CHAPTER 7

COMBINATIONAL CIRCUIT DESIGN: PRACTICE

After learning the implementation of key VHDL constructs and reviewing the synthesis process in Chapters 4, 5 and 6, we are ready to study the construction and VHDL description of more sophisticated combinational circuits. Examples will show how to transform conceptual ideas into hardware and illustrate resource-sharing and circuit-shaping techniques to reduce circuit size and increase performance. This chapter follows and demonstrates the main theme of the book: to research an efficient design and derive the VHDL code accordingly.

7.1 DERIVATION OF EFFICIENT HDL DESCRIPTION

Although the appearance of VHDL code is very different from a schematic diagram, VHDL code is just another way to describe a circuit. Synthesis software carries out a series of refinements and transforms a textual VHDL description to a cell-level netlist. Although software can perform simplification and local optimization, it does not know the meaning or intention of the code and cannot exploit alternative designs or change the architectural of the circuit.

The quality of a design and its description are two independent factors. We can express the initial design by a schematic diagram or by a textual VHDL program. Similarly, we can realize and synthesize the design either manually by paper and pencil or automatically by synthesis software. Using VHDL and synthesis software does not lead automatically to either a good or a bad design. VHDL description and synthesis software, however, can

Copyright © 2006 John Wiley & Sons, Inc.

shield tedious implementation details and greatly simplify the realization process. They allow us to have more time to explore and investigate alternative design ideas.

Derivation of an efficient, synthesizable VHDL description requires two major tasks:

- Research to find an efficient design.
- Develop VHDL code that accurately describes the design.

For a problem in digital system development, there is seldom a single unique solution. A large number of possible designs exist. The resulting implementations differ in size and performance and their quality may vary significantly. There is no simple, mechanical way to derive an efficient design. It frequently relies on a designer's experience, insight and understanding of the problem.

After we find a design, the next step is to derive VHDL code that describes the design accurately. Although the VHDL textual code cannot precisely specify the final structural implementation, it describes the "big picture" that establishes the basic skeleton of the circuit. For a complex design, it is useful to draw a rough schematic sketch to help in locating the key components and identifying the critical path.

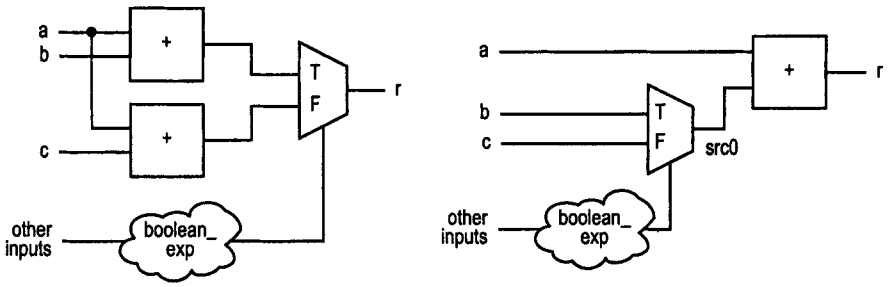
In addition to faithfully describing the intended design, good VHDL code should be clear and compact, and can be easily "scaled." Scalability concerns the amount of code modification needed when the signal width of a circuit changes. For example, after we develop a VHDL code for an 8-bit barrel shifter, how much modification is required if the input is increased to 16 bits, 32 bits or even 64 bits? The development of scalable and parameterized VHDL code is discussed in detail in Chapters 14 and 15. In this chapter, we need only be aware of this aspect of VHDL code, and discuss it when appropriate.

7.2 OPERATOR SHARING

When a VHDL program is synthesized, all statements and language constructs of the program will be mapped to hardware. One way to reduce the overall size of synthesized hardware is to identify the resources that can be used by different operations. This is known as *resource sharing*. Performing resource sharing normally introduces some overhead and may penalize performance, and thus is worthwhile only for large, complex constructs. Although the exact size depends on the underlying target technology, data from Table 6.2 provides a good estimation of the relative sizes of commonly synthesizable components. Ideally, synthesis software should identify the sharing opportunities and perform the optimization automatically. Unfortunately, in reality, software's capability varies and sometimes is rather limited in this respect. We may need to explicitly describe the desired sharing in VHDL code. This section discusses the operator sharing and the next section illustrates functionality sharing.

In certain VHDL constructs, operations are mutually exclusive; i.e., only one operation is active at a particular time. These constructs include the conditional signal assignment statement (or the equivalent if statement in a process) and the selected signal assignment statement (or the equivalent case statement in a process). Recall that the basic expression of a conditional signal assignment statement is

```
signal_name <= value_expr_1 when boolean_expr_1 else
               value_expr_2 when boolean_expr_2 else
               value_expr_3 when boolean_expr_3 else
               . . .
               value_expr_n;
```



(a) Original diagram

(b) Diagram with sharing

Figure 7.1 Simple operator sharing.

The value expressions `value_expr_1`, `value_expr_2`, ..., `value_expr_n` are mutually exclusive since only one expression needs to be evaluated and passed to output. Similarly, recall that the basic expression of a selected signal assignment statement is

```
with select_expression select
   signal_name <= value_expr_1 when choice_1,
                  value_expr_2 when choice_2,
                  value_expr_3 when choice_3,
                     . . .
                  value_expr_n when choice_n;
```

Since choices are mutually exclusive, only one expression actually has to be evaluated. The same argument can be applied to the if statement and the case statement since their implementations are similar to the conditional and selected signal assignment statements.

If the same operator is used in several different expressions, it can be shared. The sharing is normally done by routing the proper data to or from this particular operator. We demonstrate the coding technique in the following examples and discuss the degree of saving and its potential impact on system performance.

7.2.1 Sharing example 1

Consider the following code segment:

```
r <= a+b when boolean_exp else
     a+c;
```

The block diagram of this code is shown in Figure 7.1(a).

There are two adders and one multiplexer. The adder can be shared because only one addition operation is needed at any time. We can revise the code as follows:

```
src0 <= b when boolean_exp else
        c;
r <= a + src0;
```

The block diagram of the revised code is shown in Figure 7.1(b). Instead of multiplexing the addition results, it multiplexes the desired source operand to the input of the adder. One adder can be eliminated in this new implementation.

Now we compare the propagation delays of the two circuits. Let the propagation delays of the adder, the multiplexer and the `boolean_exp` circuit be T_{adder}, T_{mux} and $T_{boolean}$

respectively. In the first circuit, the adders and `boolean_exp` operate in parallel, and thus the overall propagation delay is $\max(T_{adder}, T_{boolean}) + T_{mux}$. In the second circuit, the propagation delay is $T_{boolean} + T_{mux} + T_{adder}$. This reflects the fact that the `boolean_exp` operation and addition operations are performed concurrently in the first circuit whereas they are done in cascade in the second circuit. If the `boolean_exp` circuit is very simple and its delay is negligible, there will be no performance penalty on the shared design.

7.2.2 Sharing example 2

Consider the following code segment:

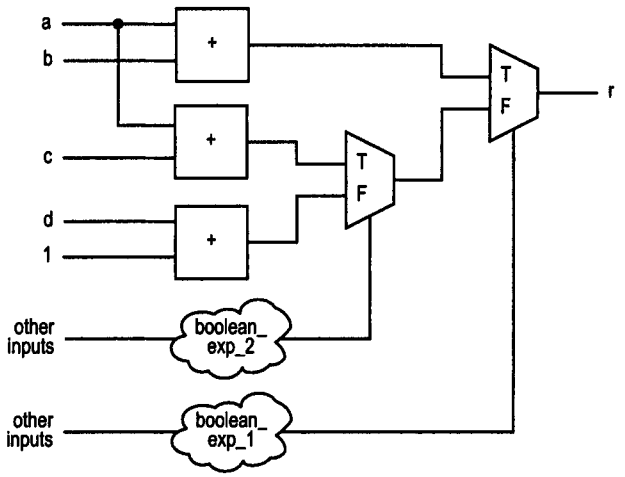
```
process(a,b,c,d,...)
begin
   if boolean_exp_1 then
      r <= a+b;
   elsif boolean_exp_2 then
      r <= a+c;
   else
      r <= d+1;
   end if;
end process;
```

The block diagram of this code is shown in Figure 7.2(a).

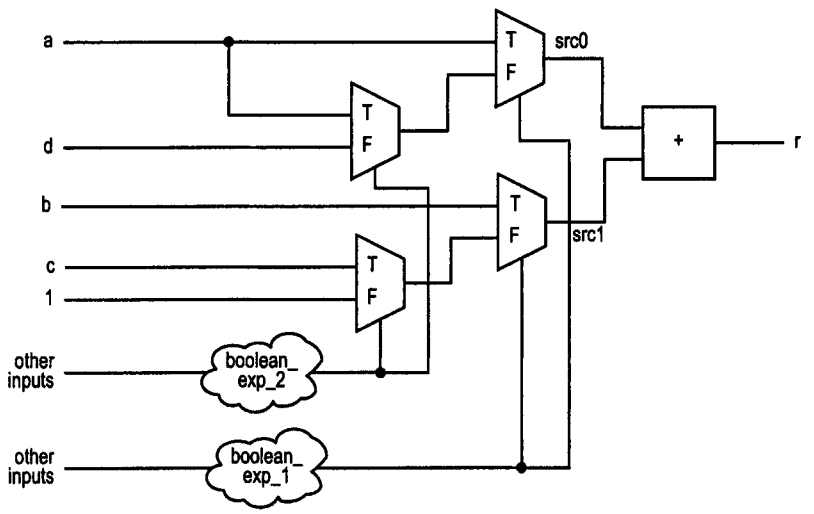
The implementation needs two adders, one incrementor and two multiplexers. The addition and increment operations can share the same adder because only one branch of the if statement is executed at a time. Assume that the signals are 8 bits wide. The revised code becomes

```
process(a,b,c,d,...)
begin
   if boolean_exp_1 then
      src0 <= a;
      src1 <= b;
   elsif boolean_exp_2 then
      src0 <= a;
      src1 <= c;
   else
      src0 <= d;
      src1 <= "00000001";
   end if;
end process;
r <= src0 + src1;
```

The block diagram of the new code is shown in Figure 7.2(b). We use two multiplexers to route the desired source operands to the inputs of the adder. The new circuit eliminates one adder and one incrementor but requires two additional multiplexers. To determine whether the sharing is worthwhile, we examine the circuit size of the adder, incrementor and multiplexer given in Table 6.2. Since a multiplexer is smaller, especially when compared with an adder, the sharing indeed leads to a smaller size. It is likely that the multiplexing circuit can be further simplified during logic synthesis, due to the duplicated input patterns (the a signal is used twice) and constant input ("00000001"). The saving will become more significant if a high-performance adder (the one optimized for delay) is used.

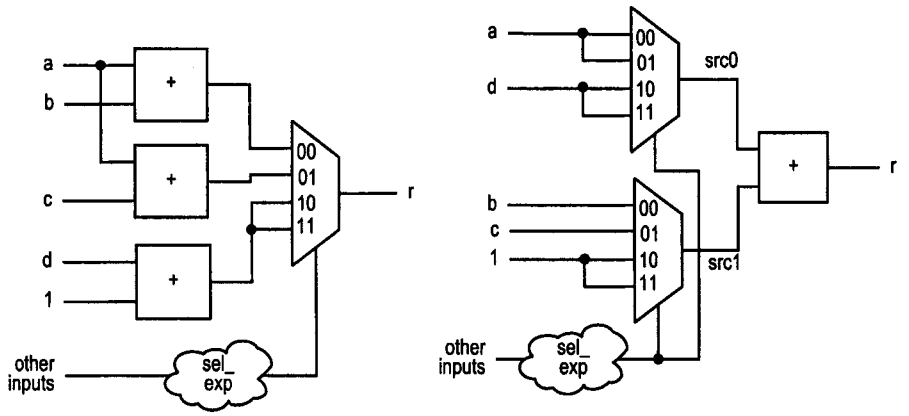


(a) Original diagram



(b) Diagram with sharing

Figure 7.2 Operator sharing based on a priority network.



(a) Original diagram

(b) Diagram with sharing

Figure 7.3 Operator sharing based on a multiplexer.

Determining the propagation delays of these circuits is more involved since they depend on the relative values of the delays of the `boolean_exp_1` circuit, the `boolean_exp_2` circuit and the multiplexer. However, observation from the previous example still applies. The two Boolean circuits and three adders operate in parallel in the first circuit whereas the Boolean circuits and the adder operate in cascade in the second circuit. Thus, the first circuit should always have a smaller propagation delay.

7.2.3 Sharing example 3

Assume that the `sel` signal is 2 bits wide. Consider the following code segment:

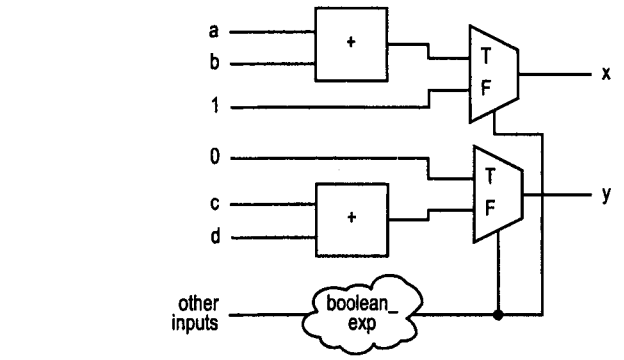
```
with sel_exp select
   r <= a+b when "00",
        a+c when "01",
        d+1 when others;
```

This example is similar to the previous one but uses the selected signal assignment statement. The block diagram of this code is shown in Figure 7.3(a).

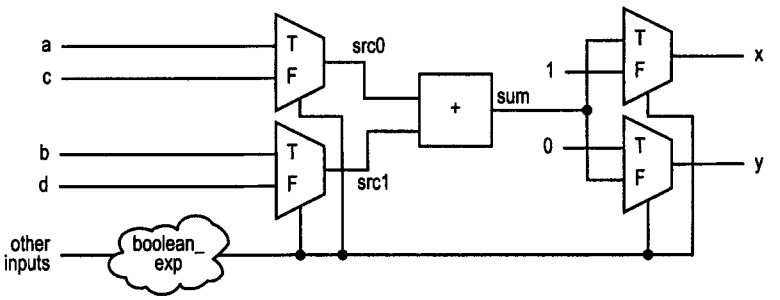
The circuit needs two adders, one incrementor and one 4-to-1 multiplexer. We can revise the code to share the adder:

```
with sel_exp select
   src0 <= a when "00"|"01",
           d when others;
with sel_exp select
   src1 <= b when "00",
           c when "01",
           "00000001" when others;
r <= src0 + src1;
```

The block diagram of the new code is shown in Figure 7.3(b). We use two multiplexers to route the desired source operands to the adder. The new circuit eliminates one adder and one incrementor but requires an additional 4-to-1 multiplexer. Since an adder and an



(a) Original diagram



(b) Diagram with sharing

Figure 7.4 Complex operator sharing.

incrementor are more complex than a multiplexer, the revision leads to a significant saving. Again, the second circuit may suffer a longer propagation delay because of the cascaded operations, as in example 1.

7.2.4 Sharing example 4

Consider the following code segment:

```
process(a,b,c,d,...)
begin
   if boolean_exp then
      x <= a + b;
      y <= (others=>'0');
   else
      x <= "00000001";
      y <= c + d;
   end if;
end process;
```

The block diagram of this code is shown in Figure 7.4(a). The implementation needs two adders and two multiplexers. The adder can be shared since the executions of two branches of the if statement are mutually exclusive. The revised code is as follows:

```
process(a,b,c,d,sum,...)
begin
   if boolean_exp then
      src0 <= a;
      src1 <= b;
      x <= sum;
      y <= (others=>'0');
   else
      src0 <= c;
      src1 <= d;
      x <= "00000001";
      y <= sum;
   end if;
end process;
sum <= src0 + src1;
```

The block diagram of this code is shown in Figure 7.4(b). This example illustrates the worst-case scenario of operator sharing, in which the operator has no common sources or destinations. We need a multiplexing structure to route one set of signals to the adder's input and a demultiplexing structure to route the addition result to one of the two output signals. The demultiplexing is done using two 2-to-1 multiplexers. Note that the addition result (the sum signal) is connected to the T port of the upper output multiplexer and the F port of the lower output multiplexer.

The new circuit eliminates one adder but adds two additional multiplexers. The merit of sharing in this circuit is less clear, and it depends on the relative sizes of an adder and two multiplexers. Again, we use the numbers given in Table 6.2 for estimation. If a slow adder ($+_a$, optimized for area) is used, the size of two multiplexers is about the same as that of one adder. On the other hand, if a faster adder ($+_d$, optimized for delay) is used, the saving is significant.

7.2.5 Summary

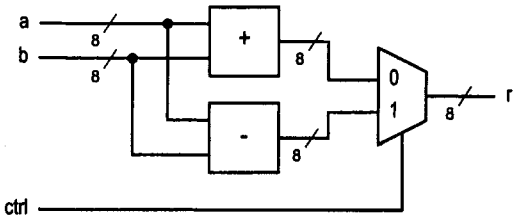
Operator sharing is done by providing additional multiplexing circuits to route input and output signals into or out of the operator. The merit of sharing and the degree of saving depend on the relative complexity of the multiplexing circuit and the operator. Substantial savings are possible for complex operators. However, sharing normally forces evaluation of the Boolean expressions and evaluation of the operators to be performed in cascade and thus may introduce extra propagation delay.

7.3 FUNCTIONALITY SHARING

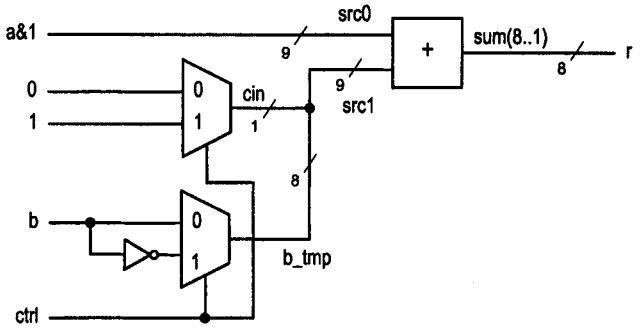
In a large, complex digital system, such as a processor, an array of functions is needed. Some functions may be related and have certain common characteristics. It is possible for several functions to share a common circuit or to utilize one function to construct another function. We call this approach *functionality sharing*. Unlike operator sharing, there is no systematic way to identify functionality sharing. This kind of sharing is done in an ad hoc, case-by-case basis and relies on the designer's insight and intimate understanding of the system. It is more difficult for synthesis software to identify functionality sharing.

ctrl	operation
0	a + b
1	a - b

(a) Function table



(b) Initial block diagram



(c) Diagram with sharing

Figure 7.5 Addition–subtraction circuit.

7.3.1 Addition–subtraction circuit

Consider a simple arithmetic circuit that performs either addition or subtraction. A control signal, `ctrl`, specifies the desired operation. The function table of this circuit is shown in Figure 7.5(a).

Our first design follows the function table, and the VHDL code is very straightforward, as shown in Listing 7.1. Note that the signals are converted to the `signed` data type internally to accommodate arithmetic operation.

Listing 7.1 Initial description of an addition–subtraction circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity addsub is
   port(
      a,b: in std_logic_vector(7 downto 0);
      ctrl: in std_logic;
      r: out  std_logic_vector(7 downto 0)
   );
end addsub;

architecture direct_arch of addsub is
   signal src0, src1, sum: signed(7 downto 0);
begin
   src0 <= signed(a);
```

```
    src1 <= signed(b);
    sum <= src0 + src1 when ctrl='0' else
           src0 - src1;
    r <= std_logic_vector(sum);
end direct_arch;
```

The conceptual diagram for this code is shown in Figure 7.5(b), which consists of an adder, a subtractor and a 2-to-1 multiplexer.

Since the adder and subtractor are different operators, we cannot directly apply the earlier operator-sharing technique. In 2's-complement representation, recall that the subtraction, $a - b$, can be calculated indirectly as $a + \bar{b} + 1$, where $\bar{b}$ is the bitwise inversion of b. Therefore, it is possible to share the functionality of the adder. After inverting b and putting a carry-in of 1, we can utilize the same adder to perform subtraction. The VHDL code is shown in Listing 7.2.

Listing 7.2 More efficient description of an addition–subtraction circuit

```
architecture shared_arch of addsub is
   signal src0, src1, sum: signed(7 downto 0);
   signal cin: signed(0 downto 0); -- carry-in bit
begin
   src0 <= signed(a);
   src1 <= signed(b) when ctrl='0' else
           signed(not b);
   cin <= "0" when ctrl='0' else
          "1";
   sum <= src0 + src1 + cin;
   r <= std_logic_vector(sum);
end shared_arch;
```

Note that the expression `a + src_b + cin` has two addition operators. Since `cin` is either `"0"` or `"1"`, it can be mapped to the carry-in port of a typical adder. In other words, the `+ cin` operation can be embedded into the `a + src_b` operation and no separate incrementor is needed. Most synthesis software should be able to derive the correct implementation.

Alternatively, we can manually describe the carry-in operation and use only one addition operator in the VHDL code. The trick is to use an extra bit in the adder to mimic the effect of carry-in operation. The internal adder is extended to 9 bits, in which the original input takes 8 MSBs and the extra bit is the LSB. The LSBs of the two operands are connected to 1 and the carry-in input, c_{in}, respectively. For example, if the two original operands are

$$a_7a_6a_5a_4a_3a_2a_1a_0 \text{ and } b_7b_6b_5b_4b_3b_2b_1b_0$$

The extended operands will be

$$a_7a_6a_5a_4a_3a_2a_1a_01 \text{ and } b_7b_6b_5b_4b_3b_2b_1b_0c_{in}$$

After the addition, the LSB will be discarded and the higher 8 bits will be used as the output. When c_{in} is 1, a carry will be propagated from the LSB to 8 MSBs, effectively adding 1 to the 8 MSBs of the adder. On the other hand, when c_{in} is 0, no carry occurs. Since the LSB of the sum is discarded, there is no impact on the addition of 8 MSBs. The VHDL code of this design is shown in Listing 7.3.

Listing 7.3 *Manual carry-in description of an addition–subtraction circuit*

```
architecture manual_carry_arch of addsub is
   signal src0, src1, sum: signed(8 downto 0);
   signal b_tmp: std_logic_vector(7 downto 0);
   signal cin: std_logic; -- carry-in bit
begin
   src0 <= signed(a & '1');
   b_tmp <= b when ctrl='0' else
            not b;
   cin <= '0' when ctrl='0' else
          '1';
   src1 <= signed(b_tmp & cin);
   sum <= src0 + src1;
   r <= std_logic_vector(sum(8 downto 1));
end manual_carry_arch;
```

The diagram for this design is shown in Figure 7.5(c).

7.3.2 Signed–unsigned dual-mode comparator

In the IEEE `numeric_std` package, the `signed` and `unsigned` data types are defined to represent an array of bits as signed and unsigned integers respectively. The `signed` data type is in 2's-complement format. An example of 4-bit binary representations and their signed and unsigned interpretations are shown as a "binary wheel" in Figure 7.6. Note that the addition and subtraction operations are identical for the two data types. The addition and subtraction of a positive amount corresponds to a move clockwise and counterclockwise along the wheel, and thus the same hardware can be used. However, this is not true for relational operators.

This example considers a greater-than comparator in which the input can be interpreted as either unsigned or signed. The input data type (or the operation mode of the comparator) is specified by a control signal, `mode`. Our first design uses two comparators, one for each data type, and then uses the `mode` signal to select the desired result. The VHDL code is shown in Listing 7.4. Note that by definition of VHDL, the comparison in `std_logic_vector` data type (i.e., `a > b`) and the comparison in `unsigned` date type (i.e., `unsigned(a) > unsigned(b)`) implies the same implementation. For clarity, we use the latter in the VHDL code.

Listing 7.4 Initial description of a dual-mode comparator

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity comp2mode is
   port(
      a,b: in std_logic_vector(7 downto 0);
      mode: in std_logic;
      agtb: out std_logic
   );
end comp2mode;

architecture direct_arch of comp2mode is
   signal agtb_signed, agtb_unsigned: std_logic;
```

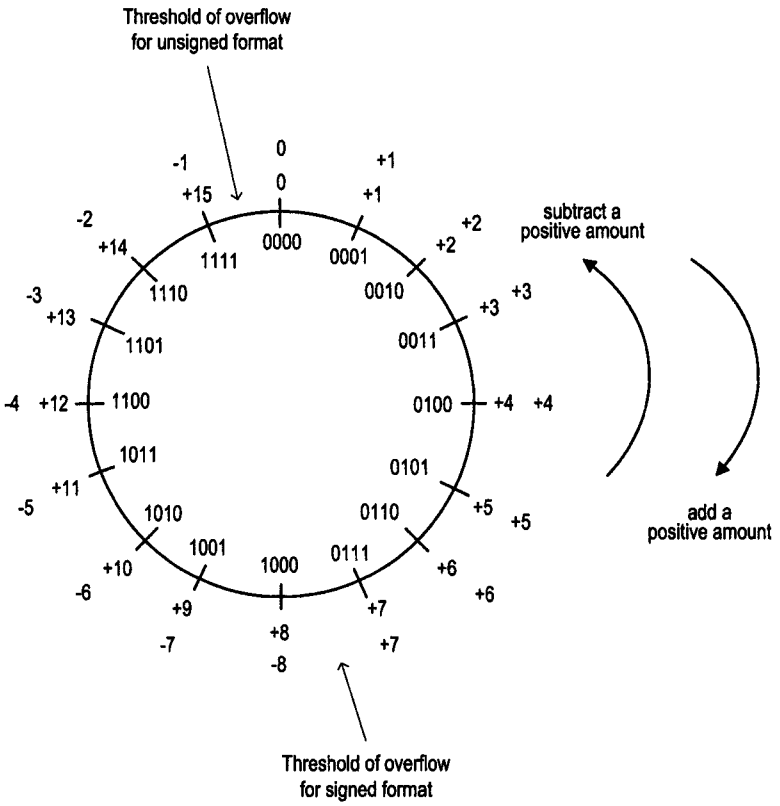


Figure 7.6 Four-bit binary wheel.

```
  begin
15    agtb_signed <= '1' when signed(a) > signed(b) else
                     '0';
      agtb_unsigned <= '1' when unsigned(a) > unsigned(b) else
                       '0';
      agtb <= agtb_unsigned when (mode='0') else
20            agtb_signed;
  end direct_arch ;
```

To identify a potential sharing opportunity, we must examine the implementation of a comparator for the `signed` data type. First, if two inputs have different sign bits, the one with '0' is greater than the one with '1' since a positive number or 0 is always greater than a negative number. If two inputs have the same sign, we can ignore the sign bit and compare the remaining bits in a regular fashion (i.e., as the `unsigned` or `std_logic_vector` data type). At first glance, this may not be obvious for two negative numbers. We can verify it by checking the binary representations of signed numbers in Figure 7.6. For example, the binary representations of -1, -4 and -7 are "1111" and "1100" and "1001". After discarding the sign bit, we can see that "111" > "100" > "001", which is consistent with $-1 > -4 > -7$. Based on this observation, we can develop the rules for a dual-mode comparator:

- If a and b have the same sign bit, compare the remaining bits in a regular fashion.

- If a's sign bit is '1' and b's sign bit is '0', a is greater than b when in unsigned mode and b is greater than a when in signed mode.
- If a's sign bit is '0' and b's sign bit is '1', reverse the previous result.

The VHDL code for the design is shown in Listing 7.5. The agtb_mag signal is the comparison result of 7 LSBs of a and b, and the a1_b0 signal is a special status indicating that the MSBs (signs) of a and b are '1' and '0' respectively. The last conditional signal assignment statement translates the previous rules into logic expressions.

Listing 7.5 More efficient description of a dual-mode comparator

```
architecture shared_arch of comp2mode is
signal a1_b0, agtb_mag: std_logic;
begin
   a1_b0 <= '1' when a(7)='1' and b(7)='0' else
            '0';
   agtb_mag <= '1' when a(6 downto 0) > b(6 downto 0) else
               '0';
   agtb <= agtb_mag when (a(7)=b(7)) else
           a1_b0 when mode='0' else
           not a1_b0;
end shared_arch;
```

The new design eliminates one comparator and reduces the circuit size of the dual-mode comparator by one half.

7.3.3 Difference circuit

Assume that we want to implement a circuit that takes two unsigned numbers and calculates their difference; i.e., performs the function |a - b|. The straightforward design is to compute both a - b and b - a, compare a and b, and then select the proper subtraction result accordingly. The VHDL code is shown in Listing 7.6. Note that the signals are converted to the unsigned data type for arithmetic operation.

Listing 7.6 Initial description of a difference circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity diff is
   port(
      a,b: in std_logic_vector(7 downto 0);
      result: out std_logic_vector(7 downto 0)
    );
end diff;

architecture direct_arch of diff is
   signal au, bu, ru, diffab, diffba: unsigned(7 downto 0);
begin
   au <= unsigned(a);
   bu <= unsigned(b);
   diffab <= au - bu;
   diffba <= bu - au;
   ru <= diffab when (au >= bu) else
         diffba;
```

```
20    result <= std_logic_vector(ru);
  end direct_arch;
```

One observation about the initial design is the implementation of the relational operation >=. The result of a >= b can be indirectly obtained from a - b by examining the sign bit of the subtraction result. If the sign bit is '0', the result is positive or 0 and thus a >= b is true. Otherwise, the result is negative and a >= b is false. We consider this scenario as functionality sharing since the operation a >= b indirectly utilizes the functionality of a - b. To apply the idea to this example, we must modify the internal representation since the original inputs a and b are interpreted as unsigned numbers and have no sign bit. We extend the internal signals by one bit and interpret them as a signed number. The VHDL code of the revised design is shown in Listing 7.7. Note that since both extended signals, as and bs, are positive (with '0' in MSB) and subtraction is performed, we need not worry about the overflow condition and can use the sign bit of a - b (i.e., diffab(8) in code) directly.

Listing 7.7 Better description of a difference circuit

```
architecture shared_arch of diff is
   signal as, bs, rs, diffab, diffba: signed(8 downto 0);
begin
   as <= signed('0'&a);
5  bs <= signed('0'&b);
   diffab <= as - bs;
   diffba <= bs - as;
   rs <= diffab when diffab(8)='0' else
         diffba;
10 result <= std_logic_vector(rs(7 downto 0));
end shared_arch;
```

The revised design can be further optimized by replacing the b - a expression with 0 - diffab (or simply -diffab). Since the 0 - diffab operation has a constant operand (i.e., 0), the circuit size is about half that of a full subtractor. The final code is listed in Listing 7.8.

Listing 7.8 Most efficient description of a difference circuit

```
architecture effi_arch of diff is
   signal as, bs, rs, diffab, diffba: signed(8 downto 0);
begin
   as <= signed('0'&a);
5  bs <= signed('0'&b);
   diffab <= as - bs;
   diffba <= 0 - diffab;
   rs <= diffab when diffab(8)='0' else
         diffba;
10 result <= std_logic_vector(rs(7 downto 0));
end effi_arch;bout
```

An alternative design approach is to use the operator-sharing technique. The code is shown in Listing 7.9. We first compare the two inputs and route the larger one to src0 and smaller one to src1, and then perform the subtraction. The design requires one subtractor and one comparator, and its size is comparable to that of the effi_arch architecture in Listing 7.8.

Listing 7.9 Alternative description of a difference circuit

```
architecture shared3_arch of diff is
   signal au, bu, ru, src0, src1: unsigned(7 downto 0);
begin
   au <= unsigned(a);
   bu <= unsigned(b);
   process(au,bu)
   begin
      if au >= bu then
         src0 <= au;
         src1 <= bu;
      else
         src0 <= bu;
         src1 <= au;
      end if;
   end process;
   ru <= src0 - src1;
   result <= std_logic_vector(ru);
end shared3_arch;
```

7.3.4 Full comparator

Assume that we need a comparator that has three outputs, indicating the greater-than, equal-to and less-than conditions respectively. The straightforward design is to use three relational operators, each for an output condition. The VHDL code for this design is shown in Listing 7.10. Clearly, three separate relational circuits are needed when it is synthesized.

Listing 7.10 Initial description of a full comparator

```
library ieee;
use ieee.std_logic_1164.all;
entity comp3 is
   port(
      a,b: in std_logic_vector(15 downto 0);
      agtb, altb, aeqb: out std_logic
   );
end comp3 ;

architecture direct_arch of comp3 is
begin
   agtb <= '1' when a > b else
           '0';
   altb <= '1' when a < b else
           '0';
   aeqb <= '1' when a = b else
           '0';
end direct_arch;
```

If we examine the three operations carefully, we can see that the three conditions are mutually exclusive, and the third one can be derived if the other two are known. Thus, the functionality of the first two relational circuits can be shared to obtain the third output. The code of the revised design is shown in Listing 7.11.

Listing 7.11 Better description of a full comparator

```
architecture share1_arch of comp3 is
   signal gt, lt: std_logic;
begin
   gt <= '1' when a > b else
         '0';
   lt <= '1' when a < b else
         '0';
   agtb <= gt;
   altb <= lt;
   aeqb <= not (gt or lt);
end share1_arch;
```

The third statement means "a is equal to b" if the condition "a is greater than b or a is less than b" is not true. This simple revision eliminates the comparison circuit for the equal-to operator.

If we look Table 6.2, the equal-to circuit is smaller and faster than the greater-than circuit (especially compared with the circuit optimized for delay). This is due to the internal implementation of these circuits. We can further optimize the circuit by using the equal-to operator to replace either the greater-than or less-than operator. The code of the final design is shown in Listing 7.12.

Listing 7.12 Most efficient description of a full comparator

```
architecture share2_arch of comp3 is
   signal eq, lt: std_logic;
begin
   eq <= '1' when a = b else
         '0';
   lt <= '1' when a < b else
         '0';
   aeqb <= eq;
   altb <= lt;
   agtb <= not (eq or lt);
end share2_arch;
```

Although the observation of mutual exclusiveness of three outputs is trivial for us, it involves the meaning (semantics) of the operators. Most synthesis software is unable to take advantage of this property and optimize the code segment.

7.3.5 Three-function barrel shifter

A barrel shifter is a circuit that can shift input data by any number of positions. Both VHDL standard and the IEEE `std_logic_1164` package define a set of shifting and rotating operators. Because of the complexity of the shifting circuit, some synthesis software is unable to synthesize these operators automatically. Shifting operations can be done in either the left or right direction and are divided into *rotate*, *logic shift* and *arithmetic shift*. In this example, we consider an 8-bit shifting circuit that can perform rotate right, logic shift right or arithmetic shift right, in which lower bits, 0's or sign bits are shifted into left positions respectively. In addition to the 8-bit data input, this circuit has a control signal, `lar` (for logic shift, arithmetic shift and rotate), which specifies the operation to be

performed, and a control signal, amt (for amount), which specifies the number of positions to be rotated or shifted.

A straightforward design is to construct a rotate-right circuit, a logic shift-right circuit and an arithmetic shift-right circuit, and then use a multiplexer to select the desired output. The VHDL code of this design is shown in Listing 7.13. The individual shifting circuit is implemented by a selected signal assignment statement.

Listing 7.13 Initial description of a barrel shifter

```
library ieee;
use ieee.std_logic_1164.all;
entity shift3mode is
   port(
      a: in std_logic_vector(7 downto 0);
      lar: in std_logic_vector(1 downto 0);
      amt: in std_logic_vector(2 downto 0);
      y: out std_logic_vector(7 downto 0)
   );
end shift3mode ;

architecture direct_arch of shift3mode is
   signal logic_result, arith_result, rot_result:
      std_logic_vector(7 downto 0);
begin
   with amt select
       rot_result <=
          a                                  when "000",
          a(0) & a(7 downto 1)               when "001",
          a(1 downto 0) & a(7 downto 2)      when "010",
          a(2 downto 0) & a(7 downto 3)      when "011",
          a(3 downto 0) & a(7 downto 4)      when "100",
          a(4 downto 0) & a(7 downto 5)      when "101",
          a(5 downto 0) & a(7 downto 6)      when "110",
          a(6 downto 0) & a(7) when others; -- 111
   with amt select
      logic_result <=
         a                                 when "000",
         "0"       & a(7 downto 1)         when "001",
         "00"      & a(7 downto 2)         when "010",
         "000"     & a(7 downto 3)         when "011",
         "0000"    & a(7 downto 4)         when "100",
         "00000"   & a(7 downto 5)         when "101",
         "000000"  & a(7 downto 6)         when "110",
         "0000000" & a(7) when others; -- 111
   with amt select
      arith_result <=
         a                                  when "000",
         a(7) & a(7 downto 1)               when "001",
         a(7)&a(7)     & a(7 downto 2)      when "010",
         a(7)&a(7)&a(7)& a(7 downto 3)      when "011",
         a(7)&a(7)&a(7)&a(7)&
             a(7 downto 4)                  when "100",
         a(7)&a(7)&a(7)&a(7)&a(7)&
             a(7 downto 5)                  when "101",
```

```
            a(7)&a(7)&a(7)&a(7)&a(7)&a(7)&
                a(7 downto 6)                when "110",
            a(7)&a(7)&a(7)&a(7)&a(7)&a(7)&a(7)&
                a(7)                         when others;
50    with lar select
         y <= logic_result when "00",
              arith_result when "01",
              rot_result   when others;
  end direct_arch;
```

The implementation includes three 8-bit 8-to-1 multiplexers and one 8-bit 3-to-1 multiplexer.

If we examine the output of three shifting operations, we can see that their patterns are very similar and the only difference is the data being shifted into the left part. It is possible to share the functionality of a shifting circuit. To take advantage of this, we use a preprocessing circuit to modify the left part of the input data to the desired format and then pass it to the shifting circuit. The VHDL code based on this idea is given in Listing 7.14.

Listing 7.14 Better description of a barrel shifter

```
  architecture shared_arch of shift3mode is
     signal shift_in: std_logic_vector(7 downto 0);
  begin
     with lar select
5       shift_in <= (others=>'0')  when "00",
                    (others=>a(7)) when "01",
                    a              when others;
     with amt select
        y <= a                                      when "000",
10           shift_in(0)          & a(7 downto 1)  when "001",
             shift_in(1 downto 0) & a(7 downto 2)  when "010",
             shift_in(2 downto 0) & a(7 downto 3)  when "011",
             shift_in(3 downto 0) & a(7 downto 4)  when "100",
             shift_in(4 downto 0) & a(7 downto 5)  when "101",
15           shift_in(5 downto 0) & a(7 downto 6)  when "110",
             shift_in(6 downto 0) & a(7)           when others;
  end shared_arch;
```

In this code, one 8-bit 3-to-1 multiplexer is used to preprocess the input. Depending on the `lar` signal, its output `shift_in` can be the a input, repetitive 0's or repetitive sign bits. The `shift_in` signal is then passed to the shifting circuit and becomes the left part of the final output. The improved design consists of one 8-bit 8-to-1 multiplexer and one 8-bit 3-to-1 multiplexer. It has a similar critical path but eliminates two 8-bit 8-to-1 multiplexers.

7.4 LAYOUT-RELATED CIRCUITS

After synthesis, placement and routing will derive the actual physical layout of a digital circuit on a silicon chip. Although we cannot use VHDL code to specify the exact layout, it is possible to outline the general "shape" of the circuit. This will help the synthesis process and the placement and routing process to derive a more efficient circuit. Examples in this section show how to shape the circuit layout in VHDL code.

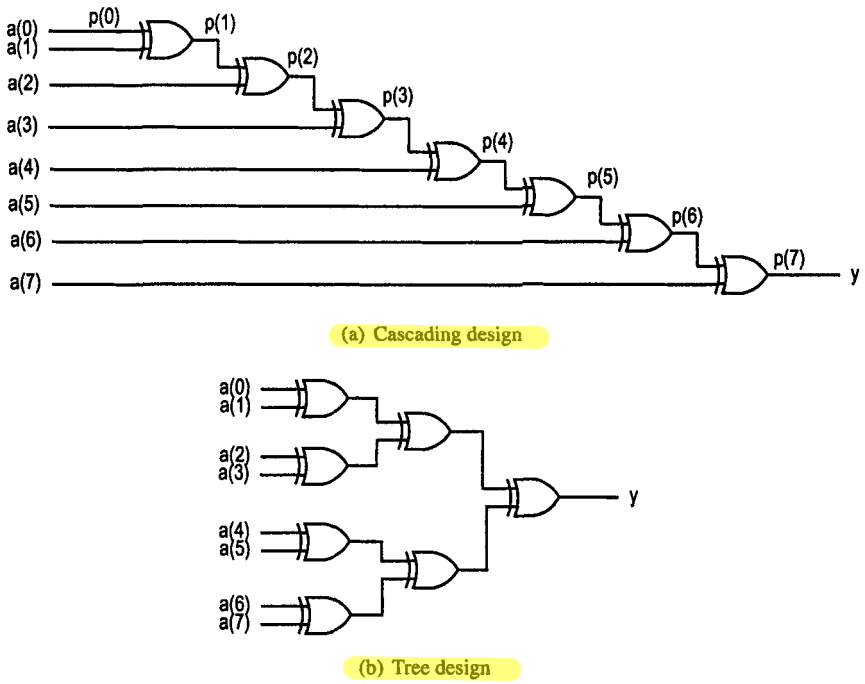


Figure 7.7 Reduced-xor circuit.

7.4.1 Reduced-xor circuit

A reduced-xor function is to apply xor operations over all bits of an input signal. For example, let $a_7a_6a_5a_4a_3a_2a_1a_0$ be an 8-bit signal. The reduced-xor of this signal is

$$a_7 \oplus a_6 \oplus a_5 \oplus a_4 \oplus a_3 \oplus a_2 \oplus a_1 \oplus a_0$$

Since this function returns '1' if there are odd number of 1's in its input, it can be used to determine the odd parity of the input signal.

A straightforward design is shown in Figure 7.7(a). This design can be easily transformed into a VHDL code, which is shown in Listing 7.15.

Listing 7.15 Initial description of a reduced-xor circuit

```
library ieee;
use ieee.std_logic_1164.all;
entity reduced_xor is
   port(
      a: in std_logic_vector(7 downto 0);
      y: out std_logic
   );
end reduced_xor;

architecture cascade1_arch of reduced_xor is
begin
   y <= a(0) xor a(1) xor a(2) xor a(3) xor
        a(4) xor a(5) xor a(6) xor a(7);
```

```
end cascade1_arch;
```

By VHDL definition, the xor operator is left associative. Thus, the expression

```
a(0) xor a(1) xor a(2) xor ... xor a(7)
```

is the same as

```
(...((a(0) xor a(1)) xor a(2)) xor ...) xor a(7))
```

We can also use an 8-bit internal signal, p, to represent the intermediate results, as in Figure 7.7(a). The code for the architecture body is shown in Listing 7.16.

Listing 7.16 Alternative description of a reduced-xor circuit

```
architecture cascade2_arch of reduced_xor is
   signal p: std_logic_vector(7 downto 0);
begin
   p(0) <= a(0);
   p(1) <= p(0) xor a(1);
   p(2) <= p(1) xor a(2);
   p(3) <= p(2) xor a(3);
   p(4) <= p(3) xor a(4);
   p(5) <= p(4) xor a(5);
   p(6) <= p(5) xor a(6);
   p(7) <= p(6) xor a(7);
   y <= p(7);
end cascade2_arch;
```

Except for the first statement, a clear pattern exists between the inputs and outputs of these statements. By Boolean algebra, we know that $x = 0 \oplus x$. We can rewrite the first statement as

```
p(0) <= '0' xor a(0);
```

to make it match the pattern. Once this is done, we can use a more compact vector form to replace these statements, as shown in Listing 7.17.

Listing 7.17 Compact description of a reduced-xor circuit

```
architecture cascade_compact_arch of reduced_xor is
   constant WIDTH: integer := 8;
   signal p: std_logic_vector(WIDTH-1 downto 0);
begin
   p <= (p(WIDTH-2 downto 0) & '0') xor a;
   y <= p(WIDTH-1);
end cascade_compact_arch;
```

Although this design uses a minimal number of xor gates, it suffers a long propagation delay. The single cascading chain of xor gates becomes the critical path and the corresponding propagation delay is proportional to the number of xor gates in the chain. As the number of inputs increases, the propagation delay increases proportionally. Thus, the delay has an order of $O(n)$. Because of the associativity of the **xor** operator, we can arbitrarily change the order of operation. The initial design can be rearranged as a tree to reduce the length of its critical path, as shown in Figure 7.7(b). In VHDL code, we can use parentheses to force the desired order of operation, and the revised architecture body is shown in Listing 7.18.

Listing 7.18 Better description of a reduced-xor circuit

```
architecture tree_arch of reduced_xor is
begin
   y <= ((a(7) xor a(6)) xor (a(5) xor a(4))) xor
        ((a(3) xor a(2)) xor (a(1) xor a(0)));
end tree_arch;
```

In this new design, the critical path is reduced to three xor gates while the number of xor gates remains unchanged. Since we achieve better performance without adding extra hardware resource, it is a better design. In general, when we rearrange a cascade structure of n elements into a treelike structure, there will be $\log_2 n$ levels in the tree. The critical path is proportional to the number of levels in the tree and thus has an order of $O(\log_2 n)$.

Since this is a trivial circuit, synthesis software should be able to automatically transform the cascade design into the tree structure either by exploring the associative property or by performing time-constraint optimization. It is likely to obtain the same synthesis results for all codes in this example. However, for a more involved circuit, synthesis software is unable to do this, and we need to manually specify the order of operation to obtain a more efficient circuit.

Finally, let us examine the scalability of these codes. Assume that we want to increase the input to 16 bits. In the `cascade1_arch`, `cascade2_arch` and `tree_arch` architectures, we have to add eight additional `xor a(i)` terms or eight additional `p(i+1) <= p(i) xor a(i)` statements respectively. The number of revisions is proportional to the number of inputs and thus is on the order of $O(n)$. In the `cascade_compact_arch` architecture, the code remains the same except that the number in the constant statement has to be changed from 8 to 16. The needed revision is $O(1)$, and this code is highly scalable.

7.4.2 Reduced-xor-vector circuit

A reduced-xor-vector function is to apply xor operations over all possible combinations of lower bits of an input signal. It can best be explained by an example. Let $a_7a_6a_5a_4a_3a_2a_1a_0$ be an 8-bit signal. Applying the reduced-xor-vector function to it returns eight values, and they are defined as

$$
\begin{aligned}
y_0 &= a_0 \\
y_1 &= a_1 \oplus a_0 \\
y_2 &= a_2 \oplus a_1 \oplus a_0 \\
y_3 &= a_3 \oplus a_2 \oplus a_1 \oplus a_0 \\
y_4 &= a_4 \oplus a_3 \oplus a_2 \oplus a_1 \oplus a_0 \\
y_5 &= a_5 \oplus a_4 \oplus a_3 \oplus a_2 \oplus a_1 \oplus a_0 \\
y_6 &= a_6 \oplus a_5 \oplus a_4 \oplus a_3 \oplus a_2 \oplus a_1 \oplus a_0 \\
y_7 &= a_7 \oplus a_6 \oplus a_5 \oplus a_4 \oplus a_3 \oplus a_2 \oplus a_1 \oplus a_0
\end{aligned}
$$

A straightforward design is to follow the definition of this function, which can easily be transformed into the VHDL code shown in Listing 7.19.

Listing 7.19 Initial description of a reduced-xor-vector circuit

```
library ieee;
use ieee.std_logic_1164.all;
entity reduced_xor_vector is
   port(
      a: in std_logic_vector(7 downto 0);
      y: out std_logic_vector(7 downto 0)
   );
end reduced_xor_vector;

architecture direct_arch of reduced_xor_vector is
begin
   y(0) <= a(0);
   y(1) <= a(1) xor a(0);
   y(2) <= a(2) xor a(1) xor a(0);
   y(3) <= a(3) xor a(2) xor a(1) xor a(0);
   y(4) <= a(4) xor a(3) xor a(2) xor a(1) xor a(0);
   y(5) <= a(5) xor a(4) xor a(3) xor a(2) xor a(1) xor a(0);
   y(6) <= a(6) xor a(5) xor a(4) xor a(3) xor a(2) xor a(1)
           xor a(0);
   y(7) <= a(7) xor a(6) xor a(5) xor a(4) xor a(3) xor a(2)
           xor a(1) xor a(0);
end direct_arch;
```

In this code, each output is described independently, and no sharing is imposed. If no optimization is performed during synthesis, the synthesized circuit needs 28 xor gates. There are lots of common expressions that can be shared to reduce the number of xor gates.

Note that there is a simple relationship between the successive output values:

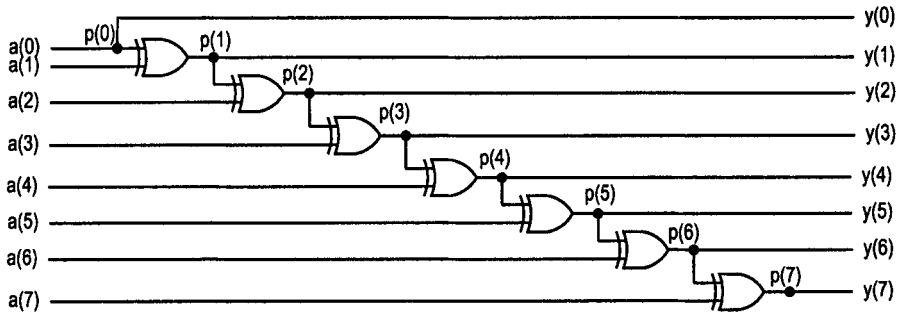
$$y_{i+1} = a_{i+1} \oplus y_i$$

The design based on this observation is shown in Figure 7.8(a), in which only seven xor gates are needed.

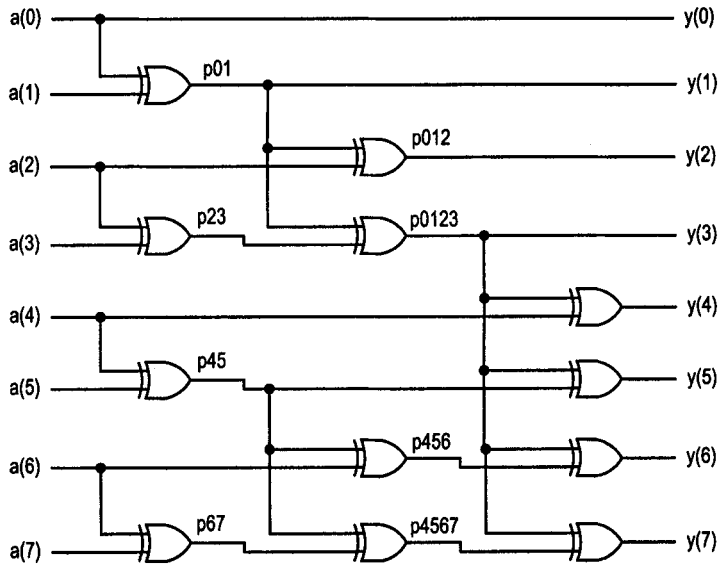
The VHDL code for this design is similar to the `cascade2_arch` architecture in Listing 7.16 except that all intermediate internal values are used as output. We need to modify the last statement and the VHDL code, as shown in Listing 7.20.

Listing 7.20 Sharing description of a reduced-xor-vector circuit

```
architecture shared1_arch of reduced_xor_vector is
   signal p: std_logic_vector(7 downto 0);
begin
   p(0) <= a(0);
   p(1) <= p(0) xor a(1);
   p(2) <= p(1) xor a(2);
   p(3) <= p(2) xor a(3);
   p(4) <= p(3) xor a(4);
   p(5) <= p(4) xor a(5);
   p(6) <= p(5) xor a(6);
   p(7) <= p(6) xor a(7);
   y <= p;
end shared1_arch;
```



(a) Cascading design



(b) Parallel-prefix design

Figure 7.8 Reduced-xor-vector circuit.

Similarly, the compact `cascade_compact_arch` architecture in Listing 7.17 can be revised, too, as shown in Listing 7.21.

Listing 7.21 Compact description of reduced-xor-vector circuit

```
architecture shared_compact_arch of reduced_xor_vector is
   constant WIDTH: integer := 8;
   signal p: std_logic_vector(WIDTH-1 downto 0);
begin
   p <= (p(WIDTH-2 downto 0) & '0') xor a;
   y <= p;
end shared_compact_arch;
```

The critical path of this circuit is the path to obtain the `y(7)` signal, which has the largest number of xor gates along the path. Our earlier discussion shows that the propagation delay is on the order of $O(n)$. To increase the performance, we have to rearrange the cascading chain to a treelike structure. The simple xor tree of `tree_arch` architecture of the previous example is not adequate since it cannot produce all the needed output values. One straightforward design is to create an independent xor tree for each output value. The design needs 28 xor gates, and the critical path of the circuit is the critical path of the tree used to implement `y(7)`, which is the largest tree and has three levels of xor gates. The VHDL code is similar to the `direct_arch` architecture except that we use parentheses to force the order of evaluation, as shown in Listing 7.22.

Listing 7.22 Tree description of a reduced-xor-vector circuit

```
architecture direct_tree_arch of reduced_xor_vector is
begin
   y(0) <= a(0);
   y(1) <= a(1) xor a(0);
   y(2) <= a(2) xor a(1) xor a(0);
   y(3) <= (a(3) xor a(2)) xor (a(1) xor a(0));
   y(4) <= (a(4) xor a(3)) xor (a(2) xor a(1)) xor a(0);
   y(5) <= (a(5) xor a(4)) xor (a(3) xor a(2)) xor
           (a(1) xor a(0));
   y(6) <= ((a(6) xor a(5)) xor (a(4) xor a(3))) xor
           ((a(2) xor a(1)) xor a(0));
   y(7) <= ((a(7) xor a(6)) xor (a(5) xor a(4))) xor
           ((a(3) xor a(2)) xor (a(1) xor a(0)));
end direct_tree_arch;
```

A more elegant design is shown in Figure 7.8(b). This design is targeted for performance and limits the critical path within three levels of xor gates. Within this constraint, it tries to share as many common expressions as possible. Instead of 28 xor gates, this design needs only 12 xor gates. We can derive the VHDL code according to the circuit diagram, as shown in Listing 7.23.

Listing 7.23 Parallel-prefix description of a reduced-xor-vector circuit

```
architecture optimal_tree_arch of reduced_xor_vector is
   signal p01, p23, p45, p67, p012,
          p0123, p456, p4567: std_logic;
begin
   p01 <= a(0) xor a(1);
   p23 <= a(2) xor a(3);
```

```
    p45 <= a(4) xor a(5);
    p67 <= a(6) xor a(7);
    p012 <= p01 xor a(2);
    p0123 <= p01 xor p23;
    p456 <= p45 xor a(6);
    p4567 <= p45 xor p67;
    y(0) <= a(0);
    y(1) <= p01;
    y(2) <= p012;
    y(3) <= p0123;
    y(4) <= p0123 xor a(4);
    y(5) <= p0123 xor p45;
    y(6) <= p0123 xor p456;
    y(7) <= p0123 xor p4567;
end optimal_tree_arch;
```

Although the same design principle can be used for a circuit with a larger number of inputs, revising the VHDL code will be very tedious and error-prone. Actually, this design is not just a lucky observation. It is based on *parallel-prefix structure*, and the systematic development of VHDL code for this circuit is discussed in Chapter 15.

There are two important observations for this example. The first is the trade-off between circuit size and performance. In a digital circuit, we normally have to use more hardware resources to improve the performance, as in this example. The cascading design needs a minimal number of xor gates, which is on the order of $O(n)$, but suffers a large propagation delay, which is also on the order of $O(n)$. The parallel-prefix design, on the other hand, requires $0.5n \log_2 n$ xor gates, but its delay is only on the order of $O(\log_2 n)$.

The second observation is about the capability of the synthesis software. Ideally, we hope the synthesis software can automatically derive the desired implementation regardless of the initial VHDL description. This is hardly possible, even for the simple function used in this example.

7.4.3 Tree priority encoder

A priority encoder is a circuit that returns the codes for the highest-priority request. We have discussed it in Chapters 4 and 5 and used different VHDL constructs to describe this circuit. The conditional signal assignment and if statements are natural to describe this function, and they specify the same priority routing network. The shape of the priority routing network is a single cascading chain, somewhat similar to the layout of cascading reduced-xor design in the previous example. Since the critical path is formed along this chain, performance suffers when the number of inputs increases. In the reduced-xor circuit, we can convert the cascading chain into a tree by rearranging the order of xor operations. This is also possible for the priority encoder, although the rearrangement is more involved. This example shows how to create an alternative treelike structure. We use a 16-to-4 priority encoder to demonstrate the scheme.

The VHDL description for the cascading design is straightforward, as shown in Listing 7.24.

Listing 7.24 Cascading description of a priority encoder

```
library ieee;
use ieee.std_logic_1164.all;
entity prio_encoder is
```

```
    port(
5       r: in std_logic_vector(15 downto 0);
        code: out std_logic_vector(3 downto 0);
        active: out std_logic
    );
end prio_encoder;
10
architecture cascade_arch of prio_encoder is
begin
    code <= "1111"  when r(15)='1' else
            "1110"  when r(14)='1' else
15          "1101"  when r(13)='1' else
            "1100"  when r(12)='1' else
            "1011"  when r(11)='1' else
            "1010"  when r(10)='1' else
            "1001"  when r(9)='1'  else
20          "1000"  when r(8)='1'  else
            "0111"  when r(7)='1'  else
            "0110"  when r(6)='1'  else
            "0101"  when r(5)='1'  else
            "0100"  when r(4)='1'  else
25          "0011"  when r(3)='1'  else
            "0010"  when r(2)='1'  else
            "0001"  when r(1)='1'  else
            "0000";
    active <= r(15) or r(14) or r(13) or r(12) or
30            r(11) or r(10) or r(9) or r(8) or
              r(7) or r(6) or r(5) or r(4) or
              r(3) or r(2) or r(1) or r(0);
end cascade_arch;
```

The diagram of the code segment is shown in Figure 7.9, which consists of a chain of 15 2-to-1 multiplexers.

To develop a tree design, we start with smaller priority encoders and then rearrange them to the desired layout. Design in this example uses a 4-to-2 priority encoder. The function table and block diagram of a 4-to-2 decoder are shown in Figure 7.10(a). The block diagram of the 16-to-4 tree priority encoder is shown in Figure 7.10(b).

The basic skeleton consists of a two-level tree. The 16 input requests are divided into four groups and fed to four 4-to-2 priority encoders in the first level. Each 4-to-2 priority encoder performs two functions. First, they generate active signals, `act1`, `act2`, `act3` and `act4`, to indicate whether a request occurs in a particular group. Each active signal can be interpreted as the request signal of that particular group. Second, due to the clever arrangement of input connection, their output codes, `code_g3`, `code_g2`, `code_g1` and `code_g0`, form the two LSBs of the final 4-bit code. For example, if the highest-priority request is `r(9)`, its code is "1001". The `r(9)` signal is connected to the second 4-to-2 priority encoder in the first level and its output , `code_g2`, is "01", which is the two LSBs of "1001".

There is only one 4-to-2 priority encoder in the second level. Its inputs are the four "group request" signals from the first level. The output, `code_msb`, is the code of the group with the highest-priority request, which forms the two MSBs of the 4-bit `code` signal. We also need a 4-to-1 multiplexer in the second level. The `code_msb` signal is used to select and route the 2 LSBs from the proper group to final output.

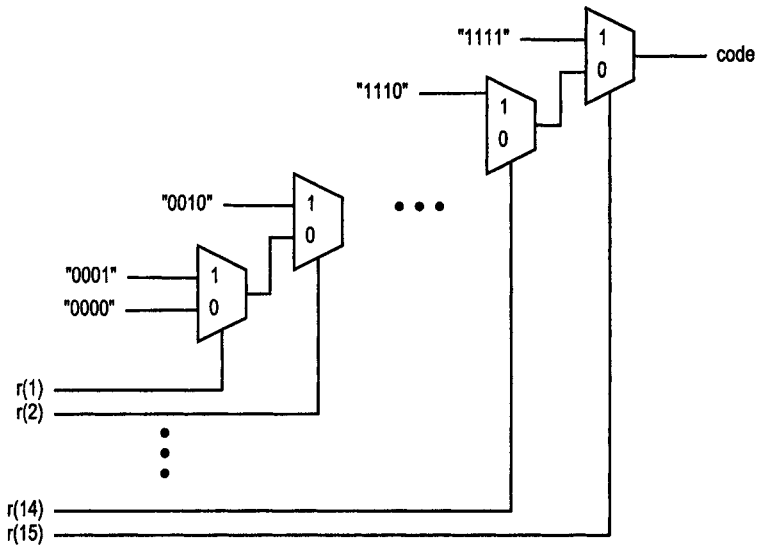


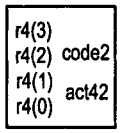
Figure 7.9 Cascading priority encoder.

Since a 4-to-2 priority encoder is used repeatedly, we use component instantiation in the code. The 4-to-2 priority encoder is coded as a regular cascading design, as shown in Listing 7.25.

Listing 7.25 4-to-2 priority encoder

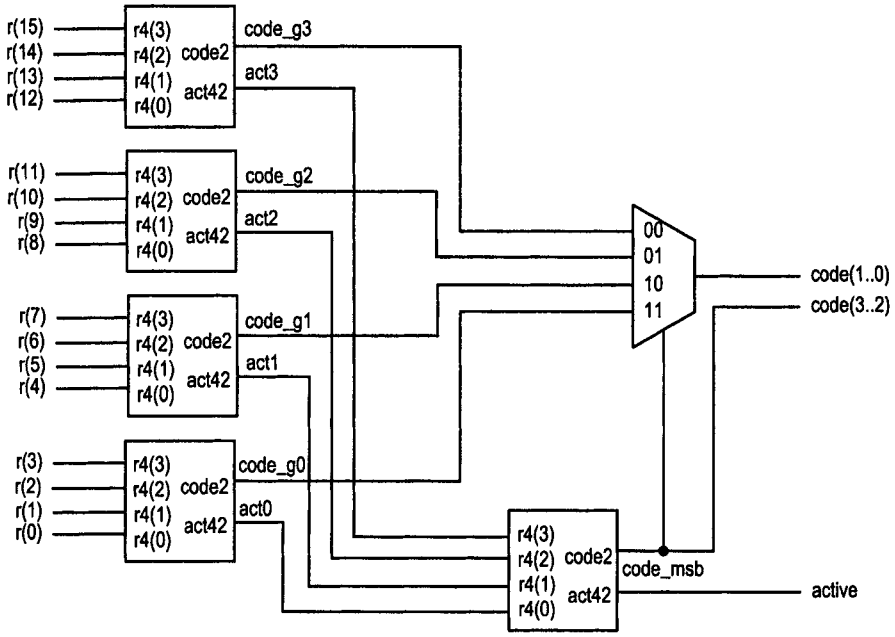
```
library ieee;
use ieee.std_logic_1164.all;
entity prio42 is
   port(
      r4: in std_logic_vector(3 downto 0);
      code2: out std_logic_vector(1 downto 0);
      act42: out std_logic
   );
end prio42;

architecture cascade_arch of prio42 is
begin
   code2 <= "11"  when r4(3)='1'  else
            "10"  when r4(2)='1'  else
            "01"  when r4(1)='1'  else
            "00";
   act42 <= r4(3) or r4(2) or r4(1) or r4(0);
end cascade_arch;
```

The VHDL code for the tree design is shown in Listing 7.26, which basically follows the diagram of Figure 7.10(b). The code uses VHDL component instantiation, which is briefly reviewed in Section 2.2.2 and discussed in Chapter 13.



r4	code2	act42
1- - -	11	1
01- -	10	1
001-	01	1
0001	00	1
0000	00	0

(a) 4-to-2 priority encoder



(b) 16-to-4 priority encoder using 4-to-2 priority encoders

Figure 7.10 Tree priority encoder.

Listing 7.26 16-to-4 priority encoder

```
architecture tree_arch of prio_encoder is
   component prio42 is
      port(
         r4: in std_logic_vector(3 downto 0);
         code2: out std_logic_vector(1 downto 0);
         act42: out std_logic
      );
   end component;
   signal code_g3, code_g2, code_g1, code_g0, code_msb:
      std_logic_vector(1 downto 0);
   signal tmp: std_logic_vector(3 downto 0);
   signal act3, act2, act1, act0: std_logic;
begin
   -- four 1st-stage 4-to-2 priority encoders
   unit_level_0_0: prio42
      port map(r4=>r(3 downto 0), code2=>code_g0,
               act42=>act0);
   unit_level_0_1: prio42
      port map(r4=>r(7 downto 4), code2=>code_g1,
               act42=>act1);
   unit_level_0_2: prio42
      port map(r4=>r(11 downto 8), code2=>code_g2,
               act42=>act2);
   unit_level_0_3: prio42
      port map(r4=>r(15 downto 12), code2=>code_g3,
               act42=>act3);
   -- 2nd stage 4-to-2 priority encoder
   tmp <= act3 & act2 & act1 & act0;
   unit_level_2: prio42
      port map(r4=>tmp, code2=>code(3 downto 2),
               act42=>active);
   -- 2 MSBs of code
   code(3 downto 2) <= code_msb;
   -- 2 LSBs of code
   with code_msb select
      code(1 downto 0) <= code_g3 when "11",
                          code_g2 when "10",
                          code_g1 when "01",
                          code_g0 when others;
end tree_arch;
```

Now let us analyze the critical path of two designs. The critical path of the first cascading design consists of fifteen 2-to-1 multiplexers. The critical path of the tree design consists of two 4-to-2 priority encoders plus one 4-to-1 multiplexer. Since the 4-to-2 priority encoder uses the regular cascading design, it is constructed by three 2-to-1 multiplexers. Thus, the critical path of the tree design consists of six 2-to-1 multiplexers and one 4-to-1 multiplexer. It is much shorter than that of the cascading design.

Although software can perform a certain degree of optimization during synthesis, the optimization tends to be local and a good initial description can make a significant impact on the final implementation. This is especially true as the number of input requests increases. We can further refine the design by utilizing a tree of 2-to-1 priority encoders inside the

4-to-2 priority encoder. The 16-to-4 priority encoder now becomes a tree consisting of four levels of 2-to-1 priority encoders.

A major drawback of the tree design is the code complexity. Revising the code for different input sizes is very involved. An alternative scalable design is discussed in Chapter 15.

7.4.4 Barrel shifter revisited

We discussed the design of a barrel shifter in Section 7.3.5. This design suffers several problems, and an alternative is developed in this section. We first examine the design of an 8-bit rotate-right circuit and then extend it to the complete three-function circuit.

In Section 7.3.5, the rotating circuit is translated directly from the function table and coded by a selected signal assignment statement. The code is repeated in Listing 7.27.

Listing 7.27 Single-level rotate-right circuit

```
library ieee;
use ieee.std_logic_1164.all;
entity rotate_right is
   port(
      a: in std_logic_vector(7 downto 0);
      amt: in std_logic_vector(2 downto 0);
      y: out std_logic_vector(7 downto 0)
   );
end rotate_right;

architecture direct_arch of rotate_right is
begin
   with amt select
       y<= a                                  when "000",
           a(0) & a(7 downto 1)               when "001",
           a(1 downto 0) & a(7 downto 2)      when "010",
           a(2 downto 0) & a(7 downto 3)      when "011",
           a(3 downto 0) & a(7 downto 4)      when "100",
           a(4 downto 0) & a(7 downto 5)      when "101",
           a(5 downto 0) & a(7 downto 6)      when "110",
           a(6 downto 0) & a(7) when others; -- 111
end direct_arch;
```

This code implies an 8-bit 8-to-1 multiplexer circuit. In actual implementation the 8-bit multiplexer is composed of eight 1-bit 8-to-1 multiplexers, as shown in Figure 7.11.

Although the conceptual diagram seems to be all right, this approach suffers some subtle problems. First, a wide multiplexer cannot be effectively mapped to certain device technologies. Second, since an input data bit is routed to all multiplexers, the connection wires grow on the order of $O(n^2)$. The wiring area becomes congested as the number of inputs grows. Finally, the basic layout of this circuit is a single narrow strip, as in Figure 7.11. This makes placing and routing more difficult.

An alternative design is to do the rotating in levels, as shown in Figure 7.12(a). In each level, a bit of the `amt` signal indicates whether the input is passed directly to the output or rotated by a fixed amount. Bits 0, 1 and 2 of the `amt` signal control the routing in levels 0, 1 and 2 respectively. The amounts are different in each level, which are the 2^0, 2^1 and 2^2 positions. After passing three levels, the total number of positions rotated is the summation

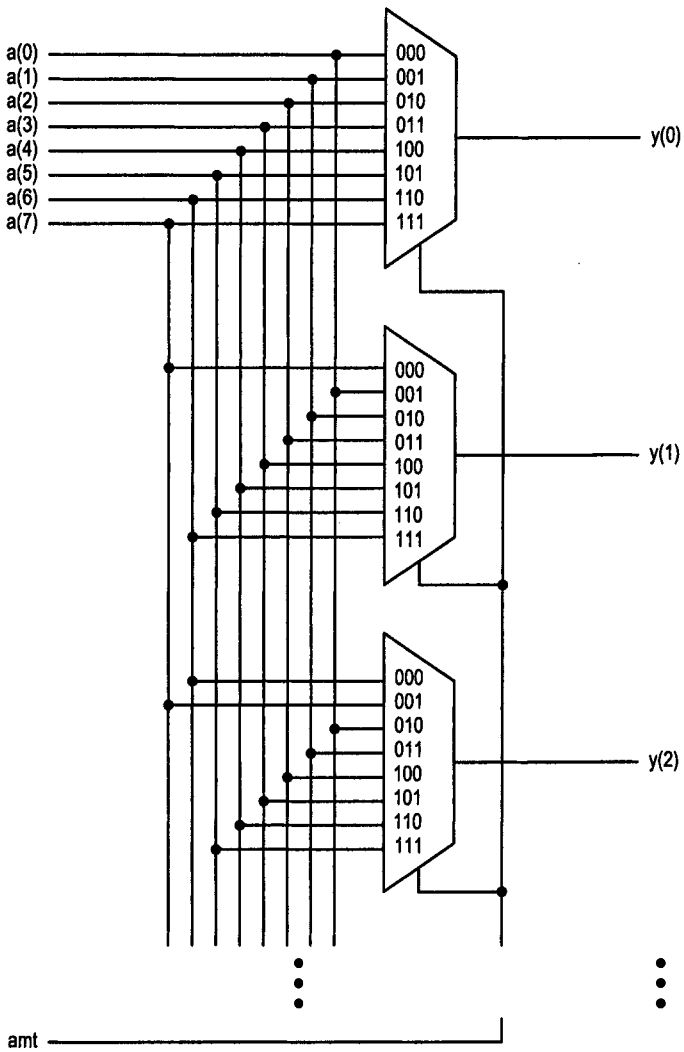
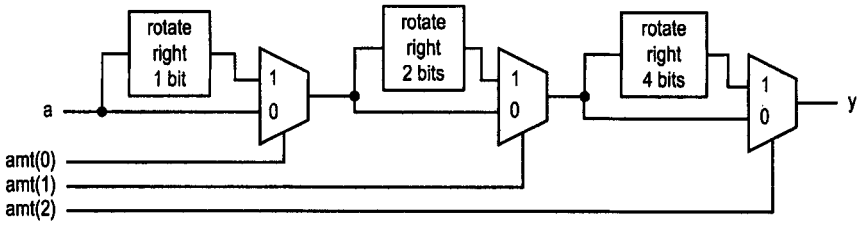
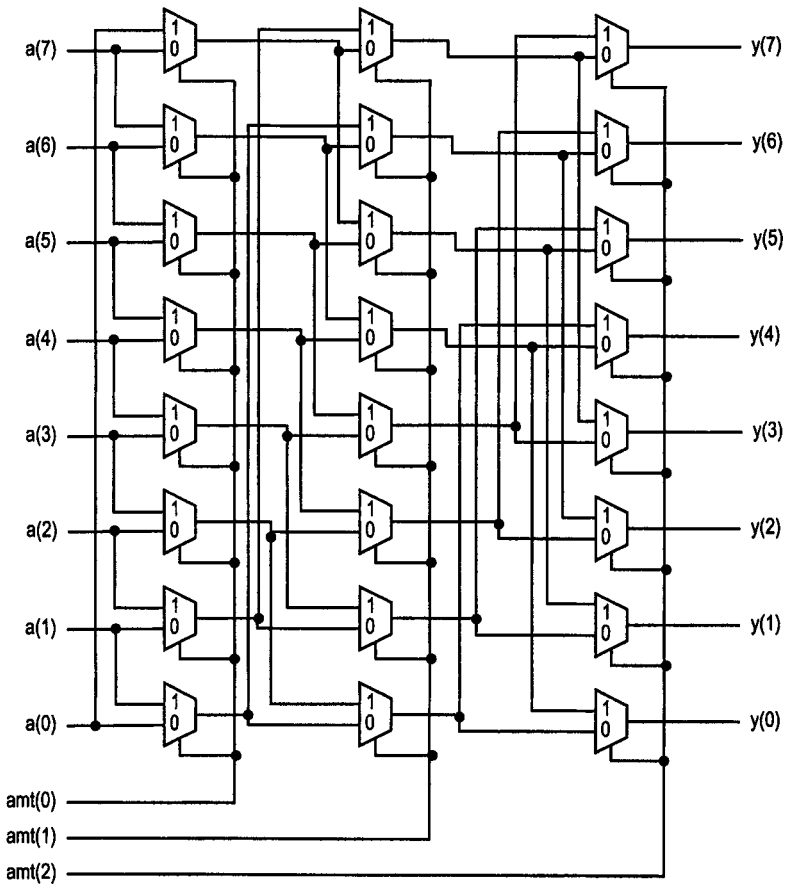


Figure 7.11 Barrel shifter using a single level of 8-to-1 multiplexers.



(a) Block diagram



(b) Detailed diagram

Figure 7.12 Barrel shifter using three levels of 2-to-1 multiplexers.

of positions rotated in each level, which is amt(2)*2^2 + amt(1)*2^1 + amt(0)*2^0. The VHDL code of this revised design is shown in Listing 7.28.

Listing 7.28 Multilevel rotate-right circuit

```
architecture multi_level_arch of rotate_right is
   signal le0_out, le1_out, le2_out:
      std_logic_vector(7 downto 0);
begin
   -- level 0, shift 0 or 1 bit
   le0_out <= a(0) & a(7 downto 1) when amt(0)='1' else
              a;
   -- level 1, shift 0 or 2 bits
   le1_out <=
     le0_out(1 downto 0) & le0_out(7 downto 2)
        when amt(1)='1' else
     le0_out;
   -- level 2, shift 0 or 4 bits
   le2_out <=
     le1_out(3 downto 0) & le1_out(7 downto 4)
        when amt(2)='1' else
     le1_out;
   -- output
   y <= le2_out;
end multi_level_arch;
```

A more detailed diagram of this design is shown in Figure 7.12(b). Note that rotating a fixed amount involves only signal routing and requires no physical components.

Comparing the two designs is more subtle. The first design needs eight 8-to-1 multiplexers and its critical path is the same as the critical path of an 8-to-1 multiplexer. The multilevel design needs eight 2-to-1 multiplexers at each level, and thus a total of twenty-four 2-to-1 multiplexers. Its critical path consists of three levels of 2-to-1 multiplexers. The implementation of these multiplexers is technology dependent and there is no clear-cut answer on circuit size and propagation delay. The additional wiring area and delay of the first design further complicates the comparison. However, when the input becomes large, the wiring and routing will become more problematic in the first design. The regular interconnection pattern of the multilevel design can scale better and thus should be the preferred choice. The VHDL code of multilevel design is also easier to scale. The amount of revision is on the order of $O(\log_2 n)$ rather than $O(n)$, as in the first design.

To extend the rotate-right circuit to incorporate the additional logic shift-right and arithmetic shift-right functions, we can apply the preprocessing idea from Section 7.3.5. Since there are three levels in the new design, preprocessing has to be performed at each level. The revised VHDL code is given in Listing 7.29.

Listing 7.29 Multilevel description of a three-function barrel shifter

```
architecture multi_level_arch of shift3mode is
   signal le0_out, le1_out, le2_out:
      std_logic_vector(7 downto 0);
   signal le0_sin: std_logic;
   signal le1_sin: std_logic_vector(1 downto 0);
   signal le2_sin: std_logic_vector(3 downto 0);
begin
   -- level 0, shift 0 or 1 bit
```

```
    with lar select
10     le0_sin <= '0'    when "00",
                  a(7)   when "01",
                  a(0)   when others;
    le0_out <= le0_sin & a(7 downto 1) when amt(0)='1' else
               a;
15  -- level 1, shift 0 or 2 bits
    with lar select
       le1_sin <=
          "00"                    when "00",
          (others => le0_out(7))  when "01",
20        le0_out(1 downto 0)     when others;
    le1_out <= le1_sin & le0_out(7 downto 2)
                  when amt(1)='1' else
               le0_out;
    -- level 2, shift 0 or 4 bits
25  with lar select
       le2_sin <=
          "0000"                  when "00",
          (others => le1_out(7))  when "01",
          le1_out(3 downto 0)     when others;
30  le2_out <= le2_sin & le1_out(7 downto 4)
                  when amt(2)='1' else
               le1_out;
    -- output
    y <= le2_out;
35 end multi_level_arch ;
```

The preprocessing utilizes three 3-to-1 multiplexers, whose widths are 1, 2 and 4 bits respectively, and their overall complexity is similar to the 8-bit 3-to-1 multiplexer used in Section 7.3.5.

7.5 GENERAL CIRCUITS

The examples of previous sections are focused on specific aspects of design and VHDL coding. Several general design examples are presented in this section.

7.5.1 Gray code incrementor

The *Gray code* is a special kind of code in that only a single bit changes between any two successive code words. It minimizes the number of transitions when a signal switches between successive words. A 4-bit Gray code and its corresponding binary code are shown in Table 7.1. A Gray code incrementor is a circuit that generates the next word in Gray code. The function table of a 4-bit Gray code incrementor is shown in Table 7.2. A straightforward design is simply to translate this table into a selected signal assignment statement, as in Listing 7.30.

Listing 7.30 Initial description of a Gray code incrementor

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

Table 7.1 4-bit Gray code

Binary code $b_3b_2b_1b_0$	**Gray code** $g_3g_2g_1g_0$
0000	0000
0001	0001
0010	0011
0011	0010
0100	0110
0101	0111
0110	0101
0111	0100
1000	1100
1001	1101
1010	1111
1011	1110
1100	1010
1101	1011
1110	1001
1111	1000

Table 7.2 Function table of a 4-bit Gray code incrementor

Gray code	**Incremented Gray code**
0000	0001
0001	0011
0011	0010
0010	0110
0110	0111
0111	0101
0101	0100
0100	1100
1100	1101
1101	1111
1111	1110
1110	1010
1010	1011
1011	1001
1001	1000
1000	0000

```
entity g_inc is
    port(
        g: in std_logic_vector(3 downto 0);
        g1: out std_logic_vector(3 downto 0)
    );
end g_inc ;

architecture table_arch of g_inc is
begin
    with g select
        g1 <= "0001" when "0000",
              "0011" when "0001",
              "0010" when "0011",
              "0110" when "0010",
              "0111" when "0110",
              "0101" when "0111",
              "0100" when "0101",
              "1100" when "0100",
              "1101" when "1100",
              "1111" when "1101",
              "1110" when "1111",
              "1010" when "1110",
              "1011" when "1010",
              "1001" when "1011",
              "1000" when "1001",
              "0000" when others; -- "1000"
end table_arch;
```

Although the VHDL code is simple, it is not scalable because the needed revision is on the order of $O(2^n)$. Unfortunately, there is no easy algorithm to derive the next Gray code word directly. Since an algorithm exists for conversion between Gray code and binary code, one possible approach is to derive it indirectly by using a binary incrementor. This design includes three stages:

1. Convert a Gray code word to the corresponding binary word.
2. Increment the binary word.
3. Convert the result back to the Gray code word.

The binary-to-Gray conversion algorithm is based on the following observation: the ith bit (i.e., g_i) of the Gray code word is '1' if the ith bit and (i+1)th bit (i.e., b_i and b_{i+1}) of the corresponding binary word are different. This observation can be translated into a logic equation:

$$g_i = b_i \oplus b_{i+1}$$

We can verify this equation by using the 4-bit code of Table 7.1:

$$g_3 = b_3 \oplus 0 = b_3$$
$$g_2 = b_2 \oplus b_3$$
$$g_1 = b_1 \oplus b_2$$
$$g_0 = b_0 \oplus b_1$$

The equation for Gray-to-binary conversion can be obtained by manipulating the previous equation:

$$b_i = g_i \oplus b_{i+1}$$

We can also expand b_{i+1} on the right-hand side recursively. For example, a 4-bit code can be expressed as

$$\begin{aligned} b_3 &= g_3 \oplus 0 = g_3 \\ b_2 &= g_2 \oplus b_3 = g_2 \oplus g_3 \\ b_1 &= g_1 \oplus b_2 = g_1 \oplus g_2 \oplus g_3 \\ b_0 &= g_0 \oplus b_1 = g_0 \oplus g_1 \oplus g_2 \oplus g_3 \end{aligned}$$

Once we know the conversion algorithm, we can derive the VHDL code. Note that the equations for the Gray-to-binary conversion are very similar to the reduced-xor-vector function discussed in Section 7.4.2. The VHDL code of the new design is shown in Listing 7.31. We use the compact vector form, similar to that in the `shared_compact_arch` architecture of Listing 7.21, for Gray-to-binary and binary-to-Gray code conversions.

Listing 7.31 Compact description of a Gray code incrementor

```
architecture compact_arch of g_inc is
   constant WIDTH: integer := 4;
   signal b, b1: std_logic_vector(WIDTH-1 downto 0);
begin
   -- Gray to binary
   b <= g xor ('0' & b(WIDTH-1 downto 1));
   -- binary increment
   b1 <= std_logic_vector((unsigned(b)) + 1);
   -- binary to Gray
   g1<= b1 xor ('0' & b1(WIDTH-1 downto 1));
end compact_arch;
```

The new code is independent of the input size and the revision is on the order of $O(1)$. Since each part can easily be identified, this design allows us to utilize the alternative implementation for the adder and Gray-to-binary circuit. If performance is an issue, we can replace them with faster but larger circuits.

7.5.2 Programmable priority encoder

In a regular priority encoder, the order of priority for each request is fixed. For example, the order of eight requests, `r(7)`, ..., `r(0)`, is normally `r(7)`, `r(6)`, ..., `r(1)` and `r(0)`. Some applications need to dynamically change the priority of a request to give fair access to each request. In this subsection, we consider a programmable 8-to-3 priority encoder in which the priority can be assigned in a wrapped-around fashion. In addition to the eight regular request signals, the circuit also has a 3-bit control signal, `c`, which specifies the request that has the highest priority. For example, if `c` is "011", `r(3)` has the highest priority and the order of the requests is `r(3)`, `r(2)`, `r(1)`, `r(0)`, `r(7)`, ..., `r(4)`. A brute-force design is to utilize eight regular priority encoders and one 8-to-1 multiplexer. Each priority encoder has a fixed request order, and the multiplexer passes the desired code to the output. While this design is straightforward, it is not very efficient.

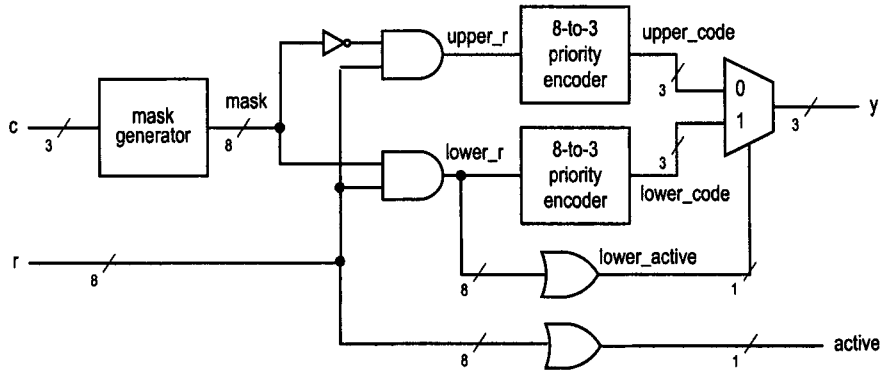


Figure 7.13 Block diagram of a programmable priority encoder.

A better design is shown in Figure 7.13. This design first uses the c signal to generate *two* 8-bit masks, which are used to clear the upper and lower parts of the requests. For example, if c is "011", the lower mask is "00000111" and the upper mask is "11111000", the inverse of the lower mask. We apply the two masks to the original requests and obtain two new masked requests, in which the lower and upper parts are cleared. For example, if a request is "11011011", the two masked requests will be "11011000" and "00000011". If the active signal is asserted in the lower group, it means that there is a request from that group and its code will be routed to the final output. Otherwise, the code from the upper priority encoder will be routed to the output. If we continue the previous example, the codes from the upper and lower priority encoders are "111" and "001", and since there is a request from the lower group, "001" will be routed to the final output. The VHDL code describing this design is shown in Listing 7.32.

Listing 7.32 Programmable priority encoder

```
library ieee;
use ieee.std_logic_1164.all;
entity fair_prio_encoder is
   port(
      r: in std_logic_vector(7 downto 0);
      c: in std_logic_vector(2 downto 0);
      code: out std_logic_vector(2 downto 0);
      active: out std_logic
   );
end fair_prio_encoder;

architecture arch of fair_prio_encoder is
   signal mask, lower_r, upper_r:
      std_logic_vector(7 downto 0);
   signal lower_code, upper_code:
      std_logic_vector(2 downto 0);
   signal lower_active: std_logic;
begin
   with c select
      mask <= "00000001" when "000",
              "00000011" when "001",
```

```
                "00000111" when "010",
                "00001111" when "011",
                "00011111" when "100",
                "00111111" when "101",
                "01111111" when "110",
                "11111111" when others;
    lower_r <= r and mask;
    upper_r <= r and (not mask);
    lower_code <= "111"  when lower_r(7)='1' else
                  "110"  when lower_r(6)='1' else
                  "101"  when lower_r(5)='1' else
                  "100"  when lower_r(4)='1' else
                  "011"  when lower_r(3)='1' else
                  "010"  when lower_r(2)='1' else
                  "001"  when lower_r(1)='1' else
                  "000";
    upper_code <= "111"  when upper_r(7)='1' else
                  "110"  when upper_r(6)='1' else
                  "101"  when upper_r(5)='1' else
                  "100"  when upper_r(4)='1' else
                  "011"  when upper_r(3)='1' else
                  "010"  when upper_r(2)='1' else
                  "001"  when upper_r(1)='1' else
                  "000";
    lower_active <= lower_r(7) or lower_r(6) or lower_r(5) or
                    lower_r(4) or lower_r(3) or lower_r(2) or
                    lower_r(1) or lower_r(0);
    code <= lower_code when lower_active='1' else
            upper_code;
    active <= r(7) or r(6) or r(5) or r(4) or
              r(3) or r(2) or r(1) or r(0);
end arch;
```

The VHDL code is much more efficient than the first design.

7.5.3 Signed addition with status

The definition of the VHDL addition operator is very simple. It takes two operands and returns the summation. In a complex digital system, such as a processor, adders frequently need additional status signals and carry signals. Status signals show various conditions of an addition operation, including zero, sign and overflow. Zero status indicates whether the result is zero, sign status indicates whether the result is a positive or negative number, and overflow status indicates whether overflow occurs during operation. Carry signals pass information between successive additions. For example, if we want to construct a 64-bit adder by using 8-bit adders, we have to utilize the carry signals to convey the relevant carry information. Carry signals include the carry-in signal, which is an input that comes from the previous stage, and the carry-out signal, which is an output signal to be passed to the next stage. We consider the addition of two signed integers in this subsection.

The derivation of status signals is trickier than it first appears because of the overflow condition. Overflow affects the determination of sign and zero status and thus must be determined first. Our derivation of overflow condition is based on the following observations:

- If the two operands have different signs, overflow can never occur since the addition of a positive number and a negative number will always decrease the magnitude.
- If the two operands and the result have the same sign, overflow does not occur since the result is still within the range.
- If the two operands have the same sign but the result has a different sign, overflow occurs. The sign change indicates that the result goes beyond the positive or negative boundary and thus is beyond the range. We can verify this by checking the binary wheel of Figure 7.6.

Let the sign bits of two operands and summation be s_a, s_b and s_s respectively. We can translate our observation into the following logic expression:

$$overflow = (s_a \cdot s_b \cdot s_s') + (s_a' \cdot s_b' \cdot s_s)$$

Once we know the overflow condition, we can determine the zero condition and the sign. Because of the potential of overflow, the addition result may not be 0 even if the summation output is 0. For example, if we add two 4-bit inputs, "1000" and "1000", the summation output is "0000" because of overflow. Thus, the zero condition should be asserted only if the summation output is 0 and there is no overflow.

From our observation on overflow, it is clear that the sign bit of the summation output is not necessarily the sign of a real addition result. In the example above, the sign bit of summation "0000" is '0' while the addition result should be negative. Thus, the sign bit of the addition result is the same as the sign bit of the summation output only if no overflow occurs. It should be inverted otherwise.

Carry signals can be handled by using two extra bits in the internal signals. One bit will be appended to the left to incorporate the carry-out signal. The other bit will be appended to the right to inject the carry-in signal, as explained in Section 7.3.1. The complete VHDL code is shown in Listing 7.33.

Listing 7.33 Signed addition with status

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity adder_status is
   port(
      a,b: in std_logic_vector(7 downto 0);
      cin: in std_logic;
      sum: out  std_logic_vector(7 downto 0);
      cout, zero, overflow, sign: out std_logic
   );
end adder_status;

architecture arch of adder_status is
   signal a_ext, b_ext, sum_ext: signed(9 downto 0);
   signal ovf: std_logic;
   alias sign_a: std_logic is a_ext(8);
   alias sign_b: std_logic is b_ext(8);
   alias sign_s: std_logic is sum_ext(8);
begin
   a_ext <= signed('0' & a & '1');
   b_ext <= signed('0' & b & cin);
   sum_ext <= a_ext + b_ext;
```

					a_3	a_2	a_1	a_0	multiplicand
$\times$					b_3	b_2	b_1	b_0	multiplier
					a_3b_0	a_2b_0	a_1b_0	a_0b_0	
				a_3b_1	a_2b_1	a_1b_1	a_0b_1		
			a_3b_2	a_2b_2	a_1b_2	a_0b_2			
$+$		a_3b_3	a_2b_3	a_1b_3	a_0b_3				
	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0	product

Figure 7.14 Multiplication as a summation of a_ib_j terms.

```
    ovf <= (sign_a and sign_b and (not sign_s)) or
           ((not sign_a) and (not sign_b) and sign_s);
25  cout <= sum_ext(9);
    sign <= sum_ext(8) when ovf='0' else
            not sum_ext(8);
    zero <= '1' when (sum_ext(8 downto 1)=0 and ovf='0') else
            '0';
30  overflow <= ovf;
    sum <= std_logic_vector(sum_ext(8 downto 1));
end arch;
```

7.5.4 Combinational adder-based multiplier

A multiplier is a fairly complicated circuit. The synthesis of the VHDL multiplication operator depends on the individual software and the underlying target technology, and cannot always be done automatically. In this example, we study a simple, portable, though not optimal, combinational adder-based multiplier.

The multiplier is based on the algorithm we learned in elementary school. The multiplication of two 4-bit numbers is illustrated in Figure 7.14, which are aligned in a specific two-dimensional pattern. This algorithm includes three tasks:

1. Multiply the digits of the multiplier (b_3, b_2, b_1 and b_0 of Figure 7.14) by the multiplicand (A of Figure 7.14) one at a time to obtain $b_3{*}A$, $b_2{*}A$, $b_1{*}A$ and $b_0{*}A$. Since b_i is a binary digit, it can only be 0 or 1, and thus $b_i{*}A$ can only be 0 or A. The $b_i{*}A$ operation becomes bitwise and operation of b_i and the digits of A; that is,

$$b_i{*}A = (a_3{\cdot}b_i,\ a_2{\cdot}b_i,\ a_1{\cdot}b_i,\ a_0{\cdot}b_i)$$

2. Shift $b_i{*}A$ to left i positions.
3. Add the shifted $b_i{*}A$ terms to obtain the final product.

The VHDL code of an 8-bit multiplier based on this algorithm is shown in Listing 7.34. We first construct an 8-bit vector, $b_ib_ib_ib_ib_ib_ib_ib_i$, for each b_i to facilitate the bitwise and operation. The vector is used to generate shifted $b_i{*}A$ terms. Note that padding 0's are inserted around $b_i{*}A$ to form a 16-bit signal. The shifted $b_i{*}A$ terms are then summated by seven adders, which are arranged as a tree to increase performance, to obtain the final result.

Listing 7.34 Initial description of an adder-based multiplier

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity mult8 is
   port(
      a, b: in std_logic_vector(7 downto 0);
      y: out std_logic_vector(15 downto 0)
   );
end mult8;

architecture comb1_arch of mult8 is
   constant WIDTH: integer:=8;
   signal au, bv0, bv1, bv2, bv3, bv4, bv5, bv6, bv7:
      unsigned(WIDTH-1 downto 0);
   signal p0,p1,p2,p3,p4,p5,p6,p7,prod:
      unsigned(2*WIDTH-1 downto 0);
begin
   au <= unsigned(a);
   bv0 <= (others=>b(0));
   bv1 <= (others=>b(1));
   bv2 <= (others=>b(2));
   bv3 <= (others=>b(3));
   bv4 <= (others=>b(4));
   bv5 <= (others=>b(5));
   bv6 <= (others=>b(6));
   bv7 <= (others=>b(7));
   p0 <="00000000" & (bv0 and au);
   p1 <="0000000" & (bv1 and au) & "0";
   p2 <="000000" & (bv2 and au) & "00";
   p3 <="00000" & (bv3 and au) & "000";
   p4 <="0000" & (bv4 and au) & "0000";
   p5 <="000" & (bv5 and au) & "00000";
   p6 <="00" & (bv6 and au) & "000000";
   p7 <="0" & (bv7 and au) & "0000000";
   prod <= ((p0+p1)+(p2+p3))+((p4+p5)+(p6+p7));
   y <= std_logic_vector(prod);
end comb1_arch;
```

Adders are the major components of this design. For a circuit with an n-bit multiplicand and an n-bit multiplier, the product has $2n$ bits. The shifted b_i*A has to be extended to $2n$ bits, and thus the design needs $n-1$ $2n$-bit adders. The code can easily be expanded for a larger multiplier, and the needed revision is on the order of $O(n)$.

One way to reduce the size of this circuit is to add shifted b_i*A terms in sequence. This reduces the width of the adder to $n+1$ bits. Operation of the new design is illustrated in Figure 7.15.

We first obtain b_0*A and form the first partial product pp0. To accommodate the carry-out of future addition, one extra bit is appended to the left of b_0*A. Note that the LSB of prod (i.e., prod(0)) is the same as the LSB of pp0 (i.e., pp0(0)), and the pp0(0) bit has no effect on the remaining addition operations. We need only add the upper bits of the pp0 to b_1*A to form the next partial sum, pp1. Note that prod(1) is same as pp1(0), and pp1(0) has no effect on the remaining additions. We can repeat the process to obtain other

				a_3	a_2	a_1	a_0	multiplicand
$\times$				b_3	b_2	b_1	b_0	multiplier
				a_3b_0	a_2b_0	a_1b_0	a_0b_0	
				$pp0_4$	$pp0_3$	$pp0_2$	$pp0_1$	$pp0_0$	partial product $pp0$
			$+$	a_3b_1	a_2b_1	a_1b_1	a_0b_1		
			$pp1_4$	$pp1_3$	$pp1_2$	$pp1_1$	$pp1_0$		partial product $pp1$
		$+$	a_3b_2	a_2b_2	a_1b_2	a_0b_2			
		$pp2_4$	$pp2_3$	$pp2_2$	$pp2_1$	$pp2_0$			partial product $pp2$
	$+$	a_3b_3	a_2b_3	a_1b_3	a_0b_3				
	$pp3_4$	$pp3_3$	$pp3_2$	$pp3_1$	$pp3_0$				partial product $pp3$
	$pp3_4$	$pp3_3$	$pp3_2$	$pp3_1$	$pp3_0$	$pp2_0$	$pp1_0$	$pp0_0$	product $prod$

Figure 7.15 Multiplication as successive summation.

partial sums in sequence. This design still needs $n - 1$ adders, but the width of the adders is decreased from $2n$ to $n + 1$, about one half of the original size. The VHDL code of an 8-bit multiplier based on this algorithm is shown in Listing 7.35.

Listing 7.35 More efficient description of an adder-based multiplier

```
architecture comb2_arch of mult8 is
   constant WIDTH: integer:=8;
   signal au,bv0,bv1,bv2,bv3,bv4,bv5,bv6,bv7:
      unsigned(WIDTH-1 downto 0);
   signal pp0,pp1,pp2,pp3,pp4,pp5,pp6,pp7:
      unsigned(WIDTH downto 0);
   signal prod: unsigned(2*WIDTH-1 downto 0);
begin
   au <= unsigned(a);
   bv0 <= (others=>b(0));
   bv1 <= (others=>b(1));
   bv2 <= (others=>b(2));
   bv3 <= (others=>b(3));
   bv4 <= (others=>b(4));
   bv5 <= (others=>b(5));
   bv6 <= (others=>b(6));
   bv7 <= (others=>b(7));
   pp0 <= "0" & (bv0 and au);
   pp1 <= ("0" & pp0(WIDTH downto 1)) + ("0" & (bv1 and au));
   pp2 <= ("0" & pp1(WIDTH downto 1)) + ("0" & (bv2 and au));
   pp3 <= ("0" & pp2(WIDTH downto 1)) + ("0" & (bv3 and au));
   pp4 <= ("0" & pp3(WIDTH downto 1)) + ("0" & (bv4 and au));
   pp5 <= ("0" & pp4(WIDTH downto 1)) + ("0" & (bv5 and au));
   pp6 <= ("0" & pp5(WIDTH downto 1)) + ("0" & (bv6 and au));
   pp7 <= ("0" & pp6(WIDTH downto 1)) + ("0" & (bv7 and au));
   prod <= pp7 & pp6(0) & pp5(0) & pp4(0) & pp3(0) &
           pp2(0) & pp1(0) & pp0(0);
```

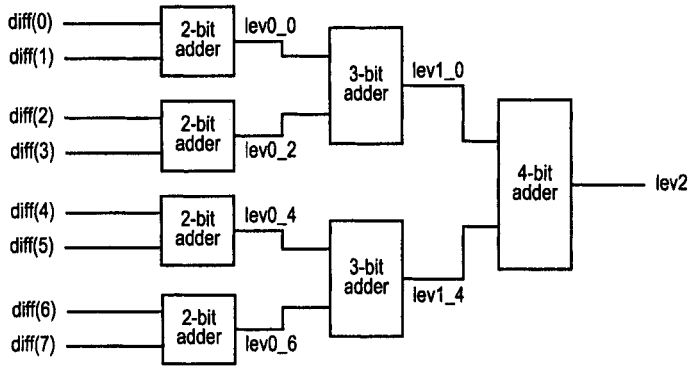


Figure 7.16 Block diagram of a population counter.

```
    y <= std_logic_vector(prod);
end comb2_arch;
```

7.5.5 Hamming distance circuit

A Hamming distance of two words is the number of bit positions in which the two words differ. For example, the Hamming distance of two 8-bit words "00010011" and "10010010" is 2 since the bits at position 0 (LSB) and position 7 (MSB) are different. The Hamming distance is used in some error correction and data compression applications. This example considers a circuit that calculates the Hamming distance of two 8-bit inputs.

Our design has two basic steps. The first step determines the bits that are different and marks them as '1'. The second step counts the number of 1's in the word, a function known as a *population counter*. For example, consider the inputs "00010011" and "10010010". The first step returns "10000001" since the bits at positions 0 and 7 are different, and the second step returns 2 since there are two 1's in the word.

We can implement the first step by using a simple bitwise xor operation. Recall that the 1-bit xor function returns '1' only if the input is "01" or "10". It can be interpreted that the function returns '1' if two inputs are different. Thus, after applying a bitwise xor operation, we can mark all the bits that are different.

Design of the population counter is more difficult. Our first design is shown in Figure 7.16. It counts the number of 1's by stages. In the first level of the circuit, we divide the 8 bits into four pairs and add the 1's in each pair. Four 2-bit adders are needed to perform the operation. In the second level, we pair the results and add them again. Two 3-bit adders are needed. The process is repeated one more time in the third level to obtain the final result. The VHDL code is shown in Listing 7.36.

Listing 7.36 Initial description of a Hamming distance circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity hamming is
   port(
      a,b: in std_logic_vector(7 downto 0);
```

```
      y: out std_logic_vector(3 downto 0)
   );
end hamming;

architecture effi_arch of hamming is
   signal diff: unsigned(7 downto 0);
   signal lev0_0, lev0_2, lev0_4, lev0_6:
      unsigned(1 downto 0);
   signal lev1_0, lev1_4: unsigned(2 downto 0);
   signal lev2: unsigned(3 downto 0);
begin
   diff <= unsigned(a xor b);
   lev0_0 <= ('0' & diff(0)) + ('0' & diff(1));
   lev0_2 <= ('0' & diff(2)) + ('0' & diff(3));
   lev0_4 <= ('0' & diff(4)) + ('0' & diff(5));
   lev0_6 <= ('0' & diff(6)) + ('0' & diff(7));
   lev1_0 <= ('0' & lev0_0) + ('0' & lev0_2);
   lev1_4 <= ('0' & lev0_4) + ('0' & lev0_6);
   lev2 <= ('0' & lev1_0) + ('0' & lev1_4);
   y <= std_logic_vector(lev2);
end effi_arch;
```

Although this population counter design is fairly efficient, the code is somewhat tedious. An alternative design is to use a clever shifting and masking scheme to rearrange the input and utilize a fixed-size 8-bit adder at each level. Assume that the 8-bit input to the population counter is $d_7d_6d_5d_4d_3d_2d_1d_0$. The algorithm is summarized below.

- *Level 0.* We first split and rearrange the original input into two words, $0d_60d_40d_20d_0$ and $0d_70d_50d_30d_1$, and then add them by an 8-bit adder. Assume that the result is $e_7e_6e_5e_4e_3e_2e_1e_0$. Because of the locations of the 0's, this adder performs essentially four 2-bit additions, and e_1e_0, e_3e_2, e_5e_4 and e_7e_6 are $d_0 + d_1$, $d_2 + d_3$, $d_4 + d_5$ and $d_6 + d_7$ respectively.
- *Level 1.* We perform splitting and addition operations similar to those at level 0. However, the input now is split into $00e_5e_400e_1e_0$ and $00e_7e_600e_3e_2$. The result should be in the form of $f_7f_6f_5f_4f_3f_2f_1f_0$. Note that f_7 and f_3 should be 0. The adder actually performs two 3-bit additions, and $f_2f_1f_0$ and $f_6f_5f_4$ are $e_1e_0 + e_3e_2$ and $e_5e_4 + e_7e_6$ respectively.
- *Level 2.* We repeat the same operation except that the input is split into $0000f_3f_2f_1f_0$ and $0000f_7f_6f_5f_4$. The result should be in the form of $0000g_3g_2g_1g_0$.

The VHDL code based on this design is shown in Listing 7.37. The splitting and rearrangement of the input can be done by masking and shifting. For example, the mask in level 0 is `mask0`, "01010101". After performing bitwise and operation of $d_7d_6d_5d_4d_3d_2d_1d_0$ and `mask0`, we obtain the first input word, $0d_60d_40d_20d_0$. We can obtain the second input word in a similar way after first shifting $d_7d_6d_5d_4d_3d_2d_1d_0$ to the right one position. The operations are similar at levels 1 and 2 but have different masking patterns and amounts of shifting.

Listing 7.37 Compact description of a Hamming distance circuit

```
architecture compact_arch of hamming is
   signal diff, lev0, lev1, lev2: unsigned(7 downto 0);
   constant MASK0: unsigned(7 downto 0) := "01010101";
   constant MASK1: unsigned(7 downto 0) := "00110011";
```

```
    constant MASK2: unsigned(7 downto 0) := "00001111";

begin
    diff <= unsigned(a xor b);
    lev0 <= (diff and MASK0) +
            (('0'& diff(7 downto 1)) and MASK0);
    lev1 <= (lev0 and MASK1) +
            (("00" & lev0(7 downto 2)) and MASK1);
    lev2 <= (lev1 and MASK2) +
            (("0000" & lev1(7 downto 4)) and MASK2);
    y <= std_logic_vector(lev2(3 downto 0));
end compact_arch;
```

This design requires more adder bits than the first version. However, its code is more compact and the needed revision is on the order of $O(\log_2 n)$.

7.6 SYNTHESIS GUIDELINES

- Operators can be shared in mutually exclusive branches by proper routing of the input operands and/or result. It is more beneficial for complex operators.
- Many operations have certain common functionality. The hardware resource can be shared by these operations.
- RT-level code can outline the general layout of the circuit. A tree- or rectangle-shaped description can help the synthesis process and placement and routing process to derive a more efficient circuit.

7.7 BIBLIOGRAPHIC NOTES

Developing efficient design and VHDL codes requires the insight and in-depth knowledge of the problem at hand. The digital systems texts, *Digital Design Principles and Practices* by J. F. Wakerly and *Contemporary Logic Design* by R. H. Katz, provide detailed discussion on the construction of many commonly used parts, such as decoders, encoders, comparators and adders. Bibliography in Chapter 15 provides more references on the design and algorithms of multiplier and arithmetic functions.

Problems

7.1 Consider an arithmetic circuit that can perform four operations: `a+b`, `a-b`, `a+1` and `a-1`, where `a` and `b` are 16-bit unsigned numbers and the desired operation is specified by a 2-bit control signal, `ctrl`.

(a) Design the circuit using two adders, one incrementor and one decrementor. Derive the VHDL code.

(b) Design the circuit using only one adder. Derive the VHDL code.

(c) Synthesize the two designs with an ASIC device. Compare the areas and performances.

(d) Synthesize the two designs with an FPGA device. Compare the areas and performances.

7.2 Design a circuit that converts an 8-bit signed input to 8-bit *sign-magnitude* output (where the MSB is the sign bit and the remaining 7 bits are magnitude). Use a minimal number of relational and arithmetic operators in your design. Draw the top-level diagram and derive the VHDL code.

7.3 Extend the dual-mode comparator of Section 7.3.2 to include sign-magnitude mode. Use only one 7-bit comparator in your design. Derive the VHDL code.

7.4 Consider a 16-bit shifting circuit that can perform rotating right or rotating left. Use selected signal assignment statements similar to that in Section 7.3.5 to implement the shifting function.

- **(a)** Design the circuit using one rotate-right circuit, one rotate-left circuit and one 2-to-1 multiplexer to select the desired result. Derive the VHDL code.
- **(b)** Design the circuit using one rotate-right circuit with a pre- and post-processing reversing circuit. The reversing circuit either passes the original input or reverses the input bit-wise (e.g., if a 4-bit input $a_3a_2a_1a_0$ is used, the reversed output becomes $a_0a_1a_2a_3$). Derive the VHDL code.
- **(c)** Draw block diagrams of the two designs and analyze and compare their size and performance.
- **(d)** Synthesize the two designs with an ASIC device. Compare the areas and performances.
- **(e)** Synthesize the two designs with an FPGA device. Compare the areas and performances.

7.5 Consider a reduced-xor-vector function with 16 inputs. Design the circuit using a parallel-prefix structure similar to that of Figure 7.8(b) and derive the VHDL code.

7.6 We can further refine the tree priority encoder in Section 7.4.3 by using 2-to-1 priority encoders.

- **(a)** Design a tree-structured 16-to-4 priority encoder using 2-to-1 priority encoders. The design should have four levels. Draw the block diagram and derive the VHDL code accordingly.
- **(b)** Synthesize the new design and the two designs in Section 7.4.3 with an ASIC device. Compare the areas and performances.
- **(c)** Synthesize the new design and the two designs in Section 7.4.3 with an FPGA device. Compare the areas and performances.

7.7 A leading zero counting circuit counts the number of consecutive 0's of an input. Consider a circuit with a 16-bit input.

- **(a)** Design the circuit using one conditional signal assignment statement and derive the VHDL code.
- **(b)** Derive a smaller 4-bit leading-zero counting circuit first. Design a 16-bit treelike leading-zero counting circuit using 4-bit counting circuits. Derive the VHDL code.
- **(c)** Synthesize the two designs with an ASIC device. Compare the areas and performances.
- **(d)** Synthesize the two designs with an FPGA device. Compare the areas and performances.

7.8 Design a 16-bit rotate-left shifting circuit using the multilevel structure discussed in Section 7.4.4.

7.9 Repeat Problem 7.4, but design the shifting circuit using the multilevel structure discussed in Section 7.4.4. Compare the area and performance with those in Problem 7.4.

7.10 We define the distance from the Gray code word a to the Gray code word b as the number of transitions from code word a to code word b. For example, consider the 4-bit Gray code words "0101" and "1111" as a and b. The distance from a to b is 4 since four transitions are needed (i.e., "0101" ⇒ "0100" ⇒ "1100" ⇒ "1101" ⇒ "1111"). Design a circuit to calculate the distance of two 4-bit Gray code words and derive the VHDL code.

7.11 Although the code is compact, the synthesize result of the `compact_arch` architecture of the Gray code incrementor may not be more efficient than the `table_arch` architecture.

(a) Synthesize the two designs with an ASIC device. Compare the areas and performances.

(b) Extend the two designs for 8-bit Gray code. The `table_arch` architecture now has 2^8 entries. You may need to write a program (using C, Java etc.) to generate the VHDL code. Synthesize the two 8-bit designs with an ASIC device. Compare the areas and performances.

(c) Synthesize the two 8-bit designs with an FPGA device. Compare the areas and performances.

(d) If you have enough hardware resources, repeat parts (b) and (c) by gradually increasing the design to 10-, 12-, 14- and 16-bit inputs.

7.12 Design a priority encoder that returns the codes of the highest and second-highest priority requests. The input is an 8-bit `req` signal and the outputs are `code1`, `code2`, `valid1` and `valid2`, which are the 3-bit codes and 1-bit valid signals of the highest and second-highest priority requests respectively.

7.13 Many instrument panels use binary-coded-decimal (BCD) format, in which 10 decimal digits are coded by using 4 bits. During an addition operation, if the sum of a digit exceeds 9, 10 will be subtracted from the current digit and a carry is generated for the next digit. Design a 3-digit BCD adder which has two 12-bit inputs, representing two 3-digit BCD numbers, and an output, which is a 4-digit (16-bit) BCD number. Draw the top-level diagram and derive the VHDL code accordingly.

7.14 In an analog amplifier, the output voltage becomes saturated (i.e., reaching the most positive voltage, $+V_{cc}$, or the most negative voltage, $-V_{cc}$) when the output exceeds the maximal range. In some digital signal processing applications, we wish to design an 8-bit signed saturation adder that mimics the behavior of an analog amplifier; i.e., if the addition result overflows, the result becomes the most positive or the most negative numbers. Draw the top-level diagram and derive the VHDL code accordingly.

7.15 The two multipliers in Section 7.5.4 utilize seven 16-bit adders and seven 8-bit adders respectively.

(a) Determine the critical path for both designs.

(b) In an area-optimized adder, the propagation is proportional to the number of bits in the adder (i.e., on the order of $O(n)$). Assume that both designs utilize this kind of adder. Compare the propagation delays for the two designs.

7.16 Revise the designs in Section 7.5.4 to accommodate inputs in signed integer format. Derive the VHDL code. (*Hint*: An 8-bit 2's-complement number, $a_7a_6a_5a_4a_3a_2a_1a_0$, has a value of $-a_7{*}2^7 + a_6{*}2^6 + a_5{*}2^5 + \cdots + a_0{*}2^0$.)

7.17 Design an 8-bit combinational divider based on a long-division algorithm, the one you learned in elementary school. The inputs are 8-bit dividend and divisor in unsigned format, and the outputs are 8-bit quotient and remainder. Derive the VHDL code. (*Hint*: Division can be done by a sequence of "comparing and subtracting" operations. This operation takes two inputs, a and b, and returns a if $a < b$ and returns $a - b$ otherwise.)

7.18 One way to implement the population counter in Section 7.5.5 is to exhaustively construct a function table and use a single selected signal assignment statement to implement the table.

(a) Derive the VHDL code for an 8-bit population counter based on function table design. You may need to write a program (using C, Java etc.) to generate the VHDL code.

(b) Synthesize this design and the other two designs in Section 7.5.5 with an ASIC device. Compare the areas and performances.

(c) Synthesize this design and the other two designs in Section 7.5.5 with an FPGA device. Compare the areas and performances.

(d) Repeat parts (a) to (c) for 10- and 12-bit inputs.