David Money Harris and Sarah L. Harris, *Digital Design and Computer Architecture*, © 2007 by Elsevier Inc.
Exercise Solutions

CHAPTER 1

SOLUTIONS

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.

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

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

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.

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

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.

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

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

1.8 $2^{32}-1 = 4,294,967,295$

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

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

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

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

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

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

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

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

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

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

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

1.20 00101010; (b) 10111111; (c) 01111100; (d) overflow; (e) overflow

1.21 (a) 00000101; (b) 11111010

1.22 (a) 00000101; (b) 00001010

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

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

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

1.26 4, 8

1.27 8

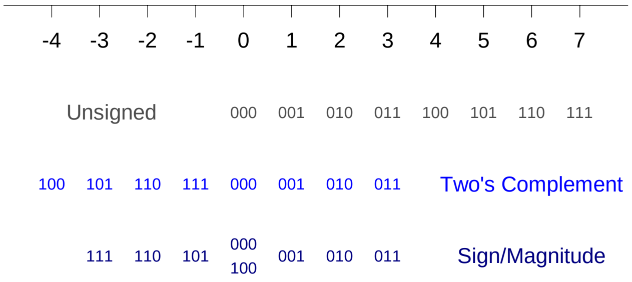
1.28 5,760,000

1.29 46.566 gigabytes

1.30 2 billion

1.31 128 kbits

1.32



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

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

1.35 (a) 11011101; (b) 110001000

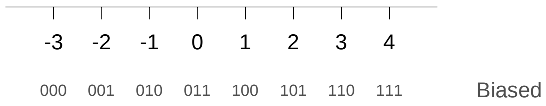
1.36 (a) 010000 + 001001 = 011001; (b) 011011 + 011111 = 111010 (overflow); (c) 111100 + 010011 = 001111; (d) 000011 + 100000 = 100011; (e) 110000 + 110111 = 100111; (f) 100101 + 100001 = 000110 (overflow)

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

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

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

1.40



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.

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

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

1.44 Both are right.

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);
}
```

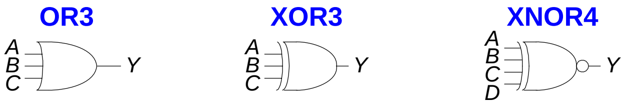
1.46

```
#include <stdio.h>

void main(void)
{
   int i = 0, j, dec = 0, digit;
   char hex[80], tmp;

   printf("Enter decimal number: ");
   scanf("%d", &dec);

   while (dec > 0) {
      digit = dec % 16;
      hex[i] = digit < 10 ? digit + '0' : digit + 'A' - 10;
      dec = dec / 16;
      i = i + 1;
   }
   hex[i] = 0;

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

   printf("The hexadecimal equivalent is %s\n", hex);
}
```

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);
}
```

1.48



$Y = A+B+C$

$Y = A \oplus B \oplus C$

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

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

(a)

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

(b)

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

(c)

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

1.50

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

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

1.52

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

Zero

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

A NOR B

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

$\overline{\text{A}}\text{B}$

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

NOT A

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

$\text{A}\overline{\text{B}}$

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

NOT B

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

XOR

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

NAND

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

AND

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

XNOR

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

B

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

$\overline{\text{A}} + \text{B}$

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

A

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

$\text{A} + \overline{\text{B}}$

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

OR

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

One

1.53 2^{2^N}

1.54 $V_{IL} = 2.5$; $V_{IH} = 3$; $V_{OL} = 1.5$; $V_{OH} = 4$; $NM_L = 1$; $NM_H = 1$

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.

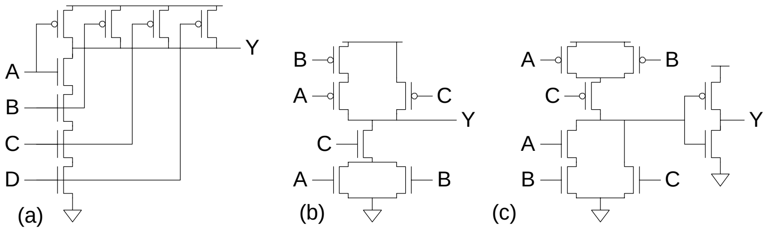
1.56 $V_{IL} = 2$; $V_{IH} = 4$; $V_{OL} = 1$; $V_{OH} = 4.5$; $NM_L = 1$; $NM_H = 0.5$

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.

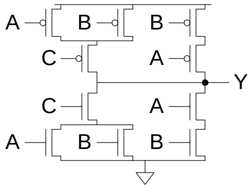
1.58 (a) AND gate; (b) $V_{IL} = 1.5$; $V_{IH} = 2.75$; $V_{OL} = 0$; $V_{OH} = 3$

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

1.60



1.61



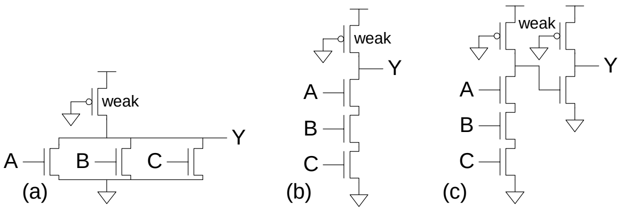
1.62 XOR

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

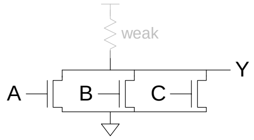
1.63

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

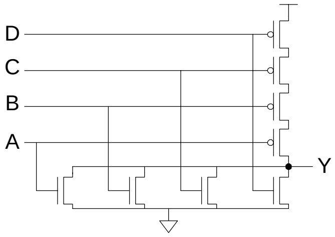
1.64



1.65



Question 1.1



Question 1.2

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

Question 1.3

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

CHAPTER 2

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)

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

(e)

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

2.2

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

(b)

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

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

(d)

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

(e)

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

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) $Y = \bar{A}\bar{B} + \bar{B}\bar{D} + AC\bar{D}$

(e)

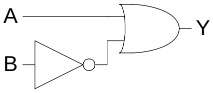
$$Y = \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}CD + \bar{A}B\bar{C}D + \bar{A}BC\bar{D} + A\bar{B}\bar{C}D + A\bar{B}C\bar{D} + AB\bar{C}\bar{D} + ABCD$$

This can also be expressed as:

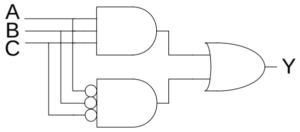
$$Y = \overline{(A \oplus B)}\,\overline{(C \oplus D)} + (A \oplus B)(C \oplus D)$$

2.4

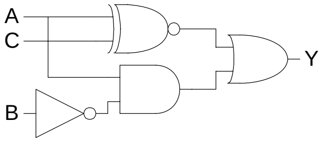
(a)



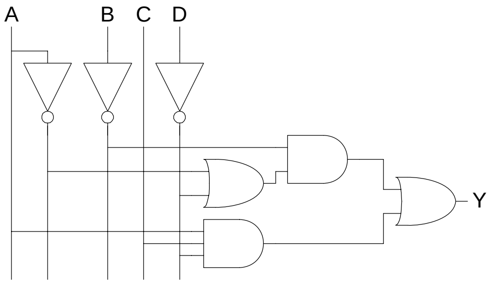
(b)



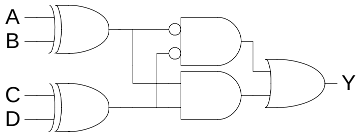
(c)



(d)



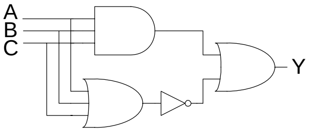
(e)



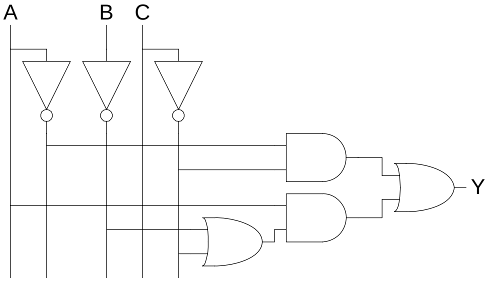
2.5

(a) Same as 2.4(a).

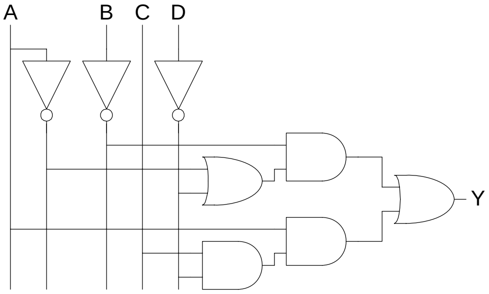
(b)



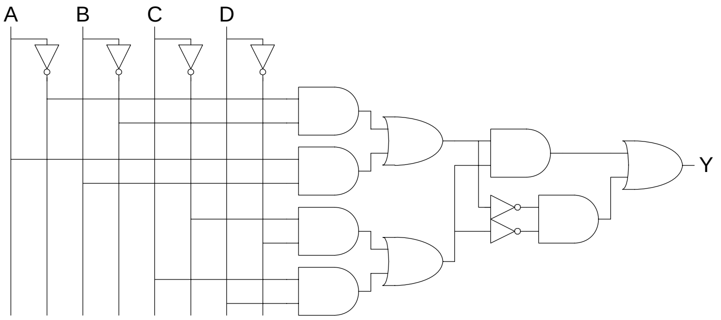
(c)



(d)

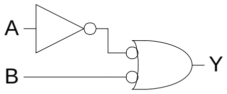


(e)

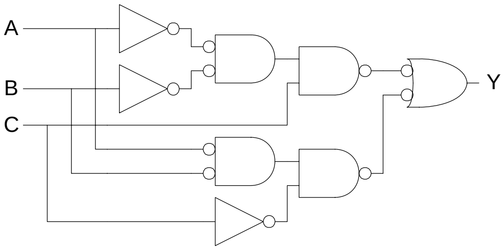


2.6

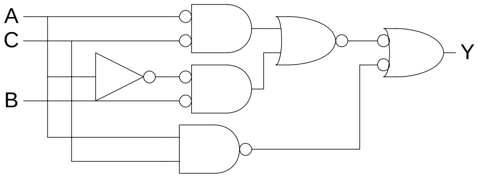
(a)



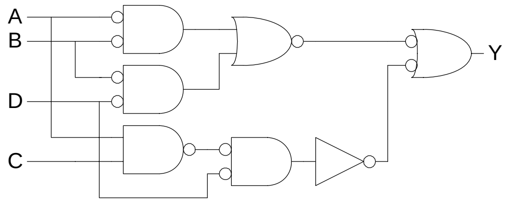
(b)



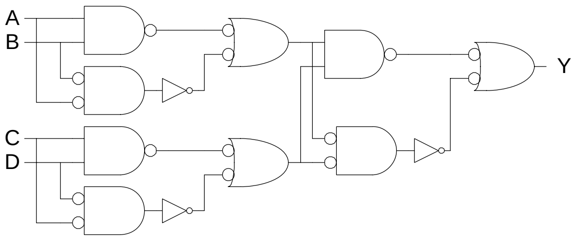
(c)



(d)



(e)



2.7

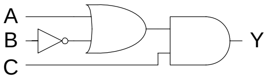
(a) $Y = AC + \overline{B}C$

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

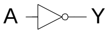
(c) $Y = \overline{A} + \overline{B}\,\overline{C} + \overline{B}\,\overline{D} + BD$

2.8

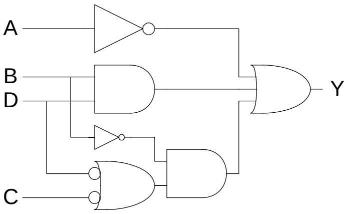
(a)



(b)

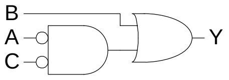


(c)

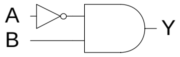


2.9

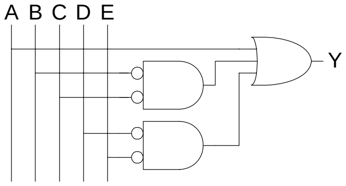
(a) $Y = B + \overline{A}\,\overline{C}$



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

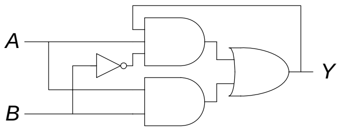


(c) $Y = A + \overline{B}\,\overline{C} + \overline{D}\,\overline{E}$



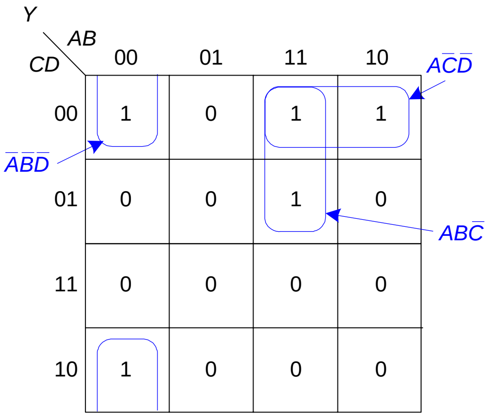
2.10 4 gigarows = 4 x 2^{30} rows = 2^{32} rows, so the truth table has 32 inputs.

2.11



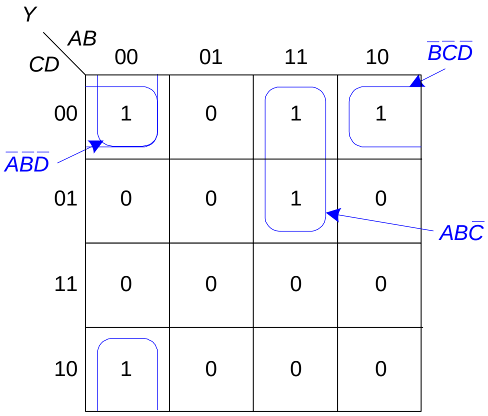
Y = *A*

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



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

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



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

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

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

2.14

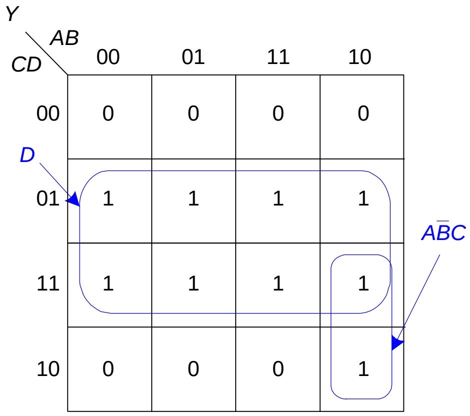
B_2	B_1	B_0	$\overline{B_2 \bullet B_1 \bullet B_0}$	$\overline{B_2} + \overline{B_1} + \overline{B_0}$
0	0	0	1	1
0	0	1	1	1
0	1	0	1	1
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	0	0

2.15

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

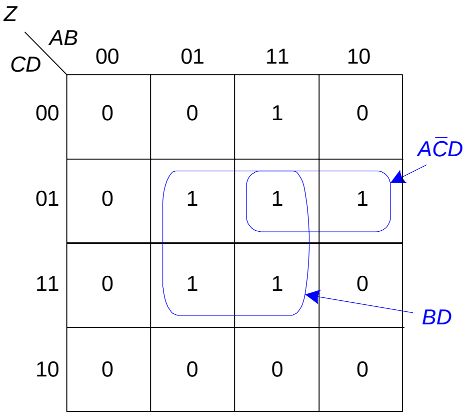
$$Z = A\bar{C}D + BD$$

2.16



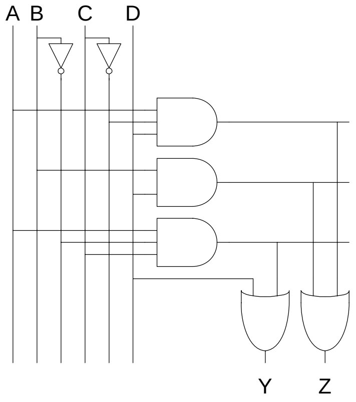
CD \ AB	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	1	1	1	1
10	0	0	0	1

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

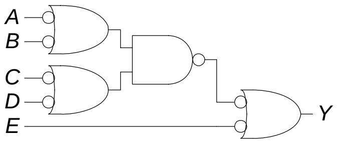


CD \ AB	00	01	11	10
00	0	0	1	0
01	0	1	1	1
11	0	1	1	0
10	0	0	0	0

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

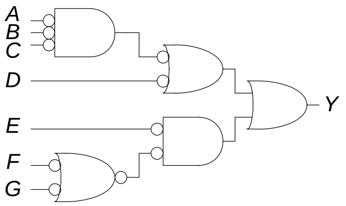


2.17



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

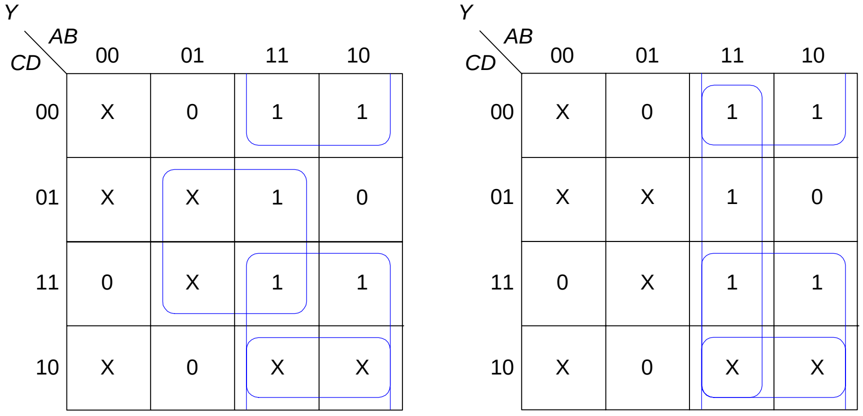
2.18



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

2.19

Two possible options are shown below:



CD \ AB	00	01	11	10
00	X	0	1	1
01	X	X	1	0
11	0	X	1	1
10	X	0	X	X

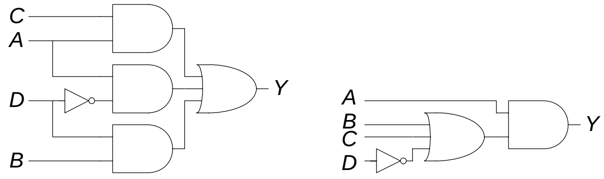
(a) $Y = A\overline{D} + AC + BD$

CD \ AB	00	01	11	10
00	X	0	1	1
01	X	X	1	0
11	0	X	1	1
10	X	0	X	X

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

2.20

Two possible options are shown below:



(a) (b)

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).

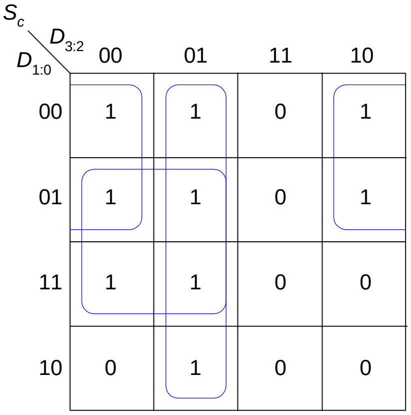
Option (b) does not have a glitch. Only one path exists from any given input to the output.

2.22

The equation can be written directly from the description:

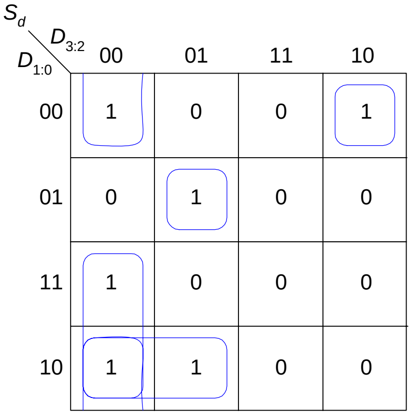
$$E = S\overline{A} + AL + H$$

2.23 (a)



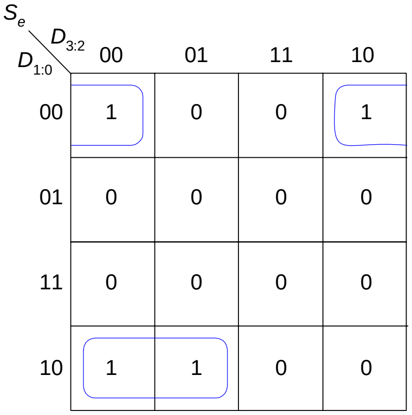
S_c: $D_{1:0}$ \ $D_{3:2}$	00	01	11	10
00	1	1	0	1
01	1	1	0	1
11	1	1	0	0
10	0	1	0	0

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



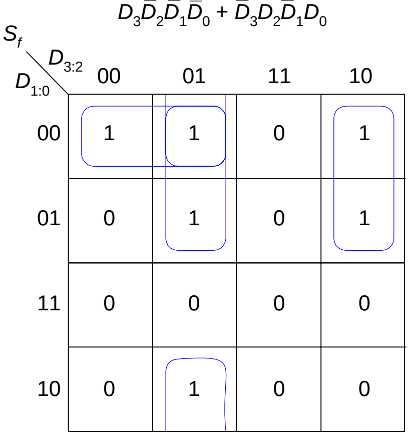
S_d: $D_{1:0}$ \ $D_{3:2}$	00	01	11	10
00	1	0	0	1
01	0	1	0	0
11	1	0	0	0
10	1	1	0	0

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



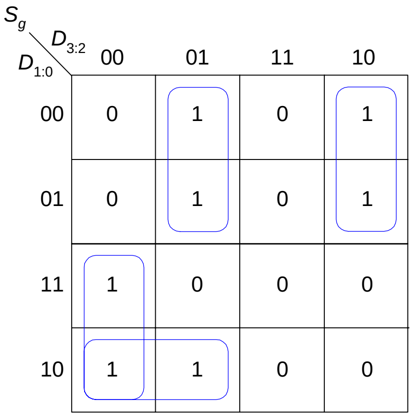
S_e: $D_{1:0}$ \ $D_{3:2}$	00	01	11	10
00	1	0	0	1
01	0	0	0	0
11	0	0	0	0
10	1	1	0	0

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



S_f: $D_{1:0}$ \ $D_{3:2}$	00	01	11	10
00	1	1	0	1
01	0	1	0	1
11	0	0	0	0
10	0	1	0	0

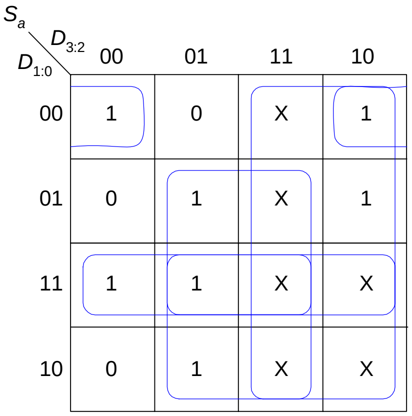
$$S_f = \overline{D}_3 \overline{D}_1 \overline{D}_0 + \overline{D}_3 D_2 \overline{D}_1 + \overline{D}_3 D_2 \overline{D}_0 + D_3 \overline{D}_2 \overline{D}_1$$



S_g $D_{1:0}$ \ $D_{3:2}$	00	01	11	10
00	0	1	0	1
01	0	1	0	1
11	1	0	0	0
10	1	1	0	0

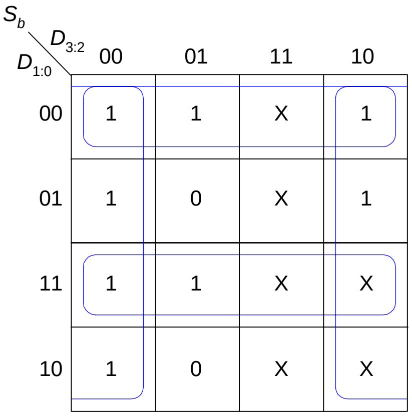
$$S_g = \overline{D_3}\,\overline{D_2}D_1 + \overline{D_3}D_1\overline{D_0} + \overline{D_3}D_2\overline{D_1} + D_3\overline{D_2}\,\overline{D_1}$$

(b)



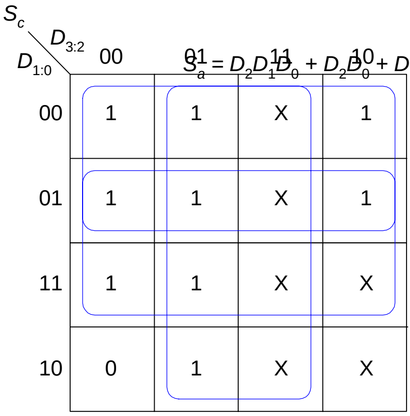
S_a $D_{1:0}$ \ $D_{3:2}$	00	01	11	10
00	1	0	X	1
01	0	1	X	1
11	1	1	X	X
10	0	1	X	X

$S_a = \overline{D}_2\overline{D}_1\overline{D}_0 + D_2D_0 + D_3 + D_2D_1 + D_1D_0$



S_b $D_{1:0}$ \ $D_{3:2}$	00	01	11	10
00	1	1	X	1
01	1	0	X	1
11	1	1	X	X
10	1	0	X	X

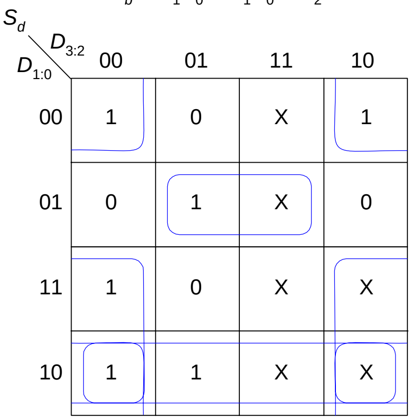
$S_b = \overline{D}_1\overline{D}_0 + D_1D_0 + \overline{D}_2$



$S_a = D_2D_1D_0 + D_2D_0 + D_3 + D_1$

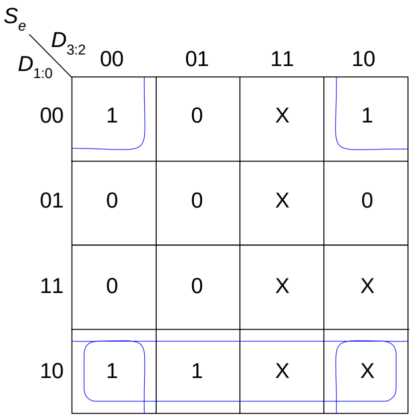
S_c $D_{1:0}$ \ $D_{3:2}$	00	01	11	10
00	1	1	X	1
01	1	1	X	1
11	1	1	X	X
10	0	1	X	X

$S_c = \overline{D}_1 + D_0 + D_2$



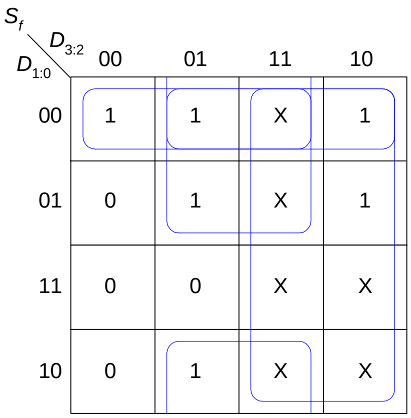
S_d $D_{1:0}$ \ $D_{3:2}$	00	01	11	10
00	1	0	X	1
01	0	1	X	0
11	1	0	X	X
10	1	1	X	X

$S_d = D_2\overline{D}_1D_0 + \overline{D}_2\overline{D}_0 + \overline{D}_2D_1 + D_1\overline{D}_0$



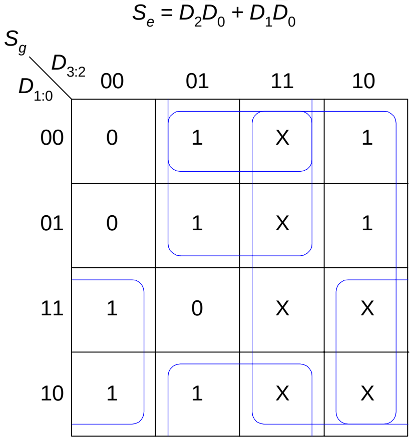
S_e $D_{1:0}$ \ $D_{3:2}$	00	01	11	10
00	1	0	X	1
01	0	0	X	0
11	0	0	X	X
10	1	1	X	X

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



S_f $D_{1:0}$ \ $D_{3:2}$	00	01	11	10
00	1	1	X	1
01	0	1	X	1
11	0	0	X	X
10	0	1	X	X

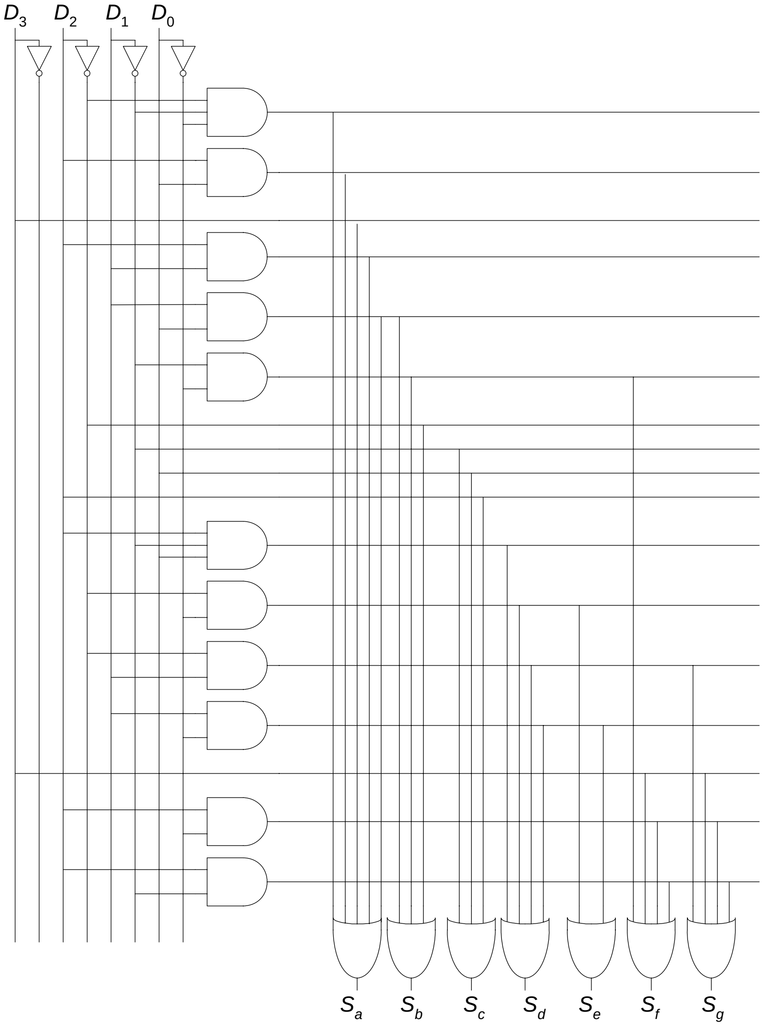
$S_f = \overline{D}_1\overline{D}_0 + D_2\overline{D}_1 + D_2\overline{D}_0 + D_3$



S_g $D_{1:0}$ \ $D_{3:2}$	00	01	11	10
00	0	1	X	1
01	0	1	X	1
11	1	0	X	X
10	1	1	X	X

$S_g = \overline{D}_2 D_1 + D_2\overline{D}_0 + D_2\overline{D}_1 + D_3$

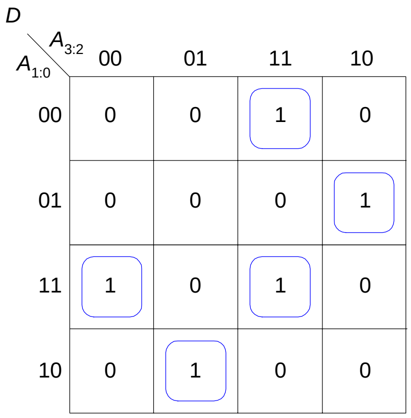
(c)



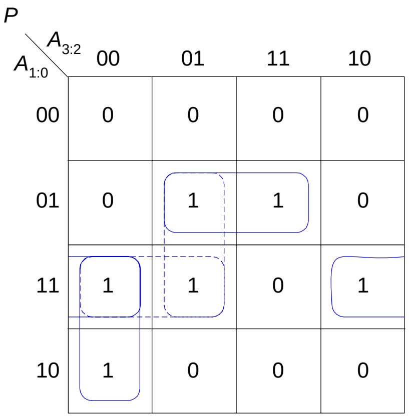
2.24

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

P has two possible minimal solutions:



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



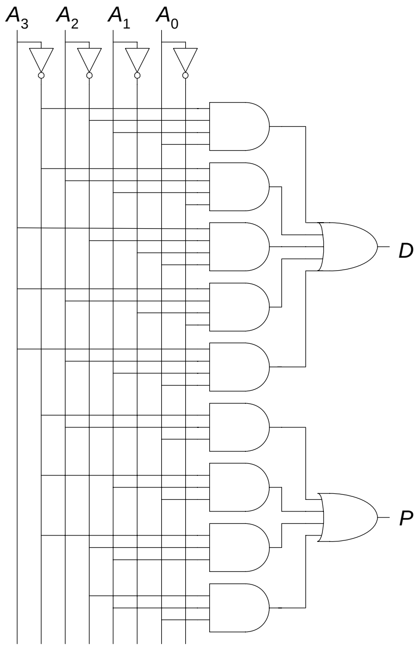
$A_{1:0}$ \ $A_{3:2}$	00	01	11	10
00	0	0	0	0
01	0	1	1	0
11	1	1	0	1
10	1	0	0	0

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

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

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

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



2.25

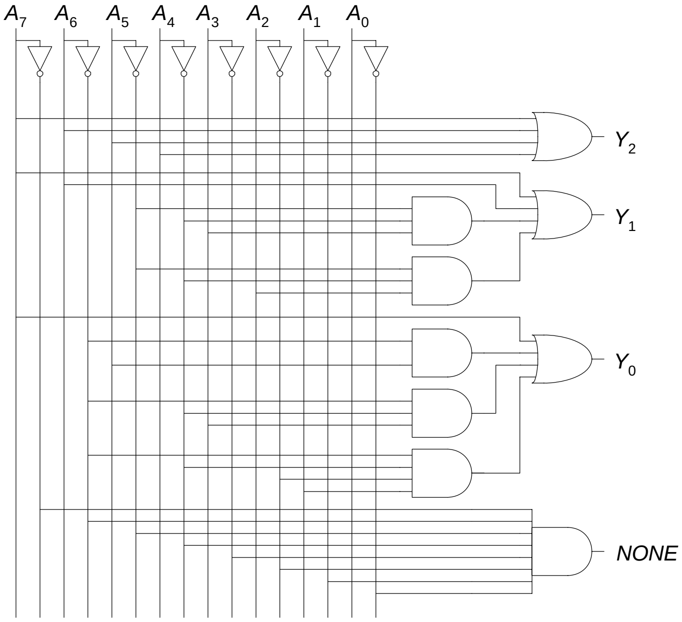
A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Y_2	Y_1	Y_0	$NONE$
0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	1	X	0	0	1	0
0	0	0	0	0	1	X	X	0	1	0	0
0	0	0	0	1	X	X	X	0	1	1	0
0	0	0	1	X	X	X	X	1	0	0	0
0	0	1	X	X	X	X	X	1	0	1	0
0	1	X	X	X	X	X	X	1	1	0	0
1	X	X	X	X	X	X	X	1	1	1	0

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

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

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

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



2.26

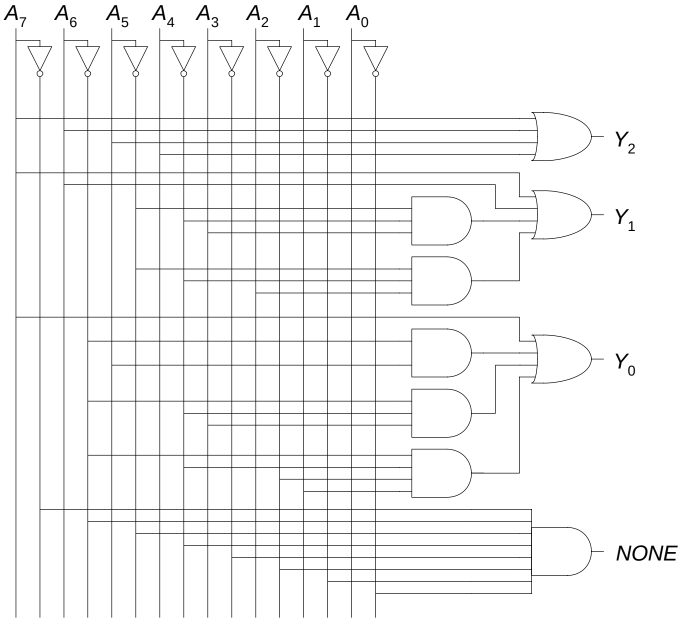
The equations and circuit for $Y_{2:0}$ is the same as in Exercise 2.25, repeated here for convenience.

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

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

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

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


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

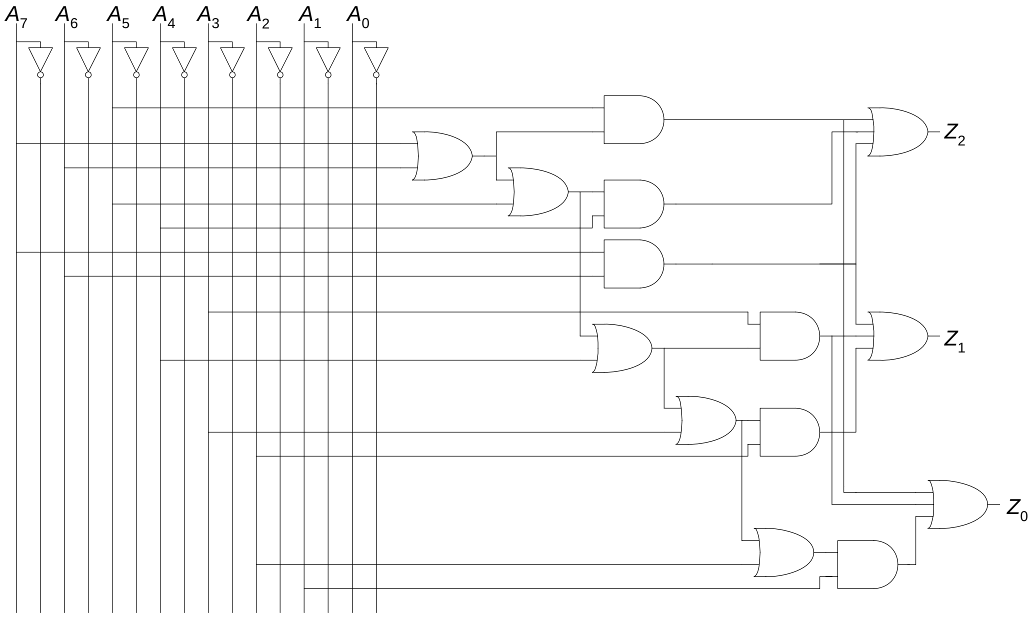
The truth table, equations, and circuit for $Z_{2:0}$ are as follows.

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

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

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

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



2.27

$$Y_6 = A_2A_1A_0$$

$$Y_5 = A_2A_1$$

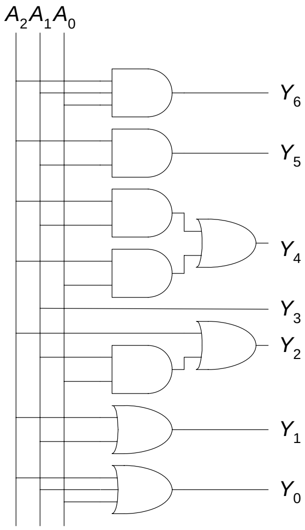
$$Y_4 = A_2A_1 + A_2A_0$$

$$Y_3 = A_2$$

$$Y_2 = A_2 + A_1A_0$$

$$Y_1 = A_2 + A_1$$

$$Y_0 = A_2 + A_1 + A_0$$

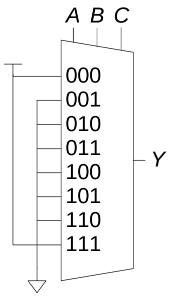


2.28 $Y = A + \overline{C \oplus D} = A + CD + \bar{C}\bar{D}$

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

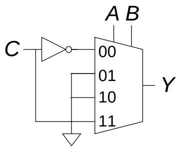
2.30

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



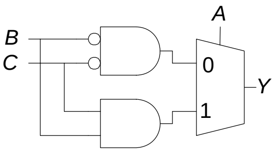
(a)

A	*B*	*Y*
0	0	$\bar{C}$
0	1	0
1	0	0
1	1	*C*



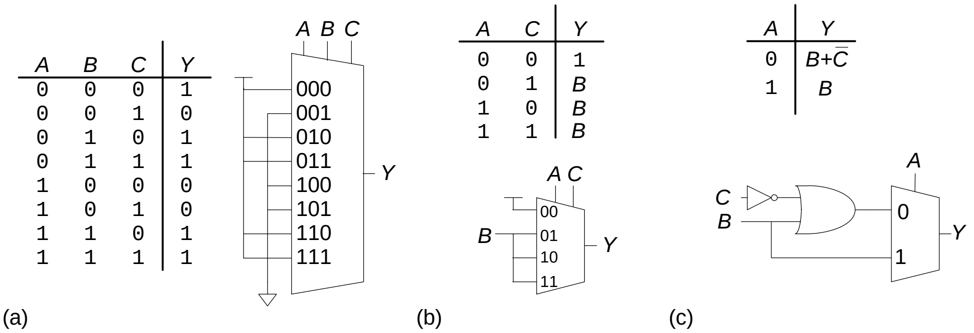
(b)

A	*Y*
0	$\bar{B}\bar{C}$
1	*BC*



(c)

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

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

(a) (b) (c)

2.32

$t_{pd} = t_{pd\_\text{AND2}} + 2t_{pd\_\text{NOR2}} + t_{pd\_\text{NAND2}}$

$= [30 + 2\ (30) + 20]$ ps

$=$ **110 ps**

$t_{cd} = 2t_{cd\_\text{NAND2}} + t_{cd\_\text{NOR2}}$

$= [2\ (15) + 25]$ ps

$=$ **55 ps**

2.33

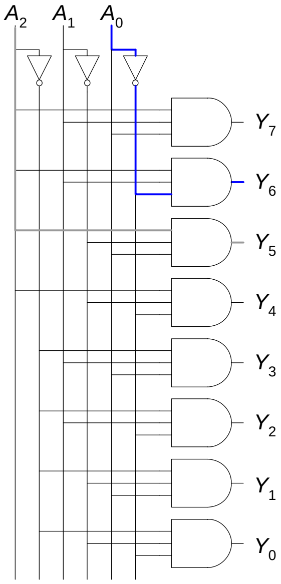
$t_{pd} = t_{pd\_\text{NOT}} + t_{pd\_\text{AND3}}$

$= 15$ ps $+ 40$ ps

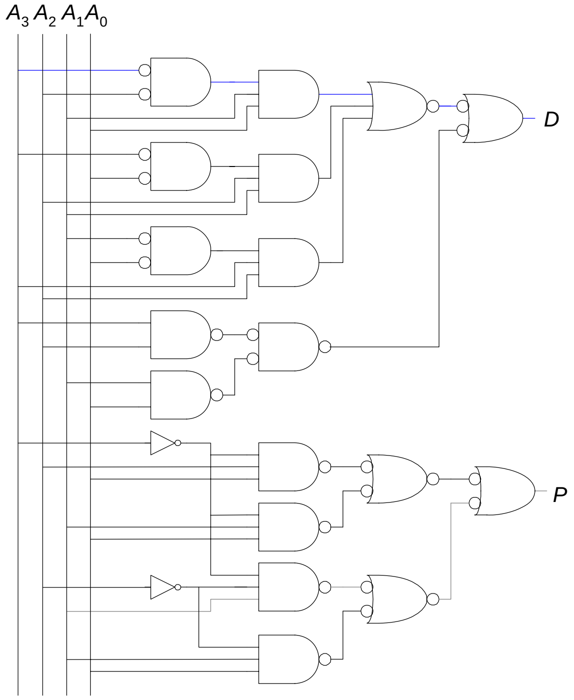
$=$ **55 ps**

$t_{cd} = t_{cd\_\text{AND3}}$

$=$ **30 ps**


A_2
A_1
A_0
Y_7
Y_6
Y_5
Y_4
Y_3
Y_2
Y_1
Y_0

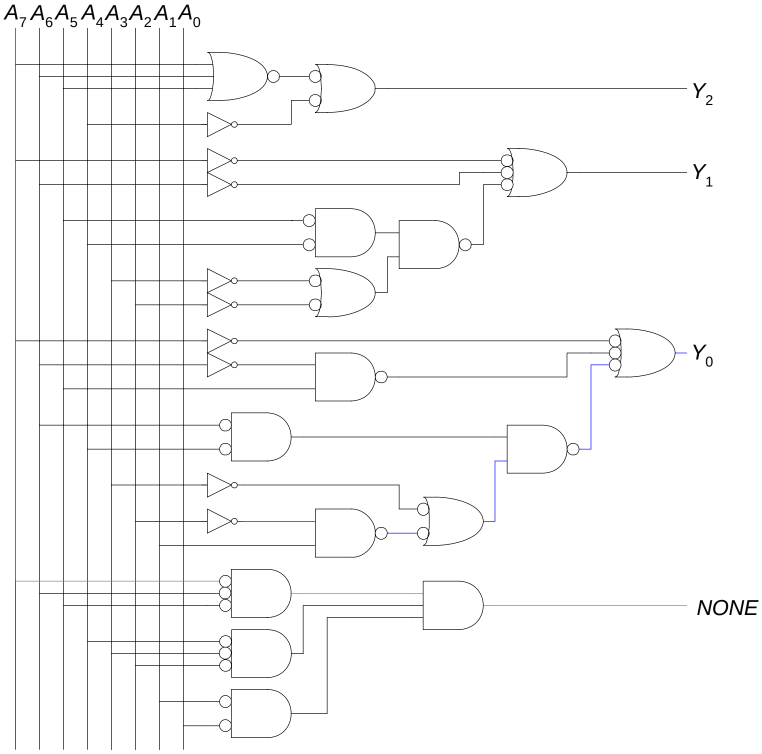
2.34



$t_{pd} = t_{pd}$_NOR2 + t_{pd}_AND3 + t_{pd}_NOR3 + t_{pd}_NAND2
= [30 + 40 + 45 + 20] ps
= **135 ps**

$t_{cd} = 2t_{cd}$_NAND2 + t_{cd}_OR2
= [2 (15) + 30] ps
= **60 ps**

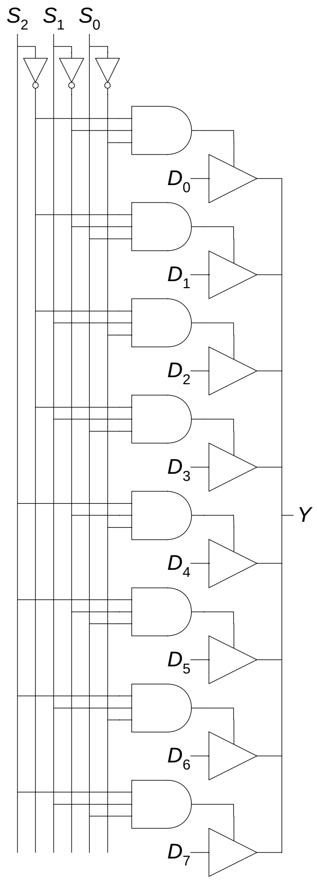
2.35



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

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

2.36



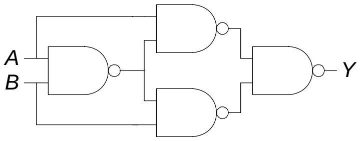
$t_{pd\_dy} = t_{pd\_\text{TRI\_AY}}$
$= \mathbf{50\ ps}$

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

$t_{pd\_sy} = t_{pd\_\text{NOT}} + t_{pd\_\text{AND3}} + t_{pd\_\text{TRI\_SY}}$
$= [30 + 80 + 35]\ \text{ps}$
$= \mathbf{145\ ps}$

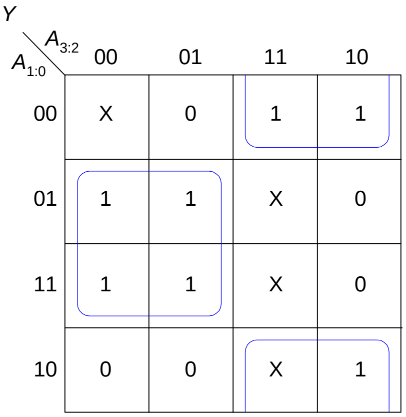
However, the problem specified to minimize the delay from data inputs to output, t_{pd_dy}.

Question 2.1



Question 2.2

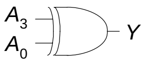
Month	A_3	A_2	A_1	A_0	Y
Jan	0	0	0	1	1
Feb	0	0	1	0	0
Mar	0	0	1	1	1
Apr	0	1	0	0	0
May	0	1	0	1	1
Jun	0	1	1	0	0
Jul	0	1	1	1	1
Aug	1	0	0	0	1
Sep	1	0	0	1	0
Oct	1	0	1	0	1
Nov	1	0	1	1	0
Dec	1	1	0	0	1



Y

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

$$Y = \bar{A}_3A_0 + A_3\bar{A}_0 = A_3 \oplus A_0$$



Question 2.3

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

Question 2.4

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

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

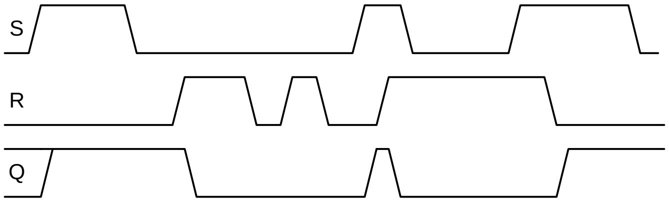
(c) The NAND gate by itself is universal. A NAND gate with its inputs tied together performs the NOT operation. A NAND gate with a NOT gate on its output performs AND. And a NAND gate with NOT gates on its inputs performs OR. Thus, a NAND gate is equivalent to the set {AND, OR, NOT} and is universal.

Question 2.5

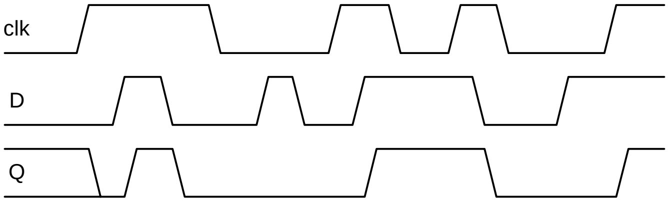
A circuit's contamination delay might be less than its propagation delay because the circuit may operate over a range of temperatures and supply voltages, for example, 3-3.6 V for LVCMOS (low voltage CMOS) chips. As temperature increases and voltage decreases, circuit delay increases. Also, the circuit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

CHAPTER 3

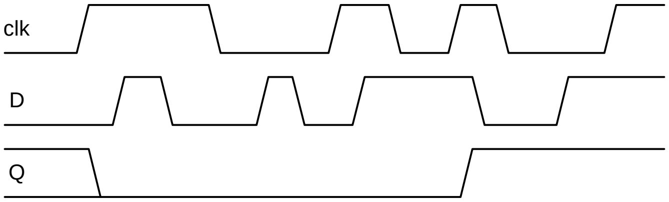
3.1



3.2



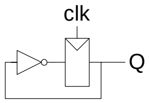
3.3



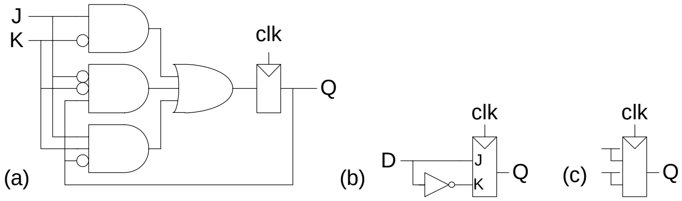
3.4 The circuit is sequential because it involves feedback and the output depends on previous values of the inputs. This is a SR latch. When $\overline{S}$ = 0 and $\overline{R}$ = 1, the circuit sets Q to 1. When $\overline{S}$ = 1 and $\overline{R}$ = 0, the circuit resets Q to 0. When both $\overline{S}$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

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.)

3.6

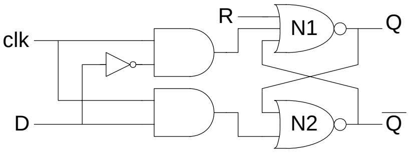


3.7

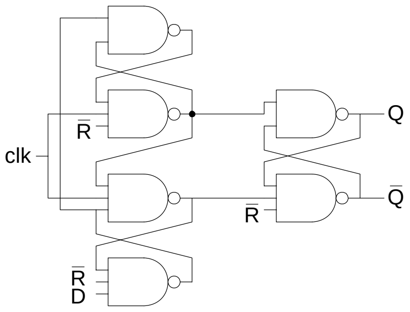


3.8 If A and B have the same value, C takes on that value. Otherwise, C retains its old value.

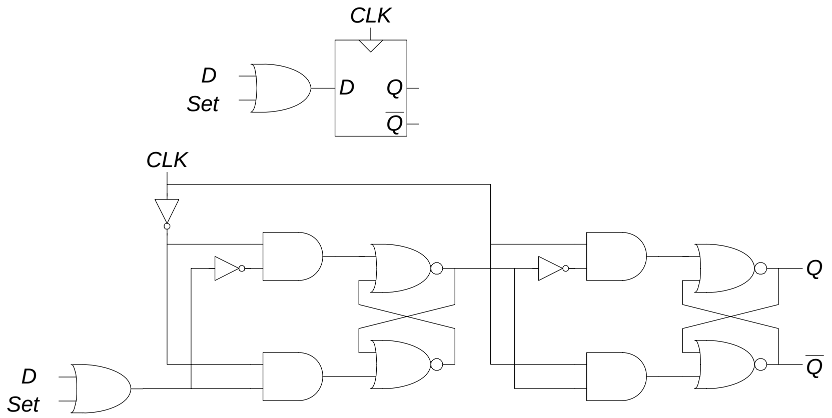
3.9 Make sure these next ones are correct too.



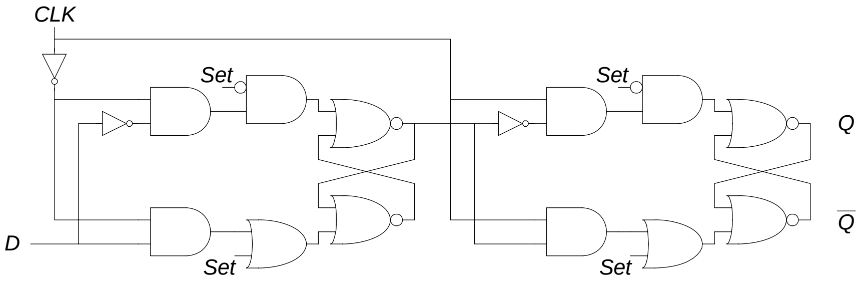
3.10



3.11



3.12



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

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

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

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

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

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

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

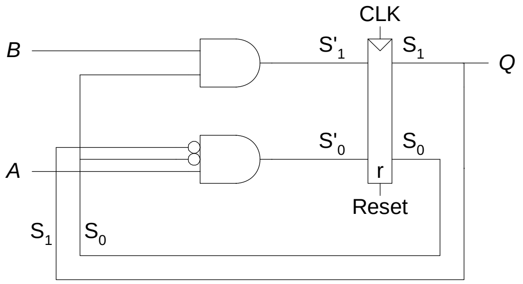
.

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

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

$$Q = S_1$$



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

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

TABLE 3.4 State encoding for Exercise 3.20

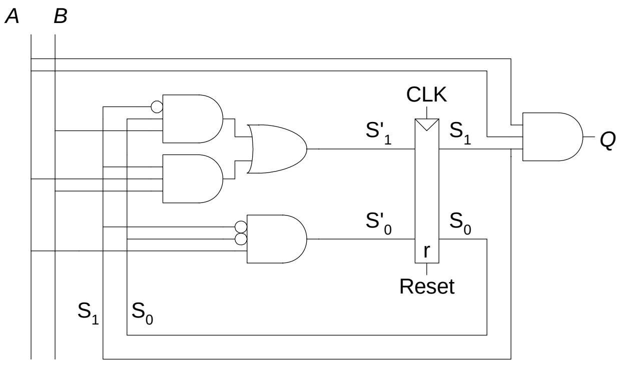
current state		inputs		next state		output
s_1	s_0	a	b	s'_1	s'_0	q
0	0	0	X	0	0	0
0	0	1	X	0	1	0
0	1	X	0	0	0	0
0	1	X	1	1	0	0
1	0	1	1	1	0	1
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0

TABLE 3.5 Combined state transition and output table with binary encodings for Exercise 3.20

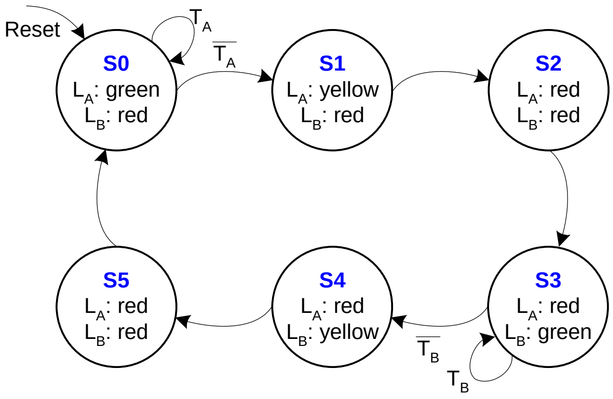
$$S'_1 = \overline{S_1}S_0B + S_1AB$$

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

$$Q' = S_1AB$$



3.21



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

TABLE 3.6 State encoding for Exercise 3.21

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

TABLE 3.6 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 3.7 State transition table with binary encodings for Exercise 3.21

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

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

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

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

TABLE 3.8 Output table for Exercise 3.21

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

TABLE 3.8 Output table for Exercise 3.21

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

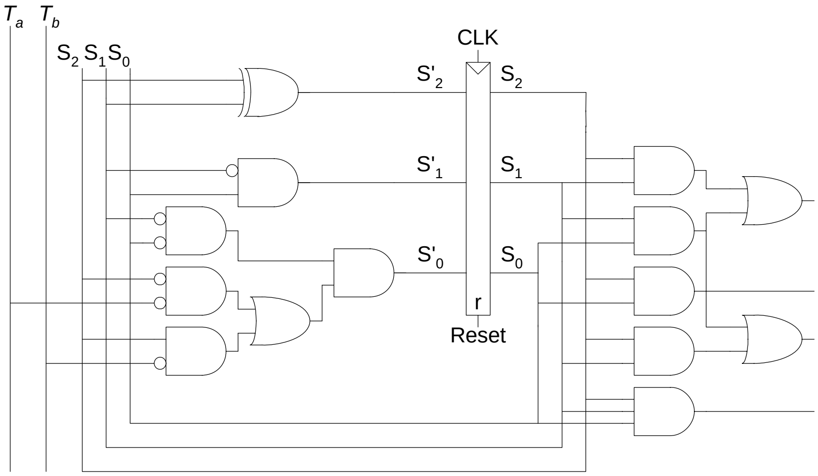
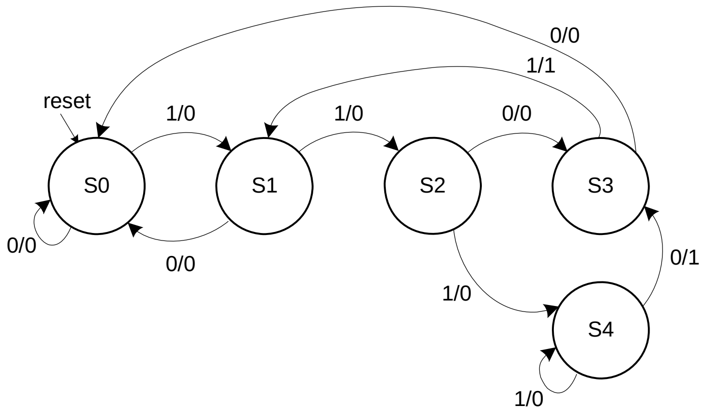


FIGURE 3.1 State machine circuit for traffic light controller for Exercise 3.21

3.22



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

TABLE 3.9 State encoding for Exercise 3.22

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

TABLE 3.10 Combined state transition and output table with binary encodings for Exercise 3.22

current state			input	next state			output
s 2	*s* 1	*s* 0	*a*	*s* ' 2	*s* ' 1	*s* ' 0	q
0	1	0	0	1	0	0	0
0	1	0	1	1	0	1	0
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	1
1	0	1	0	1	0	0	1
1	0	1	1	1	0	1	0

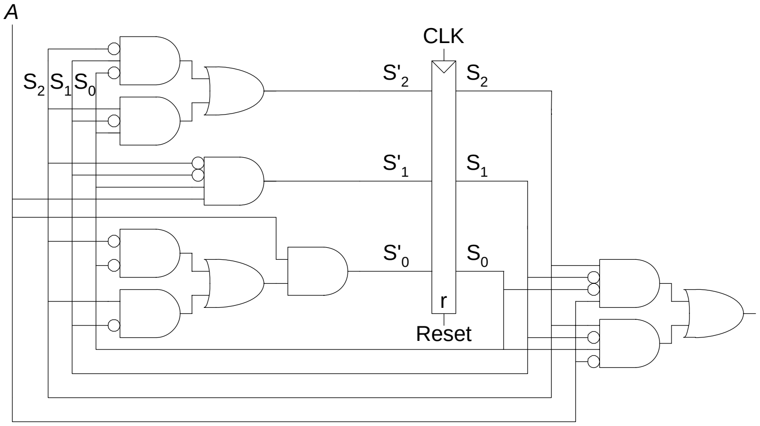
TABLE 3.10 Combined state transition and output table with binary encodings for Exercise 3.22

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

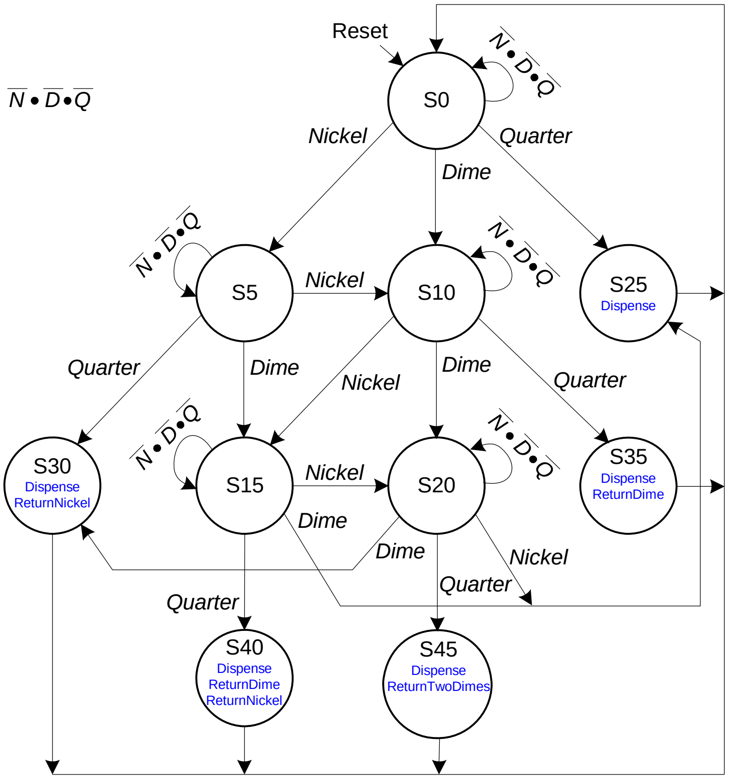
$$S'_1 = \overline{S_2}\,\overline{S_1}S_0A$$

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

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



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 3.11 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 3.11 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 3.12 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

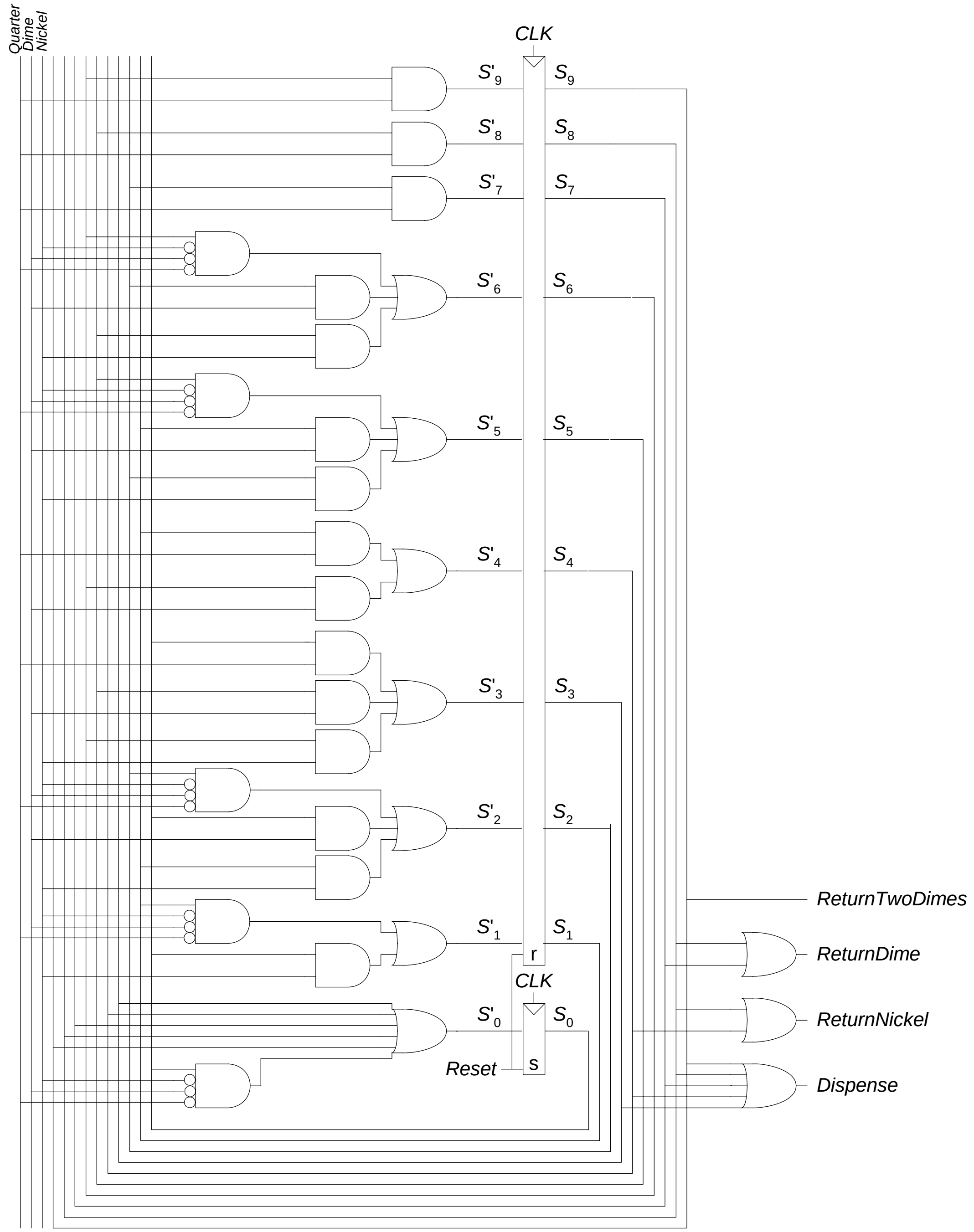
TABLE 3.12 State transition table for Exercise 3.23

$$S'_9 = S_6 Q$$

$$S'_8 = S_5 Q$$

$$
\begin{aligned}
S'_7 &= S_2Q \\
S'_6 &= S_2D + S_5N + S_6\overline{N}\overline{D}\overline{Q} \\
S'_5 &= S_1D + S_2N + S_5NDQ \\
S'_4 &= S_1Q + S_6D \\
S'_3 &= S_0Q + S_5D + S_6N \\
S'_2 &= S_0D + S_1N + S_2\overline{N}\overline{D}\overline{Q} \\
S'_1 &= S_0N + S_1NDQ \\
S'_0 &= S_0\overline{N}\overline{D}\overline{Q} + S_3 + S_4 + S_7 + S_8 + S_9
\end{aligned}
$$

$$
\begin{aligned}
Dispense &= S_3 + S_4 + S_7 + S_8 + S_9 \\
ReturnNickel &= S_4 + S_8 \\
ReturnDime &= S_7 + S_8 \\
ReturnTwoDimes &= S_9
\end{aligned}
$$


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
s
Reset
ReturnTwoDimes
ReturnDime
ReturnNickel
Dispense

3.24

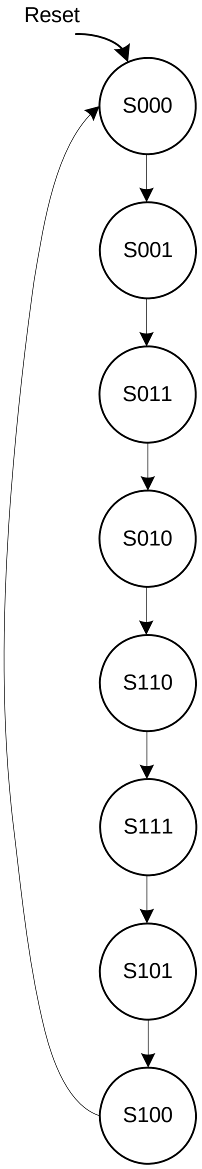


FIGURE 3.4 State transition diagram for Exercise 3.24

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

TABLE 3.13 State transition table for Exercise 3.24

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

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

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

$$Q_2 = S_2$$

$$Q_1 = S_1$$

$$Q_0 = S_0$$

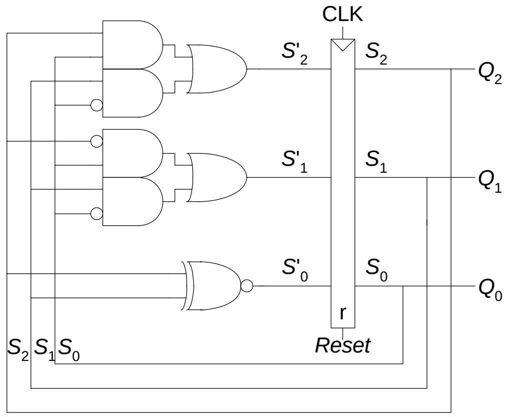


FIGURE 3.5 Hardware for Gray code counter FSM for Exercise 3.24

3.25

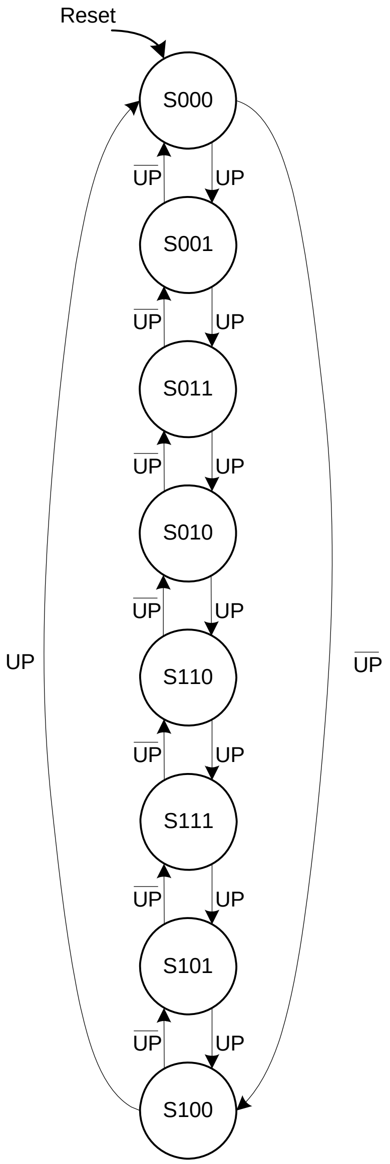


FIGURE 3.6 State transition diagram for Exercise 3.25

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 3.14 State transition table for Exercise 3.25

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

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

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

$$Q_2 = S_2$$

$$Q_1 = S_1$$

$$Q_0 = S_0$$

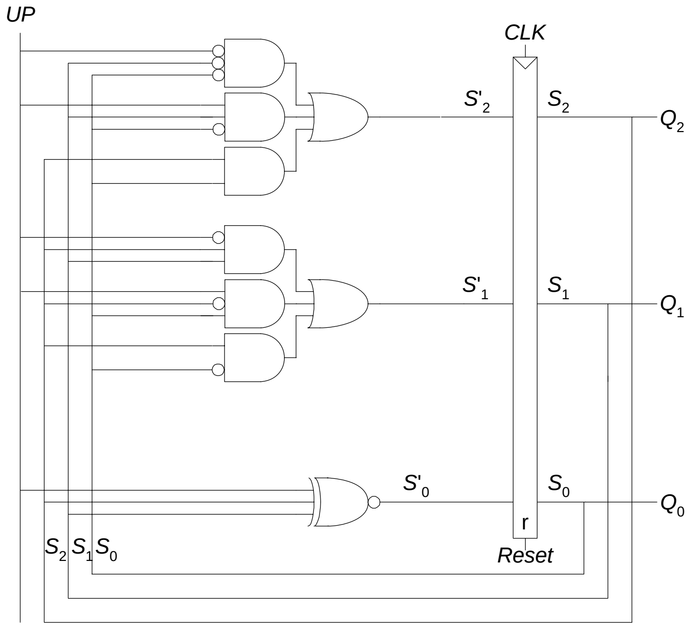


FIGURE 3.7 Finite state machine hardware for Exercise 3.25

3.26

(a)

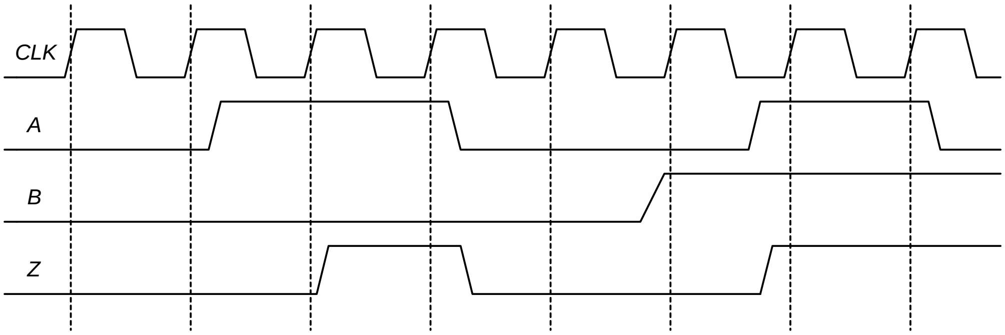


FIGURE 3.8 Waveform showing *Z* output for Exercise 3.26

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

(c)

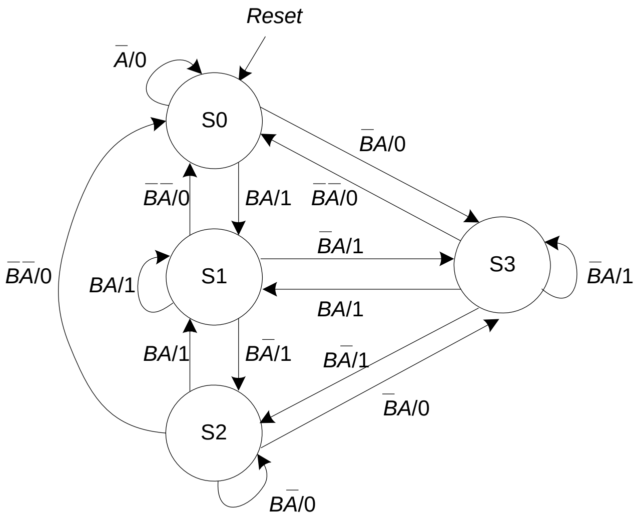


FIGURE 3.9 State transition diagram for Exercise 3.26

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

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

TABLE 3.15 State transition table for Exercise 3.26

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

TABLE 3.15 State transition table for Exercise 3.26

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

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

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

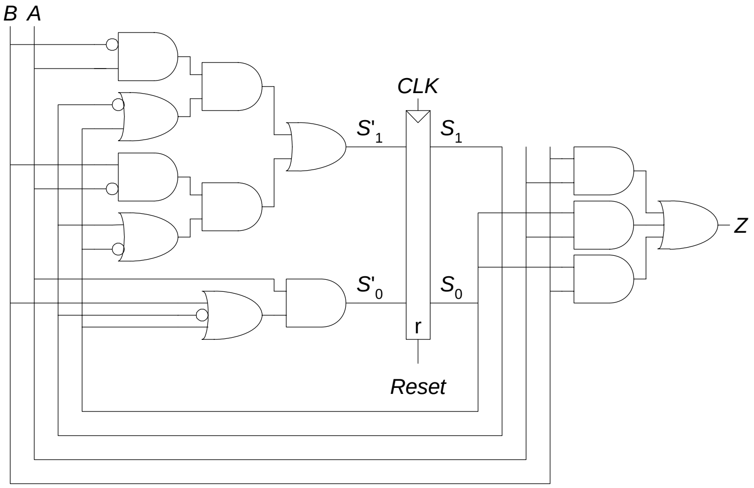


FIGURE 3.10 Hardware for FSM of Exercise 3.26

Note: One could also build this functionality by registering input *A*, producing both the logical AND and OR of input *A* and its previous (registered) value, and then muxing the two operations using *B*. The output of the mux is *Z*: *Z* = *A*Aprev (if *B* = 0); *Z* = *A* + Aprev (if *B* = 1).

3.27

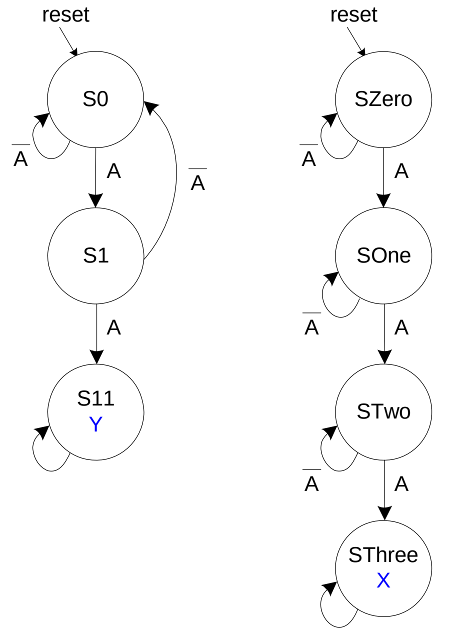


FIGURE 3.11 Factored state transition diagram for Exercise 3.27

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 3.16 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 3.17 State transition table for output *X* for Exercise 3.27

$$S'_1 = S_0(S_1 + A)$$

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

$$Y = S_1$$

$$T'_1 = T_1 + T_0A$$

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

$$X = T_1T_0$$

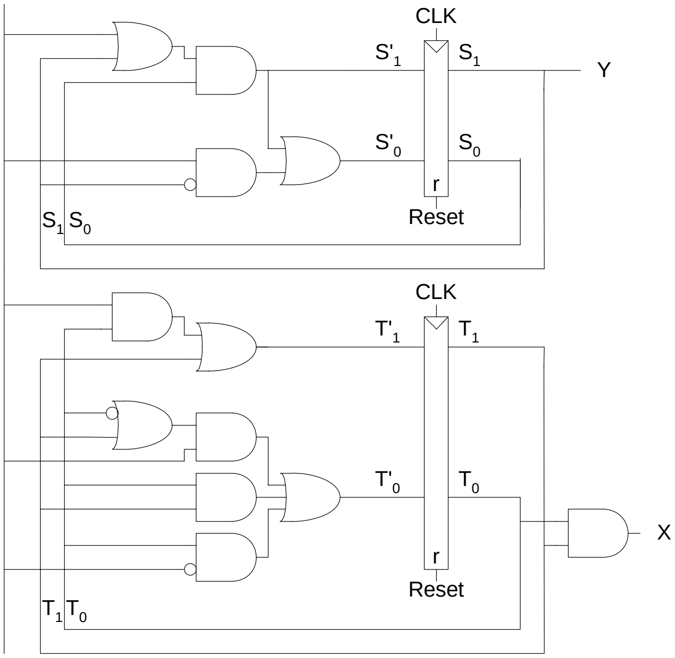


FIGURE 3.12 Finite state machine hardware for Exercise 3.27

3.28

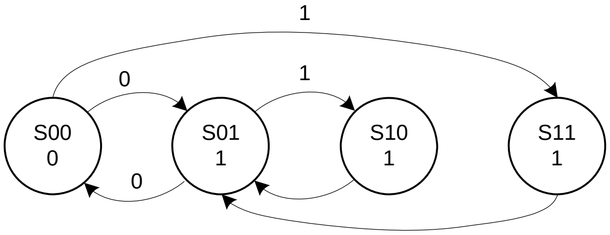
This finite state machine is a divide-by-two counter (see Section 3.4.2) when X = 0. When X = 1, the output, *Q*, is HIGH.

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

TABLE 3.18 State transition table with binary encodings for Exercise 3.28

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

TABLE 3.19 Output table for Exercise 3.28



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 3.20 State transition table with binary encodings for Exercise 3.29

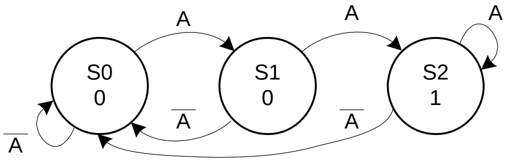


FIGURE 3.13 State transition diagram for Exercise 3.29

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

3.30

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

$t_{pd} = 3t_{pd\_\mathrm{XOR}}$
$= 3 \times 100$ ps
$=$ **300 ps**

Next, we calculate the cycle time:

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

(b)

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

Thus,

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

(c)

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

$t_{cd} \ = t_{cd\_\mathrm{XOR}}$
$= 55$ ps

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

Thus,

$t_{\mathrm{skew}} < (t_{ccq} + t_{cd}) - t_{\mathrm{hold}}$
$< (50 + 55) - 20$
< 85 ps

(d)

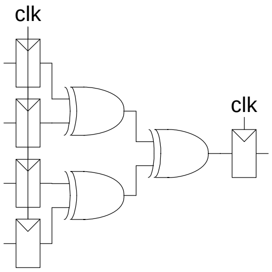


FIGURE 3.14 Alyssa's improved circuit for Exercise 3.30

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

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

Next, we calculate the cycle time:

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

(b)

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

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

3.32
(a) $T_c = 1 / 40$ MHz $= 25$ ns
$T_c \geq t_{pcq} + Nt_{\text{CLB}} + t_{\text{setup}}$
25 ns $\geq [0.72 + N(0.61) + 0.53]$ ps
Thus, N < 38.9

N = 38

(b)

$t_{\text{skew}} < (t_{ccq} + t_{cd\_\text{CLB}}) - t_{\text{hold}}$

$< [(0.5 + 0.3) - 0]$ ns

< 0.8 ns = 800 ps

3.33 1.138 ns

3.34

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

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

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

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

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

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

3.35

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

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

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

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

Thus,

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

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

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

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

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

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

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

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

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

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

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

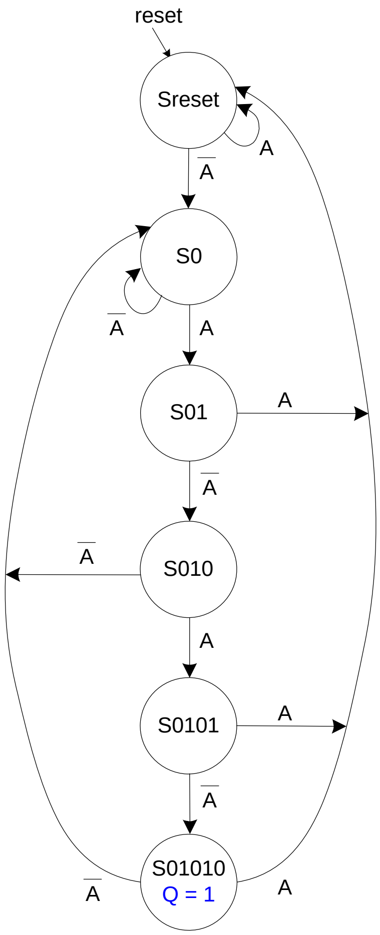


FIGURE 3.15 State transition diagram for Question 3.1

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

TABLE 3.21 State transition table for Question 3.1

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

TABLE 3.21 State transition table for Question 3.1

$$S'_5 = S_4A$$

$$S'_4 = S_3A$$

$$S'_3 = S_2A$$

$$S'_2 = S_1A$$

$$S'_1 = A(S_1 + S_3 + S_5)$$

$$S'_0 = A(S_0 + S_2 + S_4 + S_5)$$

$$Q = S_5$$

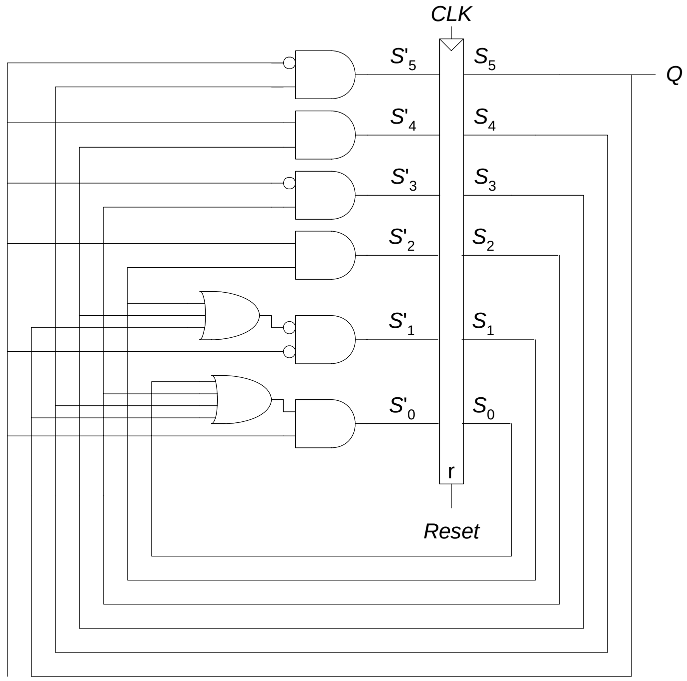


FIGURE 3.16 Finite state machine hardware for Question 3.1

Question 3.2

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

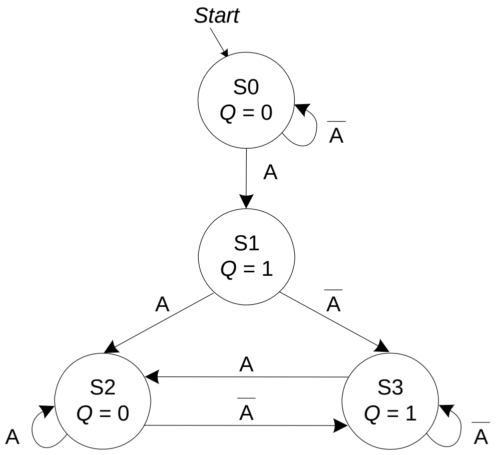


FIGURE 3.17 State transition diagram for Question 3.2

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

TABLE 3.22 State transition table for Question 3.2

$$S'_1 = S_1 + S_0$$

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

$$Q = S_0$$

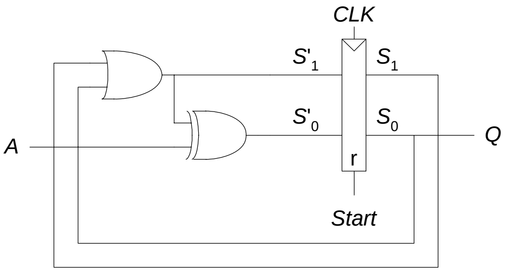


FIGURE 3.18 Finite state machine hardware for Question 3.2

Question 3.3

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

Question 3.4

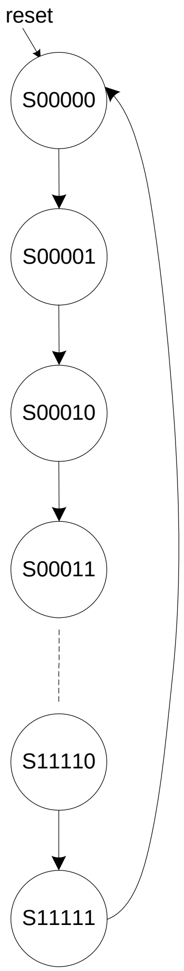


FIGURE 3.19 State transition diagram for Question 3.4

current state $S_{4:0}$	next state $S'_{4:0}$
00000	00001
00001	00010

TABLE 3.23 State transition table for Question 3.4

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

TABLE 3.23 State transition table for Question 3.4

$$S'_4 = S_4 \oplus S_3S_2S_1S_0$$

$$S'_3 = S_3 \oplus S_2S_1S_0$$

$$S'_2 = S_2 \oplus S_1S_0$$

$$S'_1 = S_1 \oplus S_0$$

$$S'_0 = \overline{S_0}$$

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

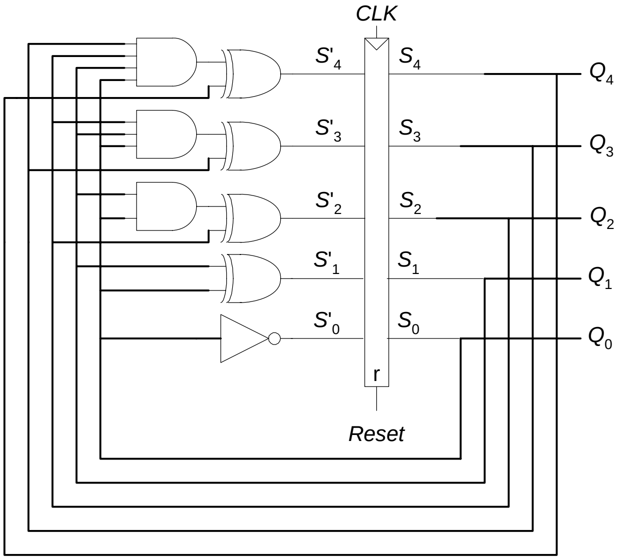


FIGURE 3.20 Finite state machine hardware for Question 3.4

Question 3.5

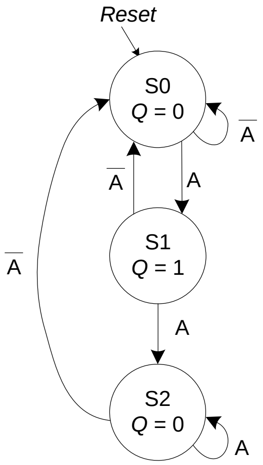


FIGURE 3.21 State transition diagram for edge detector circuit of Question 3.5

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

TABLE 3.24 State transition table for Question 3.5

$$S'_1 = AS_1$$

$$S'_0 = AS_1S_0$$

$$Q = S_1$$

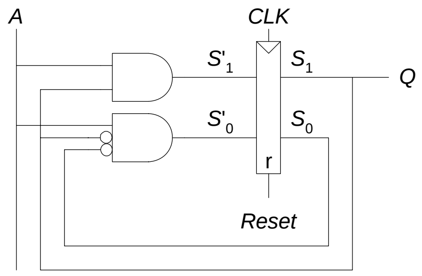


FIGURE 3.22 Finite state machine hardware for Question 3.5

Question 3.6

Pipelining divides a block of combinational logic into *N* stages, with a register between each stage. Pipelining increases throughput, the number of tasks that can be completed in a given amount of time. Ideally, pipelining increases throughput by a factor of *N*. But because of the following three reasons, the speedup is usually less than *N*: (1) The combinational logic usually cannot be divided into N equal stages. (2) Adding registers between stages adds delay called the *sequencing overhead*, the time it takes to get the signal into and out of the register, $t_{\text{setup}} + t_{pcq}$. (3) The pipeline is not always operating at full capacity: at the beginning of execution, it takes time to fill up the pipeline, and at the end it takes time to drain the pipeline. However, pipelining offers significant speedup at the cost of little extra hardware.

Question 3.7

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

Question 3.8

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

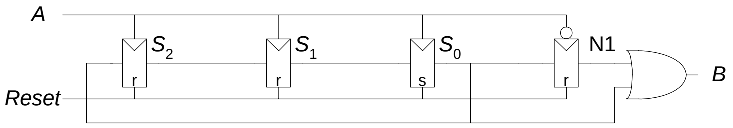


FIGURE 3.23 Hardware for Question 3.8

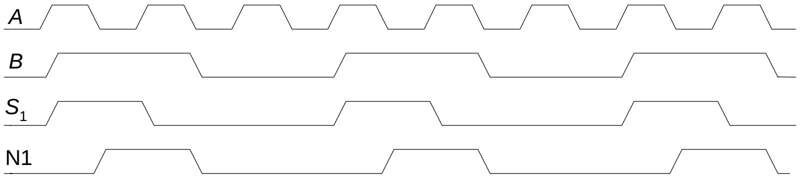


FIGURE 3.24 Waveforms for Question 3.8

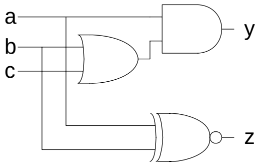
Question 3.9

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

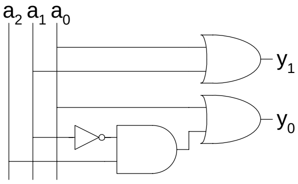
CHAPTER 4

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



4.2



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;
```

4.4

ex4_4.tv file:

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

Verilog

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

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

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

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

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

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

VHDL

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

entity ex4_4_testbench is -- no inputs or outputs
end;

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

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

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

(*VHDL continued on next page*)

(*continued from previous page*)

VHDL

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

      a <= testvectors(vectornum)(4 downto 1)
        after 1 ns;
      yexpected <= testvectors(vectornum)(0)
        after 1 ns;
    end if;
  end process;

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

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;
```

4.6

Verilog

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

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

VHDL

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

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

architecture synth of seven_seg_decoder is
begin
  process(data) begin
    case data is
--                                abcdefg
      when X"0"   => segments <= "1111110";
      when X"1"   => segments <= "0110000";
      when X"2"   => segments <= "1101101";
      when X"3"   => segments <= "1111001";
      when X"4"   => segments <= "0110011";
      when X"5"   => segments <= "1011011";
      when X"6"   => segments <= "1011111";
      when X"7"   => segments <= "1110000";
      when X"8"   => segments <= "1111111";
      when X"9"   => segments <= "1111011";
      when X"A"   => segments <= "1110111";
      when X"B"   => segments <= "0011111";
      when X"C"   => segments <= "0001101";
      when X"D"   => segments <= "0111101";
      when X"E"   => segments <= "1001111";
      when X"F"   => segments <= "1000111";
      when others => segments <= "0000000";
    end case;
  end process;
end;
```

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*)

4.8

Verilog

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

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

VHDL

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

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

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

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;
```

4.10

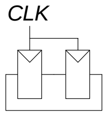
Verilog

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

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

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

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

VHDL

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

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

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

library IEEE; use IEEE.STD_LOGIC_1164.all;

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

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

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



4.12

Verilog

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

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

VHDL

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

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

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

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;
```

4.14

Verilog

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

  wire [11:0] y2_4;

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

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

VHDL

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

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

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

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

(continued on next page)

(continued from previous page)

Verilog

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

VHDL

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

4.15

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

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

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

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;
```

4.16

Verilog

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

  wire n1;

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

VHDL

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

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

architecture behave of ex4_16 is
signal n1: STD_LOGIC;
begin
  n1 <= not(not((not(a and b and c)) and d) or
      (not(e or (f and g))));
  y <= not (n1 and n1);
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;
```

4.18

Verilog

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

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

VHDL

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

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

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

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;
```

4.20

Verilog

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

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

VHDL

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

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

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

4.21

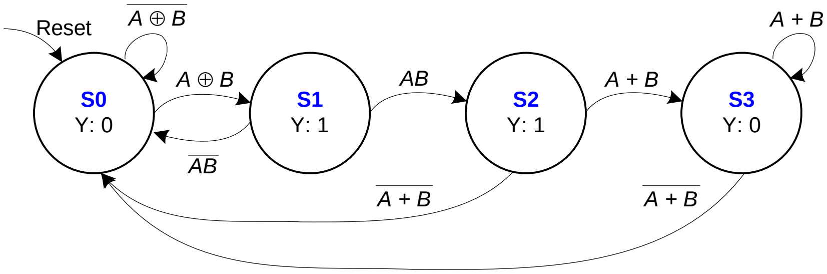
Verilog

```
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;
```

4.22



4.23

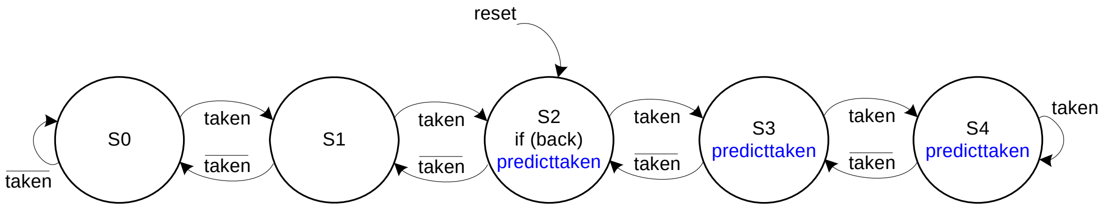

reset
S0
S1
S2
if (back)
predicttaken
S3
predicttaken
S4
predicttaken
taken
taken
taken
taken
taken
$\overline{\text{taken}}$
$\overline{\text{taken}}$
$\overline{\text{taken}}$
$\overline{\text{taken}}$
$\overline{\text{taken}}$

FIGURE 4.1 State transition diagram for Exercise 4.23

4.24

Verilog

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

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

VHDL

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

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

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

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;
```

4.26

Verilog

```
module latch3_18(input d, clk, output q);
  wire n1, n2, clk_b;

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

VHDL

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

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

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

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

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;
```

4.28 Mode Module

Verilog

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

  reg state, nextstate;

  parameter S0 = 1'b0;
  parameter S1 = 1'b1;

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

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

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

VHDL

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

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

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

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

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

(continued on next page)

Lights Module

Verilog

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

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

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

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

(continued on next page)

Controller Module

Verilog

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

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

VHDL

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

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

architecture bad of syncbad is
  signal n1: STD_LOGIC;
begin
  process(clk)
  begin
    if clk'event and clk = '1' then
      q  <= n1; -- nonblocking
      n1 <= d;  -- nonblocking
    end if;
  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;
```

4.30

Verilog

```
module fig3_65(input  clk, reset, a, b,
               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 (b) nextstate = S2;
          else   nextstate = S0;
      S2:        nextstate = S0;
      default:   nextstate = S0;
    endcase

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

VHDL

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

entity fig3_65 is
  port(clk, reset, a, b: in  STD_LOGIC;
       q:                out STD_LOGIC);
end;

architecture synth of fig3_65 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 =>      nextstate <= S0;
      when others =>  nextstate <= S0;
    end case;
  end process;

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

4.32

Verilog

```
module ex4_32(input  clk, reset, ta, tb,
              output reg [1:0] la, lb);

  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;
  parameter S5 = 3'b101;

  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:         nextstate = S3;
      S3: if (tb) nextstate = S3;
          else    nextstate = S4;
      S4:         nextstate = S5;
      S5:         nextstate = S0;
    endcase

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

VHDL

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

entity ex4_32 is
  port(clk, reset, ta, tb: in  STD_LOGIC;
       la, lb: out STD_LOGIC_VECTOR(1 downto 0));
end;

architecture synth of ex4_32 is
  type statetype is (S0, S1, S2, S3, S4, S5);
  signal state, nextstate: statetype;
  signal lalb: STD_LOGIC_VECTOR(3 downto 0);
begin
  -- state register
  process(clk, reset) begin
    if reset = '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 =>      nextstate <= S3;
      when S3 => if tb = '1' then
                      nextstate <= S3;
                 else nextstate <= S4;
                 end if;
      when S4 =>      nextstate <= S5;
      when S5 =>      nextstate <= S0;
      when others =>  nextstate <= S0;
    end case;
  end process;

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

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;
```

4.34 *(starting on next page)*

Verilog

```
module ex4_34(input clk, reset, n, d, q,
              output dispense, return5, return10,
              output return2_10);

  reg [3:0] state, nextstate;

  parameter S0 =  4'b0000;
  parameter S5 =  4'b0001;
  parameter S10 = 4'b0010;
  parameter S25 = 4'b0011;
  parameter S30 = 4'b0100;
  parameter S15 = 4'b0101;
  parameter S20 = 4'b0110;
  parameter S35 = 4'b0111;
  parameter S40 = 4'b1000;
  parameter S45 = 4'b1001;

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

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

VHDL

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

entity ex4_34 is
  port(clk, reset, n, d, q: in  STD_LOGIC;
       dispense, return5, return10: out STD_LOGIC;
       return2_10:                  out STD_LOGIC);
end;

architecture synth of ex4_34 is
  type statetype is (S0, S5, S10, S25, S30, S15, S20,
                     S35, S40, S45);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset = '1' then state <= S0;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

 -- next state logic
  process (state, n, d, q) begin
    case state is
      when S0  =>
        if n = '1'    then nextstate <= S5;
        elsif d = '1' then nextstate <= S10;
        elsif q = '1' then nextstate <= S25;
        else               nextstate <= S0;
        end if;
      when S5  =>
        if n = '1'    then nextstate <= S10;
        elsif d = '1' then nextstate <= S15;
        elsif q = '1' then nextstate <= S30;
        else               nextstate <= S5;
        end if;
      when S10 =>
        if n = '1'    then nextstate <= S15;
        elsif d = '1' then nextstate <= S20;
        elsif q = '1' then nextstate <= S35;
        else               nextstate <= S10;
        end if;
      when S25 =>          nextstate <= S0;
      when S30 =>          nextstate <= S0;
      when S15 =>
        if n = '1'    then nextstate <= S20;
        elsif d = '1' then nextstate <= S25;
        elsif q = '1' then nextstate <= S40;
        else               nextstate <= S15;
        end if;
      when S20 =>
        if n = '1'    then nextstate <= S25;
        elsif d = '1' then nextstate <= S30;
        elsif q = '1' then nextstate <= S45;
        else               nextstate <= S20;
        end if;
      when S35 =>          nextstate <= S0;
      when S40 =>          nextstate <= S0;
      when S45 =>          nextstate <= S0;
      when others =>       nextstate <= S0;
    end case;
  end process;
```

(continued from previous page)

Verilog

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

VHDL

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

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;
```

4.36

Verilog

```
module ex4_36(input clk, reset, up,
              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: if (up) nextstate = S1;
          else    nextstate = S7;
      S1: if (up) nextstate = S2;
          else    nextstate = S0;
      S2: if (up) nextstate = S3;
          else    nextstate = S1;
      S3: if (up) nextstate = S4;
          else    nextstate = S2;
      S4: if (up) nextstate = S5;
          else    nextstate = S3;
      S5: if (up) nextstate = S6;
          else    nextstate = S4;
      S6: if (up) nextstate = S7;
          else    nextstate = S5;
      S7: if (up) nextstate = S0;
          else    nextstate = S6;
    endcase

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

VHDL

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

entity ex4_36 is
  port(clk:   in  STD_LOGIC;
       reset: in  STD_LOGIC;
       up:    in  STD_LOGIC;
       q:     out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture synth of ex4_36 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"  => if up = '1' then
                       nextstate <= "001";
                     else
                       nextstate <= "100";
                     end if;
      when "001"  => if up = '1' then
                       nextstate <= "011";
                     else
                       nextstate <= "000";
                     end if;
      when "011"  => if up = '1' then
                       nextstate <= "010";
                     else
                       nextstate <= "001";
                     end if;
      when "010"  => if up = '1' then
                       nextstate <= "110";
                     else
                       nextstate <= "011";
                     end if;
```

(continued on next page)

(continued from previous page)

VHDL

```
      when "110"  => if up = '1' then
                       nextstate <= "111";
                     else
                       nextstate <= "010";
                     end if;
      when "111"  => if up = '1' then
                       nextstate <= "101";
                     else
                       nextstate <= "110";
                     end if;
      when "101"  => if up = '1' then
                       nextstate <= "100";
                     else
                       nextstate <= "111";
                     end if;
      when "100"  => if up = '1' then
                       nextstate <= "000";
                     else
                       nextstate <= "101";
                     end if;
      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;
```

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;
```

4.38

Verilog

```
module fsm_y(input  clk, reset, a,
             output y);

  reg [1:0] state, nextstate;

  parameter S0  = 2'b00;
  parameter S1  = 2'b01;
  parameter S11 = 2'b11;

  // 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 = S11;
           else   nextstate = S0;
      S11:        nextstate = S11;
      default:    nextstate = S0;
    endcase

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

VHDL

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

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

architecture synth of fsm_y is
  type statetype is (S0, S1, S11);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset = '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 <= S11;
                  else nextstate <= S0;
                  end if;
      when S11 =>      nextstate <= S11;
      when others =>   nextstate <= S0;
    end case;
  end process;

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

(continued on next page)

(continued from previous page)

Verilog

```
module fsm_x(input  clk, reset, a,
             output x);

  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: if   (a) nextstate = S1;
          else     nextstate = S0;
      S1: if   (a) nextstate = S2;
          else     nextstate = S1;
      S2: if   (a) nextstate = S3;
          else     nextstate = S2;
      S3:          nextstate = S3;
    endcase

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

VHDL

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

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

architecture synth of fsm_x is
  type statetype is (S0, S1, S2, S3);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset = '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 <= S2;
                  end if;
      when S1 =>  if a = '1' then
                       nextstate <= S2;
                  else nextstate <= S1;
                  end if;
      when S2 =>  if a = '1' then
                       nextstate <= S3;
                  else nextstate <= S2;
                  end if;
      when S3 =>       nextstate <= S3;
      when others =>   nextstate <= S0;
    end case;
  end process;

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

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;
```

4.40

Verilog

```
module ex4_40(input  clk, reset, x,
              output q);

  reg [1:0] state, nextstate;

  parameter S00 = 2'b00;
  parameter S01 = 2'b01;
  parameter S10 = 2'b10;
  parameter S11 = 2'b11;

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

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

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

VHDL

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

entity ex4_40 is
  port(clk, reset, x: in  STD_LOGIC;
       q:             out STD_LOGIC);
end;

architecture synth of ex4_40 is
  type statetype is (S00, S01, S10, S11);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset = '1' then state <= S00;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

 -- next state logic
  process (state, x) begin
    case state is
      when S00 =>  if x = '1' then
                       nextstate <= S11;
                  else nextstate <= S01;
                  end if;
      when S01 =>  if x = '1' then
                       nextstate <= S10;
                  else nextstate <= S00;
                  end if;
      when S10 =>      nextstate <= S01;
      when S11 =>      nextstate <= S01;
      when others =>   nextstate <= S00;
    end case;
  end process;

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

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;
```

4.42 (a)

Verilog

```
module ex4_42a(input     clk, a, b, c, d,
              output reg q);

  reg areg, breg, creg, dreg;

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

VHDL

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

entity ex4_42a is
  port(clk, a, b, c, d: in  STD_LOGIC;
       q:               out STD_LOGIC);
end;

architecture synth of ex4_42a is
  signal 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;
      q <= ((areg xor breg) xor creg) xor dreg;
    end if;
  end process;
end;
```

(d)

Verilog

```
module ex4_42d(input     clk, a, b, c, d,
              output reg q);

  reg areg, breg, creg, dreg;

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

VHDL

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

entity ex4_42d is
  port(clk, a, b, c, d: in  STD_LOGIC;
       q:               out STD_LOGIC);
end;

architecture synth of ex4_42d is
  signal 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;
      q <= (areg xor breg) xor (creg xor dreg);
    end if;
  end process;
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;
```

4.44

When a signal is declared as `reg` it means that is assigned in an `always` statement, i.e. it is on the left hand side of <= or = in an `always` statement. A signal declared as `reg` is **not** necessarily the output of a register or flip-flop.

4.45

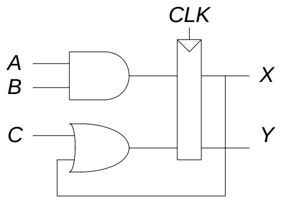
Verilog

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

  reg n1;

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

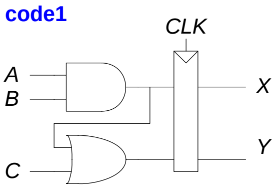
VHDL

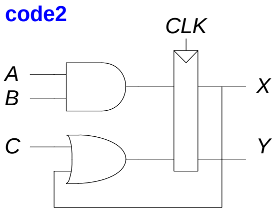
```
library IEEE; use IEEE.STD_LOGIC_1164.all;

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;
```

4.46 They have the same function.



4.47 They do not have the same function.





4.48

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

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

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

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

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

   always @(*)   // this line could also be replaced by “always @(a, b)”
      begin
         y1 = a & b;
         y2 = a | b;
         y3 = a ^ b;
         y4 = ~(a & b);
         y5 = ~(a | b);
      end
endmodule
```

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

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

  always @(s, d0, d1) //“posedge” removed from sensitivity list, d0, d1 added
                       // could also be replaced by always @(*)
      if (s) y = d1;
      else   y = d0;

endmodule
```

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

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

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

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

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

   reg  state, nextstate;

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

   // next state logic
   always @(*)
     case (state)
       1'b0: if (a) state <= 1'b1;
             else state <= 1'b0;
       1'b1: if (~a) state <= 1'b0;
             else state <= 1'b1;
     endcase

   // output logic (combinational)
   always @ (*)
```

```
      if (state == 0) {out1, out2} = {1'b1, 1'b0};
      else            {out1, out2} = {1'b0, 1'b1};

endmodule
```

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

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

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

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

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

   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 @(state)
      case (state)
         S0:      nextstate = S1;
         S1:      nextstate = S2;
         S2:      nextstate = S0;
         default: nextstate = S0;
      endcase

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

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

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

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

endmodule
```

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

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

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

(j) Problem: this is a combinational module, so nonconcurrent assignment statements (=) should be used in the always statement, not concurrent assignment statements (<=).

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

   reg tmp;

   always @ (a, b, c)
   begin
      tmp = a & b;
      y   = tmp & c;
   end
endmodule
```

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'`).

4.50

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

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

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

```
architecture proc of gates is
begin
  process(a, b) begin
    y1 <= a and b;
    y2 <= a or b;
    y3 <= a xor b;
    y4 <= a nand b;
    y5 <= a nor b;
  end process;
end;
```

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

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

(d) **Problem:** The final else statement is missing.

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

(e) **Problem:** The `default` statement is missing in the nextstate case statement.

```
architecture synth of divideby3FSM is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset = '1' then state <= S0;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

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

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

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

```
architecture struct of mux2 is
```

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

(g) **Problem:** The q output cannot be assigned in two process or assignment statements.

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

(h) **Problem:** VHDL does not allow STD_LOGIC type and BOOLEAN type to be used interchangeably.

```
architecture synth of mux3 is
begin
  y <= d2 when s(1) = '1' else
       d1 when s(0) = '1' else d0;
end;
```

Question 4.1

Verilog

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

VHDL

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

Question 4.2

HDLs support *blocking* and *nonblocking assignments* in an `always` / `process` statement. A group of blocking assignments are evaluated in the order they appear in the code, just as one would expect in a standard programming language. A group of nonblocking assignments are evaluated concurrently; all of the statements are evaluated before any of the left hand sides are updated.

Verilog

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

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

VHDL

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

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

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

See Examples 4.24 and 4.29 for comparisons of blocking and non-blocking assignments. Blocking and nonblocking assignment guidelines are given on page 203.

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:

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

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

5.2

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

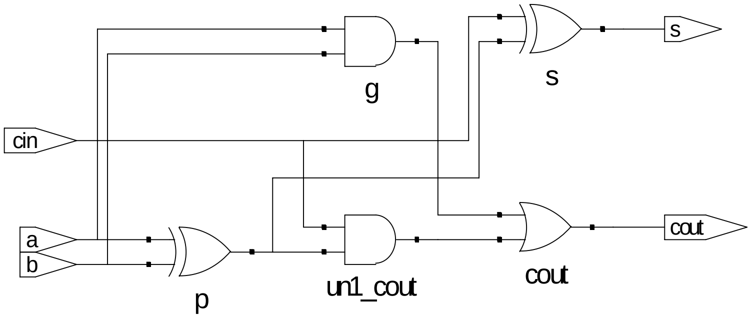


FIGURE 5.1 Full adder implementation

The full adder delay is three two-input gates.

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

The full adder area is five two-input gates.

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

The full adder capacitance is five two-input gates.

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

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

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

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

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

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

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

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

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

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

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

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

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

Now solving for power using Equation 1.4,

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

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

.

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

TABLE 5.1 CLA and ripple-carry adder comparison

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

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.

5.4

Verilog

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

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


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

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

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

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

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

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

endmodule
```

VHDL

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

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

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

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

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

  signal p, g: STD_LOGIC_VECTOR(14 downto 0);
  signal pij_0, gij_0, pij_1, gij_1,
         pij_2, gij_2, pij_3, gij_3:
               STD_LOGIC_VECTOR(7 downto 0);
  signal gen:  STD_LOGIC_VECTOR(15 downto 0);
  signal pik_0, pik_1, pik_2, pik_3,
         gik_0, gik_1, gik_2, gik_3,
         pkj_0, pkj_1, pkj_2, pkj_3,
         gkj_0, gkj_1, gkj_2, gkj_3, dummy:
               STD_LOGIC_VECTOR(7 downto 0);
begin
  pgblock_top: pgblock
    port map(a(14 downto 0), b(14 downto 0), p, g);

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

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

(*continued from previouspage*)

Verilog

VHDL

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

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

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

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

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

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

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

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

(*continued from previous page*)

Verilog

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

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

endmodule

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

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

endmodule

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

  assign s = a ^ b ^ g;

endmodule
```

VHDL

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

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

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


library IEEE; use IEEE.STD_LOGIC_1164.all;

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

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


library IEEE; use IEEE.STD_LOGIC_1164.all;

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

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

5.5

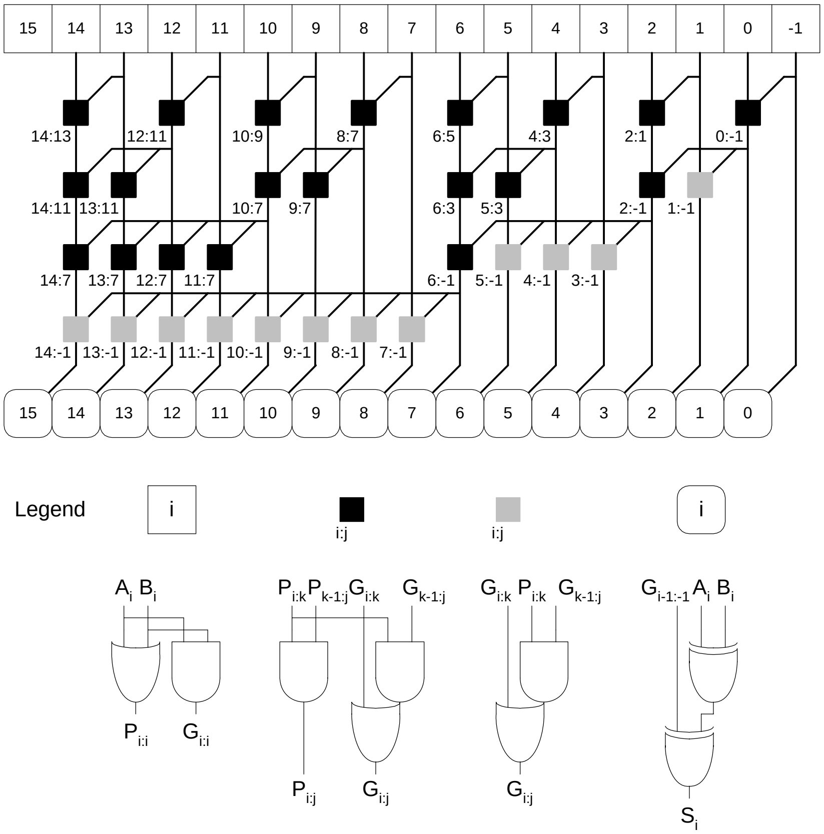


FIGURE 5.2 16-bit prefix adder with "gray cells"

5.6

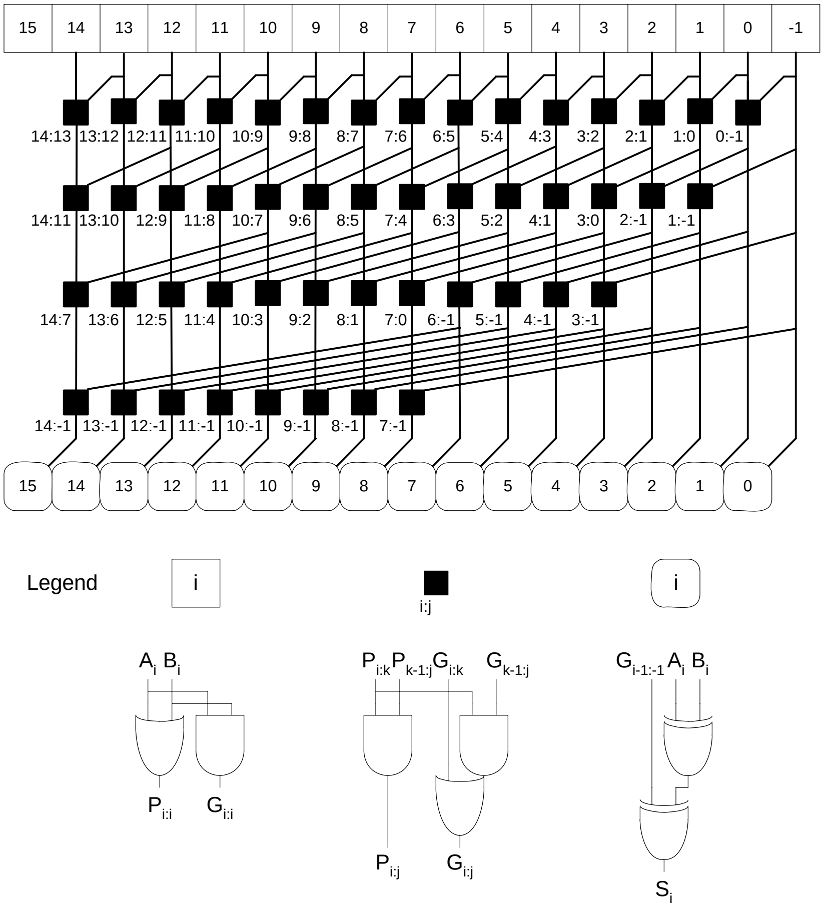


FIGURE 5.3 Schematic of a 16-bit Kogge-Stone adder

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

$$t_{pd\_\text{priority}} = (\log_2 N+1)t_{pd\_\text{AND2}} + \text{t}_{pd\_\text{OR}N/2}$$

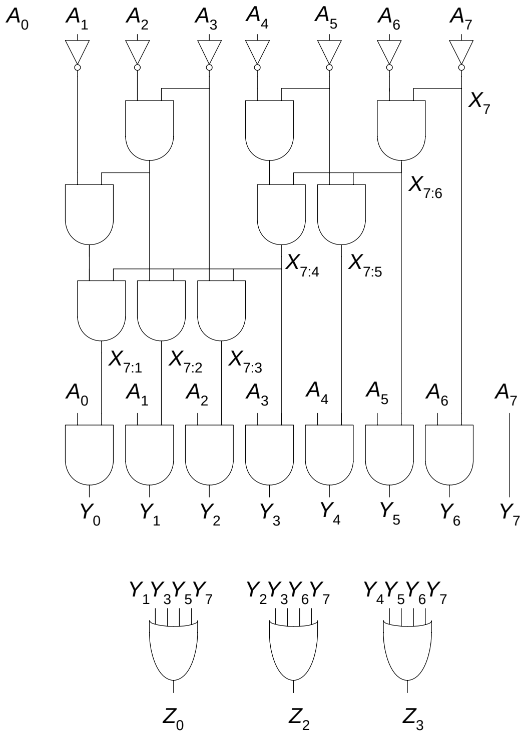


FIGURE 5.4 8-input priority encoder

Verilog

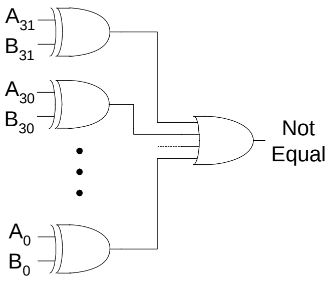
```
module priorityckt(input  [7:0] a,
                output [2:0] z);


  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

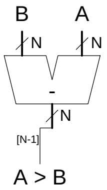
```
library IEEE; use IEEE.STD_LOGIC_1164.all;

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.

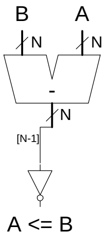
5.8 (a)



(b)



(c)



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 <=
      ("0000000000000000000000000000000" & S(31));
      when others => Y <= X"00000000";
    end case;
  end process;
end;
```

5.10
(a)
When adding:

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

When subtracting:

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

In equation form:
Overflow = ADD & (A & B & ~S[31] | ~A & ~B & S[31]) |
                    SUB & (A & ~B & ~S[31] | ~A & B & S[31]);
// note: S is the output of the adder

When the ALU is performing addition, F[2:1] = 01. With subtraction, F[2:1] = 11. Thus,

Overflow = ~F[2] & F[1] & (A & B & ~S[31] | ~A & ~B & S[31]) |
F[2] & F[1] & (~A & B & S[31] | A & ~B & ~S[31]);

5.10 (b)

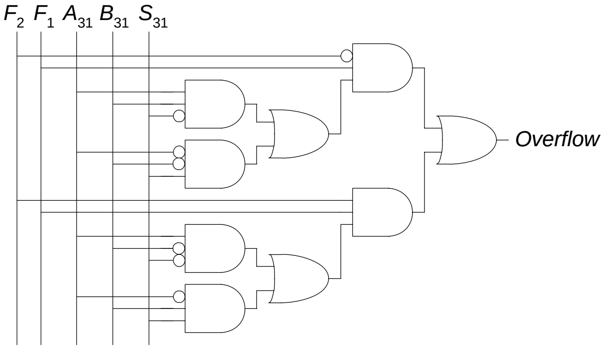


FIGURE 5.5 Overflow circuit

5.10 (c)

Verilog

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

  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:        out STD_LOGIC_VECTOR(31 downto 0);
   Overflow: out STD_LOGIC);
end;

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

  -- alu function
  process (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 <=
 ("0000000000000000000000000000000" & S(31));
      when others => Y <= X"00000000";
    end case;
  end process;

  -- 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;
```

5.11

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 <=
        ("0000000000000000000000000000000" & 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;
```

5.12

The following shows the contents of the file test_alu32.tv and test_alu32_vhdl.tv. Note that VHDL does not use underscores (“_”) to separate the hex digits.

test_alu32.tv

```
0_A_00000000_00000000_00000000
0_2_00000000_FFFFFFFF_FFFFFFFF
0_A_00000001_FFFFFFFF_00000000
0_2_000000FF_00000001_00000100
0_E_00000000_00000000_00000000
0_6_00000000_FFFFFFFF_00000001
0_E_00000001_00000001_00000000
0_6_00000100_00000001_000000FF
0_F_00000000_00000000_00000000
0_7_00000000_00000001_00000001
0_F_00000000_FFFFFFFF_00000000
0_F_00000001_00000000_00000000
0_7_FFFFFFFF_00000000_00000001
0_0_FFFFFFFF_FFFFFFFF_FFFFFFFF
0_0_FFFFFFFF_12345678_12345678
0_0_12345678_87654321_02244220
0_8_00000000_FFFFFFFF_00000000
0_1_FFFFFFFF_FFFFFFFF_FFFFFFFF
0_1_12345678_87654321_97755779
0_1_00000000_FFFFFFFF_FFFFFFFF
0_9_00000000_00000000_00000000
1_2_FFFFFFFF_80000000_7FFFFFFF
1_2_00000001_7FFFFFFF_80000000
1_6_80000000_00000001_7FFFFFFF
1_6_7FFFFFFF_FFFFFFFF_80000000
```

test_alu32_vhdl.tv

```
0a000000000000000000000000
0200000000ffffffffffffffff
0a00000001ffffffff00000000
02000000ff0000000100000100
0e000000000000000000000000
0600000000ffffffff00000001
0e000000010000000100000000
060000010000000001000000ff
0f000000000000000000000000
07000000000000000100000001
0f00000000ffffffff00000000
0f000000010000000000000000
07ffffffff0000000000000001
00ffffffffffffffffffffffff
00ffffffff1234567812345678
00123456788765432102244220
0800000000ffffffff00000000
01ffffffffffffffffffffffff
01123456788765432197755779
0100000000ffffffffffffffff
09000000000000000000000000
12ffffffff800000007fffffff
12000000017fffffff80000000
1680000000000000017fffffff
167fffffffffffffff80000000
```

(continued on next page)

Testbench

Verilog

```
module test_alu32_v;

  // Inputs
  reg [31:0] A;
  reg [31:0] B;
  reg [2:0] F;

  // Outputs
  wire [31:0] Y;
  wire Zero, Overflow;

  // Internal signals
  reg clk;

  // Simulation variables
  reg  [31:0]   vectornum, errors;
  reg  [100:0]  testvectors[10000:0];
  reg  [31:0]   ExpectedY;
  reg           ExpectedZero;
  reg           ExpectedOverflow;

  // Instantiate the Unit Under Test (UUT)
  alu32 uut (A, B, F, Y, Zero, Overflow);

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

VHDL

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

entity test_alu32_vhd is -- no inputs or outputs
end;

architecture sim of test_alu32_vhd is
  component alu32
  port(a, b:     in    STD_LOGIC_VECTOR(31 downto 0);
       f:         in    STD_LOGIC_VECTOR(2 downto 0);
       y:        inout STD_LOGIC_VECTOR(31 downto 0);
       zero:      out   STD_LOGIC;
       overflow: out   STD_LOGIC);
  end component;

  signal a, b:       STD_LOGIC_VECTOR(31 downto 0);
  signal f:          STD_LOGIC_VECTOR(2 downto 0);
  signal y:          STD_LOGIC_VECTOR(31 downto 0);
  signal zero:       STD_LOGIC;
  signal overflow:   STD_LOGIC;
  signal clk, reset: STD_LOGIC;
  signal yexpected:  STD_LOGIC_VECTOR(31 downto 0);
  signal oexpected:  STD_LOGIC;
  signal zexpected:  STD_LOGIC;
  constant MEMSIZE: integer := 25;
  type tvarray is array(MEMSIZE downto 0) of
  STD_LOGIC_VECTOR(100 downto 0);
  shared variable testvectors: tvarray;
  shared variable vectornum, errors: integer;
begin

  -- instantiate device under test
  dut: alu32 port map(a, b, f, y, zero, overflow);

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

(continued on next page)

(continued from previous page)

Verilog

```
  // at start of test, load vectors
  initial
    begin
      $readmemh("test_alu32.tv", testvectors);
      vectornum = 0; errors = 0;
    end
```

VHDL

```
  -- at start of test, load vectors
  -- and pulse reset
  process is
    file tv: TEXT;
    variable i, index, count: integer;
    variable L: line;
    variable ch: character;
    variable result: integer;
  begin
    -- read file of test vectors
    i := 0;
    index := 0;
    FILE_OPEN(tv, "test_alu32_vhdl.tv", READ_MODE);
    report "Opening file\n";
    while not endfile(tv) loop
      readline(tv, L);
      result := 0;
      count := 3;
      for i in 1 to 26 loop
        read(L, ch);
        if '0' <= ch and ch <= '9' then
            result := result*16 + character'pos(ch)
                      - character'pos('0');
        elsif 'a' <= ch and ch <= 'f' then
           result := result*16 + character'pos(ch)
                     - character'pos('a')+10;
        else report "Format error on line " &
             integer'image(index) & " i = " &
             integer'image(i) & " char = " &
             character'image(ch)
             severity error;
        end if;

        -- load vectors
        -- assign first 5 bits
        if (i = 2) then
          testvectors(index)( 100 downto 96) :=
             CONV_STD_LOGIC_VECTOR(result, 5);
          count := count - 1;
          result := 0;         -- reset result
        -- assign the rest of testvectors in
        -- 32-bit increments
       elsif ((i = 10) or (i = 18) or (i = 26)) then
          testvectors(index)( (count*32 + 31)
                              downto (count*32)) :=
             CONV_STD_LOGIC_VECTOR(result, 32);
          count := count - 1;
          result := 0;         -- reset result
        end if;
      end loop;

      index := index + 1;
    end loop;

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

  end process;
```

(continued on next page)

(continued from previous page)

Verilog

```
 // apply test vectors on rising edge of clk

  always @ (posedge clk)
  begin
    #1; {ExpectedOverflow, ExpectedZero, F, A, B,
         ExpectedY} = testvectors[vectornum];
    end

  // check results on falling edge of clk
  always @(negedge clk)
  begin
     if ({Y, Zero, Overflow} !==
        {ExpectedY, ExpectedZero, ExpectedOverflow})
       begin
         $display("Error: inputs: F = %h, A = %h,
         B = %h", F, A, B);
         $display("  Y = %h, Zero = %b
           Overflow = %b\n (Expected Y = %h,
           Expected Zero = %b, Expected Overflow
           = %b)", Y, Zero, Overflow,
           ExpectedY, ExpectedZero,
           ExpectedOverflow);
         errors = errors + 1;
       end
     vectornum = vectornum + 1;
     if (testvectors[vectornum] === 101'hx)
       begin
         $display("%d tests completed with %d
           errors", vectornum, errors);
         $finish;
       end
  end

endmodule
```

VHDL

```
  -- apply test vectors on rising edge of clk
  process (clk) begin
    if (clk'event and clk = '1') then
      oexpected <= testvectors(vectornum)(100)
        after 1 ns;
     zexpected <= testvectors(vectornum)(99)
        after 1 ns;
     f      <= testvectors(vectornum)(98 downto 96)
        after 1 ns;
     a       <= testvectors(vectornum)(95 downto 64)
        after 1 ns;
     b       <= testvectors(vectornum)(63 downto 32)
        after 1 ns;
    yexpected <= testvectors(vectornum)(31 downto 0)
        after 1 ns;
    end if;
  end process;

  -- check results on falling edge of clk
  process (clk) begin
    if (clk'event and clk = '0' and reset = '0') then
      assert y = yexpected
        report "Error: vectornum = " &
          integer'image(vectornum) &
          "y = " & integer'image(CONV_INTEGER(y)) &
         ", a = " & integer'image(CONV_INTEGER(a)) &
         ", b = " & integer'image(CONV_INTEGER(b)) &
          ", f = " & integer'image(CONV_INTEGER(f));
      assert overflow = oexpected
        report "Error: overflow = " &
          STD_LOGIC'image(overflow);
      assert zero = zexpected
        report "Error: zero = " &
          STD_LOGIC'image(zero);
      if ( (y /= yexpected) or
           (overflow /= oexpected) or
           (zero /= zexpected) ) then
        errors := errors + 1;
      end if;
      vectornum := vectornum + 1;
      if (is_x(testvectors(vectornum))) then
        if (errors = 0) then
          report "Just kidding -- " &
                 integer'image(vectornum) &
                 " tests completed successfully."
                 severity failure;
        else
          report integer'image(vectornum) &
                 " tests completed, errors = " &
                 integer'image(errors)
                 severity failure;
        end if;
      end if;
    end if;
  end process;
end;
```

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

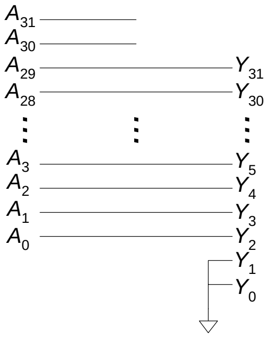


FIGURE 5.6 2-bit left shifter, 32-bit input and output

2-bit Left Shifter

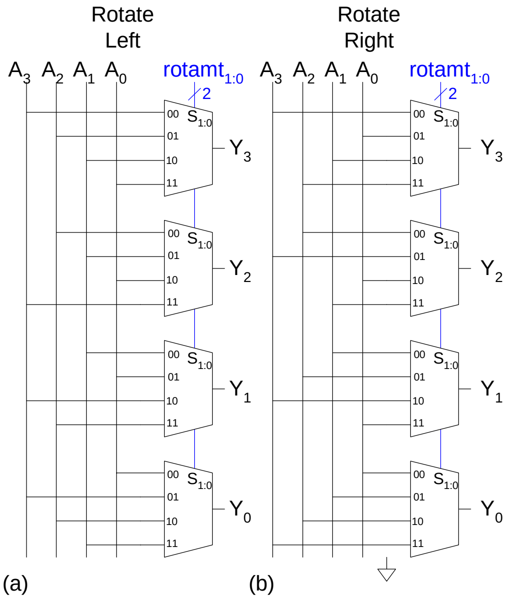
Verilog

```
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;
```

5.14


Rotate Left
Rotate Right
A3 A2 A1 A0
rotamt1:0
2
S1:0
00
01
10
11
Y3
Y2
Y1
Y0
(a)
(b)

4-bit Left and Right Rotator

Verilog

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

 // right rotated
 always @ ( * )
   case(shamt)
     2'b00: right_rotated = {a[3], a[2], a[1], a[0]};
     2'b01: right_rotated = {a[0], a[3], a[2], a[1]};
     2'b10: right_rotated = {a[1], a[0], a[3], a[2]};
     2'b11: right_rotated = {a[2], a[1], a[0], a[3]};
    endcase

 // left rotated
 always @ ( * )
   case(shamt)
     2'b00: left_rotated = {a[3], a[2], a[1], a[0]};
     2'b01: left_rotated = {a[2], a[1], a[0], a[3]};
     2'b10: left_rotated = {a[1], a[0], a[3], a[2]};
     2'b11: left_rotated = {a[0], a[3], a[2], a[1]};
   endcase

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

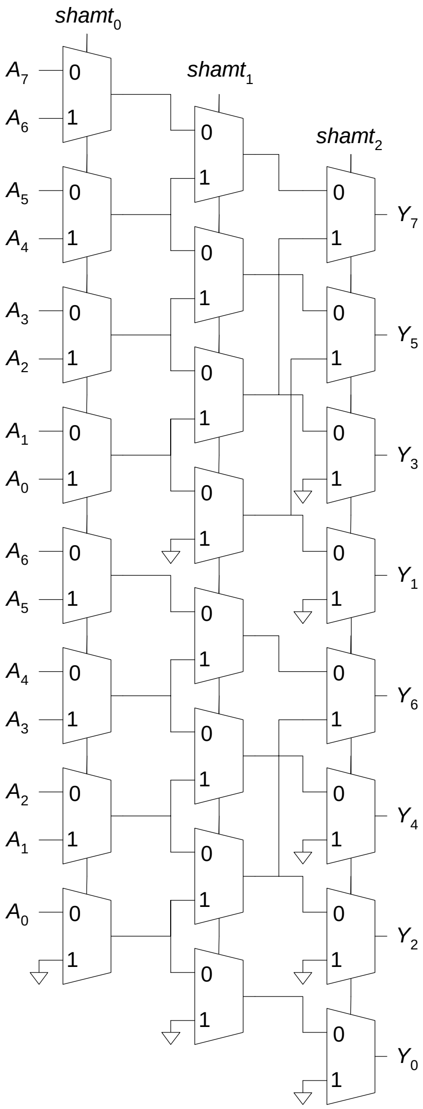


FIGURE 5.7 8-bit left shifter using 24 2:1 multiplexers

5.16

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

5.17

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

(b) $B = A_{N\text{-}1}$ (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*

5.18

t_{pd}_MULT4 = $t_{\text{AND}} + 3t_{FA}$

An $N \times N$ multiplier has N-bit operands, N partial products, and N-1 stages of 1-bit adders. So the propagation is:

t_{pd}_MULTN = $t_{\text{AND}} + (N\text{-}1)t_{FA}$

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 $-2^{\text{N-1}}$. Thus, the product of two N-bit complement numbers, y and x is:

P = -y$_{N\text{-}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 5.8 shows a 4 x 4 multiplier. Figure 5.8 (b) shows the partial products using the above equation. Figure 5.8

(c) shows a simplified version, pushing through the 1's. This is known as a *modified Baugh-Wooley multiplier*. It can be built using a hierarchy of adders.

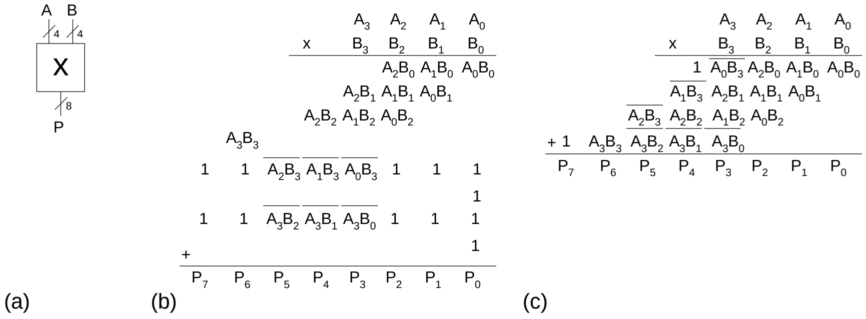


FIGURE 5.8 Multiplier: (a) symbol, (b) function, (c) simplified function

5.20

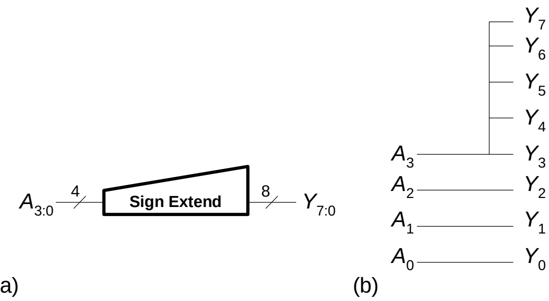


FIGURE 5.9 Sign extension unit (a) symbol, (b) underlying hardware

Verilog

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

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

endmodule
```

VHDL

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

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

architecture synth of signext4_8 is
begin
  y <= a(3) & a(3) & a(3) & a(3) & a(3 downto 0);
end;
```

5.21

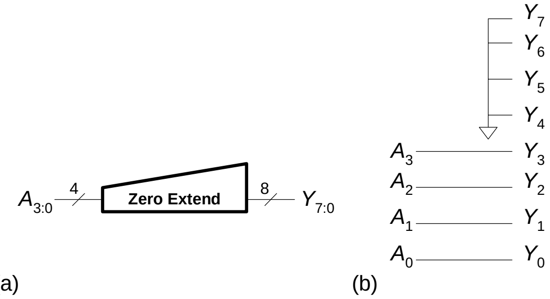


FIGURE 5.10 Zero extension unit (a) symbol, (b) underlying hardware

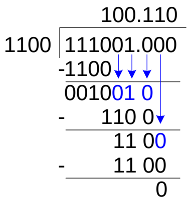
Verilog

```
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;
```

5.22



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

5.24
(a) 1000 1101 . 1001 0000 = 0x8D90
(b) 0010 1010 . 0101 0000 = 0x2A50
(c) 1001 0001 . 0010 1000 = 0x9128

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

5.26
(a) -1101.1001 = -1.1011001 × 2^3
Thus, the biased exponent = 127 + 3 = 130 = $1000\ 0010_2$
In IEEE 754 single-precision floating-point format:
1 1000 0010 101 1001 0000 0000 0000 0000 = **0xC1590000**

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

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

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

5.28

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

5.29

(a)

0xC0D20004 = 1 1000 0001 101 0010 0000 0000 0000 0100
= - 1.101 0010 0000 0000 0000 01 × 2^2
0x72407020 = 0 1110 0100 100 0000 0111 0000 0010 0000
= 1.100 0000 0111 0000 001 × 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 × 2^2
0x40DC0004 = 0 1000 0001 101 1100 0000 0000 0000 0100
= 1.101 1100 0000 0000 0000 01 × 2^2

1.101 1100 0000 0000 0000 01 × 2^2
- 1.101 0010 0000 0000 0000 01 × 2^2
= 0.000 1010 × 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.

5.30

We only need to change step 5.

1. Extract exponent and fraction bits.
2. Prepend leading 1 to form the mantissa.
3. Compare exponents.
4. Shift smaller mantissa if necessary.
5. If one number is negative: Subtract it from the other number. If the result is negative, take the absolute value of the result and make the sign bit 1.

   If both numbers are negative: Add the numbers and make the sign bit 1.
   If both numbers are positive: Add the numbers and make the sign bit 0.
6. Normalize mantissa and adjust exponent if necessary.
7. Round result
8. Assemble exponent and fraction back into floating-point number

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.

5.32

(a) 245 = 11110101 = 1.1110101×2^7
= 0 1000 0110 111 0101 0000 0000 0000 0000
= 0x43750000

0.0625 = 0.0001 = 1.0×2^{-4}
= 0 0111 1011 000 0000 0000 0000 0000 0000
= 0x3D800000

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

(c)

1.1110101×2^7 = 0 0000 0111 111 0101 0000 0000 0000 0000
= 0x03F50000

1.0×2^{-4} = 0 1111 1100 000 0000 0000 0000 0000 0000
= 0x7E000000

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

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

5.33

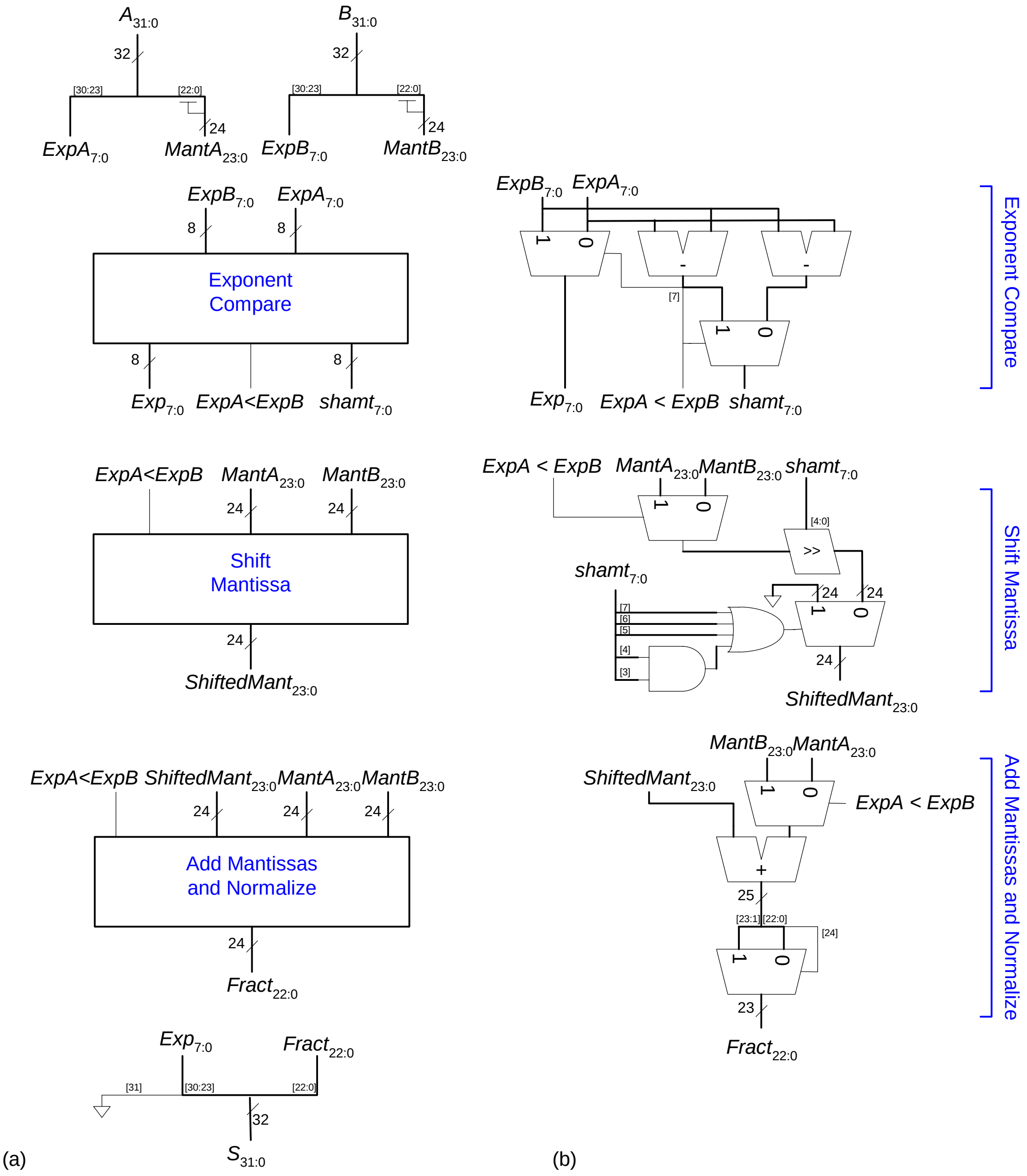


FIGURE 5.11 Floating-point adder hardware: (a) block diagram, (b) underlying hardware

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;
```

5.34
(a)

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

(b)

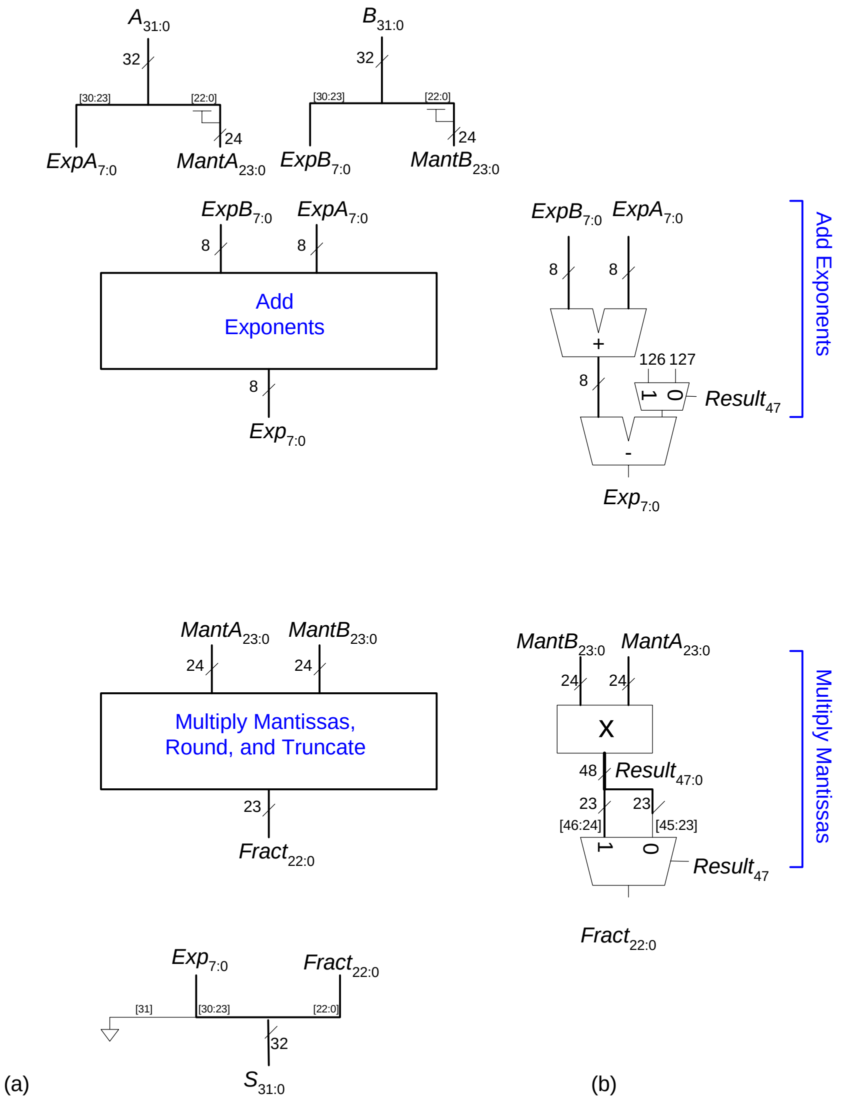


FIGURE 5.12 Floating-point multiplier block diagram

(c)

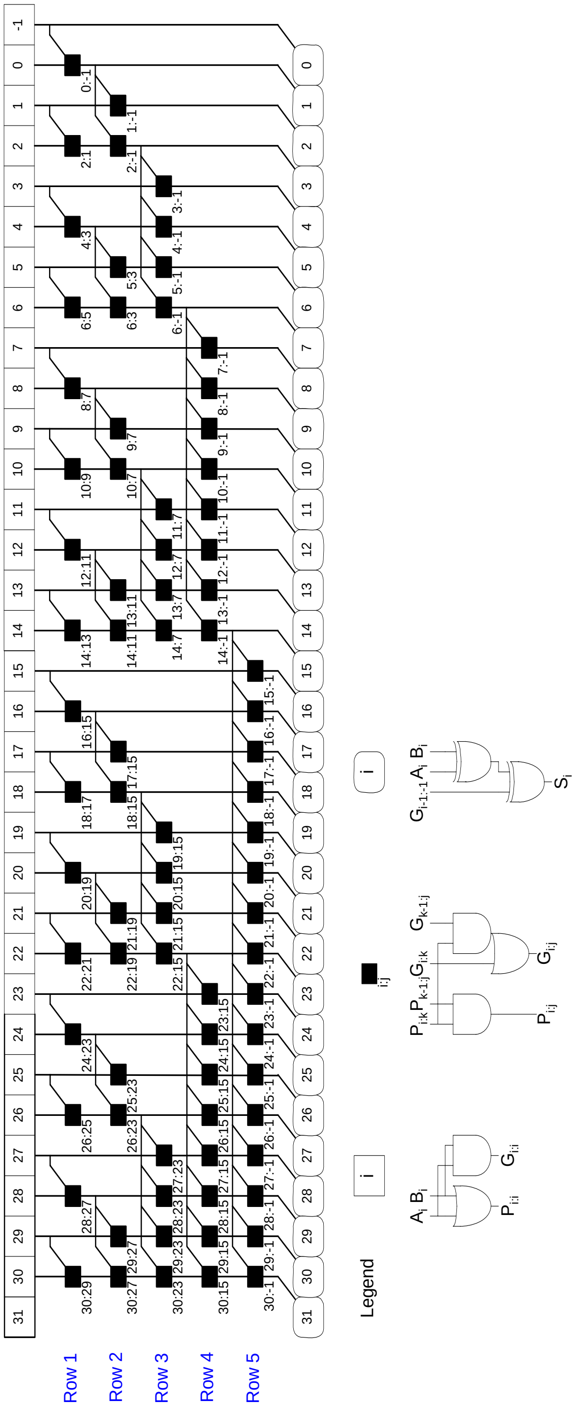
Verilog

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

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

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

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

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

endmodule
```

VHDL

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

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

architecture synth of fpmult is
  signal expa, expb, exp:
    STD_LOGIC_VECTOR(7 downto 0);
  signal manta, mantb:
    STD_LOGIC_VECTOR(23 downto 0);
  signal fract:
    STD_LOGIC_VECTOR(22 downto 0);
  signal result:
    STD_LOGIC_VECTOR(47 downto 0);
begin


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

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

end;
```

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


-1
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
Row 1
0:-1
2:1
4:3
6:5
8:7
10:9
12:11
14:13
16:15
18:17
20:19
22:21
24:23
26:25
28:27
30:29
Row 2
1:-1
2:-1
5:3
6:3
9:7
10:7
13:11
14:11
17:15
18:15
21:19
22:19
25:23
26:23
29:27
30:27
Row 3
3:-1
4:-1
5:-1
6:-1
11:7
12:7
13:7
14:7
19:15
20:15
21:15
22:15
27:23
28:23
29:23
30:23
Row 4
7:-1
8:-1
9:-1
10:-1
11:-1
12:-1
13:-1
14:-1
23:15
24:15
25:15
26:15
27:15
28:15
29:15
30:15
Row 5
15:-1
16:-1
17:-1
18:-1
19:-1
20:-1
21:-1
22:-1
23:-1
24:-1
25:-1
26:-1
27:-1
28:-1
29:-1
30:-1
Legend
i
A_i B_i
$P_{i:i}$
$G_{i:i}$
i:j
$P_{i:k}$ $P_{k-1:j}$ $G_{i:k}$ $G_{k-1:j}$
$G_{i:j}$
$P_{i:j}$
i
$G_{i-1:-1}$ 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;
```

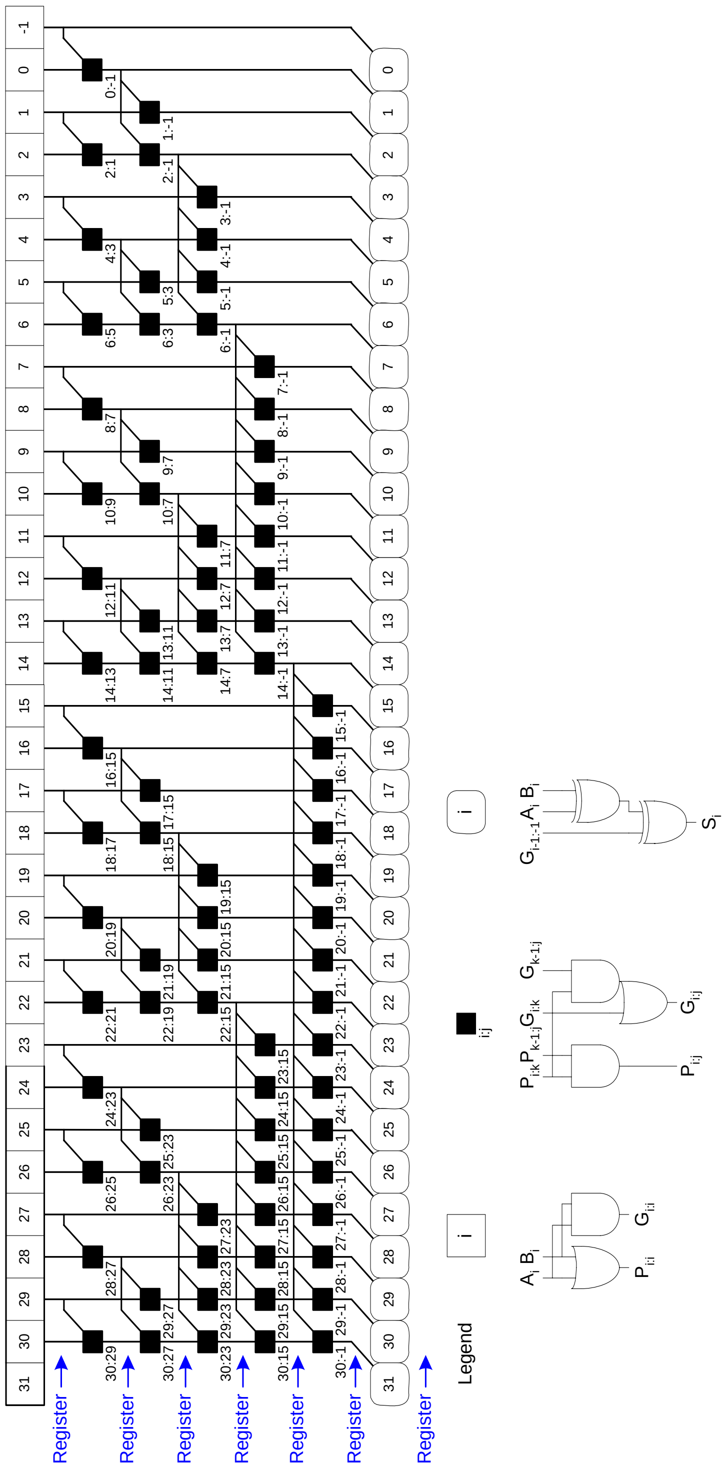
5.35 (c) Using Equation 5.11 to find the delay of the prefix adder:

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

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


-1
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
0:-1
1:-1
2:1
2:-1
3:-1
4:3
4:-1
5:3
5:-1
6:5
6:3
6:-1
7:-1
8:7
8:-1
9:7
9:-1
10:9
10:7
10:-1
11:7
11:-1
12:11
12:7
12:-1
13:11
13:7
13:-1
14:13
14:11
14:7
14:-1
15:-1
16:15
16:-1
17:15
17:-1
18:17
18:15
18:-1
19:15
19:-1
20:19
20:15
20:-1
21:19
21:15
21:-1
22:21
22:19
22:15
22:-1
23:15
23:-1
24:23
24:15
24:-1
25:23
25:15
25:-1
26:25
26:23
26:15
26:-1
27:23
27:15
27:-1
28:27
28:23
28:15
28:-1
29:27
29:23
29:15
29:-1
30:29
30:27
30:23
30:15
30:-1
Register
Register
Register
Register
Register
Register
Register
Legend
i
i:j
i
A_i B_i
P_i:i
G_i:i
P_i:k P_k-1:j G_i:k
G_k-1:j
G_i:j
P_i:j
G_i-1:-1 A_i B_i
S_i

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;
```

5.36

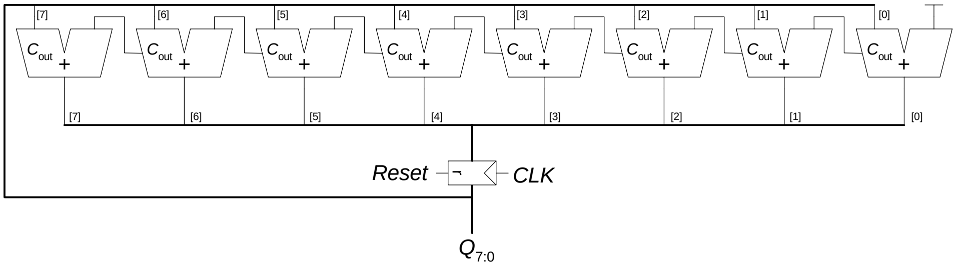


FIGURE 5.13 Incrementer built using half adders

5.37

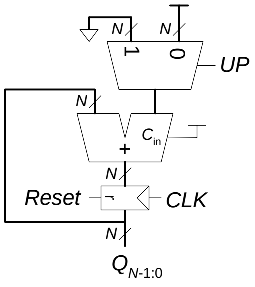


FIGURE 5.14 Up/Down counter

5.38

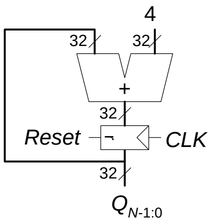


FIGURE 5.15 32-bit counter that increments by 4 on each clock edge

5.39

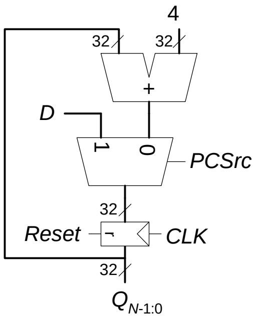


FIGURE 5.16 32-bit counter that increments by 4 or loads a new value, *D*

5.40
(a)
0000
1000
1100
1110
1111
0111
0011
0001
(repeat)

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

5.40 (c)

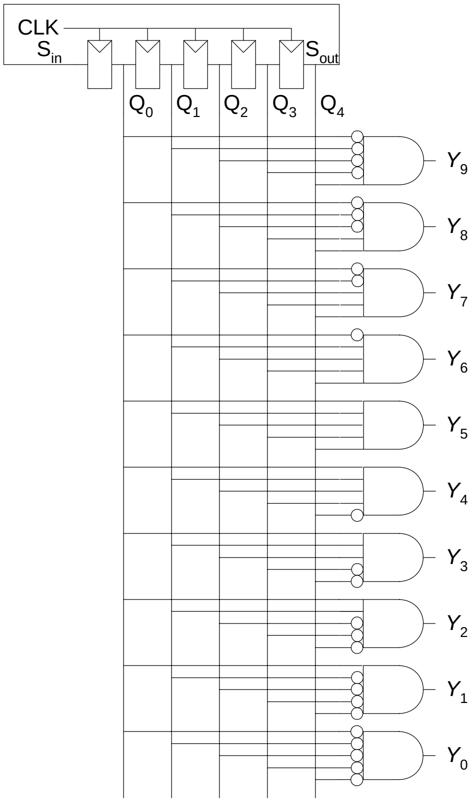


FIGURE 5.17 10-bit decimal counter using a 5-bit Johnson counter

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

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;
```

5.42
(a)

value $a_{1:0}$	encoding $y_{4:0}$
00	00001
01	01010
10	10100
11	11111

TABLE 5.2 Possible encodings

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

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

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

TABLE 5.3 Memory array values for Exercise 5.42

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

TABLE 5.3 Memory array values for Exercise 5.42

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

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 5.18).

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

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

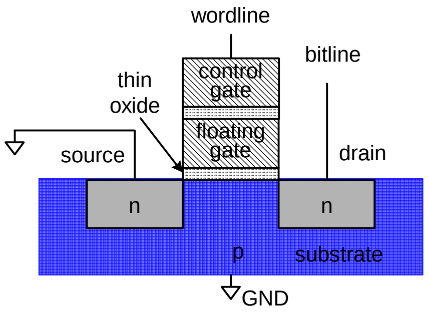


FIGURE 5.18 Flash EEPROM

5.44

(a)

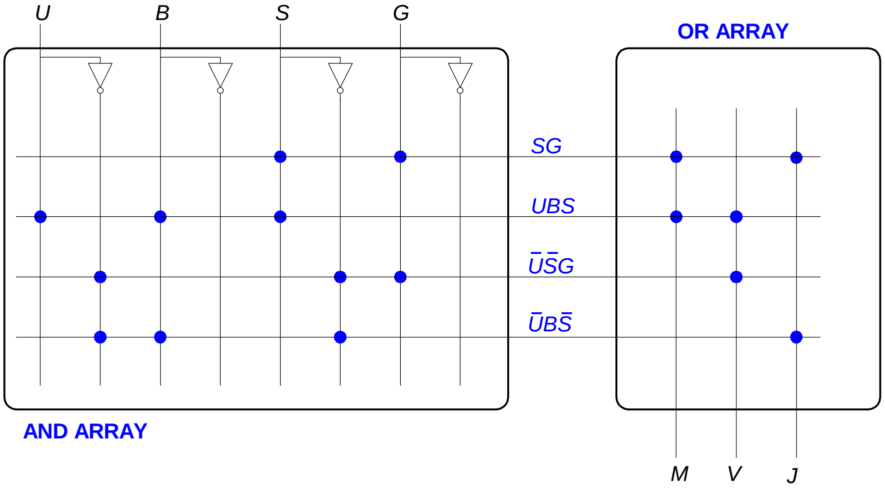


FIGURE 5.19 4 x 4 x 3 PLA implementing Exercise 5.44

5.44 (b)

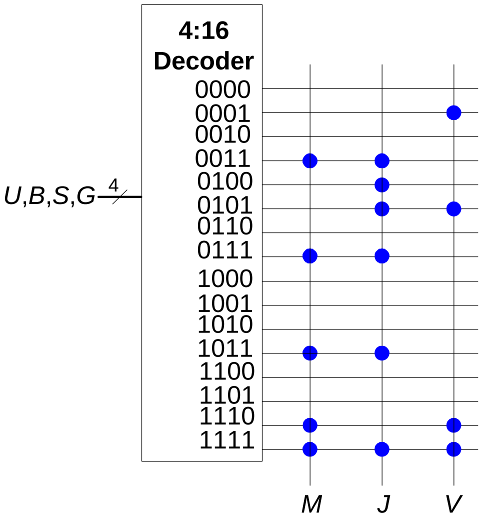


FIGURE 5.20 16 x 3 ROM implementation of Exercise 5.44

5.44 (c)

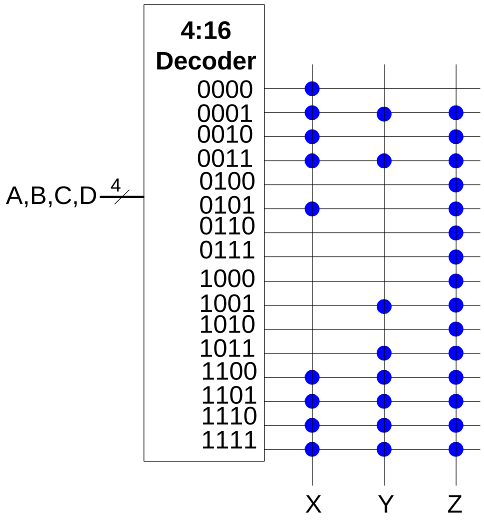
Verilog

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

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

VHDL

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

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

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

5.45



5.46

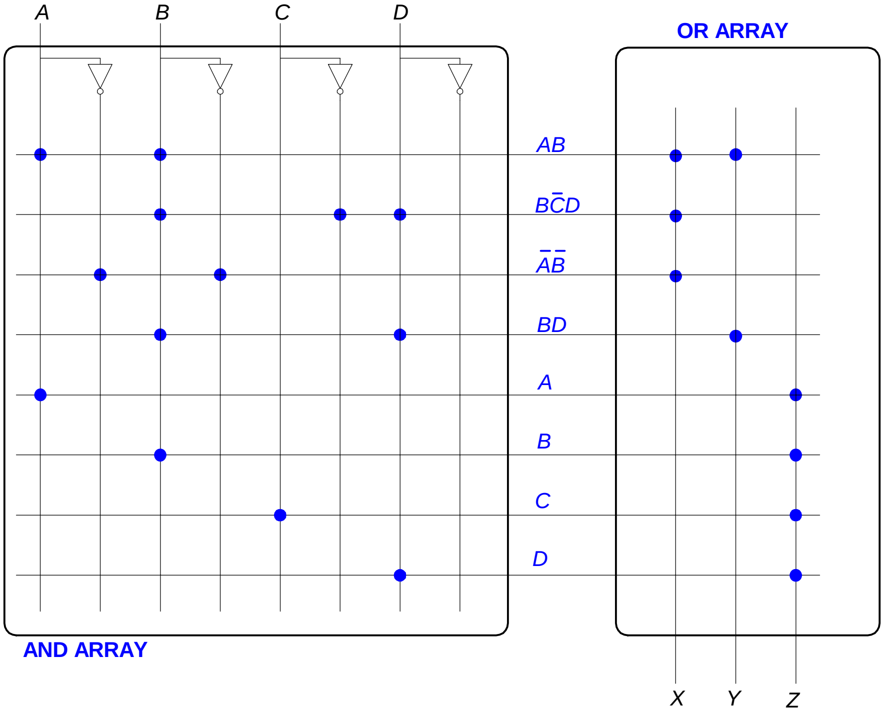


FIGURE 5.21 4 x 8 x 3 PLA for Exercise 5.46

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.

5.48

(a) Yes. Both circuits can compute any function of *K* inputs and *K* outputs.

(b) No. The second circuit can only represent 2^K states. The first can represent more.

(c) Yes. Both circuits compute any function of 1 input, *N* outputs, and 2^K states.

(d) No. The second circuit forces the output to be the same as the state encoding, while the first one allows outputs to be independent of the state encoding.

5.49

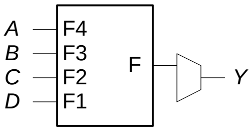
(a) $Y = F(A,B,C,D,E,F,G,H,I) = ABCDEFGHI$

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

5.50

(a) 1 CLB

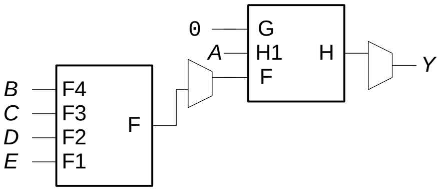
(*A*)	(*B*)	(*C*)	(*D*)	(*Y*)
F4	F3	F2	F1	F
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1



(b) 1 CLB

(*B*)	(*C*)	(*D*)	(*E*)	
F4	F3	F2	F1	F
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

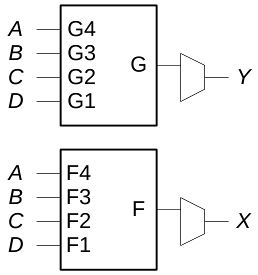
(*A*)			(*Y*)
H1	F	G	H
0	0	X	0
0	1	X	1
1	0	X	1
1	1	X	1



5.50 (c) 1 CLB

(*A*)	(*B*)	(*C*)	(*D*)	(Y)
F4	F3	F2	F1	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

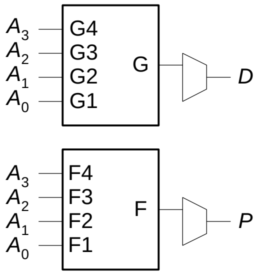
(A)	(*B*)	(*C*)	(*D*)	(*Z*)
G4	G3	G2	G1	G
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1



(d) 1 CLB

(A_3)	(A_2)	(A_1)	(A_0)	(*D*)
F4	F3	F2	F1	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

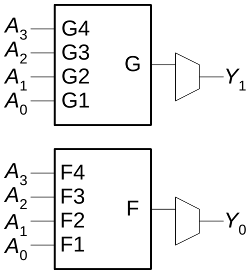
(A_3)	(A_2)	(A_1)	(A_0)	(*P*)
G4	G3	G2	G1	G
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0



5.50 (e) 1 CLB

(A_3)	(A_2)	(A_1)	(A_0)	(Y_0)
F4	F3	F2	F1	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

(A_3)	(A_2)	(A_1)	(A_0)	(Y_1)
G4	G3	G2	G1	G
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

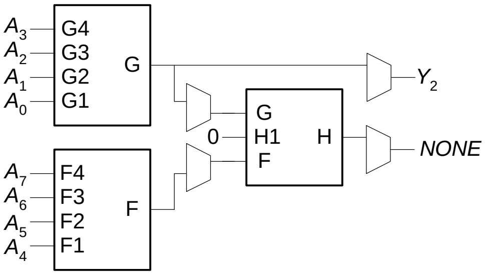


5.50 (f) 3 CLBs

(A_7)	(A_6)	(A_5)	(A_4)	
F4	F3	F2	F1	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

(A_3)	(A_2)	(A_1)	(A_0)	
G4	G3	G2	G1	G
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

			(NONE)
H1	F	G	G
X	0	0	1
X	0	1	0
X	1	0	0
X	1	1	0



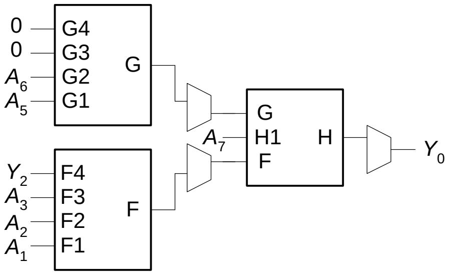
(continued on next page)

5.50 (f) *(continued from previous page)*

(Y_2)	(A_3)	(A_2)	(A_1)	
F4	F3	F2	F1	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

		(A_6)	(A_5)	
G4	G3	G2	G1	G
X	X	0	0	0
X	X	0	1	1
X	X	1	0	0
X	X	1	1	0

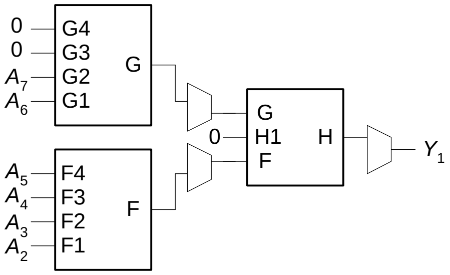
(A_7)			(Y_0)
H1	F	G	G
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	0	1



(A_5)	(A_4)	(A_3)	(A_2)	
F4	F3	F2	F1	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0

		(A_7)	(A_6)	
G4	G3	G2	G1	G
X	X	0	0	0
X	X	0	1	1
X	X	1	0	1
X	X	1	1	1

			(Y_0)
H1	F	G	G
X	0	0	0
X	0	1	1
X	1	0	1
X	1	1	1



5.50 (g) 4 CLBs

	(A_2)	(A_1)	(A_0)	(Y_0)
F4	F3	F2	F1	F
X	0	0	0	1
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

	(A_2)	(A_1)	(A_0)	(Y_1)
G4	G3	G2	G1	G
X	0	0	0	0
X	0	0	1	1
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

	(A_2)	(A_1)	(A_0)	(Y_2)
F4	F3	F2	F1	F
X	0	0	0	0
X	0	0	1	0
X	0	1	0	1
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

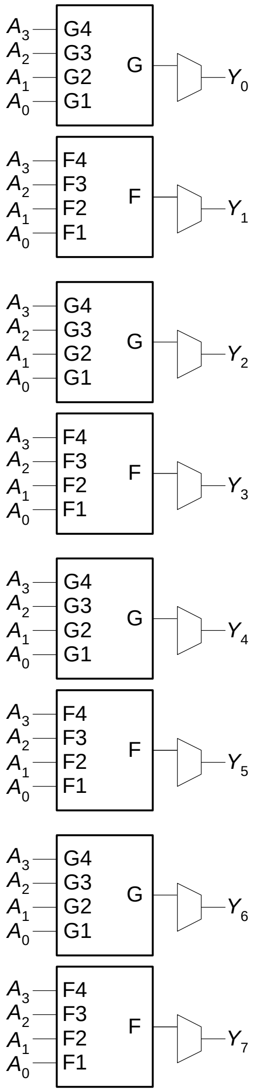
	(A_2)	(A_1)	(A_0)	(Y_3)
G4	G3	G2	G1	G
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	1
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

	(A_2)	(A_1)	(A_0)	(Y_4)
F4	F3	F2	F1	F
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	1
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

	(A_2)	(A_1)	(A_0)	(Y_5)
G4	G3	G2	G1	G
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	1
X	1	1	0	0
X	1	1	1	0

	(A_2)	(A_1)	(A_0)	(Y_6)
F4	F3	F2	F1	F
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	1
X	1	1	1	0

	(A_2)	(A_1)	(A_0)	(Y_7)
G4	G3	G2	G1	G
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	1

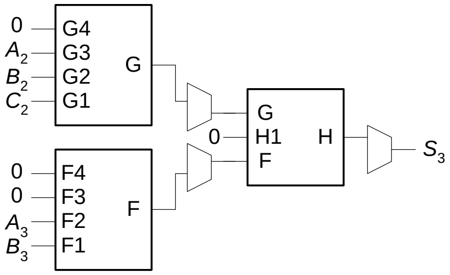


5.50 (h) 3 CLBs

		(A_3)	(B_3)	
F4	F3	F2	F1	F
X	X	0	0	0
X	X	0	1	1
X	X	1	0	1
X	X	1	1	0

	(A_2)	(B_2)	(C_2)	
G4	G3	G2	G1	G
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	1
X	1	0	0	0
X	1	0	1	1
X	1	1	0	1
X	1	1	1	1

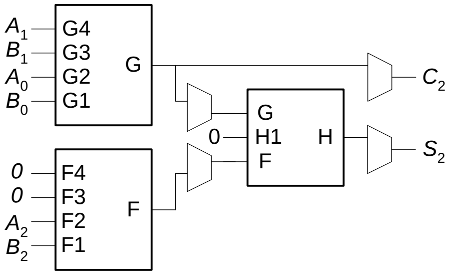
			(S_3)
H1	F	G	H
X	0	0	0
X	0	1	1
X	1	0	1
X	1	1	0



		(A_2)	(B_2)	
F4	F3	F2	F1	F
X	X	0	0	0
X	X	0	1	1
X	X	1	0	1
X	X	1	1	0

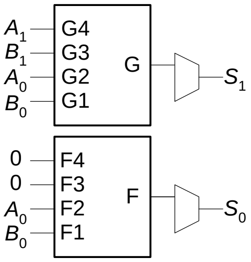
(A_1)	(B_1)	(A_0)	(B_0)	
G4	G3	G2	G1	G
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

			(S_2)
H1	F	G	H
X	0	0	0
X	0	1	1
X	1	0	1
X	1	1	0



		(A_0)	(B_0)	(S_0)
F4	F3	F2	F1	F
X	X	0	0	0
X	X	0	1	1
X	X	1	0	1
X	X	1	1	0

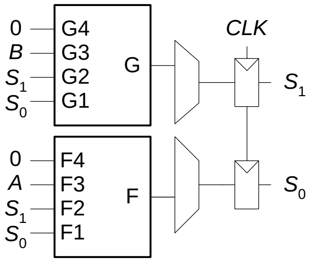
(A_1)	(B_1)	(A_0)	(B_0)	(S_1)
G4	G3	G2	G1	G
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0



5.50 (i) 1 CLB

	(A)	(S_1)	(S_0)	(S_0')
F4	F3	F2	F1	F
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	1
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

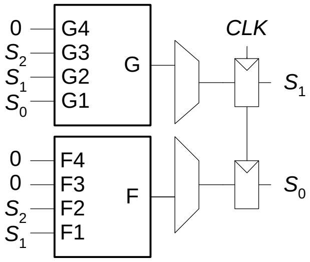
	(B)	(S_1)	(S_0)	(S_1')
G4	G3	G2	G1	G
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	1
X	1	1	0	0
X	1	1	1	0



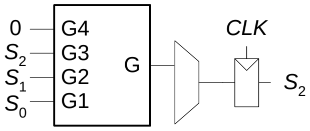
5.50 (j) 3 CLBs

		(S_2)	(S_1)	(S_0')
F4	F3	F2	F1	F
X	X	0	0	1
X	X	0	1	0
X	X	1	0	0
X	X	1	1	1

	(S_2)	(S_1)	(S_0)	(S_1')
G4	G3	G2	G1	G
X	0	0	0	0
X	0	0	1	1
X	0	1	0	1
X	0	1	1	1
X	1	0	0	0
X	1	0	1	0
X	1	1	0	1
X	1	1	1	0



	(S_2)	(S_1)	(S_0)	(S_2')
G4	G3	G2	G1	G
X	0	0	0	0
X	0	0	1	0
X	0	1	0	1
X	0	1	1	0
X	1	0	0	0
X	1	0	1	1
X	1	1	0	1
X	1	1	1	1



5.51

(a) 1 CLB

(b)

$t_{pd} = t_{\text{CLB}}$

$= 2.7$ ns

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

$\geq [2.8 + 2.7 + 3.9]$ 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

$t_{\text{skew}} < (t_{ccq} + t_{cd}) - t_{\text{hold}}$

$< [(2.8 + 2.7) - 0]$ 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.

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

$\geq [2.8 + 2.7 + 3.9 + 5]$ ns

$= 14.4$ ns

$f = 1 / 14.4$ ns = **69 MHz**

5.52

First, we find the cycle time:

$T_c = 1/f = 1/100$ MHz = 10 ns

$T_c \geq t_{pcq} + Nt_{\text{CLB}} + t_{\text{setup}}$

10 ns $\geq [0.72 + N(0.61) + 0.53]$ ns

Thus, $N \leq 14.3$

The maximum number of CLBs on the critical path is 14.

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

$T_c \geq [0.72 + 0.61 + 0.53]$ ns

≥ 1.86 ns

$f = 1 / 1.86$ ns = **537 MHz**

Question 5.1

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

Question 5.2

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

Question 5.3

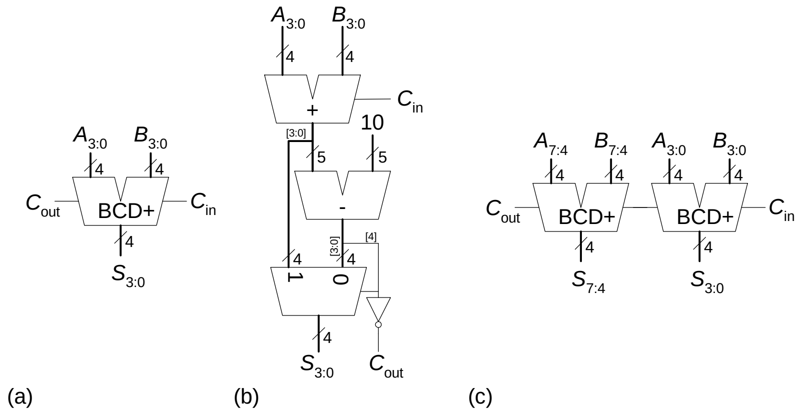


FIGURE 5.22 BCD adder: (a) 4-bit block, (b) underlying hardware, (c) 8-bit BCD adder

(continued from previous page)

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`).

6.2

Yes, it is possible to design a computer architecture without a register set. For example, an architecture could use memory as a very large register set. Each instruction would require a memory access. For example, an add instruction might look like this:

```
add 0x10, 0x20, 0x24
```

This would add the values stored at memory addresses 0x20 and 0x24 and place the result in memory address 0x10. Other instructions would follow the same pattern, accessing memory instead of registers. Some advantages of the architecture are that it would require fewer instructions. Load and store operations are now unnecessary. This would make the decoding hardware simpler and faster.

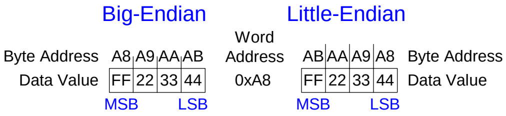
Some disadvantages of this architecture over the MIPS architecture is that each operation would require a memory access. Thus, either the processor would need to be slow or the memory small. Also, because the instructions must encode memory addresses instead of register numbers, the instruction size would be large in order to access all memory addresses. Or, alternatively, each instruction can only access a smaller number of memory addresses. For example, the architecture might require that one of the source operands is also a destination operand, reducing the number of memory addresses that must be encoded.

6.3

(a) $42 \times 4 = 42 \times 2^2 = 101010_2 << 2 = 10101000_2 = \text{0xA8}$

(b) 0xA8 through 0xAB

(c)



6.4

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

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

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

6.5

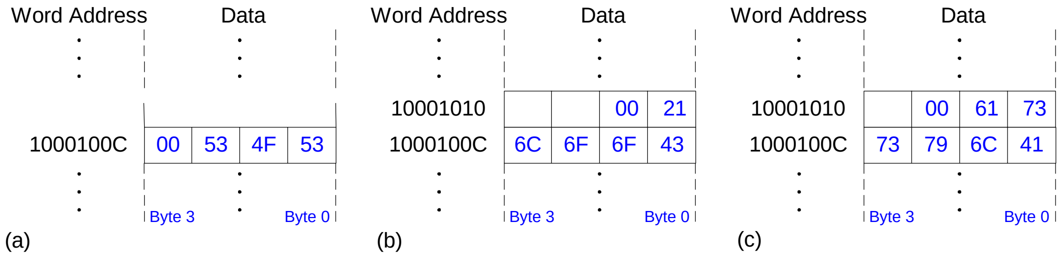
(a) 0x534F5300

(b) 0x436F6F6C2100

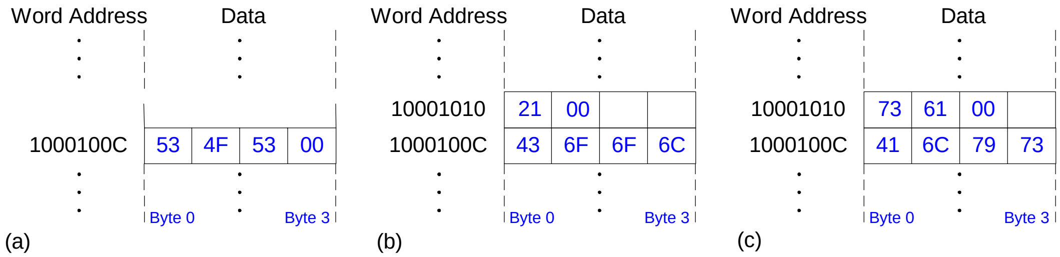
(c) 0x416C7973736100 (depends on the persons name)

6.6

Little-Endian Memory



Big-Endian Memory



6.7
0x02114020
0x8de80020
0x2010fff6

6.8
0x20100049
0xad49fff9
0x02f24822

6.9
(a)

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

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

6.10

The program is:

```
[0x00400000]  0x20080000          addi $t0, $0, 0
[0x00400004]  0x20090001          addi $t1, $0, 1
[0x00400008]  0x0089502a  loop:   slt  $t2, $a0, $t1
[0x0040000c]  0x15400003           bne  $t2, $0, finish
[0x00400010]  0x01094020          add  $t0, $t0, $t1
[0x00400014]  0x21290002          addi $t1, $t1, 2
[0x00400018]  0x08100002          j    loop
[0x0040001c]  0x01001020  finish: add  $v0, $t0, $0
```

An equivalent C program would be (assuming `temp` = `$t0`, `i` = `$t1`, `n` = `$a0`, `result` = `$v0`):

```
temp = 0;
for (i = 1; i<= n; i = i+2)
    temp = temp + i;

result = temp;
```

This program sums the odd integers up to n and places the result in the return register, `$v0`.

6.11

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

6.12

(a)

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

(b)

```
slt  $t0, $s0, $s1 # if g < h, $t0 = 1
bne  $t0, $0, else # if $t0 != 0, do else
addi $s0, $s0, 1   # g = g + 1
j    done          # jump past else block
else:  addi $s1, $s1, -1  # h = h - 1
```

```
done:
```

(c)

```
slt $t0, $s1, $s0  # if h < g, $t0 = 1
bne $t0, $0, else  # if $t0 != 0, do else
add $s0, $0, $0    # g = 0
j   done           # jump past else block
else:   sub $s1, $0, $0    # h = 0
done:
```

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;
}
```

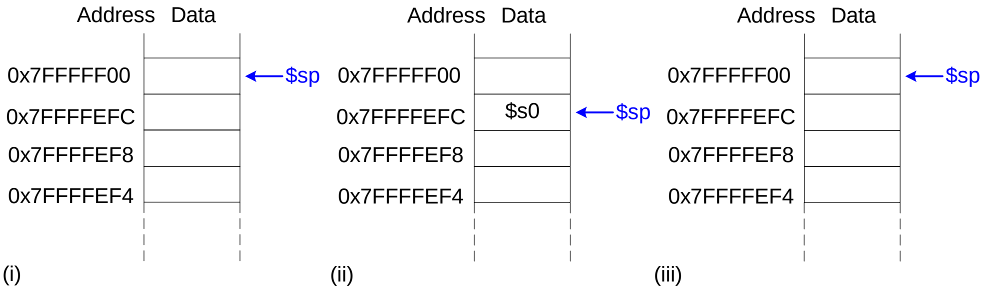
6.14

(a)

```
# MIPS assembly code
# base address of array x = $a0
# base address of array y = $a1
# i = $s0

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

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

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

(b) The stack (i) before, (ii) during, and (iii) after the `strcpy` procedure call.



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
```

6.16

(a)

fib(0) = 0

fib(-1) = 1

(b)

```
int fib(int n)
{
  int prevresult = 1;                  // fib(n-1)
  int result = 0;                      // fib(n)

  while (n != 0) {                    // calculate new fibonacci number
    result = result + prevresult;     // fib(n) = fib(n-1) + fib(n-2)
    prevresult = result - prevresult; // fib(n-1) = fib(n) - fib(n-2)
    n = n - 1;
  }
  return result;
}
```

(c)

```
# fib.asm
# 2/21/07 Sarah Harris (Sarah_Harris@hmc.edu)
#
# The fib() procedure computes the nth Fibonacci number.
# n is passed to fib() in $a0, and fib() returns the result in $v0.

main:  addi $a0,$0,9     # initialize procedure argument: n = 9
       jal  fib          # call fibonacci procedure
       ...               # more code
 fib:  addi $t0,$0,1     # $t0 = fib(n-1) = fib(-1) = 1
       addi $t1,$0,0     # $t1 = fib(n) = fib(0) = 0
loop:  beq  $a0,$0, end  # done?
       add  $t1,$t1,$t0  # Compute next Fib #, fib(n)
       sub  $t0,$t1,$t0  # Update fib(n-1)
       addi $a0,$a0,-1   # decrement n
       j    loop         # Repeat
 end:  add  $v0,$t1,$0   # Put result in $v0
       jr   $ra          # return result
```

6.17

(a) The stack frames of each procedure are:

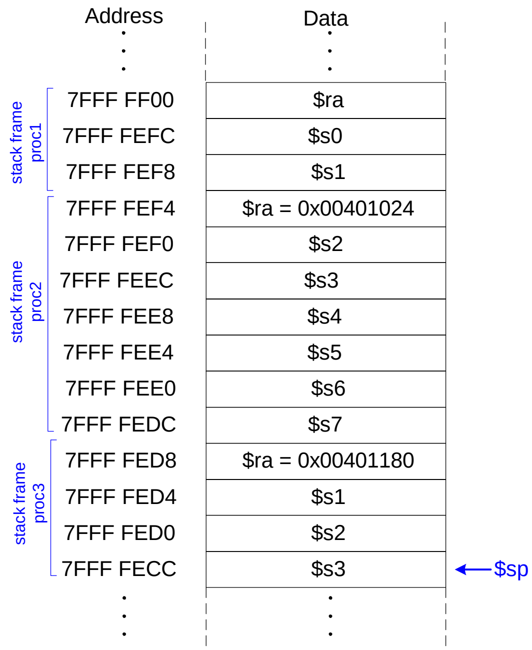
proc1: 3 words deep (for `$s0` - `$s1`, `$ra`)

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.



Address	Data
⋮	⋮
7FFF FF00	$ra
7FFF FEFC	$s0
7FFF FEF8	$s1
7FFF FEF4	$ra = 0x00401024
7FFF FEF0	$s2
7FFF FEEC	$s3
7FFF FEE8	$s4
7FFF FEE4	$s5
7FFF FEE0	$s6
7FFF FEDC	$s7
7FFF FED8	$ra = 0x00401180
7FFF FED4	$s1
7FFF FED0	$s2
7FFF FECC	$s3
⋮	⋮

6.18

(a) `$v0` ends with 19, as desired.

(b) The program will (2) crash. The jump register instruction (`jr $ra`) at instruction address 0x0040004c will return to the most recent value of `$ra` (0x00400030). It will then repeatedly restore values from the stack and increment the stack pointer. Eventually the stack shrinks beyond the dynamic data segment and the program crashes.

(c)

i: The program will (3) produce an incorrect value in `$v0`. `a ($a0)` is not restored, but instead holds the value 3 passed to `f2`. So `f` computes

j + a + f2(b) = 5 + 3 + 3*3 = 17

ii: The program will (4) run correctly despite the deleted lines. `b` is not restored, but is never used again.

iii: (4) `$s0` is not restored. This could corrupt a calling procedure that depended on `$s0` being saved, but does not directly corrupt the result of `f`.

iv: (1) Now `f2` does not move and restore the stack pointer, so values saved to the stack get overwritten. When `f2` tries to return, it always finds `$ra` pointing to 0x04000078. Therefore, we enter an infinite loop from 0x04000078 to 0x0400008c

v: (3) `$s0` is overwritten with the value 3 in `f2` and not restored. Thus, when we return to `f1`, `j`=3 instead of the desired 5. So j + a + f2(b) = 3 + 5 + 3*3 = 17

vi: (2) As in case iv, `$ra` is left pointing to 0x04000078 rather than 0x04000030 when we are all done with the recursion on `f2`. Therefore, we get in an infinite loop from 0x04000078 to 0x0400008c. However, on each loop `$sp` is incremented by 12. Eventually it points to an illegal memory location and the program crashes.

vii: (4) `x` is not restored in `f2` after it recursively calls itself. But `x` is not used after the recursive call, so the program still produces the correct answer.

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

6.20

(a)

```
0x00400028         add  $a0, $a1, $0    # 0x00a02020
0x0040002c         jal  f2              # 0x0C10000D
0x00400030    f1:  jr   $ra             # 0x03e00008
0x00400034    f2:  sw   $s0, 0($s2)     # 0xae500000
0x00400038         bne  $a0, $0, else   # 0x14800001
0x0040003c         j    f1              # 0x0810000C
0x00400040  else:  addi $a0, $a0, -1    # 0x2084FFFF
0x00400044         j    f2              # 0x0810000D
```

(b)

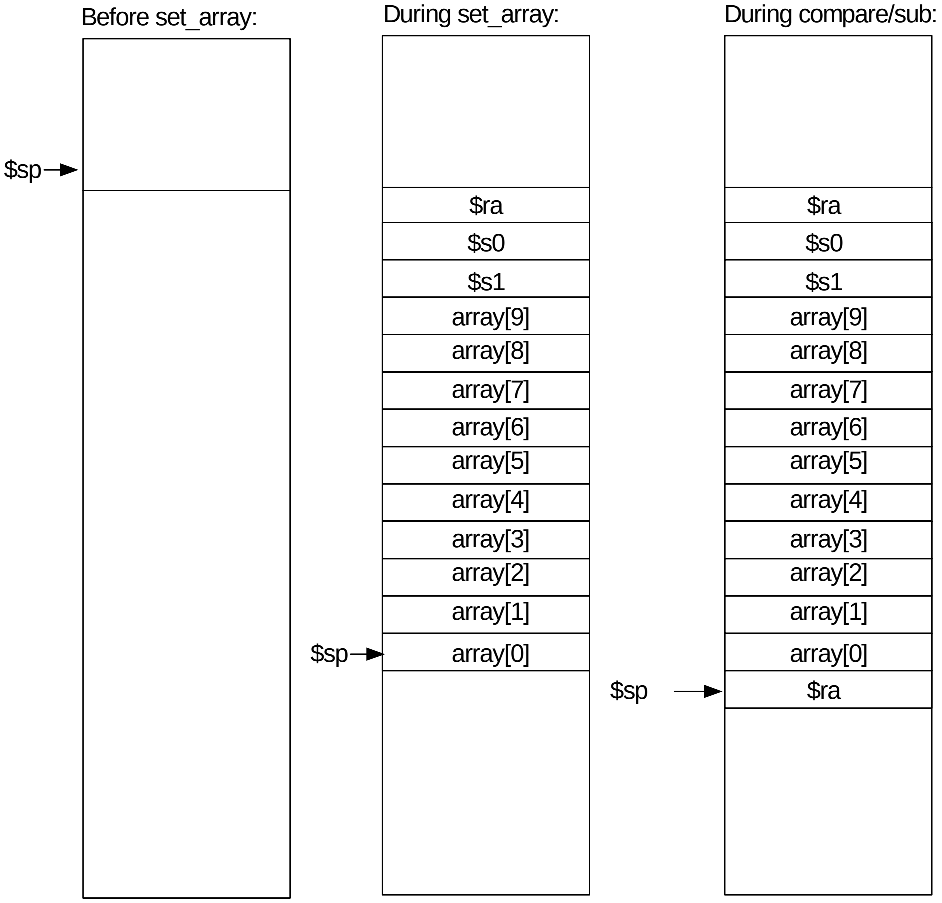
```
0x00400028         add  $a0, $a1, $0    # register only
0x0040002c         jal  f2              # pseudo-direct
0x00400030    f1:  jr   $ra             # register only
0x00400034    f2:  sw   $s0, 0($s2)     # base addressing
0x00400038         bne  $a0, $0, else   # PC-relative
0x0040003c         j    f1              # pseudo-direct
0x00400040  else:  addi $a0, $a0, -1    # immediate
0x00400044         j    f2              # pseudo-direct
```

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.

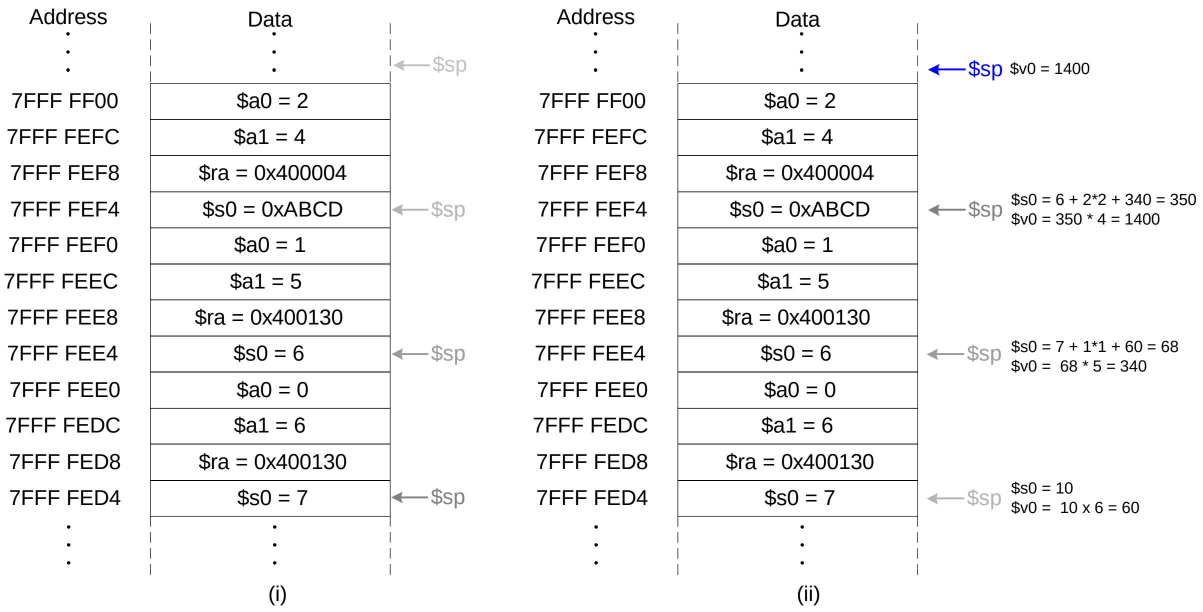
6.22

(a)

```
# MIPS assembly code
0x400100 f:     addi $sp, $sp, -16  # decrement stack
0x400104        sw   $a0, 0xc($sp)  # save registers on stack
0x400108        sw   $a1, 0x8($sp)
```

```
0x40010C        sw   $ra, 0x4($sp)
0x400110        sw   $s0, 0x0($sp)
0x400114        addi $s0, $a1, 2     # b = k + 2
0x400118        bne  $a0, $0, else   # if (n!=0) do else block
0x40011C        addi $s0, $0, 10     # b = 10
0x400120        j    done

0x400124 else:  addi $a0, $a0, -1    # update arguments
0x400128        addi $a1, $a1, 1
0x40012C        jal  f               # call f()
0x400130        lw   $a0, 0xc($sp)   # restore arguments
0x400134        lw   $a1, 0x8($sp)
0x400138        mult $a0, $a0        # {[hi],[lo]} = n*n
0x40013C        mflo $t0             # $t0 = lo  (assuming 32-bit result)
0x400140        add  $s0, $s0, $t0   # b = b + n*n
0x400144        add  $s0, $s0, $v0   # b = b + n*n  + f(n-1,k+1)

0x400148 done:  mult $s0, $a1        # {[hi],[lo]} = b * k
0x40014C        mflo $v0             # $v0 = lo (assuming 32-bit result)
0x400150        lw   $ra, 0x4($sp)   # restore registers
0x400154        lw   $s0, 0x0($sp)
0x400158        addi $sp, $sp, 16    # restore stack pointer
0x40015C        jr   $ra             # return
```

6.22 (b) The stack (i) after the last recursive call, and (ii) after return. The final value of `$v0` is 1400.



(i)

Address	Data	
⋮	⋮	← $sp
7FFF FF00	$a0 = 2	
7FFF FEFC	$a1 = 4	
7FFF FEF8	$ra = 0x400004	
7FFF FEF4	$s0 = 0xABCD	← $sp
7FFF FEF0	$a0 = 1	
7FFF FEEC	$a1 = 5	
7FFF FEE8	$ra = 0x400130	
7FFF FEE4	$s0 = 6	← $sp
7FFF FEE0	$a0 = 0	
7FFF FEDC	$a1 = 6	
7FFF FED8	$ra = 0x400130	
7FFF FED4	$s0 = 7	← $sp
⋮	⋮	

(ii)

Address	Data		
⋮	⋮	← $sp	$v0 = 1400
7FFF FF00	$a0 = 2		
7FFF FEFC	$a1 = 4		
7FFF FEF8	$ra = 0x400004		
7FFF FEF4	$s0 = 0xABCD	← $sp	$s0 = 6 + 2*2 + 340 = 350 $v0 = 350 * 4 = 1400
7FFF FEF0	$a0 = 1		
7FFF FEEC	$a1 = 5		
7FFF FEE8	$ra = 0x400130		
7FFF FEE4	$s0 = 6	← $sp	$s0 = 7 + 1*1 + 60 = 68 $v0 = 68 * 5 = 340
7FFF FEE0	$a0 = 0		
7FFF FEDC	$a1 = 6		
7FFF FED8	$ra = 0x400130		
7FFF FED4	$s0 = 7	← $sp	$s0 = 10 $v0 = 10 x 6 = 60
⋮	⋮		

6.23

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

6.24

(a) In the worst case, a jump instruction can only jump one instruction forward. For example, the following code is impossible. The jump instruction (`j loop`) below can only jump forward to 0x0FFFFFFC (at most).

```
0x0FFFFFF8         j    loop
0x0FFFFFFC
0x10000000  loop:  ...
```

(b) In the best case, a jump instruction can jump forward 2^{26} instructions. For example,

```
0x0FFFFFFC         j    loop
0x10000000         ...
0x1FFFFFFC  loop:  ...
```

(c)

In the worst case, a jump instruction cannot jump backward. For example, the following code is impossible. Because the jump instruction appends the four most significant bits of PC + 4, this jump instruction cannot even jump to itself, let alone further backwards.

```
0x0FFFFFFC  loop:  j    loop
0x10000000         ...
```

(d) In the best case, a jump instruction can jump backward at most 2^{26} - 2 instructions. For example,

```
0x10000000  loop:  ...
...                ...
0x1FFFFFF8         j    loop
```

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.

6.26

```
0x00400000           j Loop1
...
0x0FFFFFFC    Loop1:  j Loop2
...
0x10400000    Loop2:  ...
```

Another option:

```
0x00400000    lui $t1, 0x1040
0x00400004    jr  $t1
```

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:
```

6.28

(a)

```
void concat(char[] string1, char[] string2, char[] stringconcat)
{
  int i, j;
  i = 0;
  j = 0;

  while (string1[i] != 0) {
    stringconcat[i] = string1[i];
    i = i + 1;
  }
  while (string2[j] != 0) {
    stringconcat[i] = string2[j];
    i = i + 1;
    j = j + 1;
  }
  stringconcat[i] = 0;  // append null at end of string
}
```

(b)

```
concat:  lb   $t0, 0($a0)      # $t0 = string1[i]
         beq  $t0, $0, string2 # if end of string1, go to string2
         sb   $t0, 0($a2)      # stringconcat[i] = string1[i]
         addi $a0, $a0, 1      # increment index into string1
         addi $a2, $a2, 1      # increment index into stringconcat
         j    concat           # loop back
string2: lb   $t0, 0($a1)      # $t0 = string2[j]
         beq  $t0, $0, done    # if end of string2, return
         sb   $t0, 0($a2)      # stringconcat[j] = string2[j]
         addi $a1, $a1, 1      # increment index into string2
         addi $a2, $a2, 1      # increment index into stringconcat
done:    sb   $0, 0($a2)       # append null to end of string
         jr   $ra
```

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
```

6.30

(a)

```
0x00400000  main:   lw  $a0, x
0x00400004          lw  $a1, y
0x00400008          jal diff
0x0040000C          jr  $ra
0x00400010  diff:   sub $v0, $a0, $a1
0x00400014          jr  $ra
```

(b)

symbol	address
x	0x10000000
y	0x10000004
main	0x00400000
diff	0x00400010

TABLE 6.1 Symbol table

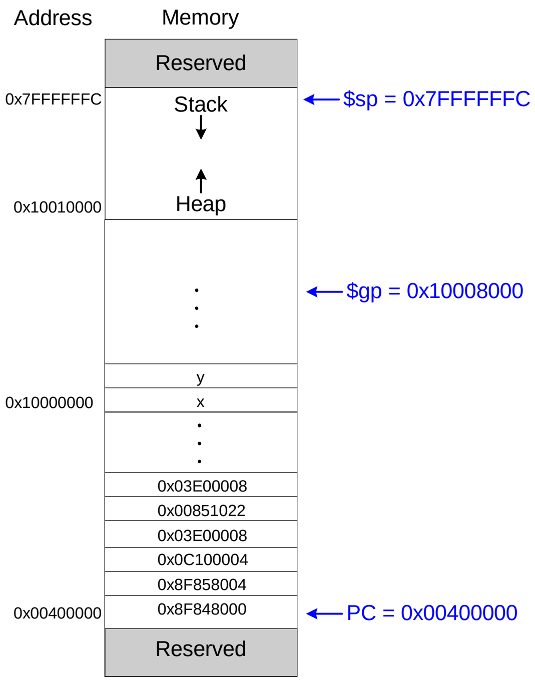
(c)

Executable file header	Text Size	Data Size	
	0x18 (24 bytes)	0x8 (8 bytes)	
Text segment	Address	Instruction	
	0x00400000	0x8F848000	lw $a0, x
	0x00400004	0x8F858004	lw $a1, y
	0x00400008	0x0C100004	jal diff
	0x0040000C	0x03E00008	jr $ra
	0x00400010	0x00851022	sub $v0, $a0, $a1
	0x00400014	0x03E00008	jr $ra
Data segment	Address	Data	
	0x10000000	x	
	0x10000004	y	

(d)

The data segment is 8 bytes and the text segment is 24 (0x18) bytes.

6.30 (e)



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
```

6.31 (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 6.2

High-Level Code

```
// algorithm for finding subset of array with
// largest sum


max = -2,147,483,648; // -2^31
start = 0;
end = 0;



for (i=0; i<length; i=i+1) {

  sum = 0;


  for (j=i; j<length; j=j+1) {




   sum = sum + array[j];

    if (sum > max) {
      max = sum;
      start = i;
      end = j;
    }
  }
}


count = 0;
for ( i = start; i <= end; i=i+1) {
  array2[count] = array[i];
  count = count + 1;
}
```

MIPS Assembly Code

```
# $a0 = base address of array, $a1 = length of array
# $t0 = max, $t1 = start, $t2 = end
# $t3 = i, $t4 = j, $t5 = sum

        li   $t0, 0x80000000 # $t0 = large neg #
        addi $t1, $0, 0      # start = 0
        addi $t2, $0, 0      # end = 0
        addi $t3, $0, -4     # i = -4
        sll  $a1, $a1, 2     # length = length*4

loop1:  addi $t3, $t3, 4     # i = i+4
        slt  $t6, $t3, $a1   # i<length?
        beq  $t6, $0, finish # branch if not
        addi $t5, $0, 0      # reset sum
        addi $t4, $t3, -4    # j = i-4
loop2:  addi $t4, $t4, 4     # j = j+4
        slt  $t6, $t4, $a1   # j<length?
        beq  $t6, $0, loop1  # branch if not
        add  $t6, $a0, $t4   # $t6 = &array[j]
        lw   $t6, 0($t6)     # $t6 = array[j]
        add  $t5, $t5, $t6   # sum = sum + $t6
        slt  $t6, $t0, $t5   # max < sum?
        beq  $t6, $0, loop2  # branch if not
        add  $t0, $t5, $0    # max = sum
        add  $t1, $t3, $0    # start = i
        add  $t2, $t4, $0    # end = j
        j    loop2


finish: addi $t3, $t1, -4    # i = start - 4
loop3:  add  $t3, $t3, 4     # i = i + 4
        slt  $t6, $t2, $t3   # end < i?
        bne  $t6, $0, return # if yes, return
        add  $t6, $t3, $a0   # $t6 = &array[i]
        lw   $t6, 0($t6)     # $t6 = array[i]
       sw   $t6, 0($a2)    # array2[count] = array[i]
        add  $a2, $a2, 4     # increment count
        j    loop3

return: jr   $ra
```

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 6.4

High-Level Code

```
int count = 0;

while (num != 0) {
  if (num && 0x1)  // if num AND 0x1
    count = count + 1;

 num = num >> 1;
}
```

MIPS Assembly Code

```
# $a0 = num, $v0 = count
        add  $v0, $0, $0     # count = 0

count:  beq  $a0, $0, done   # if num == 0, done
        andi $t0, $a0, 0x1   # $t0 = num AND 0x1
        beq  $t0, $0, shift  # if 0, only shift
        addi $v0, $v0, 1     # count = count + 1
shift:  srl $a0, $a0, 1      # num = num >> 1
        j   count

done:
```

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 6.6

```
  addu $t4, $t2, $t3
  xor  $t5, $t2, $t3        # compare sign bits
  srl  $t5, $t5, 31         # $t5 = 1 if sign bits different
  bne  $t5, $0, nooverflow
  xor  $t5, $t4, $t3        # compare with result sign bit
  srl  $t5, $t5, 31         # $t5 = 0 if sign bits same
  beq  $t5, $0, nooverflow
overflow:

nooverflow:
```

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`

7.2

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

7.3

(a) `sll`

First, we modify the ALU.

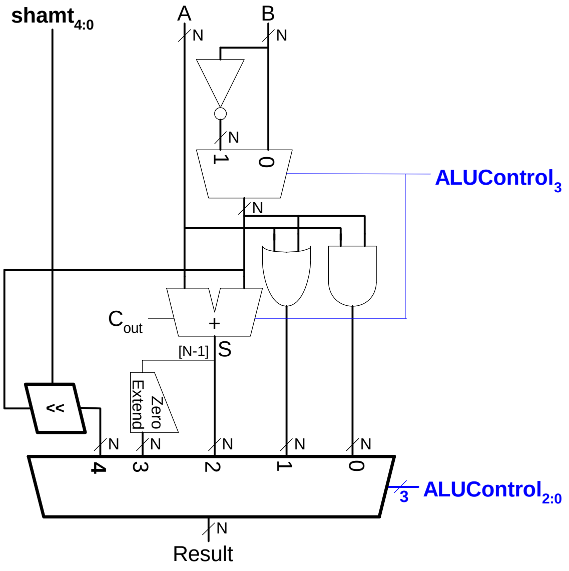


FIGURE 7.1 Modified ALU to support `sll`

ALUControl$_{3:0}$	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	000000 (`sll`)	0100 (shift left logical)

TABLE 7.2 ALU decoder truth table

Then we modify the datapath.

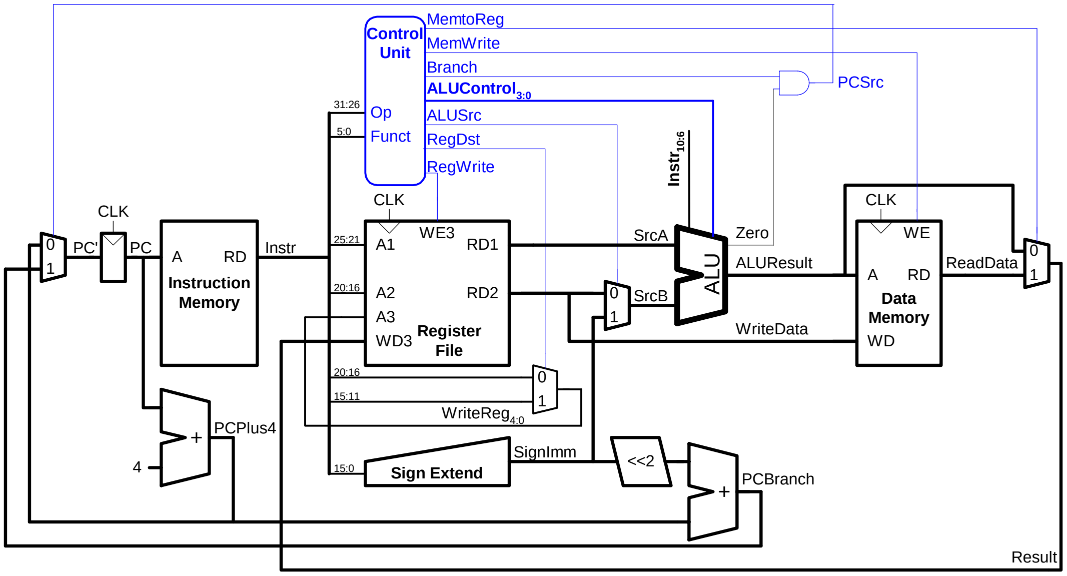


FIGURE 7.2 Modified single-cycle MIPS processor extended to run `sll`

7.3 (b) `lui`

Note: the 5-bit rs field of the `lui` instruction is 0.

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`

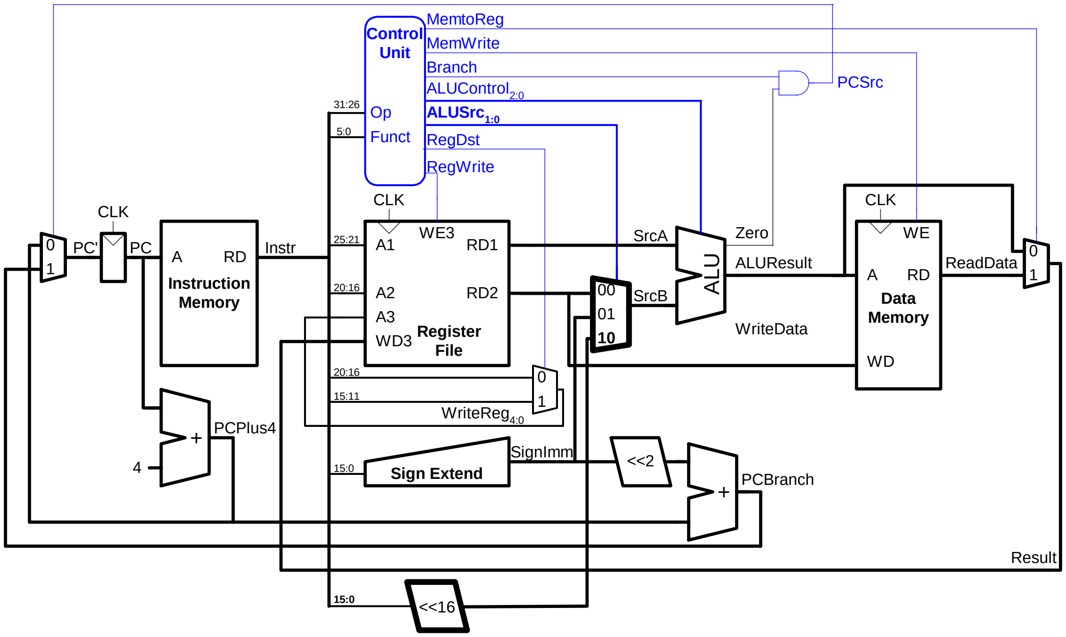


FIGURE 7.3 Modified single-cycle datapath to support `lui`

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)**

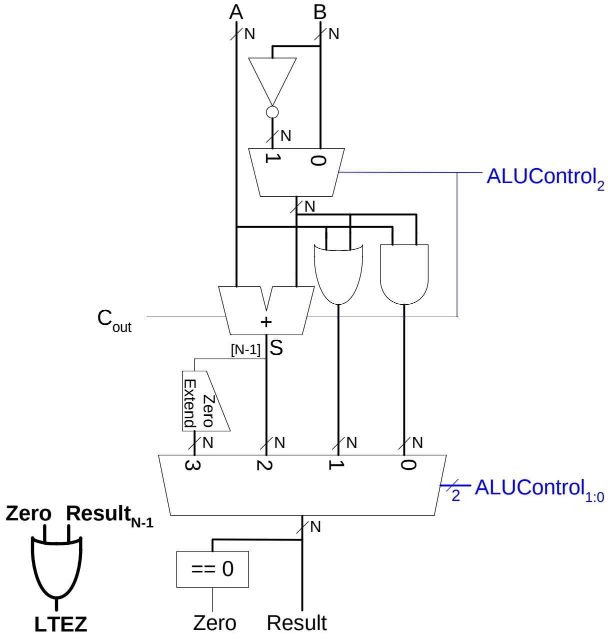
TABLE 7.4 ALU decoder truth table

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`	**001010**	**1**	**0**	**1**	**0**	**0**	**0**	**11**

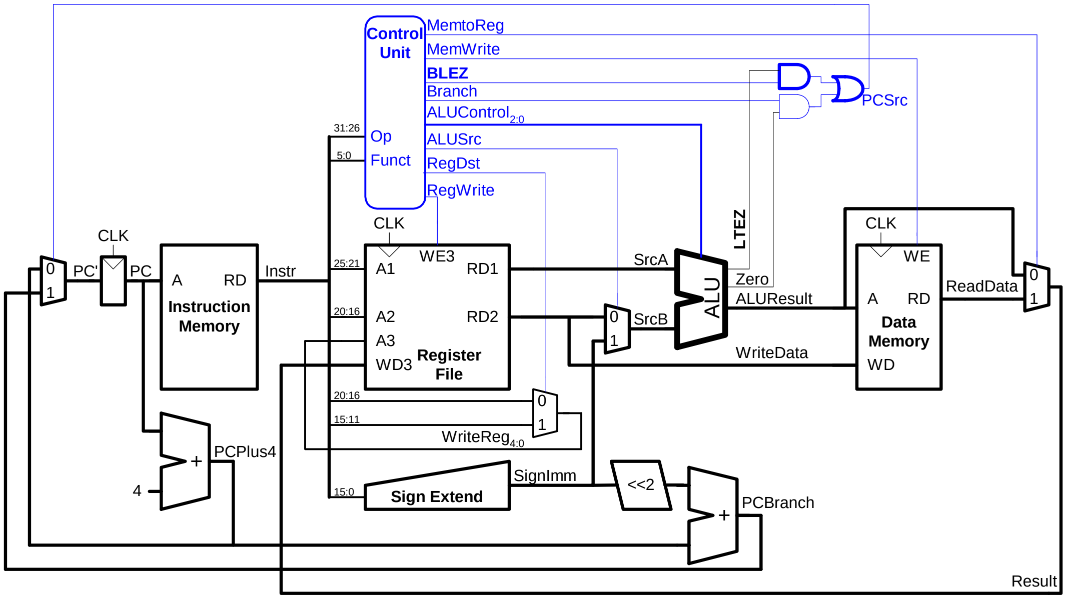
TABLE 7.5 Main decoder truth table enhanced to support `slti`

7.3 (d) `blez`

First, we modify the ALU



Then, we modify the datapath

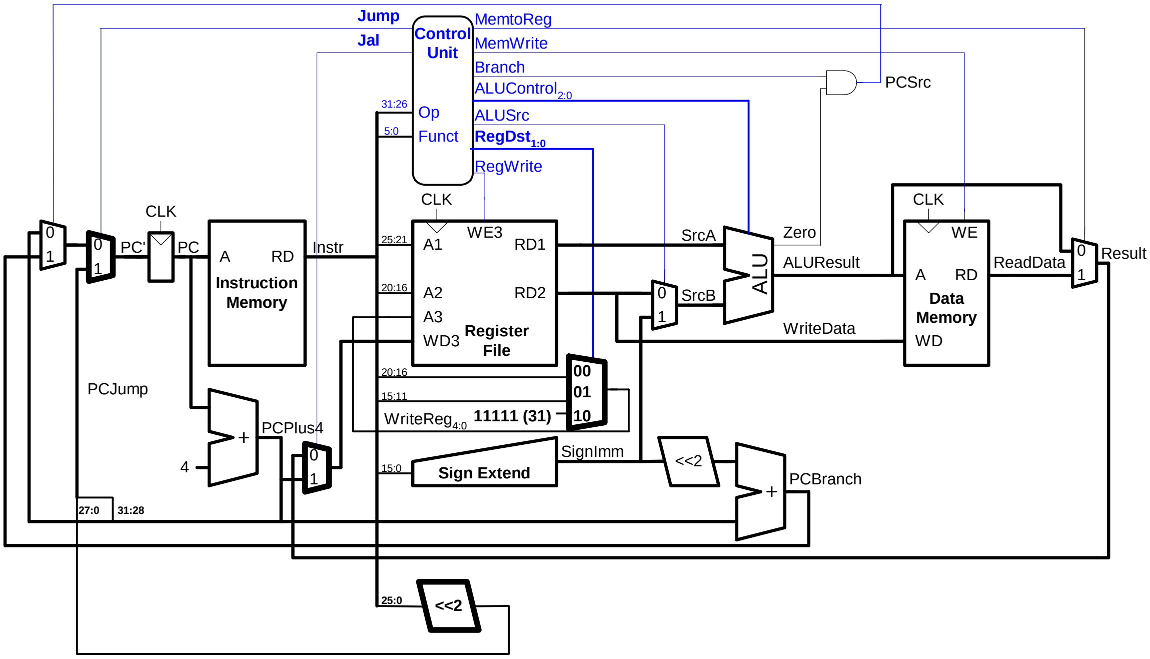


.

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	**X**X	1	0	1	X	00	**0**	**0**
`beq`	000100	0	**X**X	0	1	0	X	01	**0**	**0**
`addi`	001000	1	**00**	1	0	0	0	00	**0**	**0**
`j`	000010	0	**X**X	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`

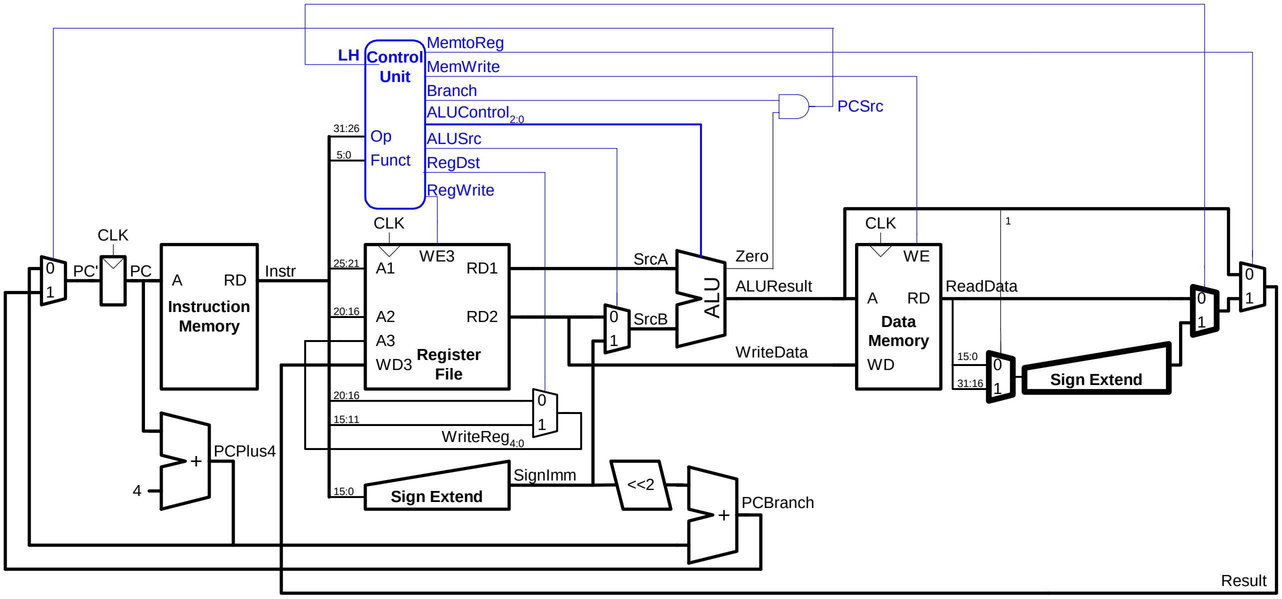


FIGURE 7.4 Modified single-cycle datapath to support `lh`

.

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`

7.4

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

We modify the register file and datapath as shown in Figure 7.5.

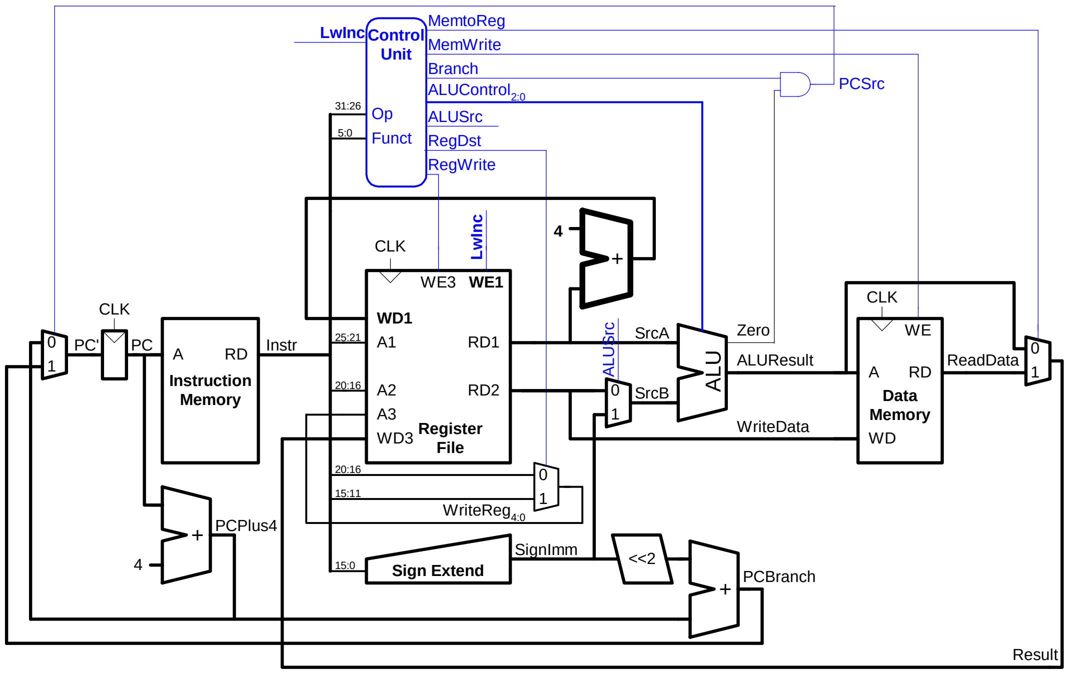
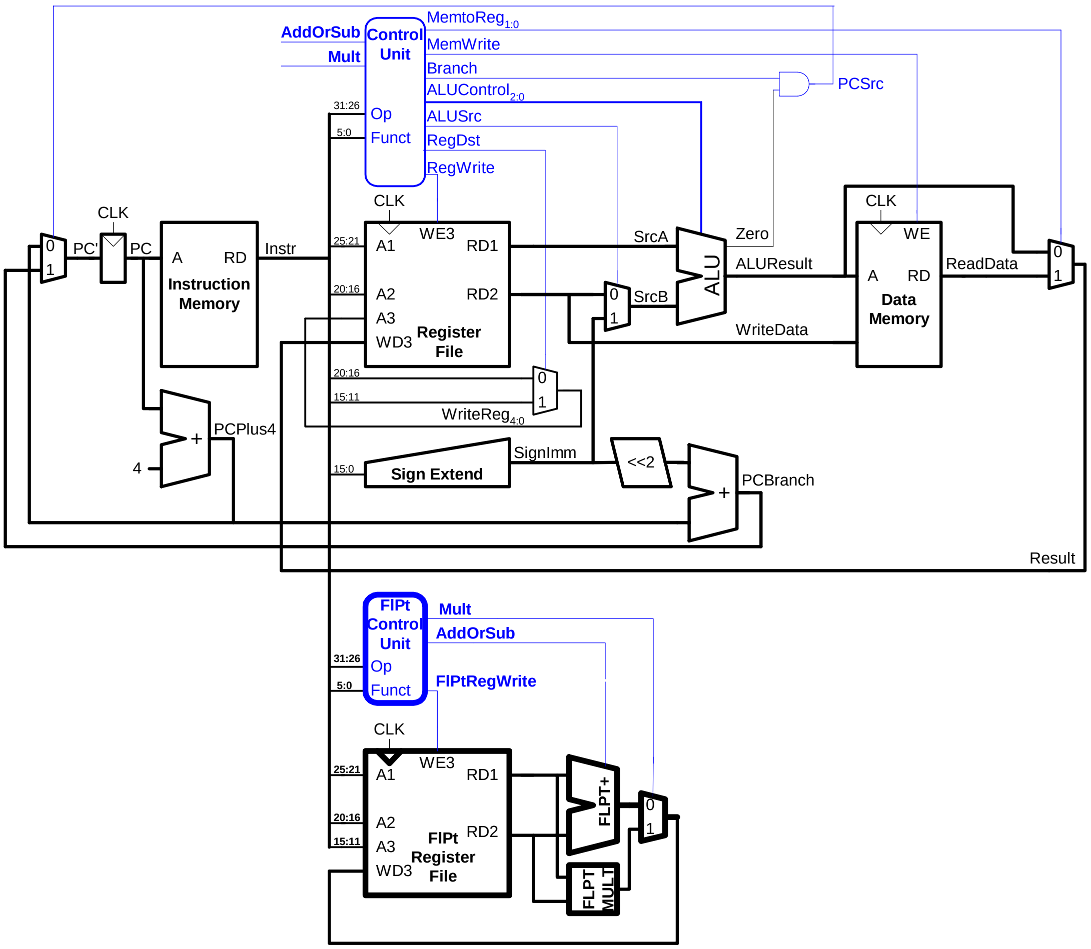


FIGURE 7.5 Modified datapath for Exercise 7.4

Instruction	opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	**Lwinc**
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**
`lwinc`		**1**	**0**	**1**	**0**	**0**	**1**	**00**	**1**

TABLE 7.9 Main decoder truth table enhanced to support `lwinc`

7.5



Instruction	opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	00	10
`lw`	100011	1	0	1	0	0	01	00
`sw`	101011	0	X	1	0	1	XX	00

TABLE 7.10 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	X	0	1	0	XX	01
`f-type`	**010001**	**0**	**X**	**X**	**0**	**0**	**XX**	**XX**

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

Instruction	opcode	FlPtRegWrite
`f-type`	**010001**	**1**
`others`		**0**

TABLE 7.11 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 7.12 Adder/subtractor decoder truth table

7.6

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

$T_c = t_{pcq\_PC} + 2t_{mem} + t_{RF\text{read}} + t_{\text{mux}} + t_{\text{ALU}} + t_{RF\text{setup}}$

$= 30 + 2(250) + 150 + 25 + 200 + 20 =$ **925ps**

The unit that your friend could speed up that would make the largest reduction in cycle time would be the memory unit. So tmem_new = 125ps, and the new cycle time is:

$\boldsymbol{T_c}$ **= 675 ps**

7.7 From Equation 7.3, page 380 in the text (see Errata):
$T_c = t_{pcq\_PC} + 2t_{mem} + t_{RF\text{read}} + t_{\text{mux}} + t_{\text{ALU}} + t_{RF\text{setup}}$
= 30 + 2(250) + 150 + 25 + 180 + 20 = **905ps**

It would take **90.5 seconds** to execute 100 billion instructions.

7.8
(a) `lw`
(b) `beq`
(c) `beq, j`

7.9
(a) R-type, `addi`
(b) `lw`, `sw`, `addi`, R-type
(c) all instructions

7.10
We modify the HDL for the single-cycle MIPS processor to include all instructions from Exercise 7.3.

Single Cycle Processor

Verilog

```
module mipssingle(input         clk, reset,
                 output [31:0] pc,
                 input  [31:0] instr,
                 output        memwrite,
                output [31:0] aluresult, writedata,
                 input  [31:0] readdata);

  wire        memtoreg;
  wire [1:0]  alusrc;  // LUI
  wire [1:0]  regdst;  // JAL
  wire        regwrite, jump, pcsrc, zero;
  wire [3:0]  alucontrol;  // SLL
  wire        ltez;  // BLEZ
  wire        jal;   // JAL
  wire        lh;    // LH

  controller c(instr[31:26], instr[5:0], zero,
              memtoreg, memwrite, pcsrc,
              alusrc, regdst, regwrite, jump,
              alucontrol,
              ltez,  // BLEZ
              jal,   // JAL
              lh);   // LH
  datapath dp(clk, reset, memtoreg, pcsrc,
             alusrc, regdst, regwrite, jump,
             alucontrol,
             zero, pc, instr,
             aluresult, writedata, readdata,
             ltez,  // BLEZ
             jal,   // JAL
             lh);   // LH
endmodule
```

VHDL

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

architecture struct of mipssingle is
  component controller
    port(op, funct:          in  STD_LOGIC_VECTOR(5 downto 0);
         zero:               in  STD_LOGIC;
         memtoreg, memwrite: out STD_LOGIC;
         pcsrc:              out STD_LOGIC;
         alusrc:             out STD_LOGIC_VECTOR(1 downto 0); --LUI
         regdst:             out STD_LOGIC_VECTOR(1 downto 0); --JAL
         regwrite:           out STD_LOGIC;
         jump:               out STD_LOGIC;
         alucontrol:         out STD_LOGIC_VECTOR(3 downto 0); --SLL
         ltez:               inout STD_LOGIC;                 --BLEZ
         jal:                out STD_LOGIC;                    --JAL
         lh:                 out STD_LOGIC);                   --LH
  end component;
  component datapath
    port(clk, reset:        in  STD_LOGIC;
         memtoreg, pcsrc:   in  STD_LOGIC;
         alusrc:            in  STD_LOGIC_VECTOR(1 downto 0); --LUI
         regdst:            in  STD_LOGIC_VECTOR(1 downto 0); --JAL
         regwrite, jump:    in  STD_LOGIC;
         alucontrol:        in  STD_LOGIC_VECTOR(3 downto 0); --SLL
         zero:              inout STD_LOGIC;
         pc:                inout STD_LOGIC_VECTOR(31 downto 0);
         instr:             in  STD_LOGIC_VECTOR(31 downto 0);
         aluresult, writedata: inout STD_LOGIC_VECTOR(31 downto 0);
         readdata:          in  STD_LOGIC_VECTOR(31 downto 0);
         ltez:              out STD_LOGIC;  --BLEZ
         jal:               in  STD_LOGIC;  --JAL
         lh:                in  STD_LOGIC); --LH
  end component;
  signal memtoreg: STD_LOGIC;
  signal alusrc: STD_LOGIC_VECTOR(1 downto 0); --LUI
  signal regdst: STD_LOGIC_VECTOR(1 downto 0); --JAL
  signal regwrite, jump: STD_LOGIC;
  signal pcsrc, zero: STD_LOGIC;
  signal alucontrol: STD_LOGIC_VECTOR(3 downto 0); --SLL
  signal ltez: STD_LOGIC; --BLEZ
  signal jal: STD_LOGIC;  --JAL
  signal lh: STD_LOGIC;   --LH
begin
  cont: controller port map(instr(31 downto 26), instr(5 downto 0),
                            zero, memtoreg, memwrite, pcsrc, alusrc,
                            regdst, regwrite, jump, alucontrol,
                            ltez,  --BLEZ
                            jal,   --JAL
                            lh);   --LH
  dp: datapath port map(clk, reset, memtoreg, pcsrc, alusrc, regdst,
                        regwrite, jump, alucontrol, zero, pc, instr,
                        aluresult, writedata, readdata,
                        ltez,   --BLEZ
                        jal,    --JAL
                        lh);    --LH
end;
```

Controller

Verilog

```
module controller(input  [5:0] op, funct,
                  input        zero,
                  output       memtoreg, memwrite,
                  output       pcsrc,
                  output [1:0] alusrc,      // LUI
                  output [1:0] regdst,      // JAL
                  output       regwrite,
                  output       jump,
                  output [3:0] alucontrol,  // SLL
                  input        ltez,        // BLEZ
                  output       jal,         // JAL
                  output       lh);         // LH

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

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

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

endmodule
```

VHDL

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


architecture struct of controller is
  component maindec
  port(op:                 in  STD_LOGIC_VECTOR(5 downto 0);
       memtoreg, memwrite: out STD_LOGIC;
       branch:             out STD_LOGIC;
       alusrc:             out STD_LOGIC_VECTOR(1 downto 0); --LUI
       regdst:             out STD_LOGIC_VECTOR(1 downto 0); --JAL
       regwrite:           out STD_LOGIC;
       jump:               out STD_LOGIC;
       aluop:              out STD_LOGIC_VECTOR(1 downto 0);
       blez:               out STD_LOGIC;  --BLEZ
       jal:                out STD_LOGIC;  --JAL
       lh:                 out STD_LOGIC); --LH
  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(3 downto 0));  --SLL
  end component;
  signal aluop: STD_LOGIC_VECTOR(1 downto 0);
  signal branch: STD_LOGIC;
  signal blez: STD_LOGIC;  --BLEZ
begin
  md: maindec port map(op, memtoreg, memwrite, branch,
                      alusrc, regdst, regwrite, jump, aluop,
                      blez, jal, lh);  --BLEZ, JAL, LH
  ad: aludec port map(funct, aluop, alucontrol);

  pcsrc <= (branch and zero) or (blez and ltez);  --BLEZ
end;
```

Main Decoder

Verilog

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

  // increase control width for LUI, BLEZ, JAL, LH
  reg [13:0] controls;

  assign {regwrite, regdst, alusrc,
          branch, memwrite,
          memtoreg, jump, aluop,
          blez,   // BLEZ
          jal,    // JAL
          lh}     // LH
          = controls;

  always @(*)
    case(op)
      6'b000000: controls <= 14'b10100000010000;
//Rtype
      6'b100011: controls <= 14'b10001001000000;
//LW
      6'b101011: controls <= 14'b00001010000000;
//SW
      6'b000100: controls <= 14'b00000100001000;
//BEQ
      6'b001000: controls <= 14'b10001000000000;
//ADDI
      6'b000010: controls <= 14'b00000000100000;
//J
      6'b001010: controls <= 14'b10001000011000;
//SLTI
      6'b001111: controls <= 14'b10010000000000;
//LUI
      6'b000110: controls <= 14'b00000000001100;
//BLEZ
      6'b000011: controls <= 14'b11000000100010;
//JAL
      6'b100001: controls <= 14'b10001001000001;
//LH
     default:   controls <= 14'bxxxxxxxxxxxxxx; //???
    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:             out STD_LOGIC;
       alusrc:             out STD_LOGIC_VECTOR(1 downto 0); --LUI
       regdst:             out STD_LOGIC_VECTOR(1 downto 0); --JAL
       regwrite:           out STD_LOGIC;
       jump:               out STD_LOGIC;
       aluop:              out STD_LOGIC_VECTOR(1 downto 0);
       blez:               out STD_LOGIC;  --BLEZ
       jal:                out STD_LOGIC;  --JAL
       lh:                 out STD_LOGIC); --LH
end;

architecture behave of maindec is
  -- increase number of control signals for LUI, BLEZ, JAL, LH
  signal controls: STD_LOGIC_VECTOR(13 downto 0);
begin
  process(op) begin
    case op is
      when "000000" => controls <= "10100000010000"; --Rtype
      when "100011" => controls <= "10001001000000"; --LW
      when "101011" => controls <= "00001010000000"; --SW
      when "000100" => controls <= "00000100001000"; --BEQ
      when "001000" => controls <= "10001000000000"; --ADDI
      when "000010" => controls <= "00000000100000"; --J
      when "001010" => controls <= "10001000011000"; --SLTI
      when "001111" => controls <= "10010000000000"; --LUI
      when "000110" => controls <= "00000000001100"; --BLEZ
      when "000011" => controls <= "11000000100010"; --JAL
      when "100001" => controls <= "10001001000001"; --LH
      when others   => controls <= "--------------"; -- illegal op
    end case;
  end process;

  regwrite <= controls(13);
  regdst   <= controls(12 downto 11);
  alusrc   <= controls(10 downto 9);
  branch   <= controls(8);
  memwrite <= controls(7);
  memtoreg <= controls(6);
  jump     <= controls(5);
  aluop    <= controls(4 downto 3);
  blez     <= controls(2); --BLEZ
  jal      <= controls(1); --JAL
  lh       <= controls(0); --LH
end;
```

ALU Decoder

Verilog

```
module aludec(input      [5:0] funct,
              input      [1:0] aluop,
              output reg [3:0] alucontrol);
                  // increase to 4 bits for SLL

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

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(3 downto 0));  --SLL
end;

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

Datapath

Verilog

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

  wire [4:0]  writereg;
  wire [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
  wire [31:0] signimm, signimmsh;
  wire [31:0] upperimm;  // LUI
  wire [31:0] srca, srcb;
  wire [31:0] result;
  wire [31:0] writeresult;  // JAL
  wire [15:0] half;         // LH
  wire [31:0] signhalf, memdata;     // LH

  // next PC logic
  flopr #(32) pcreg(clk, reset, pcnext, pc);
  adder       pcadd1(pc, 32'b100, pcplus4);
  sl2         immsh(signimm, signimmsh);
  adder       pcadd2(pcplus4, signimmsh, pcbranch);
  mux2 #(32)  pcbrmux(pcplus4, pcbranch, pcsrc,
                      pcnextbr);
  mux2 #(32)  pcmux(pcnextbr, {pcplus4[31:28],
                    instr[25:0], 2'b00},
                    jump, pcnext);

  // register file logic
  regfile     rf(clk, regwrite, instr[25:21],
                 instr[20:16], writereg,
                 writeresult,
                 srca, writedata);

  mux2 #(32)  wamux(result, pcplus4, jal,
                    writeresult);  // JAL
 mux3 #(5)   wrmux(instr[20:16], instr[15:11], 5'd31,
                    regdst, writereg);  // JAL
```

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;
       memtoreg, pcsrc:   in  STD_LOGIC;
       alusrc:            in  STD_LOGIC_VECTOR(1 downto 0); --LUI
       regdst:            in  STD_LOGIC_VECTOR(1 downto 0); --JAL
       regwrite, jump:    in  STD_LOGIC;
       alucontrol:        in  STD_LOGIC_VECTOR(3 downto 0); --SLL
       zero:              inout STD_LOGIC;
       pc:                inout STD_LOGIC_VECTOR(31 downto 0);
       instr:             in  STD_LOGIC_VECTOR(31 downto 0);
       aluresult, writedata: inout STD_LOGIC_VECTOR(31 downto 0);
       readdata:          in  STD_LOGIC_VECTOR(31 downto 0);
       ltez:              out STD_LOGIC;  --BLEZ
       jal:               in  STD_LOGIC;  --JAL
       lh:                in  STD_LOGIC); --LH
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); --SLL
         shamt: in    STD_LOGIC_VECTOR(4 downto 0); --SLL
         Y:    inout STD_LOGIC_VECTOR(31 downto 0);
         Zero: inout  STD_LOGIC;  --BLEZ
         ltez: out    STD_LOGIC); --BLEZ
  end component;
  component regfile
    port(clk:           in  STD_LOGIC;
         we3:           in  STD_LOGIC;
         ra1, ra2, wa3: in  STD_LOGIC_VECTOR(4 downto 0);
         wd3:           in  STD_LOGIC_VECTOR(31 downto 0);
         rd1, rd2:      out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component adder
    port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
         y:    out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component sl2
    port(a: in  STD_LOGIC_VECTOR(31 downto 0);
         y: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component signext
    port(a: in  STD_LOGIC_VECTOR(15 downto 0);
         y: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component flopr generic(width: integer);
    port(clk, reset: in  STD_LOGIC;
         d:          in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:          out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component mux2 generic(width: integer);
    port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s:      in  STD_LOGIC;
         y:      out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component upimm --LUI
  port(a: in  STD_LOGIC_VECTOR(15 downto 0);
       y: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component mux3 generic(width: integer);  --LUI
  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 on next page)

(continued from previous page)

Verilog

```
  // hardware to support LH
  mux2 #(16)lhmux1(readdata[15:0],
                   readdata[31:16],
                   aluresult[1], half);  // LH
  signext     lhse(half, signhalf);        // LH
  mux2 #(32)  lhmux2(readdata, signhalf, lh,
                     memdata); // LH

  mux2 #(32)  resmux(aluresult, memdata, memtoreg,
                    result); // LH
  signext     se(instr[15:0], signimm);
  upimm       ui(instr[15:0], upperimm);  // LUI

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

VHDL

```
  signal writereg: STD_LOGIC_VECTOR(4 downto 0);
  signal pcjump, pcnext, pcnextbr,
         pcplus4, pcbranch:  STD_LOGIC_VECTOR(31 downto 0);
  signal signimm, signimmsh: STD_LOGIC_VECTOR(31 downto 0);
  signal upperimm: STD_LOGIC_VECTOR(31 downto 0); --LUI
  signal srca, srcb, result: STD_LOGIC_VECTOR(31 downto 0);
  signal writeresult: STD_LOGIC_VECTOR(31 downto 0); --JAL
  signal half: STD_LOGIC_VECTOR(15 downto 0); --LH
  signal signhalf, memdata: STD_LOGIC_VECTOR(31 downto 0); --LH
begin
  -- next PC logic
  pcjump <= pcplus4(31 downto 28) & instr(25 downto 0) & "00";
  pcreg: flopr generic map(32) port map(clk, reset, pcnext, pc);
  pcadd1: adder port map(pc, X"00000004", pcplus4);
  immsh: sl2 port map(signimm, signimmsh);
  pcadd2: adder port map(pcplus4, signimmsh, pcbranch);
  pcbrmux: mux2 generic map(32) port map(pcplus4, pcbranch,
                                         pcsrc, pcnextbr);
  pcmux: mux2 generic map(32) port map(pcnextbr, pcjump, jump, pcnext);

  -- register file logic
  rf: regfile port map(clk, regwrite, instr(25 downto 21),
                       instr(20 downto 16), writereg,
                       writeresult,  --JAL
                       srca, writedata);

  ramux: mux2 generic map(32)  port map(result, pcplus4, jal,
                        writeresult);  -- JAL

  wrmux: mux3 generic map(5) port map(instr(20 downto 16),
                                     instr(15 downto 11), "11111", --JAL
                                      regdst, writereg);

  -- hardware to support LH
  lhmux1: mux2 generic map(16) port map(readdata(15 downto 0),
                                        readdata(31 downto 16),
                                        aluresult(1), half); --LH
  lhse: signext port map(half, signhalf); --LH
  lhmux2: mux2 generic map(32) port map(readdata, signhalf,
                                        lh, memdata);  --LH

  resmux: mux2 generic map(32) port map(aluresult, memdata,  --LH
                                        memtoreg, result);
  se: signext port map(instr(15 downto 0), signimm);
  ue: upimm port map(instr(15 downto 0), upperimm); --LUI

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

Additional Building Blocks

Verilog

```
// upimm module needed for LUI
module upimm(input  [15:0] a,
             output [31:0] y);

  assign y = {a, 16'b0};
endmodule

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

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

VHDL

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

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

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

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

Modified ALU

Verilog

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

  wire [31:0] S, Bout;

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

  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);  // SLL
    endcase

  assign Zero = (Y == 32'b0);
  assign ltez = Zero | S[31];  // BLEZ

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); --SLL
       shamt: in     STD_LOGIC_VECTOR(4 downto 0); --SLL
       Y:     inout  STD_LOGIC_VECTOR(31 downto 0);
       Zero:  inout  STD_LOGIC;  --BLEZ
       ltez:  out    STD_LOGIC); --BLEZ
end;

architecture synth of alu is
  signal S, Bout:      STD_LOGIC_VECTOR(31 downto 0);
begin
  Bout <= (not B) when (F(3) = '1') else B;
  S <= A + Bout + F(3);   --SLL

  -- alu function
  process (F(2 downto 0), A, Bout, S) begin
    case F(2 downto 0) is
      when "000" => Y <= A and Bout;
      when "001" => Y <= A or Bout;
      when "010" => Y <= S;
      when "011" => Y <=
        ("0000000000000000000000000000000" & S(31));
      when "100" => Y <=
      to_stdlogicvector(to_bitvector(Bout) sll conv_integer(shamt));-- SLL
      when others => Y <= X"00000000";
    end case;
  end process;

  Zero <= '1' when (Y = X"00000000") else '0';
  ltez <= Zero or S(31);  -- BLEZ

end;
```

We modify the test code to include the extended instructions.

```
# mipstest.asm
# Sarah_Harris@hmc.edu 20 February 2007
#
# Test the MIPS processor.
#  add, sub, and, or, slt, addi, lw, sw, beq, j
# If successful, it should write the value 4135 to address 92

#       Assembly                  Description           Address Machine
main:   addi $2, $0, 5          # initialize $2 = 5     0       20020005
        lui  $2, 0xEFE          # $2 = 0x0EFE0000       4       3C020efe
        sll  $2, $2, 4          # $2 = 0xEFE00000       8       00021100
        jal  forward                                    c       0c000006
        addi $3, $0, 14         # not executed          10      2263000e
back:   blez $2, here           # should be taken       14      18400002
forward:addi $3, $ra, -4        # $3 <= $ra - 4 = 12    18      23e3fffc
        j    back                                       1c      08000005
here:   addi $7, $3, -9         # initialize $7 = 3     20      2067fff7
        addi $6, $0, 5          # initialize $6 = 5     24      20060005
        or   $4, $7, $6         # $4 <= 3 or 5 = 7      28      00e62025
        and  $5, $3, $4         # $5 <= 12 and 3 = 4    2c      00642824
        add  $5, $5, $4         # $5 = 4 + 7 = 11       30      00a42820
        beq  $5, $7, end        # shouldn't be taken    34      10a7000c
        slti $4, $3, 7          # $4 = 12 < 7 = 0       38      28640007
        beq  $4, $0, around     # should be taken       3c      10800001
        addi $5, $0, 0          # shouldn't happen      40      20050000
around: slti $4, $7, 5          # $4 = 3 < 5 = 1        44      28e40005
        add  $7, $4, $5         # $7 = 1 + 11 = 12      48      00853820
        sub  $7, $7, $6         # $7 = 12 - 5 = 7       4c      00e63822
        sw   $7, 68($3)         # [68+12] = [80] = 7    50      ac670044
        sw   $2, 88($0)         # [88] = 0xEFE00000     54      ac020058
        lw   $2, 80($0)         # $2 = [80] = 7         58      8c020050
        lh   $3, 90($0)         # $2 = 0xFFFFEFE0       5c      8403005a
        j    end                # should be taken       60      0800001a
        addi $2, $0, 1          # shouldn't happen      64      20020001
end:    sub  $8, $2, $3         # $8 = 7-(-4128) = 4135 68      00434022
        sw   $8, 92($0)         # [92] = 4135           6c      ac08005c
```

FIGURE 7.6 Assembly and machine code for MIPS test program

Modified Testbench

Verilog

```
module testbench();

  reg         clk;
  reg         reset;

  wire [31:0] aluout, writedata, readdata;
  wire memwrite;

  // instantiate device to be tested
  topsingle dut(clk, reset, readdata, writedata,
                aluout, 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 & aluout == 92) begin
        if(writedata == 4135)
          $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 top
    port(clk, reset:           in    STD_LOGIC;
         readdata:             inout STD_LOGIC_VECTOR(31 downto 0);
         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: top port map(clk, reset, readdata, 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' and
        (conv_integer(dataadr) = 92)) then
      if (conv_integer(writedata) = 4135) then
        report "Simulation succeeded"
          severity failure;
      else
        report "Simulation failed"
          severity failure;
      end if;
    end if;
  end process;
end;
```

7.11

(a) `srlv`

First, we show the modifications to the ALU.

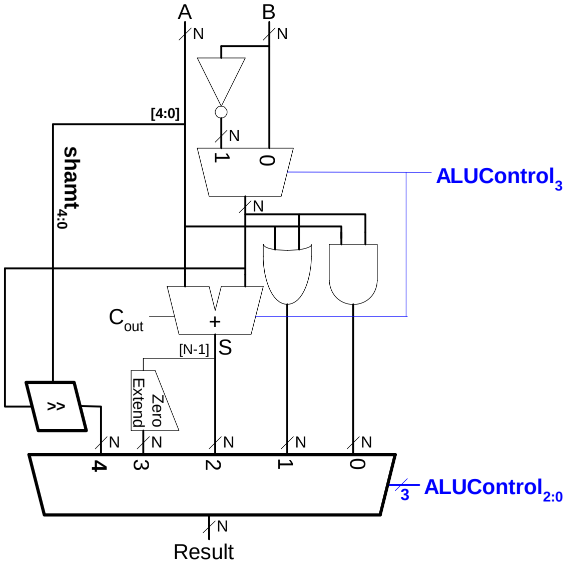


FIGURE 7.7 Modified ALU to support `srlv`

Next, we show the modifications to the ALU decoder.

$ALUControl_{3:0}$	Function
0**0**00	A AND B
0**0**01	A OR B
0**0**10	A + B
0**0**11	not used
1**0**00	A AND $\overline{B}$
1**0**01	A OR $\overline{B}$
1**0**10	A - B
1**0**11	SLT
0100	**SRLV**

FIGURE 7.8 Modified ALU operations to support `srlv`

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	**000110 (`srlv`)**	**0100 (shift right logical variable)**

TABLE 7.13 ALU decoder truth table

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.

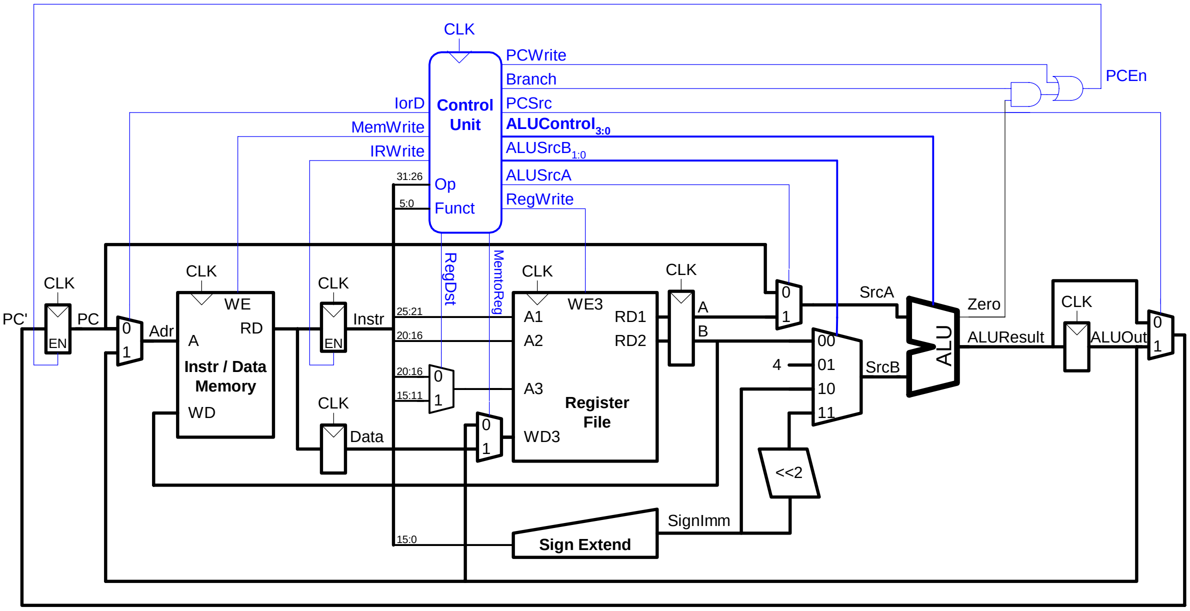


FIGURE 7.9 Modified multicycle MIPS datapath to support `sll`

7.11 (b) `ori`

We add a zero extension unit to the datapath, extend the *ALUSrcB* signal from 2 bits to 3 bits, and extend the SrcB multiplexer from 4 inputs to 5 inputs. We also modify the ALU decoder and main control FSM.

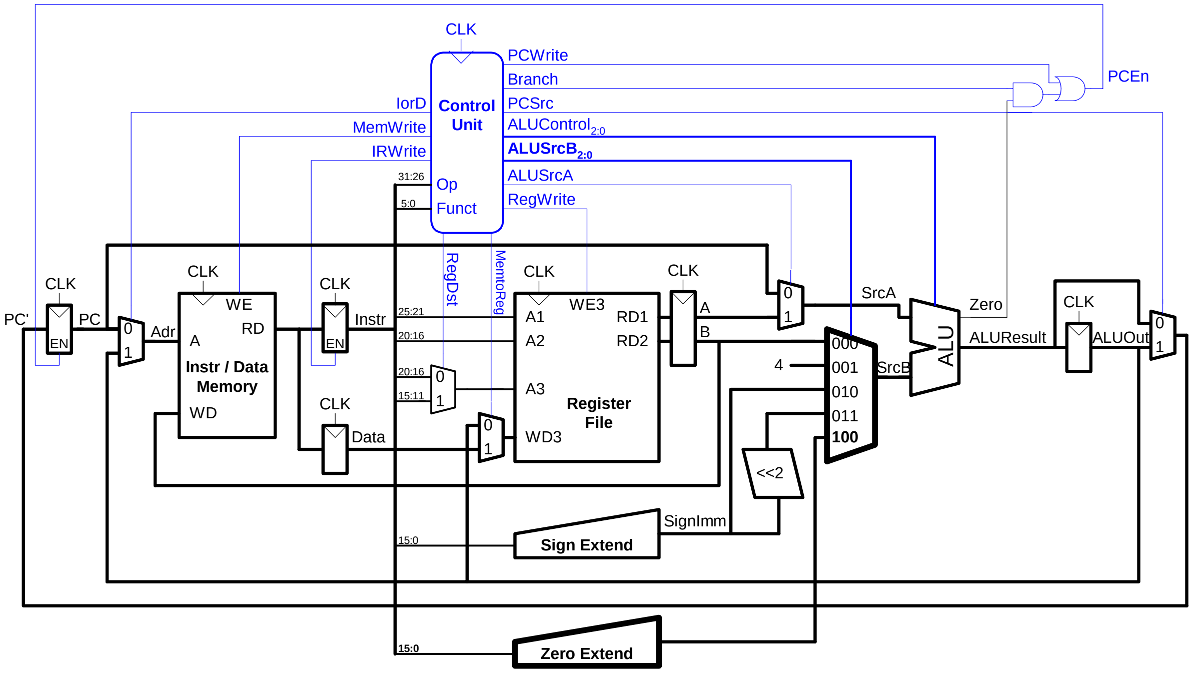


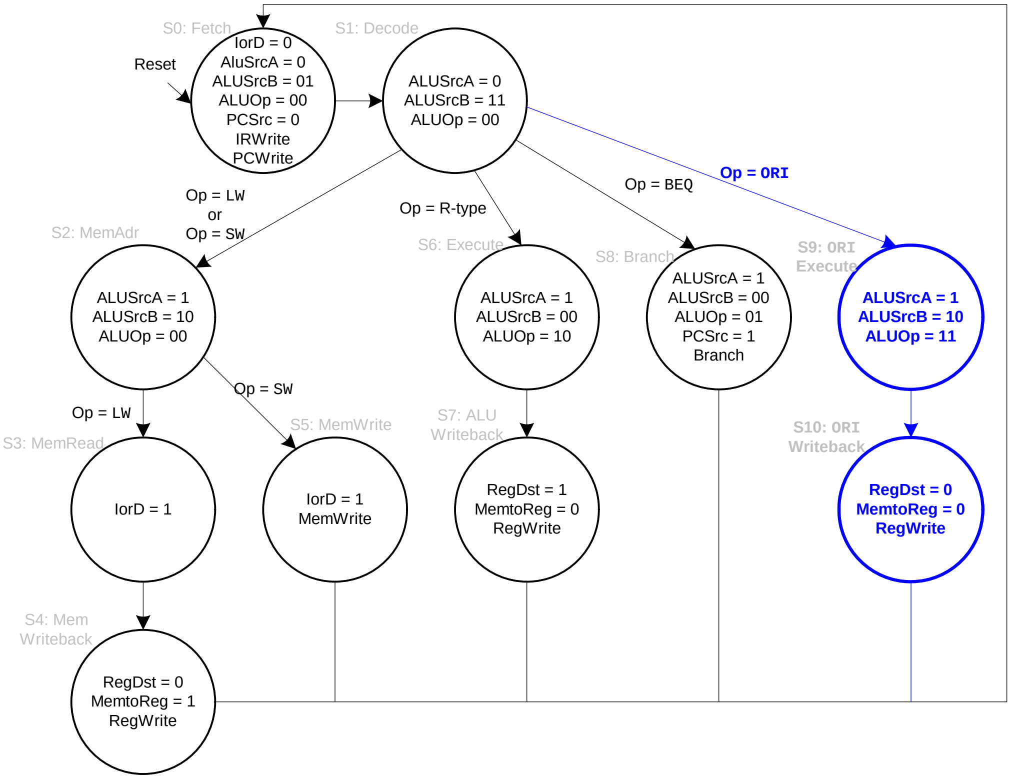
FIGURE 7.10 Modified datapath for `ori`

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 7.14 ALU decoder truth table

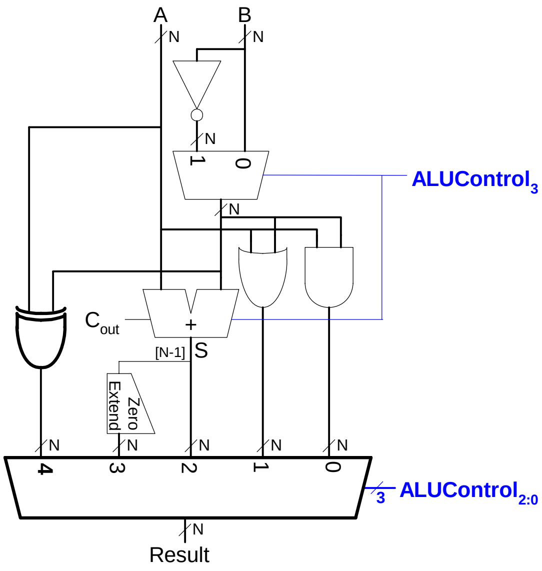
ALUOp	Funct	ALUControl
10	100101 (`or`)	001 (or)
10	101010 (`slt`)	111 (set less than)

TABLE 7.14 ALU decoder truth table



7.11(c) `xori`

First, we modify the ALU and the ALU decoder.



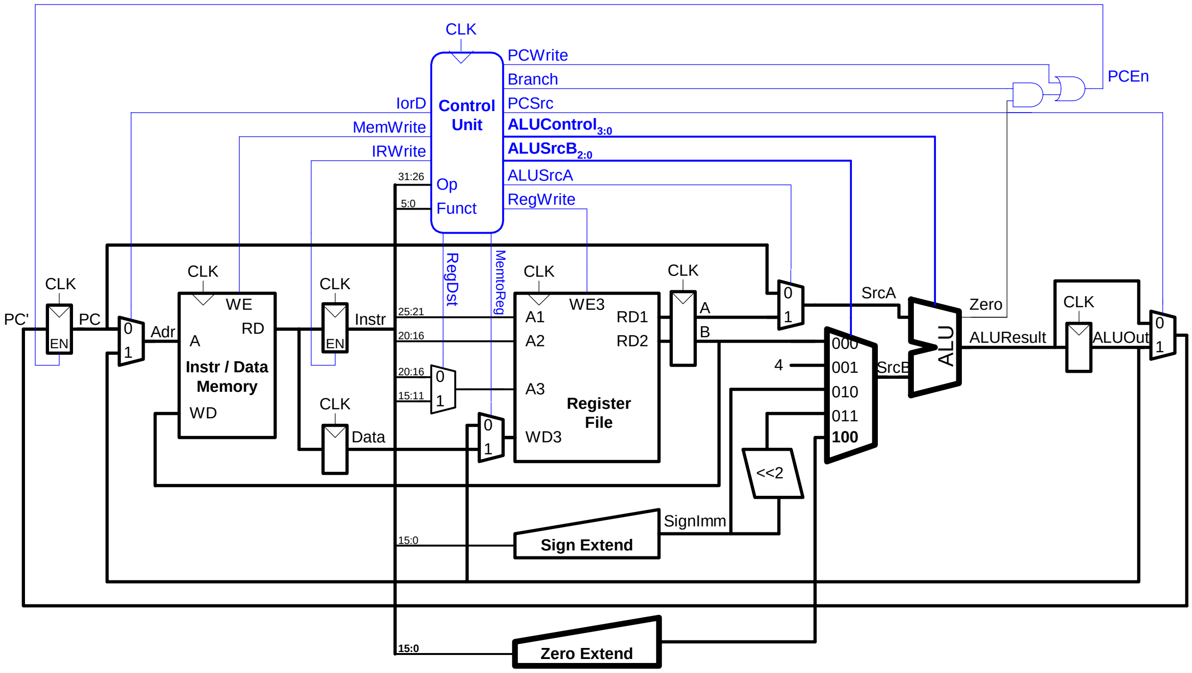
l

ALUControl$_{3:0}$	Function
0**0**00	A AND B
0**0**01	A OR B
0**0**10	A + B
0**0**11	not used
1**0**00	A AND $\overline{B}$
1**0**01	A OR $\overline{B}$
1**0**10	A - B
1**0**11	SLT
0100	**A XOR B**

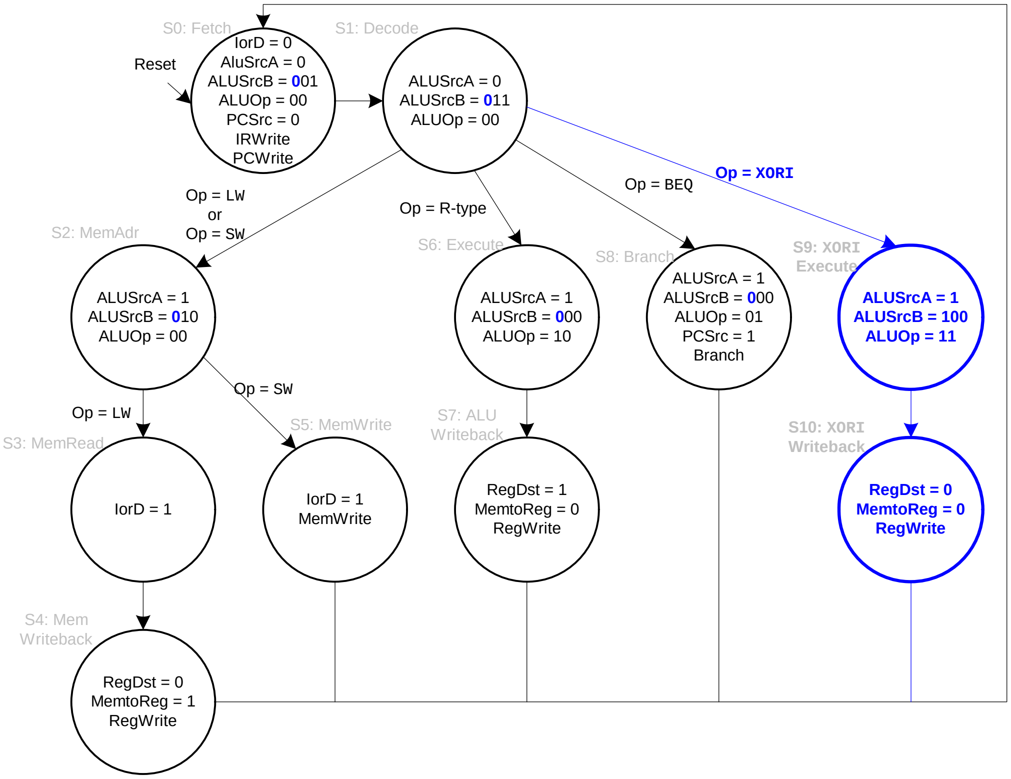
ALUOp	Funct	ALUControl
00	X	0010 (add)
01	X	1010 (subtract)
11	**X**	**0100 (xor)**
10	100000 (`add`)	0010 (add)
10	100010 (`sub`)	1010 (subtract)
10	100100 (`and`)	0000 (and)
10	100101 (`or`)	0001 (or)
10	101010 (`slt`)	1011 (set less than)

TABLE 7.15 ALU decoder truth table for `xori`

Next, we modify the datapath. We change the buswidth of the *ALUControl* signal from 3 bits to 4 bits and the *ALUSrcB* signal from 2 bits to 3 bits. We also extend the SrcB mux and add a zero-extension unit.



And finally, we modify the main control FSM.



7.11 (d) `jr`

First, we extend the ALU Decoder for `jr`.

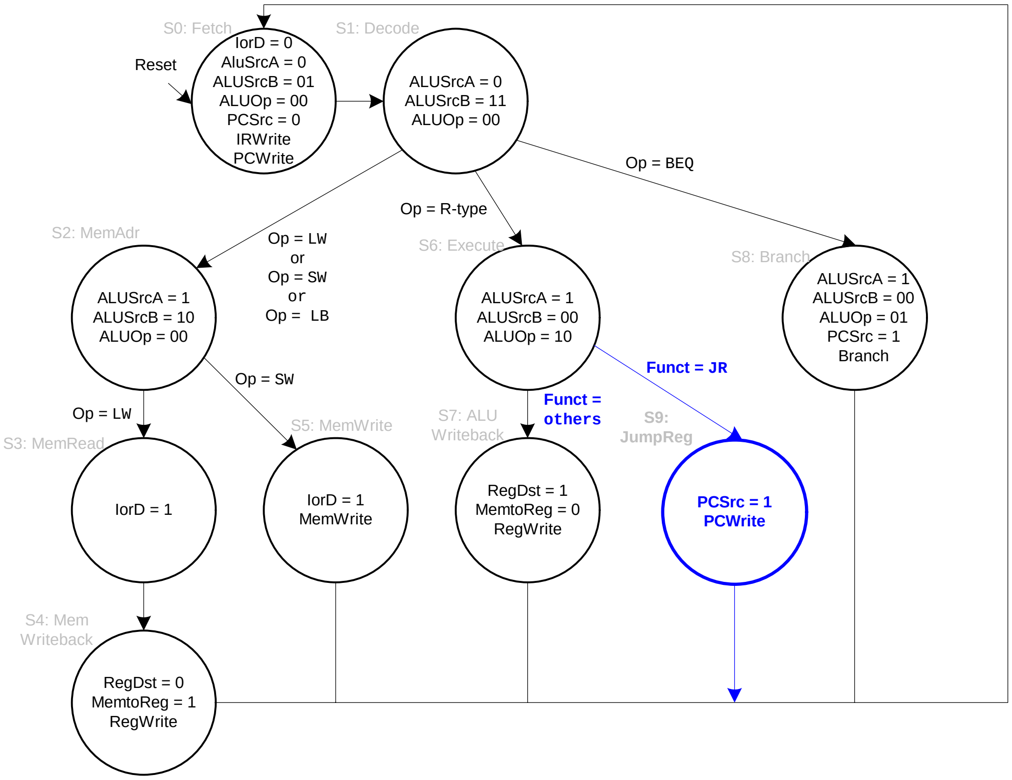
ALUOp	Funct	ALUControl
00	X	010 (add)
X1	X	110 (subtract)
1X	100000 (add)	010 (add)
1X	100010 (sub)	110 (subtract)
1X	100100 (and)	000 (and)
1X	100101 (or)	001 (or)

TABLE 7.16 ALU decoder truth table with `jr`

ALUOp	Funct	ALUControl
1X	101010 (`slt`)	111 (set less than)
1X	001000 (`jr`)	010 (add)

TABLE 7.16 ALU decoder truth table with `jr`

Next, we modify the main controller. The datapath requires no modification.



7.11 (e) bne

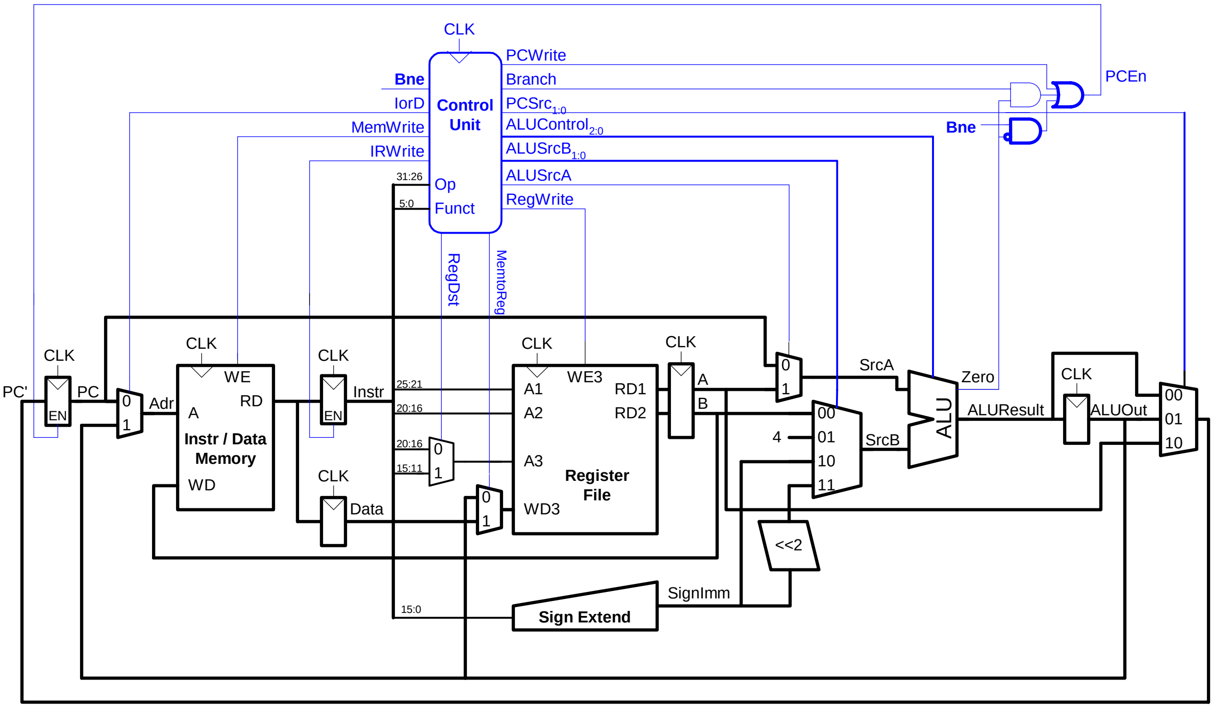


FIGURE 7.11 Modified datapath for bne

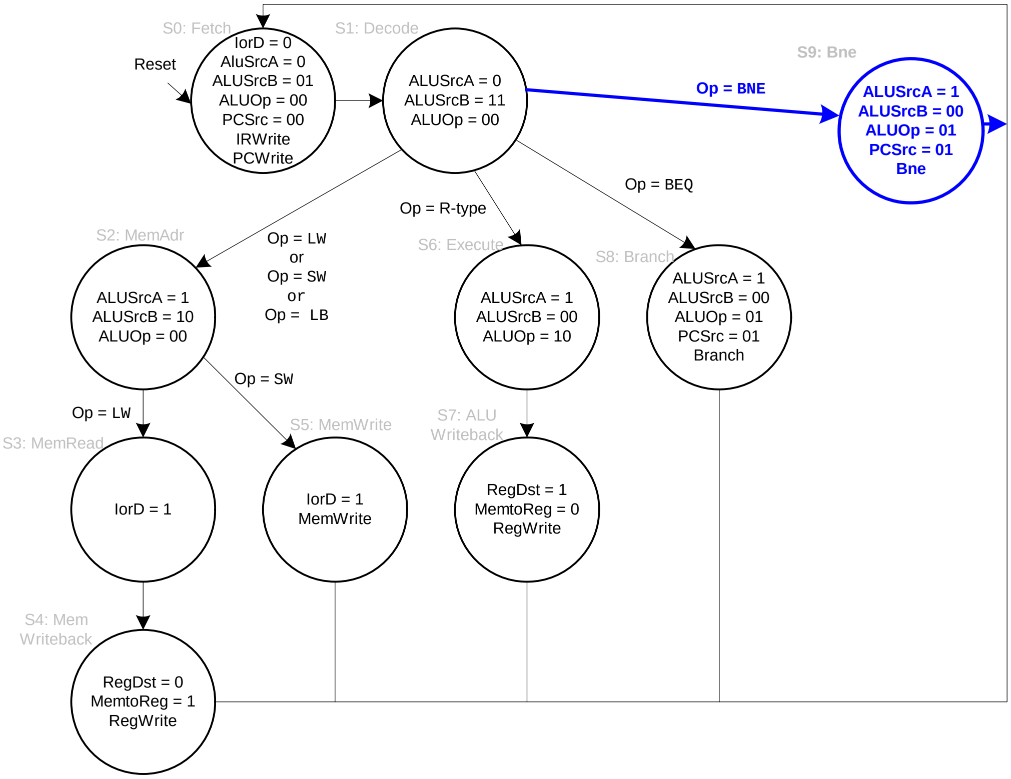
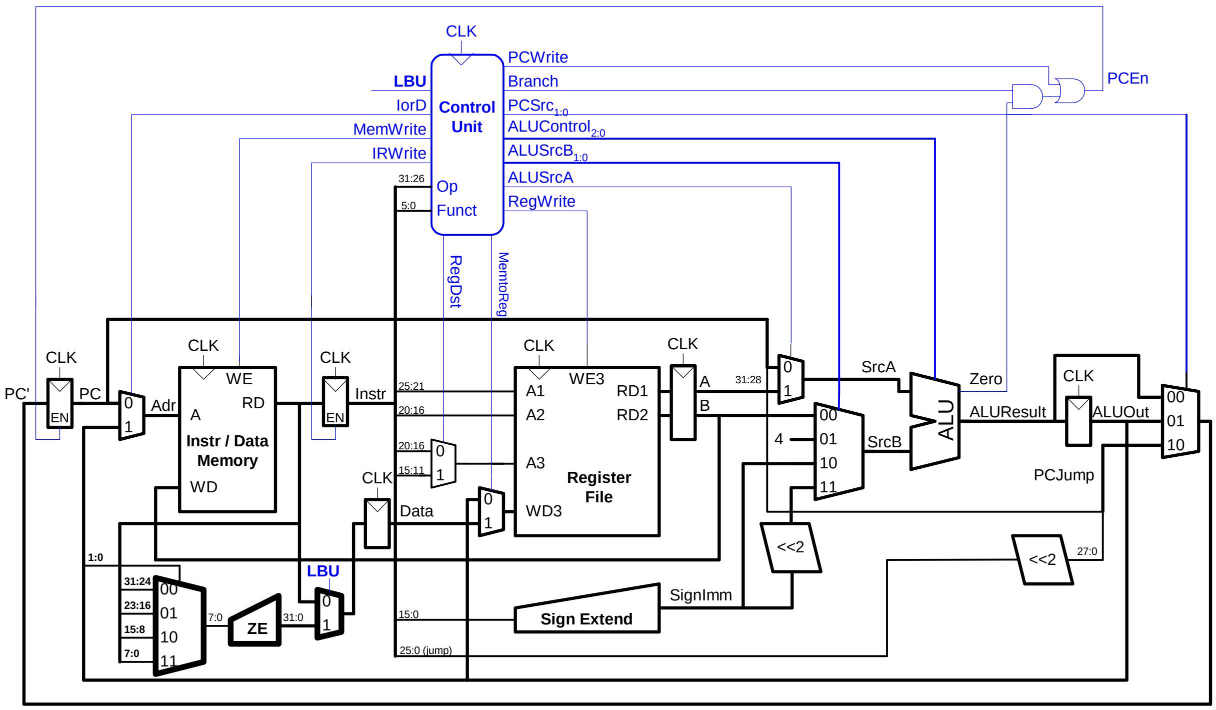
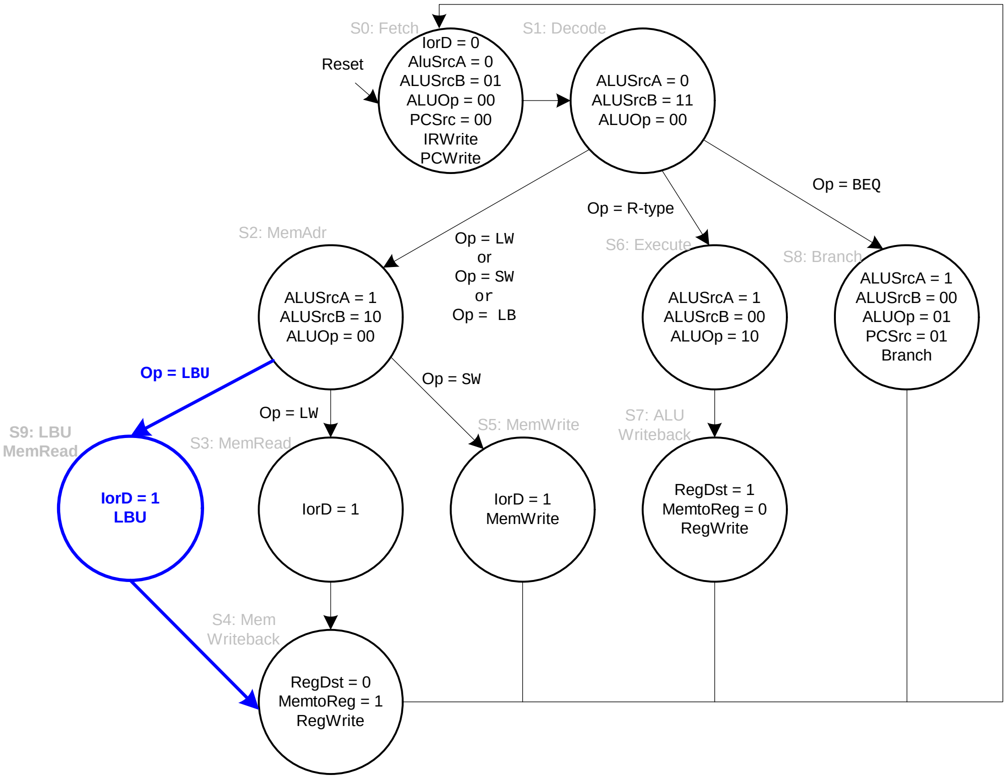


FIGURE 7.12 Modified FSM for bne

7.11 (f) `lbu`

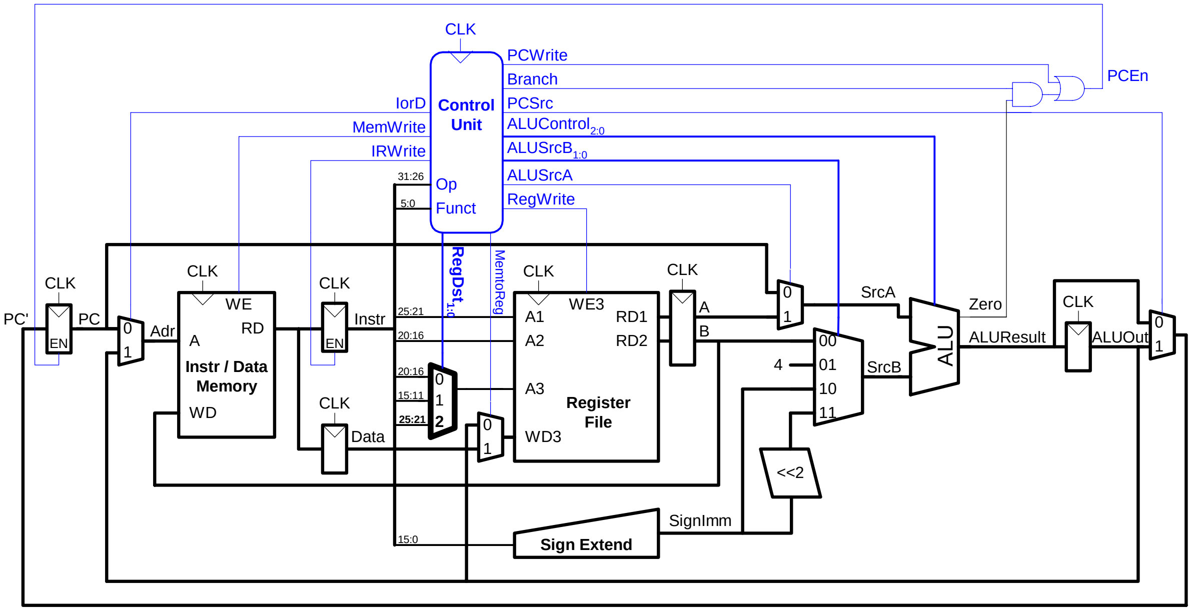


* The ZE unit is a zero extension unit.

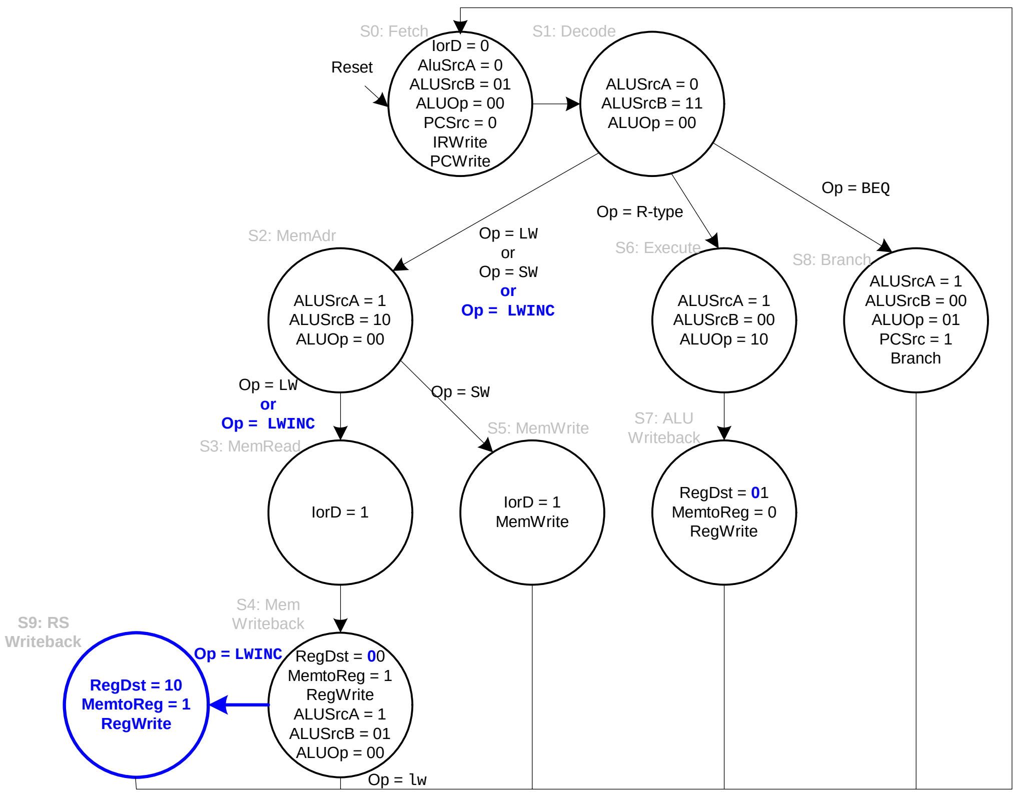


7.12

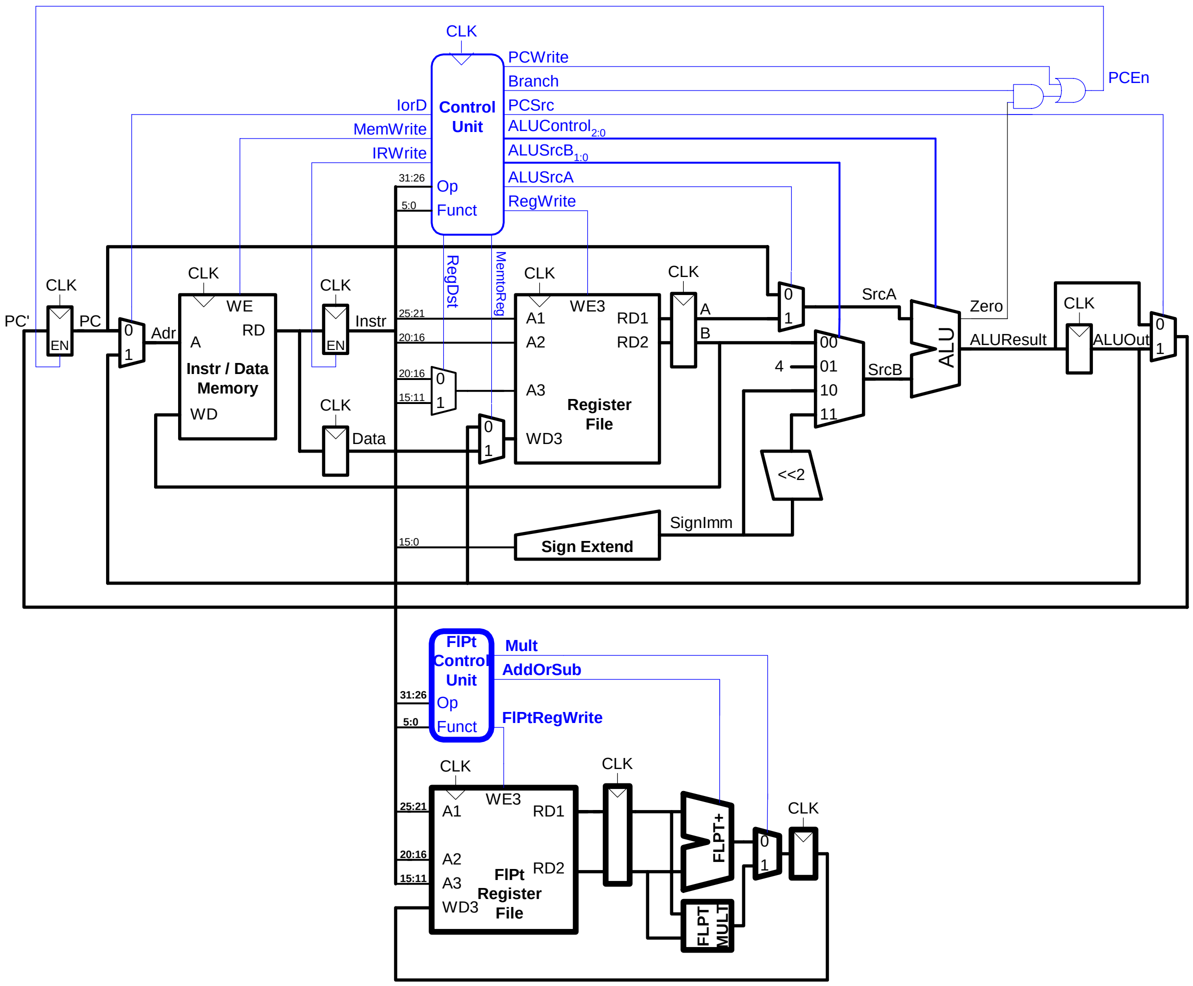
Yes, it is possible to add this instruction without modifying the register file. First we show the modifications to the datapath. The only modification is adding the `rs` field of the instruction ($Instruction_{25:21}$) to the input of the write address mux of the register file. *RegDst* must be expanded to two bits.

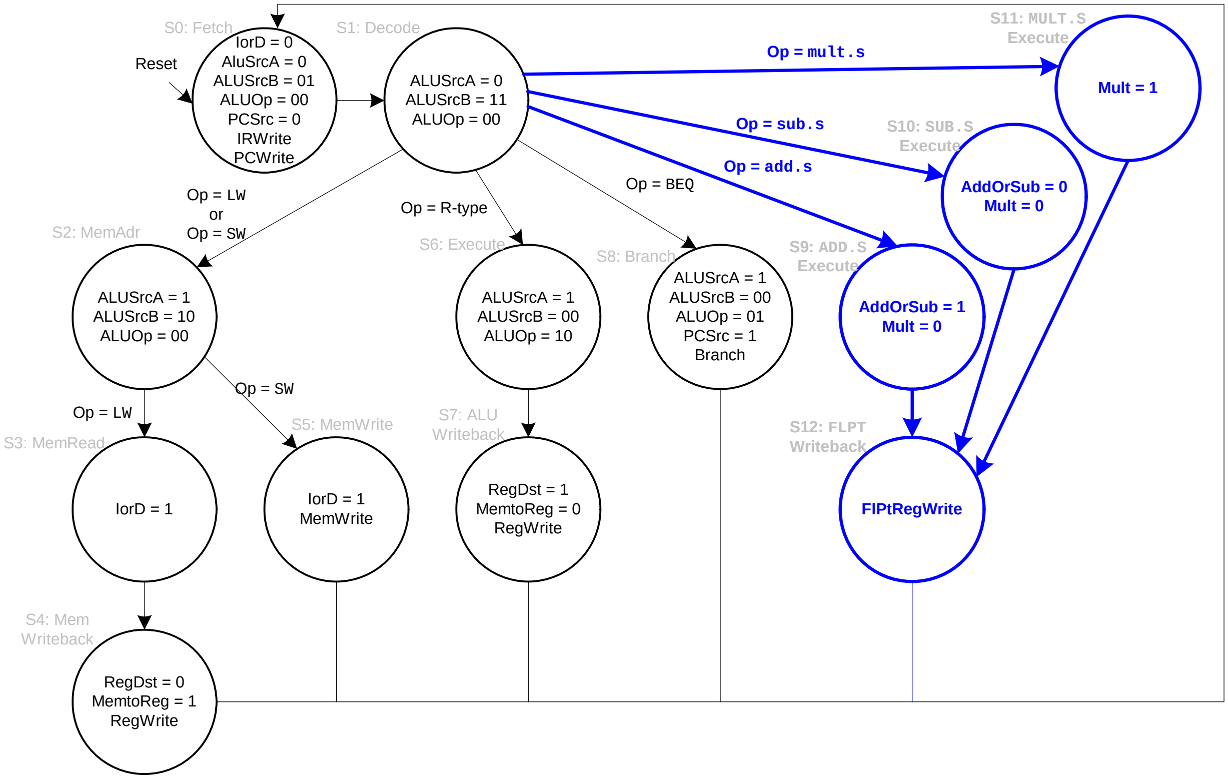


The finite state machine requires another state to write the `rs` register. If execution time is critical, another adder could be placed just after the A/B register to add 4 to A. Then in State 3, as memory is read, the register file could be written back with the incremented `rs`. In that case, `lwinc` would require the same number of cycles as `lw`. The penalty, however, would be chip area, and thus power and cost.

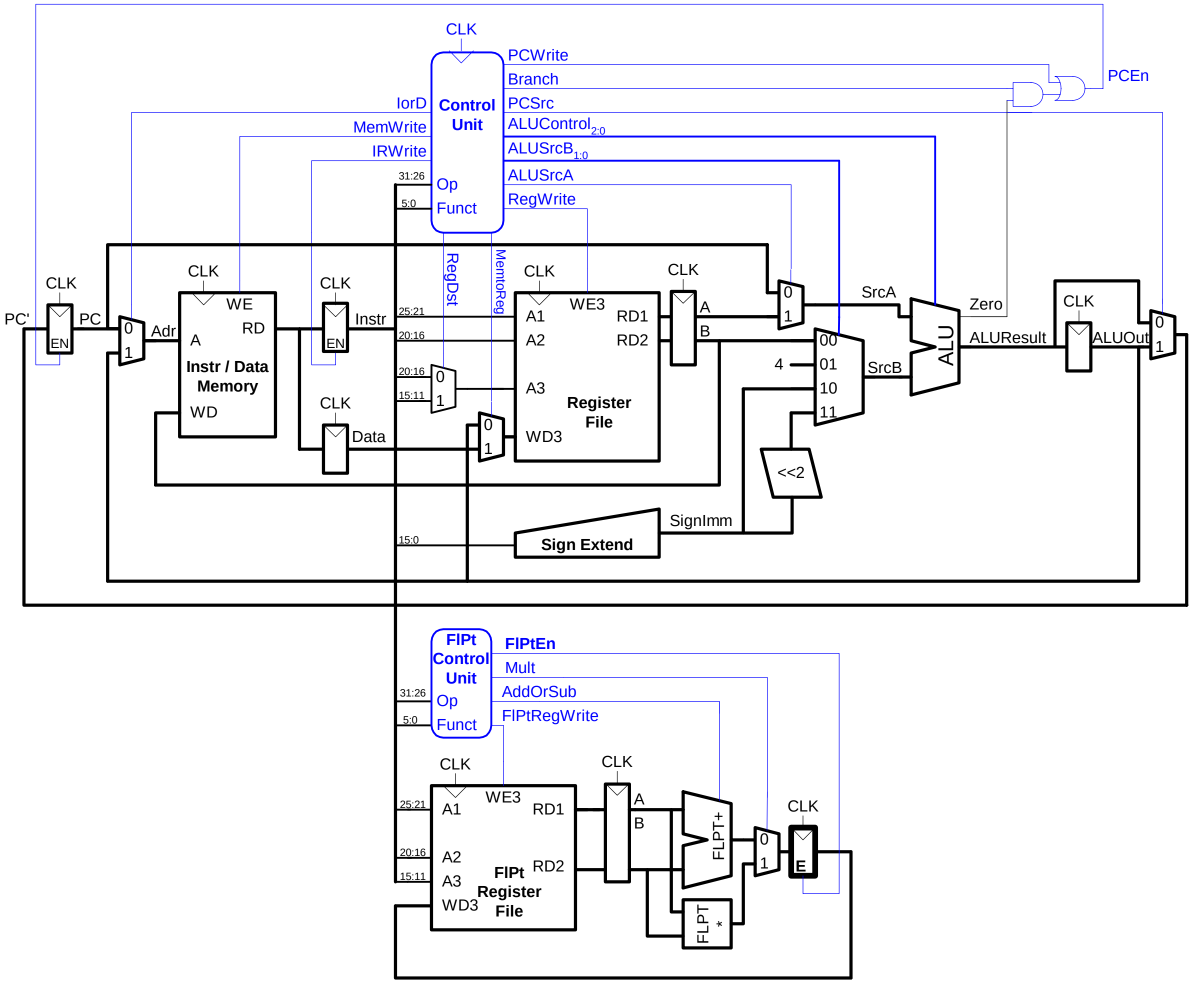

S0: Fetch
Reset
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite
S1: Decode
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
Op = BEQ
Op = R-type
Op = LW
or
Op = SW
or
Op = LWINC
S2: MemAdr
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
S6: Execute
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
S8: Branch
ALUSrcA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

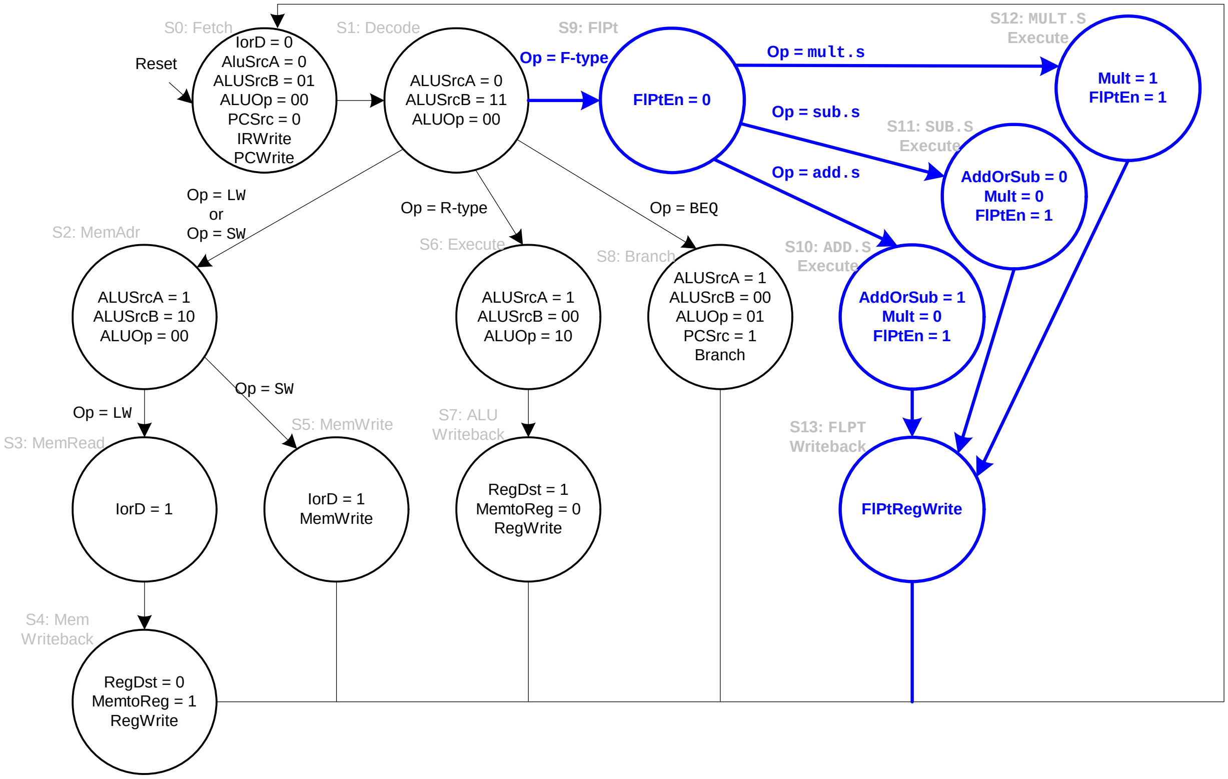
7.13


CLK
PCWrite
Branch
PCSrc
ALUControl$_{2:0}$
ALUSrcB$_{1:0}$
ALUSrcA
RegWrite
PCEn
IorD
MemWrite
IRWrite
Control Unit
Op
Funct
31:26
5:0
RegDst
MemtoReg
PC'
PC
EN
Adr
A
RD
WE
Instr / Data Memory
WD
Instr
Data
25:21
20:16
15:11
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
A
B
SrcA
SrcB
4
00
01
10
11
<<2
ALU
Zero
ALUResult
ALUOut
SignImm
Sign Extend
15:0
0
1
FlPt Control Unit
Mult
AddOrSub
FlPtRegWrite
FlPt Register File
FLPT+
FLPT MULT


S0: Fetch
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite
Reset
S1: Decode
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
S11: MULT.S
Execute
Op = mult.s
Mult = 1
Op = sub.s
S10: SUB.S
Execute
AddOrSub = 0
Mult = 0
Op = add.s
Op = BEQ
Op = LW
or
Op = SW
Op = R-type
S2: MemAdr
S6: Execute
S8: Branch
S9: ADD.S
Execute
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCSrc = 1
Branch
AddOrSub = 1
Mult = 0
Op = SW
Op = LW
S5: MemWrite
S7: ALU
Writeback
S12: FLPT
Writeback
S3: MemRead
IorD = 1
IorD = 1
MemWrite
RegDst = 1
MemtoReg = 0
RegWrite
FlPtRegWrite
S4: Mem
Writeback
RegDst = 0
MemtoReg = 1
RegWrite

7.14 We add an enable signal, *FlPtEn*, to the result register.





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.

7.16

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

7.17

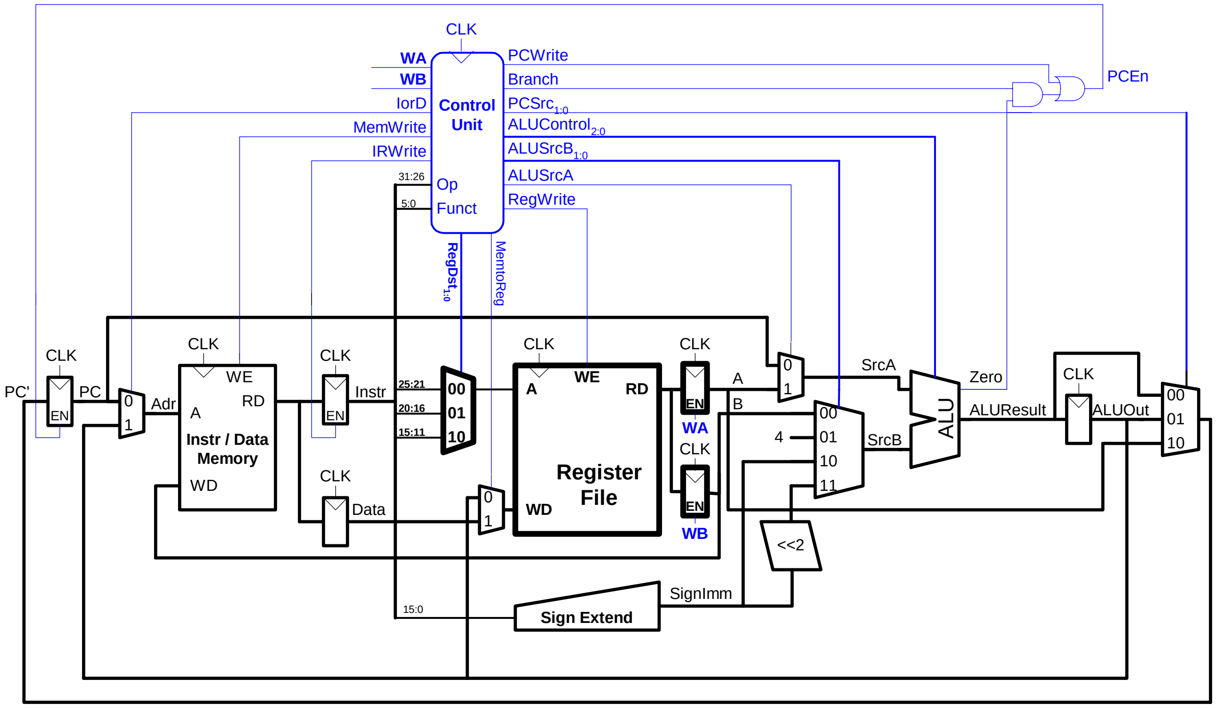
No, Alyssa should not switch to the slower but lower power register file for her multicycle processor design.

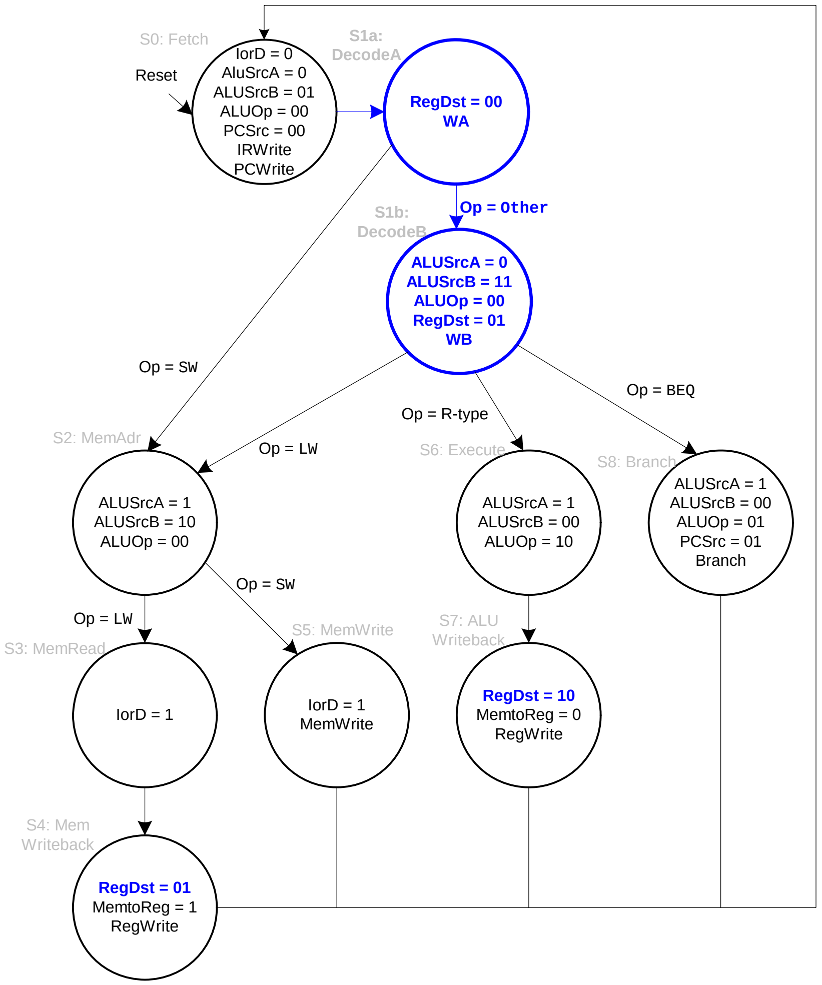
Doubling the delay of the register file puts it on the critical path. (2 x 150 ps = 300 ps for a read). The critical path is:

$$T_c = t_{pcq} + \max(t_{\text{RFread}}, t_{\text{mux}} + t_{\text{mem}}) + t_{\text{setup}}$$

Because t_{RFread} (300 ps) is larger than $t_{\text{mux}} + t_{\text{mem}}$ (25 + 250 = 275 ps), the cycle time would increase by 25 ps with the slower but lower power register file.

7.18





7.19

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

7.20

4 + (3 + 4 + 3) × 5 + 3 = **57 clock cycles**

The number of instructions executed is 1 + (3 × 5) + 1 = 17. Thus, the CPI = 57 clock cycles / 17 instructions = **3.35 CPI**

7.21

$(4 \times 3) + (4 + 3 + 4 + 4 + 3) \times 10 + (4 + 3) =$ **199 clock cycles**

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

7.22

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 [1:0]  alusrcb, pcsrc;
  wire [2:0]  alucontrol;
  wire [5:0]  op, funct;

  controller c(clk, reset, op, funct, zero,
               pcen, memwrite, irwrite, regwrite,
               alusrca, iord, memtoreg, regdst,
               alusrcb, pcsrc, alucontrol);
  datapath dp(clk, reset,
              pcen, irwrite, regwrite,
              alusrca, iord, memtoreg, regdst,
              alusrcb, pcsrc, alucontrol,
              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, pcsrc:    out STD_LOGIC_VECTOR(1 downto 0);
         alucontrol:        out STD_LOGIC_VECTOR(2 downto 0));
  end component;
  component datapath
  port(clk, reset:        in  STD_LOGIC;
       pcen, irwrite:     in  STD_LOGIC;
       regwrite, alusrca: in  STD_LOGIC;
       iord, memtoreg:    in  STD_LOGIC;
       regdst:            in  STD_LOGIC;
       alusrcb, pcsrc:    in  STD_LOGIC_VECTOR(1 downto 0);
       alucontrol:        in  STD_LOGIC_VECTOR(2 downto 0);
       readdata:          in  STD_LOGIC_VECTOR(31 downto 0);
       op, funct:         out STD_LOGIC_VECTOR(5 downto 0);
       zero:              out STD_LOGIC;
       adr:               out STD_LOGIC_VECTOR(31 downto 0);
       writedata:         inout STD_LOGIC_VECTOR(31 downto 0));
  end component;
  signal zero, pcen, irwrite, regwrite, alusrca, iord, memtoreg,
         regdst: STD_LOGIC;
  signal alusrcb, pcsrc: STD_LOGIC_VECTOR(1 downto 0);
  signal alucontrol: STD_LOGIC_VECTOR(2 downto 0);
  signal op, funct: STD_LOGIC_VECTOR(5 downto 0);
begin
  c: controller port map(clk, reset, op, funct, zero,
                         pcen, memwrite, irwrite, regwrite,
                         alusrca, iord, memtoreg, regdst,
                         alusrcb, pcsrc, alucontrol);
  dp: datapath port map(clk, reset,
                        pcen, irwrite, regwrite,
                        alusrca, iord, memtoreg, regdst,
                        alusrcb, pcsrc, alucontrol,
                        readdata, op, funct, zero,
                        adr, writedata);
end;
```

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 [1:0] alusrcb, pcsrc,
                  output [2:0] alucontrol);

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

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

  assign pcen = pcwrite | (branch & zero);

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, pcsrc:    out STD_LOGIC_VECTOR(1 downto 0);
       alucontrol:        out STD_LOGIC_VECTOR(2 downto 0));
end;


architecture struct of controller is
  component maindec
    port(clk, reset:        in  STD_LOGIC;
         op:                in  STD_LOGIC_VECTOR(5 downto 0);
         pcwrite, memwrite: out STD_LOGIC;
         irwrite, regwrite: out STD_LOGIC;
         alusrca, branch:   out STD_LOGIC;
         iord, memtoreg:    out STD_LOGIC;
         regdst:            out STD_LOGIC;
         alusrcb, pcsrc:    out STD_LOGIC_VECTOR(1 downto 0);
         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;
  signal aluop: STD_LOGIC_VECTOR(1 downto 0);
  signal branch, pcwrite: STD_LOGIC;
begin
  md: maindec port map(clk, reset, op,
                       pcwrite, memwrite, irwrite, regwrite,
                       alusrca, branch, iord, memtoreg, regdst,
                       alusrcb, pcsrc, aluop);
  ad: aludec port map(funct, aluop, alucontrol);

  pcen <= pcwrite or (branch and zero);

end;
```

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 [1:0] alusrcb, pcsrc,
               output [1:0] aluop);

  parameter   FETCH   = 4'b0000;   // State 0
  parameter   DECODE  = 4'b0001;   // State 1
  parameter   MEMADR  = 4'b0010;// State 2
  parameter   MEMRD   = 4'b0011;// State 3
  parameter   MEMWB   = 4'b0100;// State 4
  parameter   MEMWR   = 4'b0101;// State 5
  parameter   RTYPEEX = 4'b0110;// State 6
  parameter   RTYPEWB = 4'b0111;// State 7
  parameter   BEQEX   = 4'b1000;// State 8
  parameter   ADDIEX  = 4'b1001;// State 9
  parameter   ADDIWB  = 4'b1010;// state 10
  parameter   JEX     = 4'b1011;// State 11

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

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

  // state register
  always @(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, pcsrc:     out STD_LOGIC_VECTOR(1 downto 0);
 aluop:              out STD_LOGIC_VECTOR(1 downto 0));
end;

architecture behave of maindec is
  type statetype is (FETCH, DECODE, MEMADR, MEMRD, MEMWB, MEMWR,
                     RTYPEEX, RTYPEWB, BEQEX, ADDIEX, ADDIWB, JEX);
  signal state, nextstate: statetype;
  signal controls: STD_LOGIC_VECTOR(14 downto 0);
begin
  --state register
  process(clk, reset) begin
    if reset = '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 "000000" =>  nextstate <= RTYPEEX;
            when "000100" =>  nextstate <= BEQEX;
            when "001000" =>  nextstate <= ADDIEX;
            when "000010" =>  nextstate <= JEX;
            when others    => nextstate <= FETCH; -- should never happen
          end case;
      when MEMADR =>
          case op is
            when "100011" =>  nextstate <= MEMRD;
            when "101011" =>  nextstate <= MEMWR;
            when others    => nextstate <= FETCH; -- should never happen
          end case;
      when hen 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 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;
```

MIPS Multicycle 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 Multicycle Datapath

Verilog

```
module datapath(input         clk, reset,
               input        pcen, irwrite, regwrite,
                input         alusrca, iord,
                              memtoreg, regdst,
                input  [1:0]  alusrcb, pcsrc,
                input  [2:0]  alucontrol,
                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;   // sign-extended immediate
  wire [31:0] signimmsh; // sign-extended immediate
                         // shifted left by 2
  wire [31:0] wd3, rd1, rd2;

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

  // datapath
  flopenr #(32) pcreg(clk, reset, pcen, pcnext, pc);
  mux2    #(32) adrmux(pc, aluout, iord, adr);
  flopenr #(32) instrreg(clk, reset, irwrite,
                         readdata, instr);
  flopr   #(32) datareg(clk, reset, readdata, data);
 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);
  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);
  mux4    #(32) srcbmux(writedata, 32'b100,
                        signimm, signimmsh,
                        alusrcb, srcb);
  alu           alu(srca, srcb, alucontrol,
                    aluresult, zero);
 flopr   #(32) alureg(clk, reset, aluresult, aluout);
  mux3    #(32) pcmux(aluresult, aluout,
                    {pc[31:28], instr[25:0], 2'b00},
                      pcsrc, pcnext);

endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all; use IEEE.STD_LOGIC_ARITH.all;
entity datapath is  -- MIPS datapath
  port(clk, reset:        in  STD_LOGIC;
       pcen, irwrite:     in  STD_LOGIC;
       regwrite, alusrca: in  STD_LOGIC;
       iord, memtoreg:    in  STD_LOGIC;
       regdst:            in  STD_LOGIC;
       alusrcb, pcsrc:    in  STD_LOGIC_VECTOR(1 downto 0);
       alucontrol:        in  STD_LOGIC_VECTOR(2 downto 0);
       readdata:          in  STD_LOGIC_VECTOR(31 downto 0);
       op, funct:         out STD_LOGIC_VECTOR(5 downto 0);
       zero:              out STD_LOGIC;
       adr:               out STD_LOGIC_VECTOR(31 downto 0);
       writedata:         inout STD_LOGIC_VECTOR(31 downto 0));
end;

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

(continued from previous page)

Verilog

VHDL

```
begin
  -- op and funct fields to controller
  op <= instr(31 downto 26);
  funct <= instr(5 downto 0);

  -- datapath
  pcreg: flopenr generic map(32) port map(clk, reset, pcen, pcnext, pc);
  adrmux: mux2 generic map(32) port map(pc, aluout, iord, adr);
  instrreg: flopenr generic map(32) port map(clk, reset, irwrite,
                                             readdata, instr);
  datareg: flopr generic map(32) port map(clk, reset, readdata, data);
  regdstmux: mux2 generic map(5) port map(instr(20 downto 16),
                                          instr(15 downto 11),
                                          regdst, writereg);
  wdmux: mux2 generic map(32) port map(aluout, data, memtoreg, 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);
  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: mux4 generic map(32) port map(writedata,
                   "00000000000000000000000000000100",
                   signimm, signimmsh, alusrcb, srcb);
  alu32: alu port map(srca, srcb, alucontrol, aluresult, zero);
  alureg: flopr generic map(32) port map(clk, reset, aluresult, aluout);
  pcjump <= pc(31 downto 28)&instr(25 downto 0)&"00";
  pcmux: mux3 generic map(32) port map(aluresult, aluout,
                                       pcjump, pcsrc, pcnext);
end;
```

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

MIPS Multicycle Building Blocks

Verilog

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

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

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

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

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

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

VHDL

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

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


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

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


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

architecture behave of mux4 is
begin
  process(s, d0, d1, d2, d3) begin
    case s is
      when "00" =>   y <= d0;
      when "01" =>   y <= d1;
      when "10" =>   y <= d2;
      when "11" =>   y <= d3;
      when others => y <= d0; -- should never happen
    end case;
  end process;
end;
```

7.23

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

MIPS Top-Level Module

Verilog

```
module top(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);
  mem mem(clk, memwrite, adr, writedata,
              readdata);

endmodule
```

VHDL

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

entity top is -- top-level design of MIPS multicycle processor
  port(clk, reset:           in    STD_LOGIC;
       writedata, dataadr:   inout STD_LOGIC_VECTOR(31 downto 0);
       memwrite:             inout STD_LOGIC);
end;

architecture synth of top 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 mem
    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: mem 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;
```

(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
```

VHDL

```
-- 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;
```

(continued on next page)

(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
```

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); --XORI
  lbu      <= controls(0);  --LBU
end;
```

Modified MIPS Multicycle ALU Decoder

Verilog

```
module aludec(input      [5:0] funct,
             input      [2:0] aluop,     // XORI, ORI
          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,  // SRLV, XORI
           input [4:0] shamt, // SRLV
           output reg [31:0] Y, output Zero);

  wire [31:0] S, Bout;

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

  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); --SRLV, XORI
       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, XORI
  S <= A + Bout + F(3);  --SRLV, XORI

  -- 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 <= ("0000000000000000000000000000000" & 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 0xFE0B to memory address 108

#       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 Ox14   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 7.13 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)   // 65035=0xFE0B
          $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;
```

7.24

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

7.25

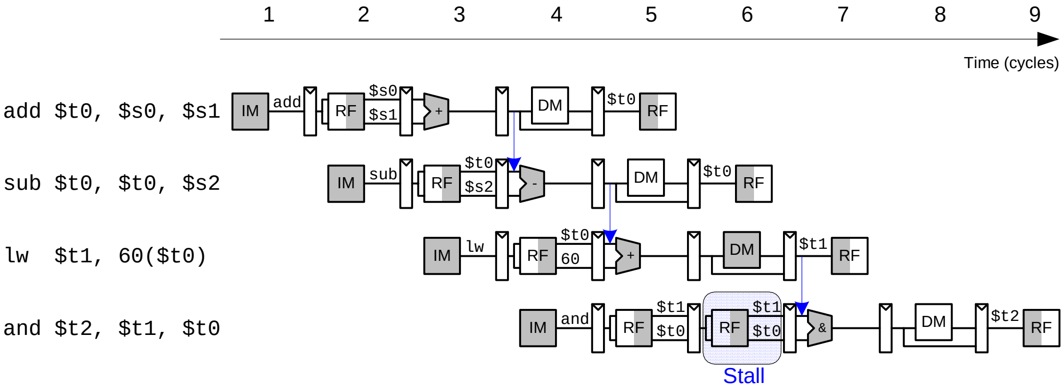


FIGURE 7.14 Abstract pipeline for Exercise 7.25

7.26

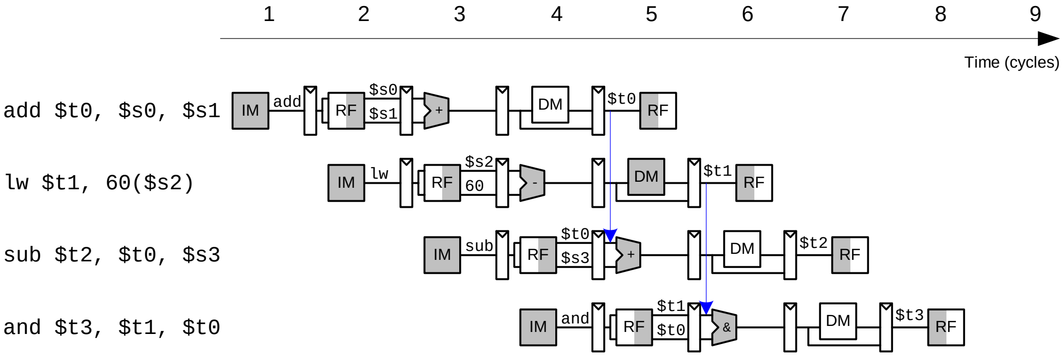


FIGURE 7.15 Abstract pipeline for Exercise 7.26

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.

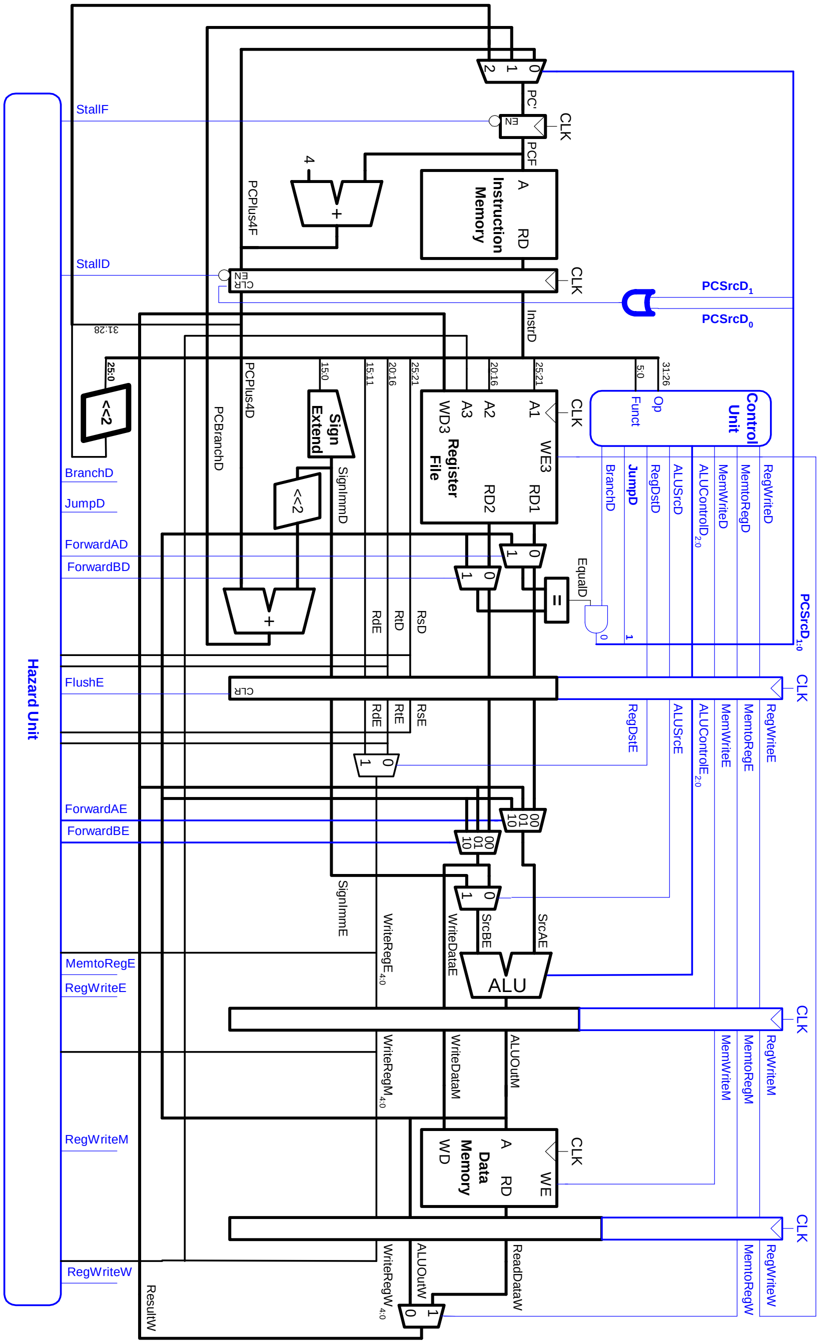
7.28

`addi` requires no additional changes to the datapath, only changes to the control. The main decoder is modified to accommodate `addi` the same as it was for the single-cycle processor (Table 7.4) repeated in Table 7.17 for convenience.

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
`addi`	**001000**	**1**	**0**	**1**	**0**	**0**	**0**	**00**

TABLE 7.17 Main decoder truth table enhanced to support `addi`

7.29


Hazard Unit
StallF
StallD
BranchD
JumpD
ForwardAD
ForwardBD
FlushE
ForwardAE
ForwardBE
MemtoRegE
RegWriteE
RegWriteM
RegWriteW
0
1
2
PC'
CLK
EN
PCF
A
RD
Instruction Memory
4
+
PCPlus4F
CLR
EN
CLK
InstrD
PCSrcD1
PCSrcD0
31:28
25:0
<<2
PCPlus4D
PCBranchD
15:0
Sign Extend
SignImmD
<<2
+
15:11
20:16
25:21
20:16
25:21
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
CLK
5:0
31:26
Op
Funct
Control Unit
RegWriteD
MemtoRegD
MemWriteD
ALUControlD2:0
ALUSrcD
RegDstD
JumpD
BranchD
PCSrcD1:0
EqualD
=
RdE
RtD
RsD
CLR
CLK
RdE
RtE
RsE
RegWriteE
MemtoRegE
MemWriteE
ALUControlE2:0
ALUSrcE
RegDstE
00
01
10
SrcAE
SrcBE
WriteDataE
WriteRegE4:0
SignImmE
ALU
CLK
RegWriteM
MemtoRegM
MemWriteM
ALUOutM
WriteDataM
WriteRegM4:0
A
RD
WD
WE
Data Memory
CLK
RegWriteW
MemtoRegW
ReadDataW
ALUOutW
WriteRegW4:0
ResultW

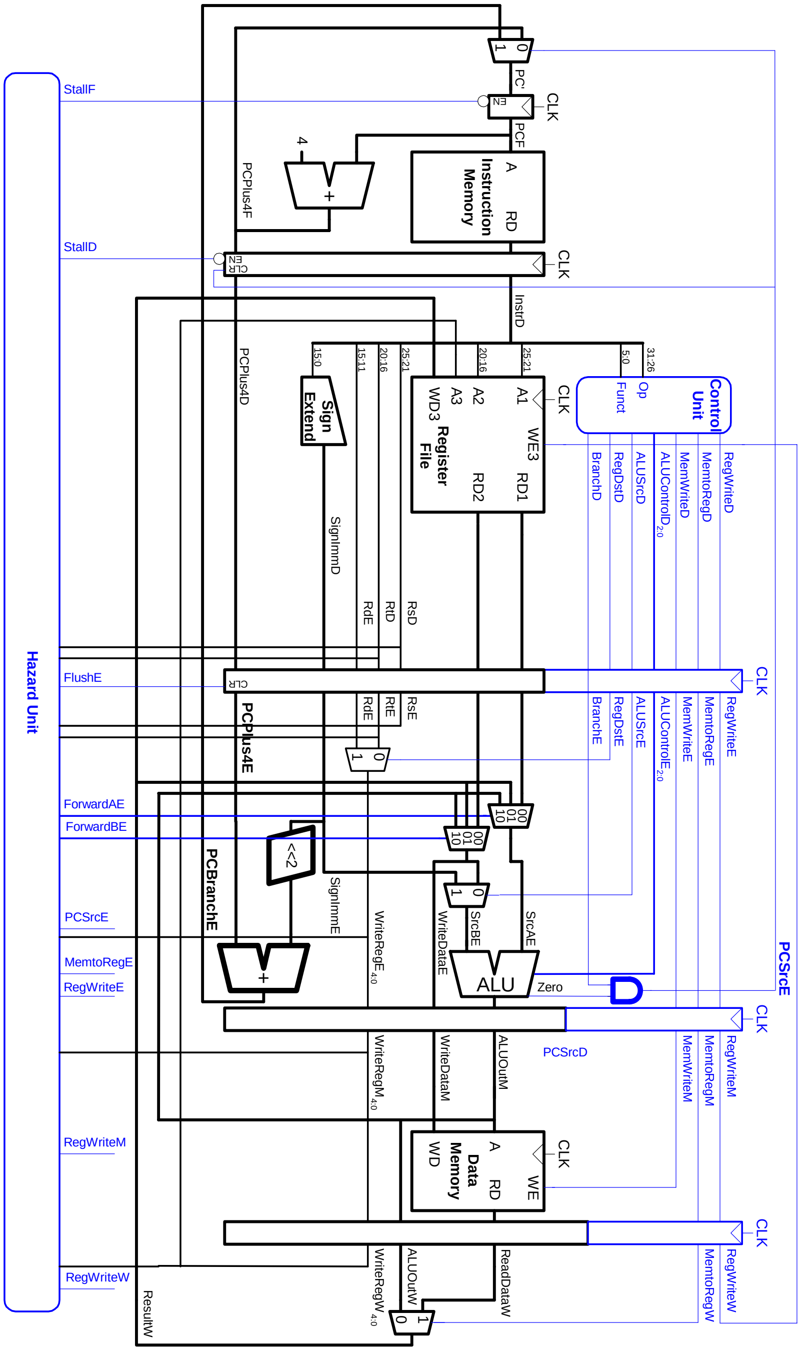
Instruction	opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	**JumpD**
R-type	000000	1	1	0	0	0	0	10	**0**
`lw`	100011	1	0	1	0	0	1	00	**0**
`sw`	101011	0	X	1	0	1	X	00	**0**
`beq`	000100	0	X	0	1	0	X	01	**0**
`j`	**000010**	**0**	**X**	**X**	**0**	**0**	**X**	**XX**	**1**

TABLE 7.18 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
```

7.30


Hazard Unit
StallF
StallD
FlushE
ForwardAE
ForwardBE
PCSrcE
MemtoRegE
RegWriteE
RegWriteM
RegWriteW
CLK
PC'
EN
PCF
Instruction Memory
A
RD
4
+
PCPlus4F
CLR
InstrD
31:26
5:0
25:21
20:16
15:11
15:0
Control Unit
Op
Funct
RegWriteD
MemtoRegD
MemWriteD
ALUControlD$_{2:0}$
ALUSrcD
RegDstD
BranchD
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
Sign Extend
SignImmD
PCPlus4D
RsD
RtD
RdE
RegWriteE
MemtoRegE
MemWriteE
ALUControlE$_{2:0}$
ALUSrcE
RegDstE
BranchE
RsE
RtE
RdE
PCPlus4E
0
1
00
01
10
<<2
PCBranchE
SignImmE
WriteRegE$_{4:0}$
WriteDataE
SrcBE
SrcAE
ALU
Zero
PCSrcE
PCSrcD
RegWriteM
MemtoRegM
MemWriteM
WriteRegM$_{4:0}$
WriteDataM
ALUOutM
Data Memory
WE
WD
RegWriteW
MemtoRegW
WriteRegW$_{4:0}$
ALUOutW
ReadDataW
ResultW

The Hazard Unit must no longer produce the ForwardAD and ForwardBD signals. PCPlus4 must also be passed through the pipeline register to the Execute stage. Also, the FlushE signal requires new hardware, as described by the following Boolean expression:

```
FlushE = lwstall OR PCSrcE
```

CPI calculation:

Loads take 1 clock cycle when there is no dependency and 2 when the processor must stall for a dependency, so they have a CPI of (0.6)(1) + (0.4)(2) = 1.4. Branches take 1 clock cycle when they are predicted properly and 3 when they are not, so they have a CPI of (0.75)(1) + (0.25)(3) = 1.5. Jumps always have a CPI of 2. All other instructions have a CPI of 1. Hence, for this benchmark, Average CPI = (0.25)(1.4) + (0.1)(1) + (0.11)(1.5) + (0.02)(2) + (0.52)(1) = 1.175.

Cycle time calculation:

We modify Equation 7.5 to correspond to our new hardware. Only the Decode and Execute stages change.

$$T_c = max\left(\left.\begin{matrix} t_{pcq} + t_{memread} + t_{setup} \\ 2(t_{RFread} + t_{setup}) \\ t_{pcq} + t_{mux} + t_{mux} + t_{ALU} + t_{AND} + t_{mux} + t_{setup} \\ t_{pcq} + t_{memwrite} + t_{setup} \\ 2(t_{pcq} + t_{mux} + t_{RFwrite}) \end{matrix}\right\} \begin{matrix} \text{Fetch} \\ \text{Decode} \\ \text{Execute} \\ \text{Memory} \\ \text{Writeback} \end{matrix}\right)$$

Thus, the cycle time of the pipelined processor is T_{c3} = max(30 + 250 + 20, 2(150 + 20), 30 + 25 + 25 + 200 + 15 + 25 + 20, 30 + 220 + 20, 2(30 + 25 + 100)) = max(300, 340, 340, 270, 310) = **340 ps**. The Decode and Execute stages tie for the critical path.

According to Equation 7.1, the total execution time is T_3 = (100 x 10^9 instructions)(1.175 cycles / instruction)(340 x 10^{-12} s/cycle) = **39.95 s**.

This compares with the execution time of 63.3 s when the branch prediction was performed in the Decode stage (see Example 7.10).

7.31

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

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

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

The new cycle time is **310 ps**.

7.32

Increasing the ALU delay by 20% (from 200 ps to 240 ps) does not change the critical path (the critical path is the Decode stage in either case, not the Execute stage). Thus, a 20% increase or reduction in delay in the ALU will not affect the cycle time.

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 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;
```

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 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,
instrF, 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,
signimmE);
  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,
aluoutE);
  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 pipelined 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

```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

```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;
```

7.34

MIPS Pipelined Processor Hazard Unit

Verilog

```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

```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;
```

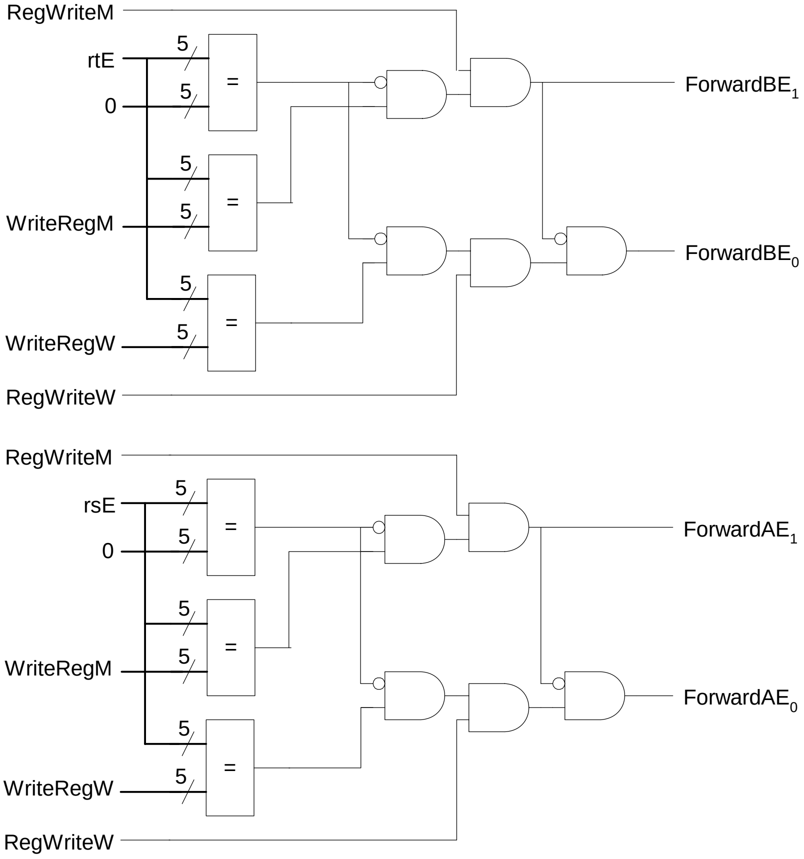


FIGURE 7.16 Hazard unit hardware for forwarding to the Execution stage

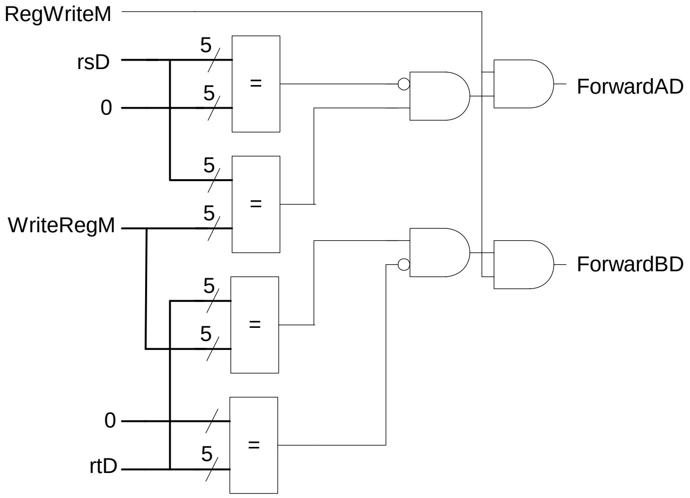


FIGURE 7.17 Hazard unit hardware for forwarding to the Decode stage

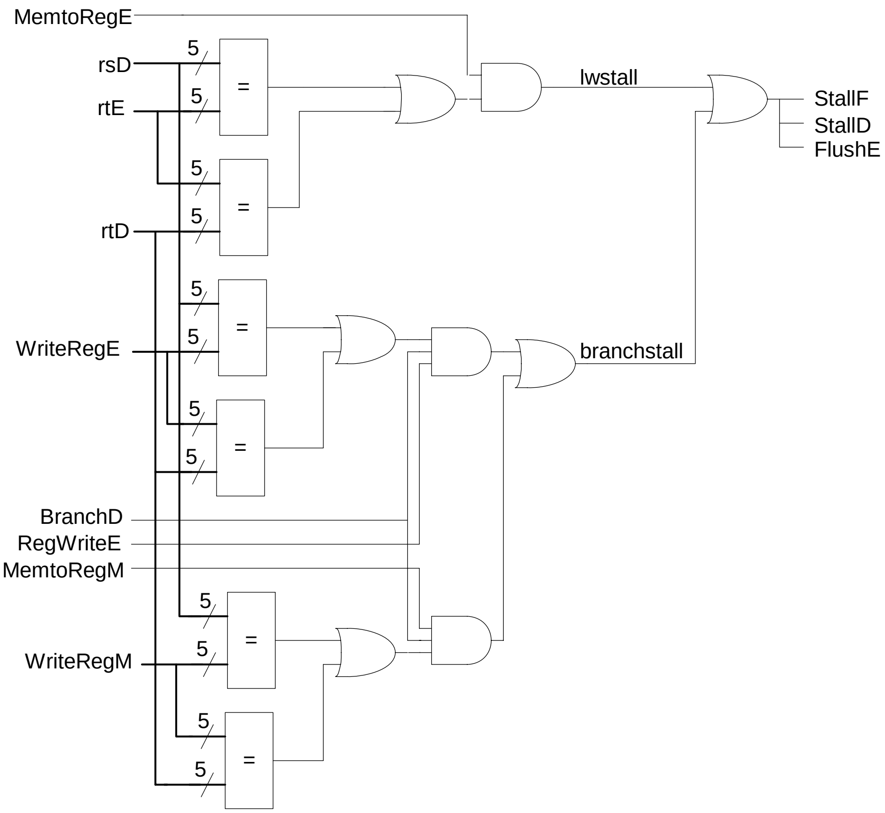
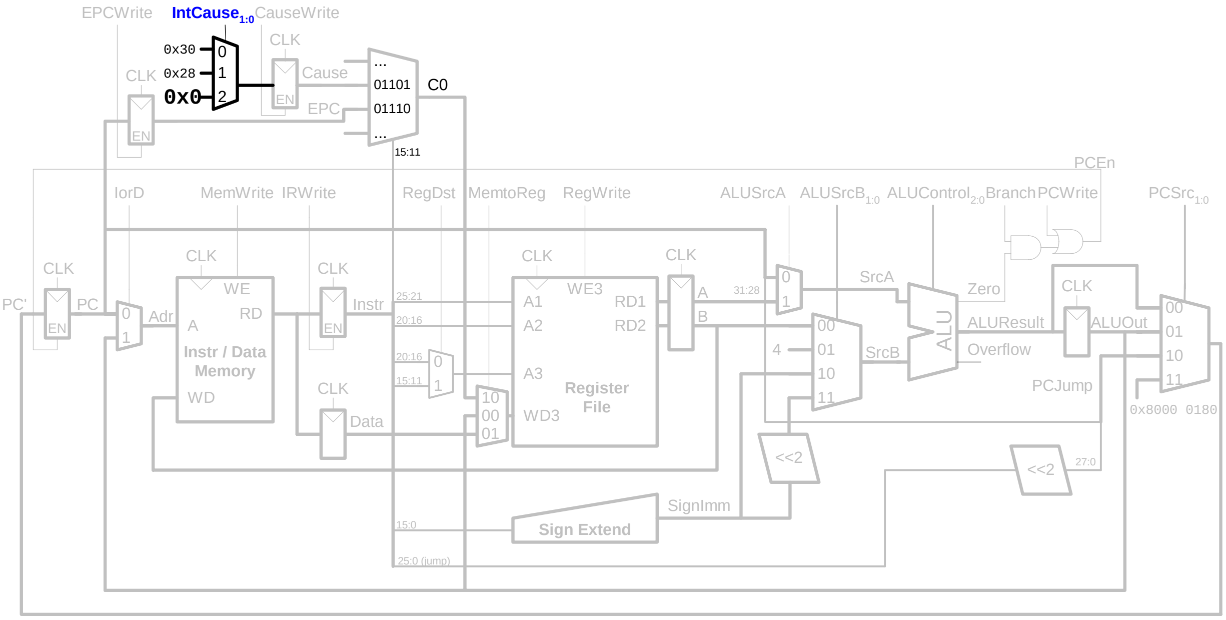
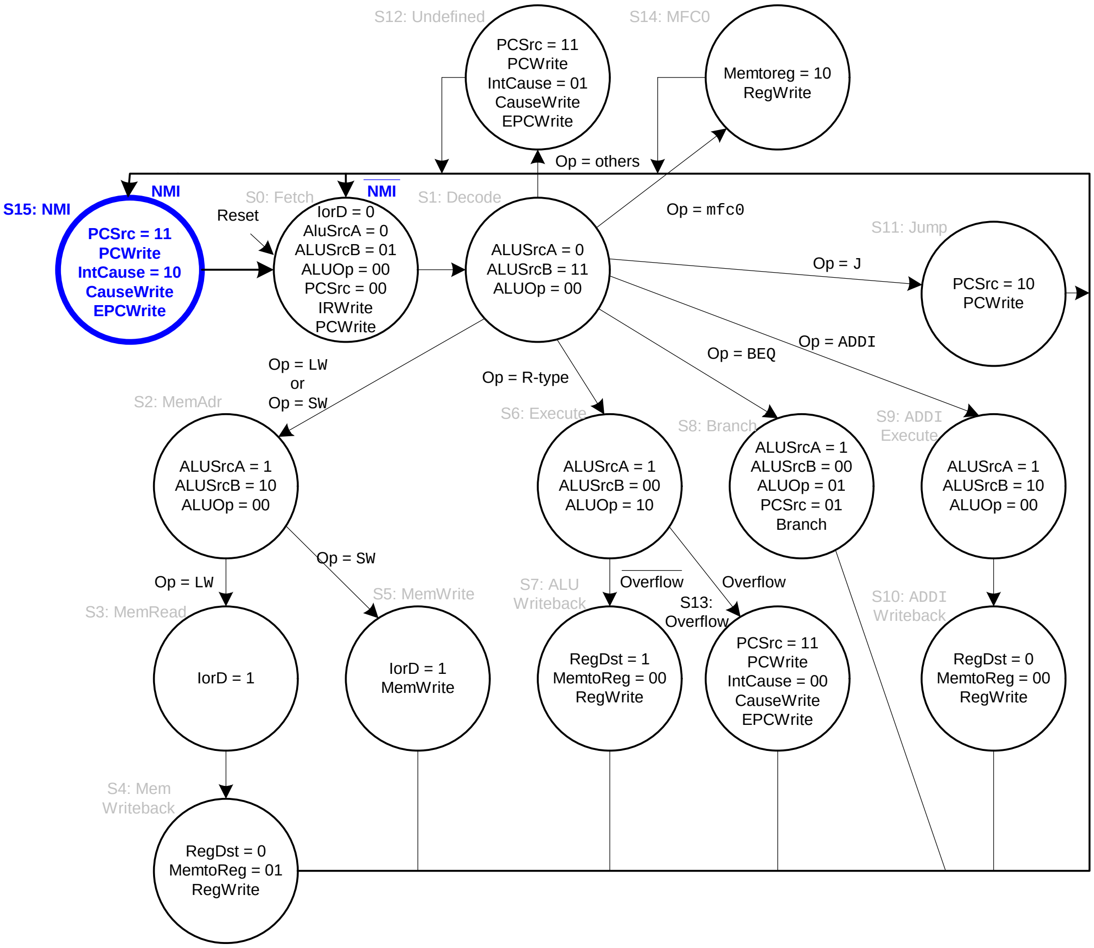


FIGURE 7.18 Hazard unit hardware for stalling/flushing in the Fetch, Decode, and Execute stages

7.35


EPCWrite
IntCause1:0
CauseWrite
0x30
0x28
0x0
0
1
2
CLK
Cause
EN
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
WD3
WE3
RD1
RD2
Register File
31:28
SrcA
SrcB
4
ALU
Zero
ALUResult
Overflow
ALUOut
PCJump
0x8000 0180
00
01
10
11
<<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 7.2

While pipelining offers speedup, it still has its costs. The speedup of an N stage processor is not N because of (1) sequencing overhead (t_{pcq} + t_{setup}, the delay of inserting a register), (2) unequal delays of pipeline stages, (3) time to fill up the pipeline (at the beginning of a program), (4) time to drain the pipeline (at the end of a program), and (5) dependencies stalling or flushing the pipeline.

Question 7.3

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

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

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

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

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

Question 7.4

A superscalar processor duplicates the datapath hardware to execute multiple instructions (in the same stage of a pipelined processor) at once. Ideally, the fetch stage can fetch multiple instructions per clock cycle. However, due to dependencies, this may be impossible. Thus, the costs of implementing a superscalar processor are (1) more hardware (additional register file and memory ports, additional functional units, more hazard detection and forwarding hardware, etc.) and (2) more complex fetch and commit (execution completion) algorithms. Also, because of dependencies, superscalar processors are often underutilized. Thus, for programs with a large amount of dependencies, superscalar processors can consume more area, power and cost (because of the additional hardware) without providing any speedup.

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.

8.2

Spatial locality: One program that exhibits spatial locality is an mp3 player. Suppose a song is stored in a file as a long string of bits. If the computer is playing one part of the song, it will need to fetch the bits immediately adjacent to the ones currently being read (played).

Temporal locality: An application that exhibits temporal locality is a Web browser. If a user recently visited a Web site, the user is likely to peruse that Web site again 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%.

8.4

Repeat data accesses to the following addresses:

0x0 0x40 0x80 0xC0

They all map to set 0 of the direct-mapped cache, but they fit in the fully associative cache. After many repetitions, the miss rate for the fully associative cache approaches 0%. The miss rate for the direct-mapped cache is 100%.

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.

8.6

Usually. Associative caches usually have better miss rates than direct-mapped caches of the same capacity and block size because they have fewer conflict misses. However, pathological cases exist where thrashing can occur, causing the set associative cache to have a worse miss rate.

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.

8.8

(a) $b \times S \times N \times 4$ bytes

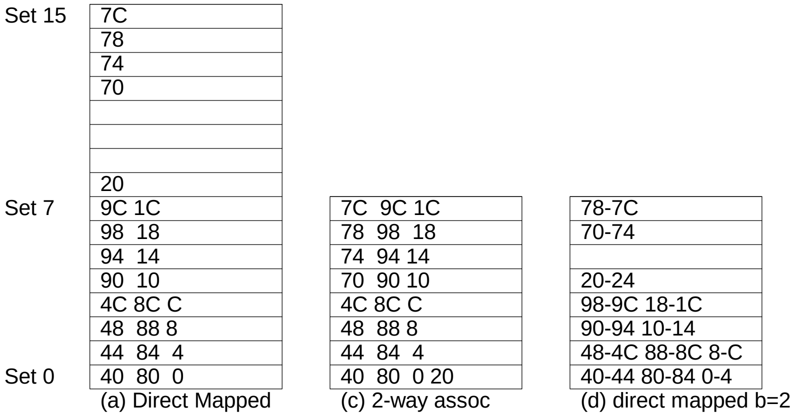
(b) $[A - (\log_2(S) + \log_2(b) + 2)] \times S \times N$

(c) $S = 1$, $N = C/b$

(d) $S = C/b$

8.9

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



Set 15	7C		
	78		
	74		
	70		
	20		
Set 7	9C 1C	7C 9C 1C	78-7C
	98 18	78 98 18	70-74
	94 14	74 94 14	
	90 10	70 90 10	20-24
	4C 8C C	4C 8C C	98-9C 18-1C
	48 88 8	48 88 8	90-94 10-14
	44 84 4	44 84 4	48-4C 88-8C 8-C
Set 0	40 80 0	40 80 0 20	40-44 80-84 0-4
	(a) Direct Mapped	(c) 2-way assoc	(d) direct mapped b=2

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%.

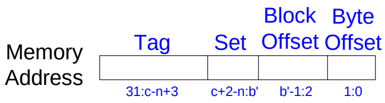
8.10

(a) 128

(b) 100%

(c) ii

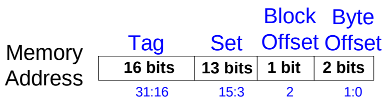
8.11
(a - b)



(c) Each tag is 32 - (c+2-n) bits = (30 - (c-n)) bits

(d) # tag bits × # blocks = (30 - (c-n)) × $2^{c+2 - b'}$

8.12
(a)



(b) Each tag is 16 bits. There are 32Kwords / (2 words / block) = 16K blocks and each block needs a tag: 16 × 16K = 2^{18} = 256 Kbits of tags.

(c) Each cache block requires: 2 status bits, 16 bits of tag, and 64 data bits, thus each set is 2 × 82 bits = **164 bits**.

(d) The design must use enough RAM chips to handle both the total capacity and the number of bits that must be read on each cycle. For the data, the SRAM must provide a capacity of 128 KB and must read 64 bits per cycle (one 32-bit word from each way). Thus the design needs at least 128KB / (8KB/RAM) = 16 RAMs to hold the data and 64 bits / (4 pins/RAM) = 16 RAMs to supply the number of bits. These are equal, so the design needs exactly 16 RAMs for the data.

For the tags, the total capacity is 32 KB, from which 32 bits (two 16-bit tags) must be read each cycle. Therefore, only 4 RAMs are necessary to meet the capacity, but 8 RAMs are needed to supply 32 bits per cycle. Therefore, the design will need 8 RAMs, each of which is being used at half capacity.

With 8Ksets, the status bits require another 8K × 4-bit RAM. We use a 16K × 4-bit RAM, using only half of the entries.

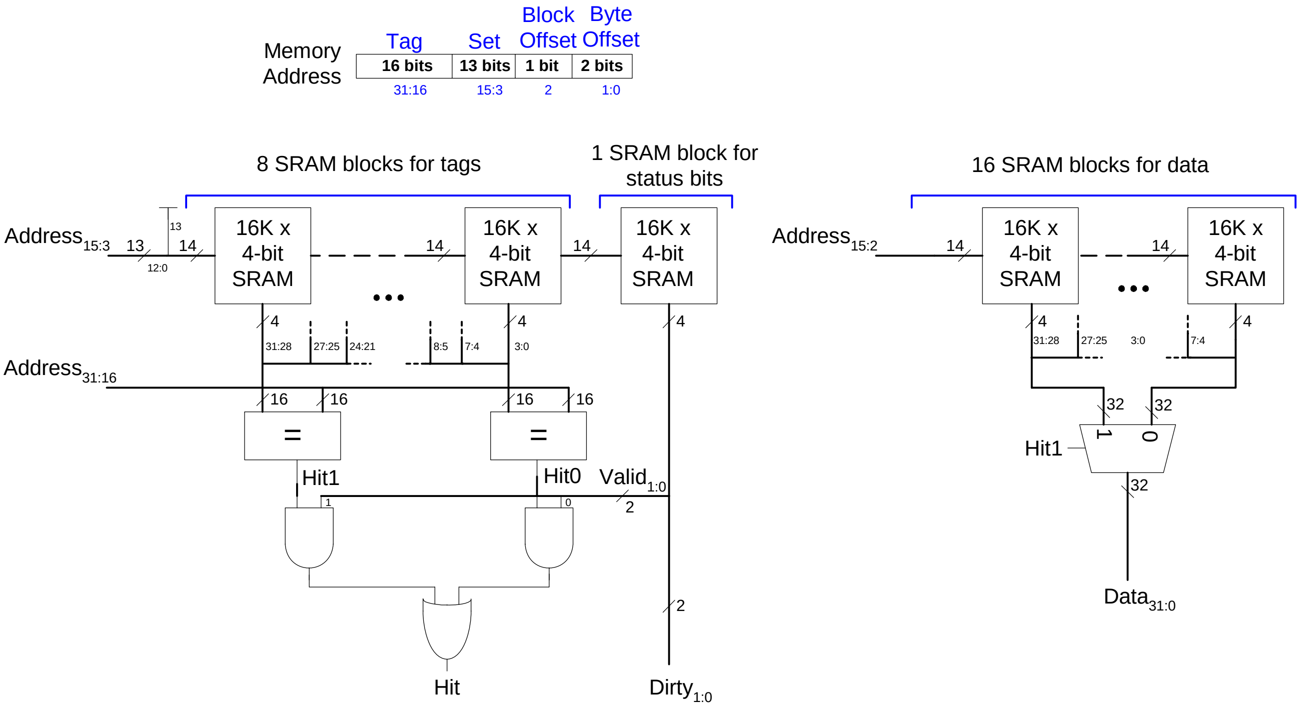


FIGURE 8.2 Cache design for Exercise 8.12

Bits 15:2 of the address select the word within a set and block. Bits 15-3 select the set. Bits 31:16 of the address are matched against the tags to find a hit in one (or none) of the two blocks with each set.

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, $AMAT = t_{\text{cache}} + MR_{\text{cache}}(t_{\text{MM}} + MR_{\text{MM}}\, t_{\text{VM}})$. In this case, there is no virtual memory but there is an L2 cache. Thus,

$AMAT = t_{\text{cache}} + MR_{\text{cache}}(t_{\text{L2cache}} + MR_{\text{L2cache}}\, t_{\text{MM}})$

Where, *MR* is the miss rate. In terms of hit rate, $MR_{\text{cache}} = 1 - HR_{\text{cache}}$, and $MR_{\text{L2cache}} = 1 - HR_{\text{L2cache}}$. 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.

8.14

(a)

FIFO:

FIFO replacement approximates LRU replacement by discarding data that has been in the cache longest (and is thus least likely to be used again). A FIFO cache can be stored as a queue, so the cache need not keep track of the least recently used way in an *N*-way set-associative cache. It simply loads a new cache block into the next way upon a new access. FIFO replacement doesn’t work well when the *least recently used* data is not also the data fetched *longest ago*.

Random:

Random replacement requires less overhead (storage and hardware to update status bits). However, a random replacement policy might randomly evict recently used data. In practice random replacement works quite well.

8.14 (b)

FIFO replacement would work well for an application that accesses a first set of data, then the second set, then the first set again. It then accesses a third set of data and finally goes back to access the second set of data. In this case, FIFO would replace the first set with the third set, but LRU would replace the second set. The LRU replacement would require the cache to pull in the second set of data twice.

8.15

(a)

From Equation 8.2, $AMAT = t_{\text{cache}} + MR_{\text{cache}} (t_{\text{MM}} + MR_{\text{MM}}\, t_{\text{VM}})$. In this case, there is no virtual memory. Thus,

$AMAT = t_{\text{cache}} + MR_{\text{cache}}\, t_{\text{MM}}$

With a cycle time of 1/1 GHz = 1 ns,

$\boldsymbol{AMAT} \textbf{ = 1 ns + 0.05(60 ns) = 4 ns}$

(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**

8.16

2^{64} bytes = 2^4 exabytes = **16 exabytes**.

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.

8.18

(a) **23 bits**

(b) $2^{32}/2^{12}$ = $\mathbf{2^{20}}$ **virtual pages**

(c) 8 MB / 4 KB = $2^{23}/2^{12}$ = $\mathbf{2^{11}}$ **physical pages**

(d) virtual page number: **20 bits**; physical page number = **11 bits**

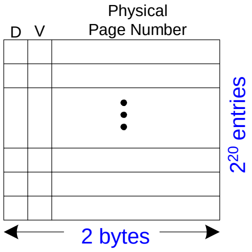
(e) # virtual pages / # physical pages = $\mathbf{2^9}$ virtual pages mapped to each physical page.

Imagine a program around memory address 0x01000000 operating on data around address 0x00000000. Physical page 0 would constantly be swapped between these two virtual pages, causing severe thrashing.

(f) $\mathbf{2^{20}}$ page table entries (one for each virtual page).

(g) Each entry uses 11 bits of physical page number and 2 bits of status information. Thus, **2 bytes** are needed for each entry (rounding 13 bits up to the nearest number of bytes).

(h) The total table size is $\mathbf{2^{21}}$ **bytes**.



8.19

(a) From Equation 8.2, $AMAT = t_{\text{cache}} + MR_{\text{cache}}\,(t_{MM} + MR_{MM}\,t_{VM})$. However, each data access now requires an address translation (page table or TLB lookup). Thus,

Without the TLB:

$AMAT = t_{MM} + [t_{\text{cache}} + MR_{\text{cache}}\,(t_{MM} + MR_{MM}\,t_{VM})]$

$AMAT$ = 100 + [1 + 0.02(100 + 0.000003(1,000,000))] cycles = **103.06 cycles**

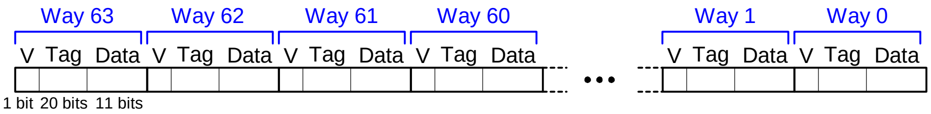
With the TLB:

$AMAT = [t_{TLB} + MR_{TLB}(t_{MM})] + [t_{\text{cache}} + MR_{\text{cache}}\,(t_{MM} + MR_{MM}\,t_{VM})]$

$AMAT$ = [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**

8.20

(a)

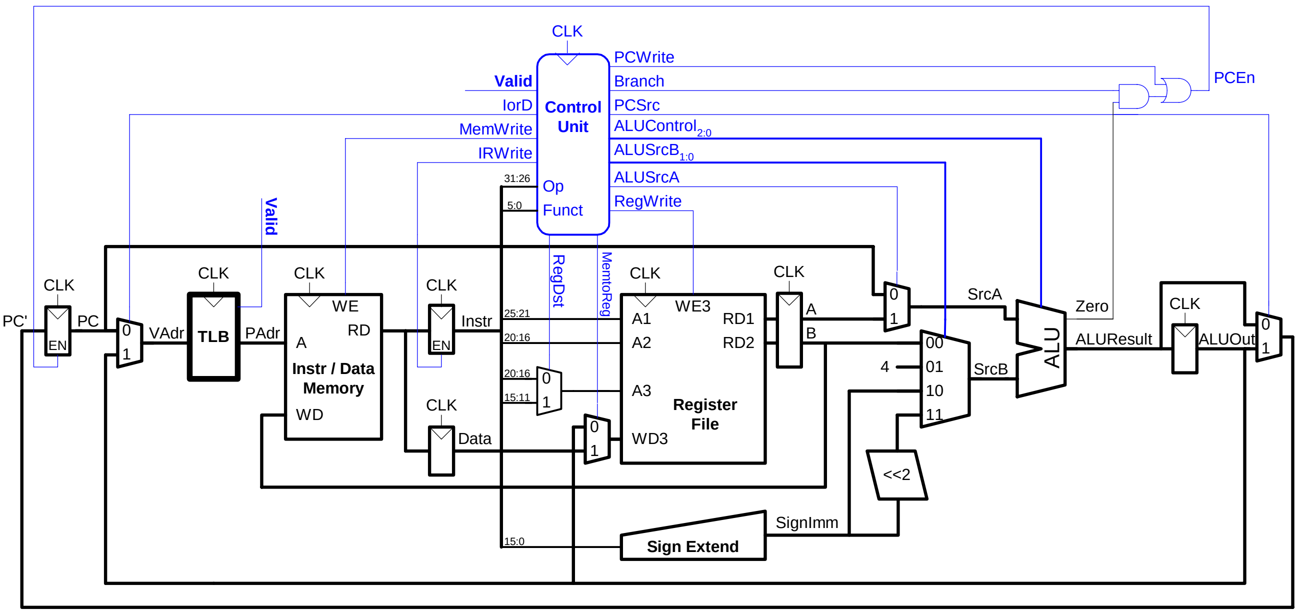


FIGURE 8.3 Multicycle MIPS processor with TLB

(b) Each instruction and data access now takes at least one additional clock cycle. On each access, the virtual address (*VAdr* in Figure 8.3) needs to be translated to a physical address (*PAdr*). Upon a TLB miss, the page table in main memory must be accessed.

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**

8.21 (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.

8.22

An application that accesses large amounts of data might be written to localize data accesses to a small number of virtual pages. Particularly, data accesses can be localized to the number of pages that fit in physical memory. If the virtual memory has a TLB that has fewer entries than the number of physical pages, accesses could be localized to the number of entries in the TLB, to avoid the need of accessing the page table to perform address translation.

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.

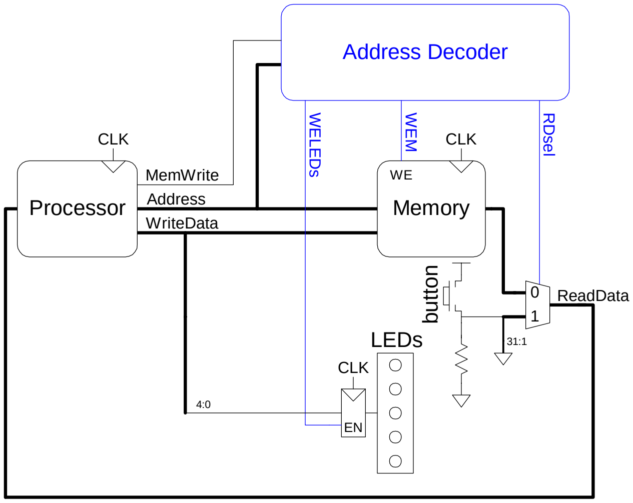
(c) The amount of physical memory affects how many physical pages can be accessed at once. With a small main memory, if many applications run at once or a single application accesses addresses from many different pages, thrashing can occur. This can make the applications dreadfully slow.

8.24

(a)

```
#  MIPS code for Exercise 8.24
#  The LEDs are mapped to the 5 least significant bits at
#  memory address 0xFFFFFF14

       addi $t1, $0, 0x15    # the random pattern to display to the LEDs
Loop0: lw   $t0, 0xFF10($0)  # $t0 = button value
       beq  $t0, $0, Loop0   # if button is not pressed, keep checking
       sw   $t1, 0xFF14($0)  # if $t0 == 1, display pattern on LEDs
Loop1: lw   $t0, 0xFF10($0)  # $t0 = button value
       bne  $t0, $0, Loop1   # if button is still pressed, loop
       sw   $0, 0xFF14($0)   # clear LEDs when button is not pressed
       j    Loop0            # check when button is pressed
```

(b)


Address Decoder
CLK
Processor
MemWrite
Address
WriteData
WELEDs
WEM
RDsel
CLK
WE
Memory
button
0
1
ReadData
31:1
LEDs
CLK
EN
4:0

8.24 (c) Address Decoder for Exercise 8.24

Verilog

```
module addrdec(input [31:0] addr, input memwrite,
               output reg   WELEDs, Mwrite,
               output reg   rdselect);

  parameter  B    = 16'hFF10;    // push button
  parameter  LEDs = 16'hFF14;    // LEDs

  wire [15:0] addressbits;

  assign addressbits = addr[15:0];

  always @ ( * )
    if (addr[31:16] == 16'hFFFF) begin
      // writedata control
      if (memwrite)
        if (addressbits == LEDs)
          {WELEDs, Mwrite, rdselect} = 3'b100;
        else
          {WELEDs, Mwrite, rdselect} = 3'b010;

      // readdata control
      else
        if ( addressbits == B )
          {WELEDs, Mwrite, rdselect} = 3'b001;
        else
          {WELEDs, Mwrite, rdselect} = 3'b000;
    end
    else
      {WELEDs, Mwrite, rdselect} =
        {1'b0, memwrite, 1'b0};

endmodule
```

VHDL

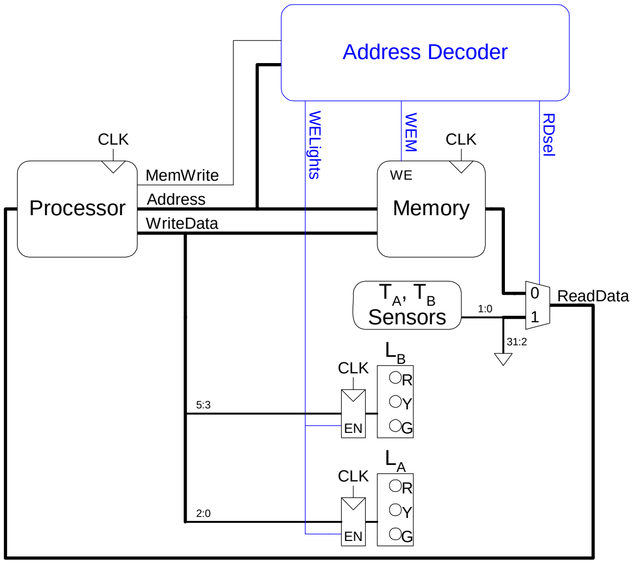
```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity addrdec is -- address decoder
  port(addr:                       in STD_LOGIC_VECTOR(31 downto 0);
       memwrite:                   in STD_LOGIC;
       WELEDs, Mwrite, rdselect: out STD_LOGIC);
end;

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"FF14") then     -- LEDs
           WELEDs <= '1'; Mwrite <= '0'; rdselect <= '0';
         else
           WELEDs <= '0'; Mwrite <= '1'; rdselect <= '0';
         end if;

       -- readdata control
       else
         if (addr(15 downto 0) = X"FF10" ) then  -- pushbutton
           WELEDs <= '0'; Mwrite <= '0'; rdselect <= '1';
         else
           WELEDs <= '0'; Mwrite <= '0'; rdselect <= '0';
         end if;
       end if;

     -- not a memory-mapped address
     else
       WELEDs <= '0'; Mwrite <= memwrite; rdselect <= '0';
     end if;
  end process;
end;
```

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 = TA
       bne  $t4, $0,  S0     # if TA == 1, loop back to S0

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

       sw   $t0, 0xF004($0)  # lights = green / red
S2:    lw   $t4, 0xF000($0)  # $t4 = sensor values
       andi $t4, $t4, 0x1    # $t4 = TB
       bne  $t4, $0,  S2     # if TB == 1, loop back to S2

S3:    sw   $t1, 0xF004($0)  # lights = yellow / red
       j    Start
```

8.25 (b)


Address Decoder
WELights
WEM
RDsel
CLK
Processor
MemWrite
Address
WriteData
CLK
WE
Memory
T_A, T_B Sensors
1:0
31:2
0
1
ReadData
L_B
CLK
EN
R
Y
G
5:3
L_A
CLK
EN
R
Y
G
2:0

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* direct-mapped caches. The last set has *N*+1 blocks that map to it. This access pattern then repeats.

In the direct-mapped cache, the accesses to the same set conflict, causing a 100% miss rate. But in the set-associative cache all accesses (except the last one) don't conflict. Because the number of block accesses in the repeated pattern is one more than the number of blocks in the cache, the fully associative cache also has a 100% miss rate.

A **fully associative cache** performs better than the other two when the direct-mapped and set-associative accesses conflict and the fully associative accesses don't. Thus, the repeated pattern must access at most *B* blocks that map to conflicting sets in the direct and set-associative caches.

Question 8.2

Virtual memory systems use a hard disk to provide an illusion of more capacity than actually exists in the main (physical) memory. The main memory can be viewed as a cache for the most commonly used pages from the hard disk. Pages in virtual memory may or may not be resident in physical memory. The processor detects which pages are in virtual memory by reading the page table, that tells where a page is resident in physical memory or that it is resident on the hard disk only. The page table is usually so large that it is resident in physical memory. Thus, each data access requires potentially two main memory accesses instead of one. A translation lookaside buffer (TLB) holds a subset of the most recently accessed TLB entries to speedup the translation from virtual to physical addresses.

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 8.4

If the virtual page size is large, a single cache miss could have a large miss penalty. However, if the application has a large amount of spatial locality, that page will likely be accessed again, thus amortizing the penalty over many accesses. On the other hand, if the virtual page size is small, cache accesses might require frequent accesses to the hard disk.

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.