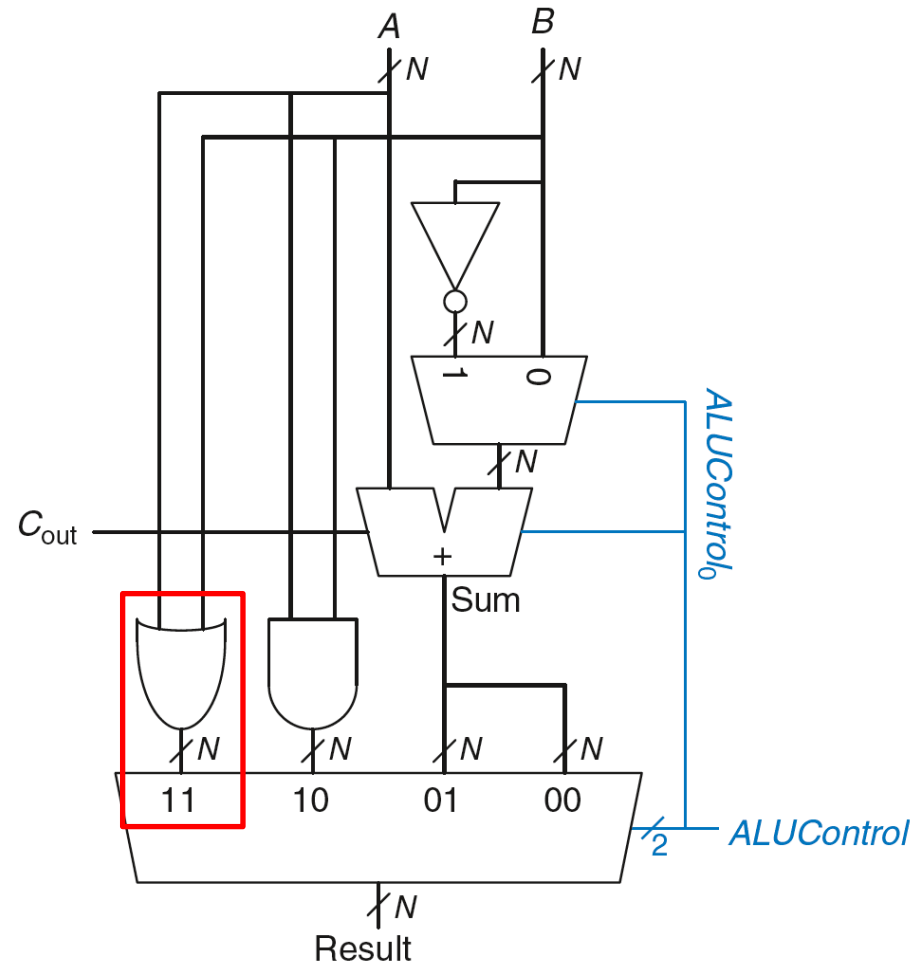
ALUControl _{1:0}	Function
00	Add
01	Subtract
10	AND
11	OR

Esempio: $A \text{ OR } B$

$ALUControl_{1:0} = 11$

Mux seleziona l'output della porta OR

$Result = A \text{ OR } B$



ALU: Arithmetic Logic Unit

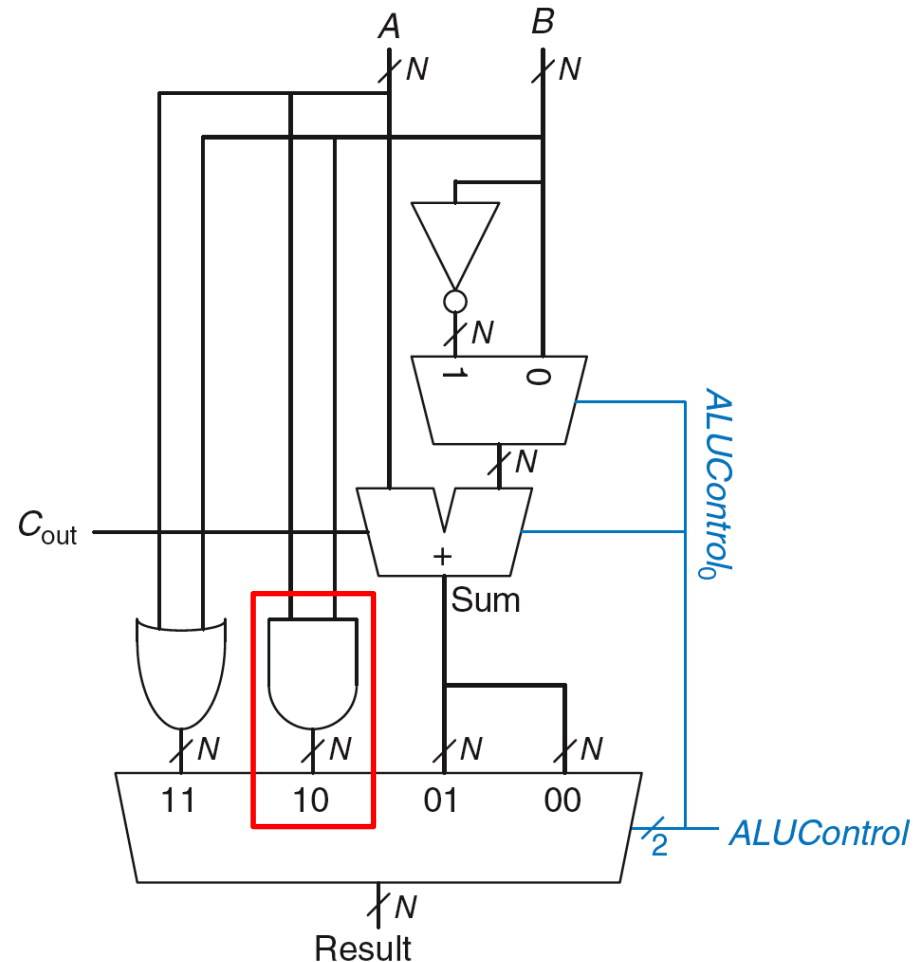
ALUControl _{1:0}	Function
00	Add
01	Subtract
10	AND
11	OR

Esempio: $A \text{ AND } B$

$ALUControl_{1:0} = 10$

Mux seleziona l'output della porta AND

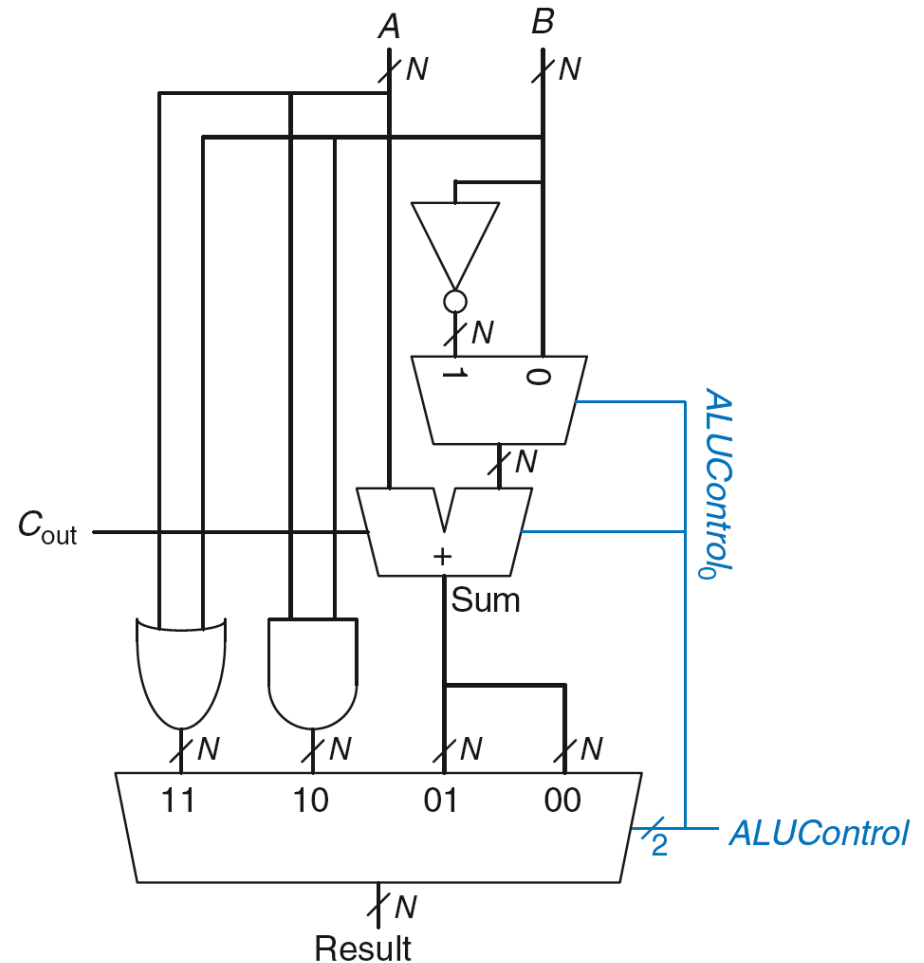
$Result = A \text{ AND } B$



ALU: Arithmetic Logic Unit

ALUControl _{1:0}	Function
00	Add
01	Subtract
10	AND
11	OR

Esempio: $A + B$

$$ALUControl_{1:0} = 00$$


ALU: Arithmetic Logic Unit

ALUControl _{1:0}	Function
00	Add
01	Subtract
10	AND
11	OR

Esempio: $A + B$

$ALUControl_{1:0} = 00$

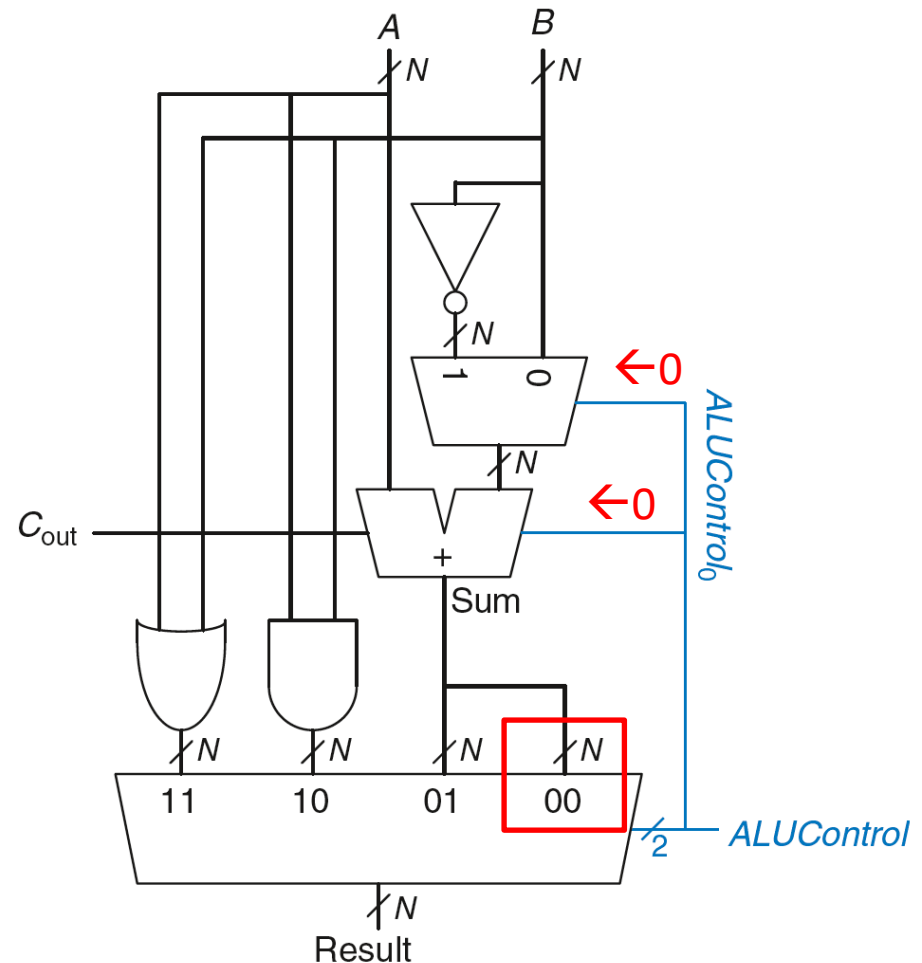
$ALUControl_0 = 0$, quindi:

$C_{in} = 0$

il 2nd input dell'adder è B

Mux seleziona Sum come $Result$

$Result = A + B$



ALU: Arithmetic Logic Unit

ALUControl _{1:0}	Function
00	Add
01	Subtract
10	AND
11	OR

Esempio: $A - B$

$ALUControl_{1:0} = 01$

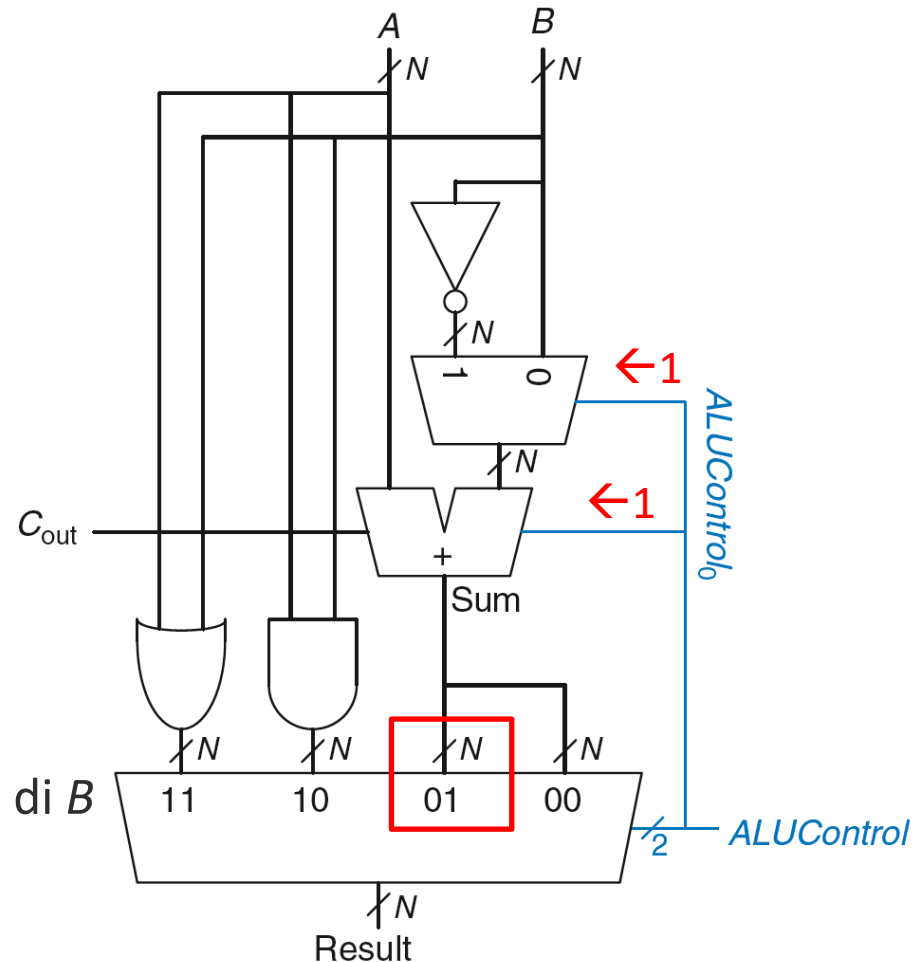
$ALUControl_0 = 1$, quindi:

$C_{in} = 1$

2nd input dell'adder è il complemento di B

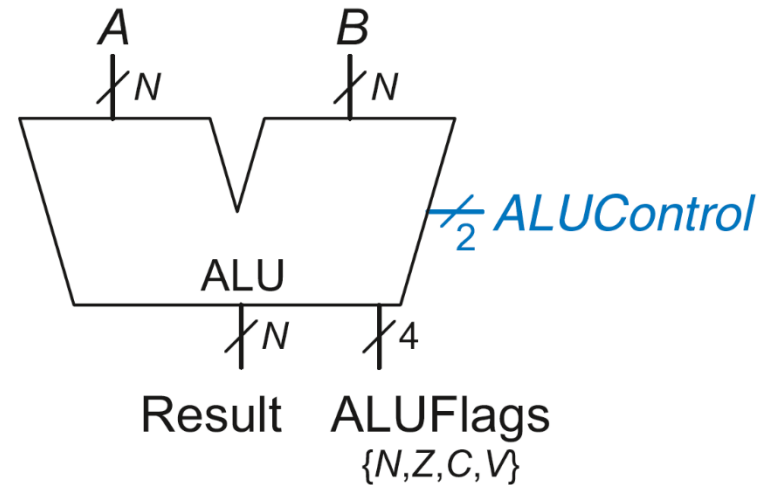
Mux seleziona Sum come $Result$

$Result = A - B$

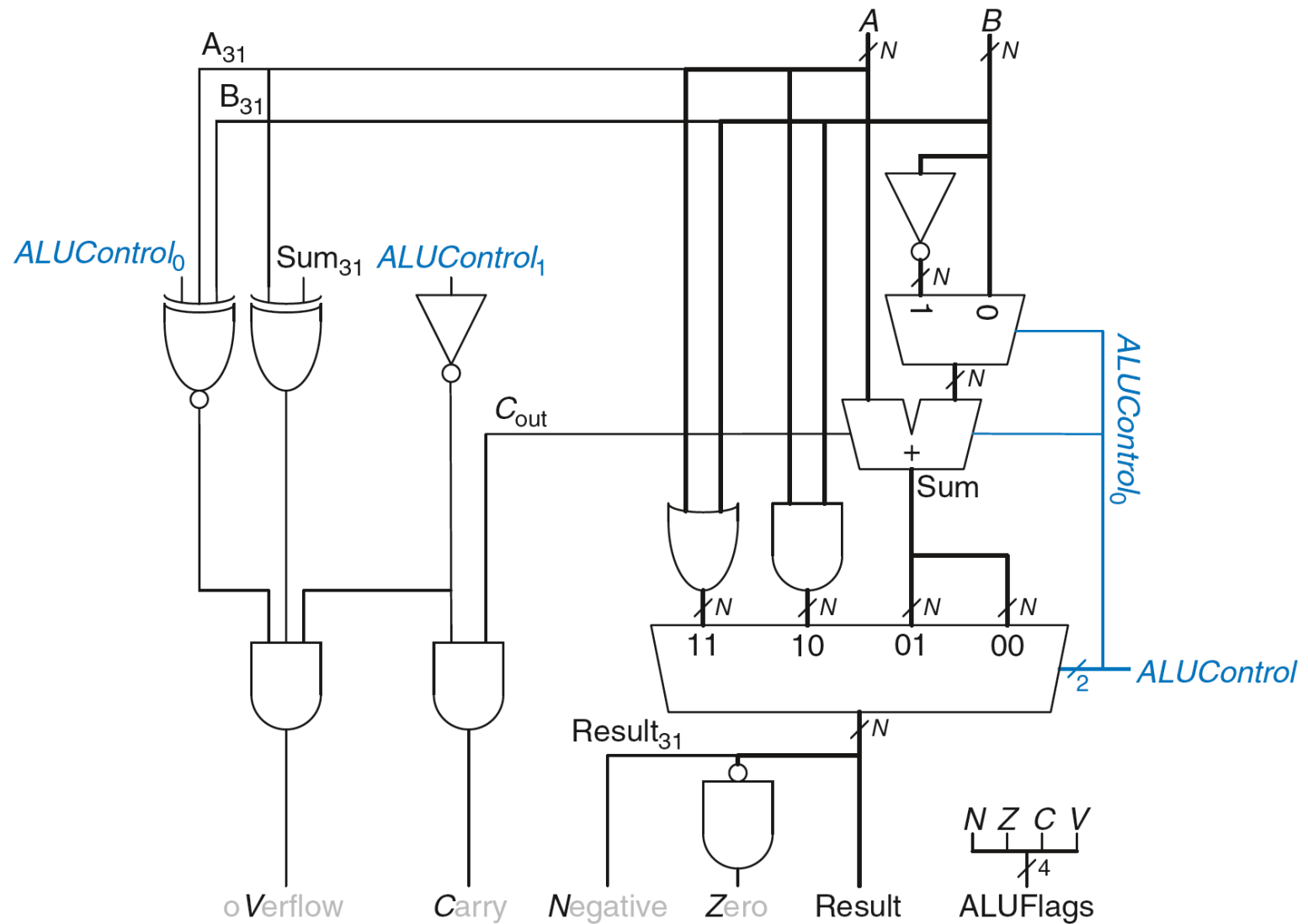


ALU con flags di stato

Flag	Description
<i>N</i>	Result is N egative
<i>Z</i>	Result is Z ero
<i>C</i>	Adder produces C arry out
<i>V</i>	Adder o V erflowed

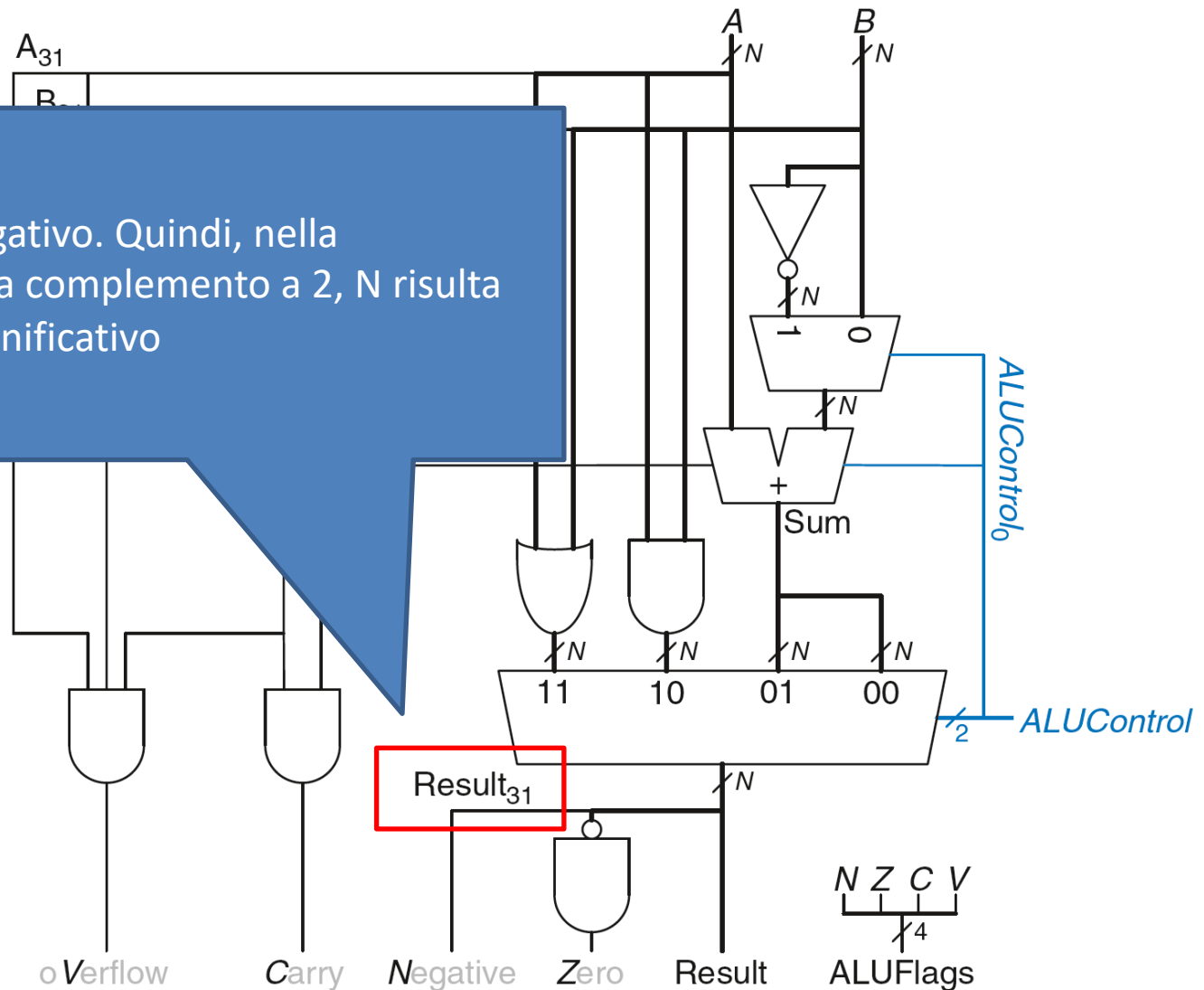


ALU con flags di stato

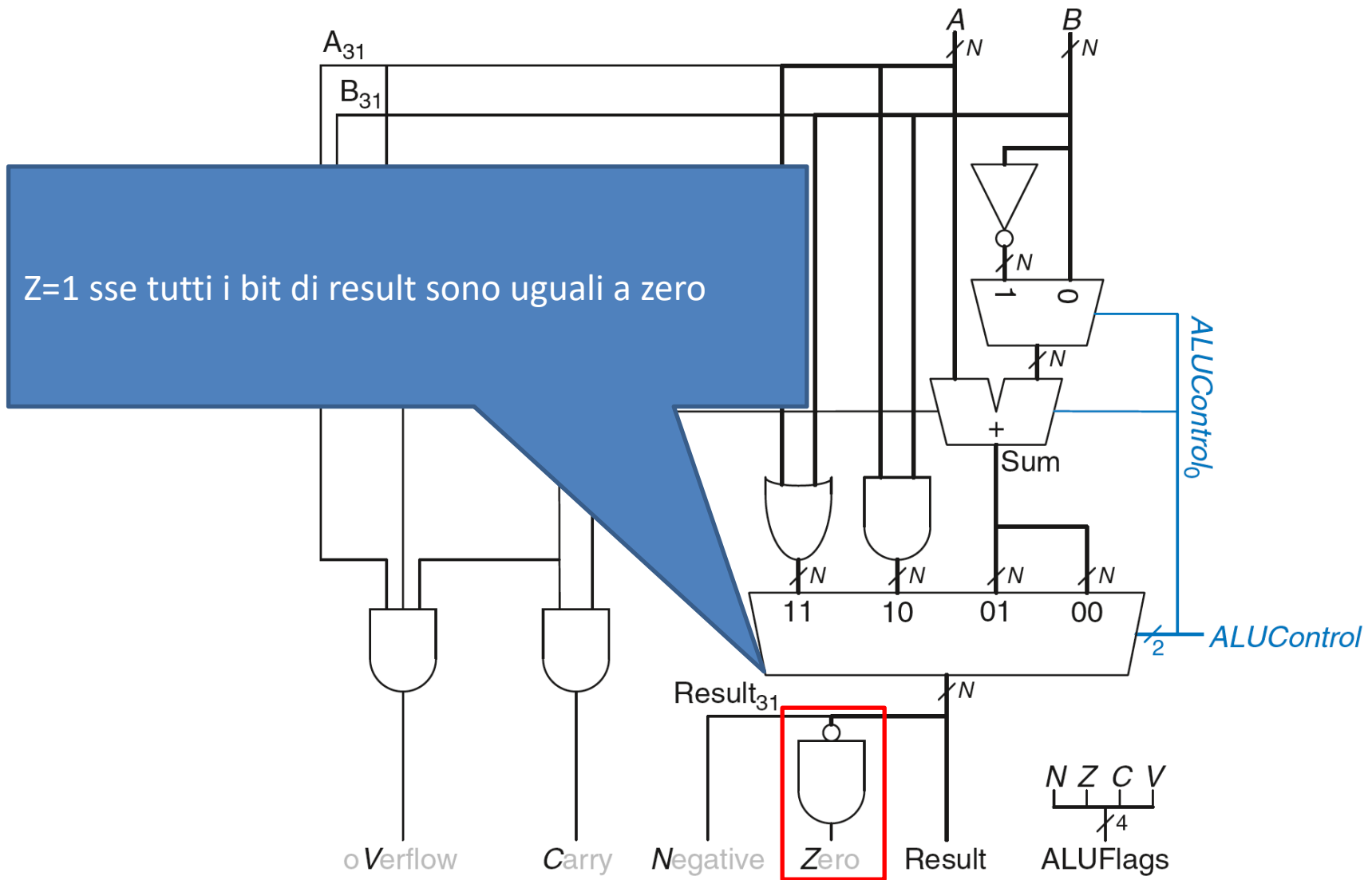


ALU con flags di stato: Negative

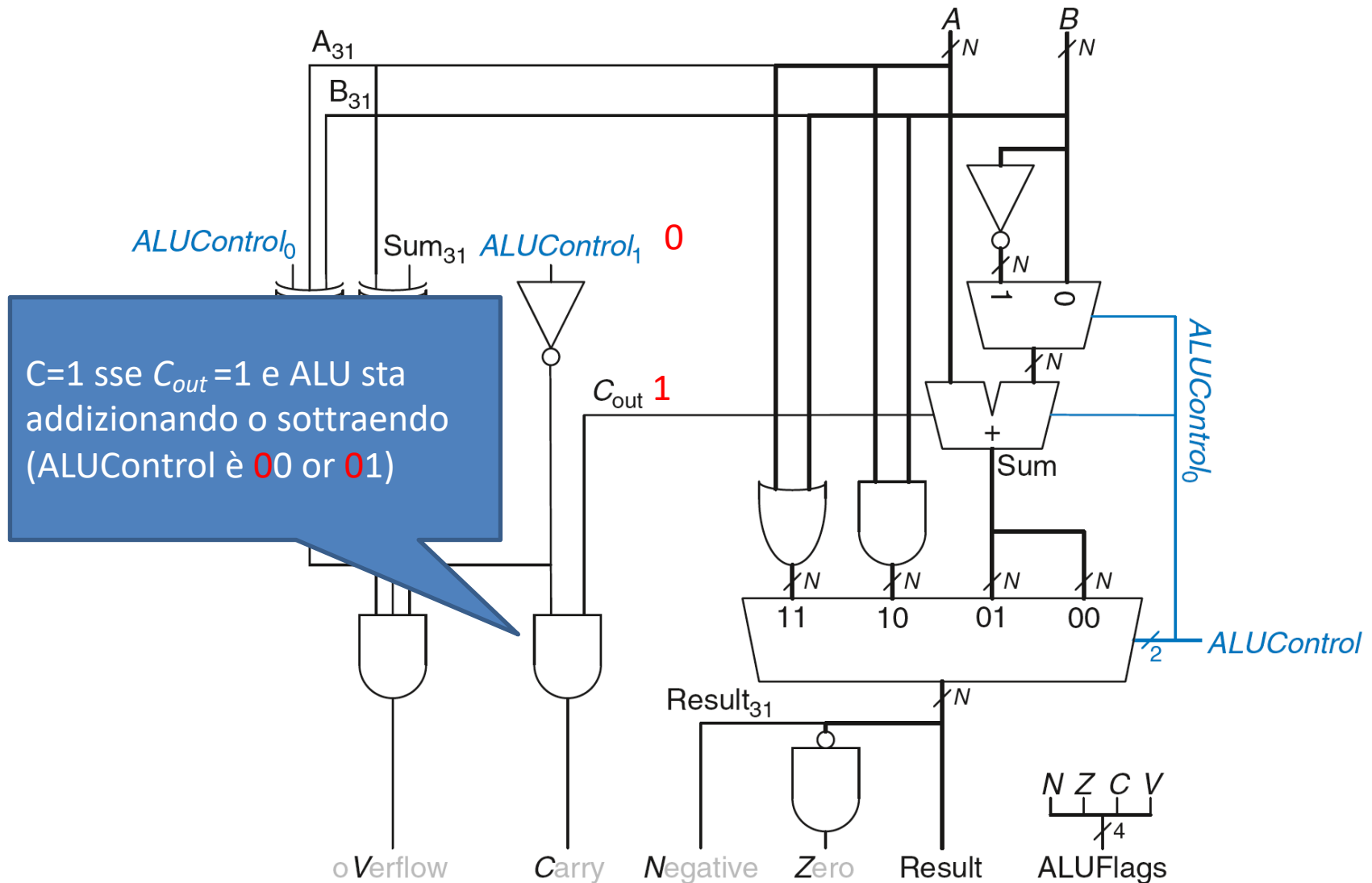
$N=1$ sse Result negativo. Quindi, nella rappresentazione a complemento a 2, N risulta essere il bit più significativo



ALU con flags di stato: Zero

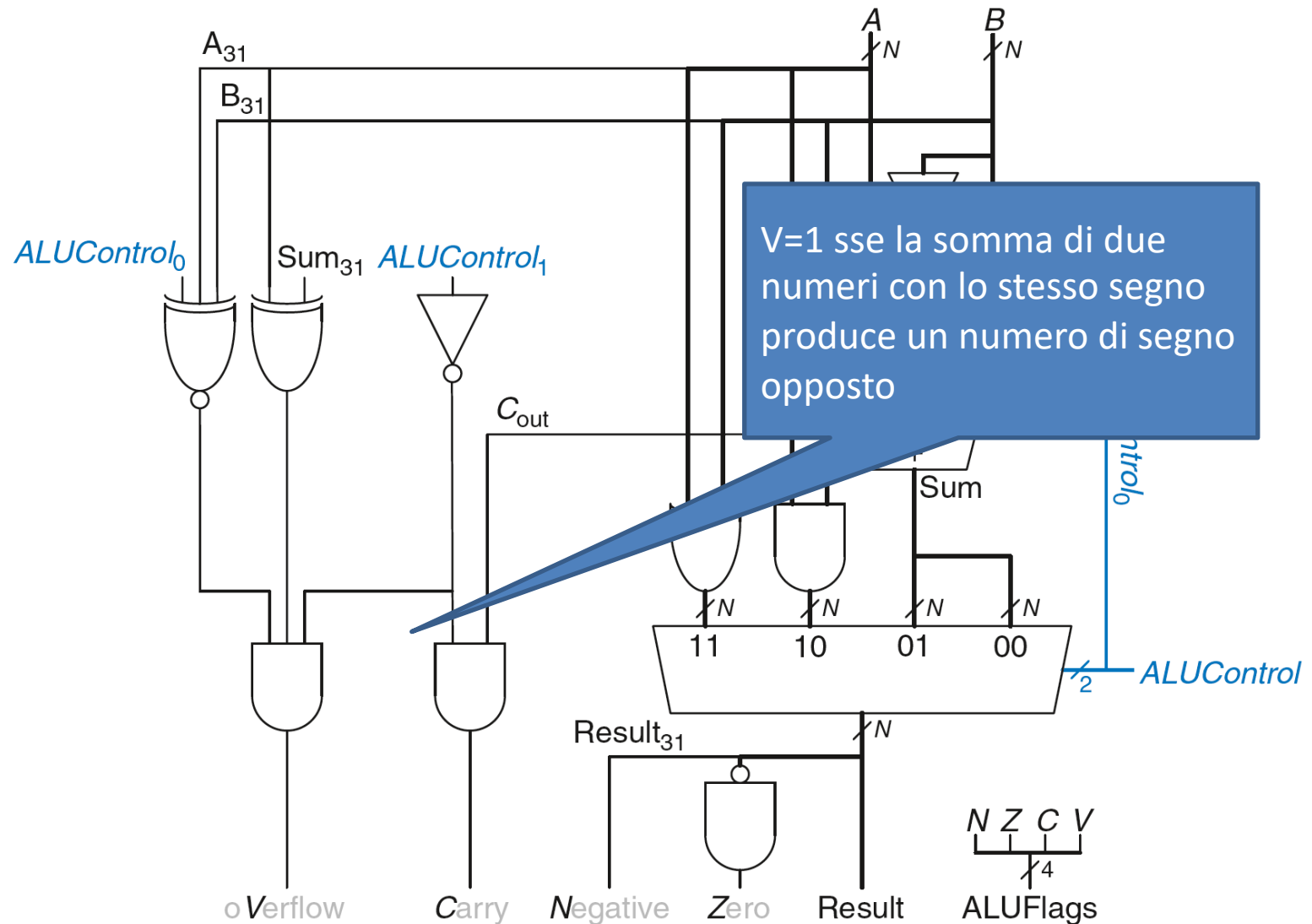


ALU con flags di stato: Carry

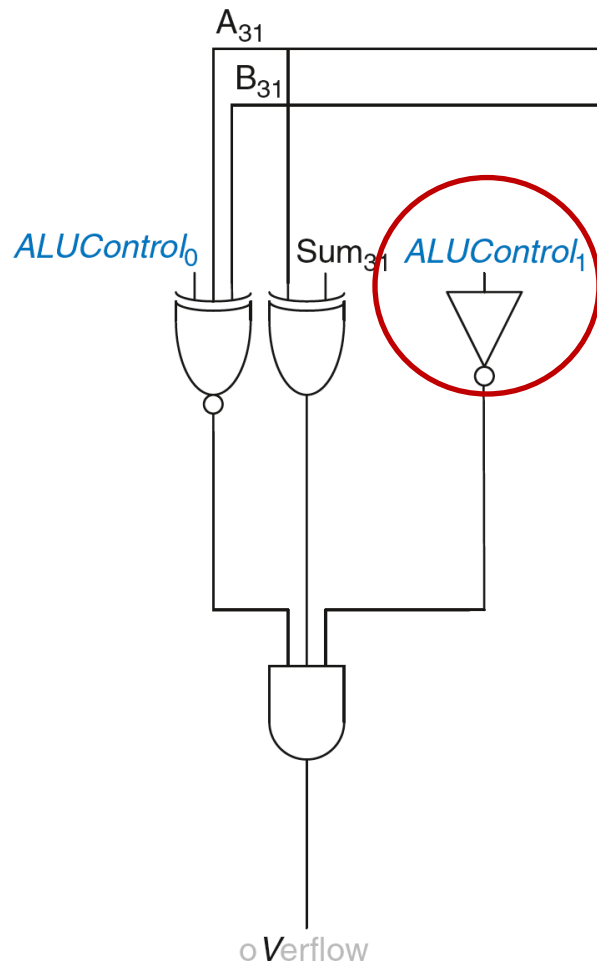


C=1 sse $C_{out}=1$ e ALU sta
addizionando o sottraendo
(ALUControl è 00 or 01)

ALU con flags di stato: Overflow



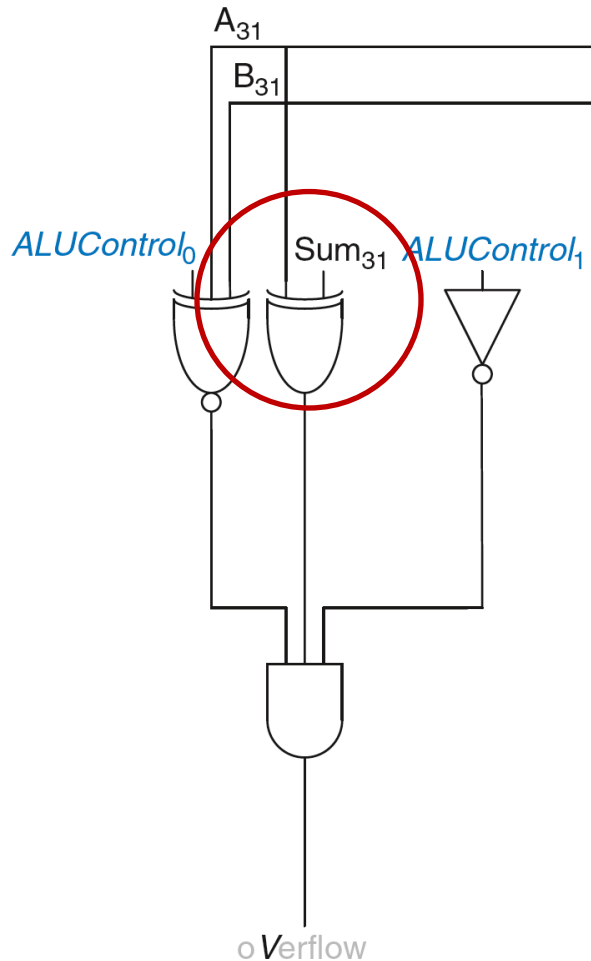
ALU con flags di stato: Overflow



V = 1 sse:

ALU esegue una addizione o sottrazione
($ALUControl_1 = 0$)

ALU con flags di stato: Overflow



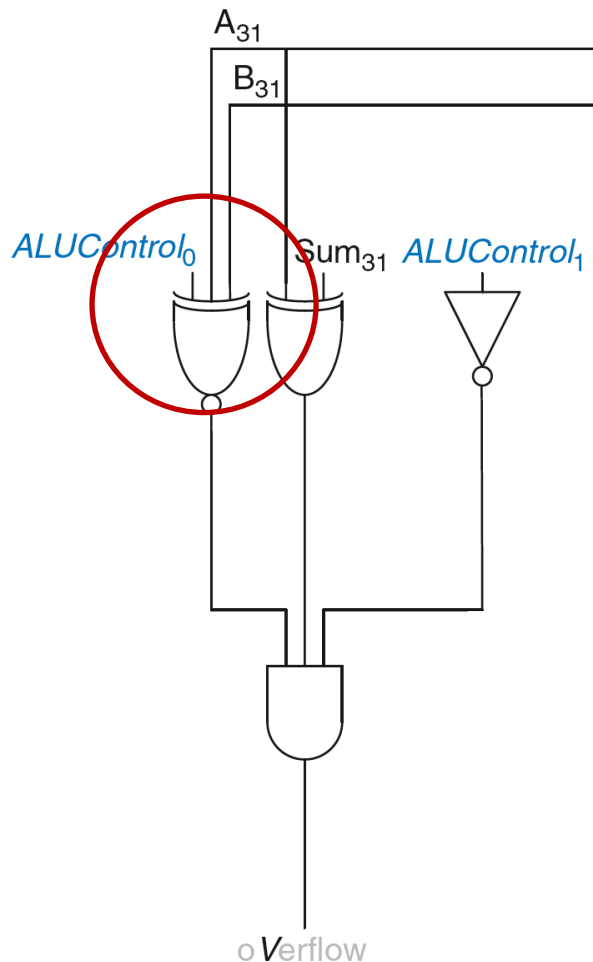
$V = 1$ sse:

ALU esegue una addizione o sottrazione
($ALUControl_1 = 0$)

AND

A e Sum hanno segno opposto

ALU con flags di stato: Overflow



$V = 1$ sse:

ALU esegue una addizione o sottrazione
($ALUControl_1 = 0$)

AND

A e Sum hanno segno opposto

AND

A e B hanno lo stesso segno sotto addizione
($ALUControl_0 = 0$) **OR**

A e B hanno segni differenti sotto sottrazione
($ALUControl_0 = 1$)

ALU con flags di stato

