

Digital Logic and Verilog

Lecture 13&14 & 15:
Adders and Verilog Basic 2

Xiaoping Huang
Fall 2019

Outline

Lecture 13 & 14 & 15:

Chapter: 3.2-3.5, 3.8, 3.7.3 Page 122-166 and 178-194

Addition of Unsigned Numbers

- ▶ 1-bit Addition (half adder and full adder)
- ▶ XOR gates
- ▶ Multi-bit Addition and subtraction
- ▶ Ripple-Carry Adder
- ▶ Fast Adders
- ▶ Multiplication of Unsigned Numbers(*)
- ▶ Verilog Basic 2 and Examples

3.2 Addition of Unsigned Number

1-Bit Addition

- ▶ Two 1-bit addends: $x, y \in \{0, 1\}$
 $\Rightarrow \text{sum} \in \{0, 1, 2\}$

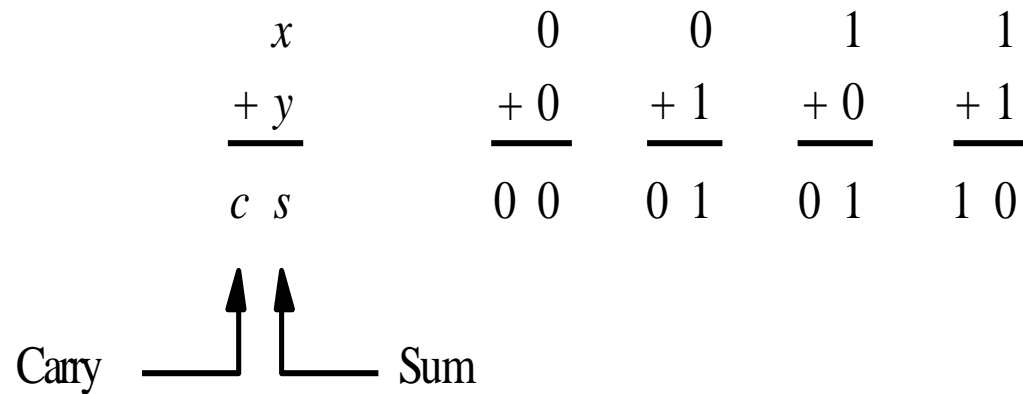
\therefore Need 2 bits to represent output

Low Bit: S (Sum)

High Bit: C (Carry)

C is useful when adding multi-bit numbers

1-Bit Addition



(a) The four possible cases

1-Bit Addition

▶ Truth Table

x	Y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

▶ SOP Realization

S

	y	0	1
x			
0		0	1
1		1	0

$$s = \bar{x} \cdot y + x \cdot \bar{y}$$

c

	y	0	1
x			
0		0	0
1		0	1

$$c = x \cdot y$$

1-Bit Addition

▶ Truth Table

x	Y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

▶ SOP Realization

S	x \ y	0	1
	0	0	1
	1	1	0

$$s = \bar{x} \cdot y + x \cdot \bar{y}$$

C	x \ y	0	1
	0	0	0
	1	0	1

$$c = x \cdot y$$



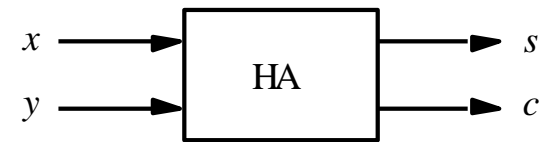
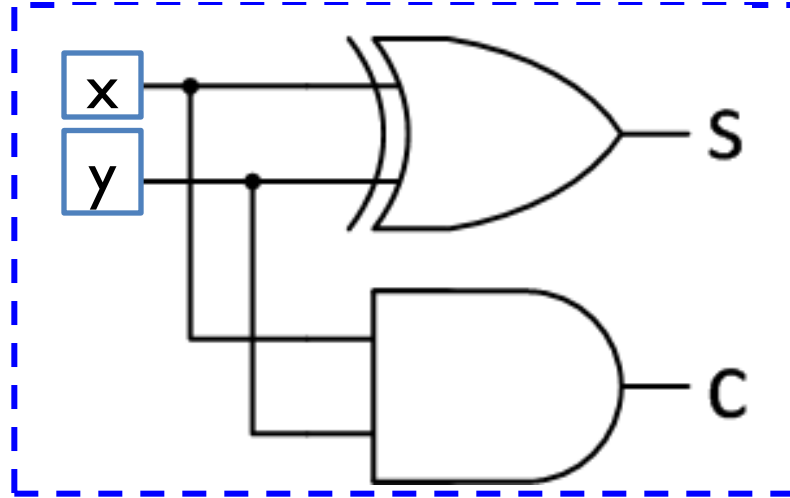
1-Bit Addition

$$s = \bar{x} \cdot y + x \cdot \bar{y}$$

$$c = x \cdot y$$

Note that this is just an exclusive-or operation

$$\Rightarrow s = x \oplus y$$



(d) Graphical symbol

HALF-ADDER:

- Uses only 2 gates
- Does not require inverted inputs

Multi-bit Addition

- ▶ To extend our results to larger inputs & outputs, consider using a half-adder to add each pair of bits.
- ▶ Ex:

$$\begin{array}{rcccc} & & \textcolor{blue}{|} & & \textcolor{blue}{|} \\ & & \textcolor{blue}{1} & & \textcolor{blue}{0} \\ & & \textcolor{blue}{1} & & \textcolor{blue}{0} \\ + & 0 & 1 & 1 & 0 \\ \hline 1 & 0 & 0 & 0 & 1 \end{array}$$

Multi-bit Addition

- Consider the addition of each pair of bit x_i and y_i separately

Generated carries \longrightarrow 1 1 1 0

$$\begin{array}{r}
 X = x_4x_3x_2x_1x_0 \quad 01111 \quad (15)_{10} \\
 + Y = y_4y_3y_2y_1y_0 \quad + 01010 \quad + (10)_{10} \\
 \hline
 S = s_4s_3s_2s_1s_0 \quad 11001 \quad (25)_{10}
 \end{array}$$

$$\begin{array}{cccc}
 \dots & c_{l+1} & c_l & \dots \\
 \dots & \dots & x_l & \dots \\
 \dots & \dots & y_l & \dots \\
 \hline
 \dots & \dots & s_l & \dots
 \end{array}$$

- $x_i, y_i, c_i, s_i, c_{i+1}$

- Problem:**

If a carry is generated at one stage, it must be added to the next stage. *No input available for this on half adder.*

Full Adder

- ▶ 2 (1-bit addends): $\mathbf{x_i}, \mathbf{y_i} \in \{0, 1\}$
- 1 (1-bit carry-in): $\mathbf{c_i} \in \{0, 1\}$
 $\Rightarrow \text{sum} \in \{0, 1, 2, 3\}$

still only need 2-bit output

low bit: $\mathbf{s_i}$ (sum)

high bit: $\mathbf{c_{i+1}}$ (carry-out)

Full Adder

▶ Truth Table

x_i	y_i	c_i	s_i	c_{i+1}
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

- ▶ $s_i = 1$ iff 1 or 3 inputs = 1
- ▶ $c_{i+1} = 1$ iff 2 or 3 inputs = 1

Full Adder

► SOP Realization

S_i

$x_i y_i$

c_i

	00	01	11	10
0				
1				

C_{i+1}

$x_i y_i$

c_i

	00	01	11	10
0				
1				

Full Adder

► SOP Realization

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

$$s_i = \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + \bar{x}_i \bar{y}_i c_i + x_i y_i c_i$$

$$\text{Cost} = 5 + 16 = 21$$

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

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

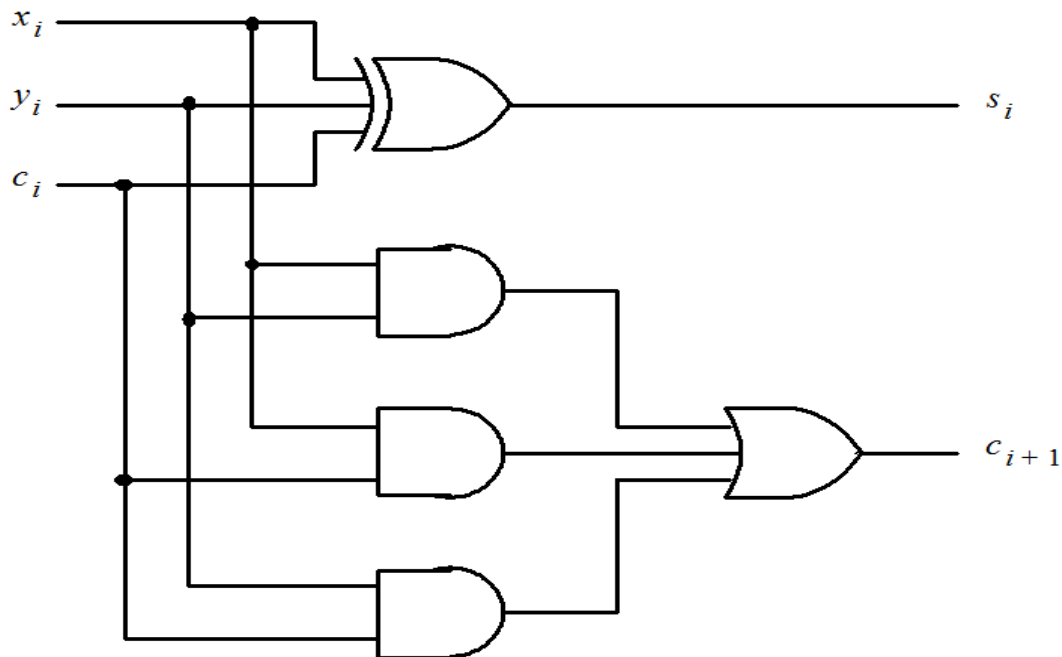
$$\text{Cost} = 4 + 9 = 13$$

$$\text{Total cost} = 34$$

Full Adder

► Use of XOR

$$s_i = x_i \oplus y_i \oplus c_i \quad c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$



$$\text{Total cost} = 5 + 12 = 17$$

XOR Gates

- ▶ XOR operations are sometimes very useful in optimizing logic circuits because **they cover non-adjacent squares** in a K-map.
- ▶ 2-input XOR

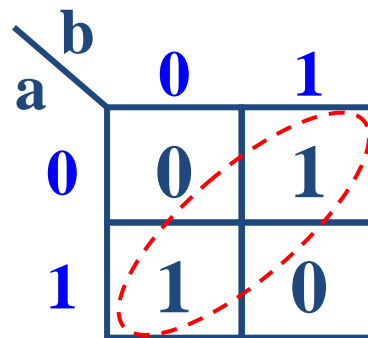
$$\begin{aligned} f &= a \oplus b \\ &= \bar{a} \cdot b + a \cdot \bar{b} \\ &= b \oplus a \end{aligned}$$

		b	
		0	1
a	0	0	1
	1	1	0

XOR Gates

- ▶ XOR operations are sometimes very useful in optimizing logic circuits because **they cover non-adjacent squares** in a K-map.
- ▶ 2-input XOR

$$\begin{aligned}f &= a \oplus b \\&= \bar{a} \cdot b + a \cdot \bar{b} \\&= b \oplus a\end{aligned}$$



A Karnaugh map for the 2-input XOR function. The map is a 2x2 grid. The horizontal axis is labeled 'b' with values 0 and 1. The vertical axis is labeled 'a' with values 0 and 1. The cells contain the following values: (a=0, b=0) is 0; (a=0, b=1) is 1; (a=1, b=0) is 1; (a=1, b=1) is 0. A red dashed line connects the two cells where the output is 1, illustrating that the function covers non-adjacent squares.

	b	0	1
a	0	0	1
1	1	1	0

XOR Gates – 3 input XOR

$$f = a \oplus b \oplus c = (a \oplus b) \oplus c = a \oplus (b \oplus c)$$

- ▶ $f = 1$ if an odd number of inputs = 1
- ▶ $f = 0$ if an even number of inputs = 1

		$a \oplus b \oplus c$			
f	bc	00	01	11	10
	a				
0					
1					

		$b \oplus c$			
a	bc	00	01	11	10
0					
1					

		$a \oplus b$			
a	bc	00	01	11	10
0					
1					

		$a \oplus c$			
a	bc	00	01	11	10
0					
1					

XOR Gates – 3 input XOR

$$f = a \oplus b \oplus c = (a \oplus b) \oplus c = a \oplus (b \oplus c)$$

- ▶ $f = 1$ if an odd number of inputs = 1
- ▶ $f = 0$ if an even number of inputs = 1

		$a \oplus b \oplus c$			
f	bc	00	01	11	10
a	0	0	1	0	1
	1	1	0	1	0

SOP: 5 gates
XOR: 1 gate

		$b \oplus c$			
a	bc	00	01	11	10
0	0	0	1	0	1
1	0	0	1	0	1

		$a \oplus b$			
a	bc	00	01	11	10
0	0	0	0	1	1
1	0	1	1	0	0

		$a \oplus c$			
a	bc	00	01	11	10
0	0	0	1	1	0
1	0	1	0	0	1

SOP: 3 gates; XOR: 1 gate

Features of XOR

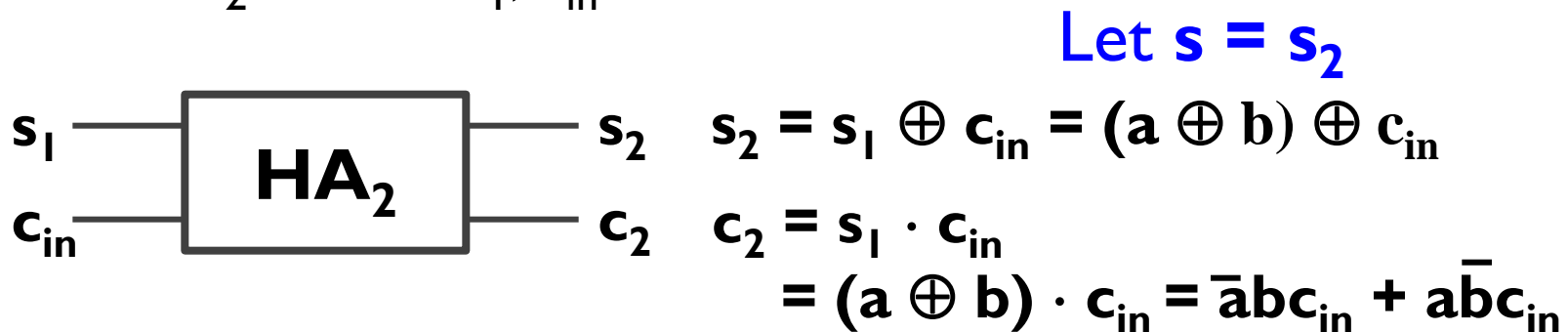
- ▶ XOR is also referred to as the odd function
- ▶ For the Two-input XOR , one input can be thought as the control signal to determine whether the true or complemented of the other input.
- ▶ XNOR, which is the complement of XOR, is also referred to as the coincidence operation.
- ▶ The symbol of the XNOR.

3.2.1 Decomposed full adder

- ▶ Can we make a full adder using two half adders?
- ▶ Use HA_1 to add a, b



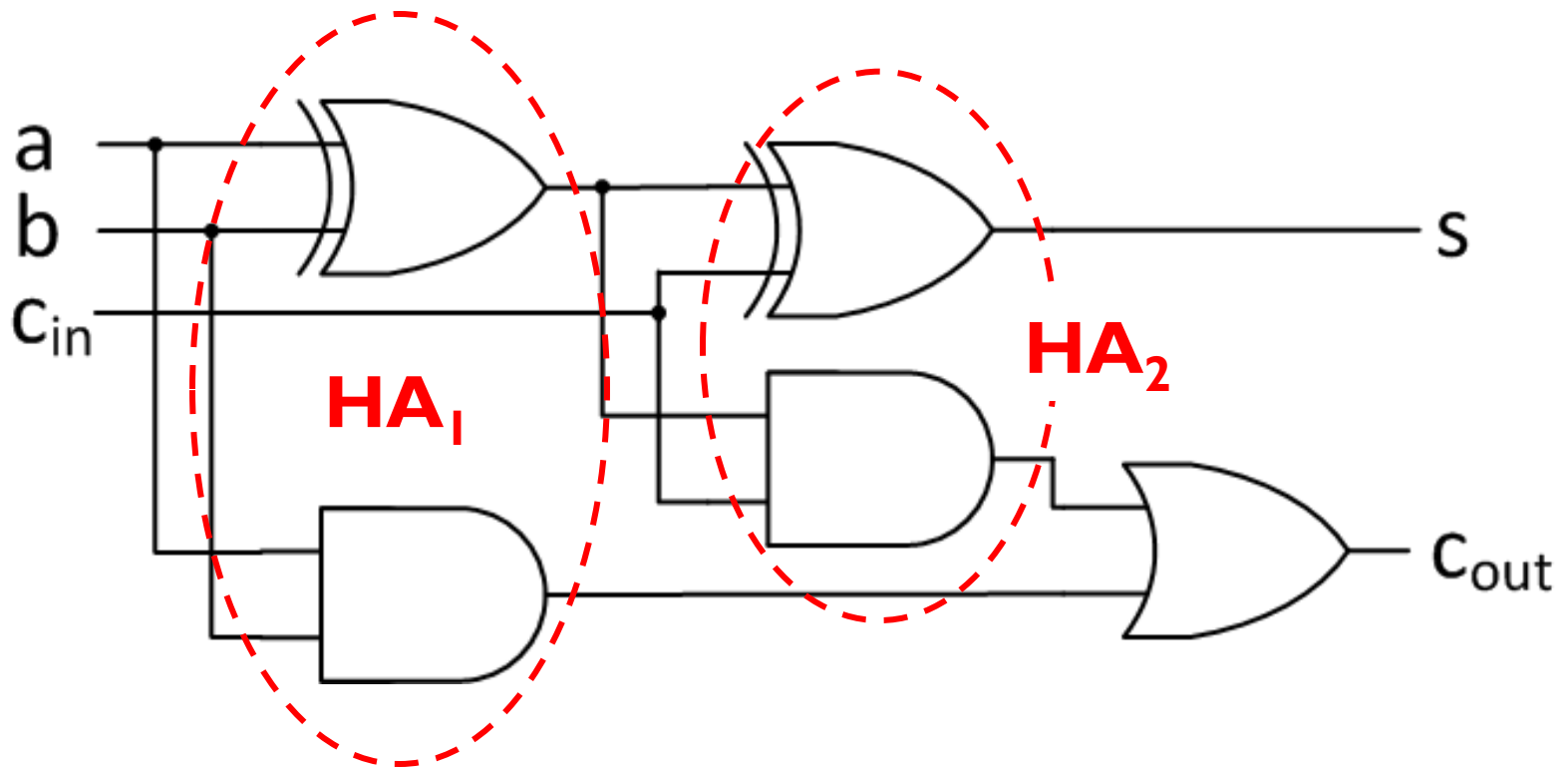
- ▶ Use HA_2 to add s_1, c_{in}



$$\text{Let } c_{out} = c_1 + c_2 = \bar{a}bc_{in} + a\bar{b}c_{in} + ab\bar{c}_{in} + abc_{in}$$

Decomposed full adder

► Circuit



$$\text{Total cost} = 5 + 10 = 15$$

3.3.2 Ripple-Carry Adder

- ▶ We can make an n -bit adder by using n full adders

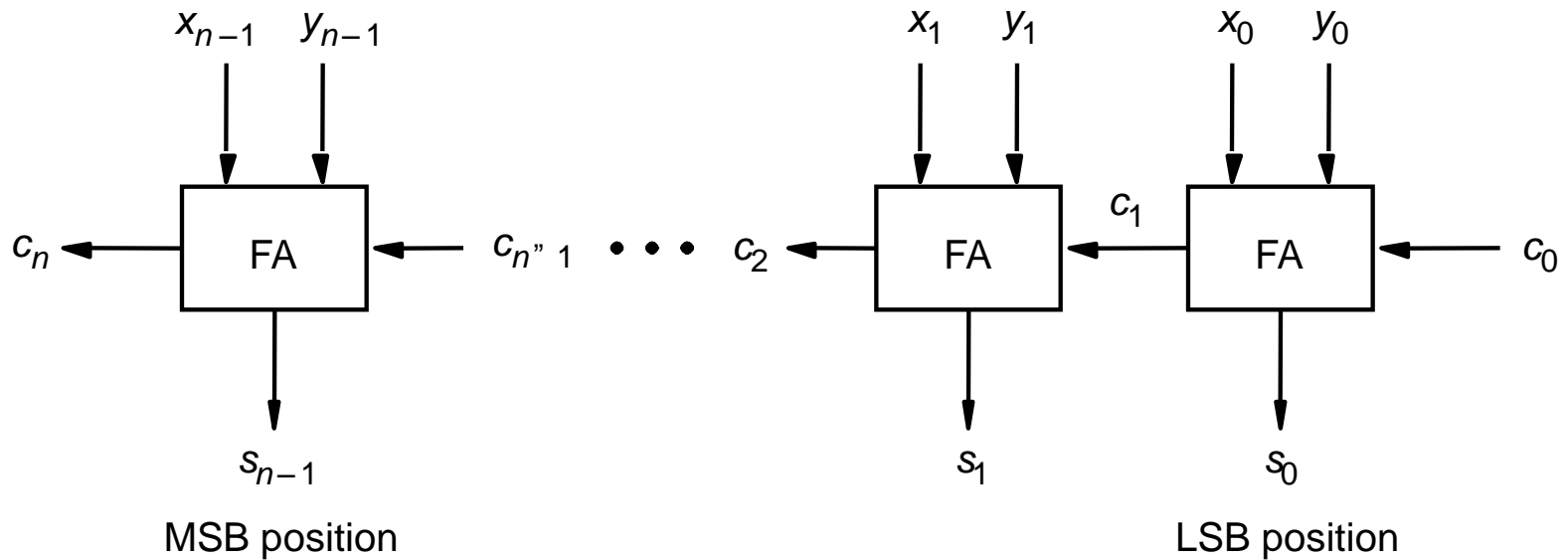
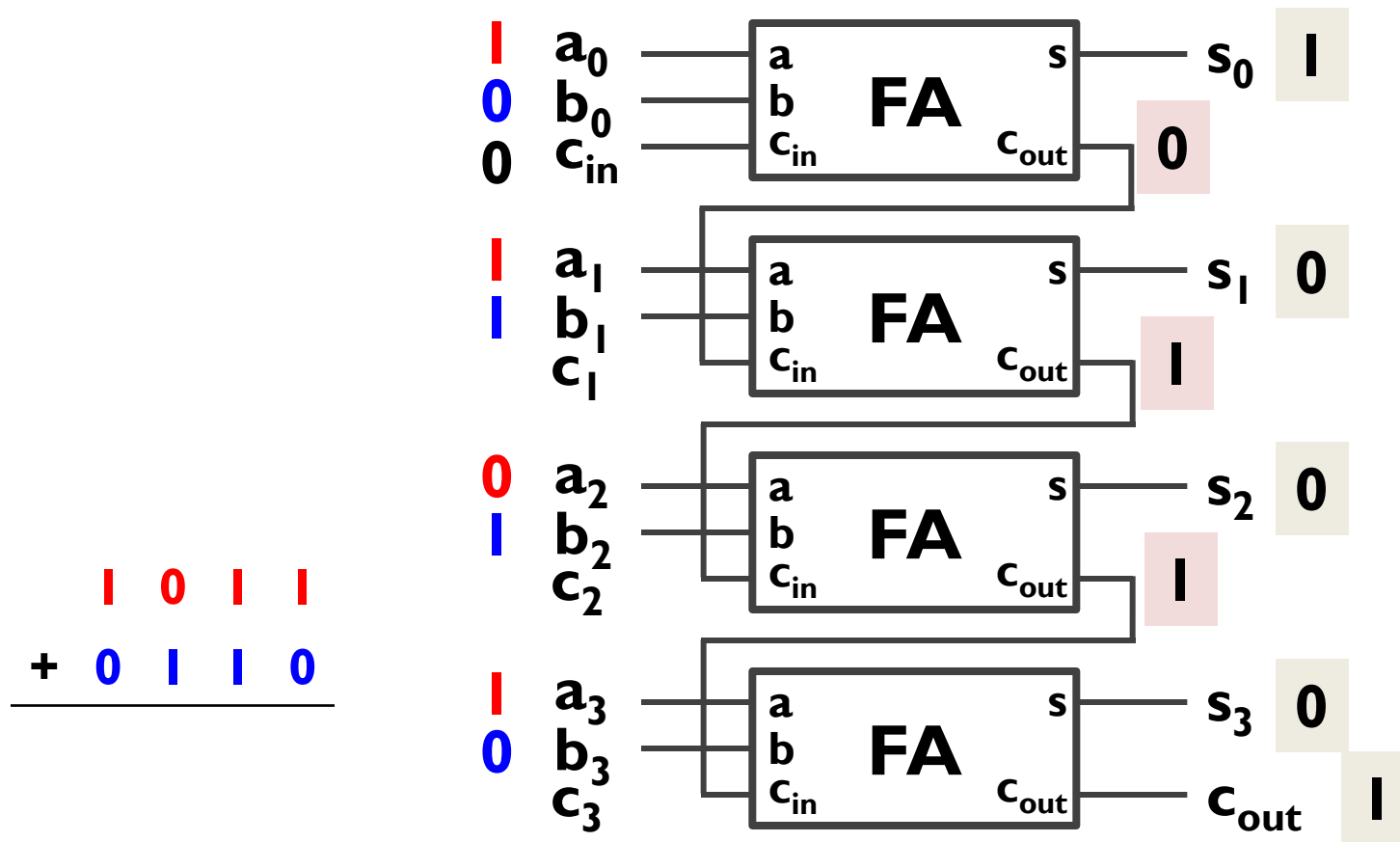
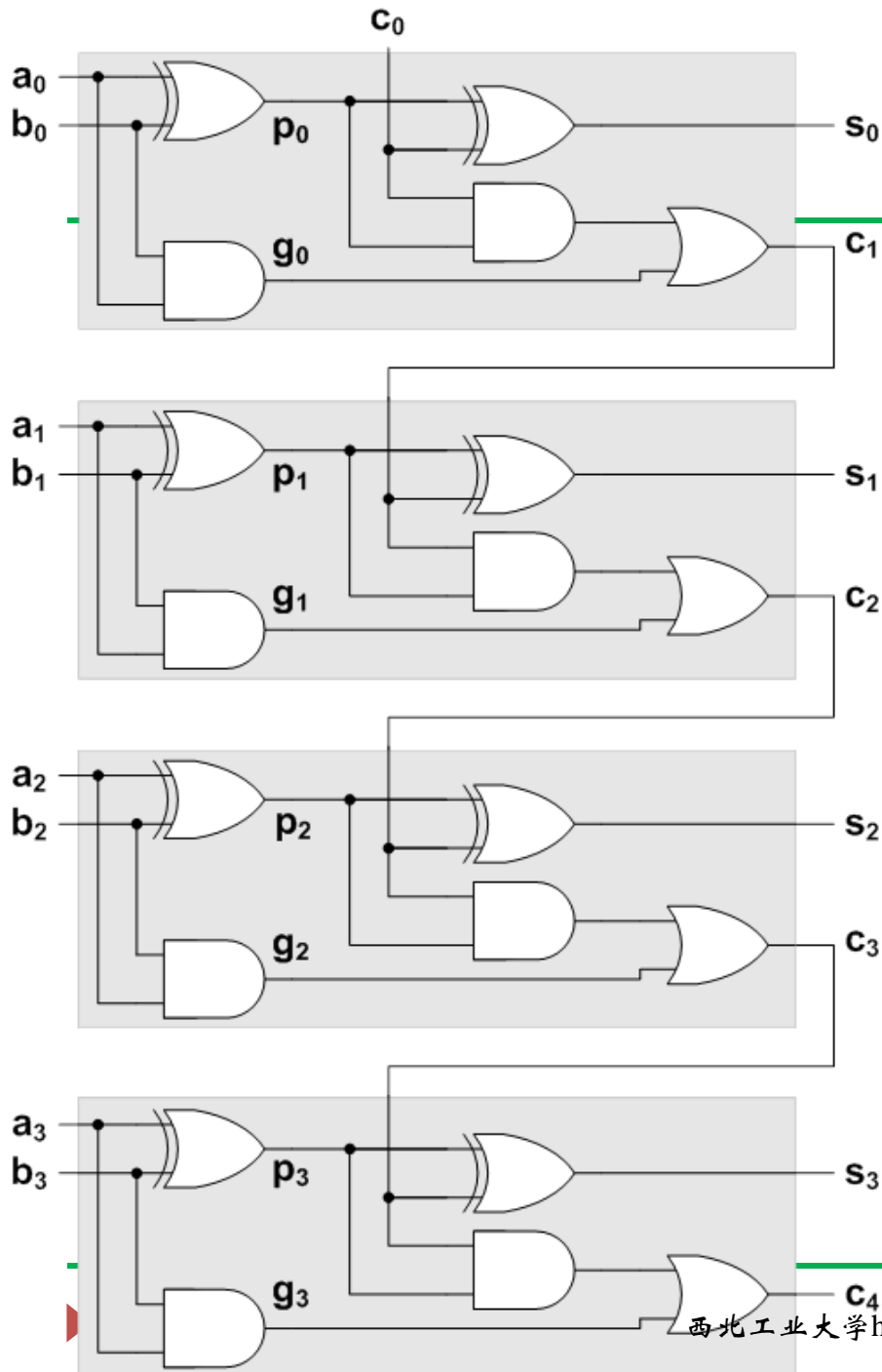


Figure 3.5. An n -bit ripple-carry adder.

3.3.2 Ripple-Carry Adder

- ▶ We can make an n -bit adder by using n full adders
- ▶ Ex: $n = 4$





4-Bit Ripple-Carry Adder

- ▶ The delay increases as more bits are added.

- ▶ Ex: 4-bit ripple-carry adder:

Max delay from inputs to $c_1 = 3$ gates

Max delay from inputs to $c_2 =$

Max delay from inputs to $c_3 =$

Max delay from inputs to c_{out}

?

Critical path delay

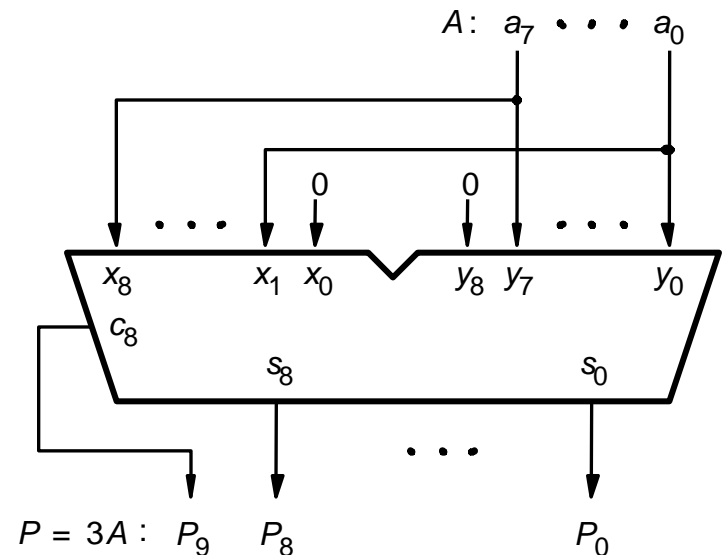
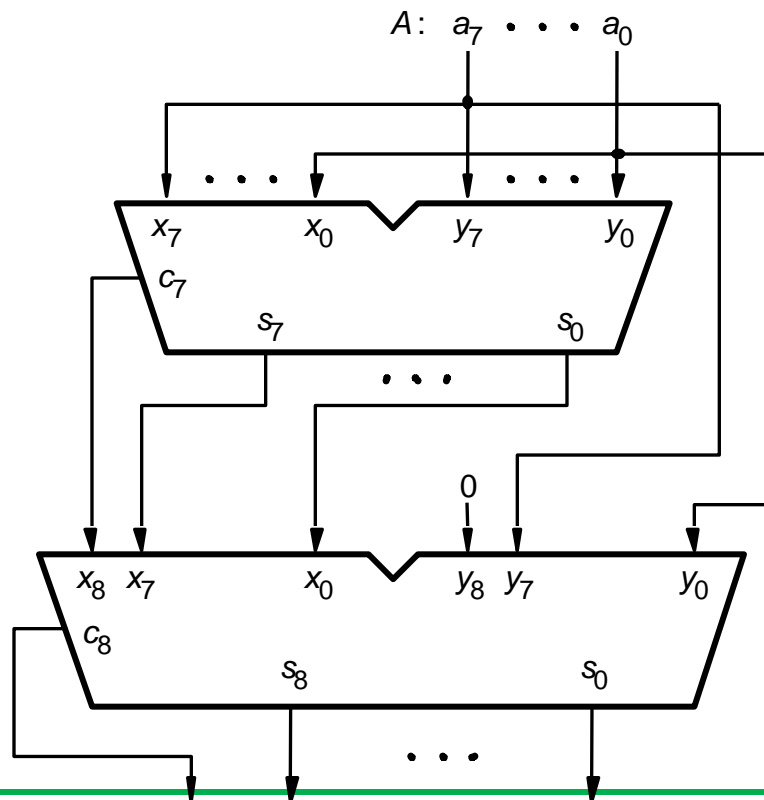
- ▶ Note that delays to s_1, s_2, s_3 are each one gate more than delays to c_1, c_2, c_3 .
 \Rightarrow If we can reduce delays to carry bits, delay to sum bits will also be reduced.

Analysis of the ripple-carry adder

- ▶ For ripple-carry adder, the total delay depends on the size of the number.
- ▶ When 32 or 64 bit number, the delay is unacceptable
- ▶ As a result, we need to design a new architecture in the following chapter
- ▶ Until now, we talk about the unsigned adder.

3.2.3 design Example

- Suppose that we need a circuit that multiplies an eight-bit unsigned number by 3. Let $A = a_7a_6 \cdots a_1a_0$ denote the number and $P = p_9p_8 \cdots p_1p_0$ denote the product $P = 3A$



More efficient!!

3.3.2 Addition and Subtraction

► 1's Complement Addition

$\begin{array}{r} (+5) \\ + (+2) \\ \hline (+7) \end{array}$	$\begin{array}{r} 0101 \\ + 0010 \\ \hline 0111 \end{array}$	$\begin{array}{r} (-5) \\ + (+2) \\ \hline (-3) \end{array}$	$\begin{array}{r} 1010 \\ + 0010 \\ \hline 1100 \end{array}$
$\begin{array}{r} (+5) \\ + (-2) \\ \hline (+3) \end{array}$	$\begin{array}{r} 0101 \\ + 1101 \\ \hline 10010 \\ \text{C} \rightarrow 1 \\ \hline 0011 \end{array}$	$\begin{array}{r} (-5) \\ + (-2) \\ \hline (-7) \end{array}$	$\begin{array}{r} 1010 \\ + 1101 \\ \hline 10111 \\ \text{C} \rightarrow 1 \\ \hline 1000 \end{array}$

Figure 3.8. Examples of 1's complement addition.

So: Correction may be needed, which means extra addition will be involved, the circuit is not simple as we expected.

Addition and Subtraction

► 2's Complement Addition

$$\begin{array}{r} (+5) \\ + (+2) \\ \hline (+7) \end{array} \quad \begin{array}{r} 0101 \\ + 0010 \\ \hline 0111 \end{array}$$

$$\begin{array}{r} (-5) \\ + (+2) \\ \hline (-3) \end{array} \quad \begin{array}{r} 1011 \\ + 0010 \\ \hline 1101 \end{array}$$

$$\begin{array}{r} (+5) \\ + (-2) \\ \hline (+3) \end{array} \quad \begin{array}{r} 0101 \\ + 1110 \\ \hline 10011 \end{array}$$

$$\begin{array}{r} (-5) \\ + (-2) \\ \hline (-7) \end{array} \quad \begin{array}{r} 1011 \\ + 1110 \\ \hline 11001 \end{array}$$

↑
ignore

↑
ignore

Figure 3.9. Examples of 2's complement addition.

So: The result is always correct, and the circuit is also simple, just perform one time addition.

Addition and Subtraction

► 2's Complement Subtraction

The easiest way of performing subtraction is to negate the subtrahend and add it to the minuend.

$$\begin{array}{r}
 (+5) \quad 0101 \\
 - (+2) \quad 0010 \\
 \hline
 (+3)
 \end{array}
 \Rightarrow
 \begin{array}{r}
 0101 \\
 + 1110 \\
 \hline
 10011
 \end{array}$$

$$\begin{array}{r}
 1011 \\
 + 1110 \\
 \hline
 11001
 \end{array}$$

ignore

$$\begin{array}{r}
 (-5) \quad 1011 \\
 - (+2) \quad 0010 \\
 \hline
 (-7)
 \end{array}
 \Rightarrow
 \begin{array}{r}
 1011 \\
 + 1110 \\
 \hline
 11001
 \end{array}$$

ignore

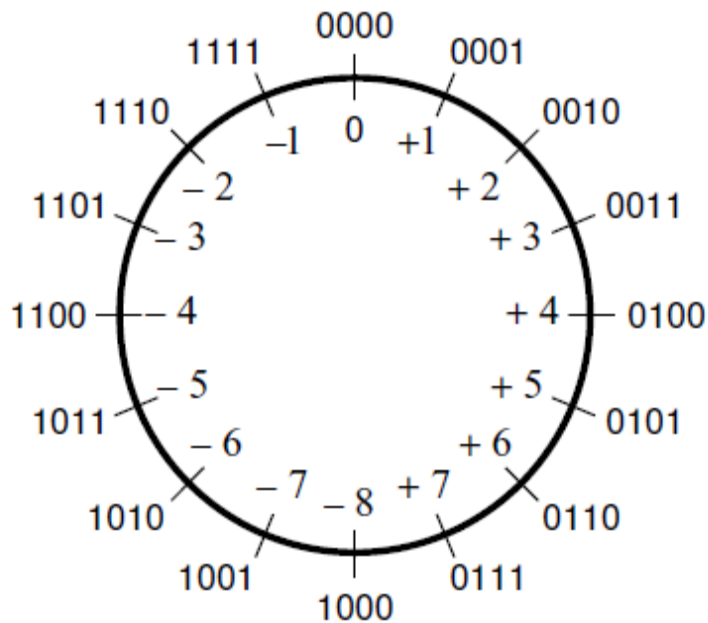
$$\begin{array}{r}
 (+5) \quad 0101 \\
 - (-2) \quad 1110 \\
 \hline
 (+7)
 \end{array}
 \Rightarrow
 \begin{array}{r}
 0101 \\
 + 0010 \\
 \hline
 0111
 \end{array}$$

$$\begin{array}{r}
 (-5) \quad 1011 \\
 - (-2) \quad 1110 \\
 \hline
 (-3)
 \end{array}
 \Rightarrow
 \begin{array}{r}
 1011 \\
 + 0010 \\
 \hline
 1101
 \end{array}$$

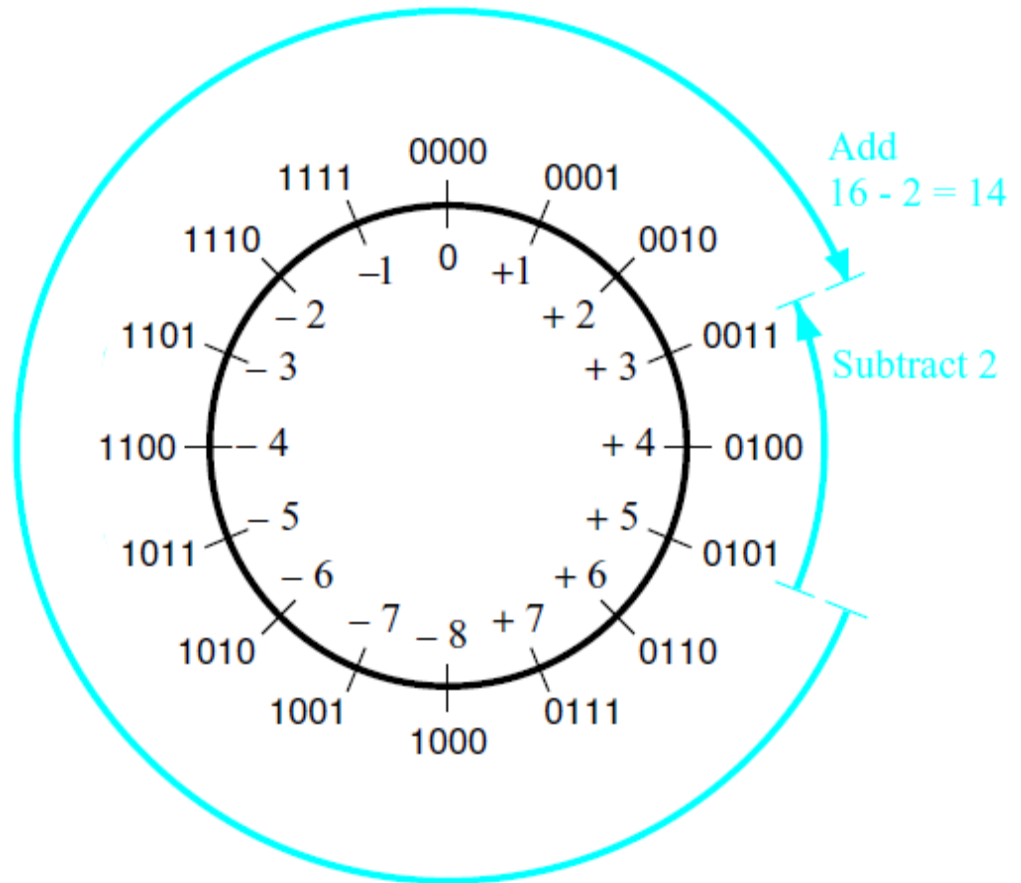
Figure 3.10. Examples of 2's complement subtraction.

Addition and Subtraction

► 2's Complement Subtraction



(a) The number circle



(b) Subtracting 2 by adding its 2's complement



Graphical interpretation of four-bit 2's complement numbers

3.3.3 Adder and Subtraction Unit

- ▶ For $X - Y = X + (-Y)$, we need the 2's complement of Y , which can be obtained by adding 1 to the 1's complement of Y . In other words, **Subtraction** can be completed by the addition
- ▶ So : We need to perform $X + Y$ and $X + \overline{Y} + 1$ In just one circuit.
- ▶ How can we build the unify circuit to perform addition and subtraction.
- ▶ Think about the feature of XOR.

Adder/Subtractor unit

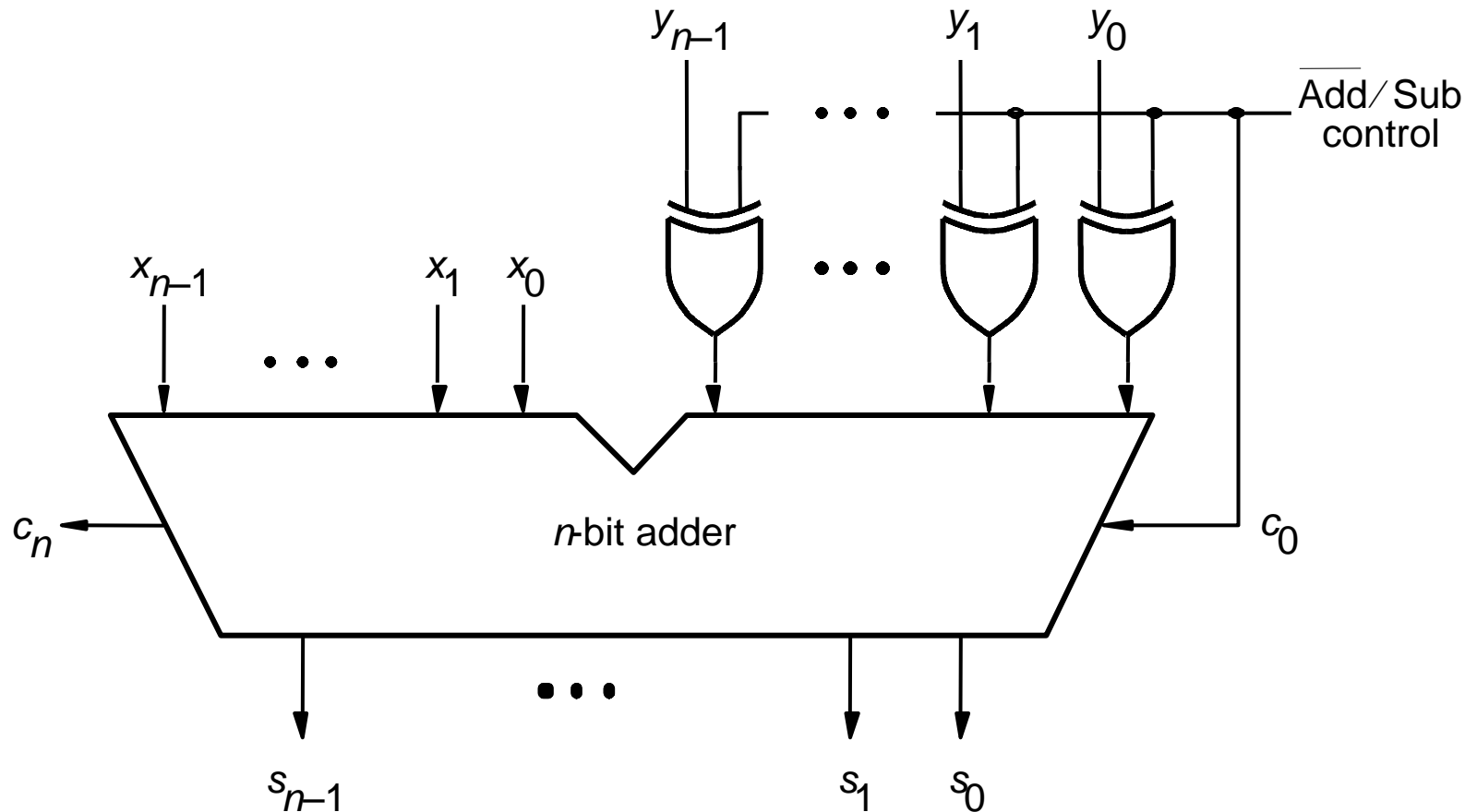


Figure 3.12. Adder/Subtractor unit.

Hints:

- ▶ When design the digital circuit:
 - ▶ As flexible as possible
 - ▶ As many tasks as possible
- ▶ To minimize the area and reduce the wiring complexity

3.3.5 Arithmetic Overflow

- ▶ What is overflow?
 - ▶ The result beyonds the range of the 2's complement of n-bit signed number.
- ▶ So:The occurrence of overflow should be detected and reported the system.
- ▶ Investigate some example in the textbook.

Analysis of the Overflow

- ▶ The four cases where 2's-complement numbers with magnitudes of 7 and 2 are added.

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

Figure 3.13. Examples of determination of overflow.



How to detect the overflow?

- ▶ Analysis Figure 3.9 and Figure 3.13
- ▶ We can draw the conclusion:
- ▶ for the n-bit numbers ,we have

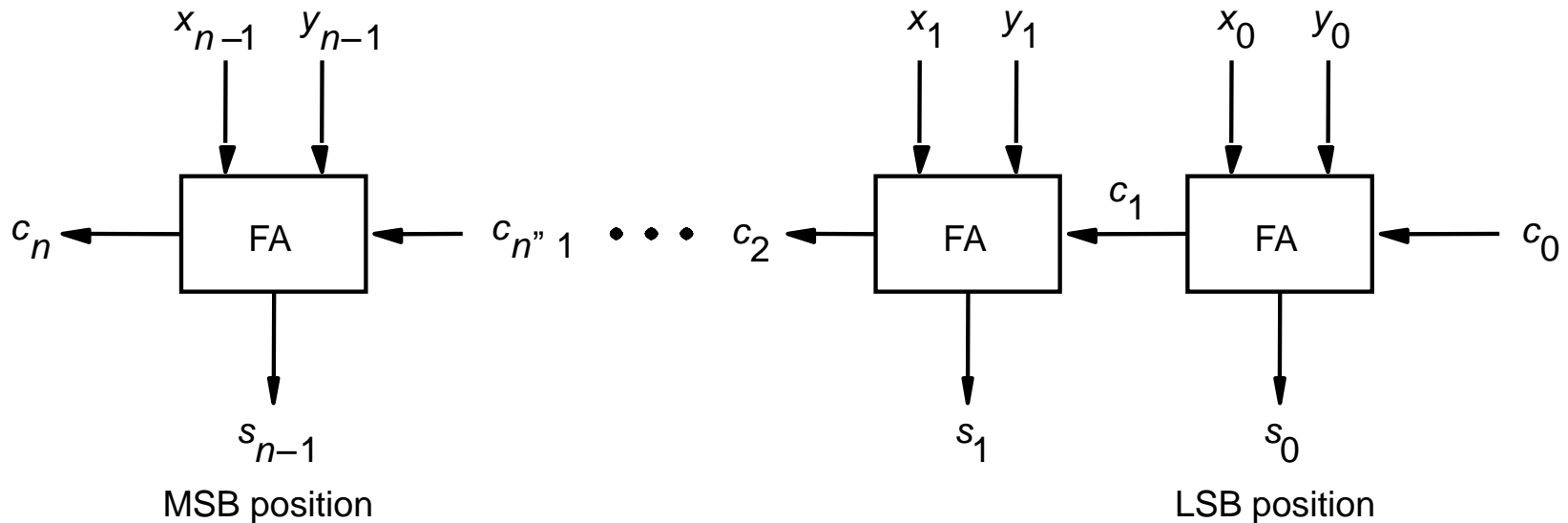
$$\text{Overflow} = c_{n-1} \oplus c_n$$

*Note: c_{n-1} is the carry out from the MSB position
 C_n is the carry out form the Sign position*

- ▶ Another way to detect the overflow is to compare the sign bit of sum with the sign of summands.
- ▶ So: ??
- ▶ Understand the difference of carry out and the overflow

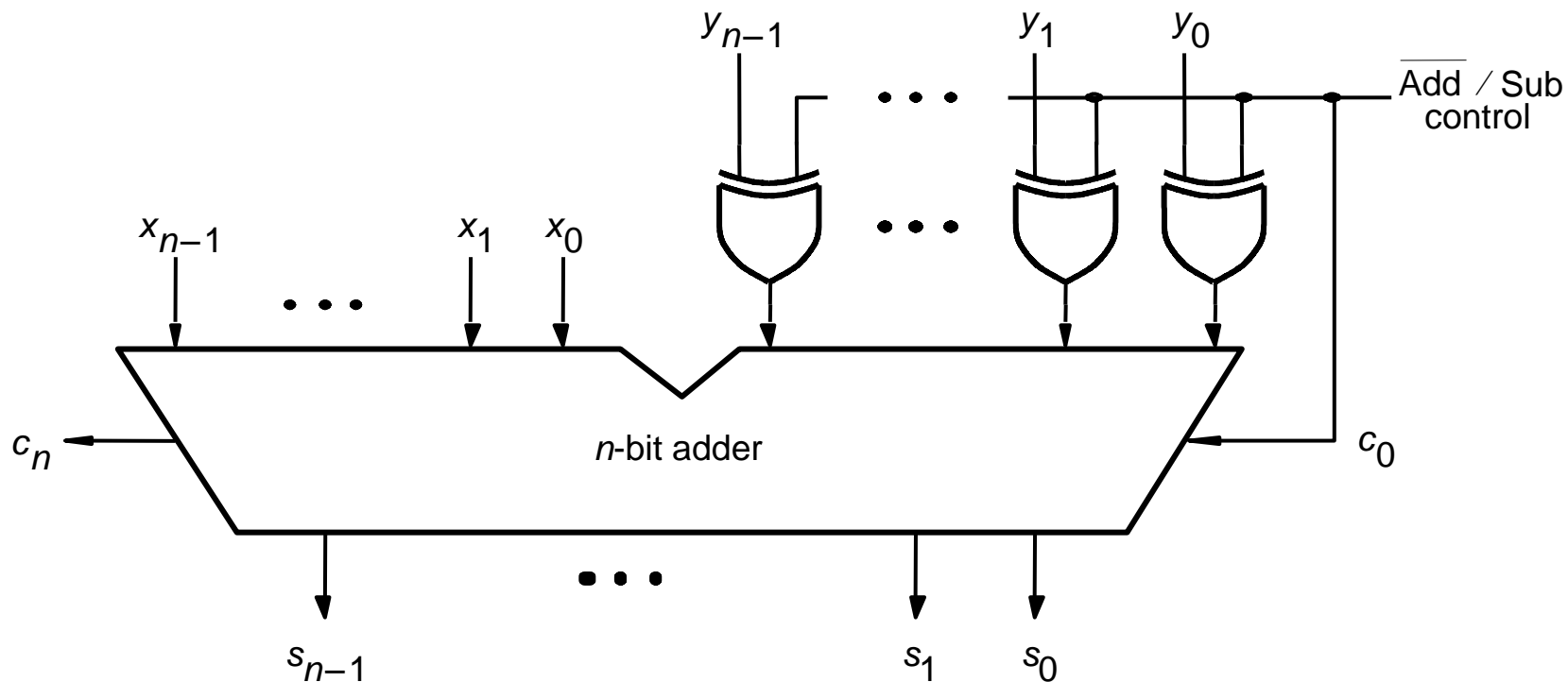
3.3.6 Performance Issues

- ▶ A commonly used indicator of the value of a system is its price/performance ratio.



○ Circuit delay: $(2n+1)\Delta t$

Performance Issues



- ▶ The longest delay is often referred to as the critical-path delay.
- ▶ The path that causes this delay is called the critical path.

Outline

Lecture 13 & 14 & 15:

Chapter: 3.2-3.5,3.8, Page 122-166 and 178-194

Addition of Unsigned Numbers

- ▶ 1-bit Addition (half adder and full adder)
- ▶ XOR gates
- ▶ Multi-bit Addition and subtraction
- ▶ Ripple-Carry Adder
- ▶ Fast Adders
- ▶ Multiplication of Unsigned Numbers(*)
- ▶ Verilog Basic 2 and Examples

3.4 Fast Adder

- ▶ We can reduce the delays to the carry bits by calculating them directly from **the inputs**, not from the outputs of the previous stage.
- ▶ The trade-off is an increase in complexity for a decrease in delay.
- ▶ There are two ways to produce a carry output from each stage.
 - ▶ Carry Generation
 - ▶ Carry Propagate

Fast Adder

- ▶ Carry Generation: in a given stage, a carry is generated if both a and b are 1.

$$\Rightarrow c_{n+1} = 1 \text{ if } a_n = b_n = 1$$

$$\begin{array}{r} \\ \\ + \\ \hline 1 \end{array}$$

↑ ↑
No carry at stage 0
Carry generated at stage 1

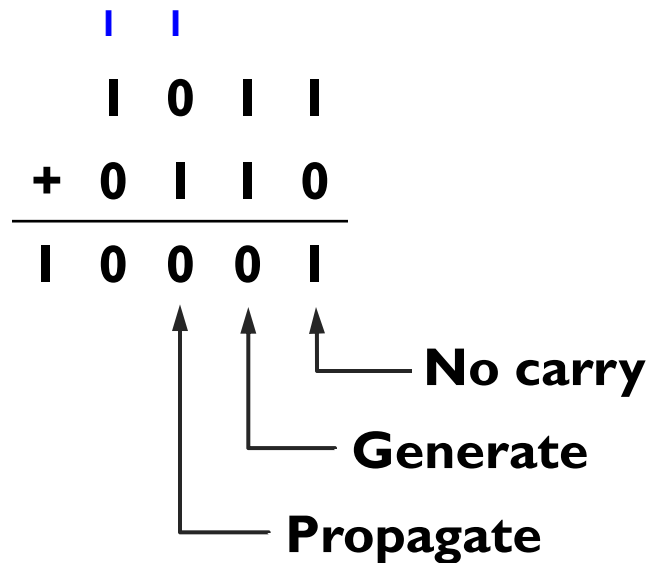
$$\Rightarrow g_n = a_n b_n$$



Fast Adder

- ▶ Carry Propagate: if a carry comes in to a given stage, we will propagate that carry to the next stage if either a or b is 1.

$$\Rightarrow c_{n+1} = 1 \text{ if } c_n = 1 \text{ and } (a_n = 1 \text{ or } b_n = 1)$$



Note: if $a_n = b_n = c_n = 1$, it is customary to count this as a generate, but not a propagate. (Book is different.)

$$\Rightarrow p_n = a_n \oplus b_n$$
$$\Rightarrow p_n = a_n + b_n$$

Fast Adder

- ▶ Note: g_n and p_n are already produced by the full adder circuit!!!

- ▶ Adder outputs

$$s_n = a_n \oplus b_n \oplus c_n = p_n \oplus c_n$$

$$c_{n+1} = a_n b_n + (a_n + b_n) c_n = g_n + p_n c_n$$

Fast Adder

- ▶ To reduce delays to carry outputs, rewrite c_{n+1} in terms of c_0 , g , and p signals

- ▶ Bit 0: $c_1 = g_0 + p_0 c_0$

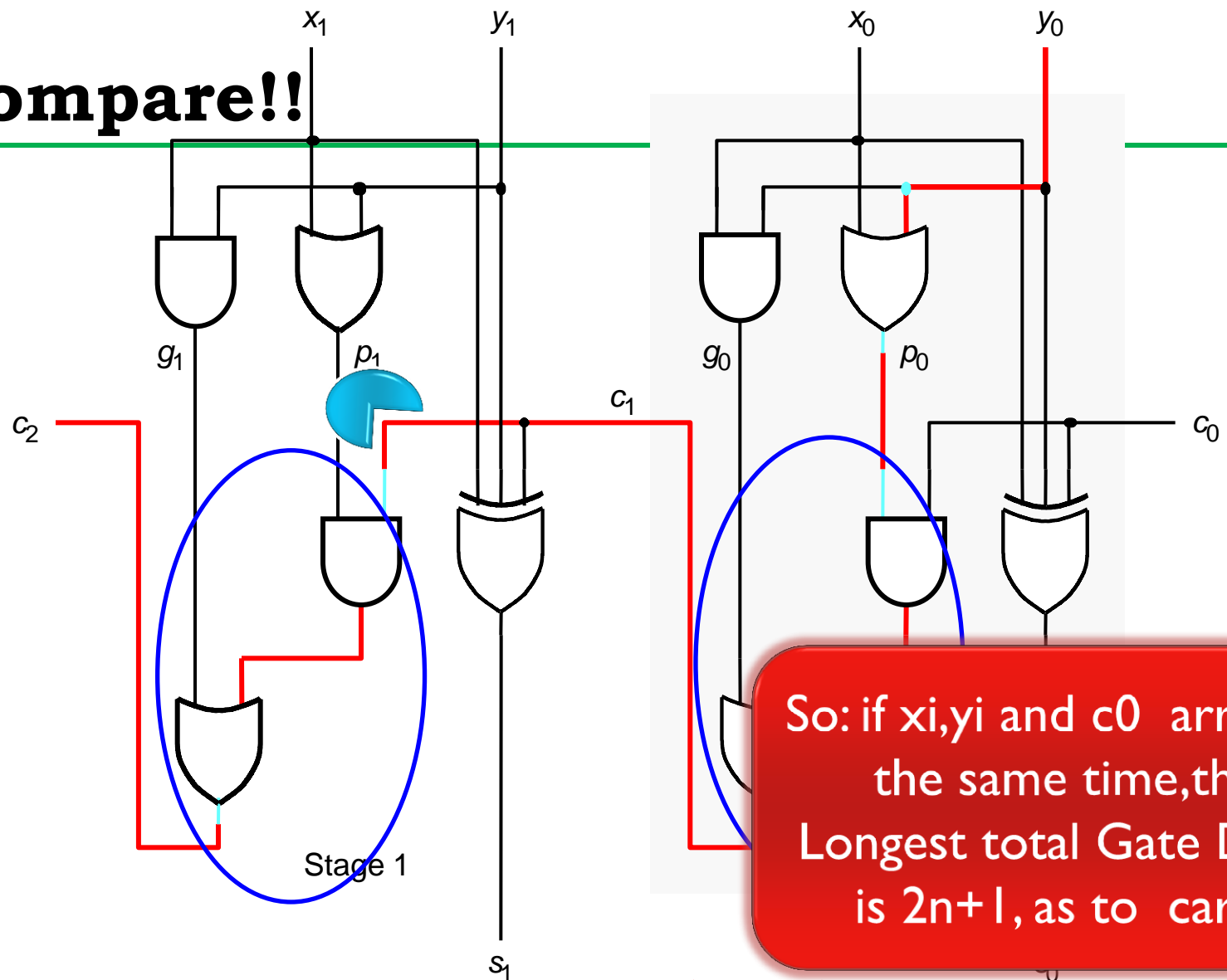
- ▶ Bit 1: $c_2 = g_1 + p_1 c_1$
 $= g_1 + p_1 g_0 + p_1 p_0 c_0$

- ▶ Bit 2: $c_3 = g_2 + p_2 c_2$
 $= g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$

- ▶ Bit 3: $c_4 = g_3 + p_3 c_3$
 $= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$

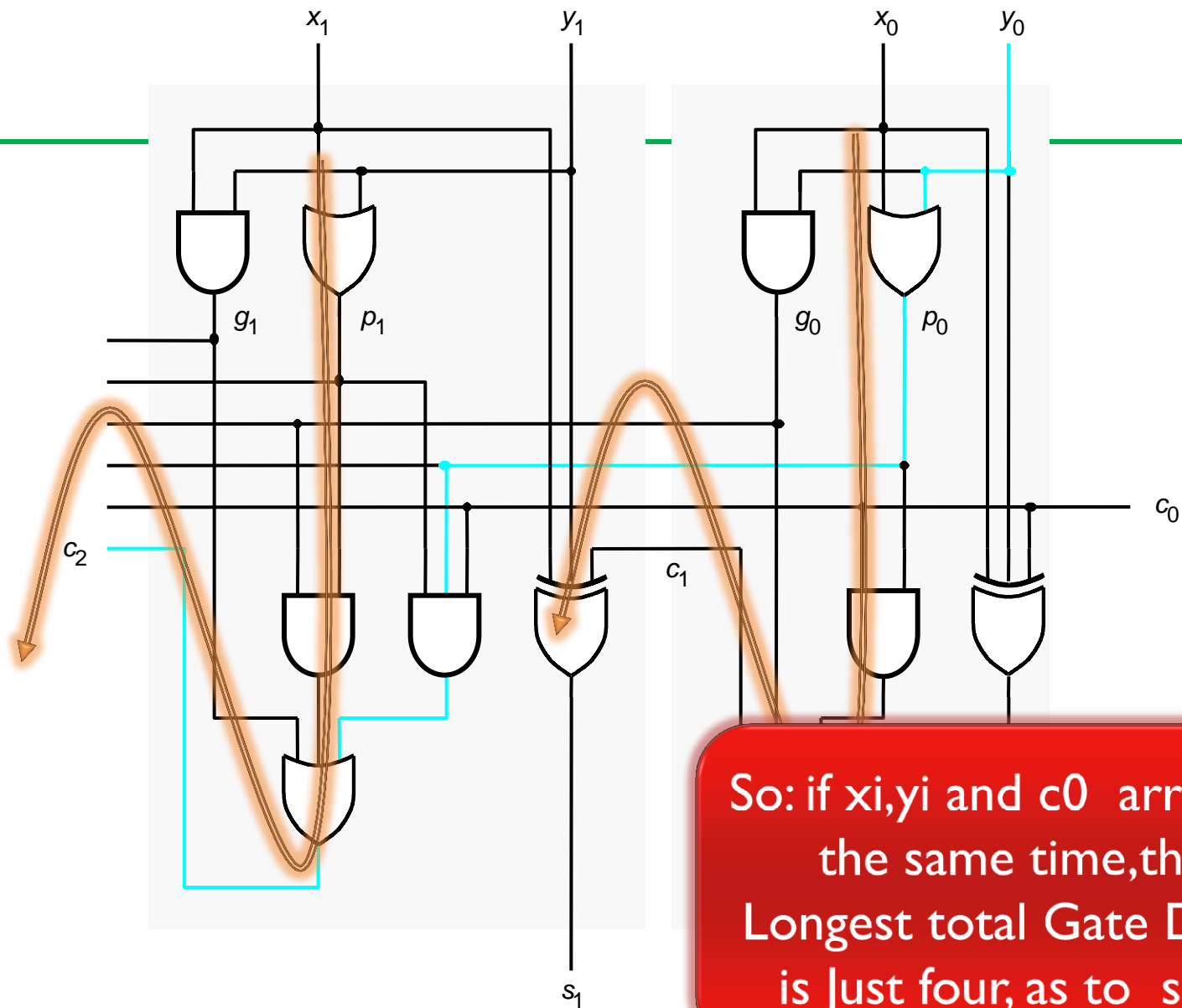
- ▶ Result: Carry-lookahead adder
- ▶ Two level AND-OR circuit in which c_i can be evaluated very quickly

Compare!!



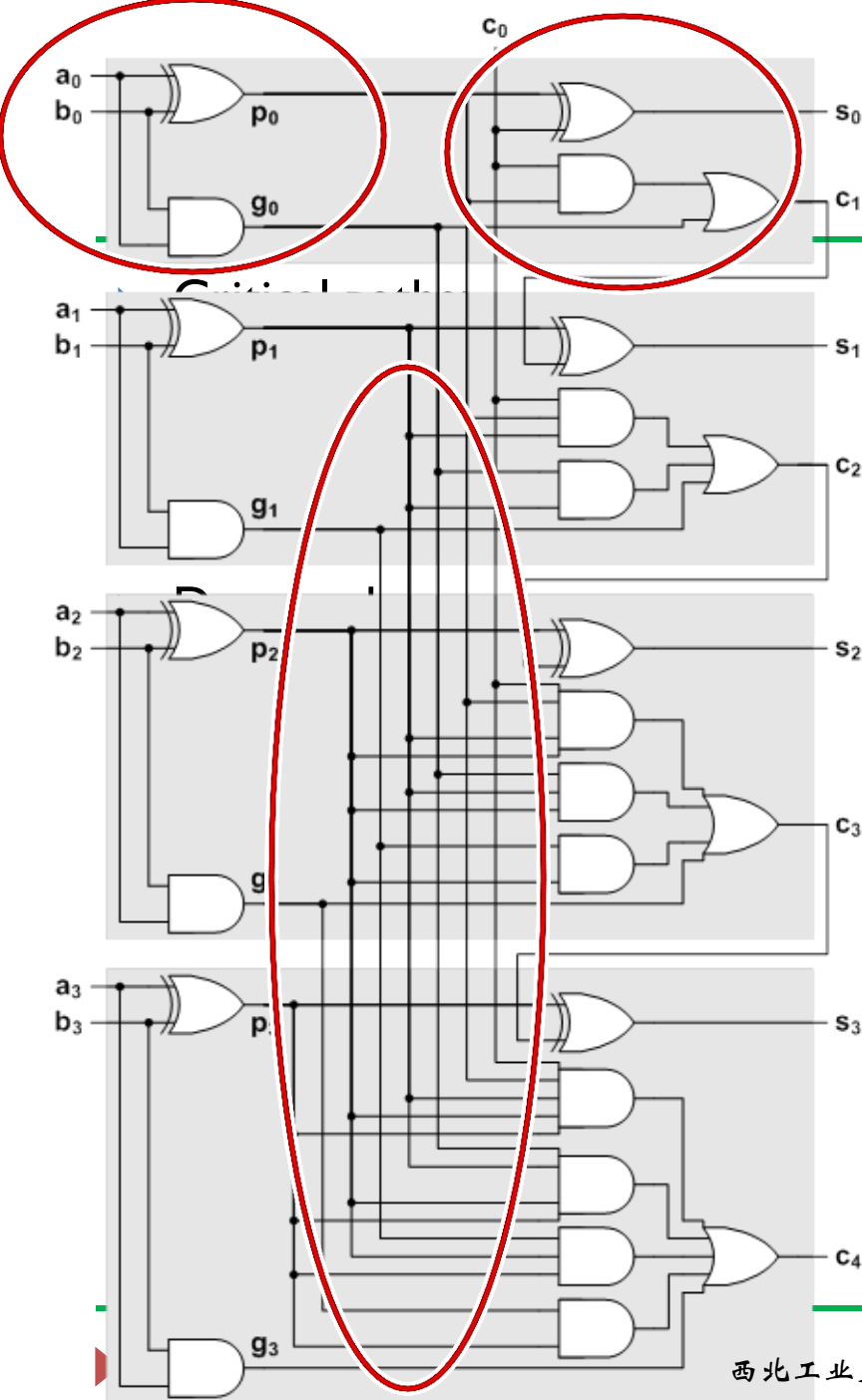
So: if x_i, y_i and c_0 arrive at the same time, the Longest total Gate Delay is $2n+1$, as to carry

Figure 3.14. A ripple-carry adder *based on Expression 3.3.*



So: if x_i, y_i and c_0 arrive at the same time, the Longest total Gate Delay is just four, as to sum

► Figure 3.15. The first two stages of a carry-lookahead adder. 46 高北工业大学 huangxp@nwnpu.edu.cn 2019/9/26



4-Bit Carry-Look-Ahead Adder **based** **On the Figure 3.4**

- ▶ Critical paths:
 - inputs to s_1, s_2, s_3
- ▶ Critical path delay = 4 gates
- ▶ Drawbacks:
 - ▶ Complexity, fan-ins
 - ▶ Remember: the number of fan-in is
A part of Cost

So: Fan-in is limited

- ▶ Comprise:
 - You should recognize the
Critical path and calculate
the critical path delay

How to Build larger Adder:

Parallelization Inside Block

Serialization Between Blocks

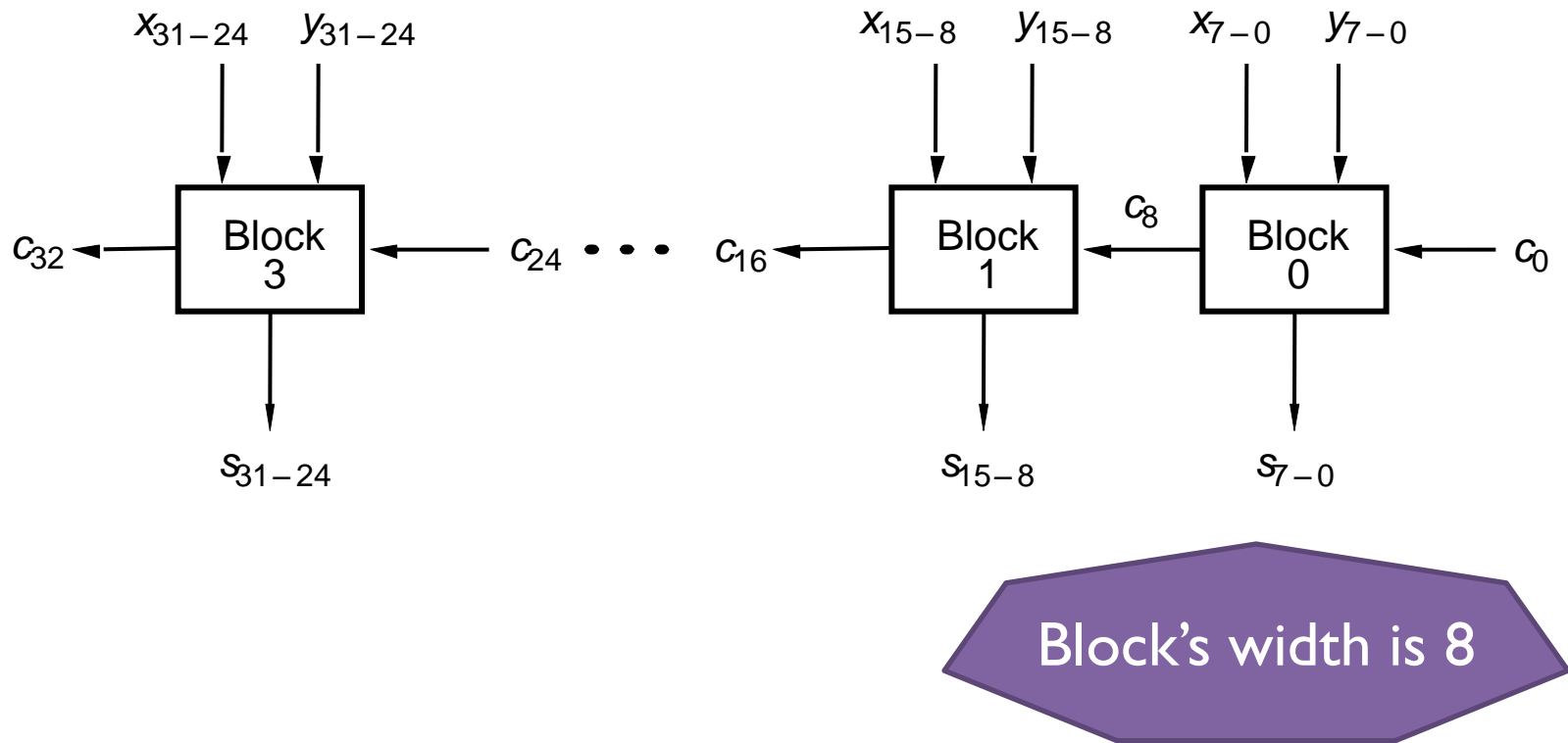


Figure 3.16. A hierarchical carry-lookahead adder with ripple-carry between blocks.

Parallelization Inside Block and Between Blocks

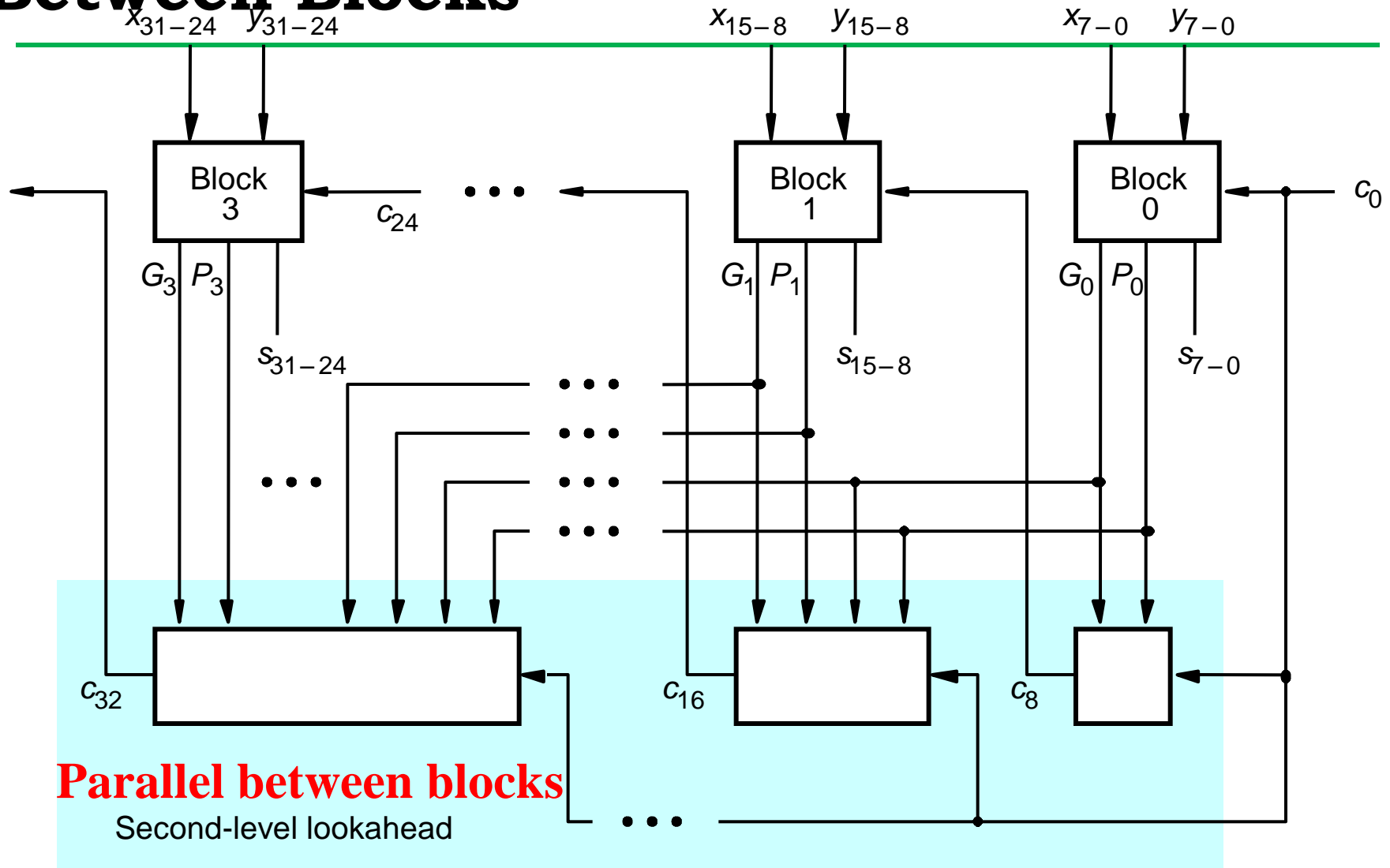
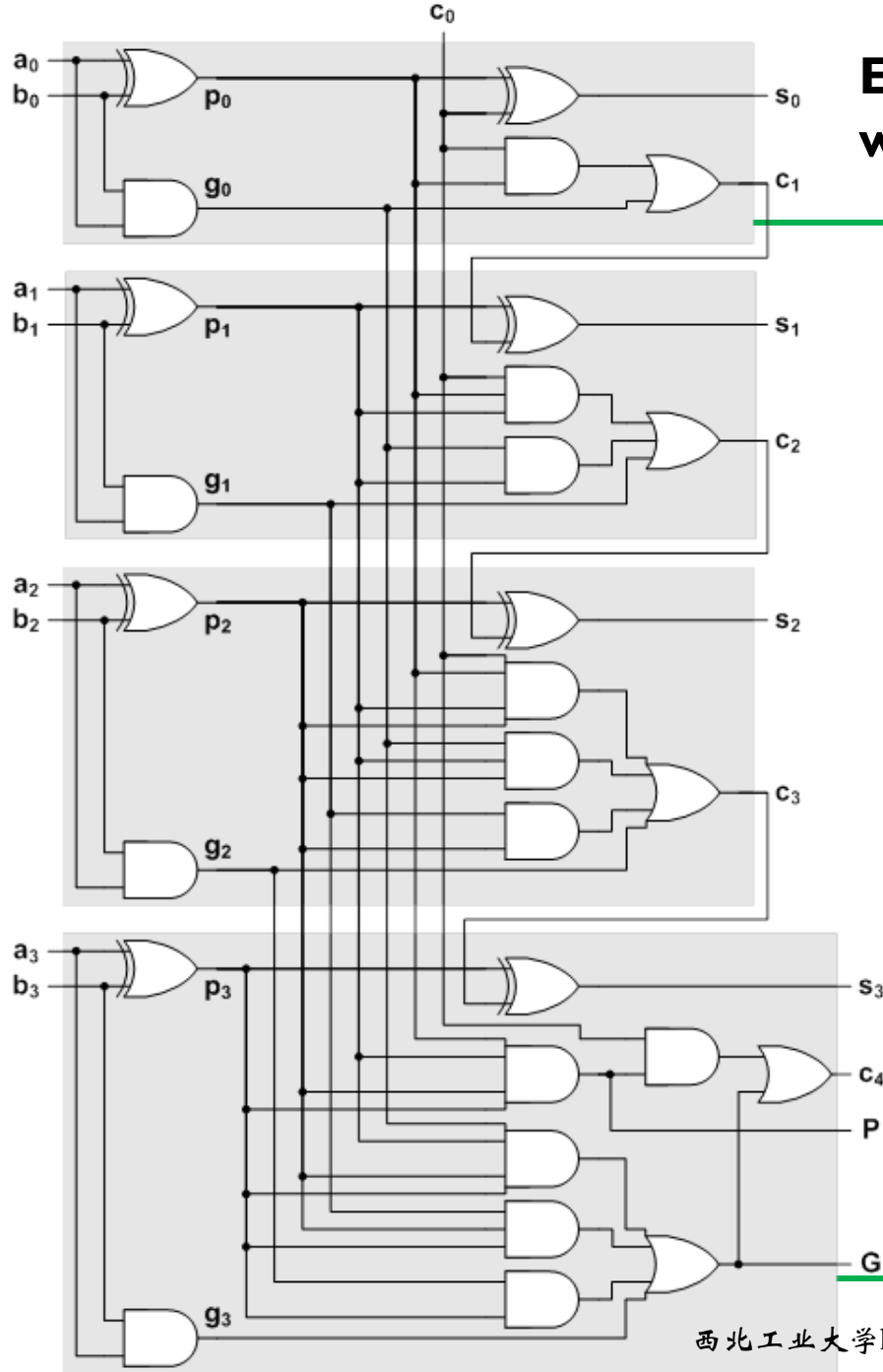


Figure 3.17. A hierarchical carry-lookahead adder. 2019/9/26

Example: 4-Bit Carry-Look-Ahead Adder w/ Block Generate and Propagate



- When will a 4-bit adder propagate a carry from c_0 to c_4 ?

A: When it is propagated at every stage

$$P = p_3 p_2 p_1 p_0$$

- When will a 4-bit adder generate a carry output c_4 ?

A: (1) if a carry is generated at the last stage

– **OR** –

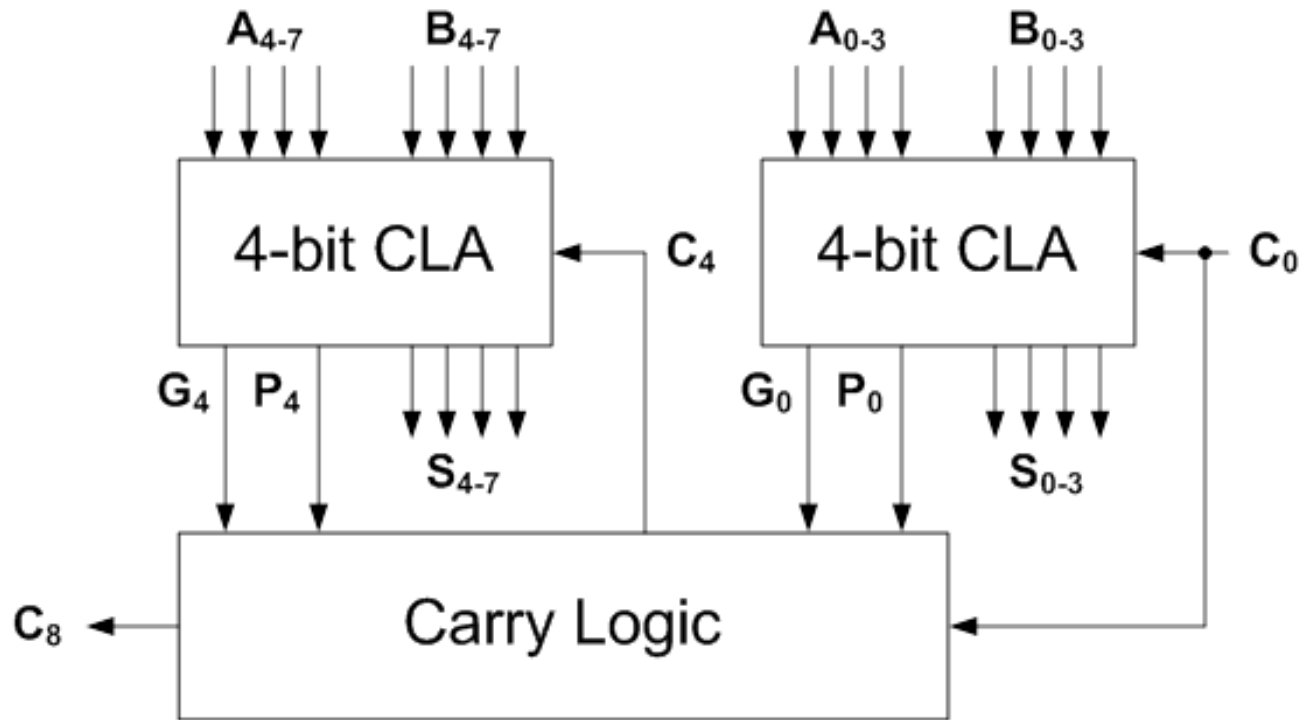
(2) If it is generated at any stage and propagated through all following stages

$$G = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$$

$$c_4 = G + P c_0$$

This is how the carry out is computed from the block generate and propagate

Example



Creating Larger Adders Using
4-bit Carry-Look-Ahead Blocks

3.7.3

Binary-Coded-Decimal Representation

- If $X + Y \leq 9$, then the addition is the same as the addition of 2 four-bit unsigned binary numbers.
- if $X + Y > 9$, then the result requires two BCD digits.
- Two cases where some correction has to be made:

$$Z = X + Y$$

If $Z \leq 9$, then $S = Z$ and

carry-out = 0

If $Z > 9$, then $S = Z + 6$ and

carry-out = 1

X	0 1 1 1	7
+ Y	+ 0 1 0 1	+ 5
Z	1 1 0 0	12
	+ 0 1 1 0	
carry →	1 0 0 1 0	
	<u> </u>	
	S = 2	

X	1 0 0 0	8
+ Y	+ 1 0 0 1	+ 9
Z	1 0 0 0 1	17
	+ 0 1 1 0	
carry →	1 0 1 1 1	
	<u> </u>	
	S = 7	

Figure 3.38. Addition of BCD digits.

Binary-Coded-Decimal Representation

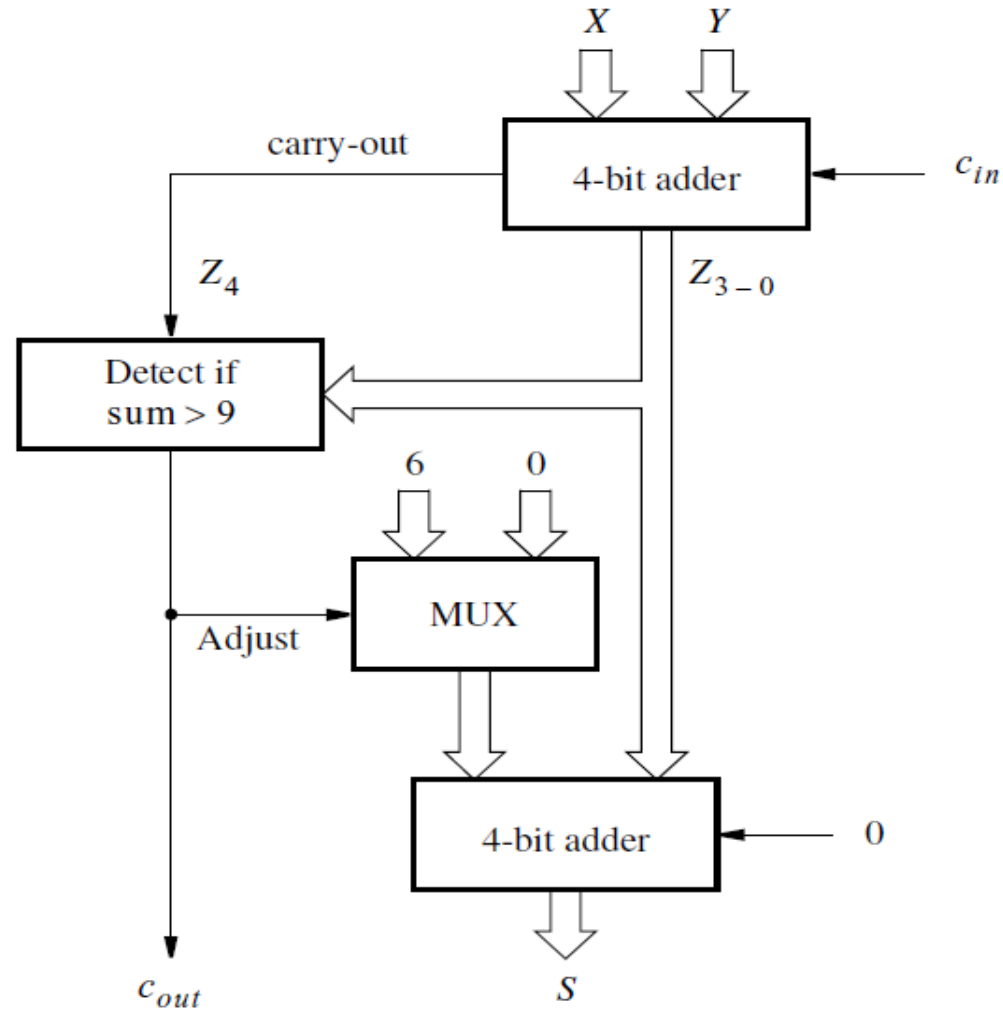


Figure 3.39. Block diagram for a one-digit BCD adder.

Binary-Coded-Decimal Representation

```
module bcdadd(Cin, X, Y, S, Cout);  
    input Cin;  
    input [3:0] X, Y;  
    output reg [3:0] S;  
    output reg Cout;  
    reg [4:0] Z;  
  
    always @ (X, Y, Cin)  
    begin  
        Z = X + Y + Cin;  
        if (Z < 10)  
            {Cout, S} = Z;  
        else  
            {Cout, S} = Z + 6;  
    end  
  
endmodule
```

Figure 3.40. Verilog code for a one-digit BCD adder.

Binary-Coded-Decimal Representation

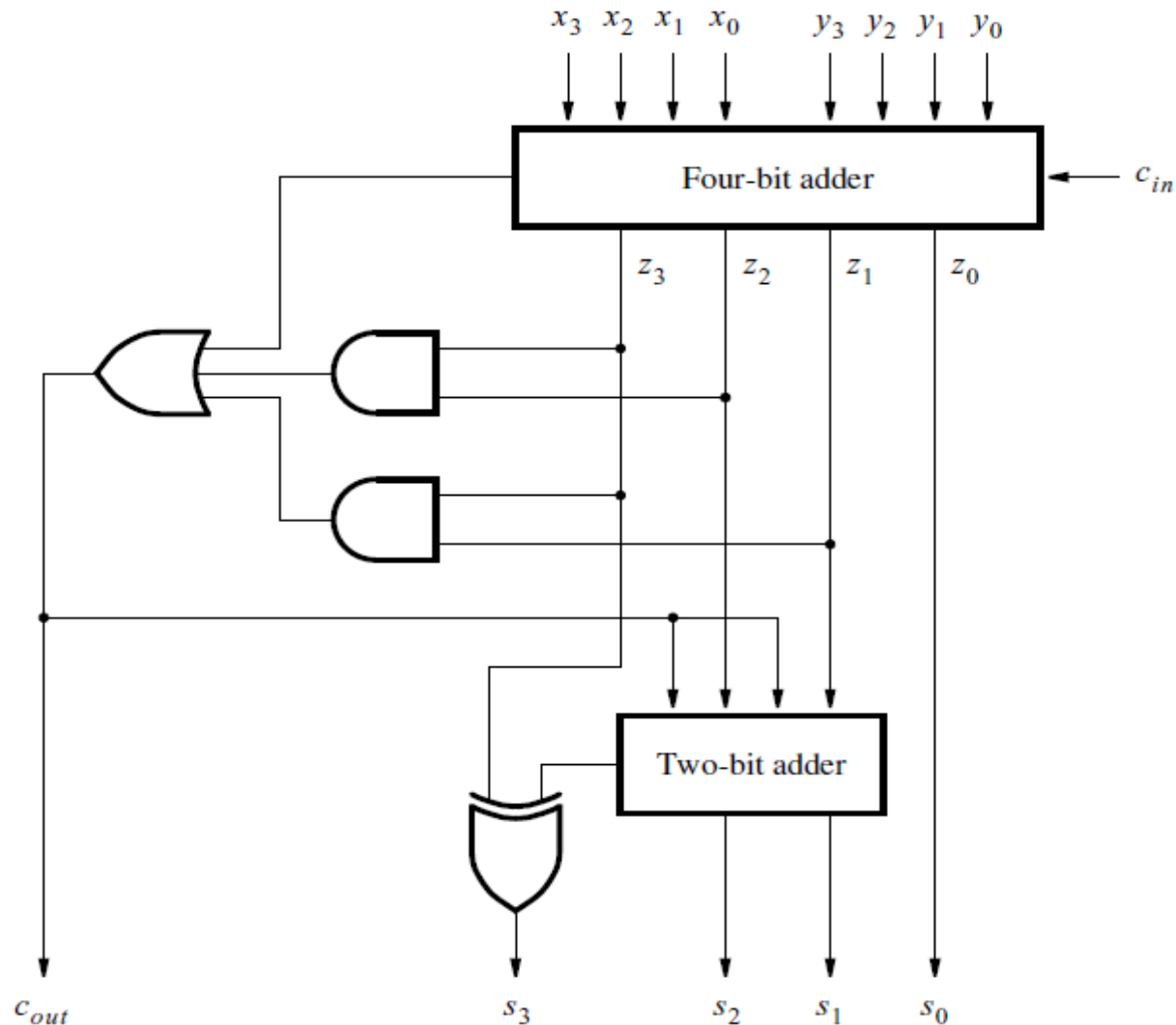


Figure 3.41. Circuit for a one-digit BCD adder. (SKIP)

Outline

Lecture 13 & 14 & 15:

Chapter: 3.2-3.5,3.8, Page 122-166 and 178-194

Addition of Unsigned Numbers

- ▶ 1-bit Addition (half adder and full adder)
- ▶ XOR gates
- ▶ Multi-bit Addition and subtraction
- ▶ Ripple-Carry Adder
- ▶ Fast Adders
- ▶ Multiplication of Unsigned Numbers(*)
- ▶ Verilog Basic 2 and Examples

3.5 Design of Arithmetic Circuits Using CAD Tools

DESIGN OF ARITHMETIC CIRCUITS USING VERILOG

How to write the hierarchical code for a ripple-carry adder?

```
module fulladd (Cin, x, y, s, Cout);  
    input Cin, x, y;  
    output s, Cout;
```

```
    xor (s, x, y, Cin);  
    and (z1, x, y);  
    and (z2, x, Cin);  
    and (z3, y, Cin);  
    or (Cout, z1, z2, z3);
```



endmodule

Figure 3.18. Verilog code for the full-adder using gate level primitives.

```
module fulladd (Cin, x, y, s, Cout);  
    input Cin, x, y;  
    output s, Cout;
```

```
    xor (s, x, y, Cin);  
    and (z1, x, y),  
        (z2, x, Cin)  
        (z3, y, Cin);  
    or (Cout, z1, z2, z3);
```

Not Suggested!!

endmodule

Figure 3.19. Another version of Verilog code from Figure 3.18.

3.5.2 DESIGN OF ARITHMETIC CIRCUITS USING VERILOG

```
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
```

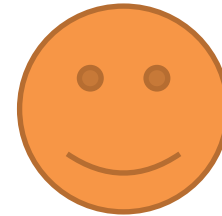


Figure 3.20. Verilog code for the full-adder using continuous assignment.

```
module fulladd (Cin, x, y, s, Cout);  
    input Cin, x, y;  
    output s, Cout;  
  
    assign s = x ^ y ^ Cin,  
        Cout = (x & y) | (x & Cin) | (y & Cin);  
  
endmodule
```

Not Suggested!!

3.5.2

DESIGN OF ARITHMETIC CIRCUITS USING VERILOG

```
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
```



Maintain the sequence

Figure 3.22. Verilog code for a four-bit adder.

3.5.3 USING VECTORED SIGNALS

Multibit signals can be represented in Verilog code as a multibit vector. An example of an input vector is

```
input [3:0] X;      wire [3:1] C;  
module adder4 (carryin, X, Y, S, carryout);  
    input carryin;  
    input [3:0] X, Y;  
    output [3:0] S;  
    output carryout;  
    wire [3:1] C;  
  
    fulladd stage0 (carryin, X[0], Y[0], S[0], C[1]);  
    fulladd stage1 (C[1], X[1], Y[1], S[1], C[2]);  
    fulladd stage2 (C[2], X[2], Y[2], S[2], C[3]);  
    fulladd stage3 (C[3], X[3], Y[3], S[3], carryout);  
endmodule
```



Support partial
selection

Figure 3.23. A four-bit adder using vectors.

USING A GENERIC SPECIFICATION

How to define a module that could be used to implement an adder of any size?

Verilog allows the use of general parameters that can be given a specific value as desired.

For example:

$X[n-1:0]$.
parameter n = 4;
the bit range of X is [3:0]

Default Value, may be updated
When instantiation

Figure 3.23. A four-bit adder using vectors.

USING A GENERIC SPECIFICATION

```
module addern (carryin, X, Y, S, carryout);
```

```
  parameter n=32;
```

```
  input carryin;
```

```
  input [n-1:0] X, Y;
```

```
  output reg [n-1:0] S;
```

```
  output reg carryout;
```

```
  reg [n:0] C;
```

```
  integer k;
```

```
  always @(X, Y, carryin)
```

```
  begin
```

```
    C[0] = carryin;
```

```
    for (k = 0; k < n; k = k+1)
```

```
    begin
```

```
      S[k] = X[k] ^ Y[k] ^ C[k];
```

```
      C[k+1] = (X[k] & Y[k]) | (X[k] & C[k]) | (Y[k] & C[k]);
```

```
    end
```

```
    carryout = C[n];
```

```
  end
```

```
endmodule
```

**‘For’ statement
Not Suggested!!**

USING A GENERIC SPECIFICATION

```
module addern (carryin, X, Y, S, carryout);
```

```
  parameter n=32;
```

```
  input carryin;
```

```
  input [n-1:0] X, Y;
```

```
  output [n-1:0] S;
```

```
  output carryout;
```

```
  wire [n:0] C;
```

```
  genvar i;
```

```
  assign C[0] = carryin;
```

```
  assign carryout = C[n];
```

```
  generate
```

```
    for (i = 0; i <= n - 1; i = i+1)
```

```
      begin:addbit
```

```
        fulladd stage (C[i], X[i], Y[i], S[i], C[i+1]);
```

```
      end
```

```
  endgenerate
```

```
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
```

Figure 3.25. A ripple-carry adder specified by using the **generate** statement.

3.5.5 NETS AND VARIABLES IN VERILOG

- **Nets**: Connections between logic elements are defined using nets.

1). A net represents a node in a circuit.

2). It can be a scalar that represents a single connection or a vector that represents multiple connections.

- **Variables**: Signals produced by procedural statements are referred to as variables.

2). A variable can be assigned a value in one Verilog statement, and it retains this value until it is overwritten by a subsequent assignment statement.

3). There are two types of variables: **reg** and **integer**.

3.5.6 ARITHMETIC ASSIGNMENT STATEMENTS

- Verilog implements such operations using arithmetic assignment statements and vectors.

For example:

```
input [n-1:0] X,Y;  
output [n-1:0] S;  
S = X + Y;
```

```
module addern (carryin, X, Y, S);  
    parameter n = 32;  
    input carryin;  
    input [n-1:0] X, Y;  
    output reg [n-1:0] S;  
  
    always @(X, Y, carryin)  
        S = X + Y + carryin;  
  
endmodule
```

*EDA tool will
synthesize the “+”
into the connection
of basic logic gates*

ARITHMETIC ASSIGNMENT STATEMENTS

```
module addern (carryin, X, Y, S, carryout, overflow);  
    parameter n = 32;  
    input carryin;  
    input [n-1:0] X, Y;  
    output reg [n-1:0] S;  
    output reg carryout, overflow;  
  
    always @(X, Y, carryin)  
    begin  
        S = X + Y + carryin;  
        carryout = (X[n-1] & Y[n-1]) | (X[n-1] & ~S[n-1]) | (Y[n-1] & ~S[n-1]);  
        overflow = (X[n-1] & Y[n-1] & ~S[n-1]) | (~X[n-1] & ~Y[n-1] & S[n-1]);  
    end  
  
endmodule
```

Figure 3.27. An n -bit adder with carry-out and overflow signals.

ARITHMETIC ASSIGNMENT STATEMENTS

```
module addern (carryin, X, Y, S, carryout, overflow);  
    parameter n = 32;  
    input carryin;  
    input [n-1:0] X, Y;  
    output reg [n-1:0] S;  
    output reg carryout, overflow;  
    reg [n:0] Sum;  
  
    always @(X, Y, carryin)  
    begin  
        Sum = {1'b0,X} + {1'b0,Y} + carryin;  
        S = Sum[n-1:0];  
        carryout = Sum[n];  
        overflow = (X[n-1] & Y[n-1] & ~S[n-1]) | (~X[n-1] & ~Y[n-1] & S[n-1]);  
    end  
  
    endmodule
```

Figure 3.28. An alternative specification of an n -bit adder with carry-out and overflow signals.



ARITHMETIC ASSIGNMENT STATEMENTS

```
module addern (carryin, X, Y, S, carryout, overflow);  
    parameter n = 32;  
    input carryin;  
    input [n-1:0] X, Y;  
    output reg [n-1:0] S;  
    output reg carryout, overflow;  
  
    always @(X, Y, carryin)  
    begin  
        {carryout, S} = X + Y + carryin;  
        overflow = (X[n-1] & Y[n-1] & ~S[n-1]) | (~X[n-1] & ~Y[n-1] & S[n-1]);  
    end  
  
endmodule
```

Figure 3.29. Simplified complete specification of an n -bit adder.

ARITHMETIC ASSIGNMENT STATEMENTS

```
module fulladd (Cin, x, y, s, Cout);  
    input Cin, x, y;  
    output reg s, Cout;  
  
    always @(x, y, Cin)  
        { Cout, s } = x + y + Cin;  
  
endmodule
```

Figure 3.30. Behavioral specification of a full-adder.

3.5.7 MODULE HIERARCHY IN VERILOG CODE

```
module adder_hier (A, B, C, D, S, T, overflow);  
    input [15:0] A, B;  
    input [7:0] C, D;  
    output [16:0] S;  
    output [8:0] T;  
    output overflow;  
  
    wire o1, o2; // used for the overflow signals  
  
    addern U1 (1'b0, A, B, S[15:0], S[16], o1);  
    defparam U1.n = 16;  
    addern U2 (1'b0, C, D, T[7:0], T[8], o2);  
    defparam U2.n = 8;  
  
    assign overflow = o1 | o2;  
  
endmodule
```

Figure 3.31. An example of setting parameter values in Verilog code.



MODULE HIERARCHY IN VERILOG CODE

```
module adder_hier (A, B, C, D, S, T, overflow);  
    input [15:0] A, B;  
    input [7:0] C, D;  
    output [16:0] S;  
    output [8:0] T;  
    output overflow;  
  
    wire o1, o2; // used for the overflow signals  
  
    addern #(16) U1 (1'b0, A, B, S[15:0], S[16], o1);  
    addern #(8) U2 (1'b0, C, D, T[7:0], T[8], o2);  
  
    assign overflow = o1 | o2;  
  
endmodule
```

3.5.8 REPRESENTATION OF NUMBERS IN VERILOG CODE

- Numbers can be given as constants in Verilog code.
- They can be given as **binary (b)**, **octal (o)**, **hexadecimal (h)**, or **decimal (d)** numbers.

12'b100010101001

12'o4251

12'h8A9

12'd2217

'b100010110

'o426

'h116

278

REPRESENTATION OF NUMBERS IN VERILOG CODE

- The value of a positive number does not change if 0s are appended as the most-significant bits;
- the value of a negative number does not change if 1s are appended as the most-significant bits.
- Such replication of the sign bit is called sign extension.

Suppose that A is an eight-bit vector and B is a four-bit vector.

$$S = A + B;$$

$$S = A + \{4\{B[3]\}, B\};$$



3.8 Examples

- ▶ You Read them after class and now we Skip

Conclusion

- ▶ Understanding the ADDER and fast ADDER
- ▶ XOR Gate
- ▶ Difference of the ripple-carry and look-ahead carry adder
- ▶ Some concept like delay, critical path, fan-in, fan-out ..
- ▶ Advanced Verilog

- ▶ Textbook Reading: 3.2-3.5, 3.8, 3.7.3

- ▶ Assignment: **3.1, 3.2, 3.4**
▶ **(if you have done ,please skip)**
▶ **3.5, 3.7(how to prove), 3.14**
▶ **3.21, 3.22**

Next Lecture

- ▶ **Lecture 16:**
Combinational Logic: Circuit Building Blocks

