

EXERCISE 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 1.3

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 re-specify 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 1.5

- (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 1.7

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

Exercise 1.9

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

Exercise 1.11

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

Exercise 1.13

- (a) 10; (b) 54; (c) 240; (d) 2215

Exercise 1.15

- (a) A; (b) 36; (c) F0; (d) 8A7

Exercise 1.17

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

Exercise 1.19

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

Exercise 1.21

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

Exercise 1.23

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

Exercise 1.25

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

Exercise 1.27

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

Exercise 1.29

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

Exercise 1.31

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

Exercise 1.33

- (a) 00000101; (b) 11111010

Exercise 1.35

- (a) 00000101; (b) 00001010

Exercise 1.37

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

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 1.41

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

Exercise 1.43

4, 8

Exercise 1.45

5,760,000

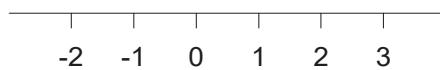
EExercise 1.47

46.566 gigabytes

Exercise 1.49

128 kbits

Exercise 1.51



Unsigned 00 01 10 11

10 11 00 01

Two's Complement

11 00
10 01

Sign/Magnitude

Exercise 1.53

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

Exercise 1.55

(a) 11011101; (b) 110001000

Exercise 1.57

$$(a) 000111 + 001101 = 010100$$

(b) $010001 + 011001 = 101010$, overflow

$$(c) \quad 100110 + 001000 = 101110$$

- (d) $011111 + 110010 = 010001$
(e) $101101 + 101010 = 010111$, overflow
(f) $111110 + 100011 = 100001$

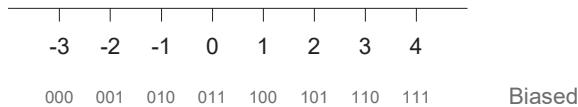
Exercise 1.59

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

Exercise 1.61

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

Exercise 1.63



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 1.67

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

Exercise 1.69

```
#include <stdio.h>

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

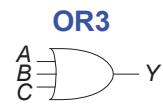
    printf("Enter binary number: ");
    scanf("%s", bin);
```

```

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 1.71



$$Y = A+B+C$$

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

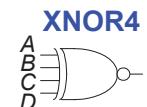
(a)



$$Y = A \oplus B \oplus C$$

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)



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

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 1.73

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 1.75

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 1.77

$$2^{2^N}$$

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.

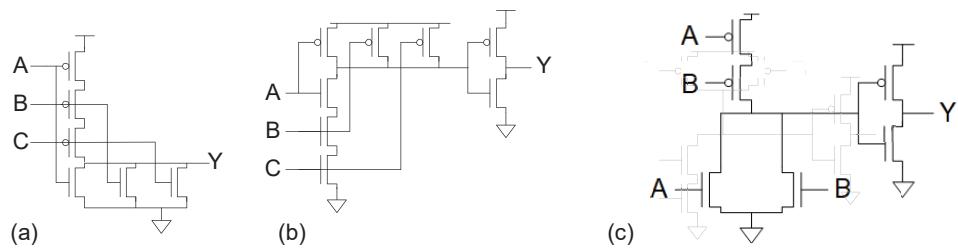
Exercise 1.81

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 1.83

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

Exercise 1.85

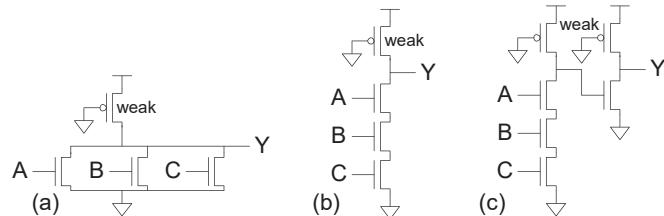


Exercise 1.87

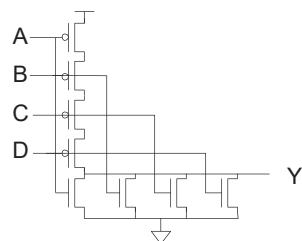
XOR

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

Exercise 1.89



Question 1.1



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) $Y = \overline{A}\overline{B} + A\overline{B} + AB$

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

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

(d)

$$Y = \overline{A}\overline{B}\overline{C}\overline{D} + \overline{A}\overline{B}\overline{C}D + \overline{A}\overline{B}C\overline{D} + \overline{A}\overline{B}CD + A\overline{B}\overline{C}\overline{D} + A\overline{B}\overline{C}D + A\overline{B}C\overline{D} + ABC\overline{D}$$

(e)

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

Exercise 2.3

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

(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)

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

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

(e)

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

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

Exercise 2.5

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

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

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

(d) $Y = \overline{AB} + \overline{BD} + ACD$

(e)

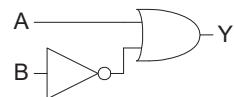
$$Y = \overline{\overline{A}\overline{B}\overline{C}\overline{D}} + \overline{\overline{A}\overline{B}CD} + \overline{A}\overline{B}\overline{C}D + \overline{ABC}\overline{D} + A\overline{B}\overline{C}\overline{D} + A\overline{B}\overline{C}D + A\overline{B}\overline{C}D + ABCD$$

This can also be expressed as:

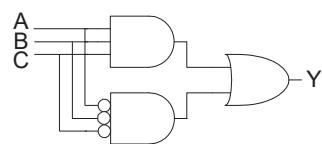
$$Y = \overline{(A \oplus B)}(C \oplus D) + (A \oplus B)(C \oplus D)$$

Exercise 2.7

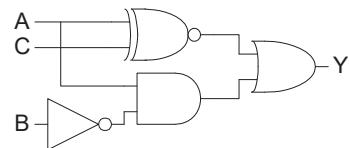
(a)



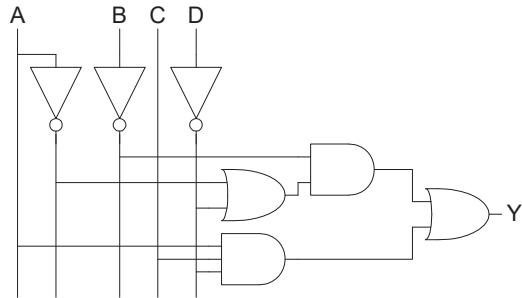
(b)



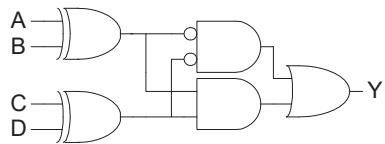
(c)



(d)

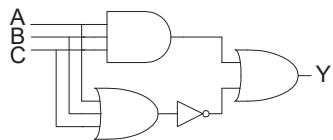


(e)

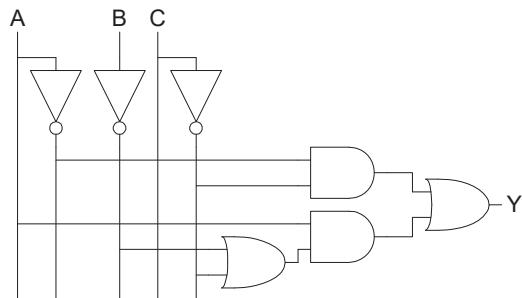


Exercise 2.9

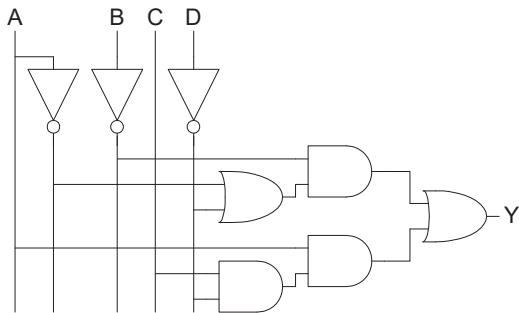
- (a) Same as 2.7(a)
(b)



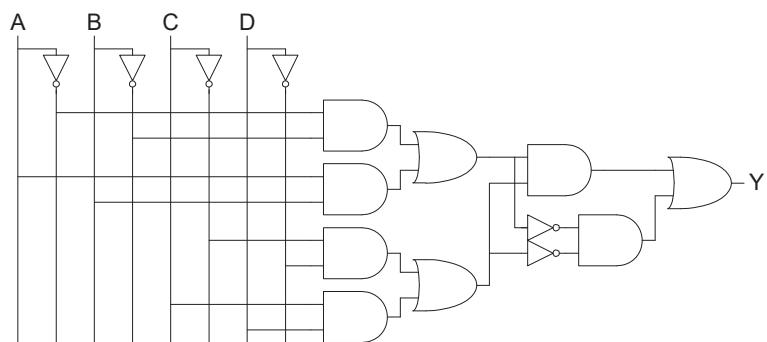
(c)



(d)

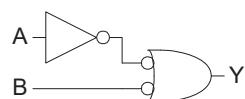


(e)

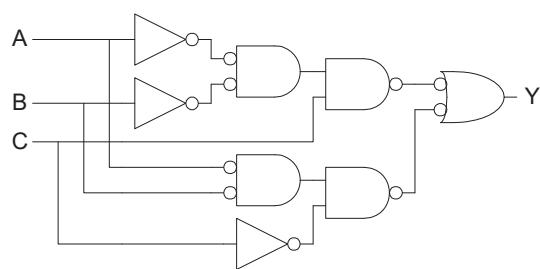


Exercise 2.11

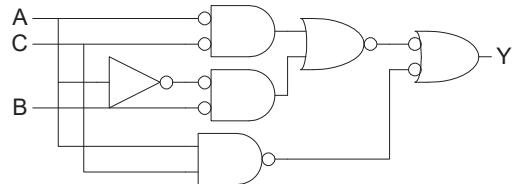
(a)



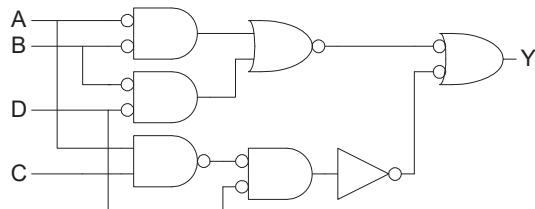
(b)



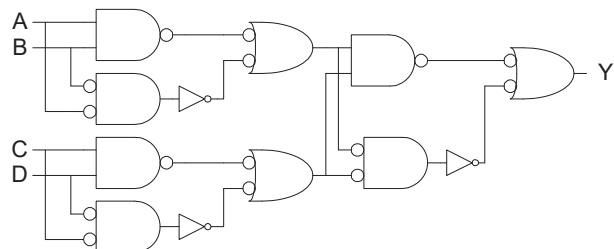
(c)



(d)



(e)

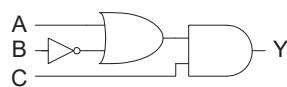


Exercise 2.13

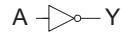
- (a) $Y = AC + \overline{B}C$
- (b) $Y = \overline{A}$
- (c) $Y = \overline{A} + \overline{B}\overline{C} + \overline{B}\overline{D} + BD$

Exercise 2.15

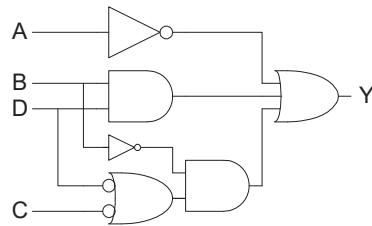
(a)



(b)



(c)

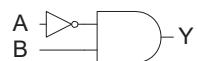


Exercise 2.17

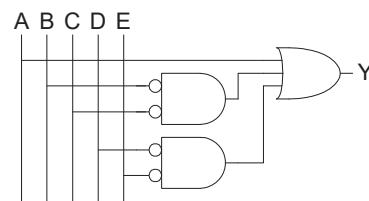
(a) $Y = B + \overline{A}\overline{C}$



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



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

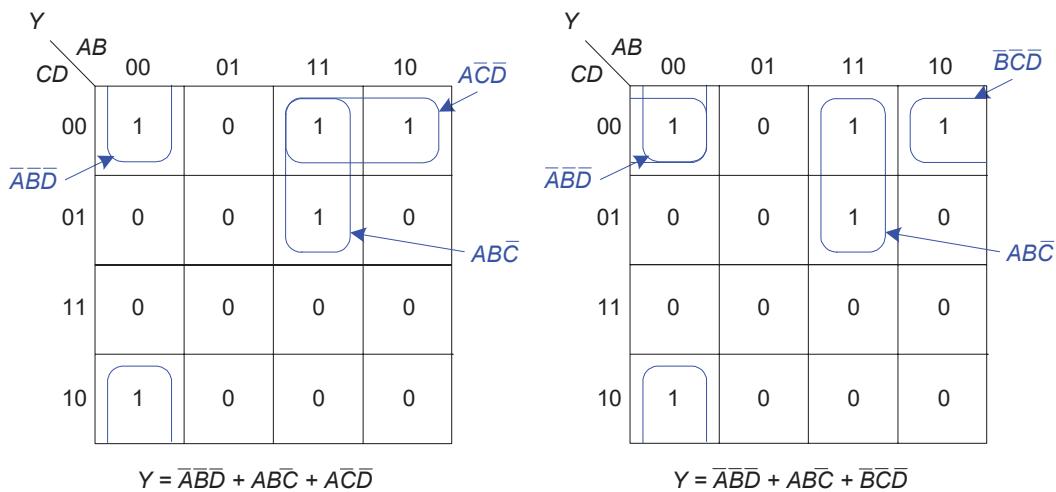


Exercise 2.19

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

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 $\bar{B}CD$ are both prime implicants, the minimal sum-of-products expression does not have both of them.



Exercise 2.23

B_2	B_1	B_0	$\bar{B}_2 \bullet B_1 \bullet B_0$	$\bar{B}_2 + \bar{B}_1 + \bar{B}_0$
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

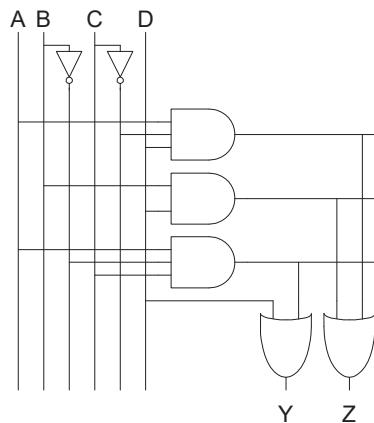
Exercise 2.25

	AB	00	01	11	10
CD	00	0	0	0	0
D	01	1	1	1	1
	11	1	1	1	1
	10	0	0	0	1

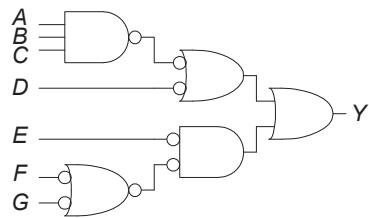
$$Y = A\bar{B}C + D$$

	AB	00	01	11	10
CD	00	0	0	1	0
D	01	0	1	1	1
	11	0	1	1	0
	10	0	0	0	0

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



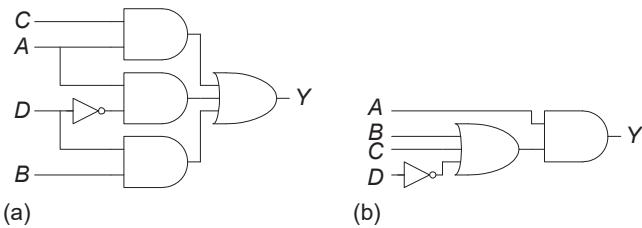
Exercise 2.27



$$\begin{aligned} Y &= ABC + \bar{D} + (\bar{F} + \bar{G})\bar{E} \\ &= ABC + \bar{D} + \bar{E}\bar{F} + \bar{E}\bar{G} \end{aligned}$$

Exercise 2.29

Two possible options are shown below:



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 2.33

The equation can be written directly from the description:

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

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	$A_{3:2}$	00	01	11	10
$A_{1:0}$		00	01	11	10
00		0	0	1	0
01		0	0	0	1
11		1	0	1	0
10		0	1	0	0

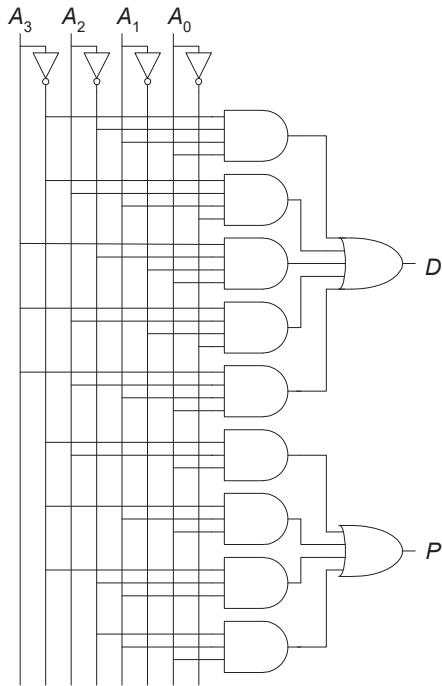
$$D = \bar{A}_3 \bar{A}_2 A_1 A_0 + \bar{A}_3 A_2 A_1 \bar{A}_0 + A_3 \bar{A}_2 \bar{A}_1 A_0 \\ + A_3 A_2 \bar{A}_1 \bar{A}_0 + A_3 A_2 A_1 A_0$$

P	$A_{3:2}$	00	01	11	10
$A_{1:0}$		00	01	11	10
00		0	0	0	0
01		0	1	1	0
11		1	1	0	1
10		1	0	0	0

$$P = \bar{A}_3 \underline{A}_2 A_0 + \bar{A}_3 A_1 A_0 + \bar{A}_3 \bar{A}_2 A_1 \\ + A_2 A_1 A_0$$

$$P = \bar{A}_3 A_1 A_0 + \bar{A}_3 \bar{A}_2 A_1 + \bar{A}_2 A_1 A_0 \\ + \underline{A}_2 \underline{A}_1 A_0$$

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

**Exercise 2.37**

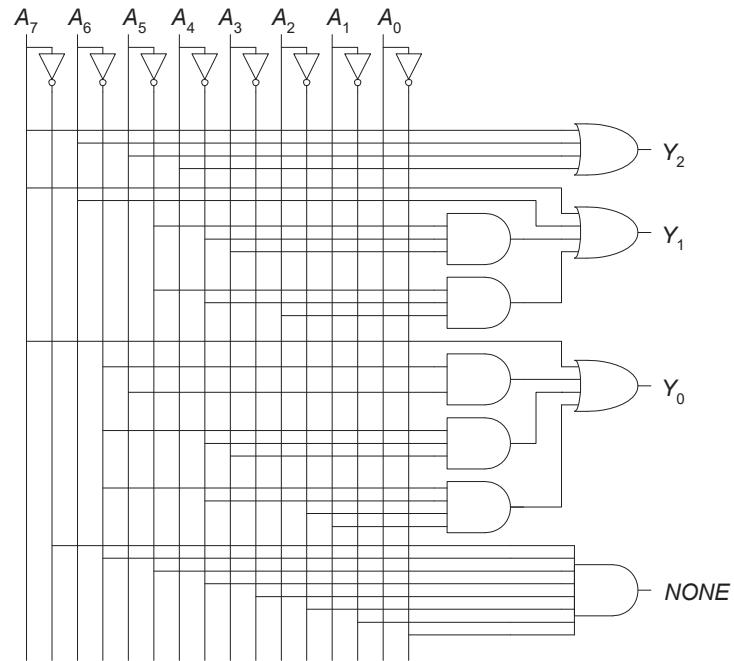
The equations and circuit for \$Y_{2:0}\$ is the same as in Exercise 2.25, repeated here for convenience.

\$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

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

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

$$Y_0 = A_7 + \overline{A}_6 A_5 + \overline{A}_6 \overline{A}_4 A_3 + \overline{A}_6 \overline{A}_4 \overline{A}_2 A_1$$



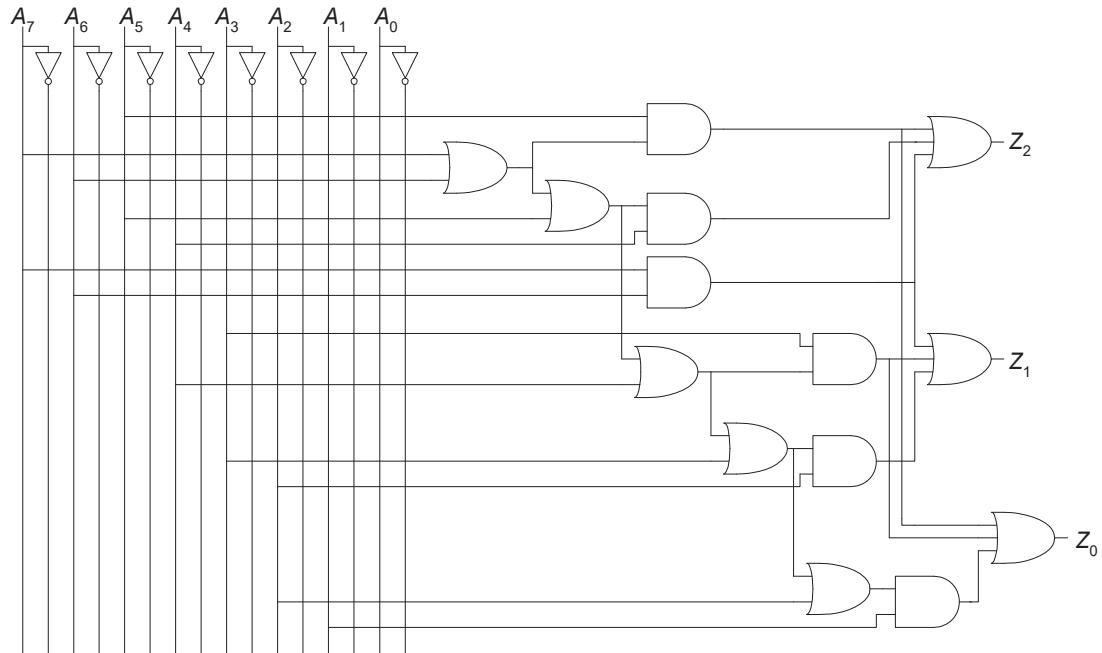
The truth table, equations, and circuit for $Z_{2:0}$ are as follows.

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

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

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

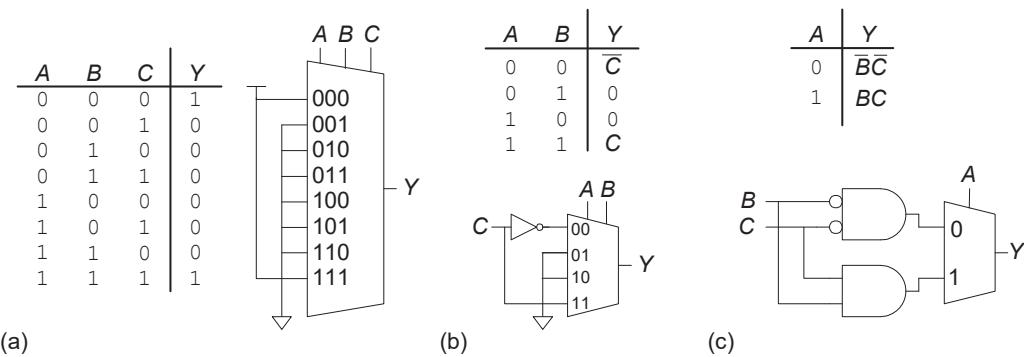
$$Z_0 = A_1(A_2 + A_3 + A_4 + A_5 + A_6 + A_7) + \\ A_3(A_4 + A_5 + A_6 + A_7) + A_5(A_6 + A_7)$$



Exercise 2.39

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

Exercise 2.41



Exercise 2.43

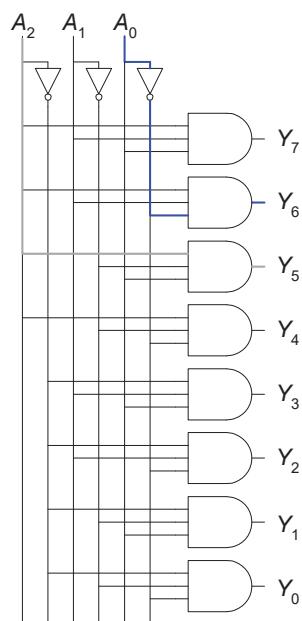
$$t_{pd} = 3t_{pd_NAND2} = \mathbf{60 \text{ ps}}$$

$$t_{cd} = t_{cd_NAND2} = \mathbf{15 \text{ ps}}$$

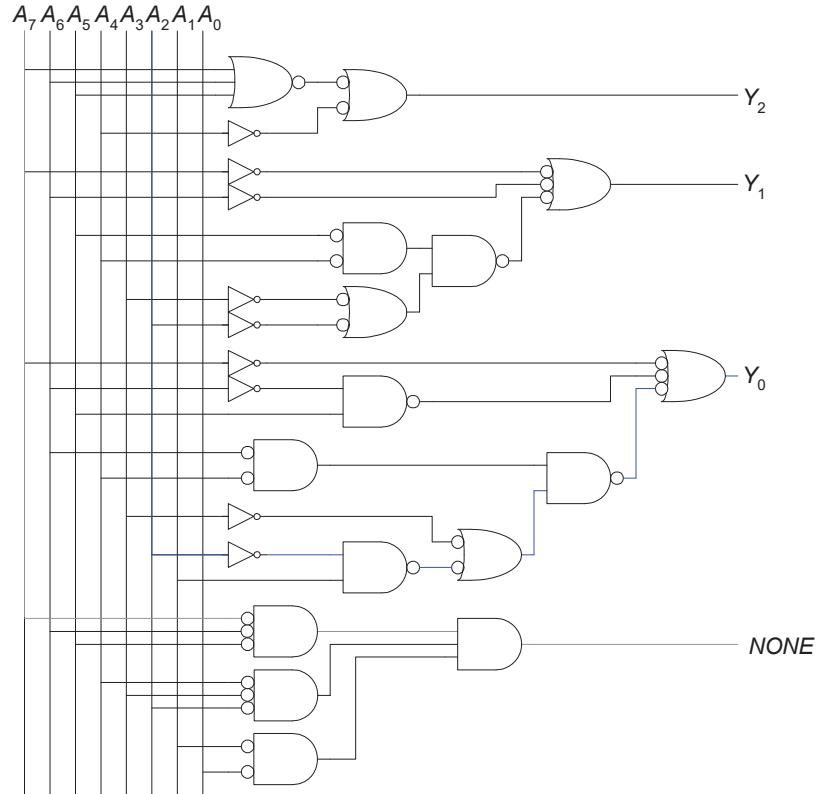
Exercise 2.45

$$\begin{aligned} t_{pd} &= t_{pd_NOT} + t_{pd_AND3} \\ &= 15 \text{ ps} + 40 \text{ ps} \\ &= \mathbf{55 \text{ ps}} \end{aligned}$$

$$\begin{aligned} t_{cd} &= t_{cd_AND3} \\ &= \mathbf{30 \text{ ps}} \end{aligned}$$



Exercise 2.47



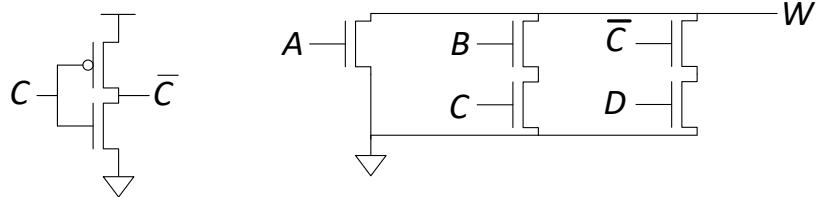
$$\begin{aligned} t_{pd} &= t_{pd_INV} + 3t_{pd_NAND2} + t_{pd_NAND3} \\ &= [15 + 3(20) + 30] \text{ ps} \\ &= \mathbf{105 \text{ ps}} \end{aligned}$$

$$\begin{aligned} t_{cd} &= t_{cd_NOT} + t_{cd_NAND2} \\ &= [10 + 15] \text{ ps} \\ &= \mathbf{25 \text{ ps}} \end{aligned}$$

Exercise 2.49

(a) $W = \overline{A + BC + \bar{C}D}$

Pull-down network directly from equation:



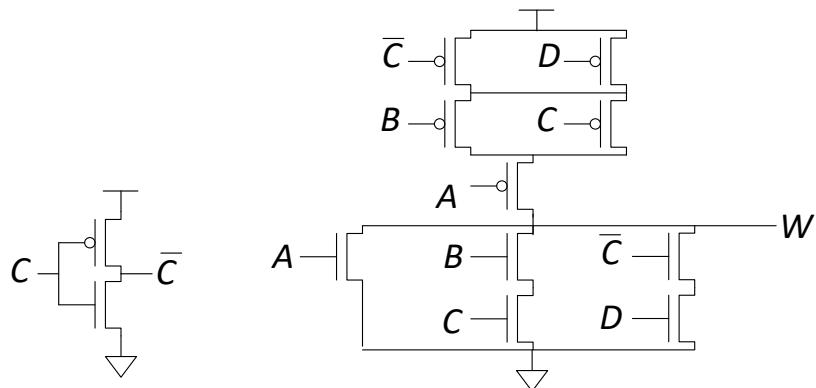
Now perform DeMorgan's on the equation to make it easy to draw the pull-up network.

$W = \overline{A + BC + \bar{C}D}$

$W = \bar{A} * \overline{BC} * \overline{\bar{C}D}$

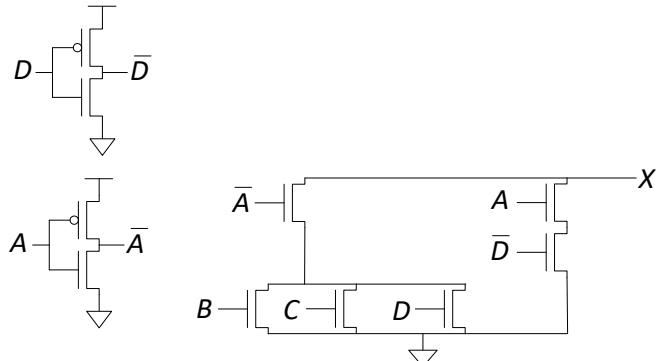
$W = \bar{A} * (\bar{B} + \bar{C}) * (C + \bar{D})$

Now we add the pull-up network according to this equation. Note that the method we used in Chapter 1 of changing parallel (OR) connections to serial (AND) connections and vice versa is graphically applying DeMorgan's theorem.



$$(b) \quad X = \overline{\bar{A}(B + C + D) + A\bar{D}}$$

First, we build the pull-down network directly using the equation above:

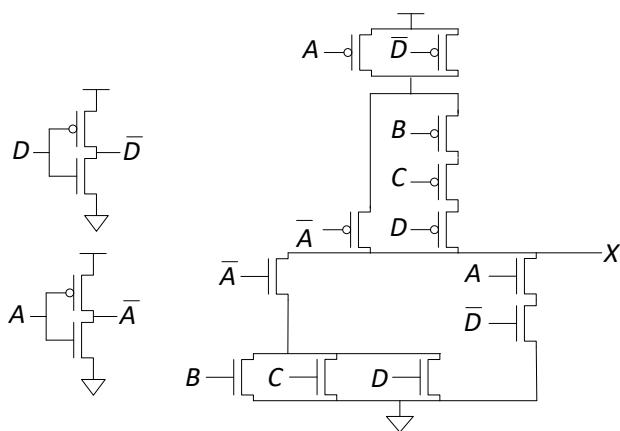


Now perform DeMorgan's on the equation to draw the pull-up network.

$$\begin{aligned} X &= \overline{\bar{A}(B + C + D) + A\bar{D}} \\ X &= \overline{\bar{A}(B + C + D)} * \overline{A\bar{D}} \\ X &= (A + (\bar{B} + \bar{C} + \bar{D})) * (\bar{A} + D) \\ X &= (A + \bar{B}\bar{C}\bar{D}) * (\bar{A} + D) \end{aligned}$$

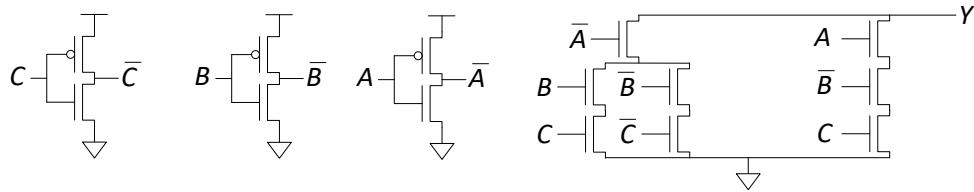
(Note that we could reduce it further by using distributivity, but that wouldn't reduce the number of transistors.)

Now we add the pull-up network according to this equation. Note that the method we used in Chapter 1 of changing parallel (OR) connections to serial (AND) connections and vice versa is graphically applying DeMorgan's theorem.



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

First, we build the pull-down network directly using the equation above:

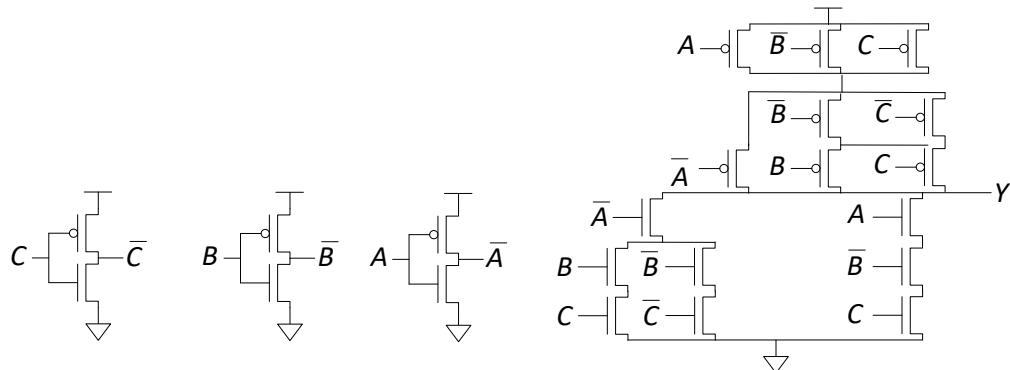


Now perform DeMorgan's on the equation to draw the pull-up network.

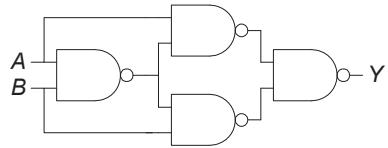
$$\begin{aligned} Y &= \overline{\bar{A}(BC + \bar{B}\bar{C})} + A\bar{B}C \\ Y &= \overline{\bar{A}(BC + \bar{B}\bar{C})} * \overline{A\bar{B}C} \\ Y &= (A + (\overline{BC + \bar{B}\bar{C}})) * (\bar{A} + B + \bar{C}) \\ Y &= (A + (\overline{BC} * \overline{\bar{B}\bar{C}})) * (\bar{A} + B + \bar{C}) \\ Y &= (A + ((\bar{B} + \bar{C}) * (B + C))) * (\bar{A} + B + \bar{C}) \end{aligned}$$

(Note that we could reduce it further by using distributivity, but that wouldn't reduce the number of transistors.)

Now we add the pull-up network according to this equation. Note that the method we used in Chapter 1 of changing parallel (OR) connections to serial (AND) connections and vice versa is graphically applying DeMorgan's theorem.



Question 2.1



Question 2.3

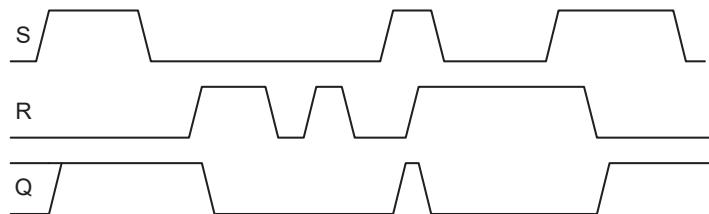
A tristate buffer has two inputs and three possible outputs: 0, 1, and Z. One of the inputs is the data input and the other input is a control input, often called the *enable* input. When the enable input is 1, the tristate buffer transfers the data input to the output; otherwise, the output is high 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 2.5

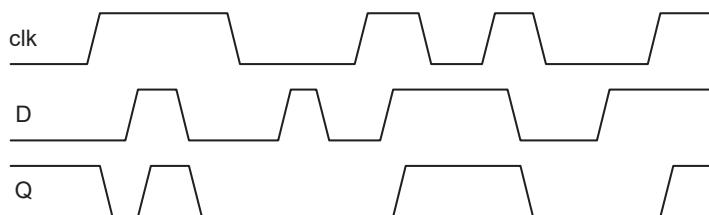
A circuit's contamination delay might be less than its propagation delay because the circuit may operate over a range of temperatures and supply voltages, for example, 3-3.6 V for LVC MOS (low voltage CMOS) chips. As temperature increases and voltage decreases, circuit delay increases. Also, the circuit 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

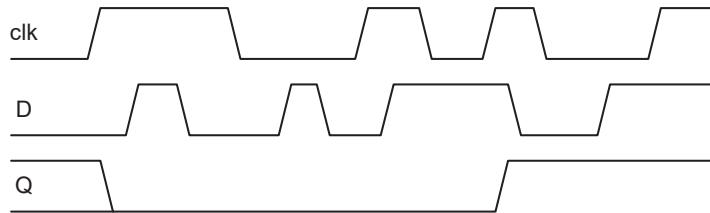
Exercise 3.1



Exercise 3.3



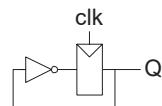
Exercise 3.5



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 $\bar{S} = 0$ and $\bar{R} = 1$, the circuit sets Q to 1. When $\bar{S} = 1$ and $\bar{R} = 0$, the circuit resets Q to 0. When both \bar{S} and \bar{R} are 1, the circuit remembers the old value. And when both \bar{S} and \bar{R} are 0, the circuit drives both outputs to 1.

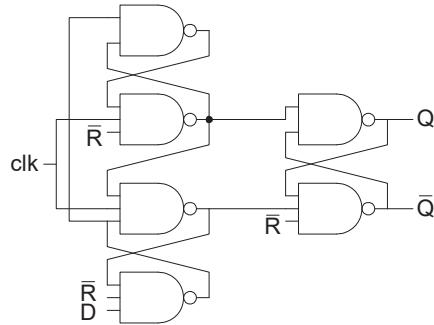
Exercise 3.9



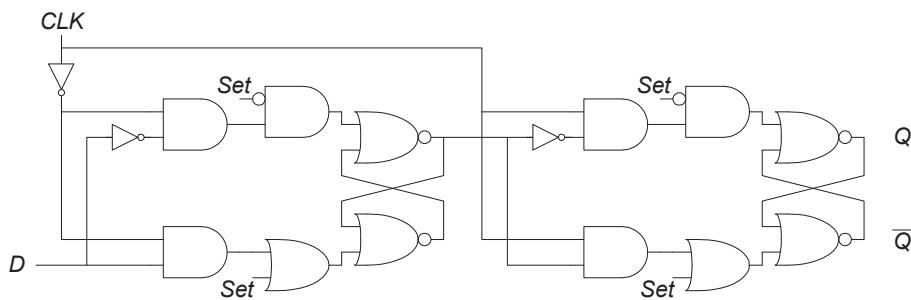
Exercise 3.11

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

Exercise 3.13



Exercise 3.15



Exercise 3.17

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

Exercise 3.19

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

Exercise 3.21

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 3.23

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 3.1 State encoding for Exercise 3.23

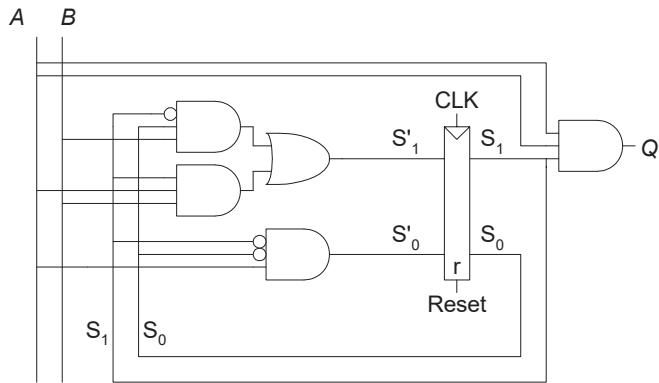
current state		inputs		next state		output
s ₁	s ₀	a	b	s' ₁	s' ₀	q
0	0	0	X	0	0	0
0	0	1	X	0	1	0
0	1	X	0	0	0	0
0	1	X	1	1	0	0
1	0	1	1	1	0	1
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0

TABLE 3.2 Combined state transition and output table with binary encodings for Exercise 3.23

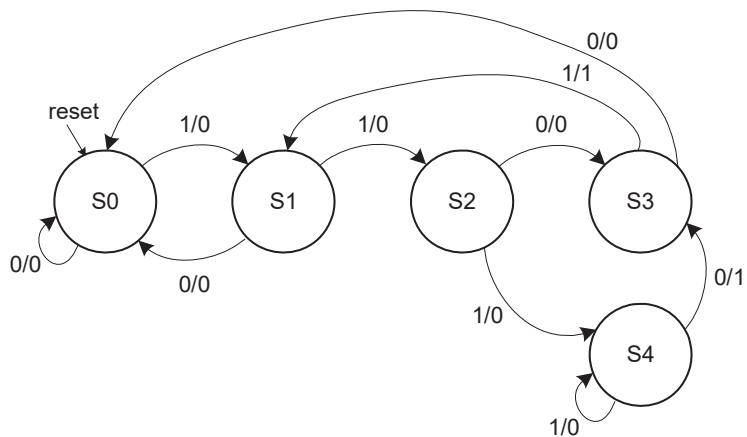
$$S_1 = \overline{S}_1 S_0 B + S_1 A B$$

$$S_0 = \overline{S}_1 \overline{S}_0 A$$

$$Q' = S_1 A B$$



Exercise 3.25



state	encoding $s_{1:0}$
S_0	000
S_1	001

TABLE 3.3 State encoding for Exercise 3.25

state	encoding s _{1:0}
S2	010
S3	100
S4	101

TABLE 3.3 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
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

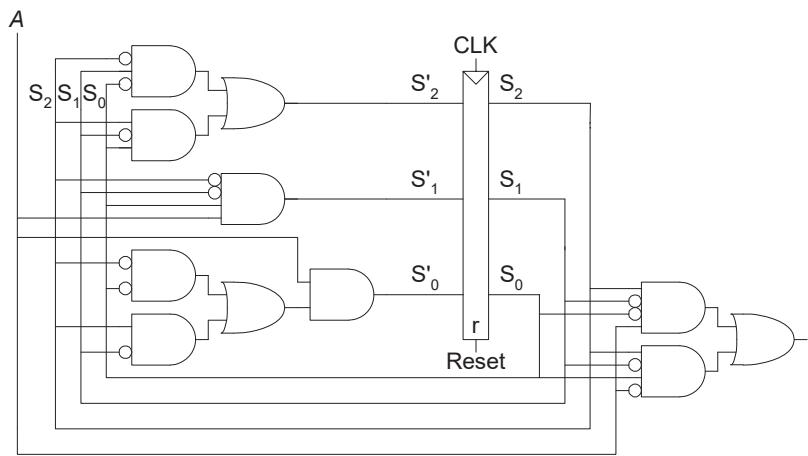
TABLE 3.4 Combined state transition and output table with binary encodings for Exercise 3.25

$$S_2 = \overline{S}_2 S_1 \overline{S}_0 + S_2 \overline{S}_1 S_0$$

$$S_1 = \overline{S}_2 \overline{S}_1 S_0 A$$

$$S_0 = A(\overline{S}_2 \overline{S}_0 + S_2 \overline{S}_1)$$

$$Q = S_2 \overline{S}_1 \overline{S}_0 A + S_2 \overline{S}_1 S_0 \overline{A}$$



Exercise 3.27

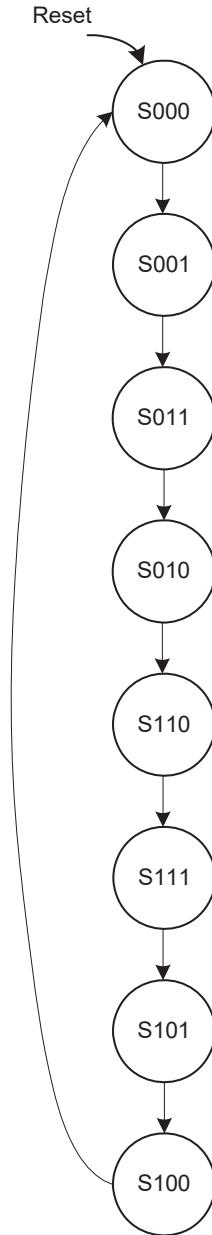


FIGURE 3.1 State transition diagram for Exercise 3.27

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 3.5 State transition table for Exercise 3.27

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

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

$$S'_0 = \overline{S_2 \oplus S_1}$$

$$Q_2 = S_2$$

$$Q_1 = S_1$$

$$Q_0 = S_0$$

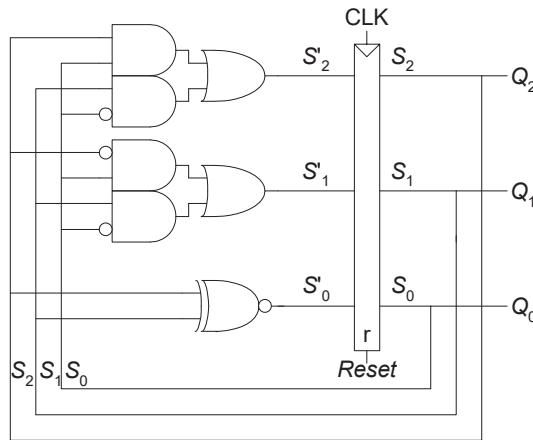


FIGURE 3.2 Hardware for Gray code counter FSM for Exercise 3.27

Exercise 3.29

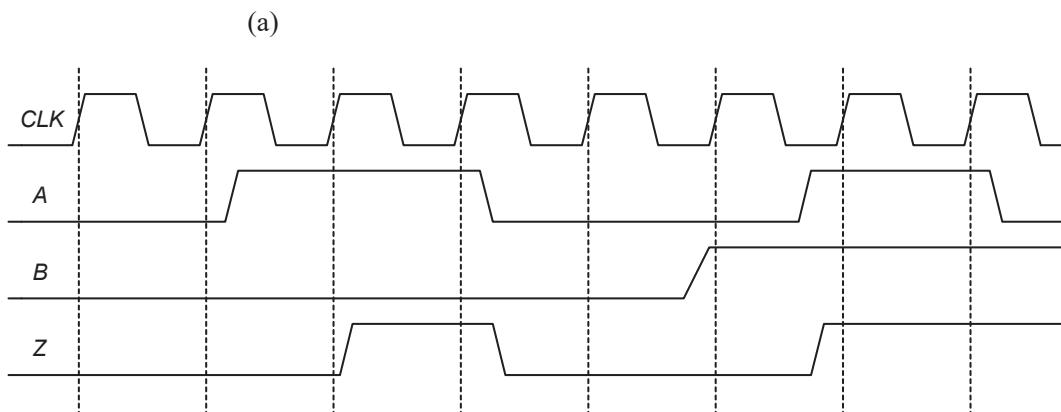


FIGURE 3.3 Waveform showing Z output for Exercise 3.29

(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)

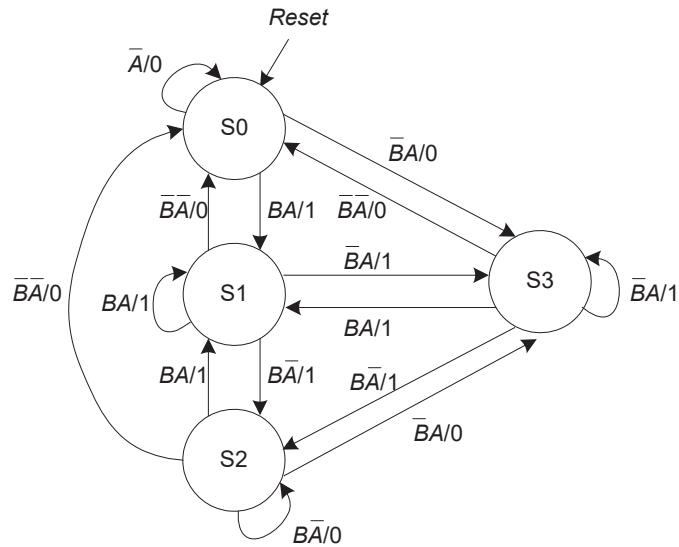


FIGURE 3.4 State transition diagram for Exercise 3.29

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

current state $s_{1:0}$	inputs		next state $s'_{1:0}$	output z
	b	a		
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 3.6 State transition table for Exercise 3.29

current state $s_{1:0}$	inputs		next state $s'_{1:0}$	output z
	b	a		
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 3.6 State transition table for Exercise 3.29

$$S_1 = \bar{B}A(\bar{S}_1 + S_0) + B\bar{A}(S_1 + \bar{S}_0)$$

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

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

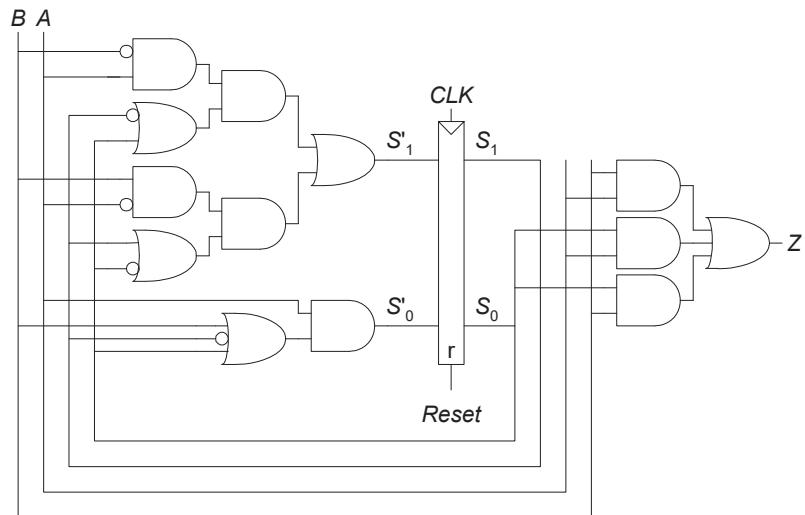


FIGURE 3.5 Hardware for FSM of Exercise 3.26

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 3.31

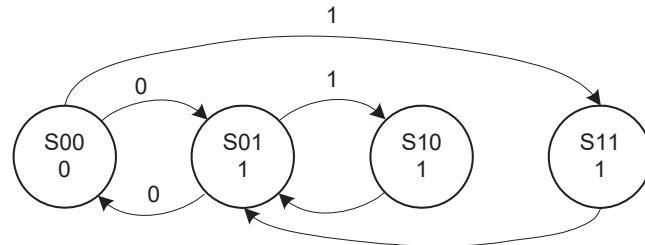
This finite state machine is a divide-by-two counter (see Section 3.4.2) when $X = 0$. When $X = 1$, the output, Q , is HIGH.

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
0	1	1	1	0
1	X	X	0	1

TABLE 3.7 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 3.8 Output table for Exercise 3.31

**Exercise 3.33**

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

$$\begin{aligned} t_{pd} &= 3t_{pd_XOR} \\ &= 3 \times 100 \text{ ps} \\ &= \mathbf{300 \text{ ps}} \end{aligned}$$

Next, we calculate the cycle time:

$$\begin{aligned} T_c &\geq t_{pcq} + t_{pd} + t_{setup} \\ &\geq [70 + 300 + 60] \text{ ps} \\ &= 430 \text{ ps} \\ f &= 1 / 430 \text{ ps} = \mathbf{2.33 \text{ GHz}} \end{aligned}$$

(b)

$$\begin{aligned} T_c &\geq t_{pcq} + t_{pd} + t_{setup} + t_{skew} \\ \text{Thus,} \end{aligned}$$

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

(c)

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

$$\begin{aligned} t_{cd} &= t_{cd_XOR} \\ &= 55 \text{ ps} \end{aligned}$$

$$t_{ccq} + t_{cd} > t_{hold} + t_{skew}$$

Thus,

$$\begin{aligned} t_{skew} &< (t_{ccq} + t_{cd}) - t_{hold} \\ &< (50 + 55) - 20 \\ &< \mathbf{85 \text{ ps}} \end{aligned}$$

(d)

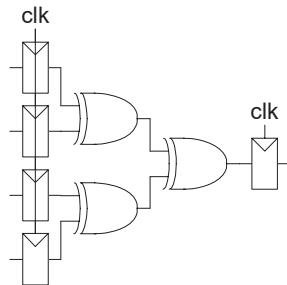


FIGURE 3.6 Alyssa's improved circuit for Exercise 3.33

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

$$\begin{aligned} t_{pd} &= 2t_{pd_XOR} \\ &= 2 \times 100 \text{ ps} \\ &= \mathbf{200 \text{ ps}} \end{aligned}$$

$$\begin{aligned} t_{cd} &= 2t_{cd_XOR} \\ &= 2 \times 55 \text{ ps} \\ &= \mathbf{110 \text{ ps}} \end{aligned}$$

Next, we calculate the cycle time:

$$\begin{aligned} T_c &\geq t_{pcq} + t_{pd} + t_{\text{setup}} \\ &\geq [70 + 200 + 60] \text{ ps} \\ &= 330 \text{ ps} \\ f &= 1 / 330 \text{ ps} = \mathbf{3.03 \text{ GHz}} \end{aligned}$$

$$\begin{aligned} t_{\text{skew}} &< (t_{ccq} + t_{cd}) - t_{\text{hold}} \\ &< (50 + 110) - 20 \\ &< \mathbf{140 \text{ ps}} \end{aligned}$$

Exercise 3.35

$$(a) T_c = 1 / 40 \text{ MHz} = 25 \text{ ns}$$

$$\begin{aligned} T_c &\geq t_{pcq} + Nt_{\text{CLB}} + t_{\text{setup}} \\ 25 \text{ ns} &\geq [0.72 + N(0.61) + 0.53] \text{ ps} \\ \text{Thus, } N &< 38.9 \\ \mathbf{N} &= \mathbf{38} \end{aligned}$$

(b)

$$\begin{aligned} t_{\text{skew}} &< (t_{ccq} + t_{cd_CLB}) - t_{\text{hold}} \\ &< [(0.5 + 0.3) - 0] \text{ ns} \\ &< \mathbf{0.8 \text{ ns} = 800 \text{ ps}} \end{aligned}$$

Exercise 3.37

$$P(\text{failure})/\text{sec} = 1/\text{MTBF} = 1/(50 \text{ years} * 3.15 \times 10^7 \text{ sec/year}) = \mathbf{6.34 \times 10^{-10}} \quad (\text{EQ 3.26})$$

$$\begin{aligned} P(\text{failure})/\text{sec} \text{ waiting for one clock cycle: } & N * (T_0/T_c) * e^{-(T_c-t_{\text{setup}})/\tau} \\ & = 0.5 * (110/1000) * e^{-(1000-70)/100} = 5.0 \times 10^{-6} \end{aligned}$$

$$\begin{aligned} P(\text{failure})/\text{sec} \text{ waiting for two clock cycles: } & N * (T_0/T_c) * [e^{-(T_c-t_{\text{setup}})/\tau}]^2 \\ & = 0.5 * (110/1000) * [e^{-(1000-70)/100}]^2 = 4.6 \times 10^{-10} \end{aligned}$$

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

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:

$$\begin{aligned} P(\text{failure}) &= e^{-\frac{t}{\tau}} \\ MTBF &= \frac{1}{P(\text{failure})} = e^{\frac{T_c - t_{\text{setup}}}{\tau}} \\ \frac{MTBF_2}{MTBF_1} &= 10 = e^{\frac{T_{c2} - T_{c1}}{30ps}} \end{aligned}$$

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 T_0 (20 ps) and the increased time (69 ps).

Question 3.1

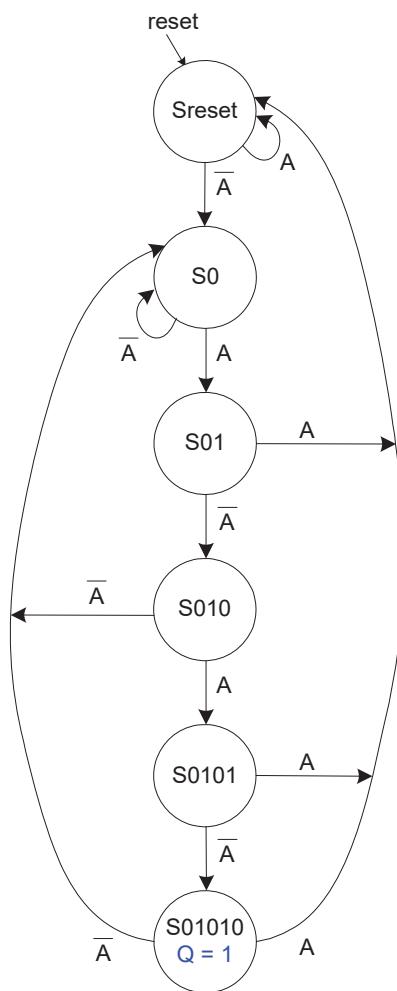


FIGURE 3.7 State transition diagram for Question 3.1

current state $s_5 : 0$	input a	next state
		$s' 5 : 0$
000001	0	000010
000001	1	000001
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 3.9 State transition table for Question 3.1

$$S_5 = S_4 A$$

$$S_4 = S_3 A$$

$$S_3 = S_2 A$$

$$S_2 = S_1 A$$

$$S_1 = A(S_1 + S_3 + S_5)$$

$$S_0 = A(S_0 + S_2 + S_4 + S_5)$$

$$Q = S_5$$

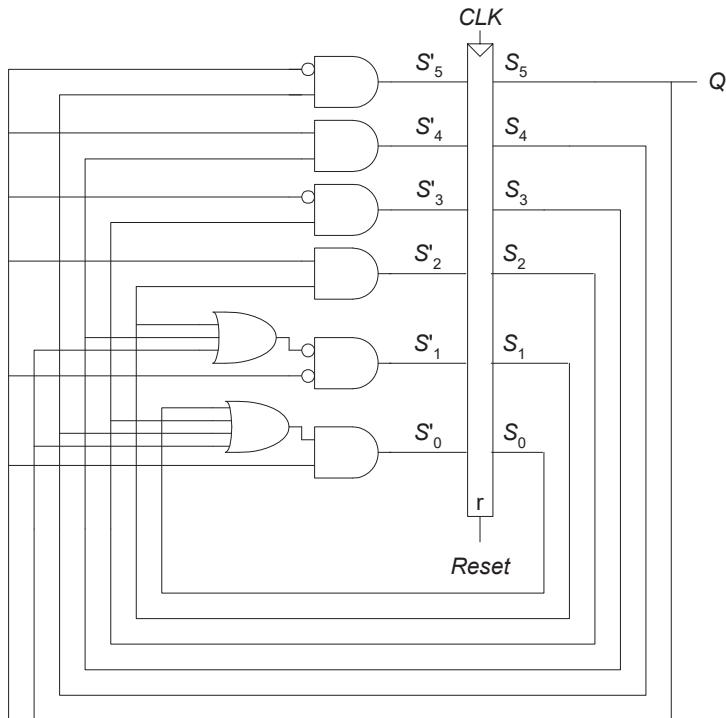


FIGURE 3.8 Finite state machine hardware for Question 3.1

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.

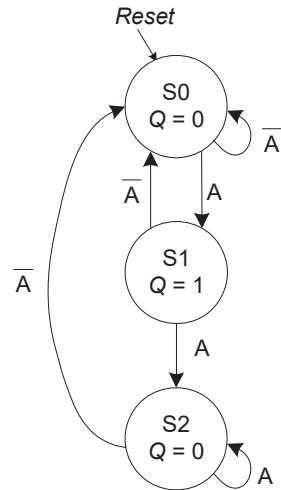
Question 3.5

FIGURE 3.9 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 3.10 State transition table for Question 3.5

$$s'_1 = AS_1$$

$$s'_0 = AS_1S_0$$

$$Q = S_1$$

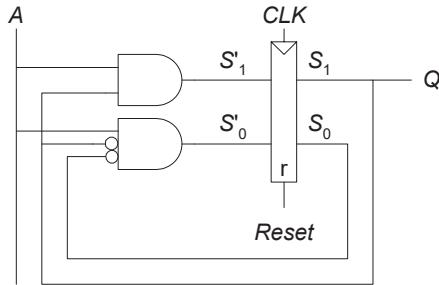


FIGURE 3.10 Finite state machine hardware for Question 3.5

Question 3.7

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

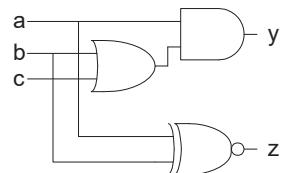
Question 3.9

Without the added buffer, the propagation delay through the logic, t_{pd} , must be less than or equal to $T_c - (t_{pcq} + t_{\text{setup}})$. 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_{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 $T_c + t_{cd_BUF} - (t_{pcq} + t_{\text{setup}})$.

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 4.3

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 4.5

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 4.7

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)
    begin
        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'bxx) begin
                $display("%d tests completed with %d errors",
                         vectornum, errors);
                $finish;
            end
        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;
    end process;

```

(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 4.9

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;

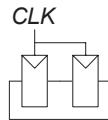
entity ex4_9 is
  port(a,
        b,
        c: in STD_LOGIC;
        y: out STD_LOGIC_VECTOR(0 downto 0));
end;

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 4.11

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

**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 4.15

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

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 4.17

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 4.19

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 4.21

SystemVerilog

```

module priority_encoder2(input logic [7:0] a,
                        output logic [2:0] y, z,
                        output logic      none);
begin
    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'b00000001?: begin y = 3'd1; none = 1'b0; end
            8'b0000001???: 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'b0000101???: z = 3'b010;
            8'b0001001???: z = 3'b010;
            8'b0010001???: z = 3'b010;
            8'b0100001???: z = 3'b010;
            8'b1000001???: z = 3'b010;
            8'b0000111???: z = 3'b011;
            8'b00001011???: z = 3'b011;
            8'b00010011???: z = 3'b011;
            8'b00100011???: z = 3'b011;
            8'b01000011???: z = 3'b011;
            8'b10000011???: z = 3'b011;
            8'b0001111???: z = 3'b100;
            8'b0010111???: z = 3'b100;
            8'b0101111???: z = 3'b100;
            8'b0101111???: z = 3'b101;
            8'b1011111???: z = 3'b101;
            8'b1111111???: 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 "00000101-" => z <= "001";
            when "0001001-" => z <= "001";
            when "0010001-" => z <= "001";
            when "0100001-" => z <= "001";
            when "1000001-" => z <= "001";
            when "00000111" => z <= "010";
            when "0001011-" => z <= "010";
            when "0100011-" => z <= "010";
            when "1000011-" => z <= "010";
            when "0000111--" => z <= "011";
            when "0001011--" => z <= "011";
            when "0100011--" => z <= "011";
            when "1000011--" => z <= "011";
            when "0001111--" => z <= "100";
            when "0101111--" => z <= "100";
            when "1001111--" => z <= "100";
            when "0111111--" => z <= "101";
            when "1011111--" => z <= "101";
            when "1111111--" => z <= "110";
            when others      => z <= "000";
        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;
```

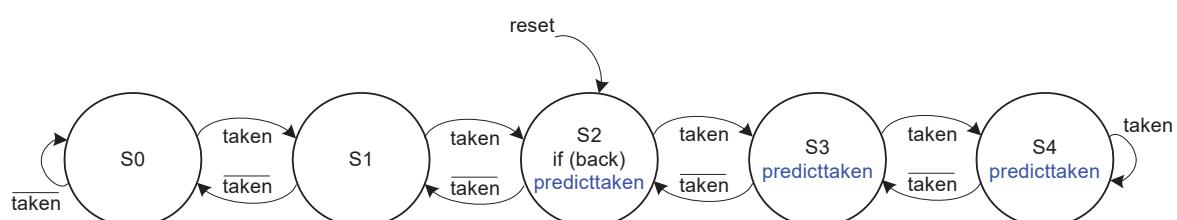
Exercise 4.25

FIGURE 4.1 State transition diagram for Exercise 4.25

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 4.29

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 4.31

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 4.33

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 4.35

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 4.37**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 4.39

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, nland, 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 4.41

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 4.43

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 4.45

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 4.47

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

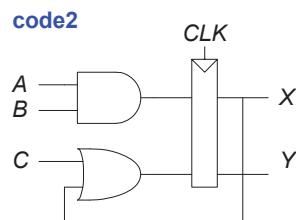
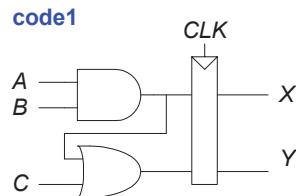
```
library IEEE; use IEEE.STD_LOGIC_1164.all;

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 4.49

They do not have the same function.

**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').

Question 4.1

SystemVerilog

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

VHDL

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

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

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:

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

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

(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:

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

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

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.

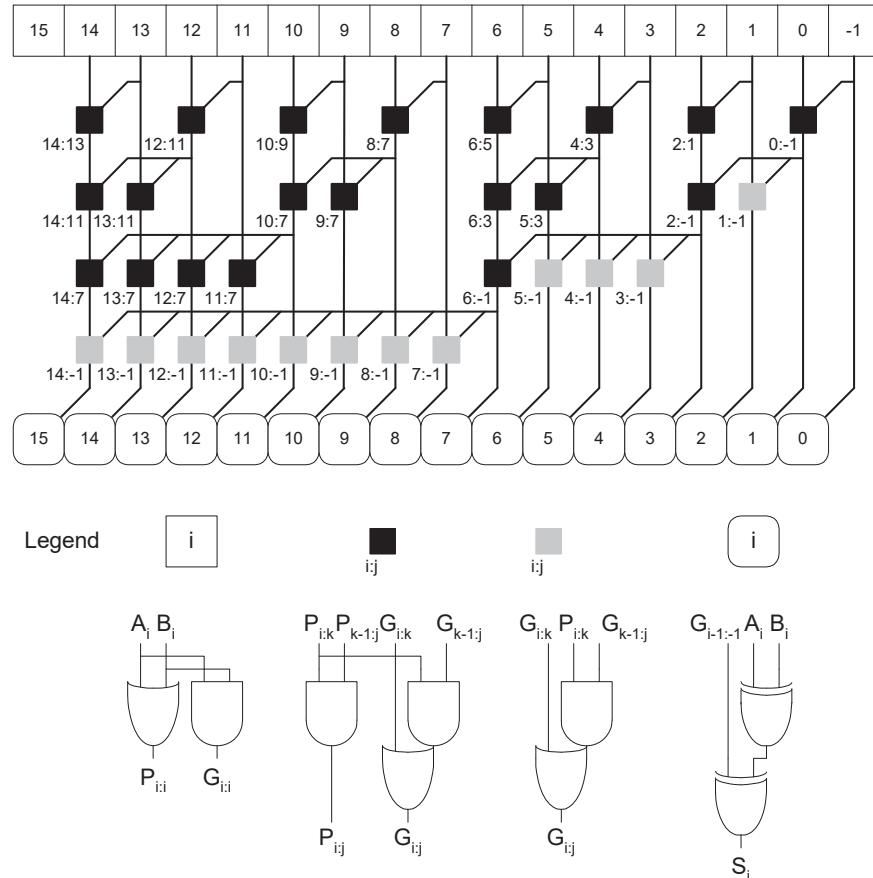
Exercise 5.5

FIGURE 5.1 16-bit prefix adder with “gray cells”

Exercise 5.7

(a) We show an 8-bit priority circuit in Figure 5.2. In the figure $X_7 = \bar{A}_7$, $X_{7:6} = \bar{A}_7\bar{A}_6$, $X_{7:5} = \bar{A}_7\bar{A}_6\bar{A}_5$, and so on. The priority encoder’s delay is $\log_2 N$ 2-input AND gates followed by a final row of 2-input AND gates. The final stage is an $(N/2)$ -input OR gate. Thus, in general, the delay of an N -input priority encoder is:

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

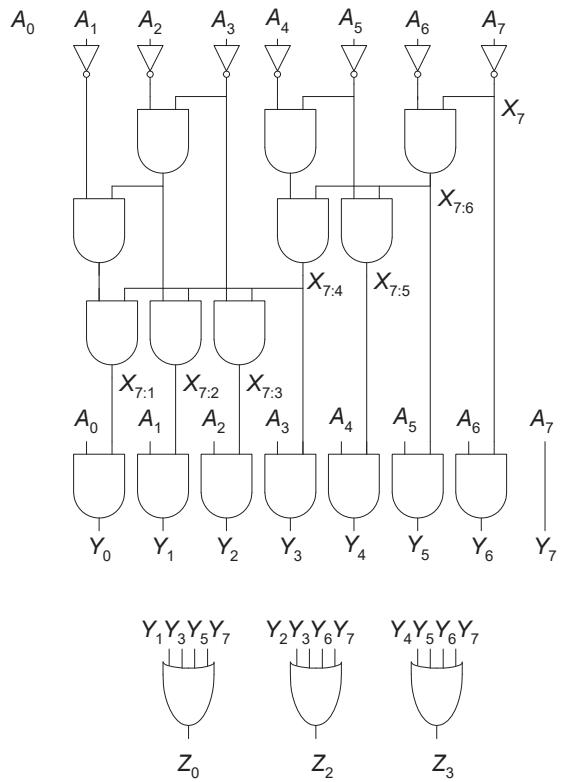


FIGURE 5.2 8-input priority encoder

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 5.9

(a) Answers will vary.

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

(b) Answers will vary.

-3 and 6: $-3 - 6 = 1101 - 0110 = 1101 + 1001 + 1 = 0111_2 (= -7)$, but overflow occurred – the result should be -9. The sign bit (most significant bit) is 0, so the 4-bit signed comparator of Figure 5.12 **incorrectly** computes that -3 is **not** less than 6.

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

Exercise 5.11

SystemVerilog

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

    logic [31:0] condinvb, sum;

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

    always_comb
        case (alucontrol)
            2'b00:   result = sum;                      // add
            2'b01:   result = sum;                      // subtract
            2'b10:   result = a & b;                    // and
            2'b11:   result = a | b;                    // or
            default: result = 32'bx;
        endcase
    endmodule
```

Exercise 5.13

SystemVerilog

```
module alu(input logic [31:0] a,
           input logic [31:0] b,
           input logic [2:0] alucontrol,
           output logic [31:0] result);

    logic [31:0] condinvb, sum;
    logic         cout;           // carry out of adder

    assign condinvb = alucontrol[0] ? ~b : b;
    assign {cout, sum} = a + condinvb + alucontrol[0];

    always_comb
        case (alucontrol)
            3'b000:   result = sum;                      // add
            3'b001:   result = sum;                      // subtract
            3'b010:   result = a & b;                    // and
            3'b011:   result = a | b;                    // or
            3'b101:   result = sum[31];                 // slt
            default: result = 32'bx;
        endcase
    endmodule
```

Exercise 5.15

SystemVerilog

```

module testbench();
    logic      clk, reset;
    logic [31:0] a, b, result, resultexpected;
    logic [1:0]  alucontrol;
    logic [31:0] vectornum, errors;
    logic [97:0] testvectors[10000:0];

    // instantiate device under test
    alu dut(a, b, alucontrol, result);

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

    // at start of test, load vectors
    // and pulse reset
    initial
        begin
            $readmemh("example.txt", testvectors);
            vectornum = 0; errors = 0;
            reset = 1; #22; reset = 0;
        end

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

    // check results on falling edge of clk
    always @ (negedge clk)
        if (~reset) begin // skip during reset
            if (result !== resultexpected) begin // check result
                $display("Error: inputs: a = %h, b = %h, alucontrol = %h,", a, b,
                         alucontrol);
                $display(" outputs: result = %h (%h expected)", result, resultexpected);
                errors = errors + 1;
            end
            vectornum = vectornum + 1;
            if (testvectors[vectornum] === 98'bx) begin
                $display("%d tests completed with %d errors",
                         vectornum, errors);
                $stop;
            end
        end
    end
endmodule

```

Testvectors:

```
// alucontrol_a_b_resultexpected
0_00000007_00000005_0000000C // add
0_AABBCCDD_00000005_ABBCCE2 // add
0_FF123456_FABCDEF1_F9CF1347 // add
1_00000007_00000005_00000002 // sub
1_00000005_00000007_FFFFFFFE // sub
1_AABBCCDD_11445588_99777755 // sub
2_FFFF0123_ABCDEFAA_ABCD0122 // and
2_AABBCCDD_00000008_00000008 // and
3_FFFF0123_ABCDEFAA_FFFFEBAB // or
3_AABBCCDD_00000008_AABBCCDD // or
```

Exercise 5.17**SystemVerilog**

```
module testbench();
    logic      clk, reset;
    logic [31:0] a, b, result, resultexpected;
    logic [2:0]  alucontrol;
    logic [31:0] vectornum, errors;
    logic [98:0] testvectors[10000:0];

    // instantiate device under test
    alu dut(a, b, alucontrol, result);

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

    // at start of test, load vectors
    // and pulse reset
    initial
    begin
        $readmemh("example.txt", testvectors);
        vectornum = 0; errors = 0;
        reset = 1; #22; reset = 0;
    end

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

    // check results on falling edge of clk
    always @ (negedge clk)
    if (~reset) begin // skip during reset
        if (result != resultexpected) begin // check result
            $display("Error: inputs: a = %h, b = %h, alucontrol = %h,", a, b,
                     alucontrol);
            $display(" outputs: result = %h (%h expected)", result, resultexpected);
            errors = errors + 1;
        end
    end
endmodule
```

```

    end
    vectornum = vectornum + 1;
    if (testvectors[vectornum] === 99'bx) begin
        $display("%d tests completed with %d errors",
            vectornum, errors);
        $stop;
    end
end
endmodule

```

Testvectors:

```

// alucontrol_a_b_resultexpected_flags
0_00000007_00000005_0000000C // add
0_AABBCCDD_00000005_AABCCE2 // add
0_FF123456_FABCDEF1_F9CF1347 // add
0_7FFFFFFF_00000002_80000001 // add
0_80000000_81234567_01234567 // add
1_80000000_81234567_FEDCBA99 // sub
1_7FFFFFFF_FFFFFFFE_80000001 // sub
1_00000007_00000005_00000002 // sub
1_00000005_00000007_FFFFFFFE // sub
1_AABBCCDD_11445588_99777755 // sub
1_7FFFFFFF_FFFFFFFF_80000000 // sub
2_FFFF0123_ABCDEFAA_ABCD0122 // and
2_AABBCCDD_00000008_00000008 // and
3_FFFF0123_ABCDEFAA_FFFFEFAB // or
3_AABBCCDD_00000008_AABBCCDD // or
5_00000007_00000005_00000000 // slt
5_80000000_81234567_00000001 // slt
5_80000000_00000001_00000000 // slt - wrong result due to overflow
5_7FFFFFFF_FFFFFFFE_00000001 // slt - wrong result due to overflow
5_0000FFF1_000FFF1_00000001 // slt

```

Exercise 5.19

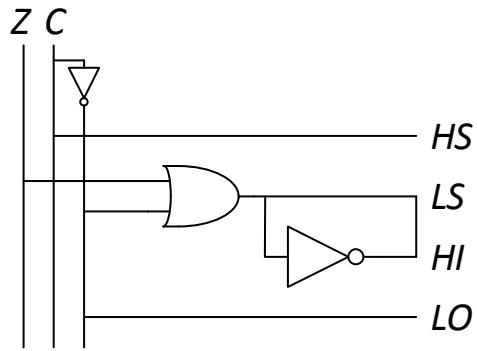
(a) $HS = C$

$LS = Z + \bar{C}$

$HI = \bar{Z}C = \bar{L}\bar{S}$

$LO = \bar{C} = \bar{H}\bar{S}$

(b)



Exercise 5.21

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.

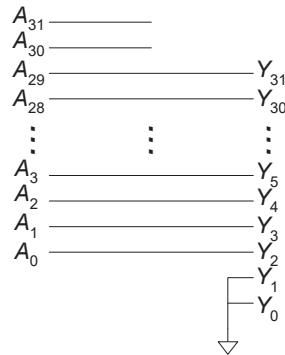


FIGURE 5.3 2-bit left shifter, 32-bit input and output

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 5.23

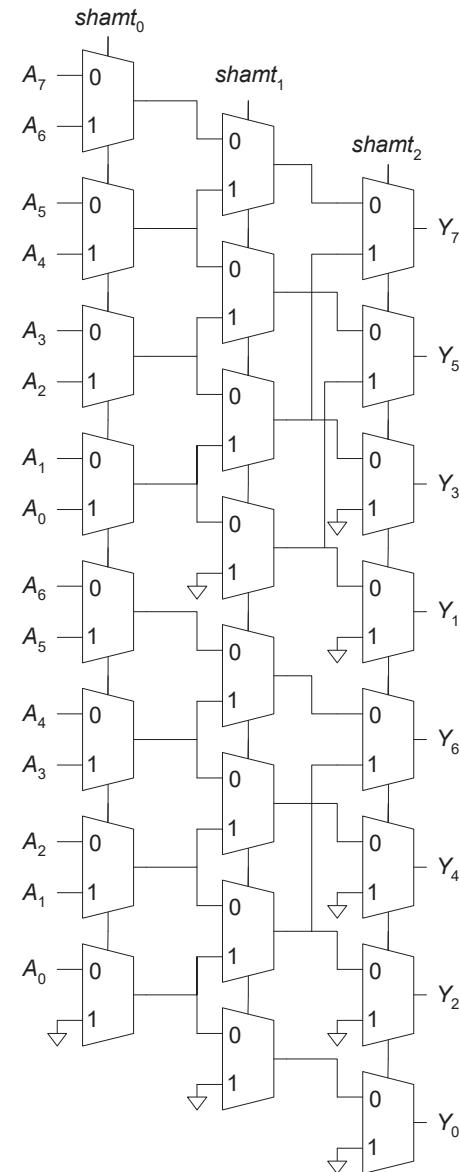


FIGURE 5.4 8-bit left shifter using 24 2:1 multiplexers

Exercise 5.25

(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 5.27

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

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

Exercise 5.29

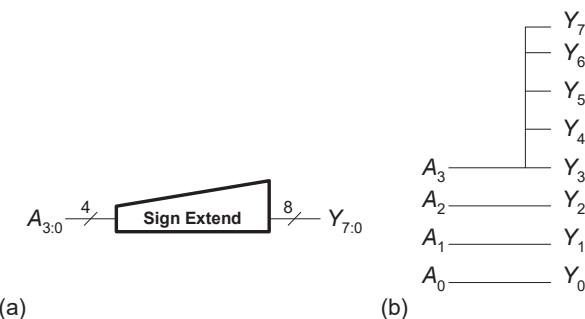


FIGURE 5.5 Sign extension unit (a) symbol, (b) underlying hardware

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 5.31

$$\begin{array}{r}
 & \begin{array}{c} 100.110 \\[-4pt] 111001.000 \end{array} \\
 1100 & \left[\begin{array}{r} 111001.000 \\ -1100 \\ \hline 001001\textcolor{blue}{0} \end{array} \right] \\
 & \left[\begin{array}{r} 001001\textcolor{blue}{0} \\ -1100 \\ \hline 11\textcolor{blue}{00} \end{array} \right] \\
 & \left[\begin{array}{r} 11\textcolor{blue}{00} \\ -1100 \\ \hline 0 \end{array} \right]
 \end{array}$$

Exercise 5.33

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

Exercise 5.35

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

Exercise 5.37

(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:
 $1\ 1000\ 0010\ 101\ 1001\ 0000\ 0000\ 0000 = \mathbf{0xC1590000}$

(b) $101010.0101 = 1.010100101 \times 2^5$
 Thus, the biased exponent = $127 + 5 = 132 = 1000\ 0100_2$
 In IEEE 754 single-precision floating-point format:
 $0\ 1000\ 0100\ 010\ 1001\ 0100\ 0000\ 0000 = \mathbf{0x42294000}$

(c) $-10001.00101 = -1.000100101 \times 2^4$
 Thus, the biased exponent = $127 + 4 = 131 = 1000\ 0011_2$
 In IEEE 754 single-precision floating-point format:
 $1\ 1000\ 0011\ 000\ 1001\ 0100\ 0000\ 0000 = \mathbf{0xC1894000}$

Exercise 5.39

- (a) 5.5
- (b) $-0000.0001_2 = -0.0625$
- (c) -8

Exercise 5.41

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 5.43

(a)

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

When adding these two numbers together, $0xC0D20004$ becomes:

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

$0x72407020$

(b)

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

$$1.101\ 1100\ 0000\ 0000\ 0000\ 01 \times 2^2$$

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

(c)

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

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

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

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 5.45

$$\begin{aligned}
 \text{(a) } 2(2^{31} - 1 - 2^{23}) &= 2^{32} - 2 - 2^{24} = 4,278,190,078 \\
 \text{(b) } 2(2^{31} - 1) &= 2^{32} - 2 = 4,294,967,294
 \end{aligned}$$

(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 5.47

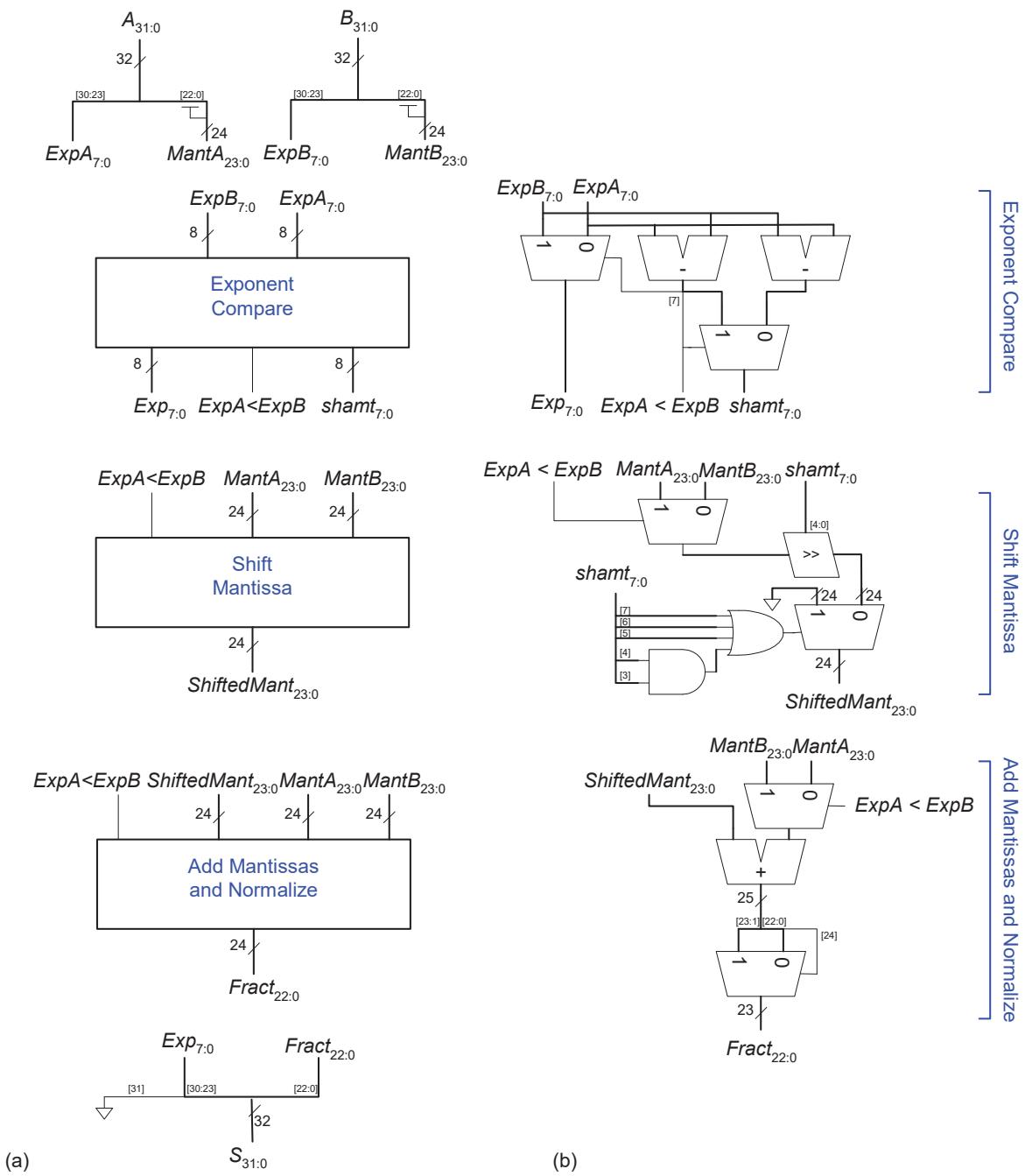


FIGURE 5.6 Floating-point adder hardware: (a) block diagram, (b) underlying hardware

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, bminusb;

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

    always_comb
        if (alessb) begin
            exp = expb;
            shamt = bminusb;
        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 bminusb: STD_LOGIC_VECTOR(7 downto 0);
begin
    begin
        aminusb <= expa - expb;
        bminusb <= expb - expa;
        alessb <= aminusb(7);

        exp <= expb when alessb = '1' else expa;
        shamt <= bminusb 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 adddmant(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_integer(unsigned(shamt)) when alessb = '1'
                           else SHIFT_RIGHT( unsigned(mantb), to_integer(unsigned(shamt)) );

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

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

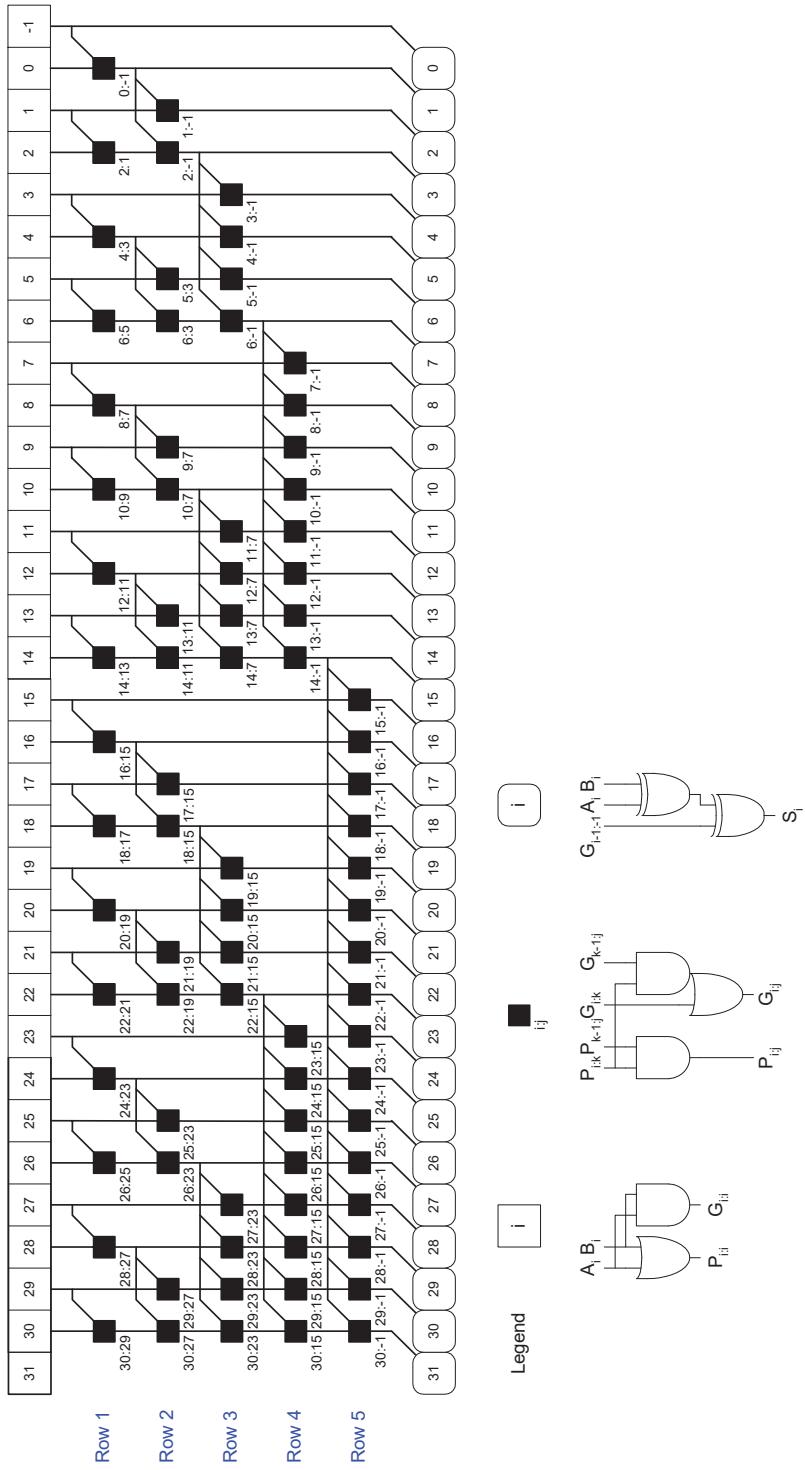
entity adddmant is
  port(alessb: in STD_LOGIC;
       manta: in STD_LOGIC_VECTOR(23 downto 0);
       mantb: in STD_LOGIC_VECTOR(23 downto 0);
       shmant: in STD_LOGIC_VECTOR(23 downto 0);
       exp_pre: in STD_LOGIC_VECTOR(7 downto 0);
       fract: out STD_LOGIC_VECTOR(22 downto 0);
       exp: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

architecture synth of adddmant 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 5.49

(a) Figure on next page



5.49 (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, gjk: in STD_LOGIC_VECTOR(15 downto 0);
          pij: out STD_LOGIC_VECTOR(15 downto 0);
          gjj: 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,
         gjk_1, gjk_2, gjk_3, gjk_4, gjk_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');
  gjk_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, gjk_1,
             p1, g1);

```

*(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, gjk_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, gjk_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] pnex, gnex);

    assign pnex = pleft & pright;
    assign gnex = 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, gjj:
          out STD_LOGIC_VECTOR(15 downto 0));
end;

architecture synth of pgblackblock is
begin
    pij <= pik and pkj;
    gjj <= 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;

```

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

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

We find the delays for each block:

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

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

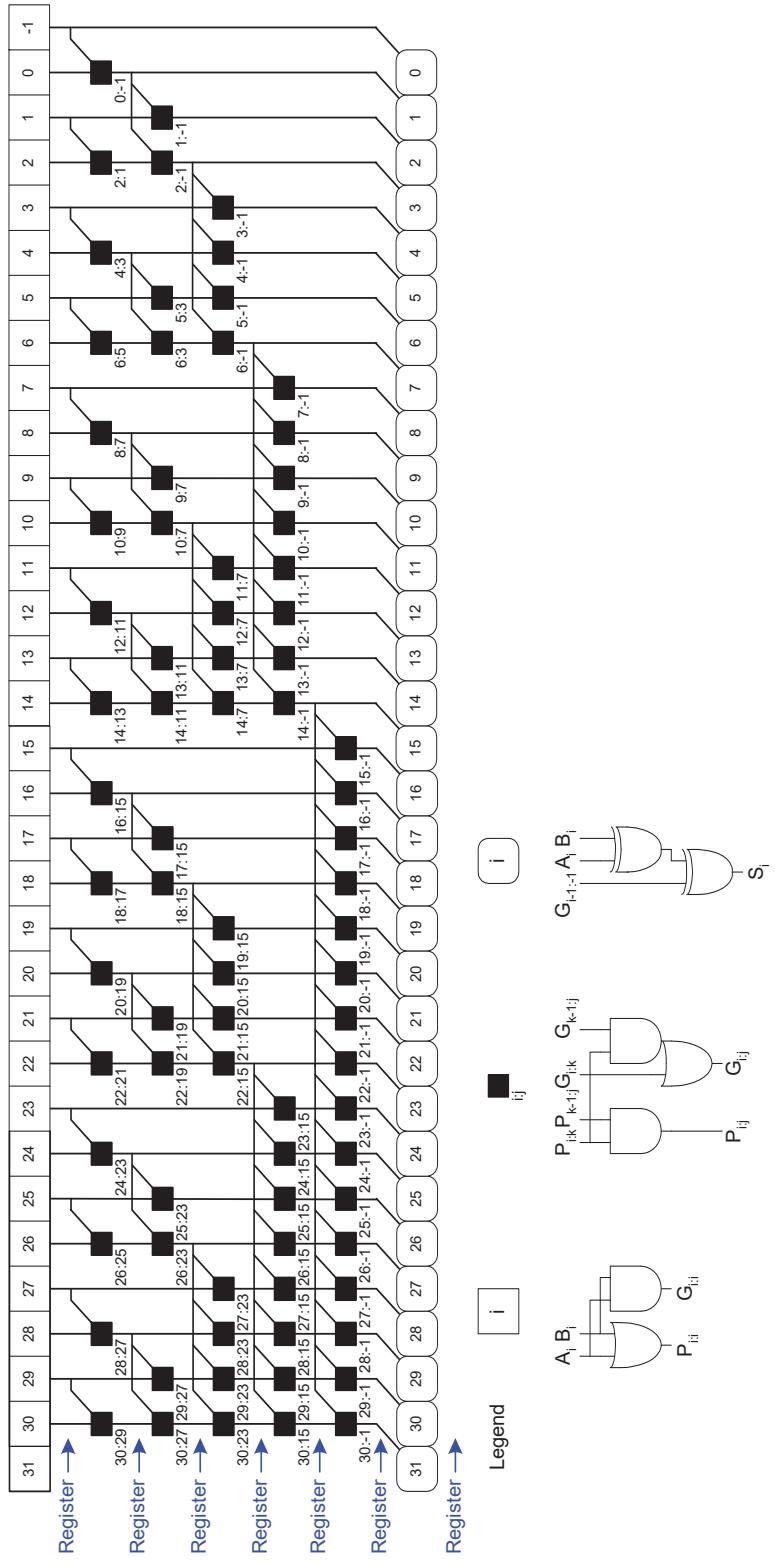
$$t_{XOR} = 100 \text{ ps}$$

Thus,

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

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

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



5.49 (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)          #(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]}, flop1_pg,
  {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)          #(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

```

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

```

```

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

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

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

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

begin

  -- p and g calculations
  row0_reg: pgblock port map(clk, a(30 downto 0), b(30 downto 0), p0, g0);
  row1_reg: row1 port map(clk, p0, g0, p_1_0, g_1_0, p1, g1);
  row2_reg: row2 port map(clk, p1, g1, 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'; flop1_g0: flop1 port map (clk, cin, g_1_0);
  flop1_p1: flop1 port map (clk, p_1_0, p_1_1);
  flop1_g1: flop1 port map (clk, g_1_0, g_1_1);
  flop1_p2: flop1 port map (clk, p_1_1, p_1_2);
  flop1_g2: flop1 port map (clk, g_1_1, g_1_2);
  flop1_p3: flop1 port map (clk, p_1_2, p_1_3); flop1_g3:
  flop1 port map (clk, g_1_2, g_1_3);
  flop1_p4: flop1 port map (clk, p_1_3, p_1_4);
  flop1_g4: flop1 port map (clk, g_1_3, g_1_4);
  flop1_p5: flop1 port map (clk, p_1_4, p_1_5);
  flop1_g5: flop1 port map (clk, g_1_4, g_1_5);

  -- generate sum and cout
  g5_all <= (g5&g_1_5);
  row6: ssumblock 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, gjj:
              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;
            gjj <= gik or (pik and gkj);
        end if;
    end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

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

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

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

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

```

```

        end if;
    end process;
end;

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

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

library IEEE; use IEEE.STD_LOGIC_1164.all;

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

architecture synth of row1 is
    component blackbox is
        port (clk:      in STD_LOGIC;
              pik, pkj: in STD_LOGIC_VECTOR(15 downto 0);
              gik, gjk: 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, gjk_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, 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, gjk_1, pij_1, gjij_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) <= gjij_1(15 downto 14);
g2(26 downto 25) <= gjij_1(13 downto 12);
g2(22 downto 21) <= gjij_1(11 downto 10);
g2(18 downto 17) <= gjij_1(9 downto 8);
g2(14 downto 13) <= gjij_1(7 downto 6); g2(10 downto 9) <= gjij_1(5 downto 4);
g2(6 downto 5) <= gjij_1(3 downto 2); g2(2 downto 1) <= gjij_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, gjk: in STD_LOGIC_VECTOR(15 downto 0);
          pij:       out STD_LOGIC_VECTOR(15 downto 0);
          gjij:      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
    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)&q3(22 downto 15)&q3(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(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 5.51

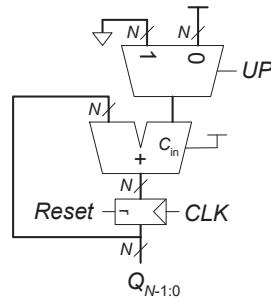
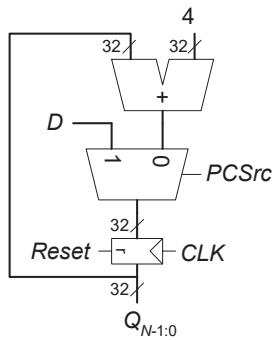


FIGURE 5.7 Up/Down counter

Exercise 5.53

FIGURE 5.8 32-bit counter that increments by 4 or loads a new value, D **Exercise 5.55**

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 5.57

<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 5.9).

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.

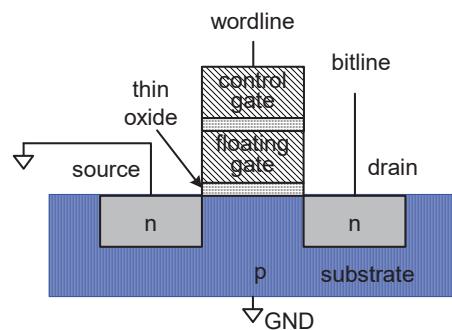
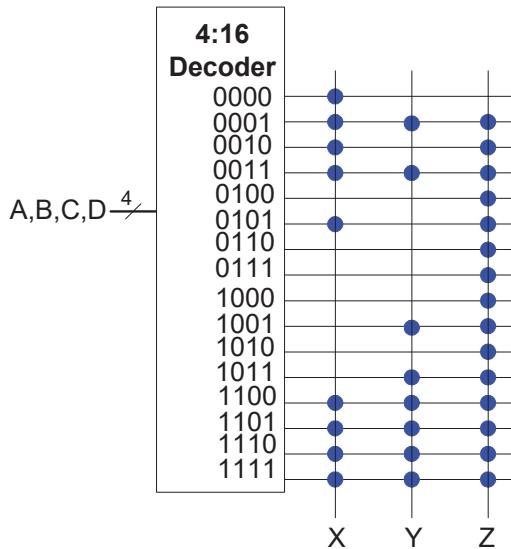


FIGURE 5.9 Flash EEPROM

Exercise 5.59



Exercise 5.61

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

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

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

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

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

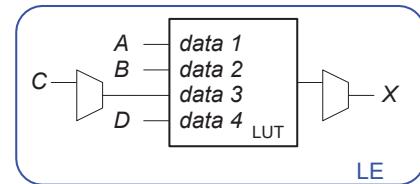
Thus, this would require a $2^{16} \times 4$ -bit ROM.

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

Exercise 5.63

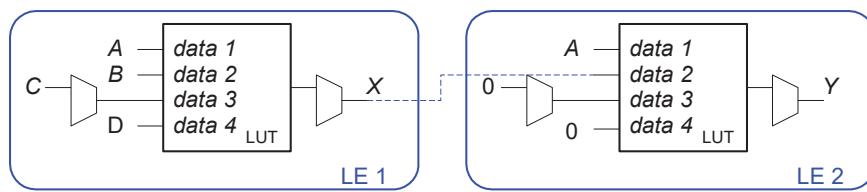
(a) 1 LE

(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



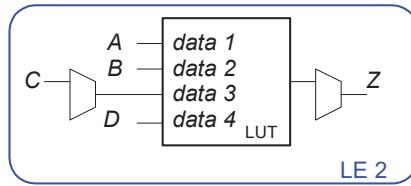
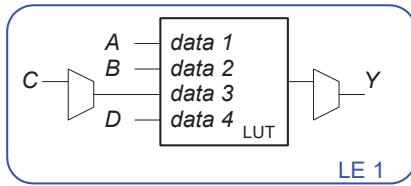
(b) 2 LEs

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



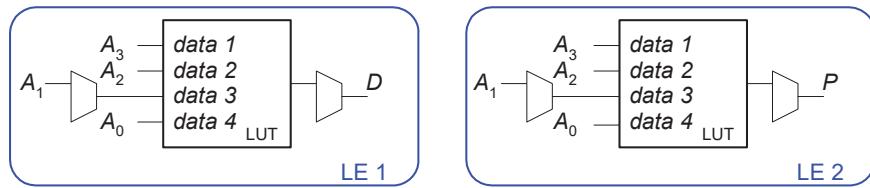
(c) 2 LEs

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



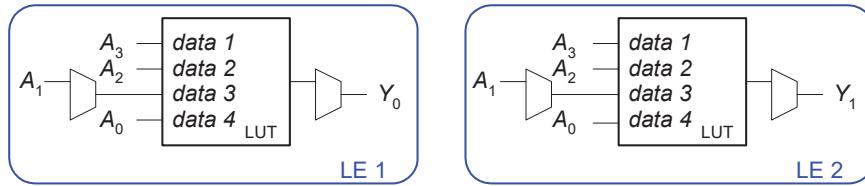
(d) 2 LEs

(A_3)	(A_2)	(A_1)	(A_0)	(D)	LUT output	(A_3)	(A_2)	(A_1)	(A_0)	(P)	LUT output
0	0	0	0		0	0	0	0	0		0
0	0	0	1		0	0	0	0	1		0
0	0	1	0		0	0	0	1	0		1
0	0	1	1		1	0	0	1	1		1
0	1	0	0		0	0	1	0	0		0
0	1	0	1		0	0	1	0	1		1
0	1	1	0		1	0	1	1	0		0
0	1	1	1		0	0	1	1	1		1
1	0	0	0		0	1	0	0	0		0
1	0	0	1		1	1	0	0	1		0
1	0	1	0		0	1	0	1	0		0
1	0	1	1		0	1	0	1	1		1
1	1	0	0		1	1	1	0	0		0
1	1	0	1		0	1	1	0	1		1
1	1	1	0		0	1	1	1	0		0
1	1	1	1		1	1	1	1	0		0
1	1	1	1		1	1	1	1	1		1



(e) 2 LEs

(A_3)	(A_2)	(A_1)	(A_0)	(Y_0)	(A_3)	(A_2)	(A_1)	(A_0)	(Y_1)
data 1	data 2	data 3	data 4	LUT output	data 1	data 2	data 3	data 4	LUT output
0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	1	0
0	0	1	0	1	0	0	1	0	0
0	0	1	1	1	0	0	1	1	0
0	1	0	0	0	0	1	0	0	1
0	1	0	1	0	0	1	0	1	1
0	1	1	0	0	0	1	1	0	1
0	1	1	1	0	0	1	1	1	1
1	0	0	0	1	1	0	0	0	1
1	0	0	1	1	1	0	0	1	1
1	0	1	0	1	1	0	1	0	1
1	0	1	1	1	1	0	1	1	1
1	1	0	0	1	1	1	0	0	1
1	1	0	1	1	1	1	0	1	1
1	1	1	0	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1	1

**Exercise 5.65**

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

(b)

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

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

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

(c)

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

$$t_{cd_LE} = t_{pd_LE} = 381 \text{ ps}$$

$$t_{cd} = t_{cd_LE} + t_{wire} = 627 \text{ ps}$$

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

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

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

$$\begin{aligned} T_c &\geq t_{pcq} + t_{pd} + t_{\text{setup}} + t_{\text{skew}} \\ &\geq [0.902 + 3] \text{ ns} \\ &= 3.902 \text{ ns} \\ f &= 1 / 3.902 \text{ ns} = \mathbf{256 \text{ MHz}} \end{aligned}$$

Exercise 5.67

First, we find the cycle time:

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

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

Thus, $N \leq 15.5$

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

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

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

Question 5.1

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

Question 5.3

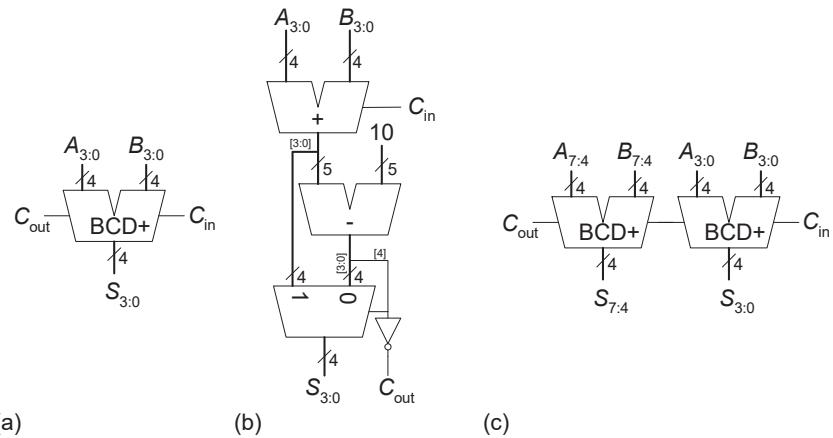


FIGURE 5.10 BCD adder: (a) 4-bit block, (b) underlying hardware, (c) 8-bit BCD adder

(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
  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;

```

CHAPTER 6

Exercise 6.1

(a) **Regularity supports simplicity:**

- Each instruction has a 7-bit opcode in the 7 least significant bits (lsb's) of the instruction which makes the hardware for decoding the instruction simpler.
- RISC-V has four instruction formats (R-, I-, S/B-, and U/J-type) that allows for some regularity among instructions, which then leads to simpler decoder hardware.
- Immediate bit locations are consistent across instruction formats, which minimizes wire routing and the number of multiplexers needed.
- Each instruction is 32 bits, making decoding hardware simpler.

(b) **Make the common case fast:**

- Only simple, commonly used instructions are included, which results in simpler, and thus faster, hardware.
- Registers make the access to recently used variables fast.
- Most instructions require all 32 bits of an instruction, so all instructions are 32 bits (even though some would have an advantage of a larger instruction size and others a smaller instruction size). The instruction size is chosen to make the common instructions fast.

(c) **Smaller is faster:**

- RISC-V includes a small number of commonly used instructions, which keeps the hardware small and fast.
- The instruction size is kept small to allow for fast instruction fetching and simpler decoder logic, which is, thus, faster.
- The register file only has 32 registers, which allows for fast access to them.

(d) **Good design demands good compromises:**

- RISC-V includes four instruction formats, instead of just one, which accommodates different instruction needs while still allowing for some regularity between instruction formats.
- Since RISC-V is a RISC architecture, only simple instructions are supported. However, some more complex pseudoinstructions are provided to the programmer for convenience. For example, some pseudoinstructions, such as `li`, can translate into multiple RISC-V instructions.
- Although memory access is not as fast as register access, it is a required compromise to allow for complex programs.

Exercise 6.3

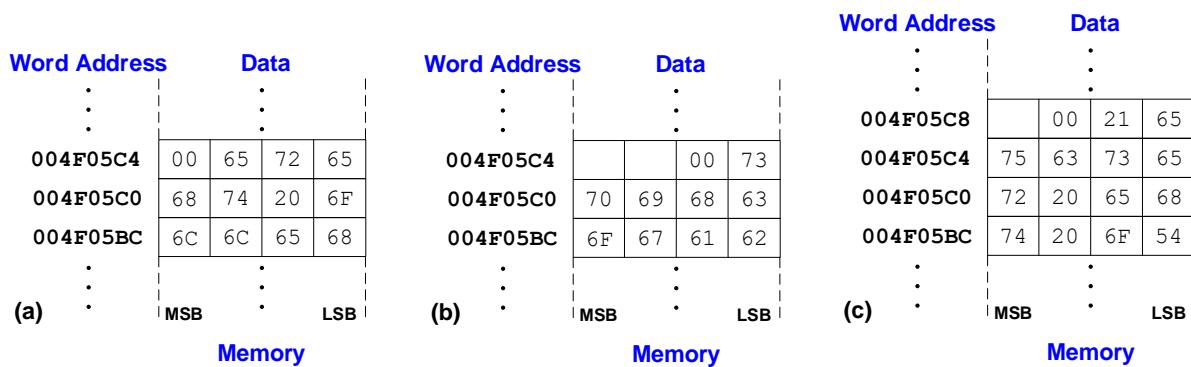
(a)	Character	h	e	l	l	o		t	h	e	r	e	NULL
	ASCII	68	65	6C	6C	6F	20	74	68	65	72	65	00

(b)	Character	b	a	g	o	c	h	i	p	s	NULL
	ASCII	62	61	67	6F	63	68	69	70	73	00

(c)	Character	T	o		t	h	e		r	e	s	c	u	e	!	NULL
	ASCII	54	6F	20	74	68	65	20	72	65	73	63	75	65	21	00

* Recall that in C, the null character (0x00) specifies the end of a string.

Exercise 6.5



Exercise 6.7

```
or    s3, s4, s5
xori s3, s3, -1
```

Exercise 6.9

```
a.)      # a0 = g, a1 = h
        bge a1, a0, else    # do else if (g <= h)
        addi a0, a0, 1       # g = g + 1
        j done                # jump past else block
else: addi a1, a1, -1      # h = h - 1
done:

b.)      # a0 = g, a1 = h
        blt a1, a0, else    # do else if (g > h)
        addi a0, zero, 0     # g = 0
        j done                # jump past else block
else: addi a1, zero, 0      # h = 0
done:
```

Exercise 6.11

```

# t1 = array1 base adr, t2 = array2 base adr
    addi t0, zero, 0          # t0(i) = 0
    addi t3, zero, 100        # t3 = 100
for: bge t0, t3, done        # if i >= 100, array fully copied
    slli t4, t0, 2            # i *= 4
    add t5, t1, t4            # t5 = address of array1[i]
    add t4, t2, t4            # t4 = address of array2[i]
    lw   t6, 0(t4)           # t6 = array2[i]
    sw   t6, 0(t5)           # array1[i] = array2[i]
    addi t0, t0, 1             # i += 1
    j    for                  # loop
done:                         # end of code snippet

```

Exercise 6.13

-
- (a) addi s7, zero, 29 # s7 = 29
 - (b) addi s7, zero, -214 # s7 = -214
 - (c) lui s7, 0xFFFFFFF # s7 = 0xFFFFFFF000
 - addi s7, s7, 0x449 # s7 = 0xFFFFF449 = -2999
 - (d) lui s7, 0xABCD E # s7 = 0xABCD E000
 - (e) lui s7, 0xEDCBA # s7 = 0xEDCBA000
 - addi s7, s7, 0x123 # s7 = 0xEDCBA123
 - (f) lui s7, 0xEEEEF # s7 = 0xEEEEF000
 - addi s7, s7, -85 # s7 = 0xEEEEFAB (-85 = 0xFAB)

Exercise 6.15

```

int find42(int array[], int size) {
    for (int i = 0; i < size; i++) {
        if (array[i] == 42)
            return i;
    }
    return -1;
}

```

Exercise 6.17

```

find42:
    addi t0, zero, 42      # t0 = 42
    addi t1, zero, 0        # i = 0
loop:
    bge t1, a1, notFound # loop if i < size (if not, end reached
                          # and 42 not found)
    slli t2, t1, 2          # i * 4 (to find byte offset)
    add t2, a0, t2          # t2 = address of array[i]
    lw   t2, 0(t2)          # t2 = array[i]

```

```

beq t2, t0, found      # if array[i] == 42, go to found
addi t1, t1, 1          # i++ (i = i + 1)
j loop                  # next iteration
found:
    add a0, zero, t1      # a0 = i
    jr ra                  # return
notFound:
    addi a0, zero, -1     # a0 = -1
    jr ra                  # return

```

Exercise 6.19(a) $fib(0) = 0$, $fib(-1) = 1$ (b) **High Level Code**

```

int fib(int n) {
    int i;
    int current = 0;           // fib(i) - initialized to fib(0)
    int prev = 1;              // fib(i-1) - initialized to fib(-1)

    for (i = 1; i <= n; i++) {
        current = current + prev; // fib(i) = fib(i-1) + fib(i-2)
        prev = current - prev;   // update prev:
                                // fib(i-1) = fib(i) - fib(i-2)
    }
    return current;            // return fib(n)
}

```

(c) **RISC-V Assembly Code**

```

addi a0, zero, 9          # n = 9
jal fib                  # call fib(n), where n = 9
...                      # code after function call

fib:
    addi sp, sp, -12       # make room on stack for 3 registers
    sw s0, 8(sp)           # save s0 on stack
    sw s1, 4(sp)           # save s1 on stack
    sw s2, 0(sp)           # save s2 on stack
    addi s0, zero, 0         # current = 0 (fib(i))
    addi s1, zero, 1         # prev = 1 (fib(i-1))
    addi s2, zero, 1         # i = 1
for:
    blt a0, s2, result     # if i > n then loop ends
    add s0, s0, s1           # fib(i) = fib(i - 1) + fib(i - 2)
    sub s1, s0, s1           # fib(i - 1) = fib(i) - fib(i - 2)
    addi s2, s2, 1             # i = i + 1
    j for                   # repeat loop
result:
    add a0, zero, s0         # return fib(n) (put fib(n) in a0)
    lw s0, 8(sp)             # restore registers from stack
    lw s1, 4(sp)
    lw s2, 0(sp)
    addi sp, sp, 12           # restore stack pointer
    jr ra                    # return

```

Exercise 6.21

- (a) By the time the program reaches the loop label, register $a0$ will hold the value **19**, which is $(5+5) + (3+3+3) = 2a + 3b = 19$, as intended.
- (b) 3 – The program will produce an incorrect value in register $a0$. The store word instruction (`sw a0, 0xC(sp)`) puts the original value of $a0$ ($a = 5$) on the stack. When the instruction at `0x8030` tries to load that value, $t0 =$ an unknown value (i.e., whatever was in the stack memory location before). The value in $a0$ can't be determined, but it will be: $5 + t0 + 9 = 14 + t0$.
- (c)
- (i) 3 – The program will produce an incorrect value in register $a0$. The same explanation as part (b) applies. You can't determine what is in $a0$ because you would need to know the previous value of $t0$.
 - (ii) 2 – The program would crash due to the stack growing beyond the dynamic data segment. This is due to instruction `0x8040` being removed. ra is no longer restored from the stack and retains its current value, which is `0x8030`. It now repeatedly executes the instructions from `0x8030` to `0x8048`. Instruction `0x8044` increments the stack pointer (sp) so this occurs until the stack pointer increases beyond the dynamic data segment.
 - (iii) 4 – The program would run correctly despite the deleted lines. However, the value of register $s4$ before the call to f would not be restored. This doesn't affect the return value of $f(5, 3)$, but if the caller (in this case, `test`) needs to use $s4$ after the function call to f , the program retrieves an incorrect value.
 - (iv) 3 – The program will produce an incorrect value in register $a0$. The same explanation as in part (b) applies.
 - (v) 3 – The program will produce an incorrect value in register $a0$. $s4 = 5$ wouldn't be saved to the stack during the first iteration of g . Because $s4$ becomes 3 in the g function, the $3b$ part of $2a + 3b$ would still be carried out correctly. During the last iteration of g , $s4$ would remain 3 so during the add instruction at `0x8038` the operation would be $(a + b)$ instead of $(a + a)$ as intended. The final result in register $a0$ would be **17** (i.e., $5 + 3 + 3*3 = 17$).
 - (vi) 4 – The program would run correctly despite the deleted lines. The explanation is the same as in part (iii) above.
 - (vii) 2 – The program would crash due to the stack growing beyond the dynamic data segment. This is similar to the scenario for ii). The return address of `0x8030` would never be stored to register ra . After the first iteration of g , $ra = 0x8070$. With the instruction at `0x8080` (`j r ra`), instructions `0x8070 – 0x8080` would loop infinitely. Because the instruction at address `0x807C` increments the stack pointer, sp continues to increase until it grows beyond the dynamic data segment. Then it would crash.

Exercise 6.23

Instruction	Machine Code
add s7, s8, s9	0x019C0BB3
srai t0, t1, 0xC	0x40C35293
ori s3, s1, -1348	0xABC4E993
lw s4, 0x5C(t3)	0x05CE2A03

Assembly	Field Values	Machine Code
add s7, s8, s9 add x23, x24, x25	funct7: 0, rs2: 25, rs1: 24, funct3: 0, rd: 23, op: 51	funct7: 0000_000, rs2: 11001, rs1: 11000, funct3: 000, rd: 10111, op: 011_0011 (0x019C0BB3)
srai t0, t1, 0xC srai x5, x6, 0xC	imm11: 0xC (and 1 in bit 30), rs1: 6, funct3: 5, rd: 5, op: 19	imm11: 0100 0000, rs1: 00110, funct3: 101, rd: 00101, op: 001 0011 (0x40C35293)
ori s3, s1, -1348 ori x19, x9, -1348	imm11: -1348, rs1: 9, funct3: 6, rd: 19, op: 19	imm11: 1010 1011 1100, rs1: 01001, funct3: 110, rd: 10011, op: 001 0011 (0xABC4E993)
lw s4, 0x5C(t3) lw x20, 0x5C(x28)	imm11: 0x5C, rs1: 28, funct3: 2, rd: 20, op: 3	imm11: 0000 0101 1100, rs1: 11100, funct3: 010, rd: 10100, op: 000 0011 (0x05CE2A03)

Exercise 6.25

(a) Instruction	(b) Type
srai t0, t1, 0xC	I
ori s3, s1, -1348	I
lw s4, 0x5C(t3)	I

(c)

srai - 0xC: 5-bit: 0 1100 – 0x0C (5-bit) – not extended

ori - 1348: 12-bit: 1010 1011 1100 = 0xAB

1111 1111 1111 1111 1010 1011 1100 = **0xFFFFFABC** (32-bit)

lw - 0x5C: 12-bit: 0000 0101 1100 = 0x05C

0000 0000 0000 0000 0000 0101 1100 = **0x0000005C** (32-bit)

Exercise 6.27

(a)	0x01F00393	addi t2,	zero, 31	# t2 = 31
	0x00755E33	L1: srl t3,	a0, t2	# t3 = a0 >> t2
	0x001E7E13	andi t3,	t3, 1	# t3 = lsb of t3
	0x01C580A3	sb t3,	1(a1)	# a1[1] = t3
	0x00158593	addi a1,	a1, 1	# a1++

```

0xFFFF38393      addi t2,    t2,   -1      # t2--
0xFE03D6E3      bge  t2,    zero,  L1      # if t2 >= 0, repeat
0x00008067      jr   ra                  # return

```

(b) # a0 = val, a1 = base address of array
t2 = shiftAmt, t3 = tmp
void decToBin(int val, char array[]){
 int shiftAmt = 31;
 char tmp;
 int i = 1;

 do {
 tmp = (val >> shiftAmt);
 tmp = tmp & 1;
 array[i] = tmp;
 i++;
 shiftAmt--;
 } while (shiftAmt >= 0);
}

(c) This program takes in a 32-bit number, a0, and converts it from decimal to binary. The result is stored as characters in an array that is pointed to by a1 from index 1 to 32, i.e., from array[1] to array[32].

Exercise 6.29

B-Type

31:25	24:20	19:15	14:12	11:7	6:0
imm _{12,10:5}	rs2	rs1	funct3	imm _{4:1,11}	op

7 bits 5 bits 5 bits 3 bits 5 bits 7 bits

a) **blt t4, s3, Loop:** 0xAA00E130 – 0xAA00E124 = 0xC: branch 0xC bytes forward.

Branch offset = imm_{12:0} = **0 000 0000 1100**

rs1 = t4 = x29 (11101b), rs2 = s3 = x19 (10011b)

funct3 = 100 (blt), op = 1100011 (branch)

imm _{12,10:5}	rs2	rs1	funct3	imm _{4:1,11}	op
0000000	10011	11101	100	01100	1100011
= 0x013EC663					

b) **bge t1, t2, L1:** 0xC090174C – 0xC0901000 = 0x74C: branch 0x74C bytes forward.

Branch offset = imm_{12:0} = **0 0111 0100 1100**

rs1 = t1 = x6 (00110b), rs2 = t2 = x7 (00111b)

funct3 = 101 (bge), op = 1100011 (branch)

imm _{12,10:5}	rs2	rs1	funct3	imm _{4:1,11}	op
0111010	00111	00110	101	01100	1100011

= **0x74735663**

- c) **bne s10, s11, Back:** $0x1230D908 - 0x1230D10C = 0x7FC$: branch $0x7FC$ bytes backward.

Flip sign of: $0x7FC$ ($0\ 0111\ 1111\ 1100$): $1\ 1000\ 0000\ 0011 + 1 = 1\ 1000\ 0000\ 0100$
 Branch offset = $\text{imm}_{12:0} = 0x1804$: **1 1000 0000 0100**

$rs1 = s10 = x26$ (11010b), $rs2 = s11 = x27$ (11011b)
 $funct3 = 001$ (bne), op = 1100011 (branch)

imm_{12,10:5}	rs2	rs1	funct3	imm_{4:1,11}	op
1000000	11011	11010	001	00101	1100011
= 0x81BD12E3					

- d) **beq a0, s1, L2:** $0xAB0CA0FC - 0xAB0C99A8 = 0x754$: branch $0x754$ bytes forward.

Branch offset = $\text{imm}_{12:0} = 0x754$: **0 0111 0101 0100**
 $rs1 = a0 = x10$ (01010b), $rs2 = s1 = x9$ (01001b)
 $funct3 = 000$ (beq), op = 1100011 (branch)

imm_{12,10:5}	rs2	rs1	funct3	imm_{4:1,11}	op
0111010	01001	01010	000	10100	1100011
= 0x74950A63					

- e) **blt s1, t3, L3:** $0xFFABD640 - 0xFFABC04 = 0x73C$: branch $0x73C$ bytes backward.

Flip sign of: $0x73C$ ($0\ 0111\ 0011\ 1100$): $1\ 1000\ 1100\ 0011 + 1 = 1\ 1000\ 1100\ 0100$
 Branch offset = $\text{imm}_{12:0} = 0x18C4$: **1 1000 1100 0100**
 $rs1 = s1 = x9$ (01001b), $rs2 = t3 = x28$ (11100b)
 $funct3 = 100$ (blt), op = 1100011 (branch)

imm_{12,10:5}	rs2	rs1	funct3	imm_{4:1,11}	op
1000110	11100	01001	100	00101	1100011
= 0x8DC4C2E3					

Exercise 6.31**J-Type**

$\text{imm}_{20,10:1,11,19:12}$	rd	op
20 bits	5 bits	7 bits

- a) $0x0000EEEC - 0x0000ABC0 = 0x432C$: jump $0x432C$ bytes forward.

Jump offset = $\text{imm}_{20:0} = 0x432C = \textcolor{blue}{0} \textcolor{red}{0000} \textcolor{green}{0100} \textcolor{red}{0011} \textcolor{green}{0010} \textcolor{red}{1100}$

$\text{rd} = \text{ra} = x1$ (00001b), $\text{op} = 1101111$ (J-type)

$\text{imm}_{20,10:1,11,19:12}$	rd	op
$\textcolor{blue}{0} \textcolor{red}{011} \textcolor{green}{0010} \textcolor{red}{1100} \textcolor{blue}{0} \textcolor{red}{0000} \textcolor{green}{0100}$	00001	1101111
$= \boxed{0x32C040EF}$		

- b) $0x000F1230 - 0x0000C10C = 0xE5124$: jump $0xE5124$ bytes backward.

Flip sign of: $0xE5124$: 0 1110 0101 0001 0010 0100:

$$\begin{array}{l} 1\ 0001\ 1010\ 1110\ 1101\ 1011 + 1 = \\ 1\ 0001\ 1010\ 1110\ 1101\ 1100 \end{array}$$

Jump offset = $\text{imm}_{20:0} = 0x11AEDC = \textcolor{blue}{1} \textcolor{red}{0001} \textcolor{black}{1010} \textcolor{red}{1110} \textcolor{green}{1101} \textcolor{red}{1100}$

$\text{rd} = \text{ra} = x1$ (00001b), $\text{op} = 1101111$ (J-type)

$\text{imm}_{20,10:1,11,19:12}$	rd	op
$\textcolor{blue}{1} \textcolor{red}{110} \textcolor{green}{1101} \textcolor{red}{1100} \textcolor{blue}{1} \textcolor{red}{0001} \textcolor{black}{1010}$	00001	1101111
$= \boxed{0xEDD1A0EF}$		

- c) $0x008FFFDC - 0x00801000 = 0xFEFDC$: jump $0xFEFDC$ bytes forward.

Jump offset = $\text{imm}_{20:0} = 0xFEFDC = \textcolor{blue}{0} \textcolor{red}{1111} \textcolor{black}{1110} \textcolor{red}{1110} \textcolor{green}{1101} \textcolor{red}{1101} \textcolor{green}{1100}$

$\text{rd} = \text{ra} = x1$ (00001b), $\text{op} = 1101111$ (J-type)

$\text{imm}_{20,10:1,11,19:12}$	rd	op
$\textcolor{blue}{0} \textcolor{red}{110} \textcolor{green}{1101} \textcolor{red}{1100} \textcolor{blue}{1} \textcolor{red}{1111} \textcolor{black}{1110}$	00001	1101111
$= \boxed{0x6DDFE0EF}$		

- d) $0xA131347C - 0xA1234560 = 0xDEF1C$: jump $0xDEF1C$ bytes forward.

Jump offset = $\text{imm}_{20:0} = 0xDEF1C = \textcolor{blue}{0} \textcolor{red}{1101} \textcolor{black}{1110} \textcolor{red}{1111} \textcolor{green}{0001} \textcolor{red}{1100}$

$\text{rd} = \text{ra} = x0$ (00000b), $\text{op} = 1101111$ (J-type)

$\text{imm}_{20,10:1,11,19:12}$	rd	op
$\textcolor{blue}{0} \textcolor{red}{111} \textcolor{green}{0001} \textcolor{red}{1100} \textcolor{blue}{1} \textcolor{red}{1101} \textcolor{black}{1110}$	00000	1101111
$= \boxed{0x71DDE06F}$		

- e) $0xF0CBCCD4 - 0xF0BBCCD4 = 0x100000$: branch $0x100000$ bytes backward.

Flip sign of: $0x100000$: 1 0000 0000 0000 0000 0000:

$$0\ 1111\ 1111\ 1111\ 1111\ 1111 + 1 =$$

1 0000 0000 0000 0000 0000

Jump offset = $\text{imm}_{20:0} = 0x100000 = \textcolor{blue}{1} \textcolor{green}{0000} \textcolor{green}{0000} \textcolor{red}{0} \textcolor{green}{0000} \textcolor{green}{0000} \textcolor{green}{0000} \theta$
rd = x0 (00000b), op = 1101111 (J-type)

imm_{20, 10:1, 11, 19:12}	rd	op
1 000 0000 000 0 0000 0000	00000	1101111
= 0x8000006F		

Exercise 6.33

(a)

```

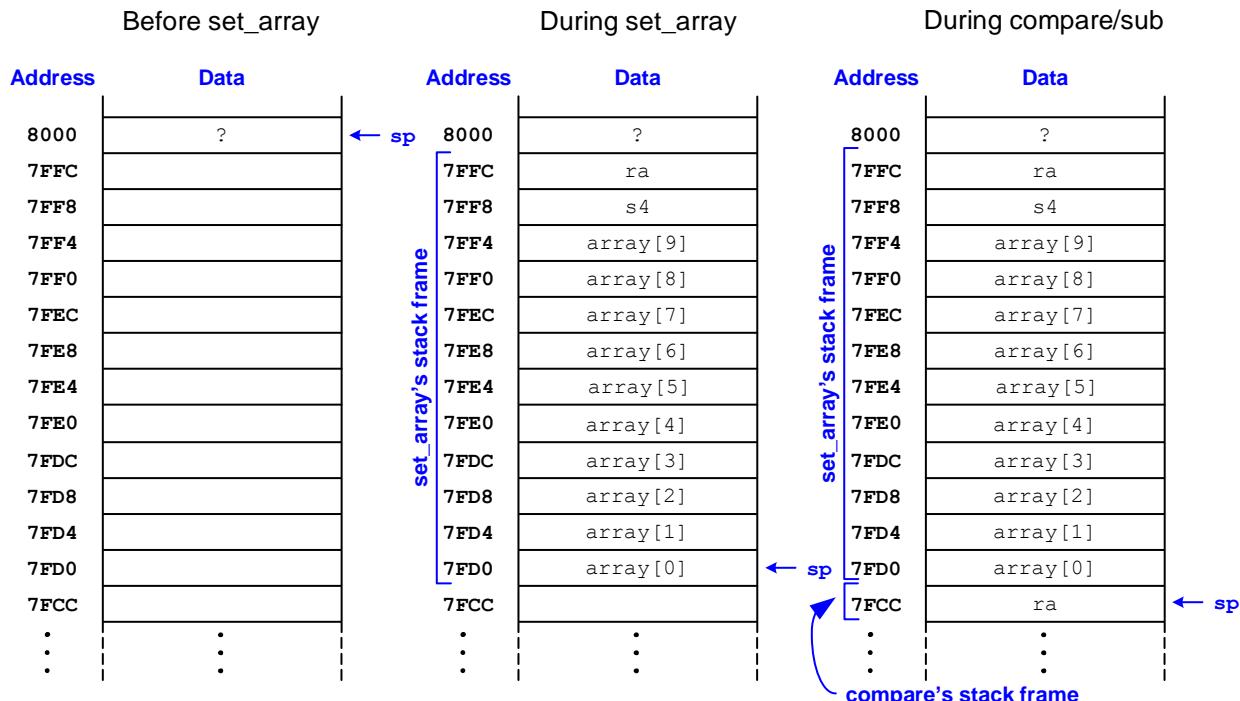
set_array: # a0 = num, s4 = i
    addi sp, sp, -48      # create space on the stack
    sw ra, 44(sp)        # store ra on the stack
    sw s4, 40(sp)        # store s4 on the stack
    addi s4, zero, 0      # i = 0
    addi t0, zero, 10     # t0 = 10 (# iterations)
loop:
    bge s4, t0, done      # if i >= 10, exit loop
    add a1, s4, zero       # a1 = i (second parameter)
    jal compare            # compare(num, i)
    slli t1, s4, 2          # t1 = i*4
    add t2, sp, t1          # t2 = address of array[i]
    sw a0, 0(t2)           # array[i] = compare(num, i)
    addi s4, s4, 1          # i++
    j loop                  # repeat loop
done:
    lw ra, 44(sp)          # restore ra
    lw s4, 40(sp)          # restore s4
    addi sp, sp, 48         # restore stack pointer
    jr ra                   # return

compare: # a0 = a, a1 = b
    addi sp, sp, -4        # create space in the stack
    sw ra, 0(sp)           # save ra on the stack
    jal sub                 # call sub(a, b)
    slt a0, a0, zero        # a0 = 1 if sub(a,b) < 0
    xor a0, a0, -1          # invert a0
    andi a0, a0, 1           # isolate a0 and return
    lw ra, 0(sp)           # restore ra
    addi sp, sp, 4           # restore sp
    jr ra                   # return

sub: # a0 = a, a1 = b
    sub a0, a0, a1          # a0 = a - b
    jr ra                   # return

```

(b)



(c) If `ra` were not saved on the stack, when the `compare` function attempts to return to `set_array` it would instead return to the instruction following the `sub` function call (`slt a0, a0, zero`) and would continue returning there every time it attempts to return. Since it will continue incrementing the stack pointer, it will eventually crash due to exceeding the stack space.

Exercise 6.35

Regardless of the situation, since all branches encode the *offset* in their immediate field, they can always jump forward 1,023 instructions. Specifically, a 13-bit signed (i.e., two's complement) number can encode an offset of up to: $2^{12} - 1$ bytes / (4 bytes/instruction) = 2^{10} instructions – 1/4. But we can't have 1/4 of an instruction, so it can encode a forward branch offset of up to 2^{10} instructions – 1 instructions = 1,023 instructions.

Exercise 6.37

```
0x8000    lui t0, 0x408    # t0 = 0x40 8000
0x8004    jr t0           # PC = 0x40 8000
```

Explanation: 2^{20} instructions = 2^{22} bytes. This is address: 0100 0000 0000 0000 0000 0000 (i.e., 0x40 0000) bytes beyond the current address (0x8000), so the code should branch to 0x40 8000.

Exercise 6.39

We show two options:

Option 1:

(a) High Level Code

```
void swapEndianness(int arr[]) {
    for(int i = 0; i < 10; i++) {
        arr[i] = ((arr[i] << 24) |
                   ((arr[i] & 0xFF00) << 8) |
                   ((arr[i] & 0xFF0000) >> 8) |
                   ((arr[i] >> 24) & 0xFF));
    }
}
```

(b) RISC-V Assembly Code

```
# a0 = base address of arr, t0 = i
swapEndianness:
    addi  t0, zero, 0      # i = 0
    addi  t1, zero, 10     # t4 = 10 (temp value)
    lui   t4, 0xFF0          # t4 = 0xFF0000
    srlt  t3, t4, 8         # t3 = 0xFF00
L1:
    bge   t0, t1, done      # if i >= 10 return
    slli  t5, t0, 2          # t5 = i * 4 (byte offset)
    add   t5, t5, a0         # t5 = address of arr[i]
    lw    t6, 0(t5)          # t6 = arr[i]
    slli  a1, t6, 24         # a1 = arr[i] << 24
    and   a2, t6, t3         # a2 = arr[i] & 0xFF00
    slli  a2, a2, 8           # a2 = (arr[i] & 0xFF00) << 8
    and   a3, t6, t4         # a3 = arr[i] & 0xFF0000
    srlt  a3, a3, 8           # a3 = (arr[i] & 0xFF0000) >> 8
    srlt  a4, t6, 24         # a4 = arr[i] >> 24
    or    a0, a1, a2         # a0 = combine most significant bytes
    or    a0, a0, a3         # a0 = combine 3 most significant bytes
    or    a0, a0, a4         # a0 = combine all bytes
    sw    a0, 0(t5)          # arr[i] = value with other endianness
    addi t0, t0, 1            # i++
    j     L1                  # loop
done:
```

Option 2:

(a) High Level Code

```
void swapEndianness(int arr[]) {
    char *arrBytes = (char *)arr;
    char tmp0, tmp1, tmp2, tmp3;
    int i = 10;

    do {
        tmp0 = arrBytes[0];
        tmp1 = arrBytes[1];
        tmp2 = arrBytes[2];
```

```

    tmp3 = arrBytes[3];
    arrBytes[0] = tmp3;
    arrBytes[1] = tmp2;
    arrBytes[2] = tmp1;
    arrBytes[3] = tmp0;
    arrBytes += 4;
    i--;
} while (i != 0);
return;
}

```

(b) RISC-V Assembly Code

```

# a0 = base address of array
swapEndianness:
    addi t4, zero, 10    # i = t4 = 10 (loop counter)
loop:
    lb    t0, 0(a0)      # t0 = byte 0 of arr[i]
    lb    t1, 1(a0)      # t1 = byte 1 of arr[i]
    lb    t2, 2(a0)      # t2 = byte 2 of arr[i]
    lb    t3, 3(a0)      # t3 = byte 3 of arr[i]
    sb    t3, 0(a0)      # byte 0 of arr[i] = original byte 3
    sb    t2, 1(a0)      # byte 1 of arr[i] = original byte 2
    sb    t1, 2(a0)      # byte 2 of arr[i] = original byte 1
    sb    t0, 3(a0)      # byte 3 of arr[i] = original byte 0
    addi a0, a0, 4       # increment array index
    addi t4, t4, -1      # i--
    beq t4, zero, done   # exit loop if i == 0
    j     loop
done:
    jr    ra

```

Exercise 6.41

```

# a0 = first value, a1 = second value
addFloat:
extract:
    lui   t4, 0x800
    addi t4, t4, -1      # t4 = 0x007FFFFF (mantissa mask)
    and   t0, a0, t4      # t0 = a0 mantissa
    and   t1, a1, t4      # t1 = a1 mantissa
    lui   t4, 0x800
    or    t0, t0, t4      # add implicit 1 to a0 mantissa
    or    t1, t1, t4      # add implicit 1 to a1 mantissa
    lui   t4, 0x7F800000
    and   t2, a0, t4      # t2 = a0 exponent
    srlt t2, t2, 23       # shift a0 exponent right
    and   t3, a1, t4      # t3 = a1 exponent
    srli t3, t3, 23       # shift a1 exponent right
compare:
    beq t2, t3, addMant # check if exponents match

```

```

        bgeu t2, t3, shift1 # check which exponent is larger
shift0:
    sub t4, t3, t2      # calculate the difference of exponents
    sra t0, t0, t4      # shift a0 by above difference
    add t2, t2, t4      # update a0's exponent
    j addMant           # next we add the mantissas
shift1:
    sub t4, t2, t3      # calculate the difference of exponents
    sra t1, t1, t4      # shift a1 by above difference
    add t3, t3, t4      # update a1's exponent (for regularity)
addMant:
    add t5, t0, t1      # add the mantissas
norm:
    lui t4, 0x1000      # t4 = 0x01000000 (overflow bit mask)
    and t4, t5, t4      # t4 = masked bit 24
    beq t4, zero, done  # no need to right shift if no overflow
    srli t5, t5, 1      # shift mantissa by right by one
    addi t2, t2, 1      # increment the exponent
done:
    lui t4, 0x800      # t4 = 0x007FFFFF (mantissa mask)
    addi t4, t4, -1     # t4 = masked result mantissa
    and t4, t5, t4      # align the exponent in proper place
    slli t2, t2, 23     # t4 = 0x7F800000 (exponent mask)
    lui t4, 0x7F800     # t2 = result exponent
    and t2, t2, t4      # result stored in a0
    or a0, t5, t2       # result stored in a0
    jr ra               # return

```

Exercise 6.43

(a) High Level Code

```

// sorts a 10-element array using Bubble Sort
void sort(int scores[]){
    for (int i = 0; i < 9; i++){
        for (int j = 0; j < 9-i; j++){
            // swap places if next element is larger
            if(scores[j] > scores[j+1]){
                scores[j] = scores[j]^scores[j+1];
                scores[j+1] = scores[j]^scores[j+1];
                scores[j] = scores[j]^scores[j+1];
            }
        }
    }
}

```

(b) RISC-V Assembly Code

```

# a0 = address of scores array
# we assume 16-bit (4-byte) integer size
sort:
    addi t0, zero, 0    # i = 0

```

```

    addi t1, zero, 9      # t1 = 9
outerLoop:
    bge t0, t1, done1 # i >= 9?
    addi t2, zero, 0      # j = 0
    sub t3, t1, t0      # t3 = 9-i
innerLoop:
    bge t2, t3, done2 # j >= 9-i?
    slli t4, t2, 1      # t4 = scores array offset (j*2)
    add t4, t4, a0      # t4 = address of scores[j]
    lh   t5, 0(t4)      # t5 = scores[j]
    lh   t6, 2(t4)      # t6 = scores[j+1]
    bge t6, t5, skip   # skip if scores[j+1] >= scores[j]
    sb   t5, 2(t4)      # scores[j] = original scores[j+1]
    sb   t6, 0(t4)      # scores[j+1] = original scores[j]
skip:
    addi t2, t2, 1      # j++
    j     innerLoop      # repeat innerLoop
done2:
    addi t0, t0, 1      # i++
    j     outerLoop      # repeat outerLoop
done1:
    jr   ra                # return

```

Exercise 6.45

(a) **0x8534** main: addi sp, sp, -8
0x8538 sw ra, 4(sp)
0x853C sw s4, 0(sp)
0x8540 addi s4, zero, 15
0x8544 sw s4, -300(gp) # g = 15
0x8548 addi a1, zero, 27 # arg1 = 27
0x854C sw a1, -296(gp) # h = 27
0x8550 lw a0, -300(gp) # arg0 = g = 15
0x8554 jal greater
0x8558 lw s4, 0(sp)
0x855C lw ra, 4(sp)
0x8560 addi sp, sp, 8
0x8564 jr ra
0x8568 greater: blt a1, a0, isGreater
0x856C addi a0, zero, 0
0x8570 jr ra
0x8574 isGreater: addi a0, zero, 1
0x8578 jr ra

(b) Symbol Table:

Address	Size	Symbol
0x8534	00000034	main
0x8568	0000000C	greater
0x8574	00000008	isGreater
0x1305C	00000004	g

0x13060 00000004 h

(d) **0x8534** main: addi sp, sp, -8 # 0xFF810113
0x8538 sw ra, 4(sp) # 0x00112223
0x853C sw s4, 0(sp) # 0x01412023
0x8540 addi s4, zero, 15 # 0x00F00A13
0x8544 sw s4, -300(gp) # 0xED41AA23
0x8548 addi a1, zero, 27 # 0x01B00593
0x854C sw a1, -296(gp) # 0xECB1AC23
0x8550 lw a0, -300(gp) # 0xED41A503
0x8554 jal greater # 0x014000EF
0x8558 lw s4, 0(sp) # 0x00012A03
0x855C lw ra, 4(sp) # 0x00412083
0x8560 addi sp, sp, 8 # 0x00810113
0x8564 jr ra # 0x00008067
0x8568 greater: blt a1, a0, isGreater # 0x00A5C663
0x856C addi a0, zero, 0 # 0x00000513
0x8570 jr ra # 0x00008067
0x8574 isGreater: addi a0, zero, 1 # 0x00100513
0x8578 jr ra # 0x00008067

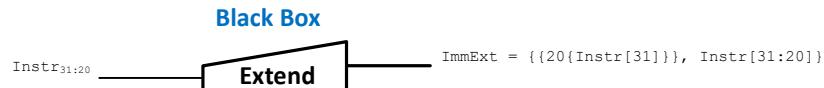
(d) The global data segment is 8 bytes and the text segment is 72 bytes.

(e)

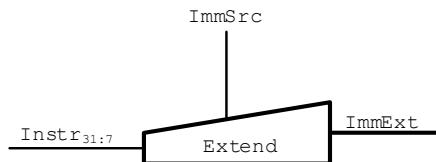
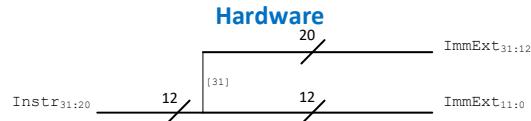


Exercise 6.47

(a)

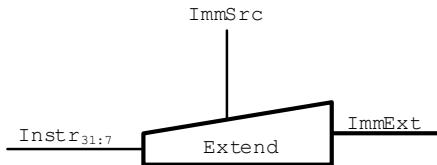


(b)



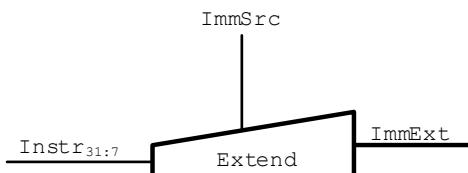
ImmSrc	ImmExt	Type
0	$\{\{20\{\text{Instr}[31]\}\}, \text{Instr}[31:20]\}$	I
1	$\{\{20\{\text{Instr}[31]\}\}, \text{Instr}[31:25], \text{Instr}[11:7]\}$	S

c)



ImmSrc	ImmExt	Type
00	$\{\{20\{\text{Instr}[31]\}\}, \text{Instr}[31:20]\}$	I
01	$\{\{20\{\text{Instr}[31]\}\}, \text{Instr}[31:25], \text{Instr}[11:7]\}$	S
11	$\{\{20\{\text{Instr}[31]\}\}, \text{Instr}[7], \text{Instr}[30:25], \text{Instr}[11:8], 1'b0\}$	B

d)



ImmSrc	ImmExt	Type
00	$\{\{20\{\text{Instr}[31]\}\}, \text{Instr}[31:20]\}$	I
01	$\{\{20\{\text{Instr}[31]\}\}, \text{Instr}[31:25], \text{Instr}[11:7]\}$	S
10	$\{\{20\{\text{Instr}[31]\}\}, \text{Instr}[7], \text{Instr}[30:25], \text{Instr}[11:8], 1'b0\}$	B
11	$\{\{12\{\text{Instr}[31]\}\}, \text{Instr}[19:12], \text{Instr}[20], \text{Instr}[30:21], 1'b0\}$	J

The figure below shows how the instruction bits are used to recreate the immediate for all instruction formats that encode an immediate that is extended to 32 bits. To create the 32-bit immediate, most of the immediate bits require only a 2:1 mux, thereby choosing amongst only two possible instruction bit locations. This is in contrast to the worst case, where each immediate bit would require a 5:1 mux, to select amongst five instruction bit locations for each of the five instruction formats.

instruction bit	wire	2:1 mux	2:1 mux	4:1 mux	2:1 mux	3:1 mux	3:1 mux	I
31	31	31	31	31	30:25	24:21	20	S
31	31	31	31	31	30:25	11:8	7	B
31	31	31	31	7	30:25	11:8	0	U
31	30:20	19:12	19:12	0	0	0	0	J
31	31	19:12	20	30:25	24:21	20		
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	immediate bit							

Exercise 6.49

- (a) `jal` can jump forward $(2^{(21-1)} - 1)/4 = 2^{18} - 1 = 262,143$ instructions forward.
(b) `jal` can jump backward $-[2^{(21-1)}]/4 = -2^{18} = 262,144$ instructions backward.

Exercise 6.51

- a) $15 \times 4 = 15 \times 2^2 = 1111_2 \ll 2 = 111100_2 = 0x3C$
b) 0x3C through 0x3F
c)



Question 6.1

```
xor a0, a0, a1 # a0 = a0 XOR a1
xor a1, a0, a1 # a1 = original a0
xor a0, a0, a1 # a0 = original a1
```

Example:

$a0 = 1011\ 0101$ (binary)
 $a1 = 0010\ 1111$ (binary)

```
xor a0, a0, a1 # a0 = 1001\ 1010 (1's wherever different)
xor a1, a0, a1 # a1 = original a0 = 1011\ 0101
xor a0, a0, a1 # a0 = original a1 = 0010\ 1111
```

Question 6.3

High-Level Algorithm

```

void reverseWords(char arr[]) {
    // find the length of the string
    int length;
    for (length=0; arr[length] != 0; length++);

    // first reverse the entire string
    reverse(arr, 0, length-1);

    // next reverse each individual word back
    int begin = 0;
    int end = 0;

    // find start and end positions of each word
    while (end <= length) {
        if ((end != length) && (arr[end] != 0x20))
            end++;
        else {
            reverse(arr, begin, end-1);
            end++;
            begin = end;
        }
    }
}

// This function reverses the characters of the passed array
// between the passed begin and end index positions
void reverse(char arr[], int begin, int end) {
    while (begin < end) {
        // swap characters
        arr[begin] = arr[begin]^arr[end];
        arr[end] = arr[begin]^arr[end];
        arr[begin] = arr[begin]^arr[end];
        // move index positions in
        begin++;
        end--;
    }
}

```

RISC-V Assembly Code

```

# a0 = address of arr[], a1 = begin, a2 = end
reverseWords:
    addi t0, zero, 0      # t0 = length = 0
for:
    add  t1, a0, t0       # t1 = address of arr[length]
    lb   t1, 0(t1)        # t1 = arr[length]
    beq  t1, zero, done   # stop counting when we hit null
    addi t0, t0, 1         # length++
    j    for               # repeat loop

```

```

done:
    addi a1, zero, 0      # begin = 0
    addi a2, t0, -1       # end = length - 1
    jal reverse           # reverse(arr, 0, length-1)
    addi a1, zero, 0      # a1 = begin
    addi a2, zero, 0      # a2 = end
while:
    blt t0, a1, done2    # length < begin?
    beq a2, t0, else     # end == length?
    addi t1, zero, 0x20   # t1 = 0x20
    add t2, a0, a2        # t2 = address of arr[end]
    lb t2, 0(t2)          # t2 = arr[end]
    beq t2, t1 else      # arr[end] == 0x20?
    addi a2, a2, 1        # end++
    j done2               # skip over else
else:
    addi a2, a2, -1       # end = end-1
    jal reverse           # reverse(arr, begin, end-1)
    addi a2, a2, 1        # end = end+1
    addi a2, a2, 1        # end++
    add a1, a2, zero      # begin = end
    j while               # repeat loop
done2:
    jr ra                 # return
# a0 = address of arr[], a1 = begin, a2 = end
reverse:
    add t1, a1, a0        # t1 = address of arr[begin]
    add t3, a2, a0        # t3 = address of arr[end]
while2:
    bge t1, t3 done3     # begin >= end?
    lb t2, 0(t1)          # t2 = arr[begin]
    lb t4, 0(t3)          # t4 = arr[end]
    sb t4, 0(t1)          # arr[begin] = original arr[end]
    sb t2, 0(t3)          # arr[end] = original arr[begin]
    addi t1, t1, 1         # address of arr[begin]++
    addi t3, t3, -1        # address of arr[end]--
    j while2               # repeat loop
done3:
    jr ra                 # return:

```

Question 6.5

```

# a0 = register on which to reverse bits
reverseBits:
    addi t1, zero, 31      # t1 = 31
    addi t2, zero, 0        # t2 = 0
    addi t3, zero, 0        # t3 = 0
L7:
    beq t1, zero, done3   # if t1 = 0, done

```

```

srl  t0, a0, t1      # t0 = a0 >> t1
andi t0, t0, 1       # isolate lsb of t0
sll  t0, t0, t2      # t0 = t0 << t2
or   t3, t3, t0      # combine bits
addi t1, t1, -1     # t1--
addi t2, t2, 1       # t2++
j    L7              # repeat the loop
done3:
add  a0, t3, zero   # set a0 equal to its reversed self
jr  ra              # return

```

Question 6.7

High-Level Algorithm

```

bool isPalindrome(char str[]) {
    int begin = 0;
    int end = 0;

    // first find the index of the last character
    for (end = 0; str[end] != 0; end++);
    end--;

    // check if each character pair matches
    // if one does not, the string is not a palindrome
    while(end > begin){
        if (str[begin] != str[end])
            return false;
        begin++;
        end--;
    }
    // if the above check passed then it is a palindrome
    return true;
}

```

RISC-V Assembly Code

```

isPalindrome:
# a0 = base address of str[]
    addi t0, zero, 0      # t0 = begin = 0
    addi t1, zero, 0      # t1 = end = 0
for:
    add  t2, a0, t1      # t2 = address of str[end]
    lb   t2, 0(t2)        # t2 = str[end]
    beq t2, zero, done   # stop counting when str[end] is null
    addi t1, t1, 1        # end++
    j    for              # repeat loop
done:
    addi t1, t1, -1      # end--
while:
    bge t0, t1, yes      # if all chars matched, then jump to yes
    add  t2, t0, a0        # t2 = address of str[begin]

```

```
lb    t2, 0(t2)      # t2 = str[begin]
add  t3, t1, a0       # t3 = address of str[end]
lb    t3, 0(t3)      # t3 = str[end]
bne  t2, t3, isnt   # not a palindrome if not equal
addi t0, t0, 1        # begin++
addi t1, t1, -1       # end--
j     while          # repeat loop

yes:
addi a0, zero, 1      # set a0 to 1: it is a palindrome
jr ra                  # return

isnt:
addi a0, zero, 0       # set a0 to 0: it isn't a palindrome
jr ra                  # return
```

CHAPTER 7

Exercise 7.1

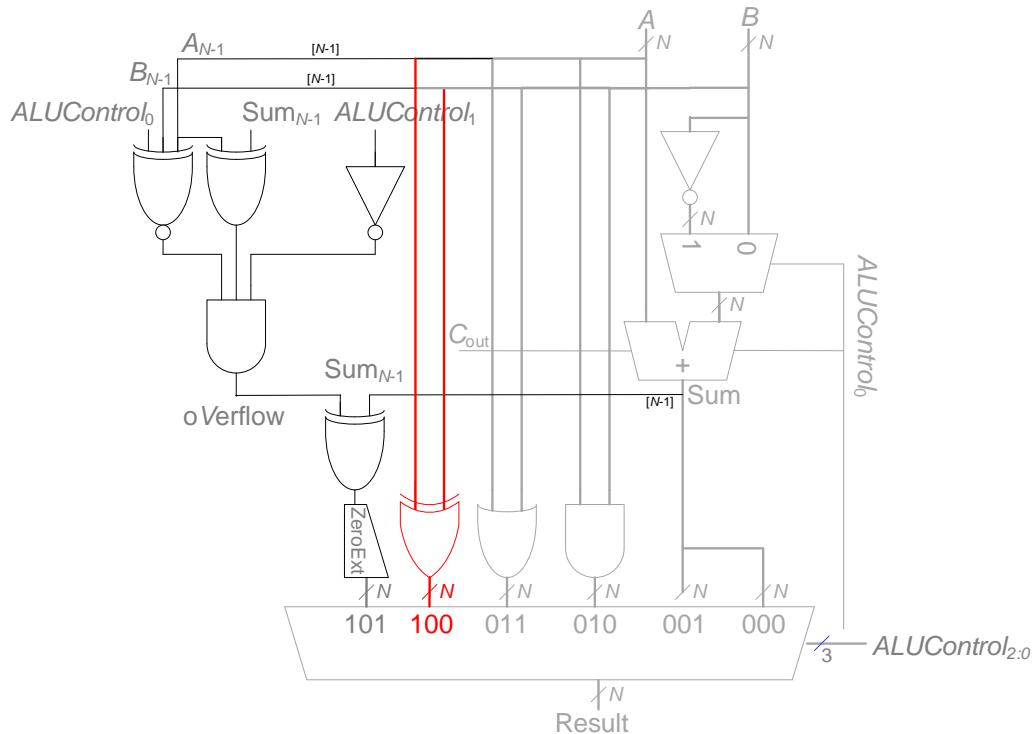
- (a) **RegWrite:** lw, addi, jal, and R-type instructions – WE3 will be 0, so no data will be written to the Register File.
- (b) **ALUOp₁:** R-type instructions except add – These instructions all require a 1 in ALUOp₁ for the ALU Decoder to produce the correct FsALUControl signal.
- (c) **ALUOp₀:** beq – The ALU would incorrectly add the registers rather than subtracting them before checking the Zero flag.
- (d) **MemWrite:** sw - WE will not enable the Data Memory to write.
- (e) **ImmSrc₁:** beq and jal – The incorrect immediates (offsets) are selected.
- (f) **ImmSrc₀:** sw and jal – The incorrect immediates are selected.
- (g) **ResultSrc₁:** jal – ALUResult will be selected instead of PCPlus4 as the result to write to the Register File.
- (h) **ResultSrc₀:** lw – ALUResult will be selected instead of ReadData as the result to write to the Register File.
- (i) **PCSrc:** beq and jal – PCPlus4 will always be selected as PCNext instead of the new PCTarget.
- (j) **ALUSrc:** lw, sw, and addi (and other I-type ALU instructions) – SrcB for the ALU will incorrectly select RD2 instead of ImmExt.

Exercise 7.3

- (a) xor

The datapath does not require any changes to its interfaces. Only the ALU needs to be modified: we add another input to the multiplexer and N 2-bit XOR gates within the ALU. We also update the ALU Decoder truth table / logic. The Main Decoder truth table need not be updated because it already supports R-type instructions. These changes are shown below.

Modified ALU to support xor



Modified ALU operations to support xor

$ALUControl_{2:0}$	Function
000	add
001	subtract
010	and
011	or
100	xor
101	SLT

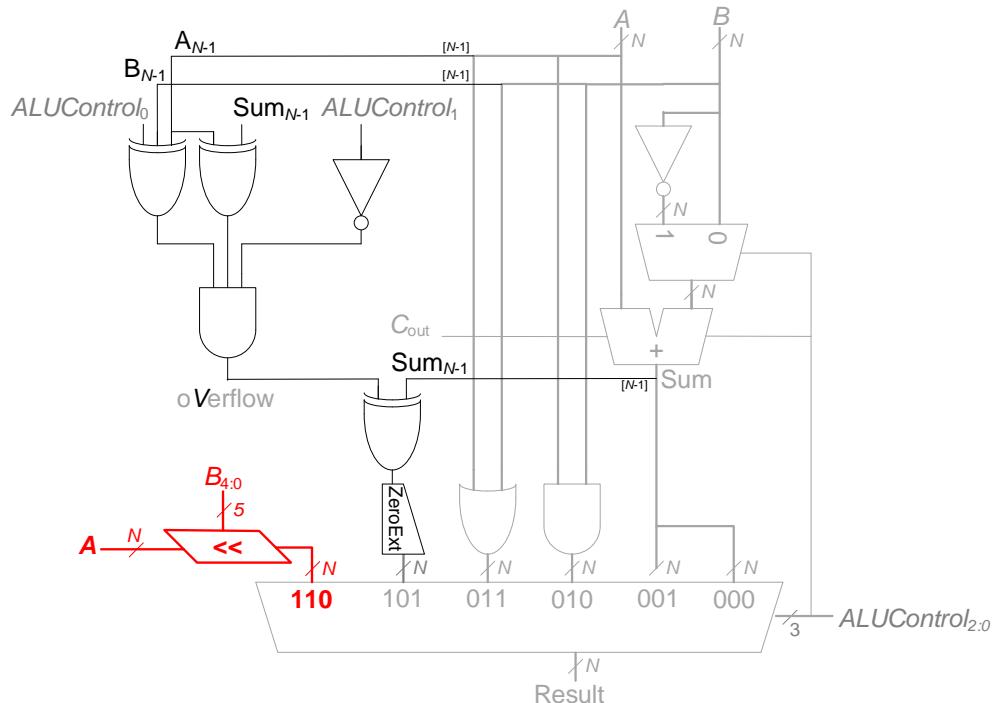
Modified ALU Decoder truth table to support xor

$ALUOp$	$funct3$	$ops, funct7_5$	$ALUControl$	Instruction
00	x	x	000 (add)	lw, sw
01	x	x	001 (subtract)	beq
10	000	00, 01, 10	000 (add)	add, addi
	000	11	001 (subtract)	sub
	010	x	101 (set less than)	slt, slti
	100	x	100 (xor)	xor, xori
	110	x	011 (or)	or, ori
	111	x	010 (and)	and, andi

(b) sll

The overall datapath (interfaces and units) need not be changed. We only modify the ALU and the ALU Decoder, as shown below. We add a shifter and expand the multiplexer inside the ALU.

Modified ALU to support sll



Modified ALU operations to support sll

<i>ALUControl_{2:0}</i>	Function
000	add
001	subtract
010	and
011	or
101	SLT
110	sll

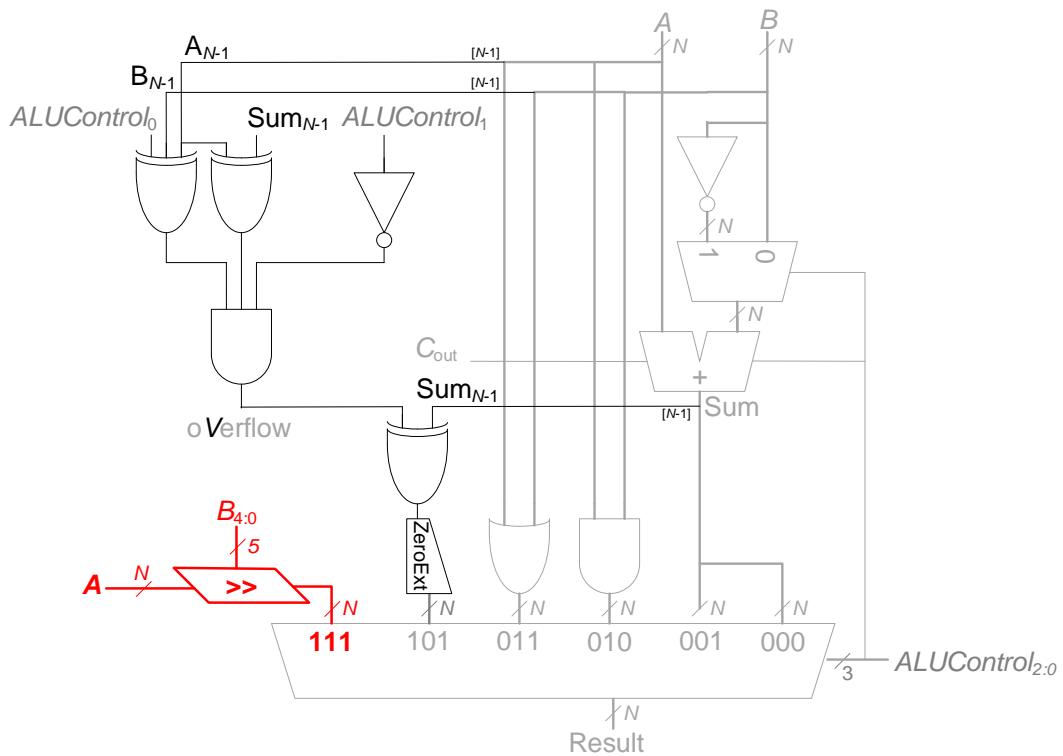
Modified ALU Decoder truth table to support sll

<i>ALUOp</i>	<i>funct3</i>	<i>ops₅, funct7₅</i>	<i>ALUControl</i>	<i>Instruction</i>
00	x	x	000 (add)	lw, sw
01	x	x	001 (subtract)	beq
10	000	00, 01, 10	000 (add)	add, addi
	000	11	001 (subtract)	sub
	001	x	110 (shift left logical)	sll, slli
	010	x	101 (set less than)	slt, slti
	110	x	011 (or)	or, ori
	111	x	010 (and)	and, andi

(c) srl

The overall datapath (interfaces and units) need not be changed. We only modify the ALU and the ALU Decoder, as shown below. We add a shifter and expand the multiplexer inside the ALU.

Modified ALU to support srl



Modified ALU operations to support srl

$ALUControl_{2:0}$	Function
000	add
001	subtract
010	and
011	or
101	SLT
111	srl

Modified ALU Decoder truth table to support srl

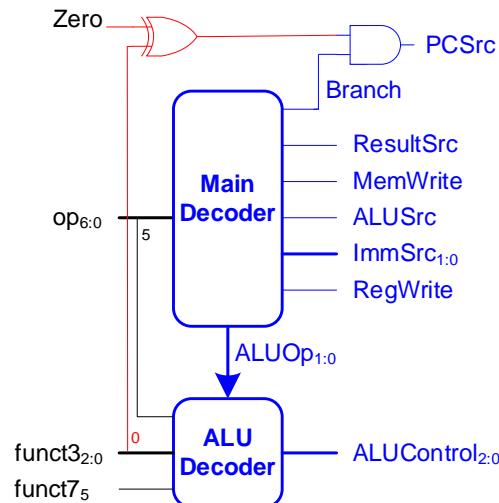
ALUOp	funct3	op5, funct7₅	ALUControl	Instruction
00	x	xx	000 (add)	lw, sw
01	x	xx	001 (subtract)	beq
10	000	00, 01, 10	000 (add)	add, addi
	000	11	001 (subtract)	sub
	001	x0	111 (shift right logical)	srl, srli
	010	xx	101 (set less than)	slt, slti
	110	xx	011 (or)	or, ori
	111	xx	010 (and)	and, andi

(d) bne

bne is the opposite of beq. beq and bne can be identified by **func3₀**, which is high when bne is the instruction. To implement, we simply need to change the control unit to branch when Zero is 0 and bne is the instruction or when Zero is 1 and beq is the instruction. This is easily achieved with Zero XOR **func3₀**.

Main Decoder truth table enhanced to support bne

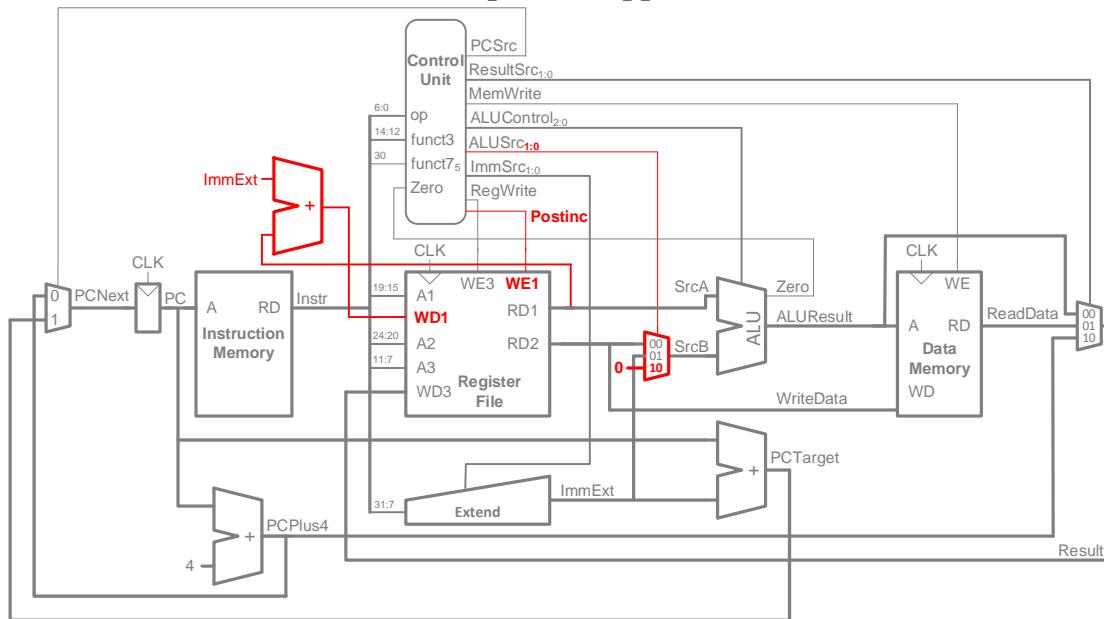
Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
Lw	0000011	1	00	1	0	01	0	00	0
Sw	0100011	0	01	1	1	xx	0	00	0
R-type	0110011	1	xx	0	0	00	0	10	0
beq/bne	1100011	0	10	0	0	xx	1	01	0
Addi	0010011	1	00	0	0	00	0	10	0
Jal	1101111	1	11	x	0	10	0	xx	1

Enhanced control unit for bne

Exercise 7.5

To implement `lwpostinc`, we have to modify the register file by adding another write port and another adder. We also add a new control signal, `Postinc`, and expand the ALUSrc multiplexer to include 0 as an input. That way, the address is calculated as `rs1 + 0`. The extra adder produces `rs1 + ImmExt` and writes it back to `rs1`.

Enhanced datapath to support `lwpostinc`



Main Decoder truth table enhanced to support `lwpostinc`

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump	PostInc
lw	0000011	1	000	01	0	01	0	00	0	0
sw	0100011	0	001	01	1	xx	0	00	0	0
R-type	0110011	1	xxx	00	0	00	0	10	0	0
beq	1100011	0	010	00	0	xx	1	01	0	0
I-type ALU	0010011	1	000	00	0	00	0	10	0	0
jal	1101111	1	011	xx	0	10	0	xx	1	0
lwpostinc	new op	1	000	01	0	01	0	00	0	1

Exercise 7.7

To increase performance most (i.e., decrease cycle time), the crack circuit designer should speed up the Memory Unit. From Equation 7.3:

$$\begin{aligned} T_{c_new} &= t_{pcq_PC} + 2t_{mem} + t_{RFread} + t_{ALU} + t_{mux} + t_{RFsetup} \\ &= 40 + 2(100) + 100 + 120 + 30 + 60 = 550 \text{ ps} \end{aligned}$$

Exercise 7.9

RISC-V single-cycle processor SystemVerilog:

```
// Modified to include all Exercise 7.3 instructions (xor, sll, srl, bne)

module testbench();

    logic      clk;
    logic      reset;

    logic [31:0] WriteData, DataAddr;
    logic      MemWrite;

    // instantiate device to be tested
    top dut(clk, reset, WriteData, DataAddr, MemWrite);

    // initialize test
    initial
    begin
        reset <= 1; # 22; reset <= 0;
    end

    // generate clock to sequence tests
    always
    begin
        clk <= 1; # 5; clk <= 0; # 5;
    end

    // check results
    always @ (negedge clk)
    begin
        if(MemWrite) begin
            if(DataAddr === 216 & WriteData === 4140) begin
                $display("Simulation succeeded");
                $stop;
            end
        end
    end
endmodule

module top(input  logic      clk, reset,
           output logic [31:0] WriteData, DataAddr,
           output logic      MemWrite);

    logic [31:0] PC, Instr, ReadData;

    // instantiate processor and memories
    riscvsingle rvsingle(clk, reset, PC, Instr, MemWrite, DataAddr,
                         WriteData, ReadData);
    imem imem(PC, Instr);
    dmem dmem(clk, MemWrite, DataAddr, WriteData, ReadData);
endmodule

module riscvsingle(input  logic      clk, reset,
                   output logic [31:0] PC,
                   input  logic [31:0] Instr,
```

```

        output logic      MemWrite,
        output logic [31:0] ALUResult, WriteData,
        input  logic [31:0] ReadData);

logic      ALUSrc, RegWrite, Jump, Zero;
logic [1:0] ResultSrc, ImmSrc;
logic [2:0] ALUControl;

controller c(Instr[6:0], Instr[14:12], Instr[30], Zero,
             ResultSrc, MemWrite, PCSrc,
             ALUSrc, RegWrite, Jump,
             ImmSrc, ALUControl);
datapath dp(clk, reset, ResultSrc, PCSrc,
            ALUSrc, RegWrite,
            ImmSrc, ALUControl,
            Zero, PC, Instr,
            ALUResult, WriteData, ReadData);
endmodule

module controller(input logic [6:0] op,
                  input logic [2:0] funct3,
                  input logic      funct7b5,
                  input logic      Zero,
                  output logic [1:0] ResultSrc,
                  output logic      MemWrite,
                  output logic      PCSrc, ALUSrc,
                  output logic      RegWrite, Jump,
                  output logic [1:0] ImmSrc,
                  output logic [2:0] ALUControl);

logic [1:0] ALUOp;
logic      Branch;

maindec md(op, ResultSrc, MemWrite, Branch,
           ALUSrc, RegWrite, Jump, ImmSrc, ALUOp);
aludec ad(op[5], funct3, funct7b5, ALUOp, ALUControl);

// added XOR gate for bne
assign PCSrc = (Branch & (Zero ^ funct3[0])) | Jump;
endmodule

module maindec(input logic [6:0] op,
               output logic [1:0] ResultSrc,
               output logic      MemWrite,
               output logic      Branch, ALUSrc,
               output logic      RegWrite, Jump,
               output logic [1:0] ImmSrc,
               output logic [1:0] ALUOp);

logic [10:0] controls;

assign {RegWrite, ImmSrc, ALUSrc, MemWrite,
        ResultSrc, Branch, ALUOp, Jump} = controls;

always_comb
  case(op)
    // RegWrite_ImmSrc_ALUSrc_MemWrite_ResultSrc_Branch_ALUOp_Jump

```

```

7'b00000011: controls = 11'b1_00_1_0_01_0_00_0; // lw
7'b0100011: controls = 11'b0_01_1_1_00_0_00_0; // sw
7'b0110011: controls = 11'b1_xx_0_0_00_0_10_0; // R-type
7'b1100011: controls = 11'b0_10_0_0_00_1_01_0; // beq
7'b0010011: controls = 11'b1_00_1_0_00_0_10_0; // I-type ALU
7'b1101111: controls = 11'b1_11_0_0_10_0_00_1; // jal
default: controls = 11'bx_xx_x_x_xx_x_xx_x; // non-implemented
instruction
    endcase
endmodule

module aludec(input logic      opb5,
               input logic [2:0] funct3,
               input logic      funct7b5,
               input logic [1:0] ALUOp,
               output logic [2:0] ALUControl);

    logic RtypeSub;
    assign RtypeSub = funct7b5 & opb5; // TRUE for R-type subtract instruction

    always_comb
        case(ALUOp)
            2'b00:           ALUControl = 3'b000; // addition
            2'b01:           ALUControl = 3'b001; // subtraction
            default:         case(funct3) // R-type or I-type ALU
                3'b000: if (RtypeSub)
                    ALUControl = 3'b001; // sub
                else
                    ALUControl = 3'b000; // add, addi
                3'b001:   ALUControl = 3'b110; // sll, slli
                3'b010:   ALUControl = 3'b101; // slt, slti
                3'b100:   ALUControl = 3'b100; // xor, xori
                3'b101:   ALUControl = 3'b111; // srl, srli
                3'b110:   ALUControl = 3'b011; // or, ori
                3'b111:   ALUControl = 3'b010; // and, andi
                default:  ALUControl = 3'bxxx; // ???
        endcase
    endcase
endmodule

module datapath(input logic      clk, reset,
                 input logic [1:0] ResultSrc,
                 input logic      PCSrc, ALUSrc,
                 input logic      RegWrite,
                 input logic [1:0] ImmSrc,
                 input logic [2:0] ALUControl,
                 output logic     Zero,
                 output logic [31:0] PC,
                 input logic [31:0] Instr,
                 output logic [31:0] ALUResult, WriteData,
                 input logic [31:0] ReadData);

    logic [31:0] PCNext, PCPlus4, PCTarget;
    logic [31:0] ImmExt;
    logic [31:0] SrcA, SrcB;
    logic [31:0] Result;

```

```

// next PC logic
flop #(32) pcreg(clk, reset, PCNext, PC);
adder      pcadd4(PC, 32'd4, PCPlus4);
adder      pcaddbranch(PC, ImmExt, PCTarget);
mux2 #(32) pcmux(PCPlus4, PCTarget, PCSrc, PCNext);

// register file logic
regfile    rf(clk, RegWrite, Instr[19:15], Instr[24:20],
             Instr[11:7], Result, SrcA, WriteData);
extend     ext(Instr[31:7], ImmSrc, ImmExt);

// ALU logic
mux2 #(32) srcbmux(WriteData, ImmExt, ALUSrc, SrcB);
alu        alu(SrcA, SrcB, ALUControl, ALUResult, Zero);
mux3 #(32) resultmux(ALUResult, ReadData, PCPlus4, ResultSrc, Result);
endmodule

module regfile(input  logic      clk,
               input  logic      we3,
               input  logic [4:0] a1, a2, a3,
               input  logic [31:0] wd3,
               output logic [31:0] rd1, rd2);

  logic [31:0] rf[31:0];

  // three ported register file
  // read two ports combinationally (A1/RD1, A2/RD2)
  // write third port on rising edge of clock (A3/WD3/WE3)
  // register 0 hardwired to 0

  always_ff @(posedge clk)
    if (we3) rf[a3] <= wd3;

  assign rd1 = (a1 != 0) ? rf[a1] : 0;
  assign rd2 = (a2 != 0) ? rf[a2] : 0;
endmodule

module adder(input  [31:0] a, b,
              output [31:0] y);

  assign y = a + b;
endmodule

module extend(input  logic [31:7] instr,
              input  logic [1:0] immsrc,
              output logic [31:0] immext);

  always_comb
    case(immsrc)
      // I-type
      2'b00:  immext = {{20{instr[31]}}, instr[31:20]};
      // S-type (stores)
      2'b01:  immext = {{20{instr[31]}}, instr[31:25], instr[11:7]};
      // B-type (branches)
      2'b10:  immext = {{20{instr[31]}}, instr[7], instr[30:25],
                         instr[11:8], 1'b0};
      // J-type (jal)

```

```

2'b11:  immext = {{12{instr[31]}}, instr[19:12], instr[20],
instr[30:21], 1'b0};
      default: immext = 32'b0; // undefined
      endcase
endmodule

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

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= d;
endmodule

module mux2 #(parameter WIDTH = 8)
    (input logic [WIDTH-1:0] d0, d1,
     input logic           s,
     output logic [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule

module mux3 #(parameter WIDTH = 8)
    (input logic [WIDTH-1:0] d0, d1, d2,
     input logic [1:0]        s,
     output logic [WIDTH-1:0] y);

  assign y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule

module imem(input logic [31:0] a,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  initial
    $readmemh("example.txt",RAM);

  assign rd = RAM[a[31:2]]; // word aligned
endmodule

module dmem(input logic      clk, we,
            input logic [31:0] a, wd,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  assign rd = RAM[a[31:2]]; // word aligned

  always_ff @(posedge clk)
    if (we) RAM[a[31:2]] <= wd;
endmodule

module alu(input logic [31:0] a, b,
            input logic [2:0]  alucontrol,

```

```

        output logic [31:0] result,
        output logic          zero);

logic [31:0] condinvb, sum;
logic      v;                      // overflow
logic      isAddSub;               // true when is add or subtract operation

assign condinvb = alucontrol[0] ? ~b : b;
assign sum = a + condinvb + alucontrol[0];
assign isAddSub = ~alucontrol[2] & ~alucontrol[1] |
                  ~alucontrol[1] & alucontrol[0];

always_comb
  case (alucontrol)
    3'b000: result = sum;           // add
    3'b001: result = sum;           // subtract
    3'b010: result = a & b;         // and
    3'b011: result = a | b;         // or
    3'b100: result = a ^ b;         // xor
    3'b101: result = sum[31] ^ v;   // slt
    3'b110: result = a << b[4:0]; // sll
    3'b111: result = a >> b[4:0]; // srl
    default: result = 32'bx;
  endcase

  assign zero = (result == 32'b0);
  assign v = ~(alucontrol[0] ^ a[31] ^ b[31]) & (a[31] ^ sum[31]) & isAddSub;
endmodule

```

Modified test program:

# If successful, it should write the value 4140 (0x102C) to address 216 (0xd8).				
	RISC-V Assembly	Description	Address	Machine Code
main:	addi x2, x0, 5	# x2 = 5	0	00500113
	addi x3, x0, 12	# x3 = 12	4	00C00193
	addi x7, x3, -9	# x7 = (12 - 9) = 3	8	FF718393
	or x4, x7, x2	# x4 = (3 OR 5) = 7	C	0023E233
	and x5, x3, x4	# x5 = (12 AND 7) = 4	10	0041F2B3
	add x5, x5, x4	# x5 = (4 + 7) = 11	14	004282B3
	sll x5, x5, x3	# x5 = 11 << 12 = 45,056	18	003292b3
	srl x5, x5, x2	# x5 = 45,056 >> 5 = 1408	1C	0022d2b3
	bne x5, x3, skip	# 1408 != 12: branch taken	20	00329463
	sll x5, x5, x3	# shouldn't execute	24	003292b3
skip:	beq x5, x7, end	# shouldn't be taken	28	02728863
	slt x4, x3, x4	# x4 = (12 < 7) = 0	2C	0041A233
	beq x4, x0, around	# should be taken	30	00020463
	addi x5, x0, 0	# shouldn't execute	34	00000293
around:	slt x4, x7, x2	# x4 = (3 < 5) = 1	38	0023A233
	add x7, x4, x5	# x7 = (1 + 1408) = 1409	3C	005203B3
	sub x7, x7, x2	# x7 = (1409 - 5) = 1404	40	402383B3
	sw x7, 200(x3)	# [212] = 1404	44	0c71a423
	lw x2, 212(x0)	# x2 = [212] = 1404	48	0d402103
	add x9, x2, x5	# x9 = (1404 + 1408) = 2812	4C	005104B3
	jal x3, end	# jump to end, x3 = 0x54	50	008001EF
	addi x2, x0, 1	# shouldn't execute	54	00100113
end:	add x2, x2, x9	# x2 = (1404 + 2812) = 4216	58	00910133
	addi x4, x0, -1	# x4 = 0xFFFFFFFF	5C	ffff00213
	addi x5, x0, 1	# x5 = 1	60	00100293
	addi x6, x0, 31	# x6 = 31	64	01f00313
	sll x6, x5, x6	# x6 = 0x80000000	68	00629333
	xor x5, x4, x6	# x5 = 0x7FFFFFFF	6C	006242b3
	slt x6, x5, x4	# x6 = 0	70	0042a333
wrong:	bne x6, x0, wrong	# shouldn't be taken	74	00031063

```

xor  x2, x2, x3      # x2 = 4216 ^ 0x54 = 4140    78      00314133
sw   x2, 0x84(x3)    # mem[216] = 0x102C = 4140    7C      0821a223
done: beq  x2, x2, done  # infinite loop           80      00210063

```

Exercise 7.11

- (a) ***ResultSrc₁*:** All instructions will fail because the program counter will not update to PC+4. Instead, it will be incorrectly updated to the previous ALU result because *ResultSrc* would be 00 instead of 10.
- (b) ***ResultSrc₀*:** `lw` – Instead of loading the data read from the memory into the register, the previous ALU result will be stored as the result because *ResultSource* would be 00 instead of 01.
- (c) ***ALUSrcB₁*:** All instructions will fail because during the Fetch state, the PC will increment by whatever happens to be in *WriteData* at the time instead of by 4 because to the ALU receiving the wrong source.
- (d) ***ALUSrcB₀*:** All instructions that use an immediate (`addi`, `beq`, `lw`, `sw`, etc) will fail because the immediate can never be selected and used due to the stuck-at-0 *ALUSrcB₀*.
- (e) ***ALUSrcA₁*:** `lw`, `sw`, `beq`, R-type, and I-type ALU instructions – RD1 will never be able to be selected and used due to the stuck-at-0 *ALUSrcA₁*.
- (f) ***ALUSrcA₀*:** `jal` and `beq` – The branch/jump target address will not be able to be calculated because *OldPC* can never be selected and used due to the stuck-at-0 *ALUSrcA₀*.
- (g) ***ImmSrc₁*:** `jal` and `beq` – The branch/jump offsets (immediates) could not be selected.
- (h) ***ImmSrc₀*:** `sw` and `jal` – The address and jump offsets (immediates), respectively, could not be selected.
- (i) ***RegWrite*:** `lw`, `jal`, R-type, and I-type ALU instructions – The register file will never be written to.
- (j) ***PCUpdate*:** All instructions will fail because the program counter will never be updated and, thus, we will execute the same instruction repeatedly.
- (k) ***Branch*:** `beq` will malfunction whenever the branch is taken. Because *Branch* is stuck at zero, the PC will never update to the target PC whenever a branch should be taken.
- (l) ***AdrSrc*:** `lw` and `sw` – Instead of the proper data address being used for loads and stores, the PC will be used which in the case of a load would fetch garbage data and, in the case of a store, would corrupt the program.
- (m) ***MemWrite*:** `sw` – Memory cannot be written.
- (n) ***IRWrite*:** All instructions will fail because the instruction would never be written to the instruction register and thus no instructions could execute.

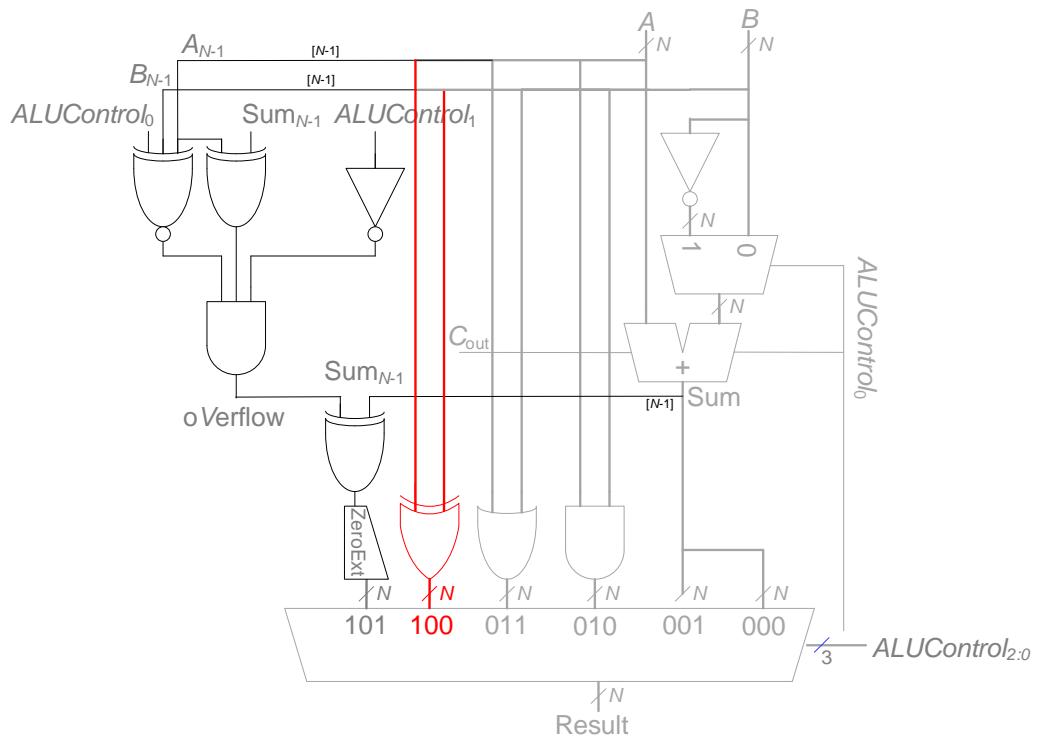
Exercise 7.13

(a) xor

Neither the datapath or the Main FSM need to be modified, because they already support R-type instructions.

Only the ALU needs to be modified: we add another input to the multiplexer and N 2-bit XOR gates within the ALU. We also update the ALU Decoder truth table / logic. The changes are shown below.

Modified ALU to support xor



Modified ALU operations to support xor

$ALUControl_{2:0}$	Function
000	add
001	subtract
010	and
011	or
100	xor
101	SLT

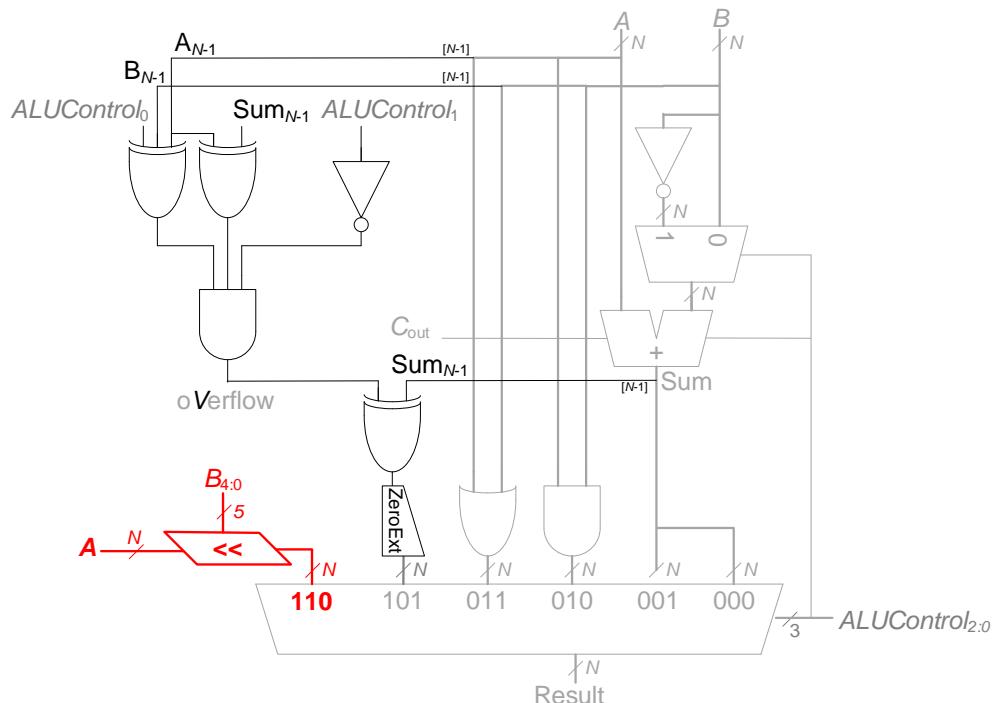
Modified ALU Decoder truth table to support xor

ALUOp	funct3	ops, funct7s	ALUControl	Instruction
00	x	x	000 (add)	lw, sw
01	x	x	001 (subtract)	beq
10	000	00, 01, 10	000 (add)	add, addi
	000	11	001 (subtract)	sub
	010	x	101 (set less than)	slt, slti
	100	x	100 (xor)	xor, xorri
	110	x	011 (or)	or, ori
	111	x	010 (and)	and, andi

(b) sll

Neither the datapath or the Main FSM need to be modified, because they already support R-type instructions.

We only modify the ALU and the ALU Decoder, as shown below. We add a shifter and expand the multiplexer inside the ALU.

Modified ALU to support sll

Modified ALU operations to support sll

<i>ALUControl_{2:0}</i>	Function
000	add
001	subtract
010	and
011	or
101	SLT
110	sll

Modified ALU Decoder truth table to support sll

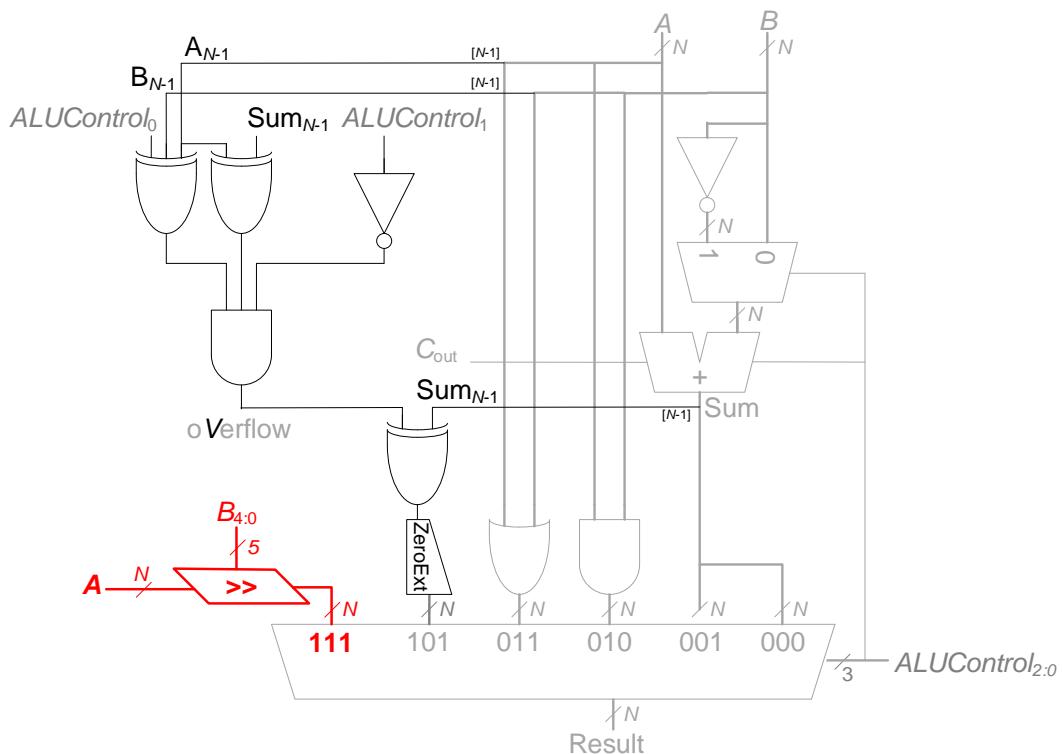
<i>ALUOp</i>	<i>funct3</i>	<i>op₅, funct_{7:5}</i>	<i>ALUControl</i>	<i>Instruction</i>
00	x	x	000 (add)	lw, sw
01	x	x	001 (subtract)	beq
10	000	00, 01, 10	000 (add)	add, addi
	000	11	001 (subtract)	sub
	001	x	110 (shift left logical)	sll, slli
	010	x	101 (set less than)	slt, slti
	110	x	011 (or)	or, ori
	111	x	010 (and)	and, andi

(c) srl

Neither the datapath or the Main FSM need to be modified, because they already support R-type instructions.

We only modify the ALU and the ALU Decoder, as shown below. We add a shifter and expand the multiplexer inside the ALU.

Modified ALU to support srl



Modified ALU operations to support srl

$ALUControl_{2:0}$	Function
000	add
001	subtract
010	and
011	or
101	SLT
111	srl

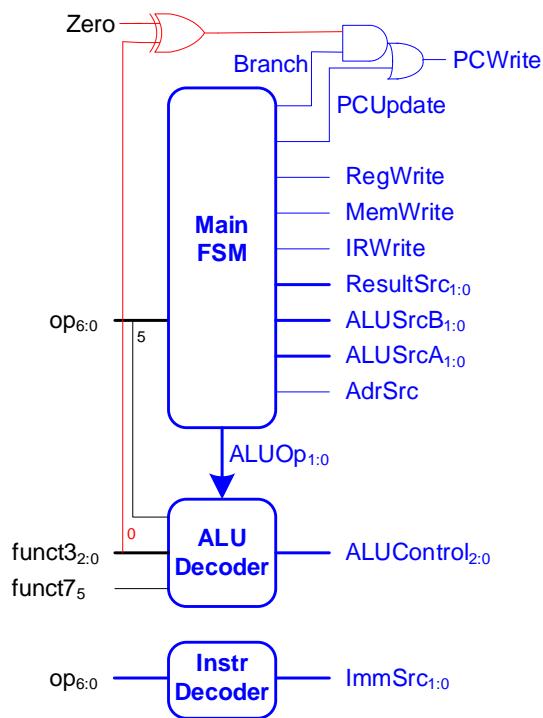
Modified ALU Decoder truth table to support srl

<i>ALUOp</i>	<i>funct3</i>	<i>op5, funct7s</i>	<i>ALUControl</i>	<i>Instruction</i>
00	x	xx	000 (add)	lw, sw
01	x	xx	001 (subtract)	beq
10	000	00, 01, 10	000 (add)	add, addi
	000	11	001 (subtract)	sub
	001	x0	111 (shift right logical)	srl, srli
	010	xx	101 (set less than)	slt, slti
	110	xx	011 (or)	or, ori
	111	xx	010 (and)	and, andi

(d) bne

bne is the opposite of beq. beq and bne can be identified by **func30**, which is high when bne is the instruction and low for beq. To implement, we simply need to change the control unit to branch when *Zero* is 0 and bne is the instruction or when *Zero* is 1 and beq is the instruction. This is easily achieved with *Zero* XOR **func30**.

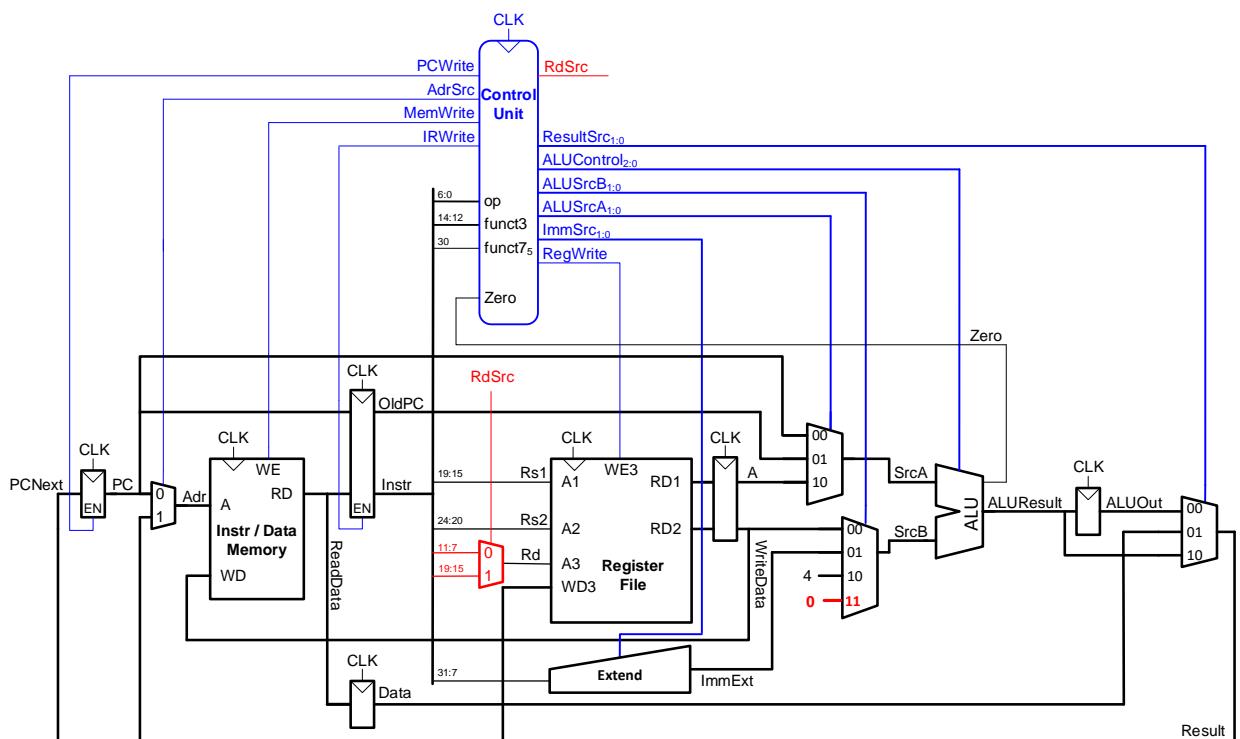
Enhanced control unit for bne



Exercise 7.15

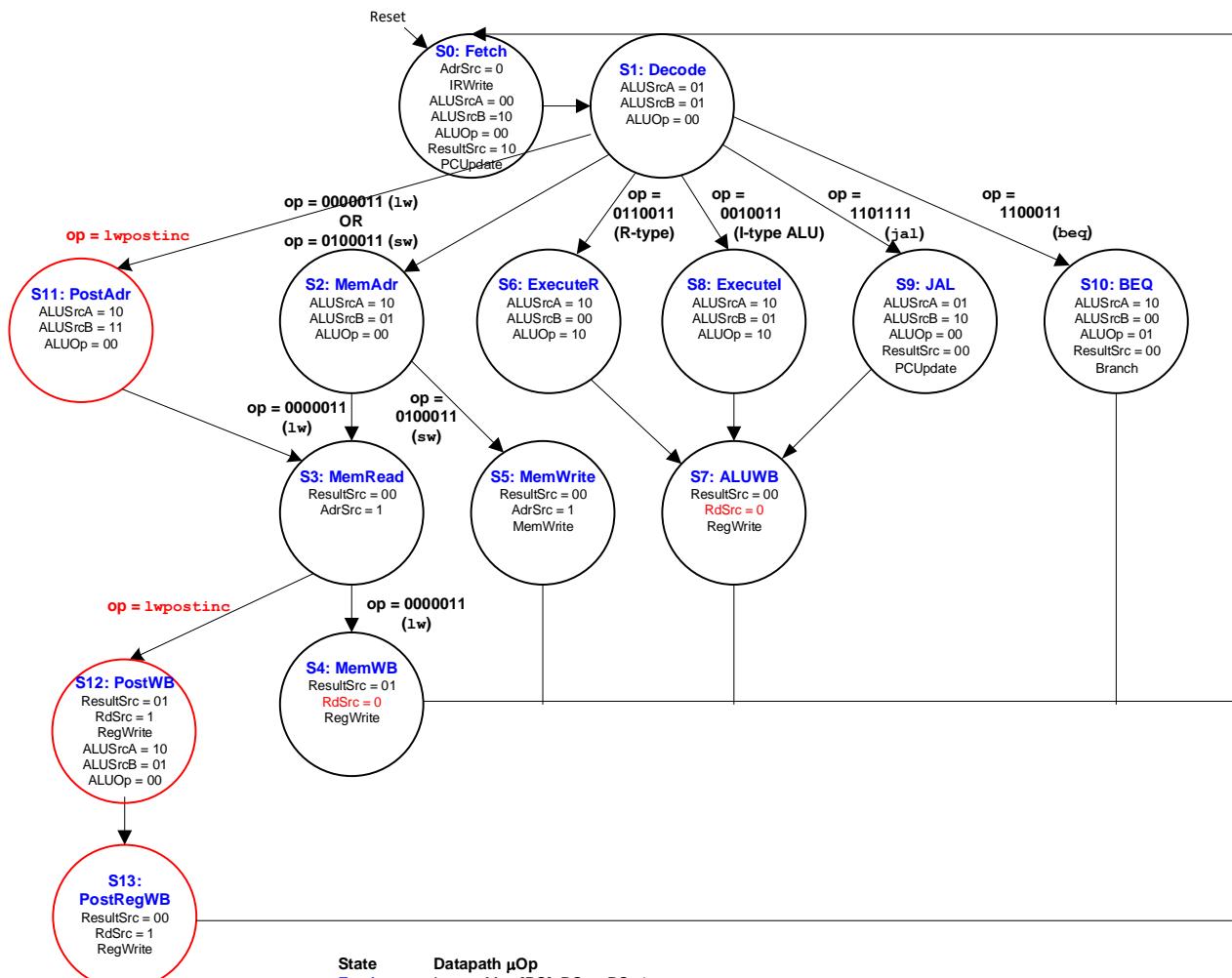
It is possible to add the `lwpostinc` instruction without modifying the register file. We modify the datapath so that the address can be calculated as: **rs1** + 0, as shown below. We added a 0 input to the **ALUSrcB** multiplexer, a multiplexer to choose between **rs1** and **rd** as the register to be written, and a **RdSrc** signal for the added mux.

Enhanced datapath to support `lwpostinc`



We add three states to the Main FSM: PostAdr, PostWB, PostRegWB. PostAdr calculates the address as **rs1** + 0. `lwpostinc` then proceeds to the MemRead state followed by the PostWB state, where it writes the loaded data to **rd** (and simultaneously calculates **rs1** + imm). Finally the instruction writes **rs1** + imm back to **rs1** in the PostRegWB state.

Main FSM showing added states and signals to support lwpostinc



State	Datapath μOp
Fetch	Instr ← Mem[PC]; PC ← PC+4
Decode	ALUOut ← PCTarget
MemAddr	ALUOut ← rs1 + imm
MemRead	Data ← Mem[ALUOut]
MemWB	rd ← Data
MemWrite	Mem[ALUOut] ← rd
ExecuteR	ALUOut ← rs1 op rs2
Executel	ALUOut ← rs1 op imm
ALUWB	rd ← ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC = ALUOut
JAL	PC = ALUOut; ALUOut = PC+4
PostAddr	ALUOut ← rs1 + 0
PostWB	rd ← ALUOut; ALUOut ← rs1 + imm
PostRegWB	rs1 ← ALUOut

Exercise 7.17

The crack circuit designer should speed up the Memory Unit.

$$\begin{aligned}
 T_{c_multi_new} &= t_{pcq} + t_{dec} + 2t_{mux} + \max [t_{ALU}, t_{mem}] + t_{setup} \\
 &= 40 + 25 + 2(30) + 120 + 50 = 295 \text{ ps}
 \end{aligned}$$

Exercise 7.19

The crack circuit designer should speed up the Memory Unit and should make it equal to the delay of the ALU (120ps). The cycle time would then be 295 ps, the same calculation as in Exercise 7.17.

Exercise 7.21

Alyssa should switch to the slower but lower power register file. By doubling the delay of the register file, it still does not place it on the critical path. This means that power will be saved without affecting the cycle time.

Specifically, the path that includes the register files would require the following constraints:

$$\begin{aligned} T_{c_multi_RF} &= t_{pcq} + t_{RFread} + t_{setup} \\ T_{c_multi_RF} &= (40 + 100 + 50) \text{ ps} = 190 \text{ ps} \end{aligned}$$

With twice as much register file (RF) delay, this constraint would be:

$T_{c_multi_RF} = (40 + 2 * 100 + 50) \text{ ps} = 290 \text{ ps}$, which is still less than the 375 ps cycle time required by the path through memory.

Exercise 7.23

The program will execute 6 addi (4 cycles each), 6 bge (3 cycles each), and 5 jal (4 cycles each) instructions for a total of $(6 \times 4) + (6 \times 3) + (5 \times 4) = 62 \text{ clock cycles}$ for 17 instructions. Thus, the CPI of this program is $62/17 = 3.65 \text{ CPI}$. (Remember that $j _L1$ is a pseudoinstruction for $jal _x0, _L1$.)

Exercise 7.25

RISC-V multicycle processor

```
module testbench();
    logic      clk;
    logic      reset;

    logic [31:0] WriteData, DataAdr;
    logic      MemWrite;

    // instantiate device to be tested
    top dut(clk, reset, WriteData, DataAdr, MemWrite);

    // initialize test
    initial
        begin
            reset <= 1; # 22; reset <= 0;
        end

```

```

// generate clock to sequence tests
always
begin
    clk <= 1; # 5; clk <= 0; # 5;
end

// check results
always @ (negedge clk)
begin
    if (MemWrite) begin
        if (DataAddr === 100 & WriteData === 25) begin
            $display("Simulation succeeded");
            $stop;
        end else if (DataAddr !== 96) begin
            $display("Simulation failed");
            $stop;
        end
    end
end
endmodule

module top (input logic      clk, reset,
            output logic [31:0] WriteDataM, DataAddrM,
            output logic          MemWriteM);

    logic [31:0] PCF, InstrF, ReadDataM;

    // instantiate processor and memories
    riscv riscv(clk, reset, PCF, InstrF, MemWriteM, DataAddrM,
                WriteDataM, ReadDataM);
    imem imem(PCF, InstrF);
    dmem dmem(clk, MemWriteM, DataAddrM, WriteDataM, ReadDataM);
endmodule

module riscv (input logic      clk, reset,
              output logic [31:0] PCF,
              input logic [31:0] InstrF,
              output logic          MemWriteM,
              output logic [31:0] ALUResultM, WriteDataM,
              input logic [31:0] ReadDataM);

    logic [6:0] opD;
    logic [2:0] funct3D;
    logic       funct7b5D;
    logic [1:0] ImmSrcD;
    logic       ZeroE;
    logic       PCSrcE;
    logic [2:0] ALUControlE;
    logic       ALUSrcE;
    logic       ResultSrcEb0;
    logic       RegWriteM;
    logic [1:0] ResultSrcW;
    logic       RegWriteW;

    logic [1:0] ForwardAE, ForwardBE;
    logic       StallF, StallD, FlushD, FlushE;

```

```

logic [4:0] Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW;

controller c(clk, reset,
              opD, funct3D, funct7b5D, ImmSrcD,
              FlushE, ZeroE, PCSrcE, ALUControlE, ALUSrcE, ResultSrcEb0,
              MemWriteM, RegWriteM,
              RegWriteW, ResultSrcW);

datapath dp(clk, reset,
            StallF, PCF, InstrF,
            opD, funct3D, funct7b5D, StallD, FlushD, ImmSrcD,
            FlushE, ForwardAE, ForwardBE, PCSrcE, ALUControlE,
            ALUSrcE, ZeroE,
            MemWriteM, WriteDataM, ALUResultM, ReadDataM,
            RegWriteW, ResultSrcW,
            Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW);

hazard hu(Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW,
           PCSrcE, ResultSrcEb0, RegWriteM, RegWriteW,
           ForwardAE, ForwardBE, StallF, StallD, FlushD, FlushE);

endmodule

module controller(input logic clk, reset,
                  // Decode stage control signals
                  input logic [6:0] opD,
                  input logic [2:0] funct3D,
                  input logic funct7b5D,
                  output logic [1:0] ImmSrcD,
                  // Execute stage control signals
                  input logic FlushE,
                  input logic ZeroE,
                  output logic PCSrcE, // for datapath and
Hazard Unit
                  output logic [2:0] ALUControlE,
                  output logic ALUSrcE,
                  output logic ResultSrcEb0, // for Hazard Unit
                  // Memory stage control signals
                  output logic MemWriteM,
                  output logic RegWriteM, // for Hazard Unit

                  // Writeback stage control signals
                  output logic RegWriteW, // for datapath and
Hazard Unit
                  output logic [1:0] ResultSrcW);

// pipelined control signals
logic RegWriteD, RegWriteE;
logic [1:0] ResultSrcD, ResultSrcE, ResultSrcM;
logic MemWriteD, MemWriteE;
logic JumpD, JumpE;
logic BranchD, BranchE;
logic [1:0] ALUOpD;
logic [2:0] ALUControlD;
logic ALUSrcD;

```

```

// Decode stage logic
maindec md(opD, ResultSrcD, MemWriteD, BranchD,
           ALUSrcD, RegWrittenD, JumpD, ImmSrcD, ALUOpD);
aludec ad(opD[5], funct3D, funct7b5D, ALUOpD, ALUControlD);

// Execute stage pipeline control register and logic
flopvc #(10) controlregE(clk, reset, FlushE,
                           {RegWrittenD, ResultSrcD, MemWrittenD, JumpD, BranchD,
                           ALUControlD, ALUSrcD},
                           {RegWrittenE, ResultSrcE, MemWrittenE, JumpE, BranchE,
                           ALUControlE, ALUSrcE});

assign PCSrcE = (BranchE & ZeroE) | JumpE;
assign ResultSrcEb0 = ResultSrcE[0];

// Memory stage pipeline control register
flopv #(4) controlregM(clk, reset,
                        {RegWrittenE, ResultSrcE, MemWrittenE},
                        {RegWrittenM, ResultSrcM, MemWrittenM});

// Writeback stage pipeline control register
flopv #(3) controlregW(clk, reset,
                        {RegWrittenM, ResultSrcM},
                        {RegWrittenW, ResultSrcW});
endmodule

module maindec(input  logic [6:0] op,
               output logic [1:0] ResultSrc,
               output logic      MemWrite,
               output logic      Branch, ALUSrc,
               output logic      RegWrite, Jump,
               output logic [1:0] ImmSrc,
               output logic [1:0] ALUOp);

logic [10:0] controls;

assign {RegWrite, ImmSrc, ALUSrc, MemWrite,
        ResultSrc, Branch, ALUOp, Jump} = controls;

always_comb
  case(op)
    // RegWrite_ImmSrc_ALUSrc_MemWrite_ResultSrc_Branch_ALUOp_Jump
    7'b0000011: controls = 11'b1_00_1_0_01_0_00_0; // lw
    7'b0100011: controls = 11'b0_01_1_1_00_0_00_0; // sw
    7'b0110011: controls = 11'b1_xx_0_0_00_0_10_0; // R-type
    7'b1100011: controls = 11'b0_10_0_0_00_1_01_0; // beq
    7'b0010011: controls = 11'b1_00_1_0_00_0_10_0; // I-type ALU
    7'b1101111: controls = 11'b1_11_0_0_10_0_00_1; // jal
    7'b0000000: controls = 11'b0_00_0_0_00_0_00_0; // need valid values at
reset
    default:     controls = 11'bx_xx_x_x_xx_x_xx_x; // non-implemented
instruction
  endcase
endmodule

module aludec(input  logic          opb5,

```

```

        input  logic [2:0] funct3,
        input  logic          funct7b5,
        input  logic [1:0] ALUOp,
        output logic [2:0] ALUControl);

logic RtypeSub;
assign RtypeSub = funct7b5 & opb5; // TRUE for R-type subtract instruction

always_comb
case(ALUOp)
    2'b00:           ALUControl = 3'b000; // addition
    2'b01:           ALUControl = 3'b001; // subtraction
    default: case(funct3) // R-type or I-type ALU
        3'b000: if (RtypeSub)
                    ALUControl = 3'b001; // sub
                else
                    ALUControl = 3'b000; // add, addi
        3'b010:   ALUControl = 3'b101; // slt, slti
        3'b110:   ALUControl = 3'b011; // or, ori
        3'b111:   ALUControl = 3'b010; // and, andi
        default:  ALUControl = 3'bxxxx; // ???
    endcase
endcase
endmodule

module datapath(input logic clk, reset,
                // Fetch stage signals
                input  logic          StallF,
                output logic [31:0] PCF,
                input  logic [31:0] InstrF,
                // Decode stage signals
                output logic [6:0]  opD,
                output logic [2:0]  funct3D,
                output logic          funct7b5D,
                input  logic          StallD, FlushD,
                input  logic [1:0]  ImmSrcD,
                // Execute stage signals
                input  logic          FlushE,
                input  logic [1:0]  ForwardAE, ForwardBE,
                input  logic          PCSrcE,
                input  logic [2:0]  ALUControle,
                input  logic          ALUSrcE,
                output logic          ZeroE,
                // Memory stage signals
                input  logic          MemWriteM,
                output logic [31:0] WriteDataM, ALUResultM,
                input  logic [31:0] ReadDataM,
                // Writeback stage signals
                input  logic          RegWriteW,
                input  logic [1:0]  ResultSrcW,
                // Hazard Unit signals
                output logic [4:0] Rs1D, Rs2D, Rs1E, Rs2E,
                output logic [4:0] RdE, RdM, RdW);

// Fetch stage signals
logic [31:0] PCNextF, PCPlus4F;
// Decode stage signals

```

```

logic [31:0] InstrD;
logic [31:0] PCD, PCPlus4D;
logic [31:0] RD1D, RD2D;
logic [31:0] ImmExtD;
logic [4:0] RdD;
// Execute stage signals
logic [31:0] RD1E, RD2E;
logic [31:0] PCE, ImmExtE;
logic [31:0] SrcAE, SrcBE;
logic [31:0] ALUResultE;
logic [31:0] WriteDataE;
logic [31:0] PCPlus4E;
logic [31:0] PCTargetE;
// Memory stage signals
logic [31:0] PCPlus4M;
// Writeback stage signals
logic [31:0] ALUResultW;
logic [31:0] ReadDataW;
logic [31:0] PCPlus4W;
logic [31:0] ResultW;

// Fetch stage pipeline register and logic
mux2 #(32) pcmux(PCPlus4F, PCTargetE, PCSrcE, PCNextF);
flopnr #(32) pcreg(clk, reset, ~StallF, PCNextF, PCF);
adder pcadd(PCF, 32'h4, PCPlus4F);

// Decode stage pipeline register and logic
flopncr #(96) regD(clk, reset, FlushD, ~StallD,
                     {InstrF, PCF, PCPlus4F},
                     {InstrD, PCD, PCPlus4D});
assign opD = InstrD[6:0];
assign funct3D = InstrD[14:12];
assign funct7b5D = InstrD[30];
assign Rs1D = InstrD[19:15];
assign Rs2D = InstrD[24:20];
assign RdD = InstrD[11:7];

regfile rf(clk, RegWriteW, Rs1D, Rs2D, RdW, ResultW, RD1D, RD2D);
extend ext(InstrD[31:7], ImmSrcD, ImmExtD);

// Execute stage pipeline register and logic
floprc #(175) regE(clk, reset, FlushE,
                     {RD1D, RD2D, PCD, Rs1D, Rs2D, RdD, ImmExtD, PCPlus4D},
                     {RD1E, RD2E, PCE, Rs1E, Rs2E, RdE, ImmExtE, PCPlus4E});

mux3 #(32) faemux(RD1E, ResultW, ALUResultM, ForwardAE, SrcAE);
mux3 #(32) fbemux(RD2E, ResultW, ALUResultM, ForwardBE, WriteDataE);
mux2 #(32) srcbmux(WriteDataE, ImmExtE, ALUSrcE, SrcBE);
alu alu(SrcAE, SrcBE, ALUControlE, ALUResultE, ZeroE);
adder branchadd(ImmExtE, PCE, PCTargetE);

// Memory stage pipeline register
floprr #(101) regM(clk, reset,
                     {ALUResultE, WriteDataE, RdE, PCPlus4E},
                     {ALUResultM, WriteDataM, RdM, PCPlus4M});

// Writeback stage pipeline register and logic

```

```

flop#(101) regW(clk, reset,
                  {ALUResultM, ReadDataM, RdM, PCPlus4M},
                  {ALUResultW, ReadDataW, RdW, PCPlus4W});
mux3 #(32) resultmux(ALUResultW, ReadDataW, PCPlus4W, ResultSrcW,
ResultW);
endmodule

// Hazard Unit: forward, stall, and flush
module hazard(input logic [4:0] Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW,
               input logic PCSrcE, ResultSrcEb0,
               input logic RegWriteM, RegWriteW,
               output logic [1:0] ForwardAE, ForwardBE,
               output logic StallF, StallD, FlushD, FlushE);

logic lwStallD;

// forwarding logic
always_comb begin
    ForwardAE = 2'b00;
    ForwardBE = 2'b00;
    if (Rs1E != 5'b0)
        if ((Rs1E == RdM) & RegWriteM) ForwardAE = 2'b10;
        else if ((Rs1E == RdW) & RegWriteW) ForwardAE = 2'b01;

    if (Rs2E != 5'b0)
        if ((Rs2E == RdM) & RegWriteM) ForwardBE = 2'b10;
        else if ((Rs2E == RdW) & RegWriteW) ForwardBE = 2'b01;
end

// stalls and flushes
assign lwStallD = ResultSrcEb0 & ((Rs1D == RdE) | (Rs2D == RdE));
assign StallD = lwStallD;
assign StallF = lwStallD;
assign FlushD = PCSrcE;
assign FlushE = lwStallD | PCSrcE;
endmodule

module regfile(input logic clk,
                input logic we3,
                input logic [4:0] a1, a2, a3,
                input logic [31:0] wd3,
                output logic [31:0] rd1, rd2);

logic [31:0] rf[31:0];

// three ported register file
// read two ports combinationally (A1/RD1, A2/RD2)
// write third port on rising edge of clock (A3/WD3/WE3)
// write occurs on falling edge of clock
// register 0 hardwired to 0

always_ff @(negedge clk)
    if (we3) rf[a3] <= wd3;

assign rd1 = (a1 != 0) ? rf[a1] : 0;
assign rd2 = (a2 != 0) ? rf[a2] : 0;
endmodule

```

```

module adder(input [31:0] a, b,
             output [31:0] y);

    assign y = a + b;
endmodule

module extend(input logic [31:7] instr,
              input logic [1:0] immsrc,
              output logic [31:0] immext);

    always_comb
        case(immsrc)
            // I-type
            2'b00: immext = {{20{instr[31]}}, instr[31:20]};
            // S-type (stores)
            2'b01: immext = {{20{instr[31]}}, instr[31:25], instr[11:7]};
            // B-type (branches)
            2'b10: immext = {{20{instr[31]}}, instr[7], instr[30:25],
instr[11:8], 1'b0};
            // J-type (jal)
            2'b11: immext = {{12{instr[31]}}, instr[19:12], instr[20],
instr[30:21], 1'b0};
            default: immext = 32'bx; // undefined
        endcase
    endmodule

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

    always_ff @(posedge clk, posedge reset)
        if (reset) q <= 0;
        else q <= d;
    endmodule

module fopenr #(parameter WIDTH = 8)
    (input logic clk, reset, en,
     input logic [WIDTH-1:0] d,
     output logic [WIDTH-1:0] q);

    always_ff @(posedge clk, posedge reset)
        if (reset) q <= 0;
        else if (en) q <= d;
    endmodule

module fopenrc #(parameter WIDTH = 8)
    (input logic clk, reset, clear, en,
     input logic [WIDTH-1:0] d,
     output logic [WIDTH-1:0] q);

    always_ff @(posedge clk, posedge reset)
        if (reset) q <= 0;
        else if (en)
            if (clear) q <= 0;
            else q <= d;
    endmodule

```

```

endmodule

module flopvc #(parameter WIDTH = 8)
    (input  logic clk,
     input  logic reset,
     input  logic clear,
     input  logic [WIDTH-1:0] d,
     output logic [WIDTH-1:0] q);

    always_ff @(posedge clk, posedge reset)
        if (reset) q <= 0;
        else
            if (clear) q <= 0;
            else       q <= d;
endmodule

module mux2 #(parameter WIDTH = 8)
    (input  logic [WIDTH-1:0] d0, d1,
     input  logic             s,
     output logic [WIDTH-1:0] y);

    assign y = s ? d1 : d0;
endmodule

module mux3 #(parameter WIDTH = 8)
    (input  logic [WIDTH-1:0] d0, d1, d2,
     input  logic [1:0]         s,
     output logic [WIDTH-1:0] y);

    assign y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule

module imem(input  logic [31:0] a,
            output logic [31:0] rd);

    logic [31:0] RAM[63:0];

    initial
        $readmemh("riscvtest.txt",RAM);

    assign rd = RAM[a[31:2]]; // word aligned
endmodule

module dmem(input  logic      clk, we,
            input  logic [31:0] a, wd,
            output logic [31:0] rd);

    logic [31:0] RAM[63:0];

    assign rd = RAM[a[31:2]]; // word aligned

    always_ff @(posedge clk)
        if (we) RAM[a[31:2]] <= wd;
endmodule

module alu(input  logic [31:0] a, b,
            input  logic [2:0]   alucontrol,

```

```

        output logic [31:0] result,
        output logic          zero);

logic [31:0] condinvb, sum;
logic      v;           // overflow
logic      isAddSub;    // true when is add or subtract operation

assign condinvb = alucontrol[0] ? ~b : b;
assign sum = a + condinvb + alucontrol[0];
assign isAddSub = ~alucontrol[2] & ~alucontrol[1] |
                  ~alucontrol[1] & alucontrol[0];

always_comb
  case (alucontrol)
    3'b000: result = sum;          // add
    3'b001: result = sum;          // subtract
    3'b010: result = a & b;        // and
    3'b011: result = a | b;        // or
    3'b100: result = a ^ b;        // xor
    3'b101: result = sum[31] ^ v;  // slt
    3'b110: result = a << b[4:0]; // sll
    3'b111: result = a >> b[4:0]; // srl
    default: result = 32'bx;
  endcase

  assign zero = (result == 32'b0);
  assign v = ~(alucontrol[0] ^ a[31] ^ b[31]) & (a[31] ^ sum[31]) & isAddSub;
endmodule

```

Exercise 7.27

RISC-V multicycle processor

Enhanced to support all instructions from **Exercise 7.13**:

xor, sll, srl, bne

```

module testbench();
  logic      clk;
  logic      reset;

  logic [31:0] WriteData, DataAdr;
  logic      MemWrite;

  // instantiate device to be tested
  top dut(clk, reset, WriteData, DataAdr, MemWrite);

  // initialize test
  initial
    begin
      reset <= 1; # 22; reset <= 0;
    end

  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
    end

```

```

    end

    // check results
    always @ (negedge clk)
    begin
        if (MemWrite) begin
            if (DataAddr === 216 & WriteData === 4140) begin
                $display("Simulation succeeded");
                $stop;
            end
        end
    end
endmodule

module top (input logic      clk, reset,
            output logic [31:0] WriteData, DataAddr,
            output logic          MemWrite);

    logic [31:0] ReadData;

    // instantiate processor and memories
    riscvmulti rvmulti (clk, reset, MemWrite, DataAddr,
                         WriteData, ReadData);
    mem mem (clk, MemWrite, DataAddr, WriteData, ReadData);
endmodule

module riscvmulti (input logic      clk, reset,
                    output logic      MemWrite,
                    output logic [31:0] Adr, WriteData,
                    input logic [31:0] ReadData);

    logic      RegWrite, jump;
    logic [1:0] ResultSrc;
    logic [2:0] ImmSrc;           // expand to 3-bits for lui and auipc
    logic [3:0] ALUControl;
    logic      PCWrite;
    logic      IRWrite;
    logic [1:0] ALUSrcA;
    logic [1:0] ALUSrcB;
    logic      AdrSrc;
    logic [3:0] Flags;           // added for other branches
    logic [6:0] op;
    logic [2:0] funct3;
    logic      funct7b5;
    logic      LoadType;          // added for lbu
    logic      StoreType;         // added for sb
    logic      PCTargetSrc;       // added for jalr

    controller c (clk, reset, op, funct3, funct7b5, Flags,
                  ImmSrc, ALUSrcA, ALUSrcB,
                  ResultSrc, AdrSrc, ALUControl,
                  IRWrite, PCWrite, RegWrite, MemWrite,
                  LoadType, StoreType,                      // lbu, sb
                  PCTargetSrc);                            // jalr

    datapath dp (clk, reset,
                 ImmSrc, ALUSrcA, ALUSrcB,

```

```

        ResultSrc, AdrSrc, IRWrite, PCWrite,
        RegWrite, MemWrite, ALUControl,
        LoadType, StoreType, PCTargetSrc,
        op, funct3,
        funct7b5, Flags, Adr, ReadData, WriteData);
endmodule

module controller(input logic      clk,
                  input logic      reset,
                  input logic [6:0] op,
                  input logic [2:0] funct3,
                  input logic      funct7b5,
                  input logic [3:0] Flags,
                  output logic [2:0] ImmSrc,
                  output logic [1:0] ALUSrcA, ALUSrcB,
                  output logic [1:0] ResultSrc,
                  output logic      AdrSrc,
                  output logic [3:0] ALUControl,
                  output logic      IRWrite, PCWrite,
                  output logic      RegWrite, MemWrite,
                  output logic      LoadType, // lbu
                  output logic      StoreType, // sb
                  output logic      PCTargetSrc); // jalr

logic [1:0] ALUOp;
logic      Branch, PCUpdate;
logic      branchtaken; // added for other branches

// Main FSM
mainfsm fsm(clk, reset, op,
             ALUSrcA, ALUSrcB, ResultSrc, AdrSrc,
             IRWrite, PCUpdate, RegWrite, MemWrite,
             ALUOp, Branch);

// ALU Decoder
aludec ad(op[5], funct3, funct7b5, ALUOp, ALUControl);

// Instruction Decoder
instrdec id(op, ImmSrc);

// Branch logic
lsu      lsu(funct3, LoadType, StoreType);
bu       branchunit/Branch, Flags, funct3, branchtaken); // added for bne,
blt, etc.

assign PCWrite = branchtaken | PCUpdate;

endmodule

module mainfsm(input logic      clk,
               input logic      reset,
               input logic [6:0] op,
               output logic [1:0] ALUSrcA, ALUSrcB,
               output logic [1:0] ResultSrc,
               output logic      AdrSrc,
               output logic      IRWrite, PCUpdate,
               output logic      RegWrite, MemWrite,

```

```

        output logic [1:0]    ALUOp,
        output logic          Branch);

typedef enum logic [3:0] {FETCH, DECODE, MEMADR, MEMREAD, MEMWB, MEMWRITE,
                        EXECUTER, EXECUTEI, ALUWB,
                        BEQ, JAL, UNKNOWN} statetype;

statetype state, nextstate;
logic [14:0] controls;

// state register
always @(posedge clk or posedge reset)
  if (reset) state <= FETCH;
  else state <= nextstate;

// next state logic
always_comb
  case(state)
    FETCH:                           nextstate = DECODE;
    DECODE:  casez (op)
      7'b0?00011:      nextstate = MEMADR;      // lw or sw
      7'b0110011:      nextstate = EXECUTER;     // R-type
      7'b0010011:      nextstate = EXECUTEI;     // addi
      7'b1100011:      nextstate = BEQ;          // beq
      7'b1101111:      nextstate = JAL;          // jal
      default:          nextstate = UNKNOWN;
    endcase
    MEMADR:
      if (op[5])
        nextstate = MEMWRITE; // sw
      else
        nextstate = MEMREAD; // lw
    MEMREAD:
    EXECUTER:
    EXECUTEI:
    JAL:
    default:
      nextstate = FETCH;
  endcase

// state-dependent output logic
always_comb
  case(state)
    FETCH:      controls = 15'b00_10_10_0_1100_00_0;
    DECODE:     controls = 15'b01_01_00_0_0000_00_0;
    MEMADR:     controls = 15'b10_01_00_0_0000_00_0;
    MEMREAD:    controls = 15'b00_00_00_1_0000_00_0;
    MEMWRITE:   controls = 15'b00_00_00_1_0001_00_0;
    MEMWB:      controls = 15'b00_00_01_0_0010_00_0;
    EXECUTER:   controls = 15'b10_00_00_0_0000_10_0;
    EXECUTEI:   controls = 15'b10_01_00_0_0000_10_0;
    ALUWB:      controls = 15'b00_00_00_0_0010_00_0;
    BEQ:        controls = 15'b10_00_00_0_0000_01_1;
    JAL:        controls = 15'b01_10_00_0_0100_00_0;
    default:    controls = 15'bxx_xx_xx_x_xxxx_xx_x;
  endcase

assign {ALUSrcA, ALUSrcB, ResultSrc, AdrSrc, IRWrite, PCUpdate,
RegWrite, MemWrite, ALUOp, Branch} = controls;

```

```

endmodule

module aludec(input logic      opb5,
               input logic [2:0] funct3,
               input logic      funct7b5,
               input logic [1:0] ALUOp,
               output logic [3:0] ALUControl); // expand to 4 bits for sra

logic RtypeSub;
assign RtypeSub = funct7b5 & opb5; // TRUE for R-type subtract instruction

always_comb
case(ALUOp)
  2'b00:           ALUControl = 4'b000; // addition
  2'b01:           ALUControl = 4'b001; // subtraction
  default: case(funct3) // R-type or I-type ALU
    3'b000: if (RtypeSub)
              ALUControl = 4'b0001; // sub
            else
              ALUControl = 4'b0000; // add, addi
    3'b001:   ALUControl = 4'b0110; // sll, slli
    3'b010:   ALUControl = 4'b0101; // slt, slti
    3'b100:   ALUControl = 4'b0100; // xor, xori
    3'b101: if (funct7b5)
              ALUControl = 4'b1000; // sra, srai
            else
              ALUControl = 4'b0111; // srl, srli
    3'b110:   ALUControl = 4'b0011; // or, ori
    3'b111:   ALUControl = 4'b0010; // and, andi
    default:  ALUControl = 4'bxxxx; // ???
  endcase
endcase
endmodule

module instrdec (input logic [6:0] op,
                 output logic [2:0] ImmSrc);
  always_comb
  case(op)
    7'b0110011: ImmSrc = 3'bxxx; // R-type
    7'b0010011: ImmSrc = 3'b000; // I-type ALU
    7'b0000011: ImmSrc = 3'b000; // lw / lbu
    7'b0100011: ImmSrc = 3'b001; // sw / sb
    7'b1100011: ImmSrc = 3'b010; // branches
    7'b1101111: ImmSrc = 3'b011; // jal
    7'b0110111: ImmSrc = 3'b100; // lui
    7'b1100111: ImmSrc = 3'b000; // jalr
    7'b0010111: ImmSrc = 3'b100; // auipc
    default:    ImmSrc = 3'bxxx; // ???
  endcase
endmodule

module datapath(input logic      clk, reset,
                input logic [2:0] ImmSrc,
                input logic [1:0] ALUSrcA, ALUSrcB,
                input logic [1:0] ResultSrc,
                                input logic      AdrSrc,
                input logic      IRWrite, PCWrite,

```

```

        input  logic      RegWrite, MemWrite,
        input  logic [3:0] alucontrol,
        input  logic      LoadType, StoreType, // lbu, sb
        input  logic      PCTargetSrc,
        output logic [6:0] op,
        output logic [2:0] funct3,
        output logic      funct7b5,
        output logic [3:0] Flags,
        output logic [31:0] Adr,
        input  logic [31:0] ReadData,
        output logic [31:0] WriteData);

logic [31:0] PC, OldPC, Instr, immext, ALUResult;
logic [31:0] SrcA, SrcB, RD1, RD2, A;
logic [31:0] Result, Data, ALUOut;

// next PC logic
fopenr #(32) pcreg(clk, reset, PCWrite, Result, PC);
fopenr #(32) oldpcreg(clk, reset, IRWrite, PC, OldPC);

// memory logic
mux2   #(32) adrmux(PC, Result, AdrSrc, Adr);
fopenr #(32) ir(clk, reset, IRWrite, ReadData, Instr);
flopr  #(32) datareg(clk, reset, ReadData, Data);

// register file logic
regfile rf(clk, RegWrite, Instr[19:15], Instr[24:20],
           Instr[11:7], Result, RD1, RD2);
extend  ext(Instr[31:7], ImmSrc, immext);
flopr  #(32) srcareg(clk, reset, RD1, A);
flopr  #(32) wdreg(clk, reset, RD2, WriteData);

// ALU logic
mux3   #(32) srcamux(PC, OldPC, A, ALUSrcA, SrcA);
mux3   #(32) srcbmux(WriteData, immext, 32'd4, ALUSrcB, SrcB);
alu     alu(SrcA, SrcB, alucontrol, ALUResult, Flags);
flopr  #(32) aluoutreg(clk, reset, ALUResult, ALUOut);
mux3   #(32) resmux(ALUOut, Data, ALUResult, ResultSrc, Result);

// outputs to control unit
assign op      = Instr[6:0];
assign funct3  = Instr[14:12];
assign funct7b5 = Instr[30];

endmodule

module regfile(input  logic      clk,
                input  logic      we3,
                input  logic [4:0] a1, a2, a3,
                input  logic [31:0] wd3,
                output logic [31:0] rd1, rd2);

logic [31:0] rf[31:0];

// three ported register file

```

```

// read two ports combinationally (A1/RD1, A2/RD2)
// write third port on rising edge of clock (A3/WD3/WE3)
// register 0 hardwired to 0

always_ff @(posedge clk)
  if (we3) rf[a3] <= wd3;

assign rd1 = (a1 != 0) ? rf[a1] : 0;
assign rd2 = (a2 != 0) ? rf[a2] : 0;
endmodule

module adder(input [31:0] a, b,
              output [31:0] y);

  assign y = a + b;
endmodule

module extend(input logic [31:7] instr,
              input logic [2:0] immsrc, // extended to 3 bits for lui
              output logic [31:0] immext);

  always_comb
    case(immsrc)
      // I-type
      3'b000: immext = {{20{instr[31]}}, instr[31:20]};
      // S-type (stores)
      3'b001: immext = {{20{instr[31]}}, instr[31:25], instr[11:7]};
      // B-type (branches)
      3'b010: immext = {{20{instr[31]}}, instr[7], instr[30:25],
instr[11:8], 1'b0};
      // J-type (jal)
      3'b011: immext = {{12{instr[31]}}, instr[19:12], instr[20],
instr[30:21], 1'b0};
      // U-type (lui, auipc)
      3'b100: immext = {instr[31:12], 12'b0};
      default: immext = 32'bx; // undefined
    endcase
  endmodule

  // zeroextend module added for lbu
  module zeroextend(input logic [7:0] a,
                     output logic [31:0] zeroimmext);

    assign zeroimmext = {24'b0, a};
  endmodule

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

    always_ff @(posedge clk, posedge reset)
      if (reset) q <= 0;
      else q <= d;
  endmodule

```

```

module fopenr #(parameter WIDTH = 8)
    (input logic clk, reset, en,
     input logic [WIDTH-1:0] d,
     output logic [WIDTH-1:0] q);

    always_ff @(posedge clk, posedge reset)
        if (reset) q <= 0;
        else if (en) q <= d;
    endmodule

module mux2 #(parameter WIDTH = 8)
    (input logic [WIDTH-1:0] d0, d1,
     input logic s,
     output logic [WIDTH-1:0] y);

    assign y = s ? d1 : d0;
endmodule

module mux3 #(parameter WIDTH = 8)
    (input logic [WIDTH-1:0] d0, d1, d2,
     input logic [1:0] s,
     output logic [WIDTH-1:0] y);

    assign y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule

module mux4 #(parameter WIDTH = 8)
    (input logic [WIDTH-1:0] d0, d1, d2, d3,
     input logic [1:0] s,
     output logic [WIDTH-1:0] y);

    assign y = s[1] ? (s[0] ? d3: d2) : (s[0] ? d1 : d0);
endmodule

module mem(input logic clk, we,
            input logic [31:0] a, wd,
            output logic [31:0] rd);

    logic [31:0] RAM[127:0]; // increased size of memory

    initial
        $readmemh("example.txt",RAM);

    assign rd = RAM[a[31:2]]; // word aligned

    always_ff @(posedge clk)
        if (we) RAM[a[31:2]] <= wd;
    endmodule

module alu(input logic [31:0] a,
            input logic [31:0] b,
            input logic [3:0] alucontrol, // expanded to 4 bits for sra
            output logic [31:0] result,
            output logic [3:0] flags); // added for blt and other
branches

    logic [31:0] condinvb, sum;

```

```

logic      v, c, n, z;      // flags: overflow, carry out, negative, zero
logic      cout;           // carry out of adder
logic      isAdd;          // true if is an add operation
logic      isSub;          // true if is a subtract operation

assign flags = {v, c, n, z};
assign condinvb = alucontrol[0] ? ~b : b;
assign {cout, sum} = a + condinvb + alucontrol[0];
assign isAddSub = ~alucontrol[3] & ~alucontrol[2] & ~alucontrol[1] |  

                  ~alucontrol[3] & ~alucontrol[1] & alucontrol[0];

always_comb
case (alucontrol)
  4'b0000: result = sum;           // add
  4'b0001: result = sum;           // subtract
  4'b0010: result = a & b;         // and
  4'b0011: result = a | b;         // or
  4'b0100: result = a ^ b;         // xor
  4'b0101: result = sum[31] ^ v;    // slt
  4'b0110: result = a << b[4:0];   // sll
  4'b0111: result = a >> b[4:0];   // srl
  4'b1000: result = $signed(a) >>> b[4:0]; // sra
  default: result = 32'bx;
endcase

// added for blt and other branches
assign z = (result == 32'b0);
assign n = result[31];
assign c = cout & isAddSub;
assign v = ~(alucontrol[0] ^ a[31] ^ b[31]) & (a[31] ^ sum[31]) & isAddSub;
endmodule

// Load/store Unit (lsu) added for lbu
module lsu(input logic [2:0] funct3,
            output logic      LoadType, StoreType);
  always_comb
    case(funct3)
      3'b000: {LoadType, StoreType} = 2'b01;
      3'b010: {LoadType, StoreType} = 2'b00;
      3'b100: {LoadType, StoreType} = 2'b1x;
      default: {LoadType, StoreType} = 2'bxx;
    endcase
  endmodule

// Branch Unit (bu) added for bne, blt, bltu, bge, bgeu
module bu (input logic      Branch,
            input logic [3:0] Flags,
            input logic [2:0] funct3,
            output logic      taken);
  logic v, c, n, z;      // Flags: overflow, carry out, negative, zero
  logic cond;             // cond is 1 when condition for branch met
  assign {v, c, n, z} = Flags;
  assign taken = cond & Branch;

  always_comb
    case (funct3)
      3'b000: cond = z;        // beq

```

000x | 0x01

```

3'b001: cond = ~z;      // bne
3'b100: cond = (n ^ v); // blt
3'b101: cond = ~(n ^ v); // bge
3'b110: cond = ~c;      // bltu
3'b111: cond = c;       // bgeu
default: cond = 1'b0;
endcase
endmodule

```

Test Program:

Use the same test program as shown in the solutions for Exercise 7.9.

Exercise 7.29

The code executes as follows:

Pipeline stages

Cycle	Fetch	Decode	Execute	Memory	Writeback
1	xor				
2	addi	xor			
3	lw	addi	xor		
4	sw	lw	addi	xor	
5	or	sw	lw	addi	xor

In cycle 5, the sw instruction is in the Decode stage and xor is in the Writeback stage. So, s1 is written (by xor) during the first half of cycle 5. s1 and s4 are read (by the sw) during the second half of cycle 5.

Notice that s1 is both written and read in cycle 5. Also note that no hazards exist in this code.

Exercise 7.31

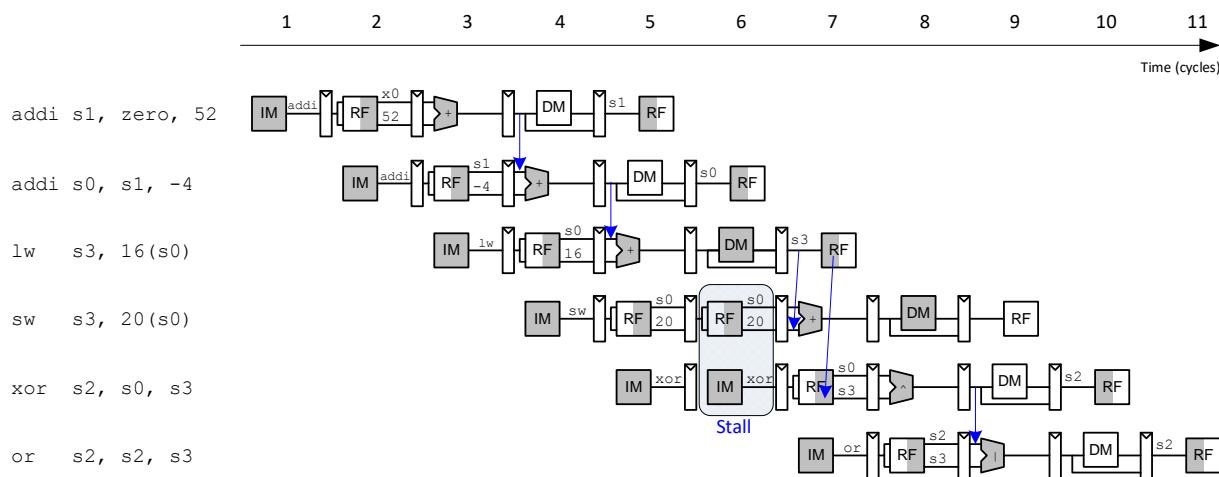
The code executes as follows:

Pipeline stages

Cycle	Fetch	Decode	Execute	Memory	Writeback
1	addi s1,x0,52				
2	addi s0,s1,-4	addi s1,x0,52			
3	lw s3,16(s0)	addi s0,s1,-4	addi s1,x0,52		
4	sw s3,20(s0)	lw s3,16(s0)	addi s0,s1,-4	addi s1,x0,52	
5	xor s2,s0,s3	sw s3,20(s0)	lw s3,16(s0)	addi s0,s1,-4	addi s1,x0,52
6	xor s2,s0,s3	sw s3,20(s0)	bubble	lw s3,16(s0)	addi s0,s1,-4

In cycle 5, **s1** is being written (by addi) in the Decode stage, and **s0** and **s3** are being read by sw in the Decode stage. Note that in this cycle, sw detects that it needs to stall in the next cycle – because a lw is in the Execute stage that will produce one of its source registers (s3), and the lw won't have that operand ready until the end of the Memory stage.

Exercise 7.33



Exercise 7.35

It takes **7 clock cycles** to issue all the instructions.

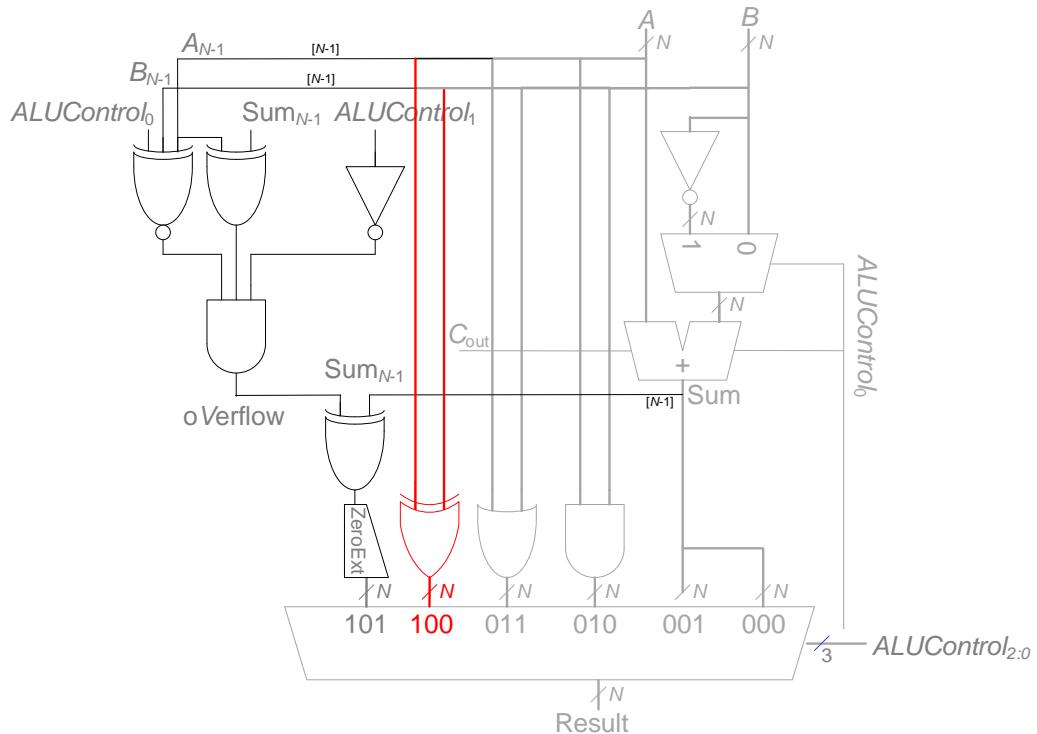
instructions = 6.

CPI = 7 clock cycles / 6 instructions = **1.17 CPI**.

Exercise 7.37

The datapath already supports R-type instructions. So no changes need to be made to the datapath. Only the ALU needs to be modified: we add another input to the multiplexer and N 2-bit XOR gates within the ALU. We also update the ALU Decoder truth table / logic. The Main Decoder truth table need not be updated because it already supports R-type instructions. These changes are shown below.

Modified ALU to support xor



Modified ALU operations to support xor

$ALUControl_{2:0}$	Function
000	add
001	subtract
010	and
011	or
100	xor
101	SLT

Modified ALU Decoder truth table to support xor

$ALUOp$	$funct3$	$ops, funct7_s$	$ALUControl$	Instruction
00	x	x	000 (add)	lw, sw
01	x	x	001 (subtract)	beq
10	000	00, 01, 10	000 (add)	add, addi
	000	11	001 (subtract)	sub
	010	x	101 (set less than)	slt, slti
	100	x	100 (xor)	xor, xori
	110	x	011 (or)	or, ori
	111	x	010 (and)	and, andi

Exercise 7.39

From Equation 7.5:

$$T_{c3} = \max((40 + 200 + 50), (2(100 + 50)), (40 + 4(30) + 120 + 20 + 50), (40 + 200 + 50), (2(40 + 30 + 60))) = \max(290, 300, 350, 290, 260)$$

The slowest stage is the Execute stage at 350ps. The next slowest stage is the Decode stage at 300ps. Thus, the execute stage should be reduced by 50 ps to make it as fast as the next slowest stage, Decode. She should, thus reduce the ALU delay to: $120 - 50 = 70\text{ps}$.

The new cycle time is then **300 ps**.

Exercise 7.41

Loads take one clock cycle when there is no dependency and seven clock cycles when there is (load plus 6 stalls), so they have an average CPI of $(0.5)(1) + (0.5)(7) = 4$.

Branches take one clock cycle when they are predicted properly and two when they are not, so their average CPI is $(0.7)(1) + (0.3)(2) = 1.3$. The remaining instructions have a CPI of 1. Hence, the average CPI for the SPECINT2000 benchmark is:

$$\text{CPI} = 0.25(4) + 0.10(1) + 0.13(1.3) + 0.52(1) = \mathbf{1.79 \text{ CPI}}$$

Execution time = $(100 \times 10^9 \text{ instructions})(1.79 \text{ cycles/instruction})(400 \times 10^{-12} \text{ s/cycle}) = \mathbf{71.6 \text{ seconds}}$

Exercise 7.43

RISC-V Pipelined Processor – modified to support xor.

```
module testbench();
    logic          clk;
    logic          reset;

    logic [31:0] WriteData, DataAdr;
    logic          MemWrite;

    // instantiate device to be tested
    top dut(clk, reset, WriteData, DataAdr, MemWrite);

    // initialize test
    initial
        begin
            reset <= 1; # 22; reset <= 0;
        end

    // generate clock to sequence tests
    always
        begin
            clk <= 1; # 5; clk <= 0; # 5;
        end

    // check results
    always @ (negedge clk)
        begin
```

```

if(MemWrite) begin
    if(DataAddr === 132 & WriteData === 32'hABCDE02E) begin
        $display("Simulation succeeded");
        $stop;
    end
end
endmodule

module top(input logic      clk, reset,
            output logic [31:0] WriteDataM, DataAddrM,
            output logic          MemWriteM);

    logic [31:0] PCF, InstrF, ReadDataM;

    // instantiate processor and memories
    riscv riscv(clk, reset, PCF, InstrF, MemWriteM, DataAddrM,
                 WriteDataM, ReadDataM);
    imem imem(PCF, InstrF);
    dmem dmem(clk, MemWriteM, DataAddrM, WriteDataM, ReadDataM);
endmodule

module riscv(input logic      clk, reset,
              output logic [31:0] PCF,
              input logic [31:0] InstrF,
              output logic          MemWriteM,
              output logic [31:0] ALUResultM, WriteDataM,
              input logic [31:0] ReadDataM);

    logic [6:0] opD;
    logic [2:0] funct3D;
    logic      funct7b5D;
    logic [2:0] ImmSrcD;
    logic      ZeroE;
    logic      PCSrcE;
    logic [2:0] ALUControlE;
    logic      ALUSrcAE, ALUSrcBE;
    logic      ResultSrcEb0;
    logic      RegWriteM;
    logic [1:0] ResultSrcW;
    logic      RegWriteW;

    logic [1:0] ForwardAE, ForwardBE;
    logic      StallF, StallD, FlushD, FlushE;

    logic [4:0] Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW;

    controller c(clk, reset,
                  opD, funct3D, funct7b5D, ImmSrcD,
                  FlushE, ZeroE, PCSrcE, ALUControlE, ALUSrcAE, ALUSrcBE,
                  ResultSrcEb0,
                  MemWriteM, RegWriteM,
                  RegWriteW, ResultSrcW);

    datapath dp(clk, reset,
                StallF, PCF, InstrF,
                opD, funct3D, funct7b5D, StallD, FlushD, ImmSrcD,

```

```

          FlushE, ForwardAE, ForwardBE, PCSrcE, ALUControlE,
ALUSrcAE, ALUSrcBE, ZeroE,
          MemWriteM, WriteDataM, ALUResultM, ReadDataM,
          RegWriteW, ResultSrcW,
          Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW);

hazard  hu(Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW,
          PCSrcE, ResultSrcEb0, RegWriteM, RegWriteW,
          ForwardAE, ForwardBE, StallF, StallD, FlushD, FlushE);

endmodule

module controller(input logic                 clk, reset,
                  // Decode stage control signals
                  input logic [6:0]   opD,
                  input logic [2:0]   funct3D,
                  input logic          funct7b5D,
                  output logic [2:0]  ImmSrcD,
                  // Execute stage control signals
                  input logic          FlushE,
                  input logic          ZeroE,
                  output logic         PCSrcE,           // for datapath and
Hazard Unit
                  output logic [2:0]  ALUControlE,
                  output logic          ALUSrcAE,
                  output logic          ALUSrcBE,        // for lui
                  output logic          ResultSrcEb0, // for Hazard Unit
                  // Memory stage control signals
                  output logic          MemWriteM,
                  output logic          RegWriteM,       // for Hazard Unit

                  // Writeback stage control signals
                  output logic          RegWriteW,      // for datapath and
Hazard Unit
                  output logic [1:0]  ResultSrcW);

// pipelined control signals
logic          RegWriteD, RegWriteE;
logic [1:0]  ResultSrcD, ResultSrcE, ResultSrcM;
logic          MemWriteD, MemWriteE;
logic          JumpD, JumpE;
logic          BranchD, BranchE;
logic [1:0]  ALUOpD;
logic [2:0]  ALUControlD;
logic          ALUSrcAD;
logic          ALUSrcBD; // for lui

// Decode stage logic
maindec md(opD, ResultSrcD, MemWriteD, BranchD,
          ALUSrcAD, ALUSrcBD, RegWriteD, JumpD, ImmSrcD, ALUOpD);
aludec ad(opD[5], funct3D, funct7b5D, ALUOpD, ALUControlD);

// Execute stage pipeline control register and logic
flopvc #(11) controlregE(clk, reset, FlushE,
                         {RegWriteD, ResultSrcD, MemWriteD, JumpD, BranchD,
                          ALUControlD, ALUSrcAD, ALUSrcBD}),

```

```

        {RegWriteE, ResultSrcE, MemWriteE, JumpE, BranchE,
        ALUControle, ALUSrcAE, ALUSrcBE});

assign PCSrcE = (BranchE & ZeroE) | JumpE;
assign ResultSrcEb0 = ResultSrcE[0];

// Memory stage pipeline control register
flop #(4) controlregM(clk, reset,
                      {RegWriteE, ResultSrcE, MemWriteE},
                      {RegWriteM, ResultSrcM, MemWriteM});

// Writeback stage pipeline control register
flop #(3) controlregW(clk, reset,
                      {RegWriteM, ResultSrcM},
                      {RegWriteW, ResultSrcW});

endmodule

module maindec(input logic [6:0] op,
               output logic [1:0] ResultSrc,
               output logic MemWrite,
               output logic Branch,
               output logic ALUSrcA,           // for lui
               output logic ALUSrcB,
               output logic RegWrite, Jump,
               output logic [2:0] ImmSrc,
               output logic [1:0] ALUOp);

logic [12:0] controls;

assign {RegWrite, ImmSrc, ALUSrcA, ALUSrcB, MemWrite,
       ResultSrc, Branch, ALUOp, Jump} = controls;

always_comb
  case(op)
    // RegWrite_ImmSrc_ALUSrcA_ALUSrcB_MemWrite_ResultSrc_Branch_ALUOp_Jump
    7'b0000011: controls = 13'b1_000_0_1_0_01_0_00_0; // lw
    7'b0100011: controls = 13'b0_001_0_1_1_00_0_00_0; // sw
    7'b0110011: controls = 13'b1_xxx_0_0_0_00_0_10_0; // R-type
    7'b1100011: controls = 13'b0_010_0_0_0_00_1_01_0; // beq
    7'b0010011: controls = 13'b1_000_0_1_0_00_0_10_0; // I-type ALU
    7'b1101111: controls = 13'b1_011_0_0_0_10_0_00_1; // jal
    7'b0110111: controls = 13'b1_100_1_1_0_00_0_00_0; // lui
    7'b0000000: controls = 13'b0_000_0_0_0_00_0_00_0; // need valid values
                                                       // at reset
    default:   controls = 13'bx_xxx_x_x_x_xx_x_xx_x; // non-implemented
                                                       // instruction
  endcase
endmodule

module aludec(input logic      opb5,
               input logic [2:0] funct3,
               input logic      funct7b5,
               input logic [1:0] ALUOp,
               output logic [2:0] ALUControl);

logic RtypeSub;
assign RtypeSub = funct7b5 & opb5; // TRUE for R-type subtract instruction

```

```

always_comb
case(ALUOp)
  2'b00:           ALUControl = 3'b000; // addition
  2'b01:           ALUControl = 3'b001; // subtraction
  default: case(func3) // R-type or I-type ALU
    3'b000: if (RtypeSub)
              ALUControl = 3'b001; // sub
            else
              ALUControl = 3'b000; // add, addi
    3'b010:   ALUControl = 3'b101; // slt, slti
    3'b100:   ALUControl = 3'b100; // xor
    3'b110:   ALUControl = 3'b011; // or, ori
    3'b111:   ALUControl = 3'b010; // and, andi
    default:  ALUControl = 3'bxxx; // ???
  endcase
endcase
endmodule

module datapath(input logic clk, reset,
  // Fetch stage signals
  input logic      StallF,
  output logic [31:0] PCF,
  input logic [31:0] InstrF,
  // Decode stage signals
  output logic [6:0] opD,
  output logic [2:0] funct3D,
  output logic      funct7b5D,
  input logic       StallD, FlushD,
  input logic [2:0] ImmSrcD,
  // Execute stage signals
  input logic      FlushE,
  input logic [1:0] ForwardAE, ForwardBE,
  input logic      PCSrcE,
  input logic [2:0] ALUControle,
  input logic      ALUSrcAE,      // needed for lui
  input logic      ALUSrcBE,
  output logic     ZeroE,
  // Memory stage signals
  input logic      MemWriteM,
  output logic [31:0] WriteDataM, ALUResultM,
  input logic [31:0] ReadDataM,
  // Writeback stage signals
  input logic      RegWriteW,
  input logic [1:0] ResultSrcW,
  // Hazard Unit signals
  output logic [4:0] Rs1D, Rs2D, Rs1E, Rs2E,
  output logic [4:0] RdE, RdM, RdW);

  // Fetch stage signals
  logic [31:0] PCNextF, PCPlus4F;
  // Decode stage signals
  logic [31:0] InstrD;
  logic [31:0] PCD, PCPlus4D;
  logic [31:0] RD1D, RD2D;
  logic [31:0] ImmExtD;
  logic [4:0]  RdD;

```

```

// Execute stage signals
logic [31:0] RD1E, RD2E;
logic [31:0] PCE, ImmExtE;
logic [31:0] SrcAE, SrcBE;
logic [31:0] SrcAEforward;
logic [31:0] ALUResultE;
logic [31:0] WriteDataE;
logic [31:0] PCPlus4E;
logic [31:0] PCTargetE;
// Memory stage signals
logic [31:0] PCPlus4M;
// Writeback stage signals
logic [31:0] ALUResultW;
logic [31:0] ReadDataW;
logic [31:0] PCPlus4W;
logic [31:0] ResultW;

// Fetch stage pipeline register and logic
mux2 #(32) pcdux(PCPlus4F, PCTargetE, PCSrcE, PCNextF);
fopenr #(32) pcreg(clk, reset, ~StallF, PCNextF, PCF);
adder pcadd(PCF, 32'h4, PCPlus4F);

// Decode stage pipeline register and logic
fopenrc #(96) regD(clk, reset, FlushD, ~StallD,
                     {InstrF, PCF, PCPlus4F},
                     {InstrD, PCD, PCPlus4D});
assign opD      = InstrD[6:0];
assign funct3D = InstrD[14:12];
assign funct7b5D = InstrD[30];
assign Rs1D     = InstrD[19:15];
assign Rs2D     = InstrD[24:20];
assign RdD      = InstrD[11:7];

regfile rf(clk, RegWriteW, Rs1D, Rs2D, RdW, ResultW, RD1D, RD2D);
extend ext(InstrD[31:7], ImmSrcD, ImmExtD);

// Execute stage pipeline register and logic
flopvc #(175) regE(clk, reset, FlushE,
                     {RD1D, RD2D, PCD, Rs1D, Rs2D, RdD, ImmExtD, PCPlus4D},
                     {RD1E, RD2E, PCE, Rs1E, Rs2E, RdE, ImmExtE, PCPlus4E});

mux3 # (32) faemux(RD1E, ResultW, ALUResultM, ForwardAE, SrcAEforward);
mux2 # (32) srcamux(SrcAEforward, 32'b0, ALUSrcAE, SrcAE); // for lui
mux3 # (32) fbemux(RD2E, ResultW, ALUResultM, ForwardBE, WriteDataE);
mux2 # (32) srcbmux(WriteDataE, ImmExtE, ALUSrcBE, SrcBE);
alu      alu(SrcAE, SrcBE, ALUControlE, ALUResultE, ZeroE);
adder   branchadd(ImmExtE, PCE, PCTargetE);

// Memory stage pipeline register
flopv #(101) regM(clk, reset,
                     {ALUResultE, WriteDataE, RdE, PCPlus4E},
                     {ALUResultM, WriteDataM, RdM, PCPlus4M});

// Writeback stage pipeline register and logic
flopv #(101) regW(clk, reset,
                     {ALUResultM, ReadDataM, RdM, PCPlus4M},
                     {ALUResultW, ReadDataW, RdW, PCPlus4W});

```

```

mux3    #(32) resultmux(ALUResultW, ReadDataW, PCPlus4W, ResultSrcW,
ResultW);
endmodule

// Hazard Unit: forward, stall, and flush
module hazard(input logic [4:0] Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW,
               input logic      PCSrcE, ResultSrcEb0,
               input logic      RegWriteM, RegWriteW,
               output logic [1:0] ForwardAE, ForwardBE,
               output logic      StallF, StallD, FlushD, FlushE);

logic lwStallD;

// forwarding logic
always_comb begin
    ForwardAE = 2'b00;
    ForwardBE = 2'b00;
    if (Rs1E != 5'b0)
        if ((Rs1E == RdM) & RegWriteM) ForwardAE = 2'b10;
        else if ((Rs1E == RdW) & RegWriteW) ForwardAE = 2'b01;

    if (Rs2E != 5'b0)
        if ((Rs2E == RdM) & RegWriteM) ForwardBE = 2'b10;
        else if ((Rs2E == RdW) & RegWriteW) ForwardBE = 2'b01;
end

// stalls and flushes
assign lwStallD = ResultSrcEb0 & ((Rs1D == RdE) | (Rs2D == RdE));
assign StallD = lwStallD;
assign StallF = lwStallD;
assign FlushD = PCSrcE;
assign FlushE = lwStallD | PCSrcE;
endmodule

module regfile(input logic      clk,
                input logic      we3,
                input logic [4:0] a1, a2, a3,
                input logic [31:0] wd3,
                output logic [31:0] rd1, rd2);

logic [31:0] rf[31:0];

// three ported register file
// read two ports combinationaly (A1/RD1, A2/RD2)
// write third port on rising edge of clock (A3/WD3/WE3)
// write occurs on falling edge of clock
// register 0 hardwired to 0

always_ff @(negedge clk)
    if (we3) rf[a3] <= wd3;

assign rd1 = (a1 != 0) ? rf[a1] : 0;
assign rd2 = (a2 != 0) ? rf[a2] : 0;
endmodule

module adder(input [31:0] a, b,
             output [31:0] y);

```

```

assign y = a + b;
endmodule

module extend(input logic [31:7] instr,
               input logic [2:0] immsrc, // extended to 3 bits for lui
               output logic [31:0] immext);

  always_comb
    case(immsrc)
      // I-type
      3'b000:   immext = {{20{instr[31]}}, instr[31:20]};
      // S-type (stores)
      3'b001:   immext = {{20{instr[31]}}, instr[31:25], instr[11:7]};
      // B-type (branches)
      3'b010:   immext = {{20{instr[31]}}, instr[7], instr[30:25],
instr[11:8], 1'b0};
      // J-type (jal)
      3'b011:   immext = {{12{instr[31]}}, instr[19:12], instr[20],
instr[30:21], 1'b0};
      // U-type (lui, auipc)
      3'b100:   immext = {instr[31:12], 12'b0};
      default:   immext = 32'bx; // undefined
    endcase
  endmodule

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

  always_ff @ (posedge clk, posedge reset)
    if (reset) q <= 0;
    else q <= d;
  endmodule

module fopenr #(parameter WIDTH = 8)
  (input logic clk, reset, en,
   input logic [WIDTH-1:0] d,
   output logic [WIDTH-1:0] q);

  always_ff @ (posedge clk, posedge reset)
    if (reset) q <= 0;
    else if (en) q <= d;
  endmodule

module fopenrc #(parameter WIDTH = 8)
  (input logic clk, reset, clear, en,
   input logic [WIDTH-1:0] d,
   output logic [WIDTH-1:0] q);

  always_ff @ (posedge clk, posedge reset)
    if (reset) q <= 0;
    else if (en)
      if (clear) q <= 0;
      else q <= d;
  endmodule

```

```

module flopvc #(parameter WIDTH = 8)
    (input logic clk,
     input logic reset,
     input logic clear,
     input logic [WIDTH-1:0] d,
     output logic [WIDTH-1:0] q);

    always_ff @(posedge clk, posedge reset)
        if (reset) q <= 0;
        else
            if (clear) q <= 0;
            else q <= d;
endmodule

module mux2 #(parameter WIDTH = 8)
    (input logic [WIDTH-1:0] d0, d1,
     input logic s,
     output logic [WIDTH-1:0] y);

    assign y = s ? d1 : d0;
endmodule

module mux3 #(parameter WIDTH = 8)
    (input logic [WIDTH-1:0] d0, d1, d2,
     input logic [1:0] s,
     output logic [WIDTH-1:0] y);

    assign y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule

module imem(input logic [31:0] a,
             output logic [31:0] rd);

    logic [31:0] RAM[63:0];

    initial
        $readmemh("example.txt",RAM);

    assign rd = RAM[a[31:2]]; // word aligned
endmodule

module dmem(input logic clk, we,
            input logic [31:0] a, wd,
            output logic [31:0] rd);

    logic [31:0] RAM[63:0];

    assign rd = RAM[a[31:2]]; // word aligned

    always_ff @(posedge clk)
        if (we) RAM[a[31:2]] <= wd;
endmodule

module alu(input logic [31:0] a, b,
            input logic [2:0] alucontrol,
            output logic [31:0] result,

```

```
        output logic      zero);

logic [31:0] condinvb, sum;
logic          v;           // overflow
logic          isAddSub;    // true when is add or subtract operation

assign condinvb = alucontrol[0] ? ~b : b;
assign sum = a + condinvb + alucontrol[0];
assign isAddSub = ~alucontrol[2] & ~alucontrol[1] |
                  ~alucontrol[1] & alucontrol[0];

always_comb
case (alucontrol)
  3'b000: result = sum;          // add
  3'b001: result = sum;          // subtract
  3'b010: result = a & b;        // and
  3'b011: result = a | b;        // or
  3'b100: result = a ^ b;        // xor
  3'b101: result = sum[31] ^ v;  // slt
  default: result = 32'bx;
endcase

assign zero = (result == 32'b0);
assign v = ~(alucontrol[0] ^ a[31] ^ b[31]) & (a[31] ^ sum[31]) & isAddSub;

endmodule
```

Test Program:

```

# If successful, it should write the value 0xABCD02E to address 132 (0x84)
#           RISC-V Assembly          Description          Address    Machine
Code
main:   addi x2, x0, 5           # x2 = 5           0          00500113
        addi x3, x0, 12          # x3 = 12          4          00C00193
        addi x7, x3, -9          # x7 = (12 - 9) = 3 8          FF718393
        or x4, x7, x2            # x4 = (3 OR 5) = 7 C          0023E233
        fw( xor x5, x3, x4 )fw  # x5 = (12 XOR 7) = 11 10         0041C2B3
        fw( add x5, x5, x4 )fw  # x5 = (11 + 7) = 18 14         004282B3
        fw( beq x5, x7, end     # shouldn't be taken 18        02728863
        fw( slt x4, x3, x4     # x4 = (12 < 7) = 0 1C         0041A233
        fw( beq x4, x0, around  # should be taken 20        00020463
        addi x5, x0, 0            # shouldn't happen 24        00000293
around:  slt x4, x7, x2 )fw  # x4 = (3 < 5) = 1 28        0023A233
        fw( add x7, x4, x5 )fw  # x7 = (1 + 18) = 19 2C        005203B3
        sub x7, x7, x2 )fw      # x7 = (19 - 5) = 14 30        402383B3
        sw x7, 84(x3) )fw       # [96] = 14          34        0471AA23
        lw x2, 96(x0)           # x2 = [96] = 14          38        06002103
        stall { add x9, x2, x5 } # x9 = (14 + 18) = 32          3C        005104B3
        jal x3, end              # jump to end, x3 = 0x44          40        008001EF
        addi x2, x0, 1            # shouldn't happen          44        00100113
end:    add x2, x2, x9           # x2 = (14 + 32) = 46          48        00910133
        addi x4, x0, 1            # x4 = 1                  4C        00100213
        lui x5, 0x800000          # x5 = 0x80000000          50        800002b7
        slt x6, x5, x4 )fw      # x6 = 1                  54        0042a333
wrong:  beq x6, x0, wrong      # shouldn't be taken          58        00030063
        lui x9, 0xABCD000         # x9 = 0xABCD000          5C        ABCDE4B7
        add x2, x2, x9 )fw      # x2 = 0xABCD02E          60        00910133
        sw x2, 0x40(x3) )fw     # mem[132] = 0xABCD02E       64        0421a023
done:   beq x2, x2, done        # infinite loop             68        00210063
        stall

```

Exercise 7.45

SystemVerilog

```

module hazard(input logic [4:0] Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW,
               input logic      PCSrcE, ResultSrcEb0,
               input logic      RegWriteM, RegWriteW,
               output logic [1:0] ForwardAE, ForwardBE,
               output logic      StallF, StallD, FlushD, FlushE);

    logic lwStallD;

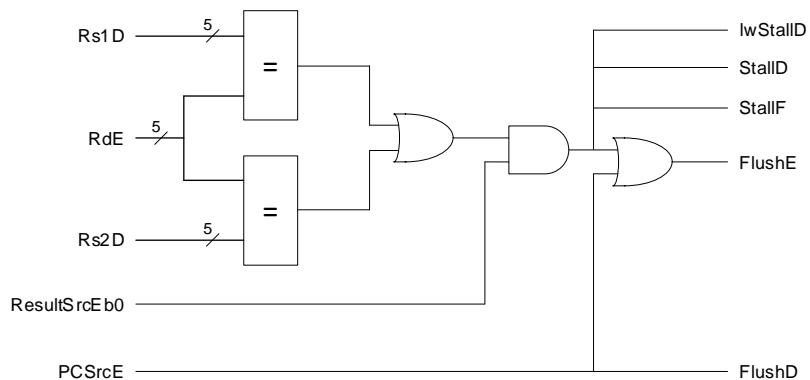
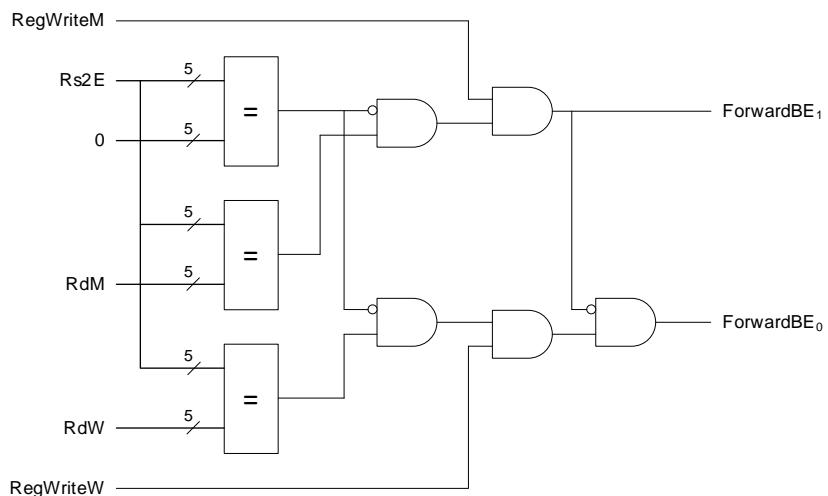
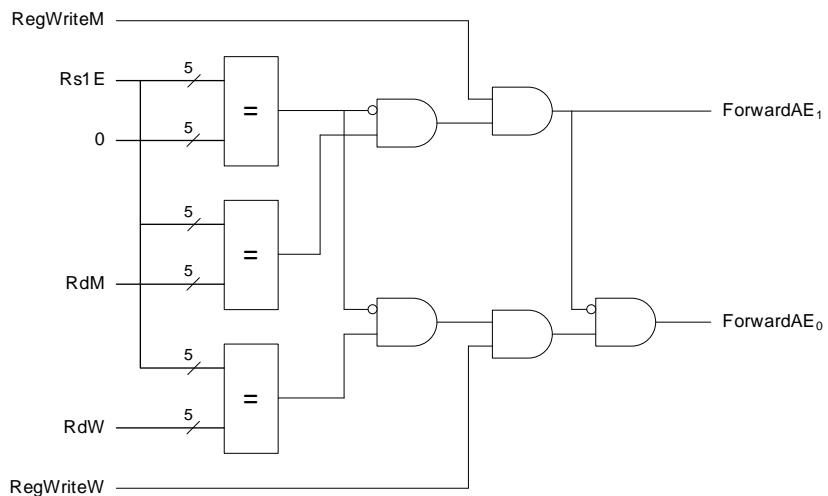
    // forwarding logic
    always_comb begin
        ForwardAE = 2'b00;
        ForwardBE = 2'b00;
        if (Rs1E != 5'b0)
            if ((Rs1E == RdM) & RegWriteM) ForwardAE = 2'b10;
            else if ((Rs1E == RdW) & RegWriteW) ForwardAE = 2'b01;

        if (Rs2E != 5'b0)
            if ((Rs2E == RdM) & RegWriteM) ForwardBE = 2'b10;
            else if ((Rs2E == RdW) & RegWriteW) ForwardBE = 2'b01;
    end

    // stalls and flushes
    assign lwStallD = ResultSrcEb0 & ((Rs1D == RdE) | (Rs2D == RdE));
    assign StallD = lwStallD;
    assign StallF = lwStallD;
    assign FlushD = PCSrcE;
    assign FlushE = lwStallD | PCSrcE;
endmodule

```

Pipelined RISC-V Processor Hazard Unit

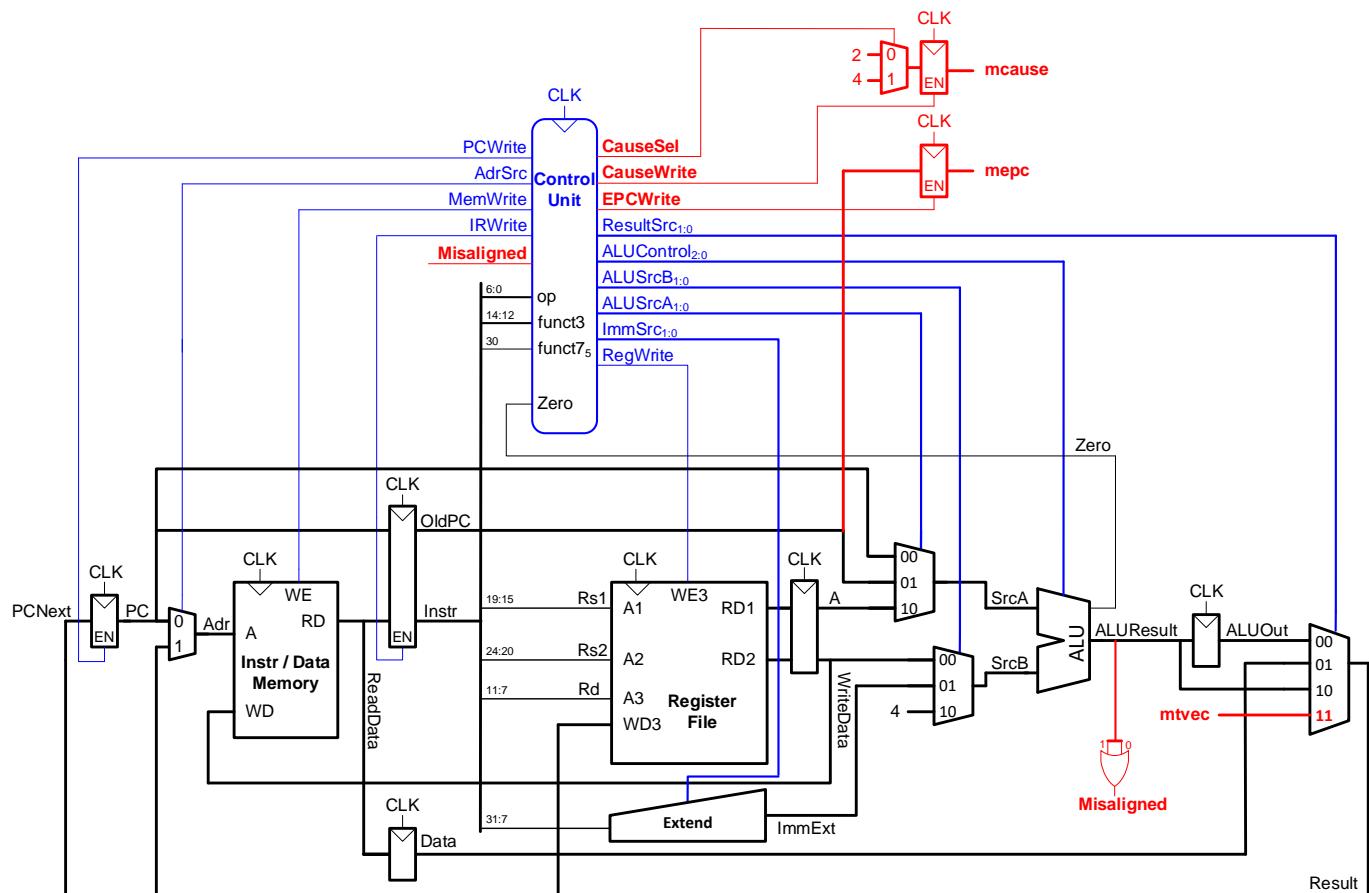


Exercise 7.47

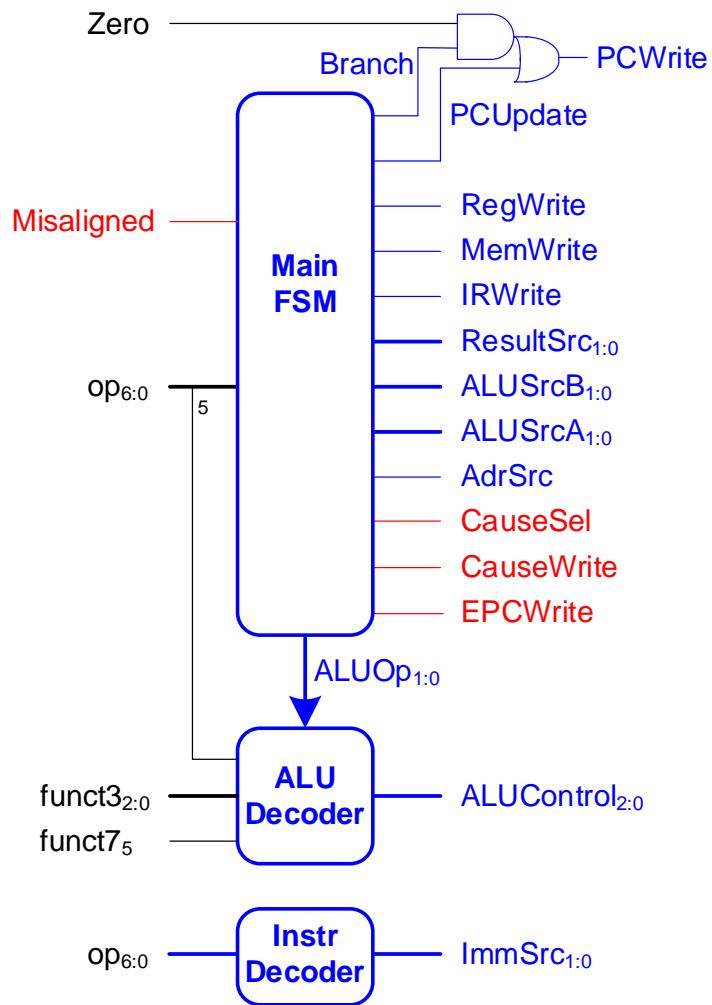
We add a datapath signal, *Misaligned*, that is true when any of the two least significant bits of the address (*ALUResult*_{1:0}) is 1. *Misaligned* feeds into the Control Unit.

We also expand the Result multiplexer so that it can select *mtvec* (the instruction address of the exception handler). We also add enabled registers to hold *mcause* and *mepc*, the cause code of the exception (*mcause*) and the PC where the exception occurred (*mepc*). We expand the control unit to produce enables for these registers (*CauseWrite* and *EPCWrite*). A select signal, *CauseSel*, chooses amongst exception causes (i.e., cause codes). In this case, we are only supporting a single exception cause, the misaligned load exception (cause code = 4), so the multiplexer isn't actually needed. But adding it allows us to expand to support other exception causes in the future.

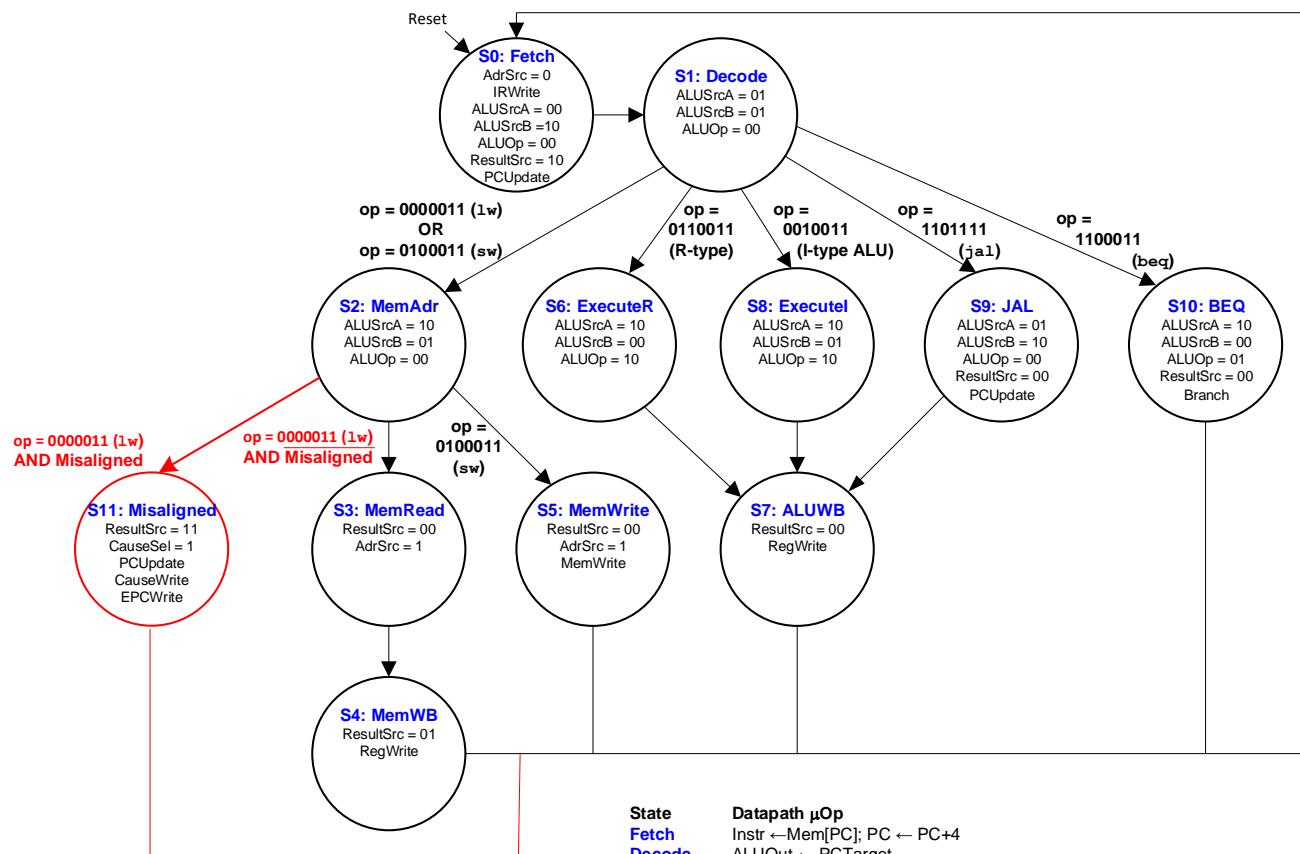
RISC-V multicycle processor modified to support the misaligned load exception



RISC-V multicycle control modified to support the misaligned load exception



Main FSM modified to support the misaligned load exception



State	Datapath μOp
Fetch	Instr ← Mem[PC]; PC ← PC+4
Decode	ALUOut ← PCTarget
MemAddr	ALUOut ← rs1 + imm
MemRead	Data ← Mem[ALUOut]
MemWB	rd ← Data
MemWrite	Mem[ALUOut] ← rd
ExecuteR	ALUOut ← rs1 op rs2
Executel	ALUOut ← rs1 op imm
ALUWB	rd ← ALUOut
BEQ	ALUResult = rs1 - rs2; if Zero, PC = ALUOut
JAL	PC = ALUOut; ALUOut = PC+4
Misaligned	PC ← mtvec; mcause ← 4; mepc ← PC

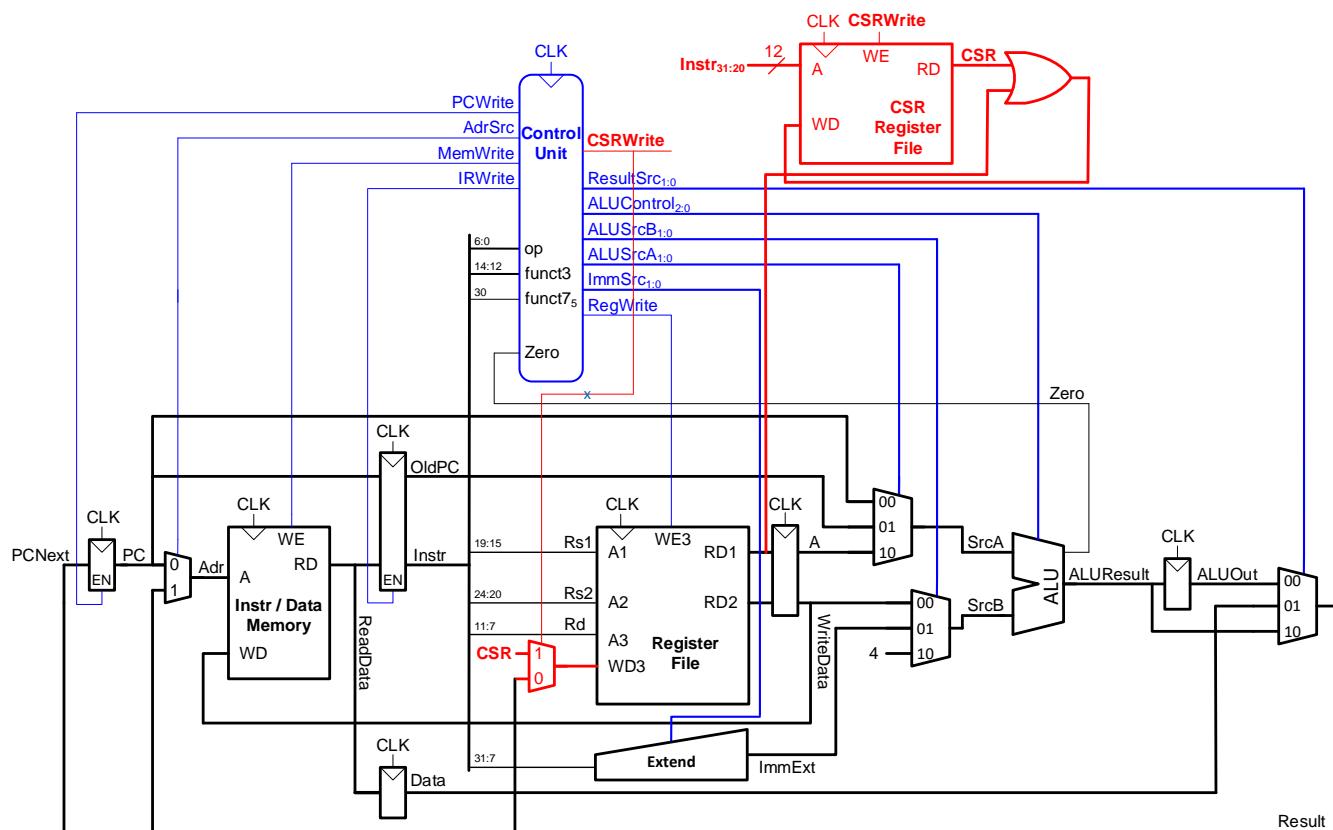
Exercise 7.49

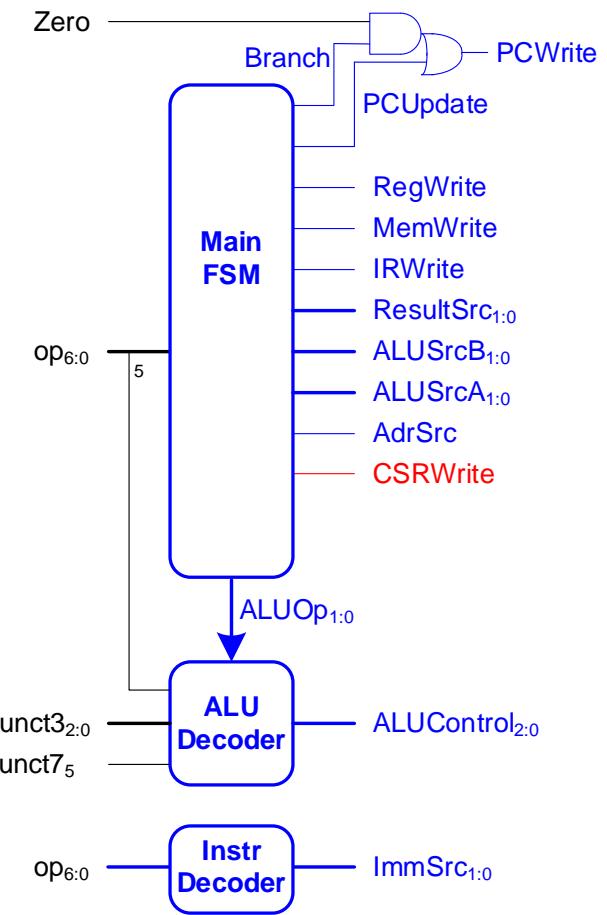
We add a control signal, *CSRWrite*, that is true to both write the CSR with $rs1 \mid CSR$ and write the CSR to *rd*.

We add the following hardware:

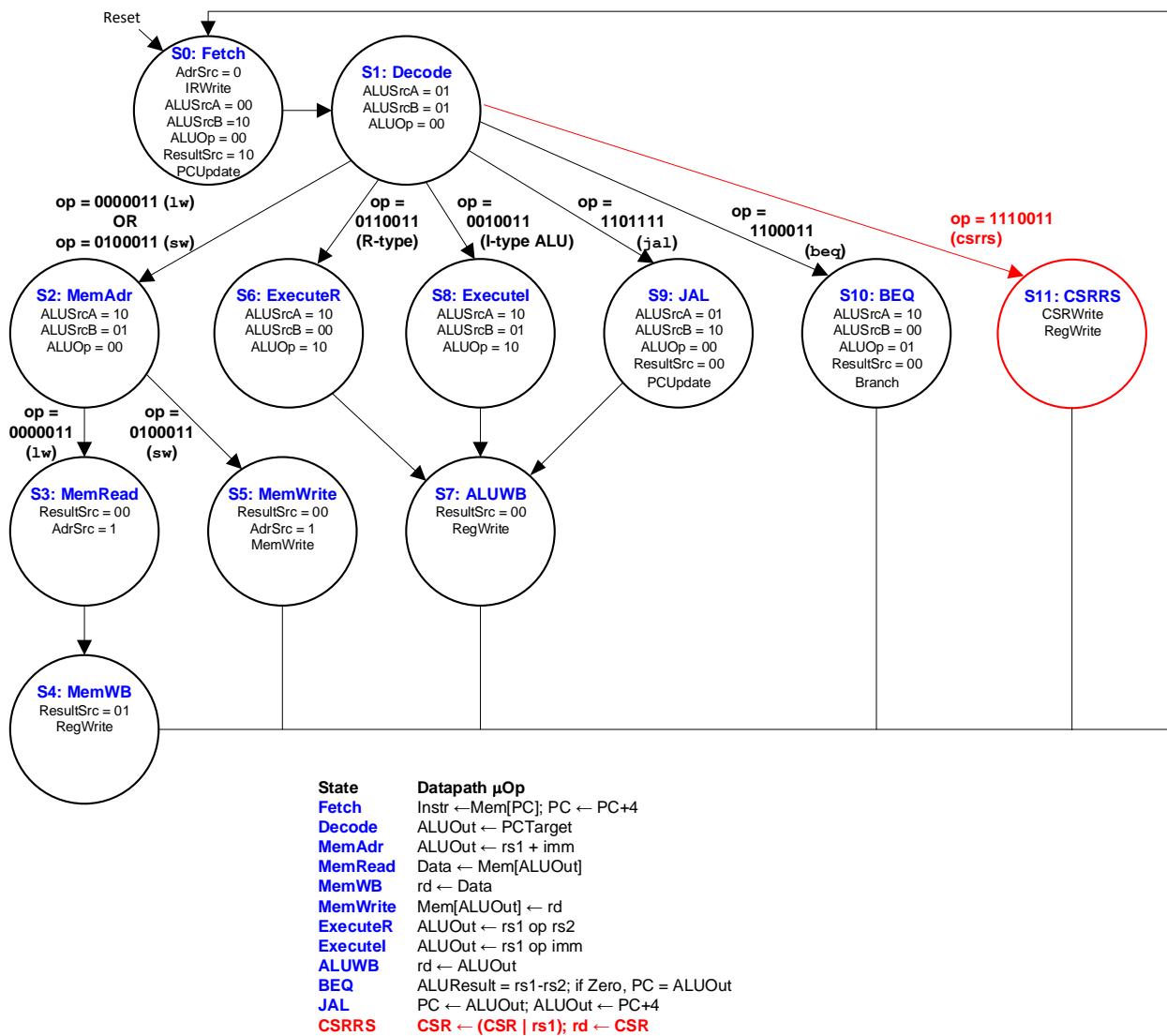
- a multiplexer in front of the WD3 port to select either *Result* or $(rs1 \mid CSR)$ to write to the register file.
- A CSR register file. The CSR number is given in the immediate field ($instr_{31:20}$) of the instruction. The CSR register file has a single read/write port. The OR gate performs $rs1 \mid CSR$ and feeds the result to the CSR register file's WD (writedata) port.

RISC-V multicycle processor modified to support the `csrrs` instruction



RISC-V multicycle control modified to support the `csrrs` instruction

Main FSM modified to support the `csrrs` instruction



Question 7.1

The primary advantages of pipelined processors are faster cycle time and temporal parallelism. Under ideal conditions, an N -stage pipelined processor is N times faster than a nonpipelined processor. The speedup comes at the cost of additional hardware, mainly pipelined registers and a hazard unit.

Question 7.3

A hazard in a pipelined microprocessor occurs when one instruction is dependent on the results of another that has not yet completed. The hazard is either a data hazard or a control hazard. A data hazard happens when an instruction tries to read a register that has not yet been written back by a previous instruction. A control hazard happens when the decision of what instruction to fetch has not been made before the first fetch takes place. Several options exist for dealing with these hazards:

- 1) **Require the programmer/complier to insert nop instructions or reorder the code to eliminate dependencies.**

Pros: No additional hardware or hazard unit is needed, which reduces cost and power usage.

Cons: This complicates programming and degrades the performance of the microprocessor.

- 2) **Have the hardware stall (or flush the pipeline for branches/jumps) when a dependency exists.**

Pros: Requires minimal added hardware.

Cons: Performance is not maximized (cases where forwarding can be used instead of stalling).

- 3) **Forward the result from the Memory/Writeback Stage to the dependent instruction in the Execute stage and stall/flush when not possible.**

Pros: Greatest performance advantage.

Cons: Requires the most additional hardware (Hazard Unit and multiplexers).

CHAPTER 8

Exercise 8.1

Answers will vary.

Temporal locality: (1) making phone calls (if you called someone recently, you're likely to call them again soon). (2) using a textbook (if you used a textbook recently, you will likely use it again soon).

Spatial locality: (1) reading a magazine (if you looked at one page of the magazine, you're likely to look at next page soon). (2) walking to locations on campus - if a student is visiting a professor in the engineering department, she or he is likely to visit another professor in the engineering department soon.

Exercise 8.3

Repeat data accesses to the following addresses:

0x0 0x10 0x20 0x30 0x40

The miss rate for the fully associative cache is: 100%. Miss rate for the direct-mapped cache is $2/5 = 40\%$.

Exercise 8.5

(a) Increasing block size will increase the cache's ability to take advantage of spatial locality. This will reduce the miss rate for applications with spatial locality. However, it also decreases the number of locations to map an address, possibly increasing conflict misses. Also, the miss penalty (the amount of time it takes to fetch the cache block from memory) increases.

(b) Increasing the associativity increases the amount of necessary hardware but in most cases decreases the miss rate. Associativities above 8 usually show only incremental decreases in miss rate.

(c) Increasing the cache size will decrease capacity misses and could decrease conflict misses. It could also, however, increase access time.

Exercise 8.7

(a) False.

Counterexample: A 2-word cache with block size of 1 word and access pattern:

048

This has a 50% miss rate with a direct-mapped cache, and a 100% miss rate with a 2-way set associative cache.

(b) True.

The 16KB cache is a superset of the 8KB cache. (Note: it's possible that they have the same miss rate.)

(c) Usually true.

Instruction memory accesses display great spatial locality, so a large block size reduces the miss rate.

Exercise 8.9

The figure below shows where each address maps for each cache configuration.

Set 15	7C 78 74 70
Set 7	20 9C 1C 98 18 94 14 90 10 4C 8C C 48 88 8 44 84 4
Set 0	40 80 0

(a) Direct Mapped

(a) Direct Mapped

7C	9C	1C
78	98	18
74	94	14
70	90	10
4C	8C	C
48	88	8
44	84	4
40	80	0 20

(c) 2-way assoc

78-7C
70-74
20-24
98-9C 18-1C
90-94 10-14
48-4C 88-8C 8-C
40-44 80-84 0-4
(d) direct measured

(d) direct mapped b=2

- (a) **80% miss rate.** Addresses 70-7C and 20 use unique cache blocks and are not removed once placed into the cache. Miss rate is $20/25 = 80\%$.
 - (b) **100% miss rate.** A repeated sequence of length greater than the cache size produces no hits for a fully-associative cache using LRU.

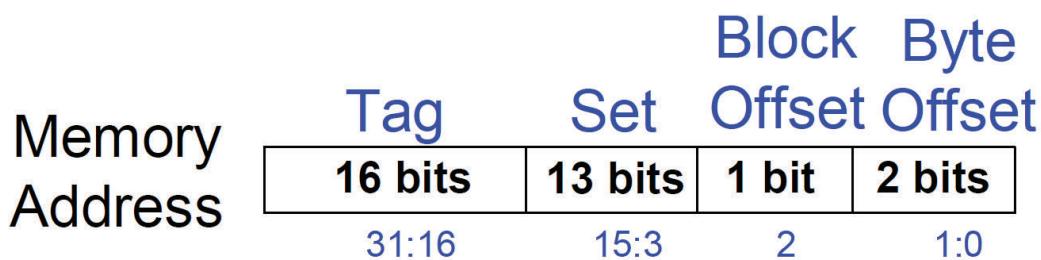
- (c) **100% miss rate.** The repeated sequence makes at least three accesses to each set during each pass. Using LRU replacement, each value must be replaced each pass through.
- (d) **40% miss rate.** Data words from consecutive locations are stored in each cache block. The larger block size is advantageous since accesses in the given sequence are made primarily to consecutive word addresses. A block size of two cuts the number of block fetches in half since two words are obtained per block fetch. The address of the second word in the block will always hit in this type of scheme (e.g. address 44 of the 40-44 address pair). Thus, the second consecutive word accesses always hit: 44, 4C, 74, 7C, 84, 8C, 94, 9C, 4, C, 14, 1C. Tracing block accesses (see Figure 8.1) shows that three of the eight blocks (70-74, 78-7C, 20-24) also remain in memory. Thus, the hit rate is: $15/25 = 60\%$ and miss rate is 40%.

Exercise 8.11

- (a) 128
- (b) 100%
- (c) ii

Exercise 8.13

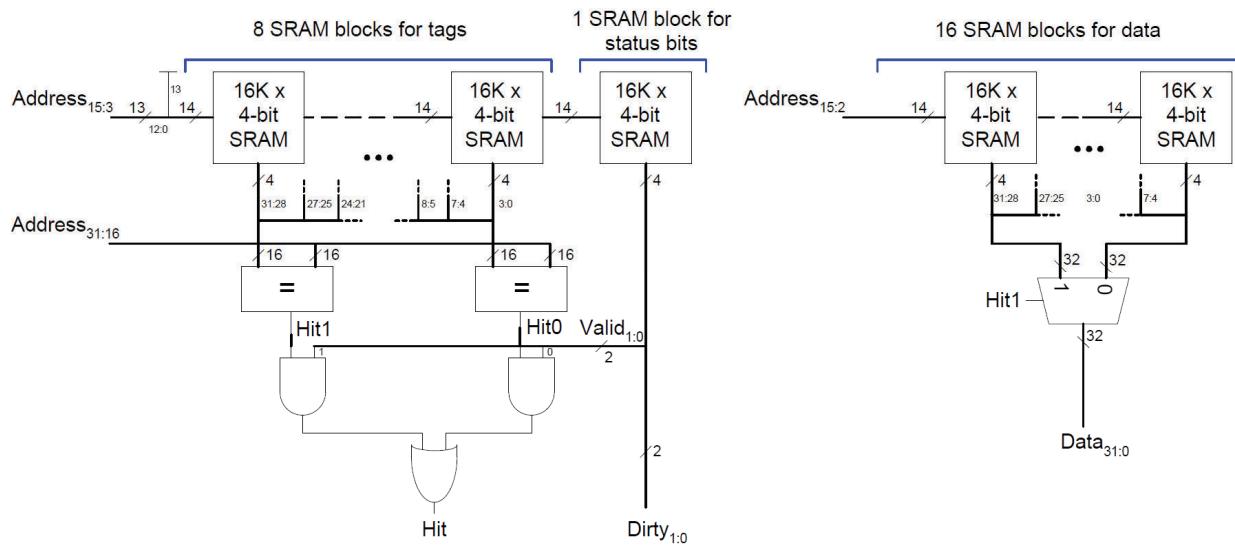
- (a)



- (b) Each tag is 16 bits. There are $32K \text{ words} / (2 \text{ words / block}) = 16K \text{ blocks}$ and each block needs a tag: $16 \times 16K = 218 = \mathbf{256 \text{ Kbits}}$ of tags.
- (c) Each cache block requires: 2 status bits, 16 bits of tag, and 64 data bits, thus each set is $2 \times 82 \text{ bits} = \mathbf{164 \text{ bits}}$.
- (d) See figure below. The design must use enough RAM chips to handle both the total capacity and the number of bits that must be read on each cycle. For the data, the SRAM must provide a capacity of 128 KB and must read 64 bits per cycle (one 32-bit word from each way). Thus the design needs at least $128\text{KB} / (8\text{KB/RAM}) = 16 \text{ RAMs}$ to hold the data and $64 \text{ bits} / (4 \text{ pins/RAM}) = 16 \text{ RAMs}$ to supply the number of bits. These are equal, so the design needs exactly 16 RAMs for the data.

For the tags, the total capacity is 32 KB, from which 32 bits (two 16-bit tags) must be read each cycle. Therefore, only 4 RAMs are necessary to meet the capacity, but 8 RAMs are needed to supply 32 bits per cycle. Therefore, the design will need 8 RAMs, each of which is being used at half capacity.

With 8K sets, the status bits require another $8K \times 4$ -bit RAM. We use a $16K \times 4$ -bit RAM, using only half of the entries.



Bits 15:2 of the address select the word within a set and block. Bits 15-3 select the set. Bits 31:16 of the address are matched against the tags to find a hit in one (or none) of the two blocks with each set.

Exercise 8.15

(a) **FIFO:** FIFO replacement approximates LRU replacement by discarding data that has been in the cache longest (and is thus least likely to be used again). A FIFO cache can be stored as a queue, so the cache need not keep track of the least recently used way in an N-way set-associative cache. It simply loads a new cache block into the next way upon a new access. FIFO replacement doesn't work well when the least recently used data is not also the data fetched longest ago.

Random: Random replacement requires less overhead (storage and hardware to update status bits). However, a random replacement policy might randomly evict recently used data. In practice random replacement works quite well.

(b) FIFO replacement would work well for an application that accesses a first set of data, then the second set, then the first set again. It then accesses a third set of data and finally goes back to access the second set of data. In this case, FIFO would replace the first set with the third set, but LRU would replace the second set. The LRU replacement would require the cache to pull in the second set of data twice.

Exercise 8.17

(a) $AMAT = t_{cache} + MR_{cache} t_{MM}$

With a cycle time of 1/1 GHz = 1 ns,

$$AMAT = 1 \text{ ns} + 0.15(200 \text{ ns}) = 31 \text{ ns}$$

(b) $CPI = 31 + 4 = 35 \text{ cycles}$ (for a load)
 $CPI = 31 + 3 = 34 \text{ cycles}$ (for a store)

(c) Average CPI = $(0.11 + 0.02)(3) + (0.52)(4) + (0.1)(34) + (0.25)(35) = 14.6$

(d) Average CPI = $14.6 + 0.1(200) = 34.6$

Exercise 8.19

From Figure 8.4, \$1 million will buy about $(\$1 \text{ million} / (\$0.05/\text{GB})) = 20 \text{ million GB}$ of hard disk:

$$20 \text{ million GB} \approx 2^{25} \times 2^{30} \text{ bytes} = 2^{55} \text{ bytes} = 2^5 \text{ petabytes} = 32 \text{ petabytes}$$

\$1 million will buy about $(\$1,000,000 / (\$7/\text{GB})) \approx 143,000 \text{ GB}$ of DRAM.

$$143,000 \text{ GB} \approx 2^7 \times 2^{10} \times 2^{30} = 2^{47} \text{ bytes} = 2^7 \text{ terabytes} = 128 \text{ terabytes}$$

Thus, the system would need **47 bits** for the physical address and **55 bits** for the virtual address.

Exercise 8.21

(a) **31 bits**

(b) $2^{50}/2^{12} = 2^{38}$ **virtual pages**

(c) $2 \text{ GB} / 4 \text{ KB} = 2^{31}/2^{12} = 2^{19}$ **physical pages**

(d) virtual page number: **38 bits**; physical page number = **19 bits**

(e) 2^{38} page table entries (one for each virtual page).

(f) Each entry uses 19 bits of physical page number and 2 bits of status information. Thus, **3**

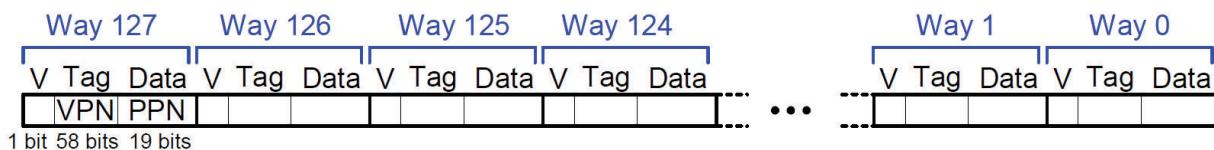
bytes are needed for each entry (rounding 21 bits up to the nearest number of bytes).

(g) The total table size is **3×2^{38} bytes**.

Exercise 8.23

$$(a) 1 \text{ valid bit} + 19 \text{ data bits (PPN)} + 38 \text{ tag bits (VPN)} \times 128 \text{ entries} = 58 \times 128 \text{ bits} = \mathbf{7424 \text{ bits}}$$

(b)



(c) 128×58 -bit SRAM

Exercise 8.25

(a) Each entry in the page table has 2 status bits (V and D), and a physical page number ($22-16 = 6$ bits). The page table has $2^{25-16} = 2^9$ entries.

Thus, the total page table size is $2^9 \times 8$ bits = **4096 bits**

(b) This would increase the virtual page number to $25 - 14 = 11$ bits, and the physical page number to $22 - 14 = 8$ bits. This would increase the page table size to:

$$2^{11} \times 10 \text{ bits} = \mathbf{20480 \text{ bits}}$$

This increases the page table by 5 times, wasted valuable hardware to store the extra page table bits.

(c) Yes, this is possible. In order for concurrent access to take place, the number of set + block offset + byte offset bits must be less than the page offset bits.

(d) It is impossible to perform the tag comparison in the on-chip cache concurrently with the page table access because the upper (most significant) bits of the physical address are unknown until after the page table lookup (address translation) completes.

Exercise 8.27

$$(a) 2^{32} \text{ bytes} = 4 \text{ gigabytes}$$

(b) The amount of the hard disk devoted to virtual memory determines how many applications can run and how much virtual memory can be devoted to each application.

(c) The amount of physical memory affects how many physical pages can be accessed at once. With a small main memory, if many applications run at once or a single application accesses

addresses from many different pages, thrashing can occur. This can make the applications dreadfully slow.

Question 8.1

Caches are categorized based on the number of blocks (B) in a set. In a direct-mapped cache, each set contains exactly one block, so the cache has $S = B$ sets. Thus a particular main memory address maps to a unique block in the cache. In an N -way set associative cache, each set contains N blocks. The address still maps to a unique set, with $S = B / N$ sets. But the data from that address can go in any of the N blocks in the set. A fully associative cache has only $S = 1$ set. Data can go in any of the B blocks in the set. Hence, a fully associative cache is another name for a B -way set associative cache.

A **direct mapped cache** performs better than the other two when the data access pattern is to sequential cache blocks in memory with a repeat length one greater than the number of blocks in the cache.

An **N -way set-associative cache** performs better than the other two when N sequential block accesses map to the same set in the set-associative and direct-mapped caches. The last set has $N+1$ blocks that map to it. This access pattern then repeats.

In the direct-mapped cache, the accesses to the same set conflict, causing a 100% miss rate. But in the set-associative cache all accesses (except the last one) don't conflict. Because the number of block accesses in the repeated pattern is one more than the number of blocks in the cache, the fully associative cache also has a 100% miss rate.

A **fully associative cache** performs better than the other two when the direct- mapped and set- associative accesses conflict and the fully associative accesses don't. Thus, the repeated pattern must access at most B blocks that map to conflicting sets in the direct and set-associative caches.

Question 8.3

The advantages of using a virtual memory system are the illusion of a larger memory without the expense of expanding the physical memory, easy relocation of programs and data, and protection between concurrently running processes. The disadvantages are a more complex

memory system and the sacrifice of some physical and possibly virtual memory to store the page table.