Contents

Chapter 4

Standard Combinational Components

As with many construction projects, it is often easier to build in a hierarchical fashion. Initially, we use the very basic building blocks to build slightly larger building blocks, and then from these larger building blocks, we build yet larger building blocks, and so on. Similarly, in constructing large digital circuits, instead of starting with the basic logic gates as building blocks each time, we often start with larger building blocks. Many of these larger building blocks are often used over and over again in different digital circuits, and therefore, are considered as standard components for large digital circuits. In order to reduce the design time, these standard components are often made available in standard libraries so that they do not have to be redesigned each time that they are needed. For example, many digital circuits require the addition of two numbers; therefore, an adder circuit is considered a standard component and is available in most standard libraries.

Standard **combinational components** are combinational circuits that are available in standard libraries. These combinational components are used mainly in the construction of datapaths. For example, in our microprocessor road map, the standard combinational components are the multiplexer, ALU, comparator, and tri-state buffer. Other standard combinational components include adders, subtractors, decoders, encoders, shifters, rotators, and multipliers. Although the next-state logic and output logic circuits in the control unit are combinational circuits, they are not considered as standard combinational components because they are designed uniquely for a particular control unit to solve a specific problem and usually are never reused in another design.

In this chapter, we will design some standard combinational components. These components will be used in later chapters for the building of the datapath in the microprocessor. When we use these components to build the datapath, we do not need to know the detailed construction of these components. Instead, we only need to know how these components operate, and how they connect to other components. Nevertheless, in order to see the whole picture, we should understand how these individual components are designed.

4.1 Signal Naming Conventions

So far in our discussion, we have always used the words "high" and "low" to mean 1 or 0, or "on" or "off", respectively. However, this is somewhat arbitrary, and there is no reason why we can't say a 0 is a high or a 1 is off. In fact, many standard off-the-shelf components use what we call **negative logic** where 0 is for on and 1 is for off. Using negative logic usually is more difficult to understand because we are used to **positive logic** where 1 is for on and 0 is for off. In all of our discussions, we will use the more natural, positive logic that we are familiar with.

Nevertheless, in order to prevent any confusion as to whether we are using positive logic or negative logic, we often use the words "assert," "de-assert," "active-high," and "active-low." Regardless of whether we are using positive or negative logic, **active-high** always means that a 1 (i.e., a high) will cause the signal to be active or enabled and that a 0 will cause the signal to be inactive or disabled. For example, if there is an active-high signal called *add* and we want to enable it (i.e. to make it do what it is intended for, which in this case is to add something), then we need to set this signal line to a 1. Setting this signal to a 0 will cause this signal to be disabled or inactive. An **active-low** signal, on the other hand, means that a 0 will cause the signal to be active or enabled, and that a 1 will cause the signal to be inactive or disabled. So if the signal *add* is an active-low signal, then we need to set it to a 0 to make it add something.

We also use the word "**assert**" to mean: to make a signal active or to enable the signal. To **de-assert** a signal is to disable the signal or to make it inactive. For example, to assert the active-high *add* signal line means to set the *add* signal to a 1. To de-assert an active-low line also means to set the line to a 1—since a 0 will enable the line (active-low)—and we want to disable (de-assert) it.

4.2 Adder

4.2.1 Full Adder

To construct an adder for adding two *n*-bit binary numbers, $X = x_{n-1} \ldots x_0$ and $Y = y_{n-1} \ldots y_0$, we need to first consider the addition of a single bit slice, x_i with y_i, together with the carry-in bit, c_i, from the previous bit position on the right. The result from this addition is a sum bit, s_i, and a carry-out bit, c_{i+1}, for the next bit position. In other words, $s_i = x_i + y_i + c_i$, and $c_{i+1} = 1$ if there is a carry from the addition to the next bit on the left. Note that the + operator in this equation is addition and not the logical OR.

For example, consider the following addition of the two 4-bit binary numbers, $X = 1001$ and $Y = 0011$.

The result of the addition is 1100. The addition is performed just like that for decimal numbers, except that there is a carry whenever the sum is either a 2 or a 3 in decimals, since 2 is 10 in binary and 3 is 11. The most significant bit in the 10 or the 11 is the carry-out bit. Looking at the bit slice that is highlighted in blue where $x_1 = 0$, $y_1 = 1$, and $c_1 = 1$, the addition for this bit slice is $x_1 + y_1 + c_1 = 0 + 1 + 1 = 10$. Therefore, the sum bit is $s_1 = 0$, and the carry-out bit is $c_2 = 1$.
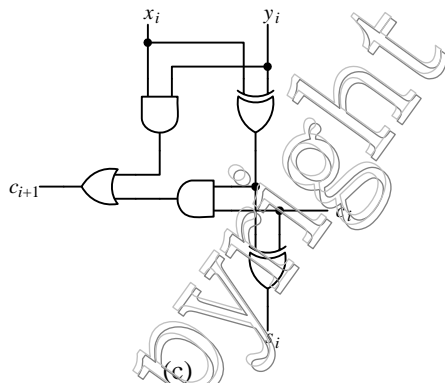
The circuit for the addition of a single bit slice is known as a **full adder** (**FA**), and its truth table is shown in Figure 4.1(a). The derivation of the equations for s_i and c_{i+1} are shown in Figure 4.1(b). From these two equations, we get the circuit for the full adder, as shown in Figure 4.1(c). Figure 4.1(d) shows the logic symbol for it. The dataflow VHDL code for the full adder is shown in Figure 4.2.

x_i	y_i	c_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(a)

$$\begin{aligned}
s_i &= x_i'y_i'c_i + x_i'y_ic_i' + x_iy_i'c_i' + x_iy_ic_i \\
&= (x_i'y_i + x_iy_i')c_i' + (x_i'y_i' + x_iy_i)c_i \\
&= (x_i \oplus y_i)c_i' + (x_i \oplus y_i)'c_i \\
&= x_i \oplus y_i \oplus c_i
\end{aligned}$$

$$\begin{aligned}
c_{i+1} &= x_i'y_ic_i + x_iy_i'c_i + x_iy_ic_i' + x_iy_ic_i \\
&= x_iy_i(c_i' + c_i) + c_i(x_i'y_i + x_iy_i') \\
&= x_iy_i + c_i(x_i \oplus y_i)
\end{aligned}$$

(b)

(c)

(d)

Figure 4.1 Full adder: (a) truth table; (b) equations for s_i and c_{i+1}; (c) circuit; (d) logic symbol.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY fa IS PORT (
  Ci, Xi, Yi: IN STD_LOGIC;
  Ci1, Si: OUT STD_LOGIC);
END fa;
```

```
ARCHITECTURE Dataflow OF fa IS
BEGIN
  Ci1 <= (Xi AND Yi) OR (Ci AND (Xi XOR Yi));
  Si <= Xi XOR Yi XOR Ci;
END Dataflow;
```

Figure 4.2 Dataflow VHDL code for a 1-bit full adder.

4.2.2 Ripple-carry Adder

The full adder is for adding two operands that are only one bit wide. To add two operands that are, say, four bits wide, we connect four full adders together in series. The resulting circuit (shown in Figure 4.3) is called a **ripple-carry adder** for adding two 4-bit operands.

Since a full adder adds the three bits, x_i, y_i and c_i, together, we need to set the first carry-in bit, c_0, to 0 in order to perform the addition correctly. Moreover, the output signal, c_{out}, is a 1 whenever there is an overflow in the addition.

The structural VHDL code for the 4-bit ripple-carry adder is shown in Figure 4.4. Since we need to duplicate the full adder component four times, we can use either the PORT MAP statement four times or the FOR-GENERATE statement, as shown in the code, to automatically generate the four components. The statement FOR k IN 3 DOWNTO 0 GENERATE determines how many times to repeat the PORT MAP statement that is in the body of the GENERATE statement and the values used for k. The vector signal *Carryv* is used to propagate the carry bit from one FA to the next.

Figure 4.3 Ripple-carry adder.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY Adder4 IS PORT (
  A, B: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
  Cout: OUT STD_LOGIC;
  SUM: OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END Adder4;

ARCHITECTURE Structural OF Adder4 IS
  COMPONENT FA PORT (
    ci, xi, yi: IN STD_LOGIC;
    co, si: OUT STD_LOGIC);
  END COMPONENT;

  SIGNAL Carryv: STD_LOGIC_VECTOR(4 DOWNTO 0);

BEGIN
  Carryv(0) <= '0';

  Adder: FOR k IN 3 DOWNTO 0 GENERATE
    FullAdder: FA PORT MAP (Carryv(k), A(k), B(k), Carryv(k+1), SUM(k));
```

```
   END GENERATE Adder;

   Cout <= Carryv(4);
END Structural;
```

Figure 4.4 VHDL code for a 4-bit ripple-carry adder using a FOR-GENERATE statement.

4.2.3 * Carry-Lookahead Adder

The ripple-carry adder is slow because the carry-in for each full adder is dependent on the carry-out signal from the previous FA. So before FA_i can output valid data, it must wait for FA_{i-1} to have valid data. In the **carry-lookahead adder**, each bit slice eliminates this dependency on the previous carry-out signal and instead uses the values of the two input operands, X and Y, directly to deduce the needed signals. This is possible from the following observations regarding the carry-out signal. For each FA_i, the carry-out signal, c_{i+1}, is set to a 1 if either one of the following two conditions is true:

$x_i = 1$ and $y_i = 1$

or

$(x_i = 1$ or $y_i = 1)$ and $c_i = 1$

In other words,

$$c_{i+1} = x_i y_i + c_i(x_i + y_i) \tag{4.1}$$

At first glance, this carry-out equation looks completely different from the carry-out equation deduced in Figure 4.1(b). However, they are equivalent (see Problem P2.6(h)).

If we let

$g_i = x_i \, y_i$

and

$p_i = x_i \, + y_i$

then Equation (4.1) can be rewritten as

$$c_{i+1} = g_i + p_i c_i \tag{4.2}$$

Using Equation (4.2) for c_{i+1}, we can recursively expand it to get the carry-out equations for any bit slice, c_i, that is dependent only on the two input operands, X and Y, and the initial carry-in bit, c_0. Using this technique, we get the following carry-out equations for the first four bit slices

$$c_1 \quad = g_0 + p_0 c_0 \tag{4.3}$$

$$\begin{aligned} c_2 \quad &= g_1 + p_1 c_1 \\ &= g_1 + p_1(g_0 + p_0 c_0) \\ &= g_1 + p_1 g_0 + p_1 p_0 c_0 \end{aligned} \tag{4.4}$$

$$\begin{aligned} c_3 \quad &= g_2 + p_2 c_2 \\ &= g_2 + p_2 \, (g_1 + p_1 g_0 + p_1 p_0 c_0) \\ &= g_2 + p_2 \, g_1 + p_2 p_1 g_0 + p_2 p_1 \, p_0 c_0 \end{aligned} \tag{4.5}$$

$$\begin{aligned} c_4 \quad &= g_3 + p_3 c_3 \\ &= g_3 + p_3(g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0) \\ &= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0 \end{aligned} \tag{4.6}$$

Using Equations (4.3) to (4.6), we obtain the circuit for generating the carry-lookahead signals for c_1 to c_4, as shown in Figure 4.5(a). Note that each equation is translated to a three-level combinational logic—one level for generating the g_i and p_i, and two levels (for the sum-of-products format) for generating the c_i expression. This carry-

lookahead circuit can be reduced even further because we want c_0 to be a 0 when performing additions and this 0 will cancel the rightmost product term in each equation.

The full adder for the carry-lookahead adder can also be made simpler, since it is no longer required to generate the carry-out signal for the next bit slice. In other words, the carry-in signal for the full adder now comes from the new carry-lookahead circuit rather than from the carry-out signal of the previous bit slice. Thus, this full adder only needs to generate the sum_i signal. Figure 4.5(b) shows one bit slice of the carry-lookahead adder. For an n-bit carry-lookahead adder, we use n bit slices. These n bit slices are not connected in series as with the ripple-carry adder; otherwise, it defeats the purpose of having the more complicated carry-out circuit.
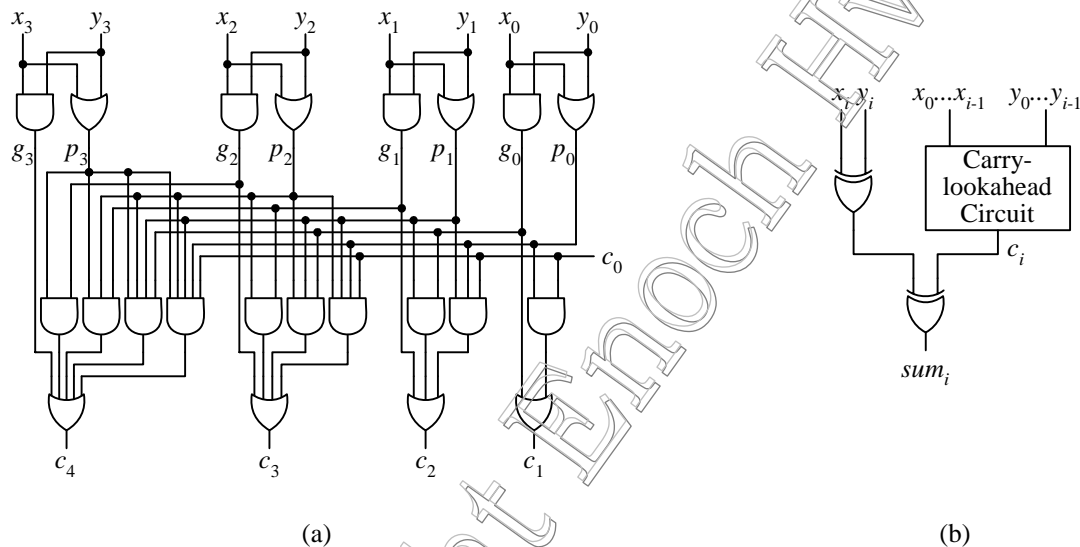

(a)   (b)

Figure 4.5 (a) Circuit for generating the carry-lookahead signals, c_1 to c_4; (b) one bit slice of the carry-lookahead adder.

4.3 Two's Complement Binary Numbers

Before introducing subtraction circuits, we need to review how negative numbers are encoded using **two's complement** representation. Binary encoding of numbers can be interpreted as either signed or unsigned. Unsigned numbers include only positive numbers and zero, whereas signed numbers include positive, negative, and zero. For signed numbers, the most significant bit (MSB) tells whether the number is positive or negative. If the most significant bit is a 1, then the number is negative; otherwise, the number is positive. The value of a positive signed number is obtained exactly as for unsigned numbers described in Section 2.1. For example, the value for the positive signed number 01101001_2 is just $1 \times 2^6 + 1 \times 2^5 + 1 \times 2^3 + 1 \times 2^0 = 105$ in decimal format.

However, to determine the value of a negative signed number, we need to perform a two-step process: (1) flip all the 1 bits to 0's and all the 0 bits to 1's, and (2) add a 1 to the result obtained from Step 1. The number obtained from applying this two-step process is evaluated as an unsigned number for its value. The negative of this resulting value is the value of the original negative signed number.

Example 4.1: Finding the value for a signed number

Given the 8-bit signed number 11101001_2, we know that it is a negative number because of the leading 1. To find out the value of this negative number, we perform the two-step process as follows.

11101001	(original number)
00010110	(flip bits)
00010111	(add a 1 to the previous number)

The value for the resulting number 00010111 is $1 \times 2^4 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 23$. Therefore, the value of the original number 11101001 is negative 23 (–23).   ♦

Example 4.2: Finding the value for a signed number

To find the value for the 4-bit signed number 1000, we apply the two-step process to the number as follows.

1000	(original number)
0111	(flip bits)
1000	(add a 1 to the previous number)

The resulting number 1000 is exactly the same as the original number! This, however, should not confuse us if we follow exactly the instructions for the conversion process. We need to interpret the resulting number as an unsigned number to determine the value. Interpreting the resulting number 1000 as an unsigned number gives us the value of 8. Therefore, the original number, which is also 1000, is negative 8 (–8). ♦

Figure 4.6 shows the two's complement numbers for four bits. The range goes from –8 to 7. In general, for an n-bit two's complement number, the range is from -2^{n-1} to $2^{n-1} - 1$.

4-bit Binary	Two's Complement
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	– 8
1001	– 7
1010	– 6
1011	– 5
1100	– 4
1101	– 3
1110	– 2
1111	– 1

Figure 4.6 4-bit two's complement numbers.

The nice thing about using two's complement to represent negative numbers is that when we add a number with the negative of the same number, the result is zero as expected. This is shown in the next example.

Example 4.3: Adding 4-bit signed numbers

Use 4-bit signed arithmetic to perform the following addition.

$$
\begin{array}{rcr}
3 & = & 0011 \\
+ (-3) & = & + 1101 \\
\hline
0 & = & 10000
\end{array}
$$

The result 10000 has five bits. But since we are using 4-bit arithmetic (that is, the two operands are 4-bits wide) the result must also be in 4-bits. The leading 1 in the result is, therefore, an overflow bit. By dropping the leading one, the remaining result 0000 is the correct answer for the problem. Although this addition resulted in an overflow bit, by dropping this extra bit, we obtained the correct answer. ♦

Example 4.4: Adding 4-bit signed numbers

Use 4-bit signed arithmetic to perform the following addition.

$$
\begin{array}{rcl}
6 & = & 0110 \\
\underline{+\,3} & = & \underline{+\,0011} \\
9 & \neq & 1001
\end{array}
$$

The result 1001 is a 9 if we interpret it as an unsigned number. However, since we are using signed numbers, we need to interpret the result as a signed number. Interpreting 1001 as a signed number gives –7, which of course is incorrect. The problem here is that the range for a 4-bit signed number is from –8 to +7, and +9 is outside of this range.   ◆

Although the addition in this example did not resulted in an overflow bit, but the final answer is incorrect. In order to correct this problem, we need to add (at least) one extra bit by sign extending the number. The corrected arithmetic is shown in Example 4.5.

Example 4.5: Adding 5-bit signed numbers

Use 5-bit signed arithmetic to perform the following addition.

$$
\begin{array}{rcl}
6 & = & 00110 \\
\underline{+\,3} & = & \underline{+\,00011} \\
9 & = & 01001
\end{array}
$$

The result 01001, when interpreted as a signed number, is 9.   ◆

To extend a signed number, we need to add leading 0's or 1's depending on whether the original most significant bit is a 0 or a 1. If the most significant bit is a 0, we sign extend the number by adding leading 0's. If the most significant bit is a 1, we sign extend the number by adding leading 1's. By performing this sign extension, the value of the number is not changed, as shown in Example 4.6.

Example 4.6: Performing sign extensions

Sign extend the numbers 10010 and 0101 to 8-bits.

For the number 10010, since the most significant bit is a 1, therefore, we need to add leading 1's to make the number 8-bits long. The resulting number is 11110010. For the number 0101, since the most significant bit is a 0, therefore, we need to add leading 0's to make the number 8-bits long. The resulting number is 00000101. The following shows that the two resulting numbers have the same value as the two original numbers. Since the first number is negative (because of the leading 1 bit) we need to perform the two-step process to evaluate its value. The second number is positive, so we can evaluate its value directly.

	Original Number	Sign Extended		Original Number	Sign Extended
	10010	11110010		0101	00000101
Flip bits	01101	00001101			
Add 1	01110	00001110			
Value	– 14	– 14		5	5

◆

4.4 Subtractor

We can construct a one-bit subtractor circuit similar to the method used for constructing the full adder. However, instead of the sum bit, s_i, for the addition, we have a difference bit, d_i, for the subtraction, and instead of having carry-in and carry-out signals, we have borrow-in (b_i) and borrow-out (b_{i+1}) signals. So, when we subtract the ith bit of the two operands, x_i and y_i, we get the difference of $d_i = x_i - y_i$. If, however, the previous bit on the right has to borrow from this i^{th} bit, then input b_i will be set to a 1, and the equation for the difference will be $d_i = x_i - b_i - y_i$. On the other hand, if the ith bit has to borrow from the next bit on the left for the subtraction, then the output b_{i+1} will be set to a 1. The value borrowed is a 2, and so the resulting equation for the difference will be $d_i = x_i - b_i + 2b_{i+1} - y_i$. Note that the symbols + and − used in this equation are for addition and subtraction, and not for logical operations. The term $2b_{i+1}$ is "2 multiply by b_{i+1}." Since b_{i+1} is a 1 when we have to borrow and we borrow a 2 each

time, the equation just adds a 2 when there is a borrow. When there is no borrow, b_{i+1} is 0, and so the term b_{i+1} cancels out to 0.

For example, consider the following subtraction of the two 4-bit binary numbers, $X = 0100$ and $Y = 0011$:

Consider the bit position that is highlighted in blue. Since the subtraction for the previous bit on the right has to borrow, b_i is a 1. Moreover, b_{i+1} is also a 1, because the current bit has to borrow from the next bit on the left. When it borrows, it gets a 2. Therefore, $d_i = x_i - b_i + 2b_{i+1} - y_i = 0 - 1 + 2(1) - 1 = 0$.

The truth table for the 1-bit subtractor is shown in Figure 4.7(a), from which the equations for d_i and b_{i+1}, as shown in Figure 4.7(b), are derived. From these two equations, we get the circuit for the subtractor, as shown in Figure 4.7(c). Figure 4.7(d) shows the logic symbol for the subtractor.
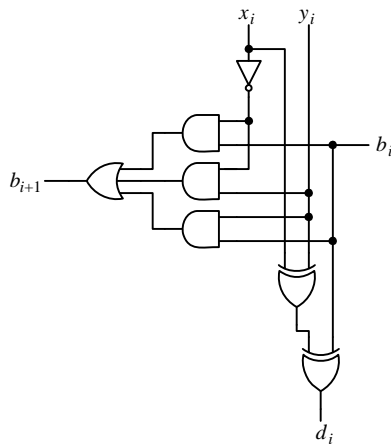
Building a subtractor circuit for subtracting an n-bit operand can be done by daisy-chaining n 1-bit subtractor circuits together, similar to the adder circuit shown in Figure 4.3. However, there is a much better subtractor circuit, as shown in the next section.

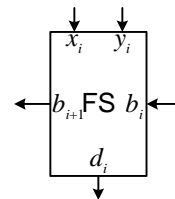| x_i	y_i	b_i	b_{i+1}	d_i
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

$$
\begin{aligned}
d_i &= x_i'y_i'b_i + x_i'y_ib_i' + x_iy_i'b_i' + x_iy_ib_i \\
&= (x_i'y_i + x_iy_i')b_i' + (x_i'y_i' + x_iy_i)b_i \\
&= (x_i \oplus y_i)b_i' + (x_i \oplus y_i)'b_i \\
&= x_i \oplus y_i \oplus b_i
\end{aligned}
$$

$$
\begin{aligned}
b_{i+1} &= x_i'y_i'b_i + x_i'y_ib_i' + x_i'y_ib_i + x_iy_ib_i \\
&= x_i'b_i(y_i' + y_i) + x_i'y_i(b_i' + b_i) + y_ib_i(x_i' + x_i) \\
&= x_i'b_i + x_i'y_i + y_ib_i
\end{aligned}
$$

(a)

(b)

(c)

(d)

Figure 4.7 1-bit subtractor: (a) truth table; (b) equations for d_i and b_{i+1}; (c) circuit; (d) logic symbol.

4.5   Adder-Subtractor Combination

It turns out that, instead of having to build a separate adder and subtractor units, we can modify the ripple-carry adder (or the carry-lookahead adder) slightly to perform both operations. The modified circuit performs subtraction by adding the negated value of the second operand. In other words, instead of performing the subtraction $A - B$, the addition operation $A + (-B)$ is performed.

Recall that in two's complement representation, to negate a value involves inverting all the 0 bits to 1's and 1's to 0's, and then adding a 1. Hence, we need to modify the adder circuit so that we selectively can do either one of two things: (1) flip the bits of the B operand and then add an extra 1 for the subtraction operation, or (2) not flip the bits and not add an extra 1 for the addition operation.

For this adder-subtractor combination circuit (in addition to the two input operands A and B), a select signal, s, is needed to select which operation to perform. The assignment of the two operations to the select signal s is shown in Figure 4.8(a). When $s = 0$, we want to perform an addition, and when $s = 1$, we want to perform a subtraction. When $s = 0$, B does not need to be modified, and like the adder circuit from Section 4.2.2, the initial carry-in signal c_0 needs to be set to a 0. On the other hand, when $s = 1$, we need to invert the bits in B and add a 1. The addition of a 1 is accomplished by setting the initial carry-in signal c_0 to a 1. Two circuits are needed for handling the above situations: one for inverting the bits in B and one for setting c_0. Both of these circuits are dependent on s.

The truth table for these two circuits is shown in Figure 4.8(b). The input variable b_i is the ith bit of the B operand. The output variable y_i is the output from the circuit that either inverts or does not invert the bits in B. From this truth table, we can conclude that the circuit for y_i is just a 2-input XOR gate, while the circuit for c_0 is just a direct connection from s. Putting everything together, we obtain the adder-subtractor combination circuit (for four bits), as shown in Figure 4.8(c). The logic symbol for the circuit is shown in Figure 4.8(d).
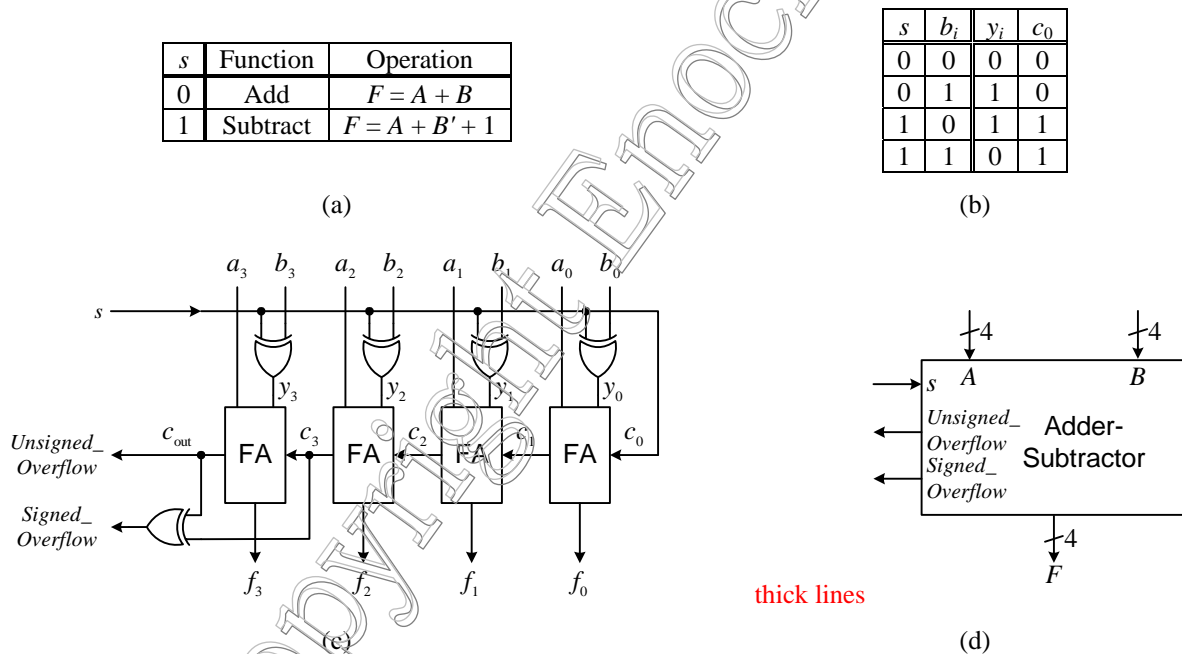
s	Function	Operation
0	Add	$F = A + B$
1	Subtract	$F = A + B' + 1$

(a)

s	b_i	y_i	c_0
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

(b)

(c)

(d)

thick lines

Figure 4.8 Adder-subtractor combination: (a) operation table; (b) truth table for y_i and c_0; (c) circuit; (d) logic symbol.

Notice the adder-subtractor circuit in Figure 4.8(c) has two different overflow signals, *Unsigned_Overflow* and *Signed_Overflow*. This is because the circuit can deal with both signed and unsigned numbers. When working with unsigned numbers only, the output signal *Unsigned_Overflow* is sufficient to determine whether there is an overflow or not. However, for signed numbers, we need to perform the XOR of *Unsigned_Overflow* with c_3, producing the *Signed_Overflow* signal in order to determine whether there is an overflow or not.

For example, the valid range for a 4-bit *signed* number goes from -2^3 to $2^3 - 1$ (i.e., from –8 to 7). Adding the two signed numbers, $4 + 5 = 9$ should result in a signed number overflow, since 9 is outside the range. However, the valid range for a 4-bit *unsigned* number goes from 0 to $2^4 - 1$ (i.e., 0 to 15). If we treat the two numbers 4 and 5 as unsigned numbers, then the result of adding these two unsigned numbers, 9, is inside the range. So when adding the two numbers 4 and 5, the *Unsigned_Overflow* signal should be de-asserted, while the *Signed_Overflow* signal should be asserted. Performing the addition of $4 + 5$ in binary as shown here:

$$c_3$$

```
              0 / 1   0   0
Unsigned  +   0 ↘1  1   0   1
Overflow  ↘ 0   1   0   0   1
```

```
                0 XOR 1 = 1
Signed    ─────────────────↗
Overflow
```

we get $0100 + 0101 = 1001$, which produces a 0 for the *Unsigned_Overflow* signal. However, the addition produces a 1 for c_3, and XORing these two values, 0 for *Unsigned_Overflow* and 1 for c_3, results in a 1 for the *Signed_Overflow* signal.

In another example, adding the two 4-bit signed numbers, $-4 + (-3) = -7$ should not result in a signed overflow. Performing the arithmetic in binary, $-4 = 1100$ and $-3 = 1101$, as shown here:

$$c_3$$

```
              1 / 1   0   0
Unsigned  +   1 ↘1  1   0   1
Overflow  ↘ 1   1   0   0   1
```

```
                1 XOR 1 = 0
Signed    ─────────────────↗
Overflow
```

we get $1100 + 1101 = 11001$, which produces a 1 for both *Unsigned_Overflow* and c_3. XORing these two values together gives a 0 for the *Signed_Overflow* signal. On the other hand, if we treat the two binary numbers, 1100 and 1101, as unsigned numbers, then we are adding $12 + 13 = 25$. Then 25 is outside the unsigned number range, and so the *Unsigned_Overflow* signal should be asserted.

The behavioral VHDL code for the 4-bit adder-subtractor combination circuit is shown in Figure 4.9. The GENERIC keyword declares a read-only constant identifier, *n*, of type INTEGER having a default value of 4. This constant identifier then is used in the declaration of the STD_LOGIC_VECTOR size for the three vectors: *A*, *B*, and *F*.

The *Unsigned_Overflow* bit is obtained by performing the addition or subtraction operation using $n + 1$ bits. The two operands are zero extended using the & symbol for concatenation before the operation is performed. The result of the operation is stored in the $n + 1$ bit vector, *result*. The most significant bit of this vector, *result(n)*, is the *Unsigned_Overflow* bit.

To get the *Signed_Overflow* bit, we need to XOR the *Unsigned_Overflow* bit with the carry bit, c_3, from the second-to-last bit slice. The c_3 bit is obtained just like how the *Unsigned_Overflow* bit is obtained, except that the operation is performed on only the first $n - 1$ bits of the two operands. The vector *c3* of length *n* is used for storing the result of the operation. The *Signed_Overflow* signal is the XOR of *result(n)* with *c3(n–1)*.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
ENTITY AddSub IS
GENERIC(n: INTEGER :=4);  -- default number of bits = 4
PORT(S: IN STD_LOGIC;  -- select subtract signal
  A: IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
  B: IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
  F: OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0);
  unsigned_overflow: OUT STD_LOGIC;
  signed_overflow: OUT STD_LOGIC);
END AddSub;

ARCHITECTURE Behavioral OF AddSub IS
  -- temporary result for extracting the unsigned overflow bit
  SIGNAL result: STD_LOGIC_VECTOR(n DOWNTO 0);
  -- temporary result for extracting the c3 bit
  SIGNAL c3:  STD_LOGIC_VECTOR(n-1 DOWNTO 0);
BEGIN
  PROCESS(S, A, B)
  BEGIN
    IF (S = '0') THEN     -- addition
      -- the two operands are zero extended one extra bit before adding
      -- the & is for string concatination
      result <= ('0' & A) + ('0' & B);
      c3  <= ('0' & A(n-2 DOWNTO 0)) + ('0' & B(n-2 DOWNTO 0));
      F <= result(n-1 DOWNTO 0);        -- extract the n-bit result
      unsigned_overflow <= result(n);    -- get the unsigned overflow bit
      signed_overflow <= result(n) XOR c3(n-1); -- get signed overflow bit
    ELSE      -- subtraction
      -- the two operands are zero extended one extra bit before subtracting
      -- the & is for string concatination
      result <= ('0' & A) - ('0' & B);
      c3  <= ('0' & A(n-2 DOWNTO 0)) - ('0' & B(n-2 DOWNTO 0));
      F <= result(n-1 DOWNTO 0);        -- extract the n-bit result
      unsigned_overflow <= result(n);    -- get the unsigned overflow bit
      signed_overflow <= result(n) XOR c3(n-1); -- get signed overflow bit
    END IF;
  END PROCESS;
END Behavioral;
```

Figure 4.9 Behavioral VHDL code for a 4-bit adder-subtractor combination component.

4.6   Arithmetic Logic Unit

The **arithmetic logic unit** (**ALU**) is one of the main components inside a microprocessor. It is responsible for performing arithmetic and logic operations, such as addition, subtraction, logical AND, and logical OR. The ALU, however, is not used to perform multiplications or divisions. It turns out that, in constructing the circuit for the ALU, we can use the same idea as for constructing the adder-subtractor combination circuit, as discussed in the previous section. Again, we will use the ripple-carry adder as the building block and then insert some combinational logic circuitry in front of the two input operands to each full adder. This way, the primary inputs will be modified accordingly, depending on the operations being performed before being passed to the full adder. The general, overall circuit for a 4-bit ALU is shown in Figure 4.10(a) and its logic symbol in Figure 4.10(b).

As we can see in the Figure 4.10(a), the two combinational circuits in front of the full adder (FA) are labeled LE and AE. The logic extender (LE) is for manipulating all logical operations; whereas, the arithmetic extender (AE) is for manipulating all arithmetic operations. The LE performs the actual logical operations on the two primary operands, a_i and b_i, before passing the result to the first operand, x_i, of the FA. On the other hand, the AE only modifies the second operand, b_i, and passes it to the second operand, y_i, of the FA where the actual arithmetic operation is performed.
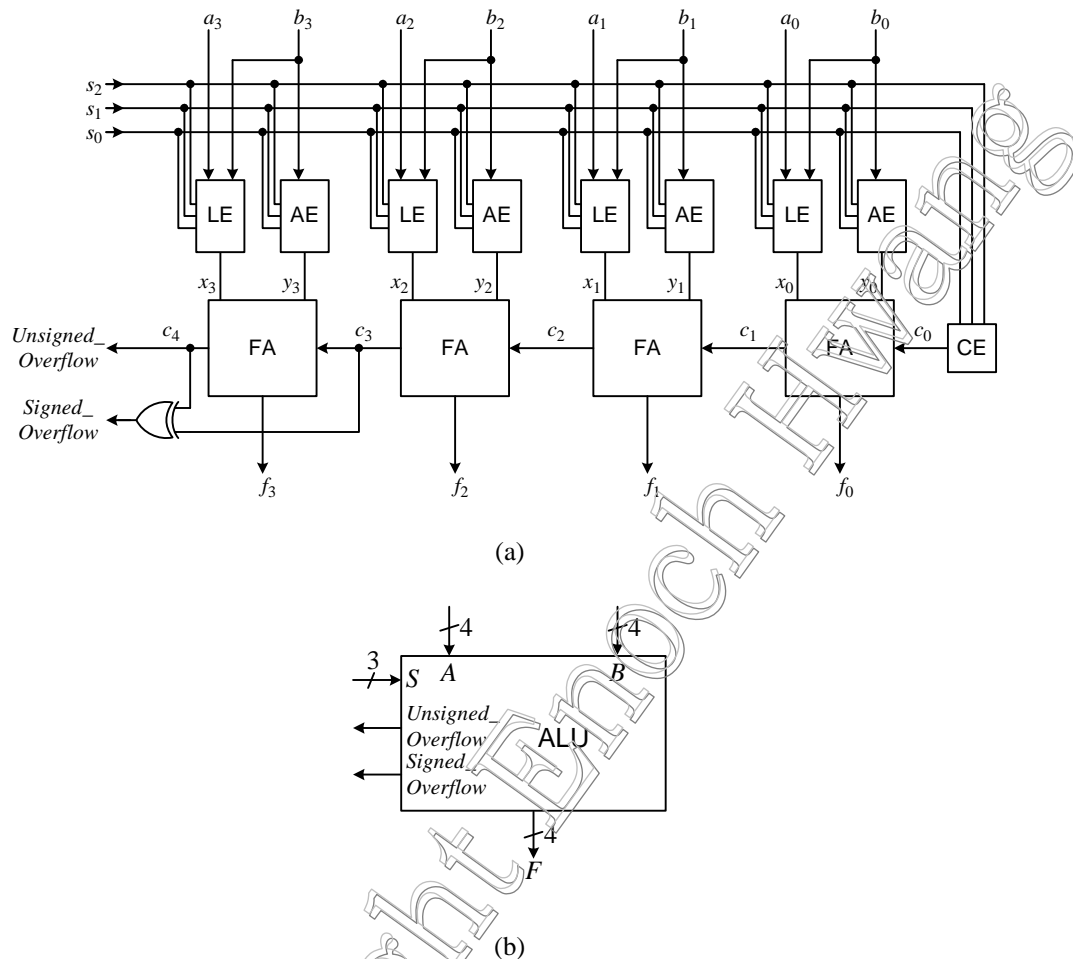

(a)

(b)

Figure 4.10 4-bit ALU: (a) circuit; (b) logic symbol.

We saw from the adder-subtractor circuit that, to perform additions and subtractions, we only need to modify y_i (the second operand to the FA) so that all operations can be done with additions. Thus, the AE only takes the second operand of the primary input, b_i, as its input and modifies the value depending on the operation being performed. Its output is y_i, and it is connected to the second operand input of the FA. As in the adder-subtractor circuit, the addition is performed in the FA. When arithmetic operations are being performed, the LE must pass the first operand unchanged from the primary input a_i to the output x_i for the FA.

Unlike the AE (where it only modifies the operand), the LE performs the actual logical operations. Thus, for example, if we want to perform the operation A OR B, the LE for each bit slice will take the corresponding bits, a_i and b_i, and OR them together. Hence, one bit from both operands, a_i and b_i, are inputs to the LE. The output of the LE is passed to the first operand, x_i, of the FA. Since this value is already the result of the logical operation, we do not want the FA to modify it but to simply pass it on to the primary output, f_i. This is accomplished by setting both the second operand, y_i, of the FA and c_0 to 0, since adding a 0 will not change the resulting value.

The combinational circuit labeled CE (for carry extender) is for modifying the primary carry-in signal, c_0, so that arithmetic operations are performed correctly. Logical operations do not use the carry signal, so c_0 is set to 0 for all logical operations.

s_2	s_1	s_0	Operation Name	Operation	x_i (LE)	y_i (AE)	c_0 (CE)
0	0	0	Pass	Pass A to output	a_i	0	0
0	0	1	AND	A AND B	a_i AND b_i	0	0
0	1	0	OR	A OR B	a_i OR b_i	0	0
0	1	1	NOT	A'	a_i'	0	0
1	0	0	Addition	$A + B$	a_i	b_i	0
1	0	1	Subtraction	$A - B$	a_i	b_i'	1
1	1	0	Increment	$A + 1$	a_i	0	1
1	1	1	Decrement	$A - 1$	a_i	1	0

(a)

s_2	s_1	s_0	x_i
0	0	0	a_i
0	0	1	$a_i b_i$
0	1	0	$a_i + b_i$
0	1	1	a_i'
1	×	×	a_i

(b)

s_2	s_1	s_0	b_i	y_i
0	×	×	×	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

(c)

s_2	s_1	s_0	c_0
0	×	×	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

(d)

Figure 4.11 ALU operations: (a) function table; (b) LE truth table; (c) AE truth table; (d) CE truth table.

In the circuit shown in Figure 4.10, three select lines, s_2, s_1, and s_0, are used to select the operations of the ALU. With these three select lines, the ALU circuit can implement up to eight different operations. Suppose that the operations that we want to implement in our ALU are as defined in Figure 4.11(a). The x_i column shows the values that the LE must generate for the different operations. The y_i column shows the values that the AE must generate. The c_0 column shows the carry signals that the CE must generate.

For example, for the pass-through operation, the value of a_i is passed through without any modifications to x_i. For the AND operation, x_i gets the result of a_i AND b_i. As mentioned before, both y_i and c_0 are set to 0 for all of the logical operations, because we do not want the FA to change the results. The FA is used only to pass the results from the LE straight through to the output, F. For the subtraction operation, instead of subtracting B, we want to add $-B$. Changing B to $-B$ in two's complement format requires flipping the bits of B and then adding a 1. Thus, y_i gets the inverse of b_i, and the 1 is added through the carry-in, c_0. To increment A, we set y_i to all 0's, and add the 1 through the carry-in, c_0. To decrement A, we add a -1 instead. Negative one in two's complement format is a bit string with all 1's. Hence, we set y_i to all 1's and the carry-in c_0 to 0. For all the arithmetic operations, we need the first operand, A, unchanged for the FA. Thus, x_i gets the value of a_i for all arithmetic operations.

Figure 4.11(b), (c) and (d) show the truth tables for the LE, AE, and CE, respectively. The LE circuit is derived from the x_i column of Figure 4.11(b); the AE circuit is derived from the y_i column of Figure 4.11(c); and the CE circuit is derived from the c_0 column of Figure 4.11(d). Notice that x_i is dependent on five variables, s_2, s_1, s_0, a_i, and b_i; whereas, y_i is dependent on only four variables, s_2, s_1, s_0, and b_i; and c_0 is dependent on only the three select lines, s_2, s_1, and s_0. The K-maps, equations, and schematics for these three circuits are shown in Figure 4.12.

The behavioral VHDL code for the ALU is shown in Figure 4.13, and a sample simulation trace for all the operations using the two inputs 5 and 3 is shown in Figure 4.14.

$$x_i = s_2a_i + s_0'a_i + s_1'a_ib_i + s_2's_1a_i'b_i + s_2's_1s_0a_i'$$
$$= s_2a_i + s_0'a_i + s_1'a_ib_i + s_2's_1a_i'(b_i + s_0)$$

(a)

$$y_i = s_2s_1s_0 + s_2s_0b_i' + s_2s_1's_0'b_i$$
$$= s_2s_0(s_1 + b_i') + s_2s_1's_0'b_i$$

(b)

$$c_0 = s_2s_1's_0 + s_2s_1s_0'$$
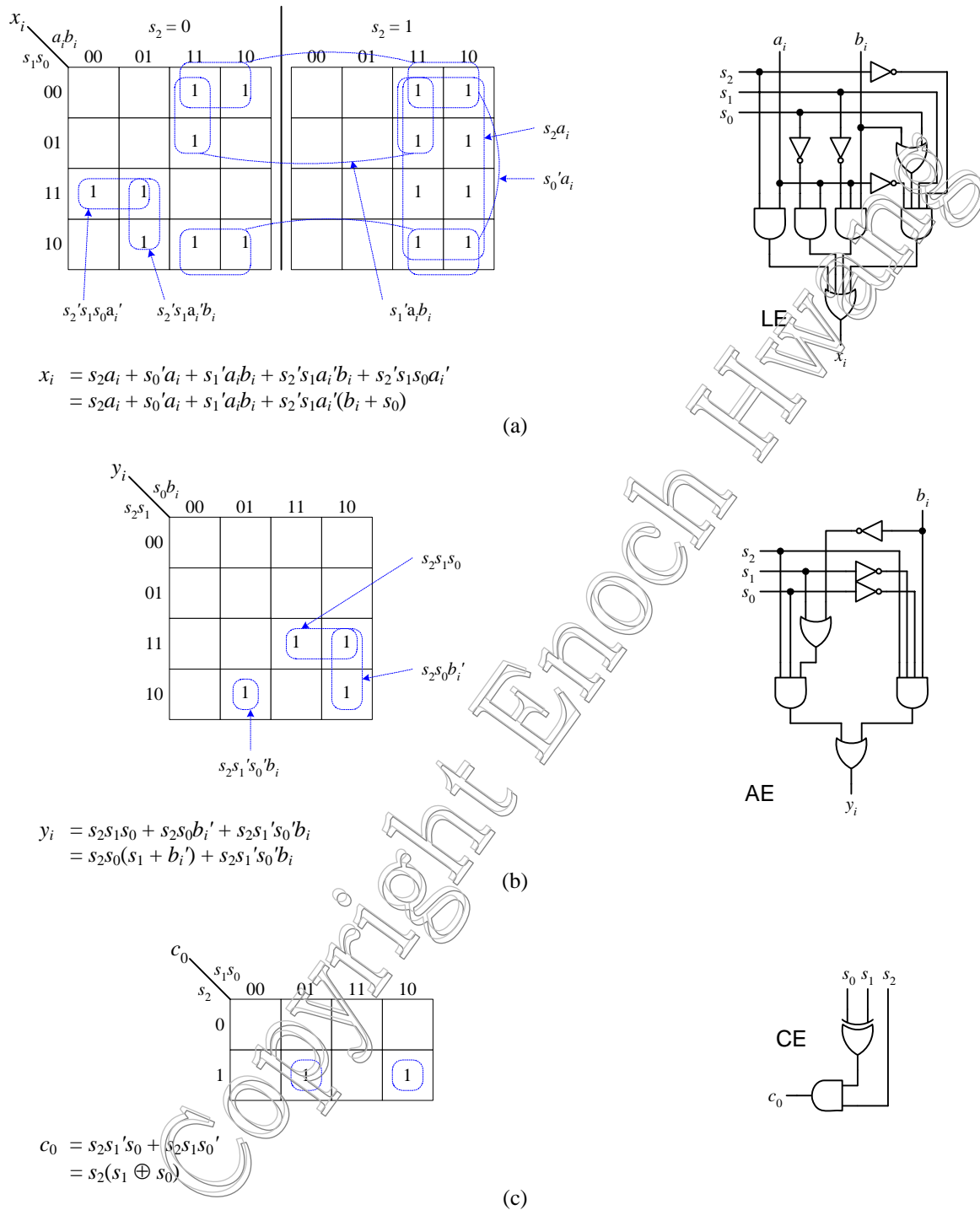$$= s_2(s_1 \oplus s_0)$$

(c)

Figure 4.12 K-maps, equations, and schematics for: (a) LE; (b) AE; and (c) CE.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
-- The following package is needed so that the STD_LOGIC_VECTOR signals
-- A and B can be used in unsigned arithmetic operations.
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY alu IS PORT (
  S: IN STD_LOGIC_VECTOR(2 DOWNTO 0);      -- select for operations
  A, B: IN STD_LOGIC_VECTOR(3 DOWNTO 0); -- input operands
  F: OUT STD_LOGIC_VECTOR(3 DOWNTO 0));  -- output
END alu;

ARCHITECTURE Behavior OF alu IS
BEGIN
  PROCESS(S, A, B)
  BEGIN
    CASE S IS
    WHEN "000" =>  -- pass A through
      F <= A;
    WHEN "001" =>  -- AND
      F <= A AND B;
    WHEN "010" =>  -- OR
      F <= A OR B;
    WHEN "011" =>  -- NOT A
      F <= NOT A;
    WHEN "100" =>  -- add
      F <= A + B;
    WHEN "101" =>  -- subtract
      F <= A - B;
    WHEN "110" =>  -- increment
      F <= A + 1;
    WHEN OTHERS => -- decrement
      F <= A - 1;
    END CASE;
  END PROCESS;
END Behavior;
```
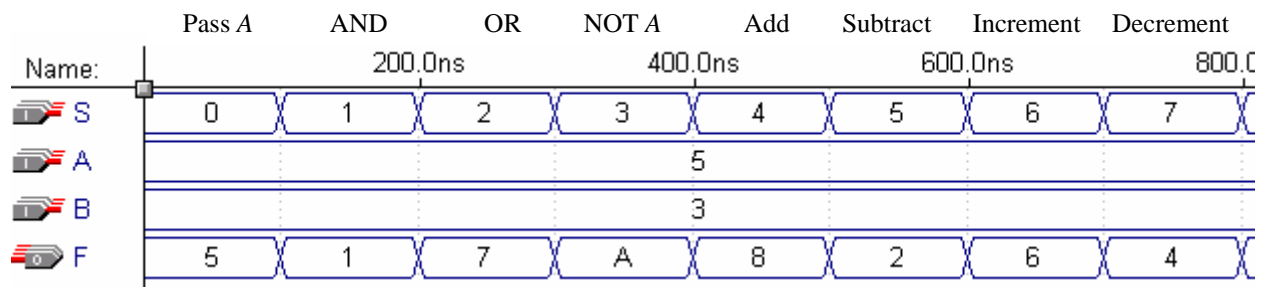
Figure 4.13 Behavioral VHDL code for an ALU.

Figure 4.14 Sample simulation trace with the two input operands, 5 and 3, for all of the eight operations.

4.7 Decoder

A **decoder**, also known as a **demultiplexer**, asserts one out of n output lines, depending on the value of an m-bit binary input data. In general, an m-to-n decoder has m input lines, A_{m-1}, …, A_0, and n output lines, Y_{n-1}, …, Y_0, where $n = 2^m$. In addition, it has an enable line, E, for enabling the decoder. When the decoder is disabled with E set to 0, all of the output lines are de-asserted. When the decoder is enabled, then the output line whose index is equal to the value of the input binary data is asserted. For example, for a 3-to-8 decoder, if the input address is 101, then the output line Y_5 is asserted (set to 1 for active-high), while the rest of the output lines are de-asserted (set to 0 for active-high).

A decoder is used in a system having multiple components, and we want only one component to be selected or enabled at any one time. For example, in a large memory system with multiple memory chips, only one memory chip is enabled at a time. One output line from the decoder is connected to the enable input on each memory chip. Thus, an address presented to the decoder will enable that corresponding memory chip. The truth table, circuit, and logic symbol for a 3-to-8 decoder are shown in Figure 4.15.

A larger size decoder can be implemented using several smaller decoders. For example, Figure 4.16 uses seven 1-to-2 decoders to implement a 3-to-8 decoder. The correct operation of this circuit is left as an exercise for the reader.

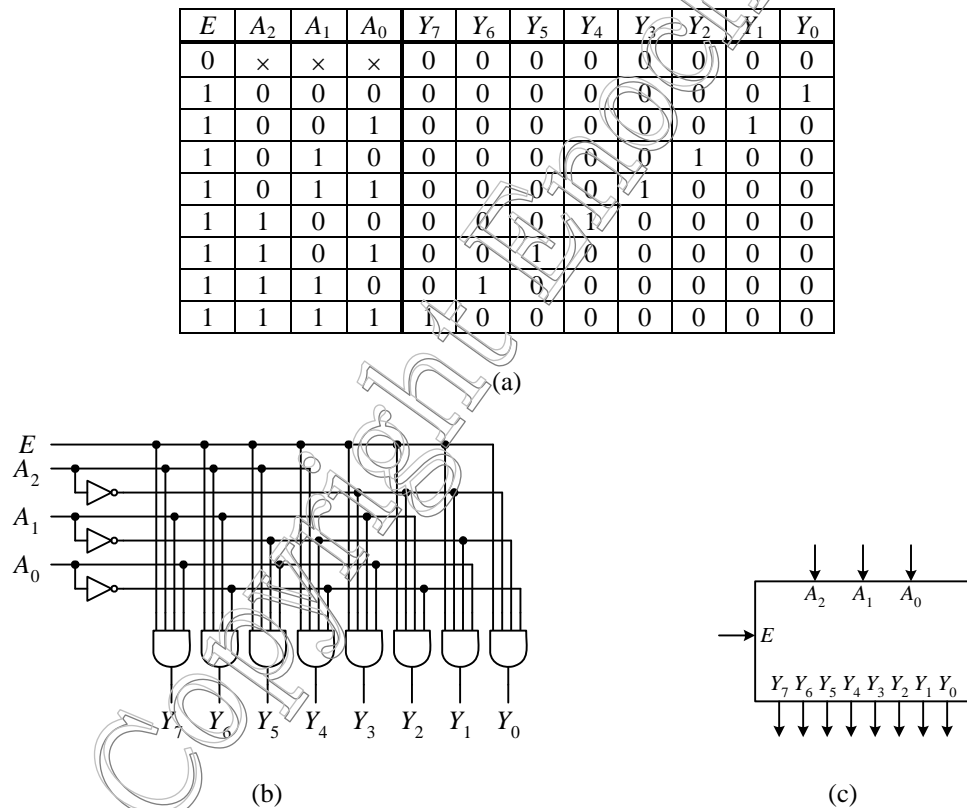The behavioral VHDL code for the 3-to-8 decoder is shown in Figure 4.17.

E	A_2	A_1	A_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	×	×	×	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

(a)

Figure 4.15 A 3-to-8 decoder: (a) truth table; (b) circuit; (c) logic symbol.
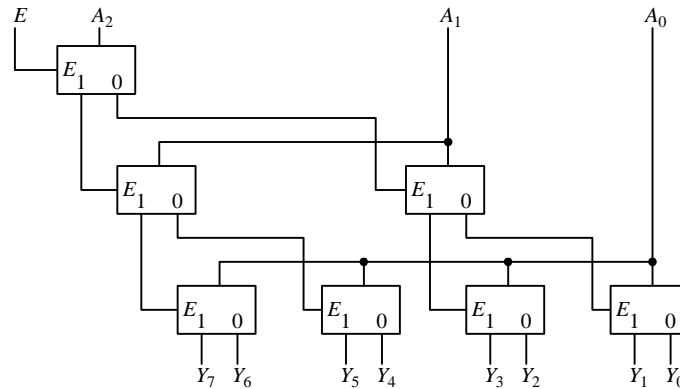
Figure 4.16 A 3-to-8 decoder implemented with seven 1-to-2 decoders

```
-- A 3-to-8 decoder
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY Decoder IS PORT(
  E: IN STD_LOGIC;                  -- enable
  A: IN STD_LOGIC_VECTOR(2 DOWNTO 0); -- 3 bit address
  Y: OUT STD_LOGIC_VECTOR(7 DOWNTO 0)); -- data bus output
END Decoder;

ARCHITECTURE Behavioral OF Decoder IS
BEGIN
  PROCESS (E, A)
  BEGIN
    IF (E = '0') THEN                 -- disabled
      Y <= (OTHERS => '0');           -- 8-bit vector of 0
    ELSE
      CASE A IS                       -- enabled
        WHEN "000" => Y <= "00000001";
        WHEN "001" => Y <= "00000010";
        WHEN "010" => Y <= "00000100";
        WHEN "011" => Y <= "00001000";
        WHEN "100" => Y <= "00010000";
        WHEN "101" => Y <= "00100000";
        WHEN "110" => Y <= "01000000";
        WHEN "111" => Y <= "10000000";
        WHEN OTHERS => NULL;
      END CASE;
    END IF;
  END PROCESS;
END Behavioral;
```

Figure 4.17 Behavioral VHDL code for a 3-to-8 decoder.

4.8 Encoder

An **encoder** is almost like the inverse of a decoder where it encodes a 2^n-bit input data into an n-bit code. The encoder has 2^n input lines and n output lines, as shown by the logic symbol in Figure 4.18(d) for $n = 3$. The operation of the encoder is such that exactly one of the input lines should have a 1 while the remaining input lines should have 0's. The output is the binary value of the index of the input line that has the 1. The truth table for an 8-to-3 encoder is shown in Figure 4.18(a). For example, when input I_3 is a 1, the three output bits Y_2, Y_1, and Y_0, are set to 011, which is the binary number for the index 3. Entries having multiple 1's in the truth table inputs are ignored, since we are assuming that only one input line can be a 1.

Looking at the three output columns in the truth table, we obtain the three equations shown in Figure 4.18(b) and the resulting circuit in Figure 4.18(c). The logic symbol is shown in Figure 4.18(d).

Encoders are used to reduce the number of bits needed to represent some given data either in data storage or in data transmission. Encoders are also used in a system with 2^n input devices, each of which may need to request for service. One input line is connected to one input device. The input device requesting for service will assert the input line that is connected to it. The corresponding n-bit output value will indicate to the system which of the 2^n devices is requesting for service. For example, if device 5 requests for service, it will assert the I_5 input line. The system will know that device 5 is requesting for service, since the output will be 101 = 5. However, this only works correctly if it is guaranteed that only one of the 2^n devices will request for service at any one time.

If two or more devices request for service at the same time, then the output will be incorrect. For example, if devices 1 and 4 of the 8-to-3 encoder request for service at the same time, then the output will also be 101, because I_4 will assert the Y_2 signal, and I_1 will assert the Y_0 signal. To resolve this problem, a priority is assigned to each of the input lines so that when multiple requests are made, the encoder outputs the index value of the input line with the highest priority. This modified encoder is known as a **priority encoder**.

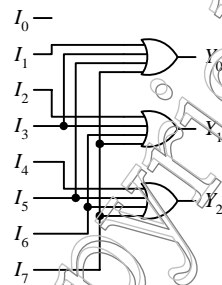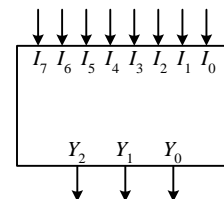| I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

(a)

$$Y_0 = I_1 + I_3 + I_5 + I_7$$
$$Y_1 = I_2 + I_3 + I_6 + I_7$$
$$Y_2 = I_4 + I_5 + I_6 + I_7$$

(b)

(c)

(d)

Figure 4.18 An 8-to-3 encoder: (a) truth table; (b) equations; (c) circuit; (d) logic symbol.

4.8.1 * Priority Encoder

The truth table for an active-high 8-to-3 priority encoder is shown in Figure 4.19. The table assumes that input I_7 has the highest priority, and I_0 has the lowest priority. For example, if the highest priority input asserted is I_3, then it doesn't matter whether the lower priority input lines, I_2, I_1 and I_0, are asserted or not; the output will be for that of I_3, which is 011. Since it is possible that no inputs are asserted, there is an extra output, Z, that is needed to differentiate between when no inputs are asserted and when one or more inputs are asserted. Z is set to a 1 when one or more inputs are asserted; otherwise, Z is set to 0. When Z is 0, all of the Y outputs are meaningless.

I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	Y_2	Y_1	Y_0	Z
0	0	0	0	0	0	0	0	×	×	×	0
0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	0	1	×	0	0	1	1
0	0	0	0	0	1	×	×	0	1	0	1
0	0	0	0	1	×	×	×	0	1	1	1
0	0	0	1	×	×	×	×	1	0	0	1
0	0	1	×	×	×	×	×	1	0	1	1
0	1	×	×	×	×	×	×	1	1	0	1
1	×	×	×	×	×	×	×	1	1	1	1

Figure 4.19 An 8-to-3 priority encoder truth table.

An easy way to derive the equations for the 8-to-3 priority encoder is to define a set of eight intermediate variables, v_0, …, v_7, such that v_k is a 1 if I_k is the highest priority 1 input. Thus, the equations for v_0 to v_7 are:

$$v_0 = I_7' \, I_6' \, I_5' \, I_4' \, I_3' \, I_2' \, I_1' \, I_0$$
$$v_1 = I_7' \, I_6' \, I_5' \, I_4' \, I_3' \, I_2' \, I_1$$
$$v_2 = I_7' \, I_6' \, I_5' \, I_4' \, I_3' \, I_2$$
$$v_3 = I_7' \, I_6' \, I_5' \, I_4' \, I_3$$
$$v_4 = I_7' \, I_6' \, I_5' \, I_4$$
$$v_5 = I_7' \, I_6' \, I_5$$
$$v_6 = I_7' \, I_6$$
$$v_7 = I_7$$

Using these eight intermediate variables, the final equations for the priority encoder are similar to the ones for the regular encoder, namely:

$$Y_0 = v_1 + v_3 + v_5 + v_7$$
$$Y_1 = v_2 + v_3 + v_6 + v_7$$
$$Y_2 = v_4 + v_5 + v_6 + v_7$$

Finally, the equation for Z is simply

$$Z = I_7 + I_6 + I_5 + I_4 + I_3 + I_2 + I_1 + I_0$$

4.9 Multiplexer

The **multiplexer**, or **MUX** for short, allows the selection of one input signal among n signals, where $n > 1$ and is a power of two. Select lines connected to the multiplexer determine which input signal is selected and passed to the output of the multiplexer. In general, an n-to-1 multiplexer has n data input lines, m select lines where $m = \log_2 n$ (i.e., $2^m = n$), and one output line. For a 2-to-1 multiplexer, there is one select line, s, to select between the two inputs, d_0 and d_1. When $s = 0$, the input line, d_0, is selected, and the data present on d_0 is passed to the output, y. When $s = 1$, the input line, d_1, is selected and the data on d_1 is passed to y. The truth table, equation, circuit, and logic symbol for a 2-to-1 multiplexer are shown in Figure 4.20.

s	d_1	d_0	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$$y = s'd_1'd_0 + s'd_1d_0 + sd_1d_0' + sd_1d_0$$
$$= s'd_0(d_1' + d_1) + sd_1(d_0' + d_0)$$
$$= s'd_0 + sd_1$$

(a)   (b)

(c)   (d)
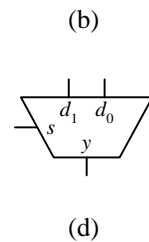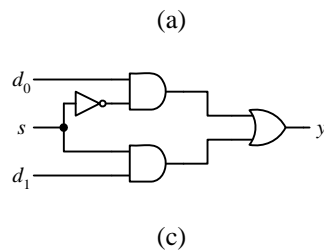
Figure 4.20 A 2-to-1 multiplexer: (a) truth table; (b) equation; (c) circuit; (d) logic symbol.

Constructing a larger-sized multiplexer, such as the 8-to-1 multiplexer, can be done similarly. In addition to having the eight data input lines, d_0 to d_7, the 8-to-1 multiplexer has three ($2^3 = 8$) select lines, s_0, s_1, and s_2. Depending on the value of the three select lines, one of the eight input lines will be selected and the data on that input line will be passed to the output. For example, if the value of the select lines is 101, then the input line d_5 is selected, and the data that is present on d_5 will be passed to the output.

The truth table, circuit, and logic symbol for the 8-to-1 multiplexer are shown in Figure 4.21. The truth table is written in a slightly different format. Instead of including the d's in the input columns and enumerating all $2^{11} = 2048$ rows (the eleven variables come from the eight d's and the three (s's), the d's are written in the entry under the output column. For example, when the select line value is 101, the entry under the output column is d_5, which means that y takes on the value of the input line d_5.

To understand the circuit in Figure 4.21(b), notice that each AND gate acts as a switch and is turned on by one combination of the three select lines. When a particular AND gate is turned on, the data at the corresponding d input is passed through that AND gate. The outputs of the remaining AND gates are all 0's.

s_2	s_1	s_0	y
0	0	0	d_0
0	0	1	d_1
0	1	0	d_2
0	1	1	d_3
1	0	0	d_4
1	0	1	d_5
1	1	0	d_6
1	1	1	d_7

(a)   (b)   (c)
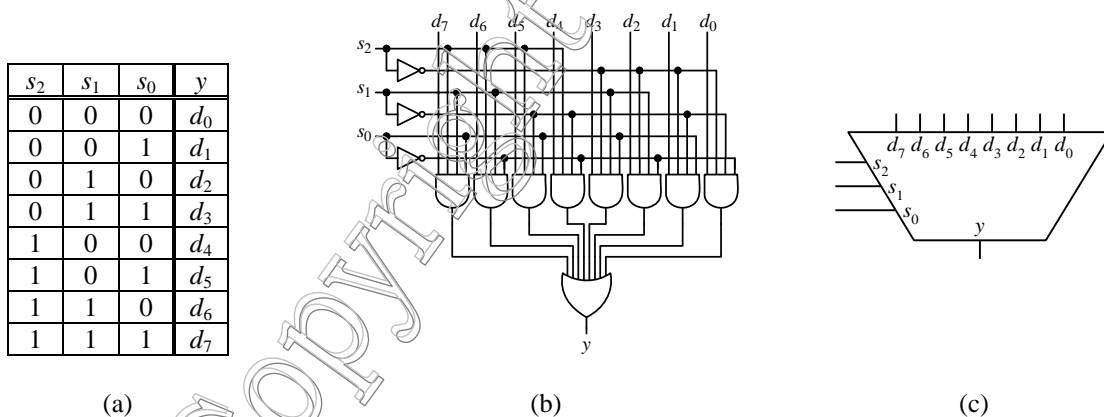
Figure 4.21 An 8-to-1 multiplexer: (a) truth table; (b) circuit; (c) logic symbol.

Instead of using 4-input AND gates (where three of its inputs are used by the three select lines to turn it on) we can use 2-input AND gates, as shown in Figure 4.22(a). This way the AND gate is turned on with just one line. The

eight 2-input AND gates can be turned on individually from the eight outputs of a 3-to-8 decoder. Recall from Section 4.7 that the decoder asserts only one output line at any time.

Larger multiplexers can also be constructed from smaller multiplexers. For example, an 8-to-1 multiplexer can be constructed using seven 2-to-1 multiplexers, as shown in Figure 4.22(b). The four top-level 2-to-1 multiplexers provide the eight data inputs and all are switched by the same least significant select line s_0. This top level selects one from each group of two data inputs. The middle level then groups the four outputs from the top level again into groups of two, and selects one from each group using the select line s_1. Finally, the multiplexer at the bottom level uses the most significant select line s_2 to select one of the two outputs from the middle level multiplexers.

The VHDL code for an 8-bit wide 4-to-1 multiplexer is shown in Figure 4.23. Two different implementations of the same multiplexer are shown. Figure 4.23(a) shows the architecture code written at the behavioral level, since it uses a PROCESS statement. Inside the PROCESS block, a CASE statement is used to select between the four choices for S. Figure 4.23(b) shows a dataflow level architecture code using a concurrent selected signal assignment statement using the keyword WITH … SELECT. In the first choice, if S is equal to 00, then the value $D0$ is assigned to Y. If S does not match any one of the four choices, 00, 01, 10, and 11, then the WHEN OTHERS clause is selected. The syntax (OTHERS => 'U') straight quotes means to fill the entire vector with the value "U".
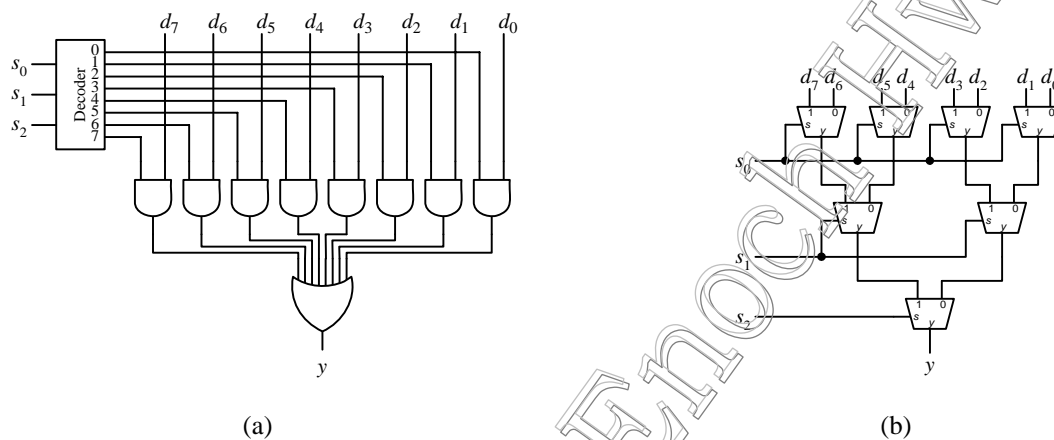

Figure 4.22 An 8-to-1 multiplexer implemented using: (a) a 3-to-8 decoder; (b) seven 2-to-1 multiplexers.

```
-- A 4-to-1 8-bit wide multiplexer
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY Multiplexer IS PORT (
  S: IN STD_LOGIC_VECTOR(1 DOWNTO 0);                      -- select lines
  D0, D1, D2, D3: IN STD_LOGIC_VECTOR(7 DOWNTO 0);        -- data bus input
  Y: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));                    -- data bus output
END Multiplexer;

-- Behavioral level code
ARCHITECTURE Behavioral OF Multiplexer IS
BEGIN
  PROCESS (S,D0,D1,D2,D3)
  BEGIN
    CASE S IS
      WHEN "00" => Y <= D0;
      WHEN "01" => Y <= D1;
      WHEN "10" => Y <= D2;
      WHEN "11" => Y <= D3;
      WHEN OTHERS => Y <= (OTHERS => 'U');                 -- 8-bit vector of U
```

```
     END CASE;
   END PROCESS;
END Behavioral;
```

(a)

```
-- Dataflow level code
ARCHITECTURE Dataflow OF Multiplexer IS
BEGIN
  WITH S SELECT Y <=
     D0 WHEN "00",
     D1 WHEN "01",
     D2 WHEN "10",
     D3 WHEN "11",
     (OTHERS => 'U') WHEN OTHERS;                    -- 8-bit vector of U
END Dataflow;
```

(b)

Figure 4.23 VHDL code for an 8-bit wide 4-to-1 multiplexer: (a) behavioral level; (b) dataflow level.

4.9.1 * Using Multiplexers to Implement a Function

Multiplexers can be used to implement a Boolean function very easily. In general, for an n-variable function, a 2^n-to-1 multiplexer (that is, a multiplexer with n select lines) is needed. An n-variable function has 2^n minterms, and each minterm corresponds to one of the 2^n multiplexer inputs. The n input variables are connected to the n select lines of the multiplexer. Depending on the values of the n variables, one data input line will be selected, and the value on that input line is passed to the output. Therefore, all we need to do is to connect all of the data input lines to either a 1 or a 0, depending on whether we want that corresponding minterm to be a 1-minterm or a 0-minterm, respectively.

Figure 4.24 shows the implementation of the 3-variable function, $F(x, y, z) = x'y'z' + x'yz' + xy'z + xyz' + xyz$. The 1-minterms for this function are m_0, m_2, m_5, m_6, and m_7, so the corresponding data input lines d_0, d_2, d_5, d_6, and d_7 are connected to a 1, while the remaining data input lines are connected to a 0. For example, the 0-minterm $x'yz$ has the value 011, which will select the d_3 input, so a 0 passes to the output. On the other hand, the 1-minterm $xy'z$ has the value 101, which will select the d_5 input, so a 1 passes to the output.
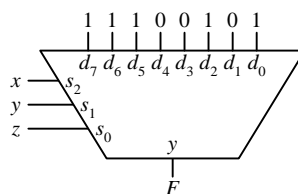

Figure 4.24 Using an 8-to-1 multiplexer to implement the function $F(x, y, z) = x'y'z' + x'yz' + xy'z + xyz' + xyz$.

4.10 Tri-state Buffer

A **tri-state** buffer, as the name suggests, has three states: 0, 1, and a third state denoted by Z. The value Z represents a high-impedance state, which for all practical purposes acts like a switch that is opened or a wire that is cut. Tri-state buffers are used to connect several devices to the same bus. A *bus* is one or more wire for transferring signals. If two or more devices are connected directly to a bus without using tri-state buffers, signals will get corrupted on the bus because the devices are always outputting either a 0 or a 1. However, with a tri-state buffer in between, devices that are not using the bus can disable the tri-state buffer so that it acts as if those devices are

physically disconnected from the bus. At any one time, only one active device will have its tri-state buffers enabled, and thus, use the bus.

The truth table and symbol for the tri-state buffer is shown in Figure 4.25(a) and (b). The active-high enable line E turns the buffer on or off. When E is de-asserted with a 0, the tri-state buffer is disabled, and the output y is in its high-impedance Z state. When E is asserted with a 1, the buffer is enabled, and the output y follows the input d.

A circuit consisting of only logic gates cannot produce the high-impedance state required by the tri-state buffer, since logic gates can only output a 0 or a 1. To provide the high impedance state, the tri-state buffer circuit uses two transistors in conjunction with logic gates, as shown in Figure 4.25(c). Section 5.3 will discuss the operations of these two transistors in detail. For now, we will keep it simple. The top PMOS transistor is enabled with a 0 at the node labeled A, and when it is enabled, a 1 signal from Vcc passes down through the transistor to y. The bottom NMOS transistor is enabled with a 1 at the node labeled B, and when it is enabled, a 0 signal from ground passes up through the transistor to y. When the two transistors are disabled (with $A = 1$ and $B = 0$) they will both output a high impedance Z value; so y will have a Z value.

Having the two transistors, we need a circuit that will control these two transistors so that together they realize the tri-state buffer function. The truth table for this control circuit is shown in Figure 4.25(d). The truth table is derived as follows. When $E = 0$ (it does not matter what the input d is) we want both transistors to be disabled so that the output y has the Z value. The PMOS transistor is disabled when the input $A = 1$; whereas, the NMOS transistor is disabled when the input $B = 0$. When $E = 1$ and $d = 0$, we want the output y to be a 0. To get a 0 on y, we need to enable the bottom NMOS transistor and disable the top PMOS transistor so that a 0 will pass through the NMOS transistor to y. To get a 1 on y for when $E = 1$ and $d = 1$, we need to do the reverse by enabling the top PMOS transistor and disabling the bottom NMOS transistor.

The resulting circuit is shown in Figure 4.25(c). When $E = 0$, the output of the NAND gate is a 1 regardless of what the other input is, and so the top PMOS transistor is turned off. Similarly, the output of the AND gate is a 0, and so the bottom NMOS transistor is also turned off. Thus, when $E = 0$, both transistors are off, so the output y is in the Z state.

When $E = 1$, the outputs of both the NAND and AND gates are equal to d'. So if $d = 0$, the output of the two gates are both 1, so the bottom transistor is turned on while the top transistor is turned off. Thus, y will have the value 0, which is equal to d. On the other hand, if $d = 1$, the top transistor is turned on while the bottom transistor is turned off, and y will have the value 1.

The behavioral VHDL code for an 8-bit wide tri-state buffer is shown in Figure 4.26.

E	y
0	Z
1	d

E	d	A	B	y
0	0	1	0	Z
0	1	1	0	Z
1	0	1	1	0
1	1	0	0	1

(a)   (b)   (c)   (d)
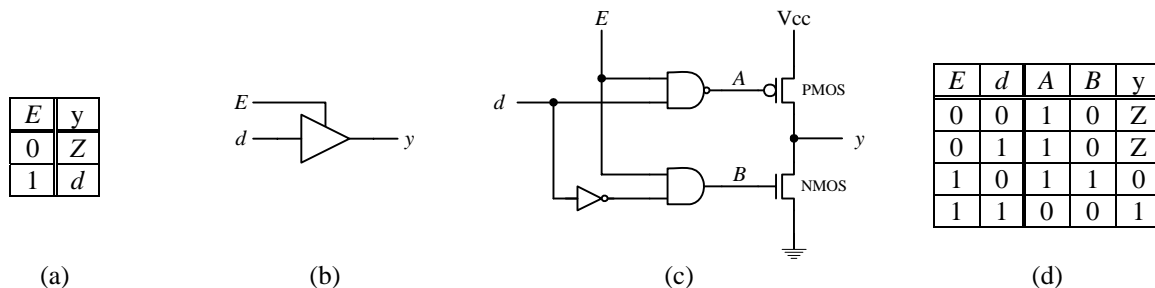
Figure 4.25 Tri-state buffer: (a) truth table; (b) logic symbol; (c) circuit; (d) truth table for the control portion of the tri-state buffer circuit.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;


ENTITY TriState_Buffer IS PORT (
  E: IN STD_LOGIC;
  d: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
  y: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END TriState_Buffer;
```

```
ARCHITECTURE Behavioral OF TriState_Buffer IS
BEGIN
  PROCESS (E, d)
  BEGIN
    IF (E = '1') THEN
      y <= d;
    ELSE
      y <= (OTHERS => 'Z');   -- to get 8 Z values
    END IF;
  END PROCESS;
END Behavioral;
```

Figure 4.26 VHDL code for an 8-bit wide tri-state buffer.

4.11 Comparator

Quite often, we need to compare two values for their arithmetic relationship (equal, greater, less than, etc.). A **comparator** is a circuit that compares two binary values and indicates whether the relationship is true or not. To compare whether a value is equal or not equal to a constant value, a simple AND gate can be used. For example, to compare a 4-bit variable x with the constant 3, the circuit in Figure 4.27(a) can be used. The AND gate outputs a 1 when the input is equal to the value 3. Since 3 is 0011 in binary, therefore, x_3 and x_2 must be inverted.

The XOR and XNOR gates can be used for comparing inequality and equality, respectively, between two values. The XOR gate outputs a 1 when its two input values are different. Hence, we can use one XOR gate for comparing each bit pair of the two operands. A 4-bit inequality comparator is shown in Figure 4.27(b). Four XOR gates are used, with each one comparing the same bit from the two operands. The outputs of the XOR gates are ORed together so that if any bit pair is different then the two operands are different, and the resulting output is a 1. Similarly, an equality comparator can be constructed using XNOR gates instead, since the XNOR gate outputs a 1 when its two input values are the same.

To compare the greater-than or less-than relationships, we can construct a truth table and build the circuit from it. For example, to compare whether a 4-bit value X is less than five, we get the truth table, equation, and circuit shown in Figure 4.27(c).

(a)

(b)

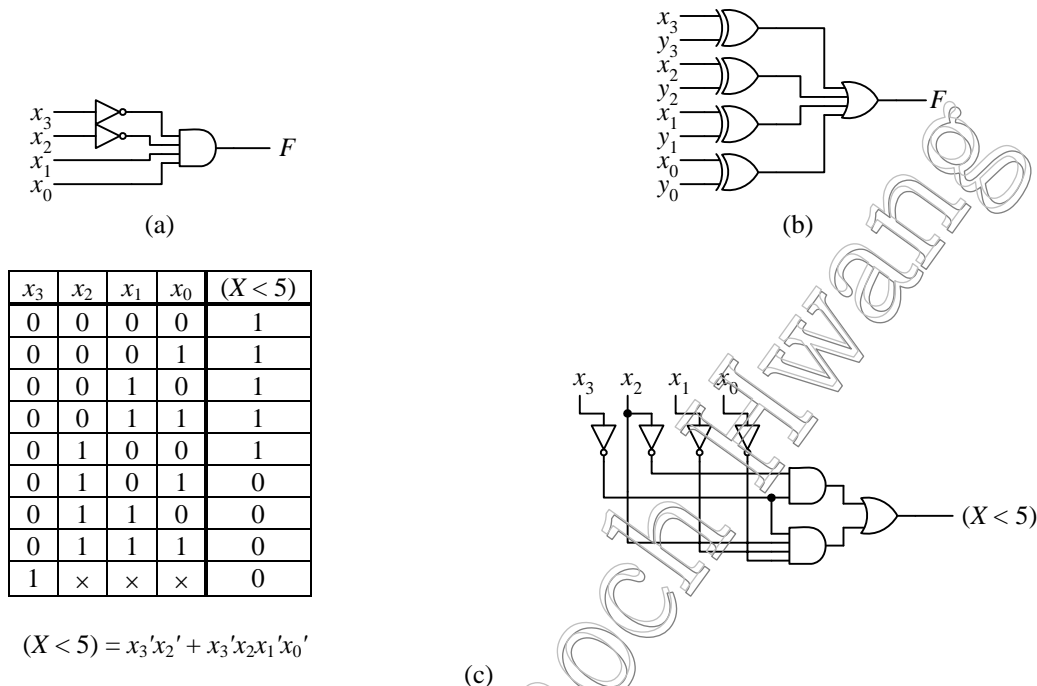| x_3	x_2	x_1	x_0	$(X < 5)$
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	×	×	×	0

$$(X < 5) = x_3'x_2' + x_3'x_2x_1'x_0'$$

(c)

Figure 4.27 Simple 4-bit comparators for: (a) $X = 3$; (b) $X \neq Y$; (c) $X < 5$.

Instead of constructing a comparator for a fixed number of bits for the input values, we often prefer to construct an **iterative circuit** by constructing a 1-bit slice comparator and then daisy chaining n of them together to make an n-bit comparator. The 1-bit slice comparator will have (in addition to the two input operand bits, x_i and y_i) a p_i bit that keeps track of whether all the previous bit pairs compared so far are true or not for that particular relationship. The circuit outputs a 1 if $p_i = 1$, and the relationship is true for the current bit pair, x_i and y_i. Figure 4.28(a) shows a 1-bit slice comparator for the equal relationship. If the current bit pair, x_i and y_i, is equal, the XNOR gate will output a 1. Hence, $p_{i+1} = 1$ if the current bit pair is equal and the previous bit pair, p_i, is a 1. To obtain a 4-bit iterative equality comparator, we connect four 1-bit equality comparators in series, as shown in Figure 4.28(b). The initial p_0 bit must be set to a 1. Thus, if all four bit pairs are equal, then the last bit, p_4, will be a 1; otherwise, p_4 will be a 0.
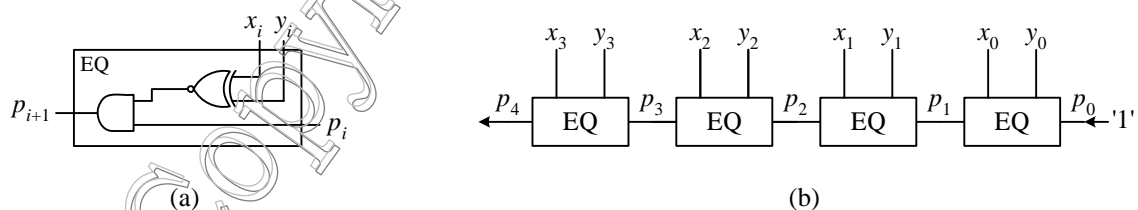

(a)

(b)

Figure 4.28 Iterative comparators: (a) 1-bit slice for $x_i = y_i$; (b) 4-bit $X = Y$.

Building an iterative comparator circuit for the greater-than relationship is slightly more difficult. The 1-bit slice comparator circuit for the condition $x_i > y_i$ is constructed as follows. In addition to the two operand input bits, x_i and y_i, there are also two status input bits, g_{in} and e_{in}. Here, g_{in} is a 1 if the condition $x_i > y_i$ is true for the previous bit slice; otherwise, g_{in} is a 0. Furthermore, e_{in} is a 1 if the condition $x_i = y_i$ is true; otherwise, e_{in} is a 0. The circuit also has two status output bits, g_{out} and e_{out}, having the same meaning as the g_{in} and e_{in} signals. These two input and two output status bits allow the bit slices to be daisy-chained together. Following the above description of the 1-bit slice, we obtain the truth table shown in Figure 4.29(a). The equations for e_{out} and g_{out} are shown in Figure 4.29(b), and the 1-bit slice circuit in Figure 4.29(c).

In order for the bit slices to operate correctly, we need to perform the comparisons from the most significant bit to the least significant bit. The complete 4-bit iterative comparator circuit for the condition $x > y$ is shown in Figure 4.29(d). The initial values for g_{in} and e_{in} must be set to $g_{in} = 0$ and $e_{in} = 1$.

If $x = y$, then the last e_{out} is a 1; otherwise, e_{out} is a 0. If the last e_{out} is a 0, then the last g_{out} can be either a 1 or a 0. If $x > y$ then g_{out} is a 1; otherwise, g_{out} is a 0. Notice that both e_{out} and g_{out} cannot be both 1's. The operation of this comparator circuit is summarized in Figure 4.29(e).

g_{in}	e_{in}	x_i	y_i	Meaning	g_{out}	e_{out}
0	0	×	×	<	0	0
0	1	0	0	=	0	1
0	1	0	1	<	0	0
0	1	1	0	>	1	0
0	1	1	1	=	0	1
1	0	×	×	>	1	0
1	1	×	×	Invalid	1	1

(a)

$g_{out} = g_{in} + e_{in}x_i y_i'$

$e_{out} = g_{in}e_{in} + e_{in}x_i'y_i' + e_{in}x_i y_i$

(b)

(c)

(d)

Condition	e_{out}	g_{out}
Invalid	1	1
$x = y$	1	0
$x > y$	0	1
$x < y$	0	0

(e)

Figure 4.29 Comparator for $x > y$: (a) truth table for a 1-bit slice; (b) K-maps and equations for g_{out} and e_{out}; (c) circuit for 1-bit slice; (d) 4-bit $x > y$ comparator circuit; (e) operational table.

4.12 Shifter

The **shifter** is used for shifting bits in a binary word one position either to the left or to the right. The operations for the shifter are referred to either as **shifting** or **rotating**, depending on how the end bits are shifted in or out. For a shift operation, the two end bits do not wrap around; whereas for a rotate operation, the two end bits wrap around. Figure 4.30 shows six different shift and rotate operations.

For example, for the "Shift left with 0" operation, all of the bits are shifted one position to the left. The original leftmost bit is shifted out (i.e., discarded) and the rightmost bit is filled with a 0. For the "Rotate left" operation, all of the bits are shifted one position to the left. However, instead of discarding the leftmost bit, it is shifted in as the rightmost bit (i.e., it rotates around).

For each bit position, a multiplexer is used to move a bit from either the left or right to the current bit position. The size of the multiplexer will determine the number of operations that can be implemented. For example, we can use a 4-to-1 multiplexer to implement the four operations, as specified by the table in Figure 4.31(a). Two select lines, s_1 and s_0, are needed to select between the four different operations. For a 4-bit operand, we will need to use four 4-to-1 multiplexers, as shown in Figure 4.31(b). How the inputs to the multiplexers are connected will depend on the given operations.

Operation	Comment	Example
Shift left with 0	Shift bits to the left one position. The leftmost bit is discarded and the rightmost bit is filled with a 0.	10110100 X01101000←
Shift left with 1	Same as above, except that the rightmost bit is filled with a 1.	10110100 X01101001←
Shift right with 0	Shift bits to the right one position. The rightmost bit is discarded and the leftmost bit is filled with a 0.	10110100 →01011010X
Shift right with 1	Same as above, except that the leftmost bit is filled with a 1.	10110100 →11011010X
Rotate left	Shift bits to the left one position. The leftmost bit is moved to the rightmost bit position.	10110100 01101001
Rotate right	Shift bits to the right one position. The rightmost bit is moved to the leftmost bit position.	10110100 01011010

Figure 4.30 Shifter and rotator operations.

s_1	s_0	Operation
0	0	Pass through
0	1	Shift left and fill with 0
1	0	Shift right and fill with 0
1	1	Rotate right

(a)

(b)   (c)
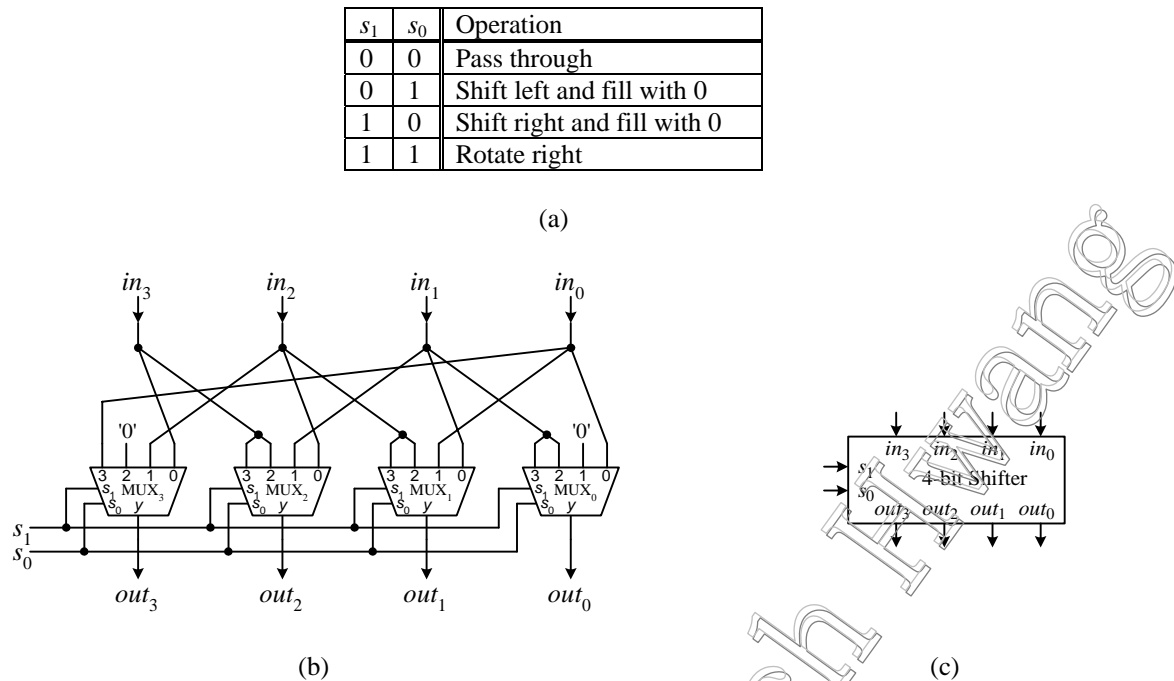
Figure 4.31 A 4-bit shifter: (a) operation table; (b) circuit; (c) logic symbol.

In this example, when $s_1 = s_0 = 0$, we want to pass the bit straight through without shifting (i.e., we want the value from in_i to pass to out_i). Given $s_1 = s_0 = 0$, d_0 of the multiplexer is selected, hence, in_i is connected to d_0 of MUX$_i$, which outputs to out_i. For $s_1 = 0$ and $s_0 = 1$, we want to shift left (i.e., we want the value from in_i to pass to out_{i+1}). With $s_1 = 0$ and $s_0 = 1$, d_1 of the multiplexer is selected, hence, in_i is connected to d_1 of MUX$_{i+1}$, which outputs to out_{i+1}. For this selection, we also want to shift in a 0 bit, so d_1 of MUX$_0$ is connected directly to a 0.

The behavioral VHDL code for an 8-bit shifter having the functions as defined in Figure 4.31(a) is shown in Figure 4.32.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY shifter IS PORT (
  S: IN STD_LOGIC_VECTOR(1 DOWNTO 0);      -- select for operations
  input: IN STD_LOGIC_VECTOR(7 DOWNTO 0);       -- input
  output: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));   -- output
END shifter;

ARCHITECTURE Behavior OF shifter IS
BEGIN
  PROCESS(S, input)
  BEGIN
    CASE S IS
    WHEN "00" =>   -- pass through
      output <= input;
    WHEN "01" =>   -- shift left with 0
      output <= input(6 DOWNTO 0) & '0';
    WHEN "10" =>   -- shift right with 0
      output <= '0' & input(7 DOWNTO 1);
    WHEN OTHERS =>    -- rotate right
      output <= input(0) & input(7 DOWNTO 1);
```

```
      END CASE;
   END PROCESS;
END Behavior;
```

Figure 4.32 Behavioral VHDL code for an 8-bit shifter having the operations as defined in Figure 4.31(a).
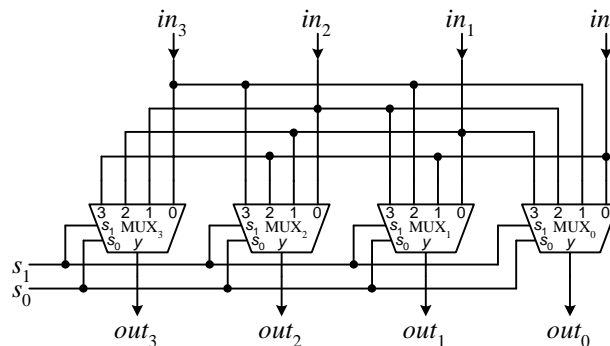
4.12.1 * Barrel Shifter

A **barrel shifter** is a shifter that can shift or rotate the data by any number of bits in a single operation. The select lines for a barrel shifter are used, not to determine what kind of operations (shift or rotate) to perform as for the general shifter, but rather, to determine how many bits to move. Hence, only one particular operation can be implemented in a barrel shifter circuit. In general, an n-bit barrel shifter can shift the data bits by as much as $n - 1$ bit distance away in one operation.

Figure 4.33(a) shows the operation table of a 4-bit barrel shifter implementing the rotate left operation. When $s_1 s_0 = 00$, no rotation is performed (i.e., a pass through). When $s_1 s_0 = 01$, the data bits are rotated one position to the left. When $s_1 s_0 = 10$, the data bits are rotated two positions to the left. The corresponding circuit is shown in Figure 4.33(b).

Select $s_1\ s_0$	Operation	Output $out_3\ out_2\ out_1\ out_0$
00	No rotation	$in_3\ in_2\ in_1\ in_0$
01	Rotate left by 1 bit position	$in_2\ in_1\ in_0\ in_3$
10	Rotate left by 2 bit positions	$in_1\ in_0\ in_3\ in_2$
11	Rotate left by 3 bit positions	$in_0\ in_3\ in_2\ in_1$

(a)

(b)

Figure 4.33 A 4-bit barrel shifter for the rotate left operation: (a) operation table; (b) circuit.

*4.13 * Multiplier*

In grade school, we were taught to multiply two numbers using a shift-and-add procedure. Regardless of whether the two numbers are in decimal or binary, we use the same shift-and-add procedure for multiplying them. In fact, multiplying with binary numbers is even easier, because you are always multiplying with either a 0 or a 1. Figure 4.34(a) shows the multiplication of two 4-bit unsigned binary numbers—the multiplicand M ($m_3 m_2 m_1 m_0$) with the multiplier Q ($q_3 q_2 q_1 q_0$)—to produce the resulting product P ($p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0$). Notice that the intermediate products are always either the same as the multiplicand (if the multiplier bit is a 1) or it is zero (if the multiplier bit is a 0).

We can derive a combinational multiplication circuit based on this shift-and-add procedure, as shown in Figure 4.34(b). Each intermediate product is obtained by ANDing the multiplicand M with one bit of the multiplier q_i. Since q_i is always a 1 or a 0, the output of the AND gates is always either m_i or 0. For example, bit zero of the first intermediate product is obtained by ANDing m_0 with q_0; bit one is obtained by ANDing m_1 with q_0; and so on. Hence, the four bits for the first intermediate product are m_3q_0, m_2q_0, m_1q_0, and m_0q_0; the four bits for the second intermediate product are m_3q_1, m_2q_1, m_1q_1, and m_0q_1; and so on.

Multiple adders are used to sum all of the intermediate products together to give the final product. Each intermediate product is shifted over to the correct bit position for the addition. For example, p_0 is just m_0q_0; p_1 is the sum of m_1q_0 and m_0q_1; p_2 is the sum of m_2q_0, m_1q_1 and m_0q_2; and so on. The four full adders (1-bit adders) in each row are connected, as in the ripple-carry adder with each carry-out signal connected to the carry-in of the next full adder. The carry-out of the last full adder is connected to the input of the last full adder in the row below. The last carry-out from the last row of adders is the value for p_7 of the final product. As in the ripple-carry adder, all of the initial carry-ins, c_0, are set to a 0.
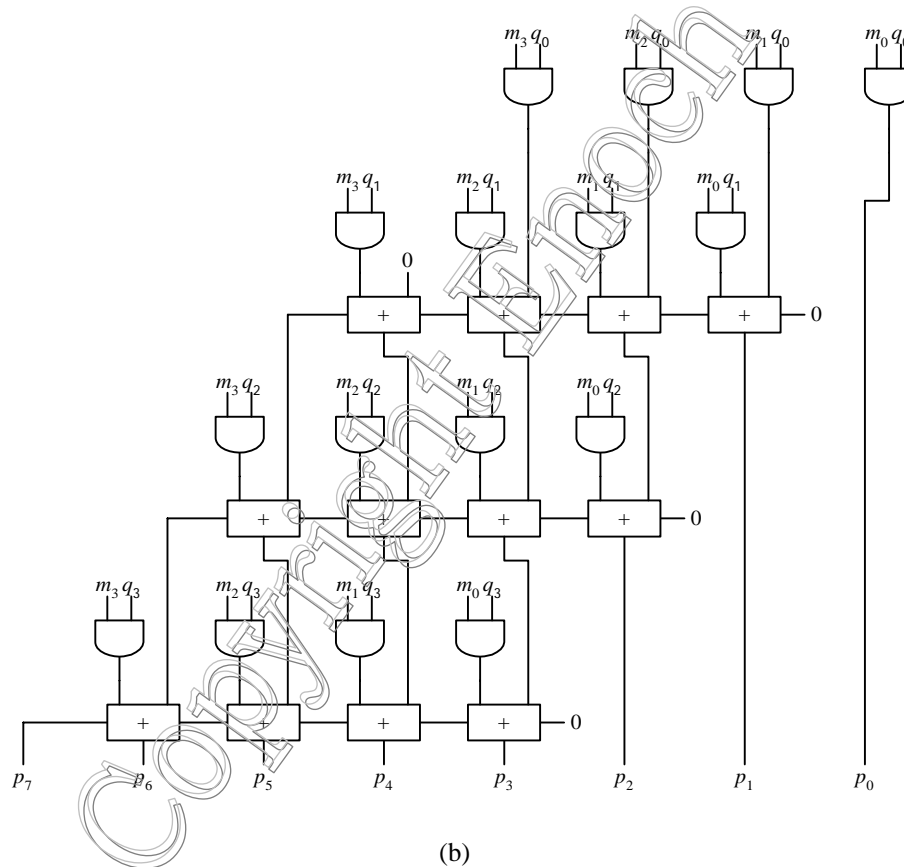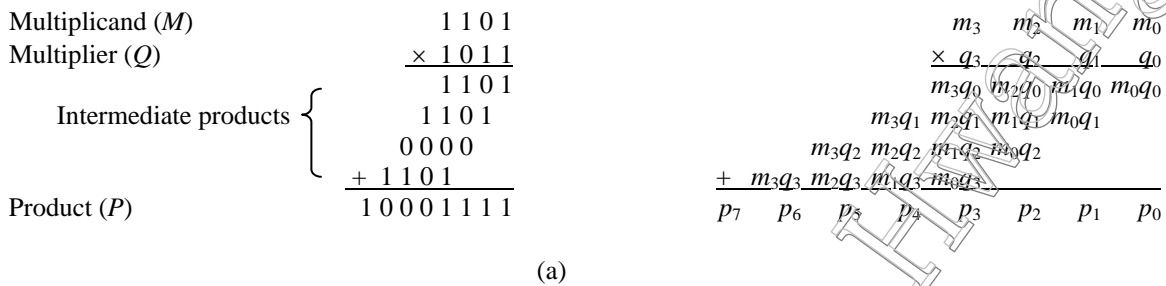

(a)

(b)

Figure 4.34 Multiplication: (a) method; (b) circuit.

4.14 Summary Checklist

- ❑ Full adder
- ❑ Ripple-carry adder
- ❑ Carry-lookahead adder
- ❑ Two's complement
- ❑ Sign extension
- ❑ Subtractor
- ❑ Arithmetic logic unit (ALU)
- ❑ Arithmetic extender (AE)
- ❑ Logic extender (LE)
- ❑ Carry extender (CE)
- ❑ Decoder
- ❑ Encoder
- ❑ Priority encoder
- ❑ Multiplexer (MUX)
- ❑ Tri-state buffer
- ❑ Z value
- ❑ Comparator
- ❑ Shifter
- ❑ Barrel Shifter
- ❑ Multiplier

4.15 Problems

4.1. Convert the following numbers to 12-bit binary numbers using two's complement representation.
   a) 234_{10}
   b) -234_{10}
   c) 234_8
   d) $BC4_{16}$
   e) -472_{10}

4.2. Convert the following two's complement binary numbers to decimal, octal, and hexadecimal formats.
   a) 1001011
   b) 011110
   c) 101101
   d) 1101011001
   e) 0110101100

4.3. Write the complete structural VHDL code for the full adder circuit shown in Figure 4.1(c).

4.4. Draw the smallest possible complete circuit for a 2-bit carry-lookahead adder.

4.5. Draw the complete circuit for a 4-bit carry-lookahead adder.

4.6. Derive the carry-lookahead equation and circuit for c_5.

4.7. Show that when adding two n-bit signed numbers, $A_{n-1}...A_0$ and $B_{n-1}...B_0$, producing the result, $S_{n-1}...S_0$, the *Signed_Overflow* flag can be deduced by the equation:

$$Signed\_Overflow = A_{n-1} \text{ XOR } B_{n-1} \text{ XOR } S_{n-1} \text{ XOR } S_n$$

4.8. Draw the complete 4-bit ALU circuit having the following operations. Use K-maps to reduce all of the equations to standard form.

s_2	s_1	s_0	Operations
0	0	0	$B - 1$
0	0	1	A NOR B
0	1	0	$A - B$
0	1	1	A XNOR B
1	0	0	1
1	0	1	A NAND B
1	1	0	$A + B$
1	1	1	A'

4.9. Draw the complete 4-bit ALU circuit having the following operations. Don't-care values are assigned to unused select combinations. Use K-maps to reduce all of the equations to standard form.

s_2	s_1	s_0	Operations
0	0	0	Pass A through the LE
0	0	1	Pass B through the LE
0	1	0	NOT A
0	1	1	NOT B
1	0	0	$A - B$
1	0	1	$B - A$
1	1	0	$B + 1$

4.10. Draw the complete 4-bit ALU circuit having the following operations. Use K-maps to reduce all of the equations to standard form.

s_2	s_1	s_0	Operations
0	0	0	A plus B
0	0	1	Increment A
0	1	0	Increment B
0	1	1	Pass A
1	0	0	$A - B$
1	0	1	A XOR B
1	1	0	A AND B

4.11. Draw the complete 4-bit ALU circuit having the following operations. Use K-maps to reduce all of the equations to standard form.

s_2	s_1	s_0	Operations
0	0	0	Pass A
0	0	1	Pass B through the AE
0	1	0	A plus B
0	1	1	A'
1	0	0	A XOR B
1	0	1	A NAND B
1	1	0	$A - 1$
1	1	1	$A - B$

4.12. Given the following K-maps for the LE, AE, and CE of an ALU, determine the ALU operations assigned to each of the select line combinations.

4.13. A four-function ALU has the following equations for its LE, AE, and CE:

$$x_i = a_i + s_1's_0b_i$$

$$y_i = s_1's_0' + s_1s_0b_i'$$

$$c_0 = s_1s_0$$

Determine the four functions in the correct order that are implemented in this ALU. Show all of your work.

4.14. Draw the circuit for the 2-to-4 decoder.

4.15. Derive the truth table for a 3-to-8 decoder using negative logic.

4.16. Draw the circuit for the 4-to-16 decoder using only 2-to-4 decoders.

4.17. Draw the circuit for the 4-to-2 priority encoder using only 2-input AND, 2-input OR, and NOT gates.

4.18. Draw the circuit for an 8-to-3 priority encoder.

4.19. Draw the circuit for the 4-to-2 priority encoder using only 2-to-1 priority encoders and 2-to-1 multiplexers.

4.20. Write the behavioral VHDL code for the 8-to-3 priority encoder.

4.21. Draw the circuit for a 16-to-1 multiplexer using only 4-to-1 multiplexers.

4.22. Draw the circuit for a 16-to-1 multiplexer using only 2-to-1 multiplexers.

4.23. Use only 2-to-1 multiplexers to implement the function: $f(w,x,y,z) = \Sigma(0,2,5,7,13,15)$.

4.24. Use only 2-to-1 multiplexers (as many as you need) to implement the function: $F(x, y, z) = \Pi(0, 3, 4, 5, 7)$.

4.25. Use one 8-to-1 multiplexer to implement the function: $F(x,y,z) = \Sigma(0,3,4,6,7)$.

4.26. Use 2-to-1 multiplexers to implement the function: $F(x,y,z) = \Sigma(0,2,4,5)$.

4.27. Derive the truth table for comparing two unsigned 2-bit operands for the less-than-or-equal-to relationship. Derive the equation and circuit from this truth table.

4.28. Construct the circuit for one bit slice of an n-bit magnitude comparator that compares $x_i \geq y_i$.

4.29. Draw the circuit for a 4-bit iterative comparator that tests for the greater-than-or-equal-to relationship.

4.30. Draw the circuit for a 4-bit shifter that realizes the following operation table:

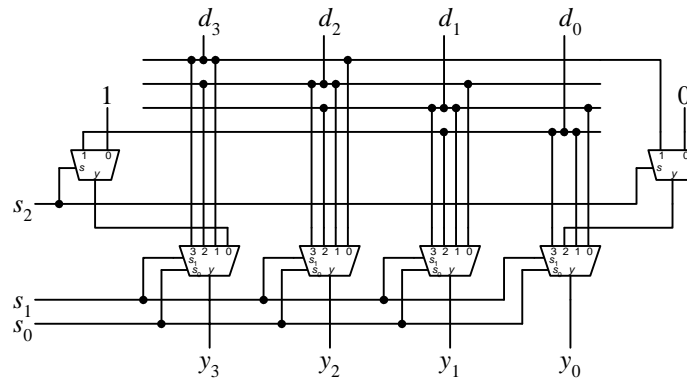| s_2	s_1	s_0	Operation
0	0	0	Pass through
0	0	1	Rotate left
0	1	0	Shift right and fill with 1
0	1	1	Not used
1	0	0	Shift left and fill with 0
1	0	1	Pass through
1	1	0	Rotate right
1	1	1	Shift right and fill with 0

4.31. Draw a 4-bit shifter circuit for the following operational table. Use only the basic gates AND, OR, and NOT (i.e. do not use multiplexers).

s_1	s_0	Operation
0	0	Shift left fill with 0
0	1	Shift right fill with 0
1	0	Rotate left
1	1	Rotate right

4.32. Draw a 4-bit shifter circuit for the following operation table using only six 2-to-1 multiplexers.

Operation
Shift left fill with 0
Shift right fill with 0
Rotate left
Rotate right

4.33. Derive the truth table for the following combinational circuit. Write also the operation name for each row in the table.

4.34. Draw a 4-bit barrel shifter circuit for the rotate right operation.

4.35. Draw the complete detail circuit diagram for the 4-bit multiplier based on the circuit shown in Figure 4.34(b).

Index

2's complement. *See* Two's complement

A

Active-high, 3
Active-low, 3
Adder, 3, 11
  carry-lookahead, 6
  full, 3
  ripple-carry, 5
AE. *See* Arithmetic logic unit.
ALU. *See* Arithmetic logic unit.
Arithmetic extender. *See* Arithmetic logic unit.
Arithmetic logic unit, 13
  AE arithmetic extender, 13
  CE carry extender, 14
  LE logic extender, 13
Assert, 3

B

Barrel shifter, 31

C

Carry extender. *See* Arithmetic logic unit.
Carry-lookahead adder, 6
CE. *See* Arithmetic logic unit.
Combinational components, **3**
Comparator, 26

D

De-assert, 3
Decoder, 18
Demultiplexer, 18

E

Encoder, 20
  priority, 20

F

FA. *See* Full adder.
Full adder, 3

I

Iterative circuit, 27

L

LE. *See* Arithmetic logic unit.
Logic extender. *See* Arithmetic logic unit.

M

Multiplexer, 21
Multiplier, 31
MUX. *See* Multiplexer.

N

Negative binary numbers, 7
Negative logic, 3

P

Positive logic, 3
Priority encoder, 20

R

Ripple-carry adder, 5
Rotating bits, 29
Rotator, 29

S

Shifter, 29
Shifting bits, 29
Sign extension, 9
Subtractor, 3, 9, 11

T

Tri-state buffer, 24
Two's-complement, 7

V

VHDL code
  3-to-8 decoder, 19
  4-to-1 multiplexer, 24
  adder/subtractor, 13
  arithmetic logic unit (ALU), 17
  full adder, 5
  shifter, 31
  tri-state buffer, 26