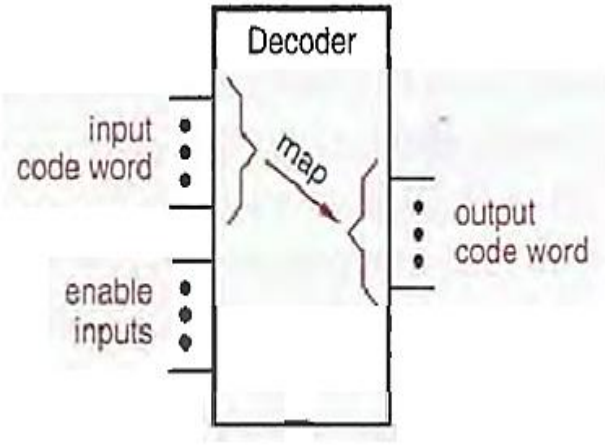
UNIT-4 COMBINATIONAL LOGIC DESIGN

DECODERS:

A decoder is a multiple-input, multiple-output logic circuit that converts coded inputs into coded outputs, where the input and output codes are different. The input code generally has fewer bits than the output code, and there is a one-to one mapping from input code words into output code words. In a one-to-one mapping, each input code word produces a different output code word.



The general structure of a decoder circuit is shown in Figure 1. The enable inputs, if present, must be asserted for the decoder to perform its normal mapping function. Otherwise, the decoder maps all input code words into a single, ―disabled,‖ output code word.

The most commonly used output code is a 1-out-of-m code, which contains m bits, where one bit is asserted at any time. Thus, in a 1-out-of-4 code with active-high outputs, the code words are 0001, 0010, 0100, and 1000. With active-low outputs, the code words are 1110, 1101, 1011, and 0111.

BINARY DECODERS

The most common decoder circuit is an n-to-2n decoder or binary decoder. Such a decoder has an n-bit binary input code and a 1-out-of-2n output code. A binary decoder is used when you need to activate exactly one of 2n outputs based on an n-bit input value.

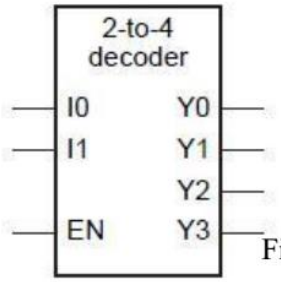


Fig 3: 2 to 4 decoder logic symbo

Table 1 is the truth table of a 2-to-4 decoder. The input code word I1,I0 represents an integer in the range 0–3. The output code word Y3,Y2,Y1,Y0 has Yi equal to 1 if and only if the input code word is the binary representation of i and the enable input EN is 1. If EN is 0, then all of the outputs are 0. A gate-level circuit for the 2-to-4 decoder is shown in Figure 2 Each AND gate decodes one combination of the input code word I1,I0.

Inputs			*Outputs*			
EN	I1	I0	Y3	Y2	Y1	Y0
0	x	x	0	0	0	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

Table 1: 2 to 4 decoder

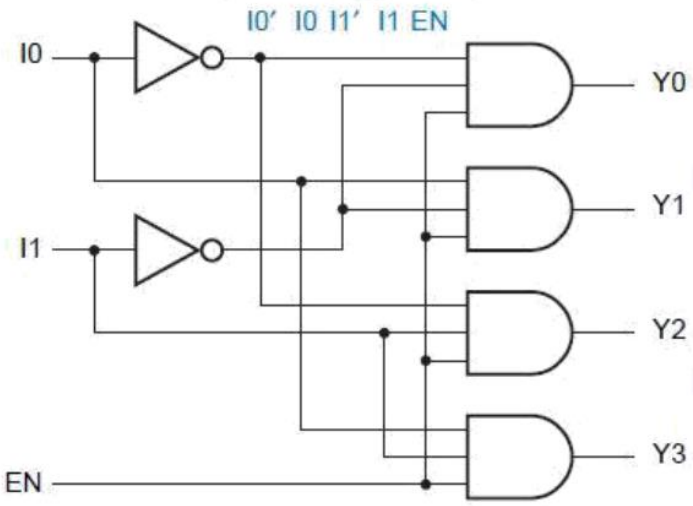
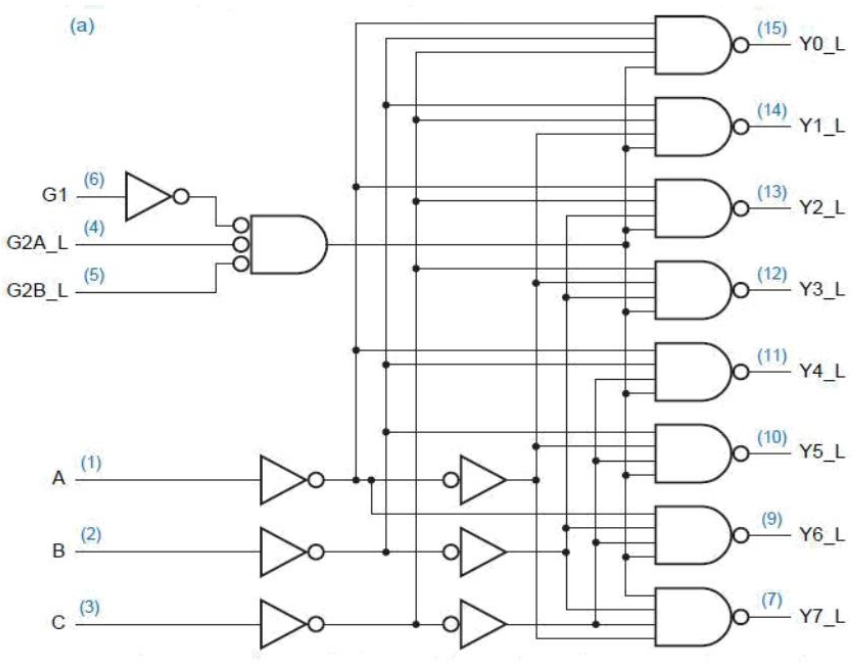


Fig 4: logic diagram of 2 to 4 decoder

74x138 3-to-8 Decoder

The 74x138 is a commercially available MSI 3-to-8 decoder whose gate-level circuit diagram and symbol are shown in Figure 7; its truth table is given in Table. Like the 74x139, the 74x138 has active-low outputs, and it has three enable inputs (G1, /G2A, /G2B), all of which must be asserted for the selected output to be asserted.

The logic function of the 74X138 is straightforward—an output is asserted if and only if the decoder is enabled and the output is selected.



Inputs						Outputs							
G1	G2A_L	G2B_L	C	B	A	Y7_L	Y6_L	Y5_L	Y4_L	Y3_L	Y2_L	Y1_L	Y0_L
0	x	x	x	x	x	1	1	1	1	1	1	1	1
x	1	x	x	x	x	1	1	1	1	1	1	1	1
x	x	1	x	x	x	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	1	1	1	1	1	0
1	0	0	0	0	1	1	1	1	1	1	1	0	1
1	0	0	0	1	0	1	1	1	1	1	0	1	1
1	0	0	0	1	1	1	1	1	1	0	1	1	1
1	0	0	1	0	0	1	1	1	0	1	1	1	1
1	0	0	1	0	1	1	1	0	1	1	1	1	1
1	0	0	1	1	0	1	0	1	1	1	1	1	1
1	0	0	1	1	1	0	1	1	1	1	1	1	1

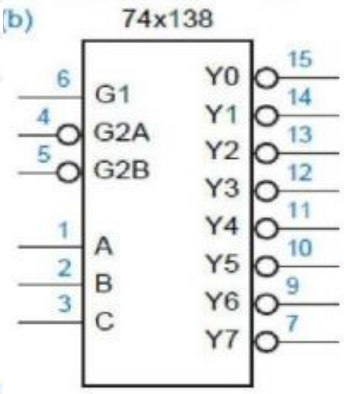


Figure 7: logic symbol of 74X138

Thus, we can easily write logic equations for an internal output signal such as Y5 in terms of the internal input signals:

$$Y5 = \underbrace{G1 \cdot G2A \cdot G2B}_{\text{enable}} \cdot \underbrace{C \cdot B' \cdot A}_{\text{select}}$$

However, because of the inversion bubbles, we have the following relations between internal and external signals:

$$G2A = G2A\_L'$$
$$G2B = G2B\_L'$$
$$Y5 = Y5\_L'$$

Therefore, if we're interested, we can write the following equation for the external output signal Y5_L in terms of external input signals:

$$Y5\_L = Y5' = (G1 \cdot G2A\_L' \cdot G2B\_L' \cdot C \cdot B' \cdot A)'$$
$$= G1' + G2A\_L + G2B\_L + C' + B + A'$$

On the surface, this equation doesn't resemble what you might expect for a decoder, since it is a logical sum rather than a product. However, if you practice bubble-to-bubble logic design, you don't have to worry

about this; you just give the output signal an active-low name and remember that it's active low when you connect it to other inputs.

The 74x139 Dual 2-to-4 Decoder:

Two independent and identical 2-to-4 decoders are contained in a single MSI part, the 74x139. The gate-level circuit diagram for this IC is shown in Figure 5.

1. The outputs and the enable input of the '139 are active-low.

2. Most MSI decoders were originally designed with active-low outputs, since TTL inverting gates are generally faster than non inverting ones.

3. '139 has extra inverters on its select inputs. Without these inverters, each select input would present three AC or DC loads instead of one, consuming much more of the fanout budget of the device that drives it.

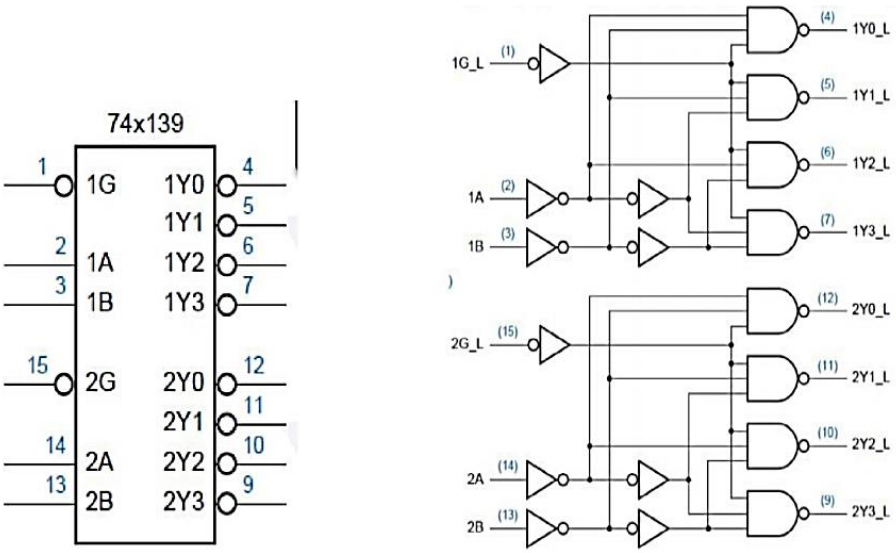


Figure 6 The 74x139 dual 2-to-4 decoder: (a) traditional logic symbol (b) logic diagram, including pin numbers for a standard 16-pin dual in-line package;

In this case, the assignment of the generic function to one half or the other of a particular '139 package can be deferred until the schematic is completed Table 5-6 is the truth table for a 74x139-type decoder.

CASCADING BINARY DECODERS

Multiple binary decoders can be used to decode larger code words. Figure 5-38 shows how two 3-to-8 decoders can be combined to make a 4-to-16 decoder. The availability of both active-high and active-low enable inputs on the 74x138 makes it possible to enable one or the other directly based on the state of the most significant input bit. The top decoder (U1) is enabled when N3 is 0, and the bottom one (U2) is enabled when N3 is 1.

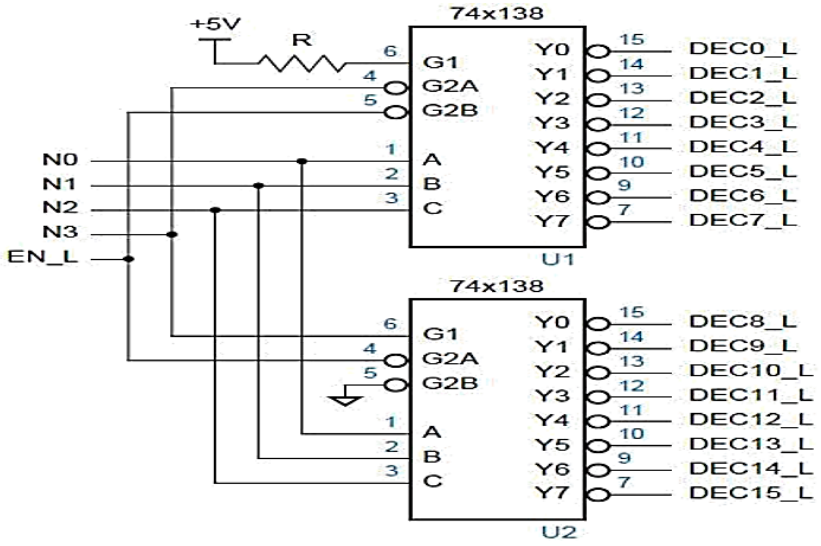
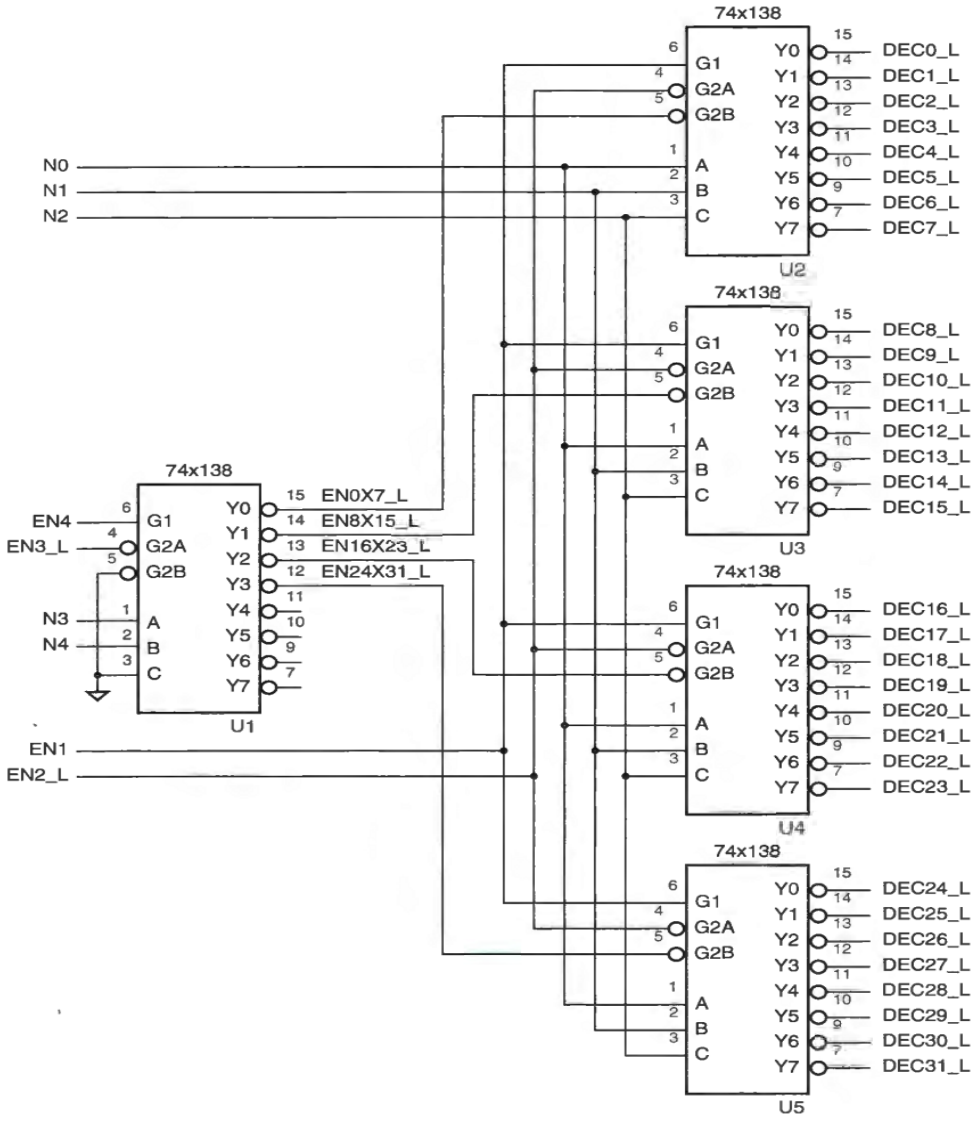


Figure 5-38 Design of a 4-to-16 decoder using 74x138s.

EXAMPLE 1: DESIGN 5 TO 32 DECODER USING 74X138



EXAMPLE VHDL PROGRAM FOR DECODER USING DIFFERENT MODELLING STYLE

Table 6-13 VHDL structural program for the decoder in Figure 6-32.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity V2to4dec is
  port (I0, I1, EN: in STD_LOGIC;
        Y0, Y1, Y2, Y3: out STD_LOGIC );
end V2to4dec;

architecture V2to4dec_s of V2to4dec is
  signal NOTI0, NOTI1: STD_LOGIC;
  component inv port (I: in STD_LOGIC; O: out STD_LOGIC ); end component;
  component and3 port (I0, I1, I2: in STD_LOGIC; O: out STD_LOGIC ); end component;
begin
  U1: inv port map (I0,NOTI0);
  U2: inv port map (I1,NOTI1);
  U3: and3 port map (NOTI0,NOTI1,EN,Y0);
  U4: and3 port map (   I0,NOTI1,EN,Y1);
  U5: and3 port map (NOTI0,   I1,EN,Y2);
  U6: and3 port map (   I0,   I1,EN,Y3);
end V2to4dec_s;
```

Table 6-14 Dataflow-style VHDL program for a 74x138-like 3-to-8 binary decoc

```
library IEEE;
use IEEE.std_logic_1164.all;

entity V74x138 is
    port (G1, G2A_L, G2B_L: in STD_LOGIC;       -- enable inputs
          A: in STD_LOGIC_VECTOR (2 downto 0);   -- select inputs
          Y_L: out STD_LOGIC_VECTOR (0 to 7) );  -- decoded outputs
 end V74x138;

architecture V74x138_a of V74x138 is
  signal Y_L_i: STD_LOGIC_VECTOR (0 to 7);
begin
  with A select Y_L_i <=
    "01111111" when "000",
    "10111111" when "001",
    "11011111" when "010",
    "11101111" when "011",
    "11110111" when "100",
    "11111011" when "101",
    "11111101" when "110",
    "11111110" when "111",
    "11111111" when others;
  Y_L <= Y_L_i when (G1 and not G2A_L and not G2B_L)='1' else "11111111";
end V74x138_a;
```

Table 6-15 VHDL architecture with a maintainable approach to active-level handling.

```
architecture V74x138_b of V74x138 is
  signal G2A, G2B: STD_LOGIC;              -- active-high version of inputs
  signal Y: STD_LOGIC_VECTOR (0 to 7);     -- active-high version of outputs
  signal Y_s: STD_LOGIC_VECTOR (0 to 7);   -- internal signal
begin
  G2A <= not G2A_L; -- convert inputs
  G2B <= not G2B_L; -- convert inputs
  Y_L <= not Y;     -- convert outputs
  with A select Y_s <=
    "10000000" when "000",
    "01000000" when "001",
    "00100000" when "010",
    "00010000" when "011",
    "00001000" when "100",
    "00000100" when "101",
    "00000010" when "110",
    "00000001" when "111",
    "00000000" when others;
  Y <= Y_s when (G1 and G2A and G2B)='1' else "00000000";
end V74x138_b;
```

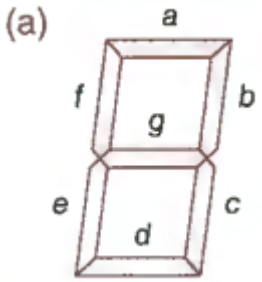
Table 6-18 Behavioral-style architecture definition for a 3-to-8 decoder.

```
architecture V3to8dec_b of V3to8dec is
  signal Y_s: STD_LOGIC_VECTOR (0 to 7);
begin
process(A, G1, G2, G3, Y_s)
  begin
    case A is
      when "000" => Y_s <= "10000000";
      when "001" => Y_s <= "01000000";
      when "010" => Y_s <= "00100000";
      when "011" => Y_s <= "00010000";
      when "100" => Y_s <= "00001000";
      when "101" => Y_s <= "00000100";
      when "110" => Y_s <= "00000010";
      when "111" => Y_s <= "00000001";
      when others => Y_s <= "00000000";
    end case;
    if (G1 and G2 and G3)='1' then Y <= Y_s;
    else Y <= "00000000";
    end if;
  end process;
end V3to8dec_b;
```

SEVEN-SEGMENT DECODERS

Look at your wrist and you'll probably see a seven-segment display. This type of display, which normally uses light-emitting diodes (LEDs) or liquid-crystal display (LCD) elements, is used in watches, calculators, and instruments to display decimal data.

A seven-segment decoder has 4-bit BCD as its input code and the "sevenencoder 2n-to-n encoder binary encoder segment code," which is . graphically depicted in Figure 6-44(b), as its output code. Table 6-26 is a Verilog program for a seven-segment decoder with 4-bit BCD input A-D (D being the MSB), active-high enable input EN, and segment outputs SEGA-SEGG. Note the use of concatenation and an auxiliary variable SEGS to make the program more readable. The program can be easily modified for different encodings and features, for example, to add "tails" to digits 6 and 9 (Exercise 6.47) or to display hexadecimal digits A-F instead of treating these input combinations as "don't cares"



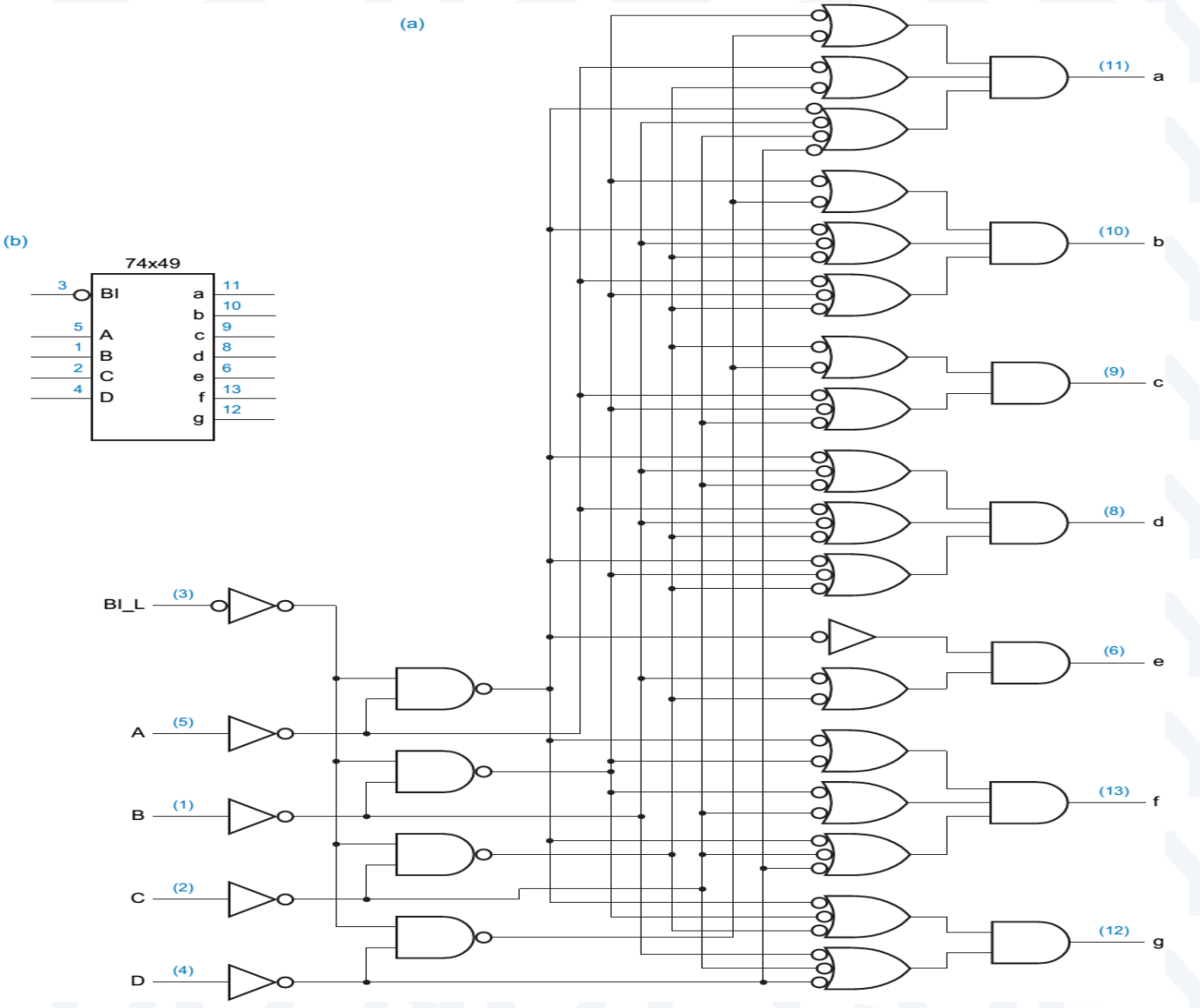


Figure 5-44 The 74x49 seven-segment decoder: (a) logic diagram, including pin numbers; (b) traditional logic symbol.

Table 5-20 Truth table for a 74x49 seven-segment decoder.

Inputs					Outputs						
BI_L	D	C	B	A	a	b	c	d	e	f	g
0	x	x	x	x	0	0	0	0	0	0	0
1	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
1	0	0	1	0	1	1	0	1	1	0	1
1	0	0	1	1	1	1	1	1	0	0	1
1	0	1	0	0	0	1	1	0	0	1	1
1	0	1	0	1	1	0	1	1	0	1	1
1	0	1	1	0	0	0	1	1	1	1	1
1	0	1	1	1	1	1	1	0	0	0	0
1	1	0	0	0	1	1	1	1	1	1	1
1	1	0	0	1	1	1	1	0	0	1	1
1	1	0	1	0	0	0	0	1	1	0	1
1	1	0	1	1	0	0	1	1	0	0	1
1	1	1	0	0	0	1	0	0	0	1	1
1	1	1	0	1	1	0	0	1	0	1	1
1	1	1	1	0	0	0	0	1	1	1	1
1	1	1	1	1	0	0	0	0	0	0	0

Example 4.9 : *Write a VHDL source code for BCD to 7-segment decoder.*

Solution :

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
    ENTITY seg7 IS
        PORT ( bcd    : IN STD_LOGIC_VECTOR(3 DOWNTO 0)
               leds   : OUT STD_LOGIC_VECTOR(1 TO 7));
        END seg7;
    ARCHITECTURE BCD7SEG OF seg7 IS
    BEGIN
        PROCESS (bcd)
        BEGIN
            CASE bed IS                              -- abcdefg
                WHEN "0000"  = > leds <="1111110";
                WHEN "0001"  = > leds <="0110000";
                WHEN "0010"  = > leds <="1101101";
                WHEN "0011"  = > leds <="1111001";
                WHEN "0100"  = > leds <="0110011";
                WHEN "0101"  = > leds <="1011011";

                WHEN "0110"  = > leds <="1011111";
                WHEN "0111"  = > leds <="1110000";
                WHEN "1000"  = > leds <="1111111";
                WHEN "1001"  = > leds <="1111011";
                WHEN OTHER   = > leds <= "-------";
    END CASE;
END PROCESS;
END BCD7SEG;
```

ENCODERS

A decoder's output code normally has more bits than its input code. If the device's output code has fewer bits than the input code, the device is usually called an encoder. Probably the simplest encoder to build is a 2^n-to-n or binary encoder. As shown in Figure 6-45(a), it has just the opposite function as a binary decoder— its input code is the 1-out-of-2^n code and its output code is n-bit binary.

The corresponding logic circuit is shown in (b). In general, a 2^n-to-n encoder can be built from n 2^n 1-input OR gates. Bit i of the input code is connected to OR gate j if bit j in the binary representation of i is 1.

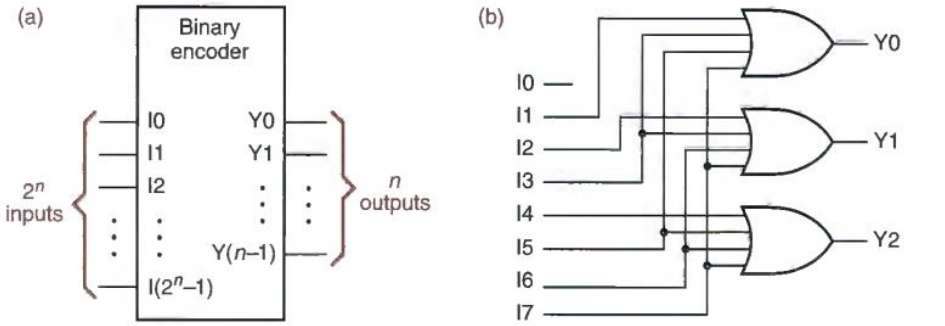


Figure 6-45
Binary encoder:
(a) general structure;
(b) 8-to-3 encoder.

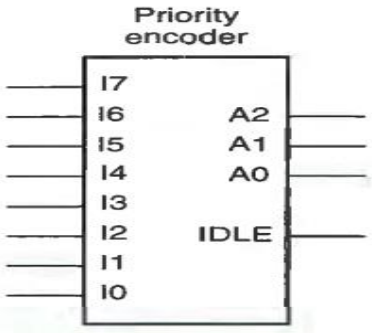
A Truth Table representing 8:3 Encoder.

INPUT								OUTPUT		
I7	I6	I5	I4	I3	I2	I1	I0	Y2	Y1	Y0
0	0	0	0	0	0	0	0	X	X	X
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

$$Y0 = I1 + I3 + I5 + I7$$
$$Y1 = I2 + I3 + I6 + I7$$
$$Y2 = I4 + I5 + I6 + I7$$

PRIORITY ENCODERS

The 1-out-of-2^n coded outputs of an n-bit binary decoder are generally used to control a set of 2^n devices, where at most one device is supposed to be active at any time. Conversely, consider a system with 2n inputs, each of which indicates a request for service. This structure is often found in microprocessor input/output subsystems, where the inputs might be interrupt requests.



the encoding device produces the number of the highest-priority requestor. Such a device is called a priority encoder. Input I7 has the highest priority. Outputs A2–A0 contain the number of the highest-priority asserted input, if any. The IDLE output is asserted if no inputs are asserted.

In order to write logic equations for the priority encoder's outputs, we first define eight intermediate variables H0–H7, such that Hi is 1 if and only if Ii is the highest priority 1 input: Using these signals, the equations for the A2–A0 outputs are similar to the ones for a simple binary encoder:

H7=I7 (Highest Priority)
H6=I6.I7'
H5=I5.I6'.I7'
H4=I4.I5'.I6'.I7'
H3=I3.I4'.I5'.I6'.I7'
H2=I2.I3'.I4'.I5'.I6'.I7'
H1=I1.I2'.I3'.I4'.I5'.I6'.I7'
H0=I0.I1'.I2'.I3'.I4'.I5'.I6'.I7'
IDLE=I0'.I1'.I2'.I3'.I4'.I5'.I6'.I7'

Using these signals, the equations for the A2–A0 outputs are similar to the ones for a simple binary encoder:

$$A2 = H4 + H5 + H6 + H7$$
$$A1 = H2 + H3 + H6 + H7$$
$$A0 = H1 + H3 + H5 + H7$$

THE 74X148 PRIORITY ENCODER

The 74x148 is a commercially available, MSI 8-input priority encoder it has an enable input, EI_L, that must be asserted for any of its outputs to be asserted. The complete truth table is given in Table 5-22. Instead of an IDLE output, the '148 has a GS_L output that is asserted when the device is enabled and one or more of the request inputs is asserted. The manufacturer calls this "Group Select" but it's easier to remember as "Got Something" The EO_L signal is an enable output designed to be connected to the EI_L input of another '148 that handles lower-priority requests. /EO is asserted if EI_L is asserted but no request input is asserted; thus, a lower-priority '148 may be enabled.

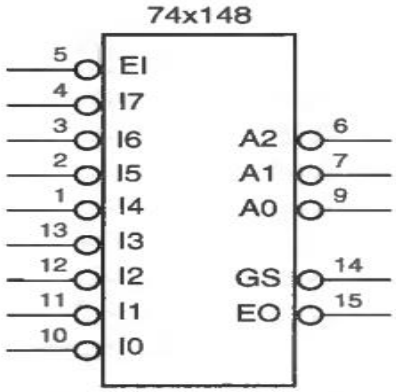


Figure 6-48
Logic symbol for the 74x148 8-input priority encoder.

Figure 6-49 shows how four 74x148s can be connected in this way to accept 32 request inputs and produce a 5-bit output, RA4-RAO, indicating the highest-priority requestor. Since the A2-AO outputs of at most one' 148 will be enabled at any time, the outputs of the individual' 148s can be ORed to produce RA2-RAO. Likewise, the individual GS_L outputs can be combined in a 4-to-2 encoder to produce RA4 and RA3. The RGS output is asserted if any GS output is asserted.

Table 6-27 Truth table for a 74x148 8-input priority encoder.

Inputs									*Outputs*				
EI_L	I0_L	I1_L	I2_L	I3_L	I4_L	I5_L	I6_L	I7_L	A2_L	A1_L	A0_L	GS_L	EO_L
1	x	x	x	x	x	x	x	x	1	1	1	1	1
0	x	x	x	x	x	x	x	0	0	0	0	0	1
0	x	x	x	x	x	x	0	1	0	0	1	0	1
0	x	x	x	x	x	0	1	1	0	1	0	0	1
0	x	x	x	x	0	1	1	1	0	1	1	0	1
0	x	x	x	0	1	1	1	1	1	0	0	0	1
0	x	x	0	1	1	1	1	1	1	0	1	0	1
0	x	0	1	1	1	1	1	1	1	1	0	0	1
0	0	1	1	1	1	1	1	1	1	1	1	0	1
0	1	1	1	1	1	1	1	1	1	1	1	1	0

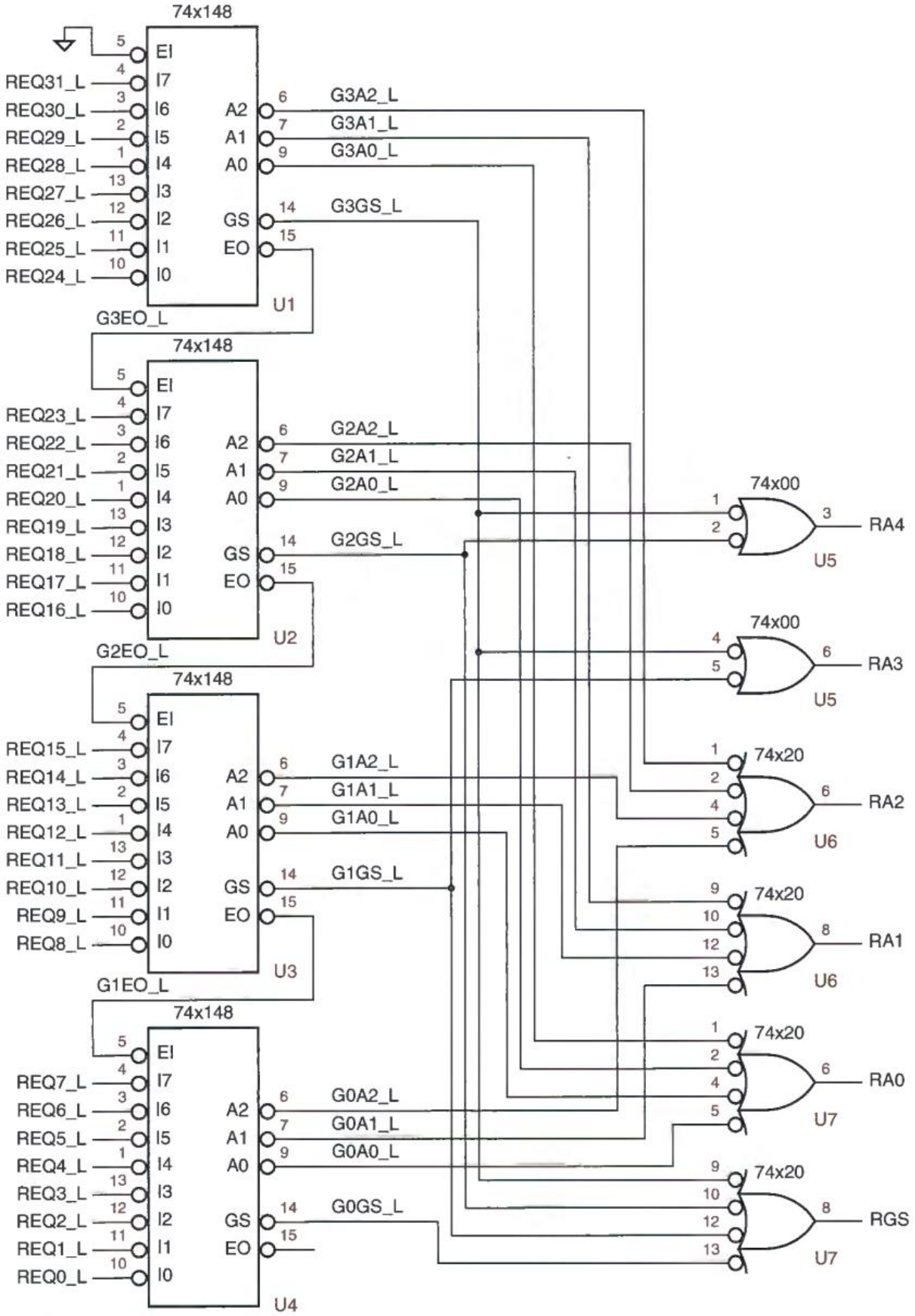


Figure 6-49 Four 74x148s cascaded to handle 32 requests.

PROGRAM FOR ENCODER

Table 6-30 Behavioral VHDL program for a 74x148-like 8-input priority encoder.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity V74x148 is
    port (
        EI_L: in STD_LOGIC;
        I_L: in STD_LOGIC_VECTOR (7 downto 0);
        A_L: out STD_LOGIC_VECTOR (2 downto 0);
        EO_L, GS_L: out STD_LOGIC
    );
end V74x148;

architecture V74x148p of V74x148 is
  signal EI: STD_LOGIC;                    -- active-high version of input
  signal I: STD_LOGIC_VECTOR (7 downto 0); -- active-high version of inputs
  signal EO, GS: STD_LOGIC;                -- active-high version of outputs
  signal A: STD_LOGIC_VECTOR (2 downto 0); -- active-high version of outputs
begin
  process (EI_L, I_L, EI, EO, GS, I, A)
  variable j: INTEGER range 7 downto 0;
  begin
    EI <= not EI_L; -- convert input
    I <= not I_L;   -- convert inputs
    EO <= '1'; GS <= '0'; A <= "000";
    if (EI)='0' then EO <= '0';
    else for j in 7 downto 0 loop
        if I(j)='1' then
          GS <= '1'; EO <= '0'; A <= CONV_STD_LOGIC_VECTOR(j,3);
          exit;
        end if;
      end loop;
    end if;
    EO_L <= not EO; -- convert output
    GS_L <= not GS; -- convert output
    A_L <= not A;   -- convert outputs
  end process;
end V74x148p;
```

THREE-STATE DEVICES

Three-State Buffers

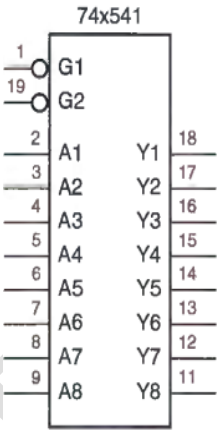
The most basic three-state device is a three-state buffer, often called a three-state driver.

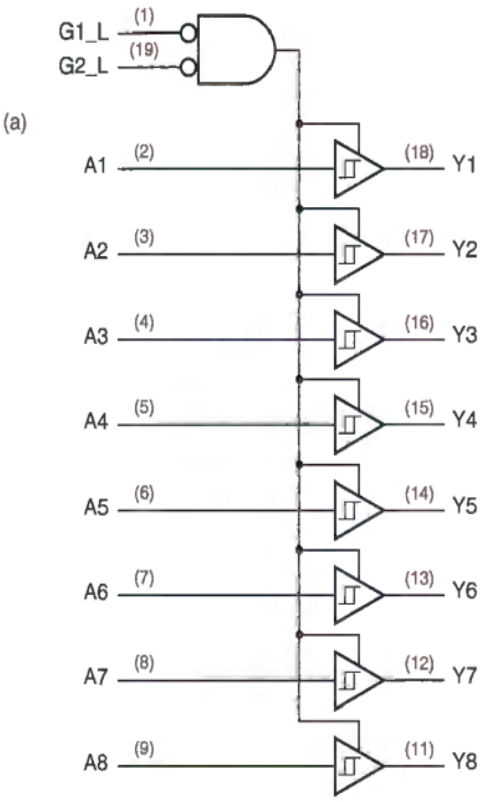
The logic symbols for four physically different three-state buffers are shown in Figure

The basic symbol is that of a noninverting buffer (a, b) or an inverter (c, d). The extra signal at the top of the symbol is a three-state enable input, which may be active high (a, c) or active low (b, d). When the enable input is asserted,

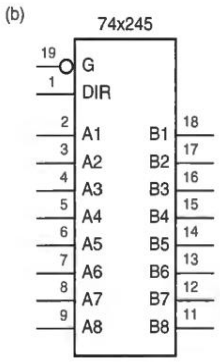
the device behaves like an ordinary buffer or inverter. When the enable input is negated, the device output ―floats‖; that is, it goes to a high impedance (Hi-Z), disconnected state and functionally behaves as if it weren't even there.

Both enable inputs, G1_L and G2_L, must be asserted to enable the device's three-state outputs. The little rectangular symbols inside the buffer symbols indicate hysteresis, an electrical characteristic of the inputs that improves noise immunity. The 74x541 inputs typically have 0.4 volts of hysteresis.





Following figure shows the logic diagram and symbol for a 74x245 octal ***three-state transceiver***. The DIR input determines the direction 74x245 of transfer, from A to B (DIR= 1) or from B to A (DIR= 0). The three-state buffer for the selected direction is enabled only if G_L is asserted.



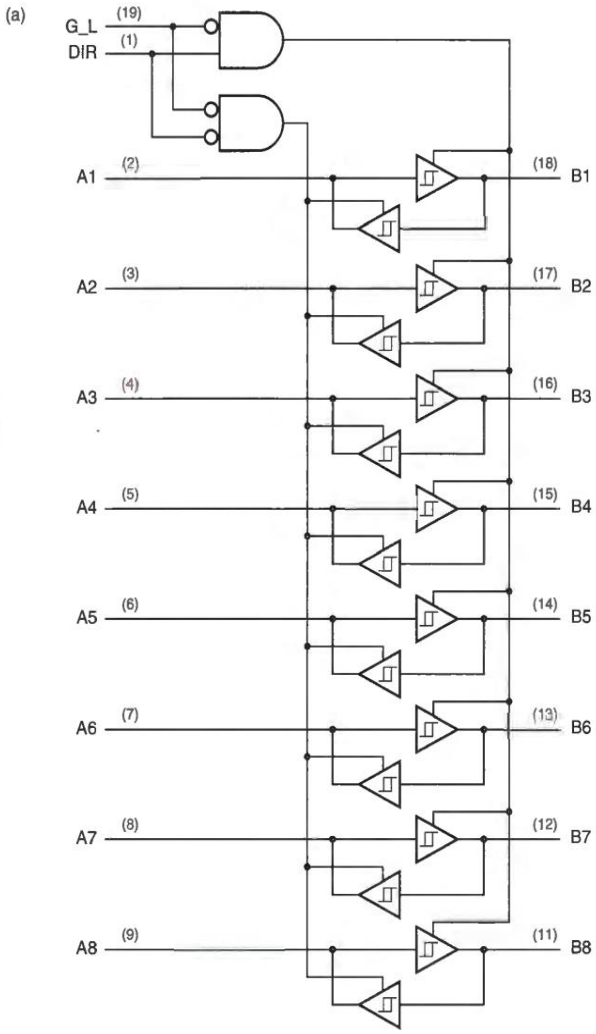
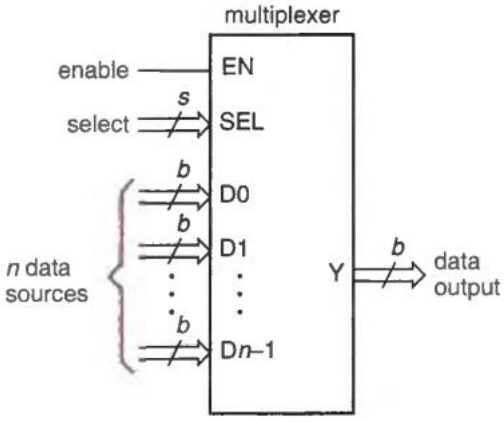


Figure 6-56
The 74x245 octal three-state transceiver:
(a) logic diagram;
(b) traditional logic symbol.

Table 6-38 VHDL program with four 8-bit three-state drivers.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity V3statex is
    port (
        G_L: in STD_LOGIC;                          -- Global output enable
        SEL: in STD_LOGIC_VECTOR (1 downto 0);      -- Input select 0,1,2,3 ==> A,B,C,D
        A, B, C, D: in STD_LOGIC_VECTOR (1 to 8); -- Input buses
        X: out STD_LOGIC_VECTOR (1 to 8)            -- Output bus (three-state)
    );
end V3statex;

architecture V3states of V3statex is
begin
  process (G_L, SEL, A)
  begin
    if G_L='0' and SEL = "00" then X <= A;
    else X <= (others => 'Z');
    end if;
  end process;

  process (G_L, SEL, B)
  begin
    if G_L='0' and SEL = "01" then X <= B;
    else X <= (others => 'Z');
    end if;
  end process;

  process (G_L, SEL, C)
  begin
    if G_L='0' and SEL = "10" then X <= C;
    else X <= (others => 'Z');
    end if;
  end process;

  process (G_L, SEL, D)
  begin
    if G_L='0' and SEL = "11" then X <= D;
    else X <= (others => 'Z');
    end if;
  end process;

end V3states;
```

MULTIPLEXERS



A multiplexer is a digital switch—it connects data from one of n sources to its output. Figure 5-61(a) shows the inputs and outputs of an n-input, b-bit multiplexer. There are n sources of data, each of which is b bits wide. A multiplexer is often called a mux for short. A multiplexer can use addressing bits to select one of several input bits to be the output. A selector chooses a single data input and passes it to the mux output.it has one output selected at a time.

Figure shows a switch circuit that is roughly equivalent to the multiplexer. However, unlike a mechanical switch, a multiplexer is a unidirectional device: information flows only from inputs (on the left) to outputs (on the right. Multiplexers are obviously useful devices in any application in which data must be switched from multiple sources to a destination. A common application in computers is the multiplexer between the processor's registers and its arithmetic logic unit (ALU). For example, consider a 16-bit processor in which each instruction has a 3-bit field that specifies one of eight registers to use. This 3-bit field is connected

to the select inputs of an 8-input, 16-bit multiplexer. The multiplexer's data inputs are connected to the eight registers, and its data outputs are connected to the ALU to execute the instruction using the selected register.

STANDARD MSI MULTIPLEXERS

The sizes of commercially available MSI multiplexers are limited by the number of pins available in an inexpensive IC package. Commonly used muxes come in 16-pin packages. Shown in fig which selects among eight 1-bit inputs. The select inputs are named C, B, and A, where C is most significant numerically. The enable input EN_L is active low; both active-high (Y) and active-low (Y_L) versions of the output are provided.

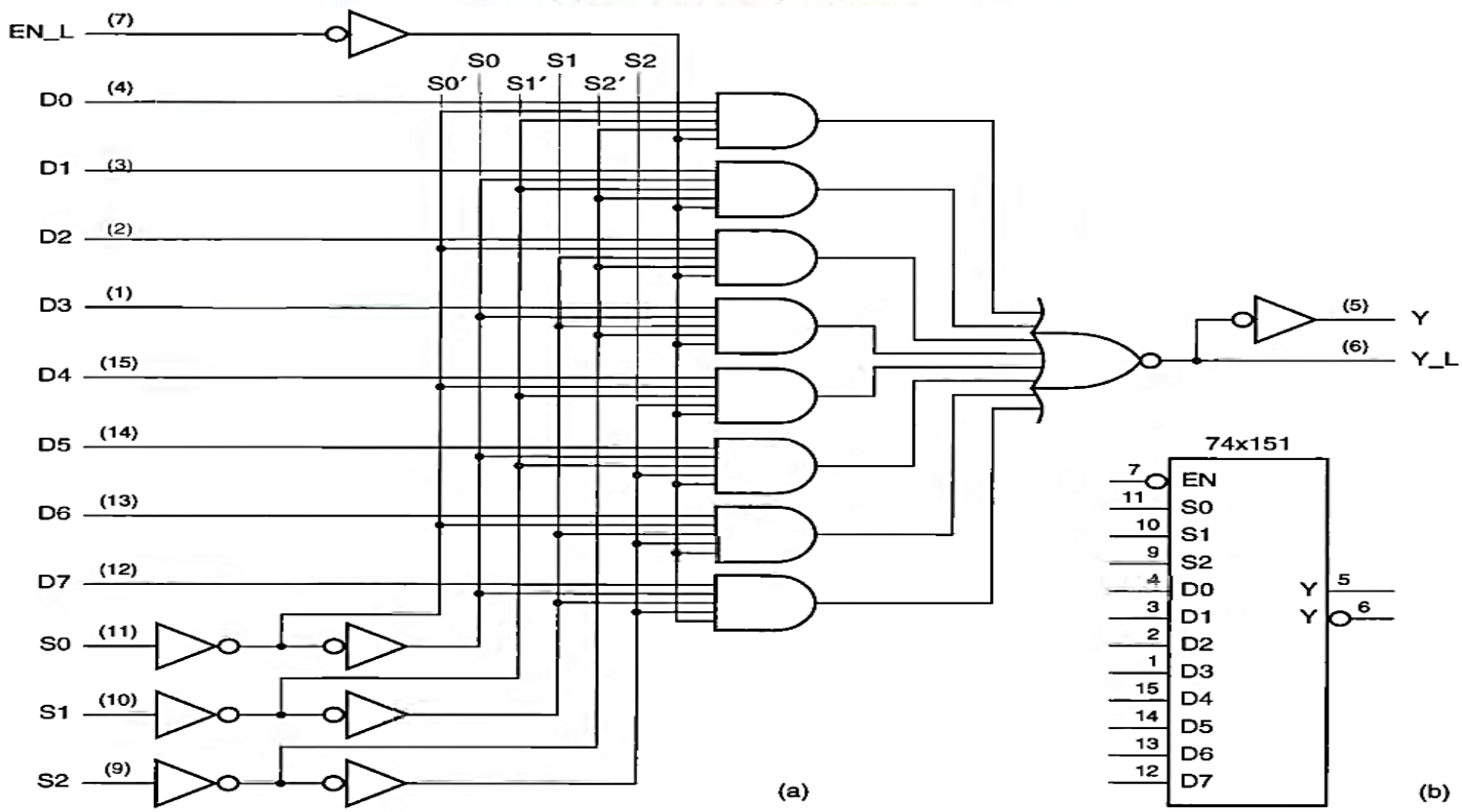


Table 6-42
Truth table for a 74x151 8-input, 1-bit multiplexer.

Inputs				*Outputs*	
EN_L	S2	S1	S0	Y	Y_L
1	x	x	x	0	1
0	0	0	0	D0	D0′
0	0	0	1	D1	D1′
0	0	1	0	D2	D2′
0	0	1	1	D3	D3′
0	1	0	0	D4	D4′
0	1	0	1	D5	D5′
0	1	1	0	D6	D6′
0	1	1	1	D7	D7′

At the other extreme of muxes in 16-pin packages, we have the 74x157, shown in Figure, which selects between two 4-bit inputs. Just to confuse things, the manufacturer has named the select input S and the active-low enable input G_L. Also note that the data sources are named A and B

Figure 6-61 The 74x157 2-input, 4-bit multiplexer: (a) logic diagram; (b) traditional logic symbol.

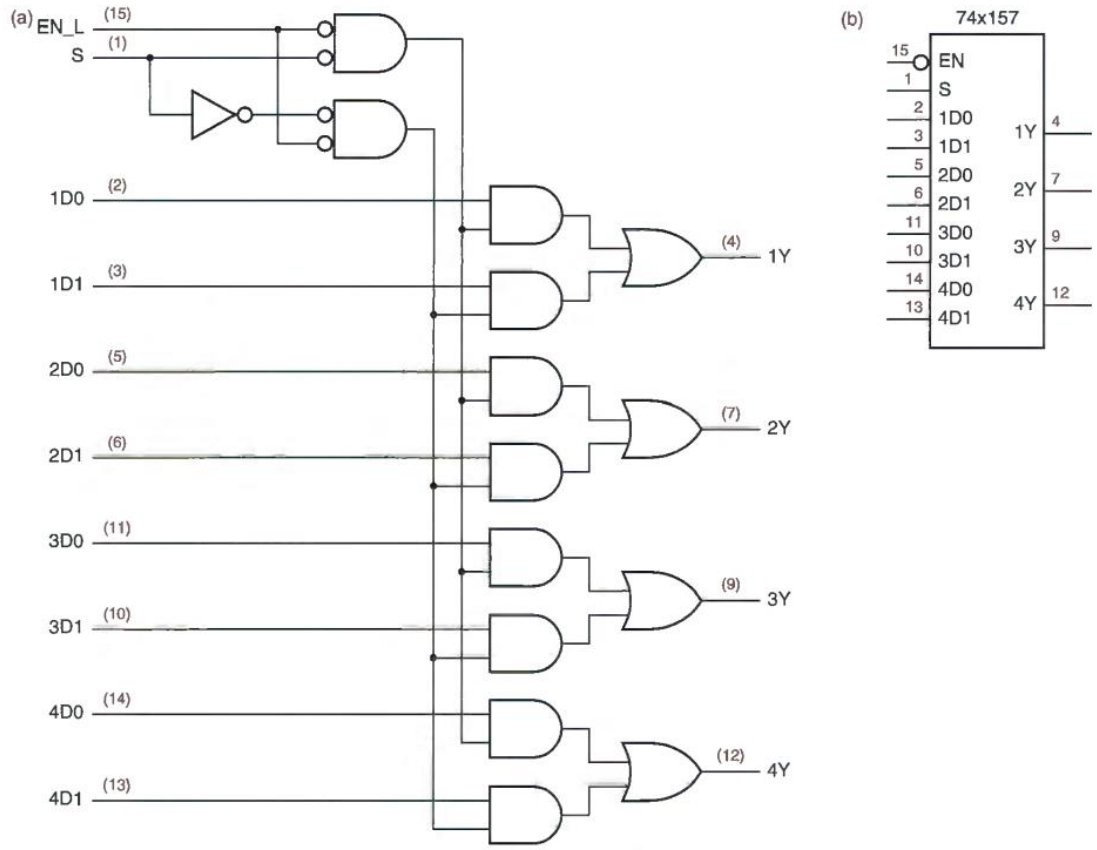


Table 6-43 Truth table for a 74x157 2-input, 4-bit multiplexer.

Inputs		*Outputs*			
EN_L	S	1Y	2Y	3Y	4Y
1	x	0	0	0	0
0	0	1D0	2D0	3D0	4D0
0	1	1D1	2D1	3D1	4D1

EXPANDING MULTIPLEXERS

Seldom does the size of an MSI multiplexer match the characteristics of the problem at hand. For example, we suggested earlier that an 8-input, 16-bit multiplexer might be used in the design of a computer processor. This function could be performed by 16 74x151 8-input, 1-bit multiplexers or equivalent ASIC cells, each handling one bit of all the inputs and the output. The processor's 3-bit register-select field would be connected to the A, B, and C inputs of all 16 muxes, so they would all select the same register source at any given time.

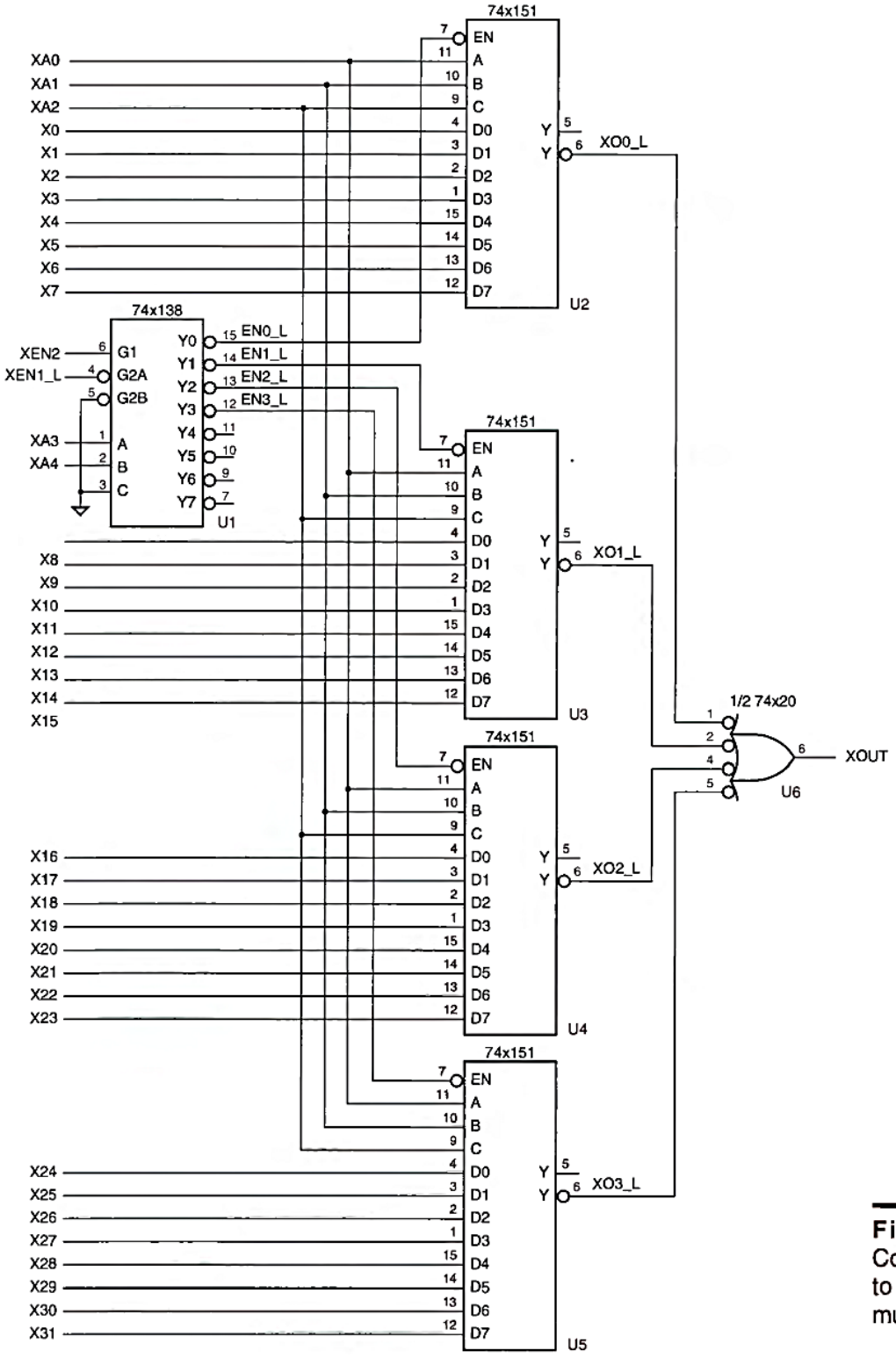


Figure 6-62
Combining 74x151s to make a 32-to-1 multiplexer.

Example 4.18 : *Implement the following Boolean function using 8 :1 MUX.*

$F(P, Q, R, S) = \Sigma m (0, 1, 3, 4, 8, 9, 15)$

Solution : Fig. 4.58 shows the implementation of given Boolean function with 8 : 1 multiplexer.

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\overline{A}$	(0)	(1)	2	(3)	(4)	5	6	7
A	(8)	(9)	10	11	12	13	14	(15)
	1	1	0	$\overline{A}$	$\overline{A}$	0	0	A

Fig. 4.58 (a) Implementation table

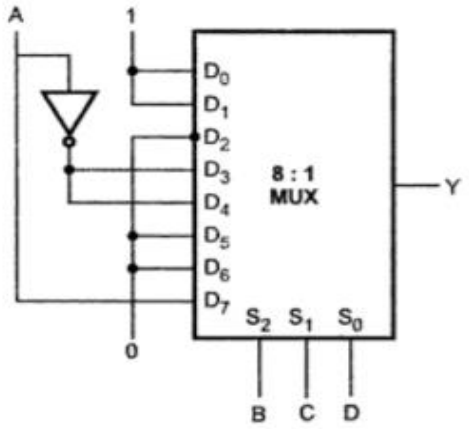


Fig. 4.58 (b) Multiplexer Implementation

Table 6-48 Dataflow VHDL program for a 4-input, 8-bit multiplexer.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mux4in8b is
    port (
        S: in STD_LOGIC_VECTOR (1 downto 0);       -- Select inputs, 0-3 ==> A-D
        A, B, C, D: in STD_LOGIC_VECTOR (1 to 8); -- Data bus input
        Y: out STD_LOGIC_VECTOR (1 to 8)           -- Data bus output
    );
end mux4in8b;

architecture mux4in8b of mux4in8b is
begin
    with S select Y <=
      A when "00",
      B when "01",
      C when "10",
      D when "11",
      (others => 'U') when others; -- this creates an 8-bit vector of 'U'
end mux4in8b;
```

Table 6-49 Behavioral architecture for a 4-input, 8-bit multiplexer.

```
architecture mux4in8p of mux4in8b is
begin
process(S, A, B, C, D)
  begin
    case S is
      when "00" => Y <= A;
      when "01" => Y <= B;
      when "10" => Y <= C;
      when "11" => Y <= D;
      when others => Y <= (others => 'U');  -- 8-bit vector of 'U'
    end case;
  end process;
end mux4in8p;
```

4.6 Demultiplexer

A demultiplexer is a circuit that receives information on a single line and transmits this information on one of 2^n possible output lines. The selection of specific output line is controlled by the values of n selection lines. Fig. 4.64 shows 1 : 4 demultiplexer. The single input variable D_{in} has a path to all four outputs, but the input information is directed to only one of the output lines.

Enable	S_1	S_0	D_{in}	Y_0	Y_1	Y_2	Y_3
0	X	X	X	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	0	0	0	0
1	0	1	1	0	1	0	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	0
1	1	1	0	0	0	0	0
1	1	1	1	0	0	0	1

Table 4.18 Function table for 1:4 demultiplexer

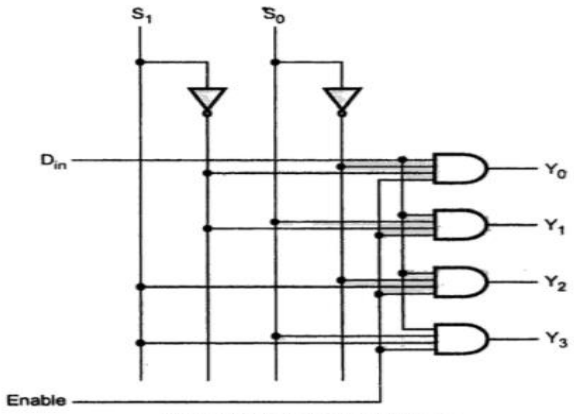


Fig. 4.64 (a) Logic diagram

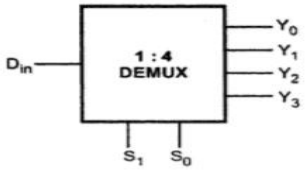


Fig. 4.64 (b) Logic symbol

Example 4.25 : *Design 1 : 8 demultiplexer using two 1 : 4 demultiplexers.*

Solution : The cascading of demultiplexers is similar to the cascading of decoder. Fig. 4.65 shows cascading of two 1 : 4 demultiplexers to form 1 : 8 demultiplexer.

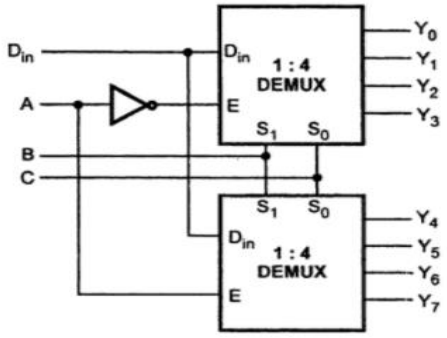


Fig. 4.65 Cascading of demultiplexers

Example 4.26 : *Implement full subtractor using demultiplexer.*

Solution : Let us see the truth table of full subtractor.

A	B	B_{in}	D	B_{out}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Table 4.19 Truth table of full subtractor

For full subtractor difference D function can be written as $D = f(A, B, C) = \Sigma m (1, 2, 4, 7)$ and B_{out} function can be written as

$$B_{out} = F(A, B, C) = \Sigma m (1, 2, 3, 7)$$

With D_{in} input 1, demultiplexer gives minterms at the output so by logically ORing required minterms we can implement Boolean functions for full subtractor. Fig. 4.66 shows the implementation of full subtractor using demultiplexer.

4.7 Code Converters

There is a wide variety of binary codes used in digital systems. Some of these codes are binary-coded-decimal (BCD), Excess-3, Gray, and so on. Many times it is required to convert one code to another.

This section explains the design of various code converters.

4.7.1 Binary to BCD Converter

Let us see the truth table for binary to BCD converter

Binary code				BCD code				
D	C	B	A	B_4	B_3	B_2	B_1	B_0
0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	1
0	0	1	0	0	0	0	1	0
0	0	1	1	0	0	0	1	1
0	1	0	0	0	0	1	0	0
0	1	0	1	0	0	1	0	1
0	1	1	0	0	0	1	1	0
0	1	1	1	0	0	1	1	1
1	0	0	0	0	1	0	0	0
1	0	0	1	0	1	0	0	1
1	0	1	0	1	0	0	0	0

1	0	1	1	1	0	0	0	1
1	1	0	0	1	0	0	1	0
1	1	0	1	1	0	0	1	1
1	1	1	0	1	0	1	0	0
1	1	1	1	1	0	1	0	1

Table 4.20

K-map simplification

For B_0

DC \ BA	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	0	1	1	0
10	0	1	1	0

$B_0 = A$

For B_1

DC \ BA	00	01	11	10
00	0	0	1	1
01	0	0	1	1
11	1	1	0	0
10	0	0	0	0

$B_1 = DC\overline{B} + \overline{D}B$

For B_2

DC \ BA	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	0	0	1	1
10	0	0	0	0

$B_2 = \overline{D}C + CB$

For B_3

DC \ BA	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	0	0
10	1	1	0	0

$B_3 = D\overline{C}\,\overline{B}$

For B_4

DC \ BA	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	1	1
10	0	0	1	1

$B_4 = DC + DB$

Logic diagram

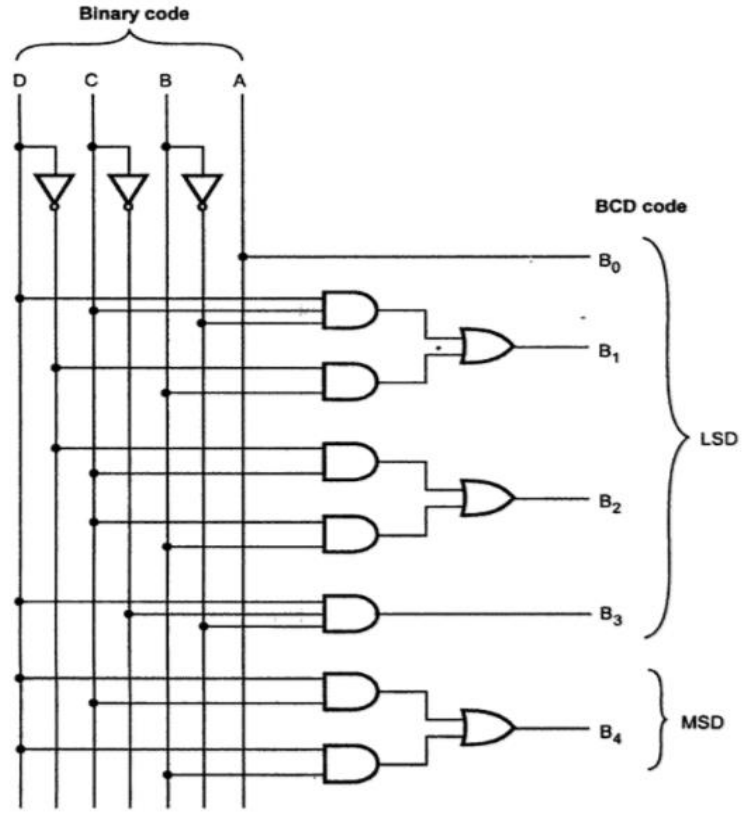


Fig. 4.68 Binary to BCD converter

4.7.2 BCD to Binary Converter

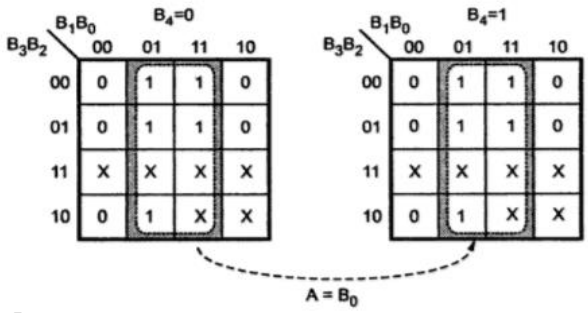
Let us see the truth table for BCD to binary converter.

B_4	B_3	B_2	B_1	B_0	E	D	C	B	A
0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	0	0	1	0
0	0	0	1	1	0	0	0	1	1
0	0	1	0	0	0	0	1	0	0
0	0	1	0	1	0	0	1	0	1
0	0	1	1	0	0	0	1	1	0
0	0	1	1	1	0	0	1	1	1
0	1	0	0	0	0	1	0	0	0
0	1	0	0	1	0	1	0	0	1
1	0	0	0	0	0	1	0	1	0
1	0	0	0	1	0	1	0	1	1
1	0	0	1	0	0	1	1	0	0
1	0	0	1	1	0	1	1	0	1
1	0	1	0	0	0	1	1	1	0
1	0	1	0	1	0	1	1	1	1
1	0	1	1	0	1	0	0	0	0
1	0	1	1	1	1	0	0	0	1
1	1	0	0	0	1	0	0	1	0
1	1	0	0	1	1	0	0	1	1

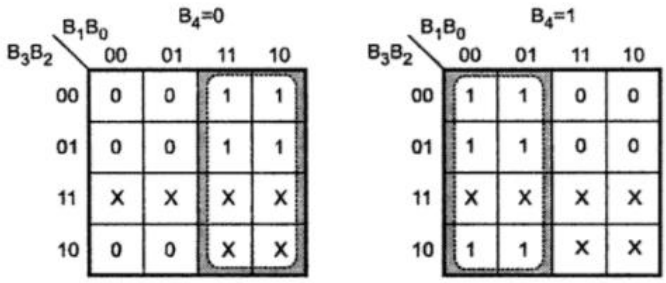
Table 4.21 Truth table for BCD to binary converter

K-map simplification

For A

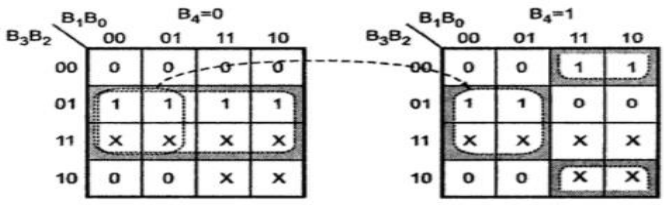


For B



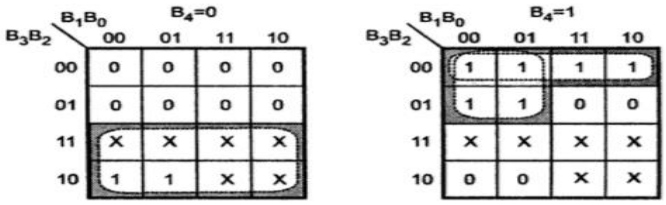
$$B = B_1\overline{B}_4 + \overline{B}_1B_4$$
$$= B_1 \oplus B_4$$

For C



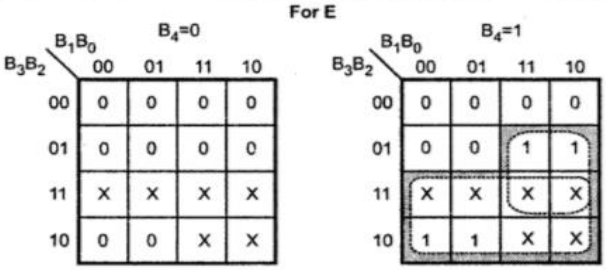
$$C = \overline{B}_4B_2 + B_2\overline{B}_1 + B_4\overline{B}_2B_1$$

For D



$$D = \overline{B}_4B_3 + B_4\overline{B}_3\overline{B}_2 + B_4\overline{B}_3B_1$$

For E



$$E = B_4B_3 + B_4B_2B_1$$

Fig. 4.69

Logic diagram

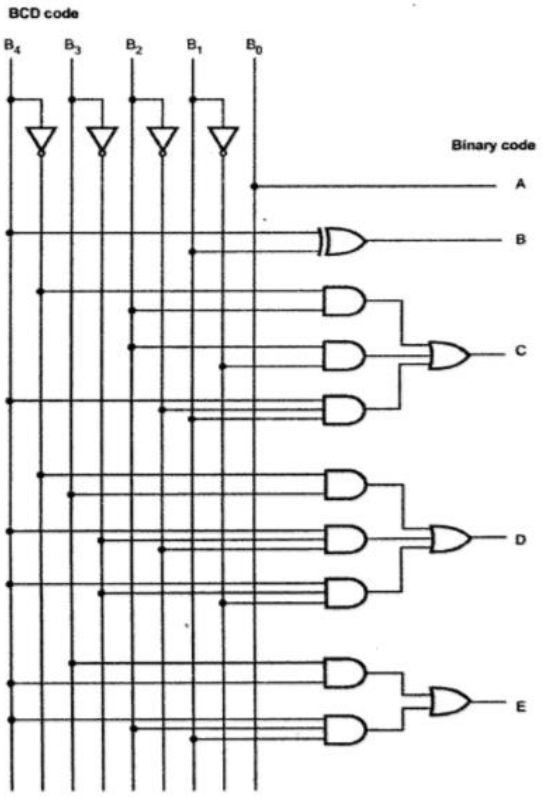


Fig. 4.70 BCD to binary code converter

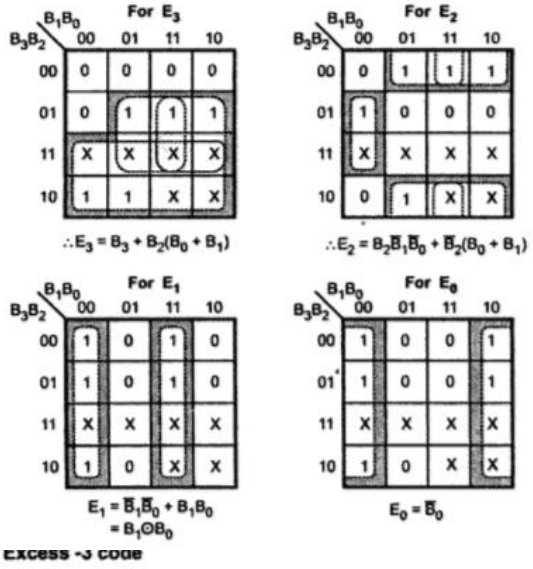
4.7.3 BCD to Excess 3

Excess-3 code is a modified form of a BCD number. The Excess-3 code can be derived from the natural BCD code by adding 3 to each coded number. For example, decimal 12 can be represented in BCD as 0001 0010. Now adding 3 to each digit we get Excess-3 code as 0100 0101 (12 in decimal). With this information the truth table for BCD to Excess-3 code converter can be determined as shown in Table 4.22.

Decimal	B_3	B_2	B_1	B_0	E_3	E_2	E_1	E_0
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

Table 4.22

K-map simplification



Logic diagram

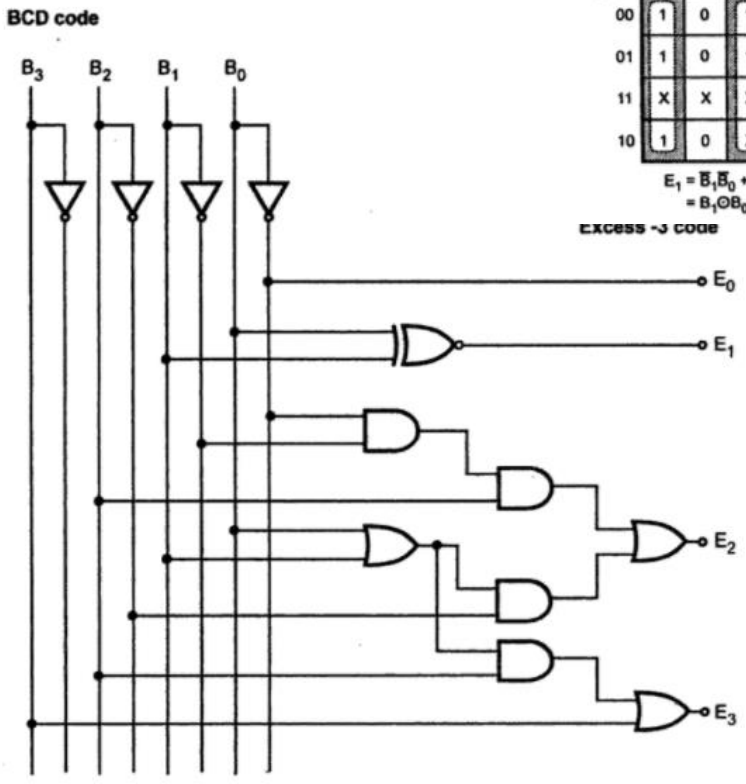


Fig. 4.72 BCD to Excess-3 code converter

4.7.4 Excess-3 to BCD Code Converter

The truth table for Excess-3 to BCD code converter is as shown in the Table 4.23.

E_3	E_2	E_1	E_0	B_3	B_2	B_1	B_0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	1
0	1	0	1	0	0	1	0
0	1	1	0	0	0	1	1
0	1	1	1	0	1	0	0
1	0	0	0	0	1	0	1
1	0	0	1	0	1	1	0
1	0	1	0	0	1	1	1
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	1

Table 4.23 Truth table for Excess-3 to BCD converter

K-map simplification

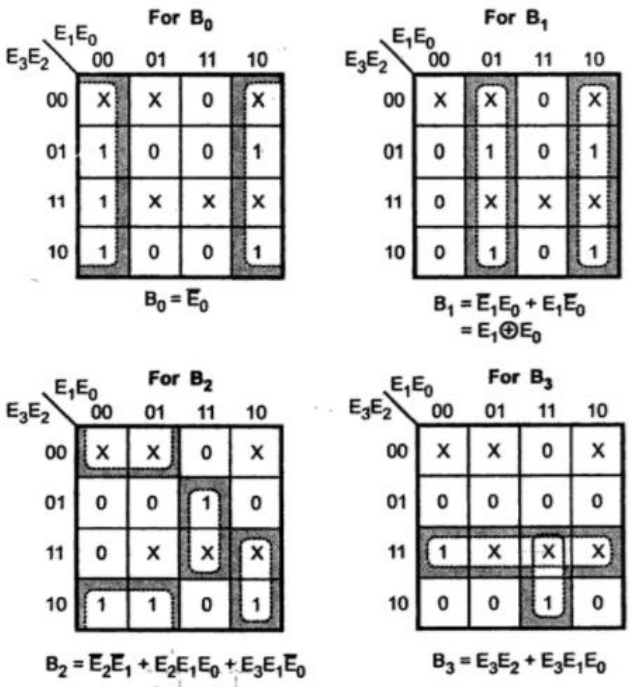


Fig. 4.73

Logic diagram

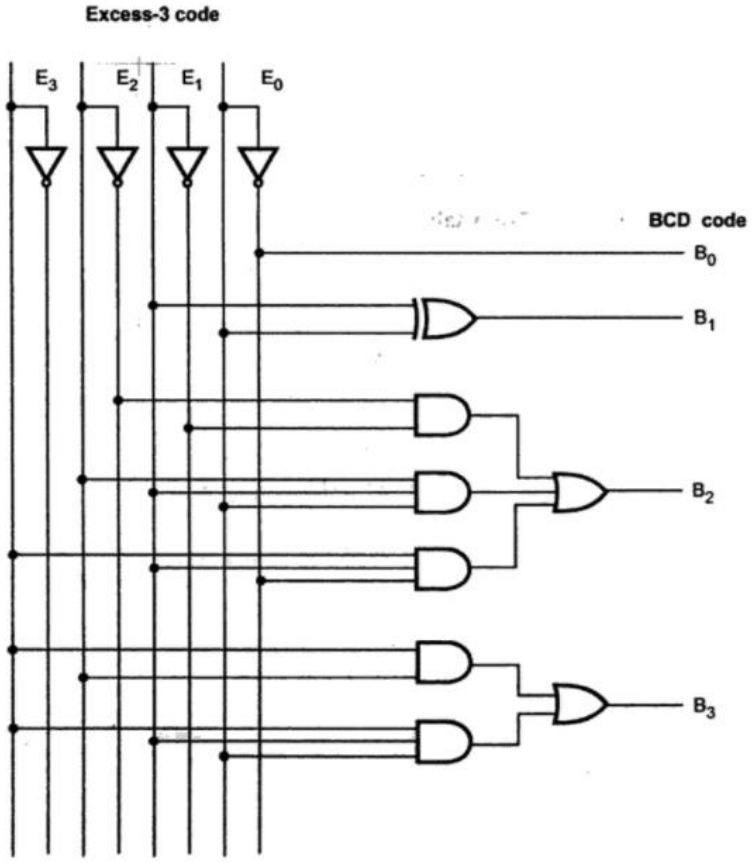


Fig. 4.74 Excess-3 to BCD code converter

4.7.5 Binary to Gray Code Converter

The Gray code is often used in digital systems because it has the advantage that only one bit in the numerical representation changes between successive numbers. Table 4.24 shows decimal and Binary codes and corresponding Gray code.

Decimal	Binary code				Gray code			
	D	C	B	A	G_3	G_2	G_1	G_0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	0	1	0
4	0	1	0	0	0	1	1	0
5	0	1	0	1	0	1	1	1
6	0	1	1	0	0	1	0	1
7	0	1	1	1	0	1	0	0
8	1	0	0	0	1	1	0	0
9	1	0	0	1	1	1	0	1
10	1	0	1	0	1	1	1	1
11	1	0	1	1	1	1	1	0
12	1	1	0	0	1	0	1	0
13	1	1	0	1	1	0	1	1
14	1	1	1	0	1	0	0	1
15	1	1	1	1	1	0	0	0

K-map simplification

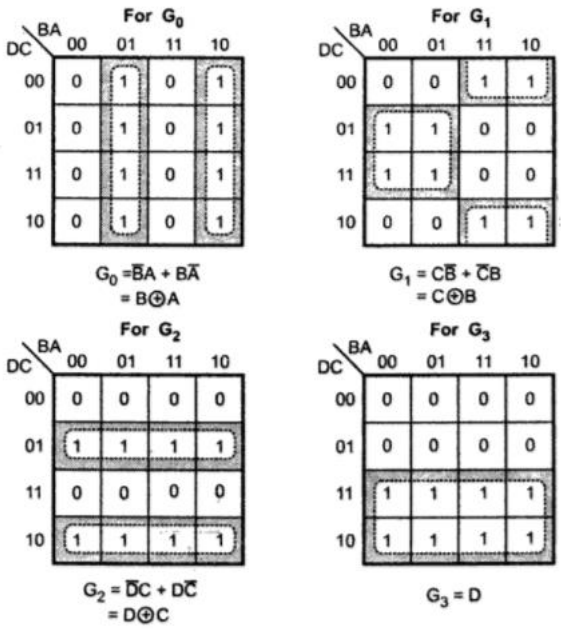


Fig. 4.75

Logic diagram

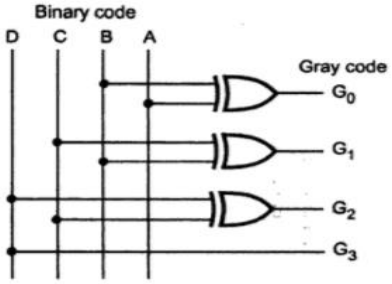


Fig. 4.76 Binary to gray code converter

4.7.6 Gray Code to Binary Code Converter

Table 4.25 shows the truth table for gray code to binary code converter.

Gray code				Binary code			
G_3	G_2	G_1	G_0	D	C	B	A
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	1	0	0	1	0
0	0	1	0	0	0	1	1
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	1
0	1	0	1	0	1	1	0
0	1	0	0	0	1	1	1
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	1
1	1	1	1	1	0	1	0
1	1	1	0	1	0	1	1
1	0	1	0	1	1	0	0
1	0	1	1	1	1	0	1
1	0	0	1	1	1	1	0
1	0	0	0	1	1	1	1

Table 4.25 Truth table for gray code to binary code converter

K-map simplification

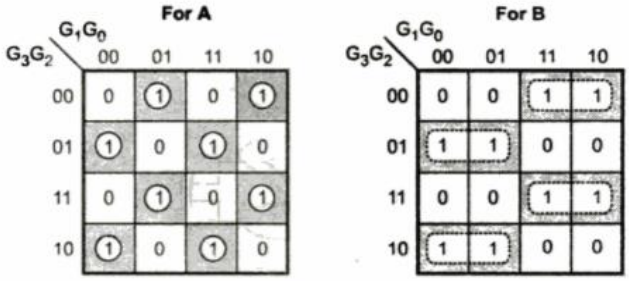


Fig. 4.77

$$A = (\overline{G}_3G_2 + G_3\overline{G}_2)\,\overline{G}_1\overline{G}_0 + (\overline{G}_3\,\overline{G}_2 + G_3G_2)\,\overline{G}_1G_0 + (\overline{G}_3G_2 + G_3\,\overline{G}_2)\,G_1G_0 + (\overline{G}_3\overline{G}_2 + G_3G_2)\,G_1\,\overline{G}_0$$

$$= (G_3 \oplus G_2)\,\overline{G}_1\overline{G}_0 + (G_3 \odot G_2)\overline{G}_1G_0 + (G_3 \oplus G_2)\,G_1G_0 + (G_3 \odot G_2)\,G_1\overline{G}_0$$

$$= (G_3 \oplus G_2)\,(\overline{G}_1\overline{G}_0 + G_1G_0) + (G_3 \odot G_2)\,(\overline{G}_1G_0 + G_1\overline{G}_0)$$

$$= (G_3 \oplus G_2)\,(G_1 \odot G_0) + (G_3 \odot G_2)\,(G_1 \oplus G_0)$$

$$= (G_3 \oplus G_2)\,(\overline{G_1 \oplus G_0}) + (\overline{G_3 \oplus G_2})\,(G_1 \oplus G_0)$$

$$= (G_3 \oplus G_2) \oplus (G_1 \oplus G_0)$$

$$B = (\overline{G}_3\overline{G}_2 + G_3G_2)G_1 + (\overline{G}_3G_2 + G_3\overline{G}_2)\overline{G}_1$$

$$= (G_3 \odot G_2)G_1 + (G_3 \oplus G_2)\overline{G}_1$$

$$= (\overline{G_3 \oplus G_2})\,G_1 + (G_3 \oplus G_2)\,\overline{G}_1$$

$$= G_3 \oplus G_2 \oplus G_1$$

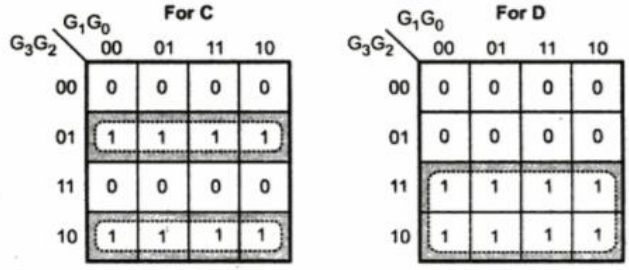


Fig. 4.78

$$C = \overline{G}_3G_2 + G_3\overline{G}_2 \qquad D = G_3$$

$$= G_3 \oplus G_2$$

Logic diagram

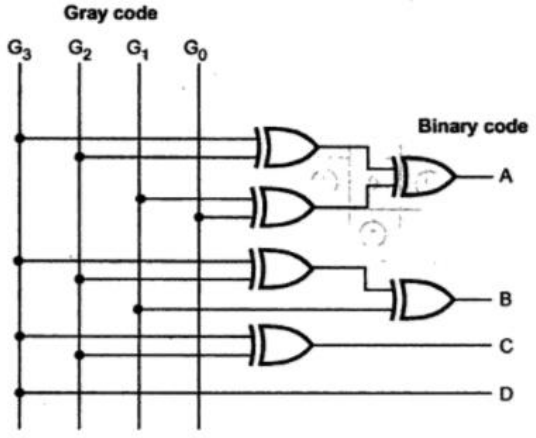


Fig. 4.79 Gray code to binary code converter

4.7.7 BCD to Gray Code Converter

Table 4.26 shows truth table for BCD to gray code converter.

BCD code				Gray code			
B_3	B_2	B_1	B_0	G_3	G_2	G_1	G_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1

Table 4.26 Truth table for BCD to gray code converter

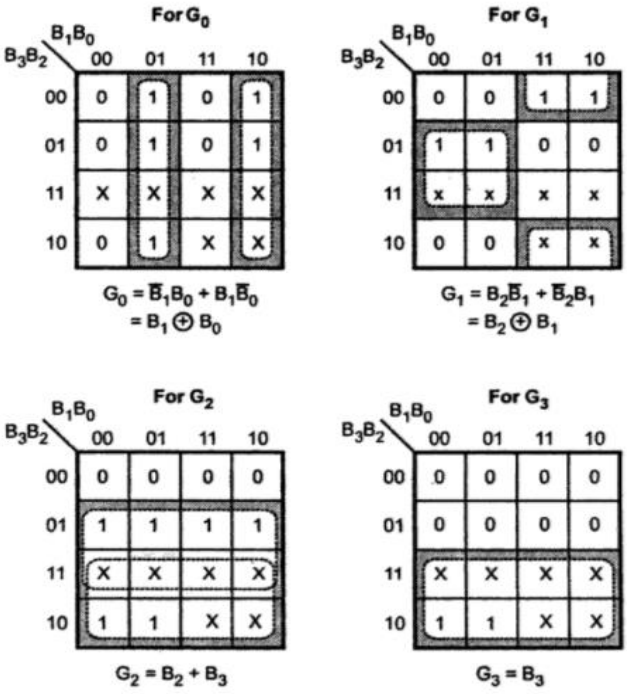


Fig. 4.80

Logic diagram

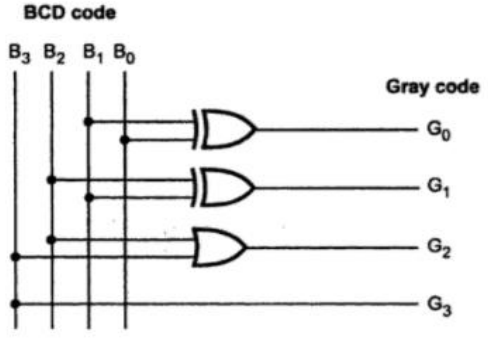
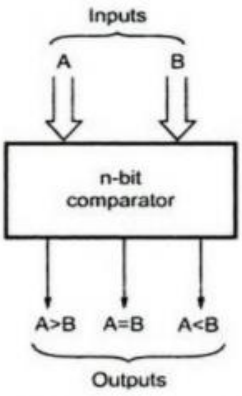


Fig. 4.81 BCD to gray code converter

COMPARATORS

A circuit that compares comparator two binary words and indicates whether they are equal is called a comparator. Some comparators interpret their input words as signed or unsigned numbers and also indicate an arithmetic relationship (greater or less than) between the words. These devices are often called magnitude comparators.

4.10 Comparators



A comparator is a special combinational circuit designed primarily to compare the relative magnitude of two binary numbers. Fig. 4.92 shows the block diagram of an n-bit comparator. It receives two n-bit numbers A and B as inputs and the outputs are A > B, A = B and A < B. Depending upon the relative magnitudes of the two number, one of the outputs will be high.

Fig. 4.92 Block diagram of n-bit comparator

Example 4.27 : *Design 2-bit comparator using gates.*

Solution : The truth table for 2-bit is given in Table 4.30.

Inputs				Outputs		
A_1	A_0	B_1	B_0	A > B	A = B	A< B
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	1	0	0
0	1	0	1	0	1	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	0

Table 4.30

lification

A > B

A_1A_0 \ B_1B_0	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	1	1	0	1
10	1	1	0	0

$$A>B = A_0\overline{B}_1\overline{B}_0 + A_1\overline{B}_1 + A_1A_0\overline{B}_0$$

A = B

A_1A_0 \ B_1B_0	00	01	11	10
00	1	0	0	0
01	0	1	0	0
11	0	0	1	0
10	0	0	0	1

A < B

A_1A_0 \ B_1B_0	00	01	11	10
00	0	1	1	1
01	0	0	1	1
11	0	0	0	0
10	0	0	1	0

Fig. 4.93

$$(A = B) = \overline{A}_1\overline{A}_0\overline{B}_1\overline{B}_0 + \overline{A}_1A_0\overline{B}_1B_0 + A_1A_0B_1B_0 + A_1\overline{A}_0B_1\overline{B}_0$$

$$= \overline{A}_1\overline{B}_1(\overline{A}_0\overline{B}_0 + A_0B_0) + A_1B_1(A_0B_0 + \overline{A}_0\overline{B}_0)$$

$$= (A_0 \odot B_0)(A_1 \odot B_1)$$

$$(A < B) = \overline{A}_1\overline{A}_0B_0 + \overline{A}_0B_1B_0 + \overline{A}_1B_1$$

agram

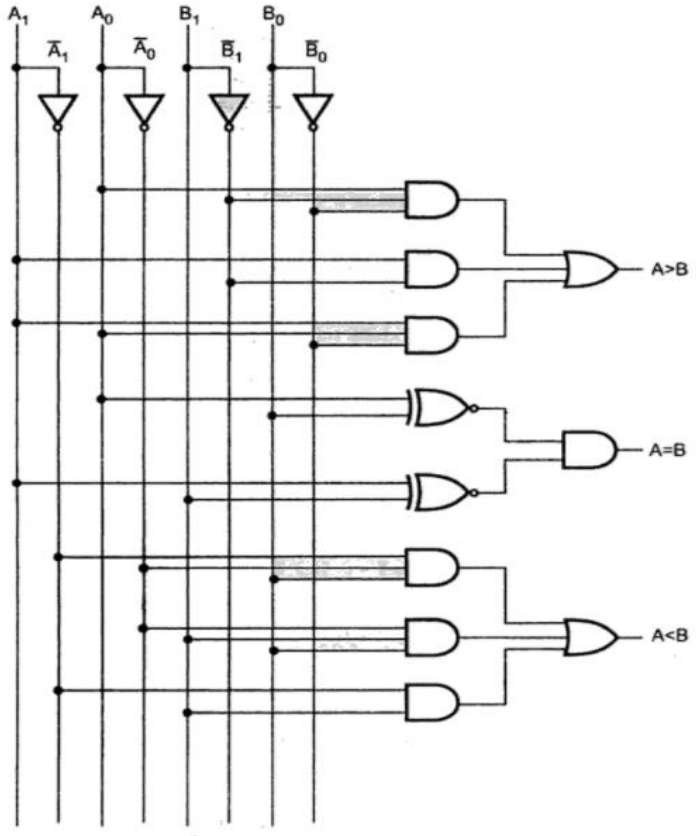
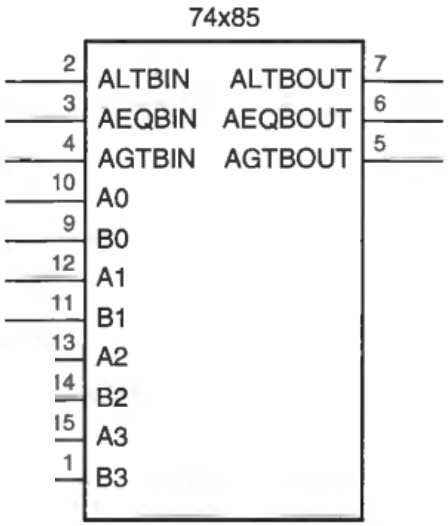


Fig. 4.94

STANDARD MSI MAGNITUDE COMPARATORS

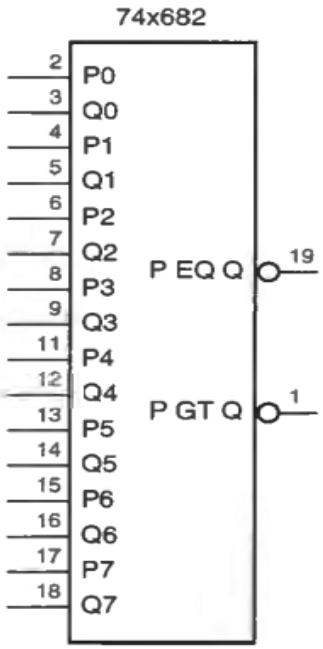
Comparator applications are common enough that magnitude comparators have been developed commercially as MSI parts. The 74x85 is a 4-bit comparator with the logic symbol shown in Figure 6-78. It provides a greater-than output (AGTBOUT) and a less-than output (AL TBOUT) as well as an equal output (AEQBOUT). The '85 also has cascading inputs (AGTBIN, ALTBIN, AEQBIN) for combining multiple '85s to create comparators for more than four bits. Both the cascading inputs and the outputs are arranged in a l-out-of-3 code, since in normal operation exactly one input and one output should be asserted.



Comparing Inputs	Cascading Inputs			Outputs		
A B	I(A > B)	I(A = B)	I(A < B)	A > B	A = B	A < B
A > B	X	X	X	1	0	0
A = B	1	0	0	1	0	0
	X	1	X	0	1	0
	0	0	1	0	0	1
	0	0	0	1	0	1
	1	0	1	0	0	0
A < B	X	X	X	0	0	1

Table 4.31 Function table for IC 7485

Several 8-bit MSI comparators are also available. The simplest of these is the 74x682, whose logic symbol is shown in Figure 6-80 and whose internal logic diagram is shown in Figure 6-82 on the next page. The top half of the circuit checks the two 8-bit input words for equality. Each XNOR-gate output is asserted if its inputs are equal, and the PEQQ_L output is asserted if all eight input-bit pairs are equal. The bottom half of the circuit compares the input words arithmetically and asserts PGTQ_L if P[7--0] >



ITERATIVE COMPARATOR CIRCUITS

1. Set EQ0 to 1 and set i to 0.
2. If EQi is 1 and Xi and Yi are equal, set EQi +1 to 1. Else set EQi+1 to 0.
3. Increment i.
4. If i < n, go to step 2.

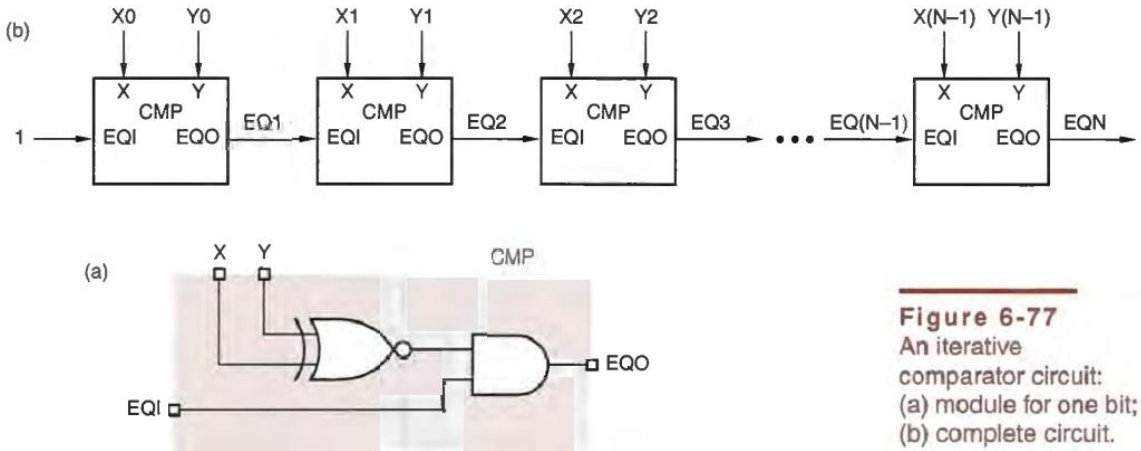


Figure 6-77
An iterative comparator circuit:
(a) module for one bit;
(b) complete circuit.

4.10.3 Comparators in VHDL

The following example illustrates the VHDL source code for comparing 8-bit unsigned intergers.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
ENTITY compare IS
     PORT ( A,B              :    IN STD_LOGIC_VECTOR(7 DOWNTO 0)
          AeqB, AgtB, AltB   :    OUT STD_LOGIC);
     END compare;
ARCHITECTURE Behavior OF compare IS
BEGIN
          AeqB <= '1'   WHEN A=B ELSE '0';
          AgtB <= '1'   WHEN A>B ELSE '0';
          AltB <= '1'   WHEN A<B ELSE '0';
END Behavior;
```

Example 4.28 : *Design an 8-bit comparator using two 7485 ICs .*

Solution : Fig. 4.96 shows an 8-bit comparator using two 7485 ICs.

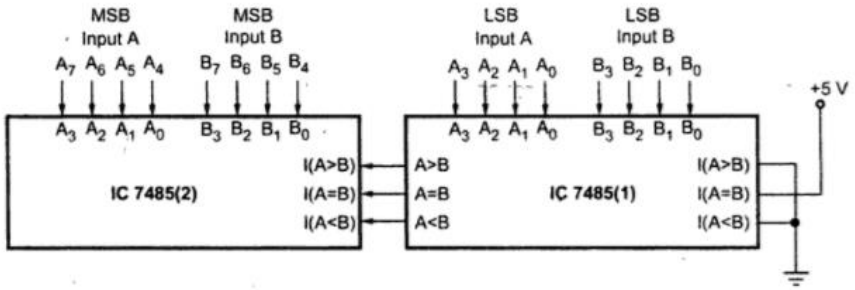


Fig. 4.96 8-bit comparator

4.11 Adders

Digital computers perform various arithmetic operations. The most basic operation, no doubt, is the addition of two binary digits. This simple addition consists of four possible elementary operations, namely,

$0 + 0 = 0$

$0 + 1 = 1$

$1 + 0 = 1$

$1 + 1 = 10_2$

The first three operations produce a sum whose length is one digit, but when the last operation is performed sum is two digits. The higher significant bit of this result is called a **carry**, and lower significant bit is called **sum**. The logic circuit which performs this operation is called a **half-adder**. The circuit which performs addition of three bits (two significant bits and a previous carry) is a **full-adder**. Let us see the logic circuits to perform half-adder and full-adder operations.

4.11.1 Half Adder

The half-adder operation needs two binary inputs : augend and addend bits; and two binary outputs : sum and carry. The truth table shown in Table 4.32 gives the relation between input and output variables for half-adder operation.

Inputs		Outputs	
A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Table 4.32 Truth table for half-adder

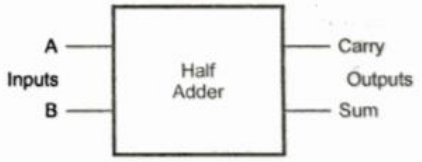


Fig. 4.99 Block schematic of half-adder

K-map simplification for carry and sum

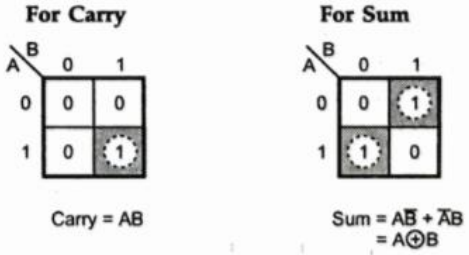


Fig. 4.100 Maps for half-adder

Logic Diagram

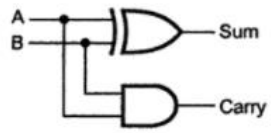


Fig. 4.101 Logic diagram for half-adder

Limitations of Half-Adder :

In multidigit addition we have to add two bits alongwith the carry of previous digit addition. Effectively such addition requires addition of three bits. This is not possible with half adder. Hence half-adders are not used in practice.

4.11.2 Full-Adder

A full-adder is a combinational circuit that forms the arithmetic sum of three input bits. It consists of three inputs and two outputs. Two of the input variables, denoted by A and B, represent the two significant bits to be added. The third input C_{in}, represents the carry from the previous lower significant position. The truth table for full-adder is shown in Table 4.33.

Inputs			Outputs	
A	B	C_{in}	Carry	Sum
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 4.33 Truth table for full-adder

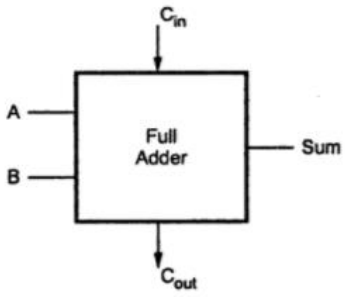
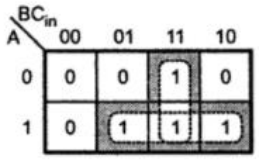


Fig. 4.102 Block schematic of full-adder

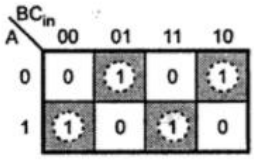
K-map simplification for carry and sum

For Carry (C_{out})



$C_{out} = AB + A C_{in} + B C_{in}$

For Sum



$Sum = \overline{A}\,\overline{B}C_{in} + \overline{A}B\overline{C}_{in} + A\overline{B}\,\overline{C}_{in} + ABC_{in}$

Fig. 4.103 Maps for full-adder

With this simplified Boolean function circuit for full-adder can be implemented as shown in the Fig. 4.105.

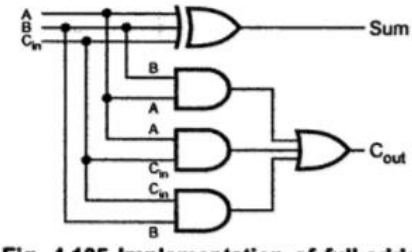


Fig. 4.105 Implementation of full-adder

A full-adder can also be implemented with two half-adders and one OR gate, as shown in the Fig. 4.106. The sum output from the second half-adder is the exclusive-OR of C_{in} and the output of the first half-adder, giving

$$Sum = C_{in} \oplus (A \oplus B)$$

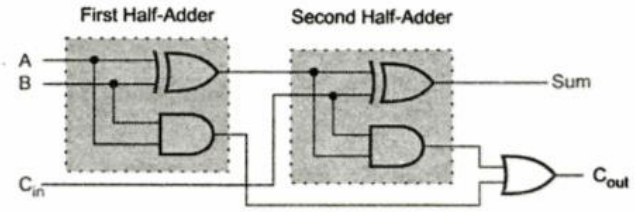


Fig. 4.106 Implementation of a full-adder with two half-adders and an OR gate

4.12 Subtractors

4.12.1 Half-Subtractor

A half-subtractor is a combinational circuit that does the subtraction between two bits and produces their difference. It also has an output to specify if a 1 has been borrowed. Let us designate minuend bit as A and the subtrahend bit as B. The result of operation A – B for all possible values of A and B is tabulated in Table 4.34.

Inputs		Outputs	
A	B	Difference	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Table 4.34 Truth table for half-subtractor

As shown in the Table 4.34, half-subtractor has two input variables and two output variables. The Boolean expression for the outputs of half-subtractor can be determined as follows.

K-map simplification for half-subtractor

For Difference

A \ B	0	1
0	0	1
1	1	0

Difference = $A\overline{B} + \overline{A}B$
= $A \oplus B$

For Borrow

A \ B	0	1
0	0	1
1	0	0

Borrow = $\overline{A}B$

Fig. 4.107

Logic Diagram

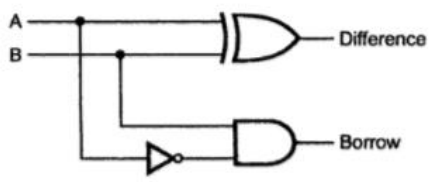


Fig. 4.108 Implem ntation of half-subtractor

Limitations of Half Subtractor :

In multidigit subtraction, we have to subtract bit alongwith the borrow of the previous digit subtraction. Effectively such subtraction requires subtraction between three bits. This is not possible with half-subtractor.

4.12.2 Full-Subtractor

A full-subtractor is a combinational circuit that performs a subtraction between two bits, taking into account borrow of the lower significant stage. This circuit has three inputs and two outputs. The three inputs are A, B and B_{in}, denote the minuend, subtrahend, and previous borrow, respectively. The two outputs, D and B_{out}, represent the difference and output borrow, respectively. The Table 4.35 shows the truth table for full-subtractor.

Inputs			Outputs	
A	B	B_{in}	D	B_{out}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Table 4.35 Truth table for full-subtractor

K-map simplification of D and B_{out}

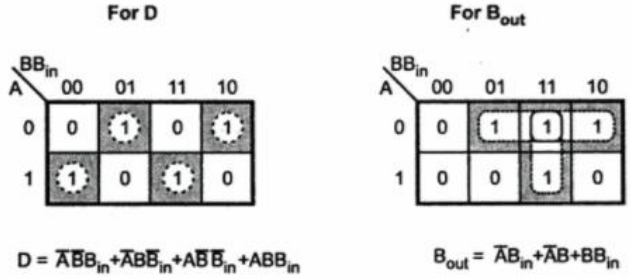


Fig. 4.109 Maps for full-subtractor

$$= B_{in} \oplus (A \oplus B)$$

With this simplified Boolean function circuit for full-subtractor can be implemented as shown in the Fig. 4.111.

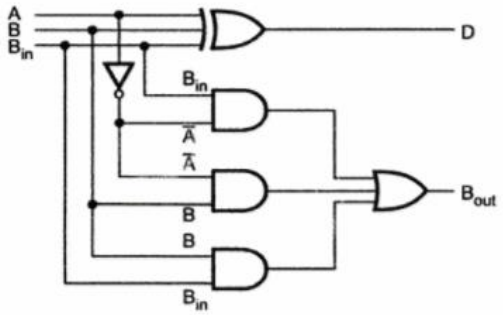


Fig. 4.111 Implementation of full-subtractor

4.13 n-Bit Parallel Adder or Ripple Adder

We have seen, a single full-adder is capable of adding two one-bit numbers and an input carry. In order to add binary numbers with more than one bit, additional full-adders must be employed. A n-bit, parallel adder can be constructed using number of full adder circuits connected in parallel. Fig. 4.113 shows the block diagram of n-bit parallel adder using number of full-adder circuits connected in cascade, i.e. the carry output of each adder is connected to the carry input of the next higher-order adder.

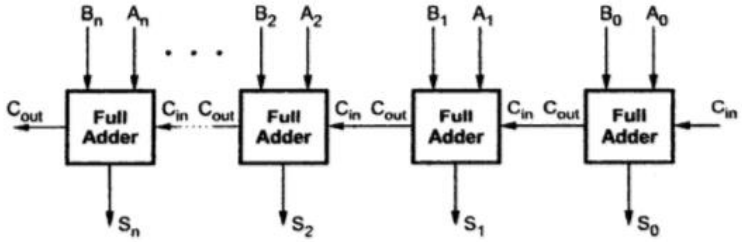


Fig. 4.113 Block diagram of n-bit parallel adder

It should be noted that either a half-adder can be used for the least significant position or the carry input of a full-adder is made 0 because there is no carry into the least significant bit position.

Example 4.29 : *Design a 4-bit parallel adder using full adders .*

Solution : Fig. 4.114 shows the block diagram and Fig. 4.115 shows logic symbol for 4-bit parallel adder. Here, for least significant position, carry input of full-adder is made 0.

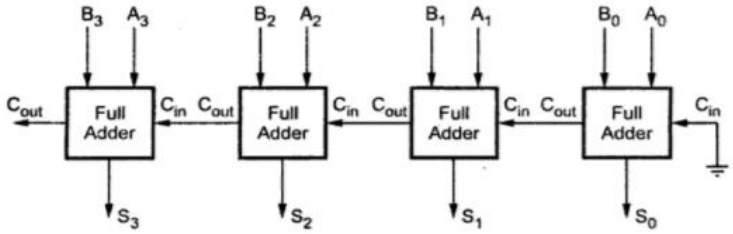


Fig. 4.114 Block diagram of 4-bit full adder

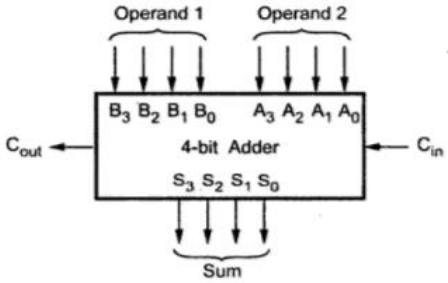


Fig. 4.115 Logic diagram of 4-bit parallel adder

4.14 Look Ahead Carry Generator (IC 74182)

The parallel adder discussed in the last paragraph is ripple carry type in which the carry output of each full-adder stage is connected to the carry input of the next higher-order stage. Therefore, the sum and carry outputs of any stage cannot be produced until the input carry occurs; this leads to a time delay in the addition process. This delay is known as *carry propagation delay*, which can be best explained by considering the following addition.

$$\begin{array}{cccccc} & 0 & 1 & 0 & 1 \\ + & 0 & 0 & 1 & 1 \\ \hline & 1 & 0 & 0 & 0 \end{array}$$

Addition of the LSB position produces a carry into the second position. This carry, when added to the bits of the second position (stage), produces a carry into the third position. The latter carry, when added to the bits of the third position, produces a carry

into the last position. The key thing to notice in this example is that the sum bit generated in the last position (MSB) depends on the carry that was generated by the addition in the previous positions. This means that, adder will not produce correct result until LSB carry has propagated through the intermediate full-adders. This represents a time delay that depends on the propagation delay produced in an each full-adder. For example, if each full-adder is considered to have a propagation delay of 30 ns, then S_3 will not reach its correct value until 90 ns after LSB carry is generated. Therefore, total time required to perform addition is 90+30 = 120 ns.

Obviously, this situation becomes much worse if we extend the adder circuit to add a greater number of bits. If the adder were handling 16-bit numbers, the carry propagation delay could be 480 ns.

One method of speeding up this process by eliminating inter stage carry delay is called **look ahead-carry addition.** This method utilizes logic gates to look at the lower-order bits of the augend and addend to see if a higher-order carry is to be generated. It uses two functions : carry generate and carry propagate.

Consider the circuit of the full adder shown in Fig. 4.116. Here, we define two functions : carry generate and carry propagate.

$$P_i = A_i \oplus B_i$$

$$G_i = A_i B_i$$

The output sum and carry can be expressed as

$$S_i = P_i \oplus C_i$$

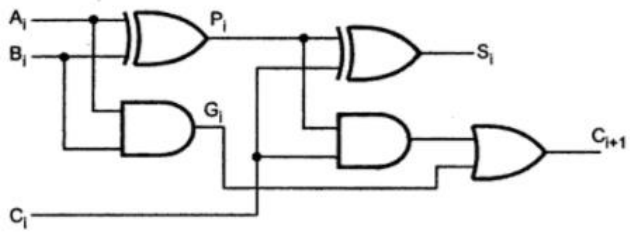


Fig. 4.116 Full adder circuit

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

G_i is called a carry generate and it produces on carry when both A_i and B_i are one, regardless of the input carry. P_i is called a carry propagate because it is term associated with the propagation of the carry from C_i to C_{i+1}.

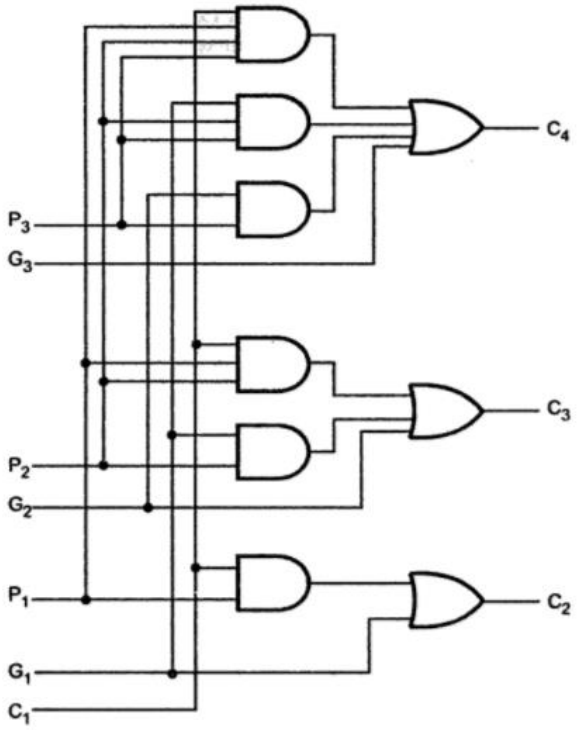


Fig. 4.117 Logic diagram of a look-ahead carry generator

Now the Boolean function for the carry output of each stage can be written as follows.

$$C_2 = G_1 + P_1C_1$$

$$C_3 = G_2 + P_2C_2 = G_2 + P_2 (G_1 + P_1C_1)$$

$$= G_2 + P_2G_1 + P_2 P_1C_1$$

$$C_4 = G_3 + P_3 C_3 = G_3 + P_3 (G_2 + P_2 G_1 + P_2 P_1C_1)$$

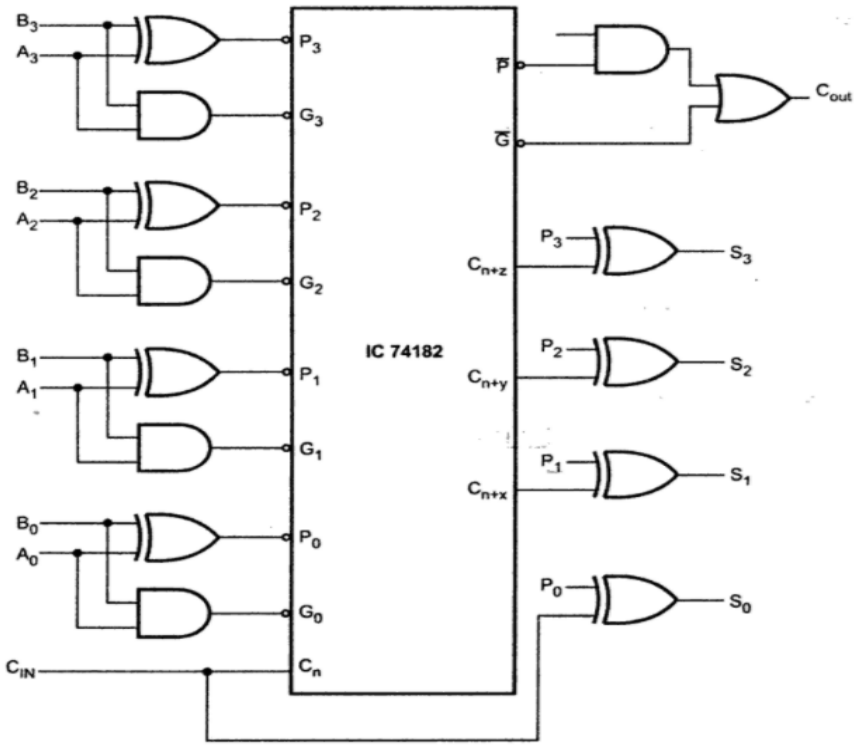
$$= G_3 + P_3G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 C_1$$

From the above Boolean function it can be seen that C_4 does not have to wait for C_3 and C_2 to propagate; in fact C_4 is propagated at the same time as C_2 and C_3.

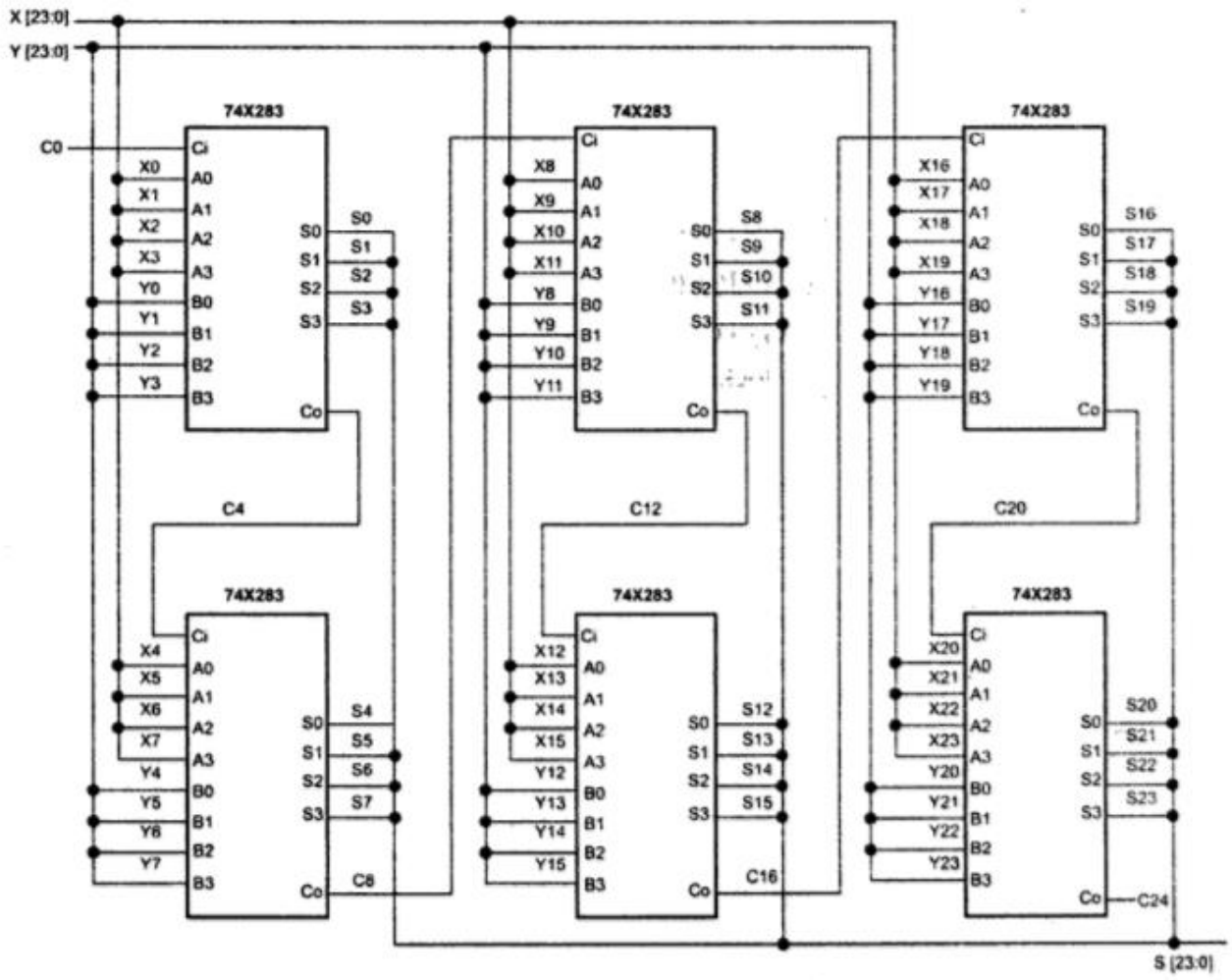
The Boolean functions for each output carry are expressed in sum-of product form, thus they can be implemented using AND-OR logic or NAND-NAND logic. Fig. 4.117 shows implementation of Boolean functions for C_2, C_3 and C_4 using AND-OR logic.

Example 4.30 : *Show the construction of 4-bit parallel adder using IC 74182.*

Solution : Fig. 4.120 shows the construction of 4-bit parallel adder using IC 74182. The last carry output can be generated using $\overline{P}$ and $\overline{G}$ outputs of IC 74182, as shown in the Fig. 4.120.



24 BIT GROUP RIPPLE ADDER USING 74X283



4.18 Adders and Subtractors in VHDL

The following examples illustrates the VHDL source code for various types of adders.

Example 4.34 : *Write a VHDL source code for full adder.*

Solution :

```
LIBRARY IEEE ;
USE IEEE.std_logic_1164.all ;
ENTITY fulladd IS
     PORT (   Cin, A, B : IN STD_LOGIC ;
              Sum, Cout : OUT STD_LOGIC) ;
END fulladd ;
ARCHITECTURE LogicFunc OF fulladd IS
BEGIN
     sum < = A XOR B XOR Cin ;
     Cout < = (A AND B) OR (Cin AND A) OR (Cin AND B) ;
END LogicFunc ;
```

Example 4.35 : *Write a VHDL source code for 4-bit adder.*

Solution : The VHDL source code for 4-bit adder is as follows.

```
LIBRARY IEEE ;
USE IEEE.std_logic_1164.all;
ENTITY adder4 IS
     PORT     (CIN              : IN STD_LOGIC;
               A3, A2, A1, A0   : IN STD_LOGIC;
               B3, B2, B1, B0   : IN STD_LOGIC;
               S3, S2, S1, S0   : OUT STD_LOGIC;
               Cout             : OUT STD_LOGIC);
END adder4;
ARCHITECTURE Structure OF adder4 IS
     SIGNAL c1, c2, c3 : STD_LOGIC;
     COMPONENT fulladd
          PORT (   CIN, A, B        : IN STD_LOGIC;
                   S, cOUT          : OUT STD_LOGIC);
     END COMPONENT;
BEGIN
stage0 : fulladd PORT MAP (CIN, A0, B0, S0, C1);
stage1 : fulladd PORT MAP (C1, A1, B1, S1, C2);
stage2 : fulladd PORT MAP (C2, A2, B2, S2, C3);
stage3 : fulladd PORT MAP
         CIN=>C3, A=>A3, B=>B3, S=>S3, COUT=> COUT);
END Structure;
```

Example 4.36 : *Write a VHDL source code for a 16-bit adder.*

Solution : Let A and B are the 16-bit numbers and the addition of CIN, A and B generates 16-bit sum, S and COUT and overflow signals.

```
LIBRARY IEEE ;
USE IEEE.Std_logic_1164.all ;
USE IEEE.Std_logic_signed.all ;
ENTITY adder16 IS
     PORT (   CIN           : IN STD_LOGIC ;
              A, B          : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
              S             : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
         COUT, OVERFLOW     : OUT STD_LOGIC) ;
END adder16 ;
ARCHITECTURE Behavior OF adder16 IS
     SIGNAL SUM: STD_LOGIC_VECTOR (16 DOWNTO 0) ;
BEGIN
     SUM < =('0' & A) + B + CIN ;
     S < = SUM (15 DOWNTO 0) ;
     COUT < = SUM (16) ;
     OVERFLOW < = SUM (16) XOR A(15) XOR B(15) XOR SUM (15) ;
END Behavior ;
```

Note : To get 17-bit sum, it is necessary to have at least one of A or B to be a 17-bit number. In ('0' and A), the 0 is concatenated to the 16-bit signal A to create a 17-bit signal. The and is a concatenated operator is VHDL.

Example 4.37 : *Write a VHDL source code for 4-bit subtractor.*

Solution : Refer Fig. 4.124 for 4-bit parallel subtractor. The VHDL source code for this circuit is as follows.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
ENTITY sub4 IS
     PORT (   CIN                    :    IN STD_LOGIC;
                  A3,A2, A1, A0      :    IN STD_LOGIC;
                  B3, B2, B1, B0     :    IN STD_LOGIC;
                  S3, S2, S1, S0     :    OUT STD_LOGIC;
                  COUT               :    OUT STD_LOGIC);
END sub4;
ARCHITECTURE structure OF sub4 IS;
     SIGNAL C1, C2, C3 :    STD_LOGIC;
     B0BAR <= NOT B0;
     B1BAR <= NOT B1;
     B2BAR <= NOT B2;
     B3BAR <= NOT B3;
     SIGNAL B0BAR, B1BAR, B2BAR, B3BAR : STD_LOGIC;
     COMPONENT fulladd
          PORT (    CIN, A, B     :    IN STD_LOGIC;
                         S, COUT  :    OUT STD_LOGIC);
     END COMPONENT;
     BEGIN
          stage0 : fulladd PORT MAP (CIN, A0, B0BAR, S0, C1);
          stage1 : fulladd PORT MAP (C1, A1, B1BAR, S1, C2);
          stage2 : fulladd PORT MAP (C2, A2, B2BAR, S2, C3);
          stage3 : fulladd PORT MAP(CIN=>C3,A=>A3,
                                    B=>B3,S=>s3,COUT=>COUT);
END Structure;
```

4.19 Arithmetic Logic Unit

The arithmetic and logic unit performs all the necessary arithmetic and logical operations. It requires one or two operands upon which it operates and produces a result. It is basically a multifunction combinational logic circuit. It provides select input to select the particular operation. In this section we study the very popular ALU IC, IC 74LS181.

IC74LS181

It is a 4-bit Arithmetic Logic Unit (ALU). Its features are as given below :

Features

- Provides 16 arithmetic operations : add, subtract, compare, double, plus twelve other arithmetic operations.
- Provides all 16 logic operations of two variables : exclusive-OR, compare, AND, NAND, OR, NOR, plus ten other logic operations.
- Full look ahead for high speed arithmetic operation on long words.

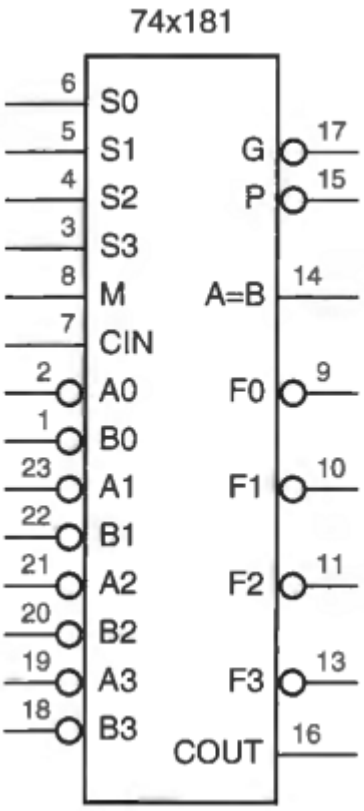


Table 6-70 Functions performed by the 74x181 4-bit ALU.

Inputs				Function	
S3	S2	S1	S0	*M = 0 (arithmetic)*	*M = 1 (logic)*
0	0	0	0	F = A minus 1 plus CIN	F = A′
0	0	0	1	F = A · B minus 1 plus CIN	F = A′ + B′
0	0	1	0	F = A · B′ minus 1 plus CIN	F = A′ + B
0	0	1	1	F = 1111 plus CIN	F = 1111
0	1	0	0	F = A plus (A + B′) plus CIN	F = A′ · B′
0	1	0	1	F = A · B plus (A + B′) plus CIN	F = B′
0	1	1	0	F = A minus B minus 1 plus CIN	F = A ⊕ B′
0	1	1	1	F = A + B′ plus CIN	F = A + B′
1	0	0	0	F = A plus (A + B) plus CIN	F = A′ · B
1	0	0	1	F = A plus B plus CIN	F = A ⊕ B
1	0	1	0	F = A · B′ plus (A + B) plus CIN	F = B
1	0	1	1	F = A + B plus CIN	F = A + B
1	1	0	0	F = A plus A plus CIN	F = 0000
1	1	0	1	F = A · B plus A plus CIN	F = A · B′
1	1	1	0	F = A · B′ plus A plus CIN	F = A · B
1	1	1	1	F = A plus CIN	F = A

The 181 's M input selects between arithmetic and logical operations. When M = 1, logical operations are selected, and each output Fi is a function only of the corresponding data inputs, Ai and Bi. The S3-SO inputs select a particular logical operation; any of the 16 different combinational logic functions on two variables may be selected.
When M = 0, arithmetic operations are selected, carries propagate between the stages, and CIN is used as a carry input to the least significant stage.

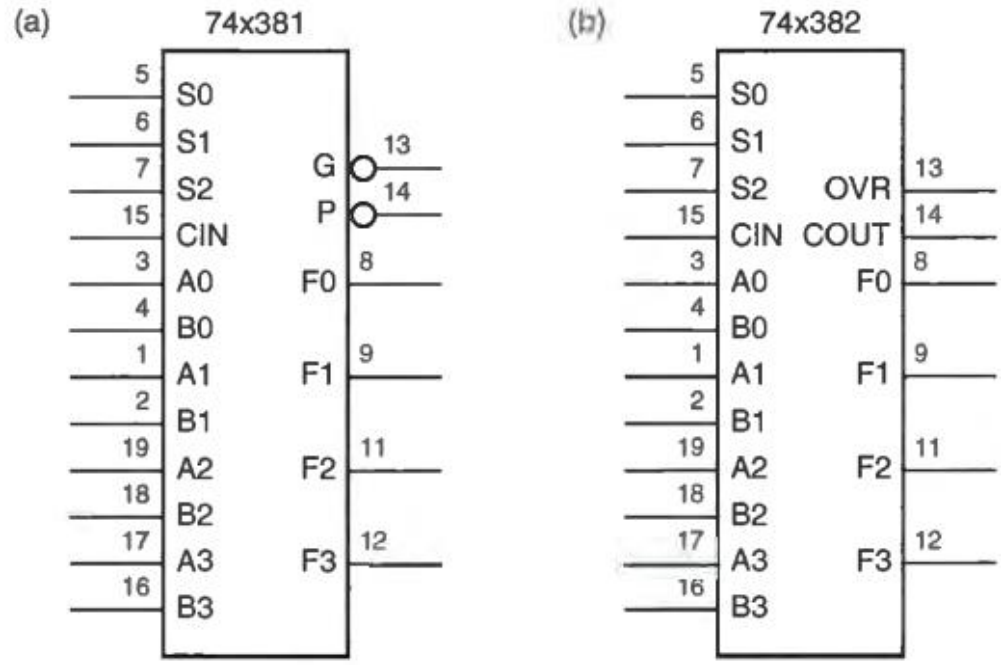


Figure 6-91
Logic symbols for 4-bit ALUs: (a) 74x381; (b) 74x382.

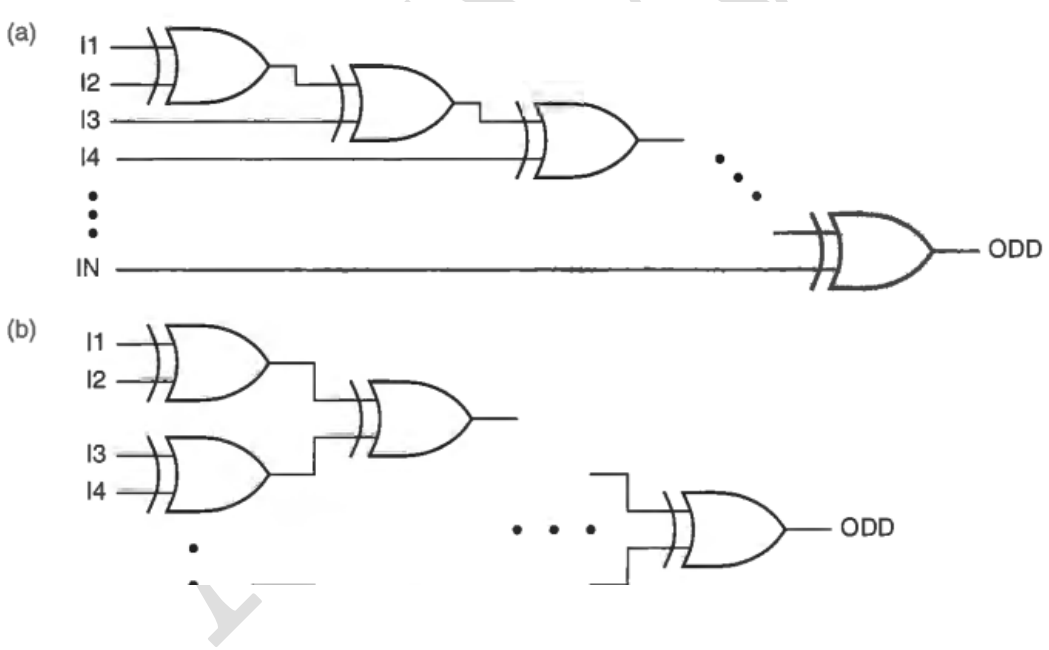
Two other MSI ALUs, the 74x381 and 74x382 shown in Figure 6-91, encode their select inputs more compactly, and provide only eight different but useful functions, as detailed in Table 6-71. The only difference between the '381 and '382 is that one provides group-carry-lookahead outputs (which we explain next), while the other provides ripple carry and overflow outputs.

Table 6-71 Functions performed by the 74x381 and 74x382 4-bit ALUs.

Inputs			Function
S2	S1	S0	
0	0	0	F = 0000
0	0	1	F = B minus A minus 1 plus CIN
0	1	0	F = A minus B minus 1 plus CIN
0	1	1	F = A plus B plus CIN
1	0	0	F = A ⊕ B
1	0	1	F = A + B
1	1	0	F = A · B
1	1	1	F = 1111

PARITY CIRCUITS:

This is called an odd-parity circuit, because its output is 1 if an odd number of its inputs are 1. The circuit in (b) is also an oddparity circuit, but it's faster because its gates are arranged in a treelike structure. If the output of either circuit is inverted, we get an even-parity circuit, whose output is 1 if an even number of its inputs are 1.



74X280 9-BIT PARITY GENERATOR

To detect errors in the transmission and storage of data. In an evenparity code, the parity bit is chosen so that the total number of 1 bits in a code word is even. Parity circuits like the 74x280 are used both to generate the correct value of the parity bit when a code word is stored or transmitted and to check the parity bit when a code word is retrieved or received.

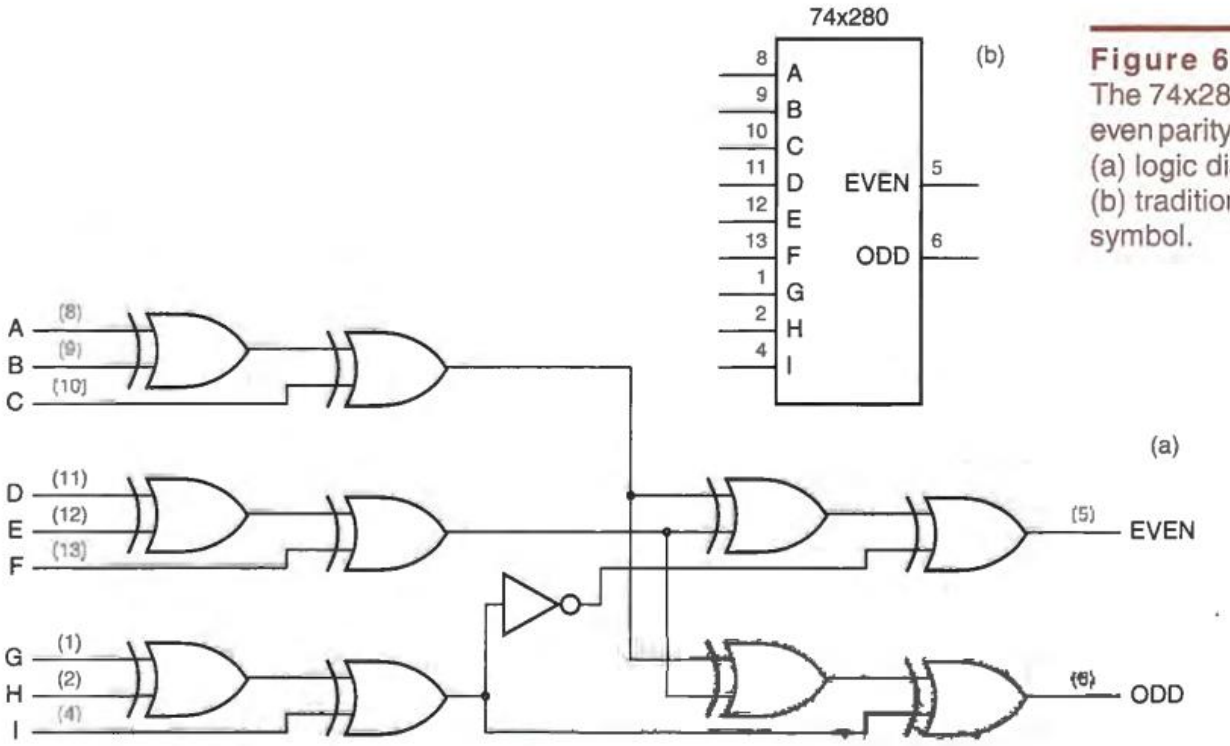


Figure 6-71
The 74x280 9-bit odd/even parity generator: (a) logic diagram; (b) traditional logic symbol.

APPLICATION: ERROR DETECTOR

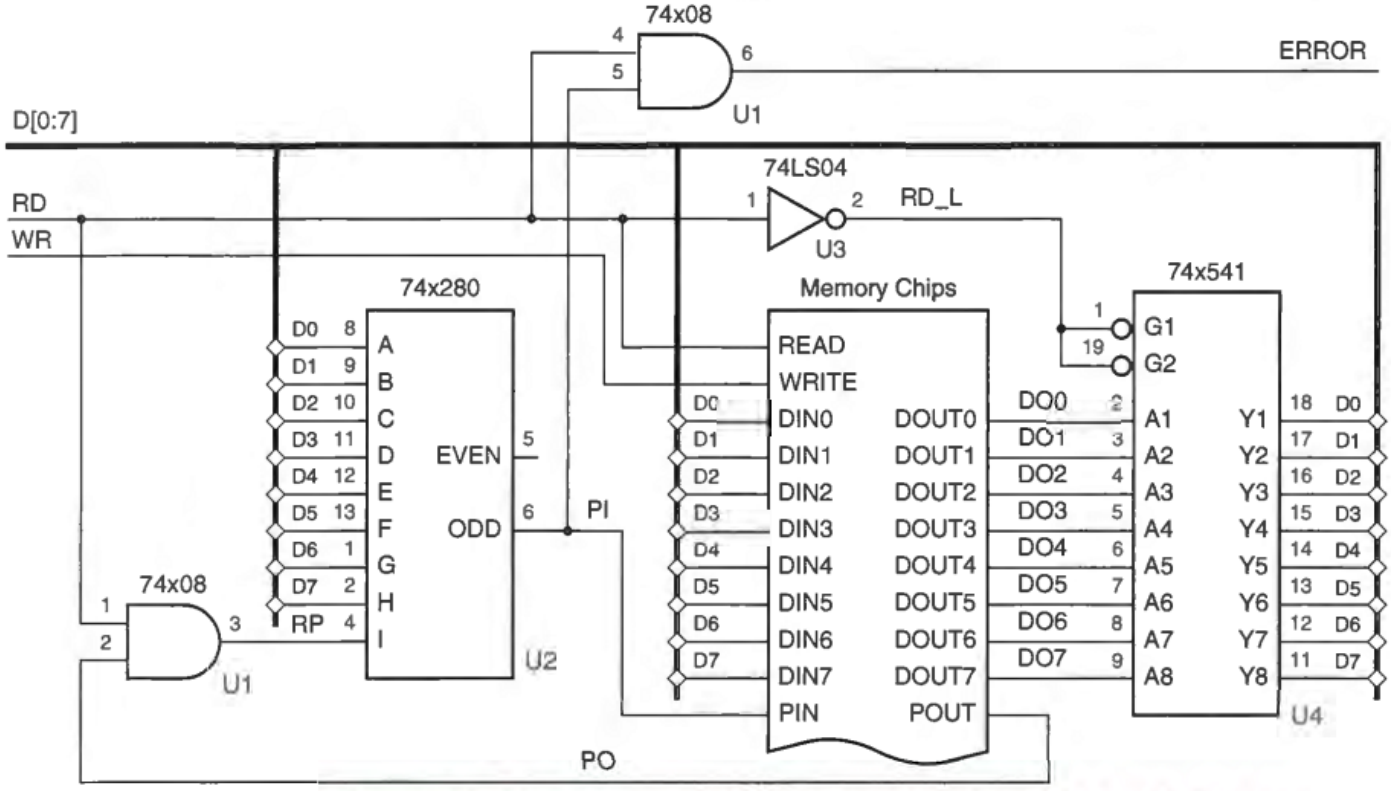


Figure 6-72 Parity generation and checking for an 8-bit-wide memory.

Table 6-56 Behavioral VHDL program for a 9-input parity checker.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity parity9 is
    port (
        I: in STD_LOGIC_VECTOR (1 to 9);
        EVEN, ODD: out STD_LOGIC
    );
end parity9;

architecture parity9p of parity9 is
begin
process (I)
  variable p : STD_LOGIC;
  begin
    p := I(1);
    for j in 2 to 9 loop
      if I(j) = '1' then p := not p; end if;
    end loop;
    ODD <= p;
    EVEN <= not p;
  end process;
end parity9p;
```

XOR GATE

Exclusive-OR and Exclusive-NOR Gates An Exclusive-OR (XOR) gate is a 2-input gate whose output is I if exactly one of its inputs is 1. Stated another way, an XOR gate produces a I output if its inputs are different. An Exclusive NOR (XNOR) or Equivalence gate is just the opposite-it produces a 1 output if its inputs are the same. A truth table for these functions is shown in Table 6-54.

$$X \oplus Y = X' \cdot Y + X \cdot Y'$$

X	Y	X ⊕ Y (XOR)	(X ⊕ Y)′ (XNOR)
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

Table 6-54 Truth table for XOR and XNOR functions.

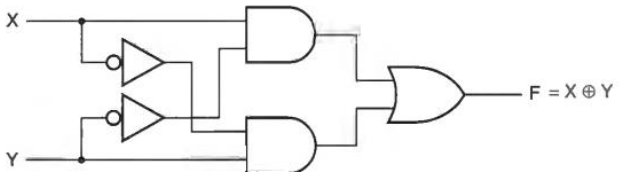


Figure 6-68 Multigate designs for the 2-input XOR function: (a) AND-OR; (b) three-level NAND.

GROUP-CARRY LOOKAHEAD

The' 181 and '381 provide group-carry-lookahead outputs that allow multiple ALUs to be cascaded without rippling carries between 4-bit groups. Like the 74x283, the ALUs use carry lookahead to produce carries internally.

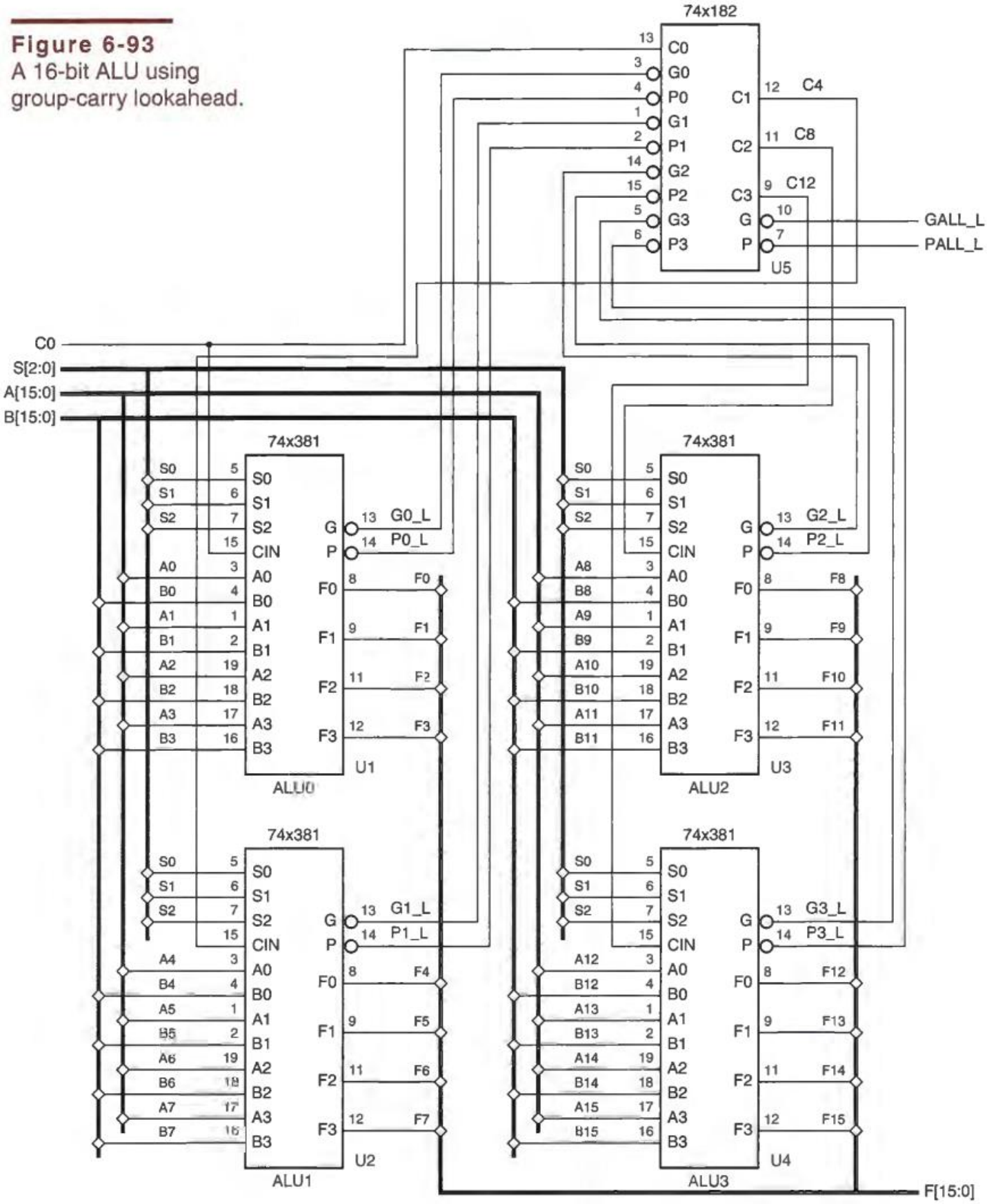


Figure 6-93
A 16-bit ALU using group-carry lookahead.

4.19.2 ALU in VHDL

Example 4.40 : *Write a VHDL source code for an ALU chip, 74381.*

Solution : The VHDL code for an ALU 74381 is given below.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
ENTITY ALU IS
     PORT ( S : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
       A,B    : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
       F      : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
       (IN    : OUT STD_LOGIC);
END ALU;
ARCHITECTURE Behavior OF ALU IS
BEGIN
  PROCESS (S, A, B)
    BEGIN
        CASE S IS
          WHEN "000"=> F<="0000";
          WHEN "001"=> F<=B-A-1+CIN;
          WHEN "010"=> F<=A-B-1+CIN;
          WHEN "011"=> F<=A+B+CIN;
          WHEN "100"=> F<=A XOR B;
          WHEN "101"=> F<=A OR B;
          WHEN "110"=> F<=A AND B;
          WHEN OTHERS => F<=1111;
          END CASE;
        END PROCESS;
      END Behavior;
```

Example 4.50 : *Design full adder using 74138 IC.*

Solution : Truth table for full adder is as shown in the Table 4.41.

Inputs			Outputs	
A	B	C_{in}	Carry	Sum
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 4.41 Truth table for full-adder

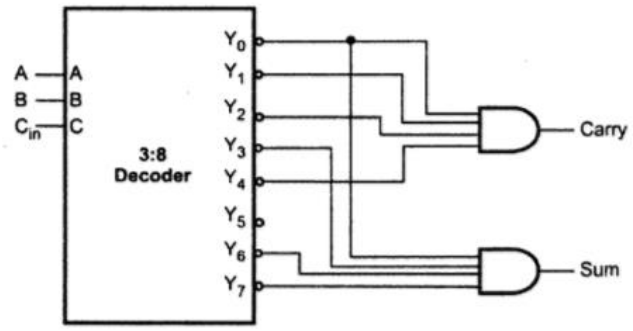


Fig. 4.143

Note : IC 74138 has active low outputs.

Example 4.51 : *Implement the following functions with a 8 : 1 MUX.*

i) F (w, x, y, z) = $\sum$ m (0, 1, 3, 5, 8, 9, 15)

ii) F (A, B, C) = π M (2, 3, 4, 7)

Solution : i)

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\bar{w}$	(0)	(1)	2	(3)	4	(5)	6	7
w	(8)	(9)	10	11	12	13	14	(15)
	1	1	0	$\bar{w}$	0	$\bar{w}$	0	w

(a) Implementation table

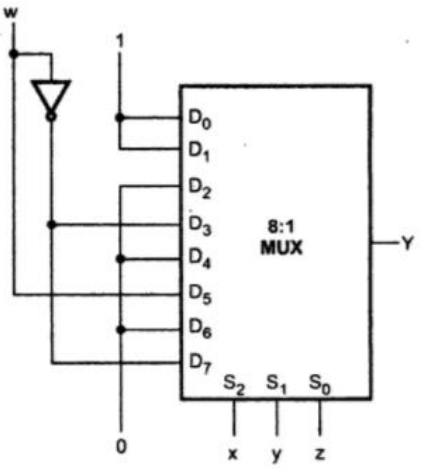


Fig. 4.144 **(b) Multiplexer implementation**

ii)

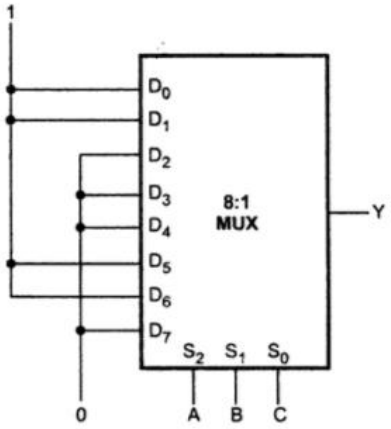


Fig. 4.145 Multiplexer implementation

Example 4.52 : *Implement the following multi-output combinational logic circuit using 4 line : 16 line decoder IC AND external gates :*

$F_1 = \sum m\ (2, 3, 9, 11)$ $F_2 = \sum m\ (10, 12, 13, 14)$ $F_3 = \sum m\ (2, 4, 8)$

Example 4.63 : *Write a data flow style VHDL program for 4 to 16 decoder.*
[Nov.-2004, Set-2, 8 Marks]

Solution : The VHDL program for a 4 to 16 decoder can be written as follows.

```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity Dec is
 Port G, GA, GB : in STD_LOGIC;
              A : (in STD_LOGIC_VECTOR (3 downto 0);
              Y : out STD_LOGIC_VECTOR (0 to 15);
end Dec;
architecture 4to16dec of Dec is
Signal Y_L : STD_LOGIC_VECTOR (0 to 15);
begin
     with A select Y_L <=
     "1000 0000 0000 0000" when "0000",
     "0100 0000 0000 0000" when "0001",
     "0010 0000 0000 0000" when "0010",
     "0001 0000 0000 0000" when "0011",
     "0000 1000 0000 0000" when "0100",
     "0000 0100 0000 0000" when "0101",
     "0000 0010 0000 0000" when "0110",
     "0000 0001 0000 0000" when "0111",
     "0000 0000 1000 0000" when "1000",
     "0000 0000 0100 0000" when "1001",
     "0000 0000 0010 0000" when "1010",
     "0000 0000 0001 0000" when "1011",
     "0000 0000 0000 1000" when "1100",
     "0000 0000 0000 0100" when "1101",
     "0000 0000 0000 0010" when "1110",
     "0000 0000 0000 0001" when "1111",
Y <= Y_L when (G and not GA and not GB) = '1'
else "0000 0000 0000 0000";
end 4to16dec;
```

Gray to Binary Conversion VHDL source code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity gray2binary is
Port ( g : in STD_LOGIC_VECTOR (3 downto 0);
b : out STD_LOGIC_VECTOR (3 downto 0));
end gray2binary;

architecture Behavioral of gray2binary is
begin
b(3)<= g(3);
b(2)<= g(3) xor g(2);
b(1)<= g(3) xor g(2) xor g(1);
b(0)<= g(3) xor g(2) xor g(1) xor g(0);
end behavioral;
```

Binary to Gray Conversion VHDL source code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity BinarytoGray is
port( b: in std_logic_vector(3 downto 0); --binary
```

```
g: out std_logic_vector(3 downto 0)); --gray
end BinarytoGray;

architecture behavioral of BinarytoGray is
begin
b(3)<= g(3);
b(2)<= g(3) xor g(2);
b(1)<= g(2) xor g(1);
b(0)<= g(1) xor g(0);
end behavioral;
```

5.2 Barrel Shifter

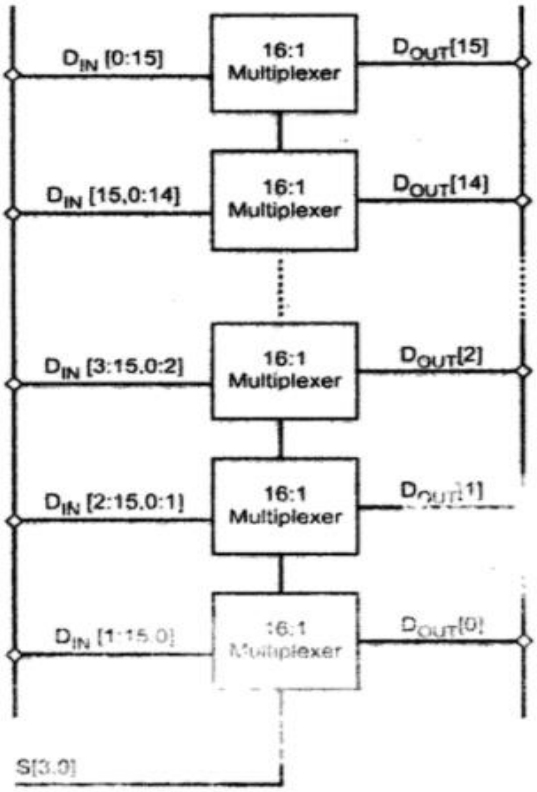


Fig. 5.1 Barrel shifter **using sixteen 16 : 1 multiplexers**

A barrel shifter is a combinational logic circuit. It has n data inputs, n data outputs, and a set of control inputs that specify how to shift the data between input and output. Let us see two different design approaches for barrel shifter.

The Fig. 5.1 shows the first approach to design barrel shifter. Here, sixteen 16 : 1 multiplexers are used to build the 16-bit barrel shifter. The multiplexer inputs are driven such that when we have to select inputs [S3 : 0] = 0000 normal data is at the output and when we have to select inputs [S3 : 0] = 0001 we get left shifted data by 1-bit at the output. Similarly, when we have to select inputs [S3 : 0] = 0010 we get left shifted data by 2-bit at the output. Therefore, the binary number on the select input decides the number of bit by which data is shifted on the left side. In this design approach we require sixteen 16 : 1 multiplexers (IC 74 × 150).

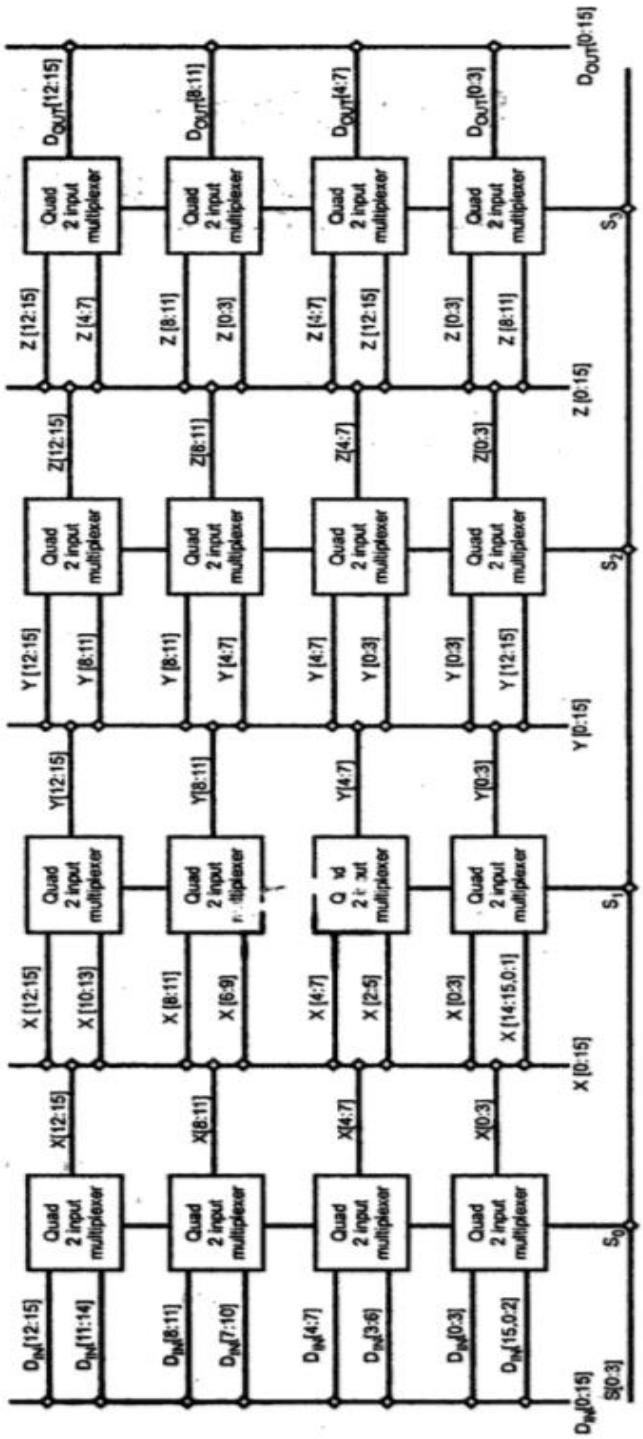


Fig. 5.2

The second approach to design barrel shifter is shown in Fig. 5.2. Here, sixteen quad 2 input multiplexers are used to design barrel shifter. The multiplexers are connected in four sets. The first set of multiplexers is controlled by S_0 to shift the input data by 0 or 1 bit. The data outputs of this set are connected to the inputs of a second set, controlled by S_1, which shifts its input word left by 0 or 2 bits. Similarly, the third and the fourth sets are controlled by S_2 and S_3 to shift selectively by 4 and 8 bits, as shown in Fig. 5.2.