

---

# Logic and Computer Design Fundamentals

## Chapter 3 – Combinational Logic Design

### Part 3 – Arithmetic Functions

Ming Cai

cm@zju.edu.cn

College of Computer Science and Technology  
Zhejiang University

# Overview

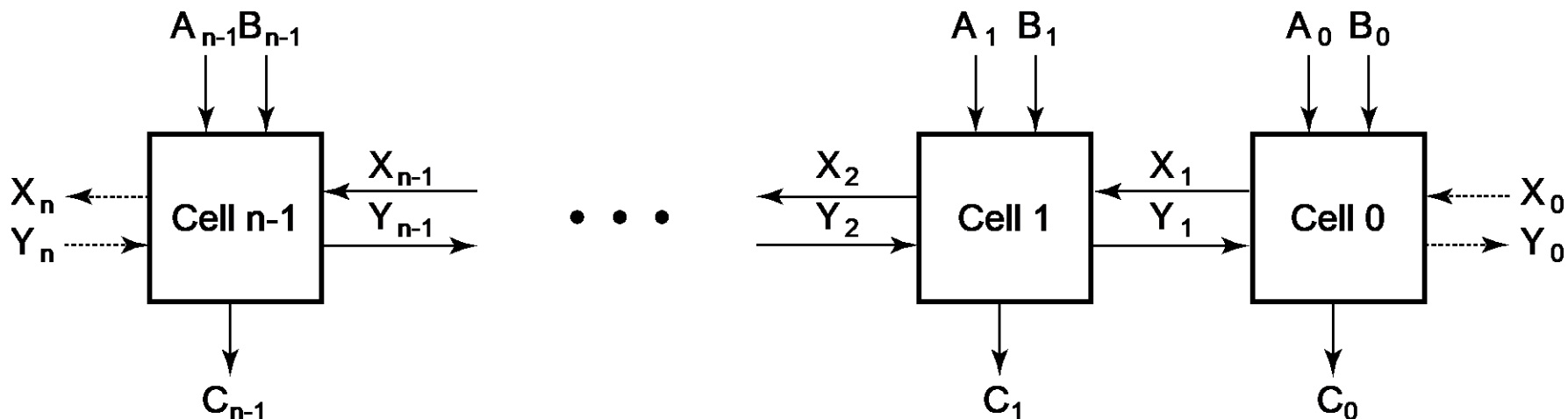
---

- **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

# Block Diagram of a 1D Iterative Array

$$\begin{array}{r}
 \phantom{+} 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \text{ Addend} \\
 + \phantom{0} 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 1 \text{ Augend} \\
 \hline
 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \text{ Sum}
 \end{array}$$

- **Example: addition of two 32-bit binary integers**
  - Number of inputs = **64**
  - Truth table rows =  **$2^{64}$**
  - Equations with up to **64 variables**
  - **Design impractical !**
- **Iterative array takes advantage of the regularity to make design feasible**



# Iterative Combinational Circuits

---

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

# 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:

<b>X</b>	<b>0</b>	<b>0</b>	<b>1</b>	<b>1</b>
<b>+ Y</b>	<b>+ 0</b>	<b>+ 1</b>	<b>+ 0</b>	<b>+ 1</b>
<b>C S</b>	<b>0 0</b>	<b>0 1</b>	<b>0 1</b>	<b>1 0</b>

- 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  $\Rightarrow$

<b>X</b>	<b>Y</b>	<b>S</b>	<b>C</b>
<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>
<b>0</b>	<b>1</b>	<b>1</b>	<b>0</b>
<b>1</b>	<b>0</b>	<b>1</b>	<b>0</b>
<b>1</b>	<b>1</b>	<b>0</b>	<b>1</b>

# Logic Simplification: Half-Adder

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

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

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

- and

$$C = XY$$

$$C = \overline{\overline{(XY)}}$$

- These equations lead to several implementations.

S		C	
	Y		Y
	0	0	1
X	1 <sub>2</sub>	X	1 <sub>3</sub>

# Five Implementations: Half-Adder

---

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

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

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

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

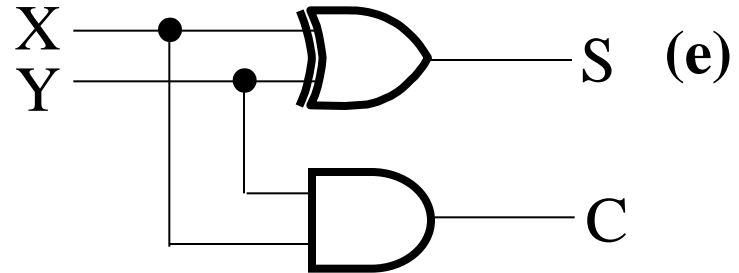
- (a), (b), and (e) are SOP, POS, and XOR implementations for S.
- In (c), the C function is used as a term in the AND-NOR implementation of S, and in (d), the  $\bar{C}$  function is used in a POS term for S.



# Implementations: Half-Adder

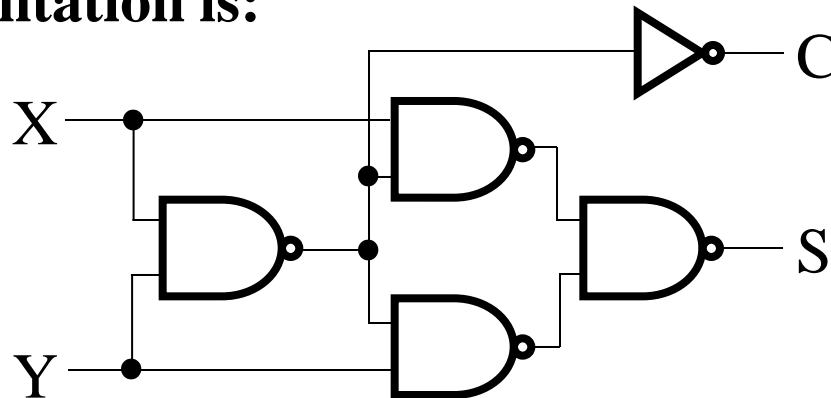
- The most common half adder implementation is:

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



- A NAND only implementation is:

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



# 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
<u>+ Y</u>	<u>+ 0</u>	<u>+ 1</u>	<u>+ 0</u>	<u>+ 1</u>
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
<u>+ Y</u>	<u>+ 0</u>	<u>+ 1</u>	<u>+ 0</u>	<u>+ 1</u>
C S	0 1	1 0	1 0	1 1

# Logic Optimization: Full-Adder

- Full-Adder Truth Table:

X	Y	Z	S	C
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 K-Map:

K-Map for Sum (S):

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

K-Map for Carry (C):

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

## Equations: Full-Adder

- **From the K-Map, we get:**

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

$$C = XY + \overline{X}\overline{Y}Z + \overline{X}YZ = XY + XZ + YZ = XY + (X + Y)Z$$

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

$$\mathbf{S} = \mathbf{X} \oplus \mathbf{Y} \oplus \mathbf{Z}$$

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

$$\mathbf{C} = \mathbf{X} \mathbf{Y} + (\mathbf{X} \oplus \mathbf{Y}) \mathbf{Z}$$

- The term  $X \cdot Y$  is *carry generate*.
- The term  $X \oplus Y$  is *carry propagate*.

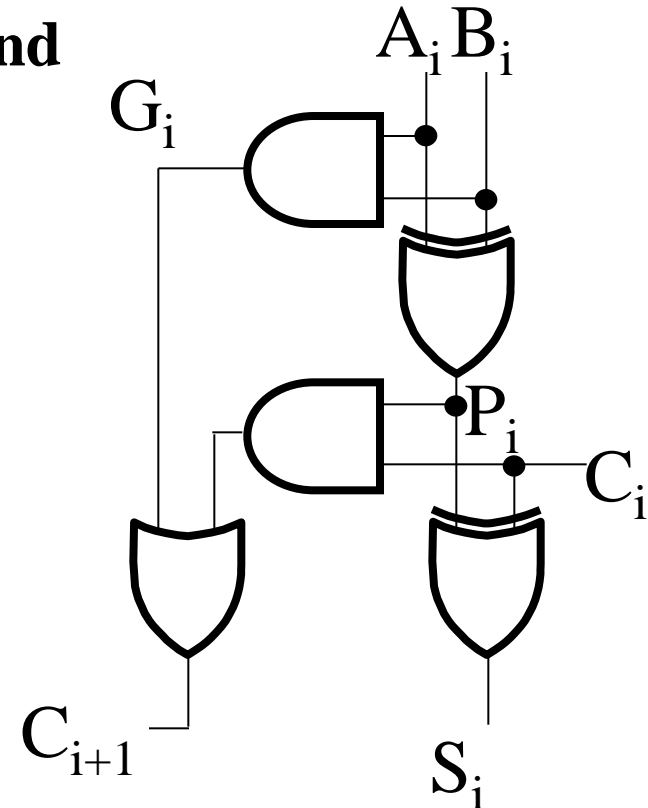
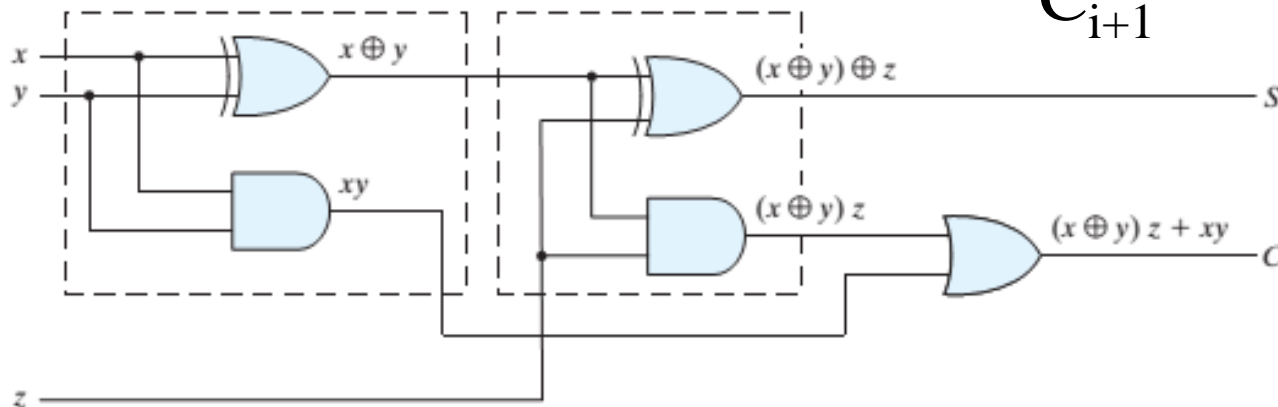
# Implementation: Full Adder

- $S_i$  is the XOR result of variable  $A_i$ ,  $B_i$  and  $C_i$ . And carry function  $C_{i+1}$  can be expressed as bellow:

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

- 1)  $G_i$  is called **generate function**
- 2)  $P_i$  is called **propagate function**

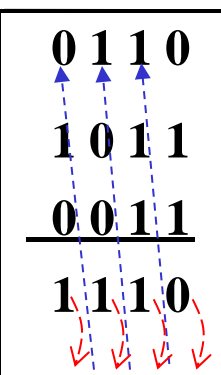
- Implementation of full adder with two half adders and an OR gate



# Binary Adders

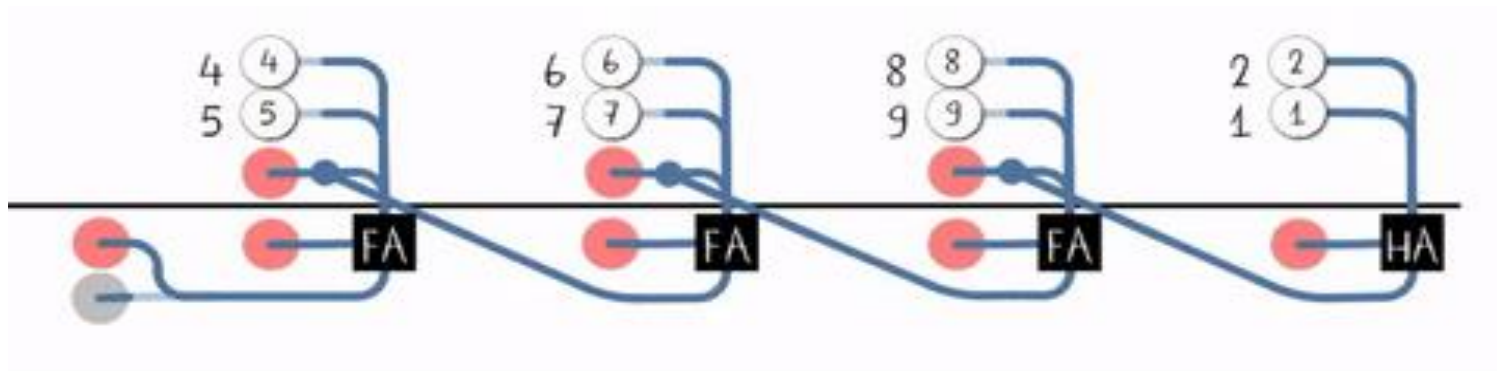
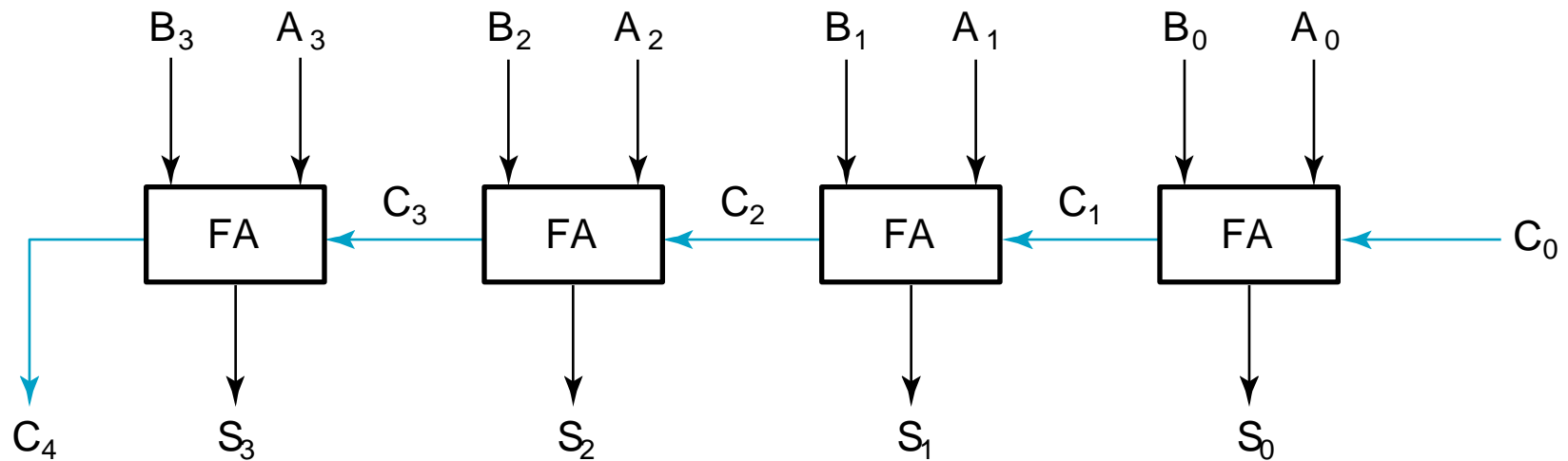
- To add multiple operands, we “bundle” logical signals together into vectors and use functional blocks that operate on the vectors
- **Example: 4-bit ripple carry adder:** Adds input vectors  $A(3:0)$  and  $B(3:0)$  to get a sum vector  $S(3:0)$
- **Note:** carry out of cell  $i$  becomes carry in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



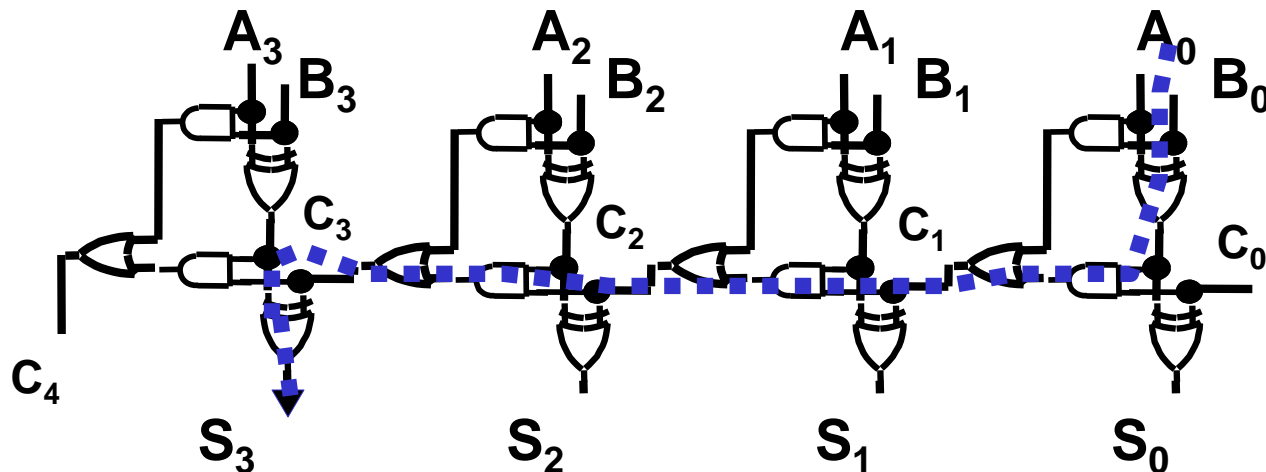
# 4-bit Ripple-Carry Binary Adder

- 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:

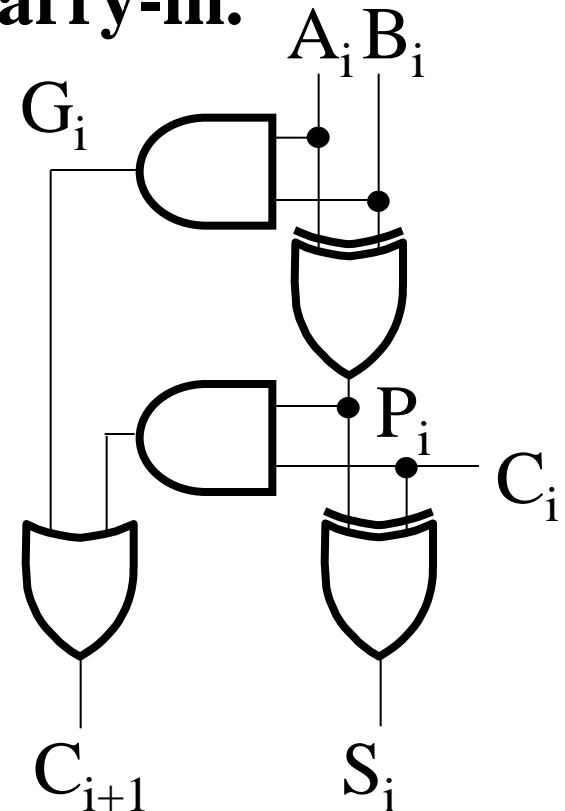


- Note: The "long path" is from  $A_0$  or  $B_0$  through the circuit to  $S_3$ .



# 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.
- These two signal conditions are called *generate*, denoted as  $G_i$ , and *propagate*, denoted as  $P_i$  respectively and are identified in the circuit:



# 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  $C_0$ :

$$C_1 = G_0 + P_0 C_0$$

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

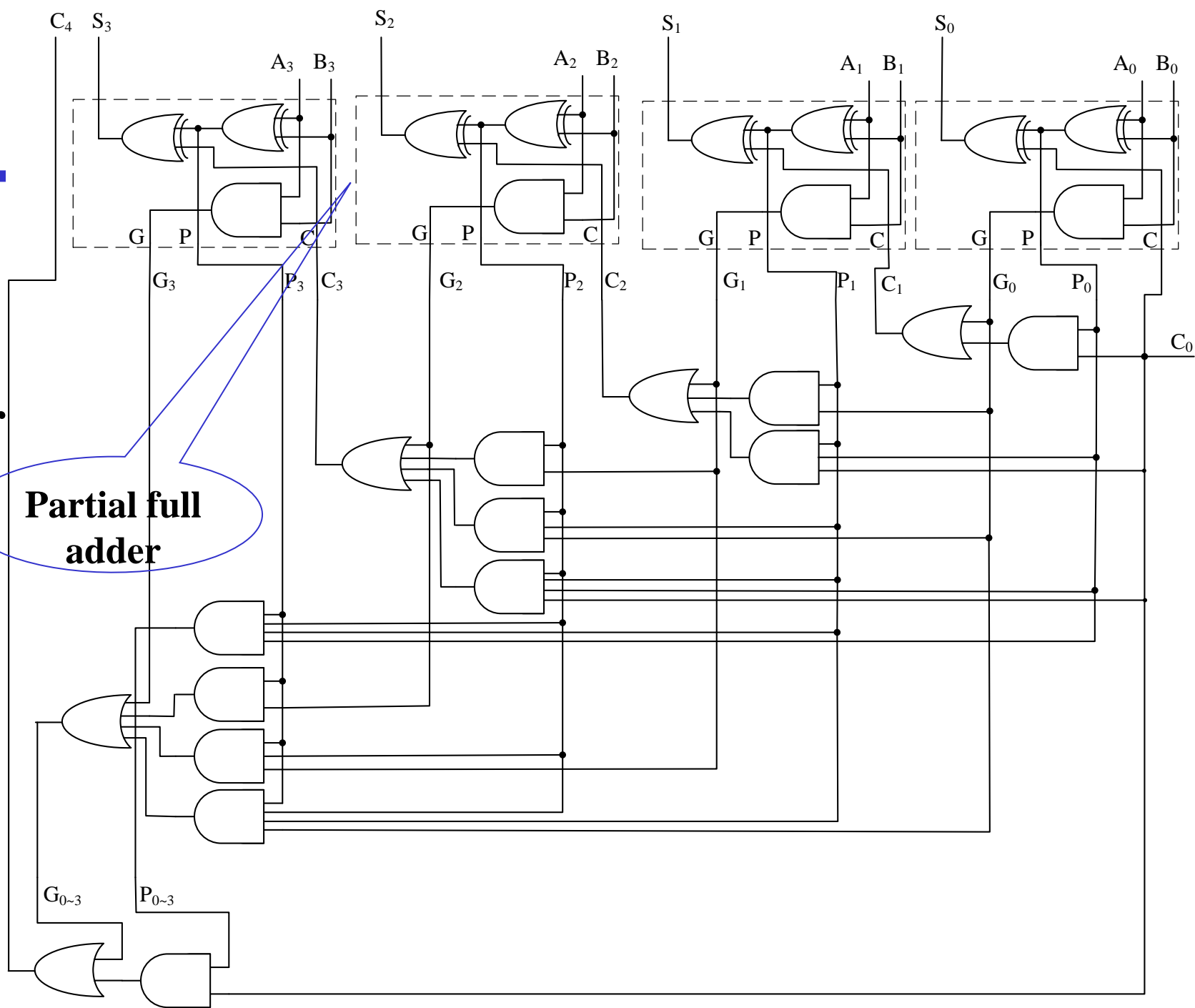
$$\begin{aligned} C_3 &= G_2 + P_2 C_2 = G_2 + P_2(G_1 + P_1 G_0 + P_1 P_0 C_0) \\ &= \mathbf{G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0} \end{aligned}$$

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

- By using these formulas, time complexity of the adders can be improved from  $O(n)$  to  $O(1)$ .

# Carry Look-ahead Adder

Partial full adder



# Group Carry Lookahead Logic

---

- Next slide 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.
- $C_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$
- Instead, the concept is extended another level by considering *group generate* ( $G_{0-3}$ ) and *group propagate* ( $P_{0-3}$ ) functions:

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

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

- 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

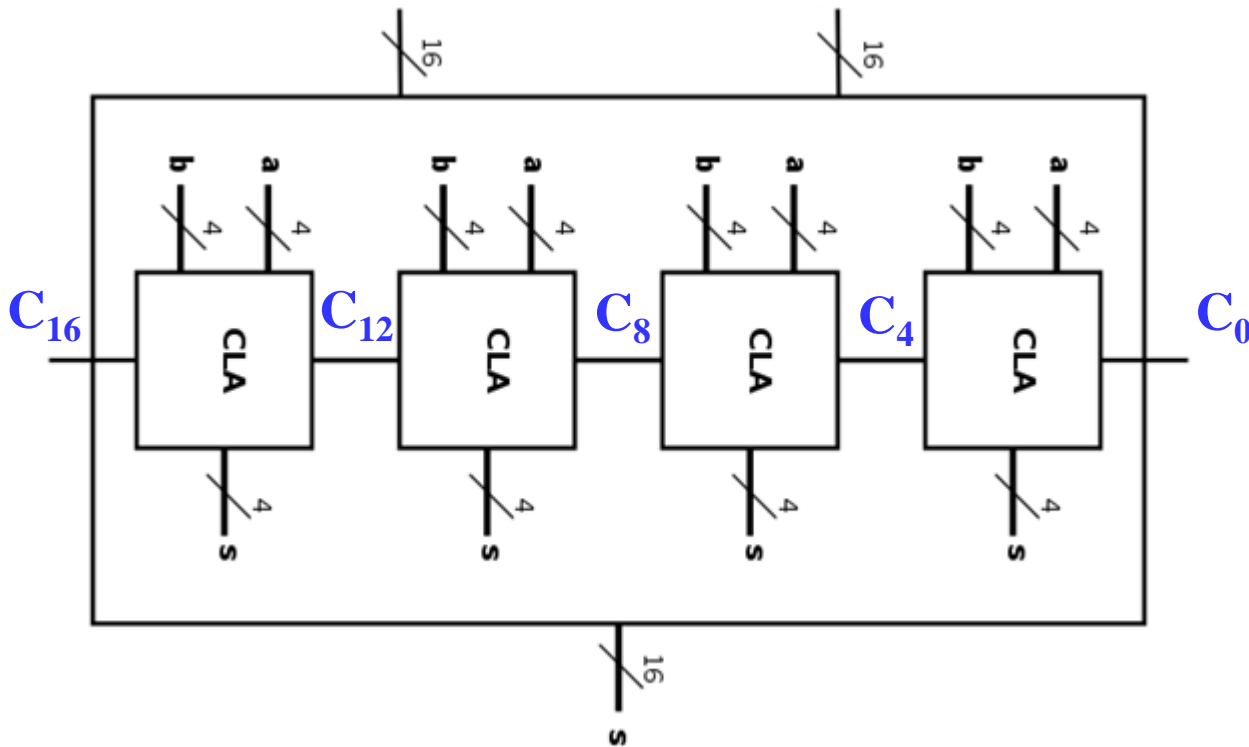
# Group Carry Lookahead Logic (Cont.)

---

- $C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 = G_{0\sim3} + P_{0\sim3}C_0$
- $C_8 = G_7 + P_7G_6 + P_7P_6G_5 + P_7P_6P_5G_4 + P_7P_6P_5P_4C_4 = G_{4\sim7} + P_{4\sim7}C_4$
- $C_{12} = G_{11} + P_{11}G_{10} + P_{11}P_{10}G_9 + P_{11}P_{10}P_9G_8 + P_{11}P_{10}P_9P_8C_8 = G_{8\sim11} + P_{8\sim11}C_8$
- $C_{16} = G_{15} + P_{15}G_{14} + P_{15}P_{14}G_{13} + P_{15}P_{14}P_{13}G_{12} + P_{15}P_{14}P_{13}P_{12}C_{12} = G_{12\sim15} + P_{12\sim15}C_{12}$

# Group Carry Lookahead Logic (Cont.)

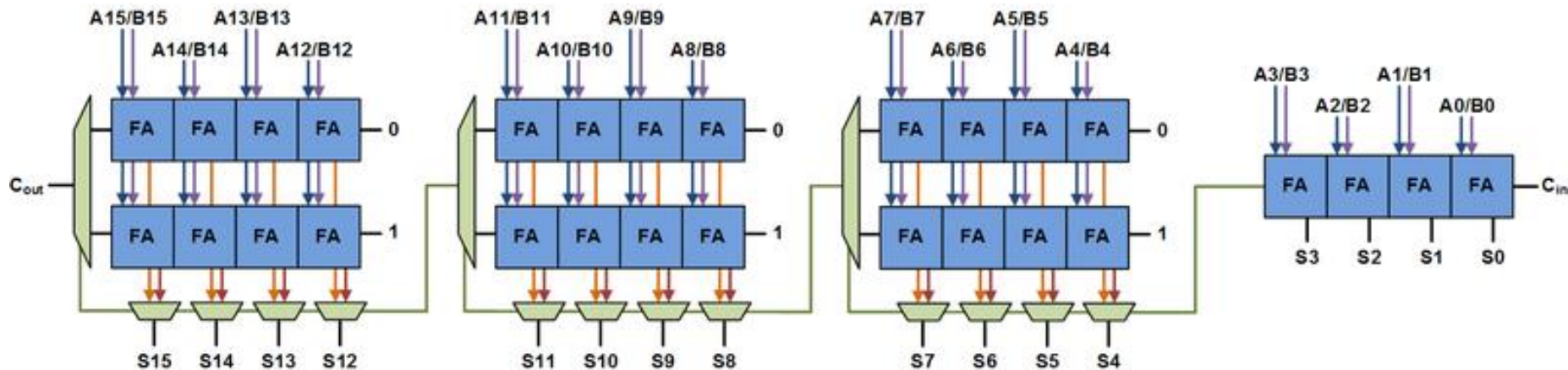
- An example of a 16-bit CLA
  - put four CLAs together in a ripple-carry manner to get a hybrid 16-bit adder



- Appendix A: Fast Carry Lookahead

# Space and Time Tradeoffs for Adder

- Several tradeoffs have been proposed between time complexity and space complexity.
- Carry-select adder
  - time complexity:  $O(\sqrt{n})$



A 16-bit carry-select adder

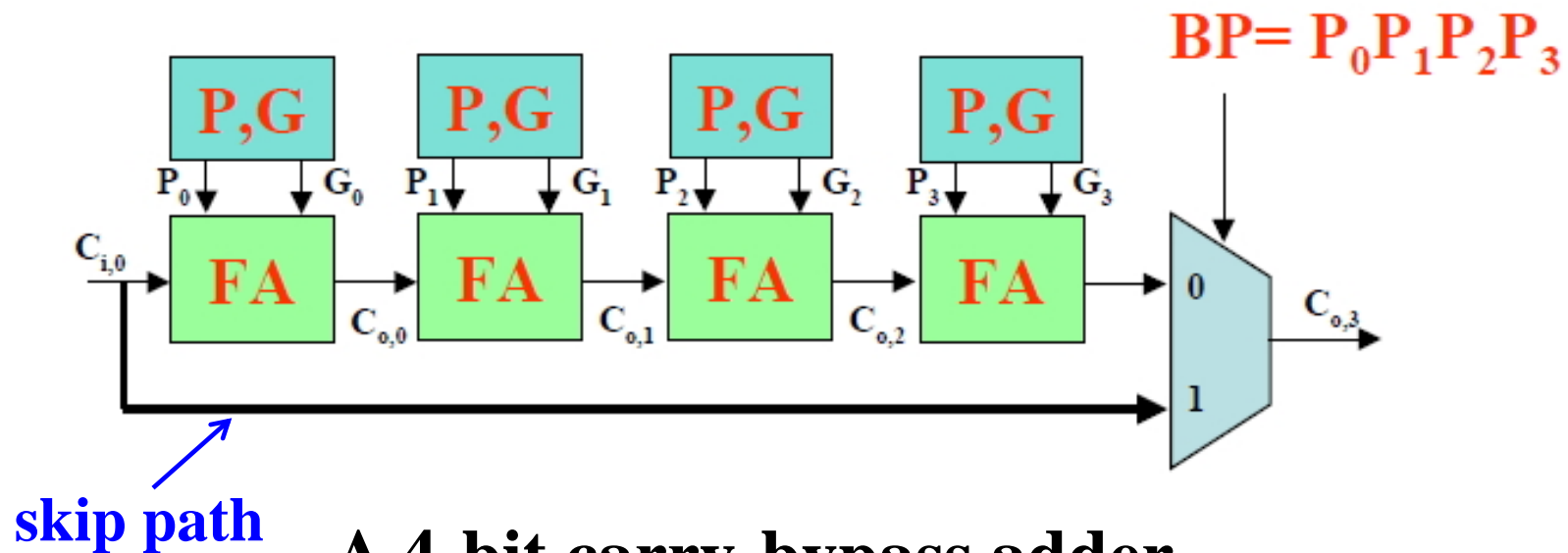


# Space and Time Tradeoffs for Adder (continued)

- Carry-skip / Carry-bypass adder

$$C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + \mathbf{P_3P_2P_1P_0} C_0$$


- Key idea: **if  $(P_3P_2P_1P_0)$  then  $C_4 = C_0$**



# 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  $M - N + 2^n$  is subtracted from  $2^n$ , the result must be **negative**, and so we need to **correct** its magnitude.

## ■ Examples:

0	1
1001	0100
– 0111	– 0111
0010	1101

# Unsigned Subtraction (continued)

---

## ■ Algorithm:

- subtracting the preceding formula from  $2^n$ :

$$2^n - (M - N + 2^n) = N - M$$

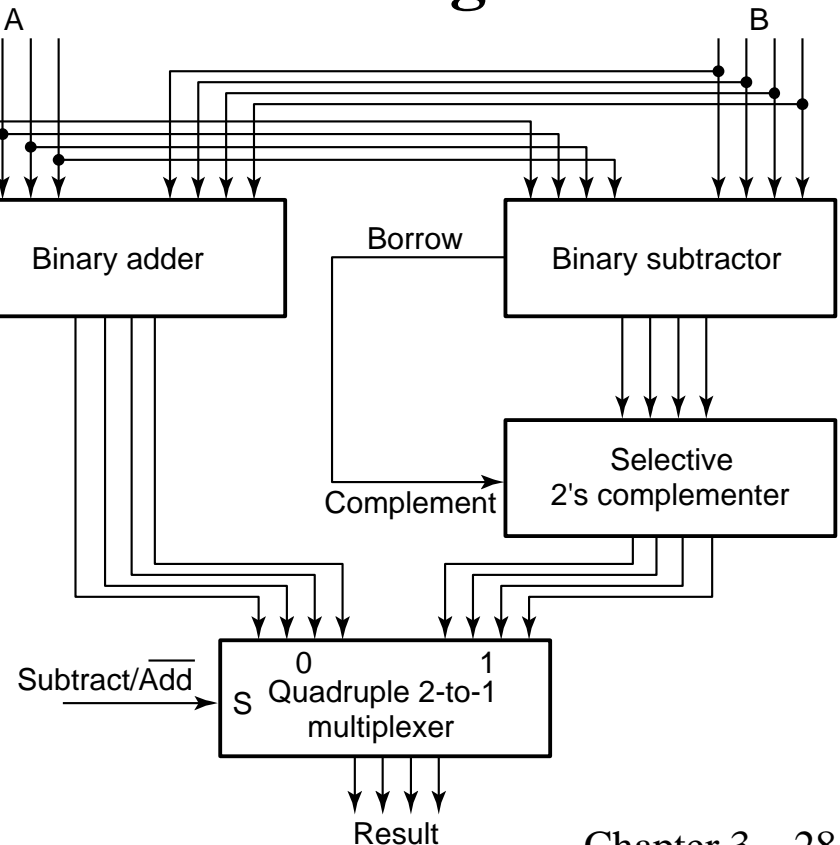
- a **minus sign** should be appended to the result.

## ■ 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:**
  - **Radix Complement**
    - $r$ 's complement for radix  $r$
    - 2's complement in binary
    - Defined as  $r^n - N$
  - **Diminished Radix Complement of  $N$** 
    - $(r - 1)$ 's complement for radix  $r$
    - 1's complement for radix 2
    - Defined as  $(r^n - 1) - N$
- **Subtraction is done by adding the complement of the subtrahend**
- **If the result is negative, takes its 2's complement**

# Binary 1's Complement(反码)

---

- 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:

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

- 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(补码)

---

- 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:

$$\begin{array}{r} 100000000 \\ - 01110011 \\ \hline 10001101 \end{array}$$

- 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

$$\begin{array}{r} \text{P=1} \quad \text{G=1} \\ \text{1} \quad \text{01101100} \\ \uparrow \quad \downarrow \\ + \quad \underline{10010100} \\ \hline 100000000 \end{array}$$



# 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  $01010100_2 - 01000011_2$

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

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

## Unsigned 2's Complement Subtraction Example 2

---

- Find  $01000011_2 - 01010100_2$

$$\begin{array}{r} 01000011 \\ - 01010100 \\ \hline \end{array} \xrightarrow{\text{2's comp}} \begin{array}{r} \overset{0}{0}1000011 \\ + 10101100 \\ \hline 11101111 \\ \hline \end{array} \xrightarrow{\text{2's comp}} 00010001$$

- 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:

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

where:

$s = 0$  for Positive numbers

$s = 1$  for Negative numbers

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

# Signed Integers (continued)

---

**The leftmost bit represents the sign in machine number**

**Binary Code**

**Machine number**

Example: +1011 →

sign	number value			
0	1	0	1	1

-1011 →

sign	number value			
1	1	0	1	1

# 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

- $r = 2, n = 3$

$$S2C(X) = -w_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} w_i \cdot 2^i$$



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:  $2^n - (N - M)$ 
    - 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:**     $\begin{array}{r} 0010 \\ + 0101 \\ \hline \end{array}$      $\begin{array}{r} 010 \\ + 101 \\ \hline \end{array}$   
parity of the signs is 0     $\rightarrow$      $\begin{array}{r} 111 \end{array}$      $\rightarrow$      $\boxed{0}111$
- **Example 2:**     $\begin{array}{r} 0010 \\ + 1101 \\ \hline \end{array}$      $\begin{array}{r} 010 \\ - 101 \\ \hline \end{array}$   
parity of the signs is 1     $\rightarrow$      $\boxed{1}101$      $\rightarrow$      $\boxed{1}011$
- **Example 3:**     $\begin{array}{r} 1010 \\ - 0101 \\ \hline \end{array}$      $\begin{array}{r} 010 \\ + 101 \\ \hline \end{array}$   
parity of the signs is 0     $\rightarrow$      $\begin{array}{r} 111 \end{array}$      $\rightarrow$      $\boxed{1}111$

# 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:

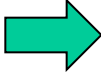
Form the complement of the number you are subtracting and **follow the rules for addition**.

# Signed 2's Complement Examples

---

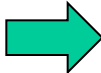
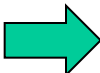
- **Example 1: 1101**

**+0011**

**10000     0000**

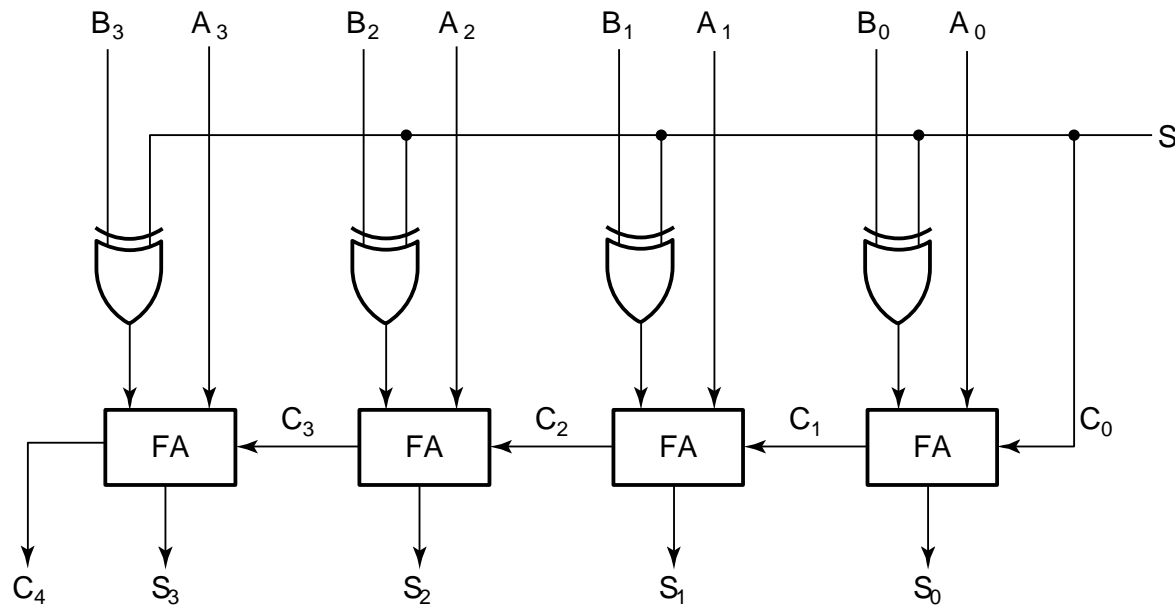
- **Example 2: 1101**

**-0011**

**1101  
+ 1101  
 11010     1010**

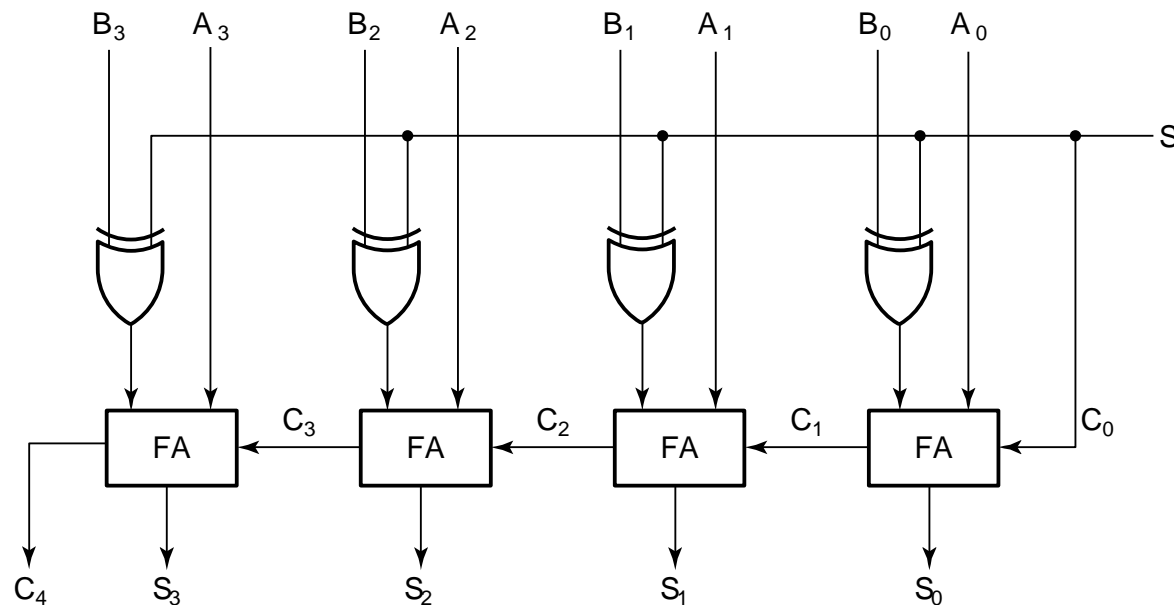
# 2's Complement Adder/Subtractor

- 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  $A - B$ :



# 2's Complement Adder/Subtractor (continued)

- For  $S = 0$ , add,  $B$  is passed through unchanged.
- For  $S = 1$ , subtract, the 2's complement of  $B$  is formed by using XORs to form the 1's complement and adding the 1 applied to  $C_0$ .



# 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
- **Examples:**

		sign bit	
Carries:	01	Carries:	10
+70	01000110	-70	10111010
+80	01010000	-80	10110000
<hr/>		<hr/>	
+150	10010110	-150	01101010
∨		∧	
<b>+127</b>		<b>-128</b>	

# Overflow Detection (continued)

- If we add two k bit numbers:  $X_{k-1}...X_0$  and  $Y_{k-1}...Y_0$ . The sum is  $S_{k-1}...S_0$ . One formula for detecting overflow is:

$$V = X_{k-1}Y_{k-1}\bar{S}_{k-1} + \bar{X}_{k-1}\bar{Y}_{k-1}S_{k-1}$$

- A Simpler Formula for Overflow

$$V = C_{k-1} \oplus C_{k-2}$$

- Case 1: 0 carried in, and 1 carried out of the leftmost full adder
- Case 2: 1 carried in, and 0 carried out of the leftmost full adder

Carries: 10

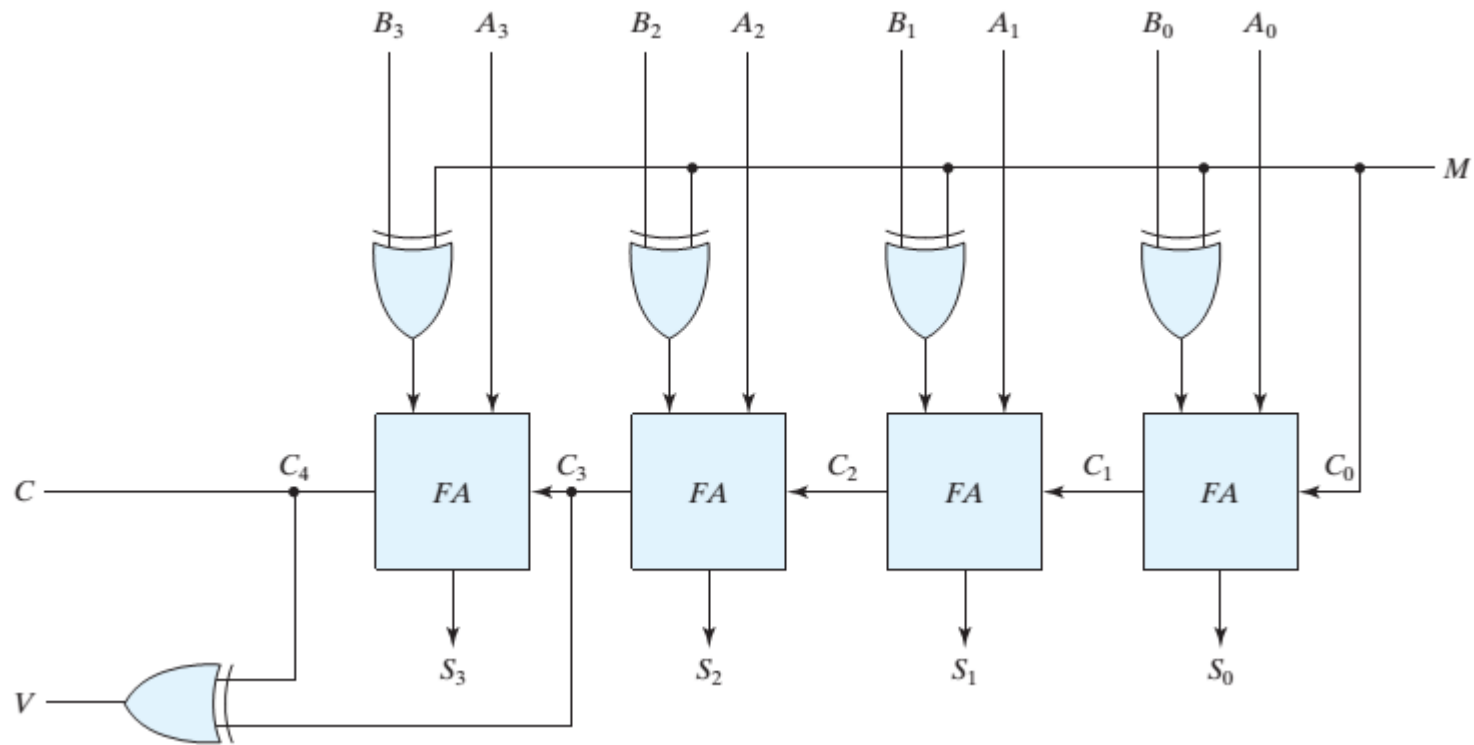
-70	10111010
-80	10110000
<u>-150</u>	<u>01101010</u>

Carries: 01

+70	01000110
+80	01010000
<u>+150</u>	<u>10010110</u>

# Overflow Detection (continued)

- **2's Complement Adder/subtractor with overflow detection:**





# Other Arithmetic Functions

---

- Other arithmetic functions beyond  $+$ ,  $-$ ,  $*$  and  $/$ , are quite important. Among these are incrementing, decrementing, multiplication and division by a constant, etc.
- Each can be implemented for multiple-bit operands by using an **iterative array** of 1-bit cells.
- Instead of using these basic approaches, a combination of rudimentary functions and a new technique called **contraction** is used.

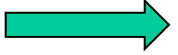
# Design by Contraction

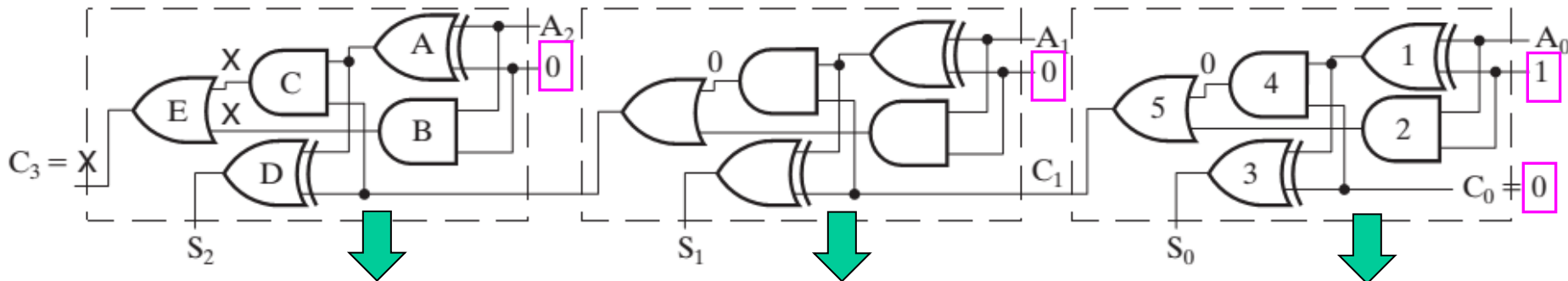
---

- The goal of contraction is to accomplish the design of a logic circuit or functional block by **using results from past designs.**
- We can implement new functions by using similar techniques on a given circuit and then **contracting it for a specific application to a simpler circuit.**
- Contraction can be applied to simplify an initial circuit with **value fixing, transferring, and inverting on its inputs** in order to obtain a target circuit.

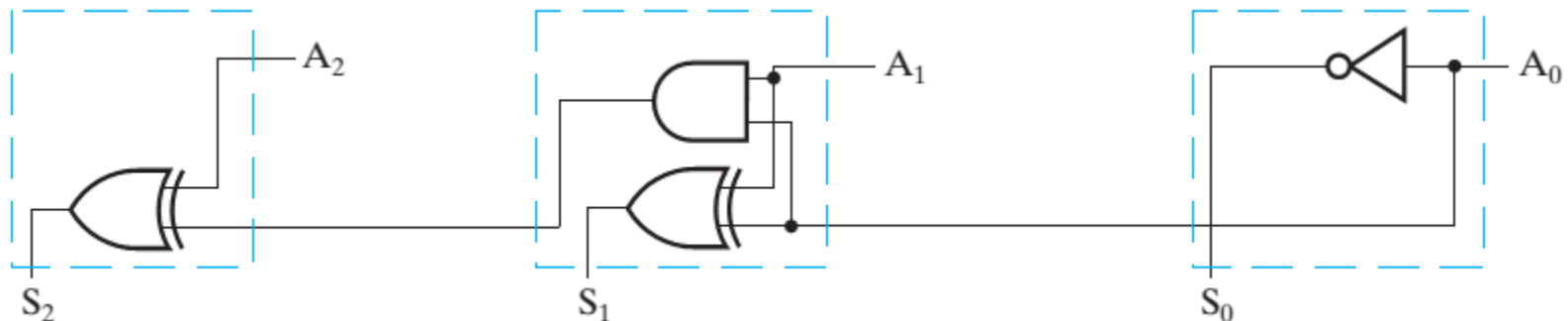
# Design by Contraction Example

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

1. set  $B = 001$   **value fixing**



2. simplifying the logic  **contracting**



# Incrementing & Decrementing

---

## ■ Incrementing

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

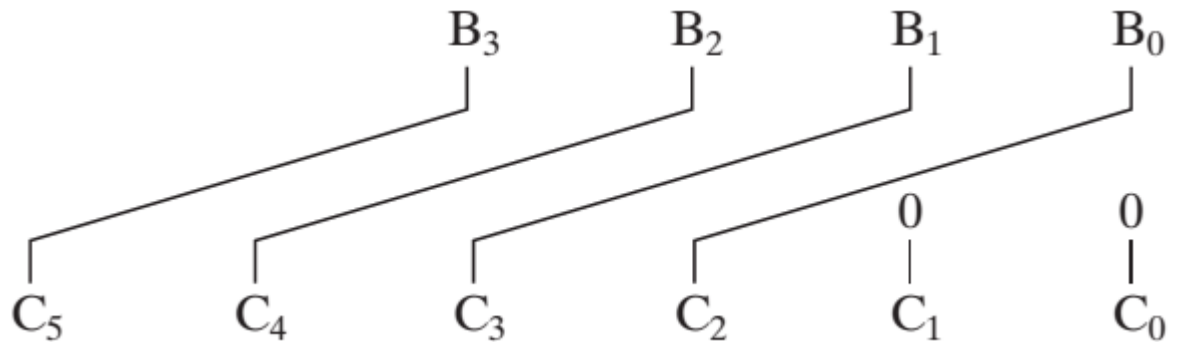
## ■ Decrementing

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

# Multiplication/Division by $2^n$

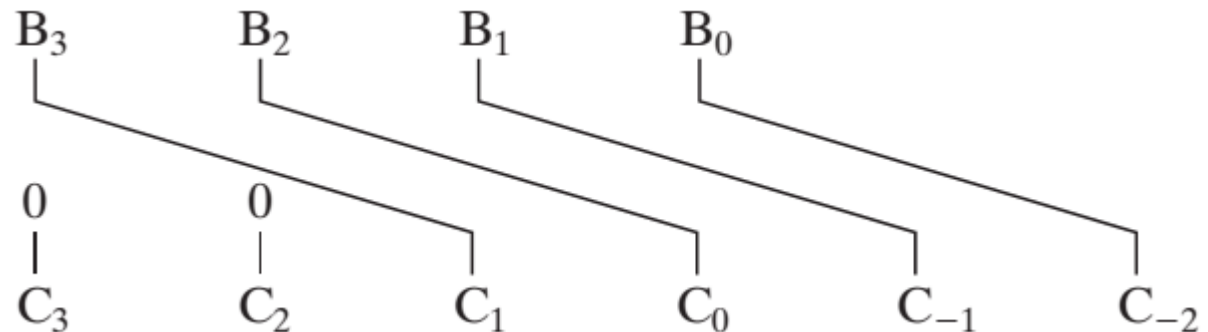
## ■ Multiplication

- Multiplication by 100
- Shift left by 2



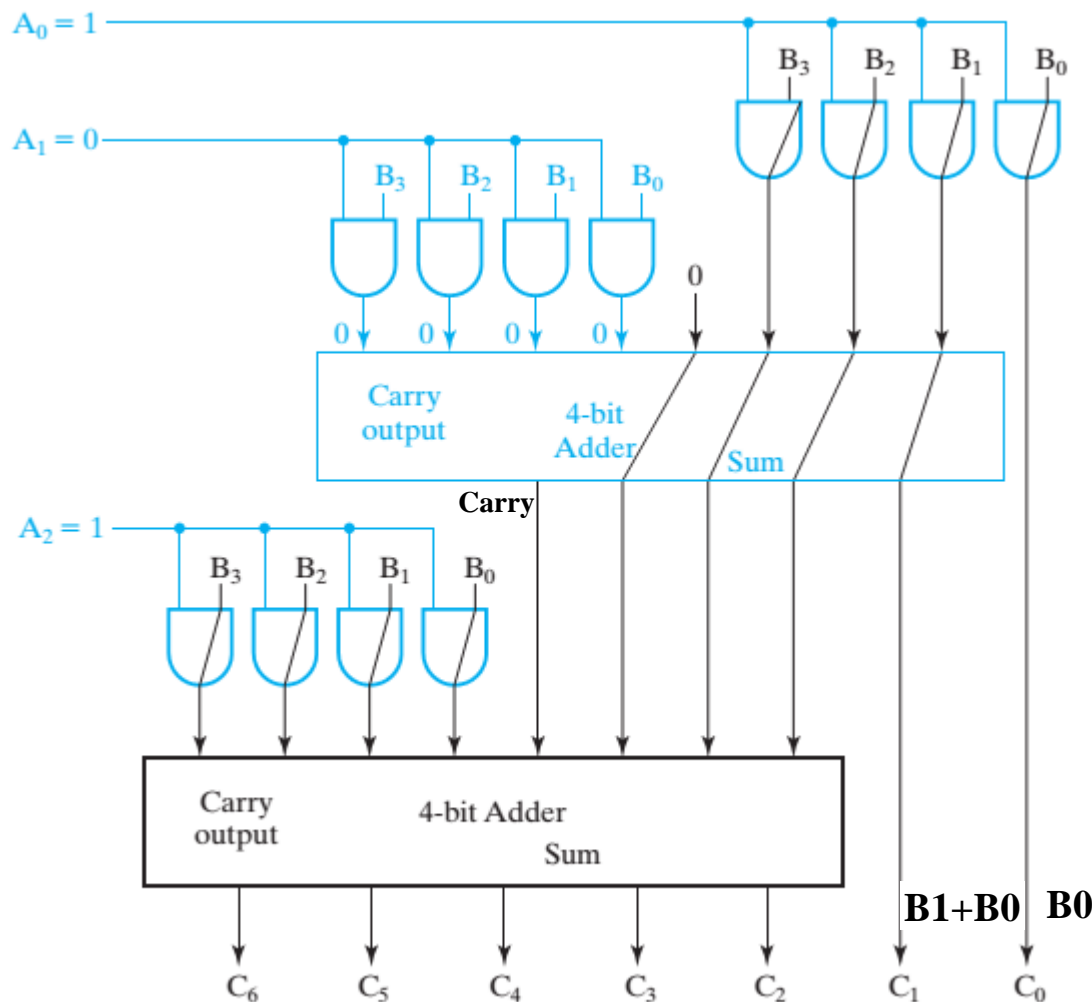
## ■ Division

- Division by 100
- Shift right by 2
- Remainder preserved



# Multiplication by a Constant

## ■ Multiplication of B(3:0) by 101

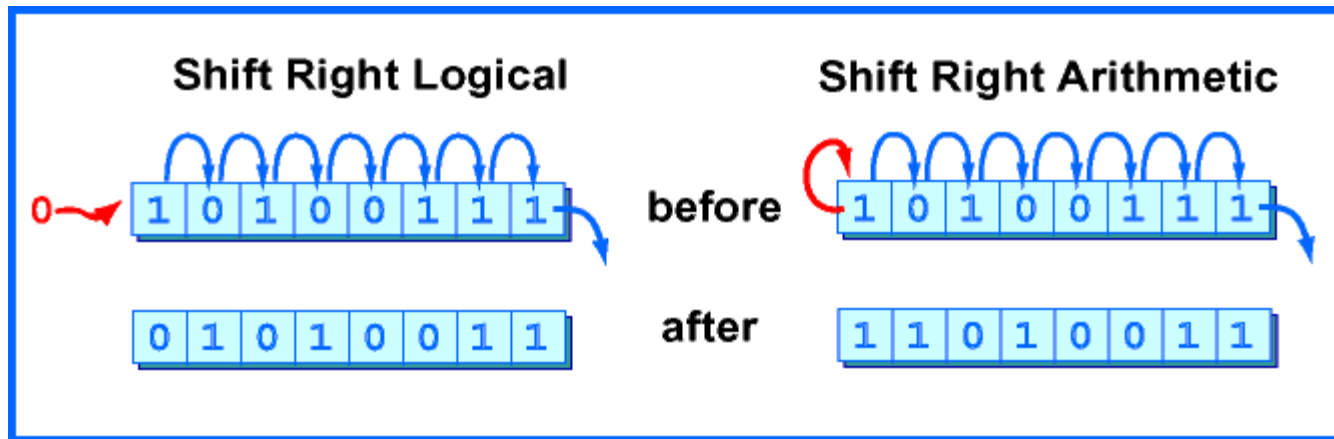


$$\begin{array}{r}
 \begin{array}{cccccc}
 0 & B_3 & B_2 & B_1 & B_0 & \\
 + & B_3 & B_2 & B_1 & B_0 & \\
 + & B_3 & B_2 & B_1 & B_0 & \\
 \hline
 C_6 & C_5 & C_4 & C_3 & C_2 & C_1 & C_0
 \end{array}
 \end{array}$$

$B_1+B_0 \quad B_0$

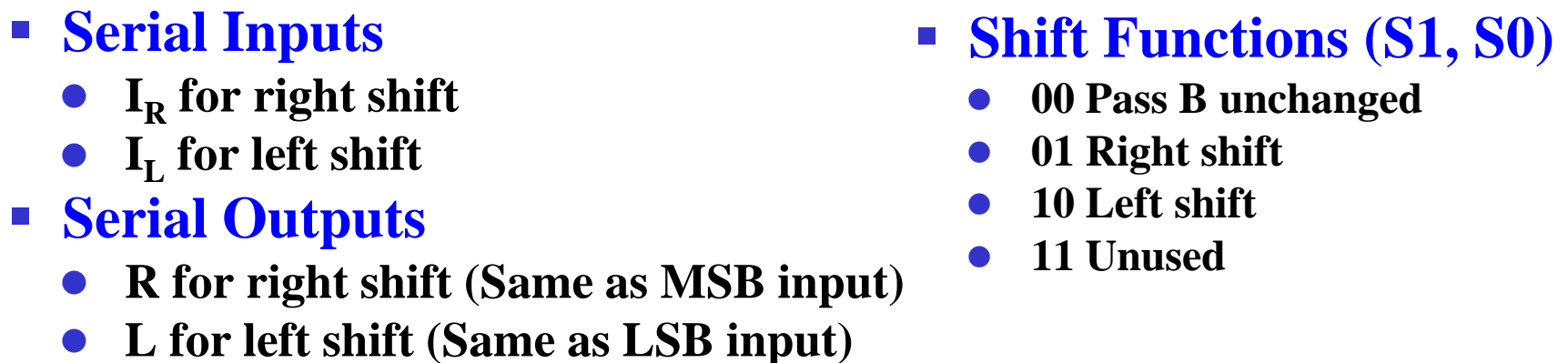
# Combinational Shifter Parameters

- **Direction**
  - Left, Right
- **Number of positions**
  - Single bit, Multiple bit
- **Operation**
  - Logic shift, Arithmetic shift, Rotate/barrel shift



- **Filling of vacant positions**
  - Zero fill, Extension

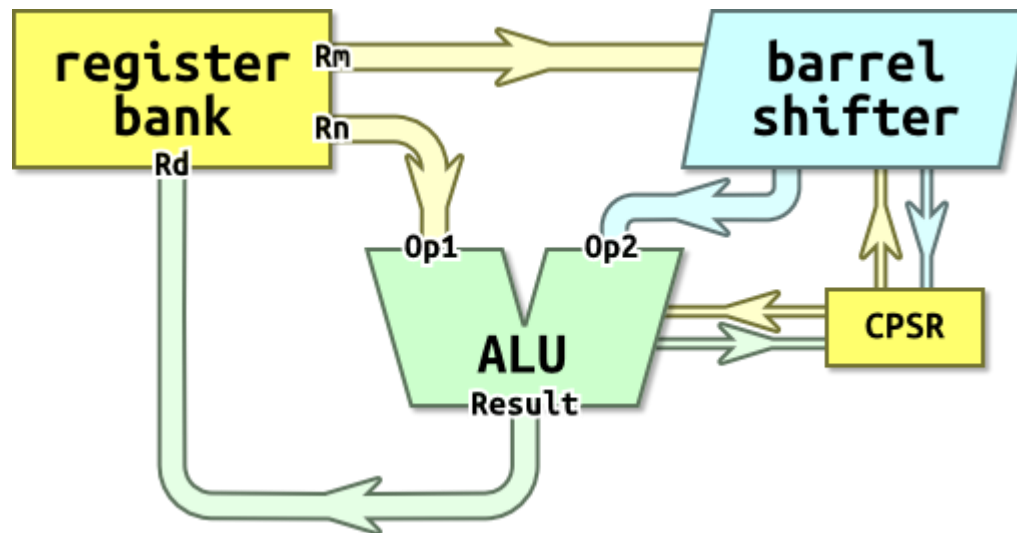
\_\_\_\_\_



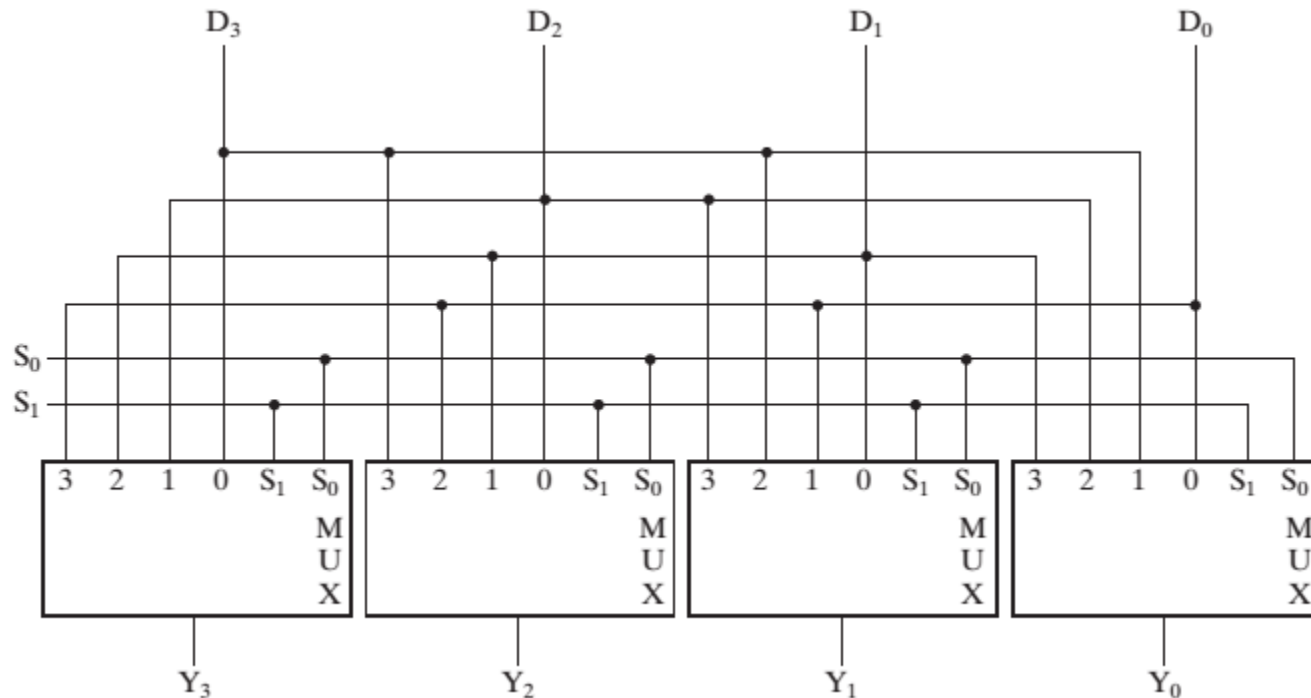


# Barrel Shifter

- A **barrel shifter** is simply a **bit-rotating** shift register. The bits shifted out the MSB end of the register are shifted back into the LSB end of the register.
- In a barrel shifter, the bits are shifted the desired number of bit positions in a **single clock cycle**.



# Barrel Shifter (continued)



- The circuit rotates its contents left from 0 to 3 positions depending on S:

- $S = 00$  position unchanged
- $S = 01$  rotate left by 1 position
- $S = 10$  rotate left by 2 positions
- $S = 11$  rotate left by 3 positions

Appendix B: How to  
construct a **larger shifter**?

# 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

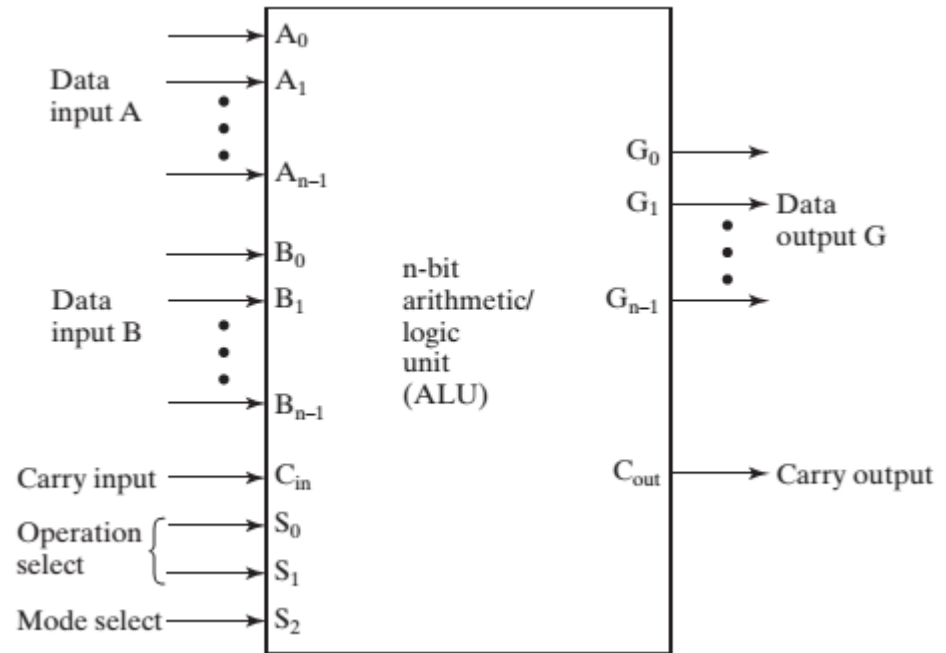
# Extension

---

- **Extension** - increase in the number of bits at the **MSB end** of an operand by using a **complement representation**
- **Examples**
  - 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)

- ALU performs integer arithmetic and logical operations.
- Idea of building arithmetic circuit: use adders and **control some of the inputs.**



Implementation

A block of logic that selects four choices for the B input to the adder

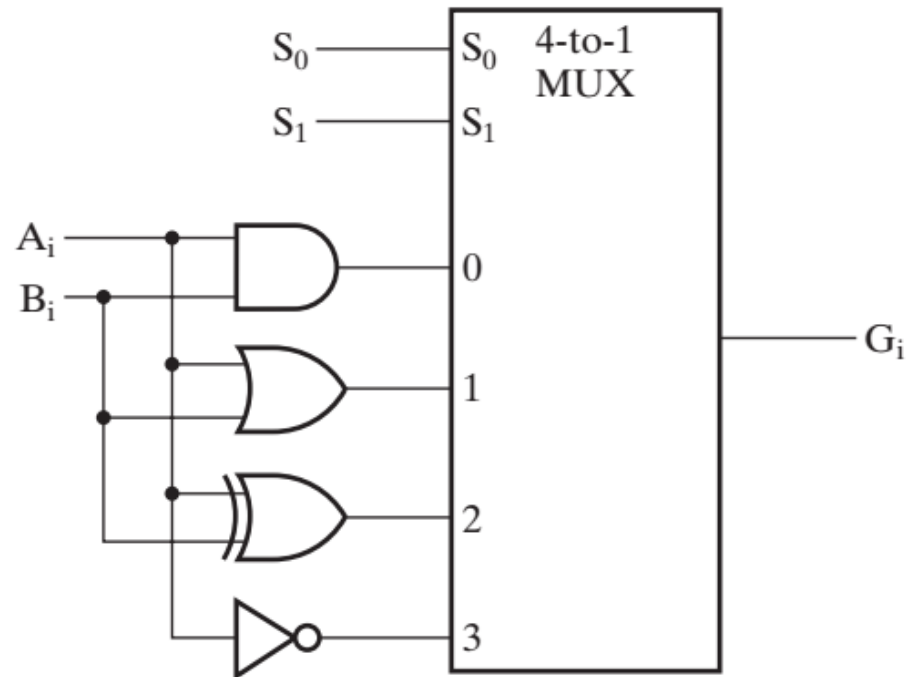
Change generate function  $G_i$  and propagate function  $P_i$

# Logic Circuit Design

- Each of logic operations can be generated through a gate that performs the required logic.

$S_1$	$S_0$	Output	Operation
0	0	$G = A \wedge B$	AND
0	1	$G = A \vee B$	OR
1	0	$G = A \oplus B$	XOR
1	1	$G = \overline{A}$	NOT

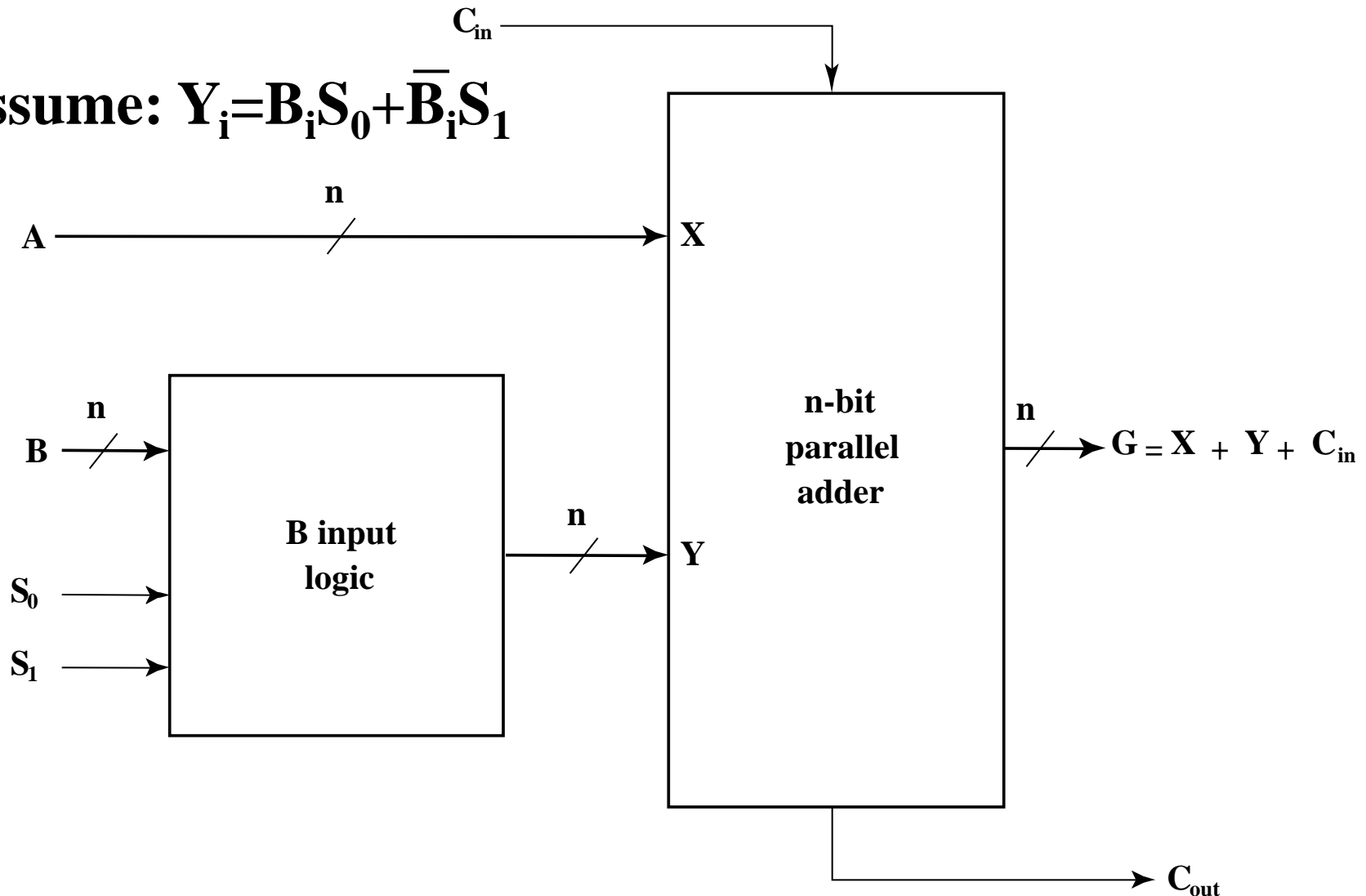
**Function table for logic operation**



**Logic operation circuit**

# Arithmetic Circuit Design

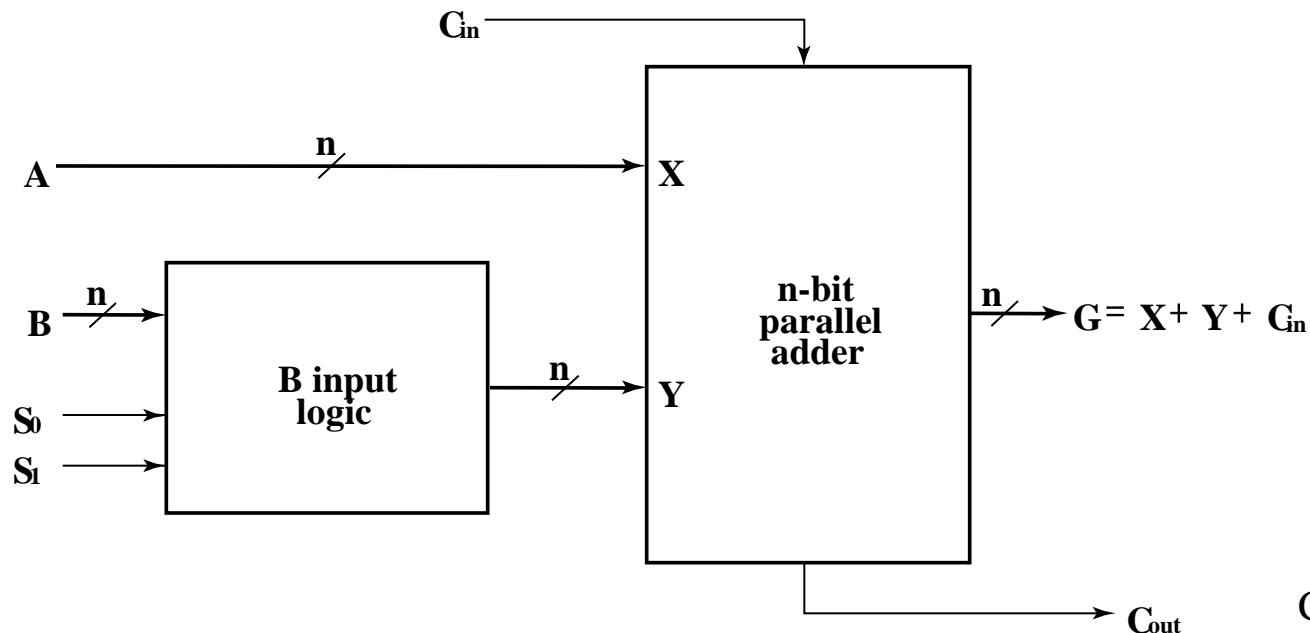
Assume:  $Y_i = B_i S_0 + \bar{B}_i S_1$



# Arithmetic Circuit Design (continued)

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

	$C_{in} = 0$	$C_{in} = 1$
$Y$ <ul style="list-style-type: none"> <li>• 0</li> <li>• <math>B</math></li> <li>• <math>\overline{B}</math></li> <li>• 1</li> </ul>	$G = A$	$G = A + 1$
	$G = A + B$	$G = A + B + 1$
	$G = A + \overline{B}$	$G = A + \overline{B} + 1$
	$G = A - 1$	$G = A$

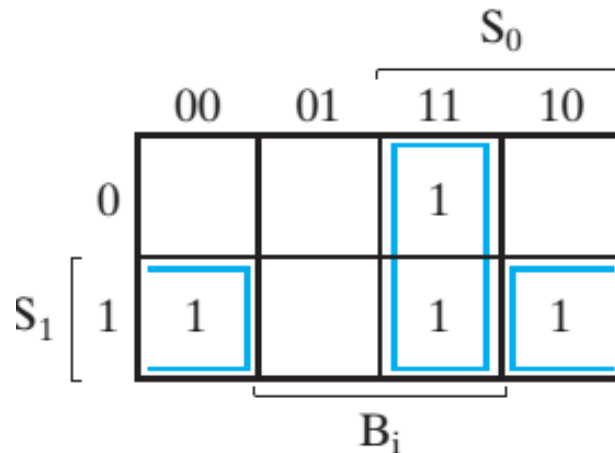




# Arithmetic Circuit Design (continued)

- Multiplexing
- Redesign the input of B

Inputs			Output
$S_1$	$S_0$	$B_i$	$Y_i$
0	0	0	0 $Y_i = 0$
0	0	1	0
0	1	0	0 $Y_i = B_i$
0	1	1	1
1	0	0	1 $Y_i = \overline{B_i}$
1	0	1	0
1	1	0	1 $Y_i = 1$
1	1	1	1



$$Y_i = B_i S_0 + \overline{B_i} S_1$$

# Arithmetic Circuit Design (continued)

- 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

# Assignment

---

## Reading:

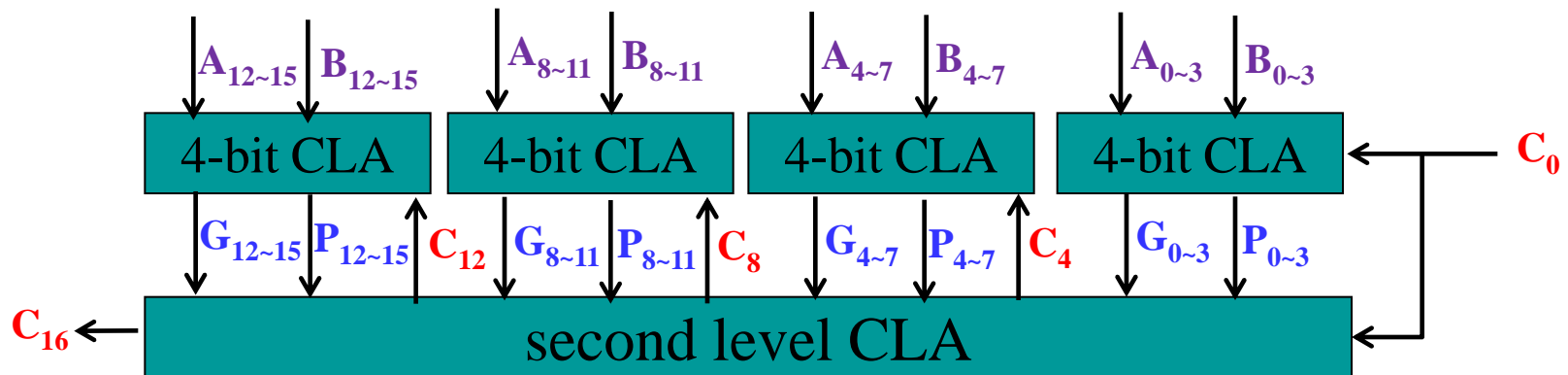
■ 3.8-3.12

## Problem assignment:

■ 3-50; 3-51; 3-52; 3-59

# Appendix A: Fast Carry Using the Second Level of Abstraction

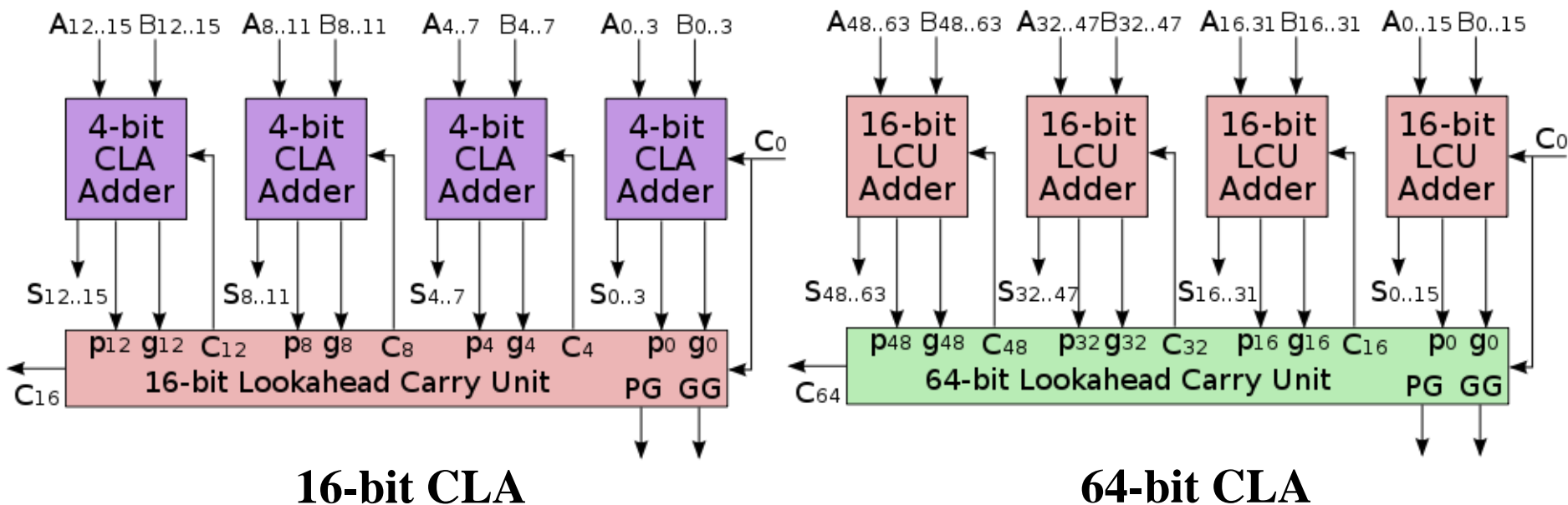
- $C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 = G_{0\sim3} + P_{0\sim3}C_0$
- $C_8 = G_7 + P_7G_6 + P_7P_6G_5 + P_7P_6P_5G_4 + P_7P_6P_5P_4C_4 = G_{4\sim7} + P_{4\sim7}C_4$
- $C_{12} = G_{11} + P_{11}G_{10} + P_{11}P_{10}G_9 + P_{11}P_{10}P_9G_8 + P_{11}P_{10}P_9P_8C_8 = G_{8\sim11} + P_{8\sim11}C_8$
- $C_{16} = G_{15} + P_{15}G_{14} + P_{15}P_{14}G_{13} + P_{15}P_{14}P_{13}G_{12} + P_{15}P_{14}P_{13}P_{12}C_{12} = G_{12\sim15} + P_{12\sim15}C_{12} = G_{12\sim15} + P_{12\sim15}(G_{8\sim11} + P_{8\sim11}(G_{4\sim7} + P_{4\sim7}(G_{0\sim3} + P_{0\sim3}C_0))) = G_{12\sim15} + P_{12\sim15}G_{8\sim11} + P_{12\sim15}P_{8\sim11}G_{4\sim7} + P_{12\sim15}P_{8\sim11}P_{4\sim7}G_{0\sim3} + P_{12\sim15}P_{8\sim11}P_{4\sim7}P_{0\sim3}C_0$



Note: Carry-in is generated by second level CLA, not individual adders!

# Fast Carry Lookahead Example

- We can **cascade** the CLA to form a larger CLA. This larger block can then be cascaded into a larger CLA using the same **2-level CLA** method.
- Examples of cascading adders



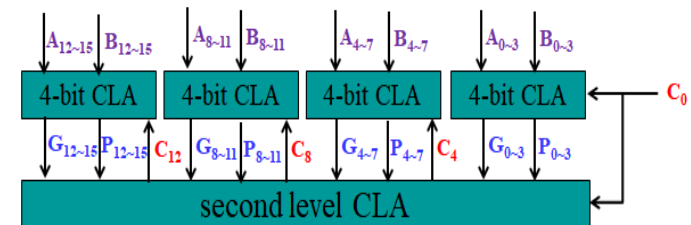
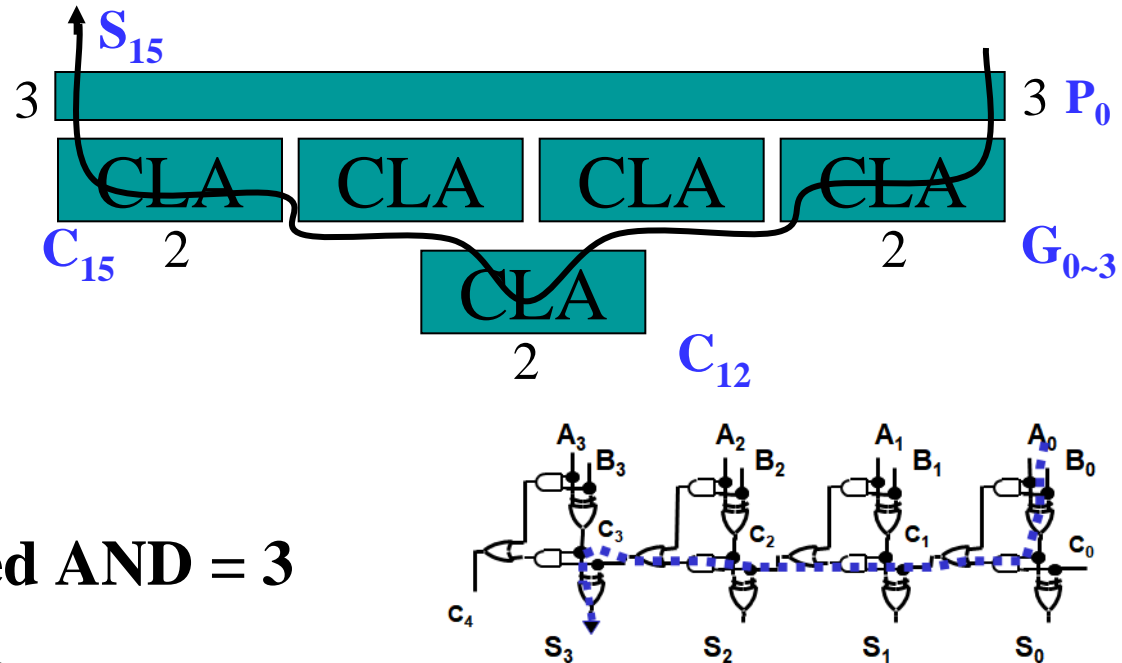
# Carry Lookahead Example

## ■ Specifications:

- 16-bit CLA
- Delays:
  - NOT = 1
  - AND-OR = 2
  - XOR = Isolated AND = 3

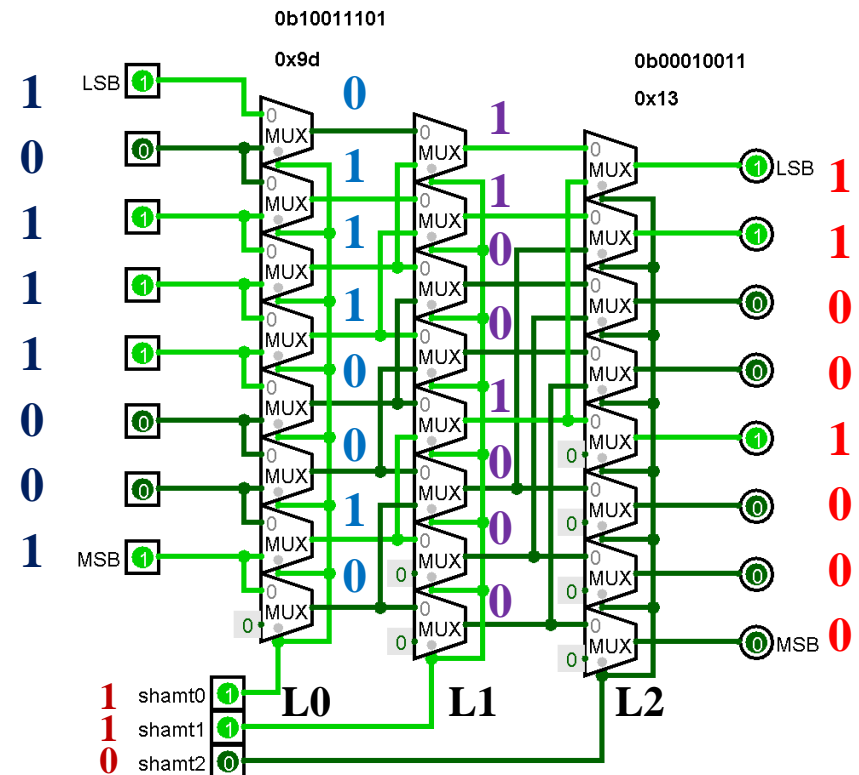
## ■ Longest Delays:

- Ripple carry adder\* =  $3 + 15 \times 2 + 3 = 36$ 
  - time complexity:  $O(n)$
- CLA =  $3 + 3 \times 2 + 3 = 12$ 
  - time complexity:  $O(\log n)$



# Appendix B: Larger Shifter

- Large shifters can be constructed by using **layers of multiplexers**
  - Implementing 8-bit shifter by using  $2 \times 1$  multiplexers
    - Layer 0 shifts by 0, 1
    - Layer 1 shifts by 0, 2
    - Layer 2 shifts by 0, 4



# Larger Shifter (continued)

- Implementing 8-bit shifter by using  $2 \times 1$  multiplexers

