CHAPTER 3

RT-LEVEL COMBINATIONAL CIRCUIT

3.1 INTRODUCTION

The gate-level circuits discussed in Chapter 1 utilize simple logical operators to describe gate-level design, which is composed of simple logic cells. In this chapter, we examine the HDL description of module-level circuits, which are composed of intermediate-sized components, such as adders, comparators, and multiplexers. Since these components are the basic building blocks used in *register transfer methodology*, it is sometimes referred to as RT-level design. We first discuss more sophisticated VHDL operators and routing constructs and then demonstrate the RT-level combinational circuit design through a series of examples.

3.2 RT-LEVEL COMPONENTS

In addition to the logical operators, relational operators and several arithmetic operators can also be synthesized automatically. These operators correspond to intermediate-sized module-level components, such as comparators and adders. We examine these operators in this section and also cover miscellaneous synthesis-related VHDL constructs. Tables 3.1 and 3.2 summarize the operators and their applicable data types used in this book.

Table 3.1 Operators and data types of VHDL-93 and IEEE `std_logic_1164` package

Operator	Description	Data type of operands	Data type of result
`a ** b`	exponentiation	`integer`	`integer`
`a * b`	multiplication		
`a / b`	division	*integer type for constants and*	
`a + b`	addition	*array boundaries, not synthesis*	
`a - b`	subtraction		
`a & b`	concatenation	1-D array, element	1-D array
`a = b`	equal to	any	`boolean`
`a /= b`	not equal to		
`a < b`	less than	scalar or 1-D array	`boolean`
`a <= b`	less than or equal to		
`a > b`	greater than		
`a >= b`	greater than or equal to		
not `a`	negation	`boolean`, `std_logic`,	same as operand
`a` **and** `b`	and	`std_logic_vector`	
`a` **or** `b`	or		
`a` **xor** `b`	xor		

Table 3.2 Overloaded operators and data types in the IEEE `numeric_std` package

Overloaded operator	Description	Data type of operands	Data type of result
`a * b`	arithmetic	`unsigned`, `natural`	`unsigned`
`a + b`	operation	`signed`, `integer`	`signed`
`a - b`			
`a = b`			
`a /= b`			
`a < b`	relational	`unsigned`, `natural`	`boolean`
`a <= b`	operation	`signed`, `integer`	`boolean`
`a > b`			
`a >= b`			

Table 3.3 Type conversions between `std_logic_vector` and numeric data types

Data type of `a`	To data type	Conversion function/type casting
`unsigned`, `signed`	`std_logic_vector`	`std_logic_vector(a)`
`signed`, `std_logic_vector`	`unsigned`	`unsigned(a)`
`unsigned`, `std_logic_vector`	`signed`	`signed(a)`
`unsigned`, `signed`	`integer`	`to_integer(a)`
`natural`	`unsigned`	`to_unsigned(a, size)`
`integer`	`signed`	`to_signed(a, size)`

3.2.1 Relational operators

Six relational operators are defined in the VHDL standard: = (equal to), /= (not equal to), < (less than), <= (less than or equal to), > (greater than), and >= (greater than or equal to). These operators compare operands of the same data type and return a value of the `boolean` data type. In this book, we don't use the `boolean` data type directly, but embed it in routing constructs. This is discussed in Sections 3.3 and 3.5. During synthesis, comparators are inferred for these operators.

3.2.2 Arithmetic operators

In the VHDL standard, arithmetic operations are defined for the `integer` data type and for the `natural` data type, which is a subtype of `integer` containing zero and positive integers. We usually prefer to have more control in synthesis and define the exact number of bits and format (i.e., signed or unsigned). The IEEE `numeric_std` package is developed for this purpose. In this book, we use the `integer` and `natural` data types for constants and array boundaries but not for synthesis.

IEEE numeric_std package The IEEE `numeric_std` package adds two new data types, `unsigned` and `signed`, and defines the relational and arithmetic operators over the new data types (known as *operator overloading*). The `unsigned` and `signed` data types are defined as an array with elements of the `std_logic` data type. The array is interpreted as the binary representation of unsigned or signed integers. We have to add an additional use statement to invoke the package:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numric_std.all;    -- invoke numeric_std package
```

The synthesizable overloaded operators are summarized in Table 3.2.

Multiplication is a complicated operation, and synthesis of the multiplication operator * depends on synthesis software and target device technology. Xilinx Spartan-3 FPGA family contains prefabricated combinational multiplier blocks. The Xilinx XST software can infer these blocks during synthesis, and thus the multiplication operator can be used in HDL code. The XCS200 device of the S3 board consists of twelve 18-by-18 multiplier blocks. While the synthesis of the multiplication operator is supported, we need to be aware of the limitation on the number and input width of these blocks and use them with care. **Xilinx specific**

Type conversion Because VHDL is a strongly typed language, `std_logic_vector`, `unsigned`, and `signed` are treated as different data types even when all of them are defined as an array with elements of the `std_logic` data type. A *conversion function* or *type casting* is needed to convert signals of different data types. The conversion is summarized in Table 3.3. Note that the `std_logic_vector` data type is not interpreted as a number and thus cannot be converted directly to an integer, and vice versa.

The following examples illustrate the common mistakes and remedies for type conversion. Assume that some signals are declared as follows:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
. . .
signal s1, s2, s3, s4, s5, s6: std_logic_vector(3 downto 0);
```

```
signal u1, u2, u3, u4, u5, u6, u7: unsigned(3 downto 0);
. . .
```

Let us first consider the following assignment statements:

```
u1 <= s1;   -- not ok, type mismatch
u2 <= 5;    -- not ok, type mismatch
s2 <= u3;   -- not ok, type mismatch
s3 <= 5;    -- not ok, type mismatch
```

They are all invalid because of type mismatch. The right-hand-side expression must be converted to the data type of the left-hand-side signal:

```
u1 <= unsigned(s1);       -- ok, type casting
u2 <= to_unsigned(5,4); -- ok, conversion function
s2 <= std_logic_vector(u3); -- ok, type casting
s3 <= std_logic_vector(to_unsigned(5,4)); -- ok
```

Note that two type conversions are needed for the last statement.

Let us consider statements that involve arithmetic operations. The following statements are valid since the + operator is defined with the `unsigned` and `natural` types in the IEEE `numeric_std` package.

```
u4 <= u2 + u1;  -- ok, both operands unsigned
u5 <= u2 + 1;   -- ok, operands unsigned and natural
```

On the other hand, the following statements are invalid since no overloaded arithmetic operation is defined for the `std_logic_vector` data type:

```
s5 <= s2 + s1; -- not ok, + undefined over the types
s6 <= s2 + 1;  -- not ok, + undefined over the types
```

To fix the problem, we must convert the operands to the `unsigned` (or `signed`) data type, perform addition, and then convert the result back to the `std_logic_vector` data type. The revised code becomes

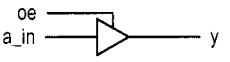
```
s5 <= std_logic_vector(unsigned(s2) + unsigned(s1)); -- ok
s6 <= std_logic_vector(unsigned(s2) + 1);            -- ok
```

Nonstandard arithmetic packages There are several non-IEEE arithmetic packages, which are `std_logic_arith`, `std_logic_unsigned`, and `std_logic_signed`. The `std_logic_arith` package is similar to the `numeric_std` package. The other two packages do not introduce any new data type but define overloaded arithmetic operators over the `std_logic_vector` data type. This approach eliminates the need for data conversion. Although using these packages seems to be less cumbersome initially, it is not good practice since these packages are not a part of IEEE standards and may introduce a compatibility problem in the long run. We do not use these packages in this book.

3.2.3 Other synthesis-related VHDL constructs

Concatenation operator The concatenation operator, &, combines segments of elements and small arrays to form a large array. The following example illustrates its use:

```
signal a1: std_logic;
signal a4: std_logic_vector(3 downto 0);
signal b8, c8, d8: std_logic_vector(7 downto 0);
```



oe	y
0	Z
1	a_in

Figure 3.1 Symbol and functional table of a tri-state buffer.

```
. . .
b8 <= a4 & a4;
c8 <= a1 & a1 & a4 & "00";
d8 <= b8(3 downto 0) & c8(3 downto 0);
```

Implementation of the concatenation operator involves reconnection of the input and output signals and only requires "wiring."

One major application of the & operator is to perform shifting operations. Although both VHDL standard and `numeric_std` package define shift functions, they sometimes cannot be synthesized automatically. The & operator can be used for shifting a signal for a fixed amount, as shown in the following example:

```
signal a: std_logic_vector(7 downto 0);
signal rot, shl, sha: std_logic_vector(7 downto 0);
. . .
-- rotate a to right 3 bits
rot <= a(2 downto 0) & a(8 downto 3);
-- shift a to right 3 bits and insert 0 (logic shift)
shl <= "000" & a(8 downto 3);
-- shift a to right 3 bits and insert MSB
-- (arithmetic shift)
sha <= a(8) & a(8) & a(8) & a(8 downto 3);
```

An additional routing circuit is needed if the amount of shifting is not fixed. The design of a barrel shifter is discussed in Section 3.7.3.

'Z' value of std_logic The `std_logic` data type has a value of `'Z'`, which implies *high impedance* or an open circuit. It is not a normal logic value and can only be synthesized by a *tri-state buffer*. The symbol and function table of a tri-state buffer are shown in Figure 3.1. Operation of the buffer is controlled by an enable signal, `oe` ("output enable"). When it is `'1'`, the input is passed to output. On the other hand, when it is `'0'`, the y output appears to be an open circuit. The code of the tri-state buffer is

```
y <= a_in when oe='1' else 'Z';
```

The most common application for a tri-state buffer is to implement a *bidirectional port* to better utilize a physical I/O pin. A simple example is shown in Figure 3.2. The `dir` signal controls the direction of signal flow of the `bi` pin. When it is '0', the tri-state buffer is in the high-impedance state and the `sig_out` signal is blocked. The pin is used as an input port and the input signal is routed to the `sig_in` signal. When the `dir` signal is '1', the pin is used as an output port and the `sig_out` signal is routed to an external circuit. The HDL code can be derived according to the diagram:

```
entity bi_demo is
   port(
```

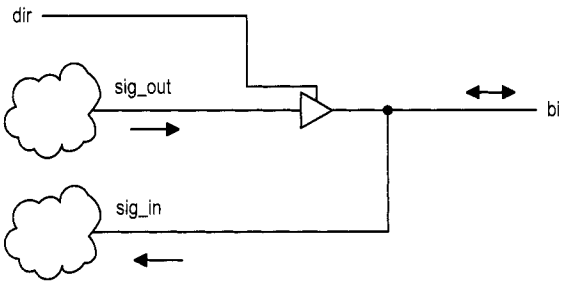


Figure 3.2 Single-buffer bidirectional I/O port.

```
        bi: inout std_logic;
        . . .
    )
begin
    sig_out <= output_expression;
    . . .
    some_signal <= expression_with_sig_in;
    . . .
    bi <= sig_out when dir='1' else 'Z';
    sig_in <= bi;
    . . .
```

Note that the mode of the `bi` port must be declared as **inout** for bidirectional operation.

Xilinx specific

For a Xilinx Spartan-3 device, a tri-state buffer exists only in the I/O block (IOB) of a physical pin. Thus, the tri-state buffer can only be used for I/O ports that are mapped to the physical pins of an FPGA device.

3.2.4 Summary

Because of the nature of a strongly typed language, the data type frequently confuses a new VHDL user. Since this book is focused on synthesis, only a small set of data types and operators are needed. Their uses can be summarized as follows:

- Use the `std_logic` and `std_logic_vector` data types in entity port declaration and for the internal signals that involve no arithmetic operations.
- Use the 'Z' value only to infer a tri-state buffer.
- Use the IEEE `numeric_std` package and its `unsigned` or `signed` data types for the internal signals that involve arithmetic operation.
- Use the data type casting or conversion functions in Table 3.3 to convert signals and expressions among the `std_logic_vector` and various numerical data types.
- Use VHDL's built-in `integer` data type and arithmetic operators for constant and array boundary expressions, but not for synthesis (i.e., not used as a data type for a signal).
- Embed the result of a relational operation, which is in the `boolean` data type, in routing constructs (discussed in Section 3.3).
- Use a user-defined two-dimensional data type for two-dimensional storage array (discussed in Section 4.2.3).

- Use a user-defined *enumerate data type* for the symbolic states of a finite state machine (discussed in Chapter 5).

3.3 ROUTING CIRCUIT WITH CONCURRENT ASSIGNMENT STATEMENTS

The *conditional signal assignment* and *selected signal assignment* statements are concurrent statements. Their behaviors are somewhat like the if and case statements of a conventional programming language. Instead of being executed sequentially, these statements are mapped to a routing network during synthesis.

3.3.1 Conditional signal assignment statement

Syntax and conceptual implementation The simplified syntax 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_n;
```

The Boolean expressions are evaluated successively in turn until one is found to be `true` and the corresponding value expression is assigned to the signal. The `value_expr_n` is assigned if all Boolean expressions are evaluated to be `false`.

The conditional signal assignment statement implies a cascading priority routing network. Consider the following statement:

```
r <= a + b + c when m = n else
     a - b     when m > n else
     c + 1;
```

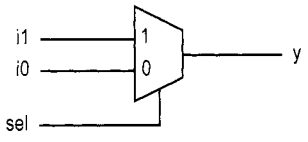
The routing is done by a sequence of 2-to-1 multiplexers. The diagram and truth table of a 2-to-1 multiplexer are shown in Figure 3.3(a), and the conceptual diagram of the statement is shown in Figure 3.3(b). It the first Boolean condition (i.e., `m=n`) is `true`, the result of `a+b+c` is routed to `r`. Otherwise, the data connected to the 0 port is passed to `r`. We need to trace the path along the 0 port and check the next Boolean condition (i.e., `m>n`) to determine whether the result of `a-b` or `c+1` is routed to the output.

Note that all the Boolean expressions and value expressions are evaluated concurrently. The values from the Boolean circuits set the selection signals of the multiplexers to route the desired value to the output. The number of cascading stages increases proportionally to the number of when-else clauses. A large number of when-else clauses will lead to a long cascading chain and introduce a large propagation delay.

Examples We use two simple examples to demonstrate use of the conditional signal assignment statement. The first example is a priority encoder. The priority encoder has four requests, `r(4)`, `r(3)`, `r(2)`, and `r(1)`, which are grouped as a single 4-bit `r` input, and `r(4)` has the highest priority. The output is the binary code of the highest-order request. The function table is shown in Table 3.4. The HDL code is shown in Listing 3.1.

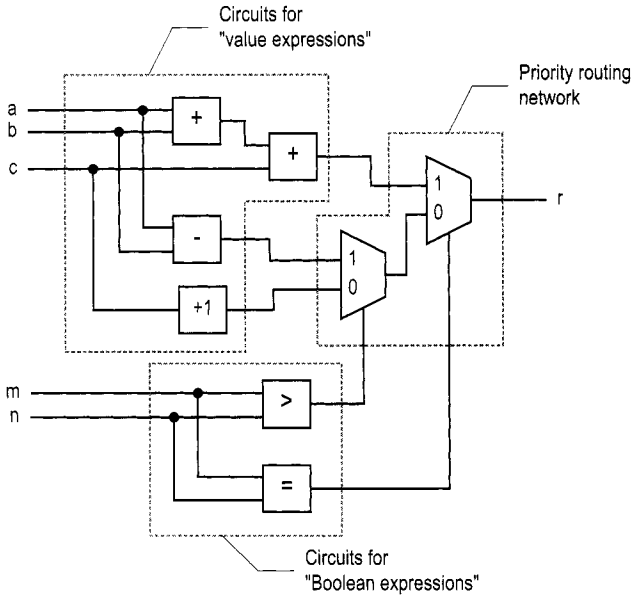
Listing 3.1 Priority encoder using a conditional signal assignment statement

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



sel	y
0 (false)	i0
1 (true)	i1

(a) Diagram of a 2-to-1 multiplexer



(b) Diagram of a conditional signal assignment statement

Figure 3.3 Implementation of a conditional signal assignment statement.

Table 3.4 Function table of a four-request priority encoder

input r	output pcode
1 – – –	100
0 1 – –	011
0 0 1 –	010
0 0 0 1	001
0 0 0 0	000

Table 3.5 Truth table of a 2-to-4 decoder with enable

input			output
en	a(1)	a(0)	y
0	–	–	0000
1	0	0	0001
1	0	1	0010
1	1	0	0100
1	1	1	1000

```
entity prio_encoder is
   port(
      r: in std_logic_vector(4 downto 1);
      pcode: out std_logic_vector(2 downto 0)
   );
end prio_encoder;

architecture cond_arch of prio_encoder is
begin
   pcode <= "100" when (r(4)='1') else
            "011" when (r(3)='1') else
            "010" when (r(2)='1') else
            "001" when (r(1)='1') else
            "000";
end cond_arch;
```

The code first checks the `r(4)` request and assigns `"100"` to `pcode` if it is asserted. It continues to check the `r(3)` request if `r(4)` is not asserted and repeats the process until all requests are examined.

The second example is a binary decoder. An n-to-2^n binary decoder asserts 1 bit of the 2^n-bit output according to the input combination. The functional table of a 2-to-4 decoder is shown in Table 3.5. The circuit also has a control signal, `en`, which enables the decoding function when asserted. The HDL code is shown in Listing 3.2.

Listing 3.2 Binary decoder using a conditional signal assignment statement

```
library ieee;
use ieee.std_logic_1164.all;
entity decoder_2_4 is
   port(
      a: in std_logic_vector(1 downto 0);
      en: in std_logic;
      y: out std_logic_vector(3 downto 0)
   );
end decoder_2_4;

architecture cond_arch of decoder_2_4 is
begin
    y <= "0000" when (en='0') else
         "0001" when (a="00") else
         "0010" when (a="01") else
```

```
            "0100" when (a="10") else
            "1000";    -- a="11"
end cond_arch;
```

The code first checks whether en is not asserted. If the condition is `false` (i.e., en is `'1'`), it tests the four binary combinations in sequence.

3.3.2 Selected signal assignment statement

Syntax and conceptual implementation The simplified syntax of a selected signal assignment statement is

```
with sel select
   sig <= value_expr_1 when choice_1,
          value_expr_2 when choice_2,
          value_expr_3 when choice_3,
          . . .
          value_expr_n when others;
```

The selected signal assignment statement is somewhat like a case statement in a traditional programming language. It assigns an expression to a signal according to the value of the `sel` signal. A choice (i.e., `choice_i`) must be a valid value or a set of valid values of `sel`. The choices have to be *mutually exclusive* (i.e., no value can be used more than once) and *all inclusive* (i.e., all values must be used). In other words, all possible values of `sel` must be covered by one and only one choice. The reserved word, **others**, is used in the end to cover unused values. Since the `sel` signal usually has the `std_logic_vector` data type, the **others** term is always needed to cover the unsynthesizable values (`'X'`, `'U'`, etc.).

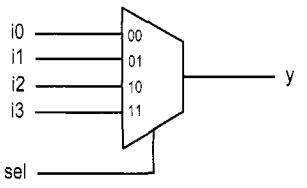
The selected signal assignment statement implies a multiplexing structure. Consider the following statement:

```
signal sel: std_logic_vector(1 downto 0);
. . .
with sel select
   r <= a + b + c   when "00",
        a - b       when "10",
        c + 1       when others;
```

For synthesis purposes, the `sel` signal can assume four possible values: "00", "01", "10", and "11". It implies a 2^2-to-1 multiplexer with `sel` as the selection signal. The diagram and functional table of the 2^2-to-1 multiplexer are shown in Figure 3.4(a), and the conceptual diagram of the statement is shown in Figure 3.4(b). The evaluated result of `a+b+c` is routed to `r` when `sel` is "00", the result of `a-b` is routed when `sel` is "10", and the result of `c+1` is routed when `sel` is "01" or "11".

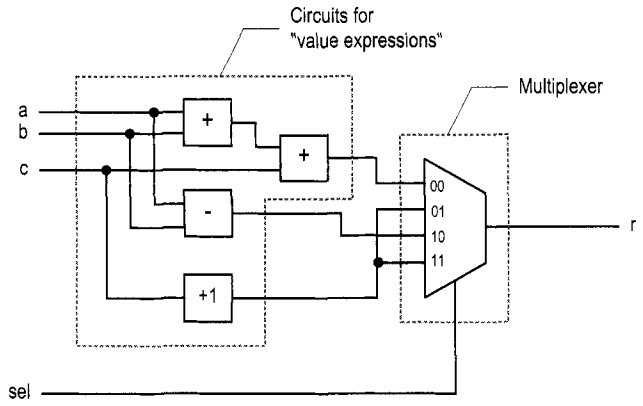
Again, note that all value expressions are evaluated concurrently. The `sel` signal is used as the selection signal to route the desired value to the output. The width (i.e., number of input ports) of the multiplexer increases geometrically with the number of bits of the `sel` signal.

Example We use the same encoder and decoder circuits to illustrate use of the selected signal assignment statement. The code for the priority encoder is shown in Listing 3.3. The entity declaration is identical to that in Listing 3.1 and is omitted.



sel	y
00	i0
01	i1
10	i2
11	i3

(a) Diagram and functional table of a 4-to-1 multiplexer



(b) Diagram of a selected signal assignment statement

Figure 3.4 Implementation of a selected signal assignment statement.

Listing 3.3 Priority encoder using a selected signal assignment statement

```
architecture sel_arch of prio_encoder is
begin
   with r select
      pcode <= "100" when "1000"|"1001"|"1010"|"1011"|
                          "1100"|"1101"|"1110"|"1111",
               "011" when "0100"|"0101"|"0110"|"0111",
               "010" when "0010"|"0011",
               "001" when "0001",
               "000" when others;    -- r="0000"
end sel_arch;
```

The code exhaustively lists all possible combinations of the `r` signal and the corresponding output values. Note that the | symbol is used if the choice is more than one value.

The code for the 2-to-4 decoder is shown in Listing 3.4.

Listing 3.4 Binary decoder using a selected signal assignment statement

```
architecture sel_arch of decoder_2_4 is
   signal s: std_logic_vector(2 downto 0);
begin
   s <= en & a;
   with s select
      y <= "0000" when "000"|"001"|"010"|"011",
           "0001" when "100",
```

```
            "0010" when "101",
            "0100" when "110",
            "1000" when others;    -- s="111"
end sel_arch;
```

We concatenate en and a to form a 3-bit signal, s, and use it as the selection signal. The remaining code again exhaustively lists all possible combinations and the corresponding output values.

3.4 MODELING WITH A PROCESS

3.4.1 Process

To facilitate system modeling, VHDL contains a number of *sequential statements*, which are executed in sequence. Since their behavior is different from that of a normal concurrent circuit model, these statements are encapsulated inside a *process*. A process itself is a concurrent statement. It can be thought of as a black box whose behavior is described by sequential statements.

Sequential statements include a rich variety of constructs, but many of them don't have clear hardware counterparts. A poorly coded process frequently leads to unnecessarily complex implementation or cannot be synthesized at all. Detailed discussion of sequential statements and processes is beyond the scope of this book. For synthesis, we restrict the use of the process to two purposes:

- Describe routing structures with *if* and *case* statements.
- Construct templates for memory elements (discussed in Chapter 4).

The simplified syntax of a process with a sensitivity list is

```
process(sensitivity_list)
begin
   sequential statement;
   sequential statement;
  . . .
end process;
```

The sensitivity_list is a list of signals to which the process responds (i.e., is "sensitive to"). For a combinational circuit, all the input signals should be included in this list. The body of a process is composed of any number of sequential statements.

3.4.2 Sequential signal assignment statement

The simplest sequential statement is a *sequential* signal assignment statement. The simplified syntax is

```
sig <= value_expression;
```

The statement must be encapsulated inside a process.

Although its syntax is similar to that of a simple *concurrent* signal assignment statement, the semantics are different. When a signal is assigned multiple times inside a process, only the last assignment takes effect. For example, the code segment

```
process(a,b)
begin
```

```
    c <= a and b;
    c <= a or b;
end process;
```

is the same as

```
process(a,b)
begin
    c <= a or b;
end process;
```

On the other hand, if they are concurrent signal assignment statements, as in

```
-- not within a process
c <= a and b;
c <= a or b;
```

the code infers an and cell and an or cell, whose outputs are tied together. It is not allowed in most device technology and thus is a design error.

The semantics of assigning a signal multiple times inside a process is subtle and can sometimes be error-prone. Detailed explanations can be found in the references cited in the Bibliographic section. We use multiple assignments only to avoid unintended memory, as discussed in Section 3.5.4.

3.5 ROUTING CIRCUIT WITH IF AND CASE STATEMENTS

If and *case* statements are two other commonly used sequential statements. In synthesis, they can be used to describe routing structures.

3.5.1 If statement

Syntax and conceptual implementation The simplified syntax of an if statement is

```
if boolean_expr_1 then
    sequential_statements;
elsif boolean_expr_2 then
    sequential_statements;
elsif boolean_expr_3 then
    sequential_statements;
. . .
else
    sequential_statements;
end if;
```

It has one *then branch*, one or more optional *elsif branches*, and one optional *else branch*. The Boolean expressions are evaluated sequentially until an expression is evaluated as `true` or the else branch is reached, and the statements in the corresponding branch will be executed.

An if statement and a concurrent conditional signal assignment statement are somewhat similar. The two statements are equivalent if each branch of the if statement contains only a single sequential signal assignment statement. For example, the previous statement

```
r <= a + b + c when m = n else
     a - b     when m > 0 else
     c + 1;
```

can be rewritten as

```
process(a,b,c,m,n)
begin
   if m = n then
      r <= a + b + c;
   elsif m > 0 then
      r <= a - b;
   else
     r <= c + 1;
   end if;
end;
```

As in a conditional signal assignment statement, the if statement infers a similar priority routing structure during synthesis.

Example The codes of the same priority encoder and written with an if statement are shown in Listings 3.5 and 3.6. They are similar to those in Listings 3.1 and 3.2. Note that the if statement must be encapsulated inside a process.

Listing 3.5 Priority encoder using an if statement

```
architecture if_arch of prio_encoder is
begin
   process(r)
   begin
      if (r(4)='1') then
         pcode <= "100";
      elsif (r(3)='1')then
         pcode <= "011";
      elsif (r(2)='1')then
         pcode <= "010";
      elsif (r(1)='1')then
         pcode <= "001";
      else
         pcode <= "000";
      end if;
   end process;
end if_arch;
```

Listing 3.6 Binary decoder using an if statement

```
architecture if_arch of decoder_2_4 is begin
   process(en,a)
   begin
      if (en='0') then
         y <= "0000";
      elsif (a="00") then
         y <= "0001";
      elsif (a="01")then
         y <= "0010";
      elsif (a="10")then
         y <= "0100";
      else
         y <= "1000";
```

```
        end if;
    end process;
end if_arch;
```

3.5.2 Case statement

Syntax and conceptual implementation The simplified syntax of a case statement is

```
case sel is
    when choice_1 =>
        sequential statements;
    when choice_2 =>
        sequential statements;
    . . .
    when others =>
        sequential statements;
end case;
```

A case statement uses the `sel` signal to select a set of sequential statements for execution. As in a selected signal assignment statement, a choice (i.e., `choice_i`) must be a valid value or a set of valid values of `sel`, and the choices have to be mutually exclusive and all inclusive. Note that the **others** term at the end covers the unused values.

A case statement and a concurrent selected signal assignment statement are somewhat similar. The two statements are equivalent if each branch of the case statement contains only a single sequential signal assignment statement. For example, the previous statement

```
with sel select
    r <= a + b + c   when "00",
         a - b       when "10",
         c + 1       when others;
```

can be rewritten as

```
process(a,b,c,sel)
begin
    case sel is
        when "00" =>
            r <= a + b + c;
        when "10" =>
            r <= a - b;
        when others =>
            r <= c + 1;
    end case;
end;
```

As in a selected signal assignment statement, the case statement infers a similar multiplexing structure during synthesis.

Example The codes of the same priority encoder and decoder written with a case statement are shown in Listings 3.7 and 3.8. As in Listings 3.3 and 3.4, the codes exhaustively lists all possible input combinations and the corresponding output values.

Listing 3.7 Priority encoder using a case statement

```
architecture case_arch of prio_encoder is
begin
   process(r)
   begin
      case r is
         when "1000"|"1001"|"1010"|"1011"|
              "1100"|"1101"|"1110"|"1111" =>
            pcode <= "100";
         when "0100"|"0101"|"0110"|"0111" =>
            pcode <= "011";
         when "0010"|"0011" =>
            pcode <= "010";
         when "0001" =>
            pcode <= "001";
         when others =>
            pcode <= "000";
      end case;
   end process;
end case_arch;
```

Listing 3.8 Binary decoder using a case statement

```
architecture case_arch of decoder_2_4 is
   signal s: std_logic_vector(2 downto 0);
begin
   s <= en & a;
   process(s)
   begin
      case s is
         when "000"|"001"|"010"|"011" =>
            y <= "0001";
         when "100" =>
            y <= "0001";
         when "101" =>
            y <= "0010";
         when "110" =>
            y <= "0100";
         when others =>
            y <= "1000";
      end case;
   end process;
end case_arch;
```

3.5.3 Comparison to concurrent statements

The preceding subsections show that the simple if and case statements are equivalent to the conditional and selected signal assignment statements. However, an if or case statement allows *any number* and *any type* of sequential statements in their branches and thus is more flexible and versatile. Disciplined use can make the code more descriptive and even make a circuit more efficient.

This can be illustrated by two code segments. First, consider a circuit that sorts the values of two input signals and routes them to the `large` and `small` outputs. This can be done by using two conditional signal assignment statements:

```
large <= a when a > b else
         b;
small <= b when a > b else
         a;
```

Since there are two relation operators (i.e., two >) in code, synthesis software may infer two greater-than comparators. The same function can be coded by a single if statement:

```
process(a,b)
begin
   if a > b then
      large <= a;
      small <= b;
   else
      large <= b;
      small <= a;
   end if;
end;
```

The code consists of only a single relational operator.

Second, let us consider a circuit that routes the maximal value of three input signals to the output. This can be clearly described by nested two-level if statements:

```
process(a,b,c)
begin
   if (a > b) then
      if (a > c) then
         max <= a;
      else
         max <= c;
      end if;
   else
      if (b > c) then
         max <= b;
      else
         max <= c;
      end if;
   end if;
end process;
```

We can translate the if statement to a "single-level" conditional signal assignment statement:

```
max <= a when ((a > b) and (a > c)) else
       c when (a > b) else
       b when (b > c) else
       c;
```

Since no nesting is allowed, the code is less intuitive. If concurrent statements must be used, a better alternative is to describe the circuit with three conditional signal assignment statements:

```
signal ac_max, bc_max: std_logic;
. . .
```

```
ac_max <= a when (a > c) else
          c;
bc_max <= b when (b > c) else
          c;
max <= ac_max when (a > b) else
       bc_max;
```

3.5.4 Unintended memory

Although a process is flexible, a subtle error in code may infer incorrect implementation. One common problem is the inclusion of intended memory in a combinational circuit. The VHDL standard specifies that a signal will *keep its previous value* if it is not assigned in a process. During synthesis, this infers an internal state (via a closed feedback loop) or a memory element (such as a latch).

To prevent unintended memory, we should observe the following rules while developing code for a combinational circuit:

- Include all input signals in the sensitivity list.
- Include the else branch in an if statement.
- Assign a value to every signal in every branch.

For example, the following code segment tries to generate a greater-than (i.e., `gt`) and an equal-to (i.e., `eq`) output signal:

```
process(a)                -- b missing from sensitivity list
begin
   if (a > b) then        -- eq not assigned in this branch
      gt <= '1';
   elsif (a = b) then     -- gt not assigned in this branch
      eq <= '1';
   end if;                -- else branch is omitted
end process;
```

Although the syntax is correct, it violates all three rules. For example, `gt` will keep its previous value when the `a>b` expression is `false` and a latch will be inferred accordingly. The correct code should be

```
process(a,b)
begin
   if (a > b) then
      gt <= '1';
      eq <= '0';
   elsif (a = b) then
      gt <= '0';
      eq <= '1';
   else
      gt <= '0';
      eq <= '0';
   end if;
end process;
```

Since multiple sequential signal assignment statements are allowed inside a process, we can correct the problem by assigning a default value in the beginning:

```
process(a,b)
begin
   gt <= '0';              -- assign default value
   eq <= '0';
   if (a > b) then
      gt <= '1';
   elsif (a = b) then
      eq <= '1';
   end if;
end process;
```

The gt and eq signals assume '0' if they are not assigned a value later. As discussed earlier, assigning a signal multiple times inside a process can be error-prone. For synthesis, this should not be used in other context and should be considered as shorthand to satisfy the "assigning all signals in all branches" rule.

3.6 CONSTANTS AND GENERICS

3.6.1 Constants

HDL code frequently uses constant values in expressions and array boundaries. One good design practice is to replace the "hard literals" with symbolic constants. It makes code clear and helps future maintenance and revision. The constant declaration can be included in the architecture's declaration section, and it syntax is

```
constant const_name: data_type := value_expression;
```

For example, we can declare two constants as

```
constant DATA_BIT: integer := 8;
constant DATA_RANGE: integer := 2**DATA_BIT - 1;
```

The constant expression is evaluated during preprocessing and thus requires no physical circuit. In this book, we use capital letters for constants.

The use of a constant can best be explained by an example. Assume that we want to design an adder with the carry-out bit. One way to do it is to extend the input by 1 bit and then perform regular addition. The MSB of the summation becomes the carry-out bit. The code is shown in Listing 3.9.

Listing 3.9 Adder using a hard literal

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity add_w_carry is
   port(
      a, b: in std_logic_vector(3 downto 0);
      cout: out std_logic;
      sum: out std_logic_vector(3 downto 0)
   );
end add_w_carry;

architecture hard_arch of add_w_carry is
   signal a_ext, b_ext, sum_ext: unsigned(4 downto 0);
```

```
begin
   a_ext <= unsigned('0' & a);
   b_ext <= unsigned('0' & b);
   sum_ext <= a_ext + b_ext;
   sum <= std_logic_vector(sum_ext(3 downto 0));
   cout <= sum_ext(4);
end hard_arch;
```

The code is for a 4-bit adder. Hard literals, such as 3 and 4, are used for the ranges, as in `unsigned(4 downto 0)` and `sum_ext(3 downto 0)`, and the MSB, as in `sum_ext(4)`. If we want to revise the code for an 8-bit adder, these literals have to be modified manually. This will be a tedious and error-prone process if the code is complex and the literals are referred to in many places.

To improve the readability, we can use a symbolic constant, `N`, to represent the number of bits of the adder. The revised architecture body is shown in Listing 3.10.

Listing 3.10 Adder using a constant

```
architecture const_arch of add_w_carry is
   constant N: integer := 4;
   signal a_ext, b_ext, sum_ext: unsigned(N downto 0);
begin
   a_ext <= unsigned('0' & a);
   b_ext <= unsigned('0' & b);
   sum_ext <= a_ext + b_ext;
   sum <= std_logic_vector(sum_ext(N-1 downto 0));
   cout <= sum_ext(N);
end const_arch;
```

The constant makes the code easier to understand and maintain.

3.6.2 Generics

VHDL provides a construct, known as a *generic*, to pass information into an entity and component. Since a generic cannot be modified inside the architecture, it functions somewhat like a constant. A generic is declared inside an entity declaration, just before the port declaration:

```
entity entity_name is
   generic(
      generic_name: data_type := default_values;
      generic_name: data_type := default_values;
      . . .
      generic_name: data_type := default_values
   )
   port(
      port_name: mode data_type;
      . . .
   );
end entity_name;
```

For example, the previous adder code can be modified to use the adder width as a generic, as shown in Listing 3.11.

Listing 3.11 Adder using a generic

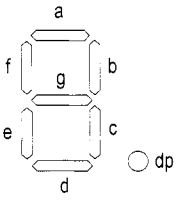
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity gen_add_w_carry is
   generic(N: integer:=4);
   port(
      a, b: in std_logic_vector(N-1 downto 0);
      cout: out std_logic;
      sum: out std_logic_vector(N-1 downto 0)
   );
end gen_add_w_carry;

architecture arch of gen_add_w_carry is
   signal a_ext, b_ext, sum_ext: unsigned(N downto 0);
begin
   a_ext <= unsigned('0' & a);
   b_ext <= unsigned('0' & b);
   sum_ext <= a_ext + b_ext;
   sum <= std_logic_vector(sum_ext(N-1 downto 0));
   cout <= sum_ext(N);
end arch;
```

The N generic is declared in line 5 with a default value of 4. After N is declared, it can be used in the port declaration and architecture body, just like a constant.

If the adder is later used as a component in other code, we can assign the desired value to the generic in component instantiation. This is known as *generic mapping*. The default value will be used if generic mapping is omitted. Use of the generic in component instantiation is shown below.

```
signal a4, b4, sum4: unsigned(3 downto 0);
signal a8, b8, sum8: unsigned(7 downto 0);
signal a16, b16, sum16: unsigned(15 downto 0);
signal c4, c8, c16: std_logic;
. . .
-- instantiate 8-bit adder
adder_8_unit: work.gen_add_w_carry(arch)
   generic map(N=>8)
   port map(a=>a8, b=>b8, cout=>c8, sum=>sum8));
-- instantiate 16-bit adder
adder_16_unit: work.gen_add_w_carry(arch)
   generic map(N=>16)
   port map(a=>a16, b=>b16, cout=>c16, sum=>sum16));
-- instantiate 4-bit adder
-- (generic mapping omitted, default value 4 used)
adder_4_unit: work.gen_add_w_carry(arch)
   port map(a=>a4, b=>b4, cout=>c4, sum=>sum4));
```

A generic provides a mechanism to create *scalable code*, in which the "width" of a circuit can be adjusted to meet a specific need. This makes code more portable and encourages design reuse.



(a) Diagram of a seven-segment LED display

(b) Hexadecimal digit patterns

Figure 3.5 Seven-segment LED display and hexadecimal patterns.

3.7 DESIGN EXAMPLES

3.7.1 Hexadecimal digit to seven-segment LED decoder

The sketch of a seven-segment LED display is shown in Figure 3.5(a). It consists of seven LED bars and a single round LED decimal point. On the prototyping board, the seven-segment LED is configured as active low, which means that an LED segment is lit if the corresponding control signal is '0'.

A hexadecimal digit to seven-segment LED decoder treats a 4-bit input as a hexadecimal digit and generates appropriate LED patterns, as shown in Figure 3.5(b). For completeness, we assume that there is also a 1-bit input, `dp`, which is connected directly to the decimal point LED. The LED control signals, `dp`, `a`, `b`, `c`, `d`, `e`, `f`, and `g`, are grouped together as a single 8-bit signal, `sseg`. The code is shown in Listing 3.12. It uses one selected signal assignment statement to list all the desired patterns for the seven LSBs of the `sseg` signal. The MSB is connected to `dp`.

Listing 3.12 Hexadecimal digit to seven-segment LED decoder

```
library ieee;
use ieee.std_logic_1164.all;
entity hex_to_sseg is
   port(
      hex: in std_logic_vector(3 downto 0);
      dp: in std_logic;
      sseg: out std_logic_vector(7 downto 0)
   );
end hex_to_sseg;

architecture arch of hex_to_sseg is
begin
   with hex select
      sseg(6 downto 0) <=
         "0000001" when "0000",
         "1001111" when "0001",
```

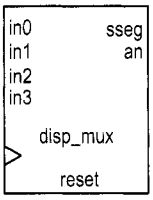
```
            "0010010" when "0010",
            "0000110" when "0011",
            "1001100" when "0100",
            "0100100" when "0101",
            "0100000" when "0110",
            "0001111" when "0111",
            "0000000" when "1000",
            "0000100" when "1001",
            "0001000" when "1010", --a
            "1100000" when "1011", --b
            "0110001" when "1100", --c
            "1000010" when "1101", --d
            "0110000" when "1110", --e
            "0111000" when others; --f
    sseg(7) <= dp;
end arch;
```

There are four seven-segment LED displays on the prototyping board. To save the number of FPGA chip's I/O pins, a time-multiplexing scheme is used. The block diagram of the time-multiplexing module, `disp_mux`, is shown in Figure 3.6(a). The inputs are `in0`, `in1`, `in2`, and `in3`, which correspond to four 8-bit seven-segment LED patterns, and the outputs are `an`, which is a 4-bit signal that enables the four displays individually, and `sseg`, which is the shared 8-bit signal that controls the eight LED segments. The circuit generates a properly timed enable signal and routes the four input patterns to the output alternatively. The design of this module is discussed in Chapter 4. For now, we just treat it as a black box that takes four seven-segment LED patterns, and instantiate it in the code.

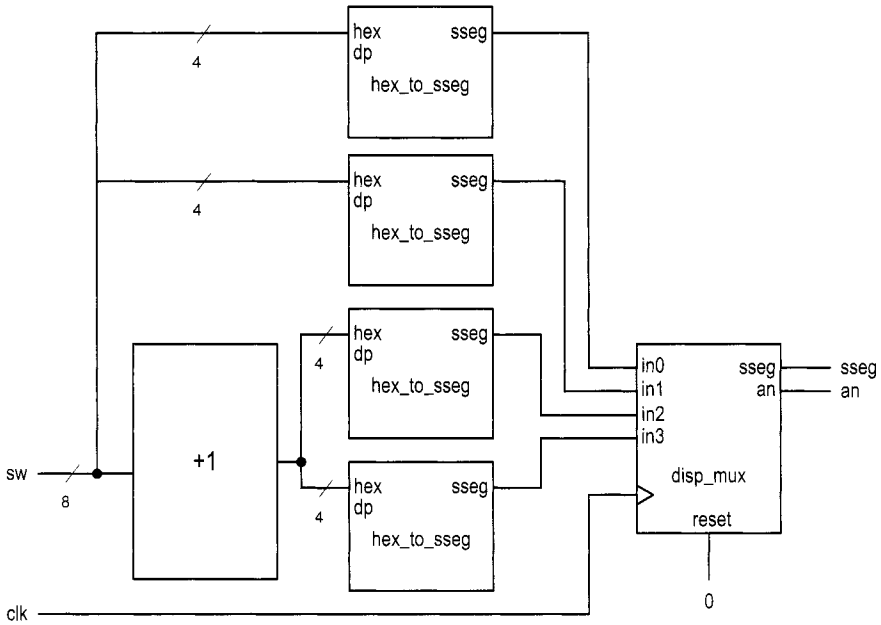
Testing circuit We use a simple 8-bit increment circuit to verify operation of the decoder. The sketch is shown in Figure 3.6(b). The `sw` input is the 8-bit switch of the prototyping board. It is fed to an incrementor to obtain `sw+1`. The original and incremented `sw` signals are then passed to four decoders to display the four hexadecimal digits on seven-segment LED displays. The code is shown in Listing 3.13.

Listing 3.13 Hex-to-LED decoder testing circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity hex_to_sseg_test is
   port(
      clk: in std_logic;
      sw: in std_logic_vector(7 downto 0);
      an: out std_logic_vector(3 downto 0);
      sseg: out std_logic_vector(7 downto 0)
   );
end hex_to_sseg_test;

architecture arch of hex_to_sseg_test is
   signal inc: std_logic_vector(7 downto 0);
   signal led3, led2, led1, led0: std_logic_vector(7 downto 0);
begin
   -- increment input
   inc <= std_logic_vector(unsigned(sw) + 1);
```



(a) Block diagram of an LED time-multiplexing module



(b) Block diagram of a decoder testing circuit

Figure 3.6 LED time-multiplexing module and decoder testing circuit.

```
   -- instantiate four instances of hex decoders
   -- instance for 4 LSBs of input
   sseg_unit_0: entity work.hex_to_sseg
      port map(hex=>sw(3 downto 0), dp =>'0', sseg=>led0);
   -- instance for 4 MSBs of input
   sseg_unit_1: entity work.hex_to_sseg
      port map(hex=>sw(7 downto 4), dp =>'0', sseg=>led1);
   -- instance for 4 LSBs of incremented value
   sseg_unit_2: entity work.hex_to_sseg
      port map(hex=>inc(3 downto 0), dp =>'1', sseg=>led2);
   -- instance for 4 MSBs of incremented value
   sseg_unit_3: entity work.hex_to_sseg
      port map(hex=>inc(7 downto 4), dp =>'1', sseg=>led3);

   -- instantiate 7-seg LED display time-multiplexing module
   disp_unit: entity work.disp_mux
      port map(
         clk=>clk, reset=>'0',
         in0=>led0, in1=>led1, in2=>led2, in3=>led3,
         an=>an, sseg=>sseg);
end arch;
```

We can follow the procedure in Chapter 2 to synthesize and implement the circuit on the prototyping board. Note that the disp_mux.vhd file, which contains the code for the time-multiplexing module, and the ucf constraint file must be included in the Xilinx ISE project during synthesis.

3.7.2 Sign-magnitude adder

An integer can be represented in *sign-magnitude* format, in which the MSB is the sign and the remaining bits form the magnitude. For example, 3 and −3 become "0011" and "1011" in 4-bit sign-magnitude format.

A sign-magnitude adder performs an addition operation in this format. The operation can be summarized as follows:

- If the two operands have the same sign, add the magnitudes and keep the sign.
- If the two operands have different signs, subtract the smaller magnitude from the larger one and keep the sign of the number that has the larger magnitude.

One possible implementation is to divide the circuit into two stages. The first stage sorts the two input numbers according to their magnitudes and routes them to the `max` and `min` signals. The second stage examines the signs and performs addition or subtraction on the magnitude accordingly. Note that since the two numbers have been sorted, the magnitude of `max` is always larger than that of `min` and the final sign is the sign of `max`.

The code is shown in Listing 3.14, which realizes the two-stage implementation scheme. For clarity, we split the input number internally and use separate sign and magnitude signals. A generic, `N`, is used to represent the width of the adder. Note that the relevant magnitude signals are declared as `unsigned` to facilitate the arithmetic operation, and type conversions are performed at the beginning and end of the code.

Listing 3.14 Sign-magnitude adder

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity sign_mag_add is
   generic(N: integer:=4);  -- default 4 bits
   port(
      a, b: in std_logic_vector(N-1 downto 0);
      sum: out std_logic_vector(N-1 downto 0)
   );
end sign_mag_add;

architecture arch of sign_mag_add is
   signal mag_a, mag_b: unsigned(N-2 downto 0);
   signal mag_sum, max, min: unsigned(N-2 downto 0);
   signal sign_a, sign_b, sign_sum: std_logic;
begin
   mag_a <= unsigned(a(N-2 downto 0));
   mag_b <= unsigned(b(N-2 downto 0));
   sign_a <= a(N-1);
   sign_b <= b(N-1);
   -- sort according to magnitude
   process(mag_a,mag_b,sign_a,sign_b)
   begin
      if mag_a > mag_b then
         max <= mag_a;
         min <= mag_b;
         sign_sum <= sign_a;
      else
         max <= mag_b;
         min <= mag_a;
         sign_sum <= sign_b;
      end if;
   end process;
   -- add/sub magnitude
   mag_sum <= max + min when sign_a=sign_b else
              max - min;
   --form output
   sum <= std_logic_vector(sign_sum & mag_sum);
end arch;
```

Testing circuit We use a 4-bit sign-magnitude adder to verify the circuit operation. The sketch of the testing circuit is shown in Figure 3.7. The two input numbers are connected to the 8-bit switch, and the sign and magnitude are shown on two seven-segment LED displays. Two pushbuttons are used as the selection signal of a multiplexer to route an operand or the sum to the display circuit. The rightmost even-segment LED shows the 3-bit magnitude, which is appended with a '0' in front and fed to the hexadecimal to seven-segment LED decoder. The next LED displays the sign bit, which is blank for the plus sign and is lit with a middle LED segment for the minus sign. The two LED patterns are then fed to the time-multiplexing module, `disp_mux`, as explained in Section 3.7.1. The code is shown in Listing 3.15.

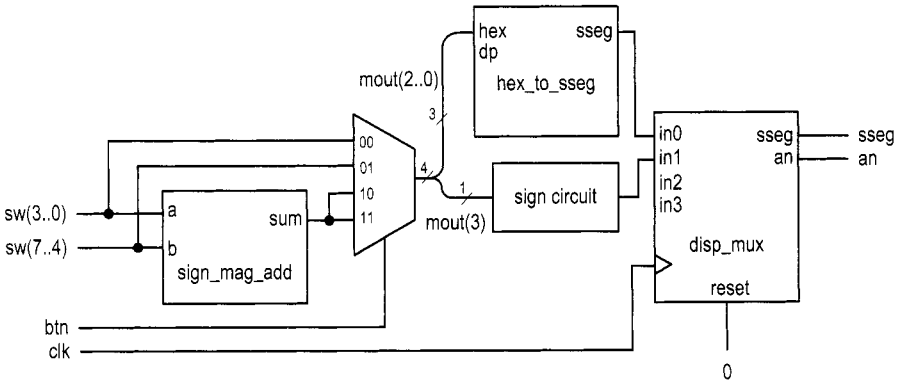


Figure 3.7 Sign-magnitude adder testing circuit.

Listing 3.15 Sign-magnitude adder testing circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity sm_add_test is
   port(
      clk: in std_logic;
      btn: in std_logic_vector(1 downto 0);
      sw: in std_logic_vector(7 downto 0);
      an: out std_logic_vector(3 downto 0);
      sseg: out std_logic_vector(7 downto 0)
   );
end sm_add_test;

architecture arch of sm_add_test is
   signal sum, mout, oct: std_logic_vector(3 downto 0);
   signal led3, led2, led1, led0: std_logic_vector(7 downto 0);
begin
   -- instantiate adder
   sm_adder_unit: entity work.sign_mag_add
      generic map(N=>4)
      port map(a=>sw(3 downto 0), b=>sw(7 downto 4),
               sum=>sum);

   -- 3-to-1 mux to select a number to display
   with btn select
      mout <= sw(3 downto 0) when "00",   -- a
              sw(7 downto 4) when "01",   -- b
              sum when others;            -- sum

   -- magnitude displayed on rightmost 7-seg LED
   oct <= '0' & mout(2 downto 0);
   sseg_unit: entity work.hex_to_sseg
      port map(hex=>oct, dp=>'0', sseg=>led0);
   -- sign displayed on 2nd 7-seg LED
```

```
    led1 <= "11111110" when mout(3)='1' else -- middle bar
            "11111111";                      -- blank
    -- other two 7-seg LEDs blank
    led2 <= "11111111";
    led3 <= "11111111";

    -- instantiate display multiplexer
    disp_unit: entity work.disp_mux
       port map(
          clk=>clk, reset=>'0',
          in0=>led0, in1=>led1, in2=>led2, in3=>led3,
          an=>an, sseg=>sseg);
end arch;
```

3.7.3 Barrel shifter

Although VHDL has built-in shift functions, they sometimes cannot be synthesized automatically. In this subsection, we examine an 8-bit barrel shifter that rotates an arbitrary number of bits to right. The circuit has an 8-bit data input, a, and a 3-bit control signal, amt, which specifies the amount to be rotated. The first design uses a selected signal assignment statement to exhaustively list all combinations of the amt signal and the corresponding rotated results. The code is shown in Listing 3.16.

Listing 3.16 Barrel shifter using a selected signal assignment statement

```
library ieee;
use ieee.std_logic_1164.all;
entity barrel_shifter 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 barrel_shifter ;

architecture sel_arch of barrel_shifter 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 sel_arch;
```

While the code is straightforward, it will become cumbersome when the number of input bits increases. Furthermore, a large number of choices implies a wide multiplexer, which makes synthesis difficult and leads to a large propagation delay. Alternatively, we can construct the circuit by stages. In the nth stage, the input signal is either passed directly to

output or rotated right by 2^n positions. The nth stage is controlled by the nth bit of the amt signal. Assume that the 3 bits of amt are $m_2m_1m_0$. The total rotated amount after three stages is $m_2 2^2 + m_1 2^1 + m_0 2^0$, which is the desired rotating amount. The code for this scheme is shown in Listing 3.17.

Listing 3.17 Barrel shifter using multi-stage shifts

```
architecture multi_stage_arch of barrel_shifter is
   signal s0, s1: std_logic_vector(7 downto 0);
begin
   -- stage 0, shift 0 or 1 bit
   s0 <= a(0) & a(7 downto 1) when amt(0)='1' else
         a;
   -- stage 1, shift 0 or 2 bits
   s1 <= s0(1 downto 0) & s0(7 downto 2) when amt(1)='1' else
         s0;
   -- stage 2, shift 0 or 4 bits
   y <= s1(3 downto 0) & s0(7 downto 4) when amt(2)='1' else
        s1;
end multi_stage_arch ;
```

Testing circuit To test the circuit, we can use the 8-bit switch for the a signal, three pushbutton switches for the amt signal, and the eight discrete LEDs for output. Instead of deriving a new constraint file for pin assignment, we create a new HDL file that wraps the barrel shifter circuit and maps its signals to the prototyping board's signals. The code is shown in Listing 3.18.

Listing 3.18 Barrel shifter testing circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity shifter_test is
   port(
      sw: in std_logic_vector(7 downto 0);
      btn: in std_logic_vector(2 downto 0);
      led: out std_logic_vector(7 downto 0)
   );
end shifter_test;

architecture arch of shifter_test is
begin
   shift_unit: entity work.barrel_shifter(multi_stage_arch)
      port map(a=>sw, amt=>btn, y=>led);
end arch;
```

3.7.4 Simplified floating-point adder

Floating point is another format to represent a number. With the same number of bits, the range in floating-point format is much larger than that in signed integer format. Although VHDL has a built-in floating-point data type, it is too complex to be synthesized automatically.

		sort	align	add/sub	normalize
eg. 1	+0.54E3	-0.87E4	-0.87E4	-0.87E4	-0.87E4
	-0.87E4	+0.54E3	+0.05E4	+0.05E4	+0.05E4
				-0.82E4	-0.82E4
eg. 2	+0.54E3	-0.55E3	-0.55E3	-0.55E3	-0.55E3
	-0.55E3	+0.54E3	+0.54E3	+0.54E3	+0.54E3
				-0.01E3	-0.10E2
eg. 3	+0.54E0	-0.55E0	-0.55E0	-0.55E0	-0.55E0
	-0.55E0	+0.54E0	+0.54E0	+0.54E0	+0.54E0
				-0.01E0	-0.00E0
eg. 4	+0.56E3	+0.56E3	+0.56E3	+0.56E3	+0.56E3
	+0.52E3	+0.52E3	+0.52E3	+0.52E3	+0.52E3
				+1.07E3	+0.10E4

Figure 3.8 Floating-point addition examples.

Detailed discussion of floating-point representation is beyond the scope of this book. We use a simplified 13-bit format in this example and ignore the round-off error. The representation consists of a sign bit, s, which indicates the sign of the number (1 for negative); a 4-bit exponent field, e, which represents the exponent; and an 8-bit significand field, f, which represents the significand or the fraction. In this format, the value of a floating-point number is $(-1)^s * .f * 2^e$. The $.f * 2^e$ is the magnitude of the number and $(-1)^s$ is just a formal way to state that "s equal to 1 implies a negative number." Since the sign bit is separated from the rest of the number, floating-point representation can be considered as a variation of the sign-magnitude format.

We also make the following assumptions:

- Both exponent and significand fields are in unsigned format.
- The representation has to be either normalized or zero. *Normalized representation* means that the MSB of the significand field must be '1'. If the magnitude of the computation result is smaller than the smallest normalized nonzero magnitude, $0.10000000 * 2^{0000}$, it must be converted to zero.

Under these assumptions, the largest and smallest nonzero magnitudes are $0.11111111 * 2^{1111}$ and $0.10000000 * 2^{0000}$, and the range is about 2^{16} (i.e., $\frac{0.11111111*2^{1111}}{0.10000000*2^{0000}}$).

Our floating-point adder design follows the process of adding numbers manually in scientific notation. This process can best be explained by examples. We assume that the widths of the exponent and significand are 2 and 1 digits, respectively. Decimal format is used for clarity. The computations of several representative examples are shown in Figure 3.8. The computation is done in four major steps:

1. *Sorting*: puts the number with the larger magnitude on the top and the number with the smaller magnitude on the bottom (we call the sorted numbers "big number" and "small number").
2. *Alignment*: aligns the two numbers so they have the same exponent. This can be done by adjusting the exponent of the small number to match the exponent of the big

number. The significand of the small number has to shift to the right according to the difference in exponents.

3. *Addition/subtraction*: adds or subtracts the significands of two aligned numbers.
4. *Normalization*: adjusts the result to normalized format. Three types of normalization procedures may be needed:
   - After a subtraction, the result may contain leading zeros in front, as in example 2.
   - After a subtraction, the result may be too small to be normalized and thus needs to be converted to zero, as in example 3.
   - After an addition, the result may generate a carry-out bit, as in example 4.

Our binary floating-point adder design uses a similar algorithm. To simplify the implementation, we ignore the rounding. During alignment and normalization, the lower bits of the significand will be discarded when shifted out. The design is divided into four stages, each corresponding to a step in the foregoing algorithm. The suffixes, 'b', 's', 'a', 'r', and 'n', used in signal names are for "big number," "small number," "aligned number," "result of addition/subtraction," and "normalized number," respectively. The code is developed according to these stages, as shown in Listing 3.19.

Listing 3.19 Simplified floating-point adder

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity fp_adder is
   port (
      sign1, sign2: in  std_logic;
      exp1, exp2: in  std_logic_vector(3 downto 0);
      frac1, frac2: in  std_logic_vector(7 downto 0);
      sign_out: out std_logic;
      exp_out: out std_logic_vector(3 downto 0);
      frac_out: out std_logic_vector(7 downto 0)
   );
end fp_adder ;

architecture arch of fp_adder is
   -- suffix b, s, a, n for
   --        big, small, aligned, normalized number
   signal signb, signs: std_logic;
   signal expb, exps, expn: unsigned(3 downto 0);
   signal fracb, fracs, fraca, fracn: unsigned(7 downto 0);
   signal sum_norm: unsigned(7 downto 0);
   signal exp_diff: unsigned(3 downto 0);
   signal sum: unsigned(8 downto 0); --one extra for carry
   signal lead0: unsigned(2 downto 0);
begin
   -- 1st stage: sort to find the larger number
   process (sign1, sign2, exp1, exp2, frac1, frac2)
   begin
      if (exp1 & frac1) > (exp2 & frac2) then
         signb <= sign1;
         signs <= sign2;
         expb <= unsigned(exp1);
         exps <= unsigned(exp2);
```

```
      fracb <= unsigned(frac1);
      fracs <= unsigned(frac2);
   else
      signb <= sign2;
      signs <= sign1;
      expb <= unsigned(exp2);
      exps <= unsigned(exp1);
      fracb <= unsigned(frac2);
      fracs <= unsigned(frac1);
   end if;
end process;

-- 2nd stage: align smaller number
exp_diff <= expb - exps;
with exp_diff select
   fraca <=
      fracs                              when "0000",
      "0"       & fracs(7 downto 1)      when "0001",
      "00"      & fracs(7 downto 2)      when "0010",
      "000"     & fracs(7 downto 3)      when "0011",
      "0000"    & fracs(7 downto 4)      when "0100",
      "00000"   & fracs(7 downto 5)      when "0101",
      "000000"  & fracs(7 downto 6)      when "0110",
      "0000000" & fracs(7)               when "0111",
      "00000000"                         when others;

-- 3rd stage: add/subtract
sum <= ('0' & fracb) + ('0' & fraca) when signb=signs else
       ('0' & fracb) - ('0' & fraca);

-- 4th stage: normalize
-- count leading 0s
lead0 <= "000" when (sum(7)='1') else
         "001" when (sum(6)='1') else
         "010" when (sum(5)='1') else
         "011" when (sum(4)='1') else
         "100" when (sum(3)='1') else
         "101" when (sum(2)='1') else
         "110" when (sum(1)='1') else
         "111";
-- shift significand according to leading 0
with lead0 select
   sum_norm <=
      sum(7 downto 0)                  when "000",
      sum(6 downto 0) & '0'            when "001",
      sum(5 downto 0) & "00"           when "010",
      sum(4 downto 0) & "000"          when "011",
      sum(3 downto 0) & "0000"         when "100",
      sum(2 downto 0) & "00000"        when "101",
      sum(1 downto 0) & "000000"       when "110",
      sum(0) &          "0000000"      when others;

-- normalize with special conditions
```

```
    process(sum,sum_norm,expb,lead0)
    begin
       if sum(8)='1' then -- w/ carry out; shift frac to right
          expn <= expb + 1;
          fracn <= sum(8 downto 1);
       elsif (lead0 > expb) then  -- too small to normalize;
          expn <= (others=>'0');  -- set to 0
          fracn <= (others=>'0');
       else
          expn <= expb - lead0;
          fracn <= sum_norm;
       end if;
    end process;

    -- form output
    sign_out <= signb;
    exp_out <= std_logic_vector(expn);
    frac_out <= std_logic_vector(fracn);
end arch;
```

The circuit in the first stage compares the magnitudes and routes the big number to the `signb`, `expb`, and `fracb` signals and the smaller number to the `signs`, `exps`, and `fracs` signals. The comparison is done between `exp1&frac1` and `exp2&frac2`. It implies that the exponents are compared first, and if they are the same, the significands are compared.

The circuit in the second stage performs alignment. It first calculates the difference between the two exponents, which is `expb-exps`, and then shifts the significand, `fracs`, to the right by this amount. The aligned significand is labeled `fraca`. The circuit in the third stage performs sign-magnitude addition, similar to that in Section 3.7.2. Note that the operands are extended by 1 bit to accommodate the carry-out bit.

The circuit in the fourth stage performs normalization, which adjusts the result to make the final output conform to the normalized format. The normalization circuit is constructed in three segments. The first segment counts the number of leading zeros. It is somewhat like a priority encoder. The second segment shifts the significands to the left by the amount specified by the leading-zero counting circuit. The last segment checks the carry-out and zero conditions and generates the final normalized number.

Testing circuit The floating-point adder has two 13-bit input operands. Since the prototyping board has only one 8-bit switch and four 1-bit pushbuttons, it cannot provide enough number of physical inputs to test the circuit. To accommodate the 26 bits of the floating-point adder, we must create a testing circuit and assign constants or duplicated switch signals to the adder's input operands. An example is shown in Listing 3.20. It assigns one operand as constant and uses duplicated switch signals for the other operand. The addition result is passed to the hexadecimal decoders and the sign circuit and is shown on the seven-segment LED display.

Listing 3.20 Floating-point adder testing circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity fp_adder_test is
   port(
```

```
        clk: in std_logic;
        sw: in std_logic_vector(7 downto 0);
        btn: in std_logic_vector(3 downto 0);
        an: out std_logic_vector(3 downto 0);
        sseg: out std_logic_vector(7 downto 0)
    );
end fp_adder_test;

architecture arch of fp_adder_test is
    signal sign1, sign2: std_logic;
    signal exp1, exp2: std_logic_vector(3 downto 0);
    signal frac1, frac2: std_logic_vector(7 downto 0);
    signal sign_out: std_logic;
    signal exp_out: std_logic_vector(3 downto 0);
    signal frac_out: std_logic_vector(7 downto 0);
    signal led3, led2, led1, led0:
              std_logic_vector(7 downto 0);
begin
    -- set up the fp adder input signals
    sign1 <= '0';
    exp1 <= "1000";
    frac1<= '1' &  sw(1) & sw(0) & "10101";
    sign2 <= sw(7);
    exp2 <= btn;
    frac2 <= '1' & sw(6 downto 0);

    -- instantiate fp adder
    fp_add_unit: entity work.fp_adder
       port map(
          sign1=>sign1, sign2=>sign2, exp1=>exp1, exp2=>exp2,
          frac1=>frac1, frac2=>frac2,
          sign_out=>sign_out, exp_out=>exp_out,
          frac_out=>frac_out
       );

    -- instantiate three instances of hex decoders
    -- exponent
    sseg_unit_0: entity work.hex_to_sseg
       port map(hex=>exp_out, dp=>'0', sseg=>led0);
    -- 4 LSBs of fraction
    sseg_unit_1: entity work.hex_to_sseg
       port map(hex=>frac_out(3 downto 0),
                dp=>'1', sseg=>led1);
    -- 4 MSBs of fraction
    sseg_unit_2: entity work.hex_to_sseg
       port map(hex=>frac_out(7 downto 4),
                dp=>'0', sseg=>led2);
    -- sign
    led3 <= "11111110" when sign_out='1' else -- middle bar
            "11111111";                       -- blank

    -- instantiate 7-seg LED display time-multiplexing module
    disp_unit: entity work.disp_mux
```

```
        port map(
60          clk=>clk, reset=>'0',
            in0=>led0, in1=>led1, in2=>led2, in3=>led3,
            an=>an, sseg=>sseg
        );
end arch;
```

3.8 BIBLIOGRAPHIC NOTES

The Designer's Guide to VHDL by P. J. Ashenden provides detailed coverage on the VHDL constructs discussed in this chapter, and the author's *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability* discusses the coding and optimization schemes and gives additional design examples.

3.9 SUGGESTED EXPERIMENTS

3.9.1 Multi-function barrel shifter

Consider an 8-bit shifting circuit that can perform rotating right or rotating left. An additional 1-bit control signal, `lr`, specifies the desired direction.

1. 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 code.
2. Derive a testbench and use simulation to verify operation of the code.
3. Synthesize the circuit, program the FPGA, and verify its operation.
4. This circuit can also be implemented by one rotate-right shifter with pre- and post-reversing circuits. The reversing circuit either passes the original input or reverses the input bitwise (for example, if an 8-bit input is $a_7a_6a_5a_4a_3a_2a_1a_0$, the reversed result becomes $a_0a_1a_2a_3a_5a_5a_6a_7$). Repeat steps 2 and 3.
5. Check the report files and compare the number of logic cells and propagation delays of the two designs.
6. Expand the code for a 16-bit circuit and synthesize the code. Repeat steps 1 to 5.
7. Expand the code for a 32-bit circuit and synthesize the code. Repeat steps 1 to 5.

3.9.2 Dual-priority encoder

A dual-priority encoder returns the codes of the highest or second-highest priority requests. The input is a 12-bit `req` signal and the outputs are `first` and `second`, which are the 4-bit binary codes of the highest and second-highest priority requests, respectively.

1. Design the circuit and derive the code.
2. Derive a testbench and use simulation to verify operation of the code.
3. Design a testing circuit that displays the two output codes on the seven-segment LED display of the prototyping board, and derive the code.
4. Synthesize the circuit, program the FPGA, and verify its operation.

3.9.3 BCD incrementor

The binary-coded-decimal (BCD) format uses 4 bits to represent 10 decimal digits. For example, 259_{10} is represented as "0010 0101 1001" in BCD format. A BCD incrementor

adds 1 to a number in BCD format. For example, after incrementing, "0010 0101 1001" (i.e., 259_{10}) becomes "0010 0110 0000" (i.e., 260_{10}).

1. Design a three-digit 12-bit incrementor and derive the code.
2. Derive a testbench and use simulation to verify operation of the code.
3. Design a testing circuit that displays three digits on the seven-segment LED display and derive the code.
4. Synthesize the circuit, program the FPGA, and verify its operation.

3.9.4 Floating-point greater-than circuit

A floating-point greater-than circuit compares two floating-point numbers and asserts output, `gt`, when the first number is larger than the second number. Assume that the two numbers are represented in the format discussed in Section 3.7.4.

1. Design the circuit and derive the code.
2. Derive a testbench and use simulation to verify operation of the code.
3. Design a testing circuit and derive the code.
4. Synthesize the circuit, program the FPGA, and verify its operation.

3.9.5 Floating-point and signed integer conversion circuit

A number may need to be converted to different formats in a large system. Assume that we use the 13-bit format in Section 3.7.4 for the floating-point representation and the 8-bit `signed` data type for the integer representation. An integer-to-floating-point conversion circuit converts an 8-bit integer input to a normalized, 13-bit floating-point output. A floating-point-to-integer conversion circuit reverses the operation. Since the range of a floating-point number is much larger, conversion may lead to the underflow condition (i.e., the magnitude of the converted number is smaller than "00000001") or the overflow condition (i.e., the magnitude of the converted number is larger than "01111111").

1. Design an integer-to-floating-point conversion circuit and derive the code.
2. Derive a testbench and use simulation to verify operation of the code.
3. Design a testing circuit and derive the code.
4. Synthesize the circuit, program the FPGA, and verify its operation.
5. Design a floating-point-to-integer conversion circuit. In addition to the 8-bit integer output, the design should include two status signals, `uf` and `of`, for the underflow and overflow conditions. Derive the code and repeat steps 2 to 4.

3.9.6 Enhanced floating-point adder

The floating-point adder in Section 3.7.4 discards the lower bits when they are shifted out (it is known as *round to zero*). A more accurate method is to *round to the nearest even*, as defined in the *IEEE Standard for Binary Floating-Point Arithmetic* (IEEE Std 754). Three extra bits, known as the *guard*, *round*, and *sticky bits*, are required to implement this method. If you learned floating-point arithmetic before, modify the floating-point adder in Section 3.7.4 to accommodate the round-to-the-nearest-even method.