Combinational Logic

Chapter 4

Outline

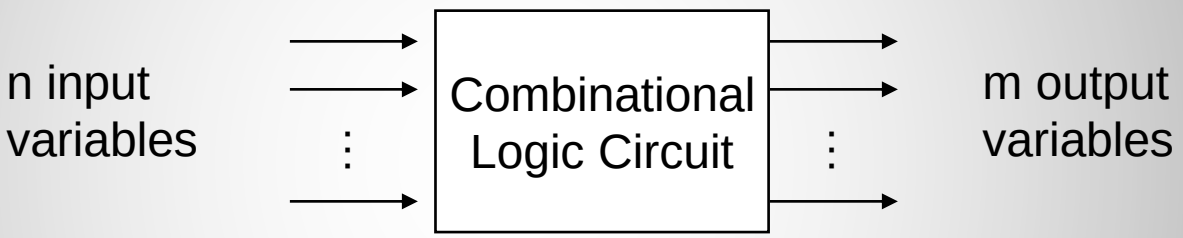
- Introducing combinational circuits
- Design and analysis of combinational circuits
- Iterative systems
  - Arithmetic circuits
  - Magnitude comparators
- Combinational building blocks
  - Decoders
  - Encoders
  - Multiplexers

Combinational Circuits vs Sequential Circuits

- Logic circuits for digital system
  - Combinational circuits
    - consist of logic gates
    - the outputs at any time are determined from only the present combination of inputs
  - Sequential circuits
    - contain memory elements and logic gates
    - the outputs are a function of the current inputs and the state of the memory elements
    - i.e., the outputs also depend on past inputs

Combinational Circuits

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



Analysis Procedure

- A combinational circuit
  - make sure that it is combinational not sequential
    - No feedback path

Analysis Example

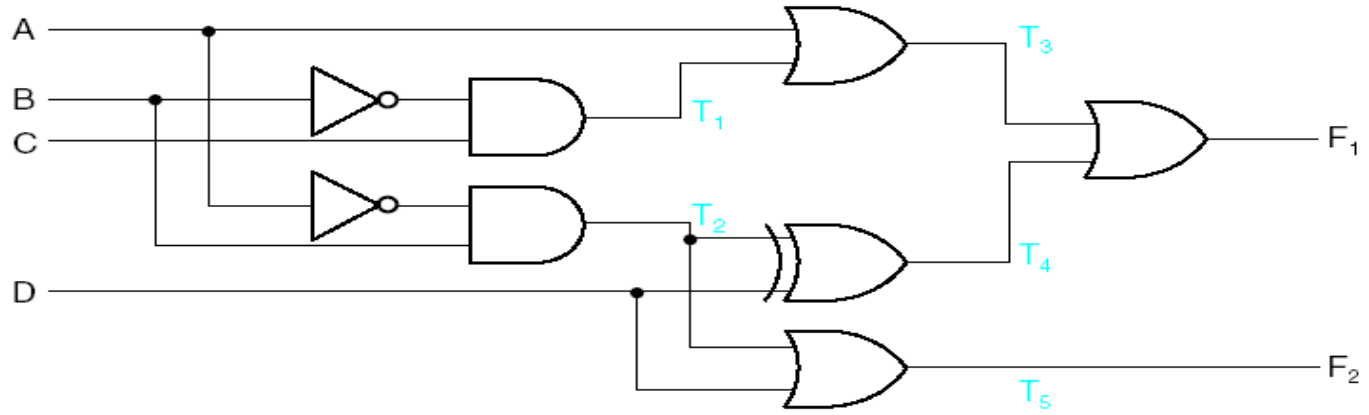


Fig. 3-5 Logic Diagram for Analysis Example

$T_1 = B'C \qquad T_2 = A'B$

$F_2 = T_2 + D = A'B + D$

$T_3 = A + T_1 = A + B'C$

$T_4 = T_2 \oplus D = (A'B) \oplus D = A'BD' + AD + B'D$

$F_1 = T_3 + T_4 = A + B'C + A'BD' + AD + B'D = A + B'C + BD' + B'D$

Design Example 1

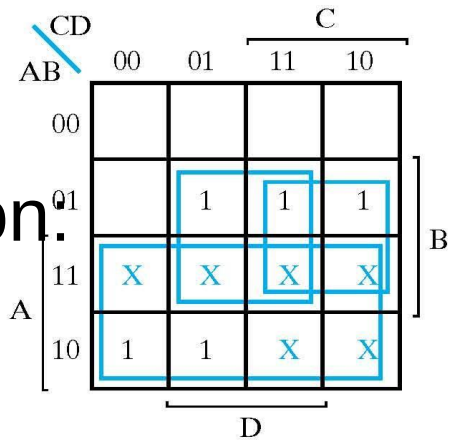
- Design a BCD-to-Excess-3 code converter (the *excess-3 code* of a decimal digit *p* is the binary code of *p*+*3*).

TABLE 3-1
Truth Table for Code Converter Example

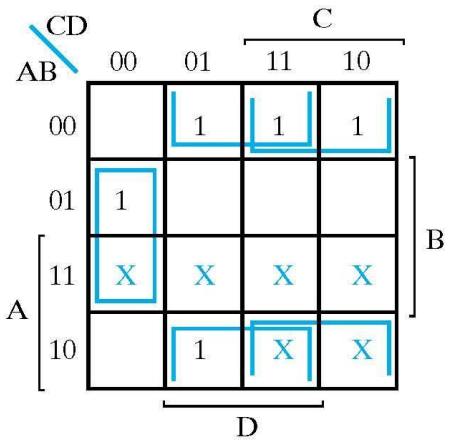
Decimal Digit	Input BCD				Output Excess-3			
	A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

Design Example 1 (cont'd)

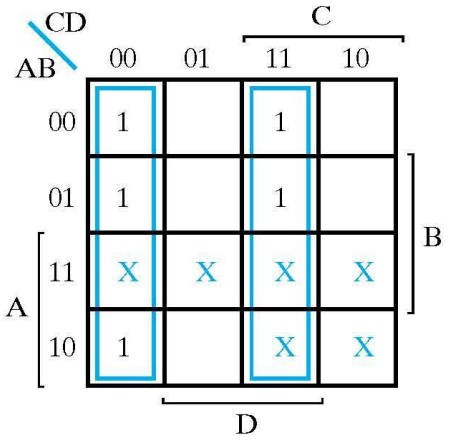
2-level simplification:



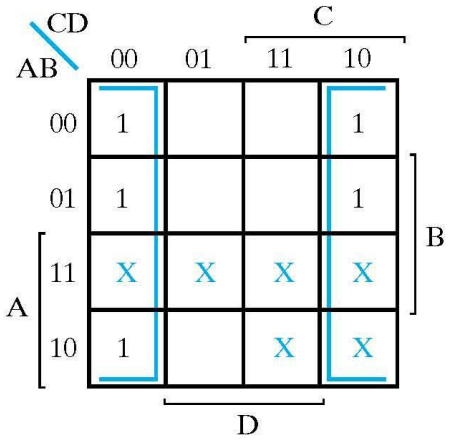
$W = A + BC + BD$



$X = \overline{B}C + \overline{B}D + B\overline{C}\overline{D}$



$Y = CD + \overline{C}\overline{D}$



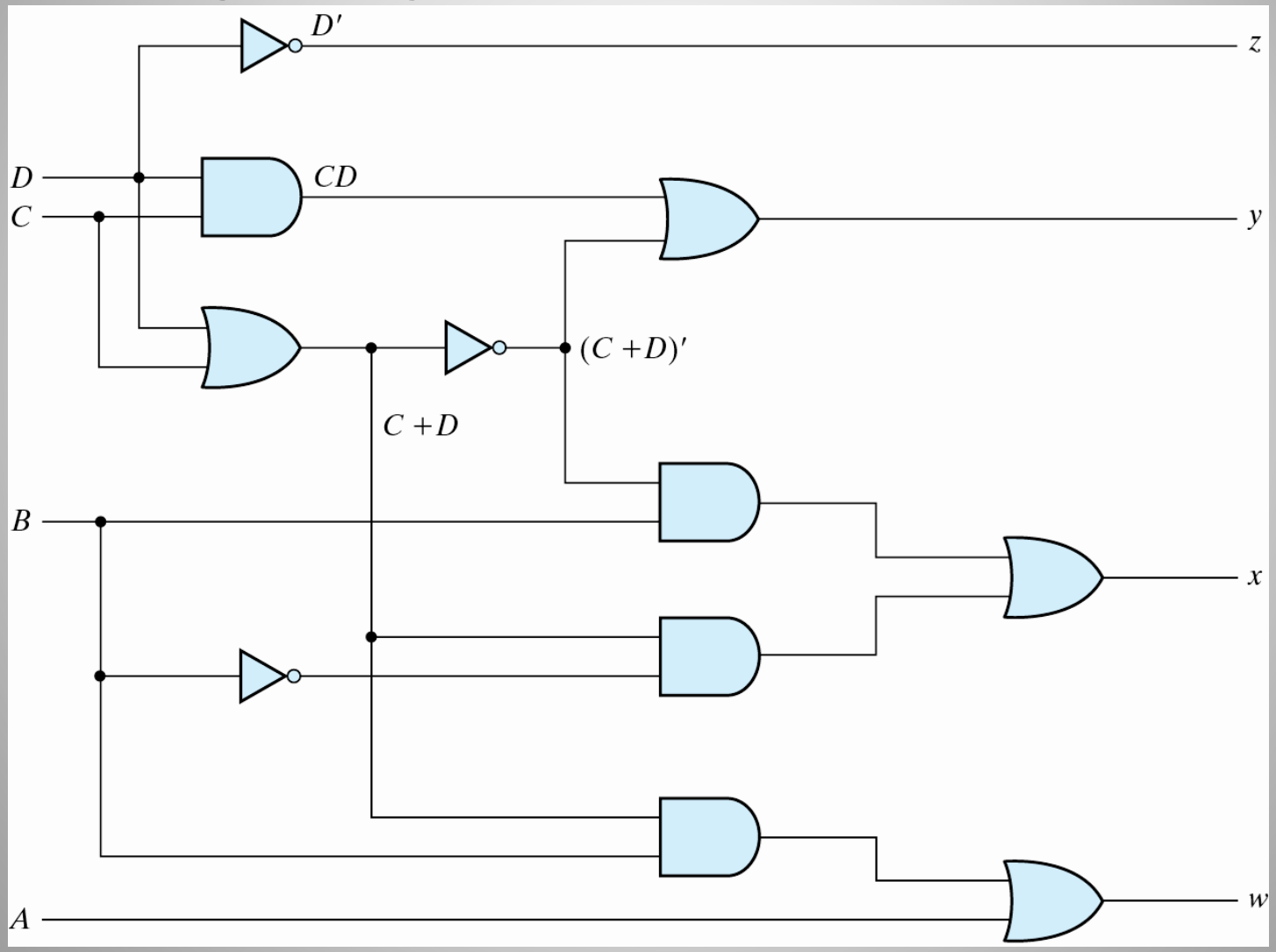
$Z = \overline{D}$

Design Example 1(cont'd)

- The simplified functions
  - z = D'
  - y = CD +C'D'
  - x = B'C + B'D+BC'D'
  - w = A+BC+BD
- Further simplification by extraction
  - T = C + D
  - z = D'
  - y = CD +C'D' = CD + (C+D)' = CD + T'
  - x = B'C + B'D+BC'D' = B'(C+D) +B(C+D)' = B'T + BT'
  - w = A+BC+BD = A + B(C+D) = A + BT

Design Example 1(cont'd)

- The logic diagram



Half Adder

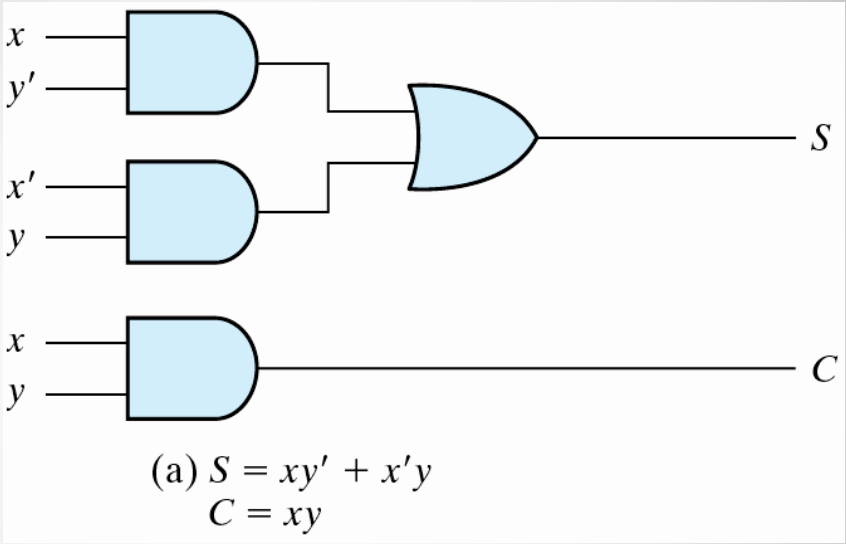
- A half adder performs addition of two bits
  - 2 input bits: x, y
  - 2 output bits: C (carry), S (sum)

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

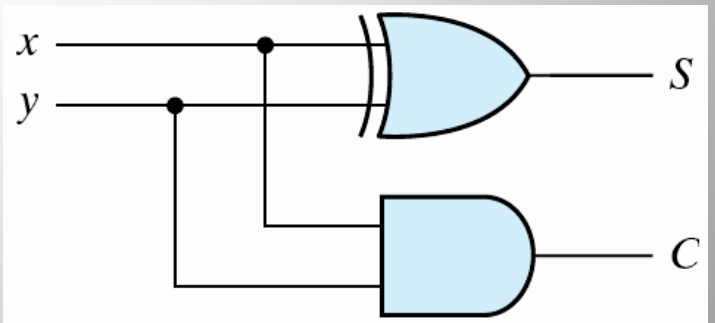
Half Adder

- Different implemenations
  - S = x'y+xy'
    C = xy



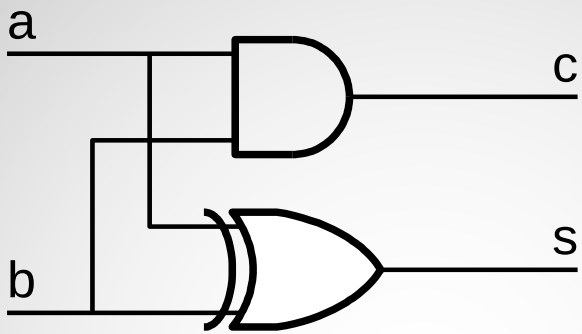
(a) $S = xy' + x'y$
$C = xy$

  - S = x'y+xy' = S=x⊕y
    C = xy



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

Verilog Description of Half Adder




```
// half adder
module HalfAdder(a,b,c,s);
  input a,b ;
  output c,s ;  // carry and sum
  wire s = a ^ b ;
  wire c = a & b ;
endmodule
```

Full Adder

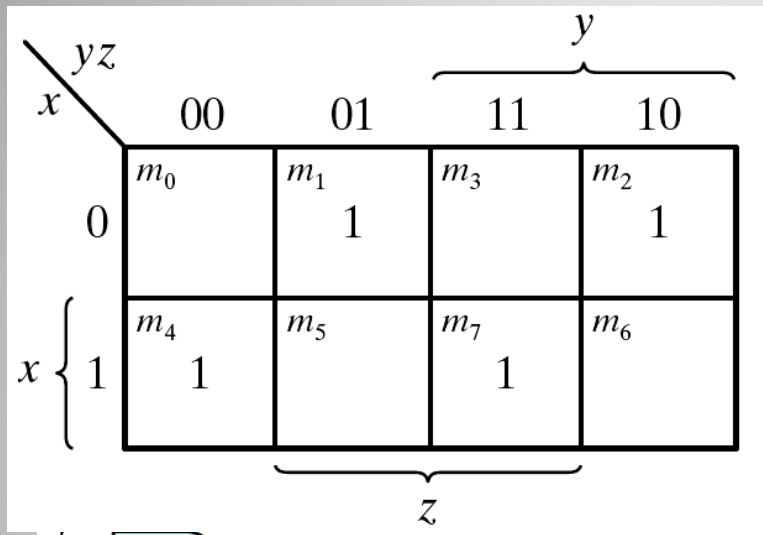
- A full adder is for adding three bits
  - 3 input bits
    - x, y: two significant bits
    - z: the carry bit from the previous lower significant bit
  - 2 output bits: C, S

Table 4.4
Full Adder

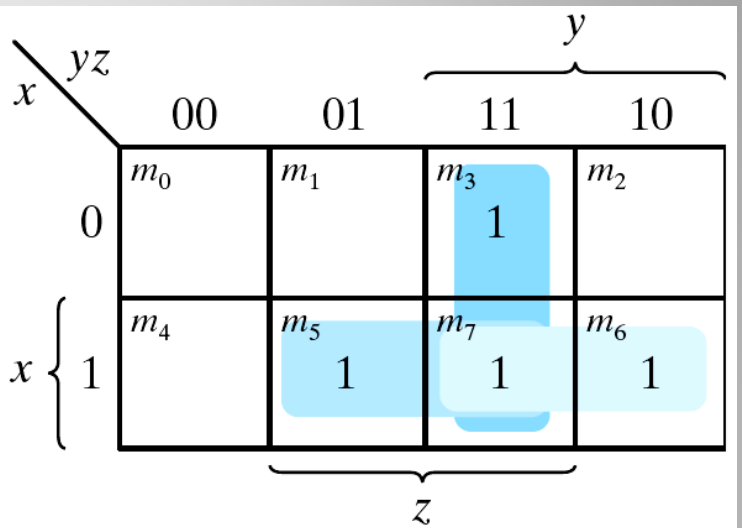
x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

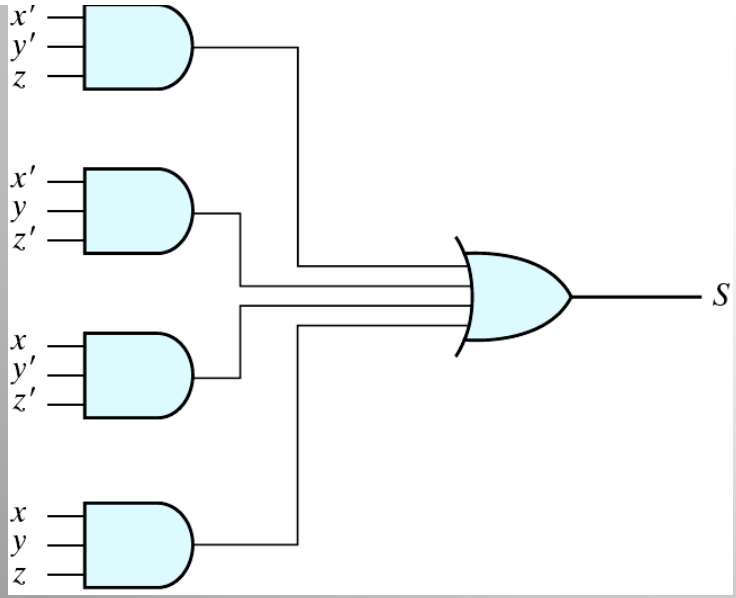
SOP Implementation of Full Adder

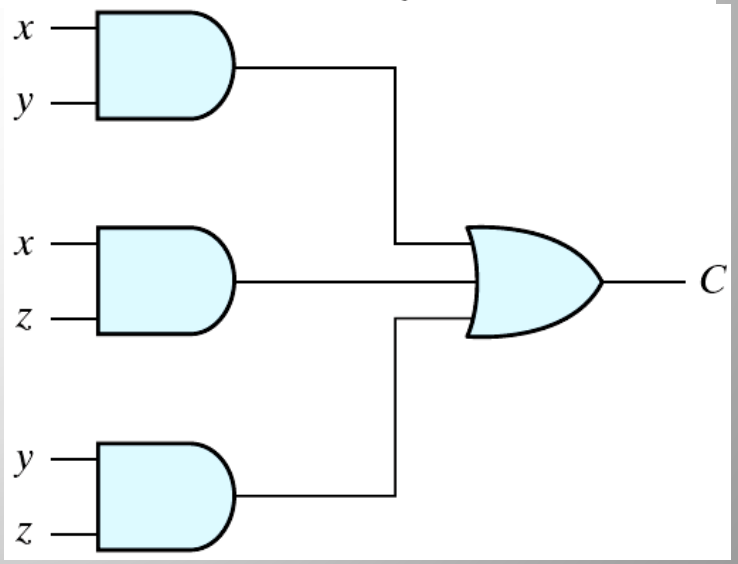
S:



C:







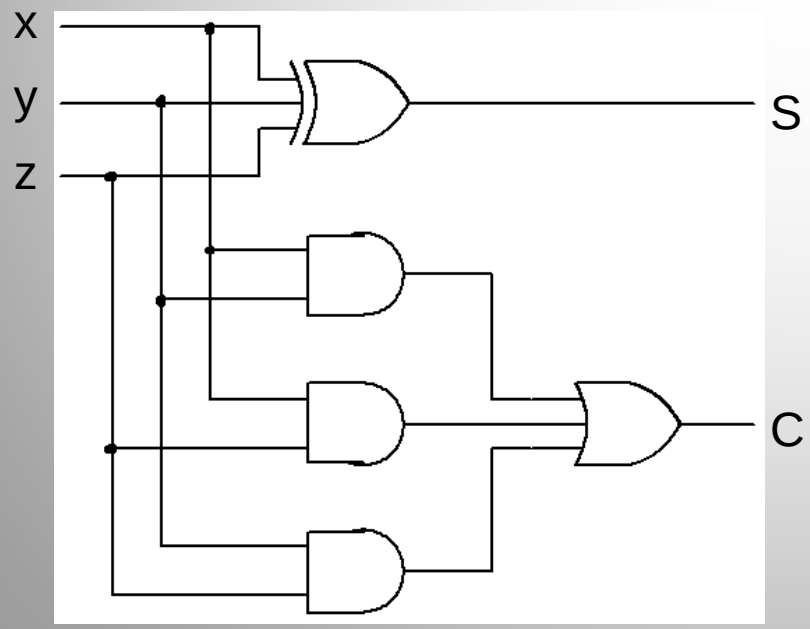
Alternative Implementation of Full Adder

S:

x \ yz	00	01	11	10
0	m_0	m_1 1	m_3	m_2 1
1	m_4 1	m_5	m_7 1	m_6

C:

x \ yz	00	01	11	10
0	m_0	m_1	m_3 1	m_2
1	m_4	m_5 1	m_7 1	m_6 1



C = xy + xz + xy

Implementation based on Half Adders

- A full adder can be implemented with two half adders and one OR gate
- S = x'y'z+x'yz'+ xy'z'+xyz = (x⊕y) ⊕z

  C = xy + xz + yz = xy + z(x⊕y)

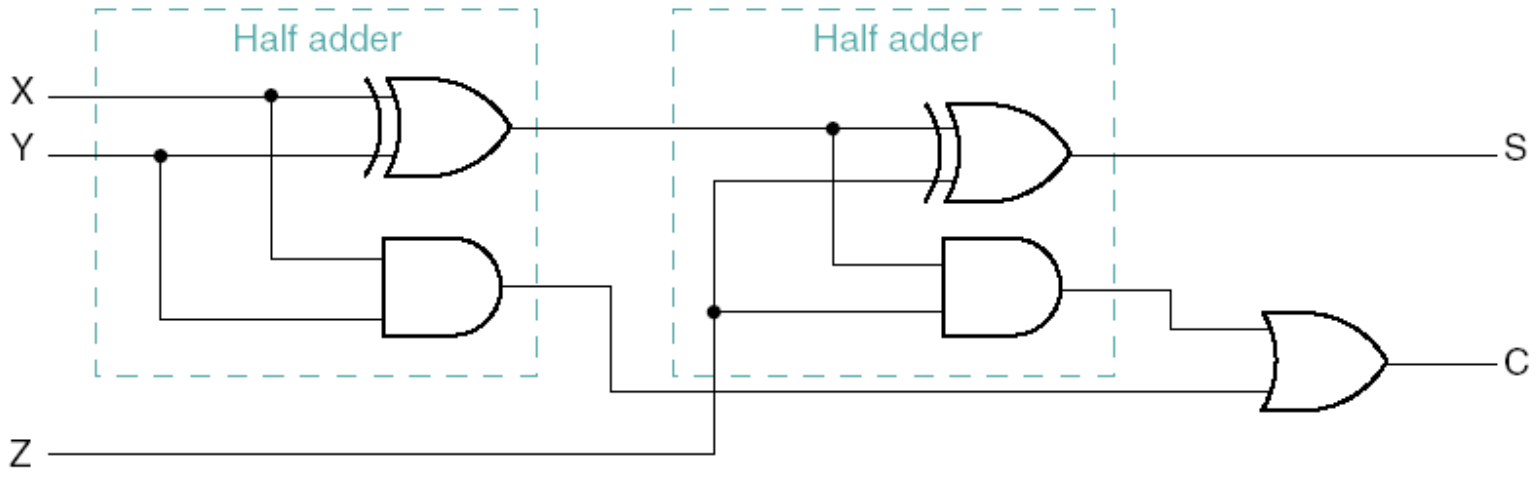


Fig. 3-27 Logic Diagram of Full Adder

Full Adder in Verilog

```
// full adder - logical
module FullAdder2(a, b, cin, cout, s);
  input  a, b, cin;
  output cout, s;
  wire   s = a ^ b ^ cin;
  wire   cout = (a & b) | (a & cin) | (b & cin); // majority
endmodule
```

```
// full adder - from half adders
module FullAdder1(a, b, cin, cout, s);
  input a, b, cin;
  output cout, s;  // carry and sum
  wire g, p;          // generate and propagate
  wire cp;
  HalfAdder ha1(a, b, g, p);
  HalfAdder ha2(cin, p, cp, s);
  assign cout = g | cp;
endmodule
```

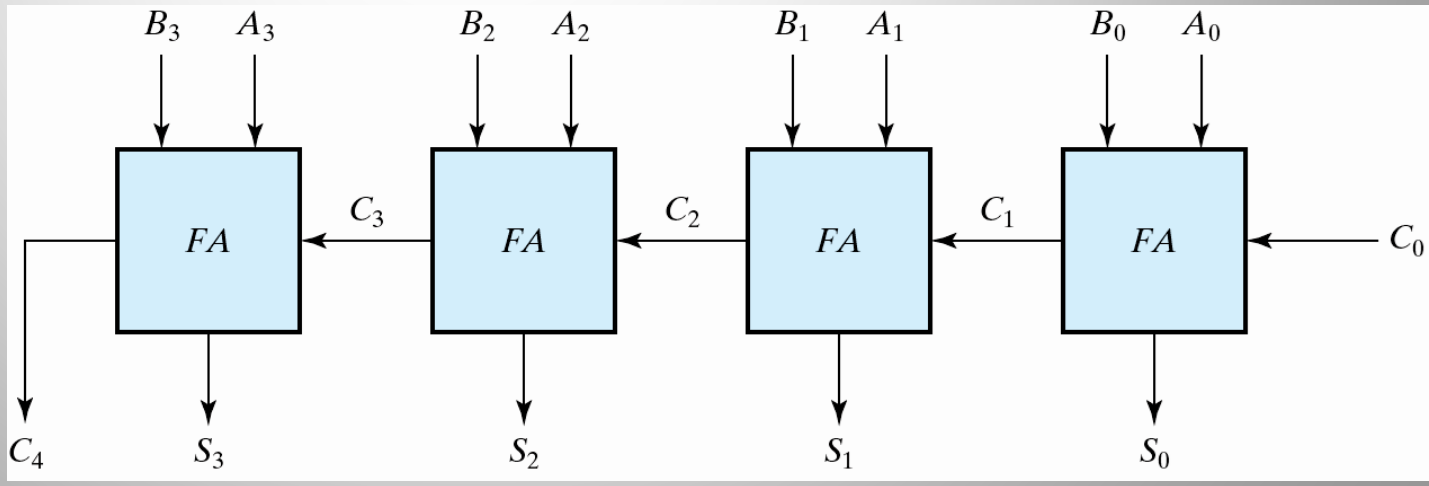
Lesson:

Even very simple functions, like a full adder, can be implemented in many different ways.

Ripple Carry Adder

- Add two *n*-bit numbers
- Like addition by hand, progress from least-significant digit to most-significant digit.
- If a carry is produced in position *i*, it is added to the operands in position *i+1*.
- A ripple carry adder is formed by cascading *n* full adders.

e.g.



Signed Numbers in 2's Complement

- Recall that in 2's complement scheme
  - $-X$ is obtained by taking the 2's complement of X
  - E.g. $X = 001111 \Rightarrow -X = 110001$

2's Complement Signed Binary Arithmetic

- Addition
  - Add the nos. (including the sign bits). Discard any carry out of the sign bit position.

	+7	00000111
+	+10	00001010
	+17	00010001

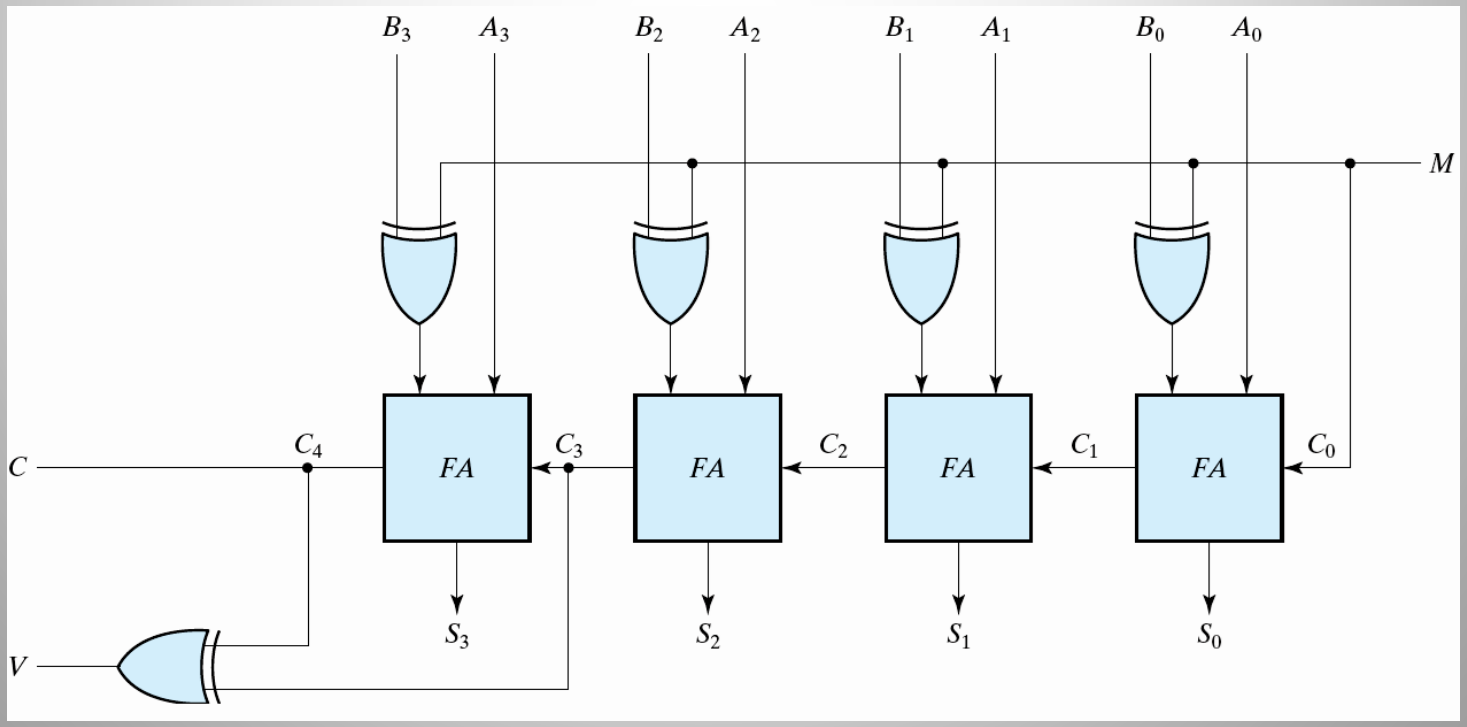
	+7	00000111
+	-10	11110110
	-3	11111101

	-7	11111001
+	+10	00001010
	+3	(1)00000011

	-7	11111001
+	-10	11110110
	-17	(1)11101111

2's Complement Signed Binary Arithmetic

- Subtraction
  - *A* - *B* = *A*+ (-*B*)
  - i.e., *A* - *B* = *A*+ (2's complement of *B*)
- 4-bit Adder-Subtractor
  - *M* =0, compute *A*+*B*; *M* =1, compute *A*+*B'*+1(= *A*-*B*)



Arithmetic Overflow

- Overflow
  - The storage is limited
  - Add two +ve numbers and obtain a -ve number
  - Add two -ve numbers and obtain a +ve number

	+70	01000110
+	+80	01010000
	+150	10010110

↑ Negative!!

	-70	10111010
+	-80	10110000
	-150	(1)01101010

↑ Positive!!

$$\text{overflow} = A_{n-1}B_{n-1}S'_{n-1} + A'_{n-1}B'_{n-1}S_{n-1}$$

Alternatively, $\text{overflow} = C_{n-1} \oplus C_n$

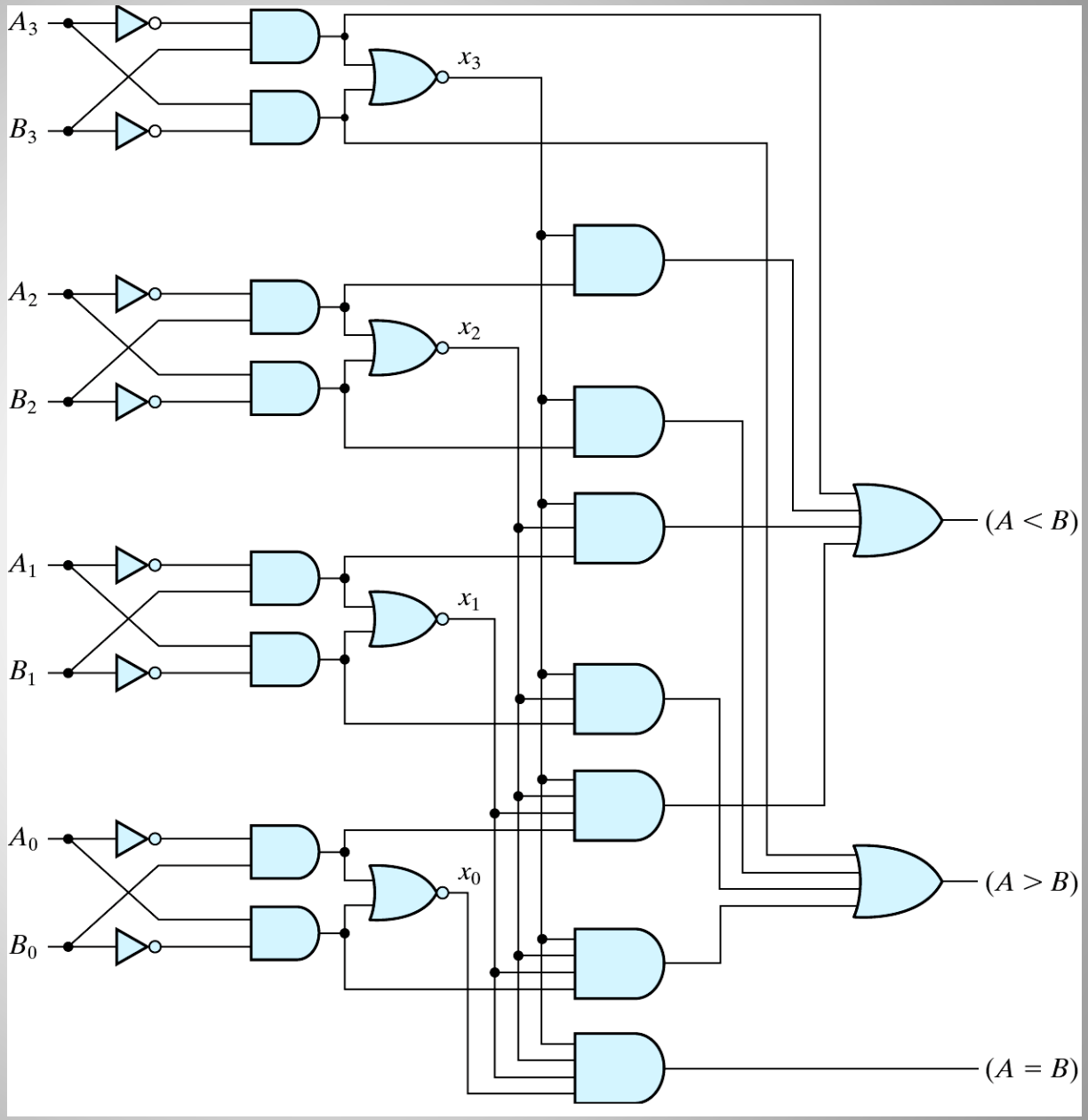
Magnitude Comparator

- Compare two numbers
  - outputs: A>B, A=B, A<B
- Design Approaches
  - truth table
    - 2^{2n} entries - too cumbersome for large n
  - use inherent regularity of the problem
    - reduce design efforts
    - reduce human errors

Magnitude Comparator

- 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: $x_i= A_iB_i+A_i'B_i'$
    - (A=B) iff $x_3x_2x_1x_0=1$
  - (A>B) iff $A_3B_3'+x_3A_2B_2'+x_3x_2A_1B_1'+x_3x_2x_1\,A_0B_0'$
  - (A<B) iff $A_3'B_3+x_3A_2'B_2+x_3x_2A_1'B_1+x_3x_2x_1\,A_0'B_0$
- Reduce gate count by noting
  - $x_i = A_iB_i+A_i'B_i' = (A_iB_i'+A_i'B_i)'$

Logic diagram of a 4-bit magnitude comparator

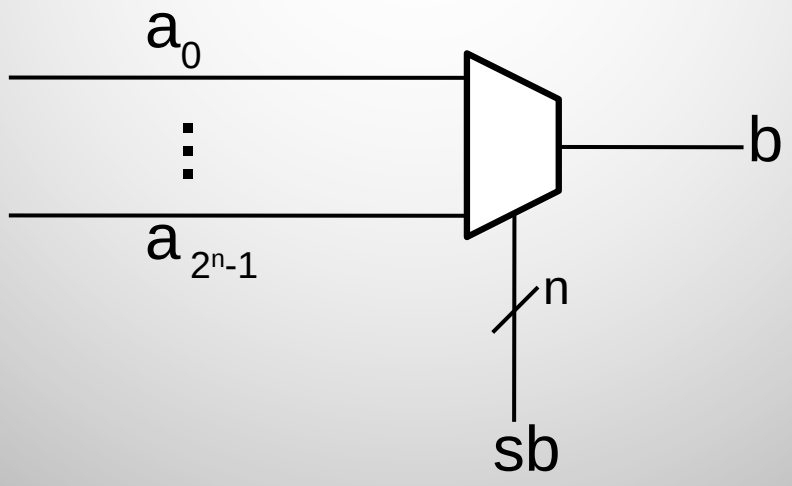


Lesson:

From algorithm to logic design e.g. adders, multipliers, comparators.

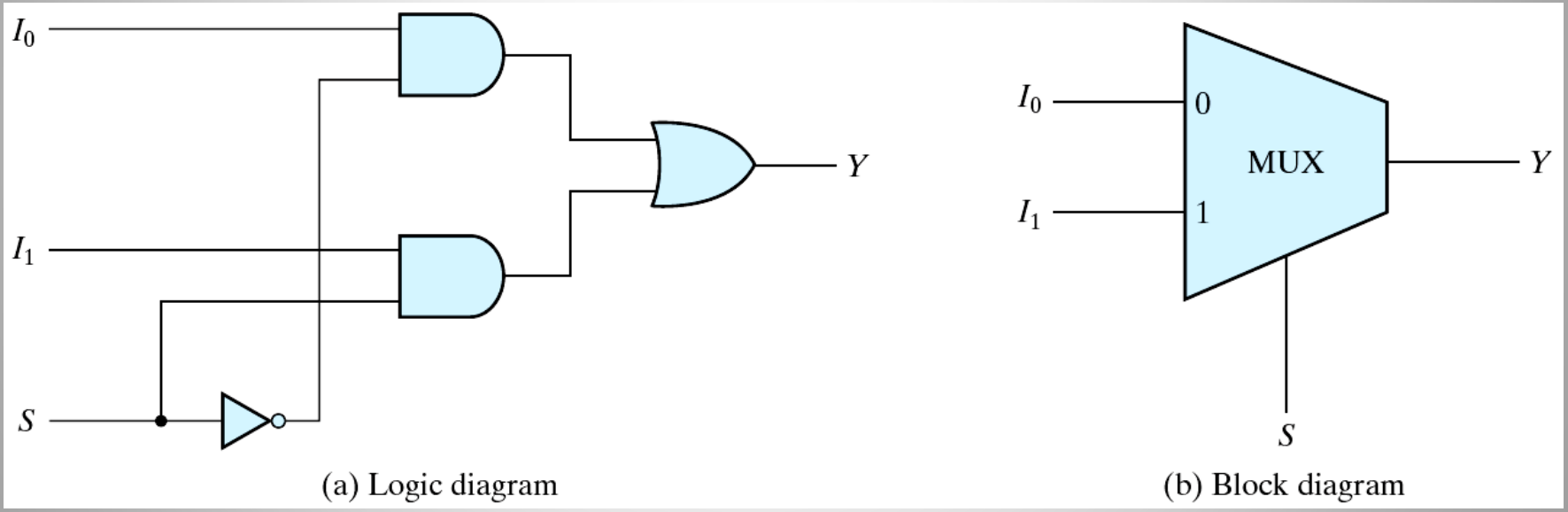
Multiplexer

- A data selector
- 2^n data input lines, n selection lines and one output line
- b = a[i] if sb = i



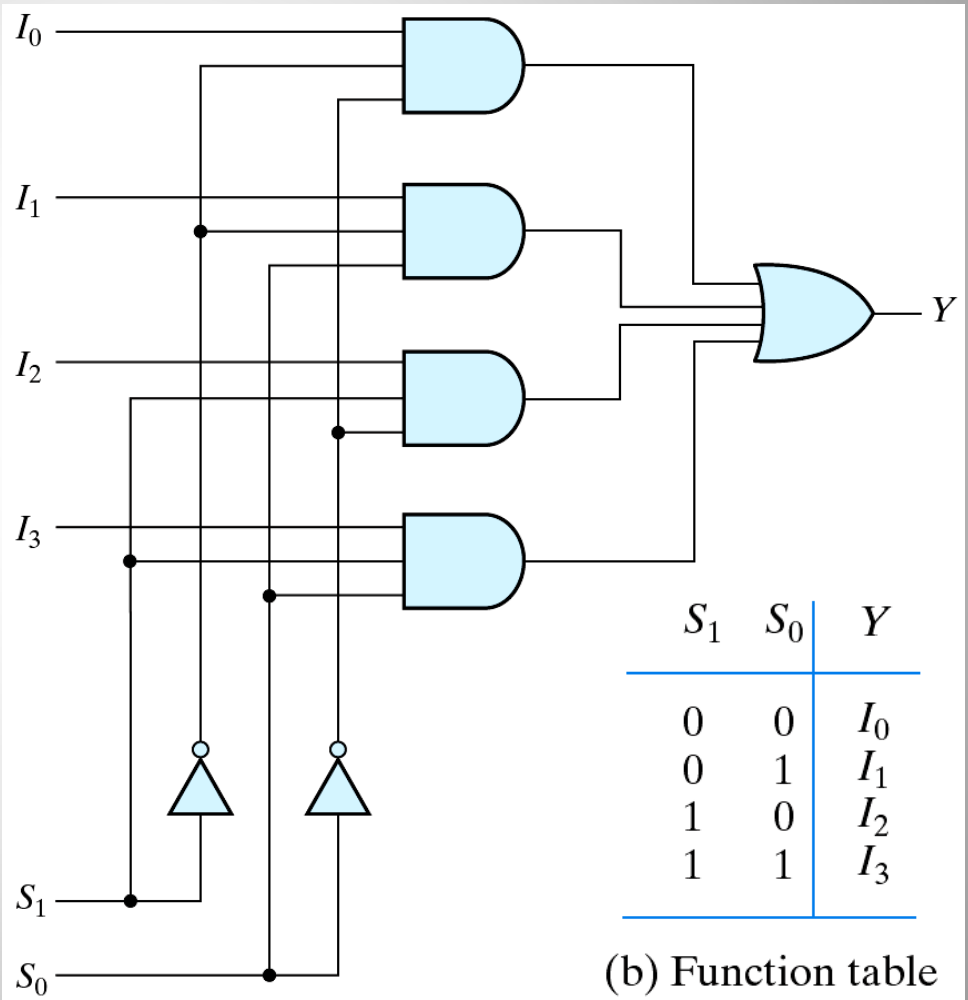
Multiplexer

- e.g. 2-to-1 multiplexer



Multiplexer

- e.g. 4-to-1 multiplexer



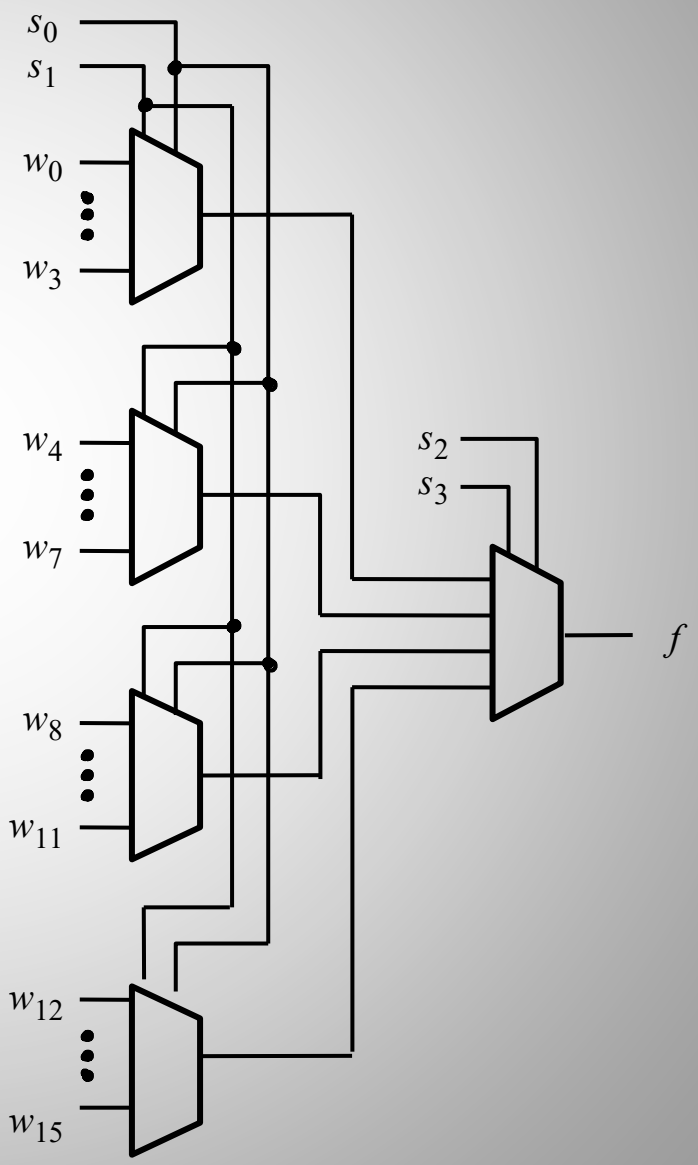
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b) Function table

(a) Logic diagram

Mux Tree

- e.g. A 16-to-1 mux



Verilog Description of a 4-to-1 MUX Using case

```
module Mux4to1(a3, a2, a1, a0, sb, b);
   input  a0, a1, a2, a3;  // inputs
   input  [1:0]     sb;           // binary select
   output b;
   reg    b;

   always @(*) begin
      case(sb)
        2'b00: b = a0;
        2'b01: b = a1;
        2'b10: b = a2;
        2'b11: b = a3;
        default: b = x;
      endcase
  end
endmodule
```

Using MUX to Implement Boolean Function

- A MUX is a data selector, for a 4x1 MUX:

  $Y = S'_1S'_0I_0 + S'_1S_0I_1 + S_1S'_0I_2 + S_1S_0I_3$

- In general, for a 2^nx1 MUX:

  $Y = \sum m_kI_k$

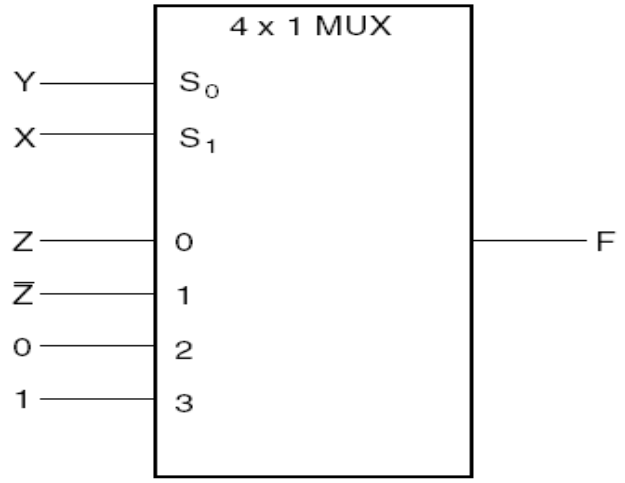
- Observation: Any function of *n* variables can be implemented with a single MUX with *n*-1 selection inputs.

Example 1. F = X'Y'Z + X'YZ' + XY

Suppose we use X and Y as the selection inputs S_1 and S_0, resp., then

X	Y	Z	F	
0	0	0	0	F = Z
0	0	1	1	
0	1	0	1	F = $\overline{Z}$
0	1	1	0	
1	0	0	0	F = 0
1	0	1	0	
1	1	0	1	F = 1
1	1	1	1	

(a) Truth table



(b) Multiplexer implementation

Fig. 3-22 Implementing a Boolean Function with a Multiplexer

Q. How to implement F using Y, Z as the selection inputs?

Example 2. F(A,B,C,D) = ∑ m(1,3,4,11,12,13,14,15)

Using A, B, and C as the selection inputs S_2, S_1, S_0:

A	B	C	D	F	
0	0	0	0	0	F = D
0	0	0	1	1	
0	0	1	0	0	F = D
0	0	1	1	1	
0	1	0	0	1	$F = \overline{D}$
0	1	0	1	0	
0	1	1	0	0	F = 0
0	1	1	1	0	
1	0	0	0	0	F = 0
1	0	0	1	0	
1	0	1	0	0	F = D
1	0	1	1	1	
1	1	0	0	1	F = 1
1	1	0	1	1	
1	1	1	0	1	F = 1
1	1	1	1	1	

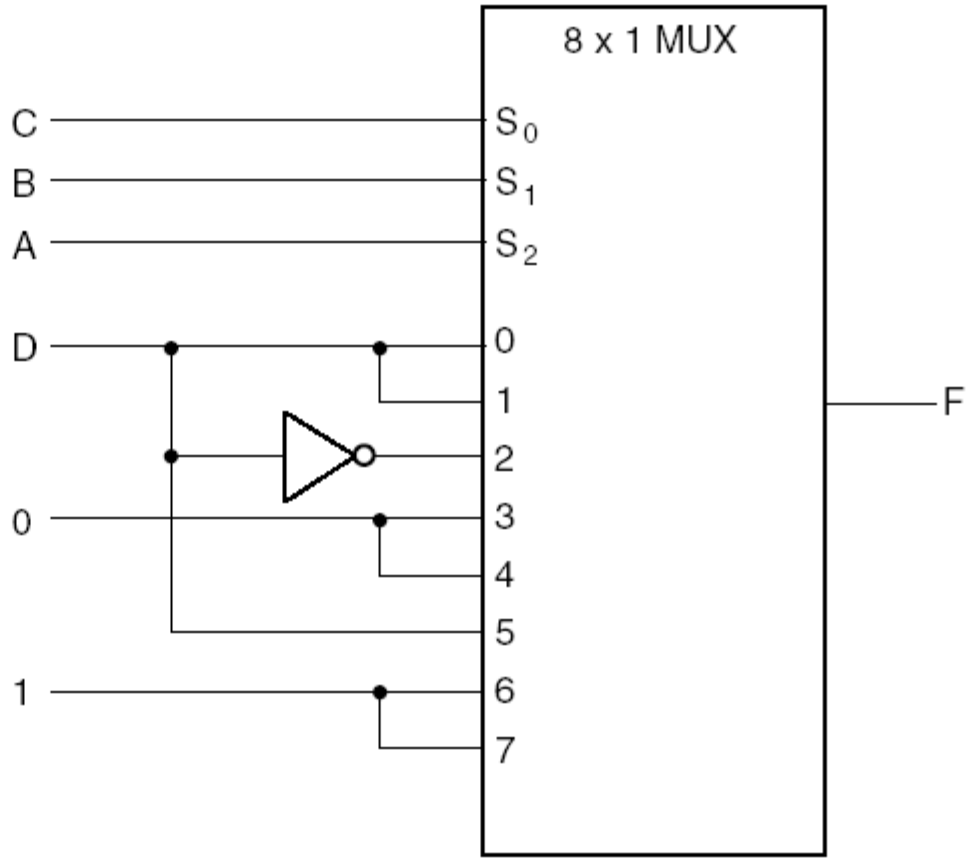
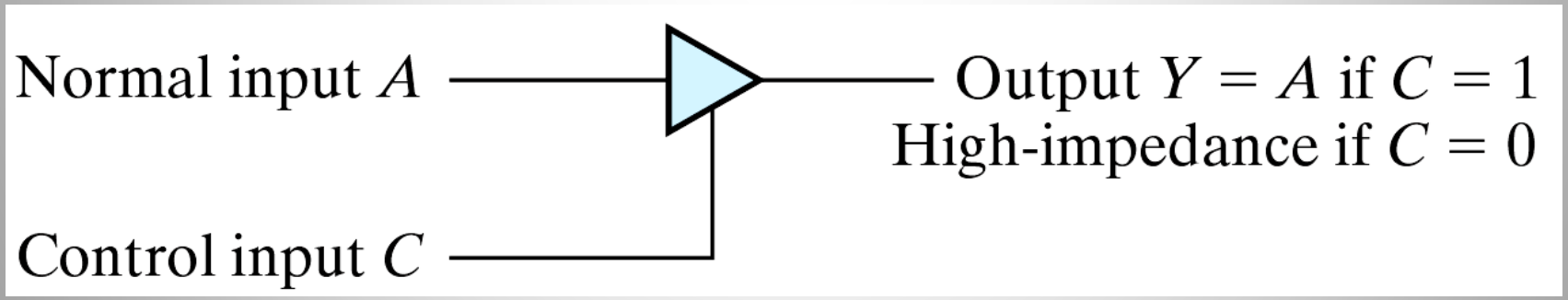


Fig. 3-23 Implementing a Four-Input Function with a Multiplexer

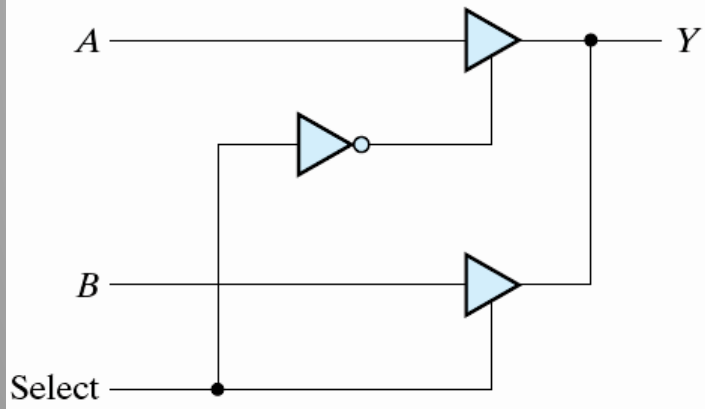
Tri-state Buffer

- A type of buffer
- Output state: 0, 1, and high-impedance (open circuit)

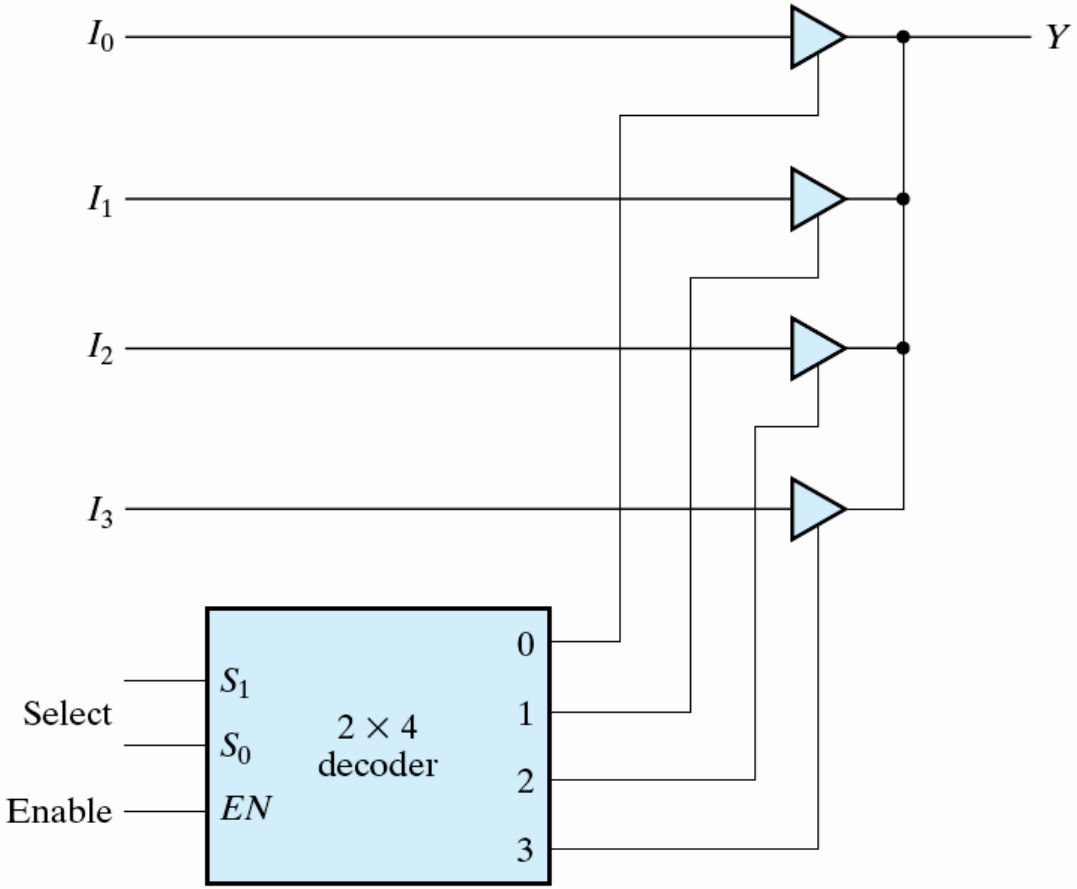


Example with Tri-State Buffer

- Muxes with tri-state buffers



(a) 2-to-1-line mux



(b) 4-to-1-line mux

Verilog Description of a Binary Adder

```
module binary_adder(a, b, c_in, c_out, sum);
  parameter k = 4;
  input  [k - 1:0] a, b;
  input  c_in;
  output c_out;
  output [k - 1:0] sum;

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

Verilog Description of a Magnitude Comparator

```
module magnitude_comparator(a, b, a_lt_b, a_eq_b, a_gt_b);
  parameter k = 4;
  input  [k - 1:0] a, b;
  output a_lt_b, a_eq_b, a_gt_b;

  assign a_lt_b = (a < b);
  assign a_eq_b = (a == b);
  assign a_gt_b = (a > b);
endmodule
```

附錄

Decimal Adder

- Each stage add two BCD digits
  - 9 inputs: two BCD digits and one carry-in
  - 5 outputs: one BCD digit and one carry-out
- Design approaches
  - A truth table with 2^9 entries
  - Use 4-bit binary adders
    - the sum ≤ 9 + 9 + 1 = 19
    - Need binary to BCD conversion

Convert binary sum to a BCD digit and a carry-out

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

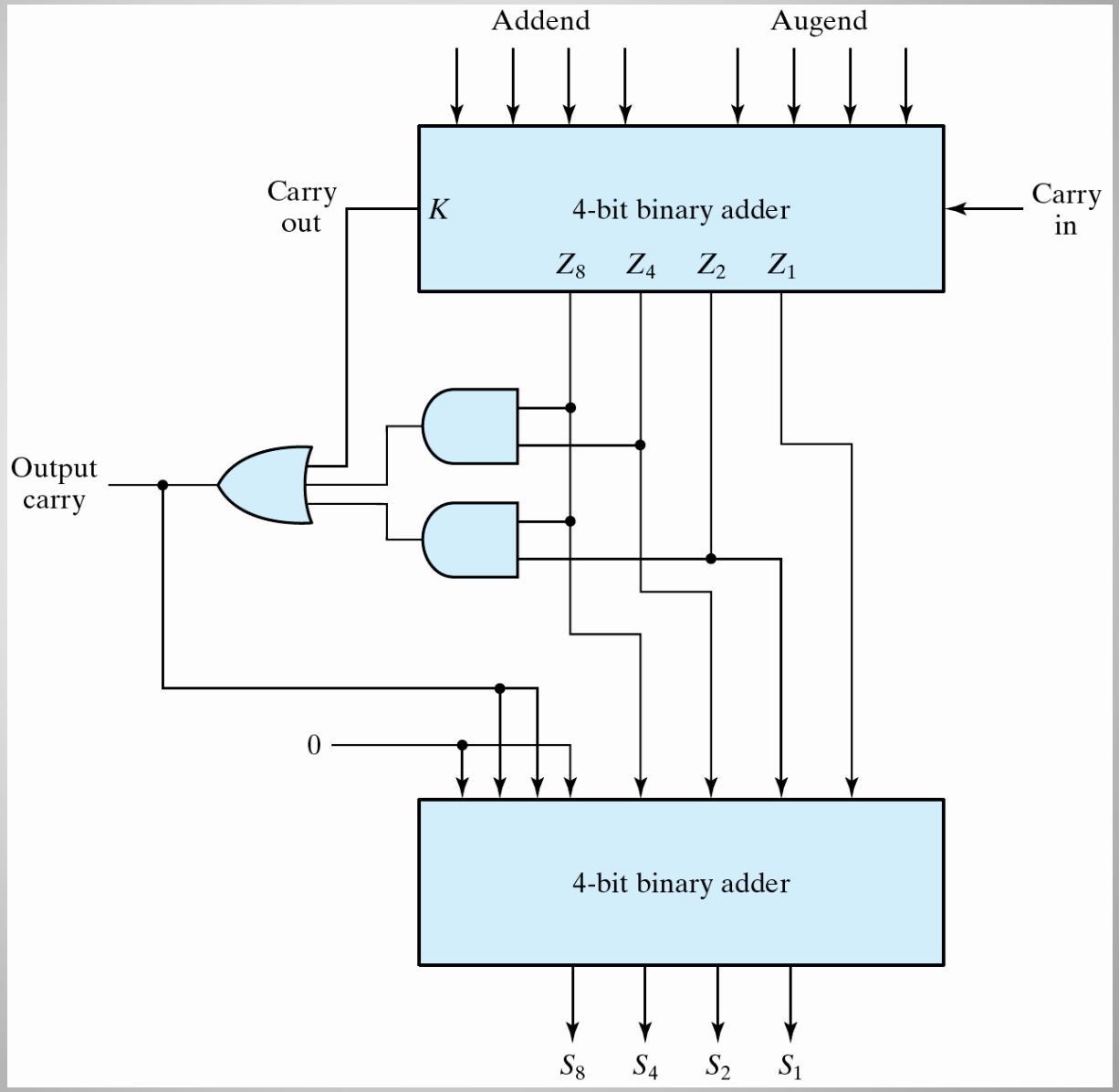
Decimal Adder

- Modifications are needed if result > 9
- result > 9 $\Leftrightarrow$ K = 1 or Z_8Z_4 = 1 or Z_8Z_2 = 1

$C = K + Z_8Z_4 + Z_8Z_2$

Add 6 to the binary sum to convert to the correct BCD representation when C = 1

Block diagram for one stage of BCD addition

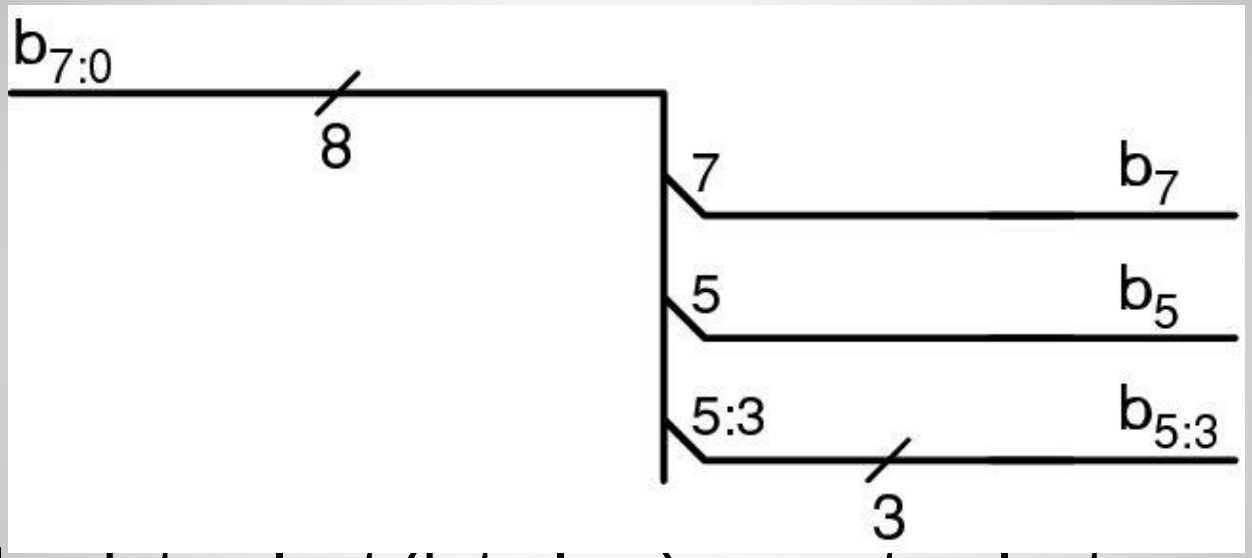


Combinational Building Blocks

- A few types of logic circuit are often used as building blocks in larger design
  - Decoder (binary to one-hot)
  - Encoder (one-hot to binary)
  - Multiplexer (select one of *N*)

Multi-Bit Notation

- Multi-bit signal or a bus



- Verilog bit-select (bit-slice) or part-select
  - `b[7:0]`
  - `b[7]`
  - `b[5:3]`

One-Hot Representation

- Represent a set of *N* elements with *N* bits
- Exactly one bit is set
- Example – to encode numbers 0-7

Binary	One-hot
000	00000001
001	00000010
010	00000100
…	…
110	01000000
111	10000000

- What operations are simpler with one-hot representation? With binary?

Decoder

- A n-to-m decoder
  - n input variables; up to 2^n output lines
  - binary to one-hot decoder
    - only one output can be high at any time
    - each output corresponds to one valuation of inputs

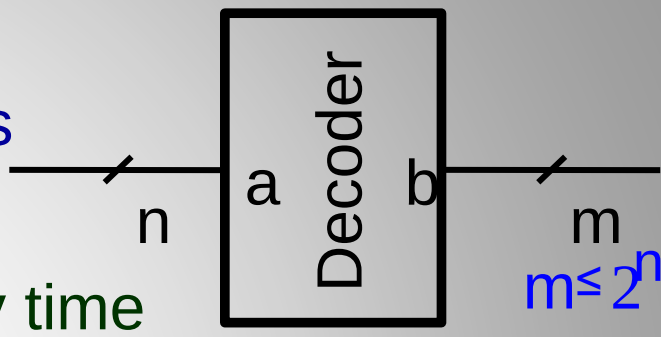


Table 4.6
Truth Table of a Three-to-Eight-Line Decoder

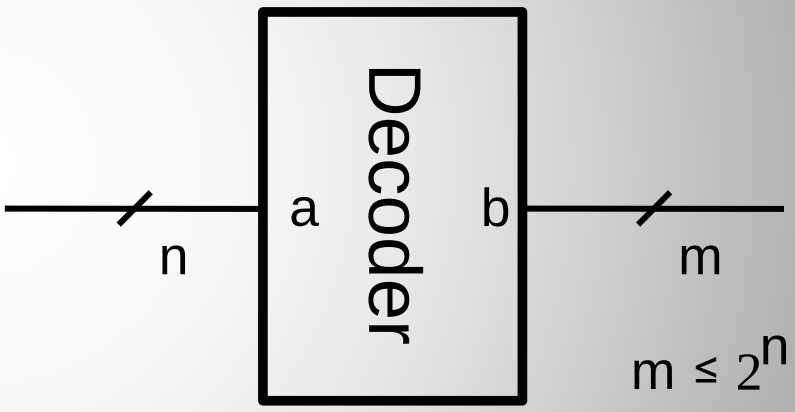
Inputs			Outputs							
x	*y*	*z*	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Verilog Description of Decoder

```
// a - binary input   (n bits wide)
// b - one hot output (m bits wide)
module Dec(a, b);
  parameter n = 2;
  parameter m = 4;

  input  [n - 1:0] a;
  output [m - 1:0] b;

  wire [m - 1:0] b = 1 << a;
endmodule
```



Parameterized Module

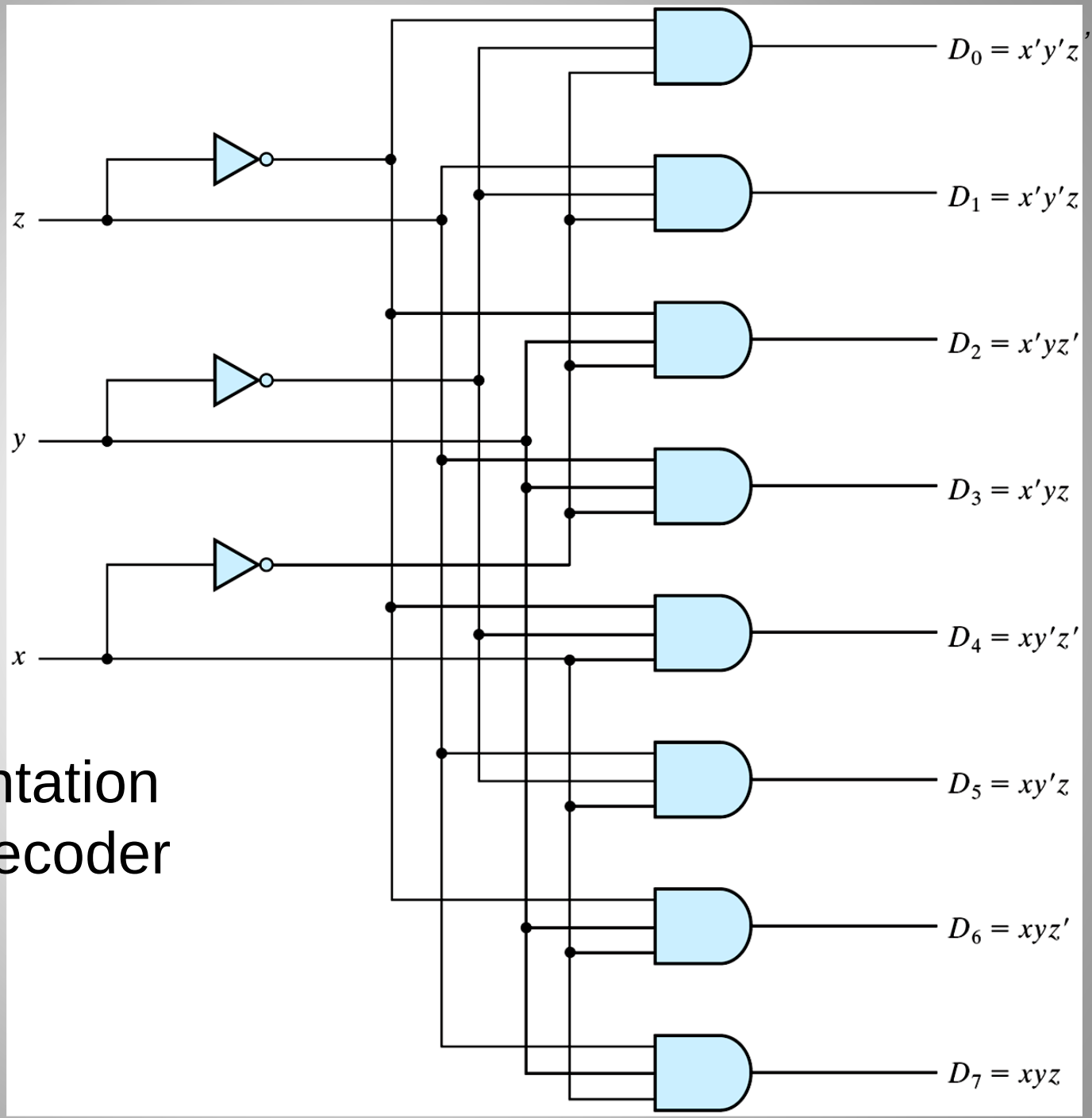
- Instantiate the decoder with default parameters

```
Dec dec24(a, b);
```

- Instantiate the decoder overriding the default parameters

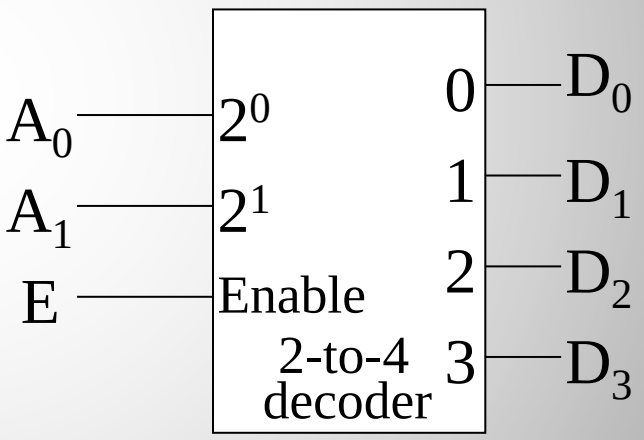
```
Dec #(3, 8) dec38(a, b);
Dec #(.n(4), .m(10)) dec410(a, b);
```

An implementation
of a 3-to-8 decoder



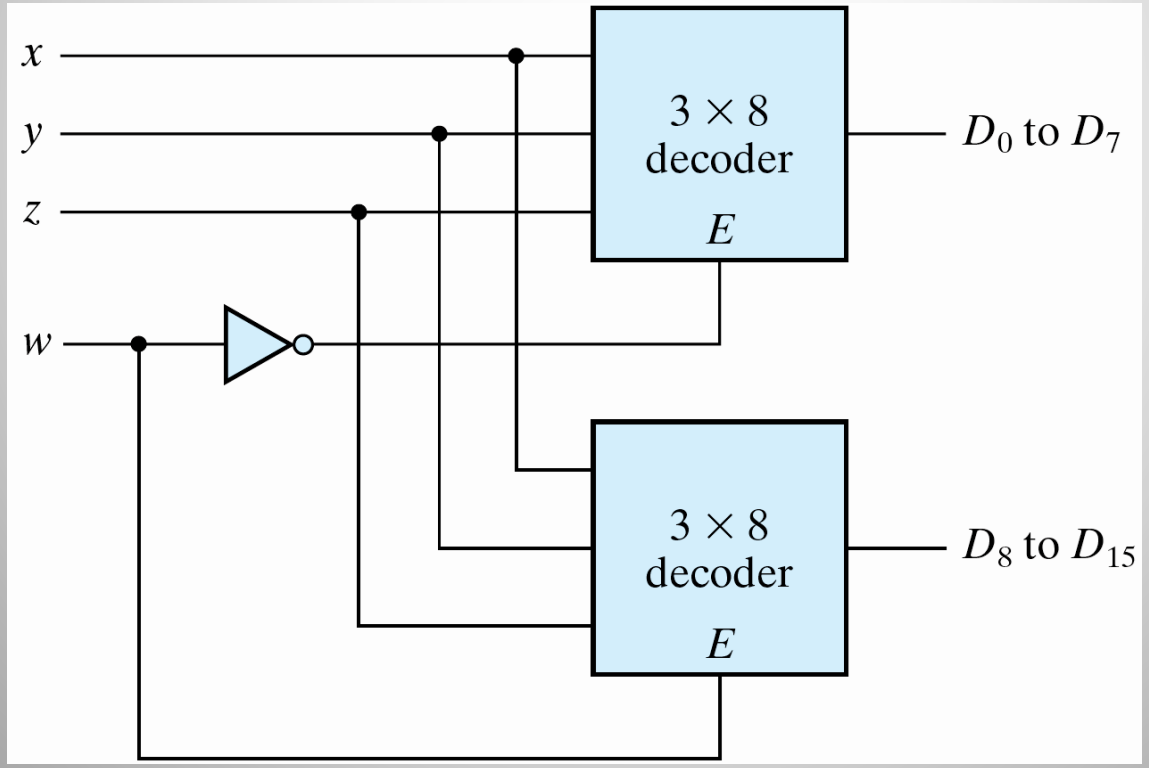
Decoder with Enable Input

inputs			outputs			
E	A_1	A_0	D_3	D_2	D_1	D_0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0
0	*	*	0	0	0	0



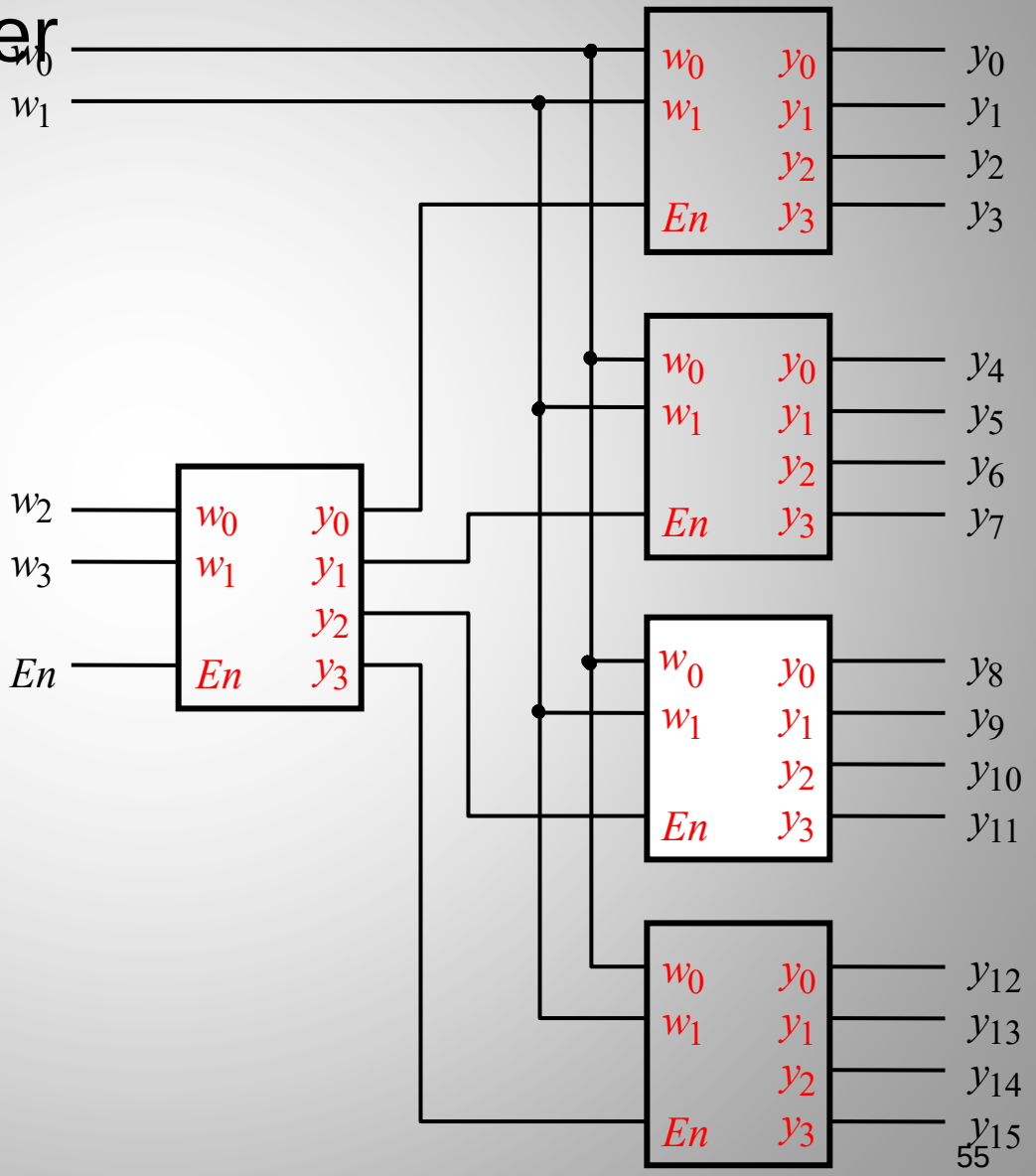
Larger Decoder from Small Ones

- Expansion
  - e.g. A 4-to-16 decoder from two 3-to-8 decoders



Decoder Tree

- A 4-to-16 decoder with enable



Using Decoder to Implement Boolean Function

- Each output of decoder = a minterm
- May use a decoder and an external OR gate to implement any Boolean function of *n* input variables
- e.g. Implement a full adder

TABLE 3-8
Truth Table of Full Adder

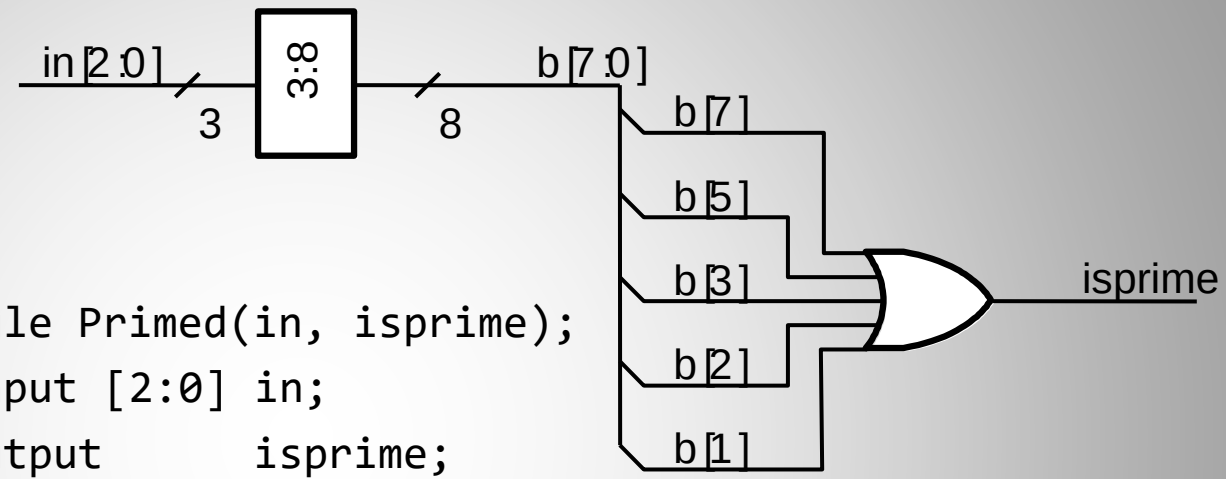
Inputs			Outputs	
X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 3-8 Truth Table of Full Adder

Example: Using Decoder to Implement Prime Number Detector

$f = \Sigma m(1, 2, 3, 5, 7)$

in[2:0]	isprime
0 0 0	0
0 0 1	1
0 1 0	1
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	0
1 1 1	1



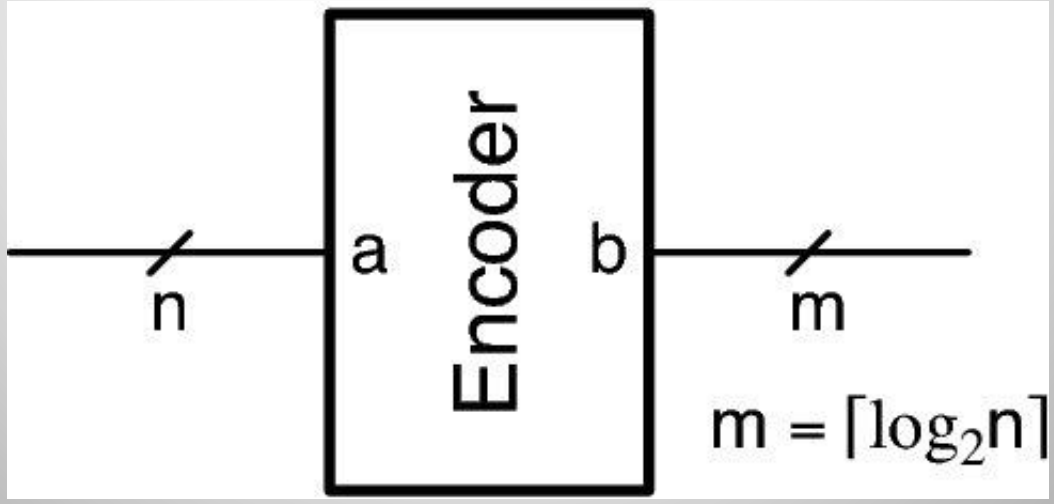

```
module Primed(in, isprime);
  input [2:0] in;
  output      isprime;
  wire  [7:0] b;

  // compute the output as the OR of
  // the required minterms
  wire isprime = b[1] | b[2] | b[3] | b[5] | b[7];

  // instantiate a 3->8 decoder
  Dec #(.n(3), .m(8)) d(.a(in), .b(b));
endmodule
```

Encoder

- An encoder provides the inverse function of a decoder
- Encoder converts an *n*-bit one-hot input signal to an *m*-bit binary-encoded output signal



Encoder

- E.g. 8-to-3 Encoder

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.

Priority Encoder

- Allows multiple input lines to be asserted
- Higher-numbered input line has higher priority
- e.g. 4-to-2-line priority encoder

TABLE 3-6
Truth Table of Priority Encoder

Inputs				Outputs		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

Table 3-6 Truth Table of Priority Encoder

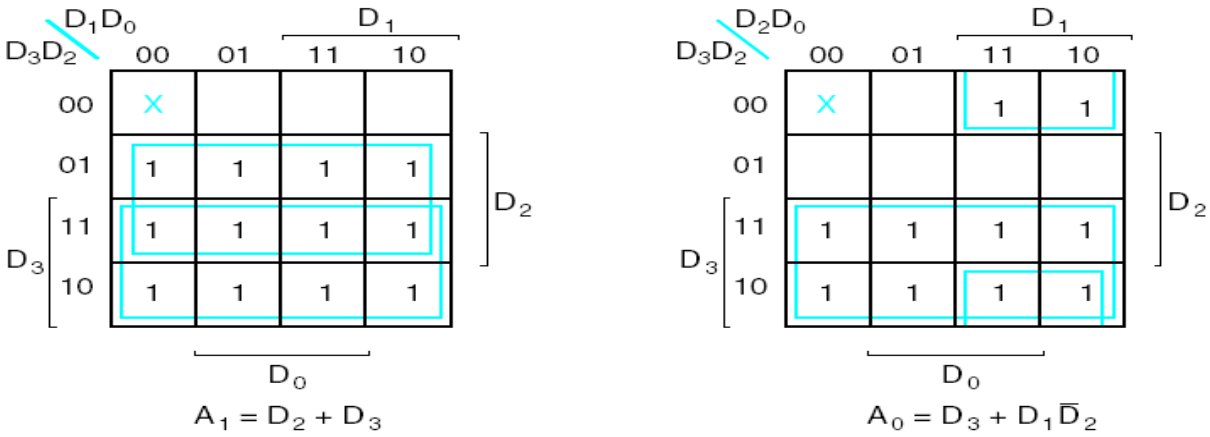


Fig. 3-17 Maps for Priority Encoder

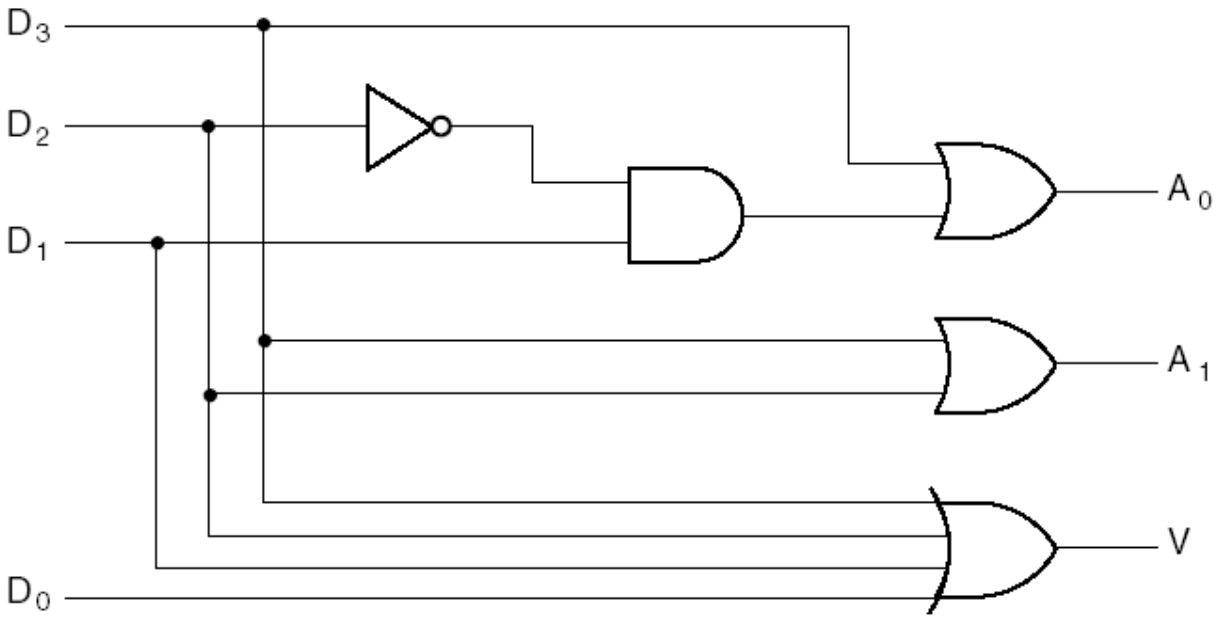


Fig. 3-18 Logic Diagram of a 4-Input Priority Encoder