

# DIGITAL SYSTEM DESIGN

**Sub Code: CSE\_ 2153**

**Dr. Rashmi R**  
**Assistant Professor**  
**MIT, Bengaluru**

# Module-2

## Arithmetic Circuits

❑ In This Module,

- **Why are arithmetic circuits so important**
- **Adders**
  - ✓ Adding two binary numbers
  - ✓ Adding more than two binary numbers
  - ✓ Circuits Based on Adders
- **Multipliers**
- **Functions that do not use adders**
- **Design of Arithmetic circuits**

## Motivation: Arithmetic Circuits

- ❑ **Core of every digital circuit**

- ✓ Heart of the digital system

- ❑ **Determines the performance of the system**

- ✓ Dictates clock rate, speed
- ✓ If arithmetic circuits are optimized performance will improve

- ❑ **Opportunities for improvement**

- ✓ Novel algorithms require novel combinations of arithmetic circuits.

# Addition

- Two types


- Unsigned Addition

- Signed Addition

# Addition of Unsigned Numbers

- The addition of 2 one-bit numbers entails four possible combinations
- Two bits are needed to represent the result of the addition.
- The right-most bit - *sum, s*.
- The left-most bit, which is produced as a *carry-out* when both bits being added are equal to 1, is called the *carry, c*.
- This circuit, which implements the addition of only two bits, - *half-adder*.

$x$	0	0	1	1
$+y$	$+0$	$+1$	$+0$	$+1$
<hr/>	<hr/>	<hr/>	<hr/>	<hr/>
$c\ s$	0 0	0 1	0 1	1 0

Carry  Sum

(a) The four possible cases

# Adder

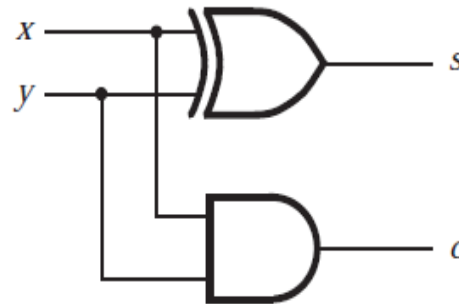
- It is an example of **iterative design**.
- Design a 1 bit adder circuit, then expand to n- bit adder
- Look at-
  - ❑ **Half adder** – which is a 2-bit adder
    - ❖ Inputs are bits to be added
    - ❖ Outputs: result and possible carry
  - ❑ **Full Adder** – includes carry in, 3- bit adder

# HALF-ADDER

- Half-Adder (2,2) Counter
- The *Half Adder* (HA) is the simplest arithmetic block
- It can add two 1-bit numbers, result is a 2-bit number
- Can be realized easily

$x$ $y$		Carry $c$	Sum $s$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(b) Truth table



(c) Circuit



(d) Graphical symbol

Half-adder.



$$\begin{array}{rcl}
 X = x_4x_3x_2x_1x_0 & 0\ 1\ 1\ 1\ 1 & (15)_{10} \\
 + Y = y_4y_3y_2y_1y_0 & 0\ 1\ 0\ 1\ 0 & (10)_{10} \\
 \hline
 & 1\ 1\ 1\ 0 & \leftarrow \text{Generated carries} \\
 \hline
 S = s_4s_3s_2s_1s_0 & 1\ 1\ 0\ 0\ 1 & (25)_{10}
 \end{array}$$

An example of addition

# FULL-ADDER

## ❑ Full-Adder (3,2) Counter

- The Full Adder (FA) is the *essential* arithmetic block
- It can add three 1-bit numbers, result is a 2-bit number
- There are many realizations both at gate and transistor level.
- Since it is used in building many arithmetic operations, the performance of one FA influences the overall performance greatly.

$c_i$	$x_i$	$y_i$	$c_{i+1}$	$s_i$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(a) Truth table

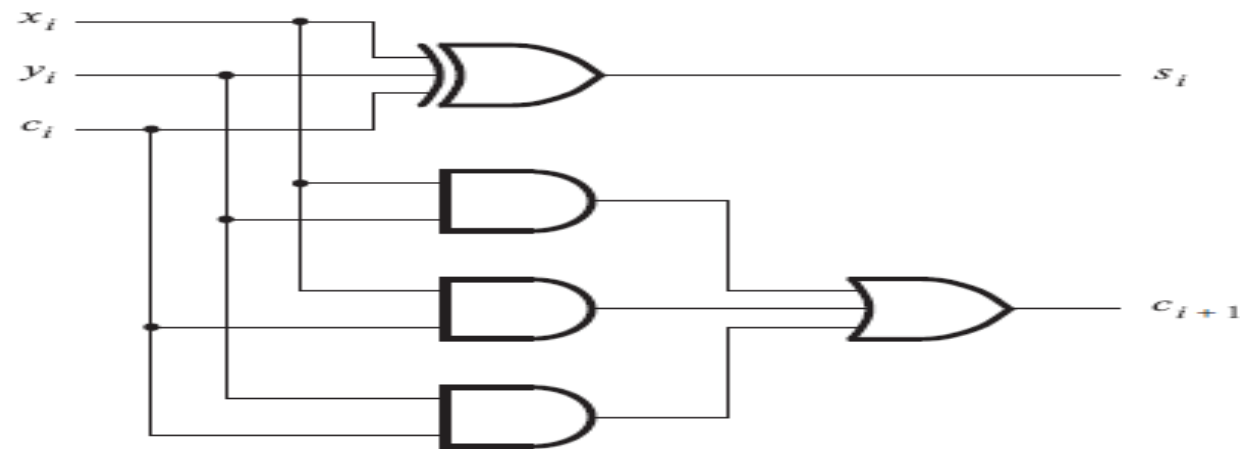
$c_i \backslash x_i y_i$				
	00	01	11	10
0		1		1
1	1		1	

$$s_i = x_i \oplus y_i \oplus c_i$$

$c_i \backslash x_i y_i$				
	00	01	11	10
0			1	
1		1	1	1

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

(b) Karnaugh maps



(c) Circuit

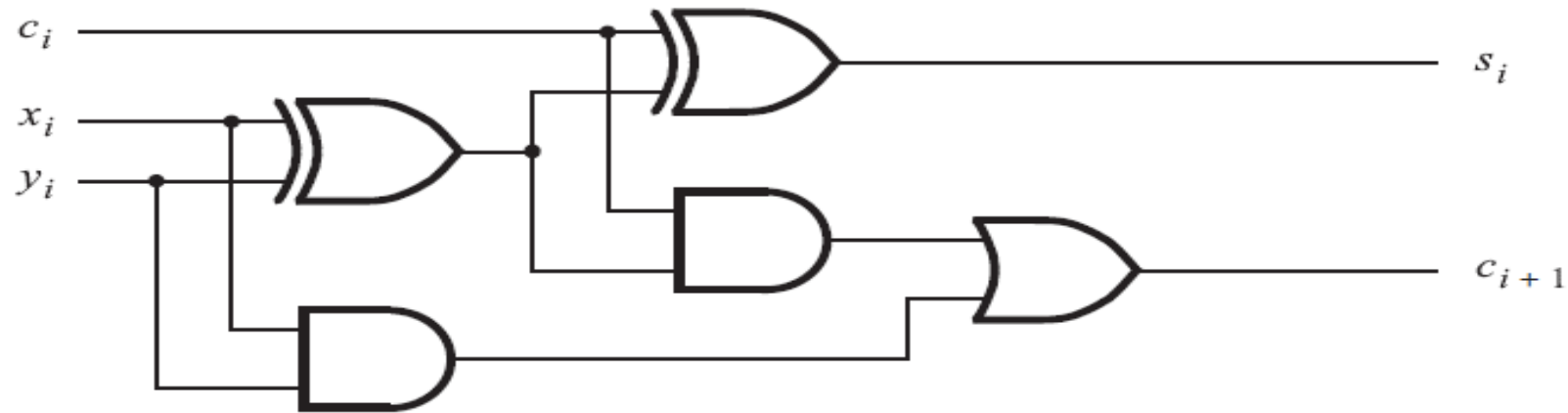
Full-adder

# Decomposed Full-Adder

- It uses two half-adders to form a full-adder.



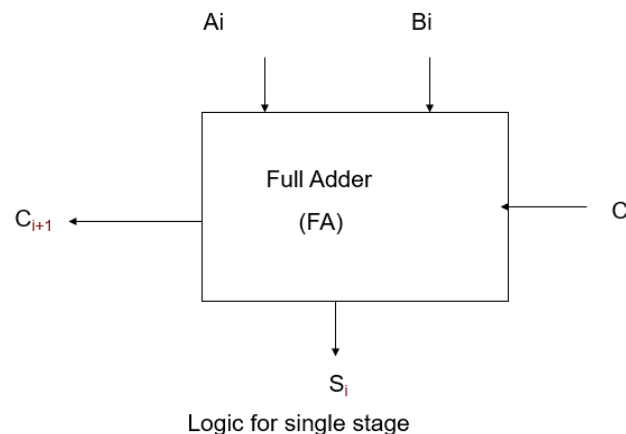
(a) Block diagram



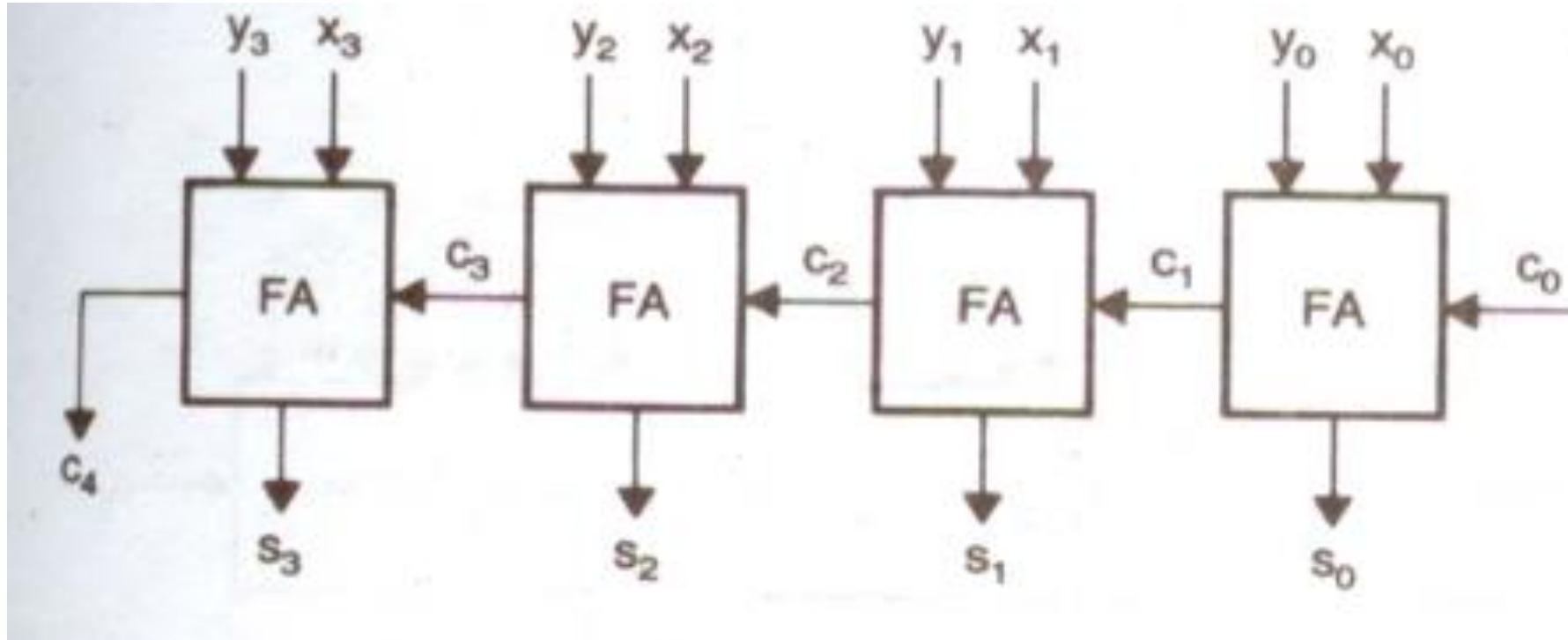
(b) Detailed diagram

# Ripple-Carry Adder

- The operands  $A_i$  and  $B_i$  are applied as inputs to the adder, it takes time before the output sum,  $S_i$ , is valid. Each full-adder introduces a certain delay before its  $s_i$  and  $c_{i+1}$ . Outputs are valid. Delay be denoted as  $t$ .
- Carry-out from the first stage,  $c_1$ , arrives at the second stage  $t$  after the application of the  $A$  and  $B$  inputs.
- The carry-out from the second stage,  $c_2$ , arrives at the third stage with a  $2t$  delay.
- The signal  $c_{n-1}$  is valid after a delay of  $(n - 1)t$ , which means - the complete sum is available after a delay of  $nt$ . Because of the way the carry signals “**ripple**” through the full-adder stages, the circuit is called a **ripple-carry adder**.
- The delay incurred to produce the final sum and carry-out in a ripple-carry adder depends on the size of the numbers.



## Cascade of 4-bit adders



**b. Four Full Adders are Cascaded to implement a 4-bit Ripple-Carry Adder**

# Delay Calculation of Ripple Carry Adders

- The time requires for generation of sum and carry in Full-Adder is 3 and 2 .

- If 4-bit ripple carry adder is considered then the total delay will be 9

- { time requires to generate  $c1 = 2$

$$c2 = 2$$

$$c3 = 2$$

- For  $c4$  and sum  $s3$  is = 3 units }

- In general, for addition of n-bits time units require is n delay.

# 16-bit adders using 4 bit adder and delay calculation

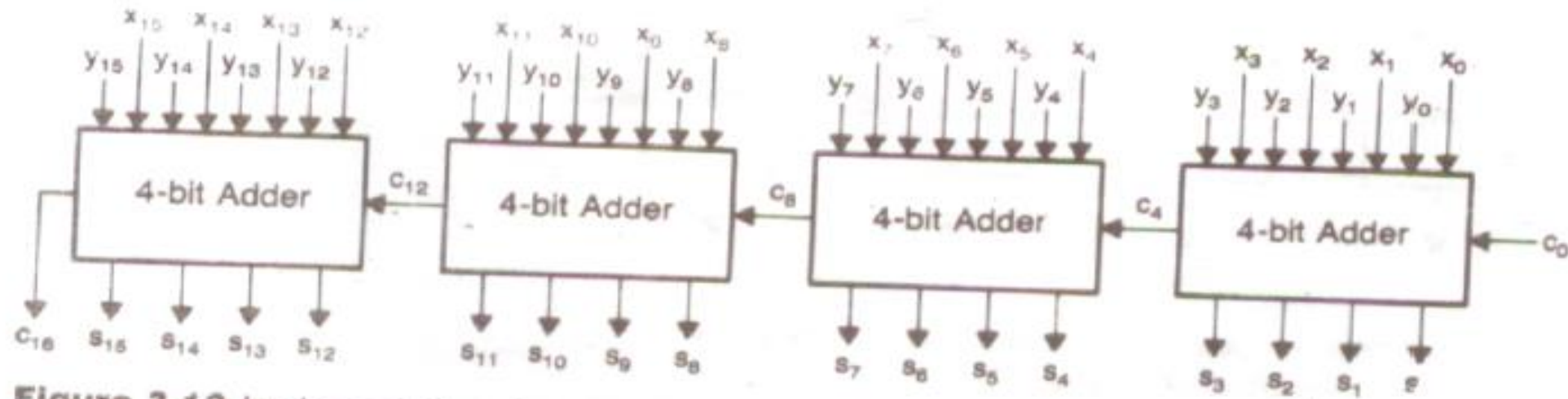


Figure 2.13

To generate  $S_{15}$ ,  $C_{15}$  must be available. The generation of  $C_{15}$  depends on the availability of  $C_{14}$ , which must wait for  $C_{13}$  to become available. In the worst case, the carry propagates through 15 full adders. Therefore, the worst-case add-time of the 16-bit CPA can be estimated as follows:

Time taken for carry to propagate  
through 15 full adders (the delay  
involved in the path from  $C_0$  to  $C_{15}$ ) =  $15 * 2\Delta$

Time taken to generate  $S_{15}$  from  $C_{15}$  =  $3\Delta$

$33\Delta$



# Hierarchical 4-Bit Adder

- We can easily use hierarchy here
- Design half adder
- Use in full adder
- Use full adder in 4-bit adder

# ADDITION AND SUBTRACTION

- It is a basic operation in computer arithmetic.
- Multiplication and division are based on addition and subtraction.
- Adders / subtractors are desirable not only for speed up fundamental operations, but also for accelerating the multiplication and division which involves massive addition and subtraction .

# NEGATION

- Given a number  $A$ , the negation operation finds  $-A$ , that is the complement number of  $A$ .
- If  $A$  is a positive number, after negation we have a negative number.

Conversely, if  $A$  is a negative number then negation will result in a positive number.

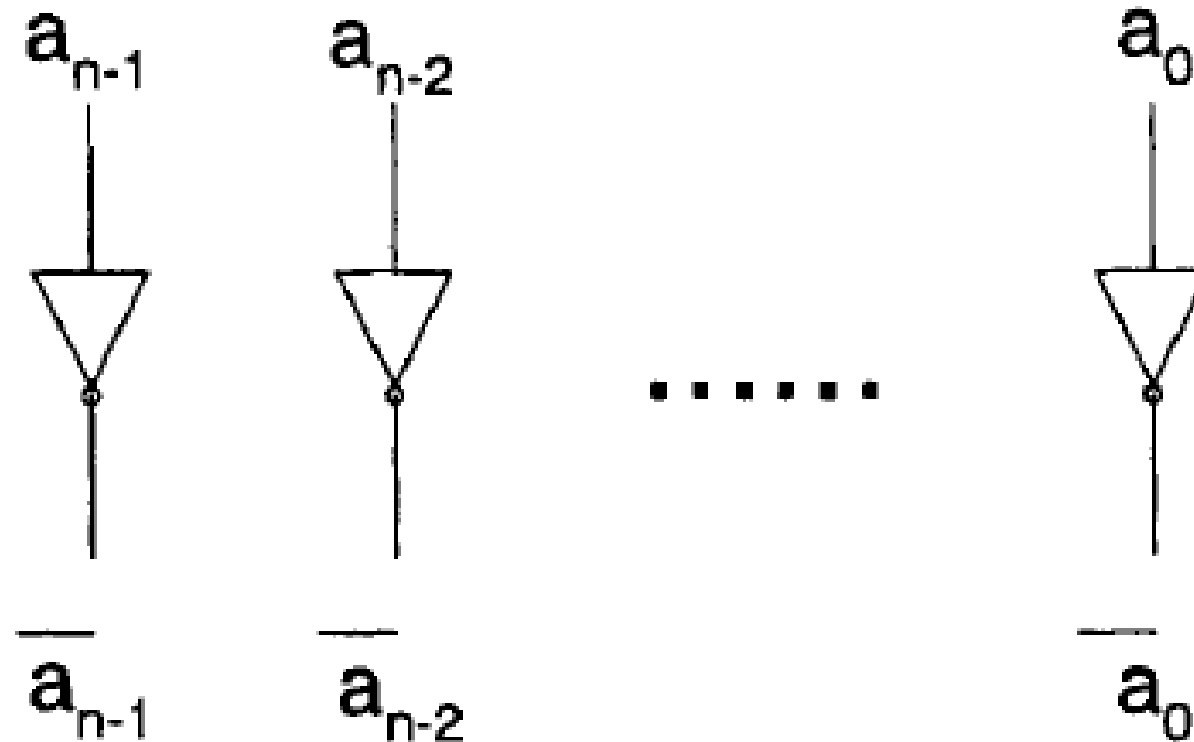
## Negation in One's Complement System

<b>E</b>	<b>a<sub>i</sub></b>	<b>a'</b>
<b>0</b>	<b>0</b>	<b>0</b>
<b>0</b>	<b>1</b>	<b>1</b>
<b>1</b>	<b>0</b>	<b>1</b>
<b>1</b>	<b>1</b>	<b>0</b>

# 1. Negation of One's Complement System

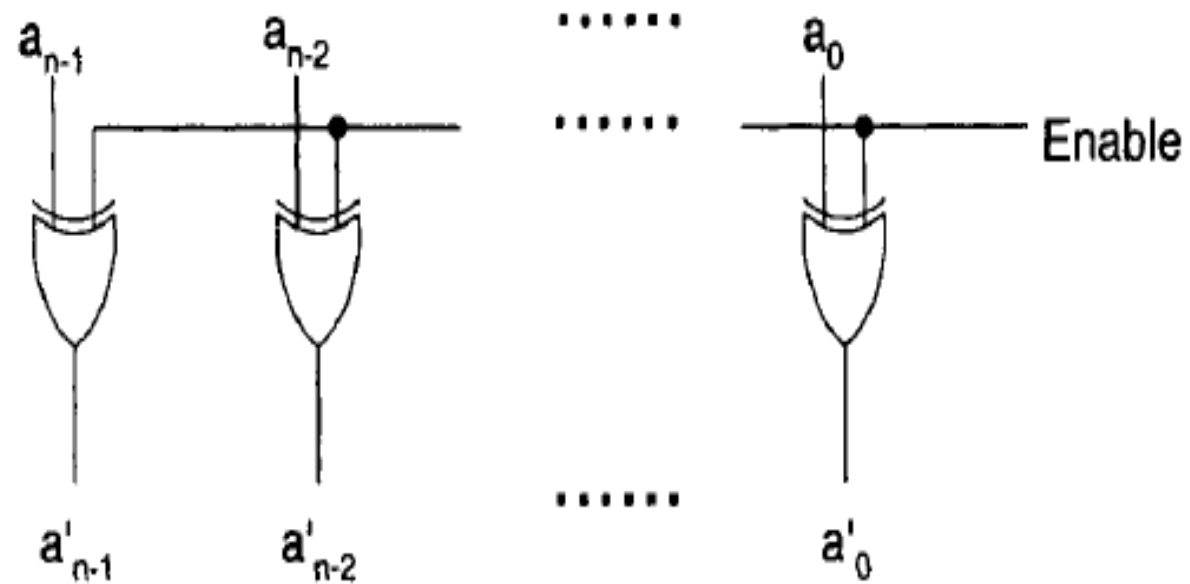
In the 1's complement system we perform a bit-wise NOT function to obtain a negative number of a given number, that is, change 0 to 1 and change 1 to 0. The circuit shown in Figure (a) conducts the negation of an n-bit number in the 1's complement system.

## Realization by NOT gates



(a) Realizing Negation by NOT Gates

$E$	$a_i$	$a'_i$
0	0	0
0	1	1
1	0	1
1	0	0



(b) Negation by XOR Gates

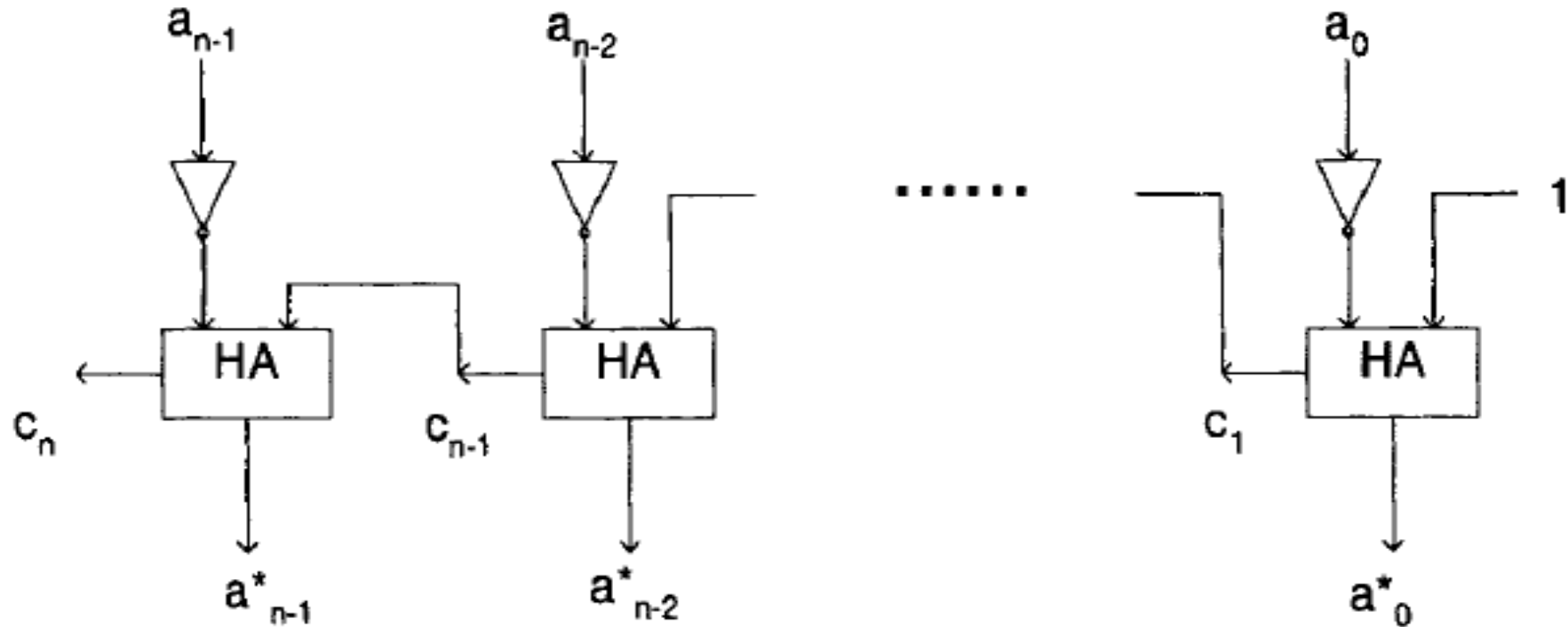
## 2. Negation in two's Complement System

### □ First method

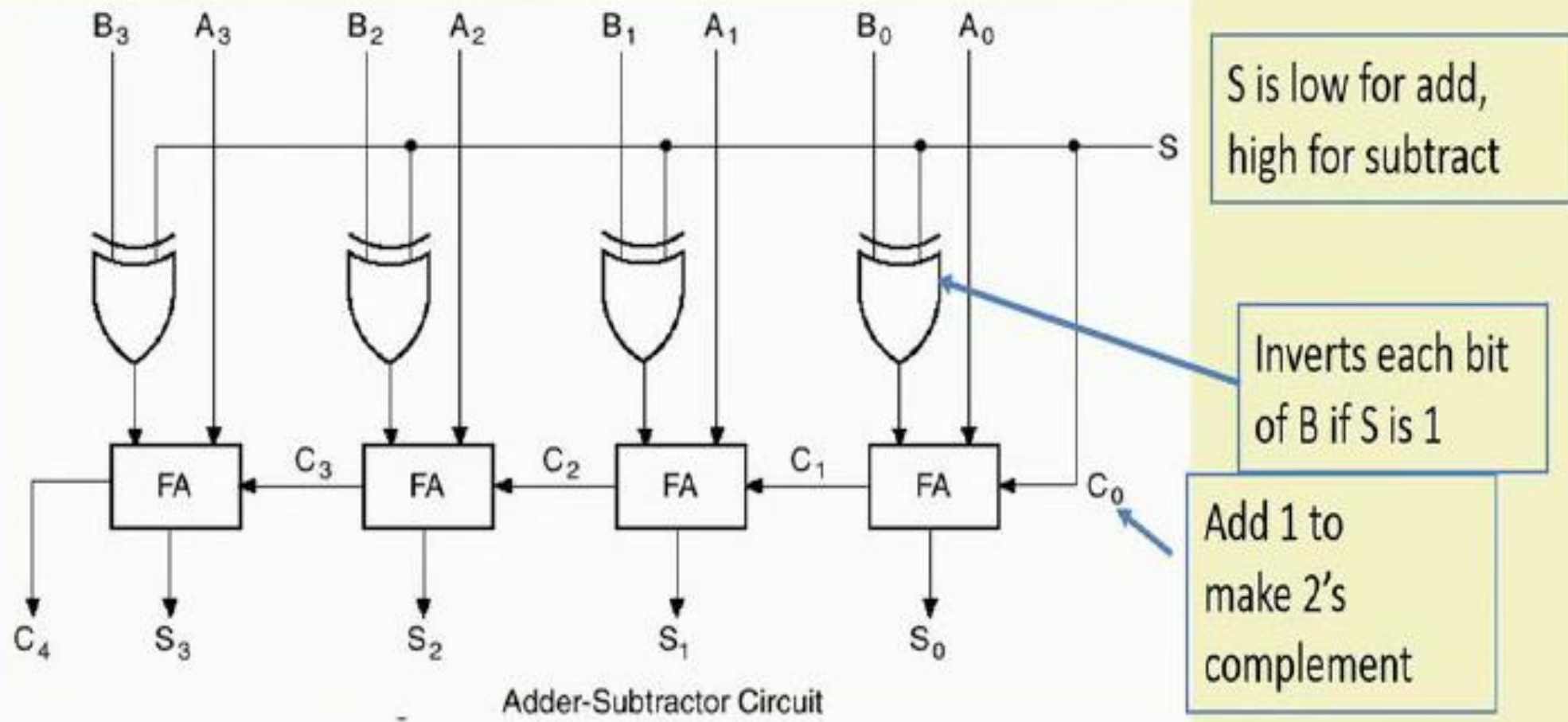
- Find the 1's complement number of it and then increase by 1 using an n-bit half adder.
- Note that there is no carry-in but a “1” to add on for the LSB position,



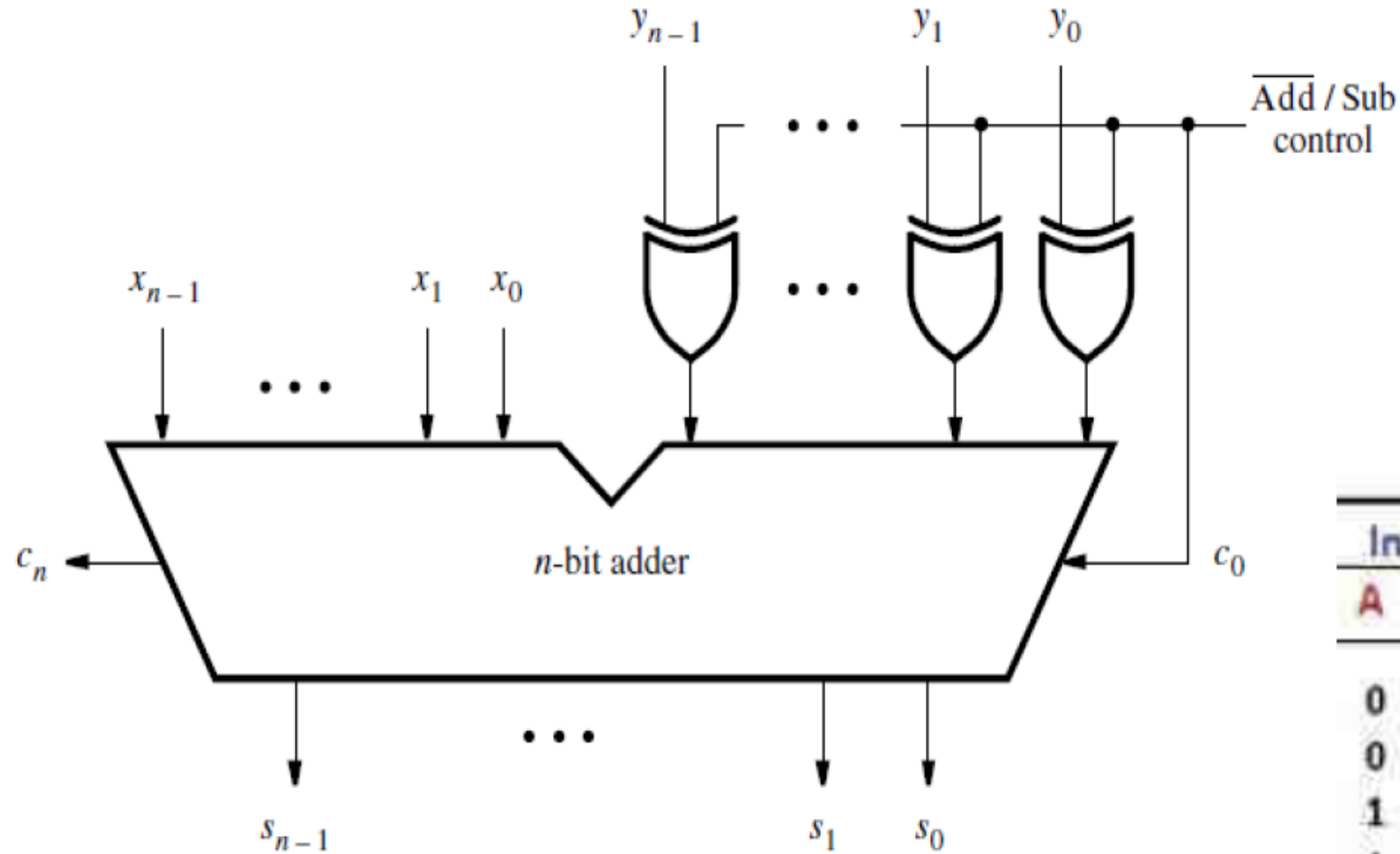
## 2. Negation in two's Complement System



## Adder-Subtractor Design: $(A+B)$ OR $(A-B)$



# Adder and Subtractor Unit

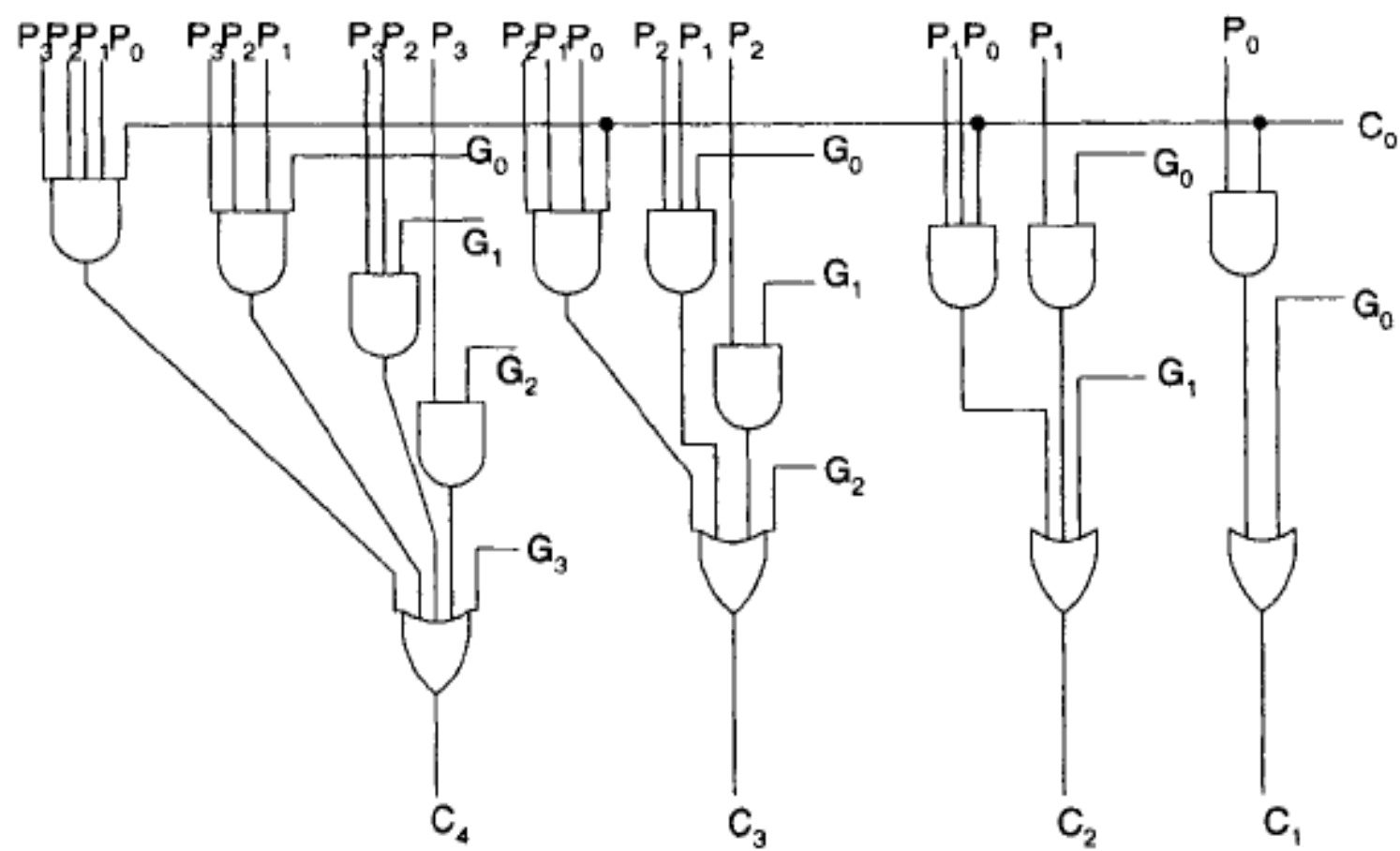


Inputs		Output
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

- When Add/Sub = 0, it performs  $\mathbf{X + Y}$ .
- When Add/Sub = 1, it performs
- $\mathbf{X + Y' + 1}$
- $\mathbf{= X + 2's\ complement\ of\ Y}$
- $\mathbf{= X - Y}$

# Fast Adders-Carry-Lookahead Adder

- In the ripple-carry adder, the carries in different bit positions are generated sequentially.
- That is,  $C_{i+1}$  is dependent on  $C_i$ , and  $C_i$  cannot be determined unless  $C_{i-1}$  is known.
- Carry – lookahead adder is an adder which can generate all the carries in parallel. No carry propagation is in the *cause of delay*.



(b) Carry-Lookahead Unit

$$P^* = P_0 P_1 \cdots P_{m-1}$$

$$G^* = G_{m-1} + G_{m-2}P_{m-1} + G_{m-3}P_{m-2}P_{m-1} + \cdots \\ + G_1P_2P_3 \cdots P_{m-1} + G_0P_1P_2P_3 \cdots P_{m-1}.$$

$P^* = 1$  if the block can propagate a carry.

$G^* = 1$  if the block can generate a carry.

# Carry look ahead adder

stage. For this reason, the function  $P_i$  is often referred to as *carry-propagate function*. Using  $G_i$  and  $P_i$ ,  $C_1$ ,  $C_2$ ,  $C_3$ , and  $C_4$  can be expressed as follows:

$$C_1 = G_0 + P_0C_0$$

$$C_2 = G_1 + P_1C_1$$

$$C_3 = G_2 + P_2C_2$$

$$C_4 = G_3 + P_3C_3$$



# Carry look ahead adder

$$C_1 = G_0 + P_0C_0$$

$$C_2 = G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0$$

$$C_3 = G_2 + P_2C_2 = G_2 + P_2(G_1 + P_1G_0 + P_1P_0C_0)$$

$$= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$$

$$C_4 = G_3 + P_3C_3 = G_3 + P_3(G_2 + P_2G_1 + P_2P_1G_0 + P_1P_1P_0C_0)$$

$$= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$$

## Equation..

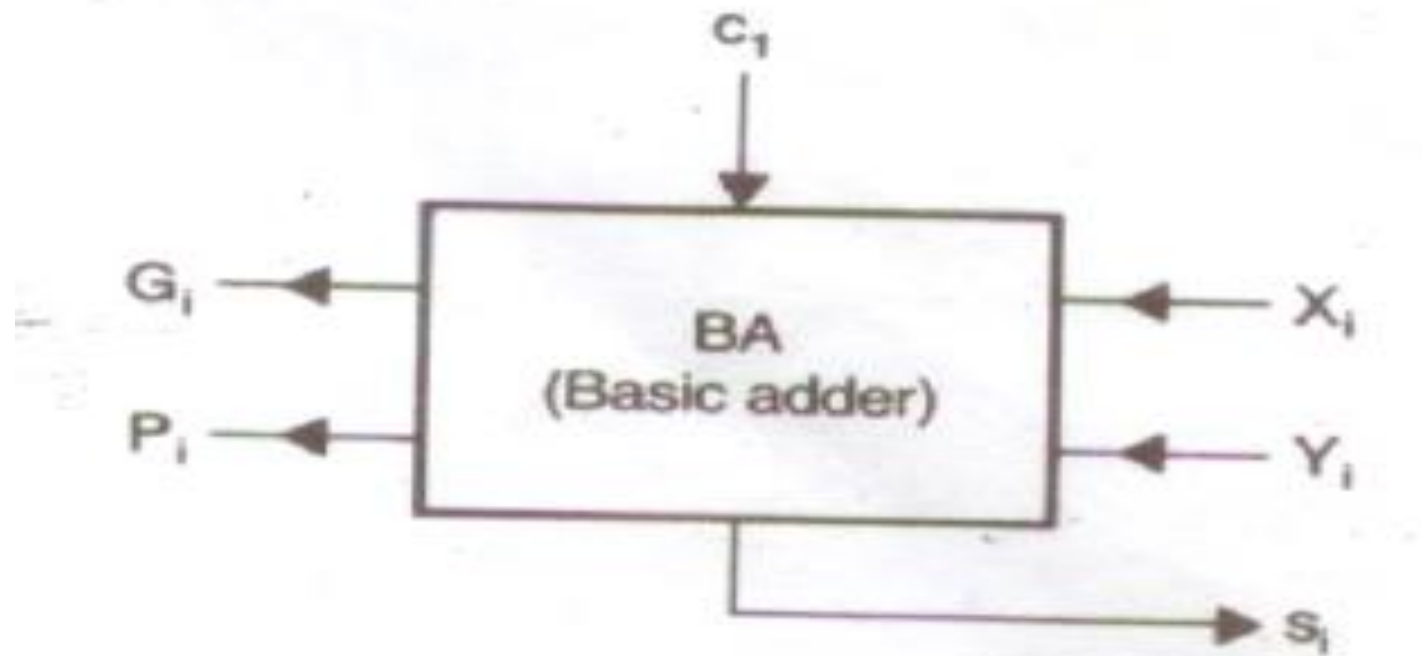
$$\begin{aligned}C_4 &= G_3 + P_3 C_3 = G_3 + P_3 (G_2 + P_2 G_1 + P_2 P_1 G_0 + P_1 P_1 P_0 C_0) \\&= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0\end{aligned}$$

This result suggests that  $C_1$ ,  $C_2$ ,  $C_3$ , and  $C_4$  can be generated directly from  $C_0$ . For this reason, these equations are called *carry look-ahead equations*, and the hardware that implements these equations is called a *4-stage carry look-ahead circuit (4-CLC)*. The

# Carry look ahead adder

- A 4-clc (Carry look ahead Circuit) can be implemented using a two-level AND/OR gate
- Fan in may be constraint
- If all  $G_i, P_i$  are available ( $1 \leq i \leq 3$ ) then all  $C_i$  's can be generated with  $I$ ,  $1 \leq i \leq 4$  with two gate delays..
- Output  $g_0, p_0$  are useful to obtain higher order look ahead system.

## CLC- basic cell



**Figure 3.15** Basic CLA Cell

# Arithmetic Overflow

- The result of addition or subtraction is supposed to fit within the significant bits used to represent the numbers.
- If  $n$  bits are used to represent signed numbers, then the result must be in the range  $-2^{n-1}$  to  $2^{n-1} - 1$ .
- If the result does not fit in this range, then we say that *arithmetic overflow* has occurred.
- To ensure the correct operation of an arithmetic circuit, it is important to be able to detect the occurrence of overflow.

➤Figure 5.14 presents the four cases where 2's-complement numbers with magnitudes of 7 and 2 are added. Because we are using four-bit numbers, there are three significant bits , $b_{2-0}$ .

$\begin{array}{r} (+7) \\ + (+2) \\ \hline (+9) \end{array}$	$\begin{array}{r} 0111 \\ + 0010 \\ \hline 1001 \\ c_4 = 0 \\ c_3 = 1 \end{array}$	$\begin{array}{r} (-7) \\ + (+2) \\ \hline (-5) \end{array}$	$\begin{array}{r} 1001 \\ + 0010 \\ \hline 1011 \\ c_4 = 0 \\ c_3 = 0 \end{array}$
$\begin{array}{r} (+7) \\ + (-2) \\ \hline (+5) \end{array}$	$\begin{array}{r} 0111 \\ + 1110 \\ \hline 10101 \\ c_4 = 1 \\ c_3 = 1 \end{array}$	$\begin{array}{r} (-7) \\ + (-2) \\ \hline (-9) \end{array}$	$\begin{array}{r} 1001 \\ + 1110 \\ \hline 10111 \\ c_4 = 1 \\ c_3 = 0 \end{array}$

**Figure 5.14** Examples for determination of overflow.

- When the numbers have **opposite signs**, there is no overflow.
- But if both numbers have the **same sign**, the magnitude of the result is 9, which cannot be represented with just three significant bits; therefore, **overflow occurs**.
- The key to determining whether overflow occurs is the carry-out from the MSB position, called  $c_3$  in the figure, and from the sign-bit position, called  $c_4$ .
- The figure indicates that overflow occurs when these carry-outs have different values, and a correct sum is produced when they have the same value.
- Indeed, this is true in general for both addition and subtraction of 2's-complement numbers.

➤ The occurrence of overflow is detected by,

$$\begin{aligned}\text{Overflow} &= c_3 \bar{c}_4 + \bar{c}_3 c_4 \\ &= c_3 \oplus c_4\end{aligned}$$

➤ For  $n$ -bit numbers we have,

$$\text{Overflow} = c_{n-1} \oplus c_n$$

➤ So, overflow can be detected with the addition of one XOR gate.



# Array Multiplier

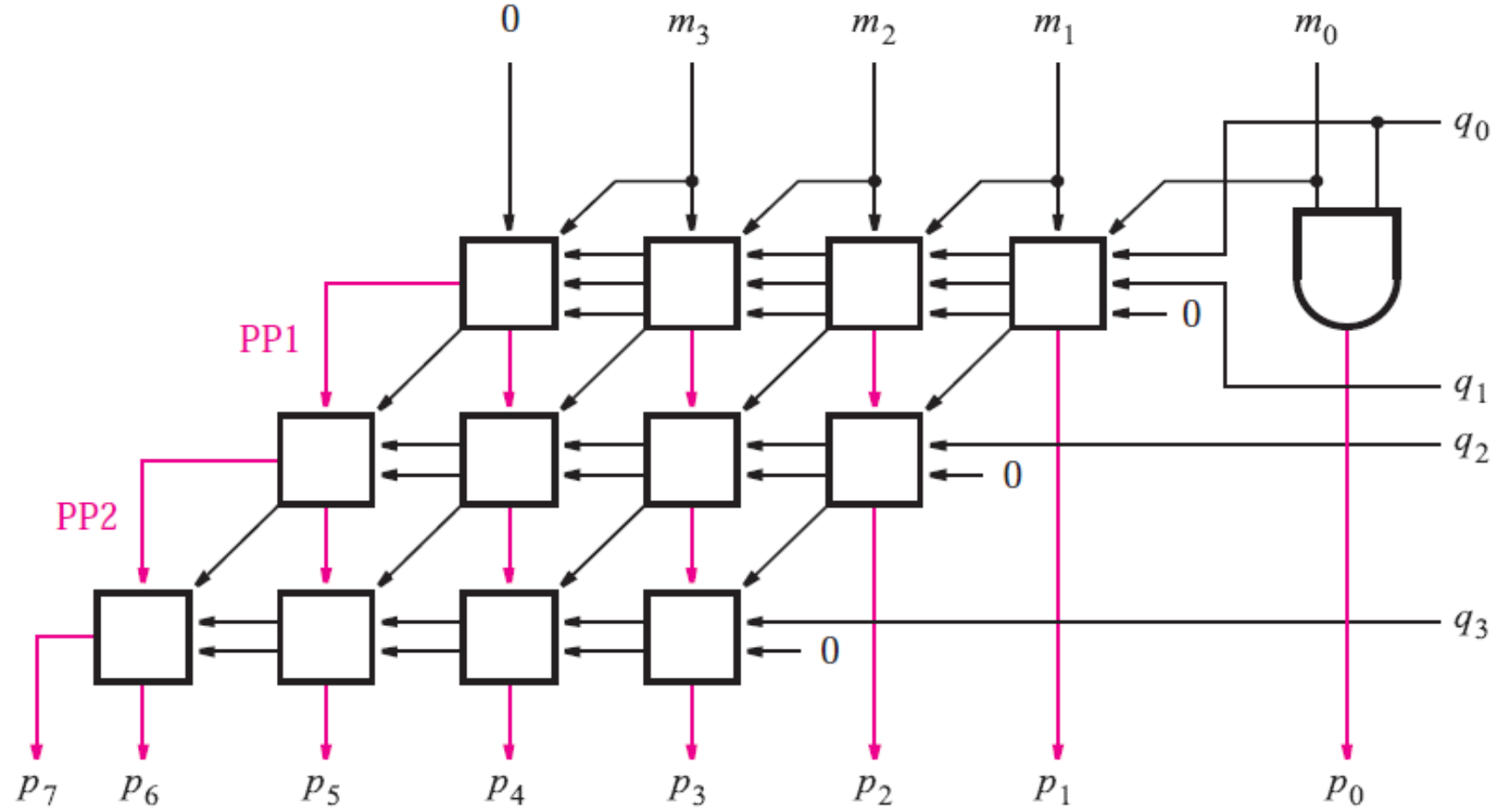
- Multiplication of binary numbers is performed in the same way as in decimal numbers.
- The **multiplicand is multiplied** by each bit of the multiplier starting from the least significant bit.
- Each such multiplication forms a partial product.
- Successive partial products are shifted one position to the left.
- The final product is obtained from the sum of the partial products.

Multiplicand M	(14)	1 1 1 0
Multiplier Q	(11)	× 1 0 1 1
		<hr/>
		1 1 1 0
		1 1 1 0
		0 0 0 0
		1 1 1 0
		<hr/>
Product P	(154)	1 0 0 1 1 0 1 0

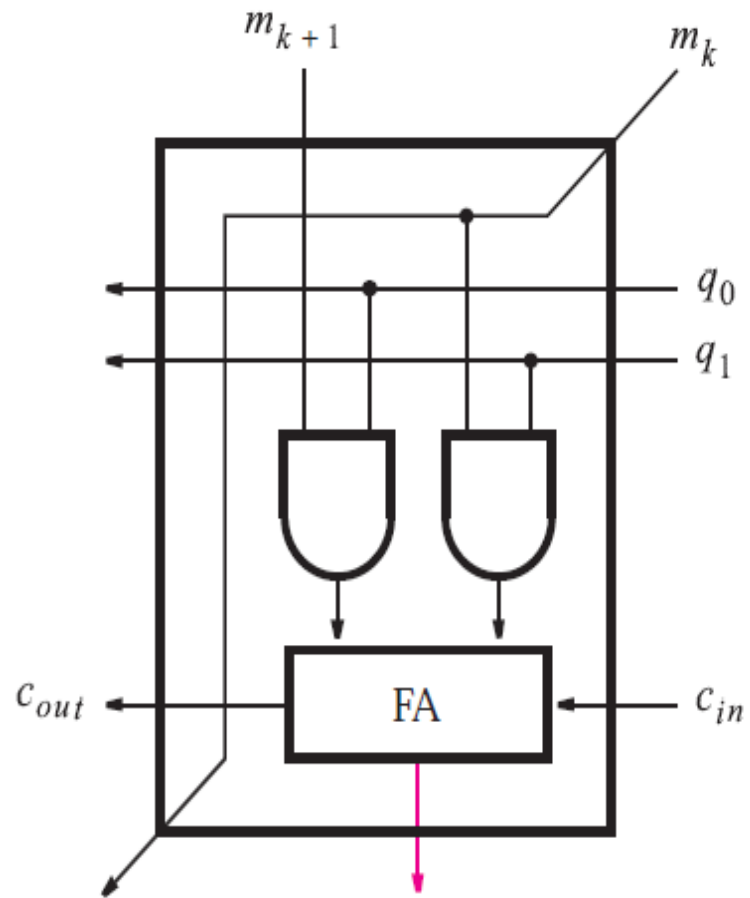
(a) Multiplication by hand

Multiplicand M	(11)	1 1 1 0
Multiplier Q	(14)	× 1 0 1 1
		-----
Partial product 0		1 1 1 0
		+ 1 1 1 0
		-----
Partial product 1		1 0 1 0 1
		+ 0 0 0 0
		-----
Partial product 2		0 1 0 1 0
		+ 1 1 1 0
		-----
Product P	(154)	1 0 0 1 1 0 1 0

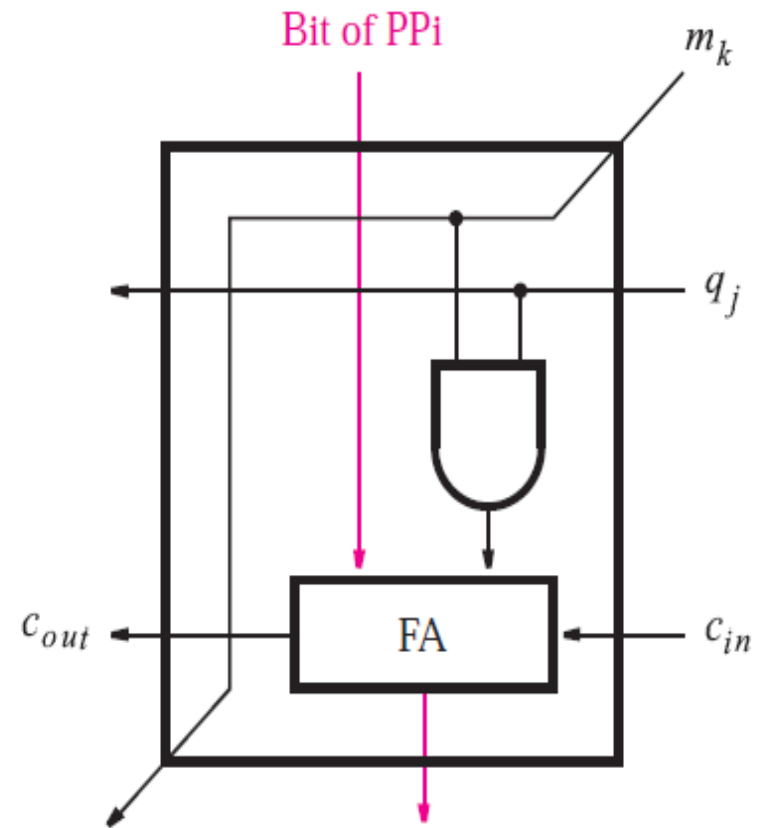
(b) Multiplication for implementation in hardware



(a) Structure of the circuit



(b) A block in the top row



(c) A block in the bottom two rows

A  $4 \times 4$  multiplier circuit

# Binary-Coded-Decimal Representation

**Table 3.3**

Binary-coded  
decimal digits.

Decimal digit	BCD code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

## BCD Addition

$$\begin{array}{r} X \\ + Y \\ \hline Z \end{array} \qquad \begin{array}{r} 0111 \\ + 0101 \\ \hline 1100 \\ + 0110 \\ \hline \text{carry} \rightarrow 10010 \\ \underbrace{\hspace{1.5cm}} \\ S = 2 \end{array} \qquad \begin{array}{r} 7 \\ + 5 \\ \hline 12 \end{array}$$

$$\begin{array}{r} X \\ + Y \\ \hline Z \end{array} \qquad \begin{array}{r} 1000 \\ + 1001 \\ \hline 10001 \\ + 0110 \\ \hline \text{carry} \rightarrow 10111 \\ \underbrace{\hspace{1.5cm}} \\ S = 7 \end{array} \qquad \begin{array}{r} 8 \\ + 9 \\ \hline 17 \end{array}$$

K	Binary Sum				BCD Sum					Decimal
	Z <sub>8</sub>	Z <sub>4</sub>	Z <sub>2</sub>	Z <sub>1</sub>	C	S <sub>8</sub>	S <sub>4</sub>	S <sub>2</sub>	S <sub>1</sub>	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

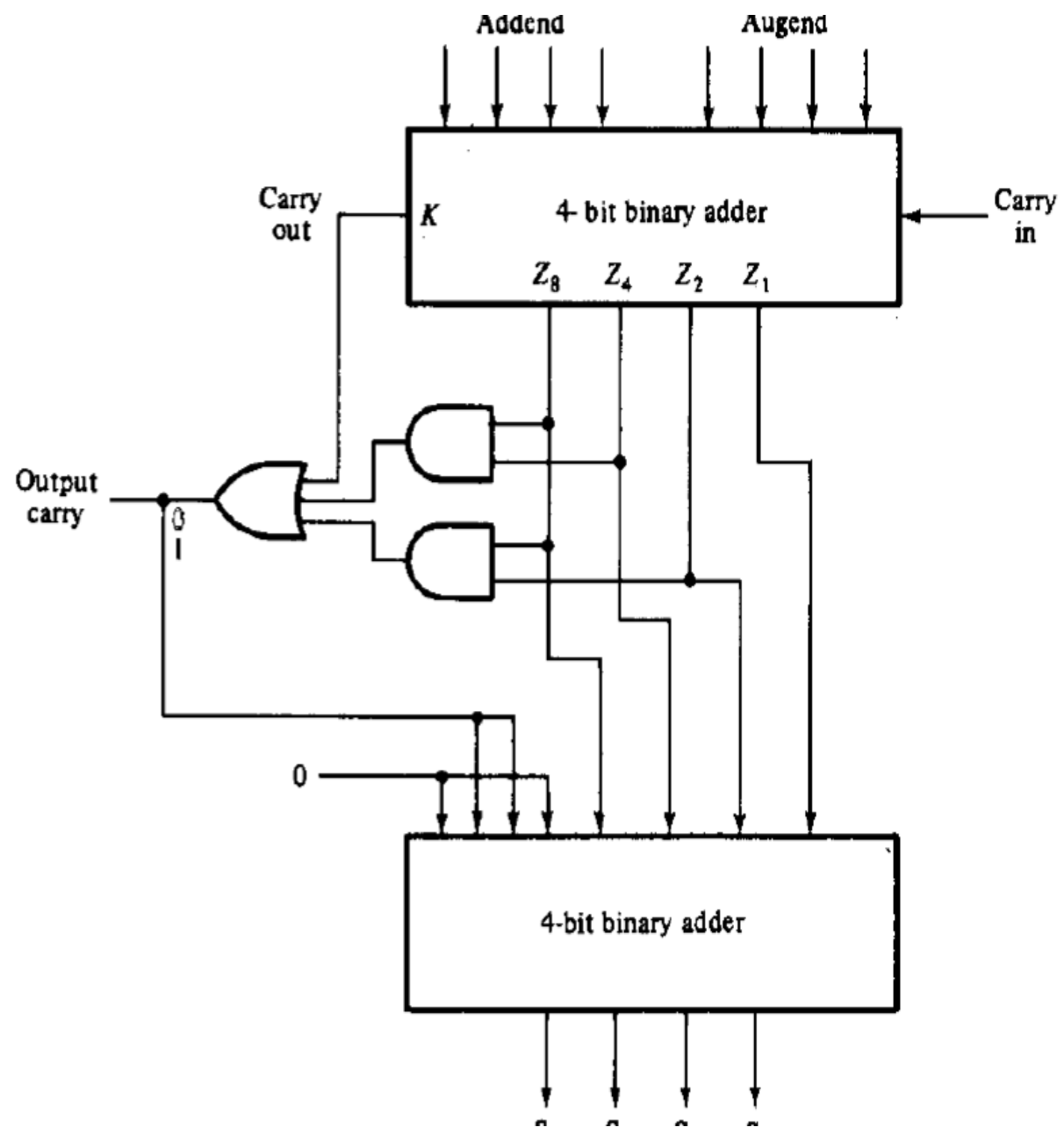
$z_2, z_1$		00	01	11	10
$z_8, z_4$	00	0	0	0	0
	01	0	0	0	0
	11	1	1	1	1
	10	0	0	1	1

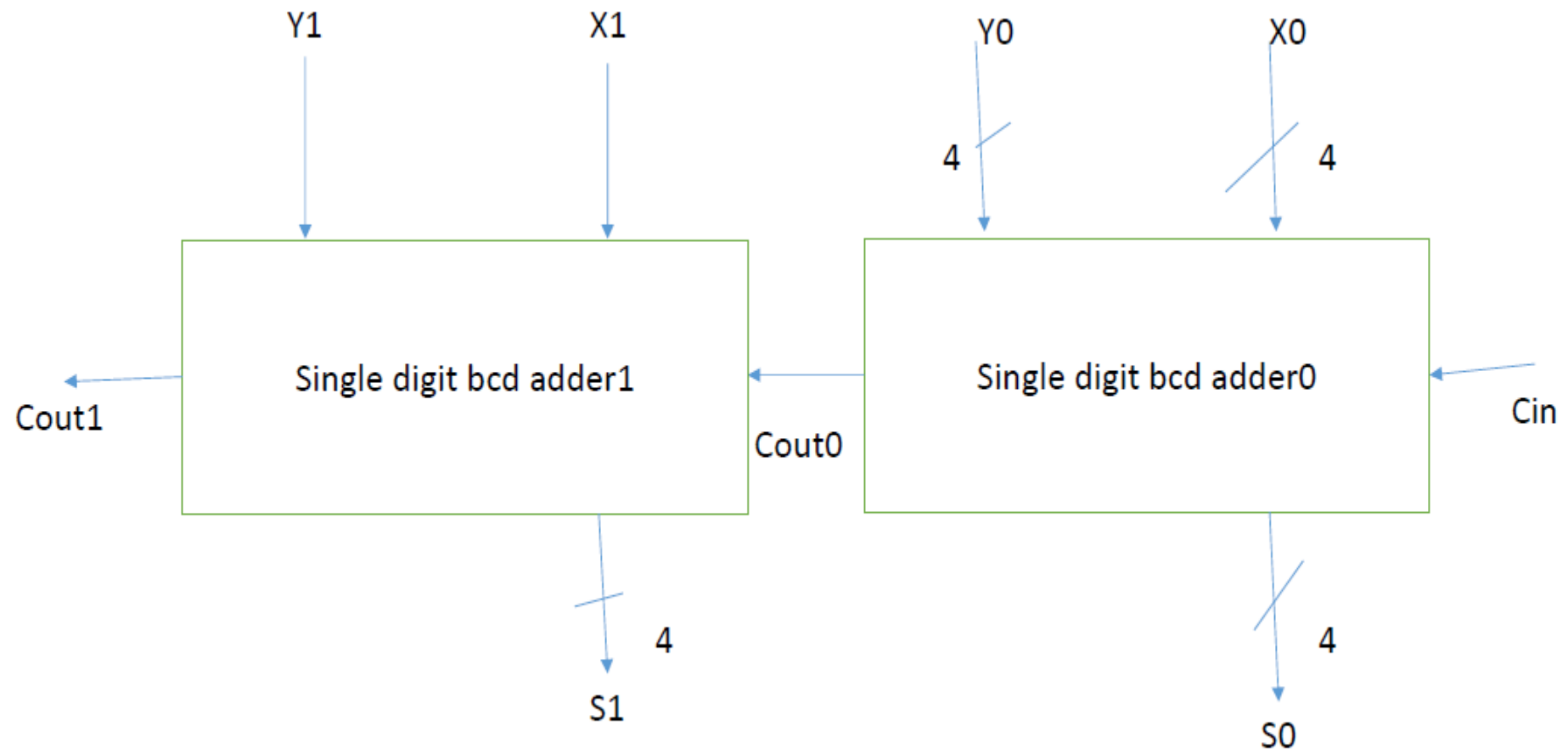
Correction should be done when the carry  $k = 1$  or when the expression  $z_8z_4 + z_8z_2$  evaluates to 1. Therefore the expression for the correction circuit is

$$C = K + Z_8Z_4 + Z_8Z_2$$



$$C = K + Z_8Z_4 + Z_8Z_2$$





# Design of Arithmetic Circuits using Verilog

## ➤ Representation of Digital Circuits in Verilog

## ➤ Three styles of modeling:

### 1. Dataflow Modeling:

- ✓ Executes a set of concurrent assignments- Parallel execution.
- ✓ Statements – expression

### 2. Behavioral Modeling:

- ✓ Set of sequential assignment statements.
- ✓ **Initial statement** – executes only once ( similar to dataflow expression)
- ✓ **Always statement** – executes multiple number of times

### 3. Structural Modeling:

- ✓ A larger circuit is defined by writing code that connects simple circuit elements together.

Write the Verilog code to implement a 4-bit adder.

```
module adder4 (carryin, x3, x2, x1, x0, y3, y2, y1, y0, s3, s2, s1, s0, carryout);  
    input carryin, x3, x2, x1, x0, y3, y2, y1, y0;  
    output s3, s2, s1, s0, carryout;  
    fulladd stage0 (carryin, x0, y0, s0, c1);  
    fulladd stage1 (c1, x1, y1, s1, c2);  
    fulladd stage2 (c2, x2, y2, s2, c3);  
    fulladd stage3 (c3, x3, y3, s3, carryout);  
endmodule  
  
module fulladd (Cin, x, y, s, Cout);  
    input Cin, x, y;  
    output s, Cout;  
    assign s = x ^ y ^ Cin;  
    assign Cout = (x & y) | (x & Cin) | (y & Cin);  
endmodule
```