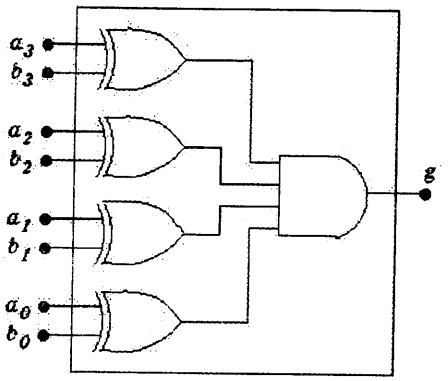
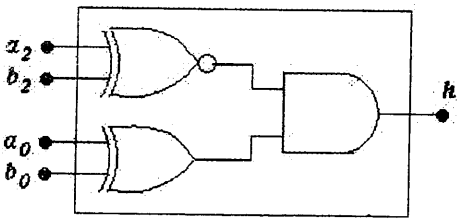
Chapter 8 Logic Components

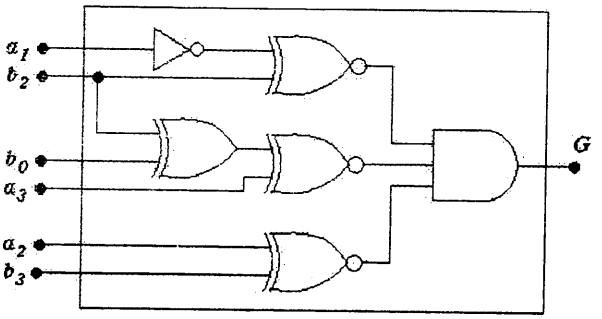
[8.1]



[8.2]
Note that only 4 out of 8 inputs are needed, namely a_0, a_2 and b_0, b_2, to produce the output h.

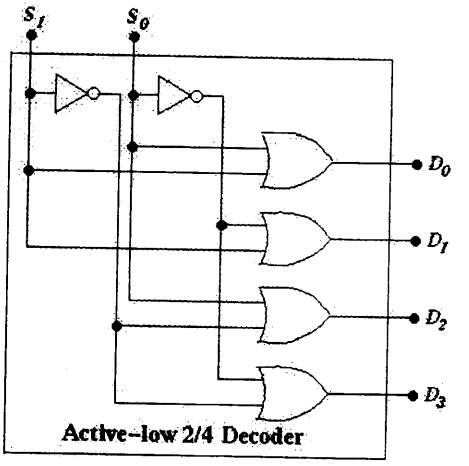


[8.3]



[8.4] For an active-low 2/4 decoder using OR gates,

$$D_0 = \overline{\overline{S_1} \cdot \overline{S_0}} = S_1 + S_0$$
$$D_1 = \overline{\overline{S_1} \cdot S_0} = S_1 + \overline{S_0}$$
$$D_2 = \overline{S_1 \cdot \overline{S_0}} = \overline{S_1} + S_0$$
$$D_3 = \overline{S_1 \cdot S_0} = \overline{S_1} + \overline{S_0}$$

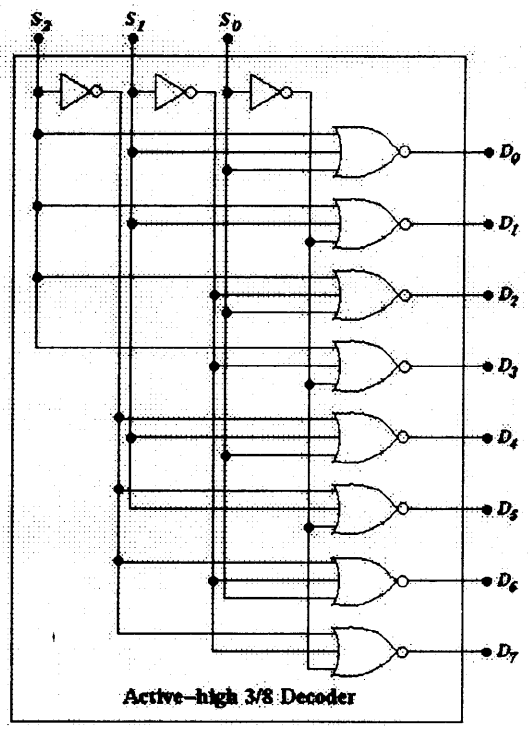


[8.5] An active-high 3/8 decoder can be designed using either AND or NOR gates as shown by the logic equations below:

$$D_0 = \overline{S_2} \cdot \overline{S_1} \cdot \overline{S_0} = \overline{S_2 + S_1 + S_0}$$
$$D_1 = \overline{S_2} \cdot \overline{S_1} \cdot S_0 = \overline{S_2 + S_1 + \overline{S_0}}$$
$$D_2 = \overline{S_2} \cdot S_1 \cdot \overline{S_0} = \overline{S_2 + \overline{S_1} + S_0}$$
$$D_3 = \overline{S_2} \cdot S_1 \cdot S_0 = \overline{S_2 + \overline{S_1} + \overline{S_0}}$$
$$D_4 = S_2 \cdot \overline{S_1} \cdot \overline{S_0} = \overline{\overline{S_2} + S_1 + S_0}$$
$$D_5 = S_2 \cdot \overline{S_1} \cdot S_0 = \overline{\overline{S_2} + S_1 + \overline{S_0}}$$
$$D_6 = S_2 \cdot S_1 \cdot \overline{S_0} = \overline{\overline{S_2} + \overline{S_1} + S_0}$$
$$D_7 = S_2 \cdot S_1 \cdot S_0 = \overline{\overline{S_2} + \overline{S_1} + \overline{S_0}}$$

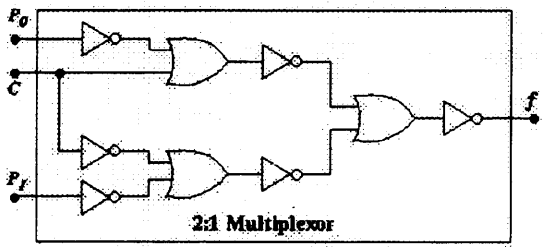
S_2	S_1	S_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

The logic diagram below shows the NOR-implementation of an active-high 3/8 decoder.

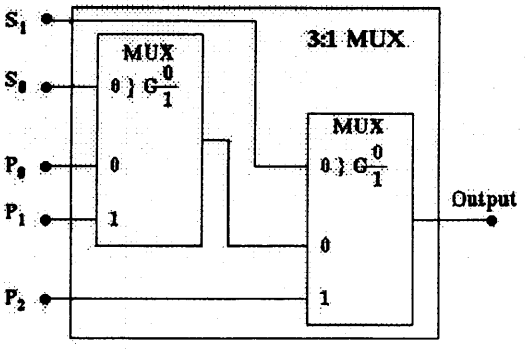


[8.6] $f = \overline{P_0 \cdot \overline{C} + P_1 \cdot C} = \overline{(\overline{\overline{P_0}} + C) + (\overline{\overline{P_1}} + \overline{C})}$

The logic diagram below shows the implementation using only inverters and OR gates.



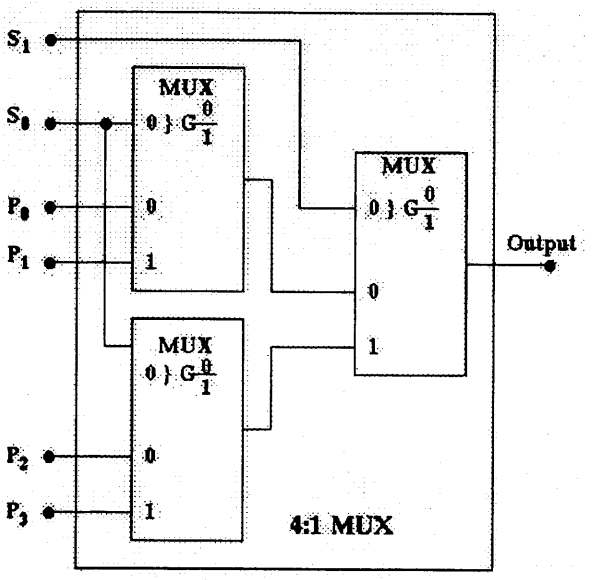
[8.7] A 3:1 MUX can be designed using two 2:1 MUX's as shown below:



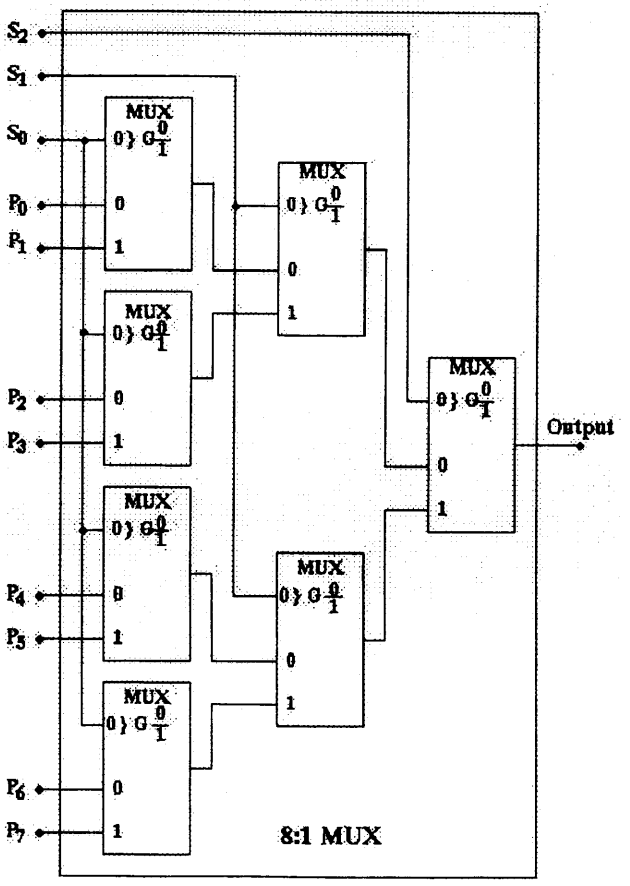
However, the two select codewords will return the same output, namely P_2, as indicated in the table below:

S_1	S_0	Output
0	0	P_0
0	1	P_1
1	0	P_2
1	1	P_2

[8.8]



[8.9]

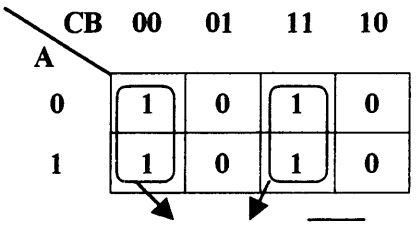


[8.10]

$f = \bar{b} \cdot \bar{a} \cdot 1 + \bar{b} \cdot a \cdot 0 + b \cdot \bar{a} \cdot 0 + b \cdot a \cdot 0 = \bar{b} \cdot \bar{a}$

[8.11]

$f = \bar{C} \cdot \bar{B} \cdot \bar{A} + \bar{C} \cdot \bar{B} \cdot A + C \cdot B \cdot \bar{A} + C \cdot B \cdot A$

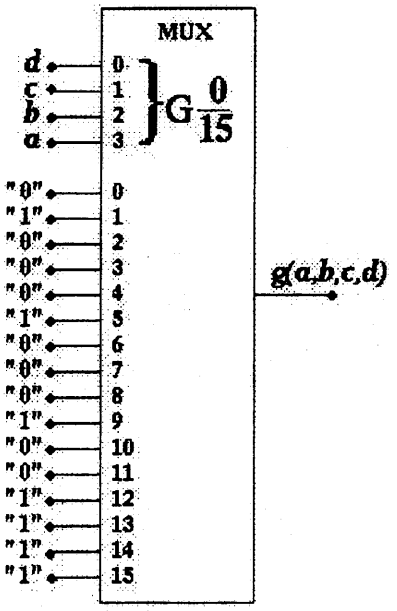


$f = \bar{C} \cdot \bar{B} + C \cdot B = \overline{C \oplus B}$

[8.12]

$g(a,b,c,d) = a \cdot b + \bar{c} \cdot d$

$= \sum m(1,5,9,12,13,14,15)$

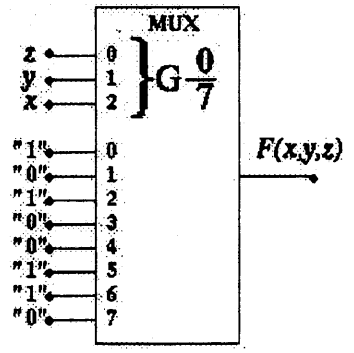


[8.13]

$F(x,y,z) = x \cdot \bar{y} \cdot z + (\bar{x} + y) \cdot \bar{z}$

$= x \cdot \bar{y} \cdot z + \bar{x} \cdot \bar{z} + y \cdot \bar{z}$

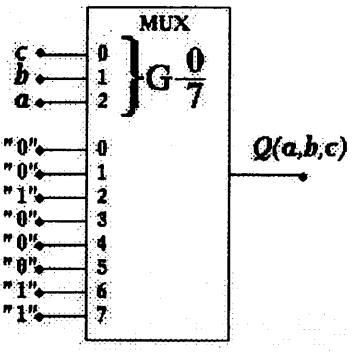
$= \sum m(0,2,5,6)$



[8.14]

$Q(a,b,c) = a \cdot b \cdot c + b \cdot \bar{c}$

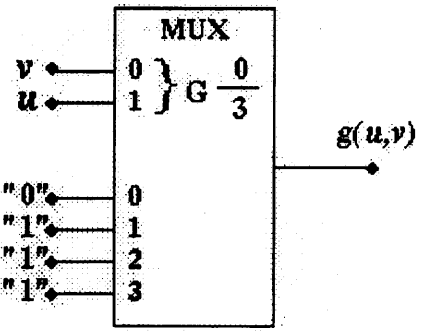
$= \sum m(2,6,7)$



[8.15]

$g(u,v) = u \cdot v + \bar{u} \cdot v + u \cdot \bar{v}$

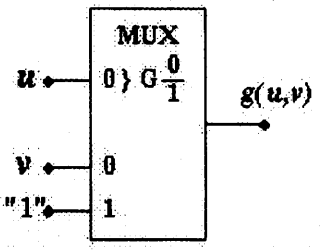
$= \sum m(1,2,3)$



The implementation using the 4:1 MUX is **not** at all an efficient design methodology in this case because the function can be minimized to simply an OR gate:

$g(u,v) = u \cdot v + \bar{u} \cdot v + u \cdot \bar{v} = u + \bar{u} \cdot v = u + v$

Another method to implement this function is to use a 2:1 MUX is as shown below:

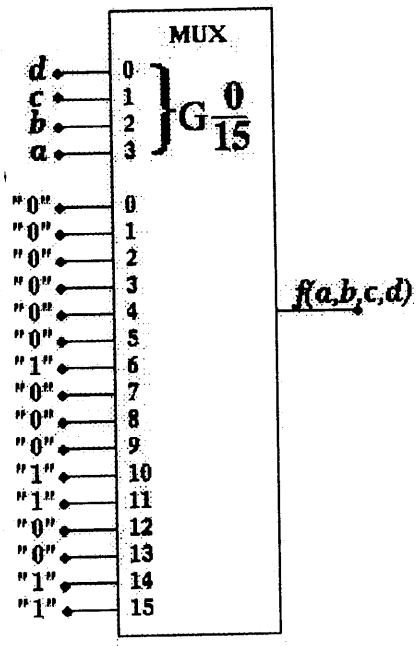


This will be a more efficient implementation than just merely passing "0" or "1" from the inputs of the MUX. Indeed, this technique can be quite efficient to reduce the size of the MUX required. However, in this case it cannot be more efficient than using a single OR gate.

[8.16]

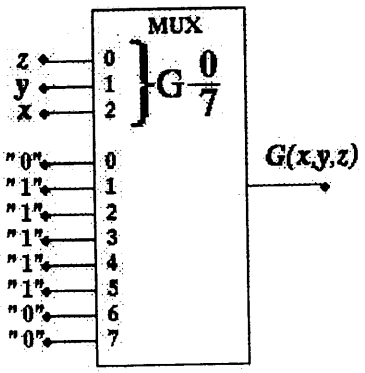
$f(a,b,c,d) = a \bullet \bar{b} \bullet c + b \bullet c \bullet \bar{d} + a \bullet b \bullet c \bullet d$

$= \Sigma m(6,10,11,14,15)$



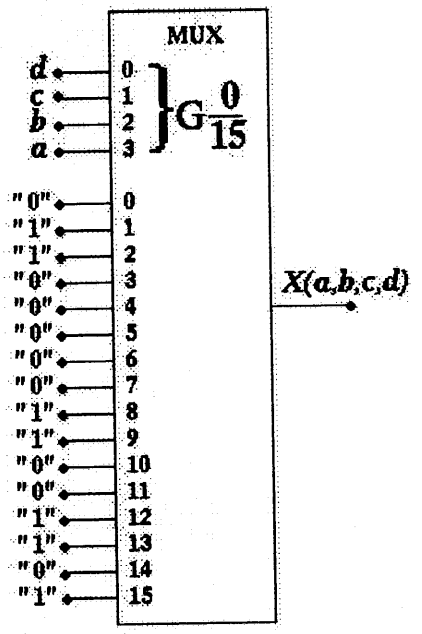
[8.17]

$G(x,y,z) = \Sigma m(1,2,3,4,5)$



[8.18]

$X(a,b,c,d) = \Sigma m(1,2,8,9,12,13,15)$



[8.19] A 1:8 DeMUX has a separate expression for each output as shown below where X is the input and P_0-P_7 are the outputs:

$$P_0 = \overline{S_2} \bullet \overline{S_1} \bullet \overline{S_0} \bullet X = \overline{S_2 + S_1 + S_0 + \overline{X}}$$

$$P_1 = \overline{S_2} \bullet \overline{S_1} \bullet S_0 \bullet X = \overline{S_2 + S_1 + \overline{S_0} + \overline{X}}$$

$$P_2 = \overline{S_2} \bullet S_1 \bullet \overline{S_0} \bullet X = \overline{S_2 + \overline{S_1} + S_0 + \overline{X}}$$

$$P_3 = \overline{S_2} \bullet S_1 \bullet S_0 \bullet X = \overline{S_2 + \overline{S_1} + \overline{S_0} + \overline{X}}$$

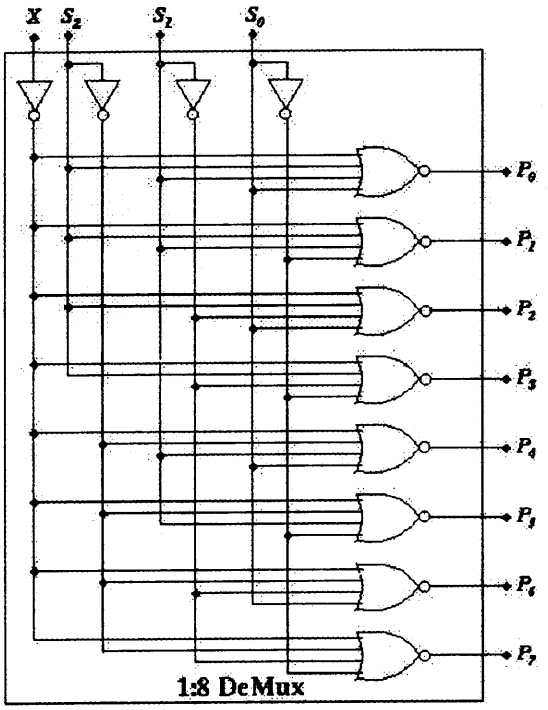
$$P_4 = S_2 \bullet \overline{S_1} \bullet \overline{S_0} \bullet X = \overline{\overline{S_2} + S_1 + S_0 + \overline{X}}$$

$$P_5 = S_2 \bullet \overline{S_1} \bullet S_0 \bullet X = \overline{\overline{S_2} + S_1 + \overline{S_0} + \overline{X}}$$

$$P_6 = S_2 \bullet S_1 \bullet \overline{S_0} \bullet X = \overline{\overline{S_2} + \overline{S_1} + S_0 + \overline{X}}$$

$$P_7 = S_2 \bullet S_1 \bullet S_0 \bullet X = \overline{\overline{S_2} + \overline{S_1} + \overline{S_0} + \overline{X}}$$

As the expressions show, a 1:8 DeMUX can be implemented either by AND gates or NOR gates. The logic diagram below shows the implementation using NOR gates.



[8.20]

(a) 11
1 1 0 0 +
0 1 0 1
1 0 0 0 1

(b) 1 1
0 1 1 0 +
0 1 1 0
1 1 0 0

(c) 1 1 1
0 0 1 1 +
0 1 0 1
1 0 0 0

[8.21]

(a)
1 1 1 1 1
1 1 0 0 0 1 1 0 +
0 1 0 1 1 1 0 0
1 0 0 1 0 0 0 1 0

(b)
1 1 1
1 1 1 1 0 0 0 0 +
1 0 1 0 1 0 1 0
1 1 0 0 1 1 0 1 0

(c)
1 1 1
1 0 1 0 0 0 1 0 +
1 1 1 1 0 0 0 1
1 1 0 0 1 0 0 1 1

[8.22]

(a)
1 1 1 1 1
0xAE = 1 0 1 0 1 1 1 0 +
0x37 = 0 0 1 1 0 1 1 1
1 1 1 0 0 1 0 1 = 0xE5

(b)
1 1 1 1
0x35 = 0 0 1 1 0 1 0 1 +
0xF2 = 1 1 1 1 0 0 1 0
1 0 0 1 0 0 1 1 1 = 0x127

(c)
1 1 1 1 1 1
0xB7 = 1 0 1 1 0 1 1 1 +
0x5C = 0 1 0 1 1 1 0 0
1 0 0 0 1 0 0 1 1 = 0x113

(d)
1
0x90 = 1 0 0 1 0 0 0 0 +
0x83 = 1 0 0 0 0 0 1 1
1 0 0 0 1 0 0 1 1 = 0x113

[8.23]

(a) 7 +A = 11
(b) 9 + B = 14
(c) 3 + F = 12
(d) 5 + 4 = 9
(e) E + C = 1A
(f) 1 + 9 = A

[8.24]

```
entity AOI_Gate_1 is
  port(a, b, c: in bit;
       f: out bit);
end AOI_Gate_1;

architecture Logic of AOI_Gate_1 is
begin
  f <= (a and b) nor (a and c) nor (b and c);
end Logic;

entity AOI_Gate_2 is
  port(a, b, c, d, e: in bit;
       f: out bit);
end AOI_Gate_2;

architecture Logic of AOI_Gate_2 is
begin
  f <= (a and b and c) nor (d and e);
end Logic;

entity Inv is
  port(x: in bit;
       y: out bit);
end Inv;

architecture Logic of Inv is
begin
  y <= not x;
end Logic;

entity OR_3 is
  port(a, b, c: in bit;
       f: out bit);
end OR_3;
```

```
architecture Logic of OR_3 is
begin
  f <= a or b or c;
end Logic;

entity AOI_full_adder is
  port(a_n, b_n, c_n: in bit;
       s_n, c_o: out bit);
end AOI_full_adder;

architecture Structural of AOI_full_adder is

component AOI_Gate_1
  port(a, b, c: in bit;
       f: out bit);
end component;
component AOI_Gate_2
  port(a, b, c, d, e: in bit;
       f: out bit);
end component;
component Inv
  port(x: in bit;
       y: out bit);
end component;
component OR3
  port(a, b, c: in bit;
       f: out bit);
end component;

signal X, Y, Z;

begin
  AOI1: AOI_Gate_1 port map
        (a_n, b_n, c_n, X);
  OR1: OR_3 port map
        (a_n, b_n, c_n, Y);
  AOI2: AOI_Gate_2 port map
        (a_n, b_n, c_n, X, Y, Z);
  INV1: Inv port map(X, c_o);
  INV2: Inv port map(Z, s_n);
end Structural;
```

[8.25]

```
entity 4bit_full_adder is
  port(a, b: in bit_vector (3 downto 0);
       c_in in bit;
       s: out bit_vector (3 downto 0);
       c_o: out bit);
end 4bit_full_adder;

architecture Structural of 4bit_full_adder is
component Full_Adder
  port(a, b, c_i: in bit;
       s, c_o: out bit);
end component;

signal c1, c2, c3;

begin
  FA1: Full_Adder port map
       (a(0), b(0), c_in, s(0), c1);
  FA2: Full_Adder port map
       (a(1), b(1), c1, s(1), c2);
  FA3: Full_Adder port map
       (a(2), b(2), c2, s(2), c3);
  FA4: Full_Adder port map
       (a(3), b(3), c3, s(3), c_out);
end Structural;
```

[8.26] Since a full-adder produces the sum bit first, the total delay = number of bits × delay for the carry-out bit.
(a) Total delay = 4×1.8 *ns* = 7.2 *ns*.
(b) Total delay = 8×1.8 *ns* = 14.4 *ns*.

[8.27] Because delay for the sum bit is greater than delay for the carry-out bit in this case, the total delay for 4-bit adder is 4× delay of the sum bit = 4×2.1 *ns* = 8.4 *ns*.

[8.28]

(a) 0111
1s complement: 1000
2s complement: 1001

(b) 1010
1s complement: 0101
2s complement: 0110

(c) 1111
1s complement: 0000
2s complement: 0001

(d) 10110101
1s complement: 01001010
2s complement: 01001011

(e) 11001100
1s complement: 00110011
2s complement: 00110100

(f) 10100101
1s complement: 01011010
2s complement: 01011011

[8.29]

(a) 14 = 1110
6 = 0110
⇒ 2s complement: 1010

```
1 1 1
  1 1 1 0 +
  1 0 1 0
1 1 0 0 0 = 8
```

(b) 34 = 0010 0010
21 = 0001 0101
⇒ 2s complement: 1110 1011

```
1 1 1       1
  0 0 1 0 0 0 1 0 +
  1 1 1 0 1 0 1 1
1 0 0 0 0 1 1 0 1 = 13
```

(c) 134 = 1000 0110
62 = 0011 1110
⇒ 2s complement: 1100 0010

```
1         1 1
  1 0 0 0 0 1 1 0 +
  1 1 0 0 0 0 1 0
1 0 1 0 0 1 0 0 0 = 72
```

(d) 196 = 1100 0100
118 = 0111 0110
⇒ 2s complement: 1000 1010

```
1
  1 1 0 0 0 1 0 0 +
  1 0 0 0 1 0 1 0
1 0 1 0 0 1 1 1 0 = 78
```

[8.30]

The following is the logic table for 1-bit addition:

Inputs			Outputs	
A_n	B_n	C_n	S_n	C_{n+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

After logic simplication,
$S_n = (A_n \oplus B_n) \oplus C_n$
$C_{n+1} = A_nB_n + (A_n \oplus B_n)C_n$

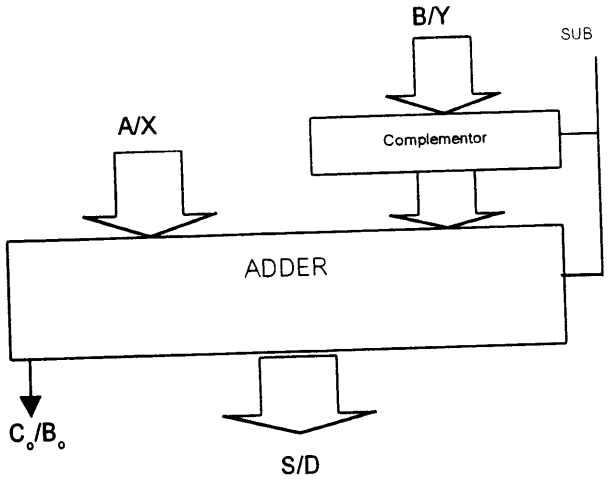
The following is the logic table for 1-bit subtraction:

Inputs			Outputs	
X_n	Y_n	B_n	D_n	B_{n+1}
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

After logic simplification,
$D_n = X_n \oplus (Y_n \oplus B_n)$ (Same as S_n in addition)
$B_{n+1} = Y_nB_n + (Y_n \oplus B_n)\overline{X_n}$

From the equations above, it shows the similarity between the two logic required for a 1-bit adder and a 1-bit subtractor.

The correct question to be asked is what is the advantage of using the 2s complement method over the borrowing method in subtraction. The answer is that using the 2s complement method will enable us to use the same hardware for both addition and subtraction with an additional circuitry for complementor which is usually much smaller than adding another substractor as shown by the block diagram below:



[8.31]

Bit	s	a_2	a_1	a_0
Decimal weight	-8	4	2	1

[8.32]

(a)

```
      1 0 1 0 (10) ×
      1 0 1 1 (11)
      1 0 1 0
    1 0 1 0
  1 0 1 0
  1 1 0 1 1 1 0 (110)
```

(b)

```
      1 1 1 0 (14) ×
      0 0 1 0 ( 2)
    1 1 1 0
    1 1 1 0 0 (28)
```

(c)

```
      1 0 1 1 (11) ×
      0 1 1 1 ( 7)
      1 0 1 1
    1 0 1 1
  1 0 1 1
  1 0 0 1 1 0 1 (77)
```

(d)

```
      0 1 0 0 ( 4) ×
      1 0 1 1 (11)
      0 1 0 0
    0 1 0 0
  0 1 0 0
  0 1 0 1 1 0 0 (44)
```

[8.33]

```
          0 0 1 1  0 1 0 0 (0x34) ×
          0 0 0 1  0 1 1 1 (0x17)
          0 0 1 1  0 1 0 0
        0 0 1 1 0  1 0 0
      0 0 1 1 0 1  0 0
  0 0 1 1  0 1 0 0
  0 1 0 0  1 0 1 0  1 1 0 0 (0x4AC)
```

[8.34] (a)

			a_2	a_1	a_0	×
			b_2	b_1	b_0	
c_5	c_4	c_3	c_2	c_1		
			a_2b_0	a_1b_0	a_0b_0	
		a_2b_1	a_1b_1	a_0b_1		
	a_2b_2	a_1b_2	a_0b_2			
p_5	p_4	p_3	p_2	p_1	p_0	

$p_0 = a_0b_0$

$p_1 = a_1b_0 \oplus a_0b_1 \oplus c_1$

$p_2 = a_2b_0 \oplus a_1b_1 \oplus a_0b_2 \oplus c_2$

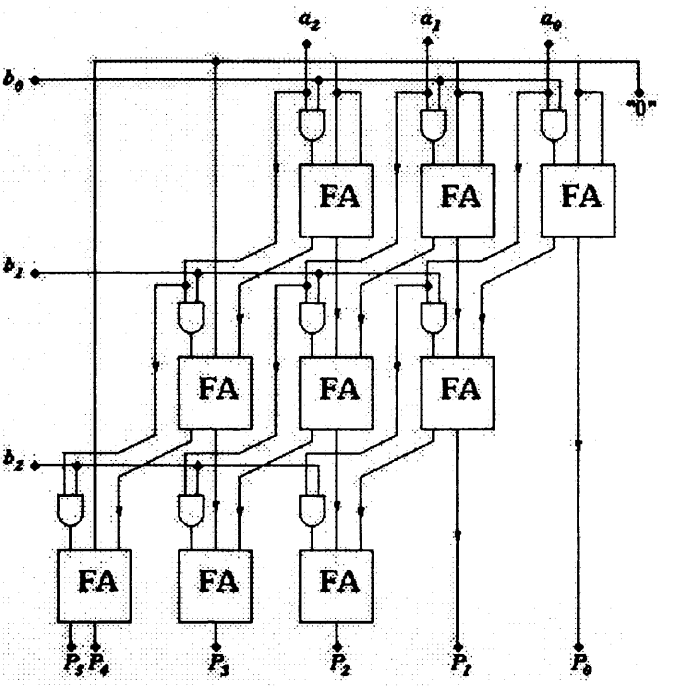
$p_3 = a_2b_1 \oplus a_1b_2 \oplus c_3$

$p_4 = a_2b_2 \oplus c_4$

$p_5 = c_5$

Note that c_i are the carry bits from the previous bit in addition.

(b)

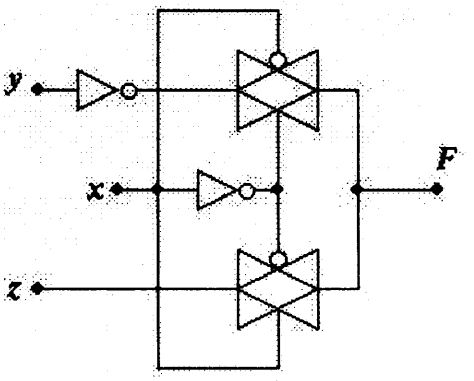


[8.35] Students who understand the concept of the transmission gate network well enough would be able to tell that there is something wrong about the question. Clearly, it would make more sense to design the transmission gate for the following functions:

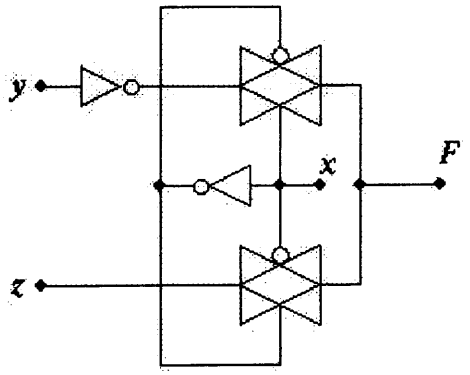
$F = \bar{x} \cdot \bar{y} + x \cdot z$ or $F = x \cdot \bar{y} + \bar{x} \cdot z$

rather than $F = x \cdot \bar{y} + x \cdot z$

The first network below shows the transmission gate network for $F = \bar{x} \cdot \bar{y} + x \cdot z$:

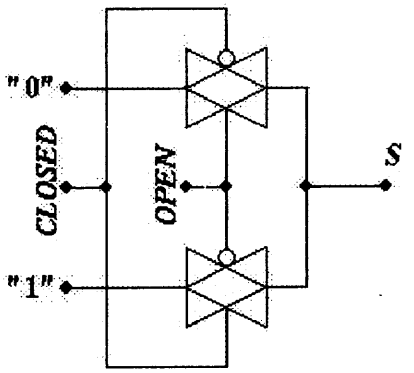


The second network below shows the transmission gate network for $F = x \cdot \bar{y} + \bar{x} \cdot z$:



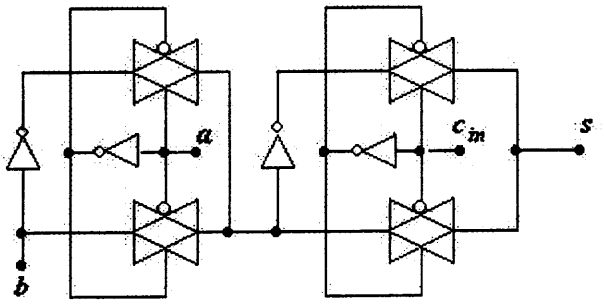
[8.36]

$S = OPEN \cdot 0 + CLOSE \cdot$



[8.37]

$s = \overline{c_{in}}(a \cdot \bar{b} + \bar{a} \cdot b) + c_{in} \cdot \overline{(a \cdot \bar{b} + \bar{a} \cdot b)}$



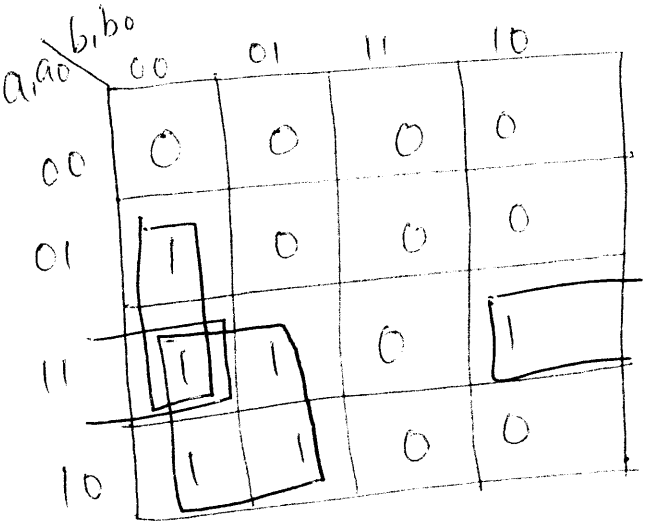
[8.38] The only similarity there is between CMOS inverter and a transmission gate is that both require one nFET and one pFET to implement. But there are many differences between the two as listed below:

1) A transmission gate does not require a power supply (V_{DD}) but it requires another input to drive the output to the appropriate voltages when either of the FET's is turned on whereas an inverter uses a power supply or ground to drive the output.
2) A transmission gate requires the signal and its complement at the gate of an nFET and a pFET respectively whereas an inverter only require the same gate signal for both nFET and pFET. Thus, in terms of layout, there are two poly gates in a transmission gate while only one poly gate for an inverter.
3) In terms of circuit operation, both nFET and pFET will be turned on and off at the same time for a transmission gate whereas only one of the FETs is on for an inverter. (Note that during the switching period where the output is changing from logic high to low or vice versa, both FETs of an inverter will be turned on but only for a short period of time.)

$A = a_1 a_0$
$B = b_1 b_0$

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

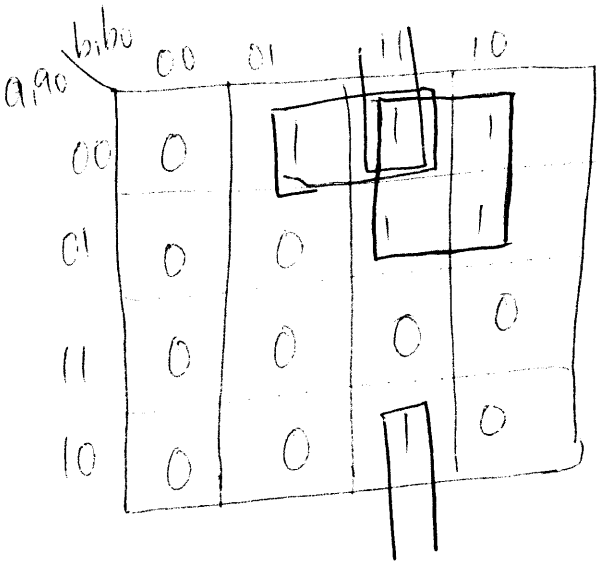
A>B



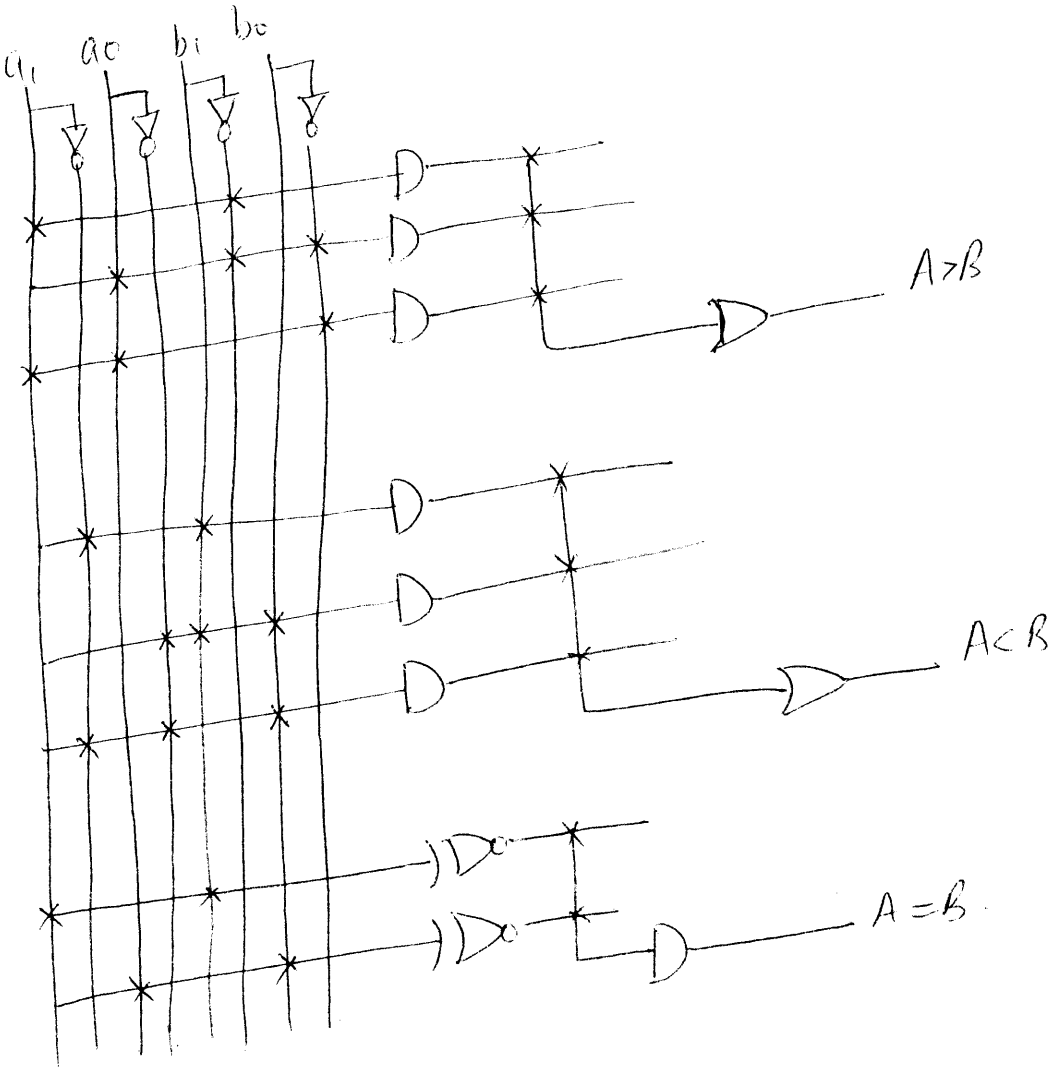
$A>B = a_1\bar{b_1} + a_0\bar{b_1}\bar{b_0} + a_1a_0\bar{b_0}$

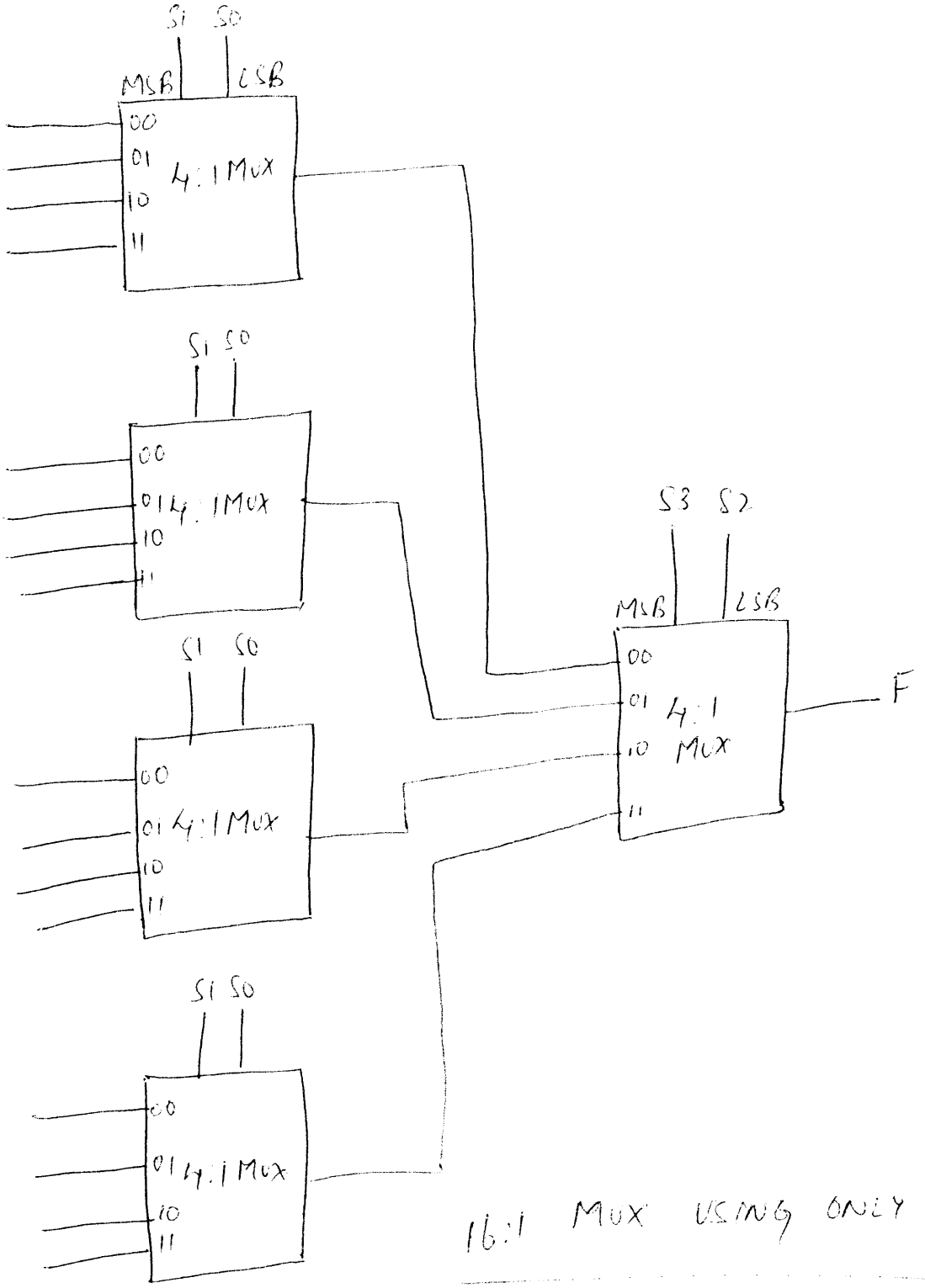
$A = B = \overline{(a_1 \oplus b_1)} \cdot \overline{(a_0 \oplus b_0)}$

$A < B$



$A < B = \bar{a_1} b_1 + \overline{a_0} b_1 b_0 + \bar{a_1} \overline{a_0} b_0$





16:1 MUX USING ONLY 4:1 MUX