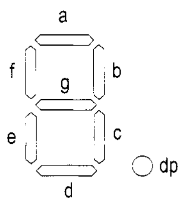
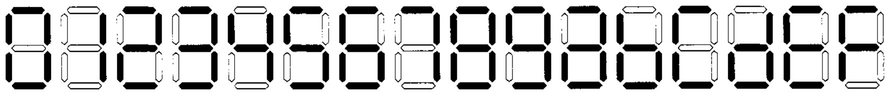


(a) Diagram of a seven-segment LED display



(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;
```

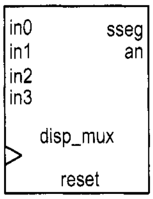
There are four seven-segment LED displays on the prototyping board. To save the number of FPGA chip's I/O pins, a time-multiplexing scheme is used. The block diagram of the time-multiplexing module, disp_mux, is shown in Figure 3.6(a). The inputs are in0, in1, in2, and in3, which correspond to four 8-bit 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.

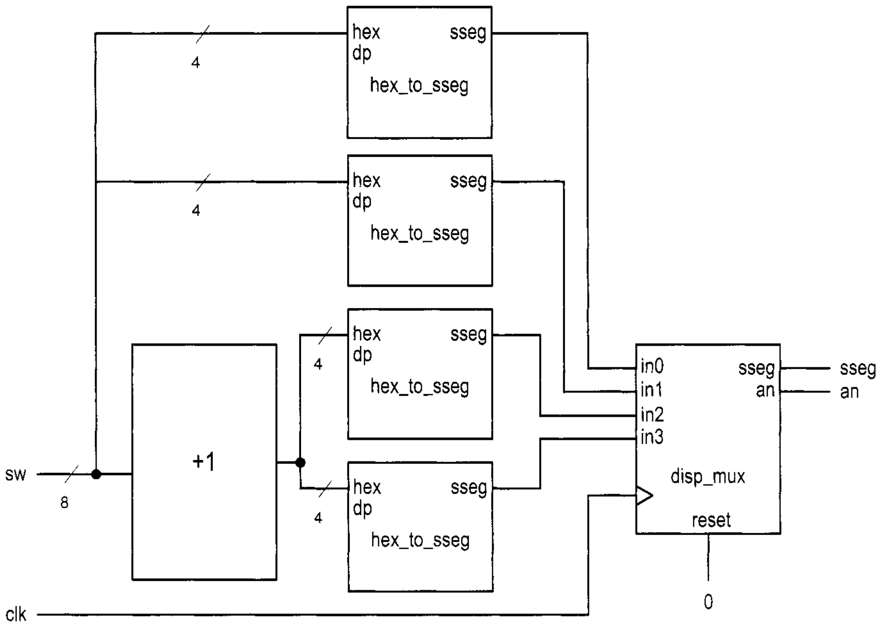
Listing 3.13 Hex-to-LED decoder testing circuit

```
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.

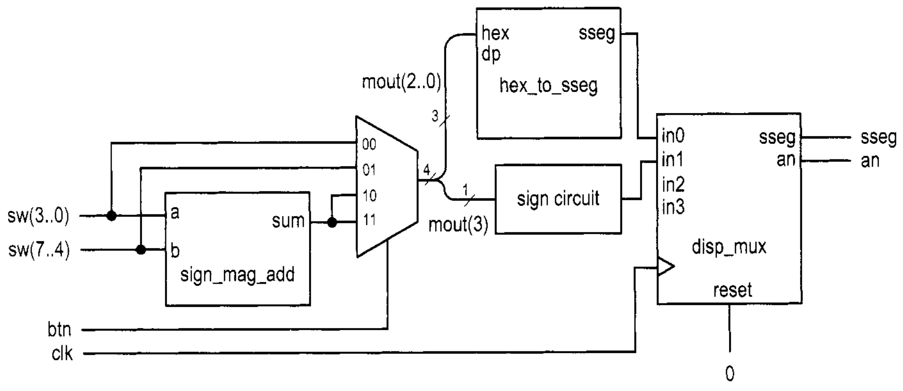


Figure 3.7 Sign-magnitude adder testing circuit.

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 $m_22^2 + m_12^1 + m_02^0$, 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(
         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