CHAPTER 1

SOLUTIONS

2 CHAPTER solutions

CHAPTER 1

Exercise 1.1

(a) Biologists study cells at many levels. The cells are built from organelles such as the mitochondria, ribosomes, and chloroplasts. Organelles are built of macromolecules such as proteins, lipids, nucleic acids, and carbohydrates. These biochemical macromolecules are built simpler molecules such as carbon chains and amino acids. When studying at one of these levels of abstraction, biologists are usually interested in the levels above and below: what the structures at that level are used to build, and how the structures themselves are built.

(b) The fundamental building blocks of chemistry are electrons, protons, and neutrons (physicists are interested in how the protons and neutrons are built). These blocks combine to form atoms. Atoms combine to form molecules. For example, when chemists study molecules, they can abstract away the lower levels of detail so that they can describe the general properties of a molecule such as benzene without having to calculate the motion of the individual electrons in the molecule.

Exercise 1.2

(a) Automobile designers use hierarchy to construct a car from major assemblies such as the engine, body, and suspension. The assemblies are constructed from subassemblies; for example, the engine contains cylinders, fuel injectors, the ignition system, and the drive shaft. Modularity allows components to be swapped without redesigning the rest of the car; for example, the seats can be cloth, leather, or leather with a built in heater depending on the model of the vehicle, so long as they all mount to the body in the same place. Regularity involves the use of interchangeable parts and the sharing of parts between different vehicles; a 65R14 tire can be used on many different cars.

(b) Businesses use hierarchy in their organization chart. An employee reports to a manager, who reports to a general manager who reports to a vice president who reports to the president. Modularity includes well-defined interfaces between divisions. The salesperson who spills a coke in his laptop calls a single number for technical support and does not need to know the detailed organization of the information systems department. Regularity includes the use of standard procedures. Accountants follow a well-defined set of rules to calculate profit and loss so that the finances of each division can be combined to determine the finances of the company and so that the finances of the company can be reported to investors who can make a straightforward comparison with other companies.

Exercise 1.3

Ben can use a hierarchy to design the house. First, he can decide how many bedrooms, bathrooms, kitchens, and other rooms he would like. He can then jump up a level of hierarchy to decide the overall layout and dimensions of the house. At the top-level of the hierarchy, he material he would like to use, what kind of roof, etc. He can then jump to an even lower level of hierarchy to decide the specific layout of each room, where he would like to place the doors, windows, etc. He can use the principle of regularity in planning the framing of the house. By using the same type of material, he can scale the framing depending on the dimensions of each room. He can also use regularity to choose the same (or a small set of) doors and windows for each room. That way, when he places a new door or window he need not redesign the size, material, layout specifications from scratch. This is also an example of modularity: once he has designed the specifications for the windows in one room, for example, he need not respecify them when he uses the same windows in another room. This will save him both design time and, thus, money. He could also save by buying some items (like windows) in bulk.

Exercise 1.4

An accuracy of +/- 50 mV indicates that the signal can be resolved to 100 mV intervals. There are 50 such intervals in the range of 0-5 volts, so the signal represents $\log_2 50 = 5.64$ bits of information.

Exercise 1.5

(a) The hour hand can be resolved to 12 * 4 = 48 positions, which represents $\log_2 48 = 5.58$ bits of information. (b) Knowing whether it is before or after noon adds one more bit.

Exercise 1.6

Each digit conveys $\log_2 60 = 5.91$ bits of information. $4000_{10} = 1\ 6\ 40_{60}$ (1 in the 3600 column, 6 in the 60's column, and 40 in the 1's column).

Exercise 1.7

$2^{16} = 65{,}536$ numbers.

Exercise 1.8

$2^{32}-1 = 4{,}294{,}967{,}295$

Exercise 1.9

(a) $2^{16}-1 = 65535$; (b) $2^{15}-1 = 32767$; (c) $2^{15}-1 = 32767$

Exercise 1.10

(a) $2^{32}-1 = 4{,}294{,}967{,}295$; (b) $2^{31}-1 = 2{,}147{,}483{,}647$; (c) $2^{31}-1 = 2{,}147{,}483{,}647$

Exercise 1.11

(a) 0; (b) $-2^{15} = -32768$; (c) $-(2^{15}-1) = -32767$

Exercise 1.12

(a) 0; (b) $-2^{31} = -2{,}147{,}483{,}648$; (c) $-(2^{31}-1) = -2{,}147{,}483{,}647$;

Exercise 1.13

(a) 10; (b) 54; (c) 240; (d) 6311

Exercise 1.14

(a) 14; (b) 36; (c) 215; (d) 15,012

Exercise 1.15

(a) A; (b) 36; (c) F0; (d) 18A7

Exercise 1.16

(a) E; (b) 24; (c) D7; (d) 3AA4

Exercise 1.17

(a) 165; (b) 59; (c) 65535; (d) 3489660928

Exercise 1.18

(a) 78; (b) 124; (c) 60,730; (d) 1,077,915, 649

Exercise 1.19

(a) 10100101; (b) 00111011; (c) 1111111111111111;
(d) 11010000000000000000000000000000

Exercise 1.20

(a) 1001110; (b) 1111100; (c) 1110110100111010; (d) 100 0000 0011 1111 1011 0000 0000 0001

Exercise 1.21

(a) -6; (b) -10; (c) 112; (d) -97

Exercise 1.22

(a) -2 (-8+4+2 = -2 or magnitude = 0001+1 = 0010: thus, -2); (b) -29 (-32 + 2 + 1 = -29 or magnitude = 011100+1 = 011101: thus, -29); (c) 78; (d) -75

Exercise 1.23

(a) -2; (b) -22; (c) 112; (d) -31

Exercise 1.24

(a) -6; (b) -3; (c) 78; (d) -53

Exercise 1.25

(a) 101010; (b) 111111; (c) 11100101; (d) 1101001101

Exercise 1.26

(a) 1110; (b) 110100; (c) 101010011; (d) 1011000111

Exercise 1.27

(a) 2A; (b) 3F; (c) E5; (d) 34D

Exercise 1.28

(a) E; (b) 34; (c) 153; (d) 2C7;

Exercise 1.29

(a) 00101010; (b) 11000001; (c) 01111100; (d) 10000000; (e) overflow

Exercise 1.30

(a) 00011000; (b) 11000101; (c) overflow; (d) overflow; (e) 01111111\

Exercise 1.31

00101010; (b) 10111111; (c) 01111100; (d) overflow; (e) overflow

Exercise 1.32

(a) 00011000; (b) 10111011; (c) overflow; (d) overflow; (e) 01111111

Exercise 1.33

(a) 00000101; (b) 11111010

Exercise 1.34

(a) 00000111; (b) 11111001

Exercise 1.35

(a) 00000101; (b) 00001010

Exercise 1.36

(a) 00000111; (b) 00001001

Exercise 1.37

(a) 52; (b) 77; (c) 345; (d) 1515

Exercise 1.38

(a) 0o16; (b) 0o64; (c) 0o339; (d) 0o1307

Exercise 1.39

(a) 100010_2, 22_{16}, 34_{10}; (b) 110011_2, 33_{16}, 51_{10}; (c) 010101101_2, AD_{16}, 173_{10}; (d) 011000100111_2, 627_{16}, 1575_{10}

Exercise 1.40

(a) 0b10011; 0x13; 19; (b) 0b100101; 0x25; 37; (c) 0b11111001; 0xF9; 249; (d) 0b10101110000; 0x570; 1392

Exercise 1.41

15 greater than 0, 16 less than 0; 15 greater and 15 less for sign/magnitude

Exercise 1.42

(26-1) are greater than 0; 26 are less than 0. For sign/magnitude numbers, (26-1) are still greater than 0, but (26-1) are less than 0.

Exercise 1.43

4, 8

Exercise 1.44

8

Exercise 1.45

5,760,000

Exercise 1.46

(5 × 109 bits/second)(60 seconds/minute)(1 byte/8 bits) = 3.75 × 1010 bytes

Exercise 1.47

46.566 gigabytes

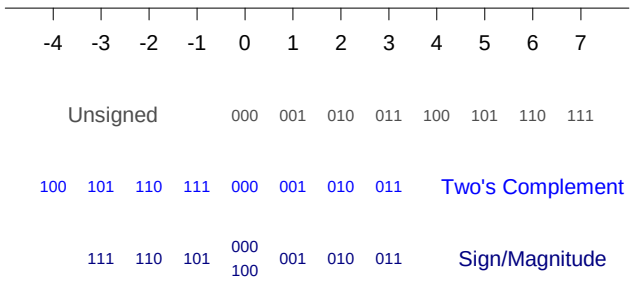
Exercise 1.48

2 billion

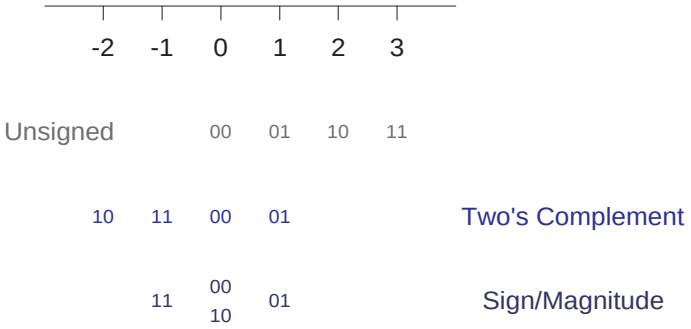
Exercise 1.49

128 kbits

Exercise 1.50



Exercise 1.51



Exercise 1.52

(a) 1101; (b) 11000 (overflows)

Exercise 1.53

(a) 11011101; (b) 110001000 (overflows)

Exercise 1.54

(a) 11012, no overflow; (b) 10002, no overflow

Exercise 1.55

(a) 11011101; (b) 110001000

Exercise 1.56

(a) 010000 + 001001 = 011001;
(b) 011011 + 011111 = 111010 (overflow);
(c) 111100 + 010011 = 001111;
(d) 000011 + 100000 = 100011;
(e) 110000 + 110111 = 100111;
(f) 100101 + 100001 = 000110 (overflow)

Exercise 1.57

(a) 000111 + 001101 = 010100
(b) 010001 + 011001 = 101010, overflow
(c) 100110 + 001000 = 101110
(d) 011111 + 110010 = 010001
(e) 101101 + 101010 = 010111, overflow
(f) 111110 + 100011 = 100001

Exercise 1.58

(a) 10; (b) 3B; (c) E9; (d) 13C (overflow)

Exercise 1.59

(a) 0x2A; (b) 0x9F; (c) 0xFE; (d) 0x66, overflow

Exercise 1.60

(a) 01001 - 00111 = 00010; (b) 01100 - 01111 = 11101; (c) 11010 - 01011 = 01111; (d) 00100 - 11000 = 01100

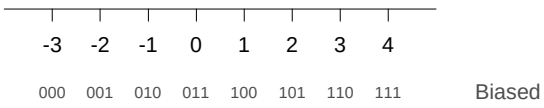
Exercise 1.61

(a) 010010 + 110100 = 000110; (b) 011110 + 110111 = 010101; (c) 100100 + 111101 = 100001; (d) 110000 + 101011 = 011011, overflow

Exercise 1.62

(a) 3; (b) 01111111; (c) $00000000_2 = -127_{10}$; $11111111_2 = 128_{10}$

Exercise 1.63



Exercise 1.64

(a) 001010001001; (b) 951; (c) 1000101; (d) each 4-bit group represents one decimal digit, so conversion between binary and decimal is easy. BCD can also be used to represent decimal fractions exactly.

Exercise 1.65

(a) 0011 0111 0001
(b) 187
(c) 95 = 1011111
(d) Addition of BCD numbers doesn't work directly. Also, the representation doesn't maximize the amount of information that can be stored; for example 2 BCD digits requires 8 bits and can store up to 100 values (0-99) - unsigned 8-bit binary can store 28 (256) values.

Exercise 1.66

Three on each hand, so that they count in base six.

Exercise 1.67

Both of them are full of it. $42_{10} = 101010_2$, which has 3 1's in its representation.

Exercise 1.68

Both are right.

Exercise 1.69

```
#include <stdio.h>
```

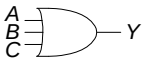
```
void main(void)
{
   char bin[80];
   int i = 0, dec = 0;

   printf("Enter binary number: ");
   scanf("%s", bin);

   while (bin[i] != 0) {
      if (bin[i] == '0') dec = dec * 2;
      else if (bin[i] == '1') dec = dec * 2 + 1;
      else printf("Bad character %c in the number.\n", bin[i]);
      i = i + 1;
   }
   printf("The decimal equivalent is %d\n", dec);
}
```

Exercise 1.70

```
/* This program works for numbers that don't overflow the
   range of an integer. */

#include <stdio.h>

void main(void)
{
   int b1, b2, digits1 = 0, digits2 = 0;
   char num1[80], num2[80], tmp, c;
   int digit, num = 0, j;

   printf ("Enter base #1: "); scanf("%d", &b1);
   printf ("Enter base #2: "); scanf("%d", &b2);
   printf ("Enter number in base %d ", b1); scanf("%s", num1);

   while (num1[digits1] != 0) {
      c = num1[digits1++];
      if (c >= 'a' && c <= 'z') c = c + 'A' - 'a';
      if (c >= '0' && c <= '9') digit = c - '0';
      else if (c >= 'A' && c <= 'F') digit = c - 'A' + 10;
      else printf("Illegal character %c\n", c);
      if (digit >= b1) printf("Illegal digit %c\n", c);
      num = num * b1 + digit;

   }
   while (num > 0) {
      digit = num % b2;
      num = num / b2;
      num2[digits2++] = digit < 10 ? digit + '0' : digit + 'A' -
10;
   }
   num2[digits2] = 0;

   for (j = 0; j < digits2/2; j++) { // reverse order of digits
      tmp = num2[j];
      num2[j] = num2[digits2-j-1];
      num2[digits2-j-1] = tmp;
   }

   printf("The base %d equivalent is %s\n", b2, num2);
}
```

Exercise 1.71

OR3

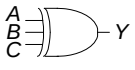


$Y = A+B+C$

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

(a)

XOR3

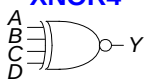


$Y = A \oplus B \oplus C$

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

(b)

XNOR4



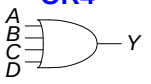
$Y = \overline{A \oplus B \oplus C \oplus D}$

A	C	B	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

(c)

Exercise 1.72

OR4

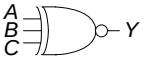


$Y = A+B+C+D$

(a)

A	C	B	D	Y
0	0	0	0	0
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	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

XNOR3

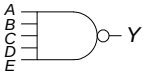


$Y = \overline{A \oplus B \oplus C}$

(b)

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

NAND5



$Y = \overline{ABCDE}$

(c)

A	B	C	D	E	Y
0	0	0	0	0	1
0	0	0	0	1	1
0	0	0	1	0	1
0	0	0	1	1	1
0	0	1	0	0	1
0	0	1	0	1	1
0	0	1	1	0	1
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	0	1	1
0	1	0	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
0	1	1	0	1	1
0	1	1	1	0	1
0	1	1	1	1	1
1	0	0	0	0	1
1	0	0	0	1	1
1	0	0	1	0	1
1	0	0	1	1	1
1	0	1	0	0	1
1	0	1	0	1	1
1	0	1	1	0	1
1	0	1	1	1	1
1	1	0	0	0	1
1	1	0	0	1	1
1	1	0	1	0	1
1	1	0	1	1	1
1	1	1	0	0	1
1	1	1	0	1	1
1	1	1	1	0	1
1	1	1	1	1	0

Exercise 1.73

A	*B*	*C*	*Y*
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Exercise 1.74

A	*B*	*C*	*Y*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Exercise 1.75

A	*B*	*C*	*Y*
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Exercise 1.76

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	0

Zero

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

A NOR B

A	B	Y
0	0	0
0	1	1
1	0	0
1	1	0

$\overline{A}B$

A	B	Y
0	0	1
0	1	1
1	0	0
1	1	0

NOT A

A	B	Y
0	0	0
0	1	0
1	0	1
1	1	0

$A\overline{B}$

A	B	Y
0	0	1
0	1	0
1	0	1
1	1	0

NOT B

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

XOR

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

NAND

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

AND

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

XNOR

A	B	Y
0	0	0
0	1	1
1	0	0
1	1	1

B

A	B	Y
0	0	1
0	1	1
1	0	0
1	1	1

$\overline{A} + B$

A	B	Y
0	0	0
0	1	0
1	0	1
1	1	1

A

A	B	Y
0	0	1
0	1	0
1	0	1
1	1	1

$A + \overline{B}$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

OR

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	1

One

Exercise 1.77

2^{2^N}

Exercise 1.78

$V_{IL} = 2.5$; $V_{IH} = 3$; $V_{OL} = 1.5$; $V_{OH} = 4$; $NM_L = 1$; $NM_H = 1$

Exercise 1.79

No, there is no legal set of logic levels. The slope of the transfer characteristic never is better than -1, so the system never has any gain to compensate for noise.

Exercise 1.80

$V_{IL} = 2$; $V_{IH} = 4$; $V_{OL} = 1$; $V_{OH} = 4.5$; $NM_L = 1$; $NM_H = 0.5$

Exercise 1.81

The circuit functions as a buffer with logic levels V_{IL} = 1.5; V_{IH} = 1.8; V_{OL} = 1.2; V_{OH} = 3.0. It can receive inputs from LVCMOS and LVTTL gates because their output logic levels are compatible with this gate's input levels. However, it cannot drive LVCMOS or LVTTL gates because the 1.2 V_{OL} exceeds the V_{IL} of LVCMOS and LVTTL.

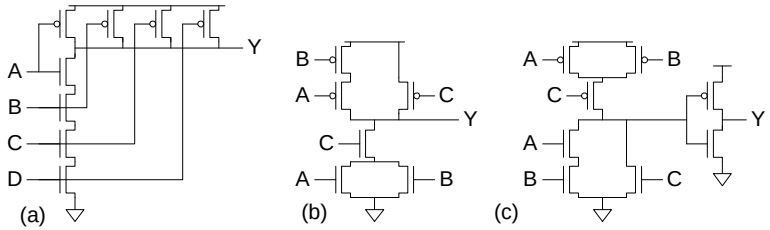
Exercise 1.82

(a) AND gate; (b) V_{IL} = 1.5; V_{IH} = 2.75; V_{OL} = 0; V_{OH} = 3

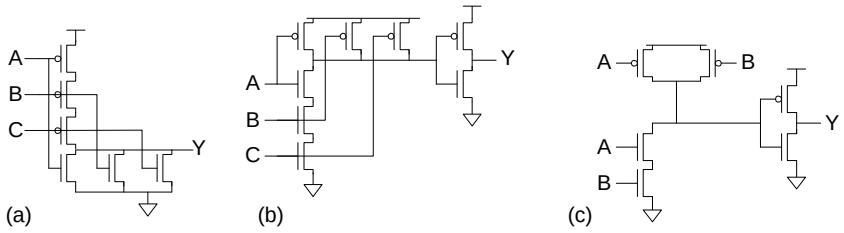
Exercise 1.83

(a) XOR gate; (b) V_{IL} = 1.25; V_{IH} = 2; V_{OL} = 0; V_{OH} = 3

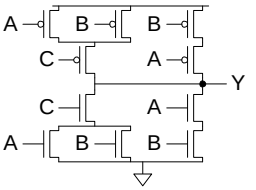
Exercise 1.84



Exercise 1.85



Exercise 1.86



Exercise 1.87

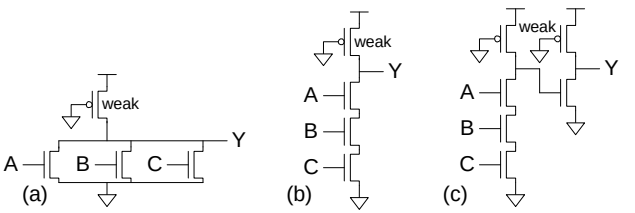
XOR

A	*B*	*Y*
0	0	0
0	1	1
1	0	1
1	1	0

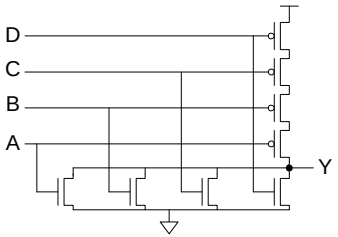
Exercise 1.88

A	*B*	*C*	*Y*
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

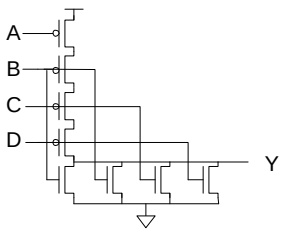
Exercise 1.89



Exercise 1.90



Question 1.1



Question 1.2

4 times. Place 22 coins on one side and 22 on the other. If one side rises, the fake is on that side. Otherwise, the fake is among the 20 remaining. From the group containing the fake, place 8 on one side and 8 on the other. Again, identify which group contains the fake. From that group, place 3 on one side and 3 on the other. Again, identify which group contains the fake. Finally, place 1 coin on each side. Now the fake coin is apparent.

Question 1.3

17 minutes: (1) designer and freshman cross (2 minutes); (2) freshman returns (1 minute); (3) professor and TA cross (10 minutes); (4) designer returns (2 minutes); (5) designer and freshman cross (2 minutes).

CHAPTER 2

Exercise 2.1

(a) $Y = \bar{A}\bar{B} + A\bar{B} + AB$

(b) $Y = \bar{A}\bar{B}\bar{C} + ABC$

(c) $Y = \bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C} + A\bar{B}\bar{C} + A\bar{B}C + ABC$

(d)

$$Y = \bar{A}\bar{B}\bar{C}D + \bar{A}\bar{B}C\bar{D} + \bar{A}B\bar{C}\bar{D} + \bar{A}BCD + A\bar{B}\bar{C}\bar{D} + A\bar{B}C\bar{D} + ABC\bar{D}$$

(e)

$$Y = \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}CD + \bar{A}B\bar{C}D + \bar{A}BC\bar{D} + A\bar{B}\bar{C}D + A\bar{B}C\bar{D} + AB\bar{C}\bar{D} + ABCD$$

Exercise 2.2

(a) $Y = \bar{A}B + A\bar{B} + AB$

(b) $Y = \bar{A}\bar{B}C + \bar{A}B\bar{C} + \bar{A}BC + A\bar{B}\bar{C} + AB\bar{C}$

(c) $Y = \bar{A}\bar{B}C + AB\bar{C} + ABC$

(d) $Y = \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}C\bar{D} + \bar{A}\bar{B}CD + \bar{A}B\bar{C}\bar{D} + \bar{A}BCD + A\bar{B}\bar{C}\bar{D} + A\bar{B}C\bar{D}$

(e) $Y = \bar{A}\bar{B}CD + \bar{A}BC\bar{D} + \bar{A}BCD + A\bar{B}\bar{C}\bar{D} + A\bar{B}\bar{C}D + A\bar{B}C\bar{D} + A\bar{B}CD$

Exercise 2.3

(a) $Y = (A + \bar{B})$

(b)

$$Y = (A + B + \overline{C})(A + \overline{B} + C)(A + \overline{B} + \overline{C})(\overline{A} + B + C)(\overline{A} + B + \overline{C})(\overline{A} + \overline{B} + C)$$

(c) $Y = (A + B + \overline{C})(A + \overline{B} + \overline{C})(\overline{A} + \overline{B} + C)$

(d)

$$Y = (A + \overline{B} + C + D)(A + \overline{B} + C + \overline{D})(A + \overline{B} + \overline{C} + D)(A + \overline{B} + \overline{C} + \overline{D})(\overline{A} + B + C + \overline{D})$$
$$(\overline{A} + B + \overline{C} + \overline{D})(\overline{A} + \overline{B} + C + D)(\overline{A} + \overline{B} + C + \overline{D})(\overline{A} + \overline{B} + \overline{C} + \overline{D})$$

(e)

$$Y = (A + B + C + \overline{D})(A + B + \overline{C} + D)(A + \overline{B} + C + D)(A + \overline{B} + \overline{C} + \overline{D})(\overline{A} + B + C + D)$$
$$(\overline{A} + B + \overline{C} + \overline{D})(\overline{A} + \overline{B} + C + \overline{D})(\overline{A} + \overline{B} + \overline{C} + D)$$

Exercise 2.4

(a) $Y = A + B$

(b) $Y = (A + B + C)(\overline{A} + B + \overline{C})(\overline{A} + \overline{B} + \overline{C})$

(c) $Y = (A + B + C)(A + \overline{B} + C)(A + \overline{B} + \overline{C})(\overline{A} + B + C)(\overline{A} + B + \overline{C})$

(d)

$$Y = (A + B + C + \overline{D})(A + \overline{B} + C + D)(A + \overline{B} + C + \overline{D})(\overline{A} + B + C + \overline{D})$$
$$(\overline{A} + B + \overline{C} + \overline{D})(\overline{A} + \overline{B} + C + D)(\overline{A} + \overline{B} + C + \overline{D})(\overline{A} + \overline{B} + \overline{C} + D)$$
$$(\overline{A} + \overline{B} + \overline{C} + \overline{D})$$

(e)

$$Y = (A + B + C + D)(A + B + C + \overline{D})(A + B + \overline{C} + D)(A + \overline{B} + C + D)$$
$$(A + \overline{B} + C + \overline{D})(\overline{A} + \overline{B} + C + D)(\overline{A} + \overline{B} + C + \overline{D})(\overline{A} + \overline{B} + \overline{C} + D)$$
$$(\overline{A} + \overline{B} + \overline{C} + \overline{D})$$

Exercise 2.5

(a) $Y = A + \overline{B}$

(b) $Y = \overline{A}\,\overline{B}\,\overline{C} + ABC$

(c) $Y = \overline{A}\,\overline{C} + A\overline{B} + AC$

(d) $Y = \overline{A}\,\overline{B} + \overline{B}\,\overline{D} + AC\overline{D}$

(e)

$$Y = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}CD + \overline{A}B\overline{C}D + \overline{A}BC\overline{D} + A\overline{B}\,\overline{C}D + A\overline{B}C\overline{D} + AB\overline{C}\,\overline{D} + ABCD$$

This can also be expressed as:

$$Y = \overline{(A \oplus B)}\,\overline{(C \oplus D)} + (A \oplus B)(C \oplus D)$$

Exercise 2.6

(a) $Y = A + B$

(b) $Y = A\bar{C} + \bar{A}C + B\bar{C}$ or $Y = A\bar{C} + \bar{A}C + \bar{A}B$

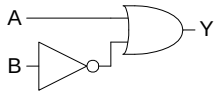
(c) $Y = AB + \bar{A}\bar{B}C$

(d) $Y = BC + \bar{B}\bar{D}$

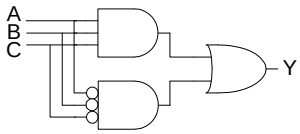
(e) $Y = A\bar{B} + \bar{A}BC + \bar{A}CD$ or $Y = A\bar{B} + \bar{A}BC + \bar{B}CD$

Exercise 2.7

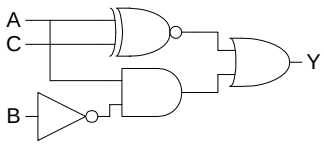
(a)



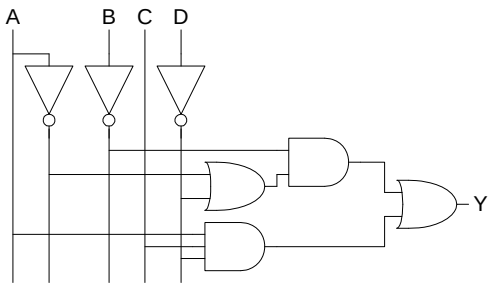
(b)



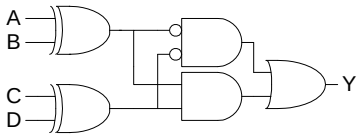
(c)



(d)

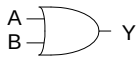


(e)

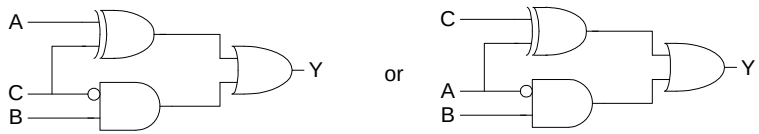


Exercise 2.8

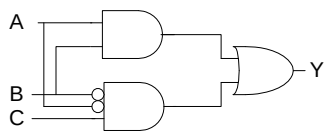
(a)



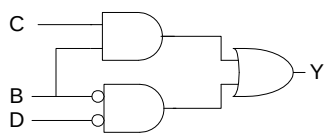
(b)



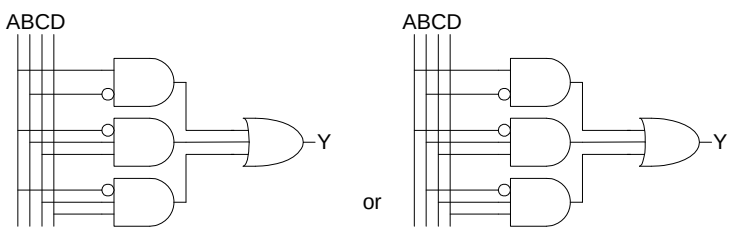
(c)



(d)

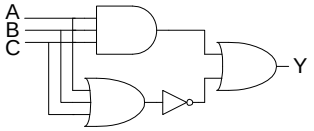


(e)

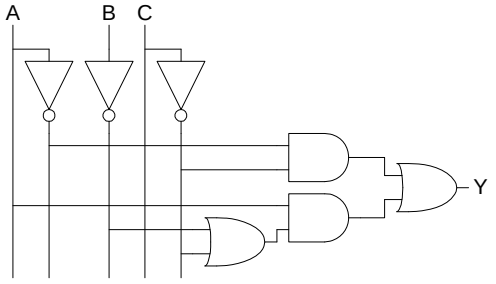


Exercise 2.9

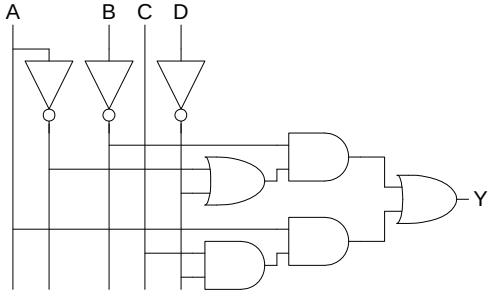
(a) Same as 2.7(a)
(b)



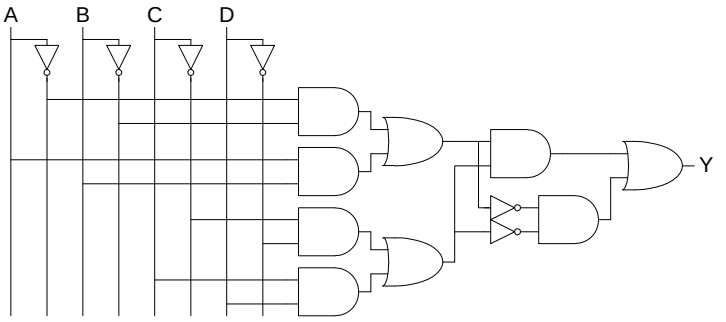
(c)



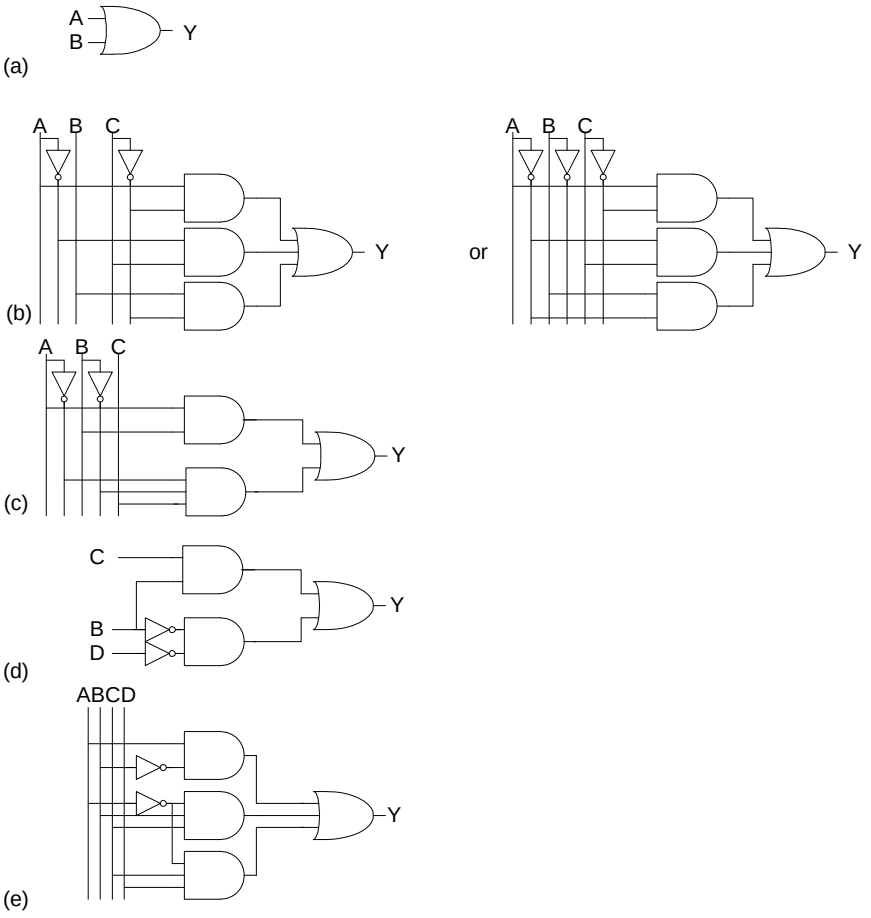
(d)



(e)

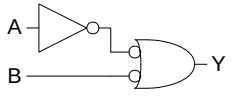


Exercise 2.10

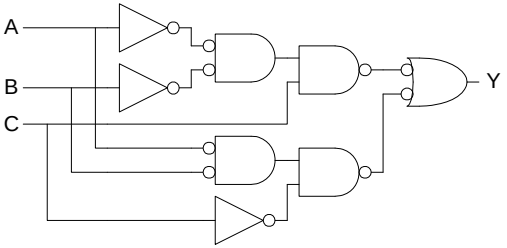


Exercise 2.11

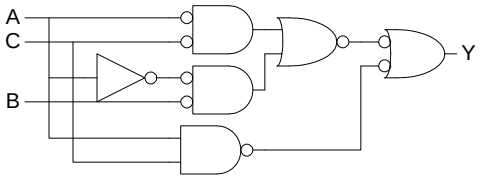
(a)



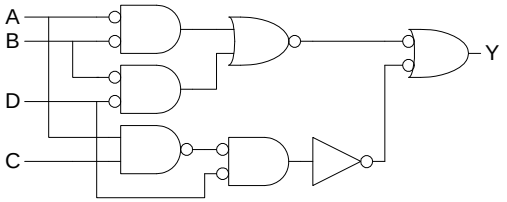
(b)



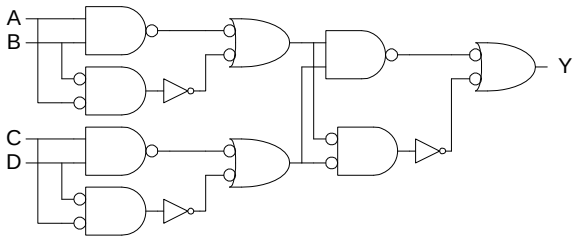
(c)



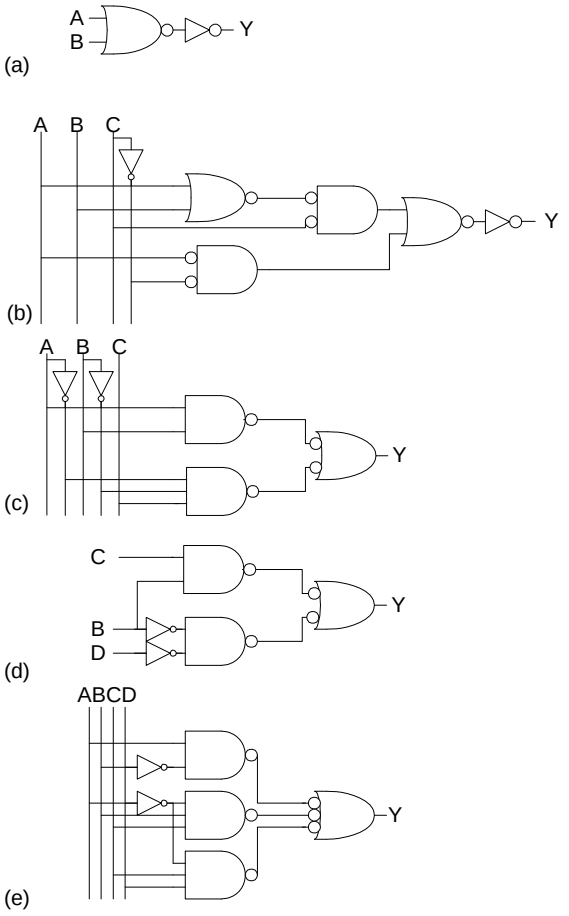
(d)



(e)



Exercise 2.12



Exercise 2.13

(a) $Y = AC + \overline{B}C$
(b) $Y = \overline{A}$
(c) $Y = \overline{A} + \overline{B}\,\overline{C} + \overline{B}\,\overline{D} + BD$

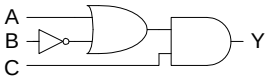
Exercise 2.14

(a) $Y = \overline{A}B$
(b) $Y = \overline{A} + \overline{B} + \overline{C} = \overline{ABC}$

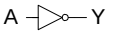
(c) $Y = A(\overline{B} + \overline{C} + \overline{D}) + \overline{B}\overline{C}\overline{D} = A\overline{BCD} + \overline{B}\overline{C}\overline{D}$

Exercise 2.15

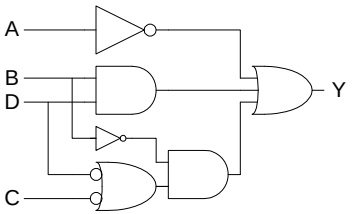
(a)



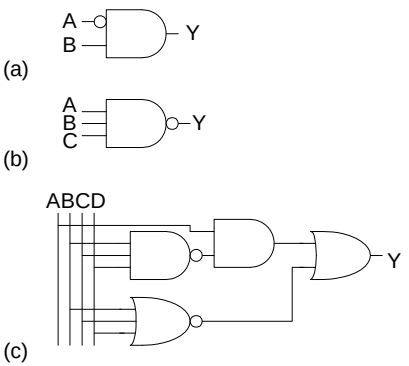
(b)



(c)



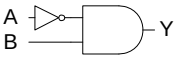
Exercise 2.16



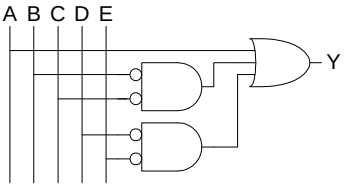
Exercise 2.17

(a) $Y = B + \overline{A}\,\overline{C}$

(b) $Y = \overline{A}B$



(c) $Y = A + \overline{B}\,\overline{C} + \overline{D}\,\overline{E}$



Exercise 2.18

(a) $Y = \overline{B} + C$

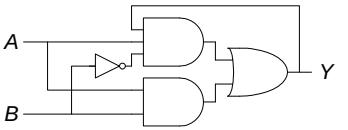
(b) $Y = (A + \overline{C})D + B$

(c) $Y = B\overline{D}E + BD(\overline{A \oplus C})$

Exercise 2.19

4 gigarows = 4 x 2^{30} rows = 2^{32} rows, so the truth table has 32 inputs.

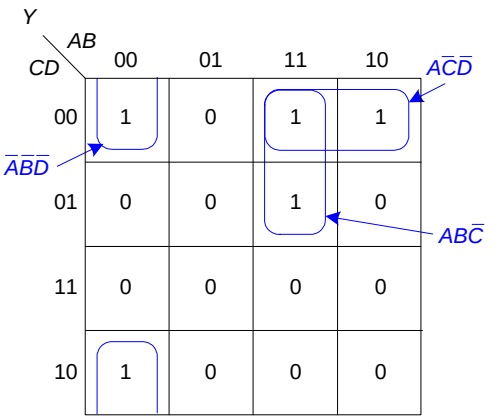
Exercise 2.20



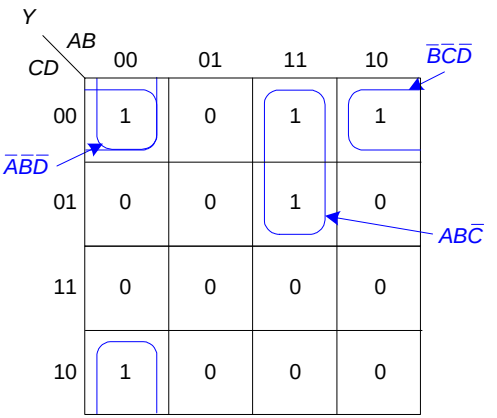
$Y = A$

Exercise 2.21

Ben is correct. For example, the following function, shown as a K-map, has two possible minimal sum-of-products expressions. Thus, although $A\bar{C}\bar{D}$ and $\bar{B}\bar{C}\bar{D}$ are both prime implicants, the minimal sum-of-products expression does not have both of them.



$Y = \bar{A}\bar{B}\bar{D} + AB\bar{C} + A\bar{C}\bar{D}$



$Y = \bar{A}\bar{B}\bar{D} + AB\bar{C} + \bar{B}\bar{C}\bar{D}$

Exercise 2.22

(a)

B	$B \bullet B$
0	0
1	1

(b)

B	C	D	$(B \bullet C) + (B \bullet D)$	$B \bullet (C + D)$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

(c)

B	C	$(B \bullet C) + (B \bullet \overline{C})$
0	0	0
0	1	0
1	0	1
1	1	1

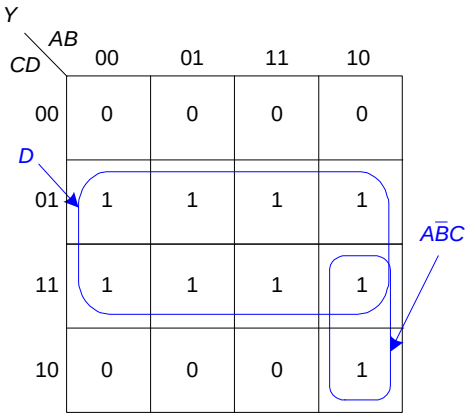
Exercise 2.23

B_2	B_1	B_0	$\overline{B_2 \bullet B_1 \bullet B_0}$	$\overline{B}_2 + \overline{B}_1 + \overline{B}_0$
0	0	0	1	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	0	0

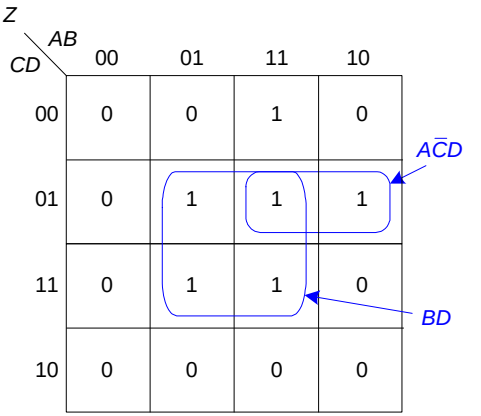
Exercise 2.24

$$Y = \overline{A}D + A\overline{B}C + A\overline{C}D + ABCD$$
$$Z = A\overline{C}D + BD$$

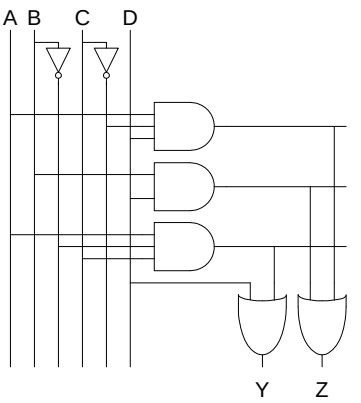
Exercise 2.25



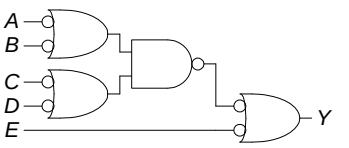
$Y = A\bar{B}C + D$



$Z = A\bar{C}D + BD$

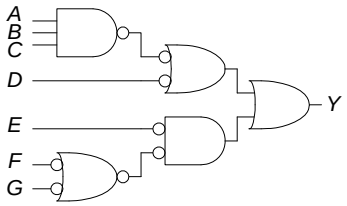


Exercise 2.26



$$Y = (\overline{A} + \overline{B})(\overline{C} + \overline{D}) + \overline{E}$$

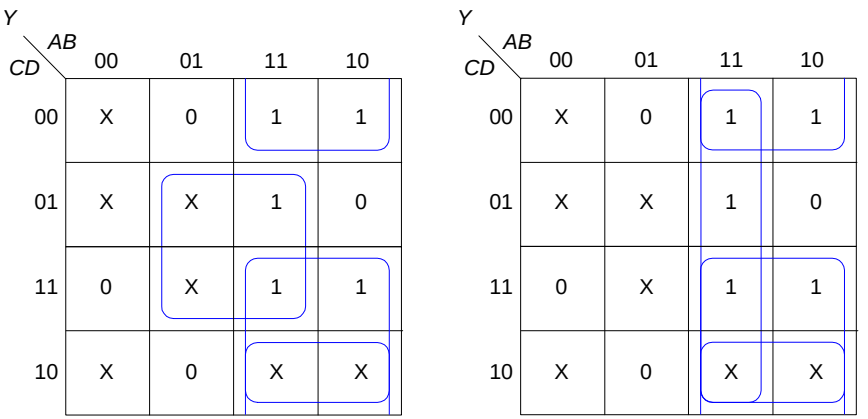
Exercise 2.27



$Y = ABC + \bar{D} + (\bar{F} + \bar{G})\bar{E}$
$= ABC + \bar{D} + \bar{E}\bar{F} + \bar{E}\bar{G}$

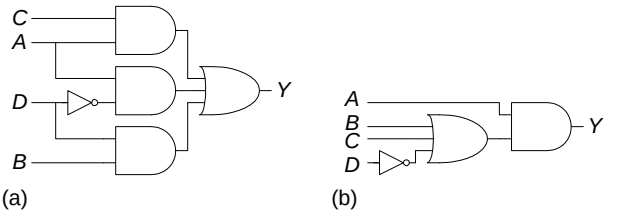
Exercise 2.28

Two possible options are shown below:



Exercise 2.29

Two possible options are shown below:



Exercise 2.30

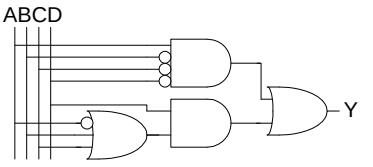
Option (a) could have a glitch when A=1, B=1, C=0, and D transitions from 1 to 0. The glitch could be removed by instead using the circuit in option (b).

Option (b) does not have a glitch. Only one path exists from any given input to the output.

Exercise 2.31

$$Y = \bar{A}D + A\bar{B}\bar{C}\bar{D} + BD + CD = A\bar{B}\bar{C}\bar{D} + D(\bar{A} + B + C)$$

Exercise 2.32



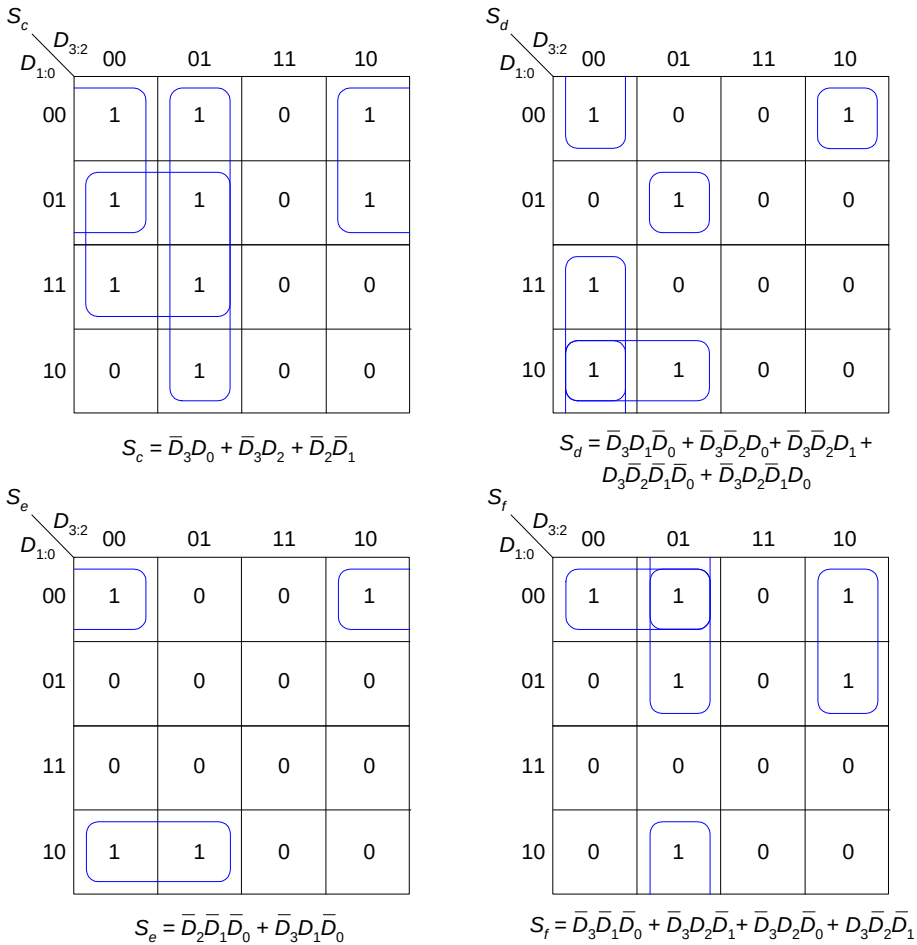
Exercise 2.33

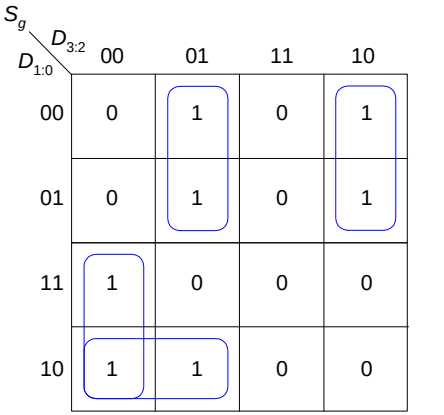
The equation can be written directly from the description:

$$E = S\bar{A} + AL + H$$

Exercise 2.34

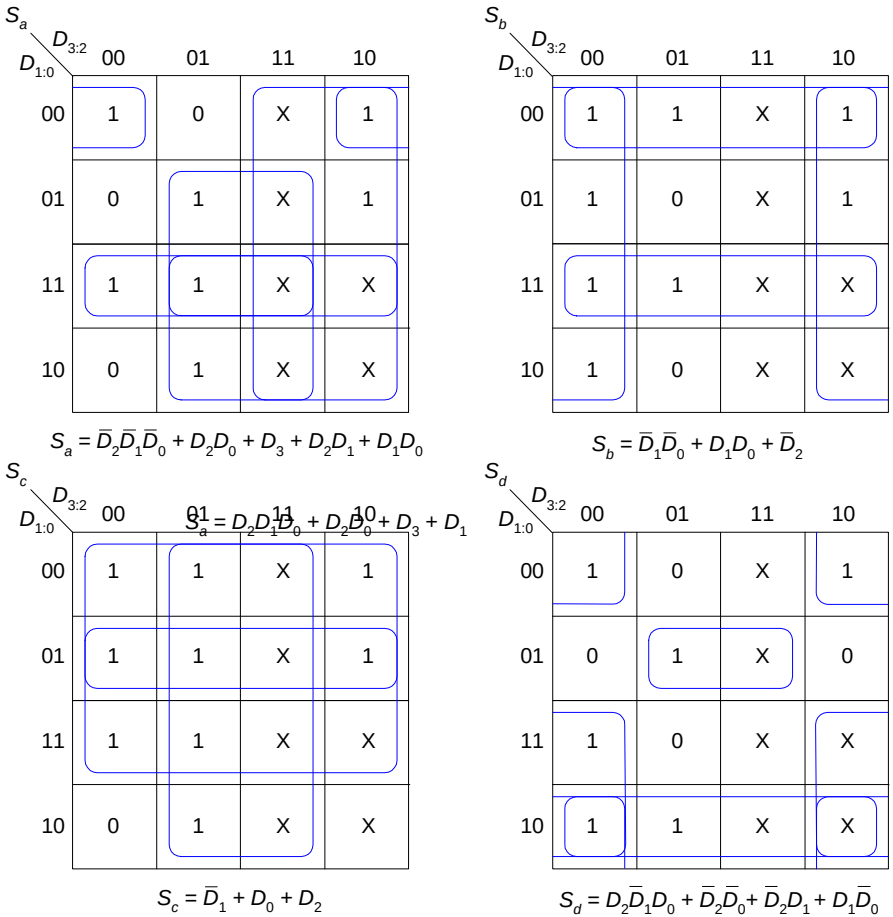
(a)



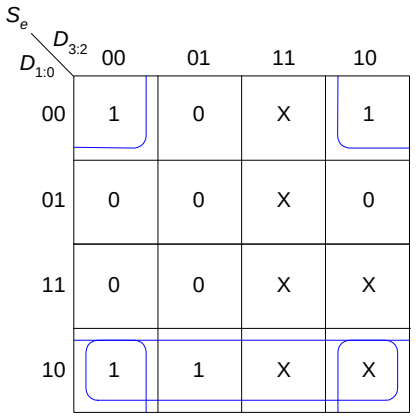


$$S_g = \bar{D}_3\bar{D}_2D_1 + \bar{D}_3D_1\bar{D}_0 + \bar{D}_3D_2\bar{D}_1 + D_3\bar{D}_2\bar{D}_1$$

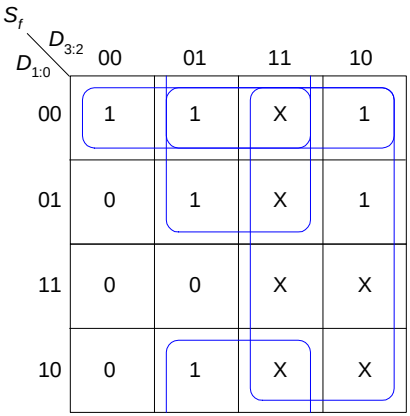
(b)



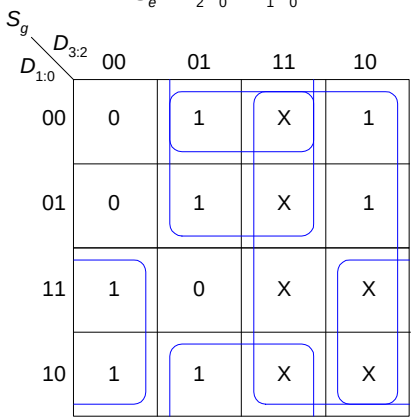
$S_a = D_2D_1D_0 + D_2D_0 + D_3 + D_1$



$S_e = \bar{D}_2\bar{D}_0 + D_1\bar{D}_0$

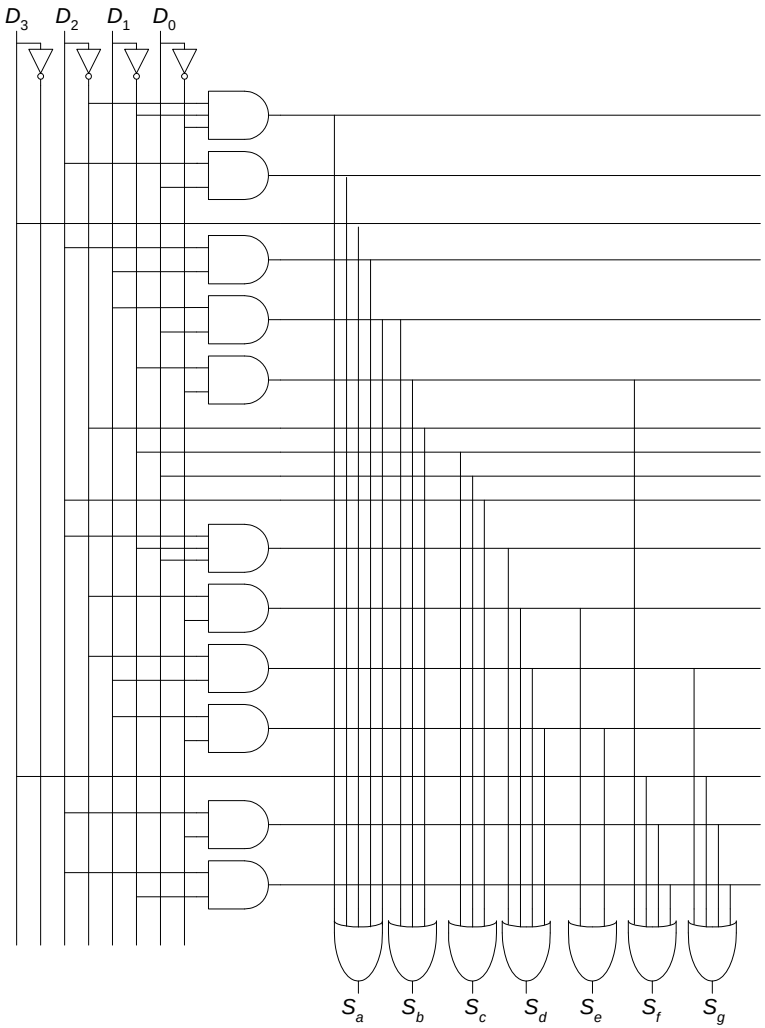


$S_f = \bar{D}_1\bar{D}_0 + D_2\bar{D}_1 + D_2\bar{D}_0 + D_3$



$S_g = \bar{D}_2D_1 + D_2\bar{D}_0 + D_2\bar{D}_1 + D_3$

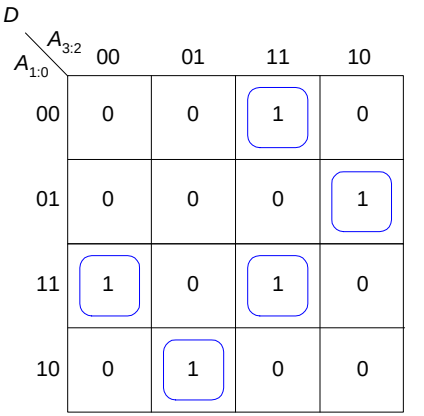
(c)



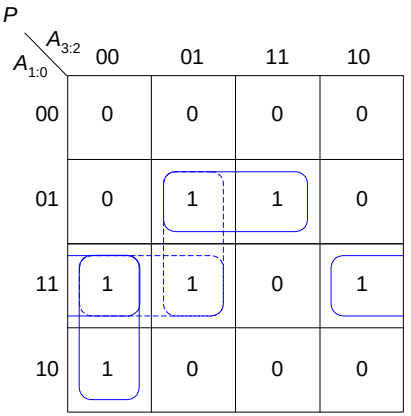
Exercise 2.35

Decimal Value	A_3	A_2	A_1	A_0	D	P
0	0	0	0	0	0	0
1	0	0	0	1	0	0
2	0	0	1	0	0	1
3	0	0	1	1	1	1
4	0	1	0	0	0	0
5	0	1	0	1	0	1
6	0	1	1	0	1	0
7	0	1	1	1	0	1
8	1	0	0	0	0	0
9	1	0	0	1	1	0
10	1	0	1	0	0	0
11	1	0	1	1	0	1
12	1	1	0	0	1	0
13	1	1	0	1	0	1
14	1	1	1	0	0	0
15	1	1	1	1	1	0

P has two possible minimal solutions:



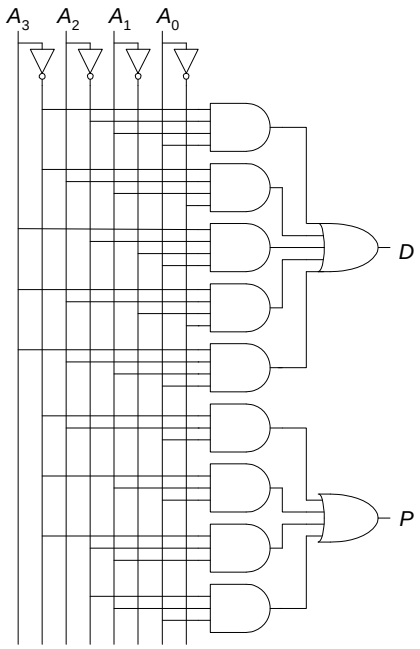
$$D = \bar{A}_3\bar{A}_2A_1A_0 + \bar{A}_3A_2A_1\bar{A}_0 + A_3\bar{A}_2\bar{A}_1A_0 + A_3A_2\bar{A}_1\bar{A}_0 + A_3A_2A_1A_0$$



$$P = \bar{A}_3A_2A_0 + \bar{A}_3A_1A_0 + \bar{A}_3\bar{A}_2A_1 + \bar{A}_2A_1A_0$$

$$P = \bar{A}_3A_1A_0 + \bar{A}_3\bar{A}_2A_1 + \bar{A}_2A_1A_0 + A_2\bar{A}_1A_0$$

Hardware implementations are below (implementing the first minimal equation given for P).



Exercise 2.36

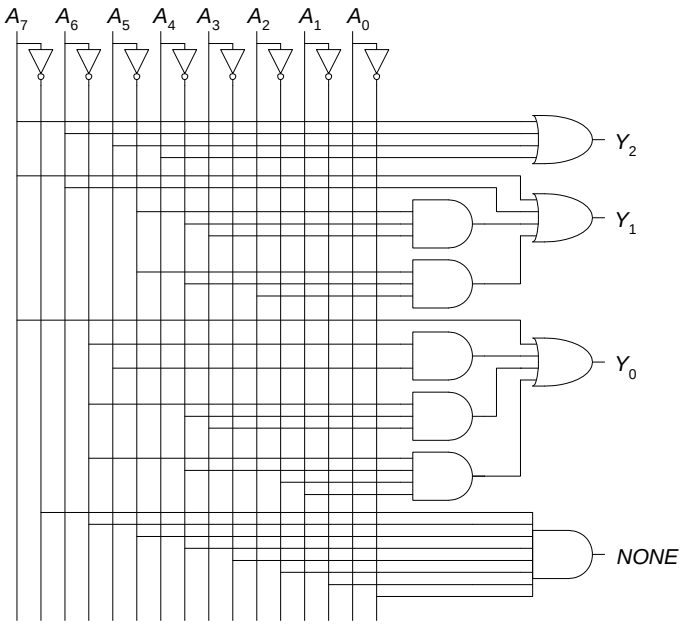
A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Y_2	Y_1	Y_0	*NONE*
0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	1	X	0	0	1	0
0	0	0	0	0	1	X	X	0	1	0	0
0	0	0	0	1	X	X	X	0	1	1	0
0	0	0	1	X	X	X	X	1	0	0	0
0	0	1	X	X	X	X	X	1	0	1	0
0	1	X	X	X	X	X	X	1	1	0	0
1	X	X	X	X	X	X	X	1	1	1	0

$$Y_2 = A_7 + A_6 + A_5 + A_4$$

$$Y_1 = A_7 + A_6 + \overline{A_5}\,\overline{A_4}A_3 + \overline{A_5}\,\overline{A_4}A_2$$

$$Y_0 = A_7 + \overline{A_6}A_5 + \overline{A_6}\,\overline{A_4}A_3 + \overline{A_6}\,\overline{A_4}\,\overline{A_2}A_1$$

$$NONE = \overline{A_7}\,\overline{A_6}\,\overline{A_5}\,\overline{A_4}\,\overline{A_3}\,\overline{A_2}\,\overline{A_1}\,\overline{A_0}$$



Exercise 2.37

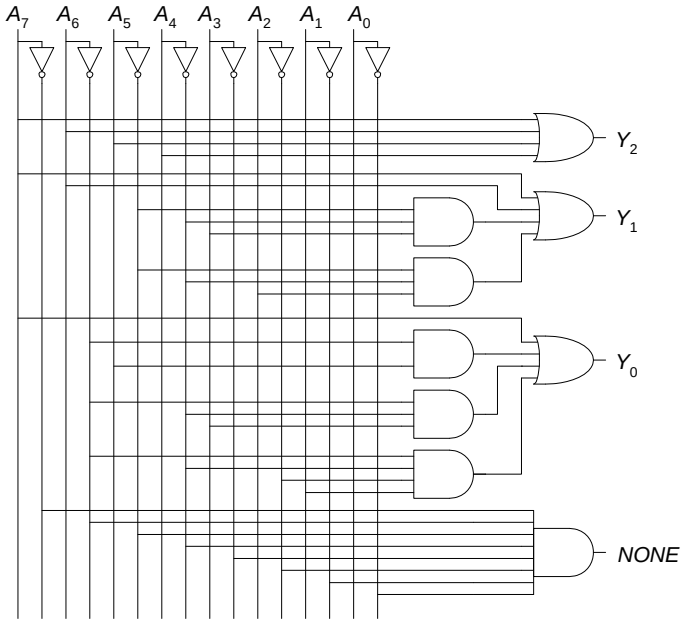
The equations and circuit for $Y_{2:0}$ is the same as in Exercise 2.25, repeated here for convenience.

A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	X	0	0	1
0	0	0	0	0	1	X	X	0	1	0
0	0	0	0	1	X	X	X	0	1	1
0	0	0	1	X	X	X	X	1	0	0
0	0	1	X	X	X	X	X	1	0	1
0	1	X	X	X	X	X	X	1	1	0
1	X	X	X	X	X	X	X	1	1	1

$$Y_2 = A_7 + A_6 + A_5 + A_4$$

$$Y_1 = A_7 + A_6 + \overline{A_5}\,\overline{A_4}A_3 + \overline{A_5}\,\overline{A_4}A_2$$

$$Y_0 = A_7 + \overline{A_6}A_5 + \overline{A_6}\,\overline{A_4}A_3 + \overline{A_6}\,\overline{A_4}\,\overline{A_2}A_1$$


A_7
A_6
A_5
A_4
A_3
A_2
A_1
A_0
Y_2
Y_1
Y_0
NONE

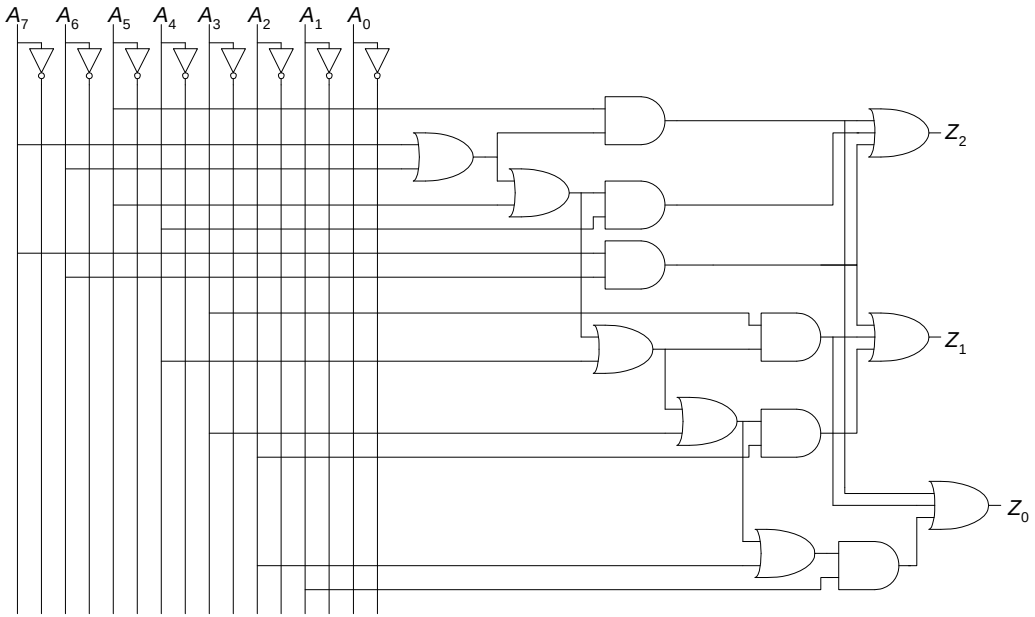
The truth table, equations, and circuit for $Z_{2:0}$ are as follows.

A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Z_2	Z_1	Z_0
0	0	0	0	0	0	1	1	0	0	0
0	0	0	0	0	1	0	1	0	0	0
0	0	0	0	1	0	0	1	0	0	0
0	0	0	1	0	0	0	1	0	0	0
0	0	1	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	1	1	X	0	0	1
0	0	0	0	1	0	1	X	0	0	1
0	0	0	1	0	0	1	X	0	0	1
0	0	1	0	0	0	1	X	0	0	1
0	1	0	0	0	0	1	X	0	0	1
1	0	0	0	0	0	1	X	0	0	1
0	0	0	0	1	1	X	X	0	1	0
0	0	0	1	0	1	X	X	0	1	0
0	0	1	0	0	1	X	X	0	1	0
0	1	0	0	0	1	X	X	0	1	0
1	0	0	0	0	1	X	X	0	1	0
0	0	0	1	1	X	X	X	0	1	1
0	0	1	0	1	X	X	X	0	1	1
0	1	0	0	1	X	X	X	0	1	1
1	0	0	0	1	X	X	X	0	1	1
0	0	1	1	X	X	X	X	1	0	0
0	1	0	1	X	X	X	X	1	0	0
1	0	0	1	X	X	X	X	1	0	0
0	1	1	X	X	X	X	X	1	0	1
1	0	1	X	X	X	X	X	1	0	1
1	1	X	X	X	X	X	X	1	1	0

$$Z_2 = A_4(A_5 + A_6 + A_7) + A_5(A_6 + A_7) + A_6A_7$$

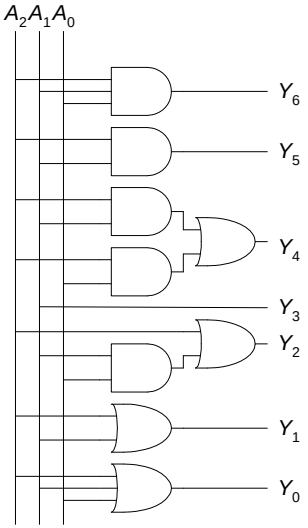
$$Z_1 = A_2(A_3 + A_4 + A_5 + A_6 + A_7) + A_3(A_4 + A_5 + A_6 + A_7) + A_6A_7$$

$$Z_0 = A_1(A_2 + A_3 + A_4 + A_5 + A_6 + A_7) + A_3(A_4 + A_5 + A_6 + A_7) + A_5(A_6 + A_7)$$



Exercise 2.38

$$Y_6 = A_2A_1A_0$$
$$Y_5 = A_2A_1$$
$$Y_4 = A_2A_1 + A_2A_0$$
$$Y_3 = A_2$$
$$Y_2 = A_2 + A_1A_0$$
$$Y_1 = A_2 + A_1$$
$$Y_0 = A_2 + A_1 + A_0$$



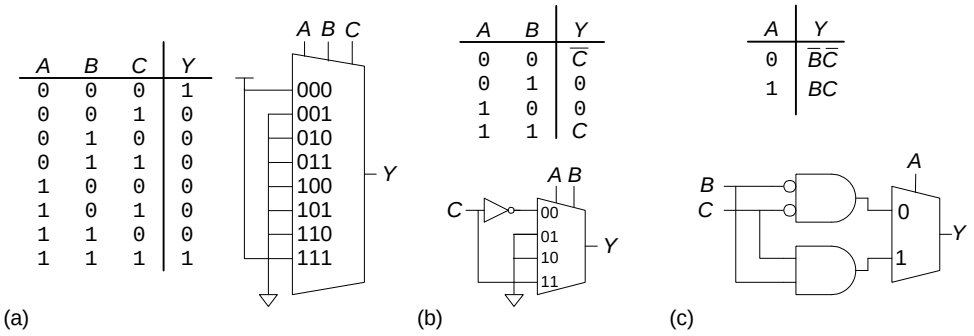
Exercise 2.39

$$Y = A + \overline{C \oplus D} = A + CD + \bar{C}\bar{D}$$

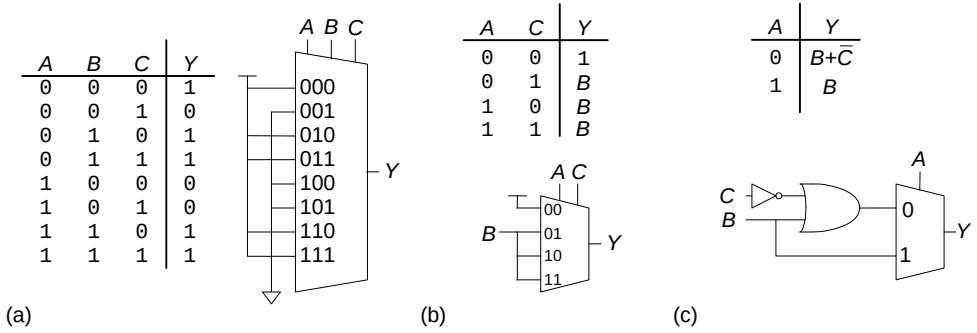
Exercise 2.40

$$Y = \bar{C}\bar{D}(A \oplus B) + \bar{A}\bar{B} = \bar{A}\bar{C}\bar{D} + \bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}$$

Exercise 2.41



Exercise 2.42



Exercise 2.43

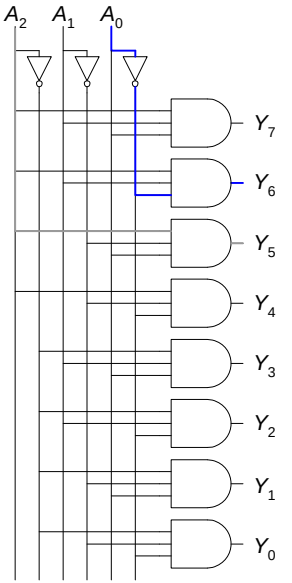
$t_{pd} = 3t_{pd\_NAND2} =$ **60 ps**
$t_{cd} = t_{cd\_NAND2} =$ **15 ps**

Exercise 2.44

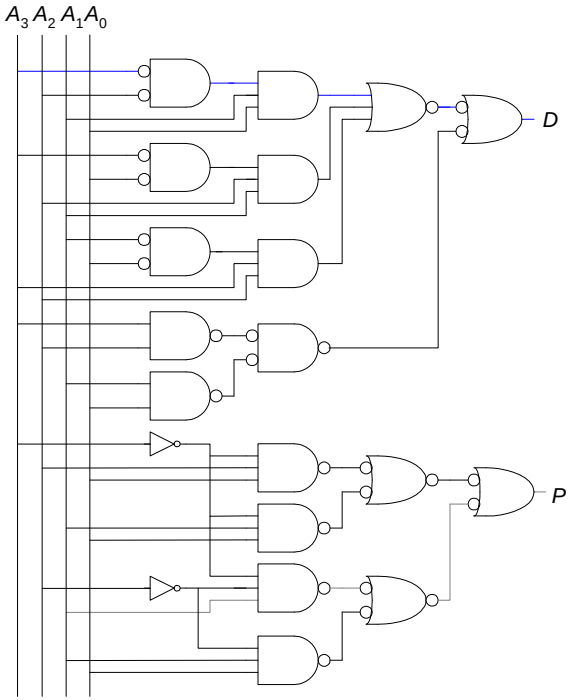
$t_{pd} = t_{pd\_AND2} + 2t_{pd\_NOR2} + t_{pd\_NAND2}$
$= [30 + 2\ (30) + 20]$ ps
$=$ **110 ps**
$t_{cd} = 2t_{cd\_NAND2} + t_{cd\_NOR2}$
$= [2\ (15) + 25]$ ps
$=$ **55 ps**

Exercise 2.45

$t_{pd} = t_{pd\_NOT} + t_{pd\_AND3}$
$= 15 \text{ ps} + 40 \text{ ps}$
$= \mathbf{55 \text{ ps}}$
$t_{cd} = t_{cd\_AND3}$
$= \mathbf{30 \text{ ps}}$

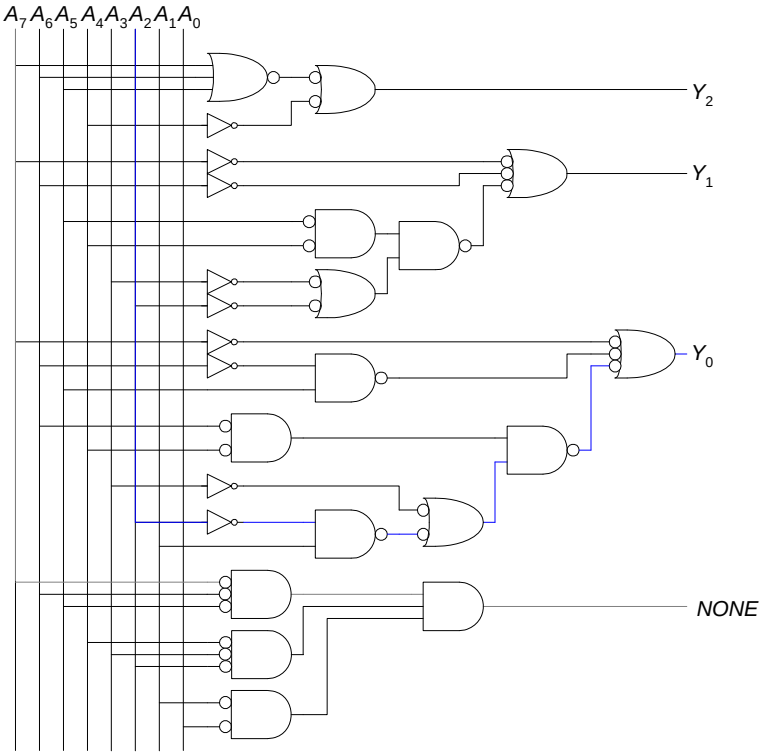


Exercise 2.46



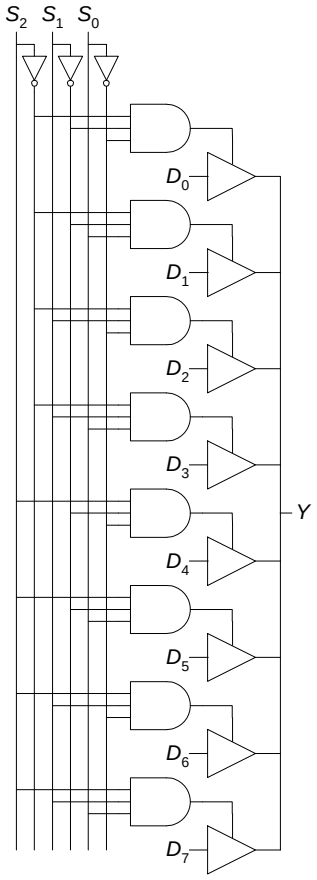
$t_{pd} = t_{pd\_NOR2} + t_{pd\_AND3} + t_{pd\_NOR3} + t_{pd\_NAND2}$
$= [30 + 40 + 45 + 20]$ ps
$=$ **135 ps**
$t_{cd} = 2t_{cd\_NAND2} + t_{cd\_OR2}$
$= [2\ (15) + 30]$ ps
$=$ **60 ps**

Exercise 2.47



$t_{pd} = t_{pd\_INV} + 3t_{pd\_NAND2} + t_{pd\_NAND3}$
$= [15 + 3\ (20) + 30]$ ps
$=$ **105 ps**
$t_{cd} = t_{cd\_NOT} + t_{cd\_NAND2}$
$= [10 + 15]$ ps
$=$ **25 ps**

Exercise 2.48



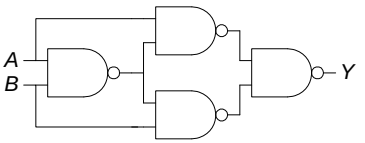
$t_{pd\_dy} = t_{pd\_TRI\_AY}$
= **50 ps**

Note: the propagation delay from the control (select) input to the output is the circuit's critical path:

$t_{pd\_sy} = t_{pd\_NOT} + t_{pd\_AND3} + t_{pd\_TRI\_SY}$
= [30 + 80 + 35] ps
= **145 ps**

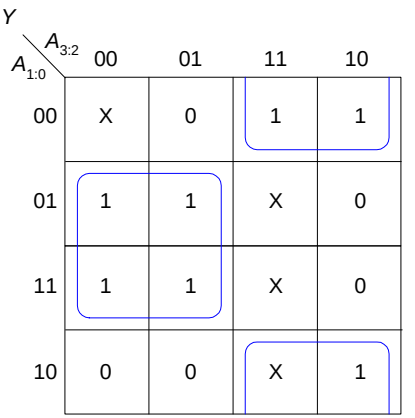
However, the problem specified to minimize the delay from data inputs to output, t_{pd_dy}.

Question 2.1



Question 2.2

Month	A_3	A_2	A_1	A_0	Y
Jan	0	0	0	1	1
Feb	0	0	1	0	0
Mar	0	0	1	1	1
Apr	0	1	0	0	0
May	0	1	0	1	1
Jun	0	1	1	0	0
Jul	0	1	1	1	1
Aug	1	0	0	0	1
Sep	1	0	0	1	0
Oct	1	0	1	0	1
Nov	1	0	1	1	0
Dec	1	1	0	0	1



$$Y = \bar{A}_3 A_0 + A_3 \bar{A}_0 = A_3 \oplus A_0$$

Question 2.3

A tristate buffer has two inputs and three possible outputs: 0, 1, and Z. One of the inputs is the data input and the other input is a control input, often called the *enable* input. When the enable input is 1, the tristate buffer transfers the data input to the output; otherwise, the output is high impedance, Z. Tristate buffers are used when multiple sources drive a single output at different times. One and only one tristate buffer is enabled at any given time.

Question 2.4

(a) An AND gate is not universal, because it cannot perform inversion (NOT).

(b) The set {OR, NOT} is universal. It can construct any Boolean function. For example, an OR gate with NOT gates on all of its inputs and output performs the AND operation. Thus, the set {OR, NOT} is equivalent to the set {AND, OR, NOT} and is universal.

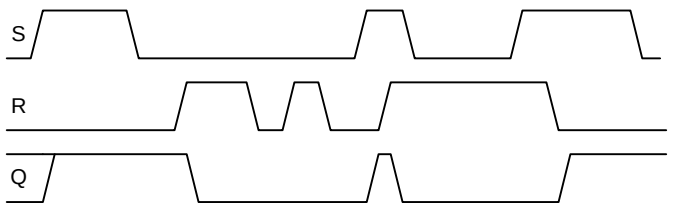
(c) The NAND gate by itself is universal. A NAND gate with its inputs tied together performs the NOT operation. A NAND gate with a NOT gate on its output performs AND. And a NAND gate with NOT gates on its inputs performs OR. Thus, a NAND gate is equivalent to the set {AND, OR, NOT} and is universal.

Question 2.5

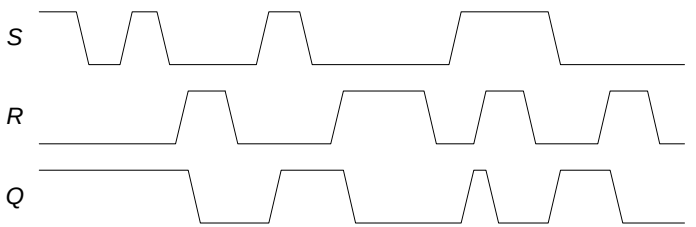
A circuit's contamination delay might be less than its propagation delay because the circuit may operate over a range of temperatures and supply voltages, for example, 3-3.6 V for LVCMOS (low voltage CMOS) chips. As temperature increases and voltage decreases, circuit delay increases. Also, the circuit may have different paths (critical and short paths) from the input to the output. A gate itself may have varying delays between different inputs and the output, affecting the gate's critical and short paths. For example, for a two-input NAND gate, a HIGH to LOW transition requires two nMOS transistor delays, whereas a LOW to HIGH transition requires a single pMOS transistor delay.

CHAPTER 3

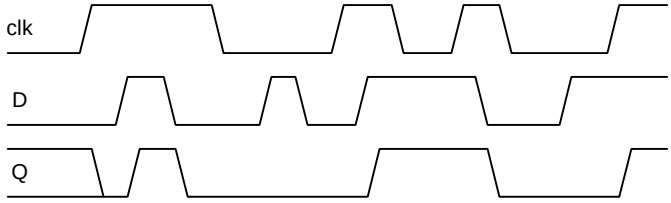
Exercise 3.1



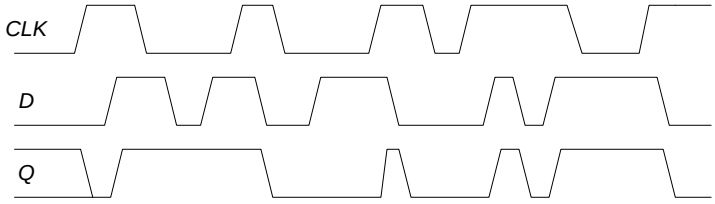
Exercise 3.2



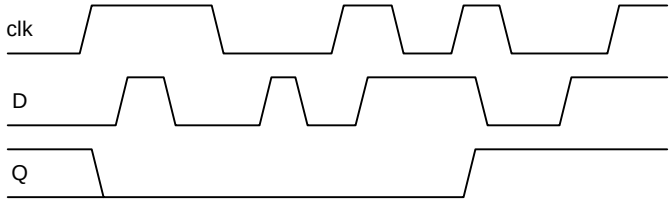
Exercise 3.3



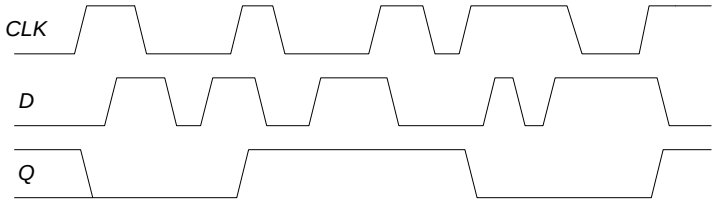
Exercise 3.4



Exercise 3.5



Exercise 3.6



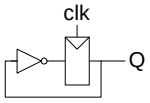
Exercise 3.7

The circuit is sequential because it involves feedback and the output depends on previous values of the inputs. This is a SR latch. When $\overline{S}$ = 0 and $\overline{R}$ = 1, the circuit sets Q to 1. When $\overline{S}$ = 1 and $\overline{R}$ = 0, the circuit resets Q to 0. When both $\overline{S}$ and $\overline{R}$ are 1, the circuit remembers the old value. And when both $\overline{S}$ and $\overline{R}$ are 0, the circuit drives both outputs to 1.

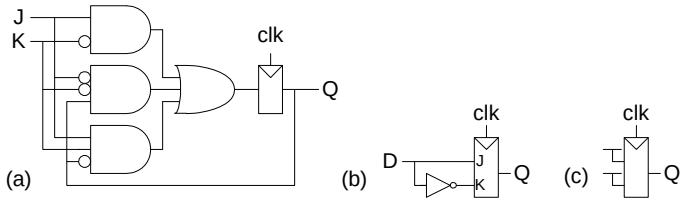
Exercise 3.8

Sequential logic. This is a D flip-flop with active low asynchronous set and reset inputs. If $\overline{S}$ and $\overline{R}$ are both 1, the circuit behaves as an ordinary D flip-flop. If $\overline{S}$ = 0, Q is immediately set to 0. If $\overline{R}$ = 0, Q is immediately reset to 1. (This circuit is used in the commercial 7474 flip-flop.)

Exercise 3.9



Exercise 3.10

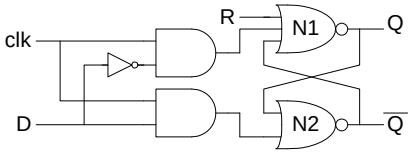


Exercise 3.11

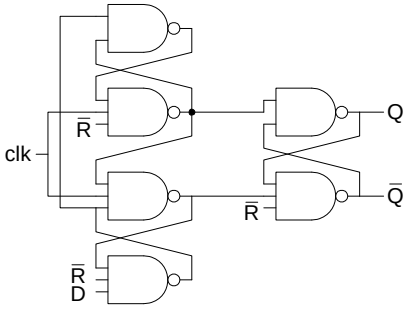
If A and B have the same value, C takes on that value. Otherwise, C retains its old value.

Exercise 3.12

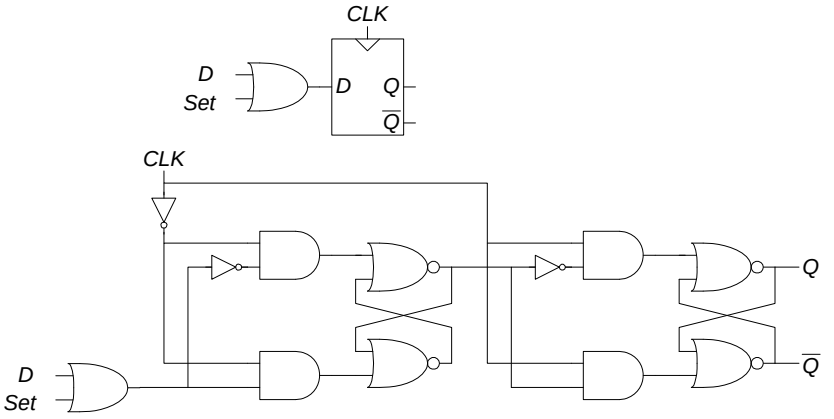
Make sure these next ones are correct too.



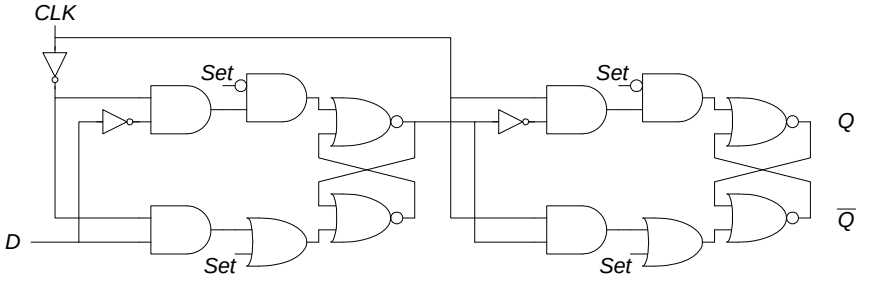
Exercise 3.13



Exercise 3.14



Exercise 3.15



Exercise 3.16

From $\frac{1}{2Nt_{pd}}$ to $\frac{1}{2Nt_{cd}}$.

Exercise 3.17

If N is even, the circuit is stable and will not oscillate.

Exercise 3.18

(a) No: no register. (b) No: feedback without passing through a register. (c) Yes. Satisfies the definition. (d) Yes. Satisfies the definition.

Exercise 3.19

The system has at least five bits of state to represent the 24 floors that the elevator might be on.

Exercise 3.20

The FSM has $5^4 = 625$ states. This requires at least 10 bits to represent all the states.

Exercise 3.21

The FSM could be factored into four independent state machines, one for each student. Each of these machines has five states and requires 3 bits, so at least 12 bits of state are required for the factored design.

Exercise 3.22

This finite state machine asserts the output Q for one clock cycle if A is TRUE followed by B being TRUE.

state	encoding $S_{1:0}$
S0	00
S1	01
S2	10

TABLE 3.1 State encoding for Exercise 3.22

.

current state		inputs		next state	
s_1	s_0	a	b	s'_1	s'_0
0	0	0	X	0	0
0	0	1	X	0	1

TABLE 3.2 State transition table with binary encodings for Exercise 3.22

current state		inputs		next state	
s_1	s_0	a	b	s'_1	s'_0
0	1	X	0	0	0
0	1	X	1	1	0
1	0	X	X	0	0

TABLE 3.2 State transition table with binary encodings for Exercise 3.22

.

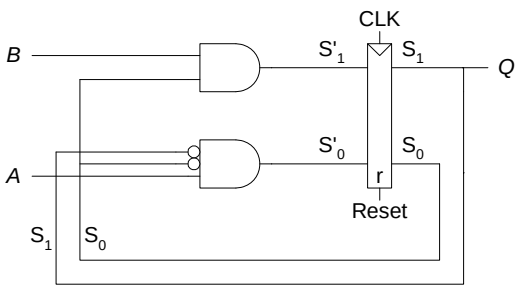
current state		output
s_1	s_0	q
0	0	0
0	1	0
1	0	1

TABLE 3.3 Output table with binary encodings for Exercise 3.22

$$S'_1 = S_0 B$$

$$S'_0 = \overline{S_1}\,\overline{S_0} A$$

$$Q = S_1$$



Exercise 3.23

This finite state machine asserts the output Q when A AND B is TRUE.

state	encoding $s_{1:0}$
S0	00
S1	01
S2	10

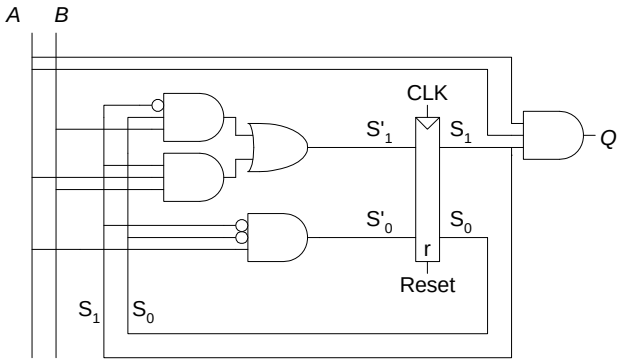
TABLE 3.4 State encoding for Exercise 3.23

current state		inputs		next state		output
s_1	s_0	a	b	s'_1	s'_0	q
0	0	0	X	0	0	0
0	0	1	X	0	1	0
0	1	X	0	0	0	0
0	1	X	1	1	0	0
1	0	1	1	1	0	1
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0

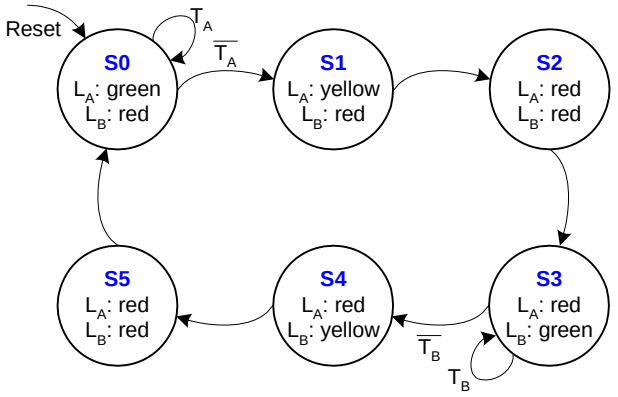
TABLE 3.5 Combined state transition and output table with binary encodings for Exercise 3.23

$$S'_1 = \overline{S}_1 S_0 B + S_1 A B$$
$$S'_0 = \overline{S}_1 \overline{S}_0 A$$

$$Q' = S_1 A B$$



Exercise 3.24



state	encoding $S_{1:0}$
S0	000
S1	001
S2	010

TABLE 3.6 State encoding for Exercise 3.24

state	encoding $s_{1:0}$
S3	100
S4	101
S5	110

TABLE 3.6 State encoding for Exercise 3.24

current state			inputs		next state		
s_2	s_1	s_0	t_a	t_b	s'_2	s'_1	s'_0
0	0	0	0	X	0	0	1
0	0	0	1	X	0	0	0
0	0	1	X	X	0	1	0
0	1	0	X	X	1	0	0
1	0	0	X	0	1	0	1
1	0	0	X	1	1	0	0
1	0	1	X	X	1	1	0
1	1	0	X	X	0	0	0

TABLE 3.7 State transition table with binary encodings for Exercise 3.24

$$S'_2 = S_2 \oplus S_1$$
$$S'_1 = \overline{S_1} S_0$$
$$S'_0 = \overline{S_1}\,\overline{S_0}(\overline{S_2}\,\overline{t_a} + S_2\,\overline{t_b})$$

current state			outputs			
s_2	s_1	s_0	l_{a1}	l_{a0}	l_{b1}	l_{b0}
0	0	0	0	0	1	0
0	0	1	0	1	1	0
0	1	0	1	0	1	0
1	0	0	1	0	0	0
1	0	1	1	0	0	1
1	1	0	1	0	1	0

TABLE 3.8 Output table for Exercise 3.24

$$
\begin{aligned}
L_{A1} &= S_1\overline{S_0} + S_2\overline{S_1} \\
L_{A0} &= \overline{S_2}S_0 \\
L_{B1} &= \overline{S_2}\,\overline{S_1} + S_1\overline{S_0} \\
L_{B0} &= S_2\overline{S_1}S_0
\end{aligned}
\tag{3.1}
$$

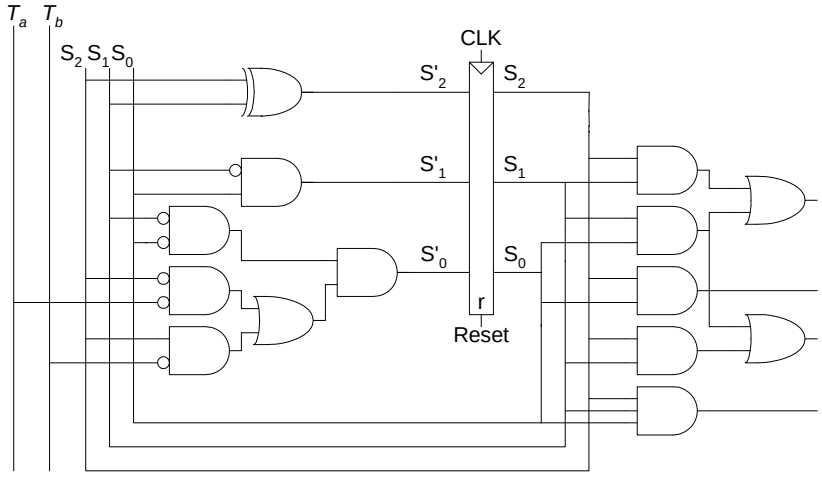
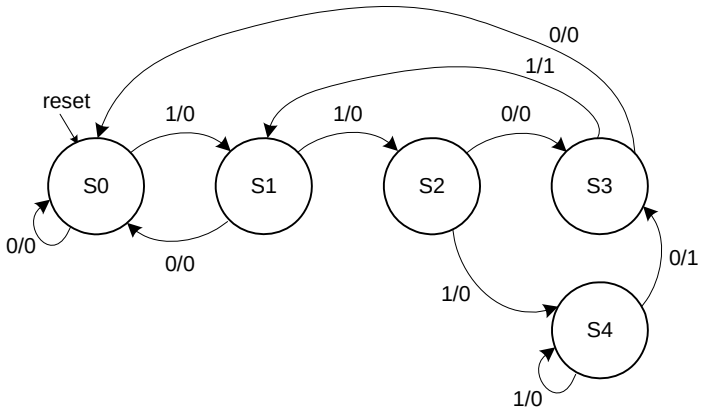


FIGURE 3.1 State machine circuit for traffic light controller for Exercise 3.21

Exercise 3.25



state	encoding $s_{1:0}$
S0	000
S1	001
S2	010
S3	100
S4	101

TABLE 3.9 State encoding for Exercise 3.25

current state			input	next state			output
s_2	s_1	s_0	a	s'_2	s'_1	s'_0	q
0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	0

TABLE 3.10 Combined state transition and output table with binary encodings for Exercise 3.25

current state			input	next state			output
s_2	s_1	s_0	a	s'_2	s'_1	s'_0	q
0	0	1	0	0	0	0	0
0	0	1	1	0	1	0	0
0	1	0	0	1	0	0	0
0	1	0	1	1	0	1	0
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	1
1	0	1	0	1	0	0	1
1	0	1	1	1	0	1	0

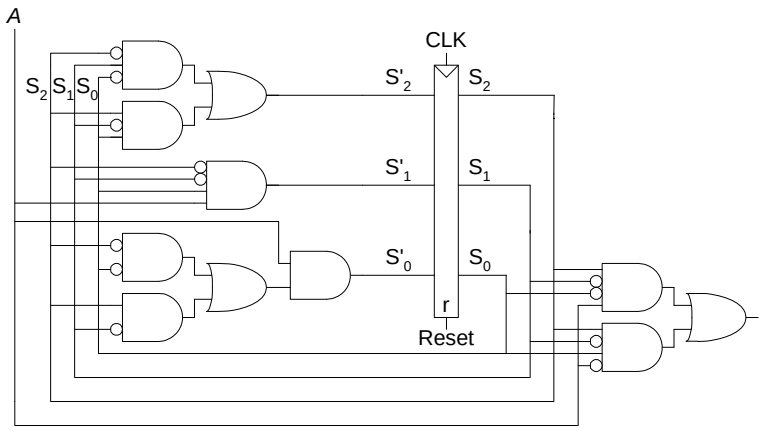
TABLE 3.10 Combined state transition and output table with binary encodings for Exercise 3.25

$$S'_2 = \overline{S_2}S_1\overline{S_0} + S_2\overline{S_1}S_0$$

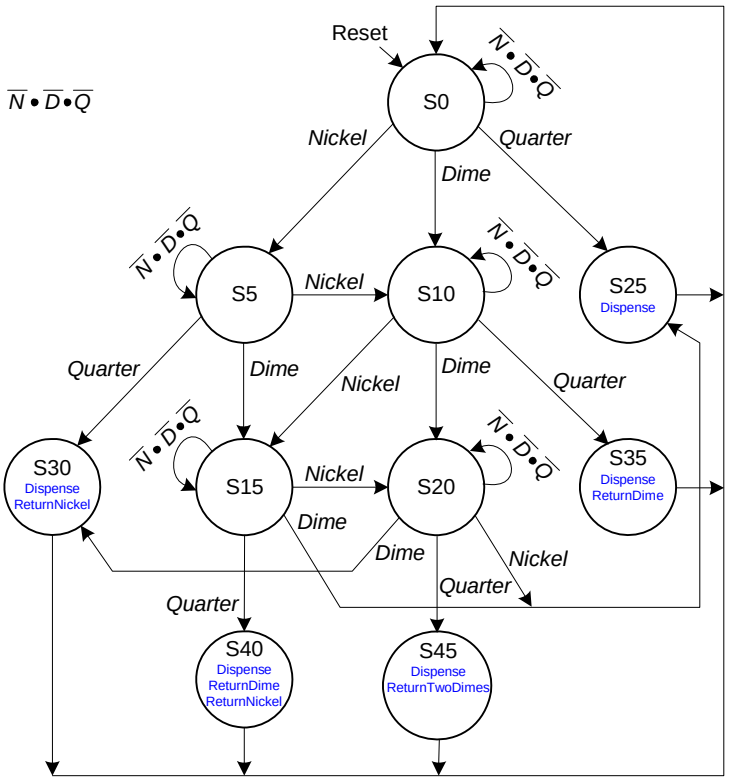
$$S'_1 = \overline{S_2}\,\overline{S_1}S_0A$$

$$S'_0 = A(\overline{S_2}\,\overline{S_0} + S_2\overline{S_1})$$

$$Q = S_2\overline{S_1}\,\overline{S_0}A + S_2\overline{S_1}S_0\overline{A}$$



Exercise 3.26



Note: $\overline{N} \bullet \overline{D} \bullet \overline{Q} = \overline{Nickel} \bullet \overline{Dime} \bullet \overline{Quarter}$

FIGURE 3.2 State transition diagram for soda machine dispense of Exercise 3.23

state	encoding $s_{9:0}$
S0	0000000001
S5	0000000010
S10	0000000100
S25	0000001000
S30	0000010000
S15	0000100000
S20	0001000000
S35	0010000000
S40	0100000000
S45	1000000000

FIGURE 3.3 State Encodings for Exercise 3.26

current state s	inputs			next state s'
	nickel	*dime*	*quarter*	
S0	0	0	0	S0
S0	0	0	1	S25
S0	0	1	0	S10
S0	1	0	0	S5
S5	0	0	0	S5
S5	0	0	1	S30
S5	0	1	0	S15
S5	1	0	0	S10
S10	0	0	0	S10

TABLE 3.11 State transition table for Exercise 3.26

current state *s*	inputs			next state *s'*
	nickel	*dime*	*quarter*	
S10	0	0	1	S35
S10	0	1	0	S20
S10	1	0	0	S15
S25	X	X	X	S0
S30	X	X	X	S0
S15	0	0	0	S15
S15	0	0	1	S40
S15	0	1	0	S25
S15	1	0	0	S20
S20	0	0	0	S20
S20	0	0	1	S45
S20	0	1	0	S30
S20	1	0	0	S25
S35	X	X	X	S0
S40	X	X	X	S0
S45	X	X	X	S0

TABLE 3.11 State transition table for Exercise 3.26

current state *s*	inputs			next state *s'*
	nickel	*dime*	*quarter*	
0000000001	0	0	0	0000000001
0000000001	0	0	1	0000001000
0000000001	0	1	0	0000000100
0000000001	1	0	0	0000000010

TABLE 3.12 State transition table for Exercise 3.26

current state s	inputs			next state s'
	nickel	*dime*	*quarter*	
0000000010	0	0	0	0000000010
0000000010	0	0	1	0000010000
0000000010	0	1	0	0000100000
0000000010	1	0	0	0000000100
0000000100	0	0	0	0000000100
0000000100	0	0	1	0010000000
0000000100	0	1	0	0001000000
0000000100	1	0	0	0000100000
0000001000	X	X	X	0000000001
0000010000	X	X	X	0000000001
0000100000	0	0	0	0000100000
0000100000	0	0	1	0100000000
0000100000	0	1	0	0000001000
0000100000	1	0	0	0001000000
0001000000	0	0	0	0001000000
0001000000	0	0	1	1000000000
0001000000	0	1	0	0000010000
0001000000	1	0	0	0000001000
0010000000	X	X	X	0000000001
0100000000	X	X	X	0000000001
1000000000	X	X	X	0000000001

TABLE 3.12 State transition table for Exercise 3.26

$$S'_9 = S_6 Q$$

$$S'_8 = S_5 Q$$

$$S'_7 = S_2 Q$$

$$S'_6 = S_2 D + S_5 N + S_6 \overline{N}\,\overline{D}\,\overline{Q}$$

$$S'_5 = S_1 D + S_2 N + S_5 NDQ$$

$$S'_4 = S_1 Q + S_6 D$$

$$S'_3 = S_0 Q + S_5 D + S_6 N$$

$$S'_2 = S_0 D + S_1 N + S_2 \overline{N}\,\overline{D}\,\overline{Q}$$

$$S'_1 = S_0 N + S_1 NDQ$$

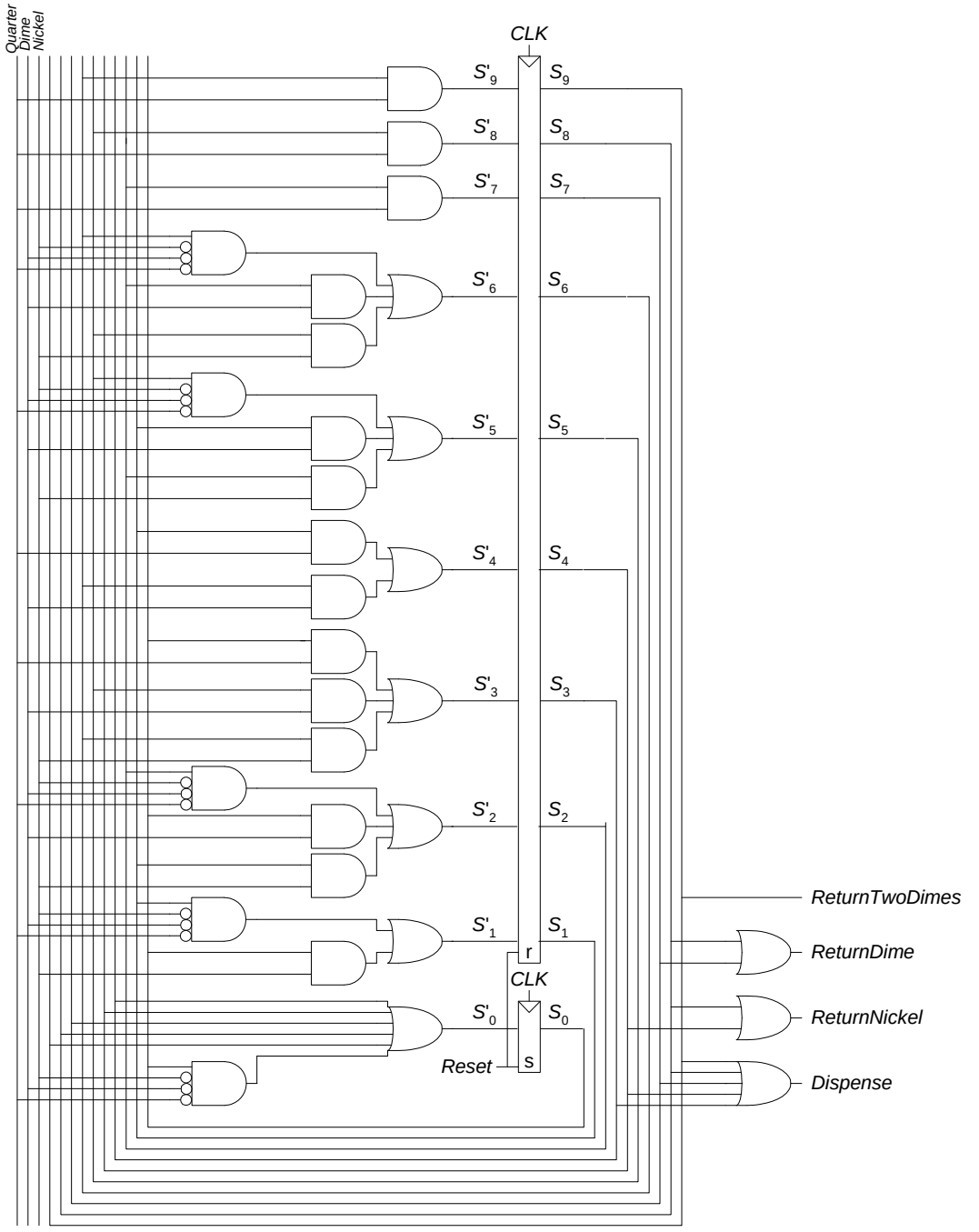
$$S'_0 = S_0 \overline{N}\,\overline{D}\,\overline{Q} + S_3 + S_4 + S_7 + S_8 + S_9$$

$$Dispense = S_3 + S_4 + S_7 + S_8 + S_9$$

$$ReturnNickel = S_4 + S_8$$

$$ReturnDime = S_7 + S_8$$

$$ReturnTwoDimes = S_9$$


Quarter
Dime
Nickel
CLK
S'9
S9
S'8
S8
S'7
S7
S'6
S6
S'5
S5
S'4
S4
S'3
S3
S'2
S2
S'1
S1
r
CLK
S'0
S0
Reset
s
ReturnTwoDimes
ReturnDime
ReturnNickel
Dispense

Exercise 3.27

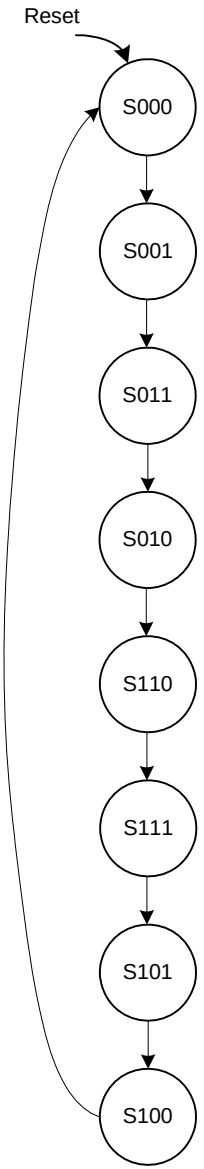


FIGURE 3.4 State transition diagram for Exercise 3.27

current state $s_{2:0}$	next state $s'_{2:0}$
000	001
001	011
011	010
010	110
110	111
111	101
101	100
100	000

TABLE 3.13 State transition table for Exercise 3.27

$$S'_2 = S_1\overline{S_0} + S_2S_0$$
$$S'_1 = \overline{S_2}S_0 + S_1\overline{S_0}$$
$$S'_0 = \overline{S_2 \oplus S_1}$$
$$Q_2 = S_2$$
$$Q_1 = S_1$$
$$Q_0 = S_0$$

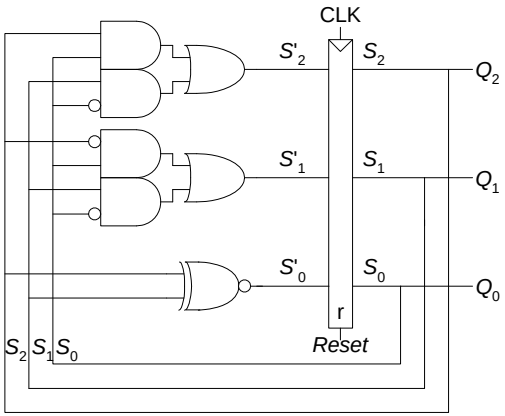


FIGURE 3.5 Hardware for Gray code counter FSM for Exercise 3.27

Exercise 3.28

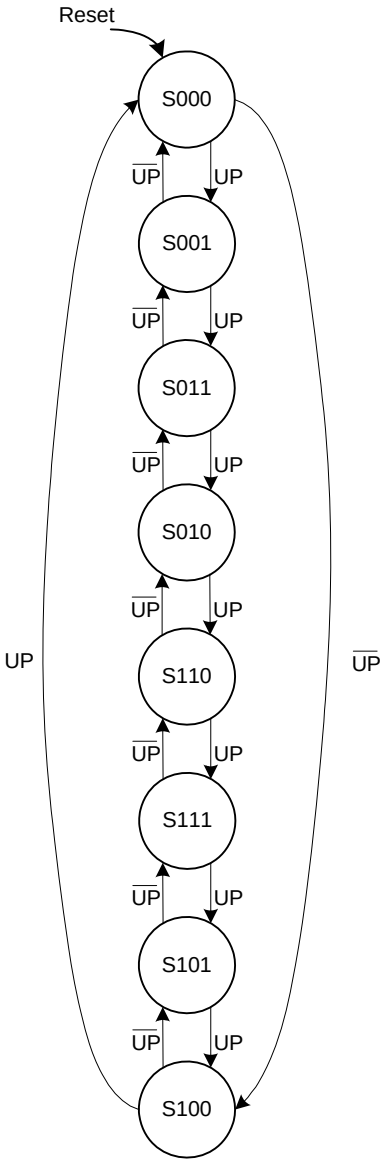


FIGURE 3.6 State transition diagram for Exercise 3.28

current state $S_{2:0}$	input up	next state $S'_{2:0}$
000	1	001
001	1	011
011	1	010
010	1	110
110	1	111
111	1	101
101	1	100
100	1	000
000	0	100
001	0	000
011	0	001
010	0	011
110	0	010
111	0	110
101	0	111
100	0	101

TABLE 3.14 State transition table for Exercise 3.28

$$S'_2 = UPS_1\overline{S_0} + \overline{UP}\,\overline{S_1}\,\overline{S_0} + S_2S_0$$

$$S'_1 = S_1\overline{S_0} + UP\overline{S_2}S_0 + \overline{UP}S_2S_1$$

$$S'_0 = UP \oplus S_2 \oplus S_1$$

$$Q_2 = S_2$$

$$Q_1 = S_1$$

$$Q_0 = S_0$$

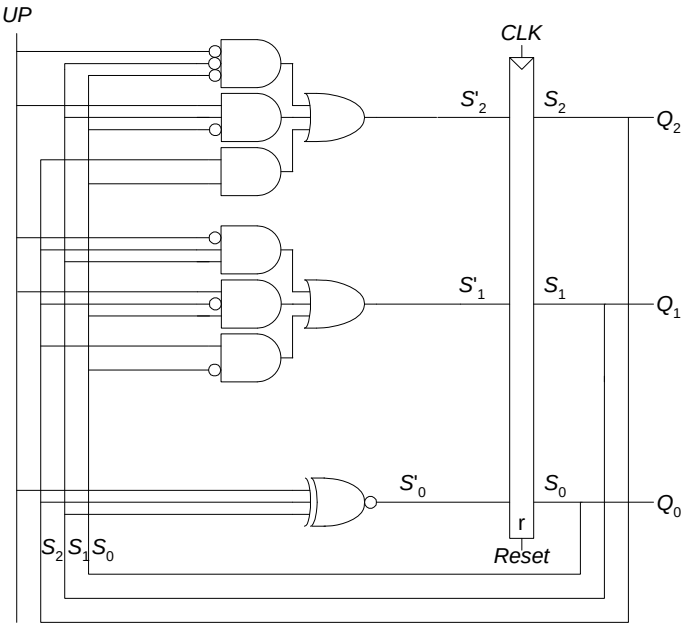


FIGURE 3.7 Finite state machine hardware for Exercise 3.28

Exercise 3.29

(a)

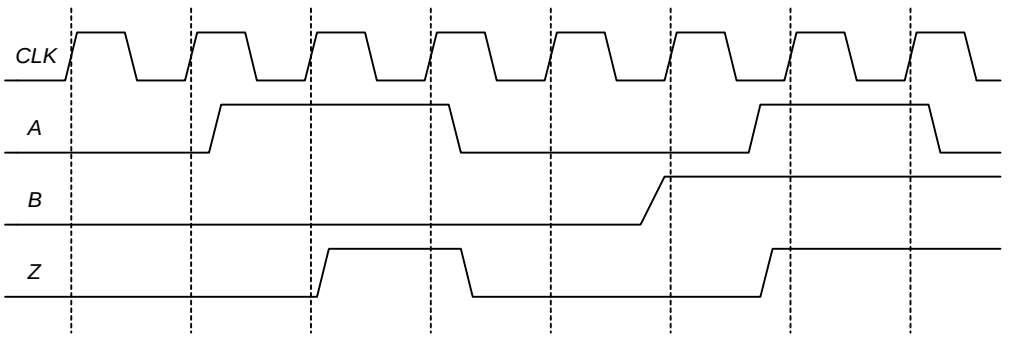


FIGURE 3.8 Waveform showing *Z* output for Exercise 3.29

(b) This FSM is a Mealy FSM because the output depends on the current value of the input as well as the current state.

(c)

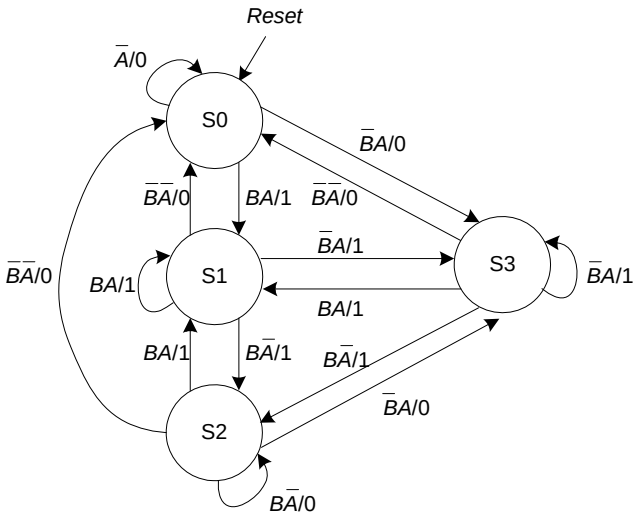


FIGURE 3.9 State transition diagram for Exercise 3.29

(Note: another viable solution would be to allow the state to transition from S0 to S1 on $B\bar{A}/0$. The arrow from S0 to S0 would then be $\bar{B}\bar{A}/0$.)

current state $s_{1:0}$	inputs		next state $s'_{1:0}$	output
	b	a		z
00	X	0	00	0
00	0	1	11	0
00	1	1	01	1
01	0	0	00	0
01	0	1	11	1
01	1	0	10	1
01	1	1	01	1
10	0	X	00	0
10	1	0	10	0

TABLE 3.15 State transition table for Exercise 3.29

current state $s_{1:0}$	inputs		next state $s'_{1:0}$	output
	b	*a*		z
10	1	1	01	1
11	0	0	00	0
11	0	1	11	1
11	1	0	10	1
11	1	1	01	1

TABLE 3.15 State transition table for Exercise 3.29

$$S'_1 = \bar{B}A(\overline{S_1} + S_0) + B\bar{A}(S_1 + \overline{S_0})$$

$$S'_0 = A(\overline{S_1} + S_0 + B)$$

$$Z = BA + S_0(A + B)$$

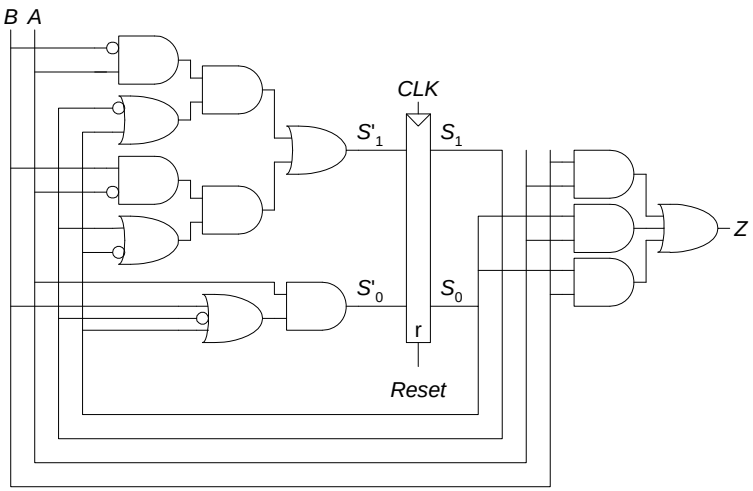


FIGURE 3.10 Hardware for FSM of Exercise 3.26

Note: One could also build this functionality by registering input *A*, producing both the logical AND and OR of input *A* and its previous (registered)

value, and then muxing the two operations using B. The output of the mux is Z: $Z = AA\text{prev}$ (if $B = 0$); $Z = A + A\text{prev}$ (if $B = 1$).

Exercise 3.30

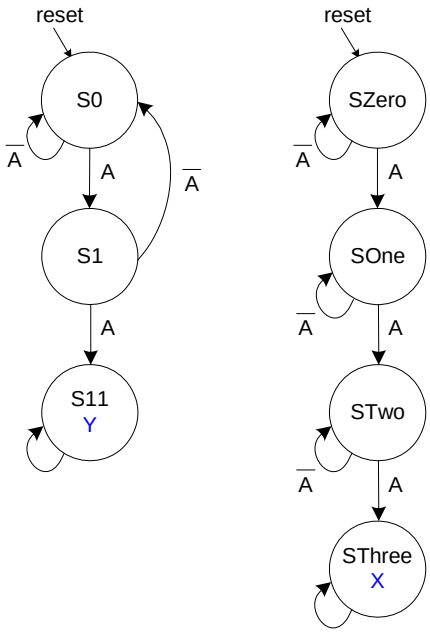


FIGURE 3.11 Factored state transition diagram for Exercise 3.30

current state $s_{1:0}$	input a	next state $s'_{1:0}$
00	0	00
00	1	01
01	0	00

TABLE 3.16 State transition table for output Y for Exercise 3.30

current state $s_{1:0}$	input a	next state $s'_{1:0}$
01	1	11
11	X	11

TABLE 3.16 State transition table for output *Y* for Exercise 3.30

current state $t_{1:0}$	input a	next state $t'_{1:0}$
00	0	00
00	1	01
01	0	01
01	1	10
10	0	10
10	1	11
11	X	11

TABLE 3.17 State transition table for output *X* for Exercise 3.30

$$S'_1 = S_0(S_1 + A)$$
$$S'_0 = \overline{S_1}A + S_0(S_1 + A)$$
$$Y = S_1$$

$$T'_1 = T_1 + T_0A$$
$$T'_0 = A(T_1 + \overline{T_0}) + \overline{A}T_0 + T_1T_0$$
$$X = T_1T_0$$

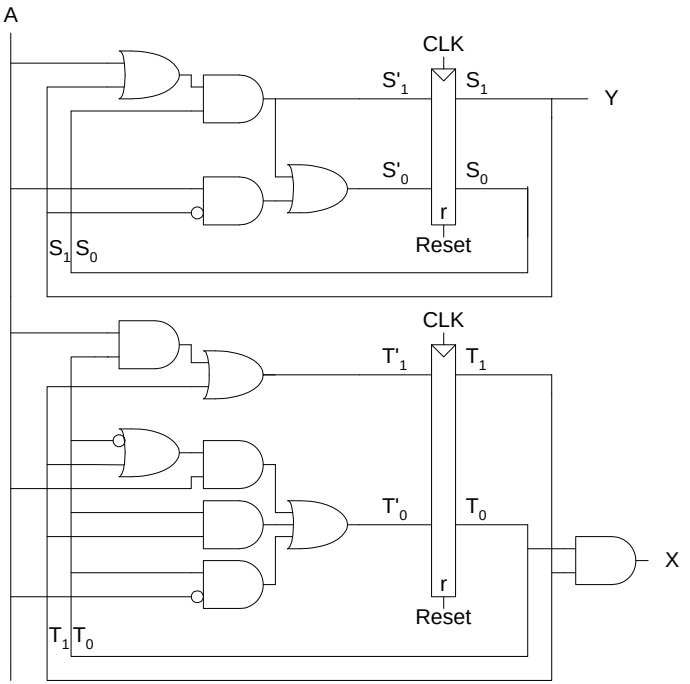


FIGURE 3.12 Finite state machine hardware for Exercise 3.30

Exercise 3.31

This finite state machine is a divide-by-two counter (see Section 3.4.2) when X = 0. When X = 1, the output, *Q*, is HIGH.

current state		input	next state	
s_1	s_0	x	s'_1	s'_0
0	0	0	0	1
0	0	1	1	1
0	1	0	0	0

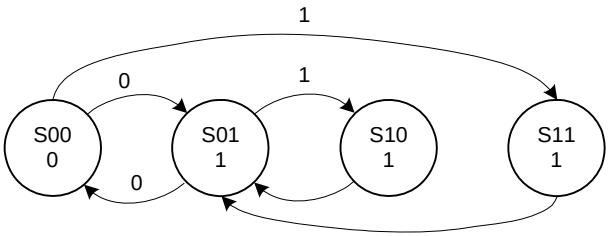
TABLE 3.18 State transition table with binary encodings for Exercise 3.31

current state		input	next state	
s_1	s_0	x	s'_1	s'_0
0	1	1	1	0
1	X	X	0	1

TABLE 3.18 State transition table with binary encodings for Exercise 3.31

current state		output
s_1	s_0	q
0	0	0
0	1	1
1	X	1

TABLE 3.19 Output table for Exercise 3.31



Exercise 3.32

current state			input	next state		
s_2	s_1	s_0	a	s'_2	s'_1	s'_0
0	0	1	0	0	0	1
0	0	1	1	0	1	0
0	1	0	0	0	0	1

TABLE 3.20 State transition table with binary encodings for Exercise 3.32

current state			input	next state		
s_2	s_1	s_0	a	s'_2	s'_1	s'_0
0	1	0	1	1	0	0
1	0	0	0	0	0	1
1	0	0	1	1	0	0

TABLE 3.20 State transition table with binary encodings for Exercise 3.32

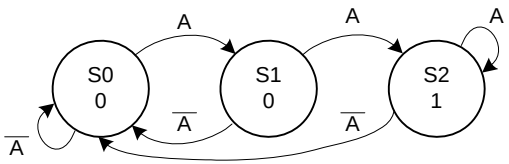


FIGURE 3.13 State transition diagram for Exercise 3.32

Q asserts whenever *A* is HIGH for two or more consecutive cycles.

Exercise 3.33

(a) First, we calculate the propagation delay through the combinational logic:

$t_{pd} = 3t_{pd\_XOR}$
$= 3 \times 100$ ps
$=$ **300 ps**

Next, we calculate the cycle time:

$T_c \geq t_{pcq} + t_{pd} + t_{setup}$
$\geq [70 + 300 + 60]$ ps
$= 430$ ps
$f = 1 / 430$ ps = **2.33 GHz**

(b)

$T_c \geq t_{pcq} + t_{pd} + t_{setup} + t_{skew}$

Thus,

$t_{skew} \leq T_c - (t_{pcq} + t_{pd} + t_{setup})$, where $T_c = 1 / 2$ GHz = 500 ps
$\leq [500 - 430]$ ps = **70 ps**

(c)

First, we calculate the contamination delay through the combinational logic:

$t_{cd} = t_{cd\_XOR}$
$= 55$ ps

$t_{ccq} + t_{cd} > t_{hold} + t_{skew}$
Thus,
$t_{skew} < (t_{ccq} + t_{cd}) - t_{hold}$
$< (50 + 55) - 20$
< 85 ps

(d)

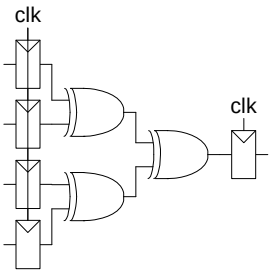


FIGURE 3.14 Alyssa's improved circuit for Exercise 3.33

First, we calculate the propagation and contamination delays through the combinational logic:

$t_{pd} = 2t_{pd\_XOR}$
$= 2 \times 100$ ps
$=$ **200 ps**
$t_{cd} = 2t_{cd\_XOR}$
$= 2 \times 55$ ps
$=$ **110 ps**

Next, we calculate the cycle time:

$T_c \geq t_{pcq} + t_{pd} + t_{setup}$
$\geq [70 + 200 + 60]$ ps
$= 330$ ps
$f = 1 / 330$ ps = **3.03 GHz**

$t_{skew} < (t_{ccq} + t_{cd}) - t_{hold}$
$< (50 + 110) - 20$
< 140 ps

Exercise 3.34

(a) 9.09 GHz
(b) 15 ps
(c) 26 ps

Exercise 3.35

(a) T_c = 1 / 40 MHz = 25 ns
$T_c \geq t_{pcq} + Nt_{\text{CLB}} + t_{\text{setup}}$
25 ns $\geq$ [0.72 + N(0.61) + 0.53] ps
Thus, N < 38.9
N = 38

(b)
$t_{\text{skew}} < (t_{ccq} + t_{cd\_\text{CLB}}) - t_{\text{hold}}$
< [(0.5 + 0.3) - 0] ns
< 0.8 ns = 800 ps

Exercise 3.36

1.138 ns

Exercise 3.37

P(failure)/sec = 1/MTBF = 1/(50 years * 3.15 x 10^7 sec/year) = **6.34 x 10^{-10}** (EQ 3.26)

P(failure)/sec waiting for one clock cycle: $N*(T_0/T_c)*e^{-(Tc-tsetup)/Tau}$

= 0.5 * (110/1000) * $e^{-(1000-70)/100}$ = 5.0 x 10^{-6}

P(failure)/sec waiting for two clock cycles: $N*(T_0/T_c)*[e^{-(Tc-tsetup)/Tau}]^2$

= 0.5 * (110/1000) * [e-$^{(1000-70)/100}$]^2 = 4.6 x 10^{-10}

This is just less than the required probability of failure (6.34 x 10^{-10}). Thus, **2 cycles** of waiting is just adequate to meet the MTBF.

Exercise 3.38

(a) You know you've already entered metastability, so the probability that the sampled signal is metastable is 1. Thus,

$$P(\text{failure}) = 1 \times e^{-\frac{t}{\tau}}$$

Solving for the probability of still being metastable (failing) to be 0.01:

$$P(\text{failure}) = e^{-\frac{t}{\tau}} = 0.01$$

Thus,

$$t = -\tau \times \ln(P(failure)) = -20 \times \ln((0.01)) = \textbf{92 seconds}$$

(b) The probability of death is the chance of still being metastable after 3 minutes

$$\text{P(failure)} = 1 \times e^{-(3\text{ min} \times 60\text{ sec}) / 20\text{ sec}} = \textbf{0.000123}$$

Exercise 3.39

We assume a two flip-flop synchronizer. The most significant impact on the probability of failure comes from the exponential component. If we ignore the T_0/T_c term in the probability of failure equation, assuming it changes little with increases in cycle time, we get:

$$P(\text{failure}) = e^{-\frac{t}{\tau}}$$

$$MTBF = \frac{1}{P(failure)} = e^{\frac{T_c - t_{setup}}{\tau}}$$

$$\frac{MTBF_2}{MTBF_1} = 10 = e^{\frac{T_{c2} - T_{c1}}{30ps}}$$

Solving for T_{c2} - T_{c1}, we get:

$$T_{c2} - T_{c1} = 69ps$$

Thus, the clock cycle time must increase by **69 ps**. This holds true for cycle times much larger than T0 (20 ps) and the increased time (69 ps).

Exercise 3.40

Alyssa is correct. Ben's circuit does not eliminate metastability. After the first transition on D, D2 is always 0 because as D2 transitions from 0 to 1 or 1 to 0, it enters the forbidden region and Ben's "metastability detector" resets the first flip-flop to 0. Even if Ben's circuit could correctly detect a metastable output, it would asynchronously reset the flip-flop which, if the reset occurred around the clock edge, this could cause the second flip-flop to sample a transitioning signal and become metastable.

Question 3.1

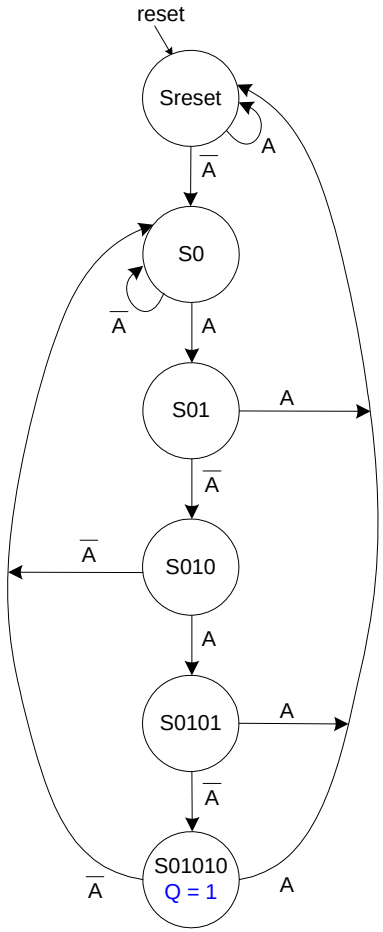


FIGURE 3.15 State transition diagram for Question 3.1

current state $s_{5:0}$	input a	next state $s'_{5:0}$
000001	0	000010
000001	1	000001

TABLE 3.21 State transition table for Question 3.1

current state $s_{5:0}$	input a	next state $s'_{5:0}$
000010	0	000010
000010	1	000100
000100	0	001000
000100	1	000001
001000	0	000010
001000	1	010000
010000	0	100000
010000	1	000001
100000	0	000010
100000	1	000001

TABLE 3.21 State transition table for Question 3.1

$$S'_5 = S_4 A$$
$$S'_4 = S_3 A$$
$$S'_3 = S_2 A$$
$$S'_2 = S_1 A$$
$$S'_1 = A(S_1 + S_3 + S_5)$$
$$S'_0 = A(S_0 + S_2 + S_4 + S_5)$$
$$Q = S_5$$

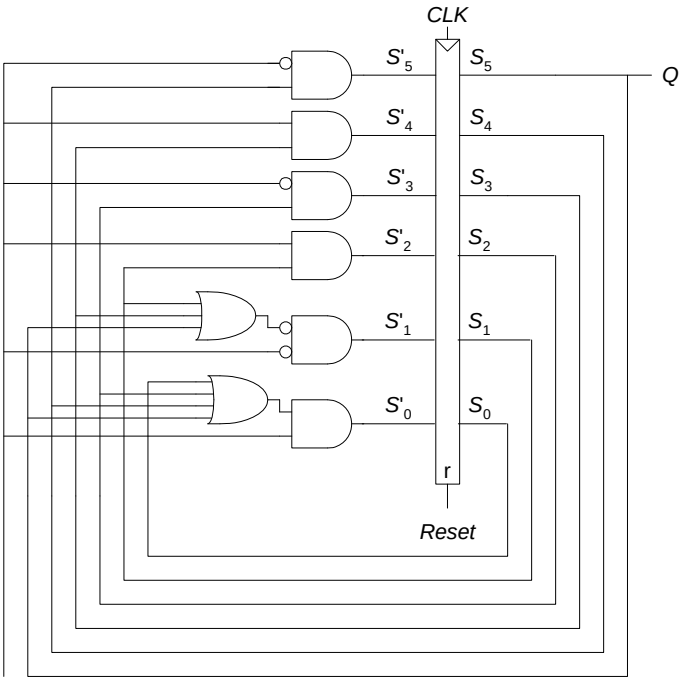


FIGURE 3.16 Finite state machine hardware for Question 3.1

Question 3.2

The FSM should output the value of *A* until after the first 1 is received. It then should output the inverse of *A*. For example, the 8-bit two's complement of the number 6 (00000110) is (11111010). Starting from the least significant bit on the far right, the two's complement is created by outputting the same value of the input until the first 1 is reached. Thus, the two least significant bits of the two's complement number are "10". Then the remaining bits are inverted, making the complete number 11111010.

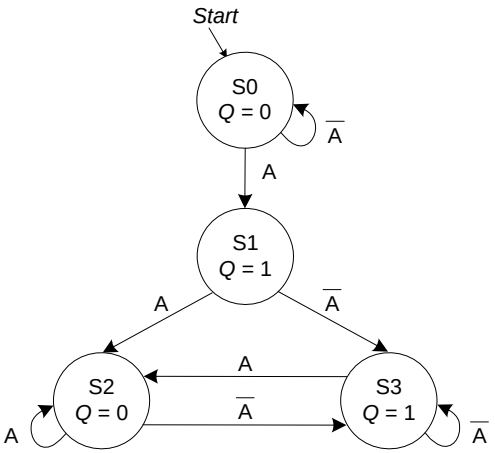


FIGURE 3.17 State transition diagram for Question 3.2

current state $s_{1:0}$	input a	next state $s'_{1:0}$
00	0	00
00	1	01
01	0	11
01	1	10
10	0	11
10	1	10
11	0	11
11	1	10

TABLE 3.22 State transition table for Question 3.2

$$S'_1 = S_1 + S_0$$

$$S'_0 = A \oplus (S_1 + S_0)$$

$$Q = S_0$$

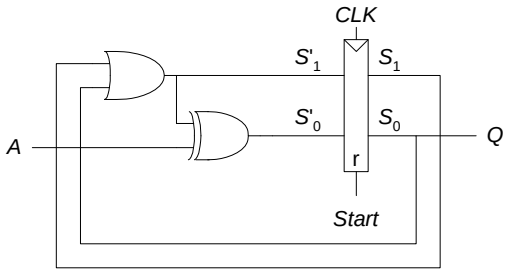


FIGURE 3.18 Finite state machine hardware for Question 3.2

Question 3.3

A latch allows input *D* to flow through to the output *Q* when the clock is HIGH. A flip-flop allows input *D* to flow through to the output *Q* at the clock edge. A flip-flop is preferable in systems with a single clock. Latches are preferable in *two-phase clocking* systems, with two clocks. The two clocks are used to eliminate system failure due to hold time violations. Both the phase and frequency of each clock can be modified independently.

Question 3.4

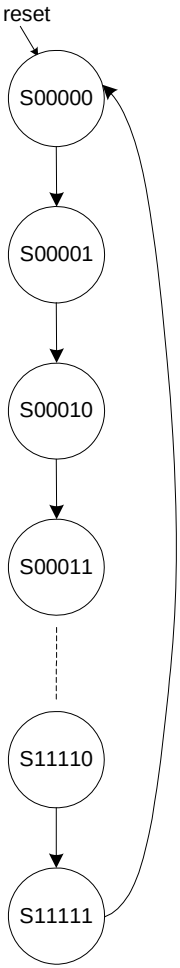


FIGURE 3.19 State transition diagram for Question 3.4

current state $s_{4:0}$	next state $s'_{4:0}$
00000	00001
00001	00010

TABLE 3.23 State transition table for Question 3.4

current state $s_{4:0}$	next state $s'_{4:0}$
00010	00011
00011	00100
00100	00101
...	...
11110	11111
11111	00000

TABLE 3.23 State transition table for Question 3.4

$$S'_4 = S_4 \oplus S_3 S_2 S_1 S_0$$
$$S'_3 = S_3 \oplus S_2 S_1 S_0$$
$$S'_2 = S_2 \oplus S_1 S_0$$
$$S'_1 = S_1 \oplus S_0$$
$$S'_0 = \overline{S_0}$$

$$Q_{4:0} = S_{4:0}$$

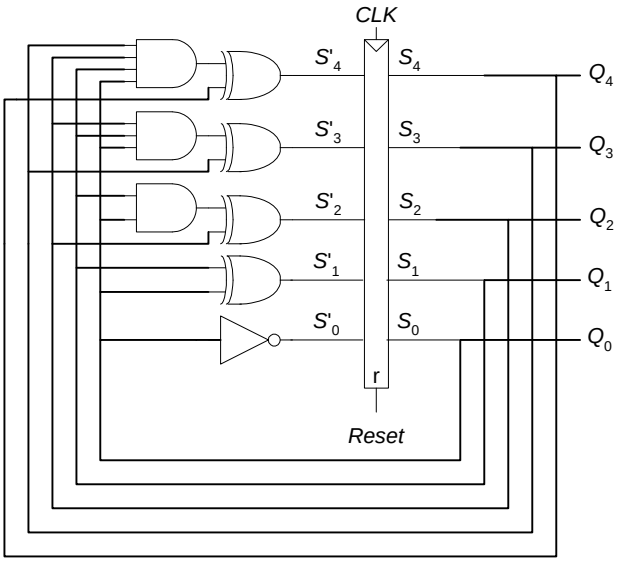


FIGURE 3.20 Finite state machine hardware for Question 3.4

Question 3.5

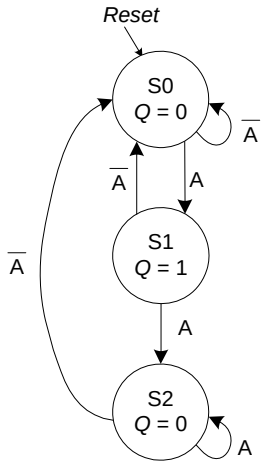


FIGURE 3.21 State transition diagram for edge detector circuit of Question 3.5

current state $s_{1:0}$	input a	next state $s'_{1:0}$
00	0	00
00	1	01
01	0	00
01	1	10
10	0	00
10	1	10

TABLE 3.24 State transition table for Question 3.5

$$S'_1 = AS_1$$
$$S'_0 = AS_1S_0$$

$$Q = S_1$$

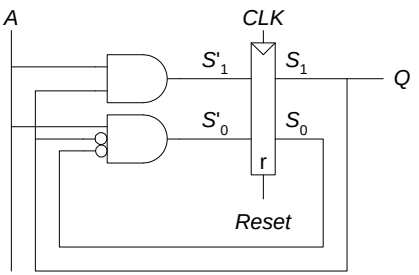


FIGURE 3.22 Finite state machine hardware for Question 3.5

Question 3.6

Pipelining divides a block of combinational logic into N stages, with a register between each stage. Pipelining increases throughput, the number of tasks that can be completed in a given amount of time. Ideally, pipelining increases throughput by a factor of N. But because of the following three reasons, the

speedup is usually less than N: (1) The combinational logic usually cannot be divided into N equal stages. (2) Adding registers between stages adds delay called the *sequencing overhead*, the time it takes to get the signal into and out of the register, $t_{\text{setup}} + t_{pcq}$. (3) The pipeline is not always operating at full capacity: at the beginning of execution, it takes time to fill up the pipeline, and at the end it takes time to drain the pipeline. However, pipelining offers significant speedup at the cost of little extra hardware.

Question 3.7

A flip-flop with a negative hold time allows D to start changing *before* the clock edge arrives.

Question 3.8

We use a divide-by-three counter (see Example 3.6 on page 155 of the textbook) with A as the clock input followed by a *negative edge-triggered* flip-flop, which samples the input, D, on the negative or falling edge of the clock, or in this case, A. The output is the output of the divide-by-three counter, S_0, OR the output of the negative edge-triggered flip-flop, N1. Figure 3.24 shows the waveforms of the internal signals, S_0 and N1.

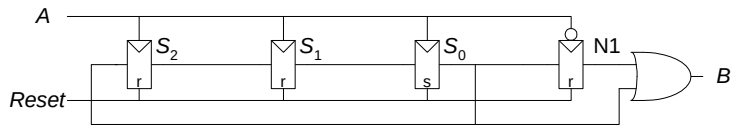


FIGURE 3.23 Hardware for Question 3.8

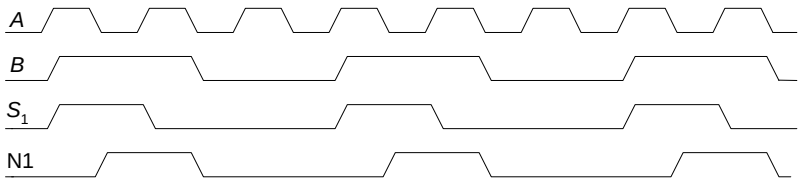


FIGURE 3.24 Waveforms for Question 3.8

Question 3.9

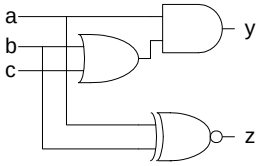
Without the added buffer, the propagation delay through the logic, t_{pd}, must be less than or equal to T_c - (t_{pcq} + t_{setup}). However, if you add a buffer to the clock input of the receiver, the clock arrives at the receiver later. The earliest that the clock edge arrives at the receiver is $t_{\text{cd\_BUF}}$ after the actual clock edge. Thus, the propagation delay through the logic is now given an extra t_{cd_BUF}. So, t_{pd} now must be less than T_c + t_{cd_BUF} - (t_{pcq} + t_{setup}).

CHAPTER 4

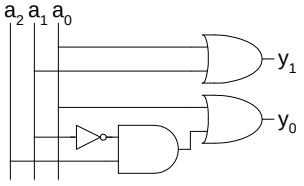
Note: the HDL files given in the following solutions are available on the textbook's companion website at:

http://textbooks.elsevier.com/9780123704979

Exercise 4.1



Exercise 4.2



Exercise 4.3

SystemVerilog

```
module xor_4(input logic [3:0] a,
             output logic      y);

   assign y = ^a;
endmodule
```

VHDL

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

entity xor_4 is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of xor_4 is
begin
  y <= a(3) xor a(2) xor a(1) xor a(0);
end;
```

Exercise 4.4

ex4_4.tv file:

```
0000_0
0001_1
0010_1
0011_0
0100_1
0101_0
0110_0
0111_1
1000_1
1001_0
1010_0
1011_1
1100_0
1101_1
1110_1
1111_0
```

SystemVerilog

```
module ex4_4_testbench();
  logic         clk, reset;
  logic  [3:0]  a;
  logic         yexpected;
  logic         y;
  logic [31:0]  vectornum, errors;
  logic [4:0]   testvectors[10000:0];

  // instantiate device under test
  xor_4 dut(a, y);

  // generate clock
  always
    begin
      clk = 1; #5; clk = 0; #5;
    end

  // at start of test, load vectors
  // and pulse reset
  initial
    begin
      $readmemb("ex4_4.tv", testvectors);
      vectornum = 0; errors = 0;
      reset = 1; #27; reset = 0;
    end

  // apply test vectors on rising edge of clk
  always @(posedge clk)
    begin
      #1; {a, yexpected} =
            testvectors[vectornum];
    end

  // check results on falling edge of clk
   always @(negedge clk)
    if (~reset) begin // skip during reset
      if (y !== yexpected) begin
        $display("Error: inputs = %h", a);
        $display("  outputs = %b (%b expected)",
                 y, yexpected);
        errors = errors + 1;
      end
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 5'bx) begin
       $display("%d tests completed with %d errors",
                vectornum, errors);
        $finish;
      end
    end
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use STD.TEXTIO.all;
use work.txt_util.all

entity ex4_4_testbench is -- no inputs or outputs
end;

architecture sim of ex4_4_testbench is
  component sillyfunction
    port(a: in  STD_LOGIC_VECTOR(3 downto 0);
         y: out STD_LOGIC);
  end component;
  signal a: STD_LOGIC_VECTOR(3 downto 0);
  signal y, clk, reset: STD_LOGIC;
  signal yexpected: STD_LOGIC;
  constant MEMSIZE: integer := 10000;
  type tvarray is array(MEMSIZE downto 0) of
    STD_LOGIC_VECTOR(4 downto 0);
  signal testvectors: tvarray;
  shared variable vectornum, errors: integer;
begin
  -- instantiate device under test
  dut: xor_4 port map(a, y);

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

  -- at start of test, load vectors
  -- and pulse reset
  process is
    file tv: TEXT;
    variable i, j: integer;
    variable L: line;
    variable ch: character;
  begin
    -- read file of test vectors
    i := 0;
    FILE_OPEN(tv, "ex4_4.tv", READ_MODE);
    while not endfile(tv) loop
      readline(tv, L);
      for j in 4 downto 0 loop
        read(L, ch);
        if (ch = '_') then read(L, ch);
        end if;
        if (ch = '0') then
          testvectors(i)(j) <= '0';
        else testvectors(i)(j) <= '1';
        end if;
      end loop;
      i := i + 1;
    end loop;
    vectornum := 0; errors := 0;
    reset <= '1'; wait for 27 ns; reset <= '0';
    wait;
  end process;
```

(*VHDL continued on next page*)

(*continued from previous page*)

VHDL

```
  -- apply test vectors on rising edge of clk
  process (clk) begin
    if (clk'event and clk = '1') then

      a <= testvectors(vectornum)(4 downto 1)
        after 1 ns;
      yexpected <= testvectors(vectornum)(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
      assert y = yexpected
        report "Error: y = " & STD_LOGIC'image(y);
      if (y /= yexpected) then
        errors := errors + 1;
      end if;
      vectornum := vectornum + 1;
      if (is_x(testvectors(vectornum))) then
        if (errors = 0) then
          report "Just kidding -- " &
                 integer'image(vectornum) &
                 " tests completed successfully."
                 severity failure;
        else
          report integer'image(vectornum) &
                 " tests completed, errors = " &
                 integer'image(errors)
                 severity failure;
        end if;
      end if;
    end if;
  end process;
end;
```

Exercise 4.5

SystemVerilog

```
module minority(input  logic a, b, c
                output logic y);

   assign y = ~a & ~b | ~a & ~c | ~b & ~c;
endmodule
```

VHDL

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

entity minority is
  port(a, b, c:   in  STD_LOGIC;
       y:         out STD_LOGIC);
end;

architecture synth of minority is
begin
  y <= ((not a) and (not b)) or ((not a) and (not c))
       or ((not b) and (not c));
end;
```

Exercise 4.6

SystemVerilog

```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);

  always_comb
    case (data)
      //                 abc_defg
      4'h0: segments = 7'b111_1110;
      4'h1: segments = 7'b011_0000;
      4'h2: segments = 7'b110_1101;
      4'h3: segments = 7'b111_1001;
      4'h4: segments = 7'b011_0011;
      4'h5: segments = 7'b101_1011;
      4'h6: segments = 7'b101_1111;
      4'h7: segments = 7'b111_0000;
      4'h8: segments = 7'b111_1111;
      4'h9: segments = 7'b111_0011;
      4'ha: segments = 7'b111_0111;
      4'hb: segments = 7'b001_1111;
      4'hc: segments = 7'b000_1101;
      4'hd: segments = 7'b011_1101;
      4'he: segments = 7'b100_1111;
      4'hf: segments = 7'b100_0111;
    endcase
endmodule
```

VHDL

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

entity seven_seg_decoder is
  port(data:     in  STD_LOGIC_VECTOR(3 downto 0);
       segments: out STD_LOGIC_VECTOR(6 downto 0));
end;

architecture synth of seven_seg_decoder is
begin
  process(all) begin
    case data is
--                              abcdefg
      when X"0"   => segments <= "1111110";
      when X"1"   => segments <= "0110000";
      when X"2"   => segments <= "1101101";
      when X"3"   => segments <= "1111001";
      when X"4"   => segments <= "0110011";
      when X"5"   => segments <= "1011011";
      when X"6"   => segments <= "1011111";
      when X"7"   => segments <= "1110000";
      when X"8"   => segments <= "1111111";
      when X"9"   => segments <= "1110011";
      when X"A"   => segments <= "1110111";
      when X"B"   => segments <= "0011111";
      when X"C"   => segments <= "0001101";
      when X"D"   => segments <= "0111101";
      when X"E"   => segments <= "1001111";
      when X"F"   => segments <= "1000111";
      when others => segments <= "0000000";
    end case;
  end process;
end;
```

Exercise 4.7

ex4_7.tv file:

```
0000_111_1110
0001_011_0000
0010_110_1101
0011_111_1001
0100_011_0011
0101_101_1011
0110_101_1111
0111_111_0000
1000_111_1111
1001_111_1011
1010_111_0111
1011_001_1111
1100_000_1101
1101_011_1101
1110_100_1111
1111_100_0111
```

Option 1:

SystemVerilog

```
module ex4_7_testbench();
  logic        clk, reset;
  logic [3:0]  data;
  logic [6:0]  s_expected;
  logic [6:0]  s;
  logic [31:0] vectornum, errors;
  logic [10:0] testvectors[10000:0];

  // instantiate device under test
  sevenseg dut(data, s);

  // generate clock
  always
    begin
      clk = 1; #5; clk = 0; #5;
    end

  // at start of test, load vectors
  // and pulse reset
  initial
    begin
      $readmemb("ex4_7.tv", testvectors);
      vectornum = 0; errors = 0;
      reset = 1; #27; reset = 0;
    end

  // apply test vectors on rising edge of clk
  always @(posedge clk)
    begin
      #1; {data, s_expected} =
            testvectors[vectornum];
    end

  // check results on falling edge of clk
   always @(negedge clk)
    if (~reset) begin // skip during reset
      if (s !== s_expected) begin
        $display("Error: inputs = %h", data);
        $display("  outputs = %b (%b expected)",
                 s, s_expected);
        errors = errors + 1;
      end
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 11'bx) begin
       $display("%d tests completed with %d errors",
                vectornum, errors);
        $finish;
      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 ex4_7_testbench is -- no inputs or outputs
end;

architecture sim of ex4_7_testbench is
  component seven_seg_decoder
  port(data:     in  STD_LOGIC_VECTOR(3 downto 0);
       segments: out STD_LOGIC_VECTOR(6 downto 0));
  end component;
  signal data: STD_LOGIC_VECTOR(3 downto 0);
  signal s:    STD_LOGIC_VECTOR(6 downto 0);
  signal clk, reset: STD_LOGIC;
  signal s_expected: STD_LOGIC_VECTOR(6 downto 0);
  constant MEMSIZE: integer := 10000;
  type tvarray is array(MEMSIZE downto 0) of
    STD_LOGIC_VECTOR(10 downto 0);
  signal testvectors: tvarray;
  shared variable vectornum, errors: integer;
begin
  -- instantiate device under test
  dut: seven_seg_decoder port map(data, s);

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

  -- at start of test, load vectors
  -- and pulse reset
  process is
    file tv: TEXT;
    variable i, j: integer;
    variable L: line;
    variable ch: character;
  begin
    -- read file of test vectors
    i := 0;
    FILE_OPEN(tv, "ex4_7.tv", READ_MODE);
    while not endfile(tv) loop
      readline(tv, L);
      for j in 10 downto 0 loop
        read(L, ch);
        if (ch = '_') then read(L, ch);
        end if;
        if (ch = '0') then
          testvectors(i)(j) <= '0';
        else testvectors(i)(j) <= '1';
        end if;
      end loop;
      i := i + 1;
    end loop;
```

(*VHDL continued on next page*)

(*continued from previous page*)

VHDL

```
    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

      data <= testvectors(vectornum)(10 downto 7)
        after 1 ns;
     s_expected <= testvectors(vectornum)(6 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
      assert s = s_expected
        report "data = " &
          integer'image(CONV_INTEGER(data)) &
          "; s = " &
          integer'image(CONV_INTEGER(s)) &
          "; s_expected = " &
           integer'image(CONV_INTEGER(s_expected));
      if (s /= s_expected) then
        errors := errors + 1;
      end if;
      vectornum := vectornum + 1;
      if (is_x(testvectors(vectornum))) then
        if (errors = 0) then
          report "Just kidding -- " &
                 integer'image(vectornum) &
                 " tests completed successfully."
                 severity failure;
        else
          report integer'image(vectornum) &
                 " tests completed, errors = " &
                 integer'image(errors)
                 severity failure;
        end if;
      end if;
    end if;
  end process;
end;
```

Option 2 (VHDL only):

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use STD.TEXTIO.all;
use work.txt_util.all;

entity ex4_7_testbench is -- no inputs or outputs
end;

architecture sim of ex4_7_testbench is
  component seven_seg_decoder
  port(data:     in  STD_LOGIC_VECTOR(3 downto 0);
       segments: out STD_LOGIC_VECTOR(6 downto 0));
  end component;
  signal data: STD_LOGIC_VECTOR(3 downto 0);
  signal s:    STD_LOGIC_VECTOR(6 downto 0);
  signal clk, reset: STD_LOGIC;
  signal s_expected: STD_LOGIC_VECTOR(6 downto 0);
  constant MEMSIZE: integer := 10000;
  type tvarray is array(MEMSIZE downto 0) of
    STD_LOGIC_VECTOR(10 downto 0);
  signal testvectors: tvarray;
  shared variable vectornum, errors: integer;
begin
  -- instantiate device under test
  dut: seven_seg_decoder port map(data, s);

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

  -- at start of test, load vectors
  -- and pulse reset
  process is
    file tv: TEXT;
    variable i, j: integer;
    variable L: line;
    variable ch: character;
  begin
    -- read file of test vectors
    i := 0;
    FILE_OPEN(tv, "ex4_7.tv", READ_MODE);
    while not endfile(tv) loop
      readline(tv, L);
      for j in 10 downto 0 loop
        read(L, ch);
        if (ch = '_') then read(L, ch);
        end if;
        if (ch = '0') then
          testvectors(i)(j) <= '0';
        else testvectors(i)(j) <= '1';
        end if;
      end loop;
      i := i + 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

      data <= testvectors(vectornum)(10 downto 7)
        after 1 ns;
     s_expected <= testvectors(vectornum)(6 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
      assert s = s_expected
        report "data = " & str(data) &
          "; s = " & str(s) &
           "; s_expected = " & str(s_expected);
      if (s /= s_expected) then
        errors := errors + 1;
      end if;
      vectornum := vectornum + 1;
      if (is_x(testvectors(vectornum))) then
        if (errors = 0) then
          report "Just kidding -- " &
                 integer'image(vectornum) &
                 " tests completed successfully."
                 severity failure;
        else
          report integer'image(vectornum) &
                 " tests completed, errors = " &
                 integer'image(errors)
                 severity failure;
        end if;
      end if;
    end if;
  end process;
end;
```

(*see Web site for file: txt_util.vhd*)

Exercise 4.8

SystemVerilog

```
module mux8
  #(parameter width = 4)
   (input logic  [width-1:0] d0, d1, d2, d3,
                             d4, d5, d6, d7,
    input logic  [2:0]       s,
    output logic [width-1:0] y);

   always_comb
     case (s)
       0: y = d0;
       1: y = d1;
       2: y = d2;
       3: y = d3;
       4: y = d4;
       5: y = d5;
       6: y = d6;
       7: y = d7;
     endcase
endmodule
```

VHDL

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

entity mux8 is
  generic(width: integer := 4);
  port(d0,
       d1,
       d2,
       d3,
       d4,
       d5,
       d6,
       d7: in  STD_LOGIC_VECTOR(width-1 downto 0);
       s:  in  STD_LOGIC_VECTOR(2 downto 0);
        y:  out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture synth of mux8 is
begin
  with s select y <=
    d0 when "000",
    d1 when "001",
    d2 when "010",
    d3 when "011",
    d4 when "100",
    d5 when "101",
    d6 when "110",
    d7 when others;
end;
```

Exercise 4.9

SystemVerilog

```
module ex4_9
   (input  logic a, b, c,
    output logic y);

   mux8 #(1) mux8_1(1'b1, 1'b0, 1'b0, 1'b1,
                    1'b1, 1'b1, 1'b0, 1'b0,
                    {a,b,c}, y);
endmodule
```

VHDL

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

entity ex4_9 is
  port(a,
       b,
       c: in  STD_LOGIC;
       y: out STD_LOGIC_VECTOR(0 downto 0));
end;

architecture struct of ex4_9 is
  component mux8
    generic(width: integer);
  port(d0, d1, d2, d3, d4, d5, d6,
       d7:  in  STD_LOGIC_VECTOR(width-1 downto 0);
       s:   in  STD_LOGIC_VECTOR(2 downto 0);
       y:   out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  signal sel: STD_LOGIC_VECTOR(2 downto 0);
begin
  sel <= a & b & c;

  mux8_1: mux8 generic map(1)
               port map("1", "0", "0", "1",
                        "1", "1", "0", "0",
                        sel, y);
end;
```

Exercise 4.10

SystemVerilog

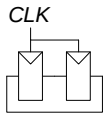
```
module ex4_10
   (input  logic a, b, c,
    output logic y);

   mux4 #(1) mux4_1( ~c, c, 1'b1, 1'b0, {a, b}, y);
endmodule

module mux4
  #(parameter width = 4)
   (input  logic [width-1:0] d0, d1, d2, d3,
    input  logic [1:0]       s,
    output logic [width-1:0] y);

   always_comb
     case (s)
       0: y = d0;
       1: y = d1;
       2: y = d2;
       3: y = d3;
     endcase
endmodule
```

VHDL

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

entity ex4_10 is
  port(a,
       b,
       c: in  STD_LOGIC;
       y: out STD_LOGIC_VECTOR(0 downto 0));
end;

architecture struct of ex4_10 is
  component mux4
    generic(width: integer);
    port(d0, d1, d2,
       d3: in  STD_LOGIC_VECTOR(width-1 downto 0);
          s: in  STD_LOGIC_VECTOR(1 downto 0);
         y: out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  signal cb:      STD_LOGIC_VECTOR(0 downto 0);
  signal c_vect:  STD_LOGIC_VECTOR(0 downto 0);
  signal sel:     STD_LOGIC_VECTOR(1 downto 0);
begin
  c_vect(0) <= c;
  cb(0) <= not c;
  sel <= (a & b);
  mux4_1: mux4 generic map(1)
              port map(cb, c_vect, "1", "0", sel, y);
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4 is
  generic(width: integer := 4);
  port(d0,
       d1,
       d2,
       d3: in  STD_LOGIC_VECTOR(width-1 downto 0);
       s:  in  STD_LOGIC_VECTOR(1 downto 0);
       y:  out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture synth of mux4 is
begin
  with s select y <=
    d0 when "00",
    d1 when "01",
    d2 when "10",
    d3 when others;
end;
```

Exercise 4.11

A shift register with feedback, shown below, cannot be correctly described with blocking assignments.



Exercise 4.12

SystemVerilog

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

  always_comb
    casez (a)
      8'b1???????: y = 8'b10000000;
      8'b01??????: y = 8'b01000000;
      8'b001?????: y = 8'b00100000;
      8'b0001????: y = 8'b00010000;
      8'b00001???: y = 8'b00001000;
      8'b000001??: y = 8'b00000100;
      8'b0000001?: y = 8'b00000010;
      8'b00000001: y = 8'b00000001;
      default:     y = 8'b00000000;
    endcase
endmodule
```

VHDL

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

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

architecture synth of priority is
begin
  process(all) begin
    if    a(7) = '1' then y <= "10000000";
    elsif a(6) = '1' then y <= "01000000";
    elsif a(5) = '1' then y <= "00100000";
    elsif a(4) = '1' then y <= "00010000";
    elsif a(3) = '1' then y <= "00001000";
    elsif a(2) = '1' then y <= "00000100";
    elsif a(1) = '1' then y <= "00000010";
    elsif a(0) = '1' then y <= "00000001";
    else                  y <= "00000000";
    end if;
  end process;
end;
```

Exercise 4.13

SystemVerilog

```
module decoder2_4(input  logic [1:0] a,
                  output logic [3:0] y);
  always_comb
    case (a)
      2'b00: y = 4'b0001;
      2'b01: y = 4'b0010;
      2'b10: y = 4'b0100;
      2'b11: y = 4'b1000;
    endcase
endmodule
```

VHDL

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

entity decoder2_4 is
  port(a: in  STD_LOGIC_VECTOR(1 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of decoder2_4 is
begin
  process(all) begin
    case a is
      when "00"   => y <= "0001";
      when "01"   => y <= "0010";
      when "10"   => y <= "0100";
      when "11"   => y <= "1000";
      when others => y <= "0000";
    end case;
  end process;
end;
```

Exercise 4.14

SystemVerilog

```
module decoder6_64(input  logic [5:0]  a,
                   output logic [63:0] y);

  logic [11:0] y2_4;

  decoder2_4 dec0(a[1:0], y2_4[3:0]);
  decoder2_4 dec1(a[3:2], y2_4[7:4]);
  decoder2_4 dec2(a[5:4], y2_4[11:8]);

  assign y[0] = y2_4[0] & y2_4[4] & y2_4[8];
  assign y[1] = y2_4[1] & y2_4[4] & y2_4[8];
  assign y[2] = y2_4[2] & y2_4[4] & y2_4[8];
  assign y[3] = y2_4[3] & y2_4[4] & y2_4[8];
  assign y[4] = y2_4[0] & y2_4[5] & y2_4[8];
  assign y[5] = y2_4[1] & y2_4[5] & y2_4[8];
  assign y[6] = y2_4[2] & y2_4[5] & y2_4[8];
  assign y[7] = y2_4[3] & y2_4[5] & y2_4[8];
  assign y[8] = y2_4[0] & y2_4[6] & y2_4[8];
  assign y[9] = y2_4[1] & y2_4[6] & y2_4[8];
  assign y[10] = y2_4[2] & y2_4[6] & y2_4[8];
  assign y[11] = y2_4[3] & y2_4[6] & y2_4[8];
  assign y[12] = y2_4[0] & y2_4[7] & y2_4[8];
  assign y[13] = y2_4[1] & y2_4[7] & y2_4[8];
  assign y[14] = y2_4[2] & y2_4[7] & y2_4[8];
  assign y[15] = y2_4[3] & y2_4[7] & y2_4[8];
  assign y[16] = y2_4[0] & y2_4[4] & y2_4[9];
  assign y[17] = y2_4[1] & y2_4[4] & y2_4[9];
  assign y[18] = y2_4[2] & y2_4[4] & y2_4[9];
  assign y[19] = y2_4[3] & y2_4[4] & y2_4[9];
  assign y[20] = y2_4[0] & y2_4[5] & y2_4[9];
  assign y[21] = y2_4[1] & y2_4[5] & y2_4[9];
  assign y[22] = y2_4[2] & y2_4[5] & y2_4[9];
  assign y[23] = y2_4[3] & y2_4[5] & y2_4[9];
  assign y[24] = y2_4[0] & y2_4[6] & y2_4[9];
  assign y[25] = y2_4[1] & y2_4[6] & y2_4[9];
  assign y[26] = y2_4[2] & y2_4[6] & y2_4[9];
  assign y[27] = y2_4[3] & y2_4[6] & y2_4[9];
  assign y[28] = y2_4[0] & y2_4[7] & y2_4[9];
  assign y[29] = y2_4[1] & y2_4[7] & y2_4[9];
  assign y[30] = y2_4[2] & y2_4[7] & y2_4[9];
  assign y[31] = y2_4[3] & y2_4[7] & y2_4[9];
```

VHDL

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

entity decoder6_64 is
  port(a: in  STD_LOGIC_VECTOR(5 downto 0);
       y: out STD_LOGIC_VECTOR(63 downto 0));
end;

architecture struct of decoder6_64 is
  component decoder2_4
    port(a:  in  STD_LOGIC_VECTOR(1 downto 0);
         y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal y2_4: STD_LOGIC_VECTOR(11 downto 0);
begin
  dec0: decoder2_4 port map(a(1 downto 0),
                            y2_4(3 downto 0));
  dec1: decoder2_4 port map(a(3 downto 2),
                            y2_4(7 downto 4));
  dec2: decoder2_4 port map(a(5 downto 4),
                            y2_4(11 downto 8));

  y(0) <= y2_4(0) and y2_4(4) and y2_4(8);
  y(1) <= y2_4(1) and y2_4(4) and y2_4(8);
  y(2) <= y2_4(2) and y2_4(4) and y2_4(8);
  y(3) <= y2_4(3) and y2_4(4) and y2_4(8);
  y(4) <= y2_4(0) and y2_4(5) and y2_4(8);
  y(5) <= y2_4(1) and y2_4(5) and y2_4(8);
  y(6) <= y2_4(2) and y2_4(5) and y2_4(8);
  y(7) <= y2_4(3) and y2_4(5) and y2_4(8);
  y(8) <= y2_4(0) and y2_4(6) and y2_4(8);
  y(9) <= y2_4(1) and y2_4(6) and y2_4(8);
  y(10) <= y2_4(2) and y2_4(6) and y2_4(8);
  y(11) <= y2_4(3) and y2_4(6) and y2_4(8);
  y(12) <= y2_4(0) and y2_4(7) and y2_4(8);
  y(13) <= y2_4(1) and y2_4(7) and y2_4(8);
  y(14) <= y2_4(2) and y2_4(7) and y2_4(8);
  y(15) <= y2_4(3) and y2_4(7) and y2_4(8);
  y(16) <= y2_4(0) and y2_4(4) and y2_4(9);
  y(17) <= y2_4(1) and y2_4(4) and y2_4(9);
  y(18) <= y2_4(2) and y2_4(4) and y2_4(9);
  y(19) <= y2_4(3) and y2_4(4) and y2_4(9);
  y(20) <= y2_4(0) and y2_4(5) and y2_4(9);
  y(21) <= y2_4(1) and y2_4(5) and y2_4(9);
  y(22) <= y2_4(2) and y2_4(5) and y2_4(9);
  y(23) <= y2_4(3) and y2_4(5) and y2_4(9);
  y(24) <= y2_4(0) and y2_4(6) and y2_4(9);
  y(25) <= y2_4(1) and y2_4(6) and y2_4(9);
  y(26) <= y2_4(2) and y2_4(6) and y2_4(9);
  y(27) <= y2_4(3) and y2_4(6) and y2_4(9);
  y(28) <= y2_4(0) and y2_4(7) and y2_4(9);
  y(29) <= y2_4(1) and y2_4(7) and y2_4(9);
  y(30) <= y2_4(2) and y2_4(7) and y2_4(9);
  y(31) <= y2_4(3) and y2_4(7) and y2_4(9);
```

(continued on next page)

(continued from previous page)

SystemVerilog

```
  assign y[32] = y2_4[0] & y2_4[4] & y2_4[10];
  assign y[33] = y2_4[1] & y2_4[4] & y2_4[10];
  assign y[34] = y2_4[2] & y2_4[4] & y2_4[10];
  assign y[35] = y2_4[3] & y2_4[4] & y2_4[10];
  assign y[36] = y2_4[0] & y2_4[5] & y2_4[10];
  assign y[37] = y2_4[1] & y2_4[5] & y2_4[10];
  assign y[38] = y2_4[2] & y2_4[5] & y2_4[10];
  assign y[39] = y2_4[3] & y2_4[5] & y2_4[10];
  assign y[40] = y2_4[0] & y2_4[6] & y2_4[10];
  assign y[41] = y2_4[1] & y2_4[6] & y2_4[10];
  assign y[42] = y2_4[2] & y2_4[6] & y2_4[10];
  assign y[43] = y2_4[3] & y2_4[6] & y2_4[10];
  assign y[44] = y2_4[0] & y2_4[7] & y2_4[10];
  assign y[45] = y2_4[1] & y2_4[7] & y2_4[10];
  assign y[46] = y2_4[2] & y2_4[7] & y2_4[10];
  assign y[47] = y2_4[3] & y2_4[7] & y2_4[10];
  assign y[48] = y2_4[0] & y2_4[4] & y2_4[11];
  assign y[49] = y2_4[1] & y2_4[4] & y2_4[11];
  assign y[50] = y2_4[2] & y2_4[4] & y2_4[11];
  assign y[51] = y2_4[3] & y2_4[4] & y2_4[11];
  assign y[52] = y2_4[0] & y2_4[5] & y2_4[11];
  assign y[53] = y2_4[1] & y2_4[5] & y2_4[11];
  assign y[54] = y2_4[2] & y2_4[5] & y2_4[11];
  assign y[55] = y2_4[3] & y2_4[5] & y2_4[11];
  assign y[56] = y2_4[0] & y2_4[6] & y2_4[11];
  assign y[57] = y2_4[1] & y2_4[6] & y2_4[11];
  assign y[58] = y2_4[2] & y2_4[6] & y2_4[11];
  assign y[59] = y2_4[3] & y2_4[6] & y2_4[11];
  assign y[60] = y2_4[0] & y2_4[7] & y2_4[11];
  assign y[61] = y2_4[1] & y2_4[7] & y2_4[11];
  assign y[62] = y2_4[2] & y2_4[7] & y2_4[11];
  assign y[63] = y2_4[3] & y2_4[7] & y2_4[11];
endmodule
```

VHDL

```
  y(32) <= y2_4(0) and y2_4(4) and y2_4(10);
  y(33) <= y2_4(1) and y2_4(4) and y2_4(10);
  y(34) <= y2_4(2) and y2_4(4) and y2_4(10);
  y(35) <= y2_4(3) and y2_4(4) and y2_4(10);
  y(36) <= y2_4(0) and y2_4(5) and y2_4(10);
  y(37) <= y2_4(1) and y2_4(5) and y2_4(10);
  y(38) <= y2_4(2) and y2_4(5) and y2_4(10);
  y(39) <= y2_4(3) and y2_4(5) and y2_4(10);
  y(40) <= y2_4(0) and y2_4(6) and y2_4(10);
  y(41) <= y2_4(1) and y2_4(6) and y2_4(10);
  y(42) <= y2_4(2) and y2_4(6) and y2_4(10);
  y(43) <= y2_4(3) and y2_4(6) and y2_4(10);
  y(44) <= y2_4(0) and y2_4(7) and y2_4(10);
  y(45) <= y2_4(1) and y2_4(7) and y2_4(10);
  y(46) <= y2_4(2) and y2_4(7) and y2_4(10);
  y(47) <= y2_4(3) and y2_4(7) and y2_4(10);
  y(48) <= y2_4(0) and y2_4(4) and y2_4(11);
  y(49) <= y2_4(1) and y2_4(4) and y2_4(11);
  y(50) <= y2_4(2) and y2_4(4) and y2_4(11);
  y(51) <= y2_4(3) and y2_4(4) and y2_4(11);
  y(52) <= y2_4(0) and y2_4(5) and y2_4(11);
  y(53) <= y2_4(1) and y2_4(5) and y2_4(11);
  y(54) <= y2_4(2) and y2_4(5) and y2_4(11);
  y(55) <= y2_4(3) and y2_4(5) and y2_4(11);
  y(56) <= y2_4(0) and y2_4(6) and y2_4(11);
  y(57) <= y2_4(1) and y2_4(6) and y2_4(11);
  y(58) <= y2_4(2) and y2_4(6) and y2_4(11);
  y(59) <= y2_4(3) and y2_4(6) and y2_4(11);
  y(60) <= y2_4(0) and y2_4(7) and y2_4(11);
  y(61) <= y2_4(1) and y2_4(7) and y2_4(11);
  y(62) <= y2_4(2) and y2_4(7) and y2_4(11);
  y(63) <= y2_4(3) and y2_4(7) and y2_4(11);
end;
```

Exercise 4.15

(a) $Y = AC + \bar{A}\bar{B}C$

SystemVerilog

```
module ex4_15a(input  logic a, b, c,
               output logic y);

  assign y = (a & c) | (~a & ~b & c);
endmodule
```

VHDL

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

entity ex4_15a is
  port(a, b, c: in  STD_LOGIC;
       y:       out STD_LOGIC);
end;

architecture behave of ex4_15a is
begin
  y <= (not a and not b and c) or (not b and c);
end;
```

(b) $Y = \bar{A}\bar{B} + \bar{A}B\bar{C} + \overline{(A + \bar{C})}$

SystemVerilog

```
module ex4_15b(input  logic a, b, c,
               output logic y);

  assign y = (~a & ~b) | (~a & b & ~c) | ~(a | ~c);
endmodule
```

VHDL

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

entity ex4_15b is
  port(a, b, c: in  STD_LOGIC;
       y:       out STD_LOGIC);
end;

architecture behave of ex4_15b is
begin
  y <= ((not a) and (not b)) or ((not a) and b and
       (not c)) or (not(a or (not c)));
end;
```

(c) $Y = \bar{A}B\bar{C}\bar{D} + A\bar{B}\bar{C} + A\bar{B}C\bar{D} + ABD + \bar{A}\bar{B}C\bar{D} + B\bar{C}D + \bar{A}$

SystemVerilog

```
module ex4_15c(input  logic a, b, c, d,
               output logic y);

  assign y = (~a & ~b & ~c & ~d) | (a & ~b & ~c) |
             (a & ~b & c & ~d) | (a & b & d) |
             (~a & ~b & c & ~d) | (b & ~c & d) | ~a;
endmodule
```

VHDL

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

entity ex4_15c is
  port(a, b, c, d: in  STD_LOGIC;
       y:       out STD_LOGIC);
end;

architecture behave of ex4_15c is
begin
 y <= ((not a) and (not b) and (not c) and (not d)) or
       (a and (not b) and (not c)) or
       (a and (not b) and c and (not d)) or
       (a and b and d) or
       ((not a) and (not b) and c and (not d)) or
       (b and (not c) and d) or (not a);
end;
```

Exercise 4.16

SystemVerilog

```
module ex4_16(input  logic a, b, c, d, e,
              output logic y);

  assign y = ~(~(~(a & b) & ~(c & d)) & e);
endmodule
```

VHDL

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

entity ex4_16 is
  port(a, b, c, d, e: in  STD_LOGIC;
       y:             out STD_LOGIC);
end;

architecture behave of ex4_16 is
begin
  y <= not((not((not(a and b)) and
                (not(c and d)))) and e);

end;
```

Exercise 4.17

SystemVerilog

```
module ex4_17(input  logic a, b, c, d, e, f, g
              output logic y);

  logic n1, n2, n3, n4, n5;

  assign n1 = ~(a & b & c);
  assign n2 = ~(n1 & d);
  assign n3 = ~(f & g);
  assign n4 = ~(n3 | e);
  assign n5 = ~(n2 | n4);
  assign y = ~(n5 & n5);
endmodule
```

VHDL

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

entity ex4_17 is
  port(a, b, c, d, e, f, g: in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of ex4_17 is
  signal n1, n2, n3, n4, n5: STD_LOGIC;
begin
  n1 <= not(a and b and c);
  n2 <= not(n1 and d);
  n3 <= not(f and g);
  n4 <= not(n3 or e);
  n5 <= not(n2 or n4);
  y <= not (n5 or n5);
end;
```

Exercise 4.18

Verilog

```
module ex4_18(input  logic a, b, c, d,
              output logic y);

  always_comb
    casez ({a, b, c, d})
      // note: outputs cannot be assigned don't care
       0: y = 1'b0;
       1: y = 1'b0;
       2: y = 1'b0;
       3: y = 1'b0;
       4: y = 1'b0;
       5: y = 1'b0;
       6: y = 1'b0;
       7: y = 1'b0;
       8: y = 1'b1;
       9: y = 1'b0;
      10: y = 1'b0;
      11: y = 1'b1;
      12: y = 1'b1;
      13: y = 1'b1;
      14: y = 1'b0;
      15: y = 1'b1;
    endcase
endmodule
```

VHDL

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

entity ex4_18 is
  port(a, b, c, d: in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of ex4_17 is
signal vars: STD_LOGIC_VECTOR(3 downto 0);
begin
  vars <= (a & b & c & d);
  process(all) begin
    case vars is
      -- note: outputs cannot be assigned don't care
      when X"0" => y <= '0';
      when X"1" => y <= '0';
      when X"2" => y <= '0';
      when X"3" => y <= '0';
      when X"4" => y <= '0';
      when X"5" => y <= '0';
      when X"6" => y <= '0';
      when X"7" => y <= '0';
      when X"8" => y <= '1';
      when X"9" => y <= '0';
      when X"A" => y <= '0';
      when X"B" => y <= '1';
      when X"C" => y <= '1';
      when X"D" => y <= '1';
      when X"E" => y <= '0';
      when X"F" => y <= '1';
      when others => y <= '0';--should never happen
    end case;
  end process;
end;
```

Exercise 4.19

SystemVerilog

```
module ex4_18(input  logic [3:0] a,
              output logic       p, d);

  always_comb
    case (a)
       0: {p, d} = 2'b00;
       1: {p, d} = 2'b00;
       2: {p, d} = 2'b10;
       3: {p, d} = 2'b11;
       4: {p, d} = 2'b00;
       5: {p, d} = 2'b10;
       6: {p, d} = 2'b01;
       7: {p, d} = 2'b10;
       8: {p, d} = 2'b00;
       9: {p, d} = 2'b01;
      10: {p, d} = 2'b00;
      11: {p, d} = 2'b10;
      12: {p, d} = 2'b01;
      13: {p, d} = 2'b10;
      14: {p, d} = 2'b00;
      15: {p, d} = 2'b01;
    endcase
endmodule
```

VHDL

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

entity ex4_18 is
  port(a:    in  STD_LOGIC_VECTOR(3 downto 0);
       p, d: out STD_LOGIC);
end;

architecture synth of ex4_18 is
signal vars: STD_LOGIC_VECTOR(1 downto 0);
begin
  p <= vars(1);
  d <= vars(0);
  process(all) begin
    case a is
      when X"0"   => vars <= "00";
      when X"1"   => vars <= "00";
      when X"2"   => vars <= "10";
      when X"3"   => vars <= "11";
      when X"4"   => vars <= "00";
      when X"5"   => vars <= "10";
      when X"6"   => vars <= "01";
      when X"7"   => vars <= "10";
      when X"8"   => vars <= "00";
      when X"9"   => vars <= "01";
      when X"A"   => vars <= "00";
      when X"B"   => vars <= "10";
      when X"C"   => vars <= "01";
      when X"D"   => vars <= "10";
      when X"E"   => vars <= "00";
      when X"F"   => vars <= "01";
      when others => vars <= "00";
    end case;
  end process;
end;
```

Exercise 4.20

SystemVerilog

```
module priority_encoder(input  logic [7:0] a,
                        output logic [2:0] y,
                        output logic       none);

  always_comb
    casez (a)
       8'b00000000: begin y = 3'd0;  none = 1'b1; end
       8'b00000001: begin y = 3'd0;  none = 1'b0; end
       8'b0000001?: begin y = 3'd1;  none = 1'b0; end
       8'b000001??: begin y = 3'd2;  none = 1'b0; end
       8'b00001???: begin y = 3'd3;  none = 1'b0; end
       8'b0001????: begin y = 3'd4;  none = 1'b0; end
       8'b001?????: begin y = 3'd5;  none = 1'b0; end
       8'b01??????: begin y = 3'd6;  none = 1'b0; end
       8'b1???????: begin y = 3'd7;  none = 1'b0; end
    endcase
endmodule
```

VHDL

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

entity priority_encoder is
  port(a:    in  STD_LOGIC_VECTOR(7 downto 0);
       y:    out STD_LOGIC_VECTOR(2 downto 0);
       none: out STD_LOGIC);
end;

architecture synth of priority_encoder is
begin
  process(all) begin
    case? a is
      when "00000000" => y <= "000"; none <= '1';
      when "00000001" => y <= "000"; none <= '0';
      when "0000001-" => y <= "001"; none <= '0';
      when "000001--" => y <= "010"; none <= '0';
      when "00001---" => y <= "011"; none <= '0';
      when "0001----" => y <= "100"; none <= '0';
      when "001-----" => y <= "101"; none <= '0';
      when "01------" => y <= "110"; none <= '0';
      when "1-------" => y <= "111"; none <= '0';
      when others     => y <= "000"; none <= '0';
    end case?;
  end process;
end;
```

Exercise 4.21

SystemVerilog

```
module priority_encoder2(input  logic [7:0] a,
                         output logic [2:0] y, z,
                         output logic       none);

  always_comb
  begin
    casez (a)
       8'b00000000: begin y = 3'd0; none = 1'b1; end
       8'b00000001: begin y = 3'd0; none = 1'b0; end
       8'b0000001?: begin y = 3'd1; none = 1'b0; end
       8'b000001??: begin y = 3'd2; none = 1'b0; end
       8'b00001???: begin y = 3'd3; none = 1'b0; end
       8'b0001????: begin y = 3'd4; none = 1'b0; end
       8'b001?????: begin y = 3'd5; none = 1'b0; end
       8'b01??????: begin y = 3'd6; none = 1'b0; end
       8'b1???????: begin y = 3'd7; none = 1'b0; end
    endcase

    casez (a)
      8'b00000011: z = 3'b000;
      8'b00000101: z = 3'b000;
      8'b00001001: z = 3'b000;
      8'b00010001: z = 3'b000;
      8'b00100001: z = 3'b000;
      8'b01000001: z = 3'b000;
      8'b10000001: z = 3'b000;
      8'b0000011?: z = 3'b001;
      8'b0000101?: z = 3'b001;
      8'b0001001?: z = 3'b001;
      8'b0010001?: z = 3'b001;
      8'b0100001?: z = 3'b001;
      8'b1000001?: z = 3'b001;
      8'b000011??: z = 3'b010;
      8'b000101??: z = 3'b010;
      8'b001001??: z = 3'b010;
      8'b010001??: z = 3'b010;
      8'b100001??: z = 3'b010;
      8'b00011???: z = 3'b011;
      8'b00101???: z = 3'b011;
      8'b01001???: z = 3'b011;
      8'b10001???: z = 3'b011;
      8'b0011????: z = 3'b100;
      8'b0101????: z = 3'b100;
      8'b1001????: z = 3'b100;
      8'b011?????: z = 3'b101;
      8'b101?????: z = 3'b101;
      8'b11??????: z = 3'b110;
      default:     z = 3'b000;
  end
endmodule
```

VHDL

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

entity priority_encoder2 is
  port(a:    in  STD_LOGIC_VECTOR(7 downto 0);
       y, z: out STD_LOGIC_VECTOR(2 downto 0);
       none: out STD_LOGIC);
end;

architecture synth of priority_encoder is
begin
  process(all) begin
    case? a is
      when "00000000" => y <= "000"; none <= '1';
      when "00000001" => y <= "000"; none <= '0';
      when "0000001-" => y <= "001"; none <= '0';
      when "000001--" => y <= "010"; none <= '0';
      when "00001---" => y <= "011"; none <= '0';
      when "0001----" => y <= "100"; none <= '0';
      when "001-----" => y <= "101"; none <= '0';
      when "01------" => y <= "110"; none <= '0';
      when "1-------" => y <= "111"; none <= '0';
      when others     => y <= "000"; none <= '0';
    end case?;
    case? a is
      when "00000011" => z <= "000";
      when "00000101" => z <= "000";
      when "00001001" => z <= "000";
      when "00001001" => z <= "000";
      when "00010001" => z <= "000";
      when "00100001" => z <= "000";
      when "01000001" => z <= "000";
      when "10000001" => z <= "000";
      when "0000011-" => z <= "001";
      when "0000101-" => z <= "001";
      when "0001001-" => z <= "001";
      when "0010001-" => z <= "001";
      when "0100001-" => z <= "001";
      when "1000001-" => z <= "001";
      when "000011--" => z <= "010";
      when "000101--" => z <= "010";
      when "001001--" => z <= "010";
      when "010001--" => z <= "010";
      when "100001--" => z <= "010";
      when "00011---" => z <= "011";
      when "00101---" => z <= "011";
      when "01001---" => z <= "011";
      when "10001---" => z <= "011";
      when "0011----" => z <= "100";
      when "0101----" => z <= "100";
      when "1001----" => z <= "100";
      when "011-----" => z <= "101";
      when "101-----" => z <= "101";
      when "11------" => z <= "110";
      when others     => z <= "000";
    end case?;
  end process;
end;
```

Exercise 4.22

SystemVerilog

```
module thermometer(input  logic [2:0] a,
                   output logic [6:0] y);

  always_comb
    case (a)
       0: y = 7'b0000000;
       1: y = 7'b0000001;
       2: y = 7'b0000011;
       3: y = 7'b0000111;
       4: y = 7'b0001111;
       5: y = 7'b0011111;
       6: y = 7'b0111111;
       7: y = 7'b1111111;
    endcase
endmodule
```

VHDL

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

entity thermometer is
  port(a:    in  STD_LOGIC_VECTOR(2 downto 0);
       y:    out STD_LOGIC_VECTOR(6 downto 0));
end;

architecture synth of thermometer is
begin
  process(all) begin
    case a is
      when "000"  => y <= "0000000";
      when "001"  => y <= "0000001";
      when "010"  => y <= "0000011";
      when "011"  => y <= "0000111";
      when "100"  => y <= "0001111";
      when "101"  => y <= "0011111";
      when "110"  => y <= "0111111";
      when "111"  => y <= "1111111";
      when others => y <= "0000000";
    end case;
  end process;
end;
```

Exercise 4.23

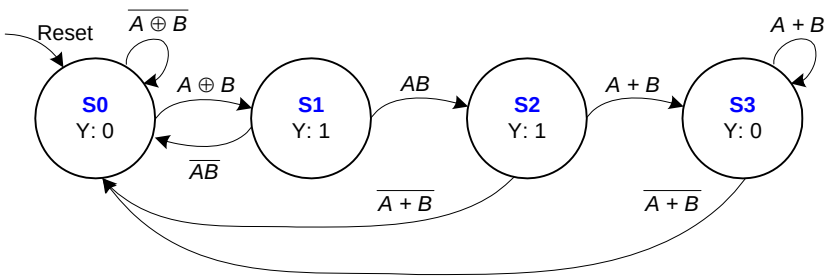
SystemVerilog

```
module month31days(input  logic [3:0] month,
                   output logic       y);

  always_comb
    casez (month)
       1:       y = 1'b1;
       2:       y = 1'b0;
       3:       y = 1'b1;
       4:       y = 1'b0;
       5:       y = 1'b1;
       6:       y = 1'b0;
       7:       y = 1'b1;
       8:       y = 1'b1;
       9:       y = 1'b0;
       10:      y = 1'b1;
       11:      y = 1'b0;
       12:      y = 1'b1;
       default: y = 1'b0;
    endcase
endmodule
```

VHDL

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

entity month31days is
  port(a:    in  STD_LOGIC_VECTOR(3 downto 0);
       y:    out STD_LOGIC);
end;

architecture synth of month31days is
begin
  process(all) begin
    case a is
      when X"1"   => y <= '1';
      when X"2"   => y <= '0';
      when X"3"   => y <= '1';
      when X"4"   => y <= '0';
      when X"5"   => y <= '1';
      when X"6"   => y <= '0';
      when X"7"   => y <= '1';
      when X"8"   => y <= '1';
      when X"9"   => y <= '0';
      when X"A"   => y <= '1';
      when X"B"   => y <= '0';
      when X"C"   => y <= '1';
      when others => y <= '0';
    end case;
  end process;
end;
```

Exercise 4.24



Exercise 4.25

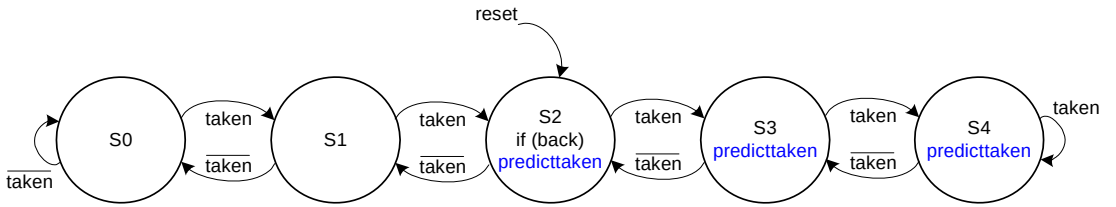


FIGURE 4.1 State transition diagram for Exercise 4.25

Exercise 4.26

SystemVerilog

```
module srlatch(input  logic s, r,
               output logic q, qbar);

  always_comb
    case ({s,r})
      2'b01: {q, qbar} = 2'b01;
      2'b10: {q, qbar} = 2'b10;
      2'b11: {q, qbar} = 2'b00;
    endcase
endmodule
```

VHDL

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

entity srlatch is
  port(s, r:      in  STD_LOGIC;
       q, qbar:   out STD_LOGIC);
end;

architecture synth of srlatch is
signal qqbar: STD_LOGIC_VECTOR(1 downto 0);
signal sr: STD_LOGIC_VECTOR(1 downto 0);
begin
  q <= qqbar(1);
  qbar <= qqbar(0);
  sr <= s & r;
  process(all) begin
    if s = '1' and r = '0'
      then qqbar <= "10";
    elsif s = '0' and r = '1'
      then qqbar <= "01";
    elsif s = '1' and r = '1'
      then qqbar <= "00";
    end if;
  end process;
end;
```

Exercise 4.27

SystemVerilog

```
module jkflop(input  logic j, k, clk,
              output logic q);

  always @(posedge clk)
    case ({j,k})
      2'b01: q <= 1'b0;
      2'b10: q <= 1'b1;
      2'b11: q <= ~q;
    endcase
endmodule
```

VHDL

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

entity jkflop is
  port(j, k, clk: in    STD_LOGIC;
       q:         inout STD_LOGIC);
end;

architecture synth of jkflop is
signal jk: STD_LOGIC_VECTOR(1 downto 0);
begin
  jk <= j & k;
  process(clk) begin
    if rising_edge(clk) then
      if j = '1' and k = '0'
        then q <= '1';
      elsif j = '0' and k = '1'
        then q <= '0';
      elsif j = '1' and k = '1'
        then q <= not q;
      end if;
    end if;
  end process;
end;
```

Exercise 4.28

SystemVerilog

```
module latch3_18(input  logic d, clk,
                 output logic q);

  logic n1, n2, clk_b;

  assign #1 n1 = clk & d;
  assign    clk_b = ~clk;
  assign #1 n2 = clk_b & q;
  assign #1 q = n1 | n2;
endmodule
```

VHDL

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

entity latch3_18 is
  port(d, clk: in    STD_LOGIC;
       q:      inout STD_LOGIC);
end;

architecture synth of latch3_18 is
signal n1, clk_b, n2: STD_LOGIC;
begin
  n1 <= (clk and d) after 1 ns;
  clk_b <= (not clk);
  n2 <= (clk_b and q) after 1 ns;
  q <= (n1 or n2) after 1 ns;
end;
```

This circuit is in error with any delay in the inverter.

Exercise 4.29

SystemVerilog

```
module trafficFSM(input  logic clk, reset, ta, tb,
                  output logic [1:0] la, lb);

  typedef enum logic [1:0] {S0, S1, S2, S3}
    statetype;
  statetype [1:0] state, nextstate;

  parameter green  = 2'b00;
  parameter yellow = 2'b01;
  parameter red    = 2'b10;

  // State Register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // Next State Logic
  always_comb
    case (state)
      S0: if (ta) nextstate = S0;
          else    nextstate = S1;
      S1:         nextstate = S2;
      S2: if (tb) nextstate = S2;
          else    nextstate = S3;
      S3:         nextstate = S0;
    endcase

  // Output Logic
  always_comb
    case (state)
      S0: {la, lb} = {green, red};
      S1: {la, lb} = {yellow, red};
      S2: {la, lb} = {red, green};
      S3: {la, lb} = {red, yellow};
    endcase
endmodule
```

VHDL

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

entity trafficFSM is
  port(clk, reset, ta, tb: in  STD_LOGIC;
       la, lb: inout STD_LOGIC_VECTOR(1 downto 0));
end;

architecture behave of trafficFSM is
  type statetype is (S0, S1, S2, S3);
  signal state, nextstate: statetype;
  signal lalb: STD_LOGIC_VECTOR(3 downto 0);
begin
  -- state register
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

 -- next state logic
  process(all) begin
    case state is
      when S0 => if ta then
                    nextstate <= S0;
                 else nextstate <= S1;
                 end if;
      when S1 => nextstate <= S2;
      when S2 => if tb then
                    nextstate <= S2;
                 else nextstate <= S3;
                 end if;
      when S3 => nextstate <= S0;
      when others =>  nextstate <= S0;
    end case;
  end process;

  -- output logic
  la <= lalb(3 downto 2);
  lb <= lalb(1 downto 0);
  process(all) begin
    case state is
      when S0 =>     lalb <= "0010";
      when S1 =>     lalb <= "0110";
      when S2 =>     lalb <= "1000";
      when S3 =>     lalb <= "1001";
      when others => lalb <= "1010";
    end case;
  end process;
end;
```

Exercise 4.30

Mode Module

SystemVerilog

```
module mode(input  logic clk, reset, p, r,
            output logic m);

  typedef enum logic {S0, S1} statetype;
  statetype state, nextstate;

  // State Register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // Next State Logic
  always_comb
    case (state)
      S0: if (p) nextstate = S1;
          else   nextstate = S0;
      S1: if (r) nextstate = S0;
          else   nextstate = S1;
    endcase

  // Output Logic
  assign m = state;
endmodule
```

VHDL

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

entity mode is
  port(clk, reset, p, r: in  STD_LOGIC;
       m:                out STD_LOGIC);
end;

architecture synth of mode is
  type statetype is (S0, S1);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

 -- next state logic
  process(all) begin
    case state is
      when S0 => if p then
                      nextstate <= S1;
                 else nextstate <= S0;
                 end if;
      when S1 => if r then
                      nextstate <= S0;
                 else nextstate <= S1;
                 end if;
      when others =>  nextstate <= S0;
    end case;
  end process;

  -- output logic
  m <= '1' when state = S1 else '0';
end;
```

(continued on next page)

Lights Module

SystemVerilog

```
module lights(input  logic clk, reset, ta, tb, m,
              output logic [1:0] la, lb);

  typedef enum logic [1:0] {S0, S1, S2, S3}
    statetype;

  statetype [1:0] state, nextstate;

  parameter green  = 2'b00;
  parameter yellow = 2'b01;
  parameter red    = 2'b10;

  // State Register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // Next State Logic
  always_comb
    case (state)
      S0: if (ta)     nextstate = S0;
          else        nextstate = S1;
      S1:             nextstate = S2;
      S2: if (tb | m) nextstate = S2;
          else        nextstate = S3;
      S3:             nextstate = S0;
    endcase

  // Output Logic
  always_comb
    case (state)
      S0: {la, lb} = {green, red};
      S1: {la, lb} = {yellow, red};
      S2: {la, lb} = {red, green};
      S3: {la, lb} = {red, yellow};
    endcase
endmodule
```

VHDL

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

entity lights is
  port(clk, reset, ta, tb, m: in  STD_LOGIC;
       la, lb: out STD_LOGIC_VECTOR(1 downto 0));
end;

architecture synth of lights is
  type statetype is (S0, S1, S2, S3);
  signal state, nextstate: statetype;
  signal lalb: STD_LOGIC_VECTOR(3 downto 0);
begin
  -- state register
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

 -- next state logic
  process(all) begin
    case state is
      when S0 => if ta then
                   nextstate <= S0;
                 else nextstate <= S1;
                 end if;
      when S1 =>      nextstate <= S2;
      when S2 => if ((tb or m) = '1') then
                   nextstate <= S2;
                 else nextstate <= S3;
                 end if;
      when S3 =>      nextstate <= S0;
      when others =>  nextstate <= S0;
    end case;
  end process;

  -- output logic
  la <= lalb(3 downto 2);
  lb <= lalb(1 downto 0);
  process(all) begin
    case state is
      when S0 =>     lalb <= "0010";
      when S1 =>     lalb <= "0110";
      when S2 =>     lalb <= "1000";
      when S3 =>     lalb <= "1001";
      when others => lalb <= "1010";
    end case;
  end process;
end;
```

(continued on next page)

Controller Module

SystemVerilog

```
module controller(input  logic clk, reset, p,
                           r, ta, tb,
                  output logic [1:0] la, lb);

  mode modefsm(clk, reset, p, r, m);
  lights lightsfsm(clk, reset, ta, tb, m, la, lb);
endmodule
```

VHDL

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

entity controller is
  port(clk, reset: in  STD_LOGIC;
       p, r, ta:   in  STD_LOGIC;
       tb:         in  STD_LOGIC;
       la, lb: out STD_LOGIC_VECTOR(1 downto 0));
end;

architecture struct of controller is
  component mode
    port(clk, reset, p, r: in  STD_LOGIC;
         m:                out STD_LOGIC);
  end component;
  component lights
    port(clk, reset, ta, tb, m: in  STD_LOGIC;
         la, lb: out STD_LOGIC_VECTOR(1 downto 0));
  end component;

begin
  modefsm:   mode   port map(clk, reset, p, r, m);
  lightsfsm: lights port map(clk, reset, ta, tb,
                             m, la, lb);
end;
```

Exercise 4.31

SystemVerilog

```
module fig3_42(input  logic clk, a, b, c, d,
               output logic x, y);

  logic n1, n2;
  logic areg, breg, creg, dreg;

  always_ff @(posedge clk) begin
    areg <= a;
    breg <= b;
    creg <= c;
    dreg <= d;
    x <= n2;
    y <= ~(dreg | n2);
  end

  assign n1 = areg & breg;
  assign n2 = n1 | creg;
endmodule
```

VHDL

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

entity fig3_42 is
  port(clk, a, b, c, d: in  STD_LOGIC;
       x, y:            out STD_LOGIC);
end;

architecture synth of fig3_40 is
  signal n1, n2, areg, breg, creg, dreg: STD_LOGIC;
begin
  process(clk) begin
    if rising_edge(clk) then
      areg <= a;
      breg <= b;
      creg <= c;
      dreg <= d;
      x <= n2;
      y <= not (dreg or n2);
    end if;
  end process;

  n1 <= areg and breg;
  n2 <= n1 or creg;
end;
```

Exercise 4.32

SystemVerilog

```
module fig3_69(input  logic clk, reset, a, b,
               output logic q);
  typedef enum logic [1:0] {S0, S1, S2} statetype;
  statetype [1:0] state, nextstate;

  // State Register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // Next State Logic
  always_comb
    case (state)
      S0: if (a) nextstate = S1;
          else   nextstate = S0;
      S1: if (b) nextstate = S2;
          else   nextstate = S0;
      S2:        nextstate = S0;
      default:   nextstate = S0;
    endcase

  // Output Logic
  assign q = state[1];
endmodule
```

VHDL

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

entity fig3_69 is
  port(clk, reset, a, b: in  STD_LOGIC;
       q:                out STD_LOGIC);
end;

architecture synth of fig3_69 is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

 -- next state logic
  process(all) begin
    case state is
      when S0 => if a then
                   nextstate <= S1;
                 else nextstate <= S0;
                 end if;
      when S1 => if b then
                   nextstate <= S2;
                 else nextstate <= S0;
                 end if;
      when S2 =>      nextstate <= S0;
      when others =>  nextstate <= S0;
    end case;
  end process;

  -- output logic
  q <= '1' when state = S2 else '0';
end;
```

Exercise 4.33

SystemVerilog

```
module fig3_70(input  logic clk, reset, a, b,
               output logic q);
  typedef enum logic [1:0] {S0, S1, S2} statetype;
  statetype [1:0] state, nextstate;

  // State Register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // Next State Logic
  always_comb
    case (state)
      S0: if (a)     nextstate = S1;
          else       nextstate = S0;
      S1: if (b)     nextstate = S2;
          else       nextstate = S0;
      S2: if (a & b) nextstate = S2;
          else       nextstate = S0;
      default:       nextstate = S0;
    endcase

  // Output Logic
  always_comb
    case (state)
      S0:            q = 0;
      S1:            q = 0;
      S2: if (a & b) q = 1;
          else       q = 0;
      default:       q = 0;
    endcase
endmodule
```

VHDL

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

entity fig3_70 is
  port(clk, reset, a, b: in  STD_LOGIC;
       q:                out STD_LOGIC);
end;

architecture synth of fig3_70 is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

 -- next state logic
  process(all) begin
    case state is
      when S0 => if a then
                      nextstate <= S1;
                 else nextstate <= S0;
                 end if;
      when S1 => if b then
                      nextstate <= S2;
                 else nextstate <= S0;
                 end if;
      when S2 => if (a = '1' and b = '1') then
                      nextstate <= S2;
                 else nextstate <= S0;
                 end if;
      when others =>  nextstate <= S0;
    end case;
  end process;

  -- output logic
  q <= '1' when ( (state = S2) and
                  (a = '1' and b = '1'))
           else '0';
end;
```

Exercise 4.34

SystemVerilog

```
module ex4_34(input  logic clk, reset, ta, tb,
              output logic [1:0] la, lb);
  typedef enum logic [2:0] {S0, S1, S2, S3, S4, S5}
    statetype;
  statetype [2:0] state, nextstate;

  parameter green  = 2'b00;
  parameter yellow = 2'b01;
  parameter red    = 2'b10;

  // State Register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // Next State Logic
  always_comb
    case (state)
      S0: if (ta) nextstate = S0;
          else    nextstate = S1;
      S1:         nextstate = S2;
      S2:         nextstate = S3;
      S3: if (tb) nextstate = S3;
          else    nextstate = S4;
      S4:         nextstate = S5;
      S5:         nextstate = S0;
    endcase

  // Output Logic
  always_comb
    case (state)
      S0: {la, lb} = {green, red};
      S1: {la, lb} = {yellow, red};
      S2: {la, lb} = {red, red};
      S3: {la, lb} = {red, green};
      S4: {la, lb} = {red, yellow};
      S5: {la, lb} = {red, red};
    endcase
endmodule
```

VHDL

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

entity ex4_34 is
  port(clk, reset, ta, tb: in  STD_LOGIC;
       la, lb: out STD_LOGIC_VECTOR(1 downto 0));
end;

architecture synth of ex4_34 is
  type statetype is (S0, S1, S2, S3, S4, S5);
  signal state, nextstate: statetype;
  signal lalb: STD_LOGIC_VECTOR(3 downto 0);
begin
  -- state register
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

 -- next state logic
  process(all) begin
    case state is
      when S0 => if ta = '1' then
                      nextstate <= S0;
                 else nextstate <= S1;
                 end if;
      when S1 =>      nextstate <= S2;
      when S2 =>      nextstate <= S3;
      when S3 => if tb = '1' then
                      nextstate <= S3;
                 else nextstate <= S4;
                 end if;
      when S4 =>      nextstate <= S5;
      when S5 =>      nextstate <= S0;
      when others =>  nextstate <= S0;
    end case;
  end process;

  -- output logic
  la <= lalb(3 downto 2);
  lb <= lalb(1 downto 0);
  process(all) begin
    case state is
      when S0 =>     lalb <= "0010";
      when S1 =>     lalb <= "0110";
      when S2 =>     lalb <= "1010";
      when S3 =>     lalb <= "1000";
      when S4 =>     lalb <= "1001";
      when S5 =>     lalb <= "1010";
      when others => lalb <= "1010";
    end case;
  end process;
end;
```

Exercise 4.35

SystemVerilog

```
module daughterfsm(input  logic clk, reset, a,
                   output logic smile);
  typedef enum logic [1:0] {S0, S1, S2, S3, S4}
    statetype;
  statetype [2:0] state, nextstate;

  // State Register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // Next State Logic
  always_comb
    case (state)
      S0: if (a) nextstate = S1;
          else   nextstate = S0;
      S1: if (a) nextstate = S2;
          else   nextstate = S0;
      S2: if (a) nextstate = S4;
          else   nextstate = S3;
      S3: if (a) nextstate = S1;
          else   nextstate = S0;
      S4: if (a) nextstate = S4;
          else   nextstate = S3;
      default:   nextstate = S0;
     endcase

  // Output Logic
  assign smile = ((state == S3) & a) |
                 ((state == S4) & ~a);
endmodule
```

VHDL

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

entity daughterfsm is
  port(clk, reset, a: in  STD_LOGIC;
       smile:         out STD_LOGIC);
end;

architecture synth of daughterfsm is
  type statetype is (S0, S1, S2, S3, S4);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

 -- next state logic
  process(all) begin
    case state is
      when S0 => if a then
                      nextstate <= S1;
                 else nextstate <= S0;
                 end if;
      when S1 => if a then
                      nextstate <= S2;
                 else nextstate <= S0;
                 end if;
      when S2 => if a then
                      nextstate <= S4;
                 else nextstate <= S3;
                 end if;
      when S3 => if a then
                      nextstate <= S1;
                 else nextstate <= S0;
                 end if;
      when S4 => if a then
                      nextstate <= S4;
                 else nextstate <= S3;
                 end if;
      when others =>  nextstate <= S0;
    end case;
  end process;

  -- output logic
  smile <= '1' when ( ((state = S3) and (a = '1')) or
                      ((state = S4) and (a = '0')) )
           else '0';
end;
```

Exercise 4.36

(starting on next page)

SystemVerilog

```
module ex4_36(input  logic clk, reset, n, d, q,
              output logic dispense,
                           return5, return10,
                           return2_10);
  typedef enum logic [3:0] {S0  = 4'b0000,
                            S5  = 4'b0001,
                            S10 = 4'b0010,
                            S25 = 4'b0011,
                            S30 = 4'b0100,
                            S15 = 4'b0101,
                            S20 = 4'b0110,
                            S35 = 4'b0111,
                            S40 = 4'b1000,
                            S45 = 4'b1001}
  statetype;
  statetype [3:0] state, nextstate;

  // State Register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // Next State Logic
  always_comb
    case (state)
      S0:       if (n) nextstate = S5;
           else if (d) nextstate = S10;
           else if (q) nextstate = S25;
           else        nextstate = S0;
      S5:       if (n) nextstate = S10;
           else if (d) nextstate = S15;
           else if (q) nextstate = S30;
           else        nextstate = S5;
      S10:      if (n) nextstate = S15;
           else if (d) nextstate = S20;
           else if (q) nextstate = S35;
           else        nextstate = S10;
      S25:             nextstate = S0;
      S30:             nextstate = S0;
      S15:      if (n) nextstate = S20;
           else if (d) nextstate = S25;
           else if (q) nextstate = S40;
           else        nextstate = S15;
      S20:      if (n) nextstate = S25;
           else if (d) nextstate = S30;
           else if (q) nextstate = S45;
           else        nextstate = S20;
      S35:             nextstate = S0;
      S40:             nextstate = S0;
      S45:             nextstate = S0;
      default:         nextstate = S0;
    endcase
```

VHDL

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

entity ex4_36 is
  port(clk, reset, n, d, q: in  STD_LOGIC;
       dispense, return5, return10: out STD_LOGIC;
       return2_10:                  out STD_LOGIC);
end;

architecture synth of ex4_36 is
  type statetype is (S0, S5, S10, S25, S30, S15, S20,
                     S35, S40, S45);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

 -- next state logic
  process(all) begin
    case state is
      when S0  =>
        if n    then nextstate <= S5;
        elsif d then nextstate <= S10;
        elsif q then nextstate <= S25;
        else         nextstate <= S0;
        end if;
      when S5  =>
        if n    then nextstate <= S10;
        elsif d then nextstate <= S15;
        elsif q then nextstate <= S30;
        else         nextstate <= S5;
        end if;
      when S10 =>
        if n    then nextstate <= S15;
        elsif d then nextstate <= S20;
        elsif q then nextstate <= S35;
        else         nextstate <= S10;
        end if;
      when S25 =>    nextstate <= S0;
      when S30 =>    nextstate <= S0;
      when S15 =>
        if n    then nextstate <= S20;
        elsif d then nextstate <= S25;
        elsif q then nextstate <= S40;
        else         nextstate <= S15;
        end if;
      when S20 =>
        if n    then nextstate <= S25;
        elsif d then nextstate <= S30;
        elsif q then nextstate <= S45;
        else         nextstate <= S20;
        end if;
      when S35 =>    nextstate <= S0;
      when S40 =>    nextstate <= S0;
      when S45 =>    nextstate <= S0;
      when others => nextstate <= S0;
    end case;
  end process;
```

(continued from previous page)

SystemVerilog

```
  // Output Logic
  assign dispense   = (state == S25) |
                      (state == S30) |
                      (state == S35) |
                      (state == S40) |
                      (state == S45);
  assign return5    = (state == S30) |
                      (state == S40);
  assign return10   = (state == S35) |
                      (state == S40);
  assign return2_10 = (state == S45);
endmodule
```

VHDL

```
  -- output logic
  dispense   <= '1' when ((state = S25) or
                          (state = S30) or
                          (state = S35) or
                          (state = S40) or
                          (state = S45))
                else '0';
  return5    <= '1' when ((state = S30) or
                          (state = S40))
                else '0';
  return10   <= '1' when ((state = S35) or
                          (state = S40))
                else '0';
  return2_10 <= '1' when (state = S45)
                else '0';
end;
```

Exercise 4.37

SystemVerilog

```
module ex4_37(input  logic       clk, reset,
              output logic [2:0] q);
  typedef enum logic [2:0] {S0 = 3'b000,
                            S1 = 3'b001,
                            S2 = 3'b011,
                            S3 = 3'b010,
                            S4 = 3'b110,
                            S5 = 3'b111,
                            S6 = 3'b101,
                            S7 = 3'b100}
    statetype;

  statetype [2:0] state, nextstate;

  // State Register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // Next State Logic
  always_comb
    case (state)
      S0: nextstate = S1;
      S1: nextstate = S2;
      S2: nextstate = S3;
      S3: nextstate = S4;
      S4: nextstate = S5;
      S5: nextstate = S6;
      S6: nextstate = S7;
      S7: nextstate = S0;
    endcase

  // Output Logic
  assign q = state;
endmodule
```

VHDL

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

entity ex4_37 is
  port(clk:   in  STD_LOGIC;
       reset: in  STD_LOGIC;
       q:     out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture synth of ex4_37 is
  signal state:     STD_LOGIC_VECTOR(2 downto 0);
  signal nextstate: STD_LOGIC_VECTOR(2 downto 0);
begin
  -- state register
  process(clk, reset) begin
    if reset then state <= "000";
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

  -- next state logic
  process(all) begin
    case state is
      when "000"  => nextstate <= "001";
      when "001"  => nextstate <= "011";
      when "011"  => nextstate <= "010";
      when "010"  => nextstate <= "110";
      when "110"  => nextstate <= "111";
      when "111"  => nextstate <= "101";
      when "101"  => nextstate <= "100";
      when "100"  => nextstate <= "000";
      when others => nextstate <= "000";
    end case;
  end process;

  -- output logic
  q <= state;
end;
```

Exercise 4.38

SystemVerilog

```
module ex4_38(input  logic       clk, reset, up,
              output logic [2:0] q);

  typedef enum logic [2:0] {
    S0 = 3'b000,
    S1 = 3'b001,
    S2 = 3'b011,
    S3 = 3'b010,
    S4 = 3'b110,
    S5 = 3'b111,
    S6 = 3'b101,
    S7 = 3'b100} statetype;
  statetype [2:0] state, nextstate;

  // State Register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // Next State Logic
  always_comb
    case (state)
      S0: if (up) nextstate = S1;
          else    nextstate = S7;
      S1: if (up) nextstate = S2;
          else    nextstate = S0;
      S2: if (up) nextstate = S3;
          else    nextstate = S1;
      S3: if (up) nextstate = S4;
          else    nextstate = S2;
      S4: if (up) nextstate = S5;
          else    nextstate = S3;
      S5: if (up) nextstate = S6;
          else    nextstate = S4;
      S6: if (up) nextstate = S7;
          else    nextstate = S5;
      S7: if (up) nextstate = S0;
          else    nextstate = S6;
    endcase

  // Output Logic
  assign q = state;
endmodule
```

(continued on next page)

VHDL

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

entity ex4_38 is
  port(clk:   in  STD_LOGIC;
       reset: in  STD_LOGIC;
       up:    in  STD_LOGIC;
       q:     out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture synth of ex4_38 is
  signal state:     STD_LOGIC_VECTOR(2 downto 0);
  signal nextstate: STD_LOGIC_VECTOR(2 downto 0);
begin
  -- state register
  process(clk, reset) begin
    if reset then state <= "000";
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

  -- next state logic
  process(all) begin
    case state is
      when "000"  => if up then
                       nextstate <= "001";
                     else
                       nextstate <= "100";
                     end if;
      when "001"  => if up then
                       nextstate <= "011";
                     else
                       nextstate <= "000";
                     end if;
      when "011"  => if up then
                       nextstate <= "010";
                     else
                       nextstate <= "001";
                     end if;
      when "010"  => if up then
                       nextstate <= "110";
                     else
                       nextstate <= "011";
                     end if;
```

(continued from previous page)

VHDL

```
      when "110"  => if up then
                       nextstate <= "111";
                     else
                       nextstate <= "010";
                     end if;
      when "111"  => if up then
                       nextstate <= "101";
                     else
                       nextstate <= "110";
                     end if;
      when "101"  => if up then
                       nextstate <= "100";
                     else
                       nextstate <= "111";
                     end if;
      when "100"  => if up then
                       nextstate <= "000";
                     else
                       nextstate <= "101";
                     end if;
      when others => nextstate <= "000";
    end case;
  end process;

  -- output logic
  q <= state;
end;
```

Exercise 4.39

Option 1

SystemVerilog

```
module ex4_39(input  logic clk, reset, a, b,
              output logic z);
  typedef enum logic [1:0] {S0, S1, S2, S3}
    statetype;
  statetype [1:0] state, nextstate;

  // State Register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // Next State Logic
  always_comb
    case (state)
      S0: case ({b,a})
            2'b00: nextstate = S0;
            2'b01: nextstate = S3;
            2'b10: nextstate = S0;
            2'b11: nextstate = S1;
          endcase
      S1: case ({b,a})
            2'b00: nextstate = S0;
            2'b01: nextstate = S3;
            2'b10: nextstate = S2;
            2'b11: nextstate = S1;
          endcase
      S2: case ({b,a})
            2'b00: nextstate = S0;
            2'b01: nextstate = S3;
            2'b10: nextstate = S2;
            2'b11: nextstate = S1;
          endcase
      S3: case ({b,a})
            2'b00: nextstate = S0;
            2'b01: nextstate = S3;
            2'b10: nextstate = S2;
            2'b11: nextstate = S1;
          endcase
      default:     nextstate = S0;
    endcase

  // Output Logic
  always_comb
    case (state)
      S0:      z = a & b;
      S1:      z = a | b;
      S2:      z = a & b;
      S3:      z = a | b;
      default: z = 1'b0;
    endcase
endmodule
```

VHDL

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

entity ex4_39 is
  port(clk:   in  STD_LOGIC;
       reset: in  STD_LOGIC;
       a, b:  in  STD_LOGIC;
       z:     out STD_LOGIC);
end;

architecture synth of ex4_39 is
  type statetype is (S0, S1, S2, S3);
  signal state, nextstate: statetype;
  signal ba: STD_LOGIC_VECTOR(1 downto 0);
begin
  -- state register
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

  -- next state logic
  ba <= b & a;
  process(all) begin
    case state is
      when S0  =>
        case (ba) is
          when "00"   => nextstate <= S0;
          when "01"   => nextstate <= S3;
          when "10"   => nextstate <= S0;
          when "11"   => nextstate <= S1;
          when others => nextstate <= S0;
        end case;
      when S1  =>
        case (ba) is
          when "00"   => nextstate <= S0;
          when "01"   => nextstate <= S3;
          when "10"   => nextstate <= S2;
          when "11"   => nextstate <= S1;
          when others => nextstate <= S0;
        end case;
      when S2  =>
        case (ba) is
          when "00"   => nextstate <= S0;
          when "01"   => nextstate <= S3;
          when "10"   => nextstate <= S2;
          when "11"   => nextstate <= S1;
          when others => nextstate <= S0;
        end case;
      when S3  =>
        case (ba) is
          when "00"   => nextstate <= S0;
          when "01"   => nextstate <= S3;
          when "10"   => nextstate <= S2;
          when "11"   => nextstate <= S1;
          when others => nextstate <= S0;
        end case;
      when others     =>   nextstate <= S0;
    end case;
  end process;
```

(continued from previous page)

VHDL

```
  -- output logic
  process(all) begin
    case state is
      when S0     => if (a = '1' and b = '1')
                     then z <= '1';
                     else z <= '0';
                     end if;
      when S1     => if (a = '1' or b = '1')
                     then z <= '1';
                     else z <= '0';
                     end if;
      when S2     => if (a = '1' and b = '1')
                     then z <= '1';
                     else z <= '0';
                     end if;
      when S3     => if (a = '1' or b = '1')
                     then z <= '1';
                     else z <= '0';
                     end if;
      when others => z <= '0';
    end case;
  end process;
end;
```

Option 2

SystemVerilog

```
module ex4_37(input  logic clk, a, b,
              output logic z);

  logic aprev;

  // State Register
  always_ff @(posedge clk)
    aprev <= a;

  assign z = b ? (aprev | a) : (aprev & a);
endmodule
```

VHDL

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

entity ex4_37 is
  port(clk:   in  STD_LOGIC;
       a, b:  in  STD_LOGIC;
       z:     out STD_LOGIC);
end;

architecture synth of ex4_37 is
  signal aprev, n1and, n2or: STD_LOGIC;
begin
  -- state register
  process(clk) begin
    if rising_edge(clk) then
      aprev <= a;
    end if;
  end process;


  z <= (a or aprev) when b = '1' else
       (a and aprev);
end;
```

Exercise 4.40

SystemVerilog

```
module fsm_y(input  clk, reset, a,
             output y);
  typedef enum logic [1:0] {S0=2'b00, S1=2'b01,
    S11=2'b11} statetype;
  statetype [1:0] state, nextstate;

  // State Register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // Next State Logic
  always_comb
    case (state)
      S0:  if (a) nextstate = S1;
           else   nextstate = S0;
      S1:  if (a) nextstate = S11;
           else   nextstate = S0;
      S11:        nextstate = S11;
      default:    nextstate = S0;
    endcase

  // Output Logic
  assign y = state[1];
endmodule
```

VHDL

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

entity fsm_y is
  port(clk, reset, a: in  STD_LOGIC;
       y:             out STD_LOGIC);
end;

architecture synth of fsm_y is
  type statetype is (S0, S1, S11);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

 -- next state logic
  process(all) begin
    case state is
      when S0 =>  if a then
                       nextstate <= S1;
                  else nextstate <= S0;
                  end if;
      when S1 =>  if a then
                       nextstate <= S11;
                  else nextstate <= S0;
                  end if;
      when S11 =>      nextstate <= S11;
      when others =>   nextstate <= S0;
    end case;
  end process;

  -- output logic
  y <= '1' when (state = S11) else '0';
end;
```

(continued on next page)

(continued from previous page)

SystemVerilog

```
module fsm_x(input  logic clk, reset, a,
             output logic x);
  typedef enum logic [1:0] {S0, S1, S2, S3}
    statetype;
  statetype [1:0] state, nextstate;

  // State Register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // Next State Logic
  always_comb
    case (state)
      S0: if   (a) nextstate = S1;
          else     nextstate = S0;
      S1: if   (a) nextstate = S2;
          else     nextstate = S1;
      S2: if   (a) nextstate = S3;
          else     nextstate = S2;
      S3:          nextstate = S3;
    endcase

  // Output Logic
  assign x = (state == S3);
endmodule
```

VHDL

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

entity fsm_x is
  port(clk, reset, a: in  STD_LOGIC;
       x:             out STD_LOGIC);
end;

architecture synth of fsm_x is
  type statetype is (S0, S1, S2, S3);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

 -- next state logic
  process(all) begin
    case state is
      when S0 =>  if a then
                       nextstate <= S1;
                  else nextstate <= S2;
                  end if;
      when S1 =>  if a then
                       nextstate <= S2;
                  else nextstate <= S1;
                  end if;
      when S2 =>  if a then
                       nextstate <= S3;
                  else nextstate <= S2;
                  end if;
      when S3 =>       nextstate <= S3;
      when others =>   nextstate <= S0;
    end case;
  end process;

  -- output logic
  x <= '1' when (state = S3) else '0';
end;
```

Exercise 4.41

SystemVerilog

```
module ex4_41(input  logic clk, start, a,
              output logic q);
  typedef enum logic [1:0] {S0, S1, S2, S3}
    statetype;
  statetype [1:0] state, nextstate;

  // State Register
  always_ff @(posedge clk, posedge start)
    if (start) state <= S0;
    else       state <= nextstate;

  // Next State Logic
  always_comb
    case (state)
      S0: if   (a) nextstate = S1;
          else     nextstate = S0;
      S1: if   (a) nextstate = S2;
          else     nextstate = S3;
      S2: if   (a) nextstate = S2;
          else     nextstate = S3;
      S3: if   (a) nextstate = S2;
          else     nextstate = S3;
    endcase

  // Output Logic
  assign q = state[0];
endmodule
```

VHDL

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

entity ex4_41 is
  port(clk, start, a: in  STD_LOGIC;
       q:             out STD_LOGIC);
end;

architecture synth of ex4_41 is
  type statetype is (S0, S1, S2, S3);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, start) begin
    if start then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

 -- next state logic
  process(all) begin
    case state is
      when S0 =>  if a then
                       nextstate <= S1;
                  else nextstate <= S0;
                  end if;
      when S1 =>  if a then
                       nextstate <= S2;
                  else nextstate <= S3;
                  end if;
      when S2 =>  if a then
                       nextstate <= S2;
                  else nextstate <= S3;
                  end if;
      when S3 =>  if a then
                       nextstate <= S2;
                  else nextstate <= S3;
                  end if;
      when others =>   nextstate <= S0;
    end case;
  end process;

  -- output logic
  q <= '1' when ((state = S1) or (state = S3))
       else '0';
end;
```

Exercise 4.42

SystemVerilog

```
module ex4_42(input  logic clk, reset, x,
              output logic q);
  typedef enum logic [1:0] {S0, S1, S2, S3}
    statetype;
  statetype [1:0] state, nextstate;

  // State Register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S00;
    else       state <= nextstate;

  // Next State Logic
  always_comb
    case (state)
      S00: if (x) nextstate = S11;
           else   nextstate = S01;
      S01: if (x) nextstate = S10;
           else   nextstate = S00;
      S10:        nextstate = S01;
      S11:        nextstate = S01;
    endcase

  // Output Logic
  assign q = state[0] | state[1];
endmodule
```

VHDL

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

entity ex4_42 is
  port(clk, reset, x: in  STD_LOGIC;
       q:             out STD_LOGIC);
end;

architecture synth of ex4_42 is
  type statetype is (S00, S01, S10, S11);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset then state <= S00;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

 -- next state logic
  process(all) begin
    case state is
      when S00 =>  if x then
                       nextstate <= S11;
                   else nextstate <= S01;
                   end if;
      when S01 =>  if x then
                       nextstate <= S10;
                   else nextstate <= S00;
                   end if;
      when S10 =>      nextstate <= S01;
      when S11 =>      nextstate <= S01;
      when others =>   nextstate <= S00;
    end case;
  end process;

  -- output logic
  q <= '0' when (state = S00) else '1';
end;
```

Exercise 4.43

SystemVerilog

```
module ex4_43(input  clk, reset, a,
              output q);
  typedef enum logic [1:0] {S0, S1, S2} statetype;
  statetype [1:0] state, nextstate;

  // State Register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // Next State Logic
  always_comb
    case (state)
      S0: if (a) nextstate = S1;
          else   nextstate = S0;
      S1: if (a) nextstate = S2;
          else   nextstate = S0;
      S2: if (a) nextstate = S2;
          else   nextstate = S0;
      default:   nextstate = S0;
    endcase

  // Output Logic
  assign q = state[1];
endmodule
```

VHDL

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

entity ex4_43 is
  port(clk, reset, a: in  STD_LOGIC;
       q:             out STD_LOGIC);
end;

architecture synth of ex4_43 is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

 -- next state logic
  process(all) begin
    case state is
      when S0 =>  if a then
                       nextstate <= S1;
                  else nextstate <= S0;
                  end if;
      when S1 =>  if a then
                       nextstate <= S2;
                  else nextstate <= S0;
                  end if;
      when S2 =>  if a then
                       nextstate <= S2;
                  else nextstate <= S0;
                  end if;
      when others =>   nextstate <= S0;
    end case;
  end process;

  -- output logic
  q <= '1' when (state = S2) else '0';
end;
```

Exercise 4.44

(a)

SystemVerilog

```
module ex4_44a(input logic clk, a, b, c, d,
               output logic q);

  logic areg, breg, creg, dreg;

  always_ff @(posedge clk)
    begin
      areg <= a;
      breg <= b;
      creg <= c;
      dreg <= d;
      q    <= ((areg ^ breg) ^ creg) ^ dreg;
    end
endmodule
```

VHDL

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

entity ex4_44a is
  port(clk, a, b, c, d: in  STD_LOGIC;
       q:               out STD_LOGIC);
end;

architecture synth of ex4_44a is
  signal areg, breg, creg, dreg: STD_LOGIC;
begin
  process(clk) begin
    if rising_edge(clk) then
      areg <= a;
      breg <= b;
      creg <= c;
      dreg <= d;
      q <= ((areg xor breg) xor creg) xor dreg;
    end if;
  end process;
end;
```

(d)

SystemVerilog

```
module ex4_44d(input logic clk, a, b, c, d,
               output logic q);

  logic areg, breg, creg, dreg;

  always_ff @(posedge clk)
    begin
      areg <= a;
      breg <= b;
      creg <= c;
      dreg <= d;
      q    <= (areg ^ breg) ^ (creg ^ dreg);
    end
endmodule
```

VHDL

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

entity ex4_44d is
  port(clk, a, b, c, d: in  STD_LOGIC;
       q:               out STD_LOGIC);
end;

architecture synth of ex4_44d is
  signal areg, breg, creg, dreg: STD_LOGIC;
begin
  process(clk) begin
    if rising_edge(clk) then
      areg <= a;
      breg <= b;
      creg <= c;
      dreg <= d;
      q <= (areg xor breg) xor (creg xor dreg);
    end if;
  end process;
end;
```

Exercise 4.45

SystemVerilog

```
module ex4_45(input  logic       clk, c,
              input  logic [1:0] a, b,
              output logic [1:0] s);

  logic  [1:0] areg, breg;
  logic        creg;
  logic [1:0]  sum;
  logic        cout;

  always_ff @(posedge clk)
    {areg, breg, creg, s} <= {a, b, c, sum};

  fulladder fulladd1(areg[0], breg[0], creg,
                     sum[0], cout);
  fulladder fulladd2(areg[1], breg[1], cout,
                     sum[1], );
endmodule
```

VHDL

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

entity ex4_45 is
  port(clk, c: in  STD_LOGIC;
       a, b:   in  STD_LOGIC_VECTOR(1 downto 0);
       s:      out STD_LOGIC_VECTOR(1 downto 0));
end;

architecture synth of ex4_45 is
  component fulladder is
    port(a, b, cin:  in  STD_LOGIC;
         s, cout:    out STD_LOGIC);
  end component;
  signal creg: STD_LOGIC;
  signal areg, breg, cout: STD_LOGIC_VECTOR(1 downto
0);
  signal sum:        STD_LOGIC_VECTOR(1 downto 0);
begin
  process(clk) begin
    if rising_edge(clk) then
      areg <= a;
      breg <= b;
      creg <= c;
      s <= sum;
    end if;
  end process;

  fulladd1: fulladder
   port map(areg(0), breg(0), creg, sum(0), cout(0));
  fulladd2: fulladder
       port  map(areg(1),  breg(1),  cout(0),  sum(1),
cout(1));
end;
```

Exercise 4.46

A signal declared as `tri` can have multiple drivers.

Exercise 4.47

SystemVerilog

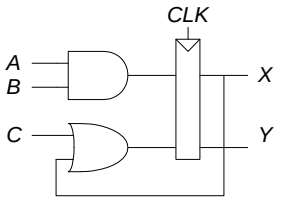
```
module syncbad(input  logic clk,
               input  logic d,
               output logic q);

  logic n1;

  always_ff @(posedge clk)
    begin
      q  <= n1;// nonblocking
      n1 <= d; // nonblocking
   end
endmodule
```

VHDL

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

entity syncbad is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC;
       q:   out STD_LOGIC);
end;

architecture bad of syncbad is
begin
  process(clk)
    variable n1: STD_LOGIC;
  begin
    if rising_edge(clk) then
      q  <= n1; -- nonblocking
      n1 <= d;  -- nonblocking
    end if;
  end process;
end;
```

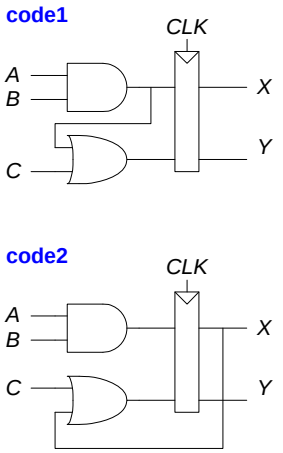
Exercise 4.48

They have the same function.



Exercise 4.49

They do not have the same function.



Exercise 4.50

(a) Problem: Signal `d` is not included in the sensitivity list of the always statement. Correction shown below (changes are in bold).

```
module latch(input  logic     clk,
             input  logic [3:0] d,
             output logic [3:0] q);

   always_latch
      if (clk) q <= d;
endmodule
```

(b) Problem: Signal `b` is not included in the sensitivity list of the always statement. Correction shown below (changes are in bold).

```
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);

   always_comb
      begin
         y1 = a & b;
         y2 = a | b;
         y3 = a ^ b;
         y4 = ~(a & b);
         y5 = ~(a | b);
      end
endmodule
```

(c) Problem: The sensitivity list should not include the word "`posedge`". The `always` statement needs to respond to any changes in `s`, not just the positive edge. Signals `d0` and `d1` need to be added to the sensitivity list. Also, the always statement implies combinational logic, so blocking assignments should be used.

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output logic [3:0] y);

   always_comb
      if (s) y = d1;
      else   y = d0;
endmodule
```

(d) Problem: This module will actually work in this case, but it's good practice to use nonblocking assignments in `always` statements that describe sequential logic. Because the `always` block has more than one statement in it, it requires a begin and end.

```
module twoflops(input  logic clk,
                input  logic d0, d1,
                output logic q0, q1);

   always_ff @(posedge clk)
   begin
      q1 <= d1;           // nonblocking assignment
      q0 <= d0;           // nonblocking assignment
   end
endmodule
```

(e) Problem: out1 and out2 are not assigned for all cases. Also, it would be best to separate the next state logic from the state register. reset is also missing in the input declaration.

```
module FSM(input  logic clk,
           input  logic reset,
           input  logic a,
           output logic out1, out2);

   logic  state, nextstate;

   // state register
   always_ff @(posedge clk, posedge reset)
     if (reset)
       state <= 1'b0;
     else
       state <= nextstate;

   // next state logic
   always_comb
     case (state)
       1'b0: if (a) nextstate = 1'b1;
             else nextstate = 1'b0;
       1'b1: if (~a) nextstate = 1'b0;
             else nextstate = 1'b1;
     endcase

   // output logic (combinational)
   always_comb
      if (state == 0) {out1, out2} = {1'b1, 1'b0};
      else            {out1, out2} = {1'b0, 1'b1};
endmodule
```

(f) Problem: A priority encoder is made from combinational logic, so the HDL must completely define what the outputs are for all possible input combinations. So, we must add an else statement at the end of the `always` block.

```
module priority(input  logic [3:0] a,
```

```
               output logic [3:0] y);

   always_comb
      if      (a[3]) y = 4'b1000;
      else if (a[2]) y = 4'b0100;
      else if (a[1]) y = 4'b0010;
      else if (a[0]) y = 4'b0001;
      else           y = 4'b0000;
endmodule
```

(g) Problem: the next state logic block has no default statement. Also, state S2 is missing the S.

```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic out);

   logic  [1:0] state, nextstate;

   parameter S0 = 2'b00;
   parameter S1 = 2'b01;
   parameter S2 = 2'b10;

   // State Register
   always_ff @(posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;

   // Next State Logic
   always_comb
      case (state)
         S0:      nextstate = S1;
         S1:      nextstate = S2;
         S2:      nextstate = S0;
         default: nextstate = S0;
      endcase

   // Output Logic
   assign out = (state == S2);
endmodule
```

(h) Problem: the ~ is missing on the first tristate.

```
module mux2tri(input  logic [3:0] d0, d1,
               input  logic       s,
               output logic [3:0] y);

   tristate t0(d0, ~s, y);
   tristate t1(d1, s, y);

endmodule
```

(i) Problem: an output, in this case, `q`, cannot be assigned in multiple always or assignment statements. Also, the flip-flop does not include an enable, so it should not be named `floprsen`.

```
module floprs(input  logic       clk,
              input  logic       reset,
              input  logic       set,
              input  logic [3:0] d,
              output logic [3:0] q);

   always_ff @(posedge clk, posedge reset, posedge set)
```

```
      if (reset)    q <= 0;
      else if (set) q <= 1;
      else          q <= d;
endmodule
```

(j) Problem: this is a combinational module, so nonconcurrent (blocking) assignment statements (=) should be used in the always statement, not concurrent assignment statements (<=). Also, it's safer to use always @(*) for combinational logic to make sure all the inputs are covered.

```
module and3(input  logic a, b, c,
            output logic y);

   logic tmp;

   always_comb
     begin
      tmp = a & b;
      y   = tmp & c;
     end
endmodule
```

Exercise 4.51

It is necessary to write

```
q <= '1' when state = S0 else '0';
```

rather than simply

```
q <= (state = S0);
```

because the result of the comparison `(state = S0)` is of type `Boolean` (`true` and `false`) and q must be assigned a value of type STD_LOGIC (`'1'` and `'0'`).

Exercise 4.52

(a) **Problem:** both clk and d must be in the process statement.

```
architecture synth of latch is
begin
  process(clk, d) begin
    if clk = '1' then q <= d;
    end if;
  end process;
end;
```

(b) **Problem:** both a and b must be in the process statement.

```
architecture proc of gates is
begin
  process(all) begin
    y1 <= a and b;
    y2 <= a or b;
    y3 <= a xor b;
```

```
    y4 <= a nand b;
    y5 <= a nor b;
  end process;
end;
```

(c) **Problem:** The end if and end process statements are missing.

```
architecture synth of flop is
begin
  process(clk)
    if clk'event and clk = '1' then
      q <= d;
    end if;
  end process;
end;
```

(d) **Problem:** The final else statement is missing. Also, it's better to use "process(all)" instead of "process(a)"

```
architecture synth of priority is
begin
  process(all) begin
    if    a(3) = '1' then y <= "1000";
    elsif a(2) = '1' then y <= "0100";
    elsif a(1) = '1' then y <= "0010";
    elsif a(0) = '1' then y <= "0001";
    else                  y <= "0000";
    end if;
  end process;
end;
```

(e) **Problem:** The `default` statement is missing in the nextstate case statement. Also, it's better to use the updated statements: "if reset", "rising_edge(clk)", and "process(all)".

```
architecture synth of divideby3FSM is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

   process(all) begin
    case state is
      when S0 =>      nextstate <= S1;
      when S1 =>      nextstate <= S2;
      when S2 =>      nextstate <= S0;
      when others =>  nextstate <= S0;
    end case;
  end process;

  q <= '1' when state = S0 else '0';
end;
```

(f) **Problem:** The select signal on tristate instance t0 must be inverted. However, VHDL does not allow logic to be performed within an instance declaration. Thus, an internal signal, `sbar`, must be declared.

```
architecture struct of mux2 is
  component tristate
```

```
    port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
         en: in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal sbar: STD_LOGIC;
begin
  sbar <= not s;
  t0: tristate port map(d0, sbar, y);
  t1: tristate port map(d1, s, y);
end;
```

(g) **Problem:** The q output cannot be assigned in two process or assignment statements. Also, it's better to use the updated statements: "if reset", and "rising_edge(clk)".

```
architecture asynchronous of flopr is
begin
  process(clk, reset, set) begin
    if reset then
      q <= '0';
    elsif set then
      q <= '1';
    elsif rising_edge(clk) then
      q <= d;
    end if;
  end process;
end;
```

Question 4.1

SystemVerilog

```
assign result = sel ? data : 32'b0;
```

VHDL

```
result <= data when sel = '1' else X"00000000";
```

Question 4.2

HDLs support *blocking* and *nonblocking assignments* in an `always` / `process` statement. A group of blocking assignments are evaluated in the order they appear in the code, just as one would expect in a standard programming

language. A group of nonblocking assignments are evaluated concurrently; all of the statements are evaluated before any of the left hand sides are updated.

SystemVerilog

In a SystemVerilog `always` statement, `=` indicates a blocking assignment and `<=` indicates a nonblocking assignment.

Do not confuse either type with continuous assignment using the `assign` statement. `assign` statements are normally used outside `always` statements and are also evaluated concurrently.

VHDL

In a VHDL `process` statement, `:=` indicates a blocking assignment and `<=` indicates a nonblocking assignment (also called a concurrent assignment). This is the first section where `:=` is introduced.

Nonblocking assignments are made to outputs and to signals. Blocking assignments are made to variables, which are declared in `process` statements (see the next example).

`<=` can also appear outside `process` statements, where it is also evaluated concurrently.

See HDL Examples 4.24 and 4.29 for comparisons of blocking and nonblocking assignments. Blocking and nonblocking assignment guidelines are given on page 206.

Question 4.3

The SystemVerilog statement performs the bit-wise AND of the 16 least significant bits of data with 0xC820. It then ORs these 16 bits to produce the 1-bit result.

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:

$$t_{CLA} = t_{\text{pg}} + t_{\text{pg\_block}} + \left(\frac{N}{k} - 1\right) t_{\text{AND\_OR}} + kt_{\text{FA}}$$

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

(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:

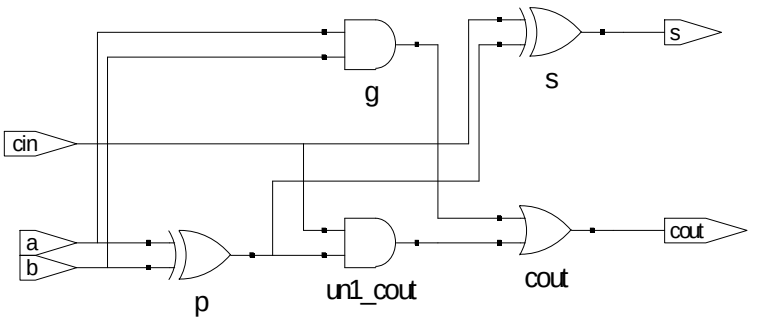


FIGURE 5.1 Full adder implementation

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:

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

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

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

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,

$$P_{\text{dynamic\_ripple}} = \frac{1}{2}CV_{DD}^2 f = \frac{1}{2}(6.4\ \text{pF})(1.2\ \text{V})^2(100\text{MHz}) = 0.461\ mW$$

$$P_{\text{dynamic\_CLA}} = \frac{1}{2}CV_{DD}^2 f = \frac{1}{2}(12.48\ \text{pF})(1.2\ \text{V})^2(100\text{MHz}) = 0.899\ mW$$

.

	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 previouspage*)

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

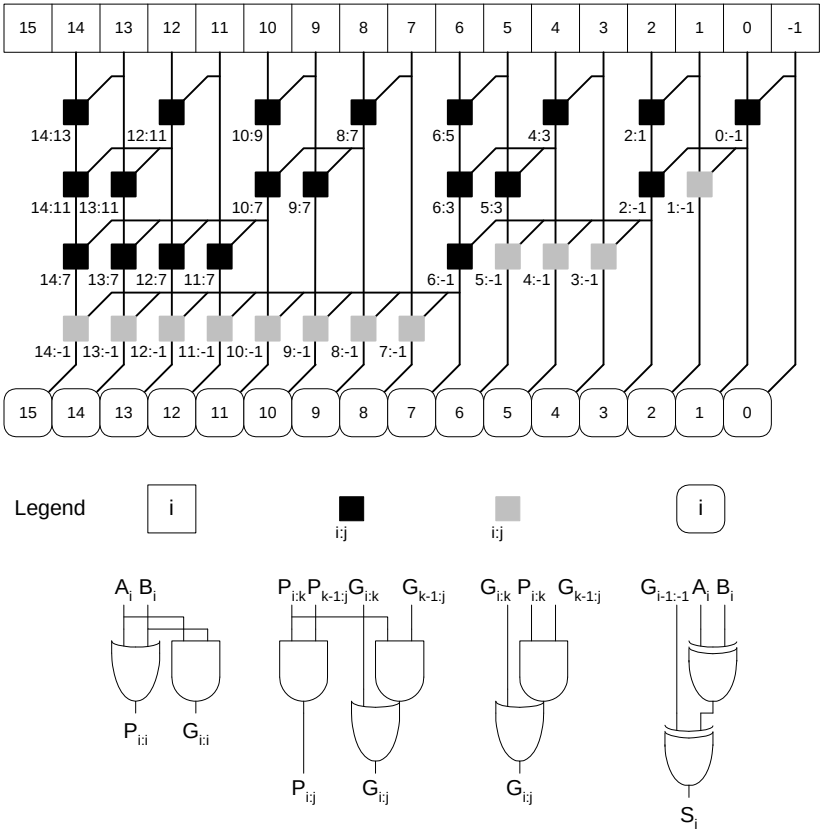


FIGURE 5.2 16-bit prefix adder with “gray cells”

Exercise 5.6

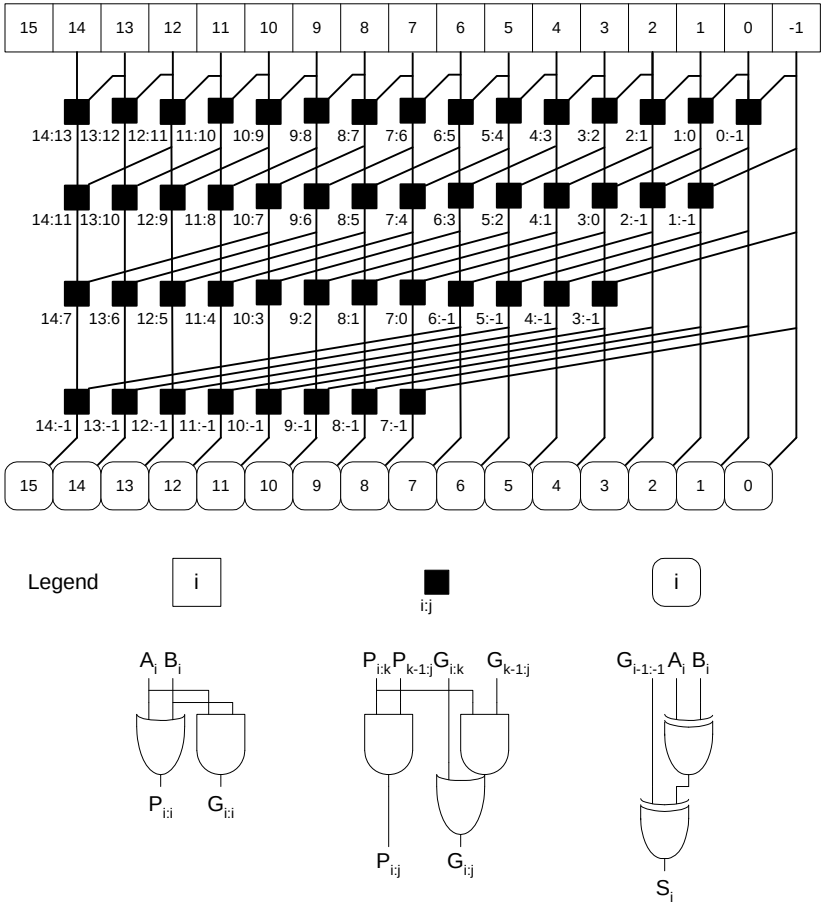


FIGURE 5.3 Schematic of a 16-bit Kogge-Stone adder

Exercise 5.7

(a) We show an 8-bit priority circuit in Figure 5.4. In the figure $X_7 = \overline{A}_7$, $X_{7:6} = \overline{A}_7\overline{A}_6$, $X_{7:5} = \overline{A}_7\overline{A}_6\overline{A}_5$, 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:

$t_{pd\_priority} = (\log_2 N+1)t_{pd\_AND2} + t_{pd\_ORN/2}$

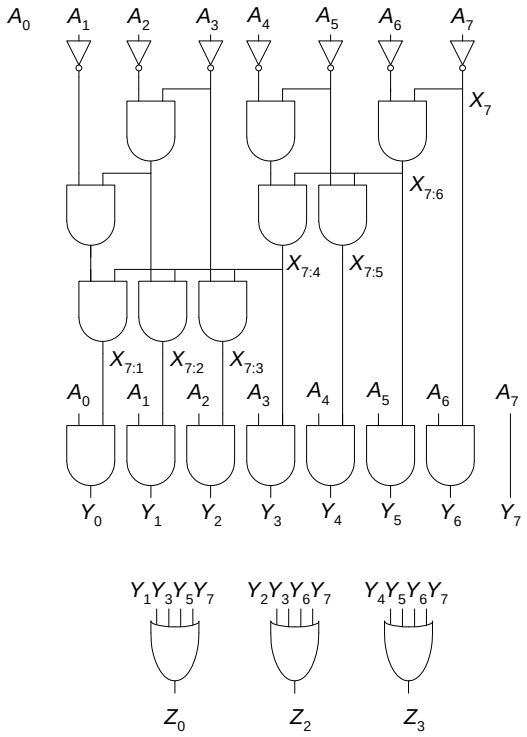


FIGURE 5.4 8-input priority encoder

SystemVerilog

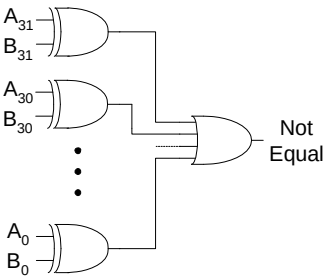
```
module priorityckt(input  logic [7:0] a,
                   output logic [2:0] z);
  logic [7:0] y;
  logic       x7, x76, x75, x74, x73, x72, x71;
  logic       x32, x54, x31;
  logic [7:0] abar;

  // 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

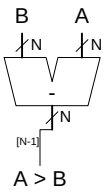
```
library IEEE; use IEEE.STD_LOGIC_1164.all;

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

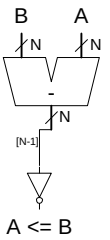
(a)



(b)



(c)



Exercise 5.9

SystemVerilog

```
module alu32(input  logic [31:0] A, B,
             input  logic [2:0]  F,
             output logic [31:0] Y);

  logic [31:0] S, Bout;

  assign Bout = F[2] ? ~B : B;
  assign S = A + Bout + F[2];

  always_comb
    case (F[1:0])
      2'b00: Y <= A & Bout;
      2'b01: Y <= A | Bout;
      2'b10: Y <= S;
      2'b11: Y <= S[31];
    endcase

endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity alu32 is
  port(A, B: in  STD_LOGIC_VECTOR(31 downto 0);
       F:    in  STD_LOGIC_VECTOR(2 downto 0);
       Y:    out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture synth of alu32 is
  signal S, Bout:     STD_LOGIC_VECTOR(31 downto 0);
begin
  Bout <= (not B) when (F(2) = '1') else B;
  S <= A + Bout + F(2);

  process(all) begin
    case F(1 downto 0) is
      when "00" => Y <= A and Bout;
      when "01" => Y <= A or Bout;
      when "10" => Y <= S;
      when "11" => Y <=
      ("00000000000000000000000000000000" & S(31));
      when others => Y <= X"00000000";
    end case;
  end process;
end;
```

Exercise 5.10

(a)
When adding:

- If both operands are positive and output is negative, overflow occurred.
- If both operands are negative and the output is positive, overflow occurred.

When subtracting:

- If the first operand is positive and the second is negative, if the output of the adder unit is negative, overflow occurred.
- If first operand is negative and second operand is positive, if the output of the adder unit is positive, overflow occurred.

In equation form:
Overflow = ADD & (A & B & ~S[31] | ~A & ~B & S[31]) |

```
                SUB & (A & ~B & ~S[31] | ~A & B & S[31]);
// note:  S is the output of the adder
```

When the ALU is performing addition, F[2] = 0. With subtraction, F[2] = 1. Thus,

```
Overflow = ~F[2] & (A & B & ~S[31] | ~A & ~B & S[31]) |
           F[2] & (~A & B & S[31] |  A & ~B & ~S[31]);
```

(b)

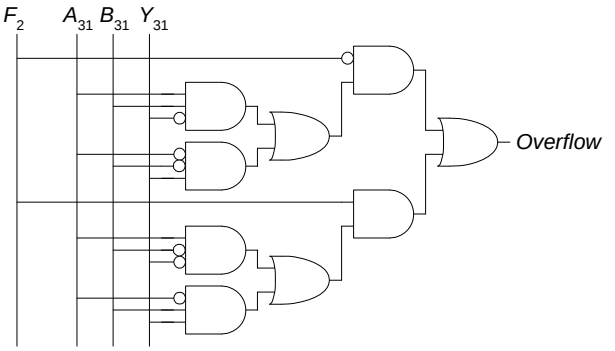


FIGURE 5.5 Overflow circuit

(c)

SystemVerilog

```
module alu32(input  logic [31:0] A, B,
             input  logic [2:0] F,
             output logic [31:0] Y,
             output logic Overflow);
  logic [31:0] S, Bout;

  assign Bout = F[2] ? ~B : B;
  assign S = A + Bout + F[2];

  always_comb
    case (F[1:0])
      2'b00: Y = A & Bout;
      2'b01: Y = A | Bout;
      2'b10: Y = S;
      2'b11: Y = S[31];
    endcase

  always_comb
    case (F[2])
      1'b0: Overflow =  A[31] & B[31] & ~S[31] |
                       ~A[31] & ~B[31] & S[31];
      1'b1: Overflow = ~A[31] & B[31] & S[31] |
                        A[31] & ~B[31] & ~S[31];
      default: Overflow = 1'b0;
    endcase

endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity alu32 is
  port(A, B: in  STD_LOGIC_VECTOR(31 downto 0);
   F:        in  STD_LOGIC_VECTOR(2 downto 0);
   Y:        out STD_LOGIC_VECTOR(31 downto 0);
   Overflow: out STD_LOGIC);
end;

architecture synth of alu32 is
  signal S, Bout:     STD_LOGIC_VECTOR(31 downto 0);
begin
  Bout <= (not B) when (F(2) = '1') else B;
  S <= A + Bout + F(2);

  -- alu function
  process(all) begin
    case F(1 downto 0) is
      when "00" => Y <= A and Bout;
      when "01" => Y <= A or Bout;
      when "10" => Y <= S;
      when "11" => Y <=
 ("00000000000000000000000000000000" & S(31));
      when others => Y <= X"00000000";
    end case;
  end process;

  -- overflow circuit
  process(all) begin
    case F(2 downto 1) is
      when "01" => Overflow <=
        (A(31) and B(31) and (not (S(31)))) or
        ((not A(31)) and (not B(31)) and S(31));
      when "11" => Overflow <=
        ((not A(31)) and B(31) and S(31)) or
        (A(31) and (not B(31)) and (not S(31)));
      when others => Overflow <= '0';
    end case;
  end process;
end;
```

Exercise 5.11

SystemVerilog

```
module alu32(input  logic [31:0] A, B,
             input  logic [2:0]  F,
             output logic [31:0] Y,
             output logic Zero, Overflow);
  logic [31:0] S, Bout;

  assign Bout = F[2] ? ~B : B;
  assign S = A + Bout + F[2];

  always_comb
    case (F[1:0])
      2'b00: Y <= A & Bout;
      2'b01: Y <= A | Bout;
      2'b10: Y <= S;
      2'b11: Y <= S[31];
    endcase

  assign Zero = (Y == 32'b0);

  always_comb
    case (F[2:1])
      2'b01: Overflow <= A[31] & B[31] & ~S[31] |
                         ~A[31] & ~B[31] & S[31];
      2'b11: Overflow <= ~A[31] & B[31] & S[31] |
                          A[31] & ~B[31] & ~S[31];
      default: Overflow <= 1'b0;
    endcase
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity alu32 is
  port(A, B:     in    STD_LOGIC_VECTOR(31 downto 0);
       F:        in    STD_LOGIC_VECTOR(2 downto 0);
       Y:    inout STD_LOGIC_VECTOR(31 downto 0);
       Overflow: out   STD_LOGIC;
       Zero:     out   STD_LOGIC);
end;

architecture synth of alu32 is
  signal S, Bout:     STD_LOGIC_VECTOR(31 downto 0);
begin
  Bout <= (not B) when (F(2) = '1') else B;
  S <= A + Bout + F(2);

  -- alu function
  process(all) begin
    case F(1 downto 0) is
      when "00" => Y <= A and Bout;
      when "01" => Y <= A or Bout;
      when "10" => Y <= S;
      when "11" => Y <=
        ("00000000000000000000000000000000" & S(31));
      when others => Y <= X"00000000";
    end case;
  end process;

  Zero <= '1' when (Y = X"00000000") else '0';

  -- overflow circuit
  process(all) begin
    case F(2 downto 1) is
      when "01" => Overflow <=
        (A(31) and B(31) and (not (S(31)))) or
        ((not A(31)) and (not B(31)) and S(31));
      when "11" => Overflow <=
        ((not A(31)) and B(31) and S(31)) or
        (A(31) and (not B(31)) and (not S(31)));
      when others => Overflow <= '0';
    end case;
  end process;
end;
```

Exercise 5.12

The following shows the contents of the file test_alu32.tv and test_alu32_vhdl.tv. Note that VHDL does not use underscores ("_") to separate the hex digits.

test_alu32.tv

```
0_A_00000000_00000000_00000000
0_2_00000000_FFFFFFFF_FFFFFFFF
0_A_00000001_FFFFFFFF_00000000
0_2_000000FF_00000001_00000100
0_E_00000000_00000000_00000000
0_6_00000000_FFFFFFFF_00000001
0_E_00000001_00000001_00000000
0_6_00000100_00000001_000000FF
0_F_00000000_00000000_00000000
0_7_00000000_00000001_00000001
0_F_00000000_FFFFFFFF_00000000
0_F_00000001_00000000_00000000
0_7_FFFFFFFF_00000000_00000001
0_0_FFFFFFFF_FFFFFFFF_FFFFFFFF
0_0_FFFFFFFF_12345678_12345678
0_0_12345678_87654321_02244220
0_8_00000000_FFFFFFFF_00000000
0_1_FFFFFFFF_FFFFFFFF_FFFFFFFF
0_1_12345678_87654321_97755779
0_1_00000000_FFFFFFFF_FFFFFFFF
0_9_00000000_00000000_00000000
1_2_FFFFFFFF_80000000_7FFFFFFF
1_2_00000001_7FFFFFFF_80000000
1_6_80000000_00000001_7FFFFFFF
1_6_7FFFFFFF_FFFFFFFF_80000000
```

test_alu32_vhdl.tv

```
0a00000000000000000000000
020000000ffffffffffffffff
0a00000001ffffffff00000000
02000000ff0000000100000100
0e00000000000000000000000
0600000000ffffffff00000001
0e00000001000000010000000
060000010000000001000000ff
0f00000000000000000000000
0700000000000000010000001
0f00000000ffffffff00000000
0f00000001000000000000000
07ffffffff0000000000000001
00fffffffffffffffffffffff
00fffffffff1234567812345678
00123456788765432102244220
0800000000ffffffff00000000
01fffffffffffffffffffffff
01123456788765432197755779
0100000000fffffffffffffff
09000000000000000000000000
12ffffffff800000007fffffff
120000000017fffffff80000000
168000000000000017fffffff
167fffffffffffffff80000000
```

(continued on next page)

Testbench

SystemVerilog

```
module test_alu32_v;

  // Inputs
  logic [31:0] A;
  logic [31:0] B;
  logic [2:0] F;

  // Outputs
  logic [31:0] Y;
  logic Zero, Overflow;

  // Internal signals
  reg clk;

  // Simulation variables
  logic  [31:0]   vectornum, errors;
  logic  [100:0]  testvectors[10000:0];
  logic  [31:0]   ExpectedY;
  logic           ExpectedZero;
  logic           ExpectedOverflow;

  // Instantiate the Unit Under Test (UUT)
  alu32 uut (A, B, F, Y, Zero, Overflow);

  // generate clock
  always
    begin
      clk = 1; #5; clk = 0; #5;
    end
```

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 test_alu32_vhd is -- no inputs or outputs
end;

architecture sim of test_alu32_vhd is
  component alu32
  port(a, b:     in    STD_LOGIC_VECTOR(31 downto 0);
       f:        in    STD_LOGIC_VECTOR(2 downto 0);
       y:        inout STD_LOGIC_VECTOR(31 downto 0);
       zero:     out   STD_LOGIC;
       overflow: out   STD_LOGIC);
  end component;

  signal a, b:       STD_LOGIC_VECTOR(31 downto 0);
  signal f:          STD_LOGIC_VECTOR(2 downto 0);
  signal y:          STD_LOGIC_VECTOR(31 downto 0);
  signal zero:       STD_LOGIC;
  signal overflow:   STD_LOGIC;
  signal clk, reset: STD_LOGIC;
  signal yexpected:  STD_LOGIC_VECTOR(31 downto 0);
  signal oexpected:  STD_LOGIC;
  signal zexpected:  STD_LOGIC;
  constant MEMSIZE: integer := 25;
  type tvarray is array(MEMSIZE downto 0) of
  STD_LOGIC_VECTOR(100 downto 0);
  shared variable testvectors: tvarray;
  shared variable vectornum, errors: integer;
begin

  -- instantiate device under test
  dut: alu32 port map(a, b, f, y, zero, overflow);

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

(continued on next page)

(continued from previous page)

SystemVerilog

```
// at start of test, load vectors
initial
  begin
    $readmemh("test_alu32.tv", testvectors);
    vectornum = 0; errors = 0;
  end
```

VHDL

```
-- at start of test, load vectors
-- and pulse reset
process is
  file tv: TEXT;
  variable i, index, count: integer;
  variable L: line;
  variable ch: character;
  variable result: integer;
begin
  -- read file of test vectors
  i := 0;
  index := 0;
  FILE_OPEN(tv, "test_alu32_vhdl.tv", READ_MODE);
  report "Opening file\n";
  while not endfile(tv) loop
    readline(tv, L);
    result := 0;
    count := 3;
    for i in 1 to 26 loop
      read(L, ch);
      if '0' <= ch and ch <= '9' then
          result := result*16 + character'pos(ch)
                    - character'pos('0');
      elsif 'a' <= ch and ch <= 'f' then
         result := result*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 5 bits
      if (i = 2) then
        testvectors(index)( 100 downto 96) :=
           CONV_STD_LOGIC_VECTOR(result, 5);
        count := count - 1;
        result := 0;         -- reset result
      -- assign the rest of testvectors in
      -- 32-bit increments
      elsif ((i = 10) or (i = 18) or (i = 26)) then
        testvectors(index)( (count*32 + 31)
                            downto (count*32)) :=
           CONV_STD_LOGIC_VECTOR(result, 32);
        count := count - 1;
        result := 0;         -- reset result
      end if;
    end loop;

    index := index + 1;
  end loop;

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

end process;
```

(continued on next page)

(continued from previous page)

SystemVerilog

```
 // apply test vectors on rising edge of clk

  always @(posedge clk)
  begin
    #1; {ExpectedOverflow, ExpectedZero, F, A, B,
         ExpectedY} = testvectors[vectornum];
    end

  // check results on falling edge of clk
  always @(negedge clk)
  begin
     if ({Y, Zero, Overflow} !==
        {ExpectedY, ExpectedZero, ExpectedOverflow})
       begin
         $display("Error: inputs: F = %h, A = %h,
         B = %h", F, A, B);
         $display("  Y = %h, Zero = %b
           Overflow = %b\n (Expected Y = %h,
           Expected Zero = %b, Expected Overflow
           = %b)", Y, Zero, Overflow,
           ExpectedY, ExpectedZero,
           ExpectedOverflow);
         errors = errors + 1;
       end
     vectornum = vectornum + 1;
     if (testvectors[vectornum] === 101'hx)
       begin
         $display("%d tests completed with %d
           errors", vectornum, errors);
         $finish;
       end
  end

endmodule
```

VHDL

```
  -- apply test vectors on rising edge of clk
  process (clk) begin
    if (clk'event and clk = '1') then
      oexpected <= testvectors(vectornum)(100)
        after 1 ns;
     zexpected <= testvectors(vectornum)(99)
        after 1 ns;
     f       <= testvectors(vectornum)(98 downto 96)
        after 1 ns;
     a       <= testvectors(vectornum)(95 downto 64)
        after 1 ns;
     b       <= testvectors(vectornum)(63 downto 32)
        after 1 ns;
    yexpected <= 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
      assert y = yexpected
        report "Error: vectornum = " &
          integer'image(vectornum) &
          "y = " & integer'image(CONV_INTEGER(y)) &
         ", a = " & integer'image(CONV_INTEGER(a)) &
         ", b = " & integer'image(CONV_INTEGER(b)) &
          ", f = " & integer'image(CONV_INTEGER(f));
      assert overflow = oexpected
        report "Error: overflow = " &
          STD_LOGIC'image(overflow);
      assert zero = zexpected
        report "Error: zero = " &
          STD_LOGIC'image(zero);
      if ( (y /= yexpected) or
           (overflow /= oexpected) or
           (zero /= zexpected) ) then
        errors := errors + 1;
      end if;
      vectornum := vectornum + 1;
      if (is_x(testvectors(vectornum))) then
        if (errors = 0) then
          report "Just kidding -- " &
                 integer'image(vectornum) &
                 " tests completed successfully."
                 severity failure;
        else
          report integer'image(vectornum) &
                 " tests completed, errors = " &
                 integer'image(errors)
                 severity failure;
        end if;
      end if;
    end if;
  end process;
end;
```

Exercise 5.13

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.

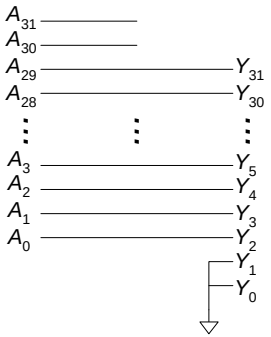


FIGURE 5.6 2-bit left shifter, 32-bit input and output

2-bit Left Shifter

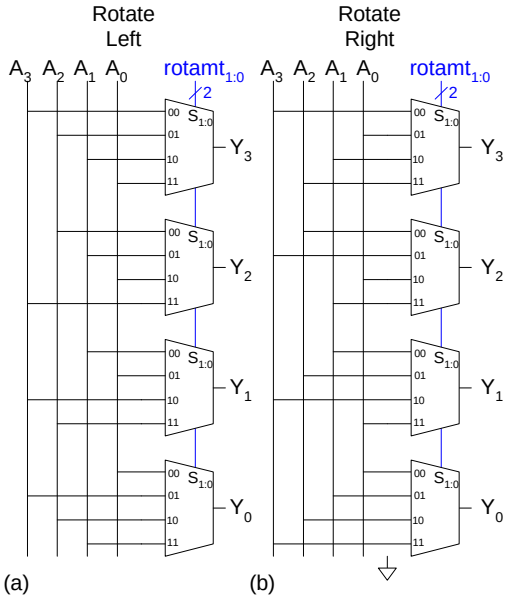
SystemVerilog

```
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 5.14


Rotate Left
Rotate Right
A3 A2 A1 A0
rotamt1:0
2
00
01
10
11
S1:0
Y3
Y2
Y1
Y0
(a)
(b)

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 5.15

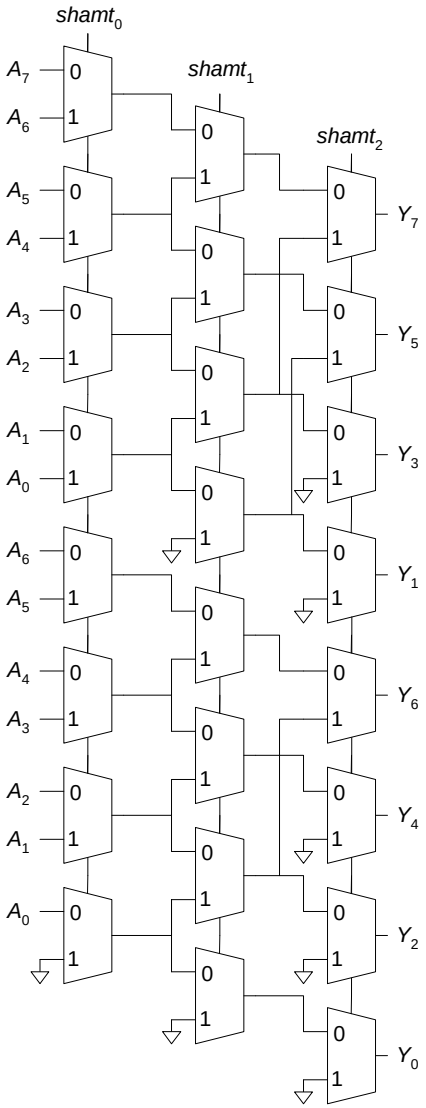


FIGURE 5.7 8-bit left shifter using 24 2:1 multiplexers

Exercise 5.16

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 5.17

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

Exercise 5.18

$t_{pd\_\mathrm{MULT4}} = t_{\mathrm{AND}} + 3t_{FA}$

An $N \times N$ multiplier has N-bit operands, N partial products, and N-1 stages of 1-bit adders. So the propagation is:

$t_{pd\_\mathrm{MULT}N} = t_{\mathrm{AND}} + (N\text{-}1)t_{FA}$

Exercise 5.19

$t_{pd\_\mathrm{DIV4}} = 4\,(4t_{FA} + t_{\mathrm{MUX}}) = 16t_{FA} + 4t_{\mathrm{MUX}}$

$t_{pd\_\mathrm{DIV}N} = N^2 t_{FA} + Nt_{\mathrm{MUX}}$

Exercise 5.20

Recall that a two's complement number has the same weights for the least significant N-1 bits, regardless of the sign. The sign bit has a weight of -2^{N-1}. Thus, the product of two N-bit complement numbers, y and x is:

$$P = \left(-y_{N-1}2^{N-1} + \sum_{j=0}^{N-2} y_j 2^j\right)\left(-x_{N-1}2^{N-1} + \sum_{i=0}^{N-2} x_i 2^i\right)$$

Thus,

$$\sum_{i=0}^{N-2}\sum_{j=0}^{N-2} x_i y_j 2^{i+j} + x_{N-1}y_{N-1}2^{2N-2} - \sum_{i=0}^{N-2} x_i y_{N-1}2^{i+N-1} - \sum_{j=0}^{N-2} x_{N-1}y_j 2^{j+N-1}$$

The two negative partial products are formed by taking the two's complement (inverting the bits and adding 1). Figure 5.8 shows a 4 x 4 multiplier. Figure 5.8 (b) shows the partial products using the above equation. Figure 5.8 (c) shows a simplified version, pushing through the 1's. This is known as a *modified Baugh-Wooley multiplier*. It can be built using a hierarchy of adders.

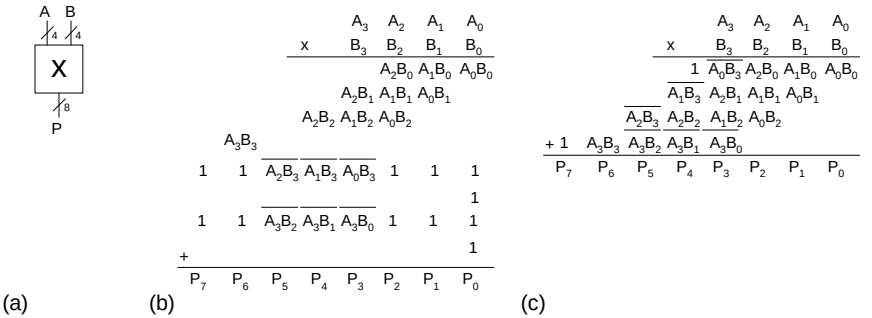


FIGURE 5.8 Multiplier: (a) symbol, (b) function, (c) simplified function

Exercise 5.21

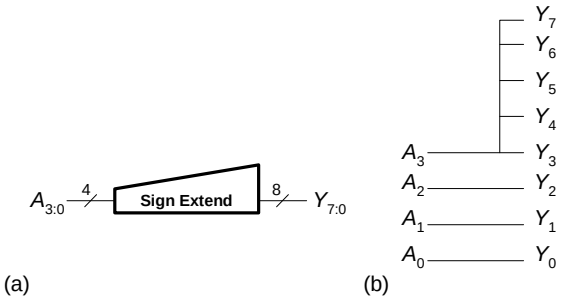


FIGURE 5.9 Sign extension unit (a) symbol, (b) underlying hardware

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 5.22

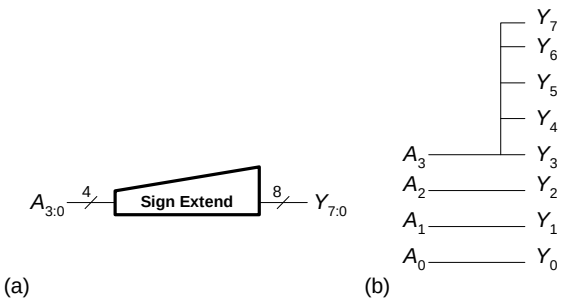


FIGURE 5.10 Zero extension unit (a) symbol, (b) underlying hardware

SystemVerilog

```
module zeroext4_8(input  logic [3:0] a,
                  output logic [7:0] y);

  assign y = {4'b0, a};

endmodule
```

VHDL

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

entity zeroext4_8 is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of zeroext4_8 is
begin
  y <= "0000" & a(3 downto 0);
end;
```

Exercise 5.23

```
        100.110
1100 | 111001.000
     -1100
      001001 0
     -   110 0
          11 00
     -    11 00
               0
```

Exercise 5.24

(a) $\left[0, \left(2^{12} - 1 + \frac{2^{12} - 1}{2^{12}}\right)\right]$

(b) $\left[-\left(2^{11} - 1 + \frac{2^{12} - 1}{2^{12}}\right), \left(2^{11} - 1 + \frac{2^{12} - 1}{2^{12}}\right)\right]$

(c) $\left[-\left(2^{11} + \frac{2^{12} - 1}{2^{12}}\right), \left(2^{11} - 1 + \frac{2^{12} - 1}{2^{12}}\right)\right]$

Exercise 5.25

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

Exercise 5.26

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

Exercise 5.27

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

Exercise 5.28

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

Exercise 5.29

(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 = **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 = **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 = **0xC1894000**

Exercise 5.30

(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 5.31

(a) 5.5
(b) $-0000.0001_2 = -0.0625$
(c) -8

Exercise 5.32

(a) 29.65625
(b) -25.1875
(c) -23.875

Exercise 5.33

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 5.34

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

Exercise 5.35

(a)
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 $\times 2^{101}$

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)
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 $\times 2^2$

1.101 1100 0000 0000 0000 01 $\times 2^2$
- 1.101 0010 0000 0000 0000 01 $\times 2^2$
= 0.000 1010 $\times 2^2$
= 1.010 $\times 2^{-2}$

= 0 0111 1101 010 0000 0000 0000 0000 0000
= 0x3EA00000

(c)
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 $\times 2^{64}$

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

And, $(1.011\ 1110\ 01 \times 2^{64} + 1.111\ 1 \times 2^{0}) - 1.101\ 1110\ 01 \times 2^{64} =$
$- 0.01 \times 2^{64} = -1.0 \times 2^{64}$
$= 1\ 1011\ 1101\ 000\ 0000\ 0000\ 0000\ 0000\ 0000$
= 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 5.36

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 5.37

(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 5.38

(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

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

(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

(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 5.39

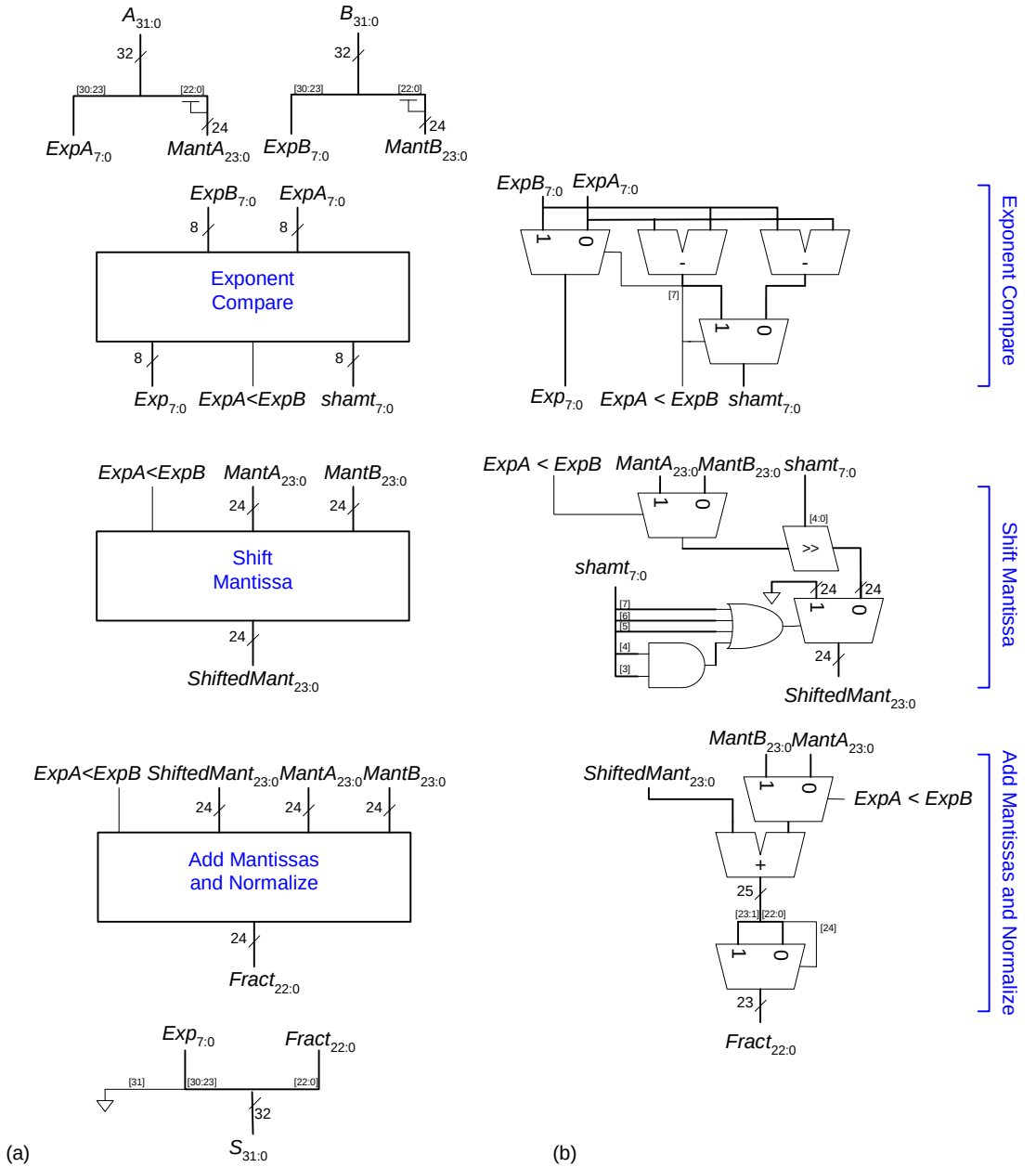


FIGURE 5.11 Floating-point adder hardware: (a) block diagram, (b) underlying hardware

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   expcomp1(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;

  expcomp1:  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
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,
                                   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_in-
teger(unsigned(shamt))) when alessb = '1'
           else SHIFT_RIGHT( unsigned(mantb), to_in-
teger(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 5.40

(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

(b)

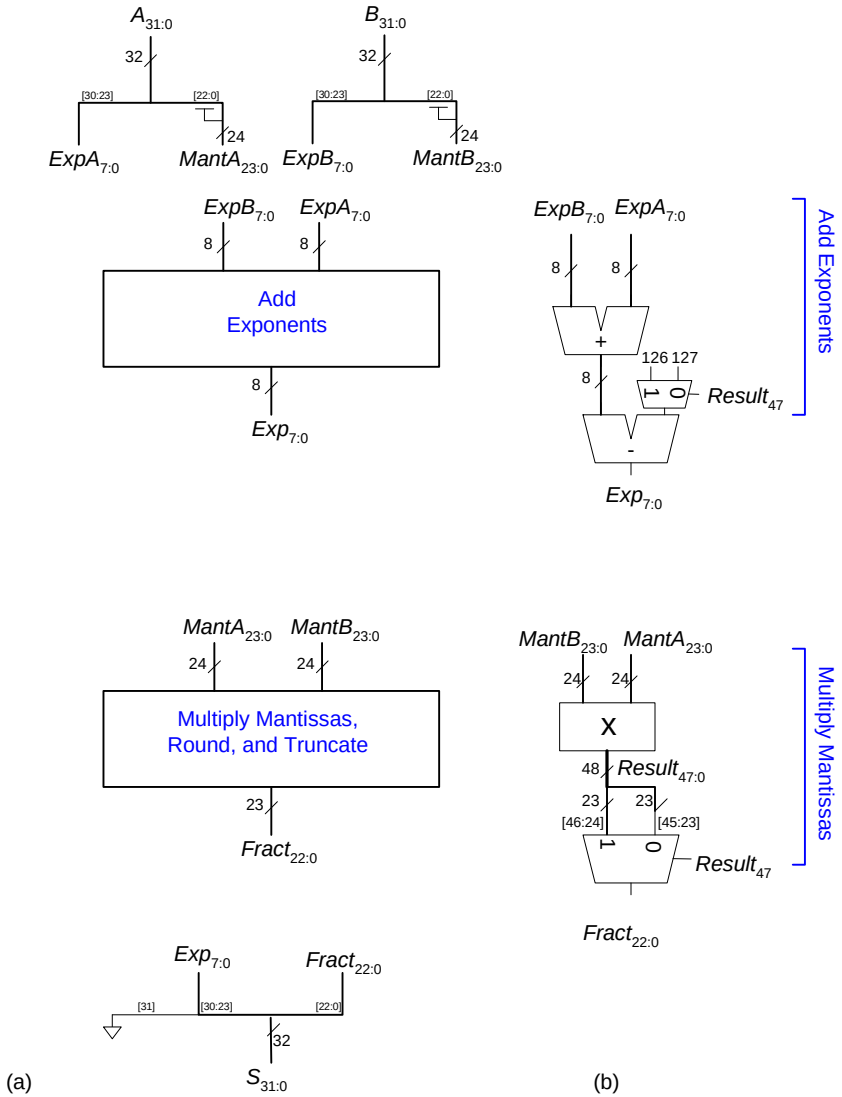


FIGURE 5.12 Floating-point multiplier block diagram

(c)

SystemVerilog

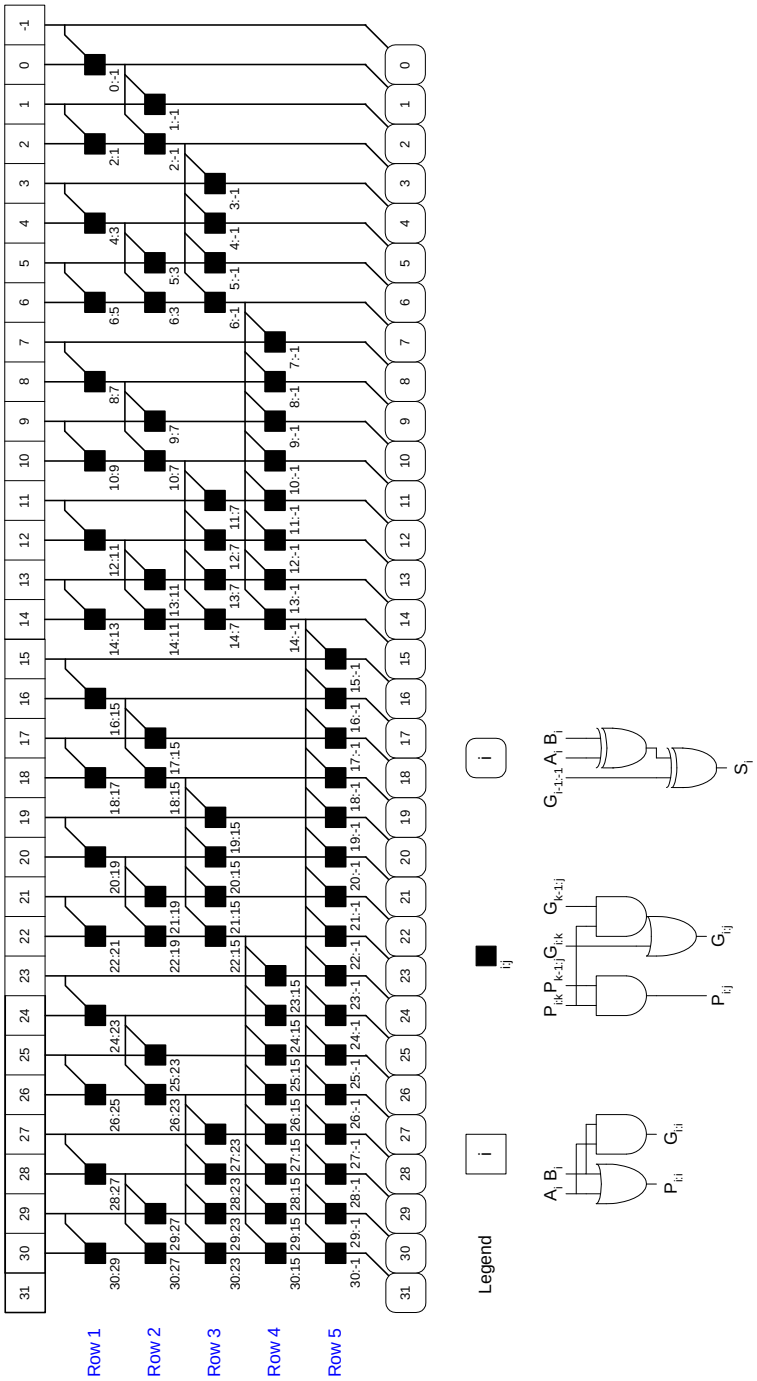
```
module fpmult(input  logic [31:0] a, b,
              output logic [31:0] m);

  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 5.41

(a) Figure on next page


Row 1
Row 2
Row 3
Row 4
Row 5
Legend
i
ij
A_i B_i
P_{ij} G_{ij}
P_{ik} $P_{k-1:j}$ G_{ik} $G_{k-1:j}$
P_{ij} G_{ij}
$G_{i-1:-1}$ A_i B_i
S_i

5.41 (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 sumblock 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: sumblock
        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:

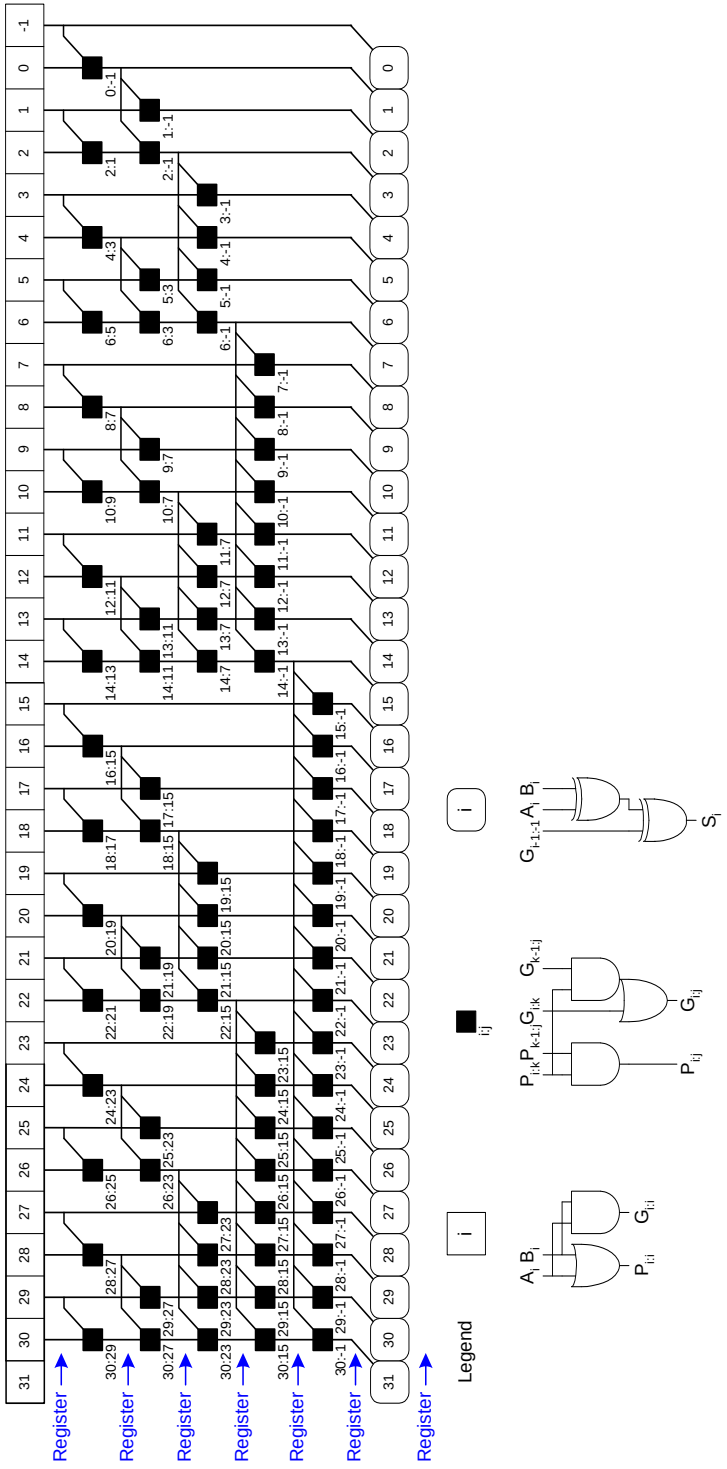
$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_prefix}) + t_{XOR}$$

We find the delays for each block:
t_{pg} = 100 ps
t_{pg_prefix} = 200 ps
t_{XOR} = 100 ps

Thus,
t_{PA} = [100 + 5(200) + 100] ps = 1200 ps = **1.2 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, $t_{pq} + t_{\text{setup}} = 80$ps. Thus each cycle is 280 ps and the design can run at 3.57 GHz.


31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 -1
Register
30:29 28:27 26:25 24:23 22:21 20:19 18:17 16:15 14:13 12:11 10:9 8:7 6:5 4:3 2:1 0:-1
30:27 29:27 28:23 27:23 26:23 25:23 24:23 23:15 22:19 21:19 20:19 19:15 18:15 17:15 16:15 15:-1 14:11 13:11 12:11 11:7 10:7 9:7 8:7 7:-1 6:3 5:3 4:3 3:-1 2:-1 1:-1
30:23 29:23 28:23 27:23 26:23 25:23 24:23 23:15 22:15 21:15 20:15 19:15 18:15 17:15 16:15 15:-1 14:7 13:7 12:7 11:7 10:7 9:7 8:7 7:-1 6:-1 5:-1 4:-1 3:-1
30:15 29:15 28:15 27:15 26:15 25:15 24:15 23:15
30:-1 29:-1 28:-1 27:-1 26:-1 25:-1 24:-1 23:-1 22:-1 21:-1 20:-1 19:-1 18:-1 17:-1 16:-1 15:-1 14:-1 13:-1 12:-1 11:-1 10:-1 9:-1 8:-1 7:-1
Legend
i
i:j
A_i B_i $P_{i:i}$ $G_{i:i}$
$P_{i:k}$ $P_{k-1:j}$ $G_{i:k}$ $G_{k-1:j}$ $P_{i:j}$ $G_{i:j}$
$G_{i-1:-1}$ A_i B_i S_i

5.41 (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
```

5.41 (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);
         p_1_0, g_1_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, p2, g2);
  row3_reg: row3 port map(clk, p2, g2, p3, g3);
  row4_reg: row4 port map(clk, p3, g3, p4, g4);
  row5_reg: row5 port map(clk, p4, g4, 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'; flop_1_g0: flop1 port map (clk, cin, g_1_0);
  flop_1_p1: flop1 port map (clk, p_1_0, p_1_1);
  flop_1_g1: flop1 port map (clk, g_1_0, g_1_1);
  flop_1_p2: flop1 port map (clk, p_1_1, p_1_2);
  flop_1_g2: flop1 port map (clk, g_1_1, g_1_2);
  flop_1_p3: flop1 port map (clk, p_1_2, p_1_3); flop_1_g3:
  flop1 port map (clk, g_1_2, g_1_3);
  flop_1_p4: flop1 port map (clk, p_1_3, p_1_4);
  flop_1_g4: flop1 port map (clk, g_1_3, g_1_4);
  flop_1_p5: flop1 port map (clk, p_1_4, p_1_5);
  flop_1_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;

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) <= pg1_out(5); g1(9) <= pg1_out(4); g1(7) <= pg1_out(3);
  g1(5) <= pg1_out(2); g1(3) <= pg1_out(1); g1(1) <= pg1_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 pg1_in, pg2_out: STD_LOGIC_VECTOR(29 downto 0);

begin
  pg1_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, pg1_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(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 5.42

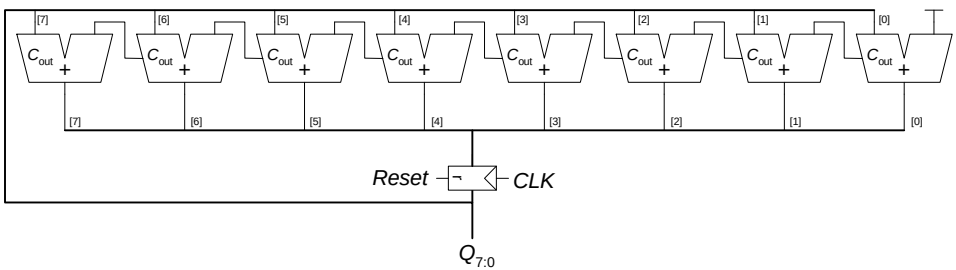


FIGURE 5.13 Incrementer built using half adders

Exercise 5.43

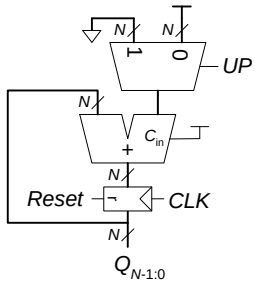


FIGURE 5.14 Up/Down counter

Exercise 5.44

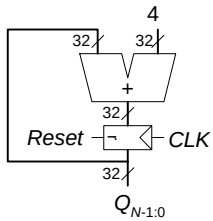


FIGURE 5.15 32-bit counter that increments by 4 on each clock edge

Exercise 5.45

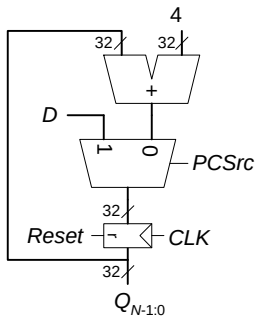


FIGURE 5.16 32-bit counter that increments by 4 or loads a new value, *D*

Exercise 5.46

(a)
0000
1000
1100
1110
1111
0111
0011
0001
(repeat)

(b)
2*N*. 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.

5.46 (c)

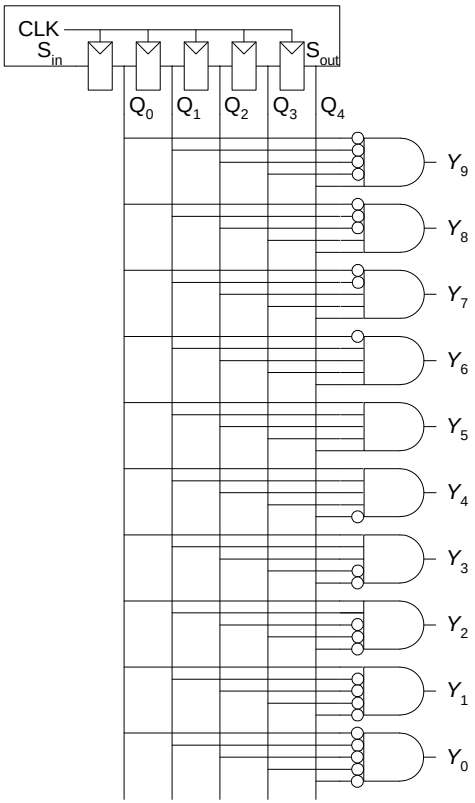


FIGURE 5.17 10-bit decimal counter using a 5-bit Johnson counter

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

Exercise 5.47

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 5.48

(a)

value $a_{1:0}$	encoding $y_{4:0}$
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.

5.48 (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 5.49

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.

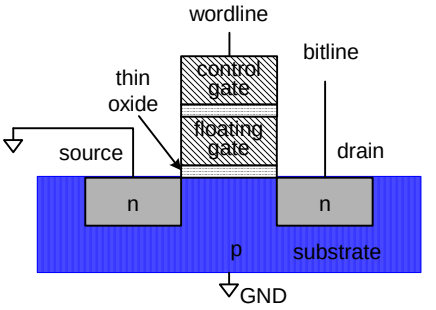


FIGURE 5.18 Flash EEPROM

Exercise 5.50

(a)

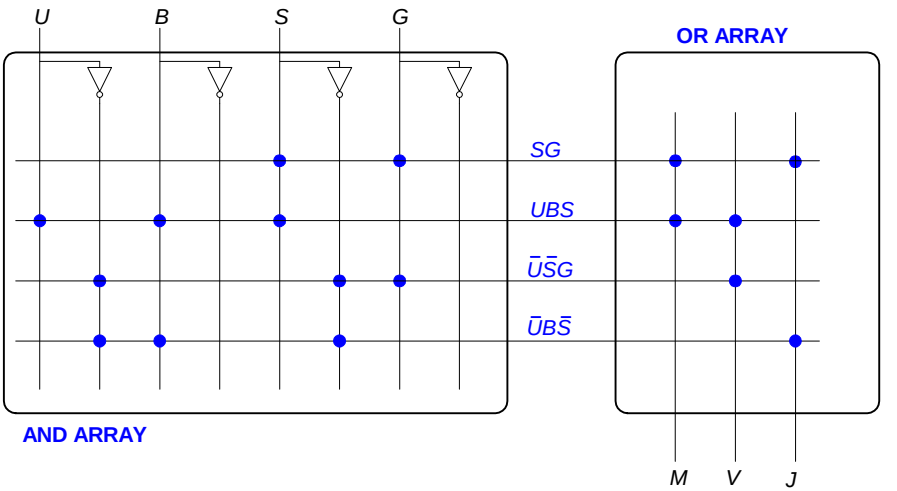


FIGURE 5.19 4 x 4 x 3 PLA implementing Exercise 5.44

5.50 (b)

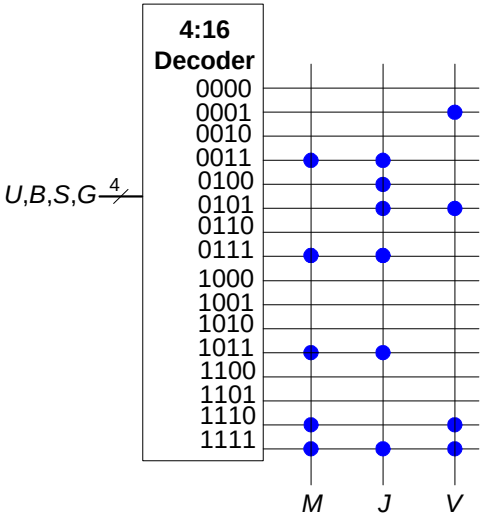


FIGURE 5.20 16 x 3 ROM implementation of Exercise 5.44

5.50 (c)

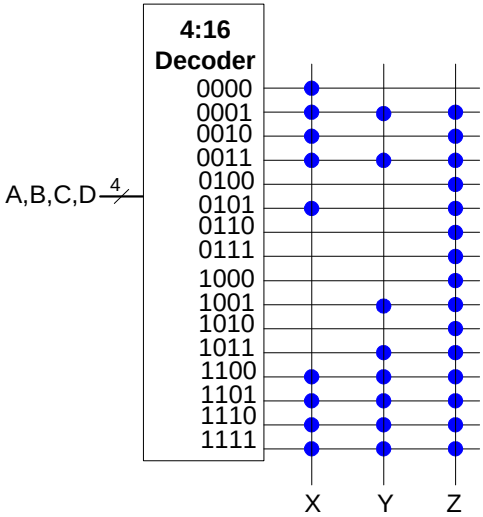
SystemVerilog

```
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 5.51



Exercise 5.52

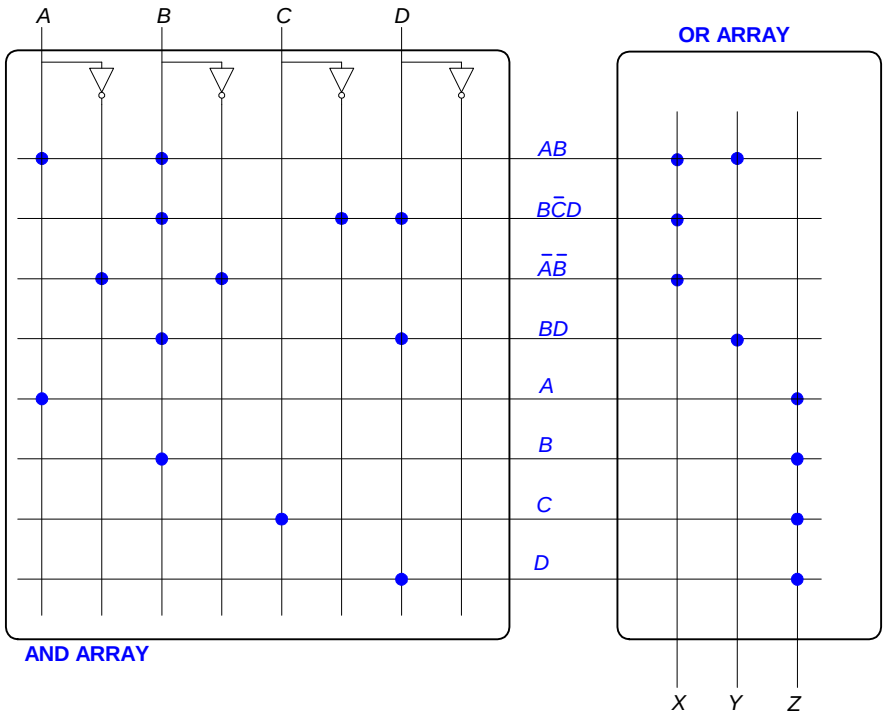


FIGURE 5.21 4 x 8 x 3 PLA for Exercise 5.52

Exercise 5.53

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

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

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

Thus, this would require a 2^{16} x 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 5.54

(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.

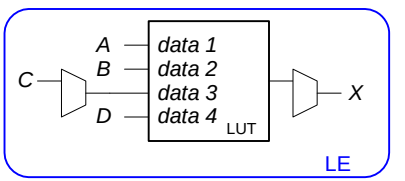
(c) Yes. Both circuits compute any function of 1 input, N outputs, and 2^K states.

(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 5.55

(a) 1 LE

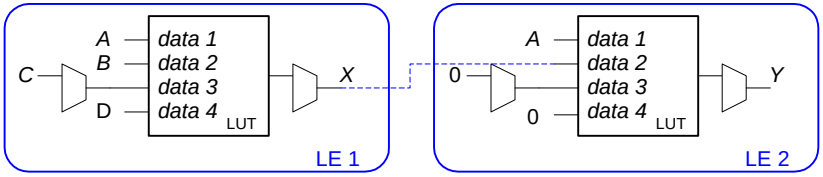
(A) data 1	(B) data 2	(C) data 3	(D) data 4	(Y) LUT output
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1



(b) 2 LEs

(B) *data 1*	(C) *data 2*	(D) *data 3*	(E) *data 4*	(X) 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	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	0
1	1	1	1	0

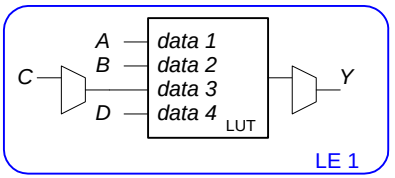
(A) *data 1*	(X) *data 2*	*data 3*	*data 4*	(Y) LUT output
0	0	X	X	0
0	1	X	X	1
1	0	X	X	1
1	1	X	X	1

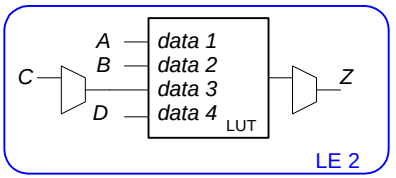


(c) 2 LEs

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

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

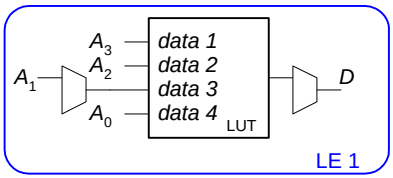


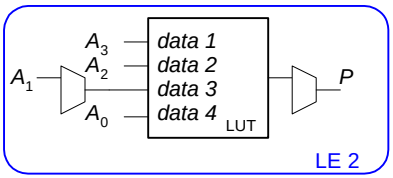


(d) 2 LEs

(A_3) *data 1*	(A_2) *data 2*	(A_1) *data 3*	(A_0) *data 4*	(D) LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

(A_3) *data 1*	(A_2) *data 2*	(A_1) *data 3*	(A_0) *data 4*	(P) LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

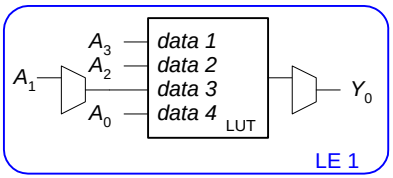


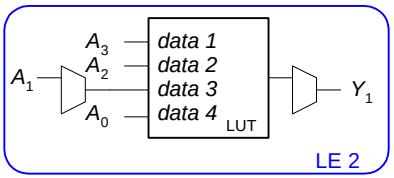


(e) 2 LEs

(A_3) *data 1*	(A_2) *data 2*	(A_1) *data 3*	(A_0) *data 4*	(Y_0) LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
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

(A_3) *data 1*	(A_2) *data 2*	(A_1) *data 3*	(A_0) *data 4*	(Y_1) LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
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	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1





Exercise 5.56

(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_3) 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

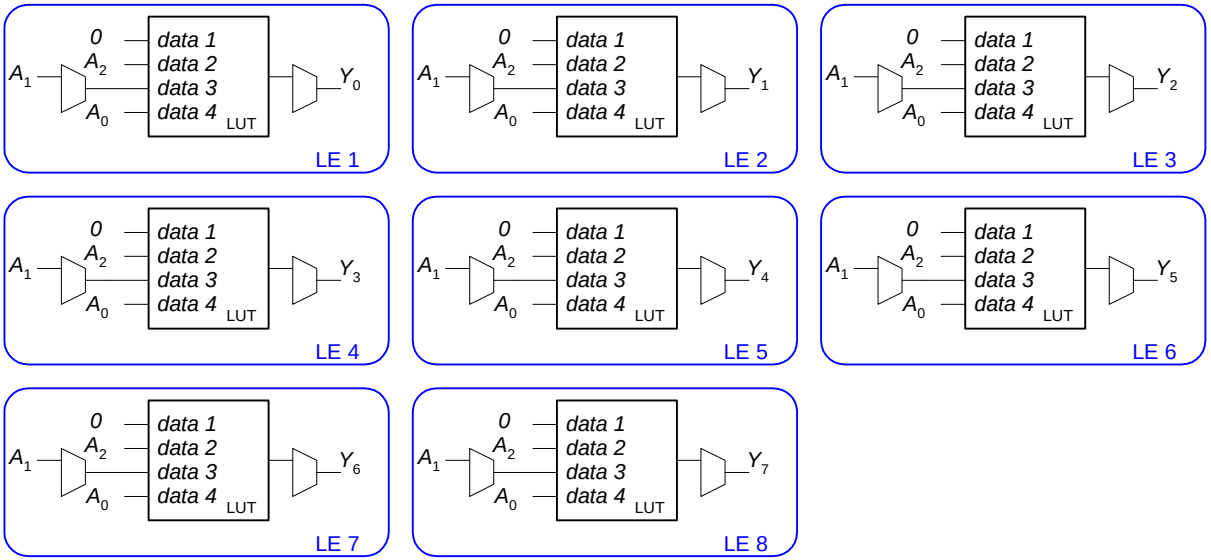
data 1	(A_2) data 2	(A_1) data 3	(A_0) data 4	(Y_5) LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	1
X	1	1	0	0
X	1	1	1	0

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

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

LE 6

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 5

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 4

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 3

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 2

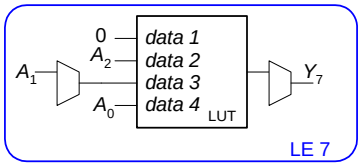
data 1	(A_2) data 2	(A_1) data 3	(A_0) data 4	(Y_2) LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	1
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

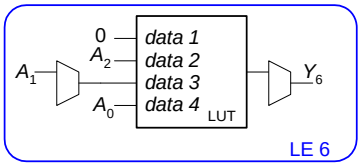
LE 1

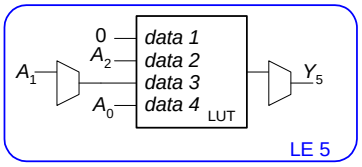
data 1	(A_2) data 2	(A_1) data 3	(A_0) data 4	(Y_1) LUT output
X	0	0	0	0
X	0	0	1	1
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

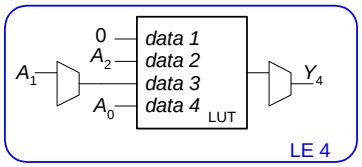
LE 0

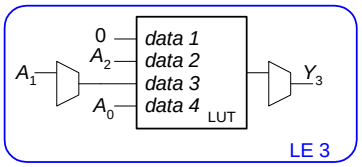
data 1	(A_2) data 2	(A_1) data 3	(A_0) data 4	(Y_0) LUT output
X	0	0	0	1
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

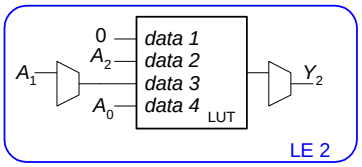


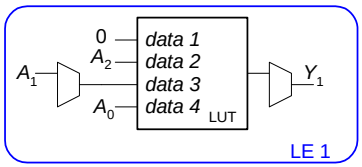


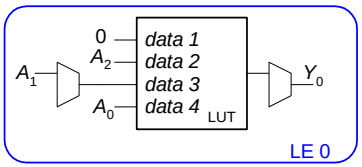












(c) 6 LEs (see next page for figure)

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

(A_0) data 1	(B_0) data 2	(A_1) data 3	(B_1) 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

(A_0) data 1	(B_0) data 2	(A_1) data 3	(B_1) data 4	(C_1) LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

LE 4

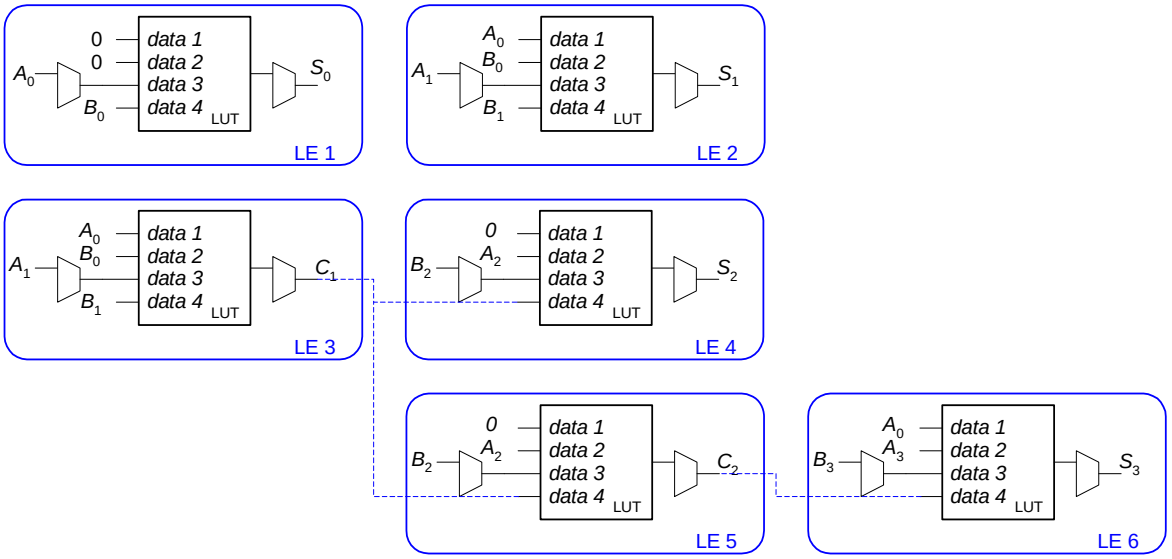
data 1	(A_2) data 2	(B_2) data 3	(C_1) data 4	(S_2) LUT output
X	0	0	0	0
X	0	0	1	1
X	0	1	0	1
X	0	1	1	0
X	1	0	0	1
X	1	0	1	0
X	1	1	0	0
X	1	1	1	1

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

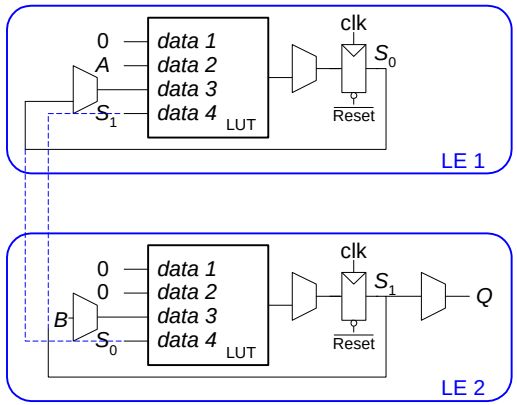
data 1	(A_3) data 2	(B_3) data 3	(C_2) data 4	(S_3) LUT output
X	0	0	0	0
X	0	0	1	1
X	0	1	0	1
X	0	1	1	0
X	1	0	0	1
X	1	0	1	0
X	1	1	0	0
X	1	1	1	1



(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')
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

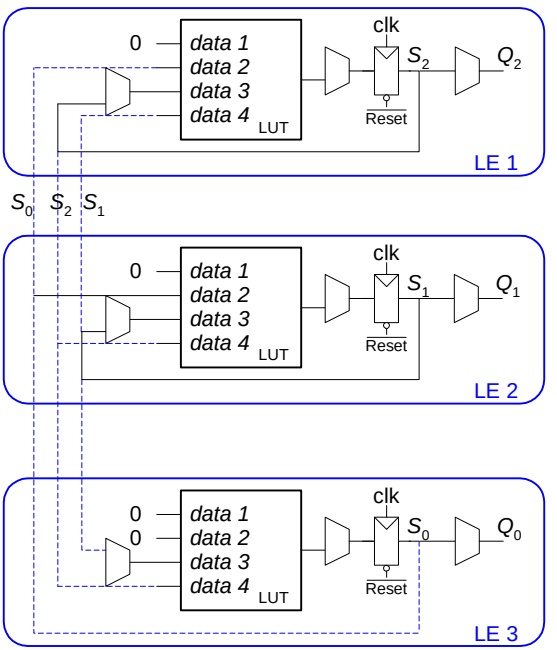


(e) 3 LEs

	(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 5.57

(a) 5 LEs (2 for next state logic and state registers, 3 for output logic)
(b)
$t_{pd} = t_{pd\_LE} + t_{wire}$
$= (381+246)$ ps
$= 627$ ps
$T_c \geq t_{pcq} + t_{pd} + t_{setup}$
$\geq [199 + 627 + 76]$ ps
$= 902$ ps
$f = 1 / 902$ ps = **1.1 GHz**
(c)
First, we check that there is no hold time violation with this amount of clock skew.
$t_{cd\_LE} = t_{pd\_LE} = 381$ ps
$t_{cd} = t_{cd\_LE} + t_{wire} = 627$ ps

$t_{skew} < (t_{ccq} + t_{cd}) - t_{hold}$
$< [(199 + 627) - 0]$ ps
< 826 ps
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.
$T_c \geq t_{pcq} + t_{pd} + t_{setup} + t_{skew}$
$\geq [0.902 + 3]$ ns
$= 3.902$ ns
$f = 1 / 3.902$ ns = **256 MHz**

Exercise 5.58

(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 5.59

First, we find the cycle time:
$T_c = 1/f = 1/100$ MHz = 10 ns
$T_c \geq t_{pcq} + Nt_{LE+wire} + t_{setup}$
$10 \text{ ns} \geq [0.199 + N(0.627) + 0.076]$ ns

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:

$T_c \geq [0.199 + 0.627 + 0.076]$ ns

≥ 0.902 ns

$f = 1 / 0.902$ ns = **1.1 GHz**

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

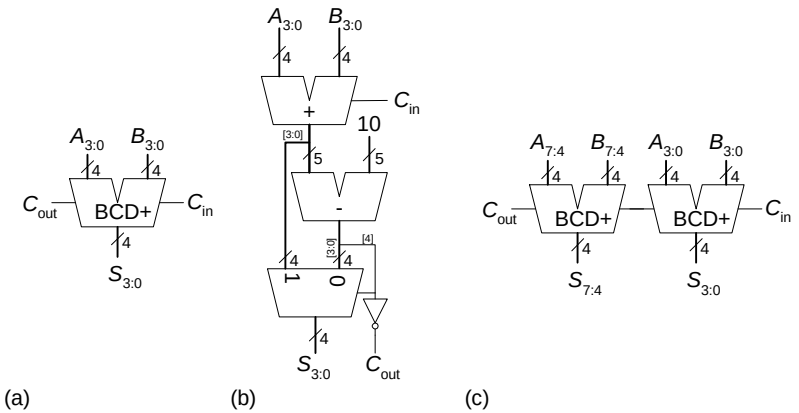


FIGURE 5.22 BCD adder: (a) 4-bit block, (b) underlying hardware, (c) 8-bit BCD adder

(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;
```