CHAPTER 1

SOLUTIONS

to odd-numbered exercises

2 CHAPTER

CHAPTER 1

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.

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

not respecify them when he uses the same windows in another room. This will save him both design time and, thus, money. He could also save by buying some items (like windows) in bulk.

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

1.7 $2^{16} = 65536$ numbers.

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

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

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

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

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

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

1.21 (a) 00000101; (b) 11111010

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

1.25 15 greater than 0, 16 less than 0; 15 greater and 15 less for sign/magnitude

1.27 8

1.29 46.566 gigabytes

1.31 128 kbits

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

1.35 (a) 11011101; (b) 110001000

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

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

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

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

1.45

```
#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);
}
```

1.47

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

#include <stdio.h>

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

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

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

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

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

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

1.49

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

1.51

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

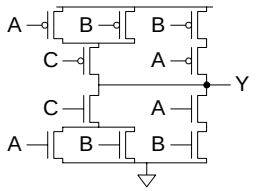
1.53 2^{2^N}

1.55 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.

1.57 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.

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

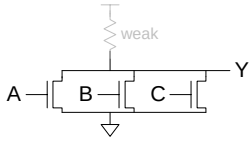
1.61



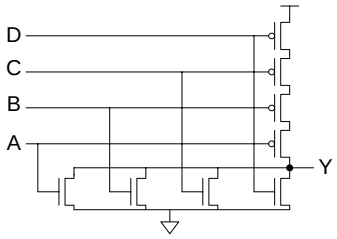
1.63

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

1.65



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

2.1

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

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

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

(d)

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

(e)

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

2.3

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

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

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

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

(e)

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

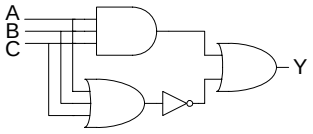
This can also be expressed as:

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

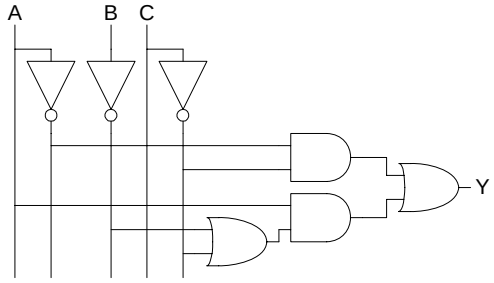
2.5

(a) Same as 2.4(a).

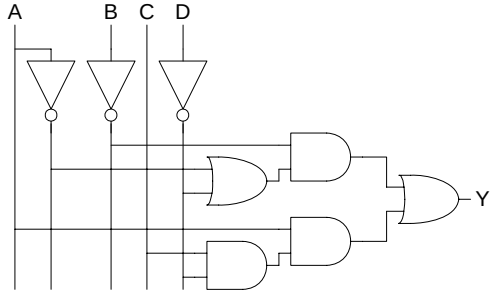
2.4 (b)



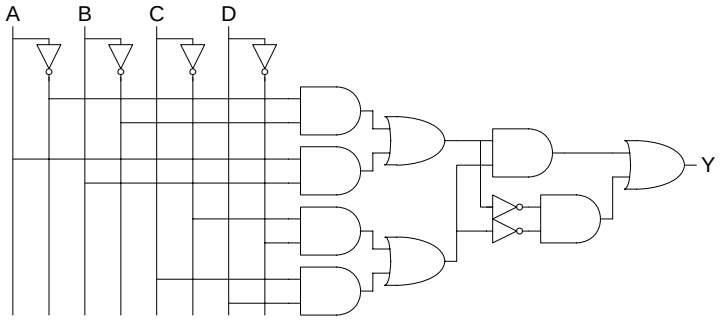
(c)



(d)



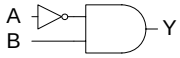
(e)



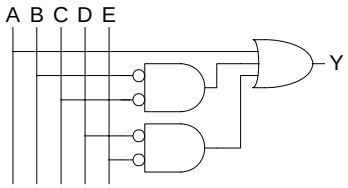
2.7
(a) $Y = AC + \overline{B}C$
(b) $Y = \overline{A}$
(c) $Y = \overline{A} + \overline{B}\,\overline{C} + \overline{B}\,\overline{D} + BD$

2.9
(a) $Y = B + \overline{A}\,\overline{C}$

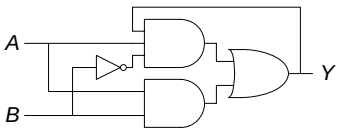
(b) $Y = \overline{A}B$



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



2.11



$Y = A$

2.13

(a)

B	$B \bullet B$
0	0
1	1

(b)

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

(c)

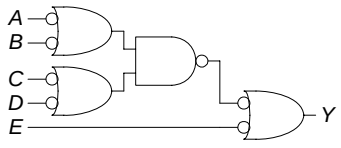
B	C	$(B \bullet C) + (B \bullet \overline{C})$
0	0	0
0	1	0
1	0	1
1	1	1

2.15

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

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

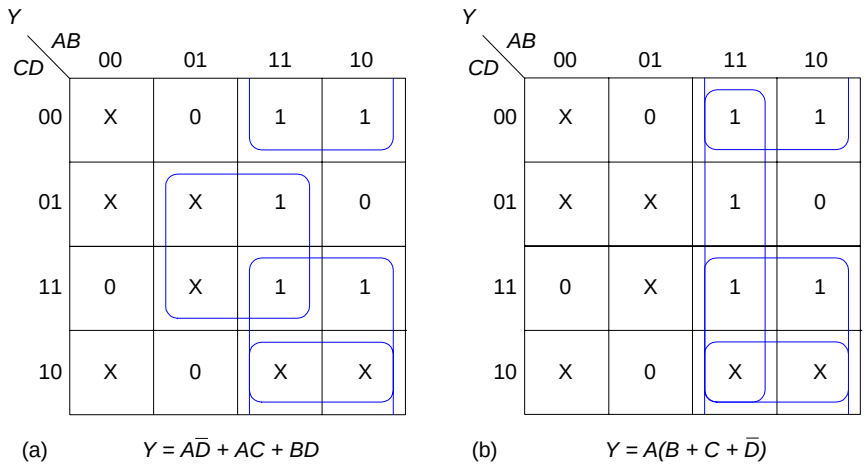
2.17



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

2.19

Two possible options are shown below:

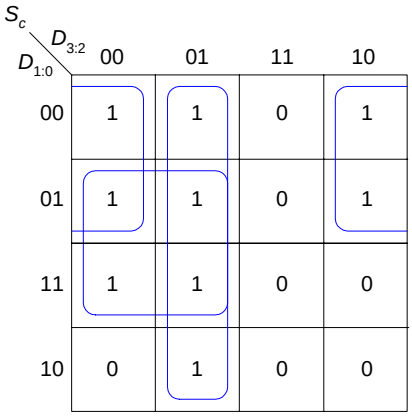


2.21

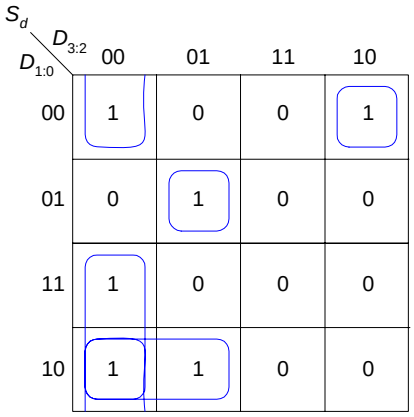
Option (a) could have a glitch when A=1, B=1, C=0, and D transitions from 1 to 0. The glitch could be removed by instead using the circuit in option (b).

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

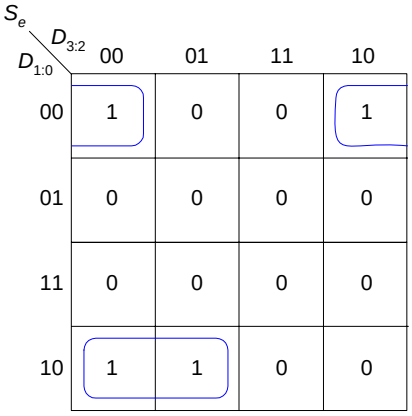
2.23 (a)



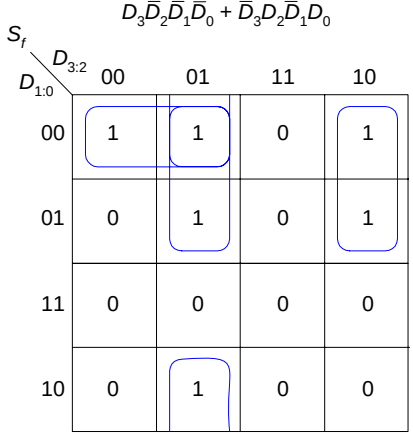
$S_c = \bar{D}_3 D_0 + \bar{D}_3 D_2 + \bar{D}_2 \bar{D}_1$



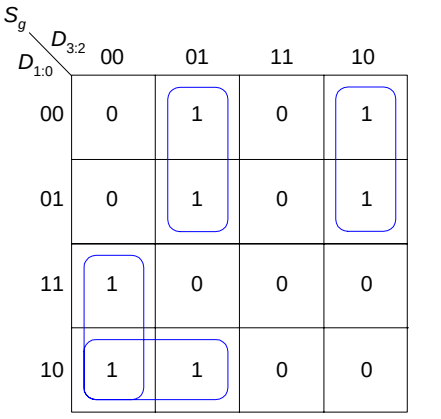
$S_d = \bar{D}_3 D_1 \bar{D}_0 + \bar{D}_3 \bar{D}_2 D_0 + \bar{D}_3 \bar{D}_2 D_1 + D_3 \bar{D}_2 \bar{D}_1 \bar{D}_0 + \bar{D}_3 D_2 \bar{D}_1 D_0$



$S_e = \bar{D}_2 \bar{D}_1 \bar{D}_0 + \bar{D}_3 D_1 \bar{D}_0$

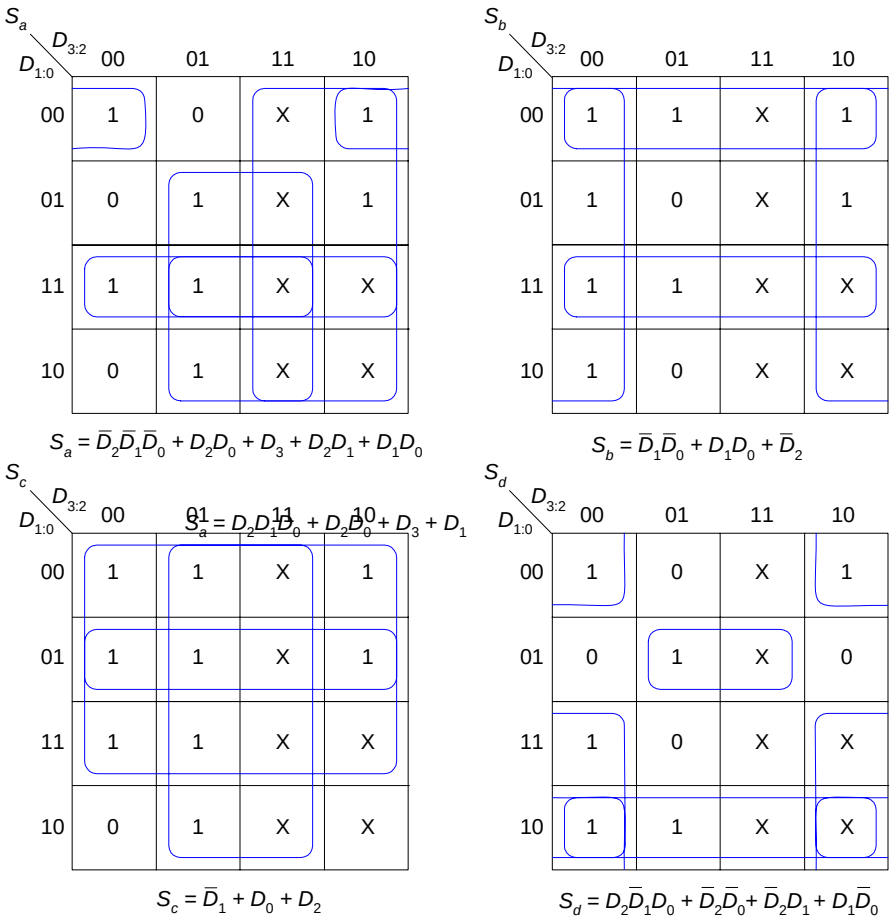


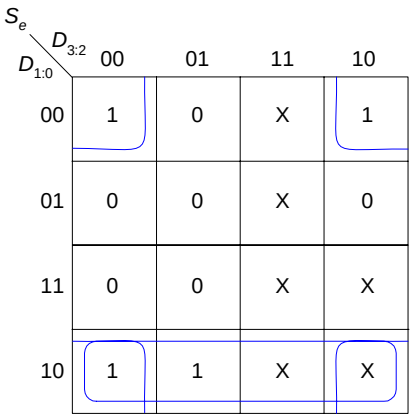
$S_f = \bar{D}_3 \bar{D}_1 \bar{D}_0 + \bar{D}_3 D_2 \bar{D}_1 + \bar{D}_3 D_2 \bar{D}_0 + D_3 \bar{D}_2 \bar{D}_1$



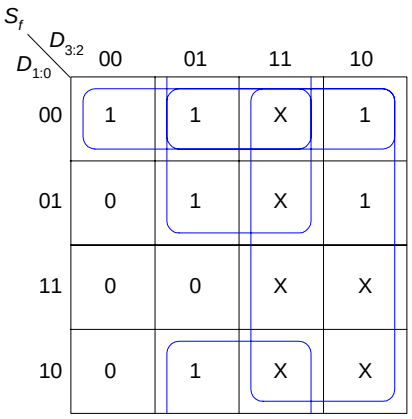
$$S_g = \bar{D}_3\bar{D}_2D_1 + \bar{D}_3D_1\bar{D}_0 + \bar{D}_3D_2\bar{D}_1 + D_3\bar{D}_2\bar{D}_1$$

(b)

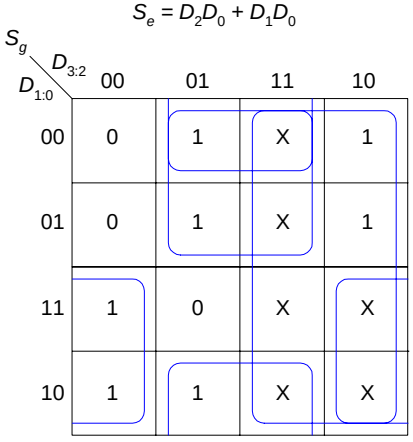

S_a
$D_{3:2}$
$D_{1:0}$
00 01 11 10
00 1 0 X 1
01 0 1 X 1
11 1 1 X X
10 0 1 X X
$S_a = \bar{D}_2\bar{D}_1\bar{D}_0 + D_2D_0 + D_3 + D_2D_1 + D_1D_0$
S_b
$D_{3:2}$
$D_{1:0}$
00 01 11 10
00 1 1 X 1
01 1 0 X 1
11 1 1 X X
10 1 0 X X
$S_b = \bar{D}_1\bar{D}_0 + D_1D_0 + \bar{D}_2$
S_c
$D_{3:2}$
$D_{1:0}$
00 01 11 10
00 1 1 X 1
01 1 1 X 1
11 1 1 X X
10 0 1 X X
$S_c = \bar{D}_1 + D_0 + D_2$
S_d
$D_{3:2}$
$D_{1:0}$
00 01 11 10
00 1 0 X 1
01 0 1 X 0
11 1 0 X X
10 1 1 X X
$S_d = D_2\bar{D}_1D_0 + \bar{D}_2\bar{D}_0 + \bar{D}_2D_1 + D_1\bar{D}_0$



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

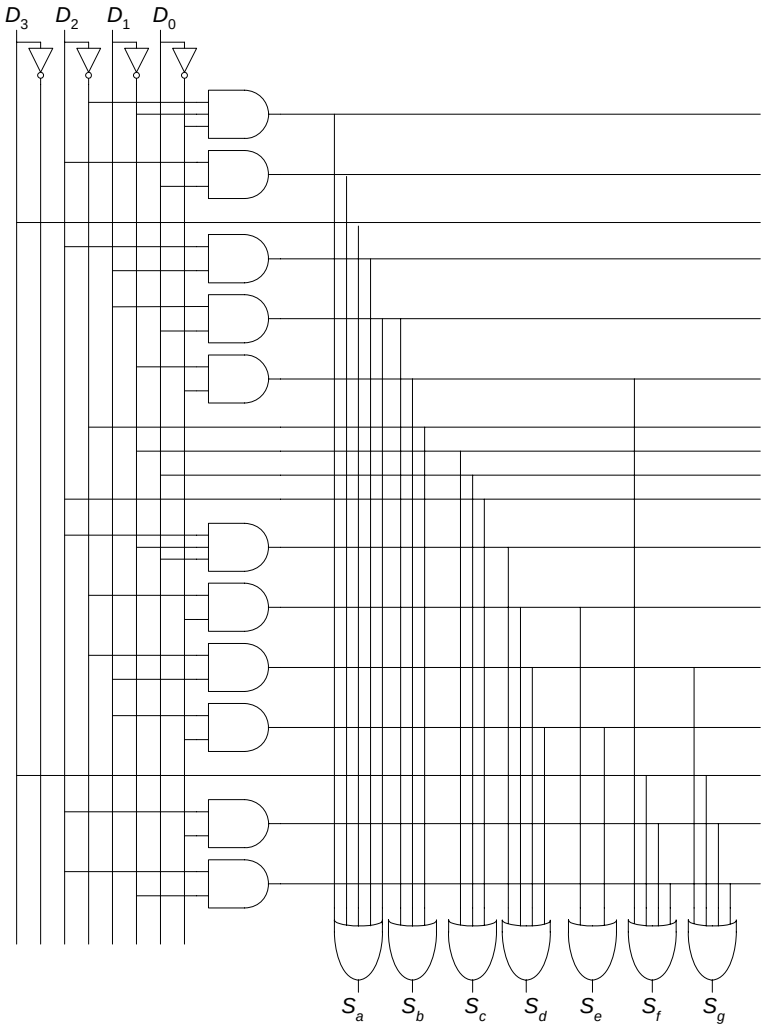


$S_f = \bar{D}_1\bar{D}_0 + D_2\bar{D}_1 + D_2\bar{D}_0 + D_3$



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

(c)



2.25

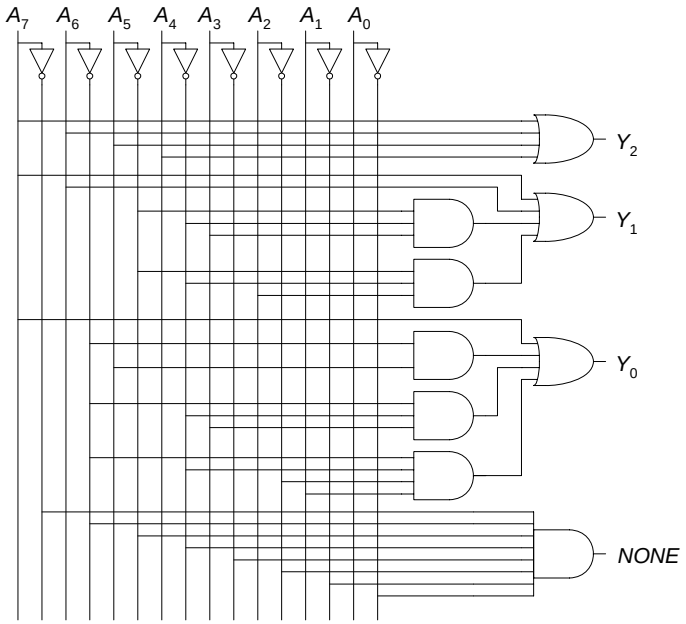
A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Y_2	Y_1	Y_0	*NONE*
0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	1	X	0	0	1	0
0	0	0	0	0	1	X	X	0	1	0	0
0	0	0	0	1	X	X	X	0	1	1	0
0	0	0	1	X	X	X	X	1	0	0	0
0	0	1	X	X	X	X	X	1	0	1	0
0	1	X	X	X	X	X	X	1	1	0	0
1	X	X	X	X	X	X	X	1	1	1	0

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

$$Y_1 = A_7 + A_6 + \overline{A}_5\overline{A}_4A_3 + \overline{A}_5\overline{A}_4A_2$$

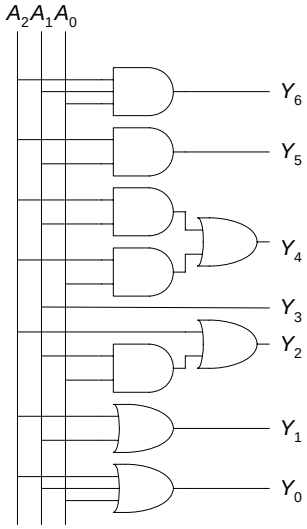
$$Y_0 = A_7 + \overline{A}_6A_5 + \overline{A}_6\overline{A}_4A_3 + \overline{A}_6\overline{A}_4\overline{A}_2A_1$$

$$NONE = \overline{A}_7\overline{A}_6\overline{A}_5\overline{A}_4\overline{A}_3\overline{A}_2\overline{A}_1\overline{A}_0$$



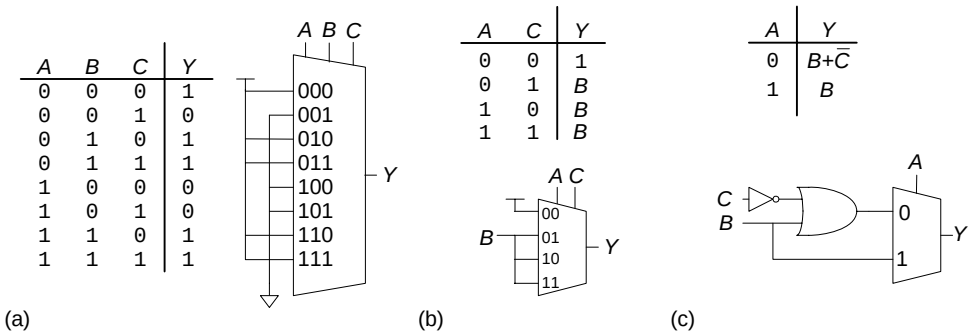
2.27

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



2.29 $Y = \overline{C}\,\overline{D}(A \oplus B) + \overline{A}\,\overline{B} = \overline{A}\,\overline{C}\,\overline{D} + \overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}$

2.31



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

(a)

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

(b)

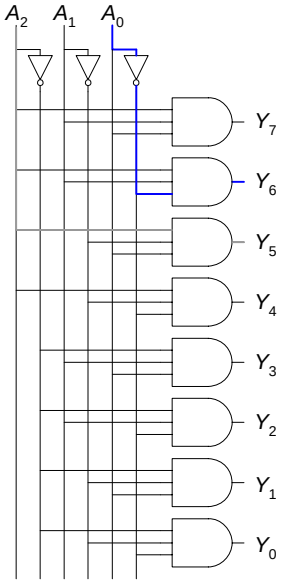
A	Y
0	$B+\overline{C}$
1	B

(c)

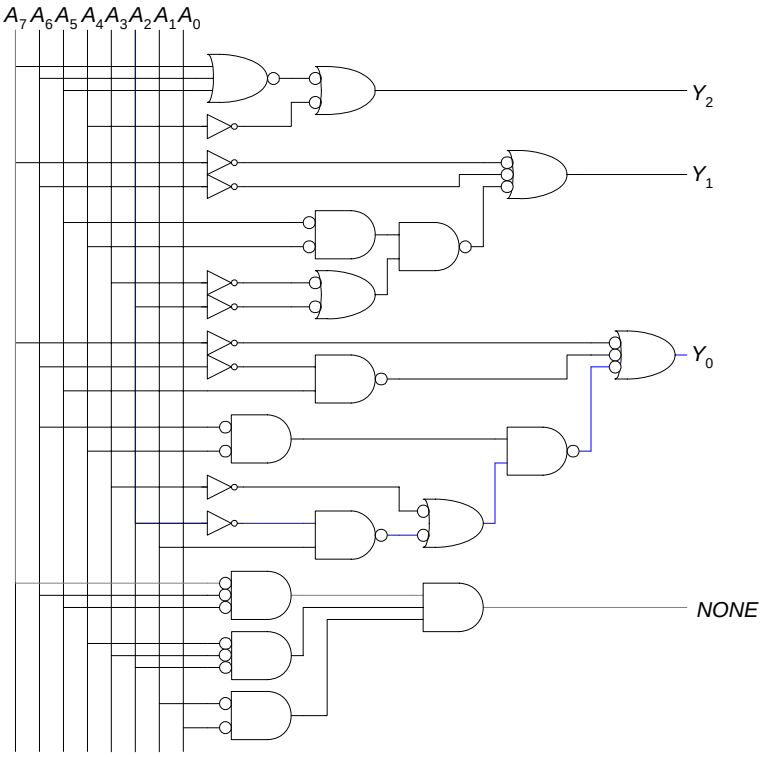
2.33

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

$t_{cd} = t_{cd\_AND3}$
$= \mathbf{30 \text{ ps}}$

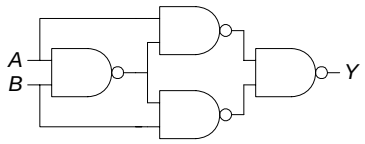


2.35



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

Question 2.1



Question 2.3

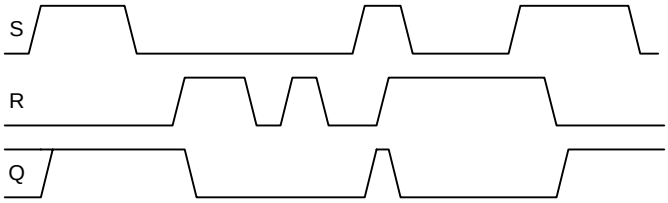
A tristate buffer has two inputs and three possible outputs: 0, 1, and Z. One of the inputs is the data input and the other input is a control input, often called the *enable* input. When the enable input is 1, the tristate buffer transfers the data input to the output; otherwise, the output is high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

Question 2.5

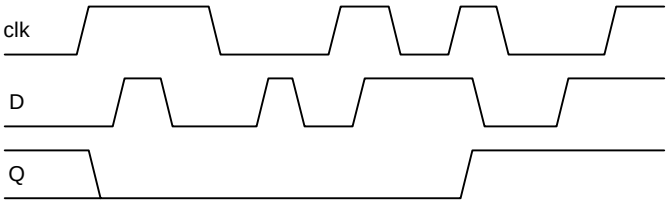
A circuit's contamination delay might be less than its propagation delay because the circuit may operate over a range of temperatures and supply voltages, for example, 3-3.6 V for LVCMOS (low voltage CMOS) chips. As temperature increases and voltage decreases, circuit delay increases. Also, the circuit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

CHAPTER 3

3.1

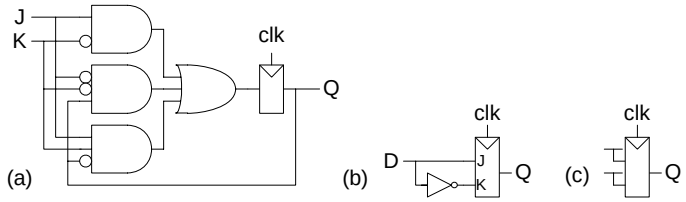


3.3

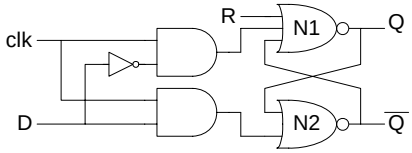


3.5 Sequential logic. This is a D flip-flop with active low asynchronous set and reset inputs. If $\overline{S}$ and $\overline{R}$ are both 1, the circuit behaves as an ordinary D flip-flop. If $\overline{S}$ = 0, Q is immediately set to 0. If $\overline{R}$ = 0, Q is immediately reset to 1. (This circuit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

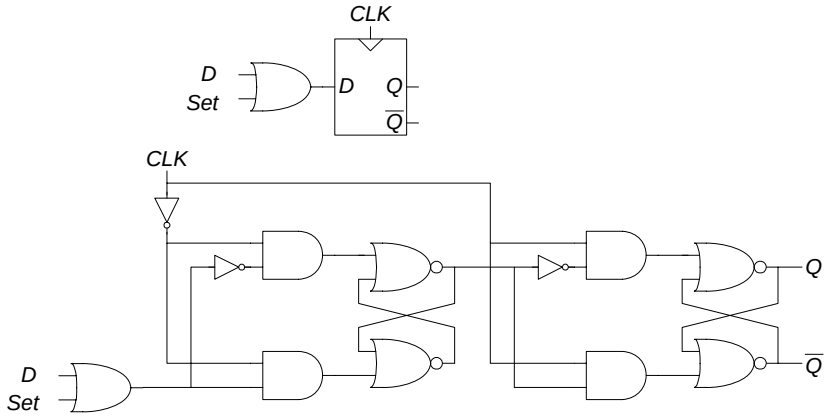
3.7



3.9 Make sure these next ones are correct too.



3.11



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

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

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

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

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

TABLE 3.1 State encoding for Exercise 3.19

.

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

TABLE 3.2 State transition table with binary encodings for Exercise 3.19

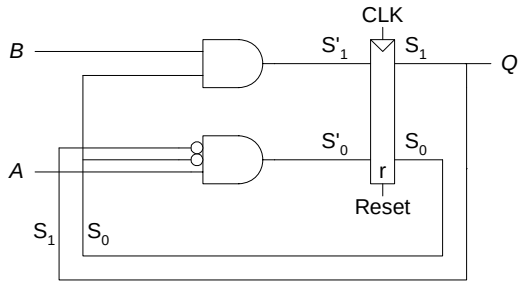
.

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

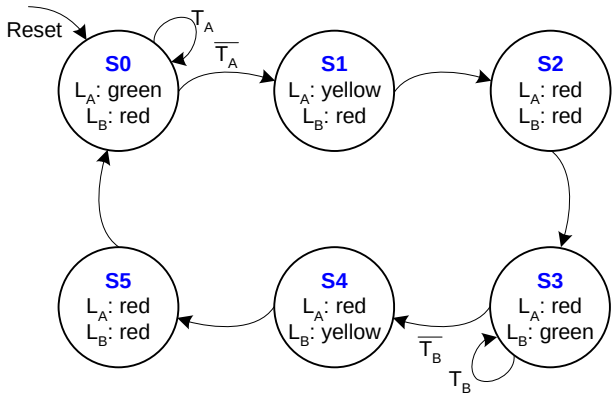
TABLE 3.3 Output table with binary encodings for Exercise 3.19

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

$$Q = S_1$$



3.21



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

TABLE 3.4 State encoding for Exercise 3.21

state	encoding $s_{1:0}$
S2	010
S3	100
S4	101
S5	110

TABLE 3.4 State encoding for Exercise 3.21

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

TABLE 3.5 State transition table with binary encodings for Exercise 3.21

$$S'_2 = S_2 \oplus S_1$$

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

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

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

TABLE 3.6 Output table for Exercise 3.21

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

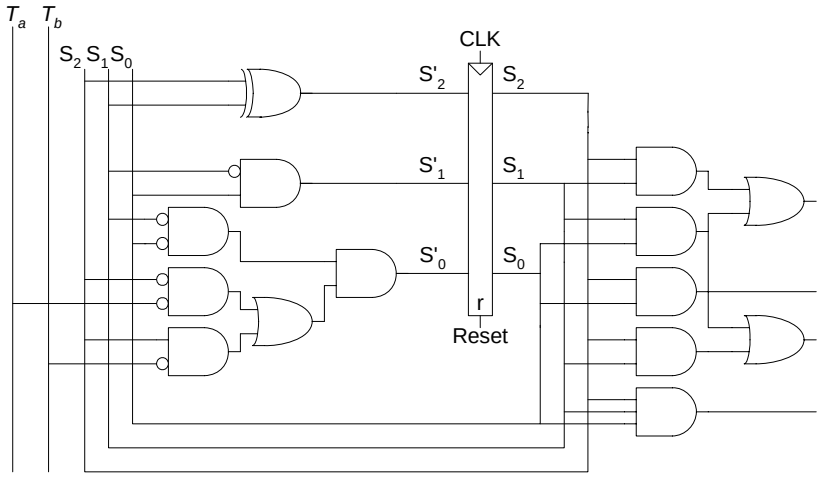
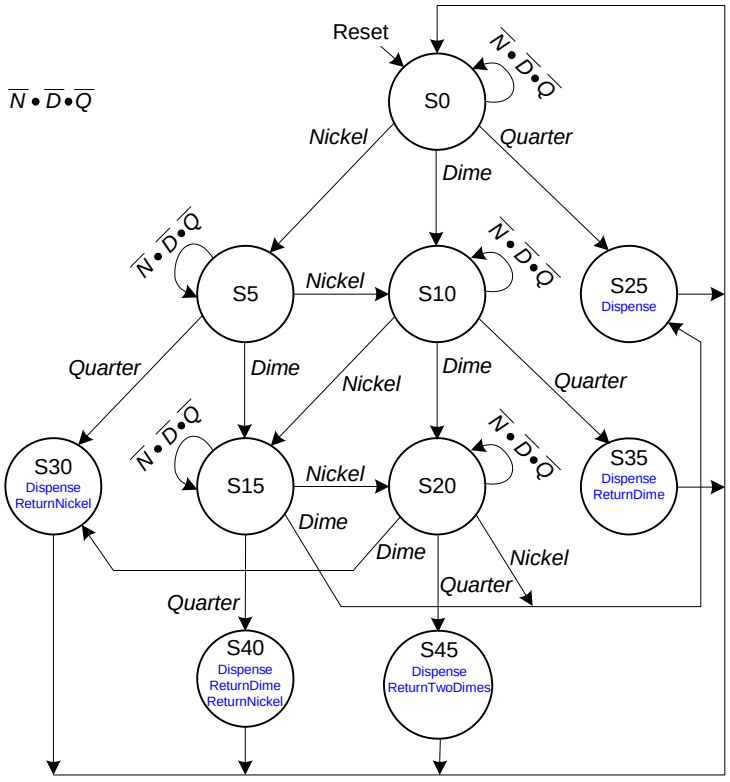


FIGURE 3.1 State machine circuit for traffic light controller for Exercise 3.21

3.23



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

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

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

FIGURE 3.3 State Encodings for Exercise 3.23

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

TABLE 3.7 State transition table for Exercise 3.23

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

TABLE 3.7 State transition table for Exercise 3.23

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

TABLE 3.8 State transition table for Exercise 3.23

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

TABLE 3.8 State transition table for Exercise 3.23

$$S'_9 = S_6 Q$$

$$S'_8 = S_5 Q$$

$$S'_7 = S_2 Q$$

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

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

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

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

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

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

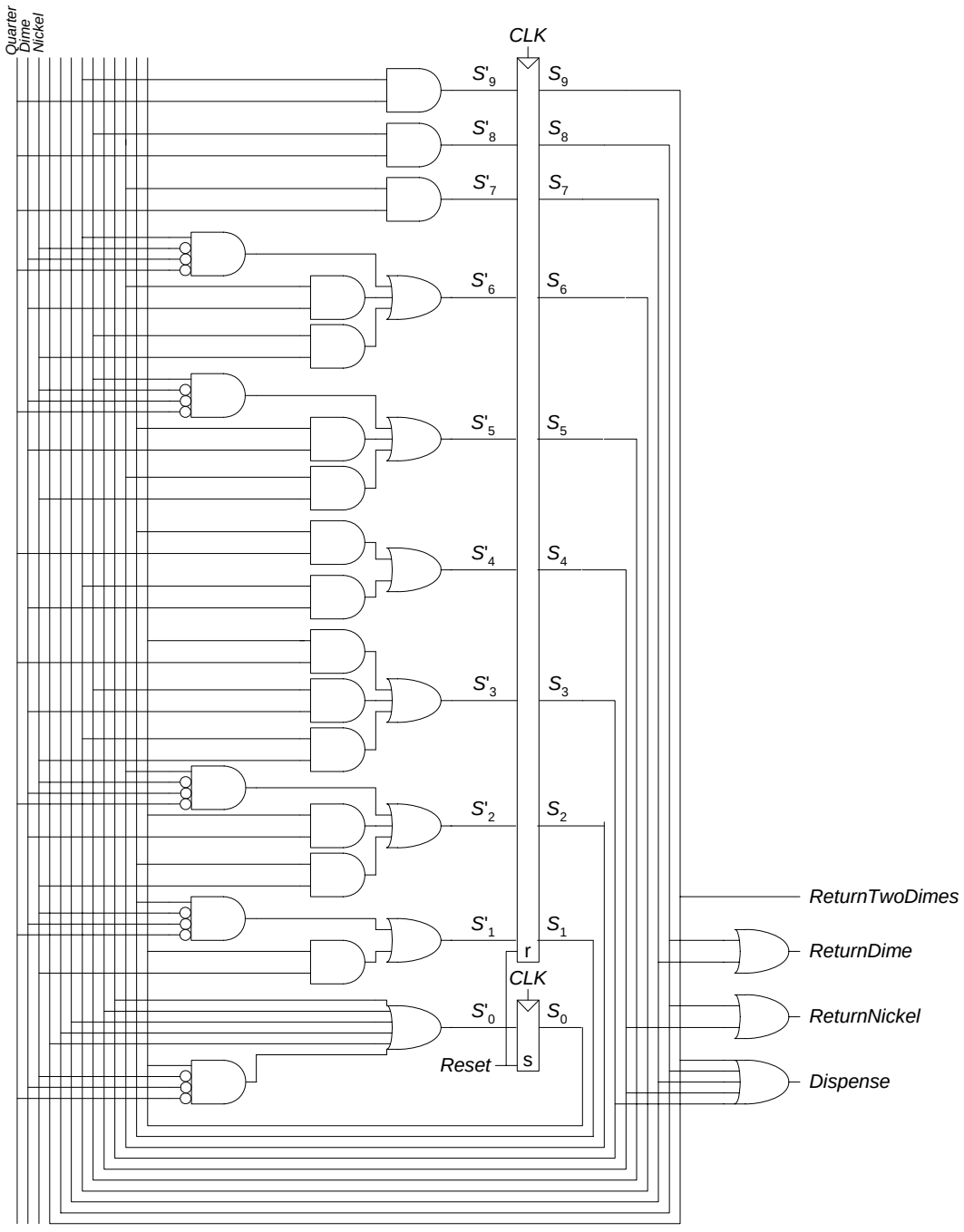
$$S'_0 = S_0 \overline{N}\,\overline{D}\,\overline{Q} + S_3 + S_4 + S_7 + S_8 + S_9$$

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

$$ReturnNickel = S_4 + S_8$$

$$ReturnDime = S_7 + S_8$$

$$ReturnTwoDimes = S_9$$


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

3.25

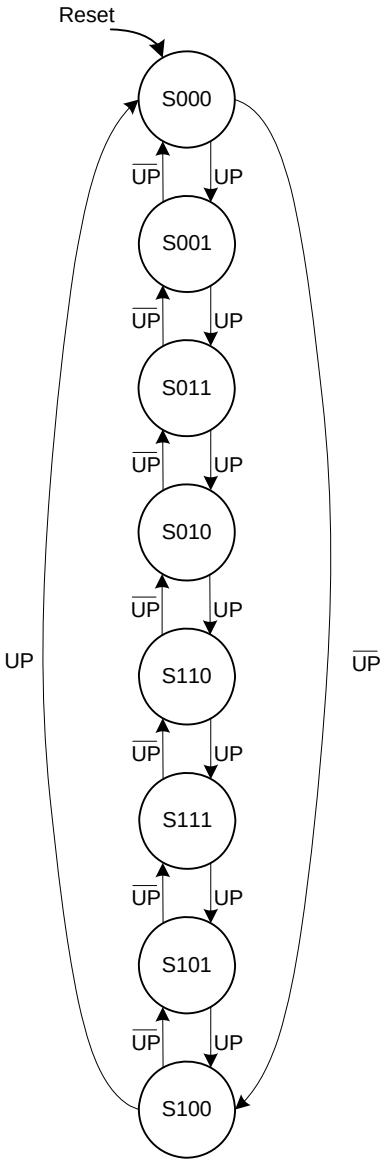


FIGURE 3.4 State transition diagram for Exercise 3.25

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

TABLE 3.9 State transition table for Exercise 3.25

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

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

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

$$Q_2 = S_2$$

$$Q_1 = S_1$$

$$Q_0 = S_0$$

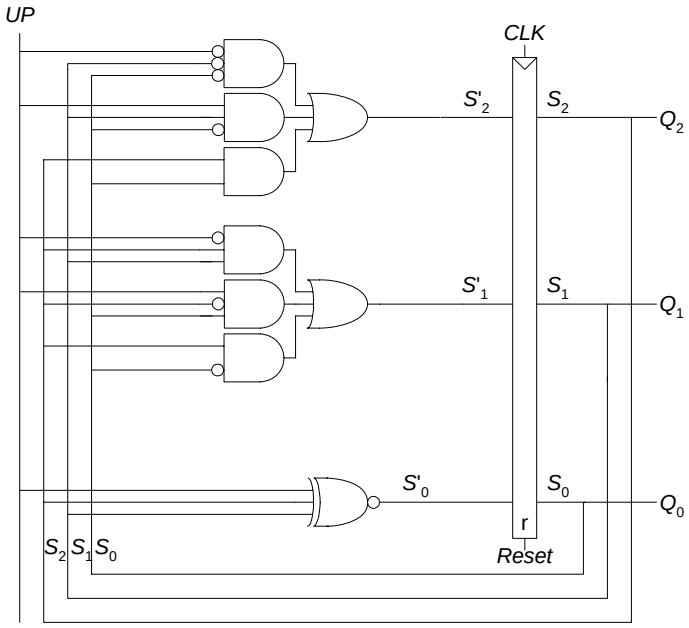


FIGURE 3.5 Finite state machine hardware for Exercise 3.25

3.27

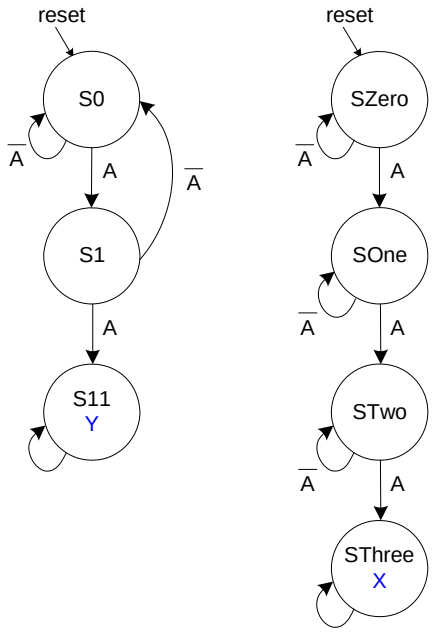


FIGURE 3.6 Factored state transition diagram for Exercise 3.27

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

TABLE 3.10 State transition table for output *Y* for Exercise 3.27

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

TABLE 3.11 State transition table for output *X* for Exercise 3.27

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

$$T_1 = T_1 + T_0A$$
$$T_0 = A(T_1 + \overline{T_0}) + \bar{A}T_0 + T_1T_0$$
$$X = T_1T_0$$

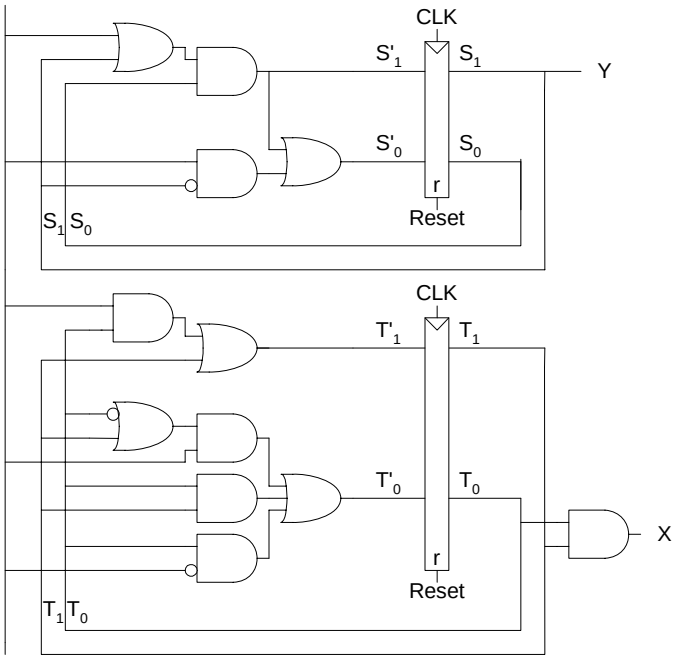


FIGURE 3.7 Finite state machine hardware for Exercise 3.27

3.29

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

TABLE 3.12 State transition table with binary encodings for Exercise 3.29

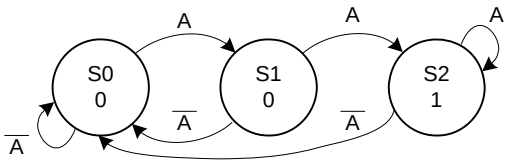


FIGURE 3.8 State transition diagram for Exercise 3.29

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

3.31
(a) 9.09 GHz
(b) 15 ps
(c) 26 ps

3.33 1.138 ns

3.35
You know you've already entered metastability, so the probability that the sampled signal is metastable is 1. Thus,

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

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

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

Thus,

$$t = -\tau \times \ln(P(failure)) = -6 \times \ln((0.01)) = 27.6 \text{ seconds}$$

(b) The probability of death is the chance of still being metastable after 5 minutes

P(failure) = 1 × e -300seconds / 6seconds = e -50 = 1.9 × 10-22

$$P(\text{failure}) = e^{-\frac{t}{\tau}} = e^{-\frac{300}{6}} = e^{-50} = 1.9 \times 10^{-22}$$

3.37 Alyssa is correct. Ben's circuit does not eliminate metastability. After the first transition on D, D2 is always 0 because as D2 transitions from 0 to 1 or 1 to 0, it enters the forbidden region and Ben's "metastability detector" resets the first flip-flop to 0. Even if Ben's circuit could correctly detect a metastable

output, it would asynchronously reset the flip-flop which, if the reset occurred around the clock edge, this could cause the second flip-flop to sample a transitioning signal and become metastable.

Question 3.1

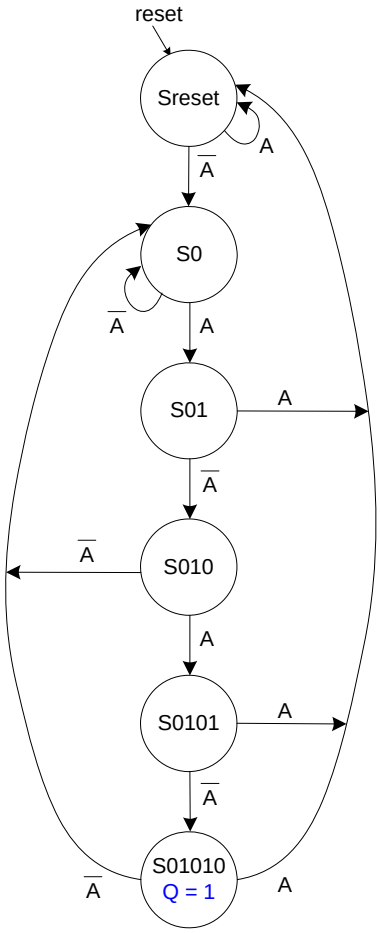


FIGURE 3.9 State transition diagram for Question 3.1

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

TABLE 3.13 State transition table for Question 3.1

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

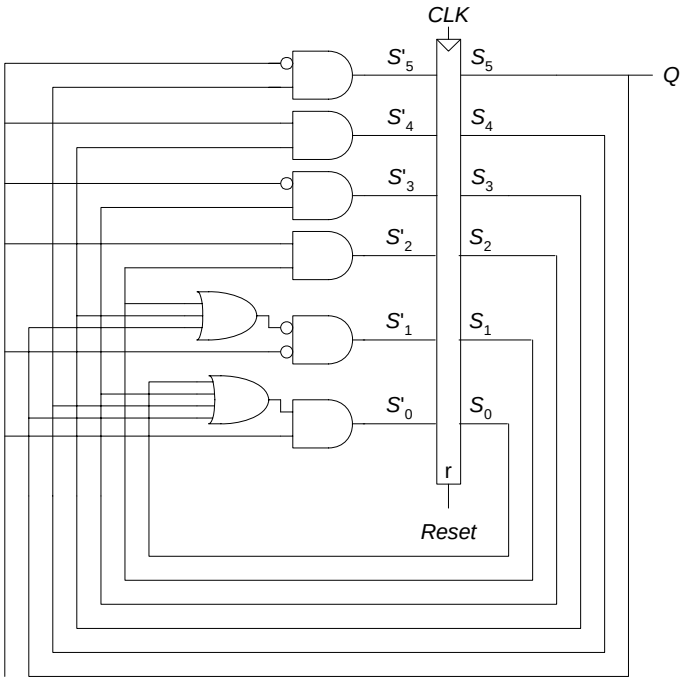


FIGURE 3.10 Finite state machine hardware for Question 3.1

Question 3.3

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

Question 3.5

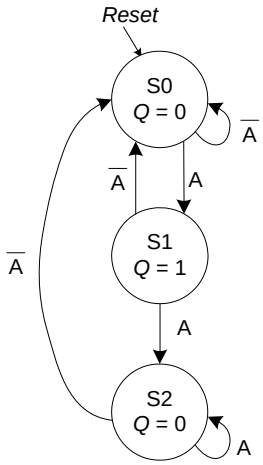


FIGURE 3.11 State transition diagram for edge detector circuit of Question 3.5

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

TABLE 3.14 State transition table for Question 3.5

$$S'_1 = AS_1$$

$$S'_0 = AS_1S_0$$

$$Q = S_1$$

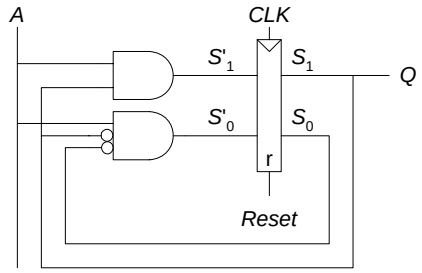


FIGURE 3.12 Finite state machine hardware for Question 3.5

Question 3.7

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

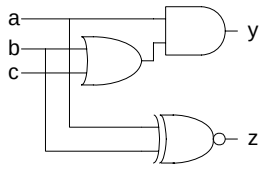
Question 3.9

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 $t_{\text{cd\_BUF}}$ after the actual clock edge. Thus, the propagation delay through the logic is now given an extra t_{cd_BUF}. So, t_{pd} now must be less than T_c + t_{cd_BUF} -(t_{pcq} + t_{setup}).

CHAPTER 4

Note: the HDL files given in the following solutions are available on the textbook's companion website at: http://textbooks.elsevier.com/9780123704979 .

4.1



4.3

Verilog

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

4.5

Verilog

```
module minority(input  a, b, c
                output    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;
```

4.7

ex4_7.tv file:

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

Option 1:

Verilog

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

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

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

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

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

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

VHDL

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

entity ex4_7_testbench is -- no inputs or outputs
end;

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

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

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

(*VHDL continued on next page*)

(*continued from previous page*)

VHDL

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

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

      data <= testvectors(vectornum)(10 downto 7)
        after 1 ns;
     s_expected <= testvectors(vectornum)(6 downto 0)
        after 1 ns;
    end if;
  end process;

  -- check results on falling edge of clk
  process (clk) begin
    if (clk'event and clk = '0' and reset = '0') then
      assert s = s_expected
        report "data = " &
          integer'image(CONV_INTEGER(data)) &
          "; s = " &
          integer'image(CONV_INTEGER(s)) &
          "; s_expected = " &
           integer'image(CONV_INTEGER(s_expected));
      if (s /= s_expected) then
        errors := errors + 1;
      end if;
      vectornum := vectornum + 1;
      if (is_x(testvectors(vectornum))) then
        if (errors = 0) then
          report "Just kidding -- " &
                 integer'image(vectornum) &
                 " tests completed successfully."
                 severity failure;
        else
          report integer'image(vectornum) &
                 " tests completed, errors = " &
                 integer'image(errors)
                 severity failure;
        end if;
      end if;
    end if;
  end process;
end;
```

Option 2 (VHDL only):

VHDL

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

entity ex4_7_testbench is -- no inputs or outputs
end;

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

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

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

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

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

      data <= testvectors(vectornum)(10 downto 7)
        after 1 ns;
     s_expected <= testvectors(vectornum)(6 downto 0)
        after 1 ns;
    end if;
  end process;

  -- check results on falling edge of clk
  process (clk) begin
    if (clk'event and clk = '0' and reset = '0') then
      assert s = s_expected
        report "data = " & str(data) &
          "; s = " & str(s) &
           "; s_expected = " & str(s_expected);
      if (s /= s_expected) then
        errors := errors + 1;
      end if;
      vectornum := vectornum + 1;
      if (is_x(testvectors(vectornum))) then
        if (errors = 0) then
          report "Just kidding -- " &
                 integer'image(vectornum) &
                 " tests completed successfully."
                 severity failure;
        else
          report integer'image(vectornum) &
                 " tests completed, errors = " &
                 integer'image(errors)
                 severity failure;
        end if;
      end if;
    end if;
  end process;
end;
```

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

4.9

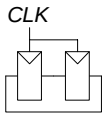
Verilog

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

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

VHDL

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

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

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

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

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



4.13

Verilog

```
module decoder2_4(input      [1:0] a,
                  output reg [3:0] y);

  always @(*)
    case (a)
      2'b00: y = 4'b0001;
      2'b01: y = 4'b0010;
      2'b10: y = 4'b0100;
      2'b11: y = 4'b1000;
    endcase
endmodule
```

VHDL

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

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

architecture synth of decoder2_4 is
begin
  process(a) 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;
```

4.15

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

Verilog

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

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

VHDL

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

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

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

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

Verilog

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

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

VHDL

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

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

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

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

Verilog

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

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

VHDL

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

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

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

4.17

Verilog

```
module ex4_17(input      a, b, c, d,
              output reg y);

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

VHDL

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

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

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

4.19

Verilog

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

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

VHDL

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

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

architecture synth of priority_encoder is
begin
  process(a) begin
    if (a(7) = '1') then
      y <= "111"; none <= '0';
    elsif (a(6) = '1') then
      y <= "110"; none <= '0';
    elsif (a(5) = '1') then
      y <= "101"; none <= '0';
    elsif (a(4) = '1') then
      y <= "100"; none <= '0';
    elsif (a(3) = '1') then
      y <= "011"; none <= '0';
    elsif (a(2) = '1') then
      y <= "010"; none <= '0';
    elsif (a(1) = '1') then
      y <= "001"; none <= '0';
    elsif (a(0) = '1') then
      y <= "000"; none <= '0';
    else
      y <= "000"; none <= '1';
    end if;
  end process;
end;
```

4.21

Verilog

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

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

4.23

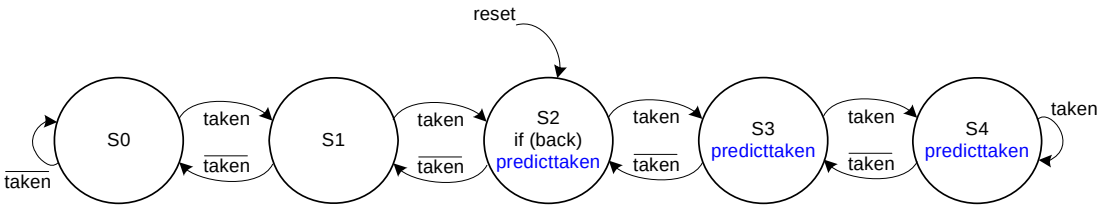


FIGURE 4.1 State transition diagram for Exercise 4.23

4.25

Verilog

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

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

VHDL

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

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

architecture synth of jkflop is
signal jk: STD_LOGIC_VECTOR(1 downto 0);
begin
  jk <= j & k;
  process(clk) begin
    if clk'event and clk = '1' 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;
```

4.27

Verilog

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

  reg  [1:0] state, nextstate;

  parameter S0 = 2'b00;
  parameter S1 = 2'b01;
  parameter S2 = 2'b10;
  parameter S3 = 2'b11;

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

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

  // Next State Logic
  always @(*)
    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 @ (*)
    case (state)
      S0: {la, lb} = {green, red};
      S1: {la, lb} = {yellow, red};
      S2: {la, lb} = {red, green};
      S3: {la, lb} = {red, yellow};
    endcase
endmodule
```

VHDL

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

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

architecture behave of trafficFSM is
  type statetype is (S0, S1, S2, S3);
  signal state, nextstate: statetype;
  signal lalb: STD_LOGIC_VECTOR(3 downto 0);
begin
  -- state register
  process(clk, reset) begin
    if reset = '1' then state <= S0;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

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

  -- output logic
  la <= lalb(3 downto 2);
  lb <= lalb(1 downto 0);
  process (state) 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;
```

4.29

Verilog

```
module fig3_40(input      clk, a, b, c, d,
               output reg x, y);

  wire n1, n2;
  reg  areg, breg, creg, dreg;

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

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

VHDL

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

entity fig3_40 is
  port(clk, a, b, c, d: in  STD_LOGIC;
       x, y:            out STD_LOGIC);
end;

architecture synth of fig3_40 is
  signal n1, n2, areg, breg, creg, dreg: STD_LOGIC;
begin
  process(clk) begin
    if clk'event and clk = '1' 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;
```

4.31

Verilog

```
module fig3_66(input clk, reset, a, b,
               output reg q);

  reg [1:0] state, nextstate;

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

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

  // Next State Logic
  always @ (*)
    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 @ (*)
    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_66 is
  port(clk, reset, a, b: in  STD_LOGIC;
       q:                out STD_LOGIC);
end;

architecture synth of fig3_66 is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset = '1' then state <= S0;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

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

4.33

Verilog

```
module daughterfsm(input  clk, reset, a,
                   output smile);

  reg [2:0] state, nextstate;

  parameter S0 = 3'b000;
  parameter S1 = 3'b001;
  parameter S2 = 3'b010;
  parameter S3 = 3'b011;
  parameter S4 = 3'b100;

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

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

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

VHDL

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

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

architecture synth of daughterfsm is
  type statetype is (S0, S1, S2, S3, S4);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset = '1' then state <= S0;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

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

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

4.35

Verilog

```
module ex4_35(input clk, reset,
              output [2:0] q);

  reg [2:0] state, nextstate;

  parameter S0 = 3'b000;
  parameter S1 = 3'b001;
  parameter S2 = 3'b011;
  parameter S3 = 3'b010;
  parameter S4 = 3'b110;
  parameter S5 = 3'b111;
  parameter S6 = 3'b101;
  parameter S7 = 3'b100;

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

  // Next State Logic
  always @ (*)
    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_35 is
  port(clk:   in  STD_LOGIC;
       reset: in  STD_LOGIC;
       q:     out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture synth of ex4_35 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 = '1' then state <= "000";
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

  -- next state logic
  process (state) 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;
```

4.37 **Option 1**

Verilog

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

  reg [1:0] state, nextstate;

  parameter S0 = 2'b00;
  parameter S1 = 2'b01;
  parameter S2 = 2'b10;
  parameter S3 = 2'b11;

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

  // Next State Logic
  always @ (*)
    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 @ (*)
    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_37 is
  port(clk:   in  STD_LOGIC;
       reset: in  STD_LOGIC;
       a, b:  in  STD_LOGIC;
       z:     out STD_LOGIC);
end;

architecture synth of ex4_37 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 = '1' then state <= S0;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

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

(continued from previous page)

VHDL

```
  -- output logic
  process (state, a, b) 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;
```

4.37 Option 2

Verilog

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

  reg aprev;

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

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

VHDL

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

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

architecture synth of ex4_37 is
  signal aprev, n1and, n2or: STD_LOGIC;
begin
  -- state register
  process(clk) begin
    if clk'event and clk = '1' then
      aprev <= a;
    end if;
  end process;


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

4.39

Verilog

```
module ex4_39(input  clk, start, a,
              output q);

  reg  [1:0] state, nextstate;

  parameter S0 = 2'b00;
  parameter S1 = 2'b01;
  parameter S2 = 2'b10;
  parameter S3 = 2'b11;

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

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

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

VHDL

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

entity ex4_39 is
  port(clk, start, a: in  STD_LOGIC;
       q:             out STD_LOGIC);
end;

architecture synth of ex4_39 is
  type statetype is (S0, S1, S2, S3);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, start) begin
    if start = '1' then state <= S0;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

 -- next state logic
  process (state, a) begin
    case state is
      when S0 =>  if a = '1' then
                       nextstate <= S1;
                  else nextstate <= S0;
                  end if;
      when S1 =>  if a = '1' then
                       nextstate <= S2;
                  else nextstate <= S3;
                  end if;
      when S2 =>  if a = '1' then
                       nextstate <= S2;
                  else nextstate <= S3;
                  end if;
      when S3 =>  if a = '1' then
                       nextstate <= S2;
                  else nextstate <= S3;
                  end if;
      when others =>   nextstate <= S0;
    end case;
  end process;

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

4.41

Verilog

```
module ex4_41(input  clk, reset, a,
              output q);

  reg [1:0] state, nextstate;

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

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

  // Next State Logic
  always @(*)
    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_41 is
  port(clk, reset, a: in  STD_LOGIC;
       q:             out STD_LOGIC);
end;

architecture synth of ex4_41 is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset = '1' then state <= S0;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

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

4.43

Verilog

```
module ex4_43(input clk, c, input [1:0] a, b,
              output reg [1:0] s);

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

  always @(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_43 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_43 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 clk'event and clk = '1' 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;
```

4.45

Verilog

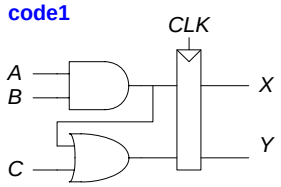
```
module syncbad(input      clk,
               input      d,
               output reg q);

  reg n1;

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

VHDL

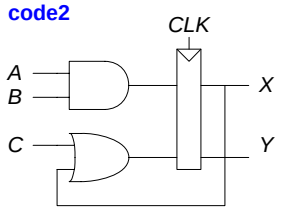
```
library IEEE; use IEEE.STD_LOGIC_1164.all;

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 clk'event and clk = '1' then
      q  <= n1; -- nonblocking
      n1 <= d;  -- nonblocking
    end if;
  end process;
end;
```

4.47 They do not have the same function.

code1



code2



4.49

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

Verilog

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

VHDL

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

Question 4.3

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

CHAPTER 5

Note: the HDL files given in the following solutions are available on the textbook's companion website at: http://textbooks.elsevier.com/9780123704979 .

5.1

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

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

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

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

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

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

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

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

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

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.

5.5

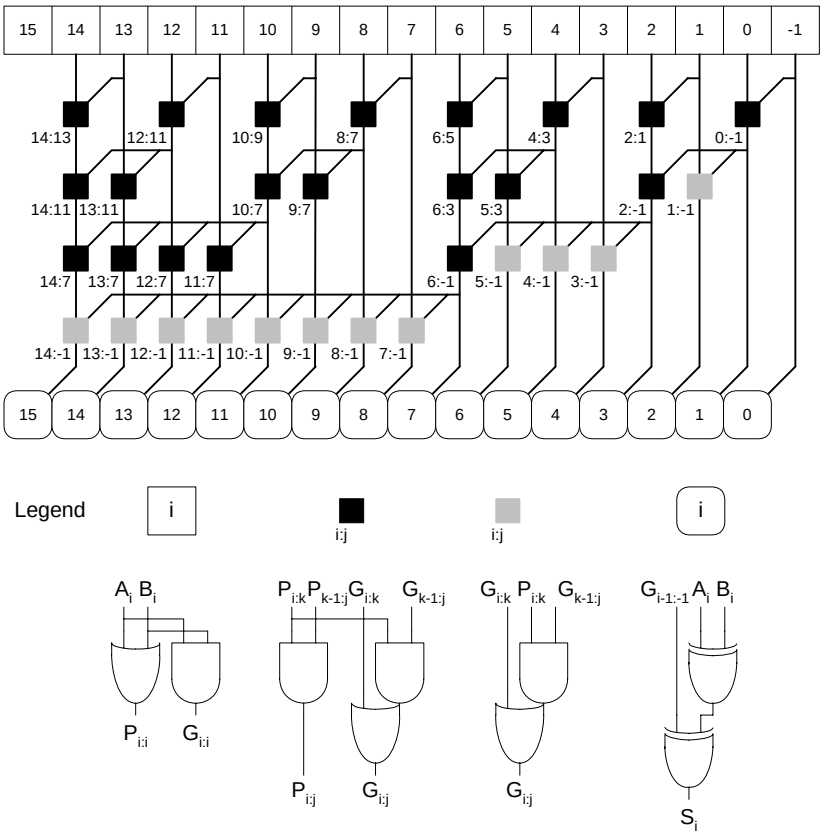


FIGURE 5.1 16-bit prefix adder with “gray cells”

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

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

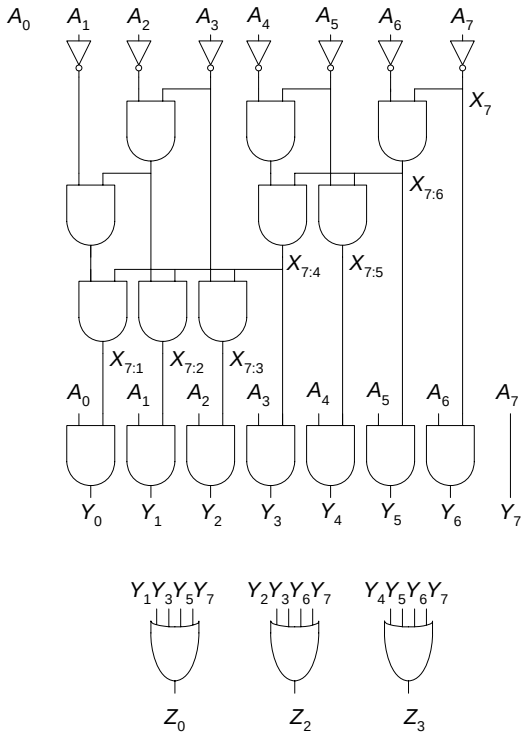


FIGURE 5.2 8-input priority encoder

Verilog

```
module priorityckt(input  [7:0] a,
                   output [2:0] z);


  wire [7:0] y;
  wire       x7, x76, x75, x74, x73, x72, x71;
  wire       x32, x54, x31;
  wire [7:0] abar;

  // row of inverters
  assign abar = ~a;

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

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

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

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

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

VHDL

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

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

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

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

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

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

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

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

end;
```

5.9

Verilog

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

  wire [31:0] S, Bout;

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

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

endmodule
```

VHDL

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

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

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

  process (F(1 downto 0), A, Bout, S) 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;
```

5.11

Verilog

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

  wire [31:0] S, Bout;

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

  always @ (*)
    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 @ (*)
    case (F[2:1])
      2'b01: Overflow <= A[31] & B[31] & ~S[31] |
                         ~A[31] & ~B[31] & S[31];
      2'b11: Overflow <= ~A[31] & B[31] & S[31] |
                          A[31] & ~B[31] & ~S[31];
      default: Overflow <= 1'b0;
    endcase

endmodule
```

VHDL

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

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

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

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

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

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

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.

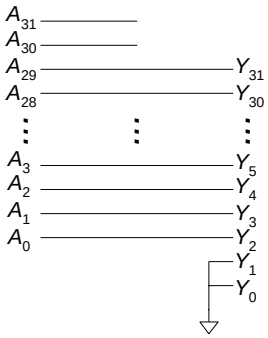


FIGURE 5.3 2-bit left shifter, 32-bit input and output

2-bit Left Shifter

Verilog

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

endmodule
```

VHDL

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

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

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

5.15

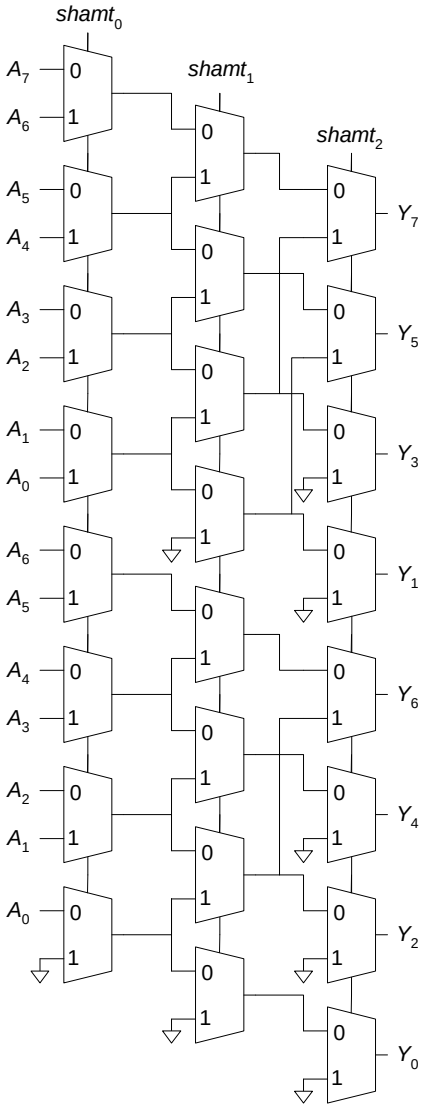


FIGURE 5.4 8-bit left shifter using 24 2:1 multiplexers

5.17

(a) $B = 0$, $C = A$, $k = shamt$

(b) $B = A_{N-1}$ (the most significant bit of A), repeated N times to fill all N bits of B

(c) $B = A$, $C = 0$, $k = N$ - *shamt*
(d) $B = A$, $C = A$, $k = shamt$
(e) $B = A$, $C = A$, $k = N$ - *shamt*

5.19

Recall that a two's complement number has the same weights for the least significant N-1 bits, regardless of the sign. The sign bit has a weight of -2^{N-1}. Thus, the product of two N-bit complement numbers, y and x is:

P = -y_{N-1}2N-1 +

$$P = \left(-y_{N-1}2^{N-1} + \sum_{j=0}^{N-2} y_j 2^j\right)\left(-x_{N-1}2^{N-1} + \sum_{i=0}^{N-2} x_i 2^i\right)$$

Thus,

$$\sum_{i=0}^{N-2}\sum_{j=0}^{N-2} x_i y_j 2^{i+j} + x_{N-1}y_{N-1}2^{2N-2} - \sum_{i=0}^{N-2} x_i y_{N-1} 2^{i+N-1} - \sum_{j=0}^{N-2} x_{N-1} y_j 2^{j+N-1}$$

The two negative partial products are formed by taking the two's complement (inverting the bits and adding 1). Figure 5.5 shows a 4 x 4 multiplier. Figure 5.5 (b) shows the partial products using the above equation. Figure 5.5 (c) shows a simplified version, pushing through the 1's. This is known as a *modified Baugh-Wooley multiplier*. It can be built using a hierarchy of adders.

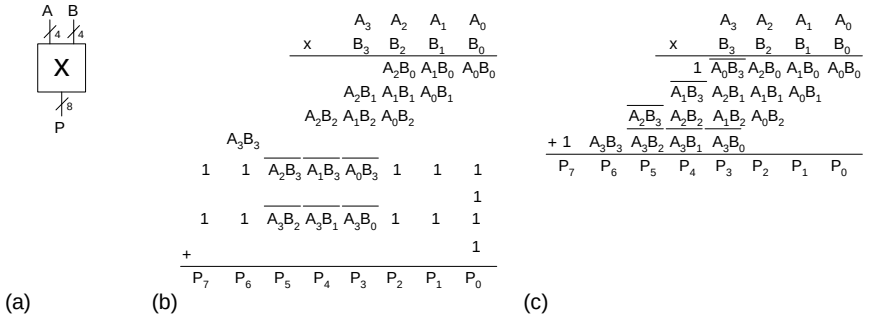


FIGURE 5.5 Multiplier: (a) symbol, (b) function, (c) simplified function

5.21

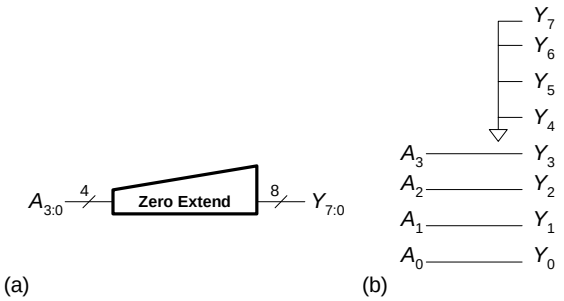


FIGURE 5.6 Zero extension unit (a) symbol, (b) underlying hardware

Verilog

```
module zeroext4_8(input [3:0] a, output [7:0] y);

  assign y = {4'b0, a};

endmodule
```

VHDL

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

entity zeroext4_8 is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of zeroext4_8 is
begin
  y <= "0000" & a(3 downto 0);
end;
```

5.23

(a) $\left[0, \left(2^{12} - 1 + \frac{2^{12} - 1}{2^{12}}\right)\right]$

(b) $\left[-\left(2^{11} - 1 + \frac{2^{12} - 1}{2^{12}}\right), \left(2^{11} - 1 + \frac{2^{12} - 1}{2^{12}}\right)\right]$

(c) $\left[-\left(2^{11} + \frac{2^{12} - 1}{2^{12}}\right), \left(2^{11} - 1 + \frac{2^{12} - 1}{2^{12}}\right)\right]$

5.25
(a) 1111 0010 . 0111 0000 = 0xF270
(b) 0010 1010 . 0101 0000 = 0x2A50
(c) 1110 1110 . 1101 1000 = 0xEED8

5.27
(a) 5.5
(b) $-0000.0001_2 = -0.0625$
(c) -8

5.29
(a)
0xC0D20004 = 1 1000 0001 101 0010 0000 0000 0000 0100
= - 1.101 0010 0000 0000 0000 01 $\times 2^2$
0x72407020 = 0 1110 0100 100 0000 0111 0000 0010 0000
= 1.100 0000 0111 0000 001 $\times 2^{101}$

When adding these two numbers together, 0xC0D20004 becomes:
0×2^{101} because all of the significant bits shift off the right when making the exponents equal. Thus, the result of the addition is simply the second number:
0x72407020

(b)
0xC0D20004 = 1 1000 0001 101 0010 0000 0000 0000 0100
= - 1.101 0010 0000 0000 0000 01 $\times 2^2$
0x40DC0004 = 0 1000 0001 101 1100 0000 0000 0000 0100
= 1.101 1100 0000 0000 0000 01 $\times 2^2$

1.101 1100 0000 0000 0000 01 $\times 2^2$
- 1.101 0010 0000 0000 0000 01 $\times 2^2$
= 0.000 1010 $\times 2^2$
= 1.010×2^{-2}

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

(c)
0x5FBE4000 = 0 1011 1111 011 1110 0100 0000 0000 0000 0000
= 1.011 1110 01 $\times 2^{64}$
0x3FF80000 = 0 0111 1111 111 1000 0000 0000 0000 0000

$= 1.111\ 1 \times 2^0$

0xDFDE4000 = 1 1011 1111 101 1110 0100 0000 0000 0000 0000

$= -\ 1.101\ 1110\ 01 \times 2^{64}$

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

And, $(1.011\ 1110\ 01 \times 2^{64} + 1.111\ 1 \times 2^0) - 1.101\ 1110\ 01 \times 2^{64} =$

$-\ 0.01 \times 2^{64} = -1.0 \times 2^{64}$

= 1 1011 1101 000 0000 0000 0000 0000 0000

= 0xDE800000

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

5.31

(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.

5.33

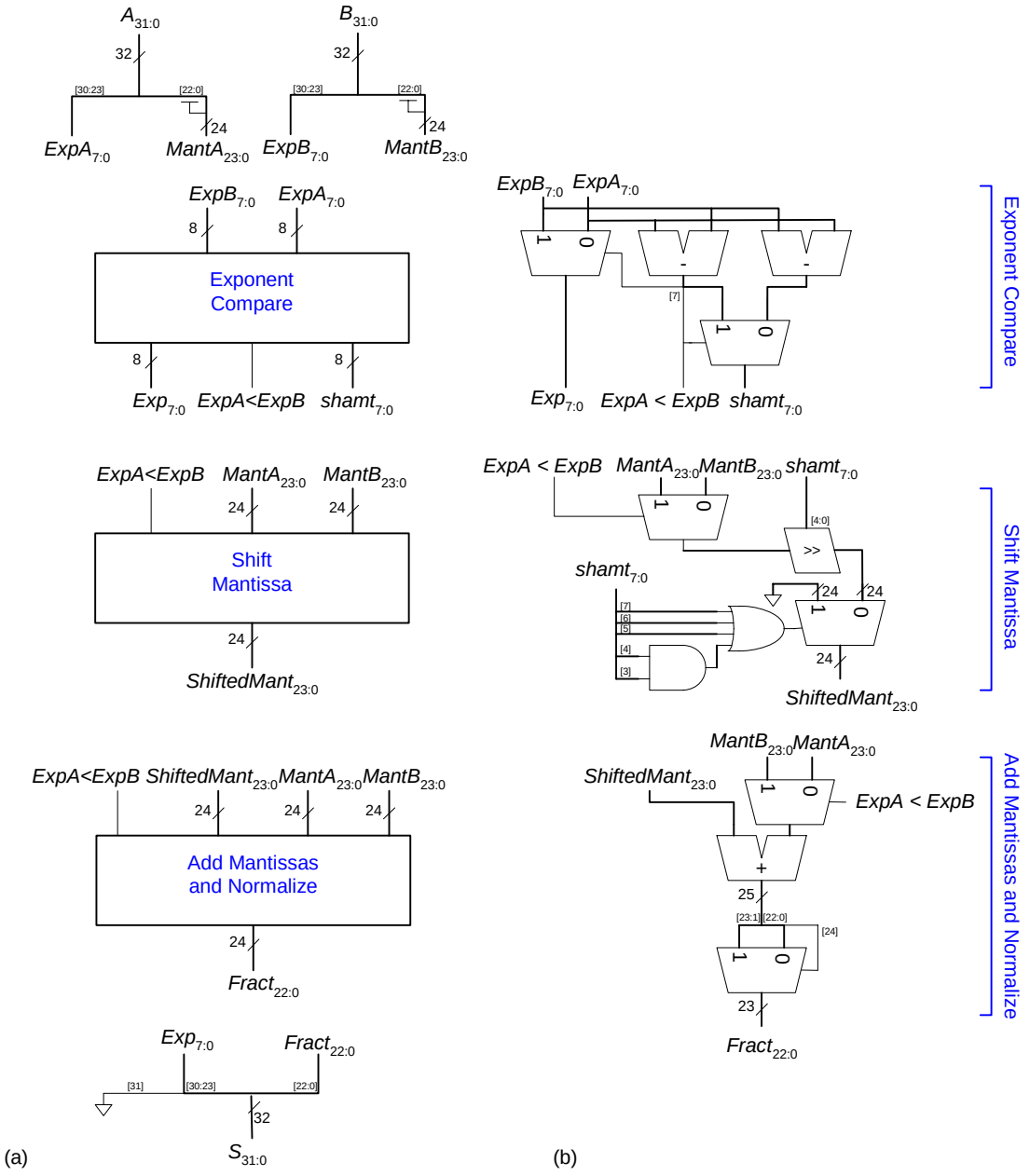


FIGURE 5.7 Floating-point adder hardware: (a) block diagram, (b) underlying hardware

Verilog

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

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

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

  expcomp   expcomp1(expa, expb, alessb, exp_pre,
                     shamt);
  shiftmant shiftmant1(alessb, manta, mantb,
                       shamt, shmant);
  addmant   addmant1(alessb, manta, mantb,
                     shmant, exp_pre, fract, exp);

endmodule
```

VHDL

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

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

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

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

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

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

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

  s     <= '0' & exp & fract;

  expcomp1:  expcomp
    port map(expa, expb, alessb, exp_pre, shamt);
  shiftmant1: shiftmant
    port map(alessb, manta, mantb, shamt, shmant);
  addmant1: addmant
    port map(alessb, manta, mantb, shmant,
             exp_pre, fract, exp);

end;
```

(continued from previous page)

Verilog

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

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

  always @(*)
    if (alessb) begin
      exp = expb;
      shamt = bminusa;
    end
    else begin
      exp = expa;
      shamt = aminusb;
    end
endmodule
```

VHDL

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

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

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

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

end;
```

(continued on next page)

(continued from previous page)

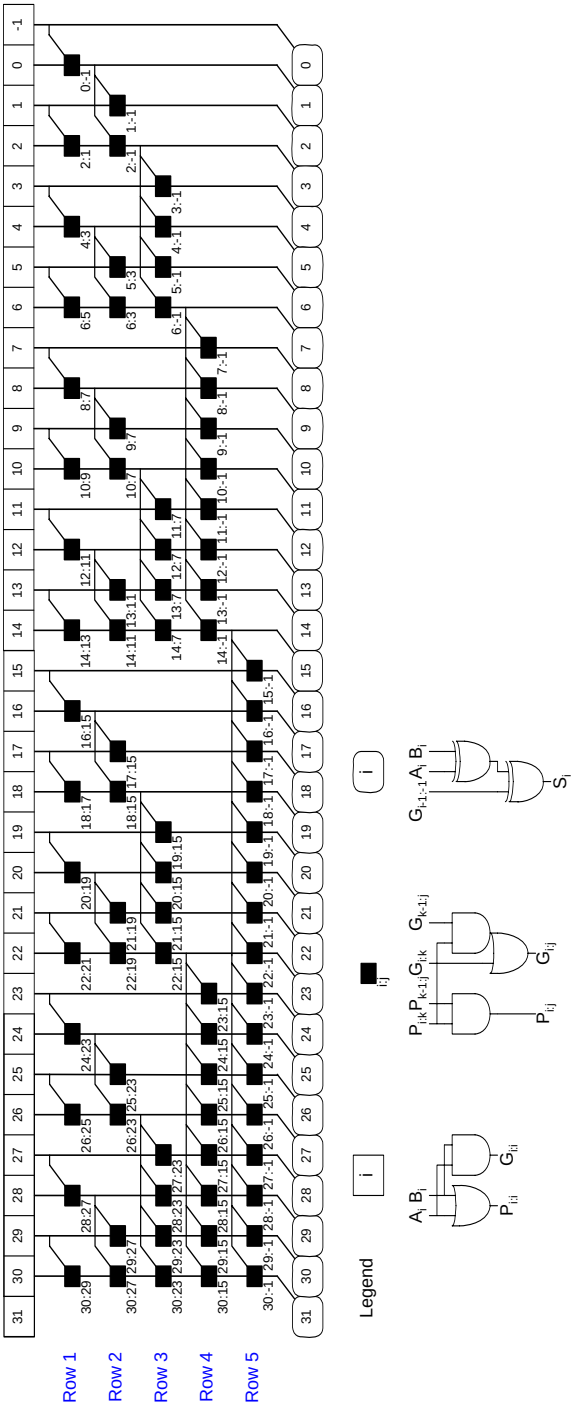
Verilog

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

  wire [23:0] shiftedval;

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

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

endmodule



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

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

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

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

endmodule
```

VHDL

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

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

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

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

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


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

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

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

end;
```

5.35 (a) *(figure on next page)*


Row 1
Row 2
Row 3
Row 4
Row 5
Legend
i
i:j
A_i B_i
$P_{i:i}$
$G_{i:i}$
$P_{i:k}$ $P_{k-1:j}$
$G_{i:k}$
$G_{k-1:j}$
$P_{i:j}$
$G_{i:j}$
$G_{i:-1}$ A_i B_i
S_i

5.35 (b)

Verilog

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

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

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

VHDL

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

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

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

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

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

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

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

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

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

(continued on next page)

(continued from previous page)

Verilog

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

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

VHDL

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

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

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

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

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

pik_3 <= p2(15)&p2(14)&p1(14)&p(27)&p2(11)&
         p2(10)&p1(10)&p(19)&p2(7)&p2(6)&
         p1(6)&p(11)&p2(3)&p2(2)&p1(2)&p(3);
gik_3 <= g2(15)&g2(14)&g1(14)&g(27)&g2(11)&
         g2(10)&g1(10)&g(19)&g2(7)&g2(6)&
         g1(6)&g(11)&g2(3)&g2(2)&g1(2)&g(3);
pkj_3 <= p2(13)&p2(13)&p2(13)&p2(13)&
         p2(9)&p2(9)&p2(9)&p2(9)&
         p2(5)&p2(5)&p2(5)&p2(5)&
         p2(1)&p2(1)&p2(1)&p2(1);
gkj_3 <= g2(13)&g2(13)&g2(13)&g2(13)&
         g2(9)&g2(9)&g2(9)&g2(9)&
         g2(5)&g2(5)&g2(5)&g2(5)&
         g2(1)&g2(1)&g2(1)&g2(1);

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

(continued on next page)

Verilog

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

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

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

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

endmodule
```

VHDL

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

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

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

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

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

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

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

end;
```

(continued on next page)

(continued from previous page)

Verilog

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

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

endmodule

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

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


module sum(input [31:0] g, a, b, output [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;
```

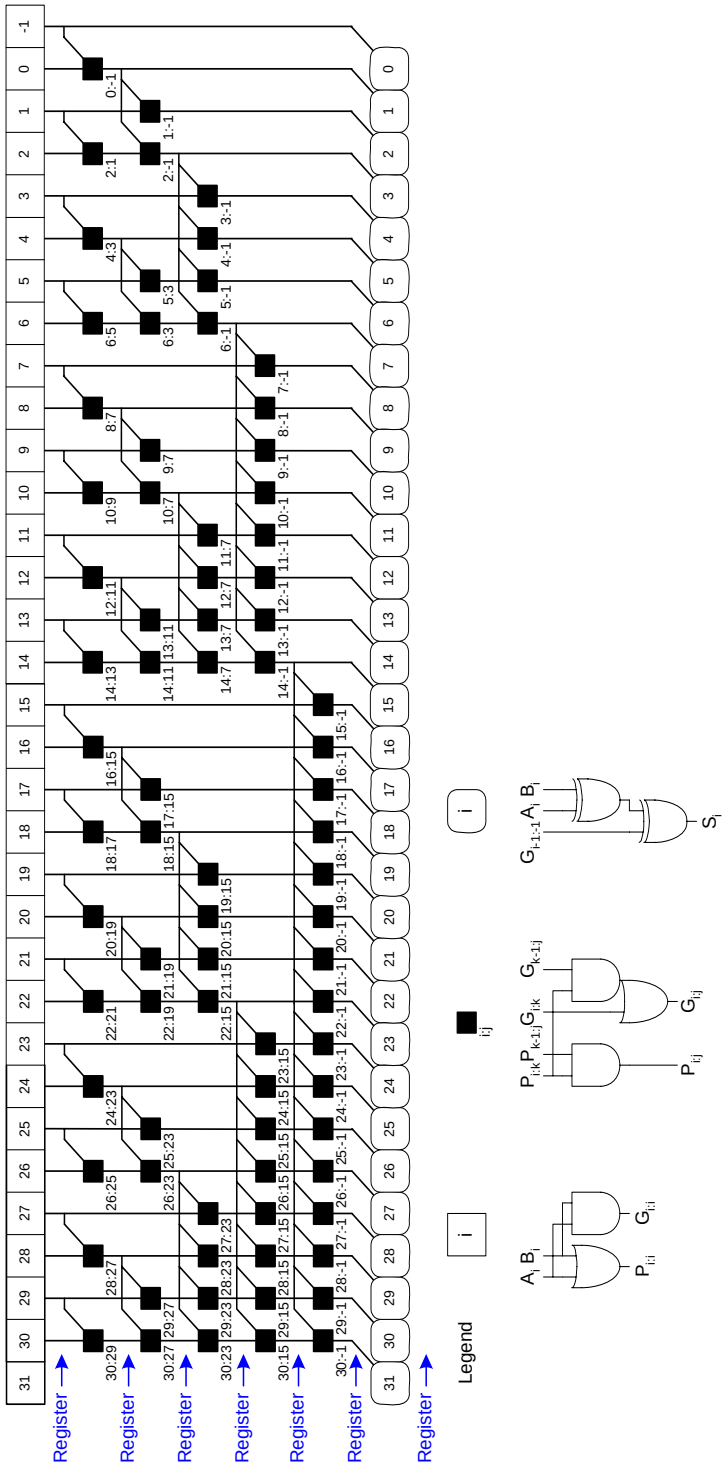
5.35 (c) Using Equation 5.11 to find the delay of the prefix adder:

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

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**

5.35 (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, $t_{pq} + t_{\text{setup}}$)



5.35 (e)

Verilog

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

  // p and g prefixes for rows 0 - 5
  wire [30:0] p0, p1, p2, p3, p4, p5;
  wire [30:0] g0, g1, g2, g3, g4, g5;
  wire 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
  wire [31:0] a0, a1, a2, a3, a4, a5,
              b0, b1, b2, b3, b4, b5;

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

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

g0[29],g0[27],g0[25],g0[23],g0[21],g0[19],g0[17],g0[15],
                           g0[13],g0[11],g0[9],g0[7],g0[5],g0[3],g0[1]},
{p1[29],p1[27],p1[25],p1[23],p1[21],p1[19],p1[17],p1[15],
                           p1[13],p1[11],p1[9],p1[7],p1[5],p1[3],p1[1],

g1[29],g1[27],g1[25],g1[23],g1[21],g1[19],g1[17],g1[15],
                           g1[13],g1[11],g1[9],g1[7],g1[5],g1[3],g1[1]});

  blackbox row1(clk,
 {p0[30],p0[28],p0[26],p0[24],p0[22],
                p0[20],p0[18],p0[16],p0[14],p0[12],
                p0[10],p0[8],p0[6],p0[4],p0[2],p0[0]},

               {p0[29],p0[27],p0[25],p0[23],p0[21],
                p0[19],p0[17],p0[15],p0[13],p0[11],
                p0[9],p0[7],p0[5],p0[3],p0[1],1'b0},

               {g0[30],g0[28],g0[26],g0[24],g0[22],
                g0[20],g0[18],g0[16],g0[14],g0[12],
                g0[10],g0[8],g0[6],g0[4],g0[2],g0[0]},

               {g0[29],g0[27],g0[25],g0[23],g0[21],
                g0[19],g0[17],g0[15],g0[13],g0[11],
                g0[9],g0[7],g0[5],g0[3],g0[1],g_1_0},

               {p1[30],p1[28],p1[26],p1[24],p1[22],p1[20],
 p1[18],p1[16],p1[14],p1[12],p1[10],p1[8],
 p1[6],p1[4],p1[2],p1[0]},

               {g1[30],g1[28],g1[26],g1[24],g1[22],g1[20],
 g1[18],g1[16],g1[14],g1[12],g1[10],g1[8],
 g1[6],g1[4],g1[2],g1[0]});

  // row 2
  flop #(2)  flop2_pg_1(clk, {p_1_1,g_1_1}, {p_1_2,g_1_2});
                          flop             #(30)             flop2_pg(clk,
{p1[28:27],p1[24:23],p1[20:19],p1[16:15],p1[12:11],
                           p1[8:7],p1[4:3],p1[0],
```

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

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

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


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

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


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


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

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

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

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

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

  blackbox row5(clk,
```

```
               p4[30:15],
 {16{p4[14]}},
 g4[30:15],
 {16{g4[14]}},
 p5[30:15], g5[30:15]);

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

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

endmodule

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

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

endmodule

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

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


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

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

endmodule

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

  always @(posedge clk)
    q <= d;
endmodule
```

5.35 (e)

VHDL

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

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

architecture synth of prefixaddpipe is
  component pgblock
    port(clk:  in  STD_LOGIC;
         a, b: in  STD_LOGIC_VECTOR(30 downto 0);
         p, g: out STD_LOGIC_VECTOR(30 downto 0));
  end component;
  component sumblock is
    port (clk:     in  STD_LOGIC;
          a, b, g: in  STD_LOGIC_VECTOR(31 downto 0);
          s:       out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component flop is generic(width: integer);
    port(clk: in  STD_LOGIC;
         d:   in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:   out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component flop1 is
    port(clk:        in  STD_LOGIC;
         d:          in  STD_LOGIC;
         q:          out STD_LOGIC);
  end component;
  component row1 is
    port(clk:  in  STD_LOGIC;
         p0, g0: in  STD_LOGIC_VECTOR(30 downto 0);
         p_1_0, g_1_0: in STD_LOGIC;
         p1, g1: out STD_LOGIC_VECTOR(30 downto 0));
  end component;
  component row2 is
    port(clk:  in  STD_LOGIC;
         p1, g1: in  STD_LOGIC_VECTOR(30 downto 0);
         p2, g2: out STD_LOGIC_VECTOR(30 downto 0));
  end component;
  component row3 is
    port(clk:  in  STD_LOGIC;
         p2, g2: in  STD_LOGIC_VECTOR(30 downto 0);
         p3, g3: out STD_LOGIC_VECTOR(30 downto 0));
  end component;
  component row4 is
    port(clk:  in  STD_LOGIC;
         p3, g3: in  STD_LOGIC_VECTOR(30 downto 0);
         p4, g4: out STD_LOGIC_VECTOR(30 downto 0));
  end component;
  component row5 is
    port(clk:  in  STD_LOGIC;
         p4, g4: in  STD_LOGIC_VECTOR(30 downto 0);
         p5, g5: out STD_LOGIC_VECTOR(30 downto 0));
  end component;

  -- p and g prefixes for rows 0 - 5
  signal p0, p1, p2, p3, p4, p5: STD_LOGIC_VECTOR(30 downto 0);
```

```
  signal g0, g1, g2, g3, g4, g5: STD_LOGIC_VECTOR(30 downto 0);

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

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

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

begin

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


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

  -- pipeline p and g for column -1
  p_1_0 <= '0'; flop_1_g0: flop1 port map (clk, cin, g_1_0);
  flop_1_p1: flop1 port map (clk, p_1_0, p_1_1);
  flop_1_g1: flop1 port map (clk, g_1_0, g_1_1);
  flop_1_p2: flop1 port map (clk, p_1_1, p_1_2);
  flop_1_g2: flop1 port map (clk, g_1_1, g_1_2);
  flop_1_p3: flop1 port map (clk, p_1_2, p_1_3); flop_1_g3:
  flop1 port map (clk, g_1_2, g_1_3);
  flop_1_p4: flop1 port map (clk, p_1_3, p_1_4);
  flop_1_g4: flop1 port map (clk, g_1_3, g_1_4);
  flop_1_p5: flop1 port map (clk, p_1_4, p_1_5);
  flop_1_g5: flop1 port map (clk, g_1_4, g_1_5);

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

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


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

```
architecture synth of pgblock is
begin
  process(clk) begin
    if clk'event and clk = '1' 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 clk'event and clk = '1' then
      pij <= pik and pkj;
      gij <= gik or (pik and gkj);
    end if;
  end process;
end;


library IEEE; use IEEE.STD_LOGIC_1164.all;

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

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

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

architecture synth of flop1 is
begin
  process(clk) begin
    if clk'event and clk = '1' then
      q <= d;
    end if;
  end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

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

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

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

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

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

  p1(29) <= pg1_out(29); p1(27)<= pg1_out(28); p1(25)<= pg1_out(27);
  p1(23) <= pg1_out(26);
  p1(21) <= pg1_out(25); p1(19) <= pg1_out(24); p1(17) <= pg1_out(23);
  p1(15) <= pg1_out(22); p1(13) <= pg1_out(21); p1(11) <= pg1_out(20);
  p1(9) <= pg1_out(19); p1(7) <= pg1_out(18); p1(5) <= pg1_out(17);
  p1(3) <= pg1_out(16); p1(1) <= pg1_out(15);
  g1(29) <= pg1_out(14); g1(27) <= pg1_out(13); g1(25) <= pg1_out(12);
  g1(23) <= pg1_out(11); g1(21) <= pg1_out(10); g1(19) <= pg1_out(9);
  g1(17) <= pg1_out(8); g1(15) <= pg1_out(7); g1(13) <= pg1_out(6);
  g1(11) <= pg1_out(5); g1(9) <= pg1_out(4); g1(7) <= pg1_out(3);
  g1(5) <= pg1_out(2); g1(3) <= pg1_out(1); g1(1) <= pg1_out(0);
```

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

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

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

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

library IEEE; use IEEE.STD_LOGIC_1164.all;

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

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

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

  -- internal signals for pipeline registers
  signal pg1_in, pg2_out: STD_LOGIC_VECTOR(29 downto 0);

begin
  pg1_in <= (p1(28 downto 27)&p1(24 downto 23)&p1(20 downto 19)&
             p1(16 downto 15)&
             p1(12 downto 11)&p1(8 downto 7)&p1(4 downto 3)&p1(0)&
             g1(28 downto 27)&g1(24 downto 23)&g1(20 downto 19)&
             g1(16 downto 15)&
             g1(12 downto 11)&g1(8 downto 7)&g1(4 downto 3)&g1(0));
  flop2_pg: flop generic map(30) port map (clk, pg1_in, pg2_out);

  p2(28 downto 27) <= pg2_out(29 downto 28);
  p2(24 downto 23) <= pg2_out(27 downto 26);
  p2(20 downto 19) <= pg2_out( 25 downto 24);
```

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

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

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

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

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

end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

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

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

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

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

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

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

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

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

end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

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

architecture synth of row4 is
  component blackbox is
    port (clk:      in  STD_LOGIC;
          pik, pkj: in STD_LOGIC_VECTOR(15 downto 0);
          gik, gkj: in STD_LOGIC_VECTOR(15 downto 0);
          pij:      out STD_LOGIC_VECTOR(15 downto 0);
          gij:      out STD_LOGIC_VECTOR(15 downto 0));
  end component;
  component flop is generic(width: integer);
    port(clk: in  STD_LOGIC;
         d:   in  STD_LOGIC_VECTOR(width-1 downto 0);
```

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

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

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

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

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

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

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

end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity row5 is
  port(clk:    in STD_LOGIC;
       p4, g4: in  STD_LOGIC_VECTOR(30 downto 0);
       p5, g5: out STD_LOGIC_VECTOR(30 downto 0));
end;
architecture synth of row5 is
  component blackbox is
    port (clk:      in  STD_LOGIC;
       pik, pkj: in STD_LOGIC_VECTOR(15 downto 0);
          gik, gkj: in STD_LOGIC_VECTOR(15 downto 0);
          pij:      out STD_LOGIC_VECTOR(15 downto 0);
          gij:      out STD_LOGIC_VECTOR(15 downto 0));
  end component;
  component flop is generic(width: integer);
    port(clk: in  STD_LOGIC;
         d:   in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:   out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;

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

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

begin
```

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

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

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

end;
```

5.37

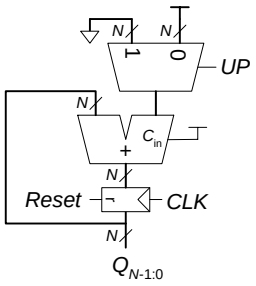


FIGURE 5.8 Up/Down counter

5.39

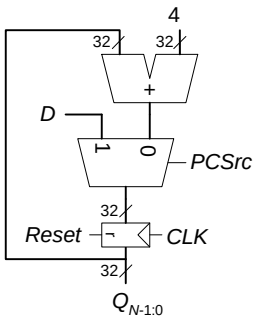


FIGURE 5.9 32-bit counter that increments by 4 or loads a new value, *D*

5.41

Verilog

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

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

  assign sout = q[3];

endmodule
```

VHDL

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

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

architecture synth of scanflop4 is
begin
  process(clk, test) begin
    if clk'event and clk = '1' then
    if test = '1' then
        q <= d;
      else
     q <= q(2 downto 0) & sin;
      end if;
    end if;
  end process;

  sout <= q(3);

end;
```

5.43

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 5.10).

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.

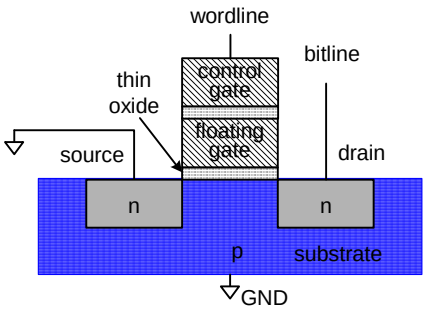
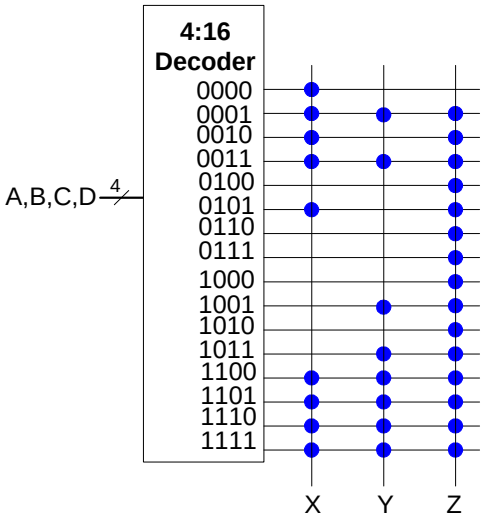


FIGURE 5.10 Flash EEPROM

5.45



5.47

(a) Number of inputs = 2 × 16 + 1 = 33
Number of outputs = 16 + 1 = 17

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

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

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

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

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

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

5.49
(a) $Y = F(A,B,C,D,E,F,G,H,I) = ABCDEFGHI$
(b) $Y = F(A,B,C,D,E,F,G,H) = ABCD + A\overline{B}C\overline{D}E + FGH$

5.51
(a) 1 CLB
(b)
$t_{pd} = t_{CLB}$
$= 2.7$ ns
$T_c \geq t_{pcq} + t_{pd} + t_{setup}$
$\geq [2.8 + 2.7 + 3.9]$ ns
$= 9.4$ ns
$f = 1 / 9.4$ ns = **106 MHz**

5.51 (c)
First, we check that there is no hold time violation with this amount of clock skew.
$t_{cd} = t_{pd} = 2.7$ ns
$t_{skew} < (t_{ccq} + t_{cd}) - t_{hold}$
$< [(2.8 + 2.7) - 0]$ ns
< 5.5 ns
5 ns is less than 5.5 ns, so there is no hold time violation.
Now we find the fastest frequency at which it can run.
$T_c \geq t_{pcq} + t_{pd} + t_{setup} + t_{skew}$
$\geq [2.8 + 2.7 + 3.9 + 5]$ ns
$= 14.4$ ns
$f = 1 / 14.4$ ns = **69 MHz**

Question 5.1

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

Question 5.3

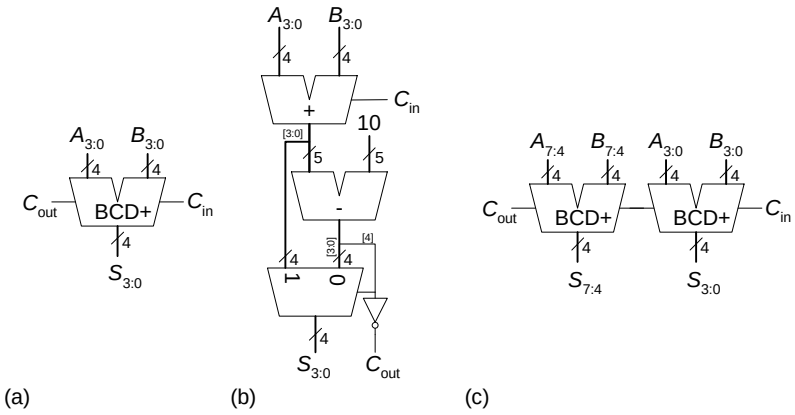


FIGURE 5.11 BCD adder: (a) 4-bit block, (b) underlying hardware, (c) 8-bit BCD adder

(continued from previous page)

Verilog

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

  wire 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  [3:0] a, b, input cin,
                output [3:0] s, output cout);

  wire [4:0] result, sub10;

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

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

endmodule
```

VHDL

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

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

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

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

end;


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

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

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

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

end;
```

CHAPTER 6

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.

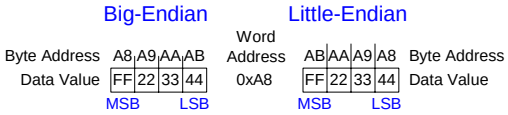
- The register file has only 32 registers.
- The ISA (instruction set architecture) includes only a small number of commonly used instructions. This keeps the hardware small and, thus, fast.

- 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`).

6.3
(a) $42 \times 4 = 42 \times 2^2 = 101010_2 << 2 = 10101000_2 = 0xA8$
(b) 0xA8 through 0xAB
(c)



6.5
(a) 0x534F5300
(b) 0x436F6F6C2100
(c) 0x416C7973736100 (depends on the persons name)

6.7
0x02114020
0x8de80020
0x2010fff6

6.9
(a)

```
addi $s0, $0, 73
sw   $t1, -7($t2)
```

(b)
0x00000049 (`addi`)
0xfffffff9 (`sw`)

6.11

```
ori $t0, $t1, 0xF234
nor $t0, $t0, $0
```

6.13

```
int find42( int array[], int size)
{
  int i;     // index into array

  for (i = 0; i < size; i = i+1)
    if (array[i] == 42)
      return i;

  return -1;
}
```

6.15

```
find42: addi $t0, $0, 0     # $t0 = i = 0
        addi $t1, $0, 42    # $t1 = 42
  loop: slt  $t3, $t0, $a1  # $t3 = 1 if i < size (not at end of array)
        beq  $t3, $0, exit  # if reached end of array, return -1
        sll  $t2, $t0, 2    # $t2 = i*4
        add  $t2, $t2, $a0  # $t2 = address of array[i]
        lw   $t2, 0($t2)    # $t2 = array[i]
        beq  $t2, $t1, done # $t2 == 42?
        addi $t0, $t0, 1    # i = i + 1
        j    loop
  done: add  $v0, $t0, $0   # $v0 = i
        jr   $ra
  exit: addi $v0, $0, -1    # $v0 = -1
        jr   $ra
```

6.17

(a) The stack frames of each procedure are:

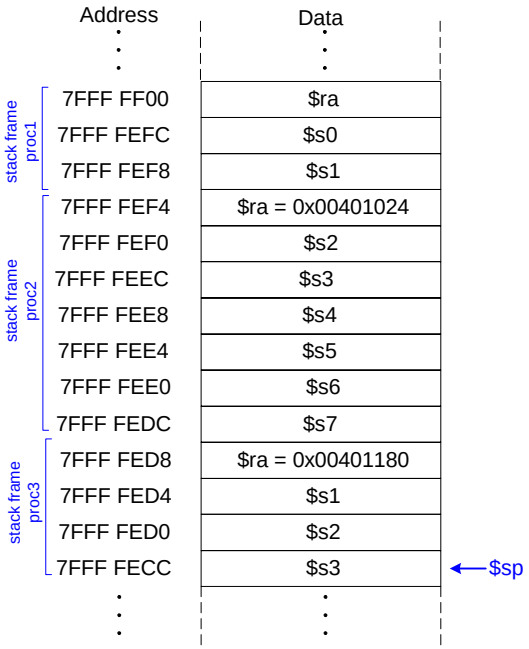
proc1: 3 words deep (for `$s0` - `$s1`, `$ra`)

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 0x7FFFFF04 just before the procedure calls.



6.19

(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
= **0x08100C01**

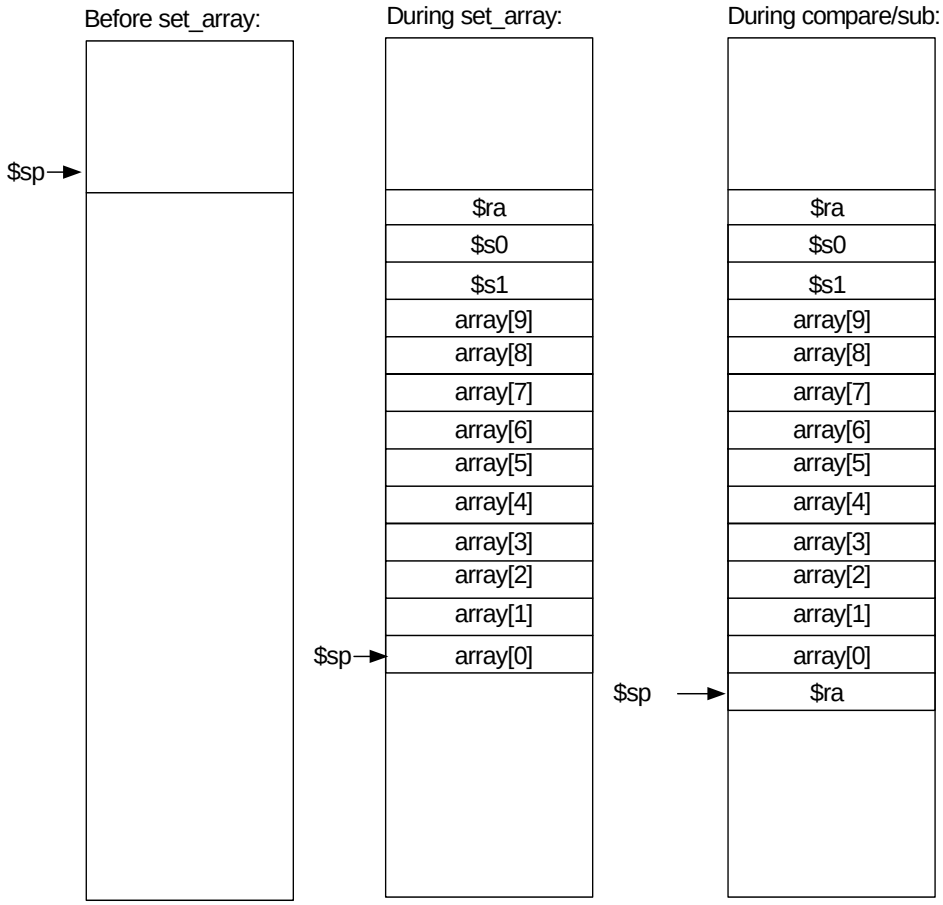
6.21

(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)
```

```
            sll  $t1,$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
```

6.21 (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.

6.23

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

6.25

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.

6.27

```
# high-level code
void little2big(int[] array)
{
  int i;

  for (i = 0; i < 10; i = i + 1) {
    array[i] = ((array[i] & 0xFF) << 24) ||
                (array[i] & 0xFF00) << 8) ||
                (array[i] & 0xFF0000) >> 8) ||
               ((array[i] >> 24) & 0xFF));
  }
}
```

```
# 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:
```

6.29

```
# define the masks in the global data segment
        .data
mmask:  .word 0x007FFFFF
emask:  .word 0x7F800000
ibit:   .word 0x00800000
obit:   .word 0x01000000

        .text

flpadd: 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
        lw $t4,emask            # load exponent mask
        and $t2,$s0,$t4         # extract exponent from $s0 (a)
        srl $t2,$t2,23          # shift exponent right
        and $t3,$s1,$t4         # extract exponent from $s1 (b)
        srl $t3,$t3,23          # shift exponent right
match:  beq $t2,$t3,addsig      # check whether the exponents match
        bgeu $t2,$t3,shiftb     # determine which exponent is larger
```

```
shifta: sub $t4,$t3,$t2         # calculate difference in exponents
        srav $t0,$t0,$t4        # shift a by calculated difference
        add $t2,$t2,$t4         # update a's exponent
        j addsig                # skip to the add
shiftb: sub $t4,$t2,$t3         # calculate difference in exponents
        srav $t1,$t1,$t4        # shift b by calculated difference
        add $t3,$t3,$t4         # update b's exponent (not necessary)
addsig: add $t5,$t0,$t1         # add the mantissas
norm:   lw $t4,obit             # load mask for bit 24 (overflow bit)
        and $t4,$t5,$t4         # mask bit 24
        beq $t4,$0,done         # right shift not needed because bit 24=0
        srl $t5,$t5,1           # shift right once by 1 bit
        addi $t2,$t2,1          # increment exponent
done:   lw $t4,mmask            # load mask
        and $t5,$t5,$t4         # mask mantissa
        sll $t2,$t2,23          # shift exponent into place
        lw $t4,emask            # load mask
        and $t2,$t2,$t4         # mask exponent
        or $v0,$t5,$t2          # place mantissa and exponent into $v0
        jr $ra                  # return to caller
```

6.31

(a)

```
beq $t1, imm31:0, L

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

(b)

```
ble $t3, $t5, L

slt $at, $t5, $t3
beq $at, $0, L
```

(c)

```
bgt $t3, $t5, L

slt $at, $t5, $t3
bne $at, $0, L
```

(d)

```
bge $t3, $t5, L

slt $at, $t3, $t5
beq $at, $0, L
```

(e)

```
# note: this is not actually a pseudo instruction supplied by MIPS
# but the functionality can be implemented as shown below
addi $t0, $2, imm31:0

lui $at, imm31:16
ori $at, $at, imm15:0
add $t0, $2, $at
```

(f)

```
lw $t5, imm31:0($s0)

lui $at, imm31:16
ori $at, $at, imm15:0
add $at, $at, $s0
lw  $t5, 0($at)
```

(g)

```
rol $t0, $t1, 5

srl $at, $t1, 27
sll $t0, $t1, 5
or  $t0, $t0, $at
```

(h)

```
ror $s4, $t6, 31

sll $at, $t6, 1
srl $s4, $t6, 31
or  $s4, $s4, $at
```

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

```
// 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

```
# $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   $t4, 0($t4)     # $t4 = array[i]
        beq  $t4, $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 $a1, $a1, -1     # i = i-1
        addi $a2, $a2, 1      # j = j+1
        j    reverse
exit:   jr  $ra
```

Question 6.5

High-Level Code

```
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

```
# $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
  sllv $t0, $t0, $a1  # $t0 = $t0 << shamt
  or   $v0, $v0, $t0  # $v0 = $v0 | $t0
  jr   $ra            # return
```

Question 6.7

High-Level Code

```
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

  int i = 0;
  while (j > i) {
    tmp = array[i];
    if (array[i] != array[j])
      return false;
    j = j-1;
    i = i+1;
  }



  return true;
}
```

MIPS Assembly Code

```
# $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

        addi $t1, $0, 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
```

CHAPTER 7

7.1

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

7.3

(a) `sll`

First, we modify the ALU.

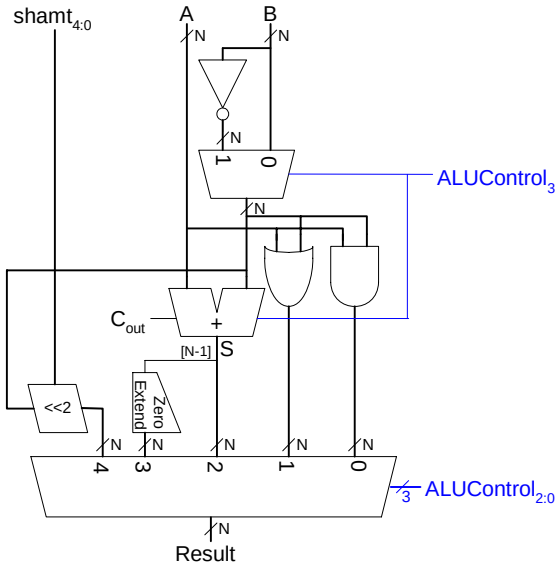


FIGURE 7.1 Modified ALU to support `sll`

$f_{3:0}$	function
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	SLL

TABLE 7.1 Modified ALU operations to support `sll`

aluop	funct	alucontrol
00	X	0010 (add)
X1	X	1010 (subtract)
1X	100000 (`add`)	0010 (add)
1X	100010 (`sub`)	1010 (subtract)
1X	100100 (`and`)	0000 (and)
1X	100101 (`or`)	0001 (or)
1X	101010 (`slt`)	1011 (set less than)
1X	101010 (`sll`)	0100 (shift left logical)

TABLE 7.2 ALU decoder truth table

Then we modify the datapath.

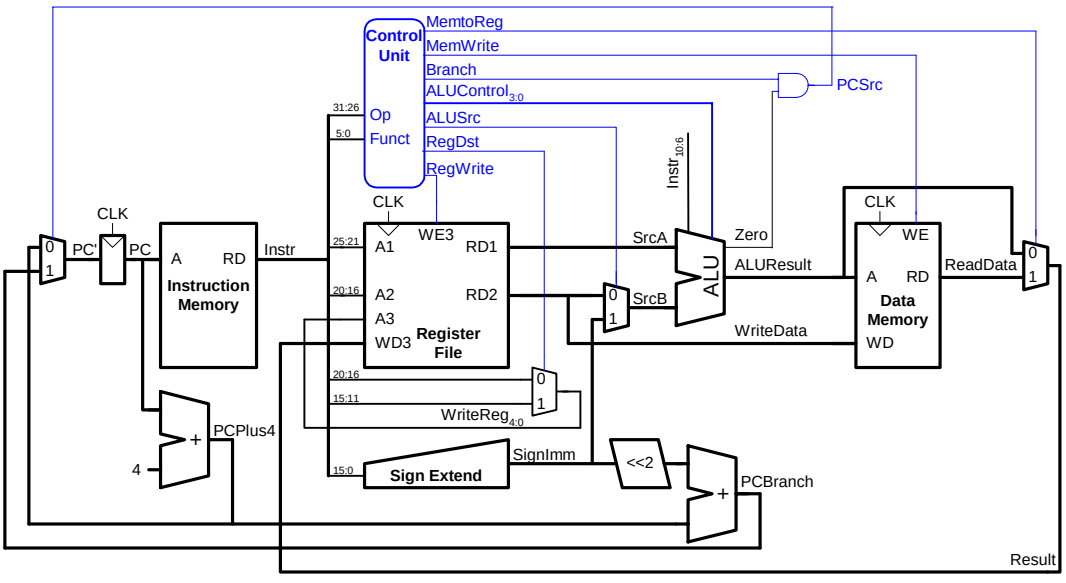


FIGURE 7.2 Modified single-cycle MIPS processor extended to run `sll`

7.3 (b) `lui`

.

instruction	opcode	regwrite	regdst	alusrc	branch	memwrite	memtoreg	aluop
R-type	000000	1	1	00	0	0	0	10
`lw`	100011	1	0	01	0	0	1	00
`sw`	101011	0	X	01	0	1	X	00
`beq`	000100	0	X	00	1	0	X	01
`lui`	001111	1	0	10	0	0	0	00

TABLE 7.3 Main decoder truth table enhanced to support `lui`

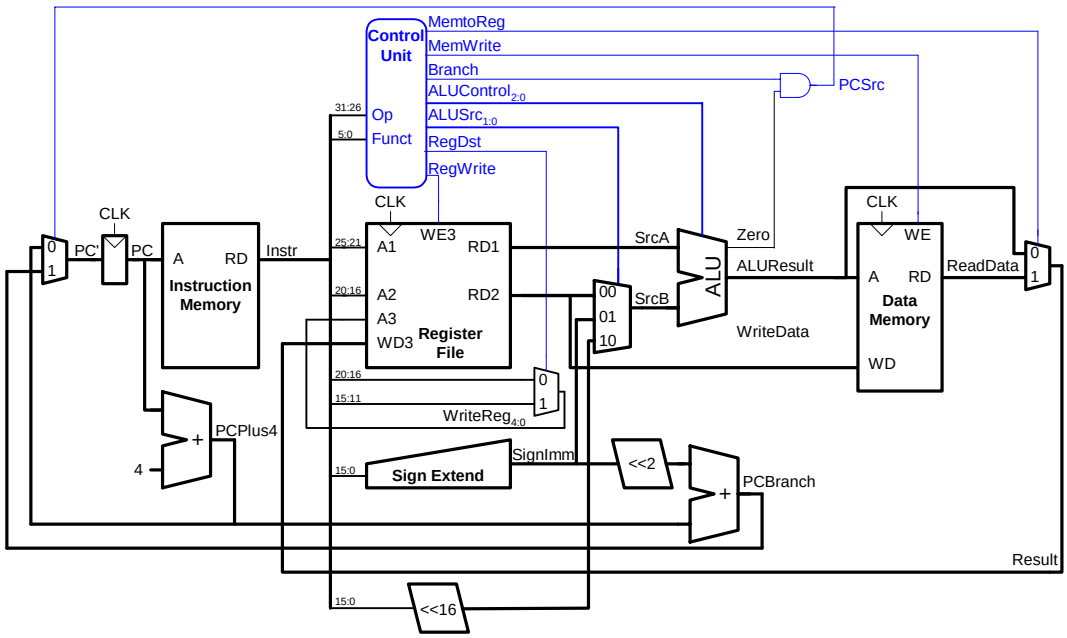


FIGURE 7.3 Modified single-cycle datapath to support `lui`

7.3 (c) `slti`

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

aluop	funct	alucontrol
00	X	010 (add)
01	X	110 (subtract)
10	100000 (`add`)	010 (add)
10	100010 (`sub`)	110 (subtract)
10	100100 (`and`)	000 (and)
10	100101 (`or`)	001 (or)
10	101010 (`slt`)	111 (set less than)
11	X	111 (set less than)

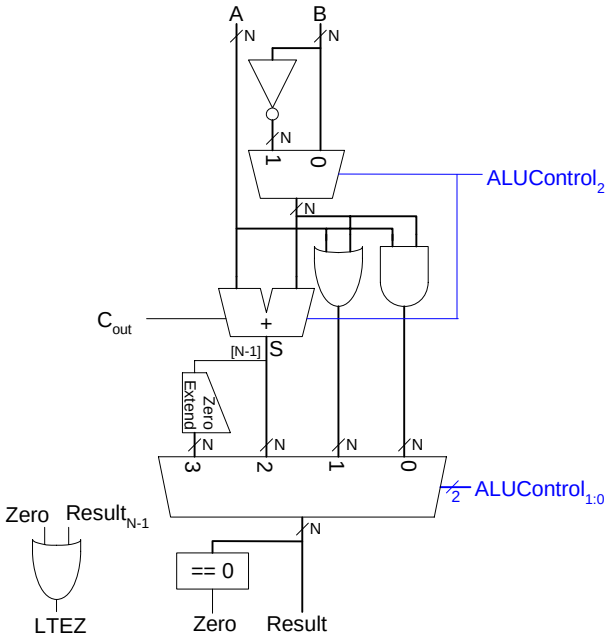
TABLE 7.4 ALU decoder truth table

instruction	opcode	regwrite	regdst	alusrc	branch	memwrite	memtoreg	aluop
R-type	000000	1	1	0	0	0	0	10
`lw`	100011	1	0	1	0	0	1	00
`sw`	101011	0	X	1	0	1	X	00
`beq`	000100	0	X	0	1	0	X	01
`slti`	001011	1	0	1	0	0	0	11

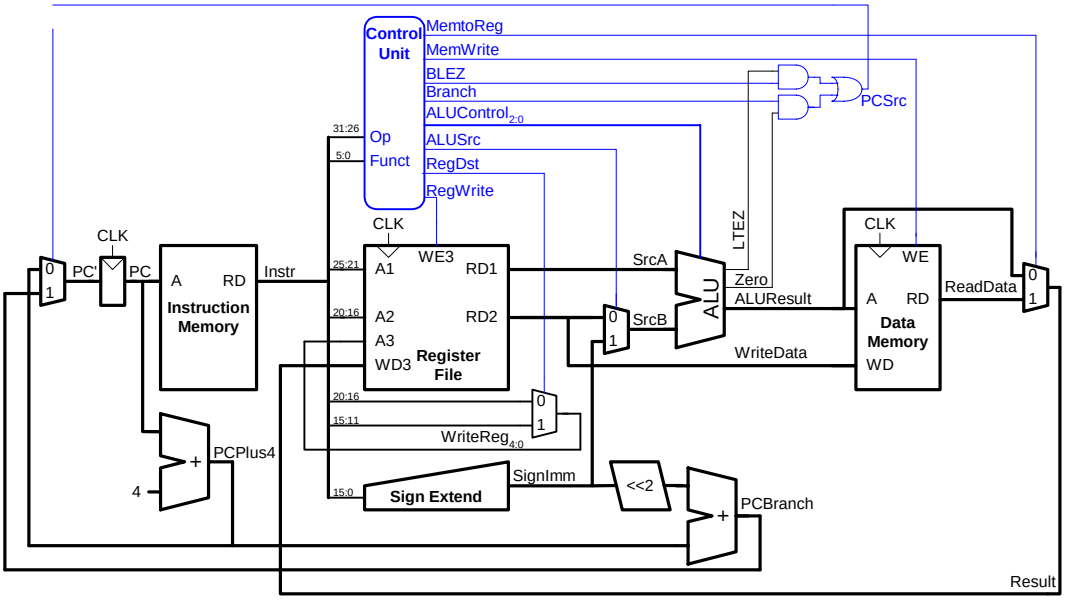
TABLE 7.5 Main decoder truth table enhanced to support `slti`

7.3 (d) `blez`

First, we modify the ALU



Then, we modify the datapath

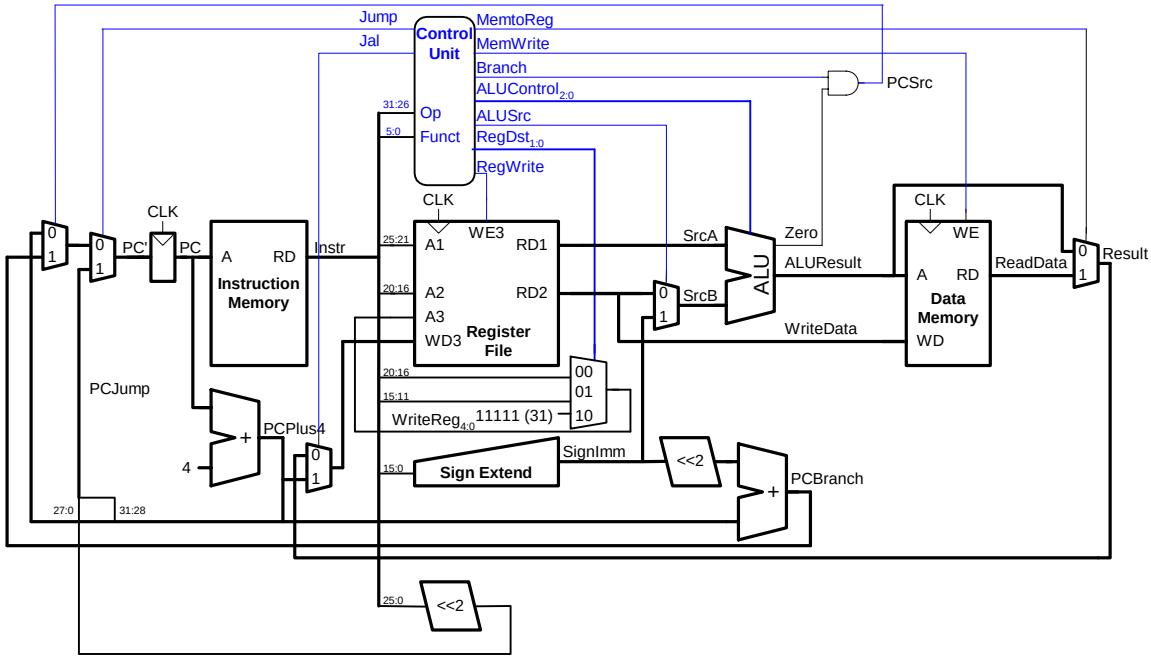


.

instruction	opcode	regwrite	regdst	alusrc	branch	memwrite	memtoreg	aluop	blez
R-type	000000	1	1	0	0	0	0	10	0
`lw`	100011	1	0	1	0	0	1	00	0
`sw`	101011	0	X	1	0	1	X	00	0
`beq`	000100	0	X	0	1	0	X	01	0
`blez`	000110	0	X	0	0	0	X	01	1

TABLE 7.6 Main decoder truth table enhanced to support `blez`

7.3 (e) `jal`



.

instruction	opcode	regwrite	regdst	alusrc	branch	memwrite	memtoreg	aluop	jump	jal
R-type	000000	1	01	0	0	0	0	10	0	0
`lw`	100011	1	00	1	0	0	1	00	0	0
`sw`	101011	0	XX	1	0	1	X	00	0	0
`beq`	000100	0	XX	0	1	0	X	01	0	0
`addi`	001000	1	00	1	0	0	0	00	0	0
`j`	000010	0	XX	X	X	0	X	XX	1	0
`jal`	000011	1	10	X	X	0	X	XX	1	1

TABLE 7.7 Main decoder truth table enhanced to support `jal`

7.3 (f) `lh`

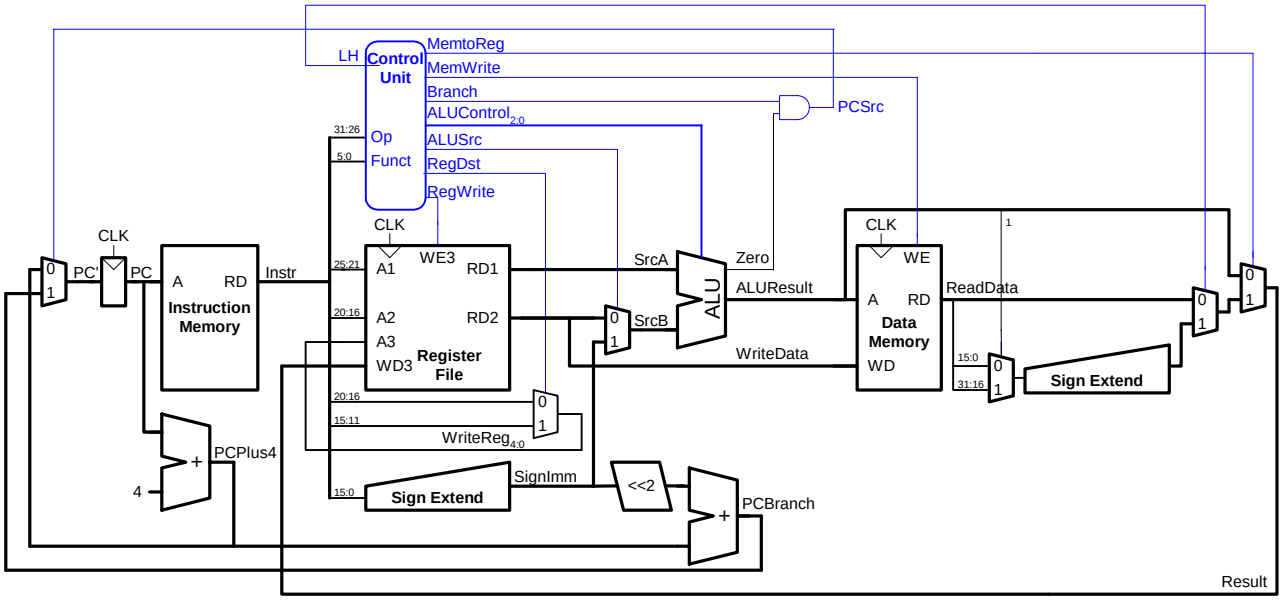


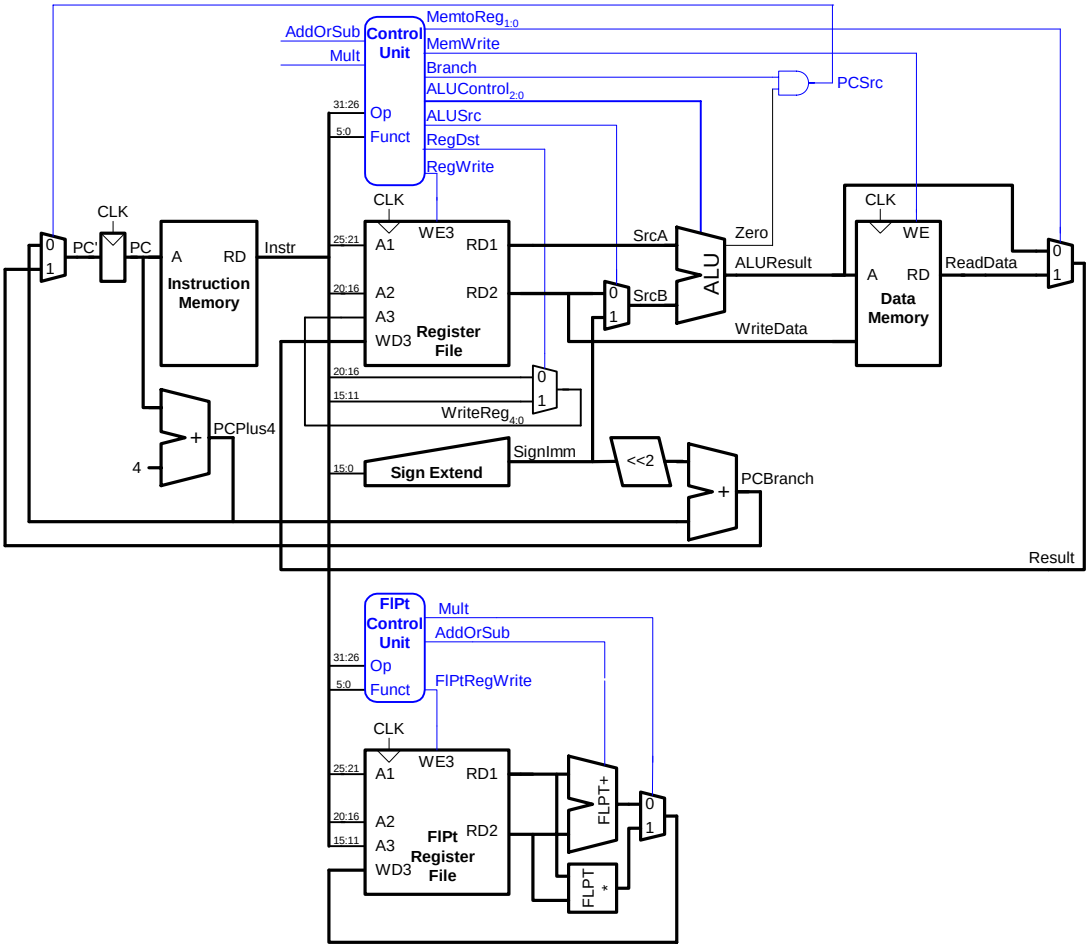
FIGURE 7.4 Modified single-cycle datapath to support `lh`

.

instruction	opcode	regwrite	regdst	alusrc	branch	memwrite	memtoreg	aluop	lh
R-type	000000	1	01	0	0	0	0	10	0
`lw`	100011	1	00	1	0	0	1	00	0
`sw`	101011	0	XX	1	0	1	X	00	0
`beq`	000100	0	XX	0	1	0	X	01	0
`lh`	100001	1	00	1	0	0	1	00	1

TABLE 7.8 Main decoder truth table enhanced to support `lh`

7.5



instruction	opcode	regwrite	regdst	alusrc	branch	memwrite	memtoreg	aluop
R-type	000000	1	01	0	0	0	00	10
`lw`	100011	1	00	1	0	0	01	00
`sw`	101011	0	XX	1	0	1	XX	00

TABLE 7.9 Main decoder truth table enhanced to support `add.s`, `sub.s`, and `mult.s`

instruction	opcode	regwrite	regdst	alusrc	branch	memwrite	memtoreg	aluop
`beq`	000100	0	XX	0	1	0	XX	01
`f-type`	010001	0	XX	X	0	0	XX	XX

TABLE 7.9 Main decoder truth table enhanced to support `add.s`, `sub.s`, and `mult.s`

instruction	opcode	flptregwrite
`f-type`	010001	1
`others`		0

TABLE 7.10 Floating point main decoder truth table enhanced to support `add.s`, `sub.s`, and `mult.s`

funct	mult	addorsub
000000 (add)	0	1
000001 (sub)	0	0
000010 (mult)	1	X

TABLE 7.11 Adder/subtractor decoder truth table

7.7

T_c = 930 ps

It would take **93 seconds** to execute 100 billion instructions.

7.9

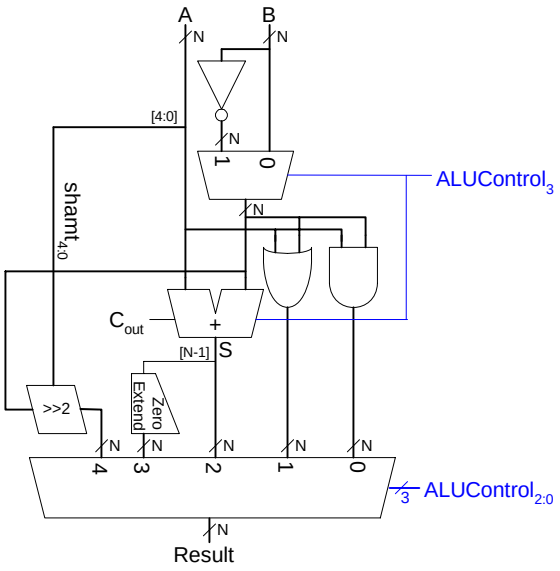
(a) R-type, `addi`

(b) `lw`, `sw`, `addi`, R-type

(c) all instructions

7.11
(a) `srlv`

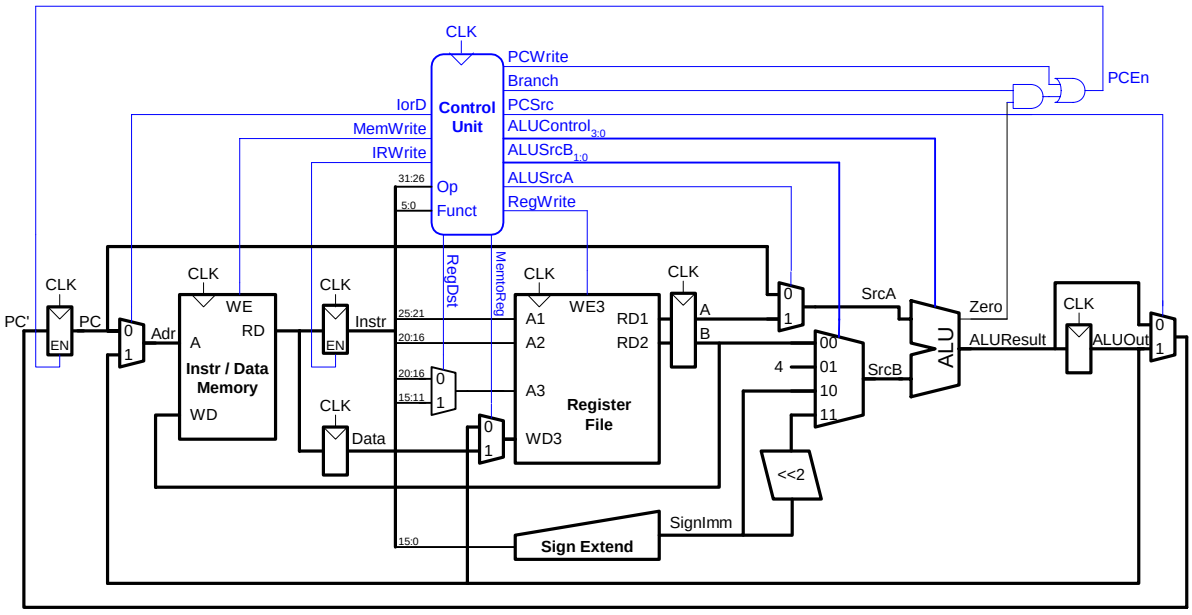
First, we show the modifications to the ALU.



Next, we show the modifications to the ALU decoder.l

$f_{3:0}$	function
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	SRLV

Next, we show the changes to the datapath. The only modification is the width of *ALUControl*. No changes are made to the datapath main control FSM.



7.11 (b) `ori`

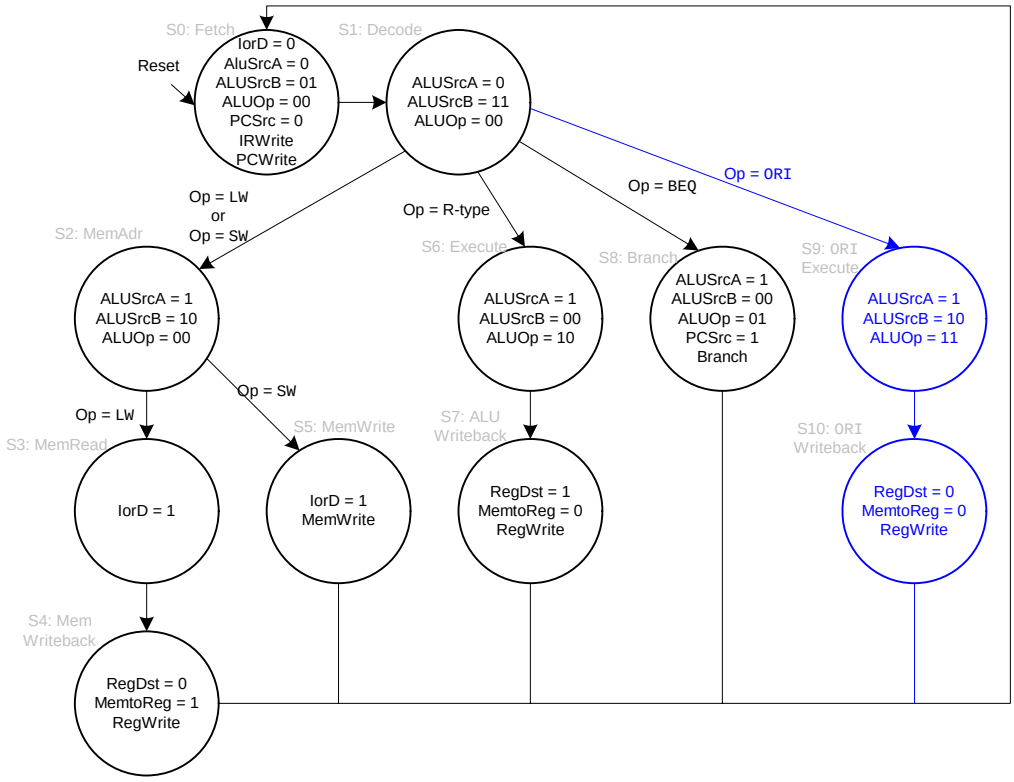
No changes are necessary in the datapath. Only the ALU decoder and main control FSM require modifications.

aluop	funct	alucontrol
00	X	010 (add)
01	X	110 (subtract)
11	X	001 (or)
10	100000 (`add`)	010 (add)
10	100010 (`sub`)	110 (subtract)
10	100100 (`and`)	000 (and)

TABLE 7.12 ALU decoder truth table

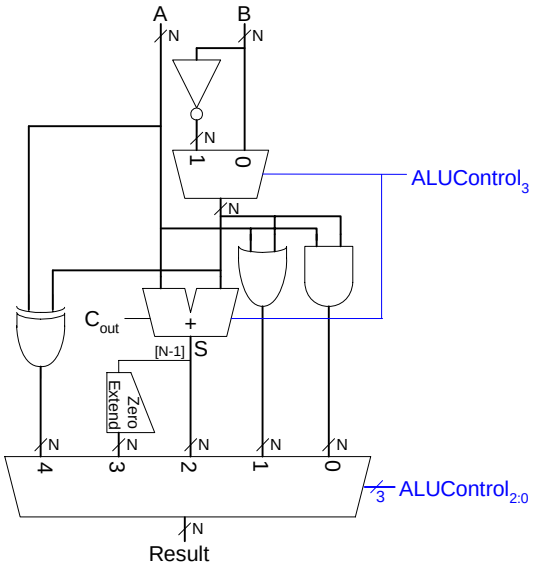
aluop	funct	alucontrol
10	100101 (or)	001 (or)
10	101010 (slt)	111 (set less than)

TABLE 7.12 ALU decoder truth table



7.11(c) `xori`

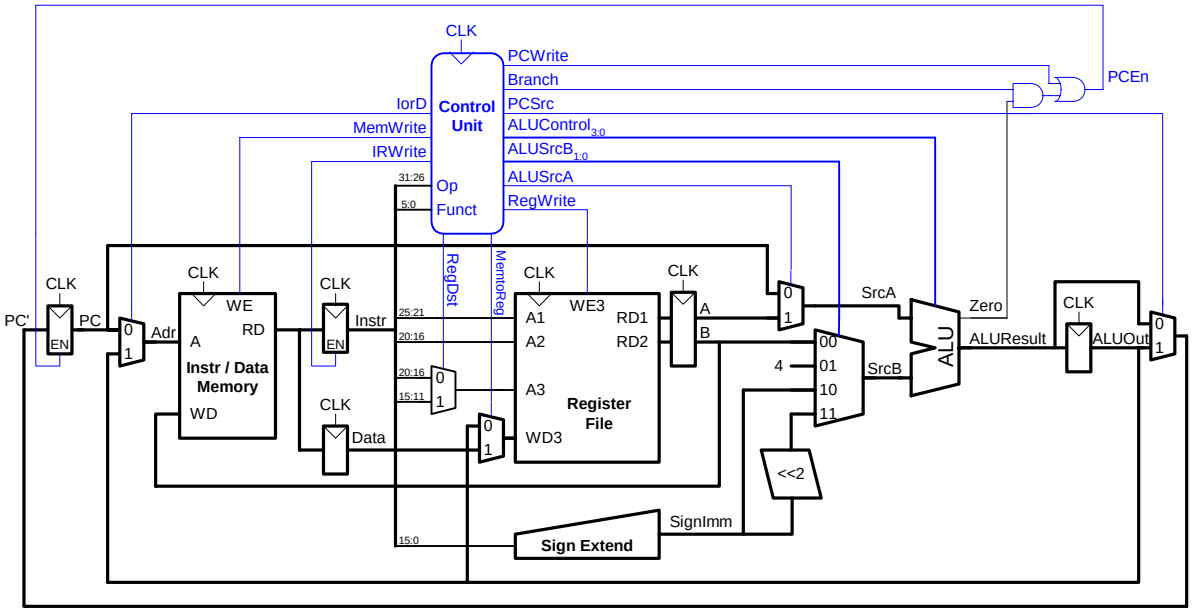
First, we modify the ALU and the ALU decoder.



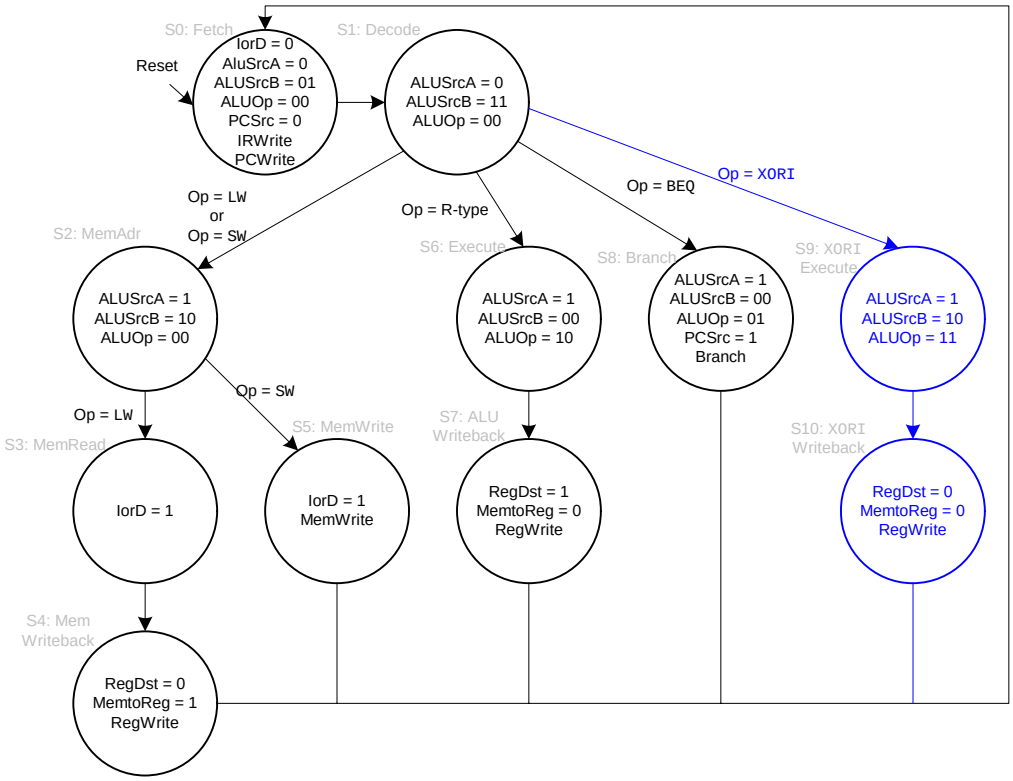
l

$f_{3:0}$	function
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

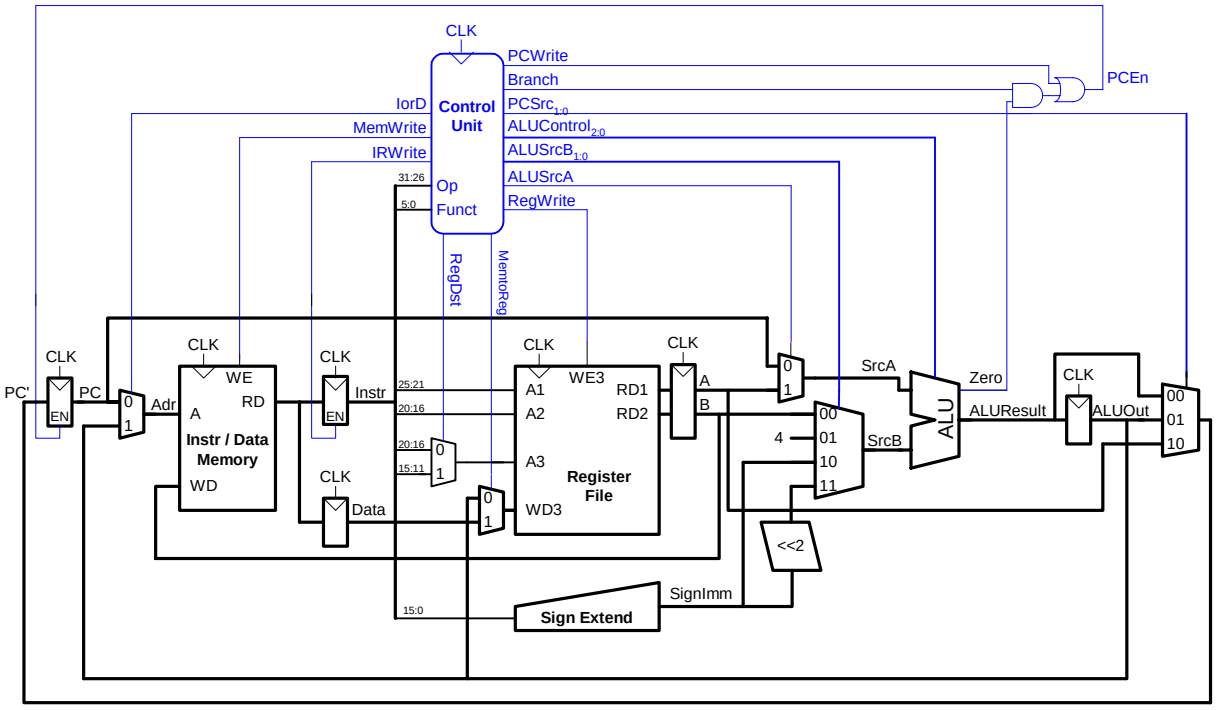
Next, we modify the datapath. Again, the only change is the buswidth of the ALUControl signal from 3 bits to 4 bits.

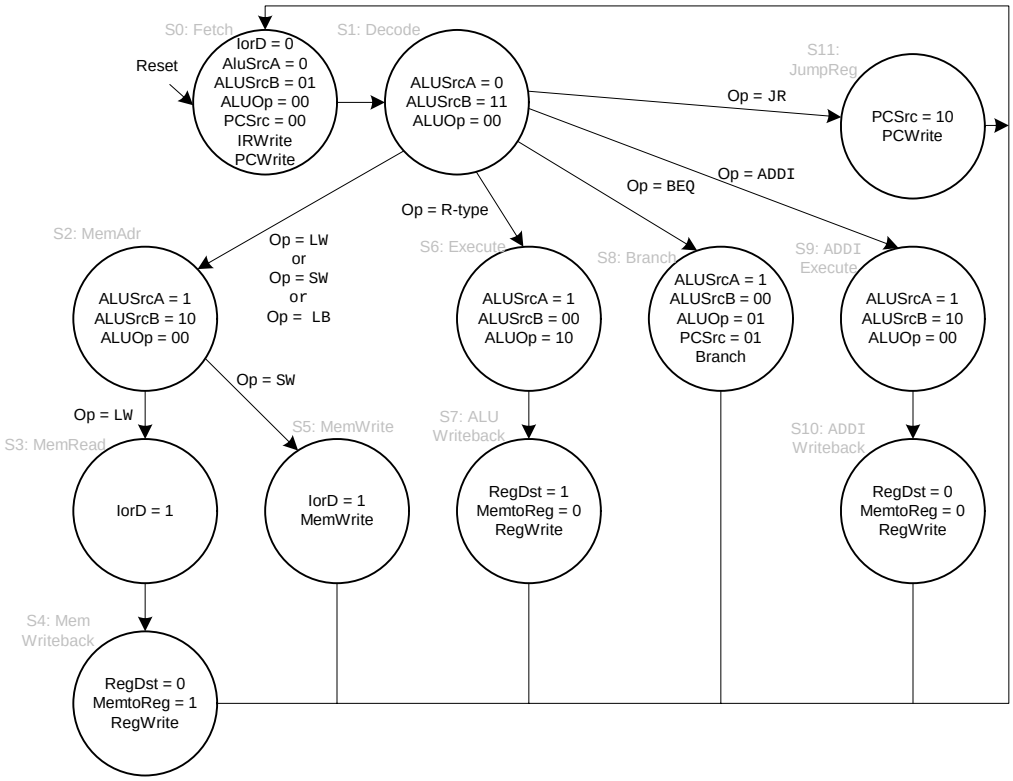


And finally, we modify the main control FSM.


S0: Fetch
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite
Reset
S1: Decode
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
Op = LW
or
Op = SW
Op = R-type
Op = BEQ
Op = XORI
S2: MemAdr
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
S6: Execute
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
S8: Branch
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCSrc = 1
Branch
S9: XORI
Execute
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 11
Op = LW
Op = SW
S3: MemRead
IorD = 1
S5: MemWrite
IorD = 1
MemWrite
S7: ALU
Writeback
RegDst = 1
MemtoReg = 0
RegWrite
S10: XORI
Writeback
RegDst = 0
MemtoReg = 0
RegWrite
S4: Mem
Writeback
RegDst = 0
MemtoReg = 1
RegWrite

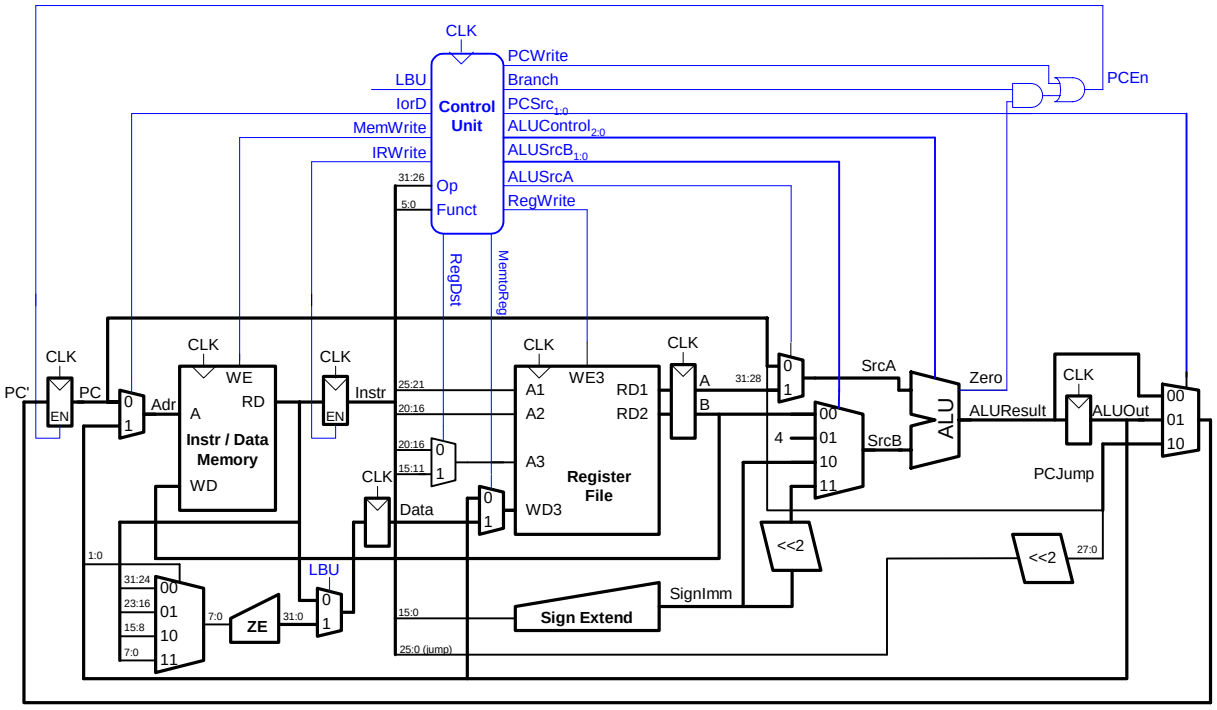
7.11 (d) `jr`


CLK
PCWrite
Branch
PCEn
IorD
MemWrite
IRWrite
Control Unit
PCSrc1:0
ALUControl2:0
ALUSrcB1:0
ALUSrcA
RegWrite
31:26
5:0
Op
Funct
RegDst
MemtoReg
PC'
PC
EN
Adr
WE
RD
A
Instr / Data Memory
WD
Instr
Data
25:21
20:16
15:11
A1
A2
A3
WE3
RD1
RD2
Register File
WD3
A
B
SrcA
SrcB
ALU
Zero
ALUResult
ALUOut
00
01
10
11
4
<<2
SignImm
15:0
Sign Extend

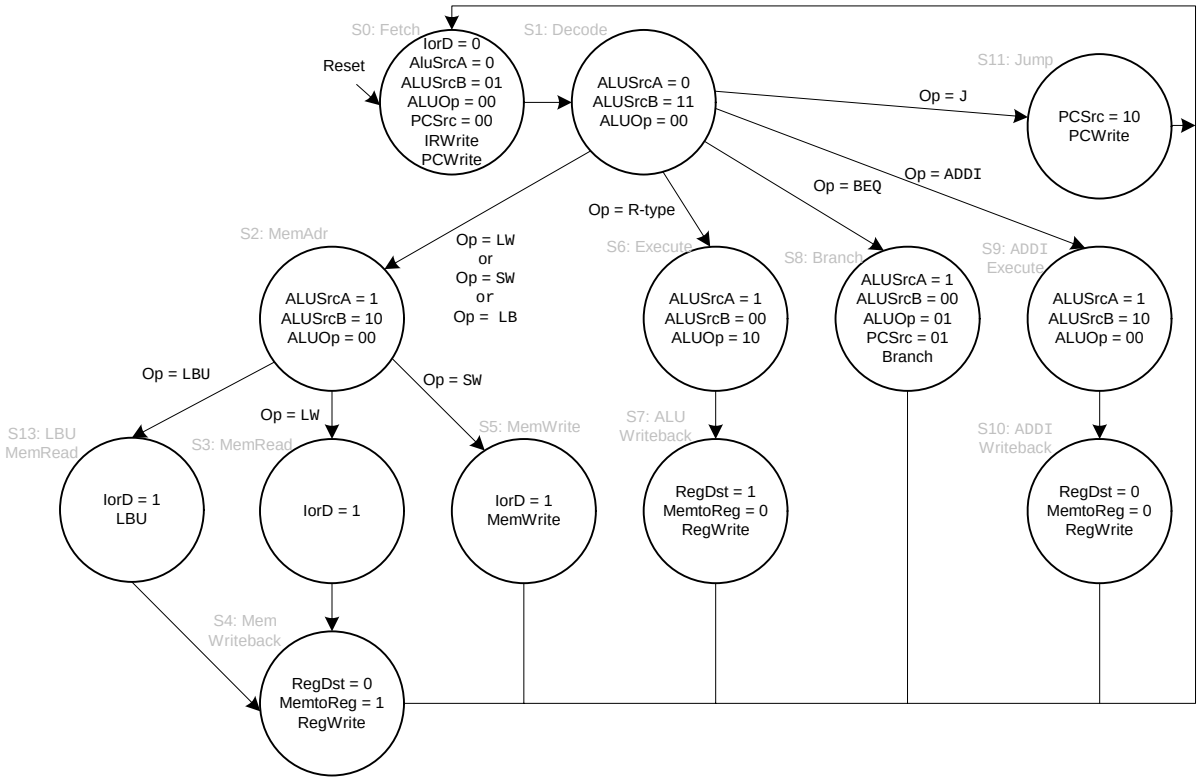

S0: Fetch
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 00
IRWrite
PCWrite
Reset
S1: Decode
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
S11: JumpReg
PCSrc = 10
PCWrite
Op = JR
Op = ADDI
Op = BEQ
Op = R-type
S2: MemAdr
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
Op = LW or Op = SW or Op = LB
S6: Execute
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
S8: Branch
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCSrc = 01
Branch
S9: ADDI Execute
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
Op = LW
Op = SW
S3: MemRead
IorD = 1
S5: MemWrite
IorD = 1
MemWrite
S7: ALU Writeback
RegDst = 1
MemtoReg = 0
RegWrite
S10: ADDI Writeback
RegDst = 0
MemtoReg = 0
RegWrite
S4: Mem Writeback
RegDst = 0
MemtoReg = 1
RegWrite

7.11 (e) bne

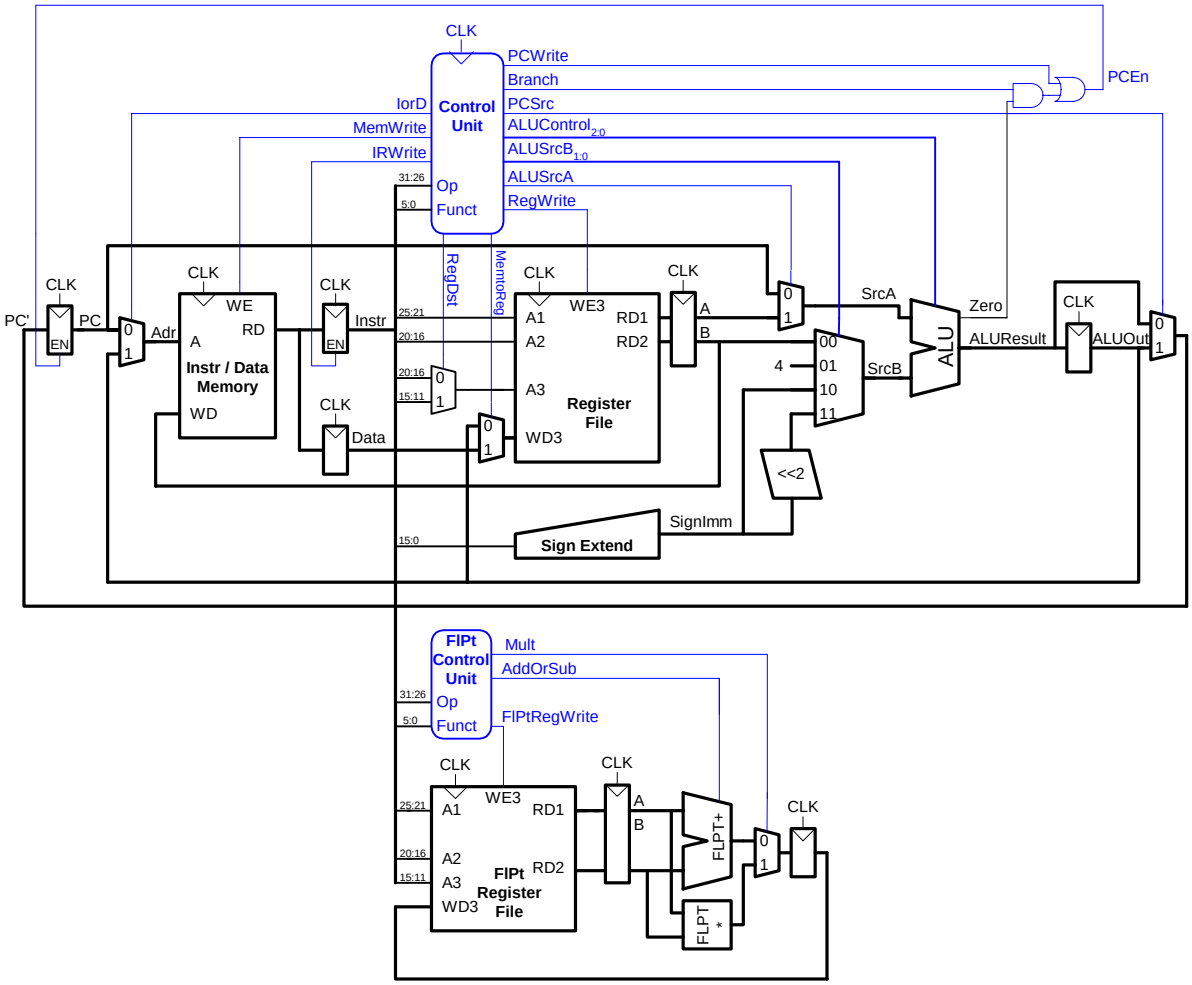
7.11 (f) `lbu`

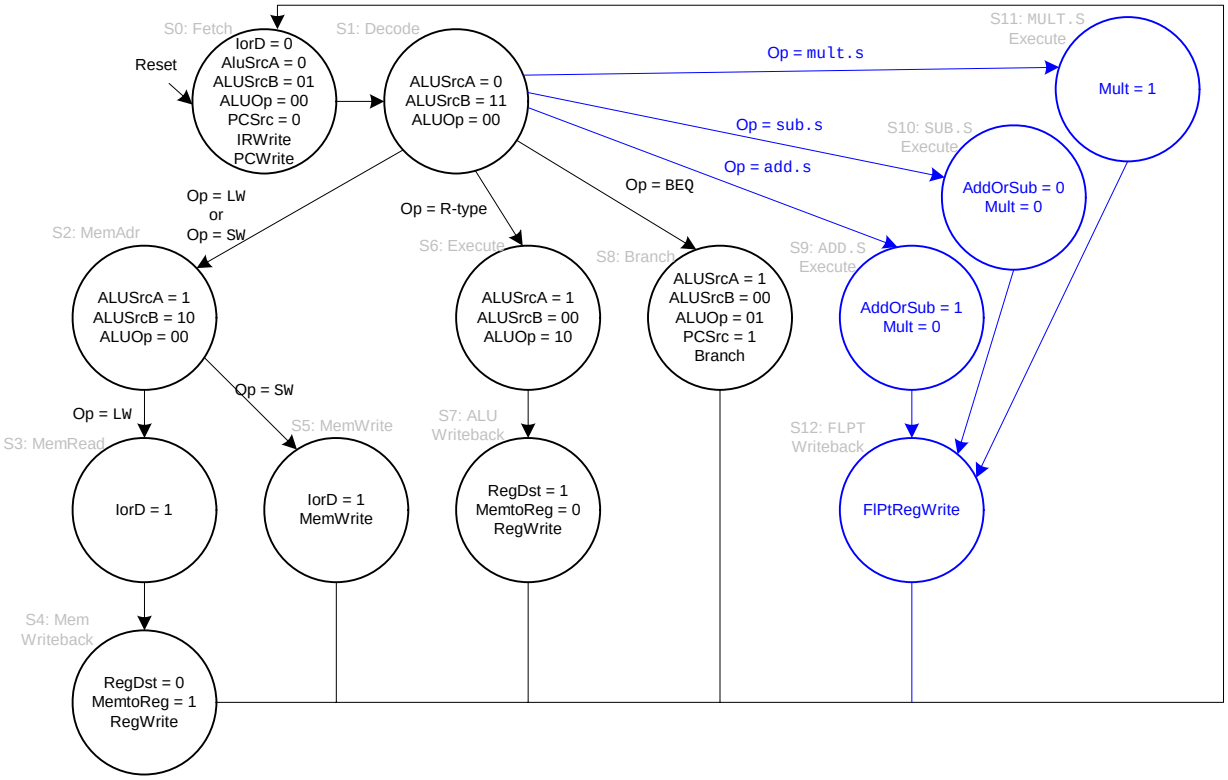


* The ZE unit is a zero extension unit.


S0: Fetch
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 00
IRWrite
PCWrite
Reset
S1: Decode
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
S11: Jump
PCSrc = 10
PCWrite
Op = J
Op = ADDI
Op = BEQ
Op = R-type
Op = LW
or
Op = SW
or
Op = LB
S2: MemAdr
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
S6: Execute
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
S8: Branch
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCSrc = 01
Branch
S9: ADDI
Execute
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
Op = LBU
Op = LW
Op = SW
S13: LBU
MemRead
IorD = 1
LBU
S3: MemRead
IorD = 1
S5: MemWrite
IorD = 1
MemWrite
S7: ALU
Writeback
RegDst = 1
MemtoReg = 0
RegWrite
S10: ADDI
Writeback
RegDst = 0
MemtoReg = 0
RegWrite
S4: Mem
Writeback
RegDst = 0
MemtoReg = 1
RegWrite

7.13


CLK
PCWrite
Branch
PCSrc
ALUControl2:0
ALUSrcB1:0
ALUSrcA
RegWrite
PCEn
IorD
MemWrite
IRWrite
Control Unit
Op
Funct
31:26
5:0
RegDst
MemtoReg
PC'
PC
EN
Adr
WE
RD
A
Instr / Data Memory
WD
Instr
Data
25:21
20:16
15:11
A1
A2
A3
WE3
WD3
RD1
RD2
Register File
A
B
SrcA
SrcB
4
00
01
10
11
ALU
Zero
ALUResult
ALUOut
<<2
SignImm
15:0
Sign Extend
FlPt Control Unit
Mult
AddOrSub
FlPtRegWrite
FlPt Register File
FLPT+
FLPT *



7.15

Your friend should work on the memory unit. It should have a delay of 225ps to equal the delay of the ALU plus multiplexer. The cycle time is now 300 ps.

7.17

Yes, Alyssa P. Hacker should switch to the slower but lower power register file for her multicycle processor design. Twice the delay still does not make the register file in the critical path (160 ps propagation delay for a read, and 140 + 25 = 165 ps for a write). The critical path is still the memory access, so using the reduced power register file does not affect the processor speed.

7.19

Average CPI = (0.25(6) + (0.52)(5) + (0.1 + 0.11)(4) + (0.02)(3) = 5

7.21

(4 × 3) + (4 + 3 + 4 + 4 + 3) × 10 + (4 + 3) = **199 clock cycles**

The number of instructions executed is 3 + (5 × 10) + 2 = 55. Thus, the CPI = 199 clock cycles / 55 instructions = **3.62 CPI**.

7.23

We modify the MIPS multicycle processor to implement all instructions from Exercise 7.11.

MIPS Top-Level Module

Verilog

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

  wire [31:0] readdata;

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

endmodule
```

VHDL

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

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

architecture synth of topmulti is
  component mips
  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 component;
  component dimem
    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 readdata: STD_LOGIC_VECTOR(31 downto 0);
begin
  -- instantiate processor and memories

  mips1: mips port map(clk, reset, dataadr, writedata, memwrite, readdata);
  mem1: dimem port map(clk, memwrite, dataadr, writedata, readdata);
end;
```

Modified MIPS Multicycle Processor

Verilog

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

  wire        zero, pcen, irwrite, regwrite,
              alusrca, iord, memtoreg, regdst;
  wire [2:0]  alusrcb;    // ORI, XORI
  wire [1:0]  pcsrc;
  wire [3:0]  alucontrol; // SRLV
  wire [5:0]  op, funct;
  wire        lbu;        // LBU

  controller c(clk, reset, op, funct, zero,
               pcen, memwrite, irwrite, regwrite,
               alusrca, iord, memtoreg, regdst,
              alusrcb, pcsrc, alucontrol, lbu);  // LBU
  datapath dp(clk, reset,
              pcen, irwrite, regwrite,
              alusrca, iord, memtoreg, regdst,
              alusrcb, pcsrc, alucontrol,
              lbu,                              // LBU
              op, funct, zero,
              adr, writedata, readdata);
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); --ORI, XORI
       pcsrc:              out STD_LOGIC_VECTOR(1 downto 0);
       alucontrol:         out STD_LOGIC_VECTOR(3 downto 0); --SRLV
       lbu:                out STD_LOGIC);  --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); --ORI, XORI
       pcsrc:             in  STD_LOGIC_VECTOR(1 downto 0);
       alucontrol:        in  STD_LOGIC_VECTOR(3 downto 0); --SRLV, XORI
       lbu:               in  STD_LOGIC; --LBU
       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);
       readdata:          in  STD_LOGIC_VECTOR(31 downto 0));
  end component;

  signal zero, pcen, irwrite, regwrite, alusrca, iord,
         memtoreg, regdst: STD_LOGIC;
  signal alusrcb: STD_LOGIC_VECTOR(2 downto 0);  --ORI, XORI
  signal pcsrc: STD_LOGIC_VECTOR(1 downto 0);
  signal alucontrol: STD_LOGIC_VECTOR(3 downto 0);  --SRLV
  signal op, funct: STD_LOGIC_VECTOR(5 downto 0);
  signal lbu: STD_LOGIC; --LBU
begin

  c: controller port map(clk, reset, op, funct, zero,
                         pcen, memwrite, irwrite, regwrite,
                         alusrca, iord, memtoreg, regdst,
                         alusrcb, pcsrc, alucontrol, lbu);  --LBU

  dp: datapath port map(clk, reset,
                        pcen, irwrite, regwrite,
                        alusrca, iord, memtoreg, regdst,
                        alusrcb, pcsrc, alucontrol,
                        lbu,  --LBU
                        op, funct, zero,
                        adr, writedata, readdata);
end;
```

Modified MIPS Multicycle Control

Verilog

```
module controller(input        clk, reset,
                  input  [5:0] op, funct,
                  input        zero,
                  output       pcen, memwrite,
                               irwrite, regwrite,
                  output       alusrca, iord,
                               memtoreg, regdst,
                output [2:0] alusrcb,    // ORI, XORI
                  output [1:0] pcsrc,
                  output [3:0] alucontrol,  // SRLV
                  output       lbu);        // LBU

  wire [2:0] aluop;  // XORI
  wire       branch, pcwrite;
  wire       bne;  // BNE

  // Main Decoder and ALU Decoder subunits.
  maindec md(clk, reset, op,
             pcwrite, memwrite, irwrite, regwrite,
            alusrca, branch, iord, memtoreg, regdst,
          alusrcb, pcsrc, aluop, bne, lbu); //BNE, LBU
  aludec  ad(funct, aluop, alucontrol);

  assign pcen = pcwrite | (branch & zero) |
                (bne & ~zero);  // BNE

endmodule
```

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:            out STD_LOGIC_VECTOR(2 downto 0); --ORI, XORI
       pcsrc:              out STD_LOGIC_VECTOR(1 downto 0);
       alucontrol:         out STD_LOGIC_VECTOR(3 downto 0); --SRLV
       lbu:                out STD_LOGIC);  --LBU
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:            out STD_LOGIC_VECTOR(2 downto 0); --ORI, XORI
         pcsrc:              out STD_LOGIC_VECTOR(1 downto 0);
         aluop:              out STD_LOGIC_VECTOR(2 downto 0); --XORI
         bne:                out STD_LOGIC; --BNE
         lbu:                out STD_LOGIC); --LBU
  end component;
  component aludec
    port(funct:      in  STD_LOGIC_VECTOR(5 downto 0);
         aluop:      in  STD_LOGIC_VECTOR(2 downto 0);  --XORI, ORI
         alucontrol: out STD_LOGIC_VECTOR(3 downto 0)); --XORI, SRLV
  end component;
  signal aluop: STD_LOGIC_VECTOR(2 downto 0); --XORI
  signal branch, pcwrite: STD_LOGIC;
  signal bne: STD_LOGIC; --BNE
begin

  md: maindec port map(clk, reset, op,
                       pcwrite, memwrite, irwrite, regwrite,
                       alusrca, branch, iord, memtoreg, regdst,
                       alusrcb, pcsrc, aluop, bne, lbu); --BNE, LBU
  ad: aludec port map(funct, aluop, alucontrol);

  pcen <= pcwrite or (branch and zero) or (bne and (not zero));
end;
```

Modified MIPS Multicycle Main Decoder FSM

Verilog

```
module maindec(input        clk, reset,
               input  [5:0] op,
               output       pcwrite, memwrite,
                            irwrite, regwrite,
               output       alusrca, branch,
                            iord, memtoreg, regdst,
               output [2:0] alusrcb, // ORI, XORI
               output [1:0] pcsrc,
               output [2:0] aluop,   // XORI
               output       bne,     // BNE
               output       lbu);    // LBU

  parameter   FETCH   = 5'b00000;   // State 0
  parameter   DECODE  = 5'b00001;   // State 1
  parameter   MEMADR  = 5'b00010;   // State 2
  parameter   MEMRD   = 5'b00011;   // State 3
  parameter   MEMWB   = 5'b00100;   // State 5
  parameter   MEMWR   = 5'b00101;   // State 5
  parameter   RTYPEEX = 5'b00110;   // State 6
  parameter   RTYPEWB = 5'b00111;   // State 7
  parameter   BEQEX   = 5'b01000;   // State 8
  parameter   ADDIEX  = 5'b01001;   // State 9
  parameter   ADDIWB  = 5'b01010;   // state a
  parameter   JEX     = 5'b01011;   // State b
  parameter   ORIEX   = 5'b01100;   // State c // ORI
  parameter   ORIWB   = 5'b01101;   // State d // ORI
  parameter   XORIEX  = 5'b01110;   // State e // XORI
  parameter   XORIWB  = 5'b01111;   // State f // XORI
  parameter   BNEEX   = 5'b10000;   // State 10 // BNE
  parameter   LBURD   = 5'b10001;   // State 11 // LBU

  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
  parameter   ORI     = 6'b001101; // Opcode for ori
  parameter   XORI    = 6'b001110; // Opcode for xori
  parameter   BNE     = 6'b000101; // Opcode for bne
  parameter   LBU     = 6'b100100; // Opcode for lbu

  reg [4:0]  state, nextstate;
  reg [18:0] controls;  // ORI, XORI, BNE, LBU

  // state register
  always @(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;
       regdst:             out STD_LOGIC;
       alusrcb:            out STD_LOGIC_VECTOR(2 downto 0); --ORI, XORI
       pcsrc:              out STD_LOGIC_VECTOR(1 downto 0);
       aluop:              out STD_LOGIC_VECTOR(2 downto 0); --XORI
       bne:                out STD_LOGIC; --BNE
       lbu:                out STD_LOGIC); --LBU
end;

architecture behave of maindec is
  type statetype is (FETCH, DECODE, MEMADR, MEMRD, MEMWB, MEMWR,
                     RTYPEEX, RTYPEWB, BEQEX, ADDIEX, ADDIWB, JEX,
                     ORIEX, ORIWB, XORIEX, XORIWB, BNEEX, LBURD);
                     --ORI, XORI, BNE, LBU
  signal state, nextstate: statetype;
  signal controls: STD_LOGIC_VECTOR(18 downto 0); --ORI, XORI, BNE, LBU
begin
  --state register
  process(clk, reset) begin
    if reset = '1' then state <= FETCH;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

  -- next state logic
  process (state, op) begin
    case state is
      when FETCH  =>          nextstate <= DECODE;
      when DECODE =>
          case op is
            when "100011" =>  nextstate <= MEMADR;
            when "101011" =>  nextstate <= MEMADR;
            when "100100" =>  nextstate <= MEMADR; --LBU
            when "000000" =>  nextstate <= RTYPEEX;
            when "000100" =>  nextstate <= BEQEX;
            when "001000" =>  nextstate <= ADDIEX;
            when "000010" =>  nextstate <= JEX;
            when "001101" =>  nextstate <= ORIEX;  --ORI
            when "001110" =>  nextstate <= XORIEX; --XORI
            when "000101" =>  nextstate <= BNEEX;  --BNE
            when others    => nextstate <= FETCH; -- should never happen
          end case;
      when MEMADR =>
          case op is
            when "100011" =>  nextstate <= MEMRD;
            when "101011" =>  nextstate <= MEMWR;
            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 ORIEX =>           nextstate <= ORIWB;  --ORI
      when ORIWB =>           nextstate <= FETCH;  --ORI
      when XORIEX =>          nextstate <= XORIWB; --XORI
      when XORIWB =>          nextstate <= FETCH;  --XORI
      when BNEEX =>           nextstate <= FETCH;  --BNE
      when LBURD =>           nextstate <= MEMWB;
      when others =>          nextstate <= FETCH; -- should never happen
    end case;
  end process;
```

Verilog

```
    // next state logic
  always @( * )
    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;
                       // should never happen
               endcase
      MEMADR:  case(op)
                 LW:      nextstate <= MEMRD;
                 SW:      nextstate <= MEMWR;
                 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;
      default: nextstate <= FETCH;
           // should never happen
    endcase

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

    always @( * )
    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'b0001_00010_0000_00;
      MEMWR:   controls <= 15'b0100_00100_0000_00;
      RTYPEEX: controls <= 15'b0000_10000_0000_10;
      RTYPEWB: controls <= 15'b0001_00001_0000_00;
      BEQEX:   controls <= 15'b0000_11000_0001_01;
      ADDIEX:  controls <= 15'b0000_10000_1000_00;
      ADDIWB:  controls <= 15'b0001_00000_0000_00;
     JEX:    controls <= 15'b1000_00000_0010_00;
      default: controls <= 15'b0000_xxxxx_xxxx_xx;
    endcase
endmodule
```

VHDL

```
  -- output logic
  process(state) begin
    case state is
      when FETCH   => controls <= "101000000010000";
      when DECODE  => controls <= "000000000110000";
      when MEMADR  => controls <= "000010000100000";
      when MEMRD   => controls <= "000000100000000";
      when MEMWB   => controls <= "000100010000000";
      when MEMWR   => controls <= "010000100000000";
      when RTYPEEX => controls <= "000010000000010";
      when RTYPEWB => controls <= "000100001000000";
      when BEQEX   => controls <= "000011000000101";
      when ADDIEX  => controls <= "000010000100000";
      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);
  alusrca  <= controls(10);
  branch   <= controls(9);
  iord     <= controls(8);
  memtoreg <= controls(7);
  regdst   <= controls(6);
  alusrcb  <= controls(5 downto 4);
  pcsrc    <= controls(3 downto 2);
  aluop    <= controls(1 downto 0);

end;
```

(continued on next page)

(continued from previous page)

Verilog

```
// next state logic
always @( * )
  case(state)
    FETCH:   nextstate <= DECODE;
    DECODE:  case(op)
               LW:      nextstate <= MEMADR;
               SW:      nextstate <= MEMADR;
               LBU:     nextstate <= MEMADR; // LBU
               RTYPE:   nextstate <= RTYPEEX;
               BEQ:     nextstate <= BEQEX;
               ADDI:    nextstate <= ADDIEX;
               J:       nextstate <= JEX;
               ORI:     nextstate <= ORIEX;  // ORI
               XORI:    nextstate <= XORIEX; // XORI
               BNE:     nextstate <= BNEEX;  // BNE
               default: nextstate <= FETCH;
                    // should never happen
             endcase
    MEMADR:  case(op)
               LW:      nextstate <= MEMRD;
               SW:      nextstate <= MEMWR;
               LBU:     nextstate <= LBURD; // LBU
               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;
    ORIEX:   nextstate <= ORIWB;  // ORI
    ORIWB:   nextstate <= FETCH;  // ORI
    XORIEX:  nextstate <= XORIWB; // XORI
    XORIWB:  nextstate <= FETCH;  // XORI
    BNEEX:   nextstate <= FETCH;  // BNE
    LBURD:   nextstate <= MEMWB;
    default: nextstate <= FETCH;
         // should never happen
  endcase
```

(continued on next page)

VHDL

```
-- output logic
process(state) begin
  case state is
    when FETCH =>   controls <= "1010000000001000000";
    when DECODE =>  controls <= "0000000000011000000";
    when MEMADR =>  controls <= "0000100000010000000";
    when MEMRD =>   controls <= "0000001000000000000";
    when MEMWB =>   controls <= "0001000100000000000";
    when MEMWR =>   controls <= "0100001000000000000";
    when RTYPEEX => controls <= "0000100000000000100";
    when RTYPEWB => controls <= "0001000010000000000";
    when BEQEX =>   controls <= "0000110000000010010";
    when ADDIEX =>  controls <= "0000100000010000000";
    when ADDIWB =>  controls <= "0001000000000000000";
    when JEX =>     controls <= "1000000000000100000";
    when ORIEX =>   controls <= "0000100000100000110"; --ORI
    when ORIWB =>   controls <= "0001000000000000000"; --ORI
    when XORIEX =>  controls <= "0000100000100001000"; --XORI
    when XORIWB =>  controls <= "0001000000000000000"; --XORI
    when BNEEX =>   controls <= "0000100001000010010"; --BNE
    when LBURD =>   controls <= "0000001000000000001"; --LBU
    when others =>  controls <= "0000---------------"; --illegal op
  end case;
end process;

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);
pcsrc    <= controls(5 downto 4);
aluop    <= controls(3 downto 1);
lbu      <= controls(0);
end;
```

(continued from previous page)

Verilog

```
  // output logic
  assign {pcwrite, memwrite, irwrite, regwrite,
          alusrca, branch, iord, memtoreg, regdst,
          bne, // BNE
          alusrcb, pcsrc,
          aluop,  // extend aluop to 3 bits // XORI
          lbu} = controls;  // LBU

  always @( * )
    case(state)
     FETCH:   controls <= 19'b1010_000000_00100_000_0;
     DECODE:  controls <= 19'b0000_000000_01100_000_0;
     MEMADR:  controls <= 19'b0000_100000_01000_000_0;
     MEMRD:   controls <= 19'b0000_001000_00000_000_0;
     MEMWB:   controls <= 19'b0001_000100_00000_000_0;
     MEMWR:   controls <= 19'b0100_001000_00000_000_0;
     RTYPEEX: controls <= 19'b0000_100000_00000_010_0;
     RTYPEWB: controls <= 19'b0001_000010_00000_000_0;
     BEQEX:   controls <= 19'b0000_110000_00001_001_0;
     ADDIEX:  controls <= 19'b0000_100000_01000_000_0;
     ADDIWB:  controls <= 19'b0001_000000_00000_000_0;
    JEX:   controls <= 19'b1000_000000_00010_000_0;
     ORIEX:   controls <= 19'b0000_100000_10000_011_0;
                          // ORI
     ORIWB:   controls <= 19'b0001_000000_00000_000_0;
                          // ORI
     XORIEX:  controls <= 19'b0000_100000_10000_100_0;
                          // XORI
     XORIWB:  controls <= 19'b0001_000000_00000_000_0;
                          // XORI
     BNEEX:   controls <= 19'b0000_100001_00001_001_0;
                          // BNE
     LBURD:   controls <= 19'b0000_001000_00000_000_1;
                          // LBU
     default: controls <= 19'b0000_xxxxxx_xxxxx_xxx_x;
            // should never happen
     endcase
endmodule
```

Modified MIPS Multicycle ALU Decoder

Verilog

```
module aludec(input      [5:0] funct,
              input      [2:0] aluop,       // XORI
            output reg [3:0] alucontrol); // XORI, SRLV

    always @( * )
    case(aluop)
      3'b000: alucontrol <= 4'b0010;  // add
      3'b001: alucontrol <= 4'b1010;  // sub
      3'b011: alucontrol <= 4'b0001;  // or  // ORI
      3'b100: alucontrol <= 4'b0101;  // xor // XORI
      3'b010: 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'b000110: alucontrol <= 4'b0100; // SRLV
          default:   alucontrol <= 4'bxxxx; // ???
        endcase
      default: alucontrol <= 4'bxxxx; // ???
    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(2 downto 0);  --XORI, ORI
       alucontrol: out STD_LOGIC_VECTOR(3 downto 0)); --XORI, SRLV
end;

architecture behave of aludec is
begin
  process(aluop, funct) begin
    case aluop is
      when "000" => alucontrol <= "0010"; -- add (for lb/sb/addi)
      when "001" => alucontrol <= "1010"; -- sub (for beq)
      when "011" => alucontrol <= "0001"; -- or  --ORI
      when "100" => alucontrol <= "0101"; -- xor --XORI
      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 "000110" => alucontrol <= "0100"; -- srlv
                       when others   => alucontrol <= "----"; -- ???
                     end case;
    end case;
  end process;
end;
```

Modified MIPS Multicycle Datapath

Verilog

```
module datapath(input         clk, reset,
                input         pcen, irwrite, regwrite,
                input         alusrca, iord,
                              memtoreg, regdst,
                input  [2:0]  alusrcb,    // ORI, XORI
                input  [1:0]  pcsrc,
                input  [3:0]  alucontrol,  // SRLV
                input         lbu,         // LBU
                output [5:0]  op, funct,
                output        zero,
                output [31:0] adr, writedata,
                input  [31:0] readdata);

  // Internal signals of the datapath module

  wire [4:0]  writereg;
  wire [31:0] pcnext, pc;
  wire [31:0] instr, data, srca, srcb;
  wire [31:0] a;
  wire [31:0] aluresult, aluout;
  wire [31:0] signimm;   // the sign-extended imm
  wire [31:0] zeroimm;   // the zero-extended imm
                         // ORI, XORI
 wire [31:0] signimmsh; // the sign-extended imm << 2
  wire [31:0] wd3, rd1, rd2;
  wire [31:0] memdata, membyteext; // LBU
  wire [7:0]  membyte; // LBU

  // 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);

  // changes for 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   lbe(membyte, membyteext);
  mux2    #(32) datamux(readdata, membyteext,
                        lbu, memdata);
```

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:           in  STD_LOGIC_VECTOR(2 downto 0); --ORI, XORI
       pcsrc:             in  STD_LOGIC_VECTOR(1 downto 0);
       alucontrol:        in  STD_LOGIC_VECTOR(3 downto 0); --SRLV, XORI
       lbu:               in  STD_LOGIC; --LBU
       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);
       readdata:          in  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(3 downto 0); --XORI, SRLV
         shamt: in     STD_LOGIC_VECTOR(4 downto 0); -- SRLV
         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;
         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;
  component mux5 is 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;
```

(continued on next page)

(continued from previous page)

Verilog

```
  mux2    #(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);
  zeroext      ze(instr[15:0], zeroimm);  // ORI, XORI
  sl2           immsh(signimm, signimmsh);
  flopr   #(32) areg(clk, reset, rd1, a);
  flopr   #(32) breg(clk, reset, rd2, writedata);
  mux2    #(32) srcamux(pc, a, alusrca, srca);
  mux5    #(32) srcbmux(writedata, 32'b100,
                        signimm, signimmsh,
                        zeroimm,  // ORI, XORI
                        alusrcb, srcb);

  // SRLV
  alu           alu(srca, srcb, alucontrol, rd1[4:0],
                    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

```
  component zeroext
    port(a: in  STD_LOGIC_VECTOR(15 downto 0);
         y: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component zeroext8_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,
         aluresult, aluout, signimm, signimmsh, wd3, rd1, rd2, pcjump:
                    STD_LOGIC_VECTOR(31 downto 0);
 signal zeroimm: STD_LOGIC_VECTOR(31 downto 0); -- zero-extended immediate
--ORI, XORI
  signal memdata, membyteext: STD_LOGIC_VECTOR(31 downto 0); --LBU
  signal membyte: STD_LOGIC_VECTOR(7 downto 0); --LBU
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, irwrite,
                                             readdata, instr);

  -- hardware for LBU
  datareg: flopr generic map(32) port map(clk, reset, memdata, data); --LBU
  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); --LBU
  lbe: zeroext8_32 port map(membyte, membyteext); --LBU
  datamux: mux2 generic map(32) port map(readdata, membyteext, lbu,
                                         memdata); --LBU

  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); --ORI, XORI

  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);

  srcbmux: mux5 generic map(32) port map(writedata,
                "00000000000000000000000000000100",
                signimm, signimmsh,
                zeroimm,  --ORI, XORI
                alusrcb, srcb);
  alu32: alu port map(srca, srcb, alucontrol, rd1(4 downto 0),  --SRLV
                      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, pcsrc, pcnext);
end;
```

The following describes the building blocks that are used in the MIPS multicycle processor that are not found in Section 7.6.2.

MIPS Multicycle Modified ALU

Verilog

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

  wire [31:0] S, Bout;

  assign Bout = F[3] ? ~B : B;
  assign S = A + Bout + F[3];  // SRLV

  always @ ( * )
    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);  // SRLV
      3'b101: Y <= A ^ Bout;  // XORI
    endcase

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

VHDL

```
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(3 downto 0); --XORI, SRLV
       shamt: in     STD_LOGIC_VECTOR(4 downto 0); -- SRLV
       Y:     buffer 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; --SRLV
  S <= A + Bout + F(3);  --SRLV

  -- alu function
  process (F(1 downto 0), A, Bout, S) begin
    case F(2 downto 0) is  --SRLV, XORI
      when "000" => Y <= A and Bout;
      when "001" => Y <= A or Bout;
      when "010" => Y <= S;
      when "011" => Y <= ("00000000000000000000000000000000" & S(31));
      when "100" => Y <=          --SRLV
        to_stdlogicvector(to_bitvector(Bout) srl conv_integer(shamt));
      when "101" => Y <= A xor Bout;  --XORI
      when others => Y <= "00000000000000000000000000000000";
    end case;
  end process;

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

end;
```

Additional MIPS Multicycle Building Blocks

Verilog

```
// mux5 is needed for ORI, XORI
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 @( * )
      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 ORI, XORI
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  [7:0] a,
               output [31:0] y);

  assign y = {24'b0, a};
endmodule
```

VHDL

```
-- mux5 is needed for ORI, XORI
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux5 is -- four-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(s, d0, d1, d2, d3, d4) 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;

--zeroext is needed for ORI, XORI
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity zeroext is -- zero extender
  port(a: in  STD_LOGIC_VECTOR(15 downto 0);
       y: out STD_LOGIC_VECTOR(31 downto 0));
end;

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

-- zeroext8_32 is needed for LBU
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity zeroext8_32 is -- zero extender
  port(a: in  STD_LOGIC_VECTOR(7 downto 0);
       y: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of zeroext8_32 is
begin
  y <= X"000000" & a;
end;
```

We modify the test code to include the extended instructions.

```
# mipstest.asm
# Sarah_Harris@hmc.edu 20 February 2007
#
# Test the MIPS multicycle processor.
#  add, sub, and, or, slt, addi, lw, sw, beq, j
#  extended instructions: srlv, ori, xori, jr, bne, lbu

# If successful, it should write the value 7 to address 84

#       Assembly                Description           Address Machine
main:   addi $2, $0, 5          # initialize $2 = 5        0    20020005
        ori  $3, $2, 0xFEFE     # $3 = 0xFEFF              4    3443fefe
        srlv $2, $3, $2         # $2 = $3 >> $2 = 0x7F7    8    00431006
        j    forward                                     c    08000006
        addi $3, $0, 14         # not executed             10   2263000e
back:   beq  $2, $2, here       # should be taken          14   10420003
forward:addi $3, $3, 12         # $3 <= 0xFF0B             18   2063000c
        addi $31, $0, 0x14      # $31 ($ra) <= 0x14        1c   201f0014
        jr   $ra                # return to address 0x14   20   08000005
here:   addi $7, $3, -9         # $7 <= 0xFF02             24   2067fff7
        xori $6, $7, 0xFF07     # $6 <= 5                  28   38e6ff07
        bne  $3, $7, around     # should be taken          2c   14670003
        slt  $4, $7, $3         # not executed             30   00e6302a
        beq  $4, $0, around     # not executed             34   10800001
        addi $5, $0, 0          # not executed             38   20050000
around: sw   $7, 95($6)         # [95+5] = [100] = 0xFF02  3c   acc7005f
        sw   $2, 104($0)        # [104] = 0x7F7            40   ac020068
        lw   $2, 100($0)        # $2 = [100] = 0xFF02      44   8c020064
        lbu  $3, 107($0)        # $3 = 0x000000F7          48   9003006b
        j    end                # should be taken          4c   08000015
        addi $2, $0, 1          # not executed             50   20020001
end:    sub  $8, $2, $3         # $8 = 0xFE0B              54   00434022
        sw   $8, 108($0)        # [108] = 0xFE0B           58   ac08006c
```

FIGURE 7.5 Assembly and machine code for multicycle MIPS test program

MIPS Multicycle Modified Testbench

Verilog

```
module testbench();

  reg         clk;
  reg         reset;

  wire [31:0] writedata, dataadr;
  wire memwrite;

  // keep track of execution status
  reg  [31:0] cycle;
  reg         succeed;

  // instantiate device to be tested
   topmulti dut(clk, reset, writedata, dataadr, mem-
write);

  // initialize test
  initial
    begin
      reset <= 1; # 12; reset <= 0;
 cycle <= 1;
 succeed <= 0;
    end

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

  // check results
  always@(negedge clk)
    begin
      if(memwrite & dataadr == 108) begin
        if(writedata == 65035)
          $display("Simulation succeeded");
        else begin
          $display("Simulation failed");
          $stop;
        end
      end
    end
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.STD_LOGIC_UNSIGNED.all;
entity testbench is
end;

architecture test of testbench is
  component topmulti
    port(clk, reset:           in    STD_LOGIC;
         writedata, dataadr:   inout STD_LOGIC_VECTOR(31 downto 0);
         memwrite:             inout STD_LOGIC);
  end component;
  signal readdata, writedata, dataadr:
           STD_LOGIC_VECTOR(31 downto 0);
  signal clk, reset, memwrite:  STD_LOGIC;
begin

  -- instantiate device to be tested
  dut: topmulti 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 65035 gets written to address 108
  -- at end of program
  process (clk) begin
    if (clk'event and clk = '0' and memwrite = '1' and
        conv_integer(dataadr) = 108) then
      if (conv_integer(writedata) = 65035) then
        report "Simulation succeeded"
          severity failure;
      else
        report "Simulation failed"
          severity failure;
      end if;
    end if;
  end process;
end;
```

7.25

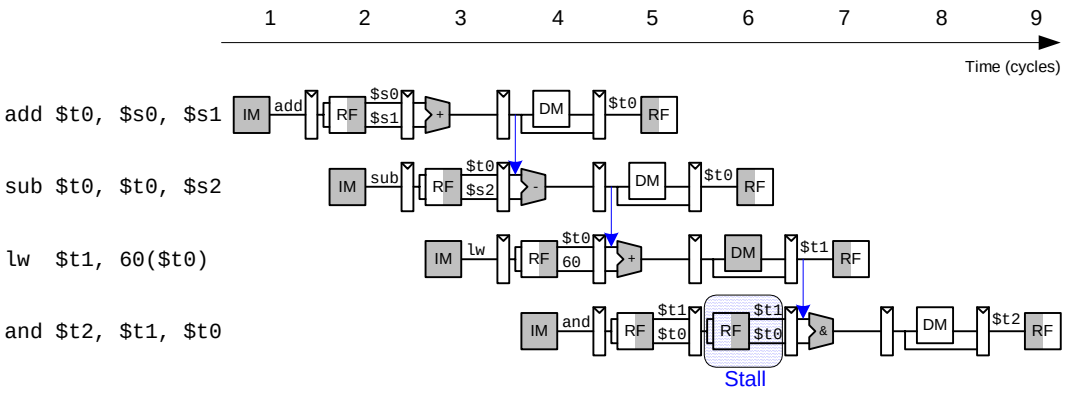


FIGURE 7.6 Abstract pipeline for Exercise 7.25

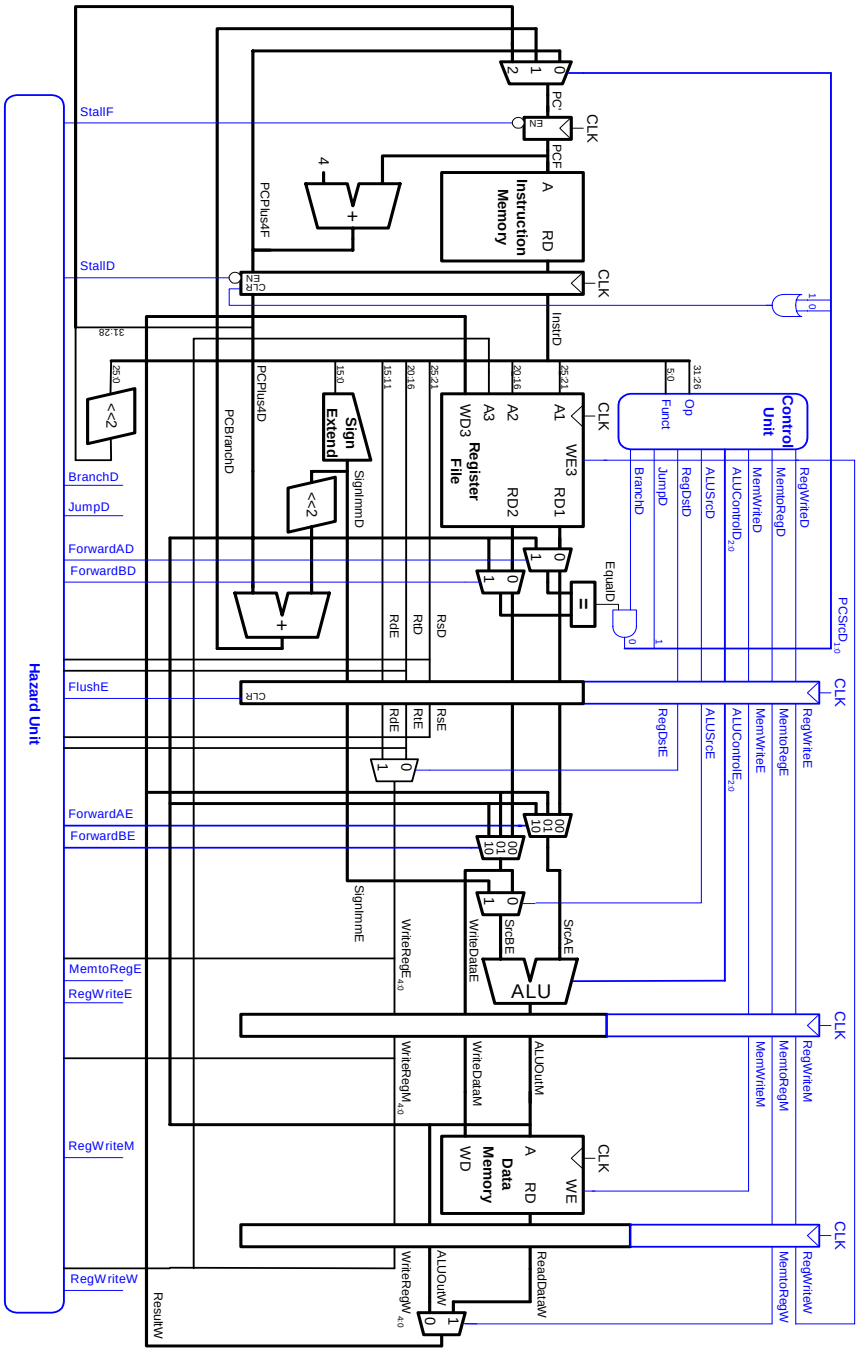
7.27

It takes 3 + 10(5) + 2 = **55 clock cycles** to issue all the instructions.

With no branch or jump penalty, it would take 4 cycles to fill up the pipeline, and then 55 clock cycles to complete execution, so the total execution time is: 59 clock cycles. Thus, the CPI is 59 clock cycles / 55 instructions = **1.07.**

If the jump target address (JTA) cannot be determined until the second stage, each jump would effectively take 2 cycles to execute (the instruction fetched after each jump would need to be discarded). Thus, it would take 3 + 10(6) + 2 = 65 clock cycles to issue all the instructions and 4 + 65 = 69 clock cycles to complete execution. Thus, CPI is 69/55 = 1.25.

7.29


Hazard Unit
StallF
StallD
FlushE
ForwardAD
ForwardBD
ForwardAE
ForwardBE
MemtoRegE
RegWriteE
RegWriteM
RegWriteW
BranchD
JumpD
PCPlus4F
PCPlus4D
PCBranchD
InstrD
SignImmD
SignImmE
Instruction Memory
Register File
Sign Extend
Control Unit
Data Memory
ALU
EqualD
PCSrcD
RegWriteD
MemtoRegD
MemWriteD
ALUControlD
ALUSrcD
RegDstD
JumpD
BranchD
RegWriteE
MemtoRegE
MemWriteE
ALUControlE
ALUSrcE
RegDstE
RegWriteM
MemtoRegM
MemWriteM
RegWriteW
MemtoRegW
WriteRegE
WriteDataE
SrcAE
SrcBE
ALUOutM
WriteDataM
WriteRegM
ReadDataW
ALUOutW
WriteRegW
ResultW
RsD
RtD
RdD
RsE
RtE
RdE
CLK
CLR
EN

instruction	opcode	regwrite	regdst	alusrc	branch	memwrite	memtoreg	aluop	jump
R-type	000000	1	1	0	0	0	0	10	0
`lw`	100011	1	0	1	0	0	1	00	0
`sw`	101011	0	X	1	0	1	X	00	0
`beq`	000100	0	X	0	1	0	X	01	0
`j`	000010	0	X	X	0	0	X	XX	1

TABLE 7.13 Main decoder truth table enhanced to support `j`

We must also write new equations for the flush signal, *FlushE*.

```
FlushE = lwstall OR branchstall OR JumpD
```

7.31

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

T_{c3} = max(30 + 250 + 20, 2(80 + 25 + 40 + 15 + 25 + 20), 30 + 25 + 25 + 200 + 20, 30 + 220 + 20, 2(30 + 25 + 70)) = 410 ps. The next slowest stages (the Fetch and Execute stages) are 300 ps each, 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 (80 ps). We need to reduce the cycle time by 410-300 = 110 ps. Thus, we need to reduce the register file read delay by 55 ps to (80 - 55) = **25 ps**.

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

7.33 MIPS Pipelined Processor

Verilog

```
// pipelined MIPS processor
module mips(input         clk, reset,
            output [31:0] pcF,
            input  [31:0] instrF,
            output        memwriteM,
            output [31:0] aluoutM, writedataM,
            input  [31:0] readdataM);

  wire [5:0]  opD, functD;
  wire        regdstE, alusrcE,
              pcsrcD,
              memtoregE, memtoregM, memtoregW,
              regwriteE, regwriteM, regwriteW;
  wire [2:0]  alucontrolE;
  wire        flushE, equalD;

  controller c(clk, reset, opD, functD, flushE,
               equalD,memtoregE, memtoregM,
               memtoregW, memwriteM, pcsrcD,
               branchD,alusrcE, regdstE, regwriteE,
               regwriteM, regwriteW, jumpD,
               alucontrolE);
  datapath dp(clk, reset, memtoregE, memtoregM,
              memtoregW, pcsrcD, branchD,
              alusrcE, regdstE, regwriteE,
              regwriteM, regwriteW, jumpD,
              alucontrolE,
              equalD, pcF, instrF,
              aluoutM, writedataM, readdataM,
              opD, functD, flushE);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mips is -- pipelined MIPS processor
  port(clk, reset:          in    STD_LOGIC;
       pcF:                 inout STD_LOGIC_VECTOR(31 downto 0);
       instrF:              in    STD_LOGIC_VECTOR(31 downto 0);
       memwriteM:           out   STD_LOGIC;
       aluoutM, writedataM: inout STD_LOGIC_VECTOR(31 downto 0);
       readdataM:           in    STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of mips is
  component controller
    port(clk, reset:              in  STD_LOGIC;
         opD, functD:             in  STD_LOGIC_VECTOR(5 downto 0);
         flushE, equalD:          in  STD_LOGIC;
         branchD:                 inout STD_LOGIC;
         memtoregE, memtoregM:    inout STD_LOGIC;
         memtoregW, memwriteM:    out STD_LOGIC;
         pcsrcD, alusrcE:         out STD_LOGIC;
         regdstE:                 out STD_LOGIC;
         regwriteE:               inout STD_LOGIC;
         regwriteM, regwriteW:    inout STD_LOGIC;
         jumpD:                   out STD_LOGIC;
         alucontrolE:             out STD_LOGIC_VECTOR(2 downto 0));
  end component;
  component datapath
    port(clk, reset:                      in  STD_LOGIC;
         memtoregE, memtoregM, memtoregW: in STD_LOGIC;
         pcsrcD, branchD:                 in  STD_LOGIC;
         alusrcE, regdstE:                in  STD_LOGIC;
         regwriteE, regwriteM, regwriteW: in STD_LOGIC;
         jumpD:                           in  STD_LOGIC;
         alucontrolE:                     in  STD_LOGIC_VECTOR(2 downto 0);
         equalD:                          out STD_LOGIC;
         pcF:                             inout STD_LOGIC_VECTOR(31 downto 0);
         instrF:                          in  STD_LOGIC_VECTOR(31 downto 0);
         aluoutM, writedataM:             inout STD_LOGIC_VECTOR(31 downto 0);
         readdataM:                       in  STD_LOGIC_VECTOR(31 downto 0);
         opD, functD:                     out STD_LOGIC_VECTOR(5 downto 0);
         flushE:                          inout STD_LOGIC);
  end component;

  signal opD, functD: STD_LOGIC_VECTOR(5 downto 0);
  signal regdstE, alusrcE, pcsrcD, memtoregE, memtoregM,
         memtoregW, regwriteE, regwriteM, regwriteW,
         branchD, jumpD:
         STD_LOGIC;
  signal alucontrolE: STD_LOGIC_VECTOR(2 downto 0);
  signal flushE, equalD: STD_LOGIC;
begin
  c: controller port map(clk, reset, opD, functD, flushE, equalD, branchD,
                memtoregE, memtoregM, memtoregW, memwriteM, pcsrcD,
                alusrcE, regdstE, regwriteE, regwriteM, regwriteW, jumpD,
                alucontrolE);
  dp: datapath port map(clk, reset, memtoregE, memtoregM, memtoregW,
                pcsrcD, branchD,
                alusrcE, regdstE, regwriteE, regwriteM, regwriteW, jumpD,
                alucontrolE,
                equalD, pcF, instrF,
                aluoutM, writedataM, readdataM,
                opD, functD, flushE);
end;
```

MIPS Pipelined Control

Verilog

```
module controller(input        clk, reset,
                  input  [5:0] opD, functD,
                  input        flushE, equalD,
                  output       memtoregE, memtoregM,
                  output       memtoregW, memwriteM,
                  output       pcsrcD, branchD, alusrcE,
                  output       regdstE, regwriteE,
                  output       regwriteM, regwriteW,
                  output       jumpD,
                  output [2:0] alucontrolE);

  wire [1:0] aluopD;

  wire       memtoregD, memwriteD, alusrcD,
             regdstD, regwriteD;
  wire [2:0] alucontrolD;
  wire       memwriteE;

  maindec md(opD, memtoregD, memwriteD, branchD,
             alusrcD, regdstD, regwriteD, jumpD,
             aluopD);
  aludec  ad(functD, aluopD, alucontrolD);

  assign pcsrcD = branchD & equalD;

  // pipeline registers
  floprc #(8) regE(clk, reset, flushE,
                   {memtoregD, memwriteD, alusrcD,
                    regdstD, regwriteD, alucontrolD},
                   {memtoregE, memwriteE, alusrcE,
                   regdstE, regwriteE,  alucontrolE});
  flopr #(3) regM(clk, reset,
                  {memtoregE, memwriteE, regwriteE},
                  {memtoregM, memwriteM, regwriteM});
  flopr #(2) regW(clk, reset,
                  {memtoregM, regwriteM},
                  {memtoregW, regwriteW});
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity controller is -- pipelined control decoder
    port(clk, reset:              in  STD_LOGIC;
         opD, functD:             in  STD_LOGIC_VECTOR(5 downto 0);
         flushE, equalD:          in  STD_LOGIC;
branchD:                  inout STD_LOGIC;
         memtoregE, memtoregM:    inout STD_LOGIC;
         memtoregW, memwriteM:    out STD_LOGIC;
         pcsrcD, alusrcE:         out STD_LOGIC;
         regdstE:                 out STD_LOGIC;
regwriteE:                inout STD_LOGIC;
         regwriteM, regwriteW:    inout STD_LOGIC;
         jumpD:                   out STD_LOGIC;
         alucontrolE:             out STD_LOGIC_VECTOR(2 downto 0));
end;


architecture struct of controller is
 component maindec
    port(op:                in  STD_LOGIC_VECTOR(5 downto 0);
         memtoreg, memwrite: out STD_LOGIC;
         branch, alusrc:    out STD_LOGIC;
         regdst, regwrite:  out STD_LOGIC;
         jump:              out STD_LOGIC;
         aluop:             out  STD_LOGIC_VECTOR(1 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;
  component flopr is 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 floprc generic(width: integer);
    port(clk, reset: in  STD_LOGIC;
         clear:      in  STD_LOGIC;
         d:          in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:          out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  signal aluopD: STD_LOGIC_VECTOR(1 downto 0);
  signal memtoregD, memwriteD, alusrcD: STD_LOGIC;
  signal regdstD, regwriteD: STD_LOGIC;
  signal alucontrolD: STD_LOGIC_VECTOR(2 downto 0);
  signal memwriteE: STD_LOGIC;

  -- internal signals
  signal d_regE: STD_LOGIC_VECTOR(7 downto 0);
  signal q_regE: STD_LOGIC_VECTOR(7 downto 0);
  signal d_regM: STD_LOGIC_VECTOR(2 downto 0);
  signal q_regM: STD_LOGIC_VECTOR(2 downto 0);
  signal d_regW: STD_LOGIC_VECTOR(1 downto 0);
  signal q_regW: STD_LOGIC_VECTOR(1 downto 0);
begin
  md: maindec port map(opD, memtoregD, memwriteD, branchD,
            alusrcD, regdstD, regwriteD, jumpD,
            aluopD);
  ad: aludec port map(functD, aluopD, alucontrolD);

  pcsrcD <= branchD and equalD;
```

(controller continued from previous page)

VHDL

```
  -- pipeline registers
  regE: floprc generic map(8) port map (clk, reset, flushE,
                d_regE, q_regE);
  regM: flopr generic map(3) port map(clk, reset,
                d_regM, q_regM);
  regW: flopr generic map(2) port map(clk, reset,
                d_regW, q_regW);


  d_regE <= memtoregD & memwriteD & alusrcD & regdstD &
            regwriteD & alucontrolD;
  memtoregE   <= q_regE(7);
  memwriteE   <= q_regE(6);
  alusrcE     <= q_regE(5);
  regdstE     <= q_regE(4);
  regwriteE   <= q_regE(3);
  alucontrolE <= q_regE(2 downto 0);

  d_regM <= memtoregE & memwriteE & regwriteE;
  memtoregM <= q_regM(2);
  memwriteM <= q_regM(1);
  regwriteM <= q_regM(0);

  d_regW <= memtoregM & regwriteM;
  memtoregW <= q_regW(1);
  regwriteW <= q_regW(0);
end;
```

MIPS Pipelined Main Decoder

Verilog

```
module maindec(input  [5:0] op,
               output       memtoreg, memwrite,
               output       branch, alusrc,
               output       regdst, regwrite,
               output       jump,
               output [1:0] aluop);

  reg [9:0] controls;

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

  always @(*)
    case(op)
      6'b000000: controls <= 9'b110000010; //Rtyp
      6'b100011: controls <= 9'b101001000; //LW
      6'b101011: controls <= 9'b001010000; //SW
      6'b000100: controls <= 9'b000100001; //BEQ
      6'b001000: controls <= 9'b101000000; //ADDI
      6'b000010: controls <= 9'b000000100; //J
      default:   controls <= 9'bxxxxxxxxx; //???
    endcase
endmodule
```

VHDL

```
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, alusrc:     out STD_LOGIC;
       regdst, regwrite:   out STD_LOGIC;
       jump:               out STD_LOGIC;
       aluop:              out  STD_LOGIC_VECTOR(1 downto 0));
end;

architecture behave of maindec is
  signal controls: STD_LOGIC_VECTOR(8 downto 0);
begin
  process(op) begin
    case op is
      when "000000" => controls <= "110000010"; -- Rtype
      when "100011" => controls <= "101001000"; -- LW
      when "101011" => controls <= "001010000"; -- SW
      when "000100" => controls <= "000100001"; -- BEQ
      when "001000" => controls <= "101000000"; -- ADDI
      when "000010" => controls <= "000000100"; -- J
      when "001010" => controls <= "101000011"; -- SLTI
      when others   => controls <= "---------"; -- illegal op
    end case;
  end process;

  regwrite <= controls(8);
  regdst   <= controls(7);
  alusrc   <= controls(6);
  branch   <= controls(5);
  memwrite <= controls(4);
  memtoreg <= controls(3);
  jump     <= controls(2);
  aluop    <= controls(1 downto 0);
end;
```

Note: `slti` is included in the VHDL implementation but could easily be removed.

MIPS Pipelined ALU Decoder

Verilog

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

  always @(*)
    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
    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(aluop, funct) 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;
```

MIPS Pipelined Datapath

Verilog

```
module datapath(input         clk, reset,
                input         memtoregE, memtoregM,
memtoregW,
                input         pcsrcD, branchD,
                input         alusrcE, regdstE,
                input         regwriteE, regwriteM,
regwriteW,
                input         jumpD,
                input  [2:0]  alucontrolE,
                output        equalD,
                output [31:0] pcF,
                input  [31:0] instrF,
                output [31:0] aluoutM, writedataM,
                input  [31:0] readdataM,
                output [5:0]  opD, functD,
                output        flushE);

  wire        forwardaD, forwardbD;
  wire [1:0]  forwardaE, forwardbE;
  wire        stallF;
  wire [4:0]  rsD, rtD, rdD, rsE, rtE, rdE;
  wire [4:0]  writeregE, writeregM, writeregW;
  wire        flushD;
    wire  [31:0]  pcnextFD,  pcnextbrFD,  pcplus4F,
pcbranchD;
  wire [31:0] signimmD, signimmE, signimmshD;
  wire [31:0] srcaD, srca2D, srcaE, srca2E;
  wire [31:0] srcbD, srcb2D, srcbE, srcb2E, srcb3E;
  wire [31:0] pcplus4D, instrD;
  wire [31:0] aluoutE, aluoutW;
  wire [31:0] readdataW, resultW;

  // hazard detection
 hazard   h(rsD, rtD, rsE, rtE, writeregE, writeregM,
          writeregW,regwriteE, regwriteM, regwriteW,
              memtoregE, memtoregM, branchD,
              forwardaD, forwardbD, forwardaE,
              forwardbE,
              stallF, stallD, flushE);

  // next PC logic (operates in fetch and decode)
  mux2 #(32)  pcbrmux(pcplus4F, pcbranchD, pcsrcD,
                      pcnextbrFD);
  mux2 #(32)  pcmux(pcnextbrFD,{pcplus4D[31:28],
                    instrD[25:0], 2'b00},
                    jumpD, pcnextFD);

  // register file (operates in decode and writeback)
  regfile     rf(clk, regwriteW, rsD, rtD, writeregW,
                 resultW, srcaD, srcbD);

  // Fetch stage logic
  flopenr #(32) pcreg(clk, reset, ~stallF,
                      pcnextFD, pcF);
  adder       pcadd1(pcF, 32'b100, pcplus4F);
```

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;
       memtoregE, memtoregM, memtoregW: in STD_LOGIC;
       pcsrcD, branchD:               in  STD_LOGIC;
       alusrcE, regdstE:              in  STD_LOGIC;
       regwriteE, regwriteM, regwriteW: in STD_LOGIC;
       jumpD:                         in  STD_LOGIC;
       alucontrolE:                   in  STD_LOGIC_VECTOR(2 downto 0);
       equalD:                        out STD_LOGIC;
       pcF:                          inout STD_LOGIC_VECTOR(31 downto 0);
       instrF:                         in  STD_LOGIC_VECTOR(31 downto 0);
       aluoutM, writedataM:          inout STD_LOGIC_VECTOR(31 downto 0);
       readdataM:                      in  STD_LOGIC_VECTOR(31 downto 0);
       opD, functD:                    out STD_LOGIC_VECTOR(5 downto 0);
       flushE:                         inout STD_LOGIC);
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 is 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 floprc is generic(width: integer);
    port(clk, reset: in  STD_LOGIC;
         clear:      in  STD_LOGIC;
         d:          in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:          out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component flopenrc is generic(width: integer);
    port(clk, reset: in  STD_LOGIC;
         en, clear:  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;
```

(continued from previous page)

Verilog

```
  // Decode stage
   flopenr #(32) r1D(clk, reset, ~stallD, pcplus4F,
pcplus4D);
  flopenrc #(32) r2D(clk, reset, ~stallD, flushD, in-
strF, instrD);
  signext     se(instrD[15:0], signimmD);
  sl2         immsh(signimmD, signimmshD);
  adder      pcadd2(pcplus4D, signimmshD, pcbranchD);
  mux2 #(32)  forwardadmux(srcaD, aluoutM, forwardaD,
srca2D);
  mux2 #(32)  forwardbdmux(srcbD, aluoutM, forwardbD,
srcb2D);
  eqcmp       comp(srca2D, srcb2D, equalD);

  assign opD = instrD[31:26];
  assign functD = instrD[5:0];
  assign rsD = instrD[25:21];
  assign rtD = instrD[20:16];
  assign rdD = instrD[15:11];

  assign flushD = pcsrcD | jumpD;

  // Execute stage
  floprc #(32) r1E(clk, reset, flushE, srcaD, srcaE);
  floprc #(32) r2E(clk, reset, flushE, srcbD, srcbE);
  floprc #(32) r3E(clk, reset, flushE, signimmD, sign-
immE);
  floprc #(5)  r4E(clk, reset, flushE, rsD, rsE);
  floprc #(5)  r5E(clk, reset, flushE, rtD, rtE);
  floprc #(5)  r6E(clk, reset, flushE, rdD, rdE);
   mux3 #(32)  forwardaemux(srcaE, resultW, aluoutM,
forwardaE, srca2E);
   mux3 #(32)  forwardbemux(srcbE, resultW, aluoutM,
forwardbE, srcb2E);
    mux2 #(32)   srcbmux(srcb2E, signimmE, alusrcE,
srcb3E);
  alu          alu(srca2E, srcb3E, alucontrolE, alu-
outE);
  mux2 #(5)   wrmux(rtE, rdE, regdstE, writeregE);

  // Memory stage
  flopr #(32) r1M(clk, reset, srcb2E, writedataM);
  flopr #(32) r2M(clk, reset, aluoutE, aluoutM);
  flopr #(5)  r3M(clk, reset, writeregE, writeregM);

  // Writeback stage
  flopr #(32) r1W(clk, reset, aluoutM, aluoutW);
  flopr #(32) r2W(clk, reset, readdataM, readdataW);
  flopr #(5)  r3W(clk, reset, writeregM, writeregW);
   mux2 #(32)  resmux(aluoutW, readdataW, memtoregW,
resultW);

endmodule
```

VHDL

```
 component eqcmp is
   port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
        y:    out STD_LOGIC);
 end component;
 component hazard
   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;
        forwardaD, forwardbD:            out STD_LOGIC;
        forwardaE, forwardbE:            out STD_LOGIC_VECTOR(1 downto 0);
        stallF, flushE:                   out STD_LOGIC;
        stallD:                           inout STD_LOGIC);
 end component;
 signal forwardaD, forwardbD: STD_LOGIC;
 signal forwardaE, forwardbE: STD_LOGIC_VECTOR(1 downto 0);
 signal stallF, stallFbar, stallD, stallDbar: STD_LOGIC;
 signal rsD, rtD, rdD, rsE, rtE, rdE: STD_LOGIC_VECTOR(4 downto 0);
 signal writeregE, writeregM, writeregW: STD_LOGIC_VECTOR(4 downto 0);
 signal flushD: STD_LOGIC;
 signal pcnextFD, pcnextbrFD, pcplus4F, pcbranchD:
   STD_LOGIC_VECTOR(31 downto 0);
 signal signimmD, signimmE, signimmshD: STD_LOGIC_VECTOR(31 downto 0);
 signal srcaD, srca2D, srcaE, srca2E: STD_LOGIC_VECTOR(31 downto 0);
 signal srcbD, srcb2D, srcbE, srcb2E, srcb3E:
   STD_LOGIC_VECTOR(31 downto 0);
 signal pcplus4D, instrD: STD_LOGIC_VECTOR(31 downto 0);
 signal aluoutE, aluoutW: STD_LOGIC_VECTOR(31 downto 0);
 signal readdataW, resultW: STD_LOGIC_VECTOR(31 downto 0);
 signal d1_pcmux: STD_LOGIC_VECTOR(31 downto 0);
begin
 -- hazard detection
 h: hazard port map(rsD, rtD, rsE, rtE, writeregE, writeregM, writeregW,
             regwriteE, regwriteM, regwriteW,
             memtoregE, memtoregM, branchD,
             forwardaD, forwardbD, forwardaE, forwardbE,
             stallF, stallD, flushE);

 -- next PC logic (operates in fetch and decode)
 d1_pcmux <= pcplus4D(31 downto 28) & instrD(25 downto 0) & "00";
 pcbrmux: mux2 generic map(32) port map(pcplus4F, pcbranchD, pcsrcD,
                                        pcnextbrFD);
 pcmux: mux2 generic map(32) port map(pcnextbrFD, d1_pcmux,
                                      jumpD, pcnextFD);

 -- register file (operates in decode and writeback)
 rf: regfile port map(clk, regwriteW, rsD, rtD, writeregW,
                 resultW, srcaD, srcbD);

 -- Fetch stage logic
 stallDbar <= (not stallD);
 stallFbar <= (not stallF);

 pcreg: flopenr generic map(32) port map(clk, reset, stallFbar,
                                         pcnextFD, pcF);
 pcadd1: adder port map(pcF, "00000000000000000000000000000100",
                        pcplus4F);

 -- Decode stage
 r1D: flopenr generic map(32) port map(clk, reset, stallDbar,
                                       pcplus4F, pcplus4D);
 r2D: flopenrc generic map(32) port map(clk, reset, stallDbar,
                                        flushD, instrF, instrD);
 se: signext port map(instrD(15 downto 0), signimmD);
 immsh: sl2 port map(signimmD, signimmshD);
 pcadd2: adder port map(pcplus4D, signimmshD, pcbranchD);
 forwardadmux: mux2 generic map(32) port map(srcaD, aluoutM,
                                             forwardaD, srca2D);
 forwardbdmux: mux2 generic map(32) port map(srcbD, aluoutM,
                                             forwardbD, srcb2D);
 comp: eqcmp port map(srca2D, srcb2D, equalD);

 opD <= instrD(31 downto 26);
 functD <= instrD(5 downto 0);
 rsD <= instrD(25 downto 21);
 rtD <= instrD(20 downto 16);
 rdD <= instrD(15 downto 11);

 flushD <= pcsrcD or jumpD;
```

(continued from previous page)

Verilog

VHDL

```
  -- Execute stage
  r1E: floprc generic map(32) port map(clk, reset, flushE, srcaD, srcaE);
  r2E: floprc generic map(32) port map(clk, reset, flushE, srcbD, srcbE);
  r3E: floprc generic map(32) port map(clk, reset, flushE, signimmD,
                                       signimmE);
  r4E: floprc generic map(5) port map(clk, reset, flushE, rsD, rsE);
  r5E: floprc generic map(5) port map(clk, reset, flushE, rtD, rtE);
  r6E: floprc generic map(5) port map(clk, reset, flushE, rdD, rdE);
  forwardaemux: mux3 generic map(32) port map(srcaE, resultW, aluoutM,
                                              forwardaE, srca2E);
  forwardbemux: mux3 generic map(32) port map(srcbE, resultW, aluoutM,
                                              forwardbE, srcb2E);
  srcbmux: mux2 generic map(32) port map(srcb2E, signimmE, alusrcE,
                                         srcb3E);
  alu1: alu port map(srca2E, srcb3E, alucontrolE, aluoutE);
  wrmux: mux2 generic map(5) port map(rtE, rdE, regdstE, writeregE);

  -- Memory stage
  r1M: flopr generic map(32) port map(clk, reset, srcb2E, writedataM);
  r2M: flopr generic map(32) port map(clk, reset, aluoutE, aluoutM);
  r3M: flopr generic map(5)  port map(clk, reset, writeregE, writeregM);

  -- Writeback stage
  r1W: flopr generic map(32) port map(clk, reset, aluoutM, aluoutW);
  r2W: flopr generic map(32) port map(clk, reset, readdataM, readdataW);
  r3W: flopr generic map(5)  port map(clk, reset, writeregM, writeregW);
  resmux: mux2 generic map(32) port map(aluoutW, readdataW, memtoregW,
                                        resultW);
end;
```

The following describes the building blocks that are used in the MIPS multicycle processor that are not found in Section 7.6.2.

MIPS Pipelined Processor Hazard Unit

Verilog

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

  wire 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 @(*)
    begin
      forwardaE = 2'b00; forwardbE = 2'b00;
      if (rsE != 0)
        if (rsE == writeregM & regwriteM)
          forwardaE = 2'b10;
        else if (rsE == writeregW & regwriteW)
          forwardaE = 2'b01;
      if (rtE != 0)
        if (rtE == writeregM & regwriteM)
          forwardbE = 2'b10;
        else if (rtE == writeregW & regwriteW)
          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;
       forwardaD, forwardbD:            out STD_LOGIC;
       forwardaE, forwardbE:            out STD_LOGIC_VECTOR(1 downto 0);
       stallF, flushE:                  out STD_LOGIC;
       stallD:                          inout STD_LOGIC);
  end;

architecture behave of hazard is
  signal lwstallD, branchstallD: STD_LOGIC;
begin

  -- forwarding sources to D stage (branch equality)
  forwardaD <= '1' when ((rsD /= "00000") and (rsD = writeregM) and
                         (regwriteM = '1'))
               else '0';
  forwardbD <= '1' when ((rtD /= "00000") and (rtD = writeregM) and
                         (regwriteM = '1'))
               else '0';

  -- forwarding sources to E stage (ALU)
  process(rsE, rtE, writeregM, regwriteM, writeregW, regwriteW) begin
    forwardaE <= "00"; forwardbE <= "00";
    if (rsE /= "00000") then
      if ((rsE = writeregM) and (regwriteM = '1')) then
        forwardaE <= "10";
      elsif ((rsE = writeregW) and (regwriteW = '1')) then
        forwardaE <= "01";
      end if;
    end if;
    if (rtE /= "00000") then
      if ((rtE = writeregM) and (regwriteM = '1')) then
        forwardbE <= "10";
      elsif ((rtE = writeregW) and (regwriteW = '1')) then
        forwardbE <= "01";
      end if;
    end if;
  end process;

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

MIPS Pipelined Processor Parts

Verilog

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

  always @(posedge clk, posedge reset)
    if (reset)      q <= #1 0;
    else if (clear) q <= #1 0;
    else            q <= #1 d;
endmodule

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

  always @(posedge clk, posedge reset)
    if      (reset) q <= #1 0;
    else if (clear) q <= #1 0;
    else if (en)    q <= #1 d;
endmodule
```

VHDL

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

architecture synchronous of floprc is
begin
  process(clk, reset, clear) begin
    if clk'event and clk = '1' then
      if reset = '1' then  q <= CONV_STD_LOGIC_VECTOR(0, width);
      elsif clear = '1' then q <= CONV_STD_LOGIC_VECTOR(0, width);
      else q <= d;
      end if;
    end if;
  end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;  use IEEE.STD_LOGIC_ARITH.all;
entity flopenrc is -- flip-flop with synchronous reset, enable, and clear
  generic(width: integer);
  port(clk, reset: in  STD_LOGIC;
       en, clear:  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 flopenrc is
begin
  process(clk, reset, clear) begin
    if clk'event and clk = '1' then
      if reset = '1' then  q <= CONV_STD_LOGIC_VECTOR(0, width);
      elsif clear = '1' then q <= CONV_STD_LOGIC_VECTOR(0, width);
      elsif en = '1' then q <= d;
      end if;
    end if;
  end process;
end;
```

MIPS Pipelined Processor Memories

Verilog

```
module imem(input  [5:0]  a,
            output [31:0] rd);

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

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

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

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

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

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

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

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

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.STD_LOGIC_UNSIGNED.all;  use IEEE.STD_LOGIC_ARITH.all;
entity 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
    file memfile: TEXT;
    variable L: line;
    variable ch: character;
    variable index, result: integer;
    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(conv_integer(i)) := CONV_STD_LOGIC_VECTOR(0, 32);
    end loop;
    index := 0;
    FILE_OPEN(memfile, "memfile.dat", READ_MODE);
    while not endfile(memfile) loop
      readline(memfile, L);
      result := 0;
      for i in 1 to 8 loop
        read(L, ch);
        if '0' <= ch and ch <= '9' then
            result := result*16 + character'pos(ch) - character'pos('0');
        elsif 'a' <= ch and ch <= 'f' then
          result := result*16 + character'pos(ch) - character'pos('a')+10;
        else report "Format error on line " & integer'image(index)
             severity error;
        end if;
      end loop;
      mem(index) := CONV_STD_LOGIC_VECTOR(result, 32);
      index := index + 1;
    end loop;

    -- read memory
    loop
      rd <= mem(CONV_INTEGER(a));
      wait on a;
    end loop;
  end process;
end;

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 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(CONV_INTEGER(a(7 downto 2))) := wd;
          end if;
      end if;
      rd <= mem(CONV_INTEGER(a(7 downto 2)));
      wait on clk, a;
    end loop;

  end process;
end;
```

MIPS Pipelined Processor Testbench

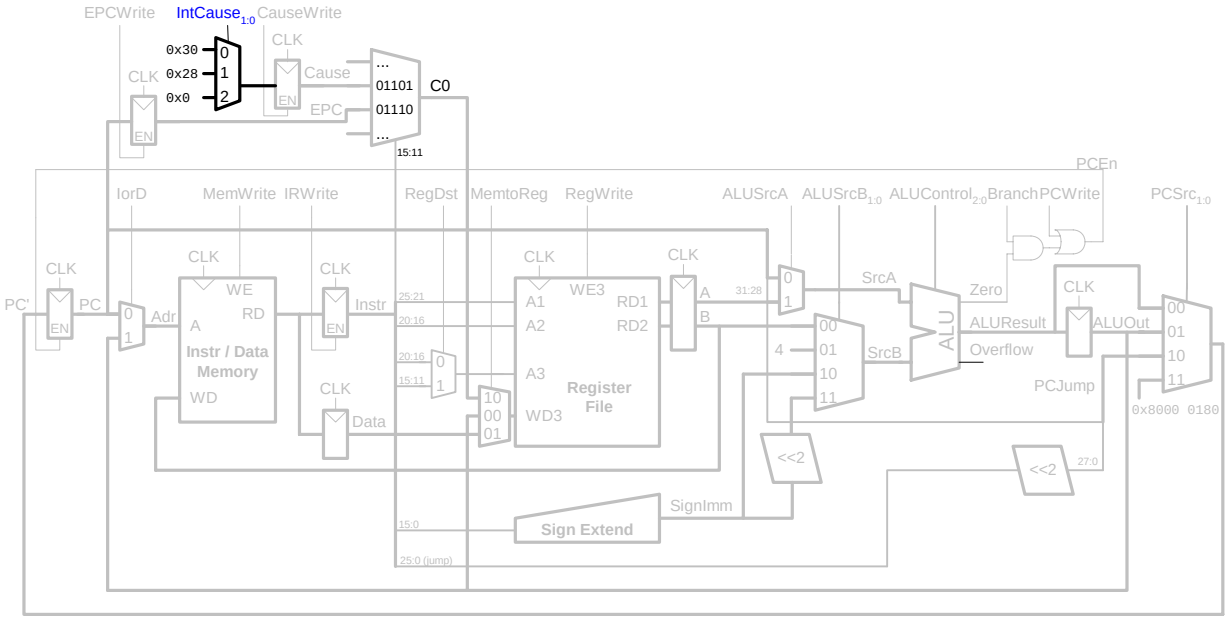
Verilog

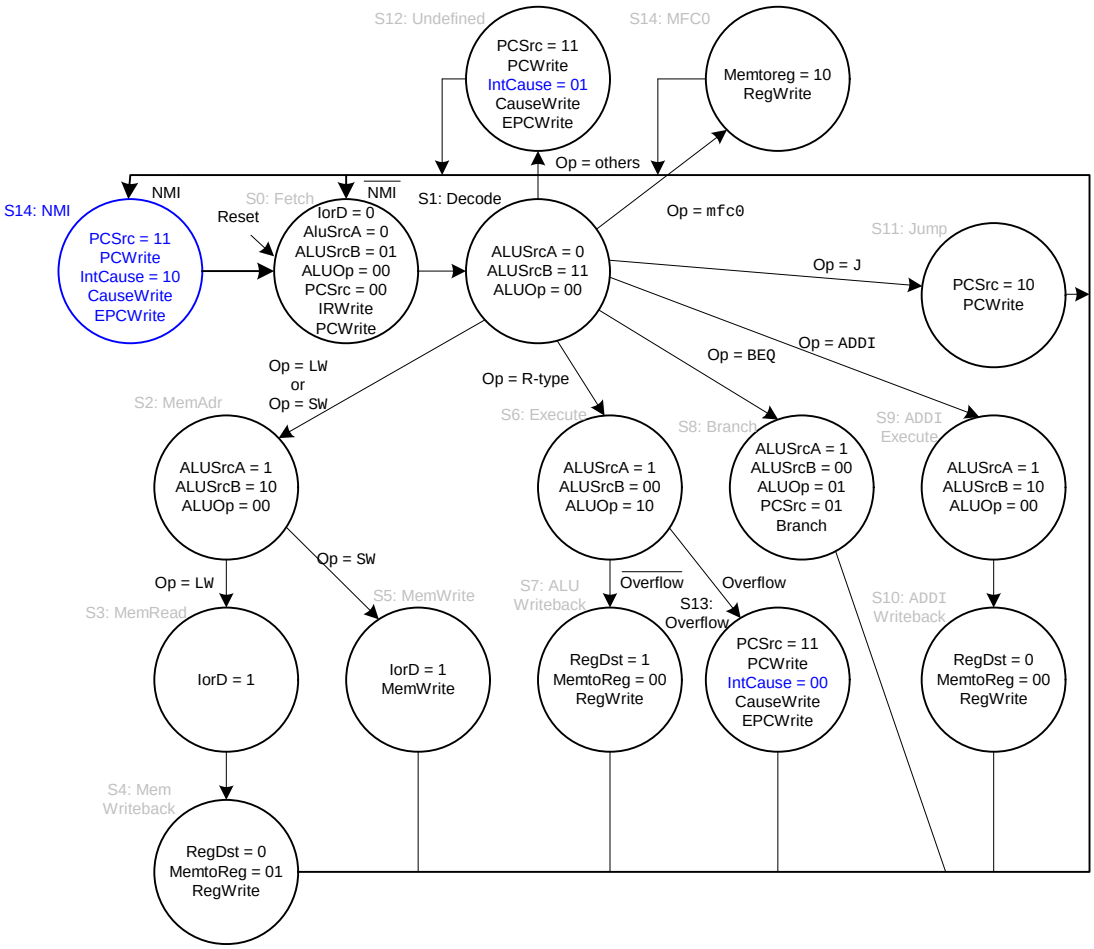
```
module testbench();

  reg         clk;
  reg         reset;

  wire [31:0] writedata, dataadr;
  wire memwrite;

  // instantiate device to be tested
  top dut(clk, reset, writedata, dataadr, memwrite);

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

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

  // check results
  always@(negedge clk)
    begin
      if(memwrite) begin
        if(dataadr === 84 & writedata === 7) begin
          $display("Simulation succeeded");
          $stop;
        end else if (dataadr !== 80) begin
          $display("Simulation failed");
          $stop;
        end
      end
    end
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.STD_LOGIC_UNSIGNED.all;
entity testbench is
end;

architecture test of testbench is
  component top is
  port(clk, reset:           in    STD_LOGIC;
       writedata, dataadr:   inout STD_LOGIC_VECTOR(31 downto 0);
       memwrite:             inout 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 (conv_integer(dataadr) = 84 and conv_integer(writedata) = 7) then
        report "Simulation succeeded"
        severity failure;
      elsif (dataadr /= 80) then
        report "Simulation failed"
        severity failure;
      end if;
    end if;
  end process;
end;
```

7.35


EPCWrite
IntCause1:0
CauseWrite
0x30
0x28
0x0
CLK
Cause
EPC
01101
01110
C0
15:11
PCEn
IorD
MemWrite
IRWrite
RegDst
MemtoReg
RegWrite
ALUSrcA
ALUSrcB1:0
ALUControl2:0
Branch
PCWrite
PCSrc1:0
PC'
PC
Adr
WE
RD
A
Instr / Data Memory
WD
Instr
Data
25:21
20:16
15:11
A1
A2
A3
WE3
WD3
RD1
RD2
Register File
A
B
31:28
SrcA
SrcB
ALU
Zero
ALUResult
Overflow
ALUOut
PCJump
0x8000 0180
<<2
27:0
SignImm
Sign Extend
15:0
25:0 (jump)



Question 7.1

A pipelined microprocessors 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 is 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

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.

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%.

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.

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 a100% 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.

8.9

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

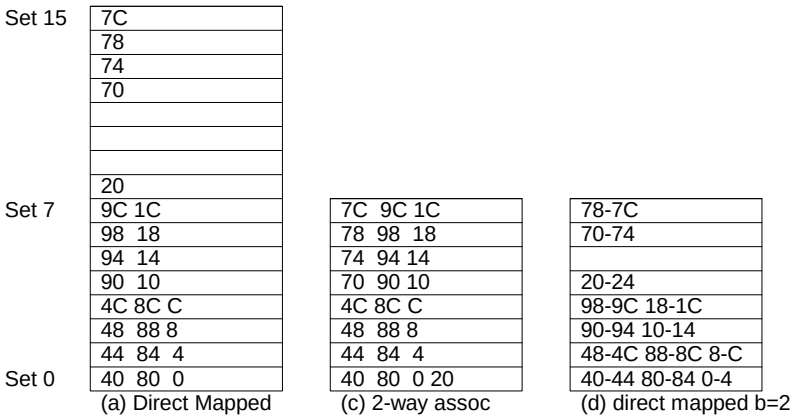


FIGURE 8.1 Address mappings for Exercise 8.9

(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.

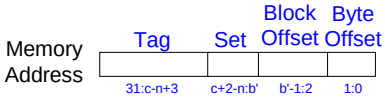
(c) **100% miss rate**. The repeated sequence makes at least three accesses to each set during each pass. Using LRU replacement, each value must be replaced each pass through.

(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%.

8.11

(a - b)



(c) Each tag is 32 - (c+2-n) bits = (30 - (c-n)) bits

(d) # tag bits × # blocks = (30 - (c-n)) × $2^{c+2-b'}$

8.13

(a) The word in memory might be found in two locations, one in the on-chip cache, and one in the off-chip cache.

(b) For the first-level cache, the number of sets, S = 512 / 4 = 128 sets. Thus, 7 bits of the address are set bits. The block size is 16 bytes / 4 bytes/word = 4 words, so there are 2 block offset bits. Thus, the number of tag bits for the first-level cache is 32 - (7+2+2) = **21 bits**.

For the second-level cache, the number of sets is equal to the number of blocks, S = 256 Ksets. Thus, 18 bits of the address are set bits. The block size is 16 bytes / 4 bytes/word = 4 words, so there are 2 block offset bits. Thus, the number of tag bits for the second-level cache is 32 - (18+2+2) = **10 bits**.

(c) From Equation 8.2, $AMAT = t_{cache} + MR_{cache}(t_{MM} + MR_{MM}\, t_{VM})$. In this case, there is no virtual memory but there is an L2 cache. Thus,

$AMAT = t_{cache} + MR_{cache}(t_{L2cache} + MR_{L2cache}\, t_{MM})$

Where, *MR* is the miss rate. In terms of hit rate, $MR_{cache} = 1 - HR_{cache}$, and $MR_{L2cache} = 1 - HR_{L2cache}$. Using the values given in Table 8.6,

$$\boldsymbol{AMAT = t_a + (1 - A)(t_b + (1 - B)\, t_m)}$$

(d)

When the first-level cache is enabled, the second-level cache receives only the "hard" accesses, ones that don't show enough temporal and spatial locality to hit in the first-level cache. The "easy" accesses (ones with good temporal and spatial locality) hit in the first-level cache, even though they would have also hit in the second-level cache. When the first-level cache is disabled, the hit rate

goes up because the second-level cache supplies both the "easy" accesses and some of the "hard" accesses.

8.15

(a)

From Equation 8.2, $AMAT = t_{cache} + MR_{cache}\,(t_{MM} + MR_{MM}\,t_{VM})$. In this case, there is no virtual memory. Thus,

$AMAT = t_{cache} + MR_{cache}\,t_{MM}$

With a cycle time of 1/1 GHz = 1 ns,

***AMAT* = 1 ns + 0.05(60 ns) = 4 ns**

(b) CPI = 4 + 4 = **8 cycles** (for a load)
CPI = 4 + 3 = **7 cyles** (for a store)

(c) Average CPI = (0.11 + 0.2)(3) + (0.52)(4) + (0.1)(7) + (0.25)(8) = **5.71**

(d) Average CPI = 5.71 + 0.07(60) = **9.91**

8.17

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.

8.19

(a) From Equation 8.2, $AMAT = t_{cache} + MR_{cache}\,(t_{MM} + MR_{MM}\,t_{VM})$. However, each data access now requires an address translation (page table or TLB lookup). Thus,

Without the TLB:

$AMAT = t_{MM} + [t_{cache} + MR_{cache}\,(t_{MM} + MR_{MM}\,t_{VM})]$

$AMAT$ = 100 + [1 + 0.02(100 + 0.000003(1,000,000))] cycles = **103.06 cycles**

With the TLB:

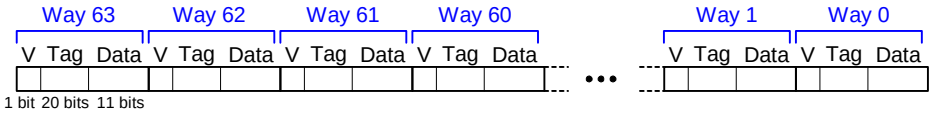
$AMAT = [t_{TLB} + MR_{TLB}(t_{MM})] + [t_{cache} + MR_{cache}\,(t_{MM} + MR_{MM}\,t_{VM})]$

$AMAT$ = [1 + 0.0005(100)] + [1 + 0.02(100 + 0.000003 × 1,000,000)] cycles = **4.11 cycles**

(b) # bits per entry = valid bit + tag bits + physical page number
1 valid bit

tag bits = virtual page number = 20 bits
physical page number = 11 bits

Thus, # bits per entry = 1 + 20 + 11 = **32 bits**
Total size of the TLB = 64 × 32 bits = **2048 bits**
8.19 (c)



(d) **1 × 2048 bit SRAM**

8.21
(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.

8.23
(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.

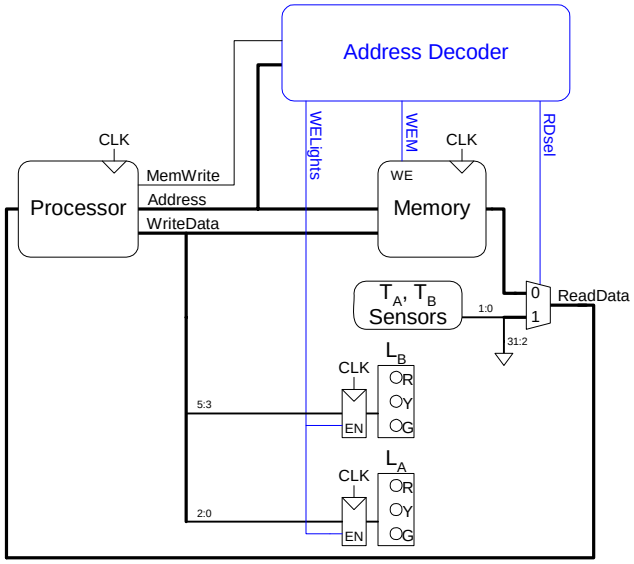
(c) The amount of physical memory affects how many physical pages can be accessed at once. With a small main memory, if many applications run at once or a single application accesses addresses from many different pages, thrashing can occur. This can make the applications dreadfully slow.

8.25

(a)

```
#  MIPS code for Exercise 8.25
       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 = T_A
       bne  $t4, $0,  S0     # if T_A == 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 = T_B
       bne  $t4, $0,  S2     # if T_B == 1, loop back to S2

S3:    sw   $t1, 0xF004($0)  # lights = yellow / red
       j    Start
```

(b)



8.25 (c) **Address Decoder for Exercise 8.25**

Verilog

```
module addrdec(input [31:0] addr, input memwrite,
               output reg   WELights, Mwrite,
               output reg   rdselect);

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

  wire [15:0] addressbits;

  assign addressbits = addr[15:0];

  always @ ( * )
    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;

      // readdata control
      else
  if ( addressbits == T )
          {WELights, Mwrite, rdselect} = 3'b001;
        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(addr, memwrite) begin
    if (addr(31 downto 16) = X"FFFF") then
      -- writedata control
      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;

      -- 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 if;
  end process;
end;
```

Question 8.1

Caches are categorized based on the number of blocks (B) in a set. In a direct mapped cache, each set contains exactly one block, so the cache has S = B sets. Thus a particular main memory address maps to a unique block in the cache. In an N-way set associative cache, each set contains N blocks. The address still maps to a unique set, with S = B / N sets. But the data from that address can go in any of the N blocks in the set. A fully associative cache has only S = 1 set. Data can go in any of the B blocks in the set. Hence, a fully associative cache is another name for a B-way set associative cache.

A **direct mapped cache** performs better than the other two when the data access pattern is to sequential cache blocks in memory with a repeat length one greater than the number of blocks in the cache.

An ***N*-way set-associative cache** performs better than the other two when *N* sequential block accesses map to the same set in the set-associative *and* di-

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.