ECE 2300
Digital Logic & Computer Organization

Fall 2016

More Binary Arithmetic
Arithmetic Logic Unit

Cornell University

Sign Extension

- **Replicate the MSB (sign bit)**

4-bit		8-bit
0100 (4)	→	00000100 (still 4)
1100 (-4)	→	11111100 (still -4)

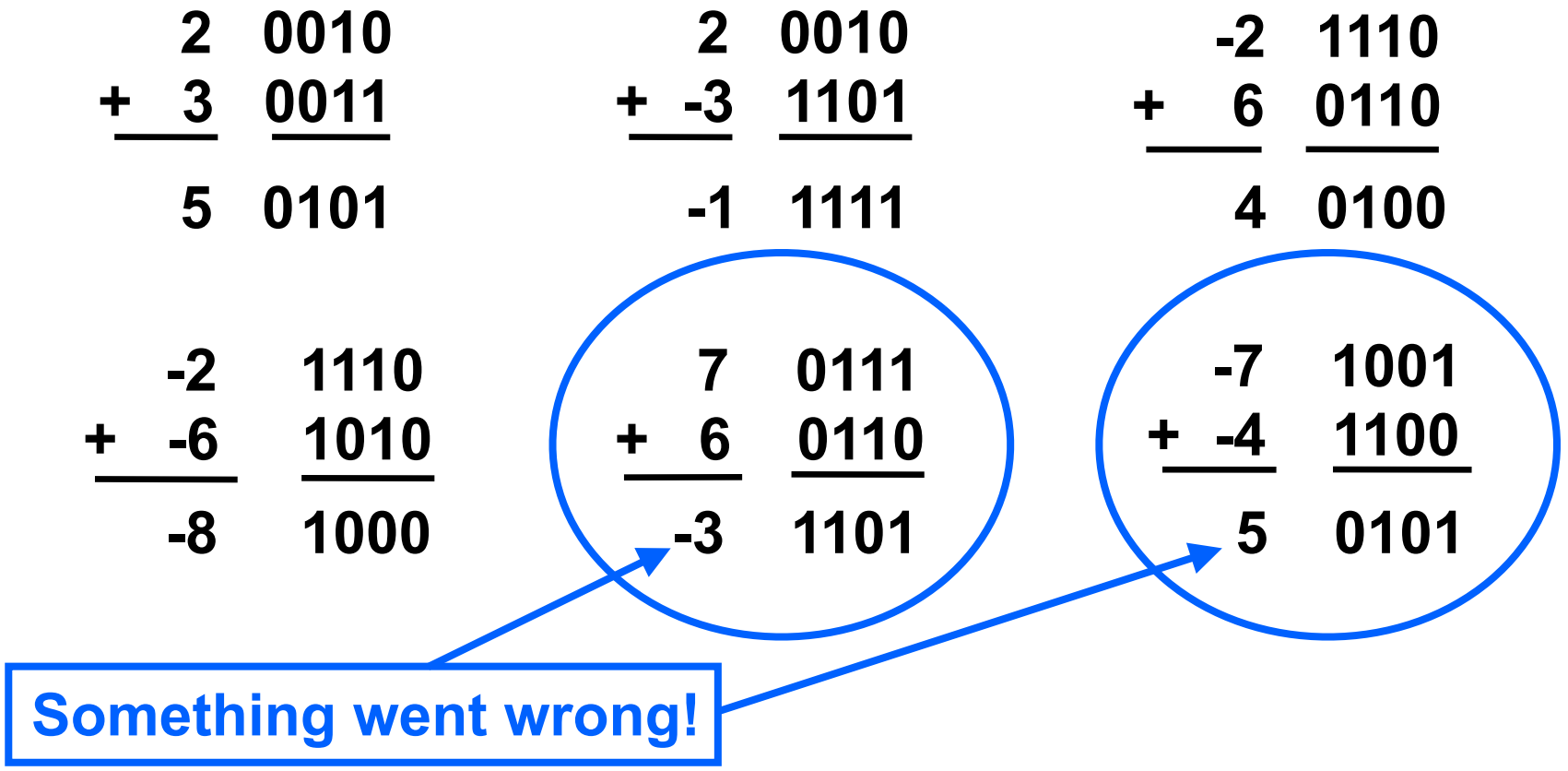
- **Necessary for adding two's complement numbers of different lengths**

Fixed Size Representation

- Computers represent numbers as fixed size n-bit values
- Result of adding two n-bit integers is stored as n bits
- Integers are typically 32 or 64 bits (*words*)
  - 4 or 8 *bytes*
  - *byte* = 8 bits

Fixed Size Addition

- Examples with n = 4



Overflow

- If operands are too big, sum cannot be represented as *n*-bit 2's complement number

01000	(8)	11000	(-8)
+01001	(9)	+10111	(-9)
10001	(-15)	01111	(+15)

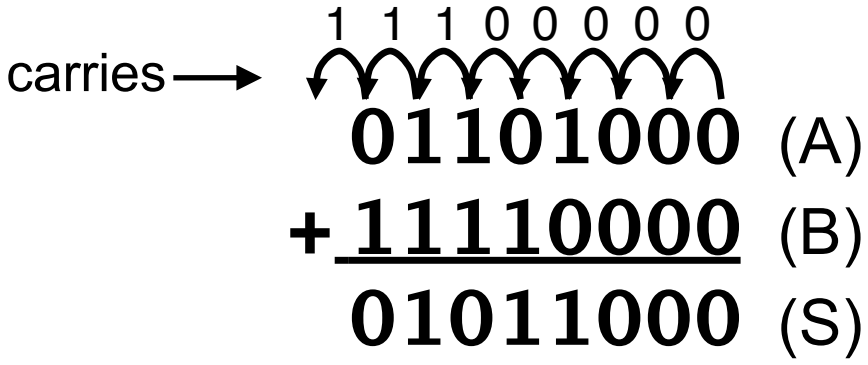
- Overflow occurs if
  - Signs of both operands are the same, and
  - Sign of sum is different
- Another test (easy to do in hardware)
  - Carry into MSB (1 or 0) does not equal carry out (1 or 0)

Did Overflow Occur?

```
  1 1 1   1
  01110010
+ 01010011
----------
  11000101
```

```
  1 1 1 1   1
  11110010
+ 01010011
----------
  01000101
```

Building a Binary Adder

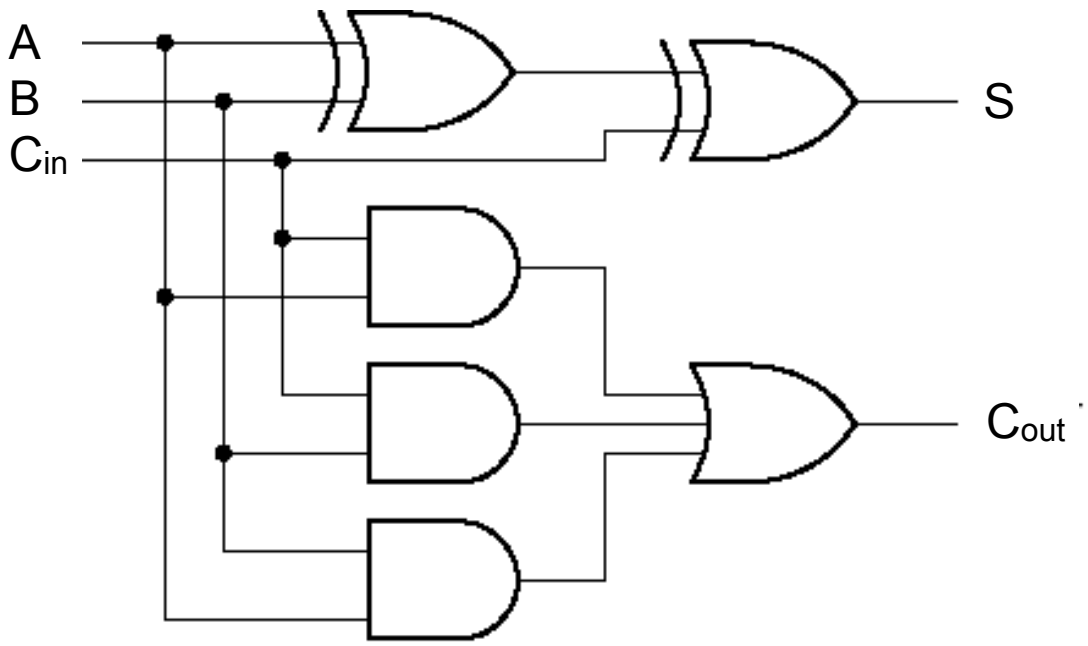


- **Same inputs and outputs for each bit position**
  - **Inputs: A, B and C_{in} (carry-in)**
  - **Outputs: S (sum) and C_{out} (carry-out)**
- **Carry-out of a bit position is the carry-in for next bit position**

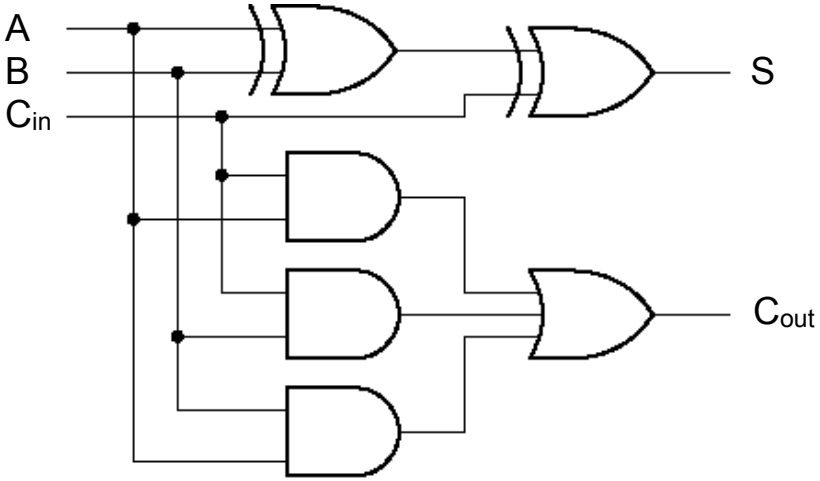
Full Adder (1 bit Adder with Sum and Carry)

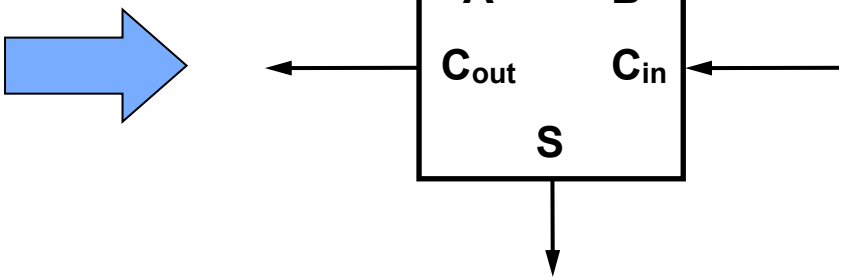
- Inputs: A, B and C_{in} (carry-in)
- Outputs: S (sum) and C_{out} (carry-out)

A	B	C_{in}	S	C_{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

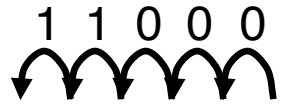


Full-Adder Circuit Symbol

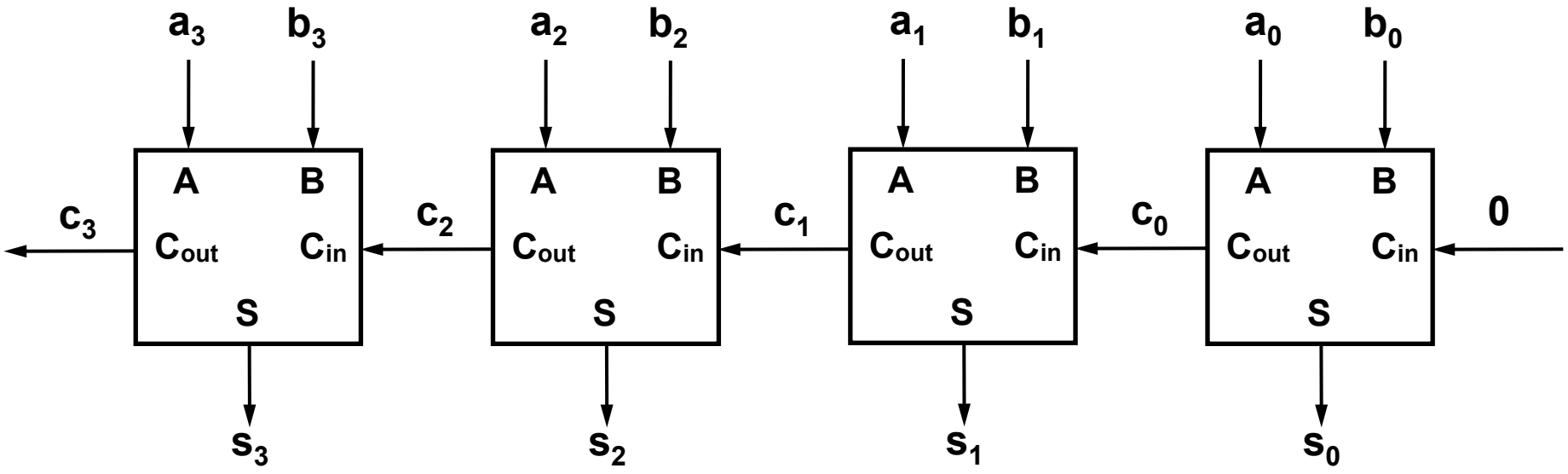




Four Bit Adder



$$\begin{array}{r} 1100 \\ +\underline{0110} \\ 0010 \end{array}$$



Ripple Carry Adder

Two's Complement Subtraction

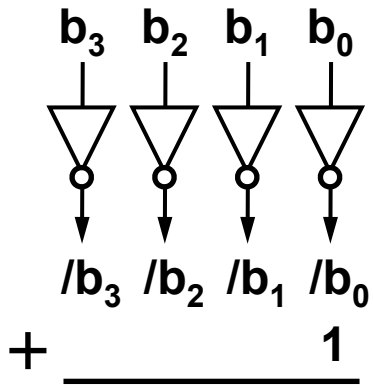
- **Negate second operand and add**

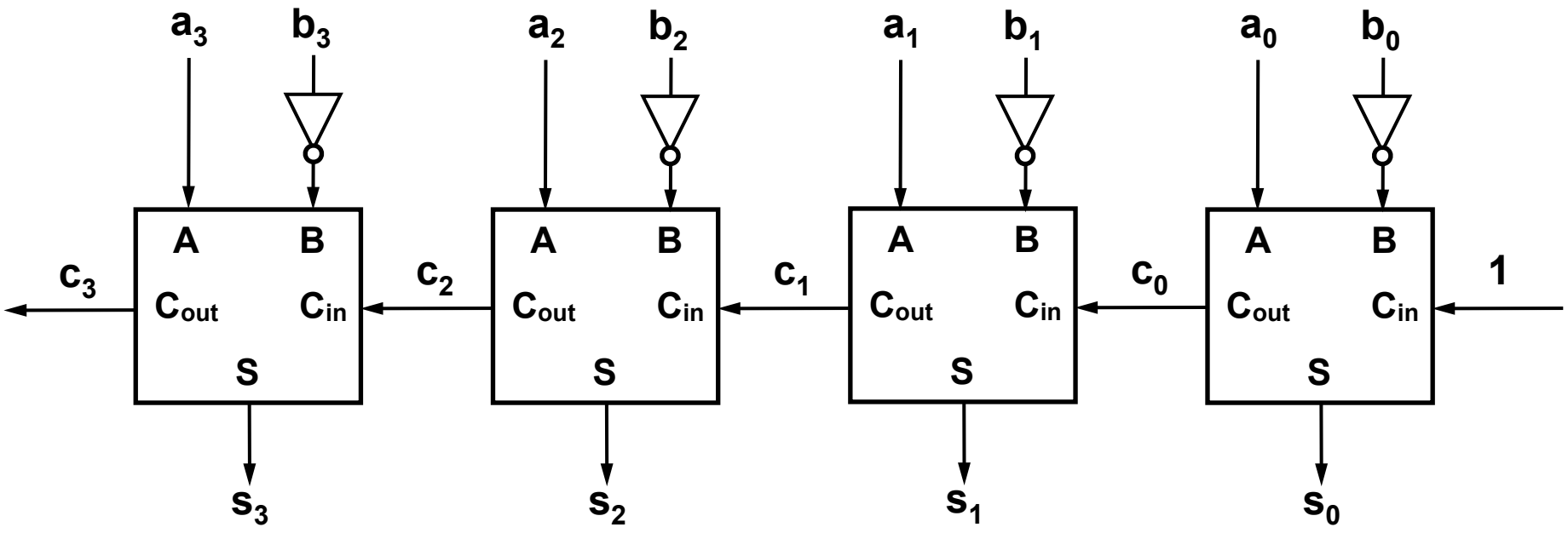
```
  01101000  (104)
- 00010001  (17)
```

```
  01101000  (104)
+ 11101111  (-17)
  01010111  (87)
```

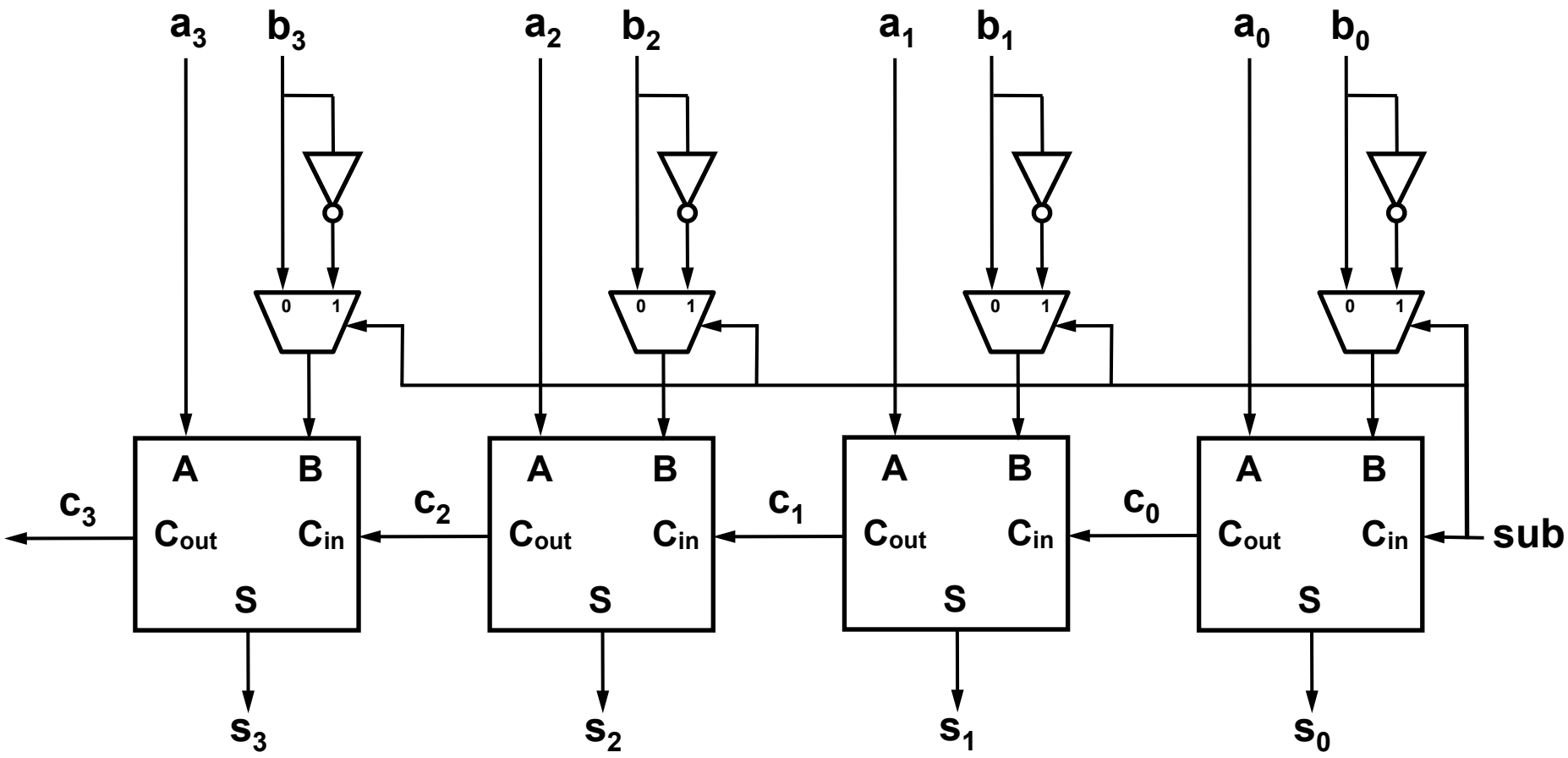
Four Bit Subtractor

negating the second operand





Four Bit Adder / Subtractor



Carry Lookahead Adder

- **Calculation of carry out of MSB is slow for large Ripple Carry Adders**
  - **Carry has to propagate from LSB to the MSB**
- **CLA adder calculates groups of carries in parallel**
- **C_{out} = 1 from a bit position happens for 2 reasons**
  - **Carry is *generated* from this bit position**
  - ***Carry in* to this position is *propagated* to the next**

Carry Generation and Propagation

```
   1   1
  01010010
+ 00010011
  01100101
```

```
    1 1 1
  11000110
+ 00101011
  11110001
```

```
  1111 111
  11110101
+ 01010011
  01001000
```

Carry Generation and Propagation

- Carry generation function for the i^{th} position

  $G_i = a_i \bullet b_i$

- Carry propagation function for the i^{th} position

  $P_i = a_i + b_i$

- Carry out of the i^{th} position

  $c_i = G_i + P_i \bullet c_{i-1}$

Carry Out Equations for 4-bit Adder

$c_0 = G_0 + P_0 \cdot c_{in}$ ← carry in to LSB

$G_i = a_i \cdot b_i$

$P_i = a_i + b_i$

$c_1 = G_1 + P_1 \cdot c_0$

$= G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot c_{in}$

$c_2 = G_2 + P_2 \cdot c_1$

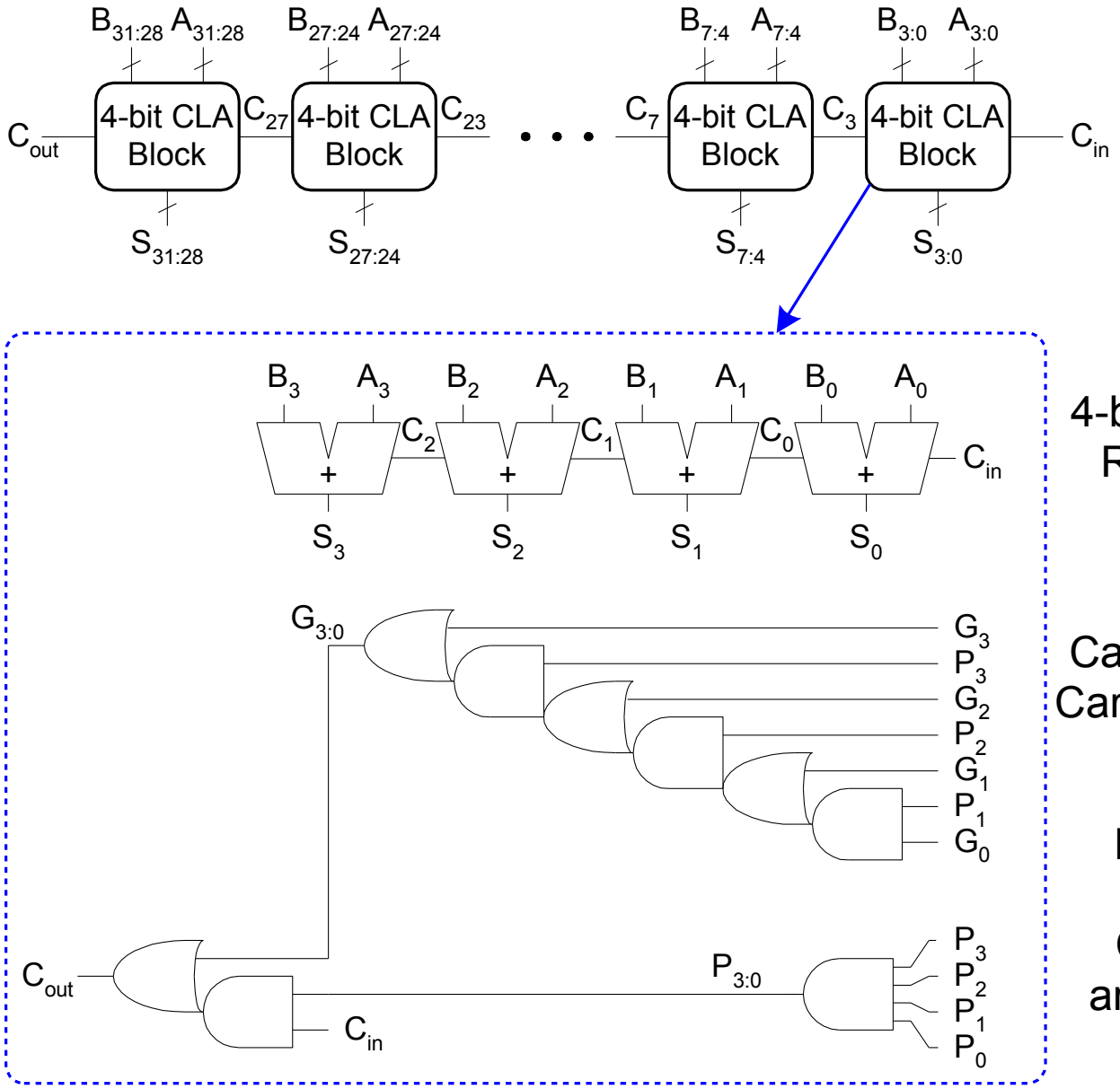
$= G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot c_{in}$

$c_3 = G_3 + P_3 \cdot c_2$

$= G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot c_{in}$

- G_3: generate carry from this position
- $P_3 \cdot G_2$: propagate carry generated in position 2
- $P_3 \cdot P_2 \cdot G_1$: propagate carry generated in position 1
- $P_3 \cdot P_2 \cdot P_1 \cdot G_0$: propagate carry generated in position 0
- $P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot c_{in}$: propagate carry in to position 0

32-bit CLA with 4-bit Building Blocks

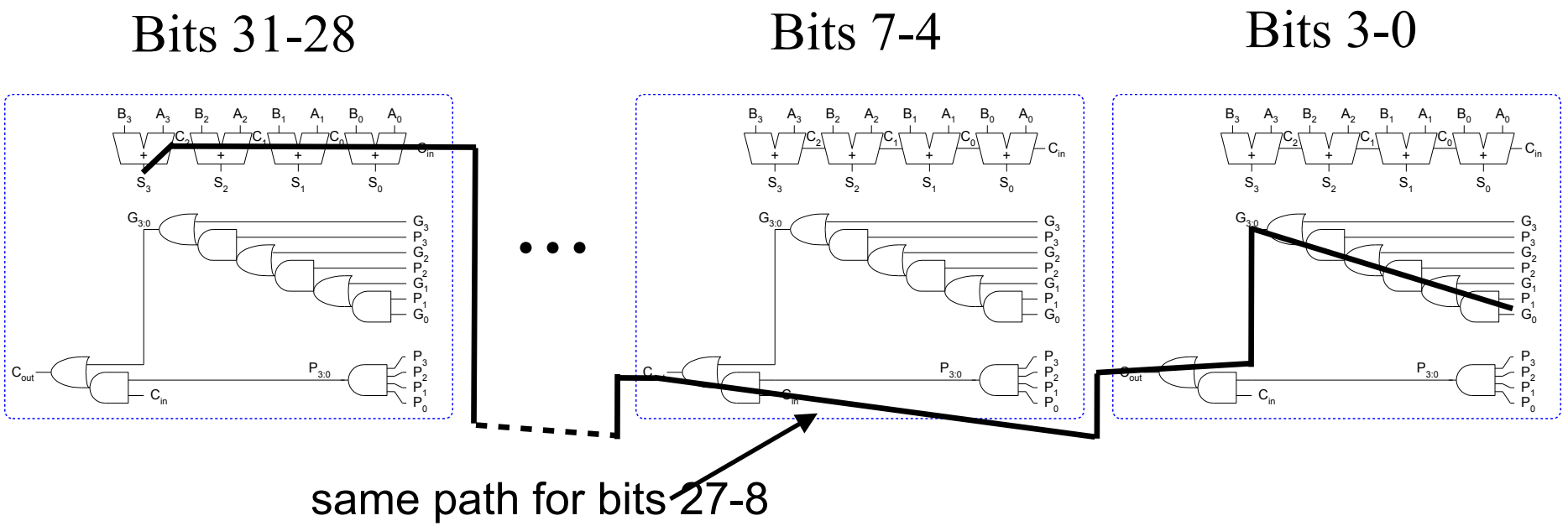


4-bit Sum using Ripple Carry Addition

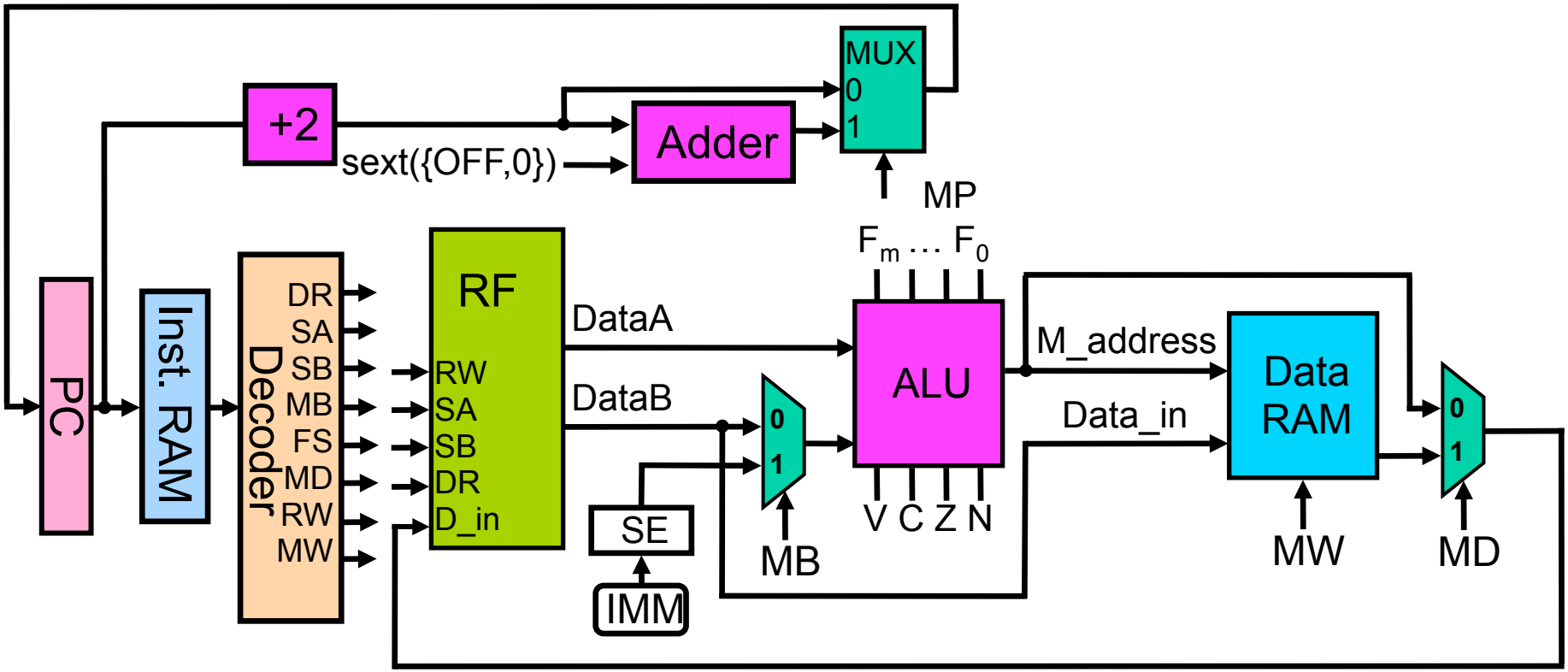
Carry Out using Carry Lookahead

P's and G's for all CLA blocks are generated *in parallel*

Longest Delay (Critical Path)



Our Processor Does Arithmetic and Logical Operations



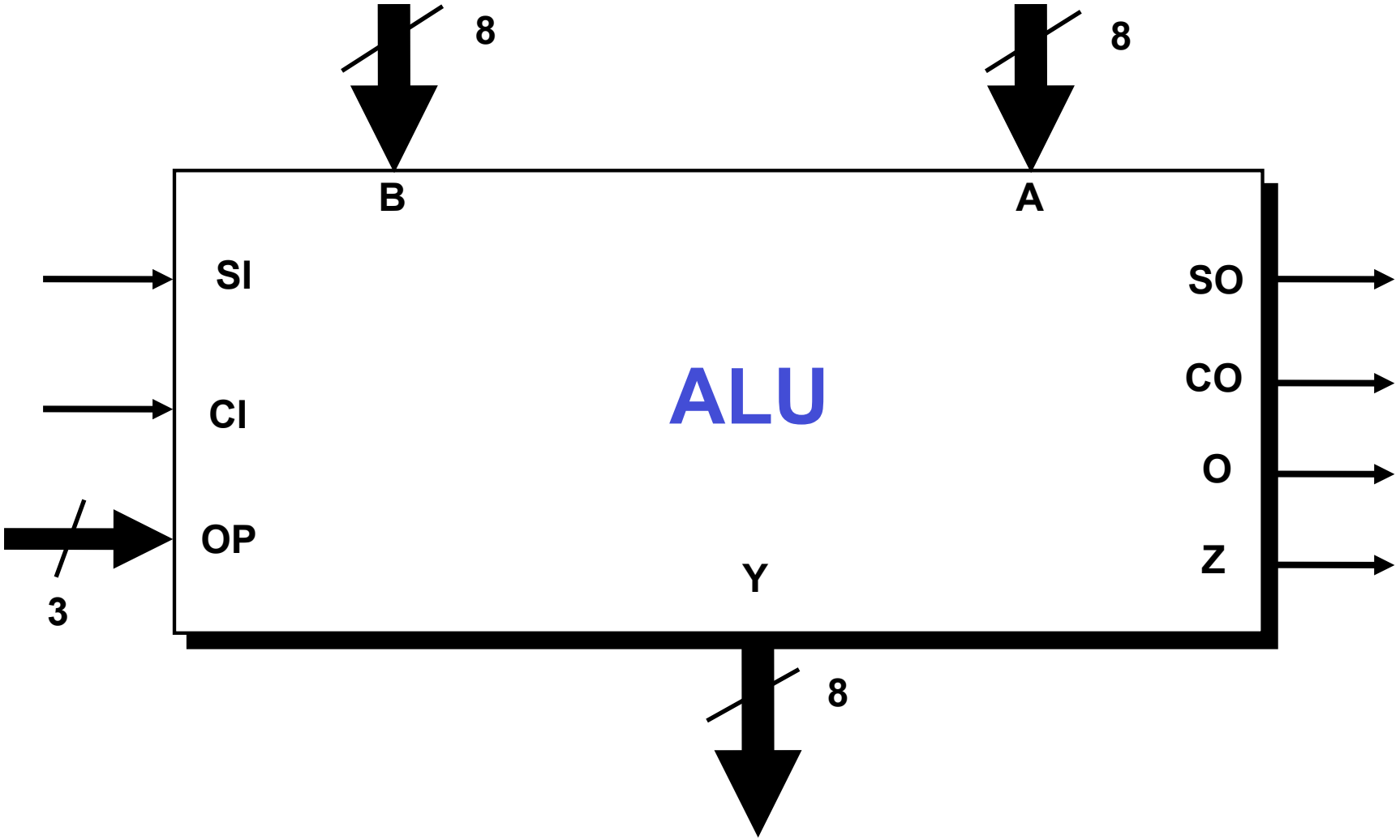
Arithmetic Logic Unit (ALU)

- Combinational logic circuit that combines a variety of operations into a single unit

- Typical operations may include
  - Addition and subtraction
  - Logical (OR, AND)
  - Shift and rotate
  - Comparisons

- Computing core of a processor

Example 8-bit Arithmetic Logic Unit

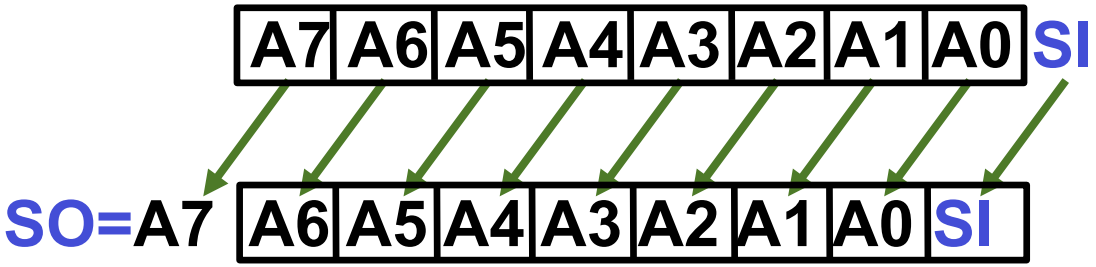


Example 8-bit Arithmetic Logic Unit

- **Operations**
  - Addition and Subtraction
  - Bitwise AND and OR
  - Shift Left and Shift Right
- **Inputs**
  - A, B, Carry In (CI)
  - Shift In (SI)
  - Code indicating operation to be performed (OP)
- **Outputs**
  - Y, Carry Out (CO)
  - Shift Out (SO)
  - Flags regarding the result of the operation

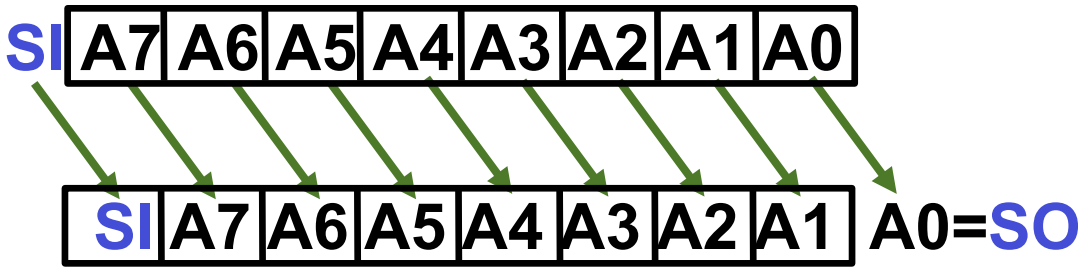
Shift Operations

- **Shift Left shifts each bit left by 1 position**



  - MSB shifted into SO, SI shifted into LSB

- **Shift Right shifts each bit right by 1 position**



  - LSB shifted into SO, SI shifted into MSB

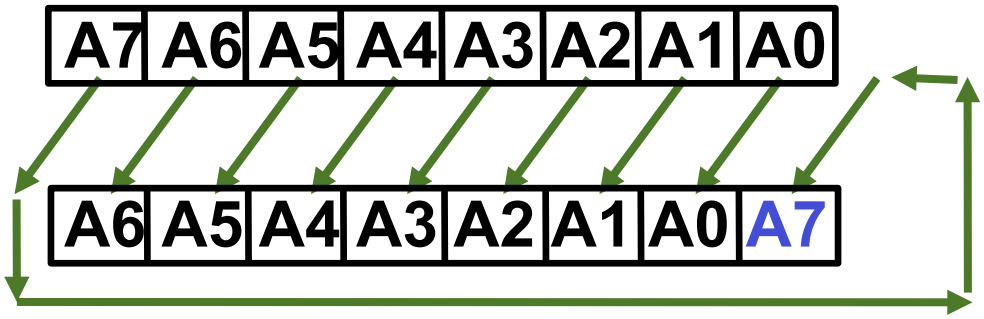
Other Common Shift Operations

- **Arithmetic Shift Right**
  - Preserves the sign bit
  - 11010000 ASR = 11101000
  - Equivalent to division by 2
  - Distinct from Logical Shift Right (previous slide)

- **Shift by n positions (not just 1)**
  - Multiplication by 2^n (left)
  - Division by 2^n (right)

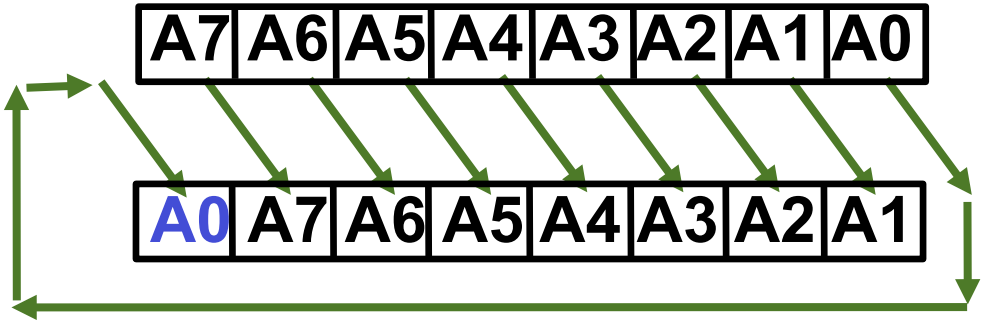
Rotate Operations

- **Rotate Left shifts each bit left by 1 position, but MSB wraps to LSB position**



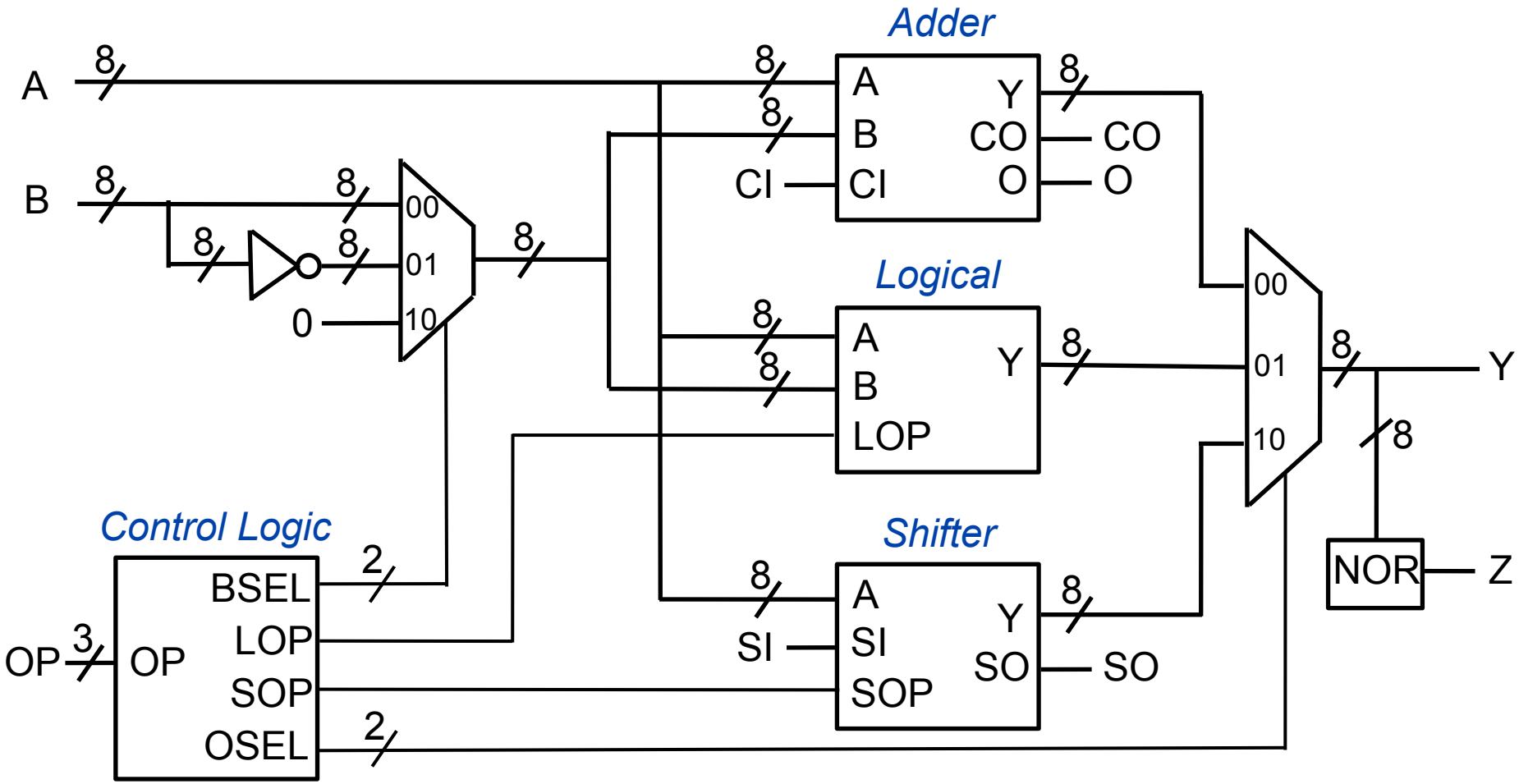
MSB shifted into LSB

- **Rotate Right shifts each bit right by 1 position, but LSB wraps to MSB position**



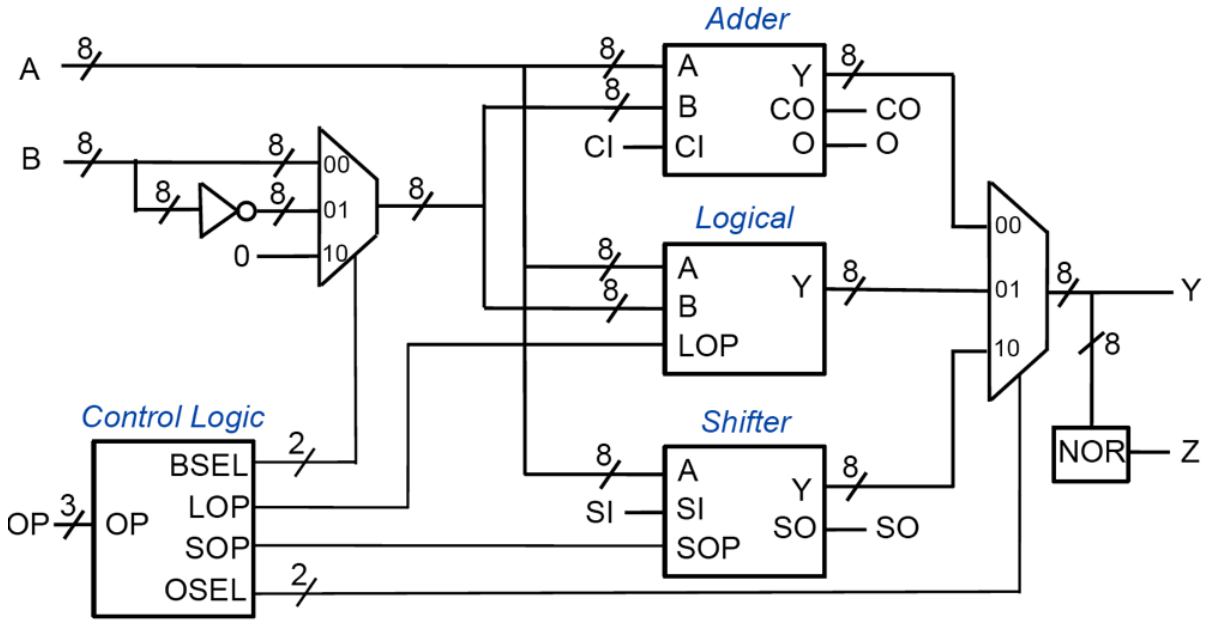
LSB shifted into MSB

ALU Block Diagram



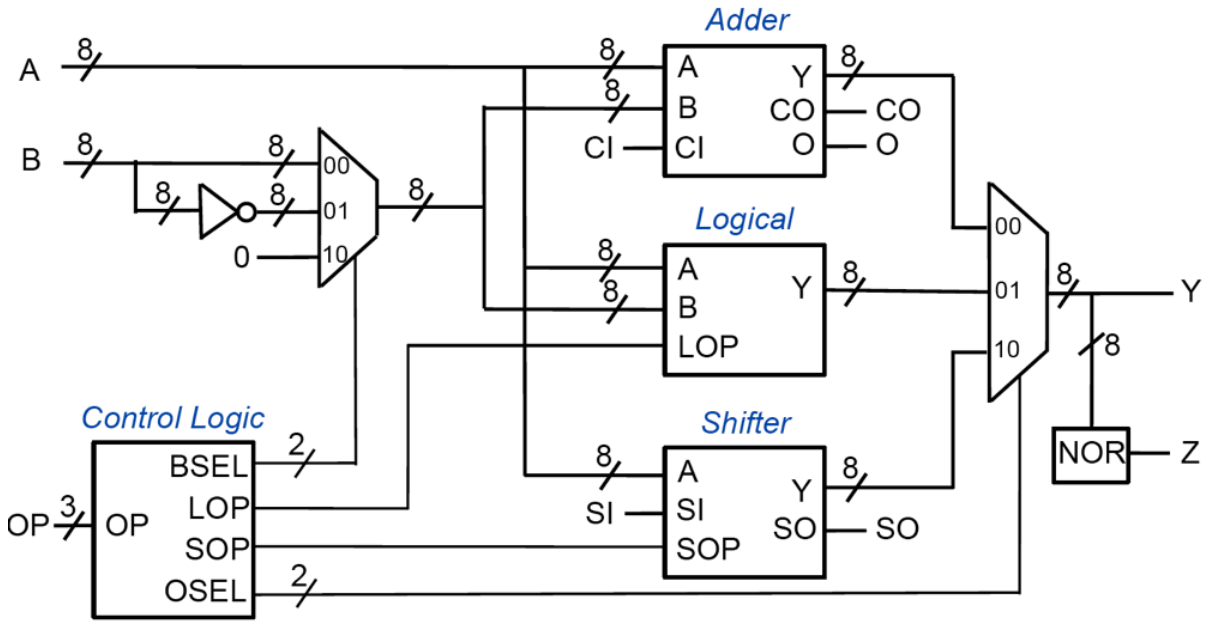
ALU Operations

NAME	OP	BSEL	CI	LOP	SOP	OSEL	Operation
ADD	000	00	0	-	-	00	Y = A + B + CI
SUB	001	01	1	-	-	00	Y = A + B' + CI
AND	011	00	-	0	-	01	Y = A AND B
OR	100	00	-	1	-	01	Y = A OR B
SHL	101	-	-	-	0	10	Y = A[6..0],SI
SHR	110	-	-	-	1	10	Y = SI,A[7..1]
PASS	111	10	0	-	-	00	Y = A + B + CI



Comparison Operations

- To compare A and B, perform A − B
  - If the result is 0, then A = B
  - Z flag set to 1 whenever ALU result is 0
  - Can check for A ≥ B and A < B by observing the MSB of the result (Y) of A − B



Before Next Class

- H&H 5.5

Next Time

Multiplication
Memories