CHAPTER 3

RT-LEVEL COMBINATIONAL CIRCUIT

3.1 INTRODUCTION

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

3.2 OPERATORS

Verilog consists of about two dozen operators. In addition to the bitwise operators discussed in Chapter 1, there are arithmetic, shift, and relational operators. These operators correspond to intermediate-sized components, such as adders and comparators. We examine these operators in this section and also cover miscellaneous synthesis-related Verilog constructs. Table 3.1 summarizes the operators.

Table 3.1 Verilog operators

Type of operation	Operator symbol	Description	Number of operands
Arithmetic	+	addition	2
	-	subtraction	2
	*	multiplication	2
	/	division	2
	%	modulus	2
	**	exponentiation	2
Shift	>>	logical right shift	2
	<<	logical left shift	2
	>>>	arithmetic right shift	2
	<<<	logical left shift	2
Relational	>	greater than	2
	<	less than	2
	>=	greater than or equal to	2
	<=	less than or equal to	2
Equality	==	equality	2
	!=	inequality	2
	===	case equality	2
	!==	case inequality	2
Bitwise	~	bitwise negation	1
	&	bitwise and	2
	|	bitwise or	2
	^	bitwise xor	2
Reduction	&	reduction and	1
	|	reduction or	1
	^	reduction xor	1
Logical	!	logical negation	1
	&&	logical and	2
	||	logical or	2
Concatenation	{ }	concatenation	any
	{ { } }	replication	any
Conditional	? :	conditional	3

Table 3.2 Operator precedence

Operator	Precedence
! ~ + - (unary)	highest
**	
* / %	
+ - (binary)	
>> << >>> <<<	
< <= > >=	
== != === !==	
&	
^	
|	
&&	
||	
?:	lowest

3.2.1 Arithmetic operators

There are six arithmetic operators: +, -, *, /, %, and **. They represent addition, subtraction, multiplication, division, modulus, and exponentiation operations, respectively. The + and - operators can also be used as unary operators, as in -a. During synthesis, the + and - operators infer the adder and subtractor and they are synthesized by FPGA's logic cells.

Multiplication is a complicated operation and synthesis of the multiplication operator * depends on synthesis software and target device technology. The 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. Although 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**

The /, %, and ** operators usually cannot be synthesized automatically.

3.2.2 Shift operators

There are four shift operators: >>, <<, >>>, and <<<. The first two represent the logical shift right and left and the last two represent the arithmetic shift right and left.

The 0's are shifted in for a logical shift operation (i.e., >> and <<). The sign bits (i.e., the MSB) are shifted in for the >>> operation and the 0's are shifted in for the <<< operation. Note that there is no difference between the << and <<< operations. The latter is included for completeness. Some shifting examples are shown in Table 3.3.

If both operands of a shift operator are signals, as in a << b, the operator infers a *barrel shifter*, which is a fairly complex circuit. On the other hand, if the shifted amount is fixed, as in a << 2, the operation infers no logic and involves only routing of the input signals.

Table 3.3 Shift operation examples

a	a >> 2	a >>> 2	a << 2	a <<< 2
0100_1111	0001_0011	0001_0011	0011_1100	0011_1100
1100_1111	0011_0011	1111_0011	0011_1100	0011_1100

This type of operation can also be described by using the catenation operator discussed in Section 3.2.5.

3.2.3 Relational and equality operators

There are four relational operators: >, <, <=, and >=. These operators compare two operands and return a Boolean result, which can be *false* (represented by 1-bit scalar value 0) or *true* (represented by 1-bit scalar value 1).

There are four equality operators: ==, !=, ===, and !==. As with the relational operators, they return *false* (1-bit 0) or *true* (1-bit 1). The === and !== operators, known as *case equality* and *case inequality* operators, take into consideration of the matches of the x and z bits in the operands. They cannot be synthesized.

The relational operators and the == and != operators infer comparators during synthesis.

3.2.4 Bitwise, reduction, and logical operators

The bitwise, reduction, and logical operators are somewhat similar and perform the and, or, xor, as well as not operations. These operators are implemented by basic logic cells.

Bitwise operators There are four basic bitwise operators: & (and), | (or), ^ (xor), and ~ (not). The first three operators require two operands. Negation and xor operation can be combined, as in ~^ or ^~, to form the xnor operator. The operations are performed on a bit-by-bit basis and thus are known as bitwise operators. For example, let a, b, and c be 4-bit signals:

```
wire [3:0] a, b, c;
```

The statement

```
assign  c = a | b;
```

is the same as

```
assign  c[3] = a[3] | b[3];
assign  c[2] = a[2] | b[2];
assign  c[1] = a[1] | b[1];
assign  c[0] = a[0] | b[0];
```

Reduction operators The previous &, |, and ^ operators may have only one operand and then are known as reduction operators. The single operand usually has an array data type. The designated operation is performed on all elements of the array and returns a 1-bit result. For example, let a be a 4-bit signal and y be a 1-bit signal:

```
wire [3:0] a;
wire y;
```

Table 3.4 Logical and bitwise operation examples

a	b	a&b	a|b	a&&b	a||b
0	1	0	1	0 (false)	1 (true)
000	000	000	000	0 (false)	0 (false)
000	001	000	001	0 (false)	1 (true)
011	001	001	011	1 (true)	1 (true)

The statement

```
assign  y = | a;  // only one operand
```

is the same as

```
assign  y = a[3] | a[2] | a[1] | a[0];
```

Logical operators There are three logical operators: && (logical and), || (logical or), and ! (logical negate). The logical operators are different from the bitwise operators. If we assume that no x or z is used, the operands of a logical operator are interpreted as false (when all bits are 0's) or true (when at least one bit is 1), and the operation always returns a 1-bit result. As the name suggests, the logical operators should be used as logical connectives of Boolean expressions, as in

```
(state==idle) || ((state==op) && (count>10))
```

Some examples are shown in Table 3.4. The corresponding bitwise operations are also included to illustrate the difference between the two types of operations. Since Verilog uses 0 and 1 to represent the false and true values, bitwise and logical operators can be used interchangeably in some situations. However, it is good practice to use logical operators for Boolean expressions and use bitwise operators for signal manipulation.

3.2.5 Concatenation and replication operators

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

```
wire a1;
wire [3:0] a4;
wire [7:0] b8, c8, d8;
. . .
assign b8 = {a4, a4};
assign c8 = {a1, a1, a4, 2'b00};
assign d8 = {b8[3:0], c8[3:0]};
```

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

One application of the concatenation operator is to shift and rotate a signal by a fixed amount, as shown in the following example:

```
wire [7:0] a;
wire [7:0] rot, shl, sha;
. . .
```

```
// rotate a to right 3 bits
assign rot = {a[2:0], a[8:3]};
// shift a to right 3 bits and insert 0 (logic shift)
assign shl = {3'b000, a[8:3]};
// shift a to right 3 bits and insert MSB
// (arithmetic shift)
assign sha = {a[8], a[8], a[8], a[8:3]};
```

The concatenation operator, `N{ }`, replicates the enclosed string. The replication constant, `N`, specifies the number of replications. For example, `{4{2'b01}}` returns `8'b01010101`. The previous arithmetic shift operation can be simplified:

```
assign sha = {3{a[8]}, a[8:3]};
```

3.2.6 Conditional operators

The conditional operator, `?:`, takes three operands and its general format is

```
[signal] = [boolean_exp] ? [true_exp] : [false_exp];
```

The `[boolean_exp]` is a Boolean expression that returns true (`1'b1`) or false (`1'b0`). The `[signal]` gets `[true_exp]` if it is true and `[false_exp]` if it is false. For example, the following circuit obtains the maximum of `a` and `b`:

```
assign max = (a>b) ? a : b;
```

The operator can be thought as a simplified if-then-else statement:

```
if [boolean_exp] then
   [signal] = [true_exp];
else
   [signal] = [false_exp];
```

Despite its simplicity, the conditional operators can be cascaded or nested to specify the desired selection. For example, the eq1 circuit described in Table 1.1 can be rewritten using conditional operators:

```
assign eq = (~i1 & ~i0) ? 1'b1 :
            (~i1 & i0)  ? 1'b0 :
            (i1 & ~i0)  ? 1'b0 :
            1'b1;
```

We can extend the maximal circuit to return the maximum of `a`, `b`, and `c`:

```
assign max = (a>b) ? ((a>c) ? a : c) :
                     ((b>c) ? b : c);
```

While synthesized, a conditional operator infers a 2-to-1 multiplexing circuit. The detailed derivation is discussed in Section 3.6.

3.2.7 Operator precedence

The operator precedence specifies the order of evaluation. The precedence is shown in Table 3.2. When an expression is evaluated, the operator with higher precedence is evaluated first. For example, in the `a + b >> 1` expression, `a + b` is evaluated first and then `>> 1` is evaluated. We can use parentheses to alter the precedence, as in `a + (b >> 1)`. It is a good practice to use parentheses to make an expression clearer, as in `(a + b) >> 1`, even when they are not required.

3.2.8 Expression bit-length adjustment

As signals in real hardware, nets and variables in a Verilog program usually have different numbers of bits (i.e., *bit lengths* or *widths*). In a Verilog statement, the bit lengths of operands can be different and the adjustment is determined by a set of implicit rules:

- Determine the maximal bit length of the operands in the context, which includes the right-hand-side expression and the left-hand-side signal.
- Extend the bit lengths of operands on the right-hand side to the maximum and evaluate the expression.
- Assign the result to the left-hand-side signal. Truncate the MSBs if the signal's bit length is smaller.

Let us first consider a simple example:

```
wire [7:0] a, b;

assign a = 8'b00000000;
assign b = 0;
```

The first statement assigns an 8-bit value, "00000000", to a. The second statement assigns the integer 0 to b. Recall that the integer in Verilog is 32 bits and thus 0 is represented as "00000000000000000000000000000000". Since b is 8 bits wide, it is truncated to "00000000" during the assignment. Although both statements assign an all-zero pattern to the signals, we need to be aware of how the values are obtained.

Let us consider another example:

```
wire [7:0] a, b;
wire [7:0] sum8;
wire [8:0] sum9;

assign sum8 = a + b;
assign sum9 = a + b;
```

In the first assignment, all operands are 8 bits wide and an 8-bit addition is performed. The carry-out bit of the addition is discarded. In the second assignment, the a and b signals are extended to 9 bits, the bit length of the sum9 signal, and a 9-bit addition is performed. The sum[9] bit gets the resulting carry-out bit. We can also use a concatenation operator if an explicit carry-out signal is desired:

```
assign {c_out,sum8} = a + b;
```

Although the basic conversion rule is simple and intuitive, the subtleties can be error-prone. For example, let a, b, sum1, and sum2 be 8-bit signals. The following statements give a different result:

```
// shift 0 to MSB of sum1
assign sum1 = (a + b) >> 1;
// shift carry-out of a+b to MSB of sum2
assign sum2 = (0 + a + b) >> 1;
```

In the first assignment, all operands are 8 bits wide and an 8-bit addition is performed. The carry-bit is discarded. When the shift operation is performed, 0 is shifted into the MSB. In the second assignment, 0 is an integer and thus is 32 bits wide. The a and b are extended to 32 bits for addition and the summation is shifted. The result is then truncated to 8 bits when assigned to sum2 and sum2[7] gets the original carry-out bit. The conversion becomes more involved when the **signed** data type is used (discussed in Section 7.3).

oe	y
0	Z
1	a_in

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

A safe but somewhat cumbersome alternative is to adjust the bit lengths of the operands manually. For example, an alternative that may be used to obtain `sum2` is

```
wire [8:0] sum_ext;          // extend sum to 9 bits
. . .
assign sum_ext = {1'b0,a} + {1'b0,b};
assign sum2 = sum_ext[9:1];
```

The code is longer but is more descriptive and less prone to error.

In summary, we must be aware of the Verilog's automatic bit-length adjustment mechanism. Unintended bit-length mismatch may lead to subtle, difficult-to-find errors. Except for trivial adjustments, such as assigning an all-zero pattern with an integer 0, we should either adjust the bit lengths manually or thoroughly document the desired automatic adjustment.

3.2.9 Synthesis of z and x values

In addition to the regular logic 0 and logic 1, net and variable can contain `z` and `x` values. Although they are not operators, we discuss the synthesis aspect of these two values in this subsection.

Synthesis of z The `z` value 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. The operation of the buffer is controlled by an enable signal, `oe` (for "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

```
assign y = (oe) ? a_in : 1'bz;
```

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 a 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:

```
module bi_demo(
       inout wire bi,
       . . .
   )

   assign sig_out = output_expression;
```

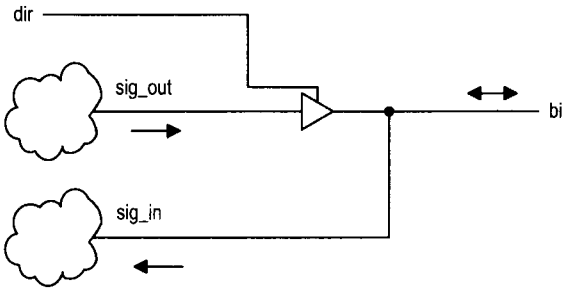


Figure 3.2 Single-buffer bidirectional I/O port.

Table 3.5 Truth table with don't-care

input i	output y
0 0	0
0 1	1
1 0	1
1 1	x

```
. . .
assign some_signal = expression_with_sig_in;
. . .
assign bi = (dir) ?  sig_out : 1'bz;
assign sig_in = bi;
. . .
```

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

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 be used only for I/O ports that are mapped to the physical pins of an FPGA device. **Xilinx specific**

Synthesis of `x` In some combinational circuits, certain input patterns may never occur and thus the output value is irrelevant. We frequently assign a "don't-care" value to the output. During synthesis, the don't-care will be assigned a value (either 0 or 1) that can help the optimization process. Consider the truth table shown in Table 3.5. We assume that the `i` will never be `11` and thus the corresponding output is specified as don't-care. In synthesis, we can use `x` for the don't-care value. One possible code for the previous table is

```
assign y = (i==2'b00) ? 1'b0 :
           (i==2'b01) ? 1'b1 :
           (i==2'b10) ? 1'b1 :
           1'bx;   // i==2'b11
```

Although this approach helps to minimize the circuit, it introduces a discrepancy between simulation and synthesis. In simulation, `x` is a unique value rather than "0 or 1". If the input is `11` in simulation, the output becomes `x` and is not consistent with the synthesized result (which can be either 0 or 1). However, since the `11` pattern should never occur in the

original specification, the appearance of the x value can be used to signal potential errors in the testbench.

3.3 ALWAYS BLOCK FOR A COMBINATIONAL CIRCUIT

To facilitate system modeling, Verilog contains a number of *procedural statements*, which are executed in sequence. Since their behavior is different from the normal concurrent circuit model, these statements are encapsulated inside an *always block* or *initial block*. The initial block is executed once when the simulation is started. It can be used in simulation, as in the testbench example in Listing 1.7. Only the always block can be synthesized and it is discussed in this section. Since the procedural statement is more abstract, this type of code is sometimes known as *behaviorial* description.

An always block can be thought of as a black box whose behavior is described by the internal procedural statements. Procedural statements include a rich variety of constructs but many of them don't have clear hardware counterparts. A poorly coded always block frequently leads to unnecessarily complex implementation or cannot be synthesized at all. The focus of this section is on the synthesis of combinational circuits and we limit the discussion to three types of statements:

- Blocking procedural assignment
- If statement
- Case statement

The latter two can be considered as constructs that infer routing structure.

3.3.1 Basic syntax and behavior

The simplified syntax of an always block with a *sensitivity list* (also known as *event control expression*) is

```
always @([sensitivity_list])
begin [optional name]
   [optional local variable declaration];

   [procedural statement];
   [procedural statement];
  . . .
end
```

The `[sensitivity_list]` term is a list of signals and events to which the always block responds (i.e., is "sensitive to"). For a combinational circuit, all the input signals should be included in this list. The body is composed of any number of procedural statements. The **begin** and **end** delimiters can be omitted if there is only one procedural statement in the body. The `@([sensitivity_list])` term is actually a timing control construct. It is usually the *only* timing control construct in a synthesizable always block.

An always block can be considered as a complex circuit part. It can be suspended or activated. When any signal of the sensitivity list changes or an event occurs, the part is activated and executes the internal procedural statements. Since there is no other timing control construct, the execution continues to the end and the part is suspended. Thus, an always block actually "loops forever" and the initiation of each loop is controlled by the sensitivity list.

3.3.2 Procedural assignment

A procedural assignment can only be used within an always block or initial block. There are two types of assignments: *blocking assignment* and *nonblocking assignment*. Their basic syntax is

```
[variable_name] = [expression];    // blocking assignment
[variable_name] <= [expression];   // nonblocking assignment
```

In a blocking assignment, the expression is evaluated and then assigned to the variable immediately, before execution of the next statement (the assignment thus "blocks" the execution of other statements). It behaves like the normal variable assignment in the C language. In a nonblocking assignment, the evaluated expression is assigned at the end of the always block (the assignment thus does not block the execution of other statements).

The blocking and nonblocking assignments frequently confuse new Verilog users and failing to comprehend their differences can lead to unexpected behavior or race conditions. The basic rule of thumb is:

- *Use blocking assignments for a combinational circuit.*
- *Use nonblocking assignments for a sequential circuit.*

This topic is explained in detail in Section 7.1. Since we focus on combinational circuits in this chapter, only the blocking statement is used.

3.3.3 Variable data types

In a procedural assignment, an expression can only be assigned to an output with one of the *variable* data types, which are **reg**, **integer**, **real**, **time**, and **realtime**. The **reg** data type is like the **wire** data type, but used with a procedural output. The **integer** data type represents a fixed-size (usually 32 bits) signed number in 2's-complement format. Since its size is fixed, we usually don't use it in synthesis. The other data types are for modeling and simulation and cannot be synthesized.

3.3.4 Simple examples

We use two simple examples to illustrate the use and behavior of the always block and procedural blocking assignment.

1-bit comparator We can rewrite the previous 1-bit comparator circuit in Listing 1.1 using an always block. The code is shown in Listing 3.1.

Listing 3.1 Always block implementation of a 1-bit comparator

```
module eq1_always
  (
   input wire i0, i1,
   output reg eq  // eq declared as reg
);

  // p0 and p1 declared as reg
  reg p0, p1;

  always @(i0, i1) // i0 an i1 must be in sensitivity list
  begin
```

```
    // the order of statements is important
    p0 = ~i0 & ~i1;
    p1 = i0 & i1;
    eq = p0 | p1;
  end

endmodule
```

Since the eq, p0, and p1 signals are assigned within the always block, they are declared as the **reg** data type. The sensitivity list consists of i0 and i1, which are separated by a comma. When one of them changes, the always block is activated. The three blocking assignments are executed sequentially, much like the statements in a C program. The order of the statements is important and p0 and p1 must be assigned values before being used.

FYI

In Verilog-1995, the keyword **or** is used in place of the comma in a sensitivity list. For example, the list

```
always @(a, b, c)
```

is written as

```
always @(a or b or c)
```

We use only commas in this book.

A combinational circuit must include all its input signals in the sensitivity list to correctly model the desired behavior. Missing a signal can lead to discrepancy between synthesis and simulation. In Verilog-2001, we can use the notation

```
always @*
```

to implicitly include all the input signals. In this book, we use this construct for the combinational circuit.

Three-input and circuit The similarity of the codes in Listings 1.1 and 3.1 is somewhat misleading. The behavior of continuous assignments and procedural statements is quite different.

Consider the code in Listing 3.2. It is a circuit that performs an and operation over a, b, and c (i.e., a & b & c).

Listing 3.2 Behaviorial reduced and circuit using a variable

```
module and_block_assign
  (
  input wire a, b, c,
  output reg y
  );

  always @*
  begin
    y = a;
    y = y & b;
    y = y & c;
  end

endmodule
```

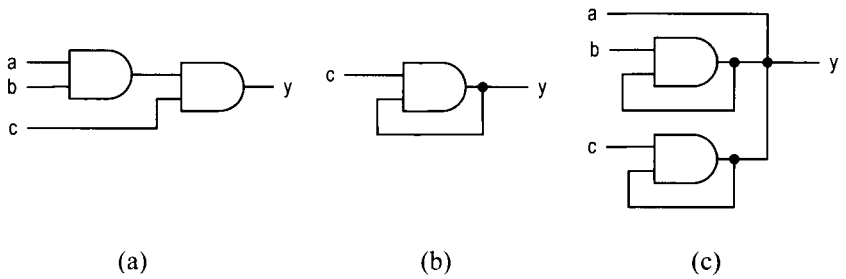


Figure 3.3 Circuits inferred from correct and incorrect code segments.

The inferred circuit is shown in Figure 3.3(a). If we use continuous assignments in a similar way, as shown in Listing 3.3, the description is incorrect.

Listing 3.3 Incorrect code for a reduced and circuit

```
module and_cont_assign
  (
  input wire a, b, c,
  output wire y
);

  assign y = a;
  assign y = y & b;
  assign y = y & c;

endmodule
```

In this code, each continuous assignment infers a circuit part. The three appearances of y on the left-hand side imply that the three outputs are tied together. The corresponding circuit diagram is shown in Figure 3.3(c) and it is clearly not the desired circuit.

3.4 IF STATEMENT

3.4.1 Syntax

The simplified syntax of an if statement is

```
if [boolean_expr]
   begin
      [procedural statement];
      [procedural statement];
      . . .
   end
else
   begin
      [procedural statement];
      [procedural statement];
      . . .
   end
```

Table 3.6 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

The `[boolean_expr]` term is a Boolean expression and is evaluated first. If it is true, the statements in the following branch are executed. Otherwise, the statements in the else branch are executed. The else branch is optional and can be omitted. The **begin** and **end** delimiters can be omitted if there is only one procedural statement in a branch.

Multiple if statements can be "cascaded" to evaluate multiple Boolean conditions and establish priorities, as in

```
if [boolean_expr_1]
   . . .
else if [boolean_expr_2]
   . . .
else if [boolean_expr_3]
   . . .
else
   . . .
```

When synthesized, the if statements infer "priority routing" networks. This topic is discussed in Section 3.6.

3.4.2 Examples

We use two simple examples to demonstrate use of the if 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 3.6. The HDL code is shown in Listing 3.4.

Listing 3.4 Priority encoder using an if statement

```
module prio_encoder_if
   (
    input wire [4:1] r,
    output reg [2:0] y
   );

   always @*
      if (r[4]==1'b1)       // can be written as (r[4])
         y = 3'b100;
      else if (r[3]==1'b1) // can be written as (r[3])
         y = 3'b011;
      else if (r[2]==1'b1) // can be written as (r[2])
```

Table 3.7 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

```
      y = 3'b010;
   else if (r[1]==1'b1) // can be written as (r[1])
      y = 3'b001;
   else
      y = 3'b000;

endmodule
```

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. Note that the Boolean expression `(r[4]==1'b1)` is true when `r[4]` is 1. Since the true value is also expressed as `1'b1` in Verilog, the expression can be written as `(r[4])` as well.

The second example is a binary decoder. An n-to-2^n binary decoder asserts one 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 3.7. The circuit also has a control signal, `en`, which enables the decoding function when asserted. The HDL code is shown in Listing 3.5.

Listing 3.5 Binary decoder using an if statement

```
module decoder_2_4_if
   (
    input wire [1:0] a,
    input wire en,
    output reg [3:0] y
   );

   always @*
      if (en==1'b0)          // can be written as (~en)
         y = 4'b0000;
      else if (a==2'b00)
         y = 4'b0001;
      else if (a==2'b01)
         y = 4'b0010;
      else if (a==2'b10)
         y = 4'b0100;
      else
         y = 4'b1000;

endmodule
```

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. Note that the Boolean expression (en==1'b0) can be written as (~en) as well.

3.5 CASE STATEMENT

3.5.1 Syntax

The simplified syntax of a case statement is

```
case [case_expr]
   [item]:
      begin
         [procedural statement];
         [procedural statement];
         . . .
      end
   [item]:
      begin
         [procedural statement];
         [procedural statement];
         . . .
      end
   [item]:
      begin
         [procedural statement];
         [procedural statement];
         . . .
      end
   . . .
   default:
      begin
         [procedural statement];
         [procedural statement];
         . . .
      end
   endcase
```

A case statement is a multiway decision statement that compares the [case_expr] expression with a number of [item] expressions. The execution jumps to the branch whose [item] matches the current value of [case_expr]. If there are multiple matched [item] expressions, execution jumps to the branch of the first match. The last item can be an optional **default** keyword. It covers all the unspecified values of the [case_expr] expression. The **begin** and **end** delimiters can be omitted if there is only one procedural statement in a branch.

3.5.2 Examples

We use the same priority encoder and decoder examples to demonstrate use of the case statement. The functional table of a 2-to-4 decoder is shown in Table 3.7. The HDL code using a case statement is shown in Listing 3.6.

Listing 3.6 Binary decoder using a case statement

```
module decoder_2_4_case
   (
    input wire [1:0] a,
    input wire en,
    output reg [3:0] y
   );

   always @*
      case({en,a})
         3'b000, 3'b001, 3'b010, 3'b011: y = 4'b0000;
         3'b100: y = 4'b0001;
         3'b101: y = 4'b0010;
         3'b110: y = 4'b0100;
         3'b111: y = 4'b1000; // default can also be used
      endcase

endmodule
```

We can group multiple values into one item expression, as in line 10, if the identical statements are used for these values. Note that all possible values of the {en,a} expression are covered by the item expressions.

The function table of the priority encoder is shown in Table 3.6. The HDL code is shown in Listing 3.7.

Listing 3.7 Priority encoder using a case statement

```
module prio_encoder_case
   (
    input wire [4:1] r,
    output reg [2:0] y
   );

   always @*
      case(r)
         4'b1000, 4'b1001, 4'b1010, 4'b1011,
         4'b1100, 4'b1101, 4'b1110, 4'b1111:
            y = 3'b100;
         4'b0100, 4'b0101, 4'b0110, 4'b0111:
            y = 3'b011;
         4'b0010, 4'b0011:
            y = 3'b010;
         4'b0001:
            y = 3'b001;
         4'b0000:      // default can also be used
            y = 3'b000;
      endcase

endmodule
```

3.5.3 The casez and casex statements

There are two variations in addition to the regular **case** statement. In a **casez** statement, the z value and the ? character in the item expression are treated as don't-care (i.e., the corresponding bit does not need to be matched). In a **casex** statement, the z and x values and the ? character in the item expression are treated as don't-care. Since the z and x values may appear in simulation, the ? character is preferred.

For example, the previous priority encoder can be coded with a **casez** statement, as shown in Listing 3.8.

Listing 3.8 Priority encoder using a casez statement

```
module prio_encoder_casez
   (
    input wire [4:1] r,
    output reg [2:0] y
   );

   always @*
      casez(r)
         4'b1???: y = 3'b100;
         4'b01??: y = 3'b011;
         4'b001?: y = 3'b010;
         4'b0001: y = 3'b001;
         4'b0000: y = 3'b000; // default can also be used
      endcase

endmodule
```

3.5.4 The full case and parallel case

In Verilog, the item expressions do not need to include all values of the [case_expr] expression and some values can be matched more than once. Consider the following **casez** statement:

```
reg [2:0] s
. . .
casez (s)
   3'b111: y = 1'b1;
   3'b1??: y = 1'b0;
   3'b000: y = 1'b1;
endcase
```

In this statement, the value 3'b111 is matched twice in the item expressions (once in 3'b111 and once in 3'b1??). Since the first match takes effect, y gets 1'b1 if s is 3'b111. If s is 3'b001, 3'b010, or 3'b011, there are no matches and y will "keep its previous value."

When all possible binary values of the [case_expr] expression are covered by the item expressions, the statement is known as a *full case* statement. For a combinational circuit, we must use a full case statement since each input combination should have an output value. We can add the **default** item to cover all the unmatched values. For example, the previous statement can be revised either as

```
casez (s)
   3'b111: y = 1'b1;
   3'b1??: y = 1'b0;
   default: y = 1'b1; // y gets 1 for unspecified values
endcase
```

or as

```
casez (s)
   3'b111: y = 1'b1;
   3'b1??: y = 1'b0;
   3'b000: y = 1'b1;
   default: y = 1'bx; // y gets don't-care
endcase
```

When the values in the item expressions are mutually exclusive (i.e., a value appears in only one item expression), the statement is know as a *parallel case* statement. For example, the previous casez statement is not a parallel case statement since the value `3'b111` appears twice. The case statements of Listings 3.6 and 3.7 are parallel case statements.

When synthesized, a parallel case statement usually infers a multiplexing routing network and a non-parallel case statement usually infers a priority routing network. This topic is discussed in the next section.

Many synthesis software packages have "full case directive" and "parallel case directive." When they are used, all case statements are treated as full case statements and parallel case statements and synthesized accordingly. Verilog-2001 has similar attributes for this purpose. Using these directives essentially overrides original semantics of Verilog code and introduces a discrepancy between simulation and synthesis. In this book, we express these conditions in code rather than applying these directives or attributes. **FYI**

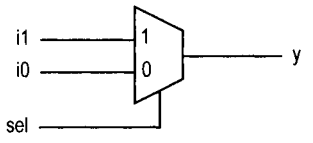
3.6 ROUTING STRUCTURE OF CONDITIONAL CONTROL CONSTRUCTS

We examine several conditional control language constructs, including the `?:` operator and the if and case statements. In the C language, these constructs are executed sequentially. There is no "sequential" control in a combinational circuit. These constructs are realized by *routing networks*. All expressions are evaluated concurrently and the routing network routes the desired result to the output. There are two types of routing structures: *priority routing network* and *multiplexing network*, which are inferred by an if-else type statement and a parallel case statement, respectively.

3.6.1 Priority routing network

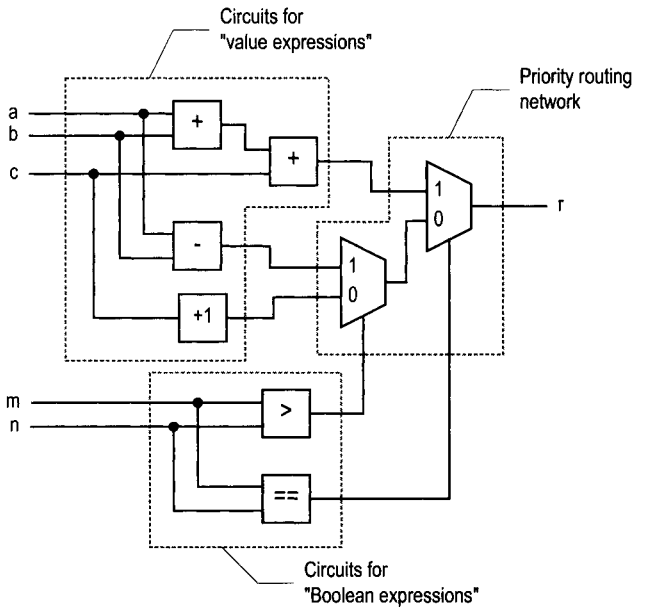
A priority routing network is implemented by a sequence of 2-to-1 multiplexers. The diagram and truth table of a 2-to-1 multiplexer are shown in Figure 3.4(a). An if-else statement implies a priority routing network. Consider the following statement:

```
if (m==n)
   r = a + b + c;
else if (m > n)
   r = a - b;
else
   r = c + 1;
```



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

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



(b) Diagram of an if statement

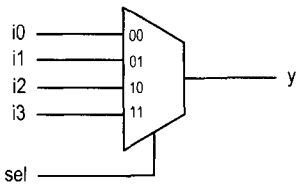
Figure 3.4 Implementation of an if statement.

The conceptual diagram of the statement is shown in Figure 3.4(b). The two 2-to-1 multiplexers form the priority routing network and other components implement various Boolean and arithmetic expressions. If 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 port 0 is passed to `r`. The next Boolean condition (i.e., `m>n`) determines whether the result of `a-b` or `c+1` is routed to the output.

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

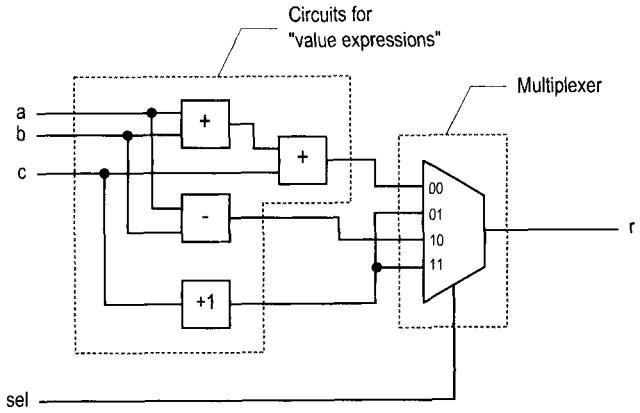
The conditional operator (`?:`) is like a simplified if-else statement and infers similar priority routing networks. A non-parallel case statement sets a preference on the first matched item and thus also infers similar priority routing networks. For example, consider the following case statement:

```
case (expr)
   item1: statement1;
   item2: statement2;
```



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 parallel case statement

Figure 3.5 Implementation of a parallel case statement.

```
    item3: statement3;
    default: statement4;
endcase
```

It can be translated to

```
if [expr==item1]
    statement1;
else if [expr==item2]
    statement2;
else if [expr==item3]
    statement3;
else
    statement4;
```

3.6.2 Multiplexing network

A multiplexing network is implemented by an n-to-1 multiplexer. The desired input port is specified by the selection signal and the corresponding input is routed to the output. The diagram and functional table of the 2^2-to-1 multiplexer are shown in Figure 3.5(a).

In a parallel case statement, we can map each value of the case expression to an input port of the multiplexer and connect the corresponding evaluated result to the port. The case expression is connected to the selection signal. The construction can best be explained by an example. Consider the following case statement:

```
wire [1:0] sel;
. . .
case (sel)
   2'b00: r = a + b + c;
   2'b10: r = a - b;
   default: r = c + 1; // 2'b01, 2'b11
endcase
```

The conceptual diagram of this statement is shown in Figure 3.5(b). The `sel` varaible 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 evaluated result of `a+b+c` is routed to `r` when `sel` is 00, the result of `a-b` is routed to `r` when `sel` is 10, and the result of `c+1` is routed to `r` when `sel` is 01 or 11.

Again, note that all value expressions are evaluated concurrently. The `sel` variable 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 `sel`.

In general, the priority routing network is more effective when a preference is given under certain conditions, such as for a priority encoder, and the multiplexing network is more effective for a truth table or function table–based description, such as for a binary decoder.

3.7 GENERAL CODING GUIDELINES FOR AN ALWAYS BLOCK

Verilog is for both modeling and synthesis. While writing code for synthesis, we need to be aware of how the various language constructs are mapped to hardware. This is especially true for an always block since variables and procedural statements can be used within the block. We should remember that the purpose of the code is to infer hardware rather than describing a sequential algorithm in C. Failing to do so frequently leads to unsynthesizable codes, unnecessarily complex implementation, or discrepancy between simulation and synthesis. In this section, we review some common errors and suggest a collection of coding guidelines.

3.7.1 Common errors in combinational circuit codes

Following are common errors found in combinational circuit codes:

- Variable assigned in multiple always blocks
- Incomplete sensitivity list
- Incomplete branch and incomplete output assignment

These errors are discussed below.

Variable assigned in multiple always blocks In Verilog, variables can be assigned (i.e., appear on the left-hand side) in multiple always blocks. For example, the `y` variable is shared by two always blocks is the following code segment:

```
reg y;
reg a, b, clear;
. . .
always @*
   if (clear) y = 1'b0;
```

```
always @*
   y = a & b;
```

Although the code is syntactically correct and can be simulated, it cannot be synthesized. Recall that each always block can be interpreted as a circuit part. The code above indicates that y is the output of both circuit parts and can be updated by each part. No physical circuit exhibits this kind of behavior and thus the code cannot be synthesized. We must group the assignment statements in a single always block, as in

```
always @*
   if (clear)
      y = 1'b0;
   else
      y = a & b;
```

Incomplete sensitivity list For a combinational circuit, the output is a function of input and thus any change in an input signal should activate the circuit. This implies that all input signals should be included in the sensitivity list. For example, a two-input and gate can be written as

```
always @(a, b)    // both a and b are in sensitivity list
   y = a & b;
```

If we forget to include b, the code becomes

```
always @(a)       // a missing from sensitivity list
   y = a & b;
```

Although the code is still syntactically correct, its behavior is very different. When a changes, the always block is activated and y gets the value of a&b. When b changes, the always block remains suspended since it is not "sensitive to" b and y keeps its previous value. No physical circuit exhibits this kind of behavior. Most synthesis software will issue a warning message and infer the and gate instead. However, the simulation software still models the intended behavior and hence introduces a discrepancy between simulation and synthesis.

In Verilog-2001, a special notation, @*, is introduced to implicitly include all the relevant input signals and thus eliminates this problem. It is a good practice to use this notation for combinational circuit description.

Incomplete branch and incomplete output assignment The output of a combinational circuit is a function of input only and the circuit should not contain any *internal state* (i.e., *memory*). One common error with an always block is the inference of unintended memory in a combinational circuit. The Verilog standard specifies that a variable will *keep its previous value* if it is not assigned a value in an always block. During synthesis, this infers an internal state (via a closed feedback loop) or a memory element (such as a latch).

To prevent unintended memory in an always block, all output signals must be assigned proper values all the time. *Incomplete branch* and *incomplete output assignment* are two common errors that lead to unintended memory. To avoid these, we should observe the following rules while developing code for combinational circuit:

- Include all the branches of an if or case statement.
- Assign a value to every output signal in every branch.

Consider the following code segment, which intends to describe a circuit that generates greater-than (i.e., gt) and equal-to (i.e., eq) output signals:

```
always @*
   if (a > b)       // eq not assigned in this branch
      gt = 1'b1;
   else if (a == b) // gt not assigned in this branch
      eq = 1'b1;
                    // final else branch is omitted
```

The segment violates both rules.

Let us first examine the incomplete branch error. There is no else branch in the segment. If both the a>b and a==b expressions are false, both gt and eq are not assigned values. According to Verilog definition, they keep their previous values (i.e., the outputs depend on the internal state) and unintended latches are inferred.

The segment also has incomplete output assignment errors. For example, when the a>b expression is true, eq is not assigned a value and thus will keep its previous state. A latch will be inferred accordingly.

There are two ways to fix the errors. The first is to add the else branch and explicitly assign all output variables. The code becomes

```
always @*
   if (a > b)
      begin
         gt = 1'b1;
         eq = 1'b0;
      end
   else if (a == b)
      begin
         gt = 1'b0;
         eq = 1'b1;
      end
   else    // i.e., a < b
      begin
         gt = 1'b0;
         eq = 1'b0;
      end
```

The alternative is to assign a default value to each variable in the beginning of the always block to cover the unspecified branch and unassigned variable. The code becomes

```
always @*
begin
   gt = 1'b0;  // default value for gt
   eq = 1'b0;  // default value for eq
   if (a > b)
      gt = 1'b1;
   else if (a == b)
      eq = 1'b1;
end
```

Both gt and eq assume 0 if they are not assigned a value later.

The case statement experiences the same errors if some values of the [case_expr] expression are not covered by the item expressions (i.e., not a full-case statement). Consider the following code segment:

```
reg [1:0] s
. . .
```

```
case (s)
   2'b00: y = 1'b1;
   2'b10: y = 1'b0;
   2'b11: y = 1'b1;
endcase
```

The 2'b01 value is not covered by any branch. If s assumes this combination, y will keep its previous value and an unintended latch is inferred. To fix the error, we must ensure that y is assigned a value all the time. One way to do this is to use the **default** keyword in the end to cover all the unspecified values. We can either replace the last item expression:

```
case (s)
   2'b00: y = 1'b1;
   2'b10: y = 1'b0;
   default: y = 1'b1;   // y gets 1 for 2'b01
endcase
```

or add a new item expression with the don't-care value:

```
case (s)
   2'b00: y = 1'b1;
   2'b10: y = 1'b0;
   2'b11: y = 1'b1;
   default: y = 1'bx; // y gets x for 2'b01
endcase
```

Alternatively, we can assign a default value in the beginning of the always block:

```
y = 1'b0;    // can also use y = 1'bx for don't-care
case (s)
   2'b00: y = 1'b1;
   2'b10: y = 1'b0;
   2'b11: y = 1'b1;
endcase
```

3.7.2 Guidelines

The always block is a flexible and powerful language construct. However, it must be used with care to infer correct and efficient circuits and to avoid any discrepancy between synthesis and simulation. Following are the coding guidelines for the description of combinational circuits:

- Assign a variable only in a single always block.
- Use blocking statements for combinational circuits.
- Use @* to include all inputs automatically in the sensitivity list.
- Make sure that all branches of the if and case statements are included.
- Make sure that the outputs are assigned in all branches.
- One way to satisfy the two previous guidelines is to assign default values for outputs in the beginning of the always block.
- Describe the desired full case and parallel case in code rather than using software directives or attributes.
- Be aware of the type of routing network inferred by different control constructs.
- Think hardware, not C code.

3.8 PARAMETER AND CONSTANT

3.8.1 Constant

HDL code frequently uses constant values in expressions and array boundaries. These values are fixed within the module and cannot be modified. One good design practice is to replace the "hard literals" with symbolic constants. It makes code clear and helps future maintenance and revision. In Verilog, a constant can be declared using the **localparam** (for "local parameter") keyword. For example, we can declare the width and range of a data bus as

```
localparam DATA_WIDTH = 8,
           DATA_RANGE = 2**DATA_WIDTH - 1;
```

or define a symbolic port name:

```
localparam UART_PORT  = 4'b0001,
           LCD_PORT   = 4'b0010,
           MOUSE_PORT = 4'b0100;
```

The expression in the declaration, such as `2**DATA_WIDTH-1`, is evaluated during pre-processing and thus infers no physical circuit. In this book, we use capital letters for constants.

The use of a constant can best be explained by an example. Consider the code of an adder with the carry-out bit. One way to do it is to extend the input manually by 1 bit, perform the regular addition, and extract the MSB of the summation as the carry-out bit. The code is shown in Listing 3.9.

Listing 3.9 Adder using a hard literal

```
module adder_carry_hard_lit
   (
    input wire [3:0] a, b,
    output wire [3:0] sum,
    output wire cout  // carry-out
   );

   // signal declaration
   wire [4:0] sum_ext;

   //body
   assign sum_ext = {1'b0, a} + {1'b0, b};
   assign sum = sum_ext[3:0];
   assign cout= sum_ext[4];

endmodule
```

The code is for a 4-bit adder. Hard literals, such as 3 and 4, are used for the ranges, as in **wire** `[4:0]` and `sum_ext[3:0]`, 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 readability, we can use a symbolic constant, `N`, to represent the number of bits of the adder. The revised code is shown in Listing 3.10.

Listing 3.10 Adder using constants

```
module adder_carry_local_par
   (
    input wire [3:0] a, b,
    output wire [3:0] sum,
    output wire cout  // carry-out
   );

   // constant declaration
   localparam N = 4,
              N1 = N-1;

   // signal declaration
   wire [N:0] sum_ext;

   //body
   assign sum_ext = {1'b0, a} + {1'b0, b};
   assign sum = sum_ext[N1:0];
   assign cout= sum_ext[N];

endmodule
```

The constant makes the code easier to understand and maintain.

3.8.2 Parameter

A Verilog module can be instantiated as a component and becomes a part of a larger design, as discussed in Section 1.6. Verilog provides a construct, known as a *parameter*, to pass information into a module. This mechanism makes the module versatile and reusable. A parameter cannot be modified inside the module and thus functions like a constant.

In Verilog-2001, a parameter declaration section can be added in the header, before the port declaration. Its simplified syntax is

```
module [module_name]
   #(
    parameter [parameter_name]=[default_value],
              . . .
              [parameter_name]=[default_value];
   )
   (
    . . .  // I/O port declaration
   );
```

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

Listing 3.11 Adder using a parameter

```
module adder_carry_para
   #(parameter N=4)
   (
     input wire [N-1:0] a, b,
     output wire [N-1:0] sum,
     output wire cout  // carry-out
```

```
    );

    // constant declaration
    localparam N1 = N-1;

    // signal declaration
    wire [N:0] sum_ext;

    //body
    assign sum_ext = {1'b0, a} + {1'b0, b};
    assign sum = sum_ext[N1:0];
    assign cout= sum_ext[N];

endmodule
```

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

If the adder is later used as a component in other code, we can assign a desired value to the parameter during component instantiation and override the default value. Similar to the port connection discussed in Section 1.6, parameter assignment can be done either *by name* or *by ordered list*. A potential problem of the by-ordered-list scheme is discussed in Section 1.6 and we always use the by-name scheme in this book. The default value will be used if the parameter assignment is omitted. The use of the parameter in component instantiation is demonstrated in Listing 3.12.

Listing 3.12 Adder instantiation example

```
module adder_insta
   (
     input wire [3:0] a4, b4,
     output wire [3:0] sum4,
     output wire c4,
     input wire [7:0] a8, b8,
     output wire [7:0] sum8,
     output wire c8
   );

   // instantiate 8-bit adder
   adder_carry_para #(.N(8)) unit1
      (.a(a8), .b(b8), .sum(sum8), .cout(c8));

   // instantiate 4-bit adder
   adder_carry_para unit2
      (.a(a4), .b(b4), .sum(sum4), .cout(c4));

endmodule
```

A parameter 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.

3.8.3 Use of parameters in Verilog-1995

The **localparam** keyword, header declaration, and assignment by name discussed earlier are all new Verilog-2001 features. In Verilog-1995, parameters are declared after the header and can only be redefined by using the by-order-list scheme or the **defparam** statement. Furthermore, constants must be declared as parameters, even though they should not be redefined. The previous adder code in Verilog-1995 syntax is shown in Listing 3.13. **FYI**

Listing 3.13 Parameter use in Verilog-1995

```
module adder_carry_95 (a, b, sum, cout);
   parameter N = 4;      // parameter declared before the port
   parameter N1 = N-1;   // no localparam in Verilog-1995
   input wire [N1:0] a, b;
   output wire [N1:0] sum;
   output wire cout;

   // signal declaration
   wire [N:0] sum_ext;

   //body
   assign sum_ext = {1'b0, a} + {1'b0, b};
   assign sum = sum_ext[N1:0];
   assign cout= sum_ext[N];

endmodule
```

When a component is instantiated, the parameter can only be redefined by using the by-ordered-list scheme, as in

```
adder_carry_95 #(8,7) unit1
   (.a(a8), .b(b8), .sum(sum8), .cout(c8));
```

or by using the **defparam** statement, as in

```
defparam unit1.N=8;
defparam unit1.N1=7;
adder_carry_95 unit1
   (.a(a8), .b(b8), .sum(sum8), .cout(c8));
```

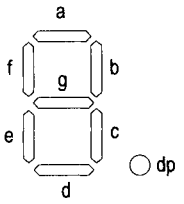
The Verilog-1995 scheme is more tedious and may introduce subtle errors and we don't use it in this book.

3.9 DESIGN EXAMPLES

3.9.1 Hexadecimal digit to seven-segment LED decoder

The sketch of a seven-segment LED display is shown in Figure 3.6(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 3.6(b). For completeness, we assume that there is also a 1-bit input, dp, which is connected directly to the decimal



(a) Diagram of a seven-segment LED display

(b) Hexadecimal digit patterns

Figure 3.6 Seven-segment LED display and hexadecimal patterns.

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 3.14. It uses one case statement to list all the desired patterns for the seven LSBs of the sseg signal. The MSB is connected to dp.

Listing 3.14 Hexadecimal digit to seven-segment LED decoder

```
module hex_to_sseg
   (
    input wire [3:0] hex,
    input wire dp,
    output reg [7:0] sseg  // output active low
   );

   always @*
   begin
      case(hex)
         4'h0: sseg[6:0] = 7'b0000001;
         4'h1: sseg[6:0] = 7'b1001111;
         4'h2: sseg[6:0] = 7'b0010010;
         4'h3: sseg[6:0] = 7'b0000110;
         4'h4: sseg[6:0] = 7'b1001100;
         4'h5: sseg[6:0] = 7'b0100100;
         4'h6: sseg[6:0] = 7'b0100000;
         4'h7: sseg[6:0] = 7'b0001111;
         4'h8: sseg[6:0] = 7'b0000000;
         4'h9: sseg[6:0] = 7'b0000100;
         4'ha: sseg[6:0] = 7'b0001000;
         4'hb: sseg[6:0] = 7'b1100000;
         4'hc: sseg[6:0] = 7'b0110001;
         4'hd: sseg[6:0] = 7'b1000010;
         4'he: sseg[6:0] = 7'b0110000;
         default: sseg[6:0] = 7'b0111000;  //4'hf
      endcase
      sseg[7] = dp;
```

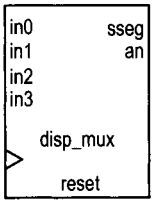
```
   end
30
endmodule
```

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 3.7(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.

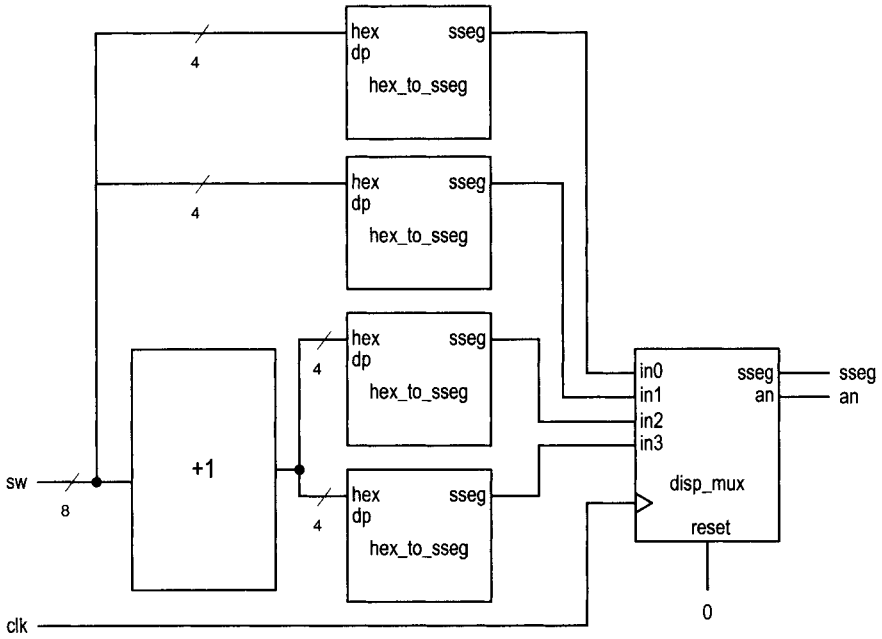
Testing circuit We use a simple 8-bit increment circuit to verify operation of the decoder. The sketch is shown in Figure 3.7(b). The `sw` input is the 8-bit switch of the prototyping board. It is fed to an incrementor to obtain `sw+1`. The original and incremented `sw` signals are then passed to four decoders to display the four hexadecimal digits on seven-segment LED displays. The code is shown in Listing 3.15.

Listing 3.15 Hex-to-LED decoder testing circuit

```
module hex_to_sseg_test
   (
     input wire clk,
     input wire [7:0] sw,
5    output wire [3:0] an,
     output wire [7:0] sseg
   );

   // signal declaration
10 wire [7:0] inc;
   wire [7:0] led0, led1, led2, led3;

   // increment input
   assign inc = sw + 1;
15
   // instantiate four instances of hex decoders
   // instance for 4 LSBs of input
   hex_to_sseg sseg_unit_0
      (.hex(sw[3:0]), .dp(1'b0), .sseg(led0));
20 // instance for 4 MSBs of input
   hex_to_sseg sseg_unit_1
      (.hex(sw[7:4]), .dp(1'b0), .sseg(led1));
   // instance for 4 LSBs of incremented value
   hex_to_sseg sseg_unit_2
25    (.hex(inc[3:0]), .dp(1'b1), .sseg(led2));
   // instance for 4 MSBs of incremented value
   hex_to_sseg sseg_unit_3
      (.hex(inc[7:4]), .dp(1'b1), .sseg(led3));

30 // instantiate 7-seg LED display time-multiplexing module
   disp_mux disp_unit
```



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



(b) Block diagram of a decoder testing circuit

Figure 3.7 LED time-multiplexing module and decoder testing circuit.

```
    (.clk(clk), .reset(1'b0), .in0(led0), .in1(led1),
     .in2(led2), .in3(led3), .an(an), .sseg(sseg));

endmodule
```

We can follow the procedure in Chapter 2 to synthesize and implement the circuit on the prototyping board. Note that the disp_mux.v 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.

3.9.2 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 3.16, which realizes the two-stage implementation scheme. For clarity, we split the input number internally and use separate sign and magnitude signals. A parameter, `N`, is used to represent the width of the adder.

Listing 3.16 Sign-magnitude adder

```
module sign_mag_add
   #(
    parameter N=4
   )
   (
    input wire [N-1:0] a, b,
    output reg [N-1:0] sum
   );

   // signal declaration
   reg [N-2:0]  mag_a, mag_b, mag_sum, max, min;
   reg sign_a, sign_b, sign_sum;

   //body
   always @*
   begin
      // separate magnitude and sign
      mag_a = a[N-2:0];
      mag_b = b[N-2:0];
      sign_a = a[N-1];
      sign_b = b[N-1];
```

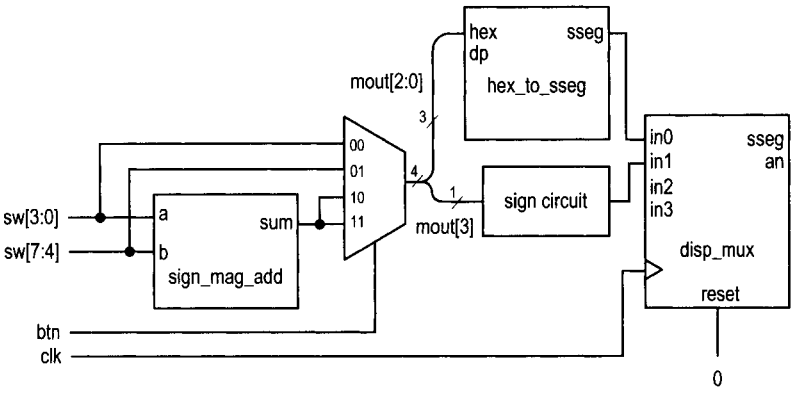


Figure 3.8 Sign-magnitude adder testing circuit.

```
        // sort according to magnitude
        if (mag_a > mag_b)
           begin
25            max = mag_a;
              min = mag_b;
              sign_sum = sign_a;
           end
        else
30         begin
              max = mag_b;
              min = mag_a;
              sign_sum = sign_b;
           end
35      // add/sub magnitude
        if (sign_a==sign_b)
           mag_sum = max + min;
        else
           mag_sum = max - min;
40      // form output
        sum = {sign_sum, mag_sum};
     end
  endmodule
```

Testing circuit We use a 4-bit sign-magnitude adder to verify circuit operation. The sketch of the testing circuit is shown in Figure 3.8. 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 3.9.1. The code is shown in Listing 3.17.

Listing 3.17 Sign-magnitude adder testing circuit

```
module sm_add_test
   (
    input wire clk,
    input wire [1:0] btn,
    input wire [7:0] sw,
    output wire [3:0] an,
    output wire [7:0] sseg
   );

   // signal declaration
   wire [3:0] sum, mout, oct;
   wire [7:0] led3, led2, led1, led0;

   // instantiate adder
   sign_mag_add #(.N(4)) sm_adder_unit
     (.a(sw[3:0]), .b(sw[7:4]), .sum(sum));

   // magnitude displayed on rightmost 7-seg LED
   assign mout = (btn==2'b00) ? sw[3:0] :
                 (btn==2'b01) ? sw[7:4] :
                 sum;
   assign oct = {1'b0, mout[2:0]};
   // instantiate hex decoder
   hex_to_sseg sseg_unit
      (.hex(oct), .dp(1'b0), .sseg(led0));

   // sign displayed on 2nd 7-seg LED
   // middle LED bar on for negative number
   assign led1 = mout[3] ? 8'b11111110 : 8'b11111111;
   // blank two other LEDs
   assign led2 = 8'b11111111;
   assign led3 = 8'b11111111;

   // instantiate 7-seg LED display time-multiplexing module
   disp_mux disp_unit
      (.clk(clk), .reset(1'b0), .in0(led0), .in1(led1),
       .in2(led2), .in3(led3), .an(an), .sseg(sseg));

endmodule
```

3.9.3 Barrel shifter

Although Verilog has built-in shift functions, there is no rotation operation. In this subsection, we examine an 8-bit barrel shifter that rotates an arbitrary number of bits to the 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 case statement to exhaustively list all combinations of the amt signal and the corresponding rotated results. The code is shown in Listing 3.18.

Listing 3.18 Barrel shifter using a case statement

```
module barrel_shifter_case
   (
     input wire [7:0] a,
     input wire [2:0] amt,
     output reg [7:0] y
    );

   // body
   always @*
      case(amt)
         3'o0: y = a;
         3'o1: y = {a[0], a[7:1]};
         3'o2: y = {a[1:0], a[7:2]};
         3'o3: y = {a[2:0], a[7:3]};
         3'o4: y = {a[3:0], a[7:4]};
         3'o5: y = {a[4:0], a[7:5]};
         3'o6: y = {a[5:0], a[7:6]};
         default: y = {a[6:0], a[7]};
      endcase

endmodule
```

While the code is straightforward, it will become cumbersome when the number of data bits increases. Furthermore, a large number of items in a case statement 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 3.19.

Listing 3.19 Barrel shifter using multistage shifts

```
module barrel_shifter_stage
   (
    input wire [7:0] a,
    input wire [2:0] amt,
    output wire [7:0] y
   );

   // signal declaration
   wire [7:0] s0, s1;

   // body
   // stage 0, shift 0 or 1 bit
   assign s0 = amt[0] ? {a[0], a[7:1]} : a;
   // stage 1, shift 0 or 2 bits
   assign s1 = amt[1] ? {s0[1:0], s0[7:2]} : s0;
   // stage 2, shift 0 or 4 bits
   assign y = amt[2] ? {s1[3:0], s1[7:4]} : s1;

endmodule
```

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 3.20.

Listing 3.20 Barrel shifter testing circuit

```
module shifter_test
   (
    input wire [2:0] btn,
    input wire [7:0] sw,
    output wire [7:0] led
   );

   // instantiate shifter
   barrel_shifter_stage shift_unit
     (.a(sw), .amt(btn), .y(led));

endmodule
```

3.9.4 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.

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 3.9. The computation is done in four major steps:

		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 3.9 Floating-point addition examples.

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 that 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 the 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 3.21.

Listing 3.21 Simplified floating-point adder

```
module fp_adder
   (
    input wire sign1, sign2,
    input wire [3:0] exp1, exp2,
    input wire [7:0] frac1, frac2,
```

```
    output reg sign_out,
    output reg [3:0] exp_out,
    output reg [7:0] frac_out
);

// signal declaration
//suffix b, s, a, n for
//       big, small, aligned, normalized number
reg signb, signs;
reg [3:0] expb, exps, expn, exp_diff;
reg [7:0] fracb, fracs, fraca, fracn, sum_norm;
reg [8:0] sum;
reg [2:0] lead0;

// body
always @*
begin
   // 1st stage: sort to find the larger number
   if ({exp1, frac1} > {exp2, frac2})
      begin
         signb = sign1;
         signs = sign2;
         expb = exp1;
         exps = exp2;
         fracb = frac1;
         fracs = frac2;
      end
   else
      begin
         signb = sign2;
         signs = sign1;
         expb = exp2;
         exps = exp1;
         fracb = frac2;
         fracs = frac1;
      end

   // 2nd stage: align smaller number
   exp_diff = expb - exps;
   fraca = fracs >> exp_diff;

   // 3rd stage: add/substract
   if (signb==signs)
      sum = {1'b0, fracb} + {1'b0, fraca};
   else
      sum = {1'b0, fracb} - {1'b0, fraca};

   // 4th stage: normalize
   // count leading 0s
   if (sum[7])
      lead0 = 3'o0;
   else if (sum[6])
      lead0 = 3'o1;
```

```
        else if (sum[5])
           lead0 = 3'o2;
        else if (sum[4])
           lead0 = 3'o3;
        else if (sum[3])
           lead0 = 3'o4;
        else if (sum[2])
           lead0 = 3'o5;
        else if (sum[1])
           lead0 = 3'o6;
        else
           lead0 = 3'o7;
        // shift significand according to leading 0
        sum_norm = sum << lead0;
        // normalize with special conditions
        if (sum[8]) // with carry out; shift frac to right
           begin
              expn = expb + 1;
              fracn = sum[8:1];
           end
        else if (lead0 > expb) // too small to normalize
           begin
              expn = 0;          // set to 0
              fracn = 0;
           end
        else
           begin
              expn = expb - lead0;
             fracn = sum_norm;
           end

        // form output
        sign_out = signb;
        exp_out = expn;
        frac_out = fracn;
     end

endmodule
```

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 3.9.2. 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 3.22. 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 3.22 Floating-point adder testing circuit

```
module fp_adder_test
   (
    input wire clk,
    input wire [1:0] btn,
    input wire [7:0] sw,
    output wire [3:0] an,
    output wire [7:0] sseg
   );

   // signal declarations
   wire sign1, sign2, sign_out;
   wire [3:0] exp1, exp2, exp_out;
   wire [7:0] frac1, frac2, frac_out;
   wire [7:0] led3, led2, led1, led0;

   // body
   // set up the fp adder input signals
   assign sign1 = 1'b0;
   assign exp1 = 4'b1000;
   assign frac1 = {1'b1, sw[1:0], 5'b10101};
   assign sign2 = sw[7];
   assign exp2 = btn;
   assign frac2 = {1'b1, sw[6:0]};

   // instantiate fp adder
   fp_adder fp_unit
      (.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
   hex_to_sseg sseg_unit_0
      (.hex(exp_out), .dp(1'b0), .sseg(led0));
   // 4 LSBs of fraction
   hex_to_sseg sseg_unit_1
      (.hex(frac_out[3:0]), .dp(1'b1), .sseg(led1));
   // 4 MSBs of fraction
   hex_to_sseg sseg_unit_2
      (.hex(frac_out[7:4]), .dp(1'b0), .sseg(led2));
```

```
    //sign
     assign led3 = (sign_out) ? 8'b11111110 : 8'b11111111;

     // instantiate 7-seg LED display time-multiplexing module
45   disp_mux disp_unit
        (.clk(clk), .reset(1'b0), .in0(led0), .in1(led1),
         .in2(led2), .in3(led3), .an(an), .sseg(sseg));

endmodule
```

3.10 BIBLIOGRAPHIC NOTES

Verilog HDL, 2nd edition, by S. Palnitkar and *Starter's Guide to Verilog 2001* by M. D. Ciletti provide detailed coverage of Verilog's syntax and constructs. The article "The IEEE Verilog 1364-2001 Standard: What's New, and Why You Need It" by S. Sutherland summarizes the new features. The article ""full_case parallel_case", the Evil Twins of Verilog Synthesis" by C. E. Cummings examines the caveats of the full-case and parallel-case directives, and his other article, "New Verilog-2001 Techniques for Creating Parameterized Models," discusses the advantage of Verilog-2001's new parameter passing scheme.

3.11 SUGGESTED EXPERIMENTS

3.11.1 Multifunction 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 (e.g., 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.

3.11.2 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.

3.11.3 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.

3.11.4 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 3.9.4.

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.

3.11.5 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 3.9.4 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.

3.11.6 Enhanced floating-point adder

The floating-point adder in Section 3.9.4 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 3.9.4 to accommodate the round-to-the-nearest-even method.