

Chapter 4 Combinational Logic

- Logic circuits for digital systems may be **combinational** or **sequential**.
- A combinational circuit consists of input variables, logic gates, and output variables.

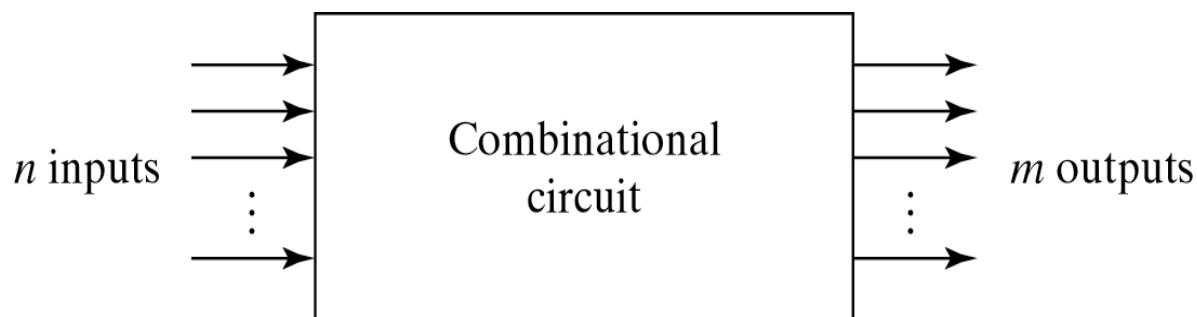
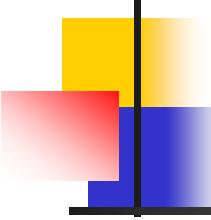
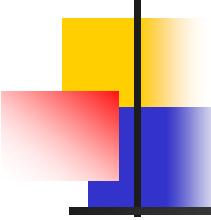


Fig. 4-1 Block Diagram of Combinational Circuit



4-2. Analysis procedure

- To obtain the output Boolean functions from a logic diagram, proceed as follows:
 1. Label all gate outputs that are a function of input variables with arbitrary symbols. Determine the Boolean functions for each gate output.
 2. Label the gates that are a function of input variables and previously labeled gates with other arbitrary symbols. Find the Boolean functions for these gates.



4-2. Analysis procedure

3. Repeat the process outlined in step 2 until the outputs of the circuit are obtained.
4. By repeated substitution of previously defined functions, obtain the output Boolean functions in terms of input variables.

Example

$$F_2 = AB + AC + BC; \quad T_1 = A + B + C; \quad T_2 = ABC; \quad T_3 = F_2'T_1;$$

$$F_1 = T_3 + T_2$$

$$F_1 = T_3 + T_2 = F_2'T_1 + ABC = A'BC' + A'B'C + AB'C' + ABC$$

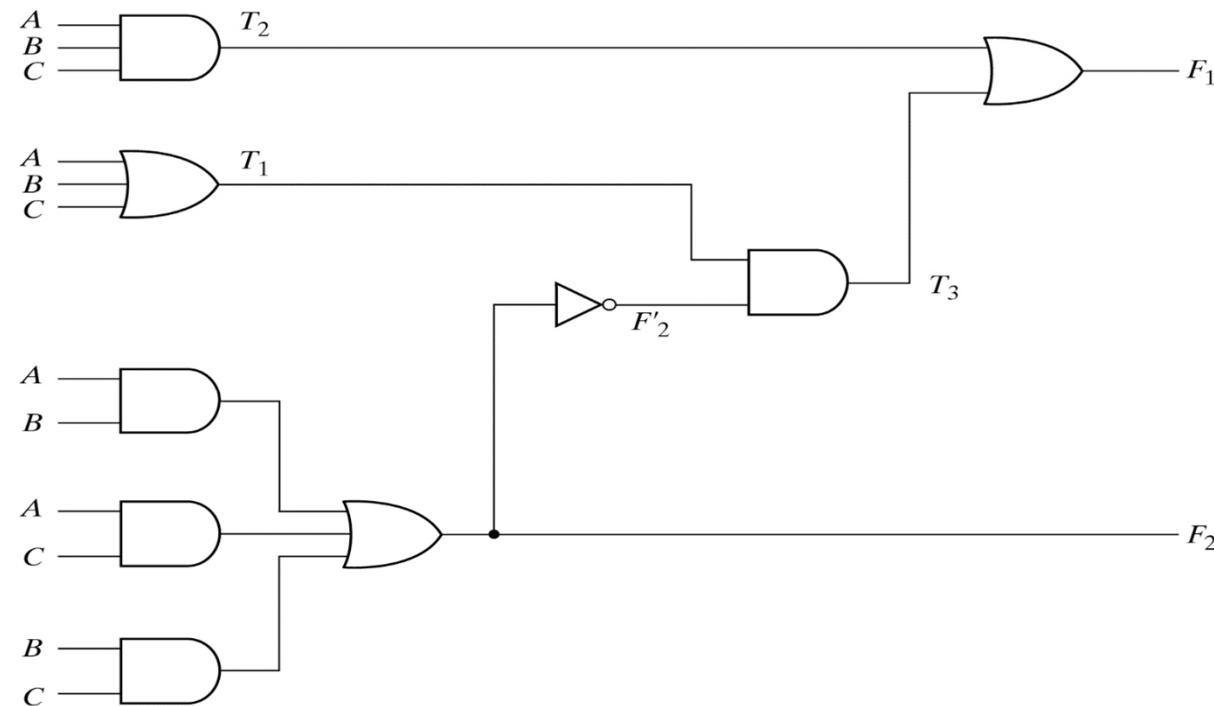
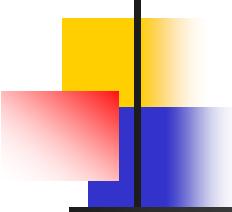


Fig. 4-2 Logic Diagram for Analysis Example

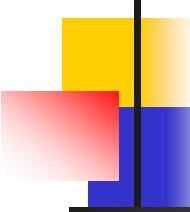


Derive truth table from logic diagram

- We can derive the truth table in Table 4-1 by using the circuit of Fig.4-2.

Table 4-1
Truth Table for the Logic Diagram of Fig. 4-2

A	B	C	F_2	F_1	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

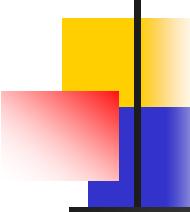


4-3. Design procedure

1. Table 4-2 is a Code-Conversion example, first, we can list the relation of the BCD and Excess-3 codes in the truth table.

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



Karnaugh map

- For each symbol of the Excess-3 code, we use 1's to draw the map for simplifying Boolean function.

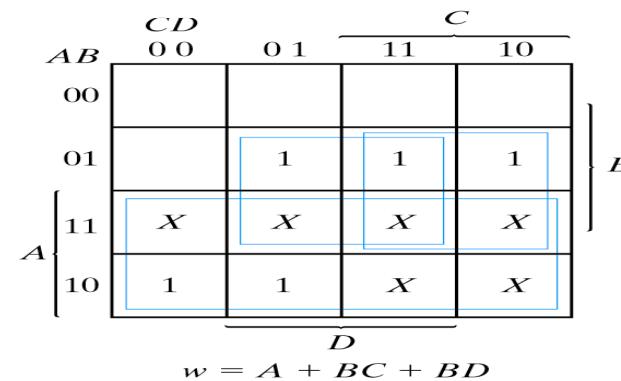
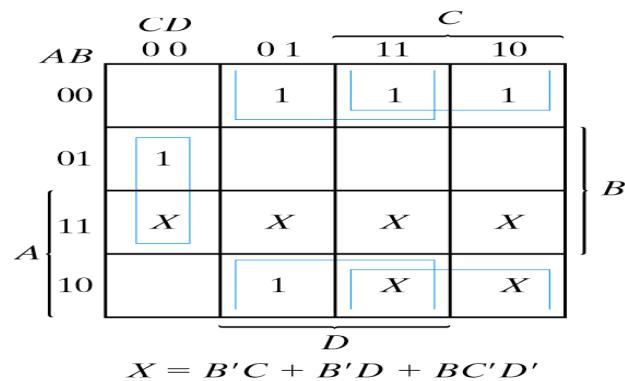
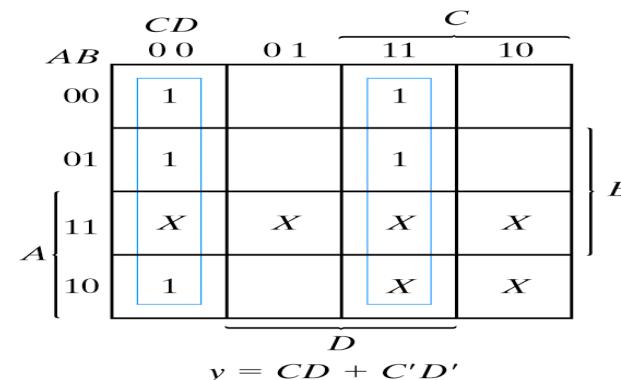
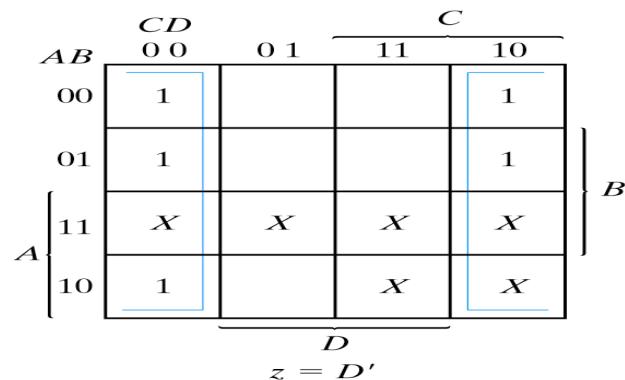


Fig. 4-3 Maps for BCD to Excess-3 Code Converter

Circuit implementation

$$z = D'; \quad y = CD + C'D' = CD + (C + D)'$$

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

$$w = A + BC + BD = A + B(C + D)$$

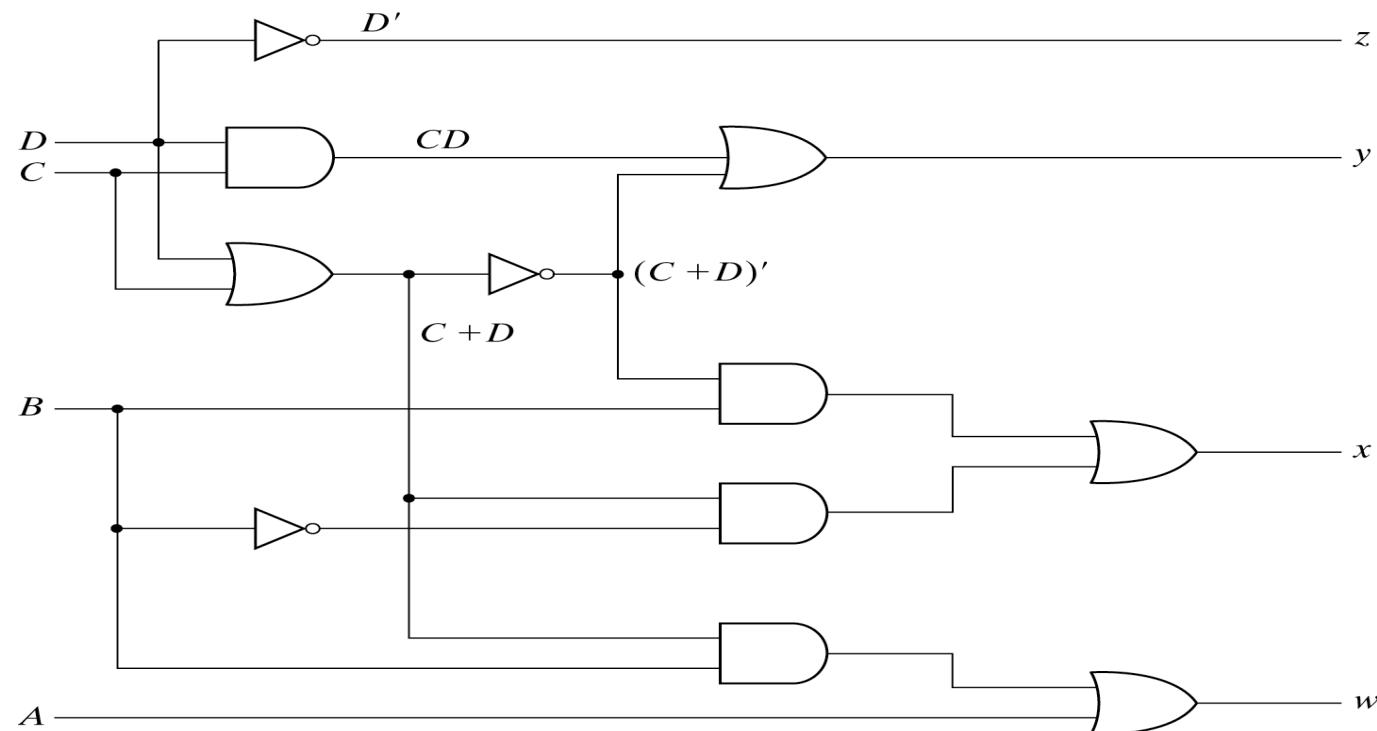
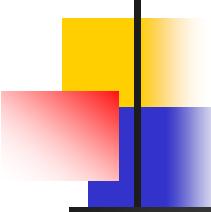


Fig. 4-4 Logic Diagram for BCD to Excess-3 Code Converter



4-4. Binary Adder-Subtractor

- A combinational circuit that performs the addition of two bits is called a **half adder**.
- The truth table for the half adder is listed below:

Table 4-3
Half Adder

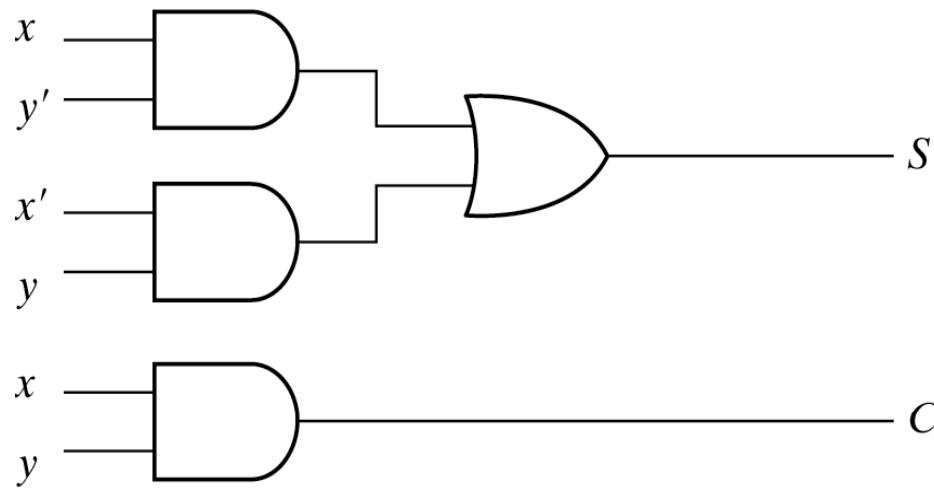
x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

S: Sum
C: Carry

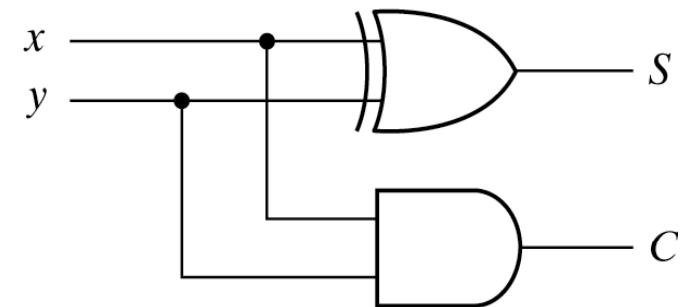
$$S = x'y + xy'$$

$$C = xy$$

Implementation of Half-Adder

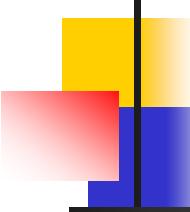


$$(a) S = xy' + x'y
C = xy$$



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

Fig. 4-5 Implementation of Half-Adder

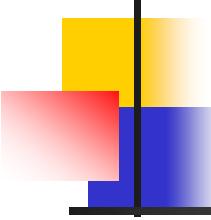


Full-Adder

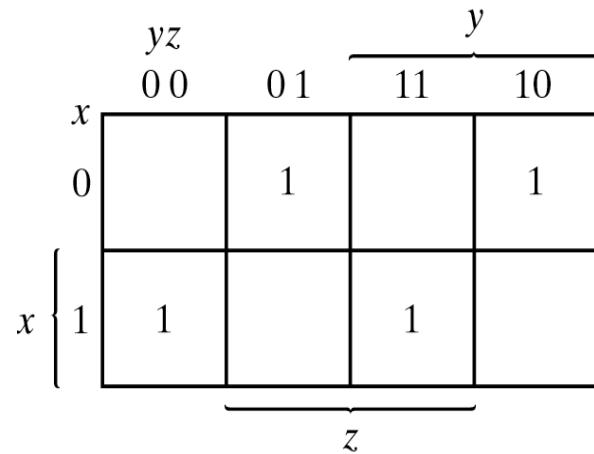
- One that performs the addition of three bits(two significant bits and a previous carry) is a **full adder**.

Table 4-4
Full Adder

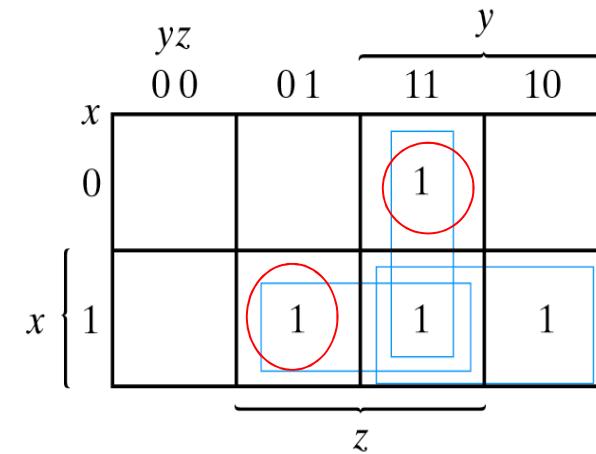
x	y	z	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Simplified Expressions



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



$$\begin{aligned} C &= xy + xz + yz \\ &= xy + \boxed{xy'z + x'yz} \end{aligned}$$

Fig. 4-6 Maps for Full Adder

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

$$C = xy + xz + yz$$

Full adder implemented in SOP

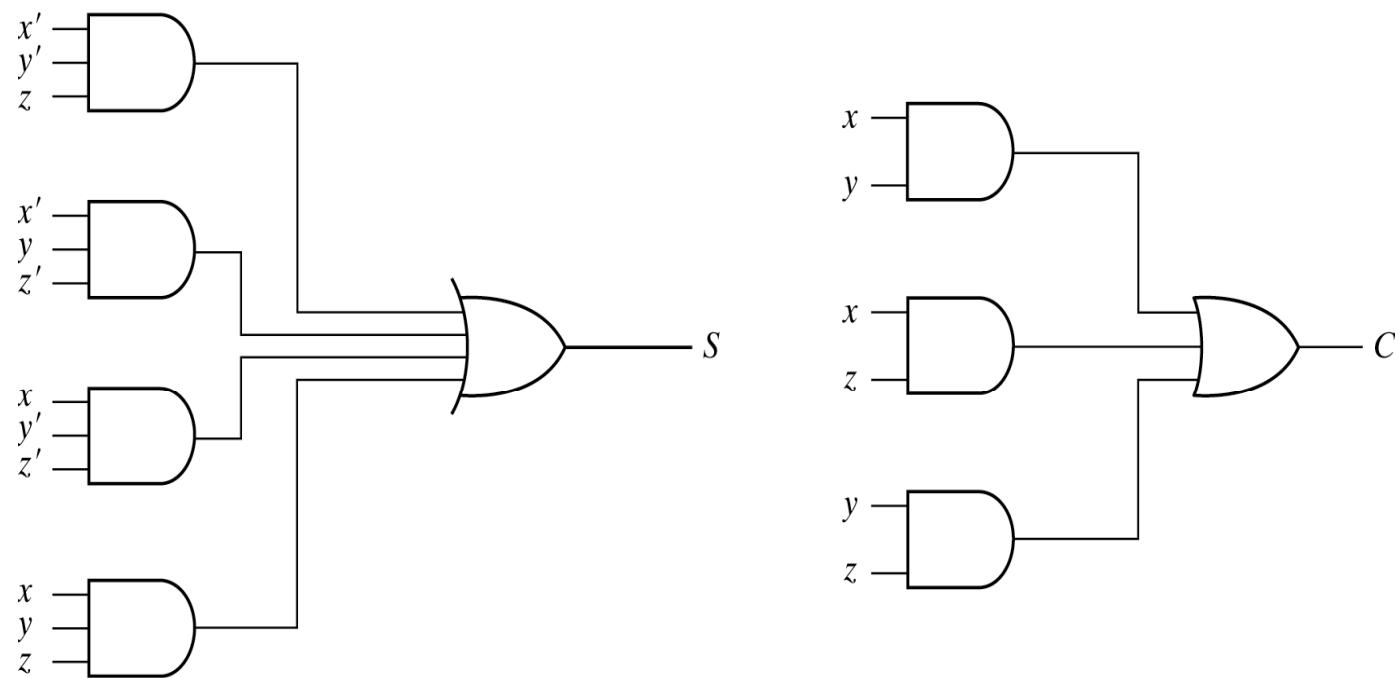


Fig. 4-7 Implementation of Full Adder in Sum of Products

Another implementation

- Full-adder can also implemented with **two half adders and one OR gate** (Carry Look-Ahead adder).

$$\begin{aligned}S &= z \oplus (x \oplus y) \\&= z'(xy' + x'y) + z(xy' + x'y)' \\&= xy'z' + x'yz' + xyz + x'y'z\end{aligned}$$

$$C = z(xy' + x'y) + xy = xy'z + x'yz + xy$$

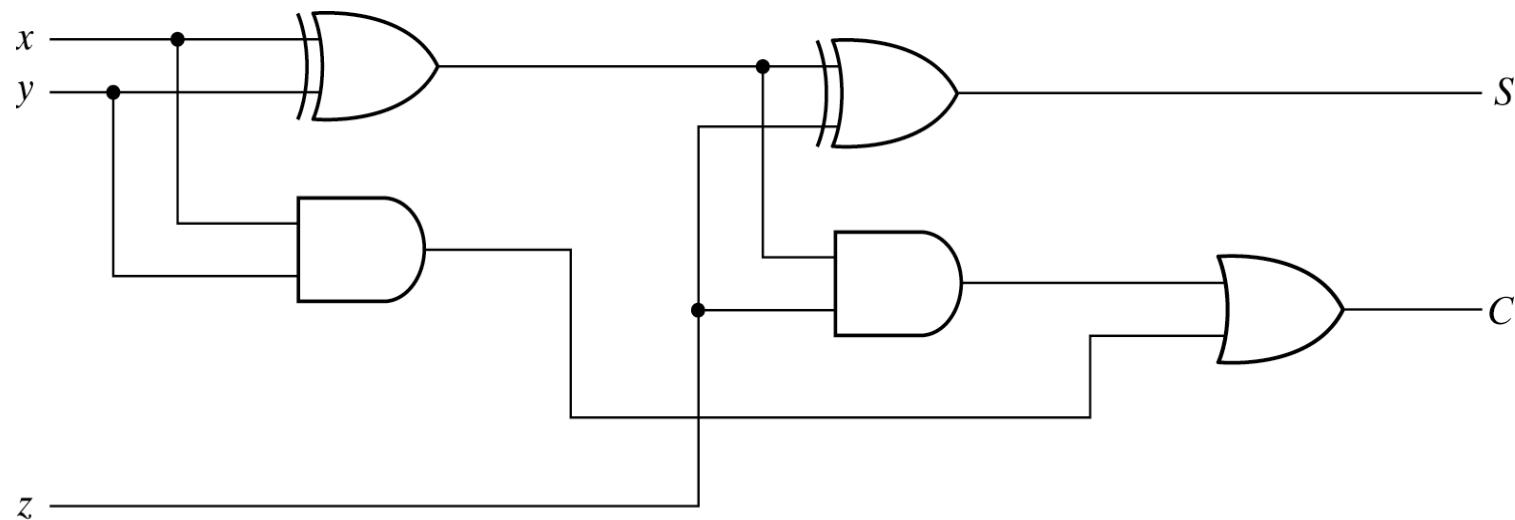


Fig. 4-8 Implementation of Full Adder with Two Half Adders and an OR Gate

Binary adder

- This is also called **Ripple Carry Adder**, because of the construction with full adders are connected in cascade.

<i>Subscript i:</i>	3	2	1	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}

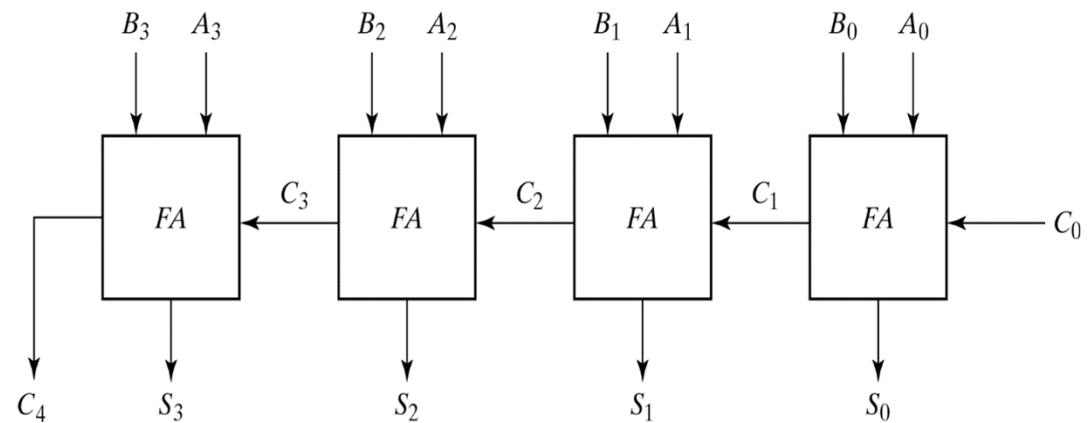
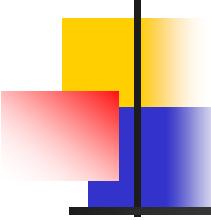


Fig. 4-9 4-Bit Adder



Carry Propagation

- Fig.4-9 causes a **unstable** factor on **carry bit**, and produces a **longest propagation delay**.
- The signal from C_i to the output carry C_{i+1} , **propagates through an AND and OR gates**, so, for an n -bit RCA, there are **$2n$** gate levels for the carry to propagate from input to output.

Carry Propagation

- Because the propagation delay will affect the output signals on different time, so the signals are **given enough time to get the precise and stable outputs.**
- The most widely used technique employs the principle of **carry look-ahead** to **improve the speed of the algorithm.**

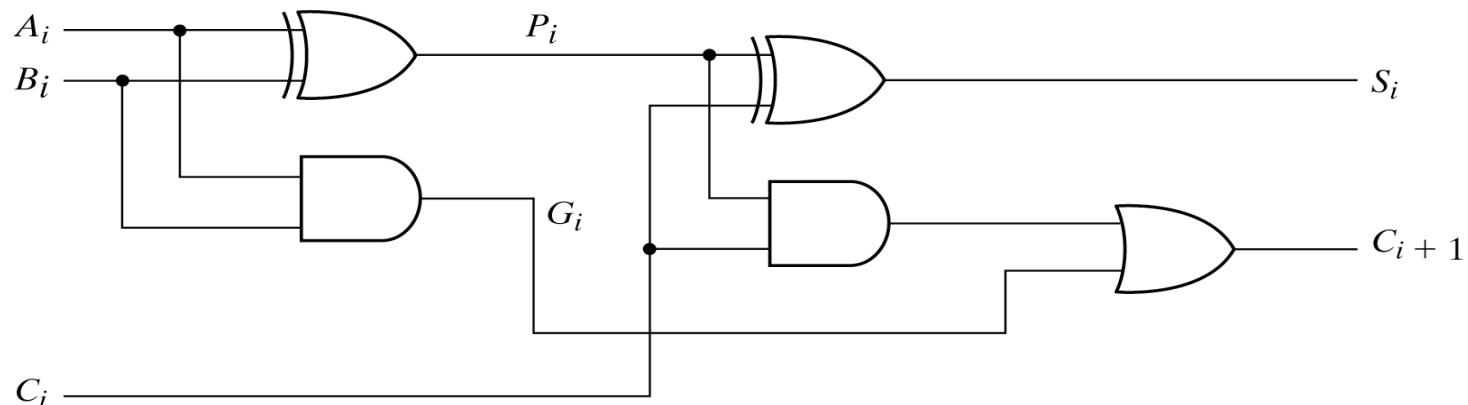
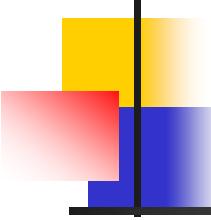


Fig. 4-10 Full Adder with P and G Shown



Boolean functions

$$P_i = A_i \oplus B_i \quad \text{steady state value}$$

$$G_i = A_i B_i \quad \text{steady state value}$$

Output sum and carry

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i C_i$$

G_i : carry generate P_i : carry propagate

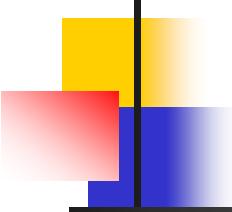
C_0 = input carry

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0$$

$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

- C_3 does not have to wait for C_2 and C_1 to propagate.



Logic diagram of carry look-ahead generator

- C_3 is propagated at the same time as C_2 and C_1 .

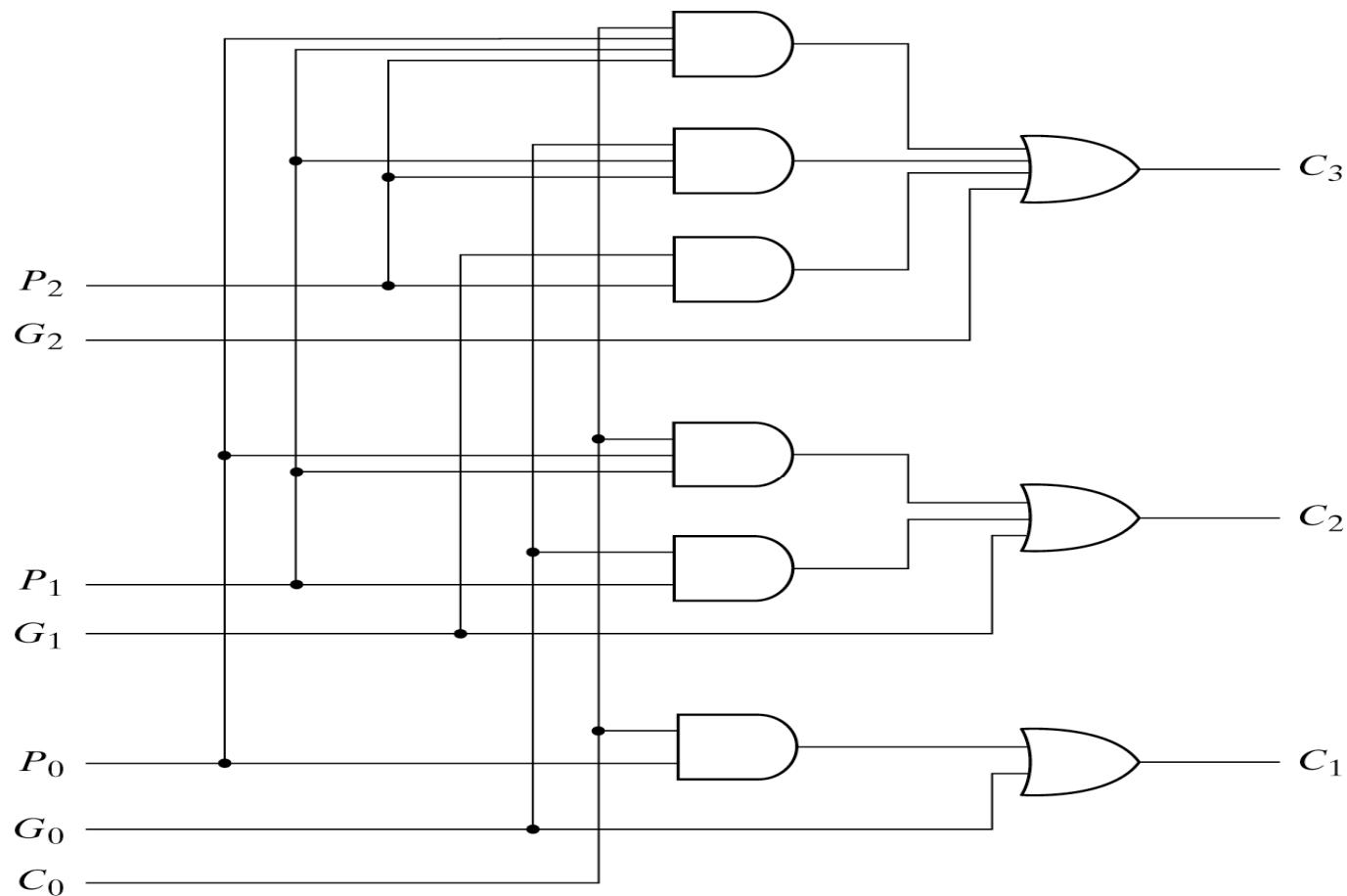


Fig. 4-11 Logic Diagram of Carry Lookahead Generator

4-bit adder with carry lookahead

- Delay time of n-bit CLAA = XOR + (AND + OR) + XOR

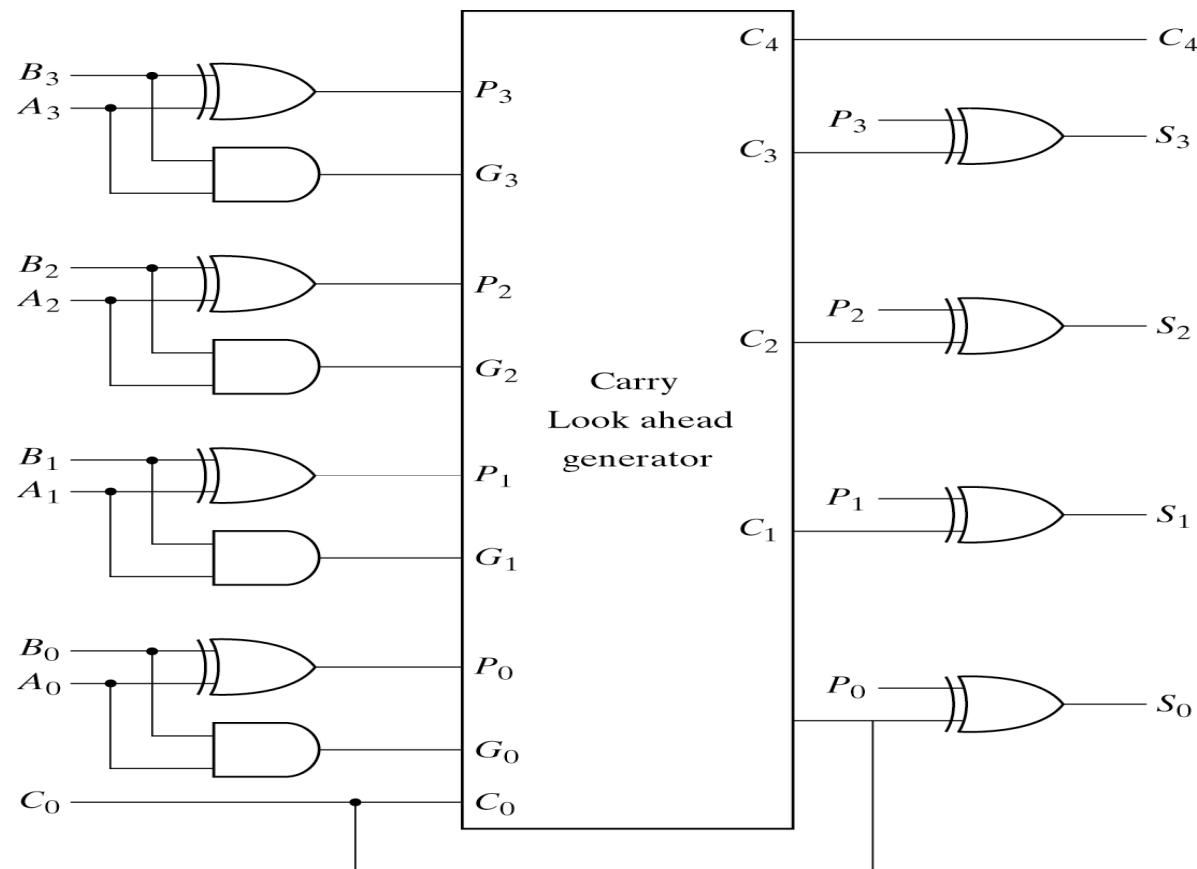


Fig. 4-12 4-Bit Adder with Carry Lookahead

Binary subtractor

$M = 1 \rightarrow \text{subtractor}$; $M = 0 \rightarrow \text{adder}$

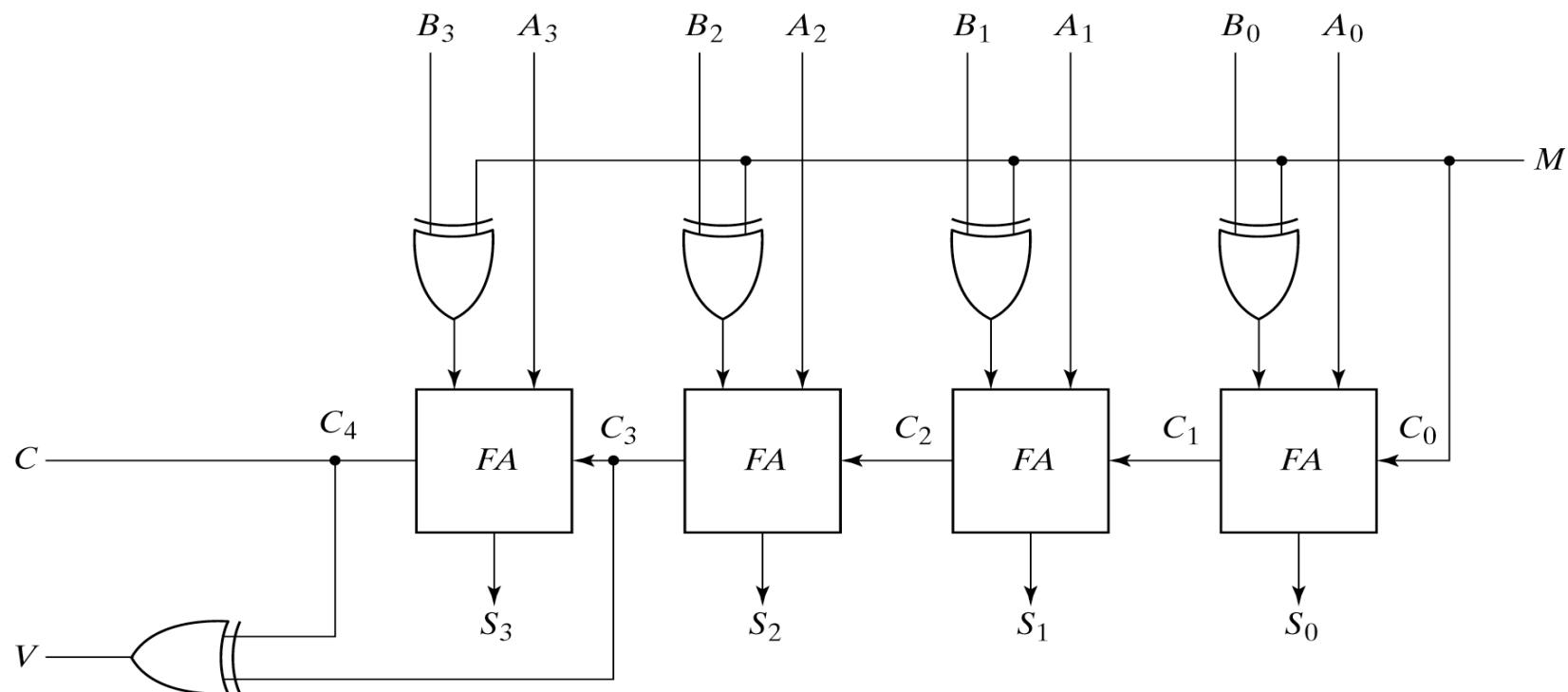
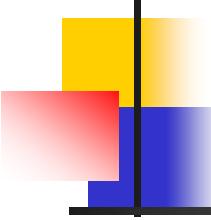
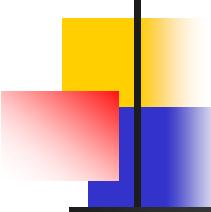


Fig. 4-13 4-Bit Adder Subtractor



Overflow

- It is **worth** noting Fig.4-13 that binary numbers in the **signed-complement system are added and subtracted** by the same basic addition and subtraction rules **as unsigned numbers**.
- Overflow is a problem in digital computers because the number of bits that hold the number is finite and a result that contains $n+1$ bits cannot be accommodated.



Overflow on signed and unsigned

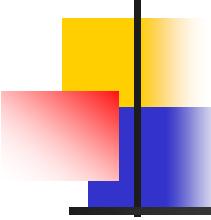
- When two **unsigned** numbers are added, an overflow is detected from the **end carry out of the MSB position**.
- When two **signed** numbers are added, the sign bit is treated as part of the number and the end carry does not indicate an overflow.
- An **overflow can't occur** after an addition if one number is **positive** and the other is **negative**.
- An overflow may occur if the two numbers added are both positive or both negative.

4-5 Decimal adder

BCD adder can't exceed 9 on each input digit. K is the carry.

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



Rules of BCD adder

- When the binary sum is **greater than 1001**, we obtain a **non-valid BCD** representation.
- The **addition of binary 6(0110)** to the binary sum **converts it to the correct BCD** representation and also produces an output carry as required.
- To distinguish them from binary 1000 and 1001, which also have a 1 in position Z_8 , we specify further that either Z_4 or Z_2 must have a 1.

$$C = K + Z_8Z_4 + Z_8Z_2$$

Implementation of BCD adder

- A decimal parallel adder that adds n decimal digits needs n BCD adder stages.
- The **output carry from one stage** must **be connected** to the input carry of the next higher-order stage.

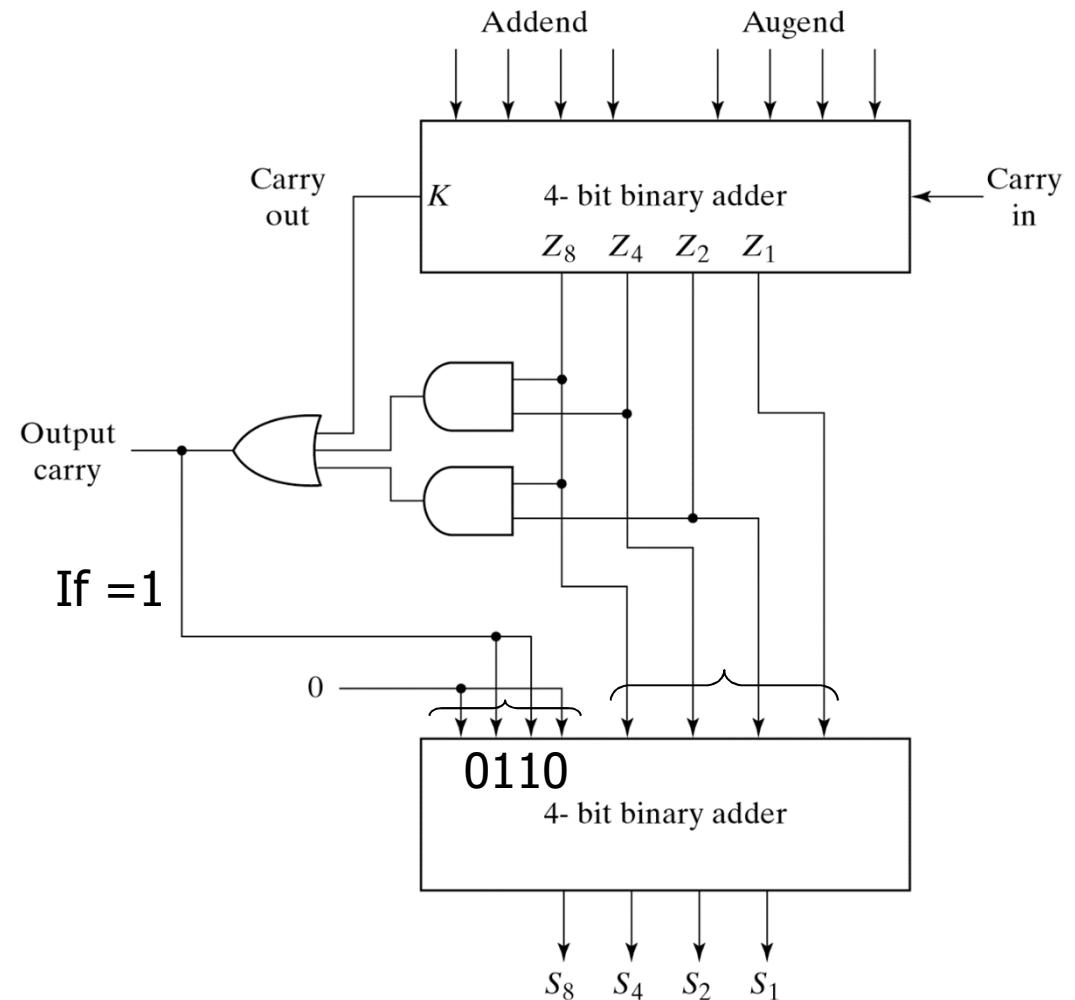


Fig. 4-14 Block Diagram of a BCD Adder

4-6. Binary multiplier

- Usually there are **more bits** in the partial products and it is necessary to use **full adders** to produce the sum of the partial products.

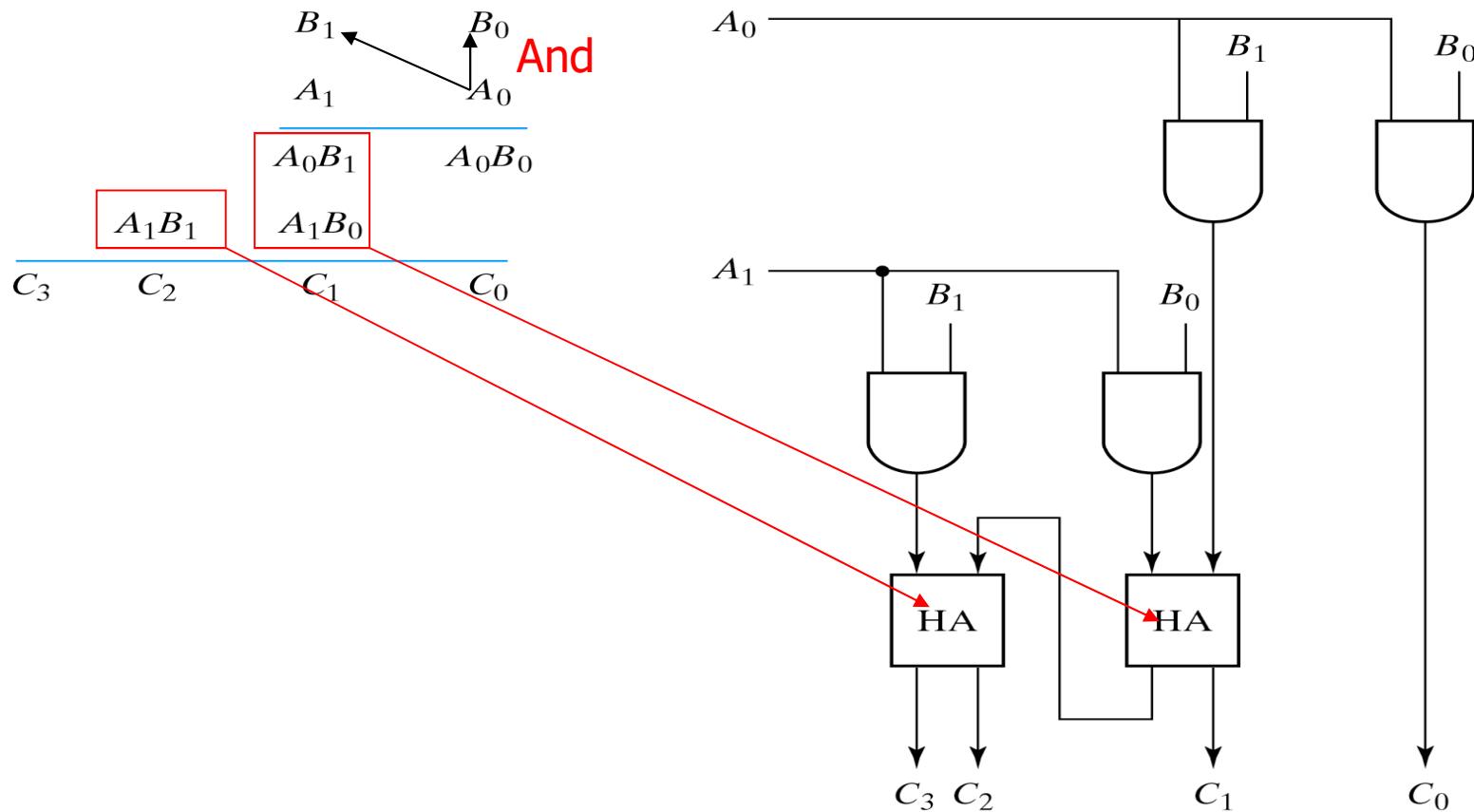


Fig. 4-15 2-Bit by 2-Bit Binary Multiplier

4-bit by 3-bit binary multiplier

- For J multiplier bits and K multiplicand bits we need $(J \times K)$ AND gates and $(J - 1)K$ -bit adders to produce a product of $J+K$ bits.
- $K=4$ and $J=3$, we need 12 AND gates and two 4-bit adders.

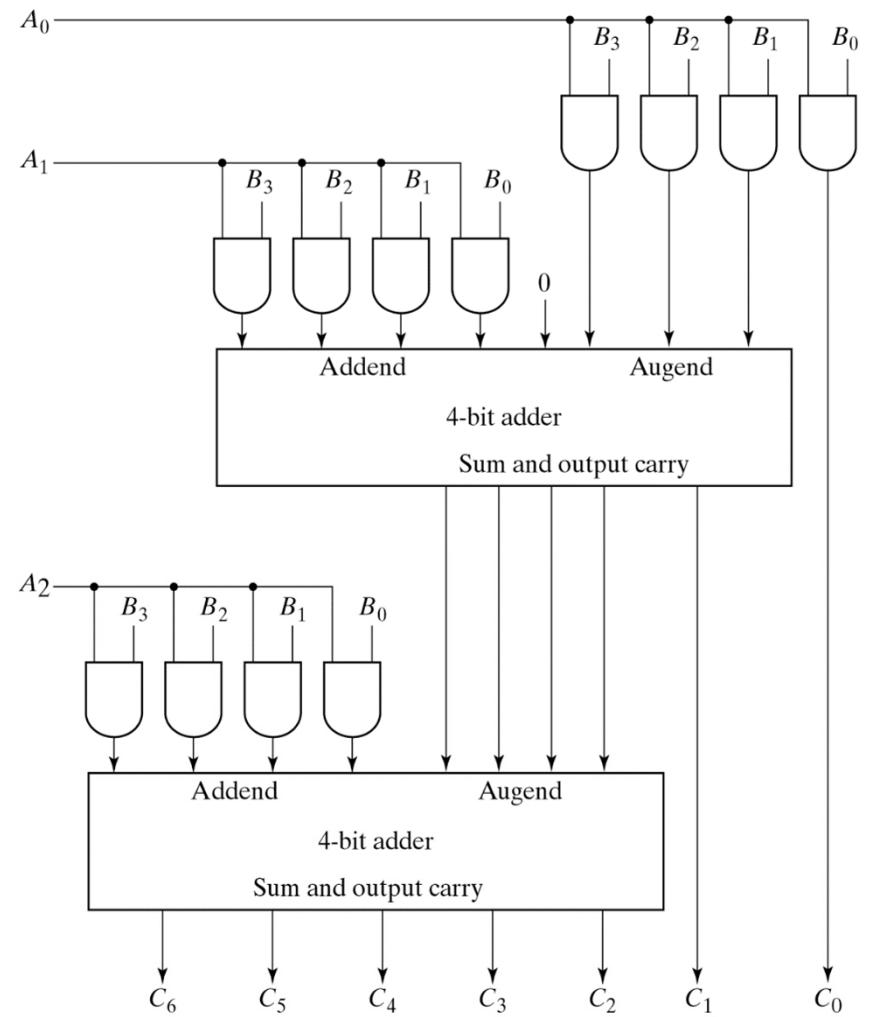


Fig. 4-16 4-Bit by 3-Bit Binary Multiplier

4-7. Magnitude comparator

- The equality relation of each pair of bits can be expressed logically with an **exclusive-NOR** function as:

$$A = A_3A_2A_1A_0 ; B = B_3B_2B_1B_0$$

$$x_i = A_iB_i + A'_iB'_i \quad \text{for } i = 0, 1, 2, 3$$

$$(A = B) = x_3x_2x_1x_0$$

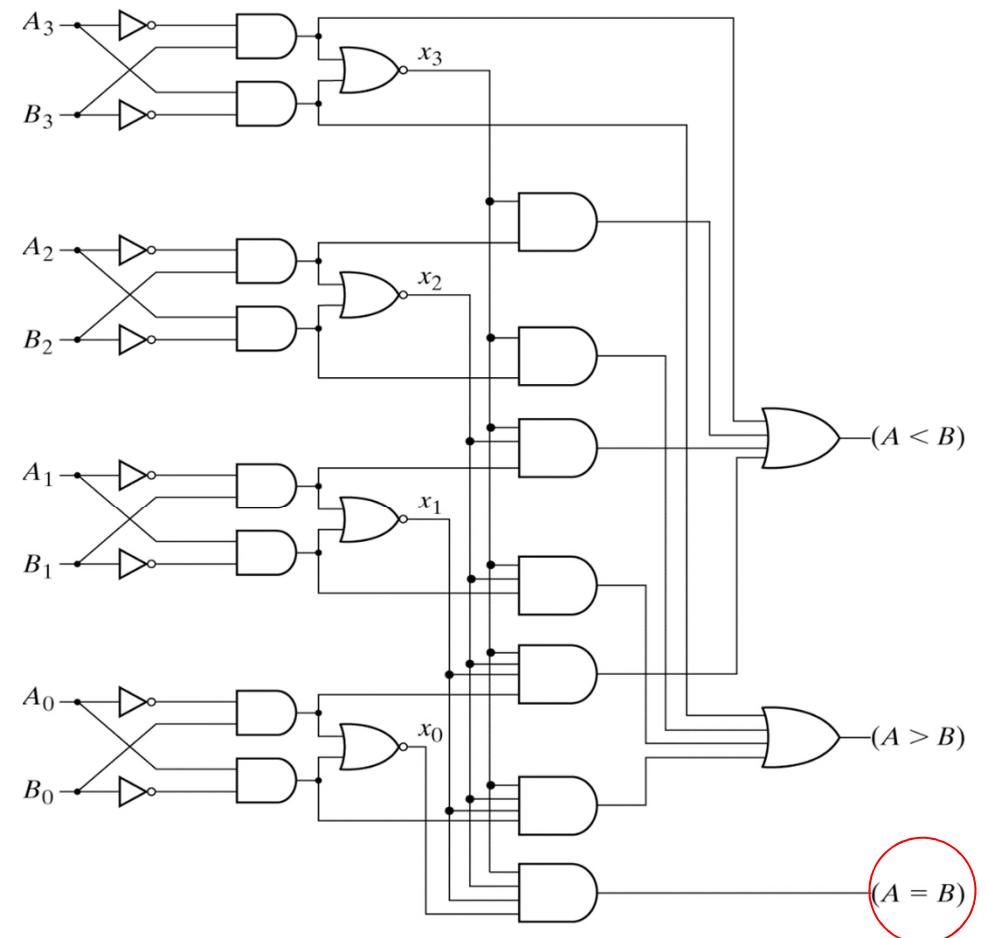


Fig. 4-17 4-Bit Magnitude Comparator

Magnitude comparator

- We inspect the relative magnitudes of pairs of MSB. If equal, we compare the next lower significant pair of digits until a pair of unequal digits is reached.
- If the corresponding digit of A is 1 and that of B is 0, we conclude that $A > B$.

$$(A > B) =$$

$$A_3 B'_3 + x_3 A_2 B'_2 + x_3 x_2 A_1 B'_1 + x_3 x_2 x_1 A_0 B'_0$$

$$(A < B) =$$

$$A'_3 B_3 + x_3 A'_2 B_2 + x_3 x_2 A'_1 B_1 + x_3 x_2 x_1 A'_0 B_0$$

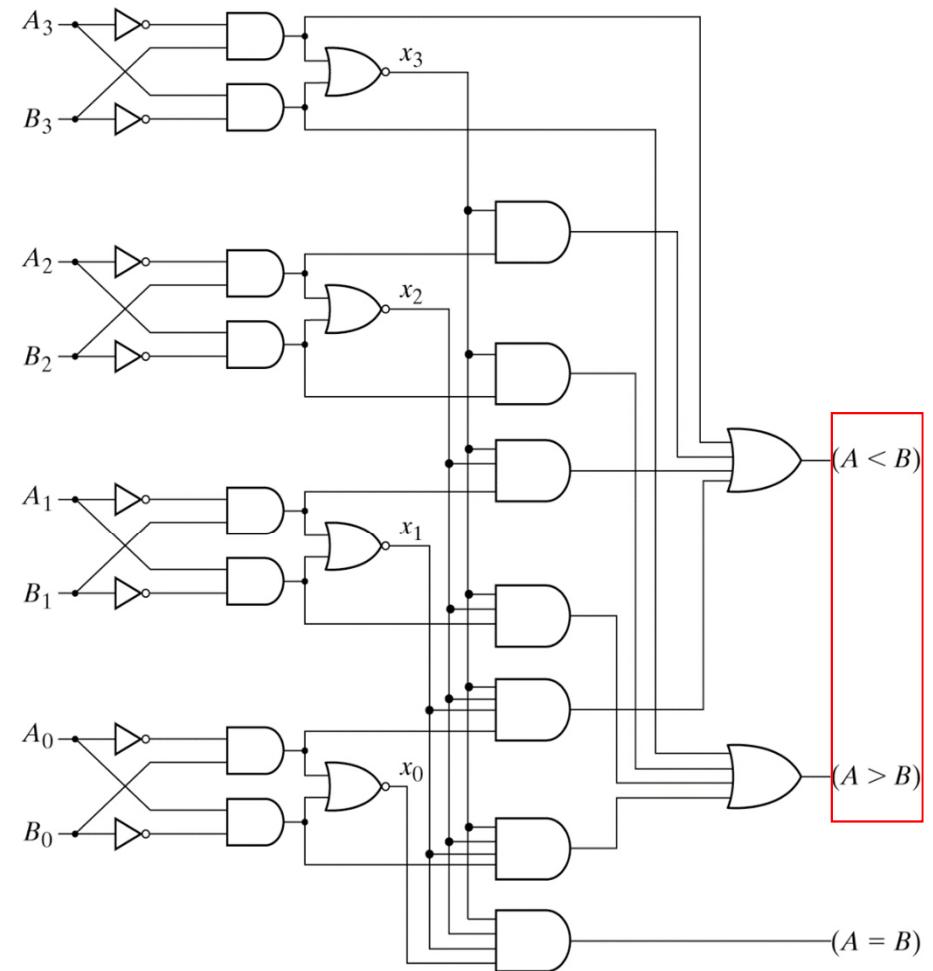
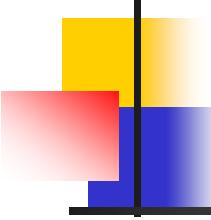


Fig. 4-17 4-Bit Magnitude Comparator



4-8. Decoders

- The decoder is called n-to-m-line decoder, where $m \leq 2^n$.
- the decoder is also used in conjunction with other code converters such as a BCD-to-seven_segment decoder.
- 3-to-8 line decoder: For each possible input combination, there are seven outputs that are equal to 0 and only one that is equal to 1.

Implementation and truth table

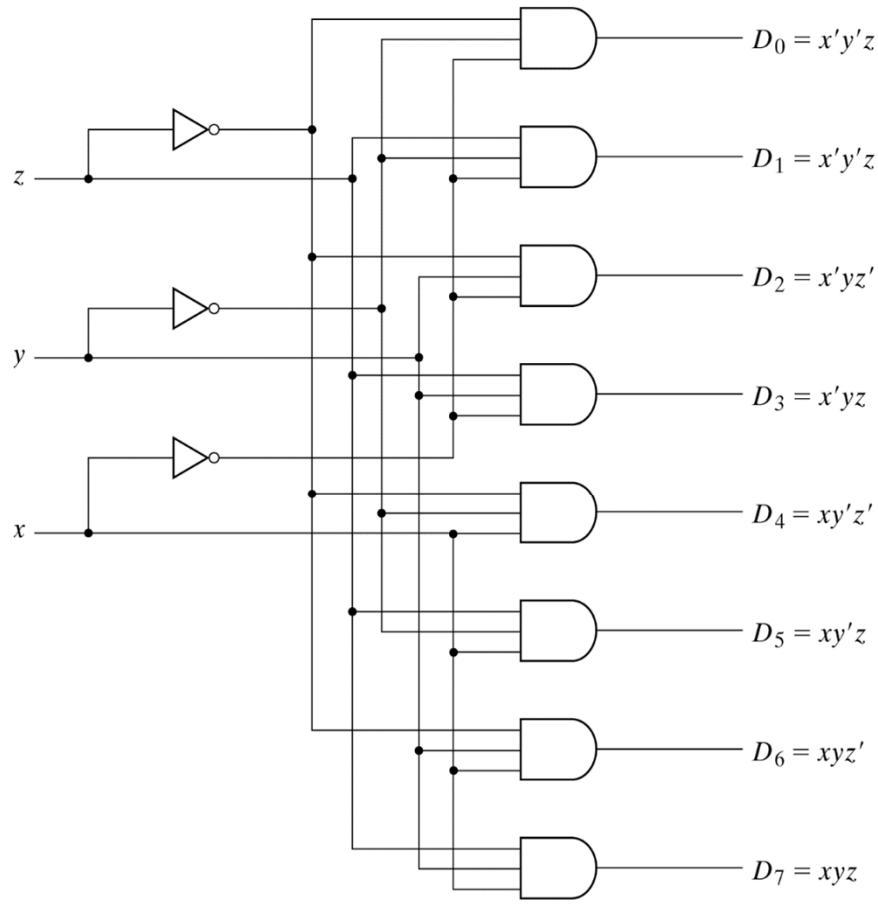


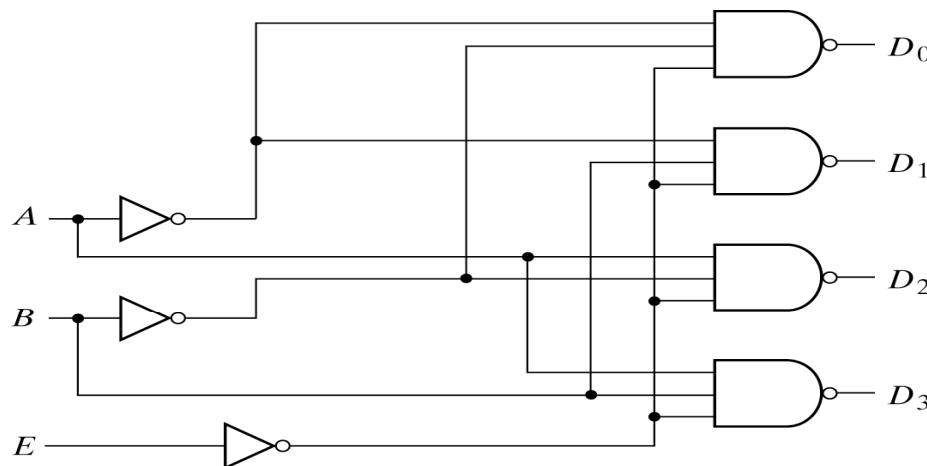
Table 4-6
Truth Table of a 3-to-8-Line Decoder

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

Fig. 4-18 3-to-8-Line Decoder

Decoder with enable input

- Some decoders are constructed with NAND gates, it becomes more economical to generate the decoder minterms in their complemented form.
- As indicated by the truth table , only one output can be equal to 0 at any given time, all other outputs are equal to 1.



(a) Logic diagram

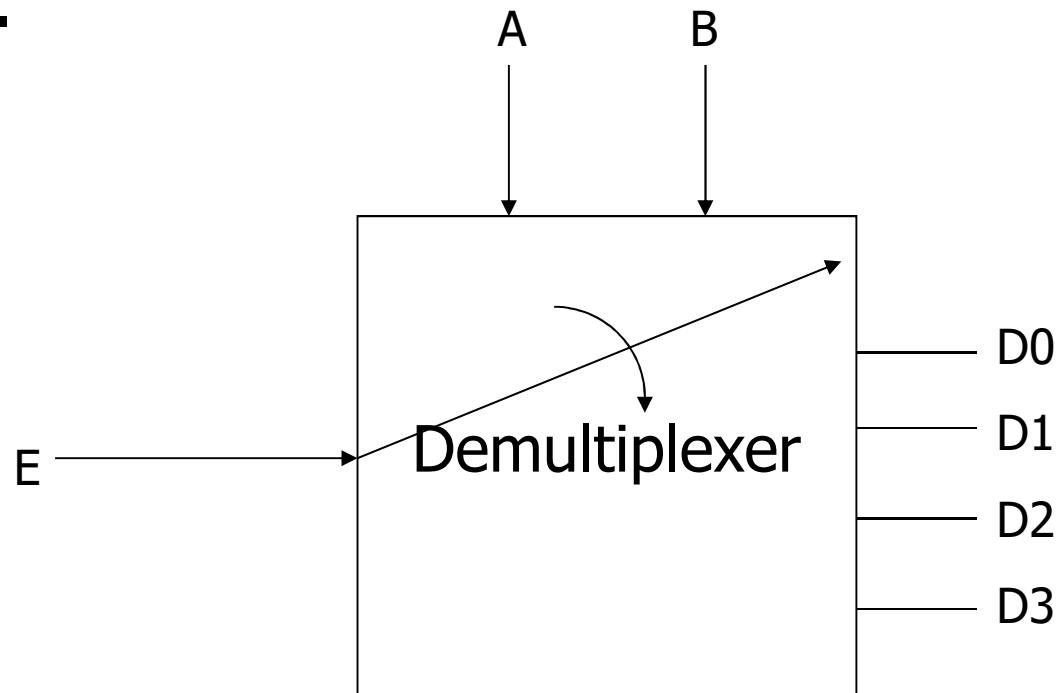
E	A	B	D ₀	D ₁	D ₂	D ₃
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

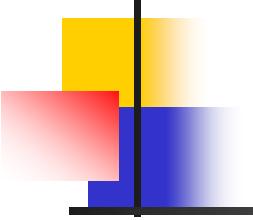
(b) Truth table

Fig. 4-19 2-to-4-Line Decoder with Enable Input

Demultiplexer

- A decoder with an enable input is referred to as a decoder/demultiplexer.
- The truth table of demultiplexer is the same with decoder.





3-to-8 decoder with enable implement the 4-to-16 decoder

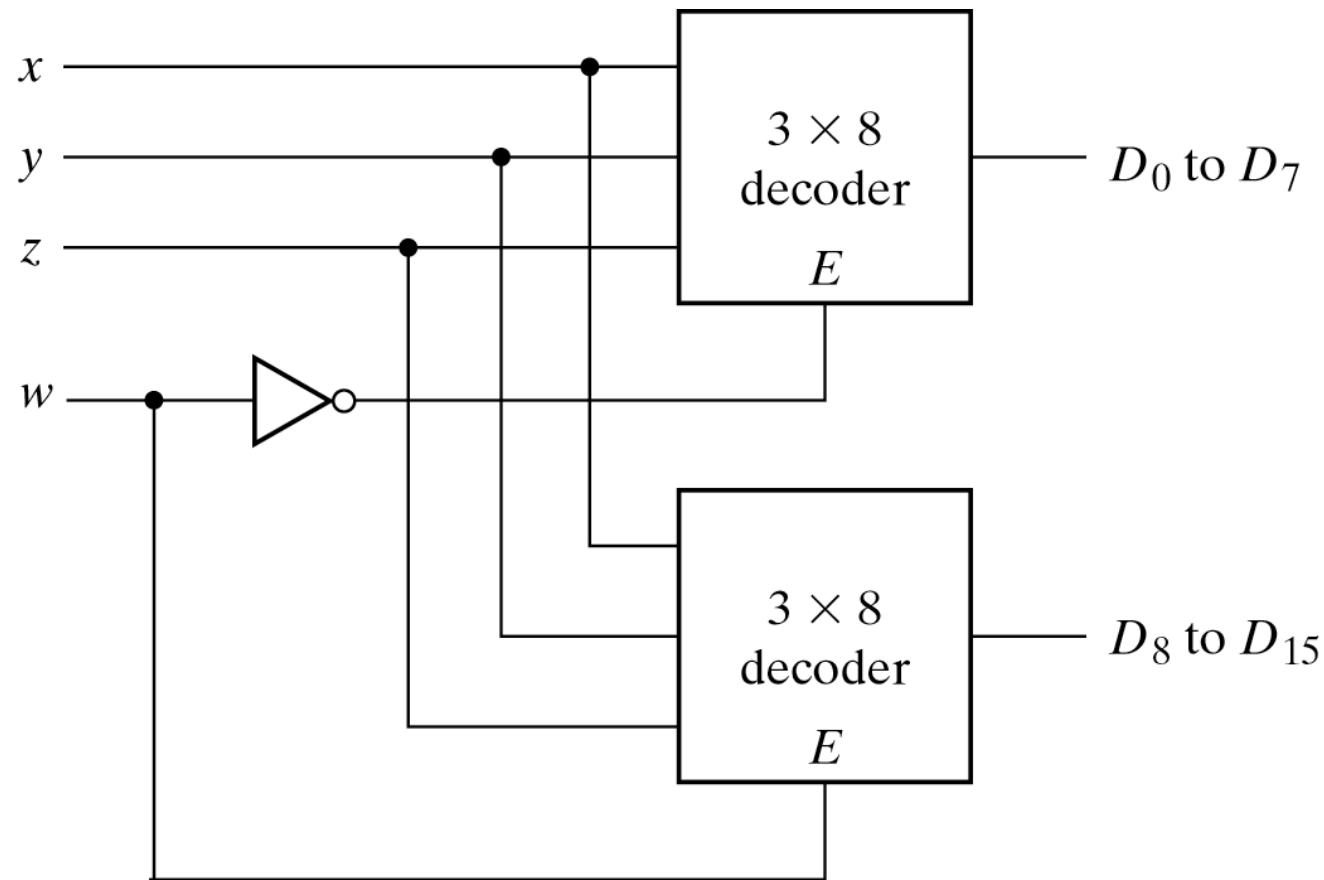
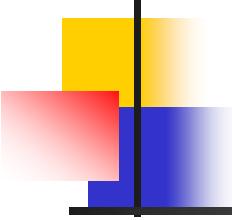


Fig. 4-20 4×16 Decoder Constructed with Two 3×8 Decoders



Implementation of a Full Adder with a Decoder

- From table 4-4, we obtain the functions for the combinational circuit in sum of minterms:

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

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

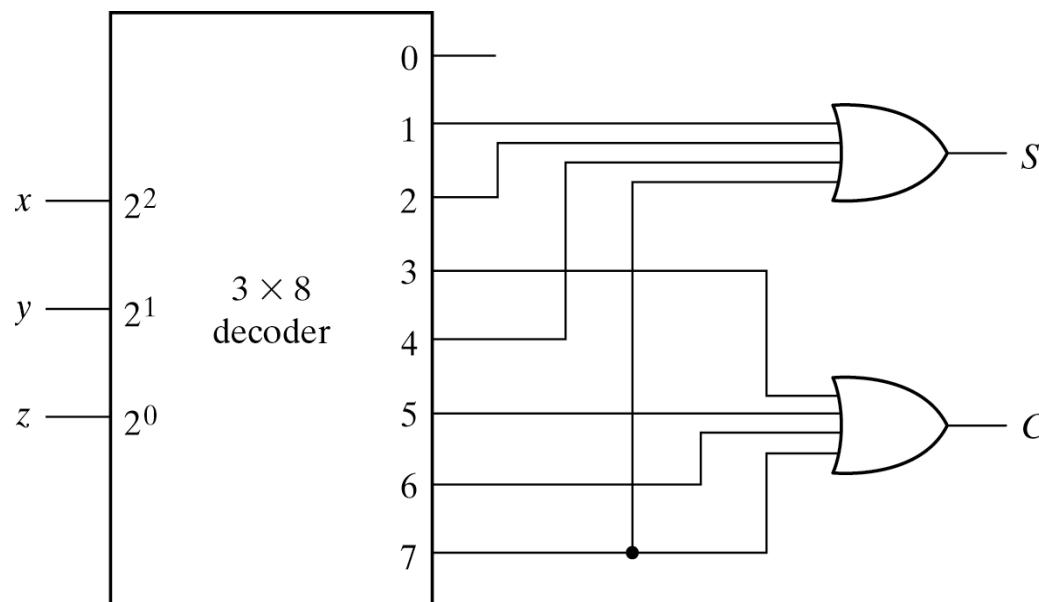


Fig. 4-21 Implementation of a Full Adder with a Decoder

4-9. Encoders

- An encoder is the inverse operation of a decoder.
- We can derive the Boolean functions by table 4-7

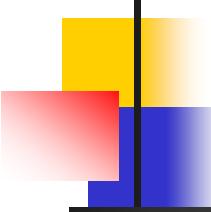
$$z = D_1 + D_3 + D_5 + D_7$$

$$y = D_2 + D_3 + D_6 + D_7$$

$$x = D_4 + D_5 + D_6 + D_7$$

Table 4-7
Truth Table of 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



Priority encoder

- If two **inputs** are **active simultaneously**, the **output** produces an **undefined combination**. We can establish an input **priority** to ensure that only one input is encoded.
- Another **ambiguity** in the octal-to-binary encoder is that an **output with all 0's** is generated when **all the inputs are 0**; the output is the same as when D_0 is equal to 1.
- The discrepancy tables on Table 4-7 and Table 4-8 can **resolve aforesaid condition by providing one more output** to indicate that at least one input is equal to 1.

Priority encoder

$V=0 \rightarrow$ no valid inputs

$V=1 \rightarrow$ valid inputs

X's in output columns represent
don't-care conditions

X's in the input columns are
useful for representing a truth
table in condensed form.

Instead of listing all 16
minterms of four variables.

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

4-input priority encoder

- Implementation of table 4-8

$$x = D_2 + D_3$$

$$y = D_3 + D_1 D'_2$$

$$V = D_0 + D_1 + D_2 + D_3$$

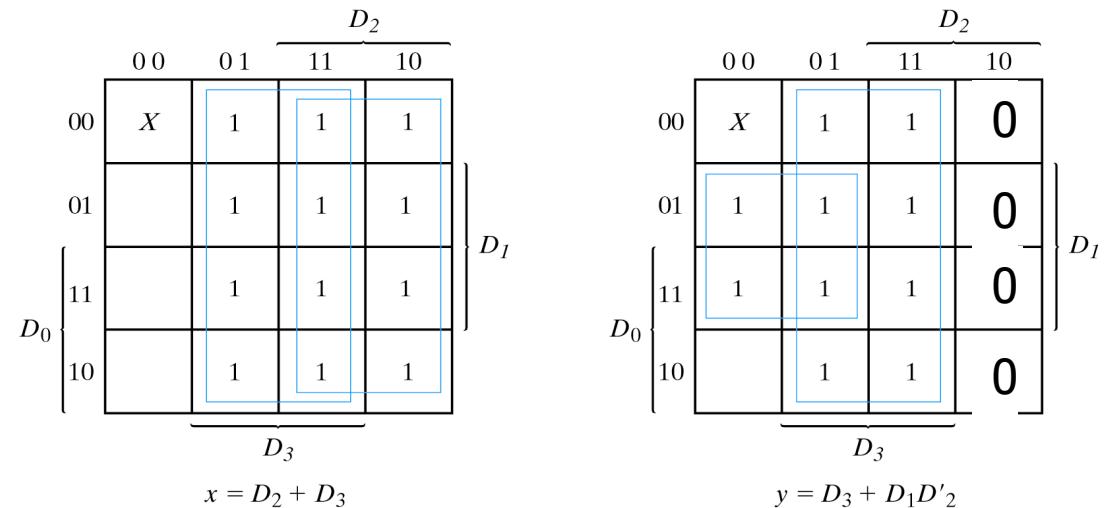


Fig. 4-22 Maps for a Priority Encoder

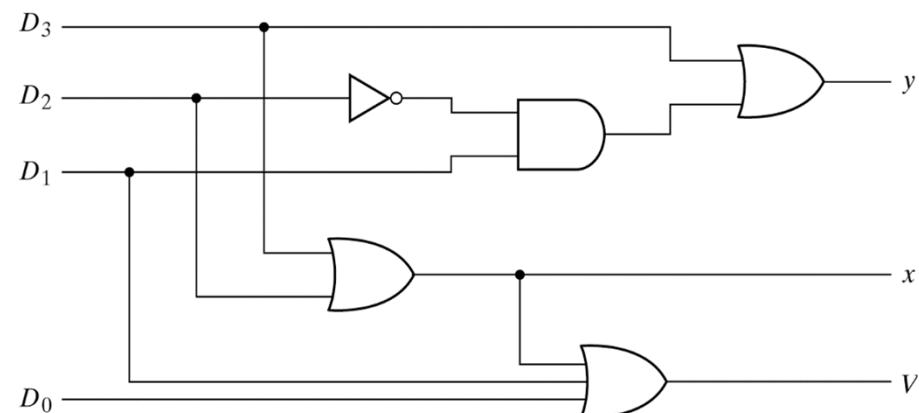


Fig. 4-23 4-Input Priority Encoder

4-10. Multiplexers

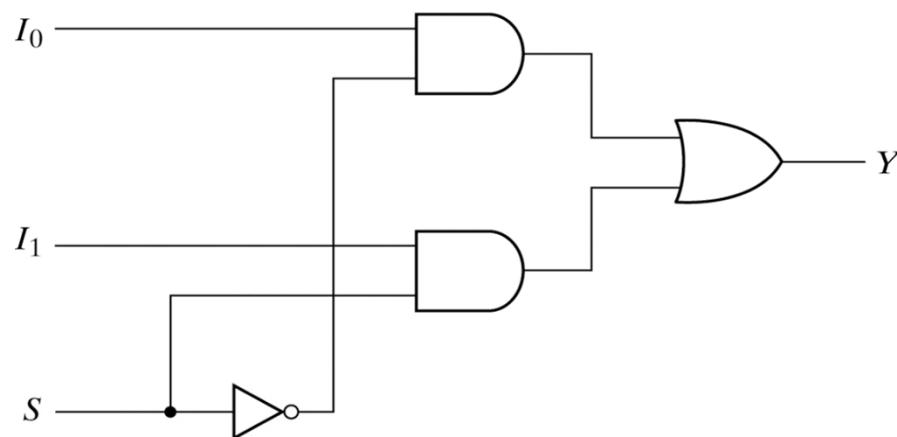
$$S = 0, Y = I_0$$

$$S = 1, Y = I_1$$

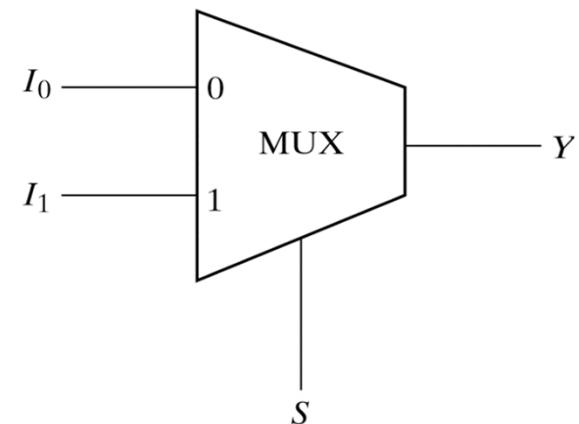
Truth Table \Rightarrow

S	Y
0	I_0
1	I_1

$$Y = S'I_0 + SI_1$$



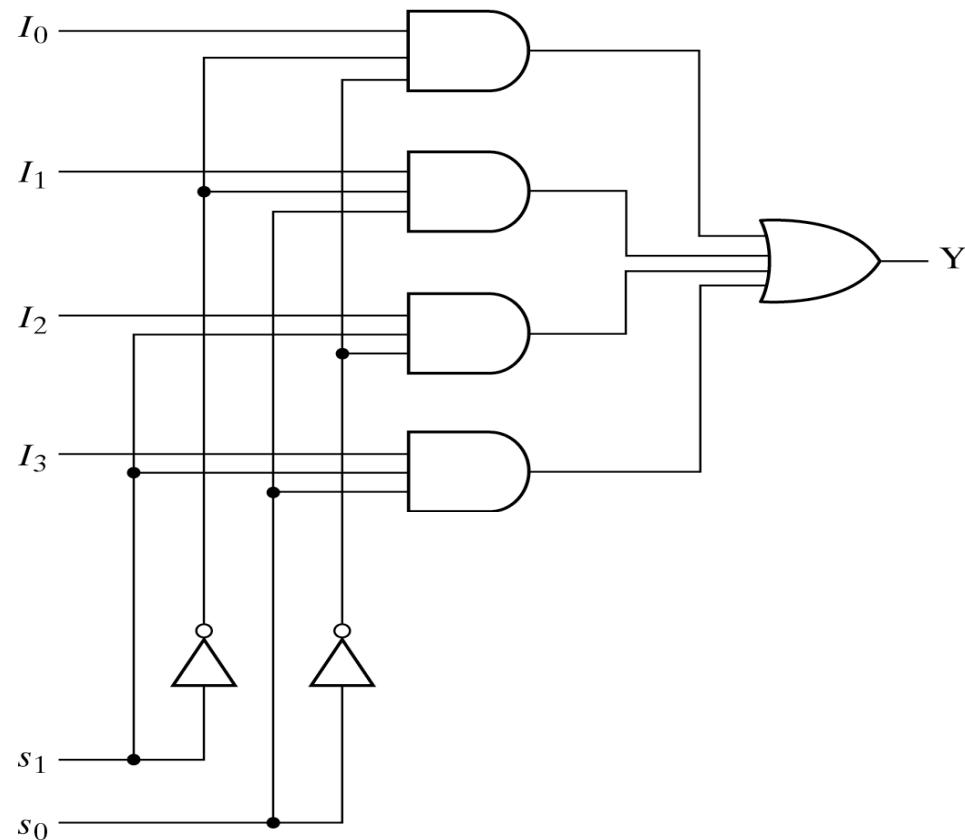
(a) Logic diagram



(b) Block diagram

Fig. 4-24 2-to-1-Line Multiplexer

4-to-1 Line Multiplexer



(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

Fig. 4-25 4-to-1-Line Multiplexer

Quadruple 2-to-1 Line Multiplexer

- Multiplexer circuits can be combined with common selection inputs to provide multiple-bit selection logic. Compare with Fig4-24.

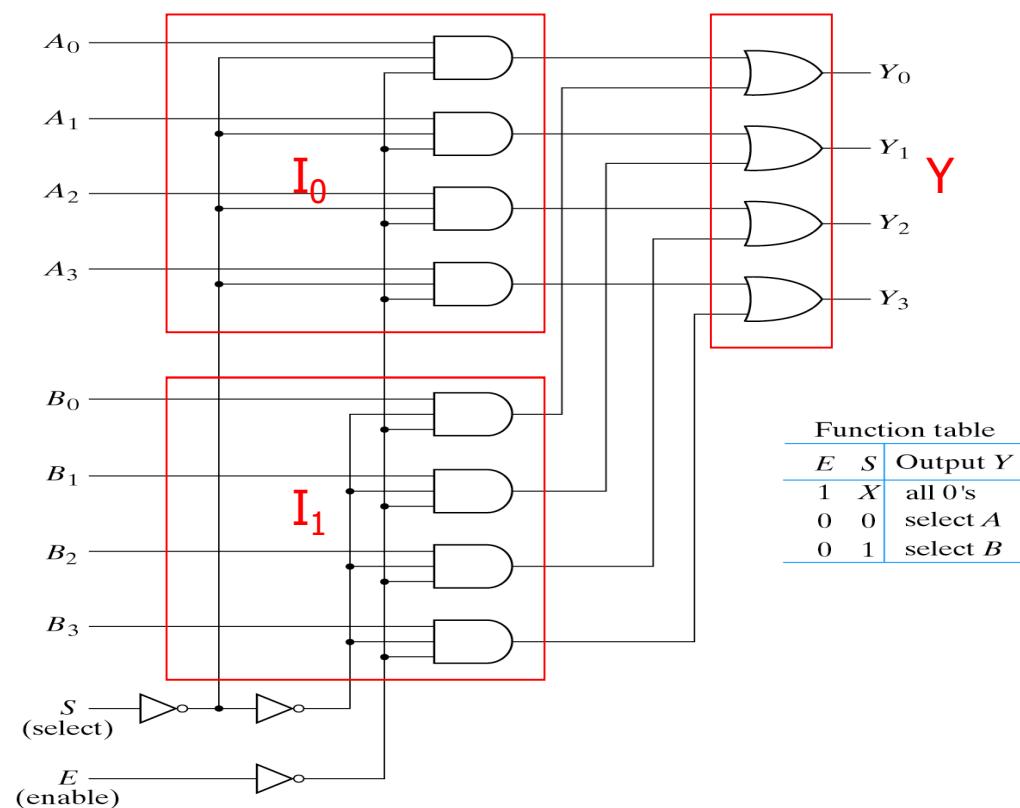


Fig. 4-26 Quadruple 2-to-1-Line Multiplexer

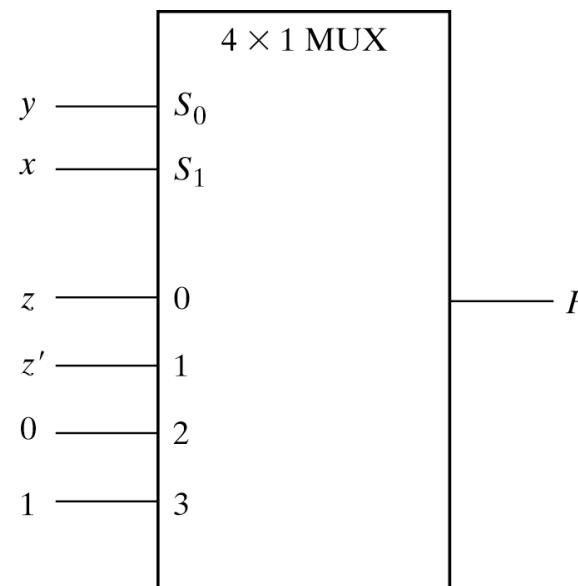
Boolean function implementation

- A more efficient method for implementing a Boolean function of n variables with a multiplexer that has $n-1$ selection inputs.

$$F(x, y, z) = \Sigma(1, 2, 6, 7)$$

x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

(a) Truth table



(b) Multiplexer implementation

Fig. 4-27 Implementing a Boolean Function with a Multiplexer

4-input function with a multiplexer

$$F(A, B, C, D) = \Sigma(1, 3, 4, 11, 12, 13, 14, 15)$$

A	B	C	D	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

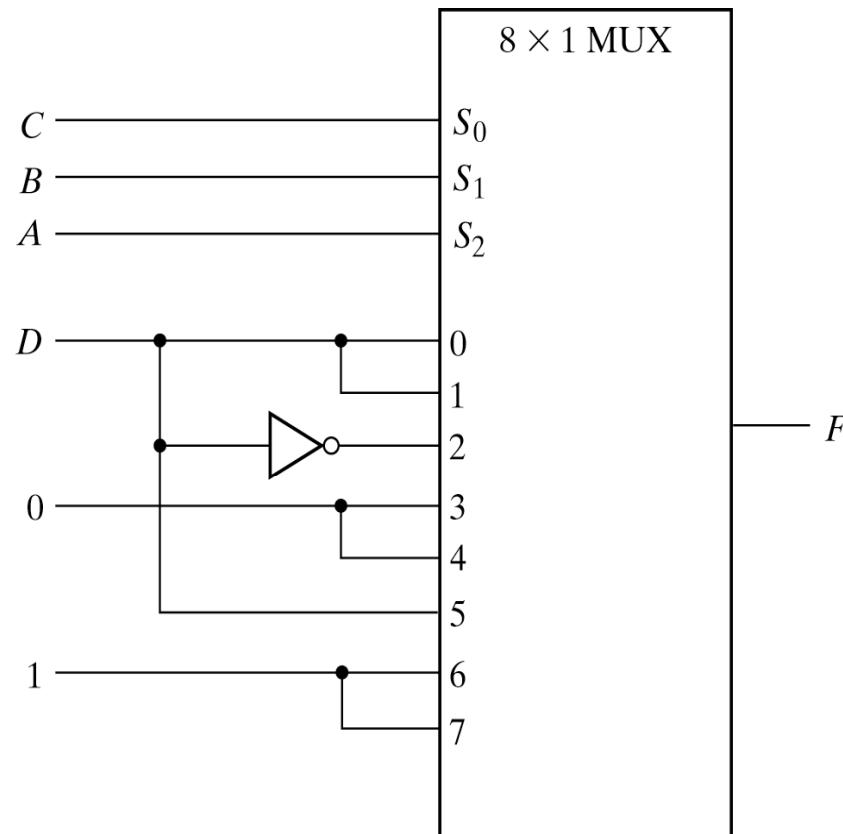


Fig. 4-28 Implementing a 4-Input Function with a Multiplexer

Three-State Gates

- A multiplexer can be constructed with three-state gates.

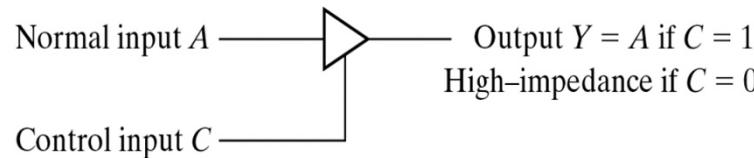
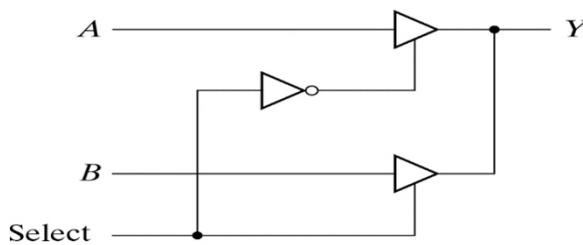
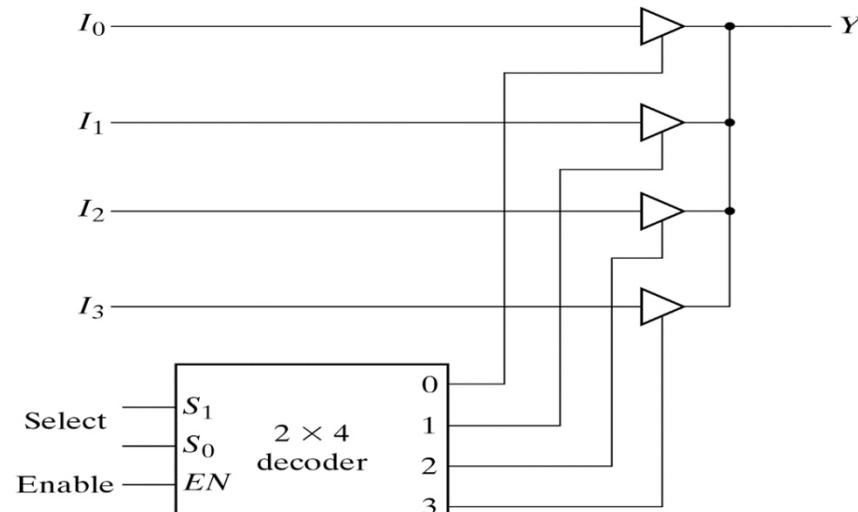


Fig. 4-29 Graphic Symbol for a Three-State Buffer

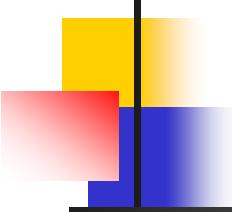


(a) 2-to-1- line mux



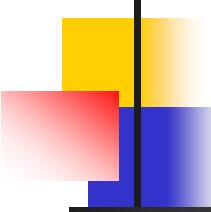
(b) 4 - to - 1 line mux

Fig. 4-30 Multiplexers with Three-State Gates



4-11. HDL for combinational circuits

- A module can be described in any one of the following modeling techniques:
 1. Gate-level modeling using instantiation of primitive gates and user-defined modules.
 2. Dataflow modeling using continuous assignment statements with keyword assign.
 3. Behavioral modeling using procedural assignment statements with keyword always.



Gate-level Modeling

- A circuit is specified by its logic gates and their interconnection.
- Verilog recognizes **12 basic gates** as **predefined primitives**.
- The logic values of each gate may be 1, 0, x(unknown), z(high-impedance).

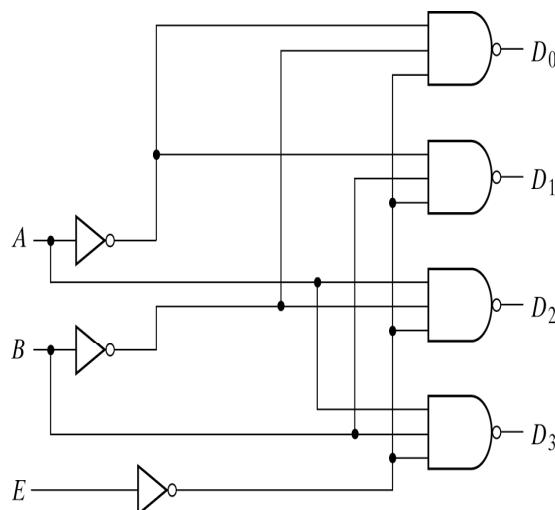
Table 4-9
Truth Table for Predefined Primitive Gates

and	0	1	x	z	or	0	1	x	z
0	0	0	0	0	0	0	1	x	x
1	0	1	x	x	1	1	1	1	1
x	0	x	x	x	x	x	1	x	x
z	0	x	x	x	z	x	1	x	x

xor	0	1	x	z	not	input	output
0	0	1	x	x		0	1
1	1	0	x	x		1	0
x	x	x	x	x		x	x
z	x	x	x	x		z	x

Gate-level description on Verilog code

The **wire** declaration is for internal connections.



(a) Logic diagram

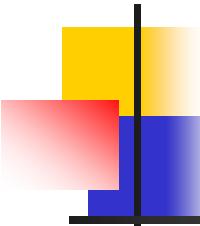
E	A	B	D ₀	D ₁	D ₂	D ₃
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

(b) Truth table

Fig. 4-19 2-to-4-Line Decoder with Enable Input

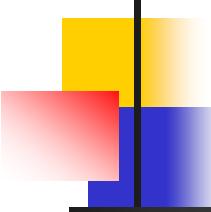
HDL Example 4-1

```
//Gate-level description of a 2-to-4-line decoder
//Figure 4-19
module decoder_gl (A,B,E,D);
    input A,B,E;
    output [0:3]D;
    wire Anot,Bnot,Enot;
    not
        n1 (Anot,A),
        n2 (Bnot,B),
        n3 (Enot,E);
    nand
        n4 (D[0],Anot,Bnot,Enot),
        n5 (D[1],Anot,B,Enot),
        n6 (D[2],A,Bnot,Enot),
        n7 (D[3],A,B,Enot);
endmodule
```



Design methodologies

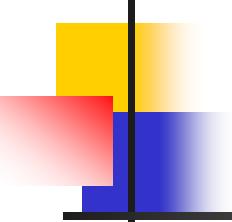
- There are two basic types of design methodologies: **top-down** and **bottom-up**.
- Top-down: the top-level block is defined and then the sub-blocks necessary to build the top-level block are identified.(Fig.4-9 binary adder)
- Bottom-up: the building blocks are first identified and then combined to build the top-level block.(Example 4-2 4-bit adder)



A bottom-up hierarchical description

HDL Example 4-2

```
//Gate-level hierarchical description of 4-bit adder
// Description of half adder (see Fig 4-5b)
module halfadder (S,C,x,y);
    input x,y;
    output S,C;
//Instantiate primitive gates
    xor (S,x,y);
    and (C,x,y);
endmodule
```



Full-adder

```
//Description of full adder (see Fig 4-8)
module fulladder (S,C,x,y,z);
    input x,y,z;
    output S,C;
    wire S1,D1,D2; //Outputs of first XOR and two AND gates
//Instantiate the halfadder
    halfadder HA1 (S1,D1,x,y),
                HA2 (S,D2,S1,z);
    or g1(C,D2,D1);
endmodule
```

4-bit adder

```
//Description of 4-bit adder (see Fig 4-9)
module _4bit_adder (S,C4,A,B,C0);
    input [3:0] A,B;
    input C0;
    output [3:0] S;
    output C4;
    wire C1,C2,C3; //Intermediate carries
//Instantiate the fulladder
    fulladder FA0 (S[0],C1,A[0],B[0],C0),
                FA1 (S[1],C2,A[1],B[1],C1),
                FA2 (S[2],C3,A[2],B[2],C2),
                FA3 (S[3],C4,A[3],B[3],C3);

endmodule
```

Three state gates

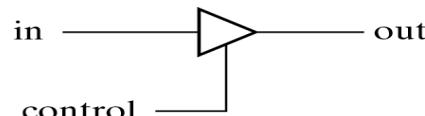
Gates statement: gate name(output, input, control)

>> **bufif1(OUT, A, control);**

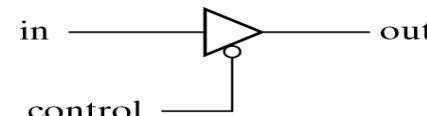
A = OUT when control = 1, OUT = z when control = 0;

>> **notif0(Y, B, enable);**

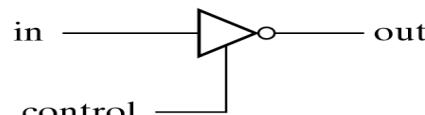
Y = B' when enable = 0, Y = z when enable = 1;



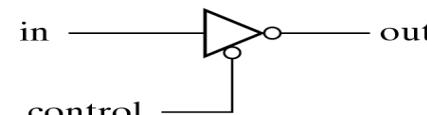
bufif1



bufif0



notif1



notif0

Fig. 4-31 Three-State Gates

2-to-1 multiplexer

- HDL uses the keyword **tri** to indicate that the output has multiple drivers.

```
module muxtri (A, B, select, OUT);
    input A,B,select;
    output OUT;
    tri OUT;
    bufif1 (OUT,A,select);
    bufif0 (OUT,B,select);
endmodule
```

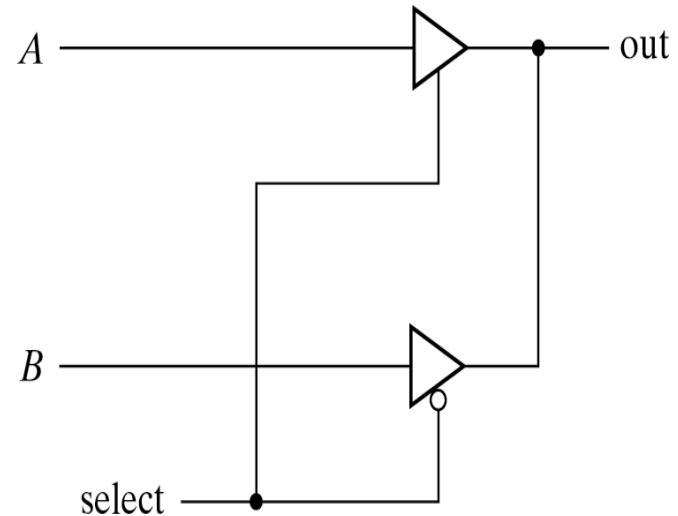
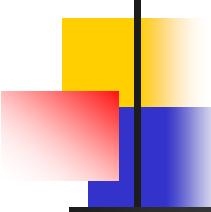


Fig. 4-32 2-to-1-Line Multiplexer with Three-State Buffers

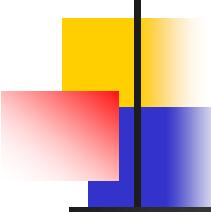


Dataflow modeling

- It uses a number of operators that act on operands to produce desired results. Verilog HDL provides about 30 operator types.

Table 4-10

Symbol	Operation
+	binary addition
-	binary subtraction
&	bit-wise AND
	bit-wise OR
^	bit-wise XOR
~	bit-wise NOT
==	equality
>	greater than
<	less than
{ }	concatenation
?:	conditional

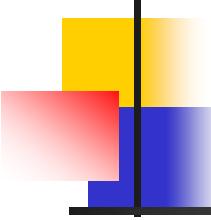


Dataflow modeling

- A continuous **assignment** is a statement that assigns a value to a net.
- The data type **net** is used in Verilog HDL to represent a physical connection **between circuit elements**.
- A **net** defines a gate output declared by an **output** or **wire**.

HDL Example 4-3

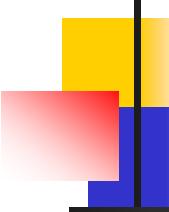
```
//Dataflow description of a 2-to-4-line decoder
//See Fig. 4-19
module decoder_df (A,B,E,D);
    input A,B,E;
    output [0:3] D;
    assign D[0] = ~(~A & ~B & ~E),
              D[1] = ~(~A & B & ~E),
              D[2] = ~(A & ~B & ~E),
              D[3] = ~(A & B & ~E);
endmodule
```



Dataflow description of 4-bit adder

HDL Example 4-4

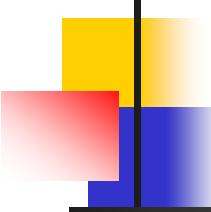
```
//Dataflow description of 4-bit adder
module binary_adder (A,B,Cin,SUM,Cout);
input [3:0] A,B;
input Cin;
output [3:0] SUM;
output Cout;
assign {Cout,SUM} = A + B +Cin;
endmodule
```



Data flow description of a 4-bit comparator

HDL Example 4-5

```
//Dataflow description of a 4-bit comparator.module magcomp (A,B,ALSB,AGTB,AEQB);  
    input [3:0] A,B;  
    output ALSB,AGTB,AEQB;  
    assign ALSB=(A < B),  
           AGTB = (A > B),  
           AEQB = (A == B);  
endmodule
```

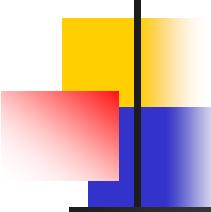


Dataflow description of 2-1 multiplexer

- **Conditional operator(? :)**
- Condition? true-expression : false-expression;

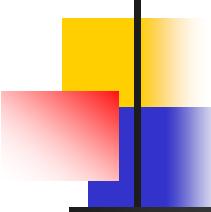
HDL Example 4-6

```
//Dataflow description of 2-to-1-line multiplexer
module mux2x1_df (A,B,select,OUT);
    input A,B,select;
    output OUT;
    assign OUT = select ? A : B;
endmodule
```



Behavioral modeling

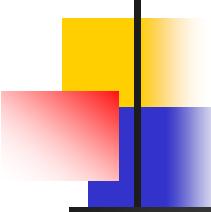
- It is used mostly to describe **sequential circuits**, but can be used also to describe **combinational circuits**.
- **Behavioral** descriptions use the keyword **always** followed by a list of procedural assignment statements.
- The **target output** of procedural assignment statements must be of the **reg** data type. Contrary to the **wire** data type, where the target output of an assignment may be continuously updated, a **reg** data type **retains its value until a new value is assigned**.



Behavioral description of 2-1 multiplexer

HDL Example 4-7

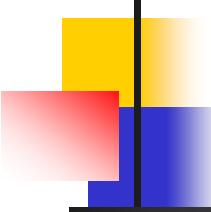
```
//Behavioral description of 2-to-1-line multiplexer
module mux2x1_bh(A,B,select,OUT);
    input A,B,select;
    output OUT;
    reg OUT;
    always @ (select or A or B)
        if (select == 1) OUT = A;
        else OUT = B;
endmodule
```



4-to-1-line Multiplexer

HDL Example 4-8

```
//Behavioral description of 4-to-1- line multiplexer
//Describes the function table of Fig. 4-25(b).
module mux4x1_bh (i0,i1,i2,i3,select,y);
    input i0,i1,i2,i3;
    input [1:0] select;
    output y;
    reg y;
    always @ (i0 or i1 or i2 or i3 or select)
        case (select)
            2'b00: y = i0;
            2'b01: y = i1;
            2'b10: y = i2;
            2'b11: y = i3;
    endcase
endmodule
```

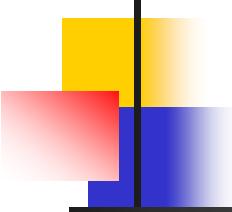


Writing a simple test bench

- In addition to the always statement, test benches use the initial statement to provide stimulus to the circuit under test.
- The **always** statement executes **repeatedly** in a loop. The **initial** statement executes **only once** starting from simulation **time=0** and may continue with any operations that are **delayed by a given number of time units** as specified by the **symbol #**.

For example:

```
initial
begin
    A = 0; B = 0;
    #10 A = 1;
    #20 A = 0; B = 1;
end
```



Stimulus and design modules interaction

- The signals of test bench as inputs to the design module are **reg** data type, the outputs of design module to test bench are **wire** data type.

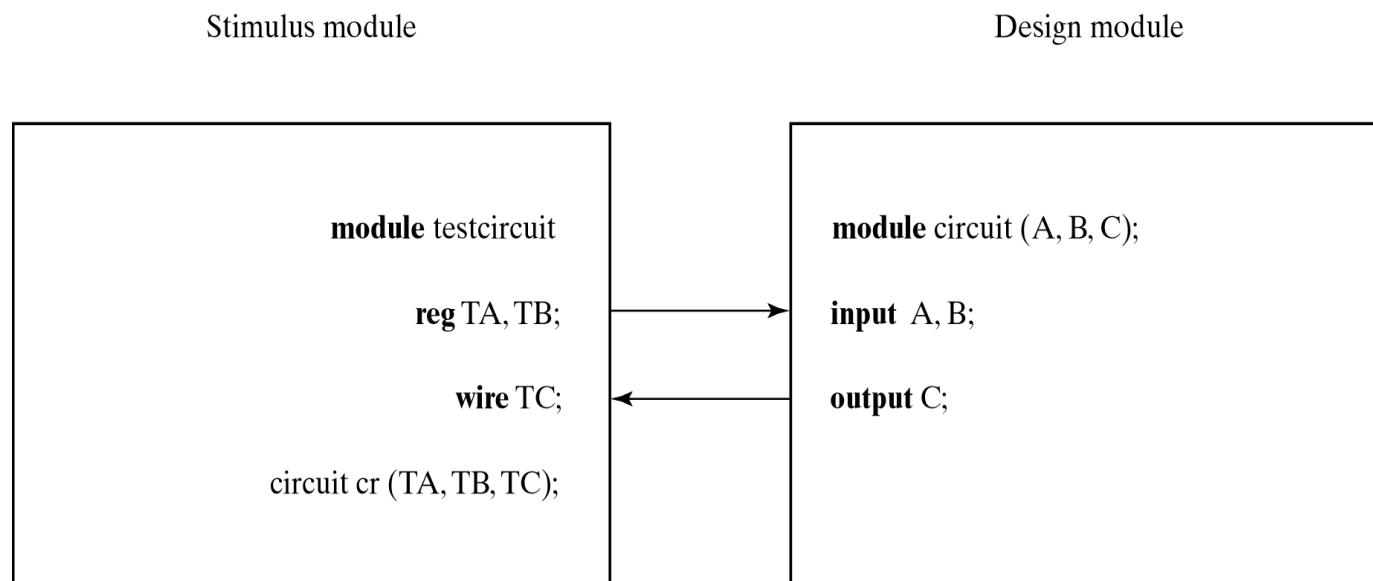


Fig. 4-33 Stimulus and Design Modules Interaction

Example 4-9

HDL Example 4-9

```
//Stimulus for mux2x1_df.
module testmux;
    reg TA,TB,TS; //inputs for mux
    wire Y; //output from mux
    mux2x1_df mx (TA,TB,TS,Y); // instantiate mux
    initial
        begin
            TS = 1; TA = 0; TB = 1;
            #10 TA = 1; TB = 0;
            #10 TS = 0;
            #10 TA = 0; TB = 1;
        end
    initial
        $monitor("select = %b A = %b B = %b OUT = %b time = %0d",
                 TS, TA, TB, Y, $time);
endmodule

//Dataflow description of 2-to-1-line multiplexer
//from Example 4-6
module mux2x1_df (A,B,select,OUT);
    input A,B,select;
    output OUT;
    assign OUT = select ? A : B;
endmodule

Simulation log:
select = 1 A = 0 B = 1 OUT = 0 time = 0
select = 1 A = 1 B = 0 OUT = 1 time = 10
select = 0 A = 1 B = 0 OUT = 0 time = 20
select = 0 A = 0 B = 1 OUT = 1 time = 30
```

Gate Level of Verilog Code of Fig.4-2

HDL Example 4-10

```
//Gate-level description of circuit of Fig. 4-2
module analysis (A,B,C,F1,F2);
    input A,B,C;
    output F1,F2;
    wire T1,T2,T3,F2not,E1,E2,E3;
    or g1 (T1,A,B,C);
    and g2 (T2,A,B,C);
    and g3 (E1,A,B);
    and g4 (E2,A,C);
    and g5 (E3,B,C);
    or g6 (F2,E1,E2,E3);
    not g7 (F2not,F2);
    and g8 (T3,T1,F2not);
    or g9 (F1,T2,T3);
endmodule
```

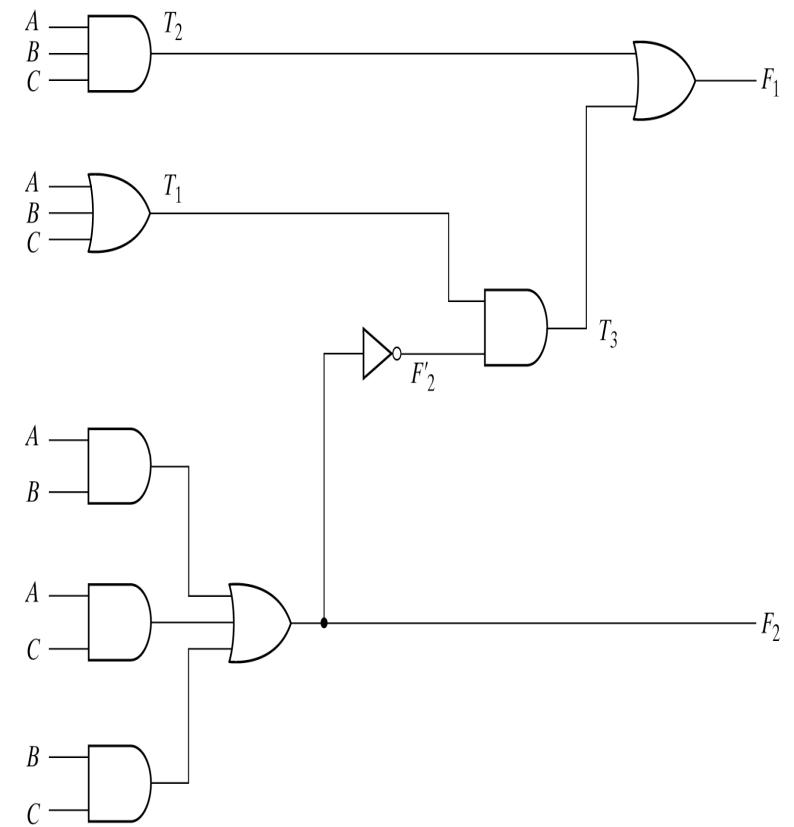


Fig. 4.2 Logic Diagram for Analysis Example

Test Bench of the Figure 4-2

```
//Stimulus to analyze the circuit
module test_circuit;
    reg [2:0]D;
    wire F1,F2;
    analysis fig42(D[2],D[1],D[0],F1,F2);
    initial
        begin
            D = 3'b000;
            repeat(7)
                #10 D = D + 1'b1;
        end
    initial
        $monitor ("ABC = %b F1 = %b F2 =%b ",D, F1, F2);
endmodule
```

Simulation log:

```
ABC = 000 F1 = 0 F2 =0
ABC = 001 F1 = 1 F2 =0
ABC = 010 F1 = 1 F2 =0
ABC = 011 F1 = 0 F2 =1
ABC = 100 F1 = 1 F2 =0
ABC = 101 F1 = 0 F2 =1
ABC = 110 F1 = 0 F2 =1
ABC = 111 F1 = 1 F2 =1
```