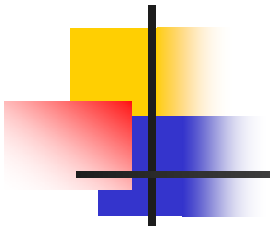
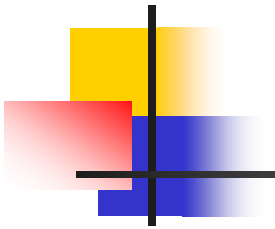


Chapter 4

Combinational Logic

Combinational Circuits



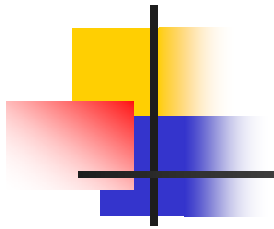
- Logic circuits for digital systems may be combinational or sequential.
- Combinational logic circuits
  - consist of logic gates
  - outputs = present combination of inputs
- Sequential logic circuits
  - employ storage elements & logic gates
  - outputs are a function of the inputs & the state of storage elements (a function of previous inputs)

Block Diagram of Combinational Circuit

- A combinational circuits
  - 2^n possible combinations of input values

- m outputs : m Boolean functions of n input variables
  - Specific functions
    - Adders, subtractors, comparators, decoders, encoders, and multiplexers
    - MSI circuits or standard cells

Purpose of Analysis



- The analysis of a combinational circuit
  - to determine the function that the circuit implements.
- Given a logic diagram. Find
  - make sure that it is combinational not sequential
    - No feedback path
  - a set of Boolean functions
  - a truth table
  - a possible explanation of the circuit operation

Analysis Procedure

- Step 1. Make sure that the given circuit is a combinational circuit
  - the circuit has logic gates
  - without feedback
  - without memory elements

- Step 2. Obtain
  - the Boolean functions, or
  - the truth table (one by one)

A Straight-Forward Procedure

$F_2 = AB+AC+BC$

$T_1 = A+B+C$

$T_2 = ABC$

$T_3 = F2'T1$

$F_1 = T3+T2$

A	B	C	F_2	F_2'	T_1	T_2	T_3	F_1	F_2
0	0	0	0	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

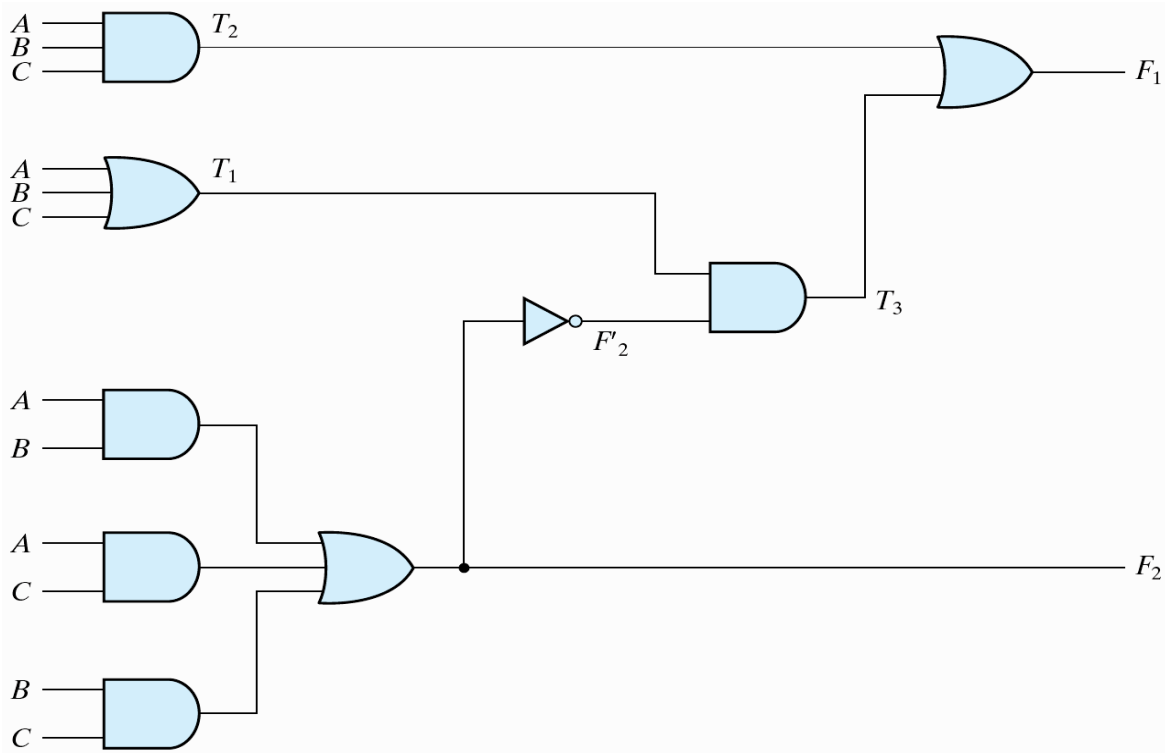


Fig. 4-2 Logic Diagram for Analysis Example

Truth Table

Table 4.1
Truth Table for the Logic Diagram of Fig. 4.2

A	B	C	F_2	F'_2	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

Design Procedure

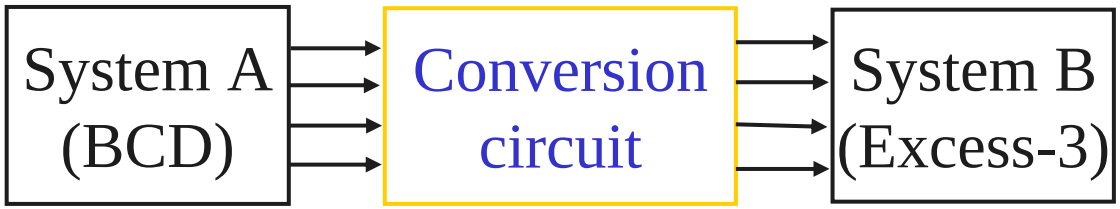
- The design procedure of combinational circuits
  - State the problem (system specifications)
  - determine the inputs and outputs
  - the input and output variables are assigned symbols
  - derive the truth table
  - derive the simplified Boolean functions of each inputs
  - draw the logic diagram and verify the correctness

implementation constraints :

# of gates # of inputs to gate
propagation time # of interconnections
driving capability of each gate

Example: BCD ➔ Excess-3 (1/3)

BCD ➔ Excess-3



1. Spec: BCD ➔ Excess-3
2. 4 inputs, 4 outputs

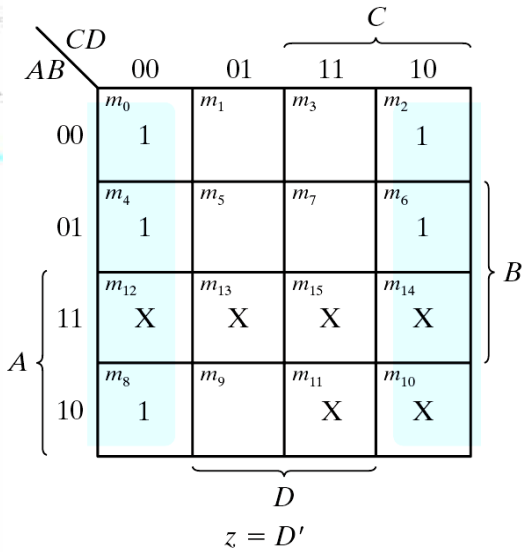
Table 4.2
Truth Table for Code-Conversion Example

Input BCD				Output Excess-3 Code			
A	*B*	*C*	*D*	*w*	*x*	*y*	*z*
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

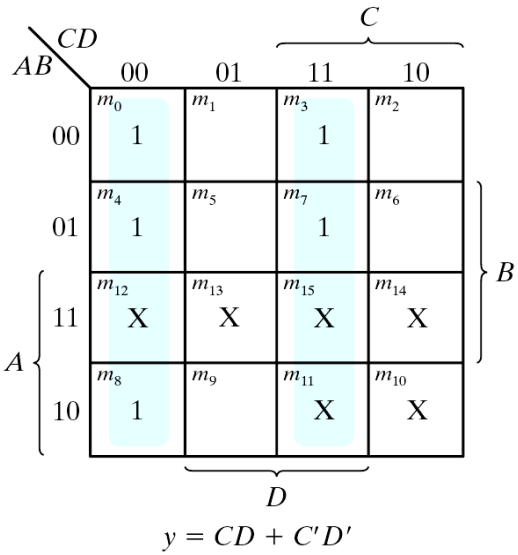
Example: BCD ➔ Excess-3 (2/3)

Table 4.2
Truth Table for Code-Conversion Example

Input BCD				Output Excess-3 Code			
A	**B**	**C**	**D**	**w**	**x**	**y**	**z**
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

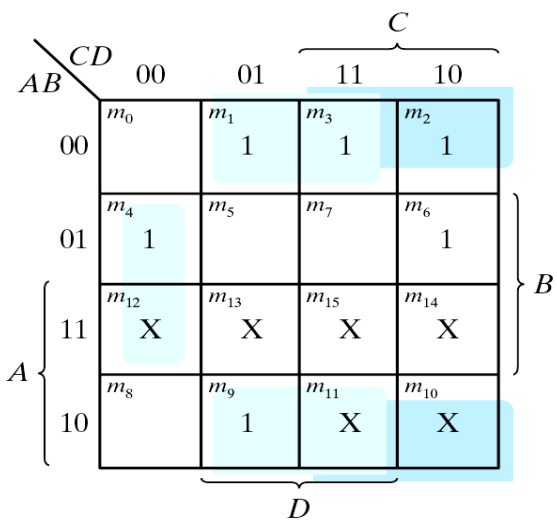


$z = D'$

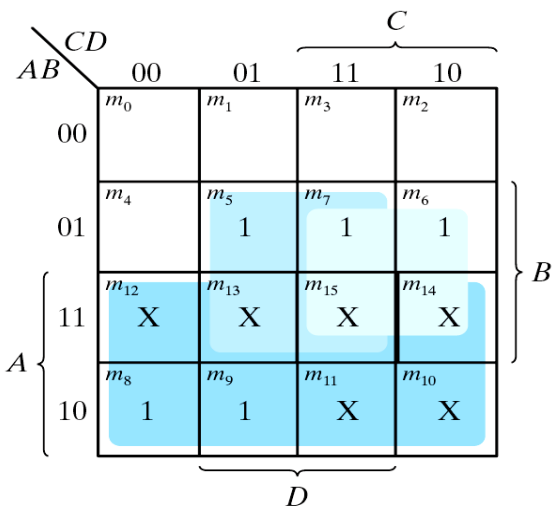


$y = CD + C'D'$

K-map
Karnaugh-map
minimization



$x = B'C + B'D + BC'D'$



$w = A + BC + BD$

$w = A+BD+BC$
$x = B'D+B'C+BC'$
$y = C'D'+CD$
$z = D$

Example: BCD ➔ Excess-3 (3/3)

Logic diagram (not unique implementation)

$w = A+BD+BC$
$x = B'D+B'C+BC'D'$
$y = C'D'+CD$
$z = D$

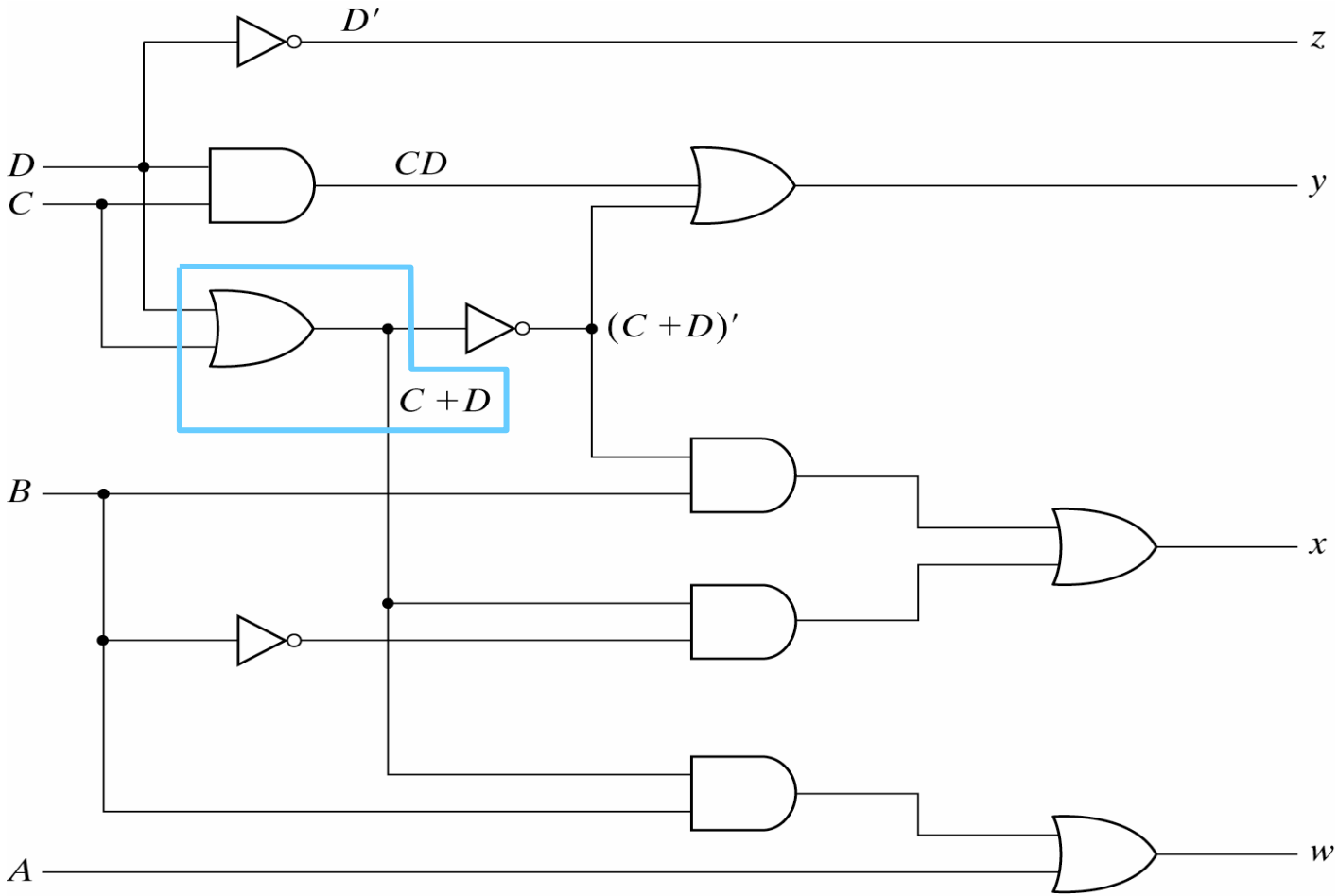


Fig. 4-4 Logic Diagram for BCD to Excess-3 Code Converter

Adder

- addition
  - the most basic arithmetic operation
- binary adder
- half adder
  - addition of 2 bits
- full adder
  - addition of 3 bits
  - (2 bits & a previous carry)

half adder

↓

full adder (FA)

↓ … ↓ n

n-bit binary adder (n FAs in cascade)

↓

subtractor (complementing circuit)

↓

adder-subtractor

Half Adder (1/2)

- Half adder (sum of two input bits)
  - 0 + 0 = 0 ; 0 + 1 = 1 ; 1 + 0 = 1 ; 1 + 1 = 10
  - two input variables: *x*, *y*
  - two output variables: C (carry), S (sum)
  - truth table

Table 4.3
Half Adder

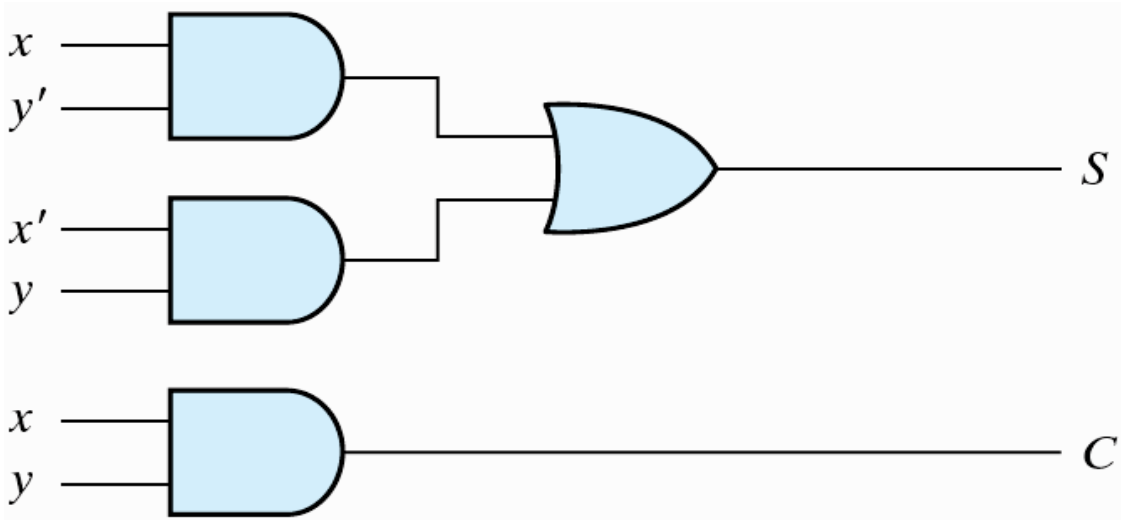
x	*y*	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

- S = $x'y + xy' = x \oplus y$ **(XOR)**
- C = xy **(AND)**

Half Adder (2/2)

Table 4.3
Half Adder

		carry	sum
x	**y**	**C**	**S**
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

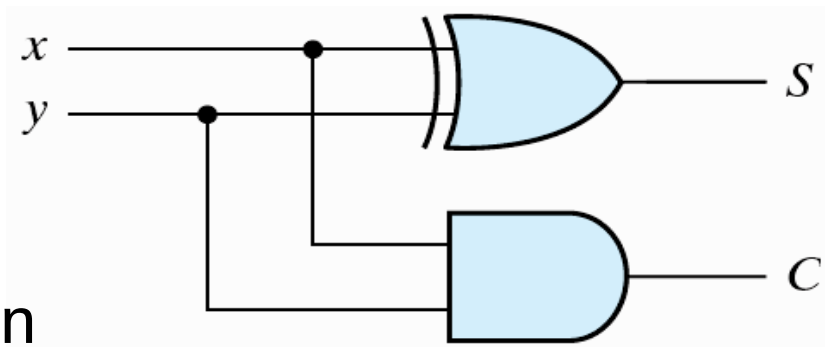


(a) $S = xy' + x'y$
$C = xy$

$$S = x'y + xy'$$

$$C = xy$$

➔ not unique implementation



(b) $S = x \oplus y$
$C = xy$

Full Adder (1/3)

- The arithmetic sum of three input bits
- three input bits
  - x, y: two significant bits
  - z: the carry bit from the previous lower significant bit
- two output bits: C, S

1. **Specifications**
2. **Determine the inputs and outputs**
3. **Derive the truth table**
4. **K-map minimization**
5. **Draw the logic diagram**

One-bit adder

Table 4.4
Full Adder

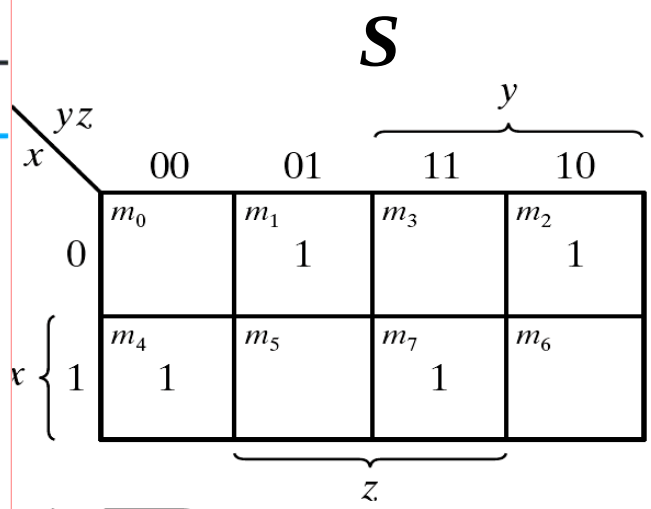
x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

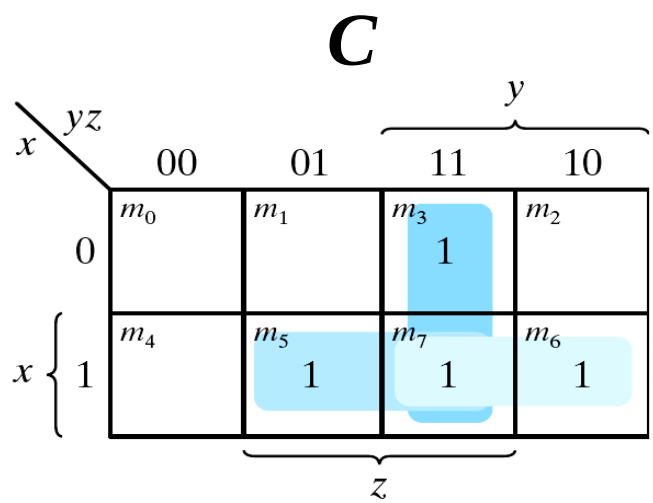
Full Adder (2/3)

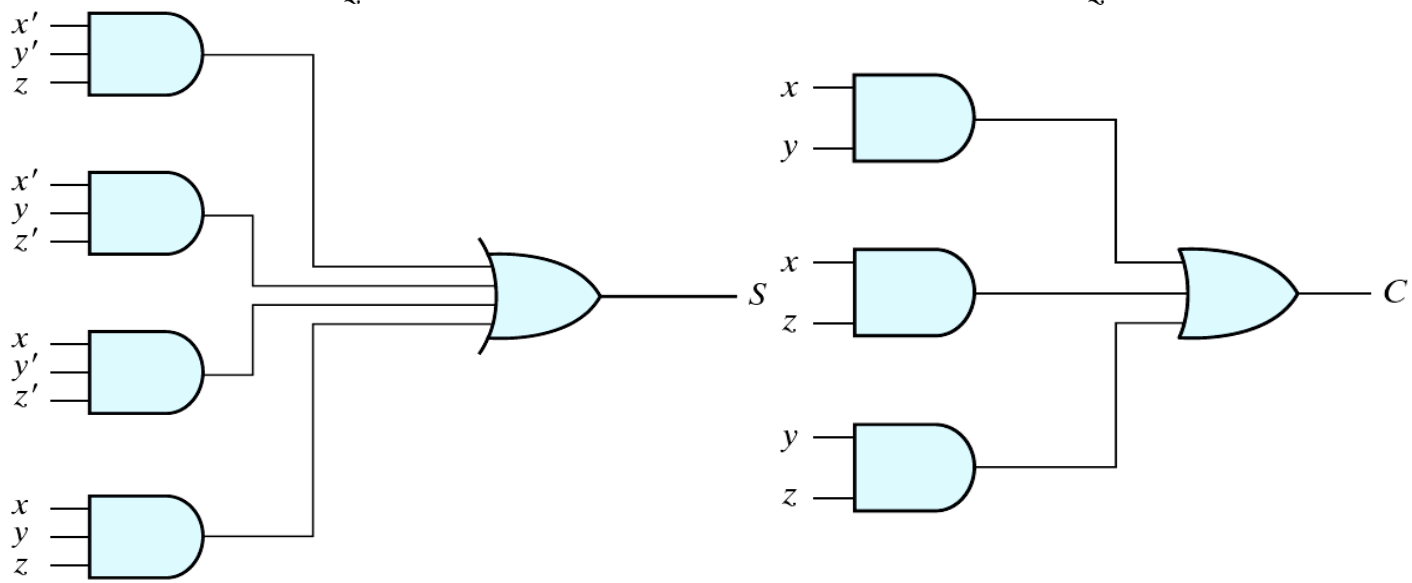
Table 4.4
Full Adder

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = x'y'z + x'yz'$$
$$+xy'z' + xyz$$
$$C = xy + xz + yz$$

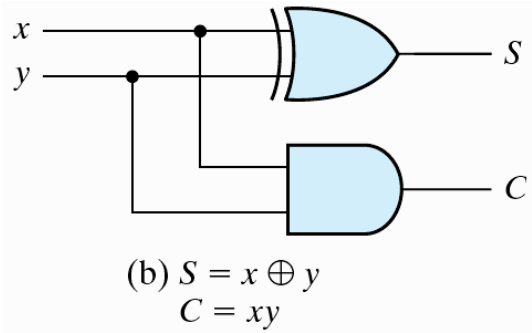






Full Adder (3/3)

- S = x'y'z+x'yz'+ xy'z'+xyz
- C = xy + xz + yz
- S = z⊕ (x⊕y)
  = z'(xy'+x'y)+z(xy'+x'y)'= …
  = xy'z+x'yz+ xy

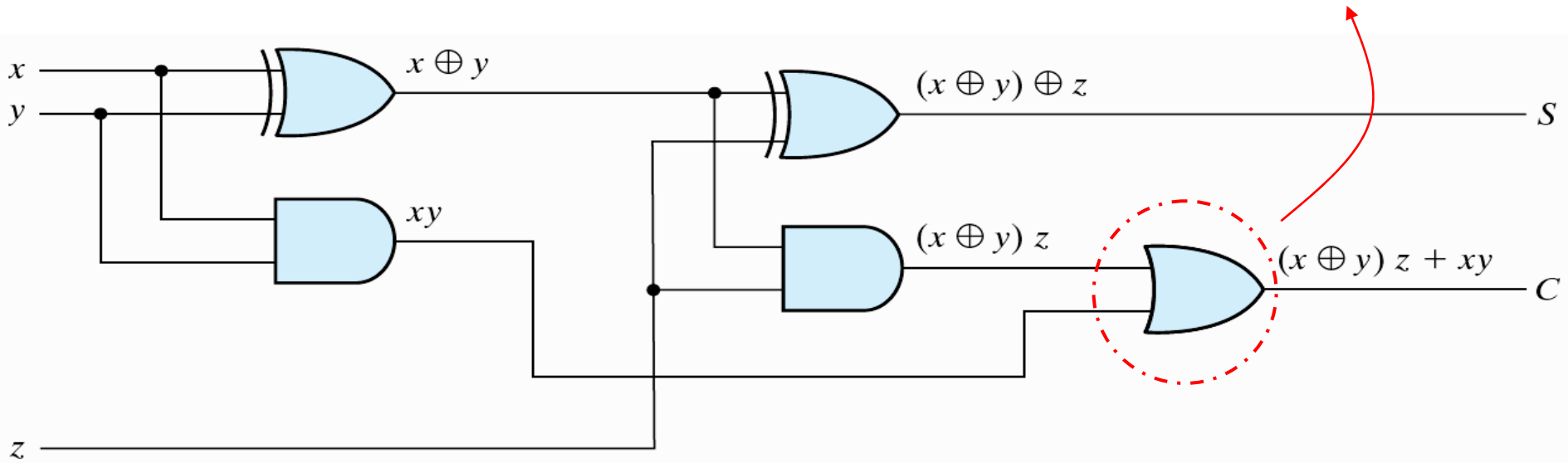


(b) $S = x \oplus y$
$C = xy$

a half adder

another implementation

a full adder == 2 half adders + 1 OR gate



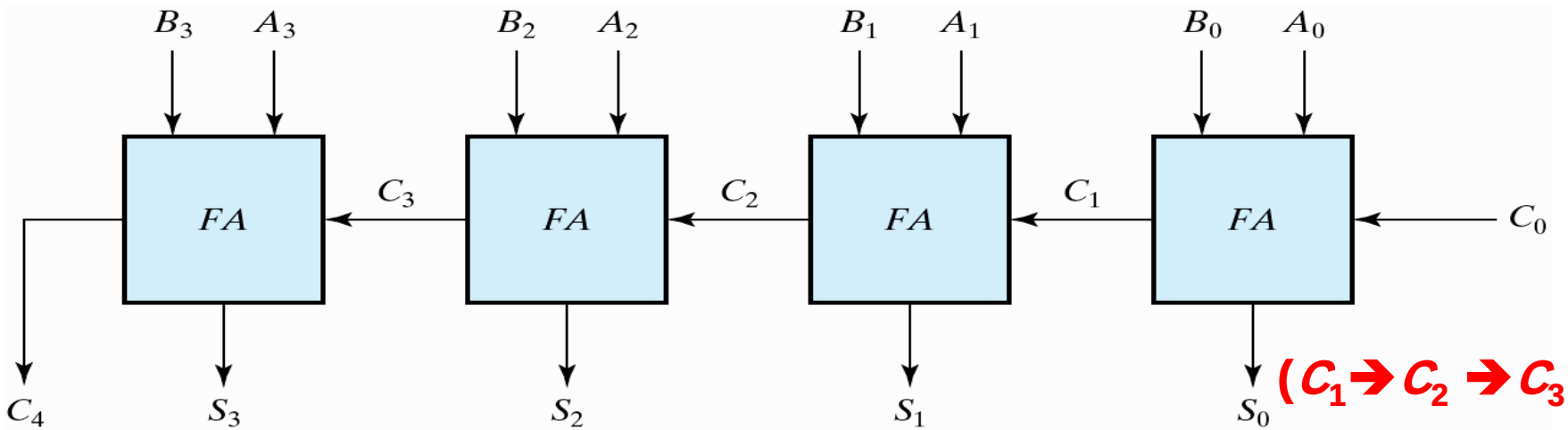
A 4-bit Binary Adder

Subscript i:	3	2	1	0	
	C_3	C_2	C_1	C_0	
Input carry	0	1	1	0	C_i
Augend	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output carry	0	0	1	1	C_{i+1}
	C_4	C_3	C_2	C_1	

Don't use the truth table with 2^9 entries

4-bit binary ripple-carry adder

Why?



(C_1 → C_2 → C_3 → C_4)

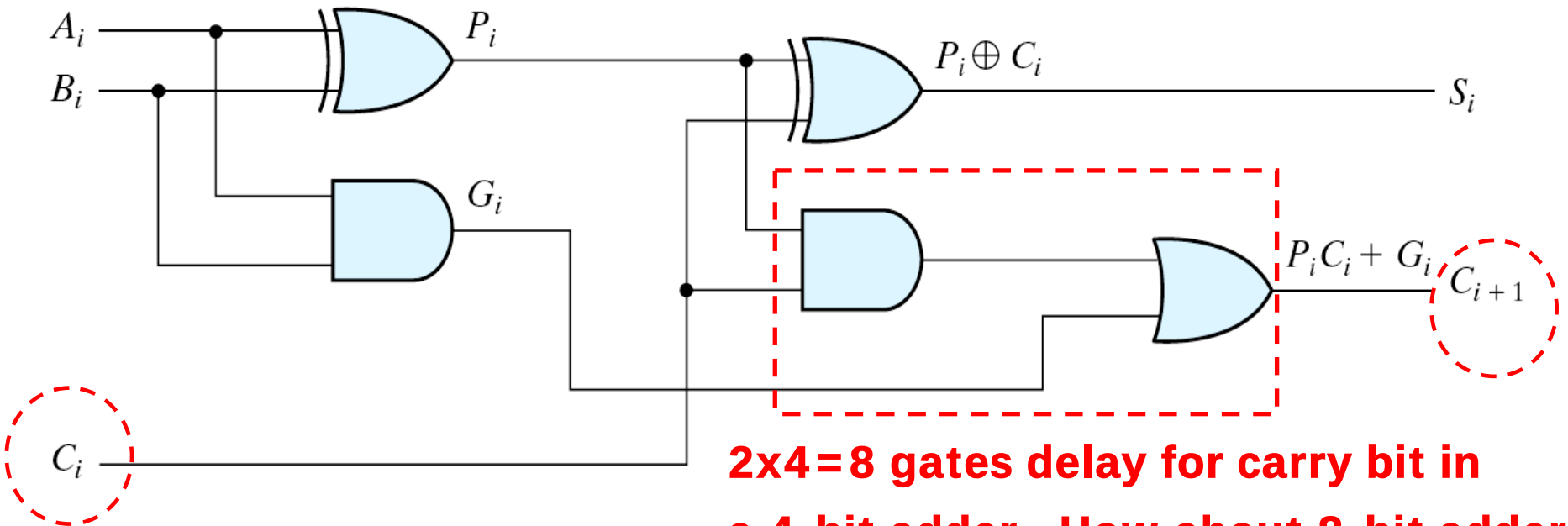
C_4 and C_3 output the wrong value at the beginning (→behavior of hardware circuit)

But C_4 becomes correct as soon as C_3 is correct + a little delay.

C_3 becomes correct as soon as C_2 is correct + a little delay. .....

Carry Propagation (1/2)

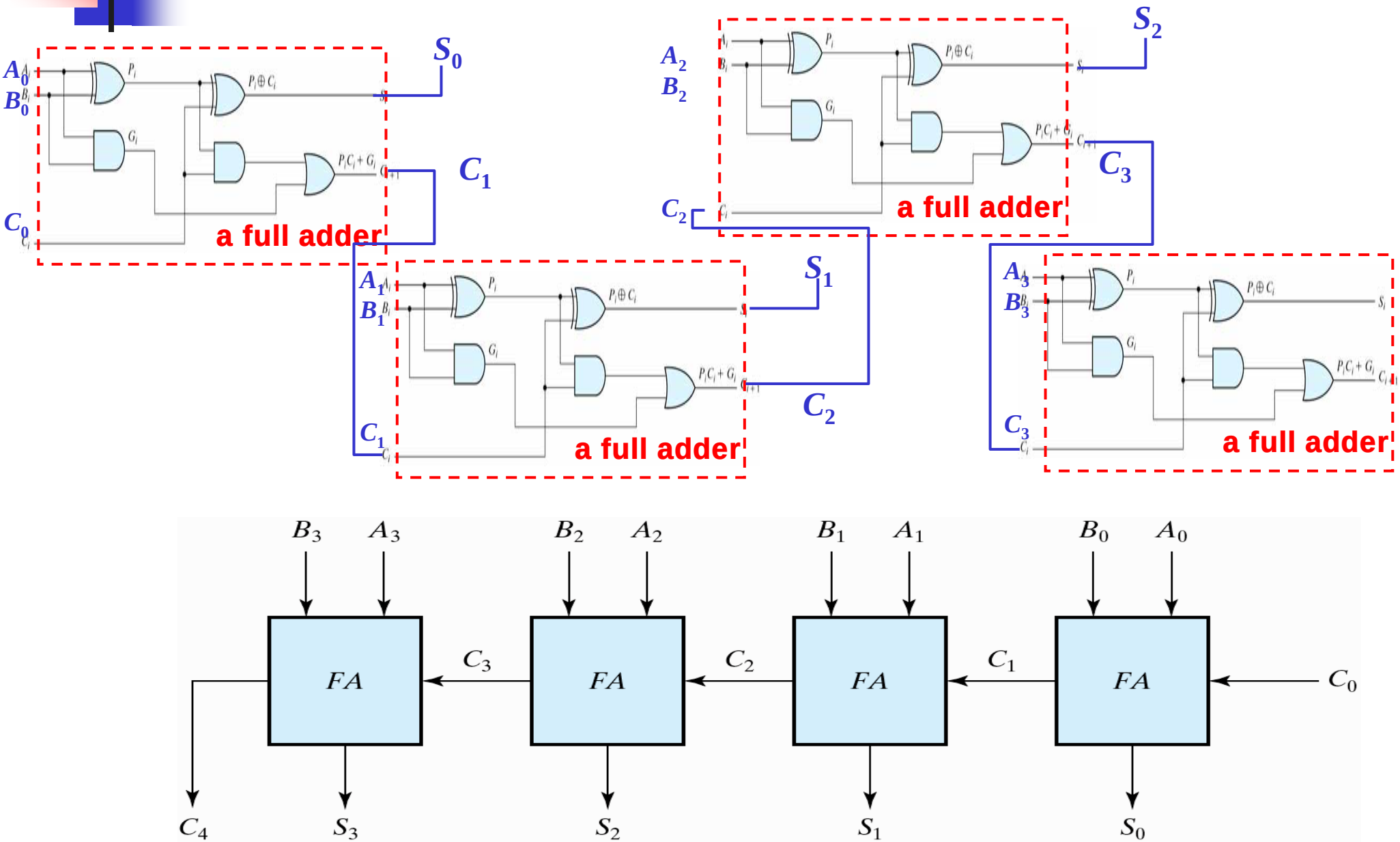
- when the correct outputs are available
- the critical path counts (the worst case)
- $(A_1,B_1,C_1) > C_2 > C_3 > C_4$
- > 8 gate levels



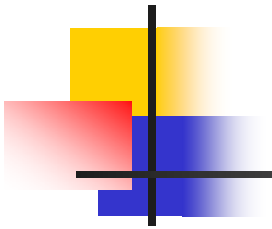
2x4=8 gates delay for carry bit in a 4-bit adder. How about 8-bit adder?

Fig. 4-10 Full Adder with P and G Shown

Carry Propagation (2/2)



Binary Ripple-Carry Adder

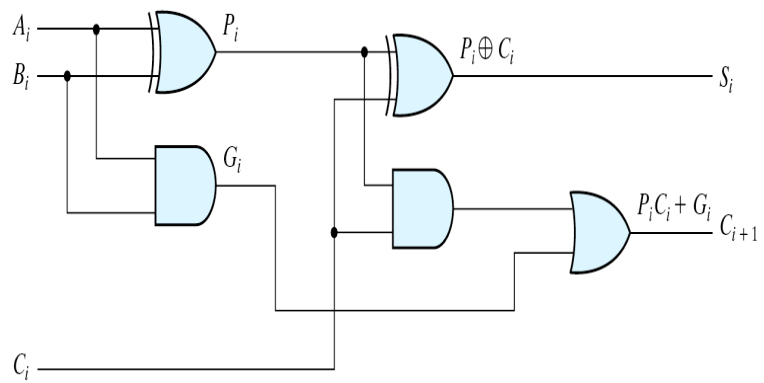


- Carry propagation time
  - 2 gate levels for each FA
  - *2n* gate levels for an *n*-bit binary ripple-carry adder
    - low speed !
- Improvements
  - Employ faster gates to reduce delays

    Carry lookahead scheme (more cost, less delay)
  - Circuit complexity (cost) v.s. delay time (speed)

Reduction of Carry Propagation Delay

- employ faster gates
- look-ahead carry (more complex mechanism, yet faster)
- carry propagate: $P_i = A_i \oplus B_i$
- carry generate: $G_i = A_i B_i$
- sum: $S_i = P_i \oplus C_i$
- carry: $C_{i+1} = G_i + P_i C_i$
- $C_1 = G_0 + P_0 C_0$
- $C_2 = G_1 + P_1 C_1 = G_1 + P_1(G_0 + P_0 C_0)$
  $= G_1 + P_1 G_0 + P_1 P_0 C_0$
- $C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$



C_{i+1} must wait for C_i

➔ break this waiting-data dependency

Let C_1, C_2 and C_3 all depend on C_0 only

Carry Lookahead

Break the data dependency !!!

Let C_1, C_2 and C_3 all depend on C_0 only.

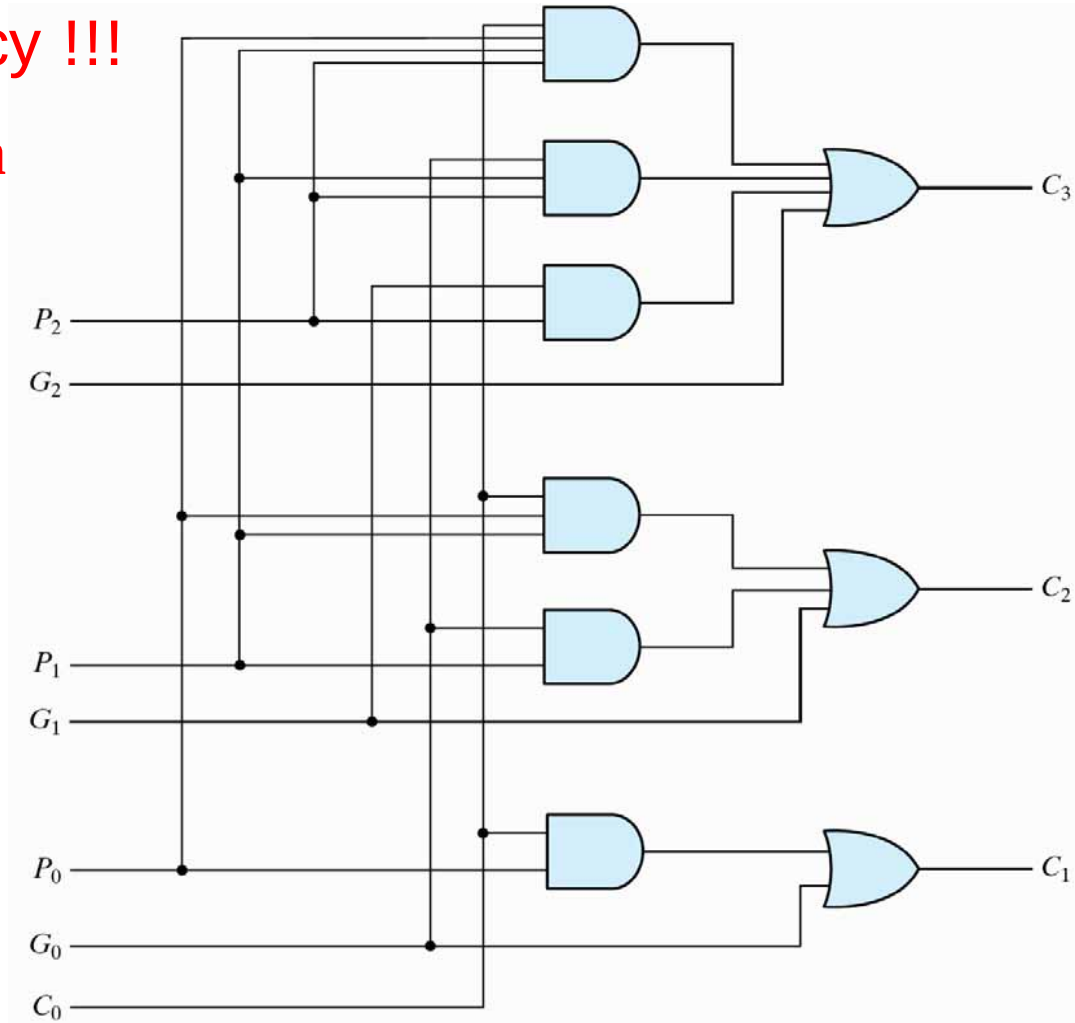
- $C_1 = G_0 + P_0C_0$
- $C_2 = G_1 + P_1C_1$

  $= G_1 + P_1(G_0 + P_0C_0)$

  $= G_1 + P_1G_0 + P_1P_0C_0$
- $C_3 = G_2 + P_2C_2$

  $= G_2 + P_2G_1 + P_2P_1G_0 +$

  $P_2P_1P_0C_0$



4-bit Carry Lookahead Adder

Less propagation delay

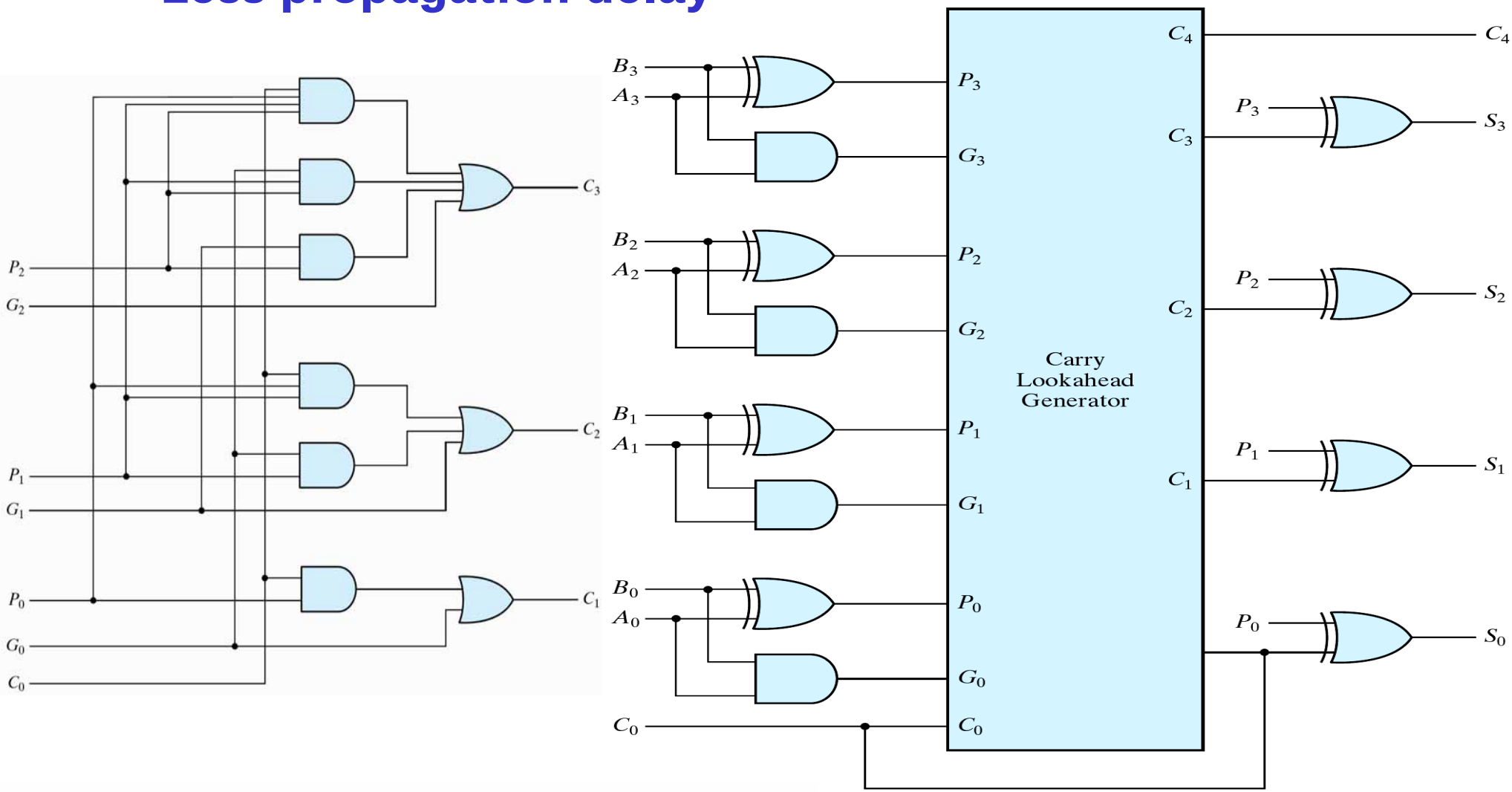


Fig. 4-12 4-Bit Adder with Carry Lookahead

4-bit Binary Adder/Subtractor

If M=0, D_3=B_3

If M=1, D_3=B_3'

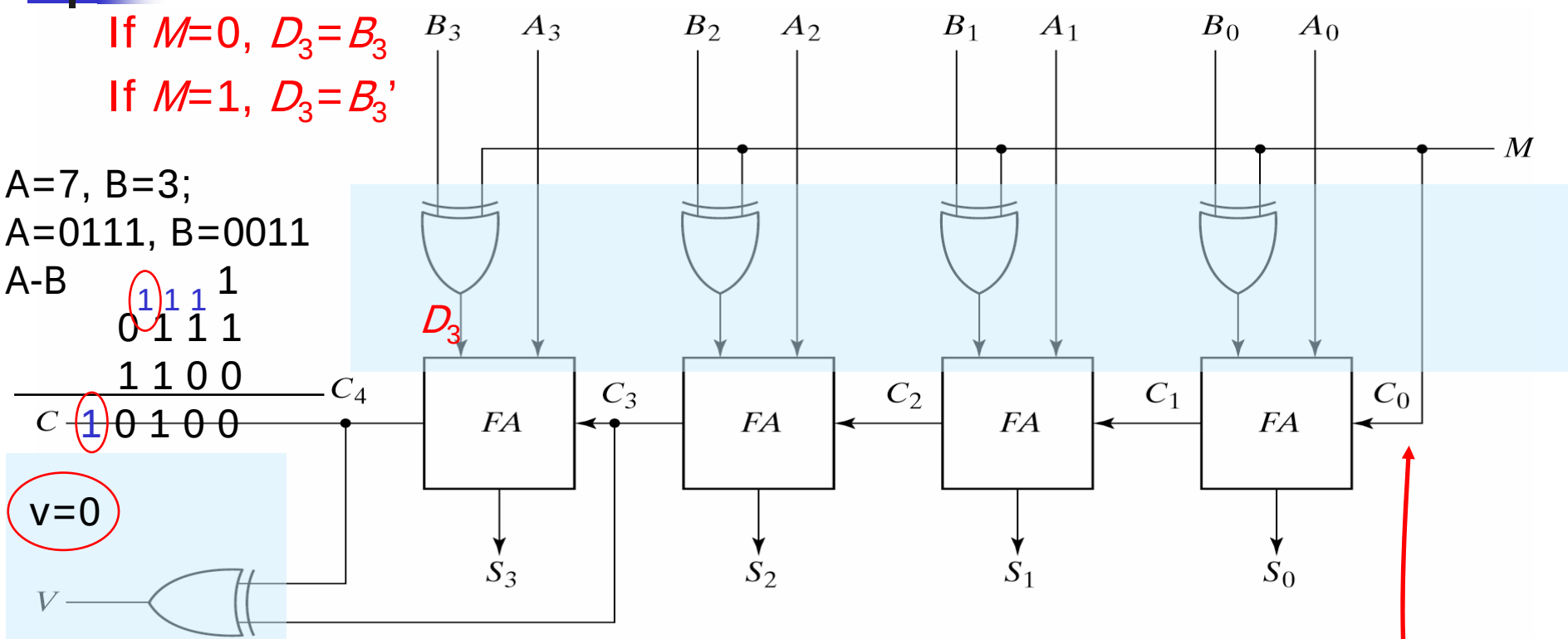


Fig. 4-13 4-Bit Adder Subtractor

A=7, B=3;
A=0111, B=0011
A-B 1
1 1 1
0 1 1 1
1 1 0 0
C 1 0 1 0 0

v=0

V=0 ➔ no overflow

A=3, B=7;
A=0011, B=0111
A-B 1
0 1 1
0 0 1 1
1 0 0 0
0 1 1 0 0

v=0

If M=0, calculate A+B (adder)

If M=1, calculate A+(-B)=A+(1's of B+1)

=A-B (subtractor)

Signed Binary Numbers

Decimal	Signed-mag.	Signed-2's C.	Signed-1's C.
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
-1	**1001**	**1111**	**1110**
-2	**1010**	**1110**	**1101**
-3	**1011**	**1101**	**1100**
-4	**1100**	**1100**	**1011**
-5	**1101**	**1011**	**1010**
-6	**1110**	**1010**	**1001**
-7	**1111**	**1001**	**1000**

(2's complement)

Overflow (1/2)

- The storage space is limited (ex. *n*-bit register)
- Add two positive numbers and obtain a negative number (overflow)
- Add two negative numbers and obtain a positive number (overflow)
- V = 0, no overflow; V = 1, overflow

8-bit register (1-bit for sign and 7-bit for magnitude, +127 ~ -128)

carries:	0	1
+70	0	1000110
+80	0	1010000
+150	1	0010110

carries:	1	0
−70	1	0111010
−80	1	0110000
−150	0	1101010

The answer is – 1101010= -106 overflow error - 0010110=-22

Overflow (2/2)

- What is an Overflow
  - Addition/subtraction of two n-digit numbers
    - ➔ a sum of (n+1) digits
- Why detect “overflow”
  - Computers need to detect “overflow” and set a flip-flop for further use
- How to detect “overflow” for signed numbers
  - By observing the carry into the sign bit position and out of the sign bit position
- How can you avoid the situation of overflow in your circuit?

Insert a bit

```
+70   0 0 1000110
+80   0 0 1010000
      0 1 0010110
```

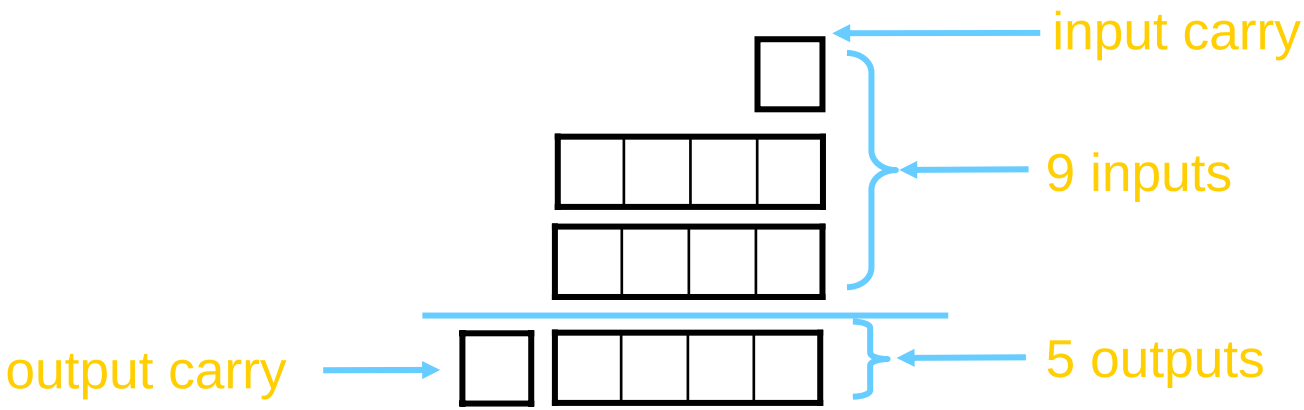
sign

extra carry bit

If -128<= a,b <= 127, use a 9-bit register for the sum (a+b).

(1-bit for sign, 1-bit for extra carry and 7-bit for magnitude)

Decimal Adder



- 4 bits are required to code a decimal digit
- There is a wide variety of possible decimal adder circuits, depending on the code (e.g. BCD, Ex-3).
- Here we consider a decimal adder for BCD code.
- max BCD sum = 9 + 9 + 1(input carry) = 19
- Design approaches for BCD
  - A truth table with 2^9 entries
  - use two binary full adders (binary to BCD)

BCD Adder: The truth table

If the binary sum<=9, BCD sum=binary sum.

If the binary sum > 9, BCD sum= 6+ binary sum.

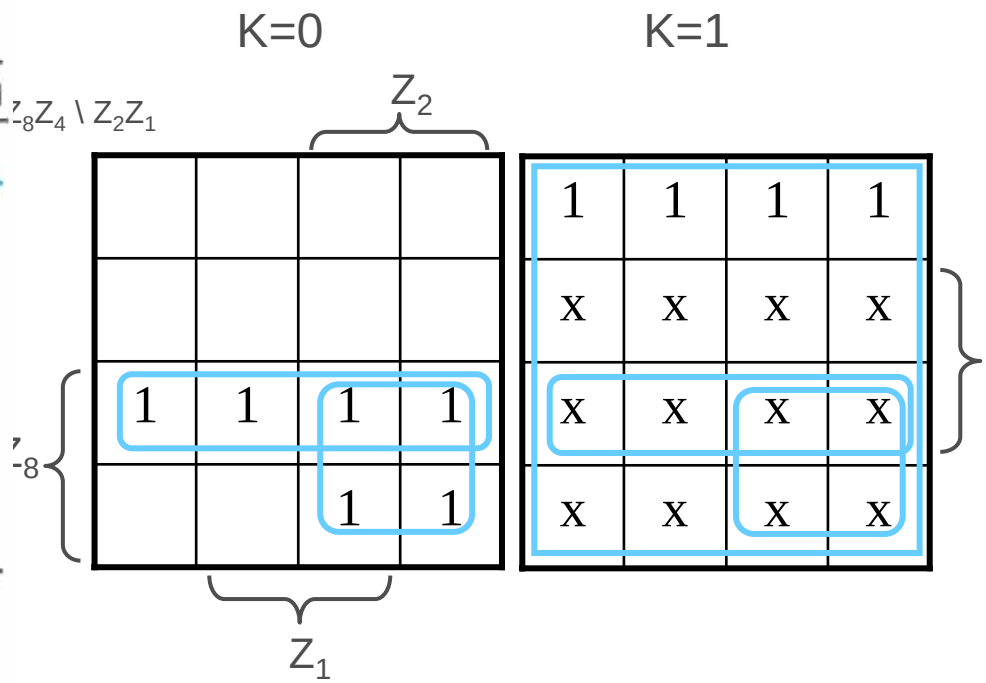
Table 4.5
Derivation of BCD Adder

Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
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

5-variable K-map

Table 4.5
Derivation of BCD Adder

Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
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



$$C = K + Z_8Z_4 + Z_8Z_2$$

$$S_8\,S_4\,S_2\,S_1 = Z_8\,Z_4\,Z_2\,Z_1 + 6$$

BCD Adder

$C = K + Z_8Z_4 + Z_8Z_2$

If the binary sum<=9,
BCD sum=binary sum.

If the binary sum > 9,
BCD sum= 6+ binary sum.

if C = 1, S = Z + 0110
if C = 0, S = Z + 0000

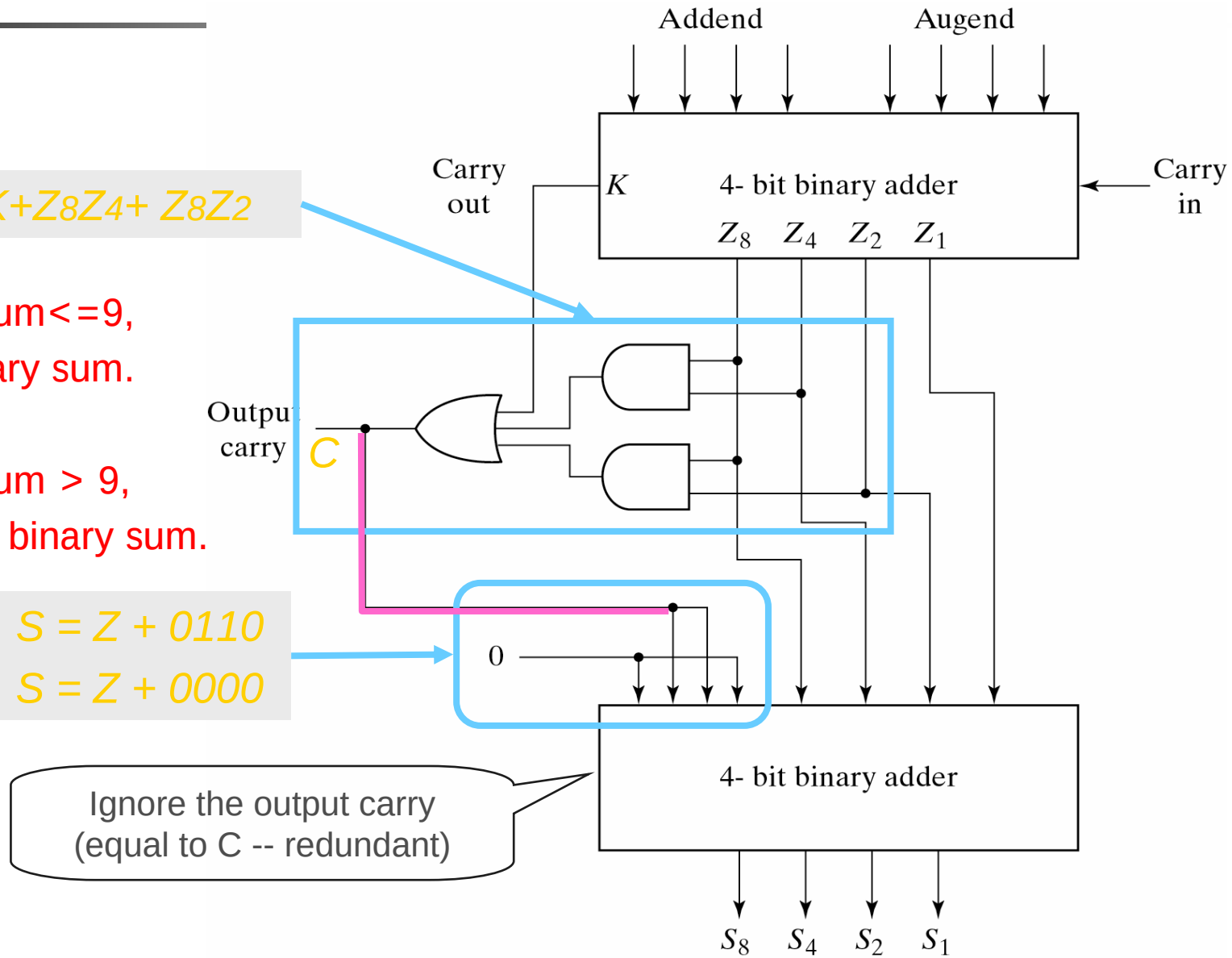
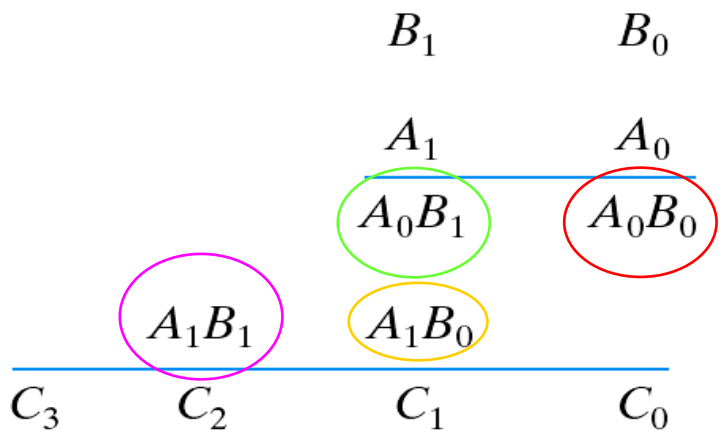


Fig. 4-14 Block Diagram of a BCD Adder

Binary Multiplier

- Partial products
  - AND operations



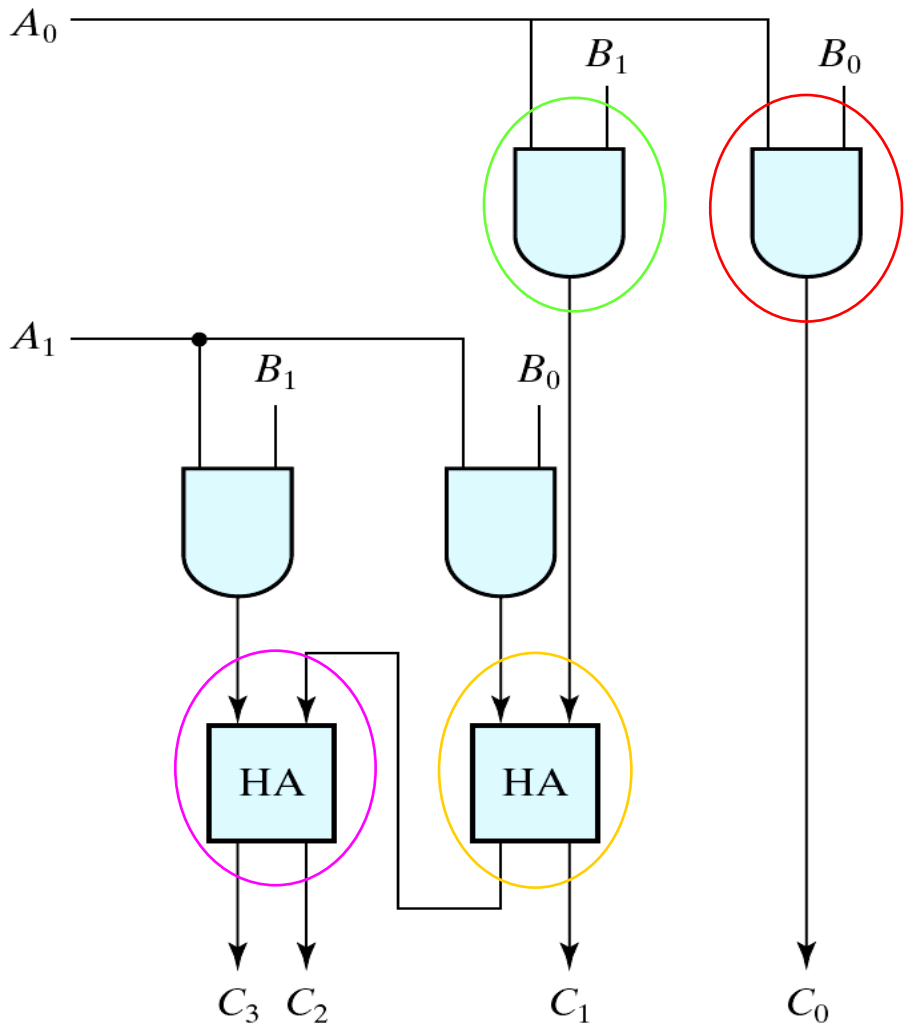
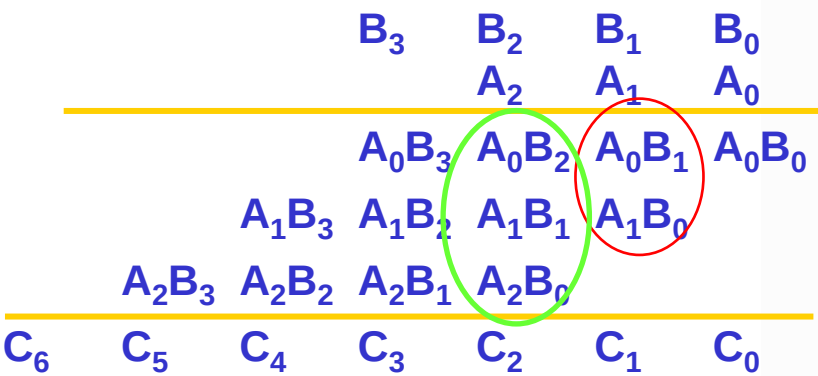
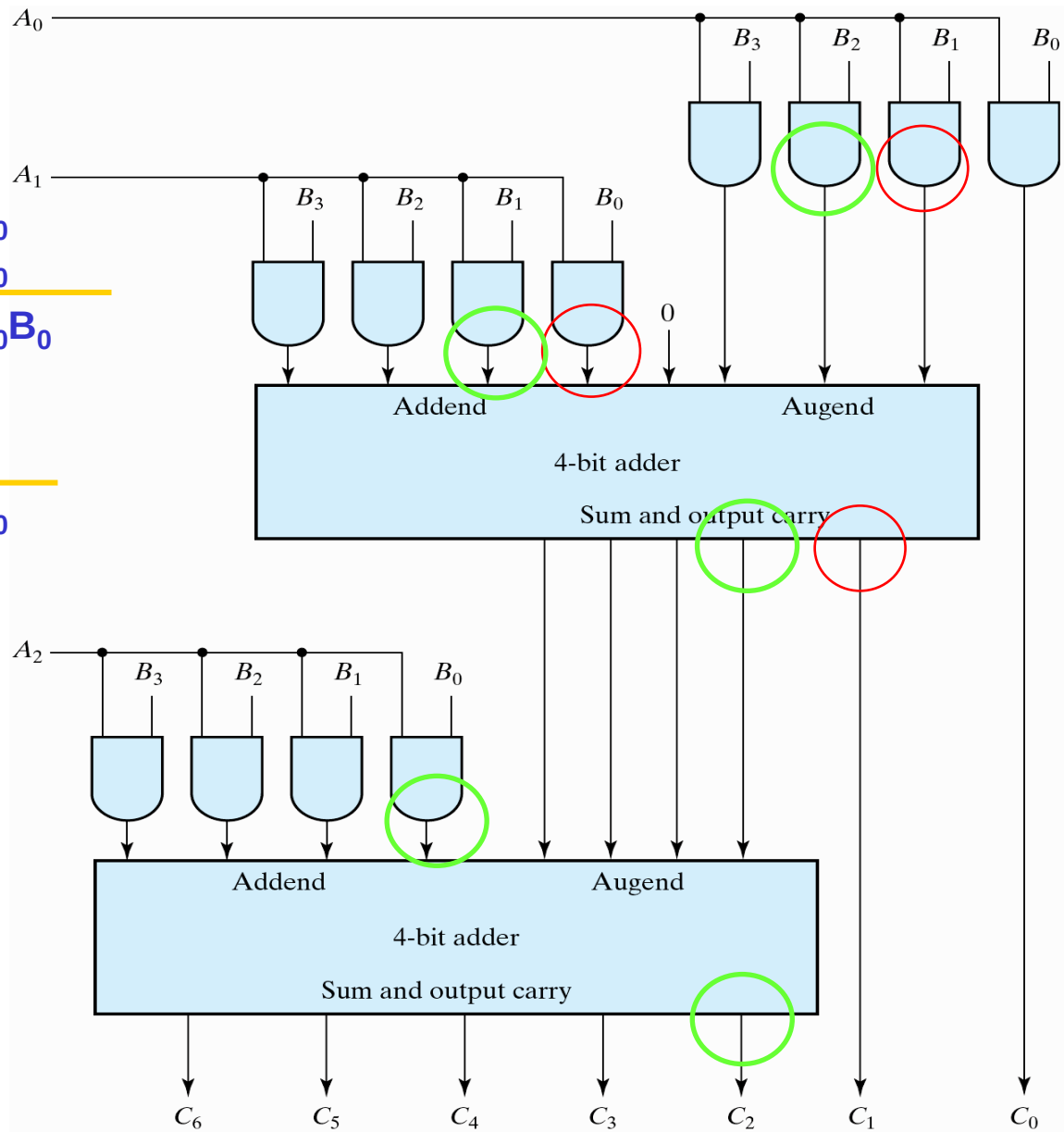


Fig. 4.15
Two-bit by two-bit binary multiplier.

4-bit by 3-bit Binary Multiplier



Four-bit by three-bit binary multiplier



Magnitude Comparator

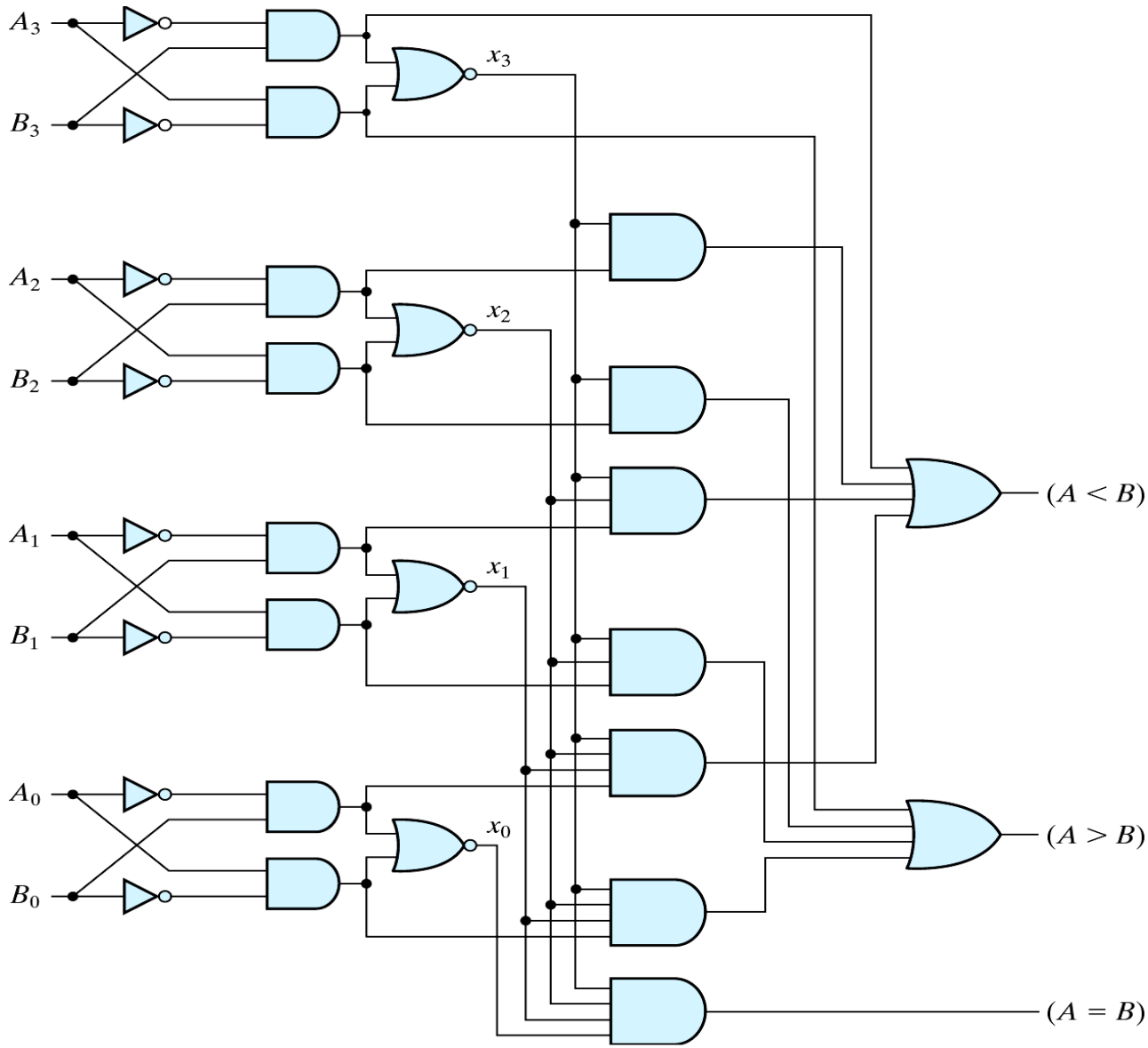
- The comparison of two n-bit numbers
  - outputs: A>B, A=B, A<B
- Design Approaches
  - the truth table ($2n$ inputs, 3 outputs)
  - $2n$ inputs → 2^{2n} possible combinations in the truth table (too cumbersome)
  - 3 binary output variables

    (A>B), (A=B), (A<B)
  - use inherent regularity of the problem
    - reduce design efforts
    - reduce human errors

Regularity for Comparison

- Algorithm -> logic
  - $A = A_3A_2A_1A_0$; $B = B_3B_2B_1B_0$
  - A=B if $A_3=B_3$, $A_2=B_2$, $A_1=B_1$and $A_1=B_1$
    - equality: $x_i = A_iB_i+A_i'B_i'$ (exclusive-nor)
    - $(A=B) = x_3x_2x_1x_0$ (If $x_i=1$ ➔ $A_i=B_i$ ➔ both o or both 1)
  - $(A>B) = A_3B_3'+x_3A_2B_2'+x_3x_2A_1B_1'+x_3x_2x_1\,A_0B_0'$
  - $(A>B) = A_3'B_3+x_3A_2'B_2+x_3x_2A_1'B_1+x_3x_2x_1\,A_0'B_0$
- Implementation
  - $x_i = (A_iB_i'+A_i'B_i)'$

Four-Bit Magnitude Comparator



Decoder

- A *n*-to-*m* decoder
  - a binary code of *n* bits = 2^n distinct information
  - *n* input variables; up to 2^n output lines (8 K-maps ???)
  - only one output can be active (high) at any time

Table 4.6
Truth Table of a Three-to-Eight-Line Decoder

Inputs			Outputs							
x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Three-to-Eight-Line Decoder

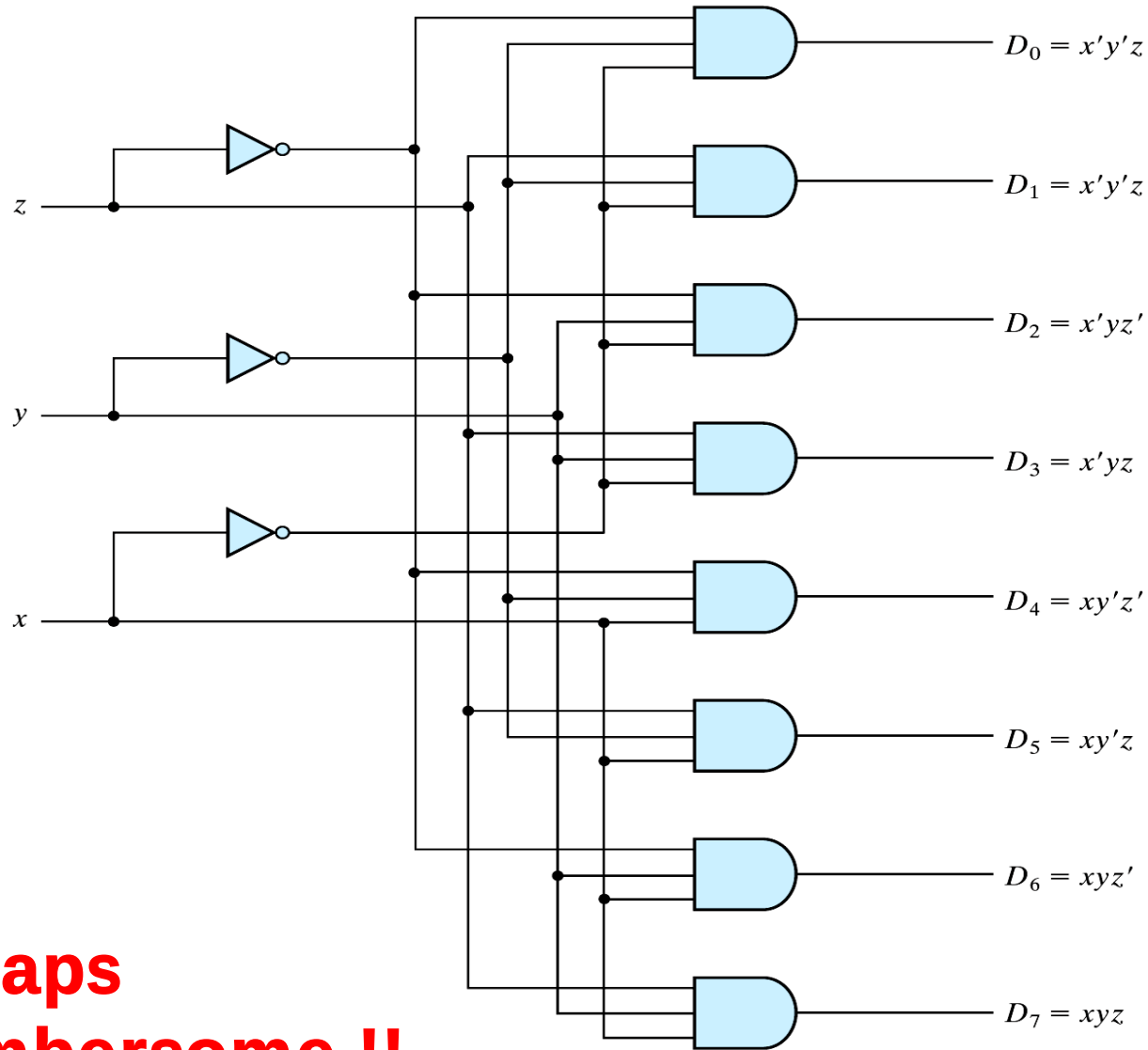
D_0

x \ yz	00	01	11	10
0	1			
1				

D_1

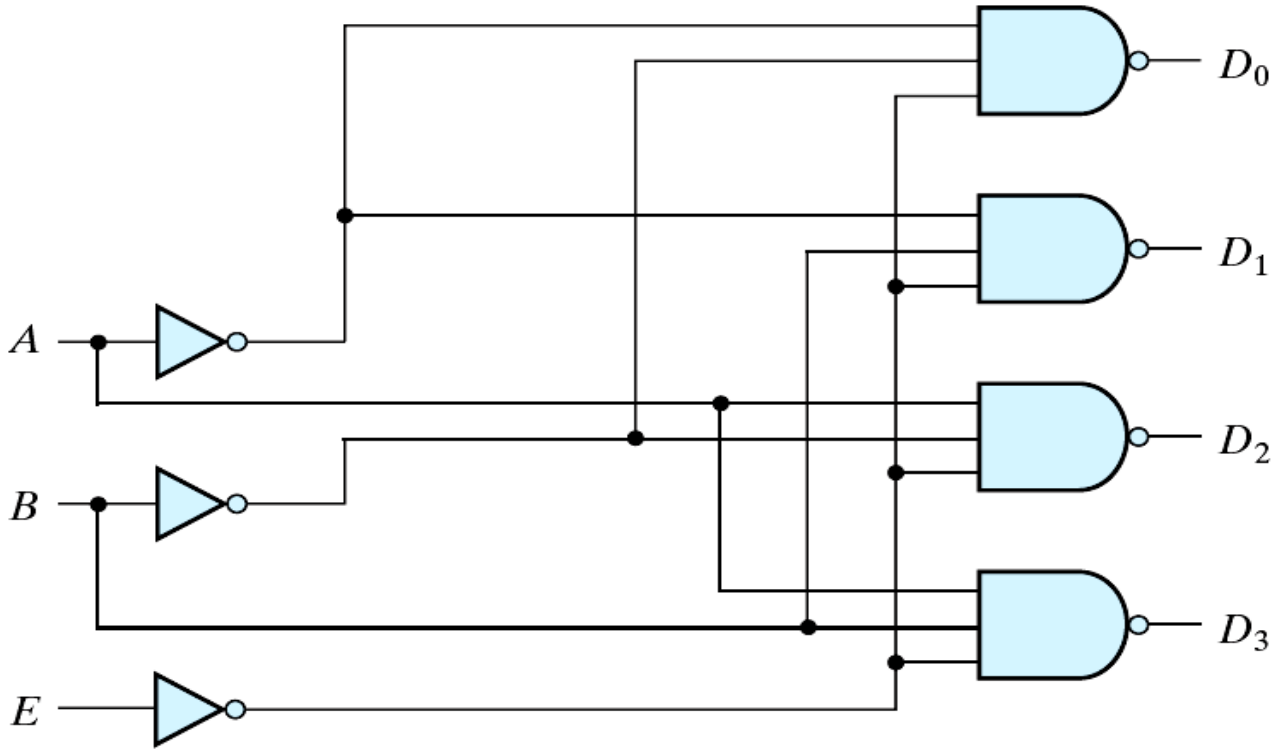
x \ yz	00	01	11	10
0		1		
1				

… D_7



8 K-maps
too cumbersome !!

Decoder with An Enable



(a) Logic diagram

The decoder is enabled when E is equal to 0 (active-low enable) is disabled when E is 1

(b) Truth table

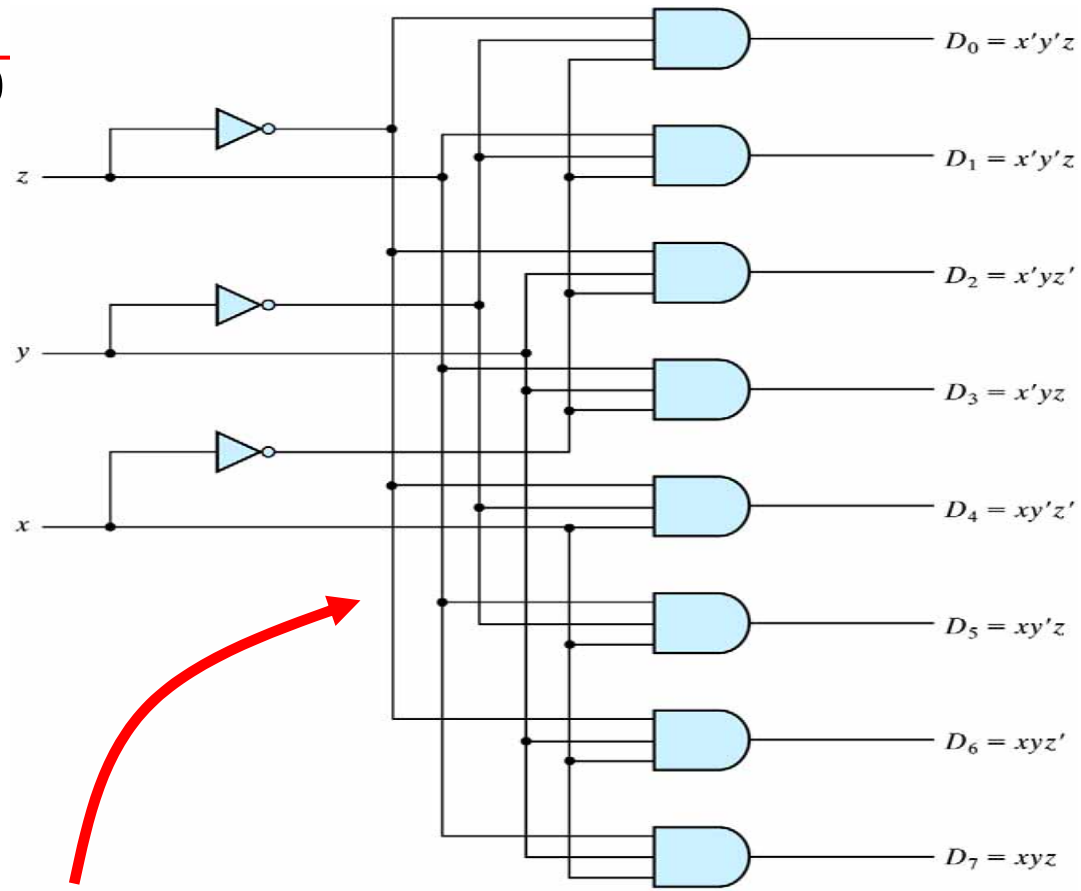
E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

Two-to-four-line decoder with enable input

4x16 Decoder

w	x	y	z	D_0	D_1	D_2	D_3	
0	0	0	0	1	0	0	0	……..0
0	0	0	1					
0	0	1	0					
0	0	1	1					
0	1	0	0					
0	1	0	1					
0	1	1	0					
0	1	1	1					
1	0	0	0					
1	0	0	1					
1	0	1	0					
1	0	1	1					
1	1	0	0					
1	1	0	1					
1	1	1	0					
1	1	1	1					

3X8



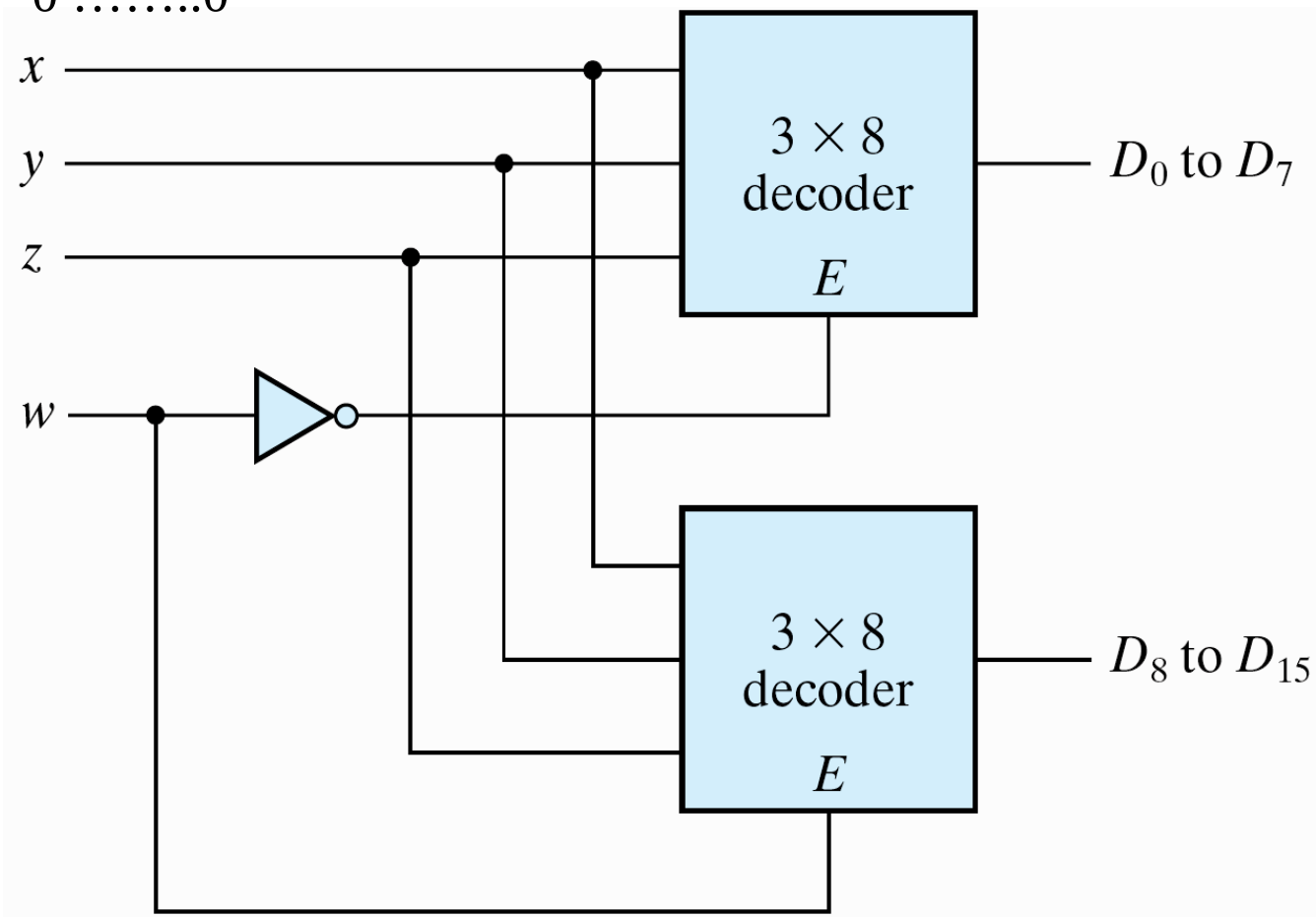
Extend to 4 inputs 16 outputs directly

$D_0 = w'x'y'z'$ $D_1 = w'x'y'z$ …….

4x16 Decoder: Two 3X8 decoders

w	x	y	z	D_0	D_1	D_2	D_3
0	0	0	0	1	0	0	0 ……..0
0	0	0	1				
0	0	1	0				
0	0	1	1				
0	1	0	0				
0	1	0	1				
0	1	1	0				
0	1	1	1				
1	0	0	0				
1	0	0	1				
1	0	1	0				
1	0	1	1				
1	1	0	0				
1	1	0	1				
1	1	1	0				
1	1	1	1				

Enable when w=0



Enable when w=1

Combination Logic Implementation

- each output of a decoder= a minterm
- use a decoder (“minterm generator”) and an external OR gate to implement any Boolean function of n input variables
- A full-adder

$S(x,y,x)=\Sigma(1,2,4,7)$

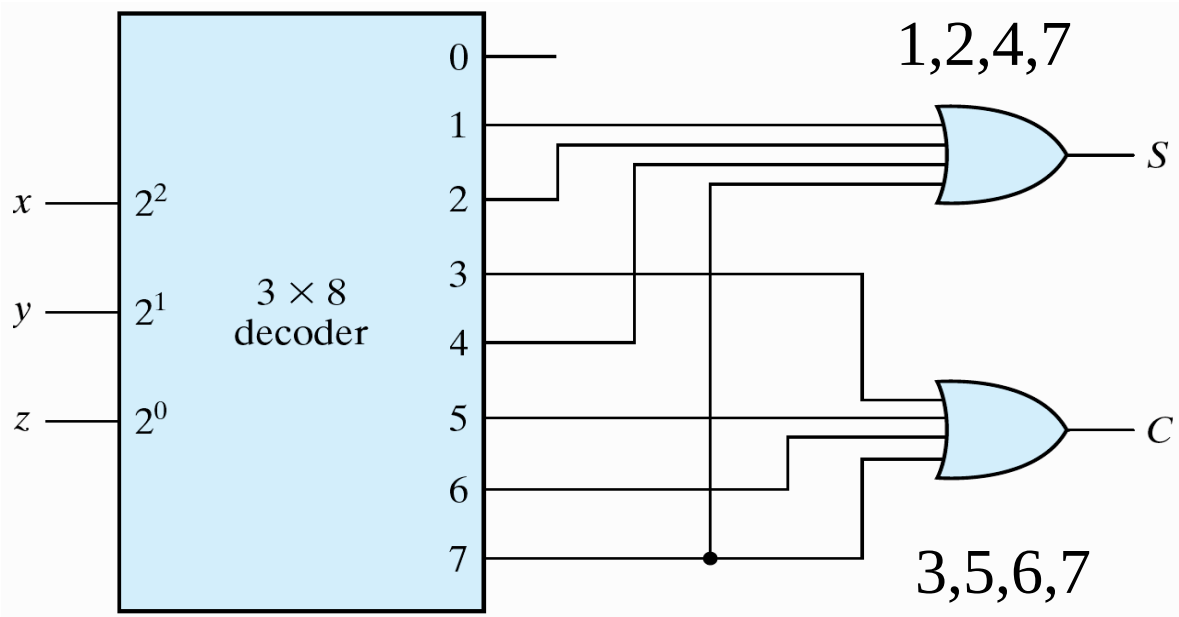
001, 010

100, 111, S=1

$C(x,y,z)=\Sigma(3,5,6,7)$

011, 101

110, 111, C=1



Encoders

The inverse operation of a decoder

only one input = 1 at any given time

Table 4.7
Truth Table of an Octal-to-Binary Encoder

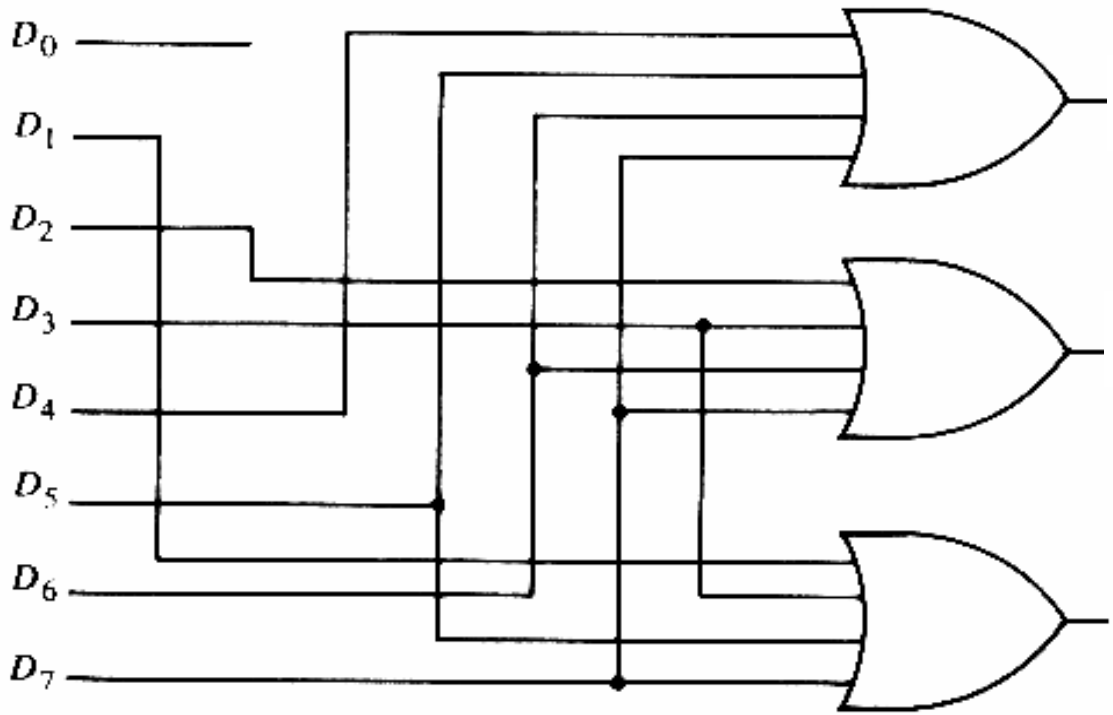
Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

$$z = D_1 + D_3 + D_5 + D_7$$
$$y = D_2 + D_3 + D_6 + D_7$$
$$x = D_4 + D_5 + D_6 + D_7$$

The encoder can be implemented with three OR gates.

Don't use K-map directly !!! Why?

Implementation of a 8-to-3 Encoder



$x = D_4 + D_5 + D_6 + D_7$

$y = D_2 + D_3 + D_6 + D_7$

$z = D_1 + D_3 + D_5 + D_7$

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

Priority Encoder

- resolve the ambiguity of illegal inputs (more inputs are equal to 1)
- only one of the input is encoded

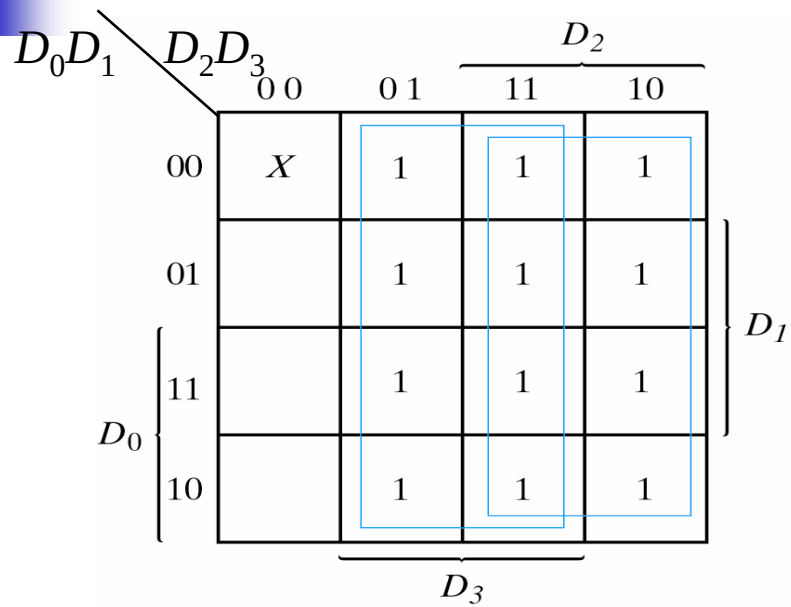
Table 4.8
Truth Table of a Priority Encoder

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

D_3 has the highest priority, D_0 has the lowest priority

X: don't-care conditions (0 or 1) , V: valid output indicator

Simplification of a Priority Encoder



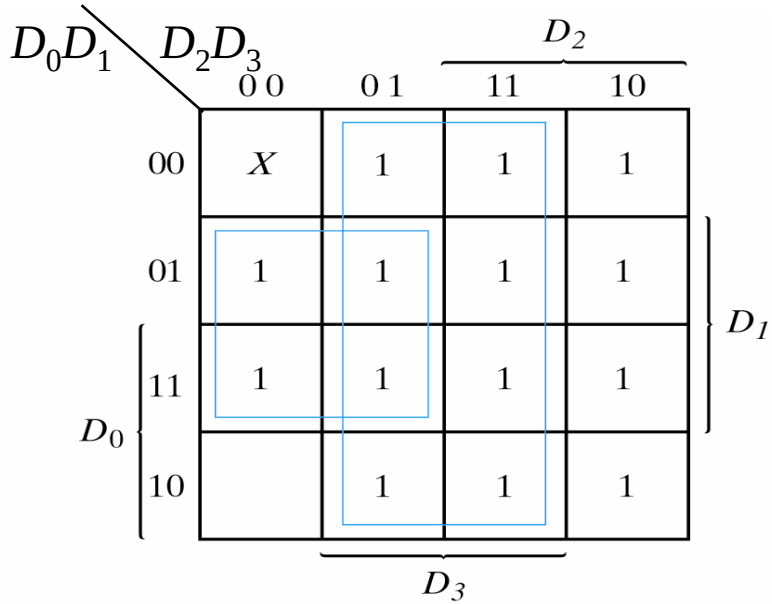
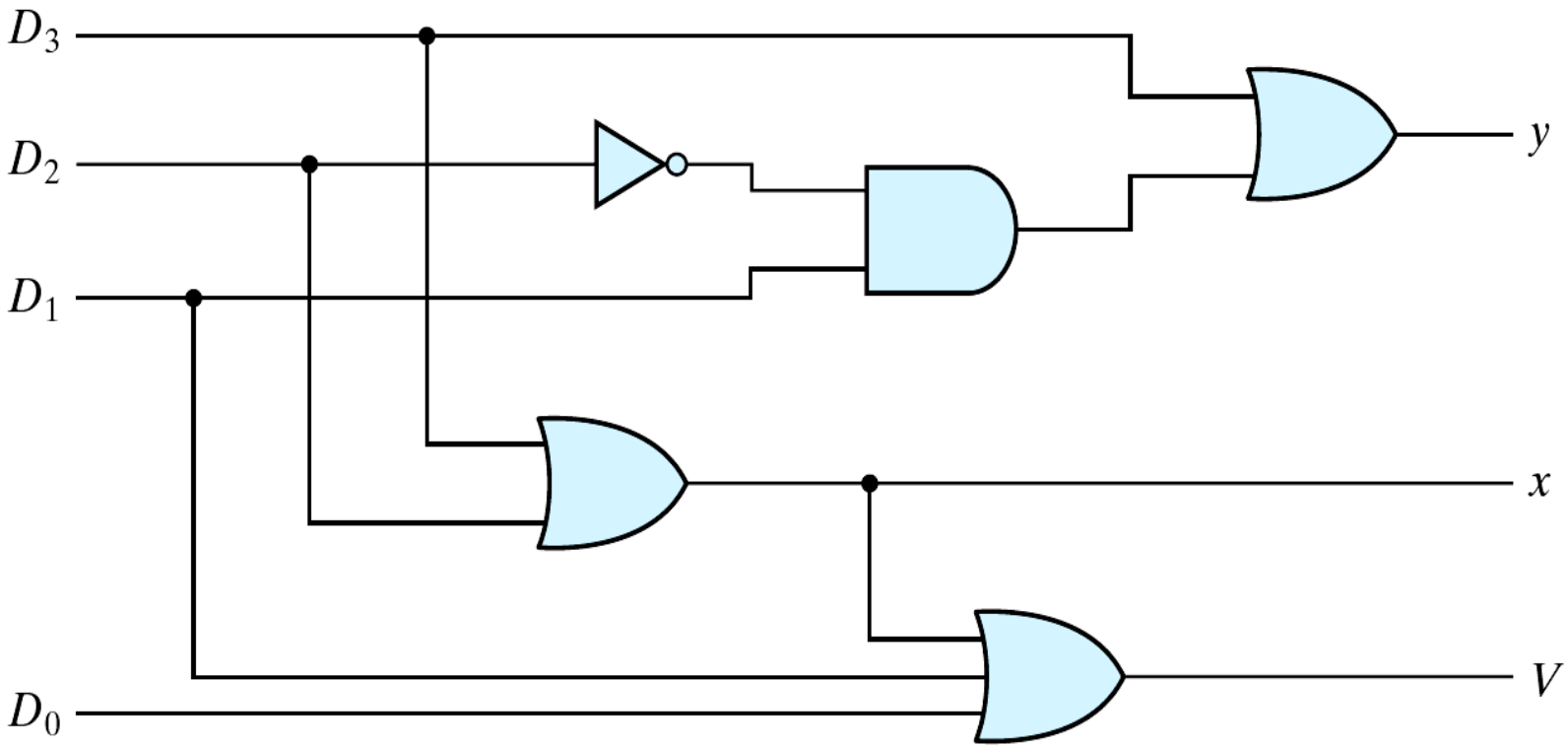


Fig. 4-22 Maps for a Priority Encoder

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

Implementation of a Priority Encoder

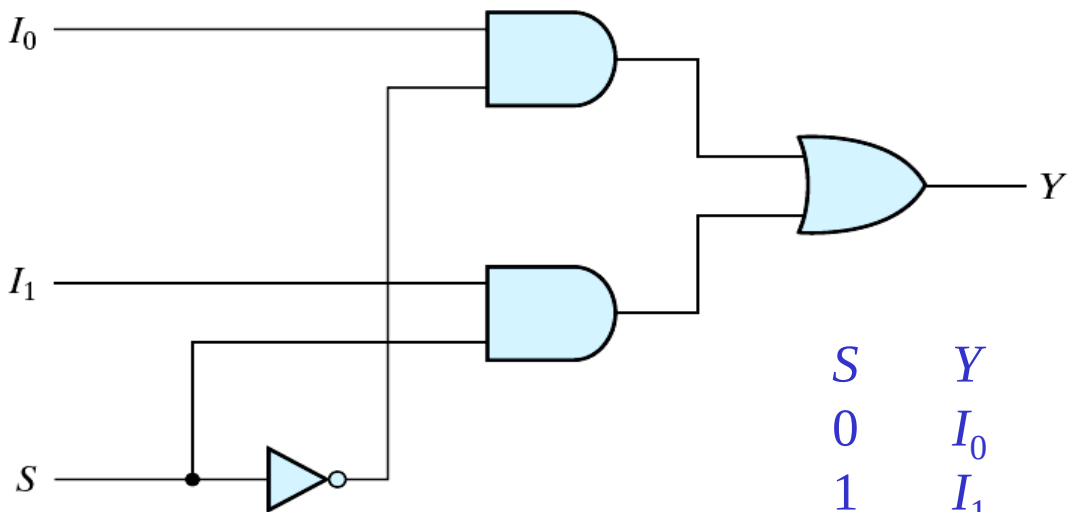
$$x = D_2 + D_3$$
$$y = D_3 + D_1 D_2'$$
$$V = D_0 + D_1 + D_2 + D_3$$



Multiplexers (Selectors)

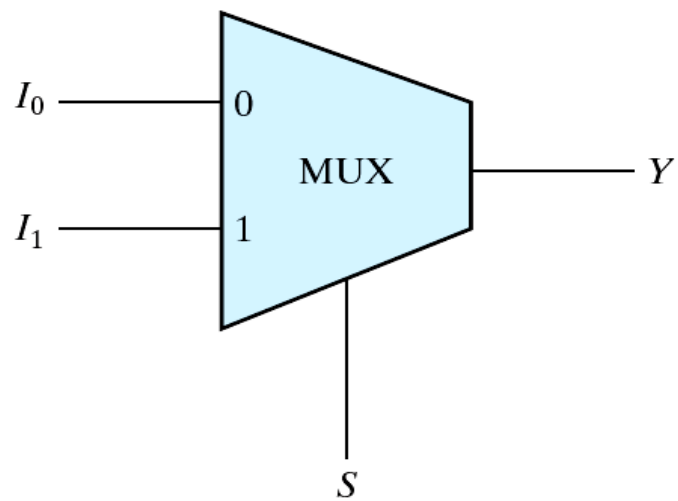
- select binary information from one of many input lines and direct it to a single output line
- 2^n input lines, n selection lines and one output line

2-to-1-line multiplexer



(a) Logic diagram

S	Y
0	I_0
1	I_1



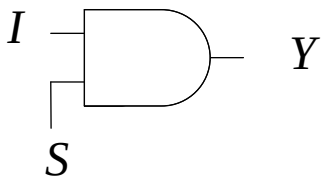
(b) Block diagram

(don't use K-map)

4-to-1-line multiplexer

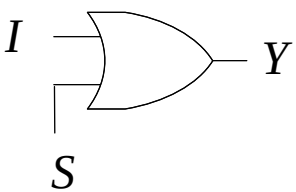
Controllability

AND gate

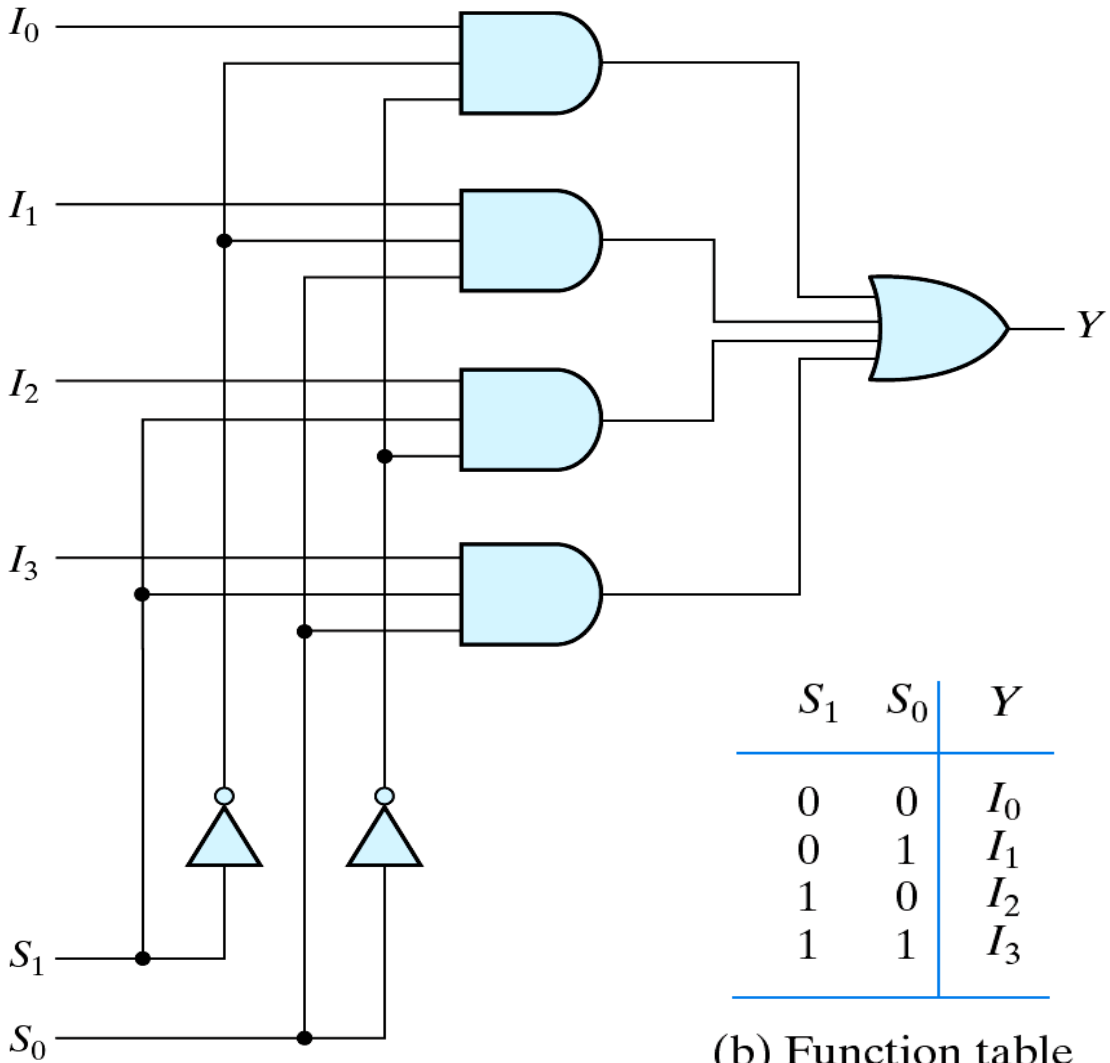


S=0, Y=0 (block input)
S=1, Y=I (pass input)

OR gate



S=1, Y=1 (block input)
S=0, Y=I (pass input)

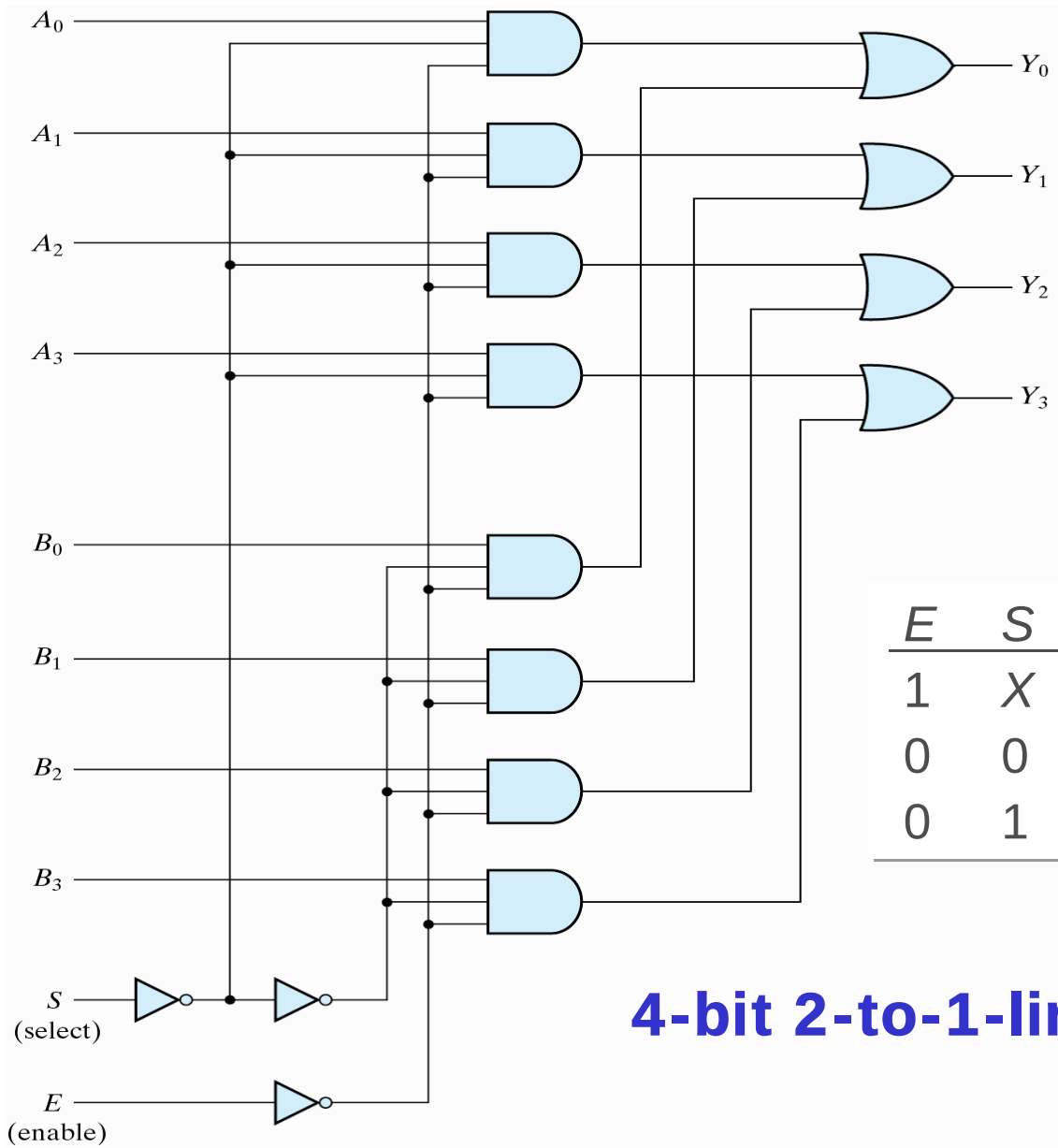


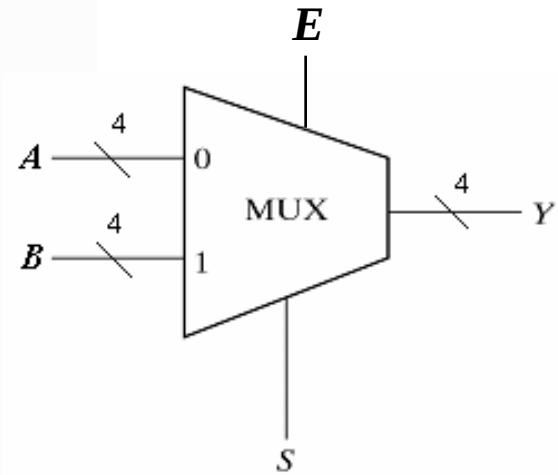
(a) Logic diagram

S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b) Function table

Quadruple Two-to-One-Line Multiplexer





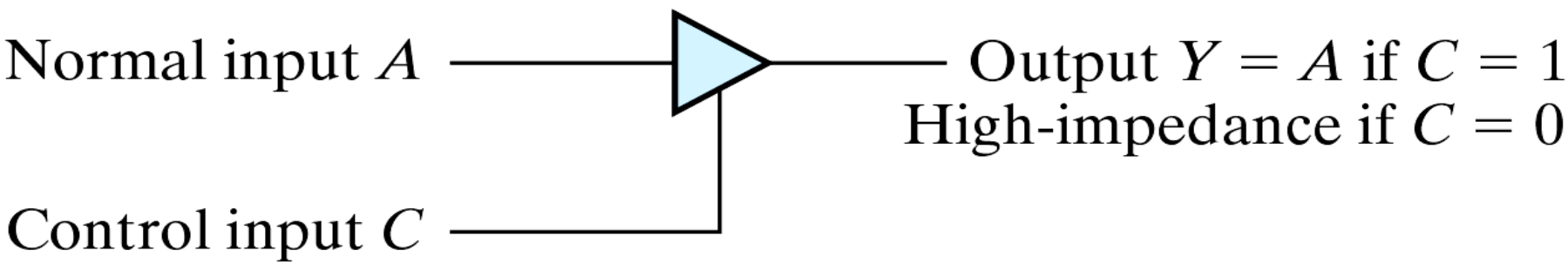
E	*S*	output *Y*
1	*X*	all 0's
0	0	select *A*
0	1	select *B*

4-bit 2-to-1-line multiplexer

Three-state gates

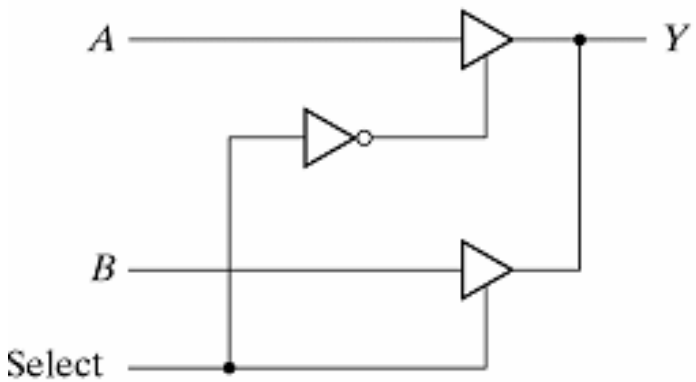
- Output states: 0, 1, and high-impedance
- High-impedance state behaves like an open circuit, which means that the output appears to be disconnected and the circuit has no logic significance.

A three-state buffer



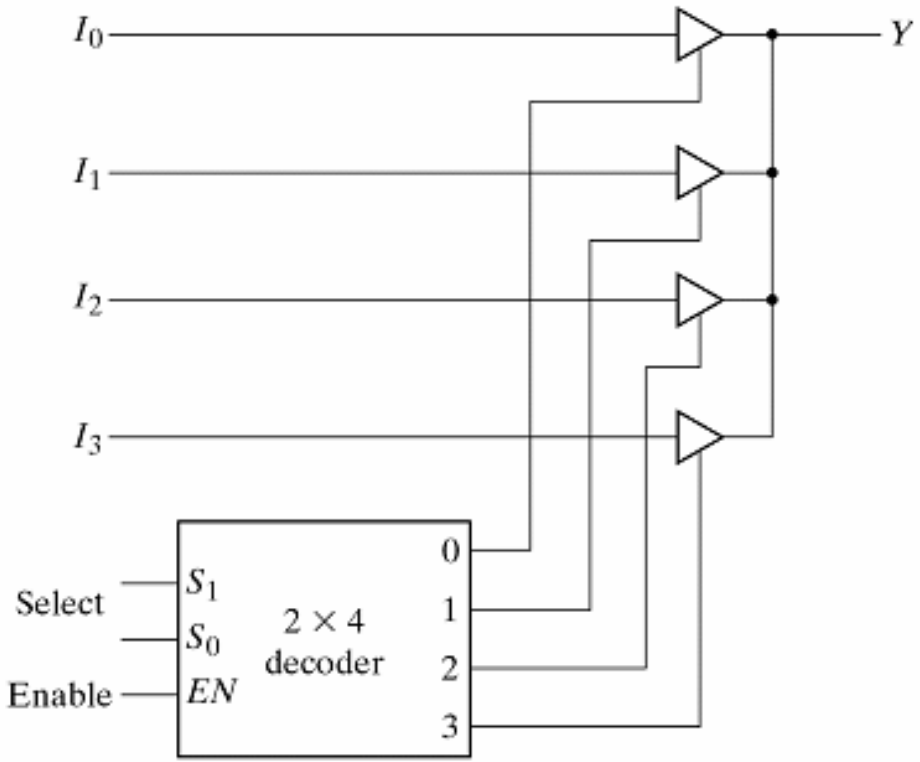
(Q: what is buffer ?)

Multiplexer with three-state gates



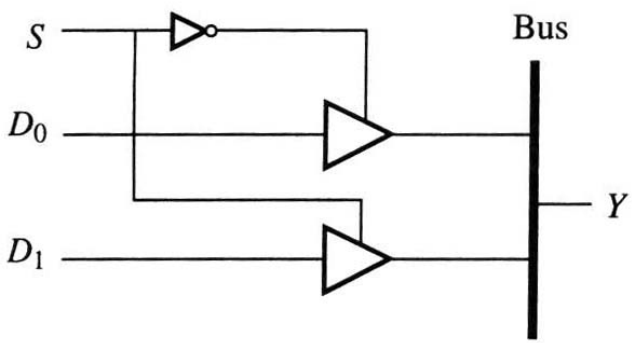
(a) 2-to-1- line mux

Fig. 4-30 Multiplexers with Three-State Gates



(b) 4 - to - 1 line mux

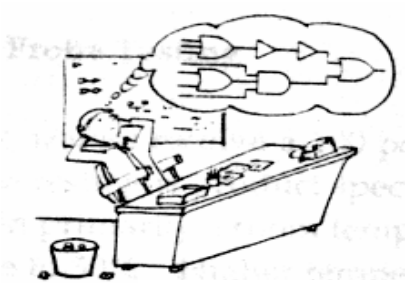
Fig. 4-30 Multiplexers with Three-State Gates



S	Y
0	D_0
1	D_1

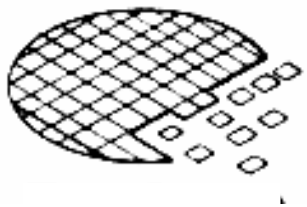
Bus need three-state gates

IC Industry in Taiwan

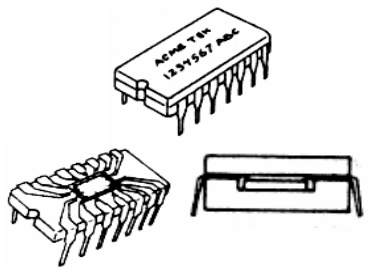


邏輯設計

光罩設計



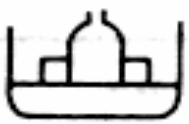
晶粒測試及切割



封　裝

成品測試

設計 → 光罩 → 製造 → 封裝 → 測試

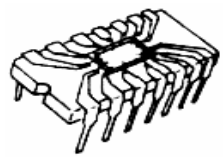


長晶

晶圓切割

晶圓

化學品



導線架

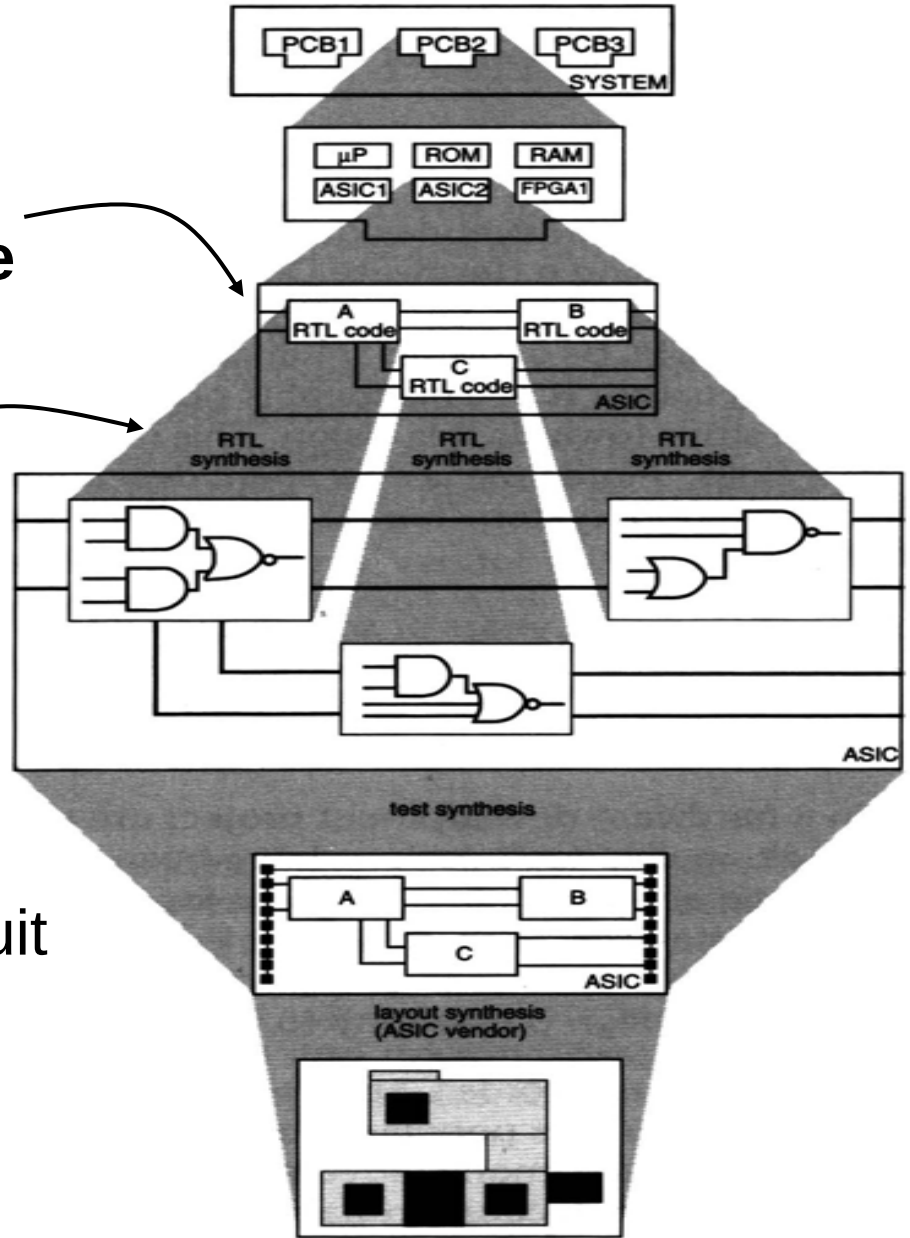
Circuits are implemented on IC (integrated circuit)

IC➔ Circuits➔ gates➔ transistors

Hierarchical Components in PCB

1. Describe the circuits with Hardware Description Language (HDL硬體描述語言)

2. Synthesis (合成) the circuits …. by using EDA tools



application specific integrated circuit (ASIC晶片)

IC or chip

Semi Custom IC Design (半客戶式IC設計)

Semi Custom ICDesign

a. Product specification

b. Modeling with HDL

c. Synthesis (by using suitable standard cell)

d. Simulation and verification

e. Physical placement and layout

-- implemented with suitable tools

f. Tape-out (real chip) -- implemented by suitable Fab companies

g. Testing -- implemented by suitable tools and mechanisms

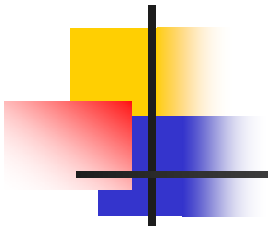
more flexible, shorter design cycle, suitable for smaller production

Two different solutions :

PLD → FPGA or CPLD — Xilinx, Altera

Standard cell → Real ASIC chip — Fab (TSMC, UMC, ..)

less flexible, long design cycle, larger-scale production to reduce price



Hardware Description Language (HDL)

- Hardware description language allows you to describe circuit at different levels of abstractions, and allows you to mix any level of abstraction in the design
- Two of the most popular HDLs

  -- Verilog -- VHDL
- HDLs can be used for both the cell-based synthesis and FPGA/CPLD implementation
- Only Verilog is introduced here

Why Verilog?

Verilog History

1. Verilog was written by gateway design automation in the early 1980
2. Cadence acquired gateway in 1990
3. Cadence released Verilog to the pubic domain in 1991
4. In 1995, the language was ratified as IEEE standard 1364

Why Verilog ?

1. Choice of many design teams
2. Most of us are familiar with C- like syntax/semantics

Verilog Features

Features:

- Procedural constructs for conditional, if-else, case and looping operations
- Arithmetic, logical, bit-wise, and reduction operations for expression
- Timing control

Basics of Verilog Language:

- Verilog Module - Identifier - Keyword
- Four Value Logic - Data Types - Numbers
- Port Mapping - Operator - Comments

Verilog Module (1/2)

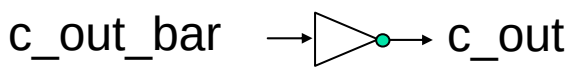
module module_name (port_name);

(1) port declaration

(2) data type declaration

(3) module functionality or structure

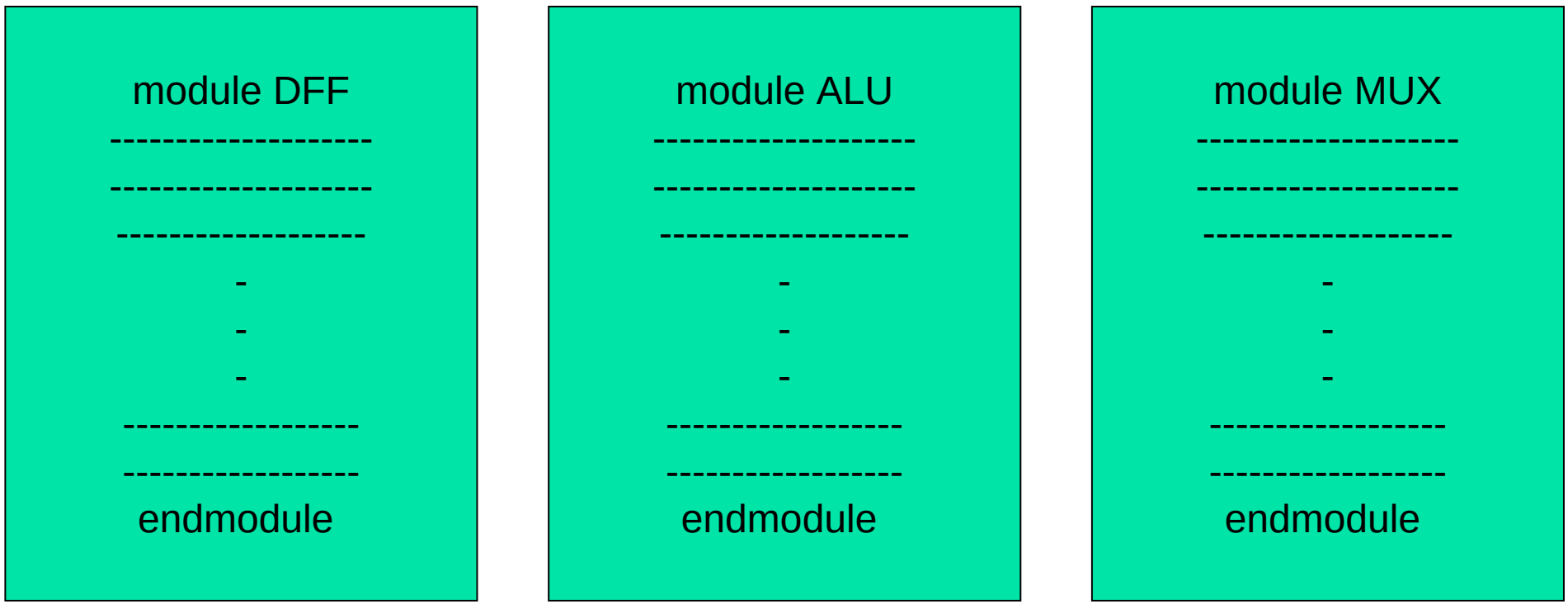
endmodule




```
module Add_half(sum, c_out, a, b);

(1)  input          a, b;
     output   sum, c_out;

(2)  wire     c_out_bar;

(3)  xor            (sum, a, b);
     nand     (c_out_bar, a, b);
     not     (c_out, c_out_bar);

endmodule
```

Verilog Module (2/2)

Verilog Module: basic building block



Structural Description

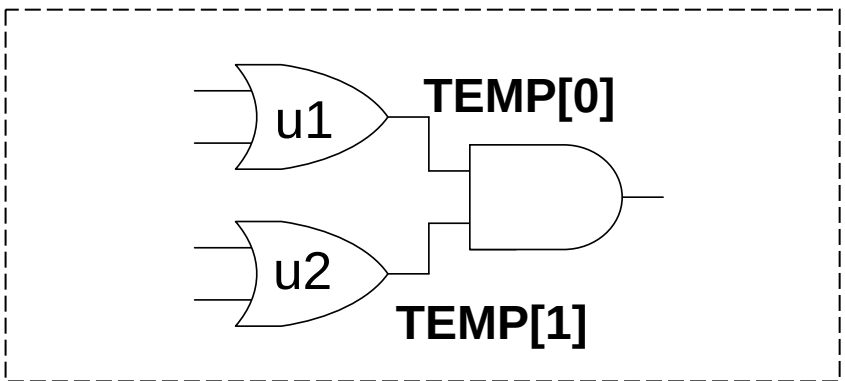
Verilog allows three kinds of descriptions for circuits:

(1) Structural description (2) Data flow description

(3) Behavioral description

Structural description:

```
1. module OR_AND_STRUCTURAL(IN,OUT);

2. input    [3:0]   IN;
3. output           OUT;
4. wire     [1:0]   TEMP;

5. or u1(TEMP[0], IN[0], IN[1]);
6. or u2(TEMP[1], IN[2], IN[3]);
7. and (OUT, TEMP[0], TEMP[1]);
8. endmodule
```



Synthesized **(synthesis) +**

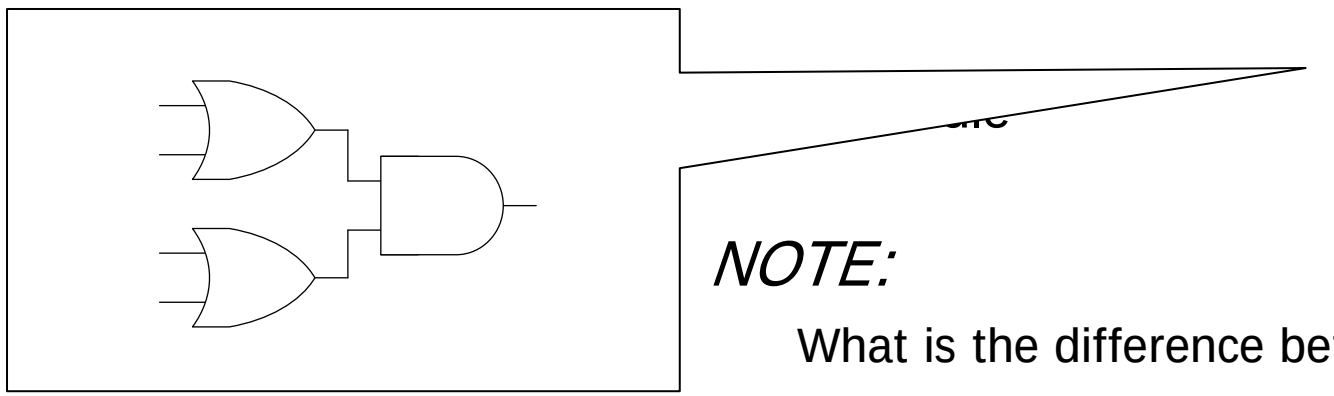
optimized by tools

Data Flow Description

Data flow description

1. module OR_AND_DATA_FLOW(IN, OUT);
2. input [3:0] IN;
3. output OUT;
4. assign OUT = (IN[0] | IN[1]) & (IN[2] | IN[3]);

Synthesized and optimized by tools



NOTE:

What is the difference between *C* and *Verilog* ?

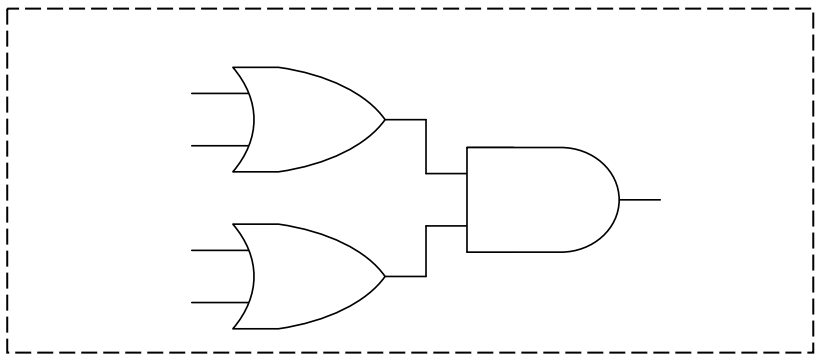
C : only one iteration (once) is implemented for assignment

Verilog : hard-wired circuit for assignment

Behavioral (RTL) Description (1/2)

Behavioral description #1

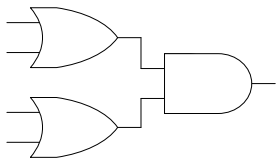
```
1.  module OR_AND_BEHAVIORAL(IN, OUT);

2.  input  [3:0]  IN;
3.  output        OUT;
4.  reg           OUT;

5.  always @(IN)
6.  begin
7.     OUT = (IN[0] | IN[1]) & (IN[2] | IN[3]);
8.  end
9.  endmodule
```



Activate OUT while any voltage transition

(0→1 or 1→0) happens at signal IN

Behavioral (RTL) Description (2/2)



Truth Table

IN[0]	IN[1]	IN[2]	IN[3]	OUT
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Behavioral description #2

```
module or_and(IN, OUT);
input [3:0] IN;   output   OUT; reg  OUT;   (Note)

always @(IN)
  begin
    case(IN)
      4'b0000: OUT = 0; 4'b0001: OUT = 0;
      4'b0010: OUT = 0; 4'b0011: OUT = 0;
      4'b0100: OUT = 0; 4'b0101: OUT = 1;
      4'b0110: OUT = 1; 4'b0111: OUT = 1;
      4'b1000: OUT = 0; 4'b1001: OUT = 1;
      4'b1010: OUT = 1; 4'b1011: OUT = 1;
      4'b1100: OUT = 0; 4'b1101: OUT = 1;
      4'b1110: OUT = 1; default: OUT = 1;
    endcase
  end
endmodule
```

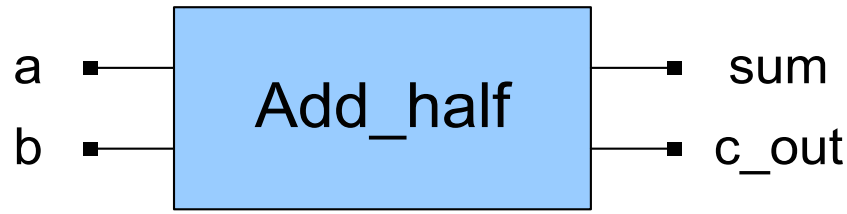
IN[0]
IN[1]
IN[2]
IN[3]
OU

Synthesized and optimized by tools

Half Adder (1/5)

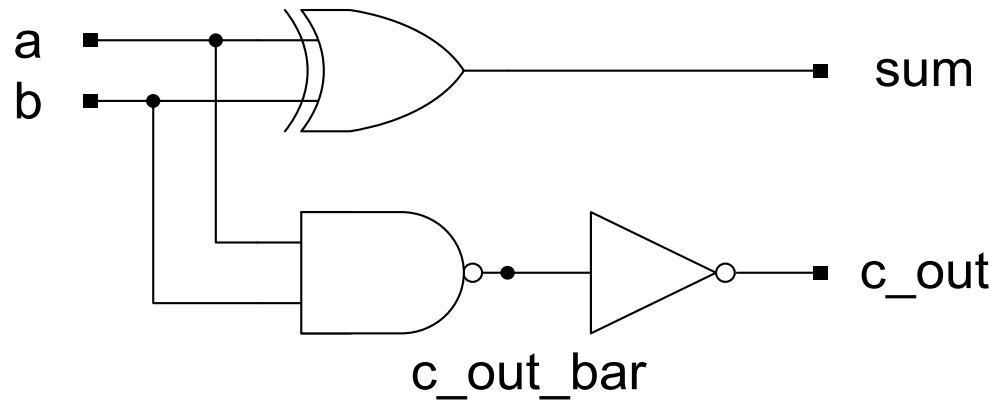
a\b	0	1
0	0	1
1	1	0

$sum = a \oplus b$

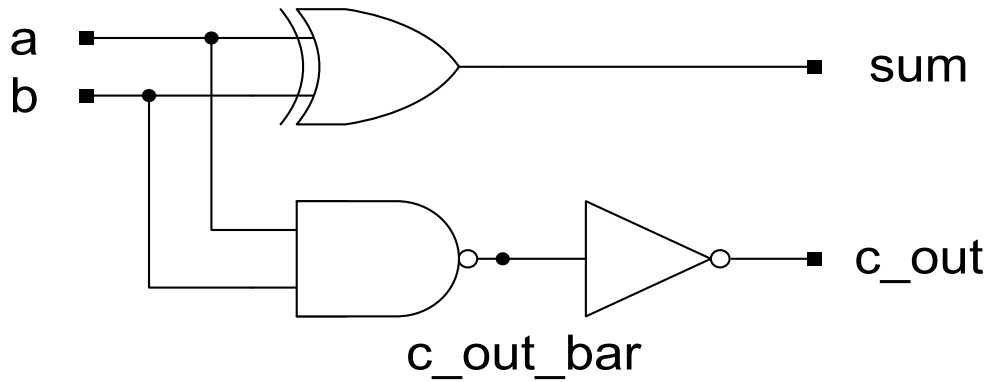


a\b	0	1
0	0	0
1	0	1

$c\_out = ab$



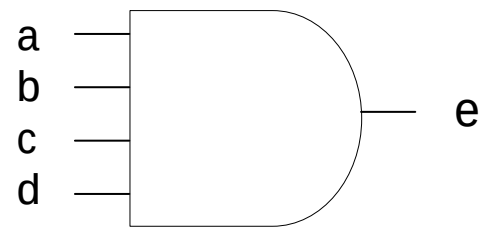
Half Adder (2/5)



Structural description

```
module Add_half(sum, c_out, a, b);
   input       a, b;
   output      sum, c_out;
   wire        c_out_bar;

   xor         (sum, a, b);
   nand        (c_out_bar, a, b);
   not         (c_out, c_out_bar);
endmodule
```

```
and (e, a, b,c,d);
```

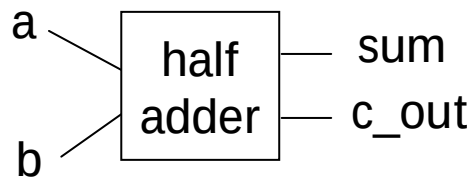


Half Adder (3/5)

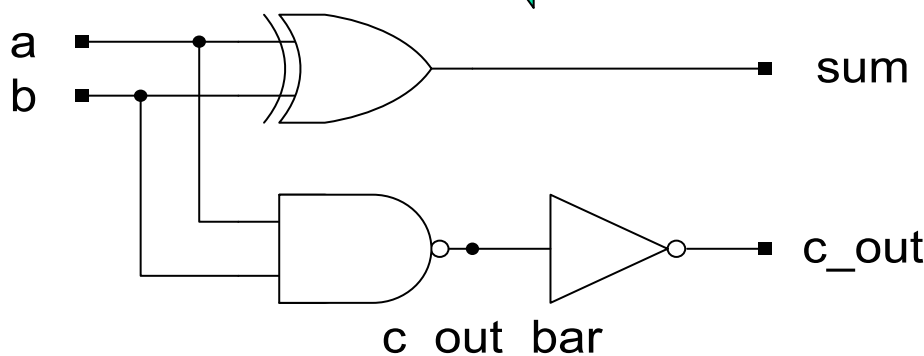
Data flow description

```
module Add_half(sum, c_out, a, b);
   input      a, b;
   output     sum, c_out;

   assign     {c_out, sum} = a + b;
endmodule
```

assign: continuous assignment



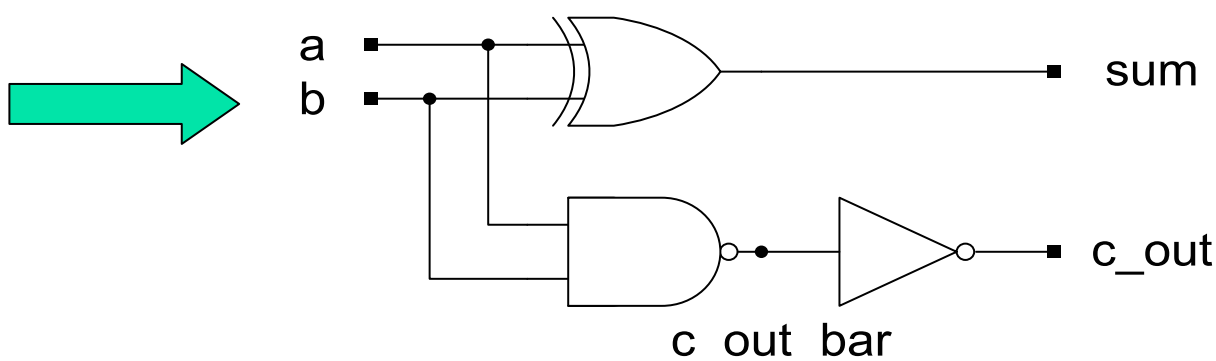
Synthesized and optimized by tools



Half Adder (4/5)

Behavioral description #1

```
module Add_half(sum, c_out, a, b);
   input      a, b;
   output     sum, c_out;
   reg        sum, c_out;

   always @ (a or b)
     begin
       sum = a ^ b;
       c_out = a & b;
     end
endmodule
```



Half Adder (5/5)

Behavioral description #2

```
module Add_half(sum, c_out, a, b);
input  a, b;
output sum, c_out;
reg    sum, c_out;
always @(a or b)
begin
    case({a,b})
      2'b00:begin
          sum = 0;c_out = 0;
          end
      2'b01:begin
          sum = 1; c_out = 0;
          end
      2'b10:begin
          sum = 1;  c_out = 0;
          end
      default:begin
          sum = 0; c_out = 1;
          end
    endcase
end
endmodule
```

a\b	0	1
0	0	1
1	1	0

sum

a\b	0	1
0	0	0
1	0	1

c_out