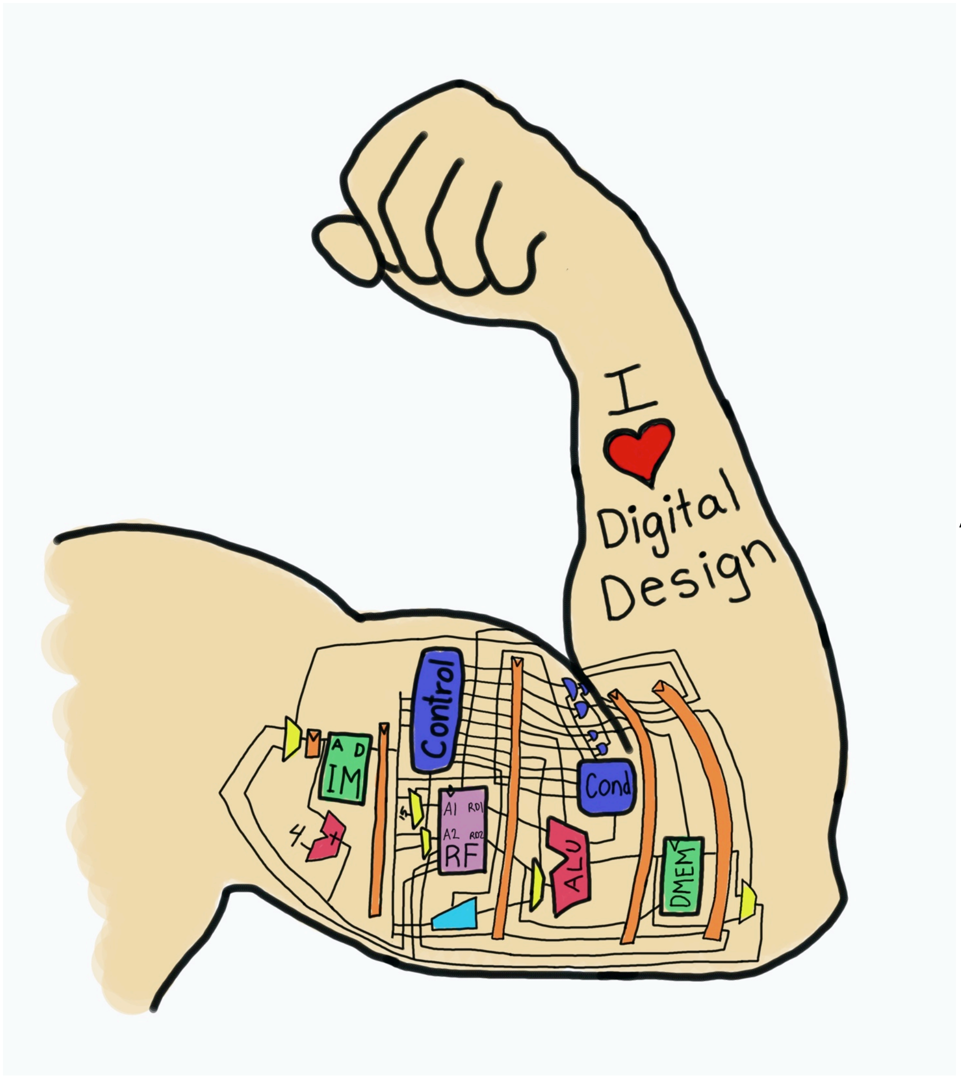
E85 Digital Design & Computer Engineering

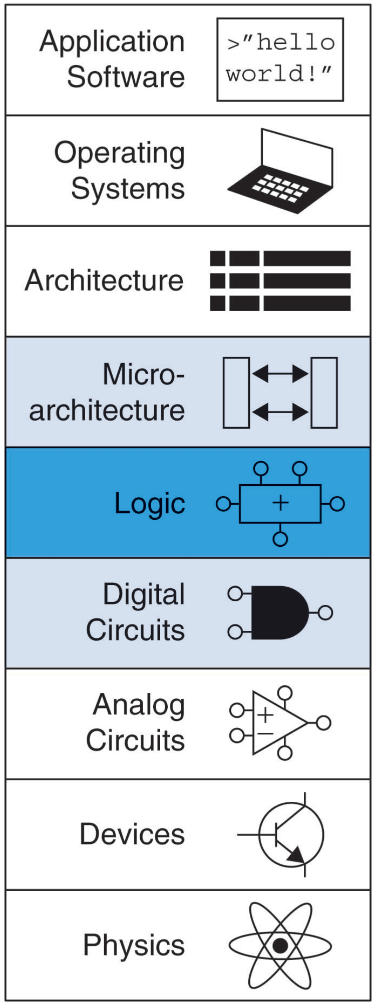


Lecture 8:
Arithmetic Circuits

HARVEY MUDD COLLEGE

Lecture 8

- **Chapter 5 Introduction**
- **Arithmetic Circuits**
  - **1-bit Adders**
  - **N-bit Adders**
    - **Ripple Adders**
    - **Carry Lookahead Adders**
    - **Prefix Adders**
  - **Subtractors**
  - **Arithmetic/Logic Units**

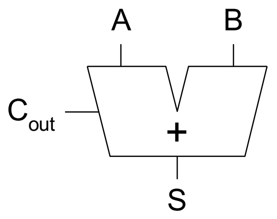


Chapter 5 Introduction

- **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

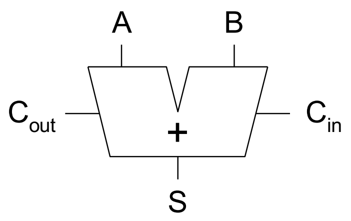


A	B	C_{out}	S
0	0		
0	1		
1	0		
1	1		

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		

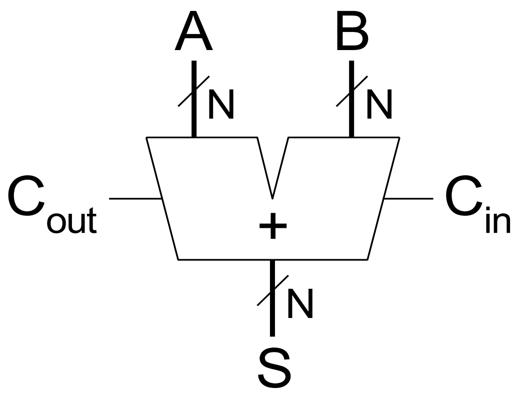
S =

C_{out} =

Multibit Adders (CPAs)

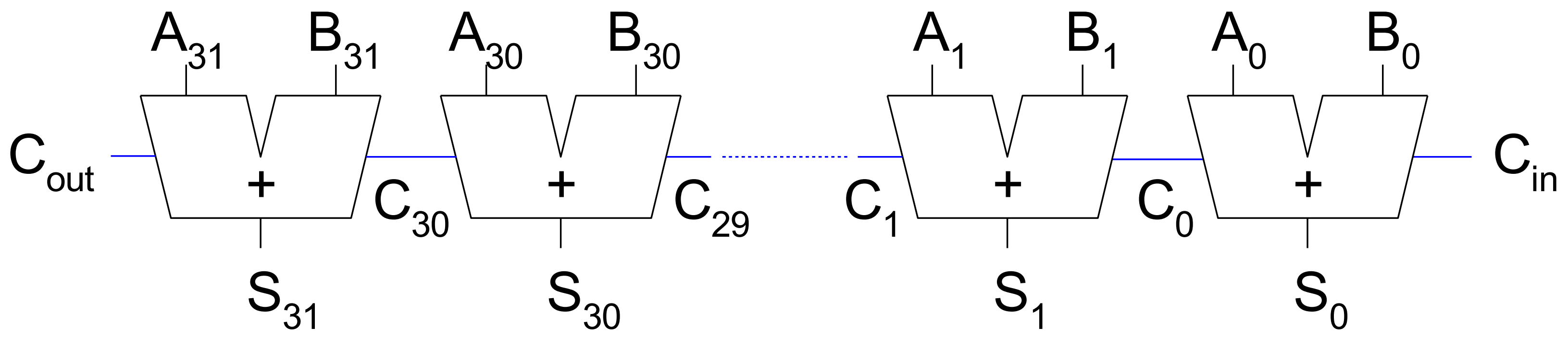
- Types of carry propagate adders (CPAs):
  - Ripple-carry (slow)
  - Carry-lookahead (fast)
  - Prefix (faster)
- Carry-lookahead and prefix adders faster for large adders but require more hardware

Symbol



Ripple-Carry Adder

- Chain 1-bit adders together
- Carry ripples through entire chain
- Disadvantage: **slow**



Ripple-Carry Adder Delay

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

where t_{FA} is the delay of a 1-bit full adder

Carry-Lookahead Adder

Compute C_{out} for *k*-bit blocks using *generate* and *propagate* signals

Some definitions:

- Column *i* produces a carry out by either ***generating*** a carry out or ***propagating*** a carry in to the carry out
- Generate (G_i) and propagate (P_i) signals for each column:
  - **Generate:** Column *i* will generate a carry out if A_i **and** B_i are both 1.

$$G_i =$$

  - **Propagate:** Column *i* will propagate a carry in to the carry out if A_i **or** B_i is 1.

$$P_i =$$

  - **Carry out:** The carry out of column *i* (C_i) is:

$$C_i =$$

Block Propagate and Generate

Now use column Propagate and Generate signals to compute ***Block Propagate*** and ***Generate*** signals for k-bit blocks, i.e.:

- Compute if a k-bit group will **propagate** a carry in (to the block) to the carry out (of the block)
- Compute if a k-bit group will **generate** a carry out (of the block)

Block Propagate and Generate Signals

- **Example:** Block propagate and generate signals for 4-bit blocks ($P_{3:0}$ and $G_{3:0}$):

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

$$\begin{aligned} G_{3:0} &= G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 \\ &= G_3 + P_3 \left(G_2 + P_2 \left(G_1 + P_1 G_0\right)\right) \end{aligned}$$

Block Propagate and Generate Signals

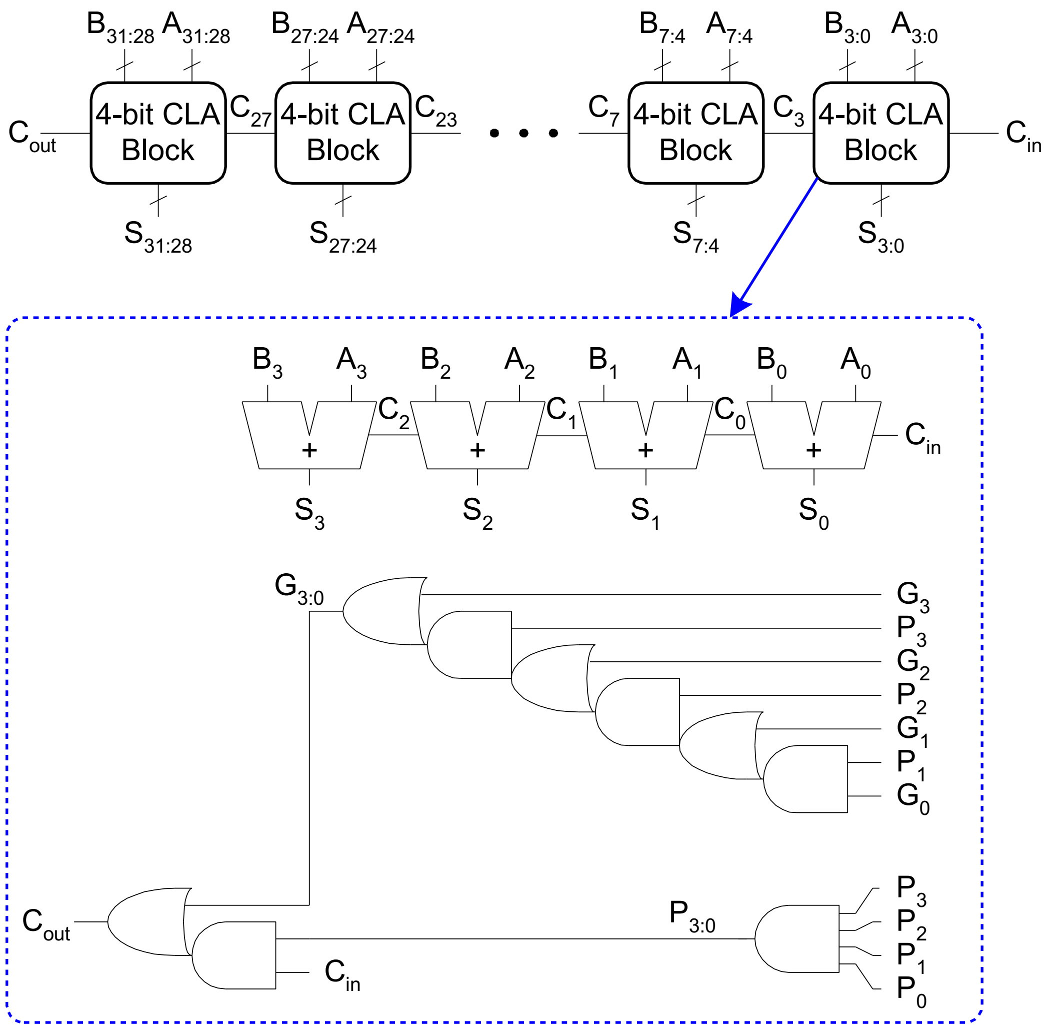
- **In general for a block spanning bits i through j,**

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

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

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

32-bit CLA with 4-bit Blocks



Carry-Lookahead Addition

- **Step 1:** Compute G_i and P_i for all columns
- **Step 2:** Compute G and P for k-bit blocks
- **Step 3:** C_{in} propagates through each k-bit propagate/generate logic (meanwhile computing sums)
- **Step 4:** Compute sum for most significant k-bit block

Carry-Lookahead Addition

- **Step 1:** Compute G_i and P_i for all columns

$$G_i = A_i B_i$$

$$P_i = A_i + B_i$$

Carry-Lookahead Addition

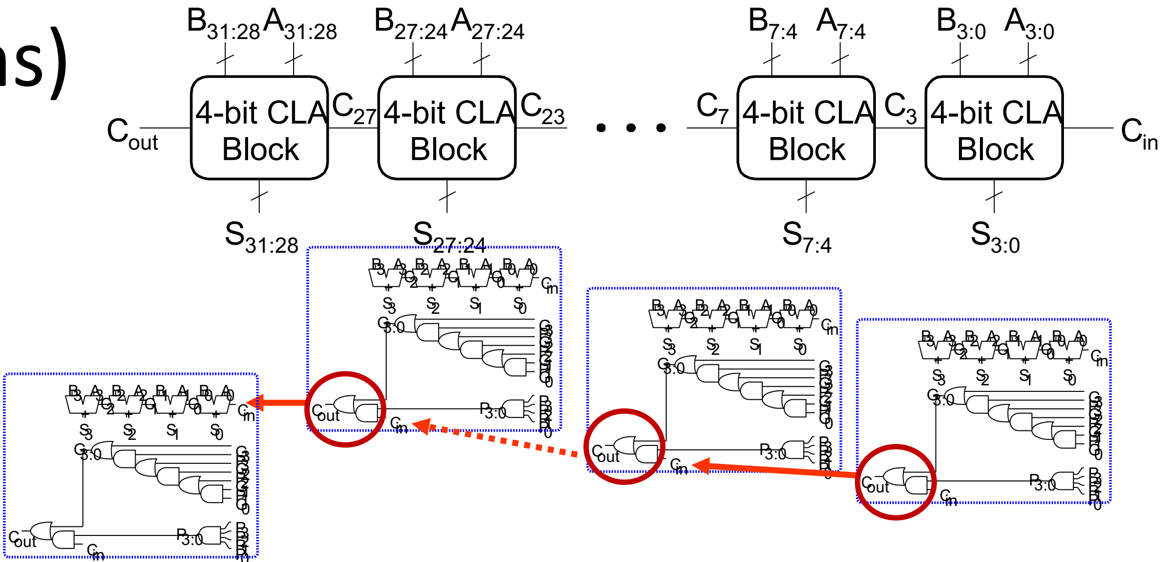
- **Step 1:** Compute G_i and P_i for all columns
- **Step 2:** Compute G and P for k-bit blocks

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

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

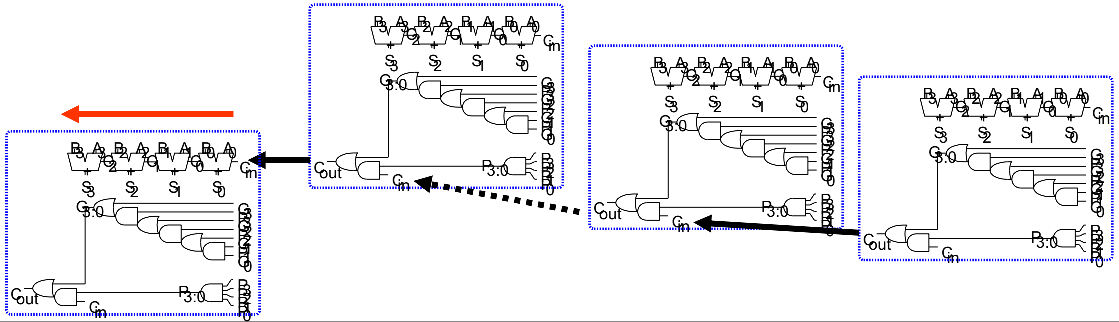
Carry-Lookahead Addition

- **Step 1:** Compute G_i and P_i for all columns
- **Step 2:** Compute *G* and *P* for *k*-bit blocks
- **Step 3:** C_{in} propagates through each *k*-bit propagate/generate logic (meanwhile computing sums)



Carry-Lookahead Addition

- **Step 1:** Compute G_i and P_i for all columns
- **Step 2:** Compute *G* and *P* for *k*-bit blocks
- **Step 3:** C_{in} propagates through each *k*-bit propagate/generate logic (meanwhile computing sums)
- **Step 4:** Compute sum for most significant k-bit block



Carry-Lookahead Adder Delay

For *N*-bit CLA with *k*-bit blocks:

$$t_{CLA} = t_{pg} + t_{pg\_block} + (N/k - 1)t_{\text{AND\_OR}} + kt_{FA}$$

- t_{pg} : delay to generate all P_i, G_i
- t_{pg_block} : delay to generate all $P_{i:j}$, $G_{i:j}$
- $t_{\text{AND\_OR}}$: delay from C_{in} to C_{out} of final AND/OR gate in *k*-bit CLA block

An *N*-bit carry-lookahead adder is generally much faster than a ripple-carry adder for *N* > 16

Prefix Adder

- Computes carry in (C_{i-1}) for each column, then computes sum:

$$S_i = (A_i \wedge B_i) \wedge C_{i-1}$$

- Computes G and P for 1-, 2-, 4-, 8-bit blocks, etc. until all G_i (carry in) known
- $\log_2 N$ stages

Prefix Adder

- Carry in either *generated* in a column or *propagated* from a previous column.
- Column -1 holds C_{in}, so

  $$G_{-1} = C_{in}$$
- Carry in to column *i* = carry out of column *i-1*:

  $$C_{i-1} = G_{i-1:-1}$$

  $G_{i-1:-1}$: generate signal spanning columns *i*-1 to -1
- Sum equation:

  $$S_i = (A_i \wedge B_i) \wedge G_{i-1:-1}$$
- **Goal:** Quickly compute $G_{0:-1}, G_{1:-1}, G_{2:-1}, G_{3:-1}, G_{4:-1}, G_{5:-1}, \ldots$ (called ***prefixes***) (= $C_0, C_1, C_2, C_3, C_4, C_5, \ldots$)

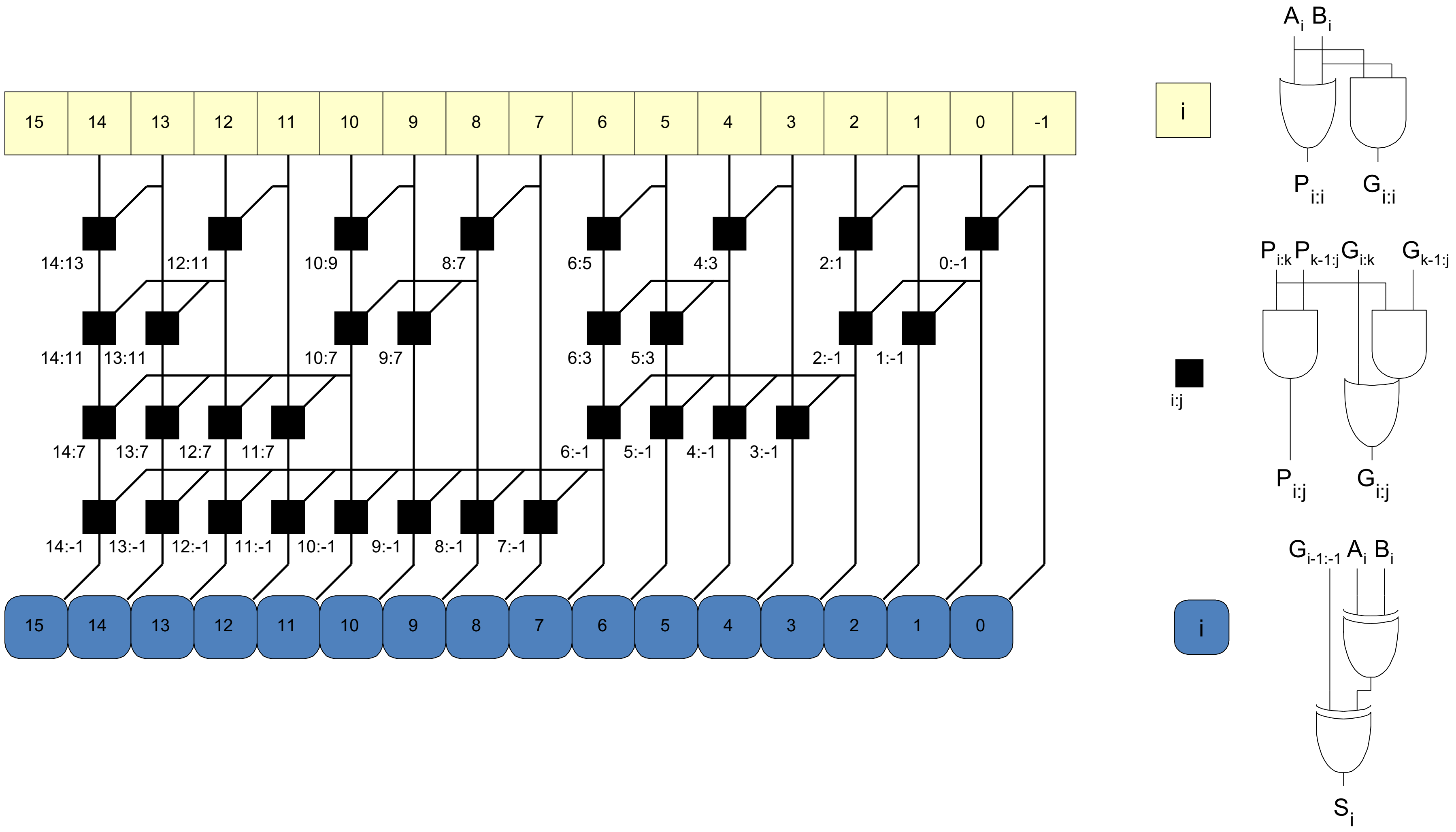
Prefix Adder

- Generate and propagate signals for a block spanning bits $i{:}j$

$$G_{i:j} = G_{i:k} + P_{i:k}\, G_{k-1:j}$$
$$P_{i:j} = P_{i:k} P_{k-1:j}$$

- In words:
  - **Generate:** block $i{:}j$ will generate a carry if:
    - upper part $(i{:}k)$ generates a carry or
    - upper part $(i{:}k)$ propagates a carry generated in lower part $(k\text{-}1{:}j)$
  - **Propagate:** block $i{:}j$ will propagate a carry if *both* the upper and lower parts propagate the carry

16-Bit Prefix Adder Schematic



Prefix Adder Delay

$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_prefix}) + t_{XOR}$$

t_{pg}: delay to produce P_i, G_i (AND or OR gate)

t_{pg_prefix}: delay of black prefix cell (AND-OR gate)

Adder Delay Comparisons

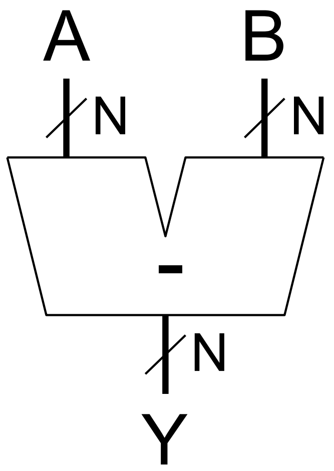
Compare delay of: 32-bit ripple-carry, CLA, and prefix adders

- CLA has 4-bit blocks
- 2-input gate delay = 10 ps; full adder delay = 30 ps

$$\begin{aligned}
t_{\text{ripple}} &= Nt_{FA} = 32(30\text{ ps}) \\
&= \mathbf{960\text{ ps}} \\
t_{CLA} &= t_{pg} + t_{pg\_\text{block}} + (N/k - 1)t_{\text{AND\_OR}} + kt_{FA} \\
&= [10 + 60 + (7)20 + 4(30)]\text{ ps} \\
&= \mathbf{330\text{ ps}} \\
t_{PA} &= t_{pg} + \log_2 N(t_{pg\_\text{prefix}}) + t_{\text{XOR}} \\
&= [10 + \log_2 32(20) + 10]\text{ ps} \\
&= \mathbf{120\text{ ps}}
\end{aligned}$$

Subtracter

Symbol



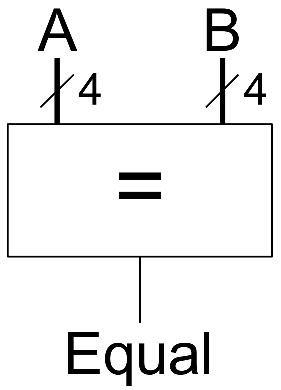
Implementation

A B

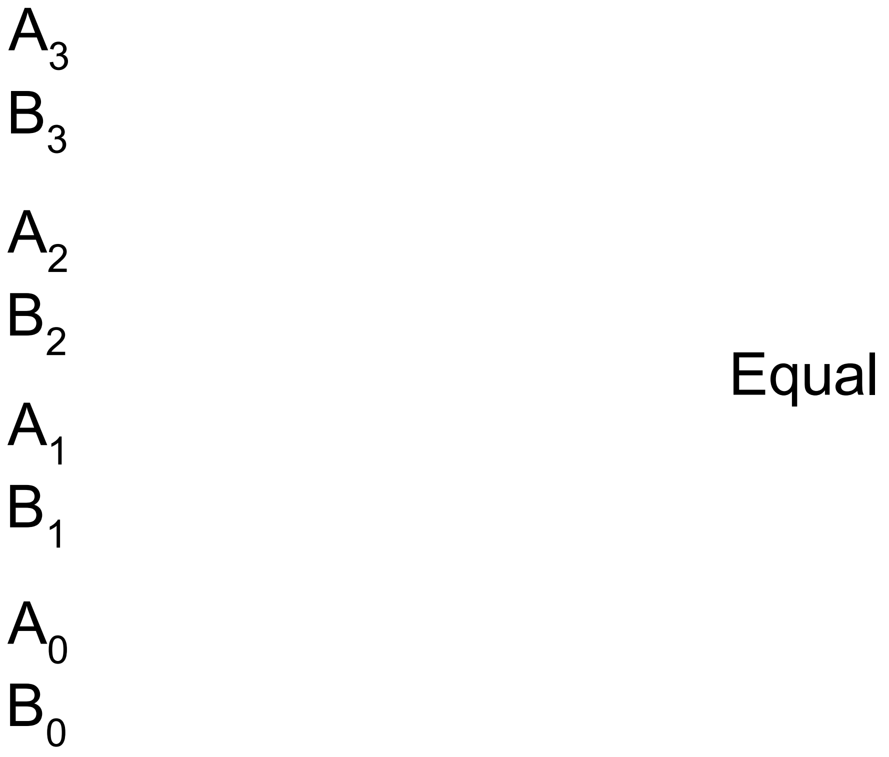
Y

Comparator: Equality

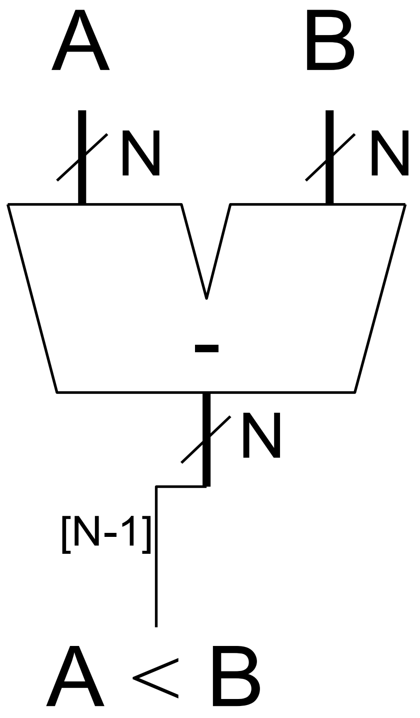
Symbol



Implementation



Comparator: Less Than



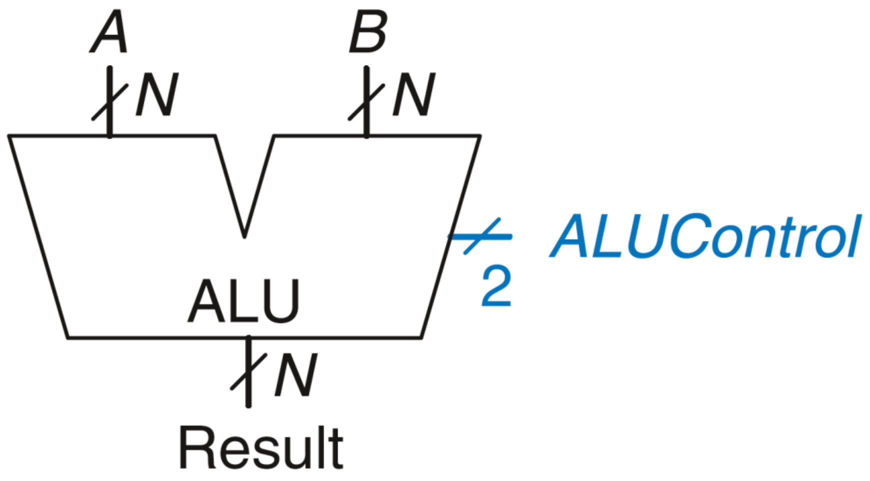
ALU: Arithmetic Logic Unit

ALU should perform:

- Addition
- Subtraction
- AND
- OR

ALU: Arithmetic Logic Unit

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



Example: Perform *A* + *B*

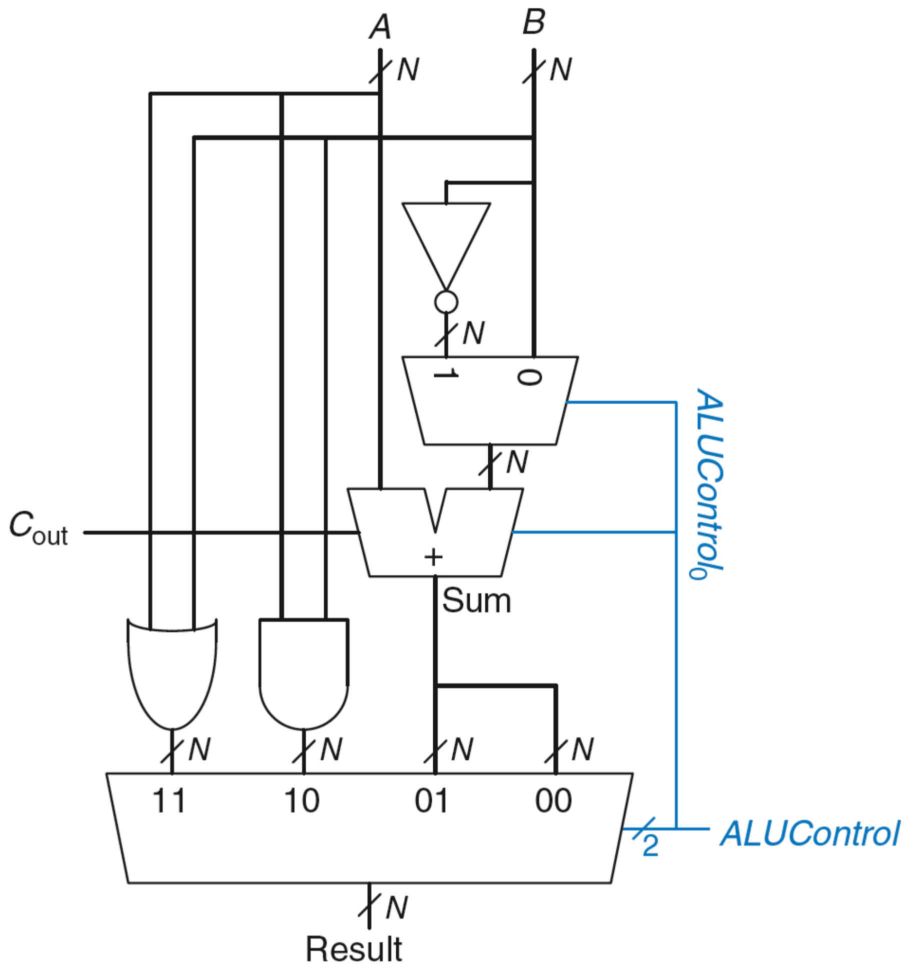
$ALUControl = 00$

$Result = A + B$

ALU: Arithmetic Logic Unit

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

Example: Perform *A* OR *B*



ALU: Arithmetic Logic Unit

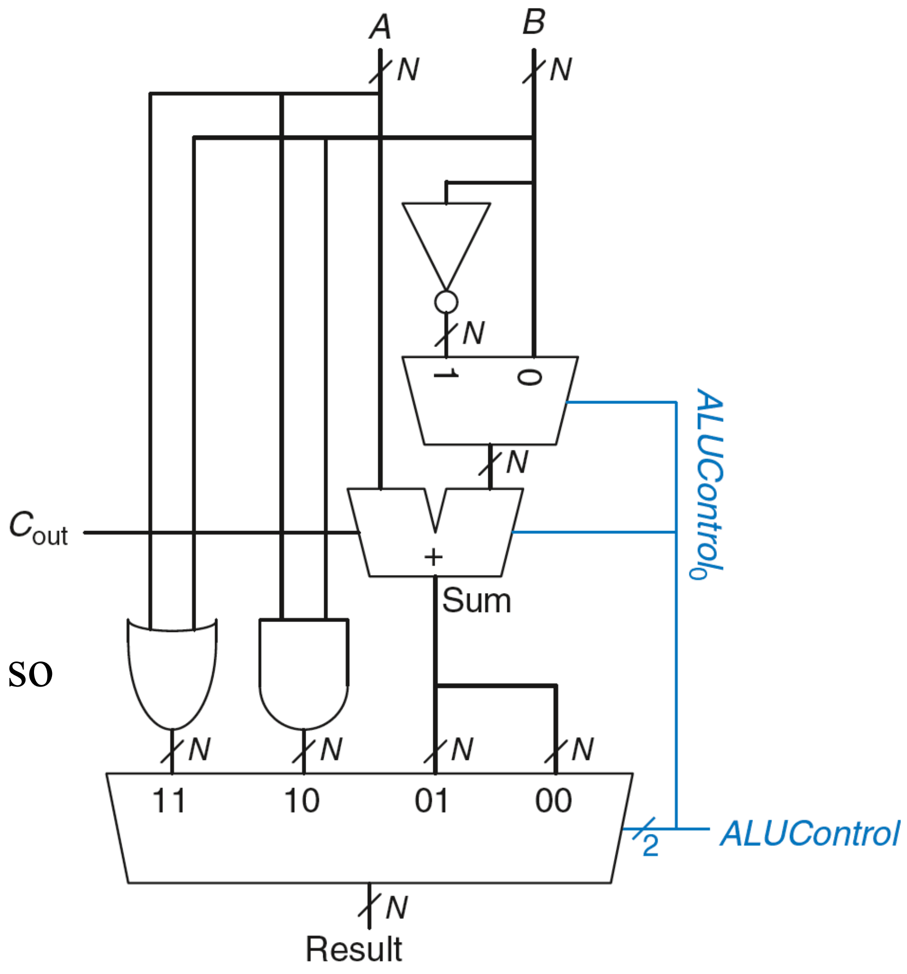
$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

Example: Perform *A* OR *B*

$ALUControl_{1:0}$ = 11

Mux selects output of OR gate as *Result,* so

Result* = *A* OR *B

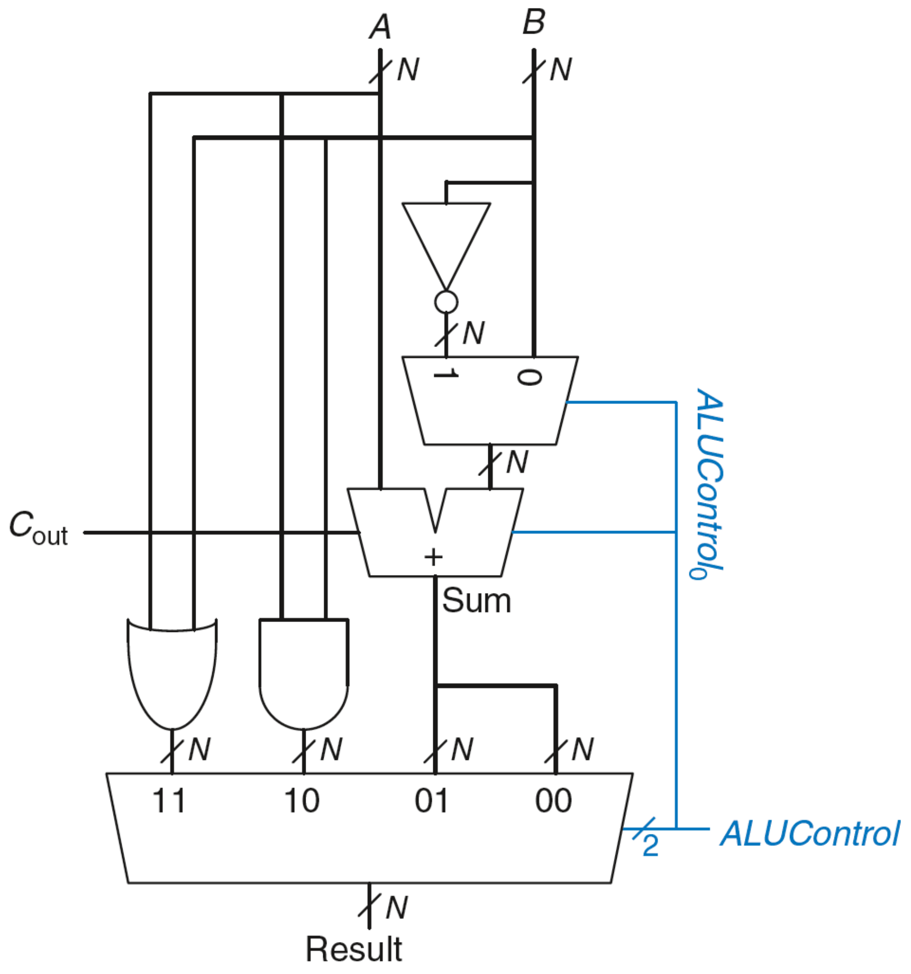


ELSEVIER

ALU: Arithmetic Logic Unit

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

Example: Perform *A* + *B*



ALU: Arithmetic Logic Unit

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

Example: Perform *A* + *B*

$ALUControl_{1:0}$ = 00

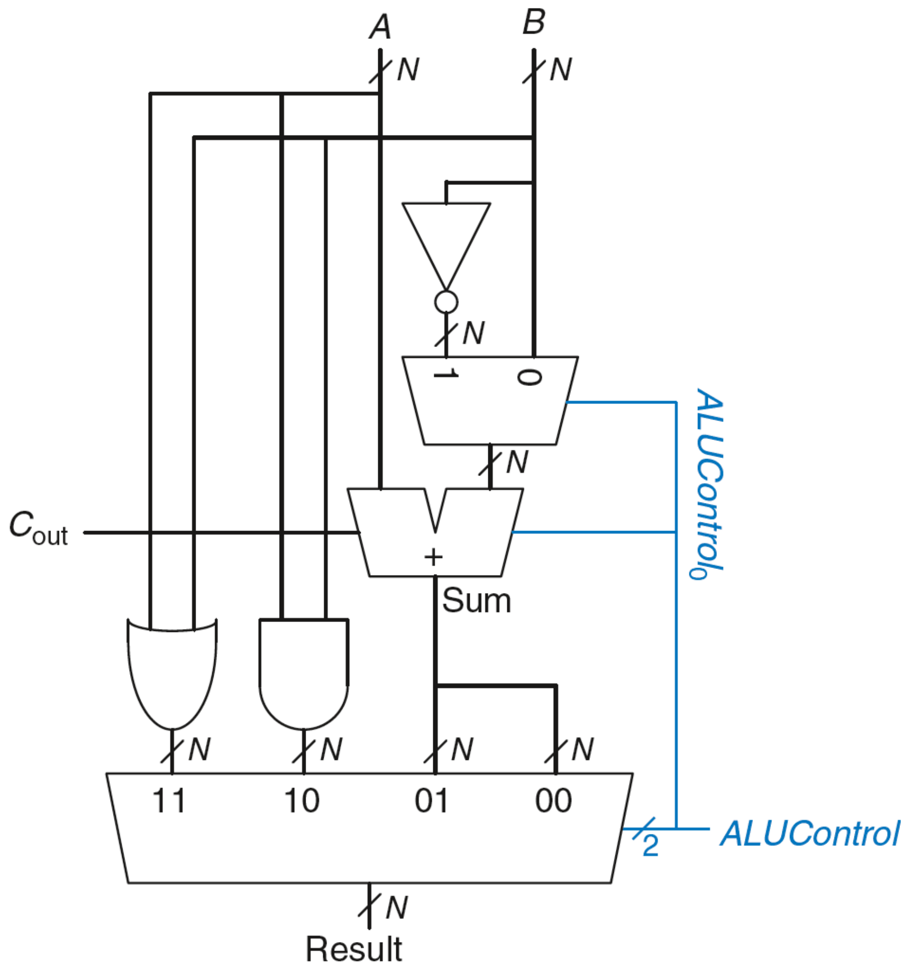
$ALUControl_0$ = 0, so:

  C_{in} to adder = 0

  2nd input to adder is *B*

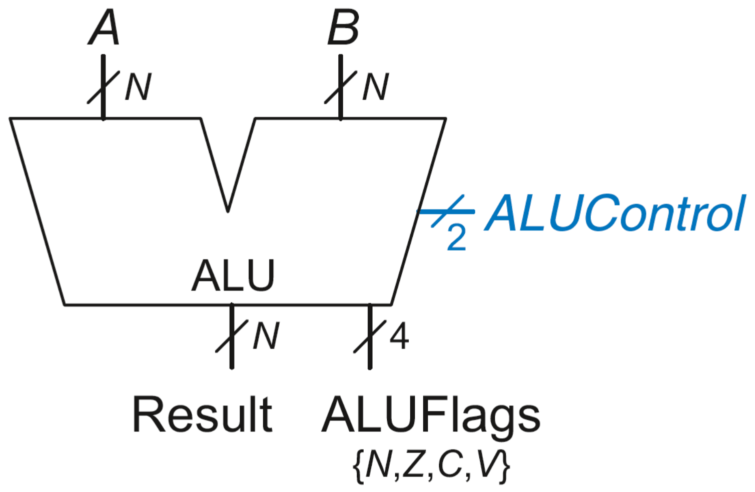
Mux selects *Sum* as *Result,* so

  Result* = *A* + *B

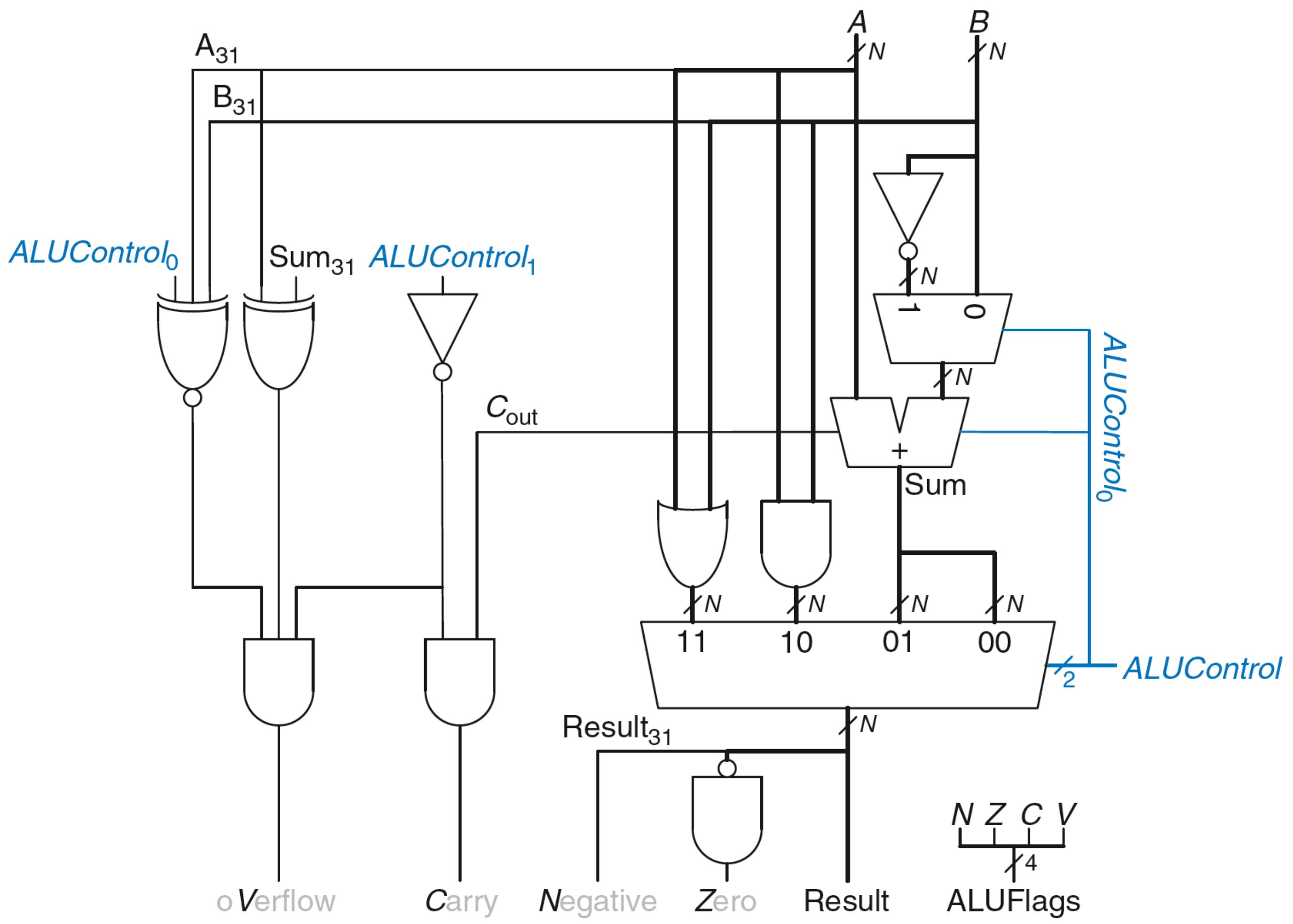


ALU with Status Flags

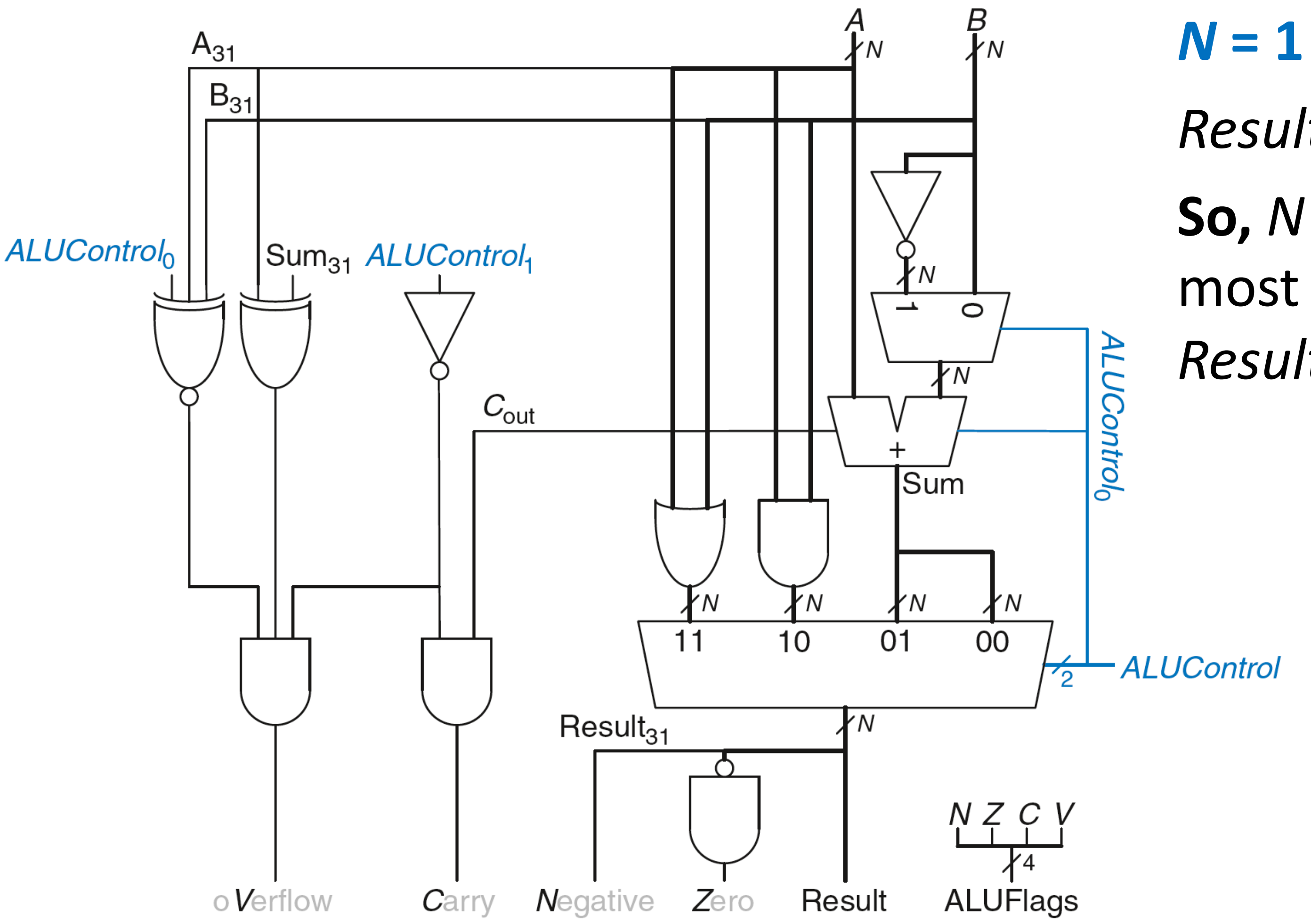
Flag	Description
N	*Result* is **N**egative
Z	*Result* is **Z**ero
C	Adder produces **C**arry out
V	Adder o**V**erflowed



ALU with Status Flags



ALU with Status Flags: Negative

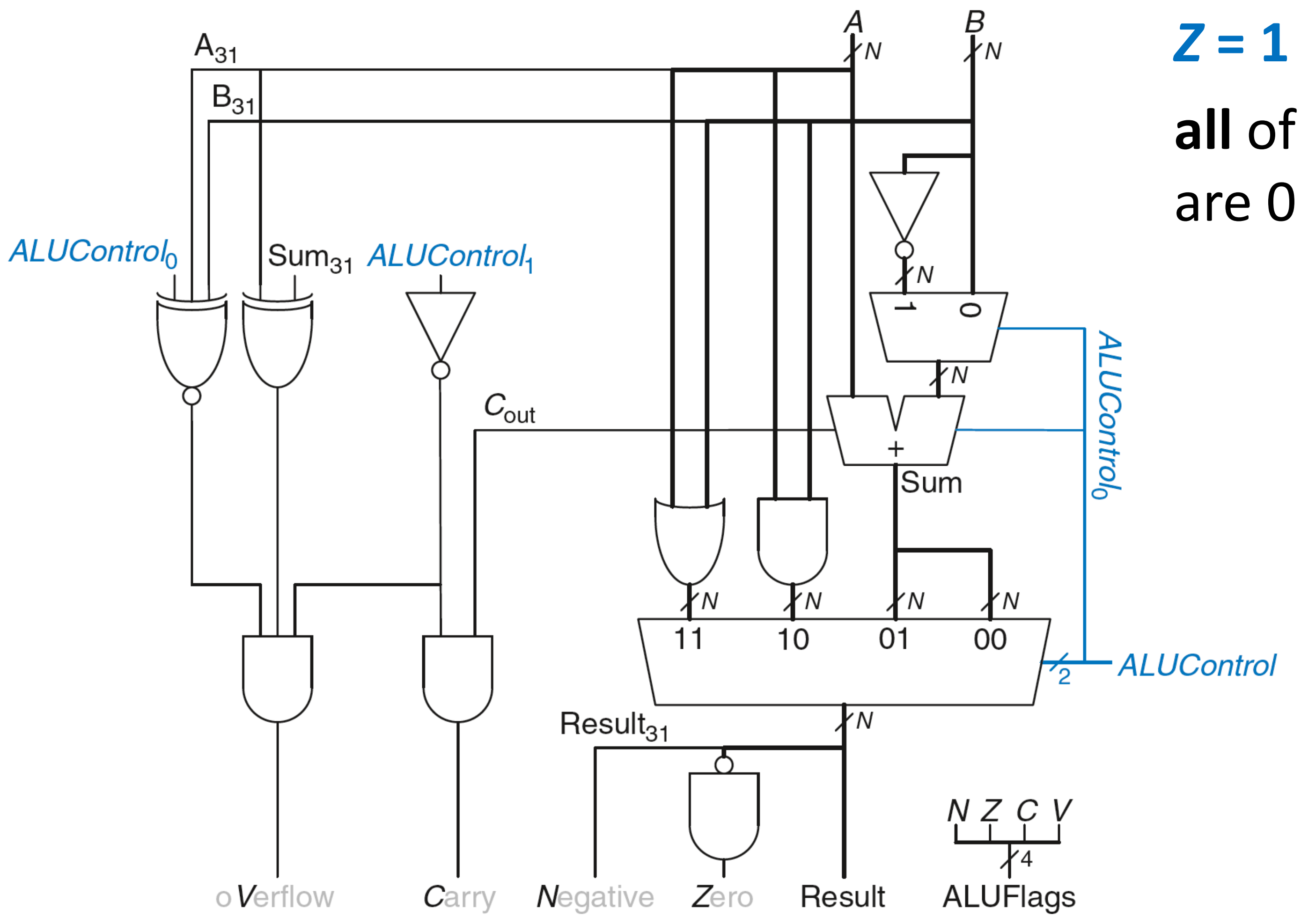


***N* = 1** if:

Result is **negative**

So, *N* is connected to most significant bit of *Result*

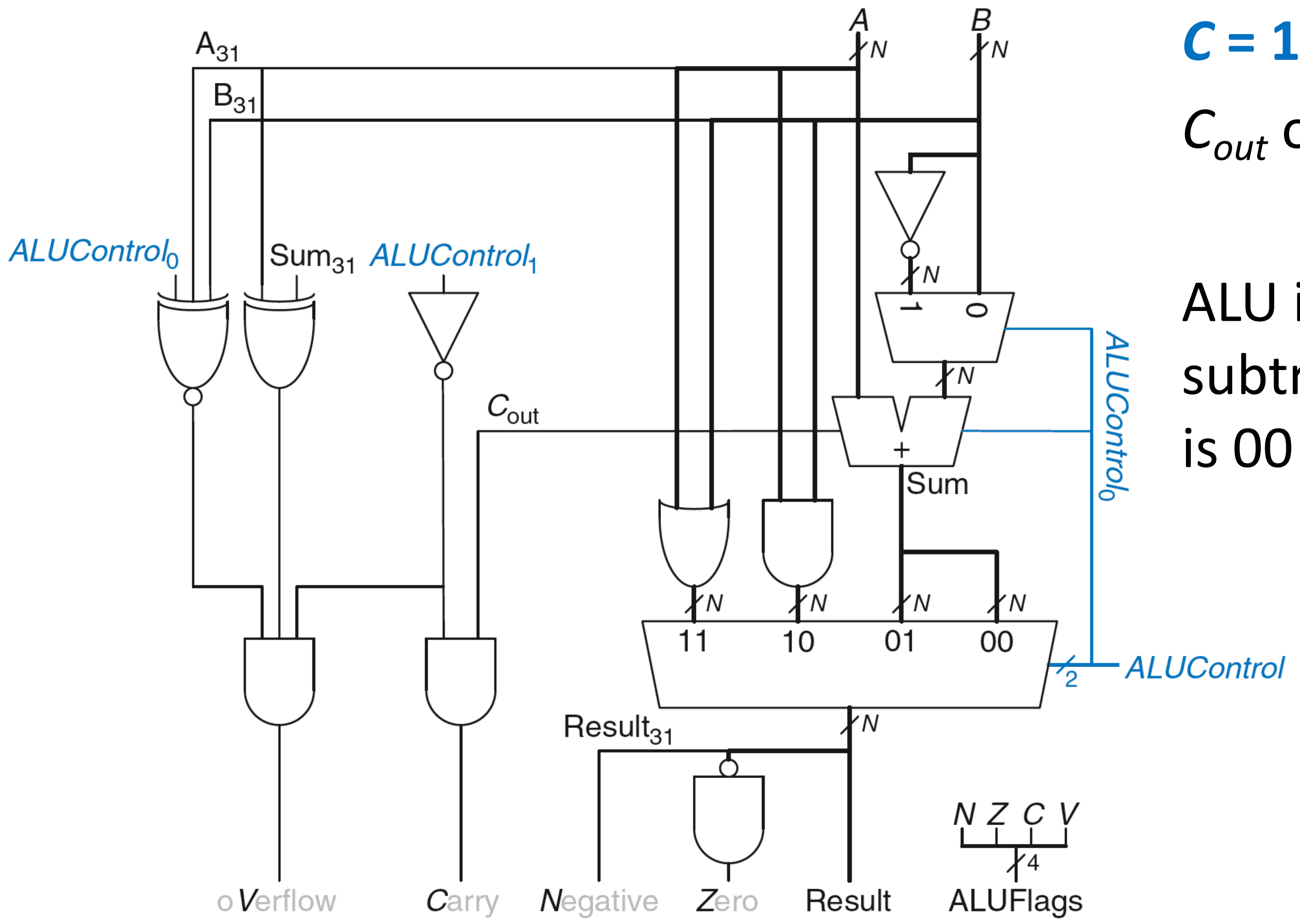
ALU with Status Flags: Zero



***Z* = 1** if:

all of the bits of *Result* are 0

ALU with Status Flags: Carry



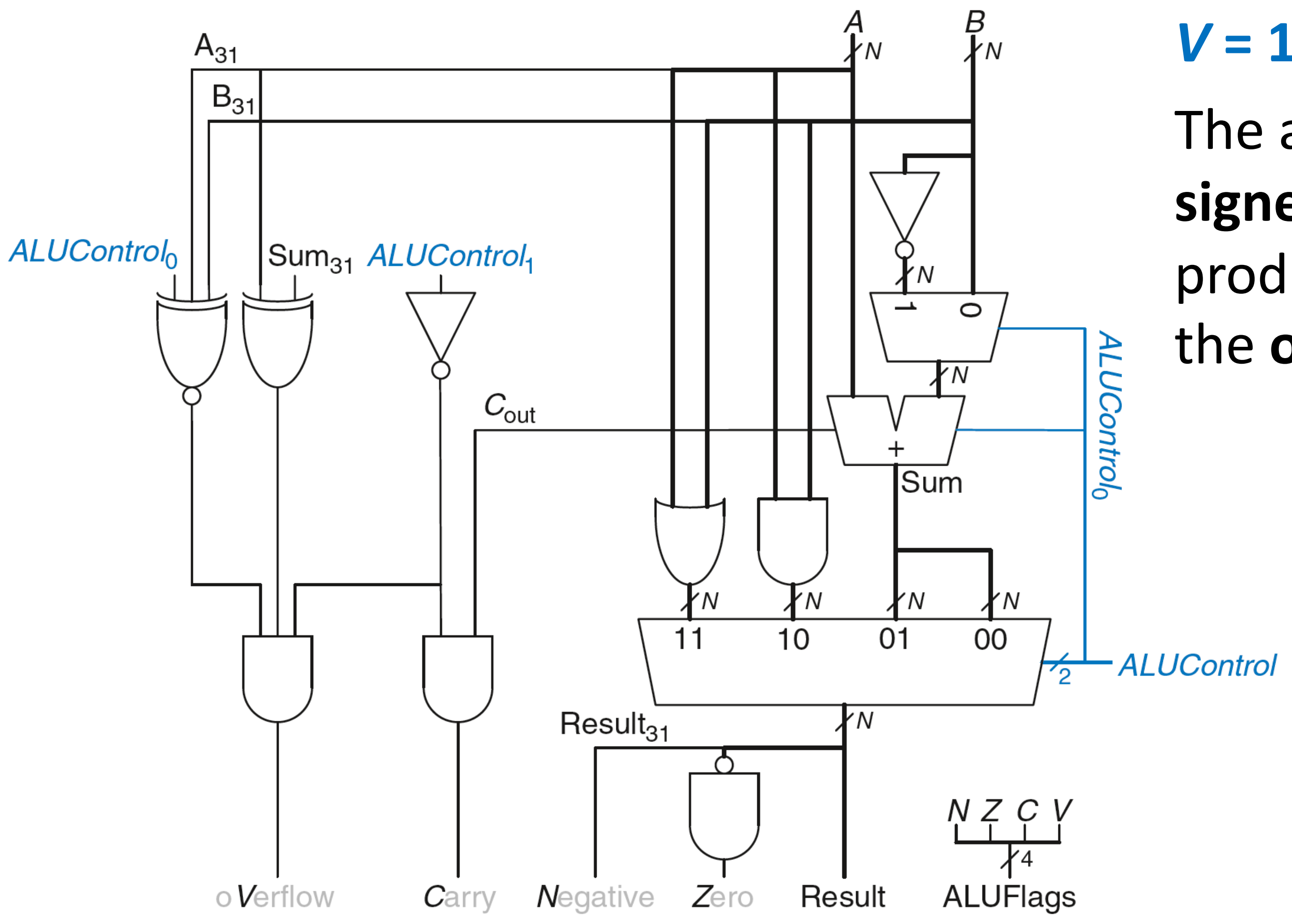
C = 1 if:

C_{out} of Adder is 1

AND

ALU is adding or subtracting (ALUControl is 00 or 01)

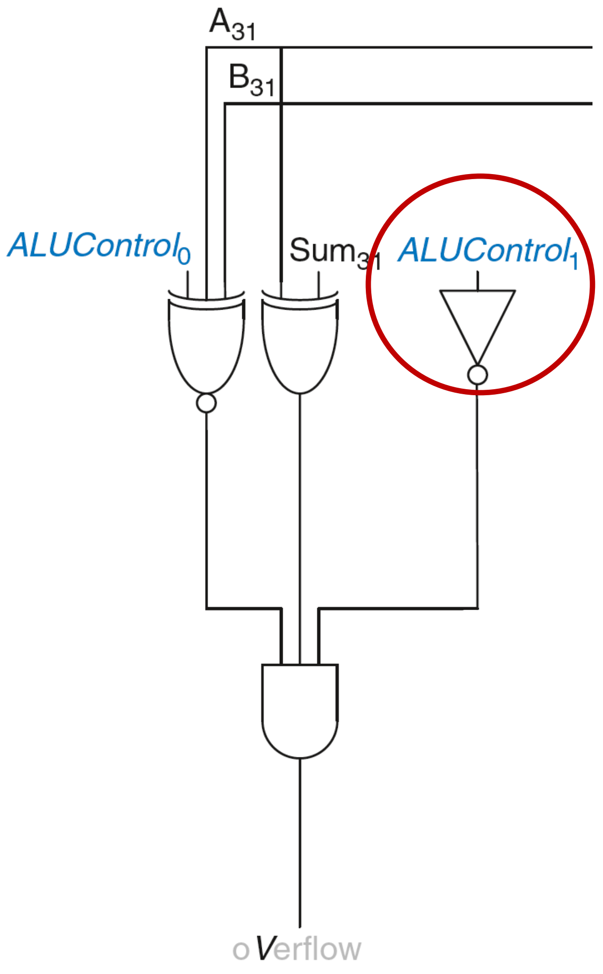
ALU with Status Flags: oVerflow



***V* = 1** if:

The addition of 2 **same-signed numbers** produces a result with the **opposite sign**

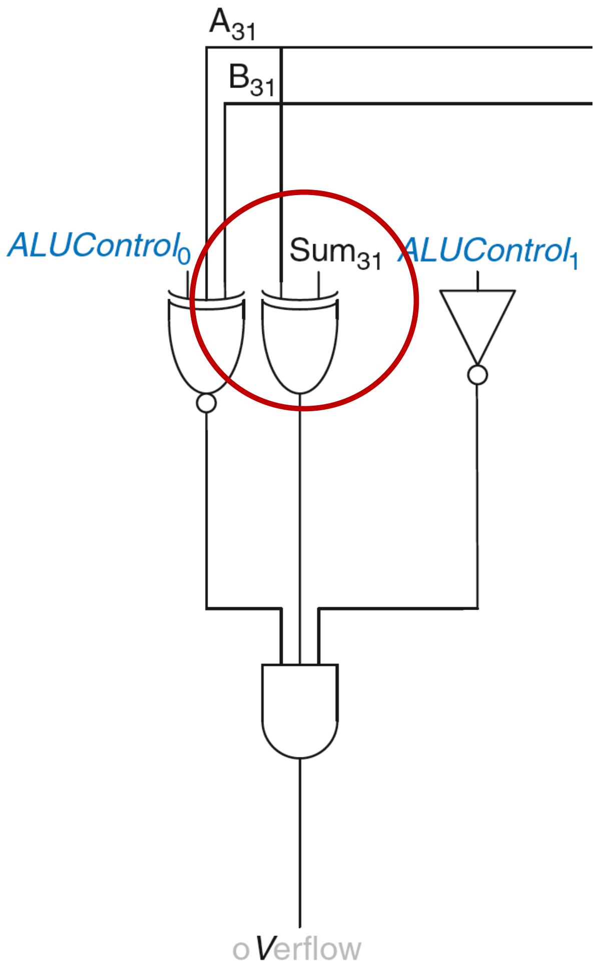
ALU with Status Flags: oVerflow



***V* = 1** if:

ALU is performing addition or subtraction ($ALUControl_1$ = 0)

ALU with Status Flags: oVerflow



***V* = 1** if:

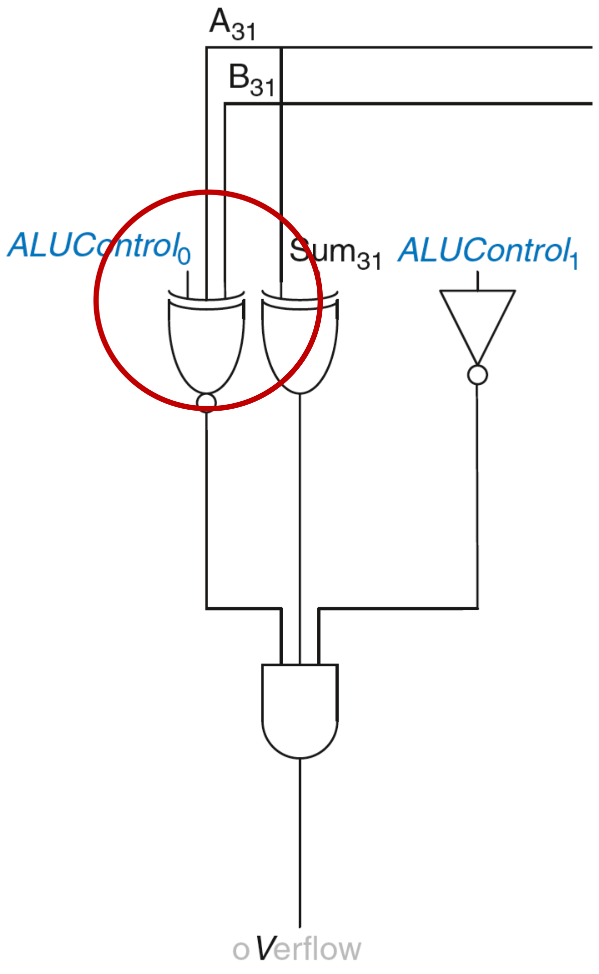
ALU is performing addition or subtraction

($ALUControl_1 = 0$)

AND

A and Sum have opposite signs

ALU with Status Flags: oVerflow



***V* = 1** if:

ALU is performing addition or subtraction

(*ALUControl*$_1$ = 0)

AND

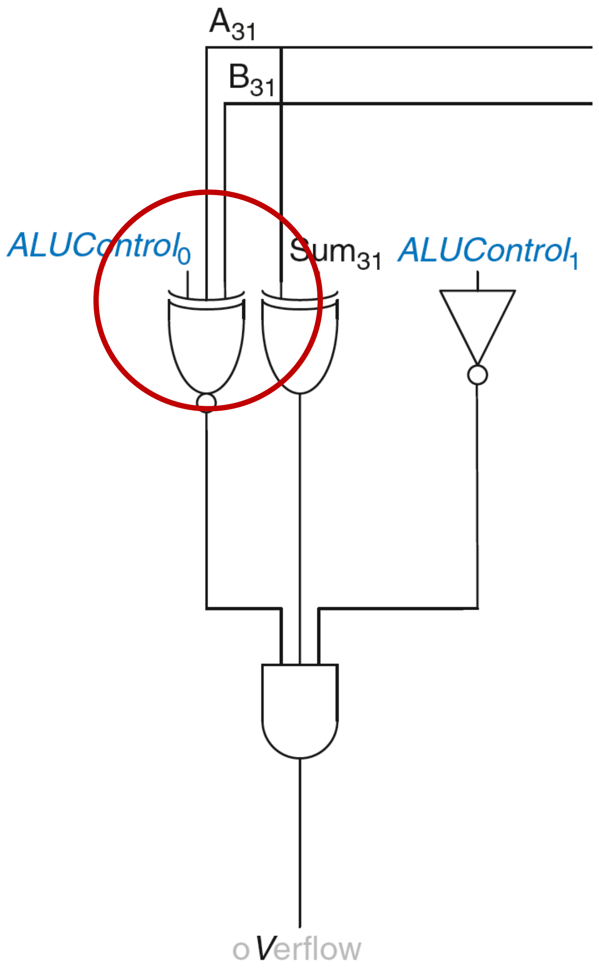
A and Sum have opposite signs

AND

A and B have same signs upon addition **OR**

A and B have different signs upon subtraction

ALU with Status Flags: oVerflow



***V* = 1** if:

ALU is performing addition or subtraction

(*ALUControl*$_1$ = 0)

AND

A and Sum have opposite signs

AND

A and B have same signs upon addition

(*ALUControl*$_0$ = 0) **OR**

A and B have different signs upon subtraction

(*ALUControl*$_0$ = 1)