Chapter 4 Designing Combinational Systems

- Full adder
- Ripple carry adders
- Carry look-ahead adders
- Decoder
- Multiplexer
- Buffer
- Bus
- Gate arrays
  - ROM
  - PLA
  - PAL

Full Adder

x_i	y_i	c_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 for full adder

c_i \ x_iy_i	00	01	11	10
0		**1**		**1**
1	**1**		**1**	

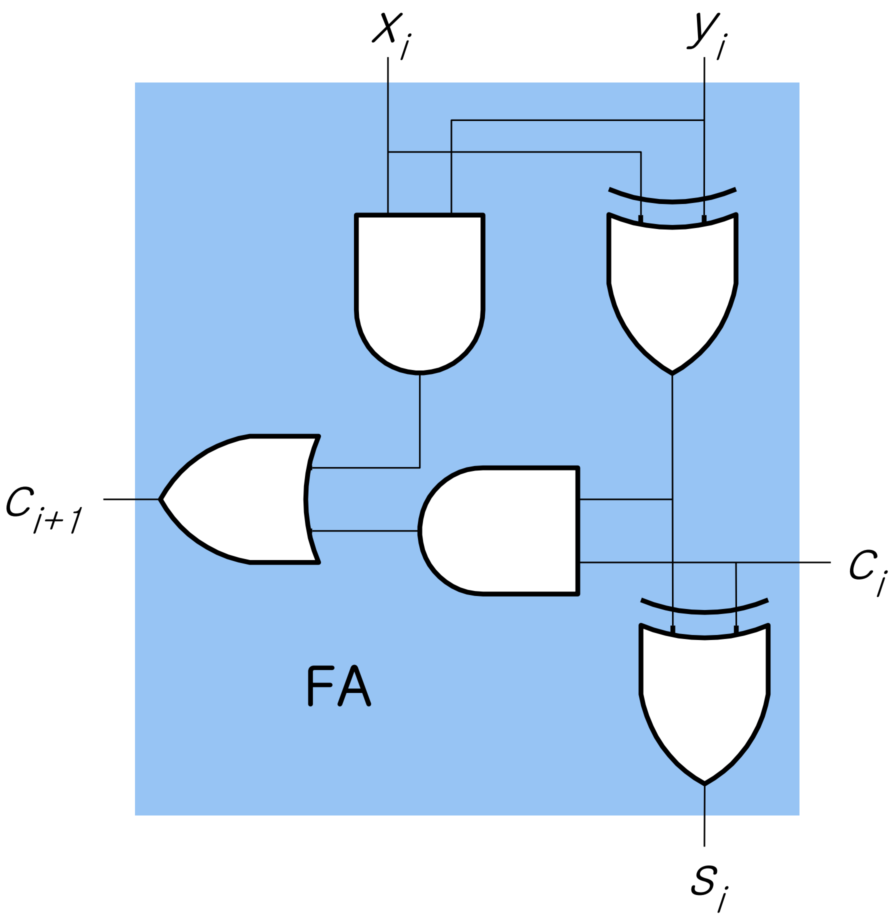
$$S_i = x_i \oplus y_i \oplus c_i$$

c_i \ x_iy_i	00	01	11	10
0			**1**	
1		**1**	**1**	**1**

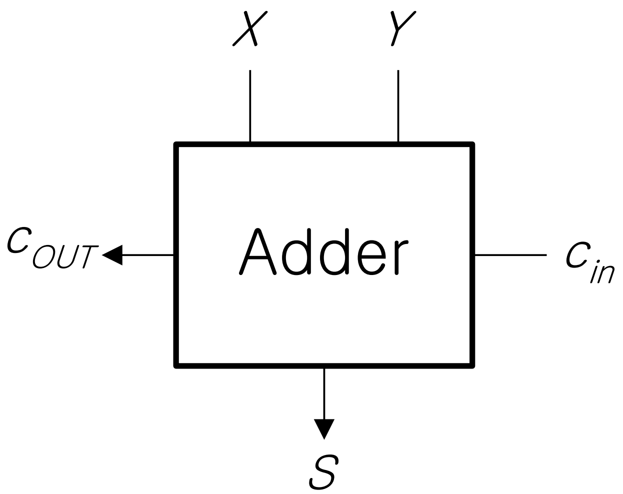
$$c_{i+1} = x_iy_i + c_i(x_i \oplus y_i)$$

(b) Map representation

Full Adder



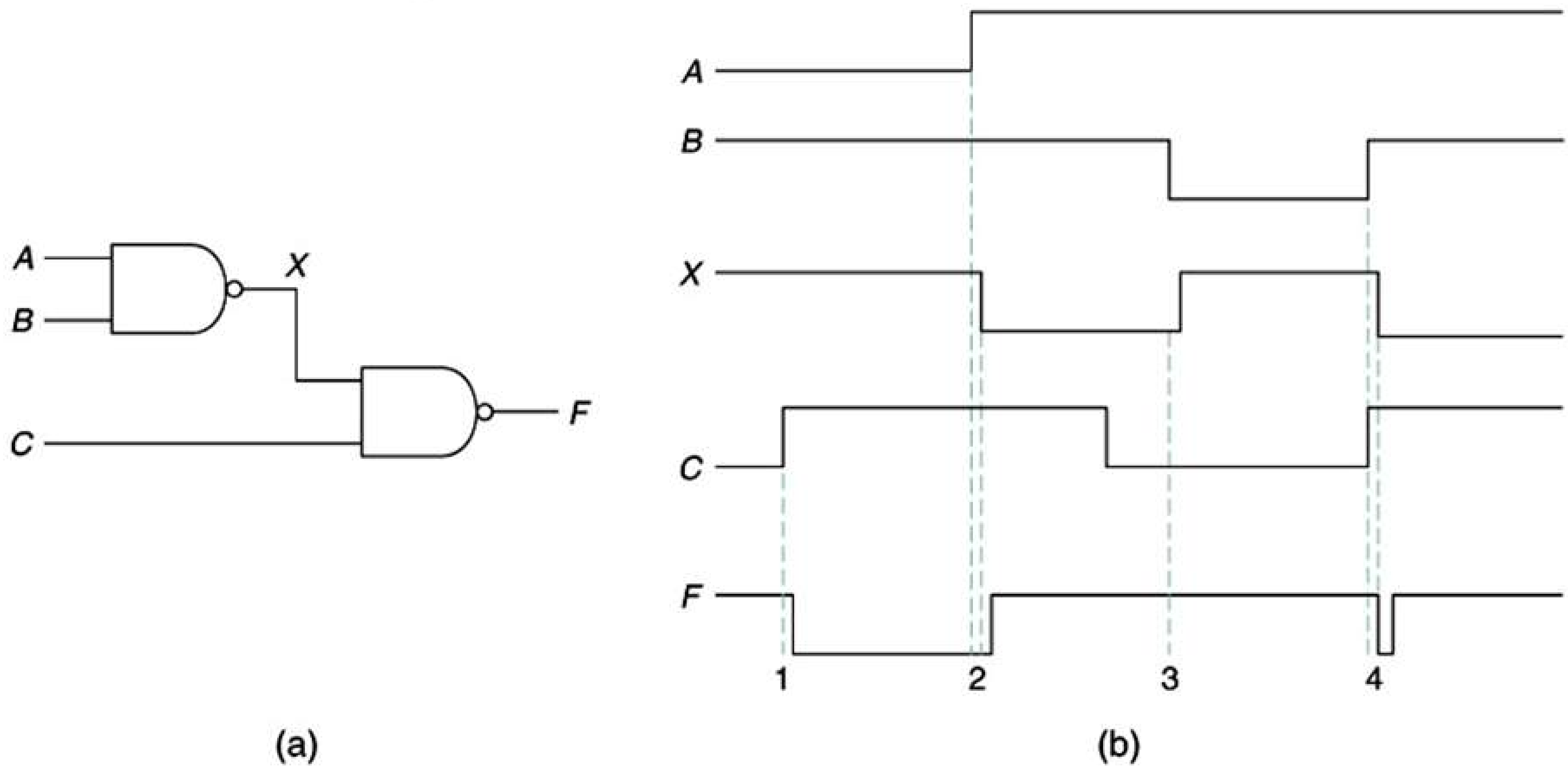
(c) Full-adder logic schematic



(e) Graphic symbol

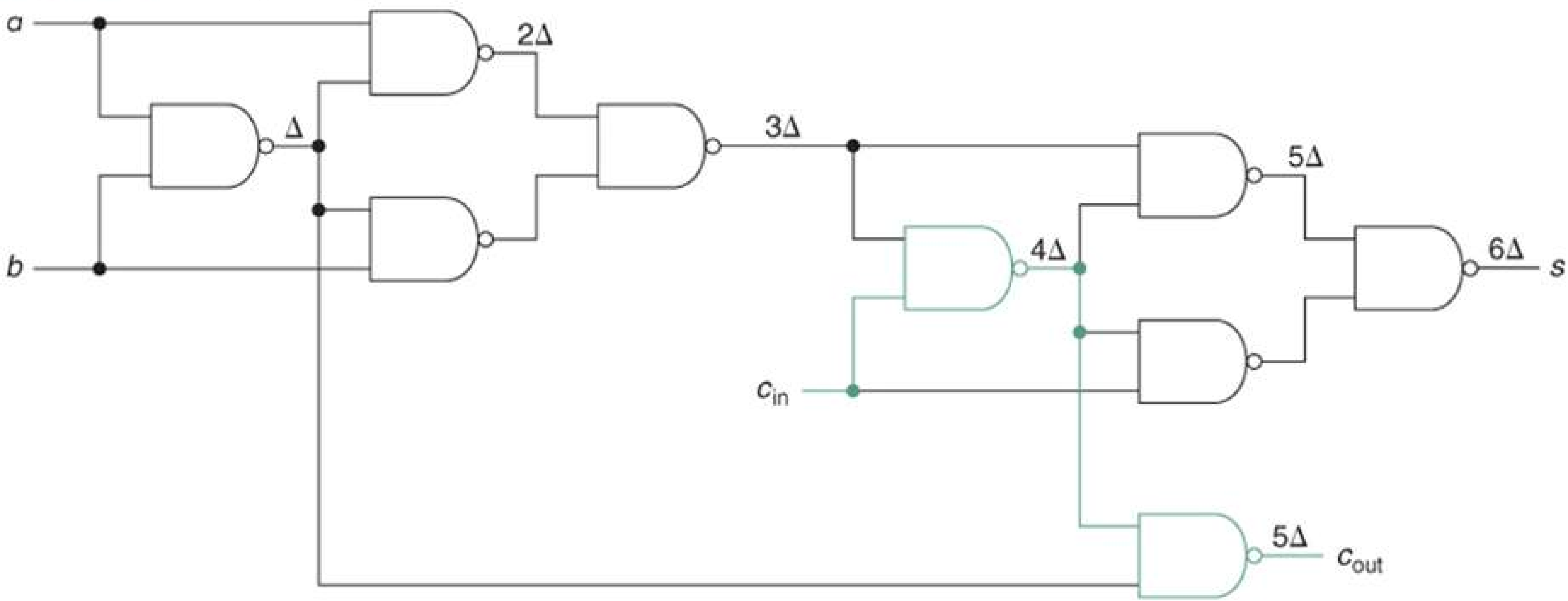
Gate delay

Figure 4.2 Illustration of gate delay.



Delay through 1-bit adder (full adder)

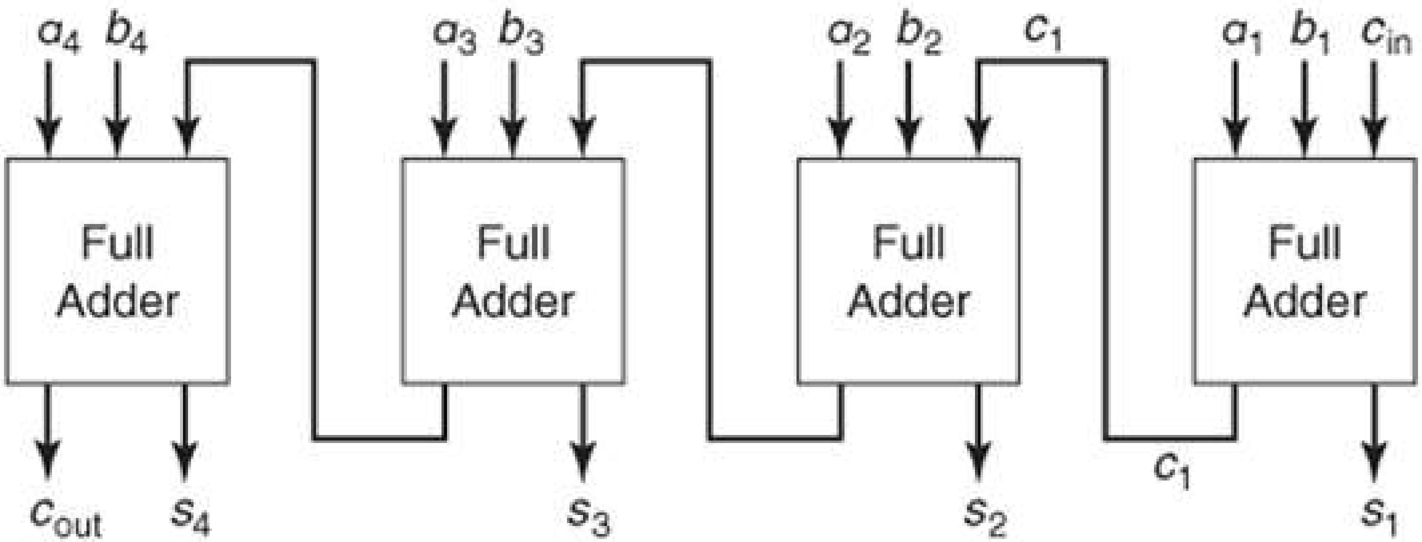
Figure 4.3 Delay through a 1-bit adder.



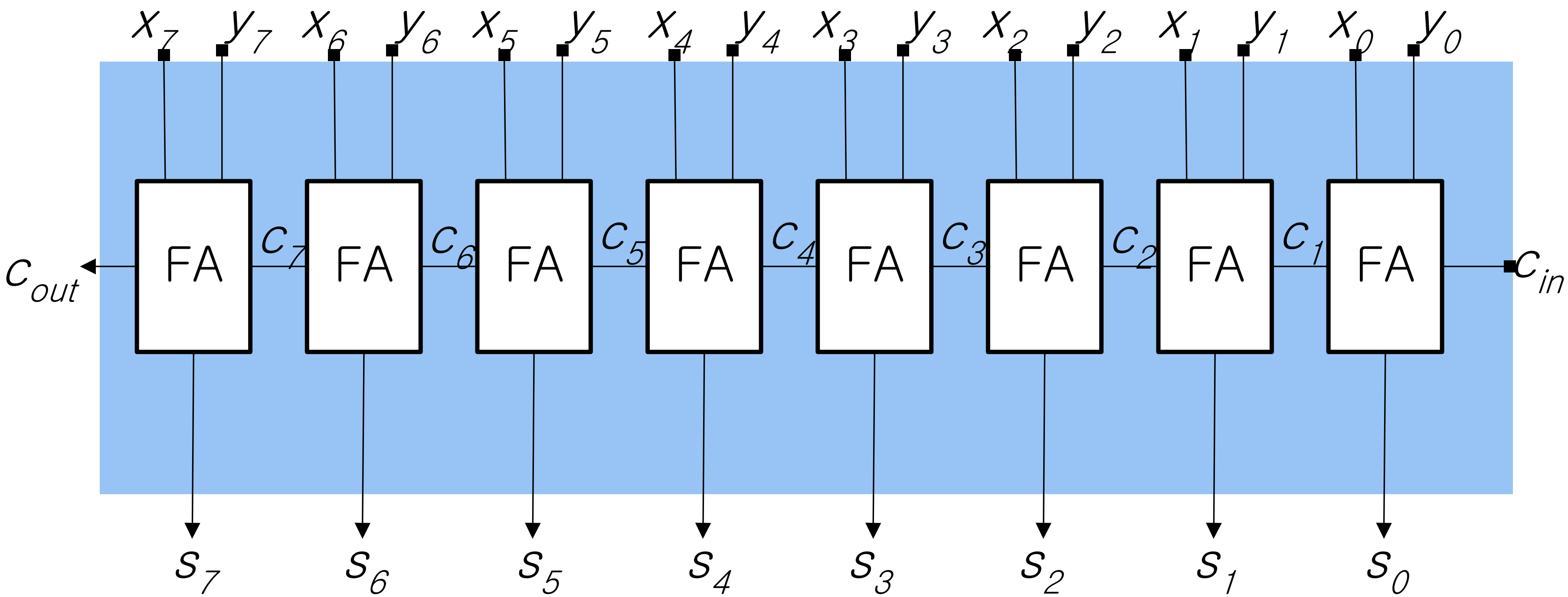
N-bit adder can be built by connecting n 1-bit adders (ripple carry adder)
Time for n-bit adder = (2n+4)Δ

4-bit ripple carry adder

Figure 4.1 A 4-bit adder.



8-bit Ripple carry adder



Carry look-ahead adders

- **Advantage of Carry-look-ahead technique**
  - reduce the delay of the carry chain in this kind of ripple-carry chain

- $\boldsymbol{x_{i+3}\,x_{i+2}\,x_{i+1}\,x_i + y_{i+3}\,y_{i+2}\,y_{i+1}\,y_i}$ **(input carry $\boldsymbol{c_i}$)**
  - carry-generate function $\boldsymbol{g_i = x_i y_i}$
  - carry-propagate function $\boldsymbol{p_i = x_i \oplus y_i}$
  - $c_{i+1} = g_i + p_i\,c_i$ $\quad$ $c_{i+2} = g_{i+1} + p_{i+1}\,c_{i+1}$

    $c_{i+3} = g_{i+2} + p_{i+2}\,c_{i+2}$ $\quad$ $c_{i+4} = g_{i+3} + p_{i+3}\,c_{i+3}$

- **Express in terms of $\boldsymbol{c_i}$**
  - $c_{i+1} = g_i + p_i c_i$

    $c_{i+2} = g_{i+1} + p_{i+1} g_i + p_{i+1} p_i c_i$

    $c_{i+3} = g_{i+2} + p_{i+2} g_{i+1} + p_{i+2} p_{i+1} g_i + p_{i+2} p_{i+1} p_i c_i$

    $c_{i+4} = g_{i+3} + p_{i+3} g_{i+2} + p_{i+3} p_{i+2} g_{i+1} + p_{i+3} p_{i+2} p_{i+1} g_i + p_{i+3} p_{i+2} p_{i+1} p_i c_i$

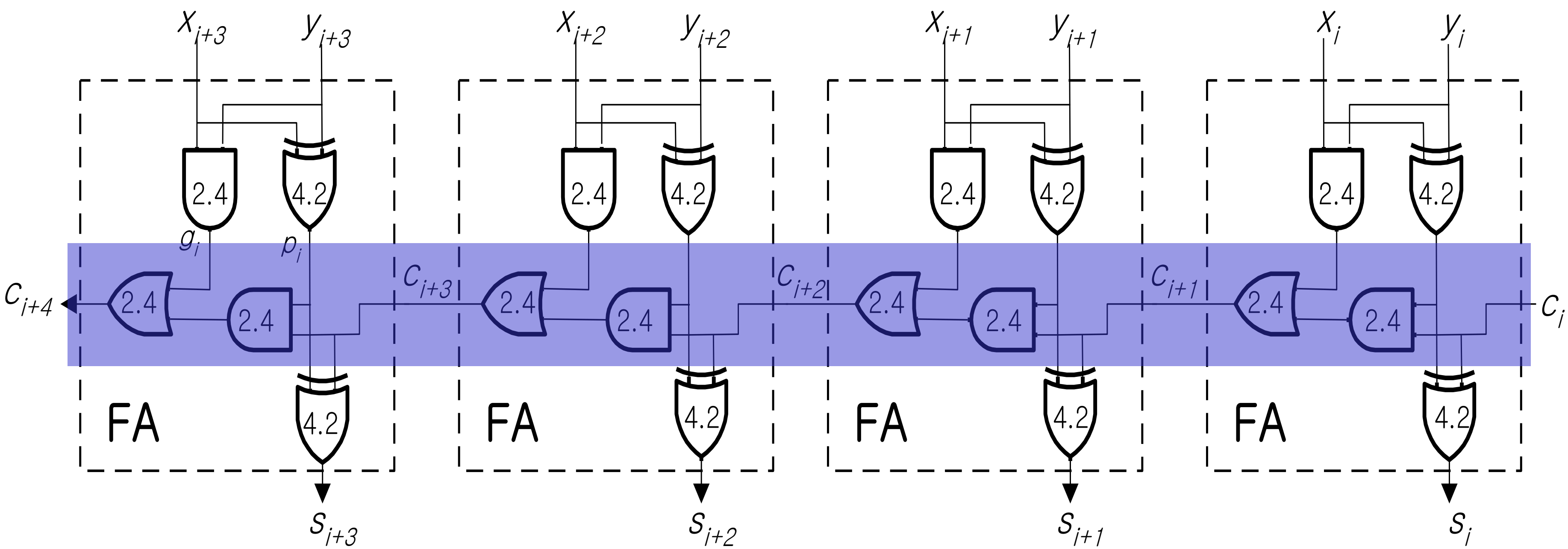
    $\Rightarrow$ can compute directly from input bits and input carry c_i

Carry look-ahead adders

- **carry-look-ahead(CLA) generator**
  - $c_{i+4} = g_{(i,i+3)}+p_{(i,i+3)}c_i$
    - $g_{(i,i+3)} = g_{i+3}+p_{i+3}g_{i+2}+p_{i+3}p_{i+2}g_{i+1}+p_{i+3}p_{i+2}p_{i+1}g_i$
    - $p_{(i,i+3)} = p_{i+3}p_{i+2}p_{i+1}p_i$
  - replace the 4-bit carry chain, so make the adder faster

- **carry-look-ahead adder v.s. ripple-carry-adder**
  - ripple-carry adder
    - consist of 4 FA carry chains
    - each FA carry chain consists of 1 AND and 1 OR gate, used to compute $c_{i+1} = g_i+p_ic_i$
  - carry-look-ahead adder
    - CLA generator can produce 4 carries with less delay than the ripple-carry chain
    - produce a faster 4-bit adder, and improve the speed of larger adders as well

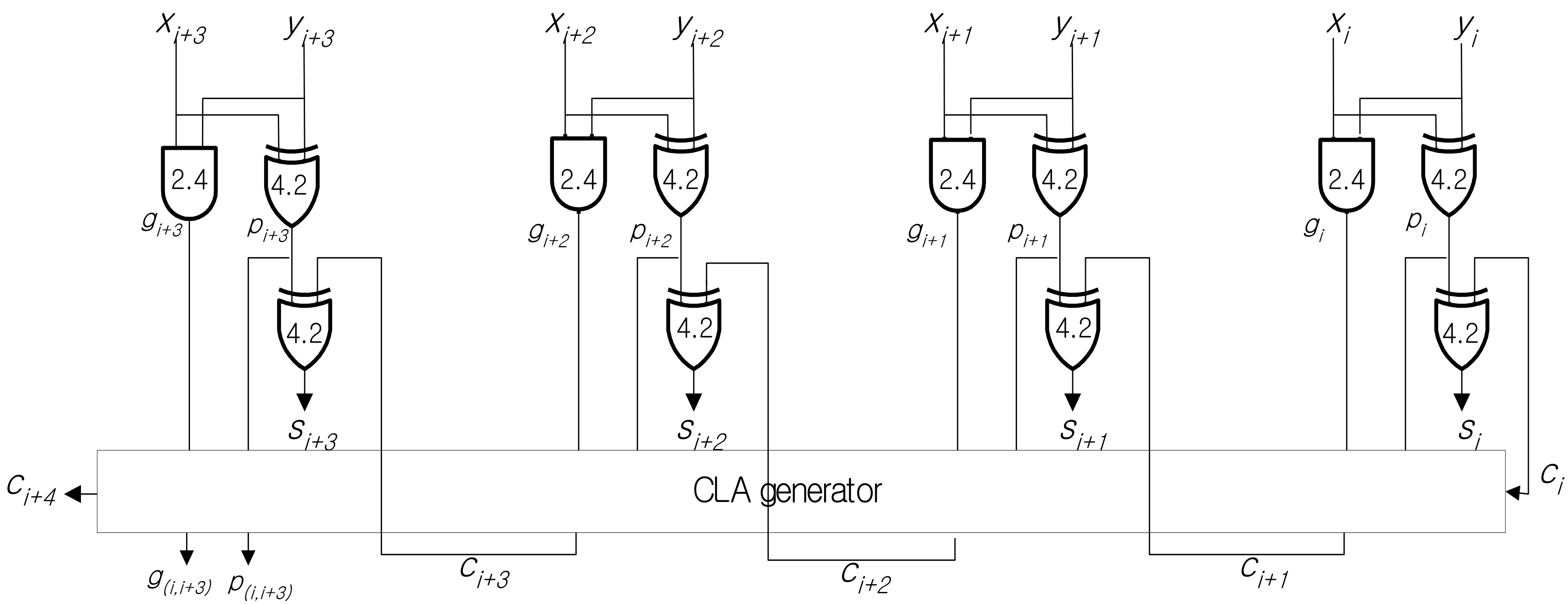
Carry generation of 4-bit ripple carry adder



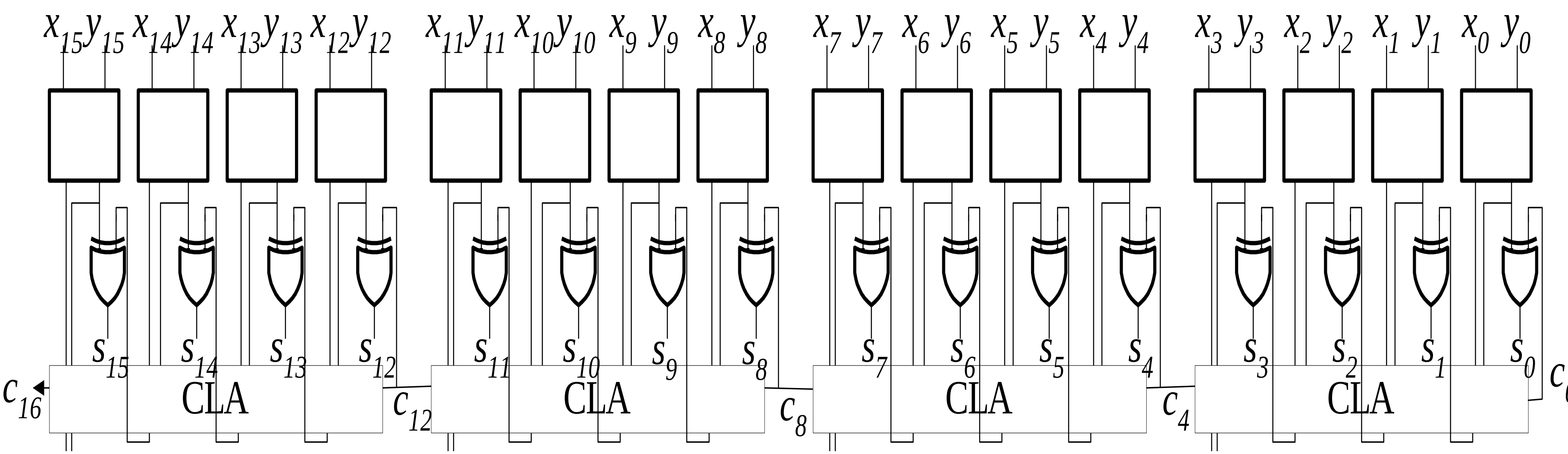
- carry-generate function $\boldsymbol{g_i = x_i y_i}$
- carry-propagate function $\boldsymbol{p_i = x_i \oplus y_i}$

$$c_{i+1} = g_i + p_i c_i$$
$$c_{i+2} = g_{i+1} + p_{i+1} g_i + p_{i+1} p_i c_i$$
$$c_{i+3} = g_{i+2} + p_{i+2} g_{i+1} + p_{i+2} p_{i+1} g_i + p_{i+2} p_{i+1} p_i c_i$$
$$c_{i+4} = g_{i+3} + p_{i+3} g_{i+2} + p_{i+3} p_{i+2} g_{i+1} + p_{i+3} p_{i+2} p_{i+1} g_i + p_{i+3} p_{i+2} p_{i+1} p_i c_i$$

4-bit carry look-ahead adder

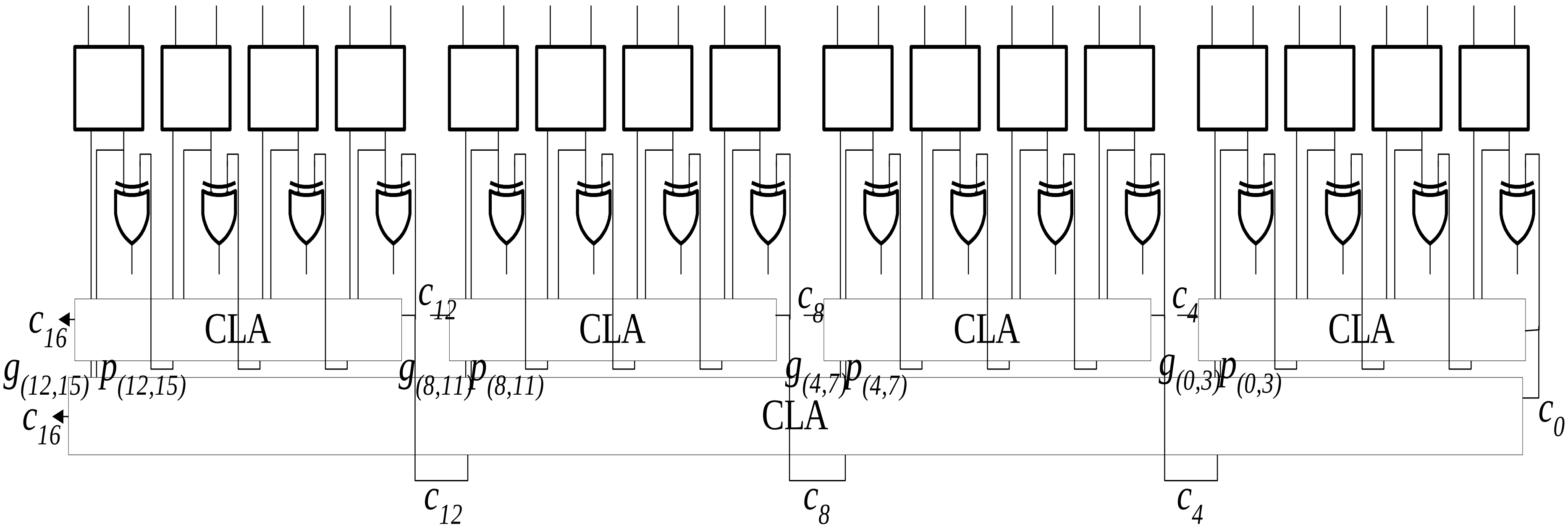


Single-level 16-bit carry look-ahead adder



2-level 16-bit carry look-ahead adder

- **Two-level CLA generator**
  - speed up by using an additional CLA generator
  - $c_4 = g_{(0,3)}+p_{(0,3)}c_0$ $\quad$ $c_8 = g_{(4,7)}+p_{(4,7)}c_4$
    $c_{12} = g_{(8,11)}+p_{(8,11)}c_8$ $\quad$ $c_{16} = g_{(12,15)}+p_{(12,15)}c_{12}$
  - second level generates these carries for the first-level CLA generator



(b) Two-level CLA generator

Delay comparison

CARRY CHAIN	RIPPLE DELAY	ONE-LEVEL CLA	TWO-LEVEL CLA
$c_0(x_0,y_0)$ to c_4	19.2 (23.4)	4.8 (13.0)	4.8 (13.2)
$c_0(x_0,y_0)$ to c_8	38.4 (42.6)	9.6 (17.8)	5.6 (16.2)
$c_0(x_0,y_0)$ to c_{12}	57.6 (61.8)	14.4 (22.6)	6.4 (17.0)
$c_0(x_0,y_0)$ to c_{16}	76.8 (81.0)	19.2 (27.4)	4.8 (19.4)

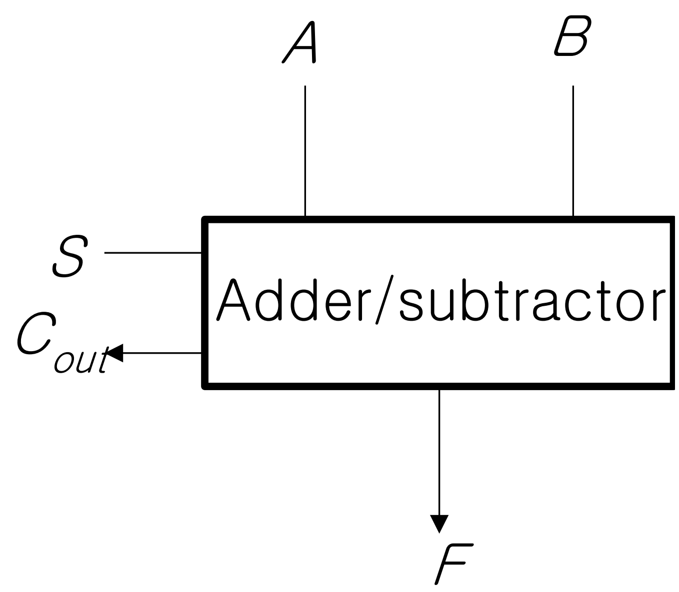
delay from x_0 or y_0

Adders/Subtractors

- **Binary subtraction**
  - add the minuend to the 2's complement of the subtrahend
- **Adders/Subtractor**
  - inputs : A = $a_{n-1} \ldots a_0$ and B = $b_{n-1} \ldots b_0$
  - output : F = $f_{n-1} \ldots f_0$
  - select signal : S

S	FUNCTION	COMMENT
0	A + B	Addition
1	A + B´+ 1	Subtraction

(a) Truth table



(b) Graphic symbol

Two's-complement adder/subtractor

Adders/Subtractors

• Subtraction : Develop the truth table for a 1-bit full subtractor and cascade as many of these as are needed, producing a borrow-ripple subtractor.

• Adder/Subtractor : signal line = 0 for addition & signal line = 1 for subtraction.

$1 \oplus x = x'$ and $0 \oplus x = x$

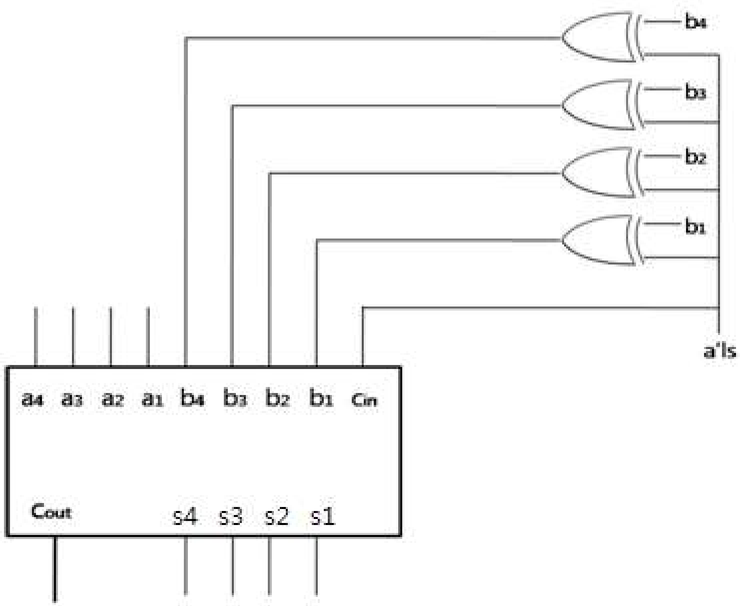


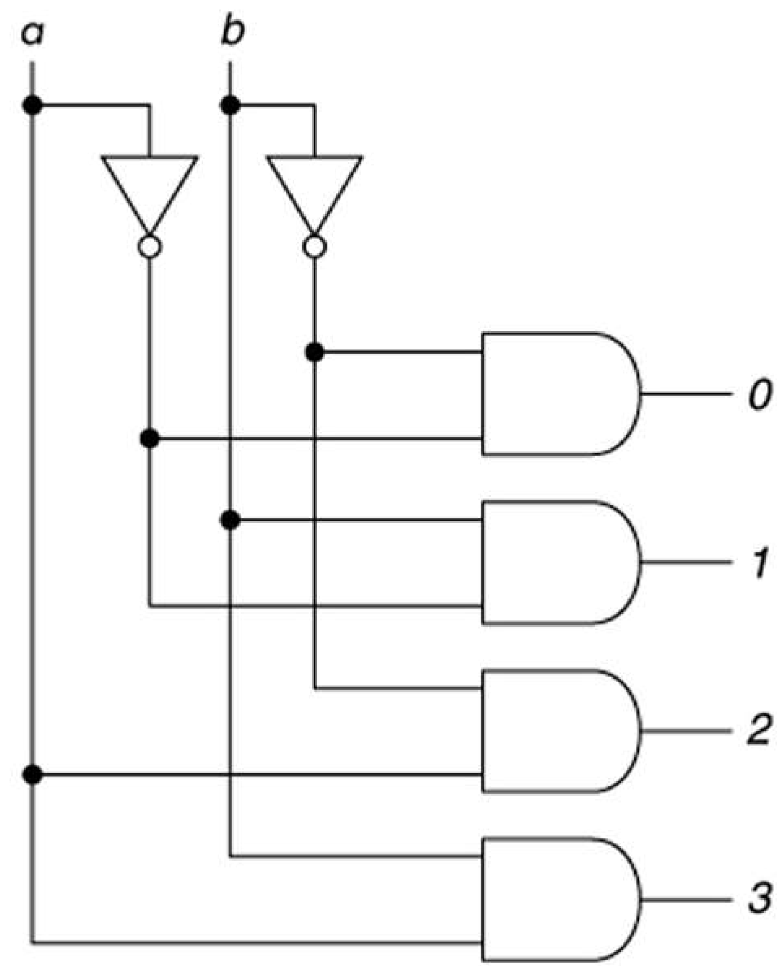
Figure 4.5

Binary decoders

Table 4.2a An active high decoder.

a	*b*	*0*	*1*	*2*	*3*
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Figure 4.8a An active high decoder.



Binary decoders

Table 4.2b An active low decoder.

a	*b*	*0*	*1*	*2*	*3*
0	0	0	1	1	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	1	1	1	0

Figure 4.8b An active low decoder.

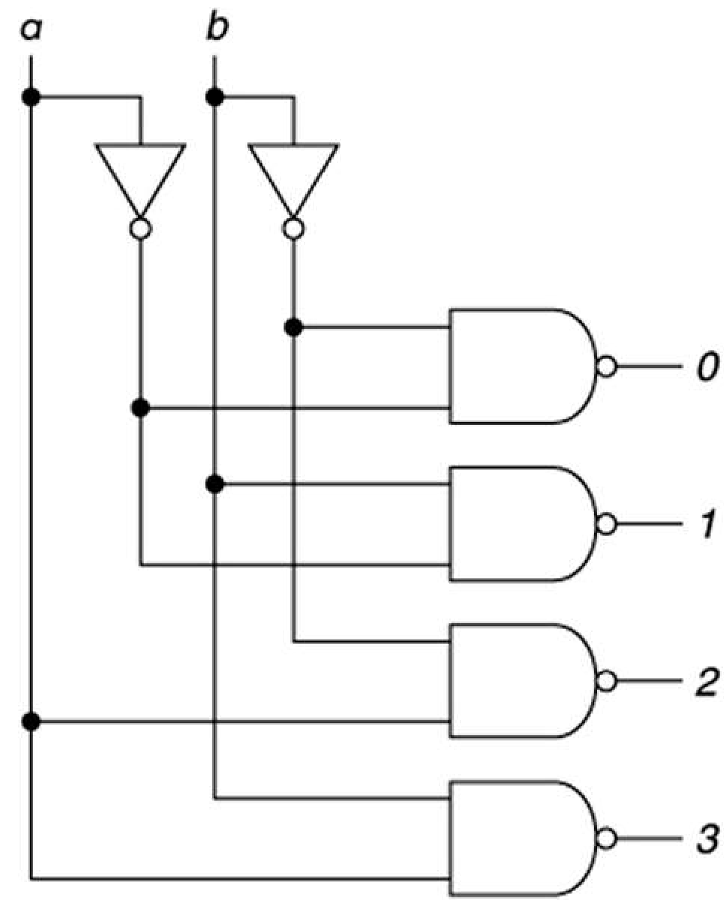
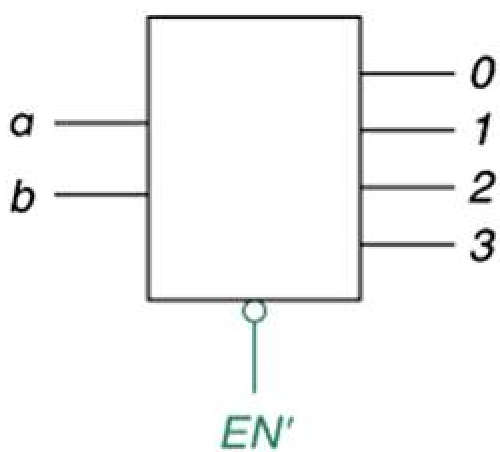
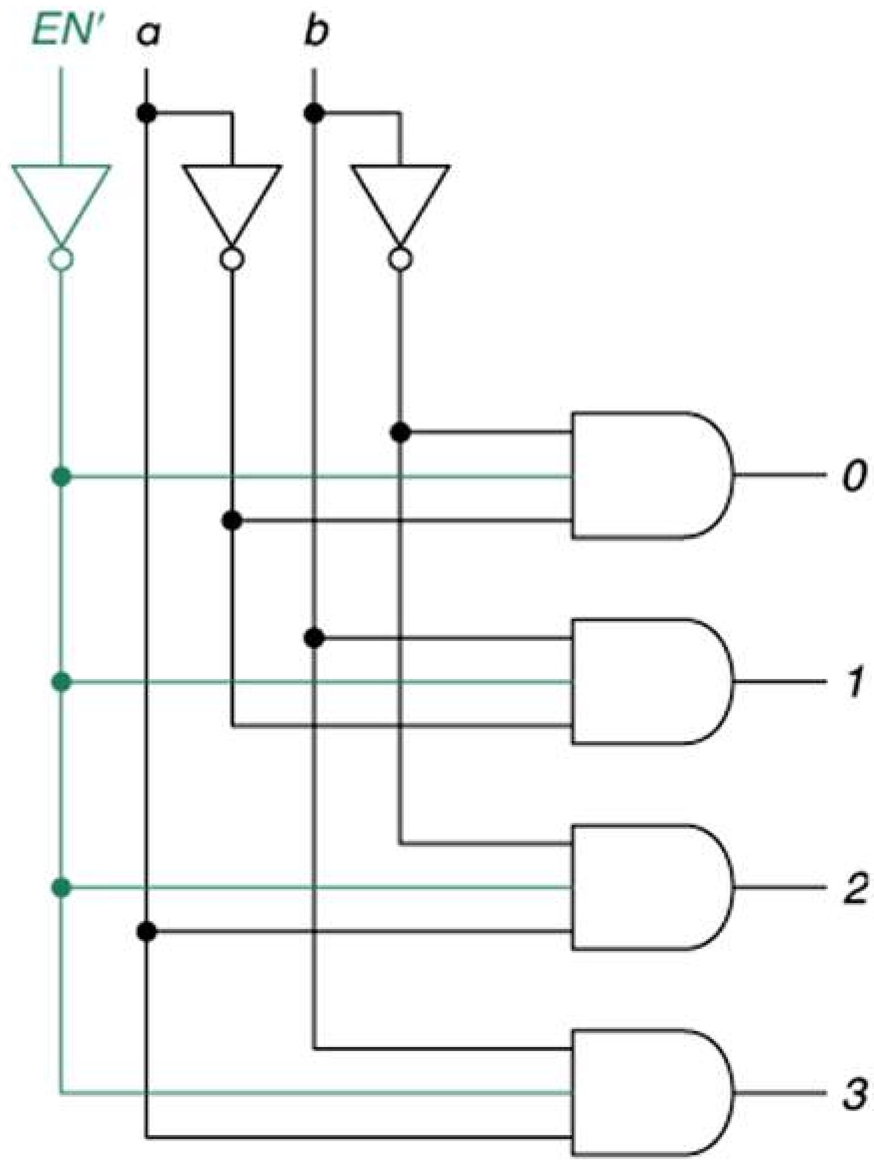


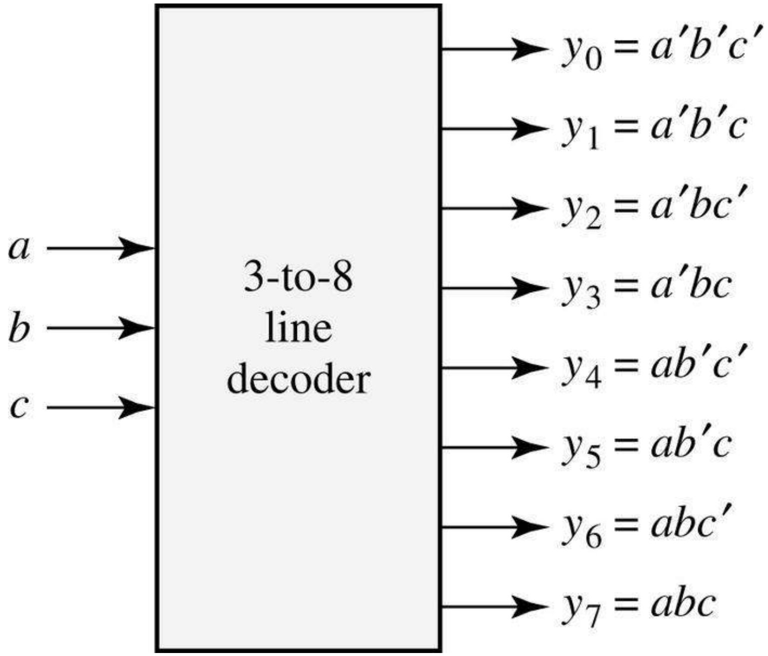
Figure 4.9 Decoder with enable.

EN'	a	b	0	1	2	3
1	X	X	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	1	0	0	0	1	0
0	1	1	0	0	0	1



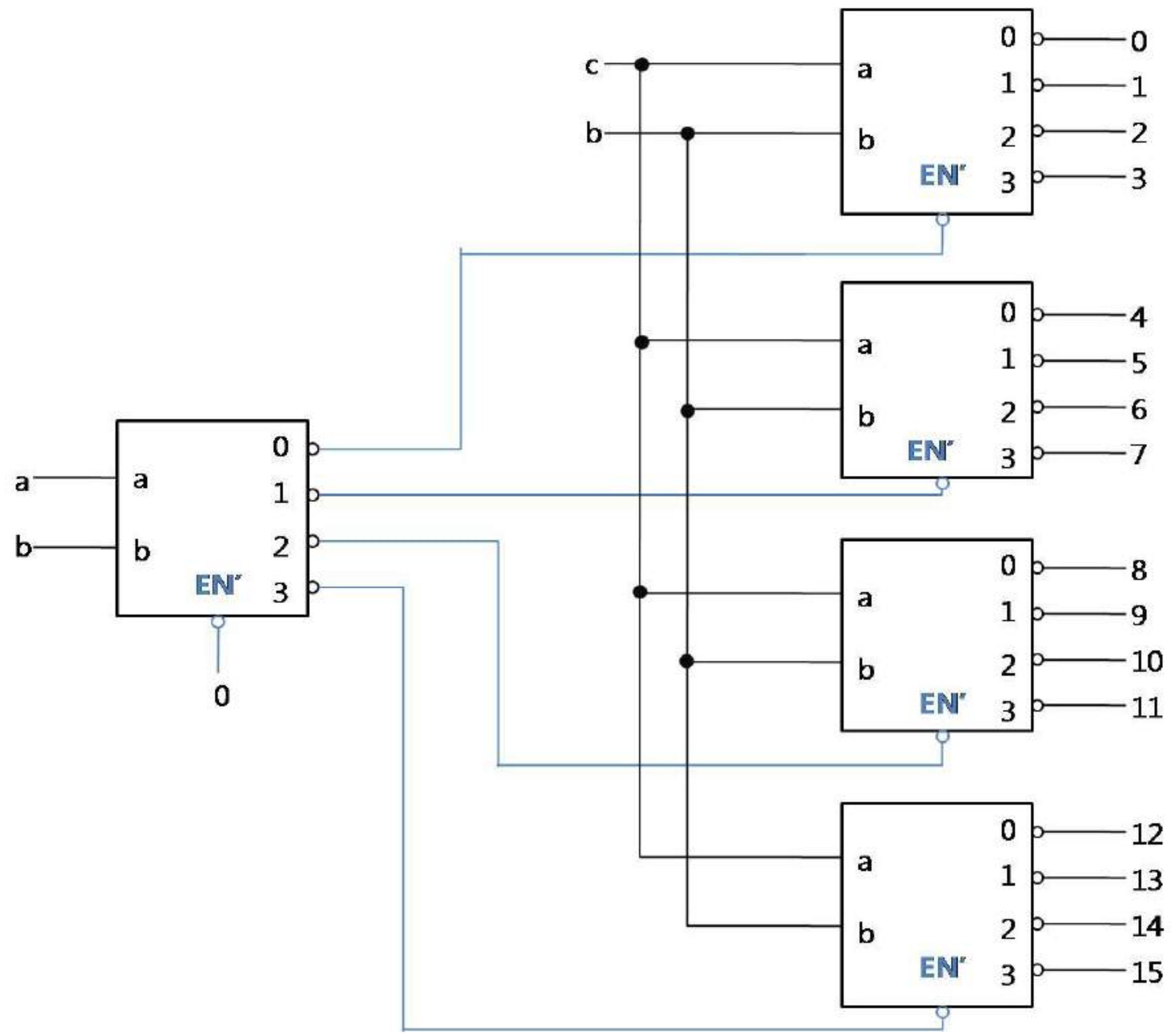


3x8 decoder



a b c	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_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

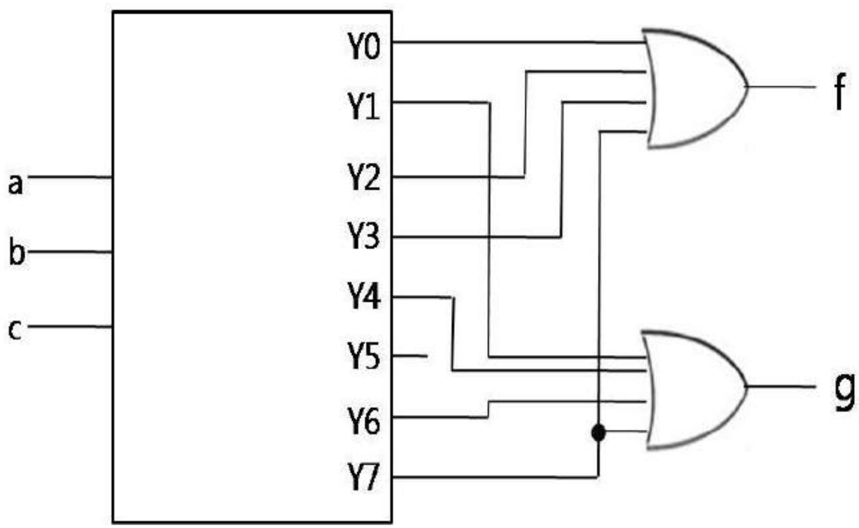
4x16 decoder using 2x4 decoders

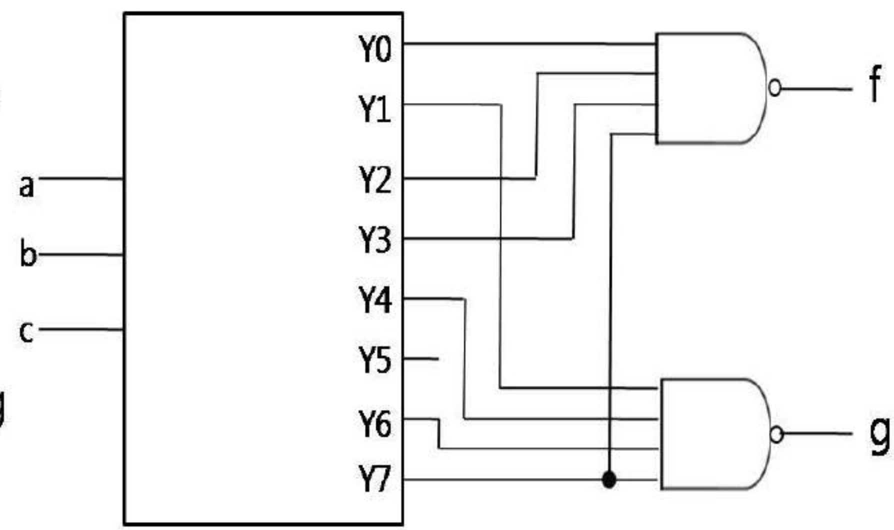


Logic function by decoders

- Example 4.3
  - Implementation of logic function

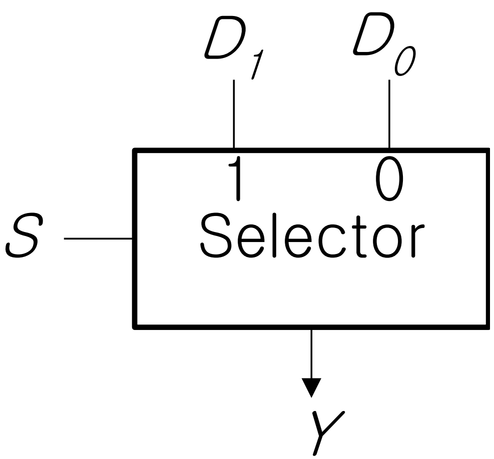
$$f(a, b, c) = \sum m(0, 2, 3, 7)$$
$$g(a, b, c) = \sum m(1, 4, 6, 7)$$





Multiplexers (data selectors)

- combinatorial component that can select one of several data sources to be used as operands for an ALU
- n inputs, one output, $\log_2 n$ select signals



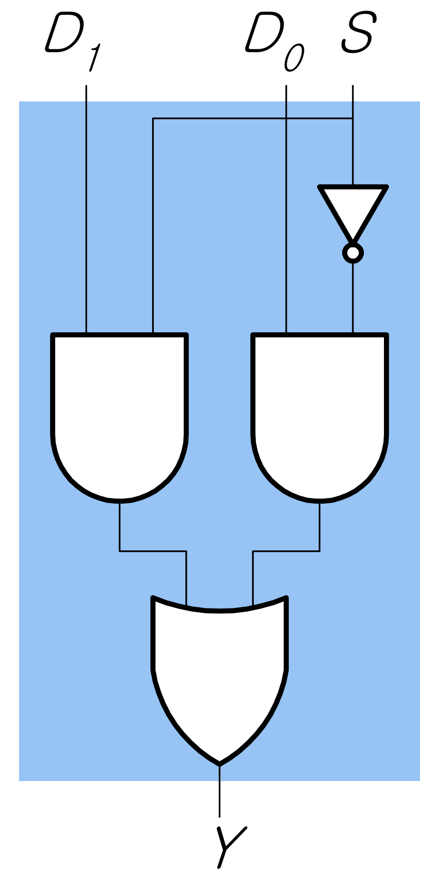
(a) Graphic symbol

S	Y
0	D_0
1	D_1

(b) Truth table

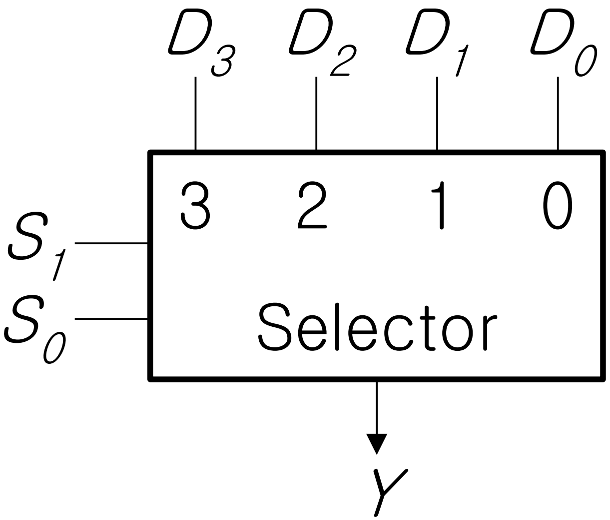
$$Y = S'D_0 + SD_1$$

(c) Boolean expression



(d) Logic diagram

4x1 Mutiplexer



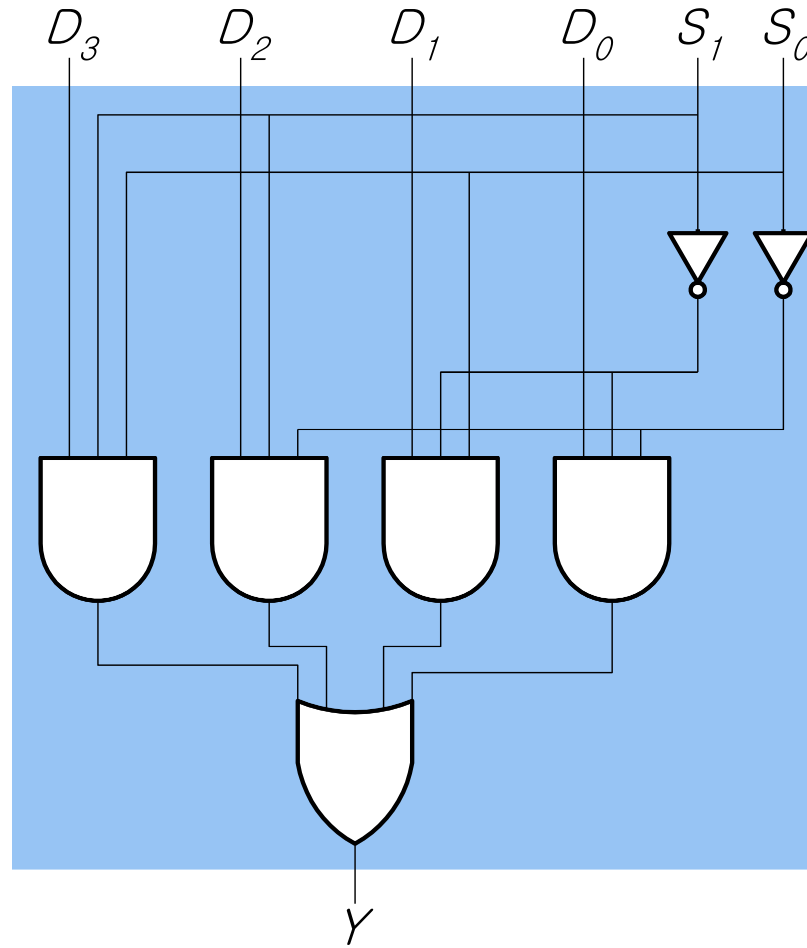
(a) Graphic symbol

S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

(b) Truth table

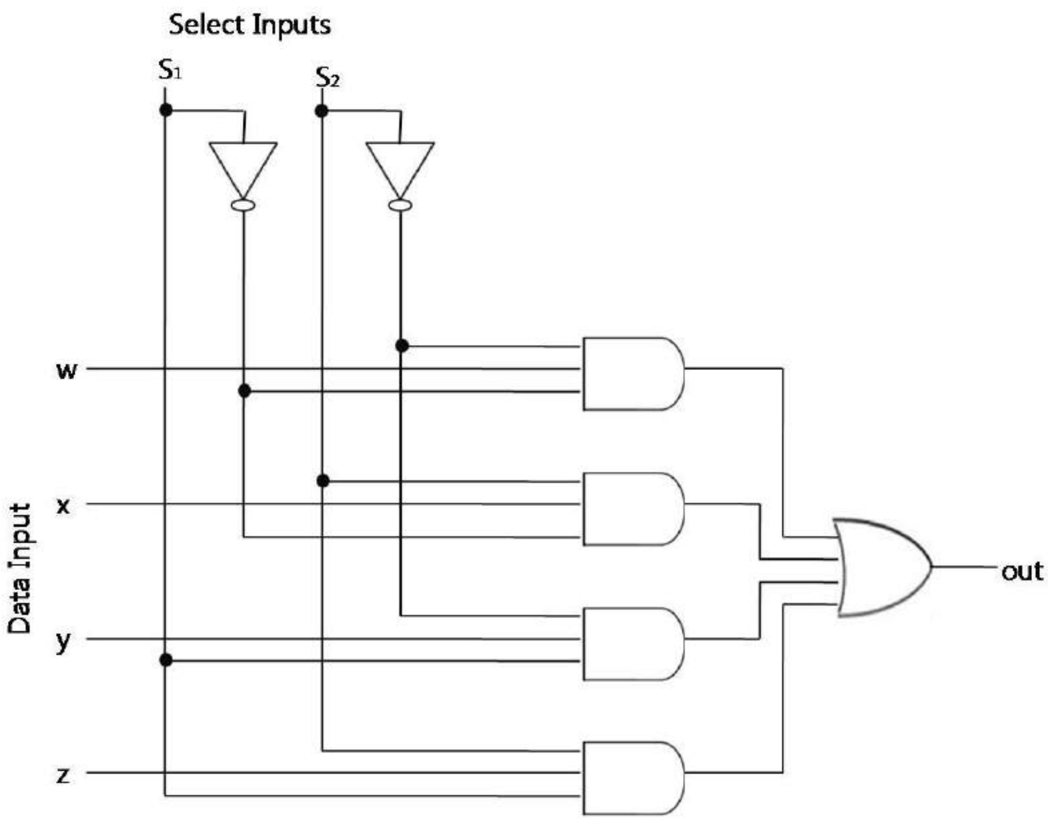
$$Y = S_1'S_0'D_0 + S_1'S_0D_1 + S_1S_0'D_2 + S_1S_0D_3$$

(c) Boolean expression

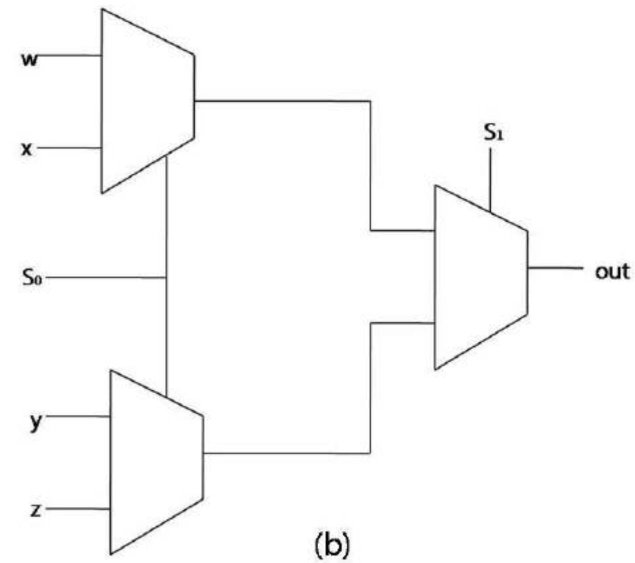


(d) Logic diagram

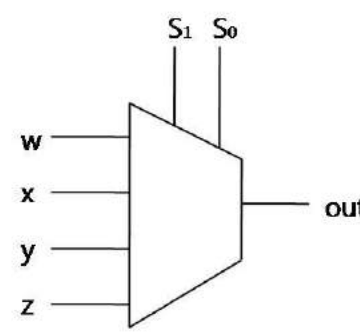
4x1 MUX using 3 (2x1) MUX



(a)



(b)

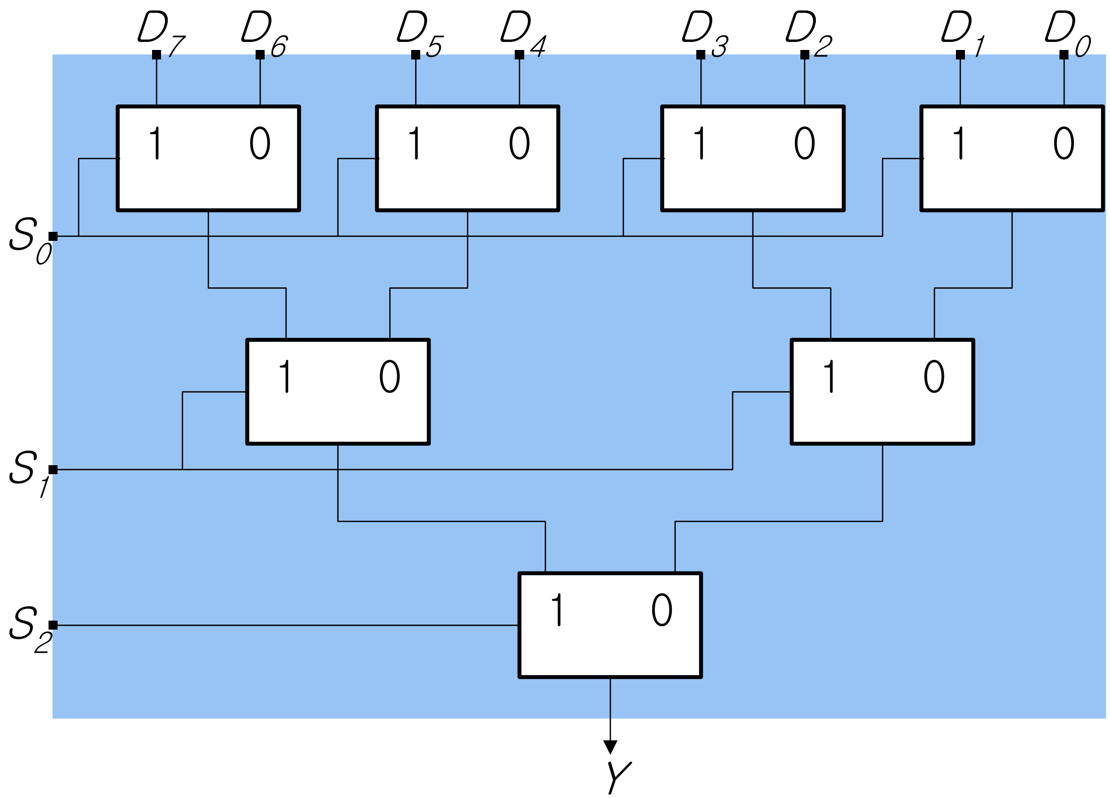


(c)

8x1 Multiplexer (1)

S_2	S_1	S_0	Y
0	0	0	D_0
0	0	1	D_1
0	1	0	D_2
0	1	1	D_3
1	0	0	D_4
1	0	1	D_5
1	1	0	D_6
1	1	1	D_7

(a) Truth table

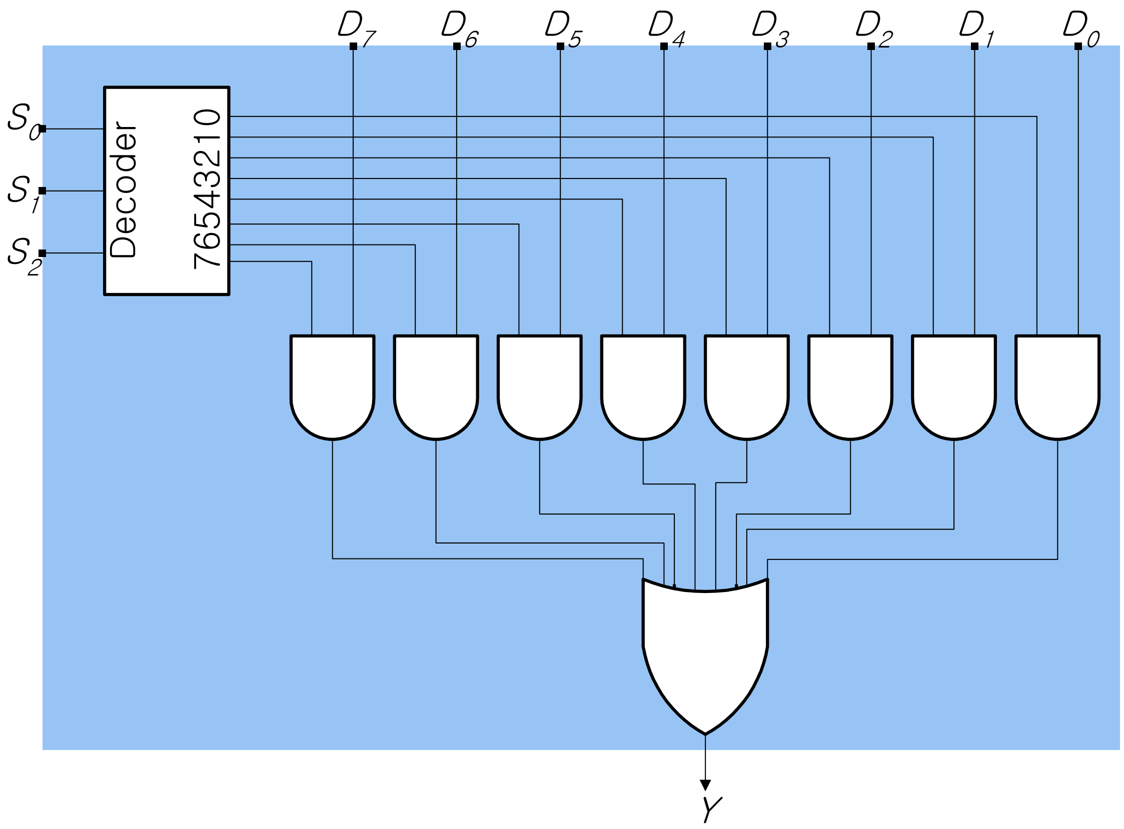


(b) Implementation with 2-to-1 selectors

8x1 multiplexer (2)

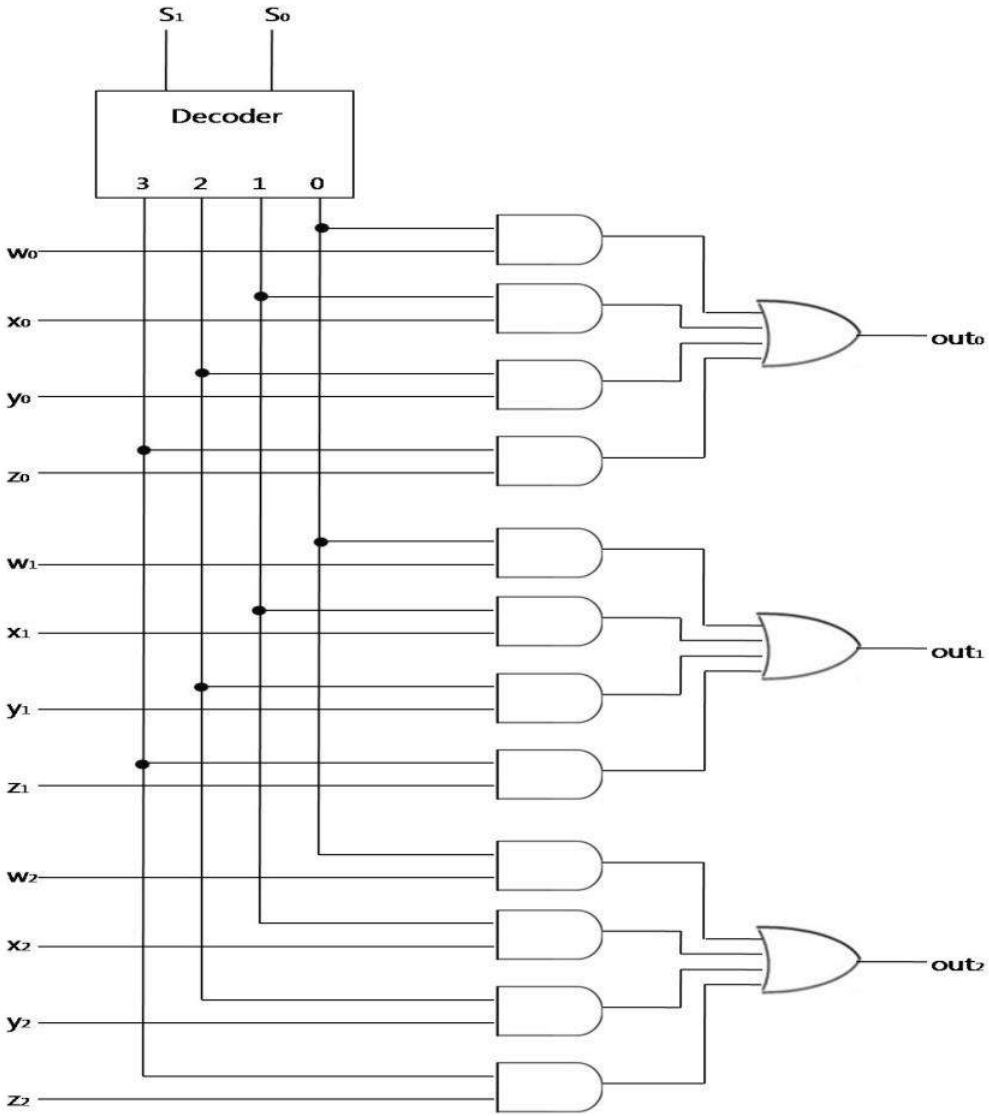
S_2	S_1	S_0	Y
0	0	0	D_0
0	0	1	D_1
0	1	0	D_2
0	1	1	D_3
1	0	0	D_4
1	0	1	D_5
1	1	0	D_6
1	1	1	D_7

(a) Truth table



(b) Implementation with a decoder

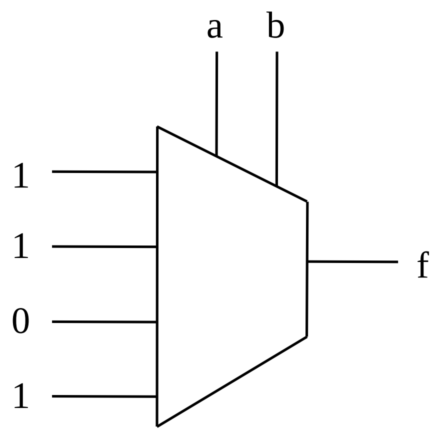
Multi-bit multiplexer



Function implementation using mux

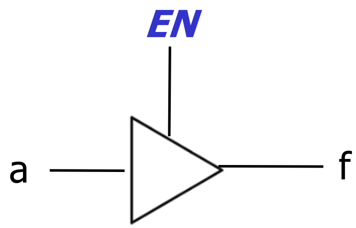
- Multiplexers can be used to implement logic functions.
- Example 4.4
  - Implement the function
    $f(a, b) = \sum m(0, 1, 3)$

a	*b*	*f*
0	0	1
0	1	1
1	0	0
1	1	1



Three state gates (buffers)

EN	a	F
0	0	Z
0	1	Z
1	0	0
1	1	1



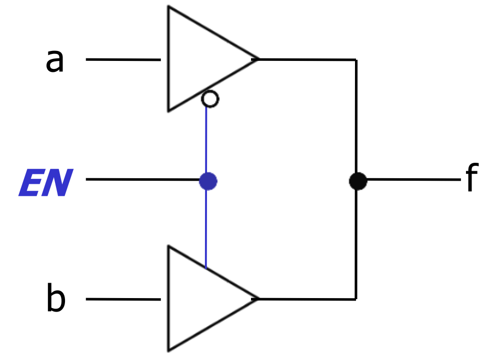
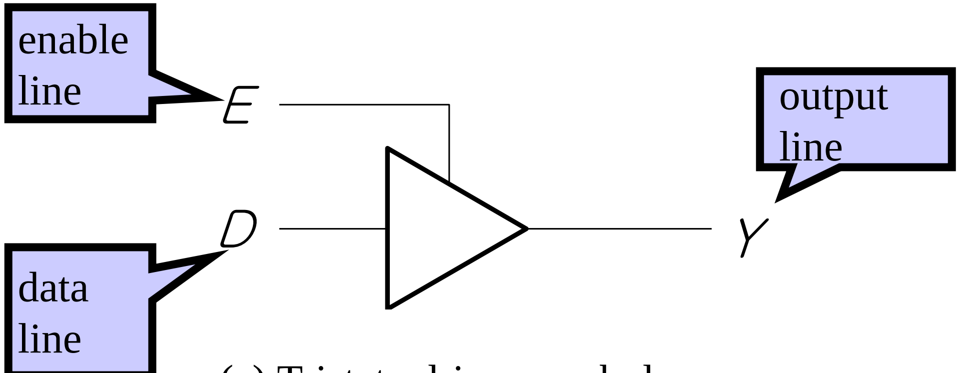


Figure 4.15

- Three-state buffers with active low enables and/or outputs exist.
- Three-state outputs also exist on other more complex gates.
- A multiplexer using three-state gates. (without the OR gate)
    - The enable is the control input : f = a (EN = 0) or f = b (EN = 1)
    - The three-state gate is often used for signals that travel between systems.
    - bus : it is a set of lines over which data is transferred.

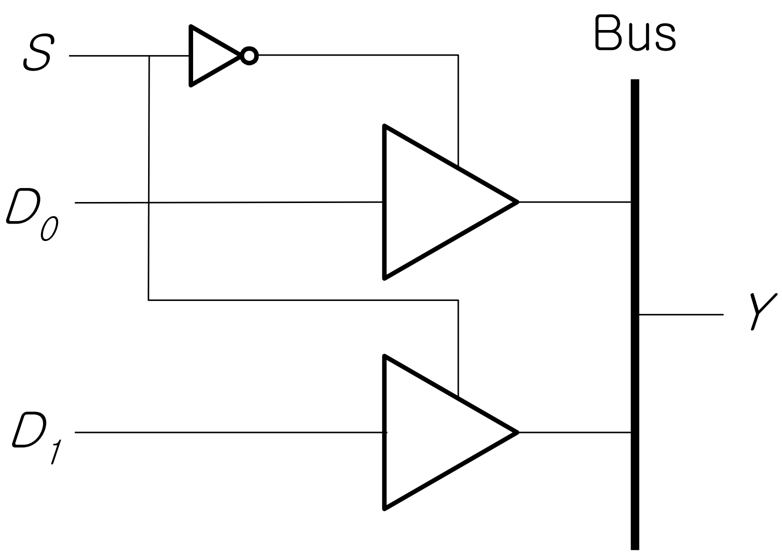
BUS



(a) Tristate driver symbol

E	Y
0	Z
1	D

(b) Truth table for tristate driver

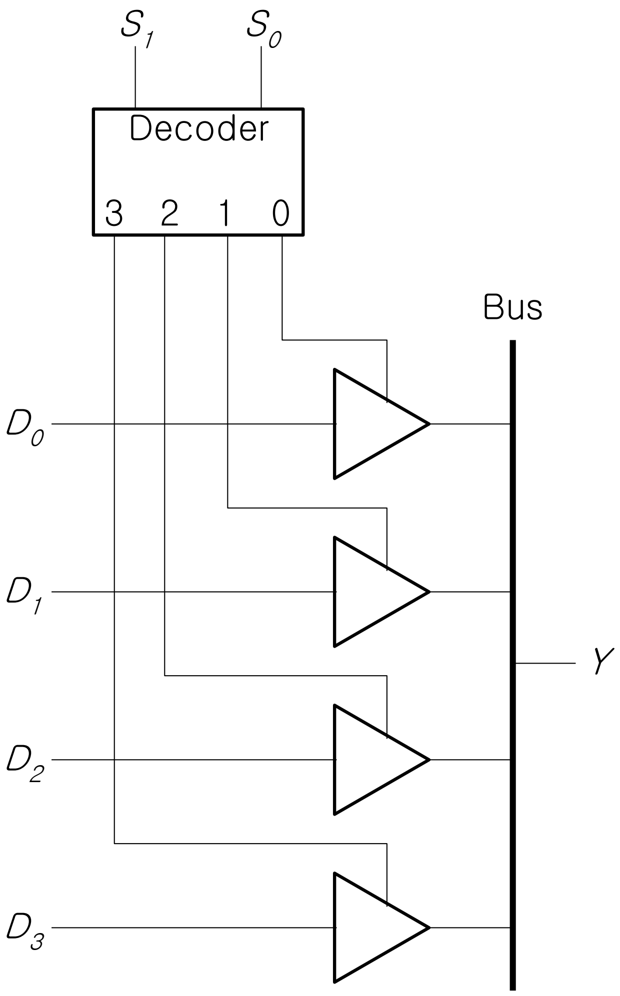


(c) 2-input bus

S	Y
0	D_0
1	D_1

(d) Truth table for 2-input bus

bus



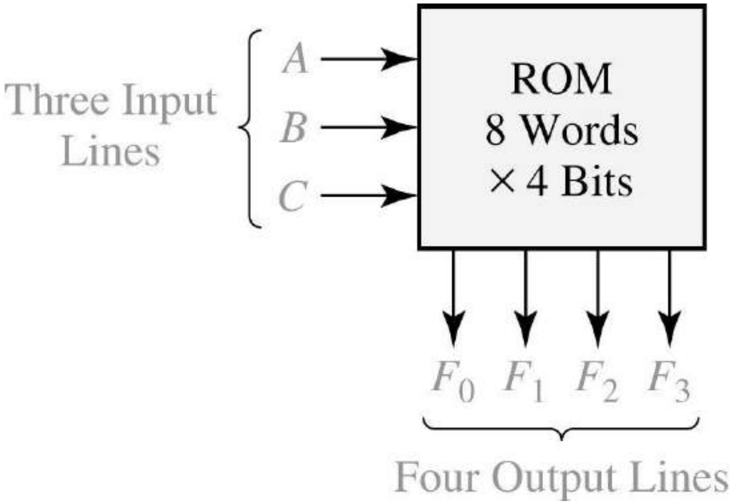
(e) 4-input bus

S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

(f) Truth table for 4-input bus

Read-Only Memory (ROM)

- An 8-Word x 4-Bit ROM



(a) Block diagram

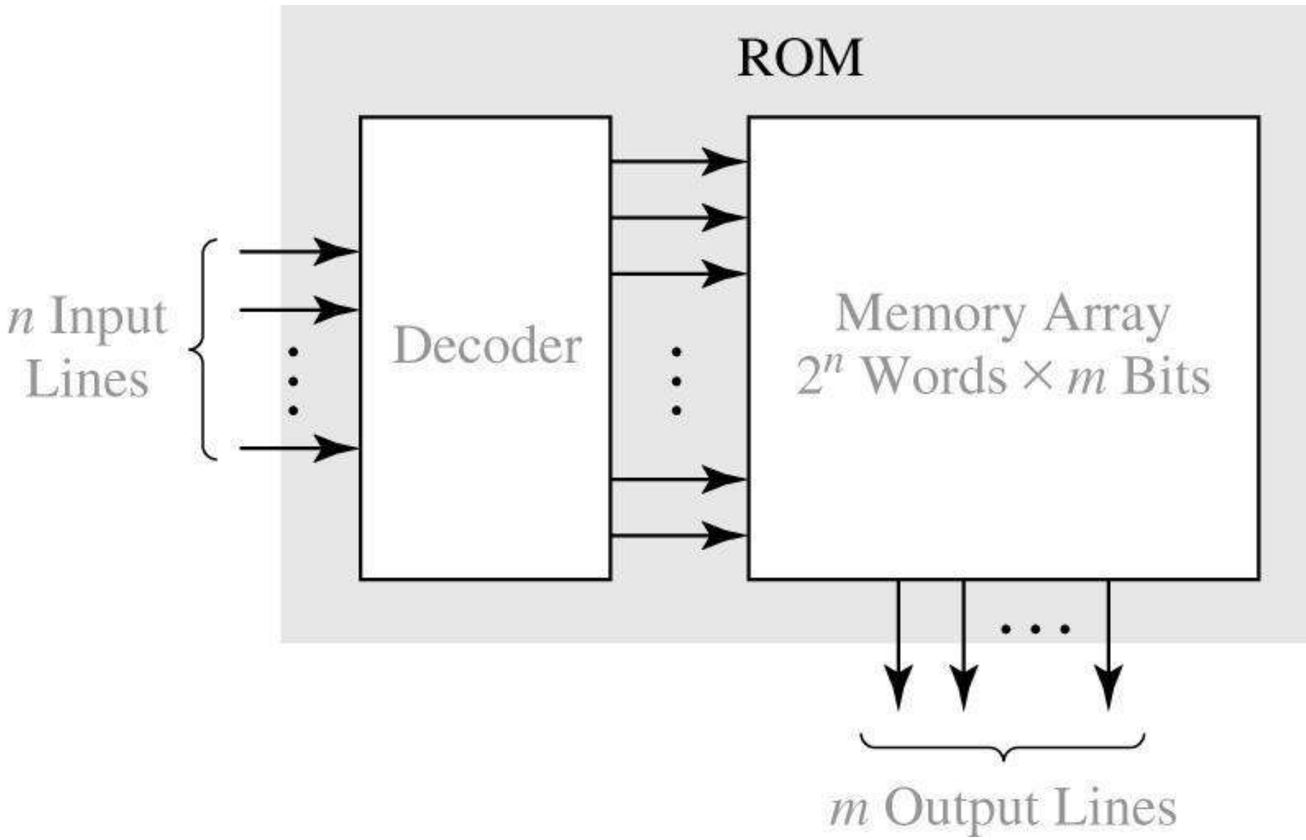
A	B	C	F_0	F_1	F_2	F_3
0	0	0	1	0	1	0
0	0	1	1	0	1	0
0	1	0	0	1	1	1
0	1	1	0	1	0	1
1	0	0	1	1	0	0
1	0	1	0	0	0	1
1	1	0	1	1	1	1
1	1	1	0	1	0	1

typical data stored in ROM

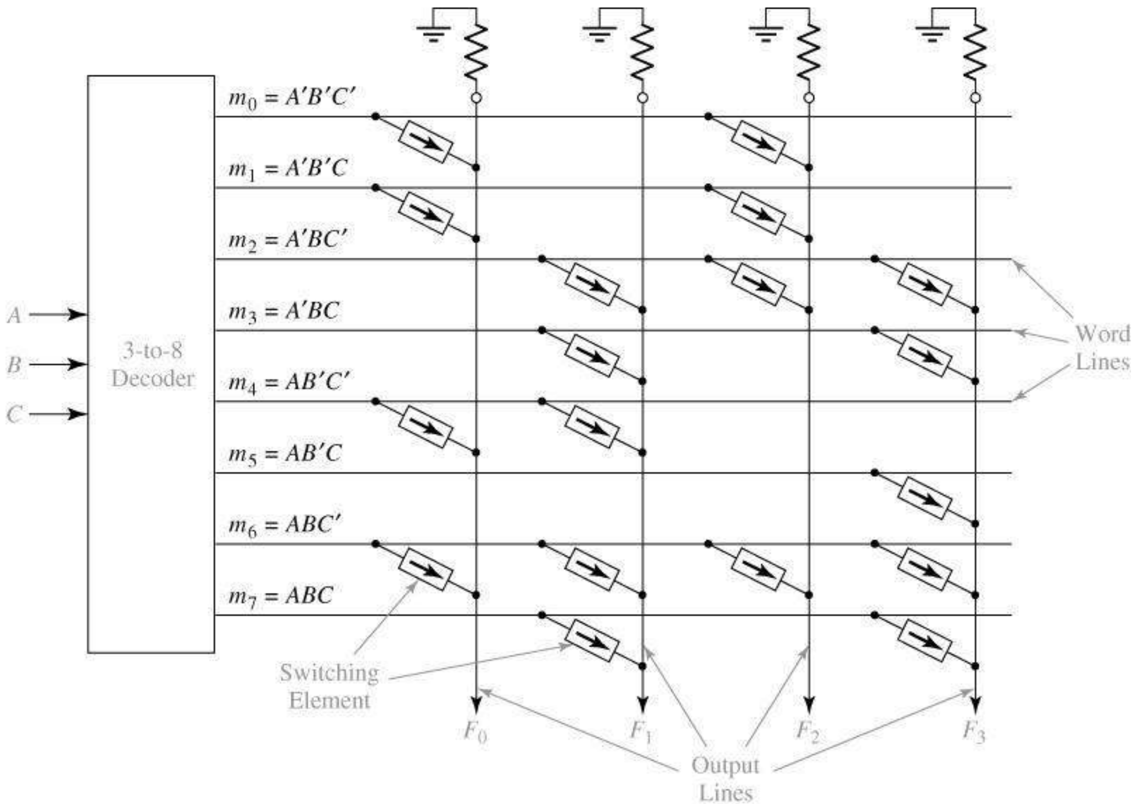
(2^3 words of 4bits each)

(b) Truth table for ROM

Basic ROM structure



An 8-Word x 4-Bit ROM



$$F_0 = \sum m(0,1,4,6) = A'B' + AC'$$

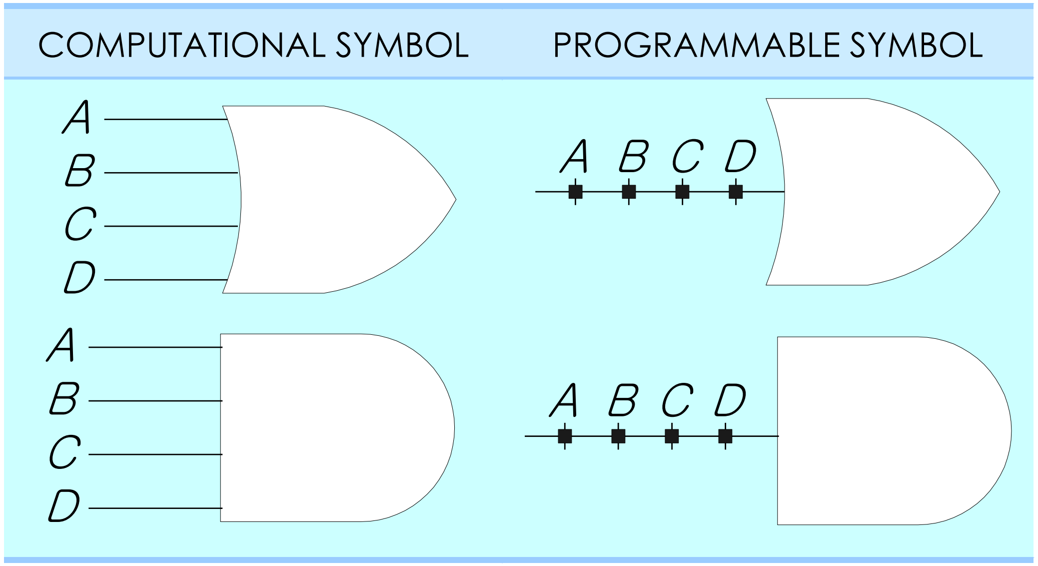
$$F_1 = \sum m(2,3,4,6,7) = B + AC'$$

$$F_2 = \sum m(0,1,2,6) = A'B' + BC'$$

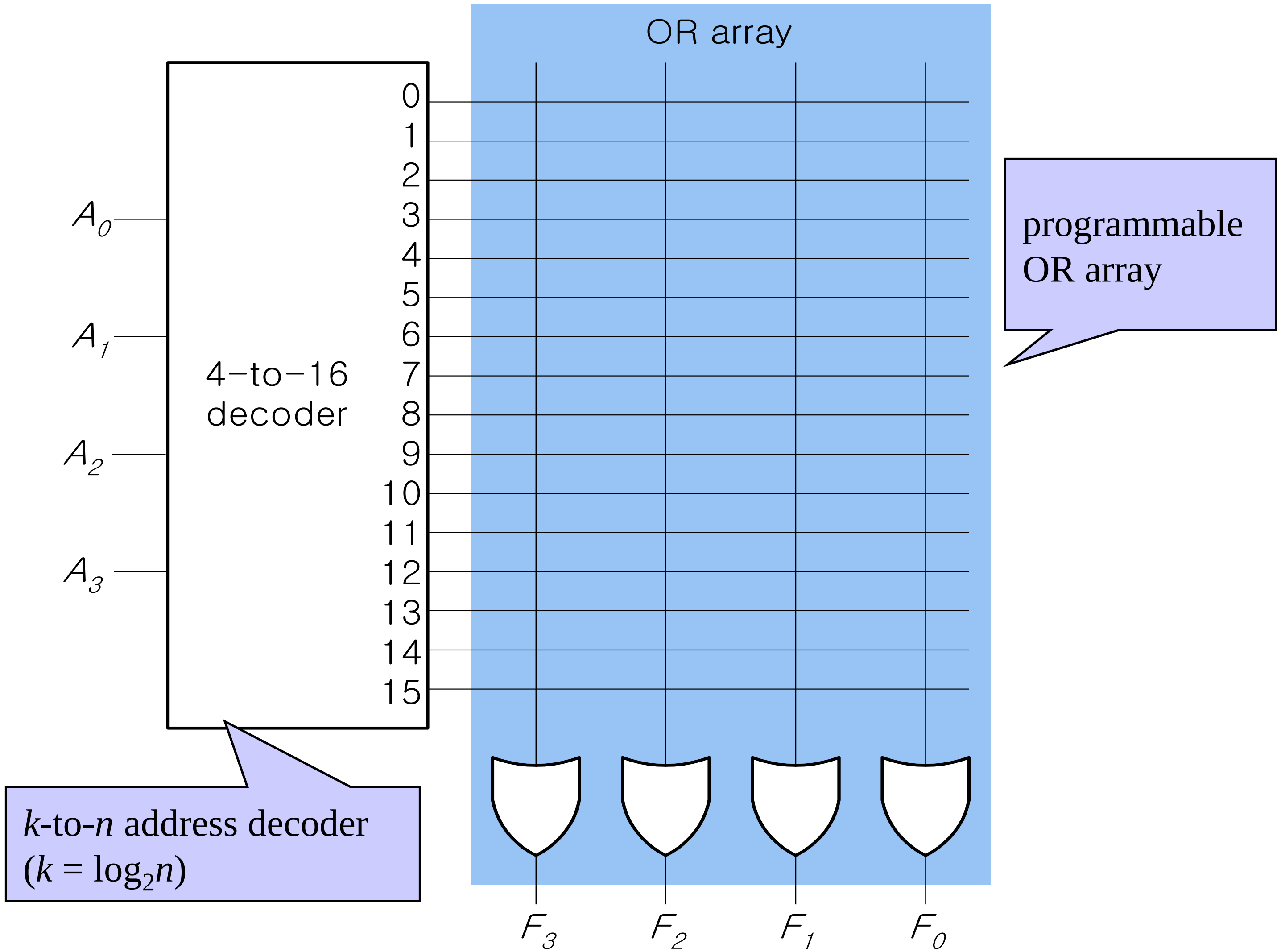
$$F_3 = \sum m(2,3,5,6,7) = AC + B4$$

ROM

- **programmable versions of the conceptional AND and OR gate**
  - connections are made
    - during manufacturing, when physically connect two lines whenever a connection is desired
    - in the field, when burn the fuse between an input line and a gate line whenever a connection is not desired



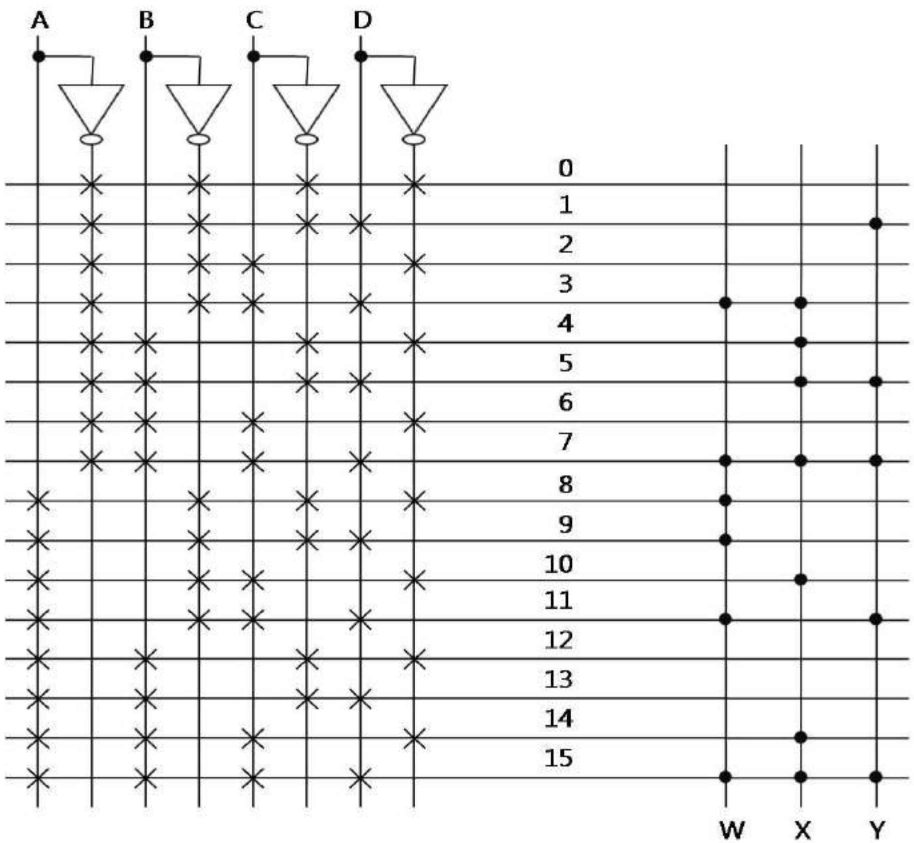
16×4 ROM



Designing with Read-Only Memories

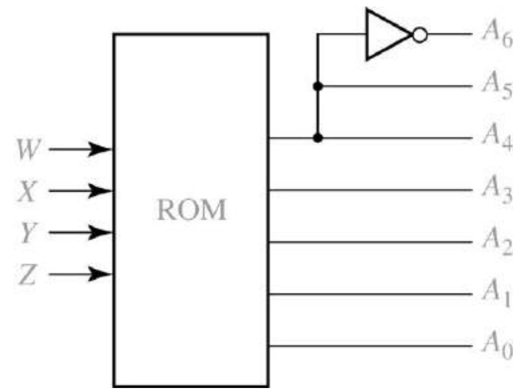
- Example 4.7

  W(A, B, C, D) = Σm(3, 7, 8, 9, 11, 15), X(A, B, C, D) = Σm(3, 4, 5, 7, 10, 14, 15), Y(A, B, C, D) = Σm(1, 5, 7, 11, 15)

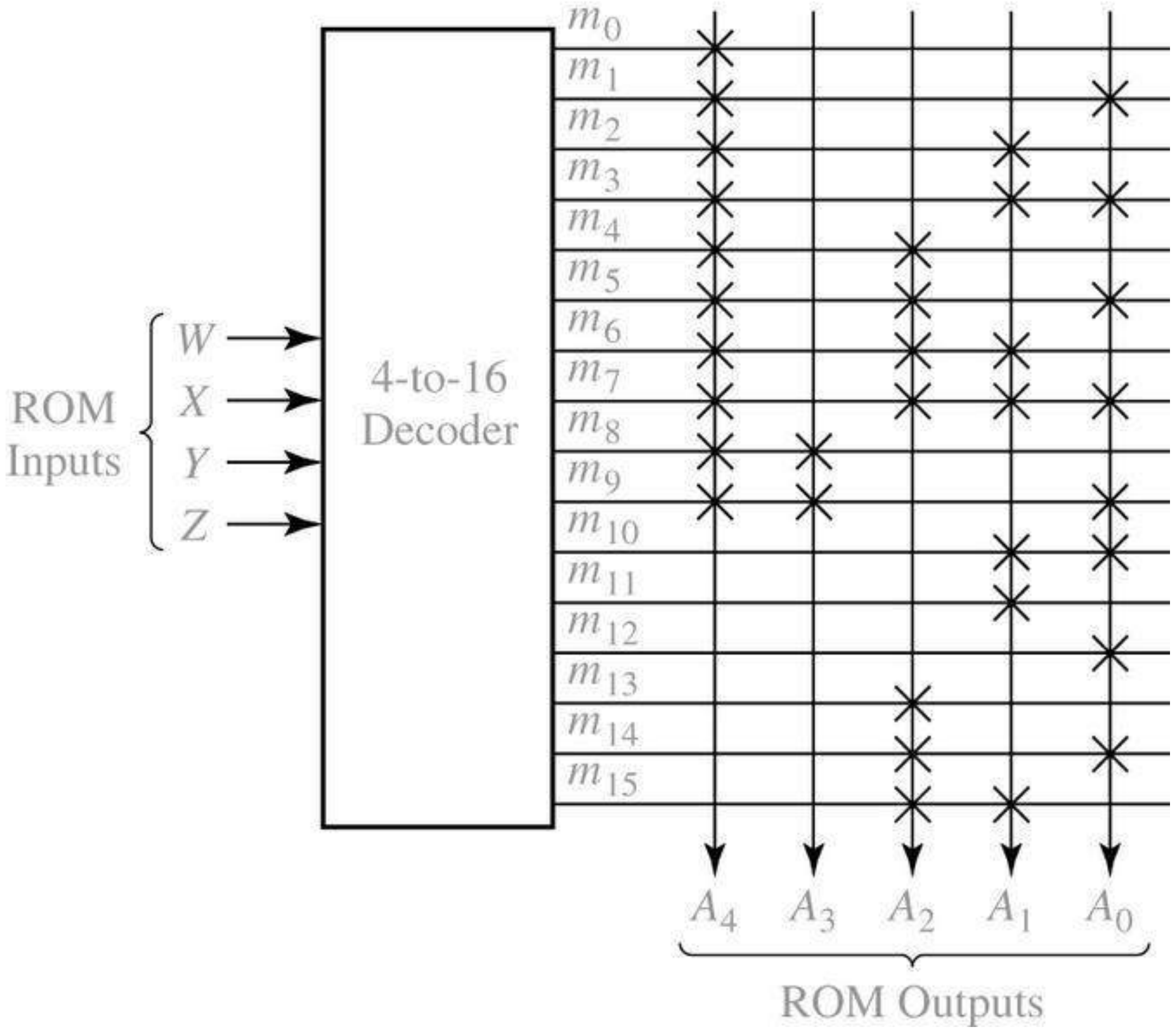


Hexadecimal to ASCII Code Converter

Input				Hex Digit	ASCII Code for Hex Digit						
W	X	Y	Z		A_6	A_5	A_4	A_3	A_2	A_1	A_0
0	0	0	0	0	0	1	1	0	0	0	0
0	0	0	1	1	0	1	1	0	0	0	1
0	0	1	0	2	0	1	1	0	0	1	0
0	0	1	1	3	0	1	1	0	0	1	1
0	1	0	0	4	0	1	1	0	1	0	0
0	1	0	1	5	0	1	1	0	1	0	1
0	1	1	0	6	0	1	1	0	1	1	0
0	1	1	1	7	0	1	1	0	1	1	1
1	0	0	0	8	0	1	1	1	0	0	0
1	0	0	1	9	0	1	1	1	0	0	1
1	0	1	0	A	1	0	0	0	0	0	1
1	0	1	1	B	1	0	0	0	0	1	0
1	1	0	0	C	1	0	0	0	0	1	1
1	1	0	1	D	1	0	0	0	1	0	0
1	1	1	0	E	1	0	0	0	1	0	1
1	1	1	1	F	1	0	0	0	1	1	0



ROM realization of code converter



PROGRAMMABLE LOGIC ARRAYS

- **ROMs**
  - have only a small number of 1's, so many of the words have a value of 0

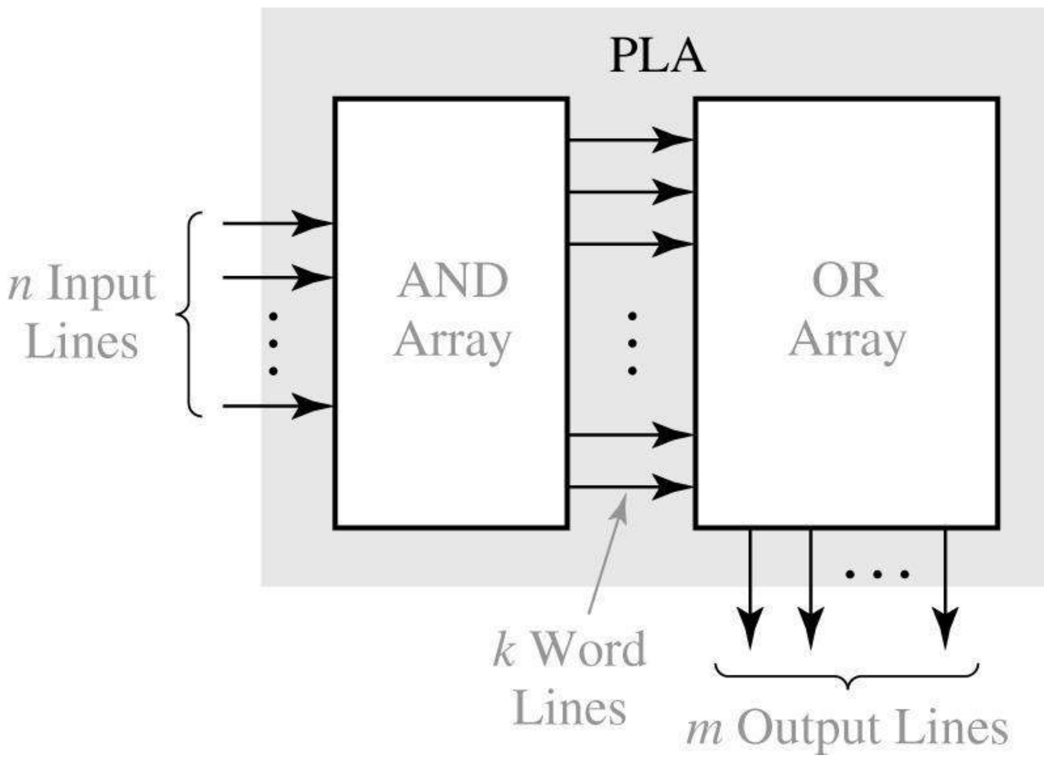
    $\Rightarrow$ programmable logic arrays (PLAs) minimize this waste

- **PLAs differ from ROMs in the address decoder**
  - instead of a full decoder, use a programmable decoder (AND array), which can be programmed to decode only those words that have a nonzero content
  - programmable output array used whenever to complement the output values

- **Advantage of PLAs**
  - no less flexible than ROMs
  - more efficient in their implementation of random logic
  - used more often for the implementation of control logics, whereas ROMs are used most frequently for tables of coefficients, startup programs, test vectors, and other random data

Programmable Logic Devices

- Programmable Logic Array Structure



PLA implementation of the full adder

A_3	A_2	A_1	A_0	F_3	F_2	F_1	F_0
	x_i	y_i	c_i			s_i	c_{i+1}
X	0	0	0	X	X	0	0
X	0	0	1	X	X	0	1
X	0	1	0	X	X	0	1
X	0	1	1	X	X	1	0
X	1	0	0	X	X	0	1
X	1	0	1	X	X	1	0
X	1	1	0	X	X	1	0
X	1	1	1	X	X	1	1

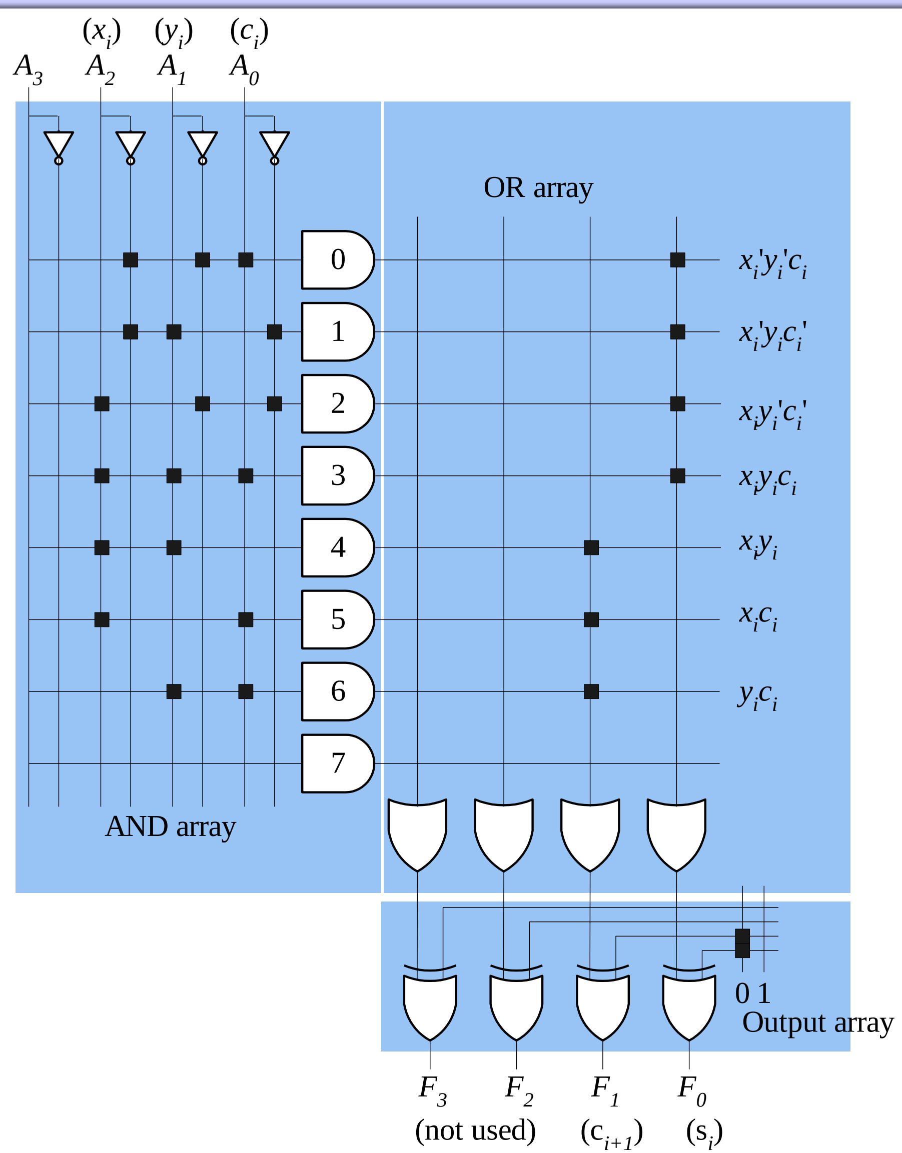
c_i \ $x_i y_i$	00	01	11	10
0		**1**		**1**
1	**1**		**1**	

$$S_i = x_i' y_i' c_i + x_i' y_i c_i' + x_i y_i' c_i' + x_i y_i c_i$$

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

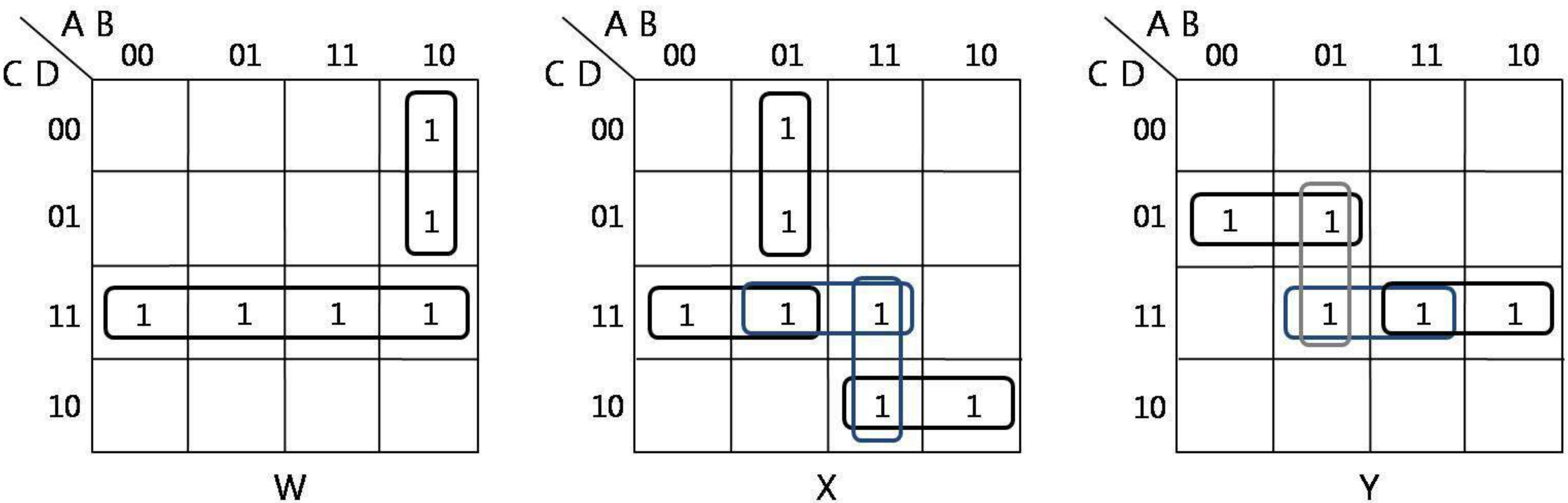
$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

PLA implementation of full-adder



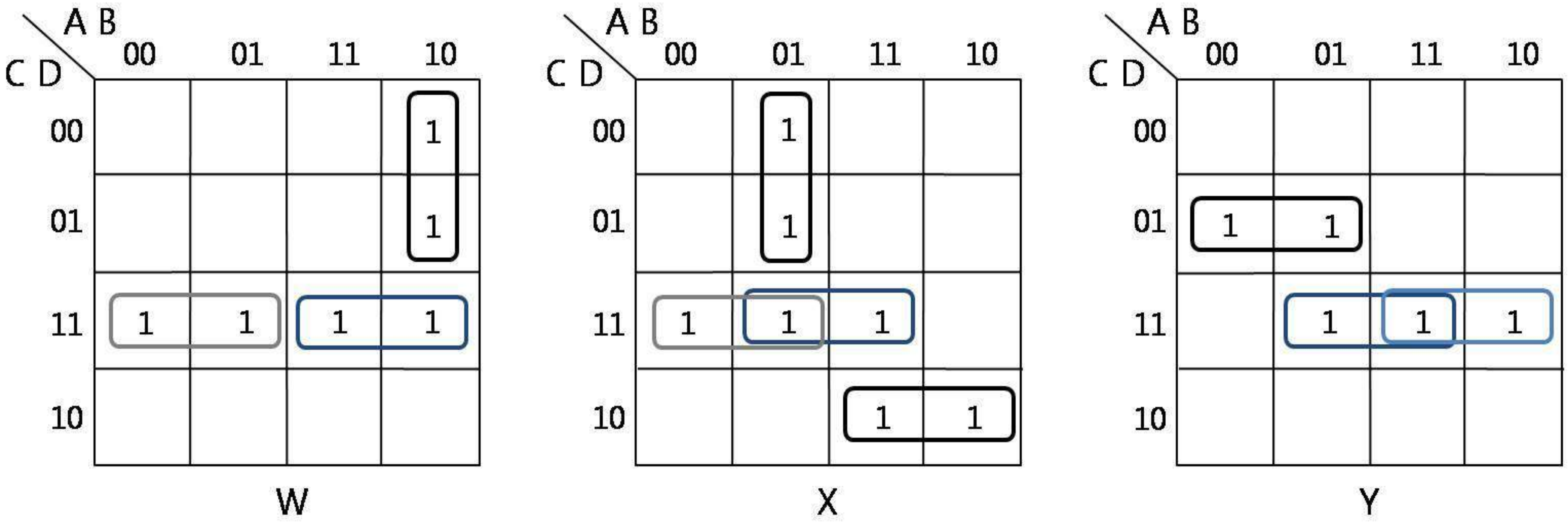
Designing with Programmable Logic Arrays

- Example 4.8
  - W(A, B, C, D) = Σm(3, 7, 8, 9, 11, 15), X(A, B, C, D) = Σm(3, 4, 5, 7, 10, 14, 15),   Y(A, B, C, D) = Σm(1, 5, 7, 11, 15)



- W= AB′C′ + CD
- X = A′BC′ + A′CD + ACD′ + {BCD or ABC}
- Y = A′C′D + ACD + {A′BD or BCD}

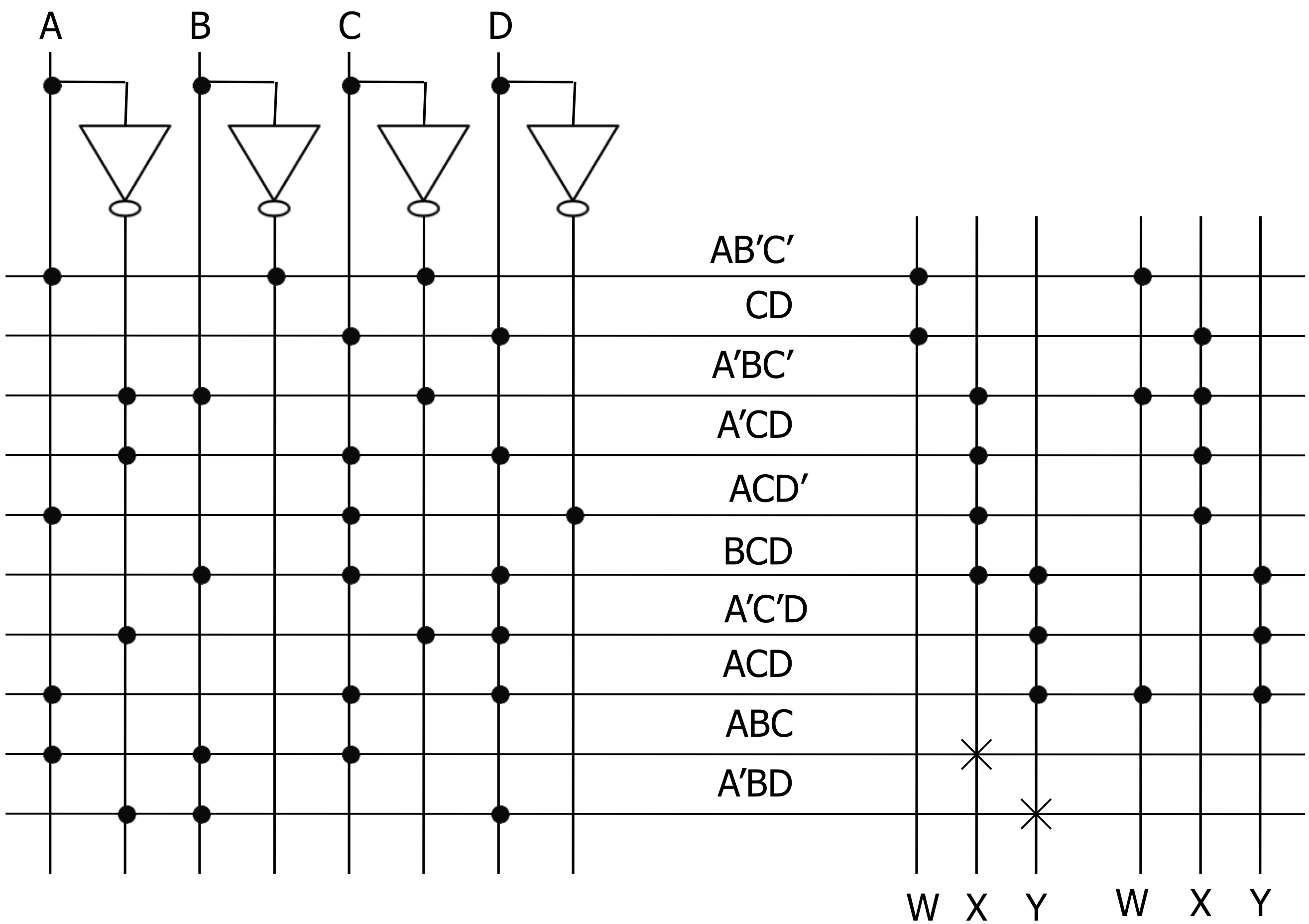
PLA example



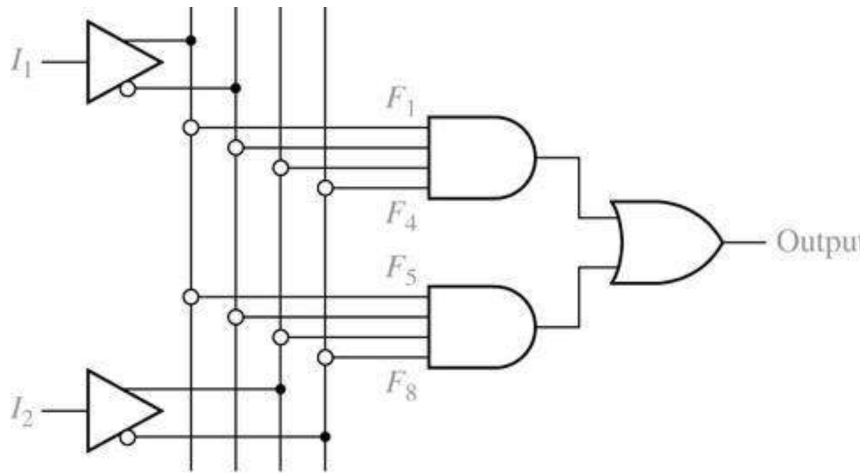
- W= AB′C′ + A′CD + ACD
- X = A′BC′ + ACD′ + A′CD + BCD
- Y = A′C′D + ACD + BCD

- This solution only uses seven terms instead of eight or nine.

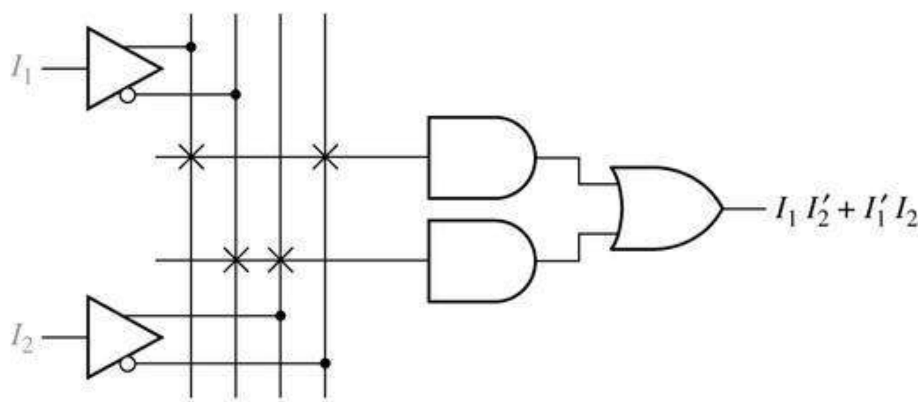
solution



Programmable logic array (PAL) devices

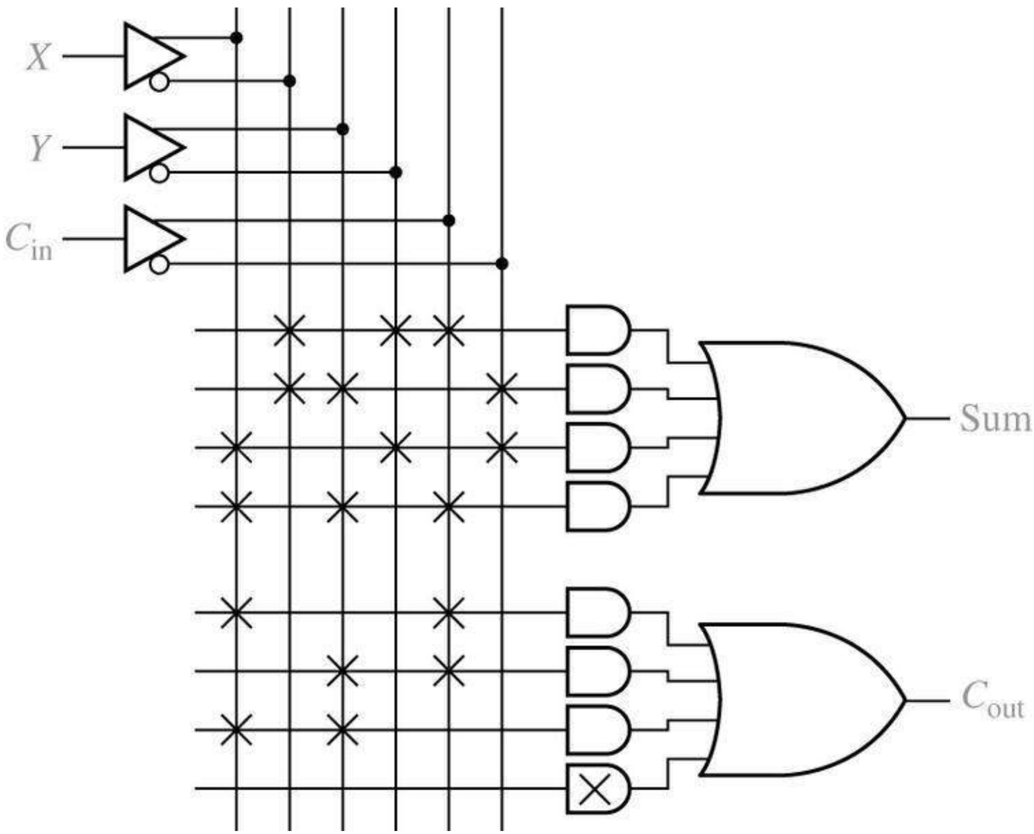


(a) Unprogrammed



(b) Programmed

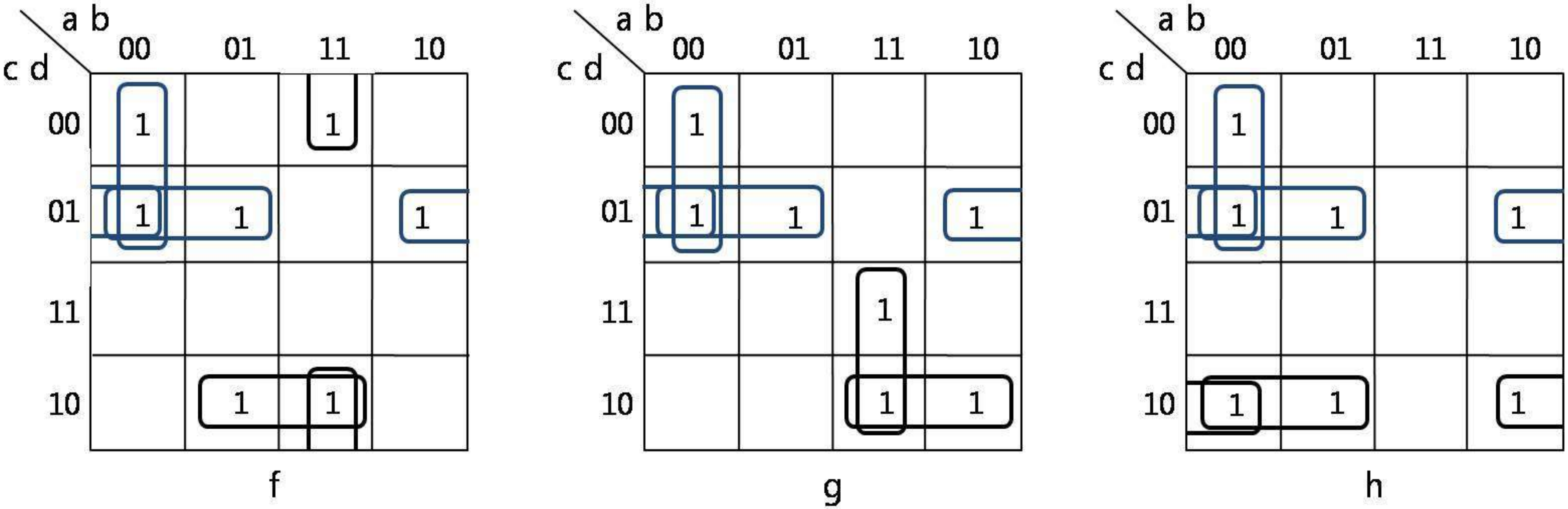
Implementation of a Full Adder Using a PAL



$$\text{Sum} = X'Y'C_{in} + X'YC'_{in} + XY'C'_{in} + XYC_{in}$$

$$C_{out} = XC_{in} + YC_{in} + XY$$

Example 4.11



- The three green terms are essential prime implicants of each function but other two are not.
  - f = a’b’c’ + a’c’d + b’c’d + abd’ + bcd’
  - g = a’b’c’ + a’c’d + b’c’d + abc + acd’
  - h = a’b’c’ + a’c’d + b’c’d + a’cd’ + b’cd’

PAL solution