



# Digital Logic Design

## Chapter 4 – Arithmetic Functions

浙江大学计算机学院 王总辉

zhwang@zju.edu.cn

<http://course.zju.edu.cn>

2023年10月

---



# Overview

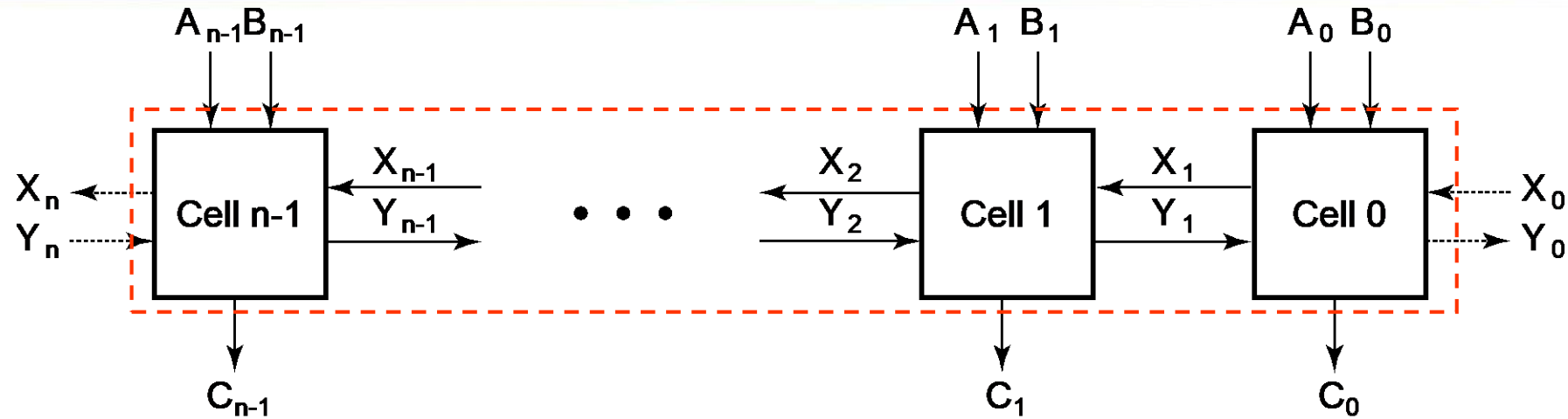
---

- **Iterative combinational circuits**
- **Binary adders**
  - Half and full adders
  - Ripple carry and carry lookahead adders
- **Binary subtraction**
- **Binary adder-subtractors**
  - Signed binary numbers
  - Signed binary addition and subtraction
  - Overflow
- **Binary multiplication**
- **Other arithmetic functions**
  - Design by contraction

# Iterative Combinational Circuits

- **Arithmetic functions**
  - Operate on binary vectors
  - Use the **same subfunction** in each bit position
- **Can design functional block for sub-function and repeat to obtain functional block for overall function**
- **Cell** – sub-function block
- **Iterative array** - a array of interconnected cells
- An iterative array can be in a **single** dimension (1D) or **multiple** dimensions

# Block Diagram of a 1D Iterative Array



- **Example:  $n = 32$**

- ❑ Number of inputs = ? 66
- ❑ Truth table rows = ?  $2^{66}$
- ❑ Equations with up to ? input variables = 66
- ❑ Equations with huge number of terms
- ❑ Design impractical!

- **Iterative array takes advantage of the regularity to make design feasible**



# Functional Blocks: Addition

---

- **Binary addition used frequently**
- **Addition Development:**
  - *Half-Adder* (HA), a 2-input bit-wise addition functional block,
  - *Full-Adder* (FA), a 3-input bit-wise addition functional block,
  - *Ripple Carry Adder*, an iterative array to perform **binary addition**, and
  - *Carry-Look-Ahead Adder* (CLA), a hierarchical structure to improve performance.

# Functional Block: Half-Adder

- A 2-input, 1-bit width binary adder that performs the following computations:

X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	0 0	0 1	0 1	1 0

- A half adder adds two bits to produce a two-bit sum
- The sum is expressed as a sum bit ,  
S and a carry bit, C
- The half adder can be specified  
as a truth table for S and C  $\Rightarrow$

X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

# Logic Simplification: Half-Adder

- The K-Map for S, C is:
- This is a pretty trivial map!

By inspection:

$$S = X \bar{Y} + \bar{X} Y = X \oplus Y$$

$$S = (X + Y) (\bar{X} + \bar{Y})$$

$$C = X Y$$

$$C = \overline{(\bar{X} \bar{Y})}$$

		Y	
		0	1
X	0	1 <sub>2</sub>	
	1		

		Y	
		0	1
X	0		
	1		1 <sub>3</sub>

- These equations lead to **several implementations.**

# Five Implementations: Half-Adder

- We can derive following sets of equations for a half-adder:

$$\begin{array}{ll} \text{(a)} \ S = X \bar{Y} + \bar{X} Y & \text{(d)} \ \underline{S} = (\underline{X} + \underline{Y}) \ \bar{C} \\ \quad C = X Y & \quad \bar{C} = (\bar{X} + \bar{Y}) \end{array}$$

$$\begin{array}{ll} \text{(b)} \ S = (X + Y) (\bar{X} + \bar{Y}) & \text{(e)} \ S = X \oplus Y \\ \quad C = X Y & \quad C = X Y \end{array}$$

$$\begin{array}{l} \text{(c)} \ S = \overline{(C + \bar{X} \bar{Y})} \\ \quad C = X Y \end{array}$$

- (a), (b), and (e) are SOP, POS, and XOR implementations for S.
- In (c), the C function is used as a term in the AND-NOR implementation of S
- in (d), the  $\bar{C}$  function is used in a POS term for S.

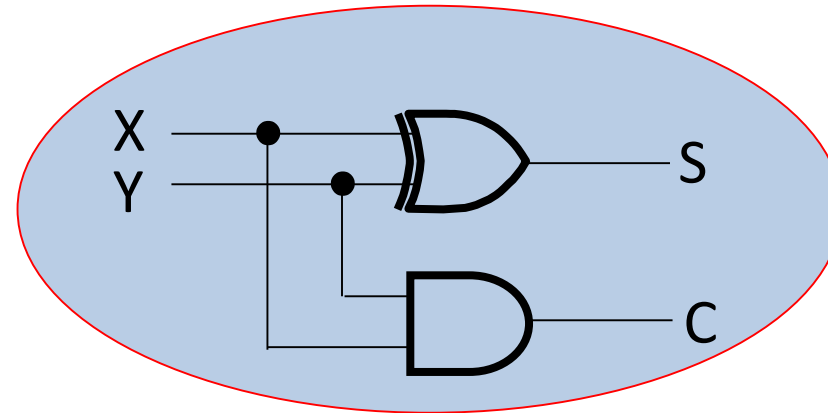


# Implementations: Half-Adder

- The most common half adder implementation is:

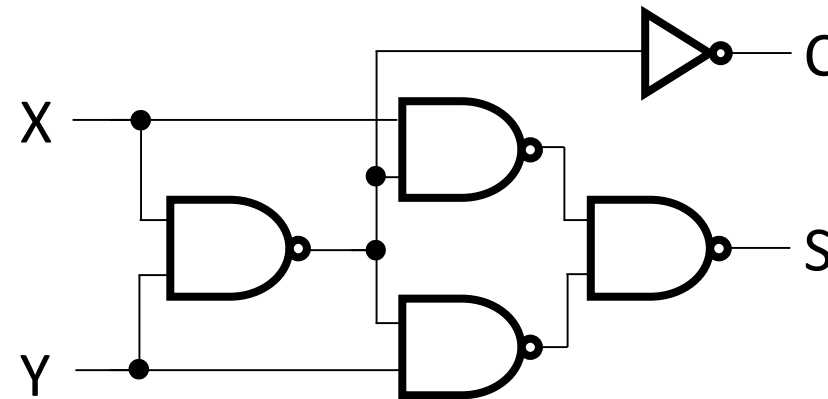
(e)

$$S = X \oplus Y$$
$$C = X \cdot Y$$



- A NAND only implementation is:

$$S = (X + Y) \cdot \overline{C}$$
$$C = \overline{(\overline{X \cdot Y})}$$



# Functional Block: Full-Adder

- A full adder is similar to a half adder, but includes a **carry-in bit** from lower stages. Like the half-adder, it computes a sum bit, S and a carry bit, C.

□ For a carry-in (Z) of 0, it is the same as the half-adder:

Z	0	0	0	0
X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	0 0	0 1	0 1	1 0

□ For a carry- in (Z) of 1:

Z	1	1	1	1
X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	0 1	1 0	1 0	1 1

# Logic Optimization: Full-Adder

- Full-Adder Truth Table:

X	Y	Z	C	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

- Full-Adder K-Map:

S		Y			
X		0	1	3	2
		1		1	
		Z			
		4	5	7	6

C		Y			
X		0	1	3	2
		1		1	
		Z			
		4	5	7	6

# Equations: Full-Adder

- From the K-Map, we get:

$$S = \overline{X} \overline{Y} Z + \overline{X} Y \overline{Z} + X \overline{Y} \overline{Z} + X Y Z$$
$$C = X Y + X Z + Y Z$$

- The S function is the three-bit XOR function (Odd Function):

$$S = X \oplus Y \oplus Z$$

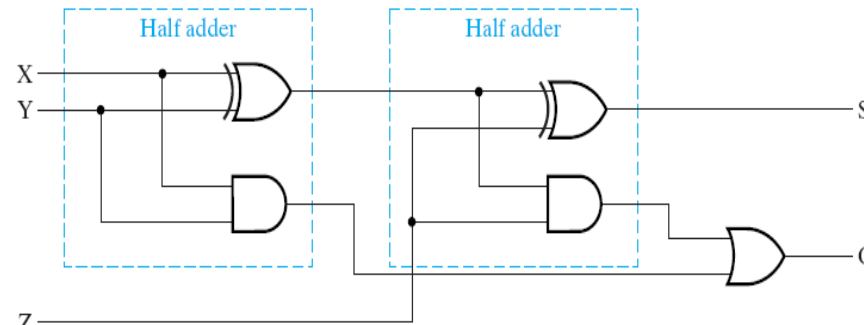
- The Carry bit C is 1 if both X and Y are 1 (the sum is 2), or if the sum is 1 and a carry-in (Z) occurs. Thus C can be re-written as:

$$C = X Y + (X \oplus Y) Z$$

- The term  $X \cdot Y$  is *carry generate*. Also:  $(X + Y)Z$
- The term  $X \oplus Y$  is *carry propagate*. Also:  $X + Y$

# Implementation: Full Adder

- Full Adder Schematic



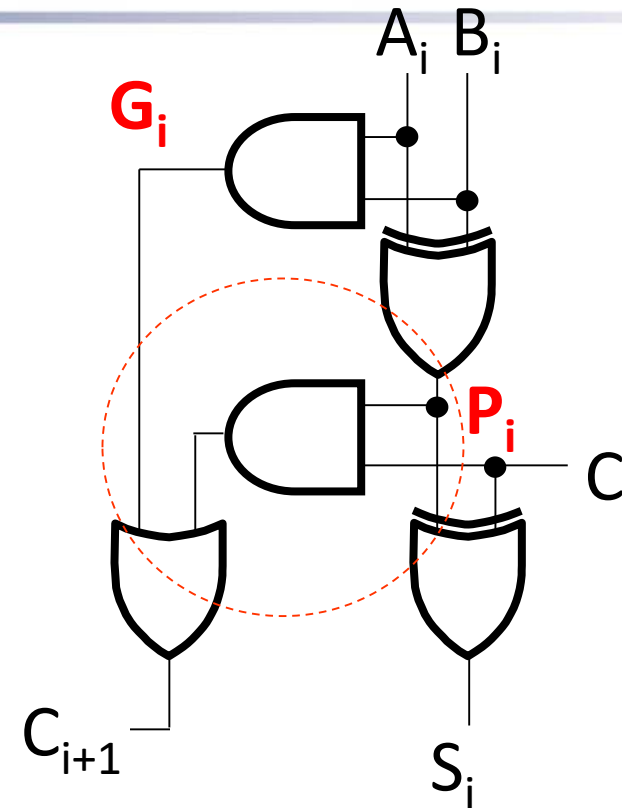
- Here X, Y, and Z, and C (from the previous pages) are A, B,  $C_i$  and  $C_o$ , respectively. Also,

G = generate and  
P = propagate.

- Note: This is really a combination of a 3-bit odd function (for S) and Carry logic (for  $C_o$ ):

(G = Generate) OR (P = Propagate AND  $C_i$  = Carry In)

$$C_o = G + P \cdot C_i$$



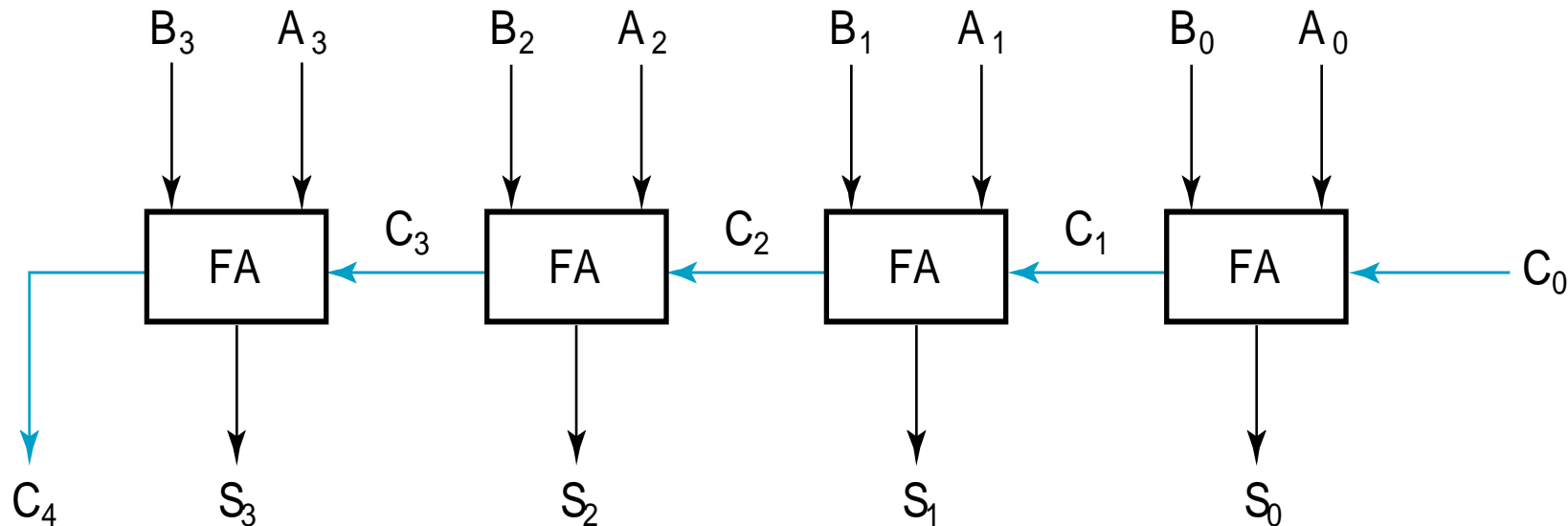
# Binary Adders

- To add multiple operands, we “bundle” logical signals together into vectors and use functional blocks that operate on the vectors
- Example: **4-bit ripple carry adder**: Adds input vectors  $A(3:0)$  and  $B(3:0)$  to get a sum vector  $S(3:0)$
- Note: carry out of cell  $i$  becomes carry in of cell  $i + 1$

Description	Subscript 3 2 1 0	Name
Carry In	0 1 1 0	$C_i$
Augend	1 0 1 1	$A_i$
Addend	<u>0 0 1 1</u>	$B_i$
Sum	1 1 1 0	$S_i$
Carry out	0 0 1 1	$C_{i+1}$

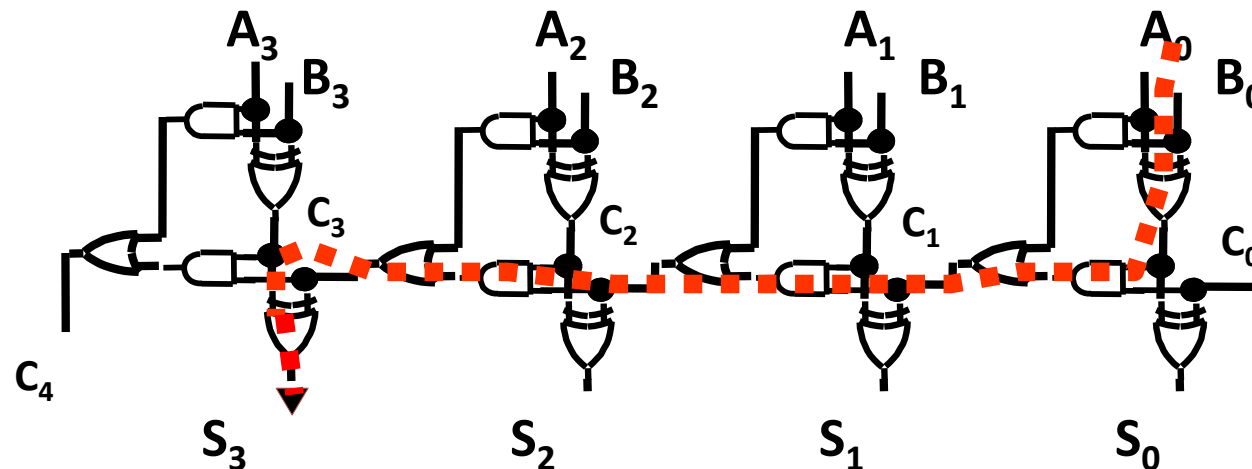
# 4-bit Ripple-Carry Binary Adder

- A four-bit Ripple Carry Adder made from four 1-bit Full Adders:



# Carry Propagation & Delay

- One problem with the addition of binary numbers is the length of time to propagate the ripple carry from the least significant bit to the most significant bit.
- The gate-level propagation path for a 4-bit ripple carry adder of the last example:

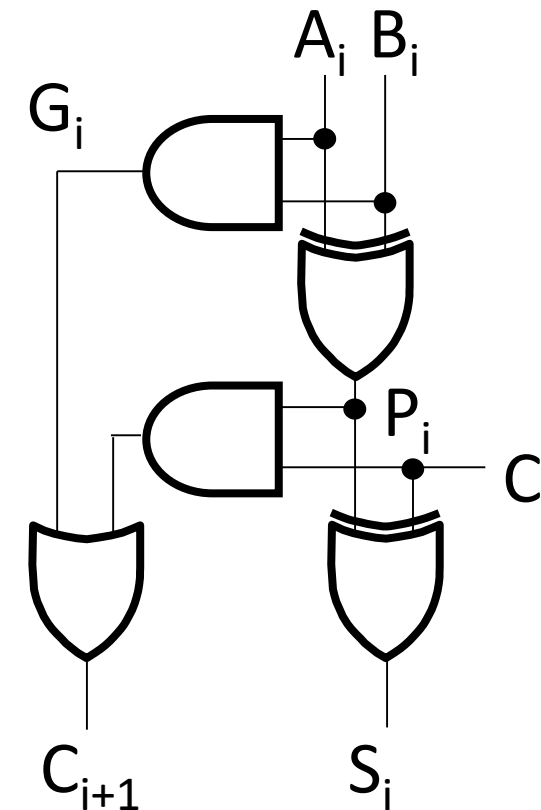


- Note: The "long path" is from  $A_0$  or  $B_0$  through the circuit to  $S_3$ .



# Carry Lookahead

- Given Stage  $i$  from a Full Adder, we know that there will be a **carry generated** when  $A_i = B_i = "1"$ , whether or not there is a carry-in.
- Alternately, there will be a **carry propagated** if the “half-sum” is “1” and a carry-in,  $C_i$  occurs, then  **$C_{i+1}=1$**
- These two signal conditions are called
  - **generate**, denoted as  $G_i$
  - **propagate**, denoted as  $P_i$



# Carry Lookahead (continued)

- In the ripple carry adder:
  - $G_i$ ,  $P_i$ , and  $S_i$  are **local** to each cell of the adder
  - $C_i$  is also **local** each cell
- In the carry lookahead adder, in order to reduce the length of the carry chain,  $C_i$  is changed to a more global function spanning multiple cells
- Defining the equations for the Full Adder in term of the  $P_i$  and  $G_i$ :

$$P_i = A_i \oplus B_i$$

$$G_i = A_i B_i$$

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i C_i$$

# Carry Lookahead Development

- $C_{i+1}$  can be removed from the cells and used to derive a set of carry equations spanning multiple cells.
- Beginning at the cell 0 with carry in  $C_0$

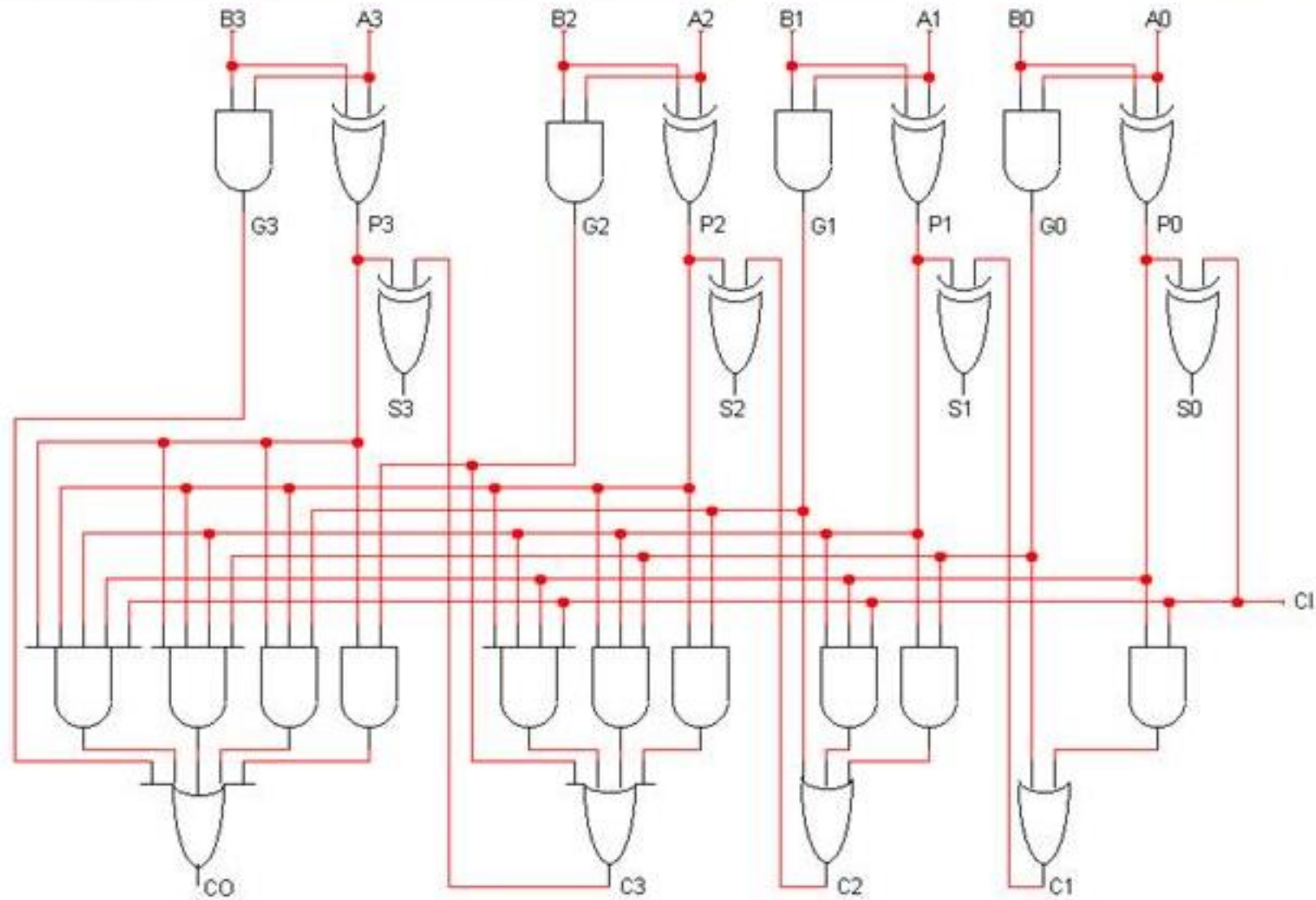
$$C_1 = G_0 + P_0 C_0$$

$$\begin{aligned} C_2 &= G_1 + P_1 C_1 = G_1 + P_1(G_0 + P_0 C_0) \\ &= G_1 + P_1 G_0 + P_1 P_0 C_0 \end{aligned}$$

$$\begin{aligned} C_3 &= G_2 + P_2 C_2 = G_2 + P_2(G_1 + P_1 G_0 + P_1 P_0 C_0) \\ &= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0 \end{aligned}$$

$$\begin{aligned} C_4 &= G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 \\ &\quad + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0 \end{aligned}$$

# Carry Lookahead Adder Schematic



# Group Carry Lookahead Logic

- Next slideshow shows the implementation of these equations for four bits. This could be extended to more than four bits; in practice, due to limited gate fan-in, such extension is not feasible.
- Instead, the concept is extended another level by considering *group generate* ( $G_{0-3}$ ) and *group propagate* ( $P_{0-3}$ ) functions:

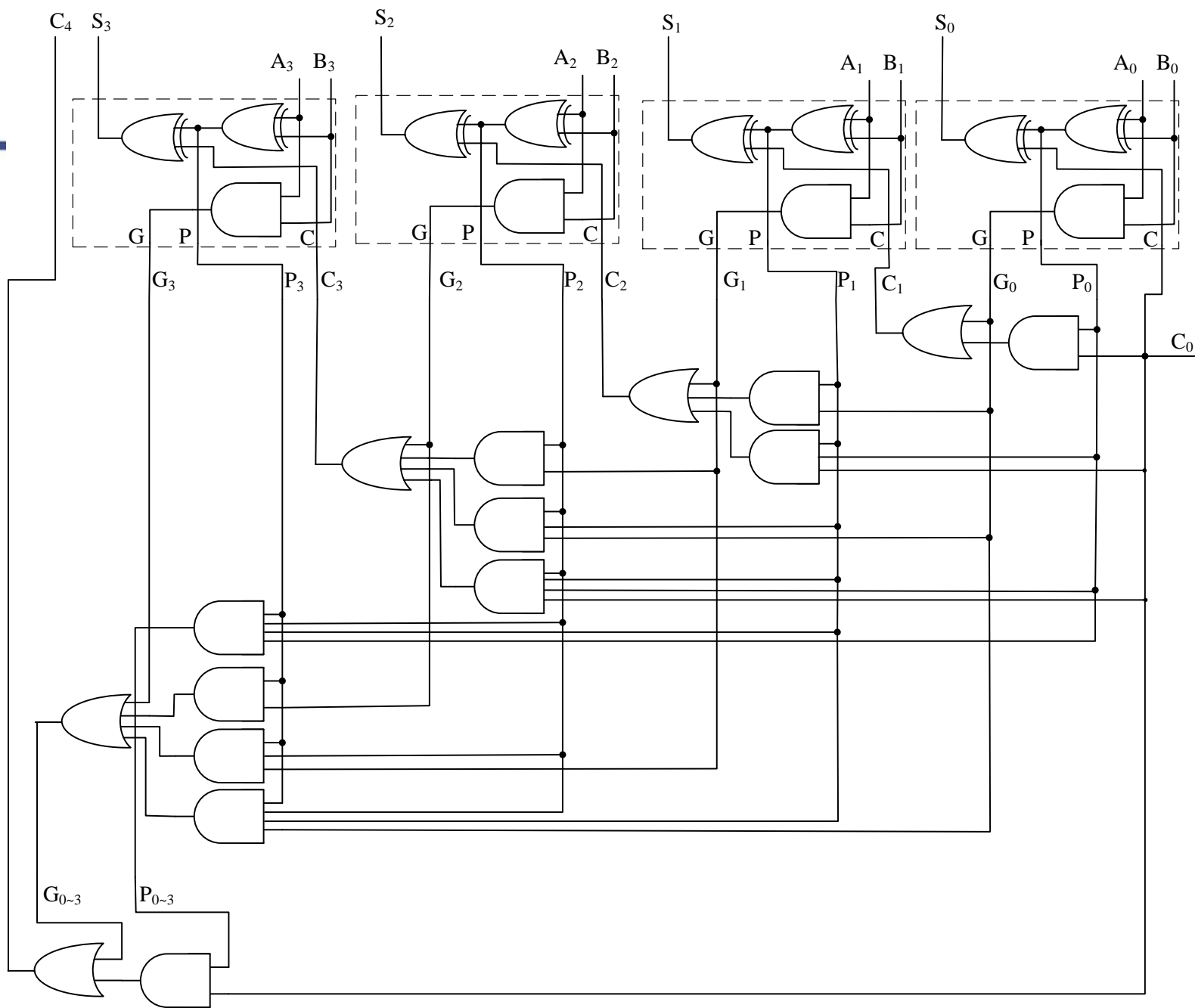
$$G_{0-3} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 P_0 G_0$$

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

- Using these two equations:

$$C_4 = G_{0-3} + P_{0-3} C_0$$

- Thus, it is possible to have four 4-bit adders use one of the same carry lookahead circuit to speed up 16-bit addition



# Extended Example:

## 16 carry lookahead adder



$$C_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0 = G_{0\sim3} + P_{0\sim3} C_0$$

$$C_8 = G_7 + P_7 G_6 + P_7 P_6 G_5 + P_7 P_6 P_5 G_4 + P_7 P_6 P_5 P_4 C_4 = G_{4\sim7} + P_{4\sim7} C_4$$

$$C_{12} = G_{11} + P_{11} G_{10} + P_{11} P_{10} G_9 + P_{11} P_{10} P_9 G_8 + P_{11} P_{10} P_9 P_8 C_8 = G_{8\sim11} + P_{8\sim11} C_8$$

$$C_{16} = G_{15} + P_{15} G_{14} + P_{15} P_{14} G_{13} + P_{15} P_{14} P_{13} G_{12} + P_{15} P_{14} P_{13} P_{12} C_{12} = G_{12\sim15} + P_{12\sim15} C_{12}$$

$$= G_{12\sim15} + P_{12\sim15} (G_{8\sim11} + P_{8\sim11} (G_{4\sim7} + P_{4\sim7} (G_{0\sim3} + P_{0\sim3} C_0)))$$

$$C_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$$

$$G_{0\sim3} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$$

$$G_{4\sim7} = G_7 + P_7 G_6 + P_7 P_6 G_5 + P_7 P_6 P_5 G_4$$

$$G_{8\sim11} = G_{11} + P_{11} G_{10} + P_{11} P_{10} G_9 + P_{11} P_{10} P_9 G_8$$

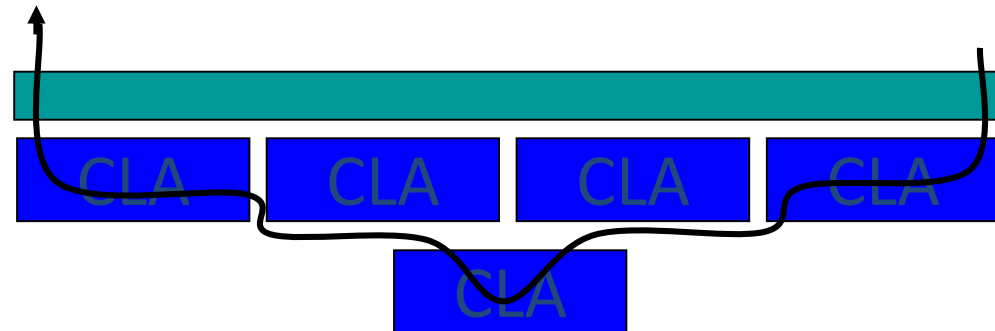
$$G_{12\sim15} = G_{15} + P_{15} G_{14} + P_{15} P_{14} G_{13} + P_{15} P_{14} P_{13} G_{12}$$

$$P_{0\sim3} = P_3 P_2 P_1 P_0$$

$$P_{4\sim7} = P_7 P_6 P_5 P_4$$

$$P_{8\sim11} = P_{11} P_{10} P_9 P_8$$

$$P_{12\sim15} = P_{15} P_{14} P_{13} P_{12}$$



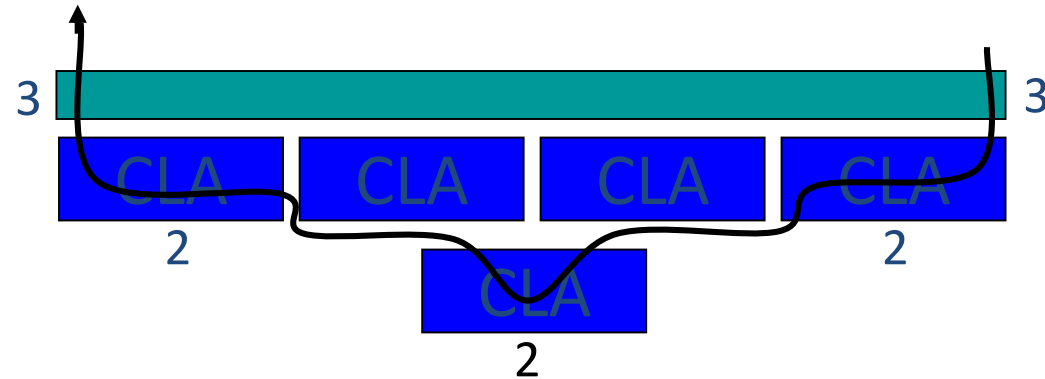
# Carry Lookahead Example

- **Specifications:**

- 16-bit CLA
- Delays:
  - NOT = 1
  - XOR = Isolated AND = 2+1
  - AND-OR = 2

- **Longest Delays:**

- Ripple carry adder =  $3 + 15 \times 2 + 3 = 36$
- CLA =  $3 + 3 \times 2 + 3 = 12$





# Unsigned Subtraction

## ● Algorithm:

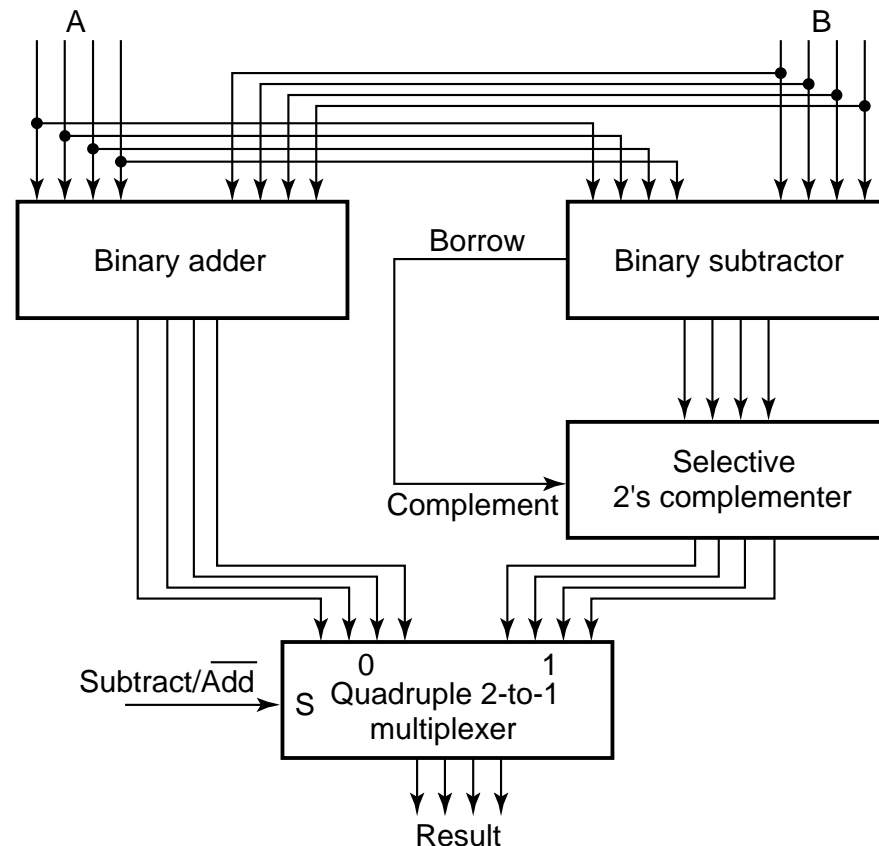
- Subtract the subtrahend N from the minuend M
- If **no end** borrow occurs, then  $M \geq N$ , and the result is a non-negative number and correct.
- If an end borrow occurs, the  $N > M$  and the difference  **$2^n - (M - N) = -(N - M)$**  is subtracted from  $2^n$ , and a minus sign is appended to the result.

## ● Examples:

0	1
1001	0100
- 0111	- 0111
0010	1101
	-----
	10000
	- 1101
	(-) 0011

# Unsigned Subtraction (continued)

- The subtraction,  $2^n - N$ , is taking the 2's complement of N
- To do both unsigned addition and unsigned subtraction requires:
  - Quite complex!
- Goal: Shared simpler logic for both addition and subtraction
- Introduce complements as an approach



# Complements

- **Two complements:**

- Diminished Radix Complement of N

- $(r - 1)$ 's complement for radix  $r$
    - 1's complement for radix 2
    - Defined as  $(r^n - 1) - N$

- Radix Complement

- $r$ 's complement for radix  $r$
    - 2's complement in binary
    - Defined as  $r^n - N$

- **Subtraction is done by adding the complement of the subtrahend**
- **If the result is negative, takes its 2's complement**



# Binary 1's Complement

- For  $r = 2$ ,  $N = 01110011_2$ ,  $n = 8$  (8 digits):

$$(r^n - 1) = 256 - 1 = 255_{10} \text{ or } 11111111_2$$

- The 1's complement of  $01110011_2$  is then:

$$\begin{array}{r} 11111111 \\ - 01110011 \\ \hline 10001100 \end{array}$$

- Since the  $2^n - 1$  factor consists of all 1's and since  $1 - 0 = 1$  and  $1 - 1 = 0$ , the one's complement is obtained by **complementing each individual bit** (bitwise NOT).



# Binary 2's Complement

- For  $r = 2$ ,  $N = 01110011_2$ ,  $n = 8$  (8 digits), we have:

$$(r^n) = 256_{10} \text{ or } 100000000_2$$

- The 2's complement of 01110011 is then:

$$100000000$$

$$- \underline{01110011}$$

$$10001101$$

- Note the result is the 1's complement plus 1, a fact that can be used in designing hardware



# Alternate 2's Complement Method

- **Given: an  $n$ -bit binary number, beginning at the least significant bit and proceeding upward:**
  - Copy all least significant 0's
  - Copy the first 1
  - Complement all bits thereafter.

- **2's Complement Example:**

10010100

- Copy underlined bits:

100

- and complement bits to the left:

01101100



# Subtraction with 2's Complement

- For n-digit, **unsigned** numbers M and N, find  $M - N$  in base 2:
  - Add the 2's complement of the subtrahend N to the minuend M:
$$M + (2^n - N) = M - N + 2^n$$
  - If  $M \geq N$ , the sum produces end carry  $r^n$  which is discarded; from above,  $M - N$  remains.
  - If  $M < N$ , the sum does not produce an end carry and, from above, is equal to  $2^n - (N - M)$ , the 2's complement of  $(N - M)$ .
  - To obtain the result  $-(N - M)$ , take the 2's complement of the sum and place a  $-$  to its left.



## Unsigned 2's Complement Subtraction Example 1

- Find  $01010100_2 - 01000011_2$   $M \geq N$

$$\begin{array}{r} 01010100 \\ - 01000011 \\ \hline \end{array} \xrightarrow{\text{2's comp}} \begin{array}{r} 1\ 01010100 \\ + 10111101 \\ \hline 00010001 \end{array}$$

- The carry of 1 indicates that no correction of the result is required.





## Unsigned 2's Complement Subtraction Example 2

- Find  $01000011_2 - 01010100_2$   $M < N$

$$\begin{array}{r} 01000011 \\ - 01010100 \\ \hline \end{array} \quad \begin{array}{r} 0\ 01000011 \\ + 10101100 \\ \hline \end{array}$$

$\xrightarrow{\text{2's comp}} 11101111 \quad \xrightarrow{\text{2's comp}}$

**$- 00010001$**

- The carry of 0 indicates that a correction of the result is required.
- Result =  $-(00010001)$



# Signed Integers

- Positive numbers and zero can be represented by unsigned  $n$ -digit, radix  $r$  numbers. We need a representation for negative numbers.
- To represent a sign (+ or –) we need exactly one more bit of information (1 binary digit gives  $2^1 = 2$  elements which is exactly what is needed).
- Since computers use binary numbers, by convention, the **most significant** bit is interpreted as a sign bit:

$$\mathbf{s} \ a_{n-2} \ \dots \ a_2 a_1 a_0$$

where:

$s = 0$  for Positive numbers

$s = 1$  for Negative numbers

and  $a_i = 0$  or  $1$  represent the magnitude in some form.



# Signed Integer Representations

---

- ***Signed-Magnitude*** – here the  $n - 1$  digits are interpreted as a positive magnitude.
- ***Signed-Complement*** – here the digits are interpreted as the rest of the complement of the number. There are two possibilities here:
  - *Signed 1's Complement*
    - Uses 1's Complement Arithmetic
  - *Signed 2's Complement*
    - Uses 2's Complement Arithmetic

# Signed Integer Representation Example

- $r=2, n=3$

Number	Sign -Mag.	1's Comp.	2's Comp.
+3	011	011	011
+2	010	010	010
+1	001	001	001
+0	000	000	000
-0	100	111	—
-1	101	110	111
-2	110	101	110
-3	111	100	101
-4	—	—	100



# Signed-Magnitude Arithmetic

---

- **If the parity of the three signs is 0:**

1. Add the magnitudes.
2. Check for overflow (a carry out of the MSB)
3. The sign of the result is the same as the sign of the first operand.

- **If the parity of the three signs is 1:**

1. Subtract the second magnitude from the first.
2. If a borrow occurs:
  - ▣ take the two's complement of result
  - ▣ and make the result sign the complement of the sign of the first operand.
3. Overflow will never occur.



# Sign-Magnitude Arithmetic Examples

---

- Example 1:  $0010$   
 $+ \underline{0101}$

- Example 2:  $0010$   
 $+ \underline{1101}$

- Example 3:  $1010$   
 $- \underline{0101}$



# Signed-Complement Arithmetic

---

- **Addition:**

1. Add the numbers including the sign bits, discarding a carry out of the sign bits (2's Complement), or using an end-around carry (1's Complement).
2. If the sign bits were the same for both numbers and the sign of the result is different, an overflow has occurred.
3. The sign of the result is computed in step 1.

- **Subtraction:**

Form the complement of the number you are subtracting and follow the rules for addition.



# Signed 2's Complement Examples

---

- **Example 1: 1101**

**+0011**

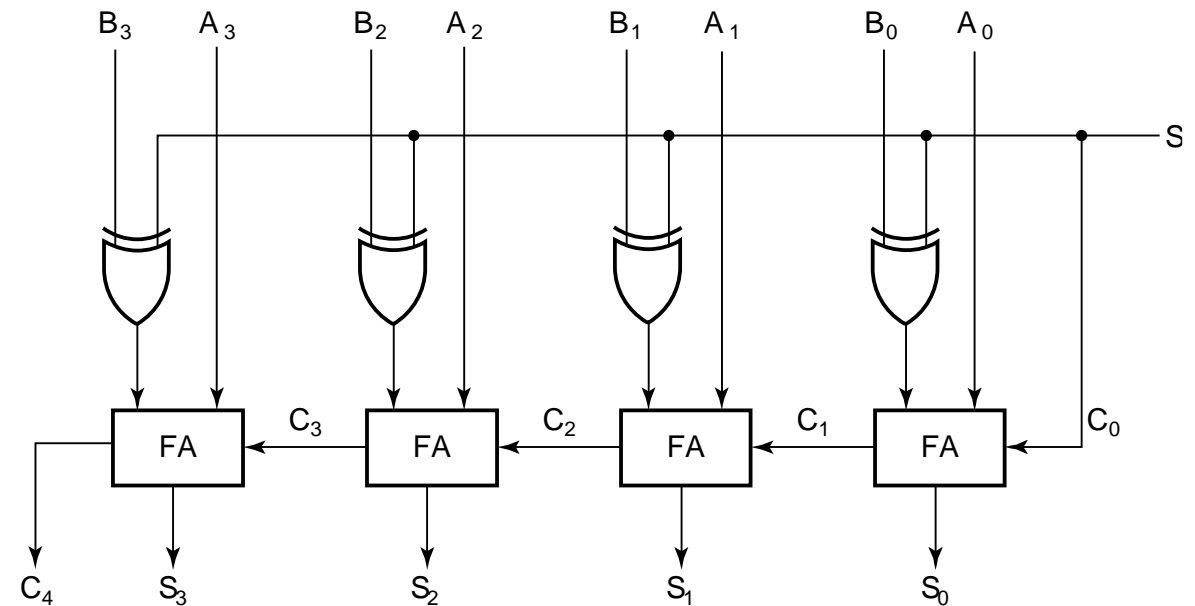
- **Example 2: 1101**

**-0011**



# 2's Complement Adder/Subtractor

- Subtraction can be done by addition of the 2's Complement.
  1. Complement each bit (1's Complement.)
  2. Add 1 to the result.
- The circuit shown computes  $A + B$  and  $A - B$ :
- For  $S = 1$ , subtract, the 2's complement of  $B$  is formed by using XORs to form the 1's comp and adding the 1 applied to  $C_0$ .
- For  $S = 0$ , add,  $B$  is passed through unchanged



# Overflow Detection

- **Overflow** occurs if  $n + 1$  bits are required to contain the result from an  $n$ -bit addition or subtraction
- **Overflow can occur for:**
  - Addition of two operands with the same sign
  - Subtraction of operands with different signs
- **Signed number overflow cases with correct result sign**

$$\begin{array}{cccc} 0 & 0 & 1 & 1 \\ +0 & -1 & -0 & +1 \\ \hline 0 & 0 & 1 & 1 \end{array}$$

- Detection can be performed by examining the result signs which should match the signs of the top operand

# Overflow Detection

- Signed number cases with carries  $C_n$  and  $C_{n-1}$  shown for correct result signs:

0	0	0	0	1	1	1	1
	0		0	1		1	
+	0	-	1	-	0	+	1
	0		0	1		1	

- Signed number cases with carries shown for erroneous result signs (indicating overflow):

0	1	0	1	1	0	1	0
	0		0	1		1	
+	0	-	1	-	0	+	1
	1		1	0		0	

- Simplest way to implement overflow  $V = C_n \oplus C_{n-1}$
- This works correctly only if 1's complement and the addition of the carry in of 1 is used to implement the complementation! Otherwise fails for -10 ... 0



# Other Arithmetic Functions

- Convenient to design the functional blocks by ***contraction*** - removal of redundancy from circuit to which input fixing has been applied
- **Functions**
  - Incrementing
  - Decrementing
  - Multiplication by Constant
  - Division by Constant
  - Zero Fill and Extension



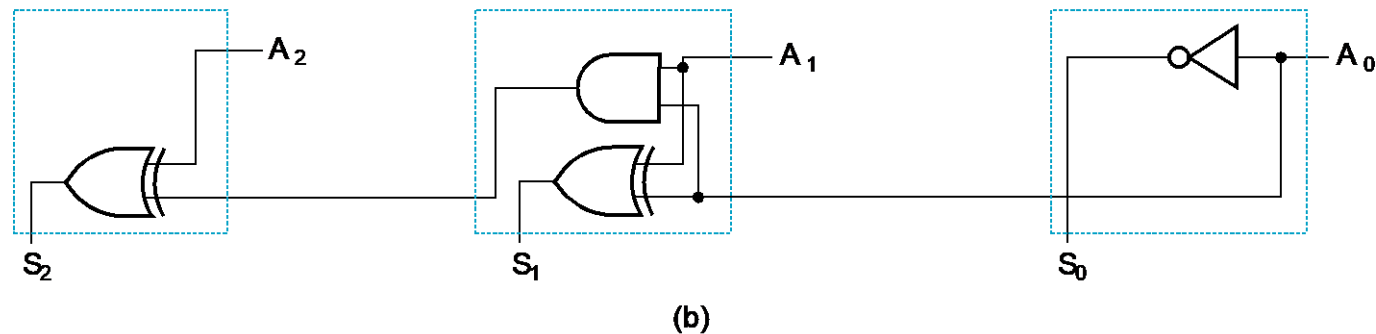
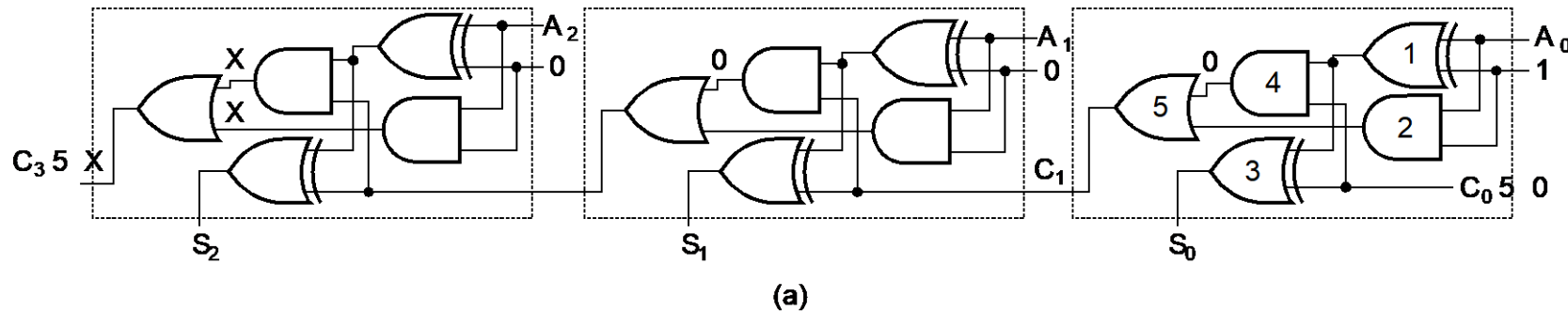
# Design by Contraction

- **Contraction is a technique for simplifying the logic in a functional block to implement a different function**
  - The new function must be realizable from the original function by applying rudimentary functions to its inputs
  - Contraction is treated here only for application of 0s and 1s (not for  $X$  and  $\overline{X}$ )
  - After application of 0s and 1s, equations or the logic diagram are simplified by using rules given on pages 224 - 225 of the text.

# Design by Contraction Example

- **Contraction of a ripple carry adder to incrementer for  $n = 3$**

- ▣ Set B = 001



- ▣ The middle cell can be repeated to make an incrementer with  $n > 3$ .



# Incrementing & Decrementing

## ● **Incrementing**

- Adding a fixed value to an arithmetic variable
- Fixed value is often 1, called *counting (up)*
- Examples:  $A + 1$ ,  $B + 4$
- Functional block is called *incrementer*

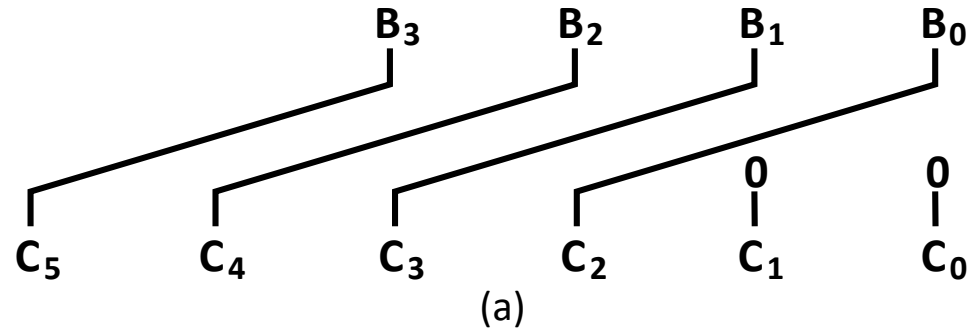
## ● **Decrementing**

- Subtracting a fixed value from an arithmetic variable
- Fixed value is often 1, called *counting (down)*
- Examples:  $A - 1$ ,  $B - 4$
- Functional block is called *decrementer*

# Multiplication/Division by $2^n$

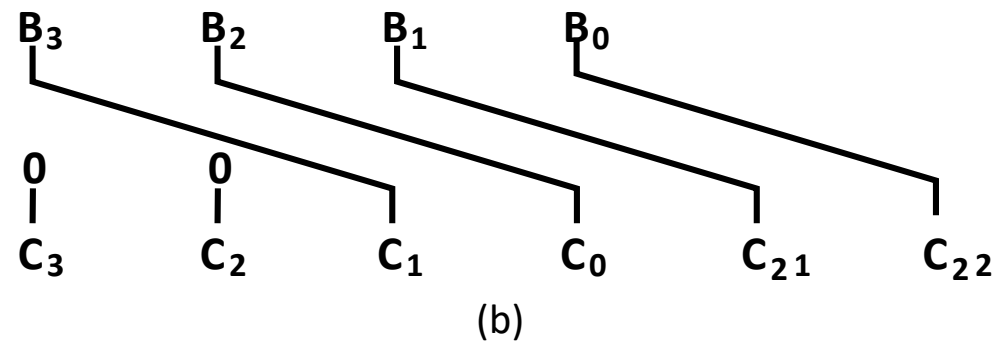
- **(a) Multiplication by 100**

- Shift left by 2



- **(b) Division by 100**

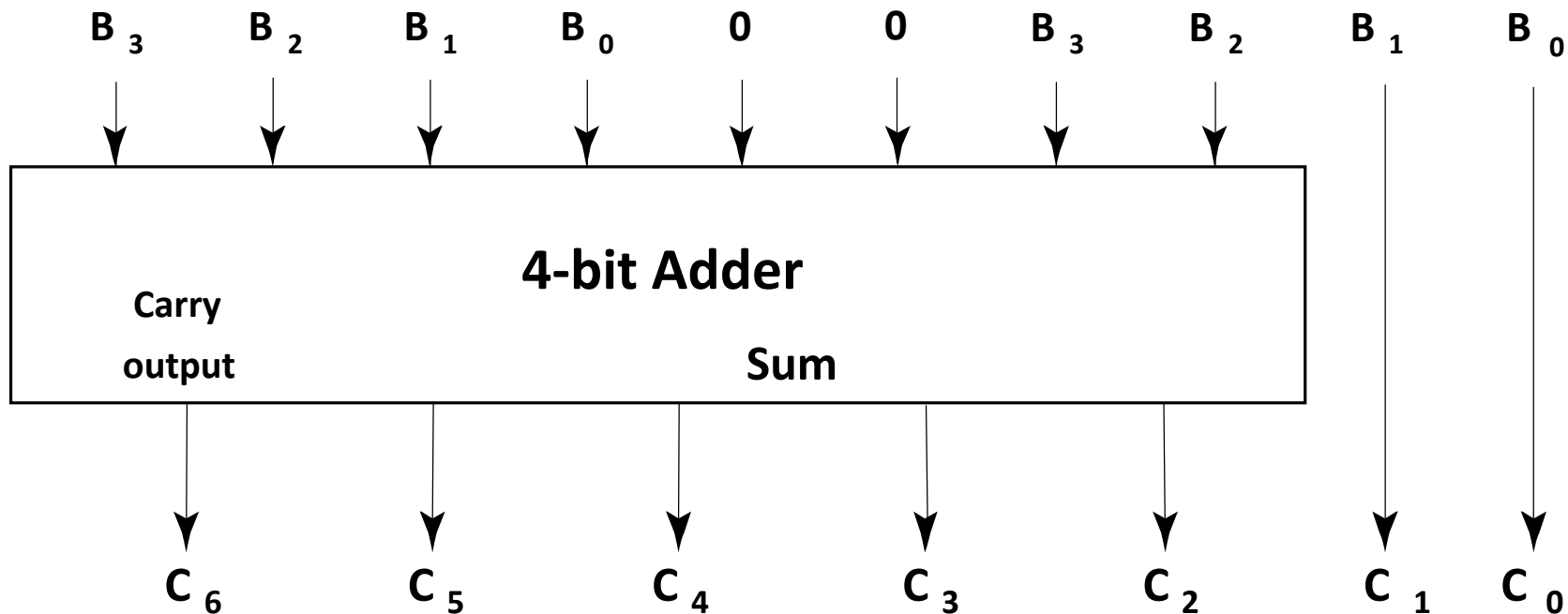
- Shift right by 2
- Remainder preserved





# Multiplication by a Constant

- Multiplication of  $B(3:0)$  by 101
- See text Figure 513 (a) for contraction





# Zero Fill

- ***Zero fill*** - filling an  $m$ -bit operand with 0s to become an  $n$ -bit operand with  $n > m$
- Filling usually is applied to the MSB end of the operand, but can also be done on the LSB end
- **Example: 11110101 filled to 16 bits**
  - MSB end: 0000000011110101
  - LSB end: 1111010100000000

# Extension

- ***Extension*** - increase in the number of bits at the MSB end of an operand by using a complement representation
  - Copies the MSB of the operand into the new positions
  - Positive operand example - 01110101 extended to 16 bits:  
0000000001110101
  - Negative operand example - 11110101 extended to 16 bits:  
1111111111110101



# Arithmetic Logic Unit(ALU) implementation

- **Not only arithmetic operations, but also for the logic operation**
- **Principle: adder as the core, through the appropriate control or transform to achieve a variety of arithmetic and logic operation**

Implementation {  
within adder, disable/masking certain gates input  
  
Transform the carry generated and propagated function  $G_i$   $P_i$



# Arithmetic Logic Unit (ALU)

---

- In this and the next section, we deal with detailed design of typical ALUs and shifters
- Decompose the ALU into:
  - An arithmetic circuit
  - A logic circuit
  - A selector to pick between the two circuits
- Arithmetic circuit design
  - Decompose the arithmetic circuit into:
    - An n-bit parallel adder
    - A block of logic that selects four choices for the B input to the adder
    - See next slide for diagram

# Arithmetic Circuit Design (continued)

- There are only four functions of  $B$  to select as  $Y$  in  $G = A + Y$ :

□ 0

□  $B$

□  $\overline{B}$

□ 1

$C_{in} = 0$

$$G = A$$

$$G = A + B$$

$$G = A + \overline{B}$$

$$G = A - 1$$

$C_{in} = 1$

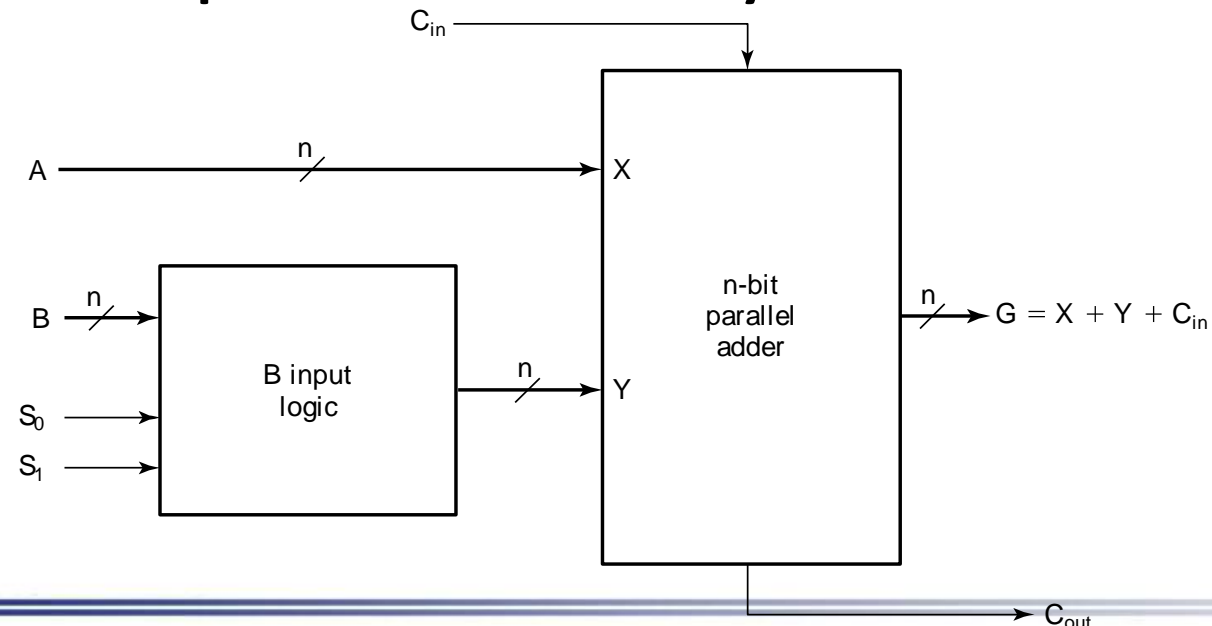
$$G = A + 1$$

$$G = A + B + 1$$

$$G = A + \overline{B} + 1$$

$$G = A$$

- What functions are implemented with carry-in to the adder = 0? =1?



# Arithmetic Circuit Design (continued)

- Adding selection codes to the functions of B:

□ TABLE 9-1

Function Table for Arithmetic Circuit

Select		Input	$G = (A + Y + C_{in})$	
$S_1$	$S_0$	$Y$	$C_{in} = 0$	$C_{in} = 1$
0	0	all 0s	$G = A$ (transfer)	$G = A + 1$ (increment)
0	1	$B$	$G = A + B$ (add)	$G = A + B + 1$
1	0	$\overline{B}$	$G = A + \overline{B}$	$G = A + \overline{B} + 1$ (subtract)
1	1	all 1s	$G = A - 1$ (decrement)	$G = A$ (transfer)

- The useful arithmetic functions are labeled in the table
- Note that all four functions of B produce at least one useful function

# Logic Circuit

- The text gives a circuit implemented using a multiplexer plus gates implementing: AND, OR, XOR and NOT
- Here we custom design a circuit for bit  $G_i$  by beginning with a truth table organized as a K-map and assigning (S1, S0) codes to AND, OR, etc.

- $$G_i = S_0 \bar{A}_i B_i + \bar{S}_1 A_i B_i + S_0 A_i \bar{B}_i + S_1 \bar{S}_0 \bar{A}_i$$

- Gate input count for MUX solution > 29
- Gate input count for above circuit < 20
- Custom design better

$S_1 S_0$	AND	OR	XOR	NOT
$A_i B_i$	0 0	0 1	1 1	1 0
0 0	0	0	0	1
0 1	0	1	1	1
1 1	1	1	0	0
1 0	0	1	1	0

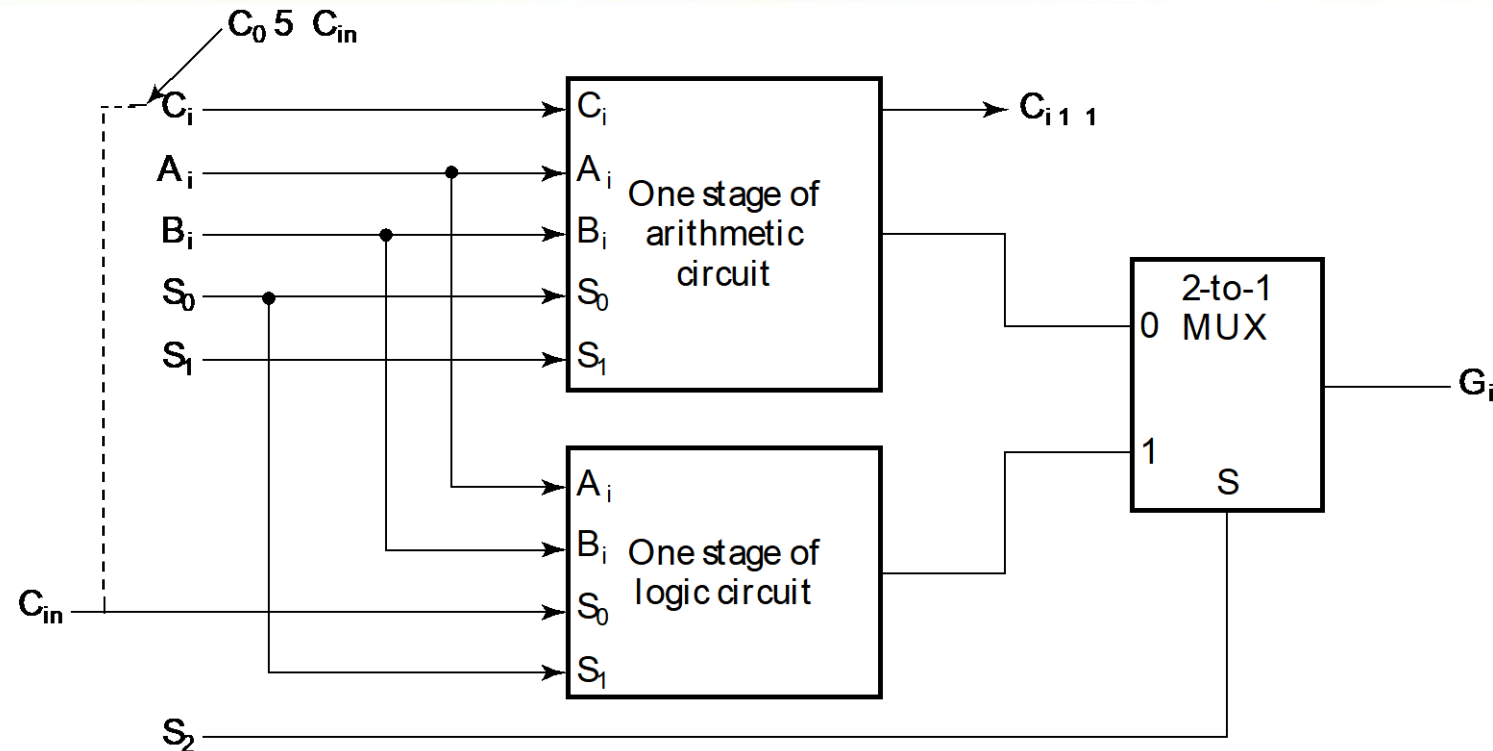




# Arithmetic Logic Unit (ALU)

- The custom circuit has interchanged the  $(S_1, S_0)$  codes for XOR and NOT compared to the MUX circuit. To preserve compatibility with the text, we use the MUX solution.
- Next, use the arithmetic circuit, the logic circuit, and a 2-way multiplexer to form the ALU. See the next slide for the bit slice diagram.
- The input connections to the arithmetic circuit and logic circuit have been assigned to prepare for seamless addition of the shifter, keeping the selection codes for the combined ALU and the shifter at 4 bits:
  - ❑ Carry-in  $C_i$  and Carry-out  $C_{i+1}$  go between bits
  - ❑  $A_i$  and  $B_i$  are connected to both units
  - ❑ A new signal  $S_2$  performs the arithmetic/logic selection
  - ❑ The select signal entering the LSB of the arithmetic circuit,  $C_{in}$ , is connected to the least significant selection input for the logic circuit,  $S_0$ .

# Arithmetic Logic Unit (ALU) (continued)



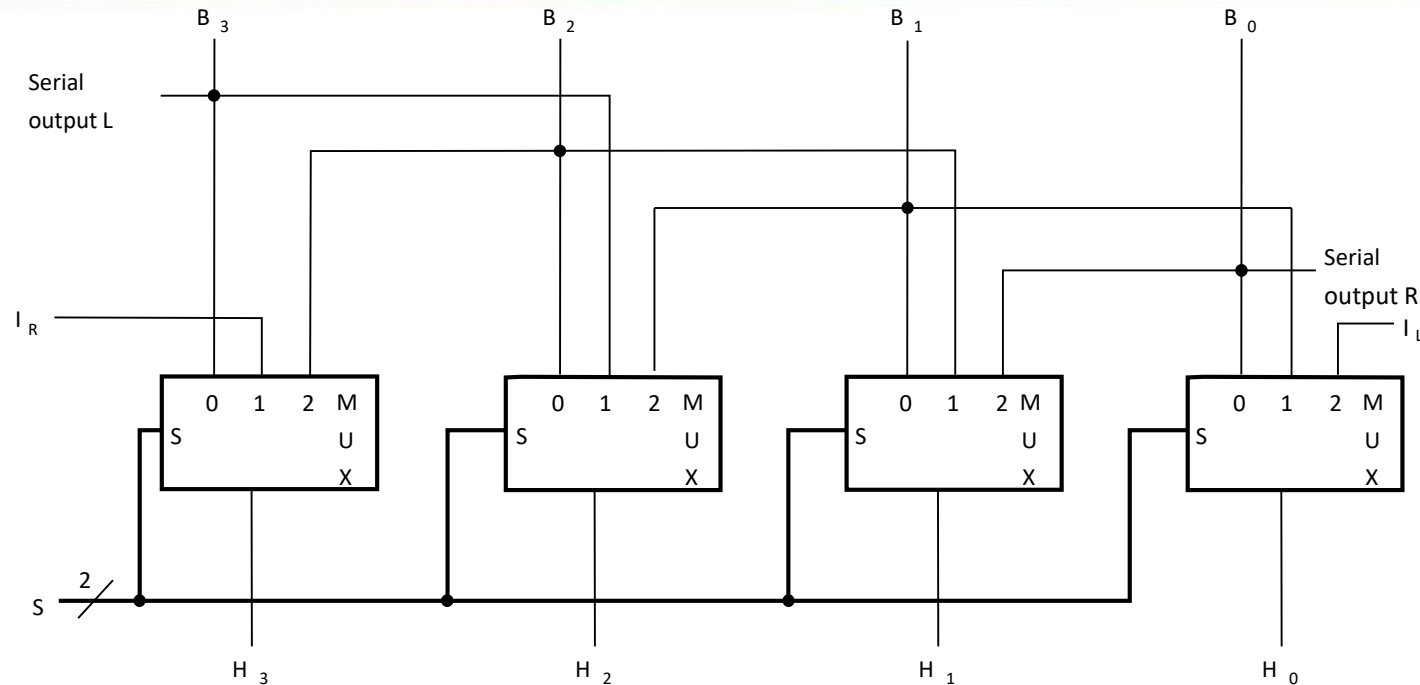
- The next most significant select signals,  $S_0$  for the arithmetic circuit and  $S_1$  for the logic circuit, are wired together, completing the two select signals for the logic circuit.
- The remaining  $S_1$  completes the three select signals for the arithmetic circuit.



# Combinational Shifter Parameters

- **Direction: Left, Right**
- **Number of positions with examples:**
  - Single bit:
    - 1 position
    - 0 and 1 positions
  - Multiple bit:
    - 1 to  $n - 1$  positions
    - 0 to  $n - 1$  positions
- **Filling of vacant positions**
  - Many options depending on instruction set
  - Here, will provide input lines or zero fill

# 4-Bit Basic Left/Right Shifter



- **Serial Inputs:**

- $I_R$  for right shift
- $I_L$  for left shift

- **Serial Outputs**

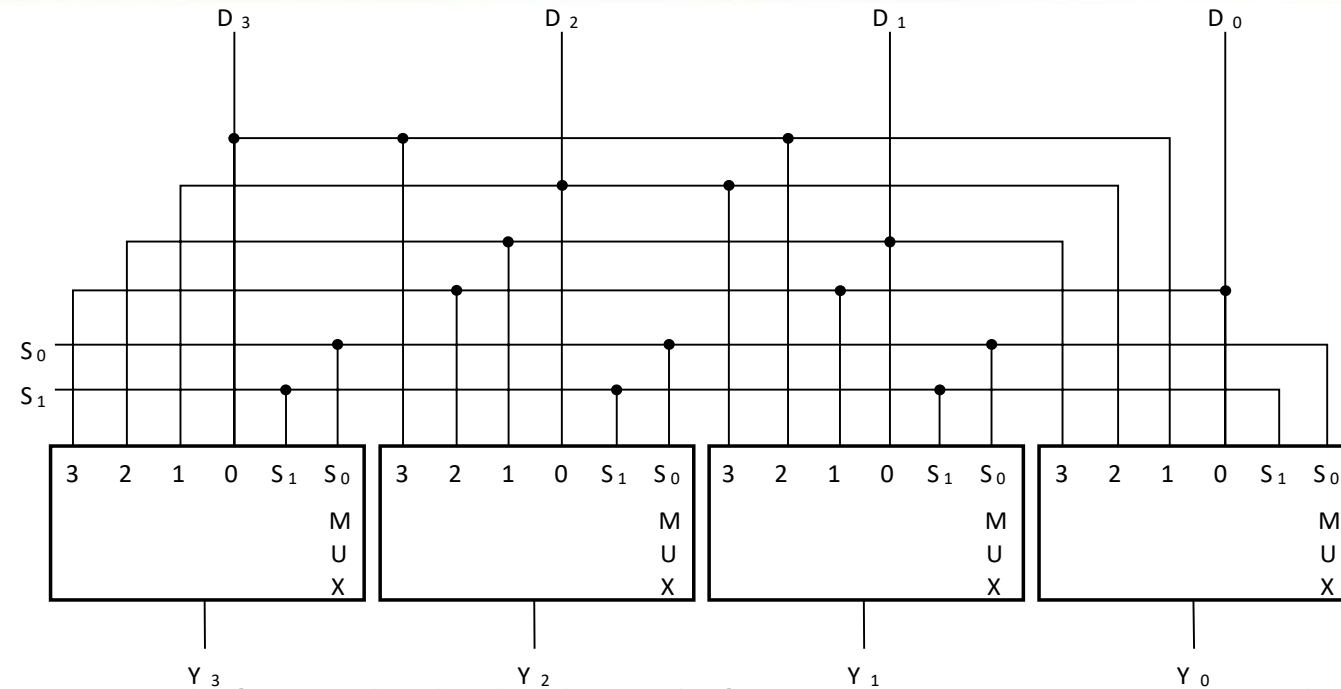
- R for right shift (Same as MSB input)
- L for left shift (Same as LSB input)



**Shift Functions:**

- (S1, S0) = 00 Pass B unchanged  
 01 Right shift  
 10 Left shift  
 11 Unused

# Barrel Shifter



- A rotate is a shift in which the bits shifted out are inserted into the positions vacated
- The circuit rotates its contents left from 0 to 3 positions depending on S:  
 S = 00 position unchanged      S = 10 rotate left by 2 positions  
 S = 01 rotate left by 1 positions      S = 11 rotate left by 3 positions
- See Table 10-3 in text for details

# Barrel Shifter (continued)

---

- **Large barrel shifters can be constructed by using:**
  - Layers of multiplexers - Example 64-bit:
    - Layer 1 shifts by 0, 16, 32, 48
    - Layer 2 shifts by 0, 4, 8, 12
    - Layer 3 shifts by 0, 1, 2, 3
    - See example in section 12-2 of the text
  - 2 - dimensional array circuits designed at the electronic level



**See Assignment Section  
of Website**



---

# Thank You !