SOLUTIONS
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 $12 \times 4 = 48$ 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**

$2^{16} = 65,536$ 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

Exercise 1.47

46.566 gigabytes

Exercise 1.49

128 kbits

Exercise 1.51

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

(a) 000111 + 001101 = 010100
(b) 010001 + 011001 = 101010, overflow
(c) 100110 + 001000 = 101110
(d) \[ 011111 + 110010 = 010001 \]
(c) \[ 101101 + 101010 = 010111, \text{ overflow} \]
(f) \[ 111110 + 100011 = 100001 \]

**Exercise 1.59**

(a) \( 0x2A \); (b) \( 0x9F \); (c) \( 0xFE \); (d) \( 0x66, \text{ overflow} \)

**Exercise 1.61**

(a) \[ 010010 + 110100 = 000110 \]; (b) \[ 011110 + 110111 = 010101 \]; (c) \[ 100100 + 111101 = 100001 \]; (d) \[ 110000 + 101011 = 011011, \text{ overflow} \]

**Exercise 1.63**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td>-2</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>110</td>
<td>111</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Exercise 1.65**

(a) \[ 0011 \ 0111 \ 0001 \]
(b) \( 187 \)
(c) \( 95 = 101111 \)
(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**

```c
#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

[Diagram of OR3, XOR3, and XNOR4 logic circuits]

Exercise 1.73

[Table of OR3 truth table]

Exercise 1.75
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

\[
\begin{array}{ccc|c}
A & B & C & Y \\
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 \\
0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 \\
\end{array}
\]
Exercise 1.87

XOR

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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) \( Y = \overline{AB} + \overline{AB} + AB \)
(b) \( Y = \overline{AB\overline{C}} + ABC \)
(c) \( Y = \overline{ABC} + \overline{ABC} + \overline{ABC} + \overline{ABC} + ABC \)
(d) \[ \begin{align*}
Y = & \overline{ABCD} + \overline{ABCD} + \overline{ABCD} + \overline{ABCD} + \overline{ABCD} + \overline{ABCD} + ABCD \\
Y = & \overline{ABCD} + \overline{ABCD} + \overline{ABCD} + \overline{ABCD} + \overline{ABCD} + \overline{ABCD} + ABCD
\end{align*} \]
(e) \[ \begin{align*}
Y = & \overline{ABCD} + \overline{ABCD} + \overline{ABCD} + \overline{ABCD} + \overline{ABCD} + \overline{ABCD} + ABCD
\end{align*} \]

Exercise 2.3

(a) \( Y = (A + \overline{B}) \)
(b) \[ \begin{align*}
Y = & (A + B + \overline{C})(A + \overline{B} + C)(A + B + \overline{C})(A + B + C)(A + B + \overline{C})(A + B + C)
\end{align*} \]
(c) \( Y = (A + B + \overline{C})(A + \overline{B} + C)(A + B + \overline{C})(A + B + C) \)
(d) \[ \begin{align*}
Y = & (A + \overline{B} + C + D)(A + \overline{B} + C + D)(A + \overline{B} + C + D)(A + \overline{B} + C + D)(A + B + C + D) \\
Y = & (A + B + C + D)(A + \overline{B} + C + D)(A + B + C + D)(A + B + C + D)(A + B + C + D)
\end{align*} \]
(e) \[ \begin{align*}
Y = & (A + B + C + D)(A + B + C + D)(A + B + C + D)(A + B + C + D)
\end{align*} \]
Exercise 2.5

(a) \( Y = A + \overline{B} \)
(b) \( Y = \overline{A}BC + ABC \)
(c) \( Y = \overline{A}C + A\overline{B} + AC \)
(d) \( Y = \overline{A}B + BD + ACD \)
(e) \[ Y = \overline{ABCD} + \overline{ABCD} + \overline{ABCD} + \overline{ABCD} + \overline{ABCD} + \overline{ABCD} + \overline{ABCD} + \overline{ABCD} \]

This can also be expressed as:
\[ Y = (A \oplus B)(C \oplus D) + (A \oplus B)(C \oplus D) \]

Exercise 2.7

(a)

![Diagram for Exercise 2.7(a)](image)

(b)

![Diagram for Exercise 2.7(b)](image)

(c)

![Diagram for Exercise 2.7(c)](image)
Exercise 2.9

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

(c)
Exercise 2.11

(a)

(b)
Exercise 2.13

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

Exercise 2.15

(a)
Exercise 2.17

(a) \( Y = B + \overline{AC} \)

(b) \( Y = \overline{AB} \)

(c) \( Y = A + \overline{BC} + \overline{DE} \)

Exercise 2.19
4 gigarows = $4 \times 2^{30}$ 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 $ACD$ and $BCD$ are both prime implicants, the minimal sum-of-products expression does not have both of them.

![K-map for Exercise 2.21](image)

$Y = \overline{AB} + \overline{ABC} + \overline{ACD}$

$Y = \overline{AB} + \overline{ABC} + BCD$

**Exercise 2.23**

<table>
<thead>
<tr>
<th>$B_2$</th>
<th>$B_1$</th>
<th>$B_0$</th>
<th>$B_2 \cdot B_1 \cdot B_0$</th>
<th>$B_2 + B_1 + B_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Exercise 2.25**
Exercise 2.27
Exercise 2.29

Two possible options are shown below:

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

Exercise 2.31

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

Exercise 2.33

The equation can be written directly from the description:

E = SA + AL + H
Exercise 2.35

$P$ has two possible minimal solutions:

```
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
```

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.

<table>
<thead>
<tr>
<th>$A_7$</th>
<th>$A_6$</th>
<th>$A_5$</th>
<th>$A_4$</th>
<th>$A_3$</th>
<th>$A_2$</th>
<th>$A_1$</th>
<th>$A_0$</th>
<th>$Y_2$</th>
<th>$Y_1$</th>
<th>$Y_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$X$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$X$</td>
<td>$X$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$X$</td>
<td>$X$</td>
<td>$X$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$X$</td>
<td>$X$</td>
<td>$X$</td>
<td>$X$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>$X$</td>
<td>$X$</td>
<td>$X$</td>
<td>$X$</td>
<td>$X$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$X$</td>
<td>$X$</td>
<td>$X$</td>
<td>$X$</td>
<td>$X$</td>
<td>$X$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

$Y_1 = A_7 \cdot A_6 + \overline{A_5} \cdot A_4 \cdot A_3 + \overline{A_5} \cdot A_4 \cdot A_2$
\[ Y_0 = A_7 + A_6A_5 + A_6A_4A_3 + A_6A_4A_2A_1 \]
The truth table, equations, and circuit for $Z_{2:0}$ are as follows.

<table>
<thead>
<tr>
<th>$A_7$</th>
<th>$A_6$</th>
<th>$A_5$</th>
<th>$A_4$</th>
<th>$A_3$</th>
<th>$A_2$</th>
<th>$A_1$</th>
<th>$A_0$</th>
<th>$Z_2$</th>
<th>$Z_1$</th>
<th>$Z_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>


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

$Z_0 = A_1(A_2 + A_3 + A_4 + A_5 + A_6 + A_7) + \quad A_3(A_4 + A_5 + A_6 + A_7) + A_5(A_6 + A_7)$
Exercise 2.39

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

Exercise 2.41
Exercise 2.43

\[ t_{pd} = 3t_{pd\_NAND2} = 60 \text{ ps} \]
\[ t_{cd} = t_{cd\_NAND2} = 15 \text{ ps} \]

Exercise 2.45

\[ t_{pd} = t_{pd\_NOT} + t_{pd\_AND3} \]
\[ = 15 \text{ ps} + 40 \text{ ps} \]
\[ = 55 \text{ ps} \]
\[ t_{cd} = t_{cd\_AND3} \]
\[ = 30 \text{ ps} \]
Exercise 2.47

\[ t_{pd} = t_{pd \_INV} + 3t_{pd \_NAND2} + t_{pd \_NAND3} \]
\[ = [15 + 3 \times 20] \text{ ps} \]
\[ = 105 \text{ ps} \]

\[ t_{cd} = t_{cd \_NOT} + t_{cd \_NAND2} \]
\[ = [10 + 15] \text{ ps} \]
\[ = 25 \text{ ps} \]
Question 2.1

![Diagram of two NAND gates connected to a tristate buffer]

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 LVCMOS (low voltage CMOS) chips. As temperature increases and voltage decreases, circuit delay increases. Also, the circuit may have different paths (critical and short paths) from the input to the output. A gate itself may have varying delays between different inputs and the output, affecting the gate’s critical and short paths. For example, for a two-input NAND gate, a HIGH to LOW transition requires two nMOS transistor delays, whereas a LOW to HIGH transition requires a single pMOS transistor delay.
Exercise 3.1

S

R

Q

Exercise 3.3

clk

D

Q
Exercise 3.5

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

Exercise 3.9

If $A$ and $B$ have the same value, $C$ takes on that value. Otherwise, $C$ retains its old value.
Exercise 3.13

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

Exercise 3.15

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

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.

<table>
<thead>
<tr>
<th>state</th>
<th>encoding $s_{1:0}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>00</td>
</tr>
<tr>
<td>S1</td>
<td>01</td>
</tr>
<tr>
<td>S2</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>current state</th>
<th>inputs</th>
<th>next state</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>$s_0$</td>
<td>$a$</td>
<td>$b$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$S'_1 = \overline{s}_1s_0B + s_1AB$

$S'_0 = \overline{s}_1\overline{s}_0A$

$Q' = s_1AB$
Exercise 3.25

State encoding for Exercise 3.25

<table>
<thead>
<tr>
<th>State</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>000</td>
</tr>
<tr>
<td>S1</td>
<td>001</td>
</tr>
</tbody>
</table>

**TABLE 3.3** State encoding for Exercise 3.25
### Table 3.3: State encoding for Exercise 3.25

<table>
<thead>
<tr>
<th>State</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>S2</td>
<td>010</td>
</tr>
<tr>
<td>S3</td>
<td>100</td>
</tr>
<tr>
<td>S4</td>
<td>101</td>
</tr>
</tbody>
</table>

### Table 3.4: Combined state transition and output table with binary encodings for Exercise 3.25

<table>
<thead>
<tr>
<th>Current State</th>
<th>Input</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_2$</td>
<td>$s_1$</td>
<td>$s_0$</td>
<td>$a$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$$S'_2 = \overline{s_2}s_1s_0 + s_2\overline{s_1}s_0$$

$$S'_1 = \overline{s_2}s_1s_0a$$

$$S'_0 = a(\overline{s_2}s_0 + s_2\overline{s_1})$$

$$Q = s_2\overline{s_1}s_0a + s_2\overline{s_1}s_0\overline{a}$$
Exercise 3.27
FIGURE 3.1 State transition diagram for Exercise 3.27
TABLE 3.5 State transition table for Exercise 3.27

<table>
<thead>
<tr>
<th>current state $s_{2:0}$</th>
<th>next state $s'_{2:0}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
</tr>
<tr>
<td>001</td>
<td>011</td>
</tr>
<tr>
<td>011</td>
<td>010</td>
</tr>
<tr>
<td>010</td>
<td>110</td>
</tr>
<tr>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>111</td>
<td>101</td>
</tr>
<tr>
<td>101</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>000</td>
</tr>
</tbody>
</table>

$S_2 = S_1 \overline{S_0} + S_2 S_0$

$S_1 = \overline{S_2} S_0 + S_1 \overline{S_0}$

$S'_0 = S_2 \oplus S_1$

$Q_2 = S_2$

$Q_1 = S_1$

$Q_0 = S_0$
FIGURE 3.2  Hardware for Gray code counter FSM for Exercise 3.27

Exercise 3.29

(a)

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 \( \overline{BA}/0 \). The arrow from S0 to S0 would then be \( \overline{BA}/0 \).)

<table>
<thead>
<tr>
<th>current state ( s_{1:0} )</th>
<th>inputs</th>
<th>next state ( s'_{1:0} )</th>
<th>output ( z )</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>X</td>
<td>00</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

**TABLE 3.6** State transition table for Exercise 3.29
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 $Z$: $Z = A A_{\text{prev}}$ (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.

<table>
<thead>
<tr>
<th>current state</th>
<th>input</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>$s_0$</td>
<td>$x$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>$X$</td>
<td>$X$</td>
</tr>
</tbody>
</table>

*TABLE 3.7* State transition table with binary encodings for Exercise 3.31

<table>
<thead>
<tr>
<th>current state</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>$s_0$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>$X$</td>
</tr>
</tbody>
</table>

*TABLE 3.8* Output table for Exercise 3.31
Exercise 3.33

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

\[ t_{pd} = 3t_{pd\_XOR} \]
\[ = 3 \times 100 \text{ ps} \]
\[ = 300 \text{ ps} \]

Next, we calculate the cycle time:

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

(b)

\[ T_c \geq t_{pcq} + t_{pd} + t_{setup} + t_{skew} \]

Thus,

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

(c)

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

\[ t_{cd} = t_{cd\_XOR} \]
\[ = 55 \text{ ps} \]

\[ t_{ccq} + t_{cd} \geq t_{hold} + t_{skew} \]

Thus,

\[ t_{skew} < (t_{ccq} + t_{cd}) - t_{hold} \]
\[ < (50 + 55) - 20 \]
\[ < 85 \text{ ps} \]
First, we calculate the propagation and contamination delays through the combinational logic:

\[ t_{pd} = 2t_{pd}_{XOR} \]
\[ = 2 \times 100 \text{ ps} \]
\[ = 200 \text{ ps} \]

\[ t_{cd} = 2t_{cd}_{XOR} \]
\[ = 2 \times 55 \text{ ps} \]
\[ = 110 \text{ ps} \]

Next, we calculate the cycle time:

\[ T_c \geq t_{pcq} + t_{pd} + t_{\text{setup}} \]
\[ \geq [70 + 200 + 60] \text{ ps} \]
\[ = 330 \text{ ps} \]

\[ f = \frac{1}{330 \text{ ps}} = 3.03 \text{ GHz} \]

\[ t_{\text{skew}} < (t_{ccq} + t_{cd}) - t_{\text{hold}} \]
\[ < (50 + 110) - 20 \]
\[ < 140 \text{ ps} \]

**Exercise 3.35**

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

\[ T_c \geq t_{pcq} + Nt_{CLB} + t_{\text{setup}} \]
\[ 25 \text{ ns} \geq [0.72 + N(0.61) + 0.53] \text{ ps} \]

Thus, \( N < 38.9 \)

\( N = 38 \)

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

**Exercise 3.37**

\[ \frac{P(\text{failure})}{\text{sec}} = \frac{1}{\text{MTBF}} = \frac{1}{(50 \text{ years} \times 3.15 \times 10^7 \text{ sec/year})} = 6.34 \times 10^{-10} \text{ (EQ 3.26)} \]

\[ P(\text{failure})/\text{sec waiting for one clock cycle}: N \times \left(\frac{T_0}{T_c}\right) \times e^{-\frac{T_c - t_{\text{setup}}}{\tau}} \]
\[ = 0.5 \times \left(\frac{110}{1000}\right) \times e^{-\frac{1000-70}{100}} = 5.0 \times 10^{-6} \]

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

This is just less than the required probability of failure (6.34 \times 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:

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

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

\[ \frac{\text{MTBF}_2}{\text{MTBF}_1} = 10 = e^{\frac{T_{c2} - T_{c1}}{30\text{ps}}} \]

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

\[ T_{c2} - T_{c1} = 69\text{ps} \]
Thus, the clock cycle time must increase by 69 ps. This holds true for cycle times much larger than T0 (20 ps) and the increased time (69 ps).

**Question 3.1**


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

![State transition diagram for edge detector circuit](image)

TABLE 3.10 State transition table for Question 3.5

<table>
<thead>
<tr>
<th>Current State $s_{1:0}$</th>
<th>Input $a$</th>
<th>Next State $s'_{1:0}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

$S'_1 = AS_1$

$S'_0 = AS_1S_0$
$Q = S_1$

**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 $T_c - (t_{pcq} + t_{setup})$. 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 $T_c + t_{cd\_BUF} - (t_{pcq} + t_{setup})$. 
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
1110_100_1111
1111_100_0111
**Option 1:**

**SystemVerilog**

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

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

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

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

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

// check results on falling edge of clk
always @(negedge clk)
if (~reset) begin // skip during reset
    if (s !== s_expected) begin
        $display("Error: inputs = %h", data);
        $display("  outputs = %b (%b expected)",
            s, s_expected);
        errors = errors + 1;
    end
    vectornum = vectornum + 1;
    if (testvectors(vectornum) === 11'bx) begin
        $display("%d tests completed with %d errors",
            vectornum, errors);
        $finish;
    end
end
endmodule
```

** VHDL**

```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
begin
    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, s: STD_LOGIC_VECTOR(3 downto 0);
    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;

    -- 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);
            end loop;
        end loop;
    end process;

    -- at start of test, load vectors
    -- and pulse reset
    process is
        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);
            end loop;
        end loop;
    end process;
```
(continued from previous page)

**VHDL**

```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 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**

```verilog
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**

```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.

![Shift Register Diagram]

Exercise 4.13

**SystemVerilog**

```verilog
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**

```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) \( Y = AC + \overline{AB}C \)

**SystemVerilog**

```systemverilog
module ex4_15a(input logic a, b, c, output logic y);
  assign y = (a & c) | (~a & ~b & c);
endmodule
```

**VHDL**

```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) \( Y = \overline{AB} + \overline{ABC} + (A + \overline{C}) \)

**SystemVerilog**

```systemverilog
module ex4_15b(input logic a, b, c, output logic y);
  assign y = (~a & ~b) | (~a & b & ~c) | ~(a | ~c);
endmodule
```

**VHDL**

```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) \( Y = \overline{ABCD} + A\overline{BC} + \overline{ABCD} + ABD + \overline{ABCD} + B\overline{CD} + \overline{A} \)

**SystemVerilog**

```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**

```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**

```verilog
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**

```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

```verilog
module priority_encoder2(input logic [7:0] a, output logic [2:0] y, z, output logic none);
always_comb begin
  casez (a)
    8'b00000000: begin y = 3'd0; none = 1'b1; end
    8'b00000001: begin y = 3'd0; none = 1'b0; end
    8'b0000001?: begin y = 3'd1; none = 1'b0; end
    8'b000001???: begin y = 3'd2; none = 1'b0; end
    8'b00001???: begin y = 3'd3; none = 1'b0; end
    8'b0001????: begin y = 3'd4; none = 1'b0; end
    8'b001?????: begin y = 3'd5; none = 1'b0; end
    8'b01??????: begin y = 3'd6; none = 1'b0; end
    8'b1???????: begin y = 3'd7; none = 1'b0; end
    default:     z = 3'b000; 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'b0000011?: z = 3'b001;
    8'b0000101?: z = 3'b001;
    8'b0001001?: z = 3'b001;
    8'b0100001?: z = 3'b001;
    8'b0010001?: z = 3'b001;
    8'b0010010?: z = 3'b010;
    8'b0100010?: z = 3'b010;
    8'b0000110?: z = 3'b010;
    8'b0001100?: z = 3'b010;
    8'b0100100?: z = 3'b010;
    8'b0101000?: z = 3'b010;
    8'b0011000?: z = 3'b010;
    8'b0110000?: z = 3'b010;
    8'b001:?????: z = 3'b011;
    8'b011:?????: z = 3'b011;
    8'b11:?????: z = 3'b101;
    default:     z = 3'b000; end
endmodule
```

### VHDL

```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);
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 <= "010"; none <= '0';
      when "000001??" => y <= "011"; none <= '0';
      when "00001---" => y <= "100"; none <= '0';
      when "001------" => y <= "101"; none <= '0';
      when "1-------" => y <= "110"; 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 "00010001" => z <= "000";
      when "01000001" => z <= "000";
      when "0000101?" => z <= "001";
      when "0001001?" => z <= "001";
      when "0010001?" => z <= "001";
      when "0100001?" => z <= "001";
      when "00011---" => z <= "010";
      when "0011000" => z <= "010";
      when "0110000" => z <= "010";
      when "1100000" => z <= "010";
      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 <= '0';
            when X"9"   => y <= '1';
            when X"A"   => y <= '0';
            when X"B"   => y <= '1';
            when X"C"   => y <= '0';
            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: i 
output 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**

```verilog
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**

```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;
      when S1 => nextstate <= S2;
      when S2 => if tb then
                      nextstate <= S2;
                  else nextstate <= S3;
      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**

```verilog
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**

```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
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

begin
-- state register
process(clk, reset) begin
if reset then state <= S0;
elif 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;

-- 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;
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;
    end case;
  end process;
Option 2

**SystemVerilog**

```verilog
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**

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

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

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

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

(continued from previous page)

**VHDL**

```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;
```
Exercise 4.41

**SystemVerilog**

```verilog
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**

```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
    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;
        when S1 =>  if a then
                       nextstate <= S2;
                       else nextstate <= S3;
        when S2 =>  if a then
                       nextstate <= S2;
                       else nextstate <= S3;
        when S3 =>  if a then
                       nextstate <= S2;
                       else nextstate <= S3;
        when others =>   nextstate <= S0;
      end case;
    end process;

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

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

**VHDL**

```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:
\[ t_{\text{ripple}} = N t_{\text{FA}} = 64(450 \text{ ps}) = 28.8 \text{ ns} \]

(b) From Equation 5.6, we find the 64-bit carry-lookahead adder delay to be:
\[ t_{\text{CLA}} = t_{\text{pg}} + t_{\text{pg\_block}} + \left(\frac{N}{k} - 1\right) t_{\text{AND\_OR}} + k t_{\text{FA}} \]
\[ t_{\text{CLA}} = \left[ 150 + (6 \times 150) + \left(\frac{64}{4} - 1\right) 300 + (4 \times 450) \right] = 7.35 \text{ ns} \]

(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:
\[ t_{\text{PA}} = t_{\text{pg}} + \log_2 N(t_{\text{pg\_prefix}}) + t_{\text{XOR}} \]
\[ t_{\text{PA}} = \left[ 150 + 6(300) + 150 \right] = 2.1 \text{ ns} \]

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

Exercise 5.7

(a) We show an 8-bit priority circuit in Figure 5.2. In the figure $X_7 = \overline{A_7}$, $X_{7:6} = \overline{A_7A_6}$, $X_{7:5} = \overline{A_7A_6A_5}$, 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:

$$t_{pd\_priority} = (\log_2 N + 1)t_{pd\_AND2} + t_{pd\_OR/N/2}$$
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[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  
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 5.11**

**SystemVerilog**

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

logic [31:0] S, Bout;

assign S = A + Bout + F[2];

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

**VHDL**

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

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

architecture synth of alu32 is
    signal S, Bout: STD_LOGIC_VECTOR(31 downto 0);

begin
    Bout <= (not B) when (F(2) = '1') else B;
    S <= A + Bout + F(2);

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

---

SystemVerilog

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

logic [31:0] S, Bout;
assign S = A + Bout + F[2];

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

assign Zero = (Y == 32'b0);

always_comb
  case (F[2:1])
    default: Overflow <= 1'b0;
  endcase
endmodule
```

VHDL

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

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

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

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

  Zero <= '1' when (Y = X"00000000") else '0';

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

Exercise 5.13
A 2-bit left shifter creates the output by appending two zeros to the least significant bits of the input and dropping the two most significant bits.

![Figure 5.3: 2-bit left shifter, 32-bit input and output](image)

**2-bit Left Shifter**

**SystemVerilog**

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

**VHDL**

```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 5.15**
FIGURE 5.4 8-bit left shifter using 24 2:1 multiplexers

Exercise 5.17

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

**Exercise 5.19**

\[
t_{pd_{-}DIV4} = 4(t_{FA} + t_{MUX}) = 16t_{FA} + 4t_{MUX}
\]

\[
t_{pd_{-}DIVN} = N^2t_{FA} + Nt_{MUX}
\]

**Exercise 5.21**

**SystemVerilog**

```verilog
module signext4_8(input logic [3:0] a,
                  output logic [7:0] y);
    assign y = { {4{a[3]}}, a};
endmodule
```

**VHDL**

```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 5.23

Exercise 5.25

(a) \(10001101.10010000 = 0x8D90\)
(b) \(00101010.01010000 = 0x2A50\)
(c) \(10010001.00101000 = 0x9128\)

Exercise 5.27

(a) \(11110010.01110000 = 0xF270\)
(b) \(00101010.01010000 = 0x2A50\)
(c) \(11101110.11011000 = 0xEED8\)

Exercise 5.29

(a) \(-1101.1001 = -1.1011001 \times 2^3\)
Thus, the biased exponent = \(127 + 3 = 130 = 10000010_2\)
In IEEE 754 single-precision floating-point format:
\[
\begin{array}{cccccccc}
1 & 1000 & 0010 & 101 & 1001 & 0000 & 0000 & 0000 \\
\end{array} = 0xC1590000
\]

(b) \(101010.0101 = 1.0101001 \times 2^5\)
Thus, the biased exponent = \(127 + 5 = 132 = 10000100_2\)
In IEEE 754 single-precision floating-point format:
\[
\begin{array}{cccccccc}
0 & 1000 & 0100 & 0101 & 0100 & 0000 & 0000 & 0000 \\
\end{array} = 0x42294000
\]

(c) \(-10001.0010 = -1.0001001 \times 2^4\)
Thus, the biased exponent = \(127 + 4 = 131 = 10000110_2\)
In IEEE 754 single-precision floating-point format:
\[
\begin{array}{cccccccc}
1 & 1000 & 0011 & 000 & 1001 & 0100 & 0000 & 0000 \\
\end{array} = 0xC1894000
\]
Exercise 5.31

(a) 5.5
(b) \(-0.000012 = -0.0625\)
(c) -8

Exercise 5.33

When adding two floating point numbers, the number with the smaller exponent is shifted to preserve the most significant bits. For example, suppose we were adding the two floating point numbers \(1.0 \times 2^0\) and \(1.0 \times 2^{-27}\). 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 \((1.00000000000000000000000000000000 \times 2^{-27})\) becomes \(0.00000000000000000000000000000000 \times 2^0\), because the 1 is shifted off to the right. If we had shifted the number with the larger exponent \((1.0 \times 2^0)\) 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 5.35

(a)
\[0xC0D20004 = 1100000110100100000000000100\]
\[= -1.101001000000000001 \times 2^2\]
\[0x72407020 = 01110010010000001110000000010000\]
\[= 1.10000000111000000000000001 \times 2^{101}\]

When adding these two numbers together, \(0xC0D20004\) becomes:

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

\[0x72407020\]

(b)
\[0xC0D20004 = 11000001101001000000000000000100\]
\[= -1.101001000000000000000000001 \times 2^2\]
\[0x40DC0004 = 01000001101110000000000000000100\]
\[= 1.101110000000000000000000001 \times 2^2\]

\[1.10111000000000000000000001 \times 2^2\]
- 1.101 0010 0000 0000 0000 01 \times 2^2
= 0.000 1010 \times 2^2
= 1.010 \times 2^{-2}

= 0 0111 1101 010 0000 0000 0000 0000
= 0x3EA00000

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

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

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

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

**Exercise 5.37**

(a) \(2(2^{31} - 1 - 2^{23}) = 2^{32} - 2 - 2^{24} = 4,278,190,078\)
(b) \(2(2^{31} - 1) = 2^{32} - 2 = 4,294,967,294\)

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

**Exercise 5.39**
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, shmant:  in  STD_LOGIC_VECTOR(7 downto 0);
         exp, 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(7 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(7 downto 0);
  signal fract: STD_LOGIC_VECTOR(22 downto 0);

begin

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

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

end;
SystemVerilog

module expcomp(input logic [7:0] expa, expb, 
          output logic alessb, 
          output logic [7:0] exp, shamt); 
logic [7:0] aminusb, bminusa;
assign aminusb = expa - expb;
assign bminusa = expb - expa;
assign alessb  = aminusb[7];
always_comb 
  if (alessb) begin
    exp = expb;
    shamt = bminusa;
  end 
  else begin
    exp = expa;
    shamt = aminusb;
  end 
endmodule

VHDL

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

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

architecture synth of expcomp is 
  signal aminusb: STD_LOGIC_VECTOR(7 downto 0);
  signal bminusa: STD_LOGIC_VECTOR(7 downto 0);
begin 
  aminusb <= expa - expb;
  bminusa <= expb - expa;
  alessb <= aminusb[7];
  exp <= expb when alessb = '1' else expa;
  shamt <= bminusa when alessb = '1' else aminusb;
end;
(continued from previous page)

### SystemVerilog

```verilog
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
        (shamt[4] & shamt[3]))
      shmant = 24'b0;
    else
      shmant = shiftedval;
endmodule
```

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

  logic [24:0] addresult;
  logic [23:0] addval;
  assign addval = alessb ? mantb : manta;
  assign addresult = shmant + addval;
  assign fract = addresult[24] ?
                  addresult[23:1] :
                  addresult[22:0];
  assign exp = addresult[24] ?
               (exp_pre + 1) :
               exp_pre;
endmodule
```

### VHDL

```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 shamt_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;
```

```vhdl
library IEEE; use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;

entity addmant is
  port(alessb: in STD_LOGIC;
       manta:  in  STD_LOGIC_VECTOR(23 downto 0);
       mantb:  in  STD_LOGIC_VECTOR(23 downto 0);
       shmant:  in  STD_LOGIC_VECTOR(23 downto 0);
       exp_pre:  in  STD_LOGIC_VECTOR(7 downto 0);
       fract:  out STD_LOGIC_VECTOR(22 downto 0);
       exp:  out  STD_LOGIC_VECTOR(7 downto 0));
end;
architecture synth of addmant is
  signal addresult: STD_LOGIC_VECTOR(24 downto 0);
  begin
    addresult <= 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 5.41

(a) Figure on next page
5.41 (b)

**SystemVerilog**

```verilog
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**

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

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

architecture synth of prefixadd is

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

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

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

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

pik_1 <=
  (p(30)&p(28)&p(26)&p(24)&p(22)&p(20)&p(18)&p(16)&
   p(14)&p(12)&p(10)&p(8)&p(6)&p(4)&p(2)&p(0));
gik_1 <=
  (g(30)&g(28)&g(26)&g(24)&g(22)&g(20)&g(18)&g(16)&
   g(14)&g(12)&g(10)&g(8)&g(6)&g(4)&g(2)&g(0));

pkj_1 <=
  (p(29)&p(27)&p(25)&p(23)&p(21)&p(19)&p(17)&p(15)&
   p(13)&p(11)&p(9)&p(7)&p(5)&p(3)&p(1)&'0');
gkj_1 <=
  (g(29)&g(27)&g(25)&g(23)&g(21)&g(19)&g(17)&g(15)&
   g(13)&g(11)&g(9)&g(7)&g(5)&g(3)&g(1)&cin);

dead1: pgblackblock
  port map(pik_1, gik_1, pkj_1, gkj_1,
           p1, g1);
```

(continued on next page)

**SystemVerilog**

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

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

**VHDL**

```
{2{p1[14]}}, {2{p1[12]}}, {2{p1[10]}},
{2{p1[8]}}, {2{p1[6]}}, {2{p1[4]}},
{2{p1[2]}}, {2{p1[0]}},
{2{g1[14]}}, {2{g1[12]}}, {2{g1[10]}},
{2{g[1][8]}}, {2{g[1][6]}}, {2{g1[4]}},
{2{g1[2]}}, {2{g1[0]}},
p2, g2);
```

```
{4{p2[13]}}, {4{p2[9]}}, {4{p2[5]}},
{4{p2[1]}},
{4{g2[13]}}, {4{g2[9]}}, {4{g2[5]}}},
{4{g2[1]}},
p3, g3);
```

```
{2{g1[14]}}, {2{g1[12]}}, {2{g1[10]}},
{2{g[1][8]}}, {2{g[1][6]}}, {2{g1[4]}},
{2{g1[2]}}, {2{g1[0]}},
gk_2 <=
g[1][8] & g[1][8] & g[1][6] & g[1][6] & g[1][4] & g[1][4] &
g[1][2] & g[1][2] & g1[0] & g1[0],
{4{g2[13]}}, {4{g2[9]}}, {4{g2[5]}}},
{4{g2[1]}},
p3, g3);
```

```
{4{p2[13]}}, {4{p2[9]}}, {4{p2[5]}},
{4{p2[1]}},
{4{g2[13]}}, {4{g2[9]}}, {4{g2[5]}}},
{4{g2[1]}},
p3, g3);
```

```
{2{g1[14]}}, {2{g1[12]}}, {2{g1[10]}},
{2{g[1][8]}}, {2{g[1][6]}}, {2{g1[4]}},
{2{g1[2]}}, {2{g1[0]}},
gk_3 <=
g[1][8] & g[1][8] & g[1][6] & g[1][6] & g[1][4] & g[1][4] &
g[1][2] & g[1][2] & g1[0] & g1[0],
{4{g2[13]}}, {4{g2[9]}}, {4{g2[5]}}},
{4{g2[1]}},
p3, g3);
```

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

blackbox row4({p[15:12],p2[13:12], p[12],p[23],p3[7:4], p[2][5:4],p[1][4],p[7]},
{0{p3[11]}},{8{p3[3]}},
{g3[15:12],g2[13:12],
g1[12],g[23],g[3][7:4],
g[5:4],g[1][4],g[7]},
{8{g3[11]}},{8{g3[3]}},
p4, g4);

blackbox row5({p[15:8],p3[11:8],p2[9:8],
p[8],p[15]},
{16{p4[7]}},
{g4[15:8],g3[11:8],g2[9:8],
g[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)&
p[5][4]&p[1][4]&p(7);
gik_4 <= g3(15 downto 12)&g2(13 downto 12)&
g1(12)&g(23)&g[3][7:4]&
g[5][4]&g[1][4]&g(7);
pkj_4 <= p3(11)&p3(11)&p3(11)&p3(11)&p3(7)&p3(7)&p3(7)&p3(7);
gkj_4 <= g3(11)&g3(11)&g3(11)&g3(11)&g[3][7:4]&g[3][7:4]&g[3][7:4]&g[3][7:4];

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);
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)));
ed;}
SystemVerilog

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

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

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

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

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

    assign s = a ^ b ^ g;
endmodule

VHDL

library IEEE; use IEEE.STD_LOGIC_1164.all;

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

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

library IEEE; use IEEE.STD_LOGIC_1164.all;

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

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

library IEEE; use IEEE.STD_LOGIC_1164.all;

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

architecture synth of sumblock is
begin
    s <= a xor b xor g;
end;
5.41 (c) Using Equation 5.11 to find the delay of the prefix adder:

\[ t_{PA} = t_{pg} \log_2 N(t_{pg \_prefix}) + t_{XOR} \]

We find the delays for each block:

- \( t_{pg} = 100 \) ps
- \( t_{pg \_prefix} = 200 \) ps
- \( t_{XOR} = 100 \) ps

Thus,

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

5.41 (d) To make a pipelined prefix adder, add pipeline registers between each of the rows of the prefix adder. Now each stage will take 200 ps plus the
sequencing overhead, $t_{pq} + t_{\text{setup}} = 80$ ps. Thus each cycle is 280 ps and the design can run at 3.57 GHz.
5.41 (e)

**SystemVerilog**

module prefixaddpipe(input logic clk, cin, a, b, output logic s, output cout);

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

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

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

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

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

// row 2
flop #(2) flop2_pg_1(clk, {p_1_1,g_1_1}, {p_1_2,g_1_2});
flop #(30) flop2_pg(clk, {p1[28:27],p1[24:23],p1[20:19],p1[16:15],p1[12:11],
p1[8:7],p1[4:3],p1[0]},
{p2[29],p2[27],p2[25],p2[23],p2[21],p2[19],p2[17],p2[15],p2[13],p2[11],p2[9],p2[7],p2[5],p2[3],p2[1],1'b0},
{g2[29],g2[27],g2[25],g2[23],g2[21],g2[19],g2[17],g2[15],g2[13],g2[11],g2[9],g2[7],g2[5],g2[3],g2[1]},
{p2[30],p2[28],p2[26],p2[24],p2[22],p2[20],
p2[18],p2[16],p2[14],p2[12],p2[10],p2[8],
p2[6],p2[4],p2[2],p2[0]},
{g2[30],g2[28],g2[26],g2[24],g2[22],g2[20],
g2[18],g2[16],g2[14],g2[12],g2[10],g2[8],
g2[6],g2[4],g2[2],g2[0]});
166 SOLUTIONS chapter 5

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,
  { 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]}  },

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

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

// row 4
flop #(2) flop4_pg_1(clk, {p_1_3, g_1_3}, {p_1_4, g_1_4});
flop #(30) flop4_pg(clk, {p3[22:15], p3[6:0],
  g3[22:15], g3[6:0]},
  {p4[22:15], p4[6:0]},
g4[22:15], g4[6:0]});
blackbox row4(clk,
  {p3[30:23], p3[14:7]},
  { 8{p3[22]}, 8{p3[6]}  },
  {g3[30:23], g3[14:7]},
  { 8{g3[22]}, 8{g3[6]}  },
  {p4[30:23], p4[14:7]},
  {g4[30:23], g4[14:7]});

// row 5
flop #(2) flop5_pg_1(clk, {p_1_4, g_1_4}, {p_1_5, g_1_5});
flop #(30) flop5_pg(clk, {p4[14:0], g4[14:0]},
  {p5[14:0], g5[14:0]});
blackbox row5(clk, 
   p4[30:15],
   {16(p4[14])},
   g4[30:15],
   {16(g4[14])},
   p5[30:15], g5[30:15]);

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

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

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

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

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

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

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

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

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

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

VHDL

library IEEE; use IEEE.STD_LOGIC_1164.all;

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

architecture synth of prefixaddpipe is
  component pgblock
    port(clk:  in  STD_LOGIC;
         a, b: in  STD_LOGIC_VECTOR(30 downto 0);
         p, g: out STD_LOGIC_VECTOR(30 downto 0));
  end component;
  component sumblock
    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
    port(clk: in  STD_LOGIC;
         d:  in  STD_LOGIC;
         q:  out STD_LOGIC);
  end component;
  component row1
    port(clk:  in  STD_LOGIC;
         p0, g0: in  STD_LOGIC_VECTOR(30 downto 0);
         p_1_0, g_1_0: in STD_LOGIC;
         p1, g1: out STD_LOGIC_VECTOR(30 downto 0));
  end component;
  component row2
    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
    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
    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
    port(clk:  in  STD_LOGIC;
         p4, g4: in  STD_LOGIC_VECTOR(30 downto 0);
         p5, g5: out STD_LOGIC_VECTOR(30 downto 0));
  end component;
-- p and g prefixes for rows 0 - 5
signal p0, p1, p2, p3, p4, p5: STD_LOGIC_VECTOR(30 downto 0);
signal g0, g1, g2, g3, g4, g5: STD_LOGIC_VECTOR(30 downto 0);

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

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

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

begin

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

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

-- pipeline p and g for column -1
p_1_0 <= '0'; flop_1_g0: flop1 port map (clk, cin, g_1_0);
flop1_p1: flop1 port map (clk, p_1_0, p_1_1);
flop1_g1: flop1 port map (clk, g_1_0, g_1_1);
flop1_p2: flop1 port map (clk, p_1_1, p_1_2);
flop1_g2: flop1 port map (clk, g_1_1, g_1_2);
flop1_p3: flop1 port map (clk, p_1_2, p_1_3); flop1_g3:
flop1 port map (clk, g_1_2, g_1_3);
flop1_p4: flop1 port map (clk, p_1_3, p_1_4);
flop1_g4: flop1 port map (clk, g_1_3, g_1_4);
flop1_p5: flop1 port map (clk, p_1_4, p_1_5);
flop1_g5: flop1 port map (clk, g_1_4, g_1_5);

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

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

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity pgblock is
  port(clk: in STD_LOGIC;
a, b: in STD_LOGIC_VECTOR(30 downto 0);
p, g: out STD_LOGIC_VECTOR(30 downto 0));
end;

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

library IEEE; use IEEE.STD_LOGIC_1164.all;

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

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

library IEEE; use IEEE.STD_LOGIC_1164.all;

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

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

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

architecture synth of flop is
begin
process(clk) begin
if rising_edge(clk) then
q <= d;
end;
library IEEE; use IEEE.STD_LOGIC_1164.all; use IEEE.STD_LOGIC_ARITH.all;
entity flop1 is -- 1-bit flip flop
port(clk: in STD_LOGIC;
    d: in STD_LOGIC;
    q: out STD_LOGIC);
end;

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

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity row1 is
port(clk: in STD_LOGIC;
    p0, g0: in STD_LOGIC_VECTOR(30 downto 0);
    p_1_0, g_1_0: in STD_LOGIC;
    p1, g1: out STD_LOGIC_VECTOR(30 downto 0));
end;

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

-- internal signals for calculating p, g
signal pik_0, gik_0, pkj_0, gkj_0,
    pij_0, gij_0: STD_LOGIC_VECTOR(15 downto 0);
-- internal signals for pipeline registers
signal pg0_in, pg1_out: STD_LOGIC_VECTOR(29 downto 0);

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

    p1(29) <= pgl_out(29); p1(27) <= pgl_out(28); p1(25) <= pgl_out(27);
    p1(23) <= pgl_out(26);
    p1(21) <= pgl_out(25); p1(19) <= pgl_out(24); p1(17) <= pgl_out(23);
    p1(15) <= pgl_out(22); p1(13) <= pgl_out(21); p1(11) <= pgl_out(20);
    p1(9) <= pgl_out(19); p1(7) <= pgl_out(18); p1(5) <= pgl_out(17);
    p1(3) <= pgl_out(16); p1(1) <= pgl_out(15);
    gl(29) <= pgl_out(14); gl(27) <= pgl_out(13); gl(25) <= pgl_out(12);
    gl(23) <= pgl_out(11); gl(21) <= pgl_out(10); gl(19) <= pgl_out(9);
    gl(17) <= pgl_out(8); gl(15) <= pgl_out(7); gl(13) <= pgl_out(6);
\begin{verbatim}
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
begin
    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);
end;
\end{verbatim}
```
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));

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

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

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

```
end;
```

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

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

```
architecture synth of row3 is
  component blackbox is
    port (clk:    in STD_LOGIC;
          pik, pkj: in STD_LOGIC_VECTOR(15 downto 0);
          gik, gkj: in STD_LOGIC_VECTOR(15 downto 0);
          pij, 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));
  end component;
end;
library IEEE; use IEEE.STD_LOGIC_1164.all;

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

architecture synth of row4 is
    component blackbox is
        port (clk:      in  STD_LOGIC;
              pik, pkj: in STD_LOGIC_VECTOR(15 downto 0);
              gik, gkj: in STD_LOGIC_VECTOR(15 downto 0);
              pij:     out STD_LOGIC_VECTOR(15 downto 0);
              gi2:     out STD_LOGIC_VECTOR(15 downto 0));
    end component;

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

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

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

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

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

    row4: blackbox
    port map(clk, pik_3, pkj_3, gik_3, gkj_3, pij_3, gij_3);
    p4(30 downto 23) <= pij_3(15 downto 8);
    p4(14 downto 7) <= pij_3(7 downto 0);
    g4(30 downto 23) <= gij_3(15 downto 8);
    g4(14 downto 7) <= gij_3(7 downto 0);

end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

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

architecture synth of row5 is
    component blackbox is
    port (clk:      in  STD_LOGIC;
          pik, pkj: in STD_LOGIC_VECTOR(15 downto 0);
          gik, gkj: in STD_LOGIC_VECTOR(15 downto 0);
          pij:      out STD_LOGIC_VECTOR(15 downto 0);
          gij:      out STD_LOGIC_VECTOR(15 downto 0));
end component;

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

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

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

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

-- pg calculations
pik_4 <= p4(30 downto 15);
gik_4 <= g4(30 downto 15);
pkj_4 <= p4(14)&p4(14)&p4(14)&p4(14)&p4(14)&p4(14)&p4(14)&p4(14);
gkj_4 <= 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 5.43

Exercise 5.45
FIGURE 5.8 32-bit counter that increments by 4 or loads a new value, \( D \)

**Exercise 5.47**

**SystemVerilog**

```verilog
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**

```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 5.49

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

Flash memory is a nonvolatile memory because it retains its contents after power is turned off. Flash memory allows the user to electrically program and erase information. Flash memory uses memory cells similar to an EEPROM, but with a much thinner, precisely grown oxide between a floating gate and the substrate (see Figure 5.9).

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 5.51

(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 5.55

(a) 1 LE

(b) 2 LEs
(c) 2 LEs

<table>
<thead>
<tr>
<th></th>
<th>(A)</th>
<th>(B)</th>
<th>(C)</th>
<th>(D)</th>
<th>LUT output</th>
<th>(A)</th>
<th>(B)</th>
<th>(C)</th>
<th>(D)</th>
<th>LUT output</th>
</tr>
</thead>
<tbody>
<tr>
<td>data 1</td>
<td>data 2</td>
<td>data 3</td>
<td>data 4</td>
<td></td>
<td></td>
<td>data 1</td>
<td>data 2</td>
<td>data 3</td>
<td>data 4</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
(d) 2 LEs

<table>
<thead>
<tr>
<th>(A₂)</th>
<th>(A₃)</th>
<th>(A₁)</th>
<th>(A₀)</th>
<th>LUT output</th>
<th>(D)</th>
<th>(A₂)</th>
<th>(A₃)</th>
<th>(A₁)</th>
<th>(A₀)</th>
<th>LUT output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

For LE 1:

For LE 2:
Exercise 5.57

(e) 2 LEs

<table>
<thead>
<tr>
<th>(A&lt;sub&gt;j&lt;/sub&gt;)</th>
<th>(A&lt;sub&gt;k&lt;/sub&gt;)</th>
<th>(A&lt;sub&gt;l&lt;/sub&gt;)</th>
<th>(A&lt;sub&gt;m&lt;/sub&gt;)</th>
<th>LUT output</th>
</tr>
</thead>
<tbody>
<tr>
<td>data 1</td>
<td>data 2</td>
<td>data 3</td>
<td>data 4</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(A&lt;sub&gt;j&lt;/sub&gt;)</th>
<th>(A&lt;sub&gt;k&lt;/sub&gt;)</th>
<th>(A&lt;sub&gt;l&lt;/sub&gt;)</th>
<th>(A&lt;sub&gt;m&lt;/sub&gt;)</th>
<th>LUT output</th>
</tr>
</thead>
<tbody>
<tr>
<td>data 1</td>
<td>data 2</td>
<td>data 3</td>
<td>data 4</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Exercise 5.57

(a) 5 LEs (2 for next state logic and state registers, 3 for output logic)
(b) 
\[ t_{pd} = t_{pd,LE} + t_{wire} \]
\[ = (381 + 246) \text{ ps} \]
\[ = 627 \text{ ps} \]
\[ T_c \geq t_{pcq} + t_{pd} + t_{\text{setup}} \]
\[ \geq [199 + 627 + 76] \text{ ps} \]
\[ = 902 \text{ ps} \]
\[ f = \frac{1}{902 \text{ ps}} = 1.1 \text{ GHz} \]
(c) 
First, we check that there is no hold time violation with this amount of clock skew.
\[ t_{cd,LE} = t_{pd,LE} = 381 \text{ ps} \]
\[ t_{cd} = t_{cd,LE} + t_{\text{wire}} = 627 \text{ ps} \]
Exercise 5.59

First, we find the cycle time:

\[ T_c = \frac{1}{f} = \frac{1}{100 \text{ MHz}} = 10 \text{ ns} \]

\[ T_c \geq t_{pcq} + Nt_{LE+wire} + t_{\text{setup}} \]

\[ 10 \text{ ns} \geq [0.199 + N(0.627) + 0.076] \text{ ns} \]

Thus, \( N \leq 15.5 \)

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

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

\[ T_c \geq [0.199 + 0.627 + 0.076] \text{ ns} \]

\[ \geq 0.902 \text{ ns} \]

\[ f = \frac{1}{0.902 \text{ ns}} = 1.1 \text{ GHz} \]

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
SystemVerilog

```verilog
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];
endmodule
```

VHDL

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

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

architecture synth of bcdadd_8 is
  component bcdadd_4
    port(a, b: in STD_LOGIC_VECTOR(3 downto 0);
         cin:  in STD_LOGIC;
         s:    out STD_LOGIC_VECTOR(3 downto 0);
         cout: out STD_LOGIC);
  end component;
  signal c0: STD_LOGIC;
begin
  bcd0: bcdadd_4
    port map(a(3 downto 0), b(3 downto 0), cin, s(3 downto 0), c0);
bcd1: bcdadd_4
    port map(a(7 downto 4), b(7 downto 4), c0, s(7 downto 4), cout);
end;

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

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

architecture synth of bcdadd_4 is
  signal result, sub10, a5, b5: STD_LOGIC_VECTOR(4 downto 0);
begin
  a5 <= '0' & a;
b5 <= '0' & b;
  result <= a5 + b5 + cin;
  sub10 <= result - "01010";
  cout <= not (sub10(4));
s <= result(3 downto 0) when sub10(4) = '1'
                 else sub10(3 downto 0);
end;
```
**Exercise 6.1**

(1) Simplicity favors regularity:
- Each instruction has a 6-bit opcode.
- MIPS has only 3 instruction formats (R-Type, I-Type, J-Type).
- Each instruction format has the same number and order of operands (they differ only in the opcode).
- Each instruction is the same size, making decoding hardware simple.

(2) Make the common case fast.
- Registers make the access to most recently accessed variables fast.
- The RISC (reduced instruction set computer) architecture, makes the common/simple instructions fast because the computer must handle only a small number of simple instructions.
- 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.

(3) Smaller is faster.
- The register file has only 32 registers.
The ISA (instruction set architecture) includes only a small number of commonly used instructions. This keeps the hardware small and, thus, fast.

The instruction size is kept small to make instruction fetch fast.

(4) Good design demands good compromises.

MIPS uses three instruction formats (instead of just one).

Ideally all accesses would be as fast as a register access, but MIPS architecture also supports main memory accesses to allow for a compromise between fast access time and a large amount of memory.

Because MIPS is a RISC architecture, it includes only a set of simple instructions, it provides pseudocode to the user and compiler for commonly used operations, like moving data from one register to another (move) and loading a 32-bit immediate (li).

Exercise 6.3

(a) $42 \times 4 = 42 \times 2^2 = 101010_2 \ll 2 = 10101000_2 = 0xA8$
(b) 0xA8 through 0xAB
(c)

<table>
<thead>
<tr>
<th>Byte Address</th>
<th>Data Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big-Endian</td>
<td>FF 22 33 44</td>
</tr>
<tr>
<td>Word Address</td>
<td>0xA8</td>
</tr>
<tr>
<td>Little-Endian</td>
<td>FF 22 33 44</td>
</tr>
<tr>
<td>Byte Address</td>
<td>0x98</td>
</tr>
</tbody>
</table>

Exercise 6.5

```
# Big-endian
li  $t0, 0xABCD9876
sw  $t0, 100($0)
lb  $s5, 101($0)  # the LSB of $s5 = 0xCD
```

```
# Little-endian
li  $t0, 0xABCD9876
sw  $t0, 100($0)
lb  $s5, 101($0)  # the LSB of $s5 = 0x98
```

In big-endian format, the bytes are numbered from 100 to 103 from left to right. In little-endian format, the bytes are numbered from 100 to 103 from right to left. Thus, the final load byte (lb) instruction returns a different value depending on the endianness of the machine.
Exercise 6.7

(a) 0x68 6F 77 64 79 00
(b) 0x6C 69 6F 6E 73 00
(c) 0x54 6F 20 74 68 65 20 72 65 73 63 75 65 21 00

Exercise 6.9

Little-Endian Memory

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Data</th>
<th>Word Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001010</td>
<td>00 79</td>
<td>10001010</td>
<td>00 73</td>
</tr>
<tr>
<td>1000100C</td>
<td>64 77 6F 68</td>
<td>1000100C</td>
<td>6E 6F 69 6C</td>
</tr>
</tbody>
</table>

Big-Endian Memory

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Data</th>
<th>Word Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001010</td>
<td>79 00</td>
<td>10001010</td>
<td>73 00</td>
</tr>
<tr>
<td>1000100C</td>
<td>68 6F 77 64</td>
<td>1000100C</td>
<td>6C 69 6F 6E</td>
</tr>
</tbody>
</table>

Exercise 6.11

0x20100049
0xad49fff9
0x02f24822

Exercise 6.13

(a)
addi $s0, $0, 73
sw $t1, -7($t2)
(b)
0x20100049 (addi)
0xad49fff9 (sw)

Exercise 6.15

```
addi $t0, $0, 31
L1:
srlv $t1, $a0, $t0
andi $t1, $t1, 1
slt $t1, $0, $t1
sb $t1, 0($a1)
addi $a1, $a1, 1
addi $t0, $t0, -1
bgez $t0, L1
jr $ra
```

(a) This program converts a number ($a0) from decimal to binary and stores it in an array pointed to by $a1.

```c
void convert2bin(int num, char binarray[])
{
  int i;
  char tmp, val = 31;

  for (i=0; i<32; i++) {
    tmp = (num >> val) & 1;
    binarray[i] = tmp;
    val--;
  }
}
```

Exercise 6.17

(a)
```
# $s0 = g, $s1 = h
slt $t0, $s1, $s0          # if h < g, $t0 = 1
beq $t0, $0, else          # if $t0 == 0, do else
add $s0, $s0, $s1          # g = g + h
j done                     # jump past else block
else:sub $s0, $s0, $s1      # g = g - h
done:
```

(b)
Exercise 6.19

(a)

```
# MIPS assembly code
# base address of array dst = $a0
# base address of array src = $a1
# i = $s0

strcpy:
    addi $sp, $sp, -4
    sw  $s0, 0($sp)   # save $s0 on the stack
    add  $s0, $0, $0   # i = 0

loop:
    add  $t1, $a1, $s0 # $t1 = address of src[i]
    lb   $t2, 0($t1)   # $t2 = src[i]
    add  $t3, $a0, $s0 # $t3 = address of dst[i]
    sb   $t2, 0($t3)   # dst[i] = src[i]
    beq  $t2, $0, done # check for null character
    addi $s0, $s0, 1   # i++
    j    loop

done:
    lw   $s0, 0($sp)   # restore $s0 from stack
    addi $sp, $sp, 4   # restore stack pointer
    jr   $ra           # return
```
Exercise 6.21

(a) The stack frames of each procedure are:
proc1: 3 words deep (for $s0 - $s1, $ra)
proc2: 7 words deep (for $s2 - $s7, $ra)
proc3: 4 words deep (for $s1 - $s3, $ra)
proc4: 0 words deep (doesn't use any saved registers or call other procedures)
(b) Note: we arbitrarily chose to make the initial value of the stack pointer 0x7FFFFFFF04 just before the procedure calls.

Exercise 6.23

(a) 120
(b) 2
(c) (i) 3 – returned value is 1
(ii) 2 (depending on what’s stored on the stack frame of the callee’s stack)
(iii) 4

Exercise 6.25

(a) 000100 01000 10001 0000 0000 0000 0010
   = 0x11110002
(b) 000100 01111 10100 0000 0100 0000 1111
   = 0x11F4040F
(c) 000100 11001 10111 1111 1000 0100 0010
   = 0x1337F842
(d) 000011 0000 0100 0001 0001 0100 0111 11
   = 0x0C10451F
(e) \[000010 \; 00 \; 0001 \; 0000 \; 0000 \; 1100 \; 0000 \; 0001\]
\[= \text{0x08100C01}\]

Exercise 6.27

(a)

```
set_array:  addi $sp,$sp,-52  # move stack pointer
            sw $ra,48($sp)   # save return address
            sw $s0,44($sp)   # save $s0
            sw $s1,40($sp)   # save $s1
            add $s0,$0,$0     # i = 0
            addi $s1,$0,10    # max iterations = 10
loop:      add $a1,$s0,$0    # pass i as parameter
            jal compare      # call compare(num, i)
            ll $t1,$s0,2     # $t1 = i*4
            add $t2,$sp,$t1   # $t2 = address of array[i]
            sw $v0,0($t2)    # array[i] = compare(num, i);
            addi $s0,$s0,1    # i++
            bne $s0,$s1,loop  # if i<10, goto loop
            lw $s1,40($sp)    # restore $s1
            lw $s0,44($sp)    # restore $s0
            lw $ra,48($sp)    # restore return address
            addi $sp,$sp,52    # restore stack pointer
            jr $ra  # return to point of call

compare:   addi $sp,$sp,-4    # move stack pointer
            sw $ra,0($sp)     # save return address on the stack
            jal subtract     # input parameters already in $a0,$a1
            slt $v0,$v0,$0    # $v0=1 if sub(a,b) < 0 (return 0)
            slti $v0,$v0,1    # $v0=1 if sub(a,b)>=0, else $v0 = 0
            lw $ra,0($sp)     # restore return address
            addi $sp,$sp,4     # restore stack pointer
            jr $ra  # return to point of call

subtract:  sub $v0,$a0,$a1    # return a-b
            jr $ra  # return to point of call
```
6.27 (b)

(c) If $ra$ were never stored on the stack, the compare function would return to the instruction after the call to `subtract (slt $v0,$v0,$0)` instead of returning to the `set_array` function. The program would enter an infinite loop in the `compare` function between `jr $ra` and `slt $v0, $v0, $0`. It would increment the stack during that loop until the stack space was exceeded and the program would likely crash.

**Exercise 6.29**

Instructions (32 K - 1) words before the branch to instructions 32 K words after the branch instruction.
Exercise 6.31

It is advantageous to have a large address field in the machine format for jump instructions to increase the range of instruction addresses to which the instruction can jump.

Exercise 6.33

```c
# high-level code
void little2big(int[] array) {
    int i;
    for (i = 0; i < 10; i = i + 1) {
        array[i] = ((array[i] << 24) | (array[i] & 0xFF00) << 8) | (array[i] & 0xFF0000) >> 8) | (array[i] >> 24) & 0xFF);
    }
}
```

```mips
# MIPS assembly code
# $a0 = base address of array
little2big:
    addi $t5, $0, 10  # $t5 = i = 10 (loop counter)
    loop: lb   $t0, 0($a0)  # $t0 = array[i] byte 0
           lb   $t1, 1($a0)  # $t1 = array[i] byte 1
           lb   $t2, 2($a0)  # $t2 = array[i] byte 2
           lb   $t3, 3($a0)  # $t3 = array[i] byte 3
           sb   $t3, 0($a0)  # array[i] byte 0 = previous byte 3
           sb   $t2, 1($a0)  # array[i] byte 1 = previous byte 2
           sb   $t1, 2($a0)  # array[i] byte 2 = previous byte 1
           sb   $t0, 3($a0)  # array[i] byte 3 = previous byte 0
           addi $a0, $a0, 4  # increment index into array
           addi $t5, $t5, -1 # decrement loop counter
           beq  $t5, $0, done
           j    loop
    done: jr   $ra
```

Exercise 6.35

```mips
# define the masks in the global data segment
.data
    mmask:  .word 0x007FFFFF
    emask:  .word 0x7F800000
    ibit:   .word 0x00800000
    obit:   .word 0x01000000
.text
    f1padd: lw $t4,mmask  # load mantissa mask
           and $t0,$s0,$t4  # extract mantissa from $s0 (a)
           and $t1,$s1,$t4  # extract mantissa from $s1 (b)
           lw $t4,ibit   # load implicit leading 1
           or $t0,$t0,$t4  # add the implicit leading 1 to mantissa
           or $t1,$t1,$t4  # add the implicit leading 1 to mantissa
```
Exercise 6.37

(a)

```
0x00400000  main:        addi $sp, $sp, -4
0x00400004  sw   $ra, 0($sp)
0x00400008  addi $t0, $0, 15
0x0040000C  sw   $t0, 0x8000($gp)
0x00400010  addi $a1, $0, 27
0x00400014  sw   $a1, 0x8004($gp)
0x00400018  lw   $a0, 0x8000($gp)
0x0040001C  jal  greater
0x00400020  lw   $ra, 0($sp)
0x00400024  addi $sp, $sp, 4
0x00400028  jr   $ra
0x0040002C  greater:     slt  $v0, $a1, $a0
0x00400030  jr   $ra
```

(b)

<table>
<thead>
<tr>
<th>symbol</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0x10000000</td>
</tr>
<tr>
<td>b</td>
<td>0x10000004</td>
</tr>
<tr>
<td>main</td>
<td>0x00400000</td>
</tr>
<tr>
<td>greater</td>
<td>0x0040002C</td>
</tr>
</tbody>
</table>
TABLE 6.1 Symbol table

<table>
<thead>
<tr>
<th>Executable file header</th>
<th>Text Size</th>
<th>Data Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x34 (52 bytes)</td>
<td>0x8 (8 bytes)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Text segment</th>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400000</td>
<td>0x23BDFFFC</td>
<td>addi $sp, $sp, -4</td>
</tr>
<tr>
<td>0x00400004</td>
<td>0xAFBD0000</td>
<td>sw $ra, 0($sp)</td>
</tr>
<tr>
<td>0x00400008</td>
<td>0x2008000F</td>
<td>addi $t0, $0, 15</td>
</tr>
<tr>
<td>0x0040000C</td>
<td>0xAF880000</td>
<td>sw $t0, 0x8000($gp)</td>
</tr>
<tr>
<td>0x00400010</td>
<td>0x2005001B</td>
<td>addi $a1, $0, 27</td>
</tr>
<tr>
<td>0x00400014</td>
<td>0xAF580004</td>
<td>sw $a1, 0x8000($gp)</td>
</tr>
<tr>
<td>0x00400018</td>
<td>0x8F848000</td>
<td>lw $a0, 0x8000($gp)</td>
</tr>
<tr>
<td>0x0040001C</td>
<td>0x0C10000B</td>
<td>jal greater</td>
</tr>
<tr>
<td>0x00400020</td>
<td>0x8F8F0000</td>
<td>lw $ra, 0($sp)</td>
</tr>
<tr>
<td>0x00400024</td>
<td>0x23BD0004</td>
<td>addi $sp, $sp, 4</td>
</tr>
<tr>
<td>0x00400028</td>
<td>0x03E00008</td>
<td>jr $ra</td>
</tr>
<tr>
<td>0x0040002C</td>
<td>0x00A4102A</td>
<td>slt $v0, $a1, $a0</td>
</tr>
<tr>
<td>0x00400030</td>
<td>0x03E0000B</td>
<td>jr $ra</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data segment</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10000000</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>0x10000004</td>
<td>b</td>
<td></td>
</tr>
</tbody>
</table>

(d) The data segment is 8 bytes and the text segment is 52 (0x34) bytes.
### Exercise 6.39

(a)

```
beq $t1, imm31:0, L
lui $at, imm31:16
ori $at, $at, imm15:0
beq $t1, $at, L
```

(b)

```
bie $t3, $t5, L
slt $at, $t5, $t3
beq $at, $0, L
```
Question 6.1

xor $t0, $t0, $t1  # $t0 = $t0 XOR $t1
xor $t1, $t0, $t1  # $t1 = original value of $t0
xor $t0, $t0, $t1  # $t0 = original value of $t1

Question 6.3
### High-Level Code

```c
// high-level algorithm
void reversewords(char[] array) {
    int i, j, length;

    // find length of string
    for (i = 0; array[i] != 0; i = i + 1) {
        length = i;
    }

    // reverse characters in string
    reverse(array, length-1, 0);

    // reverse words in string
    i = 0; j = 0;
    // check for spaces
    while (i <= length) {
        if ( (i != length) || (array[i] != 0x20) ) {
            i = i + 1;
        } else {
            reverse(array, i-1, j);
            i = i + 1; // j and i at start of next word
            j = i;
        }
    }
}

void reverse(char[] array, int i, int j) {
    char tmp;
    while (i > j) {
        tmp = array[i];
        array[i] = array[j];
        array[j] = tmp;
        i = i-1;
        j = j+1;
    }
}
```

### MIPS Assembly Code

```assembly
# $s2 = i, $s3 = j, $s1 = length
reversewords:
    addi $sp, $sp, -16  # make room on stack
    sw $ra, 12($sp)    # store regs on stack
    sw $s1, 8($sp)
    sw $s2, 4($sp)
    sw $s3, 0($sp)
    addi $s2, $0, 0  # i = 0
    length: add $t4, $a0, $s2  # $t4 = &array[i]
    lb $t3, 0($t4)  # $t3 = array[i]
    beq $t3, $0, done # end of string?
    addi $s2, $s2, 1  # i++
    j length

    done:   addi $s1, $s2, 0  # length = i
            addi $a1, $s1, -1  # $a1 = length - 1
            addi $a2, $0, 0  # $a2 = 0
            jal reverse  # call reverse
    addi $s2, $0, 0  # i = 0
    addi $s3, $0, 0  # j = 0
    addi $t5, $0, 0x20  # $t5 = "space"
    word:   slt $t4, $s1, $s2  # $t4 = 1 if length<i
            bne $t4, $0, return # return if length<i
            beq $s2, $s1, else  # if i==length, else
            add $t4, $a0, $s2  # $t4 = &array[i]
            lb $t3, 0($t4)  # $t3 = array[i]
            beq $t3, $t5, else  # if $t4==0x20,else
            addi $s2, $s2, 1  # i = i + 1
            j word
    else:   addi $a1, $s2, -1  # $a1 = i - 1
             addi $a2, $s3, 0  # $a2 = j
             jal reverse
             addi $s2, $s2, 1  # i = i + 1
             addi $s3, $s2, 0  # j = i
             j word

    return: lw $ra, 12($sp)  # restore regs
             lw $s1, 8($sp)
             lw $s2, 4($sp)
             lw $s3, 0($sp)
             addi $sp, $sp, 16  # restore $sp
             jr $ra  # return

reverse:
    slt $t0, $a2, $a1  # $t0 = 1 if j < i
    beq $t0, $0, exit  # if j < i, return
    add $t1, $a0, $a1  # $t1 = &array[i]
    lb $t2, 0($t1)  # $t2 = array[i]
    add $t3, $a0, $a2  # $t3 = &array[j]
    lb $t4, 0($t3)  # $t4 = array[j]
    sb $t4, 0($t1)  # array[i] =array[j]
    sb $t2, 0($t3)  # array[j] =array[i]
    addi $s1, $s1, -1  # i = i-1
    addi $a2, $a2, 1  # j = j+1
    j reverse

exit:   jr $ra
```
Question 6.5

High-Level Code

```c
num = swap(num, 1, 0x55555555); // swap bits
num = swap(num, 2, 0x33333333); // swap pairs
num = swap(num, 4, 0x0F0F0F0F); // swap nibbles
num = swap(num, 8, 0x00FF00FF); // swap bytes
num = swap(num, 16, 0xFFFFFFFF); // swap halves

// swap masked bits
int swap(int num, int shamt, unsigned int mask) {
    return ((num >> shamt) & mask) |
            ((num & mask) << shamt);
}
```

MIPS Assembly Code

```mips
# $t3 = num
addi $a0, $t3, 0    # set up args
addi $a1, $0, 1
li   $a2, 0x55555555
jal  swap    # swap bits
addi $a0, $v0, 0    # num = return value
addi $a1, $0, 2    # set up args
li   $a2, 0x33333333
jal  swap    # swap pairs
addi $a0, $v0, 0    # num = return value
addi $a1, $0, 4    # set up args
li   $a2, 0x0F0F0F0F
jal  swap    # swap nibbles
addi $a0, $v0, 0    # num = return value
addi $a1, $0, 8    # set up args
li   $a2, 0x00FF00FF
jal  swap    # swap bytes
addi $a0, $v0, 0    # num = return value
addi $a1, $0, 16   # set up args
li   $a2, 0xFFFFFFFF
jal  swap    # swap halves
addi $t3, $v0, 0    # num = return value

done: j done

swap:
    srlv $v0, $a0, $a1    # $v0 = num >> shamt
    and $v0, $v0, $a2    # $v0 = $v0 & mask
    and $t0, $a0, $a2    # $t0 = num & mask
    slrv $t0, $t0, $a1    # $t0 = $t0 << shamt
    or  $v0, $v0, $t0    # $v0 = $v0 | $t0
    jr  $ra    # return
```
Question 6.7

**High-Level Code**

```c
bool palindrome(char* array) {
    int i, j; // array indices
    // find length of string
    for (j = 0; array[j] != 0; j=j+1) ;
    j = j-1; // j is index of last char
    i = 0;
    while (j > i) {
        if (array[i] != array[j])
            return false;
        j = j-1;
        i = i+1;
    }
    return true;
}
```

**MIPS Assembly Code**

```mips
# $t0 = j, $t1 = i, $a0 = base address of string
palindrome:
    addi $t0, $0, 0      # j = 0
length:    add $t2, $a0, $t0   # $t2 = &array[j]
          lb  $t2, 0($t2)     # $t2 = array[j]
          beq $t2, $0, done   # end of string?
          addi $t0, $t0, 1     # j = j+1
          j    length
done:      addi $t0, $t0, -1    # j = j-1
done:
    addi $t0, $t0, 0      # i = 0
loop:      slt  $t2, $t1, $t0    # $t2 = 1 if i < j
          beq  $t2, $0, yes    # if !(i < j) return
          add  $t2, $a0, $t1    # $t2 = &array[i]
          lb   $t2, 0($t2) # $t2 = array[i]
          add  $t3, $a0, $t0    # $t3 = &array[j]
          lb   $t3, 0($t3) # $t3 = array[j]
          bne  $t2, $t3, no     # is palindrome?
          addi $t0, $t0, -1    # j = j-1
          addi $t1, $t1, 1 # i = i+1
          j    loop
yes:      # yes a palindrome
          addi $v0, $0, 1
          j yes
          jr   $ra
no:       # not a palindrome
          addi $v0, $0, 0
          j no
          jr   $ra
```


Exercise 7.1

(a) R-type, lw, addi
(b) R-type
(c) sw

Exercise 7.3

(a) sll
First, we modify the ALU.

**FIGURE 7.1** Modified ALU to support $\text{sll}$

**TABLE 7.1** Modified ALU operations to support $\text{sll}$

<table>
<thead>
<tr>
<th>ALU Control $3:0$</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>A AND B</td>
</tr>
<tr>
<td>0001</td>
<td>A OR B</td>
</tr>
<tr>
<td>0010</td>
<td>A + B</td>
</tr>
<tr>
<td>0011</td>
<td>not used</td>
</tr>
<tr>
<td>1000</td>
<td>A AND $\overline{B}$</td>
</tr>
<tr>
<td>1001</td>
<td>A OR $\overline{B}$</td>
</tr>
<tr>
<td>1010</td>
<td>A - B</td>
</tr>
<tr>
<td>1011</td>
<td>SLT</td>
</tr>
<tr>
<td>0100</td>
<td>SLL</td>
</tr>
</tbody>
</table>

**TABLE 7.1** Modified ALU operations to support $\text{sll}$
Then we modify the datapath.

**TABLE 7.2 ALU decoder truth table**

<table>
<thead>
<tr>
<th>ALUOp</th>
<th>Funct</th>
<th>ALUControl</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>X</td>
<td>0010 (add)</td>
</tr>
<tr>
<td>X1</td>
<td>X</td>
<td>1010 (subtract)</td>
</tr>
<tr>
<td>1X</td>
<td>100000 (add)</td>
<td>0010 (add)</td>
</tr>
<tr>
<td>1X</td>
<td>10010 (sub)</td>
<td>1010 (subtract)</td>
</tr>
<tr>
<td>1X</td>
<td>100100 (and)</td>
<td>0000 (and)</td>
</tr>
<tr>
<td>1X</td>
<td>100101 (or)</td>
<td>0001 (or)</td>
</tr>
<tr>
<td>1X</td>
<td>101010 (slt)</td>
<td>1011 (set less than)</td>
</tr>
<tr>
<td>1X</td>
<td>000000 (sll)</td>
<td>0100 (shift left logical)</td>
</tr>
</tbody>
</table>

**FIGURE 7.2** Modified single-cycle MIPS processor extended to run sll
7.3 (b) \texttt{lui}

Note: the 5-bit rs field of the \texttt{lui} instruction is 0.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>opcode</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>\texttt{lw}</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>\texttt{sw}</td>
<td>101011</td>
<td>0</td>
<td>X</td>
<td>01</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>00</td>
</tr>
<tr>
<td>\texttt{beq}</td>
<td>000100</td>
<td>0</td>
<td>X</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>01</td>
</tr>
<tr>
<td>\texttt{lui}</td>
<td>001111</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
</tbody>
</table>

\textbf{TABLE 7.3} Main decoder truth table enhanced to support \texttt{lui}

\textbf{FIGURE 7.3} Modified single-cycle datapath to support \texttt{lui}
7.3 (c) \texttt{slti}

The datapath doesn’t change. Only the controller changes, as shown in Table 7.4 and Table 7.5.

<table>
<thead>
<tr>
<th>ALUOp</th>
<th>Func</th>
<th>ALUControl</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>X</td>
<td>010 (add)</td>
</tr>
<tr>
<td>01</td>
<td>X</td>
<td>110 (subtract)</td>
</tr>
<tr>
<td>10</td>
<td>100000 (add)</td>
<td>010 (add)</td>
</tr>
<tr>
<td>10</td>
<td>100010 (sub)</td>
<td>110 (subtract)</td>
</tr>
<tr>
<td>10</td>
<td>100100 (and)</td>
<td>000 (and)</td>
</tr>
<tr>
<td>10</td>
<td>100101 (or)</td>
<td>001 (or)</td>
</tr>
<tr>
<td>10</td>
<td>101010 (slt)</td>
<td>111 (set less than)</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
<td>111 (set less than)</td>
</tr>
</tbody>
</table>

\textbf{TABLE 7.4} ALU decoder truth table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>opcode</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>\texttt{lw}</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>\texttt{sw}</td>
<td>101011</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>00</td>
</tr>
<tr>
<td>\texttt{beq}</td>
<td>000100</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>01</td>
</tr>
<tr>
<td>\texttt{slti}</td>
<td>001010</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
</tr>
</tbody>
</table>

\textbf{TABLE 7.5} Main decoder truth table enhanced to support \texttt{slti}
7.3 (d) \text{blez}

First, we modify the ALU

Then, we modify the datapath
Exercise 7.5

It is not possible to implement this instruction without either modifying the register file (adding another write port) or making the instruction take two cycles to execute.

We modify the register file and datapath as shown in Figure 7.4.

### Table 7.6 Main decoder truth table enhanced to support blez

<table>
<thead>
<tr>
<th>Instruction</th>
<th>opcode</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp</th>
<th>BLEZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>blez</td>
<td>000110</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>01</td>
<td>1</td>
</tr>
</tbody>
</table>
FIGURE 7.4 Modified datapath

<table>
<thead>
<tr>
<th>Instruction</th>
<th>opcode</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp</th>
<th>Lwinc</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>lwinc</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

TABLE 7.7 Main decoder truth table enhanced to support lwinc
Exercise 7.7

Before the enhancement (see Equation 7.3, page 380 in the text, also Errata):

\[ T_c = t_{pcq_{PC}} + 2t_{mem} + t_{RF\text{read}} + t_{\text{mux}} + t_{ALU} + t_{RF\text{setup}} \]

\[ = 30 + 2(250) + 150 + 25 + 200 + 20 = 925\text{ps} \]

The unit that your friend could speed up that would make the largest reduction in cycle time would be the memory unit. So \( t_{mem\_new} = 125\text{ps} \), and the new cycle time is:

\[ T_c = 675\text{ps} \]

Exercise 7.9

(a) \text{lwi}
(b) \text{beq}
(c) \text{beq, j}

Exercise 7.11

\textbf{SystemVerilog}

module top(input logic clk, reset, 
            output logic [31:0] writedata, dataadr, 
            output logic memwrite);

  logic [31:0] pc, instr, readdata;

  // instantiate processor and memories
  mips mips(clk, reset, pc, instr, memwrite, dataadr, 
            writedata, readdata);
  imem imem(pc[7:2], instr);
  dmem dmem(clk, memwrite, dataadr, writedata, readdata);
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 imem(input logic [5:0] a,
output logic [31:0] rd;

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

initial
  $readmemh("memfile.dat",RAM);

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

module mipssingle(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        memtoreg;
logic [1:0]  alusrc;  // LUI
logic        regdst;
logic        regwrite, jump, pcsrrc, zero;
logic [3:0]  alucontrol;  // SLL
logic        ltez; // BLEZ

controller c(instr[31:26], instr[5:0], zero,
  memtoreg, memwrite, pcsrrc,
  alusrc, regdst, regwrite, jump,
  alucontrol,
  ltez);  // BLEZ

datapath dp(clk, reset, memtoreg, pcsrrc,
  alusrc, regdst, regwrite, jump,
  alucontrol,
  zero, pc, instr,
  aluresult, writedata, readdata,
  ltez);  // BLEZ
endmodule

module controller(input logic [5:0] op, funct,
  input logic        zero,
  output logic        memtoreg, memwrite,
  output logic        pcsrrc,
  output logic [1:0] alusrc,   // LUI
  output logic        regdst,
  output logic        regwrite,
  output logic        jump,
  output logic [3:0]  alucontrol, // SLL
  input logic        ltez);  // BLEZ

logic [1:0] aluop;
logic branch;
logic blez; // BLEZ

maindec md(op, memtoreg, memwrite, branch,
    alusrc, regdst, regwrite, jump,
    aluop, blez); // BLEZ
aludec ad(funct, aluop, alucontrol);

// BLEZ
assign pcsrc = (branch & zero) | (blez & ltez);

endmodule

module maindec(input logic [5:0] op,
    output logic memtoreg, memwrite,
    output logic branch,
    output logic [1:0] alusrc, // LUI
    output logic regdst,
    output logic regwrite,
    output logic jump,
    output logic [1:0] aluop,
    output logic blez); // BLEZ

    // increase control width for LUI, BLEZ
    logic [10:0] controls;

    assign {regwrite, regdst, alusrc, branch, memwrite,
    memtoreg, aluop, jump, blez} = controls;

    always_comb
    case(op)
        6'b000000: controls = 11'b11000001000; // Rtype
        6'b100011: controls = 11'b10010010000; // LW
        6'b101011: controls = 11'b00010100000; // SW
        6'b000100: controls = 11'b00001000100; // BEQ
        6'b000101: controls = 11'b00000000010; // J
        6'b000110: controls = 11'b00000000101; // BLEZ
        default: controls = 11'bxxxxxxxxxx; // ???
    endcase

endmodule

module aludec(input logic [5:0] funct,
    input logic [1:0] aluop,
    output logic [3:0] alucontrol);

    // increase to 4 bits for SLL

    always_comb
    case(aluop)
        2'b00: alucontrol = 4'b0010; // add
        2'b01: alucontrol = 4'b1010; // sub
2'b11: alucontrol = 4'b1011; // slt
default: case(funct)     // RTYPE
  6'b100000: alucontrol = 4'b0010; // ADD
  6'b100010: alucontrol = 4'b1010; // SUB
  6'b100100: alucontrol = 4'b0000; // AND
  6'b100101: alucontrol = 4'b0001; // OR
  6'b101010: alucontrol = 4'b1011; // SLT
  6'b000000: alucontrol = 4'b0100; // SLL
default:   alucontrol = 4'bxxxx; // ???
endcase
endcase
endmodule

module datapath(input logic   clk, reset,
               input logic   memtoreg, pcsrc,
               input logic [1:0]  alusrc,    // LUI
               input logic        regdst,
               input logic        regwrite, jump,
               input logic [3:0]  alucontrol, // SLL
               output logic        zero,
               output logic [31:0] pc,
               input  logic [31:0] instr,
               output logic [31:0] aluresult, writedata,
               input  logic [31:0] readdata,
               output logic        ltez);   // LTEZ

logic [4:0]  writereg;
logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
logic [31:0] signimm, signimmsh;
logic [31:0] upperimm; // LUI
logic [31:0] srca, srcb;
logic [31:0] result;
logic [31:0] memdata;

// next PC logic
flopr #(32) pcreg(clk, reset, pcnext, pc);
adder       pcadd1(pc, 32'b100, pcplus4);
s12         immsh(signimm, signimmsh);
adder       pcadd2(pcplus4, signimmsh, pcbranch);
mux2 #(32)  pcbrmux(pcplus4, pcbranch, pcsrc, pcnextbr);
mux2 #(32)  pcmux(pcnextbr, {pcplus4[31:28],
                      instr[25:0], 2'b00},
                        jump, pcnext);

// register file logic
regfile     rf(clk, regwrite, instr[25:21],
               instr[20:16], writereg,
               writeresult, srca, writedata);
mux2 #(5)   wrmux(instr[20:16], instr[15:11],
                   regdst, writereg);
signext  se(instr[15:0], signimm);
upimm   ui(instr[15:0], upperimm);  // LUI

// ALU logic
mux3 #(32) srcbmux(writedata, signimm,
      upperimm, alusrc,
      srcb);  // LUI
alu     alu(srca, srcb, alucontrol,
      instr[10:6],  // SLL
      aluresult, zero,
      ltez); // BLEZ
mux2 #(32) rdmux(aluresult, readdata,
      memtoreg, result);
endmodule

// upimm module needed for LUI
module upimm(input  logic [15:0] a,
      output logic [31:0] y);
assign y = {a, 16'b0};
endmodule

// mux3 needed for LUI
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 #1 y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule

module alu(input  logic [31:0] A, B,
      input  logic [3:0]  F,
      input  logic [4:0] shamt, // SLL
      output logic [31:0] Y,
      output logic Zero,
      output logic ltez); // BLEZ

logic [31:0] S, Bout;

assign S = A + Bout + F[3];  // SLL

always_comb
    case (F[2:0])
        3'b000: Y = A & Bout;
        3'b001: Y = A | Bout;
        3'b010: Y = S;
        3'b011: Y = S[31];
        3'b100: Y = (Bout << shamt); // SLL
    endcase
assign Zero = (Y == 32'b0);
assign ltez = Zero | S[31];  // BLEZ
endmodule

module regfile(input logic clk,
               input logic we3,
               input logic [4:0] ra1, ra2, wa3,
               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
// write third port on rising edge of clk
// register 0 hardwired to 0
always_ff @(posedge clk)
  if (we3) rf[wa3] <= wd3;

assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
endmodule

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

assign y = a + b;
endmodule

module sl2(input logic [31:0] a,
           output logic [31:0] y);

// shift left by 2
assign y = {a[29:0], 2'b00};
endmodule

module signext(input logic [15:0] a,
                output logic [31:0] y);

assign y = {{16{a[15]}}, 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 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

VHDL

-- mips.vhd

library IEEE; use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD_UNSIGNED.all;

entity testbench is end;

architecture test of testbench is

component top
  port(clk, reset: in STD_LOGIC;  
  writedata, dataadr: out STD_LOGIC_VECTOR(31 downto 0);  
  memwrite: out STD_LOGIC);
end component;

signal writedata, dataadr: STD_LOGIC_VECTOR(31 downto 0);
signal clk, reset,  memwrite: STD_LOGIC;

begin

-- instantiate device to be tested
dut: top port map(clk, reset, writedata, dataadr, memwrite);

-- Generate clock with 10 ns period
process begin
  clk <= '1';
  wait for 5 ns;
  clk <= '0';
  wait for 5 ns;
end process;

-- Generate reset for first two clock cycles
process begin
  reset <= '1';
  wait for 22 ns;
  reset <= '0';
  wait;
end process;

-- check that 7 gets written to address 84 at end of program
process (clk) begin
    if (clk'event and clk = '0' and memwrite = '1') then
        if (to_integer(dataadr) = 84 and to_integer(writedata) = 7) then
            report "NO ERRORS: Simulation succeeded" severity failure;
        elsif (dataadr /= 80) then
            report "Simulation failed" severity failure;
        end if;
    end if;
end process;
end;

library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD_UNSIGNED.all;

entity top is -- top-level design for testing
    port(clk, reset: in STD_LOGIC;
         writedata, dataadr: buffer STD_LOGIC_VECTOR(31 downto 0);
         memwrite: buffer STD_LOGIC);
end;

architecture test of top is
    component mips
        port(clk, reset: in  STD_LOGIC;
             pc:      out STD_LOGIC_VECTOR(31 downto 0);
             instr:   in  STD_LOGIC_VECTOR(31 downto 0);
             memwrite: out STD_LOGIC;
             aluresult: out STD_LOGIC_VECTOR(31 downto 0);
             writedata:    out STD_LOGIC_VECTOR(31 downto 0);
             readdata:   in  STD_LOGIC_VECTOR(31 downto 0));
    end component;
    component imem
        port(a: in STD_LOGIC_VECTOR(5 downto 0);
             rd: out STD_LOGIC_VECTOR(31 downto 0));
    end component;
    component dmem
        port(clk, we: in STD_LOGIC;
             a, wd: in STD_LOGIC_VECTOR(31 downto 0);
             rd: out STD_LOGIC_VECTOR(31 downto 0));
    end component;
signal pc, instr, readdata: STD_LOGIC_VECTOR(31 downto 0);
begi
    -- instantiate processor and memories
    mips1: mips port map(clk, reset, pc, instr, memwrite, dataadr, writedata, readdata);
    imem1: imem port map(pc(7 downto 2), instr);
    dmem1: dmem port map(clk, memwrite, dataadr, writedata, re-
addata);
end;
library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity dmem is -- data memory
    port(clk, we:  in STD_LOGIC;
        a, wd:    in STD_LOGIC_VECTOR(31 downto 0);
        rd: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of dmem is
begin
    process is
        type ramtype is array (63 downto 0) of STD_LOGIC_VECTOR(31 downto 0);
        variable mem: ramtype;
        begin
            -- read or write memory
            loop
                if clk'event and clk = '1' then
                    if (we = '1') then
                        mem(to_integer(a(7 downto 2))) := wd;
                    end if;
                end if;
                rd <= mem(to_integer(a(7 downto 2)));
                wait on clk, a;
            end loop;
    end process;
end;

library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity imem is -- instruction memory
    port(a:  in  STD_LOGIC_VECTOR(5 downto 0);
        rd: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of imem is
begin
    process is
        type ramtype is array (63 downto 0) of STD_LOGIC_VECTOR(31 downto 0);
        variable mem: ramtype;
        begin
            -- initialize memory from file
            for i in 0 to 63 loop -- set all contents low
                mem(to_integer(a(7 downto 2))) := STD_LOGIC_VECTOR'('0');
            end loop;
        end process;
    end;
end;
mem(i) := (others => '0');
end loop;
index := 0;
FILE_OPEN(mem_file, "C:/docs/DDCA2e/hdl/memfile.dat", READ_MODE);
while not endfile(mem_file) loop
  readline(mem_file, L);
  result := 0;
  for i in 1 to 8 loop
    read(L, ch);
    if '0' <= ch and ch <= '9' then
      result := character'pos(ch) - character'pos('0');
    elsif 'a' <= ch and ch <= 'f' then
      result := character'pos(ch) - character'pos('a')+10;
    else report "Format error on line " & integer'image(index) severity error;
      end if;
    mem(index)(35-i*4 downto 32-i*4) := to_std_logic_vector(result,4);
  end loop;
  index := index + 1;
end loop;
-- read memory
loop
  rd <= mem(to_integer(a));
  wait on a;
end loop;
end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mips is -- single cycle MIPS processor
  port(clk, reset: in STD_LOGIC;
    pc: out STD_LOGIC_VECTOR(31 downto 0);
    instr: in STD_LOGIC_VECTOR(31 downto 0);
    memwrite: out STD_LOGIC;
    aluresult: out STD_LOGIC_VECTOR(31 downto 0);
    writedata: out STD_LOGIC_VECTOR(31 downto 0);
    readdata: in STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of mips is
  component controller
    port(op, funct: in STD_LOGIC_VECTOR(5 downto 0);
      zero: in STD_LOGIC;
      memtoreg, memwrite: out STD_LOGIC;
      pcsrc: out STD_LOGIC;
      alusrc: out STD_LOGIC_VECTOR(1 downto 0); -- LUI
      regdst, regwrite: out STD_LOGIC;
      jump: out STD_LOGIC;
  end component;
end;
alucontrol: out STD_LOGIC_VECTOR(3 downto 0); -- SLL
ltez: out STD_LOGIC; -- BLEZ
end component;
component datapath
port(clk, reset: in STD_LOGIC;
memtoreg, pcsrc: in STD_LOGIC;
alusrc, regdst: in STD_LOGIC;
regwrite, jump: in STD_LOGIC;
alucontrol: in STD_LOGIC_VECTOR(2 downto 0);
zero: out STD_LOGIC;
pc: buffer STD_LOGIC_VECTOR(31 downto 0);
instr: in STD_LOGIC_VECTOR(31 downto 0);
aluresult: buffer STD_LOGIC_VECTOR(31 downto 0);
writedata: buffer STD_LOGIC_VECTOR(31 downto 0);
readdata: in STD_LOGIC_VECTOR(31 downto 0));
end component;
signal memtoreg: STD_LOGIC;
signal alusrc: STD_LOGIC_VECTOR(1 downto 0); -- LUI
signal regdst, regwrite, jump, pcsrc: STD_LOGIC;
signal zero: STD_LOGIC;
signal alucontrol: STD_LOGIC_VECTOR(3 downto 0); -- SLL
signal ltez: STD_LOGIC; -- BLEZ
begin
cont: controller port map(instr(31 downto 26), instr(5 downto 0),
                          zero, memtoreg, memwrite, pcsrc, alusrc,
                          regdst, regwrite, jump, alucontrol, ltez); -- BLEZ
dp: datapath port map(clk, reset, memtoreg, pcsrc, alusrc,
                       regdst,
                       regwrite, jump, alucontrol, zero, pc, instr,
                       aluresult, writedata, readdata, ltez); -- BLEZ
end;
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity controller is -- single cycle control decoder
port(op, funct: in STD_LOGIC_VECTOR(5 downto 0);
zero: in STD_LOGIC;
memtoreg, memwrite: out STD_LOGIC;
pcsrc: out STD_LOGIC;
alusrc: out STD_LOGIC_VECTOR(1 downto 0); -- LUI
regdst, regwrite: out STD_LOGIC;
jump: out STD_LOGIC;
alucontrol: out STD_LOGIC_VECTOR(3 downto 0); -- SLL
ltez: out STD_LOGIC; -- BLEZ
end;
architecture struct of controller is
component maindec
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity maindec is -- main control decoder
  port(op: in STD_LOGIC_VECTOR(5 downto 0);
       memtoreg, memwrite: out STD_LOGIC;
       branch: out STD_LOGIC;
       alusrc: out STD_LOGIC_VECTOR(1 downto 0); -- LUI
       regdst, regwrite: out STD_LOGIC;
       jump: out STD_LOGIC;
       aluop: out STD_LOGIC_VECTOR(1 downto 0);
       blez: out STD_LOGIC);
end;

architecture behave of maindec is
  signal controls: STD_LOGIC_VECTOR(10 downto 0);
begin
  process(all) begin
    case op is
      when "000000" => controls <= "11000001000"; -- RTYPE
      when "100011" => controls <= "10010010000"; -- LW
      when "101011" => controls <= "00010100000"; -- SW
      when "000100" => controls <= "00001000100"; -- BEQ
      when "000010" => controls <= "00000000010"; -- J
      when "000100" => controls <= "10010000000"; -- ADDI
      when "000010" => controls <= "00000000010"; -- J
      when "000100" => controls <= "10010001100"; -- SLTI
      when "000011" => controls <= "10100000000"; -- LUI
    end case;
  end process;
end;
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity aludec is -- ALU control decoder
    port(funct: in  STD_LOGIC_VECTOR(5 downto 0);
         aluop: in  STD_LOGIC_VECTOR(1 downto 0);
         alucontrol: out STD_LOGIC_VECTOR(3 downto 0));  -- SLL
end;

architecture behave of aludec is
begin
    process(all) begin
        case aluop is
            when "00" => alucontrol <= "0010"; -- add
            when "01" => alucontrol <= "1010"; -- sub
            when "11" => alucontrol <= "1011"; -- slt
            when others => case funct is      -- R-type instructions
                when "100000" => alucontrol <= "0010";
                                -- add
                when "100010" => alucontrol <= "1010";
                                -- sub
                when "100100" => alucontrol <= "0000";
                                -- and
                when "100101" => alucontrol <= "0001";
                                -- or
                when "101010" => alucontrol <= "1011";
                                -- slt
                when "000000" => alucontrol <= "0100";
                                -- sll
                when others   => alucontrol <= "----"; -- ???
                end case;
        end case;
    end process;
end;

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

entity datapath is  -- MIPS datapath
    port(clk, reset: in  STD_LOGIC;
         memtoreg, psrc: in  STD_LOGIC;
         alusrc: in  STD_LOGIC_VECTOR(1 downto 0);  -- LUI
         alusrc, regdst: in  STD_LOGIC;
         regwrite, jump, blez: in  STD_LOGIC;
         alucontrol: in  STD_LOGIC_VECTOR(3 downto 0));  -- SLL
end;

zero: out STD_LOGIC;
pc: buffer STD_LOGIC_VECTOR(31 downto 0);
instr: in STD_LOGIC_VECTOR(31 downto 0);
alureresult: buffer STD_LOGIC_VECTOR(31 downto 0);
write-data: buffer STD_LOGIC_VECTOR(31 downto 0);
read-data: in STD_LOGIC_VECTOR(31 downto 0);
ltez: out STD_LOGIC); -- LTEZ
end;

architecture struct of datapath is
component alu
port(a, b: in STD_LOGIC_VECTOR(31 downto 0);
alucontrol: in STD_LOGIC_VECTOR(3 downto 0); --SLL
shamt: in STD_LOGIC_VECTOR(4 downto 0); --SLL
result: buffer STD_LOGIC_VECTOR(31 downto 0);
zero: buffer STD_LOGIC; --BLEZ
ltez: out STD_LOGIC); --BLEZ
end component;
component regfile
port(clk: in STD_LOGIC;
we3: in STD_LOGIC;
ra1, ra2, wa3: in STD_LOGIC_VECTOR(4 downto 0);
wd3: in STD_LOGIC_VECTOR(31 downto 0);
rd1, rd2: out STD_LOGIC_VECTOR(31 downto 0));
end component;
component adder
port(a, b: in STD_LOGIC VECTOR(31 downto 0);
y: out STD_LOGIC_VECTOR(31 downto 0));
end component;
component sl2
port(a: in STD_LOGIC_VECTOR(31 downto 0);
y: out STD_LOGIC_VECTOR(31 downto 0));
end component;
component signext
port(a: in STD_LOGIC_VECTOR(15 downto 0);
y: out STD_LOGIC_VECTOR(31 downto 0));
end component;
component upimm
port(a: in STD_LOGIC_VECTOR(15 downto 0);
y: out STD_LOGIC_VECTOR(31 downto 0));
end component;
component flopr generic(width: integer);
port(clk, reset: in STD_LOGIC;
d: in STD_LOGIC_VECTOR(width-1 downto 0);
q: out STD_LOGIC_VECTOR(width-1 downto 0));
end component;
component mux2 generic(width: integer);
port(d0, d1: in STD_LOGIC_VECTOR(width-1 downto 0);
s: in STD_LOGIC;
y: out STD_LOGIC_VECTOR(width-1 downto 0));
end component;
component mux3 generic(width: integer);
port(d0, d1, d2: in STD_LOGIC_VECTOR(width-1 downto 0);
s: in STD_LOGIC_VECTOR(1 downto 0);
y: out STD_LOGIC_VECTOR(width-1 downto 0));
end component;
signal writereg: STD_LOGIC_VECTOR(4 downto 0);
signal pcjump, pcnext, pcnextbr, pcplus4, pcbranch: STD_LOGIC_VECTOR(31 downto 0);
signal upperimm: STD_LOGIC_VECTOR(31 downto 0);  -- LUI
signal signimm, signimmsh: STD_LOGIC_VECTOR(31 downto 0);
signal srca, srcb, result: STD_LOGIC_VECTOR(31 downto 0);
begin
  -- next PC logic
  pcjump <= pcplus4(31 downto 28) & instr(25 downto 0) & "00";
  pcreg: flopr generic map(32) port map(clk, reset, pcnext, pc);
  pcadd1: adder port map(pc, X"00000004", pcplus4);
  immsh: sl2 port map(signimm, signimmsh);
  pcadd2: adder port map(pcplus4, signimmsh, pcbranch);
  pcbrmux: mux2 generic map(32) port map(pcplus4, pcbranch, pcsrc, pcnextbr);
  pcmux: mux2 generic map(32) port map(pcnextbr, pcjump, jump, pcnext);

  -- register file logic
  rf: regfile port map(clk, regwrite, instr(25 downto 21),
  instr(20 downto 16), writereg, result, srca,
  writedata);
  wrmux: mux2 generic map(5) port map(instr(20 downto 16),
  instr(15 downto 11), regdst, writereg);
  resmux: mux2 generic map(32) port map(aluresult, readdata,
  memtoreg, result);
  se: signext port map(instr(15 downto 0), signimm);
  ui: upimm port map(instr(15 downto 0), upperimm);  -- LUI

  -- ALU logic
  srcbmux: mux3 generic map(32) port map(writedata, signimm,
  upperimm, alusrc, srcb);  -- LUI
  mainalu: alu port map(srca, srcb, alucontrol, instr(10 downto 6), -- SLL
  aluresult, zero, ltez);  -- BLEZ
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity regfile is -- three-port register file
port(clk: in STD_LOGIC;
we3: in STD_LOGIC;
ral, ra2, wa3: in STD_LOGIC_VECTOR(4 downto 0);
wd3: in STD_LOGIC_VECTOR(31 downto 0);
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity adder is -- adder
    port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
          y: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of adder is
begin
    y <= a + b;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sl2 is -- shift left by 2
    port(a: in  STD_LOGIC_VECTOR(31 downto 0);
          y: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of sl2 is
begin
    y <= a(29 downto 0) & "00";
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity signext is -- sign extender
    port(a: in  STD_LOGIC_VECTOR(15 downto 0);
            y: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of signext is
begin
    y <= X"ffff" & a when a(15) else X"0000" & a;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity upimm is -- create upper immediate for LUI
    port(a: in  STD_LOGIC_VECTOR(15 downto 0);
            y: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of upimm is
begin
    y <= a & X"0000";
end;

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

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

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux2 is -- two-input multiplexer
    generic(width: integer);
    port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
            s: in  STD_LOGIC;
            y: out STD_LOGIC_VECTOR(width-1 downto 0));
end;
architecture behave of mux2 is
begin
  y <= d1 when s else d0;
end;

-- 3:1 mux needed for LUI
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux3 is -- three-input multiplexer
  generic(width: integer);
  port(d0, d1, d2: in  STD_LOGIC_VECTOR(width-1 downto 0);
       s: in  STD_LOGIC_VECTOR(1 downto 0);
       y: out STD_LOGIC_VECTOR(width-1 downto 0));
end;
architecture behave of mux3 is
begin
  y <= d1 when s(1) else (d1 when s(0) else d0);
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity alu is
  port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
        alucontrol: in  STD_LOGIC_VECTOR(3 downto 0);   --SLL
        shamt: in  STD_LOGIC_VECTOR(4 downto 0);   --SLL
        result:     buffer STD_LOGIC_VECTOR(31 downto 0);
        zero:       buffer STD_LOGIC; --BLEZ
        ltez:   out STD_LOGIC); --BLEZ
end;
architecture behave of alu is
signal condinvb, sum: STD_LOGIC_VECTOR(31 downto 0);
begin
  condinvb <= not b when alucontrol(3) else b;
  sum <= a + condinvb + alucontrol(3);
  process(all) begin
    case alucontrol(2 downto 0) is
      when "000"   => result <= a and b;
      when "001"   => result <= a or b;
      when "010"   => result <= sum;
      when "011"   => result <= (0 => sum(31), others => '0');
      when "100"   => result <= (condinvb << shamt);   --SLL
      when others  => result <= (others => 'X');
    end case;
  end process;
  zero <= '1' when result = X"00000000" else '0';
  ltez <= zero or sum(31);
end;
Exercise 7.13

(a) $srlv$

First, we show the modifications to the ALU.

![Modified ALU to support $srlv$](image-url)
Next, we show the modifications to the ALU decoder.

![ALU Control truth table]

**TABLE 7.8 ALU decoder truth table**

<table>
<thead>
<tr>
<th>ALUOp</th>
<th>Function</th>
<th>ALUControl</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>X</td>
<td>0010 (add)</td>
</tr>
<tr>
<td>X1</td>
<td>X</td>
<td>1010 (subtract)</td>
</tr>
<tr>
<td>1X</td>
<td>100000 (add)</td>
<td>0010 (add)</td>
</tr>
<tr>
<td>1X</td>
<td>100010 (sub)</td>
<td>1010 (subtract)</td>
</tr>
<tr>
<td>1X</td>
<td>100100 (and)</td>
<td>0000 (and)</td>
</tr>
<tr>
<td>1X</td>
<td>100101 (or)</td>
<td>0001 (or)</td>
</tr>
<tr>
<td>1X</td>
<td>101010 (slt)</td>
<td>1011 (set less than)</td>
</tr>
<tr>
<td>1X</td>
<td>000110 (<strong>SRLV</strong>)</td>
<td>0100 (shift right logical variable)</td>
</tr>
</tbody>
</table>
Next, we show the changes to the datapath. The only modification is the width of \textit{ALUControl}. No changes are made to the datapath main control FSM.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure7_7.png}
\caption{Modified multicycle MIPS datapath to support \texttt{sll}}
\end{figure}

(b) \texttt{ori}
We add a zero extension unit to the datapath, extend the $ALUSrcB$ signal from 2 bits to 3 bits, and extend the $SrcB$ multiplexer from 4 inputs to 5 inputs. We also modify the ALU decoder and main control FSM.

FIGURE 7.8 Modified datapath for $ori$
SOLUTIONS 263

Exercise Solutions

Exercise Solutions

TABLE 7.9 ALU decoder truth table

<table>
<thead>
<tr>
<th>ALUOp</th>
<th>Func</th>
<th>ALUControl</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100101 (or)</td>
<td>001 (or)</td>
</tr>
<tr>
<td>10</td>
<td>101010 (slt)</td>
<td>111 (set less than)</td>
</tr>
</tbody>
</table>

TABLE 7.9 ALU decoder truth table
(c) \texttt{xori}

First, we modify the ALU and the ALU decoder.

\begin{table}
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{ALUControl}_{3:0} & \textbf{Function} \\
\hline
0000 & A AND B \\
0001 & A OR B \\
0010 & A + B \\
0011 & not used \\
1000 & A AND \overline{B} \\
1001 & A OR \overline{B} \\
1010 & A - B \\
1011 & SLT \\
0100 & A XOR B \\
\hline
\end{tabular}
\end{table}
Next, we modify the datapath. We change the buswidth of the $ALUControl$ signal from 3 bits to 4 bits and the $ALUSrcB$ signal from 2 bits to 3 bits. We also extend the SrcB mux and add a zero-extension unit.

<table>
<thead>
<tr>
<th>$ALUOp$</th>
<th>$Funct$</th>
<th>$ALUControl$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>X</td>
<td>0010 (add)</td>
</tr>
<tr>
<td>01</td>
<td>X</td>
<td>1010 (subtract)</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
<td>0100 (xor)</td>
</tr>
<tr>
<td>10</td>
<td>100000 (add)</td>
<td>0010 (add)</td>
</tr>
<tr>
<td>10</td>
<td>100010 (sub)</td>
<td>1010 (subtract)</td>
</tr>
<tr>
<td>10</td>
<td>100100 (and)</td>
<td>0000 (and)</td>
</tr>
<tr>
<td>10</td>
<td>100101 (or)</td>
<td>0001 (or)</td>
</tr>
<tr>
<td>10</td>
<td>101010 (slt)</td>
<td>1011 (set less than)</td>
</tr>
</tbody>
</table>

**TABLE 7.10** ALU decoder truth table for xor
And finally, we modify the main control FSM.
(d) \texttt{jr}

First, we extend the ALU Decoder for \texttt{jr}.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\textbf{ALUOp} & \textbf{Funct} & \textbf{ALUControl} \\
\hline
00 & X & 010 (add) \\
\hline
X1 & X & 110 (subtract) \\
\hline
1X & 100000 (add) & 010 (add) \\
\hline
1X & 100010 (sub) & 110 (subtract) \\
\hline
1X & 100100 (and) & 000 (and) \\
\hline
1X & 100101 (or) & 001 (or) \\
\hline
\end{tabular}
\caption{ALU decoder truth table with \texttt{jr}}
\end{table}
Next, we modify the main controller. The datapath requires no modification.

Exercise 7.15

Yes, it is possible to add this instruction without modifying the register file. First we show the modifications to the datapath. The only modification is adding the \( rs \) field of the instruction (Instruction\text{25:21}) to the input of the write address mux of the register file. \( RegDst \) must be expanded to two bits.
The finite state machine requires another state to write the rs register. If execution time is critical, another adder could be placed just after the A/B register to add 4 to A. Then in State 3, as memory is read, the register file could be written back with the incremented rs. In that case, lwinc would require the same number of cycles as lw. The penalty, however, would be chip area, and thus power and cost.
Exercise 7.17
We add an enable signal, \( FlPtEn \), to the result register.
Exercise 7.19

Because the ALU is not on the critical path, the speedup in performance of the ALU does not affect the cycle time. Thus, the cycle time, given in Example 7.8, is still 325 ps. Given the instruction mix in Example 7.7, the overall execution time for 100 billion instructions is still 133.9 seconds.
Exercise 7.21
**Exercise 7.23**

\[ 4 + (3 + 4 + 3) \times 5 + 3 = 57 \text{ clock cycles} \]

The number of instructions executed is \(1 + (3 \times 5) + 1 = 17\). Thus, the CPI = \(\frac{57\text{ clock cycles}}{17\text{ instructions}} = 3.35\text{ CPI}\).
**Exercise 7.25**

**MIPS Multicycle Processor**

**SystemVerilog**

```verilog
module mips(input  logic clk, reset, 
            output logic [31:0] adr, writedata, 
            output logic memwrite, 
            input  logic [31:0] readdata);

logic        zero, pcen, irwrite, regwrite, 
alusrc, iord, memtoreg, regdst; 
logic [1:0]  alusrcb, pcsrc; 
logic [2:0]  alucontrol; 
logic [5:0]  op, funct;

controller c(clk, reset, op, funct, zero, 
             pcen, memwrite, irwrite, regwrite, 
alusrc, iord, memtoreg, regdst, 
alusrcb, pcsrc, alucontrol);

datapath dp(clk, reset, 
            pcen, irwrite, regwrite, 
alusrc, iord, memtoreg, regdst, 
alusrcb, pcsrc, alucontrol, 
op, funct, zero, 
            adr, writedata, readdata);

endmodule
```

**VHDL**

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

entity mips is -- multicycle MIPS processor 
port(clk, reset: in  STD_LOGIC; 
    adr: out STD_LOGIC_VECTOR(31 downto 0); 
    writedata: inout STD_LOGIC_VECTOR(31 downto 0); 
    memwrite: out STD_LOGIC; 
    readdata: in  STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of mips is 
component controller 
port(clk, reset: in  STD_LOGIC; 
    op, funct: in  STD_LOGIC_VECTOR(5 downto 0); 
    zero: in  STD_LOGIC; 
    pcen, memwrite: out STD_LOGIC; 
    irwrite, regwrite: out STD_LOGIC; 
    alusrc, iord: out STD_LOGIC; 
    memtoreg, regdst: out STD_LOGIC; 
    alusrcb, pcsrc: out STD_LOGIC_VECTOR(1 downto 0); 
    alucontrol: out STD_LOGIC_VECTOR(2 downto 0));
end component;

component datapath 
port(clk, reset: in  STD_LOGIC; 
    pcen, irwrite: in  STD_LOGIC; 
    regwrite, alusrc: in  STD_LOGIC; 
    iord, memtoreg: in  STD_LOGIC; 
    regdst: in  STD_LOGIC; 
    alusrcb, pcsrc: in STD_LOGIC_VECTOR(1 downto 0); 
    alucontrol: in STD_LOGIC_VECTOR(2 downto 0); 
    readdata: in  STD_LOGIC_VECTOR(31 downto 0); 
    op, funct: out STD_LOGIC_VECTOR(5 downto 0); 
    zero: out STD_LOGIC; 
    adr: out  STD_LOGIC_VECTOR(31 downto 0); 
    writedata: inout STD_LOGIC_VECTOR(31 downto 0));
end component;

begin 
c: controller port map(clk, reset, op, funct, zero, 
    pcen, memwrite, irwrite, regwrite, 
alusrc, iord, memtoreg, regdst, 
alusrcb, pcsrc, alucontrol);
dp: datapath port map(clk, reset, 
    pcen, irwrite, regwrite, 
alusrc, iord, memtoreg, regdst, 
alusrcb, pcsrc, alucontrol, 
    readdata, op, funct, zero, 
    adr, writedata);
end;
```
MIPS Multicycle Control

**SystemVerilog**

```verilog
module controller(input logic clk, reset, 
  input logic [5:0] op, funct, 
  input logic zero, 
  output logic pcen, memwrite, 
  output logic [1:0] alusrcb, psrc, 
  output logic [2:0] alucontrol);

logic [1:0] aluop;
logic branch, pcwrite;

// Main Decoder and ALU Decoder subunits.
maindec md(clk, reset, op, 
  pcwrite, memwrite, irwrite, regwrite, 
  alusrcb, branch, iord, memtoreg, regdst, 
  alusrcb, psrc, aluop);

aludec ad(funct, aluop, alucontrol);

assign pcen = pcwrite or (branch & zero);

endmodule
```

**VHDL**

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

entity controller is -- multicycle control decoder
port(clk, reset: in  STD_LOGIC;
  op, funct: in  STD_LOGIC_VECTOR(5 downto 0);
  zero:     in  STD_LOGIC;
  pcen, memwrite:     out STD_LOGIC;
  irwrite, regwrite:  out STD_LOGIC;
  alusrca, iord:     out STD_LOGIC;
  memtoreg, regdst:   out STD_LOGIC;
  alusrcb, psrc:     out STD_LOGIC_VECTOR(1 downto 0);
  alucontrol: out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture struct of controller is
  component maindec
    port(clk, reset:         in  STD_LOGIC;
      op:                 in  STD_LOGIC_VECTOR(5 downto 0);
      pcwrite, memwrite:  out STD_LOGIC;
      irwrite, regwrite:  out STD_LOGIC;
      alusrca, branch:    out STD_LOGIC;
      iord, memtoreg:     out STD_LOGIC;
      regdst:   out STD_LOGIC;
      alusrcb, psrc:     out STD_LOGIC_VECTOR(1 downto 0);
      alucontrol: out STD_LOGIC_VECTOR(2 downto 0));
  end component;

  component aludec
    port(funct:      in  STD_LOGIC_VECTOR(5 downto 0);
      aluop:      in  STD_LOGIC_VECTOR(1 downto 0);
      alucontrol: out STD_LOGIC_VECTOR(2 downto 0));
  end component;

  signal aluop: STD_LOGIC_VECTOR(1 downto 0);
  signal branch, pcwrite: STD_LOGIC;
begin
  md: maindec port map(clk, reset, op, 
    pcwrite, memwrite, irwrite, regwrite, 
    alusrca, branch, iord, memtoreg, regdst, 
    alusrcb, psrc, aluop);
  ad: aludec port map(funct, aluop, alucontrol);
  pcen <= pcwrite or (branch and zero);
end;
```
MIPS Multicycle Main Decoder FSM

SystemVerilog

module maindec(input logic clk, reset,
               input logic [5:0] op,
               output logic pcwrite, memwrite,
               irwrite, regwrite,
               output logic alusrca, branch, iord,
               memtoreg, regdst,
               output logic [1:0] alusrcb, pcsrc,
               output logic [1:0] aluop);

typedef enum logic [3:0] {FETCH, DECODE, MEMADR,
                         MEMRD, MEMWR, RTYPEEX,
                         RTYPEWB, BEQEX, ADDIEX,
                         ADDIWB, JEX} statetype;

statetype [3:0] state, nextstate;

parameter LW = 6'b100011; // Opcode for lw
parameter SW = 6'b101011; // Opcode for sw
parameter RTYPE = 6'b000000; // Opcode for R-type
parameter BEQ = 6'b000100; // Opcode for beq
parameter ADDI = 6'b001000; // Opcode for addi
parameter J = 6'b000010;  // Opcode for j

reg [3:0] state, nextstate;
reg [14:0] controls;

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

VHDL

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity maindec is -- main control decoder
port(clk, reset: in STD_LOGIC;
     op: in STD_LOGIC_VECTOR(5 downto 0);
     pcwrite, memwrite: out STD_LOGIC;
     irwrite, regwrite: out STD_LOGIC;
     alusrca, branch: out STD_LOGIC;
     iord, memtoreg: out STD_LOGIC;
     alusrcb, pcsrc: out STD_LOGIC_VECTOR(1 downto 0);
     aluop: out STD_LOGIC_VECTOR(1 downto 0));
end;
architecture behave of maindec is

-- next state logic
process(clk, reset) begin
if reset then state <= FETCH;
elif rising_edge(clk) then
  state <= nextstate;
end if;
end process;

architecture behav of maindec is

type statetype is (FETCH, DECODE, MEMADR, MEMRD, MEMWR, RTYPEEX,
                   RTYPEWB, BEQEX, ADDIEX, ADDIWB, JEX);

signal state, nextstate: statetype;
signal controls: STD_LOGIC_VECTOR(14 downto 0);

begin
  -- state register
  process(clk, reset) begin
    if reset then state <= FETCH;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

  -- next state logic
  process(all) begin
    case state is
      when FETCH  =>          nextstate <= DECODE;
      when DECODE =>
        case op is
          when "100011" => nextstate <= MEMADR;
          when "101011" => nextstate <= MEMWR;
          when "000000" => nextstate <= RTYPEEX;
          when "000100" => nextstate <= BEQEX;
          when "001000" => nextstate <= ADDIEX;
          when "000010" => nextstate <= JEX;
          when others => nextstate <= FETCH; -- should never happen
        end case;
      when MEMADR =>
        case op is
          when "100011" => nextstate <= MEMRD;
          when "101011" => nextstate <= MEMWR;
          when others => nextstate <= FETCH; -- should never happen
        end case;
      when MEMRD =>
        case op is
          when "100011" => nextstate <= MEMWR;
          when "101011" => nextstate <= MEMRD;
          when others => nextstate <= FETCH; -- should never happen
        end case;
      when MEMWR =>
        case op is
          when "100011" => nextstate <= MEMWR;
          when "101011" => nextstate <= MEMRD;
          when others => nextstate <= FETCH; -- should never happen
        end case;
      when RTYPEEX =>
        case op is
          when "100011" => nextstate <= RTYPEWB;
          when "101011" => nextstate <= RTYPEEX;
          when others => nextstate <= FETCH; -- should never happen
        end case;
      when BEQEX =>
        case op is
          when "100011" => nextstate <= BEQEX;
          when "101011" => nextstate <= BEQEX;
          when others => nextstate <= FETCH; -- should never happen
        end case;
      when ADDIEX =>
        case op is
          when "100011" => nextstate <= ADDIEX;
          when "101011" => nextstate <= ADDIEX;
          when others => nextstate <= FETCH; -- should never happen
        end case;
      when JEX =>
        case op is
          when "100011" => nextstate <= JEX;
          when "101011" => nextstate <= JEX;
          when others => nextstate <= FETCH; -- should never happen
        end case;
    end case;
  end process;
SystemVerilog

// next state logic
always_comb
case(state)
    FETCH:   nextstate <= DECODE;
    DECODE:  case(op)
        LW:     nextstate <= MEMADR;
        SW:     nextstate <= MEMADR;
        RTYPE:  nextstate <= RTYPEEX;
        BEQ:    nextstate <= BEQEX;
        ADDI:   nextstate <= ADDIEX;
        J:      nextstate <= JEX;
        default: nextstate <= FETCH;
    // default should never happen
endcase
MEMADR:  case(op)
    LW:     nextstate <= MEMRD;
    SW:     nextstate <= MEMWR;
    default: nextstate <= FETCH;
    // default should never happen
endcase
MEMRD:   nextstate <= MEMWB;
MEMWB:   nextstate <= FETCH;
MEMWR:   nextstate <= FETCH;
RTYPEEX: nextstate <= RTYPEWB;
RTYPEWB: nextstate <= FETCH;
BEQEX:   nextstate <= BEQEX;
ADDIEX:  nextstate <= ADDIWB;
ADDIWB:  nextstate <= FETCH;
JEX:     nextstate <= FETCH;
default: nextstate <= FETCH;
    // default should never happen
endcase

// output logic
assign {pcwrite, memwrite, irwrite, regwrite, alusrca, branch, iord, mentoreg, regdst, alusrcb, pcsrc, aluop} = controls;

always_comb
case (state)
    FETCH:   controls <= 15'b1010_00000_0100_00;
    DECODE:  controls <= 15'b0000_00000_1100_00;
    MEMADR:  controls <= 15'b0000_10000_1000_00;
    MEMRD:   controls <= 15'b0000_00100_0000_00;
    MEMWB:   controls <= 15'b0000_00010_0000_00;
    MEMWR:   controls <= 15'b0100_00100_0000_00;
    RTYPEEX: controls <= 15'b0000_10000_1000_10;
    RTYPEWB: controls <= 15'b0000_11000_0001_01;
    BEQEX:   controls <= 15'b0000_10000_1100_00;
    ADDIEX:  controls <= 15'b0000_11000_1000_00;
    ADDIWB:  controls <= 15'b0000_00000_0100_00;
    JEX:     controls <= 15'b0100_00000_0100_00;
    default: controls <= 15'b0000_xxxxx_xxxx_xx;
endcase
endmodule

VHDL

-- output logic
process(all) begin
    case state is
        when FETCH => controls <= "101000000010000";
        when DECODE => controls <= "000000000110000";
        when MEMADR => controls <= "000010000100000";
        when MEMRD => controls <= "000001000000000";
        when MEMWR => controls <= "000100000000000";
        when RTYPEEX => controls <= "000100000010010";
        when RTYPEWB => controls <= "000100000100000";
        when BEQEX => controls <= "000100000000010";
        when ADDIEX => controls <= "000100000100000";
        when ADDIWB => controls <= "000100000000000";
        when JEX => controls <= "100000000001000";
        when others => controls <= "---------------"; -- illegal op
    end case;
end process;

pcwrite  <= controls(14);
memwrite <= controls(13);
irwrite  <= controls(12);
regwrite <= controls(11);
alusrc  <= controls(10);
brANCH  <= controls(9);
iord    <= controls(8);
mentoreg <= controls(7);
regdst  <= controls(6);
alusrcb <= controls(5 downto 4);
pcsrrc  <= controls(3 downto 2);
aluop   <= controls(1 downto 0);
end;
MIPS Multicycle ALU Decoder

SystemVerilog

module aludec(input logic [5:0] funct, input logic [1:0] aluop, output logic [2:0] alucontrol);

always_comb case(aluop)
  2'b00: alucontrol <= 3'b010; // add
  2'b01: alucontrol <= 3'b110; // sub
  default: case(funct) // RTYPE
    6'b100000: alucontrol <= 3'b010; // ADD
    6'b100010: alucontrol <= 3'b110; // SUB
    6'b100100: alucontrol <= 3'b000; // AND
    6'b100101: alucontrol <= 3'b001; // OR
    6'b101010: alucontrol <= 3'b111; // SLT
    default: alucontrol <= 3'bxxx; // ???
endcase
endmodule

VHDL

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity aludec is -- ALU control decoder
  port(funct: in STD_LOGIC_VECTOR(5 downto 0);
       aluop: in STD_LOGIC_VECTOR(1 downto 0);
       alucontrol: out STD_LOGIC_VECTOR(2 downto 0));
end;
architecture behave of aludec is
begin
  process(all) begin
    case aluop is
      when "00" => alucontrol <= "010"; -- add (for lb/sb/addi)
      when "01" => alucontrol <= "110"; -- sub (for beq)
      when "11" => alucontrol <= "111"; -- slt (for slti)
      when others => case funct is -- R-type instructions
        when "100000" => alucontrol <= "010"; -- add
        when "100010" => alucontrol <= "110"; -- sub
        when "100100" => alucontrol <= "000"; -- and
        when "100101" => alucontrol <= "001"; -- or
        when "101010" => alucontrol <= "111"; -- slt
        when others => alucontrol <= "---"; // ???
      end case;
    end case;
  end process;
end;
SystemVerilog

module datapath(input logic    clk, reset, pcen, irwrite, regwrite, alusrca, iord, memtoreg, regdst, alusrcb, pcsrc, input logic [1:0] alusrcb, pcsrc, input logic [2:0] alucontrol, output logic [5:0] op, funct, output logic zero, output logic [31:0] adr, writedata, input logic [31:0] readdata);

// Internal signals of the datapath module.
logic [4:0] writereg;
logic [31:0] pcnext, pc;
logic [31:0] instr, data, srca, srcb;
logic [31:0] a;
logic [31:0] aluresult, aluout;
logic [31:0] signimm; // sign-extended immediate
logic [31:0] signimmsh; // sign-extended immediate
          // shifted left by 2
logic [31:0] wd3, rd1, rd2;
// op and funct fields to controller
assign op = instr[31:26];
assign funct = instr[5:0];

// datapath
flopenr #(32) pcreg(clk, reset, pcen, pcnext, pc);
mux2 # (32) adrmux(pc, aluout, iord, adr);

flopenr #(32) instrreg(clk, reset, irwrite, readdata, instr);

flopr # (5) regdstmux(instr[20:16], instr[15:11], regdst, writereg);
mux2 # (32) wdmux(aluout, data, memtoreg, wd3);
regfile rf(clk, regwrite, instr[25:21], instr[20:16], writereg, wd3, rd1, rd2);

signext se(instr[15:0], signimm);

s[2] imsh(signimm, signimmsh);
flopr # (32) areg(clk, reset, rd1, a);
flopr # (32) breg(clk, reset, rd2, writedata);
mux2 # (32) srcamux(pc, a, alusrc, srca);
mux4 # (32) srcbmux(writedata, 32'b100, signimm, signimmsh, alusrcb, srcb);
alu alu(srca, srcb, alucontrol, aluresult, zero);
flopr # (32) alureg(clk, reset, aluresult, aluout);
mux3 # (32) pcmux(aluresult, aluout, {pc[31:28], instr[25:0], 2'b00}, pcsrc, pcnext);

endmodule

VHDL

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

entity datapath is  -- MIPS datapath
port(clk, reset: in  STD_LOGIC;
     pcen, irwrite:     in  STD_LOGIC;
     regwrite, alusrca: in  STD_LOGIC;
iord, memtoreg:    in  STD_LOGIC;
     regdst:   in  STD_LOGIC;
     alusrcb, pcsrc:    in  STD_LOGIC_VECTOR(1 downto 0);
alucontrol: in  STD_LOGIC_VECTOR(2 downto 0);
readdata:    in  STD_LOGIC_VECTOR(31 downto 0);
op, funct:   out  STD_LOGIC_VECTOR(5 downto 0);
zero:       out  STD_LOGIC;
adr:        out  STD_LOGIC_VECTOR(31 downto 0);
writedata:  inout STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of datapath is

component alu
port(A, B: in  STD_LOGIC_VECTOR(31 downto 0);
     F:    in  STD_LOGIC_VECTOR(2 downto 0);
     Y:    buffer STD_LOGIC_VECTOR(31 downto 0);
     Zero: out  STD_LOGIC);
end component;

component regfile
port(clk:    in  STD_LOGIC;
     we3:    in  STD_LOGIC;
     ra1, ra2, wa3: in  STD_LOGIC_VECTOR(4 downto 0);
     wd3:    in  STD_LOGIC_VECTOR(31 downto 0);
     rd1, rd2: out  STD_LOGIC_VECTOR(31 downto 0));
end component;

component adder
port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
     y:    out  STD_LOGIC_VECTOR(31 downto 0));
end component;

component sl2
port(a: in  STD_LOGIC_VECTOR(31 downto 0);
     y: out  STD_LOGIC_VECTOR(31 downto 0));
end component;

component signext
port(a: in  STD_LOGIC_VECTOR(15 downto 0);
     y:    out  STD_LOGIC_VECTOR(31 downto 0));
end component;

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

component flopenr generic(width: integer);
port(clk, reset: in  STD_LOGIC;
     en:         in  STD_LOGIC;
     d:          in  STD_LOGIC_VECTOR(width-1 downto 0);
     q:          out  STD_LOGIC_VECTOR(width-1 downto 0));
end component;

component mux2 generic(width: integer);
port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
     s:      in  STD_LOGIC_VECTOR(1 downto 0);
     y:      out  STD_LOGIC_VECTOR(width-1 downto 0));
end component;

component mux3 generic(width: integer);
port(d0, d1, d2: in  STD_LOGIC_VECTOR(width-1 downto 0);
     s:                in  STD_LOGIC_VECTOR(1 downto 0);
     y:                out  STD_LOGIC_VECTOR(width-1 downto 0));
end component;

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

signal writereg: STD_LOGIC_VECTOR(31 downto 0);
signal pcnext, pc, instr, data, srca, srcb, a,
aluresult, aluout, signimm, signimmsh, wd3, rd1, rd2, pcjump:
STD_LOGIC_VECTOR(31 downto 0));

end
(continued from previous page)

**VHDL**

```vhdl
begin
  -- op and funct fields to controller
  op <= instr(31 downto 26);
  funct <= instr(5 downto 0);
  -- datapath
  pcreg: flopenr generic map(32) port map(clk, reset, pcen, pcnext, pc);
  adrmux: mux2 generic map(32) port map(pc, aluout, iord, adr);
  instrreg: flopenr generic map(32) port map(clk, reset, lrwrite, readdata, instr);
  datareg: flopr generic map(32) port map(clk, reset, readdata, data);
  regdstmux: mux2 generic map(5) port map(instr(20 downto 16), instr(15 downto 11), regdst, writereg);
  wdmux: mux2 generic map(32) port map(aluout, data, memtoreg, wd);
  rf: regfile port map(clk, regwrite, instr(20 downto 21), instr(20 downto 16), writereg, wd3, rd1, rd2);
  se: signext port map(instr(15 downto 0), signimm);
  lmmsh: s12 port map(signimm, signimmsh);
  areg: flopr generic map(32) port map(clk, reset, rd1, a);
  breg: flopr generic map(32) port map(clk, reset, rd2, writedata);
  srcamux: mux4 generic map(32) port map(pc, a, alusrcr, srca);
  srcbmunx: mux4 generic map(32) port map(writedata, "0"*0, signimm, signimmsh, alusrcb, srcb);
  alu32: alu port map(srca, srcb, alucontrol, aluresult, zero);
  alureg: flopr generic map(32) port map(clk, reset, aluresult, aluout);
  pcjump <= pc(31 downto 28)&instr(25 downto 0)&"00";
  pcmux: mux3 generic map(32) port map(aluresult, aluout, pcjump, pcsr, pcnext);
end;
```
The following HDL describes the building blocks that are used in the MIPS multicycle processor that are not found in Section 7.6.2.

**MIPS Multicycle Building Blocks**

**SystemVerilog**

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

    assign #1 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);

    always_comb
        case(s)
            2'b00: y <= d0;
            2'b01: y <= d1;
            2'b10: y <= d2;
            2'b11: y <= d3;
        endcase
endmodule
```

**VHDL**

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

entity flopenr is -- flip-flop with asynchronous reset
    generic(width: integer);
    port(clk, reset: in  STD_LOGIC;
         en: in  STD_LOGIC;
         d: in  STD_LOGIC_VECTOR(width-1 downto 0);
         q: out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture asynchronous of flopenr is
begin
    process(clk, reset)
    begin
        if reset then  q <= CONV_STD_LOGIC_VECTOR(0, width);
        elsif rising_edge(clk) and en = '1' then
            q <= d;
        end if;
    end process;
end;

entity mux3 is -- three-input multiplexer
    generic(width: integer);
    port(d0, d1, d2: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s: in  STD_LOGIC_VECTOR(1 downto 0);
         y: out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture behave of mux3 is
begin
    process(all)
    begin
        case s is
            when "00" =>   y <= d0;
            when "01" =>   y <= d1;
            when "10" =>   y <= d2;
            when others => y <= d0;
        end case;
    end process;
end;

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

architecture behave of mux4 is
begin
    process(all)
    begin
        case s is
            when "00" =>   y <= d0;
            when "01" =>   y <= d1;
            when "10" =>   y <= d2;
            when "11" =>   y <= d3;
            when others => y <= d0;
        end case;
    end process;
end;
```

Exercise 7.27

We modify the MIPS multicycle processor to implement all instructions from Exercise 7.14.

**SystemVerilog**

```verilog
module top(input logic clk, reset,
           output logic [31:0] writedata, dataadr,
           output logic memwrite);

logic [31:0] pc, instr, readdata;

// instantiate processor and memories
mips mips(clk, reset, pc, instr, memwrite, dataadr, writedata, readdata);
imem imem(pc[7:2], instr);
dmem dmem(clk, memwrite, dataadr, writedata, readdata);
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 imem(input logic [5:0] a,
             output logic [31:0] rd);

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

initial
  $readmemh("memfile.dat",RAM);

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

module top(input logic clk, reset,
           output logic [31:0] writedata, adr,
           output logic memwrite);

logic [31:0] readdata;

// instantiate processor and memory
mips mips(clk, reset, adr, writedata, memwrite, readdata);
mem mem(clk, memwrite, adr, writedata, readdata);
endmodule

module mips(input logic clk, reset,
             output logic [31:0] writedata, memwrite, input logic [31:0] readdata);
```
logic zero, pcen, irwrite, regwrite,
alusrca, iord, memtoreg, regdst;
logic [2:0] alusrcb; // ANDI
logic [1:0] pcsr;
logic [2:0] alucontrol;
logic [5:0] op, funct;
logic [1:0] lb;       // LB/LBU
controller c(clk, reset, op, funct, zero,
               pcen, memwrite, irwrite, regwrite,
               alusrca, iord, memtoreg, regdst,
               alusrcb, pcsr, alucontrol, lb);  // LB/LBU
datapath dp(clk, reset,
               pcen, irwrite, regwrite,
               alusrca, iord, memtoreg, regdst,
               alusrcb, pcsr, alucontrol,
               lb,                                // LB/LBU
               op, funct, zero,
               adr, wridedata, readdata);
endmodule

module controller (input  logic clk, reset,
                   input  logic [5:0] op, funct,
                   input logic zero,
                   output logic pcen, memwrite,
                   output logic irwrite, regwrite,
                   output logic alusrca, iord,
                   output logic memtoreg, regdst,
                   output logic [2:0] alusrcb, // ANDI
                   output logic [1:0] pcsr,
                   output logic [2:0] alucontrol,
                   output logic [1:0] lb);       // LB/LBU

logic [1:0] aluop;
logic branch, pcwrite;
logic bne;   // BNE

// Main Decoder and ALU Decoder subunits.
maindec md(clk, reset, op,
          pcwrite, memwrite, irwrite, regwrite,
          alusrca, branch, iord, memtoreg, regdst,
          alusrcb, pcsr, aluop, bne, lb); // BNE, LB
aludec ad(funct, aluop, alucontrol);
assign pcen = pcwrite | (branch & zero) |
                  (bne & ~zero); // BNE
endmodule

module maindec(input  clk, reset,
               input [5:0] op,
               output pcwrite, memwrite,
               output irwrite, regwrite,
               output alusrca, branch,
               iord, memtoreg, regdst,
               output [2:0] alusrcb, // ANDI
               output [1:0] pcsr,
               output [1:0] aluop,
               output bne,       // BNE
               output [1:0] lb);   // LB/LBU

typedef enum logic [4:0] {FETCH, DECODE, MEMADR,
                          MEMRD, MEMWR, MEMWB, RTYPEEX, RTYPEWB, BEQEX,
                          ADDIEX, ADDIWB, JEX, ANDIEX, ANDIWB,
                          BNEEX, LBURD, LBRD} statetype;
statetype [4:0] state, nextstate;

parameter RTYPE = 6'b000000;
parameter LW = 6'b100011;
parameter SW = 6'b101011;
parameter BEQ = 6'b000100;
parameter ADDI = 6'b001000;
parameter J = 6'b000010;
parameter BNE = 6'b000101;
parameter LBU = 6'b100100;
parameter LB = 6'b100000;
parameter ANDI = 6'b001100;

logic [18:0] controls;  // ANDI, BNE, LBU

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

// next state logic
always_comb
case(state)
    FETCH:   nextstate <= DECODE;
    DECODE:  case(op)
        LW:      nextstate <= MEMADR;
        SW:      nextstate <= MEMADR;
        LB:      nextstate <= MEMADR;  // LB
        LBU:     nextstate <= MEMADR;  // LBU
        RTYPE:   nextstate <= RTYPEEX;
        BEQ:     nextstate <= BEQEX;
        ADDI:    nextstate <= ADDIEX;
        J:       nextstate <= JEX;
        BNE:     nextstate <= BNEEX;  // BNE
        ANDI:    nextstate <= ADDIEX;  // ANDI
        default: nextstate <= FETCH;  // should never happen
    endcase
    MEMADR:  case(op)
        LW:      nextstate <= MEMRD;
        SW:      nextstate <= MEMWR;
        LBU:     nextstate <= LBURD;  // LBU
        LB:      nextstate <= LBRD;   // LB
        default: nextstate <= FETCH;  // should never happen
    endcase
    MEMRD:   nextstate <= MEMWB;
    MEMWB:   nextstate <= FETCH;
    MEMWR:   nextstate <= FETCH;
    RTYPEEX: nextstate <= RTYPEWB;
    RTYPEWB: nextstate <= FETCH;
    BEQEX:   nextstate <= FETCH;
    ADDIEX:  nextstate <= ADDIWB;
    ADDIWB:  nextstate <= FETCH;
    JEX:     nextstate <= FETCH;
    ANDIEX:  nextstate <= ANDIWB;  // ANDI
    ANDIWB:  nextstate <= FETCH;  // ANDI
    BNEEX:   nextstate <= FETCH;  // BNE
    LBURD:   nextstate <= MEMWB;  // LBU
    LBRD:    nextstate <= MEMWB;  // LB
    default: nextstate <= FETCH;  // should never happen
endcase

// output logic
assign {pcwrite, memwrite, irwrite, regwrite,
     alusrca, branch, iord, memtoreg, regdst,
     bne, // BNE
     alusrcb, pcsrc,
     aluop,
     lb} = controls; // LBU

always_comb
begin
  case(state)
    FETCH: controls <= 19'b1010_00000_0_00100_00_00;
    DECODE: controls <= 19'b0000_00000_0_01100_00_00;
    MEMADR: controls <= 19'b0000_10000_0_01000_00_00;
    MEMWB: controls <= 19'b0001_00010_0_00000_00_00;
    MEMWR: controls <= 19'b0100_00100_0_00000_00_00;
    RTYPEEX: controls <= 19'b0000_10000_0_00000_00_00;
    RTYPEWB: controls <= 19'b0010_10000_0_00000_00_00;
    BEQEX: controls <= 19'b0000_11000_0_00001_01_00;
    ADDIEX: controls <= 19'b0000_10000_0_01000_00_00;
    ADDIWB: controls <= 19'b0000_10000_0_00000_00_00;
    JEX: controls <= 19'b0000_10000_0_00010_00_00;
    ANDIEX: controls <= 19'b0000_10000_1_00001_01_00;
    LBURD: controls <= 19'b0000_00100_0_00000_00_00;
    LBRD: controls <= 19'b0000_00100_0_00000_00_00;
    default: controls <= 19'b0000_xxxxx_x_xxxxx_xx_xx;
                   // should never happen
  endcase
endmodule

module aludec(input logic [5:0] funct,
           input logic [1:0] aluop,
           output logic [2:0] alucontrol);
begin
  always_comb
    case(aluop)
      2'b00: alucontrol <= 3'b010; // add
      2'b01: alucontrol <= 3'b110; // sub
      2'b11: alucontrol <= 3'b000; // and
      2'b10: case(funct) // RTYPE
        6'b100000: alucontrol <= 3'b010; // ADD
        6'b100010: alucontrol <= 3'b110; // SUB
        6'b100100: alucontrol <= 3'b000; // AND
        6'b100101: alucontrol <= 3'b001; // OR
        6'b101010: alucontrol <= 3'b111; // SLT
        default: alucontrol <= 3'bxxx; // ???
      endcase
    default: alucontrol <= 3'bxxx; // ???
  endcase
endmodule

module datapath(input logic clk, reset,
                 input logic pcen, irwrite,
                 input logic regwrite,
                 input logic alusrca, iord,
                 memtoreg, regdst,
                 input logic [2:0] alusrca,
                 input logic [1:0] pcsrc,
                 input logic [2:0] alucontrol,
                 input logic [1:0] lb, // LB/LBU
                 output logic [5:0] op, funct,
                 output logic zero,
                 output logic [31:0] adr, writedata,
                 input logic [31:0] readdata);
// Internal signals of the datapath module

logic [4:0] writereg;
logic [31:0] pcnext, pc;
logic [31:0] instr, data, srca, srcb;
logic [31:0] a;
logic [31:0] aluresult, aluout;
logic [31:0] signimm; // the sign-extended imm
logic [31:0] zeroimm; // the zero-extended imm
logic [31:0] signimmsh; // the sign-extended imm << 2
logic [31:0] w3, rd1, rd2;
logic [31:0] memdata, membytezext, membytesext; // LB / LBU
logic [7:0] membyte; // LB / LBU

// op and funct fields to controller
assign op = instr[31:26];
assign funct = instr[5:0];

// datapath
flopr #(32) pcreg(clk, reset, pcen, pcnext, pc);
mux2 #(32) adrmux(pc, aluout, iord, adr);
flopr #(32) instrreg(clk, reset, irwrite, readdata, instr);

// changes for LB / LBU
flopr #(32) datareg(clk, reset, memdata, data);
mux4 #(8) lbmux(readdata[31:24],
            readdata[23:16], readdata[15:8],
            readdata[7:0], aluout[1:0],
            membyte);
zeroext8_32 lbze(membyte, membytezext);
signext8_32 lbse(membyte, membytesext);
mux3 #(32) datamux(readdata, membytezext, membytesext,
                       lb, memdata);

mux2 #(5) regdstmux(instr[20:16],
                instr[15:11], regdst, writereg);
mux2 #(32) wdmux(aluout, data, memtoreg, w3);
regfile rf(clk, regwrite, instr[25:21],
          instr[20:16],
          writereg, w3, rd1, rd2);
signext se(instr[15:0], signimm);
zeroext ze(instr[15:0], zeroimm); // ANDI
sl2 immsh(signimm, signimmsh);
flopr #(32) areg(clk, reset, rd1, a);
flopr #(32) breg(clk, reset, rd2, wdata);
mux2 #(32) srcamux(pc, a, alusrca, srca);
mux5 #(32) srcbmux(wdata, 32'b100,
            signimm, signimmsh,
            zeroimm, // ANDI
            alusrcb, srcb);
alu alu(srca, srcb, alucontrol, aluresult, zero);
flopr #(32) alureg(clk, reset, aluresult, aluout);
mux3 #(32) pcmux(aluresult, aluout,
               {pc[31:28], instr[25:0], 2'b00},
               pcsrc, pcnext);

endmodule

module alu(input logic [31:0] A, B,
           input logic [2:0] F,
           output logic [31:0] Y, output Zero);
logic [31:0] S, Bout;
assign S = A + Bout + F[2];

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

assign Zero = (Y == 32'b0);
endmodule

// mux5 is needed for ANDI
module mux5 #(parameter WIDTH = 8) (input [WIDTH-1:0] d0, d1, d2, d3, d4, input [2:0] s, output reg [WIDTH-1:0] y);

always_comb
  case(s)
    3'b000: y <= d0;
    3'b001: y <= d1;
    3'b010: y <= d2;
    3'b011: y <= d3;
    3'b100: y <= d4;
  endcase
endmodule

// zeroext is needed for ANDI
module zeroext(input [15:0] a, output [31:0] y);
  assign y = {16'b0, a};
endmodule

// zeroext8_32 is needed for LBU
module zeroext8_32(input logic [7:0] a, output logic [31:0] y);
  assign y = {24'b0, a};
endmodule

// signext8_32 is needed for LB
module signext8_32(input logic [7:0] a, output logic [31:0] y);
  assign y = {{24{a[7]}}, a};
endmodule

module alu(input logic [31:0] A, B, input logic [3:0] F, input logic [4:0] shamt, // SRL output logic [31:0] Y, output logic Zero);

logic [31:0] S, Bout;
assign S = A + Bout + F[3]; // SRL
always_comb
    case (#[2:0])
        3'b000: Y = A & Bout;
        3'b001: Y = A | Bout;
        3'b010: Y = S;
        3'b011: Y = S[31];
        3'b100: Y = (Bout >> shamt);  // SRL
    endcase

    assign Zero = (Y == 32'b0);
endmodule

module regfile(input logic clk,
                input logic we3,
                input logic [4:0] ra1, ra2, wa3,
                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
    // write third port on rising edge of clk
    // register 0 hardwired to 0
    always_ff @(posedge clk)
        if (we3) rf[wa3] <= wd3;

    assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
    assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
endmodule

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

    assign y = a + b;
endmodule

module sl2(input logic [31:0] a,
            output logic [31:0] y);

    // shift left by 2
    assign y = {a[29:0], 2'b00};
endmodule

module signext(input logic [15:0] a,
                output logic [31:0] y);

    assign y = {{16{a[15]}}, 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 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

**VHDL**

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mips is -- multicycle MIPS processor
  port(clk, reset: in  STD_LOGIC;
       adr:      out STD_LOGIC_VECTOR(31 downto 0);
       writedata:  inout STD_LOGIC_VECTOR(31 downto 0);
       memwrite:     out STD_LOGIC;
       readdata:   in  STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of mips is
  component controller
    port(clk, reset: in  STD_LOGIC;
         op, funct:   in  STD_LOGIC_VECTOR(5 downto 0);
         zero: in  STD_LOGIC;
         pcen, memwrite:     out STD_LOGIC;
         irwrite, regwrite:  out STD_LOGIC;
         alusrca, iord: out STD_LOGIC;
         memtoreg, regdst:   out STD_LOGIC;
         alusrcb:    out STD_LOGIC_VECTOR(2 downto 0); --andi
         pcsrc:     out STD_LOGIC_VECTOR(1 downto 0);
         alucontrol:  out STD_LOGIC_VECTOR(2 downto 0);
         lb:         out STD_LOGIC_VECTOR(1 downto 0)); --lb, lbu
  end component;
  component datapath
    port(clk, reset: in  STD_LOGIC;
         pcen, irwrite:     in  STD_LOGIC;
         regwrite, alusrca: in  STD_LOGIC;
         iord, memtoreg:    in  STD_LOGIC;
         regdst:in  STD_LOGIC;
         alusrcb:    in  STD_LOGIC_VECTOR(2 downto 0); --andi
         pcsrc:     in  STD_LOGIC_VECTOR(1 downto 0);
         alucontrol:  in  STD_LOGIC_VECTOR(2 downto 0);
         lb:         in  STD_LOGIC_VECTOR(1 downto 0)); --lb / lbu
  end component;
  signal zero, pcen, irwrite, regwrite, alusrca, iord, memtoreg, regdst: STD_LOGIC;
  signal alusrcb: STD_LOGIC_VECTOR(2 downto 0); --andi
  signal pcsrc: STD_LOGIC_VECTOR(1 downto 0);
  signal pcsrc: STD_LOGIC_VECTOR(2 downto 0);
  signal alucontrol: STD_LOGIC_VECTOR(1 downto 0);
  signal lb:         STD_LOGIC_VECTOR(1 downto 0)); --lb / lbu
begin
  c: controller port map(clk, reset, op, funct, zero,
      pcen, memwrite, irwrite, regwrite,
      alusrca, iord, memtoreg, regdst,
      alusrcb, pcsrc, alucontrol, lb); --lb /lbu
  dp: datapath port map(clk, reset,
      pcen, irwrite, regwrite,
      alusrca, iord, memtoreg, regdst,
      alusrcb, pcsrc, alucontrol,
      lb, --lb / lbu
      readdata, op, funct, zero,
      adr, writedata, readdata);
end;
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity controller is -- multicycle control decoder
    port(clk, reset: in STD_LOGIC;
        op, funct: in STD_LOGIC_VECTOR(5 downto 0);
        zero: in STD_LOGIC;
        pcen, memwrite: out STD_LOGIC;
        irwrite, regwrite: out STD_LOGIC;
        alusrca, iord: out STD_LOGIC;
        memtoreg, regdst: out STD_LOGIC;
        alusrcb: out STD_LOGIC_VECTOR(2 downto 0); --andi
        pcsr: out STD_LOGIC_VECTOR(1 downto 0);
        alucontrol: out STD_LOGIC_VECTOR(2 downto 0);
        lb: out STD_LOGIC_VECTOR(1 downto 0)); --lb, lbu
end;
architecture struct of controller is
    component maindec
        port(clk, reset: i
            op: i
                pcwrite, memwrite:  out STD_LOGIC;
                irwrite, regwrite:  out STD_LOGIC;
                alusrca, branch:    out STD_LOGIC;
                iord, memtoreg:     out STD_LOGIC;
                regdst: out STD_LOGIC;
                alusrcb:   out STD_LOGIC_VECTOR(2 downto 0); --andi
                pcsr:     out STD_LOGIC_VECTOR(1 downto 0);
                aluop:     out STD_LOGIC_VECTOR(1 downto 0);
                lb:        out STD_LOGIC_VECTOR(1 downto 0)); --lb / lbu
    end component;
    component aludec
        port(funct: in STD_LOGIC_VECTOR(5 downto 0);
            aluop: in STD_LOGIC_VECTOR(1 downto 0);
            alucontrol: out STD_LOGIC_VECTOR(2 downto 0));
    end component;
    signal aluop: STD_LOGIC_VECTOR(1 downto 0);
    begin
        md: maindec port map(clk, reset, op,
            pcwrite, memwrite, irwrite, regwrite,
            alusrca, branch, iord, memtoreg, regdst,
            alusrcb, pcsr, aluop, bne, lb);
        ad: aludec port map(funct, aluop, alucontrol);
        pcen <= pcwrite or (branch and zero) or (bne and (not zero)); --bne
    end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity maindec is -- main control decoder
    port(clk, reset: in STD_LOGIC;
        op: in STD_LOGIC_VECTOR(5 downto 0);
        pcwrite, memwrite:  out STD_LOGIC;
        irwrite, regwrite:  out STD_LOGIC;
        alusrca, branch:    out STD_LOGIC;
        iord, memtoreg:     out STD_LOGIC;
        regdst: out STD_LOGIC;
        alusrcb:   out STD_LOGIC_VECTOR(2 downto 0); --andi
        pcsr:     out STD_LOGIC_VECTOR(1 downto 0);
        aluop:     out STD_LOGIC_VECTOR(1 downto 0);
        bne:       out STD_LOGIC;
        aluop:     out STD_LOGIC_VECTOR(1 downto 0)); --lb / lbu
end;
architecture behave of maindec is
    type statetype is (FETCH, DECODE, MEMADR, MEMRD, MEMWB, MEMWR,
        RTYPEEX, RTYPEWB, BEQEX, ADDIEX, ADDIIWB, JEX,
        ...)
ANDIEX, ANDIWB, BNEEX, LBURD, LBRD);

signal state, nextstate: statetype;
signal controls: STD_LOGIC_VECTOR(18 downto 0);
begin
  -- state register
  process(clk, reset) begin
    if reset then state <= FETCH;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

  -- next state logic
  process(all) begin
    case state is
      when FETCH =>
        nextstate <= DECODE;
      when DECODE =>
        case op is
          when "100011" => nextstate <= MEMADR; -- LW
          when "101011" => nextstate <= MEMADR; -- SW
          when "100000" => nextstate <= MEMADR; -- LB
          when "100100" => nextstate <= MEMADR; -- LBU
          when "000000" => nextstate <= RTYPEEX; -- RTYPE
          when "000100" => nextstate <= BEQEX; -- BEQ
          when "0001000" => nextstate <= ADDIEX; -- ADDI
          when "000010" => nextstate <= JEX; -- J
          when "000101" => nextstate <= ORIEX; -- BNE
          when "001100" => nextstate <= ORIEX; -- ANDI
          when others => nextstate <= FETCH; -- should never happen
        end case;
      when MEMADR =>
        case op is
          when "100011" => nextstate <= MEMRD;
          when "101011" => nextstate <= MEMWR;
          when "100000" => nextstate <= LBURD; -- LB
          when "100100" => nextstate <= LBURD; -- LBU
          when others => nextstate <= FETCH; -- should never happen
        end case;
      when MEMRD =>
        nextstate <= MEMWB;
      when MEMWB =>
        nextstate <= FETCH;
      when MEMWR =>
        nextstate <= FETCH;
      when RTYPEEX =>
        nextstate <= RTYPEWB;
      when RTYPEWB =>
        nextstate <= FETCH;
      when BEQEX =>
        nextstate <= FETCH;
      when ADDIEX =>
        nextstate <= ADDIWB;
      when JEX =>
        nextstate <= FETCH;
      when ANDIEX =>
        nextstate <= ANDIWB; -- ANDI
      when ANDIWB =>
        nextstate <= FETCH; -- ANDI
      when BNEEX =>
        nextstate <= FETCH; -- BNE
      when LBURD =>
        nextstate <= MEMWB; -- LB
      when LBRD =>
        nextstate <= MEMWB; -- LB
      when others =>
        nextstate <= FETCH; -- should never happen
    end case;
  end process;

  -- output logic
  process(all) begin
    case state is
      when FETCH => controls <= "1010_00000_0_00100_00_00";
      when DECODE => controls <= "0000_00000_0_01100_00_00";
      when MEMADR => controls <= "0000_10000_0_01000_00_00";
      when MEMRD => controls <= "0000_00100_0_00000_00_00";
      when MEMWB => controls <= "0001_00010_0_00000_00_00";
      when MEMWR => controls <= "0100_00100_0_00000_00_00";
      when RTYPEEX => controls <= "0000_10000_0_00000_10_00";
      when RTYPEWB => controls <= "0001_00001_0_00000_00_00";
    end case;
  end process;

end chapter 7;
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity aludec is -- ALU control decoder
port(funct: in STD_LOGIC_VECTOR(5 downto 0);
    aluop: in STD_LOGIC_VECTOR(1 downto 0);
    alucontrol: out STD_LOGIC_VECTOR(2 downto 0));
end;
architecture behave of aludec is
begin
  process(all) begin
    case aluop is
      when "00" => alucontrol <= "010"; -- add (for lb/sb/addi)
      when "01" => alucontrol <= "110"; -- sub (for beq)
      when "11" => alucontrol <= "000"; -- and (for andi)
      when others => case funct is
        when "100000" => alucontrol <= "010"; -- add
        when "100010" => alucontrol <= "110"; -- sub
        when "100100" => alucontrol <= "000"; -- and
        when "100101" => alucontrol <= "001"; -- or
        when "101010" => alucontrol <= "111"; -- slt
        when others => alucontrol <= "--"; -- ???
      end case;
    end case;
  end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all; use IEEE.STD_LOGIC_ARITH.all;
entity datapath is -- MIPS datapath
port(clk, reset: in STD_LOGIC;
    pcen, irwrite: in STD_LOGIC;
    regwrite, alusrca: in STD_LOGIC;
    iord, memtoreg: in STD_LOGIC;
    regdst: in STD_LOGIC;
end;
architecture behave of datapath is
begin
  pcwrite <= controls(18);
  memwrite <= controls(17);
  irwrite <= controls(16);
  regwrite <= controls(15);
  alusrca <= controls(14);
  branch <= controls(13);
  iord <= controls(12);
  memtoreg <= controls(11);
  regdst <= controls(10);
  bne <= controls(9);
  alusrcb <= controls(8 downto 6);
  pcsr <= controls(5 downto 4);
  aluop <= controls(3 downto 1);
  lb <= controls(0);
end;
alusrcb: in STD_LOGIC_VECTOR(2 downto 0); --andi
pcsrc: in STD_LOGIC_VECTOR(1 downto 0);
alucontrol: in STD_LOGIC_VECTOR(2 downto 0);
lb: in STD_LOGIC_VECTOR(1 downto 0); --lb / lbu
readdata: out STD_LOGIC_VECTOR(31 downto 0);
zero: out STD_LOGIC;
adr: out STD_LOGIC_VECTOR(31 downto 0);
writedata: inout STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of datapath is
  component alu
    port(A, B: in STD_LOGIC_VECTOR(31 downto 0);
         F: in STD_LOGIC_VECTOR(2 downto 0);
         Y: out STD_LOGIC_VECTOR(31 downto 0);
         Zero: out STD_LOGIC);
  end component;
  component regfile
    port(clk: in STD_LOGIC;
         we3: in STD_LOGIC;
         ra1, ra2, wa3: in STD_LOGIC_VECTOR(4 downto 0);
         wd3: in STD_LOGIC_VECTOR(31 downto 0);
         rd1, rd2: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component adder
    port(a, b: in STD_LOGIC_VECTOR(31 downto 0);
         y: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component sl2
    port(a: in STD_LOGIC_VECTOR(31 downto 0);
         y: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component signext
    port(a: in STD_LOGIC_VECTOR(15 downto 0);
         y: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component zeroext
    port(a: in STD_LOGIC_VECTOR(15 downto 0);
         y: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component flopr generic(width: integer);
    port(clk, reset: in STD_LOGIC;
         d: in STD_LOGIC_VECTOR(width-1 downto 0);
         q: out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component flopenr generic(width: integer);
    port(clk, reset: in STD_LOGIC;
         en: in STD_LOGIC;
         d: in STD_LOGIC_VECTOR(width-1 downto 0);
         q: out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component mux2 generic(width: integer);
    port(d0, d1: in STD_LOGIC_VECTOR(width-1 downto 0);
         s: in STD_LOGIC;
         y: out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component mux3 generic(width: integer);
    port(d0, d1, d2: in STD_LOGIC_VECTOR(width-1 downto 0);
         s: in STD_LOGIC_VECTOR(1 downto 0);
         y: out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component mux4 generic(width: integer);
    port(d0, d1, d2, d3: in STD_LOGIC_VECTOR(width-1 downto 0);
         s: in STD_LOGIC_VECTOR(1 downto 0);
         y: out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
end;
end component;
component mux5 generic(width: integer);
  port(d0, d1, d2, d3, d4: 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;
-- lb / lbu
component zeroext8_32
  port(a: in  STD_LOGIC_VECTOR(7 downto 0);
       y: out STD_LOGIC_VECTOR(31 downto 0));
end component;
component signext8_32
  port(a: in  STD_LOGIC_VECTOR(7 downto 0);
       y: out STD_LOGIC_VECTOR(31 downto 0));
end component;

signal writereg: STD_LOGIC_VECTOR(4 downto 0);
signal pcnext, pc, instr, data, srca, srcb, a,
    alu result, aluout, signimm, signimmsh, wd3, rd1, rd2, pcjump:
    STD_LOGIC_VECTOR(31 downto 0);
-- lb / lbu
signal memdata, membytezext, membytesext: STD_LOGIC_VECTOR(31 downto 0);
signal membyte: STD_LOGIC_VECTOR(7 downto 0);

begin
  -- op and funct fields to controller
  op <= instr(31 downto 26);
  funct <= instr(5 downto 0);
  -- datapath
  pcreg: floopenr generic map(32) port map(clk, reset, pcen, pcnext, pc);
  adrmux: mux2 generic map(32) port map(pc, aluout, iord, adr);
  instrreg: floopenr generic map(32) port map(clk, reset, irwrite,
                                          readdata, instr);
  -- changes for lb / lbu
  datareg: floopenr generic map(32) port map(clk, reset, memdata, data);
  lbmux:  mux4 generic map(8) port map(readdata(31 downto 24),
                                     readdata(23 downto 16),
                                     readdata(15 downto 8),
                                     readdata(7 downto 0),
                                     aluout(1 downto 0), membyte);
  lbze: zeroext8_32 port map(membyte, membytezext);
  lbse: signext8_32 port map(membyte, membytesext);
  datamux: mux3 generic map(32) port map(readdata, membytezext, membytesext,
                                          lb, memdata);
  datareg: floopenr generic map(32) port map(clk, reset, readdata, data);
  regdstmux: mux2 generic map(5) port map(instr(20 downto 16),
                                       instr(15 downto 11),
                                       regdst, writereg);
  wdmux: mux2 generic map(32) port map(aluout, data, memtoreg, wd3);
  rf: regfile port map(clk, regwrite, instr(25 downto 21),
                        instr(20 downto 16),
                        writereg, wd3, rd1, rd2);
  se: signext port map(instr(15 downto 0), signimm);
  ze: zeroext port map(instr(15 downto 0), zeroimm);  --andi
  immsh: sl2 port map(signimm, signimmsh);
  areg: flopr generic map(32) port map(clk, reset, rd1, a);
  breg: flopr generic map(32) port map(clk, reset, rd2, writedata);
  srcamux: mux2 generic map(32) port map(pc, a, alusrca, srca);
  srcbmul: flopr generic map(32) port map(writedata,
                                          "0000000000000000000000000000000100",
                                          signimm, signimmsh, zeroimm, alusrcb, srcb);  --andi
  alu32: alu port map(srca, srcb, alucontrol, aluresult, zero);
  alureg: flopr generic map(32) port map(clk, reset, aluresult, alout);
pcjump <= pc(31 downto 28)&instr(25 downto 0)&"00";

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

architecture asynchronous of flopenr is
begin
  process(clk, reset)
  begin
    if reset then  q <= CONV_STD_LOGIC_VECTOR(0, width);
    elsif rising_edge(clk) and en = '1' then
      q <= d;
    end if;
  end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux3 is -- three-input multiplexer
  generic(width: integer);
  port(d0, d1, d2: in STD_LOGIC_VECTOR(width-1 downto 0);
       s:    in STD_LOGIC_VECTOR(1 downto 0);
       y:    out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture behave of mux3 is
begin
  process(all)
  begin
    case s is
      when "00" =>   y <= d0;
      when "01" =>   y <= d1;
      when "10" =>   y <= d2;
      when others => y <= d0;
    end case;
  end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux4 is -- four-input multiplexer
  generic(width: integer);
  port(d0, d1, d2, d3: in STD_LOGIC_VECTOR(width-1 downto 0);
       s:    in STD_LOGIC_VECTOR(1 downto 0);
       y:    out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture behave of mux4 is
begin
  process(all)
  begin
    case s is
      when "00" =>   y <= d0;
      when "01" =>   y <= d1;
      when "10" =>   y <= d2;
      when "11" =>   y <= d3;
      when others => y <= d0; -- should never happen
    end case;
  end process;
end;
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux5 is -- five-input multiplexer
  generic(width: integer);
  port(d0, d1, d2, d3, d4: in  STD_LOGIC_VECTOR(width-1 downto 0);
       s:   in             STD_LOGIC_VECTOR(2 downto 0);
       y:   out            STD_LOGIC_VECTOR(width-1 downto 0));
end;
architecture behave of mux5 is
begin
  process(all) begin
    case s is
    when "000" =>   y <= d0;
    when "001" =>   y <= d1;
    when "010" =>   y <= d2;
    when "011" =>   y <= d3;
    when "100" =>   y <= d4;
    when others => y <= d0; -- should never happen
    end case;
  end process;
end;

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

entity alu is
  port(A, B:     in  STD_LOGIC_VECTOR(31 downto 0);
       F:        in  STD_LOGIC_VECTOR(2 downto 0);
       Y:        out STD_LOGIC_VECTOR(31 downto 0);
       Zero:     out STD_LOGIC);
end;
architecture synth of alu is
signal S, Bout:      STD_LOGIC_VECTOR(31 downto 0);
begin
  Bout <= (not B) when (F(3) = '1') else B;
  S <= A + Bout + F(3);
  Zero <= '1' when (Y = X"00000000") else '0';
  process(all) begin
    case F(1 downto 0) is
    when "00" => Y <= A and Bout;
    when "01" => Y <= A or Bout;
    when "10" => Y <= S;
    when "11" => Y <=
                 ("00000000000000000000000000000000" & S(31));
    when others => Y <= X"00000000";
    end case;
  end process;
end;

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

entity signext is
  port(A:       in  STD_LOGIC_VECTOR(15 downto 0);
       Y:       out STD_LOGIC_VECTOR(31 downto 0));
end;
architecture synth of signext is
begin
  Y <= (15 downto 0 => a, others => a(15));
end;

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

entity zeroext is
  port(A: in  STD_LOGIC_VECTOR(15 downto 0);
       Y: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture synth of zeroext is
begin
  Y <= (15 downto 0 => a, others => '0');
end;

-- for lb / bzu
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.std_logic_unsigned.all;

entity signext8_32 is
  port(A: in  STD_LOGIC_VECTOR(7 downto 0);
       Y: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture synth of signext8_32 is
begin
  Y <= (7 downto 0 => a, others => a(7));
end;

-- for lb / bzu
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.std_logic_unsigned.all;

entity zeroext8_32 is
  port(A: in  STD_LOGIC_VECTOR(7 downto 0);
       Y: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture synth of zeroext8_32 is
begin
  Y <= (7 downto 0 => a, others => '0');
end;

Exercise 7.29

$s0$ is written, $t4$ and $t5$ are read in cycle 5.
Exercise 7.31

![FIGURE 7.9 Abstract pipeline for Exercise 7.31]

Exercise 7.33

It takes $3 + 6(10) + 3 = \textbf{66 clock cycles}$ to issue all the instructions.

# instructions = $3 + 5(10) + 2 = 55$

$\text{CPI} = \frac{66 \text{ clock cycles}}{55 \text{ instructions}} = 1.2.$

Exercise 7.35
We must also write new equations for the flush signal, $FlushE$.

$FlushE = lwstall \text{ OR branchstall OR JumpD}$

**Exercise 7.37**

The critical path is the Decode stage, according to Equation 7.5:

$$T_{c3} = \max(30 + 250 + 20, 2(150 + 25 + 40 + 15 + 25 + 20), 30 + 25 + 25 + 200 + 20, 30 + 220 + 20, 2(30 + 25 + 100)) = \max(300, 550, 300, 270, 310) = 550 \text{ ps}. \quad \text{The next slowest stage is 310 ps for the writeback stage, so it doesn’t make sense to make the Decode stage any faster than that.}$$

The slowest unit in the Decode stage is the register file read (150 ps). We need to reduce the cycle time by 550 - 310 = 240 ps. Thus, we need to reduce the register file read delay by $240/2 = 120$ ps to $(150 - 120) = 30 \text{ ps}$. The new cycle time is **310 ps**.

**Exercise 7.39**

$$\text{CPI} = 0.25(1+0.5*6) + 0.1(1) + 0.11(1+0.3*1)+0.02(2)+0.52(1) = 1.8$$

$$\text{Execution Time} = (100 \times 10^9 \text{ instructions})(1.8 \text{ cycles/instruction})(400 \times 10^{-12} \text{ s/cycle}) = 72 \text{s}$$
Exercise 7.41

MIPS Pipelined Processor Hazard Unit

SystemVerilog

module hazard(input logic [4:0] rsD, rtD, rsE, rtE, 
input logic [4:0] writeregE, writeregM, writeregW, 
input logic regwriteE, regwriteM, regwriteW, 
input logic memtoregE, memtoregM, branchD, 
output logic forwardaD, forwardbD, 
output logic [1:0] forwardaE, forwardbE, 
output logic stallF, stallD, 
output logic lwstallD, branchstallD;

logic lwstallD, branchstallD;

// forwarding sources to D stage (branch equality)
assign forwardaD = (rsD !=0 & rsD == writeregM & regwriteM);
assign forwardbD = (rtD !=0 & rtD == writeregM & regwriteM);

// forwarding sources to E stage (ALU)
always_comb begin
    forwardaE = 2'b00; forwardbE = 2'b00;
    if (rsE != 0) then 
        if (rsE == writeregM & regwriteM) then 
            forwardaE = 2'b10;
        else if (rsE == writeregW & regwriteW) then 
            forwardaE = 2'b01;
    if (rtE != 0) then 
        if (rtE == writeregM & regwriteM) then 
            forwardbE = 2'b10;
        else if (rtE == writeregW & regwriteW) then 
            forwardbE = 2'b01;
end

// stalls
assign #1 lwstallD = memtoregE & (rtE == rsD | rtE == rtD);
assign #1 branchstallD = branchD & 
    (regwriteE & 
    (writeregE == rsD | writeregE == rtD) | 
    memtoregM & 
    (writeregM == rsD | writeregM == rtD));

assign #1 stallD = lwstallD | branchstallD;
assign #1 stallF = stallD;

// stalling D stalls all previous stages
assign #1 flushE = stallD;
// stalling D flushes next stage

// Note: not necessary to stall D stage on store
// if source comes from load;
// instead, another bypass network could be added from W to M
endmodule

VHDL

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity hazard is -- hazard unit
    port(rsD, rtD, rsE, rtE: in STD_LOGIC_VECTOR(4 downto 0);
        writeregE, writeregM, writeregW: in STD_LOGIC_VECTOR(4 downto 0);
        regwriteE, regwriteM, regwriteW: in STD_LOGIC;
        memtoregE, memtoregM, branchD: in STD_LOGIC;
        forwardaE, forwardbE: out STD_LOGIC;
        forwardaD, forwardbD: out STD_LOGIC_VECTOR(1 downto 0);
        stallF, flushE: out STD_LOGIC;
        stallD: inout STD_LOGIC);
end;

architecture behave of hazard is

logic lwstallD, branchstallD;

-- forwarding sources to D stage (branch equality)
begin
    if (rsD != "00000") then 
        if (rsD == writeregM) then 
            forwardaD <= "10";
        else if (rsD == writeregW) then 
            forwardaD <= "01";
    end if;

    if (rtD != "00000") then 
        if (rtD == writeregM) then 
            forwardbD <= "10";
        else if (rtD == writeregW) then 
            forwardbD <= "01";
    end if;
end

-- forwarding sources to E stage (ALU)
process(all) begin
    if (rsE != "00000") then 
        if (rsE == writeregM) then 
            forwardaE <= "10";
        else if (rsE == writeregW) then 
            forwardaE <= "01";
    end if;

    if (rtE != "00000") then 
        if (rtE == writeregM) then 
            forwardbE <= "10";
        else if (rtE == writeregW) then 
            forwardbE <= "01";
    end if;
end process;

-- stalls
lwstallD <= '1' when (memtoregE = '1') and (rtE = rsD) or (rtE = rtD)) else '0';
branchstallD <= '1' when (branchD = '1') and 
    (writeregE = '1') and 
    (writeregM = '1') and 
    (memtoregE = '1') and 
    (memtoregM = '1') and 
    (writeregE = rsD) or (writeregM = rtD)) else '0';
stallD <= (lwstallD or branchstallD) after 1 ns;
flushE <= stallD after 1 ns; -- stalling D flushes all previous stages
stallF <= stallD after 1 ns; -- stalling D flushes next stage

-- not necessary to stall D stage on store if source comes from load;
-- instead, another bypass network could be added from W to M
end;
FIGURE 7.10 Hazard unit hardware for forwarding to the Execution stage
FIGURE 7.11 Hazard unit hardware for forwarding to the Decode stage
Question 7.1

A pipelined microprocessor with $N$ stages offers an ideal speedup of $N$ over nonpipelined microprocessor. This speedup comes at the cost of little extra hardware: pipeline registers and possibly a hazard unit.

Question 7.3

A hazard in a pipelined microprocessor occurs when the execution of an instruction depends on the result of a previously issued instruction that has not completed executing. Some options for dealing with hazards are: (1) to have the compiler insert nops to prevent dependencies, (2) to have the compiler reorder the code to eliminate dependencies (inserting nops when this is impossible), (3) to have the hardware stall (or flush the pipeline) when there is a dependency, (4)
to have the hardware forward results to earlier stages in the pipeline or stall when that is impossible.

Options (1 and 2): Advantages of the first two methods are that no added hardware is required, so area and, thus, cost and power is minimized. However, performance is not maximized in cases where nops are inserted.

Option 3: The advantage of having the hardware flush or stall the pipeline as needed is that the compiler can be simpler and, thus, likely faster to run and develop. Also, because there is no forwarding hardware, the added hardware is minimal. However, again, performance is not maximized in cases where forwarding could have been used instead of stalling.

Option 4: This option offers the greatest performance advantage but also costs the most hardware for forwarding, stalling, and flushing the pipeline as necessary because of dependencies.

A combination of options 2 and 4 offers the greatest performance advantage at the cost of more hardware and a more sophisticated compiler.


**CHAPTER 8**

---

**Exercise 8.1**

Answers to this question 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 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 and access pattern:  
0 4 8  
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

Figure 8.1 shows where each address maps for each cache configuration.

```
Set 15
  7C
  78
  74
  70

Set 7
  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 (c) 2-way assoc
| 7C | 9C 1C | 7C 9C 1C | 7C 9C 1C |
| 78 | 98 18 | 78 98 18 | 78-7C |
| 74 | 94 14 | 74 94 14 | 70-74 |
| 70 | 90 10 | 70 90 10 | 20-24 |
| 4C 8C C | 4C 8C C | 4C 8C C | 4C 8C C |
| 48 88 8 | 48 88 8 | 98-9C 18-1C | 98-9C 18-1C |
| 44 84 4 | 44 84 4 | 48-4C 88-8C 8-C |
| 40 80 0 | 40 80 0 20 | 40-44 80-84 0-4 |

(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)

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Tag</th>
<th>Set</th>
<th>Offset Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>16 bits</td>
<td>13 bits</td>
<td>1 bit</td>
</tr>
<tr>
<td>31:16</td>
<td>15:3</td>
<td>2</td>
<td>0:0</td>
</tr>
</tbody>
</table>

(b) Each tag is 16 bits. There are 32Kwords / (2 words / block) = 16K blocks and each block needs a tag: 16 × 16K = 2^{18} = 256 Kbits of tags.  
(c) Each cache block requires: 2 status bits, 16 bits of tag, and 64 data bits, thus each set is 2 × 82 bits = **164 bits**.  
(d) 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 128KB / (8KB/RAM) = 16 RAMs to hold the data and 64 bits / (4 pins/RAM) = 16 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 × 4-bit RAM. We use a 16K × 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)
\[ AMAT = t_{\text{cache}} + MR_{\text{cache}} t_{\text{MM}} \]

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 cycles** (for a load)
    CPI = 31 + 3 = **34 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**

1 million gigabytes of hard disk \( \approx 2^{20} \times 2^{30} = 2^{50} \) bytes = 1 petabytes
10,000 gigabytes of hard disk \( \approx 2^{14} \times 2^{30} = 2^{44} \) bytes = 16 terabytes

Thus, the system would need **44 bits** for the physical address and **50 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).

(h) The total table size is $3 \times 2^{38}$ bytes.

Exercise 8.23

(a) 1 valid bit + 19 data bits (PPN) + 38 tag bits (VPN) x 128 entries = 58 * 128 bits = 7424 bits

(b) 128 x 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:

$2^{11} \times 10$ bits = 20480 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) $2^{32}$ bytes = 4 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.

Exercise 8.29

(a)

```mips
# MIPS code for Traffic Light FSM
addi $t0, $0, 0xC   # $t0 = green / red
addi $t1, $0, 0x14  # $t1 = yellow / red
addi $t2, $0, 0x21  # $t2 = red / green
addi $t3, $0, 0x22  # $t3 = red / yellow

Start: sw   $t2, 0xF004($0)  # lights = red / green
S0:    lw   $t4, 0xF000($0)  # $t4 = sensor values
       andi $t4, $t4, 0x2    # $t4 = TA
       bne  $t4, $0,  S0     # if TA == 1, loop back to S0

S1:    sw   $t3, 0xF004($0)  # lights = red / yellow
       sw   $t0, 0xF004($0)  # lights = green / red

S2:    lw   $t4, 0xF000($0)  # $t4 = sensor values
       andi $t4, $t4, 0x1    # $t4 = TB
       bne  $t4, $0,  S2     # if TB == 1, loop back to S2

S3:    sw   $t1, 0xF004($0)  # lights = yellow / red
       j    Start
```
(c) Address Decoder for Exercise 8.29

SystemVerilog

module addrdec(input logic [31:0] addr,
                input logic memwrite,        // writedata control
                output logic WELights, Mwrite,
                output logic rdselect);      // readdata control

parameter T = 16’hF000; // traffic sensors
parameter Lights = 16’hF004; // traffic lights

logic [15:0] addressbits;

assign addressbits = addr[15:0];

always_comb
if (addr[31:16] == 16’hFFFF) begin  // writedata control
    if (memwrite)
        if (addressbits == Lights)
            {WELights, Mwrite, rdselect} = 3’b100;
        else
            {WELights, Mwrite, rdselect} = 3’b010;
    else
        {WELights, Mwrite, rdselect} = 3’b000;
end

else
    {WELights, Mwrite, rdselect} = [1’b0, memwrite, 1’b0];

endmodule

VHDL

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity addrdec is -- address decoder
    port(addr:     in STD_LOGIC_VECTOR(31 downto 0);
         memwrite: in STD_LOGIC;
         WELights, Mwrite, rdselect: out STD_LOGIC);
    end;

architecture struct of addrdec is
    begin
        process(all) begin
            if (addr(31 downto 16) = X"FFFF") then
                if (memwrite = '1') then
                    if (addr(15 downto 0) = X"F004") then     -- traffic lights
                        WELights <= '1'; Mwrite <= '0'; rdselect <= '0';
                    else
                        WELights <= '0'; Mwrite <= '1'; rdselect <= '0';
                    end if;
                end if;
            -- readdata control
            else
                if ( addr(15 downto 0) = X"F000" ) then  -- traffic sensors
                    WELights <= '0'; Mwrite <= '0'; rdselect <= '1';
                else
                    WELights <= '0'; Mwrite <= '0'; rdselect <= '0';
                end if;
            end if;
            -- not a memory-mapped address
            else
                WELights <= '0'; Mwrite <= memwrite; rdselect <= '0';
            end process;
        end;
end;
rect-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.

**Question 8.5**

No, addresses used for memory-mapped I/O may not be cached. Otherwise, repeated reads to the I/O device would read the old (cached) value. Likewise, repeated writes would write to the cache instead of the I/O device.