

---

# 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

# Arithmetic Calculations

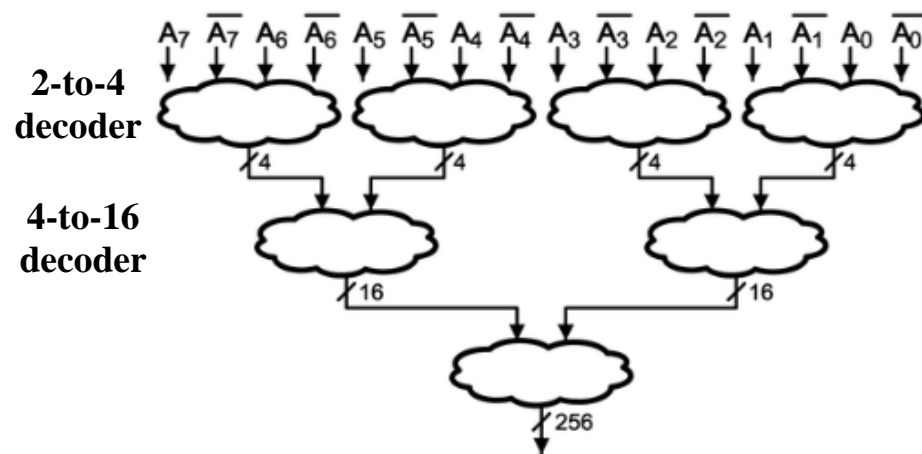
- In the view of logic circuit, an **arithmetic calculation** (e.g., addition, subtraction) is a **code translation decoder**. For example, an N-bit binary addition requires a 2N-to-N+1 decoder.
- However, when the operands become larger, it is a challenging task to build a code translation decoder.
- Example: addition of two 12-bit binary integers
  - Number of inputs =  $12 * 2$
  - Truth table rows =  $2^{24}$
  - Equations with up to **24 variables**
  - **Design impractical !**

	0	0	0	1	0	0	1	1	1	1	0	1	<i>Addend</i>	12-bit
+	0	0	0	0	1	0	1	1	0	1	1	1	<i>Augend</i>	12-bit
<hr/>														
C	0	0	0	1	1	1	1	1	0	1	0	0	<i>Sum</i>	13-bit

↑  
carry bit

# From Bisection to Iterative Array (1/2)

- A **bisection** method is used to construct a large decoder by combining outputs, but it is not effective for an arithmetic circuit.



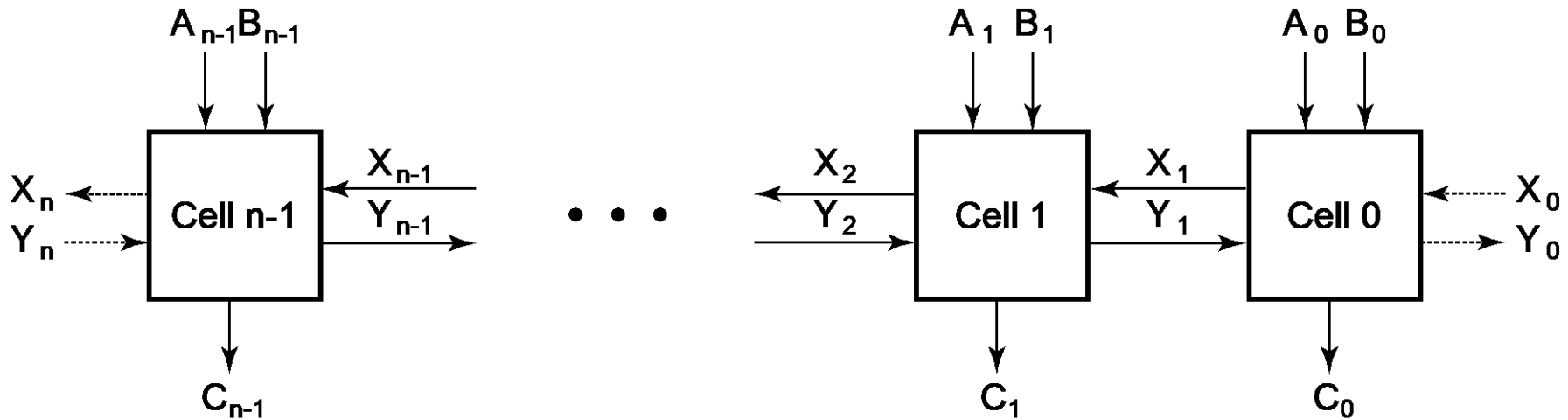
an 8-to-256 decoder

Carry in		1	0	1	0
Augend ( $A_x$ )		0	1	0	1
Addend ( $B_x$ )	+	0	1	0	1
Carry out		0	1	0	1
Result ( $R_x$ )		1	0	1	0

a 4-bit binary adder

- We should follow the **calculation rules** to abstract new functional blocks for an arithmetic function.

# From Bisection to Iterative Array (2/2)



- The functional blocks for arithmetic are referred to as **cells** and the overall implementation is an array of cells.
- The **iterative array** is a method of **calculating decomposition** and takes advantage of the regularity to make a large and multiple-level arithmetic circuit 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

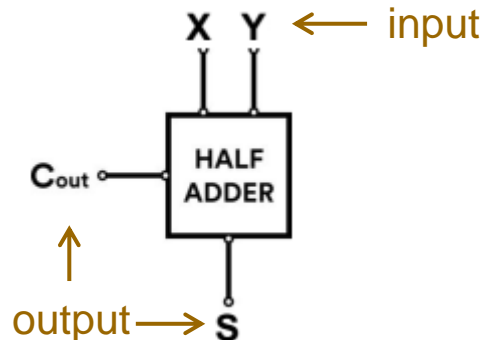
# Functional Blocks: Addition

---

- Binary addition used frequently
- Addition Development:
  - *Half-Adder* (HA), a 2-input bit-wise addition functional block (e.g.,  $X + Y = C : S$ ),
  - *Full-Adder* (FA), a 3-input bit-wise addition functional block (e.g.,  $X + Y + Z = C : S$ ),
  - *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 and produce a two-bit sum.
- In essence, **a half-adder is a 2-to-2 decoder.**
- The half adder can be specified as a truth table for S and C  $\Rightarrow$
- The sum is expressed as a sum bit, S and a carry bit, C.

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



# Logic Simplification: Half-Adder

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

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

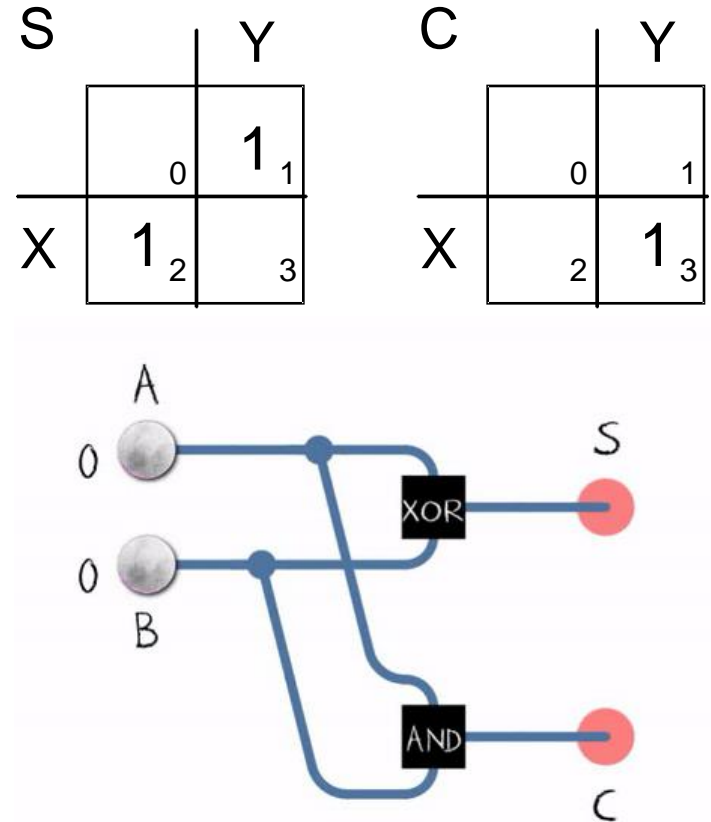
$$C = (X + Y) (\bar{X} + \bar{Y})$$

- and

$$C = X Y$$

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

- These equations lead to several implementations.



# Five Implementations: Half-Adder

---

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

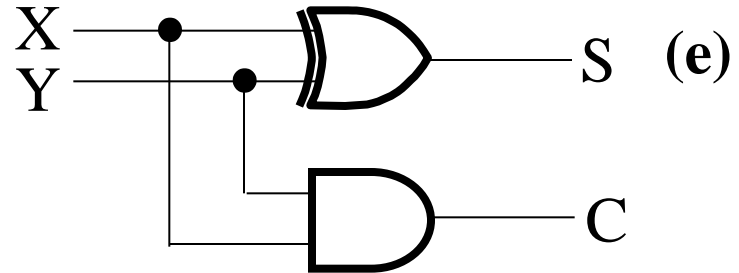
$$\begin{array}{ll}
 \text{(a)} \quad S = X \bar{Y} + \bar{X} Y & \text{(d)} \quad \underline{S} = (\underline{X} + \underline{Y}) \bar{C} \\
 \quad \quad C = X Y & \quad \quad \underline{C} = (\underline{X} + \underline{Y}) \\
 \text{(b)} \quad S = (X + Y) (\bar{X} + \bar{Y}) & \text{(e)} \quad S =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

- (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

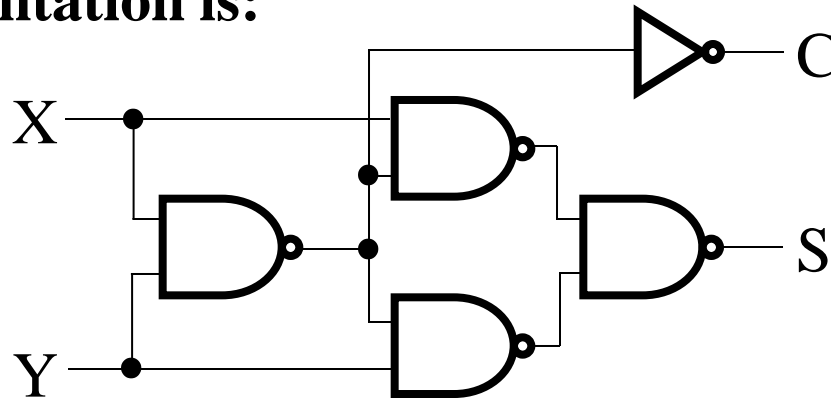
- The most common half adder implementation is:

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



- A NAND only implementation is:

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



# Functional Block: Full-Adder

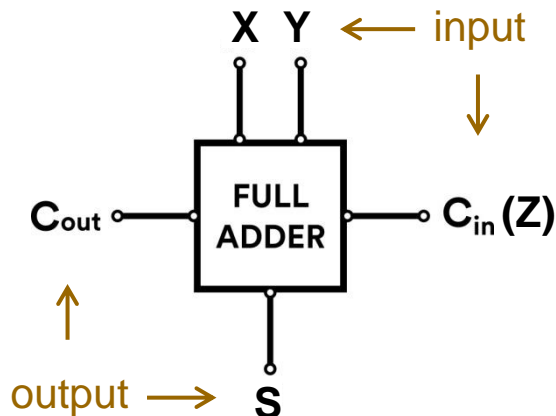
- A full adder is similar to a half adder, but includes a carry-in bit from lower stages. Like the half-adder, it computes a sum bit  $S$ , and a carry bit  $C$ .

- 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
$+ Y$	$+ 0$	$+ 1$	$+ 0$	$+ 1$
$C S$	0 0	0 1	0 1	1 0

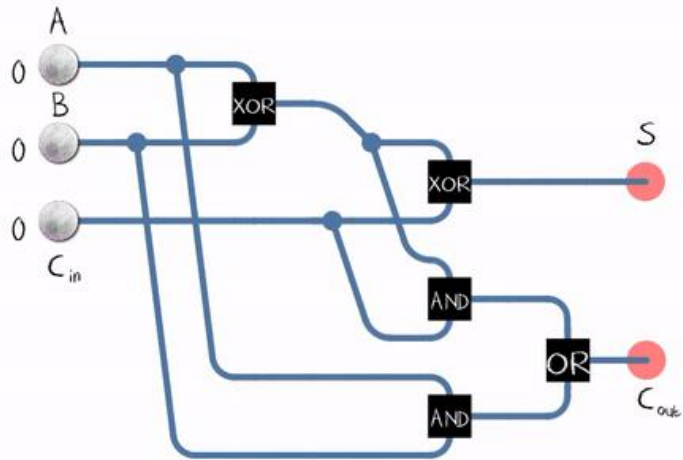
- For a carry-in ( $Z$ ) of 1:

$Z$	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	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:

**S**

	<b>Y</b>			
	0	1	3	2
<b>X</b>	1		1	
	4	5	7	6
	<b>Z</b>			

**C**

	<b>Y</b>			
	0	1	3	2
<b>X</b>		1	1	1
	4	5	7	6
	<b>Z</b>			

- In essence, **a full-adder is a 3-to-2 decoder.**

# Equations: Full-Adder

- From the K-Map, we get:

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

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

- 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 exactly one input is 1 and a carry-in (Z) occurs. Thus C can be re-written as:

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

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

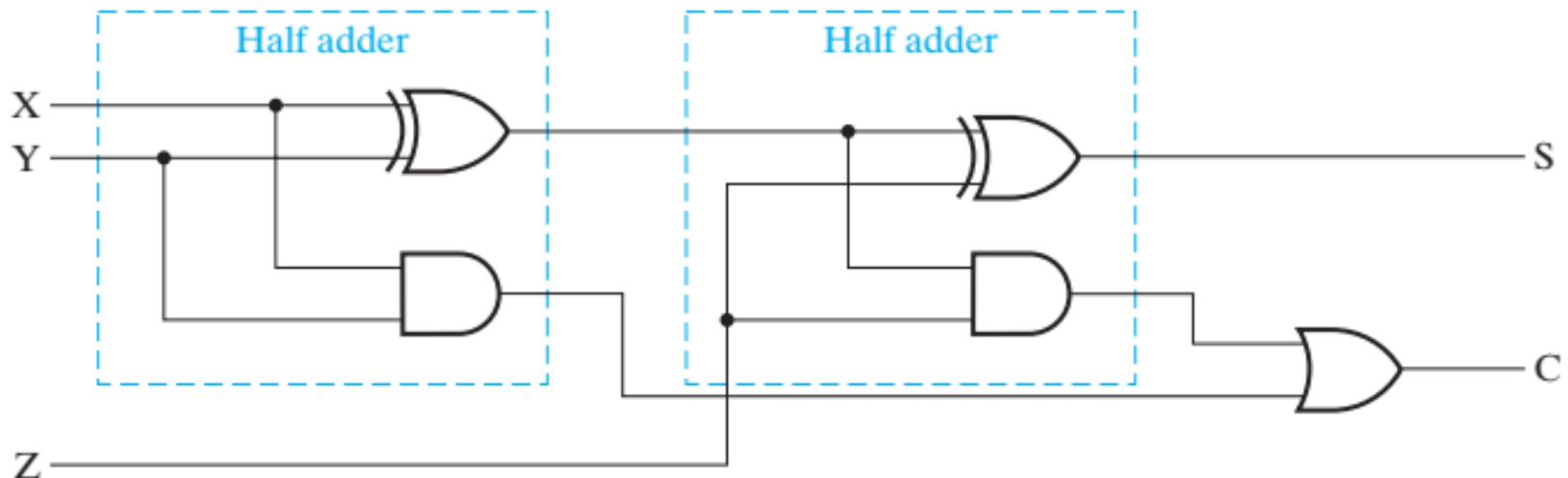
	Y			
	0	1	3	2
X	4	5	7	6
	Z			
		1	1	1

# Implementation: Full Adder

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

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

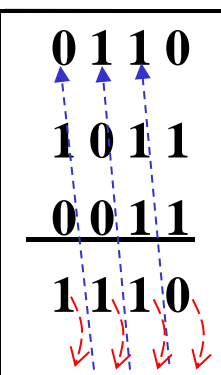
$$C = XY + (X \oplus Y)Z$$



# 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  $i + 1$

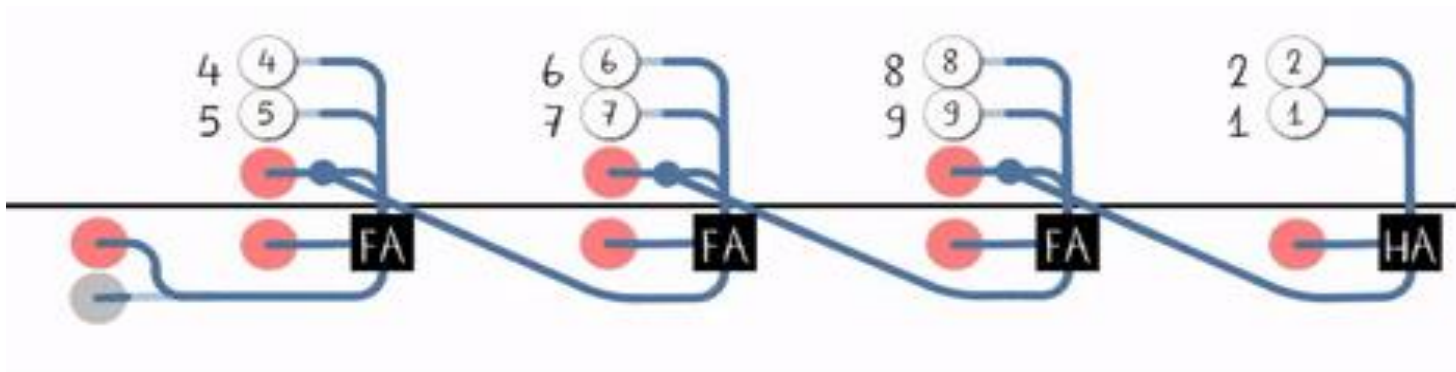
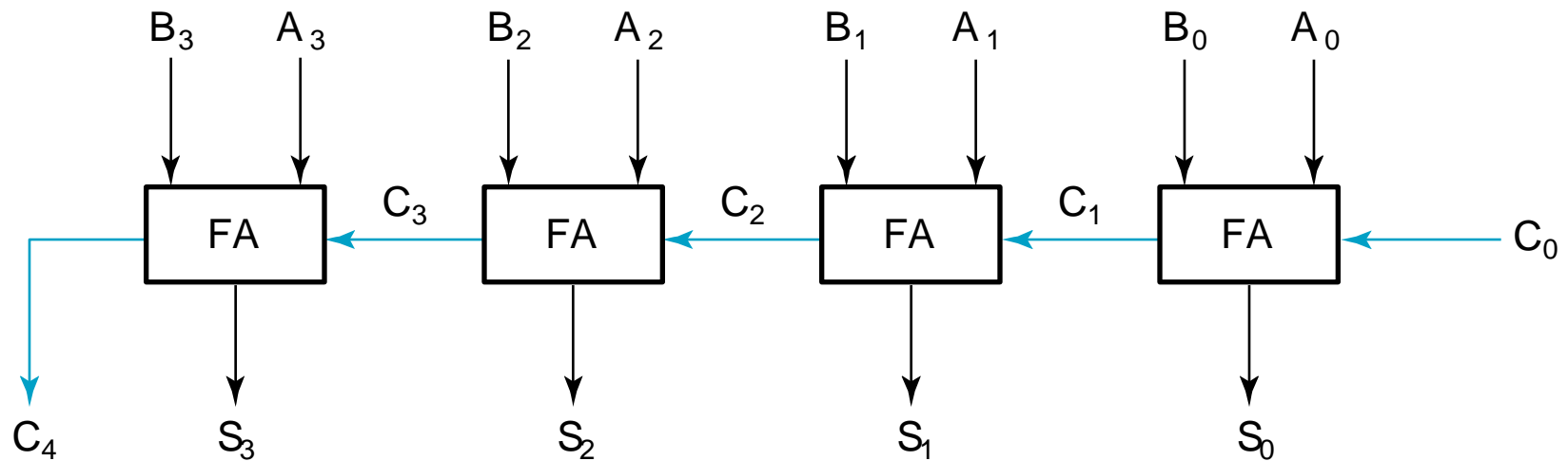
Description	Subscript 3 2 1 0	Name
Carry In	0 1 1 0	$C_i$
Augend	1 0 1 1	$A_i$
Addend	<u>0 0 1 1</u>	$B_i$
Sum	1 1 1 0	$S_i$
Carry out	0 0 1 1	$C_{i+1}$





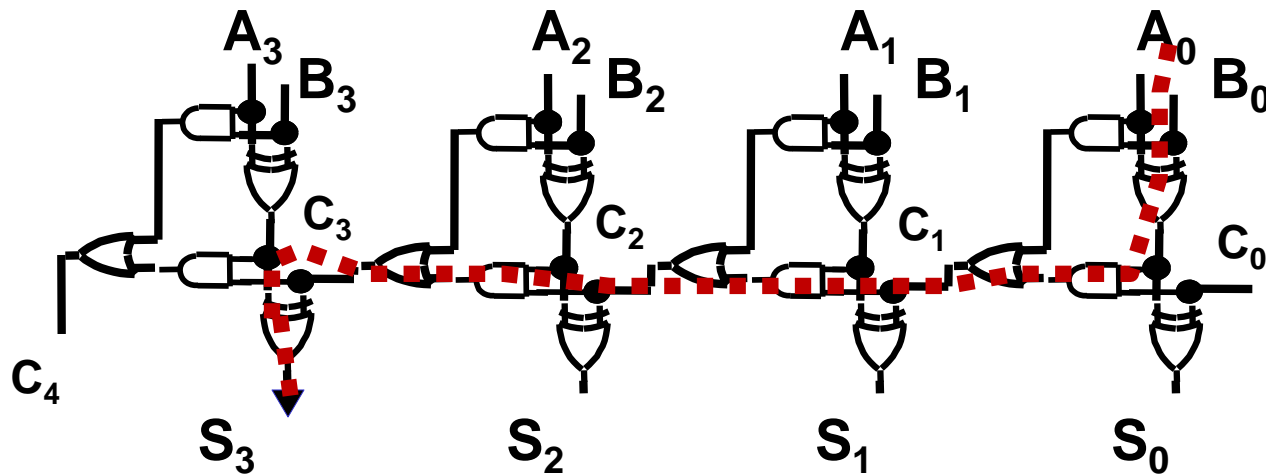
# 4-bit Ripple-Carry Binary Adder

- 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  $S_3$ .

# Revisit the Full Adder

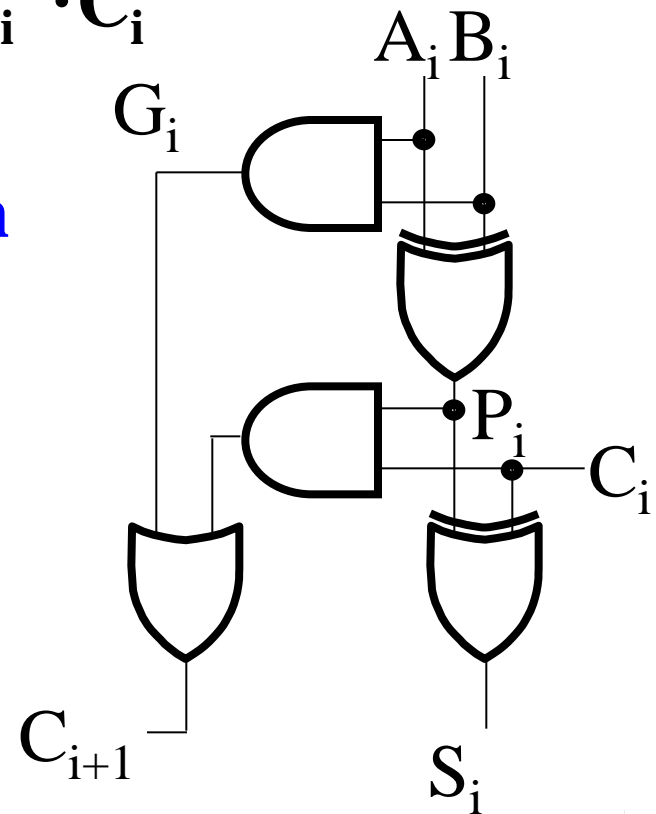
- $S_i$  is the XOR result of variable  $A_i$ ,  $B_i$  and  $C_i$ . And carry function  $C_{i+1}$  can be expressed as bellow:

$$S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i$$

$$C_{i+1} = A_i B_i + (A_i \oplus B_i) C_i = G_i + P_i \cdot C_i$$

1)  $G_i$  is called **generate function**

2)  $P_i$  is called **propagate function**



# Carry Lookahead

- Defining the equations for the Full Adder in term of the  $P_i$  and  $G_i$ :

$$\begin{array}{ll} P_i = A_i \oplus B_i & \longrightarrow S_i = P_i \oplus C_i \\ G_i = A_i B_i & C_{i+1} = G_i + P_i C_i \end{array}$$

- In the ripple carry adder:
  - $P_i$  and  $G_i$  are local to each cell of the adder
  - $S_i$  and  $C_{i+1}$  must wait for the carry  $C_i$  to arrive.
- To overcome the carry propagation problem, a carry lookahead adder can calculate the carry bits in advance based on the input with **substitution**.

# 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  $C_0$ :

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1(G_0 + P_0 C_0) \\ = G_1 + P_1 G_0 + P_1 P_0 C_0$$

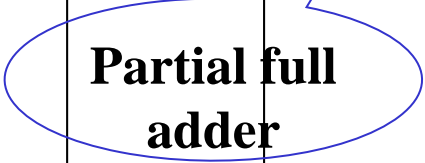
$$C_3 = G_2 + P_2 C_2 = G_2 + P_2(G_1 + P_1 G_0 + P_1 P_0 C_0) \\ = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

$$C_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 \\ + P_3 P_2 P_1 P_0 C_0$$

breaking the  
dependency on  $C_i$   
with substitution

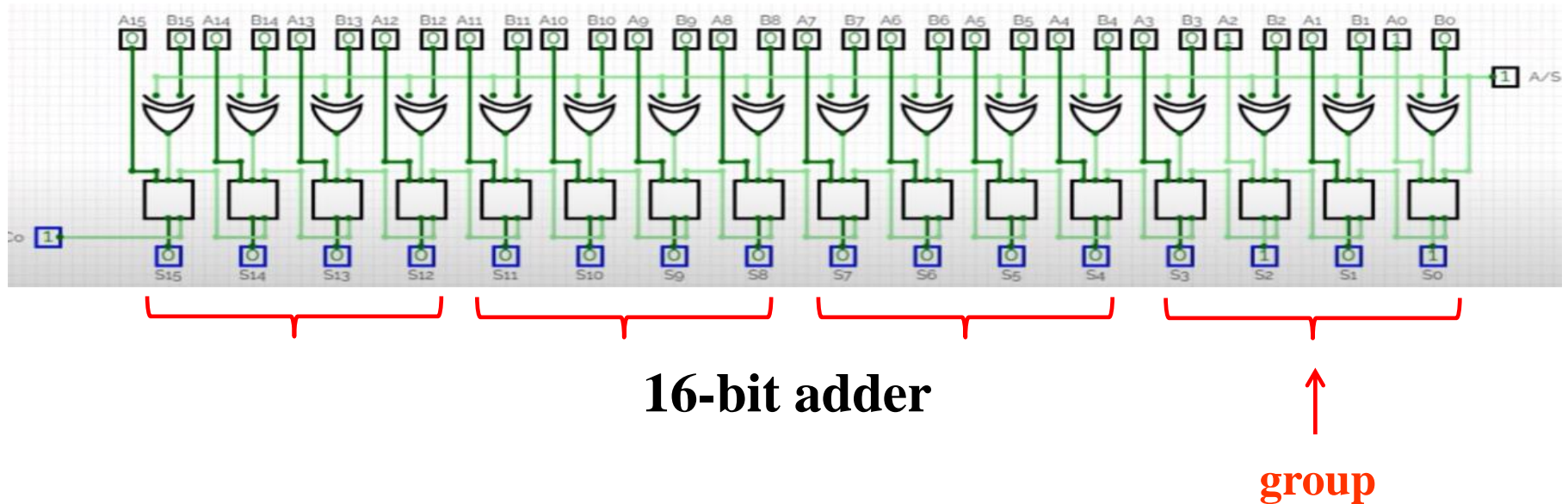
- By using these formulas, the time complexity of the adders can be improved from  $O(n)$  to  $O(1)$ .

Page 10 of 10



# Group Carry Lookahead Logic

- Due to **limited gate fan-in**, carry lookahead extension is not feasible for larger adders (e.g., 16-bit adder).



# Group Carry Lookahead Logic (Cont.)

---

- $C_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$
- The concept can be 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:

$$\begin{aligned} C_4 &= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0 \\ &= G_{0-3} + P_{0-3} C_0 \end{aligned}$$

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



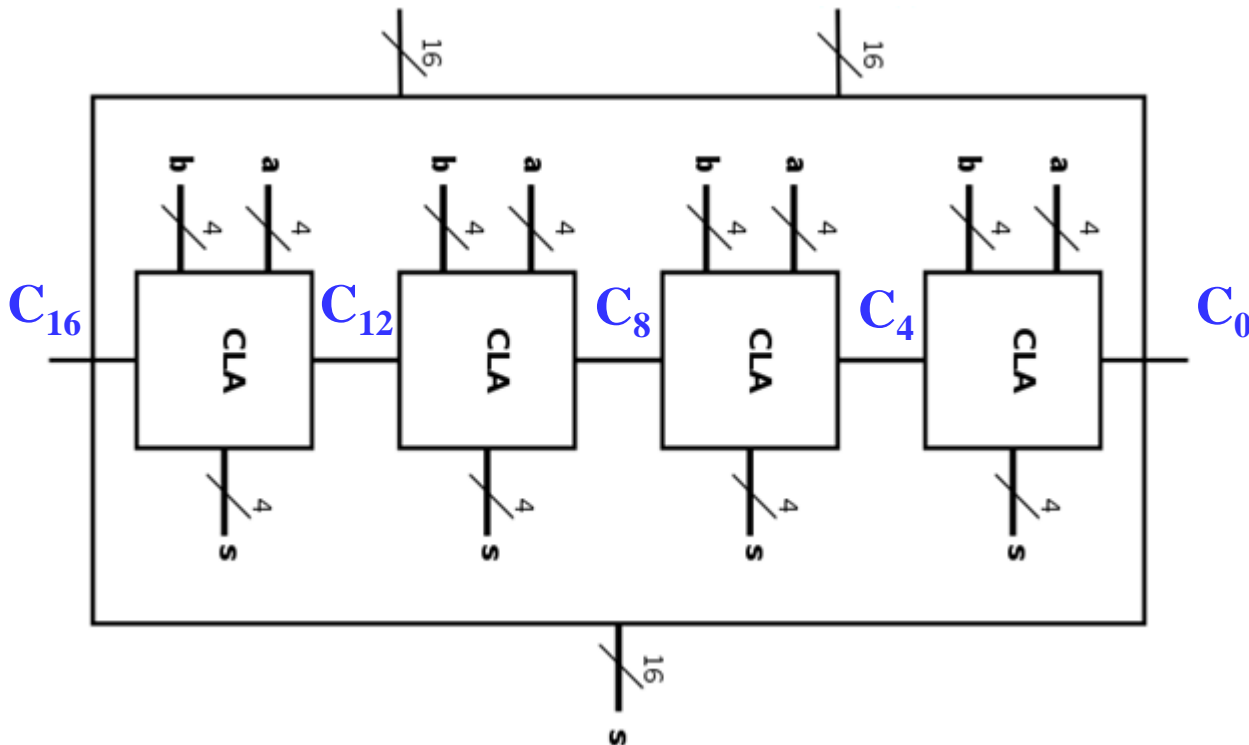
# Group Carry Lookahead Logic (Cont.)

---

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

# Group Carry Lookahead Logic (Cont.)

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



- Appendix B: Fast Carry Lookahead

# Big Picture of Binary Adder

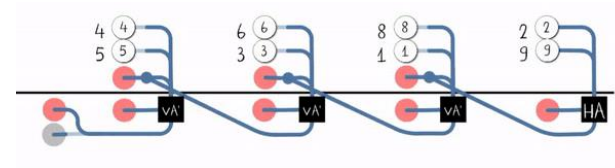
traditional  
logic design



**Two-Level  
Binary Adder**

**Problem: truth-  
table won't work**

**Solution**  
→  
**iterative array  
(multi-level logic)**



**Binary Ripple  
Carry Adder**

**Problem: carry  
propagation delay**

**Solution** ↓ **produce the  
carry ahead**

**Group Carry  
Look-Ahead Adder**

**Solution**  
←  
**group generate/  
group propagate**

**Carry Look-  
Ahead Adder**

**Problem: limited  
gate fan-in for carry**

# Unsigned Subtraction

## ■ Algorithm:

- Subtract the subtrahend  $N$  from the minuend  $M$  ( $M - N$ )
- If **no borrow occurs**, then  $M \geq N$ , and the result is a non-negative number and correct.
- If **an borrow occurs**, then  $N > M$ , and the difference is  $M - N + 2^n$  rather than  $M - N$ . The result should be **negative**, and so we need to **correct** its magnitude.

## ■ Examples:

no borrow occurs		if borrow occurs, needs correction
	0	1
	1001	0100
	– 0111	– 0111
	0010	1101
		→
		10100
		– 0111
		1101

# Unsigned Subtraction (continued)

- If an borrow occurs, the correction method is:
  - subtracting the difference ( $M - N + 2^n = 2^n - (N - M)$ ) from  $2^n$  to obtain the absolute value of  $M - N$
$$2^n - (M - N + 2^n) = 2^n - (2^n - (N - M)) = N - M$$

note: the subtraction  $2^n - X$  is called **2's complement** of  $X$

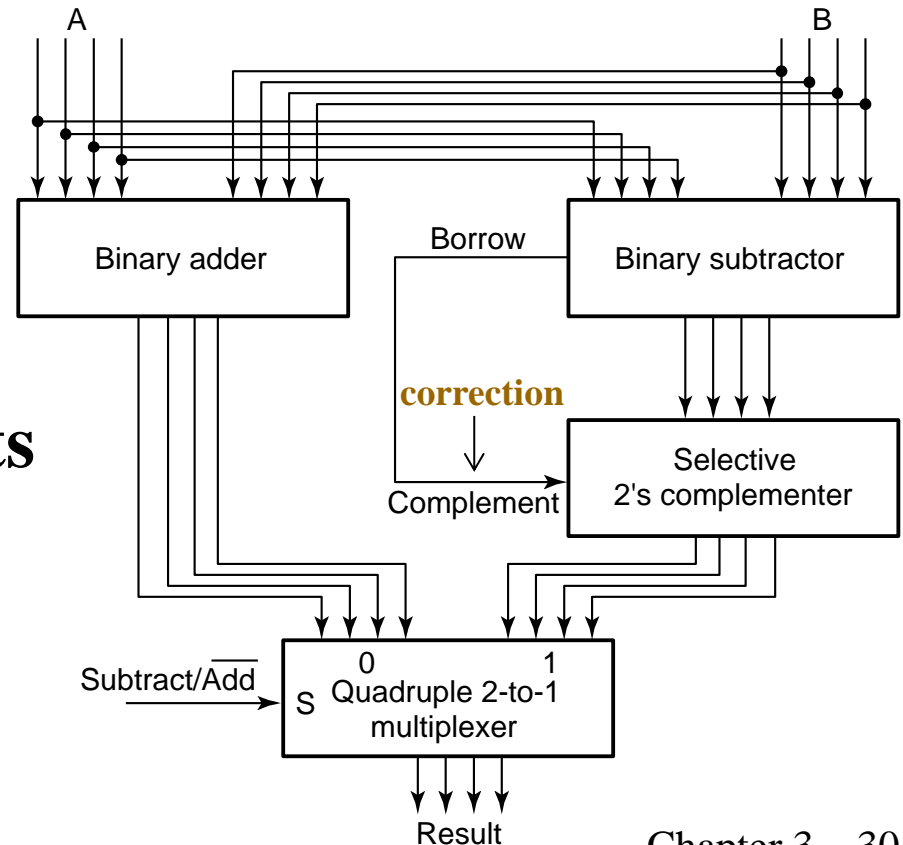
- appending a **minus sign** to the result ( $-(N - M) = M - N$ ).

- Examples:

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

# Unsigned Arithmetic

- To do both unsigned addition and unsigned subtraction requires:
  - adder, subtractor, complementer
  - 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 (补码)
    - Defined as  $r^n - N$
  - **Diminished Radix Complement of  $N$**  (反码)
    - $(r - 1)$ '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):  
 $(r^n - 1) = 2^8 - 1 = 256 - 1 = 255_{10}$  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:

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

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

## Alternative 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.
- Example: 2's Complement

**10010****100**

- **Copy underlined bits:**  
100
- **and complement bits to the left:**

**01101100**

$P=A \oplus B \quad G=AB$

$P=1 \quad G=1$

2's complement

$01101100$

$+ \quad 1$

$10010100$


$10000000$

# Three Ways to Calculate 2's Complement

---

- For an  $n$ -bit binary number  $X$ , there are three ways to calculate its 2's complement:
  - **by definition:**  $2^n - X$
  - **by 1's complement plus 1:**  $\bar{X} + 1$
  - **by scanning input from left to right**

- **Example: 2's Complement of 10010100**

- **by definition:**  $2^8 - 10010100 = 01101100$  
- **by 1's complement plus 1:**

$$\begin{array}{r} 100000000 \\ - 10010100 \\ \hline 01101100 \end{array}$$

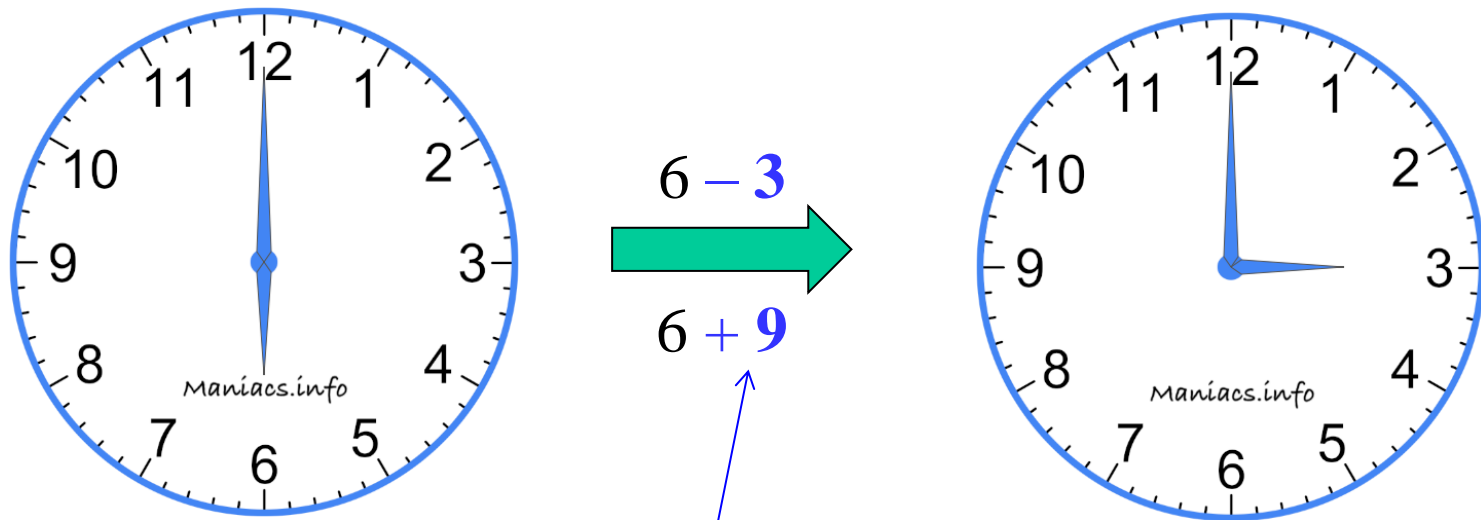
$$\overline{10010100} + 1 = 01101100$$

- **by scanning input from left to right:**

$$10010\underline{100} \longrightarrow \underline{01101}100$$

# Method of Complements (1/4)

- In **modulo N system**, overflow and underflow cause arithmetic calculations to "cycle back" to a value from 0 to N-1.
- For example, a clock is an arithmetic modulo 12 system. Subtracting 3 hours on 6:00 is the same as adding  $12-3=9$  hours. 9 is called complement of 3.



$$6 - 3 \equiv 6 + (12 - 3) \equiv 3 \pmod{12}$$

# Method of Complements (2/4)

- Why subtracting  $X$  is equivalent to adding its complement  $N-X$  in modulo  $N$ ?
    - e.g.,  $6 - 3 \equiv 6 + 9 \pmod{12}$
  - For a positive integer  $N$ , the integers  $a$  and  $b$  are **congruent modulo  $N$**  (同余), if they have the same remainder on division by  $N$ .
  - $-3$  and  $+9$  are **congruent modulo 12** ( $-3 \equiv 9 \pmod{12}$ )
    - $-3 = 12 \times 0 - 3 = 12 \times (-1) + 9 \Rightarrow -3 \pmod{12} = 9$
    - $9 = 12 \times 0 + 9 \Rightarrow 9 \pmod{12} = 9$
  - Congruent modulo  $N$  satisfies several properties:
    - $a \equiv a$
    - $a \equiv b$  and  $c \equiv d \Rightarrow a \pm c \equiv b \pm d$
    - $a \equiv b$  and  $c \equiv d \Rightarrow ac \equiv bd$
    - ...
- }  $\longrightarrow 6 - 3 \equiv 6 + 9 \pmod{12}$

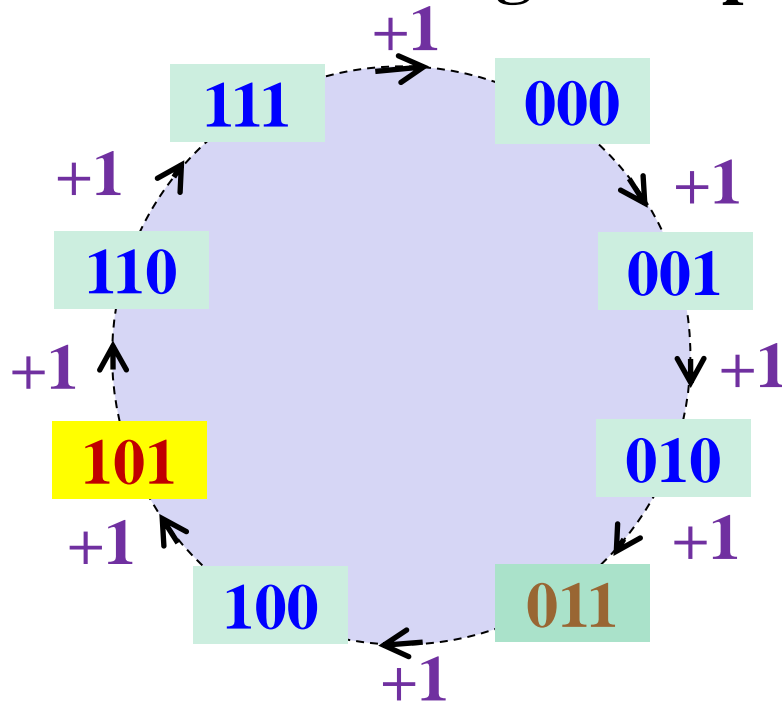
# Method of Complements (3/4)

---

- **Question:** give an integer  $a$ , how to find its congruent modulo  $N$ ?
- **Answer:** the difference of congruent modulo  $N$  is a multiple of  $N$ .
- **Lemma:**  $a \equiv b \pmod{N}$  if and only if  $N \mid (a-b)$ .
  - e.g.,  $\{..., -15, -3, 9, 21, ...\}$  are all congruent modulo 12.
- In modulo  $N$ ,  $-X$  and  $-X + N$  (namely  $N-X$ ) are congruent modulo  $N$ , therefore we can subtract  $X$  by adding its complement  $N-X$ , which is  $Y - X \equiv Y + (N - X) \pmod{N}$ .
  - e.g.,  $6 - 3 \equiv 6 + (12 - 3) \equiv 6 + 9 \pmod{12}$

# Method of Complements (4/4)

- The following example subtracts 2 from 5 (mod  $2^3$ ).



- $101 - 010 = 101 + (1000 - 010)$   
 $= 101 + 110 = 011$
- $-010$  and  $110$  are **congruent**  
**modulo  $2^3$**
- $110$  is **complement** of  $010$

- Digital systems have limited precision (e.g., 64 bits), therefore they are modulo  $2^N$  system (e.g., mod  $2^{64}$ ).
- Complements allows addition and subtraction to be done in a same and simpler way.

# 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 ( $-N$  and  $2^n - N$  are congruent modulo  $2^n$ ):

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

- 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 correct result  $-(N - M)$ 
  - take the 2's complement of the sum ( $2^n - (2^n - (N - M)) = N - M$ )
  - place a  $-$  to its left ( $-(N - M)$ ).



# Unsigned 2's Complement Subtraction Example 1

---

- Find  $01010100_2 - 01000011_2$

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

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

## Unsigned 2's Complement Subtraction Example 2

---

- Find  $01000011_2 - 01010100_2$

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

2's comp

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

# Unsigned Arithmetic vs. Unsigned 2's Complement Arithmetic

---

- **Unsigned arithmetic** requires:
  - an adder for unsigned addition
  - a subtractor for unsigned subtraction
  - a complements for correcting the magnitude
- **Unsigned 2's complement arithmetic** requires:
  - an adder for unsigned addition and subtraction
  - a complements for complementing the subtrahend and correcting the magnitude

# 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:

$$\mathbf{s} \mathbf{a}_{n-2} \dots \mathbf{a}_2 \mathbf{a}_1 \mathbf{a}_0$$

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

# Converting Signed Integer to Signed 2's Complement

---

- **Converting signed integer to signed 2's complement:**
  - **Positive:** all positive numbers in 2's complement are the same as they would be in unsigned binary
  - **Negative:** two steps for conversion
    - Find the binary representation of the absolute value of the number
    - Find the 2's complement
- **E.g., convert -7 to a 4-bit signed-complement**
  - absolute value: 0111
  - 2's complement: 1001

# Signed Integer Representation Example

- $r = 2, n = 3$

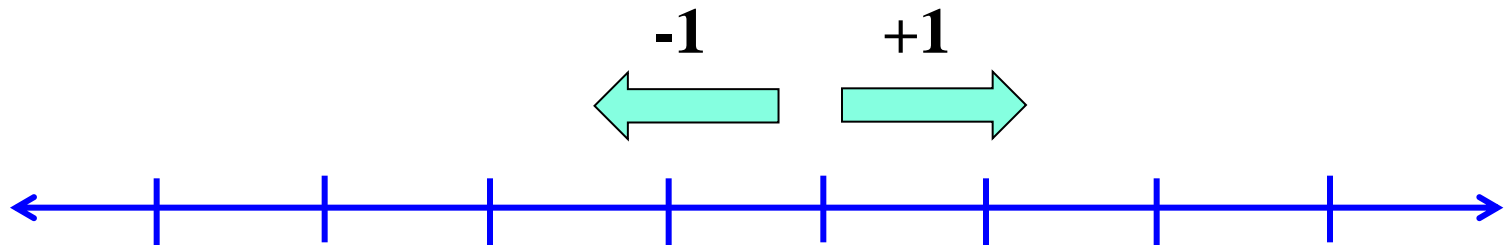
Number	Sign - Mag.	1's Comp.	2's Comp.
+3	011	011	011
+2	010	010	010
+1	001	001	001
+0	000	000	000
-0	100	111	—
-1	101	110	111
-2	110	101	110
-3	111	100	101
-4	—	—	100

⏟  
Signed-Complement



# Encoding Scheme of Signed 2's Complement

- E.g., encoding scheme of signed 2's complement for 3-bits:



**encoding:** 100 101 110 111 000 001 010 011

**value:** -4 -3 -2 -1 0 1 2 3

**weight:**  $-2^2 = -4$

↑  
negative weight

$2^0 = 1$   $2^1 = 2$

↑ ↑  
positive weight

# Converting Signed 2's Complement to Signed Integer (1/2)

---

- **Converting signed 2's complement to signed integer:**
  - **Positive:** treat it as an unsigned binary number and convert normally
  - **Negative:** three steps for conversion
    - Find the 2's complement
    - Find the absolute value of the number
    - Append a **minus sign**
- **E.g., convert 10011 to a signed integer**
  - 2's complement: 01101
  - absolute value: 13
  - append a minus sign: -13

# Converting Signed 2's Complement to Signed Integer (2/2)

---

- An alternative method to convert signed 2's complement is **negative weighting**:

- the most significant bit has negative weight  $-2^{n-1}$
- other bits hold positive weights as usual

$$\text{value of } X = -2^{n-1} \cdot B_{n-1} + \sum_{i=0}^{n-2} 2^i \cdot B_i$$

- A simple formulation for negative weighting

- Positive:  $\text{value of } X = -2^{n-1} \cdot 0 + \sum_{i=0}^{n-2} 2^i \cdot B_i$

- Negative:  $\text{value of } X = -(2^n - X) = -2^n + X$

$$= -2^n + 2^{n-1} + \sum_{i=0}^{n-2} w_i \cdot B_i$$

$$= -2^{n-1} + \sum_{i=0}^{n-2} w_i \cdot B_i$$

- E.g., convert 10011 to a signed integer

- $X = -2^4 + 2^1 + 2^0 = -13$

# Method of Signed 2's Complement

- The method of **signed 2's complements** encodes a **symmetric range of positive and negative integers in a unified way** that the same algorithm and hardware can be applied for addition and subtraction throughout the whole range.
- For example,  $-3 - 3 = -6$

$$\begin{array}{r} 1101 \\ - \underline{0011} \end{array}$$

$$\begin{array}{r} 1101 \\ + \underline{1101} \end{array}$$

$$11010 \rightarrow 1010_2 = -8 + 2 = -6$$

# Facts about Signed 2's Complement (1/2)

- **Sign extension preserves value in 2's complement.**
  - Adding leading zeros to a positive number, and leading ones to a negative number, is called **sign extension** of a 2's complement number.
  - e.g., 0111 → 0000111, 1011 → 1111011
- **Formulation for sign extension of negative integers**
  - e.g., sign extension from n-bit to m-bit ( $n < m$ )
$$\begin{aligned}X_m &= -2^{m-1} + 2^{m-2} + \dots + 2^{n-1} + \sum_{i=0}^{n-2} w_i \cdot B_i \\&= -(2 \cdot 2^{m-2} - 2^{m-2}) + 2^{m-3} \dots + 2^{n-1} + \sum_{i=0}^{n-2} w_i \cdot B_i \\&= -2^{m-2} + 2^{m-3} + \dots + 2^{n-1} + \sum_{i=0}^{n-2} w_i \cdot B_i \\&= -2^{n-1} + \sum_{i=0}^{n-2} w_i \cdot B_i \\&= X_n\end{aligned}$$

# Facts about Signed 2's Complement (2/2)

- In 2's complement arithmetic, carry out does not indicate an overflow (e.g., sign extension).
- The range of an n-bit signed 2's complement
  - upper bound:  $2^{n-2} + 2^{n-3} \dots + 2^1 + 2^0 = 2^{n-1} - 1$
  - lower bound:  $-2^{n-1}$
- Examples (e.g., 8-bit subtraction)

		carry bit		
	1		1	
-70	1011 1010		-70	1011 1010
-20	1110 1100		-80	1011 0000
<hr/>			<hr/>	
-90	1010 0110		-150	0110 1010
✓			^	
-128	sign extension		-128	overflow
	no overflow			

# 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 doesn't occur, the **sign** of the result is the **same** as the sign of the first operand.
  - 3. 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.
  - 4. Overflow will never occur.

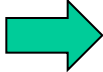
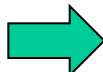
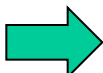

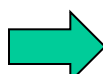
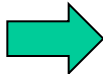
**Operand-1**: sign bit

**Operand-2**: sign bit

**Operator**: +/0, -/1

# Sign-Magnitude Arithmetic Examples

---

- **Example 1:**     **0010**     010      $2_2 + 5_2 = 7_2$   
                          + 0101     + 101  
      parity of the          111          0111  
      signs is 0
- **Example 2:**     **0010**     010      $2_2 + (-5)_2 = (-3)_2$   
                          + 1101     - 101  
      parity of the          1101          1011  
      signs is 1
- **Example 3:**     **0011**     011      $3_2 + (-3)_2 = 0_2$   
                          + 1011     - 011  
      parity of the          000          0000  
      signs is 1



# Signed-Magnitude Arithmetic

get the relations of  
two operands

Parity of the  
Three Signs

$(Op \oplus Opd1) \oplus Opd2$

perform unsigned  
addition/subtraction

Unsigned  
Arithmetic

check overflow,  
make correction

Append Sign Bit

depend on  
borrowing or not

Example 1:    **0010**    **010**  
                  +**0101**    +**101**  
parity of the    **111** → **0111**  
signs is 0

Example 2:    **0010**    **010**  
                  +**1101**    -**101**  
parity of the    **1101** → **1011**  
signs is 1

# Signed 2's 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:

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

# Signed 2's Complement Examples

---

■ Example 1:  $1101$   
 $+ \underline{0011}$

$$(-3)_2 + 3_2 = 0_2$$

$$\boxed{1}0000 \rightarrow 0000$$

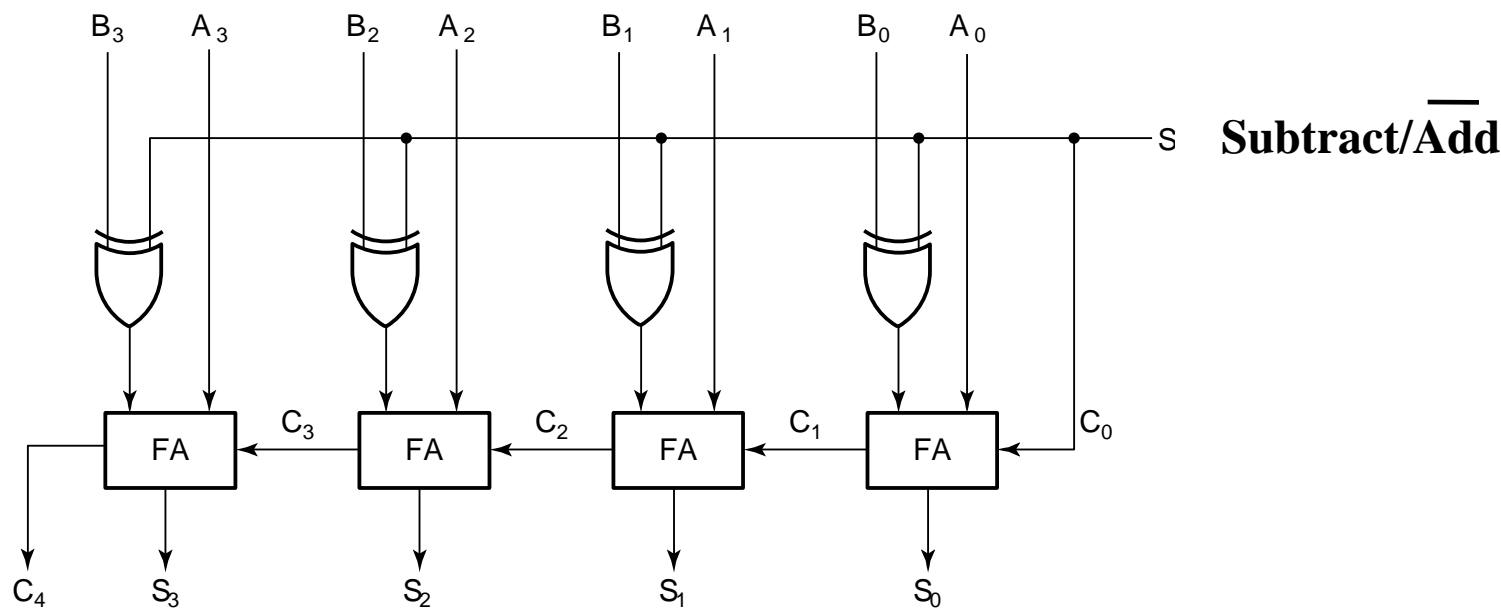
■ Example 2:  $1101$   
 $- \underline{0011}$

$$\begin{array}{r} 1101 \\ + \underline{1101} \end{array} \quad (-3)_2 - 3_2 = (-6)_2$$

$$\rightarrow \boxed{1}1010 \rightarrow 1010$$

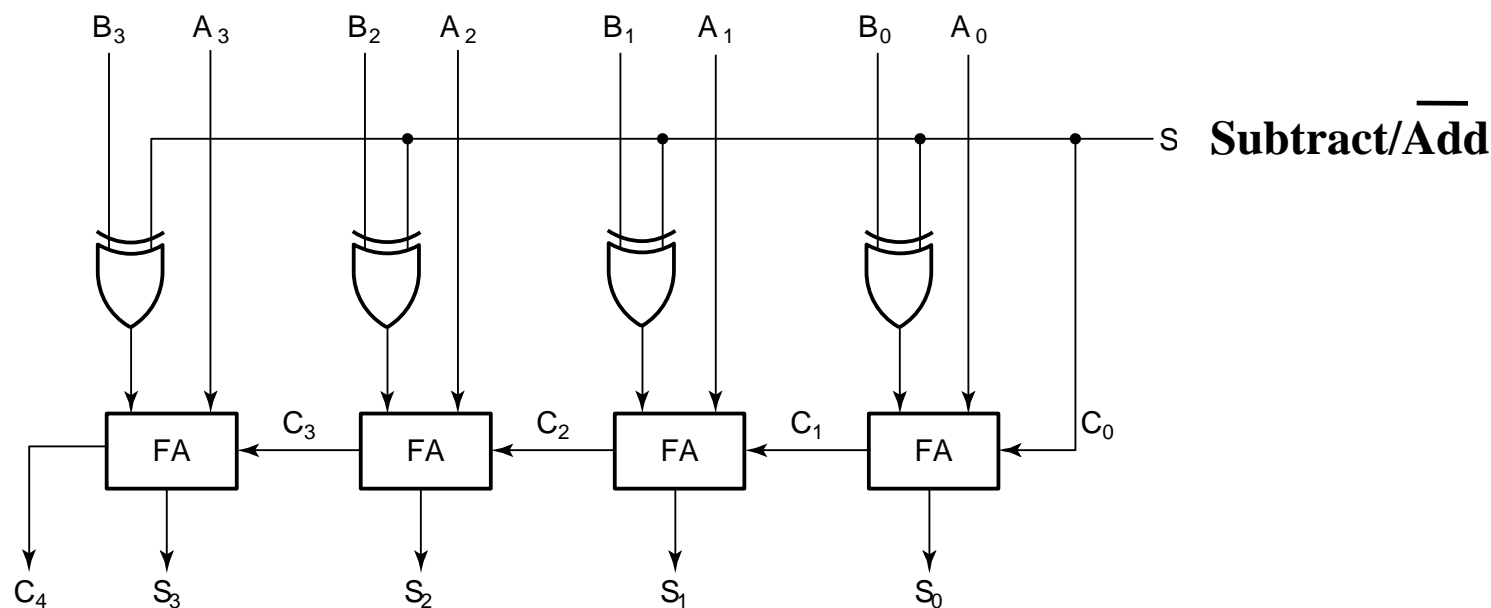
# Signed 2's Complement Adder/Subtractor (1/2)

- 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  $A - B$ :




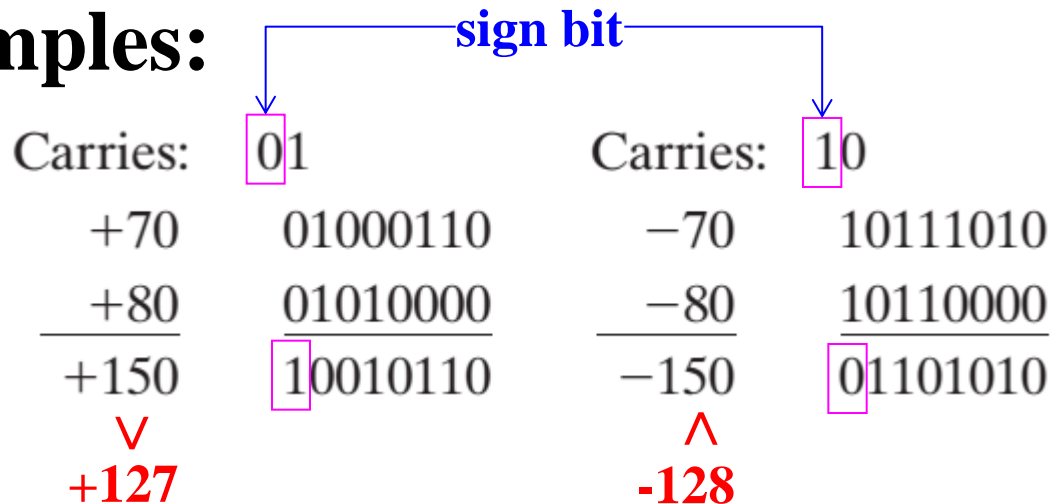
# Signed 2's Complement Adder/Subtractor (2/2)

- 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 (1/3)

- **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:** 



# Overflow Detection (2/3)

- 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: 1 carried in, and 0 carried out of the leftmost full adder (**underflow**)
- Case 2: 0 carried in, and 1 carried out of the leftmost full adder (**overflow**)

Carries: 10

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

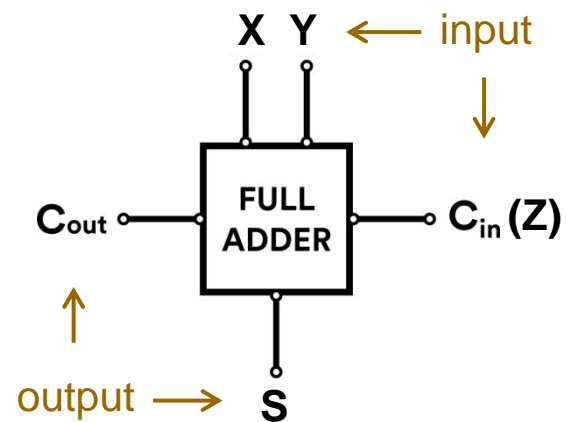
Carries: 01

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

# Overflow Detection (3/3)

- From the view of full-adder's truth table:

X	Y	Z	S	C	<u>overflow</u>
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	0	1	0
1	1	0	0	1	1
1	1	1	1	1	0



- Indicators for detecting overflow can be:

- $V_1 = \bar{X} \bar{Y} S + XY\bar{S}$

- $V_2 = Z \oplus C$

- $V_3 = \bar{X} \bar{Y} Z + XY\bar{Z}$

- $V_4 = Z\bar{C} + XY\bar{Z}$

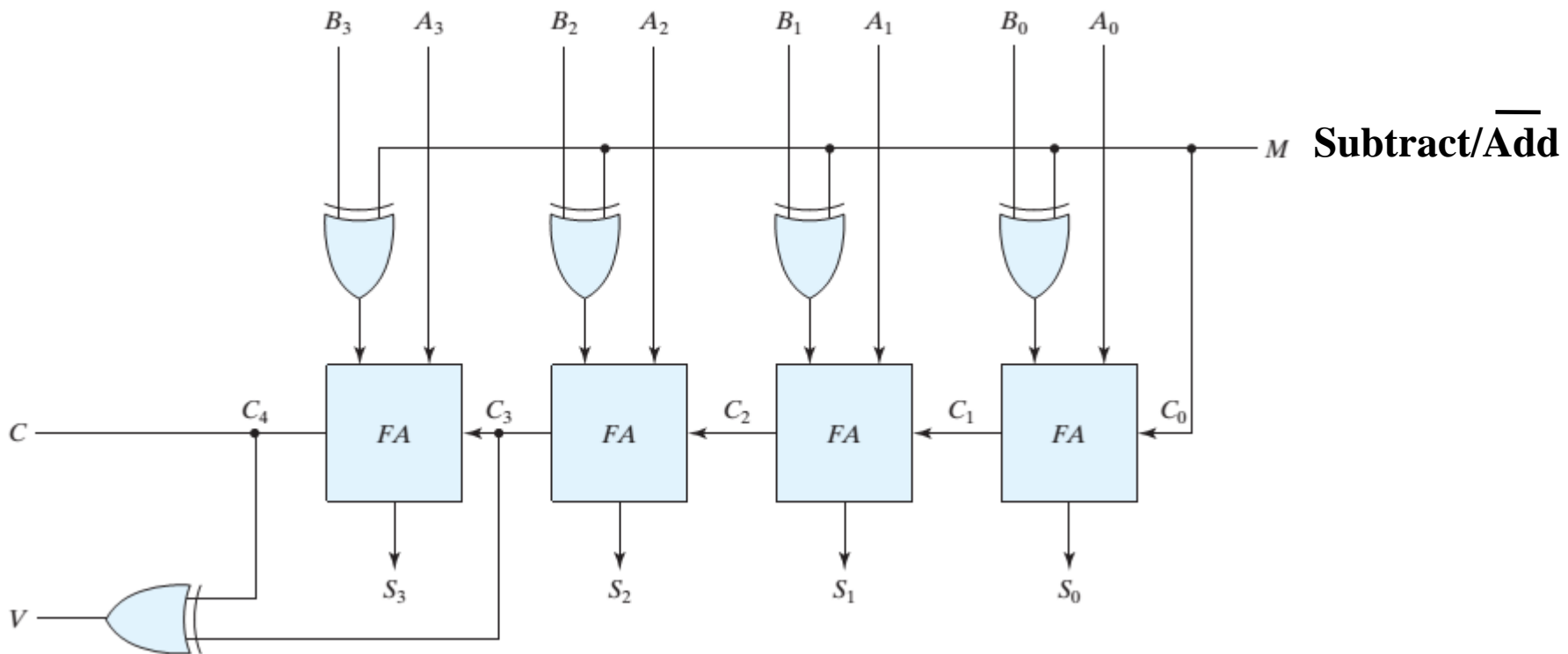
- ...

**totally 25 solutions !**



# Overflow Detection (continued)

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



# Signed-Magnitude Arithmetic vs. Signed 2's Complement Arithmetic

---

## ■ Signed-Magnitude Arithmetic:

- three-step solution

- separate handling of the sign {
- determine the sign of two operands
  - perform unsigned arithmetic
  - append the sign bit

- circuits needed: adder, subtractor, complementer

## ■ Signed 2's Complement Arithmetic

- one-step solution

- care nothing about the sign {
- encode positive and negative numbers together
  - replace a subtraction with an addition

- circuits needed: adder, XOR gates

**Everybody likes a compliment —Abraham Lincoln**

# Big Picture of 2's Complement Arithmetic

## Problem: Complex

## Result: Simplified Circuit

Unsigned  
Arithmetic

Solution

- congruent modulo N

Unsigned 2's  
Complement Arithmetic

← positive  
weight

e.g.,  $010-011 = -(011-010)$   
 $= -001$  (extra subtractor)

e.g.,  $010-011 = 010+101$   
 $= 111 \rightarrow -001$  (only adder)

Signed-  
Magnitude  
Arithmetic

Solution

- negative weighting
- congruent modulo N

Signed 2's Complement  
Arithmetic

← negative  
weight



# 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

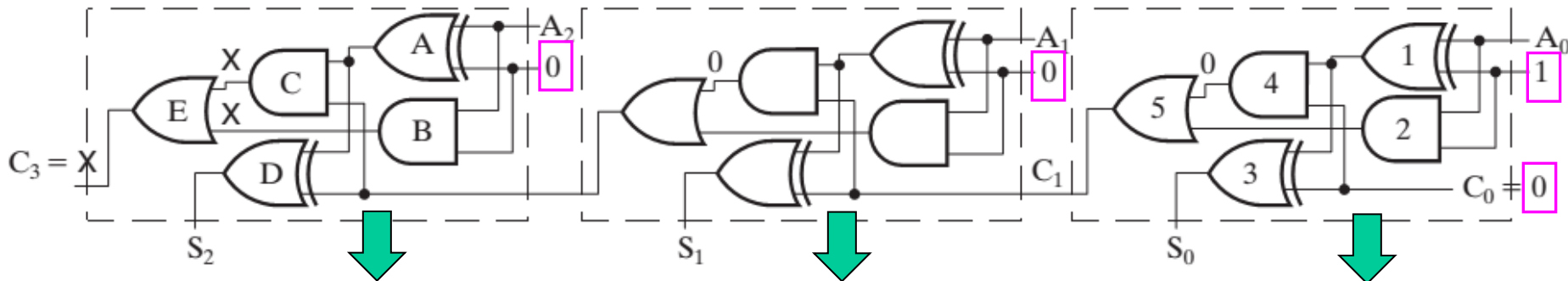
---

- 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.

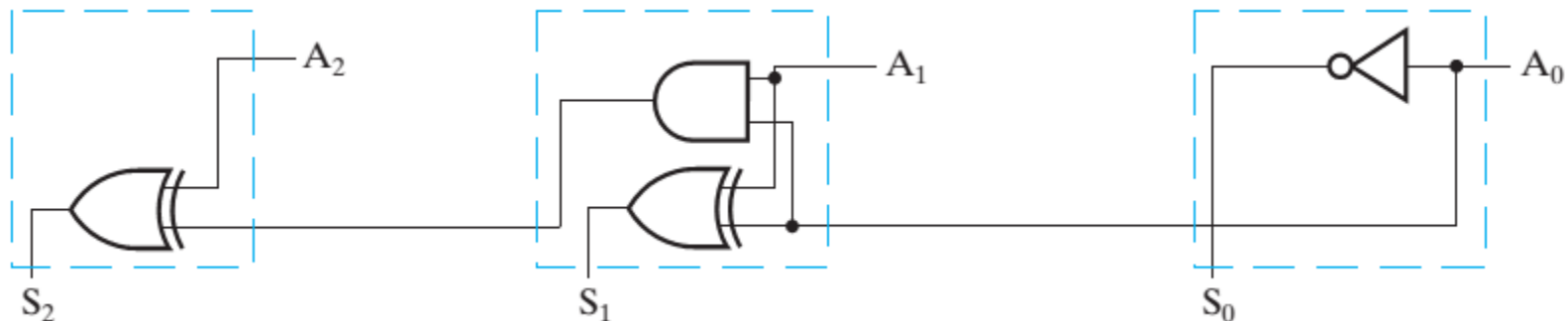
# Design by Contraction Example

- Contraction of a ripple carry adder to **incrementer** for  $n = 3$  ( $A_2A_1A_0 + 1$ )

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



2. simplifying the logic  $\rightarrow$  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)

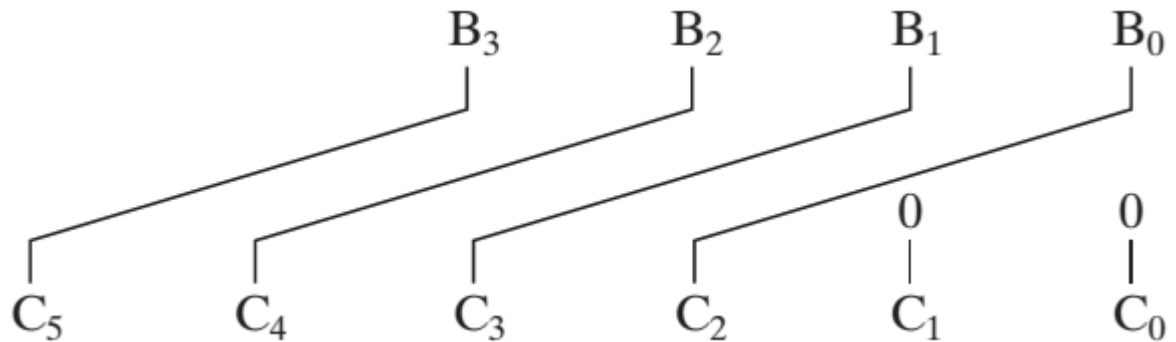
## ■ Decrementing

- 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 $2^n$

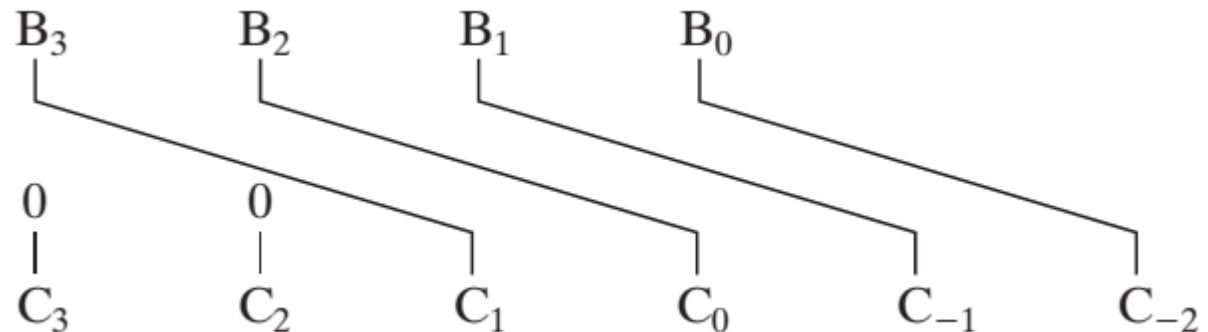
## ■ Multiplication

- Multiplication by  $100_2$
- Shift left by 2



## ■ Division

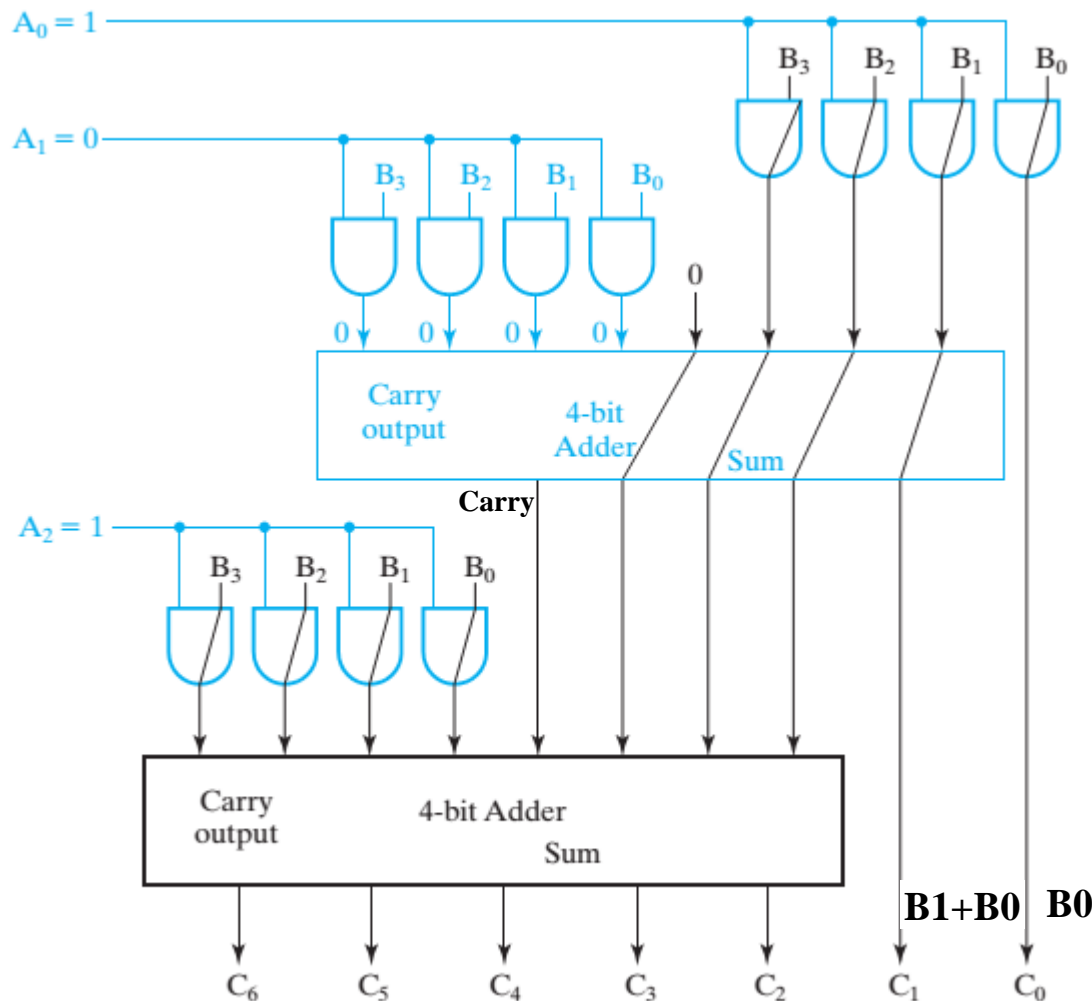
- Division by  $100_2$
- Shift right by 2
- Remainder preserved





# Multiplication by a Constant

## ■ Multiplication of B(3:0) by 101

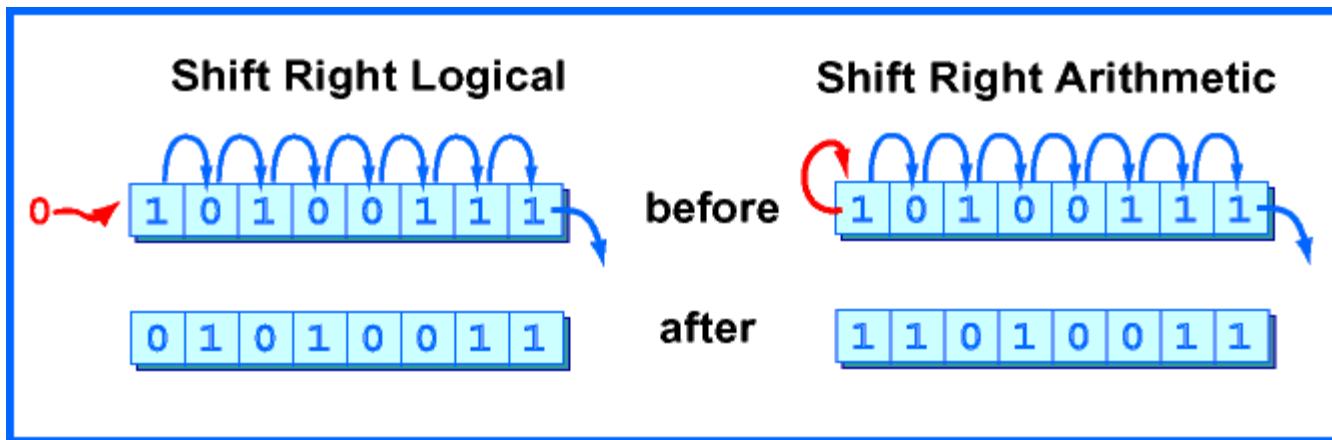


$$\begin{array}{r}
 \begin{array}{cccccc}
 0 & B_3 & B_2 & B_1 & B_0 & \\
 + & B_3 & B_2 & B_1 & B_0 & \\
 + & B_3 & B_2 & B_1 & B_0 & \\
 \hline
 C_6 & C_5 & C_4 & C_3 & C_2 & C_1 & C_0
 \end{array}
 \end{array}$$

$B_1+B_0 \quad B_0$

# Shift Parameters/Shift Functions

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



- **Filling of vacant positions**
  - Zero fill, Sign extension

# 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: 00000000 11110101
  - LSB end: 11110101 00000000

# Sign Extension

---

- **Sign 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: 00000000 01110101
  - **Negative operand** example - 11110101 extended to 16 bits: 11111111 11110101

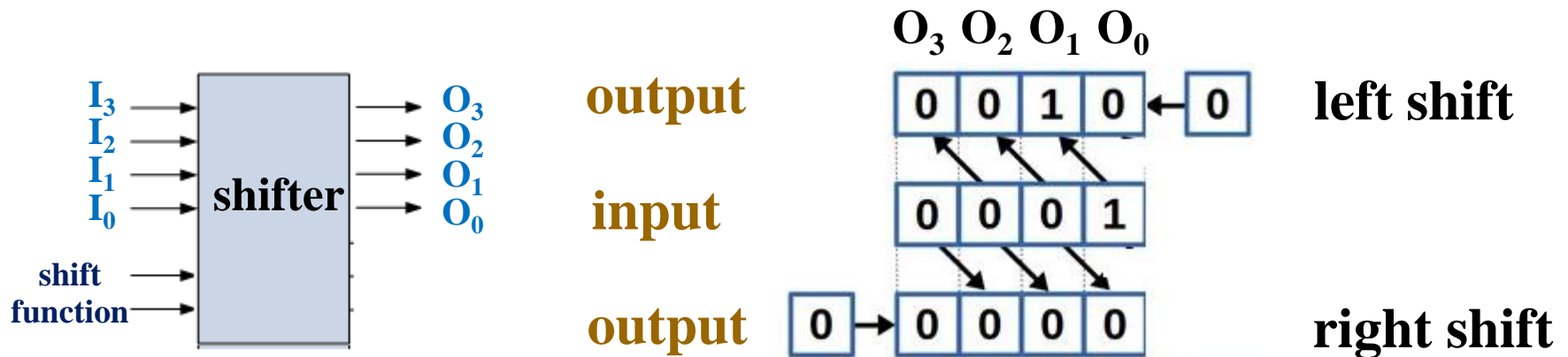
# Design Method of Combinational Shifter

---

- **Two-level logic design**
- **MUX based design**

# 4-Bit Left/Right Shifter with Two-Level Logic Design (1/2)

- **Example: Design a 4-bit left/right logical shifter with 1-bit shift**



# 4-Bit Left/Right Shifter with Two-Level Logic Design (2/2)

---

- **Example: Design a 4-bit left/right logical shifter with 1-bit shift**

shift  
function

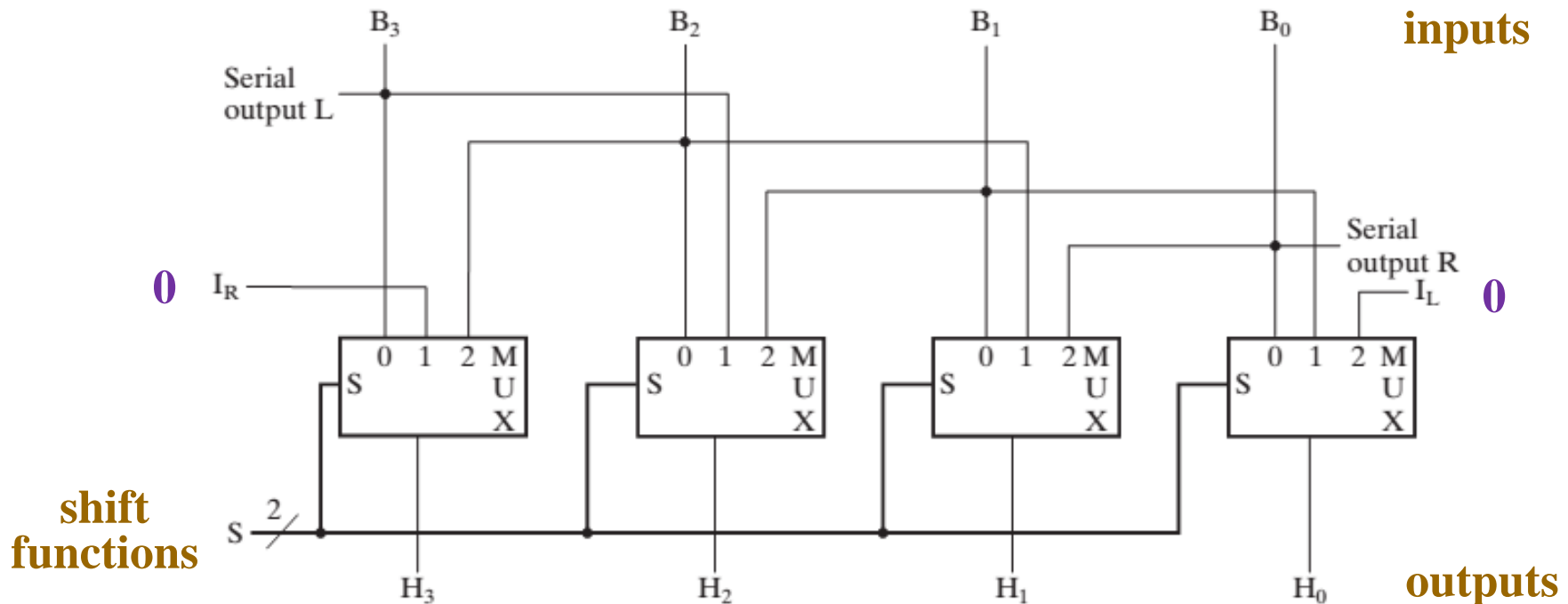
Truth table

left  
right

Direction	Operation	O <sub>3</sub>	O <sub>2</sub>	O <sub>1</sub>	O <sub>0</sub>
0	left 1-bit shift	I <sub>2</sub>	I <sub>1</sub>	I <sub>0</sub>	0
1	right 1-bit shift	0	I <sub>3</sub>	I <sub>2</sub>	I <sub>1</sub>

- $O_3 = I_2 \overline{D}$
- $O_2 = I_1 \overline{D} + I_3 D$
- $O_1 = I_0 \overline{D} + I_2 D$
- $O_0 = I_1 D$

# 4-Bit Left/Right Shifter with MUX

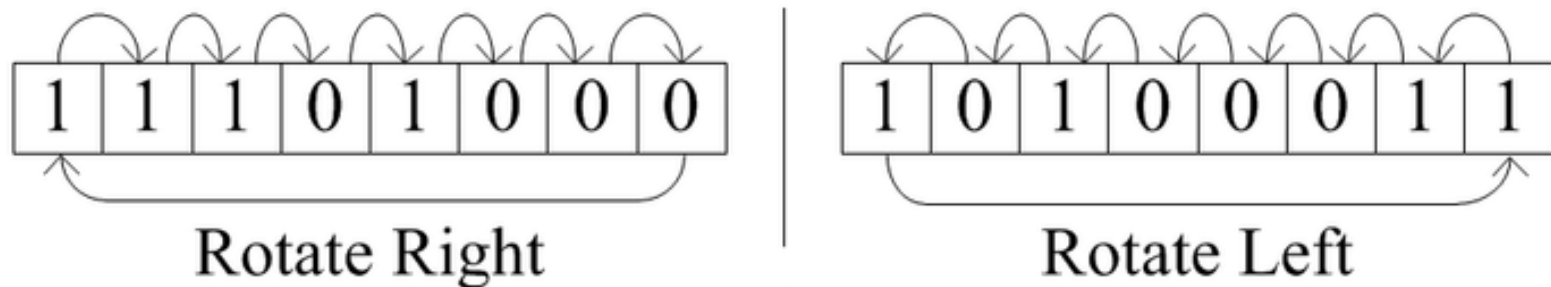


- **Shift Functions ( $S_1, S_0$ )**
  - 00 Pass B unchanged
  - 01 Right shift
  - 10 Left shift
  - 11 Unused
- **Serial Inputs**
  - $I_R$  for right shift
  - $I_L$  for left shift
- **Serial Outputs**
  - R for right shift (Same as MSB input)
  - L for left shift (Same as LSB input)



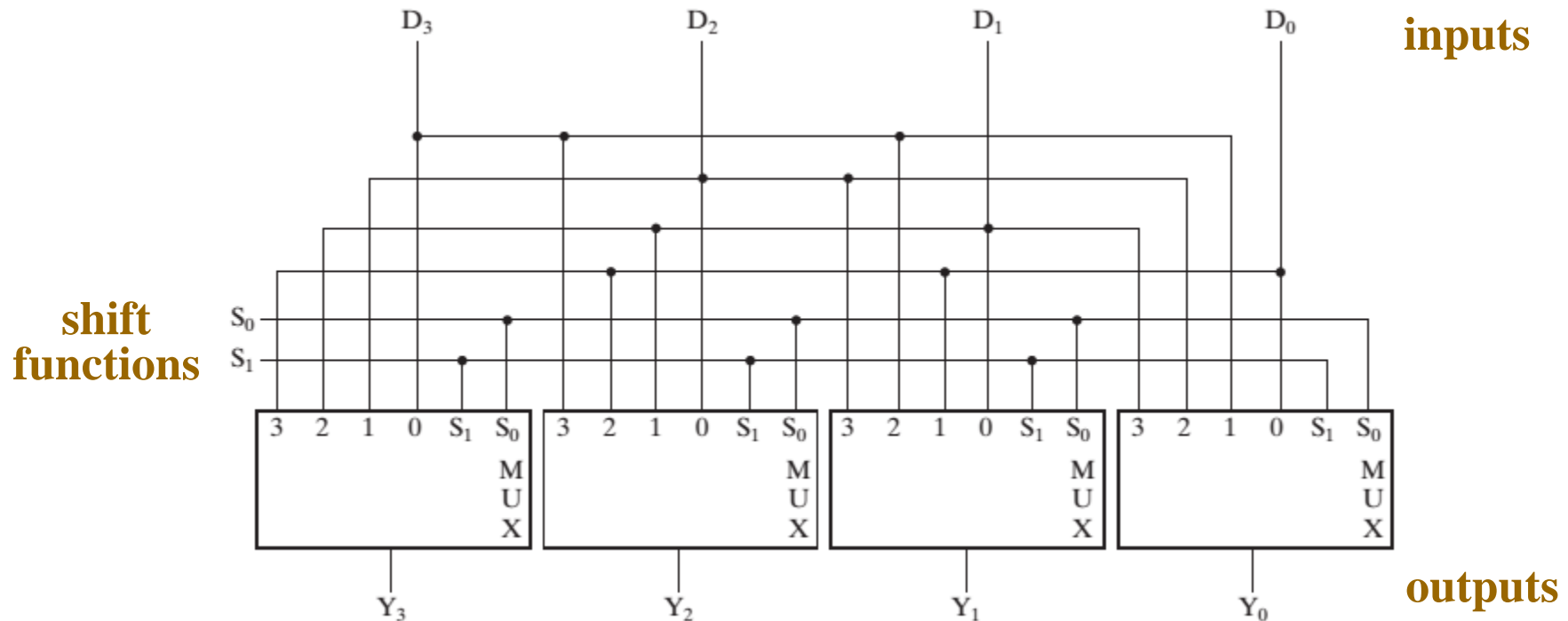
# Barrel Shifter

- 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)



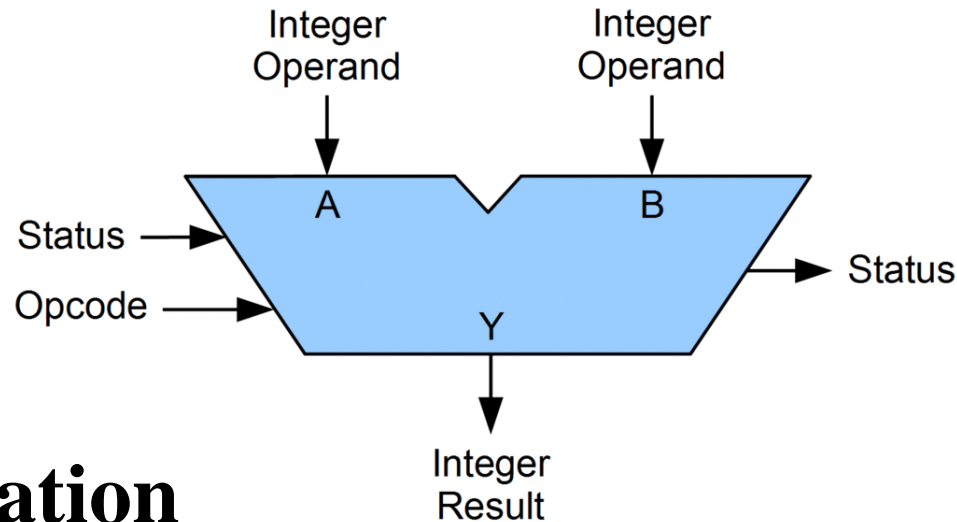
- The circuit rotates its contents left from 0 to 3 positions depending on S:

- $S = 00$  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**

# Arithmetic Logic Unit (ALU)

---



## ■ Specification

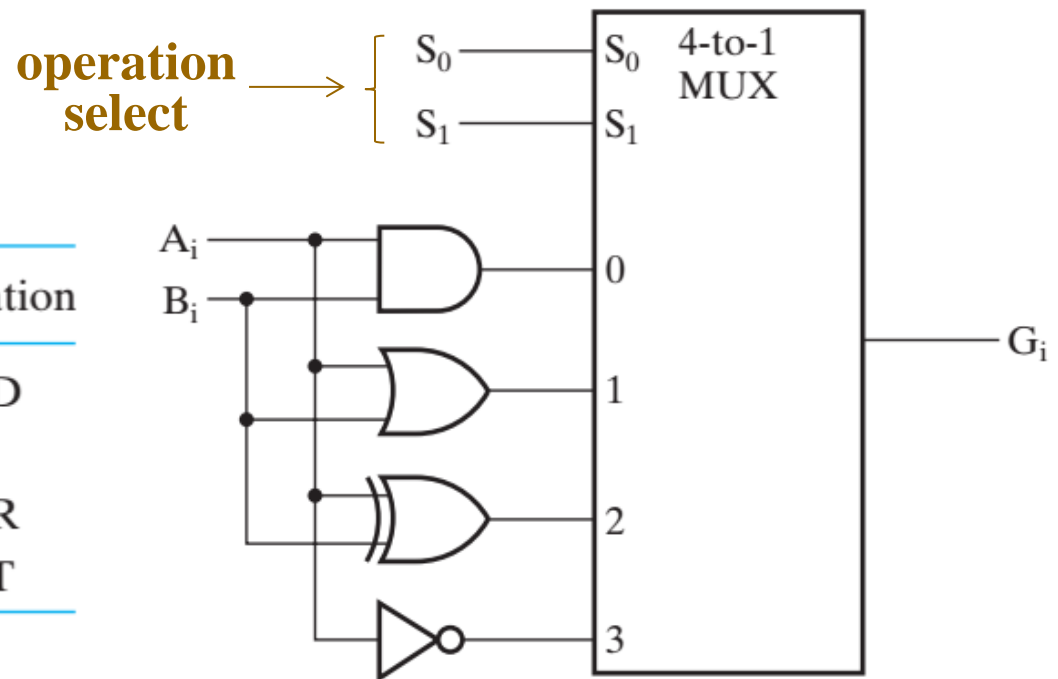
- **two n-bit input:  $A_0$ - $A_{n-1}$ ,  $B_0$ - $B_{n-1}$**
- **signed arithmetic: add/sub/inc/dec**
- **logical operation: and/or/not/xor**
- **mode selection: arithmetic/logical**
- **operation selection**
- **1-bit carry input**

# 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

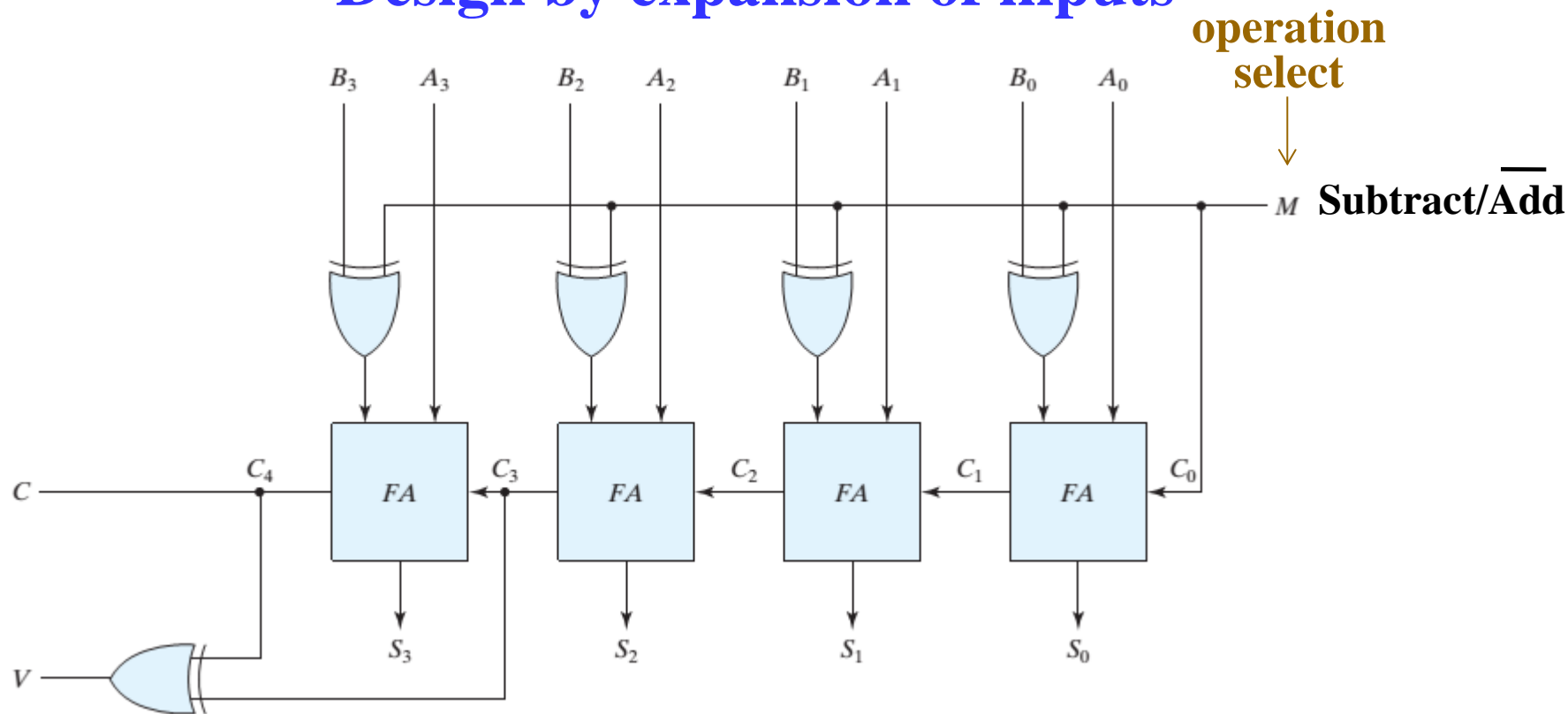


Logic operation circuit

# Revisit the Adder/Subtractor

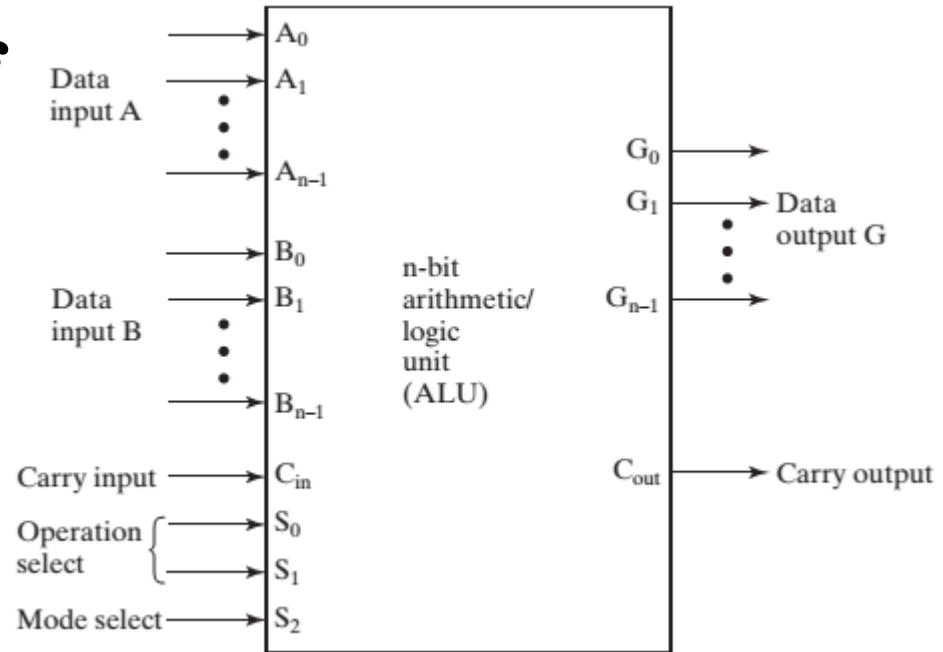
- An adder/subtractor is formed by using **a operation selection and XOR gates to control data input B**.

## Design by expansion of inputs



# Arithmetic Circuit Design Method

- Similarity, the idea of building an arithmetic circuit is to use an adder and control data input B.

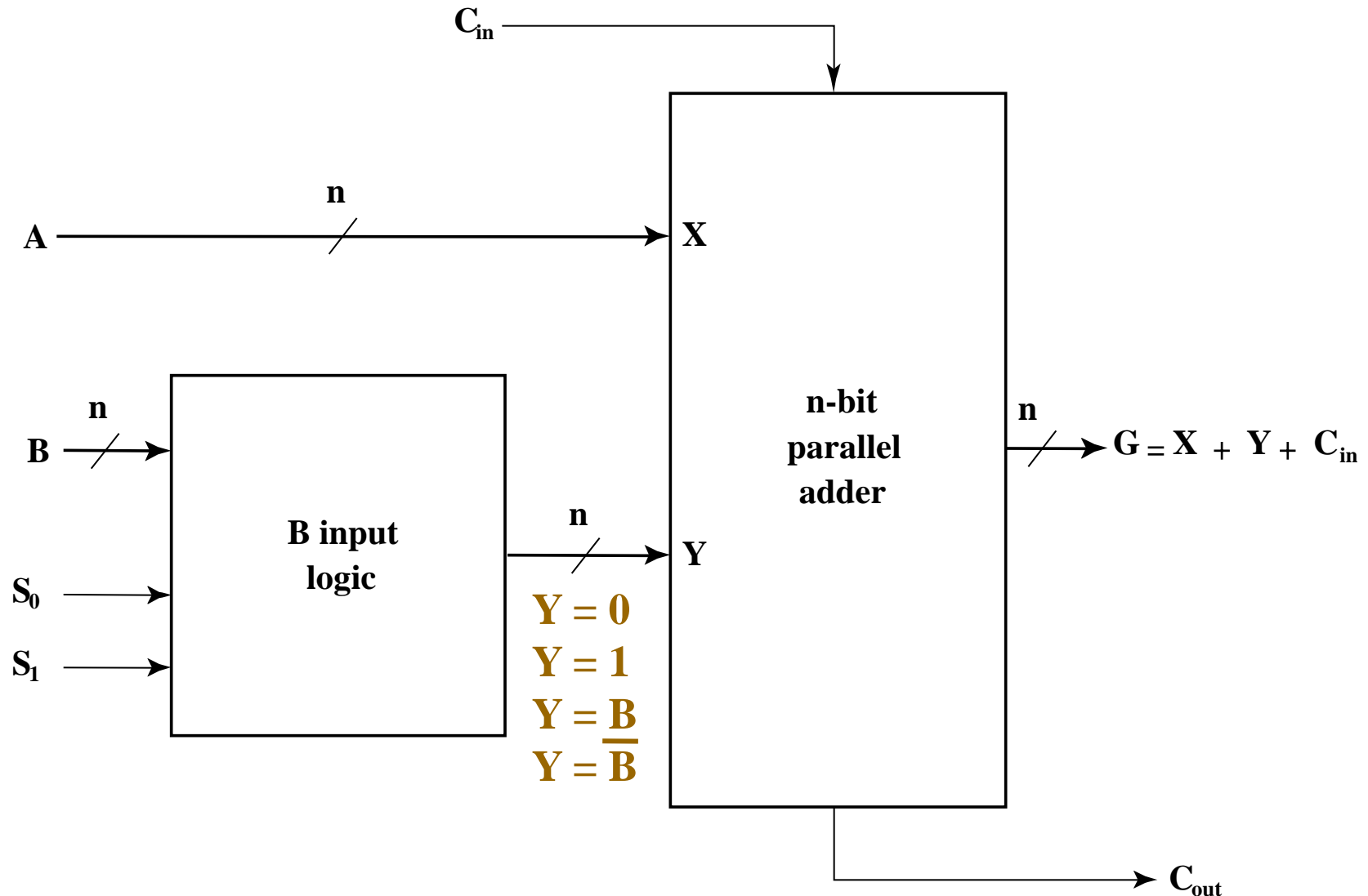


Implementation

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

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

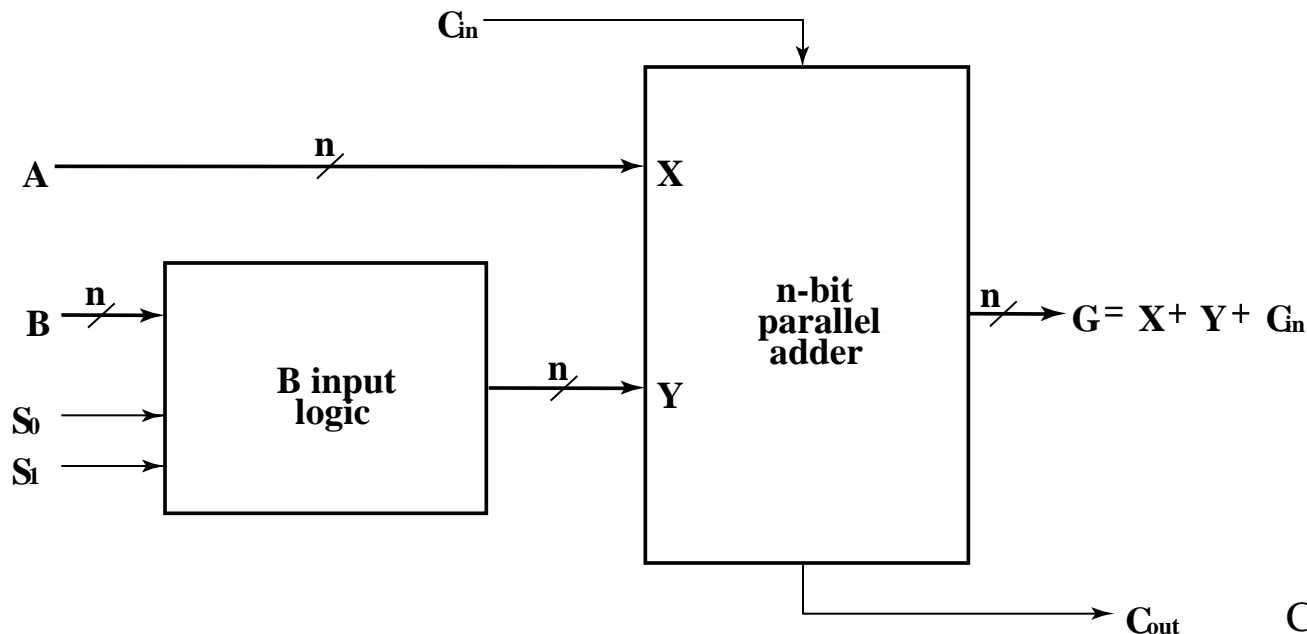
# Arithmetic Circuit Design



# Arithmetic Circuit Design (continued)

- There are only four functions of  $B$  to select as  $Y$  in  $G = X + Y + C_{in}$ :

	$C_{in} = 0$	$C_{in} = 1$
$Y$ {		
• 0	$G = A$	$G = A + 1$ ✓
• B	$G = A + B$ ✓	$G = A + B + 1$
• $\overline{B}$	$G = A + \overline{B}$	$G = A + \overline{B} + 1$ ✓
• 1	$G = A - 1$ ✓	$G = A$

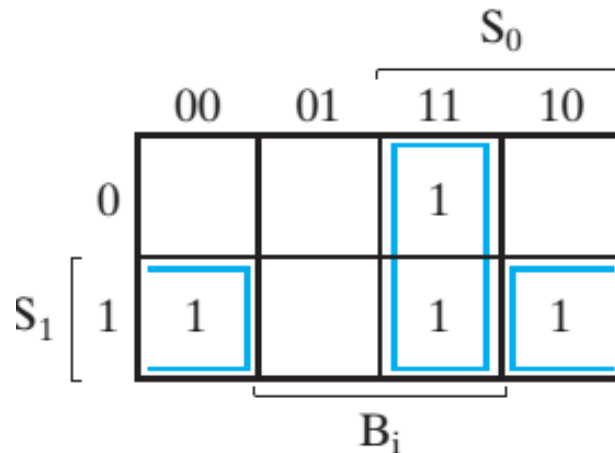




# Arithmetic Circuit Design (continued)

- 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

# Story Time

---



$$c=4195835/3145727=1.33382044....$$



$$c=4195835/3145727=1.33373906....$$



- **Pentium FDIV: the Processor Bug that Shook the World**
- **From a Tiny Flaw, a Major Lesson**
- **Close, But Not Close Enough**
- **Sorry, Wrong Number**
- .....

# Pentium FDIV: the Processor Bug that Shook the World (1/5)

---

- In the fall of 1994, Prof. Nicely was working on **twin primes** problem.....  
$$S = 1/5 + 1/7 + \dots + 1/29 + 1/31 + \dots + 1/p + 1/(p+2) + \dots$$
$$P = 824633702441$$
$$\longrightarrow \text{Brun's constant}$$
- Nicely noticed inconsistencies among several different quantities. In October, he was convinced an error in the then-new Intel Pentium processor, and reported to Intel about the **FDIV bug**.  
(FDIV—the floating point divide instruction on the Pentium)
- However, Intel ignored the request, because
  - the occurrence of the problem was 1 in 9 billion
  - processors were popular among millions of customers

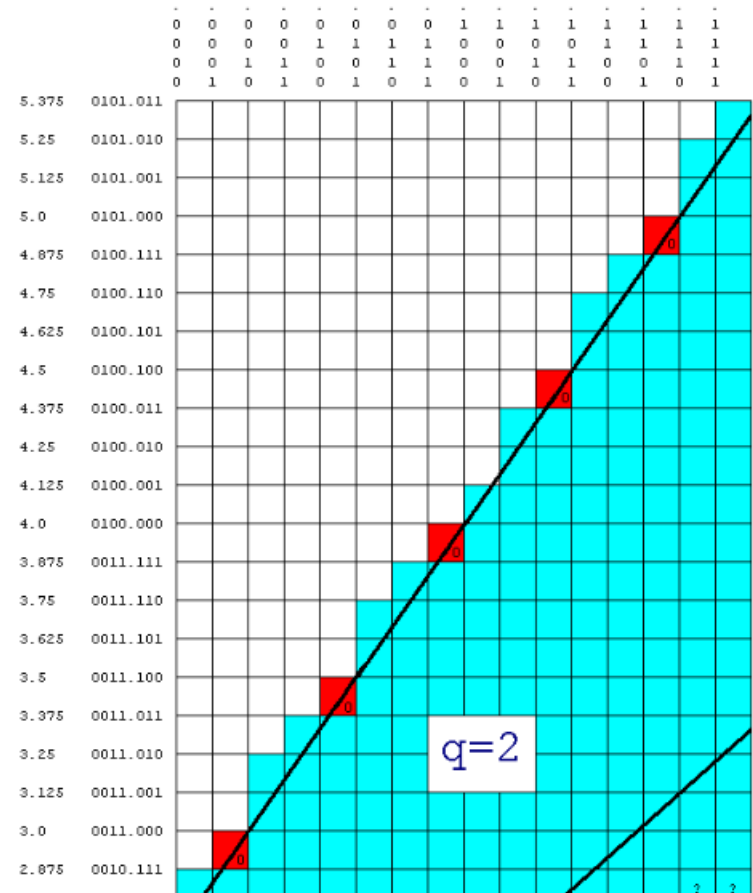
# Pentium FDIV: the Processor Bug that Shook the World (2/5)

---

- On October 30, Nicely sent an email message to a few people, announcing his discovery and then initiated a chain of Internet postings in newsgroup:
  - **Terje Mathisen**: posted a **test program** on the newsgroup
  - **Andreas Kaiser**: posted **numbers to reproduce the bug**
  - **Tim Coe** (FPU engineer): **reversed Intel's algorithm**, explained all the errors Kaiser had reported, and found **an instance of the worst case error** (**c=4195835/3145727**).
  - **Cleve Moler** (father of MATLAB): **posted a summary of FDIV bug** to both Intel and MATLAB newsgroup.
  - November 22, CNN interviewed Moler, and then led off a story about Intel's troubles. November 24, newspapers (New York Times, Boston Globe, etc.) ran stories.

# Pentium FDIV: the Processor Bug that Shook the World (3/5)

- On December 20, Intel issued an apology and announced a no-questions-asked return policy on Pentium chips, which costed Intel half a billion dollars.
- Intel employed SRT algorithm to calculate the **intermediate quotients** for FDIV, which used a **lookup table**. However, due to a programming error, five entries were not downloaded into the programmable logic array (**PLA**).



The Pentium lookup table

# Pentium FDIV: the Processor Bug that Shook the World (4/5)

---

- Cleve Moler, Tim Coe and Terje Mathisen devised a scheme where a Pentium FDIV hardware could be replaced by software that insured the correct division.
- The idea is that the divisor of each prospective division operation would be checked for the presence of certain bit patterns in the floating point fraction that made it “at risk”. When an at risk divisor and the corresponding dividend are both **scaled by 15/16**, the quotient remains unchanged, but the operation can then be done safely by the FDIV.

The algorithm is:

```
function z = (x,y)
if at_risk(y)
    x = (15/16)*x;
    y = (15/16)*y;
end
z = x/y;
```

The safety filter is:

```
function a = at_risk(y)
f = and(hex(y), '000FF00000000000');
a = any(f == ['1F'; '4F'; '7F'; 'AF'; 'DF'])
```

**Pentium-Safe Division**

from “A Tale of Two Numbers”  
written by Cleve Moler

# Pentium FDIV: the Processor Bug that Shook the World (5/5)

---

- They worked with a group at Intel, in an effort to provide the **workaround** to compiler writers and major software vendors, and to announce its availability on the Internet.
- MATLAB was the proof-of-concept for the software workaround and released a special “Pentium Aware” MATLAB.
- At the time, MathWorks had just reached its 10th anniversary. The company name was barely known in the industries, and completely unheard of in the public generally. So when a press release arrived saying **this obscure little company in Massachusetts has fixed the Pentium bug, it created quite a stir.**

**With the Pentium, there is a very small chance of making a very large error, **so do we!****



# Assignment

---

## Reading:

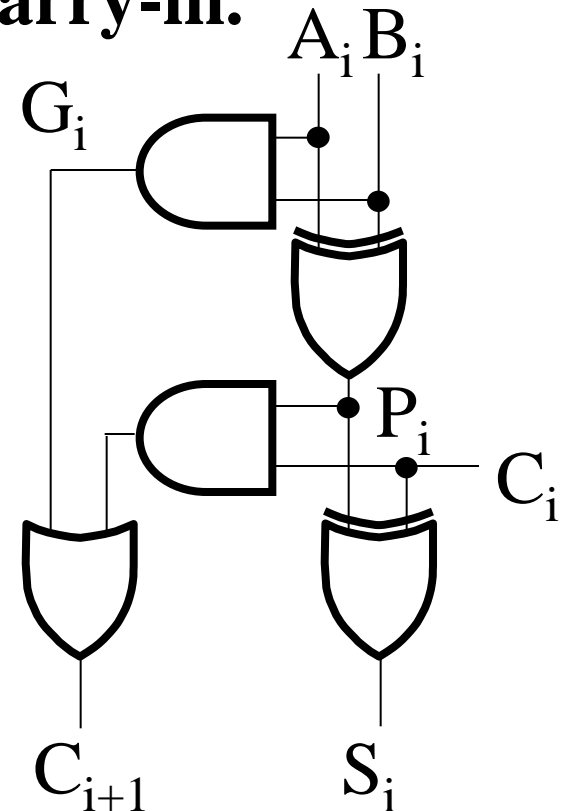
■ 3.8-3.12

## Problem assignment:

■ 3-50; 3-51; 3-52; 3-59

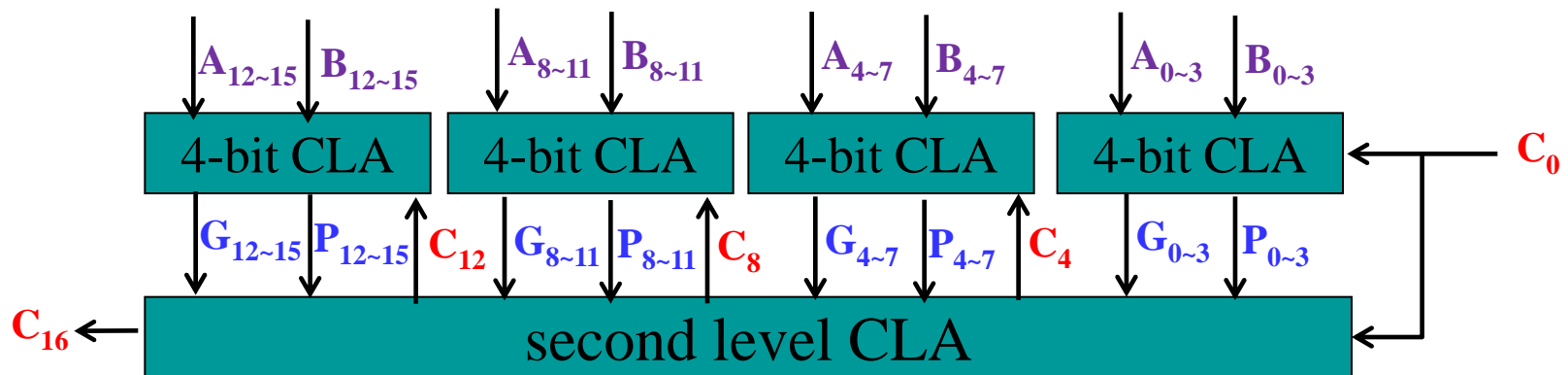
# Appendix A: Carry **Lookahead**

- 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:



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

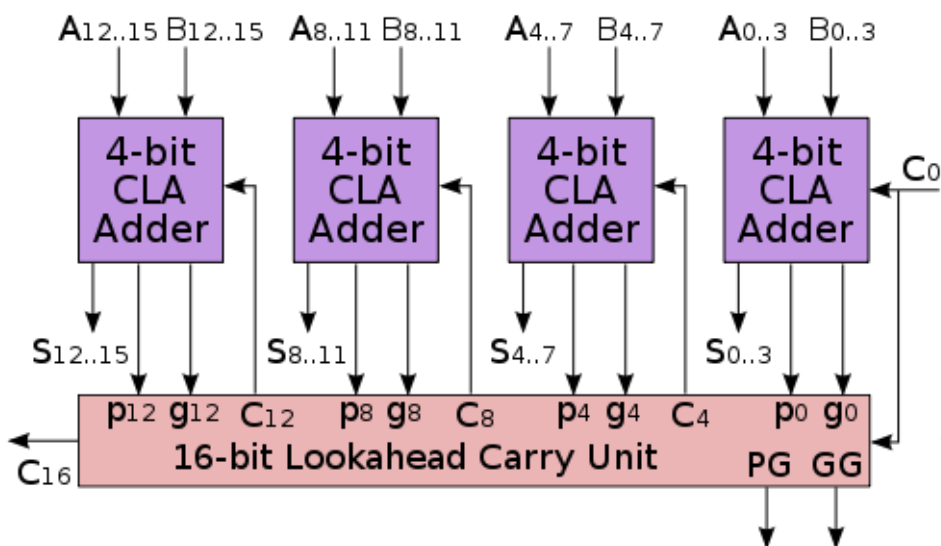
- $C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 = G_{0\sim3} + P_{0\sim3}C_0$
- $C_8 = G_7 + P_7G_6 + P_7P_6G_5 + P_7P_6P_5G_4 + P_7P_6P_5P_4C_4 = G_{4\sim7} + P_{4\sim7}C_4$
- $C_{12} = G_{11} + P_{11}G_{10} + P_{11}P_{10}G_9 + P_{11}P_{10}P_9G_8 + P_{11}P_{10}P_9P_8C_8 = G_{8\sim11}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



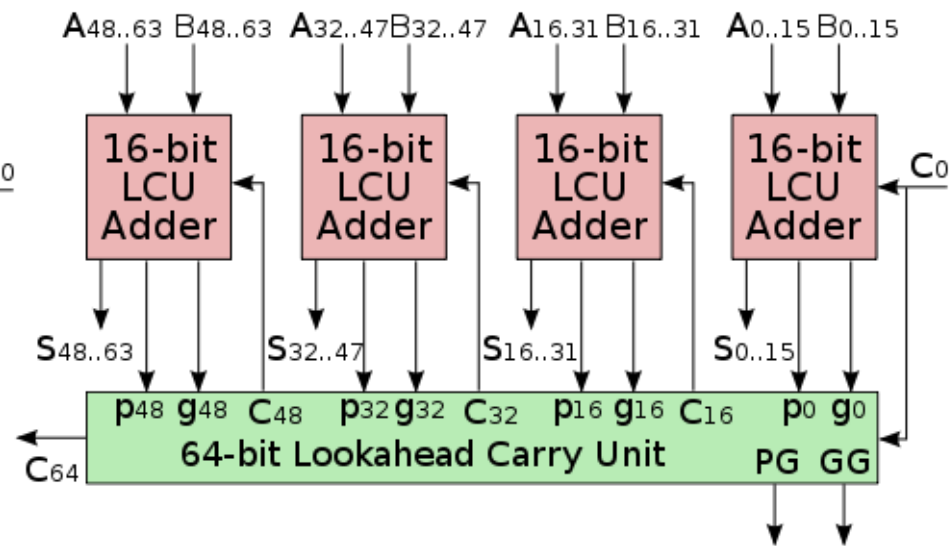
Note: Carry-in is generated by second level CLA, not individual adders!

# 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



16-bit CLA



64-bit CLA

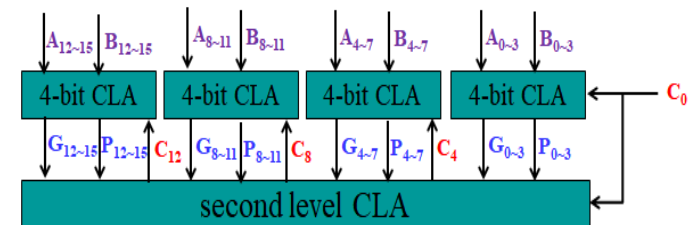
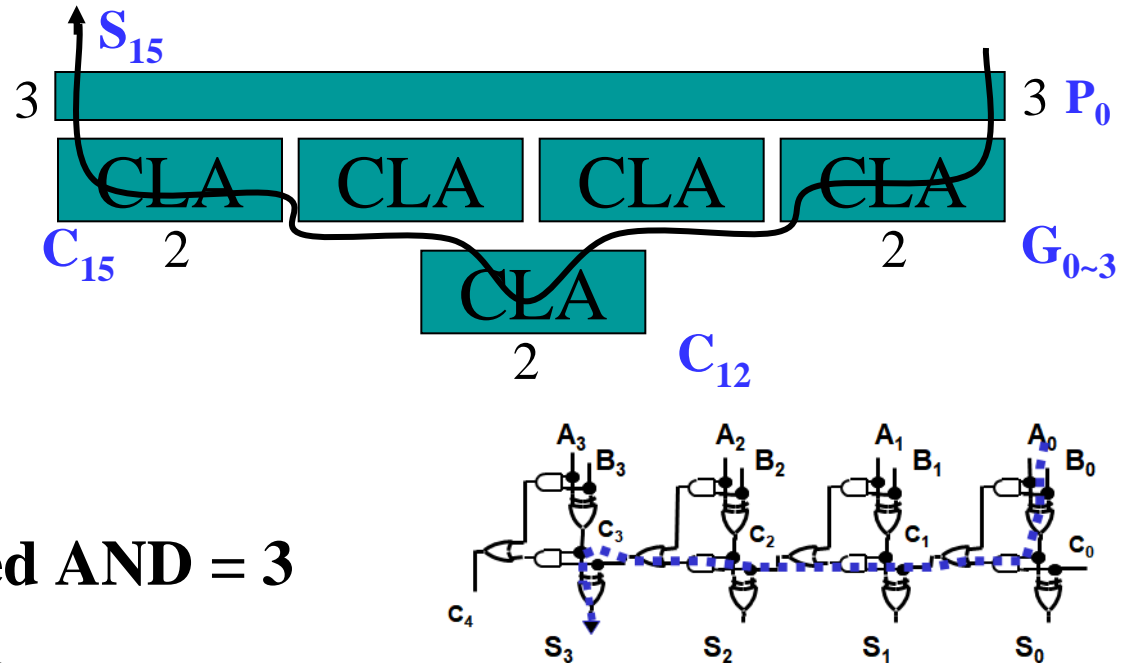
# Carry Lookahead Example

## ■ Specifications:

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

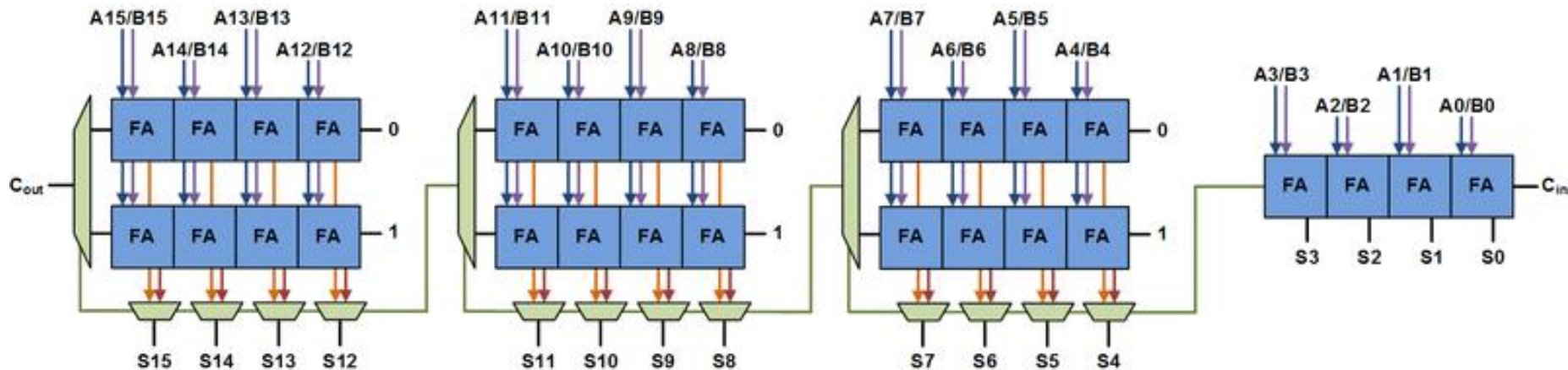
## ■ Longest Delays:

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



# Space and Time Tradeoffs for Adder

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



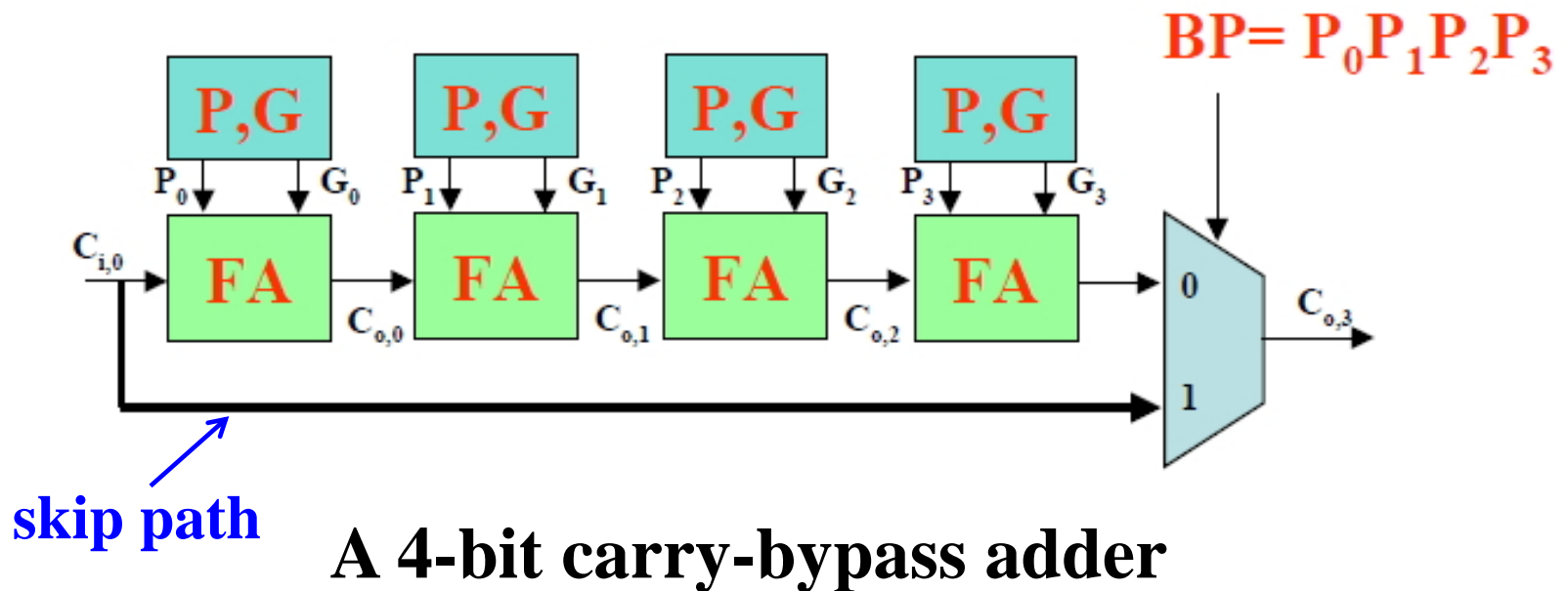
A 16-bit carry-select adder

# 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  $C_4 = C_0 \leftarrow \begin{cases} P=A \oplus B \\ G=AB \end{cases}$



# Appendix C: Another Usage of 2's Complement

---

<sup>1</sup>	01101100	Signed Integer		01101100
+	<u>10010100</u>	2's Complement	AND	<u>10010100</u>
	100000000			00000100

- To find the number of consecutive least significant zero bits of a number, e.g.,

01100010  $\longrightarrow$  1

01111000  $\longrightarrow$  3

10000000  $\longrightarrow$  7



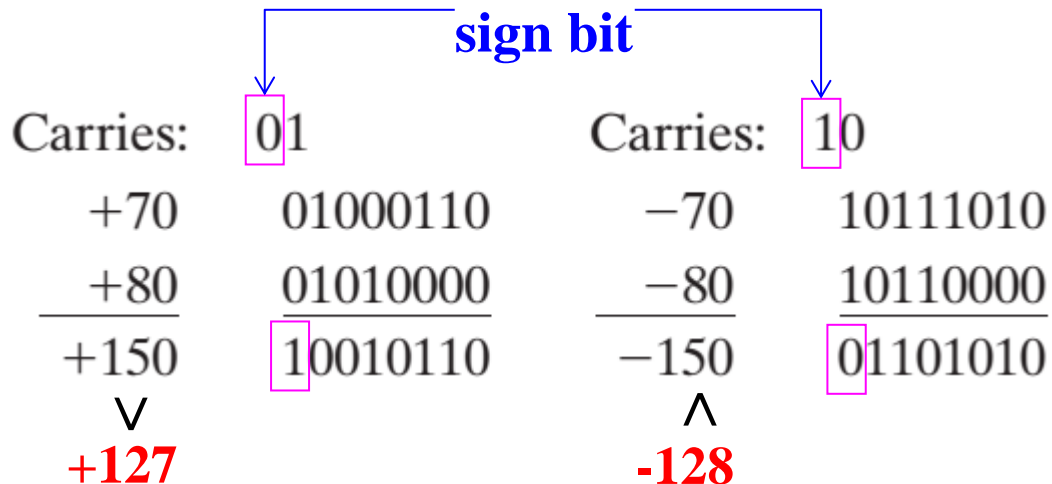
# Method of Complements (4/4)

- Digital systems have limited precision (e.g., 8 bits), therefore they are modulo  $2^N$  system (e.g., mod  $2^8$ ).
- Complements allows addition and subtraction to be done in a same and simpler way.
- The following example subtracts 5 from 8 (mod  $2^4$ ).

Decimal	Binary	Complement	
8	1000	1000	
-5	-0101	+1011	• $10000 - 0101 = 1011$
—	—	—	• 1011 is complement of 0101
3	0011	0011	

## Facts about Signed 2's Complement (2/2)

- In 2's complement arithmetic, carry out does not indicate an overflow (e.g., sign extension).
- The range of an n-bit signed 2's complement
  - upper bound:  $2^{n-2} + 2^{n-3} \dots + 2^1 + 2^0 = 2^{n-1} - 1$
  - lower bound:  $-2^{n-1}$
- Examples



**$X_{n-1}$ ,  $Y_{n-1}$ , and  $S_{n-1}$   
indicate overflow:**

- **001**: > upper bound, overflow
- **110**: < lower bound underflow

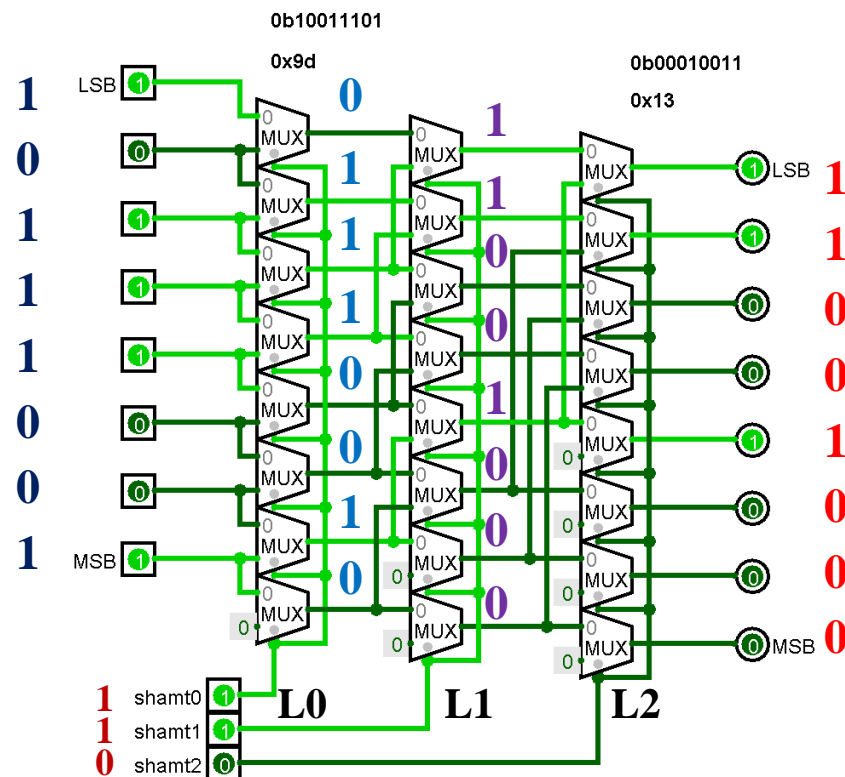
# Appendix C: Another Usage of 2's Complement

```
1  static unsigned long long deBruijn64 = 0x0218a392cd3d5dbf;
2
3  static const int deBruijnIdx64 [64] =
4  {
5      0, 1, 2, 7, 3, 13, 8, 19,
6      4, 25, 14, 28, 9, 34, 20, 40,
7      5, 17, 26, 38, 15, 46, 29, 48,
8      10, 31, 35, 54, 21, 50, 41, 57,
9      63, 6, 12, 18, 24, 27, 33, 39,
10     16, 37, 45, 47, 30, 53, 49, 56,
11     62, 11, 23, 32, 36, 44, 52, 55,
12     61, 22, 43, 51, 60, 42, 59, 58,
13 };
14
15 int find(unsigned long long b)
16 {
17     unsigned long long lsb = b & -b;
18
19     return (deBruijnIdx64[(lsb*deBruijn64) >> 58]);
20
21 }
```

- a De Bruijn sequence based algorithm

# Appendix D: 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

