

CHAPTER 5

Note: the HDL files given in the following solutions are available on the textbook's companion website at:

<http://textbooks.elsevier.com/9780123704979> .

Exercise 5.1

(a) From Equation 5.1, we find the 64-bit ripple-carry adder delay to be:

$$t_{\text{ripple}} = Nt_{\text{FA}} = 64(450 \text{ ps}) = 28.8 \text{ ns}$$

(b) From Equation 5.6, we find the 64-bit carry-lookahead adder delay to be:

$$t_{CLA} = t_{\text{pg}} + t_{\text{pg_block}} + \left(\frac{N}{k} - 1\right)t_{\text{AND_OR}} + kt_{\text{FA}} \quad 150$$

$$t_{CLA} = \left[150 + (6 \times 150) + \left(\frac{64}{4} - 1\right)300 + (4 \times 450)\right] = 7.35 \text{ ns}$$

(Note: the actual delay is only 7.2 ns because the first AND_OR gate only has a 150 ps delay.)

(c) From Equation 5.11, we find the 64-bit prefix adder delay to be:

$$t_{PA} = t_{\text{pg}} + \log_2 N(t_{\text{pg_prefix}}) + t_{\text{XOR}}$$

$$t_{PA} = [150 + 6(300) + 150] = 2.1 \text{ ns}$$

Exercise 5.2

(a) The fundamental building block of both the ripple-carry and carry-lookahead adders is the full adder. We use the full adder from Figure 4.8, shown again here for convenience:

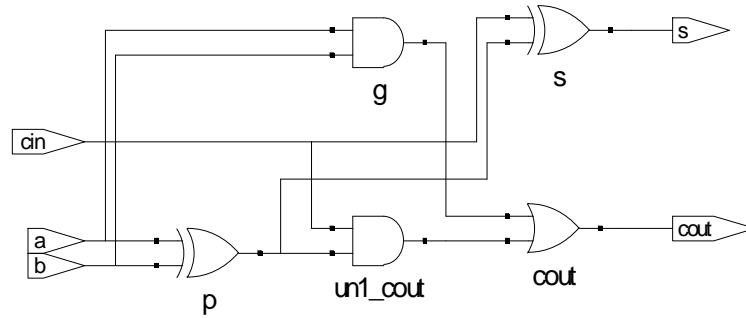


FIGURE 5.1 Full adder implementation

The full adder delay is three two-input gates.

$$t_{FA} = 3(50) \text{ ps} = 150 \text{ ps}$$

The full adder area is five two-input gates.

$$A_{FA} = 5(15 \mu m^2) = 75 \mu m^2$$

The full adder capacitance is five two-input gates.

$$C_{FA} = 5(20 \text{ fF}) = 100 \text{ fF}$$

Thus, the ripple-carry adder delay, area, and capacitance are:

$$t_{\text{ripple}} = N t_{FA} = 64(150 \text{ ps}) = 9.6 \text{ ns}$$

$$A_{\text{ripple}} = N A_{FA} = 64(75 \mu m^2) = 4800 \mu m^2$$

$$C_{\text{ripple}} = N C_{FA} = 64(100 \text{ fF}) = 6.4 \text{ pF}$$

Using the carry-lookahead adder from Figure 5.6, we can calculate delay, area, and capacitance. Using Equation 5.6:

$$t_{CLA} = [50 + 6(50) + 15(100) + 4(150)] \text{ ps} = 2.45 \text{ ns}$$

(The actual delay is only 2.4 ns because the first AND_OR gate only contributes one gate delay.)

For each 4-bit block of the 64-bit carry-lookahead adder, there are 4 full adders, 8 two-input gates to generate P_i and G_i , and 11 two-input gates to generate $P_{i:j}$ and $G_{i:j}$. Thus, the area and capacitance are:

$$A_{CLAblock} = [4(75) + 19(15)] \mu m^2 = 585 \mu m^2$$

$$A_{CLA} = 16(585) \mu m^2 = 9360 \mu m^2$$

$$C_{CLAblock} = [4(100) + 19(20)] \text{ fF} = 780 \text{ fF}$$

$$C_{CLA} = 16(780) \text{ fF} = 12.48 \text{ pF}$$

Now solving for power using Equation 1.4,

$$P_{\text{dynamic_ripple}} = \frac{1}{2} C V_{DD}^2 f = \frac{1}{2} (6.4 \text{ pF}) (1.2 \text{ V})^2 (100 \text{ MHz}) = 0.461 \text{ mW}$$

$$P_{\text{dynamic_CLA}} = \frac{1}{2} C V_{DD}^2 f = \frac{1}{2} (12.48 \text{ pF}) (1.2 \text{ V})^2 (100 \text{ MHz}) = 0.899 \text{ mW}$$

	ripple- carry	carry-lookahead	cla/ripple
Area (μm^2)	4800	9360	1.95
Delay (ns)	9.6	2.45	0.26
Power (mW)	0.461	0.899	1.95

TABLE 5.1 CLA and ripple-carry adder comparison

(b) Compared to the ripple-carry adder, the carry-lookahead adder is almost twice as large and uses almost twice the power, but is almost four times as fast. Thus for performance-limited designs where area and power are not constraints, the carry-lookahead adder is the clear choice. On the other hand, if either area or power are the limiting constraints, one would choose a ripple-carry adder if performance were not a constraint.

Exercise 5.3

A designer might choose to use a ripple-carry adder instead of a carry-lookahead adder if chip area is the critical resource and delay is not the critical constraint.

Exercise 5.4

SystemVerilog

```

module prefixadd16(input  logic [15:0] a, b,
                  input  logic      cin,
                  output logic [15:0] s,
                  output logic      cout);

    logic [14:0] p, g;
    logic [7:0]  pij_0, gij_0, pij_1, gij_1,
                pij_2, gij_2, pij_3, gij_3;
    logic [15:0] gen;

    pgblock pgblock_top(a[14:0], b[14:0], p, g);
    pgblackblock pgblackblock_0({p[14], p[12], p[10],
                                p[8], p[6], p[4], p[2], p[0]},
    {g[14], g[12], g[10], g[8], g[6], g[4], g[2], g[0]},
    {p[13], p[11], p[9], p[7], p[5], p[3], p[1], 1'b0},
    {g[13], g[11], g[9], g[7], g[5], g[3], g[1], cin},
    pij_0, gij_0);

    pgblackblock pgblackblock_1({pij_0[7], p[13],
                                pij_0[5], p[9], pij_0[3], p[5], pij_0[1], p[1]},
    {gij_0[7], g[13], gij_0[5], g[9], gij_0[3],
     g[5], gij_0[1], g[1]},
    { {2{pij_0[6]}}, {2{pij_0[4]}}, {2{pij_0[2]}},
      {2{pij_0[0]}} },
    { {2{gij_0[6]}}, {2{gij_0[4]}}, {2{gij_0[2]}},
      {2{gij_0[0]}} },
    pij_1, gij_1);

    pgblackblock pgblackblock_2({pij_1[7], pij_1[6],
                                pij_0[6], p[11], pij_1[3], pij_1[2], pij_0[2], p[3]},
    {gij_1[7], gij_1[6], gij_0[6], g[11], gij_1[3],
     gij_1[2], gij_0[2], g[3]},
    { {4{pij_1[5]}}, {4{pij_1[1]}} },
    { {4{gij_1[5]}}, {4{gij_1[1]}} },
    pij_2, gij_2);

    pgblackblock pgblackblock_3({pij_2[7], pij_2[6],
                                pij_2[5], pij_2[4], pij_1[5], pij_1[4],
                                pij_0[4], p[7]},
    {gij_2[7], gij_2[6], gij_2[5],
     gij_2[4], gij_1[5], gij_1[4], gij_0[4], g[7]},
    { 8{pij_2[3]} }, { 8{gij_2[3]} }, pij_3, gij_3);

    sumblock sum_out(a, b, gen, s);

    assign gen = {gij_3, gij_2[3:0],
                  gij_1[1:0], gij_0[0], cin};
    assign cout = (a[15] & b[15]) |
                  (gen[15] & (a[15] | b[15]));

endmodule

```

VHDL

```

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity prefixadd16 is
    port(a, b: in  STD_LOGIC_VECTOR(15 downto 0);
          cin: in  STD_LOGIC;
          s:  out STD_LOGIC_VECTOR(15 downto 0);
          cout: out STD_LOGIC);
end;

architecture synth of prefixadd16 is
    component pgblock
        port(a, b: in  STD_LOGIC_VECTOR(14 downto 0);
              p, g: out STD_LOGIC_VECTOR(14 downto 0));
    end component;

    component pgblackblock is
        port (pik, gik: in  STD_LOGIC_VECTOR(7 downto 0);
              pkj, gkj: in  STD_LOGIC_VECTOR(7 downto 0);
              pij: out STD_LOGIC_VECTOR(7 downto 0);
              gij: out STD_LOGIC_VECTOR(7 downto 0));
    end component;

    component sumblock is
        port (a, b, g: in  STD_LOGIC_VECTOR(15 downto 0);
              s:  out STD_LOGIC_VECTOR(15 downto 0));
    end component;

    signal p, g: STD_LOGIC_VECTOR(14 downto 0);
    signal pij_0, gij_0, pij_1, gij_1,
            pij_2, gij_2, gij_3:
        STD_LOGIC_VECTOR(7 downto 0);
    signal gen: STD_LOGIC_VECTOR(15 downto 0);
    signal pik_0, pik_1, pik_2, pik_3,
            gik_0, gik_1, gik_2, gik_3,
            pkj_0, pkj_1, pkj_2, pkj_3,
            gkj_0, gkj_1, gkj_2, gkj_3, dummy:
        STD_LOGIC_VECTOR(7 downto 0);

begin
    pgblock_top: pgblock
        port map(a(14 downto 0), b(14 downto 0), p, g);

    pik_0 <=
        (p(14) & p(12) & p(10) & p(8) & p(6) & p(4) & p(2) & p(0));
    gik_0 <=
        (g(14) & g(12) & g(10) & g(8) & g(6) & g(4) & g(2) & g(0));
    pkj_0 <=
        (p(13) & p(11) & p(9) & p(7) & p(5) & p(3) & p(1) & '0');
    gkj_0 <=
        (g(13) & g(11) & g(9) & g(7) & g(5) & g(3) & g(1) & cin);

    pgblackblock_0: pgblackblock
        port map(pik_0, gik_0, pkj_0, gkj_0,
                pij_0, gij_0);

```

*(continued from previous page)***Verilog****VHDL**

```

pik_1 <= (pij_0(7) & p(13) & pij_0(5) & p(9) &
          pij_0(3) & p(5) & pij_0(1) & p(1));
gik_1 <= (gij_0(7) & g(13) & gij_0(5) & g(9) &
          gij_0(3) & g(5) & gij_0(1) & g(1));
pkj_1 <= (pij_0(6) & pij_0(6) & pij_0(4) & pij_0(4) &
          pij_0(2) & pij_0(2) & pij_0(0) & pij_0(0));
gkj_1 <= (gij_0(6) & gij_0(6) & gij_0(4) & gij_0(4) &
          gij_0(2) & gij_0(2) & gij_0(0) & gij_0(0));

pgblackblock_1: pgblackblock
  port map(pik_1, gik_1, pkj_1, gkj_1,
           pij_1, gij_1);

pik_2 <= (pij_1(7) & pij_1(6) & pij_0(6) &
          p(11) & pij_1(3) & pij_1(2) &
          pij_0(2) & p(3));
gik_2 <= (gij_1(7) & gij_1(6) & gij_0(6) &
          g(11) & gij_1(3) & gij_1(2) &
          gij_0(2) & g(3));
pkj_2 <= (pij_1(5) & pij_1(5) & pij_1(5) & pij_1(5) &
          pij_1(1) & pij_1(1) & pij_1(1) & pij_1(1));
gkj_2 <= (gij_1(5) & gij_1(5) & gij_1(5) & gij_1(5) &
          gij_1(1) & gij_1(1) & gij_1(1) & gij_1(1));

pgblackblock_2: pgblackblock
  port map(pik_2, gik_2, pkj_2, gkj_2, pij_2, gij_2);

pik_3 <= (pij_2(7) & pij_2(6) & pij_2(5) &
          pij_2(4) & pij_1(5) & pij_1(4) &
          pij_0(4) & p(7));
gik_3 <= (gij_2(7) & gij_2(6) & gij_2(5) &
          gij_2(4) & gij_1(5) & gij_1(4) &
          gij_0(4) & g(7));
pkj_3 <= (pij_2(3), pij_2(3), pij_2(3), pij_2(3),
          pij_2(3), pij_2(3), pij_2(3), pij_2(3));
gkj_3 <= (gij_2(3), gij_2(3), gij_2(3), gij_2(3),
          gij_2(3), gij_2(3), gij_2(3), gij_2(3));

pgblackblock_3: pgblackblock
  port map(pik_3, gik_3, pkj_3, gkj_3, dummy,
           gij_3);

sum_out: sumblock
  port map(a, b, gen, s);

gen <= (gij_3 & gij_2(3 downto 0) & gij_1(1 downto 0) &
        gij_0(0) & cin);
cout <= (a(15) and b(15)) or
        (gen(15) and (a(15) or b(15)));
end;

```

*(continued from previous page)***SystemVerilog**

```

module pgblock(input  logic [14:0] a, b,
               output logic [14:0] p, g);

    assign p = a | b;
    assign g = a & b;

endmodule

module pgblackblock(input  logic [7:0] pik, gik,
                   pkj, gkj,
                   output logic [7:0] pij, gij);

    assign pij = pik & pkj;
    assign gij = gik | (pik & gkj);

endmodule

module sumblock(input  logic [15:0] a, b, g,
                output logic [15:0] s);

    assign s = a ^ b ^ g;

endmodule

```

VHDL

```

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity pgblock is
    port(a, b: in  STD_LOGIC_VECTOR(14 downto 0);
          p, g: out STD_LOGIC_VECTOR(14 downto 0));
end;

architecture synth of pgblock is
begin
    p <= a or b;
    g <= a and b;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity pgblackblock is
    port(pik, gik, pkj, gkj:
          in  STD_LOGIC_VECTOR(7 downto 0);
          pij, gij:
          out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of pgblackblock is
begin
    pij <= pik and pkj;
    gij <= gik or (pik and gkj);
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sumblock is
    port(a, b, g: in  STD_LOGIC_VECTOR(15 downto 0);
          s:      out STD_LOGIC_VECTOR(15 downto 0));
end;

architecture synth of sumblock is
begin
    s <= a xor b xor g;
end;

```

Exercise 5.5

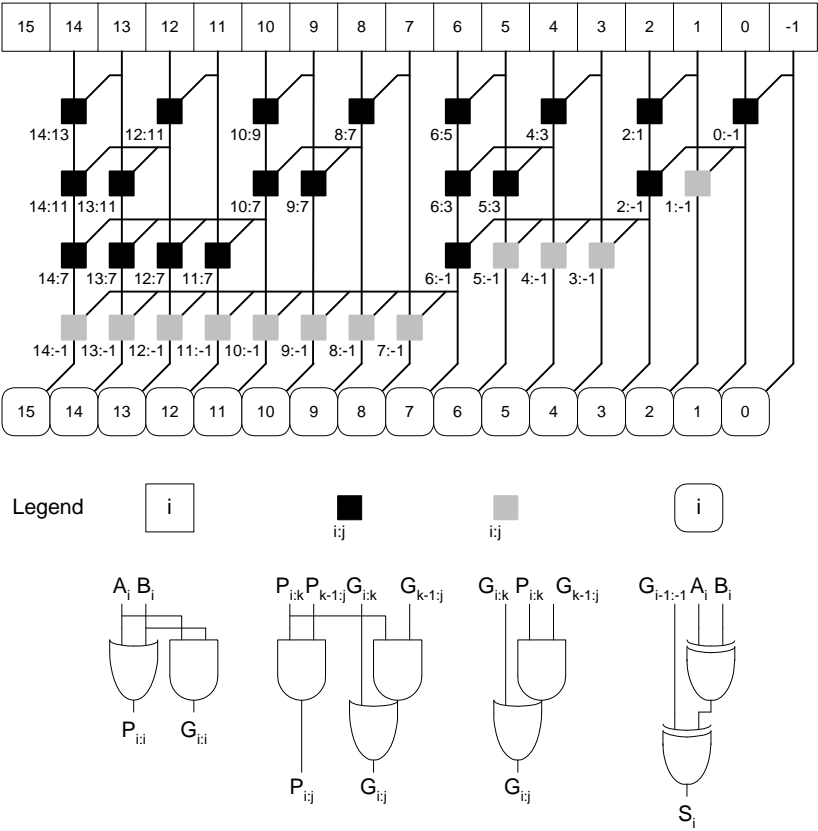


FIGURE 5.2 16-bit prefix adder with “gray cells”

Exercise 5.6

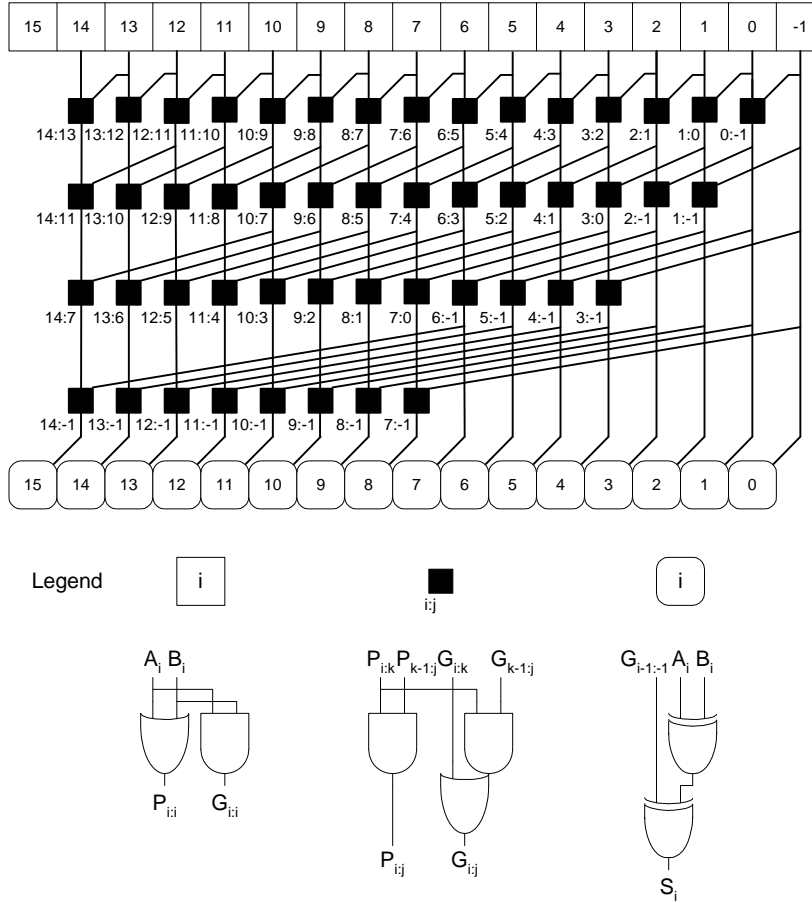


FIGURE 5.3 Schematic of a 16-bit Kogge-Stone adder

Exercise 5.7

(a) We show an 8-bit priority circuit in Figure 5.4. In the figure $X_7 = \bar{A}_7$, $X_{7:6} = \bar{A}_7 \bar{A}_6$, $X_{7:5} = \bar{A}_7 \bar{A}_6 \bar{A}_5$, and so on. The priority encoder's delay is $\log_2 N$ 2-input AND gates followed by a final row of 2-input AND gates. The final stage is an $(N/2)$ -input OR gate. Thus, in general, the delay of an N -input priority encoder is:

$$t_{pd_priority} = (\log_2 N + 1)t_{pd_AND2} + t_{pd_ORN/2}$$

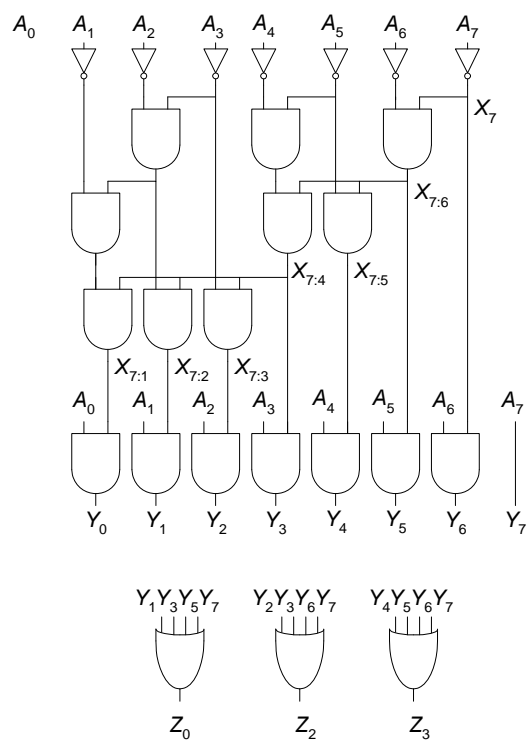


FIGURE 5.4 8-input priority encoder

SystemVerilog

```

module priorityckt(input  logic [7:0] a,
                  output logic [2:0] z);
    logic [7:0] y;
    logic      x7, x76, x75, x74, x73, x72, x71;
    logic      x32, x54, x31;
    logic [7:0] abar;

    // row of inverters
    assign abar = ~a;

    // first row of AND gates
    assign x7  = abar[7];
    assign x76 = abar[6] & x7;
    assign x54 = abar[4] & abar[5];
    assign x32 = abar[2] & abar[3];

    // second row of AND gates
    assign x75 = abar[5] & x76;
    assign x74 = x54 & x76;
    assign x31 = abar[1] & x32;

    // third row of AND gates
    assign x73 = abar[3] & x74;
    assign x72 = x32 & x74;
    assign x71 = x31 & x74;

    // fourth row of AND gates
    assign y = {a[7],      a[6] & x7,  a[5] & x76,
               a[4] & x75, a[3] & x74, a[2] & x73,
               a[1] & x72, a[0] & x71};

    // row of OR gates
    assign z = { |{y[7:4]},
               |{y[7:6], y[3:2]}},
               |{y[1], y[3], y[5], y[7]} };
endmodule

```

VHDL

```

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity priorityckt is
    port(a: in  STD_LOGIC_VECTOR(7 downto 0);
         z: out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture synth of priorityckt is
    signal y, abar:  STD_LOGIC_VECTOR(7 downto 0);
    signal x7, x76, x75, x74, x73, x72, x71,
           x32, x54, x31: STD_LOGIC;
begin
    -- row of inverters
    abar <= not a;

    -- first row of AND gates
    x7 <= abar(7);
    x76 <= abar(6) and x7;
    x54 <= abar(4) and abar(5);
    x32 <= abar(2) and abar(3);

    -- second row of AND gates
    x75 <= abar(5) and x76;
    x74 <= x54 and x76;
    x31 <= abar(1) and x32;

    -- third row of AND gates
    x73 <= abar(3) and x74;
    x72 <= x32 and x74;
    x71 <= x31 and x74;

    -- fourth row of AND gates
    y <= (a(7) & (a(6) and x7) & (a(5) and x76) &
          (a(4) and x75) & (a(3) and x74) & (a(2) and
x73) &
          (a(1) and x72) & (a(0) and x71));

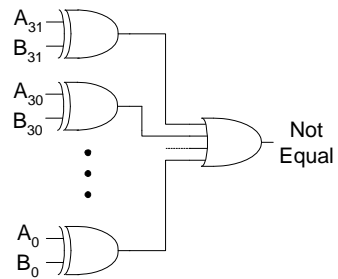
    -- row of OR gates
    z <= ( (y(7) or y(6) or y(5) or y(4)) &
          (y(7) or y(6) or y(3) or y(2)) &
          (y(1) or y(3) or y(5) or y(7)) );

end;

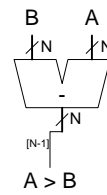
```

Exercise 5.8

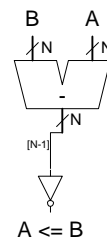
(a)



(b)



(c)

**Exercise 5.9**

(a) Answers will vary.

3 and 5: $3 - 5 = 0011_2 - 0101_2 = 0011_2 + 1010_2 + 1 = 1110_2 (= -2_{10})$. The sign bit (most significant bit) is 1, so the 4-bit signed comparator of Figure 5.12 correctly computes that 3 is less than 5.

(b) Answers will vary.

-3 and 6: $-3 - 6 = 1101 - 0110 = 1101 + 1001 + 1 = 01112 (= -7, \text{ but overflow occurred} - \text{ the result should be } -9)$. The sign bit (most significant bit) is 0, so the 4-bit signed comparator of Figure 5.12 **incorrectly** computes that -3 is **not** less than 6.

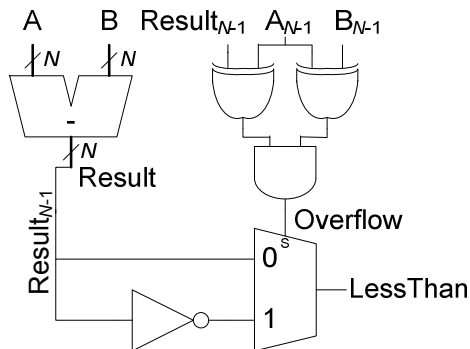
(c) In the general, the N -bit signed comparator of Figure 5.12 operates incorrectly upon overflow.

Exercise 5.10

If no overflow occurs, connect the sign bit (i.e., most significant bit) of the result to the *LessThan* output.

If overflow occurs, invert the sign bit of the result and connect it to the *LessThan* output.

Overflow occurs when (1) the two inputs have different signs, AND (2) the sign of the subtraction result has a different sign than the A input, as shown in the figure below.



We could also have built this as: $LessThan = N \oplus V$, where N is $Result_{N-1}$ and V is the *Overflow* signal.

Exercise 5.11

SystemVerilog

```
module alu(input  logic [31:0] a, b,
          input  logic [1:0] ALUControl,
          output logic [31:0] Result);

    logic [31:0] condinvb;
    logic [32:0] sum;

    assign condinvb = ALUControl[0] ? ~b : b;
    assign sum = a + condinvb + ALUControl[0];

    always_comb
        casex (ALUControl[1:0])
```

```

    2'b0?: Result = sum;
    2'b10: Result = a & b;
    2'b11: Result = a | b;
endcase

```

```
endmodule
```

VHDL

```

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity alu is
    port(a, b:          in  STD_LOGIC_VECTOR(31 downto 0);
         ALUControl: in  STD_LOGIC_VECTOR(1 downto 0);
         Result:        buffer STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of alu is
    signal condinvb: STD_LOGIC_VECTOR(31 downto 0);
    signal sum:      STD_LOGIC_VECTOR(32 downto 0);
begin
    condinvb <= not b when ALUControl(0) else b;
    sum <= ('0', a) + ('0', condinvb) + ALUControl(0);

    process(all) begin
        case? ALUControl(1 downto 0) is
            when "0-" => result <= sum(31 downto 0);
            when "10" => result <= a and b;
            when "11" => result <= a or b;
            when others => result <= (others => '-');
        end case?;
    end process;
end;

```

Exercise 5.12

SystemVerilog

```

module alu(input  logic [31:0] a, b,
           input  logic [1:0] ALUControl,
           output logic [31:0] Result,
           output logic [3:0] ALUFlags);

    logic      neg, zero, carry, overflow;
    logic [31:0] condinvb;
    logic [32:0] sum;

    assign condinvb = ALUControl[0] ? ~b : b;
    assign sum = a + condinvb + ALUControl[0];

    always_comb
        casex (ALUControl[1:0])
            2'b0?: Result = sum;
            2'b10: Result = a & b;
            2'b11: Result = a | b;
        endcase

    assign neg      = Result[31];
    assign zero     = (Result == 32'b0);
    assign carry    = (ALUControl[1] == 1'b0) & sum[32];
    assign overflow = (ALUControl[1] == 1'b0) &
        ~(a[31] ^ b[31] ^ ALUControl[0]) &
        (a[31] ^ sum[31]);

```

```

    assign ALUFlags = {neg, zero, carry, overflow};
endmodule

```

VHDL

```

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity alu is
    port(a, b:          in      STD_LOGIC_VECTOR(31 downto 0);
          ALUControl: in      STD_LOGIC_VECTOR(1 downto 0);
          Result:       buffer STD_LOGIC_VECTOR(31 downto 0);
          ALUFlags:     out     STD_LOGIC_VECTOR(3 downto 0));
end;

architecture behave of alu is
    signal condinvb: STD_LOGIC_VECTOR(31 downto 0);
    signal sum:      STD_LOGIC_VECTOR(32 downto 0);
    signal neg, zero, carry, overflow: STD_LOGIC;
begin
    condinvb <= not b when ALUControl(0) else b;
    sum <= ('0', a) + ('0', condinvb) + ALUControl(0);

    process(all) begin
        case? ALUControl(1 downto 0) is
            when "0-" => result <= sum(31 downto 0);
            when "10" => result <= a and b;
            when "11" => result <= a or b;
            when others => result <= (others => '-');
        end case?;
    end process;

    neg      <= Result(31);
    zero     <= '1' when (Result = 0) else '0';
    carry    <= (not ALUControl(1)) and sum(32);
    overflow <= (not ALUControl(1)) and
                (not (a(31) xor b(31) xor ALUControl(0))) and
                (a(31) xor sum(31));
    ALUFlags <= (neg, zero, carry, overflow);
end;

```

Exercise 5.13

SystemVerilog

```

module testbench();
    logic clk;
    logic [31:0] a, b, y, y_expected;
    logic [1:0] ALUControl;

    logic [31:0] vectornum, errors;
    logic [99:0] testvectors[10000:0];

    // instantiate device under test
    alu dut(a, b, ALUControl, y);

    // generate clock
    always begin
        clk = 1; #50; clk = 0; #50;
    end

    // at start of test, load vectors
    initial begin
        $readmemh("ex5.13_alu.tv", testvectors);
        vectornum = 0; errors = 0;
    end
end

```

```

// apply test vectors at rising edge of clock
always @(posedge clk)
begin
    #1;
    ALUControl = testvectors[vectornum][97:96];
    a = testvectors[vectornum][95:64];
    b = testvectors[vectornum][63:32];
    y_expected = testvectors[vectornum][31:0];
end

// check results on falling edge of clock
always @(negedge clk)
begin
    if (y != y_expected) begin
        $display("Error in vector %d", vectornum);
        $display(" Inputs : a = %h, b = %h, ALUControl = %b", a, b, ALUControl);
        $display(" Outputs: y = %h (%h expected)",
            y, y_expected);
        errors = errors+1;
    end
    vectornum = vectornum + 1;
    if (testvectors[vectornum][0] === 1'bx) begin
        $display("%d tests completed with %d errors", vectornum, errors);
        $stop;
    end
end
endmodule

```

VHDL

```

library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;
entity testbench is -- no inputs or outputs
end;

architecture sim of testbench is
    component alu
        port(a, b:          in      STD_LOGIC_VECTOR(31 downto 0);
              ALUControl: in      STD_LOGIC_VECTOR(1 downto 0);
              Result:       buffer STD_LOGIC_VECTOR(31 downto 0));
    end component;
    signal a, b, Result, Result_expected: STD_LOGIC_VECTOR(31 downto 0);
    signal ALUControl: STD_LOGIC_VECTOR(1 downto 0);
    signal clk, reset: STD_LOGIC;
    constant MEMSIZE: integer := 99;
    type tarray is array(MEMSIZE downto 0) of STD_LOGIC_VECTOR(99 downto 0);
    shared variable testvectors: tarray;
    shared variable vectornum, errors: integer;
begin
    -- instantiate device under test
    dut: alu port map(a, b, ALUControl, Result);

    -- generate clock
    process begin
        clk <= '1'; wait for 5 ns;
        clk <= '0'; wait for 5 ns;
    end process;

    -- at start of test, pulse reset

```



```

process begin
    reset <= '1'; wait for 27 ns; reset <= '0';
    wait;
end process;

-- run tests
-- at start of test, load vectors
process is
    file tv: TEXT;
    variable i, index, count: integer;
    variable L: line;
    variable ch: character;
    variable readvalue: integer;
begin
    -- read file of test vectors
    i := 0;
    index := 0;
    FILE_OPEN(tv, "ex5.13_alu.tv", READ_MODE);
    report "Opening file\n";
    while (not endfile(tv)) loop
        readline(tv, L);
        readvalue := 0;
        count := 3;
        for i in 1 to 28 loop
            read(L, ch);
            report "Line: " & integer'image(index) & " i = " &
                integer'image(i) & " char = " &
                character'image(ch)
                severity error;

            if '0' <= ch and ch <= '9' then
                readvalue := readvalue*16 + character'pos(ch)
                    - character'pos('0');
            elsif 'a' <= ch and ch <= 'f' then
                readvalue := readvalue*16 + character'pos(ch)
                    - character'pos('a')+10;
            else report "Format error on line " &
                integer'image(index) & " i = " &
                integer'image(i) & " char = " &
                character'image(ch)
                severity error;
            end if;

            -- load vectors
            -- assign first 4 bits (will be used for ALUControl)
            if (i = 1) then
                testvectors(index)( 99 downto 96) := CONV_STD_LOGIC_VECTOR(readvalue, 4);
                count := count - 1;
                readvalue := 0; -- reset readvalue

                -- assign a, b, and Result (in testvectors) in
                -- 32-bit increments
                elsif ((i = 10) or (i = 19) or (i = 28)) then
                    testvectors(index)( (count*32 + 31) downto (count*32)) :=
CONV_STD_LOGIC_VECTOR(readvalue, 32);
                    count := count - 1;
                    readvalue := 0; -- reset readvalue
                end if;
            end loop;
            index := index + 1;
        end loop;

        vectornum := 0; errors := 0;
    end process;

```

```

    reset <= '1'; wait for 27 ns; reset <= '0';
    wait;
end process;

-- apply test vectors on rising edge of clk
process (clk) begin
    if (clk'event and clk = '1') then
        ALUControl <= testvectors(vectornum)(97 downto 96)
            after 1 ns;
        a <= testvectors(vectornum)(95 downto 64)
            after 1 ns;
        b <= testvectors(vectornum)(63 downto 32)
            after 1 ns;
        Result_expected <= testvectors(vectornum)(31 downto 0)
            after 1 ns;
    end if;
end process;

-- check results on falling edge of clk
process (clk) begin
    if (clk'event and clk = '0' and reset = '0') then
        if (is_x(testvectors(vectornum))) then
            if (errors = 0) then
                report "Just kidding -- " & integer'image(vectornum) & " tests completed
                    successfully. NO ERRORS." severity failure;
            else
                report integer'image(vectornum) & " tests completed, errors = " &
                    integer'image(errors) severity failure;
            end if;
        end if;

        assert Result = Result_expected
        report "Error: vectornum = " &
            integer'image(vectornum) &
            ", a = " & integer'image(CONV_INTEGER(a)) &
            ", b = " & integer'image(CONV_INTEGER(b)) &
            ", Result = " & integer'image(CONV_INTEGER(Result)) &
            ", ALUControl = " & integer'image(CONV_INTEGER(ALUControl));
        if (Result /= Result_expected) then
            errors := errors + 1;
        end if;
        vectornum := vectornum + 1;
    end if;
end process;
end;

```

Testvector file (ex5.13_alu.tv)

```

0_00000000_00000000_00000000
0_00000000_ffffffff_ffffffff
0_00000001_ffffffff_00000000
0_000000ff_00000001_00000100
1_00000000_00000000_00000000
1_00000000_ffffffff_00000001
1_00000001_00000001_00000000
1_00000100_00000001_000000ff
2_ffffffff_ffffffff_ffffffff
2_ffffffff_12345678_12345678
2_12345678_87654321_02244220
2_00000000_ffffffff_00000000
3_ffffffff_ffffffff_ffffffff
3_12345678_87654321_97755779

```

```
3_00000000_ffffffff_ffffffff
3_00000000_00000000_00000000
```

Exercise 5.14

SystemVerilog

```
module testbench();
    logic clk;
    logic [31:0] a, b, y, y_expected;
    logic [1:0] ALUControl;
    logic [3:0] ALUFlags, ALUFlags_expected;

    logic [31:0] vectornum, errors;
    logic [103:0] testvectors[10000:0];

    // instantiate device under test
    alu dut(a, b, ALUControl, y, ALUFlags);

    // generate clock
    always begin
        clk = 1; #50; clk = 0; #50;
    end

    // at start of test, load vectors
    initial begin
        $readmemh("ex5.14_alu.tv", testvectors);
        vectornum = 0; errors = 0;
    end

    // apply test vectors at rising edge of clock
    always @(posedge clk)
        begin
            #1;
            ALUControl = testvectors[vectornum][101:100];
            a = testvectors[vectornum][99:68];
            b = testvectors[vectornum][67:36];
            y_expected = testvectors[vectornum][35:4];
            ALUFlags_expected = testvectors[vectornum][3:0];
        end

    // check results on falling edge of clock
    always @(negedge clk)
        begin
            if (y != y_expected || ALUFlags != ALUFlags_expected) begin
                $display("Error in vector %d", vectornum);
                $display(" Inputs : a = %h, b = %h, ALUControl = %b", a, b, ALUControl);
                $display(" Outputs: y = %h (%h expected), ALUFlags = %h (%h expected)",
                    y, y_expected, ALUFlags, ALUFlags_expected);
                errors = errors+1;
            end
            vectornum = vectornum + 1;
            if (testvectors[vectornum][0] === 1'bx) begin
                $display("%d tests completed with %d errors", vectornum, errors);
                $stop;
            end
        end
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
```

```

use IEEE.STD_LOGIC_ARITH.all;
entity testbench is -- no inputs or outputs
end;

architecture sim of testbench is
  component alu
    port(a, b:          in      STD_LOGIC_VECTOR(31 downto 0);
         ALUControl: in      STD_LOGIC_VECTOR(1 downto 0);
         Result:       buffer STD_LOGIC_VECTOR(31 downto 0);
         ALUFlags:     out     STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal a, b, Result, Result_expected: STD_LOGIC_VECTOR(31 downto 0);
  signal ALUControl: STD_LOGIC_VECTOR(1 downto 0);
  signal ALUFlags, ALUFlags_expected: STD_LOGIC_VECTOR(3 downto 0);
  signal clk, reset: STD_LOGIC;
  constant MEMSIZE: integer := 99;
  type tarray is array(MEMSIZE downto 0) of STD_LOGIC_VECTOR(103 downto 0);
  shared variable testvectors: tarray;
  shared variable vectornum, errors: integer;
begin
  -- instantiate device under test
  dut: alu port map(a, b, ALUControl, Result, ALUFlags);

  -- generate clock
  process begin
    clk <= '1'; wait for 5 ns;
    clk <= '0'; wait for 5 ns;
  end process;

  -- at start of test, pulse reset
  process begin
    reset <= '1'; wait for 27 ns; reset <= '0';
    wait;
  end process;

  -- run tests
  -- at start of test, load vectors
  process is
    file tv: TEXT;
    variable i, index, count: integer;
    variable L: line;
    variable ch: character;
    variable readvalue: integer;
  begin
    -- read file of test vectors
    i := 0;
    index := 0;
    FILE_OPEN(tv, "ex5.14_alu.tv", READ_MODE);
    report "Opening file\n";
    while (not endfile(tv)) loop
      readline(tv, L);
      readvalue := 0;
      count := 3;
      for i in 1 to 30 loop
        read(L, ch);
        report "Line: " & integer'image(index) & " i = " &
          integer'image(i) & " char = " &
          character'image(ch)
          severity error;

        if '0' <= ch and ch <= '9' then
          readvalue := readvalue*16 + character'pos(ch)
            - character'pos('0');
        end if;
      end loop;
    end loop;
  end process;

```

```

    elsif 'a' <= ch and ch <= 'f' then
        readvalue := readvalue*16 + character'pos(ch)
        - character'pos('a')+10;
    else report "Format error on line " &
        integer'image(index) & " i = " &
        integer'image(i) & " char = " &
        character'image(ch)
        severity error;
    end if;

    -- load vectors
    -- assign first 4 bits (will be used for ALUControl)
    if (i = 1) then
        testvectors(index)( 103 downto 100) := CONV_STD_LOGIC_VECTOR(readvalue, 4);
        count := count - 1;
        readvalue := 0; -- reset readvalue

        -- assign a, b, and Result (in testvectors) in
        -- 32-bit increments
        elsif ((i = 10) or (i = 19) or (i = 28)) then
            testvectors(index)( (count*32 + 35) downto (count*32+4)) :=
CONV_STD_LOGIC_VECTOR(readvalue, 32);
            count := count - 1;
            readvalue := 0; -- reset readvalue
        -- assign ALUFlags (in testvectors)
        elsif (i=30) then
            testvectors(index)(3 downto 0) := CONV_STD_LOGIC_VECTOR(readvalue, 4);
        end if;
    end loop;
    index := index + 1;
end loop;

vectornum := 0; errors := 0;
reset <= '1'; wait for 27 ns; reset <= '0';
wait;
end process;

-- apply test vectors on rising edge of clk
process (clk) begin
    if (clk'event and clk = '1') then
        ALUControl <= testvectors(vectornum)(101 downto 100)
        after 1 ns;
        a <= testvectors(vectornum)(99 downto 68)
        after 1 ns;
        b <= testvectors(vectornum)(67 downto 36)
        after 1 ns;
        Result_expected <= testvectors(vectornum)(35 downto 4)
        after 1 ns;
        ALUFlags_expected <= testvectors(vectornum)(3 downto 0)
        after 1 ns;
    end if;
end process;

-- check results on falling edge of clk
process (clk) begin
    if (clk'event and clk = '0' and reset = '0') then
        if (is_x(testvectors(vectornum))) then
            if (errors = 0) then
                report "Just kidding -- " & integer'image(vectornum) & " tests completed
successfully. NO ERRORS." severity failure;
            else
                report integer'image(vectornum) & " tests completed, errors = " &
integer'image(errors) severity failure;
            end if;
        end if;
    end if;
end process;

```

```

        end if;
    end if;

    assert Result = Result_expected
    report "Error: vectornum = " &
    integer'image(vectornum) &
    ", a = " & integer'image(CONV_INTEGER(a)) &
    ", b = " & integer'image(CONV_INTEGER(b)) &
    ", Result = " & integer'image(CONV_INTEGER(Result)) &
    ", ALUControl = " & integer'image(CONV_INTEGER(ALUControl));
    assert ALUFlags = ALUFlags_expected
    report "Error: ALUFlags = " & integer'image(CONV_INTEGER(ALUFlags));
    if ( (Result /= Result_expected) or
        (ALUFlags /= ALUFlags_expected) ) then
        errors := errors + 1;
    end if;
    vectornum := vectornum + 1;
end if;
end process;
end;

```

Testvectors file (ex5.14_alu.tv)

```

0_00000000_00000000_00000000_4
0_00000000_ffffffff_ffffffff_8
0_00000001_ffffffff_00000000_6
0_000000ff_00000001_00000100_0
1_00000000_00000000_00000000_6
1_00000000_ffffffff_00000001_0
1_00000001_00000001_00000000_6
1_00000100_00000001_000000ff_2
2_ffffffff_ffffffff_ffffffff_8
2_ffffffff_12345678_12345678_0
2_12345678_87654321_02244220_0
2_00000000_ffffffff_00000000_4
3_ffffffff_ffffffff_ffffffff_8
3_12345678_87654321_97755779_8
3_00000000_ffffffff_ffffffff_8
3_00000000_00000000_00000000_4

```

Exercise 5.15

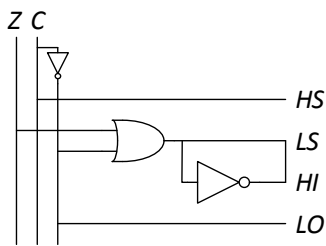
(a) $HS = C$

$LS = Z + \bar{C}$

$HI = \bar{Z}C = \overline{LS}$

$LO = \bar{C} = \overline{HS}$

(b)



Exercise 5.16

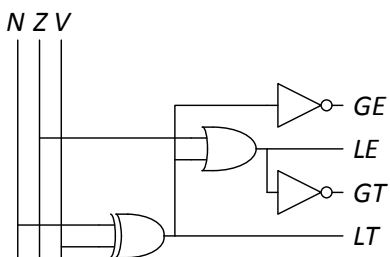
(a) $GE = \overline{N \oplus V}$

$$LE = Z + (N \oplus V)$$

$$GT = \overline{LE} = \overline{Z(N \oplus V)}$$

$$LT = \overline{GE} = N \oplus V$$

(b)



Exercise 5.17

A 2-bit left shifter creates the output by appending two zeros to the least significant bits of the input and dropping the two most significant bits.

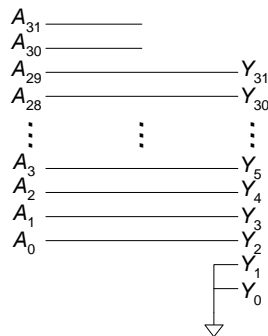


FIGURE 5.6 2-bit left shifter, 32-bit input and output

2-bit Left Shifter

SystemVerilog

```
module leftshift2_32(input  logic [31:0] a,
                    output logic [31:0] y);
    assign y = {a[29:0], 2'b0};
endmodule
```

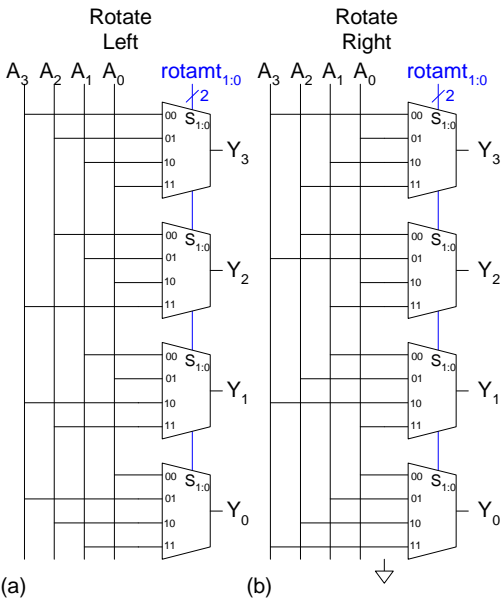
VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity leftshift2_32 is
    port(a: in  STD_LOGIC_VECTOR(31 downto 0);
         y: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture synth of leftshift2_32 is
begin
    y <= a(29 downto 0) & "00";
end;
```

Exercise 5.18



4-bit Left and Right Rotator

SystemVerilog

```

module    ex5_14(a,    right_rotated,    left_rotated,
shamt);
    input  logic [3:0] a;
    output logic [3:0] right_rotated;
    output logic [3:0] left_rotated;
    input  logic [1:0] shamt;

    // right rotated
    always_comb
    case(shamt)
        2'b00: right_rotated = a;
        2'b01: right_rotated =
            {a[0], a[3], a[2], a[1]};
        2'b10: right_rotated =
            {a[1], a[0], a[3], a[2]};
        2'b11: right_rotated =
            {a[2], a[1], a[0], a[3]};
        default: right_rotated = 4'bxxxx;
    endcase

    // left rotated
    always_comb
    case(shamt)
        2'b00: left_rotated = a;
        2'b01: left_rotated =
            {a[2], a[1], a[0], a[3]};
        2'b10: left_rotated =
            {a[1], a[0], a[3], a[2]};
        2'b11: left_rotated =
            {a[0], a[3], a[2], a[1]};
        default: left_rotated = 4'bxxxx;
    endcase
endmodule

```

VHDL

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity ex5_14 is
    port(a:    in STD_LOGIC_VECTOR(3 downto 0);
          right_rotated, left_rotated: out
              STD_LOGIC_VECTOR(3 downto 0);
          shamt: in STD_LOGIC_VECTOR(1 downto 0));
end;

architecture synth of ex5_14 is
begin

-- right-rotated
process(all) begin
    case shamt is
        when "00" => right_rotated <= a;
        when "01" => right_rotated <=
            (a(0), a(3), a(2), a(1));
        when "10" => right_rotated <=
            (a(1), a(0), a(3), a(2));
        when "11" => right_rotated <=
            (a(2), a(1), a(0), a(3));
        when others => right_rotated <= "XXXX";
    end case;
end process;

-- left-rotated
process(all) begin
    case shamt is
        when "00" => left_rotated <= a;
        when "01" => left_rotated <=
            (a(2), a(1), a(0), a(3));
        when "10" => left_rotated <=
            (a(1), a(0), a(3), a(2));
        when "11" => left_rotated <=
            (a(0), a(3), a(2), a(1));
        when others => left_rotated <= "XXXX";
    end case;
end process;
end;

```

Exercise 5.19

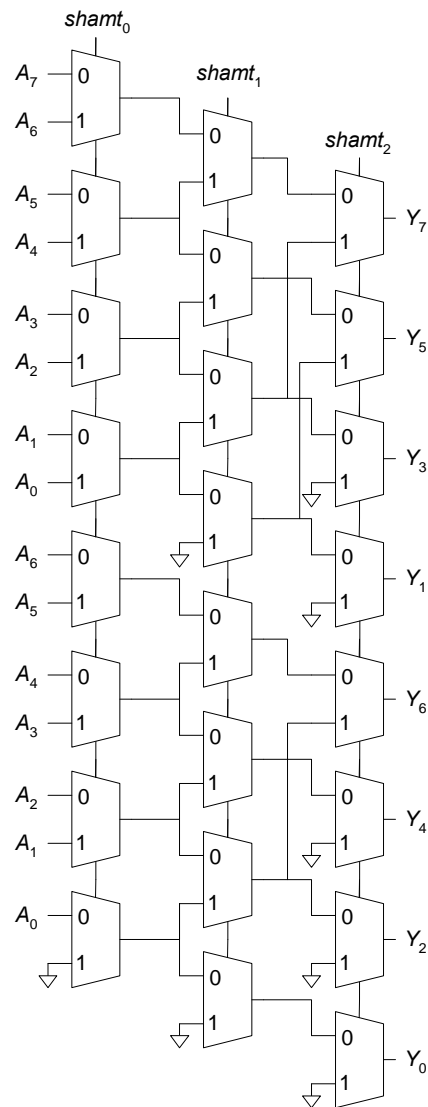


FIGURE 5.7 8-bit left shifter using 24 2:1 multiplexers

Exercise 5.20

Any N -bit shifter can be built by using $\log_2 N$ columns of 2-bit shifters. The first column of multiplexers shifts or rotates 0 to 1 bit, the second column shifts or rotates 0 to 3 bits, the following 0 to 7 bits, etc. until the final column shifts or rotates 0 to $N-1$ bits. The second column of multiplexers takes its inputs from the first column of multiplexers, the third column takes its input from the second column, and so forth. The 1-bit select input of each column is a single bit of the *shamt* (shift amount) control signal, with the least significant bit for the left-most column and the most significant bit for the right-most column.

Exercise 5.21

- (a) $B = 0, C = A, k = \text{shamt}$
- (b) $B = A_{N-1}$ (the most significant bit of A), repeated N times to fill all N bits of B
- (c) $B = A, C = 0, k = N - \text{shamt}$
- (d) $B = A, C = A, k = \text{shamt}$
- (e) $B = A, C = A, k = N - \text{shamt}$

Exercise 5.22

$$t_{pd_MULT4} = t_{AND} + 8t_{FA}$$

For $N = 1$, the delay is t_{AND} . For $N > 1$, an $N \times N$ multiplier has N -bit operands, N partial products, and $N-1$ stages of 1-bit adders. The delay is through the AND gate, then through all N adders in the first stage, and finally through 2 adder delays for each of the remaining stages. So the propagation is:

$$t_{pd_MULTN} = t_{AND} + [N + 2(N-1)]t_{FA}$$

Exercise 5.23

$$t_{pd_DIV4} = 4(4t_{FA} + t_{MUX}) = 16t_{FA} + 4t_{MUX}$$

$$t_{pd_DIVN} = N^2 t_{FA} + N t_{MUX}$$

Exercise 5.24

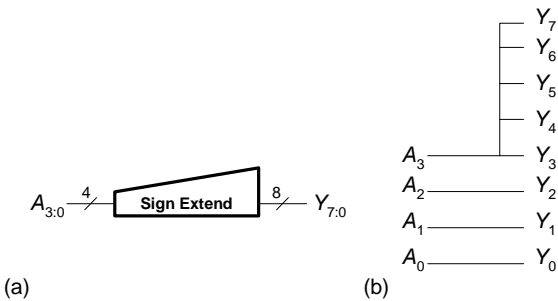


FIGURE 5.9 Sign extension unit (a) symbol, (b) underlying hardware

SystemVerilog

```
module signext4_8(input  logic [3:0] a,
                  output logic [7:0] y);

    assign y = { 4{a[3]}, a};

endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity signext4_8 is
    port(a: in  STD_LOGIC_VECTOR(3 downto 0);
          y: out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of signext4_8 is
begin
```

Exercise 5.26

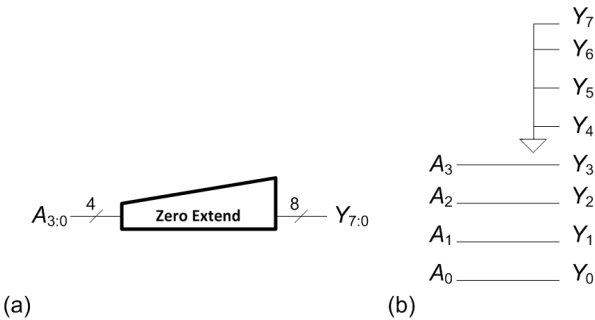


FIGURE 5.10 Zero extension unit (a) symbol, (b) underlying hardware

Exercise 5.29

- (a) $1000\ 1101\ .\ 1001\ 0000 = 0x8D90$
- (b) $0010\ 1010\ .\ 0101\ 0000 = 0x2A50$
- (c) $1001\ 0001\ .\ 0010\ 1000 = 0x9128$

Exercise 5.30

- (a) $111110.100000 = 0xFA0$
- (b) $010000.010000 = 0x410$
- (c) $101000.000101 = 0xA05$

Exercise 5.31

- (a) $1111\ 0010\ .\ 0111\ 0000 = 0xF270$
- (b) $0010\ 1010\ .\ 0101\ 0000 = 0x2A50$
- (c) $1110\ 1110\ .\ 1101\ 1000 = 0xEED8$

Exercise 5.32

- (a) $100001.100000 = 0x860$
- (b) $010000.010000 = 0x410$
- (c) $110111.111011 = 0xDFB$

Exercise 5.33

- (a) $-1101.1001 = -1.1011001 \times 2^3$
Thus, the biased exponent $= 127 + 3 = 130 = 1000\ 0010_2$
In IEEE 754 single-precision floating-point format:
 $1\ 1000\ 0010\ 101\ 1001\ 0000\ 0000\ 0000\ 0000 = \mathbf{0xC1590000}$
- (b) $101010.0101 = 1.010100101 \times 2^5$
Thus, the biased exponent $= 127 + 5 = 132 = 1000\ 0100_2$
In IEEE 754 single-precision floating-point format:
 $0\ 1000\ 0100\ 010\ 1001\ 0100\ 0000\ 0000\ 0000 = \mathbf{0x42294000}$
- (c) $-10001.00101 = -1.000100101 \times 2^4$
Thus, the biased exponent $= 127 + 4 = 131 = 1000\ 0011_2$
In IEEE 754 single-precision floating-point format:
 $1\ 1000\ 0011\ 000\ 1001\ 0100\ 0000\ 0000\ 0000 = \mathbf{0xC1894000}$

Exercise 5.34

(a) $-11110.1 = -1.111101 \times 2^4$

Thus, the biased exponent $= 127 + 4 = 131 = 1000\ 0011_2$

In IEEE 754 single-precision floating-point format:

1 1000 0011 111 1010 0000 0000 0000 0000 = **0xC1F40000**

(b) $10000.01 = 1.000001 \times 2^4$

Thus, the biased exponent $= 127 + 4 = 131 = 1000\ 0011_2$

In IEEE 754 single-precision floating-point format:

0 1000 0011 000 0010 0100 0000 0000 0000 = **0x41820000**

(c) $-1000.000101 = -1.000000101 \times 2^3$

Thus, the biased exponent $= 127 + 3 = 130 = 1000\ 0010_2$

In IEEE 754 single-precision floating-point format:

1 1000 0010 000 0001 0100 0000 0000 0000 = **0xC1014000**

Exercise 5.35

(a) 5.5

(b) $-0000.0001_2 = -0.0625$

(c) -8

Exercise 5.36

(a) 29.65625

(b) -25.1875

(c) -23.875

Exercise 5.37

When adding two floating point numbers, the number with the smaller exponent is shifted to preserve the most significant bits. For example, suppose we were adding the two floating point numbers 1.0×2^0 and 1.0×2^{-27} . We make the two exponents equal by shifting the second number right by 27 bits. Because the mantissa is limited to 24 bits, the second number ($1.000\ 0000\ 0000\ 0000\ 0000 \times 2^{-27}$) becomes $0.000\ 0000\ 0000\ 0000\ 0000 \times 2^0$, because the 1 is shifted off to the right. If we had shifted the number with the larger exponent (1.0×2^0) to the left, we would have shifted off the more significant bits (on the order of 2^0 instead of on the order of 2^{-27}).

Exercise 5.38

- (a) C0123456
- (b) D1E072C3
- (c) 5F19659A

Exercise 5.39

- (a)
- $$\begin{aligned} 0xC0D20004 &= 1\ 1000\ 0001\ 101\ 0010\ 0000\ 0000\ 0000\ 0100 \\ &= -1.101\ 0010\ 0000\ 0000\ 0000\ 01 \times 2^2 \\ 0x72407020 &= 0\ 1110\ 0100\ 100\ 0000\ 0111\ 0000\ 0010\ 0000 \\ &= 1.100\ 0000\ 0111\ 0000\ 001 \times 2^{101} \end{aligned}$$

When adding these two numbers together, 0xC0D20004 becomes:

0×2^{101} because all of the significant bits shift off the right when making the exponents equal. Thus, the result of the addition is simply the second number:

0x72407020

- (b)
- $$\begin{aligned} 0xC0D20004 &= 1\ 1000\ 0001\ 101\ 0010\ 0000\ 0000\ 0000\ 0100 \\ &= -1.101\ 0010\ 0000\ 0000\ 0000\ 01 \times 2^2 \\ 0x40DC0004 &= 0\ 1000\ 0001\ 101\ 1100\ 0000\ 0000\ 0000\ 0100 \\ &= 1.101\ 1100\ 0000\ 0000\ 0000\ 01 \times 2^2 \end{aligned}$$

$$\begin{aligned} &1.101\ 1100\ 0000\ 0000\ 0000\ 01 \times 2^2 \\ &- 1.101\ 0010\ 0000\ 0000\ 0000\ 01 \times 2^2 \\ &= 0.000\ 1010 \qquad \qquad \qquad \times 2^2 \\ &= 1.010 \times 2^{-2} \\ &= 0\ 0111\ 1101\ 010\ 0000\ 0000\ 0000\ 0000\ 0000 \\ &= 0x3EA00000 \end{aligned}$$

- (c)
- $$\begin{aligned} 0x5FBE4000 &= 0\ 1011\ 1111\ 011\ 1110\ 0100\ 0000\ 0000\ 0000\ 0000 \\ &= 1.011\ 1110\ 01 \times 2^{64} \\ 0x3FF80000 &= 0\ 0111\ 1111\ 111\ 1000\ 0000\ 0000\ 0000\ 0000 \\ &= 1.111\ 1 \times 2^0 \\ 0xDFDE4000 &= 1\ 1011\ 1111\ 101\ 1110\ 0100\ 0000\ 0000\ 0000\ 0000 \\ &= -1.101\ 1110\ 01 \times 2^{64} \end{aligned}$$

Thus, $(1.011\ 1110\ 01 \times 2^{64} + 1.111\ 1 \times 2^0) = 1.011\ 1110\ 01 \times 2^{64}$

And, $(1.011\ 1110\ 01 \times 2^{64} + 1.111\ 1 \times 2^0) - 1.101\ 1110\ 01 \times 2^{64} =$
 $- 0.01 \times 2^{64} = -1.0 \times 2^{64}$
 $= 1\ 1011\ 1101\ 000\ 0000\ 0000\ 0000\ 0000\ 0000$
 $= \mathbf{0xDE800000}$

This is counterintuitive because the second number (0x3FF80000) does not affect the result because its order of magnitude is less than 2^{23} of the other numbers. This second number's significant bits are shifted off when the exponents are made equal.

Exercise 5.40

We only need to change step 5.

1. Extract exponent and fraction bits.
2. Prepend leading 1 to form the mantissa.
3. Compare exponents.
4. Shift smaller mantissa if necessary.
5. If one number is negative: Subtract it from the other number. If the result is negative, take the absolute value of the result and make the sign bit 1.
 If both numbers are negative: Add the numbers and make the sign bit 1.
 If both numbers are positive: Add the numbers and make the sign bit 0.
6. Normalize mantissa and adjust exponent if necessary.
7. Round result
8. Assemble exponent and fraction back into floating-point number

Exercise 5.41

(a) $2(2^{31} - 1 - 2^{23}) = 2^{32} - 2 - 2^{24} = 4,278,190,078$

(b) $2(2^{31} - 1) = 2^{32} - 2 = 4,294,967,294$

(c) $\pm\infty$ and NaN are given special representations because they are often used in calculations and in representing results. These values also give useful information to the user as return values, instead of returning garbage upon overflow, underflow, or divide by zero.

Exercise 5.42

$$\begin{aligned}
 \text{(a) } 245 &= 11110101 = 1.1110101 \times 2^7 \\
 &= 0\ 1000\ 0110\ 111\ 0101\ 0000\ 0000\ 0000\ 0000 \\
 &= 0x43750000 \\
 0.0625 &= 0.0001 = 1.0 \times 2^{-4} \\
 &= 0\ 0111\ 1011\ 000\ 0000\ 0000\ 0000\ 0000\ 0000 \\
 &= 0x3D800000
 \end{aligned}$$

(b) 0x43750000 is greater than 0x3D800000, so magnitude comparison gives the correct result.

$$\begin{aligned}
 \text{(c)} \\
 1.1110101 \times 2^7 &= 0\ 0000\ 0111\ 111\ 0101\ 0000\ 0000\ 0000\ 0000 \\
 &= 0x03F50000 \\
 1.0 \times 2^{-4} &= 0\ 1111\ 1100\ 000\ 0000\ 0000\ 0000\ 0000\ 0000 \\
 &= 0x7E000000
 \end{aligned}$$

(d) No, integer comparison no longer works. $7E000000 > 03F50000$ (indicating that 1.0×2^{-4} is greater than 1.1110101×2^7 , which is incorrect.)

(e) It is convenient for integer comparison to work with floating-point numbers because then the computer can compare numbers without needing to extract the mantissa, exponent, and sign.

Exercise 5.43

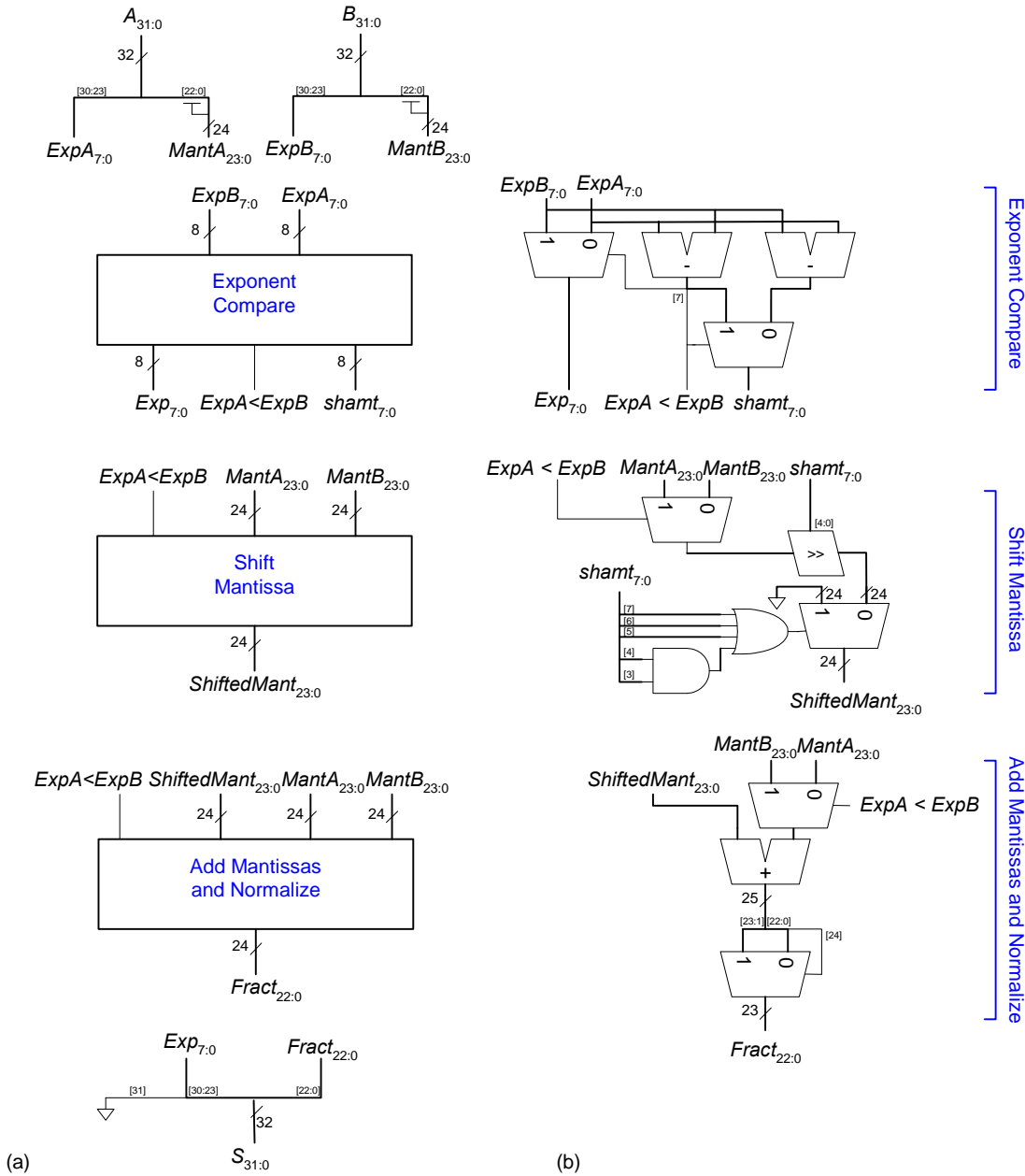


FIGURE 5.11 Floating-point adder hardware: (a) block diagram, (b) underlying hardware

SystemVerilog

```

module fpadd(input  logic [31:0] a, b,
             output logic [31:0] s);

    logic [7:0]  expa, expb, exp_pre, exp, shamt;
    logic        alessb;
    logic [23:0] manta, mantb, shmant;
    logic [22:0] fract;

    assign {expa, manta} = {a[30:23], 1'b1, a[22:0]};
    assign {expb, mantb} = {b[30:23], 1'b1, b[22:0]};
    assign s          = {1'b0, exp, fract};

    expcomp    expcompl(expa, expb, alessb, exp_pre,
                        shamt);
    shiftmant  shiftmant1(alessb, manta, mantb,
                        shamt, shmant);
    addmant    addmant1(alessb, manta, mantb,
                        shmant, exp_pre, fract, exp);

endmodule

```

VHDL

```

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;

entity fpadd is
    port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
         s:  out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture synth of fpadd is
    component expcomp
        port(expa, expb: in  STD_LOGIC_VECTOR(7 downto 0);
             alessb:  inout STD_LOGIC;
             exp,shamt: out STD_LOGIC_VECTOR(7 downto 0));
    end component;

    component shiftmant
        port(alessb: in  STD_LOGIC;
             manta: in  STD_LOGIC_VECTOR(23 downto 0);
             mantb: in  STD_LOGIC_VECTOR(23 downto 0);
             shamt: in  STD_LOGIC_VECTOR(7 downto 0);
             shmant: out STD_LOGIC_VECTOR(23 downto 0));
    end component;

    component addmant
        port(alessb: in  STD_LOGIC;
             manta: in  STD_LOGIC_VECTOR(23 downto 0);
             mantb: in  STD_LOGIC_VECTOR(23 downto 0);
             shmant: in  STD_LOGIC_VECTOR(23 downto 0);
             exp_pre: in  STD_LOGIC_VECTOR(7 downto 0);
             fract: out STD_LOGIC_VECTOR(22 downto 0);
             exp: out STD_LOGIC_VECTOR(7 downto 0));
    end component;

    signal expa, expb: STD_LOGIC_VECTOR(7 downto 0);
    signal exp_pre, exp: STD_LOGIC_VECTOR(7 downto 0);
    signal shamt: STD_LOGIC_VECTOR(7 downto 0);
    signal alessb: STD_LOGIC;
    signal manta: STD_LOGIC_VECTOR(23 downto 0);
    signal mantb: STD_LOGIC_VECTOR(23 downto 0);
    signal shmant: STD_LOGIC_VECTOR(23 downto 0);
    signal fract: STD_LOGIC_VECTOR(22 downto 0);

begin

    expa <= a(30 downto 23);
    manta <= '1' & a(22 downto 0);
    expb <= b(30 downto 23);
    mantb <= '1' & b(22 downto 0);

    s <= '0' & exp & fract;

    expcompl: expcomp
        port map(expa, expb, alessb, exp_pre, shamt);
    shiftmant1: shiftmant
        port map(alessb, manta, mantb, shamt, shmant);
    addmant1: addmant
        port map(alessb, manta, mantb, shmant,
                exp_pre, fract, exp);

end;

```

*(continued from previous page)***SystemVerilog**

```

module expcomp(input  logic [7:0] expa, expb,
               output logic   alessb,
               output logic [7:0] exp, shamt);
    logic [7:0] aminusb, bminusa;

    assign aminusb = expa - expb;
    assign bminusa = expb - expa;
    assign alessb  = aminusb[7];

    always_comb
        if (alessb) begin
            exp = expb;
            shamt = bminusa;
        end
        else begin
            exp = expa;
            shamt = aminusb;
        end
    end
endmodule

```

VHDL

```

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;

entity expcomp is
    port(expa, expb: in  STD_LOGIC_VECTOR(7 downto 0);
          alessb:   inout STD_LOGIC;
          exp,shamt: out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of expcomp is
    signal aminusb: STD_LOGIC_VECTOR(7 downto 0);
    signal bminusa: STD_LOGIC_VECTOR(7 downto 0);
begin
    aminusb <= expa - expb;
    bminusa <= expb - expa;
    alessb <= aminusb(7);

    exp <= expb when alessb = '1' else expa;
    shamt <= bminusa when alessb = '1' else aminusb;

end;

```

(continued on next page)

*(continued from previous page)***SystemVerilog**

```

module shiftmant(input  logic alessb,
                 input  logic [23:0] manta, mantb,
                 input  logic [7:0] shamt,
                 output logic [23:0] shmant);

    logic [23:0] shiftedval;

    assign shiftedval = alessb ?
        (manta >> shamt) : (mantb >> shamt);

    always_comb
        if (shamt[7] | shamt[6] | shamt[5] |
            (shamt[4] & shamt[3]))
            shmant = 24'b0;
        else
            shmant = shiftedval;

endmodule

module addmant(input  logic      alessb,
               input  logic [23:0] manta,
               input  logic [23:0] mantb, shmant,
               input  logic [7:0]  exp_pre,
               output logic [22:0] fract,
               output logic [7:0]  exp);

    logic [24:0] addresult;
    logic [23:0] addval;

    assign addval    = alessb ? mantb : manta;
    assign addresult = shmant + addval;
    assign fract     = addresult[24] ?
        addresult[23:1] :
        addresult[22:0];

    assign exp       = addresult[24] ?
        (exp_pre + 1) :
        exp_pre;

endmodule

```

VHDL

```

library IEEE; use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;
use IEEE.std_logic_unsigned.all;

entity shiftmant is
    port(alessb: in  STD_LOGIC;
          manta: in  STD_LOGIC_VECTOR(23 downto 0);
          mantb: in  STD_LOGIC_VECTOR(23 downto 0);
          shamt: in  STD_LOGIC_VECTOR(7 downto 0);
          shmant: out STD_LOGIC_VECTOR(23 downto 0));
end;

architecture synth of shiftmant is
    signal shiftedval: unsigned(23 downto 0);
    signal shiftamt_vector: STD_LOGIC_VECTOR(7 downto 0);
begin

    shiftedval <= SHIFT_RIGHT(unsigned(manta), to_integer(unsigned(shamt))) when alessb = '1'
        else SHIFT_RIGHT(unsigned(mantb), to_integer(unsigned(shamt)));

    shmant <= X"000000" when (shamt > 22)
        else STD_LOGIC_VECTOR(shiftedval);

end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;

entity addmant is
    port(alessb: in  STD_LOGIC;
          manta: in  STD_LOGIC_VECTOR(23 downto 0);
          mantb: in  STD_LOGIC_VECTOR(23 downto 0);
          shmant: in  STD_LOGIC_VECTOR(23 downto 0);
          exp_pre: in  STD_LOGIC_VECTOR(7 downto 0);
          fract: out  STD_LOGIC_VECTOR(22 downto 0);
          exp: out  STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of addmant is
    signal addresult: STD_LOGIC_VECTOR(24 downto 0);
    signal addval: STD_LOGIC_VECTOR(23 downto 0);
begin
    addval <= mantb when alessb = '1' else manta;
    addresult <= ('0' & shmant) + addval;
    fract <= addresult(23 downto 1)
        when addresult(24) = '1'
        else addresult(22 downto 0);
    exp <= (exp_pre + 1)
        when addresult(24) = '1'
        else exp_pre;

end;

```


Exercise 5.44

(a)

- Extract exponent and fraction bits.
- Prepend leading 1 to form the mantissa.
- Add exponents.
- Multiply mantissas.
- Round result and truncate mantissa to 24 bits.
- Assemble exponent and fraction back into floating-point number

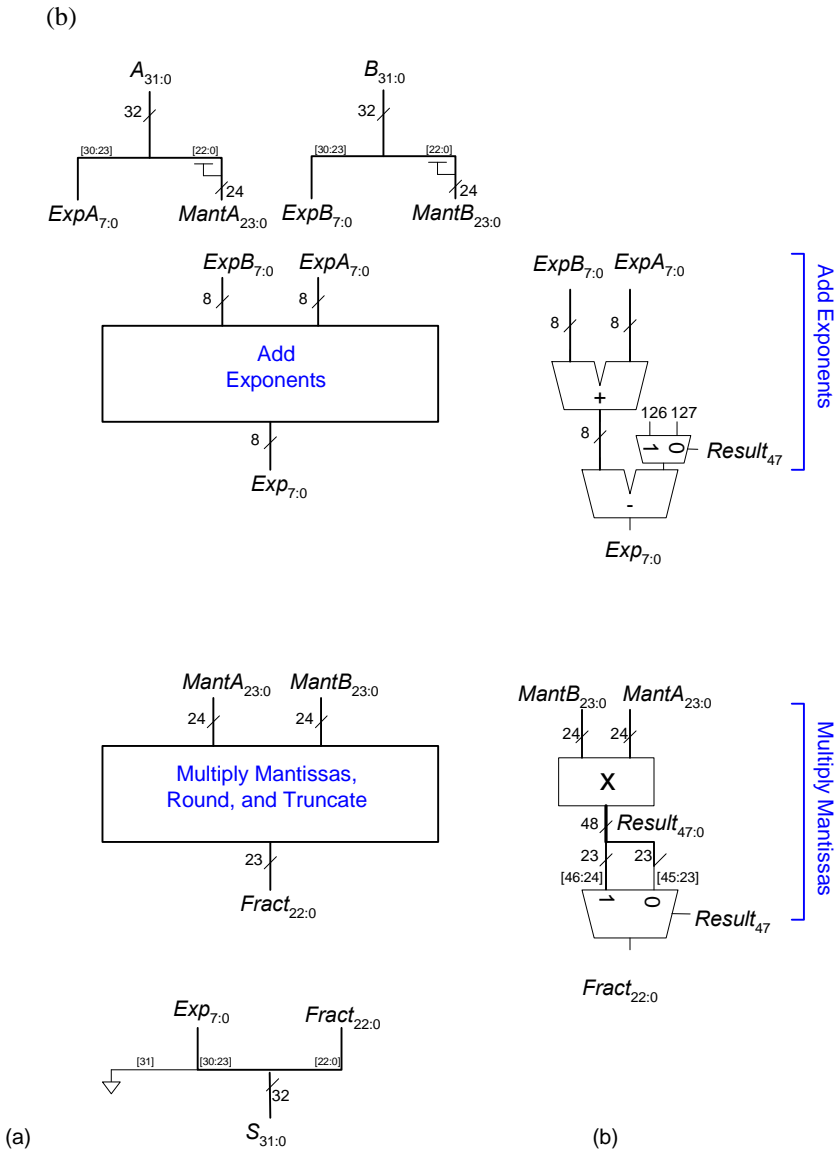


FIGURE 5.12 Floating-point multiplier block diagram

(c)

SystemVerilog

```

module fpmult(input  logic [31:0] a, b,
              output logic [31:0] m);

    logic [7:0]  expa, expb, exp;
    logic [23:0] manta, mantb;
    logic [22:0] fract;
    logic [47:0] result;

    assign {expa, manta} = {a[30:23], 1'b1, a[22:0]};
    assign {expb, mantb} = {b[30:23], 1'b1, b[22:0]};
    assign m             = {1'b0, exp, fract};

    assign result = manta * mantb;
    assign fract = result[47] ?
        result[46:24] :
        result[45:23];

    assign exp = result[47] ?
        (expa + expb - 126) :
        (expa + expb - 127);

endmodule

```

VHDL

```

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;

entity fpmult is
    port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
          m:      out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture synth of fpmult is
    signal expa, expb, exp:
        STD_LOGIC_VECTOR(7 downto 0);
    signal manta, mantb:
        STD_LOGIC_VECTOR(23 downto 0);
    signal fract:
        STD_LOGIC_VECTOR(22 downto 0);
    signal result:
        STD_LOGIC_VECTOR(47 downto 0);
begin
    expa  <= a(30 downto 23);
    manta <= '1' & a(22 downto 0);
    expb  <= b(30 downto 23);
    mantb <= '1' & b(22 downto 0);

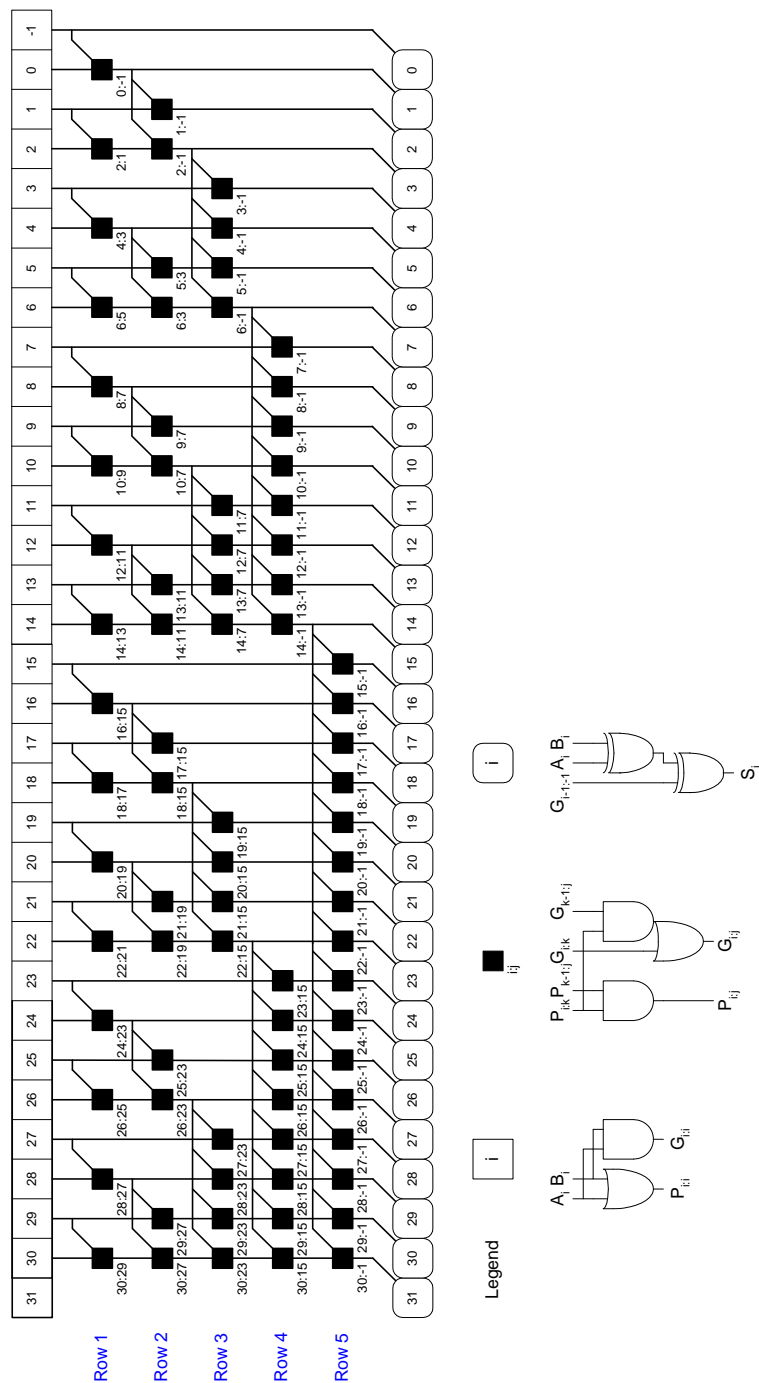
    m      <= '0' & exp & fract;
    result <= manta * mantb;
    fract  <= result(46 downto 24)
        when (result(47) = '1')
        else result(45 downto 23);
    exp    <= (expa + expb - 126)
        when (result(47) = '1')
        else (expa + expb - 127);

end;

```

Exercise 5.45

(a) Figure on next page



5.45 (b)

SystemVerilog

```

module prefixadd(input  logic [31:0] a, b,
                 input  logic      cin,
                 output logic [31:0] s,
                 output logic      cout);

    logic [30:0] p, g;
    // p and g prefixes for rows 1 - 5
    logic [15:0] p1, p2, p3, p4, p5;
    logic [15:0] g1, g2, g3, g4, g5;

    pandg row0(a, b, p, g);
    blackbox row1({p[30],p[28],p[26],p[24],p[22],
                  p[20],p[18],p[16],p[14],p[12],
                  p[10],p[8],p[6],p[4],p[2],p[0]},
                 {p[29],p[27],p[25],p[23],p[21],
                  p[19],p[17],p[15],p[13],p[11],
                  p[9],p[7],p[5],p[3],p[1],1'b0},
                 {g[30],g[28],g[26],g[24],g[22],
                  g[20],g[18],g[16],g[14],g[12],
                  g[10],g[8],g[6],g[4],g[2],g[0]},
                 {g[29],g[27],g[25],g[23],g[21],
                  g[19],g[17],g[15],g[13],g[11],
                  g[9],g[7],g[5],g[3],g[1],cin},
                 p1, g1);

```

VHDL

```

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity prefixadd is
    port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
          cin: in  STD_LOGIC;
          s:  out STD_LOGIC_VECTOR(31 downto 0);
          cout: out STD_LOGIC);
end;

architecture synth of prefixadd is
    component pgblock
        port(a, b: in  STD_LOGIC_VECTOR(30 downto 0);
              p, g: out STD_LOGIC_VECTOR(30 downto 0));
    end component;

    component pgblackblock is
        port (pik, gik: in STD_LOGIC_VECTOR(15 downto 0);
              pkj, gkj: in STD_LOGIC_VECTOR(15 downto 0);
              pij: out STD_LOGIC_VECTOR(15 downto 0);
              gij: out STD_LOGIC_VECTOR(15 downto 0));
    end component;

    component subblock is
        port (a, b, g: in STD_LOGIC_VECTOR(31 downto 0);
              s:      out STD_LOGIC_VECTOR(31 downto 0));
    end component;

    signal p, g: STD_LOGIC_VECTOR(30 downto 0);
    signal pik_1, pik_2, pik_3, pik_4, pik_5,
           gik_1, gik_2, gik_3, gik_4, gik_5,
           pkj_1, pkj_2, pkj_3, pkj_4, pkj_5,
           gkj_1, gkj_2, gkj_3, gkj_4, gkj_5,
           p1, p2, p3, p4, p5,
           g1, g2, g3, g4, g5:
        STD_LOGIC_VECTOR(15 downto 0);
    signal g6:  STD_LOGIC_VECTOR(31 downto 0);

begin
    row0: pgblock
        port map(a(30 downto 0), b(30 downto 0), p, g);

    pik_1 <=
        (p(30) & p(28) & p(26) & p(24) & p(22) & p(20) & p(18) & p(16) &
         p(14) & p(12) & p(10) & p(8) & p(6) & p(4) & p(2) & p(0));
    gik_1 <=
        (g(30) & g(28) & g(26) & g(24) & g(22) & g(20) & g(18) & g(16) &
         g(14) & g(12) & g(10) & g(8) & g(6) & g(4) & g(2) & g(0));
    pkj_1 <=
        (p(29) & p(27) & p(25) & p(23) & p(21) & p(19) & p(17) & p(15) &
         p(13) & p(11) & p(9) & p(7) & p(5) & p(3) & p(1) & '0');
    gkj_1 <=
        (g(29) & g(27) & g(25) & g(23) & g(21) & g(19) & g(17) & g(15) &
         g(13) & g(11) & g(9) & g(7) & g(5) & g(3) & g(1) & cin);

    row1: pgblackblock
        port map(pik_1, gik_1, pkj_1, gkj_1,
                 p1, g1);

```

(continued on next page)
(continued from previous page)

SystemVerilog

```
blackbox row2({p1[15],p[29],p1[13],p[25],p1[11],
  p[21],p1[9],p[17],p1[7],p[13],
  p1[5],p[9],p1[3],p[5],p1[1],p[1]},
  {{2{p1[14]}},{2{p1[12]}},{2{p1[10]}},
  {2{p1[8]}},{2{p1[6]}},{2{p1[4]}},
  {2{p1[2]}},{2{p1[0]}}},
  {g1[15],g[29],g1[13],g[25],g1[11],
  g[21],g1[9],g[17],g1[7],g[13],
  g1[5],g[9],g1[3],g[5],g1[1],g[1]},
  {{2{g1[14]}},{2{g1[12]}},{2{g1[10]}},
  {2{g1[8]}},{2{g1[6]}},{2{g1[4]}},
  {2{g1[2]}},{2{g1[0]}}},
  p2, g2);

blackbox row3({p2[15],p2[14],p1[14],p[27],p2[11],
  p2[10],p1[10],p[19],p2[7],p2[6],
  p1[6],p[11],p2[3],p2[2],p1[2],p[3]},
  {{4{p2[13]}},{4{p2[9]}},{4{p2[5]}},
  {4{p2[1]}}},
  {g2[15],g2[14],g1[14],g[27],g2[11],
  g2[10],g1[10],g[19],g2[7],g2[6],
  g1[6],g[11],g2[3],g2[2],g1[2],g[3]},
  {{4{g2[13]}},{4{g2[9]}},{4{g2[5]}},
  {4{g2[1]}}},
  p3, g3);
```

VHDL

```
pik_2 <= p1(15) & p(29) & p1(13) & p(25) & p1(11) &
  p(21) & p1(9) & p(17) & p1(7) & p(13) &
  p1(5) & p(9) & p1(3) & p(5) & p1(1) & p(1);

gik_2 <= g1(15) & g(29) & g1(13) & g(25) & g1(11) &
  g(21) & g1(9) & g(17) & g1(7) & g(13) &
  g1(5) & g(9) & g1(3) & g(5) & g1(1) & g(1);

pkj_2 <=
  p1(14) & p1(14) & p1(12) & p1(12) & p1(10) & p1(10) &
  p1(8) & p1(8) & p1(6) & p1(6) & p1(4) & p1(4) &
  p1(2) & p1(2) & p1(0) & p1(0);

gkj_2 <=
  g1(14) & g1(14) & g1(12) & g1(12) & g1(10) & g1(10) &
  g1(8) & g1(8) & g1(6) & g1(6) & g1(4) & g1(4) &
  g1(2) & g1(2) & g1(0) & g1(0);

row2: pgblackblock
  port map(pik_2, gik_2, pkj_2, gkj_2,
    p2, g2);

pik_3 <= p2(15) & p2(14) & p1(14) & p(27) & p2(11) &
  p2(10) & p1(10) & p(19) & p2(7) & p2(6) &
  p1(6) & p(11) & p2(3) & p2(2) & p1(2) & p(3);

gik_3 <= g2(15) & g2(14) & g1(14) & g(27) & g2(11) &
  g2(10) & g1(10) & g(19) & g2(7) & g2(6) &
  g1(6) & g(11) & g2(3) & g2(2) & g1(2) & g(3);

pkj_3 <= p2(13) & p2(13) & p2(13) & p2(13) &
  p2(9) & p2(9) & p2(9) & p2(9) &
  p2(5) & p2(5) & p2(5) & p2(5) &
  p2(1) & p2(1) & p2(1) & p2(1);

gkj_3 <= g2(13) & g2(13) & g2(13) & g2(13) &
  g2(9) & g2(9) & g2(9) & g2(9) &
  g2(5) & g2(5) & g2(5) & g2(5) &
  g2(1) & g2(1) & g2(1) & g2(1);

row3: pgblackblock
  port map(pik_3, gik_3, pkj_3, gkj_3, p3, g3);
```

(continued on next page)

SystemVerilog

```

blackbox row4({p3[15:12],p2[13:12],
               p1[12],p[23],p3[7:4],
               p2[5:4],p1[4],p[7]},
               {{8{p3[11]}},{8{p3[3]}},
               {g3[15:12],g2[13:12],
               g1[12],g[23],g3[7:4],
               g2[5:4],g1[4],g[7]},
               {{8{g3[11]}},{8{g3[3]}},
               p4, g4};

blackbox row5({p4[15:8],p3[11:8],p2[9:8],
               p1[8],p[15]},
               {{16{p4[7]}},
               {g4[15:8],g3[11:8],g2[9:8],
               g1[8],g[15]},
               {{16{g4[7]}},
               p5,g5});

sum row6({g5,g4[7:0],g3[3:0],g2[1:0],g1[0],cin},
         a, b, s);

// generate cout
assign cout = (a[31] & b[31]) |
              (g5[15] & (a[31] | b[31]));

endmodule

```

VHDL

```

pik_4 <= p3(15 downto 12)&p2(13 downto 12)&
         p1(12)&p(23)&p3(7 downto 4)&
         p2(5 downto 4)&p1(4)&p(7);
gik_4 <= g3(15 downto 12)&g2(13 downto 12)&
         g1(12)&g(23)&g3(7 downto 4)&
         g2(5 downto 4)&g1(4)&g(7);
pkj_4 <= p3(11)&p3(11)&p3(11)&p3(11)&
         p3(11)&p3(11)&p3(11)&p3(11)&
         p3(3)&p3(3)&p3(3)&p3(3)&
         p3(3)&p3(3)&p3(3)&p3(3);
gkj_4 <= g3(11)&g3(11)&g3(11)&g3(11)&
         g3(11)&g3(11)&g3(11)&g3(11)&
         g3(3)&g3(3)&g3(3)&g3(3)&
         g3(3)&g3(3)&g3(3)&g3(3);

row4: pgblackblock
      port map(pik_4, gik_4, pkj_4, gkj_4, p4, g4);

pik_5 <= p4(15 downto 8)&p3(11 downto 8)&
         p2(9 downto 8)&p1(8)&p(15);
gik_5 <= g4(15 downto 8)&g3(11 downto 8)&
         g2(9 downto 8)&g1(8)&g(15);
pkj_5 <= p4(7)&p4(7)&p4(7)&p4(7)&
         p4(7)&p4(7)&p4(7)&p4(7)&
         p4(7)&p4(7)&p4(7)&p4(7)&
         p4(7)&p4(7)&p4(7)&p4(7);
gkj_5 <= g4(7)&g4(7)&g4(7)&g4(7)&
         g4(7)&g4(7)&g4(7)&g4(7)&
         g4(7)&g4(7)&g4(7)&g4(7)&
         g4(7)&g4(7)&g4(7)&g4(7);

row5: pgblackblock
      port map(pik_5, gik_5, pkj_5, gkj_5, p5, g5);

g6 <= (g5 & g4(7 downto 0) & g3(3 downto 0) &
       g2(1 downto 0) & g1(0) & cin);

row6: subblock
      port map(g6, a, b, s);

-- generate cout
cout <= (a(31) and b(31)) or
        (g6(31) and (a(31) or b(31)));

end;

```

(continued on next page)

*(continued from previous page)***SystemVerilog**

```

module pandg(input  logic [30:0] a, b,
             output logic [30:0] p, g);

    assign p = a | b;
    assign g = a & b;

endmodule

module blackbox(input  logic [15:0] pleft, pright,
                gleft, gright,
                output logic [15:0] pnext, gnext);

    assign pnext = pleft & pright;
    assign gnext = pleft & gright | gleft;
endmodule

module sum(input  logic [31:0] g, a, b,
           output logic [31:0] s);

    assign s = a ^ b ^ g;

endmodule

```

VHDL

```

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity pgblock is
    port(a, b: in  STD_LOGIC_VECTOR(30 downto 0);
          p, g: out STD_LOGIC_VECTOR(30 downto 0));
end;

architecture synth of pgblock is
begin
    p <= a or b;
    g <= a and b;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity pgblackblock is
    port(pik, gik, pkj, gkj:
          in  STD_LOGIC_VECTOR(15 downto 0);
          pij, gij:
          out STD_LOGIC_VECTOR(15 downto 0));
end;

architecture synth of pgblackblock is
begin
    pij <= pik and pkj;
    gij <= gik or (pik and gkj);
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sumblock is
    port(g, a, b: in  STD_LOGIC_VECTOR(31 downto 0);
          s:      out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture synth of sumblock is
begin
    s <= a xor b xor g;
end;

```


5.41 (c) Using Equation 5.11 to find the delay of the prefix adder:

$$t_{PA} = t_{pg} + \log_2 N(t_{pg_prefix}) + t_{XOR}$$

We find the delays for each block:

$$t_{pg} = 100 \text{ ps}$$

$$t_{pg_prefix} = 200 \text{ ps}$$

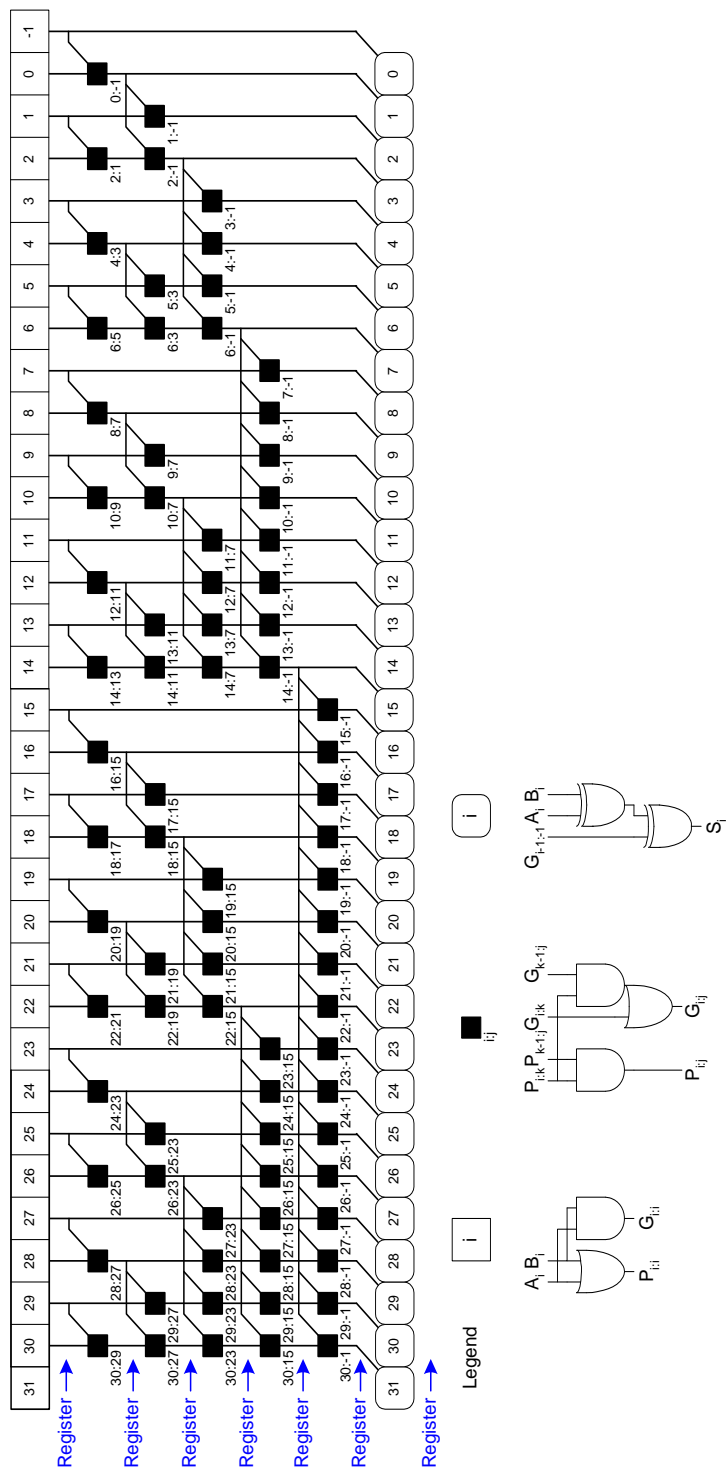
$$t_{XOR} = 100 \text{ ps}$$

Thus,

$$t_{PA} = [100 + 5(200) + 100] \text{ ps} = 1200 \text{ ps} = \mathbf{1.2 \text{ ns}}$$

5.41 (d) To make a pipelined prefix adder, add pipeline registers between each of the rows of the prefix adder. Now each stage will take 200 ps plus the

sequencing overhead, $t_{pq} + t_{\text{setup}} = 80\text{ps}$. Thus each cycle is 280 ps and the design can run at 3.57 GHz.



5.45 (e)

SystemVerilog

```

module prefixaddpipe(input  logic      clk, cin,
                    input  logic [31:0] a, b,
                    output logic [31:0] s, output cout);

    // p and g prefixes for rows 0 - 5
    logic [30:0] p0, p1, p2, p3, p4, p5;
    logic [30:0] g0, g1, g2, g3, g4, g5;
    logic p_1_0, p_1_1, p_1_2, p_1_3, p_1_4, p_1_5,
          g_1_0, g_1_1, g_1_2, g_1_3, g_1_4, g_1_5;

    // pipeline values for a and b
    logic [31:0] a0, a1, a2, a3, a4, a5,
                b0, b1, b2, b3, b4, b5;

    // row 0
    flop #(2) flop0_pg_1(clk, {1'b0,cin}, {p_1_0,g_1_0});
    pandg row0(clk, a[30:0], b[30:0], p0, g0);

    // row 1
    flop #(2) flop1_pg_1(clk, {p_1_0,g_1_0}, {p_1_1,g_1_1});
    flop #(30) flop1_pg(clk,
    {p0[29],p0[27],p0[25],p0[23],p0[21],p0[19],p0[17],p0[15],
     p0[13],p0[11],p0[9],p0[7],p0[5],p0[3],p0[1],
     g0[29],g0[27],g0[25],g0[23],g0[21],g0[19],g0[17],g0[15],
     g0[13],g0[11],g0[9],g0[7],g0[5],g0[3],g0[1]},
    {p1[29],p1[27],p1[25],p1[23],p1[21],p1[19],p1[17],p1[15],
     p1[13],p1[11],p1[9],p1[7],p1[5],p1[3],p1[1],
     g1[29],g1[27],g1[25],g1[23],g1[21],g1[19],g1[17],g1[15],
     g1[13],g1[11],g1[9],g1[7],g1[5],g1[3],g1[1]});

    blackbox row1(clk,
    {p0[30],p0[28],p0[26],p0[24],p0[22],
     p0[20],p0[18],p0[16],p0[14],p0[12],
     p0[10],p0[8],p0[6],p0[4],p0[2],p0[0]},
    {p0[29],p0[27],p0[25],p0[23],p0[21],
     p0[19],p0[17],p0[15],p0[13],p0[11],
     p0[9],p0[7],p0[5],p0[3],p0[1],1'b0},
    {g0[30],g0[28],g0[26],g0[24],g0[22],
     g0[20],g0[18],g0[16],g0[14],g0[12],
     g0[10],g0[8],g0[6],g0[4],g0[2],g0[0]},
    {g0[29],g0[27],g0[25],g0[23],g0[21],
     g0[19],g0[17],g0[15],g0[13],g0[11],
     g0[9],g0[7],g0[5],g0[3],g0[1],g_1_0},
    {p1[30],p1[28],p1[26],p1[24],p1[22],p1[20],
     p1[18],p1[16],p1[14],p1[12],p1[10],p1[8],
     p1[6],p1[4],p1[2],p1[0]},
    {g1[30],g1[28],g1[26],g1[24],g1[22],g1[20],
     g1[18],g1[16],g1[14],g1[12],g1[10],g1[8],
     g1[6],g1[4],g1[2],g1[0]});

    // row 2
    flop #(2) flop2_pg_1(clk, {p_1_1,g_1_1}, {p_1_2,g_1_2});
    flop #(30) flop2_pg(clk,
    {p1[28:27],p1[24:23],p1[20:19],p1[16:15],p1[12:11],

```

```

        p1[8:7],p1[4:3],p1[0],
g1[28:27],g1[24:23],g1[20:19],g1[16:15],g1[12:11],
g1[8:7],g1[4:3],g1[0]],
    {p2[28:27],p2[24:23],p2[20:19],p2[16:15],p2[12:11],
      p2[8:7],p2[4:3],p2[0]},
g2[28:27],g2[24:23],g2[20:19],g2[16:15],g2[12:11],
g2[8:7],g2[4:3],g2[0]]);
    blackbox row2(clk,

{p1[30:29],p1[26:25],p1[22:21],p1[18:17],p1[14:13],p1[10:9],p1[6:5],p1[2:1]
},

    { {2{p1[28]}}, {2{p1[24]}}, {2{p1[20]}}, {2{p1[16]}}, {2{p1[12]}},
      {2{p1[8]}},
      {2{p1[4]}}, {2{p1[0]}} },

{g1[30:29],g1[26:25],g1[22:21],g1[18:17],g1[14:13],g1[10:9],g1[6:5],g1[2:1]
},

    { {2{g1[28]}}, {2{g1[24]}}, {2{g1[20]}}, {2{g1[16]}}, {2{g1[12]}},
      {2{g1[8]}},
      {2{g1[4]}}, {2{g1[0]}} },

{p2[30:29],p2[26:25],p2[22:21],p2[18:17],p2[14:13],p2[10:9],p2[6:5],p2[2:1]
},

{g2[30:29],g2[26:25],g2[22:21],g2[18:17],g2[14:13],g2[10:9],g2[6:5],g2[2:1]
} );

// row 3
flop #(2) flop3_pg_1(clk, {p_1_2,g_1_2}, {p_1_3,g_1_3});
flop #(30) flop3_pg(clk, {p2[26:23],p2[18:15],p2[10:7],p2[2:0],
g2[26:23],g2[18:15],g2[10:7],g2[2:0]},
{p3[26:23],p3[18:15],p3[10:7],p3[2:0]},
g3[26:23],g3[18:15],g3[10:7],g3[2:0]]);
    blackbox row3(clk,
        {p2[30:27],p2[22:19],p2[14:11],p2[6:3]},
    { {4{p2[26]}}, {4{p2[18]}}, {4{p2[10]}}, {4{p2[2]}} },
    {g2[30:27],g2[22:19],g2[14:11],g2[6:3]},
    { {4{g2[26]}}, {4{g2[18]}}, {4{g2[10]}}, {4{g2[2]}} },
    {p3[30:27],p3[22:19],p3[14:11],p3[6:3]},
    {g3[30:27],g3[22:19],g3[14:11],g3[6:3]});

// row 4
flop #(2) flop4_pg_1(clk, {p_1_3,g_1_3}, {p_1_4,g_1_4});
flop #(30) flop4_pg(clk, {p3[22:15],p3[6:0]},
g3[22:15],g3[6:0]],
        {p4[22:15],p4[6:0]},
g4[22:15],g4[6:0]]);

    blackbox row4(clk,
        {p3[30:23],p3[14:7]},
    { {8{p3[22]}}, {8{p3[6]}} },
        {g3[30:23],g3[14:7]},
    { {8{g3[22]}}, {8{g3[6]}} },
    {p4[30:23],p4[14:7]},
    {g4[30:23],g4[14:7]});

// row 5
flop #(2) flop5_pg_1(clk, {p_1_4,g_1_4}, {p_1_5,g_1_5});
flop #(30) flop5_pg(clk, {p4[14:0],g4[14:0]},
        {p5[14:0],g5[14:0]});

```

```

        blackbox row5(clk,
                      p4[30:15],
                      {16{p4[14]}},
                      g4[30:15],
                      {16{g4[14]}},
                      p5[30:15], g5[30:15]);

        // pipeline registers for a and b
        flop #(64) flop0_ab(clk, {a,b}, {a0,b0});
        flop #(64) flop1_ab(clk, {a0,b0}, {a1,b1});
        flop #(64) flop2_ab(clk, {a1,b1}, {a2,b2});
        flop #(64) flop3_ab(clk, {a2,b2}, {a3,b3});
        flop #(64) flop4_ab(clk, {a3,b3}, {a4,b4});
        flop #(64) flop5_ab(clk, {a4,b4}, {a5,b5});

        sum row6(clk, {g5,g_1_5}, a5, b5, s);
        // generate cout
        assign cout = (a5[31] & b5[31]) | (g5[30] & (a5[31] | b5[31]));
    endmodule

    // submodules
    module pandg(input  logic      clk,
                 input  logic [30:0] a, b,
                 output logic [30:0] p, g);

        always_ff @(posedge clk)
        begin
            p <= a | b;
            g <= a & b;
        end

    endmodule

    module blackbox(input  logic clk,
                    input  logic [15:0] pleft, pright, gleft, gright,
                    output logic [15:0] pnext, gnext);

        always_ff @(posedge clk)
        begin
            pnext <= pleft & pright;
            gnext <= pleft & gright | gleft;
        end

    endmodule

    module sum(input  logic      clk,
               input  logic [31:0] g, a, b,
               output logic [31:0] s);

        always_ff @(posedge clk)
            s <= a ^ b ^ g;
    endmodule

    module flop
        #(parameter width = 8)
        (input  logic      clk,
         input  logic [width-1:0] d,
         output logic [width-1:0] q);

        always_ff @(posedge clk)
            q <= d;
    endmodule

```

5.45 (e)

VHDL

```

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity prefixaddpipe is
  port (clk: in  STD_LOGIC;
        a, b: in  STD_LOGIC_VECTOR(31 downto 0);
        cin: in  STD_LOGIC;
        s:   out STD_LOGIC_VECTOR(31 downto 0);
        cout: out STD_LOGIC);
end;

architecture synth of prefixaddpipe is
  component pgblock
    port (clk: in  STD_LOGIC;
          a, b: in  STD_LOGIC_VECTOR(30 downto 0);
          p, g: out STD_LOGIC_VECTOR(30 downto 0));
  end component;
  component sumblock is
    port (clk: in  STD_LOGIC;
          a, b, g: in  STD_LOGIC_VECTOR(31 downto 0);
          s:   out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component flop is generic(width: integer);
    port (clk: in  STD_LOGIC;
          d:   in  STD_LOGIC_VECTOR(width-1 downto 0);
          q:   out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component flop1 is
    port (clk: in  STD_LOGIC;
          d:   in  STD_LOGIC;
          q:   out STD_LOGIC);
  end component;
  component row1 is
    port (clk: in  STD_LOGIC;
          p0, g0: in  STD_LOGIC_VECTOR(30 downto 0);
          p1_0, g1_0: in STD_LOGIC;
          p1, g1: out STD_LOGIC_VECTOR(30 downto 0));
  end component;
  component row2 is
    port (clk: in  STD_LOGIC;
          p1, g1: in  STD_LOGIC_VECTOR(30 downto 0);
          p2, g2: out STD_LOGIC_VECTOR(30 downto 0));
  end component;
  component row3 is
    port (clk: in  STD_LOGIC;
          p2, g2: in  STD_LOGIC_VECTOR(30 downto 0);
          p3, g3: out STD_LOGIC_VECTOR(30 downto 0));
  end component;
  component row4 is
    port (clk: in  STD_LOGIC;
          p3, g3: in  STD_LOGIC_VECTOR(30 downto 0);
          p4, g4: out STD_LOGIC_VECTOR(30 downto 0));
  end component;
  component row5 is
    port (clk: in  STD_LOGIC;
          p4, g4: in  STD_LOGIC_VECTOR(30 downto 0);
          p5, g5: out STD_LOGIC_VECTOR(30 downto 0));
  end component;

```

```

-- p and g prefixes for rows 0 - 5
signal p0, p1, p2, p3, p4, p5: STD_LOGIC_VECTOR(30 downto 0);
signal g0, g1, g2, g3, g4, g5: STD_LOGIC_VECTOR(30 downto 0);

-- p and g prefixes for column -1, rows 0 - 5
signal p_1_0, p_1_1, p_1_2, p_1_3, p_1_4, p_1_5,
       g_1_0, g_1_1, g_1_2, g_1_3, g_1_4, g_1_5: STD_LOGIC;

-- pipeline values for a and b
signal a0, a1, a2, a3, a4, a5,
       b0, b1, b2, b3, b4, b5: STD_LOGIC_VECTOR(31 downto 0);

-- final generate signal
signal g5_all: STD_LOGIC_VECTOR(31 downto 0);

begin

-- p and g calculations
row0_reg: pgblock port map(clk, a(30 downto 0), b(30 downto 0), p0, g0);
row1_reg: row1 port map(clk, p0, g0, p_1_0, g_1_0, p1, g1);
row2_reg: row2 port map(clk, p1, g1, p_1_1, g_1_1, p2, g2);
row3_reg: row3 port map(clk, p2, g2, p_1_2, g_1_2, p3, g3);
row4_reg: row4 port map(clk, p3, g3, p_1_3, g_1_3, p4, g4);
row5_reg: row5 port map(clk, p4, g4, p_1_4, g_1_4, p5, g5);

-- pipeline registers for a and b
flop0_a: flop generic map(32) port map (clk, a, a0);
flop0_b: flop generic map(32) port map (clk, b, b0);
flop1_a: flop generic map(32) port map (clk, a0, a1);
flop1_b: flop generic map(32) port map (clk, b0, b1);
flop2_a: flop generic map(32) port map (clk, a1, a2);
flop2_b: flop generic map(32) port map (clk, b1, b2);
flop3_a: flop generic map(32) port map (clk, a2, a3);
flop3_b: flop generic map(32) port map (clk, b2, b3);
flop4_a: flop generic map(32) port map (clk, a3, a4);
flop4_b: flop generic map(32) port map (clk, b3, b4);
flop5_a: flop generic map(32) port map (clk, a4, a5);
flop5_b: flop generic map(32) port map (clk, b4, b5);

-- pipeline p and g for column -1
p_1_0 <= '0'; flop1_g0: flop1 port map (clk, cin, g_1_0);
flop1_p1: flop1 port map (clk, p_1_0, p_1_1);
flop1_g1: flop1 port map (clk, g_1_0, g_1_1);
flop1_p2: flop1 port map (clk, p_1_1, p_1_2);
flop1_g2: flop1 port map (clk, g_1_1, g_1_2);
flop1_p3: flop1 port map (clk, p_1_2, p_1_3); flop1_g3:
flop1 port map (clk, g_1_2, g_1_3);
flop1_p4: flop1 port map (clk, p_1_3, p_1_4);
flop1_g4: flop1 port map (clk, g_1_3, g_1_4);
flop1_p5: flop1 port map (clk, p_1_4, p_1_5);
flop1_g5: flop1 port map (clk, g_1_4, g_1_5);

-- generate sum and cout
g5_all <= (g5&g_1_5);
row6: sumblock port map(clk, g5_all, a5, b5, s);

-- generate cout
cout <= (a5(31) and b5(31)) or (g5(30) and (a5(31) or b5(31)));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity pgblock is
  port(clk: in  STD_LOGIC;

```



```

        a, b: in  STD_LOGIC_VECTOR(30 downto 0);
        p, g: out STD_LOGIC_VECTOR(30 downto 0));
end;

architecture synth of pgblock is
begin
    process(clk) begin
        if rising_edge(clk) then
            p <= a or b;
            g <= a and b;
        end if;
    end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity blackbox is
    port(clk: in  STD_LOGIC;
          pik, pkj, gik, gkj:
              in  STD_LOGIC_VECTOR(15 downto 0);
          pij, gij:
              out STD_LOGIC_VECTOR(15 downto 0));
end;

architecture synth of blackbox is
begin
    process(clk) begin
        if rising_edge(clk) then
            pij <= pik and pkj;
            gij <= gik or (pik and gkj);
        end if;
    end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sumblock is
    port(clk: in  STD_LOGIC;
          g, a, b: in  STD_LOGIC_VECTOR(31 downto 0);
          s:      out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture synth of sumblock is
begin
    process(clk) begin
        if rising_edge(clk) then
            s <= a xor b xor g;
        end if;
    end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all; use IEEE.STD_LOGIC_ARITH.all;
entity flop is -- parameterizable flip flop
    generic(width: integer);
    port(clk:      in  STD_LOGIC;
          d:        in  STD_LOGIC_VECTOR(width-1 downto 0);
          q:        out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture synth of flop is
begin
    process(clk) begin
        if rising_edge(clk) then
            q <= d;
        end if;
    end process;
end;

```

```

        end if;
    end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all; use IEEE.STD_LOGIC_ARITH.all;
entity flop1 is -- 1-bit flip flop
    port(clk:      in  STD_LOGIC;
          d:        in  STD_LOGIC;
          q:        out STD_LOGIC);
end;

architecture synth of flop1 is
begin
    process(clk) begin
        if rising_edge(clk) then
            q <= d;
        end if;
    end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity row1 is
    port(clk:      in  STD_LOGIC;
          p0, g0: in  STD_LOGIC_VECTOR(30 downto 0);
          p_1_0, g_1_0: in STD_LOGIC;
          p1, g1: out STD_LOGIC_VECTOR(30 downto 0));
end;

architecture synth of row1 is
    component blackbox is
        port (clk:      in  STD_LOGIC;
              pik, pkj: in  STD_LOGIC_VECTOR(15 downto 0);
              gik, gkj: in  STD_LOGIC_VECTOR(15 downto 0);
              pij:      out STD_LOGIC_VECTOR(15 downto 0);
              gij:      out STD_LOGIC_VECTOR(15 downto 0));
    end component;
    component flop is generic(width: integer);
        port(clk: in  STD_LOGIC;
              d:   in  STD_LOGIC_VECTOR(width-1 downto 0);
              q:   out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;

    -- internal signals for calculating p, g
    signal pik_0, gik_0, pkj_0, gkj_0,
           pij_0, gij_0: STD_LOGIC_VECTOR(15 downto 0);

    -- internal signals for pipeline registers
    signal pg0_in, pg1_out: STD_LOGIC_VECTOR(29 downto 0);

begin
    pg0_in <= (p0(29) & p0(27) & p0(25) & p0(23) & p0(21) & p0(19) & p0(17) & p0(15) &
               p0(13) & p0(11) & p0(9) & p0(7) & p0(5) & p0(3) & p0(1) &
               g0(29) & g0(27) & g0(25) & g0(23) & g0(21) & g0(19) & g0(17) & g0(15) &
               g0(13) & g0(11) & g0(9) & g0(7) & g0(5) & g0(3) & g0(1));
    flop1_pg: flop generic map(30) port map (clk, pg0_in, pg1_out);

    p1(29) <= pg1_out(29); p1(27) <= pg1_out(28); p1(25) <= pg1_out(27);
    p1(23) <= pg1_out(26);
    p1(21) <= pg1_out(25); p1(19) <= pg1_out(24); p1(17) <= pg1_out(23);
    p1(15) <= pg1_out(22); p1(13) <= pg1_out(21); p1(11) <= pg1_out(20);
    p1(9) <= pg1_out(19); p1(7) <= pg1_out(18); p1(5) <= pg1_out(17);
    p1(3) <= pg1_out(16); p1(1) <= pg1_out(15);
    g1(29) <= pg1_out(14); g1(27) <= pg1_out(13); g1(25) <= pg1_out(12);
    g1(23) <= pg1_out(11); g1(21) <= pg1_out(10); g1(19) <= pg1_out(9);
    g1(17) <= pg1_out(8); g1(15) <= pg1_out(7); g1(13) <= pg1_out(6);

```

```

g1(11) <= pgl_out(5); g1(9) <= pgl_out(4); g1(7) <= pgl_out(3);
g1(5) <= pgl_out(2); g1(3) <= pgl_out(1); g1(1) <= pgl_out(0);

-- pg calculations
pik_0 <= (p0(30)&p0(28)&p0(26)&p0(24)&p0(22)&p0(20)&p0(18)&p0(16)&
p0(14)&p0(12)&p0(10)&p0(8)&p0(6)&p0(4)&p0(2)&p0(0));
gik_0 <= (g0(30)&g0(28)&g0(26)&g0(24)&g0(22)&g0(20)&g0(18)&g0(16)&
g0(14)&g0(12)&g0(10)&g0(8)&g0(6)&g0(4)&g0(2)&g0(0));
pkj_0 <= (p0(29)&p0(27)&p0(25)&p0(23)&p0(21)&p0(19)&p0(17)&p0(15)&
p0(13)&p0(11)&p0(9)&p0(7)&p0(5)&p0(3)&p0(1)&p_1_0);
gkj_0 <= (g0(29)&g0(27)&g0(25)&g0(23)&g0(21)&g0(19)&g0(17)&g0(15)&
g0(13)&g0(11)&g0(9)&g0(7)&g0(5)&g0(3)&g0(1)&g_1_0);

row1: blackbox port map(clk, pik_0, pkj_0, gik_0, gkj_0, pij_0, gij_0);

p1(30) <= pij_0(15); p1(28) <= pij_0(14); p1(26) <= pij_0(13);
p1(24) <= pij_0(12); p1(22) <= pij_0(11); p1(20) <= pij_0(10);
p1(18) <= pij_0(9); p1(16) <= pij_0(8); p1(14) <= pij_0(7);
p1(12) <= pij_0(6); p1(10) <= pij_0(5); p1(8) <= pij_0(4);
p1(6) <= pij_0(3); p1(4) <= pij_0(2); p1(2) <= pij_0(1); p1(0) <= pij_0(0);

g1(30) <= gij_0(15); g1(28) <= gij_0(14); g1(26) <= gij_0(13);
g1(24) <= gij_0(12); g1(22) <= gij_0(11); g1(20) <= gij_0(10);
g1(18) <= gij_0(9); g1(16) <= gij_0(8); g1(14) <= gij_0(7);
g1(12) <= gij_0(6); g1(10) <= gij_0(5); g1(8) <= gij_0(4);
g1(6) <= gij_0(3); g1(4) <= gij_0(2); g1(2) <= gij_0(1); g1(0) <= gij_0(0);
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity row2 is
  port(clk: in STD_LOGIC;
        p1, g1: in STD_LOGIC_VECTOR(30 downto 0);
        p2, g2: out STD_LOGIC_VECTOR(30 downto 0));
end;

architecture synth of row2 is
  component blackbox is
    port (clk: in STD_LOGIC;
          pik, pkj: in STD_LOGIC_VECTOR(15 downto 0);
          gik, gkj: in STD_LOGIC_VECTOR(15 downto 0);
          pij: out STD_LOGIC_VECTOR(15 downto 0);
          gij: out STD_LOGIC_VECTOR(15 downto 0));
  end component;
  component flop is generic(width: integer);
    port(clk: in STD_LOGIC;
          d: in STD_LOGIC_VECTOR(width-1 downto 0);
          q: out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;

  -- internal signals for calculating p, g
  signal pik_1, gik_1, pkj_1, gkj_1,
        pij_1, gij_1: STD_LOGIC_VECTOR(15 downto 0);

  -- internal signals for pipeline registers
  signal pgl_in, pg2_out: STD_LOGIC_VECTOR(29 downto 0);

begin
  pgl_in <= (p1(28 downto 27)&p1(24 downto 23)&p1(20 downto 19)&
p1(16 downto 15)&
p1(12 downto 11)&p1(8 downto 7)&p1(4 downto 3)&p1(0)&
g1(28 downto 27)&g1(24 downto 23)&g1(20 downto 19)&
g1(16 downto 15)&
g1(12 downto 11)&g1(8 downto 7)&g1(4 downto 3)&g1(0));
  flop2_pg: flop generic map(30) port map (clk, pgl_in, pg2_out);

```

```

p2(28 downto 27) <= pg2_out(29 downto 28);
p2(24 downto 23) <= pg2_out(27 downto 26);
p2(20 downto 19) <= pg2_out(25 downto 24);
p2(16 downto 15) <= pg2_out(23 downto 22);
p2(12 downto 11) <= pg2_out(21 downto 20);
p2(8 downto 7) <= pg2_out(19 downto 18);
p2(4 downto 3) <= pg2_out(17 downto 16);
p2(0) <= pg2_out(15);
g2(28 downto 27) <= pg2_out(14 downto 13);
g2(24 downto 23) <= pg2_out(12 downto 11);
g2(20 downto 19) <= pg2_out(10 downto 9);
g2(16 downto 15) <= pg2_out(8 downto 7);
g2(12 downto 11) <= pg2_out(6 downto 5);
g2(8 downto 7) <= pg2_out(4 downto 3);
g2(4 downto 3) <= pg2_out(2 downto 1); g2(0) <= pg2_out(0);

-- pg calculations
pik_1 <= (p1(30 downto 29)&p1(26 downto 25)&p1(22 downto 21)&
p1(18 downto 17)&p1(14 downto 13)&p1(10 downto 9)&
p1(6 downto 5)&p1(2 downto 1));
gik_1 <= (g1(30 downto 29)&g1(26 downto 25)&g1(22 downto 21)&
g1(18 downto 17)&g1(14 downto 13)&g1(10 downto 9)&
g1(6 downto 5)&g1(2 downto 1));
pkj_1 <= (p1(28)&p1(28)&p1(24)&p1(24)&p1(20)&p1(20)&p1(16)&p1(16)&
p1(12)&p1(12)&p1(8)&p1(8)&p1(4)&p1(4)&p1(0)&p1(0));
gkj_1 <= (g1(28)&g1(28)&g1(24)&g1(24)&g1(20)&g1(20)&g1(16)&g1(16)&
g1(12)&g1(12)&g1(8)&g1(8)&g1(4)&g1(4)&g1(0)&g1(0));

row2: blackbox
port map(clk, pik_1, pkj_1, gik_1, gkj_1, pij_1, gij_1);

p2(30 downto 29) <= pij_1(15 downto 14);
p2(26 downto 25) <= pij_1(13 downto 12);
p2(22 downto 21) <= pij_1(11 downto 10);
p2(18 downto 17) <= pij_1(9 downto 8);
p2(14 downto 13) <= pij_1(7 downto 6); p2(10 downto 9) <= pij_1(5 downto 4);
p2(6 downto 5) <= pij_1(3 downto 2); p2(2 downto 1) <= pij_1(1 downto 0);

g2(30 downto 29) <= gij_1(15 downto 14);
g2(26 downto 25) <= gij_1(13 downto 12);
g2(22 downto 21) <= gij_1(11 downto 10);
g2(18 downto 17) <= gij_1(9 downto 8);
g2(14 downto 13) <= gij_1(7 downto 6); g2(10 downto 9) <= gij_1(5 downto 4);
g2(6 downto 5) <= gij_1(3 downto 2); g2(2 downto 1) <= gij_1(1 downto 0);

end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity row3 is
port(clk: in STD_LOGIC;
p2, g2: in STD_LOGIC_VECTOR(30 downto 0);
p3, g3: out STD_LOGIC_VECTOR(30 downto 0));
end;

architecture synth of row3 is
component blackbox is
port (clk: in STD_LOGIC;
pik, pkj: in STD_LOGIC_VECTOR(15 downto 0);
gik, gkj: in STD_LOGIC_VECTOR(15 downto 0);
pij: out STD_LOGIC_VECTOR(15 downto 0);
gij: out STD_LOGIC_VECTOR(15 downto 0));
end component;
component flop is generic(width: integer);
port(clk: in STD_LOGIC;
d: in STD_LOGIC_VECTOR(width-1 downto 0);

```

```

        q:   out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;

    -- internal signals for calculating p, g
    signal pik_2, gik_2, pkj_2, gkj_2,
           pij_2, gij_2: STD_LOGIC_VECTOR(15 downto 0);

    -- internal signals for pipeline registers
    signal pg2_in, pg3_out: STD_LOGIC_VECTOR(29 downto 0);

begin
    pg2_in <= (p2(26 downto 23)&p2(18 downto 15)&p2(10 downto 7)&
              p2(2 downto 0)&
              g2(26 downto 23)&g2(18 downto 15)&g2(10 downto 7)&g2(2 downto 0));
    flop3_pg: flop_generic map(30) port map (clk, pg2_in, pg3_out);
    p3(26 downto 23) <= pg3_out(29 downto 26);
    p3(18 downto 15) <= pg3_out(25 downto 22);
    p3(10 downto 7)  <= pg3_out(21 downto 18);
    p3(2 downto 0)   <= pg3_out(17 downto 15);
    g3(26 downto 23) <= pg3_out(14 downto 11);
    g3(18 downto 15) <= pg3_out(10 downto 7);
    g3(10 downto 7)  <= pg3_out(6 downto 3);
    g3(2 downto 0)   <= pg3_out(2 downto 0);

    -- pg calculations
    pik_2 <= (p2(30 downto 27)&p2(22 downto 19)&
              p2(14 downto 11)&p2(6 downto 3));
    gik_2 <= (g2(30 downto 27)&g2(22 downto 19)&
              g2(14 downto 11)&g2(6 downto 3));
    pkj_2 <= (p2(26)&p2(26)&p2(26)&p2(26)&
              p2(18)&p2(18)&p2(18)&p2(18)&
              p2(10)&p2(10)&p2(10)&p2(10)&
              p2(2)&p2(2)&p2(2)&p2(2));
    gkj_2 <= (g2(26)&g2(26)&g2(26)&g2(26)&
              g2(18)&g2(18)&g2(18)&g2(18)&
              g2(10)&g2(10)&g2(10)&g2(10)&
              g2(2)&g2(2)&g2(2)&g2(2));

    row3: blackbox
        port map(clk, pik_2, pkj_2, gik_2, gkj_2, pij_2, gij_2);

    p3(30 downto 27) <= pij_2(15 downto 12);
    p3(22 downto 19) <= pij_2(11 downto 8);
    p3(14 downto 11) <= pij_2(7 downto 4); p3(6 downto 3) <= pij_2(3 downto 0);
    g3(30 downto 27) <= gij_2(15 downto 12);
    g3(22 downto 19) <= gij_2(11 downto 8);
    g3(14 downto 11) <= gij_2(7 downto 4); g3(6 downto 3) <= gij_2(3 downto 0);

end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity row4 is
    port(clk:   in STD_LOGIC;
          p3, g3: in STD_LOGIC_VECTOR(30 downto 0);
          p4, g4: out STD_LOGIC_VECTOR(30 downto 0));
end;

architecture synth of row4 is
    component blackbox is
        port (clk:   in STD_LOGIC;
              pik, pkj: in STD_LOGIC_VECTOR(15 downto 0);
              gik, gkj: in STD_LOGIC_VECTOR(15 downto 0);
              pij:   out STD_LOGIC_VECTOR(15 downto 0);
              gij:   out STD_LOGIC_VECTOR(15 downto 0));
    end component;

```

```

component flop is generic(width: integer);
  port(clk: in  STD_LOGIC;
        d:  in  STD_LOGIC_VECTOR(width-1 downto 0);
        q:  out STD_LOGIC_VECTOR(width-1 downto 0));
end component;

-- internal signals for calculating p, g
signal pik_3, gik_3, pkj_3, gkj_3,
       pij_3, gij_3: STD_LOGIC_VECTOR(15 downto 0);

-- internal signals for pipeline registers
signal pg3_in, pg4_out: STD_LOGIC_VECTOR(29 downto 0);

begin
pg3_in <= (p3(22 downto 15)&p3(6 downto 0)&g3(22 downto 15)&g3(6 downto 0));
flop4_pg: flop generic map(30) port map (clk, pg3_in, pg4_out);
p4(22 downto 15) <= pg4_out(29 downto 22);
p4(6 downto 0) <= pg4_out(21 downto 15);
g4(22 downto 15) <= pg4_out(14 downto 7);
g4(6 downto 0) <= pg4_out(6 downto 0);

-- pg calculations
pik_3 <= (p3(30 downto 23)&p3(14 downto 7));
gik_3 <= (g3(30 downto 23)&g3(14 downto 7));
pkj_3 <= (p3(22)&p3(22)&p3(22)&p3(22)&p3(22)&p3(22)&p3(22)&p3(22)&
          p3(6)&p3(6)&p3(6)&p3(6)&p3(6)&p3(6)&p3(6)&p3(6));
gkj_3 <= (g3(22)&g3(22)&g3(22)&g3(22)&g3(22)&g3(22)&g3(22)&g3(22)&g3(22)&
          g3(6)&g3(6)&g3(6)&g3(6)&g3(6)&g3(6)&g3(6)&g3(6));

row4: blackbox
  port map(clk, pik_3, pkj_3, gik_3, gkj_3, pij_3, gij_3);

p4(30 downto 23) <= pij_3(15 downto 8);
p4(14 downto 7) <= pij_3(7 downto 0);
g4(30 downto 23) <= gij_3(15 downto 8);
g4(14 downto 7) <= gij_3(7 downto 0);

end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity row5 is
  port(clk: in  STD_LOGIC;
        p4, g4: in  STD_LOGIC_VECTOR(30 downto 0);
        p5, g5: out STD_LOGIC_VECTOR(30 downto 0));
end;

architecture synth of row5 is
  component blackbox is
    port (clk: in  STD_LOGIC;
          pik, pkj: in  STD_LOGIC_VECTOR(15 downto 0);
          gik, gkj: in  STD_LOGIC_VECTOR(15 downto 0);
          pij: out STD_LOGIC_VECTOR(15 downto 0);
          gij: out STD_LOGIC_VECTOR(15 downto 0));
  end component;
  component flop is generic(width: integer);
    port(clk: in  STD_LOGIC;
          d:  in  STD_LOGIC_VECTOR(width-1 downto 0);
          q:  out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;

  -- internal signals for calculating p, g
  signal pik_4, gik_4, pkj_4, gkj_4,
         pij_4, gij_4: STD_LOGIC_VECTOR(15 downto 0);

  -- internal signals for pipeline registers
  signal pg4_in, pg5_out: STD_LOGIC_VECTOR(29 downto 0);

```

```

begin

    pg4_in <= (p4(14 downto 0) & g4(14 downto 0));
    flop4_pg: flop generic map(30) port map (clk, pg4_in, pg5_out);
    p5(14 downto 0) <= pg5_out(29 downto 15); g5(14 downto 0) <= pg5_out(14
downto 0);

    -- pg calculations
    pik_4 <= p4(30 downto 15);
    gik_4 <= g4(30 downto 15);
    pkj_4 <= p4(14) & p4(14) & p4(14) & p4(14) &
    p4(14) & p4(14) & p4(14) & p4(14) &
    p4(14) & p4(14) & p4(14) & p4(14) &
    p4(14) & p4(14) & p4(14) & p4(14);
    gkj_4 <= g4(14) & g4(14) & g4(14) & g4(14) &
    g4(14) & g4(14) & g4(14) & g4(14) &
    g4(14) & g4(14) & g4(14) & g4(14) &
    g4(14) & g4(14) & g4(14) & g4(14);

    row5: blackbox
        port map (clk, pik_4, gik_4, pkj_4, gkj_4, pij_4, gij_4);
        p5(30 downto 15) <= pij_4; g5(30 downto 15) <= gij_4;

end;

```

Exercise 5.46

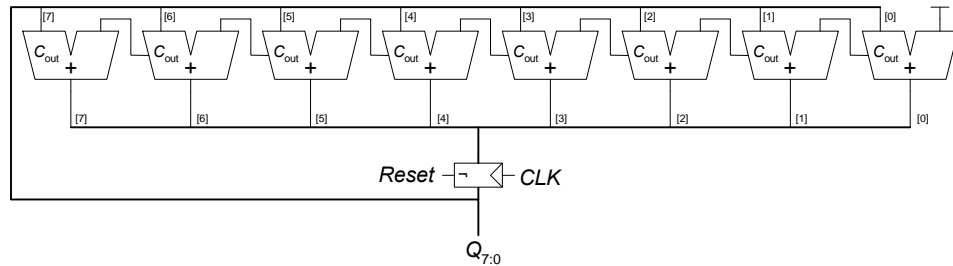


FIGURE 5.13 Incrementer built using half adders

Exercise 5.47

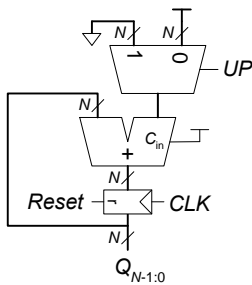


FIGURE 5.14 Up/Down counter

Exercise 5.48

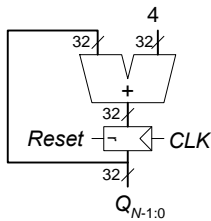


FIGURE 5.15 32-bit counter that increments by 4 on each clock edge

Exercise 5.49

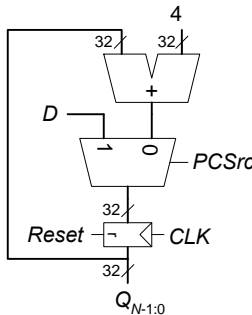


FIGURE 5.16 32-bit counter that increments by 4 or loads a new value, D

Exercise 5.50

(a)

0000

1000

1100

1110

1111

0111

0011

0001

(repeat)

(b)

$2N$. 1's shift into the left-most bit for N cycles, then 0's shift into the left bit for N cycles. Then the process repeats.

5.50 (c)

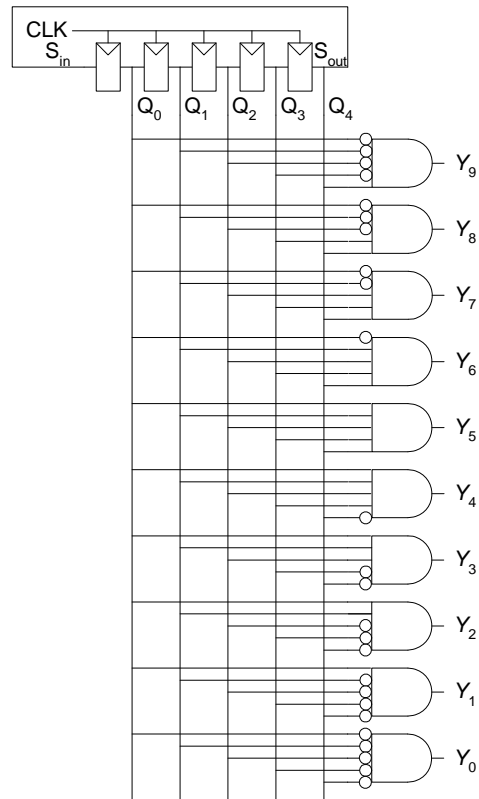


FIGURE 5.17 10-bit decimal counter using a 5-bit Johnson counter

(d) The counter uses less hardware and could be faster because it has a short critical path (a single inverter delay).

Exercise 5.51

SystemVerilog

```
module scanflop4(input  logic      clk, test, sin,
                 input  logic [3:0] d,
                 output logic [3:0] q,
                 output logic      sout);

    always_ff @(posedge clk)
        if (test)
            q <= d;
        else
            q <= {q[2:0], sin};

    assign sout = q[3];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity scanflop4 is
    port (clk, test, sin: in  STD_LOGIC;
          d: in  STD_LOGIC_VECTOR(3 downto 0);
          q: inout STD_LOGIC_VECTOR(3 downto 0);
          sout: out STD_LOGIC);
end;

architecture synth of scanflop4 is
begin
    process(clk, test) begin
        if rising_edge(clk) then
            if test then
                q <= d;
            else
                q <= q(2 downto 0) & sin;
            end if;
        end if;
    end process;

    sout <= q(3);
end;
```

Exercise 5.52

(a)

value <i>a</i> 1:0	encoding <i>y</i> 4:0
00	00001
01	01010
10	10100
11	11111

TABLE 5.2 Possible encodings

The first two pairs of bits in the bit encoding repeat the value. The last bit is the XNOR of the two input values.

5.52 (b) This circuit can be built using a 16×2 -bit memory array, with the contents given in Table 5.3.

address $a_{4:0}$	data $d_{1:0}$
00001	00
00000	00
00011	00
00101	00
01001	00
10001	00
01010	01
01011	01
01000	01
01110	01
00010	01
11010	01
10100	10
10101	10
10110	10
10000	10
11100	10
00100	10
11111	11
11110	11
11101	11
11011	11
10111	11

TABLE 5.3 Memory array values for Exercise 5.48

address $a_{4:0}$	data $d_{1:0}$
01111	11
others	XX

TABLE 5.3 Memory array values for Exercise 5.48

5.48 (c) The implementation shown in part (b) allows the encoding to change easily. Each memory address corresponds to an encoding, so simply store different data values at each memory address to change the encoding.

Exercise 5.53

<http://www.intel.com/design/flash/articles/what.htm>

Flash memory is a nonvolatile memory because it retains its contents after power is turned off. Flash memory allows the user to electrically program and erase information. Flash memory uses memory cells similar to an EEPROM, but with a much thinner, precisely grown oxide between a floating gate and the substrate (see Figure 5.18).

Flash programming occurs when electrons are placed on the floating gate. This is done by forcing a large voltage (usually 10 to 12 volts) on the control gate. Electrons quantum-mechanically tunnel from the source through the thin oxide onto the control gate. Because the floating gate is completely insulated by oxide, the charges are trapped on the floating gate during normal operation. If electrons are stored on the floating gate, it blocks the effect of the control gate. The electrons on the floating gate can be removed by reversing the procedure, i.e., by placing a large negative voltage on the control gate.

The default state of a flash bitcell (when there are no electrons on the floating gate) is ON, because the channel will conduct when the wordline is HIGH. After the bitcell is programmed (i.e., when there are electrons on the floating gate), the state of the bitcell is OFF, because the floating gate blocks the effect of the control gate. Flash memory is a key element in thumb drives, cell phones, digital cameras, Blackberries, and other low-power devices that must retain their memory when turned off.

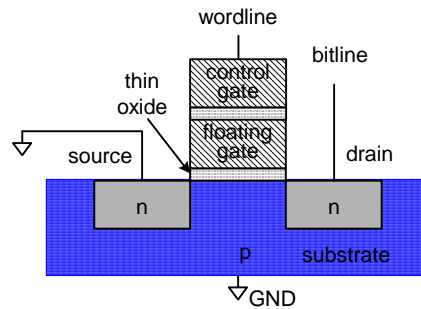


FIGURE 5.18 Flash EEPROM

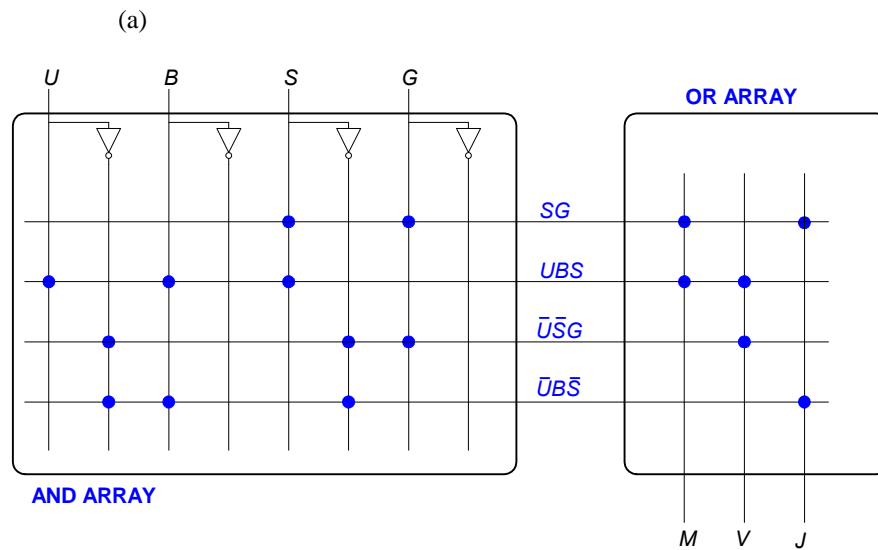
Exercise 5.54

FIGURE 5.19 4 x 4 x 3 PLA implementing Exercise 5.44

5.54 (b)

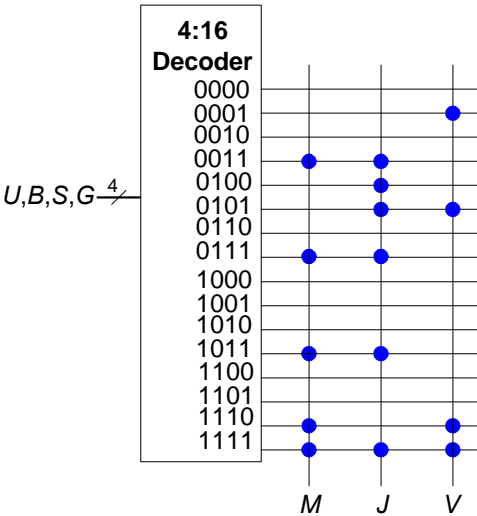


FIGURE 5.20 16 x 3 ROM implementation of Exercise 5.44

(c)

SystemVerilog

```
module ex5_44c(input logic u, b, s, g,
               output logic m, j, v);

    assign m = s&g | u&b&s;
    assign j = ~u&b&~s | s&g;
    assign v = u&b&s | ~u&~s&g;
endmodule
```

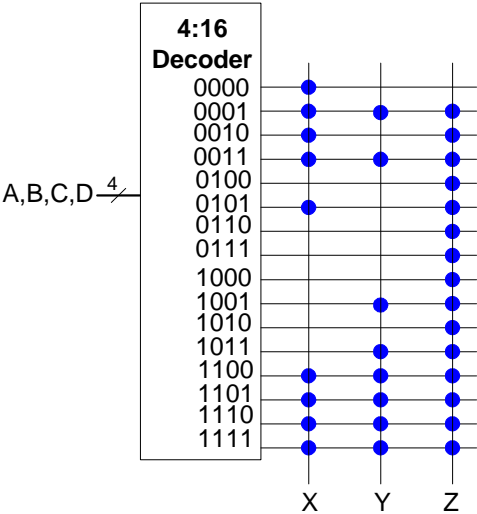
VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity ex5_44c is
    port(u, b, s, g: in STD_LOGIC;
          m, j, v: out STD_LOGIC);
end;

architecture synth of ex5_44c is
begin
    m <= (s and g) or (u and b and s);
    j <= ((not u) and b and (not s)) or (s and g);
    v <= (u and b and s) or ((not u) and (not s) and g);
end;
```

Exercise 5.55



Exercise 5.56

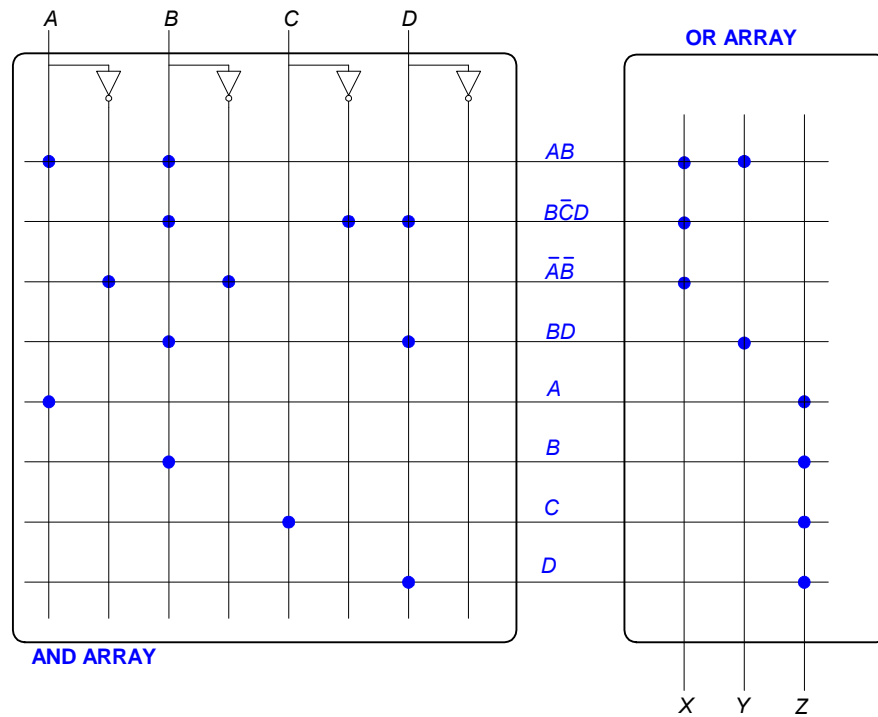


FIGURE 5.21 4 x 8 x 3 PLA for Exercise 5.52

Exercise 5.57

- (a) Number of inputs = $2 \times 16 + 1 = 33$
 Number of outputs = $16 + 1 = 17$

Thus, this would require a $2^{33} \times 17$ -bit ROM.

- (b) Number of inputs = 16
 Number of outputs = 16

Thus, this would require a $2^{16} \times 16$ -bit ROM.

- (c) Number of inputs = 16
 Number of outputs = 4

Thus, this would require a 2^{16} x 4-bit ROM.

All of these implementations are not good design choices. They could all be implemented in a smaller amount of hardware using discrete gates.

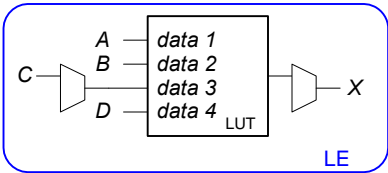
Exercise 5.58

- (a) Yes. Both circuits can compute any function of K inputs and K outputs.
- (b) No. The second circuit can only represent 2^K states. The first can represent more.
- (c) Yes. Both circuits compute any function of 1 input, N outputs, and 2^K states.
- (d) No. The second circuit forces the output to be the same as the state encoding, while the first one allows outputs to be independent of the state encoding.

Exercise 5.59

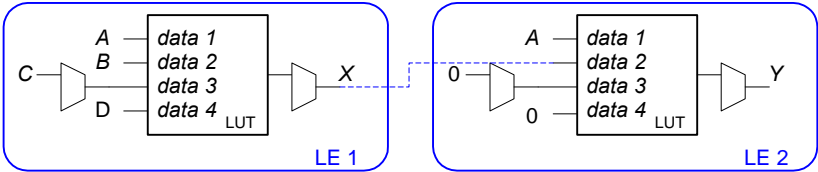
(a) 1 LE

(A)	(B)	(C)	(D)	(Y)
data 1	data 2	data 3	data 4	LUT output
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1



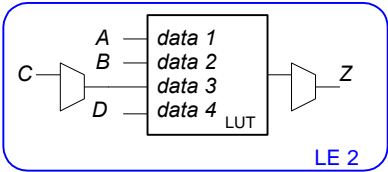
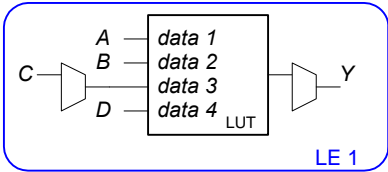
(b) 2 LEs

(B) <i>data 1</i>	(C) <i>data 2</i>	(D) <i>data 3</i>	(E) <i>data 4</i>	(X) LUT output	(A) <i>data 1</i>	(X) <i>data 2</i>	<i>data 3</i>	<i>data 4</i>	(Y) LUT output
0	0	0	0	1	0	0	X	X	0
0	0	0	1	1	0	1	X	X	1
0	0	1	0	1	1	0	X	X	1
0	0	1	1	1	1	1	X	X	1
0	1	0	0	1					
0	1	0	1	0					
0	1	1	0	0					
0	1	1	1	0					
1	0	0	0	1					
1	0	0	1	0					
1	0	1	0	0					
1	0	1	1	0					
1	1	0	0	1					
1	1	0	1	0					
1	1	1	0	1					
1	1	1	0	0					
1	1	1	1	0					
1	1	1	1	0					



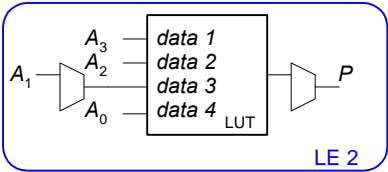
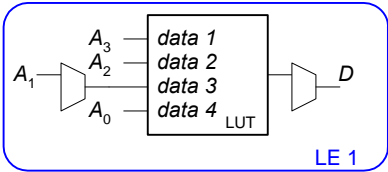
(c) 2 LEs

(A)	(B)	(C)	(D)	(Y)	(A)	(B)	(C)	(D)	(Z)
<i>data 1</i>	<i>data 2</i>	<i>data 3</i>	<i>data 4</i>	LUT output	<i>data 1</i>	<i>data 2</i>	<i>data 3</i>	<i>data 4</i>	LUT output
0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	1	0
0	0	1	0	0	0	0	1	0	0
0	0	1	1	1	0	0	1	1	0
0	1	0	0	0	0	1	0	0	0
0	1	0	1	1	0	1	0	1	1
0	1	1	0	0	0	1	1	0	0
0	1	1	1	1	0	1	1	1	1
1	0	0	0	0	1	0	0	0	0
1	0	0	1	1	1	0	0	1	1
1	0	1	0	1	1	0	1	0	0
1	0	1	1	1	1	0	1	1	0
1	1	0	0	0	1	1	0	0	0
1	1	0	1	1	1	1	0	1	1
1	1	1	0	0	1	1	1	0	0
1	1	1	1	1	1	1	1	1	1



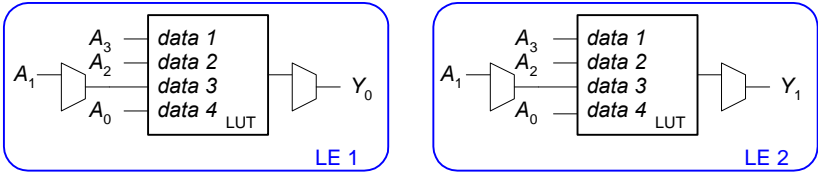
(d) 2 LEs

(A ₃)	(A ₂)	(A ₁)	(A ₀)	(D)	(A ₃)	(A ₂)	(A ₁)	(A ₀)	(P)
data 1	data 2	data 3	data 4	LUT output	data 1	data 2	data 3	data 4	LUT output
0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	1	0
0	0	1	0	0	0	0	1	0	1
0	0	1	1	1	0	0	1	1	1
0	1	0	0	0	0	1	0	0	0
0	1	0	1	0	0	1	0	1	1
0	1	1	0	1	0	1	1	0	0
0	1	1	1	0	0	1	1	1	1
1	0	0	0	0	1	0	0	0	0
1	0	0	1	1	1	0	0	1	0
1	0	1	0	0	1	0	1	0	0
1	0	1	1	0	1	0	1	1	1
1	1	0	0	1	1	1	0	0	0
1	1	0	1	0	1	1	0	1	1
1	1	1	0	0	1	1	1	0	0
1	1	1	1	1	1	1	1	1	0



(e) 2 LEs

(A ₃)	(A ₂)	(A ₁)	(A ₀)	(Y ₀)	(A ₃)	(A ₂)	(A ₁)	(A ₀)	(Y ₁)
data 1	data 2	data 3	data 4	LUT output	data 1	data 2	data 3	data 4	LUT output
0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	1	0
0	0	1	0	1	0	0	1	0	0
0	0	1	1	1	0	0	1	1	0
0	1	0	0	0	0	1	0	0	1
0	1	0	1	0	0	1	0	1	1
0	1	1	0	0	0	1	1	0	1
0	1	1	1	0	0	1	1	1	1
1	0	0	0	1	1	0	0	0	1
1	0	0	1	1	1	0	0	1	1
1	0	1	0	1	1	0	1	0	1
1	0	1	1	1	1	0	1	1	1
1	1	0	0	1	1	1	0	0	1
1	1	0	1	1	1	1	0	1	1
1	1	1	0	1	1	1	1	0	1
1	1	1	0	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1	1



Exercise 5.60

(a) 8 LEs (see next page for figure)

LE 1

data 1	(A ₂) data 2	(A ₁) data 3	(A ₀) data 4	(Y ₀) LUT output
X	0	0	0	1
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

LE 2

data 1	(A ₂) data 2	(A ₁) data 3	(A ₀) data 4	(Y ₁) LUT output
X	0	0	0	0
X	0	0	1	1
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

LE 3

data 1	(A ₂) data 2	(A ₁) data 3	(A ₀) data 4	(Y ₂) LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	1
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

LE 4

data 1	(A ₂) data 2	(A ₁) data 3	(A ₀) data 4	(Y ₃) LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	1
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

LE 5

data 1	(A ₂) data 2	(A ₁) data 3	(A ₀) data 4	(Y ₄) LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	1
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

LE 6

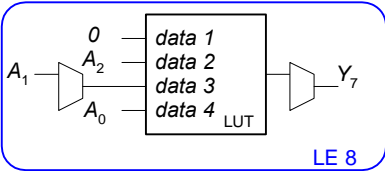
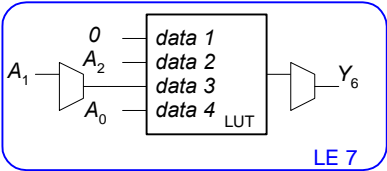
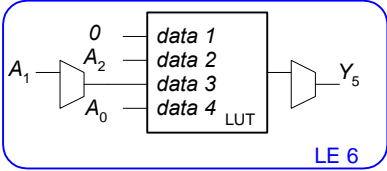
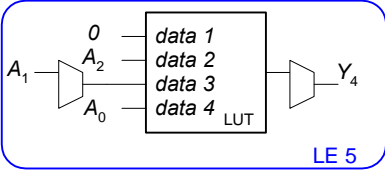
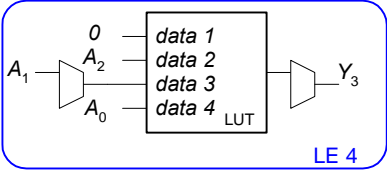
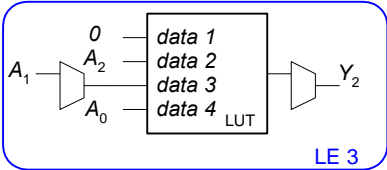
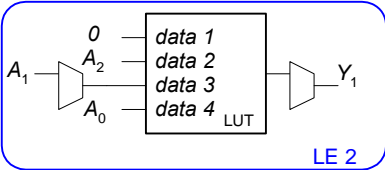
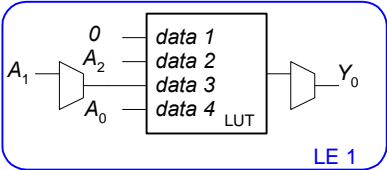
data 1	(A ₂) data 2	(A ₁) data 3	(A ₀) data 4	(Y ₅) LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	1
X	1	1	0	0
X	1	1	1	0

LE 7

data 1	(A ₂) data 2	(A ₁) data 3	(A ₀) data 4	(Y ₆) LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	1
X	1	1	1	0

LE 8

data 1	(A ₂) data 2	(A ₁) data 3	(A ₀) data 4	(Y ₇) LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	1



(b) 8 LEs (see next page for figure)

LE 7

	(A ₂)	(A ₁)	(A ₀)	(Y ₇)
data 1	data 2	data 3	data 4	LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	1

LE 6

	(A ₂)	(A ₁)	(A ₀)	(Y ₆)
data 1	data 2	data 3	data 4	LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	1
X	1	1	1	0

LE 5

	(A ₂)	(A ₁)	(A ₀)	(Y ₅)
data 1	data 2	data 3	data 4	LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	1
X	1	1	0	0
X	1	1	1	0

LE 4

	(A ₂)	(A ₁)	(A ₀)	(Y ₄)
data 1	data 2	data 3	data 4	LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	1
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

LE 3

	(A ₂)	(A ₁)	(A ₀)	(Y ₃)
data 1	data 2	data 3	data 4	LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	1
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

LE 2

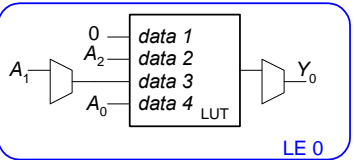
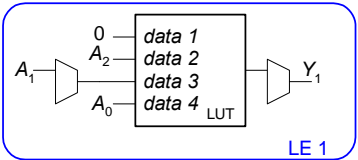
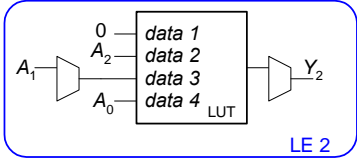
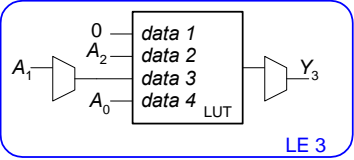
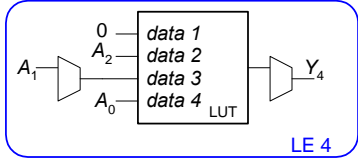
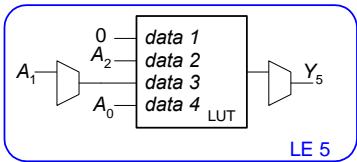
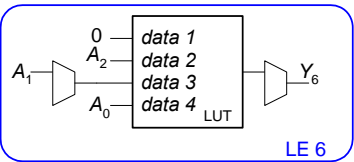
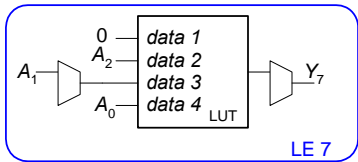
	(A ₂)	(A ₁)	(A ₀)	(Y ₂)
data 1	data 2	data 3	data 4	LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	1
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

LE 1

	(A ₂)	(A ₁)	(A ₀)	(Y ₁)
data 1	data 2	data 3	data 4	LUT output
X	0	0	0	0
X	0	0	1	1
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

LE 0

	(A ₂)	(A ₁)	(A ₀)	(Y ₀)
data 1	data 2	data 3	data 4	LUT output
X	0	0	0	1
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0



(c) 6 LEs (see next page for figure)

LE 1

data 1	data 2	(A ₀) data 3	(B ₀) data 4	(S ₀) LUT output
X	X	0	0	0
X	X	0	1	1
X	X	1	0	1
X	X	1	1	1

LE 2

data 1	data 2	(A ₀) data 3	(B ₀) data 4	(S ₁) LUT output
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

LE 3

data 1	data 2	(B ₀) data 3	(A ₁) data 4	(B ₁) data 4	(C ₁) LUT output
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	1	1
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	1	1
1	0	1	1	1	0
1	1	0	0	0	0
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	1

LE 4

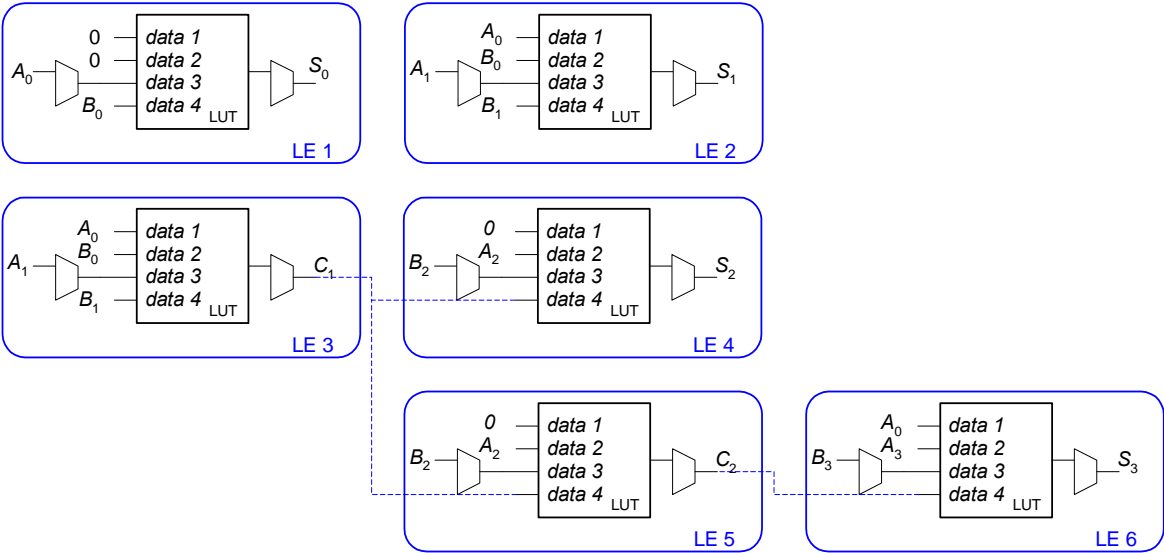
data 1	(A ₂) data 2	(B ₂) data 3	(C ₁) data 4	(S ₂) LUT output
X	0	0	0	0
X	0	0	1	1
X	0	1	0	1
X	0	1	1	0
X	1	0	0	1
X	1	0	1	0
X	1	1	0	0
X	1	1	1	1

LE 5

data 1	(A ₂) data 2	(B ₂) data 3	(C ₁) data 4	(C ₂) LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	1
X	1	0	0	0
X	1	0	1	1
X	1	1	0	1
X	1	1	1	1

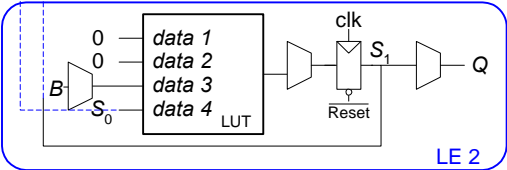
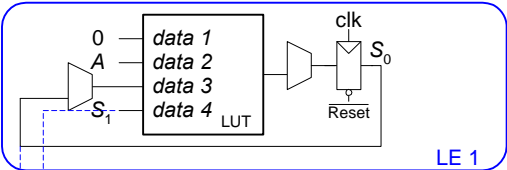
LE 6

data 1	(A ₃) data 2	(B ₃) data 3	(C ₂) data 4	(S ₃) LUT output
X	0	0	0	0
X	0	0	1	1
X	0	1	0	1
X	0	1	1	0
X	1	0	0	1
X	1	0	1	0
X	1	1	0	0
X	1	1	1	1



(d) 2 LEs

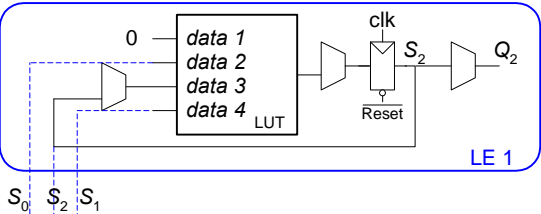
	(A)	(S ₀)	(S ₁)	(S ₀)
data 1	data 2	data 3	data 4	LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	1
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0



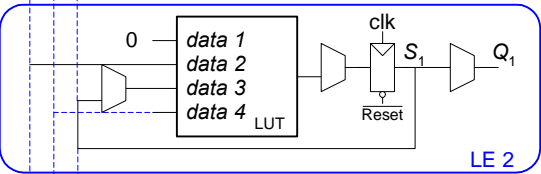
	(B)	(S ₀)	(S ₁)	(S ₁)
data 1	data 2	data 3	data 4	LUT output
X	X	0	0	0
X	X	0	1	0
X	X	1	0	0
X	X	1	1	1

(e) 3 LEs

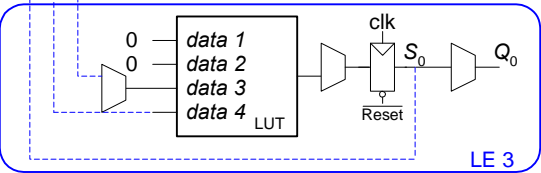
	(S ₀)	(S ₂)	(S ₁)	(S ₂)
data 1	data 2	data 3	data 4	LUT output
X	0	0	0	0
X	0	0	1	1
X	0	1	0	0
X	0	1	1	1
X	1	0	0	0
X	1	0	1	0
X	1	1	0	1
X	1	1	1	1



	(S ₀)	(S ₁)	(S ₂)	(S ₁)
data 1	data 2	data 3	data 4	LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	1
X	0	1	1	1
X	1	0	0	1
X	1	0	1	0
X	1	1	0	1
X	1	1	1	0



		(S ₁)	(S ₂)	(S ₀)
data 1	data 2	data 3	data 4	LUT output
X	X	0	0	1
X	X	0	1	0
X	X	1	0	0
X	X	1	1	1



Exercise 5.61

(a) 5 LEs (2 for next state logic and state registers, 3 for output logic)

(b)

$$\begin{aligned}
 t_{pd} &= t_{pd_LE} + t_{wire} \\
 &= (381 + 246) \text{ ps} \\
 &= 627 \text{ ps}
 \end{aligned}$$

$$\begin{aligned}
 T_c &\geq t_{pcq} + t_{pd} + t_{setup} \\
 &\geq [199 + 627 + 76] \text{ ps} \\
 &= 902 \text{ ps}
 \end{aligned}$$

$$f = 1 / 902 \text{ ps} = \mathbf{1.1 \text{ GHz}}$$

(c)

First, we check that there is no hold time violation with this amount of clock skew.

$$\begin{aligned}
 t_{cd_LE} &= t_{pd_LE} = 381 \text{ ps} \\
 t_{cd} &= t_{cd_LE} + t_{wire} = 627 \text{ ps}
 \end{aligned}$$

$$\begin{aligned}
 t_{skew} &< (t_{ccq} + t_{cd}) - t_{hold} \\
 &< [(199 + 627) - 0] \text{ ps} \\
 &< \mathbf{826 \text{ ps}}
 \end{aligned}$$

3 ns is less than 826 ps, so there is no hold time violation.

Now we find the fastest frequency at which it can run.

$$\begin{aligned}
 T_c &\geq t_{pcq} + t_{pd} + t_{setup} + t_{skew} \\
 &\geq [0.902 + 3] \text{ ns} \\
 &= 3.902 \text{ ns}
 \end{aligned}$$

$$f = 1 / 3.902 \text{ ns} = \mathbf{256 \text{ MHz}}$$

Exercise 5.62

(a) 2 LEs (1 for next state logic and state register, 1 for output logic)

(b) Same as answer for Exercise 5.57(b)

(c) Same as answer for Exercise 5.57(c)

Exercise 5.63

First, we find the cycle time:

$$T_c = 1/f = 1/100 \text{ MHz} = 10 \text{ ns}$$

$$\begin{aligned}
 T_c &\geq t_{pcq} + N t_{LE+wire} + t_{setup} \\
 10 \text{ ns} &\geq [0.199 + N(0.627) + 0.076] \text{ ns}
 \end{aligned}$$

Thus, $N \leq 15.5$

The maximum number of LEs on the critical path is **15**.

With at most one LE on the critical path and no clock skew, the fastest the FSM will run is:

$$\begin{aligned} T_c &\geq [0.199 + 0.627 + 0.076] \text{ ns} \\ &\geq 0.902 \text{ ns} \\ f &= 1 / 0.902 \text{ ns} = \mathbf{1.1 \text{ GHz}} \end{aligned}$$

Question 5.1

$$(2^N-1)(2^N-1) = 2^{2N} - 2^{N+1} + 1$$

Question 5.2

A processor might use BCD representation so that decimal numbers, such as 1.7, can be represented exactly.

Question 5.3

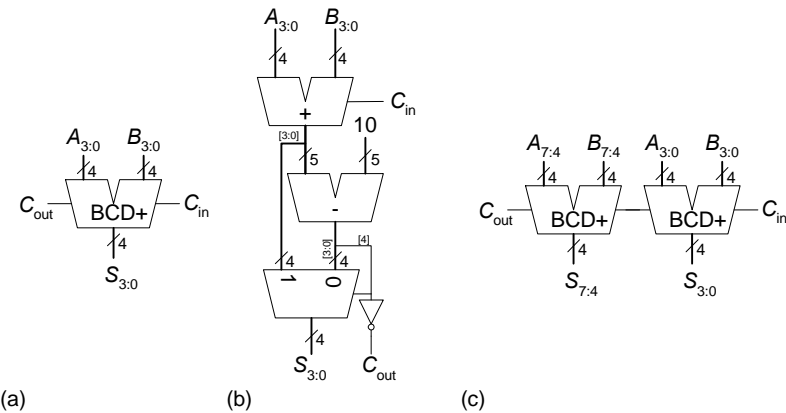


FIGURE 5.22 BCD adder: (a) 4-bit block, (b) underlying hardware, (c) 8-bit BCD adder

*(continued from previous page)***SystemVerilog**

```

module bcdadd_8(input  logic [7:0] a, b,
               input  logic      cin,
               output logic [7:0] s,
               output logic      cout);

    logic c0;

    bcdadd_4 bcd0(a[3:0], b[3:0], cin, s[3:0], c0);
    bcdadd_4 bcd1(a[7:4], b[7:4], c0, s[7:4], cout);

endmodule

module bcdadd_4(input  logic [3:0] a, b,
               input  logic      cin,
               output logic [3:0] s,
               output logic      cout);

    logic [4:0] result, sub10;

    assign result = a + b + cin;
    assign sub10 = result - 10;

    assign cout = ~sub10[4];
    assign s = sub10[4] ? result[3:0] : sub10[3:0];

endmodule

```

VHDL

```

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity bcdadd_8 is
    port(a, b: in  STD_LOGIC_VECTOR(7 downto 0);
          cin: in  STD_LOGIC;
          s:   out STD_LOGIC_VECTOR(7 downto 0);
          cout: out STD_LOGIC);
end;

architecture synth of bcdadd_8 is
    component bcdadd_4
        port(a, b: in  STD_LOGIC_VECTOR(3 downto 0);
              cin: in  STD_LOGIC;
              s:   out STD_LOGIC_VECTOR(3 downto 0);
              cout: out STD_LOGIC);
    end component;
    signal c0: STD_LOGIC;
begin

    bcd0: bcdadd_4
        port map(a(3 downto 0), b(3 downto 0), cin, s(3
downto 0), c0);
    bcd1: bcdadd_4
        port map(a(7 downto 4), b(7 downto 4), c0, s(7
downto 4), cout);

end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;

entity bcdadd_4 is
    port(a, b: in  STD_LOGIC_VECTOR(3 downto 0);
          cin: in  STD_LOGIC;
          s:   out STD_LOGIC_VECTOR(3 downto 0);
          cout: out STD_LOGIC);
end;

architecture synth of bcdadd_4 is
    signal result, sub10, a5, b5: STD_LOGIC_VECTOR(4
downto 0);
begin
    a5 <= '0' & a;
    b5 <= '0' & b;
    result <= a5 + b5 + cin;
    sub10 <= result - "01010";

    cout <= not (sub10(4));
    s <= result(3 downto 0) when sub10(4) = '1'
        else sub10(3 downto 0);

end;

```